基于 GDI 对象的 Windows 内核漏洞利用

Saif El-Sherei / BDomne@看雪论坛（译）

原文链接：https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/5A1F/DEFCON-25-5A1F-Demystifying-Kernel-Exploitation-By-Abusing-GDI-Objects-WP.pdf

注：仅针对其中讲解的前置知识点部分做了翻译。

0x00 引子

本文我们将讨论造成 Windows 内核池（Kernel Pool）破坏的整数溢出问题，并基于 `Bitmap` 和 `Palette` 这两个 GDI 对象来探究内核漏洞的利用过程。当然，文中提出的观点仅代表作者如何理解以及解决这些问题。

0x01 WinDbg 中与内核池相关的命令

- `!poolused`：此命令可用于查看具有特定标识或类型的内核池使用情况。

```
1: kd> !poolused 0x8 Gh?4

*** CacheSize too low - increasing to 102 MB

Max cache size is        : 107343872 bytes (0x1997c KB)
Total memory in cache    : 17864 bytes (0x12 KB)
Number of regions cached: 57
667 full reads broken into 680 partial reads
    counts: 618 cached/62 uncached, 90.88% cached
    bytes : 27300 cached/14200 uncached, 65.78% cached
** Transition PTEs are implicitly decoded
** Prototype PTEs are implicitly decoded

 Sorting by Session Tag

                NonPaged                    Paged
 Tag     Allocs         Used     Allocs          Used

 Gh04         0            0       5001      15040544  GDITAG_HMGR_RGN_TYPE , Bina

TOTAL         0            0       5001      15040544
```

- `!poolfind`：此命令用于查找具有特定标识的内核池分配对象。

```
1: kd> !poolfind Gh?4 -session

Scanning large pool allocation table for tag 0x343f6847 (Gh?4) (ffffe001d5d4b000


Searching session paged pool (fffff90140000000 : fffff9213fffffff) for tag 0x343

fffff90140972010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140974010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140978010 : tag Gh04, size     0xbb0, Paged session pool
fffff9014097c010 : tag Gh04, size     0xbb0, Paged session pool
fffff9014097e010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140980010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140982010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140984010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140986010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140988010 : tag Gh04, size     0xbb0, Paged session pool
fffff9014098a010 : tag Gh04, size     0xbb0, Paged session pool
fffff9014098c010 : tag Gh04, size     0xbb0, Paged session pool
fffff9014098e010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140990010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140992010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140994010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140996010 : tag Gh04, size     0xbb0, Paged session pool
fffff90140998010 : tag Gh04, size     0xbb0, Paged session pool
```

- `!pool`：此命令用于查看特殊地址所处的内核池信息。

```
fffff901425fe010 : tag Gh04, size     0xbb0, Paged session pool
fffff901425ff010 : tag Gh04, size     0xbb0, Paged session pool

...terminating - searched pool to fffff90143693000
1: kd> !pool fffff901425ff010
Pool page fffff901425ff010 region is Paged session pool
*fffff901425ff000 size:  bc0 previous size:    0  (Allocated) *Gh04
            Pooltag Gh04 : GDITAG_HMGR_RGN_TYPE, Binary : win32k.sys
 fffff901425ffbc0 size:  440 previous size:  bc0  (Free)       ....
```

0x02 内核池

内核池的类型

内核池可类比于用户态下的堆内存，不同之处在于它是在内核态中使用的。它有许多类型[1]，其中最常用的几种类型如下：

- 桌面堆（`Desktop Heap`）：主要用于窗口、类、菜单等桌面对象，分配函数为 RtlAllocateHeap() 和 DesktopAlloc()，释放函数为 RtlFreeHeap()。
- 非分页池（`Non-Paged Session Pool`）：在该类池上分配的对象，其对应的虚拟地址和物理地址是存在映射关系的，其中一些为系统对象，如信号量、事件对象等。
- 分页池（`Paged Session Pool`）：此类型是本文主要关注的，对于该类池上分配的对象，其对应的虚拟地址和物理地址并不存在一一映射关系，只需保证对象在当前执行会话中是有效的，而在其余的内核操作时并不要求这些对象必须在内存中，如 GDI 对象和一些用户对象等。

对分页池和非分页池来说，分配函数均为 ExAllocatePoolWithTag()，其中以第一个参数作为类型区分，若为 0x21，则分配到分页池，若为 0x29，则分配到非分页池。二者的释放函数为 ExFreePoolWithTag() 和 ExFreePool()。

内核池的分配

通过查看 Win32AllocPool() 函数我们可以知道内核是如何分配分页池对象的（类型参数为 0x21）。

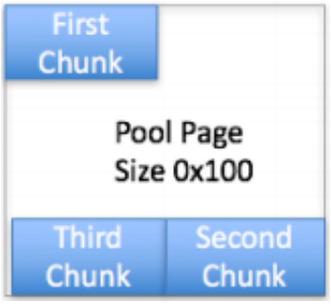
```
; Attributes: bp-based frame

; int __stdcall Win32AllocPool(SIZE_T NumberOfBytes, ULONG Tag)
_Win32AllocPool@8 proc near

NumberOfBytes= dword ptr  8
Tag= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    [ebp+Tag]       ; Tag
push    [ebp+NumberOfBytes] ; NumberOfBytes
push    21h             ; PoolType
call    ds:__imp__ExAllocatePoolWithTag@12 ; ExAllocatePoolWithTag(x,x,x)
pop     ebp
retn    8
_Win32AllocPool@8 endp
```

关于内核池需要了解的另一点是它的内存空间以 0x1000 字节大小划分成页，对每个池页面来说，初次分配的 chunk 块将位于页面的起始处，而接下去的 chunk 块在大部分情况下将从页底开始分配。



此外，在 64 位系统中，内核 `Pool Header` 结构的大小为 0x10 字节，相应的 32 位系统中的大小则为 0x8 字节。

```
x64 Pool Header: size 0x10
kd> dt nt!_POOL_HEADER
   +0x000 PreviousSize     : Pos 0, 8 Bits
   +0x000 PoolIndex        : Pos 8, 8 Bits
   +0x000 BlockSize        : Pos 16, 8 Bits
   +0x000 PoolType         : Pos 24, 8 Bits
   +0x004 PoolTag          : Uint4b
   +0x008 ProcessBilled    : Ptr64, _EPROCESS
```

```
x86 Pool Header: size 0x8
kd> dt nt!_POOL_HEADER
   +0x000 PreviousSize     : Pos 0, 9 Bits
   +0x000 PoolIndex        : Pos 9, 7 Bits
   +0x002 BlockSize        : Pos 0, 9 Bits
   +0x002 PoolType         : Pos 9, 7 Bits
   +0x004 PoolTag          : Uint4b
```

Pool Feng shui（池喷射）

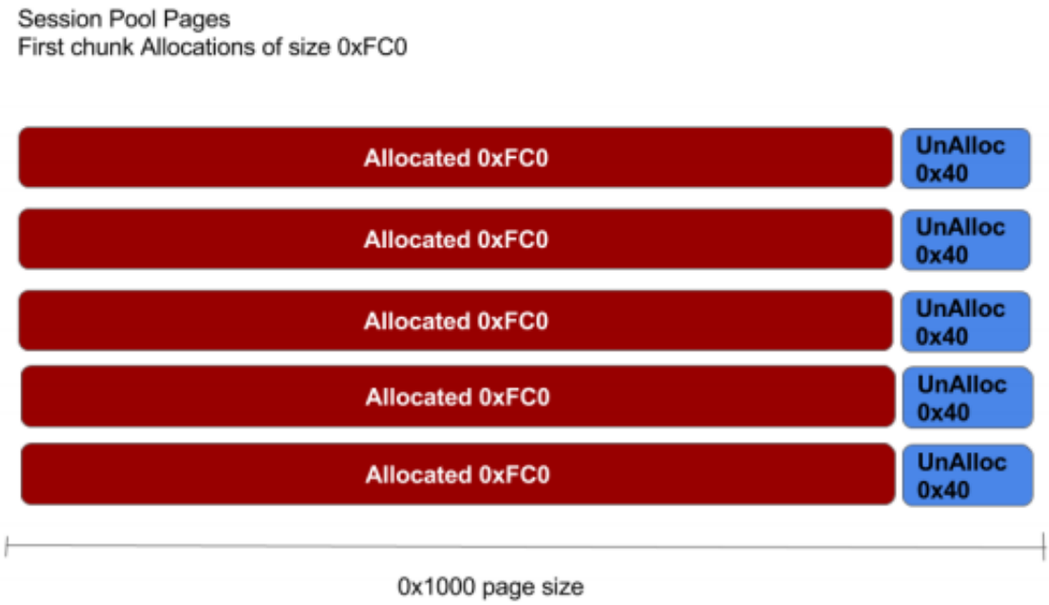
`Pool Feng shui` 背后依据的原理是通过适当操作可将池内存置于一种可预测的状态。即通过一系列的分配和释放操作来构造与漏洞对象大小相同的内存空洞（holes），以便将存在漏洞的对象分配到我们可控对象的相邻处，从而完成利用

