Defending Against Return-Oriented Programming

Michalis Polychronakis

joint work with Vasilis Pappas and Angelos Keromytis

Columbia University

DIMVA ’12 – July 27, 2012
Outline

From shellcode to ROP

ROP payload detection

In-place code randomization
Code Injection

```
buffer[]
saved
EIP
????

 Padding

... 4000B12A jmp eax ...

padding 4000B12A \x6A\x07\x59\xE8\xFF\x0F\xC1

4000B12A

eax

Code Injection ...

... 4000B12A jmp eax ...

some.dll

eax

esp
```
NX

W^X, PaX, Exec Shield, DEP

x86 support introduced by AMD, followed by Intel Pentium 4 (late models)

DEP introduced in XP SP2 (hardware-only)
    Applications can opt-in (SetProcessDEPPolicy() or /NXCOMPAT)
Ret2libc $\rightarrow$ ROP

ret2libc  [Solar Designer ‘97]

```
padding  &execve  fake ret  /bin/sh
```

ret2libc chaining  [Nergal ‘01]

```
&func1  pop; pop; ret  arg1  arg2  &func2  fake ret  arg1
```

```text
esp
```
Ret2libc $\rightarrow$ ROP

Borrowed code chunks technique [Krahmer ’05]

Pass function arguments through registers (IA-64)

0x000000000000400a82: pop %rbx
0x000000000000400a83: retq
0x000002aaaaac743d5: mov %rbx,%rax $\rightarrow$ &system
0x000002aaaaac743d8: add $0xe0,%rsp
0x000002aaaaac743df: pop %rbx
0x000002aaaaac743e0: retq
0x000002aaaaac50bf4: mov %rsp,%rdi $\rightarrow$ /bin/sh
0x000002aaaaac50bf7: callq *%eax

Return-oriented programming [Shacham ’07]

Turing-complete return-oriented “shellcode”

Jump-oriented programming [Shacham ’10]
<table>
<thead>
<tr>
<th>Payload</th>
<th>Code</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb8800000</td>
<td>0xb8800000:</td>
<td>eax = 1</td>
</tr>
<tr>
<td>0x00000001</td>
<td>pop eax</td>
<td></td>
</tr>
<tr>
<td>0xb8800010</td>
<td>ret</td>
<td>ebx = 2</td>
</tr>
<tr>
<td>0x00000002</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xb8800020</td>
<td>0xb8800010:</td>
<td>eax += ebx</td>
</tr>
<tr>
<td>0xb8800010</td>
<td>pop ebx</td>
<td></td>
</tr>
<tr>
<td>0x00400000</td>
<td>ret</td>
<td>ebx = 0x400000</td>
</tr>
<tr>
<td>0xb8800030</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xb88000030</td>
<td>0xb8800020:</td>
<td>*ebx = eax</td>
</tr>
<tr>
<td>...</td>
<td>add eax, ebx</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xb8800030:</td>
<td>mov [ebx], eax</td>
</tr>
<tr>
<td></td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Current State of ROP exploits

First-stage ROP code for bypassing DEP
  Allocate/set W+X memory (\texttt{VirtualAlloc, VirtualProtect, ...})
  Copy embedded shellcode into the newly allocated area
  Execute!

The complexity of ROP exploit code increases…
  New anti-ROP features in EMET
  ROP exploit mitigations in Windows 8

The embedded shellcode can be concealed
  ROP-based unpacker [Lu ’11]
How does ROP change attack detection?
Shellcode Detection Not Enough

```assembly
push byte 0x7f
pop ecx
call 0x7
inc ecx
pop esi
add [esi+ea],0xe0
xor [esi+ecx+0xb],cl
loop 0xe
xor [esi+ecx+0xb],cl
loop 0xe
xor [esi+ecx+0xb],cl
...  
```
ROPscan

Detect ROP payloads in arbitrary inputs

Speculative execution
  Consider each 4-byte value as the beginning of a ROP payload

Code emulation
  Initialize VM with real (non-ASLR) process image(s)
  Speculatively execute gadgets based on valid memory addresses found in the scanned input

Identify the execution of multiple unique gadgets
Runtime Heuristic

Benign inputs may trigger the execution of valid instruction sequences

Gadgets are peculiar:

*End with an indirect control transfer instruction that uses control data derived from the input buffer*

```
... 00010104 070015BB 00001000 ...
esp → 070015BB
00001000
... 00010104

read access

pop eax;eax=esp070015BB
jmp eax ;EIP = 070015BB
...```
Runtime Heuristic

An executed instruction sequence is a gadget if it

**Reads** a valid destination address from the input payload

**Transfers** control to that location

Gadget space for a typical process is very small

A few MB vs. 4GB

Accidental gadget execution rare, but still possible!

How often does it happen?

To what extent?
Random Gadgets in Benign Inputs

% of inputs

Gadgets

Random binary data
Random ASCII data
Network streams
PDF memory buffers

three or less unique gadgets
starting exec - len: 0x1000 pos 0x11: PC: 0x773E274B
773e274b D6  salc
773e274c 83F8FF cmp eax,0xffffffff
773e274f 75DB  jnz 0x773e272c
773e2751 EB77  jmp 0x773e27ca
773e27ca 83C8FF or eax,0xffffffff
773e27cd 5F  pop edi
773e27ce 5E  pop esi
773e27cf 5B  pop ebx
773e27d0 C9  leave
773e27d1 C21000 retn 0x10
-------- end of gdgt 773E274B
283b413f FFCD  dec ebp
283b4141 FF8D4D00E985 dec [ebp-0x7a16ffb3]
283b4147 25CDFF8D4D and eax,0x4d8dffcd
283b414c 58  pop eax
283b414d E948A0D1FF jmp 0x280ce19a
280ce19a E906FFFFFF jmp 0x280ce0a5
280ce0a5 56  push esi
280ce0a6 8BF1  mov esi,ecx
280ce0a8 8BF8  mov eax,[esi]
280ce0aa FF4804  dec [eax+0x4]
280ce0ad 8BF804  mov eax,[eax+0x4]
280ce0b0 85C0  test eax,eax
280ce0b2 7F1C  jnle 0x280ce0d0
280ce0d0 5E  pop esi
280ce0d1 C3  ret
-------- end of gdgt 283B413F
282f2d6b 7402  jz 0x282f2d6f
282f2d6d 8919  mov [ecx],ebx
282f2d6f 895831  mov [eax+0x31],bl
282f2d72 C6403001  mov byte [eax+0x30],0x1
282f2d76 5B  pop ebx
282f2d77 C3  ret
-------- end of gdgt 282F2D6B
283f0913 F4  hlt

Sequence of random gadgets in benign input
starting exec - len: 0x128 pos 0x0: PC: 0x070072F7
070072f7 58    pop eax
070072f8 C3    ret
------------- end of gdgt 070072F7
070015bb 59    pop ecx
070015bc C3    ret
------------- end of gdgt 070015BB
0700154d 8908    mov [eax],ecx
0700154f C3    ret
------------- end of gdgt 0700154D
070015bb 59    pop ecx
070015bc C3    ret
------------- end of gdgt 070015BB
07007fb2 8B01    mov eax,[ecx]
07007fb4 C3    ret
------------- end of gdgt 07007FB2
070015bb 59    pop ecx
070015bc C3    ret
------------- end of gdgt 070015BB
0700a8ac 8901    mov [ecx],eax
0700a8ae 33C0    xor eax,eax
0700a8b0 C3    ret
------------- end of gdgt 0700A8AC
070015bb 59    pop ecx
070015bc C3    ret
------------- end of gdgt 070015BB
0700a8ac 8901    mov [ecx],eax
0700a8ae 33C0    xor eax,eax
0700a8b0 C3    ret
------------- end of gdgt 0700A8AC
070072f7 58    pop eax
070072f8 C3    ret
------------- end of gdgt 070072F7
070052e2 FF10    call [eax]
0700d731 8B45DC    mov eax,[ebp-0x24]
0700d734 C3    ret
...
Number of unique gadgets in the same execution chain

benign inputs

ROP exploits

3 < detection threshold < ?
## Tested ROP Exploits/Generic ROP Payloads

<table>
<thead>
<tr>
<th>Exploit/Payload</th>
<th>EDB-ID</th>
<th>Tested Platform</th>
<th>Space</th>
<th>Executed Gadgets</th>
<th>Unique Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Reader v9.3.0</td>
<td>16670</td>
<td>WinXP SP3</td>
<td>17.7MB</td>
<td>47</td>
<td>8</td>
</tr>
<tr>
<td>Adobe Reader v9.3.0</td>
<td>16687</td>
<td>WinXP SP3</td>
<td>17.7MB</td>
<td>60</td>
<td>12</td>
</tr>
<tr>
<td>Adobe Reader v9.3.4</td>
<td>16619</td>
<td>Win7 SP1</td>
<td>864KB</td>
<td>33</td>
<td>10</td>
</tr>
<tr>
<td>Adobe Reader v9.3.4</td>
<td>16667</td>
<td>Win7 SP1</td>
<td>17.7MB</td>
<td>60</td>
<td>12</td>
</tr>
<tr>
<td>Winamp v5.572</td>
<td>14068</td>
<td>Win7 SP1</td>
<td>17.7MB</td>
<td>126</td>
<td>21</td>
</tr>
<tr>
<td>Integard Pro v2.2.0</td>
<td>15011</td>
<td>Win7 SP1</td>
<td>724KB</td>
<td>165</td>
<td>16</td>
</tr>
<tr>
<td>Mplayer Lite r33064</td>
<td>17124</td>
<td>Win7 SP1</td>
<td>6.4MB</td>
<td>179</td>
<td>16</td>
</tr>
<tr>
<td>All to MP3 Conv. v2.0</td>
<td>17252</td>
<td>WinXP SP3</td>
<td>9.4MB</td>
<td>388</td>
<td>16</td>
</tr>
</tbody>
</table>

3 < detection threshold < 8
Use Cases

Network-level detection
  Incorporated ROPscan in Nemu, a shellcode detector
  Detection of both layers: ROP code and legacy shellcode

PDF scanning
  Incorporated ROPscan in MDScan, a PDF file scanner
  Scan newly allocated memory by the JS interpreter

Easy to incorporate into existing detectors
  Scan arbitrary data
  At least an order of magnitude faster than shellcode detection
How does ROP change system protection?
ASLR is not Fully Adopted

Executable programs in Ubuntu Linux
Only 66 out of 1,298 binaries in /usr/bin [SAB11]

Popular third-party Windows applications
Only 2 out of 16 [Pop10]

Even applications that enable ASLR sometimes have statically mapped DLLs
EMET forced randomization
Information Leaks Break ASLR [Ser12]

[Windows 7] My heap address is 0xc6f4758 and ntdll base is 0x774d0000
ROP Defenses

- ROPdefender [DSW11]
- DROP [CXS+09]
- DROP++ [CXH+11]
- G-Free [OBL+10]
- Return-less [LWJ+10]
- CFL [BJF11]

Input: Program Binary vs. Source Code
Runtime Overhead: High vs. Low
In-Place Code Randomization

Software diversification

Applicable on third-party applications

Zero (non-measurable) performance overhead
Why In-Place?

Randomization usually changes the code size
   Need to update the control-flow graph (CFG)

Accurate disassembly of stripped binaries is hard
   Incomplete CFG (data vs. code)
   Code resize not an option

Must randomize in-place!
Code Transformations

Instruction Substitution

Instruction Reordering
   Intra Basic Block
   Register Preservation Code

Register Reassignment
Instruction Substitution

