black hat ASIA 2017

MARCH 28-31, 2017

MARINA BAY SANDS / SINGAPORE

# Never Let Your Guard Down: Finding Unguarded Gates to Bypass Control Flow Guard with Big Data


| | |
|---|---|
| Ke Sun | wildsator@gmail.com |
| Ya Ou | perfectno2015@gmail.com |
| Yanhui Zhao | wildyz.yky@gmail.com |
| Xiaomin Song | zosnetworking@gmail.com |
| Xiaoning Li | ldpatchguard@gmail.com |

# Agenda

- CFG Implementation Overview
- Previous CFG Bypass Researches
- Research Focus
- Analysis Approaches
- Results & Discussion
- Fix for the issues
- Further Discussion

# Agenda

- **CFG Implementation Overview**
- Previous CFG Bypass Researches
- Research Focus
- Analysis Approaches
- Results & Discussion
- Fix for the issues
- Further Discussion

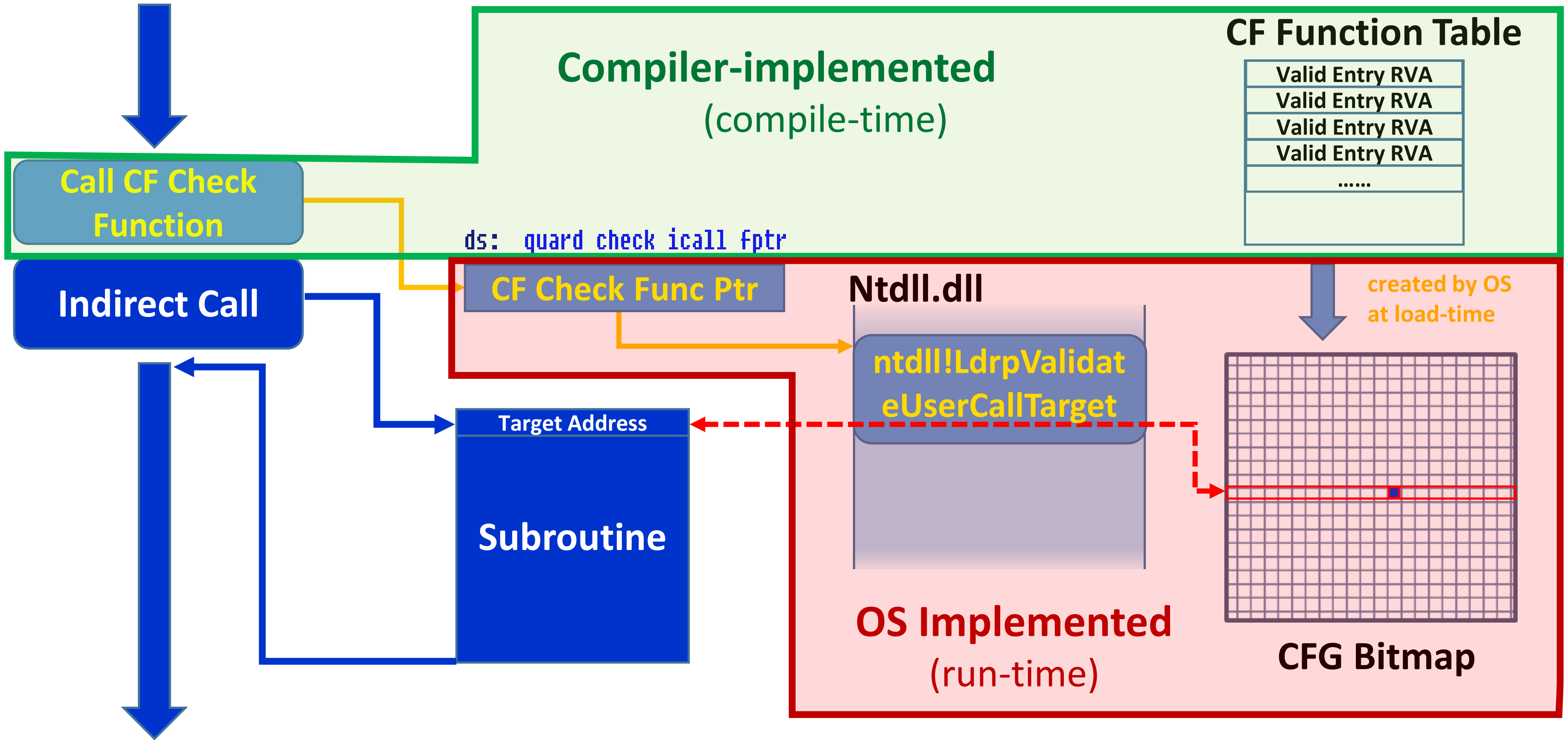
# CFG Overview

- Control Flow Guard (CFG) is a mitigation technology to prevent control flow being redirected to unintended locations, by validating the target address of an indirect branch before it takes place

| Compiler (Compile-time Support) | OS (Run-time Support) |
|---|---|
| Insert CF check function call before each indirect call/jmp | Point the CF check function pointer to ntdll!LdrpValidateUserCallTarget |
| Generate CF function table to list all legal entry addresses (RVAs) | Generate CFGBitmap when process created, based on CF function table |
| Add CFG related entries in Load Configuration Table:<br><br>1. Guard CF Check Function Pointer<br>2. Guard CF Function Table<br>3. Guard CF Function Count<br>4. Guard Flags | Handle violations when CFG check fails (terminate the process by issuing an INT 29h) |

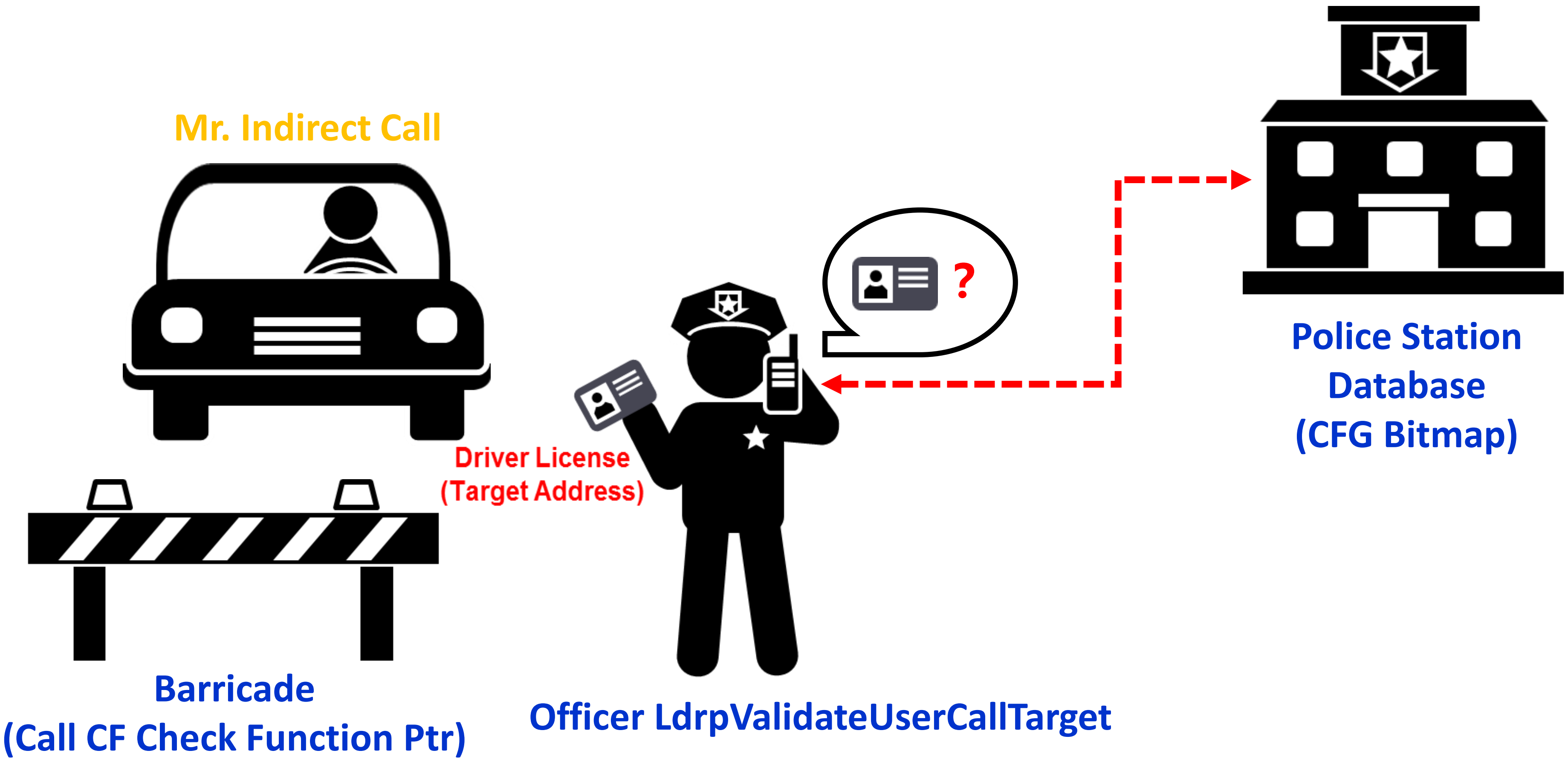
**Original Implementation of CFG**

# CFG Implementation




*In current 64-bit Windows 10 CFG by default uses “dispatch mode” instead of “check & call”*

# CFG - Indirect Call Policing



black hat ASIA 2017

# Agenda

- CFG Implementation Overview
- **Previous CFG Bypass Researches**
- Research Focus
- Analysis Approaches
- Results & Discussion
- Fix for the issue
- Further Discussion

# Previous CFG Bypass Researches

- An incomplete list of previous CFG-bypass studies (most related to JIT)






# Attack Surfaces

- **Non-CFG Module**

  - will eventually sunset with wide implementation of CFG

- **Indirect JMP**

  - already protected by CFG the same way as indirect calls

- **Return Address on Stack**

  - mitigated by newly-introduced Return Flow Guard (RFG)

