Combating the Advanced Memory Exploitation Techniques: Detecting ROP with Memory Information Leak

nEINEI, Research Scientist @ McAfee Labs
Chong Xu, Director of IPS Research @ McAfee Labs

CanSecWest2014

March 20, 2014

SAFE NEVER SLEEPS.™

Bio

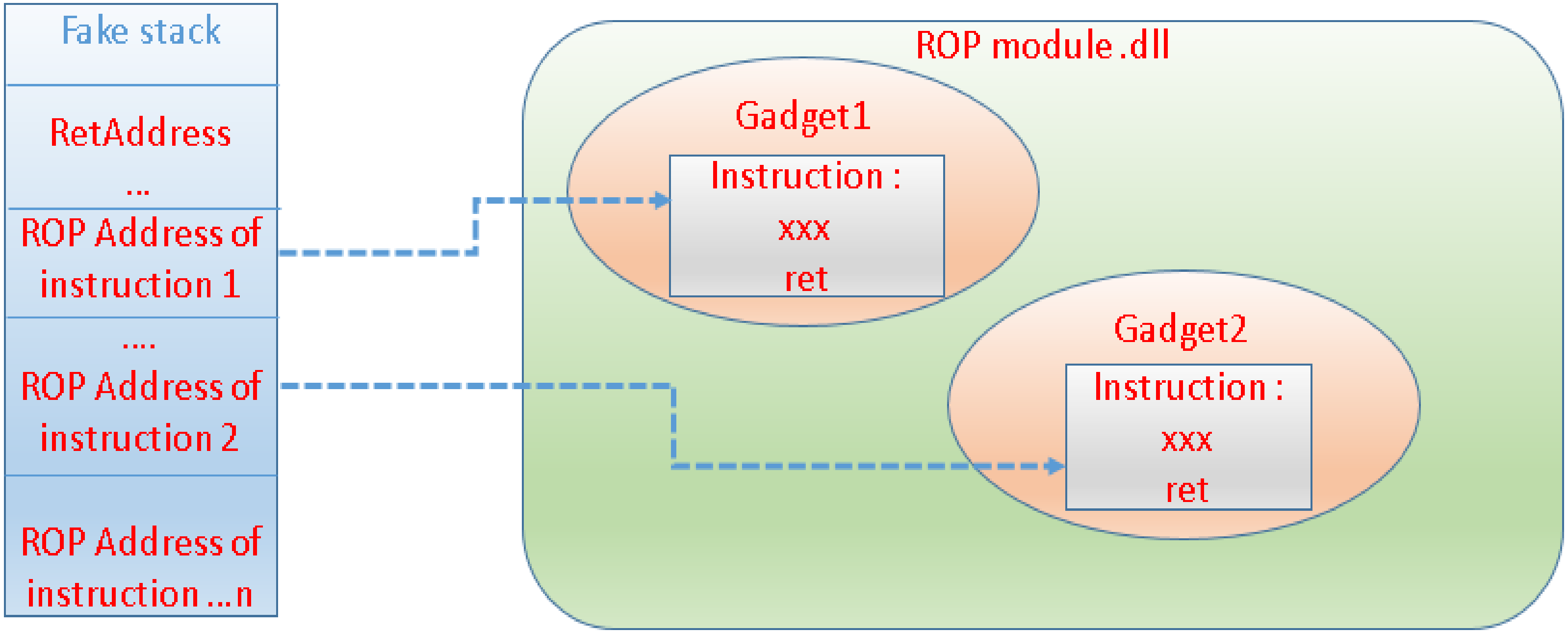
- nEINEI
  - Research Scientist @ McAfee Labs IPS team
  - Focus
    - malware research
    - vulnerability research
    - reverse engineering
- Chong Xu
  - Ph.D. from Duke University
  - Director @ McAfee Labs IPS team
  - Focus
    - advanced (0-day) exploit and malware defense
    - APT detection
    - computer networking
    - network and host security
  - NIPS, HIPS, NGFW, Threat Intelligence

Agenda

- **Background**
- Our Approach
- Case Study
- Optimization
- Summary
- Acknowledgement & Reference

Background What's ROP ?

- ROP(Return-Oriented Programming) : search those instruction sequences (gadgets) that end up with a ret instruction (0xc3) to construct the basic functionalities like memory read/write, logic operation, and flow control.
- The powerful weapon for bypassing DEP: an attacker needs to set executable flag in the memory where the shellcode resides.
- In order to make sure the ROP runs successfully, the 1st ROP gadget needs to switch the current ESP to pointing to some controllable data on the heap (stack pivot)



Background　ROP Exploitation Approaches

- Statically loaded module base information + ROP
  - Load non-ASLR modules, such as Adobe Shockwave (dirapi.dll), MSVCR71.dll,Office(HXDS.DLL) … , these modules are being loaded at some fixed addresses in the process space; therefore it's very easy to be leveraged to constructed the ROP chain.
- Memory information leak + ROP: calculate the ROP module loading base address at runtime
  - Exploiting a vulnerability, modify the array object's length field to increase the array length to achieve an out of bound arbitrary address read/write, leak ntdll.dll address from SharedUserData (CVE-2013-1690)
  - Exploiting a vulnerability, modify the null terminator of a BSTR string to be able to leak the memory information after that (CVE-2013-0640)
  - Exploiting a vulnerability, modify the length of Flash Vector object (by Flash AS) to cross-boundary read out the vtable pointer of some other subsequent object, obtain module base address -> obtain module's import table address-> obtain kernel32 base -> obtain ntdll.dll base (cve-2014-0322).

