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.

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

Original Implementation of CFG
CFG Implementation

**Compiler-implemented**

(compile-time)

**CF Function Table**

- Valid Entry RVA
- Valid Entry RVA
- Valid Entry RVA
- Valid Entry RVA
- ....

**CFG Bitmap**

created by OS at load-time

**Ntdll.dll**

- ntdll!LdrpValidateUserCallTarget

**OS Implemented**

(run-time)

**Compiler-implemented**

(compile-time)

**Call CF Check Function**

**Indirect Call**

**Target Address**

**Subroutine**

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

Mr. Indirect Call (Call CF Check Function Ptr)  
Barricade (Target Address)  
Officer LdrpValidateUserCallTarget

Police Station Database (CFG Bitmap)
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:

Use Chakra engine again to bypass CFG

Bypass DEP and CFG using JIT compiler in Chakra engine

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.

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 JITed 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>Amount</th>
<th>Year</th>
<th>Donation to Charity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thomas Garnier (@mxatone@)</td>
<td></td>
<td>$5,000</td>
<td>2017</td>
<td></td>
</tr>
<tr>
<td>Yang Junfeng (@bluerust)</td>
<td>FireEye, Inc.</td>
<td>$15,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Yanhui Zhao</td>
<td>Intel Labs</td>
<td>$7,500</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Ke Sun Ya Ou Xiaomin Song Xiaoning Li</td>
<td>Intel Security Group</td>
<td>$13,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Liu Long</td>
<td>Qihoo360</td>
<td>$10,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Henry Li</td>
<td>TrendMicro</td>
<td>$18,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Bing Sun</td>
<td></td>
<td>$13,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Andrew Wesie (@awesie)</td>
<td>Theori</td>
<td>$10,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Yu Yang (@tombkeeper)</td>
<td>Tencent’s Xuanwu Lab</td>
<td>$50,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Moritz Jodeit</td>
<td>Blue Frost Security GmbH</td>
<td>$100,000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Zhang Yunhai (@<em>f0rgetting</em>)</td>
<td>NSFOCUS Security Team</td>
<td>$30,000</td>
<td>2016</td>
<td></td>
</tr>
</tbody>
</table>
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_fp

call [ebp+var_8]
```

- 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Start</th>
<th>End</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>00401000</td>
<td>00401200</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>00403500</td>
<td>00403600</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>00407000</td>
<td>00407564</td>
<td>R</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>idata</td>
<td>00407000</td>
<td>00407612</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glids</td>
<td>00409000</td>
<td>00409520</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jglob</td>
<td>00409000</td>
<td>0040A200</td>
<td>R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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__LoadLibraryA4 ; LoadLibraryA(x)
```

```
cfgTest2\main:
01311370 55
01311371 8bec
01311373 51
01311374 60505b3101
01311379 ff1504803101
```

```
Runtime
```

Usage:
- Memory Address: 00000000 01318000
- End Address: 00000000 0131d000
- Region Size: 00000000 00005000 (20.000 kB)
- State: 00000000 MEM_COMMIT
- Protect: 00000000 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”.

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
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

![Diagram of Target Process Context]

- Data collection
  - In Kernel Mode
  - Avoid page fault
Analysis Approaches – PMU Instrumentation

- CPU performance event select register (Sandy Bridge)

<table>
<thead>
<tr>
<th>Event Num.</th>
<th>Unmask Value</th>
<th>Event Mask Mnemonic</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>88H</td>
<td>84H</td>
<td>BR_INST_EXECTaken INDIRECT_CT_JUMP_NON_CALL_RET</td>
<td>Taken speculative and retired indirect branches excluding calls and returns.</td>
<td></td>
</tr>
<tr>
<td>88H</td>
<td>88H</td>
<td>BR_INST_EXECTaken INDIRECT_CT_NEAR_RETURN</td>
<td>Taken speculative and retired indirect branches that are returns.</td>
<td></td>
</tr>
<tr>
<td>88H</td>
<td>90H</td>
<td>BR_INST_EXECTaken DIRECT_CT_NEAR_CALL</td>
<td>Taken speculative and retired direct near calls.</td>
<td></td>
</tr>
<tr>
<td>88H</td>
<td>A0H</td>
<td>BR_INST_EXECTaken INDIRECT_CT_NEAR_CALL</td>
<td>Taken speculative and retired indirect near calls.</td>
<td></td>
</tr>
</tbody>
</table>

- 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.

