The Continuing Arms Race: A Journey in the World of Runtime Exploits and Defenses

Lucas Davi, Ahmad-Reza Sadeghi

CRISP, Technische Universität Darmstadt
Intel Collaborative Research Institute for Secure Computing at TU Darmstadt, Germany
Special Session Announcement

- **Secure IoT: Utopia, Alchemy, or Possible Future?**
  - Organizers: Ahmad-Reza Sadeghi (TU Darmstadt) and Yier Jin (Univ. of Central Florida)
  - Chair: Anand Rajan (Intel Corp.)
  - Co-Chair: Saverio Fazzari (Booz Allen Hamilton, Inc.)

- **THURSDAY June 09, 10:30am - 12:00pm | 18AB**

- **Talks**
  - Things, Trouble, Trust: On Building Trust in IoT Systems
  - Exploring risk and mapping the Internet of Things with Autonomous Drones
  - Can IoT be Secured: Emerging Challenges in Connecting the Unconnected
Motivating Problem

- Software increasingly sophisticated and complex
- Various developers involved
- Native Code
- Many program bugs

Large attack surface for runtime exploits on diverse platforms
Introduction

• Vulnerabilities
  • Programs continuously suffer from program bugs, e.g., a buffer overflow
  • Memory errors
  • CVE statistics; zero-day

• Runtime Attack
  • Exploitation of program vulnerabilities to perform malicious program actions
  • Control-flow attack; runtime exploit

Focus in this tutorial
Three Decades of Runtime Attacks

- Morris Worm 1988
- return-into-libc
  - *Solar Designer* 1997
- Return-oriented programming
  - *Shacham* CCS 2007
- Continuing Arms Race
  - Borrowed Code Chunk Exploitation
    - *Krahmer* 2005
  - Code Injection
    - *AlephOne* 1996
Are these attacks relevant?
Recent Attacks

**Stagefright** [Drake, BlackHat 2015]

*These issues in Stagefright code critically expose 95% of Android devices, an estimated 950 million devices*

**Cisco Router Exploit** [2016]

*Million CISCO ASA Firewalls potentially vulnerable to attacks*
Relevance and Impact

**High Impact of Attacks**
- Web browsers repeatedly exploited in pwn2own contests
- Zero-day issues exploited in Stuxnet/Duqu [Microsoft, BH 2012]
- iOS jailbreak

**Industry Efforts on Defenses**
- Microsoft EMET (Enhanced Mitigation Experience Toolkit) includes a ROP detection engine
- Microsoft Control Flow Guard (CFG) in Windows 10
- Google’s compiler extension VTV (virtual table verification)

**Hot Topic of Research**
- A large body of recent literature on attacks and defenses
But runtime exploits have also some “good” side-effects
Apple iPhone Jailbreak

Disable signature verification and escalate privileges to root

1) Exploit PDF Viewer Vulnerability by means of Return-Oriented Programming
2) Start Jailbreak
3) Download required system files
4) Jailbreak Done

Request
http://www.jailbreakme.com/_/iPhone3,1_4.0.pdf
Tutorial Outline

1. Lecture on Runtime Exploits
   - Introduction
   - Selected Background on ARM
   - Code Injection
   - Code-Reuse Attacks
   - Modern Defense Techniques and Their Limitations
   - Hardware-Assisted Protection Schemes

2. Hands-on Lab (Runtime attacks against Android-ARM)
BASICS
What is a runtime attack?
Big Picture: Program Compilation

Source Code

C

Compile

COPY (buffer[8], *usr_input)

Executable binary

mov reg0[0-3], reg1[0-3]
mov reg0[4-n], reg1[4-n]

reg0

buffer[8]

reg1

usr_input
Big Picture: Program Execution

<table>
<thead>
<tr>
<th>MEMORY - RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize buffer[8]</td>
</tr>
<tr>
<td>Get usr_input</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CODE</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTER: 8000ABCD</td>
</tr>
<tr>
<td>buffer[4-7]: 00000000</td>
</tr>
<tr>
<td>buffer[0-3]: 00000000</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>usr_input[8-11]: 00000000</td>
</tr>
<tr>
<td>usr_input[4-7]: 00000000</td>
</tr>
<tr>
<td>usr_input[0-3]: 00000000</td>
</tr>
</tbody>
</table>
Big Picture: Program Execution

Executable binary

MEMORY - RAM

Initialize buffer[8]
Get usr_input

CODE

DATA

POINTER: 8000ABCD
buffer[4-7]: 00000000
buffer[0-3]: 00000000
... 00000000
usr_input[8-11]: CCCCCCCC
usr_input[4-7]: BBBB BBBB
usr_input[0-3]: AAAAAAAA
Big Picture: Program Execution

Executable binary

MEMORY - RAM
- Initialize buffer[8]
- Get usr_input
- COPY (buffer[8], *usr_input)

CODE

DATA
- POINTER: CCCCCCCC
- buffer[4-7]: BBBB
- buffer[0-3]: AAAAAA
- usr_input[8-11]: CCCCCC
- usr_input[4-7]: BBBB
- usr_input[0-3]: AAAAAA
Observations