Anti-ROP Solutions

- Microsoft EMET(Enhanced Mitigation Experience Toolkit)
  - Check points: stack pivot, caller check, simulate execution flow
  - Weakness: API hook based detection, subject to API hook hopping bypass.
- Leverage Intel Pin tool to achieve dynamic instruction instrumentation, dynamically monitoring the instruction sequence execution
  - Check points : The existence of some unique gadgets of ROP. Validity check on ret /call /jump.
  - Weakness: Performance hit in the real application.

Agenda

- Background
- **Our Approach**
- Case Study
- Optimization
- Summary
- Acknowledgement & Reference

Our Approach

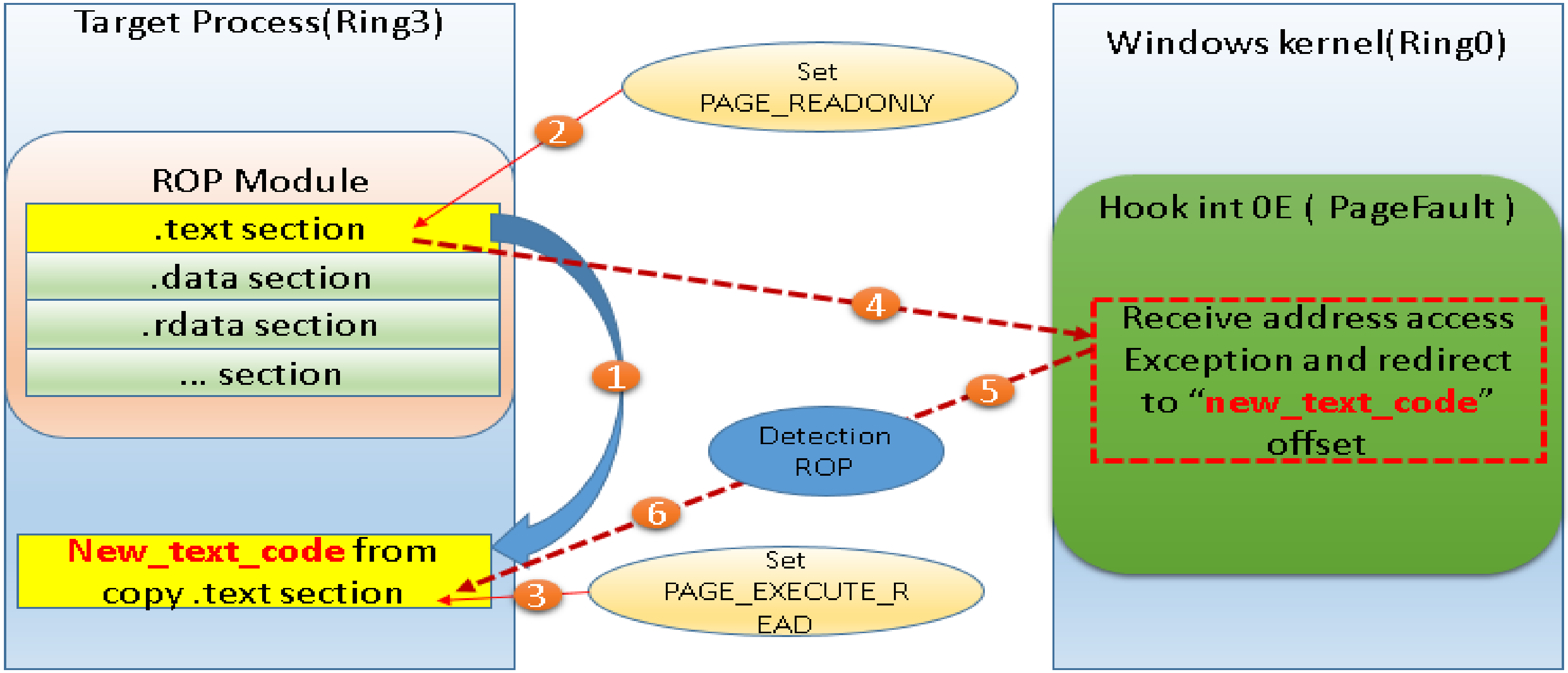
What can we do?

- Observation
  - Usually the valid entry points of a module (the target of control flow transfer, the function address in vftable, jump table, export table, etc) are pre-defined at compilation time, whereas the invalid entry points of code execution (e.g., ROP) are not; and such invalid entries typically hit the middle of a legitimate instruction.
- Our approach
  - Separate valid entries from those invalid entries of execution, and then try to trap the invalid execution.
- What ROP exploitation types can our approach cover?
  - An exploit that leverages non-ASLR modules to launch ROP
  - Memory info leak, dynamically calculate the randomized base address of the ROP module

Our Approach — What we can do?

