Runtime Attacks: Buffer Overflow and Return-Oriented Programming

Prof. Dr.-Ing. Ahmad-Reza Sadeghi
M.Sc. Lucas Davi

Course Secure, Trusted and Trustworthy Computing, Part 1
System Security Lab
http://trust.cased.de
Technische Universität Darmstadt

January 14, 2011
1 Introduction

2 Basics
   - Buffer Overflow (Stack Smashing)
   - Return-Into-Libc

3 Return-Oriented Programming
   - Introduction
   - Attack Technique
   - Countermeasures

4 Return-Oriented Programming Without Returns
   - Attack Technique
   - Countermeasures
Motivation: Runtime Attacks

- **Runtime attacks are major threats to today’s applications**
  - Control flow of an application is compromised at runtime
  - Typically, runtime attacks include *injection of malicious code*

- **Reasons for runtime attacks**
  - Software is written in unsafe languages such as C/C++
    ⇒ Thus, it suffers from *various memory-related vulnerabilities*

- **Most prominent example:** *Buffer overflow*
Motivation: Buffer Overflow

- Are known for 2 decades
- Various techniques exist
  - Stack Smashing
  - Heap Overflow
  - Integer Overflow
  - Format String
Countermeasures

- **W ⊕ X – Writable Xor Executable**
  - Prevents execution of injected code by marking memory pages either writable or executable
  - Implemented in Linux [PaXa] and Windows DEP (Data Execution Prevention) [Mic06]
  - Supported by chip manufacturers such as Intel and AMD (NX/XD Bit)

- **ASLR – Address Space Layout Randomization**
  - Randomizes base addresses of memory segments
  - Realized in Linux PaX Kernel Patch [PaXb]
  - Enabled for Windows Vista and Windows 7 [HT07]

- **Compiler Extensions**
  - Mitigate buffer overflows by introducing stack canaries, pointer encryption, bound checkers, variable reordering, etc.
Despite many countermeasures buffer overflows are still major threats of today’s applications
Buffer Overflow Vulnerabilities: Some Statistics

- Still a major threat (e.g., in Internet Explorer or Acrobat Reader, etc.)

![Bar chart showing buffer overflow vulnerabilities from 2006 to 2009.](image)

**Figure**: Buffer Overflows according to NIST Vulnerability Database
First observations

- Many applications are still suffering from buffer overflow vulnerabilities that allow code injection
- Modern systems enforce $W \oplus X$ to prevent code injection attacks

On the other hand new attack techniques bypass $W \oplus X$
Return-Oriented Programming

Arbitrary (Turing-complete) computation without the need to

- inject malicious code
- call any library function
- modify the original code
1. Introduction

2. Basics
   - Buffer Overflow (Stack Smashing)
   - Return-Into-Libc

3. Return-Oriented Programming
   - Introduction
   - Attack Technique
   - Countermeasures

4. Return-Oriented Programming Without Returns
   - Attack Technique
   - Countermeasures
Background and General Idea

- **Target of Buffer Overflow Attacks**
  - Subvert the usual execution flow of a program by redirecting it to a *injected (malicious) code*

- **The attack consists of**
  1. **injecting new (malicious) code** into some writable memory area,
  2. and **changing a code pointer** (usually the return address) in such a way that it points to the injected malicious code.

- **Code Injection**
  - Code can be injected by overflowing a local buffer allocated on the stack
  - The target of the injected code is usually to launch a shell to the adversary
  - Therefore the injected code is often referred to as **shellcode**
The Stack Frame

To understand how a buffer overflow attack works, we take a deeper look at the stack frame and its elements.
The Stack Frame (cntd.)

