Memory Hiding by VT-x
VXCON 2017

Tencent
Kelvin

About Me

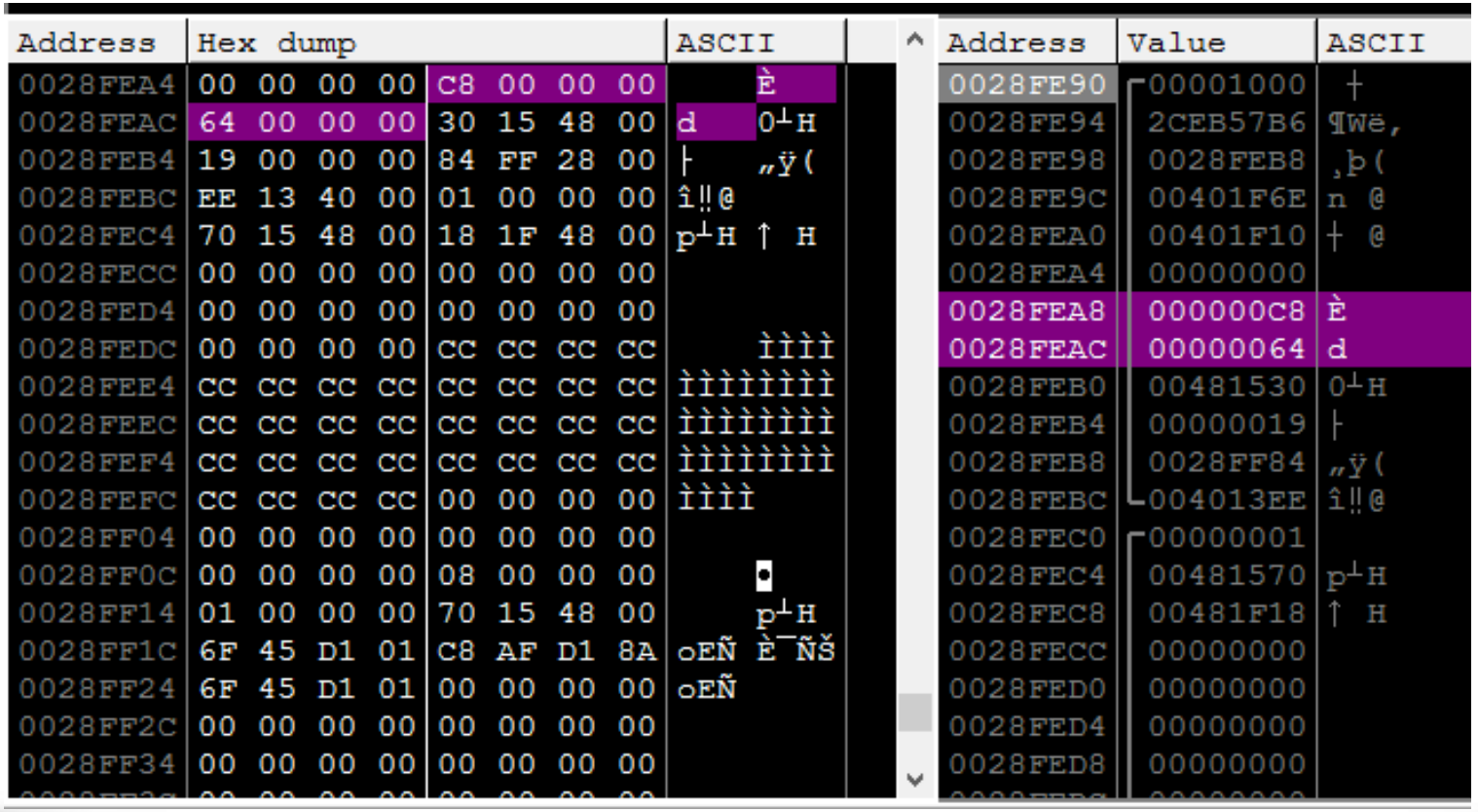
Name: Kelvin Chan
Work: Tencent Game Security and Windows Kernel Security Researcher
Facebook: Kelvin Chan
Email : KelvinChan@tencent.com

Overview

- Review some basic memory paging mechanism
- Virtualization Technology to be used to attack.
- Hidden Windows x64 user mode memory by using VT-x

What is memory hiding?

