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

Low-level view of memory

Logistics announcements

Basic memory-safety problems

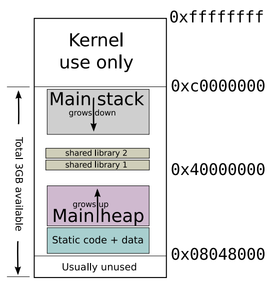
Where overflows come from

More problems

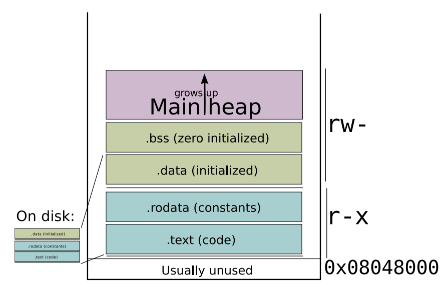
Note on x86-32 and x86-64

- 32-bit and 64-bit x86 have many similarities, but some differences
- 64-bit now more common for big systems
  - 32-bit architectures still common in embedded systems, e.g. 32-bit ARM
- This year's HA1 will still have a 32-bit vulnerable binary
  - Makes some attacks easier
  - Less translation for classic vulnerability and attack descriptions

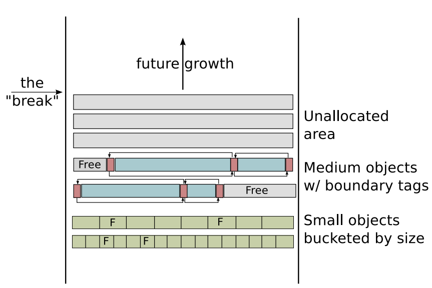
Overall layout (Linux 32-bit)



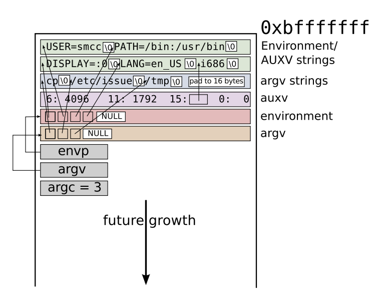
Detail: static code and data



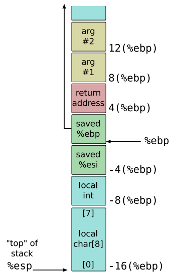
Detail: heap



Detail: initial stack



Example stack frame



Outline

Low-level view of memory

Logistics announcements

Basic memory-safety problems

Where overflows come from

More problems

Outline

Low-level view of memory

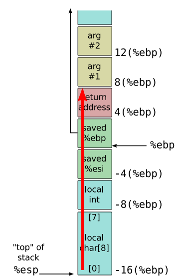
Logistics announcements

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

- 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 `strlcpy`, `strlcat`
  - Easier to use safely than "n" versions
  - Non-standard, but widely copied
- Microsoft-pushed `strcpy_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

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 `int`s 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