A Deep Dive Into Malicious Direct Syscall Detection - Palo Alto Networks Blog

By By Or Chechik and Ofir Ozer

Published: 2024-02-13 · Archived: 2026-04-05 14:02:19 UTC

Executive Summary

In this blog post we will explain how attackers use direct syscalls to overcome most EDR solutions, by first discussing the conventional Windows syscall flow and how most EDR solutions monitor those calls. Then, we will dive into different attack techniques threat actors use to stay hidden from those EDR solutions, including the use of direct syscalls.

Finally, we'll go over how Cortex XDR can monitor direct syscalls and detect their malicious uses, as demonstrated by a real-world example.

Background

Endpoint detection and response (EDR) solutions have revolutionized the cybersecurity landscape in recent years. These innovative tools were born out of a pressing need to bolster defenses against the growing sophistication of cyberattacks.

At their core, EDR solutions serve as a sentry for an organization's endpoints, such as computers and servers. They continuously monitor these endpoints by logging different actions taking place in real-time. EDR solutions can be a central pillar in cybersecurity strategies, and by collecting and analyzing vast amounts of endpoint data, they can uncover patterns, anomalies, and potential threats, thereby fortifying their ability to mitigate risks.

Naturally, threat actors have wanted to find ways to bypass or subvert the EDR detections. To achieve this goal, they have targeted one of the most critical functionalities of EDR solutions, including the ability to monitor API calls, which tracks process actions on the installed endpoint. This led them to use direct syscalls, which is one of the most prevalent techniques to bypass user mode EDR solutions.

Let’s go into the direct syscall technique and how Cortex XDR is able to detect malicious uses of them in the Windows operating system.

Windows System Call Flow

First, we need to understand what a syscall is. A system call, abbreviated syscall, is an assembly instruction that enables the transition from user mode to kernel mode for the purpose of executing a task by the kernel. Tasks such as access to hardware peripherals, reading a file or sending packets over a TCP network socket can only be executed by the kernel. In a conventional flow, the system call is implemented inside system call stubs located inside ntdll.dll or win32u.dll, Windows DLLs, and it is usually called in the context of a Windows API.

The following example is intended to illustrate how system calls work within the Windows operating system:

A user mode application requests the kernel to open a handle to a file. As seen in Figure 1, the application starts by calling the Windows API CreateFileW in kernel32.dll. Then, it calls the CreateFileW function implemented inside kernelbase.dll, which in turn calls the undocumented function CreateFileInternal.

The latter calls the Native API NtCreateFile in ntdll.dll.