- There are several observations
  1. A programming error leads to a program-flow deviation
  2. Missing **bounds checking**
     - Languages like C, C++, or assembler do not automatically enforce bounds checking on data inputs
  3. An adversary can provide inputs that influence the program flow
- What are the consequences?
General Principle of Code Injection Attacks

Control-Flow Graph (CFG)

Basic Block (BBL) A

ENTRY
asm_ins, ...
EXIT

Buffer overflow

Code Injection

Control-flow deviation

BBL B

ENTRY
asm_ins, ...
EXIT

Data flows
Program flows
General Principle of Code Reuse Attacks

Control-Flow Graph (CFG)

Basic Block (BBL) A

ENTRY
asm_ins, ...
EXIT

1 Buffer overflow

ENTRY
asm_ins, ...
EXIT

2 Control-flow deviation

Data flows
Program flows
Code Injection vs. Code Reuse

- **Code Injection** – *Adding a new node to the CFG*
  - Adversary can execute arbitrary malicious code
    - open a remote console (classical shellcode)
    - exploit further vulnerabilities in the OS kernel to install a virus or a backdoor
- **Code Reuse** – *Adding a new path to the CFG*
  - Adversary is limited to the code nodes that are available in the CFG
  - Requires reverse-engineering and static analysis of the code base of a program
BASICS
Code injection is more powerful; so why are attacks today typically using code reuse?
Data Execution Prevention (DEP)

- Prevent execution from a writeable memory (data) area

![Diagram showing Data Execution Prevention (DEP) with CODE Memory and DATA Memory with Memory Access Violation.](image-url)
Data Execution Prevention (DEP) cntd.

- Implementations
  - Modern OSes enable DEP by default (Windows, Linux, iOS, Android, Mac OSX)
  - Intel, AMD, and ARM feature a special No-Execute bit to facilitate deployment of DEP

- Side Note
  - There are other notions referring to the same principle
    - $W \oplus X$ – Writeable XOR eXecutable
    - Non-executable memory
Hybrid Exploits

• Today’s attacks combine code reuse with code injection

![Diagram showing hybrid exploits with memory and code management functions.]

1. CODE Memory I (Libraries)
   - ChangePermission()
   - CopyMemory()
   - AllocateMemory()

2. DATA Memory
   - Readable and writeable

EXECUTABLE

DATA Memory
- Readable and writeable

Malicious Code
Hybrid Exploits

- Today’s attacks combine code reuse with code injection

![Diagram showing hybrid exploits]

- CODE Memory I (Libraries) readable and executable
  - ChangePermission()
  - CopyMemory()
  - AllocateMemory()

- CODE Memory II (Libraries) readable and executable

- DATA Memory readable and writeable
  - Malicious Code

- DATA Memory readable and writeable
  - Malicious Code

1. Executable
2. Malicious Code (AllocateMemory)
3. Malicious Code (CopyMemory)
Hybrid Exploits

- Today’s attacks combine code reuse with code injection
Selected background on ARM registers, stack layout, and calling convention
ARM Overview

- ARM stands for Advanced RISC Machine
- Main application area: Mobile phones, smartphones (Apple iPhone, Google Android), music players, tablets, and some netbooks
- Advantage: Low power consumption
- Follows RISC design
  - Mostly single-cycle execution
  - Fixed instruction length
  - Dedicated load and store instructions
- ARM features XN (eXecute Never) Bit
ARM Overview

- Some features of ARM
  - Conditional Execution
  - Two Instruction Sets
    - ARM (32-Bit)
      - The traditional instruction set
    - THUMB (16-Bit)
      - Suitable for devices that provide limited memory space
  - The processor can exchange the instruction set on-the-fly
  - Both instruction sets may occur in a single program
- 3-Register-Instruction Set
  - instruction destination, source, source

\[ \text{ADD } r0, r1, r2 \]

\[ r0 = r1 + r2 \]
ARM Registers

- ARM’s 32 Bit processor features 16 registers
- All registers r0 to r15 are directly accessible

**Function arguments and results from function (caller-save)**

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
</table>

**Register variables (callee-save)**

<table>
<thead>
<tr>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
<tbody>
<tr>
<td>r8</td>
<td>r9</td>
<td>r10</td>
<td>r11</td>
</tr>
</tbody>
</table>

- Sometimes used for long jumps, i.e., branches that require the full ARM 32 Bit address space
- Intra Procedure Call Register
- Stack Pointer
- Link Register
- Program Counter
- Holds Top Address of the Stack
- Holds Return Address
- Status Register: e.g., Carry Flag
- Control Program Status Register
- Next address of instruction to be executed
- cpsr

r12/ip  
r13/sp  
r14/lr  
r15/pc
The first four arguments are passed via r0 to r3. This area is only used if more than four 4-Byte arguments are expected, or when the callee needs to save function arguments.

