Malware Mitigation when Direct System Calls are Used

By sharon

Published: 2018-11-27 · Archived: 2026-04-06 01:19:50 UTC

In 2018 we have seen an increase in the malicious use of direct system calls in order to evade security product hooks.

These hooks are used to monitor API calls which may hint to malicious activity.

Direct system calls evasion method utilizes reading system call numbers from ntdll.dll, putting the appropriate system call number on the EAX register, putting the function parameters on the stack and entering the kernel directly by using sysenter or int 0x2e commands. This way – the function from ntdll.dll isn't called and the hook is useless at detecting the malicious activity. This photo shows how most malware perform direct systems calls:


Malware Mitigation Directly executing functions of the Kernel without calling functions from ntdll.dll

Figure 1 – Directly executing functions of the Kernel without calling functions from ntdll.dll

The direct system calls are usually used to silently inject malicious code into other processes. In 32-bit systems, it is possible to monitor system calls in the kernel by hooking the SSDT.
However in Windows Vista and up (64 bits only). It is not possible because of the patch guard mechanism.

In this article, we will show some ways to mitigate these malware – showing that even without hooks, we can find traces of the malicious activity and how to spot their code injection.

Malware analyzed

We will cover 8 different malware:

- LockPoS

SHA256: 093fcbf6bd7e264201ce5405f83101cefa5588a56c5bab11cfdbabbab3ed8a28

- Flokibot

SHA256: 5e1967db286d886b87d1ec655559b9af694fc6e002fea3a6c7fd3c6b0b49ea6e

- Trickbot

SHA256: 1c81272ffc28b29a82d8313bd74d1c6030c2af1ba4b165c44dc8ea6376679d9f

- Formbook

SHA256: c2bbec7eb5efc46c21d5950bb625c02ee96f565d2b8202733e784e6210679db9

- Osiris

SHA256: e7d3181ef643d77bb33fe328d1ea58f512b4f27c8e6ed71935a2e7548f2facc0

- Neurevt

SHA256: 634a93d42b6d2d7d302377aa8f1fb4b57ab923544836c83642c059826e9969d2

- Fastcash

SHA256: 820ca1903a30516263d630c7c08f2b95f7b65dffceb21129c51c9e21cf9551c6

- CoinMiner

SHA256: 8b309c799d53ac759a8b304e068e2530958d7f1577d20775b589e17e533d132a

Due to the lack of hooks, we cannot trace the functions called from ntdll.dll. But what can we trace? What can we get our hands on? Let's collect and examine evidence and then make conclusions how we can mitigate these malware.

We recall that for code injection in the context of the system calls method, we need to perform two tasks:

- Reading ntdll.dll for the system call numbers
- Performing the functions (resulting in remote thread/queuing APC/process creation)

The two tables below summarize the techniques of each task with their evidence, pros, and cons. Later, we will elaborate on each method.

Reading ntdll.dll

Methods	Pros	Cons	Evidence	Malware
Dual load – Calling NtMapViewOfSection with a section containing a fresh copy of ntdll.dll from the disk	Looks like ntdll.dll was loaded in a typical way by the Windows loader	Can be spotted in loaded modules	Can be traced by a hook on NtMapViewOfSection	LockPoS, Flokibot, Osiris, CoinMiner
Reading from the disk – using NtReadFile and manually mapping a fresh copy of ntdll.dll from the disk	The mapping is done manually, hence we can't find ntdll.dll in loaded modules using a debugger	Reading ntdll.dll from the disk is suspicious since every process	Can be traced by a hook on NtReadFile	Trickbot, Formbook, Fashcash

		has it loaded already		
Reading the existing ntdll.dll that is already loaded to the process	Undetected by hooks, leaves no evidence	Fails if the functions are hooked by a security product	none	Neurevt

Performing the functions

Methods	Pros	Cons	Evidence	Malware
Process creation (hollowing)	Creating legitimate Windows processes that can perform malicious activities without being blocked. Low accuracy in identifying which process had injection into it	Can be monitored easily in the kernel	Process creation via the kernel	Flokibot, Trickbot, Neurevt, CoinMiner, Osiris
Remote thread	Without creating a process, execute code from the original Windows processes	Can be monitored easily in the kernel. Greater accuracy in identifying which process has injection into	Remote thread creation via the kernel	LockPoS

Queuing user APC	No process or thread creation – stealthiest method. Low accuracy in identifying which process had injection into it	Harder to implement. Need to find an alertable thread	The malware has a handle to a thread to THREAD_SET_CONTEXT rights	Formbook

Reading ntdll.dll

We know that the malware needs to obtain the system calls numbers. The system calls numbers exist in ntdll.dll, at the "mov eax,SYSCALL_NUMBER" command at the corresponding function. These numbers are changing from one operating system version to another (including service packs), and therefore can't be hardcoded in the malware. This dll is loaded to every process in the system. Hence – loading another copy of it might be suspicious.

It is not a good idea to use the originally loaded ntdll.dll, because If these functions are hooked by a security product, it will not be able to find them easily since the function prologue is changed and the "mov" command is no longer in the same location.

