

Sacrificing Suspended Processes

 optiv.com/insights/source-zero/blog/sacrificing-suspended-processes

September 26, 2022

EDR hooking has become a major part of an adversary's ability to successfully compromise an endpoint system. Over the past couple of years, there have been numerous pieces of research on this topic, including my own, which are linked below if you would like to read more on the subject:

If you are unfamiliar, hooking is a technique to alter the behavior of an application, allowing EDR tools to monitor the execution flow that occurs in a process, gather information for behavior-based analytics and detect suspicious and malicious activity. This allows for more accurate detection rates of post-initial compromise techniques (e.g., code execution) as well as post-exploitation techniques (e.g., privilege escalation, lateral movement, or ransomware activity).

Before we dive into EDR unhooking any further, we need to understand why. Systemically, the reason why EDR unhooking is such a prevalent technique is that most products utilize userland hooks to detect and prevent malicious behavior. Userland provides the widest range of functions a security product can hook to see and provides a tremendous amount of detailed telemetry for products to make determinations on the behavior going on within the process. The downside to userland hooking is that everything in the userland space has the same permissions. This means that an adversary's code has the same read and write permissions as the security product's hooks. If an adversary knows where a hook exists, they can tamper with or overwrite it, defeating the product.

This architecture has posed a challenge for EDRs as moving hooking to another area of a process, such as in the Kernel, can be quite difficult, involving a rewrite of the entire product and introducing potential stability issues. Some products over the past few years have adapted by trying to monitor known ways adversaries tamper or overwrite hooks before these actions occur. This article will walk through the discovery of another method, how it works, why it works, and what can be done to stop it. While there has been a lot of focus on EDR hooks and circumventing them, this is not intended to be a negative narrative towards EDRs. The focus on all this research is to help highlight weaknesses that adversaries can exploit and by highlighting these issues, the hope is to start a discussion of how these products can improve.

Memory Mapping

Inside every Windows based process, multiple DLLs are loaded into the process's memory to provide the functionality that allows the application to interact with the operating system. A DLL is library that contains code and data that can be used by more than one program at the same time. When loaded into memory, the DLLs occupy a space in that process's memory. As these DLLs are required for the process to run properly, they are mapped to a section of that process's memory so that the application can reference the functions stored dynamically. Every process is different but at minimum, every process needs Kernel32.dll, Kernelbase.dll, and Ntdll.dll to operate. This is because these DLLs contain the low-level instructions and API calls needed for the process to interact with the operating system. Where things get interesting is when we compare two processes on the same system, we can see the same system DLLs loaded at the same base address for both processes.

Image

Name	Base address	Size	Description
advapi32.dll	0x7ffb590d0000	696 kB	Advanced
cmd.exe	0x7ff700970000	412 kB	Windows
cmd.exe.mui	0x2317b900000	132 kB	Windows C
combase.dll	0x7ffb5a610000	3.33 MB	Microsoft C
ctiuser.dll	0x7ffb422c0000	2.73 MB	Carbon Bla
fitLib.dll	0x7ffb4fec0000	44 kB	Filter Librar
kernel32.dll	0x7ffb5ac40000	760 kB	Windows N
KernelBase.dll	0x7ffb58790000	2.78 MB	Windows N
locale.nls	0x2317b6e0000	804 kB	
ntdll.dll	0x7ffb5b070000	1.96 MB	NT Layer C
rpport4.dll	0x7ffb59180000	1.14 MB	Remote Pri
sechost.dll	0x7ffb5a210000	624 kB	Host for SK

Name	Base address	Size	Description
ncryptssp.dll	0x7ffb46010000	152 kB	Microsc
negoexts.dll	0x7ffb58110000	152 kB	NegoEx
netlogon.dll	0x7ffb57d60000	880 kB	Net Log
netlogon.dll.mui	0x23f290b0000	16 kB	Net Log
netprovfw.dll	0x7ffb58170000	88 kB	Provisio
netutils.dll	0x7ffb57cf0000	48 kB	Net Wir
ngcpcopkeysrv.dll	0x7ffb3c710000	264 kB	Microsc
normidna.nls	0x23f290f0000	80 kB	
nsi.dll	0x7ffb5ad00000	32 kB	NSI Usr
ntdll.dll	0x7ffb5b070000	1.96 MB	NT Lay
ntosapi.dll	0x7ffb50e30000	168 kB	Active T
NtLmShared.dll	0x7ffb57e40000	76 kB	NTLM S

Figure 1: Process Modules

Further investigation shows that the modules of each system DLLs are mapped to the same address points, which is odd given Address Space Layout Randomization (ASLR) has been deployed on all modern systems.

