

CSci 4271W  
Development of Secure Software Systems  
Day 14: OS Protection and Isolation

Stephen McCamant  
University of Minnesota, Computer Science & Engineering

## Outline

- Secure OS interaction
- Announcements intermission
- OS: protection and isolation
- More choices for isolation
- Testing and fuzzing

## Avoid special privileges

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

## 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 (q.v.)
- 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)

## Allow-list 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

## For more details...

- The first external reading is chapters from a web-hosted book by David A. Wheeler
- Reading questions will be due one week after they are posted on Canvas
  - In this case, next Wednesday

## Outline

Secure OS interaction  
Announcements intermission  
OS: protection and isolation  
More choices for isolation  
Testing and fuzzing

## Upcoming activities

- Problem set 1 is due Friday at 11:59pm
- Watch Piazza for midterm solution information, probably ready to give back in class on Tuesday
- First reading assignment, about OS interaction, available now with quiz due in one week

## Outline

Secure OS interaction  
Announcements intermission  
OS: protection and isolation  
More choices for isolation  
Testing and fuzzing

## OS security topics

- Resource protection
- Process isolation
- User authentication (will cover later)
- Access control (already covered)

## Protection and isolation

- Resource protection: prevent processes from accessing hardware
- Process isolation: prevent processes from interfering with each other
- Design: by default processes can do neither
- Must request access from operating system

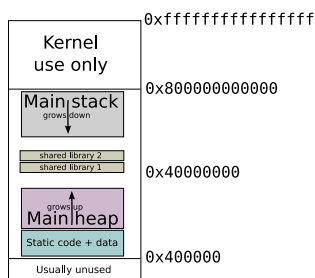
## Reference monitor

- Complete mediation: all accesses are checked
- Tamperproof: the monitor is itself protected from modification
- Small enough to be thoroughly verified

## Hardware basis: memory protection

- Historic: segments
- Modern: paging and page protection
  - Memory divided into pages (e.g. 4k)
  - Every process has own virtual to physical page table
  - Pages also have R/W/X permissions

## Linux example



## Hardware basis: supervisor bit

- Supervisor (kernel) mode: all instructions available
- User mode: no hardware or VM control instructions
- Only way to switch to kernel mode is specified entry point
- Also generalizes to multiple "rings"

## Outline

- Secure OS interaction
- Announcements intermission
- OS: protection and isolation
- More choices for isolation
- Testing and fuzzing

## Ideal: 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`

## "Trusted", TCB

- In security, "trusted" is a bad word
- X is trusted: X can break your security
- "Untrusted" = okay if it's evil
- Trusted Computing Base (TCB): minimize

## 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
  - Analogous to but predates control-flow integrity
- 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
- <http://seclab.stanford.edu/websec/chromium/>

## Outline

Secure OS interaction  
Announcements intermission  
OS: protection and isolation  
More choices for isolation  
Testing and fuzzing

## 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 of 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