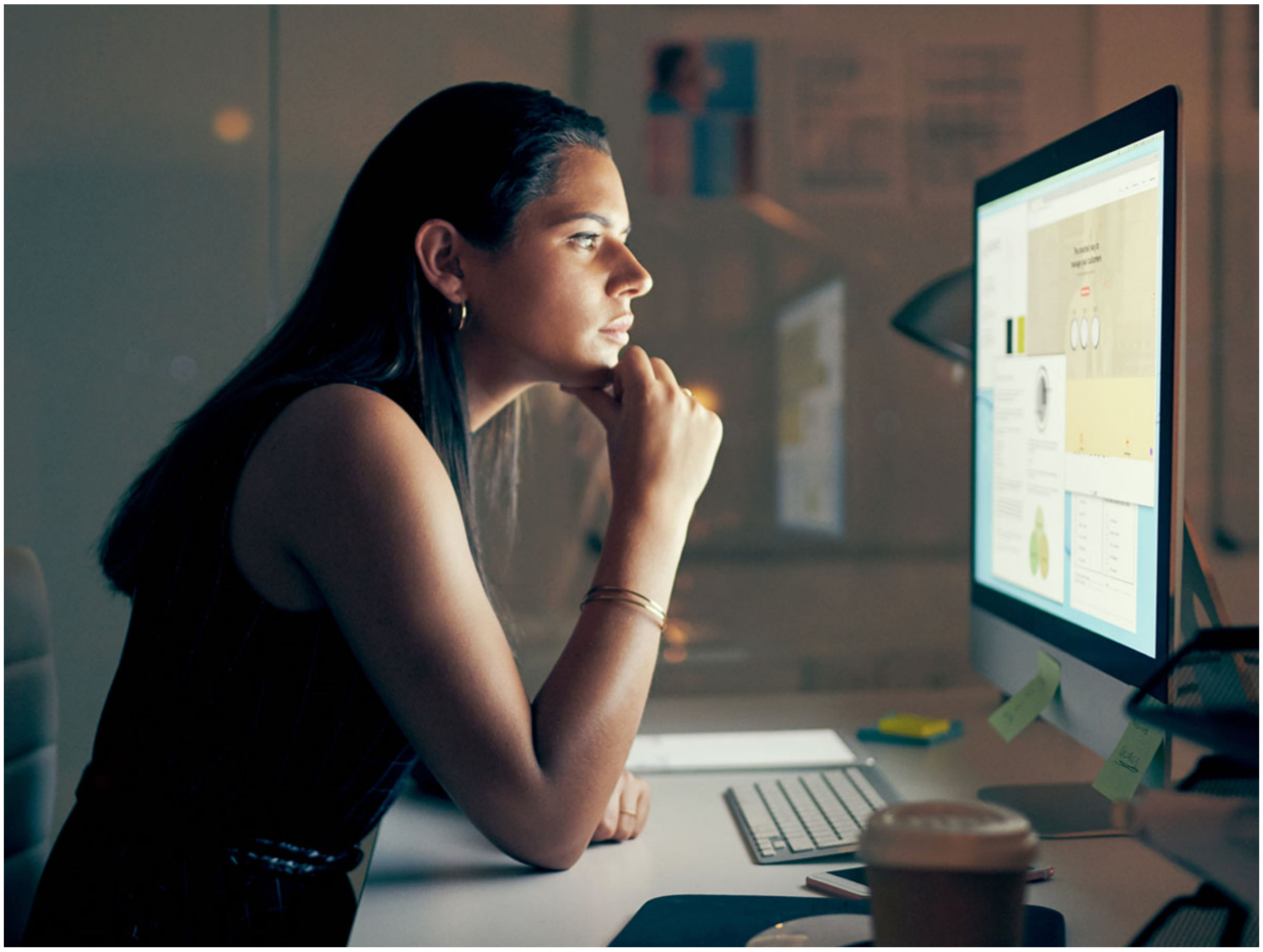
The Swan Song for Driver Signature Enforcement Tampering

