Operating Systems
COMP2006

Protection and Security
Lecture 10

Protection and Security

References:

Silberschatz, Galvin, and Gagne, *Operating System Concepts*, Chapters 14, 15

Topics:

- Goals and Domain of protection.
- Access matrix and its implementation.
- Revocation of Access Rights.
- Security Problems.
- Authentication.
- Program and System Threats.
- Security Defences.

Protection

- A computer system is a collection of processes and objects.
  - Each object can be accessed through a well-defined set of operations, e.g., read, write, etc
  - Each object should be operated only by processes with proper OS authorization
  - The various concurrent users/processes must be protected from one another
- Protection refers to a mechanism for controlling access of programs, processes, or users to the resources defined by a computer system.
  - The resources include: files, memory segments, and CPU
- **Protection problem:** ensure each object is accessed correctly only by processes that are allowed to do so.
- Protection must provide:
  - Means of specifying controls to be imposed.
  - Means of enforcement.

Protection (cont.)

- Reasons for providing protection:
  - To prevent intentional violation of access rights by bad users.
  - To prevent unintentional access by incompetence users
  - To ensure that each program in a system uses resources consistent with the stated policies for the uses of the resources.
  - To improve reliability by detecting latent errors at the interfaces between component subsystems.
- The role of protection in a system is to provide a *mechanism* for the enforcement of *policies* for the resource use.

Protection (cont.)

- Protection *policies* can be established in a variety of ways:
  - Fixed in the design of the system.
  - Formulated by the management of the system.
  - Defined by the individual user.
- *Policies* change over time and during application execution
  - Protection system must have the flexibility to enforce variety of policies that can be declared to it.
- For flexibility, we need to separate *policy* from *mechanism*.
  - *Mechanism*: How it will be done.
  - *Policy* Decide what will be done.

Domain of Protection

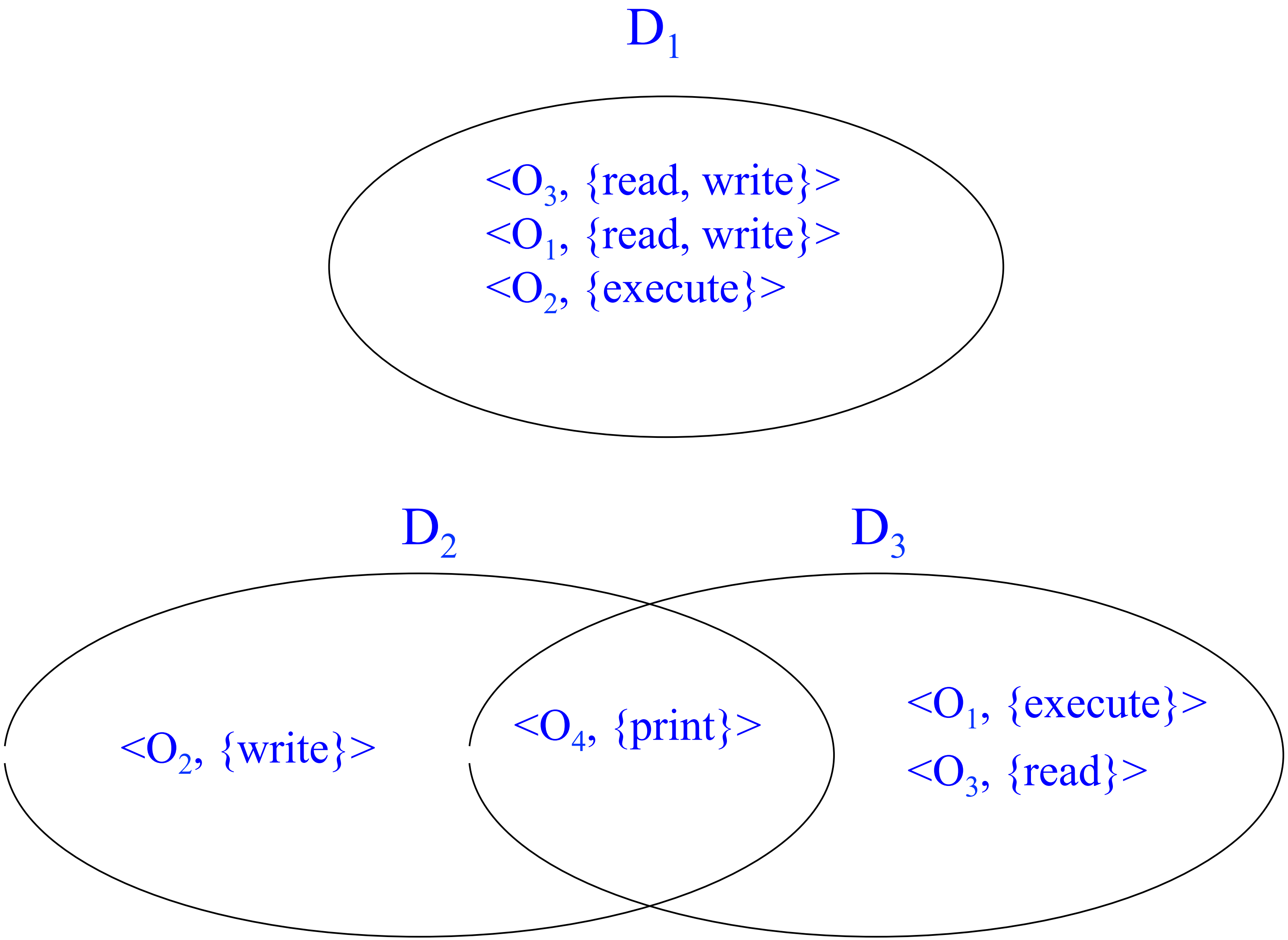
- Each object is essentially an *abstract data type*,
  - has a unique name and can be accessed using a set of operations.
  - can be a hardware (e.g., printer, CPU) or a software (e.g., semaphore, files)
- Operations that are possible depend on the object.
  - CPU: *execute*; Memory: *read/write*; Card reader: *read*;

