Abusing GDI for ring0 exploit primitives: Evolution

By Nicolas A. Economou
Who I am

- Senior Security Researcher at Blue Frost Security

- Worked 12 years as an exploit writer

- The last 10 years, specialized in Windows Kernel exploitation

- +17 years of low level programming (ASM/C)
Agenda

1. EoPs
2. Sandbox mitigations
3. Bitmap exploitation
4. GDI exploitation in Win10 RS3
5. ACG bypass
6. Demo – MS Edge sandbox escape
EoPs
EoPs

- EoP (Elevation of Privilege) ... aka “Privilege Escalation”

- Usually, attacks are done locally

- Historically, used to elevate privileges from unprivileged users
EoPs

- Privilege escalation exploits became more important with the introduction of sandbox technology

- Sandboxed browsers:
  - Chrome, Edge, IE, Firefox

- Sandboxed office tools:
  - Word, PowerPoint, Excel, Adobe Reader, etc
Sandbox Mitigations

- If the app is owned, the attacker has less privileges

- Sandboxes usually run in Low Integrity Level/AppContainer

- The idea is to restrict the access to the system and mitigate some kind of EoPs
Sandbox Mitigations

- Execution restrictions
  - No program can’t be executed from the sandbox (Edge/Chrome)

- Library restrictions (ProcessSignaturePolicy)
  - Only system libraries can be loaded from the sandbox (Edge/Chrome)

- File system restrictions
  - Writable dir: “C:\Users\XXX\AppData\Local\Temp\Low”
- Call restrictions
  - E.g. `NtQuerySystemInformation` can’t get kernel base address

- Syscall restrictions (`ProcessSystemCallDisablePolicy`)
  - E.g. “win32k” syscall prohibition (used by the Chrome renderer process)
Sandbox Mitigations

- Attackers usually want to escape from sandboxes ;-) 

- Kernel Privilege Escalation exploits are ideal for that 

- E.g. May 2017: 0-day exploit for MS Word was detected in the wild (EPS exploit + Kernel exploit (CVE-2017-0263))
Arbitrary write
Arbitrary write

- Aka: Write What Where (www)

- Result of exploiting a binary bug

- Write one value (controllable or not) at an arbitrary address
Arbitrary write

- Used a lot in Kernel EoPs

- Usually combined with some kind of memory leak (bypass KASLR!)

- The idea is to get a kernel read/write primitive from user mode
Arbitrary write

- Getting a r/w primitive **avoid** to deal with SMEP (non EIP/RIP manipulation)

- Finally, get **SYSTEM** privileges (Token Stealer technique)
GDI objects
GDI objects

GDI objects support only one handle per object. Handles to GDI objects are private to a process. That is, only the process that created the GDI object can use the object handle.

GDI objects

- Graphic Objects used by Windows
- Instanced via APIs (user mode)
- Processed in kernel mode
- Bitmaps, Brushes, DCs, Metafiles, Fonts, Palettes, Pens and Regions
GDI exploitation history

-In April 2015, Keen Team mentioned GDI objects in "This Time Font hunt you down in 4 bytes"

-A TTF kernel heap overflow was described

-Bitmaps were used for the exploitation
GDI exploitation history

-In July 2015, Hacking Team was hacked

-Some kernel 0-day exploits were leaked

-One of them used GDI objects for the exploitation
GDI exploitation history

- In October 2015 (Ekoparty), Diego Juarez (Pnx) from Core Security presented the Bitmaps technique in detail at "Abusing GDI for ring0 exploit primitives"

- In September 2016 (Ekoparty), Diego Juarez (Pnx) and I presented memory leaks and improvements at "Abusing GDI for ring0 exploit primitives: Reloaded"

- In July 2017 (Defcon), Saif El-Sherei from SensePost extended the life of the GDI object exploitation
Bitmaps
Bitmaps

- Created by CreateBitmap (gdi32.dll)

The `CreateBitmap` function creates a bitmap with the specified width, height, and color format (color planes and bits-per-pixel).