Image

Base address	act...	Use
0x7ffb5b070000		C:\Windows\System32\ntdll.dll
0x7ffb5b071000		C:\Windows\System32\ntdll.dll
0x7ffb5b18c000		C:\Windows\System32\ntdll.dll
0x7ffb5b1d4000		C:\Windows\System32\ntdll.dll
0x7ffb5b1d5000		C:\Windows\System32\ntdll.dll
0x7ffb5b1d7000		C:\Windows\System32\ntdll.dll
0x7ffb5b1e0000		C:\Windows\System32\ntdll.dll

Base address	act...	Use
0x7ffb5b070000		C:\Windows\System32\ntdll.dll
0x7ffb5b071000		C:\Windows\System32\ntdll.dll
0x7ffb5b18c000		C:\Windows\System32\ntdll.dll
0x7ffb5b1d4000		C:\Windows\System32\ntdll.dll
0x7ffb5b1d5000		C:\Windows\System32\ntdll.dll
0x7ffb5b1d7000		C:\Windows\System32\ntdll.dll
0x7ffb5b1e0000		C:\Windows\System32\ntdll.dll

Figure 2: Process Modules - DLL Mappings

Address Space Layout Randomization

ASLR is a security mechanism Microsoft implemented in 2007 (with Windows Vista) to prevent stack memory corruption-based vulnerabilities. ASLR randomizes the address space inside of a process, to ensure that all memory-mapped objects, the stack, the heap, and the executable program itself are unique. Now, this is where it gets interesting because while ASLR works, it does not for position-independent code such as DLLs. What happens with DLLs, (specifically Known System DLLs) is that the address space is randomized once, at boot time.

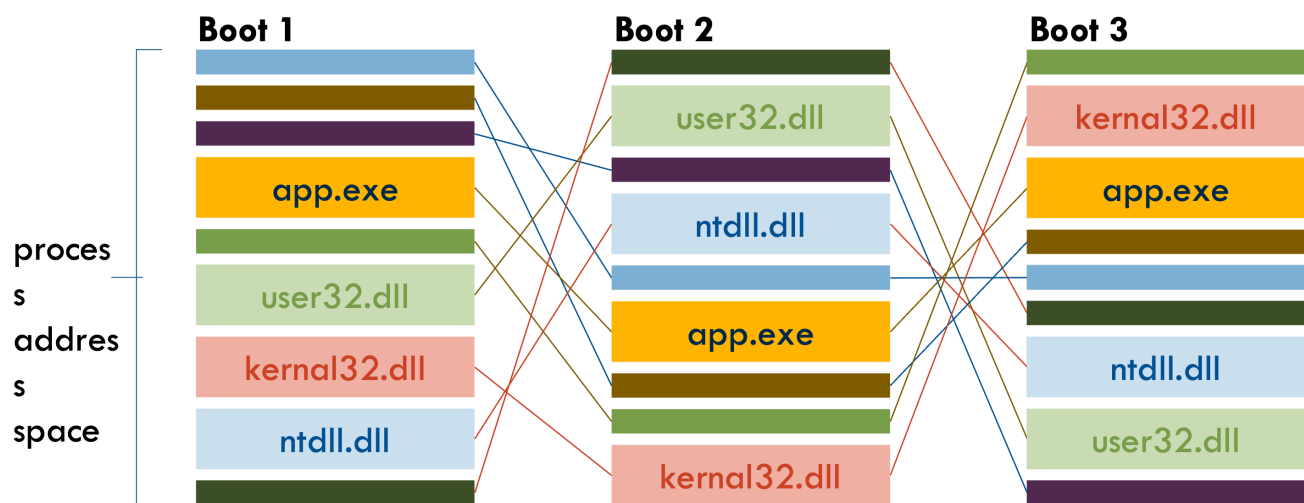


Figure 3: How DLL ASLR Works Each Boot

Process Creation

So, if we look at a process on an endpoint that contains an EDR we can see that interesting thing. The first thing we note is that system DLLs are naturally loading as part of the startup process. We can see the important DLLs are loaded along with the EDR's DLL that is used to hook into the process.

