(Anti-)Anti-Rootkit Techniques - Part I: UnKovering mapped rootkits

eversinc33.com/posts/anti-anti-rootkit-part-i/

eversinc33

bits about malware development and penetration testing

```
                                                                  o88
oooooo      oooooo
 oooooooo8 oooo    oooo oooooooo8 oo ooooo     ooooooo8 oooo   oo ooooo     ooooooo
o88     888o o88     888o
888oooooo8    888    888 888oooooo8    888     888 888ooooooo  888    888    888 888
888    88888o       88888o
888            888 888   888           888                 888 888    888    888 888
88o     o888 88o     o888
  888oooo888      888       888oooo888 o888o        88ooooooo8 o888o o888o o888o  88ooo888
88ooo88      88ooo88
```

Mar 23, 2024

14 min read

While some blog posts exist that talk about developing offensive drivers and rootkits, the only ones that I found, which really talk about anti-rootkit evasion, are those related to game cheating. After spending some time developing my rootkit Banshee, I started to become interested in anti-rootkits, their detection mechanisms and of course the various methods to

evade them. To have a transparent environment to test my rootkits evasion abilities, I developed a small anti-rootkit tool called unKover, that implements some techniques to detect rootkits, especially those manually mapped to memory.

This blog post is part I of a series, where I plan to showcase various anti-rootkit techniques, known through anti-rootkits or anti-cheats, and their implementations in unKover.

Before getting into the detection mechanisms, I will first have to briefly talk about manual driver mapping, which is what we want to detect in this first part.

- DISCLAIMER: I am neither a professional Windows kernel developer nor an expert on anti-rootkit technology - this is all personal research done in my free time. If you have anything to correct or add, please shoot me a message on X and let me know
- FURTHER DISCLAIMER: I am also aware that open-source anti-cheats exist, such as ac with much more sophisticated detections. My goal was however to implement them myself to learn more and have a small, dedicated tool that fits my needs. Please check out the various open source anti-cheats on GitHub.
- FINAL DISCLAIMER: Yes, this is babby first anti-rootkit evasion. But in the end, this is just me learning and sharing my journey as a kernel noob.

With that out of the way, let's get going.

Prelude

If you are developing a rootkit or offensive driver, chances are that you load it on your development machine with testsigning enabled. In a real world scenario however, especially in those environments where you might want to deploy a rootkit, enabling testsigning on the target machine is usually not the best idea. If you don't have a valid certificate, or want to use a leaked one, to sign your driver, you will have to resort to manually mapping your driver to load it on the target machine.

For those familiar with userland malware, the concept of manually mapping drivers is similar to that of reflective PE loading - you are not loading the program from disk, but rather manually laying out the image of it into memory. A driver however has to be mapped to kernel memory, which means we have to already have some kind of write-primitive in ring 0. Which is where BYOVD and LOLDrivers come into play - if we can exploit a signed, but vulnerable, driver to write arbitrary data into kernel space, we can write the image of our rootkit driver to memory as if it was legitimately loaded.

Manually mapping drivers with kdmapper

The arguably most well known tool to map a driver into memory is kdmapper, which exploits the vulnerable `iqvw64e.sys` driver from intel to write an arbitrary driver into the kernel. Now of course, this loldriver can and will be blacklisted in some environments. You can however

replace the memory primitives with any other vulnerable driver, preferably one that only your red team knows of, and still use kdmapper to deploy your rootkit.

The process of manual mapping, as I described above, is very similar to reflective PE injection, as in the end, a driver is only another PE. E.g., writing sections to memory, resolving imports and applying relocations, erasing the image headers for stealth and finally calling the entrypoint of the driver.

In addition to this mapping process, kdmapper also takes care of erasing traces of the intel driver being loaded. Describing this is not the goal of this post, since this is something the mapper and not the rootkit should take care of, but it e.g. involves clearing entries from various undocumented data structures such as the PiDDBCacheTable, the MmUnloadedDrivers array, the g_KernelHashBucketList and the `RuntimeDriver*` structs used by the Defender `WdFilter.sys` driver.

So, how can we detect a manually mapped driver? Before the first detection idea, we need to talk about one more topic: driver communication.