A snippet of NtMapViewOfSection API call from Windows 7 32-bit (syscall number 0xa8) and Windows 10 64-bit (0x28), shows system call numbers are different from one operating system to another.

Figure 2 – NtMapViewOfSection in ntdll.dl, Windows 7 32-bit

Figure 3 – NtMapViewOfSection in ntdll.dll, Windows 10 64-bit

In the table above, we showed 3 different techniques to achieve this:

- Dual load – Using NtMapViewOfSection to map a fresh copy from the disk of ntdll.dll
- Reading from the disk a fresh copy of ntdll.dll using NtReadFile
- Reading the existing ntdll.dll

Let's analyze each method in the context of this malware:

The first method is to call NtMapViewOfSection with a section that contains a fresh copy of ntdll.dll. This section object is created using NtCreateSection and uses a file handle to ntdll.dll which can be obtained using NtCreateFile.

This way, we will see another loaded module of ntdll.dll in the list of loaded modules.


Malware Mitigation Osiris loaded modules after it loaded a fresh copy of ntdll.dll

Figure 4 – A screenshot from Osiris loaded modules, ***after it loaded a fresh copy of ntdll.dll***

CoinMiner also uses this technique:


Malware Mitigation call to NtMapViewOfSection with the section handle 0x98 of ntdll.dll

Figure 5 – The call to NtMapViewOfSection with the section handle 0x98 of ntdll.dll. The section handle is placed on the stack in the top-right image.

The same result can be achieved using a memory-mapped file, as LockPoS & Flokibot do:


Malware Mitigation LockPoS mapping ntdll.dll as a memory mapped file using MapViewOfFile

Figure 6 – LockPoS mapping ntdll.dll as a memory mapped file using MapViewOfFile

Inside MapViewOfFile, there is a call to NtMapViewOfSection. Essentially – calling MapViewOfFile is subsequently calling NtMapViewOfSection.

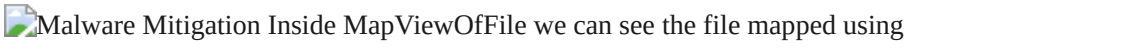
Malware Mitigation Inside MapViewOfFile we can see the file mapped using NtMapViewOfSection

Figure 7 – from kernelbase.dll – Inside MapViewOfFile we can see the file mapped using NtMapViewOfSection


Malware Mitigation Process Monitor shows reading of ntdll.dll by the malware

Figure 8 – Process Monitor shows reading of ntdll.dll by the malware

The second method is reading ntdll.dll from the disk using ReadFile or NtReadFile. Thanks to SysInternals Process Monitor we can easily spot this activity:

Trickbot reads the whole file multiple times, each time for a different system call number.

Formbook reads the whole file once for a few system calls.

Fastcash reads just a specific part of the file. It looks for NtQueryInformationProcess function – it can be used to detect debuggers. This seems to be the stealthiest way. If we thought we could recognize this behavior solely based on some signature (like, offset 0 and length = size of ntdll.dll) we were wrong!

Note the 2nd read of ntdll.dll, marked in yellow – it was not initiated by Formbook, but by the kernel as a result of page fault (Paging I/O flag indicates this). So, this call is not relevant to our analysis of the malware.

The third and least effective method, which is implemented only by Neurevt, is reading from ntdll.dll that is already loaded into the process memory.

One of the functions that Neurevt looks for is NtCreateSection. On this function, we don't have a hook, so it will successfully copy its syscall number (not only that but also the next instructions of that function) to another memory region:

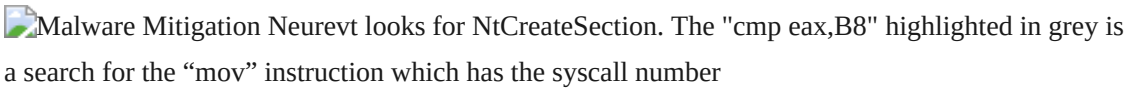
Malware Mitigation Neurevt looks for NtCreateSection. The "cmp eax,B8" highlighted in grey is a search for the "mov" instruction which has the syscall number

Figure 9 – Neurevt looks for NtCreateSection. The "cmp eax,B8***" highlighted in grey is a search for the "***mov***" instruction which has the syscall number***

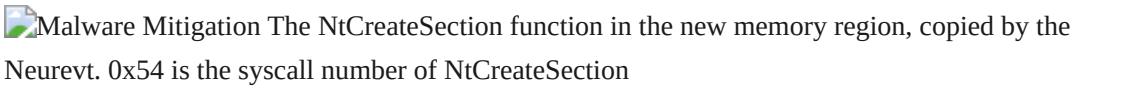
Malware Mitigation The NtCreateSection function in the new memory region, copied by the Neurevt. 0x54 is the syscall number of NtCreateSection

Figure 10 – The NtCreateSection function in the new memory region, copied by the Neurevt. 0x54 is the syscall number of NtCreateSection

But on NtUnmapViewOfSection we do have a hook, so the checks that Neurevt performs in order to see if it can copy the function will fail and therefore it won't be able to retrieve its syscall number.