对象的内存布局。

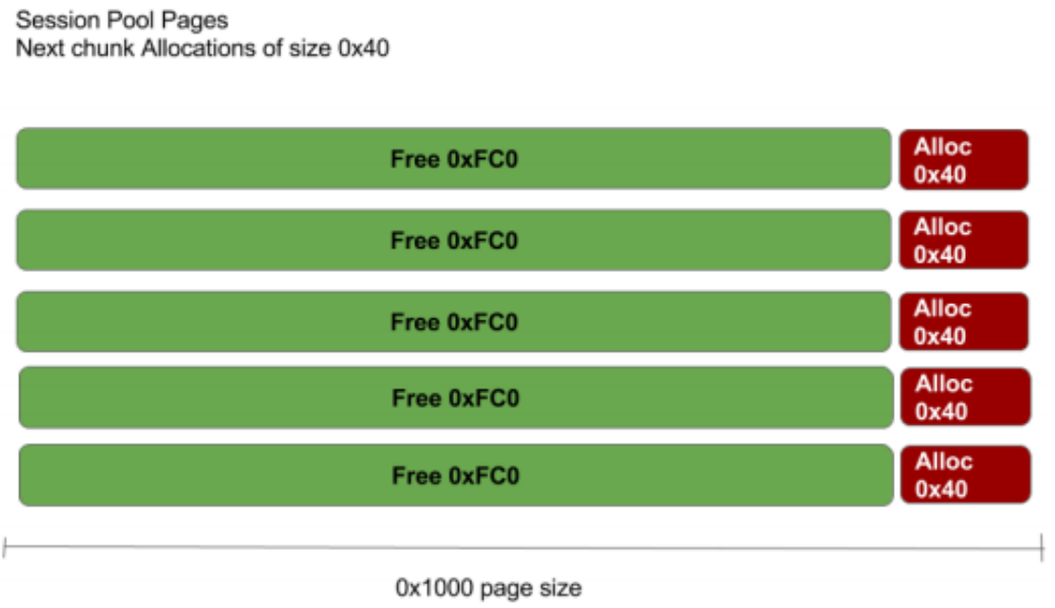
如果该漏洞对象在执行过程中没有被释放，那么需要构造的空洞可位于池页面的任何地方，但如果该对象最终被释放掉了，那么就需要确保漏洞对象被置于池页面的最后，即下一 chunk 头将不再有效，这样对象被释放后不会由于 BAD POOL HEADER 而触发蓝屏。

强制对象分配到池页面的尾部

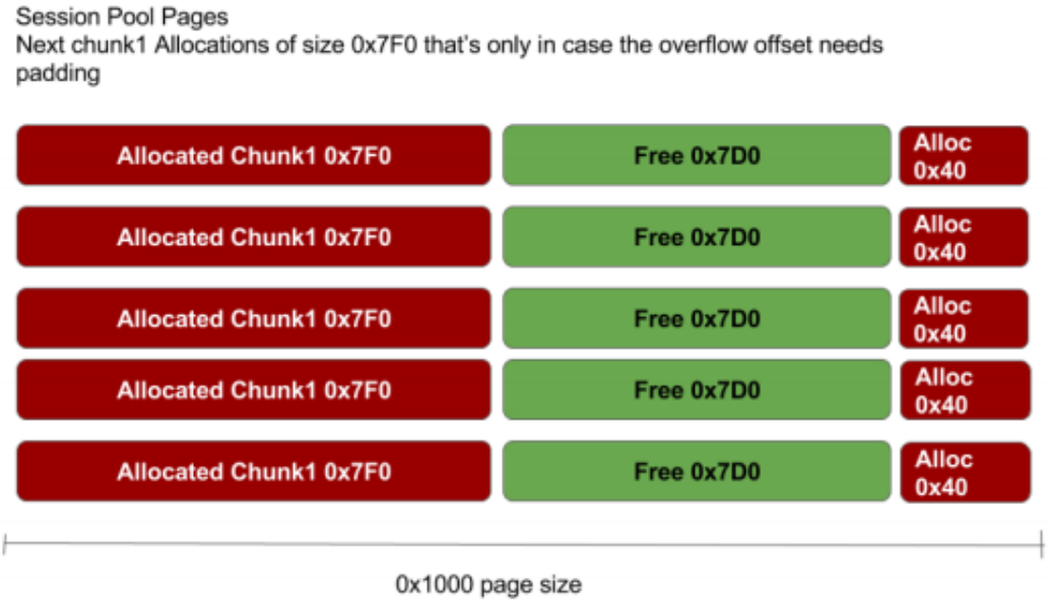
我们假设漏洞对象的大小为 0x40 字节（包含 Pool Header），则池页面初始 chunk 块的大小需要为 0x1000 – 0x40 = 0xFC0 字节（包含 Pool Header）。



之后再分配池页面中余下的 0x40 字节。



如果溢出利用中需要借助其它对象，则在漏洞对象的特定偏移处进行相应的分配操作。



Session Pool Pages
Next chunk2 Allocations of size 0x7D0 that's only in case the overflow offset needs padding

Session Pool Pages
Free 0x40 size objects to create a 0x40 memory hole at the end of Pool Page

Session Pool Pages
Allocate the vulnerable object, which will probably fall into one of the created memory holes.

0x03 池内存的破坏

引起池内存破坏的原因有很多，如释放后重用（UAF）、池的线性溢出以及池的越界写（OOBW）等。

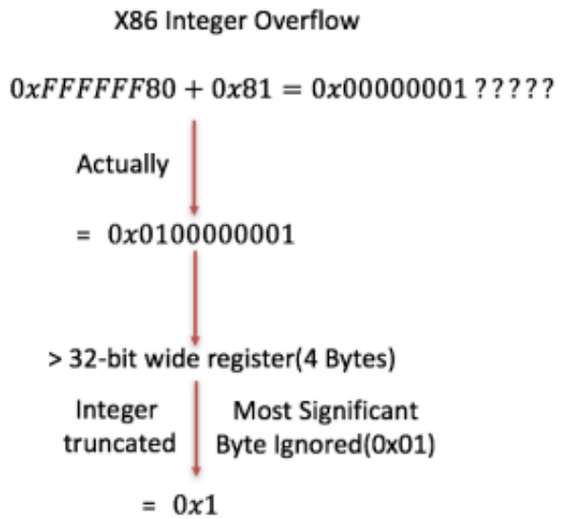
无符号整型溢出

无符号整型溢出是由于在计算过程中没有进行相应的检查使得计算结果超出了整型数所能表示的最大范围 MAX_UINT（0xFFFFFFFF，32 位），从而导致最终结果远小于预期，而按其后溢出值的不同使用情形又会造成不同的错误影响。

为了更好的理解无符号整型数的溢出，我们来看个例子：假设目标系统为 x86 架构，因此 UINT 占 4 个字节（32 位），考虑如下的加法运算：

```
0xFFFFFF80 + 0x81 = 00000001 ??
```

对 x86 系统来说上述计算结果为 0x1，然而真实的计算结果应该是 0x100000001，但这超出了 x86 系统上 4 字节 UNIT 数所能表示的范围，因此截断后得到结果 0x1。



而在 64 位系统中，虽然此概念仍然存在，但由于要求的数值过大，因此很难找到纯粹的 64 位整型数溢出的情况。不过，许多存在漏洞的函数在实际使用前会先将数值保存到 32 位寄存器中，所以又出现了前述解释的整数截断情形。