Analysis Approaches – Data Collection

- Context information collected for indirect call

![Diagram showing steps for indirect call analysis]

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:

```
0x00000000 72a6bd4b
0xc68372ab0e415ff
0x00000000 864f5025
0x571ae025 72ab01e4
0x00000000 75043cd0
0x08458bec 8b55ff8b
0x00000000 19da0025
```
Analysis Approaches - Bigdata Analysis

- Data processing pipeline in Spark

Dataset → Pre-process and remove duplicates → Code blocks disassembled by capstone

Get only memory-based indirect call

Filter out all register based indirect call

Check if memory is Writable through PTE

Identify the sources of writable memory addresses
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)

```assembly
windows.storage.dll

747d25d8 45 inc edi
747d25e8 54 mov [Windows_Storage\ATL:CSimpleArray<CLoadedItemVectorBase]
747d25e8 03f add edi, edi
747d25e8 7844 js Windows_Storage\ATL:CSimpleArray<CLoadedItemVectorBase
747d25ea 81f1ffff00 cmp edi, 0F1FFFFFh
747d25eb 773c ja Windows_Storage\ATL:CSimpleArray<CLoadedItemVectorBase
747d25ec 6800 push 0
747d25ed 57 push edi
747d25ee ff36 push dword ptr [esi]
```

.code 116005524 call dword ptr [Windows_Storage\imp_realloc (749e6000)]

```
.data:18566800 ; Segment type: Pure data
.data:18566800 ; Segment permissions: Read/Write
.data:18566800 data segment para public '!
s.data:18566800 assume cs: data
.data:18566800 ;org 18566800
.data:18566800 __imp_realloc dd offset __realloc
.data:18566800
.data:18566800 align 0
```

0 013> l addr 749e6000

Usage: Image
Base Address: 749e6000
End Address: 749fa000
Region Size: 0004000 ( 16.000 KB) NEW COMMIT
State: 00010000 PAGE_READWRITE
Protect: 0000004 PAGE_EXECUTE_WRITECOPY
Type: 01000000 MEM_IMAGE
Allocation Base: 744e0000
Allocation Protect: 00000040 PAGE_EXECUTE_WRITECOPY
Image Path: C:\Windows\System32\Windows.Storage.dll
Module Name: Windows_Storage
Results & Discussion

- 2\textsuperscript{nd} case of the 2 findings with memory-based indirect call’s target address pointers in .data section (RW)

ieapfltr.dll

```assembly
push 4
inc eax
push eax
push dword ptr [esi+1]
push dw ['i'] ['e'] ['a'] ['p'] ['f'] ['l'] ['t'] ['r'] ['l']
add esp,0Ch
```

0.013\textsuperscript{rd} instruction: push \texttt{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...

<table>
<thead>
<tr>
<th>Usage:</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Address:</td>
<td>5c0f6000</td>
</tr>
<tr>
<td>End Address:</td>
<td>5c0f9000</td>
</tr>
<tr>
<td>Region Size:</td>
<td>00003000 (12.000 KB)</td>
</tr>
<tr>
<td>State:</td>
<td>HEX_COMMIT</td>
</tr>
</tbody>
</table>
| Protect:          | 00000004 PAGE_EXECUTE 
| Type:             | 01000000 HEX_IMAGE |
| Allocation Base:  | 5c0f4000 |
| Allocation Protect: | 00000080 PAGE_EXECUTE_WRITECOPY |
| Image Path:       | C:\Windows\SYSTEM32\ieapfltr.dll |
| Module Name:      | ieapfltr |
The one case found with indirect call’s target address pointer writable and located in the IAT of .idata section

\texttt{msctf.dll}

\begin{verbatim}
74cd9fab 8945e8 mov    dword ptr [ebp-18h].eax
74cd9fae 33c9 xor    ecx,ecx
74cd9fb0 85c0 test   eax,eax
74cd9fb2 7413 je     MSCTF\CtfIImeDispatchDefImeMessage+0x1a7 (74cd9fc7)
74cd9fb4 50 push   eax
74cd9fb5 ff15a430da74 call   dword ptr [MSCTF\_imp__ImmLockIMC (74da30a4)]
74cd9fb8 8bd8 mov    ebx,eax
74cd9fbd 85db test   ebx,ebx
74cd9fbf 0f848fe40300 je     MSCTF\CtfIImeDispatchDefImeMessage+0x3e634 (74d1845)
74cd9fc5 33c9 xor    ecx,ecx
74cd9fc7 85ff test   edi,edi
0:024> !address 74da30a4
\end{verbatim}

so we “CATCH THE FLAG”!! 😊
The reason of this case:

the whole .idata segment is RW for this dll !!
Results & Discussion

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

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

```plaintext
d3d9.dll in 0x1bf of 0x1db8
.didat
0xc0000040L

0xc0000040L

d3d9.dll in 0x1c0 of 0x1db8
.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.
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.
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)
Policy #2 – Hunting Valid Gadgets

(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)
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)

Diagram:
- ① Ind Call
- ② “from” addr
- 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!
Windows 10 Control Flow Guard Internals. MJ0011, POC 2014
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
http://theori.io/research/chakra-jit-cfg-bypass
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