Syscalls - Userland Hooking

Introduction

Host-based security solutions frequently perform API hooking on syscalls to enable analysis and monitoring of programs at runtime. For instance, by hooking the `NtProtectVirtualMemory` syscall, the security solution can detect higher-level WinAPI calls such as `VirtualProtect`, even when it is concealed from the import address table of the binary. Furthermore, security solutions can access any memory region that is set to executable and scan it in search of signatures. Userland hooks are generally installed before the `syscall` instruction, which is the last step for a syscall function in user mode.

Kernel mode hooks can be implemented post-execution, after the flow is transferred to the kernel, however, Windows Patch Guard and other mitigations make it difficult for third-party applications to patch kernel memory, making the task difficult if not impossible. Placing kernel mode hooks may also result in stability implications and cause unexpected behavior, which is why it is rarely implemented.

Showcasing Userland Hooking

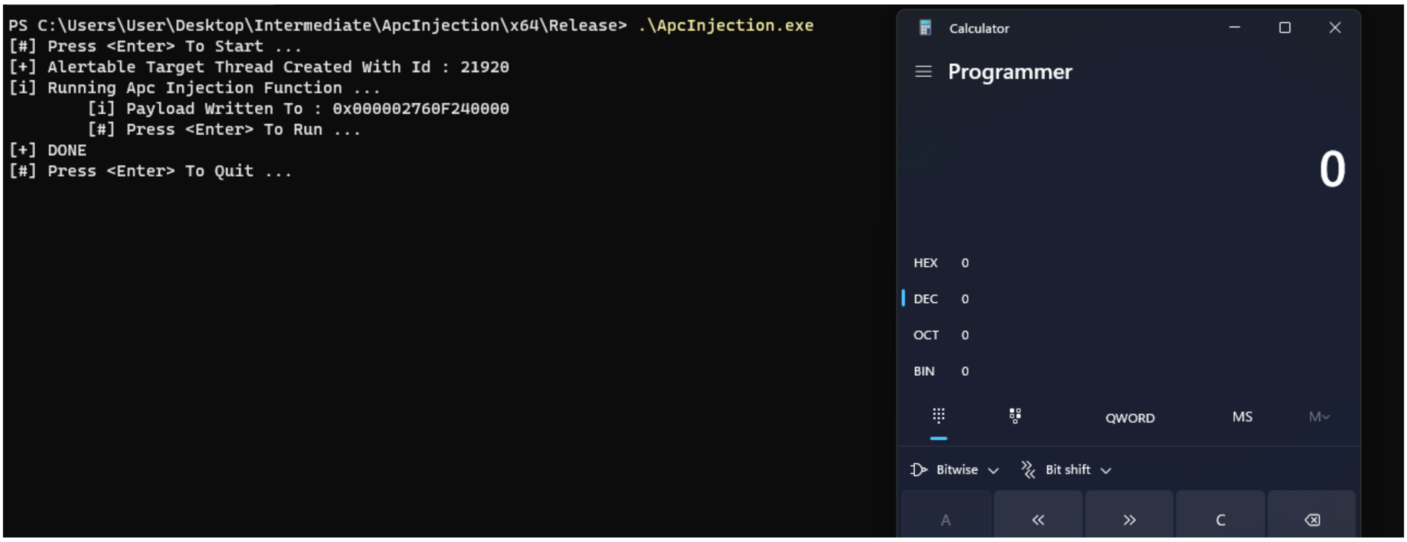
This section utilizes a DLL file which, when injected into a process, will use the Minhook Library to install a hook on `NtProtectVirtualMemory` in order to gain insight into the operations of EDRs about syscall hooking. The hook installed is equipped with the capability of dumping the memory's contents if it is set to be executable (`RX` or `RWX`). Furthermore, the process will be terminated if a `RWX` memory region is detected.

The DLL source code is available for download for testing purposes. It is not necessary to understand the code at this time, however, it contains extensive comments to make it easier to understand.