Image

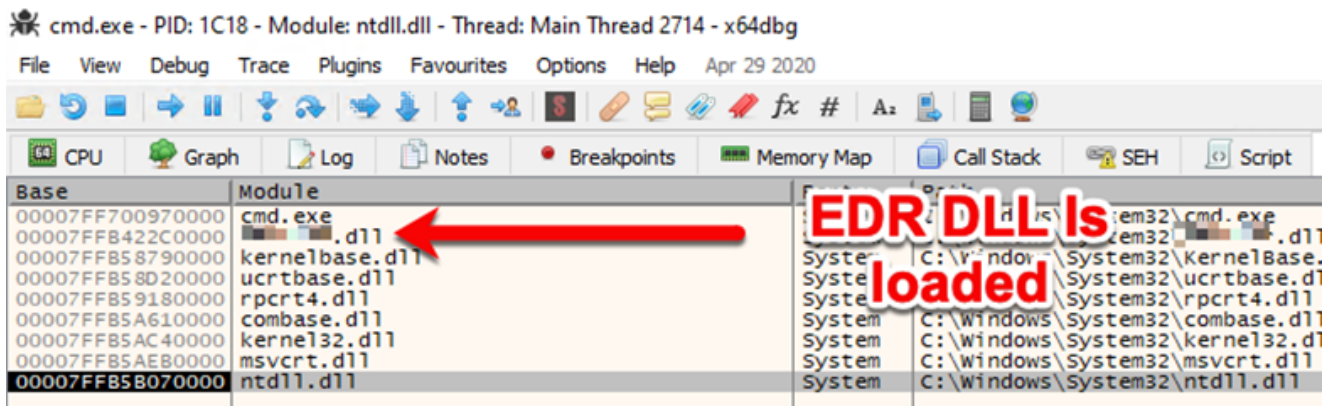


Figure 4: EDR DLL Loaded

In looking at Windows syscalls in Ntdll.dll, we can see that nothing is hooked yet. We can see this because the proper assembly is there (e.g., "mov, rcx and mov eax and the syscall number") and there are no jmp instructions here redirecting the flow of execution to an EDR's DLL. Once the process is resumed, the EDR kicks in and hooks the syscalls.

