# Operating System Security
# Computer Security
Peter Reiher
April 21, 2016

# Outline

- What does the OS protect?
- Authentication for operating systems
- Memory protection
  - Buffer overflows
- IPC protection
  - Covert channels
- Stored data protection
  - Full disk encryption

# Introduction

- Operating systems provide the lowest layer of software visible to users
- Operating systems are close to the hardware
  - Often have complete hardware access
- If the operating system isn't protected, the machine isn't protected
- Flaws in the OS generally compromise all security at higher levels

# Why Is OS Security So Important?

- The OS controls access to application memory
- The OS controls scheduling of the processor
- The OS ensures that users receive the resources they ask for
- If the OS isn't doing these things securely, practically anything can go wrong
- So almost all other security systems must assume a secure OS at the bottom

# Single User Vs. Multiple User Machines

- The majority of today's computers usually support a single user
- Some computers are still multi-user
  - Often specialized servers
- Single user machines often run multiple processes, though
  - Often through downloaded code
- Increasing numbers of embedded machines
  - Effectively no (human) user

# Trusted Computing

- Since OS security is vital, how can we be sure our OS is secure?
- Partly a question of building in good security mechanisms
- But also a question of making sure you're running the right OS
  - And it's unaltered
- That's called *trusted computing*

# How Do We Achieve Trusted Computing?

- From the bottom up
- We need hardware we can count on
- It can ensure the boot program behaves
- The boot can make sure we run the right OS
- The OS will protect at the application level

# TPM and Bootstrap Security

- Trusted Platform Module (TPM)
  - Special hardware designed to improve OS security
- Proves OS was booted with a particular bootstrap loader
  - Using tamperproof HW and cryptographic techniques
- Also provides secure key storage and crypto support

# TPM and the OS Itself

- Once the bootstrap loader is operating, it uses TPM to check the OS
- Essentially, ensures that expected OS was what got booted
- OS can request TPM to verify applications it runs
- Remote users can request such verifications, too

# Transitive Trust in TPM

- You trust the app, because the OS says to trust it
- You trust the OS, because the bootstrap says to trust it
- You trust the bootstrap, because somebody claims it's OK
- You trust the whole chain, because you trust the TPM hardware's attestations

# Trust vs. Security

- TPM doesn't guarantee security
  - It (to some extent) verifies trust
- It doesn't mean the OS and apps are secure, or even non-malicious
- It just verifies that they are versions you have said you trust
- Offers some protection against tampering with software
- But doesn't prevent other bad behavior

# Status of TPM

- Hardware widely installed
  - Not widely used
- Microsoft Bitlocker uses it
  - When available
- A secure Linux boot loader and OS work with it
- Some specialized software uses TPM

# SecureBoot

- A somewhat different approach to ensuring you boot the right thing
- Built into the boot hardware and SW
- Designed by Microsoft
- Essentially, only allows booting of particular OS versions

# Some Details of SecureBoot

- Part of the Unified Extensible Firmware Interface (UEFI)
  - Replacement for BIOS
- Microsoft insists on HW supporting these features
- Only boots systems with pre-arranged digital signatures
- Some issues of who can set those

# Hardware Security Enclaves

- Similar idea to TPM and SecureBoot
- Build some elements of computer's security into hardware
  - Typically tamper-resistant
- OS relies on the HW for some security
- E.g., login security

# The iPhone Security Enclave

- How the iPhone enforces limited login attempts
- And hides the key used to decrypt data
- HW only provides the key on login attempt
- HW enforces counting of attempts
- Different models put different parts in HW

# Authentication in Operating Systems

- The OS must authenticate all user requests
  - Otherwise, can't control access to critical resources
- Human users log in
  - Locally or remotely
- Processes run on their behalf
  - And request resources

# In-Person User Authentication

- Authenticating the physically present user
- Most frequently using password techniques
- Sometimes biometrics
- To verify that a particular person is sitting in front of keyboard and screen

# Remote User Authentication

- Many users access machines remotely
- How are they authenticated?
- Most typically by password
- Sometimes via public key crypto
- Sometimes at OS level, sometimes by a particular process
  - In latter case, what is their OS identity?
  - What does that imply for security?