Driver communication

Usually, drivers, as well as rootkits, communicate over `IOCTL` codes - control messages that are sent to a device through the `DeviceIoControl` API over a device handle. For the user mode program to acquire such a handle, the driver has to register a device object that the user mode program can use to call `CreateFile`. In a driver, this usually looks as follows:

```
NTSTATUS
DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath)
{
    IoCreateDevice(
        pDriverObject,
        0,
        &usDriverName,
        FILE_DEVICE_UNKNOWN, // not associated with any real device
        FILE_DEVICE_SECURE_OPEN,
        FALSE,
        &pDeviceObject
    );
}
```

This creates an object in the Windows Object Manager for the driver, usually in the form of `\Device\<Name>`. Now a client could call `CreateFileA("\\Device\\Rootkit", GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL)` to open a handle to the rootkit and subsequently send `IOCTL` codes to this handle to control the rootkit.

This however means that our rootkit, if we choose this standard way of Usermode/Kernelmode communication, will show up in the Windows object manager. But how would an anti-rootkit tell this apart from a legit driver? Well, if it's manually mapped, we can simply look at all device objects and check if their image in kernel memory is actually backed by a valid module. This is detection number 1) which we will cover in this post.

Detection 1: Querying device objects

To query device objects, a series of API calls is first needed to get the `\Driver` directory object:

```
/* Code skeleton stolen from https://github.com/not-wlan/driver-
hijack/blob/master/memedriver/hijack.cpp#L136 */

// Get Handle to \Driver directory
InitializeObjectAttributes(&attributes, &directoryName, OBJ_CASE_INSENSITIVE, NULL,
NULL);
ZwOpenDirectoryObject(&handle, DIRECTORY_ALL_ACCESS, &attributes);

// Get the object from the handle
ObReferenceObjectByHandle(handle, DIRECTORY_ALL_ACCESS, nullptr, KernelMode,
&directory, nullptr);
POBJECT_DIRECTORY directoryObject = (POBJECT_DIRECTORY)directory;
```

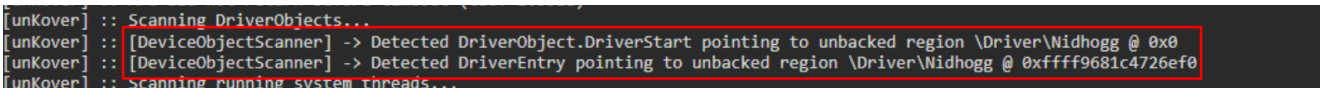
With this object, we can now start to iterate over each device object. The object manager actually organizes objects in a hashbucket with 37 entries (for technical details, see: https://www.informit.com/articles/article.aspx?p=22443&seqNum=7).

```
// acquire the lock for accessing the directory object
KeEnterCriticalRegion();
ExAcquirePushLockExclusiveEx(&directoryObject->Lock, 0);

// iterate over the hashbucket
for (POBJECT_DIRECTORY_ENTRY entry : directoryObject->HashBuckets)
{
        if (!entry)
                continue;

        // iterate over each hashbuckets entries items
        while (entry != nullptr && entry->Object)
        {
                PDRIVER_OBJECT driver = (PDRIVER_OBJECT)entry->Object;

                /*
                 * We are simply checking if the driver entry (DriverInit) memory
                 * resides inside of one of the loaded modules address spaces.
                 */
                if (GetDriverForAddress((ULONG_PTR)driver->DriverInit) == NULL)
                {
                        LOG_MSG("[DeviceObjectScanner] -> Detected DriverEntry
pointing to unbacked region %ws @ 0x%llx\n",
                                driver->DriverName.Buffer,
                                (ULONG_PTR)driver->DriverInit
                        );
                }

                entry = entry->ChainLink;
        }
}

// Release lock when done
ExReleasePushLockExclusiveEx(&g_hashBucketLock, 0);
KeLeaveCriticalRegion();
```

The implementation to check if a memory address is inside of a loaded module's address space is rather simple: we iterate over the `DriverSection->InLoadOrderLinks` linked list, which contains a `KLDR_DATA_TABLE_ENTRY` for each loaded driver (somewhat similar to the Usermode `InLoadOrderModuleList` from the `PEB` we all know and love). Here, we check if the address resides in one of those modules - if it doesn't belong to any module, it is manually mapped to memory.

```
PKLDR_DATA_TABLE_ENTRY
GetDriverForAddress(ULONG_PTR address)
{
        if (!address)
        {
                return NULL;
        }

        PKLDR_DATA_TABLE_ENTRY entry = (PKLDR_DATA_TABLE_ENTRY)(g_drvObj)-
>DriverSection;

        for (auto i = 0; i < 512; ++i)
        {
                UINT64 startAddr = UINT64(entry->DllBase);
                UINT64 endAddr = startAddr + UINT64(entry->SizeOfImage);
                if (address >= startAddr && address < endAddr)
                {
                        return (PKLDR_DATA_TABLE_ENTRY)entry;
                }
                entry = (PKLDR_DATA_TABLE_ENTRY)entry->InLoadOrderLinks.Flink;
        }

        return NULL;
}
```