**Syntax**

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

- \( nWidth \times nHeight \times cBitsPerPel = \text{data size} \)

- `lpvBits` parameter contains our data

- Our data is allocated in kernel space
Bitmaps

- Bitmaps in kernel (SURFACE.SURFOBJ structure)

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

```c
typedef struct _SURFOBJ
{
    DHSURF  dhsurf;
    HSURF   hsurf;
    DHPDEV  dhpdev;
    HDEV    hdev;
    SIZEL   siz1Bitmap;
    ULONG   cjBits;
    PVOID   pvBits;
    PVOID   pvScan0;
    LONG    lDelta;
    ULONG   iUniq;
    ULONG   iBitmapFormat;
    USHORT  iType;
    USHORT  fBfBitmap;
    // size
} SURFOBJ, *PSURFOBJ;
```
- **PvBits/PvScan0** properties point to our data

- The data is consecutive to the SURFACE structure (header + data)

- It means that only a kernel allocation is needed to contain a Bitmap
Bitmaps

- Our kernel data can be read/written by using GetBitmapBits/SetBitmapBits

- Bitmaps variants:
  - CreateCompatibleBitmap
  - CreateBitmapIndirect
  - CreateDiscardableBitmap
  - CreateDIBitmap
Abusing Bitmaps
Abusing Bitmaps

- Used to get read/write primitives
- Easy to manipulate/abuse
- Their addresses can be leaked from user mode at any integrity level
Abusing Bitmaps

PvScan0 technique (2015)

<table>
<thead>
<tr>
<th>SURFOBJ</th>
<th>BITMAP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct _SURFOBJ</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>DHSURF dhsurf;</td>
<td></td>
</tr>
<tr>
<td>HSURF hsurf;</td>
<td></td>
</tr>
<tr>
<td>DHPDEV dhpdev;</td>
<td></td>
</tr>
<tr>
<td>HDEV hdev;</td>
<td></td>
</tr>
<tr>
<td>SIZEL sizlBitmap;</td>
<td></td>
</tr>
<tr>
<td>ULONG cJBits;</td>
<td></td>
</tr>
<tr>
<td>PVOID pvBits;</td>
<td></td>
</tr>
<tr>
<td>PVOID pvScan0;</td>
<td></td>
</tr>
<tr>
<td>LONG lDelta;</td>
<td></td>
</tr>
<tr>
<td>ULONG iUniq;</td>
<td></td>
</tr>
<tr>
<td>ULONG iBitmapFormat;</td>
<td></td>
</tr>
<tr>
<td>USHORT iType;</td>
<td></td>
</tr>
<tr>
<td>USHORT fJBitmap;</td>
<td></td>
</tr>
<tr>
<td>// size</td>
<td></td>
</tr>
<tr>
<td>} SURFOBJ, *PSURFOBJ;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SURFOBJ</th>
<th>BITMAP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct _SURFOBJ</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>DHSURF dhsurf;</td>
<td></td>
</tr>
<tr>
<td>HSURF hsurf;</td>
<td></td>
</tr>
<tr>
<td>DHPDEV dhpdev;</td>
<td></td>
</tr>
<tr>
<td>HDEV hdev;</td>
<td></td>
</tr>
<tr>
<td>SIZEL sizlBitmap;</td>
<td></td>
</tr>
<tr>
<td>ULONG cJBits;</td>
<td></td>
</tr>
<tr>
<td>PVOID pvBits;</td>
<td></td>
</tr>
<tr>
<td>PVOID pvScan0;</td>
<td></td>
</tr>
<tr>
<td>LONG lDelta;</td>
<td></td>
</tr>
<tr>
<td>ULONG iUniq;</td>
<td></td>
</tr>
<tr>
<td>ULONG iBitmapFormat;</td>
<td></td>
</tr>
<tr>
<td>USHORT iType;</td>
<td></td>
</tr>
<tr>
<td>USHORT fJBitmap;</td>
<td></td>
</tr>
<tr>
<td>// size</td>
<td></td>
</tr>
<tr>
<td>} SURFOBJ, *PSURFOBJ;</td>
<td></td>
</tr>
</tbody>
</table>
Abusing Bitmaps

Extending Consecutive Bitmaps technique (2016)

```
typedef struct _SURFOBJ
{
    DHSURF dhsurf;
    HSURF hsurf;
    DHPDEV dhpdev;
    HDEV hdev;
    SIZEL siz1Bitmap;
    ULONG cjBits;
    PVOID pvBits;
    PVOID pvScan0;
    LONG lDelta;
    ULONG iUniq;
    ULONG iBitmapFormat;
    USHORT iType;
    USHORT fjBitmap;
    // size
} SURFOBJ, *PSURFOBJ;
```

