CSE 127: Computer Security

ROP, heap attacks, CFI, integer overflows

Deian Stefan

Some slides adopted from Nadia Heninger, Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, Raluca Popal, and David Wagner
• Advanced modern attack techniques
  ➢ ROP
  ➢ Heap-based attacks
• Control flow integrity
• Integer overflow attacks
Employees must wash hands before returning to libc
What if there is no code that does what we want?
Return-Oriented Programming is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.
The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

*Work done while at the Weizmann Institute of Science, Rehovot, Israel, supported by a Koshland Scholars Program postdoctoral fellowship.
Return-Oriented Programming
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  ➤ Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  ➤ Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.

• Where do you often find ret instructions?
Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in `ret` instruction
  - Overwrite saved `%eip` on stack to pointer to first gadget, then second gadget, etc.
- Where do you often find `ret` instructions?
  - End of function (inserted by compiler)
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  ➤ Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.

• Where do you often find ret instructions?
  ➤ End of function (inserted by compiler)
  ➤ Any sequence of executable memory ending in 0xc3
x86 instructions

- Variable length!
- Can begin on any byte boundary!
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3
```

```
= mov $0x1,%eax
   pop %ebx
   leave
   ret
```
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3  
    add %al, (%eax) 
    pop %ebx 
    leave 
    ret
```
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3

= add %bl,-0x37(%eax)
ret
```
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3 = 
pop %ebx
leave
ret
```
One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 = leave
ret
One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3  _  =  ret
What does this gadget do?
relevant stack:

<table>
<thead>
<tr>
<th>%esp</th>
<th>%esp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xdeadbeef</td>
<td>0x08049bbc</td>
</tr>
</tbody>
</table>

relevant register(s):

%edx = 0x00000000

relevant code:

%eip → 0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
0x08049bbd: ret
relevant stack:

%esp

0xdeadbeef
0x08049bbc

relevant register(s):

%edx = 0x00000000

relevant code:

0x08049b62: nop
%eip ← 0x08049b63: ret ...

0x08049bbc: pop %edx
0x08049bbd: ret
relevant stack:

<table>
<thead>
<tr>
<th>%esp</th>
<th>0xdeadbeef</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x08049bbc</td>
</tr>
</tbody>
</table>

relevant register(s):

%edx = 0x00000000

relevant code:

0x08049b62: nop
0x08049b63: ret
...
%eip → 0x08049bbc: pop %edx
0x08049bbd: ret
relevant register(s):

%edx = 0xdeadbeef

relevant stack:

%esp ➔

0xdeadbeef
0x08049bbc

relevant code:

0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
%eip ➔ 0x08049bbd: ret
What does this gadget do?

\[
\begin{align*}
%edx &= v_1 \\
\text{mov} &\ v_1, %edx
\end{align*}
\]
How do you use this as an attacker?

- Overflow the stack with values and addresses to such gadgets to express your program
- E.g., if shellcode needs to write a value to %edx, use the previous gadget
What does this gadget do?

```
%esp  ──%esp
   v2    mov %eax, %ebx
   ret
   v1    pop %ebx
   ret
   pop %eax
   ret
```
relevant register(s):

| %eax  | 0x00000000 |
| %ebx  | 0x00000000 |

relevant stack:

%esp

0x08049b90
0xbadcaffe
0x08049b63
0xdeadbeef
0x08049bbc

relevant memory:

0xbadcaffe: 0x00000000

relevant code:

%eip → 0x08049b00: ret
... 
0x08049b63: pop %ebx
0x08049b64: ret
... 
0x08049b90: mov %eax, %ebx
0x08049b91: ret
... 
0x08049bbc: pop %eax
0x08049bbd: ret
### relevant stack:

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08049b90</td>
</tr>
<tr>
<td>0xbadcaffe</td>
</tr>
<tr>
<td>0x08049b63</td>
</tr>
<tr>
<td>0xdeadbeef</td>
</tr>
<tr>
<td>0x08049bbc</td>
</tr>
</tbody>
</table>

### relevant memory:

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbadcaffe: 0x00000000</td>
</tr>
</tbody>
</table>

### relevant code:

```assembly
0x08049b00: ret
...
0x08049b63: pop %ebx
0x08049b64: ret
...
0x08049b90: mov %eax, %(%ebx)
0x08049b91: ret
...
%eip → 0x08049bbc: pop %eax
0x08049bbd: ret
```