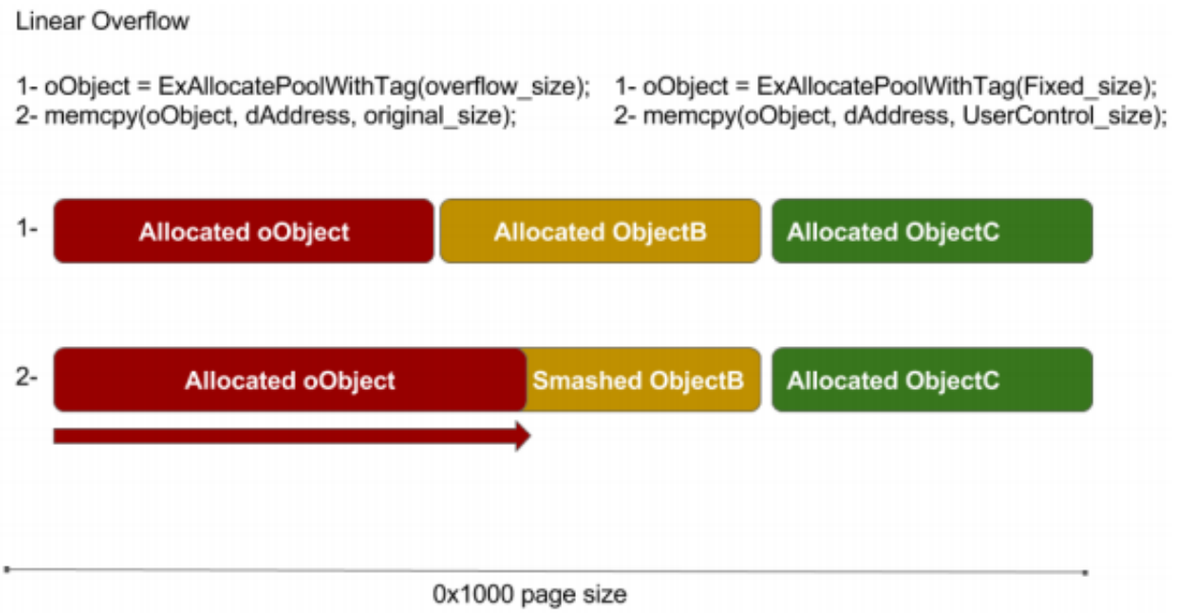
我们考虑如下的程序执行过程：

- 1. 入参变量为整型数，针对此整型数会进行一些运算操作；
- 2. 运算结果会导致整型数溢出；
- 3. 之后按此溢出值（较预期结果偏小）的大小进行新缓冲区分配操作；
- 4. 再按最初的入参整型数（未经过运算操作）进行相关操作：
  - a. 将原先内容拷贝到新申请的缓冲区（这会导致线性溢出）；
  - b. 向本应落在新分配缓冲区内的偏移进行写入操作（这会导致越界写，OOB Write）。

接着就来具体看一下。

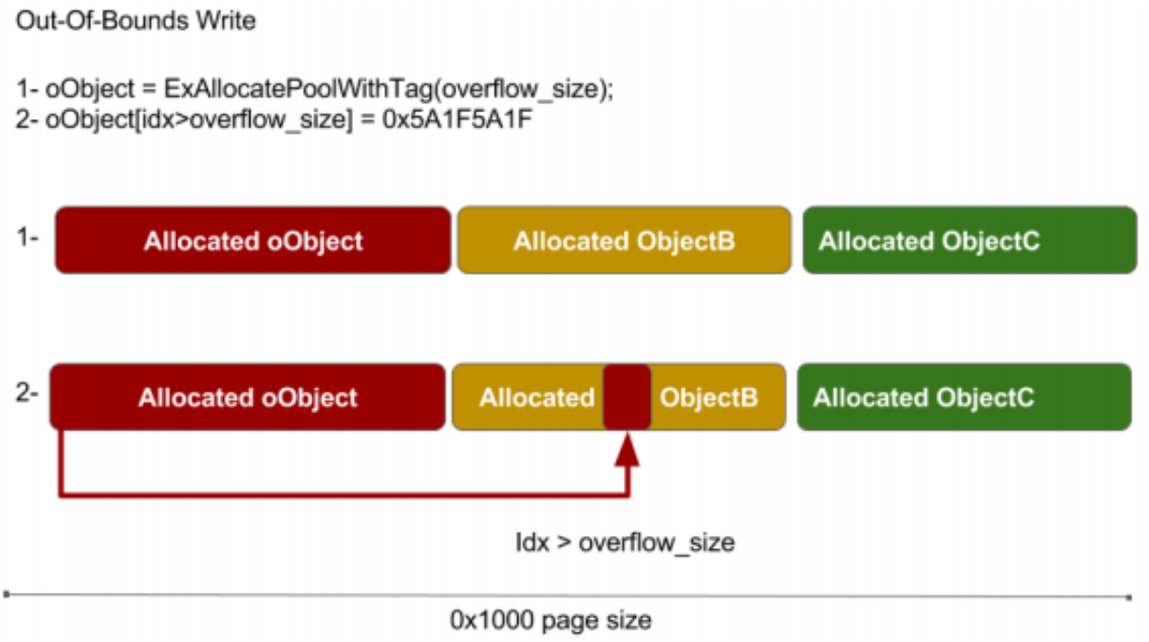
线性溢出

在对象数据拷贝过程中，如果没有对边界进行检查，那么就有可能发生线性溢出。其原因有多种，例如传给内存分配函数的大小是一个溢出值，这会导致新分配的空间偏小，而拷贝函数却按原先的大小将数据拷贝到新分配的内存空间，又或者对象本身是以固定大小分配的，而拷贝时使用的大小却是未经校验的用户输入值。



越界写（OOB Write）

对于越界写的情况，首先需有一对象，其大小大于某一特定值。而当该大小变量传给分配函数后发生了整型溢出，使结果较期望值偏小。随后，程序尝试在新分配对象中按预期索引值进行读写操作，但由于分配的大小值发生了溢出，导致该对象大小小于预期，从而造成了越界读写。



0x04 利用 GDI 对象获取 ring0 层 ARW Primitives

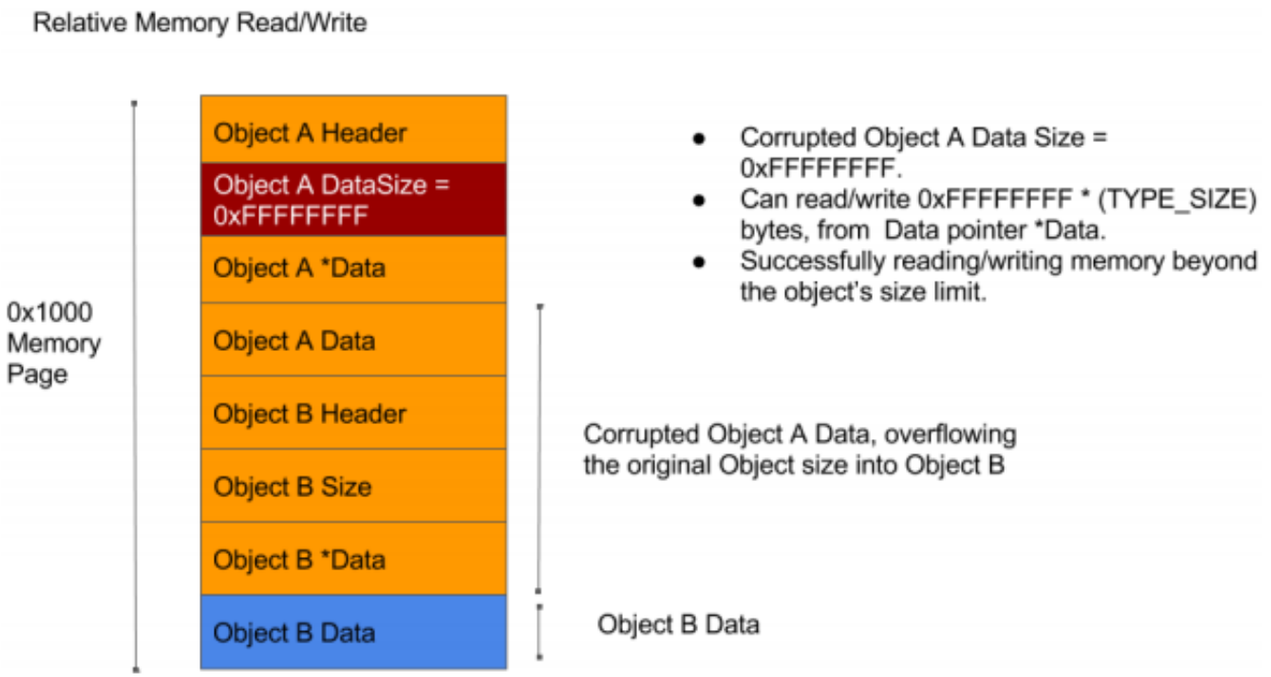
通常对 Exp 开发而言，某些经第一阶段内存破坏后的对象可被利用在获取第二阶段内存破坏的 primitives 中。这些对象一般拥有实现这些利用操作的特定成员，例如某些对象成员可以控制对象或对象中数据块的大小，因而能够实现相对的内存

