Out Of Control: Overcoming Control-Flow Integrity

Enes Gökta  
Vrije Universiteit Amsterdam  

Herbert Bos  
Vrije Universiteit Amsterdam

Elias Athanasopoulos  
FORTH-ICS

Georgios Portokalidis  
Stevens Institute of Technology

Computer Science Department

April 8, 2019
CFI has yet to be deployed for two reasons.
Problem

CFI has yet to be deployed for two reasons

1. CFI implementations require
Introduction

Problem

CFI has yet to be deployed for two reasons

1. CFI implementations require source code
Introduction

Problem

CFI has yet to be deployed for two reasons

1. CFI implementations require
   a. source code
   b. debug information that is not available in commercial products
Introduction

Problem

CFI has yet to be deployed for two reasons

1. CFI implementations require
   1. source code
   2. debug information that is not available in commercial products

2. CFI is quite expensive with high overheads
Contributions of this paper

1. Evaluate state-of-art CFI techniques and show why they don’t protect against ROP attacks
Contributions of this paper

1. Evaluate state-of-art CFI techniques and show why they don’t protect against ROP attacks
2. Develop a methodology for performing code-reuse attacks against CFI
Contributions of this paper

1. Evaluate state-of-art CFI techniques and show why they don’t protect against ROP attacks
2. Develop a methodology for performing code-reuse attacks against CFI
3. Construct a working exploit for IE 8 with DEP and ASLR on
Address Space Layout Randomization - ASLR
Address Space Layout Randomization - ASLR
Address Space Layout Randomization - ASLR

- Memory-protection process for operating systems
Address Space Layout Randomization - ASLR

- Memory-protection process for operating systems
- Guards against buffer-overflow attacks
Address Space Layout Randomization - ASLR

- Memory-protection process for operating systems
- Guards against buffer-overflow attacks

How?
Address Space Layout Randomization - ASLR

- Memory-protection process for operating systems
- Guards against buffer-overflow attacks

How?
- Randomizing the location where system executables are loaded
Stack Smashing Protector - SSP

How?
Aborts the execution if a secret value on stack is changed.
Stack Smashing Protector - SSP
Stack Smashing Protector - SSP

- Compiler feature
Stack Smashing Protector - SSP

- Compiler feature
- Helps the detection for stack buffer overruns
Stack Smashing Protector - SSP

- Compiler feature
- Helps the detection for stack buffer overruns

How?
**Stack Smashing Protector - SSP**

- Compiler feature
- Helps the detection for stack buffer overruns
  - How?
  - Aborts the execution if a secret value on stack is changed
Data Execution Prevention - DEP
Data Execution Prevention - DEP
Data Execution Prevention - DEP

- Set of hardware and software technologies
Data Execution Prevention - DEP

- Set of hardware and software technologies
- Perform additional checks memory
Data Execution Prevention - DEP

- Set of hardware and software technologies
- Perform additional checks memory

Why?
Data Execution Prevention - DEP

- Set of hardware and software technologies
- Perform additional checks memory
- Prevent malicious code from running on a system
Gadget
Gadget

Small group of instructions ending with a **RET** instruction.
For example, `mov eax, 10`
Return Oriented Programming - ROP

How ROP works:

- Gains control of the call stack to hijack control flow and execute carefully chosen machine instructions known as gadgets.
Return Oriented Programming - ROP

- Computer security exploit
Return Oriented Programming - ROP

- Computer security exploit
- Allows an attacker to execute malicious code in the presence of security defenses
Return Oriented Programming - ROP

- Computer security exploit
- Allows an attacker to execute malicious code in the presence of security defenses

HOW?

