# Defeating EDR Evading Malware with Memory Forensics

Andrew Case, Austin Sellers, David McDonald, Gustavo Moreira, Golden G. Richard III

acase@volexity.com, golden@cct.lsu.edu

June 10, 2024

## 1 Introduction

Endpoint detection and response (EDR) software has gained significant popularity and market share due to its ability to examine system state for signs of malware and attacker activity well beyond what traditional anti-virus software is capable of detecting. This deep inspection capability of EDRs has led to an arms race between EDR vendors and malware developers as the latter wants to evade the detection of EDRs while still achieving desired goals, such as code injection, lateral movement, and credential theft. This back and forth occurs in the lowest levels of hardware and software, including call stack frames, exception handlers, system calls, and manipulation of native instructions in the address spaces of processes. Given this reality, EDRs are often limited in how much lower they can operate to maintain an advantage over malicious code. The success of these bypasses has led to their use in many recent, high-profile attacks believed to be conducted by nation-state-backed actors as well as by prolific ransomware groups [1, 2, 3, 4, 5, 6]. There has also been significant ongoing industry research into new forms of bypasses, including works presented at recent Black Hat events [7, 8].

In this paper, we discuss our team's research effort that led to the development of new memory forensics techniques for the direct detection of the bypasses that malware uses to evade EDR's inspection of process activity. This discussion includes our research methodology, goals, and motivation, testing environment, developed capabilities, and insights learned during the project. Memory forensics tools reconstruct system state without reliance on operating system APIs and can fully examine this state without interference. Our techniques utilize this advantage to detect all EDR evasion techniques found in real-world malware as well as open-source toolkits and research. These capabilities were developed as new plugins to the open-source Volatility memory analysis framework, version 3, and will be released upon publication of this paper and presentation at

the conference. Given the threat posed by undetected malware across every organization, we believe that our research will provide significant value to incident response handlers, threat hunters, detection engineers, and other technical staff in defensive roles.

# 2 Research and Experimental Setup

## 2.1 Operating Systems and Versions Tested

We sought to develop capabilities that covered the Windows versions most commonly in use as well as mostly commonly targeted by EDR-evading malware. A review of operating system versions supported by the major EDR vendors showed that they generally support Windows 7 through Windows 11, including their server equivalents, and these versions match what our team generally encounters during our incident response engagements.

The following table lists the starting and ending versions tested and supported for each operating system:

| **Operating System** | **Earliest Version** | **Latest Version** |
|---|---|---|
| Windows 11 | 22000.282 | 26063.1 |
| Windows 10 | 10563 | 19045.2546 |
| Windows 7 | SP1 | Final Release |

This extensive range of versions covers Windows 10 starting from build 10563, released in 2017, to the latest release at the time this paper was written. These versions also cover all Windows 11 versions released through February 2024, including the latest insider releases. In total, beyond the specific samples made for testing during this project, discussed next, we used over 300 Windows memory samples from our test bed to ensure that we did not trigger false positives across default Windows installs plus a wide variety of benign and malicious third-party applications.

## 2.2 Memory Sample Testbed Creation

To ensure that our developed algorithms and Volatility plugins were accurate, we created many memory samples infected with malware that used the bypass techniques discussed throughout this paper. To create stable and valid memory samples, two methods for acquisition were used. The first was the use of Surge Collect Pro from Volexity [9]. This allowed us to collect from physical systems as well as automatically acquire files from disk, such as *ntdll.dll*, which we used for testing automation.

The second approach used for acquisition was snapshotting and suspending the VMware virtual machines that we used for testing. The system state files (*.vmem*, *.vmss*, *.vmsn*) created when snapshotting and/or suspending a guest contain a copy of all physical memory as well the metadata needed to perform

memory analysis. We used virtual machines when testing live malware samples from the wild and when we wanted to automate collection and analysis testing over many different Windows versions.

### 2.3 Analysis Tools and Resources

Completion of this research involved a mix of open-source code analysis as well as binary analysis of closed-source operating system components and malware samples. IDA Pro was used for all binary analysis during our research. We also created a variety of scripts to perform automated static analysis to verify several of our developed plugins. This automation was performed through Capstone's Python bindings [10].

## 3 Evasion Techniques

### 3.1 Background

#### 3.1.1 System Calls

A critical method that EDRs use to monitor process activity is monitoring the set of system calls that are often abused for malicious purposes. System calls are the boundary between process memory and the kernel (the operating system in memory), and access to any protected resource, such as another process' memory, the file system, and the registry, must be requested through a system call invocation. The kernel can then enforce security policies at the system call level, and, if access is denied, a process cannot access the requested resource except via some exploitation technique, such as privilege escalation.

The following figure shows this boundary crossing in action through using Process Explorer to monitor file access through the *CreateFile* system call.

| Frame | Module | Location |
|---|---|---|
| K 0 | FLTMGR.SYS | FltDecodeParameters + 0x210b |
| K 1 | FLTMGR.SYS | FltDecodeParameters + 0x1bba |
| K 2 | FLTMGR.SYS | FltAddOpenReparseEntry + 0x560 |
| K 3 | ntoskrnl.exe | IofCallDriver + 0x55 |
| K 4 | ntoskrnl.exe | IoGetAttachedDevice + 0x54 |
| K 5 | ntoskrnl.exe | SeCaptureSubjectContextEx + 0x134b |
| K 6 | ntoskrnl.exe | ObReferenceObjectByHandle + 0x4477 |
| K 7 | ntoskrnl.exe | ObOpenObjectByNameEx + 0x1fa |
| K 8 | ntoskrnl.exe | PsImpersonateClient + 0x18ab |
| K 9 | ntoskrnl.exe | NtCreateFile + 0x79 |
| K 10 | ntoskrnl.exe | setjmpex + 0x83f8 |
| U 11 | ntdll.dll | ZwCreateFile + 0x14 |
| U 12 | KERNELBASE.dll | CreateFileW + 0x5f9 |
| U 13 | KERNELBASE.dll | CreateFileW + 0x66 |
| U 14 | wofutil.dll | WofIsExternalFile + 0x63 |
| U 15 | msedge.dll | AugLoop_BinaryDataDownloadParams:... |
| U 16 | msedge.dll | GetHandleVerifier + 0x1aa62cf |

Figure 1: Monitoring *CreateFile*

Reading in reverse order, the system call trace starts inside of *msedge.dll* (Frame 15-16) and then eventually ends up in *KERNELBASE.dLL* followed by *ntdll.dll*. The blue *U* to start these rows signifies they are occurring in userland, which means process memory. Starting with Frame 10, the kernel is entered and we can see that *ZwCreateFile* in userland eventually reaches *NtCreateFile* in the kernel. The Windows convention is that the userland handler of system calls are named with *Zw** whereas the matching *Nt** version is the kernel handler.