- No ones (threads) can tell the truth anymore.
- Practically, Any memory address can have different value :)

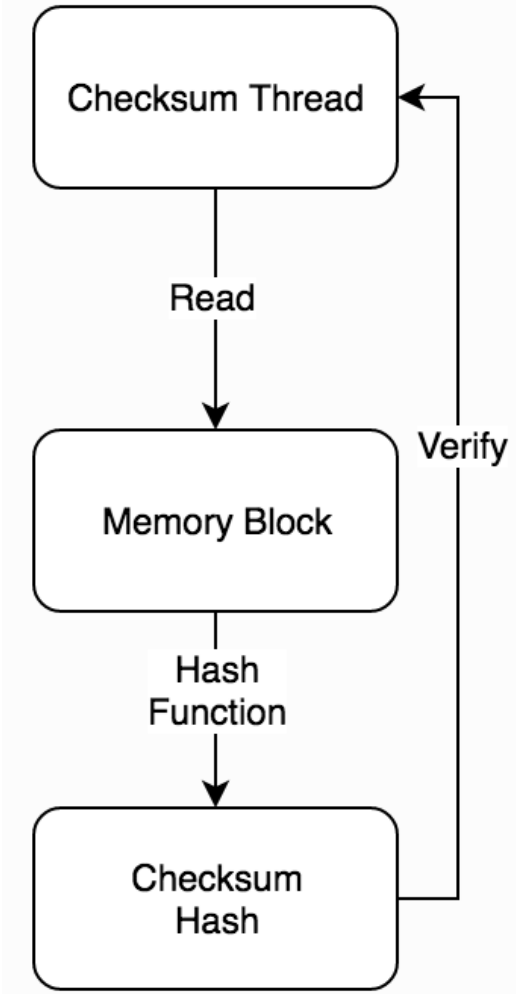


Address	Hex dump	ASCII
0028FEA4	00 00 00 00 C8 00 00 00	È
0028FEAC	64 00 00 00 30 15 48 00	d 0§H
0028FEB4	19 00 00 00 84 FF 28 00	├ „ÿ(
0028FEBC	EE 13 40 00 01 00 00 00	î‼@
0028FEC4	70 15 48 00 18 1F 48 00	p§H ↑ H
0028FECC	00 00 00 00 00 00 00 00	
0028FED4	00 00 00 00 00 00 00 00	
0028FEDC	00 00 00 00 CC CC CC CC	ÌÌÌÌ
0028FEE4	CC CC CC CC CC CC CC CC	ÌÌÌÌÌÌÌÌ
0028FEEC	CC CC CC CC CC CC CC CC	ÌÌÌÌÌÌÌÌ
0028FEF4	CC CC CC CC CC CC CC CC	ÌÌÌÌÌÌÌÌ
0028FEFC	CC CC CC CC 00 00 00 00	ÌÌÌÌ
0028FF04	00 00 00 00 00 00 00 00	
0028FF0C	00 00 00 00 08 00 00 00	◘
0028FF14	01 00 00 00 70 15 48 00	p§H
0028FF1C	6F 45 D1 01 C8 AF D1 8A	oEÑ È¯ÑŠ
0028FF24	6F 45 D1 01 00 00 00 00	oEÑ
0028FF2C	00 00 00 00 00 00 00 00	
0028FF34	00 00 00 00 00 00 00 00	

Address	Value	ASCII
0028FE90	00001000	┼
0028FE94	2CEB57B6	¶Wë,
0028FE98	0028FEB8	¸þ(
0028FE9C	00401F6E	n @
0028FEA0	00401F10	┼ @
0028FEA4	00000000	
0028FEA8	000000C8	È
0028FEAC	00000064	d
0028FEB0	00481530	0§H
0028FEB4	00000019	├
0028FEB8	0028FF84	„ÿ(
0028FEBC	004013EE	î‼@
0028FEC0	00000001	
0028FEC4	00481570	p§H
0028FEC8	00481F18	↑ H
0028FECC	00000000	
0028FED0	00000000	
0028FED4	00000000	
0028FED8	00000000	

