Chapter 9: Security
Security

- The security environment
- Basics of cryptography
- User authentication
- Attacks from inside the system
- Attacks from outside the system
- Protection mechanisms
- Trusted systems
Security environment: threats

<table>
<thead>
<tr>
<th>Goal</th>
<th>Threat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data confidentiality</td>
<td>Exposure of data</td>
</tr>
<tr>
<td>Data integrity</td>
<td>Tampering with data</td>
</tr>
<tr>
<td>System availability</td>
<td>Denial of service</td>
</tr>
</tbody>
</table>

- Operating systems have goals
  - Confidentiality
  - Integrity
  - Availability
- Someone attempts to subvert the goals
  - Fun
  - Commercial gain
What kinds of intruders are there?

- Casual prying by nontechnical users
  - Curiosity
- Snooping by insiders
  - Often motivated by curiosity or money
- Determined attempt to make money
  - May not even be an insider
- Commercial or military espionage
  - This is very big business!
Accidents cause problems, too…

- Acts of God
  - Fires
  - Earthquakes
  - Wars (is this really an “act of God”?)

- Hardware or software error
  - CPU malfunction
  - Disk crash
  - Program bugs (hundreds of bugs found in the most recent Linux kernel)

- Human errors
  - Data entry
  - Wrong tape mounted
  - rm * .o
Cryptography

- Goal: keep information from those who aren’t supposed to see it
  - Do this by “scrambling” the data
- Use a well-known algorithm to scramble data
  - Algorithm has two inputs: data & key
  - Key is known only to “authorized” users
  - Relying upon the secrecy of the algorithm is a very bad idea (see WW2 Enigma for an example…)
- Cracking codes is very difficult, Sneakers and other movies notwithstanding
Cryptography basics

- Algorithms (E, D) are widely known
- Keys (K_E, K_D) may be less widely distributed
- For this to be effective, the ciphertext should be the only information that’s available to the world
- Plaintext is known only to the people with the keys (in an ideal world…)
Secret-key encryption

- Also called symmetric-key encryption
- Monoalphabetic substitution
  - Each letter replaced by different letter
- Vignere cipher
  - Use a multi-character key
    THEMESSAGE
    ELMELMELME
    XSQQPEWLSI
- Both are easy to break!
- Given the encryption key, easy to generate the decryption key
- Alternatively, use different (but similar) algorithms for encryption and decryption
Modern encryption algorithms

- **Data Encryption Standard (DES)**
  - Uses 56-bit keys
  - Same key is used to encrypt & decrypt
  - Keys used to be difficult to guess
    - Needed to try $2^{55}$ different keys, on average
    - Modern computers can try millions of keys per second with special hardware
    - For $250K$, EFF built a machine that broke DES quickly

- **Current algorithms (AES, Blowfish)** use 128 bit keys
  - Adding one bit to the key makes it twice as hard to guess
  - Must try $2^{127}$ keys, on average, to find the right one
  - At $10^{15}$ keys per second, this would require over $10^{21}$ seconds, or 1000 billion years!
  - Modern encryption isn’t usually broken by brute force…
Unbreakable codes

- There is such a thing as an unbreakable code: one-time pad
  - Use a truly random key as long as the message to be encoded
  - XOR the message with the key a bit at a time
- Code is unbreakable because
  - Key could be anything
  - Without knowing key, message could be anything with the correct number of bits in it
- Difficulty: distributing key is as hard as distributing message
- Difficulty: generating truly random bits
  - Can’t use computer random number generator!
  - May use physical processes
    - Radioactive decay
    - Leaky diode
    - Lava lamp (!) [http://www.sciencenews.org/20010505/mathtrek.asp]
Public-key cryptography