The following picture shows the *CreateFile* system call handler inside of *ntdll.dll*.

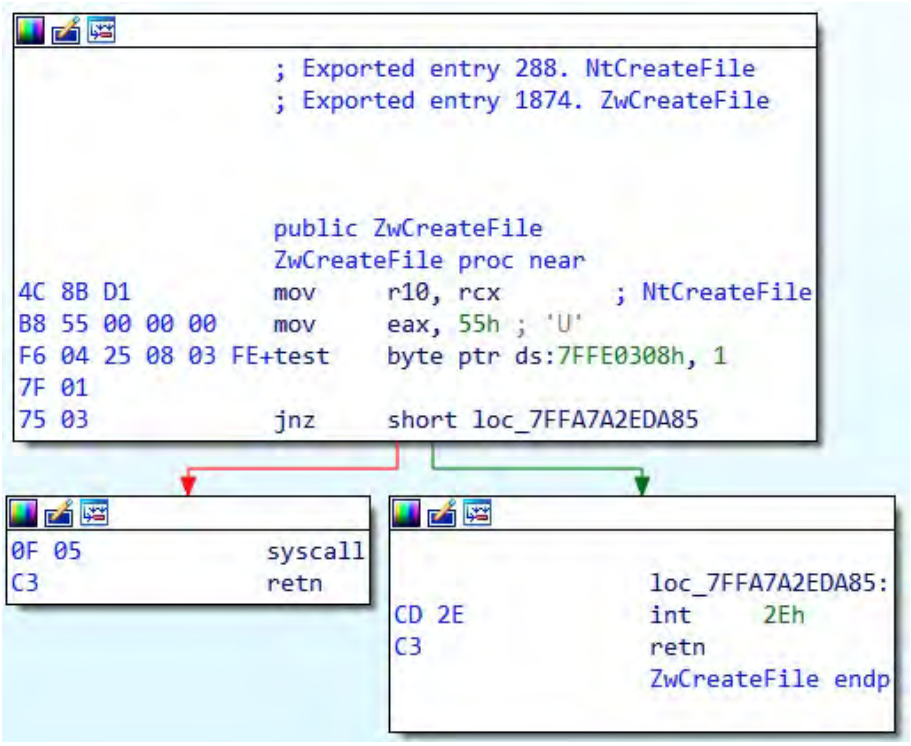



Figure 2: The *CreateFile* System Call

As described by IDA and is expected, *ZwCreateFile* and *NtCreateFile* are the same function. Understanding the instructions of this function will be essential to understanding the EDR bypasses discussed later, so we describe them now. To start, the *rcx* register is copied into *r10*. In the Windows calling convention, *rcx* is used to store the first parameter to function calls, while *r10* is used as the first parameter to system calls. All other registers maintain their order between system calls and function calls. Next, *55h* is placed into the *eax* register. This is the system call entry number, usually referred to as the system service number (SSN) on Windows. To execute a system call, the SSN of the function must be known for the particular version of Windows on which the code is executing

as the numbers change between versions. After this, the system determines if the syscall instruction can be used or if the older *int 2E* interface must be used. Both of these paths lead to the kernel being entered to handle the system call.

### 3.1.2 EDR System Call Monitoring

To monitor a wide range of activity across the system, EDRs *hook* system call handlers inside of monitored processes to gain control when processes make system calls, including the ability to inspect the parameters of the system call and deny execution if needed. For example, if a malicious system call is detected, such as the creation of a thread within a memory region that is both writable and executable (RWX), the EDR can deny the system call and prevent the malware from launching its payload. This effectively stops the malware in its tracks and allows the EDR to generate an alert of the precise activity of the malware. Malware obviously wants to avoid such detection while still using the system calls necessary to achieve desired goals. To meet this goal, modern malware deploys a number of evasion techniques aimed at disrupting an EDR's ability to monitor system calls.

Automatically detecting these evasion techniques through memory forensics was the goal of our research. We will now begin our detailed discussion of each EDR evasion technique as well as our developed detection algorithms. Each section will include background on the technique, the operating system internals necessary to understand and detect the technique, and a demonstration of our new Volatility plugins' detection capabilities.

## 3.2 Module Unhooking

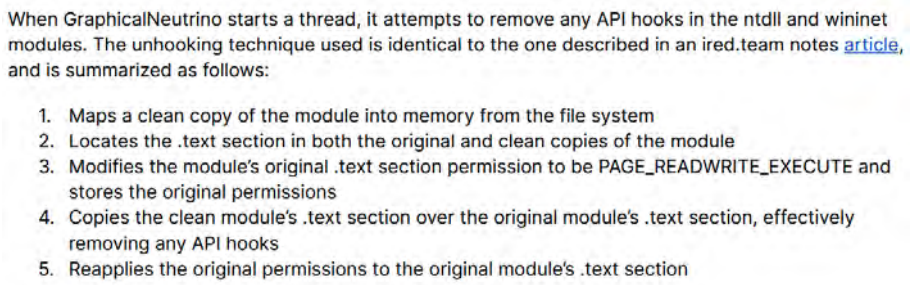
### 3.2.1 Background

The first technique we will discuss is module unhooking, which also goes by several other names including API unhooking and system call unhooking. The goal of this technique is to evade EDR detection by *unhooking* the system calls that an EDR previously hooked to monitor for malicious usage. For EDRs to hook a system call, they must first locate it in a target process and then overwrite the implementation (instructions) of the system call so that the EDR gains control when the system call is made. This overwriting creates a discrepancy between the instructions of the function on disk (the original ones) versus the EDR written ones. By unhooking a function, malware resets (rewrites) a function's implementation to its original instructions, which nullifies the EDR's visibility into the function's usage. This then allows the malware to use the unhooked APIs without concern for being detected through userland system call monitoring.