- **__guard_check_icall_fptr**

  - supposed to be RO but can be made writable in certain cases

  - reported issue fixed by adding wrapper to VirtualProtect

# Attack Surfaces (continued)

- **setjmp/longjmp**

  - jmp_buf can be modified to bypass CFG

  - mitigated by longjmp hardening in Win10 CFG improvement

- **JITed Code**

  - unprotected JITed code or overwrite temp JITed code buffer

  - mostly mitigated by CFG-aware JIT and JIT hardening

- **Valid Gadgets**

  - much less availability and difficult to exploit

# Attack Surfaces – JIT Code

- JIT compliers reported to create problem for CFG
  - Flash ActionScript JIT Compiler
  - Windows Advanced Rasterization Platform (WARP) Shader JIT Compiler
  - JavaScript Chakra JIT Compiler
- CFG-bypass methods:
  - Using unprotected indirect call/jmp from the JITed Code
  - Using JIT Spray: no target address check for indirect call/jmp to the JITed Code
  - Overwriting temporary JITed native code buffer

# Attack Surfaces – JIT Code

➢ Using unprotected indirect call/jmp from the JITed Code

**Exploiting Adobe Flash Player in the era of Control Flow Guard**

• **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
```

Francisco Falcon (@fdfalcon)

**Bypass DEP and CFG using JIT compiler in Chakra engine**

```
0:017> uf 4ff0000
04ff0000 55          push   ebp
04ff0001 8bec        mov    ebp,esp
04ff0003 8b4508      mov    eax,dword ptr [ebp+8]
04ff0006 8b4014      mov    eax,dword ptr [eax+14h]
04ff0009 8b4840      mov    ecx,dword ptr [eax+40h]
04ff000c 8d4508      lea    eax,[ebp+8]
04ff000f 50          push   eax
04ff0010 b840cb5a71  mov    eax, 715acb40h ;
jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>
04ff0015 ffe1        jmp    ecx
```

tombkeeper

This function address can pass the CFG check. Also, before jmp ecx, there is no CFG check of the target address. This can be used as a trampoline for jumping to arbitrary address. We will call it "cfgJumper" hereafter.

**Use Chakra engine again to bypass CFG**

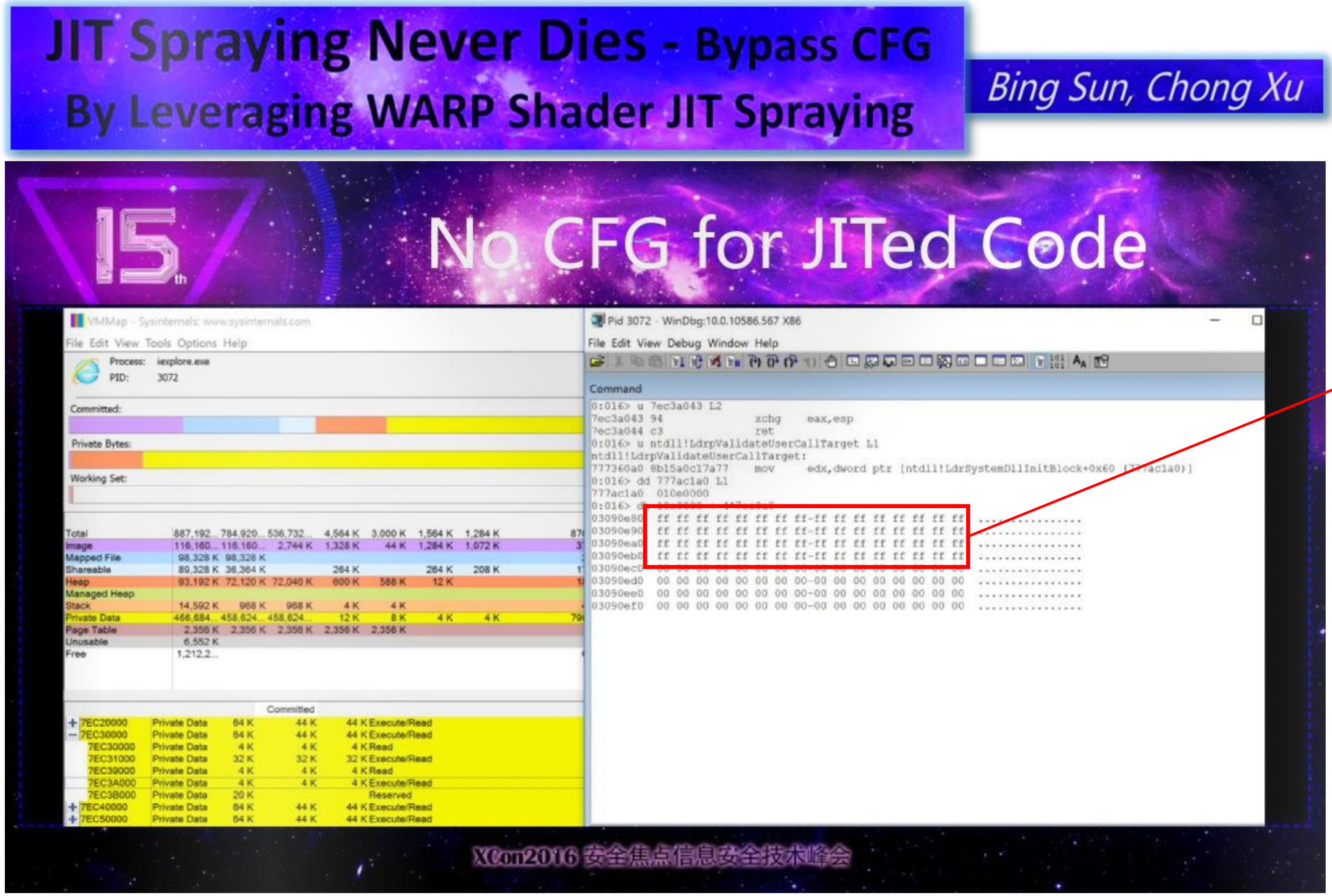
```
.text:002AB3F0 push    ebp
.text:002AB3F1 mov     ebp, esp
.text:002AB3F3 lea     eax, [esp+p_script_function]
.text:002AB3F7 push    eax              ; struct Js::ScriptFunction **
.text:002AB3F8 call    Js::JavascriptFunction::DeferredParse
.text:002AB3FD pop     ebp
.text:002AB3FE jmp     eax
```

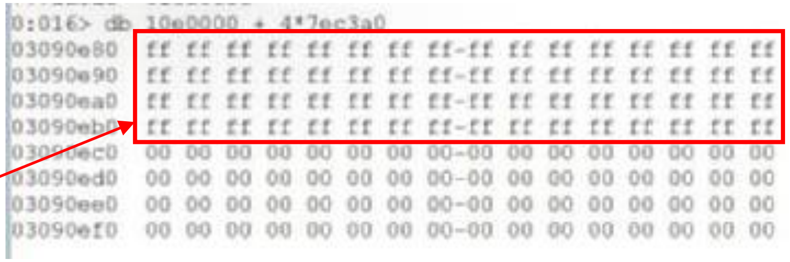
exp-sky

On this jump position, no CFG check is made on the function pointer in eax. Therefore, this can be used to hijack the eip.

# Attack Surfaces – JIT Code

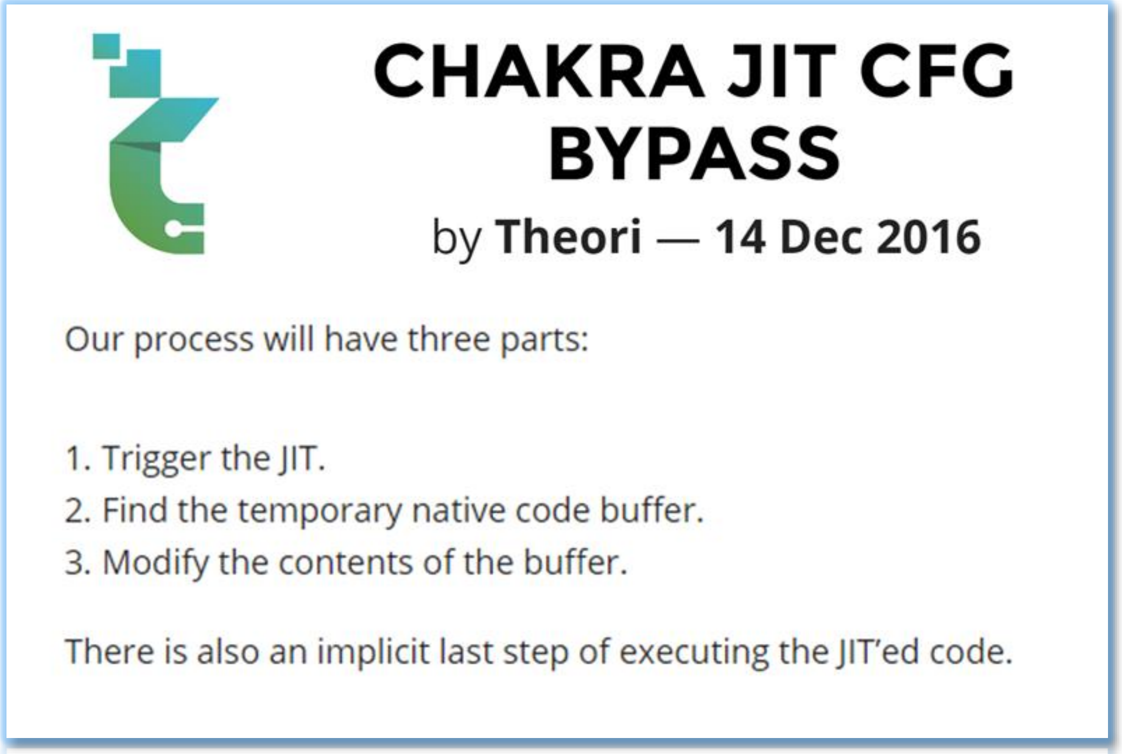
- Using JIT Spray: no target addr check for indirect call/jmp to the JITed Code



# Attack Surfaces – JIT Code

- CFG can also be bypassed by manipulating the JITed code in the temporary code buffer (writable) before it gets copied to the executable memory (non-writable)



CHAKRA JIT CFG BYPASS

by **Theori** — **14 Dec 2016**

Our process will have three parts:

1. Trigger the JIT.
2. Find the temporary native code buffer.
3. Modify the contents of the buffer.

There is also an implicit last step of executing the JIT'ed code.

# Attack Surfaces – Valid Gadget

- CFG only prevents the control flow being hijacked to unexpected locations, but does not stop the unintended use of valid gadgets at legal entry addresses

- However, with CFG, the availability of gadgets is largely reduced, making it much more difficult to exploit

# Agenda

- CFG Implementation Overview
- Previous CFG Bypass Researches
- **Research Focus**
- Analysis Approaches
- Results & Discussion
- Fix for the issues

# Research Focus

- Besides all the previous researches that have been done on CFG bypass, we are trying probing this topic from a different angle

- Instead of trying to break the CFG check logic itself or exploit the implementation issues of CFG in JIT compilers, we are focusing on another aspect that has not been extensively studied for CFG bypass: **memory-based indirect calls**

# Recognition

## Bounty Hunters: The Honor Roll

### Mitigation Bypass

| Name | Company | Amount | Year | Donation to Charity |
|---|---|---|---|---|
| **Thomas Garnier (mxatone@)** | | $5,000 | 2017 | |
| **Yang Junfeng (@bluerust)** | FireEye, Inc. | $15,000 | 2016 | |
| **Yanhui Zhao<br>Ke Sun<br>Ya Ou<br>Xiaomin Song<br>Xiaoning Li** | Intel Labs | $7,500 | 2016 | |
| **Liu Long** | Qihoo360 | $10,000 | 2016 | |
| **Henry Li** | TrendMicro | $18,000 | 2016 | |
| **Bing Sun** | Intel Security Group | $13,000 | 2016 | |
| **Andrew Wesie (awesie)** | Theori | $10,000 | 2016 | |
| **Yu Yang (@tombkeeper)** | Tencent's Xuanwu Lab | $50,000 | 2016 | |
| **Moritz Jodeit** | Blue Frost Security GmbH | $100,000 | 2016 | |
| **Zhang Yunhai (@_f0rgetting_)** | NSFOCUS Security Team | $30,000 | 2016 | |

# CFG Policy for Mem-based Indirect Calls

- Two kinds of memory-based indirect calls:
  - **Function pointer @ vulnerable memory location (CFG-protected)**
    - Example: Calling a function pointer located in .data section, which is RW at runtime
    - Compiler will insert CFG check for the target address
  - **Function pointer @ safe memory location (Non-CFG-protected)**
    - Example: Calling a function pointer from import address table (IAT), which is READ_ONLY after being initialized at runtime
    - Because such memory locations are generally considered “safe” due to their non-writable attribute, CFG check is not implemented

# Mem-based Indirect Calls - Vulnerable Location

- Function pointer @ vulnerable memory location (CFG-protected)

**CFG (/guard:cf) Turned-off**

