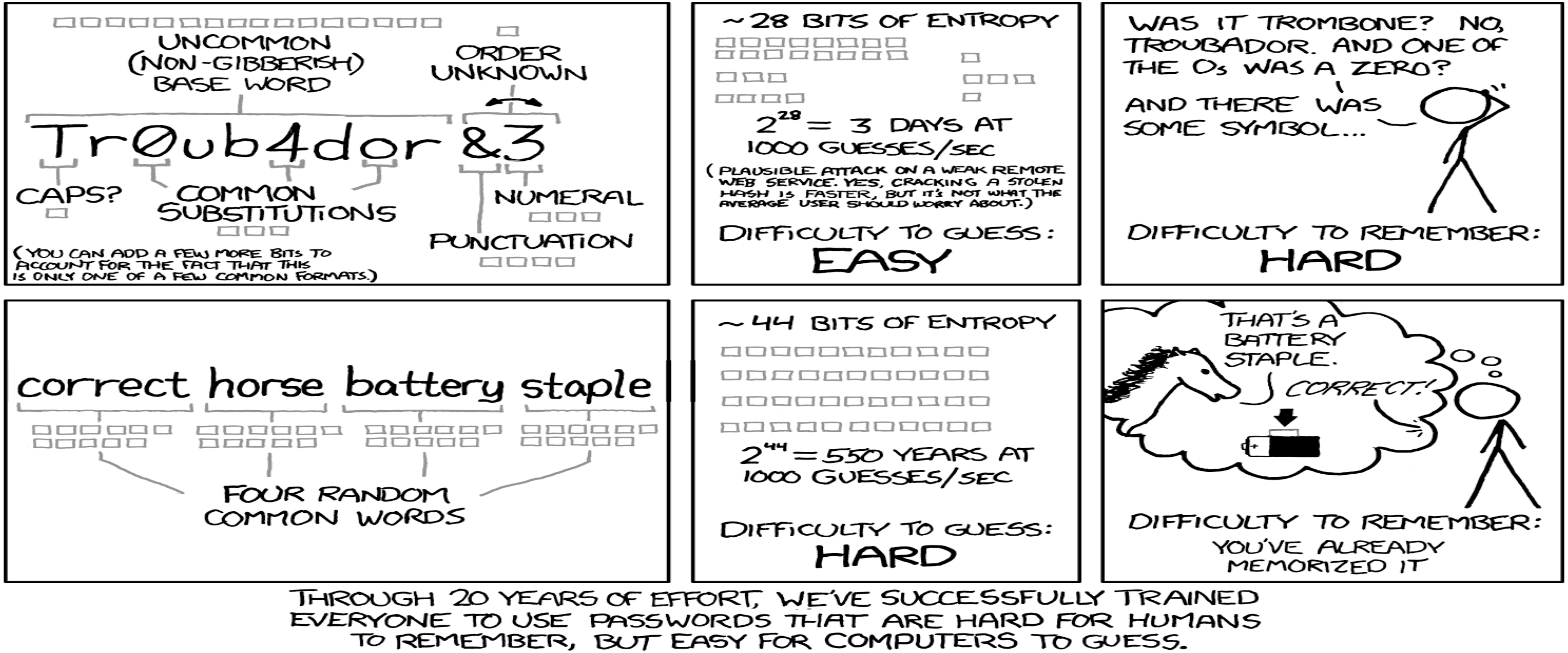
Lecture 15
October 29, 2025

ECS 235A, Computer and Information Security

Administrative Stuff

- Homework 3, Extra Credit 3 due on November 5, 2025
- Project outlines graded

People and Passwords



From *xkcd*, https://xkcd.com/936/

Guessing Through *L*

- Cannot prevent these
  - Otherwise, legitimate users cannot log in
- Make them slow
  - Backoff
  - Disconnection
  - Disabling
    - Be very careful with administrative accounts!
  - Jailing
    - Allow in, but restrict activities