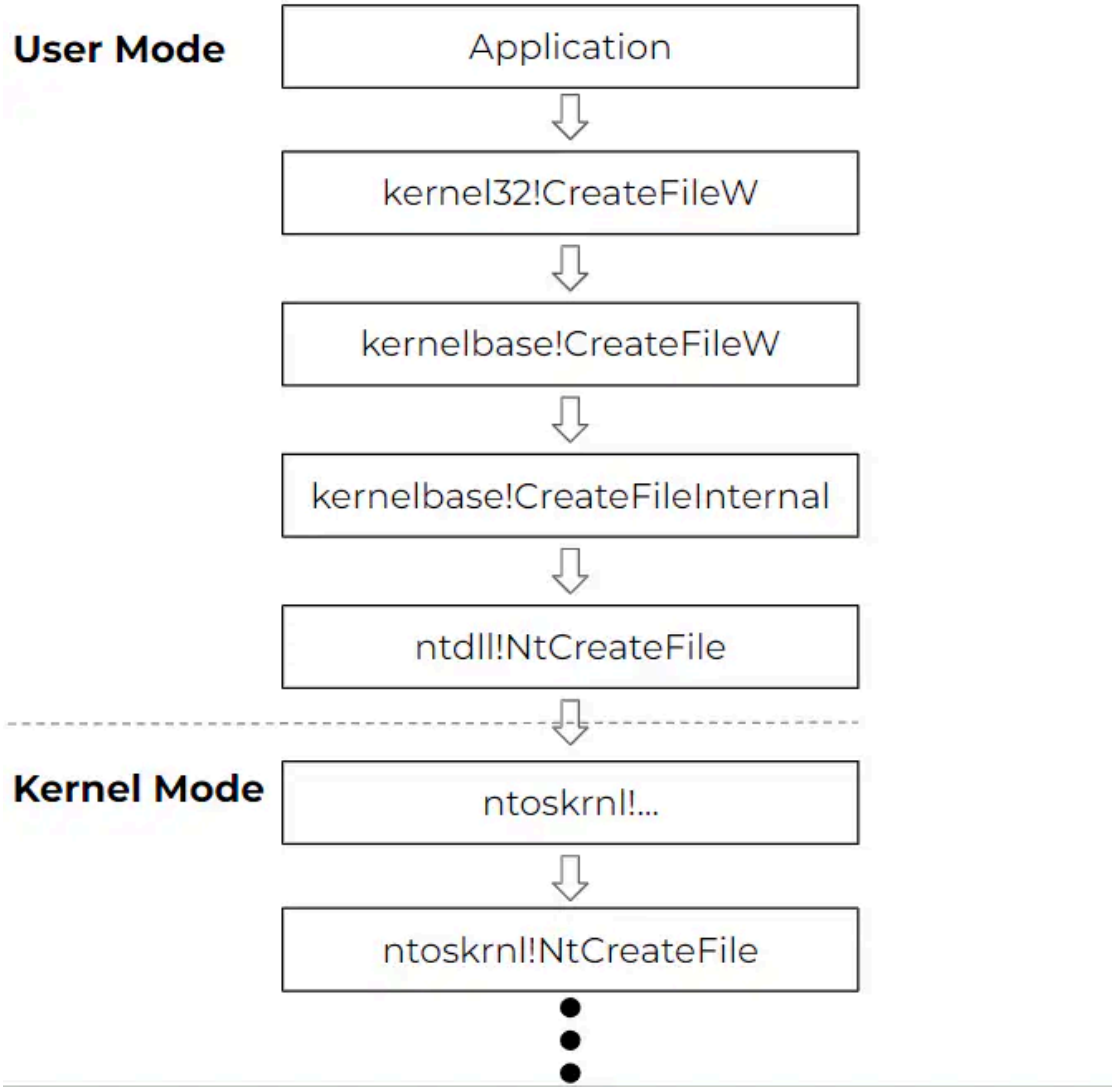

Figure 1. Conventional system call flow

As seen in Figure 2, the system call stub for NtCreateFile executes the syscall instruction with the syscall index for NtCreateFile, populated into the EAX register.

Figure 2. System call stub for ZwCreateFile in ntdll

How Most EDRs Monitor API Calls

Most EDRs monitor system calls and API calls by hooking user mode Windows DLLs. The common method EDRs use is inline hooking, which is a method of intercepting calls to target functions as a way to prevent or monitor potentially malicious operations. An example of EDR inline hooking is shown in Figure 3. By modifying the assembly instructions of the prologue of CreateFileW to a JMP instruction, the execution is redirected to the EDRs proxy function where it can monitor or prevent the operation. In turn, the EDR Proxy jumps to the trampoline function which will execute the original CreateFileW prologue and then jumps back to CreateFileW, right after the prologue, to resume the function execution.