fortinet.com/blog/threat-research/driver-signature-enforcement-tampering

August 12, 2022



Preface

Code Integrity is a threat protection feature first introduced by Microsoft over 15 years ago. On x64-based versions of Windows, kernel-mode drivers must be digitally signed and checked each time they are loaded into memory. This is also referred to as Driver Signature Enforcement (DSE). Detecting whether an unsigned driver or system file is being loaded into the kernel, or whether a system file has been modified, possibly by malicious software that runs with administrative permissions, improves the security of the operating system.

To overcome these restrictions, attackers use valid digital certificates, either issued to them or stolen, or they disable DSE during runtime. Obtaining a certificate is primarily a logistical obstacle but tampering, on the other hand, is purely a technical challenge.

Despite the efforts Microsoft dedicated to address this issue in recent years and the range of solutions provided by them, there’s been a clear increase in cases of attacks leveraging the well-known DSE tampering method. This motivated us to look deeper into the issue.

In this blog, we share the results of our research - details of two more methods for DSE tampering and how defenders might cope with this matter as long as this attack surface hasn't been eliminated.

DSE Implementation

An old blog post by j00ru, a security researcher from Google’s Project Zero, provides a high-level overview of Code Integrity implementation in Windows 7.

ntoskrnl.exe works with an additional kernel library, CI.dll (Code Integrity). At the operating system initialization phase, the kernel sets nt!g_CiEnabled and invokes CI!CiInitialize routine with a pointer to the nt!g_CiCallbacks structure to initialize CI.dll. It, in turn, sets CI!g_CiOptions and fills the addresses of CI!CiValidateImageHeader, CI!CiValidateImageData, and CI!CiQueryInformation callbacks before returning to the kernel.

To use the callbacks, wrapper functions exist for each of them in ntoskrnl.exe. They check that nt!g_CiEnabled is set to TRUE, that the appropriate callback is not NULL, and then call it.

When the kernel loads a driver, execution goes through nt!MmLoadSystemImage to nt!MmCreateSection and eventually to the nt!MiValidateImageHeader routine. From there, the nt!SeValidateImageHeader and nt!SeValidateImageData wrappers are invoked in order. Each callback is expected to return zero on success or a non-zero value otherwise.

Figure 1: Call stack on driver load

The book “Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats” (pages 76-78) covers implementation changes in Windows 8: the nt!g_CiEnabled variable was removed, so the DSE state is determined solely by CI!g_CiOptions, and more callbacks were added to the interface that CI.dll supplies. Another change not described in the book is that if the validate header succeeds, then validate data callback will not be called.

On Windows 8.1, the symbol name of the callbacks structure was changed to nt!SeCiCallbacks.

Kernel-Mode Signing Policy

Except for the cryptographic validity of the signature, Microsoft enforces that signing certificates can only be issued by cross-certificate supporting CAs. This thwarts attackers from simply installing their own CA certificate on every machine.

Beginning with Windows 10 Redstone (August 2016), the driver signing policy changed, requiring a second signature by Microsoft itself. This is done via a web portal where developers upload their signed binaries to send them to Microsoft. From an attacker's perspective, this means simply handing out your payload to the defenders, which is the opposite of what they usually want. There is at least one documented case where this new measure didn't stop a threat actor.

Kernel Patch Protection

KPP, or PatchGuard, first introduced in 2005, is a feature of x64 editions of Windows that prevents patching the kernel. "Patching the kernel" refers to modifying the code of ntoskrnl.exe and other critical system drivers and data structures (SSDT, IDT, GDT, etc.).

It works by periodically checking that these protected areas have not been modified. A BSOD will be triggered if a modification is detected, essentially stopping the system. PatchGuard is updated with each new Windows release, making it very difficult for attackers to develop a universal bypass technique for all versions.