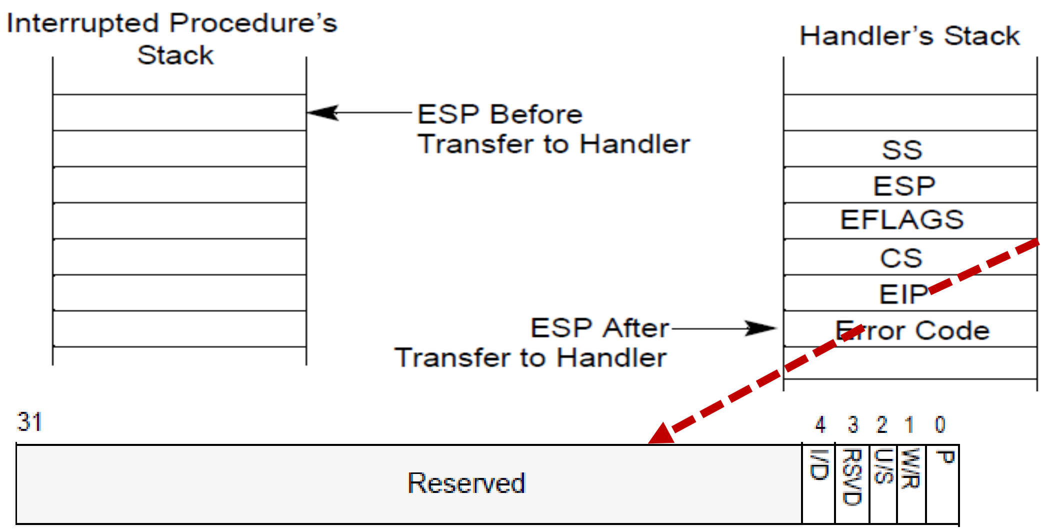
- How our approach works?
  - Copy the .text section (code section) of the ROP module to a new memory region "**new_text_code**".
  - Set the memory attribute of the original .text section of ROP module to NO_EXECUTE (Read Only).
  - Hook the INT 0xe and capture the page fault in kernel mode, and judge whether the fault is generated on the original .text section of ROP module; if so, redirect this faulting code execution access to the same point on the new .text section **new_text_code** for continue executing, in this way the page fault is handled by us transparently without the intervention of OS.
  - Since we can see and analyze each attempt to execute code on the original .text section in our page fault handler, whenever there is a ROP instruction like execution happening, we can catch/block it immediately.
- Our advantages?
  - Not subject to hook hopping bypass
  - Able to locate the 1st ROP gadget instruction, and trace back to the place where the vulnerability is triggered.

Our Approach　How it works?



- Step 5: determine the source of the page fault: from which process, the range of the faulting instruction address, and the error code value.
- Step 6: based on the faulting instruction address, calculate a new address (on the new .text section) for redirection. When the current fault handling is done, the control flow will be returned to the new calculated address, and the normal execution will resume from the new address.

Our Approach How does the page fault redirection work?



Replace the current kernel_stack->eip with the address that the handler can return to ROP detection function.

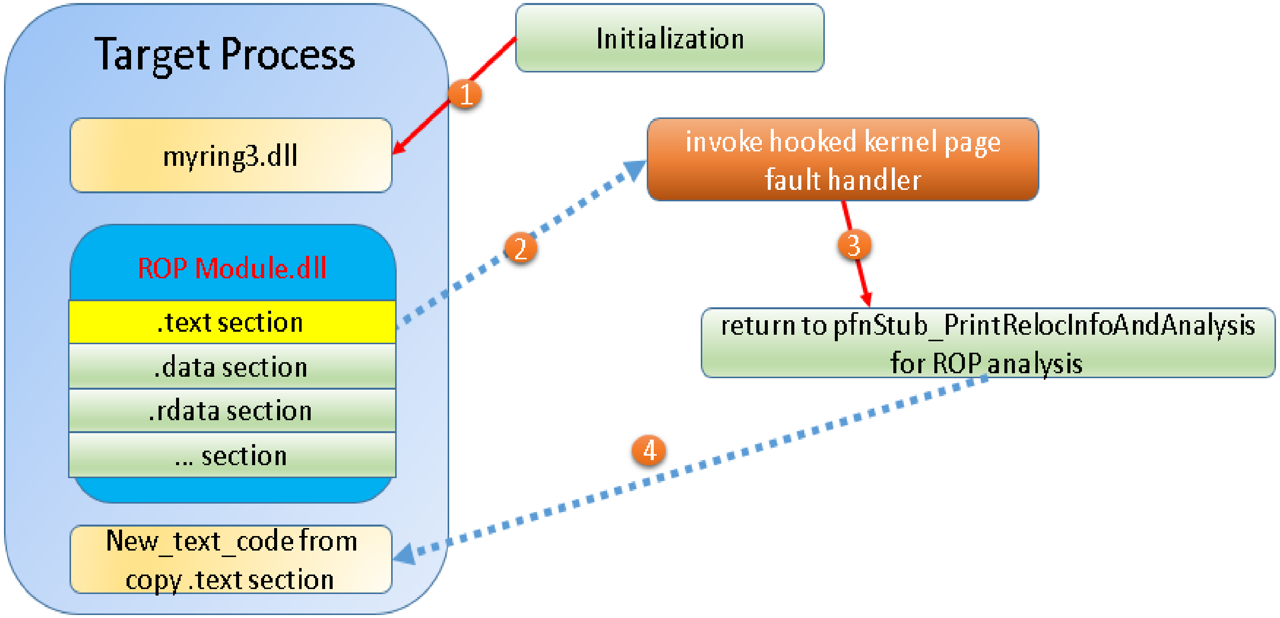
P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	The access causing the fault originated when the processor was executing in supervisor mode.
	1	The access causing the fault originated when the processor was executing in user mode.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by reserved bits set to 1 in a page directory.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.

We are interested in 0x15 = P-bit + I/D-bit + U/S-bit

Our Approach How it works under Ring3?

McAfee An Intel Company

- Initialization
  - Inject our own DLL (i.e., myring3.dll) into the target process
  - Parse the PE structure of the ROP module, and copy the entire .text section to a new allocated memory region "**new_text_code**"; Set the memory attribute of the original .text section of ROP module to PAGE_READONLY to make it NO_EXECUTE
  - Suspend all threads, except for the current thread itself
  - Notify the Ring0 driver to start the address redirection
- ROP detection
  - DLL module does instruction analysis and logs the exception information and analysis results



Our Approach Why the relocated code can still run?