### 3.2.2 Internals

Malware currently utilizes two techniques to unhook functions. For discussion purposes, we will use *ntdll.dll* as our example DLL to unhook as it's the main handler of system calls and is often the main DLL targeted for unhooking by malware. The first unhooking technique relies on all system call handling code

for *ntdll.dll* residing in its *.text* section. This also implies that any/all EDR hooks will be written inside of the *.text* section as well. To unhook all APIs at once using this realization, malware will first obtain a copy of an unhooked *ntdll.dll*, parse its metadata to locate the *.text* section, and then read it. This clean copy of the section will then be used to rewrite the *.text* section of the active *ntdll.dll* inside of the process, effectively overwriting every EDR hook at once. A 2023 report from Recorded Future [6] details this technique as utilized by the BlueBravo threat group:



When GraphicalNeutrino starts a thread, it attempts to remove any API hooks in the ntdll and wininet modules. The unhooking technique used is identical to the one described in an ired.team notes article, and is summarized as follows:

1. Maps a clean copy of the module into memory from the file system
2. Locates the .text section in both the original and clean copies of the module
3. Modifies the module's original .text section permission to be PAGE_READWRITE_EXECUTE and stores the original permissions
4. Copies the clean module's .text section over the original module's .text section, effectively removing any API hooks
5. Reapplies the original permissions to the original module's .text section

Figure 3: BlueBravo's Module Unhooking

The second form of this technique is to avoid overwriting the entire *.text* section and to instead only overwrite either every system call handler or only the handlers for the system calls needed by the malware. The end result is the same, as the hooks placed by the EDR are overwritten with the unhooked (original) versions of the system call handlers.

Interested readers can find more details about these techniques in [11, 12, 13].

### 3.2.3 New Detection Algorithm

To detect system call unhooking in a generic manner, we needed to develop new memory forensics techniques. Related previous research, including efforts conducted by members of our research team, provided new techniques for detecting API hooking to detect malicious hooks. Unfortunately, this previous research is not as useful in systems with EDRs present, since these systems will have thousands of APIs hooked, leading to thousands of false positives per memory sample unless allow lists are configured for each EDR and AV, which is not a scalable solution [14, 15, 16].

Our new approach detects system call unhooking by relying on the fact that malware will only unhook system calls in the small set of processes, generally one or two, that the malware operates and targets. Furthermore, even if new malware was released that attempted to unhook every process, modern versions of Windows restrict which processes can have code injected through restrictions on DLL loading, remote handle creation, and other code injection prerequisites. EDR and AV software will also often harden itself against such attacks and

malware often avoids the processes of such software through preconfigured allow and deny lists to avoid suspicion.

### 3.2.4 Adding Volatility Support

Our new Volatility 3 plugin, *windows.unhooked_system_calls*, automates the detection of unhooked modules by leveraging the discrepancy in instructions (code) between hooked and unhooked *ntdll.dll* modules inside of processes. It operates by locating and gathering the code of the most commonly unhooked system calls and then grouping processes by their implementation of each system call. It then reports the sets of processes that have distinct system call implementations. In practice, this means that all of the hooked processes will be grouped together while the one or two unhooked processes will be grouped.

To demonstrate our new detection algorithm using an easily reproducible scenario, we designed an experiment using an open-source EDR-equivalent along with an open-source malware toolkit that performs module unhooking. The EDR-equivalent used was SylantStrike [17], which performs system call hooking of *NtProtectVirtualMemory* to prevent memory regions from being temporarily changed to read-write-execute (RWX) protections. Readers should note that this changing of protections was listed as a prerequisite step in the previously discussed BlueBravo malware. *SylantStrike* is explained in full detail in a series of blog posts by its author [18].

For our experiment, we created a base Windows VM without any EDRs present. This meant that each process had a clean copy of *ntdll.dll* present. SylantStrike was then used to manually start two protected processes. The first process created was an instance of Notepad, the second an instance of Wordpad, and each process was verified to have the SylantStrike DLL loaded and the hook present. The following figure shows the output of our new plugin for a memory sample taken with SylvantStrike active:

```
  |               Function | Distinct Implementations           |
* | NtProtectVirtualMemory | 5780:notepad.exe, 4068:wordpad.exe |
* |          NtOpenProcess |                                    |
* |   NtQueryVirtualMemory |                                    |
* |    NtReadVirtualMemory |                                    |
* |             NtWriteFile |                                    |
```

Figure 4: SylantStrike's Active in Both Processes

As shown, the plugin reports both the Notepad process with PID 5780 and Wordpad process with PID 4068 as having distinct implementations since they are now both hooked by Sylant Strike whereas the rest of the processes have a clean *ntdll.dll*. Next, the open source R77 rootkit [19] was used to perform module unhooking of the Notepad process. R77 performs full overwriting of the *.text* section of *ntdll.dll* as part of its infection chain.

Our goal was to create a situation where the NtProtectVirtualMemory function of the Wordpad process would have the SylantStrike hook present

whereas the Notepad process would have its function reverted back to the version on disk by R77. This mimics malware performing unhooking in real attacks. The following figure shows the output of our plugin for a memory sample taken after R77 was used to perform unhooking:

```
  |                Function | Distinct Implementations          |
* | NtProtectVirtualMemory | 4068:wordpad.exe                    |
* |          NtOpenProcess |                                     |
* |   NtQueryVirtualMemory |                                     |
* |    NtReadVirtualMemory |                                     |
* |             NtWriteFile |                                    |
```

Figure 5: SylantStrike's Only Active in wordpad

Wordpad is now the only process with SylantStrike active as Notepad had its implementation reverted to the original version from *ntdll.dll* on disk. This verifies that the new plugin can detect module unhooking as deployed by real world malware. The other monitored functions are included in the plugin's checks and output as they are often valuable targets for more robust, enterprise-level EDRs.

