Operating System Security
CS 136
Computer Security
Peter Reiher
January 24, 2008

Outline

- Introduction
- Memory protection
- Buffer overflows
- Interprocess communications protection
- File protection and 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
  - Sometimes one at a time, sometimes only one ever
- Some computers are still multi-user
  - Mainframes
  - Servers
  - Network-of-workstation machines
- Single user machines often run multiple processes, though

Server Machines Vs. General Purpose Machines

- Most server machines provide only limited services
  - Web page access
  - File access
  - DNS lookup
- Security problems are simpler for them
- Some machines still provide completely general service, though
- And many server machines can run general services . . .

Downloadable Code and Single User Machines

- Applets and other downloaded code should run in a constrained mode
- Using access control on a finer granularity than the user
- Essentially the same protection problem as multiple users

Mechanisms for Secure Operating Systems

- Most operating system security is based on separation
  - Keep the bad guys away from the good stuff
  - Since you don't know who's bad, separate most things

Separation Methods

- Physical separation
  - Different machines
- Temporal separation
  - Same machine, different times
- Logical separation
  - HW/software enforcement
  - Possibly VM technology
- Cryptographic separation

The Problem of Sharing

- Separating stuff is actually pretty easy
- The hard problem is allowing controlled sharing
- How can the OS allow users to share exactly what they intend to share?
  - In exactly the ways they intend

Levels of Sharing Protection

- None
- Isolation
- All or nothing
- Access limitations
- Limited use of an object

Protecting Memory

- 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

Security Aspects of Paging

- 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

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?

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,
  - Attackers can read the physical memory
  - Then 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?

- The code continues reading data into memory
  - That's how `gets()` works
- The first 32 bytes go into `name`
- Where do the remaining bytes go?
- Onto the stack

Munging the Stack

- The temporary variable `name` is allocated on the stack
  - Close to the record of the function currently being run
- The overflow will spill into whatever's next on the stack
- If it overflows enough, it will overwrite the instruction pointer
- When the function exits, it will go to the overwritten pointer, not where it came from

Why Is This a Security Problem?

- 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 are often run on behalf of others
  - But using your identity
- Maybe it's OK for you to access some data
- But is it OK for someone who you're running a program for?

But I Never Run Programs for Anyone Else

- Oh, yes, you do
- Every time you download any form of executable
- Every time you download a file containing an executable
- Every time you allow someone to remotely access data on your system
  - E.g., via a web server
- In all cases, you're doing something for someone else

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

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 could be to 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., Microsoft CVE-2007-0948
    - Allowed VM to escape confinement

Are Buffer Overflows Common?

- You bet!
- Weekly occurrences in major systems/applications
  - Mostly stack overflows
- Probably one of the most common security bugs

Some Recent Buffer Overflows

- Cisco Security Agent for Windows
  - They should have known better
- HP OpenView Network Node Manager
  - They should have, too
- IBM Lotus Notes
  - Them, too
- 3ivx MPEG-4 Codec
- And more than 15 others in December 2007 alone
  - In code written by everyone from Microsoft to tiny software shops

Fixing Buffer Overflows

- Check the length of the input
- 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
- Don't allow execution from places in memory where buffer overflows occur (E.g., Windows DEP)
- Why aren't these things commonly done?
  - Sometimes they are
- Presumably because programmers and designers neither know nor care about security

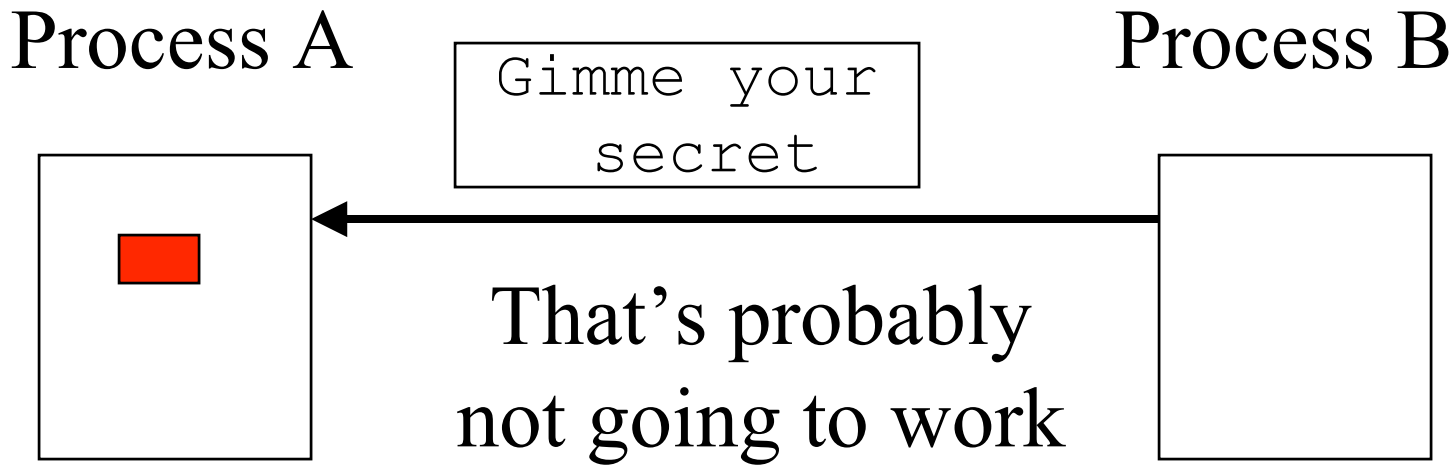
Protecting Interprocess Communications

- 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?

Message Security

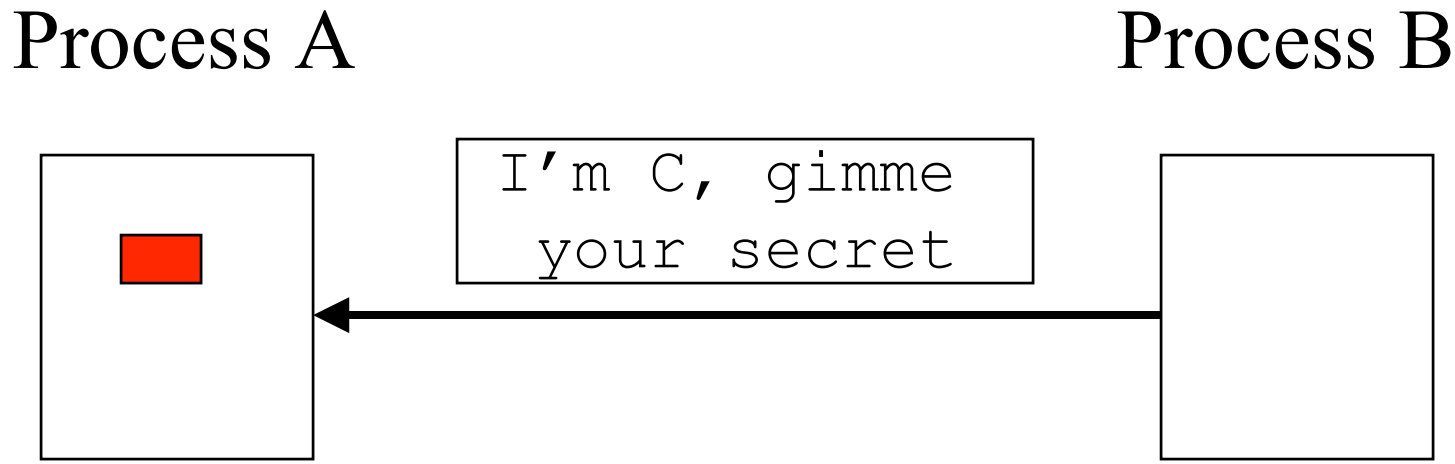


Can process B use message-based IPC to steal the secret?

How Can B Get the Secret?

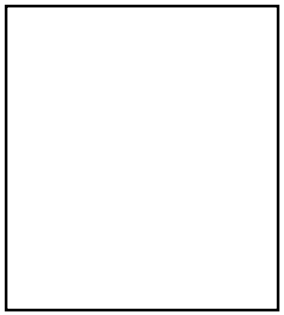
- He can convince the system he's A
  - A problem for authentication
- He can break into A's memory
  - That doesn't use message IPC
  - And is handled by page tables
- He can forge a message from someone else to get the secret
- He can "eavesdrop" on someone else who gets the secret

Forging An Identity



Will A know B is lying?

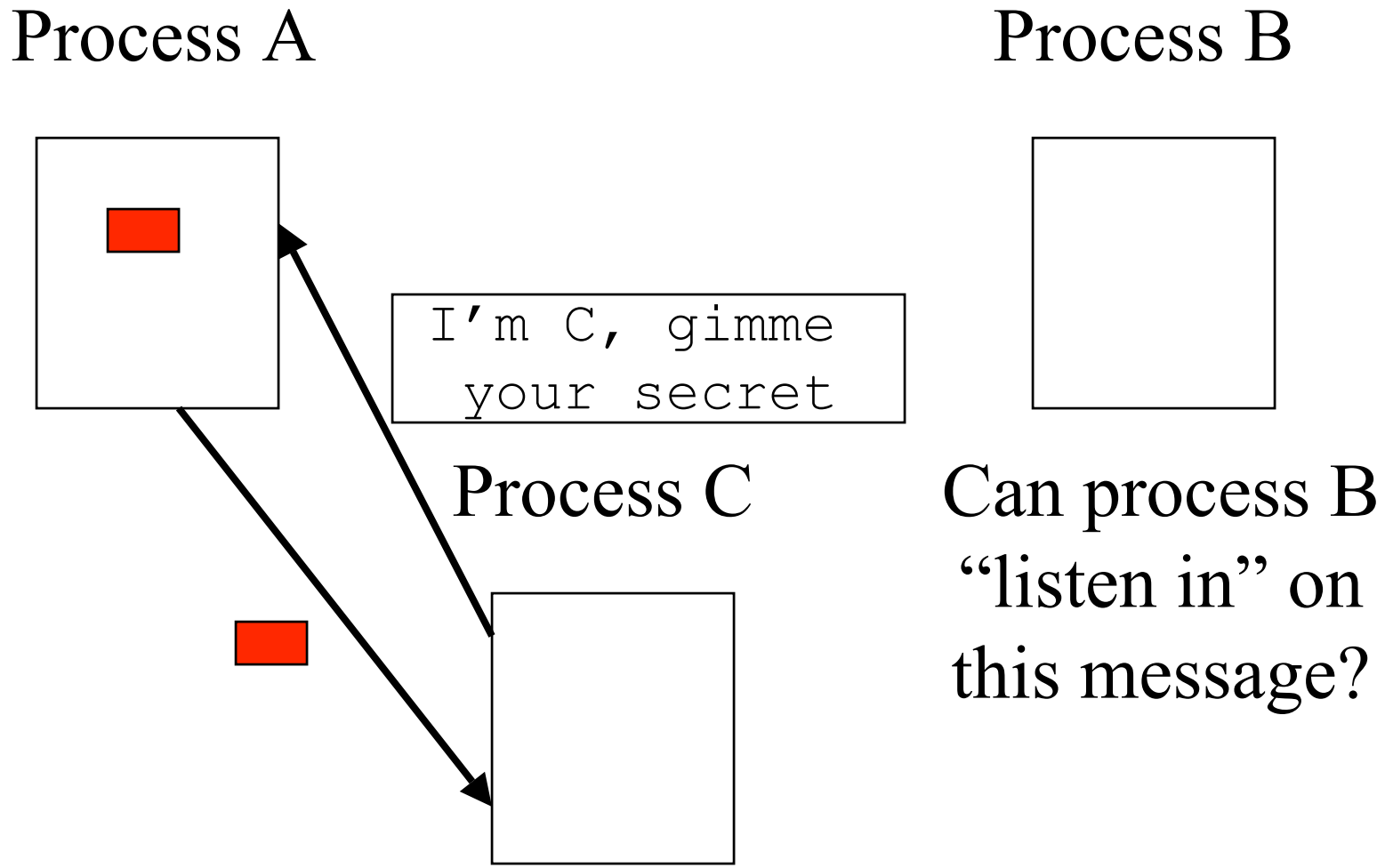
Process C



Operating System Protections

- The operating system knows who each process belongs to
- It can tag the message with the identity of the sender
  - Raw message never available outside the OS
- If the receiver cares, he can know the identity

How About Eavesdropping?



What's Really Going on Here?

- On a single machine, what is a message send, really?
- A message is copied from a process buffer to an OS buffer
  - Then from the 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"

Returning to an Earlier Issue

- What are buffers, really?
- Data held in memory pages
- Really in page frames
- Page frames are shared
  - Serially
- Will the page frame I allocate contain data from its last user?

Avoiding Page Frame "Eavesdropping"

- Zero pages on deallocation
- Zero pages on allocation
- Mark pages as unreadable until a process writes them
  - Need to ensure partial write doesn't clear the mark entirely

Other Forms of IPC

- Semaphores, sockets, shared memory, RPC
- Pretty much all the same
  - Use system calls for access
  - Which belong to some process
  - Which belongs to some principal
  - OS can check principal against access control permissions at syscall time
  - Ultimately, data is held in some type of memory
    - Which shouldn't be improperly accessible

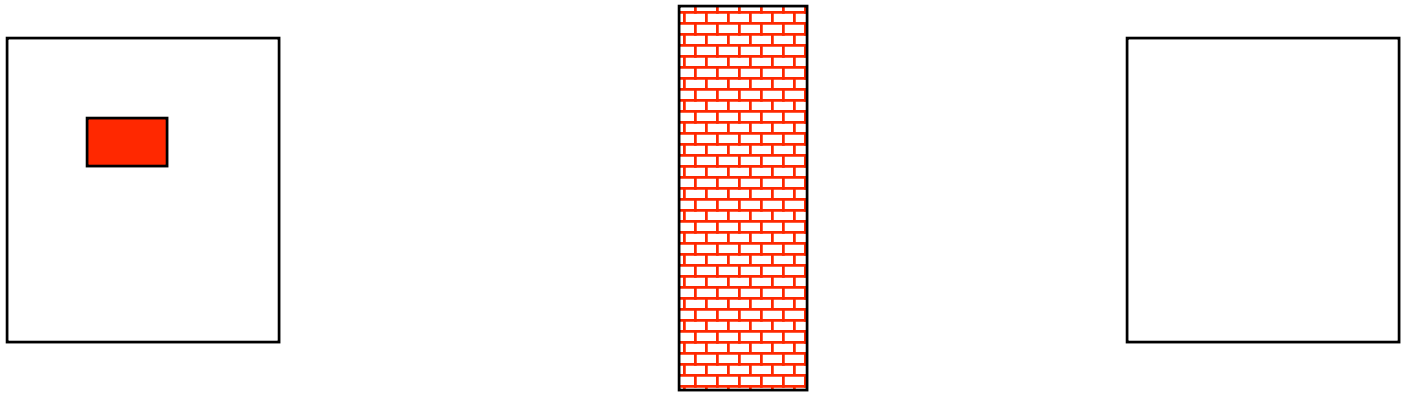
So When Is It Hard?

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 the OS has to prevent cooperating processes from sharing information?

The 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

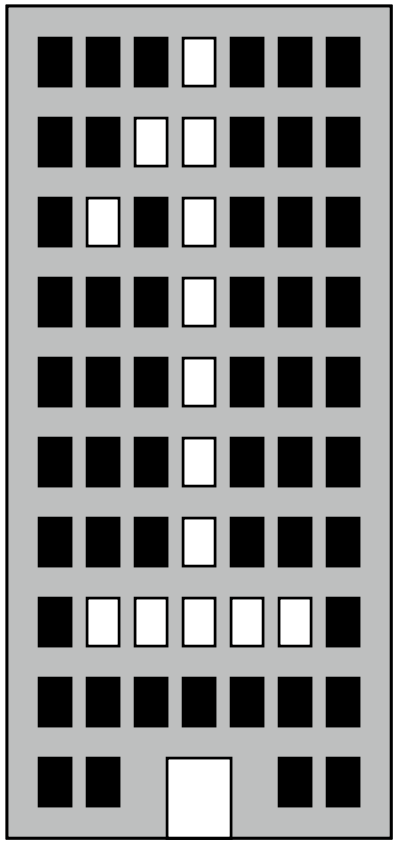
- OS can "understand" the security policy
- Can maintain labels on files, process, data pages, etc.
- Can regard any IPC or I/O as a possible leak of information
  - To be prohibited if labels don't allow it

Example

- Bell-LaPadula doesn't allow writedown
- Process A is at Top Secret clearance
- It tries to send a message to process B
  - Which is at Secret clearance
- OS understands Bell-LaPadula
  - Observes illegal access and prevents the IPC

Covert Channels

- Tricky ways to pass information
- Requires cooperation of sender and receiver
  - Generally in active attempt to deceive system
- Use something not ordinarily regarded as a communications mechanism



Covert Channels in Computers

- Generally, one process "sends" 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 what the channel is
  - 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

Dangers for Operating System Security

- Bugs in the OS
  - Not checking security, allowing access to protected resources, etc.
- Privileged users and roles
  - Superusers often can do anything
- Untrusted applications and overly broad security domains

File Protection

- How do we apply these access protection mechanisms to a real system resource?
- 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

Unix File Protection

- A model for protecting files developed in the 1970s
- Still in very wide use today
  - With relatively few modifications
- To review, three subjects
    - Owner, group, other
- and three modes
  - Read, write, execute
  - Sometimes these have special meanings

Setuid/Setgid Programs

- Unix mechanisms for changing your user identity and group identity
- Either indefinitely or for the run of a single program
- Created to deal with inflexibilities of the Unix access control model
- But the source of endless security problems

Why Are Setuid Programs Necessary?

- The print queue is essentially a file
- Someone must own that file
- How will other people put stuff in the print queue?
  - Without making the print queue writeable for all purposes
- Typical Unix answer is run the printing program setuid
  - To the owner of the print queue

Why Are Setuid Programs Dangerous?

- Essentially, setuid programs expand a user's security domain
- In an encapsulated way
  - Abilities of the program limit the operations in that domain
- Need to be damn sure that the program's abilities are limited

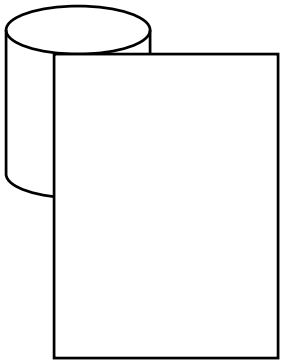
Some Examples of Setuid Dangers

- Setuid programs that allow forking of a new shell
- Setuid programs with powerful debugging modes
- Setuid programs with "interesting" side effects
  - E.g., `lpr` options that allow file deletion

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



K_s

Sqamsẹdq
$099 sn
hy
sauhmgs
abbntms

Issues for encrypted file systems:

When does the cryptography occur?

Where does the key come from?

What is the granularity of cryptography?

When Does Cryptography Occur?

- Transparently when user opens 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 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?
- 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?

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 laptops

Hardware Vs. Software Full Disk Encryption

- HW advantages:
  - Probably faster
  - Totally transparent, works for any OS
  - Setup probably easier
- HW disadvantages:
  - Not ubiquitously available today
  - More expensive (not that much, though - ~$90 vs. ~$50 for 80Gbyte disk)
  - Might not fit into a particular machine
  - Backward compatibility

An Example of Hardware Full Disk Encryption

- Seagate's Momentus 5400 FDE product line
- Hardware encryption for entire disk
  - Using AES
- Key accessed via user password
  - Hashed password stored on disk
  - Check performed by the disk itself, pre-boot
  - 44 Mbytes/sec sustained transfer rate
- Primarily for laptops

Example of Software Full Disk Encryption

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