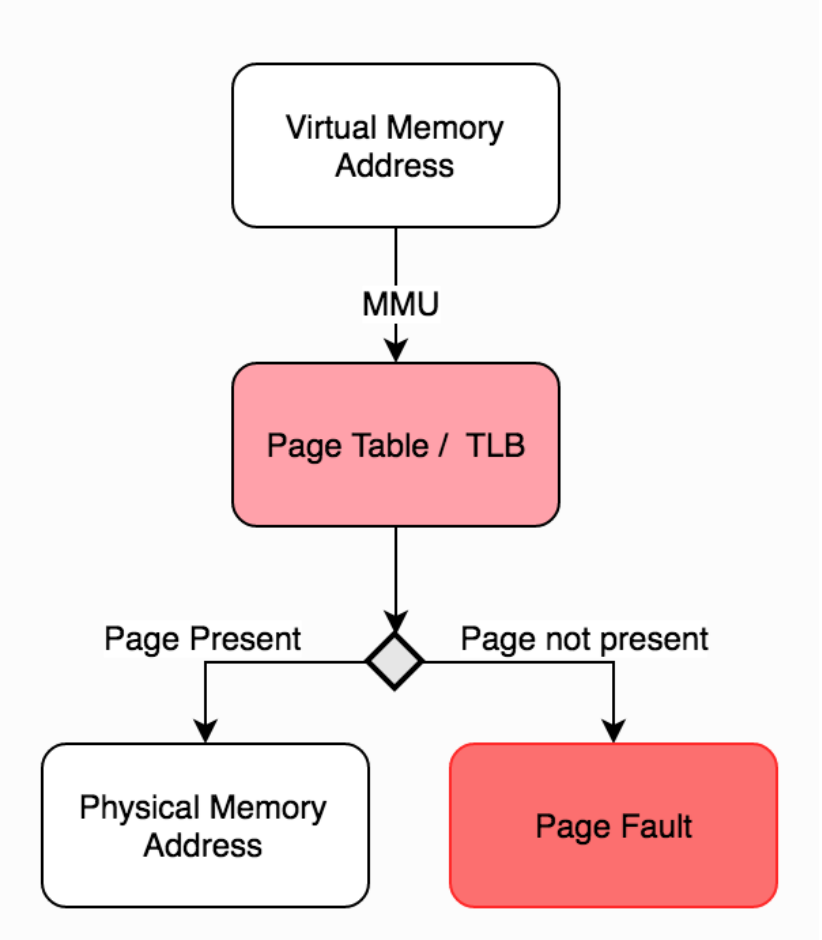
Motivation and Demands

- Software attack always involved **Hooks**
- Hooks always involved **Memory Modification**
- Checksum algorithms as an usual trick to defense these type of attack, such as, md5, crc32, sha-1, sha-2 …etc.
- So the question is, How can attacker bypasses those algorithms without research on the hash function ? such as, Windows Kernel PatchGuard



Traditional Memory Paging

- 3 or 4 level Page Table
- Accessed Page Table Entry which stored in Translation Look-aside buffer(TLB)
- Every Memory access will probably cause Page Fault (#PF)