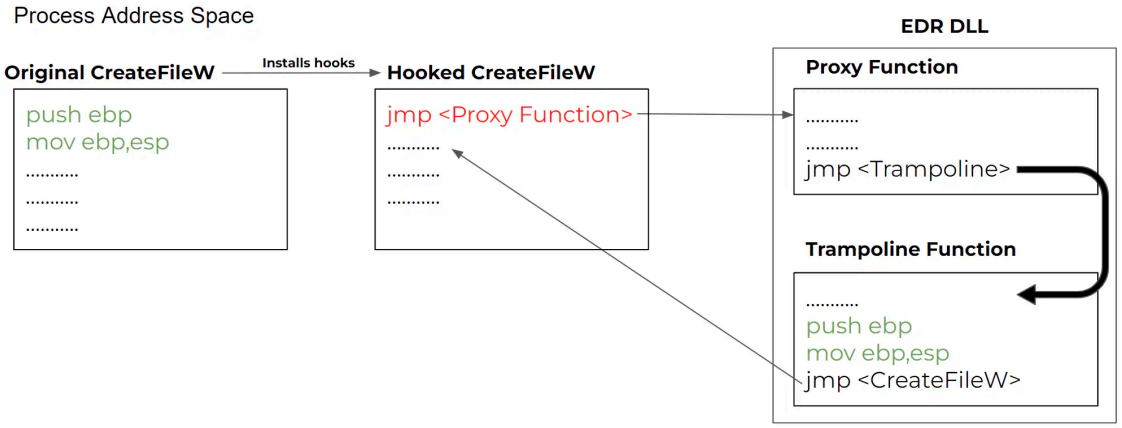

Figure 3. Inline Hooking Example

EDRs commonly inject their DLL as early as the process starts, and this DLL usually hooks Windows API and Native API functions inside the Windows libraries such as kernel32.dll, kernelbase.dll, and ntdll.dll.

How Attackers Bypass Most EDRs

Most EDRs use user mode hooks for their syscall monitoring, therefore a lot of the EDR bypasses are user mode hook bypasses.

Most of those EDR bypasses map a clean copy of the hooked DLL or try to restore the hooked DLL to its original disk content to avoid the EDR hooks.

Out of the many EDR bypass techniques, the most prominent one is the “Direct Syscall” technique. To achieve maximum stealth, attackers frequently use it in conjunction with other evasion techniques, as it is very hard to detect and differentiate between malicious and benign use of it.

Following are notable bypass techniques:

Manual Load DLL From Disk

A technique in which the attacker manually loads a clean copy of a hooked DLL using Reflective DLL Loading. This technique allows the attacker to avoid getting through the Windows Loader, and avoid the EDR callbacks that effectively install the hooks.
The attacker implements the Windows Loader functionally on its own, loading the DLL directly from memory, giving it a different mapping type than the loader would.

Clone DLL

Similarly to the previous technique, this technique loads a secondary DLL to avoid the EDR hooks. This technique however does go through the Windows Loader and uses the Windows API. The attacker copies the target DLL from %system32% to a new location with a new name, then it uses the LoadLibrary Windows API to load it from disk and use it in its code. The cloned DLL will have a different name than the hooked DLL, so the EDR won’t install hooks for it.

Direct Syscall

Attackers use the direct syscall technique to avoid going through the user mode hooks, because it allows an attacker to execute a system call without going through the Windows and the Native API. Essentially, the attacker implements the system call stub in its own application as seen in Figure 4.

```
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                                ; Move the contents of rcx to r10. This is necess
    mov eax, wNtAllocateVirtualMemory           ; Move the syscall number into the eax register.
    syscall                                     ; Execute syscall.
    ret                                         ; Return from the procedure.
NtAllocateVirtualMemory ENDP                    ; End of the procedure.
```

Figure 4. Example of system call stub that performs the syscall NtAllocateVirtualMemory

The syscall instruction forwards the execution flow to the corresponding syscall implementation in the kernel according to the syscall index. It’s the last step in user mode.

Different Windows builds and service packs use different syscall indexes.

To address this issue, attackers often parse NTDLL in runtime and find the correct syscall indexes out of the syscall stubs.

Cortex XDR is not affected by any of the mentioned techniques.