```
push    0
push    offset aTestmsgwindow ; "TestMsgWindow"
push    offset aTestMessageDis ; "Test message displayed!"
push    0
call    MyFuncPtr
```

**CFG (/guard:cf) Turned-on**

```
push    0
push    offset aTestmsgwindow ; "TestMsgWindow"
push    offset aTestMessageDis ; "Test message displayed!"
push    0
mov     eax, MyFuncPtr
mov     [ebp+var_8], eax
mov     ecx, [ebp+var_8]
call    ds:__guard_check_icall_fptr
call    [ebp+var_8]
```

```
.data:0040711D                 db    0
.data:0040711E                 db    0
.data:0040711F                 db    0
.data:00407120 MyFuncPtr       dd 0
.data:00407120
```

- For memory-based indirect calls with function pointer at vulnerable location, CFG will
  - Insert CF check function before the indirect call
  - Copy the function pointer value to stack and call it from stack instead of from the original memory location

| Name | Start | End | R | W | X |
|---|---|---|---|---|---|
| .text | 00401000 | 00404200 | R | . | X |
| .rdata | 00405000 | 00406600 | R | . | . |
| .data | 00407000 | 00407564 | R | W | . |
| .idata | 00408000 | 00408124 | R | . | . |
| .gfids | 00409000 | 00409200 | R | . | . |
| .00cfg | 0040A000 | 0040A200 | R | . | . |

# Mem-based Indirect Calls - Safe Location

- Function pointer @ safe memory location (Non-CFG-protected)
  - CFG not implemented due to function pointer being READ_ONLY at runtime
  - Form kept as memory-based indirect call: call dword ptr [mem_address]

**CFG (/guard:cf) Turned-on**

**Static**

```
push    ebp
mov     ebp, esp
push    ecx
push    offset LibFileName ; "User32.dll"
call    ds:__imp__LoadLibraryA@4 ; LoadLibraryA(x)

.idata:00408004 ; HMODULE __stdcall LoadLibraryA(LPCSTR lpLibFileName)
.idata:00408004                 extrn __imp__LoadLibraryA@4:dword ; DATA XREF: main+9↑r
.idata:00408004                                                   ; LoadLibraryA(x)↑r
```

| Name | Start | End | R | W | X |
|---|---|---|---|---|---|
| .text | 00401000 | 00404200 | R | . | X |
| .rdata | 00405000 | 00406600 | R | . | . |
| .data | 00407000 | 00407564 | R | W | . |
| .idata | 00408000 | 00408124 | R | . | . |
| .gfids | 00409000 | 00409200 | R | . | . |

**Runtime**

```
CFGTest2!main:
01311370 55              push    ebp
01311371 8bec            mov     ebp,esp
01311373 51              push    ecx
01311374 68585b3101      push    offset CFGTest2!__xt_z+0x130 (01315b58)
01311379 ff1504803101    call    dword ptr [CFGTest2!_imp__LoadLibraryA (01318004)]