SetBitmapBits

```
typedef struct _SURFOBJ
{
    DHSURF dhsurf;
    HSURF hsurf;
    DHPDEV dhpdev;
    HDEV hdev;
    SIZEL siz1Bitmap;
    ULONG cjBits;
    PVOID pvBits;
    PVOID pvScan0;
    LONG lDelta;
    ULONG iUniq;
    ULONG iBitmapFormat;
    USHORT iType;
    USHORT fjBitmap;
    // size
} SURFOBJ, *PSURFOBJ;
```

GetBitmapBits

```
typedef struct _SURFOBJ
{
    DHSURF dhsurf;
    HSURF hsurf;
    DHPDEV dhpdev;
    HDEV hdev;
    SIZEL siz1Bitmap;
    ULONG cjBits;
    PVOID pvBits;
    PVOID pvScan0;
    LONG lDelta;
    ULONG iUniq;
    ULONG iBitmapFormat;
    USHORT iType;
    USHORT fjBitmap;
    // size
} SURFOBJ, *PSURFOBJ;
```

arb.write

R/W primitive
Leaking Bitmaps
Leaking Bitmaps

- Until Windows 10 v1511 (Threshold 2)

- Leaking kernel addresses by reading user32!gSharedInfo structure

- Killed in RS1
Leaking Bitmaps

- Until Windows 10 RS1 (Anniversary Update)

- Indirect leak by using AcceleratorTables (Free List abusing)

- Leaking by reading user32!gSharedInfo structure

- Killed in RS2
Leaking Bitmaps

- Until Windows 10 RS2 (Creators Update)

- Indirect leak by using RegisterClass with WNDCLASSEX.lpszMenuName (Free List abusing)

- Leaking by reading user32!gSharedInfo structure and more until you find tagCLS.lpszMenuName

- Indirect leak killed in RS3
Fall Creators Update (RS3)
Fall Creators Update

- Released in October, 2017

- Current version: Insider Preview 16299.19

- Some security mitigations were added
Fall Creators Update

- Bitmap headers separated from Bitmap data

- Data is no longer contiguous to header

- \texttt{PvScan0/PvBits} now point to a different heap than headers
Fall Creators Update

- Bitmap headers moved to some kind of heap isolation!
- No way to predict its address until now
- Bitmap technique killed :-(

Evolution
Evolution

- In Defcon 2017, “Demystifying Kernel Exploitation by Abusing GDI Objects”

- *Saif El-Sherei* presented a GDI object alternative for Bitmap exploitation

- It’s still working in RS3
Evolution

- Bitmaps are replaced by Palettes

CreatePalette function

The `CreatePalette` function creates a logical palette.

Syntax

```cpp
HPALETTE CreatePalette(
    __In__ const LOGPALETTE *lplogp1
);
```
Evolution

-Same idea/techniques as for Bitmaps

-Same way to leak their kaddresses

-Header + data placed together
typedef struct _PALETTE {
    BASEOBJECT BaseObject;
    FLONG fLPal;
    ULONG cEntries;
    ULONG ulTime;
    HDC hdcHead;
    HDEVPPAL hSelected;
    ULONG cRefhpal;
    ULONG cRefRegular;
    PTRANSLATE ptransFore;
    PTRANSLATE ptransCurrent;
    PTRANSLATE ptransOld;
    ULONG unk_038;
    PFN pfnGetNearest;
    PFN pfnGetMatch;
    ULONG u1RGBTime;
    PRGB555XL pRGBXlate;
    PALETTEENTRY *pFirstColor;
} _PALETTE,
*PPALETTE;

kd> dd ffff9d55`84484000
fff9d55`84484000 92080be4 ffffffff 00000000 00000000
fff9d55`84484010 67915280 ffff4d8e 00000000 00000000
fff9d55`84484020 0010fd50 00000000 00000000 00000000
fff9d55`84484030 00000000 00000000 00000000 00000000
fff9d55`84484040 00000000 00000000 00000000 00000000
fff9d55`84484050 00000000 00000000 00000000 00000000
fff9d55`84484060 00000000 00000000 00000000 00000000
fff9d55`84484070 00000000 00000000 00000000 00000000
fff9d55`84484080 84484000 ffff9d55 41414141 41414141
fff9d55`84484090 41414141 41414141 41414141 41414141
fff9d55`844840a0 41414141 41414141 41414141 41414141
fff9d55`844840b0 41414141 41414141 41414141 41414141
fff9d55`844840c0 41414141 41414141 41414141 41414141
fff9d55`844840d0 41414141 41414141 41414141 41414141
fff9d55`844840e0 41414141 41414141 41414141 41414141
fff9d55`844840f0 41414141 41414141 41414141 41414141