The callbacks structure in ntoskrnl.exe and the CI!g_CiOptions variable are protected by PatchGuard starting with Windows 8 and 8.1, respectively.

DSE Tampering in the Wild

Though j00ru concluded that overwriting private symbols like nt!g_CiEnabled or CI!g_CiOptions might be relatively hard, this was the exact direction attackers chose to go. The infamous Turla APT is known to have developed such a technique, and security researchers reverse-engineered and published their code for it.

These private symbols are located by simple pattern matching, with hardly any changes between all Windows versions. Attackers overwrite the flags and quickly proceed to load their unsigned drivers. Once the loading procedure is finished, they restore the flags to their original state. Restrictions that PatchGuard imposes don't prevent these momentary changes, and as they are short-lived, the odds are they won't be detected either.

The common practice for attackers to get a write primitive is by leveraging 3rd-party drivers with vulnerabilities (or just poorly written code) that break security boundaries between the user and the kernel-mode code. Attackers usually bring such a driver with them rather than relying on one to be found running on the target machine. The considered tradeoff is first getting administrative privileges in user-mode and, in return, having a portable and sustainable capability even after patching.

Microsoft's Solutions

In light of this critical attack path, Microsoft tackled the problem in three ways:

1. Reducing the attack vector - an immediate solution using driver blocklist to make it harder to gain the write primitive.
2. Reducing the attack surface - an intermediate solution to prevent changes to CI!g_CiOptions variable using Kernel Data Protection.
3. Eliminating the attack surface - a long-term solution to prevent any code from running in the kernel without first being validated by HyperVisor-protected Code Integrity (HVCI).

Virtualization-based Security

VBS uses hardware virtualization features to create and isolate a secure region of memory from the normal operating system. Windows can use this "virtual secure mode" to host a number of security solutions, providing them with significantly increased protection from vulnerabilities in the operating system and preventing the use of malicious exploits that attempt to defeat protections. This architecture was first introduced in the initial Windows 10 release.

In VBS environments, privileges are set according to Virtual Trust Levels (VTLs). The normal NT kernel runs in a virtualized environment, called VTL0, while the secure kernel runs in an environment called VTL1.

Virtual memory management uses the Secondary Layer Address Translation (SLAT) page tables. The processor translates an initial virtual address, called Guest Virtual Address (GVA), to an intermediate physical address, called Guest Physical Address (GPA). This translation is still managed by the Guest OS page tables. The intermediate physical address needs to be translated by the processor to a Machine Physical Address, also known as a Host Physical Address (HPA) or System Physical Address (SPA), with the SLAT page tables the hypervisor maintains.

Figure 2: Regular PTE along SLAT PTE highlighted differences

On x86 architecture, SLAT PTEs (Page Table Entries) handle permissions differently from regular PTEs: read\write permissions are separate, execute permission needs to be explicitly set, and can be granted only for ring 0 (kernel-mode). The hypervisor uses SLAT page tables to enforce the isolation between VTLs and have them accessible for VTL1, so the secure kernel uses them to implement VBS features, while the hypervisor itself doesn’t implement any code/logic for the features themselves.

HyperVisor-protected Code Integrity

HVCI, originally called Device Guard, was released with the introduction of VBS. It is another layer of integrity enforcement.

Upon loading a new driver, the secure kernel is also triggered and uses its own instance of code integrity library, SKCI.dll (Secure Kernel Code Integrity). The digital signature is validated and checked to be authorized within the current policy in the Secure World (VTL1). Only then are executable and non-writable permissions applied to the SLAT page table for the corresponding GPAs. As a result, the NT kernel (VTL0) can't modify any previously loaded code or run any new code without using the secure kernel in the process.

Figure 3: Disassembly of SkciInialize and its own CiOptions variable in SKCI.dll from Windows 10

Kernel Data Protection

KDP is intended to protect drivers and software running in the Windows kernel (i.e., the OS code itself) against data-driven attacks. It was first introduced in Windows 10 20H1.

