Preventing control-flow hijacks with Code Pointer Integrity

László Szekeres
Stony Brook University

Joint work with Volodymyr Kuznetsov, Mathias Payer, George Candea, R. Sekar, Dawn Song
Problem

• C/C++ is unsafe and unavoidable today
• All of our systems have C/C++ parts
• All of them have exploitable vulnerabilities
• They all can be compromised
Control-flow hijack attack

Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer...

... to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

Execute injected code

Control-flow hijack

[Eternal War in Memory, IEEE S&P ‘13]
Control-flow hijack defenses

Make pointer out-of-bounds
Make pointer dangling

Use pointer to write
Use pointer to read

Modify a code pointer...
... to target code address

Use pointer by indir. call/jmp
Use pointer by ret instruction

Exec. gadgets or functions
Execute injected code

Control-flow hijack

[Eternal War in Memory, IEEE S&P ‘13]
Control-flow hijack defenses

[ Eternal War in Memory, IEEE S&P ‘13 ]

- Make pointer out-of-bounds
- Make pointer dangling
- Use pointer to write
- Use pointer to read
- Modify a code pointer...
  ... to target code address
- Use pointer by indir. call/jmp
- Use pointer by ret instruction
- Exec. gadgets or functions
- Execute injected code

Control-flow hijack

DEP
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]

Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer...

... to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

Execute injected code

Control-flow hijack

Cookies

DEP
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]

1. Make pointer out-of-bounds
2. Make pointer dangling
3. Use pointer to write
4. Use pointer to read
5. Modify a code pointer...
6. ... to target code address
7. Use pointer by indir. call/jmp
8. Use pointer by ret instruction
9. Exec. gadgets or functions
10. Execute injected code

Control-flow hijack
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]

- Make pointer out-of-bounds
- Make pointer dangling
- Use pointer to write
- Use pointer to read
- Modify a code pointer...
  - ... to target code address
  - Use pointer by indir. call/jmp
  - Use pointer by ret instruction
  - Exec. gadgets or functions
  - Execute injected code

- ASLR
- CFI
- DEP

Control-flow hijack
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]

Make pointer out-of-bounds
Make pointer dangling

Use pointer to write
Use pointer to read

Modify a code pointer...
... to target code address

Use pointer by indir. call/jmp
Use pointer by ret instruction

Exec. gadgets or functions
Execute injected code

Control-flow hijack

ASLR
CoCFL
DEP
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]

- Make pointer out-of-bounds
- Make pointer dangling
- Use pointer to write
- Use pointer to read
- Modify a code pointer...
  ... to target code address
  - Use pointer by indir. call/jmp
  - Use pointer by ret instruction
  - Exec. gadgets or functions
  - Execute injected code

Memory Safety
- ASLR
- CoCFL
- DEP

Control-flow hijack

Can be bypassed
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]

Make pointer out-of-bounds
Make pointer dangling

Use pointer to write
Use pointer to read

Modify a code pointer...
... to target code address

Use pointer by indir. call/jmp
Use pointer by ret instruction

Exec. gadgets or functions
Execute injected code

Control-flow hijack

Memory Safety

ASLR
CoCFI
DEP

2-4x slower
Can be bypassed
Control-flow hijack defenses

[ Eternal War in Memory, IEEE S&P ‘13 ]

- Make pointer out-of-bounds
- Make pointer dangling
- Use pointer to write
- Use pointer to read
- Modify a code pointer...
- ... to target code address
- Use pointer by indir. call/jmp
- Use pointer by ret instruction
- Exec. gadgets or functions
- Execute injected code

Memory Safety

Code Pointer Integrity

ASLR

CoCFI

DEP

Can be bypassed

2-4x slower
Code Pointer Integrity?

[ Eternal War in Memory, IEEE S&P ‘13 ]

- Make pointer out-of-bounds
- Make pointer dangling

- Use pointer to write
- Use pointer to read

- Modify a code pointer...
- ... to target code address

- Use pointer by indir. call/jmp
- Use pointer by ret instruction

- Exec. gadgets or functions
- Execute injected code

- Control-flow hijack
Code Pointer Integrity

[OSDI ’14]

• Joint work with Volodymyr Kuznetsov, Mathias Payer, George Canda, R. Sekar, Dawn Song

• It prevents all control-flow hijacks

• It has only 8% runtime overhead in average
Outline
Outline

Safe Stack
Outline

- Code Pointer Separation
- Safe Stack
Outline

- Code Pointer Integrity
- Code Pointer Separation
- Safe Stack
Safe Stack

Enforcing the integrity of return addresses
Integrity of return addresses

Stack

- int i (local variable)
- saved %ebp (base pointer)
- saved %eip (ret. address.)
- func call argument
- char buff[16]
- ...
- ...

...
Integrity of return addresses

Stack

... char buff [16] (local variable)
(save eip)
(save ebp)
(func call argument)
...

Integrity of return addresses

Stack

... 

char buff[16] 

int i 
(local variable) 

saved %ebp 
(base pointer) 

saved %eip 
(ret. address.) 

func call argument 

... 

p[\texttt{idx}]=\texttt{val};
Shadow stack

Stack

...  

char buff[16]

int i  (local variable)

saved %ebp #2  (base pointer)

saved %eip #2  (ret. address.)

func call argument

...  

Shadow stack

...  

...  

saved %ebp #2  (base pointer)

saved %eip #2  (ret. address.)

saved %ebp #1  (base pointer)

saved %eip #1  (ret. address.)

saved %ebp #0  (base pointer)

saved %eip #0  (ret. address.)

