

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

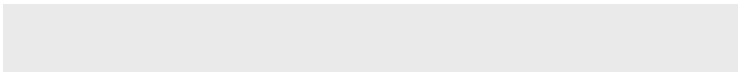


THE UNIVERSITY OF  
WESTERN AUSTRALIA

# Computer Operating Systems 214



## 7. SECURITY



- 
- 
- 
- 
- 
- 
- 
-

# Copyrights on Figures

- Figures presented in these notes with the following captions are extracted from the textbook and copyrighted by the publisher as indicated
  - (OS3e) W. Stallings, “*Operating Systems: Internals and Design Principles*”, 3rd Edition, Prentice-Hall, 1997
  - (osdi2) A.S. Tanenbaum & A.S Woodhull, “*Operating Systems: Design and Implementation*”, 2nd Edition, Prentice-Hall, 1997
- **DISCLAIMER**
  - Whilst every attempt is made to maintain the accuracy and correctness of these notes, the author, Dr. Roberto Togneri, makes no warranty or guarantee or promise express or implied concerning the content.

# Security Issues

- User Authentication
  - Human *John* has an account *john*. How does system ensure that resources used by account *john* are really from user *John*?
    - Files that reside in account *john* and processes that belong to *john*
  - OS needs to authenticate that *john* is being used by *John*
- Protection
  - Different users have different access rights and privileges to system resources
  - UNIX *root* user has the highest privileged access to system
    - Can access all files for reading, writing and deletion
    - Can increase process priority, modify memory management operation, etc.
    - Can execute programs like “shutdown” and “reboot”
  - OS needs to assign and control resources with different access rights to users and groups of users and protect resources from unauthorised use
- Security Threats
  - Systems can be attacked and security breached in several ways
    - *Technical*: due to incorrect design of OS kernel and system programs
    - *Human factors*: deceiving users and system managers, malicious intent, etc.
    - *Management*: security measures usually degrade rather than enhance ease of use  
→ so short-cuts are taken which may obviate security

# User Authentication: Concept

- Resources are allocated to users
  - **User ID (UID):** number that is used to identify the owner of resources
    - *file owner UID:* user that initially created file
      - processes create files and the file assumes the owner UID of the process owner UID
    - *process owner UID:* user that spawned the process
      - all processes owned by the same user are spawned indirectly by the initial login process that authenticated the user → child processes inherit the ownership from parents
  - Authentication
    - User indicates the account by typing the account name, and system then associates the UID with that name
      - UID = 111 belongs to account name *mary*
    - The account name is not secret so anybody can access the account simply by knowing the name
      - just type “root” and ‘hey presto’ you control the system!
    - *Solution?* Each account name has a secret password associated with it which only the real human user and OS know
      - User is challenged to type in the secret password. If it matches the password stored for the stated account name then the user has been verified since only the real user would know the password.

# User Authentication: Login sequence

- Typical login sequence
  - “login: ” prompt
    - The account name is entered, and is echoed to the screen as it is typed
  - “Password: ” prompt
    - The secret password that is associated with the stated account name is entered
    - The password is not echoed to the screen or is deliberately masked
  - The system password file is consulted to see if there is a match
    - The account name is used to search the password file and the corresponding password is then checked with the typed password → if they match user has been verified
  - OS assumes a secure channel and subsequent commands are not authenticated
    - User commands input from keyboard or mouse over the same channel (phone line, terminal, etc.) are assumed to belong the initially verified owner
      - In trusted environment this can be extended to network connections
      - In insecure environments any remote or networked channel will require additional **cryptographic** solutions to ensure commands are both secure and authentic
    - The login process which has authenticated the user spawns the user login shell and subsequent user processes

# User Authentication: Encryption

- Private-Key Encryption

- $\langle \text{cipher-text} \rangle = \text{encrypt}(\langle \text{plain-text} \rangle, \langle \text{secret key} \rangle)$ 
  - The *secret key* is used to convert the human readable plain-text message into an unreadable cipher-text
  - The encryption algorithm is public but the key is secret
- $\langle \text{plain-text} \rangle = \text{decrypt}(\langle \text{cipher-text} \rangle, \langle \text{secret key} \rangle)$ 
  - The same secret key is used to convert the unreadable cipher-text message into the readable plain-text
  - The decryption algorithm is public but the key is secret
- The same key is used by the both parties to transmit/store cipher-text
  - Without access to the key the cipher-text is useless since it is impossible to derive the plain-text from the cipher-text without running the decryption algorithm with the correct key
- Suffers from key distribution problem
  - Initial key needs to be transmitted in plain-text so both parties can agree on it
- Example algorithm: Data Encryption Standard (DES), IDEA
  - Same algorithm used for both encryption and decryption
  - Algorithm can be implemented in hardware for *on-the-fly* encryption/decryption

