CSci 4271W
Development of Secure Software Systems
Day 13: OS-level Injection Threats

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Outline

Injection vulnerabilities

- Common dangerous pattern: interpreter code with attacker control
- Interpreted language example: `eval`
- OS example: shell script injection
- Web examples: JavaScript (XSS), SQL injection
- C library example: `printf` format string

Format string attack: overwrite

- `%n` specifier: store number of chars written so far to pointer arg
  - Benign but uncommon use: account for length in other formatting
- Advance format arg pointer to other attacker-controlled data
- Control number of chars written with padding
- Net result is a "write-what-where" primitive

Practical format string challenges

- Attacker usually must control format as well as one or more arguments
- Writing a big value requires impractical output size
  - Workaround 1: overwrite two bytes with `%hn`
  - Workaround 2: use overlapping unaligned write to control byte by byte

Format string defenses

- Compilers will warn for `printf` that looks like it should just be `puts`
- Several platforms have decided to just remove `%n`
  - Android Bionic, Visual Studio
- Linux glibc by default will block `%n` if the format string is writeable
- Major remaining use is information disclosure

Demo: first steps of BCLPR format attack

- In demo: quick audit, supplying format

Outline

Two kinds of privilege escalation

- Local exploit: give higher privilege to a regular user
  - E.g., caused by bug in setuid program or OS kernel
- Remote exploit: give access to an external user who doesn't even have an account
  - E.g., caused by bug in network-facing server or client

Shell code injection

- The command shell is convenient to use, especially in scripts
  - In C: `system`, `popen`
- But it is bad to expose the shell's power to an attacker
- Key pitfall: assembling shell commands as strings
- Note: different from binary "shellcode"

Shell code injection example

- Benign: `system("cp $arg1 $arg2")`, arg1 = `"file1.txt"`
- Attack: arg1 = `"a b; echo Gotcha"`
- Command: `"cp a b; echo Gotcha file2.txt"`
- Not a complete solution: prohibit `';'`

The structure problem

- What went wrong here?
- Basic mistake: assuming string concatenation will respect language grammar
  - E.g., that attacker supplied "filename" will be interpreted that way

Best fix: avoiding the shell

- Avoid letting untrusted data get near a shell
- For instance, call external programs with lower-level interfaces
  - E.g., `fork` and `exec` instead of `system`
- May constitute a security/flexibility trade-off

Less reliable: text processing

- Allow-list: known-good characters are allowed, others prohibited
  - E.g., username consists only of letters
  - Safest, but potential functionality cost
- Deny-list: known-bad characters are prohibited, others allowed
  - Easy to miss some bad scenarios
- "Sanitization": transform bad characters into good
  - Same problem as deny-list, plus extra complexity

Terminology note

- Historically the most common terms for allow-list and deny-list have been "whitelist" and "blacklist" respectively
- These terms have been criticized for a problematic "white=good", "black=bad" association
- The push to avoid the terms got significant additional attention in summer 2020, but is still somewhat political and in flux

Different shells and multiple interpretation

- Complex Unix systems include shells at multiple levels, making these issues more complex
  - Frequent example: `scp` runs a shell on the server, so filenames with whitespace need double escaping
- Other shell-like programs also have caveats with levels of interpretation
  - Tcl before version 9 interpreted leading zeros as octal

Related local dangers

- File names might contain any character except / or the null character
- The `PATH` environment variable is user-controllable, so `cp` may not be the program you expect
- Environment variables controlling the dynamic loader cause other code to be loaded

IFS and why it was 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")`
- In modern shells, improved by not taking from environment

Outline

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 ⇒ run untrusted code anyway
- Update file when disk full ⇒ 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

Read It Twice (WOOT'12)

- Smart TV (running Linux) only accepts signed apps on USB sticks

1. Check signature on file
2. Install file

- Malicious USB device replaces app between steps
- TV "rooted"/"jailbroken"

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

Directory traversal with . .

- Program argument specifies file, found in directory `files`
- What about `files/../../../../etc/passwd`?