* Note that a subroutine does not always store all callee-save registers (r4 to r11); instead it stores those registers that it really uses/changes.
The Stack and Stack Frame Elements

- Stack is a last in, first out (LIFO) memory area where the Stack Pointer points to the last stored element on the stack.
- The stack can be accessed by two basic operations:
  1. PUSH elements onto the stack (SP is decremented)
  2. POP elements off the stack (SP is incremented)
- Stack is divided into individual stack frames:
  - Each function call sets up a new stack frame on top of the stack:
    1. Function arguments
       - Arguments provided by the caller of the function
    2. Callee-save Registers
       - Registers that a subroutine (callee) needs to reset before returning to the caller of the subroutine
    3. Return address
       - Upon function return control transfers to the code pointed to by the return address (i.e., control transfers back to the caller of the function)
    4. Saved Frame Pointer/Saved Base Pointer
       - Frame pointer/Base pointer of the calling function
       - Variables and arguments are accessed via an offset to the frame pointer/base pointer
       - Provided in register r11 (ARM code), r7 (THUMB code), or EBP (x86 code)
    5. Local variables
       - Variables that the called function uses internally
**Function Calls on ARM**

- **Branch with Link**
  - **BL addr**
  - Branches to **addr**, and stores the return address in link register **lr/r14**
  - The return address is simply the address that follows the **BL** instruction

- **BLX addr|reg**
  - Branches to **addr|reg**, and stores the return address in **lr/r14**
  - This instruction allows the exchange between ARM and THUMB
    - ARM->THUMB: LSB=1
    - THUMB->ARM: LSB=0
Function Returns on ARM

- Branches to the return address stored in the link register \( lr \)
- Register-based return for leaf functions

- \( BX \ sr \)

- Pops top of the stack into the program counter \( pc/r15 \)
- Stack-based return for non-leaf functions

\[ Branch \ with \ eXchange \ instruction \ set \]

POP \{pc\}
THUMB Example for Calling Convention

- **Function Call:** BL Function_A
  - The **BL** instruction automatically loads the return address into the link register **lr**
- **Function Prologue 1:** PUSH {r4,r7,lr}
  - Stores callee-save register r4, the frame pointer r7, and the return address lr on the stack
- **Function Prologue 2:** SUB sp,sp,#16
  - Allocates 16 Bytes for local variables on the stack
- **Function Body:** Instructions, ...
- **Function Epilogue 2:** ADD sp,sp,#16
  - reallocates the space for local variables
- **Function Epilogue 2:** POP {r4,r7,pc}
  - The POP instruction pops the callee-save register r4, the saved frame pointer r7, and the return address off the stack which is loaded it into the program counter pc
  - Hence, the execution will continue in the main function
THUMB Example for Calling Convention

- **Function Call:** `BL Function_A`
  - The `BL` instruction automatically loads the return address into the link register `lr`.
- **Function Prologue 1:** `PUSH {r4,r7,lr}`
  - Stores callee-save register `r4`, the frame pointer `r7`, and the return address `lr` on the stack.
- **Function Prologue 2:** `SUB sp,sp,#16`
  - Allocates 16 Bytes for local variables on the stack.
- **Function Body:** *Instructions, ...*
- **Function Epilogue 2:** `ADD sp,sp,#16`
  - Reallocates the space for local variables.
- **Function Epilogue 2:** `POP {r4,r7,pc}`
  - The `POP` instruction pops the callee-save register `r4`, the saved frame pointer `r7`, and the return address off the stack which is loaded into the program counter `pc`.
  - Hence, the execution will continue in the main function.
THUMB Example for Calling Convention

- Function Call: **BL Function_A**
  - The **BL** instruction automatically loads the return address into the link register **lr**
- Function Prologue 1: **PUSH {r4,r7,lr}**
  - Stores callee-save register **r4**, the frame pointer **r7**, and the return address **lr** on the stack
- Function Prologue 2: **SUB sp,sp,#16**
  - Allocates 16 Bytes for local variables on the stack
- Function Body: **Instructions, ...**
- Function Epilogue 2: **ADD sp,sp,#16**
  - Reallocates the space for local variables
- Function Epilogue 2: **POP {r4,r7,pc}**
  - The **POP** instruction pops the callee-save register **r4**, the saved frame pointer **r7**, and the return address off the stack which is loaded it into the program counter **pc**
  - Hence, the execution will continue in the main function
THUMB Example for Calling Convention

- Function Call: `BL Function_A`
  - The `BL` instruction automatically loads the return address into the link register `lr`
- Function Prologue 1: `PUSH {r4,r7,lr}`
  - Stores callee-save register `r4`, the frame pointer `r7`, and the return address `lr` on the stack
- Function Prologue 2: `SUB sp,sp,#16`
  - Allocates 16 Bytes for local variables on the stack
- Function Body: Instructions, ...
- Function Epilogue 2: `ADD sp,sp,#16`
  - Reallocation of the space for local variables
- Function Epilogue 2: `POP {r4,r7,pc}`
  - The `POP` instruction pops the callee-save register `r4`, the saved frame pointer `r7`, and the return address off the stack which is loaded into the program counter `pc`
  - Hence, the execution will continue in the main function
Let’s go back to runtime attacks
```
#include <stdio.h>
void echo()
{
    char buffer[80];
    gets(buffer);
    puts(buffer);
}
int main()
{
    echo();
    printf("Done");
    return 0;
}
```
Launching a code injection attack against the vulnerable program
Code Injection Attack on ARM

Stack
- Return Address
- SFP & Other Regs.
- Local Buffer Buffer[80]

Program Memory
- Function Prologue
- BLX gets(buffer), ...
- Function Epilogue

Code
- <main>:
  - Instruction, ...
  - BLX echo()
  - Instruction, ...
  - BLX printf(), ...

