CS155

Running Unreliable Code

John Mitchell

Topic

- How can you run code that could contain a dangerous bug or security vulnerability?
- Examples:
  - Run web server, may have buffer overflow attack
  - Run music player, may export your files to network

Several Historical Applications

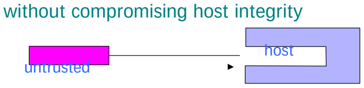
- Test and debug system code
  - Contain or monitor execution to find bugs
- Extensible Operating Systems
  - Modern trend toward smaller kernel, more functionality provided by user
- Untrusted code from network
  - Download from web
  - Code installed by browser
- Secure System Composition
  - Want to construct a secure system from mix of highly assured components and COTS

Many uses for extensibility

- OS Kernel
- Web browser
- Routers, switches, active networks
- Servers, repositories

Common problem:

- Give untrusted code limited access to resources, without compromising host integrity



This lecture

- Conventional OS: chroot and jail
- Four approaches for compiled code
  - Code modification for run-time monitoring
  - System call interposition
  - Proof-carrying code
  - Virtual machines (e.g., VMWare)
- Next lecture
  - Browser security
  - Java security

Conventional OS

- Keep processes separate
  - Each process runs in separate address space
  - Critical resources accessed through systems calls
    - File system, network, etc.
- Additional containment options
  - chroot
  - jail

Unix chroot

- chroot changes root directory
  - Originally used to test system code "safely"
  - Confines code to limited portion of file system
- Example use
  - chdir /tmp/ghostview
  - chroot /tmp/ghostview
  - su tmpuser (or su nobody)
- Caution
  - chroot changes root directory, but not current dir
    - If forget chdir, program can escape from changed root
  - If you forget to change UID, process could escape

Only root should execute chroot

- Otherwise, jailed program can escape
  - mkdir(/temp) /* create temp directory */
  - chroot(/temp) /* now current dir is outside jail */
  - chdir(" ../../../.") /* move current dir to true root dir */
  - chroot(".") /* out of jail */

  Note: this is implementation dependent
- Otherwise, anyone can become root
  - Create bogus file /tmp/etc/passwd
  - Do chroot("/tmp")
  - Run login or su (if exists in jail)

History: In Ultrix 4.0, chroot could be executed by anyone

Free BSD jail command

- Example
  - jail apache
- Stronger than chroot
  - Calls chroot
  - Also limits what root can do
    - Each jail is bound to a single IP address
      - processes within the jail may not make use of any other IP address for outgoing or incoming connections
    - Can only interact with other processes in same jail

Problems with chroot, jail approach

- Too coarse
  - Confine program to directory
    - but this directory may not contain utilities that program needs to call
  - Copy utilities into restricted environment
    - but then this begins to defeat purpose of restricted environment by providing dangerous capabilities
- Does not monitor network access
- No fine grained access control
  - Want to allow access to some files but not others

Extra programs needed in jail

- Files needed for /bin/sh

/usr/ld.so.1	shared object libraries
/dev/zero	clear memory used by shared objects
/usr/lib/libc.so.1	general C library
/usr/lib/libdl.so.1	dynamic linking access library
/usr/lib/libw.so.1	Internationalization library
/usr/lib/libintl.so.1	Internationalization library
Some others	

- Files needed for perl
  - 2610 files and 192 directories

How can we get better protection?

- Goals
  - Finer-grained protection
    - Enforce more sophisticated policies than "every process can only execute own instructions and touch own memory"
  - More efficient fault isolation
- Relevant security principles
  - Compartmentalize
  - Least privilege
  - Defense in depth

Rest of lecture

- System Monitoring
  - Software Fault Isolation
    - Modify binaries to catch memory errors
  - Wrap/trap system calls
    - Check interaction between application and OS
  - Theoretical and practical limit: safety properties
- Check code before execution
  - Proof-carrying code
    - Allow supplier to provide checkable proof
- Virtual Machines (e.g., VMWare; JVM next lecture)
  - Wrap OS with additional security layer

Software Fault Isolation (SFI)

- Wahbe, Lucco, Anderson, Graham [SOSP'93]
  - Collusa Software (founded '94, bought by Microsoft '96)
- Multiple applications in same address space
- Prevent interference from memory read/write
- Example
  - Web browser: shockwave plug-in should not be able to read credit-card numbers from other pages in browser cache

SFI is old idea in OS, made obsolete by hardware support for separate process address spaces, now considered for performance, extensible OS

Why software protection?