Image

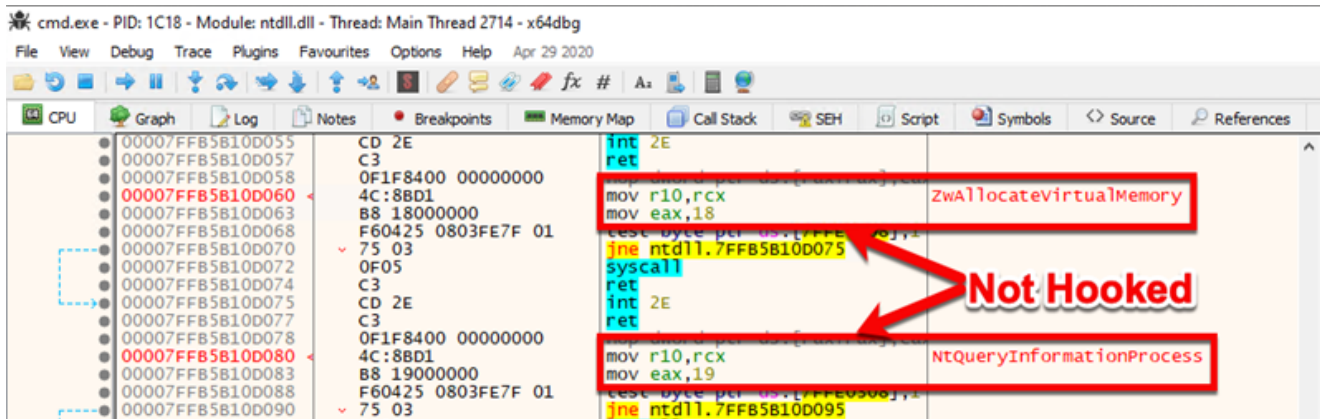


Figure 5: Process in Suspended State Not Hooked Yet

Image

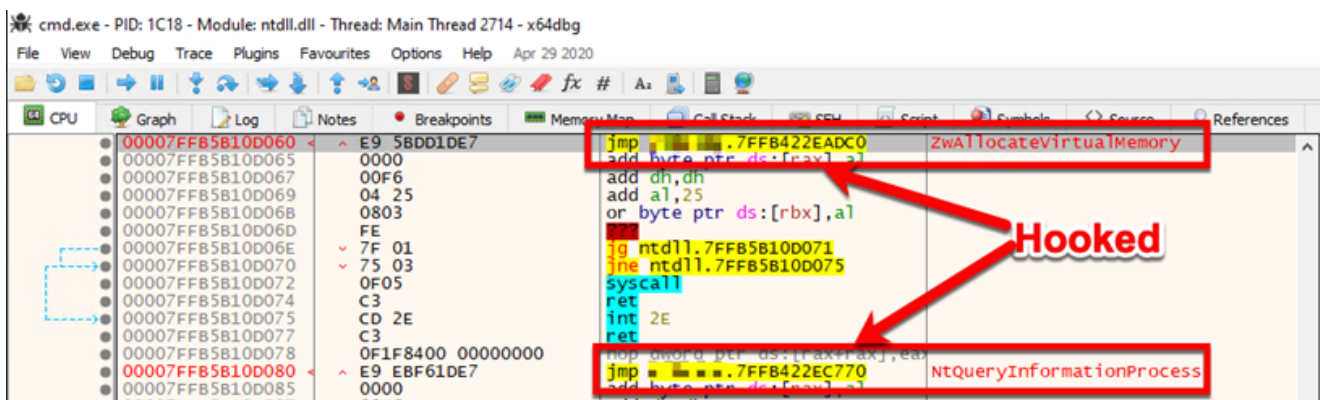


Figure 6: Process Resumed and then Hooked

This means that there is a bit of a delay before an EDR begins hooking and modifying the assembly of system DLLs. This delay is not long and the only way to see this is by setting a break point and moving from one instruction to another utilizing a debugger and is not feasible programmatically. However, if we create a process in a suspend state, we can see that hardly any DLLs are loaded, except for Ntdll.dll. You can also see no EDR's DLLs are loaded, meaning the syscalls located in Ntdll.dll are unmodified.

Image

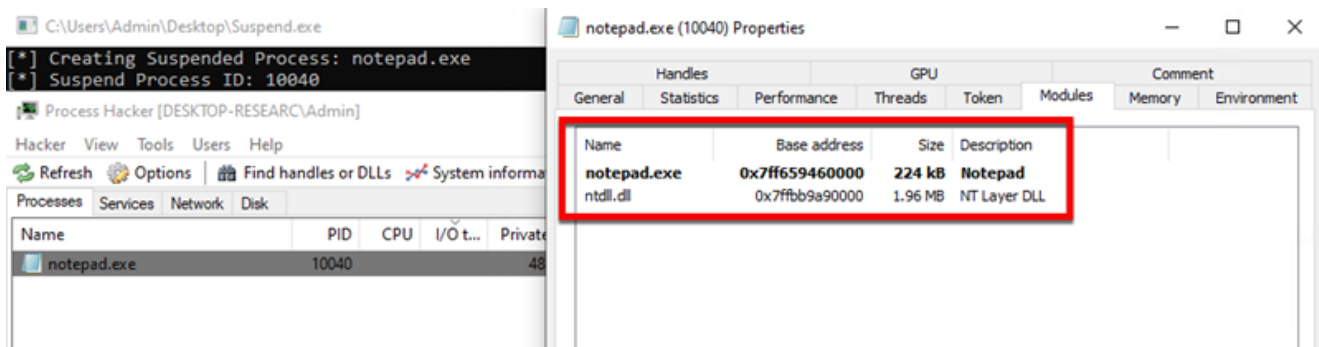


Figure 7: Notepad.exe Started in a Suspended State

We know the following:

1. All DLLs must have the same location in memory
2. Newly created suspended processes are clean

The question becomes, how to extract the clean syscalls from this suspended process? The answer is simple: ReadProcessMemory.