- <echo>:
  - Function Prologue
  - BLX gets(buffer), ...
  - Function Epilogue

Adversary

Instruction, ...
BLX echo()
Instruction, ...
BLX printf(), ...
Return Address
SFP & Other Regs.
Code Injection Attack on ARM

Adversary

Corrupt Control Structures

Stack
- NEW RETURN ADDR
- PATTERN
- SHELLCODE

sp

Code
- Function Prologue
- Instruction, ...
- BLX echo()
- Instruction, ...
- BLX printf(), ...

Program Memory
- <main>:
  - Instruction, ...
  - BLX echo()
  - Instruction, ...
  - BLX printf(), ...
- <echo>:
  - Function Prologue
  - BLX gets(buffer), ...
  - Function Epilogue
Code Injection Attack on ARM

**Adversary**

1. **Instruction, ...**
2. **BLX echo()**
3. **Instruction, ...**
4. **BLX printf(), ...**

**<main>:**

- Function Prologue
- BLX gets(buffer), ...
- Function Epilogue

**<echo>:**

- NEW RETURN ADDR
- PATTERN
- SHELLCODE

**Program Memory**

**Stack**

- sp

**Code**

- Bash Shell
Code-Reuse Attacks
It started with return-into-libc


- Basic idea of return-into-libc
  - Redirect execution to functions in shared libraries
- Main target is UNIX C library libc
  - Libc is linked to nearly every Unix program
  - Defines system calls and other basic facilities such as open(), malloc(), printf(), system(), execve(), etc.
- Attack example: `system ("/bin/sh"), exit()`
Limitations

- No branching, i.e., no arbitrary code execution
- Critical functions can be eliminated or wrapped
Generalization of return-into-libc attacks: return-oriented programming (ROP) [Shacham, ACM CCS 2007]
The Big Picture

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous ProBlogger blog, announced the winners of his latest Group Writing Project called “Reviews and Predictions”. Among the winners, Daniel commented on the success of his blog that...
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 Address Space

Data Area

ROP Payload

Code Area

Application

Shared Libraries

MEMORY

Gadget Space (e.g., Shared Libraries)

MOV ESP LNOP ADD CALL XOR STORE
ROP Attack Technique: Overview

Corrupt Control Structures

Program Stack
- Return Address 3
- Value 2
- Value 1
- Return Address 2
- Return Address 1

SP

Program Code

Sequence 1
- asm_ins
- POP {PC}

Sequence 2
- POP REG1
- POP REG2
- POP {PC}

Sequence 3
- asm_ins
- POP {PC}

REG1:

REG2:
ROP Attack Technique: Overview

Program Stack

- Return Address 1
- Value 1
- Value 2
- Return Address 2
- Return Address 3

Program Code

Sequence 1
- asm_ins
- POP {PC}

Sequence 2
- POP REG1
- POP REG2
- POP {PC}

Sequence 3
- asm_ins
- POP {PC}

REG1:

REG2:
ROP Attack Technique: Overview

Program Stack:
- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2
- SP

Program Code:

Sequence 1:
- asm_ins
- POP {PC}

Sequence 2:
- POP REG1
- POP REG2
- POP {PC}

Sequence 3:
- asm_ins
- POP {PC}

REG1:

REG2:

Value 1
ROP Attack Technique: Overview

Program Stack:
- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2

Program Code:

Sequence 1:
- asm_ins
- POP {PC}

Sequence 2:
- POP REG1
- POP REG2
- POP {PC}

Sequence 3:
- asm_ins
- POP {PC}

REG1:
- Value 1

REG2:
- Value 2
ROP Attack Technique: Overview

Program Stack
- Return Address 3
- Value 2
- Value 1
- Return Address 2
- Return Address 1

Program Code
- Sequence 1
  - asm_ins
  - POP {PC}
- Sequence 2
  - POP REG1
  - POP REG2
  - POP {PC}
- Sequence 3
  - asm_ins
  - POP {PC}

SP
Summary of Basic Idea

- Perform arbitrary computation with return-into-libc techniques
- Approach
  - Use small instruction sequences (e.g., of libc) instead of using whole functions
  - Instruction sequences range from 2 to 5 instructions
  - All sequences end with a return (POP{PC}) instruction
  - Instruction sequences are chained together to a gadget
  - A gadget performs a particular task (e.g., load, store, xor, or branch)
  - Afterwards, the adversary enforces his desired actions by combining the gadgets
Special Aspects of ROP
Code Base and Turing-Completeness

Application Code

Shared Libraries

MOV reg1, 0x1 RET

MOV reg2, 0x2 RET

ADD reg1, reg2 RET

Static Analysis
Code Base and Turing-Completeness

Application Code

Shared Libraries

GADGET SPACE

MOV

CALL

Uncond. JMP

STORE

LOAD

Arith.