读写操作，在某些情形中这足以实现 bug 的利用。更进一步，如果对象同时还拥有另一成员，即指向对象数据块的指针，那么就能将内存破坏的 primitives 转换成内存 ARW primitives，这会让利用程序的开发变得更加容易。要实现此利用技术通常需要借助两个对象，其中一个对象（manager）将用于修改第二个（通常是相邻）对象（worker）的数据指针，使其获得 ARW primitives（Game Over）。

在 Windows 内核中，GDI 对象恰好能够满足这些要求，如 Bitmap 对象利用技术，该技术首先是由 k33n 团队提出的[3]，后续被 Nicolas Economou 和 Diego Juarez 做了详细补充[4]。而我则足够幸运的发现了另一个能被利用的 GDI 对象，即 Palette 对象，Vulcan 团队同样也提及了此利用技术[10]。Palette 对象利用技术和 Bitmap 对象利用技术一样强大，也能够用于获取内核的任意内存读写能力。

相对内存读写

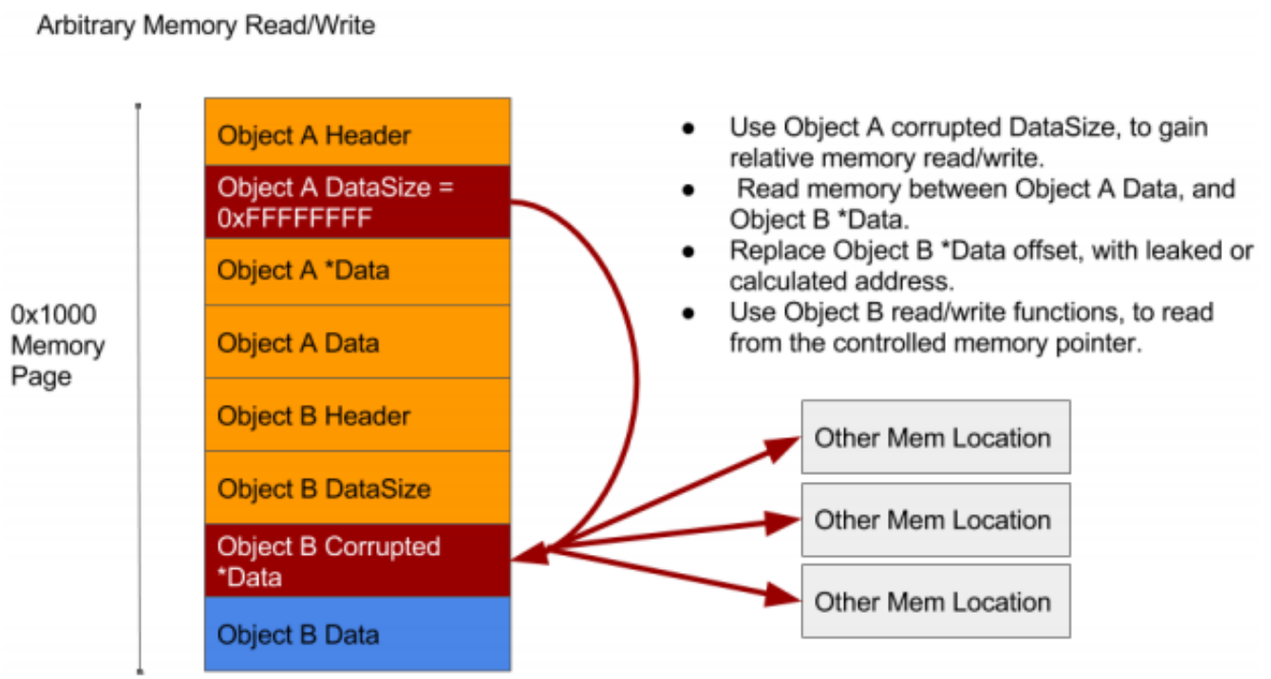
相对内存 RW primitives 允许我们对特定地址区域进行读写操作。通过破坏 GDI 对象的内存可增加其大小，这通常是触发 bug 后用于获取任意内存 RW primitives 所需迈出的第一步。



任意内存读写

对用于实现任意内存读写的对象，我们通常要求其拥有一个能够指向对象数据的指针成员。如果该指针被修改了，那么当调用相应对象数据读写函数时就会转而读写修改后指针所指向的地址，从而获取强大的任意内存 RW primitives。

为了便于理解，我们来考虑这样的 manager/worker 对象组合。对于 A 对象（manager），我们扩增了其大小，因而能够实现相对的越界读写，即实现对 B 对象（worker）数据指针的读写，而后将该指针替换为我们需要进行读写的地址，这就使得 B 对象的数据读写操作能够被我们所控制。



0x05 SURFOBJ - Bitmap 对象

Bitmap 对象在内核中对应的 Pool 标识为 `Gh?5` 或 `Gla5`，其结构体 `_SURFOBJ` 的定义在 msdn[5]、ReactOS 项目（32 位）[6] 以及 Diego Juarez 的博文（64 位）[7] 中有说明。

SURFOBJ 结构体

`SURFOBJ` 结构中最值得我们关注的成员当属 sizlBitmap，它是一个 SIZEL 结构体，系统由该变量来确定 bitmap 位图的长宽。而 pvScan0 和 pvBits 成员变量均表示指向 bitmap 位图的指针，按 bitmap 类型的不同，系统会选用二者之一。此外，在内存中 bitmap 位图通常位于 `SURFOBJ` 结构之后。

```
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;
```

```
typedef struct {
  ULONG64 dhsurf; // 0x00
  ULONG64 hsurf; // 0x08
  ULONG64 dhpdev; // 0x10
  ULONG64 hdev; // 0x18
  SIZEL sizlBitmap; // 0x20
  ULONG64 cjBits; // 0x28
  ULONG64 pvBits; // 0x30
  ULONG64 pvScan0; // 0x38
  ULONG32 lDelta; // 0x40
  ULONG32 iUniq; // 0x44
  ULONG32 iBitmapFormat; // 0x48
  USHORT iType; // 0x4C
  USHORT fjBitmap; // 0x4E
} SURFOBJ64; // sizeof = 0x50
```

分配

`CreateBitmap` 函数用于分配 Bitmap 对象，其定义如下：

```
HBITMAP CreateBitmap(
  _In_       int  nWidth,
  _In_       int  nHeight,
  _In_       UINT cPlanes,
  _In_       UINT cBitsPerPel,
  _In_ const VOID *lpvBits
);
```

Parameters

nWidth [in]
The bitmap width, in pixels.

nHeight [in]
The bitmap height, in pixels.

cPlanes [in]
The number of color planes used by the device.

cBitsPerPel [in]
The number of bits required to identify the color of a single pixel.

lpvBits [in]
A pointer to an array of color data used to set the colors in a rectangle of pixels. Each scan line in the rectangle must be word aligned (scan lines that are not word aligned must be padded with zeros). If this parameter is **NULL**, the contents of the new bitmap is undefined.

分配 2000 个 bitmap 对象：

```
for (int y = 0; y < 2000; y++) {
    HBITMAP bmp[y] = CreateBitmap(0x3A3, 1, 1, 32, NULL);
}
```

释放

`DeleteObject` 函数则用于 Bitmap 对象的释放：

```
BOOL DeleteObject(
  _In_ HGDIOBJ hObject
);
```

Parameters

hObject [in]
A handle to a logical pen, brush, font, bitmap, region, or palette.

```
DeleteObject(hBITMAP);
```

读内存

`GetBitmapBits` 函数可用于读取由 pvScan0 或 pvBits（取决于 bitmap 类型） 指针指向的 cBytes 字节的 bitmap 位图内容，其中 cBytes 的取值需小于 sizlBitmap.Width * sizlBitmap.Height * BitsPerPixel 乘积。

```
LONG GetBitmapBits(
  _In_  HBITMAP hbmp,
  _In_  LONG    cbBuffer,
  _Out_ LPVOID  lpvBits
);
```

Parameters

hbmp [in]
A handle to the device-dependent bitmap.

cbBuffer [in]
The number of bytes to copy from the bitmap into the buffer.

lpvBits [out]
A pointer to a buffer to receive the bitmap bits. The bits are stored as an array of byte values.

写内存

相对的，`SetBitmapBits` 函数则用于向 pvScan0 或 pvBits（取决于 bitmap 类型）指针指向的 bitmap 位图写入 cBytes 字节的内容，同样 cBytes 的取值也需小于 sizlBitmap.Width * sizlBitmap.Height * BitsPerPixel 的乘积。

```
LONG SetBitmapBits(
  _In_       HBITMAP hbmp,
  _In_       DWORD   cBytes,
  _In_ const VOID    *lpBits
);
```

Parameters

hbmp [in]
A handle to the bitmap to be set. This must be a compatible bitmap (DDB).

cBytes [in]
The number of bytes pointed to by the *lpBits* parameter.

lpBits [in]
A pointer to an array of bytes that contain color data for the specified bitmap.