To illustrate the effects of this experiment more clearly, we used Volatility's dlldump plugin to extract the *ntdll.dll* executables from both the Notepad and Wordpad processes after R77 was executed. The following figure shows the Wordpad process and that it is still hooked by SylantStrike:

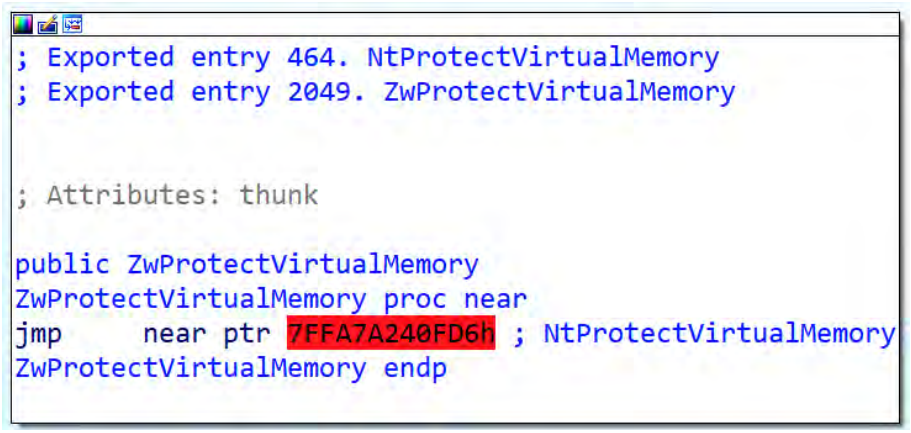



Figure 6: R77 Overwriting Notepad

The next figure shows the Notepad process, which has been reverted to the original function implementation, resulting in it having the same code as the rest of the system processes:

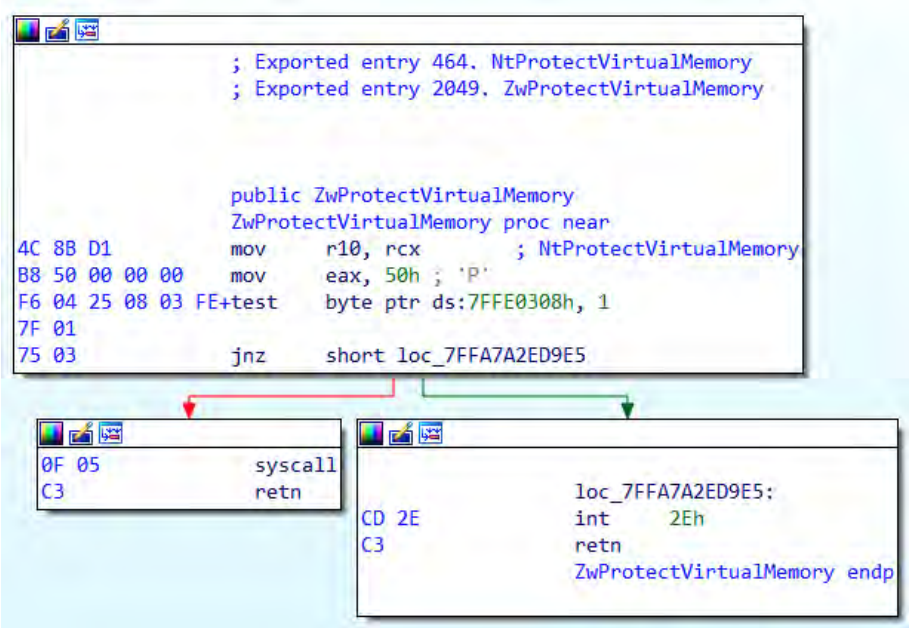



Figure 7: R77 Reverting the Instructions

## 3.3 Suspended and Cloned Processes

### 3.3.1 Background

The previous section discussed how a clean copy of *ntdll.dll* is used to overwrite all or part of the *.text* section of a hooked *ntdll.dll*. This discussion purposely glossed over *how* a clean copy is obtained as, during the course of our research, we discovered a new detection method related to how several malware samples obtain a clean *ntdll.dll*. We also noted that this alternate method of obtaining a clean copy of *ntdll.dll* partially overlapped with a novel technique presented at BlackHat Europe 2022, known as Dirty Vanity, to evade EDRs [20]. Given the abuse of overlapping APIs by both methods, we describe them here in the same section.

The default method that malware uses to get a clean copy of *ntdll.dll* is to read it from disk. Unfortunately for malware developers, some EDRs generate alerts upon observing this behavior, as reading *ntdll.dll* from disk as a regular file is not an operation expected of benign programs. To avoid detection, malware was observed to create a suspended process, by setting the *CREATE_SUSPENDED* flag to *CreateProcess*, to then read the clean *ntdll.dll* from the new process before EDRs have a chance to hook it.

The Dirty Vanity research effort used *cloned* processes to evade EDRs by having the first set of APIs needed for code injection called in the parent process and the final set called from the child. The Dirty Vanity presentation discussed

how EDRs were unable to follow this malicious API usage pattern from the parent to the child. During our analysis, we observed that within the process cloning phase, the child process is initially suspended, leading us to believe that a generic detection of suspended processes (threads) could catch a variety of EDR evasion techniques.

Readers may also recognize the use of suspended processes as a prerequisite for many process hollowing techniques, and, as discussed shortly, our new algorithm for detecting suspended processes detects several process hollowing variants along with the use of process suspension for evading EDRs.

#### 3.3.2 Internals

As documented in projects that abuse suspended processes to evade EDR, such as PeRun-Farts [21] and Freeze [22], creating a process in a suspended state will have the effect of the operating system creating a new process address space, but only mapping in the application executable and *ntdll.dll*. While in its suspended state, EDRs that have registered for process and thread creation notification routines will not yet be informed of the new process. This means that the half-created, suspended process holds a pristine copy of *ntdll.dll* in memory as it exists on disk. The controlling malware process can then read the unhooked *.text* section from its newly created child process to gather the data needed to unhook its own process.

On Windows, each thread is represented by an *_ETHREAD* structure and has a *_KTHREAD* structure embedded within it. When a process is created in the suspended state, its initial thread is set to the state of suspended. Later, if the process is resumed, this initial thread has its state changed to an operational state. This prevents detecting process suspending based on a thread's state flag. Fortunately, during our analysis of thread creation on Windows, we noticed several other side effects of creating a suspended process, including that the *SuspendCount* member of *_KTHREAD* is set to 1. We also noted that processes started normally (not suspended) have their suspend count initialized to 0. The use of *ResumeThread* to later make the thread execute will decrement this counter back to zero, but reuse of the original thread is not required for the process to function and does not occur in several of the malicious use cases we studied.

#### 3.3.3 New Detection Algorithm

Based on our observation that the *SuspendCount* member of a suspended process' main thread will start as 1 instead of 0, we updated Volatility's existing *thrdscan* plugin to report the number of times each thread was suspended based on the *SuspendCount* member as well as the backing library (DLL), if any, for these threads' starting execution address. We then ran the plugin across our previously described data set of Windows memory samples.