- Compartmentalize and use least privilege

  More compartmentalization

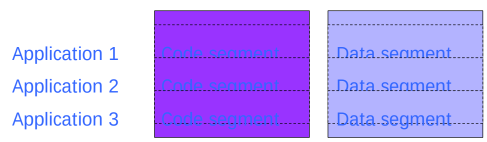
  ⇒ More processes if each is separate process

  ⇒ More context switches and inter-process communication

- Useful to achieve OS-level protection (or better) without overhead of OS context switch

SFI idea:

- Partition memory space into segments



- Add instructions to binary executables
  - Check every jump and memory access
  - Target location must be in correct segment
    - All locations in segment have same high-order bits

Slide credit: Alex Aiken

Check jumps and memory access

- Consider writes (Jumps are a little simpler)
- Replace each write by the sequence:

  dedicated-reg ⇐ target address
  scratch-reg ⇐ (dedicated-reg >> shift-size)
  scratch-reg == segment-reg
  trap if not equal
  store through dedicated-reg

- This requires several registers:
  - Dedicated-reg holds the address being computed
    - Needed in case code jumps into middle of instrumentation
  - Segment-reg hold current valid segment
  - Shift-size holds the size of the shift to perform

A Faster Approach

- Skip test; Just overwrite segment bits

  dedicated-reg ⇐ target-reg & mask-reg
  dedicated-reg ⇐ dedicated-reg | segment-reg
  store through dedicated-reg

- Tradeoffs
  - Much faster
    - Only two instructions per instrumentation point
  - Loses information about errors
    - Program may keep running with incorrect instructions and data
  - Uses five registers
    - 2 for code/data segment, 2 for code/data sandboxed addresses, 1 for segment mask

Optimizations

- Use static analysis to omit some tests
  - Writes to static variables
  - Small jumps relative to program counter
- Allocate larger segments to simplify calculations
  - Some references can fall outside of segment
  - Requires unused buffer regions around segments
  - Example: In load w/offset, sandbox register only
    - Sandboxing reg+offset requires one additional operation

When are tests added to code?

- Two options
  - Binary instrumentation
    - Most portable & easily deployed
    - Harder to implement
  - Modified compiler
    - Requires source code
    - But easier to implement
- Decision: modified compiler

Results

- Works pretty well
  - Overhead · 10% on nearly all benchmarks
  - Often significantly less (4%?)
- Provides limited protection
  - Protects memory of host code
    - does not trap system calls that could cause problems, etc.
  - Extension code unprotected from itself

Sequoia DB benchmarks:
2-7% overhead for SFI, 18-40% overhead for OS

More on Jumps

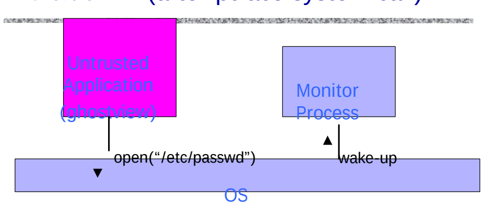
- PC-relative jumps are easy:
  - just adjust to the new instruction's offset.
- Computed jumps are not:
  - must ensure code doesn't jump *into* or *around* a check or else that it's *safe* for code to do the jump.
  - for SFI paper, they ensured the latter:
    - a dedicated register is used to hold the address that's going to be written – so all writes are done using this register.
    - only inserted code changes this value, and it's always changed (atomically) with a value that's in the data segment.
    - so at all times, the address is "valid" for writing.
    - works with little overhead for almost all computed jumps.