# User Authentication: Encryption

- Public-key Encryption
  - Two keys are used
    - *public key*: this is made available publicly for parties that are to send data
    - *private key*: this is kept by the recipient and need not be distributed
  - $\langle \text{cipher-text} \rangle = \text{encrypt}(\langle \text{plain-text} \rangle, \langle \text{public key} \rangle)$ 
    - Plain-text is encrypted using the public key
    - The public key cannot be used to decrypt the cipher-text
  - $\langle \text{plain-text} \rangle = \text{decrypt}(\langle \text{cipher-text} \rangle, \langle \text{private key} \rangle)$ 
    - Cipher-text is decrypted using the private key
    - The private key cannot be derived from the public key  
→ only intended recipient can decrypt the cipher-text
  - No key distribution problem
  - Example algorithms: RSA
    - Algorithms an order of magnitude slower than DES/IDEA
    - Cannot be implemented efficiently in hardware

# User Authentication: Passwords

- Implementation of password authentication
  - The password is stored in the system in **plaintext** → insecure!
    - Any breach of system will mean access to all account passwords
    - The privileged user (root) has access to all user account password
  - An encrypted version of the password is stored → better!
    - Only user knows the password since the plain-text password cannot be extracted from the stored cipher-text
    - There is no need to perform decryption
      - User types in plain-text password which is then encrypted
      - The cipher-text is compared directly with the stored cipher-text for that account
    - Problem using private key encryption
      - Key is needed to perform encryption
      - Knowledge of key permits decryption
      - If OS stores key then this is no better than storing plain-text password!
    - Require special one-way encryption algorithm
      - No key needed
      - Decryption cannot be done
      - Do any such algorithms exist?

# User Authentication: Passwords

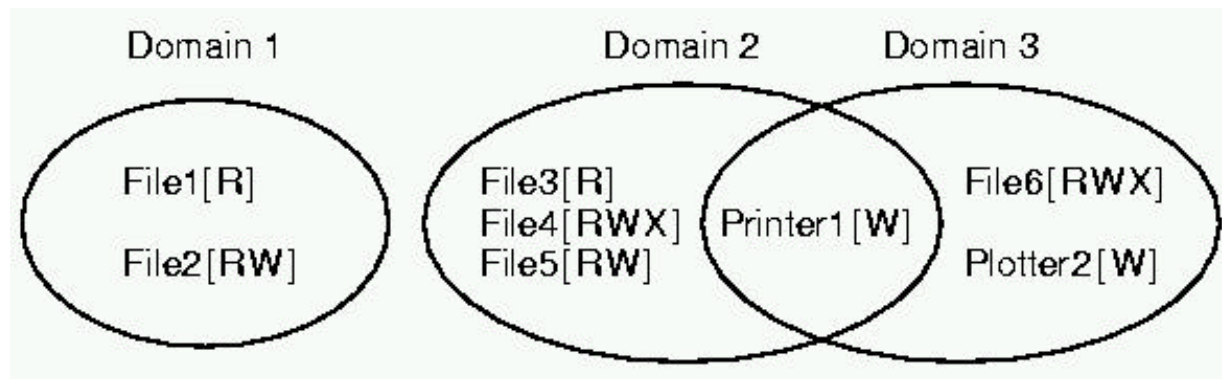
- UNIX uses a modified DES
  - crypt( ) arguments
    - *secret key* = 56-bit string formed from 8x 7-bit ASCII password that user types
    - *salt* = 12-bit random sequence
  - crypt( ) operation
    - 12-bit salt modifies operation of the DES algorithm which is used to encrypt a 64-bit block of 0's using the 56-bit secret key
  - Loading a new password
    - *cipher-text* = crypt(*secret key*, *salt*)
    - Password information that is stored by system in /etc/passwd:  
[*account name* : *salt* / *cipher-text* : UID : GID : ... : login shell]
  - Verifying a password
    - User types in *account name* and password = *secret key*
    - System extracts *salt* and *cipher-text* for the indicated *account name*
    - *cipher-check* = crypt(*secret key* , *salt*)
    - If *cipher-check* = *cipher-text* then user is verified