Logic.

Cond. JMP

Optional

Mandatory

Turing-complete language

Static Analysis
Gadget Space on Different Architectures

Architectures with no memory alignment, e.g., Intel x86

Intended Code
mov $0x13, %eax
jmp 3aae9

Unintended Code
add %al, (%eax)
add %ch, %cl
ret

Architectures with memory alignment, e.g., SPARC, ARM
Stack Pivot

[Zovi, RSA Conference 2010]

- Stack pointer plays an important role
  - It operates as an instruction pointer in ROP attacks
- Challenge
  - In order to launch a ROP exploit based on a heap overflow, we need to set the stack pointer to point to the heap
  - This is achieved by a stack pivot
Stack Pivot in Detail

*REG1 is controlled by the adversary and holds beginning of ROP payload
Stack Pivot in Detail

*REG1 is controlled by the adversary and holds beginning of ROP payload
Stack Pivot in Detail

Stack
- TOP of Stack

Heap
- Return Address 3
- Return Address 2
- Return Address 1
- label_pivot

Code
- label_pivot:
  - Stack Pivot
  - MOV SP, REG1*
  - POP {PC}

*REG1 is controlled by the adversary and holds beginning of ROP payload
ROP Variants

- Motivation: return address protection (shadow stack)
  - Validate every return (intended and unintended) against valid copies of return addresses
    [Davi et al., AsiaCCS 2011]
- Exploit indirect jumps and calls
  - ROP without returns
    [Checkoway et al., ACM CCS 2010]
CURRENT RESEARCH
1997
- ret2libc
  - Solar Designer

2001
- Advanced ret2libc
  - Nergal

2005
- Borrowed Code Chunk Exploitation
  - Krahmer

2007
- ROP on x86
  - Shacham (CCS)

2008
- ROP on SPARC
  - Buchanan et al (CCS)
- ROP on Atmel AVR
  - Francillon et al (CCS)

2009
- ROP Rootkits
  - Hund et al (USENIX)
- ROP on PowerPC
  - FX Lindner (BlackHat)
- ROP on ARM/iOS
  - Miller et al (BlackHat)

2010
- ROP without Returns
  - Checkoway et al (CCS)
- Practical ROP
  - Zovi (RSA Conference)
- Pwn2Own (iOS/IE)
  - Iozzo et al / Nils

2011/2012
- Real-World Exploits

2013
- JIT-ROP
  - Snow et al (IEEE S&P)
- Blind ROP
  - Bittau et al (IEEE S&P)
- Out-Of-Control
  - Göktas et al (IEEE S&P)

2014
- Stitching Gadgets
  - Davi et al (USENIX)
- Flushing Attacks
  - Schuster et al (RAID)
- ROP is Dangerous
  - Carlini et al (USENIX)
Our Work & Involvement

- **Attacks**
  - Return-Oriented Programming without Returns [CCS 2010]
  - Privilege Escalation Attacks on Android [ISC 2010]
  - Stitching the Gadgets [USENIX Security 2014] & [BlackHat USA 2014]
  - COOP [IEEE Security & Privacy 2015]
  - Losing Control [CCS 2015]

- **Detection & Prevention**
  - ROPdefender [AsiaCCS 2011]
  - Mobile Control-Flow Integrity (MoCFI) [NDSS 2012]
  - XIFER: Fine-Grained ASLR [AsiaCCS 2013]
  - Filtering ROP Payloads [RAID 2013]
  - Isomeron [NDSS 2015]
  - HAFIX: Fine-Grained CFI in Hardware [DAC 2014, DAC 2015, DAC 2016]
  - Readactor++ [CCS 2015]
Main Defense Techniques

(Fine-grained) Code Randomization

Control-Flow Integrity (CFI)
[Abadi et al., CCS 2005 & TISSEC 2009]

Memory

A

B

C

D

E

F

Exit(B) == Label_5
Main Defense Techniques

(Fine-grained) Code Randomization

Control-Flow Integrity (CFI)
[Abadi et al., CCS 2005 & TISSEC 2009]

Exit(B) == Label_5

Memory (randomized)
ASLR – Address Space Layout Randomization
Basics of Memory Randomization

- ASLR randomizes the base address of code/data segments

![Diagram showing memory segments and Brute-Force Attack mechanism.](image)

**Brute-Force Attack**

[Shacham et al., ACM CCS 2004]
Basics of Memory Randomization

- ASLR randomizes the base address of code/data segments

Disclosure Attack e.g., [Sotirov et al., Blackhat 2008]

1. Exploit disclosure vulnerability
2. Retrieve runtime ADDR address
3. Revert all library addresses based on ADDR
**Fine-Grained ASLR**

- **ORP** [Pappas et al., IEEE S&P 2012]: Instruction reordering/substitution within a BBL
- **ILR** [Hiser et al., IEEE S&P 2012]: Randomizing each instruction’s location
- **STIR** [Wartell et al., ACM CCS 2012] & **XIFER** [Davi et al., AsiaCCS 2013]: Permutation of BBLs
Does Fine-Grained ASLR Provide a Viable Defense in the Long Run?

