

Lecture 4: Dealing with Unreliable Code

ECE1776
David Lie

1

Topic

- Systems today are made of many diverse components. To interact in a useful way, we often have to take applications from outside sources and execute them locally:
 - To trade stocks, you must download a JAVA applet
 - To share files you must run a P2P client
 - To view a file you must download and run the viewer
- Even programs coming from reputable sources may have bugs or unintended side effects:
 - Systems are too diverse, developer cannot possibly know about all possible 2nd order effects
- End result, a great deal of code we run today is deemed "unreliable"



2

The Goal

- Ultimately, what we would like is given some system, we would like to be able to execute unreliable code without compromising the overall security of the entire system:
 - Vulnerabilities in the unreliable code may damage the code itself, but should not break the host system
- Such an environment is generally called a **sandbox**



3

This Lecture

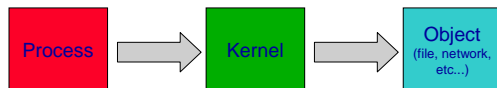
- Conventional OS sandboxing:
 - Process space isolation
 - chroot and jail
- Four approaches for sandboxing compiled code
 - Code modification for run-time monitoring
 - System call interposition
 - Proof-carrying code
 - Virtual machines (e.g., VMWare, User Mode Linux)



4

Conventional OS Protection

- Keep processes separate
 - Each process runs in separate address space
 - Critical resources accessed through systems calls
 - File system, network, etc.
- Kernel functions as a reference monitor that checks all access to objects for proper credentials



5

Conventional OS Protection

- However, for some applications, OS protection doesn't provide enough isolation
 - Programs can still see what other processes are running on machines
 - Can see the entire file system
 - All programs running on the system have the same level of access to the system call interface
- OS's don't provide any fine-grain isolation, either programs are running on the system and have access to all facilities that other programs have, or they aren't on the system at all
- OS's can't protect themselves very well. Once you get root, you can install drivers, change configuration settings, etc... You only have to compromise 1 root process to do this.



6

Unix chroot

- Can restrict how much of the file system a process can see with chroot command
 - chroot changes root directory
 - 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 is still root), process could escape



7

Only root should execute chroot

- Otherwise, jailed program can escape
 - mkdir(/tmp) /* create temp directory */
 - chroot(/tmp) /* 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)



8

FreeBSD jail command

- Stronger version of chroot
 - Jail provides a limited view of the file system, just like chroot
 - Provides a root account that only has privileges inside the jail
 - Restricts certain system calls
 - 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 (IPC, signals, etc...)
- This is coming closer to a virtual machine except:
 - No process or resource isolation
 - UID's are not isolated for example, need to be careful about UID collisions



9

Problems with chroot, jail approach

- Somewhat difficult to setup environment correctly
 - Confine program to directory
 - but this directory may not contain utilities that program needs to call
 - Copy utilities into restricted environment
 - However, copying too many files may give the attacker to escape the jail
- Does not monitor network access (even in jail, you can access any other IP, you can sniff packets)
- No fine grained access control
 - Everything inside jail can be accessed, everything outside can't. All files have to be duplicated.



10

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



11

Rest of lecture

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



12

Software Fault Isolation (SFI)

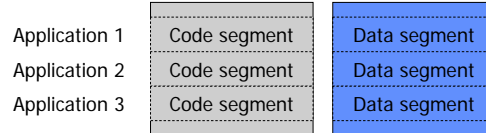
- Wahbe, Lucco, Anderson, Graham [SOSP'93]
 - Collusa Software (founded '94, bought by Microsoft '96)
 - Support multiple applications in same address space
- This is an old idea that was originally used to enforce isolation before hardware support for privilege protection domains and virtual memory
- Useful to achieve OS-level protection (or better) without overhead of OS context switch
- Current interest is in supporting applications that sometimes need to call other applications e.g.:
 - Plug-ins for web browsers



13

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



14

System call based approach

- Rather than checking every memory access, check system calls instead
 - Idea is that process cannot interact with outside world except through system call interface with the OS
- Several projects
 - Janus (Berkeley)
 - Systrace (Michigan)
- Trap system calls
 - Check parameters, deny unauthorized calls
 - Enforce mandatory access control in OS that does not provide mandatory access control



15

Janus

- One of the first papers to propose system call interposition to limit the interface a process has to the operating system:
 - Idea is to invoke Janus every time process makes a system call
 - Janus is provided with the arguments of the system call
 - Janus either decides to allow the system call, or deny the system call
- Two implementation approaches:
 - Use ptrace facility: this is the same facility used by gdb to track another process' execution
 - Use /proc file system
 - Solaris implementation allows tracing process to get arguments and return value



16

Other Projects

- David Wagner who worked on Janus, later extended the idea
 - Policies were derived from creating models from programs via static analysis of the source code:
 - A push down automata model of the program is created, including which system calls can be made when
 - If program deviates from the model, the program has had a fault
 - Assumes that the programs are not malicious initially, just may have vulnerabilities that can be exploited
- Vulnerable to race conditions:
 - Process can make a system call, when monitor runs everything looks ok.
 - During system call, process can alter some of the arguments and do something else