相对内存读写 - sizlBitmap

sizlBitmap 成员变量为 SIZEL 类型的结构体，其中包含了 bitmap 位图的长宽，SIZEL 结构体和 SIZE 结构体是等价的，定义如下：

```
typedef struct tagSIZE {
  LONG cx;
  LONG cy;
} SIZE, *PSIZE, *LPSIZE;
```

后续的所有 Bitmap 对象操作，例如 bitmap 位图读写，都依赖此变量来计算 bitmap 位图的大小以执行对应操作，其中

Size = Width * Height * BitsPerPixel。通过破坏对象的 sizlBitmap 变量可实现相对内存读写。

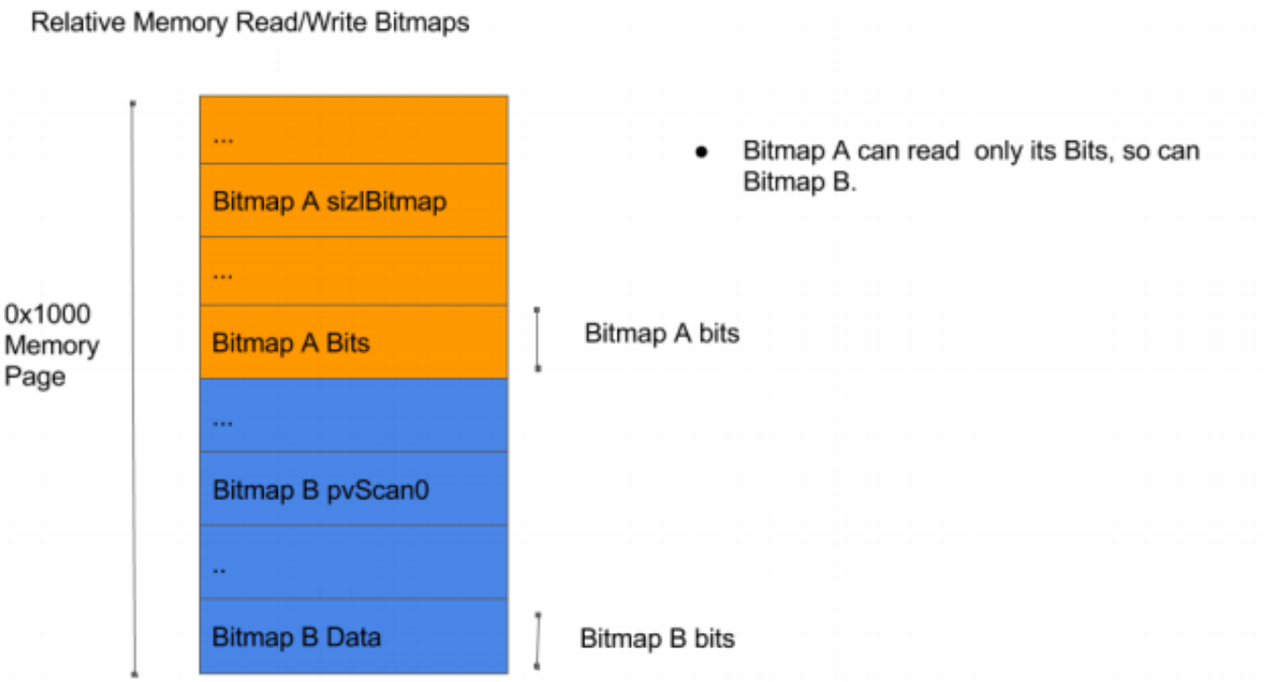
任意内存读写 - pvScan0/pvBits

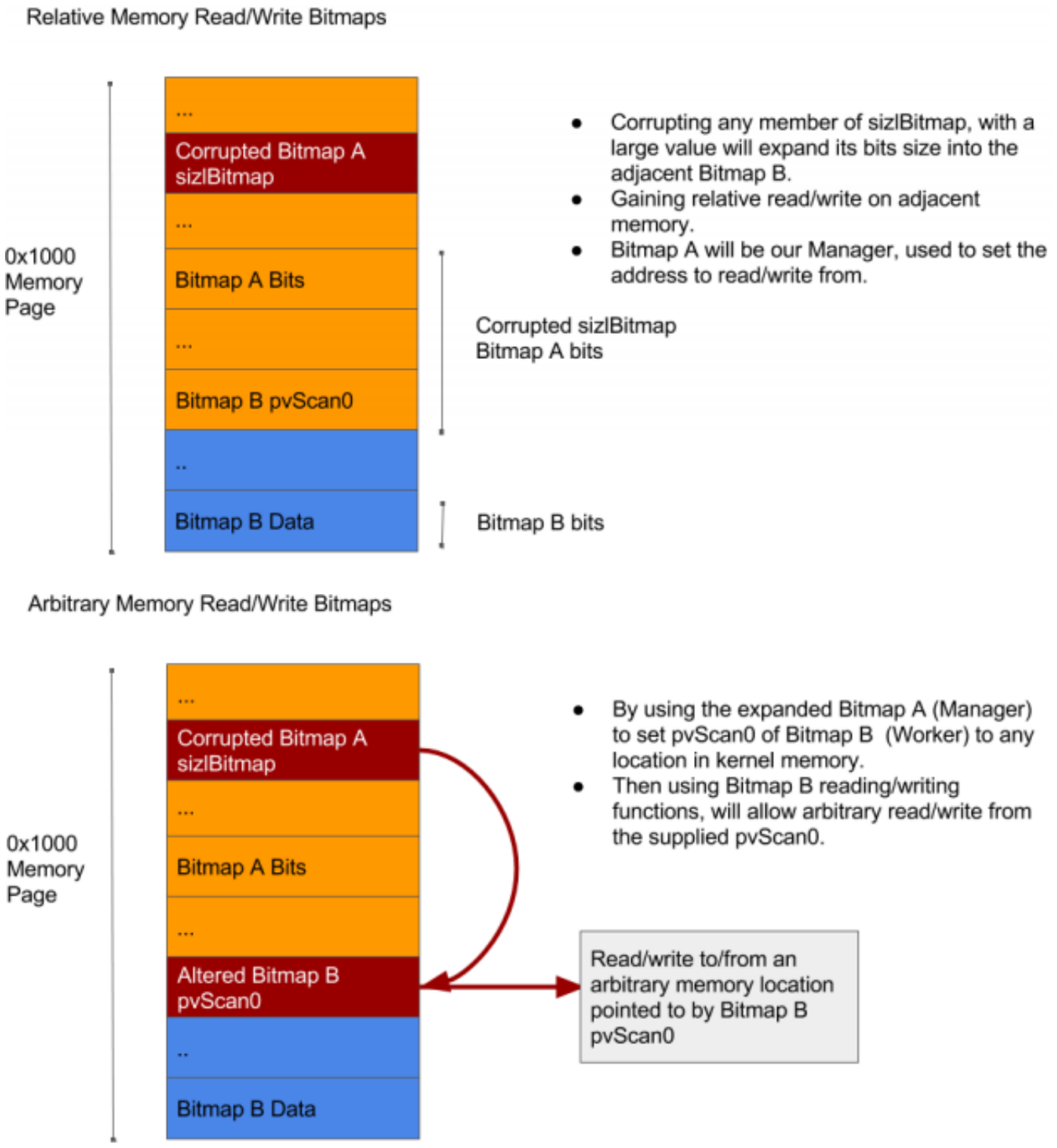
pvScan0 指针用于指向 bitmap 位图的第一行，但如果 bitmap 的格式为 `BMF_JPEG` 或 `BMF_PNG`，那么此成员变量会被置为 NULL，转而由 pvBits 指针来指向 bitmap 位图数据。这两个指针在读写 bitmap 位图数据时会用到，通过对其进行控制可以实现任意内存读写。

利用思路

Diego Juarez 和 Nicolas Economou 在之前的演讲中对借助 Manager/Worker 方式的 Bitmap 对象利用技术做了详尽分析，其思路是通过控制 Manager Bitmap 对象的 sizelBitmap 或 pvScan0 成员，从而达到控制 Worker Bitmap 对象 pvScan0 成员的目的，最终实现内核任意内存读写（ARW primitives）。

