

CSci 5271
Introduction to Computer Security
Day 3: Low-level vulnerabilities

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Preview question

In a 32-bit Linux/x86 program, which of these objects would have the lowest address (numerically least when considered as unsigned)?

- A. An environment variable
- B. The program name in `argv[0]`
- C. A command-line argument in `argv[1]`
- D. A local `float` variable in a function called by `main`
- E. A local `char` array in `main`

Outline

Vulnerabilities in OS interaction
Low-level view of memory
Logistics announcements
Basic memory-safety problems
Where overflows come from
More problems

Bad/missing error handling

- Under what circumstances could each system call fail?
- Careful about rolling back after an error in the middle of a complex operation
- Fail to drop privileges \Rightarrow run untrusted code anyway
- Update file when disk full \Rightarrow truncate

Race conditions

- Two actions in parallel; result depends on which happens first
- Usually attacker racing with you
 1. Write secret data to file
 2. Restrict read permissions on file
- Many other examples

Classic races: files in `/tmp`

- Temp filenames must already be unique
- But “unguessable” is a stronger requirement
- Unsafe design (`mktemp(3)`): function to return unused name
- Must use `O_EXCL` for real atomicity

TOCTTOU gaps

- Time-of-check (to) time-of-use races
 1. Check it's OK to write to file
 2. Write to file
- Attacker changes the file between steps 1 and 2
- Just get lucky, or use tricks to slow you down

TOCTTOU example

```
int safe_open_file(char *path) {
    int fd = -1;
    struct stat s;
    stat(path, &s);
    if (!S_ISREG(s.st_mode))
        error("only regular files allowed");
    else fd = open(path, O_RDONLY);
    return fd;
}
```

TOCTTOU example

```
int safe_open_file(char *path) {
    int fd = -1, res;
    struct stat s;
    res = stat(path, &s);
    if (res || !S_ISREG(s.st_mode))
        error("only regular files allowed");
    else fd = open(path, O_RDONLY);
    return fd;
}
```

TOCTTOU example

```
int safe_open_file(char *path) {
    int fd = -1, res;
    struct stat s;
    res = stat(path, &s);
    if (res || !S_ISREG(s.st_mode))
        error("only regular files allowed");
    else fd = open(path, O_RDONLY);
    return fd;
}
```

Changing file references

- With symbolic links
- With hard links
- With changing parent directories
- Avoid by instead using:
 - f* functions that operate on fds
 - *at functions that use an fd in place of the CWD

Directory traversal with ..

- Program argument specifies file with directory files
- What about files/../../../../etc/passwd?

Environment variables

- ☐ Can influence behavior in unexpected ways
 - PATH
 - LD_LIBRARY_PATH
 - IFS
 - ...
- ☐ Also umask, resource limits, current directory

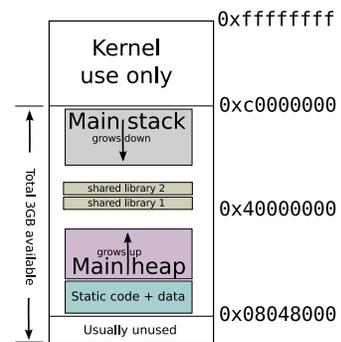
IFS and why it's a problem

- ☐ In Unix, splitting a command line into words is the shell's job
 - String → argv array
 - `grep a b c` VS. `grep 'a b' c`
- ☐ Choice of separator characters (default space, tab, newline) is configurable
- ☐ Exploit `system("/bin/uname")`

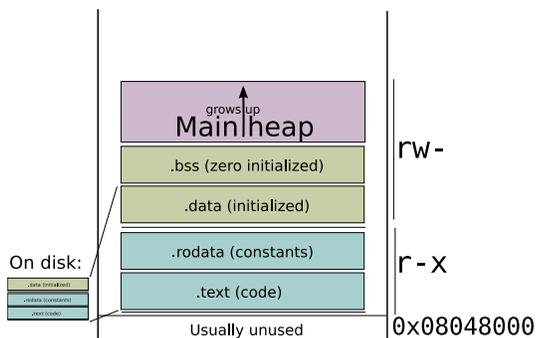
Outline

- Vulnerabilities in OS interaction
- Low-level view of memory
- Logistics announcements
- Basic memory-safety problems
- Where overflows come from
- More problems

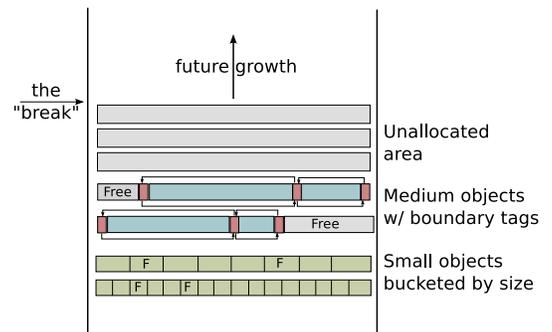
Overall layout (Linux 32-bit)



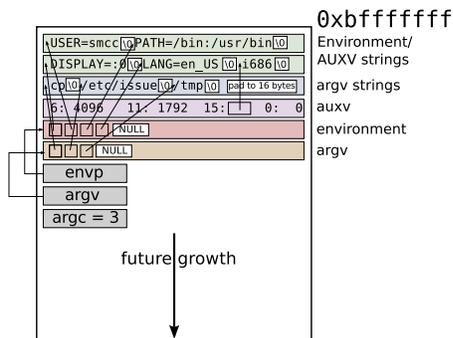
Detail: static code and data



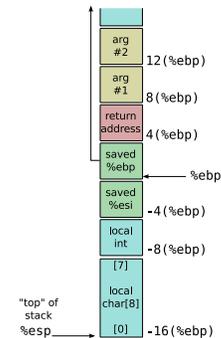
Detail: heap



Detail: initial stack



Example stack frame



Outline

- Vulnerabilities in OS interaction
- Low-level view of memory
- Logistics announcements
- Basic memory-safety problems
- Where overflows come from
- More problems