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

IEEE Security and Privacy Best Student Paper 2013

Kevin Z. Snow (UNC Chapel Hill), Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose (UNC Chapel Hill), Ahmad-Reza Sadeghi
High-Level Idea

Scripting Engine

Code Pointer

Page Start

4KB

Page End

Disassemble

Code Page 1

INS_1
INS_2
INS_3
**JMP INS_10**
INS_4
INS_5
INS_6
High-Level Idea

Code Pointer

Code Page 1

<table>
<thead>
<tr>
<th>INS_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INS_2</td>
</tr>
<tr>
<td>INS_3</td>
</tr>
<tr>
<td><strong>JMP</strong> INS_10</td>
</tr>
<tr>
<td>INS_4</td>
</tr>
<tr>
<td>INS_5</td>
</tr>
<tr>
<td>INS_6</td>
</tr>
</tbody>
</table>

Code Page 2

<table>
<thead>
<tr>
<th>INS_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>INS_8</td>
</tr>
<tr>
<td>INS_9</td>
</tr>
<tr>
<td>INS_10</td>
</tr>
<tr>
<td>INS_11</td>
</tr>
<tr>
<td>INS_12</td>
</tr>
<tr>
<td>INS_13</td>
</tr>
</tbody>
</table>

Scripting Engine
Code Randomization: Lessons Learned

1. Memory disclosure attacks are far more damaging than previously believed
   → A single address-instruction mapping leads to many leaks of code pages

2. Fine-grained ASLR can be bypassed with JIT-ROP
   → Enforce execute-only memory
     Software-based [Backes et al., CCS 2014]
     Hardware-based: Readactor(++) [with Crane et al., IEEE S&P 2015 & CCS 2015]
   → Combine code- and execution randomization
     Isomeron [with Liebchen et al., NDSS 2015]
   → Mitigating memory disclosure
Control-Flow Integrity (CFI)
[Abadi et al., CCS 2005 & TISSEC 2009]

A general defense against code-reuse attacks

Exit(B) == Label_5

A -> B
B -> C
B -> D
C -> E
D -> F

Label_1
Label_2
Label_3
Label_4
Label_5
Label_6
Many CFI checks are required if unique labels are assigned per node.

Exit(B) == [Label_3, Label_4, Label_5]
Label Granularity: Trade-Offs (2/2)

- Optimization step: Merge labels to allow single CFI check
- However, this allows for unintended control-flow paths

Exit(B) == Label_3

Exit(C) == Label_3
Label Problem for Returns

- **Static CFI label checking** leads to coarse-grained protection for returns

Program Code

- **Function A**
  - CALL C
  - Code

- **Function B**
  - CALL C
  - Code

- **Function C**
  - Code
  - RETURN

**Exit(C) == [Label_1, Label_2]**
Shadow Stack / Return Address Stack

- **Shadow stack** allows for fine-grained return address protection but incurs higher overhead.

**Exit(C) == ShadowStack[TOS]**

**Shadow Stack**

*Backup storage for return addresses*

- **CALL**
- **RET**

**Backup**

**Check**

**Return Addr** [A']
CFI: Benefits and Limitations

- Fine-grained protection
- Blackbox Vulcan (unpublished)
- Require side info (debug symbols, compiler support)
- Performance overhead
Hot Research Topic:
“Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed:

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

Open Question:
Practical and secure mitigation of code reuse attacks

Turing-completeness of return-oriented programming
Negative Result:
All current (published) coarse-grained CFI solutions can be bypassed
Big Picture

- Systematic Security Analysis of Coarse-Grained CFI
- Gadget Analysis
- Exploit Development

- CFI Policies
- Frequency of CFI Checks
- Turing-complete gadget set
- Deriving a CFI policy that combines all schemes
- Gadgets to bypass heuristics

MPlayer

PDF icon
1. Systematic Security Analysis of Coarse-Grained CFI
Coarse-grained CFI leads to CFG imprecision

Allowed paths: 1→2 and 2→1

Reducing number of labels
Main Coarse-Grained CFI Policies

- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-preceded instruction
  - No shadow stack required

- **CFI Policy 2: Behavioral-Based Heuristics**

  **Threshold Setting**
  - kBouncer: \((N=8; S<=20)\)
  - ROPecker: \((N=11; S<=6)\)

Application

<table>
<thead>
<tr>
<th>CALL A</th>
<th>INS_1</th>
<th>INS_2</th>
<th>CALL B</th>
<th>INS_3</th>
<th>CALL C</th>
<th>INS_4</th>
</tr>
</thead>
</table>