我们这里给出的思路是通过控制 Manager Bitmap 对象的 sizlBitmap 成员，以扩增 bitmap 的大小来获取相对内存读写的能力，接着再控制相邻 Worker Bitmap 对象的 pvScan0 指针达到任意内存读写。





0x06 XEPALOBJ - Palette 对象

下面我们将介绍基于 Palette 对象的新利用技术。此对象在内核中的 Pool 标识为 `Gh?8` 或 `Gla8`，调试中相应的符号名为 `_PALETTE`、`XEPALOBJ` 或 `PALOBJ`。msdn 上并没有关于该对象的公开内核结构信息，但我们可以在 ReactOS 项目中[8]找到其 x86 版的定义，而在 Deigo Juarez 的 WinDbg 插件项目 GDIObjDump 中[9]则可同时找到 x86 版和 x64 版的定义。

PALETTE 结构体

对于 `XEPALOBJ` 结构体，我们比较感兴趣的成员变量是 cEntries，它表示 PALETTEENTRY 数组中的元素个数，此外还有 pFirstColor 成员变量，它是指向 PALETTEENTRY 数组 apalColors 的指针，可以看到，apalColors 表示的数组位于此

结构体的尾部。

```
typedef struct _PALETTE
{
    BASEOBJECT      BaseObject;     // 0x00

    FLONG           flPal;          // 0x10
    ULONG           cEntries;       // 0x14
    ULONG           ulTime;         // 0x18
    HDC             hdcHead;        // 0x1c
    HDEVPPAL        hSelected;      // 0x20,
    ULONG           cRefhpal;       // 0x24
    ULONG           cRefRegular;    // 0x28
    PTRANSLATE      ptransFore;     // 0x2c
    PTRANSLATE      ptransCurrent;  // 0x30
    PTRANSLATE      ptransOld;      // 0x34
    ULONG           unk_038;        // 0x38
    PFN             pfnGetNearest;  // 0x3c
    PFN             pfnGetMatch;    // 0x40
    ULONG           ulRGBTime;      // 0x44
    PRGB555XL       pRGBXlate;      // 0x48
    PALETTEENTRY    *pFirstColor;   // 0x4c
    struct _PALETTE *ppalThis;      // 0x50
    PALETTEENTRY    apalColors[1]; // 0x54
} PALETTE, *PPALETTE;
```

```
typedef struct _PALETTE64
{
    BASEOBJECT      BaseObject;     // 0x00

    FLONG           flPal;          // 0x18
    ULONG           cEntries;       // 0x1C
    ULONGLONG       ullTime;        // 0x20
    HDC             hdcHead;        // 0x28
    HDEVPPAL        hSelected;      // 0x30
    ULONG           cRefhpal;       // 0x38
    ULONG           cRefRegular;    // 0x3c
    PTRANSLATE      ptransFore;     // 0x40
    PTRANSLATE      ptransCurrent;  // 0x48
    PTRANSLATE      ptransOld;      // 0x50
    ULONGLONG       unk_038;        // 0x58
    PFN             pfnGetNearest;  // 0x60
    PFN             pfnGetMatch;    // 0x68
    ULONGLONG       ullRGBTime;     // 0x70
    PRGB555XL       pRGBXlate;      // 0x78
    PALETTEENTRY    *pFirstColor;   // 0x80
    struct _PALETTE *ppalThis;      // 0x88
    PALETTEENTRY    apalColors[1]; // 0x90
} PALETTE64, *PPALETTE64;
```

分配

`CreatePalette` 函数用于分配 Palette 对象，唯一的入参 lplgpl 为 LOGPALETTE 结构体指针类型，对 x86 系统其分配大小需不小于 0x98 字节，相应的对 x64 系统其分配大小需不小于 0xD8 字节。

```
HPALETTE CreatePalette(
  _In_ const LOGPALETTE *lplgpl
);
```

Parameters

lplgpl [in]

A pointer to a **LOGPALETTE** structure that contains information about the colors in the logical palette.

```
typedef struct tagLOGPALETTE {
  WORD         palVersion;
  WORD         palNumEntries;
  PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

Members

palVersion
The version number of the system.

palNumEntries
The number of entries in the logical palette.

palPalEntry
Specifies an array of **PALETTEENTRY** structures that define the color and usage of each entry in the logical palette.

不论 x86 系统还是 x64 系统，PALETTEENTRY 结构都是占 4 个字节：

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
} PALETTEENTRY;
```

Members

peRed
The red intensity value for the palette entry.

peGreen
The green intensity value for the palette entry.

peBlue
The blue intensity value for the palette entry.

peFlags
Indicates how the palette entry is to be used. This member may be set to 0 or one of the following values.

分配 2000 个 Palette 对象：

```
LOGPALETTE *lPalette;
lPalette = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) + (0x1E3 - 1) * sizeof(PALETTEENTRY));
lPalette->palNumEntries = 0x1E3;
lPalette->palVersion = 0x0300;
for (int k = 0; k < 2000; k++) {
    hps[k] = CreatePalette(lPalette);
}
```

释放

而 Palette 对象的释放则由 DeleteObject 函数来完成：

```
DeleteObject(hPALETTE);
```

读内存

`GetPaletteEntries` 函数被用来读取 Palette 对象中的内容，即对应 hpal 句柄表示的 XEPALOBJ 结构中 pFirstColor 指针指向的 apalColors 数组自偏移 iStartIndex 开始的 nEntries 个元素，并将其保存到缓冲区 lppe 上。函数定义如下：

```
UINT GetPaletteEntries(
  _In_  HPALETTE       hpal,
  _In_  UINT           iStartIndex,
  _In_  UINT           nEntries,
  _Out_ LPPALETTEENTRY lppe
);
```

Parameters

hpal [in]
A handle to the logical palette.

iStartIndex [in]
The first entry in the logical palette to be retrieved.

nEntries [in]
The number of entries in the logical palette to be retrieved.

lppe [out]
A pointer to an array of **PALETTEENTRY** structures to receive the palette entries. The array must contain at least as many structures as specified by the *nEntries* parameter.

写内存

相对的， `SetPaletteEntries` 和 `AnimatePalette` 这两个函数可用来向 Palette 对象写入内容，即将缓冲区 lppe 上的 nEntries 个元素写入 hpal 句柄表示的 XEPALOBJ 结构中 pFirstColor 指针指向的 apalColors 数组自偏移 iStart 或 iStartIndex 开始的位置。

```
UINT SetPaletteEntries(
  _In_       HPALETTE     hpal,
  _In_       UINT         iStart,
  _In_       UINT         cEntries,
  _In_ const PALETTEENTRY *lppe
);
```

Parameters

hpal [in]
A handle to the logical palette.

iStart [in]
The first logical-palette entry to be set.

cEntries [in]
The number of logical-palette entries to be set.

lppe [in]
A pointer to the first member of an array of **PALETTEENTRY** structures containing the RGB values and flags.

```
BOOL AnimatePalette(
  _In_       HPALETTE     hpal,
  _In_       UINT         iStartIndex,
  _In_       UINT         cEntries,
  _In_ const PALETTEENTRY *ppe
);
```

Parameters

hpal [in]
A handle to the logical palette.

iStartIndex [in]
The first logical palette entry to be replaced.

cEntries [in]
The number of entries to be replaced.

ppe [in]
A pointer to the first member in an array of **PALETTEENTRY** structures used to replace the current entries.

相对内存读写 - cEntries

`XEPALOBJ` 结构体的 cEntries 成员用于表示 Palette 对象中 apalColors 数组元素的个数，若将其覆盖为一个大的数值，那么借助破坏后的 Palette 对象可以实现内存的越界读写。

