Lecture 15
October 24, 2022
Hardware Isolation

• Ensure the hardware is disconnected from any other system
  • This includes networking, including wireless

• Example: SCADA systems
  • 1st generation: serial protocols, not connected to other systems or networks; no security defenses needed, focus being on malfunctions
  • 2nd generation: serial networks connected to computers not connected to Internet
  • 3rd generation: TCP/IP protocol running on networks connected to Internet; need security defenses for attackers coming in over Internet

• Example: electronic voting systems
  • Physical isolation protects systems from attackers changing votes remotely
  • Required in many U.S. states, such as California: never connect them to any network
Virtual Machine

• Program that simulates hardware of a machine
  • Machine may be an existing, physical one or an abstract one
  • Uses special operating system, called *virtual machine monitor (VMM)* or *
hypervisor*, to provide environment simulating target machine

• Types of virtual machines
  • Type 1 hypervisor: runs directly on hardware
  • Type 2 hypervisor: runs on another operating system

• Existing OSes do not need to be modified
  • Run under VMM, which enforces security policy
  • Effectively, VMM is a security kernel
VH$_i$ is virtual machine $i$
T2H$_i$ is type-2 hypervisor $i$
VMM as Security Kernel

- VMM deals with subjects (the VMs)
  - Knows nothing about the processes within the VM
- VMM applies security checks to subjects
  - By transitivity, these controls apply to processes on VMs
- Thus, satisfies rule of transitive confinement
Example: Xen Hypervisor

• Xen 3.0 hypervisor on Intel virtualization technology
• Two modes, VMX root and non-root operation
• Hardware-based VMs (HVMs) are fully virtualized domains, support unmodified guest operating systems and run in non-root operation mode
  • Xen hypervisor runs in VMX root mode
• 8 levels of privilege
  • 4 in VMX root operation mode
  • 4 in VMX root operation mode
  • No need to virtualize one of the privilege levels!
Xen and Privileged Instructions

• Guest operating system executes privileged instruction
  • But this can only be done as a VMX root operation
• Control transfers to Xen hypervisor (called VM exit)
• Hypervisor determines whether to execute instruction
• After, it updates HVM appropriately and returns control to guest operating system (called VM entry)
Problem

• Physical resources shared
  • System CPU, disks, etc.

• May share logical resources
  • Depends on how system is implemented

• Allows covert channels
Sandboxes

• An environment in which actions are restricted in accordance with security policy
  • Limit execution environment as needed
    • Program not modified
    • Libraries, kernel modified to restrict actions
  • Modify program to check, restrict actions
    • Like dynamic debuggers, profilers
Example: Capsicum

• Framework developed to sandbox an application
• Capability provides fine-grained rights for accessing, manipulating underlying file
• To enter sandbox (capability mode), process issues cap_enter
• Given file descriptor, create capability with cap_new
  • Mask of rights indicates what rights are to be set; if capability exists, mask must be subset of rights in that capability
• At user level, library provides interface to start sandboxed process and delegate rights to it
  • All nondelegated file descriptors closed
  • Address space flushed
  • Socket returned to creator to enable it to communicate with new process
Example: Capsicum (con’t)

• Global namespaces not available
  • So system calls that depend on that (like `open(2)`) don’t work
    • Need to use a modified `open` that takes file descriptor for containing directory
  • Other system calls modified appropriately
    • System calls creating memory objects can create anonymous ones, not named ones (as those names are in global namespace)

• Subprocesses cannot escalate privileges
  • But a privileged process can enter capability mode

• All restrictions applied in kernel, not at system call interface
Program Confinement and TCB

- Confinement mechanisms part of trusted computing bases
  - On failure, less protection than security officers, users believe
  - “False sense of security”
- Must ensure confinement mechanism correctly implements desired security policy
Sandboxes

- An environment in which actions are restricted in accordance with security policy
  - Limit execution environment as needed
    - Program not modified
    - Libraries, kernel modified to restrict actions
  - Modify program to check, restrict actions
    - Like dynamic debuggers, profilers
Example: Capsicum