17

Other Projects

- Niels Provos at Michigan (now google) created systrace:
 - Via user configuration, a policy for what system calls and what arguments an application does is derived.
 - One main difference from Janus is that systrace has support at the kernel level.
 - Systrace solves some of the problems that affected Janus:
 - Kernel interface solves race conditions
 - Normalizes the file system to resolve symlinks
 - Kernel implementation is also fail-safe, i.e. if systrace fails, application is denied access by default
 - Policies are quite long and complex
 - More info at: <http://www.citi.umich.edu/u/provos/systrace/>



18

Proof-Carrying Code

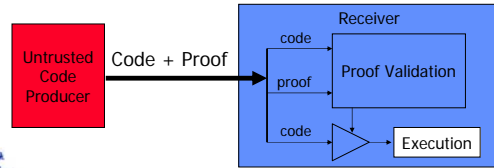
- Rather than try to isolate or sandbox code, verify:
 - That the program meets a certain safety property
- However, verification is often a very expensive and time consuming process:
 - We would not want to verify the code every time it is run
- Solution: Proof-Carrying Code
 - Proof that the code meets some property is created and integrated with the code
 - Receiver verifies the proof before execution
 - This is efficient if the time to check the proof is faster than the time to create the proof



19

Proof-Carrying Code

- A proof accompanies the code which:
 - The checker checks each statement or lemma in the structure of the proof against the code
 - If the proof validates correctly, the code is allowed to execute



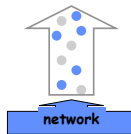
20

Example: Packet Filters

user process space



OS kernel



21

Example: Packet Filters

user process space



OS kernel



Application downloads untrusted code into the kernel: The code must be safe to run in the kernel

Use PCC!



22

Berkeley Packet Filters using

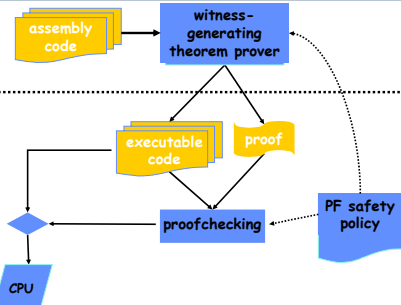
- Safety Policy:
 - Given a packet, returns yes/no
 - Packets are read only, small scratchpad
 - No loops in filter code
- Experiment Comparing: [Necula & Lee, OSDI'96]
 - Berkeley Packet Filter Interpreter (BPF)
 - Modula-3 (SPIN) - typsafe language
 - Software Fault Isolation
 - PCC



23

Packet Filters in PCC

untrusted client



24

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.



25

Security using Virtual Machines

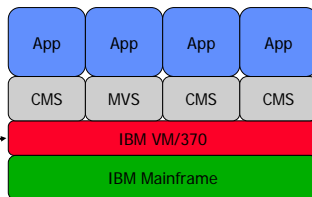
- Background
 - IBM virtual machine monitors
 - VMware virtual machine
- Security potential
 - Isolation
 - Flexible Networking
 - I/O interposition
 - Observation from the Host
- Examples
 - Intrusion Detection
 - NSA NetTop
 - Honeypots

The Ultimate Sandbox?



26

Virtual Machine Monitors



Software layer between hardware and OS, virtualizes and manages hardware resources
Low overhead ~ < 2x slowdown



27

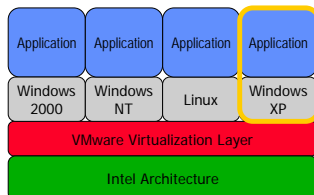
History of Virtual Machines

- 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)
- Today:
 - Processors very fast
 - Administration, reliability, security is costly
 - X86 VM Systems: UML, Xen and VMware



28

VMWare Virtual Machines



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



29

UML Virtual Machines

- Main difference is that OS is ported to target operating system
 - Written by Jeff Dike
 - UML is a port of Linux to the Linux OS
 - As a result, hardware simulation is simplified greatly
 - Very comprehensive capabilities
 - Networking
 - Virtual serial devices
 - Can have X windows connection through nested X server'
- Xen:
 - Part of the Xenoservers project out of Cambridge (Now XenSource, a company)
 - Focused on high performance and scalability



30

Isolation at multiple levels

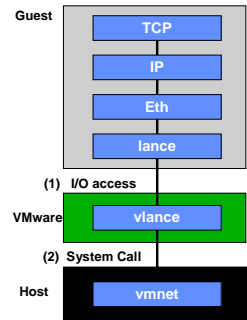
- Data security
 - Each VM is managed independently
 - Different OS, disks (files, registry), MAC address (IP address)
 - Data sharing is not possible between VM's except through standard channels (i.e. network, shared network file systems)
- Faults
 - Crashes are contained within a VM
- Security claim
 - No assumptions required for software inside a VM
- Can run multiple OS's on a single machine and communicate between them via dedicated network interface (Groupware)



