Security Architecture

- We've been looking at how particular applications are secured
- We need to secure not just a few particular applications, but many applications, running on separate machines
- We need a few more primitives first

Confining an Application

- For Web security, we used OS permissions to protect the system against compromise via a compromised Web server
- Suppose we want to isolate the Web server still further
- More precisely, we want to limit the Web server to a small subset of the system's resources

Rationale

- We wish to run powerful, complex applications that we do not completely trust
- Neither Unix nor Windows file permissions are flexible enough to do what we want
- There are other resources besides files that need to be protected

Couldn’t We Use MAC?

- MAC usually does not have *negative* permissions
- We’d have to find and change the protections of every file on the system that was writable/readable/searchable by **`other`**
- We’d have to ensure that no other such files were created
- This is all possible but difficult
- More seriously, it is not high *assurance*

Other Resources

- What other resources need to be protected?
- CPU time
- Memory, real and virtual
- Disk space
- Network identity
- Network access rights

Some Are Easy

- Operating systems already regulate access to some resources
- Unix examples: **`setrlimit()`**, file system quotas

Network Identity and Access Rights

- A machine has an IP address
- A compromised application can use this address to exploit address-based access control
- If nothing else, it can confuse intrusion detection systems

Bypassing File Permissions

- Suppose the attacker gains root privileges
- This permits overrides file permissions
- Also allows evasion of other resource limits, plus changes to network identity
  - ☞ Change the IP address and hide from the system administrator!

Goals

- Security
- High assurance
- Simple setup
- General-purpose mechanism
- Available to all applications
- We can’t get them all. . .

Change Root: `chroot()`

- Oldest Unix isolation mechanism
- Make a process believe that some subtree is the entire file system
- File outside of this subtree simply don't exist
- Sounds good, but. . .

Limitations of Chroot

- Only root can invoke it. (Why?)
- Setting up minimum necessary environment can be painful
- The program to execute generally needs to live within the subtree, where it's exposed
- Still vulnerable to root compromise
- Doesn't protect network identity

Root versus Chroot

- Suppose an ordinary user could use **`chroot()`**
- Create a link to the **`su`** command
- Create **`/etc`** and **`/etc/passwd`** with a known root password
- Create links to any files you want to read or write
- Besides, root can escape from **`chroot()`**

Escaping Chroot

- What is the current directory? If it's not under the **`chroot()`** tree, try **`chdir("../../..")`**
- Better escape: create *device files*
- On Unix, all (non-network) devices have filenames
- Even physical memory has a filename
- Create a physical memory device, open it, and change the kernel data structures to remove the restriction
- Create a disk device, and mount a file system on it. Then **`chroot()`** to the real root

Trying Chroot

```
# mkdir /usr/jail /usr/jail/bin
# cp /bin/sh /usr/jail/bin/sh
# chroot /usr/jail /bin/sh
chroot: /bin/sh: Exec format error
# mkdir /usr/jail/libexec
# cp /libexec/ld.elf_so /usr/jail/libexec
# chroot /usr/jail /bin/sh
Shared object "libc.so.12" not found
# mkdir /usr/jail/lib
# cp /lib/libc.so.12 /usr/jail/lib
# chroot /usr/jail /bin/sh
Shared object "libedit.so.2" not found
```

Trying Chroot (Continued)

```
# cp /lib/libedit.so.2 /usr/jail/lib
# chroot /usr/jail /bin/sh
Shared object "libtermcap.so.0" not found
# cp /lib/libtermcap.so.0 /usr/jail/lib
# chroot /usr/jail /bin/sh
# ls
ls: not found
# echo jailed >/jail
# ^D
# ls -l /usr/jail
total 4
drwxr-xr-x  2 root  wheel  512 Nov  1 21:50 bin
-rw-r--r--  1 root  wheel    7 Nov  1 22:31 jail
drwxr-xr-x  2 root  wheel  512 Nov  1 22:31 lib
drwxr-xr-x  2 root  wheel  512 Nov  1 22:30 libexec
```

Summary of Chroot

- It's a good, but imperfect means of restricting file access
- It's fairly useless against root
- it doesn't provide other sorts of isolation

FreeBSD “Jail”

- Like **`chroot()`** on steroids
- Assign a separate network identity to a jail partition
- Restrict root’s abilities within a jail
- Intended for nearly-complete system emulation
- Network interactions with main system’s daemons

Sandboxes

- Very restricted environment, especially for network daemons
- *Assume* that the daemon will do *anything*
- Example: Janus traps each system call and validates it against policy
- Can limit I/O to certain paths

The Java Virtual Machine

- Java executables contain *byte code*, not machine language
- Java interpreter can enforce certain restrictions
- Java *language* prevents certain dangerous constructs and operations (unlike, for example, C)
- In theory, it's safe enough that web browsers can download byte code from arbitrary web sites

Is the JVM Secure?