Analysis of these results showed that, with the exception of threads inside of browser processes pointing to *WorkFoldersShell.dll*, the only threads that had a suspend count greater than 0 were malicious ones. These included threads

created as part of EDR evasion as well as a variety of process hollowing techniques present in our test bed of memory samples.

### 3.3.4 Adding Volatility Support

To automate the detection of processes created in a suspended state for malicious purposes, we encapsulated our previously described knowledge into a new Volatility 3 plugin, *windows.suspended_threads*. This plugin enumerates every active thread of every process and then checks its suspend count. If the suspend count is greater than 0 and the hosting module of the thread is not *WorkFoldersShell.dll*, then its full information is reported. The following show this plugin detecting Dirty Vanity:

```
$ python3 vol.py  -r pretty -f Dirty-Vanity.lime windows.suspended_threads
Volatility 3 Framework 2.7.0
  |       Process |  PID |  TID |   StartAddress | Win32StartAddress
* |  FakeExe.exe | 6752 | 1964 | 0x7ffc13722680 |    0x7ffc137a6390
```

Figure 8: Detecting the Suspended Thread

The above shows the process with PID 6752 as having a previously suspended thread, and, as shown in the following figure, examining the parent/child relationship of this process shows that it was created by the parent FakeExe.exe and that it spawned the cmd.exe shellcode included with Dirty Vanity.

```
**** 7472 8152 FakeExe.exe    "C:\FakeExe.exe"
***** 6752   7472 FakeExe.exe "C:\FakeExe.exe"
****** 7188   6752 cmd.exe /k msg * Hello from Dirty Vanity
```

Figure 9: Related DirtyVanity Processes

While researching Dirty Vanity, we also came across an excellent writeup on the effects on Dirty Vanity and a deep dive of the *fork* implementation of Windows [23]. In this document, it states that a cloned process will have its initial thread's starting execution address pointing to the *RtlpProcessReflectionStartup* symbol inside of *ntdll.dll*. After reading this, we created a new Volatility 3 plugin called *windows.cloned_processes* that enumerates every thread of every process and reports if its initial thread points to the *RtlpProcessReflectionStartup* symbol. Running this plugin against the Dirty Vanity infected memory sample reported the same thread as *windows.suspended_threads*. This not only confirms the previously cited research, but also allows us to catch malware that uses either process suspending or process cloning for EDR evasion or other malicious purposes, such as process hollowing.

## 3.4 Direct System Calls

### 3.4.1 Background

The previously described evasion methods largely centered around module unhooking as this technique allows malware to make systems call without being detected. While effective, unhooking is rather drastic, particularly when overwriting all of the *.text* section, and, in edge cases, can lead to instability of the system. EDRs also have the option to check for the erasure of their hooks to immediately detect malicious activity inside of a particular process or set of processes.

With this in mind, alternate bypass methods have been developed that still provide for undetected system call usage, but that leave functions hooked by EDRs in their hooked state (no overwriting). The first of these that we will discuss is known as Direct System calls. With this method, malware obtains the SSN of a desired system call and then directly invokes the *syscall* instruction itself, avoiding the hooked system call handler inside of *ntdll.dll*. Since its inception, this technique has been implemented in many open-source tools and is still utilized by active APT groups, as documented in a recent CyCraft report [24].

### 3.4.2 Internals

One of the earliest writings to document direct system calls was from Cornelis [25]. This blog post included the release of an open source project named Dumpert that implemented direct system calls and provided detailed discussion of the technique. This early project was not usable in practice though as it relied on hardcoded system call numbers during compilation time. As mentioned previously, system call entry numbers (SSNs) change during each build of Windows and have no guaranteed order.

To work around this limitation, the Hell's Gate [26] technique, which could dynamically retrieve system call numbers, was developed. This technique allowed direct system calls to be used on any Windows version, without the requirement of knowing SSNs during compilation time. Hell's Gate operates by parsing the export table of *ntdll.dll* to dynamically find the *Nt** function for a system call of interest and then parses its instructions to find the corresponding SSN. The previously discussed Figure 2 shows how the SSN is referenced inside of each system call handler.

With the SSN available, the syscall instruction can be executed directly once the parameters are in place. The following figure shows the *HellsDescent* function of Hell's Gate, which is used to invoke the system call instruction. Note that the *wSyscall* variable holds the SSN of the system call to be invoked.

```
HellDescent PROC
        mov r10, rcx
        mov eax, wSystemCall

        syscall
        ret
HellDescent ENDP
```

Figure 10: The implementation of HellsDescent

While functional, the Hell's Gate technique has a limitation in that it needs a pristine (unhooked) copy of *ntdll.dll* to determine SSNs as otherwise it would be analyzing the hooked copy in memory, which would be error-prone and non-deterministic across EDR vendors. As discussed in previous sections, attempts to access a pristine *ntdll.dll* can be caught by EDRs and/or our new memory forensics techniques. This reality led to several twists on Hell's Gate that allowed for retrieving system call numbers without access to a clean DLL, such as Halo's Gate [27], Tartarus Gate [28], and system call table address sorting [29]. Automation of address sorting is also implemented within the SysWhispers2 project that allows for easier development of code using direct system calls [30].

### 3.4.3 New Detection Algorithm

Regardless of the technique used to acquire the SSN, all variations of the Direct System Calls technique can be detected through the presence of the syscall instruction or the older *int 2E* invocation outside of DLLs expected to host system calls, which include *ntdll.dll*, *wow64win.dll* (Wow64 system call support), and *win32u.dll* (Win32k/GUI system call support).

EDRs with in-kernel monitors have used this detection concept to detect direct system calls as EDR monitors in the kernel can acquire the callstack (list of functions and hosting module(s)) that led to a system call invocation. If the callstack reveals that the syscall instruction (last one before transitioning to the kernel) came from outside of a known system call module, then the EDR can block the call and generate an alert. Unfortunately for EDRs, this detection approach has largely been bypassed due to malware's ability to manipulate the callstack before the EDR can examine it [31, 32, 33, 34].

With memory forensics, we can avoid the callstack issue entirely by instead searching for the instructions necessary to make system call invocations throughout processes and then alerting on instructions outside of the known modules.