...
Shadow stack

Stack

... char buff[16]
int i (local variable)
saved %ebp #2 (base pointer)
saved %eip #2 (ret. address.)
func call argument
...
Protected region

Safe Stack

Unsafe stack

Safe stack (original stack)

int i (local variable)

saved %ebp (base pointer)

saved %eip (ret. address.)

func call argument

char buff[16]
Protecting the Safe Stack

movl $42, %ds:(%eax)

movl $42, %ss:(%esp)

movl $42, (%rsp)
How effective is the Safe Stack?

- **Strictly stronger** protection than stack cookies or shadow stack
- Only the Safe Stack provides **guaranteed** protection against return address corruption
- Stops **all ROP attacks** alone!
Safe Stack overhead

SPEC 2006 Benchmark

0% avg.
Safe Stack overhead

SPEC 2006 Benchmark

Perf. overhead

Cookies

Cookies-all

Shadow-stack

Safe-stack
Code Pointer Separation

Protecting function pointers
Integrity of function pointers

Heap

...  

buffer

func_ptr

int

int_ptr

...
Integrity of function pointers

Heap

...
Integrity of function pointers

Heap

\[
p[\text{idx}] = \text{val};
\]
Code Pointer Separation (CPS)

Heap

- ... 
- buffer
- func_ptr
- int
- int_ptr
- ...

Safe Pointer Store

- ... 
- ... 
- ... 
- func_ptr
- ... 
- ... 
- ...

Protected region
Code Pointer Separation (CPS)

Heap
- data_ptr
- func_ptr
- ...

Safe Pointer Store
- ...
- func_ptr
- ...

Unsafe stack
- ...
- ...
- char buff[8]
- ...

Safe stack (original stack)
- int i (local variable)
- saved %ebp (base pointer)
- saved %eip (ret. address.)
- func call argument

Protected region
Protecting the Safe Pointer Store

movl $42, %fs:(%rax)
movl $42, (%rsp)
movl $42, %ds:(%eax)
movl $42, %gs:(%eax)
movl $42, %ss:(%esp)

Regular Data Segment
Safe Pointer Store Segment
Safe Stack Segment
Safe Stack Segment

x86-32

movl $42, (%rsp)
movl $42, (%esp)

x86-64

movl $42, (%rax)
movl $42, (%rsp)
How effective is CPS?

```cpp
obj->func();
```
How effective is CPS?

```
obj->func();
```
## CPS vs. CFI

### Practical CFI solutions
- Classic CFI, CCS ‘05
- CCFIR, IEEE S&P ‘13
- binCFI, Usenix Sec ‘13
- kBouncer, Usenix Sec ‘13

### CFI attacks
- Göktaş et al., IEEE S&P ‘14
- Göktaş et al., Usenix Sec ‘14
- Davi et al., Usenix Sec ‘14
- Carlini et al., Usenix Sec ‘14

<table>
<thead>
<tr>
<th></th>
<th>CFI</th>
<th>CPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Calls can go to</strong></td>
<td>any function whose address is taken</td>
<td>any function whose address is taken and stored in memory at the current point of execution</td>
</tr>
<tr>
<td><strong>Return can go to</strong></td>
<td>any call site</td>
<td>only their actual caller</td>
</tr>
</tbody>
</table>
CPS overhead

SPEC 2006 Benchmark

2% avg.
Code Pointer Integrity

*Guaranteed* protection of *all* code pointers
obj->func();

**Issue #1**

- `obj` -> `func`
  - `func` - `do_good()`
  - `func` - `do_bad()`
Issue #1: pointer coverage

```
obj->func();
```
Issue #1: pointer coverage

```c
obj->func();
```
obj=&objs[idx]
obj->func();

ISSUE #2

objs +idx

func

do_good()

func

do_well()

func

do_bad()
obj = &objs[idx]
obj->func();
Issue #3

```
obj
  do_good()
  func
  do_bad()
```

→ delete obj;
...
obj->func();
Issue #3

```cpp
obj

do_good()
do_bad()

delete obj;
...
obj->func();
```
Issue #3: temporal safety

delete obj;
...
→ obj->func();
CPS → Code Pointer Integrity
Issue #1: pointer coverage

Protected region

Safe Pointer Store

obj_ptr
int
func_ptr
int_ptr

obj_ptr
func_ptr
Issue #2: spatial safety

Safe Pointer Store

<table>
<thead>
<tr>
<th>obj_ptr</th>
<th>lower_bound</th>
<th>upper_bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Protected region
Issue #3: temporal safety

Safe Pointer Store

<table>
<thead>
<tr>
<th>obj_ptr</th>
<th>lower_bound</th>
<th>upper_bound</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>func_ptr</td>
<td>func_ptr</td>
<td>func_ptr</td>
<td>-</td>
</tr>
</tbody>
</table>

Protected region
CPI overhead

SPEC 2006 Benchmark

8% avg.
Implementation
and case studies
Levee in LLVM/Clang

clang -fcpi
clang -fcps
clang -fsafe-stack

Get the prototype from: http://levee.epfl.ch
Control-flow hijack protected FreeBSD

- Complete FreeBSD distribution (modulo kernel)
- >100 extra packages
Summary
Summary

0% avg.

Safe Stack
Summary

- Code Pointer Separation: 2% avg.
- Safe Stack: 0% avg.
Summary

- Code Pointer Integrity: 8% avg.
- Code Pointer Separation: 2% avg.
- Safe Stack: 0% avg.
Thank you!

Questions?