```
add [edx],edi
ret

mov al,0x1
cmp al,bl
lea eax,[ebp-0x80]

add [eax],edi

fmul [ebp+0x68508045]

mov al,0x1
cmp bl,al
lea eax,[ebp-0x80]
```
Instruction Reordering (Intra Basic Block)

mov eax,[ecx+0x10]
push ebx
mov ebx,[ecx+0xC]
cmp eax,ebx
mov [ecx+0x8],eax
jle 0x5c
push ebx
or al,0xC
ret
Instruction Reordering (Intra Basic Block)

8B 41 10  mov eax,[ecx+0x10]
53         push ebx
8B 59 0C  mov ebx,[ecx+0xC]
3B C3      cmp eax,ebx
89 41 08  mov [ecx+0x8],eax
7E 4E      jle 0x5c
        inc ecx
10 89 41 08 3B C3
     adc [ecx-0x3CC4F7BF],cl
Register Preservation Code Reordering

Prolog

```
push ebx
push esi
mov ebx,ecx
push edi
mov esi,edx
...```

Epilog

```
pop edi
pop esi
pop ebx
ret
...```

```
pop esi
pop ebx
pop edi
ret
...```
Register Reassignment

function:
  push esi
  push edi
  mov edi,[ebp+0x8]
  mov eax,[edi+0x14]
  test eax,eax
  jz 0x4A80640B
  mov ebx,[ebp+0x10]
  push ebx
  lea ecx,[ebp-0x4]
  push ecx
  push edi
  call eax
  ...

