EDR bypassing via memory manipulation techniques

Written and researched by Connor Morley, WithSecure

WITH secure

Contents

Executive summary **4**
Chapter 1: Manipulation and hooking basics **5**
What is a memory manipulation technique (MMT)? **5**
Manipulating your own memory to avoid detection **6**
In-Line hooking module functions: an oldie but a goodie **7**
How to monitor for and detect in-line module function hooking 10
In-line hooking of undocumented module functions: a clever deviation **11**
Problematic detection, too much or too inaccurate 15
Legacy kernel patching for rootkit level stealth **19**
How does Kernel memory manipulation work? 20
Detecting Kernel level manipulation in and prior to build 18950 22
Is Kernel manipulation possible post 18950? 23
Chapter 2. ..... **24**
Heavens gate hooking AKA - how to be a syscall bouncer in x86 ..... **24**
How does Heavens Gate work? 25
How do we monitor for a sinful Heavens Gate? 27
Chapter 3. ..... **30**
Using Vectored Exception Handlers to side-step EDR **30**
What is a SEH and VEH exactly? 31
VEH always cutting the line to misbehave 34
PageGuard Integry bypassing 34
Force Jump 35
Silent bypassing 36
EDR targetted bypass – Firewalker 36
Detecting VEH usage 37
The problem with x86 and VEH 39
Summary **42**
Sources **43**

Executive summary

Endpoint Detection & Response systems (EDR), delivered by in-house teams or as part of a managed service, are a feature of modern intrusion detection and remediation operations. This success is a problem for attackers, and malicious actors have worked to find new ways to evade EDR detection capabilities. As with all arms races, these approaches to evading detection are creative and effective. One of the primary methods utilized in modern attack frameworks, hands-on keyboard operations and even malicious binaries revolves around memory manipulation.

Memory manipulation is nothing new; most readers will be familiar with process injection, thread hijacking, process hollowing and so on. That said, some recent tools/techniques are focused less on deployment and more on circumventing EDR telemetry acquisition techniques or alerting mechanisms. Elaborate hooking and exploitation of native functionality is now employed with impressive success rates.

This paper is broken down into three parts; the first will explain some of the memory techniques readily used by attackers to avoid detection in today's landscape, and will explain how they work and why they may be chosen. The second and third parts will focus on methods to detect the utilization of such covert mechanisms, where telemetry for detection may be acquired, and some of the difficulties that may be encountered during the integration of these solutions.

“ Memory manipulation is nothing new; most readers will be familiar with process injection, thread hijacking, process hollowing and so on.”

CHAPTER 1: Manipulation and hooking basics

What is a memory manipulation technique (MMT)?

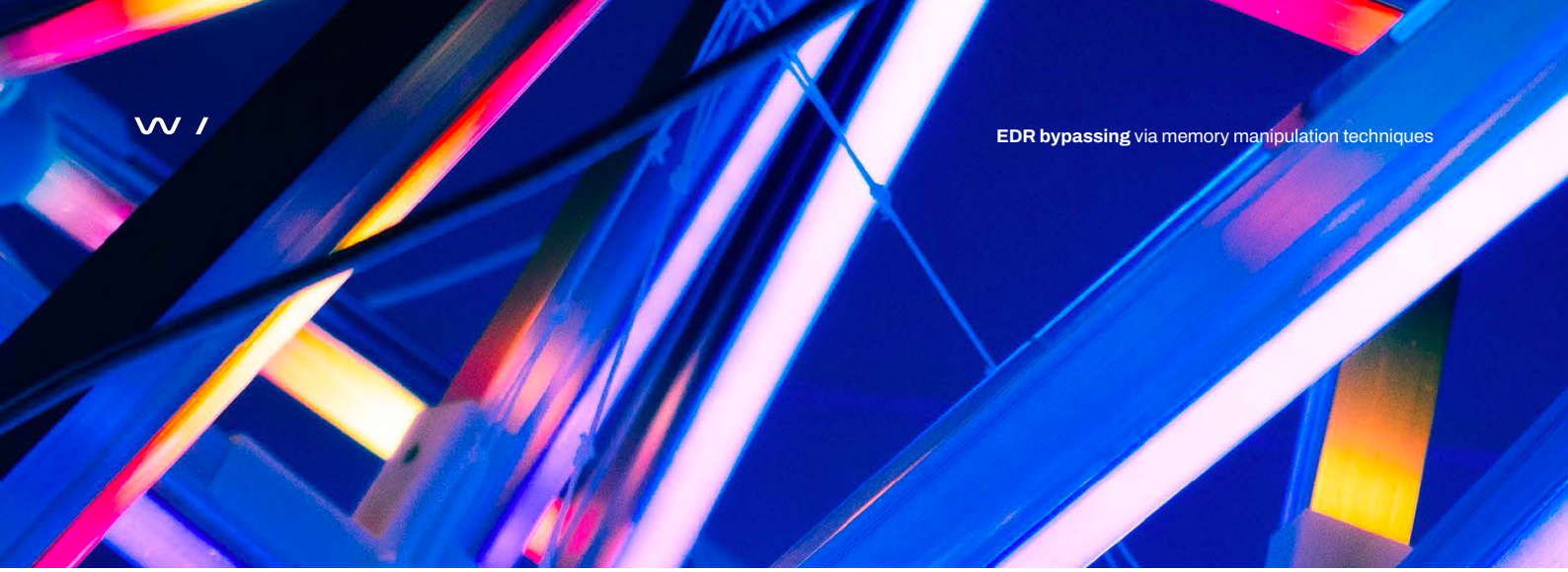
As the name implies, MMTs are the operation of altering the live memory of a system to affect some form of redirection or alternative operation flow.

As everything running on a system is loaded into live memory in order to execute (also known as volatile memory or RAM; not counting swap-space), all processes running commands are there in Opcode. Opcode is the abbreviation of “operation code”, which is another name for “instruction machine code”. Every function, comparison, reference, variable assignment and any other programmatical operation is conducted via Opcode, all of which are compiled from source code into program formats such as “.exe”.

Attackers have become adept at reading these loaded Opcodes and changing them in live programs to perform unexpected and unwanted actions. An attacker only needs to gain access to the target process handle to insert or overwrite memory values in the live process. This is remote process editing and is normally done to inject malicious code into a legitimate process via functions such as “VirtualMemoryAlloc”. The utilization of these functions, and the remote modification of a process, is something that defensive teams have known about for a long time and are well versed in detecting.

It's inevitable that attackers have changed tactics as a result. Instead of manipulating the memory of remote processes, they are instead manipulating the memory of their own malicious payloads and agents. Remote process editing requires the acquisition of a process handle and then the execution of well-known functions as outlined above. Editing the memory belonging to the current or malicious process requires no permission, as the process has the authority to edit its own memory, including that of shared library files (DLLs). This capability has made it a preferred choice for malicious attackers over recent years.

5



Manipulating your own memory to avoid detection

All running processes have full authority to edit memory related to them without restriction; this includes modules loaded into the process. Modules, commonly known in Windows environments such as Dynamic Link Libraries, allow more than one process to access the same modules' shared memory and functions, which helps efficiency. To prevent memory consumption, the first process to call a DLL will load it into a shared memory space which can be accessed by any other process. As the DLL contents are designed for reuse, any subsequent process which tries to load the same DLL will instead be referred to the previously loaded instance of the DLL and use the functionality stored within the shared libraries memory space.

This is important because of the way in which a lot of EDR telemetry collection mechanisms work, namely either via Event Tracing for Windows (ETW) or function hooking. ETW is a complex mechanism within the kernel level of the Windows Operating System (OS) - which we won't dive deep into - but it's important to know it is used to log system activity such as process execution, file access, network activity and many others. Function hooking is a mechanism by which EDR systems will set a hook on shared library functions for loaded processes. So, if a process calls that function, the hook will be triggered and the EDR agent will create appropriate telemetry. The hooking mechanism requires remote process writing operations which are conducted by a security vendor's signed/ trusted agent.

In both the examples above, the telemetry acquisition relies on a process' use of library functions to trigger either an ETW event to be generated, or an EDR hook to be entered. In both cases, the function paths required to reach to telemetry generation are dependent on Opcodes within the processes' own memory space, which as stated before, the process has full access to. If you know the command flow, how to detect function hooks or even a lynchpin function in the publication process, you can theoretically edit the Opcodes to prevent the publication ever being reached. This is how EDR evasion MMT works.

In-Line hooking module functions: an oldie but a goodie

Hooking is a technique that has been used by computer programmers for a very long time, for legitimate reasons. It's also a long-standing malicious technique. In-Line hooking refers to the interception of calls to a specific function by altering the "in-line" Opcode of the function itself, essentially overwriting part of the desired functions Opcode in order to effect a redirect. As mentioned previously, EDR and even some Anti-Malware (AM) systems use this technique to collect telemetry on the utilization of specific functions or even to alert if specific functions are utilized. Malicious actors instead use the technique to redirect command flow to malicious code sections, bypass telemetry publication or even simply to prevent function execution entirely.

Let's look at the simplest in-line hook available, the premature return hook. Almost all called functions in a DLL will have a "RET" or "return" Opcode in the functions memory section. When the "RET" command is hit, the control flow is returned to the calling function with the stack being cleaned up as part of the operation. This means that, unless there is return value checking, any function that runs to a "RET" command will return control to the calling function and the program will continue uninterrupted. Let's say, as an attacker, you wanted to open a file which involves using system call, functions. By examining the call stack to the system call, you find that during the ETW publication process, the function "EtwEventWrite" is called every single time. Examining the functions contents in the loaded DLL, you see it looks like:

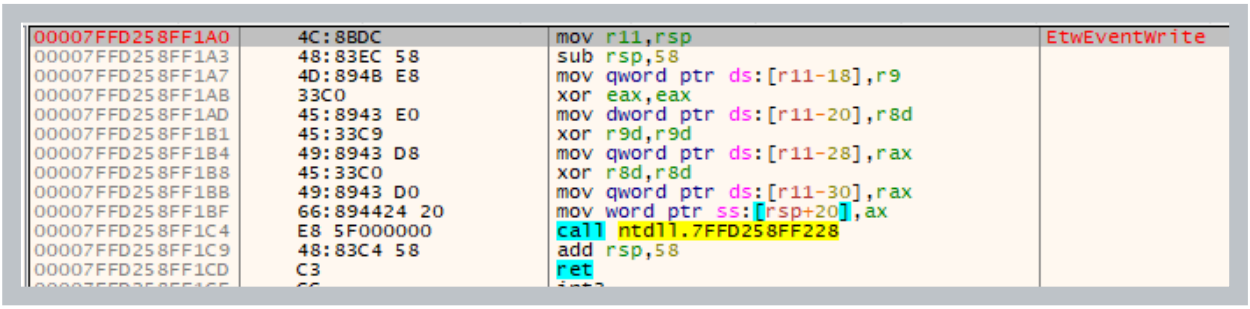


Image 1. – ntdll.dll EtwEventWrite unaltered

7

Following the “CALL” opcode, you know that this will result in an ETW event being published which logs the file opening event, and you want to prevent that from happening. So to prevent the “CALL” code from being reached, you simply replace the initial command with the value “C3”, which in Intel x64 Opcode means “RET”:

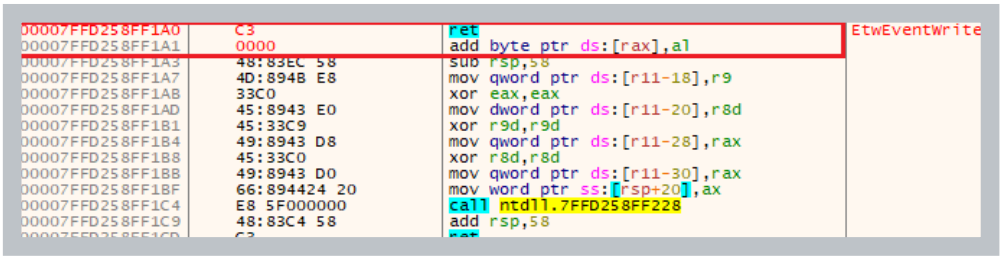


Image 2. – ntdll.dll EtwEvent-Write premature RET in-line hook injected

Now the attacker can perform any functions they want and no ETW events will be published as all control flow to the function “ETWEventWrite” will automatically return with the desired value (in this case, 0 for SUCCESS), and the program will carry on believing the publication executed as intended. This is what is called a “general bypass hook” as it does not target one specific function but instead targets a “lynchpin function” which in turn prevents all general ETW event publication from succeeding with one edit. This can be done with as little code as:

```
var patch = new byte[] { 0xc3 };
try
{
    uint prot;
    var module = Win32.LoadLibrary("ntdll.dll");
    var funcAddress = Win32.GetProcAddress(module, "EtwEventWrite");
    Win32.VirtualProtect(funcAddress, (UIntPtr)patch.Length, 0x40, out prot);
    Marshal.Copy(patch, 0, funcAddress, patch.Length);
}
catch
{
    Console.WriteLine("Error during inline hook");
}
```

Image 3. – Source for in-line hook RET injection to EtwEventWrite

Targeted hooking would instead focus on the file open function rather than at further down the call stack (such as at EtwEventWrite) and as such, would only affect that particular function. The reason why an attacker may choose a targeted solution over a general solution is to prevent detection by absence of telemetry. If every process on a system is generating at least some ETW events, monitoring for processes that are generating none is a trivial statistical analysis which would quickly highlight such hook functions.