31

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



VMware Application: Classified Networks

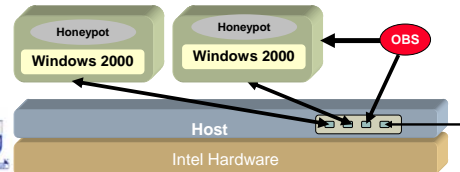
- Information Assurance requirement
 - Data cannot flow between diff classification networks
- Conventional solution
 - Military "airgap"
 - Dedicate distinct computer for access to each network
- VM Solution:
 - Have 1 computer running different VM's, each with their own security classification
 - Virtual network monitors (or in some cases, prevents) communication between VM's depending on their classification level



33

Another Application: Honeypots

- Honeypots are machines deliberately left for attackers to compromise
 - Used to gather information about attackers
 - Lure attackers into revealing their presence to protect real machines
- Virtual Machines:
 - Use the host system to observe the honeypot



34

Observation by Host System

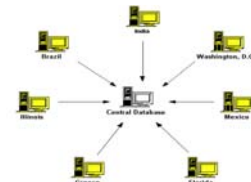
- "See without being seen" advantage
 - Very difficult within a computer, possible on host
 - Attacker can cook logs, disable monitors on compromised machine
 - Hard to disable if not visible because they're on the host
- Observation points:
 - Networking (through vmnet) Physical memory
 - Disk I/O (read and write) Any other I/O



35

Honeynet Project

- Network of honeypots world-wide
 - Collect qualitative and quantitative (statistical) information on attacks
 - System that records
 - Network packets
 - TTY Keyboard logs



36

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
 - Effective finger-printers exist
- Is VM a reliable solution to airgap problem?
 - If there are bugs in VM, this could cause problems
 - Covert channels (discuss later)



37

Summary

- Run unreliable code in protected environment
- Sources of protection
 - Modify application to check itself
 - Monitor calls to operating system
 - Use Proofs to guarantee properties
 - Put Application and OS in a VM (ultimate sandbox)
- General issues
 - Try to get a combination of:
 1. Fine grain access control
 2. High degree of security and isolation
 3. Good performance
 4. Easy of use
 - Generally hard to determine good policy for isolation in a lot of cases



38

Extra Slides

39

A Faster Approach

- Skip test; Just overwrite segment bits
 - $\text{dedicated-reg} \leftarrow \text{target-reg} \& \text{mask-reg}$
 - $\text{dedicated-reg} \leftarrow \text{dedicated-reg} | \text{segment-reg}$
 - 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



40

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



41

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 Aiken⁴²

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



43

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 ...



44

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



45

VMware Workstation: Screen shot



46

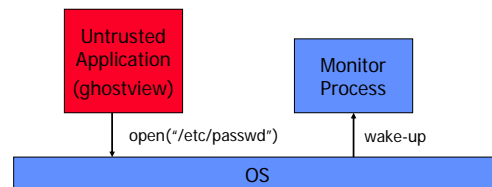
Security from virtual machine

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



47

Ptrace



- Problems
 - Coarse: trace all calls or none
 - Limited error handling
 - Cannot abort system call without killing service
- Note: Janus used ptrace initially, later discarded ...



48

/proc virtual file system under Solaris

- Can trap selected system calls
 - obtain arguments to system calls
 - can cause system call to fail with `errno = 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



49

Hard part

- Configure policy to do 1 of 3 things depending on system call:
 - Always allow
 - Always deny
 - Test
- Test is the trickiest action
 - Allow or deny depending on arguments and previous
- Example policy for file args
 - path allow read, write /tmp/*
 - path deny /etc/passwd
 - network deny all, allow XDisplay



50

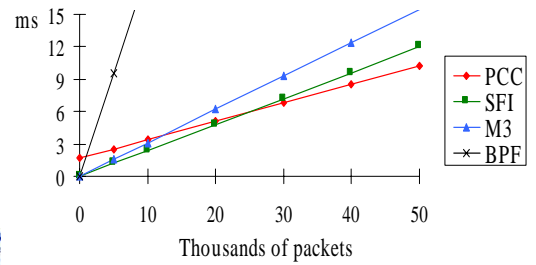
Hard Part

- Janus also had other problems:
 - In a later paper by Garfunkel: [Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools](#).
 - Because Janus is at system call interface it is prone to:
 - Certain race conditions, problems with symlinks
 - Has to do a lot of work to guess at what the current state of the kernel is
 - Overlooking indirect paths to resources (not all paths necessarily through system calls)
 - Side effects of denying system calls not dealt with
 - Janus also has portability issues
 - `/proc` and `ptrace` facilities are highly system specific



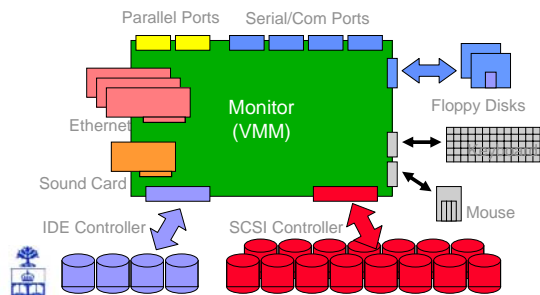
51

Results: PCC wins



52

Virtual Hardware



53