- Executing the instructions within the original .text section
  - The execution of the normal instructions or relative address control transfer within the new memory region ”**new_text_code**” continues in this region, until it hits some control transfer instructions (i.e., jmp/ret/call) that use an absolute address, which leads to an access to the original .text section, thus causes a page fault.
- External calls
  - An external module’s call into the original .text section will cause a page fault and then be redirected to the new memory region to continue execution .
- Already running threads
  - If some threads are already running into the ROP module before the .text section relocation is done, these threads will then be redirected to run on the new region; however some function return addresses that have been pushed in the thread stacks by previous function calls may still point to the original .text section. These old return address may cause some page faults for a few times, but eventually they will be gradually resolved to the new region along with the nested function call return.

Our Approach — Why can the relocated code still run?

- The code access (instruction fetch) faults, i.e., copy-unfriendly instruction/address types
  - Some control transfer instructions **“new_text_code”** using absolute address may go back to the original code region
  - A module passes information (interface pointer, function or data address etc) out to the external modules through some interface call.
  - call/jmp instructions via function address table (containing a list of absolute addresses) within a module, such as virtual function table or jump table.
  - Export function address to the external modules via PE’s export address table (EAT).

Our Approach Why can the relocated code still run?

- "Copy-friendly" instructions
  - normal instructions (mov , xor ,inc ,add…) always run unaffected no matter where you move them to
  - relative control transfer call/jump also run "self-contained" within the new region where they are moved to

relative jump

```
.text:74D3165D 8B 71 08        mov     esi, [ecx+8]
.text:74D31660 6A 01           push    1
.text:74D31662 8B C6           mov     eax, esi
.text:74D31664 E8 66 BC 0D 00  call    ?IsEditable@CElement@@QBEHH@Z ; CElement::IsEditable(int)
.text:74D31669 85 C0           test    eax, eax
```

- "Copy-unfriendly" instructions
  - Control transfer using absolute address

Jump absolute address

```
.text:750CF18D 85 C0             test    eax, eax
.text:750CF18F 74 0F             jz      short loc_750CF1A0
.text:750CF191 8B 4D 10          mov     ecx, [ebp+arg_8]
.text:750CF194 6A 01             push    1
.text:750CF196 51                push    ecx
.text:750CF197 FF 15 18 09 16 75 call    __pDestructExceptionObject
.text:750CF19D 83 C4 08          add     esp, 8
```

Our Approach Why can the relocated code still run?

- “Copy-unfriendly” instructions (cont’d)
  - Module passes interface pointer out to the external modules

```
; HRESULT __stdcall DllGetClassObject(const IID *const rclsid, const IID *const riid, LPVOID *ppv)
                public _DllGetClassObject@12
_DllGetClassObject@12 proc near          ; DATA XREF: .text:off_74DE7B80↓o


.text:74E7FB29                    ; 93:     pfnHookVftable = CreateHook();
.text:74E7FB29 E8 BE 99 0B 00                 call    ?CreateHook@@YGPAVCHook@@XZ ; CreateHook(void)
.text:74E7FB2E                    ; 94:     *ppv = pfnHookVftable;
.text:74E7FB2E 8B 4D 10                       mov     ecx, [ebp+ppv]
.text:74E7FB31 89 01                          mov     [ecx], eax


.text:74F3950C                    ; const CHook::`vftable'
.text:74F3950C 1F 94 F3 74        ??_7CHook@@6B@  dd offset ?VFormat@CHook@@UAEJKPAGHPBGPAX@Z
.text:74F3950C                                                   ; DATA XREF: CreateHook(void)+14↑o
```

Our Approach — Why can the relocated code still run?

- "Copy-unfriendly" instructions (cont'd)
  - call/jmp instructions via function address table (containing a list of absolute addresses) within a module, such as virtual function table or jump table.

```
.text:74C222A4 6C B3 14 75      ??_7CIEDevToolsHost@@6B@ dd offset ?QueryInterface@CIEDevToolsHost@@UAGJABU_GUID@@PAPAX@Z
.text:74C222A4                                          ; DATA XREF: CIEDevToolsHost::~CIEDevToolsHost(void)↓o
.text:74C222A4                                          ; CIEDevToolsHost::CIEDevToolsHost(void)+A↓o
.text:74C222A4                                          ; CIEDevToolsHost::QueryInterface(_GUID const &,void * *)
.text:74C222A8 A1 23 C2 74           dd offset ?AddRef@CUnicodeRanges@@UAGKXZ ; CUnicodeRanges::AddRef(void)
.text:74C222AC 99 B4 14 75           dd offset ?Release@CIEDevToolsHost@@UAGKXZ ; CIEDevToolsHost::Release(void)
.text:74C222B0 9D B3 14 75           dd offset ?GetBrowser@CIEDevToolsHost@@UAGJPAPAUIUnknown@@@Z ; CIEDevToolsHost::Get
.text:74C222B4 09 58 CB 74           dd offset ?FsSetUpdateInfoNewStoryCache@Ptls5@@YGJPAU_fstext@1@PAUstorycache@1@J@Z
```

direct jump

Our Approach Why can the relocated code still run?