- Stack is a last in, first out (LIFO) memory area whereas the Stack Pointer (SP) points to the top word on the stack
- On the x86 architecture the stack grows downwards
- 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 (call instruction) sets up a new stack frame on top of the stack
  1. Function arguments
  2. Return address
    - Upon function return (i.e., a ret instruction is issued), control transfers to the code pointed to by the return address (i.e., control transfers back to the caller of the function)
  3. Saved Base Pointer
    - Base pointer of the calling function
    - Variables/arguments are accessed via an offset to the base pointer
  4. Local variables
Vulnerable program

- **Simple Echo program suffering from a stack overflow vulnerability**
- **The `gets()` function does not provide bounds checking**

```c
#include <stdio.h>
void echo()
{
    char buffer[80];
    gets(buffer);
    puts(buffer);
}
int main()
{
    echo();
    printf("Done");
    return 0;
}
```
(1) Program starts

```
<main>
...
call echo()
ins...
call printf()
```
(2) The echo() function is called

```
<main>
...  
call echo()
ins...
call printf()
```
(3) Call instruction pushes return address onto the stack
(4) Allocation of saved base pointer and buffer
(5) echo() calls gets(buffer) function
(6) Adversary transmits malicious code

```
<main>
... 
call echo()
ins...
call printf()
```
(7) Malicious code contains shellcode, pattern bytes, ...
(8) . . . , and a new return address

---

**Diagram: Stack and Code**

- **Stack**
  - Return Address
  - Pattern 2
  - Pattern 1
  - SHELLCODE

- **Code**
  
  ```
  <main>
  ...
  call echo()
  ins...
  call printf()
  ```

---

Sadeghi, Davi @TU Darmstadt 2010
Secure, Trusted and Trustworthy Computing, Part 1
Runtime Attacks 22 / 103
(9) Before `echo()` returns to `main`, SP is updated

```
<main>
    ...
    call echo()
    ins...
    call printf()
```

```
Stack
Return Address
Pattern 2
Pattern 1
SHELLCODE
```

```
Code
```

```
Libraries
```

Sadeghi, Davi @TU Darmstadt 2010
Secure, Trusted and Trustworthy Computing, Part 1
Runtime Attacks 23 / 103
(10) `echo()` issues return resulting in execution of shellcode

SHELLCODE launches shell to the attacker

```
<main>
...  
call echo()
ins...
call printf()
</main>
```
Conclusion and Limitations

- **Why the attack is possible?**
  - The `gets()` function provides **no bounds-checking**
  - C/C++ includes various functions providing **no bounds-checking**, e.g.,
    - `strcpy()`: Copies a string into a buffer
    - `strcat()`: Concatenates two strings
    - `scanf()`: Read data from stdin (Standard Input)

- **General defense against code injection attacks is** \( W \oplus X \)
  - With \( W \oplus X \) memory pages can be either marked writable or executable
  - Stack is marked writable
  - Hence, the adversary can only inject his malicious code, but cannot execute it
Return-into-Libc Attacks

- **Basic idea of return-into-libc**
  - Instead of injecting code use existing code
  - Subvert the usual execution flow by redirecting it to functions in linked system libraries
  - The process’s image consists of
    1. writable memory areas like stack and heap,
    2. and executable memory areas such as the code segment and the linked system libraries
  - The target for useful code can be found in the C library libc

- **The C library libc**
  - Libc is linked to nearly every Unix program
  - This library defines system calls and other basic facilities such as open(), malloc(), printf(), system(), execve(), etc.
  - E.g., system ("/bin/sh")

- The corresponding attack is referred to as return-into-libc attack
Useful Functions in Libc

- Libc provides the following useful functions to the adversary
  - The `system()` function
    - Executes a new program within a running program.
    - Example: `system(”/bin/sh”)`
    - This function executes the `/bin/sh` file (i.e., a new shell is launched)
  - The `execve()` function
    - Execute a new program and replace the (old) running program.
    - Example: `execve(argv[0], argv, NULL);`
      - `argv` is a string array, whereas `argv[0] = ”/bin/sh”`
      - This function launches a new shell and replaces the running program
Attack Example
(1) Adversary transmits malicious input

![Diagram](image.png)
(2) Input contains pattern bytes, ...
(3) ... a new return address pointing to system(), ...
(4) ... , a return address for system(), ...
(5) . . . , and a pointer to the /bin/sh string
(6) When `echo()` returns, `system()` launches a new shell.
Limitations

- **Return-into-libc attacks** bypass security mechanisms such as the $W + X$ model, but suffer from the following restrictions:
  1. The adversary relies on functions available in libc ⇒ The designers of libc could eliminate functions such as `system()`.
  2. The adversary can only invoke one function after the other ⇒ No branching is possible.
1 Introduction

