CSci 4271W
Development of Secure Software Systems
Day 6: Memory corruption 2, attack strategies

Stephen McCamant (he/him)
University of Minnesota, Computer Science & Engineering

Memory corruption

- Memory corruption bugs happen when a program writes data to an area of memory that it shouldn't.
- Type-safe languages such as Java, OCaml, Rust, Swift, and Go can prevent most such bugs.
- In C or C++ it is easy to write a program that corrupts memory:

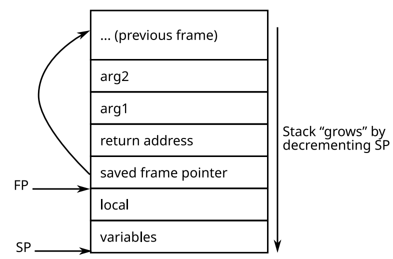
```
int x = 0x0011;
char buff[4];
buff[4] = 'a'
```

So what?

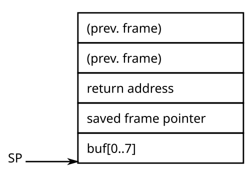
Recall, each function call has a stack frame that stores:

- Function arguments
- Local variables
- Any "callee-saved" register values
- The memory address of the next instruction from the calling function, i.e. the return address.

Typical stack frame



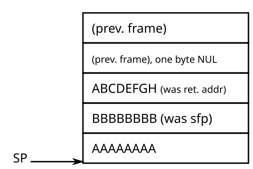
Smashing the stack




```
void func(char *str) {
  char buf[8];
  strcpy(buf, str);
  do_something(buf);
}
```

Calling `func("AAAAAAAABBBBBBBBABCDEFGH")`

Smashing the stack, cont'd

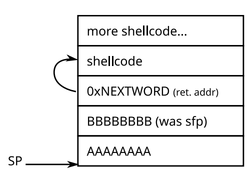



```
void func(char *str) {
  char buf[8];
  strcpy(buf, str);
  do_something(buf);
}
```

Calling `func("AAAAAAAABBBBBBBBABCDEFGH")`
After `strcpy`

Func will try to return to instruction at address ABCDEFGH

For fun and profit




```
void func(char *str) {
  char buf[8];
  strcpy(buf, str);
  do_something(buf);
}
```

Classic attack: replace ABCDEFGHI with address inside overwrite on stack, insert compiled shell code that calls `exec("bin/sh")`

`https://archives.phrack.org/issues/49/14.txt`

Last time

```
char *p = malloc(8);
char *q = "xxxxxxxxxxxxx";
while (*p++ = *q++) ;
```

```
char *p = malloc(32);
*(--p) = 'A';
```

```
char *p = malloc(8);
strcpy(p, "hello");
free(p);
long *x = malloc(8);
*x = 65;
strlen(p);
```

```
char *fmt = "%97x%13$n";
printf(fmt);
```

Outline

- Review from last lecture
- **Announcements break**
- Memory corruption strategies
- Low-level code examples

Upcoming events

- Monday's lab is on stack smashing
- Homework 2 is available now, will be due 2/18
- Project 1 information will be out next week
- Midterm 1 is Thursday 2/20

Outline

- Review from last lecture
- Announcements break
- **Memory corruption strategies**
- Low-level code examples

Memory corruption: big picture

- Modern OSes and compilers block such "easy" attacks, but more clever variants can still work.
- Why doesn't classic stack smashing work anymore?
  - Non-executable stack ($W \oplus X$ or DEP)
  - Stack canaries/cookies
  - Address space layout randomization (ASLR)

Non-executable stack

- Memory pages can be marked as writeable or an executable, but not both at noce
- This prevents jumping to code placed on the stack or heap.
  - But many library/system calls can load a new binary/shell (`exec`, `system`, `popen`) and libc is always in memory
- A "return to libc" attack works by overwriting the return address with a pointer to such a function

Non-executable stack

- Memory pages can be marked as writeable or an executable, but not both at noce
- This prevents jumping to code placed on the stack or heap.
  - But many library/system calls can load a new binary/shell (`exec`, `system`, `popen`) and libc is always in memory
- A "return to libc" attack works by overwriting the return address with a pointer to such a function
- More general: "return-oriented programming" (ROP)

Canaries/stack cookies (🍪)

A canary, aka "stack cookie", is a random value pushed on the stack between the return address and the local variables.

```
char s[128];
int local;
// do some stuff
return;
```

$\longrightarrow$

```
uint64_t cookie = 0xe99dbf2dd7ba0ad8;
char s[128];
int local;
// do some stuff
assert(cookie == 0xe99dbf2dd7ba0ad8);
return;
```

Overwrite an index, use a format string, leak the cookie... Or change something else!

What else to change?

There are other locations in C/C++ programs that can be used to direct control flow:

What else to change?

There are other locations in C/C++ programs that can be used to direct control flow:

- Function pointers: used for callbacks, Global Offset Table (GOT), ...

What else to change?

There are other locations in C/C++ programs that can be used to direct control flow:

- Function pointers: used for callbacks, Global Offset Table (GOT), ...
- Exception / signal handlers

What else to change?

There are other locations in C/C++ programs that can be used to direct control flow:

- Function pointers: used for callbacks, Global Offset Table (GOT), ...
- Exception / signal handlers
- `setjmp`/`longjmp` buffers

What else to change?

There are other locations in C/C++ programs that can be used to direct control flow:

- Function pointers: used for callbacks, Global Offset Table (GOT), ...
- Exception / signal handlers
- `setjmp`/`longjmp` buffers
- vtable pointers

What are vtables?

Virtual functions are implemented by lookups in a table pointed to by a hidden field in an object

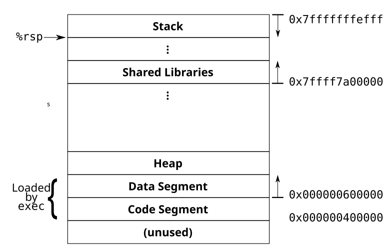
```
class Example {
  public:
  virtual void doThing() {
    ...
  }
};
```

```
void vulnerable(Example *p) {
  delete p;
  char *buf=malloc(sizeof(Example));
  fill_untrusted(buf);
  p->doThing();
}
```

Overwriting the vtable pointer (by use-after-free, heap overflow or stack overflow) can redirect a method call to an arbitrary address.

Address space...

Address space layout of a typical Linux/x86-64 process:



...layout randomization

Address space layout randomization (ASLR) randomizes:

- Stack location (always): hard to find the right address on stack to jump to
- Heap location (often): hard to find address of heap buffer to stash shellcode
- Shared libraries (often): hard to find address of libc
- Code/data segments (sometimes): hard to find address of existing code

ASLR problems

- 32-bit addresses are easy(ish) to guess (w/big NOP sled)
- Legacy code can prevent relocating libraries/code segment
- Relative offsets are maintained (for ret2libc/ROP)
- Linux default does not relocate code/data segments
- Uninitialized read, format string, interpreter bugs can leak secrets (ASLR offsets, also cookies)

Outline

- Review from last lecture
- Announcements break
- Memory corruption strategies
- Low-level code examples