Cortex XDR Direct Syscall Detection

Compared to other EDR solutions, Cortex XDR employs a kernel mode syscall interception technique, providing access into kernel structures, and the ability to synchronously block or record syscalls, ensuring robustness against user mode hook bypass methods. Using this method, Cortex XDR detects direct syscalls for many different syscalls commonly used by attackers.

Cortex XDR leverages the KTRAP_FRAME structure built on top of the base of the thread’s kernel mode stack upon syscall handling.

The syscall handler builds the KTRAP_FRAME, a structure in which the kernel saves the state of execution that gets interrupted, by an exception, an interrupt or a syscall, and stores the initial user mode context, that is, it stores the registers to restore execution in user mode once the syscall is done.

To detect a direct syscall, for every intercepted system call, Cortex XDR extracts the RIP from the KTRAP_FRAME, which is the return address to user mode, then it resolves the user mode return address to the corresponding loaded module the address points to (effectively the user mode module that called the syscall instruction).

The resolution of the address is done by a unique component of Cortex XDR called ImageTracker that can resolve addresses to image paths and to an image’s closest export. ImageTracker keeps track of the loading and unloading actions of images for all the processes in the system. Cortex XDR uses ImageTracker to resolve addresses across the entire product.

As we mentioned above, in a conventional Windows syscall flow, the captured return address would be inside the ntdll.dll or win32u.dll modules, so if the return address is inside any other DLL, or can't be mapped into any DLL (shellcode), then the event is marked as a direct syscall and the path to the initiating module (for a non-shellcode case) is recorded as part of the event.

Lastly, in order to detect if this is a malicious direct syscall, the event is sent to the cloud for statistical analytics and anomaly detection and to the Behavioral Threat Protection module which runs on the endpoint.

Are Direct Syscalls Malicious?

Using direct syscalls is not always malicious. This action can be performed frequently by legitimate software, such as security products, gaming anti-cheat modules, Chromium-based applications and more.

It is very difficult to differentiate between legitimate and malicious direct syscalls, based on single-event information or static signatures.

Our solution to this problem was leveraging the power of the Cortex XDR Analytics Engine.

How Analytics Detects a Malicious Use of Direct Syscalls

The Cortex XDR Analytics Engine is a learning mechanism used to detect attacks that are otherwise very difficult or even impossible to detect using other methods. Analytics capabilities on XDR data relies on collection and ingestion techniques that operate in a highly scalable and efficient manner.

The Analytics Engine creates aggregations dynamically based on real-time events from Cortex XDR agents. These aggregations are then used to define baselines of "common" behaviors for each customer locally and globally.

In order to establish a baseline using the Analytics Engine, we look for the appropriate questions that will enable us to determine with certainty whether an action is common or rare for each event. Those questions point us to the key artifacts we should aggregate and the relations between them.

These are the questions that we found to be most effective at profiling direct syscall behaviors:

- How common is this direct syscall?
- Does this process usually execute direct syscalls?
- How often does the mapped memory location call a direct syscall?

Using local and global aggregations, we answer those questions in real time and conclude whether a direct syscall event is benign or malicious.

Real-Life Example - Lumma Stealer

Let's take a look at a real-world attack that we discovered using Direct Syscall Analytics, which detected the injection method used by the Lumma Stealer malware.

Lumma Stealer is a malicious tool that’s part of a malware-as-a-service campaign that was first seen in 2022 when it was advertised in dark web forums. It is an info stealer that primarily targets cryptocurrency wallets and two-factor authentication (2FA) browser extensions in addition to exfiltrating sensitive data from its targets.

We will not go over all stages of the attack, but we will go over the steps it takes to perform the direct syscalls.

