Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future?

Ahmad-Reza Sadeghi
Technische Universität Darmstadt,
Intel Collaborative Research Institute for Secure Computing, Germany
Motivation

PROTECT
ALL THE NATIVE CODES
Three Decades Memory Corruption

1988
Buffer Overflow (Morris worm)

1996
Aleph1 - “Smashing The Stack For Fun And Profit” (Phrack #49)

2001
Solar Designer – Return into libc (Bugtraq)

2007
Shacham – Return-oriented Programming is Turing-complete (ACM CCS’07)
Runtime Attacks & Defenses: Continuing Arms Race

Defenses
- CFI, ROPGuard, BinCFI, ROPecker, kBouncer, CCFIR, vTint, vfGuard, SafeDispatch, MoCFI, RockJIT, TVip, StackArmor, CPI/CPS, Oxymoron, XnR, Isomeron, O-CFI, HAFIX, ...

Attacks
- ROP wo Returns, Out-of-Control, Stitching the Gadgets, SROP, JIT-ROP, BlindROP, COOP, StackDefiler, "Missing the point"

Bad news: No practical and secure solution known yet
Industry Solutions

Google:
- IFCC
- VTV

Microsoft:
- EMET
- CF Guard

More bad news:
Attacks coming up
**Code Injection Attacks**

**Control-Flow Graph (CFG)**

Entry: instruction target of a branch (e.g., first instruction of a function)

Exit: Any branch (e.g., indirect or direct jump/call, return)

**Basic Block (BBL) A**

1. Buffer overflow

2. Code Injection

3. Control-flow deviation

**Basic Block (BBL) B**

**Entry: asm_ins, ...**

**Exit:**
Raising the Bar for Code Injection: Data Execution Prevention (DEP)

- Prevent execution from a writeable memory (data) area
  - All modern systems: Windows, Linux, Mac OS X, BSD, iOS, Android...

[code reuse attacks]

But can be bypassed by code reuse attacks
Code Reuse Attacks

Control-Flow Graph (CFG)

Basic Block (BBL) A

ENTRY asm_ins, ...
EXIT

Basic Block (BBL) B

ENTRY asm_ins, ...
EXIT

1. Buffer overflow

2. Control-flow deviation

Return-oriented Programming (ROP)

Data flows

Program flows
ROP: Basic Ideas

- Use **small instruction sequences** instead of whole functions
- Instruction sequences of length 2 to 5
- All sequences end with a **return** instruction
- Instruction sequences chained together as **gadgets**
- A gadget performs a particular task, e.g., load, store, xor, or branch
- Attacks launched by combining gadgets
- Generalization of return-to-libc
ROP Attack Technique: Overview

ROP shown to be Turing complete
Adversary Model / Assumptions

0. Writable ⊕ Executable

1. Address Space Layout Randomization (ASLR)

Application

- RX (Code)
- RW (Data)
- Scripting Engine (Sandboxed)

2. Disclose arbitrary (readable) memory
3. Write data memory
4. Perform arbitrary computations at runtime
HOT RESEARCH TOPIC:
Practical and secure mitigation of code reuse attacks

Turing-completeness of return-oriented programming

Copyright minions: The Chiquita company
http://minionslovebananas.com/
MAIN DEFENSE MEASURES

1. Code Randomization

2. Control Flow Integrity (CFI)

Copyright minions: The Chiquita company
http://minionslovebananas.com/
Randomization vs. CFI

Randomization
- Low Performance Overhead
- Scales well to complex Software (OS, browser)
- Vulnerable to Information Disclosure
- Security depends on Entropy

Control-flow Integrity
- “Provable” Security (Explicit Control Flow Checks)
- Tradeoff: Performance & Security
- Challenging to integrate in complex software

Tradeoff: Performance & Security vs. “Provable” Security

Security depends on Entropy vs. Explicit Control Flow Checks

Randomization scales well to complex software, has low performance overhead, but is vulnerable to information disclosure and security depends on entropy.