1. **CALL A**
2. **INS_1**
3. **INS_2**
4. **CALL B**
5. **INS_3**
6. **CALL C**
7. **INS_4**
8. **RET**

1. **< S**
2. **< S**
3. **...**
4. **< S**
5. **> S**
6. **< S**
7. **< S**
8. **> S**
Coarse-Grained CFI Proposals

- kBouncer [USENIX Sec’13]
- ROPecker [NDSS’14]
- Paging
- Last Branch Record (LBR)

Win API / Critical Function

Binary Instrumentation

Application

CFI for COTS Binaries
[USENIX Sec’13]
CCFIR [IEEE S&P’13]

ROPGuard [Microsoft EMET]

Stack

POP
PUSH
Deriving a Combined CFI Policy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CFI Policy 1</strong>&lt;br&gt;Call-Preceded Sequences</td>
<td><img src="image1" alt="CFI Policy" /></td>
<td><img src="image2" alt="No Restriction" /></td>
<td><img src="image3" alt="CFI Policy" /></td>
<td><img src="image4" alt="CFI Policy" /></td>
<td><img src="image5" alt="CFI Policy" /></td>
</tr>
<tr>
<td><strong>CFI Policy 2</strong>&lt;br&gt;Behavioral-Based Heuristics</td>
<td><img src="image6" alt="CFI Policy" /></td>
<td><img src="image7" alt="CFI Policy" /></td>
<td><img src="image8" alt="No Restriction" /></td>
<td><img src="image9" alt="No Restriction" /></td>
<td><img src="image10" alt="CFI Policy" /></td>
</tr>
<tr>
<td>Time of CFI Check</td>
<td>WinAPI</td>
<td>2 Page Sliding Window/Critical Functions</td>
<td>WinAPI/Critical Functions</td>
<td>Indirect Branch</td>
<td>Any Time</td>
</tr>
</tbody>
</table>

Here only the core policies shown. However, we consider all other deployed policies in our analysis.
2. Gadget Analysis
Methodology

Common Library
kernel32.dll

Sequence Finder (IDA Pro)
List of Call-Preceded Sequences

Sequence Subset 1
Sequence Subset n

Sequence Filter (D Program)

Search for Gadgets
Provide filters on Reg, Ins, Opnd, Length

Gadget Generation (manual)
MOV, ESP, LNOP, LOAD, ADD, CALL, XOR, STORE
(Excerpt of) Turing-Complete Gadget Set in CFI-Protected `kernel32.dll`

<table>
<thead>
<tr>
<th>Gadget Type</th>
<th>CALL-Preceded Sequence</th>
</tr>
</thead>
</table>
| LOAD Register        | EBP := pop ebp  
                      | ESI := pop esi; pop ebp  
                      | EDI := pop edi; leave  
                      | ECX := pop ecx; leave  
                      | EBX := pop edi; pop esi; pop ebx; pop ebp  
                      | EAX := mov eax,edi; pop edi; leave  
                      | EDX := mov eax,[ebp-8]; mov edx,[ebp-4]; pop edi; leave                                                                                           |
| LOAD/STORE Memory    | LD(EAX) := mov eax,[ebp+8]; pop ebp  
                      | ST(EAX) := mov [esi],eax; xor eax,eax; pop esi; pop ebp  
                      | ST(ESI) := mov [ebp-20h],esi  
                      | ST(EDI) := mov [ebp-20h],edi                                                                                                                      |
| Arithmetic/Logical   | ADD/SUB := sub eax,esi; pop esi; pop ebp  
                      | XOR := xor eax,edi; pop edi; pop esi; pop ebp                                                                                                             |
| Branches             | unconditional branch 1 := leave  
                      | unconditional branch 2 := add esp,0Ch; pop ebp  
                      | conditional LD(EAX) := neg eax; sbb eax,eax; and eax,[ebp-4]; leave                                                                                  |
Long-NOP Gadget

ROP Gadget 1 → Store Registers → Prepare Long NOP → Long NOP → Reset Registers

- ESI
- EDI
- EBX

Stack

ROP Gadget 2 → ...

Static Constants

Arbitrary Data Area (36 Bytes)
3. Exploit Development

Adobe Reader 9.1
CVE-2010-0188

MPlayer Lite r33064 m3u
Buffer Overflow Exploit

Original exploits detected by coarse-grained CFI

Our instrumented exploits bypass coarse-grained CFI
Coarse-Grained CFI: Lessons Learned

1. Too many call sites available
   → Restrict returns to their actual caller (shadow stack)

2. Heuristics are ad-hoc and ineffective
   → Adjusted sequence length leads to high false positive

3. Too many indirect jump and call targets
   • Resolving indirect jumps and calls is non-trivial
   → Compromise: Compiler support
CURRENT RESEARCH
What’s next?

Hardware-Assisted CFI
HAFIX: Hardware Flow Integrity Extensions

Design Decisions: Why CFI Processor Support?