Password Aging

- Force users to change passwords after some time has expired
  - How do you force users not to re-use passwords?
    - Record previous passwords
    - Block changes for a period of time
  - Give users time to think of good passwords
    - Don’t force them to change before they can log in
    - Warn them of expiration days in advance

Challenge-Response

- User, system share a secret function *f* (in practice, *f* is a known function with unknown parameters, such as a cryptographic key)

user $\xrightarrow{\text{request to authenticate}}$ *system*

user $\xleftarrow[\text{(the challenge)}]{\text{random message } r}$ *system*

user $\xrightarrow[\text{(the response)}]{f(r)}$ *system*

Pass Algorithms

- Challenge-response with the function *f* itself a secret
  - Example:
    - Challenge is a random string of characters such as “abcdefg”, “ageksido”
    - Response is some function of that string such as “bdf”, “gkip”
  - Can alter algorithm based on ancillary information
    - Network connection is as above, dial-up might require “aceg”, “aesd”
  - Usually used in conjunction with fixed, reusable password

One-Time Passwords

- Password that can be used exactly *once*
  - After use, it is immediately invalidated
- Challenge-response mechanism
  - Challenge is number of authentications; response is password for that particular number
- Problems
  - Synchronization of user, system
  - Generation of good random passwords
  - Password distribution problem

S/Key

- One-time password scheme based on idea of Lamport
- *h* one-way hash function (SHA-256, for example)
- User chooses initial seed *k*
- System calculates:

$$h(k) = k_1,\ h(k_1) = k_2,\ \ldots,\ h(k_{n-1}) = k_n$$

- Passwords are reverse order:

$$p_1 = k_n,\ p_2 = k_{n-1},\ \ldots,\ p_{n-1} = k_2,\ p_n = k_1$$

S/Key Protocol

System stores maximum number of authentications n, number of next authentication i, last correctly supplied password p_{i-1}.

$$\textit{user} \xrightarrow{\{\ \text{name}\ \}} \textit{system}$$

$$\textit{user} \xleftarrow{\{\ i\ \}} \textit{system}$$

$$\textit{user} \xrightarrow{\{\ p_i\ \}} \textit{system}$$

System computes $h(p_i) = h(k_{n-i+1}) = k_{n-i} = p_{i-1}$. If matches what is stored, system replaces p_{i-1} with p_i and increments i.

Hardware Support

- Token-based
  - Used to compute response to challenge
    - May encipher or hash challenge
    - May require PIN from user
- Temporally-based
  - Every minute (or so) different number shown
    - Computer knows what number to expect when
  - User enters number and fixed password

C-R and Dictionary Attacks

- Same as for fixed passwords
  - Attacker knows challenge *r* and response *f*(*r*); if *f* encryption function, they can try different keys
    - May only need to know *form* of response; attacker can tell if guess correct by looking to see if deciphered object is of right form
    - Example: Kerberos Version 4 used DES, but keys had 20 bits of randomness; Purdue attackers guessed keys quickly because deciphered tickets had a fixed set of bits in some locations

Encrypted Key Exchange

- Defeats off-line dictionary attacks
- Idea: random challenges enciphered, so attacker cannot verify correct decipherment of challenge
- Assume Alice, Bob share secret password s
- In what follows, Alice needs to generate a random public key p and a corresponding private key q
- Also, k is a randomly generated session key, and R_A and R_B are random challenges

EKE Protocol

$$\mathit{Alice} \xrightarrow{\text{Alice} \,||\, E_s(p)} \mathit{Bob}$$

$$\mathit{Alice} \xleftarrow{E_s(E_p(k))} \mathit{Bob}$$

Now Alice, Bob share a randomly generated secret session key k

$$\mathit{Alice} \xrightarrow{E_k(R_A)} \mathit{Bob}$$

$$\mathit{Alice} \xleftarrow{E_k(R_A R_B)} \mathit{Bob}$$

$$\mathit{Alice} \xrightarrow{E_k(R_B)} \mathit{Bob}$$

Biometrics

- Automated measurement of biological, behavioral features that identify a person
  - Fingerprints: optical or electrical techniques
    - Maps fingerprint into a graph, then compares with database
    - Measurements imprecise, so approximate matching algorithms used
  - Voices: speaker verification or recognition
    - Verification: uses statistical techniques to test hypothesis that speaker is who is claimed (speaker dependent)
    - Recognition: checks content of answers (speaker independent)