Usage:                  Image
Base Address:           00000000`01318000
End Address:            00000000`0131d000
Region Size:            00000000`00005000 (  20.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000002          PAGE_READONLY
Type:                   01000000          MEM_IMAGE
Allocation Base:        00000000`01310000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
```

## Research Focus

➢ Memory-based indirect call (from READ_ONLY locations) is not CFG-protected due to it's considered "safe".

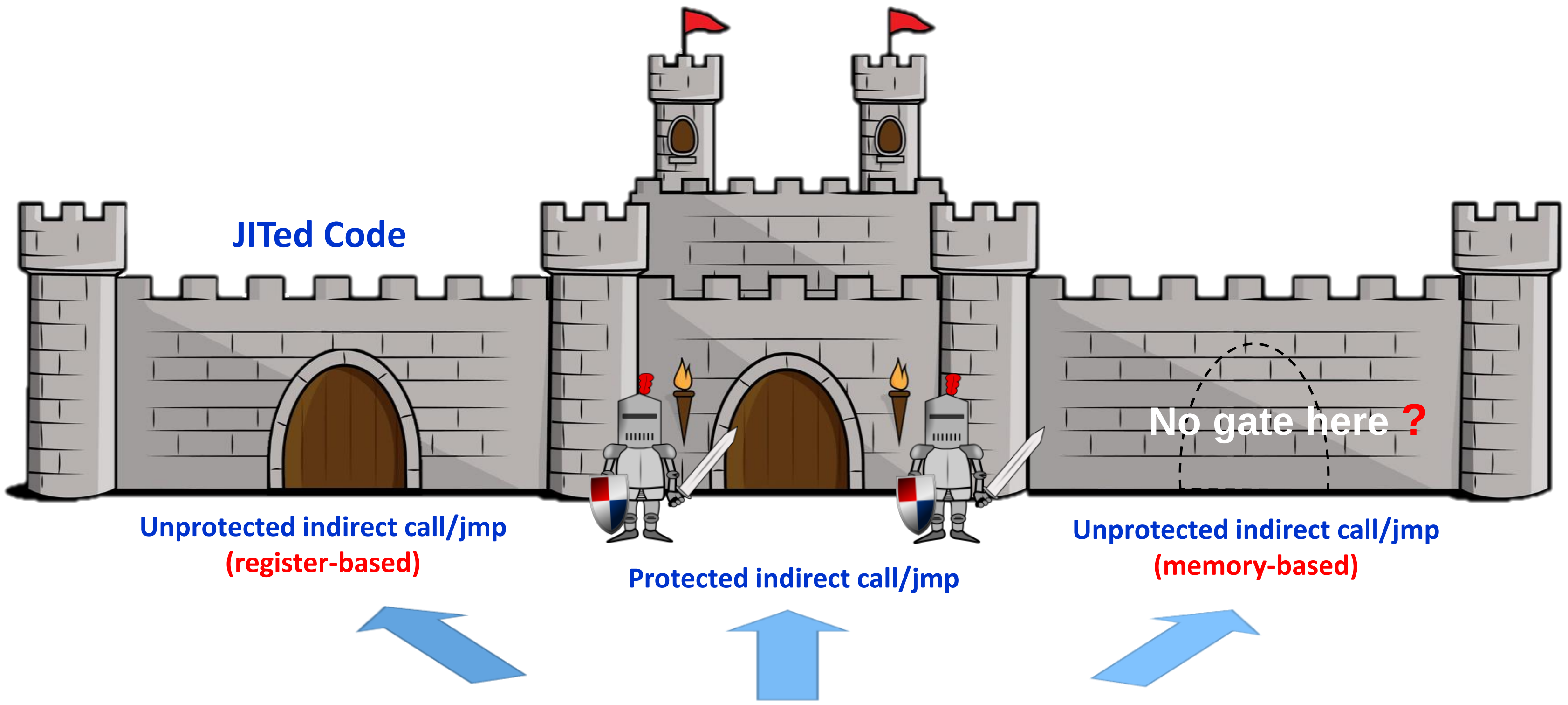
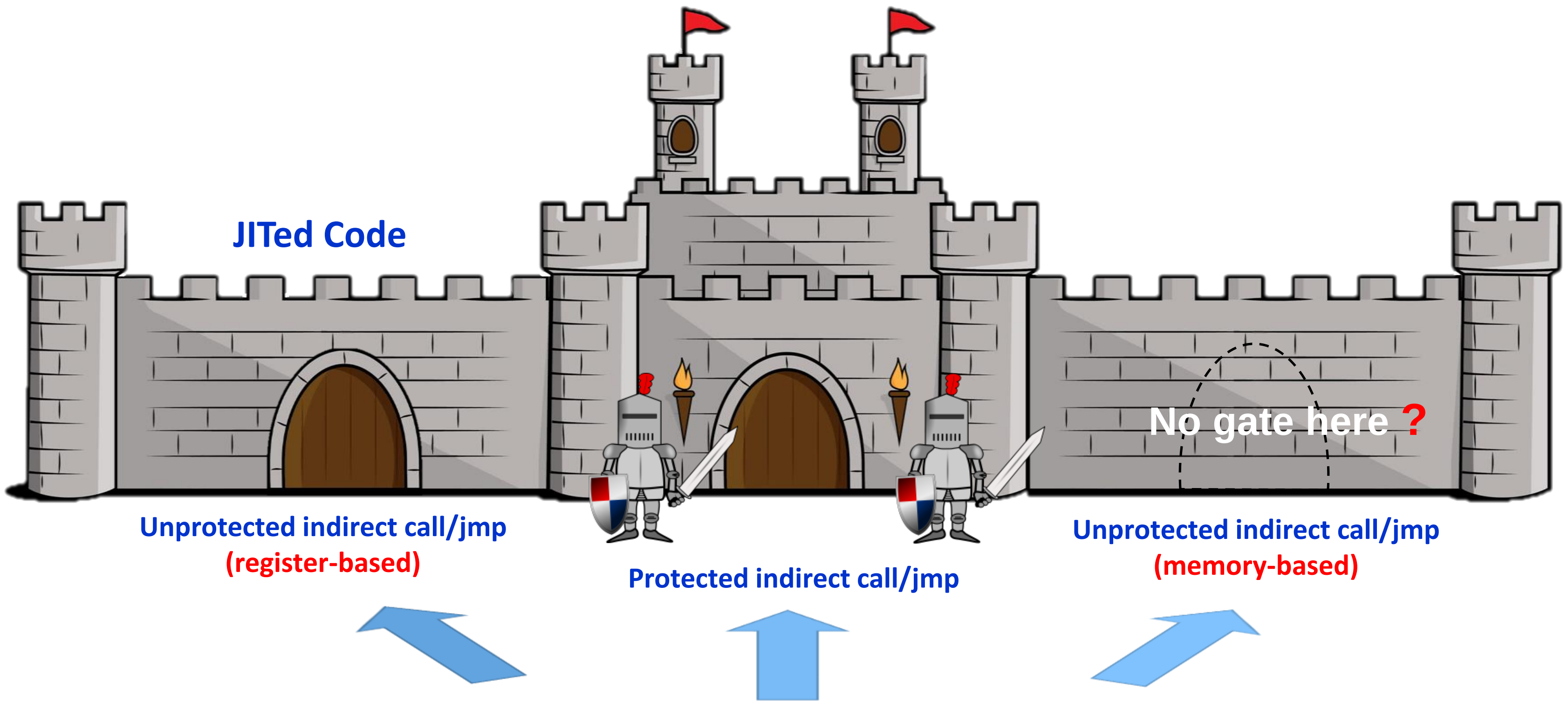



*Image source: http://www.clipartlord.com/category/structures-clip-art/castle-clip-art/, http://clipart-library.com/armor-of-god-clipart.html*

# Research Focus

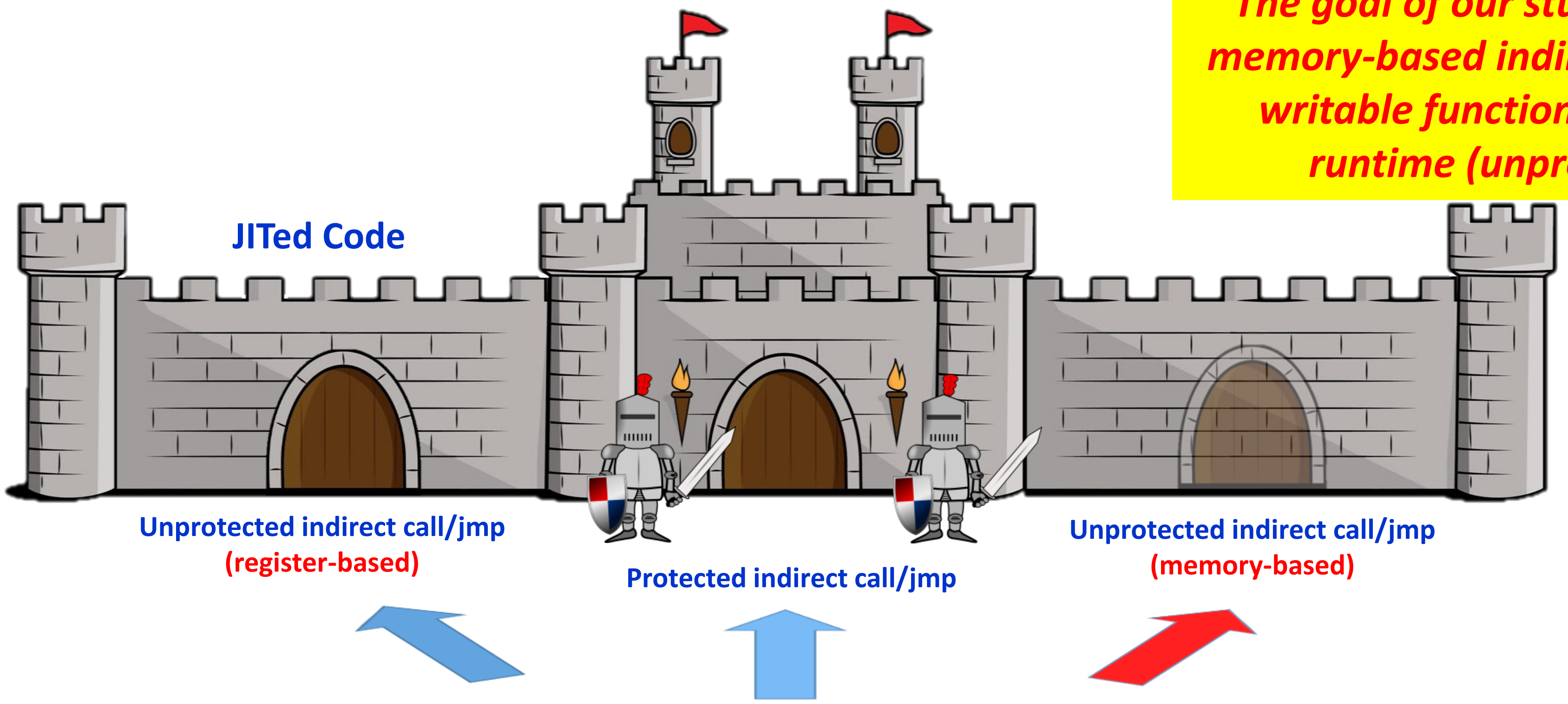
- Memory-based indirect call (from READ_ONLY locations) is not CFG-protected due to it's considered "safe", **is it?**



# Research Focus

- However, if for some reason, the target address pointer of an indirect call become writable, it will become an unguarded gate…

***The goal of our study is to find memory-based indirect calls with writable function pointer at runtime (unprotected)***



# Agenda

- CFG Implementation Overview
- Previous CFG Bypass Researches
- Research Focus
- **Analysis Approaches**
- Results & Discussion
- Fix for the issues
- Further Discussion

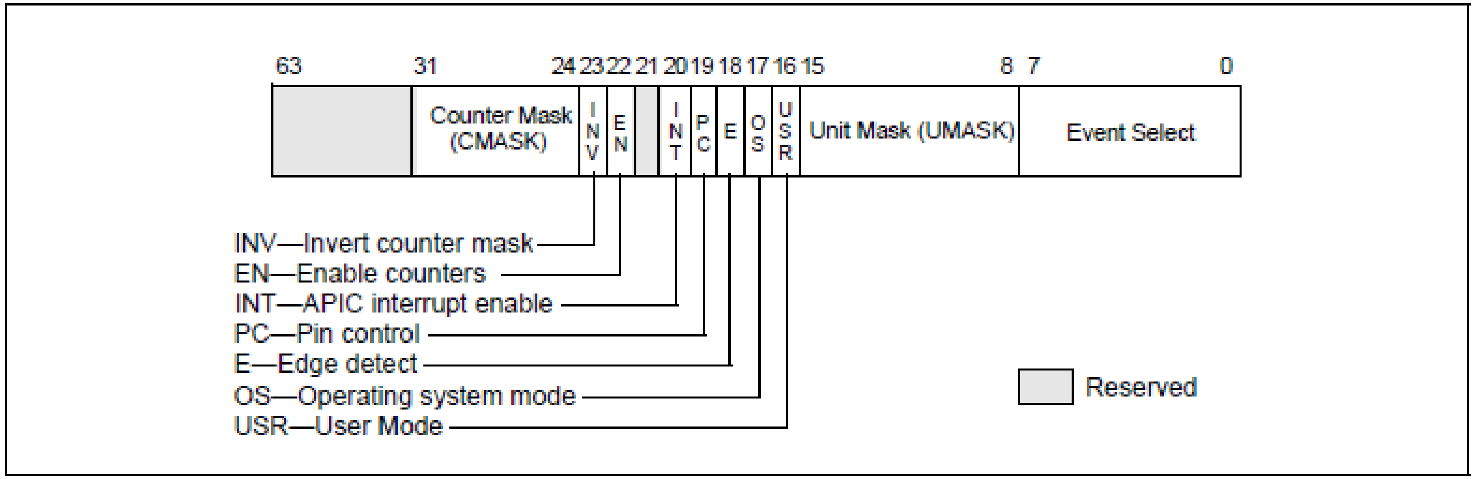
# Analysis Approaches

- To find the cases of indirect call with writable target address pointer, we use an analysis framework with
  - Performance Monitor Unit (PMU)-based instrumentation tool to collect the run-time context information for each indirect branches
  - Spark-based data analysis for large-volume data screening

# Analysis Approaches – Performance Monitoring

- First introduced in the Pentium processor with a set of model specific performance monitoring counter MSRs (Model Specific Registers)

- Permit selection of processor performance parameters to be monitored and measured



