Isomeron

Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming

Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi
CASED/Technische Universität Darmstadt, Germany

Kevin Z. Snow, Fabian Monrose
Department of Computer Science
University of North Carolina at Chapel Hill, USA
The Big Picture

Return-Oriented Programming (ROP)

<table>
<thead>
<tr>
<th>Code</th>
<th>Data (Stack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ins seq; ret</td>
<td>gadget ptr</td>
</tr>
<tr>
<td>ins seq; ret</td>
<td>gadget ptr</td>
</tr>
</tbody>
</table>

Fine-grained code randomization

<table>
<thead>
<tr>
<th>Code</th>
<th>Data (Stack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ins seq; ret</td>
<td>gadget ptr</td>
</tr>
<tr>
<td>ins seq; ret</td>
<td>gadget ptr</td>
</tr>
</tbody>
</table>

Isomeron [This talk]
ROP Adversary Model/Assumption

1. Adversary can hijack control-flow (buffer overflow)
2. Adversary knows the memory layout (memory disclosure)
3. Adversary can construct gadgets
4. Adversary can write ROP payload in the data area (stack/heap)

Application

Code Area

Data Area

ROP Payload

Shared Libraries

MEMORY
Application Address Space

Gadget Space (e.g., Shared Libraries)

MOV
ESP
LNOP
LOAD
ADD
CALL
XOR
STORE
DEFENSES

(Fine-Grained) Code Randomization
Code-Randomization Approaches

• Base address permutation: Address Space Layout Randomization (ASLR)
• Function permutation: ASLP [ACSAC’06]
• Basic block permutation: STIR [CCS’12], XIFER [ASIACCS’13]
• Instruction-level randomization: IRL [S&P’12]
• In-place randomization: ORP [S&P’12]
A severe attack against fine-grained ASLR

Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization

*IEEE Security and Privacy 2013, and Blackhat 2013*

Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, Ahmad-Reza Sadeghi
Just-In-Time ROP [IEEE S&P‘13]

1) Leak initial code pointer
2) Leak whole code page
3) Disassemble leaked page
4) Gadgets
5) Code Pointer of direct branches
6) provide
7) Output
8) Hijack instruction pointer

ROP-Compiler

Application Address Space

(randomized)
Code

Data

ROP Payload

isomeron | NDSS 2015
Defense against Just-In-Time ROP

1) Leak initial code pointer
2) Leak whole code page
8) Hijack instruction pointer

Oxymoron
Backes et al.
[USENIX SEC’14]

mov  eax, 0xcoffee
call  seg:[2]
[..]
add  eax, ebx
jmp  seg:[10]
pop  ecx
ret

ROP-Compiler

8) Output
7) ROP Payload
6) provide
4) Gadgets

Attack Description

Application Address Space

(randomized)
Code
Data

move 0xbeef
add eax, ebx
jmp 0x29A
pop ecx
ret
Can we bypass Oxymoron-like approaches?
Sources of Code Pointers

• Virtual method tables
• Stack frames
• Exception handling information
• Loader data
  – Import/export table
  – Global offset table
How to Bypass Oxymoron

- Use page addresses for normal JIT-ROP attack
- Disassemble pages
- Gadget search
- ROP compiler
- Payload execution

Application Address Space

- (randomized) Code
- virtual method table
- Method #0
- ...
- Method #N
- Data
- C++ Object

Information Leak
Our Solution:
Isomeron
Isomeron - High-level Idea

- Create a randomized isomer (copy) of the application
  - Preserve semantics of the function
  - Affects the gadget locations
Isomeron - High-level Idea

• Control-flow randomization
Isomeron - Call

Execution Diversifier (TCB)

- Identify Origin: A or \( A_{DIV} \)
- Diversifier ENTRY
- Look-Up Target Address
- Adjust return address
- Record Decision
- Diversifier Decisions
- Diversifier EXIT

Isomer

Address Space of Application A

Isomer\(_{rand}\)

Func\(_{A_{DIV}}\):
INS2
CALL Func\(_{B_{DIV}}\)
INS1

Func\(_{B_{DIV}}\):
RET

Func\(_{A}\):
INS1
CALL Func\(_{B}\)
INS2

Func\(_{B}\):
RET

Original Binary

Diversifier ENTRY

Diversifier EXIT

Identify Origin: A or \( A_{DIV} \)

Look-Up Target Address

Adjust return address

Record Decision

Diversifier Decisions
Isomeron - Return

Address Space of Application A

Isomer

Isomer_{rand}

 Func_A_{div}:
 INS2
 CALL Func_B_{div}
 INS1

 Func_B_{div}:
 RET

 Func_A:
 INS1
 CALL Func_B
 INS2

 Func_B:
 RET

Execution Diversifier (TCB)

Lookup decision

Diversifier ENTRY

Adjust return address if necessary

Diversifier EXIT

Diversifier Decisions

Isomer | NDSS 2015
Isomeron - Attack

Execution Diversifier (TCB)

