CSci 5271
Introduction to Computer Security
Day 8: Defensive programming and design, part 2

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Outline

Saltzer & Schroeder's principles
More secure design principles
Announcements intermission
Software engineering for security
Secure use of the OS
Bernstein's perspective
Techniques for privilege separation

Review from last time

- Economy of mechanism
- Fail-safe defaults
- Complete mediation

Open design

- Security must not depend on the design being secret
- If anything is secret, a minimal key
  - Design is hard to keep secret anyway
  - Key must be easily changeable if revealed
  - Design cannot be easily changed

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
- 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

- 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

Sometimes: work factor

- Cost of circumvention should match attacker and resource protected
- E.g., length of password
- But, many attacks are easy when you know the bug

Sometimes: compromise recording

- Recording a security failure can be almost as good as preventing it
- But, few things in software can't be erased by `root`

Outline

- Saltzer & Schroeder's principles
- More secure design principles
- Announcements intermission
- Software engineering for security
- Secure use of the OS
- Bernstein's perspective
- Techniques for privilege separation

Pop quiz

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

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

Outline

- Saltzer & Schroeder's principles
- More secure design principles
- Announcements intermission
- Software engineering for security
- Secure use of the OS
- Bernstein's perspective
- Techniques for privilege separation

Note to early readers

- This is the section of the slides most likely to change in the final version
- If class has already happened, make sure you have the latest slides for announcements
- In particular, the BCVI vulnerability announcement is embargoed

Alternative Saltzer & Schroeder

- Not a replacement for reading the real thing, but:
- http://emergentchaos.com/the-security-principles-of-saltzer-and-schroeder
- Security Principles of Saltzer and Schroeder, illustrated with scenes from Star Wars (Adam Shostack)

Collaboration, between groups

- Be careful: "no spoilers"
- OK to discuss general concepts
- OK to help with side tech issues
- Sharing code or written answers is never OK
- Assignment groups $\neq$ project groups

External sources

- Many assignments will allow or recommend outside (library, Internet) sources
- But you must appropriately acknowledge any outside sources you use
- Failure to do so is plagiarism

Signs something's wrong

- Getting help you wouldn't want to acknowledge
- Telling another group one-on-one something you wouldn't post to the course forum
- Turning in code where you don't understand how it works

Outline

- Saltzer & Schroeder's principles
- More secure design principles
- Announcements intermission
- Software engineering for security
- Secure use of the OS
- Bernstein's perspective
- Techniques for privilege separation

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?

Invariant

- A fact about the state of a program that should always be maintained
- Assumed in one place to guarantee in another
- Compare: proof by induction

Pre- and postconditions

- Invariants before and after execution of a function
- Precondition: should be true before call
- Postcondition: should be true after return

Dividing responsibility

- Program must ensure nothing unsafe happens
- Pre- and postconditions help divide that responsibility without gaps

When to check

- At least once before any unsafe operation
- If the check is fast
- If you know what to do when the check fails
- If you don't trust
  - your caller to obey a precondition
  - your callee to satisfy a postcondition
  - yourself to maintain an invariant

Sometimes you can't check

- Check that `p` points to a null-terminated string
- Check that `fp` is a valid function pointer
- Check that `x` was not chosen by an attacker

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

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

Fuzz testing

- Random testing can also sometimes reveal bugs
- Original ‘fuzz’ (Miller): program </dev/urandom
- Modern: small random changes to a benign input

Outline

Saltzer & Schroeder’s principles

More secure design principles

Announcements intermission

Software engineering for security

Secure use of the OS

Bernstein’s perspective

Techniques for privilege separation

Avoid special privileges

- Require users to have appropriate permissions
  - Rather than putting trust in programs
- Anti-pattern 1: setuid/setgid program
- Anti-pattern 2: privileged daemon
- But, sometimes unavoidable (e.g., email)

One slide on setuid/setgid

- Unix users and process have a user id number (UID) as well as one or more group IDs
- Normally, process has the IDs of the use who starts it
- A setuid program instead takes the UID of the program binary

Don’t use shells or Tcl

- . . . in security-sensitive applications
- String interpretation and re-parsing are very hard to do safely
- Eternal Unix code bug: path names with spaces

Prefer file descriptors

- Maintain references to files by keeping them open and using file descriptors, rather than by name
- References same contents despite file system changes
- Use `openat`, etc., variants to use FD instead of directory paths

Prefer absolute paths

- Use full paths (starting with /) for programs and files
- `$PATH` under local user control
- Initial working directory under local user control
  - But FD-like, so can be used in place of `openat` if missing

Prefer fully trusted paths

- Each directory component in a path must be write protected
- Read-only file in read-only directory can be changed if a parent directory is modified

Don't separate check from use

- Avoid pattern of e.g., `access` then `open`
- Instead, just handle failure of open
  - You have to do this anyway
- Multiple references allow races
  - And `access` also has a history of bugs