- Instead of using a single shared secret, keys come in pairs
  - One key of each pair distributed widely (*public key*), $K_p$
  - One key of each pair kept secret (*private or secret key*), $K_s$
  - Two keys are inverses of one another, but not identical
  - Encryption & decryption are the same algorithm, so
    \[ E(K_p, E(K_s, M)) = E(K_s, E(K_p, M)) = M \]
- Currently, most popular method involves primes and exponentiation
  - Difficult to crack unless large numbers can be factored
  - Very slow for large messages
The RSA algorithm for public key encryption

- Public, private key pair consists of $K_p = (d,n)$ $K_s = (e,n)$
  - $n = p \times q$ (p and q are large primes)
  - d is a randomly chosen integer with GCD $(d, (p-1) \times (q-1)) = 1$
  - e is an integer such that $(e \times d) \mod (p-1) \times (q-1) = 1$
- p & q aren’t published, and it’s hard to find them: factoring large numbers is thought to be NP-hard
- Public key is published, and can be used by anyone to send a message to the private key’s owner
- Encryption & decryption are the same algorithm: $E(K_p,M) = M^d \mod n$ (similar for $K_s$)
  - Methods exist for doing the above calculation quickly, but...
  - Exponentiation is still very slow
  - Public key encryption not usually done with large messages
One-way functions

- Function such that
  - Given formula for $f(x)$, easy to evaluate $y = f(x)$
  - Given $y$, computationally infeasible to find any $x$ such that $y = f(x)$

- Often, operate similar to encryption algorithms
  - Produce fixed-length output rather than variable length output
  - Similar to XOR-ing blocks of ciphertext together

- Common algorithms include
  - MD5: 128-bit result
  - SHA-1: 160-bit result
Digital signatures

- Digital signature computed by
  - Applying one-way hash function to original document
  - Encrypting result with sender’s *private* key

- Receiver can verify by
  - Applying one-way hash function to received document
  - Decrypting signature using sender’s public key
  - Comparing the two results: equality means document unmodified
Pretty Good Privacy (PGP)

- Uses public key encryption
  - Facilitates key distribution
  - Allows messages to be sent encrypted to a person (encrypt with person’s public key)
  - Allows person to send message that must have come from her (encrypt with person’s private key)

- Problem: public key encryption is very slow

- Solution: use public key encryption to exchange a shared key
  - Shared key is relatively short (~128 bits)
  - Message encrypted using symmetric key encryption

- PGP can also be used to authenticate sender
  - Use digital signature and send message as plaintext
User authentication

- Problem: how does the computer know who you are?
- Solution: use *authentication* to identify
  - Something the user knows
  - Something the user has
  - Something the user is
- This must be done before user can use the system
- Important: from the computer’s point of view…
  - Anyone who can duplicate your ID is you
  - Fooling a computer isn’t all that hard…
## Authentication using passwords

<table>
<thead>
<tr>
<th>Login: <strong>elm</strong></th>
<th>Login: <strong>jimp</strong></th>
<th>Login: <strong>elm</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Password: <strong>foobar</strong></td>
<td>User not found!</td>
<td>Password: <strong>barfle</strong></td>
</tr>
<tr>
<td>Welcome to Linux!</td>
<td></td>
<td>Invalid password!</td>
</tr>
</tbody>
</table>

- Successful login lets the user in
- If things don’t go so well…
  - Login rejected after name entered
  - Login rejected after name and incorrect password entered
- Don’t notify the user of incorrect user name until *after* the password is entered!
  - Early notification can make it easier to guess valid user names
Dealing with passwords

- **Passwords should be memorable**
  - Users shouldn’t need to write them down!
  - Users should be able to recall them easily

- **Passwords shouldn’t be stored “in the clear”**
  - Password file is often readable by all system users!
  - Password must be checked against entry in this file

- **Solution: use hashing to hide “real” password**
  - One-way function converting password to meaningless string of digits (Unix password hash, MD5, SHA-1)
  - Difficult to find another password that hashes to the same random-looking string
  - Knowing the hashed value and hash function gives no clue to the original password
Salting the passwords