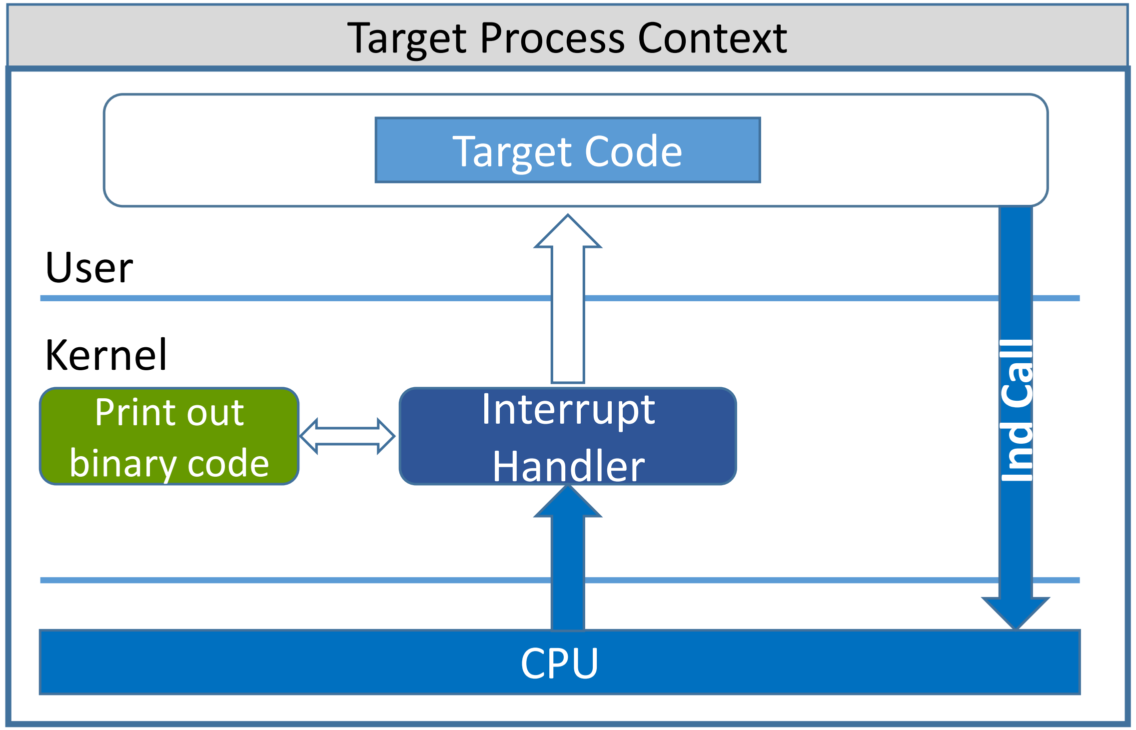

IA32_PERFEVTSELx MSR

# Analysis Approaches – PMU Instrumentation

- To collect binary data after each Ind Call, we utilized PMU to track target code execution
  - Each Ind Call triggers a PMI
  - Register the interrupt handler for PMI
    - 0xFE in IDT
    - Using a Windows API*

  *(Ref: C. Pierce BH USA 2016)*

  - Data collection
    - In Kernel Mode
    - Avoid page fault



# Analysis Approaches – PMU Instrumentation

- CPU performance event select register (Sandy Bridge)

| Event Num. | Umask Value | Event Mask Mnemonic | Description | Comment |
|---|---|---|---|---|
| 88H | 84H | BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET | Taken speculative and retired indirect branches excluding calls and returns. | |
| 88H | 88H | BR_INST_EXEC.TAKEN_INDIRECT_NEAR_RETURN | Taken speculative and retired indirect branches that are returns. | |
| 88H | 90H | BR_INST_EXEC.TAKEN_DIRECT_NEAR_CALL | Taken speculative and retired direct near calls. | |
| 88H | A0H | BR_INST_EXEC.TAKEN_INDIRECT_NEAR_CALL | Taken speculative and retired indirect near calls. | |

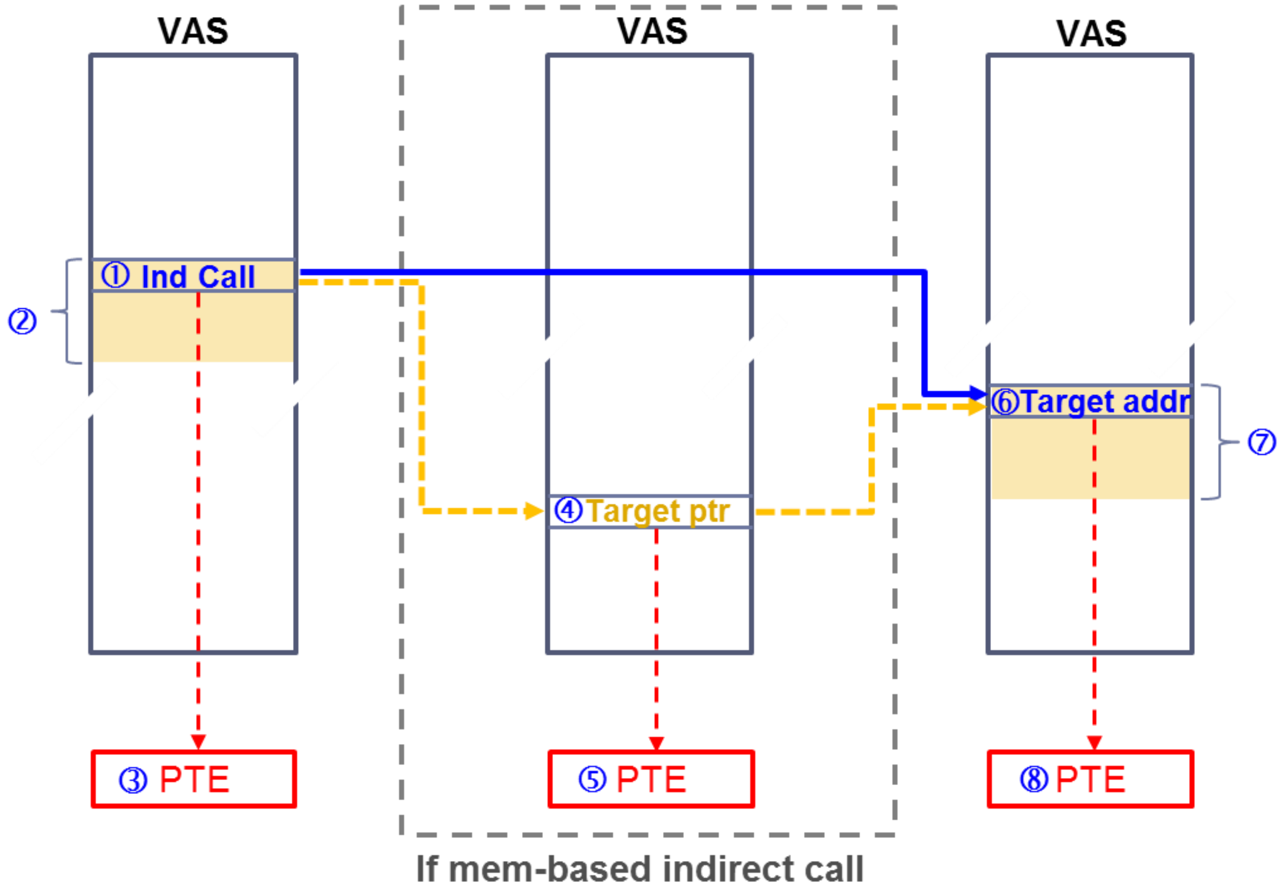
- Performance Monitor Interrupt is triggered at each indirect call instruction while running an application.

- Code stream at each legal entry of indirect call is collected for analysis.

Source: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

# Analysis Approaches – Data Collection

- Context information collected for indirect call




① "from" addr
② "from" code block
③ PTE of "from" addr

④ target ptr addr
⑤ PTE of target ptr addr

⑥ "to" addr
⑦ "to" code block
⑧ PTE of "to" addr

# Analysis Approaches – Data Collection

- Collected data format:
  [+0x00] “from” address
  [+0x08] “from” code block, 8 byte
  [+0x10] “from” address’s PTE
  [+0x18] target pointer’s address
  [+0x1c] target pointer’s PTE
  [+0x20] “to” address
  [+0x28] “to” code block, 8 bytes
  [+0x30] “to” address’s PTE

**Example:**