Other Characteristics

- Can use several other characteristics
  - Eyes: patterns in irises unique
    - Measure patterns, determine if differences are random; or correlate images using statistical tests
  - Faces: image, or specific characteristics like distance from nose to chin
    - Lighting, view of face, other noise can hinder this
  - Keystroke dynamics: believed to be unique
    - Keystroke intervals, pressure, duration of stroke, where key is struck
    - Statistical tests used

Cautions

- These can be fooled!
  - Assumes biometric device accurate *in the environment it is being used in!*
  - Transmission of data to validator is tamperproof, correct

Location

- If you know where user is, validate identity by seeing if person is where the user is
  - Requires special-purpose hardware to locate user
    - GPS (global positioning system) device gives location signature of entity
    - Host uses LSS (location signature sensor) to get signature for entity

Multiple Methods

- Example: “where you are” also requires entity to have LSS and GPS, so also “what you have”
- Can assign different methods to different tasks
  - As users perform more and more sensitive tasks, must authenticate in more and more ways (presumably, more stringently)
  - File describes authentication required; can include controls on access (time of day, *etc*.), resources, and requests to change passwords
- Example: Pluggable Authentication Modules (PAM)

PAM

- Idea: when program needs to authenticate, it checks central repository for methods to use
- Library call: *pam_authenticate*
  - Accesses file with name of program in */etc/pam_d*
- Modules do authentication checking
  - *sufficient*: succeed if module succeeds
  - *required*: fail if module fails, but all required modules executed before reporting failure
  - *requisite*: like *required*, but don't check all modules
  - *optional*: invoke only if all previous modules fail

Example PAM File

```
auth sufficient    /usr/lib/pam_ftp.so
auth required /usr/lib/pam_unix_auth.so use_first_pass
auth required /usr/lib/pam_listfile.so onerr=succeed \
    item=user sense=deny file=/etc/ftpusers
```

For ftp:

1. If user "anonymous", return success; if not, set PAM_AUTHTOK to password, PAM_RUSER to name, and fail (go to next `auth` line)
2. Now check that password in PAM_AUTHTOK belongs to that of user in PAM_RUSER; if not, fail (go to next `auth` line)
3. Now see if user in PAM_RUSER named in /etc/ftpusers; if so, fail (not authenticated); if error or not found, succeed (authenticated)

Example Problem

- Server balances bank accounts for clients
- Server security issues:
  - Record correctly who used it
  - Send *only* balancing info to client
- Client security issues:
  - Log use correctly
  - Do not save or retransmit data client sends

Generalization

- Client sends request, data to server
- Server performs some function on data
- Server returns result to client
- Access controls:
    - Server must ensure the resources it accesses on behalf of client include *only* resources client is authorized to access
    - Server must ensure it does not reveal client’s data to any entity not authorized to see the client’s data

Confinement Problem

- Problem of preventing a server from leaking information that the user of the service considers confidential

Total Isolation

- Process cannot communicate with any other process
- Process cannot be observed

Impossible for this process to leak information

- Not practical as process uses observable resources such as CPU, secondary storage, networks, etc.

Example

- 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

Covert Channel

- A path of communication not designed to be used for communication
- In example, file system is a (storage) covert channel

Rule of Transitive Confinement

- If *p* is confined to prevent leaking, and it invokes *q*, then *q* must be similarly confined to prevent leaking
- Rule: if a confined process invokes a second process, the second process must be as confined as the first

Isolation

- Constrain process execution in such a way it can only interact with other entities in a manner preserving isolation
  - Hardware isolation
  - Virtual machines
  - Library operating systems
  - Sandboxes
- Modify program or process so that its actions will preserve isolation
  - Program rewriting
  - Compiling
  - Loading