CFI Processor Support based on Instruction set architecture (ISA) extensions

- Dedicated CFI instructions
- No offline training phase
- Instant attack detection
- CFI control state
- Binding of CFI data to CFI state and instructions
**Big Picture**

**State 0**
*Normal Execution*

- Function Calls
- Indirect Jumps
- Function Returns

**CFI State**
*Only CFI instructions allowed*

- CFI Check Call
- CFI Check Jump
- CFI Check Return
Example Policy

*Returns can only target call sites of functions that are currently executing*
HAFIX State Model

State 0
Normal Execution
- Direct and Indirect Calls
- CFIDEL label_1
- Return

State 1
Function Entry
- CFIBR label_1

State 2
Function Exit
- CFIRET label_0

State 3
Attack Detection
- STOP Execution

Valid CFIBR issued
- Activate label
- Deactivate label

No CFIBR issued
- Check label

Valid CFIRET issued

No CFIRET issued or inactive label used
Remarks

- Implementation on Intel Siskiyou Peak and SPARC-LEON3
- High efficiency 1-2%
- Current prototype supports different levels of CFG precision [visit our DAC‘16 talk on Thursday, June 09, 3:30pm - 5:30pm | 19AB ]
Conclusion

- Code-reuse attacks are prevalent
  - Google and Microsoft take these attacks seriously
  - Many real-world exploits
  - Existing solutions can be bypassed

- Good News
  - Many innovative defense techniques have been proposed

- Promising new directions
  - Memory safety based on code-pointer integrity
    [Kuznetsov et al., OSDI 2014]
References
References (1/5)

• [Abadi et al., ACM CCS 2005 & ACM TISSEC 2009]
  M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti.
  Control-flow integrity: Principles, implementations, and applications.
• [Buchanan et al., ACM CCS 2008]
  E. Buchanan, R. Roemer, H. Shacham, and S. Savage.
  When good instructions go bad: Generalizing return-oriented programming to RISC.
• [Checkoway et al., EVT/WOTE 2009]
  S. Checkoway, A.J. Feldman, B. Kantor, J.A. Halderman, E.W. Felten, and H. Shacham.
  Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage.
• [Checkoway et al., ACM CCS 2010]
  Return-oriented programming without returns.
• [Cheng et al., NDSS 2014]
  ROPecker: A generic and practical approach for defending against ROP attacks.
• [Cohen, Computer & Security 1993]
  F. B. Cohen.
  Operating system protection through program evolution.
References (2/5)

- [Cowan et al., USENIX Security 1998]
  StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks.

- [Davi et al., ASIACCS 2013]
  Gadge me if you can - Secure and efficient ad-hoc instruction-level randomization for x86 and ARM.

- [Davi et al., ASIACCS 2011]
  ROPdefender: A detection tool to defend against return-oriented programming attacks.

- [Davi et al., USENIX Security 2014]
  Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection.

- [Davi et al., DAC 2014]
  Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation.
References (3/5)

- [Erlingsson, Technical Report 2007]
  Ú. Erlingsson.
  Low-level software security: Attacks and defenses.

- [Forrest et al., Hot Topics in Operating Systems 1997]
  S. Forrest, A. Somayaji, and D. Ackley.
  Building diverse computer systems.

- [Fratric, Technical Report 2012]
  I. Fratric.
  ROPGuard: Runtime prevention of return-oriented programming attacks.

- [Francillion et al., ACM CCS 2008]
  A. Francillon and C. Castelluccia.
  Code injection attacks on Harvard-architecture devices.

- [Hiser et al., IEEE Security & Privacy 2012]
  J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson.
  ILR: Where’d my gadgets go?.

- [Iozzo et al., Pwn2Own 2010]
  Ralf-Philipp Weinmann and Vincenzo Iozzo.
References (4/5)

• [Pappas et al., IEEE Security & Privacy 2012]
  V. Pappas, M. Polychronakis, and A. D. Keromytis.
  Smashing the gadgets: Hindering return-oriented programming using in-place code randomization.

• [Pappas et al., USENIX Security 2013]
  V. Pappas, M. Polychronakis, and A. D. Keromytis.
  Transparent ROP exploit mitigation using indirect branch tracing.

• [Shacham, ACM CCS 2004]
  H. Shacham.
  The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).

• [Shacham, ACM CCS 2007]
  H. Shacham, E. Jin Goh, N. Modadugu, B. Pfaff, and D. Boneh.
  On the effectiveness of address-space randomization.

• [Snow et al., IEEE Security & Privacy 2013]
  Just-in-time code reuse: On the effectiveness of fine-grained ASLR.
References (5/5)

- [Sotirov et al., BlackHat USA 2013]
  A. Sotirov and M. Dowd.
  Bypassing browser memory protections in Windows Vista.

- [Wartell et al., ACM CCS 2012]
  R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin.
  Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code.

- [Zhang et al., USENIX Security 2013]
  M. Zhang and R. Sekar.
  Control flow integrity for COTS binaries.

- [Zhang et al., IEEE Security & Privacy 2013]
  Practical control flow integrity & randomization for binary executables.

- [Zovi, RSA Conference 2010]
  D. D. Zovi.
  Practical return-oriented programming.