2 Basics
   - Buffer Overflow (Stack Smashing)
   - Return-Into-Libc

3 Return-Oriented Programming
   - Introduction
   - Attack Technique
   - Countermeasures

4 Return-Oriented Programming Without Returns
   - Attack Technique
   - Countermeasures
The Big Picture
The Big Picture

The New York Times

Saturday, January 6, 2007

Daily Blog Tips awarded the

Last week Darren House, the famous author of the Daily Blog Tips, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the winners was Daniel, who wrote about his success in content creation.

Sadeghi, Davi @TU Darmstadt 2010
Secure, Trusted and Trustworthy Computing, Part 1
Runtime Attacks
The Big Picture
ROPO attacks are applicable on a broad range of architectures:

1. Intel x86 [Sha07]
2. The SPARC Machine [BRSS08]
3. Atmel AVR [FC08]
4. Z80 Voting Machines [CFK+09]
5. PowerPC [Lin09]
6. ARM [Kor09]
Real-World Exploits

- **Apple iPhone**
  - JailbreakMe [Hal10]
  - Steal SMS Database [IW10]

- **Desktop PCs**
  - Acrobat Reader [jdu10]
  - Adobe Flashplayer [Ado10]

- **Special-purpose machines**
  - Z80 voting machine [CFK⁺09]
Jailbreak on Apple iPhone
(1) Download special crafted PDF file

http://www.jailbreakme.com/_/iPhone3%2c1_4.0.pdf

Send PDF with embedded ROP Payload
(2) ROP attack is launched

http://www.jailbreakme.com/_/iPhone3%2c1_4.0.pdf
Send PDF with embedded ROP Payload

JailbreakMe.com
(3) Download new system files

http://www.jailbreakme.com/_/iPhone3%2c1_4.0.pdf
Send PDF with embedded ROP Payload

JailbreakMe.com
(4) Jailbreak completed

http://www.jailbreakme.com/_/iPhone3%2c1_4.0.pdf
Send PDF with embedded ROP Payload

Sadeghi, Davi ©TU Darmstadt 2010
Secure, Trusted and Trustworthy Computing, Part 1
Runtime Attacks 46 / 103
Stealing Votes with ROP

- Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage

[CFK+09] http://www.youtube.com/watch?v=lsfG3KPrD1I

Sadeghi, Davi @TU Darmstadt 2010

Secure, Trusted and Trustworthy Computing, Part 1

Runtime Attacks

47 / 103
ROP Attack on Adobe Reader

- $W \oplus X$: Data Execution Prevention (DEP)
  - Adobe Reader enables DEP by default

- CVE-2010-0188
  - Integer Overflow Vulnerability in the libtiff library of Adobe Reader
  - Use a malicious TIFF image (embedded in a PDF file) to exploit the vulnerability
  - However, Adobe Reader enables DEP by default

- Attack
  1. Create a malicious PDF file containing (1) ROP code and (2) arbitrary shellcode
  2. When the user opens the file, the malicious PDF first exploits the integer vulnerability
  3. Afterwards, ROP is used to exploit $W \oplus X$ to allocate a memory page marked as writable ($W$) and executable ($X$)
  4. Finally the shellcode is copied to that memory page (by means of ROP) and executed.
How does ROP actually work?
General Idea of ROP

- **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 *ret* 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*
Relation of Instruction Sequences and Gadgets

- **Instruction sequence**
  - A sequence of instructions ending in a `ret` instruction (return)

- **Gadget**
  - Consists of several instruction sequences

![Diagram of Instruction Sequences and Gadgets]

---

Sadeghi, Davi ©TU Darmstadt 2010 Secure, Trusted and Trustworthy Computing, Part 1 Runtime Attacks 51 / 103
Attack Example
(1) Program is waiting for input from the user
(2) Adversary overflows the buffer
(3) Input contains return addresses and one argument
(4) foo() returns and first sequence is executed
(5) Return instruction transfers control to next sequence