There is another solution; a hooked filter function. If the in-line hook redirects the system call to a program defined memory space rather than simply performing a “RET” command, an attacker can install an event filter which will allow through the majority of events for publication and only omit events which would contain otherwise anomalous entries. This could be filtered by checking the target file, a registry value, a Boolean flag or any other number of indicators designed by the malicious actor. In this way the program would generate normal traffic whilst carefully removing anything that an EDR solution or threat hunting team may deem suspicious.

Another common reason for in-line hooking is simply to trigger malicious code to execute. Rather than executing a malicious memory section directly from the malicious process’ main memory space, by hooking a library function, you can instead hijack the command flow so that the malicious code is executed via the syscall command flow, preventing the need for creating a remote thread which is commonly detected.

How to monitor for and detect in-line module function hooking

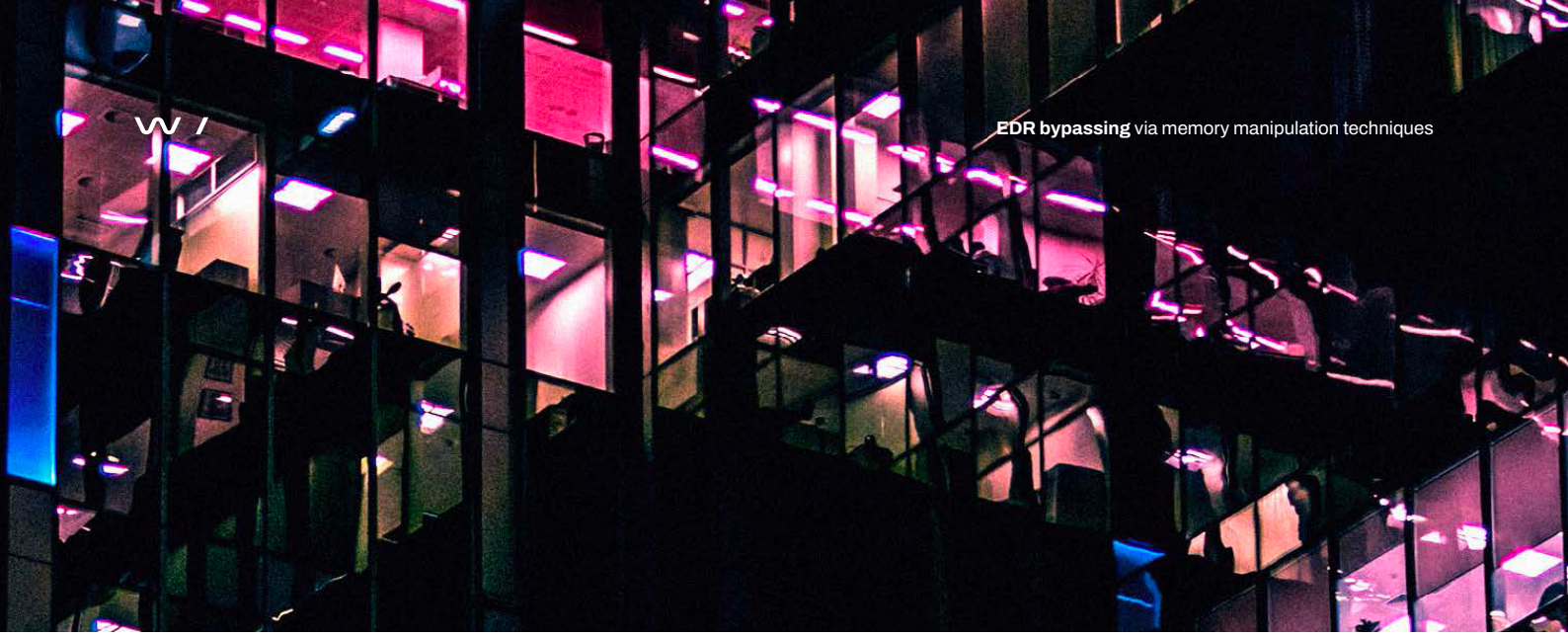
First off, there is no efficient way to monitor for this dynamically as alteration of internal/process owned memory does not necessarily require a system call ("memcpy" for example) in order to implement the required alterations. As such, remote process memory access events would need to be monitored, but as these are extremely verbose (for legitimate reasons), this creates a resourcing issue. Therefore, for resource management reasons, this is typically done via sweeps of running processes on a regular basis. The obvious caveat to this is that short lived payloads may omit detection of such hooking depending on the polling mechanism employed (time offset of initial execution or global time delay), however this should be very effective for persistent payloads or malicious agents.

To determine if a malicious alteration has taken place, a known "good copy" of the Opcode of the module must be available to compare for deviation. The "good copy" is taken from the module files found within the file system of the host OS before any in-line modification would have taken place. Thankfully, all Windows modules are signed and are available in the OS core directory. As such, it is possible to determine if malicious (unsigned) modules have been introduced as well as performing on-disk to live memory analysis. The same cannot be said for third-party modules and as such, monitoring can only compare what is available on the file system against what is in memory. This does not prevent malicious actors from directly altering the contents of third-party modules in order to affect malicious hooking (or more accurately patching). If reliable hashes are available for third-party modules, this type of monitoring would be trivial to introduce, but this is not readily supplied by all vendors.

In-line hooking of undocumented module functions: a clever deviation

This is a trickier version of module function in-line hooking to monitor, but requires the attacker to conduct additional preparation. Typically, an attacker will identify an exported function from a module such as “EtwEventWrite”, which is called as part of the publication command flow, and patch it as outlined in the previous section. This is simple because “EtwEventWrite” is documented, so its functionality can be determined. It is also an exported function, so is contained within the module's Export Table (ET), which is part of the module's Portable Executable (PE) Header. Being included in the ET means that the Relative Virtual Address (RVA) of the function within the modules memory space is specified in the ET in order to allow external processes to know where to find the function they want to use. Using the RVA (or offset), an attacker can find the Base Address (BA) of the module, which is the physical address at which the module was loaded into live memory, add the RVA and find the functions code in the modules memory space. An easier method is demonstrated in the code in image 3 by using the function “GetProcAddress” which references the ET of the loaded module “ntdll.dll” to calculate and return the address of the desired function. Alternatively, this may be resolved within the loader of the process and be accessible from the Import Address Table (IAT) if a manual approach should to be avoided.

This same function resolution can be used from a defensive perspective. As all the exported functions can be located by examining the ET of a module's PE header and then using the Process Environment Block (PEB) of the target process to find the base address of all the modules loaded by the processes, scanning all, or a subset of the available functions, is trivial. The problem is that not all the functions in a module are exported, and as with almost all programs, some



functions are private internal functions. These internal functions can be used for a limitless number of purposes such as argument sanity checking, reformatting values, creating additional variables etc. But, as they are not exported, there is no way to know where they are located within a module's memory space without access to its associated Program Database (PDB), which is generated at compilation. So, if an attacker analyses the stack trace of a specific function they want to execute from the point it reaches out to something such as an ETW publication function, and find an internal function within the module which is called between executing the command and ETW publication, they can simply edit that function instead of the initially called exported function and still prevent ETW publication.

One such example was demonstrated by researcher Adam Chester in relation to DNS queries and how by examining the call stack when the program hit the ETW publication function, you can identify internal functions within "dnsapi.dll" executed as part of the command flow. In this case, internal function (identified through PDB available on Microsoft Public Symbol Server[2]) "MCTemplateU0zqxqz" and "MCGenEventWrite", are called after execution of the DNS query elements of "DnsQuery_A" function, but before reaching "EtwEventWriteTransfer" in "ntdll.dll". Chester provided the stack trace from his research which highlights the potential target, which can be seen in Image 4.

Stack

Frame Index	Name
[0x0]	**DNSAPI!McGenEventWrite + 0x3f**
[0x1]	DNSAPI!McTemplateU0zqxqz + 0xdb
[0x2]	DNSAPI!Query_PrivateExW + 0x27ae1
[0x3]	DNSAPI!Query_Shim + 0xbd
[0x4]	DNSAPI!DnsQuery_A + 0x29
[0x5]	DnsQuery!main + 0x3c

Image 4. – Stack trace showing internal functions before ETW publication [3]

[1] https://blog.xpnsec.com/evading-sysmon-dns-monitoring/
[2] https://msdl.microsoft.com/download/symbols
[3] https://blog.xpnsec.com/evading-sysmon-dns-monitoring/

DnsQuery_A eventually calls the function “MCTemplateU0zqxqz”, which is used to create an ETW event type template for the specific action. This is then handed off via a call to “MCGenEventWrite” as shown in Image 5.

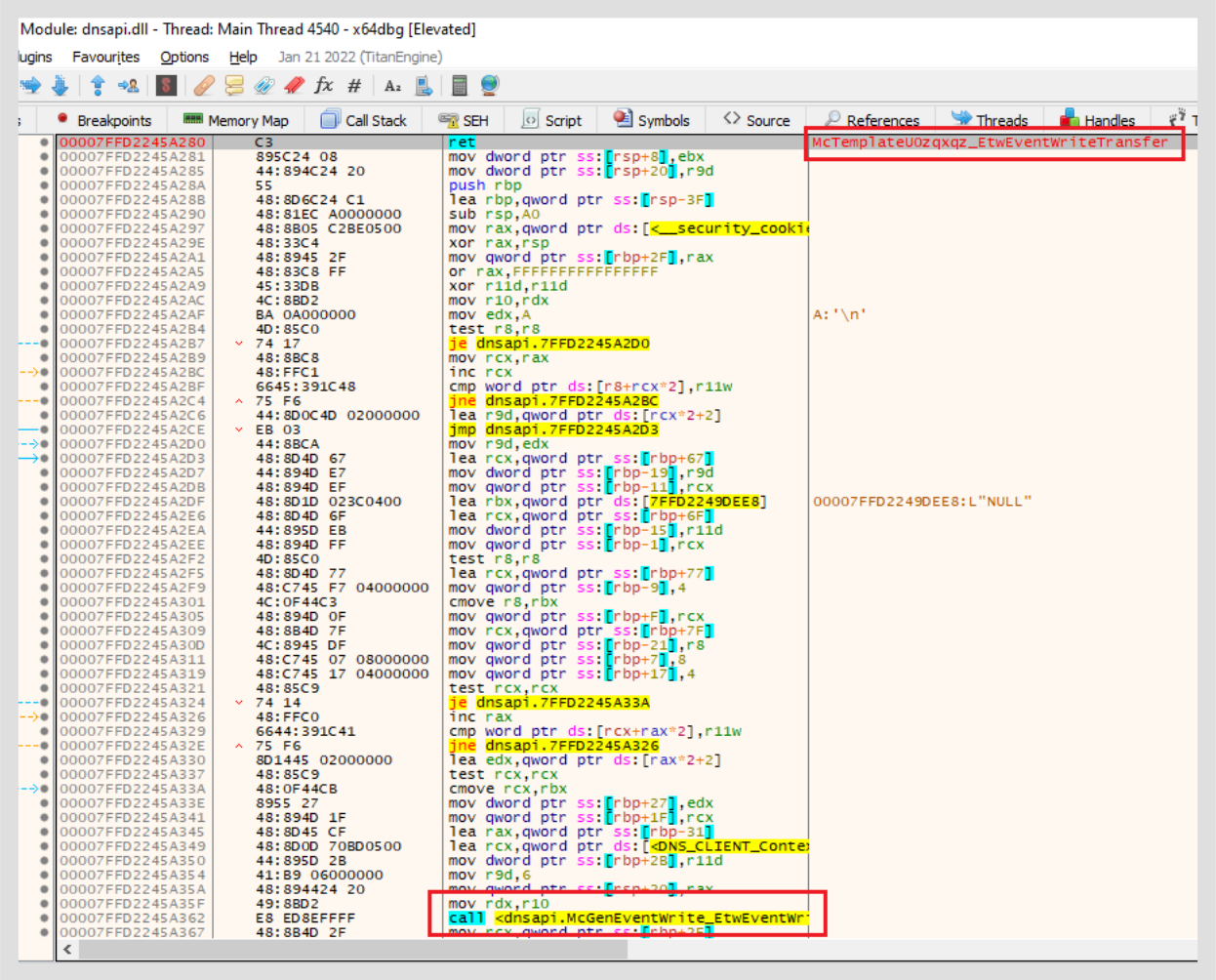


Image 5. – MCTemplateU0zqxqz content, shows hand off to McGenEventWrite

“McGenEventWrite” then creates the actual event object and calls function “EtwEventWriteTransfer” within module ntdll.dll to perform the actual event write action at the kernel level. This is demonstrated in Image 6.

