CSE 127: Introduction to Security

Memory safety and other vulnerabilities

Nadia Heninger
UCSD

Winter 2021 Lecture 4

Some slides from Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, David Wagner, and Deian Stefan
Legal Notice

This lecture will be recorded and made available to registered students on Canvas.
Today

- Return-oriented programming
- Control flow integrity
- Heap corruption
- Integer overflows
Last time: return-to-libc

- **Defense:** $W^X$ makes the stack not executable
  - Prevents attacker data from being interpreted as code

- **What can we do as the attacker?**
  - Reuse existing code (either program or libc)
    - e.g. use `system("/bin/sh")`
    - e.g. use `mprotect()` to mark stack executable
Return-to-libc is great, but...

what if there is no function that does what we want?
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  • Overwrite saved $%e\_ip$ 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.

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.
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

\[
\text{b8 01 00 00 00 5b c9 c3} = \text{add } \%\text{bl, }-0x37(\%\text{eax})
\]
\[
\text{ret}
\]
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
Why `ret`?

- Attacker overflows stack allocated buffer

- What happens when function returns?
  - Restore stack frame
    - `leave = movl %ebp, %esp; pop %ebp`
  - Return
    - `ret = pop %eip`

- If instruction sequence at `%eip` ends in `ret` what do we do?
What happens if this is what we overflow the stack with?
relevant stack:

%esp →

0xdeadbeef
0x08049bbc

relevant register(s):

%edx = 0x00000000

relevant code:

%eip →

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

%edx = 0x00000000

relevant stack:

%esp → 0xdeadbeef
  0x08049bbc

relevant code:

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

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

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

relevant code:

\[
\begin{align*}
0x08049b62: & \text{ nop} \\
0x08049b63: & \text{ ret} \\
\text{...}
\end{align*}
\]

\[
\begin{align*}
\%eip & \rightarrow 0x08049bbc: \text{ pop } \%edx \\
0x08049bcb: & \text{ ret}
\end{align*}
\]

relevant register(s):

\[
\%edx = 0x00000000
\]
relevant register(s):

%edx = 0xdeadbeef

relevant stack:

%esp 0xdeadbeef
0x08049bbc

relevant code:

0x08049b62: nop
0x08049b63: ret
...

0x08049bbc: pop %edx

%eip 0x08049bbd: ret
This is a ROP gadget!

movl v1, %edx
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
Can express arbitrary programs

Figure 5: Simple add into %eax.

Figure 16: Shellcode.
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
How do you mitigate ROP?

**Observation:** In almost all the attacks we looked at, the attacker is overwriting jump targets that are in memory (return addresses and function pointers)
Today

• Return-oriented programming
→ Control flow integrity
• Heap corruption
• Integer overflows
Control Flow Integrity

• **Idea:** Don’t try to stop the memory writes.

• **Instead:** Restrict control flow to legitimate paths
  • Ensure that jumps, calls, and returns can only go to allowed target destinations
Restrict indirect transfers of control

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

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

What are the ways to transfer control indirectly?

• **Forward path:** Jumping to or calling a function at an address in register or memory
  - e.g. qsort, interrupt handlers, virtual calls, etc.

• **Reverse path:** Returning from function using address on stack
What’s a legitimate target?