### 3.4.4 Adding Volatility Support

To automate the detection of direct system calls, we developed the *windows_direct_system_calls* plugin. It operates by first searching for instances of the *syscall; ret* or *int 0x2e* pattern inside of processes. For each instance, it uses Capstone to perform automated static analysis of the (up to) ten instructions proceeding the pattern and reports if it finds both a *mov* instruction that updates RAX/EAX as well

as a *mov* that updates R10. As discussed previously, these registers must be set correctly before a system call invocation.

While performing its analysis, the plugin takes anti-analysis precautions based on observed behaviour of several techniques, such as the source code of TarTarus Gate shown below:

```
HellDescent PROC
        nop
        mov rax, rcx
        nop
        mov r10, rax
        nop
        mov eax, wSystemCall
        nop
        syscall
        ret
HellDescent ENDP
```

Figure 11: Insertion of NOPs to Break Brittle Analysis

As can be seen, the *syscall; ret* pattern remains the same as the original HellsGate, but *nop* instruction has been inserted in between the proceeding ones as well as a two instruction process to copy RCX's value into R10. These changes would break naive approaches to detection, such as opcode scans with brittle definitions, but do not defeat our automated static analysis.

The following figure shows our plugin's detection of *HellsDescent* inside of a memory sample infected with HellsGate:

```
HellsGate.exe   424
0x7ff65f241591: mov r10, rcx
0x7ff65f241594: mov eax, dword ptr [rip + 0x3a66]
0x7ff65f24159a: syscall
0x7ff65f24159c: ret
```

Figure 12: Detecting *HellsDescent*

In this output, the HellsGate.exe process with PID 424 is detected due to the automated static analysis of the instructions proceeding the system call invocation. In verbose mode, the plugin also includes the disassembly of the analyzed instructions as is visible in the figure.

## 3.5 Indirect System Calls

### 3.5.1 Background

As mentioned previously, EDRs initially defeated direct system calls by detecting the invocation of system calls from modules outside of the normally expected DLLs. As a method to bypass this detection, the *indirect* system calls technique

was invented. This bypass operates by finding the address of a *syscall; ret* pair inside of *ntdll.dll* and then redirecting control flow, through a *jmp* instruction, to the address of the system call instruction instead of the malware invoking the system call instruction instruction itself. The use of *indirect* system calls makes it to where a location inside of *ntdll.dll* is what will appear in the callstack as the code responsible for the system call invocation, bypassing EDRs that only verify the hosting module of the invocation.

### 3.5.2 Internals

The following figure shows the implementation of system calls inside of the HellHall project [35]:

```
HellHall proc
        mov r10, rcx
        mov eax, dwSSN
        jmp qword ptr [qAddr]
        ret
HellHall endp
```

Figure 13: HellHall Implementation of Indirect System Calls

As shown, the use of *jmp* to the location of the system call instruction allows HellHall to partially hide its origin as the code responsible for the system call invocation. The code must still set EAX/RAX and R10 appropriately though like with direct system calls.

### 3.5.3 Adding Volatility Support

To detect indirect system calls, we developed a new plugin *windows.indirect_system_calls* that uses a modified approach to our detection of direct system calls. The plugin starts by searching for all *jmp [memory address]; ret* instructions pairs, ignoring *nop* instructions in between. For each found, it decodes the destination address of the jump and then checks if the first instruction is *syscall* or *int 0x2e*. As with direct system calls, it also parses the preceding instructions for RAX/EAX and R10 being initialized.

The following figure shows this plugin detecting the HellHall implementation of indirect system calls:

```
Process         PID
HellsHall.exe   2112
0x7ff6949e265e: mov r10, rcx
0x7ff6949e2661: mov eax, dword ptr [rip + 0x3999]
0x7ff6949e2667: jmp qword ptr [rip + 0x3997]
0x7ff6949e266d: ret
0x7ff6949e266e: int3
```

Figure 14: Detecting Indirect System Calls

As with our plugin for detection of direct system calls, our new plugin for detection of indirect system calls lists the process name, PID, and starting address of the system call stub. In verbose mode, as shown, a disassembly is also provided.

## 3.6 Exception Handlers and Hardware Breakpoints

### 3.6.1 Background

The final EDR bypass technique used in the wild is the abuse of exception handlers combined with hardware breakpoints. The use of debug registers by themselves to frustrate dynamic analysis is not new and was first published about by halfdead in a 2008 release of Phrack Magazine [36]. Similarly, real world APT and criminal groups have abused exception handlers to prevent analysis and gain stealth as described by recent reports from Palo Alto, Mcafee, and tccontre in reports on GuLoader and Gh0stRat [37, 38, 39].

A modern, interesting abuse of these features is using a combination of them to bypass EDR checks. There are several variants of this bypass technique, but the basic idea is that the the malware registers an exception handler and, in many variants of the technique, then sets breakpoint(s) to control execution of desired functions, without EDRs noticing the tampering. These malicious actions have the effect of the malware-controlled exception handler being activated right before the *syscall* instruction of the seemingly harmless system call is made, which allows the malware several options to bypass EDR detection. The RedOps blog has an excellent writeup on these abuses and we highly suggest that readers study this blog post after reading our paper [40].

### 3.6.2 Internals

All variants of this technique that our team studied, including those implemented in malware as well as open source toolkits, used custom exception handlers to bypass EDR monitoring while several also combined abuse of debug registers. Registering an exception handler is accomplished through the use of the *AddVectoredExceptionHandler* or *SetUnhandledExceptionFilter* functions. Once registered, the malware's handler will then be called whenever an exception is generated in the program. To bypass EDRs, malware deploys one of several observed techniques within its exception handler.

The first, as implemented in [41], encodes the desired system call number in a register and then forces an exception to trigger. This then transfers control to its exception handler, where it sets the RAX and R10 registers to the values needed for a system call and the instruction pointer to the address of a *syscall* instruction. It then lets the program continue execution, which will immediately execute the system call outside the view of EDR systems monitoring system call functions.

Another variation, as implemented in Mutation Gate [42], sets a breakpoint on the *syscall* instruction of a function unlikely to be monitored (hooked) by EDRs - *NtDrawText* in its current implementation. To make undetected

system calls, MutationGate then calls *NtDrawText* as normal, which triggers the breakpoint and redirects control flow to its malicious exception handler. Before allowing execution to continue, MutationGate sets the RAX register to the SSN of the system call it actually wants to make, such as *NtProtectVirtualMemory*, overwriting the SSN of *NtDrawText*. This has the effect of the malicious system call executing without having to interact with EDR hooks directly or from within hooked functions.