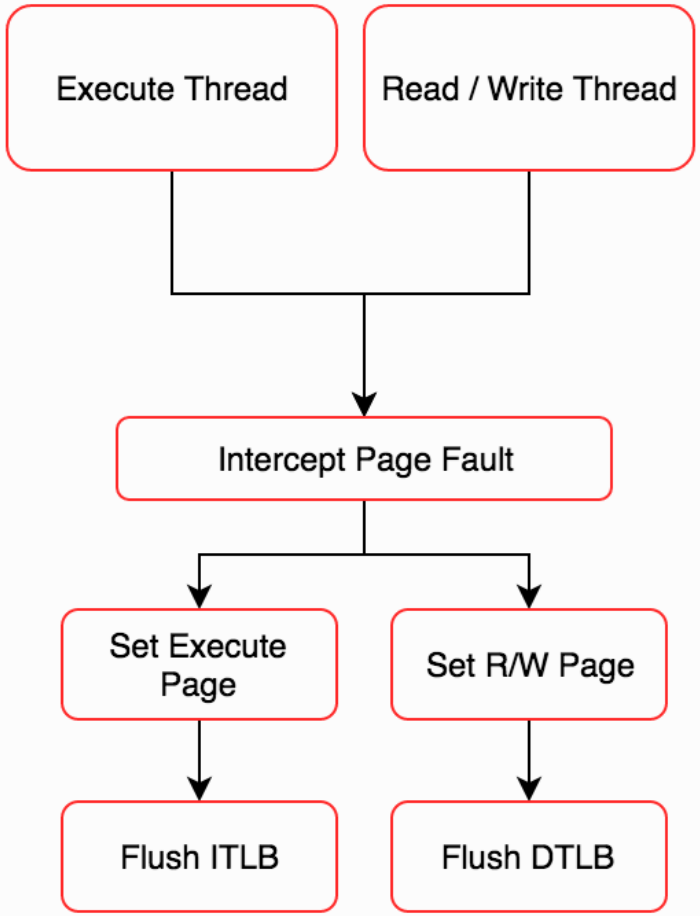


Previous Research

- No VT-x
- Invalidate, and Split the Page
- Flush the TLB (ITLB and DTLB)
- The limitation :
  - Flush a ITLB will also flush a DTLB by MMU,
  - When we execute the page, and it is invalid both type of TLB will be flushed.
  - Finally, the memory read / write won't occur a Page Fault

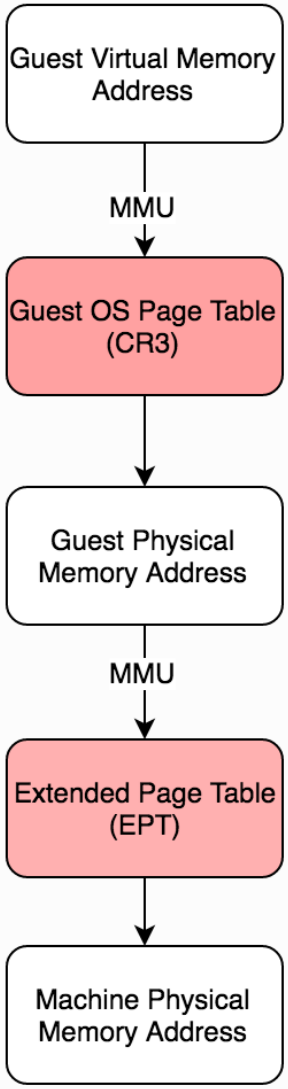
  The Memory hiding will be useless anymore.
  (we are not able to return a fake page)

  It is impossible for hiding a user mode memory address.
  But kernel.



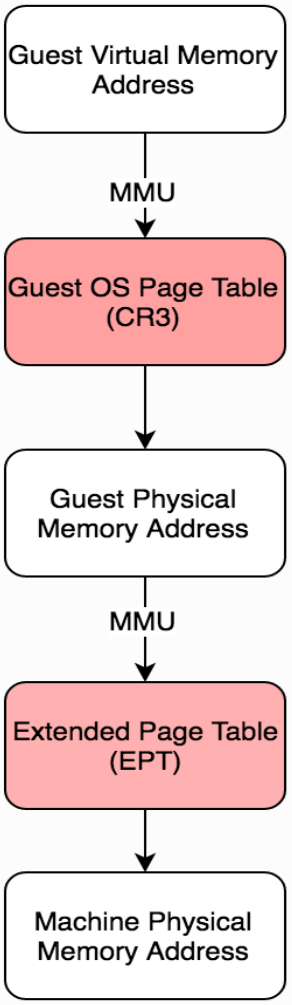
Virtualization of Virtual Memory

- Intel provides a Extended Page Table (EPT) in VT-x for memory virtualization, including the V-TLB.
- The whole process of memory mapping has been changed by hardware, when VT-x is on