```
0x0000000072a6bd4b
0xc68372ab01e415ff
0x00000000364f5025
0x571ae02572ab01e4
0x0000000075043cd0
0x08458bec8b55ff8b
0x0000000019da0025
```

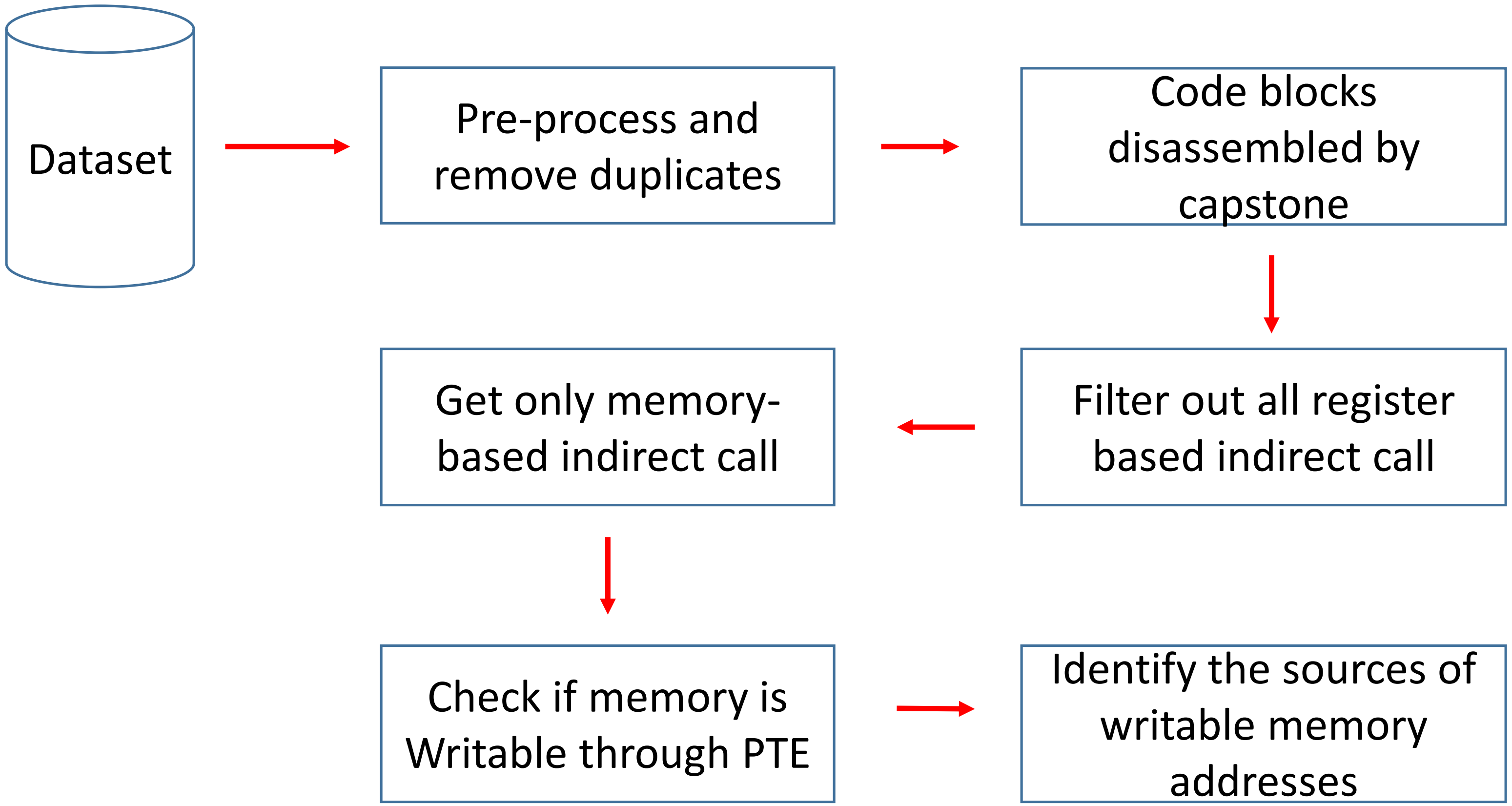
```
72a6bd4b ff15e401ab72    call    dword ptr [uxtheme!_imp__OffsetRect (72ab01e4)]
```