Look at the program control-flow graph!

```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 control flow graph?

- 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
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 a basic block
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
Control Flow Integrity 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

• Coarse-grained CFI can return to many sites
  • Can use a shadow stack to implement fully precise CFI
Today

- Return-oriented programming
- Control flow integrity
  → Heap corruption
- Integer overflows
Memory management in C/C++

- C uses explicit memory management
  - Data is allocated and freed dynamically
  - Dynamic memory is accessed via pointers

- You are on your own
  - System does not track memory liveness
  - System doesn’t ensure that pointers are live or valid

- By default C++ has same issues
The heap

• Dynamically allocated data stored on the “heap”

• Heap manager exposes API for allocating and deallocating memory

  • malloc() and free()

  • API invariant: All memory allocated by malloc() has to be released by corresponding call to free()
Heap management

• Organized in contiguous chunks of memory
  • Basic unit of memory
  • Can be free or in use
  • Metadata: size + flags
  • Allocated chunk: payload

• Heap layout evolves with malloc()s and free()s
  • Chunks may get allocated, freed, split, coalesced

• Free chunks are stored in doubly linked lists (bins)
  • Different kinds of bins: fast, unsorted, small, large, …
How can things go wrong?

- Forget to free memory
- Write/read memory we shouldn’t have access to: Overflow code pointers on the heap
- Use after free: Use pointers that point to freed object
- Double free: Free already freed objects
Most important: heap corruption

- Can bypass security checks (data-only attacks)
  - e.g. isAuthenticated, buffer_size, isAdmin, etc.

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

- Can overwrite heap management data
  - Corrupt metadata in free chunks
  - Program the heap weird machine
C++ vtables

```cpp
class Base {
    public:
        uint32_t x;
        Base(uint32_t x) : x(x) {}
        virtual void f() {
            cout << "base: " << x;
        }
    };

class Derived: public Base {
    public:
        Derived(uint32_t x) : Base(x) {}
        void f() {
            cout << "derived: " << x;
        }
    };

void bar(Base* obj) {
    obj->f();
}

int main(int argc, char* argv[]) {
    Base *b = new Base(42);
    Derived *d = new Derived(42);
    bar(b);
    bar(d);
}
```

What does this print out?
base: 42
derived: 42

What does `bar()` compile to?
*(obj->vtable[0])(obj)
Use-after-free in C++

**Victim:** Free object: `free(obj);`

**Attacker:** Overwrite the vtable of the object so entry `(obj->vtable[0])` points to attacker gadget

**Victim:** Use dangling pointer: `obj->foo()`
Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape

Matt Miller (@epakskape)
Microsoft Security Response Center (MSRC)

BlueHat IL
February 7th, 2019
Drilling down into root causes

Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

#1: heap out-of-bounds
#2: use after free
#3: type confusion
#4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories
Heap exploitation mitigations

- Safe heap implementations
  - Safe unlinking
  - Cookies/canaries on the heap
  - Heap integrity check on malloc and free

- Use Rust or a safe garbage collected language
Today

- Return-oriented programming
- Control flow integrity
- Heap corruption

→ Integer bugs
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**

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

```
#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);
}
```
What’s wrong with this program?

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

#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
Let’s fix it

```c
void safe(size_t len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```
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';
}
```
Is this program safe?

```c
void f(size_t len = 0xffffffff, char *data) {
    char *buf = malloc(len+2 = 0x00000001);
    if (buf == NULL)
        return;
    memcpy(buf, data, len = 0xffffffff);
    buf[len] = '\n';
    buf[len+1] = '\0';
}
```

No!
Three flavors of integer overflows

- Truncation bugs
  - e.g. assigning an `int64_t` into `int32_t`

- Arithmetic overflow bugs
  - e.g. adding huge unsigned numbers

- Sign bugs
  - e.g. treating signed number as unsigned
Still relevant classes of bugs

Issue 952406: Security: Possible OOB related to chrome_sqlite3_malloc
Reported by mlbr...@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
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&g=fts3UpdateDocTotals&g=0&l=3399)

(1) a = sqlite3_malloc((sizeof(u32)+10)*nStat);
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&g=fts3UpdateDocTotals&g=0&l=3416)

... (2) memset(a, 0, sizeof(u32)*nStat);
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&g=fts3UpdateDocTotals&g=0&l=3434)

Depending on optimization level etc. this may turn into:

(1)
size = mul i32 nstet 14
chrome_sqlite3_malloc(size)

(2)
tmp = sign extend nstet to i64
size = shl tmp 2
memset(size)

If nstet is a very large i32, the multiplication in step (1) *may* wrap. Nothing in (2) will wrap because of the sign extend, leading to an OOB.
Today

- Return-oriented programming
- Control flow integrity
- Heap corruption
- Integer bugs
What does all this tell us?

If you’re trying to build a secure system, use a memory and type-safe language.