McAfee An Intel Company

- “Copy-unfriendly” instructions (cont’d)
  - Export function address to the external modules via PE’s EAT.

```
                    public PlugInMain
PlugInMain          proc near                  ; DATA XREF: .rdata:off_2109D298↓o

arg_0               = dword ptr  8
arg_4               = dword ptr  0Ch
arg_8               = dword ptr  10h
arg_C               = dword ptr  14h

mov     eax, [ebp+arg_8]
mov     dword ptr [eax], offset sub_20802366
xor     eax, eax
inc     eax
```

Ordinal	RVA	Offset	Name
0000	001A69A3	001A5DA3	PlugInMain
0001	0000232E	0000172E	DllRegisterServer
0002	001A697C	001A5D7C	DllUnregisterServer

Information

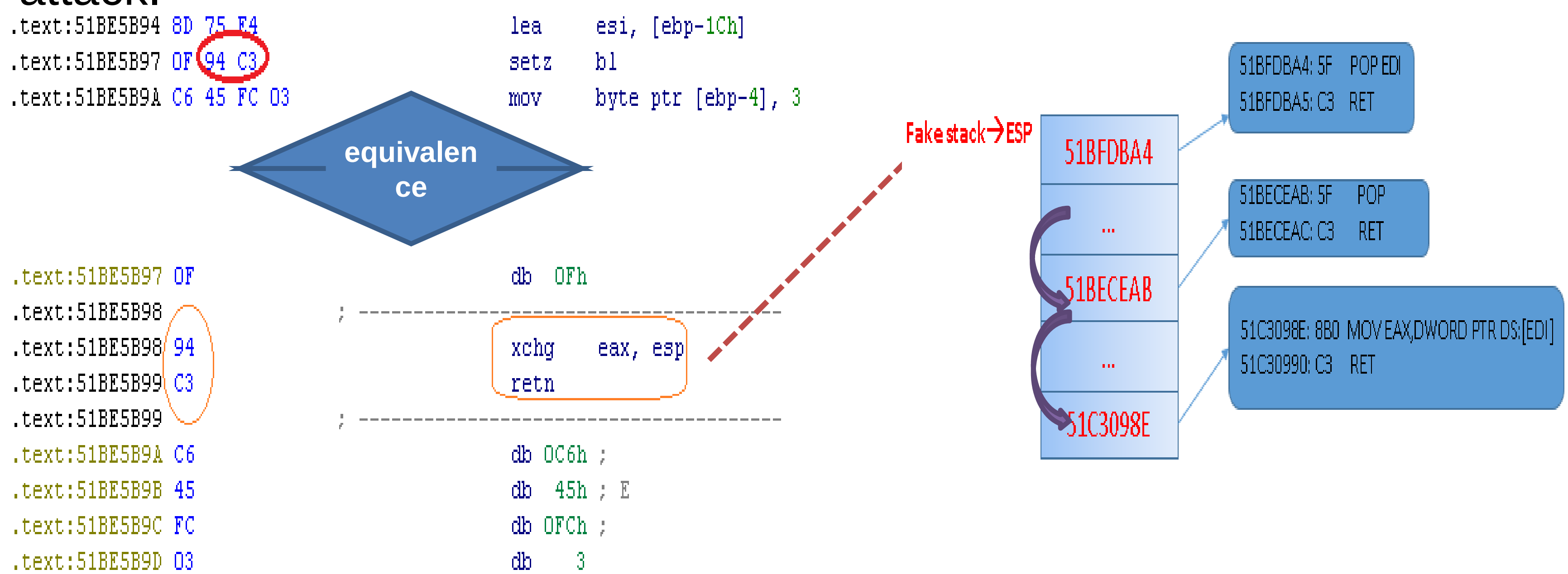
Characteristic:	00000000	NumberOfFunctions:	00000003
TimeDateStamp:	50D0B7D2	NumberOfNames:	00000003
Version:	0.0	AddressOfFunctions:	0089D298
Name:	0089D2B6 ACROFORM.API	AddressOfNames:	0089D2A4
Base:	00000001	AddressOfNameOrdinals:	0089D2B0

Our Approach – Why can the relocated code still run?

- **Summary**
  - By redirecting the faulting code access, any code execution attempt from the original .text section will be transparently forwarded to the execution of the same corresponding code from the new code region (new_text_code)
  - Page faults will be generated during this process, either by the relocated module (the new code region) or from other external modules
  - Our page fault handler is able to catch any code execution attempt on the original .text section checks for validity against the faulting instruction to determine whether this is a valid or ROP like entry

Our Approach – Detecting ROP via a module with a fixed address

- ROP exploit via a ROP module loaded to a fixed address constructs the fake stack using hard-coded sequence of addresses; such addresses point to the gadgets on the original .text section of the ROP module.
- In the following example, address attribute at 0x51BE5B98 is set to non-executable; therefore when the ROP exploit executes the needed instructions, we can catch this faulting instruction and identify the ROP attack.



Our Approach information leak

Detecting ROP with

McAfee An Intel Company

- In the case of Information leak, the base address where the ROP module is loaded to is calculated at runtime, then why is the leaked address still pointing to the original module after we do the redirection?

- For example, CVE-2013-0640
- We can see that a string is allocated first
- 58 58 58 58 00 =“XXXX”,
- by triggering the vulnerability, the null terminator
- after 0x58 is modified to ’0xfe’.