# Process Authentication

- Successful login creates a primal process
  - Under ID of user who logged in
- The OS securely ties a process control block to the process
  - Not under user control
  - Contains owner's ID
- Processes can fork off more processes
  - Usually child process gets same ID as parent
- Usually, special system calls can change a process' ID

# For Example,

- Process X wants to open file Y for read
- File Y has read permissions set for user Bill
- If process X belongs to user Bill, system ties the open call to that user
- And file system checks ID in open system call to file system permissions
- Other syscalls (e.g., RPC) similar

# Protecting Memory

- What is there to protect in memory?
- Page tables and virtual memory protection
- Special security issues for memory
- Buffer overflows

# What Is In Memory?

- Executable code
  - Integrity required to ensure secure operations
- Copies of permanently stored data
  - Secrecy and integrity issues
- Temporary process data
  - Mostly integrity issues

# Mechanisms for Memory Protection

- Most general purpose systems provide some memory protection
  - Logical separation of processes that run concurrently
- Usually through virtual memory methods
- Originally arose mostly for error containment, not security

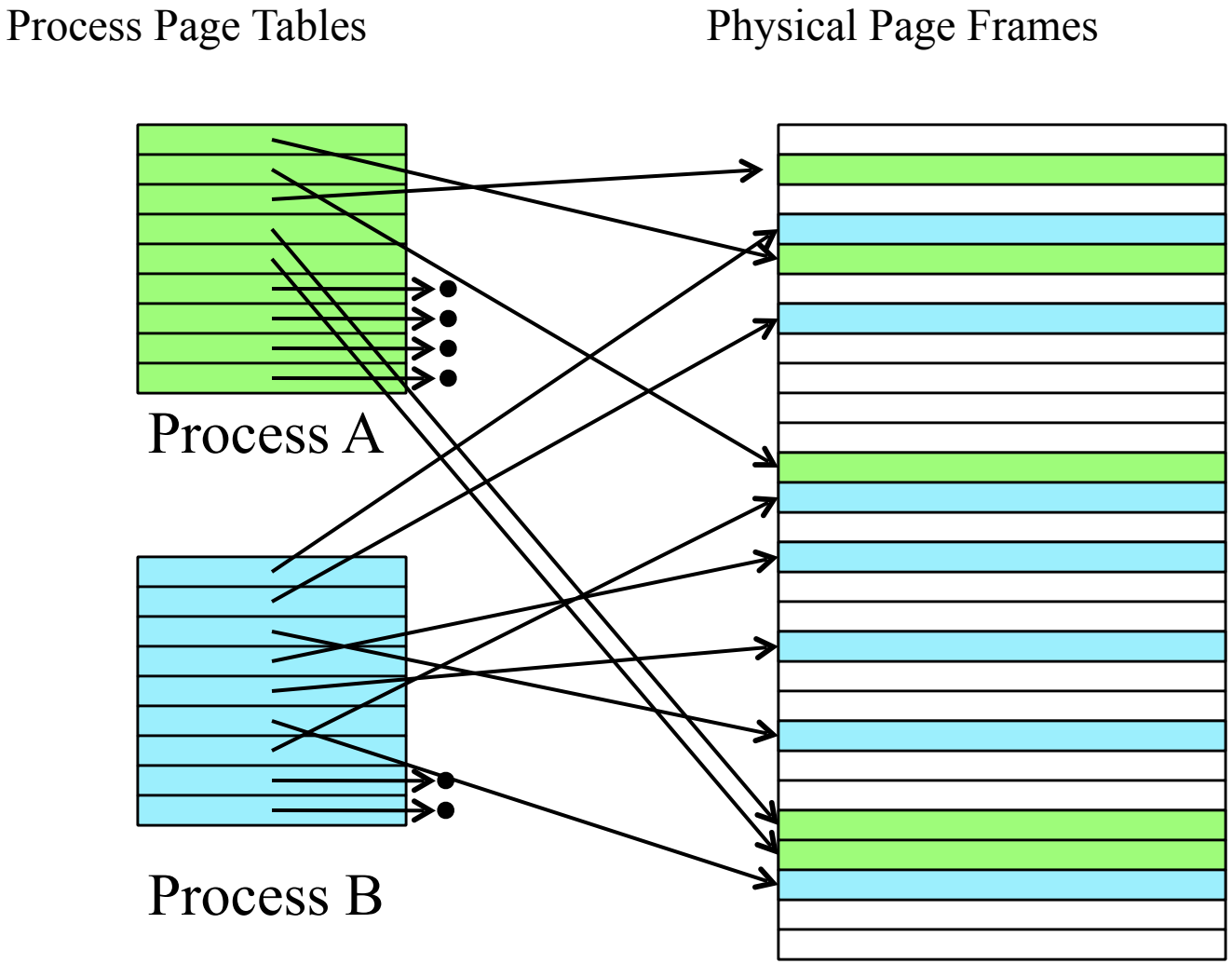
# Paging and Security

- Main memory is divided into page frames
- Every process has an address space divided into logical pages
- For a process to use a page, it must reside in a page frame
- If multiple processes are running, how do we protect their frames?