Slide credit: Alex Aiken

Wrap or trap system calls

- Several projects, e.g., Janus (Berkeley)
- Trap system calls
  - Check parameters, deny unauthorized calls
  - Enforce mandatory access control in OS that does not provide mandatory access control
- Two approaches in Unix and variants
  - ptrace system call - register a callback that will be called whenever application makes a system call
  - /proc virtual file system under Solaris
- System-independent approach
  - Wrap system calls

Ptrace (after ptrace system call)



- Problems
  - Coarse: trace all calls or none
  - Limited error handling
    - Cannot abort system call without killing service

Note: Janus used ptrace initially, later discarded ...

/proc virtual file system under Solaris

- Can trap selected system calls
  - obtain arguments to system calls
  - can cause system call to fail with errnm = EINTR
  - application can handle failed system call in whatever way it was designed to handle this condition
- Parallelism (for ptrace and /proc)
  - If service process forks, need to fork monitor => must monitor fork calls

Hard part

- Design good policy for allow, deny, test args
  - Example choice of system calls

Deny	mount, setuid
Allow	close, exit, fork, read, write
Test	open, rename, stat, kill

  - Example policy for file args

    path allow read, write /tmp/*
    path deny /etc/passwd
    network deny all, allow XDisplay

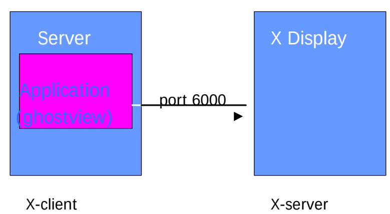
Counterintuitive, but OK if file is OK

Example: trapping X

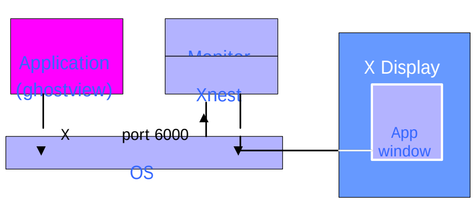
- Application, such as ghostscript
  - Can open X window, needs to make X windows calls
  - However, X allows some attacks;
    - do not want ghostscript/ghostview to read characters you type in any window
- Solution
  - X proxy called Xnest
    - application, redirected through Xnest, only has access to small nested window on display

Note: Xnest usually runs in firewall

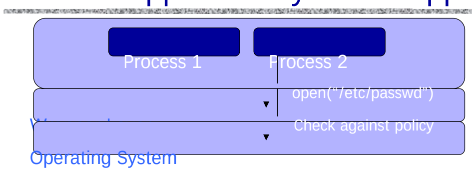
X Communication



Xnest



Another approach: syscall wrapper



- System available from TIS (NAI) 1999
  - wrapper description language
  - easy to implement policy on system calls (policy language)
  - works with many OSs
- Similar idea: TCP wrapper

Garfinkel: Interposition traps and pitfalls

- Incorrectly replicating OS semantics
  - Incorrectly mirroring OS state
  - Incorrectly mirroring OS code
- Overlooking indirect paths to resources
- Race conditions
  - Symbolic link races
  - Relative path races
  - Argument races
  - more ...
- Side effects of denying system calls

Many projects on monitoring

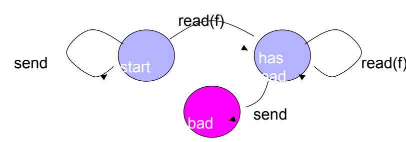
- SFI [Wahbe et al]
  - events are read, write, jump
  - enforce memory safety properties
- SASI [Erlingsson & Schneider]

  Naccio [Evans & Twyman]
  - flexible policy languages
  - not certifying compilers
- Recent workshops on run-time monitoring ...