This is what the hooked UnmapViewOfSection looks like:


Malware Mitigation NtUnmapViewOfSection hooked by Cyberbit EDR agent

Figure 11 – NtUnmapViewOfSection


Malware Mitigation The cmp eax,B8 check fails an the function will not be copied

Figure 12 – The "cmp eax,B8***" check fails and the function won't be copied***

Neurevt has another mechanism – to call the APIs if the copy fails. But in this case, it will be intercepted by the hooks.

This method has one advantage – if the API wasn't hooked by the security product, the malware will be able to use the system calls without reading ntdll.dll from the disk – and we will not leave any evidence.

Performing the functions (resulting in remote thread/queuing APC/process creation)

There are many syscall numbers which can be extracted. We will focus on the ones related to code injection/process hollowing – as powerful mechanisms that are favored by malware authors. The functions performed after extracting ntdll.dll can perform code injection. This results, as mentioned in the table above, in a remote thread creation, queuing an APC to a remote process or process creation.

First case – process creation

Flokibot, Trickbot, Neurevt, CoinMiner, and Osiris – all create a suspended process, replace its code with another code (process hollowing) and then resume its main thread. They use the direct system calls to perform it.

Flokibot, Neurevt, CoinMiner, and Osiris – all chose a legitimate Windows process for hollowing. Flokibot and Neurevet chose explorer.exe. CoinMiner chooses notepad.exe. Osiris (which uses a combination of process hollowing and process Doppelgänging) chooses wermgr.exe. Neurevt, as part of its unpacking process, created itself as a suspended process and then unpacked its code into its child (in addition to the injection to explorer.exe).

Trickbot created itself as a suspended process and injected its code into it (as did Neurevt).

We can learn that the suspended process will be a legitimate windows process or the malware itself or a web browser (Firefox/Chrome/Edge/Internet Explorer, etc.)

This is logical because Windows processes look legitimate to the user and web browsers and can send network packets, so it won't look suspicious. As for the malware itself – it also doesn't look suspicious, as it is common to spawn child processes of your own process (example: chrome.exe does it) and the malware knows that its executable definitely exists.

Second case – thread creation

If a process was not created, a remote thread might be created.

LockPoS, which we have written about extensively (How Cyberbit Researchers Discovered a New Silent LockPoS Malware Injection Technique, Jan 2018)_, creates a remote thread in a remote process. Not surprisingly – in explorer.exe (malware's absolute favorite).

Third case – user APC queued

Formbook, the most complex malware on the list, injects code into explorer.exe using an APC. But to inject an APC, a process must obtain a handle to that remote thread with THREAD_SET_CONTEXT access rights. This handle is a handle to a thread which does not belong to a child process of the malware.

One malware which was not covered here is Fastcash. This malware didn't perform code injection. However, we learned from this malware that malware might not read ntdll.dll entirely, Rather only a part of it.

Malware Mitigation

We have two pieces of evidence:

The reading of ntdll.dll which splits into 3 cases. The 3rd case (reading from the existing ntdll.dll) is not relevant because we are expected to place hooks on the calls related to code injection.

This leaves us left with 2 cases – 2 methods. We can intercept the reading of ntdll.dll in two ways:

1. Place a hook on NtMapViewOfSection on a process trying to map ntdll.dll to its own memory
2. Monitor reading of at least one byte from ntdll.dll via the kernel. The reading must not be a result of page fault

The second evidence is one of the three below:

1. Process creation – particularly of the malware itself/Windows process/web browser
2. Thread creation in a remote process
3. Queuing an APC in a remote process (not a child process)

All three cases can be monitored via the kernel without hooks.

For the first one, we have to monitor process creation and check its properties (the malware itself, Windows process, web browser)

For the second one, we have to monitor remote thread creation.

For the last one, we have to monitor handles obtained to a remote thread with THREAD_SET_CONTEXT access rights. That thread belongs to a process which is not a child process of the malware.

The mitigation can be done by combining the two pieces of evidence.

If a process is seen reading ntdll.dll as described above and performs one of the 3 cases from the second set of evidence (process creation, thread creation in a remote process or queuing an APC in a remote process) right afterward, we can mark it as suspicious. The process that has the injection into can be known from the second evidence.

Two remarks:

- We do not know with 100% certainty which process had the injection into it. Let's explain each case:

1. **Remote thread** – We can be almost 100% sure about the targeted process. We just miss the allocation and write into that process, because they were done using system calls.
2. **Process creation** – We cannot be sure about it since the malware might create other processes without injecting into them (such as executing malicious commands via Windows command prompt)
3. **Queuing APC** – We can also not be sure about it since having a handle to a thread doesn't guarantee that an APC was queued successfully.

- We can remove the condition of Windows process/web browser/the malware itself from evidence two; case one. This will lead to fewer false-negatives and probably also lead to more false-positives

Hod Gavriel is a Malware Analyst at Cyberbit.

Watch FREE webcast to learn How to Prevent the Next Financial Cyberattack with Next-Gen Technology

---