If we now load a rootkit driver based on IOCTLs, such as Nidhogg via kdmapper and run unKover, we can quickly see the DeviceObject scanner uncover the Nidhogg driver as an unbacked kernel memory region:



While several ways to evade this simple detection exist, one of the easier ones is to change the method of Usermode/Kernelmode communication and go away from `IOCTL` based communication, so that we don't even need to register a device anymore and thus won't be visible in the object manager at all. You could do anything here, but some obvious things that come to mind are communication over shared memory or named pipes. In Banshee, I am currently using shared memory, that is shared between the rootkit driver and the Usermode client (I know, I know, hooked pointers are cooler and better, but step by step).

Now with a named pipe or a shared memory that is continuosly read by our rootkit that waits for new commands (think `while true: ReadCommandFromSharedMemory()`), a new detection arises - the anti-rootkit simply has to identify the thread, by analyzing thread callstacks for frames pointing to unbacked memory.

Detection 2: Detecting unbacked system threads with APCs

Now again, there are a myriad of methods to do the above. One method (as e.g. used by the `BattleEye` anti-cheat) to detect mapped cheat drivers, is to queue an APC to all system threads that unwinds the stack frames of each thread. Then, the anti-cheat can check for each frame's address if it points to unbacked memory - if it does, we have caught a potential rootkit/cheat thread.

Now APCs in the Kernel are a super complex topic. A blog I found useful in implementing this was the APC series by Ori Damari.

First, the APC that analyzes a thread's call stack has to be defined (lots of code omitted for brevity):

```
VOID
CaptureStackAPC(
        IN PKAPC Apc,
        IN OUT PKNORMAL_ROUTINE* NormalRoutine,
        IN OUT PVOID* NormalContext,
        IN OUT PVOID* SystemArgument1,
        IN OUT PVOID* SystemArgument2
)
{
        // allocate memory for the stack frames
        PVOID* stackFrames = (PVOID*)ExAllocatePoolWithTag(NonPagedPoolNx,
MAX_STACK_DEPTH * sizeof(PVOID), POOL_TAG);
        // zero the memory
        RtlSecureZeroMemory(stackFrames, MAX_STACK_DEPTH * sizeof(PVOID));

        /*
         * Capture the stack trace.
         * All the heavy lifting is done by RtlCaptureStackBackTrace, which
         * unwinds the stack for us and gives back a pointer of
         */
        USHORT framesCaptured = RtlCaptureStackBackTrace(0, MAX_STACK_DEPTH,
stackFrames, NULL);

        // Stack trace analysis...
        for (auto i = 0; i < framesCaptured; ++i)
        {
                // Check if address of frame is from unbacked memory
                ULONG_PTR addr = (ULONG_PTR)stackFrames[i];
                if (GetDriverForAddress(addr) == NULL)
                {
                        DbgPrint("[APCStackWalk] -> Detected stack frame pointing to
unbacked region: TID: %lu @ 0x%llx\n", HandleToUlong(PsGetCurrentThreadId()), addr);
                }
        }

        if (stackFrames) { ExFreePoolWithTag(stackFrames, POOL_TAG); }

        // Free the APC and signal that the APC is done
        ExFreePoolWithTag(Apc, POOL_TAG);
        KeSetEvent(&g_kernelApcSyncEvent, 0, FALSE);
}
```

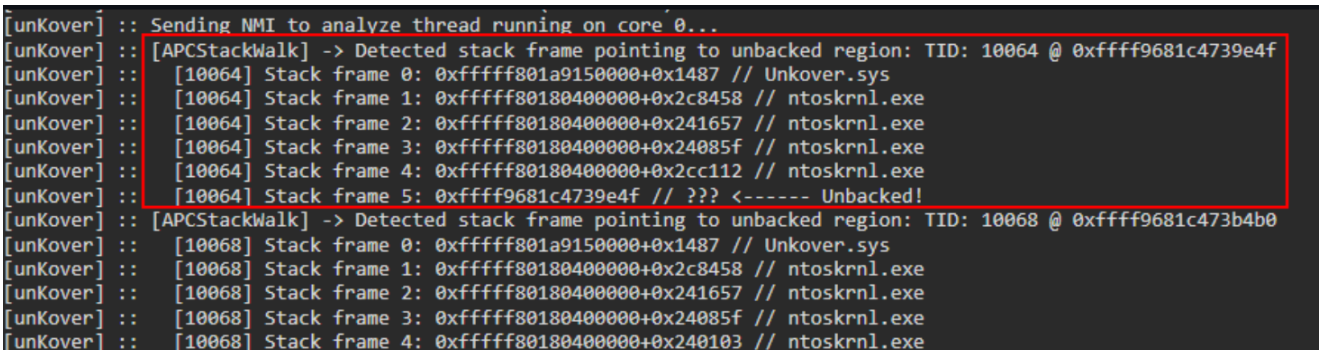
As seen, the hard work is done by `RtlCaptureStackBackTrace`, which unwinds stack frames for us and makes stack analysis a breeze.

