Sandboxing

Dawn Song
dawnsong@cs.berkeley.edu

Review

- **Preventing privilege escalation**
  - **Drop privileges asap**
  - **Privilege separation**
- **Isolation**
  - **Hardware fault isolation**
  - **Software fault isolation**

Software Fault Isolation

- **Idea: insert code in extension code to ensure certain security properties**
- **SFI [Wahbe et. al. 93]**
  - **Software fault isolation**
  - **Security property to guarantee: Extension code only writes and jumps to dedicated data and code region**
  - **How to ensure this?**

Segments

- Divide application's virtual address space into segments
  - With upper bits the same: segment identifier
- A fault domain has two segments
  - Code segments
  - Data segments
- Security property to ensure
  - Distrusted code only jumps to its code segment, only writes to its data segment

Idea

- Before every write and jump, insert code to check whether the target is within the dedicated region
- Optimization:
  - instead of checking, simply sets the high-order bits to be segment identifier
- Where to store the value of the masks?
  - Dedicated registers
- How to prevent jumping over the inserted check code?
  - Use dedicated registers

Sandboxing Untrusted Store

dedicated-reg <= target-reg & and-mask-reg
dedicated-reg <= dedicated-reg | segment-reg
store instruction uses dedicated-reg

- Sequence of instructions for each untrusted store
- Untrusted jump instruction handled similarly

Why Use Dedicated Register?

- How many dedicated register required?
- Why?
- What happens is untrusted code jumps to the middle of the sequence?

Instrumentation and Verification

- Instrumentation
  - Modify gcc compiler to emit encapsulated object code
- Verification
  - Verify when module is loaded
  - Why verification?
    - Module is untrusted
    - Verifier can be much simpler than the instrumentor
  - How to verify?

SFI Summary

- Security property ensured:
  Distrusted code only jumps to its code segment, only writes to its data segment
- Tradeoff btw computation overhead & communication overhead
- More information:
  - Efficient Software-based Fault Isolation, by Robert Wahbe, Steven Lucco, Thomas Anderson, Susan Graham

Generalization: In-line Reference Monitor

- In-line reference monitors/dynamic checks
  - IRMs enforce security policies by inserting into subject programs the code for validity checks and also any additional state that is needed for enforcement
- Idea
  - Add dynamic checks to enforce properties at run time
  - Combine with static analysis to reduce dynamic checks
  - Ensure dynamic checks are not by-passed
    - » Control & data property enforcements are intertwined
  - Verifier:
    - » Ensure dynamic checks are properly inlined

A Whole Spectrum

- Tradeoff
  - Complexity of properties enforced
  - Runtime overhead
  - Assumptions required
  - Complexity of priori analysis needed
- Properties enforced entail
  - What dynamic checks to add
  - How to add these dynamic checks
- The spectrum
  - SFI, CFI, DFI, XFI, …
  - Interpreter/emulator is one end of the spectrum

Move to a different level

- System call interposition for application sandboxing

Administravia

- **HW4 Stats:**
  - **Max 75 (out of 75)**
  - **Mean 54, s.d. 20**
  - **Median 61**

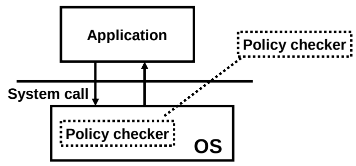
System Call Interposition

- **Malicious programs usually need to make system calls to do harm to the system**
- **System call interface is a natual place to place security checks & enforce security policies**
- **What kind of policies do we want to enforce?**
  - **A process cannot open certain files**
  - **A process may have restricted network access**
  - **A process may not send network packets after reading certain files**

How to Get the Policy

- **Manually define policy**
- **Learning policy from past good executions**
  - **Sequence of system calls**
- **Extracting policy from the program**
  - **Push-down automata, etc.**

How to Enforce a Policy?



- Intercept system calls
- Information passed on to policy checker before system call is processed
- Policy checker
  - In kernel
  - User space

Evasion Attacks

- Be careful with race conditions (TOCTTOU)
- Mimicry attacks
  - Given sequences of allowed system calls
  - One could potentially find a sequence of system calls that performs malicious tasks and yet fly under the radar

Evasion Attacks

- wuftpd

read() write() close() munmap() sigprocmask() wait4()
sigprocmask() sigaction() alarm() time() stat() read()
alarm() sigprocmask() *setreuid()* fstat() getpid()
time() write() time() getpid() sigaction() socketcall()
sigaction() close() flock() getpid() lseek() read()
kill() lseek() flock() sigaction() alarm() time()
stat() write() open() fstat() mmap() read() open()
fstat() mmap() read() close() munmap() brk() fcntl()
setregid() open() fcntl() *chroot()* *chdir()* setreuid()
lstat() lstat() lstat() lstat() open() fcntl() fstat()
lseek() getdents() fcntl() fstat() lseek() getdents()
close() write() time() open() fstat() mmap() read()
close() munmap() brk() fcntl() setregid() open() fcntl()
chroot() chdir() setreuid() lstat() lstat() lstat()
lstat() *open()* fcntl() brk() fstat() lseek() getdents()
lseek() getdents() time() stat() *write()* time() open()
getpid() sigaction() socketcall() sigaction() umask()
sigaction() alarm() time() stat() read() alarm()
getrlimit() pipe() fork() fcntl() fstat() mmap() lseek()
close() brk() time() getpid() sigaction() socketcall()
sigaction() chdir() sigaction() sigaction() write()
munmap() munmap() munmap() *exit()*

How to Protect Policy Checker?

- In different user process or in kernel
- Relying on the trust to kernel
- Can we do better?

Virtual Machine Monitors

- Virtual machine: execution envrionment that gives the illusion of a real machine
- VMM
  - sits below OS
  - Much smaller than OS, easier to verify/get right
  - Natual place to enforce security policies
  - Policy checker does not need to rely on OS