Canvas, discussions

- Canvas page started, will use for assignment turn-in
- Online discussions, including for group formation
- For spoiler questions, email both me and the TA, keep CC'd

Finding project topics

- Pre-proposal due 9/18 (one week from today)
- Don't skimp on topic selection: important to success
- Conference papers linked from class site
- Scheduling grid now available

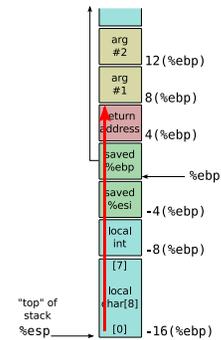
More on choosing topics

- Can't: wait to see what part of class you like best
 - But feel free to look ahead
- Think about your group's skills
 - Also: available hardware/software
- Think about where to find novelty
- Topic changes allowed, but will set you back

Outline

- Vulnerabilities in OS interaction
- Low-level view of memory
- Logistics announcements
- Basic memory-safety problems
- Where overflows come from
- More problems

Stack frame overflow



Overwriting adjacent objects

- Forward or backward on stack
 - Other local variables, arguments
- Fields within a structure
- Global variables
- Other heap objects

Overwriting metadata

- On stack:
 - Return address
 - Saved registers, incl. frame pointer
- On heap:
 - Size and location of adjacent blocks

Double free

- Passing the same pointer value to `free` more than once
- More dangerous the more other heap operations occur in between

Use after free

- AKA use of a *dangling pointer*
- Could overwrite heap metadata
- Or, access data with confused type

Outline

- Vulnerabilities in OS interaction
- Low-level view of memory
- Logistics announcements
- Basic memory-safety problems
- Where overflows come from
- More problems

Library funcs: unusable

- gets writes unlimited data into supplied buffer
- No way to use safely (unless stdin trusted)
- Finally removed in C11 standard

Library funcs: dangerous

- Big three unchecked string functions
 - strcpy(dest, src)
 - strcat(dest, src)
 - sprintf(buf, fmt, ...)
- Must know lengths in advance to use safely (complicated for sprintf)
- Similar pattern in other funcs returning a string

Library funcs: bounded

- Just add "n":
 - strncpy(dest, src, n)
 - strncat(dest, src, n)
 - snprintf(buf, size, fmt, ...)
- Tricky points:
 - Buffer size vs. max characters to write
 - Failing to terminate
 - strncpy zero-fill

More library attempts

- OpenBSD strncpy, strlcat
 - Easier to use safely than "n" versions
 - Non-standard, but widely copied
- Microsoft-pushed strncpy_s, etc.
 - Now standardized in C11, but not in glibc
 - Runtime checks that abort
- Compute size and use memcpy
- C++ std::string, glib, etc.

Still a problem: truncation

- Unexpectedly dropping characters from the end of strings may still be a vulnerability
- E.g., if attacker pads paths with // or /././././.
- Avoiding length limits is best, if implemented correctly

Off-by-one bugs

- ▣ `strlen` does not include the terminator
- ▣ Comparison with `<` vs. `<=`
- ▣ Length vs. last index
- ▣ `x++` vs. `++x`

Even more buffer/size mistakes

- ▣ Inconsistent code changes (use `sizeof`)
- ▣ Misuse of `sizeof` (e.g., on pointer)
- ▣ Bytes vs. wide chars (UCS-2) vs. multibyte chars (UTF-8)
- ▣ OS length limits (or lack thereof)

Other array problems

- ▣ Missing/wrong bounds check
 - One unsigned comparison suffices
 - Two signed comparisons needed
- ▣ Beware of clever loops
 - Premature optimization

Outline

Vulnerabilities in OS interaction

Low-level view of memory

Logistics announcements

Basic memory-safety problems

Where overflows come from

More problems

Integer overflow

- ▣ Fixed size result \neq math result
- ▣ Sum of two positive ints negative or less than addend
- ▣ Also multiplication, left shift, etc.
- ▣ Negation of most-negative value
- ▣ $(low + high)/2$

Integer overflow example

```
int n = read_int();
obj *p = malloc(n * sizeof(obj));
for (i = 0; i < n; i++)
    p[i] = read_obj();
```

Signed and unsigned

- Unsigned gives more range for, e.g., `size_t`
- At machine level, many but not all operations are the same
- Most important difference: ordering
- In C, signed overflow is **undefined behavior**

Mixing integer sizes

- Complicated rules for implicit conversions
 - Also includes signed vs. unsigned
- Generally, convert before operation:
 - E.g., `1ULL << 63`
- Sign-extend vs. zero-extend
 - `char c = 0xff; (int)c`

Null pointers

- Vanilla null dereference is usually non-exploitable (just a DoS)
- But not if there could be an offset (e.g., field of struct)
- And not in the kernel if an untrusted user has allocated the zero page

Undefined behavior

- C standard "undefined behavior": **anything** could happen
- Can be unexpectedly bad for security
- Most common problem: compiler optimizes assuming undefined behavior cannot happen

Linux kernel example

```
struct sock *sk = tun->sk;
// ...
if (!tun)
    return POLLERR;
// more uses of tun and sk
```

Format strings

- `printf` format strings are a little interpreter
- `printf(fmt)` with untrusted `fmt` lets the attacker program it
- Allows:
 - Dumping stack contents
 - Denial of service
 - Arbitrary memory modifications!

Next time

- 📄 Exploitation techniques for these vulnerabilities