Stack
- Return Address 6
- Return Address 5
- Return Address 4
- Return Address 3
- Return Address 2
- Return Address 1
- Pattern 2
- Pattern 1

Gadget 1 (e.g., Load)
- ins1
- ins2
- ins2
- ins2
- ins2
- ret
- pop
- ret
- ret

Gadget 2 (e.g., ADD)
- ins1
- ins2
- ins2
- ins2
- ret
- ins3
- ins4
- ret
(6) Return of Sequence 2 transfers control to Sequence 3
(7) Pop Argument off the stack
(8) Return instruction of Sequence 3 has been reached
(9) Return of Sequence 3 transfers control to Sequence 4
(10) Return of Sequence 4 transfers control to Gadget 2
(11) Return of Sequence 1 transfers control to Sequence 2
Unintended Instruction Sequences

- **Unintended instruction sequences**
  - A sequence of instructions ending in a `ret` instruction that was never intended by the programmer
  - These sequences can be found by jumping in the middle of a valid instruction resulting in a new unintended instruction sequence

- **Unintended instruction sequences can be found for the x86 architecture for two reasons**
  - **Variable-length instructions**: Instructions are not of fixed size
  - **Unaligned memory access**: If the native machine word is of size $N$ then an unaligned memory access means reading from an address that is not divisible by $N$. 
## Find Unintended Instruction Sequences

- **Consider the following instructions contained in libc**

<table>
<thead>
<tr>
<th>Byte values</th>
<th>Assembler</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>b8 13 00 00 00</td>
<td>mov $0x13,%eax</td>
<td>/* move 0x13 to the %eax register */</td>
</tr>
<tr>
<td>e9 c3 f8 ff ff</td>
<td>jmp 3aae9</td>
<td>/* jump to (relative) address 3aae9 */</td>
</tr>
</tbody>
</table>

- **Instead of starting the interpretation of the byte stream at b8, starting at the third byte 00 results in following unintended instruction sequence**

<table>
<thead>
<tr>
<th>Byte values</th>
<th>Assembler</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>add %al,(%eax)</td>
<td>/* add register value of %al to the word */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/* pointed to by the %eax register */</td>
</tr>
<tr>
<td>00 e9 c3</td>
<td>add %ch,%cl</td>
<td>/* add registers %cl and %ch */</td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td>/* return instruction */</td>
</tr>
</tbody>
</table>
Gadget Example: Memory Load
(1) Sequence 1 starts execution

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register

![Diagram](Image)
(2) Pop 0x8010AB8D in register %eax

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register.
(3) Return instruction transfers control to Sequence 2

**Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register

```
0x8010ABCD
Memory LOAD Gadget
movl 64(%eax),%eax
ret
ret

0xDEADBEEF
Value of %eax
8 0 1 0 A B 8 D
```

```
Stack
SP
Return Address 2
0x8010AB8D
Return Address 1
Pattern 2
Pattern 1
```

Sadeghi, Davi @TU Darmstadt 2010
Secure, Trusted and Trustworthy Computing, Part 1
Runtime Attacks 69 / 103
(4) Move 0xDEADBEEF in register %eax

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register

![Diagram showing the movement of 0xDEADBEEF into register %eax](image)
How to protect return addresses from malicious modification?
Compiler Based Solutions

- **Selected Approaches**
  - Place a **canary** before the return address
  - Backup return addresses onto a separate **shadow stack**

- **Realizations**
  1. Examples for canary based solutions
     - StackGuard [CPM+98]
     - ProPolice [Hir]
  2. Examples for shadow stack based solutions
     - Return Address Defender [CH01]
     - Stack Shield [Ven]

- **Limitations and disadvantages**
  - Compiler solutions require access to source code
  - Recompilation
  - In general, not able to detect unintended instruction sequences
Shadow Stack Approach

- Instruction
  - 1
  - 2a
  - 3a
  - 3b
  - Is Call?
  - 2b
- Is Return?
- Fetch next Instruction
- Push TOS onto Shadow Stack
- Compare TOS of both Stacks
- Program Stack
  - Return 4
  - Return 3
  - Return 2
  - Return 1
- Shadow Stack
  - Saved Return 4
  - Saved Return 3
  - Saved Return 2
  - Saved Return 1