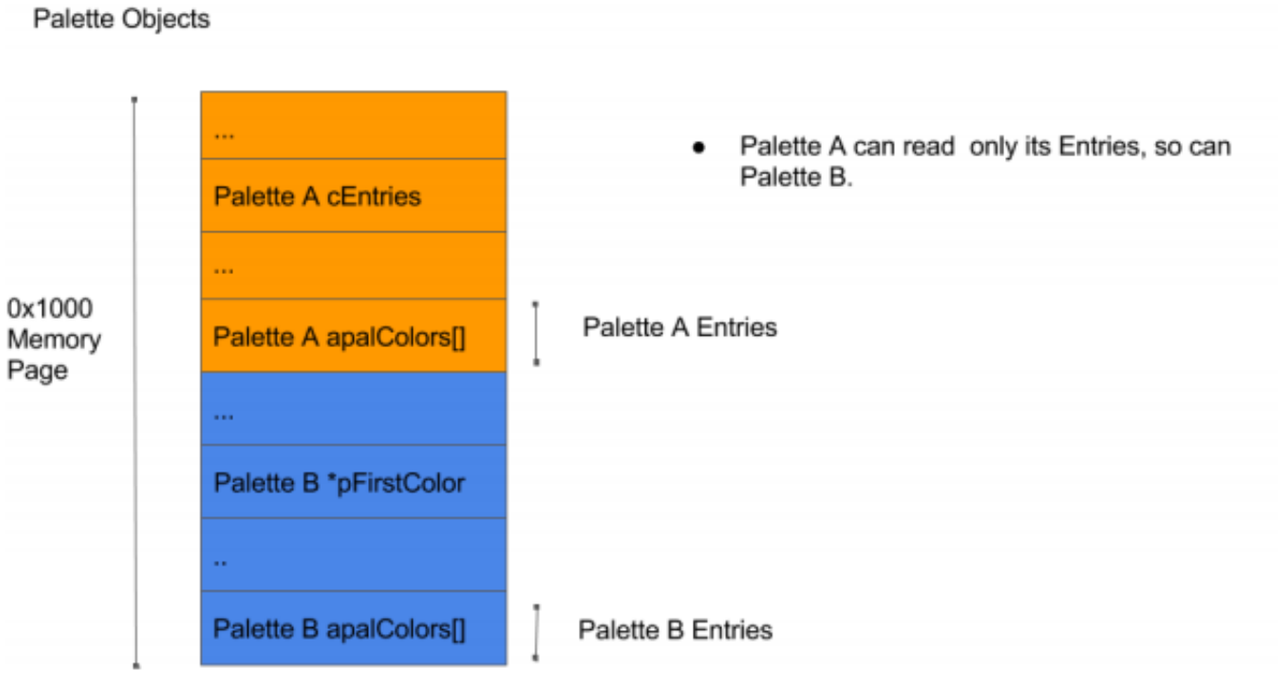
任意内存读写 - pFirstColor

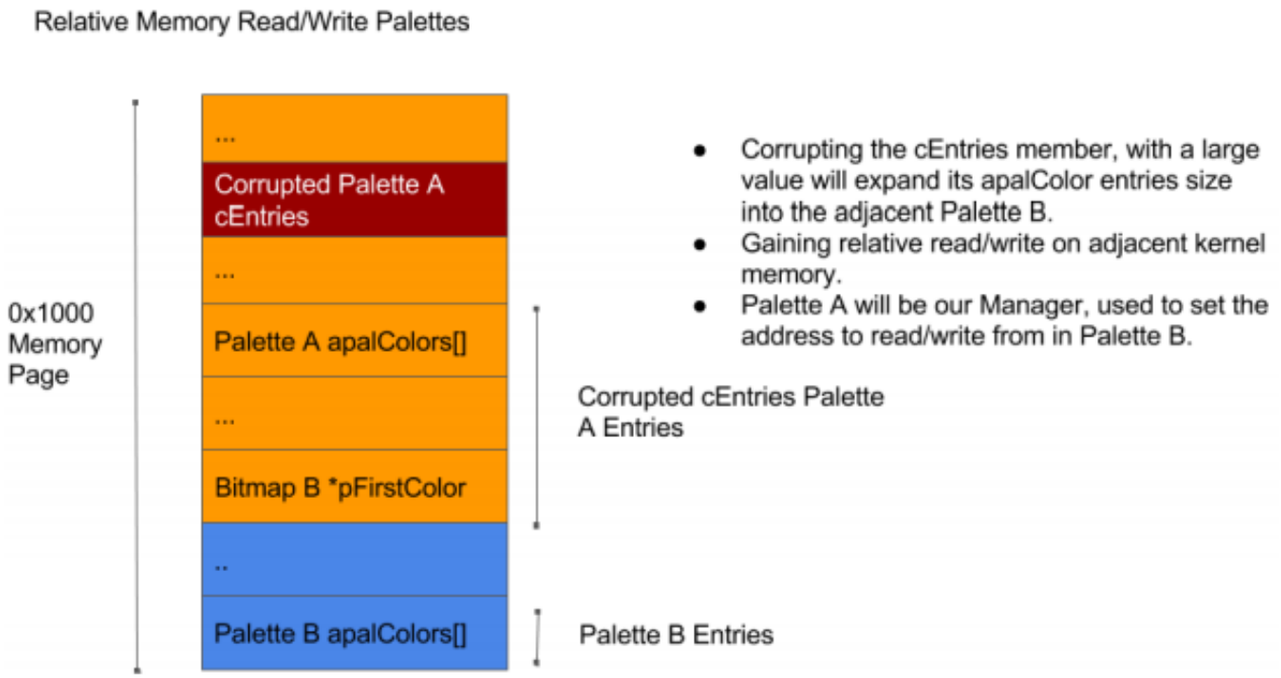
pFirstColor 指针指向的是 Palette 对象中 apalColors 数组的起始位置，通过控制该指针，可以实现内核态下内存的任意读写。

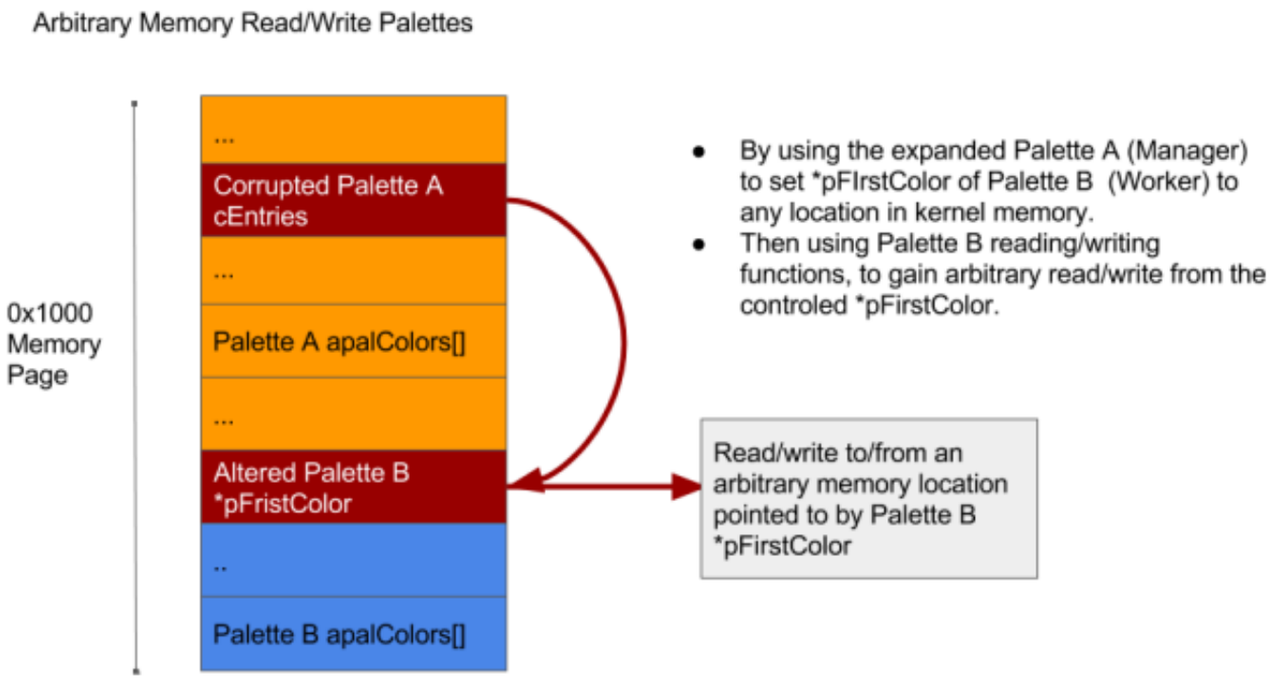
利用思路

针对 Palette 对象的利用思路和之前讨论的 Bitmap 对象利用思路是类似的，通过控制 Manager Palette 对象的 cEntries 或 pFirstColor 成员来达到控制相邻 Worker Palette 对象 pFirstColor 成员的目的，从而获取内核下的 ARW primitives。

我们这里讨论 Manager Palette 对象中 cEntries 成员可控的情况，通过对 cEntries 进行溢出使得 Manager Palette 对象获取相对内存读写的能力，再借此修改相邻 Worker Palette 对象的 pFirstColor 指针可实现任意内存读写。







0x07 基于 Palette 对象利用技术的几点限制

首先，在 x86 系统中要求 cEntires 成员溢出后得到的结果必须大于 0x26，相应的在 x64 系统中必须大于 0x36，这是因为在 `XEPALOBJ` 对象分配时，x86 系统要求其大小不小于 0x98 字节，而 x64 系统要求其大小不能小于 0xd8 字节。例如 cEntires 成员经溢出后由 0x1 变为 0x6，但这显然不满足条件，该大小 0x6 * 0x4 = 0x18 字节小于所要求分配 Palette 对象时的最小值。