### relevant register(s):

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x00000000</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
relevant register(s):

%eax = 0xdeadbeef
%ebx = 0x00000000

relevant stack:

%esp

| 0x08049b90 |
| 0xbadcaffe |
| 0x08049b63 |
| 0xdeadbeef |
| 0x08049bbc |

relevant memory:

0xbadcaffe: 0x00000000

relevant code:

0x08049b00: ret
...  
0x08049b63: pop %ebx
0x08049b64: ret
...  
0x08049b90: mov %eax, %(%ebx)
0x08049b91: ret
...  
0x08049bbc: pop %eax
%eip → 0x08049bbd: ret
relevant register(s):

%eax = 0xdeadbeef
%ebx = 0x00000000

relevant stack:

%esp

0x08049b90
0xbadcaffe
0x08049b63
0xdeadbeef
0x08049bbc

relevant memory:

0xbadcaffe: 0x00000000

relevant code:

0x08049b00: ret
...:
0x08049b63: pop %ebx
0x08049b64: ret
...
0x08049b90: mov %eax, %(%ebx)
0x08049b91: ret
...
0x08049bbc: pop %eax
0x08049bbd: ret
relevant register(s):

\[
\begin{align*}
\%eax &= 0x\text{deadbeef} \\
\%ebx &= 0xb\text{adcaffe}
\end{align*}
\]

relevant stack:

\[
\begin{array}{c}
\%esp \\
0x08049b90 \\
0xb\text{adcaffe} \\
0x08049b63 \\
0x\text{deadbeef} \\
0x08049bbc
\end{array}
\]

relevant memory:

\[
\begin{align*}
0xb\text{adcaffe} & : 0x00000000
\end{align*}
\]

relevant code:

\[
\begin{align*}
0x08049b00 & : \text{ret} \\
\ldots \\
0x08049b63 & : \text{pop} \ %ebx \\
0x08049b64 & : \text{ret} \\
\ldots \\
0x08049b90 & : \text{mov} \ %eax, \ %(%ebx) \\
0x08049b91 & : \text{ret} \\
\ldots \\
0x08049bbc & : \text{pop} \ %eax \\
0x08049bbd & : \text{ret}
\end{align*}
\]
relevant register(s):

\[
\begin{align*}
%eax &= 0x\text{deadbeef} \\
%ebx &= 0xb\text{adcaffe}
\end{align*}
\]

relevant stack:

\[
\begin{array}{l|l}
\%esp & \downarrow \\
0x08049b90 & \small\\
0xb\text{adcaffe} & \small\\
0x08049b63 & \small\\
0x\text{deadbeef} & \small\\
0x08049bbc & \small
\end{array}
\]

relevant memory:

\[
0xb\text{adcaffe}: 0x00000000
\]

relevant code:

\[
\begin{align*}
0x08049b90: & \text{ mov } %eax, %\text{(ebx)} \\
0x08049b91: & \text{ ret}
\end{align*}
\]

\[
\begin{align*}
0x08049b00: & \text{ ret} \\
0x08049b63: & \text{ pop } %ebx \\
0x08049b64: & \text{ ret}
\end{align*}
\]

\[
\begin{align*}
%eip & \rightarrow 0x08049b90: \text{ mov } %eax, %\text{(ebx)} \\
0x08049b91: & \text{ ret}
\end{align*}
\]

\[
\begin{align*}
0x08049bbc: & \text{ pop } %eax \\
0x08049bbd: & \text{ ret}
\end{align*}
\]
relevant stack:

%esp

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08049b90</td>
<td></td>
</tr>
<tr>
<td>0xbadcaffe</td>
<td></td>
</tr>
<tr>
<td>0x08049b63</td>
<td></td>
</tr>
<tr>
<td>0xdeadbeef</td>
<td></td>
</tr>
<tr>
<td>0x08049bbc</td>
<td></td>
</tr>
</tbody>
</table>

relevant memory:

0xbadcaffe: 0xdeadbeef

relevant register(s):

%eax = 0xdeadbeef
%ebx = 0xbadcaffe

relevant code:

0x08049b00: ret
  ...  
0x08049b63: pop %ebx
0x08049b64: ret
  ...  
0x08049b90: mov %eax, %(%ebx)
0x08049b91: ret
  ...  
0x08049bbc: pop %eax
0x08049bbd: ret
What does this gadget do?

\[
\begin{align*}
\text{mem} [v_2] &= v_1 \\
\text{mov} \ v_2, \ %\text{ebx} \\
\text{mov} \ v_1, \ %\text{ebx} \\
\end{align*}
\]
Can express arbitrary programs

Figure 5: Simple add into %eax.