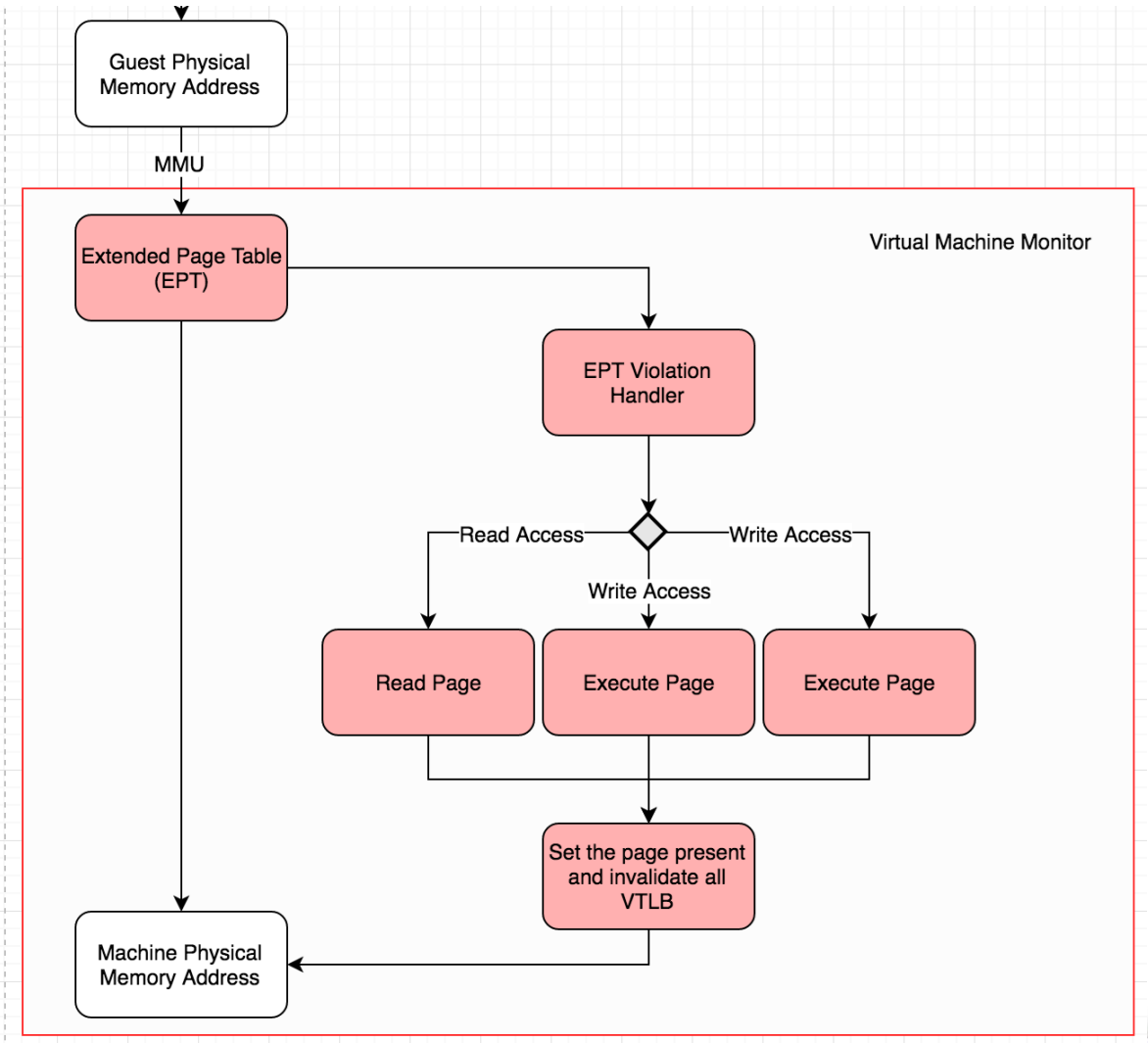
New Approach for hiding

- We can directly set the Host Physical Page as **not present**, and invalidate **the EPT-TLB** , force the system need to do the translation by walking the page table again, but directly translate GPA to MPA / HPA by EPT-TLB
- EPT Violation , CPU will be raising a new Exception when the translation process in GPA to MPA / HPA if the page is marked as not present.
- We can provide a EPT Violation Handler for CPU VT-x , and **redirect the GPA to our private / pre-allocated MPA / HPA**. And **set it is present** and return to Guest, Guest translate again.
- We call those page as execute/write page or read page.
- Next problem, it is present now, how keep hiding….?