In December 2023, Marcus Hutchins (MalwareTech) released his EDRception project that used debug registers and a top level exception handler to bypass EDR detection. In his implementation, the user of the tool can choose to use exceptions or breakpoints to bypass EDR monitoring. In our testing, the debug register method did not activate correctly, but the top level exception handler did function as intended. This allowed us to detect the use of a malicious top level handler.

The BlindSide project uses hardware breakpoints of a process it creates in a debugged state to get access to a clean ntdll.dll [43]. It achieves this by setting a hardware breakpoint on *LdrLoadDLL* within the debugged child process and then detecting when ntdll.dll is first loaded. This reaches a similar goal of previously described projects that access a clean *ntdll.dll* within a created child process before an EDR can place its hooks.

Several projects leverage hardware breakpoints to implement *patchless* bypasses and loading capabilities, such AMSI patching [44] or loading of the .NET CLR into a target process [45]. These operate placing breakpoints on functions to be hooked or called, such as *AmsiScanBuffer* to tamper with AMSI scanning. With the breakpoints active, the malware can then tamper with the input parameters received and return values of calls to functions where breakpoints are set.

### 3.6.3 Adding Volatility Support

The research in this section led to the development of three new Volatility 3 plugins. The first, *windows.veh*, enumerates the vectored exception handlers (VEH) of each process and then performs automated static analysis of the handlers to determine if they appear malicious. The algorithm to enumerate the VEH of a process is explained in two great resources by NCC Group and Dimitri Fourny [46, 47]. As noted in these references and observed in our testing, many legitimate programs use VEH for a variety of purposes, so simply enumerating handlers is not sufficient from an automated triage and analysis perspective. In terms of detecting malicious VEH, there was a talk entirely focused on malicious vectored exception handlers that was given during our research timeframe at Black Hat MEA in late 2023. Studying this talk, the main detection focus was on VEH that checked for breakpoint based exceptions. In our study of VEH handlers across real world malware and open source projects, we observed many technique variants that did not check for breakpoint exceptions and instead require more thorough static analysis to detect. This led to our detection algorithm, which checks for any of the following registers being written to within the exception's

context structure:

- R/EAX (syscall parameter, faking return function value)
- R10 (syscall parameter)
- RSP/stack pointer (several evasion purposes)
- RIP/instruction pointer (several evasion purposes)
- RCX (address of start address to thread creation)

The following figure shows detection of MutationGate's VEH within our new plugin:

```
$ python3 vol.py -f MutationGate.lime windows.veh
Volatility 3 Framework 2.7.0
Process        PID   Address
MutationGate.e 4244  0x7ff758cd1400
```

Figure 15: Detecting Malicious VEH

The second plugin developed was *windows.ueh* that enumerates the unhandled exception handler for a process. These exception handlers are called whenever no other exception handler in a process catches an exception, assuming the process is not being debugged [48]. Calling this function sets the *BasepCurrentTopLevelFilter* global variable inside of the process. Our new plugin locates *BasepCurrentTopLevelFilter* and then performs the same analysis on the handler as our new VEH plugin. The following figure shows detection of EDRception with our new plugin:

```
$ python3 vol.py -f EDRception.lime windows.ueh
Volatility 3 Framework 2.7.0
Process        PID   Address
EDRceptionForc 9144  0x7ff74d2d1100
```

Figure 16: Detecting Malicious UEH

The last plugin developed was *windows.debug_registers*, which enumerates every thread of every process and then examines if the debug registers are set. For reference, debug registers 0-3 (Dr0-Dr3) hold the target addresses of breakpoints (where they are set), debug register 7 (Dr7) holds information about which breakpoints are active, and debug register 6 (Dr6) is set by the hardware when breakpoints are triggered. Exception handlers examine Dr6 to determine the reason for the exception. Our plugin looks for processes where Dr7 is set

and then, for Dr0-Dr3, it extracts the breakpoint address, finds the memory range hosting the breakpoint, and attempts to resolve any symbol names at the address.

```
$ python3 vol.py -r pretty -f patchless_amsi.lime windows.debug_registers
Volatility 3 Framework 2.7.0
  |        Process |  PID |  TID |  Dr7 |            Dr0 | Dr0 Range|     Dr0 Symbol |
* | patchless_amsi | 3520 | 8512 | 1025 | 0x7ffa648b3860 | amsi.dll | AmsiScanBuffer |
```

Figure 17: Detecting *Patchless* Bypasses

The above figure shows our new plugin detecting the AMSI patchless bypass technique, including revealing that the breakpoint is set on *AmsiScanBuffer*. This function is the target of the bypass and a popular function for manipulation by malware. Note that the figure only includes the Dr0 related information to make the image fit as well as that the particular bypass technique implementation only uses Dr0. The full plugin output includes the same information for Dr0-Dr3.

# 4 Conclusions

In this paper, we have documented our extensive research to develop novel memory forensics techniques capable of detecting all EDR process monitoring bypass techniques used in the wild and by popular open source toolkits. The use of these techniques allow malware to operate undetected and to perform a wide variety of desired activity, such as lateral movement, code injection, and credential theft. All of our developed techniques were created as plugins to the open source Volatility 3 project and will be release during the conference, allowing incident response handlers and threat hunters to immediately utilize the capabilities in real investigations.

# References


[1] "Operation Dragon Castling: APT group targeting betting companies," https://cymulate.com/threats/operation-dragon-castling-apt-group-targeting-betting-companies/, 2023.

[2] "Defeating Guloader Anti-Analysis Technique," https://unit42.paloaltonetworks.com/guloader-variant-anti-analysis/, 2023.

[3] "A Deep Dive Into ALPHV/BlackCat Ransomware," https://securityscorecard.com/research/deep-dive-into-alphv-blackcat-ransomware/, 2024.

[4] "APT Operation Skeleton Key," https://cycraft.com/download/CyCraft-Whitepaper-Chimera_V4.1.pdf, 2023.

[5] "LockBit Ransomware Side-loads Cobalt Strike Beacon with Legitimate VMware Utility," https://www.sentinelone.com/labs/lockbit-ransomware-side-loads-cobalt-strike-beacon-with-legitimate-vmware-utility/, 2024.

[6] "BlueBravo Uses Ambassador Lure to Deploy," https://go.recordedfuture.com/hubfs/reports/cta-2023-0127.pdf, 2024.