Now we simply have to queue this APC to all system threads.

```
VOID
APCStackWalk()
{
        KeInitializeEvent(&g_kernelApcSyncEvent, NotificationEvent, FALSE);

        // Queue APCs to system threads. System thread IDs are a multiple of 4.
        // (Usually at least. See:
https://devblogs.microsoft.com/oldnewthing/20080228-00/?p=23283)
        for (auto tid = 4; tid < 0xFFFF; tid += 4)
        {
                PETHREAD ThreadObj;

                // Get ETHREAD object for TID
                if (!NT_SUCCESS(PsLookupThreadByThreadId(UlongToHandle(tid),
&ThreadObj)))
                {
                        continue;
                }

                // Ignore current thread and non system threads
                if (!PsIsSystemThread(ThreadObj) || ThreadObj ==
KeGetCurrentThread())
                {
                        ObDereferenceObject(ThreadObj);
                        continue;
                }

                // Initialize APC
                PKAPC apc = (PKAPC)ExAllocatePoolWithTag(
                        NonPagedPool,
                        sizeof(KAPC),
                        POOL_TAG
                );
                KeInitializeApc(apc,
                        ThreadObj,
                        OriginalApcEnvironment,
                        CaptureStackAPC,
                        RundownAPC, // Empty APC routine
                        NormalAPC,  // Empty APC routine
                        KernelMode,
                        NULL
                );

                // Queue APC
                NTSTATUS NtStatus = KeInsertQueueApc(apc, NULL, NULL,
IO_NO_INCREMENT);

                // Wait for event to signal that the apc is done before queueing the
next one
                LARGE_INTEGER timeout;
                timeout.QuadPart = 2000; // 2 second wait timeout
                NtStatus = KeWaitForSingleObject(&g_kernelApcSyncEvent, Executive,
```

```
KernelMode, FALSE, &timeout);
                KeResetEvent(&g_kernelApcSyncEvent);

                // Clean up
                if (ThreadObj) { ObDereferenceObject(ThreadObj); }
    }
}
```

If we now run this with Banshee, which uses Shared Memory KM/UM communication and a system thread that reads commands, we will see the thread get found almost immediately:



Thus, even though we did not register a device object, we got clapped by unKover due to our system thread that originates from unbacked memory.

Again, there are many ways to circumvent this detection: one of them of course being stack spoofing, so that we pretend to not be in unbacked memory. Another technique is based on Direct Kernel Object Modification (DKOM) of the `KTHREAD` object of our system thread - if we set its `ApcQueueable` bit to `0`, we effectively disallow any APCs being queued on our thread (https://www.unknowncheats.me/forum/anti-cheat-bypass/587069-disable-apc.html) - this is a feature used e.g. by `KeEnterCriticalRegion` (Even if an anti-rootkit could flip this bit back - this is very invasive and greatly puts the operating systems stability at risk, if APCs start getting queued in critical code regions). Keep in mind that `KTHREAD` is an undocumented structure which highly differs from Windows version to Windows version.

Now with our thread safe from APCs, are we safe from being detected?

Detection 3: Non-Maskable-Interrupts

NMIs are Non-Maskable-Interrupts, which means they are hardware-driven interrupts that are sent to a CPU, which can not be masked (i.e. prevented from occuring). In Windows, the `HalSendNmi` API can be used to send an NMI to a CPU core, which will directly interrupt the thread running on that core at the time of the interrupt and invoke NMI callbacks. An NMI callback function can be defined by any kernel driver and thus can be used to inspect the call stack of the thread running on the specific core, just like above with APCs. This means that, if we are lucky, we catch a rootkit thread running on a CPU and can walk the stack to find unbacked memory pointers, if we send enough NMIs from time to time.