Control-flow Integrity provides explicit control flow checks for "provable" security, but has a tradeoff between performance and security, and can be challenging to integrate in complex software.
EPISODE I

Code Randomization

Make the location of gadgets unpredictable
- Instruction reordering/substitution within a BBL
  **ORP** [Pappas et al., IEEE S&P 2012]

- Randomizing each instruction’s location:
  **ILR** [Hiser et al., IEEE S&P 2012]

- Permutation of BBLs:
  **STIR** [Wartell et al., CCS 2012] & **XIFER** [with Davi et al., AsiaCCS 2013]
JIT-ROP:
Severe Attack against Fine-grained ASLR

Copyright minions: The Chiquita company
http://minionslovebananas.com/

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

1. Undermines fine-grained ASLR
2. Shows memory disclosures are far more damaging than believed
3. Can be instantiated with real-world exploit
Goal: Exploit a memory disclosure
- Leak of a single address leading to leak of entire memory pages

Observations
- Leaked address will reside in a 4KB aligned memory page
- Determine the page boundaries and disassemble the 4 KB page
- Disassembled page contains references to other pages
Gadget Finding and Payload Generation

Page 1
- MOV EAX, EBX
- RET
- MOV EAX, (EBX)
- RET
- SUB EAX, EBX
- RET
- INT 0x80
- MOV ESP, EAX
- RET

Page 2
- LOAD
- SUB
- MOV
- INT

JIT-ROP Payload
- Stack Pivot
- LOAD
- MOV
- INT

Exploit Description (High-Level Language)

Gadget Pool
Code Randomization: Attack & Defense Techniques

Single pointer disclosure

Fine-grained Randomization

Direct code disclosure

Execute-only Memory

Indirect code disclosure

Code-pointer hiding

Pointers

Return Address

Function_A

Function_B

<instructions>
call Function_A
pop ???
pop ???
ret

Trampoline

Attack Timeline

The Info Leak Era on Software Exploitation [Serna BH/US’12]

Just-In-Time ROP [Snow et al. IEEE S&P’13]

Isomeron (Attack) [Davi et al. NDSS’15]
Towards Resilience to Memory Disclosure

Readactor:
Practical Code Randomization Resilient to Memory Disclosure
IEEE Security and Privacy 2015

Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz
**Readactor**

- **Goal**
  - Prevent ROP/JIT-ROP attacks
  - Avoid information leakage

- **Approach**
  - Compiler guarantees strict code and data separation
  - Fine-grained randomization
  - Hardware support marks code pages as execute only
  - Read of a code page results in process termination
  - Code pointer hiding through trampolines

- **Implementation**
  - Applied to complex software (Chromium)
  - Handles JIT Code
  - Extensive evaluation
Readactor: Architecture

Compiler

Runtime

Application

• Code / Data separation
• Fine-grained code randomization
• Code-pointer Hiding

Operating System

• Execute-only support

Thin-Hypervisor

Extended Page Tables (EPT)

Hardware

Memory Virtualization

Compile time

Runtime
EPISODE II
Control-Flow Integrity (CFI)
Restricting indirect targets
to a pre-defined control-flow graph
Original CFI Label Checking
[Abadi et al., CCS 2005 & TISSEC 2009]
Original CFI: Benefits and Limitations

- Fine-grained protection
- Blackbox Vulcan (unpublished)
- Require side info (debug symbols)
- Performance overhead
“Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed

- CCFIR [IEEE S&P’13]
- MS BlueHat Prize
- kBouncer [USENIX Sec’13]
- ROPecker [NDSS’14]
- MS BlueHat Prize
- ROPGuard [Microsoft EMET]
- CFI for COTS Binaries [USENIX Sec’13]

Bypassing Coarse-grained CFI

Stitching the Gadgets
Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, Fabian Monrose
USENIX Security 2014