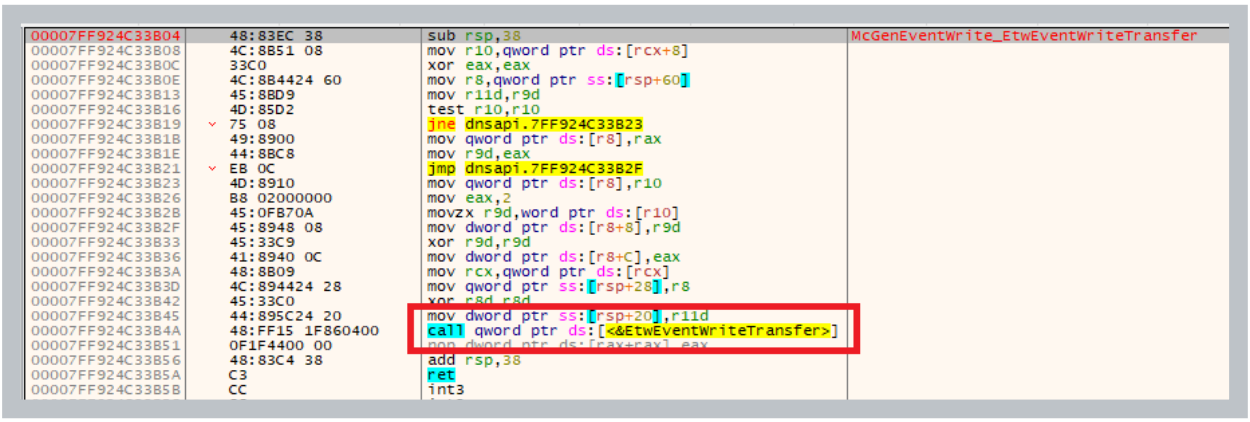
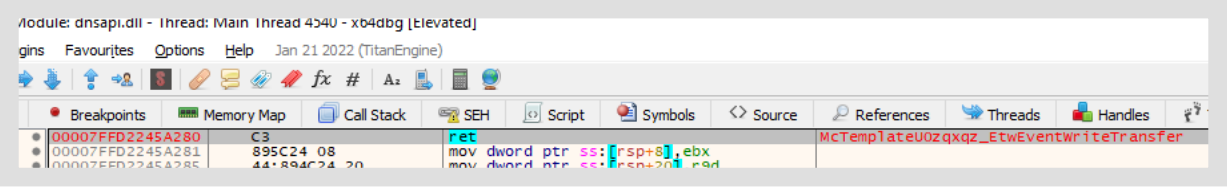


Image 6. – McGenEventWrite handing complete template to ntdll.dll EtwEventWriteTransfer

In their example, by targeting “MCTemplateU0zqxqz” and using the standard “RET” command insert technique, Chester is able to completely prevent any logging of the “DnsQuery_A” function call activity. This means that, if this was employed in a malicious agent or malware, any DNS query could be executed safely in the knowledge that no ETW telemetry was being created due to the premature “RET” opcode in the command flow. Essentially, this is a more advanced and targeted method of standard in-line patching which requires private function enumeration and targeting on the part of the attacker in order to be effective.

Image 7. – Private in-line hook with premature RET to prevent logging



We mentioned earlier that this was a trickier method to monitor for, due to the lack of entry for either “MCTemplateU0zqxqz” or “MCGenEventWrite” in the ET of the module “dnsapi.dll”. The reason this is trickier is that, without PDB files being readily available, there is no way to know where this function resides in the loaded module version with any reliability, and there is no way to find it dynamically without analyzing the call functions in the module on-the-fly, which is impractical. Therefore, we cannot specify this internal function to be monitored specifically for modification. So, effectively, we are left with no alternative other than to analyze the entire module's memory space for alterations.

Arguably, you could download PDB files as required from the Microsoft Public Symbol Server for encountered modules and then use the symbol references to determine the internal targeted function's location and size for analysis. This would require one of two things; either all available PDB files installed alongside the scanning agent, or an internet connection which allows for dynamic download of the PDB’s as required. The former option will add bloat to the agent and will need constant updating as new versions of modules are routinely available, so a local DB may quickly become inefficient. The latter option would create unnecessary internet traffic and connections which can create event bloating or could be impractical on endpoints that may have specifically limited network access due to security requirements. Either option is impractical at scale.

Problematic detection, too much or too inaccurate

In the example above, the problem is easily represented as being unable to find two specific internal functions as they are not in a module's ET. However, a hook within an internal function could be used for malicious redirect just as easily as function neutering purposes. With the latter, if this was all we were interested in, to have effective coverage, we would need to identify all potential command flows to the process being targeted for neutering. With the prior example, the interrupt was targeted at “DnsQuery_A”, but what if instead, it targeted “DnsQuery_W”? Without examination there is no way to be certain this would use the same McGen and McTemplate functions. Therefore, in order to have accurate coverage you either need to individually map every control flow (or stack trace) of all module functions of interest, or scan the entire module's memory.

This then leads to another issue. Although scanning an entire module's memory is possible, it does not allow for attribution of any detected alterations to a specific function. For example, an alteration detected at offset 0x1000 of a module does not correspond to any exported function in the ET. Although an alteration has taken place, without a PDB we cannot identify which function this is related to and as such cannot identify it as being part of any known command flow which may have been mapped out previously. This is important as due to variations in function locations between different versions of modules which may be encountered, a universal offset identifier is unreliable in practice. The image below shows some function alterations observed in different ntdll.dll versions.

Function	Remarks
AlpcRundownCompletionList	
EtwEventWriteEx	declared in Windows 10 WDK
EtwEventWriteNoRegistration	declared in Windows 10 WDK
EvtIntReportAuthzEventAndSourceAsync	
EvtIntReportEventAndSourceAsync	
ExpInterlockedPopEntrySListEnd16	x64 only; discontinued in 6.3
ExpInterlockedPopEntrySListFault16	x64 only; discontinued in 6.3
ExpInterlockedPopEntrySListResume16	x64 only; discontinued in 6.3

Image 8. – ntdll.dll version alterations – Geoff Chappell [4]

[4] https://www.geoffchappell.com/studies/windows/

Finally, there is the issue of double dependency within module structures. All modules have an ET in their PE header. However, they also may import functions from other modules which is specified in the Import Table (IT) of the PE header. It is possible that some functions in one module may in turn call functions from another module and so forth. Therefore, you will need to map all potential paths for function execution across double dependent modules in order to have effective coverage of internal functions which may be executed when passing between loaded modules. As any internal function hook can be deployed anywhere in the call stack, malicious actors can potentially map across multiple modules and input a malicious hook at any point to either return, redirect, or simply sidestep unwanted monitoring.

“As any internal function hook can be deployed anywhere in the call stack, malicious actors can potentially map across multiple modules and input a malicious hook at any point to either return, redirect, or simply sidestep unwanted monitoring.”

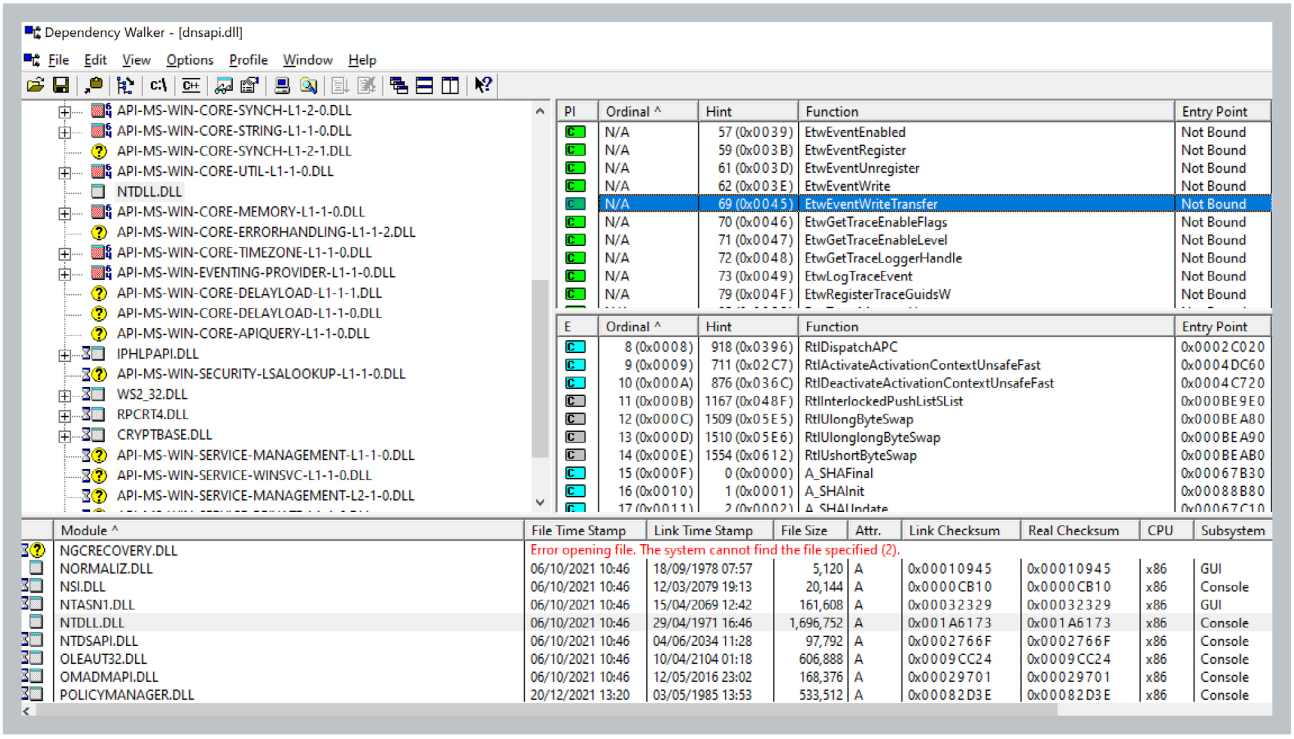


Image 9. – Dependency tree for functions within dnsapi.dll showing double linked dependencies

As such, the only viable solution is to monitor for all modifications made to modules in live memory. We mentioned earlier how loaded modules are shared libraries that are used by multiple processes accessing the same memory location in order to prevent unnecessary memory usage. So, it would be fair to assume that alteration of a loaded module's memory space would affect all processes running on the system as they are all accessing the same module, but this is not the case. Instead, the OS recognizes that the memory of the module has been modified and makes a process specific copy of the memory page of where the alteration occurred. Subsequent actions from that process which conduct activity related to that memory section, are instead redirected to the process specific copy memory page for that function.

Memory pages are the mechanism by which an OS separates memory into blocks in live memory, typically 4096 bytes on most modern OS, and is used for multiple advanced management mechanisms. In the case of a module modification, the entire page where the modification occurs, is copied to a process specific version (as explained above), called CopyOn-Write(COW). This means that 4096 bytes is copied to an isolated memory page which the process thinks is still part of the module memory space. It enables the OS to allow processes to modify modules, which is a common occurrence for legitimate purposes, without requiring the entire module to be copied or reloaded to a process specific version, instead settling for just 4096 bytes. Now that we have an understanding of how the OS handles such modifications to the memory, we are presented with two options to scan for deviations; scan every single byte, or check for COW pages.

A byte-by-byte comparison will examine every byte of a module's live memory against that stored on the file systems “good” version, much like we would use for an exported function comparison. Doing this for every single module of a process and all its dependent modules, would require the comparison of a lot of memory which would then need to be repeated for every process due to the page extraction on modification. By rough analysis, the average dual linked dependent modules for programs is 695 modules (ranging between 583 and 804) on a test system. Multiply this by every process running on the system, and the resource requirements to conduct a full scan become unmanageable. As such, although this would work, it is impractical for such large scale comparison.

An alternative solution is to scan for COW pages, which was brilliantly explained by Ollie Whitehouse at NCC Group with a released PoC made for public use[5]. We will provide a brief overview of this mechanism (which they have done excellent work in monitoring), but we strongly recommend examining the PoC and blog post[6] to get full information. Essentially, it is important to know that when modules are loaded into memory to be shared between processes, they are assigned the memory protection set “MEM_MAPPED” or “MEM_IMAGE”. Typically when a page goes from shared to private (as in the case of COW), the protection set should get changed to “MEM_PRIVATE”, however in the case of modules the protection set remains at either “MEM_MAPPED” or “MEM_IMAGE”. But, more important, is an extended attribute of the page, which is found as part of its working set information, the “shared bit” (SB). If a COW page is created when a module is modified, the page will have its SB cleared (set to 0), which should not be the case in standard shared module memory space. As such, we are able to scan entire memory pages assigned to a processes module memory section and check only the extended attributes SB to see if a modification has taken place, instead of checking byte-by-byte which is a significant decrease in overhead.

As a side note, it is important to know that this analysis requires the use of “psapi.dll”, which is not available in C#. Our initial attempts to utilise this solution via C# meant a necessity to use a DLL bridge in order to accommodate the use of the “psapi.dll” functionality. This led to a heavy translation overhead and skewed the resource requirement analysis. Running this type of analysis from appropriate language bases (such as C++) will yield much faster results.

Checking the extended attribute uses the command “QueryWorkingSetEx” from the mentioned module “psapi.dll” to acquire the virtual attributes of the page

[5] https://www.geoffchappell.com/studies/windows/

[6] https://research.nccgroup.com/wp-content/uploads/2022/11/Ollie-Whitehouse-Tales-of-Windows-detection-opportunities-for-an-implant-framework-1-1.pdf

which contains the SB value. In testing, this proved to be very quick and a potentially feasible workaround to the resource consumption issue. However, as this checks for deviations on an entire page, it is also inaccurate. A memory page can potentially contain multiple functions and the COW check does not provide accuracy on what has been altered on the relevant page. A subsequent byte-by-byte comparison will still be required to clarify what exact memory location has been modified, but in a much more targeted way.

