Preview question

What's the type of the return value of getchar?
Why?

Development of Secure Software Systems Day 23: Software Engineering and Security Stephen McCamant University of Minnesota, Computer Science & Engineering

CSci 4271W

Outline

Software engineering for security

bcimgview project introduction

Fuzz testing

Saltzer & Schroeder's principles

More secure design principles

Defensive programming

- Analogy to defensive driving: drive so that there won't be a crash even if other drivers are negligent
- 🖲 Don't just avoid bugs, reduce risks
- Aim for security even if other code and programmers are imperfect

Modularity

Divide software into pieces with well-defined functionality

Isolate security-critical code

Minimize TCB, facilitate privilege separation

Improve auditability

Minimize interfaces

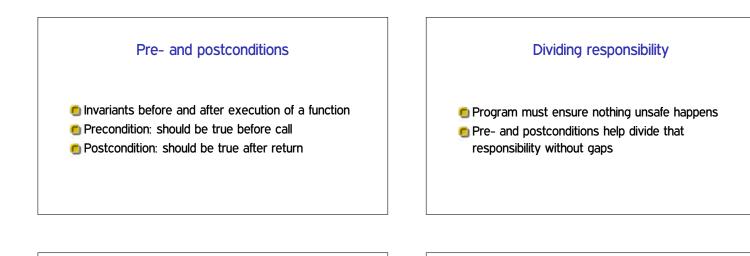
- Hallmark of good modularity: clean interface
- Particularly difficult:
 - Safely implementing an interface for malicious users
 - Safely using an interface with a malicious implementation

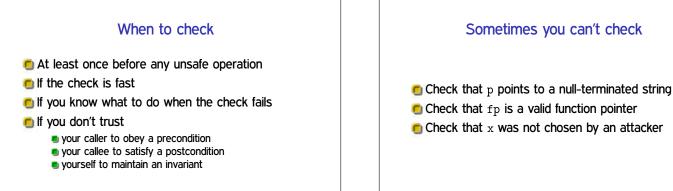
Appropriate paranoia Many security problems come down to missing checks But, it isn't possible to check everything continuously How do you know when to check what?

A fact about the state of a program that should always be maintained

Invariant

- Assumed in one place to guarantee in another
- Compare: proof by induction





Error handling

Every error must be handled

 I.e, program must take an appropriate response action

 Errors can indicate bugs, precondition violations, or situations in the environment

Error codes

Commonly, return value indicates error if any Bad: may overlap with regular result

Bad: goes away if ignored

Exceptions

Separate from data, triggers jump to handler

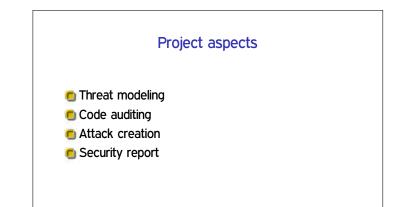
- Good: avoid need for manual copying, not dropped
- May support: automatic cleanup (finally)
- Bad: non-local control flow can be surprising

Outline

Software engineering for security

bcimgview project introduction

- Fuzz testing
- Saltzer & Schroeder's principles
- More secure design principles



Project scenario

- Benign but buggy image viewer
- Key threat class: opening untrusted images
 Imagine web or email downloads
 Similar to many historical problems

Project logistics Individual project Handout and code to be posted by tonight

Early submission (feedback only) Fri 4/23

Final submission Fri 4/30

Outline

Software engineering for security

bcimgview project introduction

Fuzz testing

Saltzer & Schroeder's principles

More secure design principles

Testing and security

- "Testing shows the presence, not the absence of bugs" – Dijkstra
- Easy versions of some bugs can be found by targeted tests:
 - Buffer overflows: long strings
 - Integer overflows: large numbers
 - Format string vulnerabilities: %x

Random or fuzz testing

Random testing can also sometimes reveal bugs
 Original 'fuzz' (Miller): program </dev/urandom
 Even this was surprisingly effective

Mutational fuzzing

- Instead of totally random inputs, make small random changes to normal inputs
- Changes are called mutations
- Benign starting inputs are called seeds
- Good seeds help in exercising interesting/deep behavior

Grammar-based fuzzing

- Observation: it helps to know what correct inputs look like
- Grammar specifies legal patterns, run backwards with random choices to generate
- Generated inputs can again be basis for mutation
- Most commonly used for standard input formats
 - Network protocols, JavaScript, etc.