Live regions

function:
  push esi
  push edi
  mov eax,[ebp+0x8]
  mov edi,[edi+0x14]
  test eax,eax
  jz 0x4A80640B
  mov ebx,[ebp+0x10]
  push ebx
  lea ecx,[ebp-0x4]
  push ecx
  push eax
  call edi
  ...

Implementation: Orp

Focused on the Windows platform
    Could be integrated in Microsoft’s EMET

CFG extraction using IDA Pro
    Implicitly used registers
    Liveness analysis (intra and inter-function)
    Register categorization (arg., preserved, …)
    Randomization
    Binary rewriting (relocations fixing, …)
Evaluation

Correctness and performance
  Used Wine’s extensive test suite with randomized versions of Windows DLLs

Randomization Coverage

Effectiveness against real-world exploits

Robustness against ROP Compilers
Randomization Coverage

Dataset: 5,235 PE files (~0.5GB code) from Windows, Firefox, iTunes, Reader
### Real-World Exploits

<table>
<thead>
<tr>
<th>Exploit/Reusable Payload</th>
<th>Unique Gadgets</th>
<th>Modifiable</th>
<th>Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Reader v9.3.4</td>
<td>11</td>
<td>6</td>
<td>287</td>
</tr>
<tr>
<td>Integard Pro v2.2.0</td>
<td>16</td>
<td>10</td>
<td>322K</td>
</tr>
<tr>
<td>Mplayer Lite r33064</td>
<td>18</td>
<td>7</td>
<td>1.1M</td>
</tr>
<tr>
<td>msvcr71.dll (While Phosphorus)</td>
<td>14</td>
<td>9</td>
<td>3.3M</td>
</tr>
<tr>
<td>msvcr71.dll (Corelan)</td>
<td>16</td>
<td>8</td>
<td>1.7M</td>
</tr>
<tr>
<td>mscorie.dll (White Phosphorus)</td>
<td>10</td>
<td>4</td>
<td>25K</td>
</tr>
<tr>
<td>mfc71u.dll (Corelan)</td>
<td>11</td>
<td>6</td>
<td>170K</td>
</tr>
</tbody>
</table>

*Modifiable gadgets were not always directly replaceable!*
ROP Compilers

*Is it possible to create a randomization-resistant ROP payload?*

Using only the remaining non-randomized gadgets

Tested two ROP payload construction tools

**mona.py**: constructs DEP+ASLR bypassing code

Allocate a WX buffer, copy shellcode, and jump to it

**Q**: state-of-the-art ROP compiler [SAB11]

Designed to be robust against small gadget sets
### ROP Compiler Results

<table>
<thead>
<tr>
<th>Non-ASLR Code Base</th>
<th>Mona</th>
<th></th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Rand.</td>
<td>Original</td>
</tr>
<tr>
<td>Adobe Reader v9.3.4</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Integard Pro v2.2.0</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Mplayer Lite r33064</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>msvcr71.dll</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>mscorie.dll</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>mfc71u.dll</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Both tools failed to construct ROP payloads using non-randomized code!
Summary

Return-Oriented Programming is increasingly used in real-world exploits

ROP payloads can be detected using speculative code execution

In-place code randomization prevents real exploits and ROP compilers

Get the code (Python)
http://nsl.cs.columbia.edu/projects/orp
References


[CDD+10] Stephen Checkoway et al. Return-oriented programming without returns. CCS, 2010


Modifiable Gadgets

![Graph showing the cumulative fraction of PE files for modifiable gadgets with and without the extracted code.](image-url)
Impact on Broken Gadgets’ Instructions

% of all broken gadgets of given length

Gadget length (instructions)

5
4
3
2

(darkest color) leftmost instr.
(lightest color) rightmost instr.
Randomization Entropy for Broken Gadgets

![Graph showing the cumulative fraction of gadgets against the number of possible randomized versions.](image-url)