

CSci 4271W  
Development of Secure Software Systems  
Day 12: More Permissions, and OS-level Injection Threats

Stephen McCamant  
University of Minnesota, Computer Science & Engineering

## Outline

- Exercise: using Unix permissions
- More Unix permissions
- Shell code injection and related threats

## Octal digits represent access

- 7 = rwx
- 6 = rw
- 5 = rx
- 4 = r
- 0 = no access

## Setting: files related to this class

- Student and course staff materials
- Imagine everything is in Unix files on CSE Labs
  - Versus reality of a mixture of Unix with web-based systems like Canvas

## Users and groups

- Users: smccaman (instructor), pakki001 (TA), stude003 (student)
- Groups: csci4271staff (instructor and TAs), csci4271all (staff and students)

## What I want from you

- Brainstorm sets of octal permissions bits that could be used
- For each permission bits set, give user, owner, and file/directory contents/use that would be sensible

## Outline

- Exercise: using Unix permissions
- More Unix permissions
- Shell code injection and related threats

## Process UIDs and `setuid(2)`

- UID is inherited by child processes, and an unprivileged process can't change it
- But there are syscalls root can use to change the UID, starting with `setuid`
- E.g., login program, SSH server

## Setuid programs, different UIDs

- If 04000 "setuid" bit set, newly exec'd process will take UID of its file owner
  - Other side conditions, like process not traced
- Specifically the *effective UID* is changed, while the *real UID* is unchanged
  - Shows who called you, allows switching back

## More different UIDs

- Two mechanisms for temporary switching:
  - Swap real UID and effective UID (BSD)
  - Remember *saved UID*, allow switching to it (System V)
- Modern systems support both mechanisms at the same time

## Setgid, games

- Setgid bit 02000 mostly analogous to setuid
- But note no supergroup, so UID 0 is still special
- Classic application: setgid `games` for managing high-score files

## Special case: `/tmp`

- We'd like to allow anyone to make files in `/tmp`
- So, everyone should have write permission
- But don't want Alice deleting Bob's files
- Solution: "sticky bit" 01000

## Special case: group inheritance

- When using group to manage permissions, want a whole tree to have a single group
- When 02000 bit set, newly created entries will have the parent's group
  - (Historic BSD behavior)
- Also, directories will themselves inherit 02000

## Other permission rules

- Only file owner or root can change permissions
- Only root can change file owner
  - Former System V behavior: "give away `chown`"
- Setuid/gid bits cleared on `chown`
  - Set owner first, then enable setuid

## Outline

Exercise: using Unix permissions

More Unix permissions

Shell code injection and related threats

## 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