What if you don't have a grammar?

Input format may be unknown, or buggy and limited
 Writing a grammar may be too much manual work
 Can the structure or interesting inputs be figured out automatically?

Coverage-driven fuzzing

- Instrument code to record what code is executed
- An input is interesting if it executes code that was not executed before
- Only interesting inputs are used as basis for future mutation

AFL

- Best known open-source tool, pioneered coverage-driven fuzzing
- American Fuzzy Lop, a breed of rabbits
- Stores coverage information in a compact hash table
- Compiler-based or binary-level instrumentation
- Has a number of other optimizations

Outline

Software engineering for security

bcimgview project introduction

Fuzz testing

Saltzer & Schroeder's principles

More secure design principles

A classic paper

Jerome H. Saltzer and Michael D. Schroeder, "The Protection of Information in Computer Systems." In *Proceedings of the IEEE*, Sept. 1975. (853 citations per IEEE)

Economy of mechanism

- Security mechanisms should be as simple as possible
- Good for all software, but security software needs special scrutiny

Fail-safe defaults

- When in doubt, don't give permission
- 🖲 Whitelist, don't blacklist
- 🖲 Obvious reason: if you must fail, fail safe
- More subtle reason: incentives

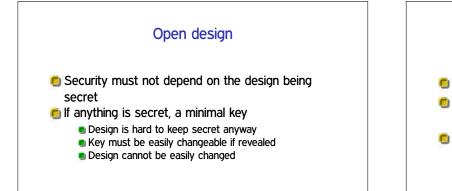
Complete mediation

Every mode of access must be checked

 Not just regular accesses: startup, maintenance, etc.

 Checks cannot be bypassed

E.g., web app must validate on server, not just client



Open design: strong version

- "The design should not be secret"
- If the design is fixed, keeping it secret can't help attackers
- But an unscrutinized design is less likely to be secure

Separation of privilege

- Real world: two-person principle
- Direct implementation: separation of duty
- Multiple mechanisms can help if they are both required
 - Password and wheel group in Unix

Least privilege

- Programs and users should have the most limited set of powers needed to do their job
- Presupposes that privileges are suitably divisible
 Contrast: Unix root

Least privilege: privilege separation

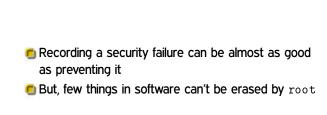
- Programs must also be divisible to avoid excess privilege
- Classic example: multi-process OpenSSH server
- **(D)** N.B.: Separation of privilege \neq privilege separation

Least common mechanism

- Minimize the code that all users must depend on for security
- Related term: minimize the Trusted Computing Base (TCB)
- E.g.: prefer library to system call; microkernel OS

 Psychological acceptability
 Sometimes: work factor

 A system must be easy to use, if users are to apply it correctly
 Make the system's model similar to the user's mental model to minimize mistakes
 E.g., length of password
 But, many attacks are easy when you know the bug



Sometimes: compromise recording

Outline

Software engineering for security beingview project introduction Fuzz testing Saltzer & Schroeder's principles

More secure design principles

Separate the control plane

- Keep metadata and code separate from untrusted data
- Bad: format string vulnerability
- Bad: old telephone systems

Defense in depth

- Multiple levels of protection can be better than one
- Especially if none is perfect
- But, many weak security mechanisms don't add up

Canonicalize names

Use unique representations of objects
 E.g. in paths, remove . , . . , extra slashes, symlinks
 E.g., use IP address instead of DNS name

Fail-safe / fail-stop

- If something goes wrong, behave in a way that's safe
- Often better to stop execution than continue in corrupted state
- E.g., better segfault than code injection