- Framework developed to sandbox an application
- *Capability* provides fine-grained rights for accessing, manipulating underlying file
- To enter sandbox (*capability mode*), process issues *cap_enter*
- Given file descriptor, create capability with *cap_new*
  - Mask of rights indicates what rights are to be set; if capability exists, mask must be subset of rights in that capability
- At user level, library provides interface to start sandboxed process and delegate rights to it
  - All nondelegated file descriptors closed
  - Address space flushed
  - Socket returned to creator to enable it to communicate with new process
Example: Capsicum (con’t)

• Global namespaces not available
  • So system calls that depend on that (like `open(2)`) don’t work
    • Need to use a modified `open` that takes file descriptor for containing directory
  • Other system calls modified appropriately
    • System calls creating memory objects can create anonymous ones, not named ones (as those names are in global namespace)

• Subprocesses cannot escalate privileges
  • But a privileged process can enter capability mode

• All restrictions applied in kernel, not at system call interface
Program Confinement and TCB

• Confinement mechanisms part of trusted computing bases
  • On failure, less protection than security officers, users believe
  • “False sense of security”

• Must ensure confinement mechanism correctly implements desired security policy
Covert Channels

• Shared resources as communication paths
  • *Covert storage channel* uses attribute of shared resource
    • Disk space, message size, etc.
  • *Covert timing channel* uses temporal or ordering relationship among accesses to shared resource
    • Regulating CPU usage, order of reads on disk
Example Storage Channel

• Processes $p$, $q$ not allowed to communicate
  • But they share a file system!

• Communications protocol:
  • $p$ sends a bit by creating a file called 0 or 1, then a second file called $send$
    • $p$ waits until $send$ is deleted before repeating to send another bit
  • $q$ waits until file $send$ exists, then looks for file 0 or 1; whichever exists is the bit
    • $q$ then deletes 0, 1, and $send$ and waits until $send$ is recreated before repeating to read another bit
Example Timing Channel

• System has two VMs
  • Sending machine $S$, receiving machine $R$

• To send:
  • For 0, $S$ immediately relinquishes CPU
    • For example, run a process that instantly blocks
  • For 1, $S$ uses full quantum
    • For example, run a CPU-intensive process

• $R$ measures how quickly it gets CPU
  • Uses real-time clock to measure intervals between access to shared resource (CPU)
Example Covert Channel

• Uses ordering of events; does not use clock
• Two VMs sharing disk cylinders 100 to 200
  • SCAN algorithm schedules disk accesses
  • One VM is High (H), other is Low (L)
• Idea: L will issue requests for blocks on cylinders 139 and 161 to be read
  • If read as 139, then 161, it’s a 1 bit
  • If read as 161, then 139, it’s a 0 bit
How It Works

• *L* issues read for data on cylinder 150
  • Relinquishes CPU when done; arm now at 150

• *H* runs, issues read for data on cylinder 140
  • Relinquishes CPU when done; arm now at 140

• *L* runs, issues read for data on cylinders 139 and 161
  • Due to SCAN, reads 139 first, then 161
  • This corresponds to a 1

• To send a 0, *H* would have issued read for data on cylinder 160
Analysis

• Timing or storage?
  • Usual definition ⇒ storage (no timer, clock)
• Modify example to include timer
  • $L$ uses this to determine how long requests take to complete
  • Time to seek to 139 < time to seek to 161 ⇒ 1; otherwise, 0
• Channel works same way
  • Suggests it’s a timing channel; hence our definition
Noisy vs. Noiseless

• Noiseless: covert channel uses resource available only to sender, receiver
• Noisy: covert channel uses resource available to others as well as to sender, receiver
  • Idea is that others can contribute extraneous information that receiver must filter out to “read” sender’s communication
Defending Against Covert Channels

- Add lots of noise
  - The idea is to prevent the receiver from being able to pick up the signal the sender is sending
- Make the events regular
  - Similar to adding noise, this hides the signal in the regularity
Vulnerability Classification

• Describe flaws from differing perspectives
  • Exploit-oriented
  • Hardware, software, interface-oriented