```
0:007> dd 72ab01e4
72ab01e4  75043cd0
```

```
75043cd0 8bff            mov     edi,edi
75043cd2 55              push    ebp
```

# Analysis Approaches - Bigdata Analysis

- Data processing pipeline in Spark



# Agenda

- CFG Implementation Overview
- Previous CFG Bypass Researches
- Research Focus
- Analysis Approaches
- **Results & Discussion**
- Fix for the issues
- Further Discussion

## Results & Discussion

- With the analysis approaches mentioned, we have
  - For Edge, collected data items: 69,341,184, data file size: 4.4G, unique combinations of “to” eip address and code block items: 20,611
  - For flash, collected data items: 9,949,184, data file size: 637M, unique combinations of “to” eip address and code block items: 688
- 3 cases of memory-based indirect calls, which are not protected by CFG per policy, have writable target address pointer:
  - 2 cases with the target address pointers located within the .data section, which is PAGE_READWRITE (windows.storage.dll and ieapfltr.dll)
  - 1 case with the writable target address pointer in the IAT of .idata section of msctf.dll, which is very interesting…

# Results & Discussion

- 1st case of the 2 findings with memory-based indirect call's target address pointers in .data section (RW)

**windows.storage.dll**

```
747d26d9 47              inc     edi
747d26da eb0c            jmp     Windows_Storage!ATL::CSimpleArray<CLoadedItemVectorBase
747d26dc 03ff            add     edi,edi
747d26de 7844            js      Windows_Storage!ATL::CSimpleArray<CLoadedItemVectorBase
747d26e0 81fffffff0f     cmp     edi,0FFFFFFFh
747d26e6 773c            ja      Windows_Storage!ATL::CSimpleArray<CLoadedItemVectorBase
747d26e8 6a08            push    8
747d26ea 57              push    edi
747d26eb ff36            push    dword ptr [esi]
747d26ed ff1500609c74    call    dword ptr [Windows_Storage!_imp___recalloc (749c6000)]
```

```
.data:10506000 ; Segment type: Pure data
.data:10506000 ; Segment permissions: Read/Write
.data:10506000 _data           segment para public 'I
.data:10506000                 assume cs:_data
.data:10506000                 ;org 10506000h
.data:10506000 __imp___recalloc dd offset __recalloc
.data:10506000
.data:10506004                 align 8
```

```
0:013> !address 749c6000

Usage:                  Image
Base Address:           749c6000
End Address:            749ca000
Region Size:            00004000 (  16.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   01000000          MEM_IMAGE
Allocation Base:        744c0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\System32\Windows.Storage.dll
Module Name:            Windows_Storage
```

# Results & Discussion

- 2nd case of the 2 findings with memory-based indirect call's target address pointers in .data section (RW)

**ieapfltr.dll**

```
5c078375 6a04              push    4
5c078377 40                inc     eax
5c078378 50                push    eax
5c078379 ff7604            push    dword ptr [esi+4]
5c07837c ff1500600f5c      call    dword ptr [ieapfltr!_imp___recalloc (5c0f6000)]
5c078382 83c40c            add     esp,0Ch
```

```
.data:72F56000 ; Segment type: Pure data
.data:72F56000 ; Segment permissions: Read/Write
.data:72F56000 _data           segment para public 'DA
.data:72F56000                 assume cs:_data
.data:72F56000                 ;org 72F56000h
.data:72F56000 __imp___recalloc dd offset __recalloc
.data:72F56000
```

```
0:013> !address 5c0f6000

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  Image
Base Address:           5c0f6000
End Address:            5c0f9000
Region Size:            00003000 (  12.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   01000000          MEM_IMAGE
Allocation Base:        5c040000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\SYSTEM32\ieapfltr.dll
Module Name:            ieapfltr
```

# Results & Discussion

- The one case found with indirect call's target address pointer writable and located in the IAT of .idata section

**msctf.dll**

```
74cd9fab 8945e8          mov     dword ptr [ebp-18h],eax
74cd9fae 33c9            xor     ecx,ecx
74cd9fb0 85c0            test    eax,eax
74cd9fb2 7413            je      MSCTF!CtfImeDispatchDefImeMessage+0x1a7 (74cd9fc7)
74cd9fb4 50              push    eax
74cd9fb5 ff15a430da74    call    dword ptr [MSCTF!_imp__ImmLockIMC (74da30a4)]
74cd9fbb 8bd8            mov     ebx,eax
74cd9fbd 85db            test    ebx,ebx
74cd9fbf 0f848fe40300    je      MSCTF!CtfImeDispatchDefImeMessage+0x3e634 (74d1845
74cd9fc5 33c9            xor     ecx,ecx
74cd9fc7 85ff            test    edi,edi

0:024> !address 74da30a4

Usage:                  Image
Base Address:           74da3000
End Address:            74da4000
Region Size:            00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   01000000          MEM_IMAGE
Allocation Base:        74cc0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\System32\MSCTF.dll
Module Name:            MSCTF
Loaded Image Name:      C:\Windows\System32\MSCTF.dll
```

**so we "CATCH THE FLAG"! ☺**

# Results & Discussion

- The reason of this case:

**the whole .idata segment is RW for this dll !!**

| Name | Start | End | R | W | X | D | L | Align | Base | Type | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|
| HEADER | 10000000 | 10001000 | ? | ? | ? | . | L | page | 0005 | public | DATA |
| .text | 10001000 | 100DF000 | R | . | X | . | L | para | 0001 | public | CODE |
| .data | 100DF000 | 100E1418 | R | W | . | . | L | para | 0002 | public | DATA |
| .idata | 100E2000 | 100E26E0 | R | W | . | . | L | para | 0003 | public | DATA |
| .idata | 100E26E0 | 100E5400 | R | W | . | . | L | para | 0003 | public | DATA |
| .idata | 100E6000 | 100E6028 | R | W | . | . | L | para | 0004 | public | DATA |
| .didat | 100E6028 | 100E6200 | R | W | . | . | L | para | 0004 | public | DATA |

# Results & Discussion

- **Bonus finding**: remember the __guard_check_icall_fptr is also in the IAT of .idata section…

```
da238 85c9              test    ecx,ecx
da23a 7423              je      MSCTF!CFunctionProviderBase::QueryInterface+0x6f (74cd
da23c 8b01              mov     eax,dword ptr [ecx]
da23e 51                push    ecx
da23f 8b7004            mov     esi,dword ptr [eax+4]
da242 8bce              mov     ecx,esi
da244 ff15e036da74      call    dword ptr [MSCTF!__guard_check_icall_fptr (74da36e0)]
da24a ffd6              call    esi
da24c 33c0              xor     eax,eax
```

```
0:024> !address 74da36e0

Usage:                  Image
Base Address:           74da3000
End Address:            74da4000
Region Size:            00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   01000000          MEM_IMAGE
Allocation Base:        74cc0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\System32\MSCTF.dll
Module Name:            MSCTF
Loaded Image Name:      C:\Windows\System32\MSCTF.dll
```

***All CFG checks in msctf.dll can be bypassed!!***

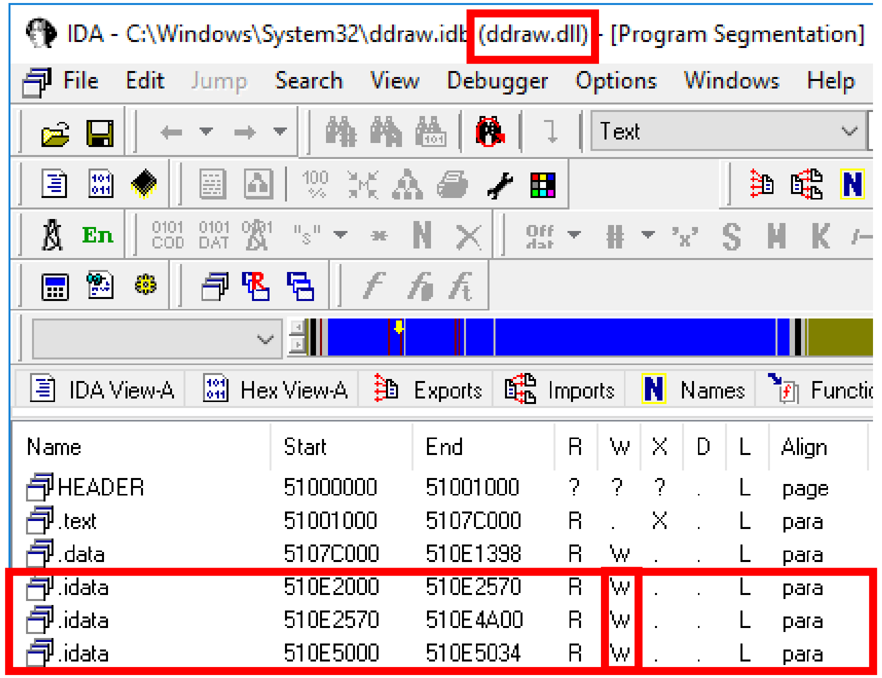
# Results & Discussion – Static Analysis

- Considering it is not likely that msctf.dll is the only black swan, we carried out a more thorough screening using static PE analysis
- Using Python script to screen for any writable .idata section in all windows dlls

```
d3d9.dll in 0x1bf of 0x1db8
.didat
0xc0000040L
d3d9.dll in 0x1c0 of 0x1dba
.didat
0xc0000040L
ddrawex.dll in 0x1c1 of 0x1dc0
.idata
0xc0000040L
ddrawex.dll in 0x1c2 of 0x1dc0
.didat
```

# Results & Discussion – Static Analysis

- 4093 Windows dll files under Windows 10 Home 32-bit system (Version 1607, OS Build 14393.477) have been screened and 4 more dlls with RW .idata sections are found
  - ddraw.dll
  - ddrawex.dll
  - msutb.dll
  - tapi32.dll




- Scan in Windows 10 Pro 64-bit system (Version 1607 OS Build 14393.953) shows the same results

# Agenda

- CFG Implementation Overview
- Previous CFG Bypass Researches
- Research Focus
- Analysis Approaches
- Results & Discussion
- **Fix for the issues**
- Further Discussion

# Fix for the Issues

- Microsoft fixed these issues on March 2017.
- Example: after the fix, In msctf.dll, the CFG function ptr is not Writable anymore.