Security Automata

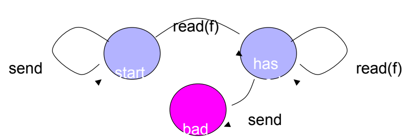
- General mechanism for specifying policy
- Specify any safety property
  - access control policies
    - "cannot access file /etc/passwd"
  - resource bound policies
    - "allocate no more than 1M of memory"
  - the Melissa policy
    - "no network send after file read"

Example



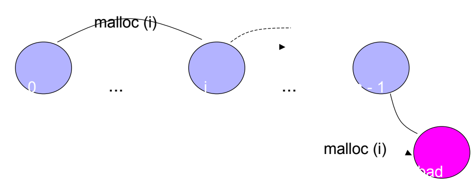
- Policy: No send operation after a read

Monitor program execution



% untrusted program	% s.a.: start state
send();	% ok $\Rightarrow$ start
read(f);	% ok $\Rightarrow$ has read
send();	% bad security violation

Bounding Resource Use



- Policy: "allocate fewer than n bytes"
  - Requires n states

Enforcing Security Autom Policy

- Wrap function calls in checks:

  send() ➔

  let next_state = $check_{send}$(current_state) in send()

- Improve performance using program analysis

Limits to Run-Time Monitoring

- What's a program?
  - A set of possible executions
- What's an execution?
  - A sequence of states
- What's a security policy?
  - A predicate on a set of executions

Safety Policies

- Monitors can only enforce safety policies
- Safety policy is a predicate on a prefix of states [Schneider98]
  - Cannot depend on future
    - Once predicate is false, remains false
  - Cannot depend on other possible executions

Security vs Safety

- Monitoring can only check *safety* properties
- Security properties
  - Can be safety properties
    - One user cannot access another's data
    - Write file only if file owner
  - But some are not
    - Availability
    - Information flow

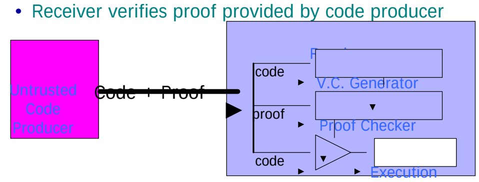
Larger Goal

- Define policies
  - high-level, flexible and system-independent specification language
- Instrument system
  - dynamic security checks and static information
- If this is done on source code ...
  - Preserve proof of security policy during compilation and optimization
  - Verify certified compiler output to reduce TCB

Trusted Computing Base: the part of the system you rely on for security

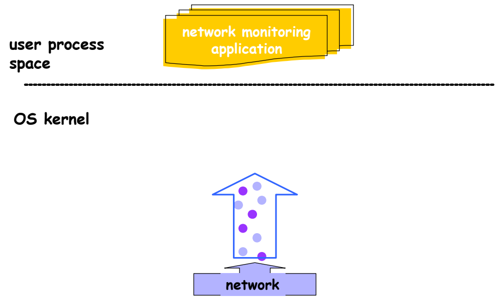
Proof-Carrying Code

- Basic idea
  - Receiver verifies proof provided by code producer

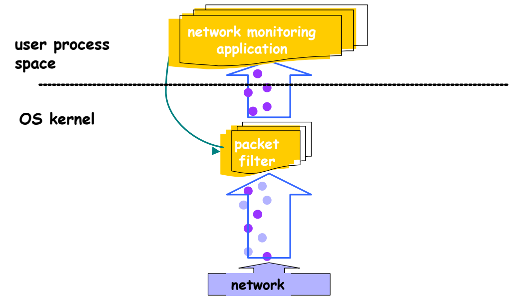


- Important:
  - finding a proof is hard
  - verifying a proof is easy
  - "not so apparent to systems people"

Example: Packet Filters



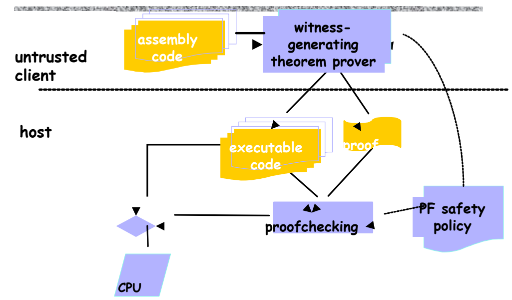
Example: Packet Filters



An Experiment:

- Safety Policy:
  - Given a packet, returns yes/no
  - Packets are read only, small scratchpad
  - No loops in filter code
- Experiment: [Necula & Lee, OSDI'96]
  - Berkeley Packet Filter Interpreter
  - Modula-3 (SPIN)
  - Software Fault Isolation
  - PCC

Packet Filters in PCC

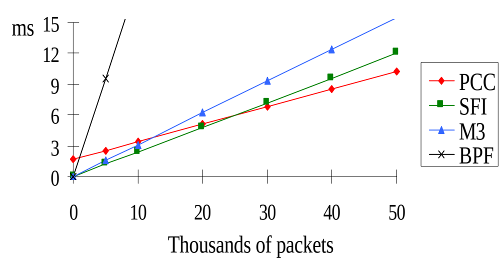


Packet Filter Summary

The PCC packet filter worked extremely well:

- BPF safety policy was easy to verify automatically.
  - r0 is aligned address of network packet (read only)
  - r1 is length of packet (>=64 bytes)
  - r2 is aligned address of writeable 16-byte array
- Allowed hand-optimized packet filters.
  - The "world's fastest packet filters".
  - 10 times faster than BPF.
- Proof sizes and checking times were small.
  - About 1ms proof checking time.
  - 100%-300% overhead for attached proof.

Results: PCC wins

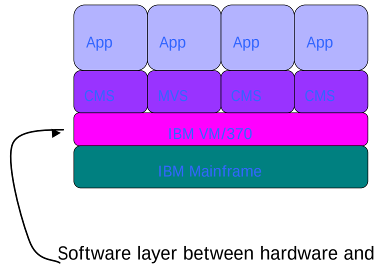


Security using Virtual Machines

- Background
  - IBM virtual machine monitors
  - VMware virtual machine
- Security potential
  - Isolation
  - Flexible Networking
  - I/O interposition
  - Observation from the Host
- Examples
  - Intrusion Detection
  - NSA NetTop

Slide credit: Ed Bugnion, *VMware Inc.*

Virtual Machine Monitors

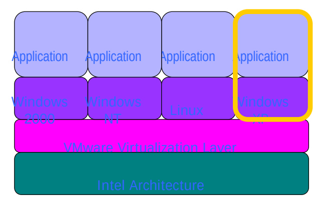


Software layer between hardware and OS, virtualizes and manages hardware resources

History of Virtual Machines

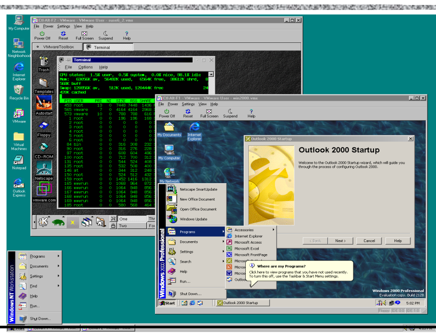
- IBM VM/370 – A VMM for IBM mainframe
  - Multiple OS environments on expensive hardware
  - Desirable when few machine around
- Popular research idea in 1960s and 1970s
  - Entire conferences on virtual machine monitor
  - Hardware/VMM/OS designed together
- Interest died out in the 1980s and 1990s
  - Hardware got cheap
  - OS became more more powerful (e.g multi-user)

VMware Virtual Machines

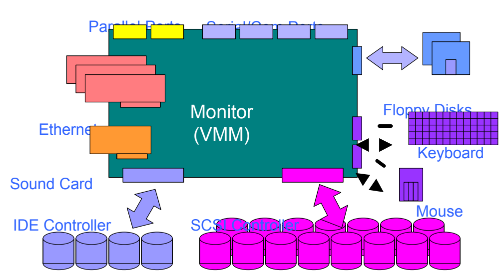


VMware *virtual machine* is an application execution environment with its own operating system