This then comes back full circle to the original problem with internal functions. Even if you can detect at scale that modifications have taken place, how do you attribute them to an internal function that may be part of an exploitable command flow? The answer is that without a PDB (which we mentioned was unmanageable), you can't. However, even if we did, there is no way to determine that this internal function is part of an exploitable command path without mapping all valuable offensive command flows across dependent modules. There is also the issue of False Positives (FPs) which are generated by the legitimate modification of modules by processes.

A primary example of this can be found in FireFox where multiple modifications of live modules occur legitimately as part of its operation. As a module scan would pick up every deviation on every page and (for resource purposes using COW) would be inaccurate, there is no way to determine what function the modification belongs to with any certainty. This is because module functions are not given a definitive size in the ET, only their location to be called from. As such, if an internal function followed an ET function, it would be easy to missattribute the modification with the ET function, which may seem legitimate. The problem of function boundaries, is an ongoing one which is being tackled from multiple angles. Some interesting research on this subject can found from back in 2019 by Jim Alves-Foss and Jia Song at the University of Idaho[7] which provides a detailed description of the problem. In their cases, they are analyzing a stripped binary, however without a PDB file, the internal functions are in essence also stripped.

```
firefox.exe, c:\windows\system32\ntdll.dll, NtOpenThread, c:\program f
firefox.exe, c:\windows\system32\ntdll.dll, NtOpenThreadToken, c:\prog
firefox.exe, c:\windows\system32\ntdll.dll, NtOpenThreadTokenEx, c:\pr
firefox.exe, c:\windows\system32\ntdll.dll, NtQueryAttributesFile, c:\
firefox.exe, c:\windows\system32\ntdll.dll, NtQueryFullAttributesFile,
firefox.exe, c:\windows\system32\ntdll.dll, NtSetInformationFile, c:\p
firefox.exe, c:\windows\system32\ntdll.dll, NtSetInformationThread, c:
firefox.exe, c:\windows\system32\ntdll.dll, ZwCreateFile, c:\program f
```

Image 10. – FireFox legitimate alterations to ntdll.dll skews results

As a result of all these issues, either from an accuracy, resource or efficacy perspective, monitoring of internal function hooking is problematic, with no single solution that best fits all scenarios. This is a primary example of detection development issues where accuracy and resource requirements have to be balanced against one another in order to come up with a viable solution that best suits the majority of cases. The most effective solution is to use a network-enabled agent to download (and archive) PDB files as required for all module versions found for running processes, and then use the COW page search as a preliminary scan with byte-by-byte scanning as a definitive address identifier. However, this does not take into account network security requirements or estate restrictions which may impede the ability for PDB file acquisition which would then hinder accurate analysis. This is a judgement call.

[7] https://dl.acm.org/doi/abs/10.1145/3359789.3359825

Legacy kernel patching for rootkit level stealth

The aforementioned is a prime example of why updating/patching is so important in relation to security posture. In Windows versions prebuild 18950 due to a flaw in one of the kernel level logging functions, it was possible for malicious actors to implement a hook within the kernel to conduct malicious activity. In the case of ETW, bypassing this allowed all user level monitoring to be untouched, yet still no ETW events would be generated according to the malicious users' objectives.

The most effective tool for this is "Ghost-In-The-Logs" (GITL), developed by offensive security expert bats3c[8]. This tool is comprised using functionality from two other pre-existing projects which are fundamental in its capabilities. These are Kernel Driver Utility (KDU) by hfiref0x[9] and InfinityHook by everdox[10]. By combining the capabilities of InfinityHook and KDU into one tool, GITL allows a script-kiddie level deployment for kernel hijacking which is impressive, and (from a security perspective) extremely concerning.

How does Kernel memory manipulation work?

In order to understand kernel manipulation, the two primary components, KDU and InfinityHook, need to be explained. The first tool to be used is KDU which allows for an arbitrary write into kernel memory space by exploiting known vulnerable drivers on a system. It is worth noting that the vulnerabilities in the drivers used by KDU are still present, even in the latest versions of Windows, and therefore can be exploited, even with full patching.

We will not go into the low-level technical explanation of how KDU works as that can be extremely lengthy and has been covered in numerous other articles. For this functionality, it is only important to understand that KDU is used to allow for primitive read & write functions into kernel memory space. Once these primitives are acquired, a malicious kernel driver can be written to kernel memory and then bootstrapped to the vulnerable driver and loaded, which effectively gives a malicious actor kernel level access through their driver.

Image 11. – Overview of KDU operation

Once the malicious driver is loaded, it uses InfinityHook capabilities to search for the array of WMI_LOGGER_CONTEXT objects within the kernel space. A pointer to this array is known to exist just after the EtwpDebuggerData function which can be signature scanned for in module memory. Once the array has been acquired, it then scans for the entry which corresponds to the “circular kernel context logger” instance as it is typically always running; if it is not running, it is enabled.

It is at this point the logging vulnerability is exploited; the vulnerability is the use of a pointer within the WMI_LOGGER_CONTEXT object type. “GetCpuClock” is a pointer which typically points to one of three system time acquiring functions. This pointer is called every time an event is created in order to apply a timestamp. By using the RW capabilities of the driver a malicious actor can simply overwrite this pointer to install a malicious hook so now on every syscall event generation, their hook will be hit before the syscall is executed.

[8] https://github.com/bats3c/Ghost-In-The-Logs
[9] https://github.com/hfiref0x/KDU
[10] https://github.com/everdox/InfinityHook

Image 12. – Overview of InfinityHook hijack operation

Within GITL, this hooking capability is used in order to prematurely exit the logging function and prevent any ETW logs from being generated. This is a generic ETW event dropping capability which works very well, it does open up the door to detection via event absence. However, due to its simplicity in use, it is an attractive option and could be used as a code base for other capabilities.

A more complex method that InfinityHook has the capacity for is to instead only drop events for specific syscalls or ETW event types. This is done by walking up the stack from the hook to find KiSystemCall64 and acquiringing the SystemCallNumber and arguments. This works because prior to calling the logging function, the kernel resolves the syscall number/ID in a variable which allows for the hook to know what is being called and with what arguments. This would allow for targetted filtering, either at the kernel level from within the driver through pre-defined rules, or from userland by providing additional IOCTL flags, which can specify whether an event should, or should not, be logged on a call by call basis.

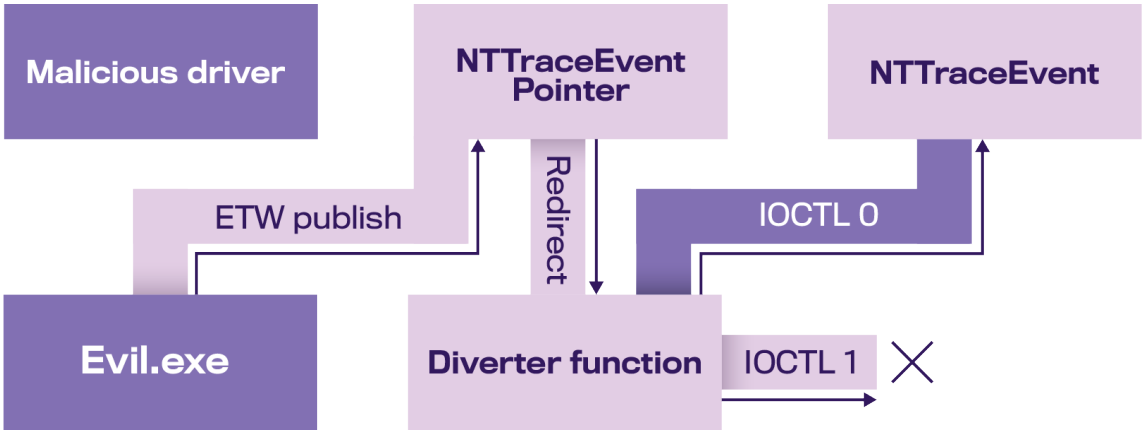


Image 13. – Overview of userland control over Kernel hook

Finally, the hook could instead be used to execute arbitrary syscalls which differ from those called within the user level. By stack walking to the defined syscall ID and arguments, the hook content could modify the contents of these variables before they are passed for execution to anything the malicious actor likes. As such, an innocent syscall from userland could be modified to conduct any syscall activity the attacker would like, with no trace log being generated and no user level activity which would correspond to it.

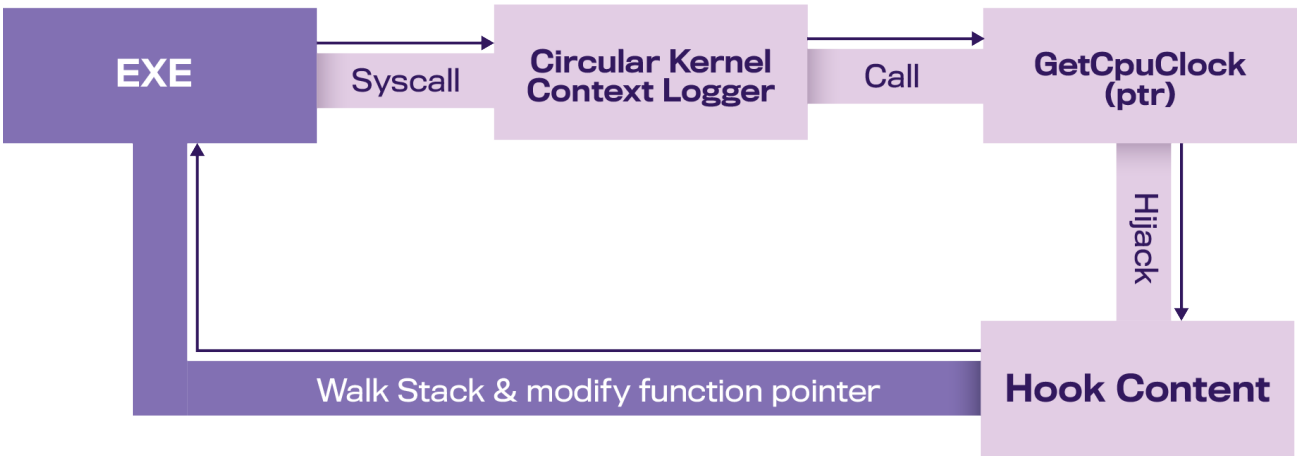


Image 14. – Overview of Kernel hook code being used to execute arbitrary syscall

21

Detecting Kernel level manipulation in, and prior, to build 18950

There are two main areas that can be monitored; the value of the GetCpuClock pointer within the kernel memory space, or the loading of known malicious drivers within the user space.

Monitoring at a kernel level would require a monitoring kernel driver to be loaded. This is so it can monitor the memory of the WMIC_LOGGER_CONTEXT entries for alterations and then communicate when an alteration happens to a userland agent, which ideally would be event-driven rather than polled. Although this is possible, it can be extremely complex and has the potential to open security holes by introducing a new user to kernel space access to control a method to determine when to check the context objects. Additionally, as with all kernel side elements, it has the capacity to introduce instability into a system which can lead to undesirable events such as Blue Screen of Death (BSOD)/kernel panics. With the potential introduction of risks into vital systems by being monitored this way, this may not be the most efficient way of monitoring for such activity.

Thankfully, due to the reliance on KDU in order to acquire the primitive read & write functions for malicious driver bootstrapping, we have a user space monitoring option. The drivers that are leveraged by KDU are well documented on regarding their vulnerabilities with associated CVEs, they have never been fixed/patched/mitigated by the vendors. Although this seems like a massive oversight, due to the extreme rarity in which these drivers are loaded, it is not much of a problem from a monitoring perspective. One of the most readily available (and exploited) drivers is “NalDrv.sys”, originally named either “IQVW32.sys”, or “IQVW64.sys”, which is the “Intel Ethernet Diagnostics Driver”. However, the legitimate use and load of the driver, or associated service, is extremely rare despite it being present on almost all Windows systems natively.

> “ The drivers that are leveraged by KDU are well documented on their vulnerabilities with associated CVEs, however they have never been fixed/patched/mitigated by the vendors. ”

If KDU does not find “NalDrv.sys”, it will then scan the system in a linear list of known vulnerable drivers to see if they are available and load the first one that is available in order to allow kernel memory access. The list is:

- NalDrv
- RTCore64
- Gdrv
- ATSZIO
- MsIo64
- GLCKIo2
- EneIo64
- WinRing0x64
- EneTechIo64
- phymemx64
- rtkio64
- EneTechIo64
- lha
- AsIO2
- DirectIo64

Therefore, monitoring systems for the presence of any of these drivers being active/loaded is typically a good indicator that KDU is being used, either maliciously or for development/research activity. In either case, an investigation into the nature of the tools use would be required due to the access it permits to the user. This can either be done via polling the system drivers or by monitoring TI ETW feeds, such as TI driver object creation and load events.

Is Kernel manipulation possible post 18950?

Using the methods in InfinityHook and by extension GITL, the kernel cannot be manipulated in the same way, post build 18950. This is because Microsoft specifically changed some of the kernel event management structure in order to prevent this type of exploitation from taking place. That means if you try to use the same exploit you end up with a “KERNEL_SECURITY_CHECK_FAILURE” panic/BSOD event. The descriptor of the BSOD is:

> *A kernel component has corrupted a critical data structure.The corruption could potentially allow a malicious user to gain control of* ***this*** *machine.*

Closer examination shows this is because Microsoft changed the value type of GetCpuClock from a pointer to an integer. Therefore, when the hook pointer is attempted to be written to the GetCpuClock variable, it results in corrupting the structure, which in turn, triggers the BSOD. This also applies to other tools which use the vulnerable pointer to circumvent security measures such as ByePG (Bye Patch Guard).