Figure 16: Shellcode.

Figure 10: An infinite loop by means of an unconditional jump.
Can find gadgets automatically

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, Dan Boneh
Stanford University

Ropper - rop gadget finder and binary information tool

You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. The following architectures are supported:

- x86 / x86_64
- Mips / Mips64
- ARM (also Thumb Mode)/ ARM64
- PowerPC / PowerPC64
Return-Oriented Programming

not even really about “returns”...
Today

- Advanced modern attack techniques
  - ROP
  - Heap-based attacks
- Control flow integrity
- Integer overflow attacks
Handling heap-allocated memory can be just as error-prone as the stack

• We may:
  ➤ Write/read memory we shouldn’t have access to
  ➤ Forget to free memory
  ➤ Free already freed objects
  ➤ Use pointers that point to freed object

• What if the attacker can cause the program to use freed objects?
Heap corruption

• Can bypass security checks (data-only attacks)
  ➤ E.g., isAuthenticatated, buffer_size, isAdmin, etc.

• Can overwrite function pointers
  ➤ Direct transfer of control when function is called
  ➤ C++ virtual tables are especially good targets
vtables

• Each object contains pointer to vtable
• Array of function pointers
  ➤ one entry per function
• Call looks up entry in vtable

Q: What does bar() compile to?
A: *(obj->vtable[0])(obj)
What does a use after free (UAF) attack look like?

Victim: Free object: free(obj);

Attacker: Overwrite the vtable of the object so entry (e.g., obj->vtable[0]) points to attacker gadget

Victim: Use dangling pointer: obj->foo()
Today

- Advanced modern attack techniques
  - ROP
  - Heap-based attacks
- Control flow integrity
- Integer overflow attacks
Control Flow Integrity

- In almost all the attacks we looked at, the attacker is overwriting jump targets that are in memory (return addresses on the stack and function pointers on the stack/heap).

- **Idea**: don’t try to stop the memory writes. Instead: restrict control flow to legitimate paths.
  - I.e., ensure that jumps, calls, and returns can only go to allowed target destinations.
Restrict indirect transfers of control
Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e., direct jumps/calls)?
Restrict indirect transfers of control

• Why do we not need to do anything about direct transfer of control flow (i.e., direct jumps/calls)?
  ➤ Address is hard-coded in instruction. Not under attacker control
Restrict indirect transfers of control
Restrict indirect transfers of control

- What are the ways to transfer control indirectly?
Restrict indirect transfers of control

• What are the ways to transfer control indirectly?

• **Forward path:** jumping to (or calling function at) an address in register or memory
  ➤ E.g., qsort, interrupt handlers, virtual calls, etc.

• **Reverse path:** returning from function (uses address on stack)
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```
What's a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```

```plaintext
What's a legitimate target?

Look at the program control-flow graph (CFG)!

```

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```
What's a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

- `sort2()`
- `sort()`
- `lt()`
- `gt()`

- Call `sort`
- Call `arg$3`
- Return
- Return
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```
How do we restrict jumps to CFG?

- Assign labels to all indirect jumps and their targets
- Before taking an indirect jump, validate that target label matches jump site
  - Like stack canaries, but for control flow target
- Need hardware support
  - Otherwise trade off precision for performance
Fine grained CFI (Abadi et al.)

• Statically compute CFG

• Dynamically ensure program never deviates
  ➤ Assign label to each target of indirect transfer
  ➤ Instrument indirect transfers to compare label of destination with the expected label to ensure it's valid
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
Fine grained CFI (Abadi et al.)

```c
void sort2(int a[], int b[], int len {  
    sort(a, len, lt);  
    sort(b, len, gt);  
}

bool lt(int x, int y) {  
    return x < y;  
}

bool gt(int x, int y) {  
    return x > y;  
}
```
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
Fine grained CFI (Abadi et al.)

```c
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

```plaintext
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Make sure that every indirect call lands on function entry

- Label for destination of rets and indirect jumps
  - Make sure every indirect jump lands at start of BB
Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Make sure that every indirect call lands on function entry

- Label for destination of rets and indirect jumps
  - Make sure every indirect jump lands at start of BB
How else can you choose labels?

$$tf = t_1^* \rightarrow t_2^* \quad C_{table} = n$$

$$C \vdash \text{call\_indirect} \ tf : t_1^* \ i32 \rightarrow t_2^*$$

```
s; (i32.const j) call\_indirect tf ↪ s\_tab(i, j)
s; (i32.const j) call\_indirect tf ↪ trap

if s\_tab(i, j)\_code == (func tf local t* e*)
else
```
How else can you choose labels?

