CSci 4271W
Development of Secure Software Systems
Day 2: Memory Safety Introduction

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Outline

Memory safety and security

Stack buffer overflow

Reversing the stack

Other safety problems

A large class of problems

- First up, a common class of vulnerabilities in C/C++ programs
- Exist because these languages do not enforce safe use of memory
- An attacker who controls program input can make the program do what they want
- Language shifts burden to code, code is incorrect

Ingredient 1: memory unsafety

- Some logical limitations on memory usage are generally not automatically checked in C/C++.
  - Motivated by speed, simplicity, history
- Accessing arrays does not check against the size
- Program must `free` memory when no longer needed, then not use
  - I.e., no garbage collection

Ingredient 2: missing input checks

- Constraints on the *untrusted* input needed for safety are not checked
- Many normal uses of the program will still work fine
  - E.g., input size not too large
- Attacks occur on inputs that are rare or only an attacker would think of
  - Usually would have been OK to reject these

Recipe for safe code

- Safe code needs to ensure that for any value of the untrusted input, nothing unsafe will happen
- From pure security perspective, stopping with an error message is generally safe
- Like other kinds of bugs, easier said than done

Safe interfaces or better checks

- General strategy: use features and libraries with an inherently safer design
  - E.g., C++ `string` class with automatic memory management
- General strategy: add more checks for unsafe or just unexpected conditions
  - Allow fewer inputs → fewer attack opportunities

Auditing and testing

- Reading code looking for security problems is called a *code audit*
  - Often more effective if the reader has fresh eyes
- Many security bugs can be found via testing
  - Especially randomized automatic testing called *fuzzing*

After something goes wrong

- At language level, no guarantees about behavior of memory-unsafe code
  - C *undefined behavior* means literally anything can happen
- On real implementations, most unsafe effects understandable from low-level perspective
  - This is where what you learned in 2021 is relevant
- How an attack succeeds in doing something interesting is more complex

Mitigation: an arms race

- Modern systems also make many changes to the compiler and runtime to try to make attacker's life harder
  - ASLR, DEP, stack canaries, . . . more details later
- But for performance and compatibility, usually not complete protections
- Attackers also have fancier techniques to avoid them

Outline

Memory safety and security

Stack buffer overflow

Reversing the stack

Other safety problems

Source-level view (1)

```
void func(void) {
    char buffer[50];
    write_200_bytes_into(buffer);
}
```

Source-level view (2)

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

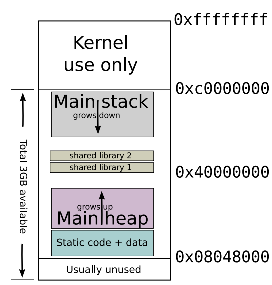
Demo break 1

- Simple palindrome checker:
  - Short input → correct behavior
  - Normal too-long input → crash
  - Malicious too-long input → exploit

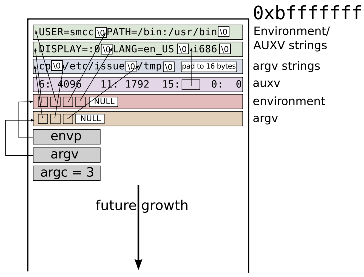
Recall: the stack

- In compiled C code, local variables and other metadata like return addresses are stored in a memory region called *the stack*
- Structured as a stack with one *frame* of data per executing function
- Starts at a numerically large address and grows to smaller addresses

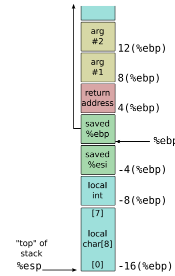
Overall layout (Linux 32-bit)



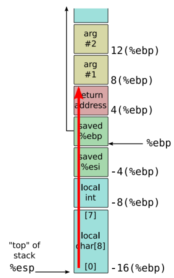
Detail: initial stack



Stack frame layout



Stack frame overflow



Demo break 2

- How did the attacker know how to overwrite the return address?

Outline

- Memory safety and security
- Stack buffer overflow
- **Reversing the stack**
- Other safety problems

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

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

What might happen in this example, for instance?

Outline

- Memory safety and security
- Stack buffer overflow
- Reversing the stack
- **Other safety problems**

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

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