Be careful with temporary files

- Create files exclusively with tight permissions and never reopen them
  - See detailed recommendations in Wheeler
- Not quite good enough: reopen and check matching device and inode
  - Fails with sufficiently patient attack

Give up privileges

- Using appropriate combinations of `set*id` functions
  - Alas, details differ between Unix variants
- Best: give up permanently
- Second best: give up temporarily
- Detailed recommendations: Setuid Demystified (USENIX'02)

Whitelist environment variables

- Can change the behavior of called program in unexpected ways
- Decide which ones are necessary
  - As few as possible
- Save these, remove any others

Outline

Saltzer & Schroeder's principles

More secure design principles

Announcements intermission

Software engineering for security

Secure use of the OS

Bernstein's perspective

Techniques for privilege separation

Historical background

- Traditional Unix MTA: Sendmail (BSD)
  - Monolithic setuid root program
  - Designed for a more trusting era
  - In mid-90s, bugs seemed endless
- Spurred development of new, security-oriented replacements
  - Bernstein's qmail
  - Venema et al.'s Postfix

Distinctive qmail features

- Single, security-oriented developer
- Architecture with separate programs and UIDs
- Replacements for standard libraries
- Deliveries into directories rather than large files

Ineffective privilege separation

- Example: prevent Netscape DNS helper from accessing local file system
- Before: bug in DNS code
  - → read user's private files
- After: bug in DNS code
  - → inject bogus DNS results
  - → man-in-the-middle attack
  - → read user's private web data

Effective privilege separation

- Transformations with constrained I/O
- General argument: worst adversary can do is control output
  - Which is just the benign functionality
- MTA header parsing (Sendmail bug)
- `jpegtopnm` inside `xloadimage`

Eliminating bugs

- Enforce explicit data flow
- Simplify integer semantics
- Avoid parsing
- Generalize from errors to inputs

Eliminating code

- Identify common functions
- Automatically handle errors
- Reuse network tools
- Reuse access controls
- Reuse the filesystem

The "qmail security guarantee"

- $500, later $1000 offered for security bug
- Never paid out
- Issues proposed:
  - Memory exhaustion DoS
  - Overflow of signed integer indexes
- Defensiveness does not encourage more submissions

qmail today

- Originally had terms that prohibited modified redistribution
  - Now true public domain
- Latest release from Bernstein: 1998; netqmail: 2007
- Does not have large market share
- All MTAs, even Sendmail, are more secure now

Outline

Saltzer & Schroeder's principles

More secure design principles

Announcements intermission

Software engineering for security

Secure use of the OS

Bernstein's perspective

Techniques for privilege separation

Restricted languages

- Main application: code provided by untrusted parties
- Packet filters in the kernel
- JavaScript in web browsers
  - Also Java, Flash ActionScript, etc.

SFI

- Software-based Fault Isolation
- Instruction-level rewriting like (but predates) CFI
- Limit memory stores and sometimes loads
- Can't jump out except to designated points
- E.g., Google Native Client

Separate processes

- OS (and hardware) isolate one process from another
- Pay overhead for creation and communication
- System call interface allows many possibilities for mischief

System-call interposition

- Trusted process examines syscalls made by untrusted
- Implement via `ptrace` (like strace, gdb) or via kernel change
- Easy policy: deny

Interposition challenges

- Argument values can change in memory (TOCTTOU)
- OS objects can change (TOCTTOU)
- How to get canonical object identifiers?
- Interposer must accurately model kernel behavior
- Details: Garfinkel (NDSS'03)

Separate users

- Reuse OS facilities for access control
- Unit of trust: program or application
- Older example: qmail
- Newer example: Android
- Limitation: lots of things available to any user

`chroot`

- Unix system call to change root directory
- Restrict/virtualize file system access
- Only available to root
- Does not isolate other namespaces

OS-enabled containers

- One kernel, but virtualizes all namespaces
- FreeBSD jails, Linux LXC, Solaris zones, etc.
- Quite robust, but the full, fixed, kernel is in the TCB

(System) virtual machines

- Presents hardware-like interface to an untrusted kernel
- Strong isolation, full administrative complexity
- I/O interface looks like a network, etc.

Virtual machine designs

- (Type 1) hypervisor: 'superkernel' underneath VMs
- Hosted: regular OS underneath VMs
- Paravirtualizaion: modify kernels in VMs for ease of virtualization

Virtual machine technologies

- Hardware based: fastest, now common
- Partial translation: e.g., original VMware
- Full emulation: e.g. QEMU proper
  - Slowest, but can be a different CPU architecture

Modern example: Chrom(ium)

- Separates "browser kernel" from less-trusted "rendering engine"
  - Pragmatic, keeps high-risk components together
- Experimented with various Windows and Linux sandboxing techniques
- Blocked 70% of historic vulnerabilities, not all new ones
- `http://seclab.stanford.edu/websec/chromium/`

Next time

- Protection and isolation
- Basic (e.g., classic Unix) access control