# Protection of Pages

- Each process is given a page table
  - Translation of logical addresses into physical locations
- All addressing goes through page table
  - At unavoidable hardware level
- If the OS is careful about filling in the page tables, a process can't even name other processes' pages

# Page Tables and Physical Pages




Any address Process A names goes through the green table

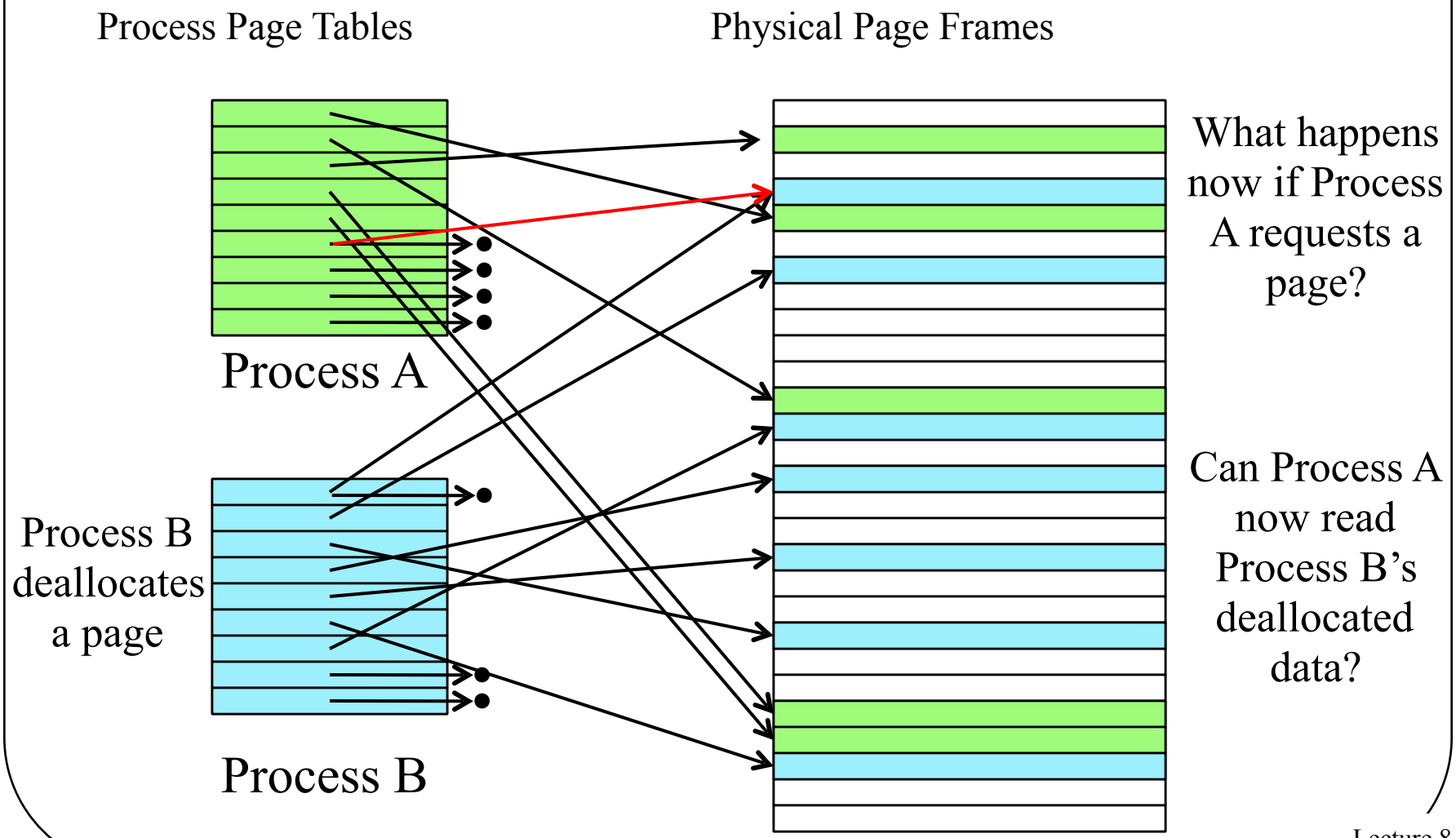
Any address Process B names goes through the blue table

They can't even name each other's pages

# Security Issues of Page Frame Reuse

- A common set of page frames is shared by all processes
- The OS switches ownership of page frames as necessary
- When a process acquires a new page frame, it used to belong to another process
  - Can the new process read the old data?