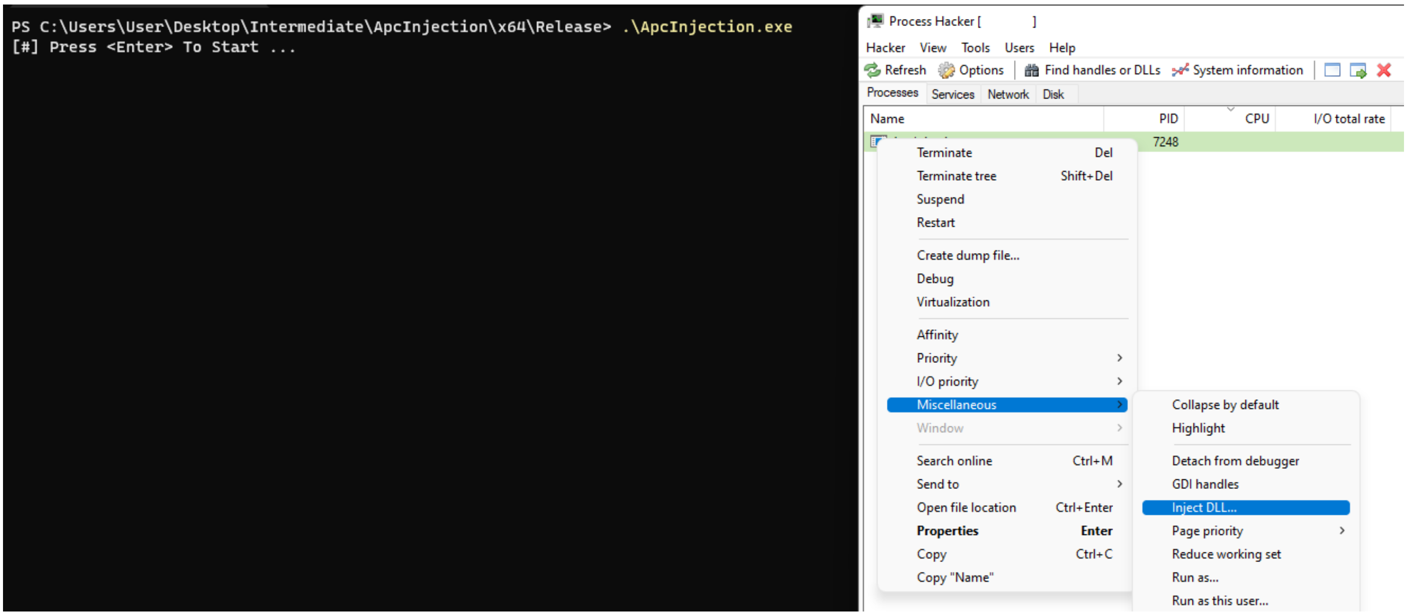
EDR Hooking Demonstration

This section demonstrates how an EDR can block the execution of a certain payload using syscall hooking. The *APC Injection* code will be the malicious binary in this demo.

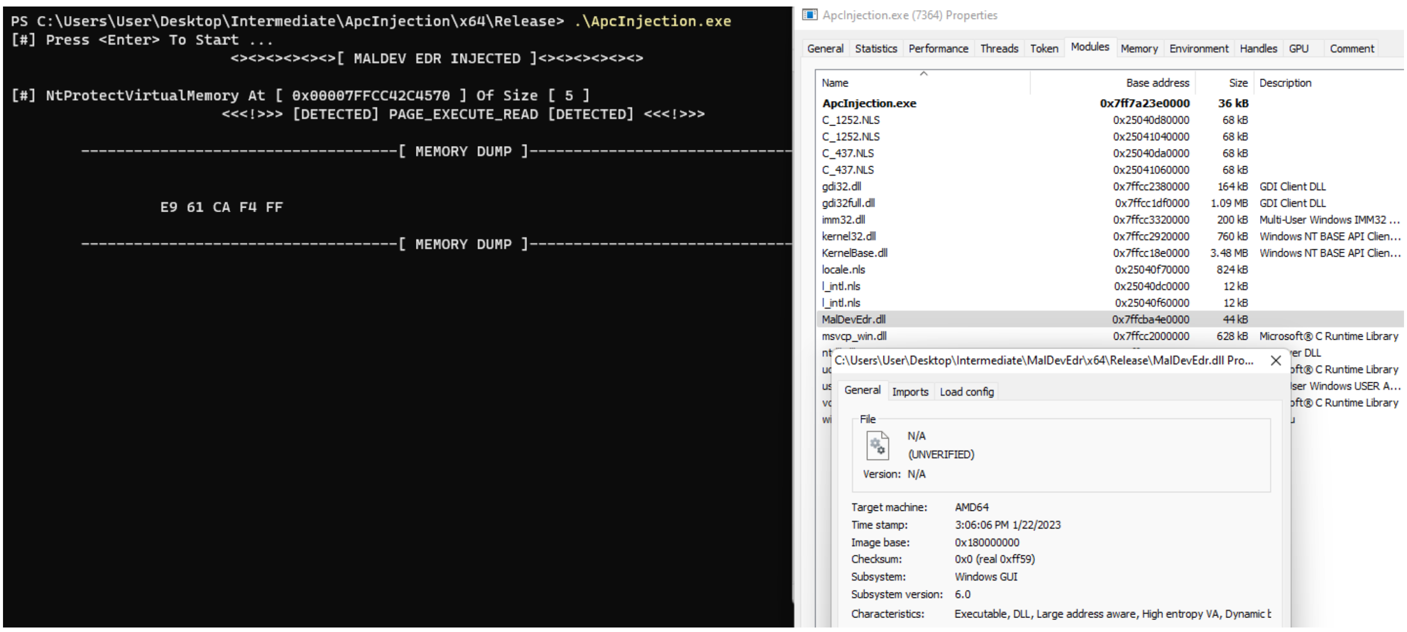
1.Running the program without hooking `NtProtectVirtualMemory`.