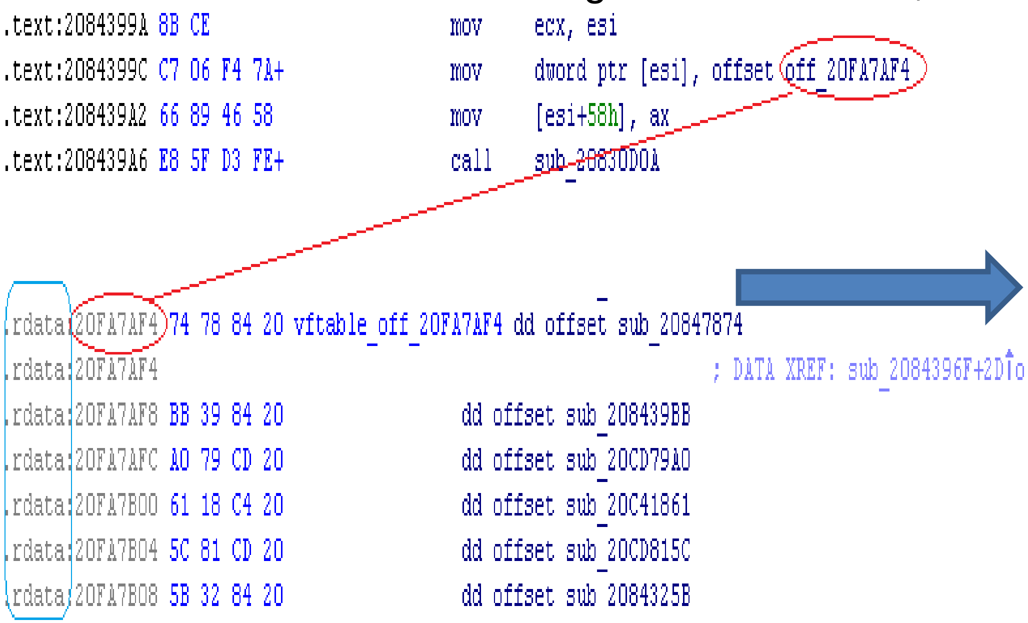
```
0:000> p
eax=002fd700 ebx=070f2610 ecx=046a5a2c edx=16405e0c esi=16405e0c edi=070f26a4
eip=046a548a esp=002fd6ec ebp=002fd720 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200216
AcroForm!PlugInMain+0xa315c:
046a548a 8975e8          mov     dword ptr [ebp-18h],esi ss:0023:002fd708=002fd77c
0:000> dd esi
16405e0c  04da4cac 00000001 00000000 04ece480
16405e1c  000000d4 118642d4 118642d2 00000000
16405e2c  00000000 00000000 16405dbc 118642d0
16405e3c  00000000 00000000 00000000 00000000
16405e4c  118642d4 118642d4 118642d4 16405ea8
16405e5c  118642d4 118642d4 118642d4 118642d4
16405e6c  118642d4 118642d4 118642d4 118642d4
16405e7c  118642d4 118642d4 118642d4 118642d4
0:000> db 118642d4
118642d4  58 58 58 58 00 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  XXXX............
118642e4  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118642f4  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f 00 0f fe 7f  ................
11864304  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
11864314  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
11864324  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
11864334  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
11864344  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................

0:000> g
eax=002fd7cc ebx=00000000 ecx=070f2c6c edx=069c1660 esi=070f2c6c edi=070f2c6c
eip=046a5478 esp=002fd7b0 ebp=002fd7f0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
AcroForm!PlugInMain+0xa314a:
046a5478 e897000000      call    AcroForm!PlugInMain+0xa31e6 (046a5514)
0:000> db 118642d4
118642d4  58 58 58 58 fe 0e fe 7f-f0 0f fe 7f f0 0f fe 7f  XXXX............
118642e4  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
118642f4  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f 00 0f fe 7f  ................
11864304  f0 0f fe 7f f0 0f fe 7f-f0 0f fe 7f f0 0f fe 7f  ................
```

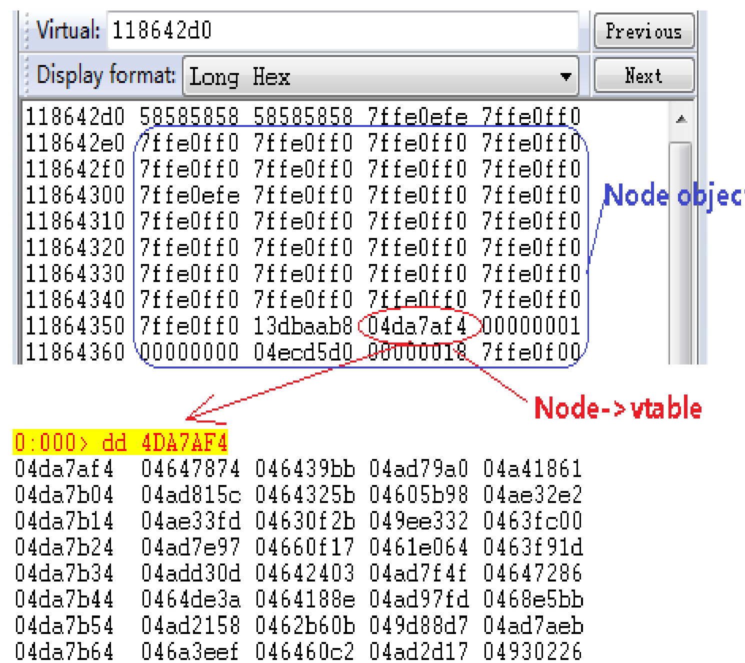
allocate the string array

the string be modified

Our Approach Detecting ROP with information leak

- The attacker deliberately places a Node object after the string array, and using a vulnerability try to out-of-bound read the Node object’s vtable address, i.e., 0x4da7af4
- The offset of vtable of Node object relative to the ROP module base (AcroForm.api) is fixed, i.e., 0x7A7AF4
- Therefore, the randomized (ASLRed) base address of the ROP module = vtable address – offset = 04da7af4 - 7A7AF4 = 0x4600000
- In this example, Node object’s vtable is in the .rdata section. We relocate the only the .text section, whereas ROP exploit calculates the randomized base address of the ROP module via the leaked vtable address in .rdata section. Since the calculated base address of the ROP module is still in the original .text section, we can catch the ROP attack.