- Gains control of the call stack to hijack control flow and execute carefully chosen machine instructions known as gadgets
Return Oriented Programming - ROP

- Computer security exploit
- Allows an attacker to execute malicious code in the presence of security defenses

**HOW?**
Gains control of the **call stack** to hijack control flow and execute **carefully chosen** machine instructions known as **gadgets**
Control-flow attacks operate by exploiting a memory corruption bug. Such attacks lead to code-reuse and code-injection attacks. CFI prevents such attack by:

1. Ensuring that the control flow remains within \textbf{CFG}
2. Checking that an indirect’s jump target matches with the allowable target \textbf{ID}
Practical CFI

Improve CFI performance by reducing the number of **IDs**

```c
bool less_than(int x, int y); bool greater_than(int x, int y);
bool sort(int a[], int len, comp_func_fptr)
{
    ... if (fptr(a[i], a[i+i]))
    ... ...
} void sort_1(int a[], int len)
{
    ... sort(a, len, less_than);
} void sort_2(int a[], int len)
{
    ... sort(a, len, greater_than);
}

Fig. 1: An example program and its CFG. Conceptually, CFI introduces labels and checks for all indirect transfers. Control-flow transfers checked by CFI are shown in solid lines.
Weaknesses

- CFI restricts control flow transfers based on a finite CFG.
Weaknesses

1. CFI restricts control flow transfers based on a **finite** CFG
2. Limiting the number of IDs as mentioned before, reduces accuracy
Traditional Code-reuse Attacks

- ROP exploitation is based on controlling the stack

1. Corrupt program stack
2. Control the return address of a function
3. When the function returns, jump to specific gadgets specified by the attacker
4. Attacker can chain several gadgets to achieve his goal
5. He can put data for his purposes between gadgets addresses
6. PROFIT
Traditional Code-reuse Attacks

- ROP exploitation is based on controlling the stack
- HOW?
Traditional Code-reuse Attacks

- ROP exploitation is based on controlling the stack
  HOW?
  1. Corrupt program stack
Traditional Code-reuse Attacks

- ROP exploitation is based on controlling the stack
- **HOW?**
  1. Corrupt program stack
  2. Control the return address of a function
Traditional Code-reuse Attacks

- ROP exploitation is based on controlling the stack

**HOW?**

1. Corrupt program stack
2. Control the return address of a function
3. When the function returns, jump to specific **gadgets** specified by the attacker

**PROFIT**
Traditional Code-reuse Attacks

- ROP exploitation is based on controlling the stack

**HOW?**

1. Corrupt program stack
2. Control the return address of a function
3. When the function returns, jump to specific **gadgets** specified by the attacker
4. Attacker can chain several gadgets to achieve his goal
## Traditional Code-reuse Attacks

ROP exploitation is based on controlling the stack

**HOW?**

1. Corrupt program stack
2. Control the return address of a function
3. When the function returns, jump to specific **gadgets** specified by the attacker
4. Attacker can chain several gadgets to achieve his goal
5. He can put data for his purposes between gadgets addresses
Traditional Code-reuse Attacks

ROP exploitation is based on controlling the stack

HOW?

1. Corrupt program stack
2. Control the return address of a function
3. When the function returns, jump to specific gadgets specified by the attacker
4. Attacker can chain several gadgets to achieve his goal
5. He can put data for his purposes between gadgets addresses
6. PROFIT ✓
a single gadget can be manipulated to perform different actions by controlling which path it is actually taking

Gadgets in CFI

Two type of Gadgets
a single gadget can be manipulated to perform different actions by controlling which path it is actually taking

<table>
<thead>
<tr>
<th>Gadgets in CFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two type of Gadgets</td>
</tr>
<tr>
<td>![ ] Call-site(CS)</td>
</tr>
</tbody>
</table>
A single gadget can be manipulated to perform different actions by controlling which path it is actually taking.

**Gadgets in CFI**

<table>
<thead>
<tr>
<th>Two type of Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call-site (CS)</strong></td>
</tr>
<tr>
<td>- blocks of instructions right after a call instruction</td>
</tr>
</tbody>
</table>
a single gadget can be manipulated to perform different actions by controlling which path it is actually taking