- Passwords can be guessed
  - Hackers can get a copy of the password file
  - Run through dictionary words and names
    - Hash each name
    - Look for a match in the file

- Solution: use “salt”
  - Random characters added to the password before hashing
  - Salt characters stored “in the clear”
  - Increase the number of possible hash values for a given password
    - Actual password is “pass”
    - Salt = “aa” => hash “passaa”
    - Salt = “bb” => hash “passbb”
  - Result: cracker has to try many more combinations

- Mmmm, salted passwords!
Sample breakin (from LBL)

LBL> `telnet elxsi`
ELXSI AT LBL
LOGIN: `root`
PASSWORD: `root`
INCORRECT PASSWORD, TRY AGAIN
LOGIN: `guest`
PASSWORD: `guest`
INCORRECT PASSWORD, TRY AGAIN
LOGIN: `uucp`
PASSWORD: `uucp`
WELCOME TO THE ELXSI COMPUTER AT LBL

Moral: change all the default system passwords!
Authentication using a physical object

- **Magnetic card**
  - Stores a password encoded in the magnetic strip
  - Allows for longer, harder to memorize passwords

- **Smart card**
  - Card has secret encoded on it, but not externally readable
  - Remote computer issues challenge to the smart card
  - Smart card computes the response and proves it knows the secret
Authentication using biometrics

- Use basic body properties to prove identity
- Examples include
  - Fingerprints
  - Voice
  - Hand size
  - Retina patterns
  - Iris patterns
  - Facial features
- Potential problems
  - Duplicating the measurement
  - Stealing it from its original owner?
Countermeasures

- Limiting times when someone can log in
- Automatic callback at number prespecified
  - Can be hard to use unless there’s a modem involved
- Limited number of login tries
  - Prevents attackers from trying lots of combinations quickly
- A database of all logins
- Simple login name/password as a trap
  - Security personnel notified when attacker bites
  - Variation: allow anyone to “log in,” but don’t let intruders do anything useful
Attacks on computer systems

- Trojan horses
- Logic bombs
- Trap doors
- Viruses
- Exploiting bugs in OS code
Trojan horses

- Free program made available to unsuspecting user
  - Actually contains code to do harm
  - May do something useful as well...

- Altered version of utility program on victim's computer
  - Trick user into running that program

- Example (getting superuser access on CATS?)
  - Place a file called `ls` in your home directory
    - File creates a shell in `/tmp` with privileges of whoever ran it
    - File then actually runs the real `ls`
  - Complain to your sysadmin that you can’t see any files in your directory
  - Sysadmin runs `ls` in your directory
    - Hopefully, he runs `your ls` rather than the real one (depends on his search path)
Login spoofing

- No difference between real & phony login screens
- Intruder sets up phony login, walks away
- User logs into phony screen
  - Phony screen records user name, password
  - Phony screen prints “login incorrect” and starts real screen
  - User retypes password, thinking there was an error
- Solution: don’t allow certain characters to be “caught”
Logic bombs

- Programmer writes (complex) program
  - Wants to ensure that he’s treated well
  - Embeds logic “flaws” that are triggered if certain things aren’t done
    - Enters a password daily (weekly, or whatever)
    - Adds a bit of code to fix things up
    - Provides a certain set of inputs
    - Programmer’s name appears on payroll (really!)

- If conditions aren’t met
  - Program simply stops working
  - Program may even do damage
    - Overwriting data
    - Failing to process new data (and not notifying anyone)

- Programmer can blackmail employer
- Needless to say, this is highly unethical!
Chapter 9: Security

Normal code

Code with trapdoor

Trap door: user’s access privileges coded into program

Example: “joshua” from *Wargames*
Buffer overflow is a big source of bugs in operating systems
- Most common in user-level programs that help the OS do something
- May appear in “trusted” daemons

Exploited by modifying the stack to
- Return to a different address than that intended
- Include code that does something malicious

Accomplished by writing past the end of a buffer on the stack.
Generic security attacks

- Request memory, disk space, tapes and just read
- Try illegal system calls
- Start a login and hit DEL, RUBOUT, or BREAK
- Try modifying complex OS structures
- Try to do specified DO NOTs
- Social engineering
  - Convince a system programmer to add a trap door
  - Beg admin's secretary (or other people) to help a poor user who forgot password
  - Pretend you’re tech support and ask random users for their help in debugging a problem
Security flaws: TENEX password problem

First page (in memory)

Second page (not in memory)

Page boundary
Design principles for security

- System design should be public
- Default should be no access
- Check for current authority
- Give each process least privilege possible
- Protection mechanism should be
  - Simple
  - Uniform
  - In the lowest layers of system
- Scheme should be psychologically acceptable
- Biggest thing: keep it simple!
Security in a networked world

- External threat
  - Code transmitted to target machine
  - Code executed there, doing damage

- Goals of virus writer
  - Quickly spreading virus
  - Difficult to detect
  - Hard to get rid of
  - Optional: does something malicious

- Virus: embeds itself into other (legitimate) code to reproduce and do its job
  - Attach its code to another program
  - Additionally, may do harm
Virus damage scenarios

- Blackmail
- Denial of service as long as virus runs
- Permanently damage hardware
- Target a competitor's computer
  - Do harm
  - Espionage
- Intra-corporate dirty tricks
  - Practical joke
  - Sabotage another corporate officer's files
How viruses work

- Virus language
  - Assembly language: infects programs
  - “Macro” language: infects email and other documents
    - Runs when email reader / browser program opens message
    - Program “runs” virus (as message attachment) automatically
  - Inserted into another program
    - Use tool called a “dropper”
    - May also infect system code (boot block, etc.)
  - Virus dormant until program executed
    - Then infects other programs
    - Eventually executes its “payload”
How viruses find executable files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>

struct stat sbuf;

search(char *dir_name) {
    DIR *dirp;
    struct dirent *dp;

dirp = opendir(dir_name);
    if (dirp == NULL) return;
    while (TRUE) {
        dp = readdir(dirp);
        if (dp == NULL) {
            chdir("..");
            break;
        }
        if (dp->d_name[0] == '.') continue;
        lstat(dp->d_name, &sbuf);
        if (S_ISLNK(sbuf.st_mode)) continue; // skip symbolic links
        if (chdir(dp->d_name) == 0) {
            search("..");
        } else {
            if (access(dp->d_name, X_OK) == 0) /* if executable, infect it */
                infect(dp->d_name);
        }
    }
    closedir(dirp);
    /* dir processed; close and return */
}
```

- Recursive procedure that finds executable files on a UNIX system
- Virus can infect some or all of the files it finds
  - Infect all: possibly wider spread
  - Infect some: harder to find?
Where viruses live in the program

- **Uninfected program**
  - Executable program
  - Header

- **Virus at start of program**
  - Executable program
  - Virus
  - Header

- **Virus at end of program**
  - Executable program
  - Virus
  - Header

- **Virus in program’s free spaces**
  - Executable program
  - Virus
  - Virus
  - Header
Viruses infecting the operating system

- Operating system
- Virus
  - Syscall traps
  - Disk vector
  - Clock vector
  - Kbd vector

Virus has captured interrupt & trap vectors

- Operating system
- Virus
  - Syscall traps
  - Disk vector
  - Clock vector
  - Kbd vector

OS retakes keyboard vector

- Operating system
- Virus
  - Syscall traps
  - Disk vector
  - Clock vector
  - Kbd vector

Virus notices, recaptures keyboard
How do viruses spread?

- Virus placed where likely to be copied
  - Popular download site
  - Photo site
- When copied
  - Infects programs on hard drive, floppy
  - May try to spread over LAN or WAN
- Attach to innocent looking email
  - When it runs, use mailing list to replicate
  - May mutate slightly so recipients don’t get suspicious
Hiding a virus in a file

- Start with an uninfected program
- Add the virus to the end of the program
  - Problem: file size changes
  - Solution: compression
- Compressed infected program
  - Decompressor: for running executable
  - Compressor: for compressing newly infected binaries
  - Lots of free space (if needed)
- Problem (for virus writer): virus easy to recognize
Using encryption to hide a virus

- Hide virus by encrypting it
  - Vary the key in each file
  - Virus “code” varies in each infected file
  - Problem: lots of common code still in the clear
    - Compress / decompress
    - Encrypt / decrypt

- Even better: leave only decryptor and key in the clear
  - Less constant per virus
  - Use polymorphic code (more in a bit) to hide even this

![Diagram showing the structure of a compressed executable program with layers for virus, decryptor, key, and header.]

**Diagram:**
- **Unused**
- **Key**
- **Encryptor**
- **Decryptor**
- **Compressor**
- **Decompressor**
- **Header**
- **Compressed executable program**
- **Virus**
Polymorphic viruses

- All of these code sequences do the same thing
- All of them are very different in machine code
- Use “snippets” combined in random ways to hide code

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A,R1</td>
<td>MOV A,R1</td>
<td>MOV A,R1</td>
<td>MOV A,R1</td>
<td>MOV A,R1</td>
</tr>
<tr>
<td>ADD B,R1</td>
<td>NOP</td>
<td>ADD #0,R1</td>
<td>OR R1,R1</td>
<td>TST R1</td>
</tr>
<tr>
<td>ADD C,R1</td>
<td>ADD B,R1</td>
<td>ADD B,R1</td>
<td>ADD B,R1</td>
<td>ADD C,R1</td>
</tr>
<tr>
<td>SUB #4,R1</td>
<td>NOP</td>
<td>OR R1,R1</td>
<td>MOV R1,R5</td>
<td>MOV R1,R5</td>
</tr>
<tr>
<td>MOV R1,X</td>
<td>ADD C,R1</td>
<td>ADD C,R1</td>
<td>ADD C,R1</td>
<td>ADD B,R1</td>
</tr>
<tr>
<td>NOP</td>
<td>SHL #0,R1</td>
<td>SHL R1,0</td>
<td>CMP R2,R5</td>
<td></td>
</tr>
<tr>
<td>SUB #4,R1</td>
<td>SUB #4,R1</td>
<td>SUB #4,R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>JMP .+1</td>
<td>ADD R5,R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV R1,X</td>
<td>MOV R1,X</td>
<td>MOV R1,X</td>
<td>MOV R5,Y</td>
<td>MOV R5,Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How can viruses be foiled?

- Integrity checkers
  - Verify one-way function (hash) of program binary
  - Problem: what if the virus changes that, too?

- Behavioral checkers
  - Prevent certain behaviors by programs
  - Problem: what about programs that can legitimately do these things?

- Avoid viruses by
  - Having a good (secure) OS
  - Installing only shrink-wrapped software (just hope that the shrink-wrapped software isn’t infected!)
  - Using antivirus software
  - Not opening email attachments

- Recovery from virus attack
  - Hope you made a recent backup!
  - Recover by halting computer, rebooting from safe disk (CD-ROM?), using an antivirus program
Worms vs. viruses

- Viruses require other programs to run
- Worms are self-running (separate process)
- The 1988 Internet Worm
  - Consisted of two programs
    - Bootstrap to upload worm
    - The worm itself
  - Exploited bugs in sendmail and finger
  - Worm first hid its existence
  - Next replicated itself on new machines
  - Brought the Internet (1988 version) to a screeching halt
Mobile code

- **Goal:** run (untrusted) code on my machine
- **Problem:** how can untrusted code be prevented from damaging my resources?
- **One solution:** sandboxing
  - Memory divided into 1 MB sandboxes
  - Accesses may not cross sandbox boundaries
  - Sensitive system calls not in the sandbox
- **Another solution:** interpreted code
  - Run the interpreter rather than the untrusted code
  - Interpreter doesn’t allow unsafe operations
- **Third solution:** signed code
  - Use cryptographic techniques to sign code
  - Check to ensure that mobile code signed by reputable organization
Security in Java

- Java is a type safe language
  - Compiler rejects attempts to misuse variable
- No “real” pointers
  - Can’t simply create a pointer and dereference it as in C
- Checks include …
  - Attempts to forge pointers
  - Violation of access restrictions on private class members
  - Misuse of variables by type
  - Generation of stack over/underflows
  - Illegal conversion of variables to another type
- Applets can have specific operations restricted
  - Example: don’t allow untrusted code access to the whole file system
Protection

- Security is mostly about *mechanism*
  - How to enforce policies
  - Policies largely independent of mechanism

- Protection is about specifying policies
  - How to decide who can access what?

- Specifications must be
  - Correct
  - Efficient
  - Easy to use (or nobody will use them!)
Protection domains

- Three protection domains
  - Each lists objects with permitted operations
- Domains can share objects & permissions
  - Objects can have different permissions in different domains
  - There need be no overlap between object permissions in different domains
- How can this arrangement be specified more formally?
### Protection matrix

<table>
<thead>
<tr>
<th>Domain</th>
<th>File1</th>
<th>File2</th>
<th>File3</th>
<th>File4</th>
<th>File5</th>
<th>Printer1</th>
<th>Mouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read</td>
<td>Read Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read</td>
<td>Read Write</td>
<td>Read Write</td>
<td>Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Write</td>
<td></td>
<td>Write</td>
<td>Write</td>
<td>Read</td>
<td></td>
</tr>
</tbody>
</table>

- Each domain has a row in the matrix
- Each object has a column in the matrix
- Entry for `<object,column>` has the permissions
- Who’s allowed to modify the protection matrix?
  - What changes can they make?
- How is this implemented efficiently?
Domains as objects in the protection matrix

- Specify permitted operations on domains in the matrix
  - Domains may (or may not) be able to modify themselves
  - Domains can modify other domains
  - Some domain transfers permitted, others not

- Doing this allows flexibility in specifying domain permissions
  - Retains ability to restrict modification of domain policies

<table>
<thead>
<tr>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Modify</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Modify</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Enter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>File1</th>
<th>File2</th>
<th>File3</th>
<th>File4</th>
<th>File5</th>
<th>Printer1</th>
<th>Mouse</th>
<th>Dom1</th>
<th>Dom2</th>
<th>Dom3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read</td>
<td>Read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Read</td>
<td>Write</td>
<td>Read</td>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Write</td>
<td></td>
<td>Write</td>
<td>Read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Representing the protection matrix

- Need to find an efficient representation of the protection matrix (also called the *access matrix*)
- Most entries in the matrix are empty!
- Compress the matrix by:
  - Associating permissions with each object: *access control list*
  - Associating permissions with each domain: *capabilities*
- How is this done, and what are the tradeoffs?
Access control lists

- Each object has a list attached to it
- List has
  - Protection domain
    - User name
    - Group of users
    - Other
  - Access rights
    - Read
    - Write
    - Execute (?)
    - Others?
- No entry for domain => no rights for that domain
- Operating system checks permissions when access is needed

File 1
- elm: <R,W>
- znm: <R>
- root: <R,W,X>

File 2
- elm: <R,X>
- uber: <R,W>
- root: <R,W>
- all: <R>
**Access control lists in the real world**

- **Unix file system**
  - Access list for each file has exactly three domains on it
    - User (owner)
    - Group
    - Others
  - Rights include read, write, execute: interpreted differently for directories and files

- **AFS**
  - Access lists only apply to directories: files inherit rights from the directory they’re in
  - Access list may have many entries on it with possible rights:
    - read, write, lock (for files in the directory)
    - lookup, insert, delete (for the directories themselves),
    - administer (ability to add or remove rights from the ACL)
Capabilities

- Each process has a capability list
- List has one entry per object the process can access
  - Object name
  - Object permissions
- Objects not listed are not accessible
- How are these secured?
  - Kept in kernel
  - Cryptographically secured
Cryptographically protected capability

- Rights include generic rights (read, write, execute) and
  - Copy capability
  - Copy object
  - Remove capability
  - Destroy object
- Server has a secret (*Check*) and uses it to verify capabilities presented to it
  - Alternatively, use public-key signature techniques

$$F(\text{Objects, Rights, Check})$$
Protecting the access matrix: summary

- **OS** must ensure that the access matrix isn’t modified (or even accessed) in an unauthorized way.

- **Access control lists**
  - Reading or modifying the ACL is a system call.
  - OS makes sure the desired operation is allowed.

- **Capability lists**
  - Can be handled the same way as ACLs: reading and modification done by OS.
  - Can be handed to processes and verified cryptographically later on.
  - May be better for widely distributed systems where capabilities can’t be centrally checked.
All system calls go through the reference monitor for security checking.
Formal models of secure systems

- Limited set of primitive operations on access matrix
  - Create/delete object
  - Create/delete domain
  - Insert/remove right

- Primitives can be combined into *protection commands*
  - May not be combined arbitrarily!

- OS can enforce policies, but can’t decide what policies are appropriate

- Question: is it possible to go from an “authorized” matrix to an “unauthorized” one?
  - In general, undecidable
  - May be provable for limited cases
Bell-La Padula multilevel security model

- Processes, objects have security level
- Simple security property
  - Process at level $k$ can only read objects at levels $k$ or lower
- * property
  - Process at level $k$ can only write objects at levels $k$ or higher
- These prevent information from leaking from higher levels to lower levels

A writes 4
Biba multilevel integrity model

- Principles to guarantee integrity of data
- Simple integrity principle
  - A process can write only objects at its security level or lower
  - No way to plant fake information at a higher level
- The integrity * property
  - A process can read only objects at its security level or higher
  - Prevent someone from getting information from above and planting it at their level
- Biba is in direct conflict with Bell-La Padula
  - Difficult to implement both at the same time!
# Orange Book security requirements

<table>
<thead>
<tr>
<th>Criterion</th>
<th>D</th>
<th>C1</th>
<th>C2</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Security policy</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discretionary access control</td>
<td>X</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>X</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Object reuse</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Labels</td>
<td>X</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Label integrity</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Exportation of labeled information</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Labeling human readable output</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Mandatory access control</td>
<td>X</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Subject sensitivity labels</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Device labels</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td><strong>Accountability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identification and authentication</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Audit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>Trusted path</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
</tbody>
</table>
Orange Book security requirements, cont’d

<table>
<thead>
<tr>
<th>Assurance</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>System architecture</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>System integrity</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Security testing</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Design specification and verification</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Covert channel analysis</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Trusted facility management</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Configuration management</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Trusted recovery</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Trusted distribution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Security features user’s guide</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trusted facility manual</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Test documentation</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design documentation</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Chapter 9: Security
Covert channels

- Circumvent security model by using more subtle ways of passing information
- Can’t directly send data against system’s wishes
- Send data using “side effects”
  - Allocating resources
  - Using the CPU
  - Locking a file
  - Making small changes in legal data exchange
- *Very* difficult to plug leaks in covert channels!
Covert channel using file locking

- Exchange information using file locking
- Assume \( n+1 \) files accessible to both A and B
- A sends information by
  - Locking files 0..\( n-1 \) according to an \( n \)-bit quantity to be conveyed to B
  - Locking file \( n \) to indicate that information is available
- B gets information by
  - Reading the lock state of files 0..\( n+1 \)
  - Unlocking file \( n \) to show that the information was received
- May not even need access to the files (on some systems) to detect lock status!
Steganography

- Hide information in other data
- Picture on right has text of 5 Shakespeare plays
  - Encrypted, inserted into low order bits of color values