2.Injecting *MalDevEdr.dll* into ApcInjection.exe using Process Hacker

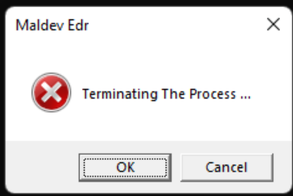


3.The DLL is injected, and it detects `RX` (this is related to the DLL injection)



4.Pressing the Enter key on the ApcInjection.exe console, triggers a call to `NtProtectVirtualMemory`, setting `0x0000025041080000` as `RWX` memory, this address is then dumped by the DLL to the screen. The content that was dumped is the Msfvenom calc payload.

```
PS C:\Users\User\Desktop\Intermediate\ApcInjection\x64\Release> .\ApcInjection.exe
[#] Press <Enter> To Start ...
                         <><><><><><>[ MALDEV EDR INJECTED ]<><><><><><>

[#] NtProtectVirtualMemory At [ 0x00007FFCC42C4570 ] Of Size [ 5 ]
                        <<<!>>> [DETECTED] PAGE_EXECUTE_READ [DETECTED] <<<!>>>

        ------------------------------------[ MEMORY DUMP ]------------------------------------


                 E9 61 CA F4 FF

        ------------------------------------[ MEMORY DUMP ]------------------------------------


[+] Alertable Target Thread Created With Id : 9032
[i] Running Apc Injection Function ...
        [i] Payload Written To : 0x0000025041080000
[#] NtProtectVirtualMemory At [ 0x0000025041080000 ] Of Size [ 272 ]
                        <<<!>>> [DETECTED] PAGE_EXECUTE_READWRITE [DETECTED] <<<!>>>

        ------------------------------------[ MEMORY DUMP ]------------------------------------


                 FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51
                 56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52
                 20 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0
                 AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED
                 52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88
                 00 00 00 48 85 C0 74 67 48 01 D0 50 8B 48 18 44
                 8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48
                 01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1
                 38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44
                 8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49
                 01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A
                 41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41
                 59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00
                 00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 8B
                 6F 87 FF D5 BB E0 1D 2A 0A 41 BA A6 95 BD 9D FF
                 D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47
                 13 72 6F 6A 00 59 41 89 DA FF D5 63 61 6C 63 00

        ------------------------------------[ MEMORY DUMP ]------------------------------------
```



Explanation

When `ApcInjection.exe` uses `VirtualProtect` with a `PAGE_EXECUTE_READWRITE` argument, it's intercepted by `MalDevEdr.dll`. `MalDevEdr.dll` will use the base address passed to `VirtualProtect` to dump the contents of that memory region. Since the memory region is being changed to `RWX`, `MalDevEdr.dll` terminates the program and blocks the payload from being executed, which is something Windows Defender Antivirus was not able to do.

This proof of concept demonstrates the power of API hooking in detecting and monitoring a program at runtime. In real-world scenarios, EDRs will typically hook a wider range of syscalls, enhancing their ability to detect malicious actions.

Bypassing Userland Syscall Hooks

Using syscalls directly is one method of bypassing userland hooks. For example, using `NtAllocateVirtualMemory` instead of the `VirtualAlloc/Ex` WinAPIs when allocating memory for the payload. There are other several ways that syscalls can be called stealthily:

- Using Direct Syscalls
- Using Indirect Syscalls
- Unhooking

Direct Syscalls

Evasion of userland syscall hooking can be achieved by obtaining a version of the syscall function coded in the assembly language and calling that crafted syscall directly from within the assembly file. The challenge lies in determining the syscall service number (SSN), as this number varies from one system to another. To overcome this, the SSN can be either hard-coded in the assembly file or calculated dynamically during runtime. A sample crafted syscall in an assembly file (`.asm`) is presented below.