Counterfeit OOP
Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz
IEEE S&P 2015

https://www.pinterest.com/search/pins/?q=minion%20searching
Coarse-grained CFI: General Idea

Reducing number of labels

Allowed paths: 1→2 and 2→1
Main Coarse-grained CFI Policies

- CFI Policy 1: Call-Preceded Sequences
  - Returns need to target a call-preceded instruction

- CFI Policy 2: Behavioral-Based Heuristics
  - Threshold Setting
    - kBouncer: (N=7; S≤20)
    - ROPecker: (N=11; S≤6)

Application

```
<table>
<thead>
<tr>
<th>CALL A</th>
<th>asm_ins</th>
<th>CALL B</th>
<th>asm_ins</th>
<th>CALL C</th>
<th>asm_ins</th>
</tr>
</thead>
</table>
```

Diagram:

1. `< S`  
2. `< S`  
   ...  
N. `< S`

1. `> S`  
2. `< S`  
   ...  
N. `< S`
# Most Restrictive Coarse-grained CFI

<table>
<thead>
<tr>
<th>CFI Policy</th>
<th>kBouncer</th>
<th>ROPecker</th>
<th>ROPGuard EMET</th>
<th>CFI for COTS Binaries</th>
<th>Über-CFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI Policy 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call-Preceded Sequences</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>CFI Policy 2</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Behavioral-Based Heuristics</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of CFI Check</td>
<td>WinAPI</td>
<td>2 Pages Sliding Window / Critical Functions</td>
<td>WinAPI / Critical Functions</td>
<td>Indirect Branch</td>
<td>Any Time</td>
</tr>
</tbody>
</table>

- ✔️ CFI Policy
- ✗ No Restriction

Here only core policies shown. Other policies available in our analysis.
Hardware-assisted CFI

HAFIX:
Hardware-Assisted Flow Integrity Extension
*Design Automation Conference (DAC 2015)*
Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl, Debayan Paul, Ahmad-Reza Sadeghi, Dean Sullivan

https://www.pinterest.com/search/pins/?q=minion%20searching
Why Leveraging Hardware for CFI?

Efficiency

- Dedicated CFI instructions
  - CFI_RETURN
  - CFI_JUMP
  - CFI_CALL

Security

- Isolated CFI storage
  - CFI Memory
    - Branch Targets
HAFIX Overview

❖ **Goal**
  ❖ Efficient hardware-based implementation of CFI

❖ **Defense Scope**
  ❖ Protecting function returns
    → *tackling shadow stack overhead*
  ❖ Assuming forward-edge CFI
    → [Tice et al., USENIX Sec. 2014]

❖ **Implementation**
  ❖ CFI hardware implementation for Intel Siskiyou Peak and SPARC
  ❖ 2% performance overhead
Function **returns** are only allowed to return to an active call site (CFIRET, CFIDEL)

Function **calls** need to target the beginning of a function (CFIBR)
HAFIX-instrumented Code

Program Code

Function A (0025)
- CFIBR 0025
- CALL Function B
- CFIRET 0025
- CFIDEL 0025; RET

Function B (0099)
- CFIBR 0099
- Instruction 3
- CFIDEL 0099; RET

Function C (0444)
- CFIBR 0444
- CALL Function X
- CFIRET 0444
- CFIDEL 0444; RET

CFI Label Memory
- 0025
- 0099

1. Activate Label 0025
2. Label 0025 active → Continue execution
3. Activate Label 0099
4. Deactivate Label 0099

Label 0444 not active → Stop execution

No CFIRET → Stop execution
Related Work

- **Hardware CFI** [Budiu et al, ASID 2006]
  - CFI state model, i.e., indirect branches lead to a CPU state switch (adopted by HAFIX)
  - Does not consider runtime call-graph for returns

- **kBouncer** [Pappas et al., USENIX Sec. 2013]
  - Performs CFI validation based on branch information recorded in LBR (Last Branch Record)
  - Deploys coarse-grained CFI policies