Hardware Isolation

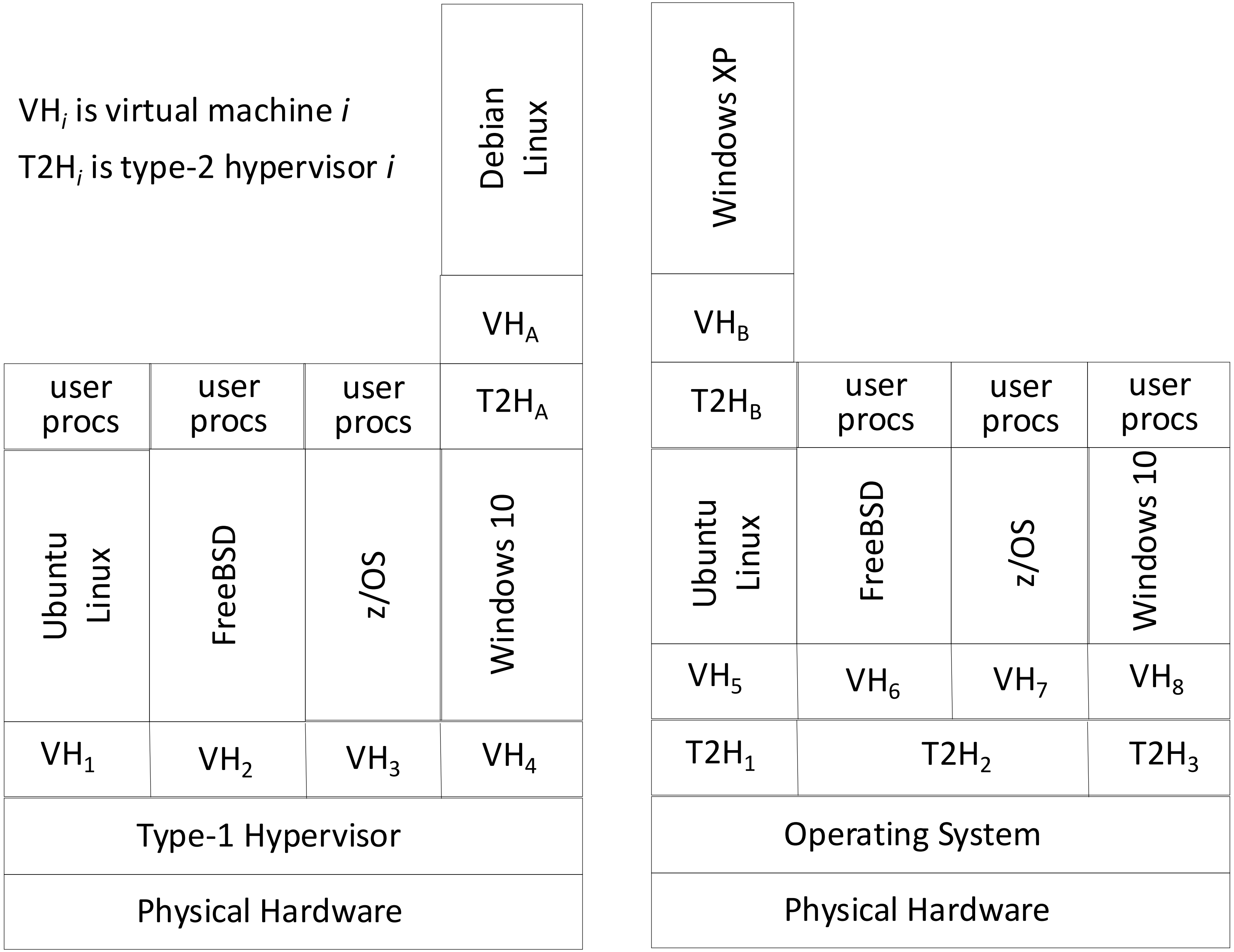
- Ensure the hardware is disconnected from any other system
  - This includes networking, including wireless
- Example: SCADA systems
  - 1st generation: serial protocols, not connected to other systems or networks; no security defenses needed, focus being on malfunctions
  - 2nd generation: serial networks 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

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

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