Sadeghi, Davi @TU Darmstadt 2010
Secure, Trusted and Trustworthy Computing, Part 1
Runtime Attacks 73 / 103
Hardware Facilitated Solutions

- **Approach**
  - Use existing hardware features or new hardware modules to enforce return address protection

- **Realizations**
  - Embedded microprocessor [FPC09]
    - Split the stack into data-only and call/return addresses-only parts
    - Enforce access control on call/return stack
  - StackGhost [FS01]
    - Stack Cookies XORed against return addresses
    - Solution specific to SPARC

- **Limitation**
  - Require new hardware features [FPC09] or are based on unique features of a special system [FS01]
Dynamic Binary Instrumentation based on a JIT-Compiler

- **Approach**
  - Add *instrumentation code* by compiling an instruction block to new instructions at runtime (*JIT – Just In Time Compilation*).
  - JIT-based instrumentation allows the detection of unintended sequences.

- **Realizations**
  - Program Shepherding [KBA02]
    - Checks if a return targets a valid call site, i.e., a return has to target an instruction which is preceded by a call instruction.
  - ROPdefender [DSW10]
    - Checks each return address against valid return addresses hold in a separate shadow stack.
  - Measure return frequency: DynIMA [DSW09], DROP [CXS⁺09]

- **Limitations**
  - JIT-based instrumentation adds high performance overhead.
  - Solutions based on measuring the frequency of returns can be bypassed by executing longer sequences.
1 Introduction

2 Basics
   - Buffer Overflow (Stack Smashing)
   - Return-Into-Libc

3 Return-Oriented Programming
   - Introduction
   - Attack Technique
   - Countermeasures

4 Return-Oriented Programming Without Returns
   - Attack Technique
   - Countermeasures
Is it possible to bypass return address checkers?
Return-Oriented Programming without Returns [CDD+10]
ROP without Returns

- **Results**
  - Countermeasures that protect return addresses are bypassed
  - Attack technique for **Intel x86** and **ARM**
  - **Turing-complete gadget set** and practical attack instantiation for both platforms **without** any return instruction

- **Approach**
  - Use **return-like sequences**
  - Candidates are **indirect jumps**
    - On Intel: jmp *%eax
    - On ARM: blx r3

- **Obstacles**
  - Target register (%eax, r3) must be initialized before
  - Returns automatically update the stack pointer; indirect jumps not
Return-Like Sequences

- **On Intel**
  - `pop %eax; jmp *%eax`
  1. Pop target address into %eax
  2. The `pop` instruction automatically increases the stack pointer by four bytes (similar to a return)
  3. Jump to the address stored in %eax

- **On ARM**
  - No pop-jump sequence present
  - Use **Update-Load-Branch** Sequence
    1. (Update) – `adds r6,#4`: Add four bytes to r6
    2. (Load) – `ldr r5, [r6]`: Load target address into r5
    3. (Branch) – `blx r5`: Branch to target address

- **Problem**
  - Return-like sequences for both platforms are rare
Trampoline

Solution

- Use a unique Update-Load-Branch (ULB) sequence after each instruction sequence
- ULB is used as a trampoline
- All other sequences have to end in an indirect jump to ULB
Attack Example
(1) Adversary launches a buffer overflow
(2a) reg1 is initialized with the address of the trampoline
(2b) Jump Address 1 points to Sequence 1
(3) Sequence 1 is executed
(4) Jump to Trampoline enforced

**Stack**
- Jump Address 3
- Jump Address 2
- Jump Address 1
- Buffer Overflow

**Value of reg1**
- Trampoline Address

**Value of reg2**

**Libraries**
- ins1
- ins2
- ins3
- ins4
- jmp *reg1

**Gadget**
- Update SP
- Load reg2
- Branch: jmp *reg2

**Update−Load−Branch (Trampoline)**
(5) Stack pointer is updated

- **Stack**
  - Jump Address 3
  - Jump Address 2
  - Jump Address 1
  - Buffer Overflow