However, that does not mean there are no other kernel memory manipulation techniques being used in the wild, or will appear in the future, but only that this method has been closed. As KDU still works in current Windows versions there are still arbitrary kernel level access capabilities (even if they are easily monitored) which could be leveraged in currently unforseeable ways in the future.

23

CHAPTER 2.

Heavens gate hooking AKA - how to be a syscall bouncer in x86

The change from 32 to 64 bit computing might feel like ancient history, but it remains relevant today. To continue using older programs and libraries, computers with a 64bit architecture need to emulate a 32bit system, and this requires the system to be able to switch CPU modes..

From a malicious standpoint, this type of memory manipulation occurs at the point between an x86 process (x32 program running on x64 architecture) and the native x64 kernel. As an overview, this works by targeting the functions which are involved in translating syscalls from the supported or emulated x32 program execution into a format or mode, which the native hardware is able to manage. For example, if a user wants to access a file from an x64 process on an x64 architecture machine, a Syscall can be made through the appropriate libraries (or via direct Syscalls, but that's a different subject) to the kernel. However, if you are running an x32 program through the Windows On Windows (WOW) architecture, then the x32 Syscall has to be translated/ switched to its x64 version before the kernel will be able to handle it. This translation or switch mechanism that is commonly referred to as "Heavens Gate", is the gateway between the two architectures and handles the CPU mode switch.

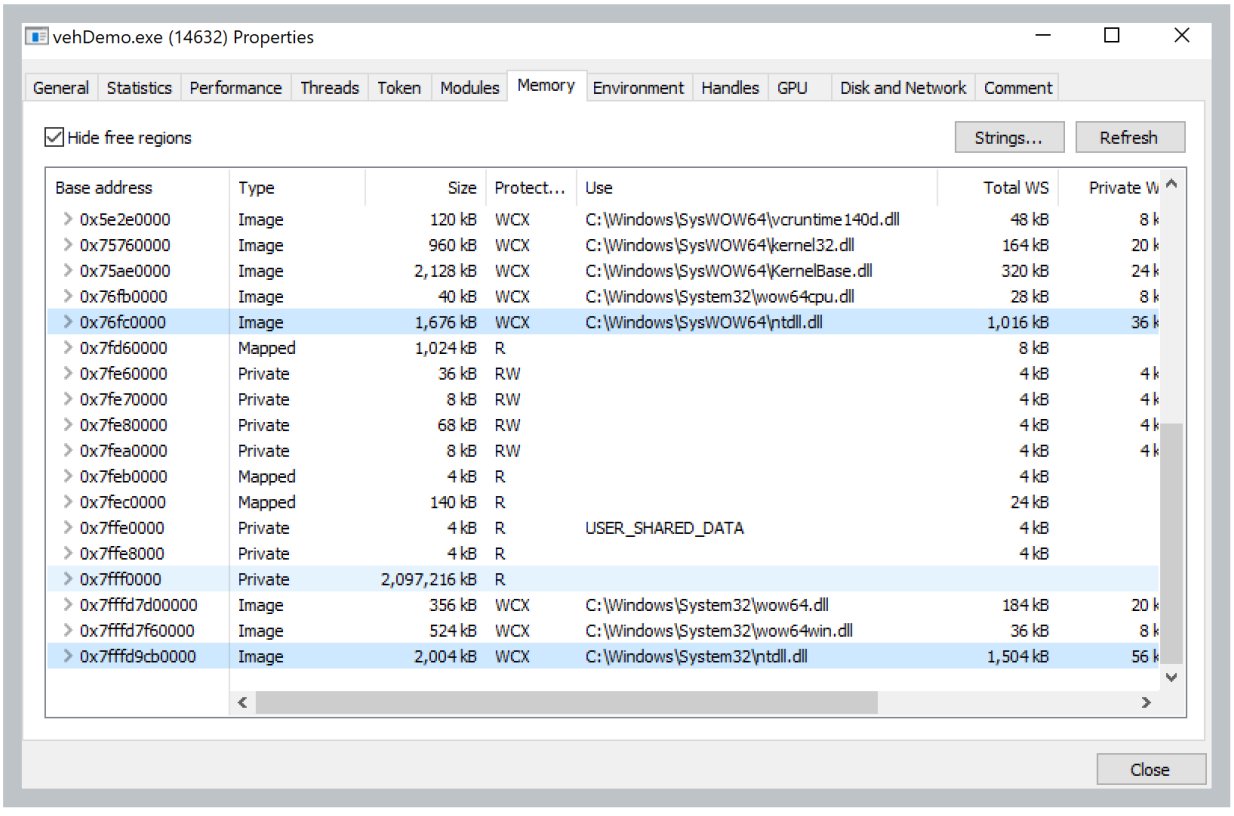
How does Heavens Gate work?

The nature of how Heaven's Gate works is complex at a lower level - so it will be explained at a moderately high level. For the following explanation, we will assume the native architecture is x64, and that x86 refers to WOW processes. If a x64 process wants to perform a syscall to the kernel, there are a number of native libraries which accommodate such functions, the primary of which is ntdll.dll. When a program needs to conduct some function, it will call the exported function of ntdll.dll, which in turn will perform the required syscall to the kernel and handles the results back to the calling process. However, you cannot perform a x32 syscall to a x64 kernel, therefore x86 processes cannot perform syscalls from the x32 user space created via WOW architecture.

> "The nature of how Heaven's Gate works is complex at a lower level - so it will be explained at a moderately high level."

To get around this problem, x86 processes actually load both the x32 and x64 ntdll.dll instances. By doing this, the program has visibility to both the emulated x32 ntdll.dll functions and those of the native version, which can perform syscalls.

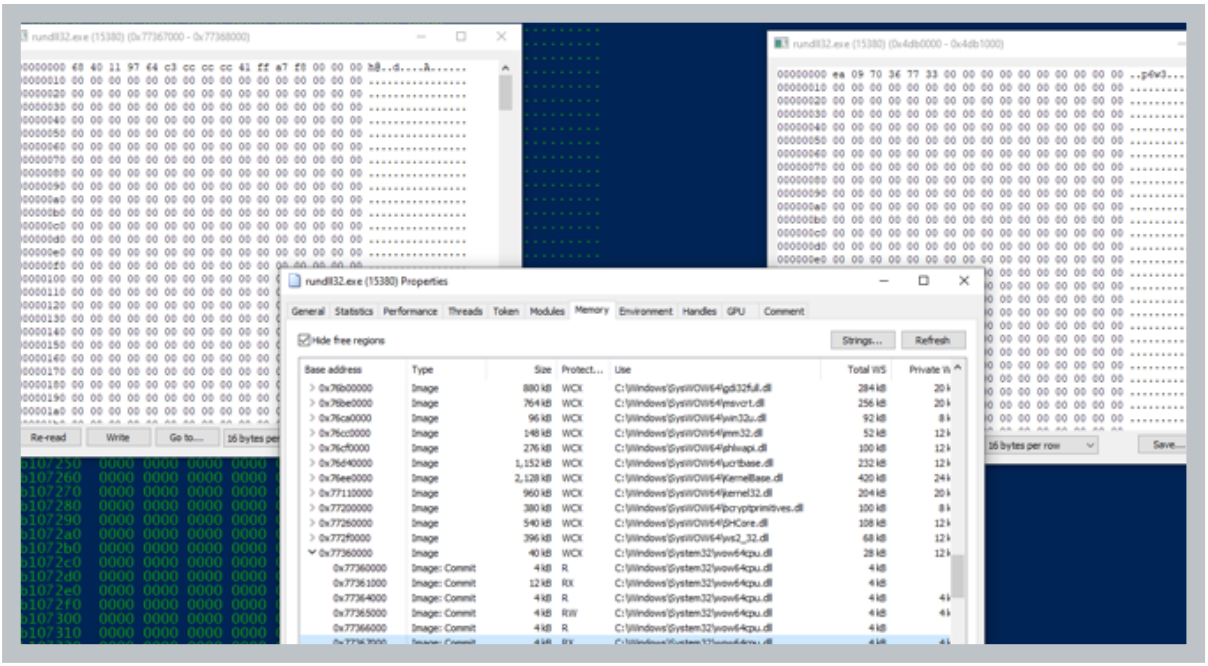
Image 15. – WOW and native ntdll.dll version loaded into x86 process



But, there is a caveat. The memory range of the x64 ntdll.dll instance, and its variables and arguments, are all of the wrong architectural format.

It's possible for Physical-Address-Extension (PAE) x86 processes to access x64 memory regions in certain hardware configurations. In the case of the native ntdll.dll module within WOW, its address are in x64 format and not run through PAE. In order to operate an x64 syscall, the program needs to switch CPU modes from x32 to x64 while the syscall is performed from the x32 ntdll.dll to native version. This action is achieved through an assembly "FAR JMP" between code sections 0033(x64) and 0023 (x32) to switch between CPU/memory modes. This JMP is conducted within the wow64cpu.dll module, typically at offset 0x7009.

Image 16. – A hijacked HeavensGate (left) and the copied trampoline function location (right)



A malicious user knows that all standard syscalls from an x86 process will have to pass through this far jump. By placing a hook on this gateway, the attacker effectively gains full control over all performed syscalls for that process from the WOW environment. As the far jump is only nine bytes, the hook can be altered, and the original replicated into a “trampoline” elsewhere with very little footprint. A trampoline is a section of code that is created solely to jump to another section of code, typically done in order to access a specific region, or return a control flow to a legitimate location specified dynamically to the trampolines code section. Once installed, syscalls can be neutered, manipulated, enumerated and perform any other activity an attacker requires.

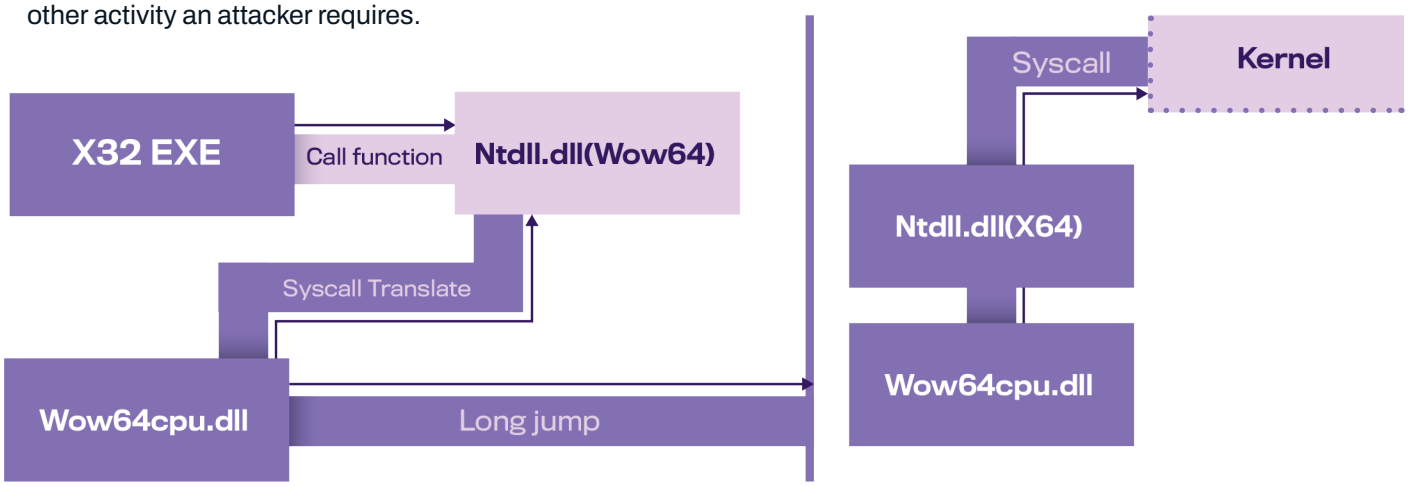


Image 17. – Overview of HeavensGate Syscall hook

How do we monitor for a sinful HeavensGate?

As with most hooking in native modules, the best way to determine if a hook has been deployed is to check for deviations from the known good values, just as with inline hooking detection. However, as the far jump operations has to go to a location relative to the native ntdll module which is assigned dynamically, the gates nine byte jump code always differs between live memory and what is on disk. The in-line hook comparison method won’t work this time around.

```
00000000 EA097010336B00          JMP 006B:33107009
00000007 0000                    ADD BYTE PTR [EAX],AL
```

Image 18. – On-disk value of HeavensGate

```
00000000 EA097048773300          JMP 0033:77487009
00000007 00                      ADD BYTE PTR [EAX],AL
```

Image 19. – Live dynamic value of HeavensGate

Let's work out how to determine the legitimate value of the gate to, then compare to the gate being used by every x86 process on the system. Another option discussed earlier, which may seem obvious, is to check for Copy-On-Write (COW) pages of the wow64cpu.dll module of process. Although it's possible to fine-tune to only monitor the page range in which the gate code resides, the page contains other functions which, if manipulated, could perform other malicious actions. It's unlikely, but possible, and as the gate code is only nine bytes, a bitwise comparison is much more accurate with very little resource consumption. As such, a targeted comparison has great value for monitoring this exploit over COW.