84484088 ffff9d55
Evolution

- `pFirstColor` property points to our PALETTE (our data)

- `cEntries` property is the PALETTE size

- `pFirstColor/cEntries = Pvcscan0/sizlBitmap`
Evolution

- GetPaletteEntries for reading

- SetPaletteEntries for writing

- iStartIndex parameter offset from pFirstColor
Leaking Palettes
Leaking Palettes

- lpszMenuName is the way to leak them

- Alloc/Free/Alloc works perfect for Palettes >= 0x1000 bytes (LARGE POOL)

- If size < 0x1000 bytes, the same address is never repeated in the next allocation
Leaking Palettes

- The idea is to maximize the predictability for sizes smaller than 4KB

- Addresses can be predicted by “Non repetition” detection

- It consist of adding one step to the alloc/free/alloc way
Leaking Palettes

RegisterClass
heap spray

UnregisterClass (X)
free address X

CreatePalette
LITTLE
heap spray

RegisterClass
LITTLE
heap spray

Is address X re-used?

YES

NO

One PALETTE was ALLOCATED in address X
Demo Time
Demo

- **Target OS:** “Windows 10” x64 RS3

- **Target browser:** Microsoft Edge

- **Objective:** Escape from sandbox
Demo

- Exploitation steps:
  - 1. Inject “fake exploit” in MicrosoftEdgeCP.exe
  - 2. Simulate kernel exploitation
  - 3. Corrupt a Palette object
  - 4. Get a read/write primitive
  - 5. Get SYSTEM privileges by Token Stealer
  - 6. Bypass ACG (next slide)
  - 7. Escape from sandbox
  - 8. Execute “notepad.exe” as SYSTEM
ACG

- Arbitrary Code Guard

- Prevents allocation of executable code in the same process and to other processes

- Designed to mitigate ROP chains with code execution

- `VirtualAlloc/VirtualAllocEx + PAGE_EXECUTE_XXX` is not allowed
ACG

- Current solution to inject code to another process:
  - CreateRemoteThread often combined with a ROP chain to allocate rwx memory in the target process

- The lpParameter argument is used to pass the ROP chain address to be used as STACK by the “stack pivoting”

- Registers rcx, rdx, r8 and r9 have to be set with the VirtualAlloc parameters
ACG

- It requires automating the search for gadgets before the process injection (sandbox escape)

- The gadget finding engine has to be good enough to not fail with multiple libraries versions

- If the target process has ACG enabled or VirtualAlloc is hooked, it will fail
- This mitigation **difficults** the last step of the sandbox escape

- Classic process injection fails

- Getting SYSTEM privileges is not enough to do that!

- See “**Mitigating arbitrary native code execution in Microsoft Edge**” article
Simple ACG bypass
Simple ACG bypass

- Mitigation flags in RS3 located now in EPROCESS structure (offset 0x828)

```
+0x828 MitigationFlags : 0x800539
+0x828 MitigationFlagsValues : <unnamed-tag>
+0x82c MitigationFlags2 : 0
+0x82c MitigationFlags2Values : <unnamed-tag>
```
Simple ACG bypass

- Since we got a kernel r/w primitive, we can modify this flags from the sandboxed process (current process)

- **Bypass:** Disable this one by overwritting EPROCESS.MitigationFlags with the 0x38 value ;-)

- **To think:** it could be used to bypass locally ACG by using kernel exploits written in ROP
A live demo now!
Conclusions
Conclusions

- **Windows 10 RS3** (Fall Creators Update) kernel exploitation is still easy to do

- GDI techniques continue evolving ;-) 

- Sandbox escapes are easy when kernel privilege escalations are used
Conclusions

- Bitmap objects are no longer available in RS3 for kernel exploitation

- Palettes are the new way :)

- Leaking GDI object addresses from user mode still remains a problem ...
Thanks!