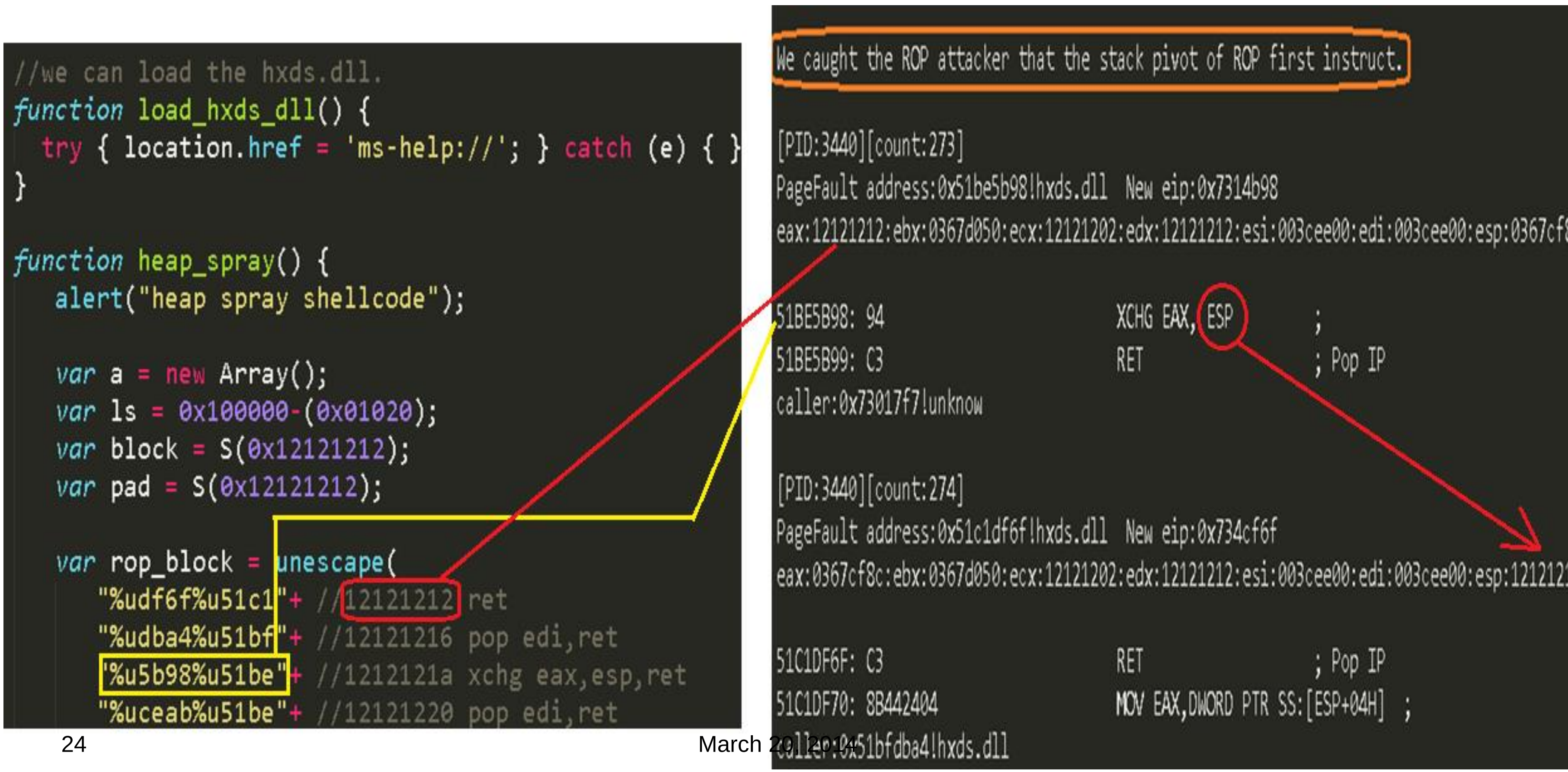


Agenda

- Background
- Our Approach
- **Case Study**
- Optimization
- Summary
- Acknowledgement & Reference

Case Study CVE-2013-3893 ROP via a module with a fixed address

- CVE-2013-3893 is an IE vulnerability, the attacker leverages a non-ASLR module hxds.dll in MS Office product to do ROP.
  - load a non-ASLR hxds.dll into IE; fixed address @ 0x51be5b98; controllable fake stack @ 0x12121212
- Demo.