VMware Workstation: Screen shot



Virtual Hardware



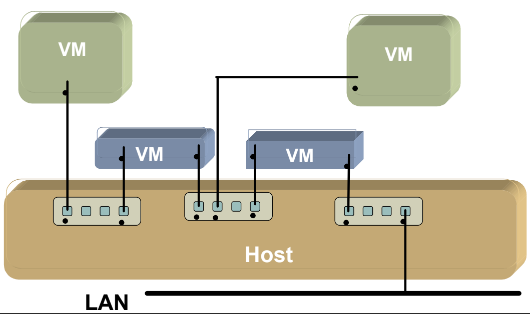
Security from virtual machine

- Strong isolation
- Flexible networking
- I/O interposition
- Observation from the host

Isolation at multiple levels

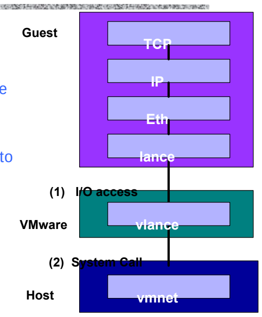
- Data security
  - Each VM is managed independently
    - Different OS, disks (files, registry), MAC address (IP address)
    - Data sharing is not possible
- Faults
  - Crashes are contained within a VM
- Performance (ESX only)
  - Can guarantee performance levels for individual VMs
- Security claim
  - No assumptions required for software inside a VM

Flexible Networking: VMnets



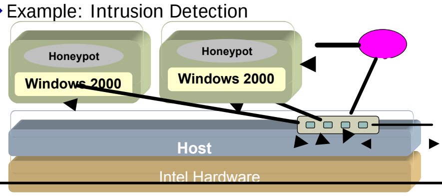
Mandatory I/O Interposition

- Two levels
  - (1) No direct Guest I/O
    - All guest I/O operations are mediated by VMware
  - (2) VMware uses host I/O
    - VMware uses system calls to execute all I/O requests
- Examples
  - Networking (shown →)
  - Disk I/O



Observation by Host system

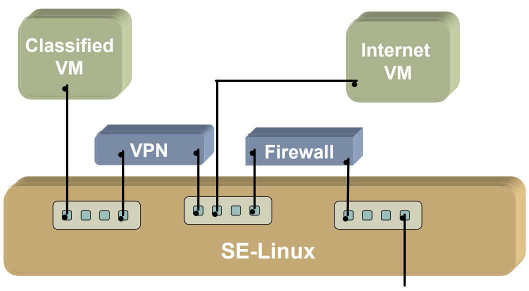
- "See without being seen" advantage
  - Very difficult within a computer, possible on host
- Observation points:
  - Networking (through vmnet)
  - Physical memory
  - Disk I/O (read and write)
  - Any other I/O
- Example: Intrusion Detection



Vmware Application: Classified Networks

- Information Assurance requirement
  - Data cannot flow between diff classification networks
- Conventional solution
  - Military "airgap"
  - Dedicate distinct computer for access to each network

National Security Agency NetTop



NetTop = SE-Linux + VMware

- SE-Linux:
  - Security-Enhanced Linux
  - Mandatory Access Control with flexible security policy
- VMware Workstation:
  - VMs configuration limited by security policy
- NetTop:
  - Locked-down SE-Linux policy
  - No networking on the host itself

Effectiveness of Virtual Machines

- VM restricts memory, disk, network access
  - Apps cannot interfere, cannot change host file sys
  - Also prevents linking software to specific hardware (e.g., MS registration feature …)
- Can software tell if running on top of VM?
  - Timing? Measure time required for disk access
    - VM may try to run clock slower to prevent this attack
    - but slow clock may break an application like music player
- Is VM a reliable solution to airgap problem?
  - If there are bugs in VM, this could cause problems
  - Covert channels (discuss later)

Summary

- Run unreliable code in protected environment
- Sources of protection
  - Modify application to check itself
  - Monitor calls to operating system
  - Put Application and OS in a VM
- General issues
  - Can achieve course-grained protection
    - Prevent file read/write, network access
  - Difficult to express, enforce fine-grained policy
    - Do not let any account numbers read from file Employee_Accts be written into file Public_BBoard