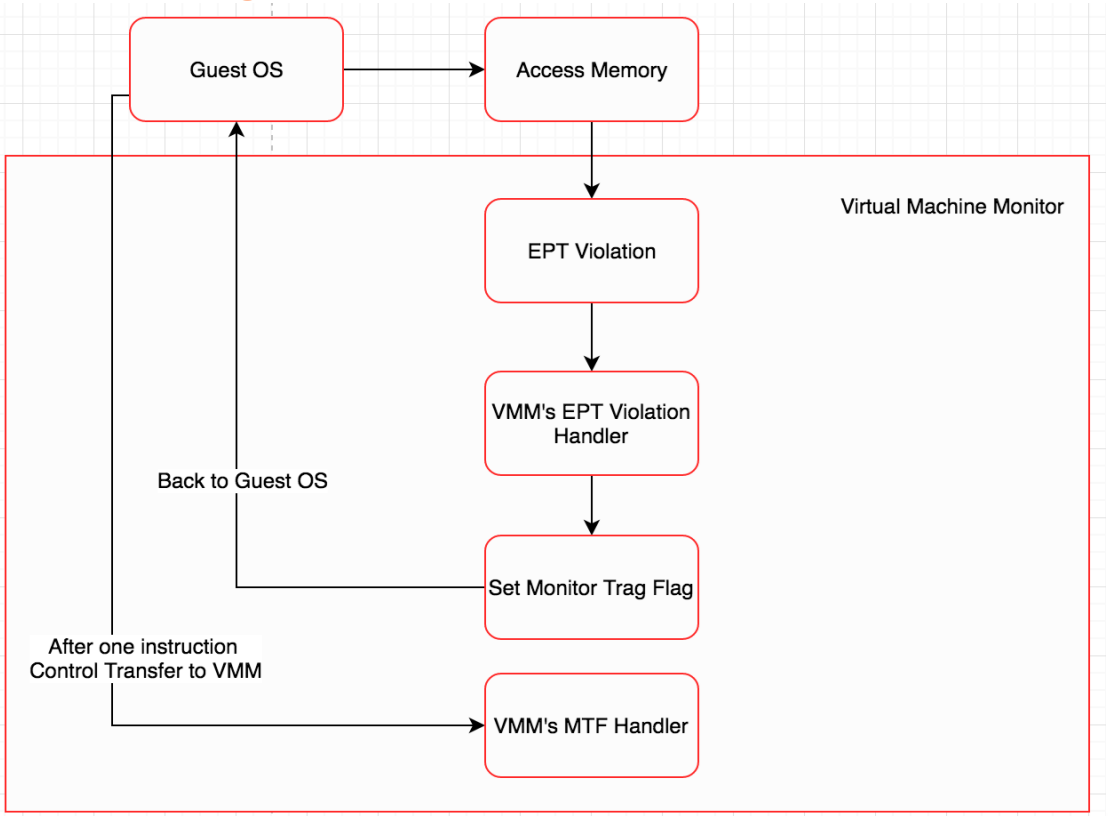
EPT Violation Handler

- Determine the Access Type
- Set the Corresponding Page to the EPT-PTE
- Invalidate all VTLB



Reboot the Hiding

- The final step of EPT handler, we will set a Monitor Trap Flag (MTF) , after the memory access by Guest OS, the control transfer to the VMM once again. In that **MTF VM-Exit handler** we have a chance **to reset EPT-PTE once again as same as the initialization of hiding**



User Mode Memory

- That's all in case of kernel memory (**non-pagable portion**) hiding since the relationship, a hidden PA are never change to other usage (swapped out) , but for **user mode memory** there are some troublesome.

1. All of User mode memory are **pagable memory**, that's mean **it is not always reside in physical memory**, it will be swapped out (**the Guest PA will not be describing the same Guest VA anymore. As a result, we will be probably hiding other's process page…**)

2. User mode memory always uses **Copy-On-Write** (COW) technique for DLL memory saving. That's mean when we write an address in a page, the relationship between VA and PA is changed.

3. The process closing.

User Mode Memory

- We can easily used MmProbeAndLockPages in Ring 0, or alternatively, VirtualLock in Ring 3 to mark that page as resident, and locks them in physical memory, to ensure the owner of that Physical Address will not be changed (always belongs to our target process.)

- The problem of COW can be easily deal with before we start the memory hidden, we write one byte data for causing the system COW once.

- Setting up the Process Notification in Ring 0 and monitor the process creation or exiting event.