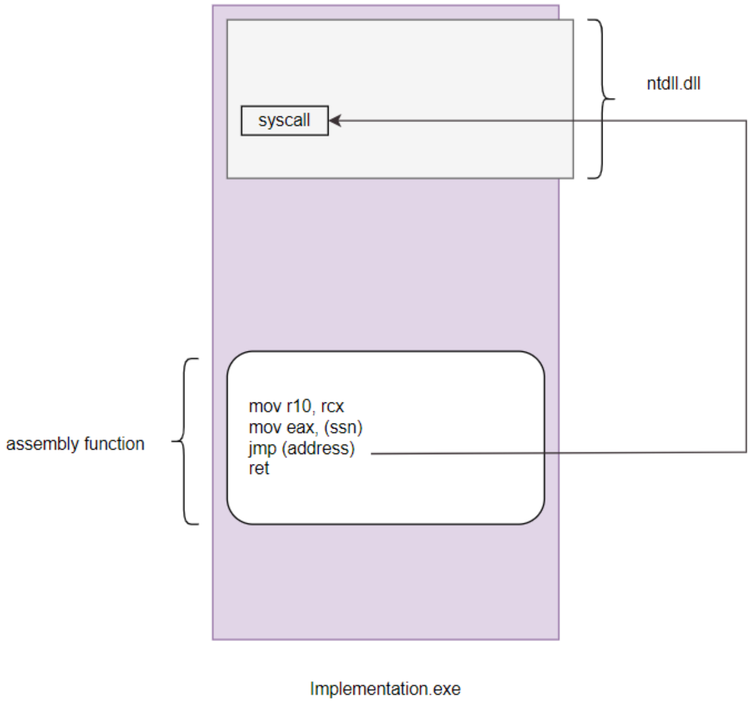
Rather than calling `NtAllocateVirtualMemory` with `GetProcAddress` and `GetModuleHandle` as previously done in this course, the assembly function below can be utilized for the same result. This eliminates the need to call `NtAllocateVirtualMemory` from within the NTDLL address space where hooks are installed, thereby avoiding the hooks.

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtAllocateVirtualMemory)
    syscall
    ret
NtAllocateVirtualMemory ENDP

NtProtectVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtProtectVirtualMemory)
    syscall
    ret
NtProtectVirtualMemory ENDP

// other syscalls ...
```

This method is utilized in tools such as SysWhispers and HellsGate both of which are discussed in upcoming modules.

Indirect Syscalls

Indirect syscalls are implemented similarly to direct syscalls where the assembly files must be manually crafted first. The distinction lies in the absence of the `syscall` instruction within the assembly function, which is instead jumped to. A visual representation is shown below.



The assembly functions for `NtAllocateVirtualMemory` and `NtProtectVirtualMemory` are shown below.

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtAllocateVirtualMemory)
    jmp (address of a syscall instruction)
    ret
NtAllocateVirtualMemory ENDP

NtProtectVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtProtectVirtualMemory)
    jmp (address of a syscall instruction)
    ret
NtProtectVirtualMemory ENDP

// other syscalls ...
```

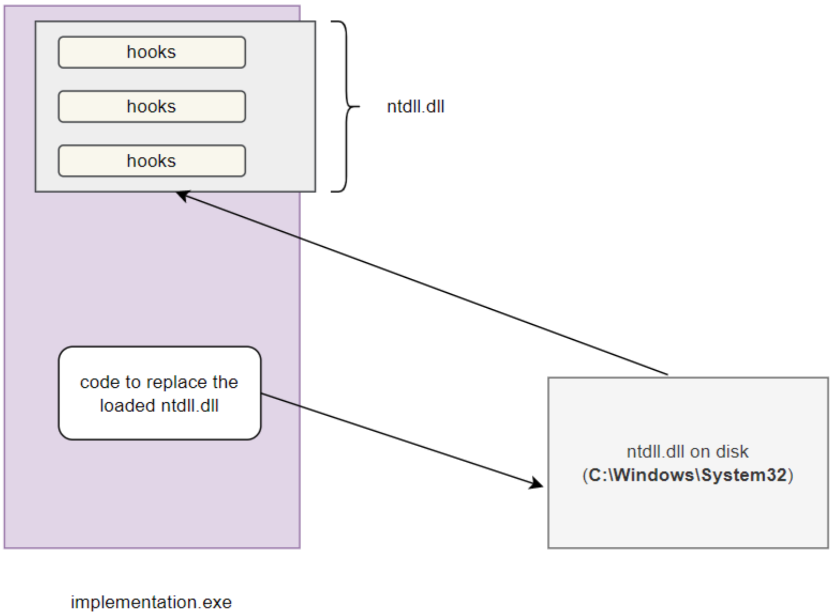
Indirect Syscalls Benefit

The benefit of performing indirect syscalls over direct syscalls is that security solutions will look for syscalls being called from outside of the NTDLL address space and consider them suspicious. With indirect syscalls, the syscall instruction is being executed from NTDLL's address space as how normal syscalls should be. Therefore, indirect syscalls are more likely to slip past security solutions than direct syscalls.

Indirect syscalls will be covered in the advanced modules.

Unhooking

Unhooking is another approach to evade hooks in which the hooked NTDLL library loaded in memory is replaced with an unhooked version. The unhooked version can be obtained from several places, but one of the common approaches is to load it directly from disk. Doing so will remove all the hooks placed inside the NTDLL library.



Unhooking will be covered in the advanced modules.