    Tape drive: *read*, *write*, *rewind*; Data files: *create*, *open*, *read*, *write*, *close*, *delete*; Program files: *read*, *write*, *execute*, *delete*.
- *Principle of least privilege*:
  - A process / user / system should be given just enough privileges to perform its tasks.
- *Need to know principle*:
  - A process should be able to access only resources required to complete its task.
- Both principles are to limit the damage a faulty process can cause in the system.

Domain Structure

- Protection *domain*: specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be used on each object.
- A domain is a collection of access rights.
  - Access rights = <*object-name*, *right-set*>: the ability to execute an operation on an object.
    - right-set is a subset of all valid operations that can be performed on the object
  - Domains may share access rights.

A system with three protection domains.



Domain Structure (cont.)

- Association of a process and a domain can be:
  - *Static*: the set of resources available to a process is fixed throughout the process's life time.
  - *Dynamic*: may change.

Example:

1st phase, a process needs *read* access right to an object.

2nd phase, the process needs *write* access right to the object.

- Using the static protection, the process needs R / W access right
  - This provides too many rights at each phase
  - *Need to know principle* is violated.
- We must allow rights to change → Keep minimum access rights.

Domain Structure (cont.)

- Domain can be realized in a variety of ways: each user, each process, each procedure.
  - A *user* may be a domain: access to objects depend on the user identity (different for each user)
    - domain changes when one user logs off and another user logs on.
  - A *process* may be a domain: access to objects is determined by the identity of the process.
    - domain is different for each process. Switch domain when one process sends a message and then waits for a response.
  - A *procedure* may be a domain: access to objects (local variables) is different for each procedure.
    - Switch domain when a different procedure is called.

Domain Structure (cont.)

Example: Two domains - *monitor* and *user* modes.

- A process in the monitor domain can:
  - Execute privileged instructions.
  - Access all memory locations.
- A process in the user domain:
  - Has a restricted instruction set.
  - Can access its own memory only.
- These two modes protect OS (in monitor domain) from user processes (in user domain).
- In multiprogramming, two protection domains are insufficient
  - Each user also wants to be protected from the other.

Access matrix

- An abstract model that views protection as a matrix (called *access matrix*)
  - Each row is a domain.
  - Each column is an object.
  - Each entry is a set of access rights:

    access (i, j): the set of operations that a process in Domain$_i$ can invoke on Object$_j$
- Access matrix is the *mechanism* for protection
  - Entries of the matrix are defined by the *policy*.
- OS determines domain for a process, while the user determines entries in the table for a particular object.

Access Matrix - Example

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read, write		read, write	

Domain switching

- Switching from domain D_i to domain D_j is allowed iff the access right *switch* $\in$ access (i, j)

object / domain	F_1	F_2	F_3	printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read, write		read, write		switch			

Use of Access Matrix

- A process in Domain D_i can do *op* on object O_j if $op \in$ access (i, j)
- It can be expanded to the dynamic protection
  - *Copy* right allows the copying of the access rights within the column;

    Copy (denoted by '*') has two variants:
    - *transfer*: a right is copied from access (i, j) to access (k, j); access (i, j) is removed.
    - *limited copy*: when R* is copied from access (i, j) to access (k, j), only the right R (not R*) is created
  - *owner* of O_i allows addition and removal of rights in O_i
    - *copy* and *owner* rights allow a process to change the entries in a column → they limit propagation of access rights.
    - However, they do not guarantee information in an object to go outside its environment → the *confinement problem*; a non-solvable problem
  - *control* right: only to *domain* objects or row

    if access (i, j) includes the *control* right, then a process executing in Domain D_i can modify any access right from Domain D_j

Example: copy

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	**read**	

Example: *owner*

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write*
D_3	execute		

object / domain	F_1	F_2	F_3
D_1	owner execute		
D_2		read* owner write*	read* owner write*
D_3		write	write

Example *control*

object / domain	F_1	F_2	F_3	printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch **control**
D_3		read	execute					
D_4	read write		read write		switch			

object / domain	F_1	F_2	F_3	printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch **control**
D_3		read	execute					
D_4	write		write		switch			

Access Matrix Design

- The design of access matrix separates the mechanism from the policy
  - Mechanism: OS provides access-matrix + rules
    - It ensures that the matrix can only be manipulated by authorized entities and that rules are strictly enforced
  - Policy: User decides on policy
    - Who can access what object and in what mode
- Each column = access list for one object.
  - Defines who can perform what operation on the object, e.g.,

    Domain 1 = read, write

    Domain 2 = read

    Domain 3 = read
- Each row = capability list for one domain.
  - For each domain, what operations are allowed on what objects, e.g.,

    Object 1 – read

    Object 4 – read, write, execute

    Object 5 – read, write, delete, copy

Implementation (cont.)

Global table

- The simplest way: it consists of a set of ordered triples
  - <Domain, Object, Rights>
  - For an operation M (e.g., read) executed in domain D_i on object O_j,
    - The table is searched for a triple $<D_i, O_j, R_k>$
    - If the triple is found with $M \in R_k$, the operation is allowed, else error.
- Drawbacks:
  - Large table
    - It cannot be in memory, and thus requires disk I/O (virtual memory techniques)
    - In general the matrix is sparse
  - It doesn't take advantage of grouping of objects or domains
    - if every process can read O_j, it must have one entry of the object in each domain

Implementation (cont.)

Access lists for objects

- One list per object / column
  - An empty entry is removed
- Each object O_j has a list of <Domain, Rights-Set>
  - The list represents all domains with access rights for the O_j
  - Extension: add a default set of rights for O_j in addition to the list
- For a process in D_i accessing O_j for operation M:
  - Search for $\langle D_i, R_k \rangle$ in O_j list.
  - If it is found with $M \in R_k$ the operation is okay
  - else if M is in the default set, the operation is still OK,
  - else error.
- Fast as many operations will not need to search the list.

Implementation (cont.)

Capability lists for domains

- A *capability list* for a domain is a list of objects together with the operations allowed on those objects
  - List of <O_j, R_k> for each domain D_i
  - An object is represented by its physical name or address called a *capability*
- To execute operation M on O_j
  - The process executes M on the capability (pointer) for object O_j as a parameter
  - Access is allowed if the process possesses the capability for O_j
- Capability list is a protected object
  - It is maintained by the OS, and is accessed by the process indirectly
  - It is inaccessible to the user to prevent modification
    - If a capability list is secure, the object it protects is also secure

Implementation (cont.)

Capability lists for domains (cont.)

Capabilities are distinguished from other data by:

1) A **tag** associated with each object
   - Determines its type (Capability or Accessible Data).
   - A tag must not be directly accessible by the user.
     - May need hardware or firmware support.
   - Needs one bit (Capability or Accessible Data), but usually have other bits - to know their types: integer? float? instruction?

2) Split address space into two parts:
   - Part A: Normal address space (instructions, data, etc.)
   - Part B: Capability list (only accessible from operating system)
   - Can use segmented memory space

Implementation (cont.)

Lock-key mechanism

- Compromise between access list and capability list
  - Each object has a list of unique bit patterns (locks)
  - Each domain has a list of unique bit patterns (keys)
- A process in domain D_i can access O_j if a key in D_i matches one of the locks in O_j
- The list of keys for each domain must be maintained by the OS
  - Users are not allowed to modify the keys and locks directly

Comparisons

Access lists

- Most appropriate for users
  - When a user creates an object, the user can define which domains that can access it and their allowed operations
- Revocation easy - just search list
- Determining access rights for each domain is difficult
- Each access to the object requires search of the list
  - time consuming for long list.

Capability list

- Not appropriate to users
  - but is useful to localize information for a particular process
- Process must present a capability to access an object
  - Protection system has to verify that capability is valid
- Revocation is inefficient
  - capabilities distributed through the system

Comparisons (cont.)

Lock-keys

- Keys passed freely between domains
- Access rights are revoked by changing some locks of an object
- Used for file access (Unix)
  - Each file has an access list
  - When a process opens a file, the directory structure is searched, and its access permission is checked
    - Information is stored in a new entry in the file table for this process
    - If a file is opened for reading, the file table entry stores a capability for only read access
    - when the file is closed, the file-table entry is deleted
  - The file-table is maintained by OS, and thus the files that can be accessed by users (capability) are those that have been opened → protection ensured.

Revocation of Access Rights

- In a dynamic protection system, sometimes it is necessary to remove access rights of shared objects
- Various questions about revocations:
  - Immediate *vs* delayed? When will it take effect?
  - Selective *vs* general? Remove for <u>all</u> users or just some?
  - Partial *vs* total? Remove <u>all</u> rights or just some?
  - Temporary *vs* permanent? Can access be re-invoked later or never?

Revocation with an Access List

- Simply search for rights to be revoked
  - Delete access rights from access list
- Revocation is immediate, and can be:
  - General or selective.
  - Total or partial.
  - Permanent or temporary.

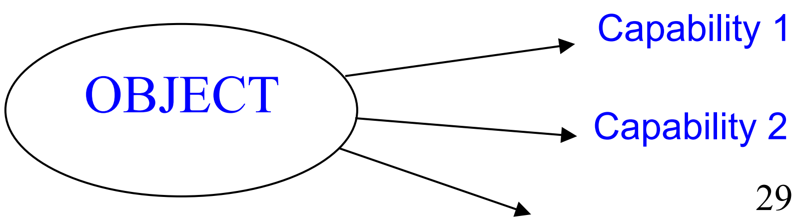
Revocation of Access Rights (cont.)

Revocation with a Capability List

- Scheme requires to locate capability in the system before capability can be revoked
  - More difficult because capabilities are distributed all over the system

Several schemes:

- Reacquisition
  - Delete capabilities for a domain periodically
  - A process can reacquire capability that has been deleted if access is not revoked.
- Back-pointers (in Multics system)
  - Use a list of pointers for each object to all capabilities for the object
  - Follow pointers to change capabilities
  - Costly (lists of pointers)



Revocation of Access Rights (cont.)

Several schemes (cont.):

- Indirection
  - Each capability points to a unique entry in a global table, which in turn points to the object
    - Search the table for the desired entry, and delete it
    - An access fails when the capability points to an illegal table entry
    - The object for a capability and the table entry must match.
  - It does not allow selective revocation.
- Keys
  - A key is a unique bit pattern associated with each capability
  - This key is created when capability is created; not accessible by a process
  - Need master-key associated with each object
  - Capability is valid <u>if</u> its <u>key</u> matches the <u>master key</u>
  - To revoke, change the master key
  - Selective revocation requires number of keys (one per domain)

The security problem

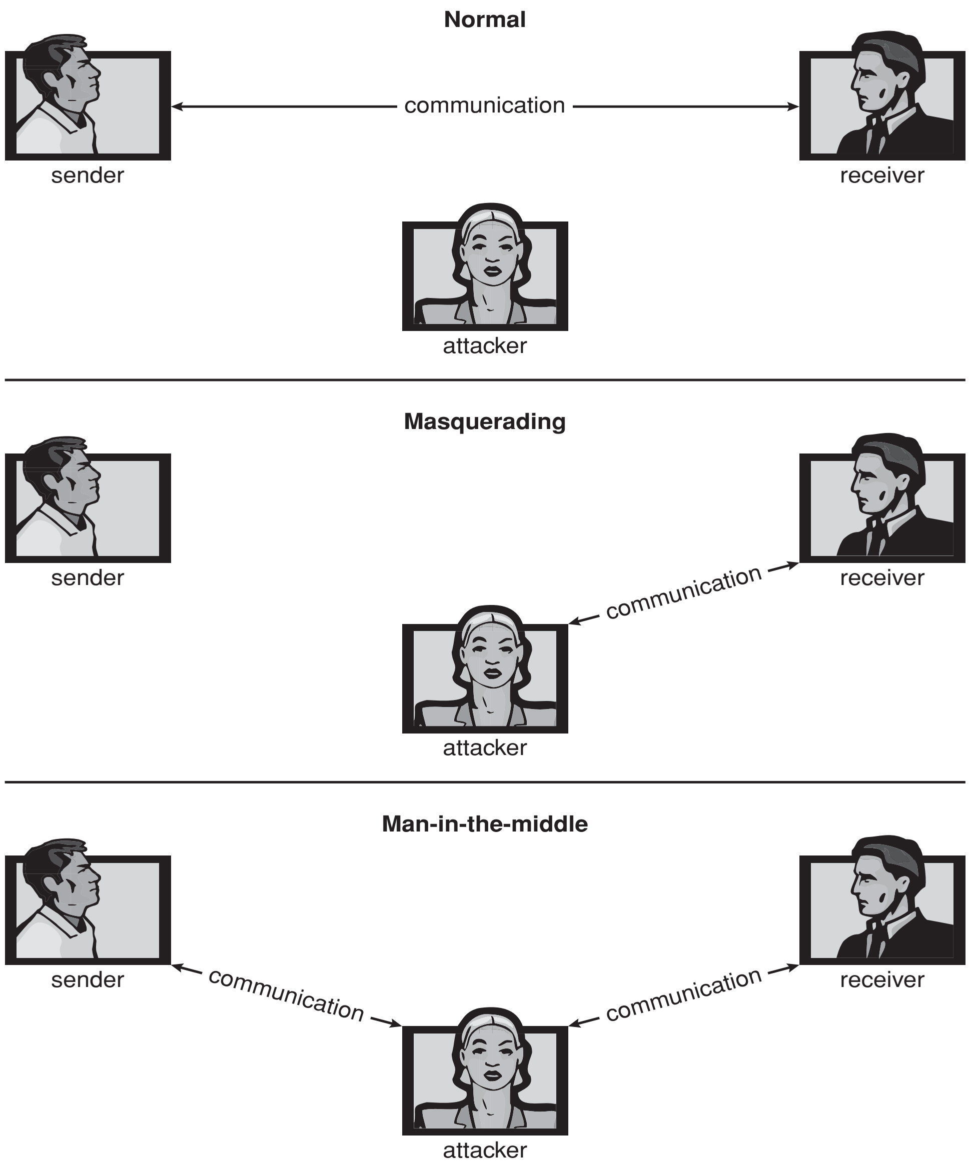
- Security considers **external environment** of the system, and protects it from:
  - Unauthorized access
  - Malicious modification or destruction
  - Accidental introduction of inconsistency
- Easier to protect against accidental than malicious misuse
  - Protection mechanism can mostly address accidental security violation
- Some security violations
  - **CIA** – **C**onfidentiality, **I**ntegrity, **A**vailability
    - **C**onfidentiality – unauthorized reading of data (information theft)
    - **I**ntegrity – unauthorized modification of data
    - **A**vailability – unauthorized destruction of data
  - Theft of service – unauthorized use of resources
  - Denial of service – prevent authorized use of system

The security problem (cont.)

- Some methods to breach security:
  - **Masquerading** – pretend to be someone else
    - Breach authentication
    - Escalate privileges
  - **Replay attack** – repeat valid data transmission for malicious purpose
    - Can be repeating a request or modification of the request
  - **Man in the middle** – located between data communication, masquerade as a legitimate sender to a legitimate receiver, and the receiver to the sender
  - **Session hijacking** – intercept a communication session

The security problem (cont.)

Figure 15.1 (textbook)



The security problem (cont.)

- Four levels of security measures:
  - **Physical.** Physically secure the location of the computer system against entry from intruders.
  - **Human.** Users must be screened so that the chance for authorizing a user who then gives access to an intruder is reduced
    - Also be aware of possible phishing; e.g., by faked email or phone
    - Other possibility: the attacker gather information to get password
  - **OS.** It must be protected against denial of service attack, stack overflow, etc.
  - **Network.** It must be protected against data interception

- Security within the OS is implemented at several levels, ranging from passwords to the isolation of concurrent processes running within the system

User Authentication

- A major security problem for OS is the authentication problem
  - Needs to identify programs and processes that are executing
    - in turn needs to identify each user of the system
- The user usually identifies himself
  - how to determine if a user identity is authentic?
- Authentication is based on one of these:
  - User possession → what the user posses (a key or a card)
  - User knowledge → user identifier and password
  - User attribute → finger print, retina pattern, or signature
- User identity is most often established through passwords

Passwords

- To access a computer, each user is asked to identify user ID and a password
  - OK if user-supplied password matches the password stored in the system.
- Difficult to keep passwords secret; passwords can be:
  - Guessed,
  - Accidentally exposed, or
  - Illegally transferred
- How to guess passwords?
  - Know the information about the user
    - People use obvious information for passwords (spouse name, street addresses, etc.).
  - Use brute force method
    - Try all combinations of letters, numbers, punctuations to find password

Passwords (cont.)

- Accidental exposed:
  - Using visual or electronic / network monitoring
  - From writing down password (for hard-to-remember password).
- Passwords can be generated by system or selected by users.
  - System generated → hard to remember that may force user to write it down
  - User-selected password → easy to guess.
- Administrators occasionally check user passwords and notify the user if the password is too short or too easy to guess.
  - Change passwords frequently.
  - Use *non-guessable* passwords: use combinations that include upper / lower cases, and punctuation, numbers.
  - Log all invalid access attempts.

Securing Passwords

- Unix uses secure hashing
  - It uses a hash function $f(x)$ to generate a hash for each password x
    - The function is fast to compute hash $f(x)$, but it is almost impossible to find x from $f(x)$
    - Store all encoded passwords – some system protects access to this file (superuser mode)
  - When a user A presents password y, compute hash $f(y)$, and check if it matches A's encoded password stored in the system
  - Encoded password file needs not be kept secret
- Disadvantage:
  - System no longer has control over the passwords
  - Attacker can run fast hash function on a set of guessed passwords and compare each of them against the encoded password file
    - Guessed passwords can be words in dictionary – some users use words in dictionary as passwords
    - Unix systems use well-known hashing algorithm – an attacker can keep a copy of passwords that have been cracked
    - Some systems consider only the first eight characters of passwords

Securing Passwords (cont.)

- Include a ‘salt’ – recorded random number - in the hashing algorithm.
  - Two plaintext passwords that are the same will have two different hash values
  - Effective against the dictionary attack
  - Latest Unix system requires superuser access to hashed password file
- Use passphrase for hard-to-crack but easy to remember password
  - Also, use upper and lower cases, punctuation, and numbers in password
    - E.g., MmMni.EG! → My mother’s Maiden name is Emily Gould

One-time Passwords

- In one-time password system, the password is different in each instance
- Use a set of paired passwords
  - Protect against password sniffing and shoulder surfing
- System randomly selects and presents one part of the password-pair as a challenge to the user
  - User must respond with the other part of the password
- Use algorithm (such as integer function) as a password
  - System and user share a symmetric password *pw* and function $f(pw, ch)$
  - System presents a random challenge *ch*, and user has to provide as password the result of the function $f(pw, ch)$
  - Only system and user know *pw*, and each result of $f(pw, ch)$ is different

One-time Passwords (cont.)

* Examples of one-time password system:
  - SecurID: use a hardware calculator (in the shape of credit card with a keypad and display). For seed, use current time, and for shared secret use personal identification number (PIN).
  - *Code book* or *one time pad*. A list of single-use passwords. Each password in the list is used in order once, and then is crossed out or erased.

Program Threats

Trojan Horse: Code segment that misuses its environment

- Exploit mechanisms to allow programs written by some user to be executed by other users, e.g.,
  - A text-editor may contain code to search for certain keywords in a file
    - If found, the entire file can be copied to an area accessible by the creator of the text-editor
  - A program that emulates a login program to steal password

Trap Door: A hole in the software purposely left by the software's designer that can be used only by the designer.

- It may circumvent normal security procedures
- It could be included in a compiler
  - The compiler can produce a trap door as part of any program's object code
- It is hard to detect
  - to detect, we have to analyse all the source code for all components of the system.

Logic bomb: Security hole that is created only when a predefined condition is met

- E.g., delete all files when the system programmer is no longer employed!

Program Threats (cont.)

Stack and Buffer Overflow

- The most common way for an attacker outside of the system.
- Some authorized user may use this for privilege escalation to gain privileges beyond those allowed for that user.
- Exploits a bug in a program
  - Programmer neglected to code for checking the bounds on an input field
    - Attacker sends more than the program was expecting.
  - Attacker can:
    - Overflow an input field or input buffer until it writes into system stack.
    - Overwrite the current return address on the stack with the exploit code loaded.
    - Write a code for the next space in the stack which the attacker wants to execute

Solution:

- Feature in CPU disallows execution of code in a stack section of memory.

Program Threats (cont.)

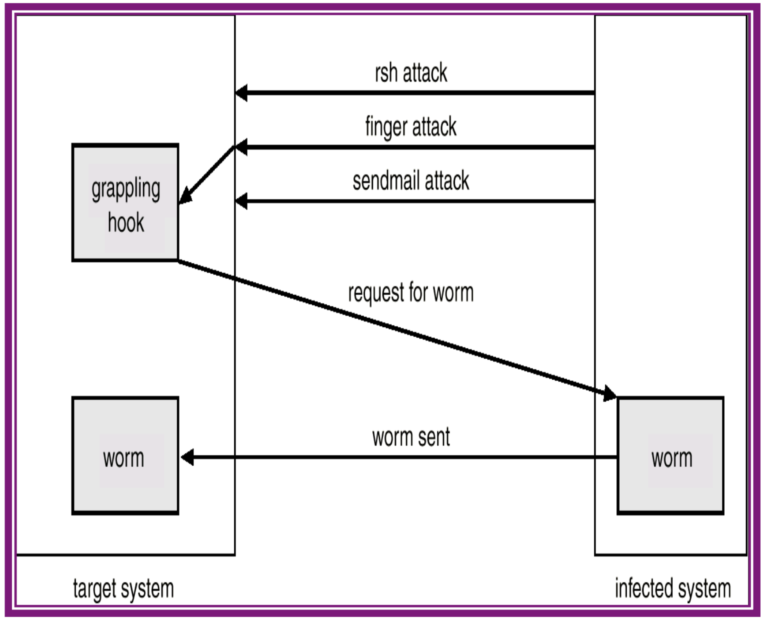
- Virus: a fragment of code that is embedded in a legitimate program.
  - Viruses are specific to computer architecture, OS, and application
    - It mainly affects personal computers
  - Viruses are designed to infect other programs
    - Self replicating
    - Spread by email, download infected program, share infected disk, document that contains macro
    - To prevent virus, do *safe* computing

Main virus category (cont.)

- Main virus category:
  - File – append virus to a file; also called parasitic virus
  - Boot – infect the boot sector of the system; also known as memory virus
  - Macro – written in high level language, e.g., Visual Basic, in contrast to in Assembly code
  - Source code – modify source code to include the virus and spread it
  - Polymorphic – can change itself each time it is installed to avoid detection
  - Encrypted – the virus is in encrypted form together with its decryption function
  - Stealth – can modify parts of the system that can be used to detect it
  - Tunnelling – install itself in interrupt handler or device drivers to avoid detection
  - Multipartite – infect multiple parts, e.g., boot sector, memory and files
  - Armored – the virus is coded in such a way to make it harder to be detected, e.g., it can be in non-viewable files or in compressed form.

System Threats

- System threat attacks system services and network connections
  - In contrast to program threat that compromises system's protection mechanisms
  - It can help starting program attack or misusing system and user resources
  - To reduce the attacks, by default, enable only necessary system services at installation
  - Example attacks: worms, port scanning, denial of service



Worms: A worm uses the spawn mechanism; it is a standalone program

- Most OS provides ways for processes to spawn other processes

Example: Internet worm by Robert Tappan Morris in 1988.

- It exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs
- The worm consists of Grappling hook (bootstrap) and main worm program
- Grappling hook program uploaded main worm program

System Threats (cont.)

Port scanning:

- A means for an attacker to find vulnerabilities in the system
  - Automated to create TCP/IP connections to specific ports
    - If connection to a port is successful, the attacker can later find out if the application running on the port has bug → an attack can be started there, e.g., install trojan horses, etc
  - Port scanning can be detected
    - Thus, an attacker uses compromised systems / machines to perform port scanning
      - E.g., zombie systems or bots

System Threats (cont.)

Denial of Service: Overload the targeted computer preventing it from doing any useful work

- The attack is to disrupt system resources → cannot use the facility
  - It is not to steal information
- Generally network based
  - E.g., SYN flooding attacks flaw in TCP connection establishment.
- Can use huge amount of system resources: e.g., CPU, bandwidth
  - Attacker can use bots to perform distributed DoS attack
- In general, it is impossible to prevent DoS attack

Security Defences

Security Policy:

- Include a statement of what is being secured
  - Different between organizations
- The policy is used by users as a guide
  - Users should know what activities are allowed, not allowed, and required
  - E.g., users should not share passwords, port scannings must be performed every 6 months, every new application programs must be reviewed before deployment, etc.
- The policy must be reviewed and updated regularly

Security Defences (cont.)

Vulnerability assessment: to determine if a security has been implemented correctly

- Cover many things, including risk assessment and penetration test or vulnerability scans
- Scan the system periodically for security holes; done when the computer is relatively unused.
- Check for:
  - Short or easy-to-guess passwords; Unauthorized privileged programs; Unauthorized programs in system directories; Unexpected long-running processes; Improper directory protections; Improper protections on system data files; Dangerous entries in the program search path (Trojan horse); Changes to system programs; monitor checksum values, etc.

Security Defences (cont.)

Intrusion detection: Detect attempts to intrude into computer system

- What activities are considered intrusions? Two approaches:
  - Signature-based: check for known behavior or patterns (signatures) that indicate attack
    - Used to detect known/recognizable attacks
    - E.g., check specific strings or network packets for known viruses
  - Anomaly detection: check for anomalous behaviors in the system
    - Can be used to find zero-day-attacks – previously unknown intrusion methods
  - The approaches must produce in-excessive:
    - False positives (false alarms)
    - False negatives (missed intrusions)