First, an NMI callback has to be defined:

```
BOOLEAN
NmiCallback(PVOID context, BOOLEAN handled)
{
        PNMI_CONTEXT nmiContext = (PNMI_CONTEXT)context;
        ULONG procNum = KeGetCurrentProcessorNumber();

        nmiContext[procNum].threadId = HandleToULong(PsGetCurrentThreadId());
        // capture the stack trace
        nmiContext[procNum].framesCaptured = RtlCaptureStackBackTrace(
                0,
                STACK_CAPTURE_SIZE,
                (PVOID*)nmiContext[procNum].stackFrames,
                NULL
        );

        return TRUE;
}
```

Since an NMI should not be running too long, for stability reasons, we save the info to a heap-allocated memory and parse its data in another thread:

```
VOID
AnalyzeNmiData()
{
        for (auto core=0u; core<g_numCores; ++core)
        {
                PETHREAD ThreadObj = NULL;
                NMI_CONTEXT nmiContext = g_NmiContext[core];

                // get the thread object
                PsLookupThreadByThreadId(ULongToHandle(nmiContext.threadId),
&ThreadObj);

                // Check each stack frame for origin
                for (auto i = 0; i < nmiContext.framesCaptured; ++i)
                {
                        ULONG_PTR addr = (ULONG_PTR)(nmiContext.stackFrames[i]);
                        PKLDR_DATA_TABLE_ENTRY driver = GetDriverForAddress(addr);

                        if (driver == NULL)
                        {
                                LOG_MSG("[NmiCallback] -> Detected stack frame
pointing to unbacked region. TID: %u @ 0x%llx", nmiContext.threadId, addr);
                        }
                }

                if (ThreadObj) { ObDereferenceObject(ThreadObj); }
        }
}
```

This logic is almost identical to the logic used in the APC parsing.

Now in a main loop, we periodically send NMIs in hope of catching a thread:

```
VOID
SendNMI(IN PVOID StartContext)
{
        NTSTATUS NtStatus;

        do
        {
                // Register callback
                g_NmiCallbackHandle = KeRegisterNmiCallback(NmiCallback,
g_NmiContext);

                // Fire NMI for each core
                for (auto core=0u; core<g_numCores; ++core)
                {
                        KeInitializeAffinityEx(g_NmiAffinity);
                        KeAddProcessorAffinityEx(g_NmiAffinity, core);
                        HalSendNMI(g_NmiAffinity);
                        // Sleep for 1 seconds between each NMI to allow completion
                        SleepMs(1000);
                }

                // Unregister the callback
                KeDeregisterNmiCallback(g_NmiCallbackHandle);

                // Analyze data
                AnalyzeNmiData();

                SleepMs(5000);

        } while (true);
}
```

Because callbacks could be removed by a rootkit, we make sure to register the callback just before firing the NMI and deregistering it afterwards.

If we let this run for a long enough time, we will sooner or later catch a thread pointing to unbacked memory (although this is not as likely as one would think, if the thread is sleeping most of the time. It is sort of like hoping to catch a beacon while active when doing memory scans, where you mostly only catch the obfuscated memory).

```
        :: Scanning DriverObjects...
[unKover] :: [NmiCallback] -> Detected stack frame pointing to unbacked region. TID: 10068 @ 0xffff9681c4739e60
```

Driver “stomping”

While, again, not using threads in the first place is the best counter-measure for this detection, this raises the bar a lot higher. Still, call stack spoofing would be a viable measure, as well as something that I did not talk about at all. If you are familiar with “module stomping” in userland, you might have already got the idea of “driver stomping”: i.e. loading your driver over another driver that was legitimately loaded and thus pretending to be backed by a module on disk.

This is implemented in a tool called GhostMapper, which maps your driver over the Windows ghost drivers (see the project’s README for more info). Again, this adds other IOCs, such as the driver in memory differing from the driver on disk, which could be verified by an anti-rootkit. With self-modifying drivers, e.g. packed drivers, this however has a potential for false positives.

Conclusion

While I am not sure how deep rootkits are integrated in common EDR product’s threat models and haven’t really faced any detections so far, this experiment was fun to learn about potential detection vectors. I believe that EDRs will be very careful when interferring with kernel components, as this poses a significant risk to the overall system’s stability. However, I plan to spend some time reversing common anti-rootkit drivers, to find out what kind of detections they implement.

I will continue to work on Banshee as well as unKover in the meantime as well, and when I am ready will continue with part 2 of this series. Stay tuned for some thread hiding fun with other undocumented data structures and more.

Happy Hacking!

Credits

- Shoutout to 0mWindyBug (@dorgerson) for some great discussions on anti-rootkits