# Reusing Pages



# Strategies for Cleaning Pages

- Don't bother
  - Basic Linux strategy
- Zero on deallocation
- Zero on reallocation
- Zero on use
- Clean pages in the background
  - Windows strategy

# Special Interfaces to Memory

- Some systems provide a special interface to memory
- If the interface accesses physical memory,
  - And doesn't go through page table protections,
  - Then attackers can read the physical memory
  - Letting them figure out what's there and find what they're looking for

# Buffer Overflows

- One of the most common causes for compromises of operating systems
- Due to a flaw in how operating systems handle process inputs
  - Or a flaw in programming languages
  - Or a flaw in programmer training
  - Depending on how you look at it

# What Is a Buffer Overflow?

- A program requests input from a user
- It allocates a temporary buffer to hold the input data
- It then reads all the data the user provides into the buffer, but . . .
- It doesn't check how much data was provided

# For Example,

```
int main(){
  char name[32];
  printf("Please type your name:  ");
  gets(name);
  printf("Hello, %s", name);
  return (0);
}
```

- What if the user enters more than 32 characters?

# Well, What If the User Does?

- Code continues reading data into memory
- The first 32 bytes go into `name` buffer
  - Allocated on the stack
  - Close to record of current function
- The remaining bytes go onto the stack
  - Right after `name` buffer
  - Overwriting current function record
  - Including the instruction pointer

# Why Is This a Security Problem?

- The attacker can cause the function to "return" to an arbitrary address
- But all attacker can do is run different code than was expected
- He hasn't gotten into anyone else's processes
  - Or data
- So he can only fiddle around with his own stuff, right?

# Is That So Bad?

- Well, yes
- That's why a media player can write configuration and data files
- Unless roles and access permissions set up very carefully, a typical program can write all its user's files

# The Core Buffer Overflow Security Issue

- Programs often run on behalf of others
  - But using your identity
- Maybe OK for you to access some data
- But is it OK for someone who you're running a program for to access it?
  - Downloaded programs
  - Users of web servers
  - Many other cases

# Using Buffer Overflows to Compromise Security

- Carefully choose what gets written into the instruction pointer
- So that the program jumps to something you want to do
  - Under the identity of the program that's running
- Such as, execute a command shell
- Usually attacker provides this code

# Effects of Buffer Overflows

- A remote or unprivileged local user runs a program with greater privileges
- If buffer overflow is in a root program, it gets all privileges, essentially
- Can also overwrite other stuff
  - Such as heap variables
- Common mechanism to allow attackers to break into machines

# Stack Overflows

- The most common kind of buffer overflow
- Intended to alter the contents of the stack
- Usually by overflowing a dynamic variable
- Usually with intention of jumping to exploit code
  - Though it could instead alter parameters or variables in other frames
  - Or even variables in current frame

# Heap Overflows

- Heap is used to store dynamically allocated memory
- Buffers kept there can also overflow
- Generally doesn't offer direct ability to jump to arbitrary code
- But potentially quite dangerous

# What Can You Do With Heap Overflows?

- Alter variable values
- "Edit" linked lists or other data structures
- If heap contains list of function pointers, can execute arbitrary code
- Generally, heap overflows are harder to exploit than stack overflows
- But they exist
  - E.g., Google Chrome one discovered February 2012

# Some Recent Buffer Overflows

- Cisco Cable Modem software and Cisco Adaptive Security Appliance
- Pro-face GP Pro-EX industrial control system
- Xen Hypervisor
  - A heap overflow
- glibc
  - A bad place for a buffer overflow

# Fixing Buffer Overflows

- Write better code (check input lengths, etc.)
- Use programming languages that prevent them
- Add OS controls that prevent overwriting the stack
- Put things in different places on the stack, making it hard to find the return pointer (e.g., Microsoft ASLR)
- Don't allow execution from places in memory where buffer overflows occur (e.g., Windows DEP)
  - Or don't allow execution of writable pages
- Why aren't these things commonly done?
  - Sometimes they are, but not always effective
- When not, presumably because programmers and designers neither know nor care about security

# Protecting Interprocess Communications

- Operating systems provide various kinds of interprocess communications
  - Messages
  - Semaphores
  - Shared memory
  - Sockets
- How can we be sure they're used properly?

# IPC Protection Issues

- How hard it is depends on what you're worried about
- For the moment, let's say we're worried about one process improperly using IPC to get info from another
  - Process A wants to steal information from process B
- How would process A do that?

# How Can B Get the Secret?

- He can convince the system he's A
  - A problem for authentication
- He can break into A's memory
  - That doesn't use message IPC
  - And is handled by page tables
- He can forge a message from someone else to get the secret
  - But OS tags IPC messages with identities
- He can "eavesdrop" on someone else who gets the secret

# Can an Attacker Really Eavesdrop on IPC Message?

- On a single machine, what is a message send, really?
- Message copied from process buffer to OS buffer
  - Then from OS buffer to another process' buffer
  - Sometimes optimizations skip some copies
- If attacker can't get at processes' internal buffers and can't get at OS buffers, he can't "eavesdrop"
- Need to handle page reuse (discussed earlier)
- Other IPC mechanisms are similar

# So When Is It Hard?

1. Always possible that there's a bug in the operating system
   - Allowing masquerading, eavesdropping, etc.
   - Or, if the OS itself is compromised, all bets are off
2. What if it's not a single machine?
3. What if the OS has to prevent cooperating processes from sharing information?

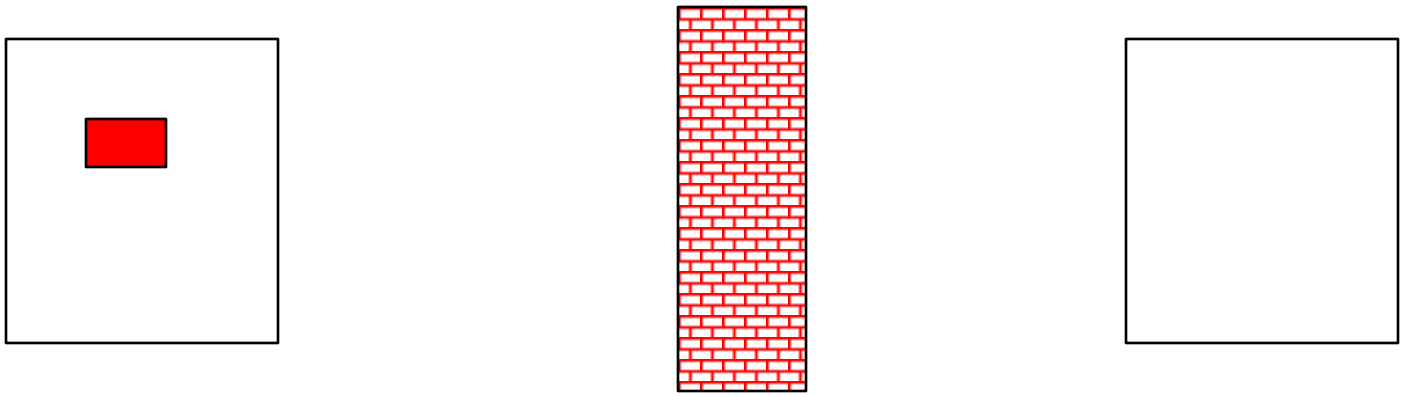
# Distributed System Issues

- What if your RPC is really remote?
- Goal of RPC is to make remote access transparent
  - Looks "just like" local
- The hard part is authentication
  - The call didn't come from your own OS
  - How do you authenticate its origin?

# The Other Hard Case

Process A

Process B



Process A wants to tell the secret to process B
But the OS has been instructed to prevent that
A necessary part of Bell-La Padula, e.g.
Can the OS prevent A and B from colluding to get the secret to B?

# OS Control of Interactions

- OS can "understand" the security policy
- Can maintain labels on files, process, data pages, etc.
- Can regard any IPC or I/O as a possible leak of information
  - To be prohibited if labels don't allow it

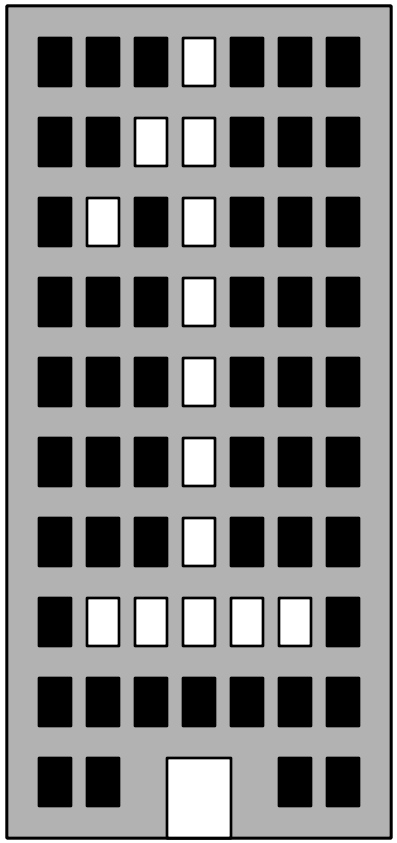
# Covert Channels

- Tricky ways to pass information
- Requires cooperation of sender and receiver

Text

  - Generally in active attempt to deceive system
- Use something not ordinarily regarded as a communications mechanism

Covert Channles are secret



# Covert Channels in Computers

- Generally, one process "sends" a covert message to another
  - But could be computer to computer
- How?
  - Disk activity
  - Page swapping
  - Time slice behavior
  - Use of a peripheral device
  - Limited only by imagination

# Handling Covert Channels

- Relatively easy if you know details of how the channel is used
  - Put randomness/noise into channel to wash out message
- Hard to impossible if you don't know what the channel is
- Not most people's problem

# Stored Data Protection

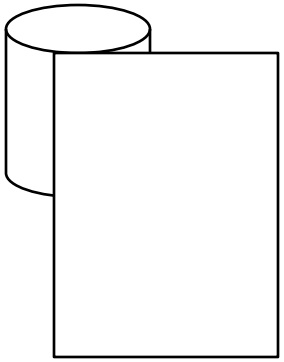
- Files are a common example of a typically shared resource
- If an OS supports multiple users, it needs to address the question of file protection
- Simple read/write access control
- What else do we need to do?
- Protect the raw disk or SSD

# Encrypted File Systems

- Data stored on disk is subject to many risks
  - Improper access through OS flaws
  - But also somehow directly accessing the disk
- If the OS protections are bypassed, how can we protect data?
- How about if we store it in encrypted form?

# An Example of an Encrypted File System

Issues for encrypted file systems:



$K_s$

Sqzmsfedq
$000 to my
szuhngsfr
abbcotnts

When does the cryptography occur?

Where does the key come from?

What is the granularity of cryptography?

encrypt files? disk?

# When Does Cryptography Occur?

- Transparently when a user opens a file?
  - In disk drive?
  - In OS?
  - In file system?
- By explicit user command?
  - Or always, implicitly?
- How long is the data decrypted?
- Where does it exist in decrypted form?

# Where Does the Key Come From?

- Provided by human user?
- Stored somewhere in file system?
- Stored on a smart card?
- Stored in the disk hardware?
- Stored on another computer?
- Where and for how long do we store the key?

# What Is the Granularity of Cryptography?

- An entire disk?
- An entire file system?
- Per file?
- Per block?
- Consider both in terms of:
  - How many keys?
  - When is a crypto operation applied?

# What Are You Trying to Protect Against With Crypto File Systems?

- Unauthorized access by improper users?
  - Why not just access control?
- The operating system itself?
  - What protection are you really getting?
  - Unless you're just storing data on the machine
- Data transfers across a network?
  - Why not just encrypt while in transit?
- Someone who accesses the device not using the OS?
  - A realistic threat in your environment?

threat because they can attack you

# Full Disk Encryption

- All data on the disk is encrypted
- Data is encrypted/decrypted as it enters/ leaves disk
- Primary purpose is to prevent improper access to stolen disks
  - Designed mostly for portable machines (laptops, tablets, etc.)

allows you to encrpt your etnrie despreevnt ipreaccessed

# HW Vs. SW Full Disk Encryption

**SW Full disk encryption is thus much more common**

- HW advantages:
  - Faster
  - Totally transparent, works for any OS
  - Setup probably easier
- HW disadvantages:
  - Not ubiquitously available today
  - More expensive (not that much, though)
  - Might not fit into a particular machine
  - Backward compatibility

# Example of Software Full Disk Encryption

- Microsoft BitLocker
- Doesn't encrypt quite the whole drive
  - Unencrypted partition holds bootstrap
- Uses AES for cryptography
- Key stored either in special hardware or USB drive
- Microsoft claims "single digit percentage" overhead
  - One independent study claims 12%