With KDP, software running in kernel-mode can protect read-only memory statically (a section of its own image) or dynamically (pool memory that can be initialized only once). KDP only establishes write protections in VTL1 for the GPAs backing a protected memory region using the SLAT page tables for the hypervisor to enforce. This way, no software running in the NT kernel (VTL0) can have the permissions needed to change the memory.

KDP does not enforce how the GVA range mapping of a protected region is translated. Based on the developer’s blog, KDP currently only periodically verifies that protected memory regions translate to the appropriate GPA.

Starting with Windows 11, CI.dll was opted to harden itself using static KDP (MmProtectDriverSection API), having all relevant CI policy variables placed in a separate section named “CiPolicy”.

Figure 4: Disassembly of CiInitializePolicy and CiPolicy section in CI.dll from Windows 11

Drivers Blocklist

This is enforced via Windows Defender Application Control (WDAC) or HVCI policy. According to several publications several third-party security product vendors have also adopted this practice. The latest blocklist can be found here.

Blocklisting denies easy access to a kernel write primitive for attackers. Although it’s quite effective, it is not a proactive measure – only previously discovered drivers can be handled. Hence, this mitigation is ineffective against zero-day vulnerabilities in drivers, and defenders must continuously chase after them, always remaining one step behind the attackers.

New Tradecraft

From the description above, a keen reader could see that without HVCI enabled, it might be possible to tamper with DSE without any code patching.

Method I - “Page Swapping”

Just because writing to CI!CiOptions is no longer possible doesn't mean its value can’t change. The variable is still being accessed by a virtual address and the translation process to a physical address takes place each time. So, we’ll change the translation result instead.

By swapping the physical pages from a KDP-protected page to one we own, we regain complete control over the memory. Swapping a GPA merely means changing the PFN (Page Frame Number) in the PTE (Page Table Entry), which essentially is just another pointer.

We can calculate the virtual address of the PTE for any given virtual address and avoid traversing all the page tables each time. The page tables reside in a region of virtual memory that the Windows Kernel uses to manage the paging structures, called “PTE Space”. PTE Base is randomized by KASLR (Kernel Address Space Layout Randomization), starting with Windows 10 Redstone. In previous research, we showed a reliable method to find it.

Performing this method requires kernel read and memory allocation primitives in addition to write. The following is a step-by-step implementation in C pseudocode:

Figure 5: C pseudocode for Page Swapping.

It’s possible to use a page from user-space and so a kernel memory allocation primitive becomes redundant. In the case of CI.dll, it’s possible to use the default values of the variables rather than copying the page. As a result, the number of reads from kernel space is significantly minimized as only the handful of necessary reads of PTEs are left.

Method II - “Callback Swapping”

For a moment, it looked like KDP raised the bar against DSE tampering since Page Swapping also requires a kernel read primitive. We looked again at how CI.dll and ntoskrnl.exe integrate, and then we thought, “why even get to a point CI.dll is used? Let’s just have our own callback instead of CI!CiValidateImageHeader”.

Figure 6: Illustration of Callback Swapping concept

Our PoC demonstrates that finding all the necessary addresses is feasible without reading any kernel space data. The general details follow:

First, locate the callback structure in ntoskrnl.exe. The structure is passed as a parameter to CI!CiInitialize, so we can obtain its address from that call. The kernel calls the function only once, so we look for a CALL or JMP instruction that uses its import table entry. Once you find the call site, walk back to an assignment of register for a parameter pointing to uninitialized memory in the “.data” section.

Figure 7: Disassembly of SepInitializeCodeIntegrity and SeCiCallbacks in ntoskrnl.exe from Windows 11

Next, look for a replacement callback function to use. We need a function that does not take parameters and returns zero. Fortunately, ntoskrnl.exe has a few exported functions that fit this bill, like FsRtlSyncVolumes or ZwFlushInstructionCache, so a simple call to GetProcAddress takes care of that.