• Goals vary; common ones are:
  • Specify, design, implement computer system without vulnerabilities
  • Analyze computer system to detect vulnerabilities
  • Address any vulnerabilities introduced during system operation
  • Detect attempted exploitations of vulnerabilities
Example Flaws

• Use these to compare classification schemes
• First one: race condition (*xterm*)
• Second one: buffer overflow on stack leading to execution of injected code (*fingerd*)
• Both are very well known, and fixes available!
  • And should be installed everywhere ...
Flaw #1: xterm

• **xterm** emulates terminal under X11 window system
  • Must run as *root* user on UNIX systems
    • No longer universally true; reason irrelevant here

• Log feature: user can log all input, output to file
  • User names file
  • If file does not exist, **xterm** creates it, makes owner the user
  • If file exists, **xterm** checks user can write to it, and if so opens file to append log to it
File Exists

• Check that user can write to file requires special system call
  • Because root can append to any file, check in open will always succeed

  Check that user can write to file “/usr/tom/X”

  if (access("/usr/tom/X", W_OK) == 0){
    Open “/usr/tom/X” to append log entries
    if ((fd = open("/usr/tom/X", O_WRONLY|O_APPEND))< 0){
        /* handle error: cannot open file */
    }
  }
Problem

- Binding of file name “/usr/tom/X” to file object can change between first and second lines
  - left is at *access*; right is at *open*
  - Note file opened is *not* file checked
Flaw #2: fingerd

- Exploited by Internet Worm of 1988
  - Recurs in many places, even now
- *finger* client sends request for information to server *fingerd* (*finger* daemon)
  - Request is name of at most 512 chars
  - What happens if you send more?
Buffer Overflow

• Extra chars overwrite rest of stack, as shown
• Can make those chars change return address to point to beginning of buffer
• If buffer contains small program to spawn shell, attacker gets shell on target system

gets local variables
other return state info
return address of main
parameter to gets
input buffer
main local variables

gets local variables
other return state info
address of input buffer
program to invoke shell
main local variables

October 24, 2022
ECS 235A, Computer and Information Security
Frameworks

- Goals dictate structure of classification scheme
  - Guide development of attack tool $\Rightarrow$ focus is on steps needed to exploit vulnerability
  - Aid software development process $\Rightarrow$ focus is on design and programming errors causing vulnerabilities

- Following schemes classify vulnerability as n-tuple, each element of n-tuple being classes into which vulnerability falls
  - Some have 1 axis; others have multiple axes
Research Into Secure Operating Systems (RISOS)

• Goal: aid computer, system managers in understanding security issues in OSes, and help determine how much effort required to enhance system security

• Attempted to develop methodologies and software for detecting some problems, and techniques for avoiding and ameliorating other problems

• Examined Multics, TENEX, TOPS-10, GECOS, OS/MVT, SDS-940, EXEC-8
Classification Scheme

- Incomplete parameter validation
- Inconsistent parameter validation
- Implicit sharing of privileged/confidential data
- Asynchronous validation/inadequate serialization
- Inadequate identification/authentication/authorization
- Violable prohibition/limit
- Exploitable logic error
Incomplete Parameter Validation

- Parameter not checked before use
- Example: emulating integer division in kernel (RISC chip involved)
  - Caller provided addresses for quotient, remainder
  - Quotient address checked to be sure it was in user’s protection domain
  - Remainder address *not* checked
    - Set remainder address to address of process’ level of privilege
    - Compute 25/5 and you have level 0 (kernel) privileges
- Check for type, format, range of values, access rights, presence (or absence)
Inconsistent Parameter Validation

• Each routine checks parameter is in proper format for that routine but the routines require different formats

• Example: each database record 1 line, colons separating fields
  • One program accepts colons, newlines as part of data within fields
  • Another program reads them as field and record separators
  • This allows bogus records to be entered
Legacy of RISOS

• First funded project examining vulnerabilities

• Valuable insight into nature of flaws
  • Security is a function of site requirements and threats
  • Small number of fundamental flaws recurring in many contexts
  • OS security not critical factor in design of OSes

• Spurred additional research efforts into detection, repair of vulnerabilities