[7] "UNMASKING THE DARK ART OF VECTORED EXCEPTION HANDLING: BYPASSING XDR AND EDR IN THE EVOLVING CYBER THREAT LANDSCAPE," https://blackhatmea.com/session/unmasking-dark-art-vectored-exception-handling-bypassing-xdr-and-edr-evolving-cyber-threat, 2023.

[8] "Dirty Vanity: A New Approach to Code injection & EDR bypass," https://i.blackhat.com/EU-22/Thursday-Briefings/EU-22-Nissan-DirtyVanity.pdf, 2022.

[9] Volexity, "Surge Collect Pro," https://www.volexity.com/products-overview/surge/, 2022.

[10] "capstone," https://www.capstone-engine.org/, 2024.

[11] "Silencing cylance: A case study in modern edrs," https://www.mdsec.co.uk/2019/03/silencing-cylance-a-case-study-in-modern-edrs/, 2019.

[12] "Av/edr evasion — malware development p — 3," https://medium.com/@0xHossam/unhooking-memory-object-hiding-3229b75618f7, 2023.

[13] "A practical guide to bypassing userland api hooking," https://www.advania.co.uk/insights/blog/a-practical-guide-to-bypassing-userland-api-hooking/, 2022.

[14] A. Case, A. Ali-Gombe, M. Sun, R. Maggio, M. Firoz-Ul-Amin, M. Jalalzai, and G. G. R. III, "HookTracer: A System for Automated and Accessible API Hooks Analysis," *Proceedings of the 18th Annual Digital Forensics Research Conference (DFRWS)*, 2019.

[15] F. Block, "Windows memory forensics: Identification of (malicious) modifications in memory-mapped image files," *Forensic Science International: Digital Investigation*, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281723000707

[16] F. Block and A. Dewald, "Windows memory forensics: Detecting (un)intentionally hidden injected code by examining page table entries," *Digital Investigation*, vol. 29, pp. S3–S12, 07 2019.

[17] "CCob," https://github.com/CCob/SylantStrike/tree/master, 2024.

[18] "Lets Create An EDR... And Bypass It! Part 1," https://ethicalchaos.dev/2020/05/27/lets-create-an-edr-and-bypass-it-part-1/, 2020.

[19] "r77 rootkit," https://github.com/bytecode77/r77-rootkit/, 2024.

[20] "Deep Vanity," https://github.com/deepinstinct/Dirty-Vanity, 2022.

[21] "Peruns-Fart," https://github.com/plackyhacker/Peruns-Fart/, 2023.

[22] "FREEZE – A PAYLOAD TOOLKIT FOR BYPASSING EDRS USING SUSPENDED PROCESSES," https://www.hawk-eye.io/2023/06/freeze-a-payload-toolkit-for-bypassing-edrs-using-suspended-processes/, 2023.

[23] "Process Cloning," https://github.com/huntandhackett/process-cloning, 2023.

[24] "APT Group Chimera," https://cycraft.com/download/CyCraft-Whitepaper-Chimera_V4.1.pdf, 2022.

[25] "Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR," https://www.outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/, 2019.

[26] "Hell's Gate," https://github.com/am0nsec/HellsGate/blob/master/hells-gate.pdf, 2020.

[27] "Halo's Gate," https://blog.sektor7.net/#!res/2021/halosgate.md, 2021.

[28] "Tartarus Gate," https://trickster0.github.io/posts/Halo's-Gate-Evolves-to-Tartarus-Gate/, 2021.

[29] "Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams," https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/, 2020.

[30] "SysWhispers2," https://github.com/jthuraisamy/SysWhispers2, 2022.

[31] "An Introduction into Stack Spoofing," https://dtsec.us/2023-09-15-StackSpoofin/, 2023.

[32] "SilentMoonwalk: Implementing a dynamic Call Stack Spoofer," https://klezvirus.github.io/RedTeaming/AV_Evasion/StackSpoofing/, 2022.

[33] "Spoofing Call Stacks To Confuse EDRs," https://labs.withsecure.com/publications/spoofing-call-stacks-to-confuse-edrs, 2022.

[34] "Behind the Mask: Spoofing Call Stacks Dynamically with Timers," https://www.cobaltstrike.com/blog/behind-the-mask-spoofing-call-stacks-dynamically-with-timers, 2022.

[35] "HellHall," https://github.com/Maldev-Academy/HellHall, 2023.

[36] http://phrack.org/issues/65/8.html#article, 2008.

[37] "Defeating Guloader Anti-Analysis Technique," https://unit42.paloaltonetworks.com/guloader-variant-anti-analysis/, 2022.

[38] "GULoader Campaigns: A Deep Dive Analysis of a highly evasive Shellcode based loader," https://www.mcafee.com/blogs/other-blogs/mcafee-labs/guloader-campaigns-a-deep-dive-analysis-of-a-highly-evasive-shellcode-based-loader/, 2023.

[39] "Gh0stRat Anti-Debugging : Nested SEH (try - catch) to Decrypt and Load its Payload," https://tccontre.blogspot.com/2021/02/gh0strat-anti-debugging-nested-seh-try.html, 2021.

[40] "Syscalls via Vectored Exception Handling," https://redops.at/en/blog/syscalls-via-vectored-exception-handling, 2024.

[41] "Bypassing AV/EDR Hooks via Vectored Syscall - POC," https://cyberwarfare.live/bypassing-av-edr-hooks-via-vectored-syscall-poc/, 2022.

[42] "MutationGate," https://github.com/senzee1984/MutationGate/tree/main, 2024.

[43] Cymulate Research, "BlindSide," https://github.com/CymulateResearch/Blindside/blob/main/Blindside/Blindside.cpp#L31, 2023.

[44] "In-Process Patchless AMSI Bypass," https://ethicalchaos.dev/2022/04/17/in-process-patchless-amsi-bypass/, 2022.

[45] "PatchlessCLR," https://github.com/VoldeSec/PatchlessCLRLoader/tree/main, 2022.

[46] "Dumping the VEH in Windows 10," https://dimitrifourny.github.io/2020/06/11/dumping-veh-win10.html, 2020.

[47] "Detecting anomalous Vectored Exception Handlers on Windows," https://research.nccgroup.com/2022/03/01/detecting-anomalous-vectored-exception-handlers-on-windows/, 2022.

[48] "SetUnhandledExceptionFilter," https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-setunhandledexceptionfilter, 2024.