```
Usage:                  Image
Base Address:           76ba2000
End Address:            76bf5000
Region Size:            00053000 ( 332.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000002          PAGE_READONLY
Type:                   01000000          MEM_IMAGE
Allocation Base:        76ac0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\System32\MSCTF.dll
Module Name:            MSCTF
Loaded Image Name:      C:\Windows\System32\MSCTF.dll
Mapped Image Name:
More info:              lmv m MSCTF
More info:              !lmi MSCTF
More info:              ln 0x76ba26e0
More info:              !dh 0x76ac0000
```

**msctf.dll**

**Before Fix**

| Name | Start | End | R | W | X | D | L |
|---|---|---|---|---|---|---|---|
| HEADER | 10000000 | 10001000 | ? | ? | ? | . | L |
| .text | 10001000 | 100DF000 | R | . | X | . | L |
| .data | 100DF000 | 100E1418 | R | W | . | . | L |
| .idata | 100E2000 | 100E26E0 | R | W | . | . | L |
| .idata | 100E26E0 | 100E5400 | R | W | . | . | L |
| .idata | 100E6000 | 100E6028 | R | W | . | . | L |
| .didat | 100E6028 | 100E6200 | R | W | . | . | L |

**After Fix**

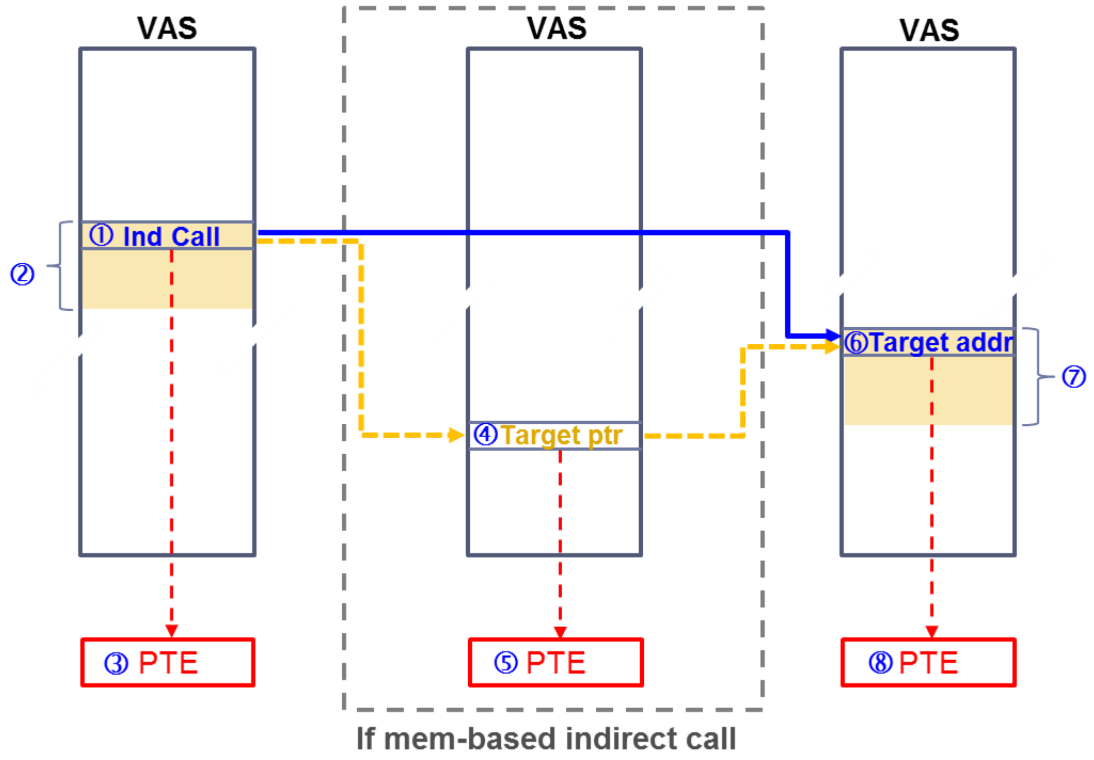
| Name | Start | End | R | W | X | D | L |
|---|---|---|---|---|---|---|---|
| HEADER | 10000000 | 10001000 | ? | ? | ? | . | L |
| .text | 10001000 | 100DE400 | R | . | X | . | L |
| .data | 100DF000 | 100E1418 | R | W | . | . | L |
| .idata | 100E2000 | 100E26E0 | R | . | . | . | L |
| .idata | 100E26E0 | 100E5400 | R | . | . | . | L |
| .idata | 100E6000 | 100E6028 | R | W | . | . | L |
| .didat | 100E6028 | 100E6200 | R | W | . | . | L |

# Agenda

- CFG Implementation Overview
- Previous CFG Bypass Researches
- Research Focus
- Analysis Approaches
- Results & Discussion
- Fix for the issues
- **Further Discussion**

# Further Discussion

- The "**PMU-instrumented data collection + Bigdata analysis**" is a very powerful framework and can be used for different bypass studies by selecting different policies with same data set



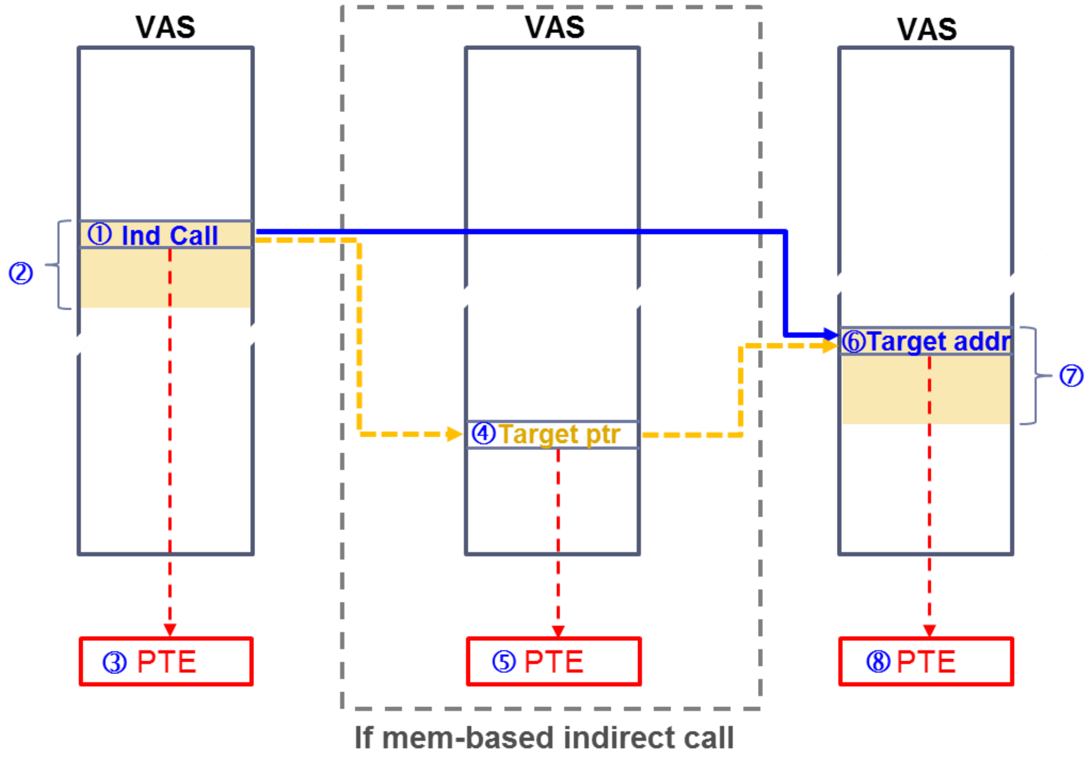

① "from" addr
② "from" code block
③ PTE of "from" addr

④ target ptr addr
⑤ PTE of target ptr addr

⑥ "to" addr
⑦ "to" code block
⑧ PTE of "to" addr

# Policy #1 – Unprotected Mem-based Ind Call

- ②④⑤ can be used to find memory-based indirect calls with writable target pointer for CFG bypass (this work)




① “from” addr

② “from” code block

③ PTE of “from” addr

④ target ptr addr

⑤ PTE of target ptr addr

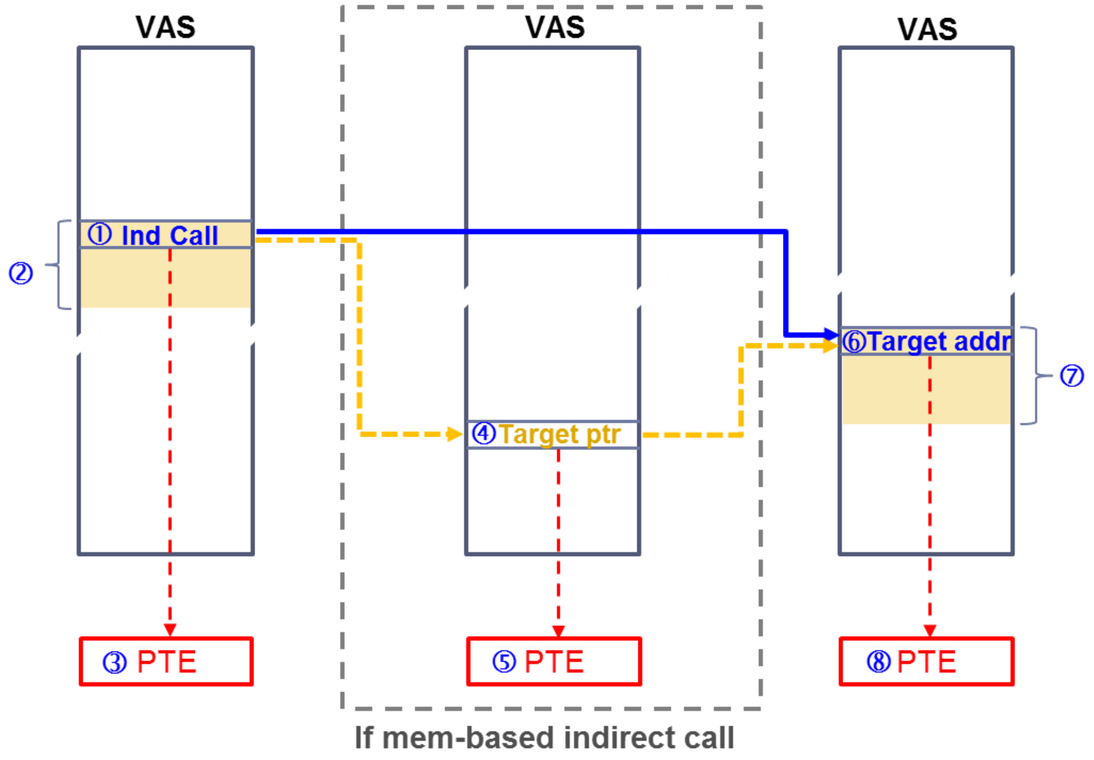
⑥ “to” addr

⑦ “to” code block

⑧ PTE of “to” addr

# Policy #2 – Hunting Valid Gadgets

- ⑦ can be used to find valid gadgets under CFG




① "from" addr

② "from" code block

③ PTE of "from" addr

④ target ptr addr

⑤ PTE of target ptr addr

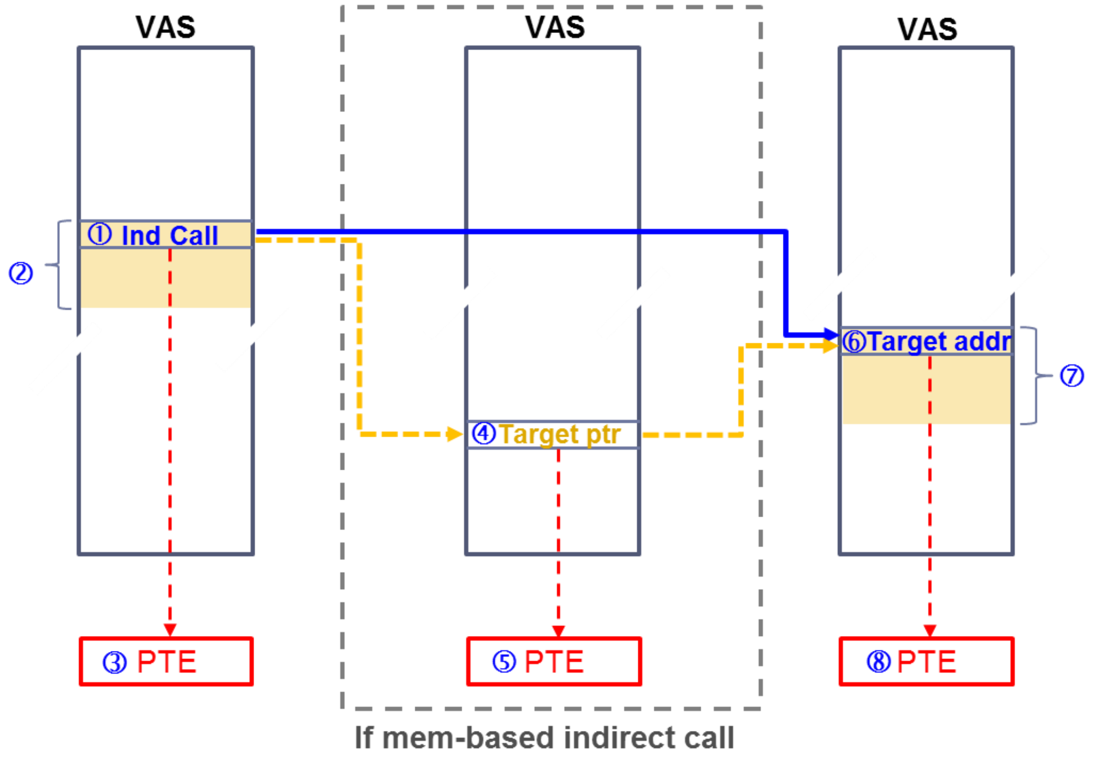
⑥ "to" addr

⑦ "to" code block

⑧ PTE of "to" addr

# Policy #3 – Unprotected Ind JMP

- CFG bypass cases can also be searched by looking for unguarded indirect jmp in ⑦, the “to” code block (work in progress)




① “from” addr

② “from” code block

③ PTE of “from” addr

④ target ptr addr

⑤ PTE of target ptr addr

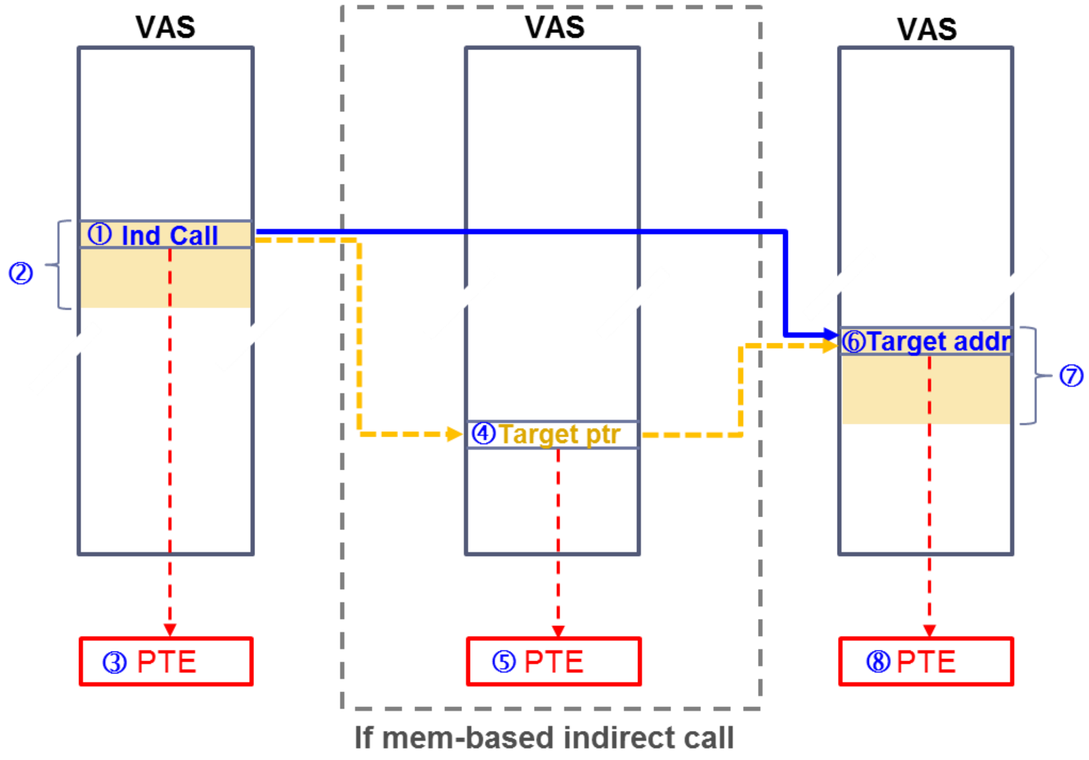
⑥ “to” addr

⑦ “to” code block

⑧ PTE of “to” addr

# Policy #4 – WX Locations in Code Flow

- ➢ ③ and ⑧ can also be used to look for cases with writable “from” or “to” address, which can also be considered CFG bypasses (work in progress)




① “from” addr

② “from” code block

③ PTE of “from” addr

④ target ptr addr

⑤ PTE of target ptr addr

⑥ “to” addr

⑦ “to” code block

⑧ PTE of “to” addr

# Summary

- CFG is a powerful mitigation technique that effectively increases the difficulty and cost for memory-corruption exploitation

- Besides multiple previous studies reporting CFG bypass approaches, this work focuses on finding memory-based indirect calls with writable target address pointer, which can be exploited for CFG bypass

- PMU-based instrumentation and Bigdata analysis are used for data collection and analysis, as well as static PE screening. Multiple results were found and reported to MSRC

- “PMU-instrumented data collection + Bigdata analysis” is a very powerful framework and can be used for different bypass studies by selecting different policies with same data set

# Thank You!

Acknowledgement:

Thanks for *Haifei Li (Intel Security)* and *Rodrigo Branco's (Intel)* review!

# Reference


- Control Flow Guard, https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx
- Exploring Control Flow Guard in Windows 10. Jack Tang. Trend Micro Threat Solution Team, 2015
- Windows 10 Control Flow Guard Internals. MJ0011, POC 2014
- http://blog.trendmicro.com/trendlabs-security-intelligence/control-flow-guard-improvements-windows-10-anniversary-update/
- Bypass Control Flow Guard Comprehensively, Yunhai Zhang, Blackhat 2015
- Exploiting CVE-2015-0311, Part II: Bypassing Control Flow Guard on Windows 8.1 Update 3, Francisco Falcón, Mar 2015
- https://www.blackhat.com/docs/eu-15/materials/eu-15-Falcon-Exploiting-Adobe-Flash-Player-In-The-Era-Of-Control-Flow-Guard.pdf
- https://securingtomorrow.mcafee.com/mcafee-labs/microsofts-june-patch-kills-potential-cfg-bypass/
- https://www.yumpu.com/en/document/view/55963117/jit-spraying-never-dies
- http://xlab.tencent.com/en/2015/12/09/bypass-dep-and-cfg-using-jit-compiler-in-chakra-engine/
- http://xlab.tencent.com/en/2016/01/04/use-chakra-engine-again-to-bypass-cfg/
- http://theori.io/research/chakra-jit-cfg-bypass
- Intel® 64 and IA-32 Architectures Software Developer Manuals, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. Liwei Yuan, Weichao Xing, Haibo Chen, Binyu Zang. APSYS 2011
- Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses, B Lan et al, 2015 IEEE Trustcom/BigDataSE/ISPA
- Capturing 0day Exploit With Perfectly Placed Hardware Traps, C. Pierce, M. Spisak, K. Fitch, Blackhat usa 2016
- IROP – interesting ROP gadgets, Xiaoning Li/Nicholas Carlini, Source Boston 2015
- Apache Spark, http://spark.apache.org/
- Capstone, http://www.capstone-engine.org/