Outline

Injection vulnerabilities: format strings (cont'd)
Shell code injection and related threats
Print server threat modeling
Good technical writing (pt. 1)

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

Injection vulnerabilities: format strings (cont'd)
Shell code injection and related threats
Print server threat modeling
Good technical writing (pt. 1)
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 last summer, 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 \rightarrow 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

Injection vulnerabilities: format strings (cont'd)
Shell code injection and related threats
Print server threat modeling
Good technical writing (pt. 1)

Data flows and trust boundaries

- Interactive in drawing program

Writing in CS versus other writing

- Key goal is accurately conveying precise technical information
- More important: careful use of terminology, structured organization
- Less important: writer's personality, persuasion, appeals to emotion

Still important: concise expression

- Don't use long words or complicated expressions when simpler ones would convey the same meaning. Examples:
  - necessitate
  - utilize
  - due to the fact that
- Beneficial for both clarity and style

Know your audience: terminology

- When technical terminology makes your point clearly, use it
- But provide definitions if a concept might be new to many readers
  - Be careful to provide the right information in the definition
  - Define at the first instead of a later use
- On other hand, avoid introducing too many new terms
  - Keep the same term when referring to the same concept
Precise explanations
- Don't say "we" do something when it's the computer that does it
- And avoid passive constructions
- Don't anthropomorphize (computers don't "know")
- Use singular by default so plural provides a distinction:
  - The students take tests
  - Each student takes a test
  - Each student takes multiple tests

Provide structure
- Use plenty of sections and sub-sections
- It's OK to have some redundancy in previewing structure
- Limit each paragraph to one concept, and not too long
  - Start with a clear topic sentence
  - Split long, complex sentences into separate ones

Know your audience: Project 1
- For projects in this course, assume your audience is another student who already understands general course concepts
  - Up to the current point in the course
  - I.e., don't need to define "buffer overflow" from scratch
- But you need to explain specifics of a vulnerable program
  - Make clear what part of the program you're referring to
  - Explain all the specific details of a vulnerability

Inclusive language
- Avoid words and grammar that implies relevant people are male
- My opinion: avoid using he/him pronouns for unknown people
- Some possible alternatives
  - "he/she"
  - Alternating genders
  - Rewrite to plural and use "they" (may be less clear)
  - Singular "they" (least traditional, but spreading)