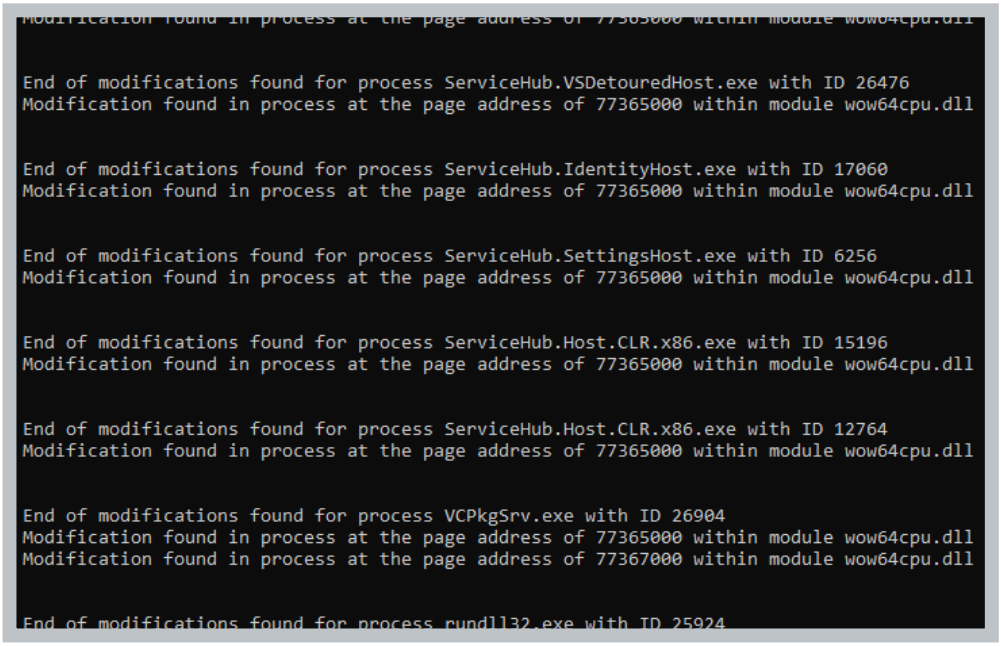


Image 20. – COW scanner for HeavensGate hook

The question remains: how do we determine a legitimate gate value that is dynamically assigned at the initial load of wow64cpu.dll when the first x32 process is loaded via WOW? The easiest solution is to use a value reserved for WOW processes within the Thread Environment Block (TEB). TEB analysis conducted by Geoff Chappell[4], and catalogued in their very useful website, shows that within the TEB at offset 0xC0 for x86 processes is a variable, with value "WOW32Reserver". In reality this reserved value is the "Fastsyscall" variable which is a pointer to the gate location of the current WOW environment used to accommodate syscall operations performed in-program. We can use this TEB value of any x86 process to find the gate memory location, and then by examining the content of a known good process, you can acquire a legitimate gate nine byte value.

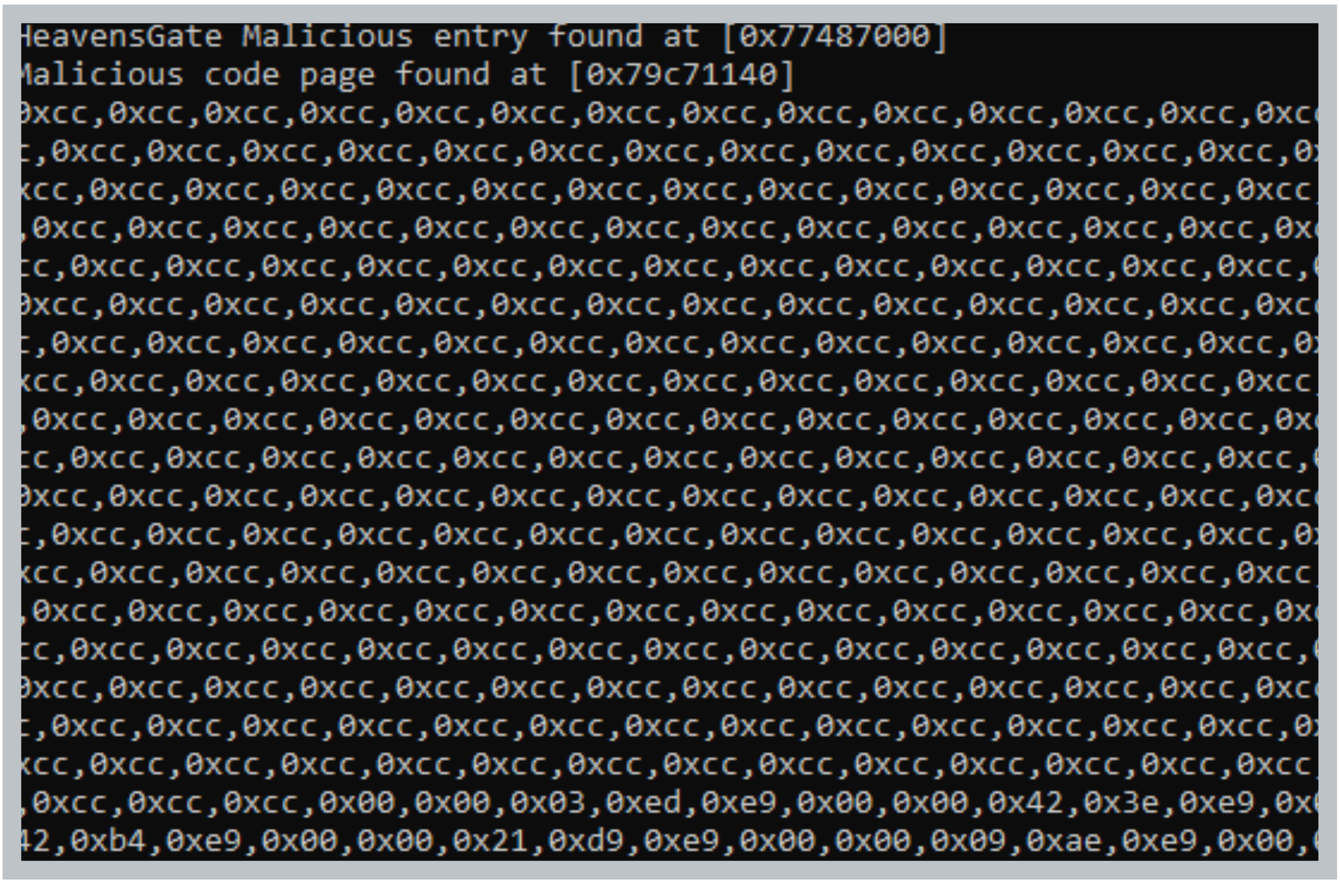
0xC0	0x0100	PVOID WOW32Reserved;	4.0 and higher	previously at 0x0708

Image 21. – TEB reserved FastSyscall value – Geoff Chappel [4]

It is worth noting at this point that wow64cpu, along with other WOW specific modules are loaded at the initialization phase of an x86 process. Similar to ntdll.dll, such processes are almost certain to share the loaded module rather than load a new instance. However, additional checks can be made to ensure wow64cpu.dll module load addresses are consistent across all scanned processes, or the TEB FastSyscall value can be extracted from each process for analysis, but this can present access issues.

Once the legitimate nine byte gate value is known, a comparison can be made for each process currently running, and any deviation flagged as being hooked. As an additional step, as the gate value is limited to nine bytes, the scanner can extract the malicious hook location and perform a dump of the hooked functions contents for analysis, or to be run through secondary detection engines.

Image 22. – Targetted 9 byte scan with Hook code content dump



CHAPTER 3.

Using Vectored Exception Handlers to side-step EDR

Vectored Exception Handlers (VEH) are an unframed exception handler mechanism introduced in Windows XP. They allow developers to override Structured Exception Handlers (SEH) within their developed code at a high level. Due to this priority in exception handling, researchers and malicious actors have devised methods to utilize this capability for circumventing intended command flow to bypass monitoring, circumvent integrity checks, or even execute malicious code.

What is a SEH and VEH exactly?

The most basic form of exception handling is SEH, which is frame-based. This is what most developers know as a try/catch block at its lowest level, with the try block making up a frame. However, each function is a frame. A running process is made up of nested frames, with each step on the stack making up a frame all the way outwards until the global frame is reached.

```
private static void test()
{
    try
    {
        PatchEtw(new byte[] { 0xc3 });
    } catch(InvalidOperationException e)
    {
        Console.WriteLine("Operation Error");
    }
}
```

Image 23. – SEH example within ETW in-line hook patch source

If you imagine in the code section above that the "PatchEtw" function throws an "InvalidOperationException" error, the catch block will handle the exception appropriately. However, what happens if the function throws an exception of a different type? In that case, the exception gets passed to the encapsuling frame, namely that of function "test()". Again, if there isn't appropriate exception handling for the calling of the "test()", it is handled there. If not, it escalates to the frame/function which is called a "test()" and so forth. If the exception gets escalated to the global frame, it typically causes the program to crash due to an unhandled exception error.

Now exception generation occurs at the CPU which is in turn handed off to the kernel. This won't go over all the elements involved in exception generation within the kernel, but will cover the bits that are important here. The kernel then generates a number of objects which include the "EnvironmentContext" and "ExceptionRecord" objects. These are created within the "TrapFrame" in the kernel space and then handed back to the user space via the function "KiUserExceptionHandler" within ntdll.dll via the EIP from the trap frame.

Once control is back in the user space and the exception information is with “KiUserExceptionHandler”, it will then call “RtlDispatchException”, which uses the information from the EnvironmentContext and ExceptionRecord to identify which frame the exception happened within. The function then scans the designated frame for appropriate handlers and if not found, checks all containing frames outwards. This is the general concept of how SEH works.

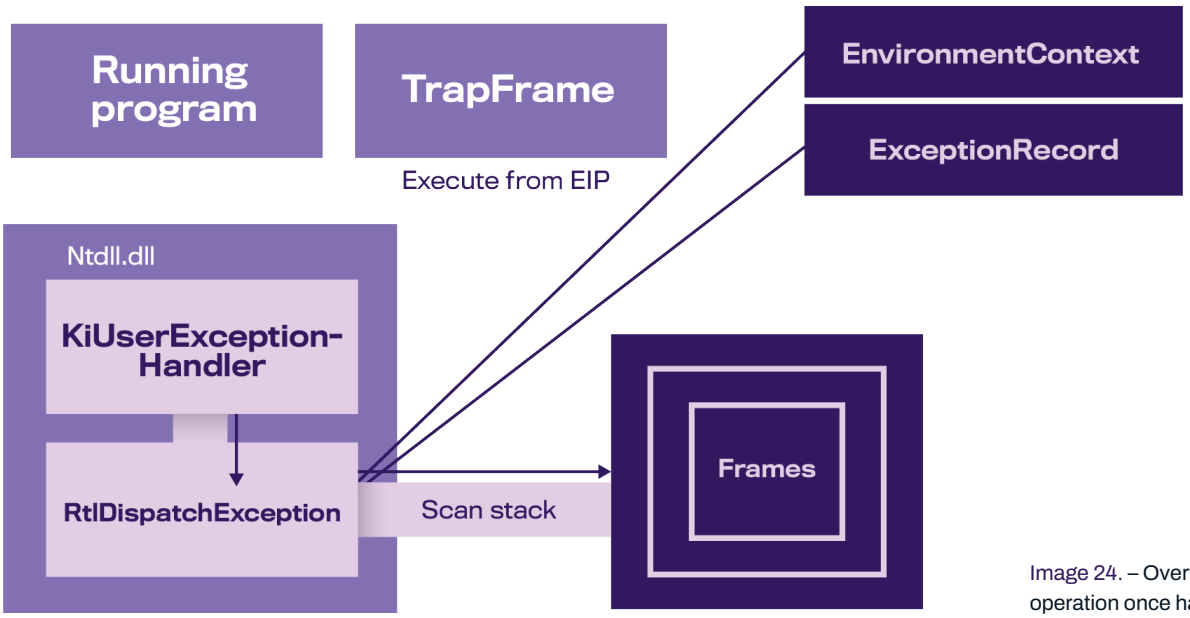


Image 24. – Overview of SEH operation once handed back to user space

So what about VEH? How is that different? The primary difference is that VEH does not take frames into account whatsoever. Whereas with SEH, it is all about the frames, and the containing frames, VEH is “frameless”. As such, it handles exceptions depending on the order in which they were added to the “Vectored_Handled_List”(VHL), which is stored within ntdll.dll module memory space. When VEH are defined, they are added to the VHL either at the beginning or the end of the list. When an exception is encountered, the OS will check the list in a linear sequence from first to last.

The VEH in the list which handles the exception that is encountered first, handles the exception regardless as to whether there is another VEH with a handler for that exception type or not. Significantly, this is done before SEH are checked and that SEH are only checked if no VEH entries for that exception type are found. Microsoft did this by altering the control flow within “KiUserExceptionHandler” by inserting a “RtlCall-VectoredExceptionHandlers” function call just before “RtlDispatchException” is called, and then, depending on the result, skipping the second function altogether.