# User Authentication: Passwords

- Features
  - Users with the same password will have different encrypted passwords due to the random value of the 12-bit *salt*
  - Effectively increases the length of the password, there are  $2^{12} = 4096$  possible encrypted passwords for the same plain-text password
    - Prevents simply storing a list of encrypted passwords generated from known plain-text passwords and comparing with stored encrypted passwords
  - Can still use a program to run the verification process for a targeted account until there is a match → *crack( )* uses all words in the dictionary, account details, etc.
- Choosing and changing Passwords
  - Passwords need to be difficult to discover but easy to remember
    - Easy to remember passwords are usually those that can be easily discovered
      - users name, spouse's name, pet's name, home address, room number, etc
      - any valid English word, proper name, etc.
    - Hard to discover passwords are not listed anywhere else (e.g. dictionary)
      - xyzyy, klmnpp, (hard to remember)
      - klumgot, PorFikt (easy to remember)
  - Password should be periodically changed
    - *Too often*: user forgets new passwords or is tired of always coming up with another password → starts selecting poor passwords
    - *Too few*: sooner or later somebody discovers the password

# Protection

- Policy and Mechanism
  - Policy: Whose data are to be protected from whom
    - Decided by system management
  - Mechanism: How the OS enforces the policy
    - Protection mechanism that the OS supports
  - Some mechanisms cannot support some policies
    - Policy decision made without regard of mechanism limitations → trouble!
- Protection Domain
  - Definitions
    - **Object**: resource that needs to be protected
      - CPU, memory, files, I/O device, etc.
    - **Rights**: operations that can be performed on object
      - file *object* rights: (R)ead, (W)rite, e(X)ecute
    - **Domain**: set of (object,rights) pairs



• Figure 5.22, osdi2: *Three protection domains*

# Protection

- Processes run in a specified protection domain
  - Protection domain is usually defined by the owner of the process
  - Processes can switch domains as required and permitted by OS
- A UID is mapped to a protection domain
  - When users are authenticated the UID is used to determine the protection domain that subsequent processes run in
- Protection Matrix
  - OS can store a matrix listing all the (object,rights) for each domain

Domain	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

• Figure 5.23, osdi2: A protection matrix

- *Problem:* Matrix is large and sparse → too expensive to use
  - One column for each and every object (e.g. file, device, etc.)
  - One row for each and every domain (user, groups of users, etc.)

# Protection: Access Control List (ACL)

- Access Control List (ACL): store protection matrix by column
  - Each object (file) has an ACL attribute associated with it
    - ACL is a set of (domain, rights) pairs
  - Example:
    - File0: (Jan , RWX)
    - File1: (Jan , RW-), (\* , R--), (Peter , RW-)
    - File2: (Peter , RWX), (Jan, R--)
    - File0: domain *Jan* has read/write/execute access, no other domain has access
    - File1: *Jan* and *Peter* have read/write access, others have read access
    - File2: *Peter* has read/write/execute, *Jan* has read access, others have no access
  - Problem?
    - Modification of access rights on a per domain basis is difficult
      - need to modify the ACL for each object which pertains to domain
      - required when user is added or deleted from system
      - Solution?* Define generic or default domain rights in each object: (\* , R--), and allow access right changes to a group of objects (e.g. directory “group” of files)
    - Must protect ACL from tampering by users
      - store as part of OS filesystem management (e.g. i-node, directory entry)