WebAssembly does it by looking at function type

\[
\begin{array}{c}
t_f = t_1^* \rightarrow t_2^* \\
C_{\text{table}} = n \\
\hline
C \vdash \text{call\_indirect } t_f : t_1^* \text{ i32} \rightarrow t_2^*
\end{array}
\]
CFI limitations

• Overhead
  ➢ Runtime: every indirect branch instruction
  ➢ Size: code before indirect branch + encode label at destination

• Scope
  ➢ CFI does not protect against data-only attacks
  ➢ Needs reliable W^X
How can you defeat CFI?

- Imprecision can allow for control-flow hijacking
  - Can jump to functions that have same label
    - E.g., even if we use Wasm’s labels `int system(char*)` and `int myFunc(char*)` share the same label
  - Can return to many more sites
    - But, real way to do backward edge CFI is to use a shadow stack. (This is actually great!)
Today

• Advanced modern attack techniques
  ➢ ROP
  ➢ Heap-based attacks

• Control flow integrity

• Integer overflow attacks
What’s wrong with this program?

```c
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```
What’s wrong with this program?

```c
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

MEMCPY(3) Linux Programmer’s Manual MEMCPY(3)

NAME  top

memcpy - copy memory area

SYNOPSIS  top

```c
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```
What’s wrong with this program?

```c
def vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

MEMCPY(3)       Linux Programmer’s Manual       MEMCPY(3)

**NAME**

memcpy - copy memory area

**SYNOPSIS**

```c
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```
What’s wrong with this program?

```c
void vulnerable(int len = 0xffffffff, char *data) {
    char buf[64];
    if (len = -1 > 64)
        return;
    memcpy(buf, data, len = 0xffffffff);
}
```

---

### NAME

`memcpy` - copy memory area

### SYNOPSIS

```c
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```
Is this program safe?

```c
void f(size_t len, char *data) {
    char *buf = malloc(len+2);
    if (buf == NULL)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len+1] = '\0';
}
```
void f(size_t len = 0xffffffff, char *data) {
    char *buf = malloc(len+2 = 0x000000001);
    if (buf == NULL)
        return;
    memcpy(buf, data, len = 0xffffffff);
    buf[len] = '\n';
    buf[len+1] = '\0';
}

No!
Still relevant classes of bugs

Issue 952406: Security: Possible OOB related to chrome_sqlite3_malloc
Reported by mlfbr...@stanford.edu on Fri, Apr 12, 2019, 1:59 PM PDT

VULNERABILITY DETAILS
Possible OOB with chrome_sqlite3_malloc

REPRODUCTION CASE
There's a pattern of using sqlite malloc functions that call chrome_sqlite3_malloc in combination with traditional memory operations (e.g., memcpy). There may be invariants that make this ok, or a principle here that I am not aware of. Thanks for your time.

chrome_sqlite3_malloc takes an int size argument, while memcpy takes a size_t size argument. On x86-64 this means that chrome_sqlite3_malloc's size argument is width 32, while memcpy's is width 64. This can lead to potentially concerning wrapping behavior for extreme allocation sizes (depending on the compiler, optimizations, etc).

For example:

Function fts3UpdateDocTotals

(1) a = sqlite3_malloc( (sizeof(u32)+10)*nStat );
...
(2) memset(a, 0, sizeof(u32))*nStat);

Depending on optimization level etc, this may turn into:

(1)
size = mul i32 nstat 14
chrome_sqlite3_malloc(size)
Three flavors of integer overflows

• Truncation bugs
  ➤ E.g., assigning an int64_t into in32_t (3rd ex)

• Arithmetic overflow bugs
  ➤ E.g., adding huge unsigned number (2nd ex)

• Signedness bugs
  ➤ E.g., treating signed number as unsigned (1st ex)
Today

• Advanced modern attack techniques
  ➤ ROP
  ➤ Heap-based attacks

• Control flow integrity

• Integer overflow attacks
What does this all tell us?

If you’re trying to build secure systems, use a memory safe language.