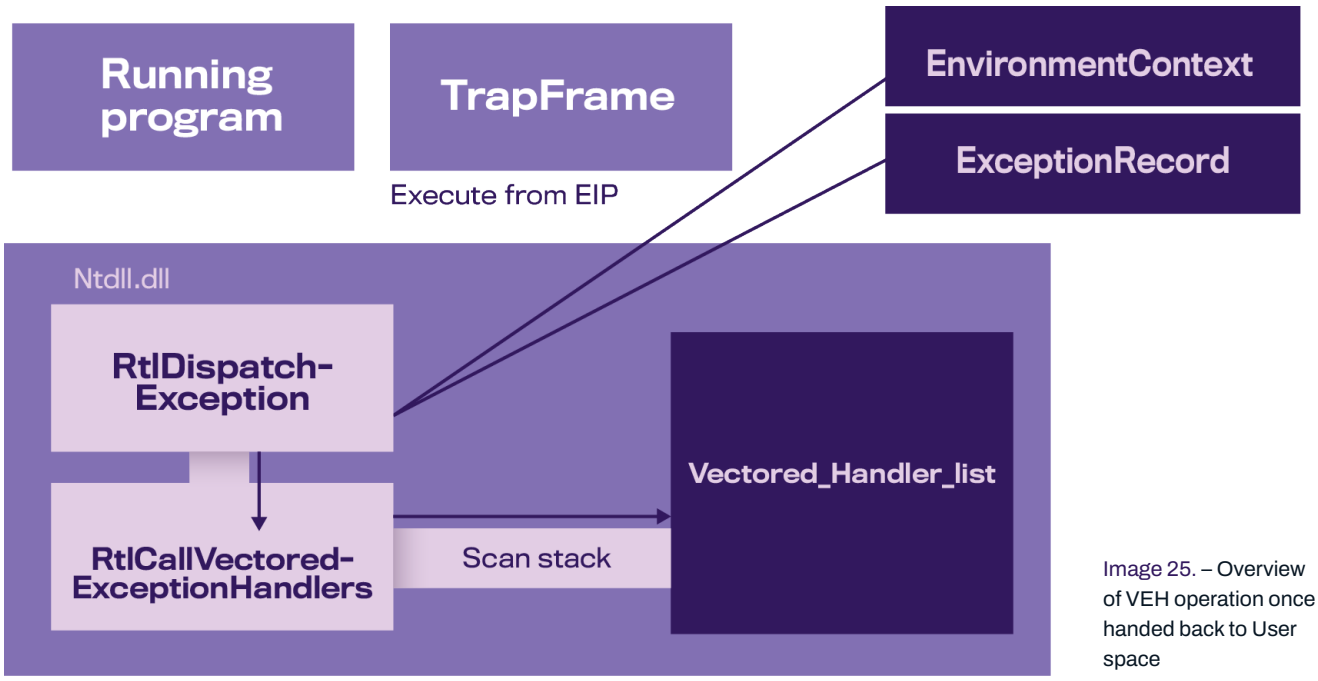


Image 25. – Overview of VEH operation once handed back to User space

The last thing to note is that VEH can have multiple exceptions defined within them as comparison statements against the contents of the generated exception information. As such, one VEH entry can handle one or hundreds of event types with no limitations.

VEH always cutting the line to misbehave

Because VEH entries will always take priority in exception handling, we will cover some of the ways in which malicious actors are abusing this capability. These range from preventing integrity checks by side stepping GuardPage(GP), performing force jumps to malicious code, silently bypassing internal functions, and finally, selectively avoiding EDR monitoring capabilities.

GuardPage Integry bypassing

GP is a memory integrity check capability which allows for an exception to be thrown if a memory page is altered. By placing GP on a memory region, a program is able to detect malicious/unauthorized manipulation of its running memory, and behave in a reactive manner by handling the generated exception.

As mentioned, GP relies on the concept that the exception "STATUS_GUARD_PAGE_VIOLATION" is handled by the program to protect itself. However, since this works on exception handling, VEH can be used to circumvent the operation. An IT specialist going by the handle SH3N[11] demonstrated hooking via this method by injecting into a program with GP enabled, and then installing a custom VEH entry for the GP violation exception. In their example, which applies to any GP bypass, their injected function defined a hooking function and its memory space, adding a malicious VEH and triggering the GP violation on a known protected memory region.

The magic happens in the malicious VEH entry, which has two exception types defined for capture- "STATUS_GUARD_PAGE_VIOLATION" and "STATUS_SINGLE_STEP"- the latter is typically used for debugging purposes, but can be forcefully triggered by modifying an ExceptionInfo object's Eflags object. When the GP violation occurs, the first handler is entered. This modifies the EIP/RIP to the location of the hook code section. Following that, the Eflags is modified to value 0x100. This flag value effectively tells the kernel this is a debugged instance to trigger a "STATUS_SINGLE_STEP" exception. Once this is set, the control of the program is returned to the injected process.

```
LONG WINAPI Handler(EXCEPTION_POINTERS* pExceptionInfo)
{

    if (pExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_GUARD_PAGE_VIOLATION) //We will catch PAGE_GUARD Violation
    {
        if (pExceptionInfo->ContextRecord->XIP == (DWORD)og_fun)
        {
            pExceptionInfo->ContextRecord->XIP = (DWORD)hk_fun;
        }

        pExceptionInfo->ContextRecord->EFlags |= 0x100;
        return EXCEPTION_CONTINUE_EXECUTION;
    }

    if (pExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_SINGLE_STEP)
    {
        //uint32_t dwOld;
        //dwOld = Controller->VirtualProtect((DWORD)og_fun, 1, PAGE_EXECUTE_READ | PAGE_GUARD);

        DWORD dwOld;
        auto addr = (PVOID)og_fun;
        auto size = (SIZE_T)((int)1);
        NTSTATUS res = makesyscall<NTSTATUS>(0x50, 0x00, 0x00, 0x00, "RtlInterlockedCompareExchange64", 0x170, 0xC2, 0x14, 0x00)
            (GetCurrentProcess(), &addr, &size, PAGE_EXECUTE_READ | PAGE_GUARD, &dwOld);

        return EXCEPTION_CONTINUE_EXECUTION;
    }

    return EXCEPTION_CONTINUE_SEARCH;
}
```

Image 26. – GuardPage bypassing to run hooked code content as part of exception – SH3N[12]

[11] https://mark.rxmsolutions.com/hook-via-vectored-exception-handling/

[12] https://mark.rxmsolutions.com/hook-via-vectored-exception-handling/

Since the single step exception flag was set, we immediately generated an exception which corresponds to the second entry in the VEH. Once in this block, the attacker can now recall the original function with GP reset. This means the hook has now executed and the original function is now executing, or the attacker can perform any modifications they want to the protected memory section, and then re-assert the GP status of the page at the end before the exception is returned. In the latter example, the malicious VEH entry fully circumvents GP, as any alteration can be made, and then the protection status reasserted without any alerting or preventative action taking place, as the OS deems the exception as properly handled.

Force Jump

One of the quirks of VEH is that the function to add new VEH to the process "AddVectoredExceptionHandler" within kernel32.dll only requires two arguments: a switch on whether to go at the front or end of the VHL, and a pointer to the defined VEH structure. If the VEH is added to the start of the VHL, the pointer will be called at the first exception handled, whether the pointers memory section has VEH handler code within it or not.

If the pointer provided to "AddVectoredExceptionHandler" instead points to a memory section filled with shellcode, that shellcode will be executed at the first exception that is encountered. As long as the memory section ends with the return value of "EXCEPTION_CONTINUE_SEARCH", the OS will continue without issue, and continue searching the VEH list and SEH for an appropriate exception handler, and the shellcode would have been successfully executed.

Image 27. – MSDN documentation for AddVetoredExceptionHandler[13]

AddVectoredExceptionHandler function (errhandlingapi.h)

Article • 04/02/2021 • 2 minutes to read

Is this page helpful?

Registers a vectored exception handler.

Syntax

C++ Copy

```
PVOID AddVectoredExceptionHandler(
  ULONG                       First,
  PVECTORED_EXCEPTION_HANDLER Handler
);
```

Parameters

`First`

The order in which the handler should be called. If the parameter is nonzero, the handler is the first handler to be called. If the parameter is zero, the handler is the last handler to be called.

`Handler`

A pointer to the handler to be called. For more information, see VectoredHandler.

[13] https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-addvectoredexceptionhandler

Silent bypassing

As pointed out in the GP bypassing mechanism, by injecting a VEH into a process or as part of a malicious process, you can effectively neuter exception-based events as you see fit. If applied on a broader scale, this means that any function that is executed, as long as an exception can be forced as part of operation, an attacker can use VEH to omit any monitoring or subsequent activity.

In this use-case it is assumed that an exception can be forced at a stage in the function that would still permit the primary purpose, but prevent the unwanted activity via exception generation. One method to do this would be to identify functions of use and determine the point at which monitoring events take place. By installing a command to trigger a divide by zero exception just before its operation, a VEH entry could be used to manipulate the control flow and prevent logging. Although it is possible, this would be a difficult use-case to implement.

EDR targetted bypass – Firewalker

FireWalker (FW) was a PoC tool developed by MDSec in 2020[14], which leveraged VEH in order to bypass certain EDR products, which then utilized hooked functions for telemetry acquisition. Essentially, this tool scans for hooks or pointer duplication in popularly monitored module functions of a running process, and if found, utilizes a similar bypass mechanism as outlined for GP.

The tool scanning capabilities are fairly complex and was explained in detail within MDSecs own material, so we will not cover the internal scanner elements and scanning/duplication that are used to affect the step over via locating the hooked and circle back jmp commands. The key point is how FW prevents the hook from executing and allows for the bypass to take place. To that end, it relies on VEH.

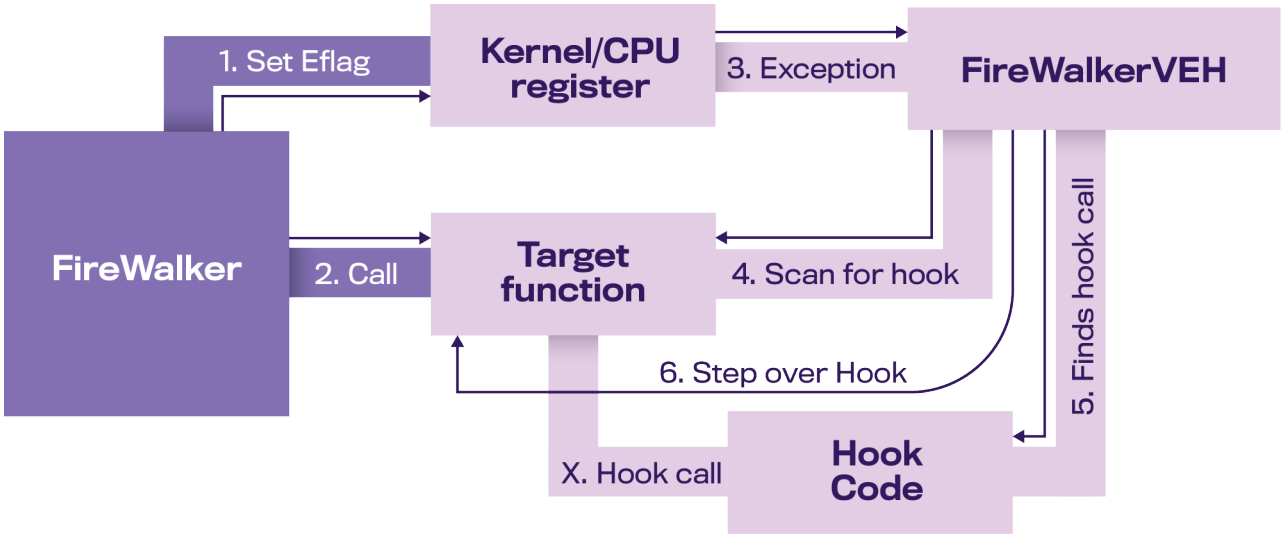


Image 28. – Overview of FireWalker operation

[14] https://www.mdsec.co.uk/2020/08/firewalker-a-new-approach-to-generically-bypass-user-space-edr-hooking/

To effect the scanning capability, FW implements "EXCEPTION_SINGLE_STEP" (which is the same as "STATUS_SINGLE_STEP") just before calling the desired function in order to effect a trap block. By manipulating the Eflags register (as with GP), they force the following command execution to create a single step exception, which is subsequently caught by their VEH entry. In the case of FW the exception then checks the next command (EIP + 1) for whether this matches to a known call or jmp opcode format. If it does, then the scan to resolve the hooks return address is made. If not, the function continues uninterrupted. If the scan does find the hook code and the return address within the hook, it modifies the EIP to point to that location before returning from the exception, meaning that the hook is effectively stepped over.

Detecting VEH usage

In order to detect the use of VEH and monitor its use and contents, there is a primary component that needs to be resolved: the VHL. The VHL contains the current list of VEH entries for a process, the contents of which can only be decoded with the process cookie for the target process.

The most evident and prominent work on VEH enumeration was conducted by Dmitri Fourny[15], who outlined where the VHL is stored and how it can be acquired. In addition, Ollie Whitehouse, over at the NCCGroup, has done some amazing work on optimizing Dmitri's work for modern systems and ironing out some deviations that have occurred during subsequent OS revisions[16]. Ollie has developed a PoC scanner code as part of their DetectWindowsCopyOnWriteForAPI[17] repository, which is what we based our testing and adaptations on.