ReadProcessMemory reads a process's memory, now this API call is commonly associated with the reading of LSASS as part of any credential-based attack, however on its own it is inherently not malicious, especially if we are just reading an arbitrary section of memory. The only time ReadProcessMemory will be flagged as part of something suspicious is if you are reading something you shouldn't, like the contents of LSASS. EDR products should never flag the fact that ReadProcessMemory was called, as there are legitimate operational uses for this function and would result in many false positives. Instead, they look for other indicators such as:

- What is being read? Is what is being read known to be sensitive?
- Does the process requesting the read have the proper access?
- Was anything else loaded before this call that would be suspicious?

The EDR typically has hooks in both the process reading data and the process where data is being read from, allowing these and other indicators to be gathered and correlated. However, our suspend process has no EDR in it so that creates a gap in the telemetry. Next, since we are reading a section of a system DLL that is found in every process on every Windows system, the data itself is not sensitive. The only thing that could potentially be an indicator is mapping the remote process's address space, as that then potentially feeds the security product monitoring our process with a way of identifying that address space. We can take this a step further by only focusing on a section of Ntdll.dll, its .text section, where all syscalls are stored. So rather than reading the entire DLL we only read a subset of it.

To do this, we need to dynamically get the address of Ntdll.dll. Since the address of every DLL is the same place per boot, we can pull this information from our own process. The most effective way to do this is to find the base address of Ntdll.dll and the offset of .text. This will provide the starting address of where Ntdll.dll is mapped in memory. We can do this using the LoadLibrary function to create a pointer to Ntdll.dll in memory. This pointer will resolve to the starting address of Ntdll.dll in memory. This is validated by monitoring the function calls using API Monitor.

Image

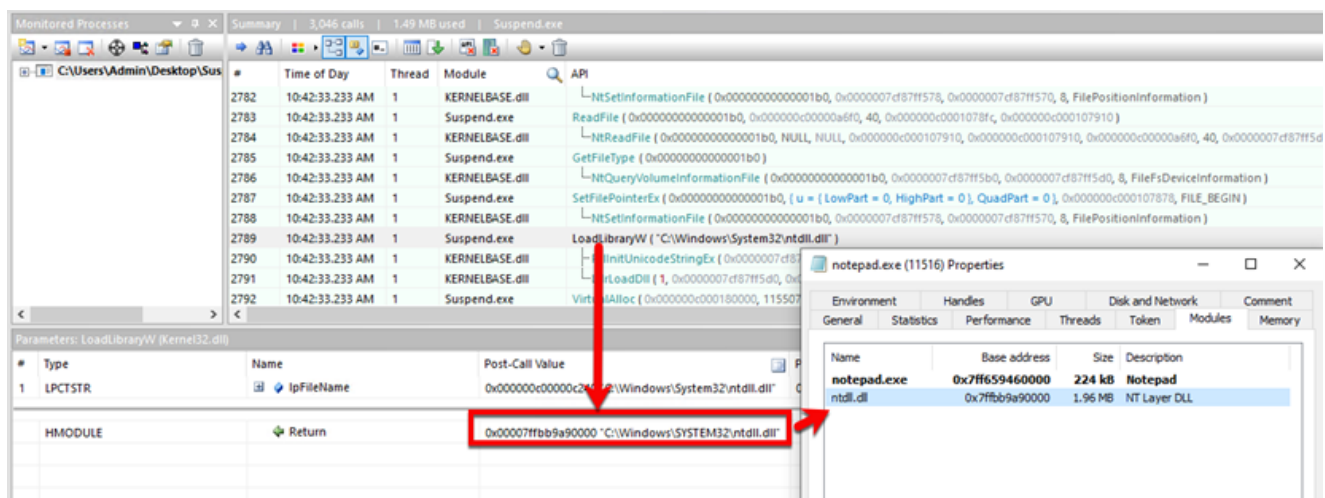


Figure 8: API Monitor - LoadLibrary Call

The next step is to figure out where Ntdll's .text section starts. We can do this a couple of ways, the first is by hardcoding the offset address. An offset denotes the exact number of bytes from the base address where they reside, given the function's location on the stack, and every system DLL has the same offset value for their .text section which is "0x1000" and is the first section. The only difference is the size which can be seen below:

Image

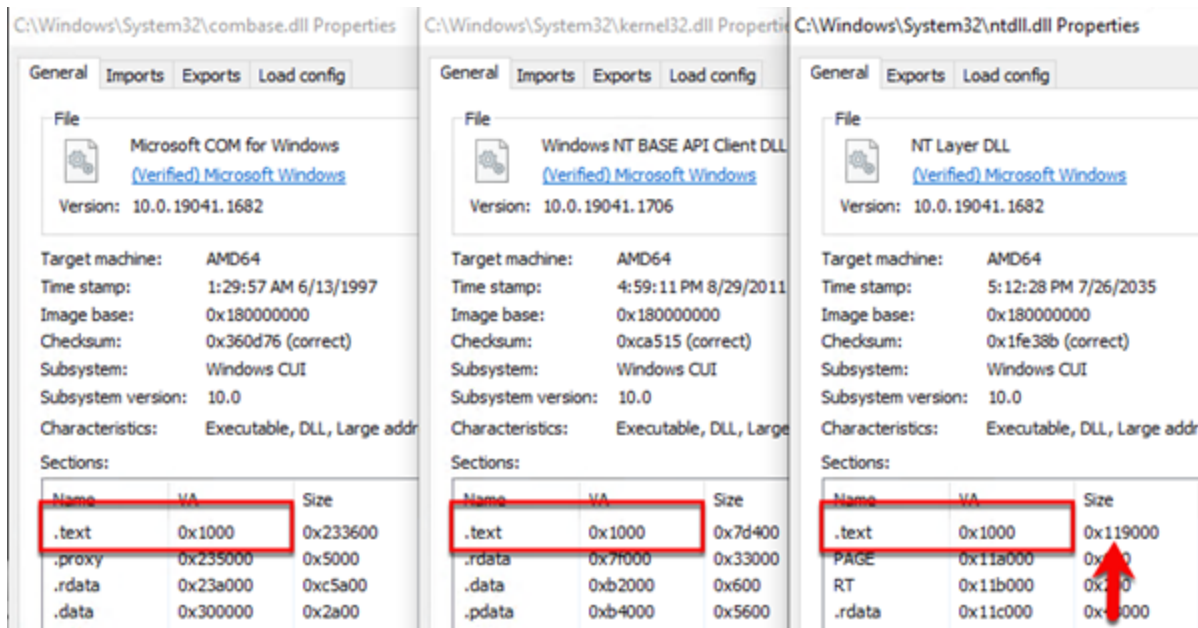


Figure 9: DLL Properties

These can all be dynamically enumerated as the size of the .text field will be different depending on the version of Windows in use.

Weaponizing This

We can use the WriteProcessMemory function to overwrite sections of memory that are not writable without calling any of the memory change API functions. When WriteProcessMemory is called to write to a section of memory that is not set up to be writable, it temporarily modifies the permissions of the region of memory to writable (that is, if you have sufficient privileges, which we do, since we own the process). It writes the value and restores the original permissions without calling the VirtualProtect function; instead, it automatically calls the associated Syscall (NtProtectVirtualMemory). Microsoft implemented it this way to make debuggers more stable. If debuggers want to modify memory on the fly, they can simply modify a section without having to perform multiple tasks. (For more information about this feature, please reference [Microsoft's article](#).)

