Evaluating a ROP Defense Mechanism

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis
Columbia University
Outline

• Background on ROP attacks
• ROP Smasher
• Evaluation strategy and results
• Discussion
Machine Code-Level Attacks & Defenses

Code Injection

Code-reuse

W ⊗ X

ASLR
Information Leaks Break ASLR [Ser12]

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

- Code randomization
- Applicable on third-party applications
- (Practically) Zero performance overhead
Return-Oriented Programming

Stack

<table>
<thead>
<tr>
<th>esp</th>
<th>0xb8800000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x00000001</td>
</tr>
<tr>
<td></td>
<td>0xb8800010</td>
</tr>
<tr>
<td></td>
<td>0x00000002</td>
</tr>
<tr>
<td></td>
<td>0xb8800020</td>
</tr>
<tr>
<td></td>
<td>0xb8800010</td>
</tr>
<tr>
<td></td>
<td>0x00400000</td>
</tr>
<tr>
<td></td>
<td>0xb8800030</td>
</tr>
</tbody>
</table>

Code

0xb8800000:
pop eax
ret

... 0xb8800010:
pop ebx
ret

... 0xb8800020:
add eax, ebx
ret

... 0xb8800030:
mov [ebx], eax
ret

Actions

eax = 1

ebx = 2
eax += ebx

ebx = 0x400000
*ebx = eax
ROP Defenses

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

Performance Overhead vs. Requires

- **Low**Requires
- **High** Requires

Program binary vs. Source code
Why In-Place?

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

• But, accurate static disassembly of stripped binaries is hard
  ➔ Incomplete CFG (data vs. code)
  ➔ Code resize not an option

• **Must randomize in-place!**
Randomizations

• 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 [edx],edi
ret

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

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

59       push ebx
0C 3B    or al,0x3B
C3       ret
Instruction Reordering (Intra BBL)

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

41       inc  ecx
10 89 41 08 3B C3  adc  [ecx-0x3CC4F7BF],cl
Register Preservation Code Reordering

Prolog

Epilog
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
eax  edi

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

8/14/2012
Implementation – Orp

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

• CFG is extracted using IDA Pro
  – Implicitly used registers
  – Liveness analysis (intra and inter-function)
  – Register categorization (arg., preserved, etc.)
  – Randomizations
  – Binary rewriting (relocations fixing, etc.)
Evaluation Strategy

• How well will it stop future attacks?
  – Function of attack surface, attack intent, and attacker creativity

• Proxy characteristics
  – How well it stops current attacks?
  – How likely it is to “destroy” gadgets?

• Other considerations
  – Performance, ease of use/deployment, impact on functionality
Evaluation

• Correctness and performance
  – Execute Wine’s test suite using randomized versions of Windows DLLs
  – This is relatively easy
Effectiveness

• What we did
  – Randomization coverage: how many gadgets do we break?
    • Challenge: how do we even determine what is a (useful) gadget?
  – Real-world exploits
  – ROP compilers

• What we did not do
  – Create attacks, or invite others to do so
Randomization Coverage

Dataset: 5,235 PE files (~0.5GB code) from Windows, Firefox, iTunes and Reader

Instruction Substitution
Intra Basic Block Reordering
Register Preservation Code Reordering
Register Reassignment
Total
### 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>mscorei.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

• Mona.py constructs DEP+ASLR bypassing code
  – Allocate a WX buffer, copy shellcode and jump

• Q [SAB11] is the state-of-the-art ROP compiler
  – Designed to be robust against small gadget sets

• *Is it possible to create a randomization-resistant payload?*
### ROP Compilers Results

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

Both failed to construct payloads from non-randomized code!
Discussion (-)

• No guarantee that we can destroy all gadgets
  – No guarantee that we can destroy all instances of those gadgets that an attacker needs for a specific attack
  – No guarantee that we can destroy all useful gadgets

• Probabilistic protection (attacker can get lucky)

• Perhaps we can use complementary techniques for the remaining cases?
  – Does not impact any other security techniques
Discussion (+)

• Reasonable risk metrics are (perhaps) possible
  – Fairly accurate estimate of number of gadgets changed
  – Relatively good estimate on average and individual-gadget entropy

• Absent bugs in our code, no impact on program correctness

• Static, low-overhead technique → good enough?
Summary

• In-place code randomization
  – Requires no source code or debug symbols
  – (Practically) Zero performance overhead
  – Breaks 80% of gadgets
  – Prevented real exploits and ROP compilers

• Even for a seemingly well-defined problem, definitive evaluation is challenging

• 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