- **Branch Regulation** [Kayaalp et al., ISCA 2012]
  - Deploys a shadow stack for return address protection
  - Hence, it needs cumbersome return address exception handling
The Quest for the Right Balance:
Security & Practicality
(Selected Summary)
**Code Reuse Defense Proposals**

### Randomization
- **Oxymoron**
  [Backes et al., USENIX Sec. 2014]
- **Execute-No-Read**
  [Backes et al., ACM CCS 2014]
- **Isomeron**
  [Davi et al., NDSS 2015]
- **O-CFI**
  [Mohan et al., NDSS 2015]
- **Readactor**
  [Crane et al., IEEE S&P 2015]

### Code Pointer Integrity
- **Crypto-enforced CFI**
  [Mashtizadeh et al., TR 2014]
- **Code Pointer Integrity**
  [Kuznetsov et al., OSDI 2014]
# Code Reuse Defense Defense Proposals

## Control-flow Integrity

- **MoCFI**
  - [Davi, NDSS 2012]

- **kBouncer**
  - [Pappas et al., USENIX Sec 2013]

- **CCFIR**
  - [Zhang, IEEE S&P 2013]

- **RockJIT**
  - [Nui et al., ACM CCS 2014]

- **SAFEDISPATCH**
  - [Jang et al., NDSS 2014]

- **ROPGuard**
  - [Fratric, 2012]

- **BinCFI**
  - [Zhang et al., USENIX Sec 2013]

- **ROPecker**
  - [Cheng et al., NDSS 2014]

- **T-VIP**
  - [Gawlik et al., ACSAC 2014]

- **Forward-Edge CFI**
  - [Tice et al., USENIX Sec. 2014]

## Industry CFI

- **EMET**
  - [Microsoft, 2009]

- **Control-flow Guard**
  - [Microsoft, Win8.1+]

- **VTint**
  - [Zhang et al., NDSS 2015]

- **vfGuard**
  - [Prakash et al., NDSS 2015]

- **HAFIX**
  - [Davi et al., DAC 2015]
**Recent Attacks**

**Control-Flow Integrity**
- Out of control
  
  [Göktas et al., IEEE S&P 2014]
- Stitching the gadgets
  
  [Davi et al., USENIX Sec. 2014]
- ROP is still dangerous
  
  [Carlini et al., USENIX Sec. 2014]
- Size does matter
  
  [Göktas et al., USENIX Sec. 2014]
- COOP
  
  [Schuster et al., IEEE S&P 2015]

**Randomization**
- JIT-ROP
  
  [Snow et al., IEEE S&P 2013]
- Blind ROP
  
  [Bittau et al., IEEE S&P 2014]
- Isomeron
  
  [Davi et al., NDSS 2015]

**Code Pointer Integrity**
- Missing the Point
  
  [Evans et al., IEEE S&P 2015]

**Digital Signatures**
- Signal-oriented Programming (SROP)
  
  [Bosman et al., IEEE S&P 2014]
...and

Code Pointer Integrity (CPI)?

[Kuznetsov et al., OSDI 2014]

- Protecting critical pointers by storing them in a safe-region (separated protected memory)

Missing the Point...

[Evans et al. , IEEE S&P 2015]

- Bypasses CPI by finding the safe-region!
Conclusion

- Runtime attacks are severe threats
  - Industry acknowledged (e.g., Intel, Microsoft)
- Discussed recent attack & defense techniques
  - Attackers quickly adapt
- Recent intense research led to new insights
  - Memory disclosure is a crucial cause of threats
  - Hardware-support highly strengthens defenses
Current Work

- Extending Readactor to mitigate return-to-libc
- Using Readactor on other platforms (e.g., Android)
- Extending hardware CFI with forward-edge integrity checks
Thank you!
Talk to me:
www.trust.cased.de