- Heavy dependency on the semantics of the Java language
- The *byte code verifier* ensures that the code corresponds only to valid Java
- The *class loader* ensures that arguments to methods match properly
- Very complex process — not high assurance
- Bugs have been found, but they're fairly subtle
- But — there have been buffer overflows in the C support library

Using the JVM For Servers

- The dangers come from untrusted executables
- If you write your applications in Java, you don't have to worry about that
- The strict type system, the array bounds-checking, and the (optional) file access control all protect you from your own bugs
- Java is a very secure language for applications (if, of course, you're not too fussy about performance)

Virtual Machines

- Give the application an entire “machine”, down to the (virtual) bare silicon
- Run an entire operating system on this
- Run the untrusted application on that OS
- It can be *very* safe

How VMs Work

- Recall the hardware access control mechanisms: privileged operations and memory protection
- Run the guest operating system unprivileged
- Any time the guest OS issues a privileged operation, it traps to the *virtual machine monitor*
- The VMM emulates the operation. For example, an attempt at disk I/O is mapped to I/O to a real file that represents the virtual disk

Virtual Devices

- Virtual disks (or part or all of a real disk)
- Virtual screens, keyboards, and mice
- Virtual Ethernets
- Other virtual devices as needed

Virtual Machine Security

- *Very* strong isolation
- *Very* high overhead. . .
- Must set up and administer an entire OS

☞ Guest copies of Microsoft Windows require just as many patches as do native copies

- Performance can be bad

Using Virtual Machines

- Great for testing OS changes
- Great for student use
- Internet hosting companies
- Can use them for executing suspected viruses and worms — but some viruses detect the presence of the VMM and hide

Interacting with a Virtual Machine

- Often don't want perfect isolation.
- Example: cut-and-paste between windows
- Performance can be dramatically enhanced if the guest OS signals the VMM
- Example: add a virtual "graphics" driver that calls the VMM, via the equivalent of a system call

Limitations of Virtual Machines

- They can be *too* real
- Would you let your enemy put a machine inside your data center?
- VMs can spread viruses, launch DoS attacks, etc.
- VMs require just as much care, administration, and monitoring as do real machines
- In many situations, they represent an *economic* mechanism rather than a security mechanism

Other Isolation Mechanisms

- Light-weight VM systems, such as Solaris Zones
- Domain and type enforcement: limit file accesses by each executable
- Systrace (on some BSD operating systems) is similar
- Sub-operating system: permission overlay on top of file system, based on *subUIDs*
- Both require fairly complex permission-setting

The Limits of Isolation

- All of the mechanisms we've described are complex (but canned scripts can help)
- Most of them require root privileges
- As a consequence, they're useful for complex system designs, but not for general application isolation

Covert Channels

- We can block ordinary file accesses and network communication
- Are there other ways to leak information?
- Yes — *covert channels*
- Very important issue in a MAC world

MAC and Covert Channels

- One goal of MAC is to prevent leakage of information between a high-security process and a low-security process
- It's (relatively) easy to close the explicit communication channels, such as shared files or network connections
- There are more subtle ways to communicate
- Two types: *storage channels* and *timing channels*

Storage Channels

- Modulate some shared resource
- Example: create and delete files in a shared directory
- The files themselves need not be readable
- MAC systems often have per-level /tmp directories, to help avoid this problem

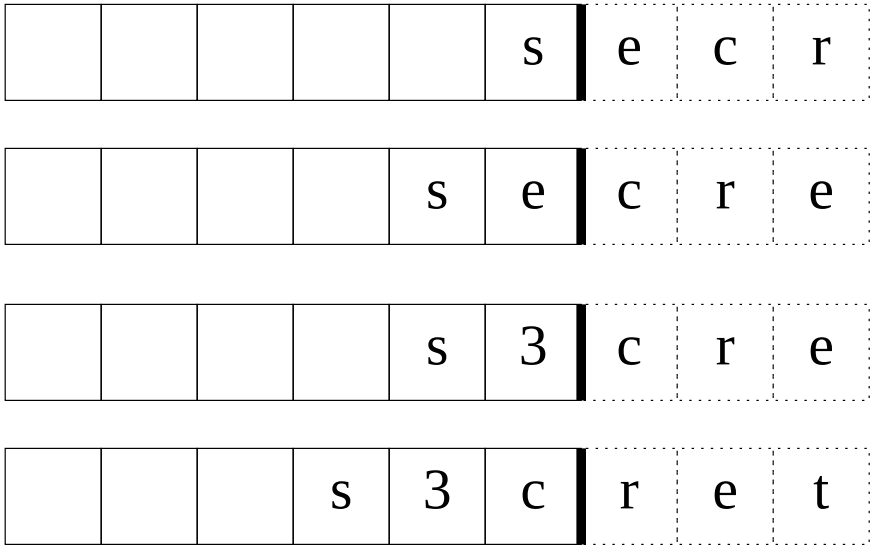
Timing Channels

- Modulate system timing in detectable way
- Example: do heavy disk I/O or refrain
- Receiver times how long it takes to do I/O operations

The Password-Checking Channel

- An old operating system (Tenex, for the PDP-10) checked (unhashed) passwords one byte at a time.
- Locate the password overlapping the end of virtual memory; ask the OS to check it
- If the first byte was wrong, it would return “fail”.
- If the byte was right, it would try to fetch the next byte, but take a page fault because it was past the edge
- Repeat as needed

Falling Off the Edge of the Earth



Defeating Covert Channels

- One approach — find them and eliminate them
- Bandwidth-limit them — cap the rate at which certain operations can be done
- Add noise to the channel