The VHL list is stored in the native ntdll.dll module memory instance and is referenced by the RtlDispatchException function during operation. In order to do this, the function obviously needs to know the location of the VHL. Dmitri found that the pointer for the VHL is stored as "LdrpVectoredHandlerList" and that it can be heuristically scanned within specific functions. By scanning the ntdll.dll for specific functions and then pattern matching the code sections, we can extract the VHL pointer. Three of the known functions to use the VHL pointer are "RtlpCallVectoredHandlers", "RtlAddVectoredExceptionHandler", and "RtlRemoveVectoredExceptionHandler", all of which you can assume directly interact with the VHL. The signature scanner in our method is "0x4c 0x8d 0x25". Then for the pointer, copy 'function address + signature offset +3' for 4 bytes, and you will have acquired the VHL.

```
ULONGLONG GetVEHOffset() {
	HMODULE ntdll = LoadLibraryA("ntdll.dll");

	ULONGLONG procAddress = (ULONGLONG)GetProcAddress(ntdll, "RtlRemoveVectoredExceptionHandler");
	BYTE* Buffer = (BYTE*)(GetProcAddress(ntdll, "RtlRemoveVectoredExceptionHandler"));

	//fwprintf(stdout, _TEXT("[i] RtlRemoveVectoredExceptionHandler [%llx]\n"), (procAddress));


	DWORD dwCount = 0;
	DWORD dwOffset = 0;
	for (dwCount = 0; dwCount < 60; dwCount++) {

		if ((*(Buffer + dwCount) == 0x4c) && (*(Buffer + dwCount + 1) == 0x8d) && (*(Buffer + dwCount + 2) == 0x25)) {
			memcpy(&dwOffset, (Buffer + dwCount + 3), 4);
			break;
		}
	}

	// ptr return by GetProcAddress + the seek until our pattern + the instruction to load the RVA
	//fwprintf(stdout, _TEXT("[i] LdrpVectorHandlerList [%llx]\n"), ((LONGLONG)Buffer + dwCount + 7 + dwOffset));

	return ((LONGLONG)Buffer + dwCount + 7 + dwOffset);
}
```

Image 29. –Acquisiton of VHL pointer [18]

[15] https://dimitrifourny.github.io/2020/06/11/dumping-veh-win10.html

[16] https://research.nccgroup.com/2022/03/01/detecting-anomalous-vectored-exception-handlers-on-windows/

[17] https://github.com/nccgroup/DetectWindowsCopyOnWriteForAPI/tree/master/d-vehimplant

[18] https://research.nccgroup.com/2022/03/01/detecting-anomalous-vectored-exception-handlers-on-windows/

In order to decode the VEH entry, you will need to decode the pointer in the VHL. To do this, you use “NtQuery-InformationProcess” to acquire the process cookie, and then perform “RotateRight64” operations on the pointer provided. After the rotation, it will return the memory address for where the VEH code has been stored, which is checked via the exception handler process. Once we have the handler code address, we can perform any number of analytics against the handler code, such as heuristics, handler location assessment, or even the exception type being used.

As a basic example, checking the exception types being caught can allow for some quick wins in this area. By scanning the VEH block for exception handler byte sequences outlined in winnt.dll, we can isolate the exception handling blocks. Once in the block, we can then scan for comparison opcode sections and check their compared value for corresponding constant exception type values, as outlined in winnt.h. In this way, we can scan the VEH block and identify all the exception types the VEH has been crafted to handle, and perform program to exception type comparisons.

Image 30. – VEH enumerator adapted for exception type scanning

```
[VCTIP.EXE] is using VEH - Vectored Exception Handler
[VCTIP.EXE] VEH handler(decoded) 0x3FFC825F8EF8BCAF which is in UNKNOWN
[VCTIP.EXE] # of VEH: 1
[vehDemo.exe] is using VEH - Vectored Exception Handler
[vehDemo.exe] VEH handler(decoded) 0x00007FF7EF1913B1 which is in vehDemo.exe
ion type found in process [16836] is the exception [STATUS_SINGLE_STEP]
[msvsmon.exe] is using VEH - Vectored Exception Handler
[msvsmon.exe] VEH handler(decoded) 0x00007FF8B5A9AD80 which is in clr.dll
ion type found in process [18116] is the exception [DBG_PRINTEXCEPTION_C]
ion type found in process [18116] is the exception [MS_VC_EXCEPTION_THREAD_NAMING]
```

For example, EXCEPTION_SINGLE_STEP is typically only used in debugging programs and platforms, and therefore its presence in arbitrary or sensitive processes can be an easy indicator of malicious VEH usage, such as in the case of FW. As all security bypass techniques rely on being able to trigger an immediate exception, they use the EXCEPTION_SINGLE_STEP, which is, otherwise rarely encountered outside of development environments, making it an easy detection criteria. Alternatively, exception types of STATUS_INTEGER_DIVIDE_BY_ZERO are rare at a VEH level, and can be used to determine anomalous VEH entries, if not outright malicious.

During experimentation using this detection criteria for exception types, only a small set of alerts were generated with a very small subset being false positives, which were easily attributed by the nature of the process (Visual Studio development environment debug flags).

The problem with x86 and VEH

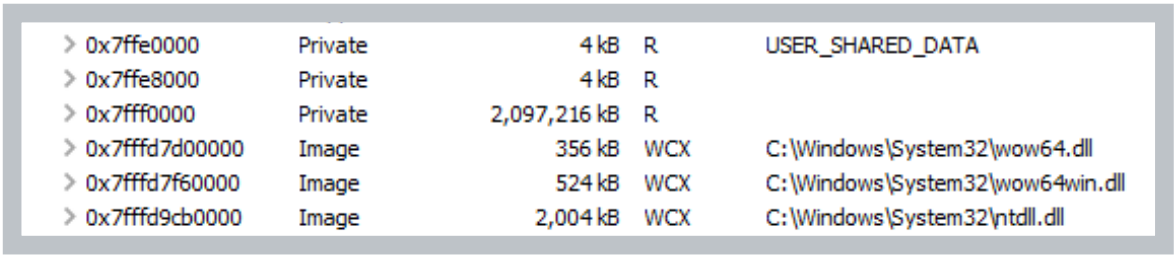
A problem alluded to by Ollie's work is that the scanner developed, only works on x64. The results of x86 processes always come out as UNKNOWN for the module resolution, and the VEH handle is unresolvable. After extensive analysis, the reason for this became clear; the VHL for the entire system including the WOW environment is stored in the native ntdll.dll module memory instance. Because of this, the memory address is in x64 format, which when attempted to be resolved against an x86 process, does not work. That process, can only work up to the x32 memory range. Yet the VEH for x86 does work during exception handling.

How can this be possible? The reason is the exception is handled in the x64 CPU mode before being returned to the x32 CPU mode in WOW. Therefore, the physical x64 address that the VEH resolves to can be accessed by the exception handler in x64 mode with the correct process permissions, and then return the results back to the calling process in WOW. However, from a scanner perspective, this throws up a huge problem. A scanner running in WOW x86 cannot directly access the native ntdll.dll for the VHL list without switching to the x64 CPU mode. And a x64 scanner instance cannot use a WOW x86 process handle to resolve an x64 physical address without getting a memory access violation.

Image 31. – Access violation error when resolving x86 process VEH pointer

Name	Value	Type
buf	0x0000003b08f105f0 "lllllllllllllllllllllllllllllllllllllllllllllllllllllllll...	char[0x00001000]
cProcess	0x0000003b08f1a830 L"vehDemo.exe"	wchar_t *
dwPID	0x00003dcc	unsigned long
dwRead	0x0000cb34767db24b	unsigned __int64
error	0x000003e6	unsigned long
handlerAddress	0xfff678bdfbb79ec9	void *
hProcess	0x0000000000000088	void *
myfile	{_Filebuffer={_Pcvt=0xcccccccccccccccc {...} _Mychar=0xcc 'Ì' _Wr...	std::basic_ofstream...
pageSize	0x00001000	unsigned long
result	0x00000000	int
sysInfo	{dwOemId=0x00000009 wProcessorArchitecture=0x0009 wReserve...	_SYSTEM_INFO

The memory access violation occurs because the x64 memory region VAD (Virtual Address Descriptor) for a WOW x86 process is restricted by the kernel. As x32 processes have a limit of 4GB of memory (with large address flag enabled) this is done to maintain stability in the system. As the x64 native system will assign the WOW x86 process, a virtual address range in the x64 physical address capacity, wherever it best fits arbitrary memory access outside the defined virtual range, can be problematic. Therefore, the handle for the WOW x86 process will always result in an access violation when trying to resolve its own VEH handle stored in an x64 address in an access violation when trying to resolve its own VEH handle stored in an x64 address.



0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA
0x7ffe8000	Private	4 kB	R	
0x7fff0000	Private	2,097,216 kB	R	
0x7fffd7d00000	Image	356 kB	WCX	C:\Windows\System32\wow64.dll
0x7fffd7f60000	Image	524 kB	WCX	C:\Windows\System32\wow64win.dll
0x7fffd9cb0000	Image	2,004 kB	WCX	C:\Windows\System32\ntdll.dll

Image 32. – VAD installed for Kernel restricted memory region in x64 range at 0x7FFF0000

This presents only two possible solutions. Either we manually switch between CPU modes within our scanner, or we resolve the physical address to a virtual address and resolve it via the WOW x86 process handle. Many researchers and attackers use manual CPU mode switching in order to achieve arbitrary x32 or x64 command execution on a system which supports both architectures (through WOW). This requires the assembly (ASM) of Heavens Gate (which we discussed earlier) to be available to the scanner code. By using a custom ASM gate, a program can jump between CPU modes arbitrarily. However, most research into this area highlights the potential instability in using such operations, as this is not intended by the OS. As such, while manual CPU switching is an option, it may not be overly stable for monitoring solution deployment unless great care is taken.

The alternative revolves around utilizing elements of the Memory Management Routine (MMR) component of the OS kernel. There is a function within the MMR called “MmGetVirtualForPhsyical”, which resolves a physical address to its virtual counterpart by tracing it back through the kernel-stored page tables. This would allow the virtual address assigned to the x64 physical address to be acquired (which should be in x32 format), and used via the WOW x86 process handler for the VEH resolution.

```
PVOID
MmGetVirtualForPhysical(
      __in PHYSICAL_ADDRESS PhysicalAddress
  );

Routine Description:

    This function returns the corresponding virtual address for a physical
    address whose primary virtual address is in system space.

Arguments:

    PhysicalAddress - Supplies the physical address for which to return the
                      virtual address.

Return Value:

    Returns the corresponding virtual address.

Environment:

    Kernel mode.  Any IRQL level.
```

Image 33. – MmGetVirtualForPhysical MMR Kernel function details[19]

Another method devised by the hacker Xerox, is that if MMR access can be gained to get a snapshot of the page-tables, a user land version of the information can be updated using system event monitoring. As such, only one such access event would be required. But this, as a prime example, also presents a problem for creating access to the MMR in that it can open up several security concerns caused by such user to kernel-level interconnectivity. Access to the MMR is heavily restricted for good reason, as manipulation of the page tables, or access to its high-level functions, can be easily abused. Purposefully introducing such access may be an unacceptable risk.

The conclusion is that monitoring WOW x86 processes from an x64 scanner is not simple and the available solutions are all prone to instability or security issues. Although this type of monitoring is possible, it has to be determined by the developer of such a scanner whether the risks outweigh the monitoring potential.

[19] http://web.archive.org/web/20220926121708/https://www.codewarrior.cn/ntdoc/wrk/mm/MmGetVirtualForPhysical.htm

Summary

MMT's are wide and varied in their complexity and in the mechanisms they choose to target in order to achieve their desired alterations. In all the techniques presented here, they can be used on both targeted/hijacked processes as well as malicious payloads. Used for either malicious operation or to disguise and evade detection for adjacent malicious operations, MMT's are a tried-and-true mechanism to achieve these ends, and consequentially, be continuously analyzed.

Security vendors and researchers use a wide range of methods to monitor and detect malicious operations. But as FW demonstrated, even those can be directly targeted by MMT's. If not, MMT's can be used to dodge and side-step otherwise revealing operations that attackers may use. Yet, in order to stay ahead of the defensive operators, the development of new MMT's must not stop either. As the security field has matured over the last decade, so has its capacity to proactively seek out new offensive techniques or vulnerabilities, and devise monitoring solutions or ethically disclose such weaknesses to the vendors.

Therefore, although the use of MMT's will in all likelihood be a never-ending arms race, the use of tried-and-true mechanisms like those described in this document are also likely to be around for a long time. With slight modifications or a razor-sharp targeted design, such mechanisms can still prove effective in certain situations. As such, defenders must be ever vigilant and roll with the punches.

Who We Are

WithSecure™, formerly F-Secure Business, is cyber security's reliable partner. IT service providers, MSSPs and businesses – along with the largest financial institutions, manufacturers, and thousands of the world's most advanced communications and technology providers – trust us for outcome-based cyber security that protects and enables their operations. Our AI-driven protection secures endpoints and cloud collaboration, and our intelligent detection and response are powered by experts who identify business risks by proactively hunting for threats and confronting live attacks. Our consultants partner with enterprises and tech challengers to build resilience through evidence-based security advice. With more than 30 years of experience in building technology that meets business objectives, we've built our portfolio to grow with our partners through flexible commercial models.

WithSecure™ Corporation was founded in 1988, and is listed on NASDAQ OMX Helsinki Ltd.