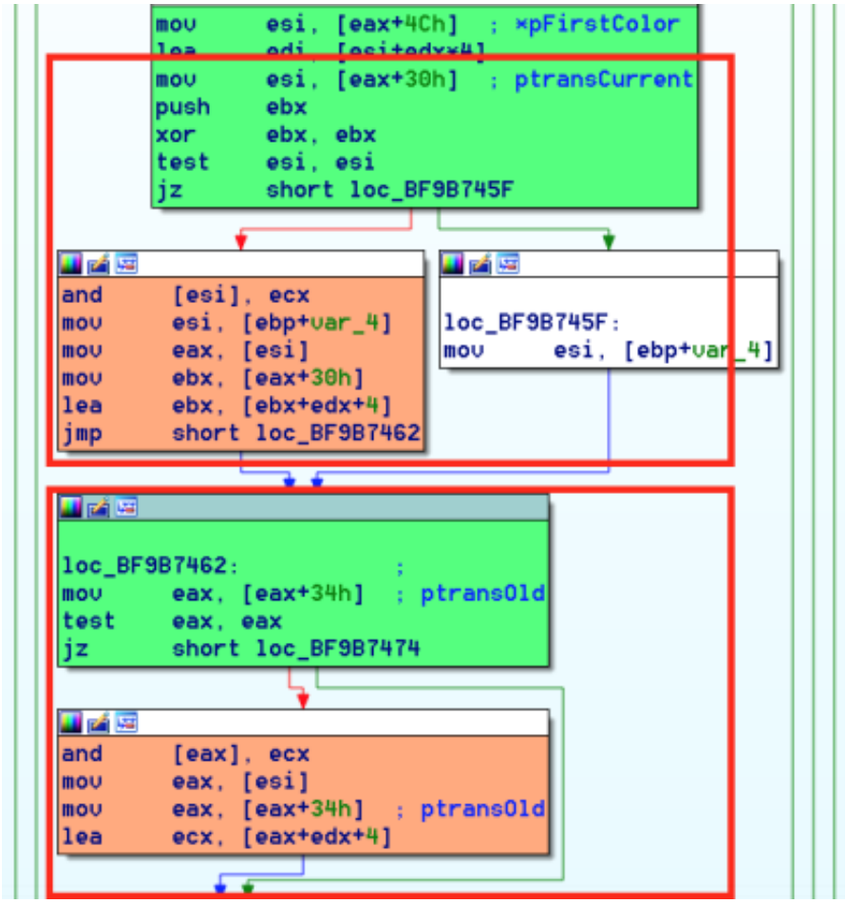
其次，如果利用程序通过 `SetPaletteEntries` 函数向内存写入数据，那么需保证 `XEPALOBJ` 结构中的 hdcHead、ptransOld 以及 ptransCurrent 成员不会被覆盖掉。

X86	X64

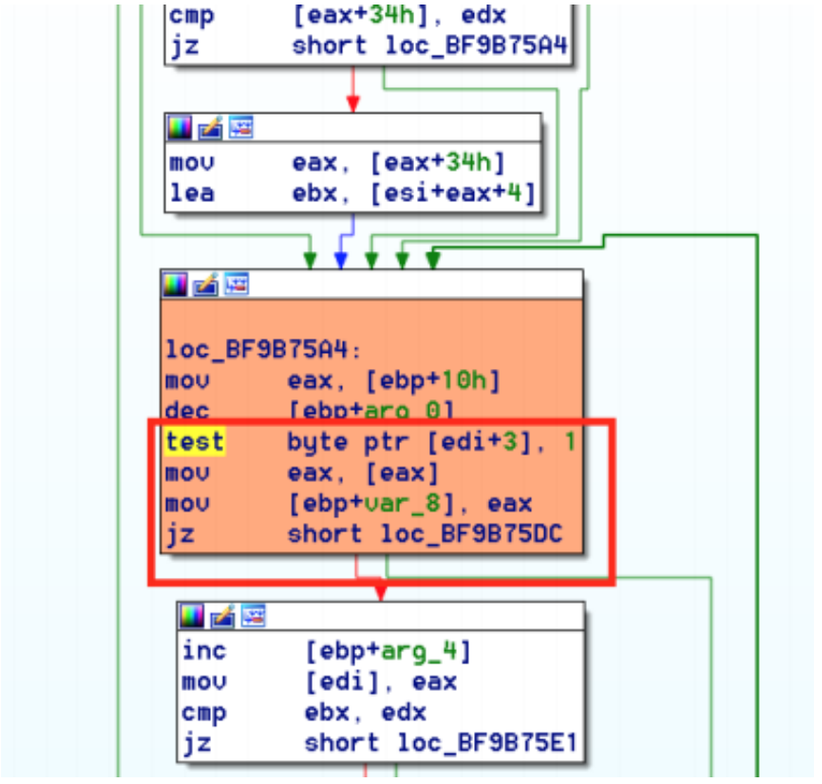
typedef struct _PALETTE64
{
..
HDC hdcHead;
// 0x1c
…
PTRANSLATE ptransCurrent; // 0x30
PTRANSLATE ptransOld;
// 0x34
 	
…
} PALETTE, *PPALETTE;
	typedef struct _PALETTE64
	{
	..
	HDC hdcHead;
	// 0x28
	…
	PTRANSLATE ptransCurrent; // 0x48
	PTRANSLATE ptransOld;
	// 0x50
	
	…
	} PALETTE64, *PPALETTE64;

ring3 层的 SetPaletteEntries 调用会经由 NTSetPaletteEntries 至 GreSetPaletteEntries 函数，此时会对 hdcHead 成员进行检查，如果该值不为零，则程序执行流程会报错或直接蓝屏死机，即下图黄色区域所示。

不过在此之前 GreSetPaletteEntries 还会先调用 XEPALOBJ::ulSetEntries 函数对 pTransCurrent 和 pTransOld 成员进行检查，如果它们非零，程序会进入下图所示的橘色区域，这有可能会导致蓝屏死机。



最后我们看下使用 AnimatePalettes 函数向 Palette 对象进行写操作的情况，唯一限制是要求 pFirstColor 指针所指内容的最高字节为奇数，对应的 XEPALOBJ::ulAnimatePalette 函数代码段如下。虽然不会导致蓝屏死机，但这使得我们无法完成写入操作。



0x08 Token 的替换

内核借助 `_EPROCESS` 结构来表示系统上运行的每一个进程，该结构包含很多重要成员，例如 ImageName、SecurityToken、ActiveProcessLinks 以及 UniqueProcessId，这些成员的偏移值因系统版本而异。

Windows 8.1 x64：

```
kd> dt nt!_EPROCESS UniqueProcessId ActiveProcessLinks Token
   +0x2e0 UniqueProcessId     : Ptr64 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY
   +0x348 Token              : _EX_FAST_REF
```

Windows 7 SP1 x86：

```
0: kd> dt _EPROCESS UniqueProcessId ActiveProcessLinks Token
dtx is unsupported for this scenario.  It only recognizes dtx
ntdll!_EPROCESS
   +0x0b4 UniqueProcessId     : Ptr32 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY
   +0x0f8 Token              : _EX_FAST_REF
```

另外，内核中 SYSTEM 进程所对应的 `EPROCESS` 结构地址可通过如下方式计算得到：

```
KernelEPROCESSAddress = kernelNTBase + (PSInitialSystemProcess() - UserNTImageBase)
```

SecurityToken

SecurityToken 表示当前进程所持有的安全级别标识，当进程请求获取特定权限时，系统会借此判断其是否拥有所请求资源的权限。

ActiveProcessLinks

ActiveProcessLinks 是一个 `LIST_ENTRY` 对象，可借此遍历各进程对应的 `EPROCESS` 结构。

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

UniqueProcessId

UniqueProcessId 表示进程 PID。

步骤

1. 获取内核中 SYSTEM 进程对应的 `EPROCESS` 结构地址；
2. 借助 Read primitive 得到相应的 SecurityToken 和 ActiveProcessLinks；
3. 遍历 ActiveProcessLinks 得到当前进程的 `EPROCESS` 结构地址，即 ActiveProcessLinks->Flink.UniqueProcessId 和 GetCurrentProcessId() 的值相同；
4. 借助 Write primitive 将当前进程的 SecurityToken 替换为 SYSTEM 进程的 SecurityToken。

**参考部分详见原文*