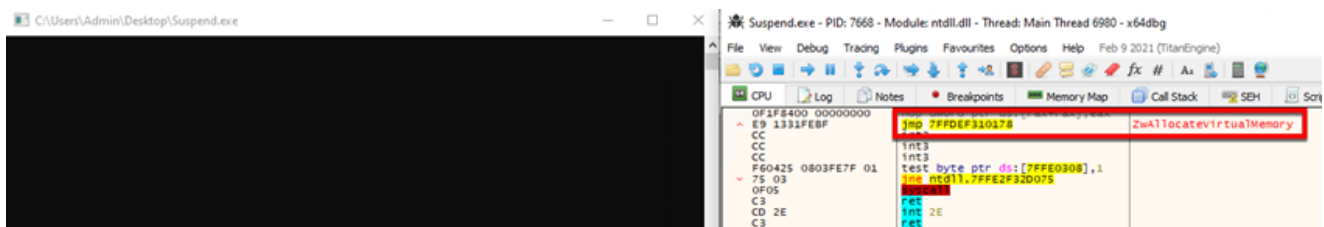


Figure 10: Before – AllocateVirtualMemory is Hooked

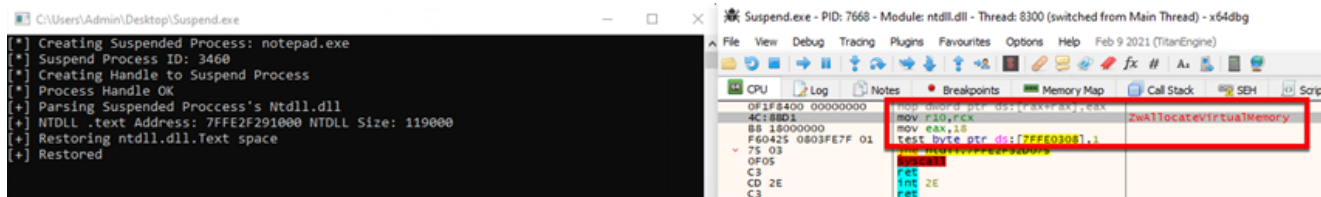


Figure 11: After – AllocateVirtualMemory is Not Hooked

Defending Against

Defense against this technique will be dependent on either a prevention/zero-trust mindset using some means of application control or execution denial (preventing unauthorized or unknown code from running at all), or via code signatures developed by antivirus vendors for malware samples using the technique. With an out-of-the-box deployment of SysMon, for example, the telemetry generated via execution will not present reliable events that would present definitive indicators of maliciousness. The anomaly presented would simply be the event indicating that a new process was executed. Endpoint controls, such as EDRs, may be able to show that a process exists in a suspended state, however suspended processes on their own are sometimes common on Windows hosts and are not necessarily a reliable indicator of attack.

These factors also make proactive threat hunting difficult for the technique on its own. Different vendors' tools may present varying degrees of process telemetry such as indicators of process suspension, but likely will not have an easy means of showing the associated usage of low-level Windows APIs such as WriteProcessMemory. Typically, low-level API calls such as those used in this technique are not presented to end users or security analysts, meaning that there is an implicit dependence on endpoint security product vendors to develop in-memory signatures specific to their product engine, where API calls and flows can be scrutinized in real time.

In theoretical terms, it appears plausible that a memory/behavioral signature could be written that identifies a sequence of events where the code calls ReadProcessMemory, which returns a value that matches the size of the .text section in ntdll.dll (0x119000, or 1150976 bytes for this version of Windows, sizes may vary for future versions). From there, if the event was followed by a WriteProcessMemory call which passes the same 0x119000 value, it might indicate that this technique is in use. Whether or not a scanning engine currently has the capability to check this kind of logic, however, is not necessarily clear. Typically, such specifics are typically closely held secrets for EDR engines. Regardless, this kind of behavioral pattern will likely not be presented to a normal security analyst or threat hunter working within a product's dashboards.

Techniques such as this one remains a persistent reminder that mousetraps must always be improving to keep pace with the cats. It is also important to consider that this code would only be one small part of a series of adverse events occurring on a computer, or elsewhere on the network, which may still be detectable with existing security controls. From an endpoint vendor perspective, you can't focus on one technique on its own, instead the focus needs to be on the bigger picture of all the different events in the sequence. Adversaries have numerous different ways of circumventing these protections to establish an initial foothold; focusing solely on detecting evasion-based or initial foothold techniques creates a narrow focus, resulting in blind spots. Post-exploitation activities still generate considerable telemetry that is often not being acted upon or ingested by organizations and fixating on a single novel method of loading malicious code such as this should not be a takeaway. Malware researchers can continue to evaluate this method to identify additional opportunities for detection, and we remain optimistic that new detection methods will emerge as research continues.