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 (cont’d)
More secure design principles
Software engineering for security
Announcements intermission
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 (cont'd)
- More secure design principles
- Software engineering for security
- Announcements intermission
- 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 (cont’d)
More secure design principles
Software engineering for security
Announcements intermission
Bernstein’s perspective
Techniques for privilege separation

BCMTA pre-release posted
Version 0.9 of source code includes most features and many vulnerabilities
This version includes a pretty obvious back-door which will be the first problem to be fixed
Can’t properly test without VM, but you can start reading the code
Reminder: register groups for a VM

What is BCMTA?
A badly coded mail-transfer agent, similar to sendmail or qmail
- Can run from the command-line
- Can receive messages over the network (SMTP on standard input)
- Needs to run as root to deliver to any user’s mailbox
  - Attacker’s goal: use root privilege to take over machine
  - Specifically: root shell

HA1 types of vulnerabilities
- OS interaction/logic errors
- Memory safety errors
  - E.g., exploit with control-flow hijacking
- Attacks may require crafted text files and chosen program inputs
Other upcoming assignments

- Project progress reports: due next Monday 2/25
  - Remember, these are individual
- Exercise set 2: due week from Wednesday, 2/27

Outline

- Saltzer & Schroeder’s principles (cont’d)
- More secure design principles
- Software engineering for security
- Announcements intermission
- 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
- Does not have large market share
- All MTAs, even Sendmail, are more secure now

Outline
- Saltzer & Schroeder’s principles (cont’d)
- More secure design principles
- Software engineering for security
- Announcements intermission
- 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
- Paravirtualization: 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

## Next time

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