Gadgets in CFI

Two type of Gadgets

Call-site(CS)
  - blocks of instructions right after a call instruction
  - terminate with a return instruction
a single gadget can be manipulated to perform different actions by controlling which path it is actually taking

<table>
<thead>
<tr>
<th>Gadgets in CFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two type of Gadgets</td>
</tr>
<tr>
<td>1. <strong>Call-site(CS)</strong></td>
</tr>
<tr>
<td>- blocks of instructions right after a call instruction</td>
</tr>
<tr>
<td>- terminate with a return instruction</td>
</tr>
<tr>
<td>2. <strong>Entry-point(EP)</strong></td>
</tr>
</tbody>
</table>
a single gadget can be manipulated to perform different actions by controlling which path it is actually taking

## Gadgets in CFI

Two type of Gadgets

1. **Call-site (CS)**
   - blocks of instructions right after a call instruction
   - terminate with a return instruction

2. **Entry-point (EP)**
   - Start at function entry point
a single gadget can be manipulated to perform different actions by controlling which path it is actually taking

<table>
<thead>
<tr>
<th>Gadgets in CFI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two type of Gadgets</strong></td>
</tr>
<tr>
<td><strong>1 Call-site(CS)</strong></td>
</tr>
<tr>
<td>- blocks of instructions right after a call instruction</td>
</tr>
<tr>
<td>- terminate with a return instruction</td>
</tr>
<tr>
<td><strong>2 Entry-point(EP)</strong></td>
</tr>
<tr>
<td>- Start at function entry point</td>
</tr>
<tr>
<td>- End with an indirect call or jump</td>
</tr>
</tbody>
</table>
ASLR makes attackers life hard.
They perform a two-stage attacks
ASLR makes attackers life hard. They perform a two-stage attacks

- learn information regarding the layout of the program
**Locating the Gadgets**

**ASLR** makes attackers life hard. They perform a two-stage attacks

1. learn information regarding the layout of the program
2. use the information to fix the location of their gadgets
Indirect calls
## Calling Functions

1. **Indirect calls**
2. **Call through Gadgets**
From Code-reuse to Code-injection

Code transformation from attack reuse to injection will grant the attacker infinite freedom

**HOW?**

- use gadgets to call an API or system call
### From Code-reuse to Code-injection

Code transformation from attack reuse to injection will grant the attacker infinite freedom.

**HOW?**
- use gadgets to call an API or system call

**WHY?**
From Code-reuse to Code-injection

Code transformation from attack reuse to injection will grant the attacker infinite freedom.

**HOW?**

1. use gadgets to call an API or system call
2. to alter the **execute bit** on an attacker controlled buffer
From Code-reuse to Code-injection

Code transformation from attack reuse to injection will grant the attacker infinite freedom

**HOW?**

1. use gadgets to call an API or system call
2. to alter the **execute bit** on an attacker controlled buffer
3. overwrite the code with our shellcode, and finally transfer control to it
Exploit

- bypasses CFI strongest version
Proof-Of-Concept Exploitation

Exploit

- bypasses CFI strongest version
- based on real heap overflow of IE which gives control to an indirect jump
Exploit

- bypasses CFI strongest version
- based on real heap overflow of IE which gives control to an indirect jump
- vulnerability triggered by *span* and *width*
Exploit

- bypasses CFI strongest version
- based on real heap overflow of IE which gives control to an indirect jump
- vulnerability triggered by *span* and *width*
- overwrites a virtual function table pointer in a button object
Proof-Of-Concept Exploitation

<table>
<thead>
<tr>
<th>Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>- bypasses CFI strongest version</td>
</tr>
<tr>
<td>- based on real heap overflow of IE which gives control to an indirect jump</td>
</tr>
<tr>
<td>- vulnerability triggered by <em>span</em> and <em>width</em></td>
</tr>
<tr>
<td>- overwrites a virtual function table pointer in a button object</td>
</tr>
<tr>
<td>- executes the malicious code</td>
</tr>
</tbody>
</table>
Evaluation

Fig. 8: Frequency of gadgets *without* branches in IE9 based on their length (instruction count).

Fig. 9: Frequency of gadgets *including* paths with branches in IE9 based on path length (instruction count).
Thank You all!
Questions?