It all starts with a legitimate looking install file called “Setup.exe”, which is the Lumma Stealer unpacker and injector. This file contains an expired Kaspersky certificate:

Property	Value
File Name	[redacted]\Setup.exe
File Type	Portable Executable 32
File Info	Microsoft Visual C++ 8
File Size	10.27 MB (10765648 bytes)
PE Size	10.26 MB (10753536 bytes)
Created	Thursday 02 November 2023, 14.08.47
Modified	Monday 06 November 2023, 14.13.07
Accessed	Wednesday 08 November 2023, 15.43.22
MD5	7A939924177E0B931B9FE2F1E2804DFE
SHA-1	89C30E4FBE264DF9FF25A2A9F4E60BD32BD03BCC

Property	Value
CompanyName	Kaspersky
FileDescription	Kaspersky [21.13.5.506.0.26.0]
FileVersion	21.13.5.506
LegalCopyright	© 2023 AO Kaspersky Lab
LegalTrademarks	Registered trademarks and service marks are the property of the...
ProductName	Kaspersky
ProductVersion	21.13.5.506
InternalName	Setup

Figure 5. LummaStealer signature information

Upon execution, the malware tries to send a beacon to arthritis[.]org - upon failure, the malware stops running.

Following the beacon, Lumma Stealer decodes a shellcode, loads the signed Windows DLL “mshtml.dll”, overwrites the start of the .text section of the DLL with the decoded shellcode and calls the address of the written shellcode directly.

Running from mshtml.dll, the shellcode decodes another shellcode that is destined to be executed in a different process - cmd.exe for another loading step.

First, the malware loads “ntdll.dll” from disk and extracts the right syscall’s index while parsing the functions it needs.

In order to do that it holds a list of hashes (calculated with MurmurHash2), those hashes are then compared to hashes of function names extracted from the loaded “ntdll.dll” file. Upon finding the right one, the function extracts the current syscall index and stores it in a variable for later use.

```
70C71836 lea     ecx, [ebp+var_808_ntdll_path]
70C7183C call    Read_file
70C71841 add     esp, 8
70C71844 mov     eax, [ebp+var_280ZwCreateSection_Hash]
70C7184A push    eax
70C7184B mov     ecx, [ebp+var_28_ntdll_buffer]
70C7184E push    ecx
70C7184F call    get_syscall_num ; ZwCreateSection
70C71854 add     esp, 8
70C71857 mov     [ebp+var_35C_ZwCreateSection_OpCode], eax
70C7185D mov     edx, [ebp+var_284_ZwMapView_Hash]
70C71863 push    edx
70C71864 mov     eax, [ebp+var_28_ntdll_buffer]
70C71867 push    eax
70C71868 call    get_syscall_num ; ZwMapViewOfSection
70C7186D add     esp, 8
70C71870 mov     [ebp+var_358_ZwMapView_OpCode], eax
```

Figure 6. Snippet of syscall index extractor functions

The syscall indexes extracted are:

- ZwCreateSection
- ZwMapViewOfSection
- ZwWriteVirtualMemory
- ZwProtectVirtualMemory
- NtSuspendThread
- ZwResumeThread
- ZwOpenProcess
- ZwGetContextThread
- NtSetContextThread

Lumma Stealer uses those syscall indexes for the next stages of the injection. Each syscall index has a dedicated function that sets up the arguments it needs and calls the syscall invoker function:

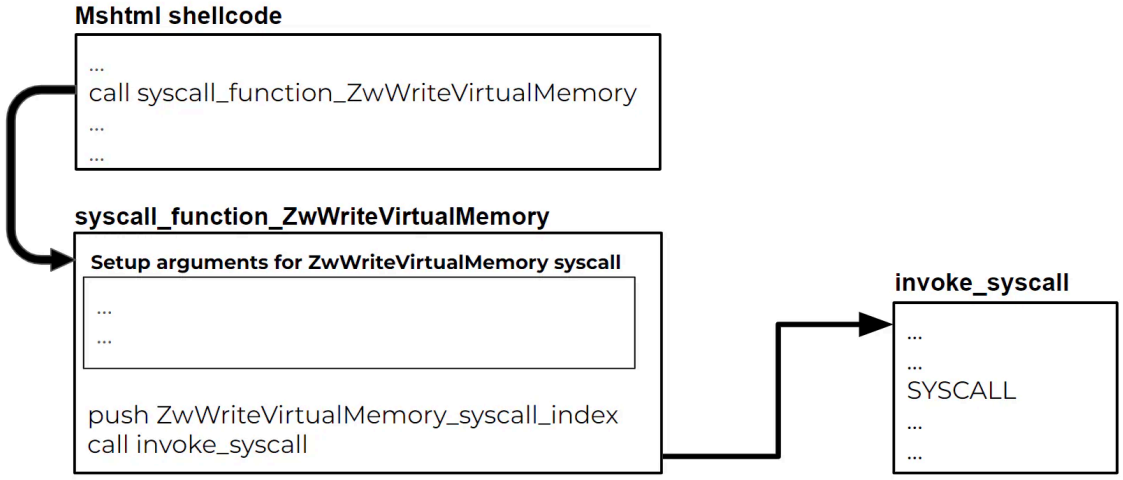

Figure 7. Lumma Stealer direct syscall invocation chart

In the syscall invoker function, we can see another technique that the malware writers used to wrap the syscall execution - Heaven's Gate.

This technique enables manual transaction of x86 CPU mode to a x64 CPU and vice versa.

The "invoke_syscall" function uses two instances of Heaven's Gate before and after the direct syscall execution. It switches the CPU mode to x64 first:

```
89 65 F4          mov     [ebp+var_C], esp
83 E4 F0          and     esp, 0FFFFFFF0h
6A 33             push    33h ; '3'
E8 00 00 00 00    call    $+5
83 04 24 05       add     [esp+80h+var_80], 5
CB                retf
```

Figure 8. x86 to x64 Heaven's Gate before direct syscall execution

Figure 8. x86 to x64 Heaven's Gate before direct syscall execution

Then the direct syscall is executed followed by another instance of Heaven's Gate that switches the CPU mode back to x86:

```
push    qword ptr [rbp-54h]
pop     r10
mov     rax, [rbp+8]
sub     rsp, 28h
syscall
mov     rcx, [rbp-3Ch]
lea     rsp, [rsp+rcx*8+28h]
pop     rdi
mov     [rbp-44h], rax
call    $+5
mov     dword ptr [rsp+4], 23h ; '#'
add     dword ptr [rsp], 0Dh
retf
```

Figure 9. Direct syscall execution and x64 to x86 Heaven's Gate

Using this method, the shellcode within mshtml.dll memory space is able to run the listed functions via direct syscalls.

In order to inject the second shellcode to cmd.exe (as mentioned above), LummaStealer executes a Wow64 cmd.exe process, using CreateProcessW (not a direct syscall). Then, it uses the direct syscalls to suspend the process, create memory sections in the target cmd.exe process and write the shellcode to them. Finally, it sets cmd.exe thread context to execute the written shellcode and resume the thread.

Cortex XDR has detected those direct syscall executions as malicious using several aggregations on local and global levels:

- "Setup.exe" is not a process that invokes those direct syscalls normally.
- The direct syscalls were executed from the mapped memory of mshtml.dll, which also doesn't usually execute those direct syscalls.

Conclusion

As seen in the case above, direct syscalls are used to subvert detection of malicious actions by most EDR solutions. This case is only one example of using direct syscall in a malicious manner, as we have seen more uses of it during our research.

To detect this elusive attack vector, we had to combine methods for monitoring and enriching direct syscall events using the agent with a learning mechanism for detecting rare and malicious direct syscall uses.

Palo Alto Networks customers with Cortex XDR are protected from this kind of attack with Cortex Analytics BIOC alerts:

Figure 10. Cortex XDR Direct syscall alerts and their source

Source: https://www.paloaltonetworks.com/blog/security-operations/a-deep-dive-into-malicious-direct-syscall-detection/