- **Value of reg1**
  - Trampoline Address

- **Value of reg2**

- **Libraries**
  - ins1
  - ins2
  - ins3
  - ins4
  - jmp *reg1

- **Gadget**
  - Update SP
  - Load reg2
  - Branch: jmp *reg2
  - Update–Load–Branch (Trampoline)
(6) Jump Address 2 is loaded in register reg2
(7) Branch to Sequence 2 is enforced

```
STACK
Jump Address 3
Jump Address 2
Jump Address 1
Buffer Overflow

LIBRARIES
ins1 ins1 ins1
ins2 ins2 ins2
ins3 jmp *reg1 ins3
ins4 jmp *reg1

GADGET
Update SP
Load reg2
Branch: jmp *reg2

UPDATE-LOAD-BRANCH (TRAMPOLINE)
```

Sadeghi, Davi @ TU Darmstadt 2010
(8) Jump to Trampoline is enforced

- **Stack**
  - Jump Address 3
  - Jump Address 2
  - Jump Address 1
  - Buffer Overflow

- **Value of reg1**
  - Trampoline Address

- **Value of reg2**
  - Jump Address 2

- **Update SP**
- **Load reg2**
- **Branch:** `jmp *reg2`

- **Gadget**
- **Libraries**
  - `ins1`, `ins2`, `ins3`, `ins4`, `jmp *reg1`
  - `jmp *reg1`

- **Sadeghi, Davi @TU Darmstadt 2010**
- **Secure, Trusted and Trustworthy Computing, Part 1**
- **Runtime Attacks**
(9) Stack Pointer is updated
(10) Jump Address 3 is loaded in register reg2
(11) Branch to Sequence 3 is enforced
**Attack instantiation**

- **Start the ROP attack**
  - Goal: Get control of the stack pointer and the instruction pointer
    - Usually stack smashing is used for conventional ROP
    - However, we want to avoid the use of any return instruction
  - Several techniques are described in [CDD+10]

- **Example: Setjmp Buffer Overwrite**
  - `setjmp()`/`longjmp()` are system calls to allow non-local gotos
    1. `setjmp()`: Store current stack frame and processor registers in a special buffer (the setjmp buffer)
    2. `longjmp()`: Return to saved stack frame and reset processor registers to the values stored in the setjmp buffer

- **Setjmp Buffer Overwrite**
  - A buffer is allocated before the setjmp buffer
  - Overflow the buffer with ROP payload and overwrite contents of the setjmp buffer
  - When `longjmp()` is called the ROP code is executed
Countermeasures

- **Control Flow Integrity (CFI) [ABEL05, ABE+06]**
  - Derives a control flow graph from a given binary
  - Labels all branch targets with a special instruction (a label ID)
  - Rewrites the binary to include new instructions that check at runtime if an indirect branch (return, jump, call) targets a valid label ID

- **Limitations of CFI**
  - Requires debugging information stored in Windows PDB files
  - CFI is built on top of the dynamic binary instrumentation framework Vulcan which is not publicly available
Address Space Layout Randomization (ASLR)

- **Approach**
  - Randomizes the base address of each segment (stack, heap, libraries, etc.)
  - Thus, an attacker does not know the start addresses of instruction sequences

- **Realizations**
  - Linux PaX Kernel Patch [PaXb]
  - Available for Windows since MS Vista [HT07]

- **Limitations**
  - Parts of the code are not randomized, allowing an attacker to construct some gadgets
    - [RMPB09]: Overwrite GOT (Global Offset Table) entries with new values.
  - Information leakage and brute-force attacks possible
    - E.g., see [SjGM+04, SD08]
G-Free: Gadget-Less Binaries

- **G-Free [OBL+10]: Technique and Approach**
  - Compiler-based approach to defeat ROP through gadget-less binaries
  - Requires recompilation
  - Possible unintended instruction sequences are eliminated through code transformations
  - Protection of intended return instructions
    - Return addresses are encrypted against a random cookie
  - Protection of intended jump and call instructions
    - Upon function entry, a function-unique cookie (function identifier xor random key) is stored on the stack
    - All indirect jumps/calls are extended with a validation block
    - The indirect jump/call is only allowed if the validation block successfully decrypts the cookie


References II


References III


References


