Outline
Stack buffer overflow, recap
Reversing the stack
Reversing the stack, discussion
Other safety problems
Logistics announcements
Integer overflow example
Code auditing

Source-level view

```c
void func(char *attacker_controlled) {
    char buffer[50];
    strcpy(buffer, attacker_controlled);
}
```

Stack frame overflow

A possible solution

- Part of what makes this classic attack easy is that the array grows in the direction toward the function's return address.
- If we made the stack grow towards higher addresses instead, this wouldn't work in the same way.
- Classic puzzler: why isn't this a solution to the problem?

A concrete example

```c
void func(char *attacker_controlled) {
    char buffer[50];
    strcpy(buffer, attacker_controlled);
}
```

What might happen in this example, for instance?
Stack direction orientation

- Higher addresses are “deeper” in the stack, and represent older stack frames (callers) and data (pushed first)
- Lower addresses are closer to the “top” of the stack, representing more-recently pushed frames (callees) and data

Stack frame normal overflow

Outline

- Stack buffer overflow, recap
- Reversing the stack
- Reversing the stack, discussion
- Other safety problems
- Logistics announcements
- Integer overflow example
- Code auditing

Reversed overflow

Non-contiguous overflow

- An overflow doesn’t have to write to the buffer in sequence
- For instance, the code might compute a single index, and store to it

Heap buffer overflow

- Overwriting a malloc`ed buffer isn’t close to a return address
- But other targets are available:
  - Metadata used to manage the heap, contents of other objects

Use after free

- A common bug is to `free` an object via one pointer and keep using it via another
- Leads to unsafe behavior after the memory is reused for another object

Uninitialized use

- `malloc`ed memory and stack variables are officially undefined to start with
- In practice, because reused from earlier in execution
- If program uses without checking, might allow attacker control
Integer overflow
- Integer types have limited size, and will wrap around if a computation is too large
- Not unsafe itself, but often triggers later bugs
  - E.g., not allocating enough space

Function pointers, etc.
- Other data used for control flow could be targeted for overwriting by an attacker
- Common C case: function pointers
- More obscure C case: setjmp/longjmp buffers

Virtual dispatch
- When C++ objects have virtual methods, which implementation is called depends on the runtime type
- Under the hood, this is implemented with a table of function pointers called a vtable
- An appealing target in attacking C++ code

Non-control data overwrite
- An attacker can also trigger undesired-to-you behavior by modifying other data
- For instance, flags that control other security checks

Format string injection
- The first argument of printf is a little language controlling output formatting
- Best practice is for the format string to be a constant
- An attacker who controls a format string can trigger other mischief

Outline
- Stack buffer overflow, recap
- Reversing the stack
- Reversing the stack, discussion
- Other safety problems
- Logistics announcements
- Integer overflow example
- Code auditing

Prof. McCamant’s office hours
- Will be Mondays, 2-3pm, in 4-225E Keller Hall
- Starting next week
- Bowen’s office hours will also be announced soon
### Integer overflow to buffer overflow

- One common pattern: overflow causes an allocation to be too small
- In machine integers, multiplication doesn't always make a value larger

### Overflow example

```c
struct obj { short ident, x, y, z; long b; double c;};
struct obj *read_objs(int num_objs) {
    unsigned int size = num_objs*(unsigned)sizeof(obj);
    struct obj *objs = malloc(size);
    struct obj *p = objs;
    for (i = 0; i < num_objs; i++) {
        fread(p, sizeof(struct obj), 1, stdin);
        if (p->ident == 0x4442) return 0;
        /* ... */ p++;
    }
    return objs;
}
```

### Overflow example questions

1. What's a value of `num_objs` that would trigger an overflow?
   - Think back to 2021 on how multiplication overflows
2. Why is the `p->ident` check relevant to exploitability?

https://www-users.cselabs.umn.edu/classes/Spring-2024/csci4271/slides/02/overflow-eg.c

### Outline

- Stack buffer overflow, recap
- Reversing the stack
- Reversing the stack, discussion
- Other safety problems
- Logistics announcements
- Integer overflow example
- Code auditing

### Auditing is...

- Reading code to find security bugs
- Threat modeling comes first, tells you what kinds of bugs you're looking for
- Bug fixing comes next (might be someone else's job)

### Tiers and triage

- You might not have time to do a complete job, so use auditing time strategically
- Which bugs are most likely, and easiest to find?
- Triage into definitely safe, definitively unsafe, hard to tell
  - Hard to tell might be improved even if safe

### Threat model and taint

- Vulnerability depends on what an attacker might control
- Another word for attacker-controlled is “tainted”
- Threat model is the best source of tainting information
  - Of course, can always be conservative

### Where to look for problems

- If you can't read all the code carefully, search for indicators of common danger spots
  - For format strings, look for `printf`
  - For buffer overflows, look at buffers and copying functions
Ideal: proof

- Given enough time, for each dangerous spot, be able to convince someone:
  - Proof of safety: reasons why a bug could never happen, could turn into assertions
  - Proof of vulnerability: example of tainted input that causes a crash