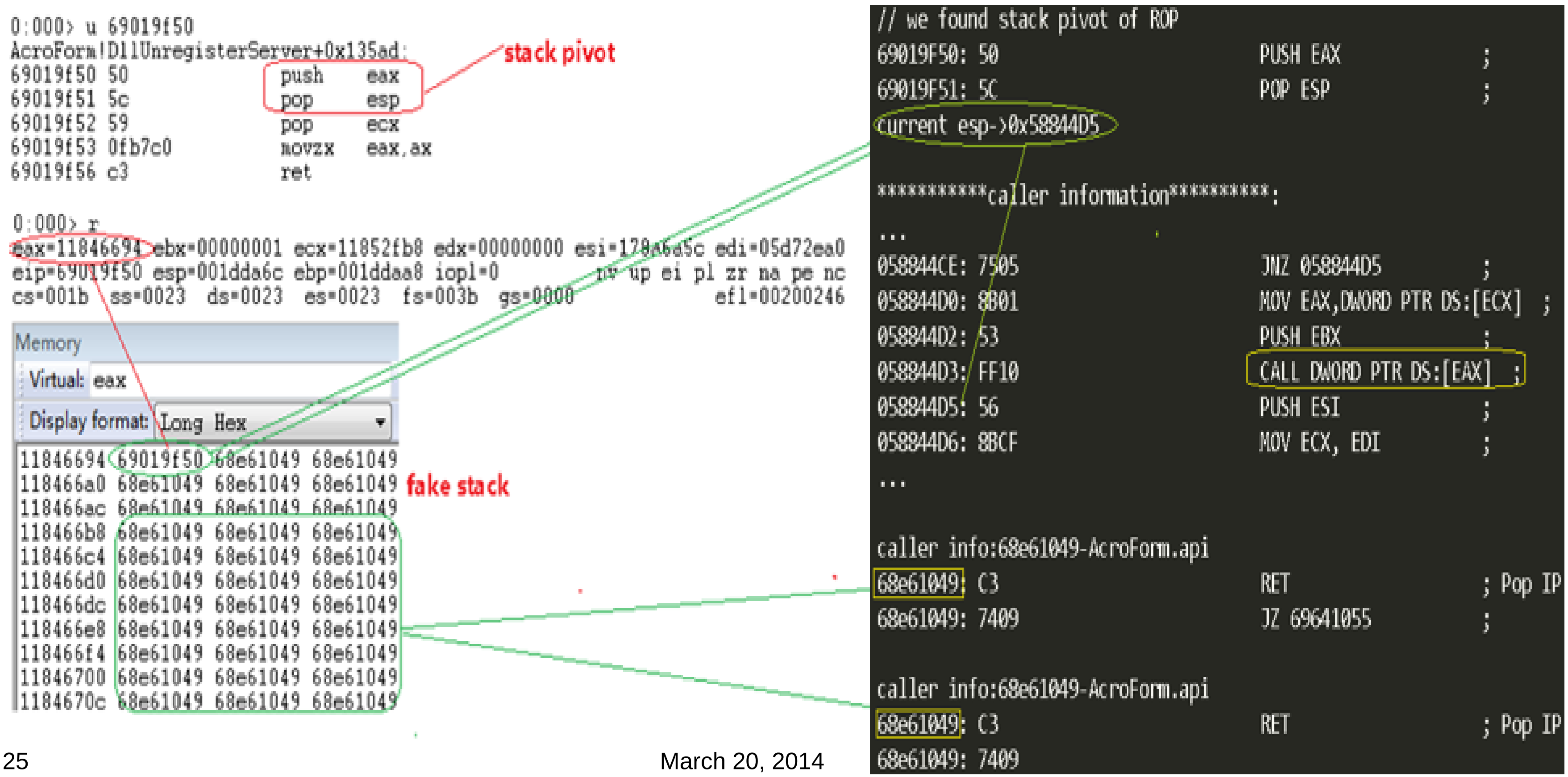


Case Study Info leak

CVE-2013-0640 ROP with

McAfee
An Intel Company

- CVE-2013-0640,Adobe Acrobat And Reader CVE-2013-0640 Remote Code Execution Vulnerability.
  - The module AcroForm.api is the target of info leak and the subsequent ROP chain construction. We can catch stack pivot from the original .text section.
- Demo.



Agenda

- Background
- Our Approach
- Case Study
- **Optimization**
- Summary
- Acknowledgement & Reference

Optimization Why we need to optimize?

- Challenge
  - Without optimization, both some legitimate entries and ROP execution attempts may cause page faults; excessive number of page faults not only elongates the exploit execution, thus may cause the exploits to fail, but also slows down the system and make the application unusable.
- Goal
  - From the ROP detection's perspective, we are only interested in those page faults that are generated by ROP
- Observation
  - The majority of the page faults are caused by control transfer to the old code section using absolute addresses and many of those originate from the function address table based call/jmp within the ROP module

Optimization　　Main factors for performance drop?

- Results
  - Taking CVE-2013-0640 as an example. The ROP module is AcroForm.api. Our internal testing showed that without any optimization we might need to experience ~15 million of page faults before the ROP instruction is identified. After fixing up the PE's relocation section, only thousands of page faults were seen, where almost all of those page faults were introduced by ROP exploits.

Agenda

- Background
- Our Approach
- Case Study
- Optimization
- **Summary**
- Acknowledgement & Reference

Summary

- Make a shadow copy of the .text code section of the ROP module and mark the original .text code section NON_EXECUTE
- Change all the necessary addresses so that the shadow copy transparently runs in lieu of the original .text code section
- Any ROP attempt into the original .text code section will cause page faults, thus will be caught by our exception handler with (ROP instruction, stack info, register info, current thread info, etc) detail

Acknowledgment & Reference ☺

- Acknowledgement
  - Thanks Bing Sun for providing the research direction and help on the implementation; we would also like to thank the colleagues of McAfee Labs IPS Team for sharing many great technical information and ideas.
  - Thank CanSecWest2014 for offering an opportunity to share our research with the whole security community.
- References
  - YAN Tao, WANG Yi-jun, XUE Zhi, “Research and Application of ROP Automatic Generation Technology on Windows”
  - Elias Bachaalany , “Inside EMET4.0”
  - Haifei Li, “Smashing the Heap with Vector: Advanced Exploitation Technique in Recent Flash Zero-day Attack”
  - Lucas Daviy, Ahmad-Reza Sadeghiy, Marcel Winandyz, “ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks”