Bypassing kernel function pointer integrity checks

secret.club/2019/11/06/kernel-code-alignment.html

November 6, 2019

vmcall

Nov 6, 2019

Ensuring (system) integrity is an important detail in software security products such as anti-cheats or anti-viruses. These are prevalent to make sure that the operating system's main functionality has not been tampered. One common integrity check is the verification of individual driver objects. These driver objects can be manipulated directly in memory (direct kernel object modification) for any number of reasons, but the specific exploitation that this article revolves around is the modification of the *major function IRP_MJ_DEVICE_CONTROL*, known as the I/O handler.

… DeviceIoControl hook?

Hooking DeviceIoControl is a common method to intercept information fetches from drivers such as the disk driver (*disk.sys*) for disk information or the network driver (*ndis.sys*) for adapter information. This type of information is quite trivial to fetch and is used by anti-cheats and other software products for identifying unique users for licensing or blacklisting. [Cheaters have been combatting this type of *hardware-ban* (Flagging certain hardware serial numbers and banning anyone with associated them) for years by intercepting *DeviceIoControl* calls to appear as if they are using completely different hardware.

A great example of this is the *hdd serial spoofer* project by namazso, that hooks the disk driver and spoofs any queries to harddrive serial numbers, preventing anti-cheats such as *BattlEye* or *Easy Anti Cheat* from blacklisting players solely through disk information.

This method is really simple, and consists of directly modifying the function pointer table *MajorFunction* in the given driver object. This table contains all of the essential functions for the driver, including the DeviceIoControl function (*IRP_MJ_DEVICE_CONTROL*) and is used by system operations to look up the respective handler of a certain event. To hook the I/O handler, we simply overwrite the function pointer to our given intermediate:

```
auto& io_handler = driver_object->MajorFunction[IRP_MJ_DEVICE_CONTROL];
ioctl_hook::original_handler = io_handler;  // SAVE ORIGINAL FUNCTION
io_handler = ioctl_hook::handler;           // SET POINTER TO HOOK
```

And as always, it is just as trivial to find out if the driver object has been modified.

Blind integrity checks

Anti-cheats such as *BattlEye* and *Easy Anti Cheat* use the same method to ensure the integrity of the major function table:

```
const auto& section = pe->sections[".text"];

for (const auto& major_function : driver->major_functions)
{
    if (major_function.address < section.address ||
        major_function.address >= section.address + section_size)
    {
        // ANOMALY FOUND
        // ..
    }
}
```

This is what I call a *blind* integrity check, as you do not actually know where the entries are supposed to point to, but you still need to make sure that they haven't been tampered with. As seen as above, this is done by comparing each entry to the memory region where they *should* reside - inside of the executable section of the driver. If the table entry points to a memory location that is outside of the driver object, it has most likely been hijacked by a malicious actor. This integrity check assumes that the target function has not been tampered with as long as it resides inside of the executable read-only section.

What this fails to detect, is if we **have** changed the table entry to something we can control, without leaving the section.

Hijacking control flow

How do we even hijack control flow when we're not able to change section data nor leave the section? Since we *can* actually control the execution flow, we can decide which byte sequence executes when the I/O handler is called, and as you probably already know, the CPU doesn't actually care about alignment. If we can find a byte sequence that yields a branch operation to a destination *outside* of the module, we can successfully hook the function without triggering the integrity check.

We can use the disk driver as a case study. First, let's dump the .text section of the loaded drisk driver by attaching windbg/kd and running the following command: `.writemem E:\disksys.dmp disk.sys+0x1000 disk.sys+0x6000`

This will output the .text section (of virtual size 0x5000) to the specified destination on disk. When that is done, we can create a simple branch analyzer using an arbitrary x86_64 disassembler and print out all **unconditional** branch operations including the destination, which we can later check for validity,

Here's a snippet of the *198* unconditional branch operations my static analyzer found:

```
.\branch_analyzer.exe E:\disksys.dmp
[+] Analyzing code branching
[=]   Loaded size.... 20481
[+] Branch
[=]   Target......... FFFFF80CCB379E7B
[=]   Instruction.... E9 45 8B 59 08
[+] Branch
[=]   Target......... FFFFF80CC2DE1095
[=]   Instruction.... E9 5F FF FF FF
[+] Branch
[=]   Target......... FFFFF80CC2DE10BE
[=]   Instruction.... E9 77 FF FF FF
[+] Branch
[=]   Target......... FFFFF80CC2DE10F5
[=]   Instruction.... EB 95
[+] Branch
[=]   Target......... FFFFF80CC2DE10F2
[=]   Instruction.... E9 77 FF FF FF
[+] Branch
[=]   Target......... FFFFF80CC2DE2B5D
[=]   Instruction.... E9 B7 19 00 00
[+] Branch
[=]   Target......... FFFFF80CC2DE2BC3
[=]   Instruction.... E9 E4 19 00 00
```

kd has a extension that allows you to look up the respective page table entry for any given address (*!pte*). Let's try this on the first target in the snippet:

```
0: kd> !pte FFFFF80CCB379E7B
                                           VA fffff80ccb379e7b
PXE at FFFF9BCDE6F37F80    PPE at FFFF9BCDE6FF0198    PDE at FFFF9BCDFE0332C8    PTE
at FFFF9BFC06659BC8
contains 00000000BFF08063  contains 0A000001DAF83863  contains 0000000000000000
pfn bff08     ---DA--KWEV  pfn 1daf83    ---DA--KWEV  not valid
```

Great! This confirms our suspicion that when a code section is sufficiently large, we can find unaligned x86_64 instructions that yield unconditional branching without having to modify the actual code. If we were to point the major function table entry to the given `E9 45 8B 59 08` code sequence, every time the I/O handler is called, the code flow goes to the memory address *FFFFF80CCB379E7B*, which is a memory region outside of the disk driver. This isn't the only instance of such an operation; the tool found **33** possible branch targets that reside in unallocated memory. Thankfully, we can automate this process by calling *MmIsAddressValid* and printing any invalid target. Now only one issue remains: how do we forcefully take control over a specific address in memory?

Manipulating page table entries

Thanks to virtual address translation, modern operating systems use some kind of page table system, which makes this quite trivial for us. All we have to do is find the respective page table entry and manually mark it as valid. The page table implementation on Windows

(Address translation are subject to change if your processor is from a different brand than Intel) is thoroughly documented in the Intel manual:

> With 4-level paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. 4-level paging translates 48-bit linear addresses to 52-bit physical addresses.1 Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time. 4-level paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table.

Here are some of the relevant structs, documented in the intel manual:

```
union PML4_BASE {
    std::uint64_t value;
#pragma warning(disable : 4201)         // UB
    struct {
        std::uint64_t ignored_1 : 3;
        std::uint64_t write_through : 1;
        std::uint64_t cache_disable : 1;
        std::uint64_t ignored_2 : 7;
        std::uint64_t pml4_p : 40;
        std::uint64_t reserved : 12;
    };
};
struct HARDWARE_PTE
{
    std::uint64_t Valid : 1;
    std::uint64_t Write : 1;
    std::uint64_t Owner : 1;
    std::uint64_t WriteThrough : 1;
    std::uint64_t CacheDisable : 1;
    std::uint64_t Accessed : 1;
    std::uint64_t Dirty : 1;
    std::uint64_t LargePage : 1;
    std::uint64_t Global : 1;
    std::uint64_t CopyOnWrite : 1;
    std::uint64_t Prototype : 1;
    std::uint64_t reserved0 : 1;
    std::uint64_t PageFrameNumber : 36;
    std::uint64_t reserved1 : 4;
    std::uint64_t SoftwareWsIndex : 11;
    std::uint64_t NoExecute : 1;
};
struct MMPTE
{
    union
    {
        std::uint64_t   Long;
        std::uint64_t   VolatileLong;
        HARDWARE_PTE    Hard;
        MMPTE_SOFTWARE  Soft;
    }u;
};
```

simplicity's sake, I will not cover address translation any further as it quite a long read. If you want to know more, you are welcome to read Intel Manual Vol. 3A chapter 4.5, which covers the specifics of 4-level paging; for the rest of the article, I will abstract that away for clarity.

When page table entry translation is done, we need to manually piece together a valid page table entry so the processor does not page fault when control flow is forced to that memory region. This can be done properly, by utilizing the page frame database, making sure that the windows operating system actually knows that this memory page is being used and thus not up for grabs *or* by simply manipulating the page table entry, causing a giant race condition as we are about to showcase ;-).