Hijack

Diversifier ENTRY

Adjust return address if necessary

Diversifier EXIT

Lookup decision

Diversifier Decisions

Isomer

Isomer_{rand}

Func_{A_{DIV}}:
  INS2
  CALL Func_{B_{DIV}}
  INS1

Func_{B_{DIV}}:
  RET

Func_{A}:
  INS1
  CALL Func_{B}
  INS2

Func_{B}:
  RET

Address Space of Application A
Isomeron - Security

- Conventional ROP
  - Code randomization
- (JIT) ROP
  - Code randomization and control flow randomization
- Ret-to-libc
  - Non-trivial in general
  - We restrict ret-to-libc to targets of benign indirect calls
Implementation & Challenges

• Multiple (randomized) copies
  – Custom dynamic binary instrumentation (DBI) framework
  – Existing DBI tools did not fulfill our requirements
  – Performance penalties

• Protect caller information
  – Segmentation (hardware dependent)
  – Software Fault Isolation
Current and Future Work

• Compiler-based randomization solutions
  – Isomeron with compiler
    • Use compiler to randomize code and hardware support to enforce real X-only memory

• CFI-based solutions
    • Bypass almost all C++ CFI solutions
Thank you.
Backup
Isomeron - Security

• Special case of gadget pairs
  – intended gadgets $G$ performs operation
  – other gadget $G_{\text{nop}}$ performs nop

• Gadget space
  – limited to $(G, G_{\text{nop}})$ where $G$ does not modify the input value
  – Examples:
    • load value from stack (stack pointer is modified)
    • load constant into register
Defenses against JIT-ROP

- Oxymoron (USENIX’14)
  - Aims at preventing JIT-ROP by obfuscating destination addresses of direct branches
Isomeron - Performance

SPEC 2006
Unfortunately randomization can be bypassed

Just-In-Time ROP - Oakloand’13

Kevin Z. Snow, Fabian Monrose
Department of Computer Science
University of North Carolina at Chapel Hill, USA

Lucas Davi, Alexandra Dmitrienko,
Christopher Liebchen, Ahmad-Reza Sadeghi
CASED/Technische Universität Darmstadt, Germany
The Big Picture

Return-oriented Programming (ROP)
- Code: `ins seq; ret`
- Data (Stack): gadget ptr

Fine grained code randomization
- Code: `ins seq; ret`
- Data (Stack): gadget ptr

Just-In-Time ROP
- Code: `ins seq; ret`
- Data (Stack): gadget ptr

Sample code:
```
mov eax, 0xc0ffee
call 0xbeef
[...]
add eax, ebx
jmp 0x29A
pop ecx
ret
```
Motivation

- Software suffers from security vulnerabilities, no end in sight
- Software complexity is increasing
  - Advanced devices
  - Many developers involved
- Complex software exposes large attack surface
- Currently runtime attacks are still a crucial threat
Return-oriented Programming

- Code-reuse attack
- Short instruction sequences ending in indirect branches
- Turing-complete
- Applicable to many architectures
Just-In-Time ROP [IEEE S&P'13]

1) Leak initial code pointer
2)* Leak whole code page
3)* Disassemble leaked page
4)* Gadgets
5)* Code Pointer of direct branches
8) Hijack instruction pointer

Disassembler
ROP-Compiler

Application Address Space
(randomized)
Code
Data
ROP Payload

Attack Description
Isomeron | NDSS 2015

1) Leak initial code pointer
2)* Leak whole code page
3)* Disassemble leaked page
4)* Gadgets
5)* Code Pointer of direct branches
8) Hijack instruction pointer
Oxymoron [USENIX Sec’14]

• **Goal**
  – Prevent conventional ROP by applying page-based randomization
  – Prevent JIT-ROP from disclosing pages by obfuscating destination addresses of direct branches
  – Allow code sharing despite randomization

• **Approach**
  – Addresses of direct branches are substituted through indirect branches
  – These indirect branches use segmentation registers
  – Destination of direct branches are maintained in a separate table allocated at a random address in memory
Oxymoron [USENIX Sec’14]

Disassemble, extract new code pointers

Leak whole code page

Application Address Space

(randomized) Code

Data

Branches are obfuscated through segment registers

Oxymoron

Disassemble, extract new code pointers
Isomeron

• Create a randomized copy of the application
• Ensure that gadgets at the same offset have different semantics
• Switch randomly between both copies at every function call
• Ensure that returns always arrive the original caller
Isomeron – High level

- Create a randomized copy of the application
- Ensure that gadgets at the same offset have different semantics
- Switch randomly between both copies at every function call
- Ensure that returns always arrive the original caller
Isomeron

• Switch randomly between both copies at every function call and save call origin
Isomeron

• Ensure that returns always arrive the original caller
• Attacker is forced to guess the call origin to execute the intended gadget
Isomeron

- Switch randomly between both copies at every function call
- Ensure that returns always arrive the original caller
Isomeron

Attacker exploitation flow:

1. Leak address of both copies
2. Specify gadget at 0
3. Call vulnerable function
4. Attach its new function execution to a different gadget
   - Gadget #0 gets original image
   - Gadget #0 gets another gadget with a different semantic

3. Randomize each call
   - P(Func) = 0.5
   - P(Func') = 0.5

4. Return to original caller (unknown to the attacker!)

Execution is redirected to Function B'

Redirected to original caller ⇒ Gadget #0

Leak

Call Function B

Redirected to original caller

Gadget #0