Lastly, find the original callback function(s) to be restored. The callbacks are set in the structure by CI!CipInitialize, so it will have references to all of them. All the callbacks are set in a single basic block of code across all Windows builds. Search for this pattern of instructions, seen in figure 8, and extract the offsets from the lea instructions. To validate that the offsets actually lead to functions, traverse the PE’s Exception Directory to look for a RUNTIME_FUNCTION entry with the same start address.

Figure 8: Disassembly of CipInitialize in CI.dll from Windows 11

KDP protection is bypassed “by design” as ntoskrnl.exe hasn’t opted in with the callbacks structure. Another advantage to changing the callbacks is that the tampering is not visible through the documented query system information API.

While resolving the addresses might take a few more lines of code, it is done in userspace, so only a kernel write primitive is required - the same as in the current well-known DSE tampering method. Although PoC has writes of 8 bytes (size of a 64-bit pointer) to kernel-space, this number can be reduced by looking for callback targets in CI.dll instead. While preparing this blog, Adam Chester from TrustedSec released his work on the subject, where he chose an alternative approach to finding the original callback by scanning for a binary signature he created.

Conclusion

Mitigation

HVCI covers all tampering methods as it performs its own validation when a driver is being loaded. Despite HVCI being available for many years now, it has been turned on by default on new Windows installations only recently. With this understanding, we tried to come up with an alternative.

We tried to find a way to confirm the state of DSE during driver loading. Moreover, one that will support blocking of the procedure. How to get visibility of the state of DSE should be evident at this point - defenders can just leverage the same strategies attackers use to find the internal variables. After all, it has proven to be stable.

Considering that a tampered state can only last for short moments, assuming the system state is valid when we start running is sound. At this point, a copy of the internal variables will be kept.

There are a few options to intercept driver loading:

1. Place a hook in userspace on NtLoadDriver API.
2. Use registry callbacks to monitor operations on a driver's registry key path.
3. Use file system minifilter callback for creation of section for a driver's file (IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION).

To know if a callback is triggered as part of driver loading, it needs to check that the current process is SYSTEM and the call stack originated from ntoskrnl.exe and not any other driver.

Once driver loading is intercepted, DSE tampering is detected by checking any variables for changes. At this point, prevention could be done simply by blocking the I\O request with an error status code or restoring the variables to their saved state.

Final Thoughts

In this blog, we covered how Microsoft attempts to protect DSE on runtime and presented two new methods to tamper with it and successfully load unsigned drivers.

While HVCI provides the utmost robust solution, we shared an additional approach to detect and protect against DSE tampering on runtime. Using the same premises as attackers can pan out to be more successful than the rest of the existing protections.

Executing code in kernel mode will remain attractive for attackers as it's usually a steppingstone to compromising higher privileged elements on the machine, such as the hypervisor, UEFI, and SMM, or beating endpoint security products. It's possible that we might see a shift in TTPs, where attackers move to use more kernel 1-day exploits for unpatched vulnerabilities due to driver blocklisting.

As hardware-assisted security capabilities become more common, along with the efforts Microsoft dedicates to utilizing them, leveraging DSE tampering by threat actors will likely fade out in the foreseeable future. Nevertheless, misconfigured and legacy systems will remain to a degree, so this avenue of attack won't be made entirely obsolete. Therefore, we urge defenders to adopt the proposed solution in this blog.

Fortinet Solutions

FortiEDR detects and blocks these methods out-of-the-box without any prior knowledge or special configuration using its post-execution prevention engine to identify malicious activities, as seen in figure 9.

Figure 9: FortiEDR blocks driver loading when DSE was tampered with.

Appendix A: Additional References

Learn more about Fortinet's FortiGuard Labs threat research and intelligence organization and the FortiGuard Security Subscriptions and Services portfolio.