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

Stephen McCamant

University of Minnesota, Computer Science & Engineering

Outline

Vulnerabilities in OS interaction

Low-level view of memory

HA1 logistics, etc.

Basic memory-safety problems

Where overflows come from

More problems

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

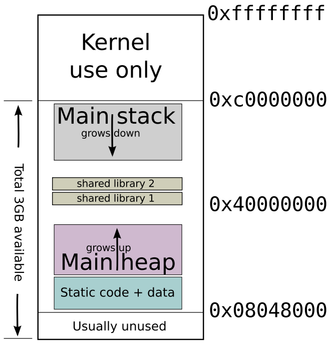
IFS and why it's a problem

- 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")`

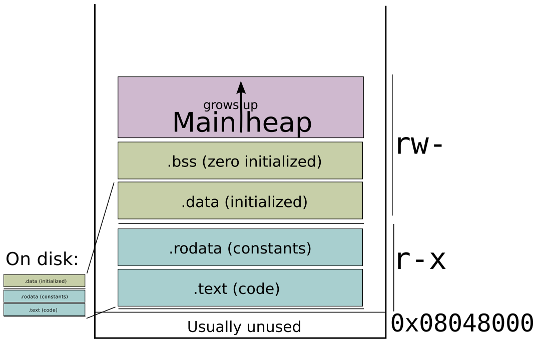
Outline

- Vulnerabilities in OS interaction
- Low-level view of memory
- HA1 logistics, etc.
- Basic memory-safety problems
- Where overflows come from
- More problems

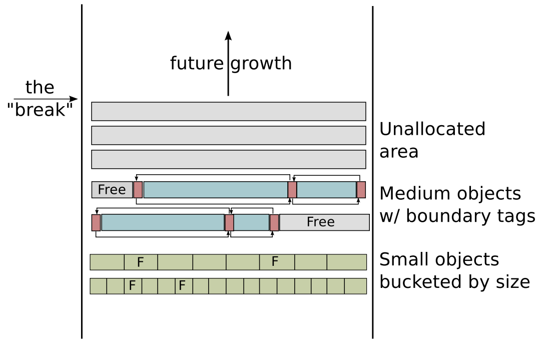
Overall layout (Linux 32-bit)



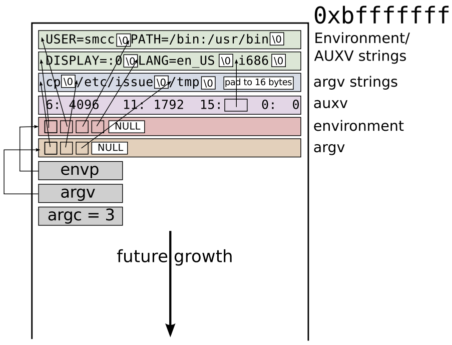
Detail: static code and data



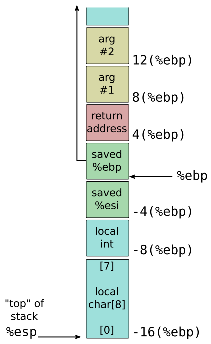
Detail: heap


future growth
the "break"
Unallocated area
Free
Medium objects w/ boundary tags
Free
F
F
F
F
Small objects bucketed by size

Detail: initial stack


0xbfffffff
USER=smcc \0 PATH=/bin:/usr/bin \0
DISPLAY=:0 \0 LANG=en_US \0 i686 \0
cp \0 /etc/issue \0 /tmp \0 pad to 16 bytes
6: 4096 11: 1792 15: 0: 0
NULL
NULL
envp
argv
argc = 3
Environment/ AUXV strings
argv strings
auxv
environment
argv
future growth

Example stack frame



Outline

Vulnerabilities in OS interaction

Low-level view of memory

HA1 logistics, etc.

Basic memory-safety problems

Where overflows come from

More problems

HA1 materials posted

- Instructions PDF
- BCVI source code
- VM instructions web page
- Discussion forum and submissions on Moodle

Getting your virtual machines

- Ubuntu 16.04 server, hosted on CSE Labs
  - 64-bit kernel but 32-bit BCVI, `gcc -m32`
- One VM per group (up to 3 students)
- For allocation, send group list to Se Eun
- Don't put off until the last minute

Sequence of exploits

- Week 1 (9/15): bad feature, 10 points
- Week 2 (9/22): easier, 20 points
- Week 3 (9/29): harder, 30 points
- Week 4 (10/6): harder, 30 points
  - Plus, design suggestions (10 points)
- Week 5 (10/13): hardest, $10 \cdot n$ extra credit

Types of vulnerabilities

- OS interaction/logic errors
- Memory safety errors
  - E.g., exploit with control-flow hijacking
- Attacks may require crafted text files and chosen program inputs

Part of challenge: automation

- Must represent your attack as an exploit script
- Must be fully automatic
  - No user interaction
  - Works reliably, within 60 seconds
- Must work on a clean VM
- Use `test-exploit` script

Still coming soon

- Research project pre-proposal due a week from today

Outline

Vulnerabilities in OS interaction

Low-level view of memory

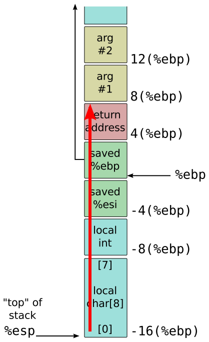
HA1 logistics, etc.

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

HA1 logistics, etc.

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

Vulnerabilities in OS interaction

Low-level view of memory

HA1 logistics, etc.

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