NoTruth Interface

```
if (!drv.IoControl("\\\\.\\NoTruth",IOCTL_HIDE_ADD, &transferData2, sizeof(TRANSFERIOCTL), &OutBuffer, sizeof(ULONG), &RetBytes))
{
        drv.Stop(SERVICE_NAME);
        drv.Remove(SERVICE_NAME);
        CloseHandle(handle);
        return;
        AfxMessageBox(L"Cannot IOCTL device \r\n");
}

if (!drv.IoControl("\\\\.\\NoTruth",IOCTL_HIDE_START, NULL, 0, NULL, 0, &RetBytes))
{
        drv.Stop(SERVICE_NAME);
        drv.Remove(SERVICE_NAME);
        CloseHandle(handle);
        return;
        AfxMessageBox(L"Cannot IOCTL device \r\n");
}
```

Test Demo Code

```
//-------------------------------------------//
int main()
{
    g_NtCreateThread = (pMyNtCreateThread)GetProcAddress(LoadLibraryA("ntdll.dll"), "NtCreateThread");

    pUnitTest UnitTest = (pUnitTest)GetProcAddress(LoadLibrary(L"VTxRing3.dll"), "UnitTest");
    pSetupInlineHook_X64 SetupInlineHook_X64 = (pSetupInlineHook_X64)GetProcAddress(LoadLibrary(L"VTxRing3.dll"), "SetupInlineHook_X64");

    printf("g_NtCreateThread: %I64x UnitTest: %I64x SetupInlineHook_X64: %I64x ", (UINT64)g_NtCreateThread, (UINT64)UnitTest, (UINT64)SetupInlineHook_X64);

    if (g_NtCreateThread&&UnitTest&&SetupInlineHook_X64)
    {
        UnitTest(g_NtCreateThread, MyNtCreateThread);
        SetupInlineHook_X64(&g_HookObj, g_NtCreateThread, MyNtCreateThread);

        for (int i = 0; i < 10; i++)
        {
            CreateThread(0, 0, (LPTHREAD_START_ROUTINE)CheckSumThread, 0, 0, 0);
            CreateThread(0, 0, (LPTHREAD_START_ROUTINE)ExecuteThread, 0, 0, 0);
        }
    }

    getchar();

    return 0;
}
```

Load a Driver

Hook it

Start to validate memory checksum,
(read memory)

Test Demo Code

```
NTSTATUS
MyNtCreateThread(
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK DesiredAccess,
    IN PVOID ObjectAttributes OPTIONAL,
    IN HANDLE ProcessHandle,
    OUT PVOID ClientId,
    IN PCONTEXT ThreadContext,
    IN PVOID InitialTeb,
    IN BOOLEAN CreateSuspended)
{

    OutputDebugString(L"Test my thread hook \r\n");
    const auto Original = FindOrignal(MyNtCreateThread, g_HookObj);
    const auto status = Original( ThreadHandle, DesiredAccess,  ObjectAttributes,   ProcessHandle,  ClientId,   ThreadContext,  InitialTeb, CreateSuspended);
    return status;
}

//-------------------------------------------//
DWORD WINAPI ExecuteThread(PVOID Param)
{
    while (1)
    {
        __try {
            g_NtCreateThread(0, 0, 0, 0, 0, 0, 0, 0);
        }
        __except (DumpExecptionCode(GetExceptionCode())){}
        Sleep(1000);
    }
    return 0;
}
//-------------------------------------------//
DWORD WINAPI CheckSumThread(PVOID Param)
{
    while (1)
    {
        ULONG value = 0;
        value = *(PULONG)g_NtCreateThread;
        printf("Checksum Value: %lx TickCount: %I64x \r\n", value, (UINT64)GetTickCount());
        Sleep(1000);
    }
    return 0;
}
```

NtCreateThread Has been hooked

Supposed it is FF 25 00 00 in case without memory hidding

Demo

Result Explanation

- After inline hook, if we read a memory, it supposed to be
  - 0xFF 0x25 0x00 0x00 0x00 0x00 with 8 byte absolute virtual address.
  - We read a ULONG, so that will be 00 00 25 FF (little-endian)
  - It is JMP instruction in assembly format
- After memory hiding engine started, we read a memory, it supposed as same as original value.
- As a result, all hash function is going to be faked by us. It is always TRUE.
- Another source
- https://github.com/Kelvinhack/kHypervisor

End