Because page table entries must be referred to by a page directory entry (which must be referred to by a page-directory-pointer-table entry, and so on), we need to initialize these structures, if need be. The full paging hierarchy is shown, as follows:

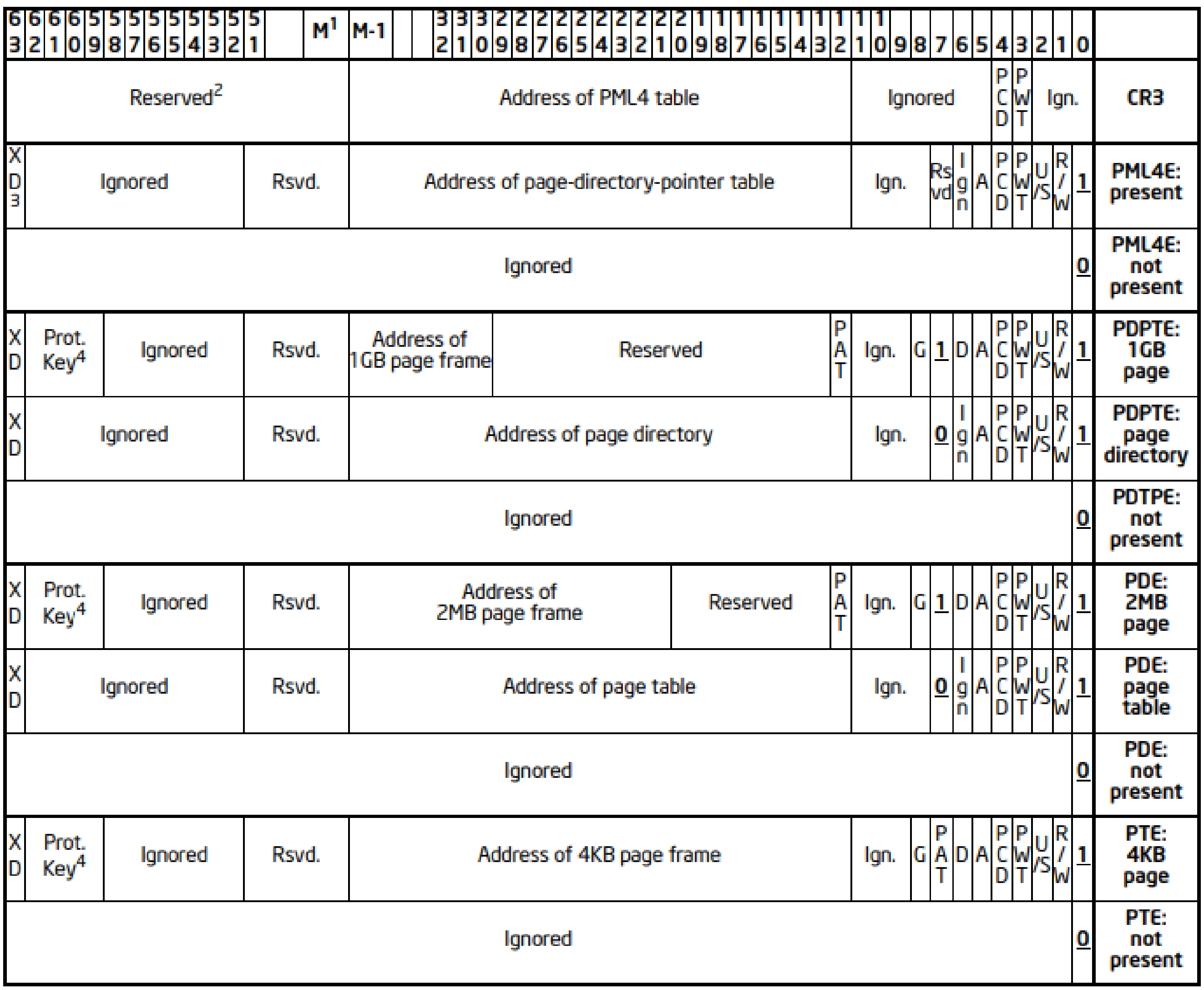

Fig.1 - Formats of CR3 and Paging-Structure Entries with 4-Level Paging, Copyright: Intel

Here is a basic reconstruction of a page table entry:

```
page_info.pte->u.Hard.Dirty = 1;
page_info.pte->u.Hard.Accessed = 1;
page_info.pte->u.Hard.Owner = 0;
page_info.pte->u.Hard.Write = write ? 1 : 0;
page_info.pte->u.Hard.NoExecute = execute ? 0 : 1;
page_info.pte->u.Hard.Valid = 1;
```

and a page directory entry:

```
page_info.pde->u.Hard.Dirty = 1;
page_info.pde->u.Hard.Accessed = 1;
page_info.pde->u.Hard.Owner = 0;
page_info.pde->u.Hard.Write = 1;
page_info.pde->u.Hard.NoExecute = 0;
page_info.pde->u.Hard.Valid = 1;
```

See? Extremely simple. That's really all you have to do; you're already in kernel mode, so you can fetch the base address of the PML4 through the control register CR3

Demonstration

You can find a proof-of-concept hardware spoofer using this type of hook on my GitHub. Make sure to follow me on Twitter and on GitHub to get notified every time I release projects :-)