# Protection: Capability Lists (C-lists)

- Capability List (C-list): store protection matrix by row
  - Each domain (UID) has a C-list attribute associated with it
    - C-list is a set of (object, rights) pairs
  - Example (from Figure 5.23):
    - Domain1: (File1, R--), (File2, RW-)
    - Domain2: (File3 , R--), (File4 , RWX), (File5 , RW-), (Printer1 , -W-)
    - Domain3: (File6, RWX), (Printer, -W-), (Plotter2, -W-)
  - Problem?
    - Modification of access rights on a per object basis is difficult
      - need to modify the C-list for each domain which pertains to object
      - required when object is added/removed from system
    - Solution?* Define generic or default object rights in each domain: (\*, R--), and allow access right changes to a group of domains (e.g. workgroups)
    - Must protect C-list from tampering by users
      - store as special part of OS password and workgroup management (C-list for each user or group listed)

# Case Study: UNIX

- Modified form of ACL
  - Domain defined by a (UID,GID) pair
    - UID = integer number that is uniquely assigned to each user
    - GID = integer number that is uniquely assigned to each group of users
    - *Example:*
      - (Bob=100 , student=20) (Jane=101 , student=20) (Albert=120 , staff=23)
  - Each process executes in *domain (UID, GID, GID[1,2...n])* where:
    - GID is the default group
    - GID[1,2,...n] are other groups a user can belong too
  - and:
    - */etc/passwd* specifies the UID and GID
    - */etc/group* specifies other GID[1,2,...n] the user can belong to
  - *Example:*
    - */etc/passwd* entry: Bob:xxxx:100:20:...:/bin/sh  
Jane:xxxx:101:20:...:/bin/sh
    - */etc/group* entry: ips205:\*:23:Bob, Jane  
plsd210:\*:24:Bob  
student:\*:20:
    - Processes from Bob belong to domain (UID = Bob, GID = student, ips205, plsd210)
    - Processes from Jane belong to domain (UID = Jane, GID = student, ips205)

# Case Study: UNIX

- Each object is associated with a fUID, a fGID and 3 different access rights
  - The process (UID, GID) domain is used to derive which access right applies
    - RWX for fUID: Process in domain (UID, ...) acquires these rights to object
    - RWX for fGID: Process in domain (UID≠fUID, GID) acquires these rights to object
    - RWX for Others: Process in domain (UID≠fUID, GID≠fGID) acquires these rights
  - The fUID, fGID and different access rights are stored in the i-node for the file
    - The access rights are stored as a 9 bit *perm* sequence:  
RWX right → 3-bits, 3 types of rights → 3 x 3 = 9 bit
    - The fUID and fGID associated with an object are derived from the (UID,GID) domain of the process that initially created that object (i.e. fUID = UID, fGID = GID)
    - The access rights associated with an object is derived from the default *umask* associated with the process that initially created that object
  - *chmod( )*: changes the 3 access rights associated with file (allowed if UID = fUID)
  - *chown( )*: changes the fUID and fGID associated with file (allowed if UID = root)
- Access rights can be displayed by '*ls -l object*' :
  - *-rwxr-xr-x 1 robert staff ... .. File1*
  - *-rw-r----- 1 jane student ... .. File2*
  - File1: Process with uid = *robert* has *rwx* access, other uids have *r-x* access
  - File2: Process with uid = *jane* has *rw-* access, every other uid ≠ *jane* but gid = *student* has *r--* access, and uid ≠ *jane* and gid ≠ *student* have no access
- Much more efficient than normal ACL, but more restrictive

# Case Study: UNIX (\*)

- Domain switching
  - If user mode process in domain = (UID,GID) makes a system call it switches to kernel mode and also switches domain (usually to UID = 0 = root)
    - Necessary since system calls may need to modify or access resources which would normally not be accessible directly by user processes
    - BUT system first checks domain = (UID,GID) to see if the system call can be allowed in the first place (e.g. *chown( )* system call is only allowed if UID=0)
  - SETUID and SETGID concept
    - Normal operation:
      - If process in domain (UID,GID) does *exec(file, ...)* the ensuing process is also in domain (UID,GID)
    - SETUID operation:
      - Executable file object, *file*, has a special execute bit set which defines an *Effective UID*, EUID = fUID associated with file
      - If process in domain (UID,GID) does *exec(file, ...)* the ensuing process is then in the domain (EUID,GID). Process has switched protection domain from (UID, GID) to (EUID,GID) and since EUID = fUID the process obtains fUID access rights
    - SETGID operation: Analog to SETUID operation, but uses EGID rather EUID
    - Why? Allows users to gain controlled access to protected objects which would otherwise be denied

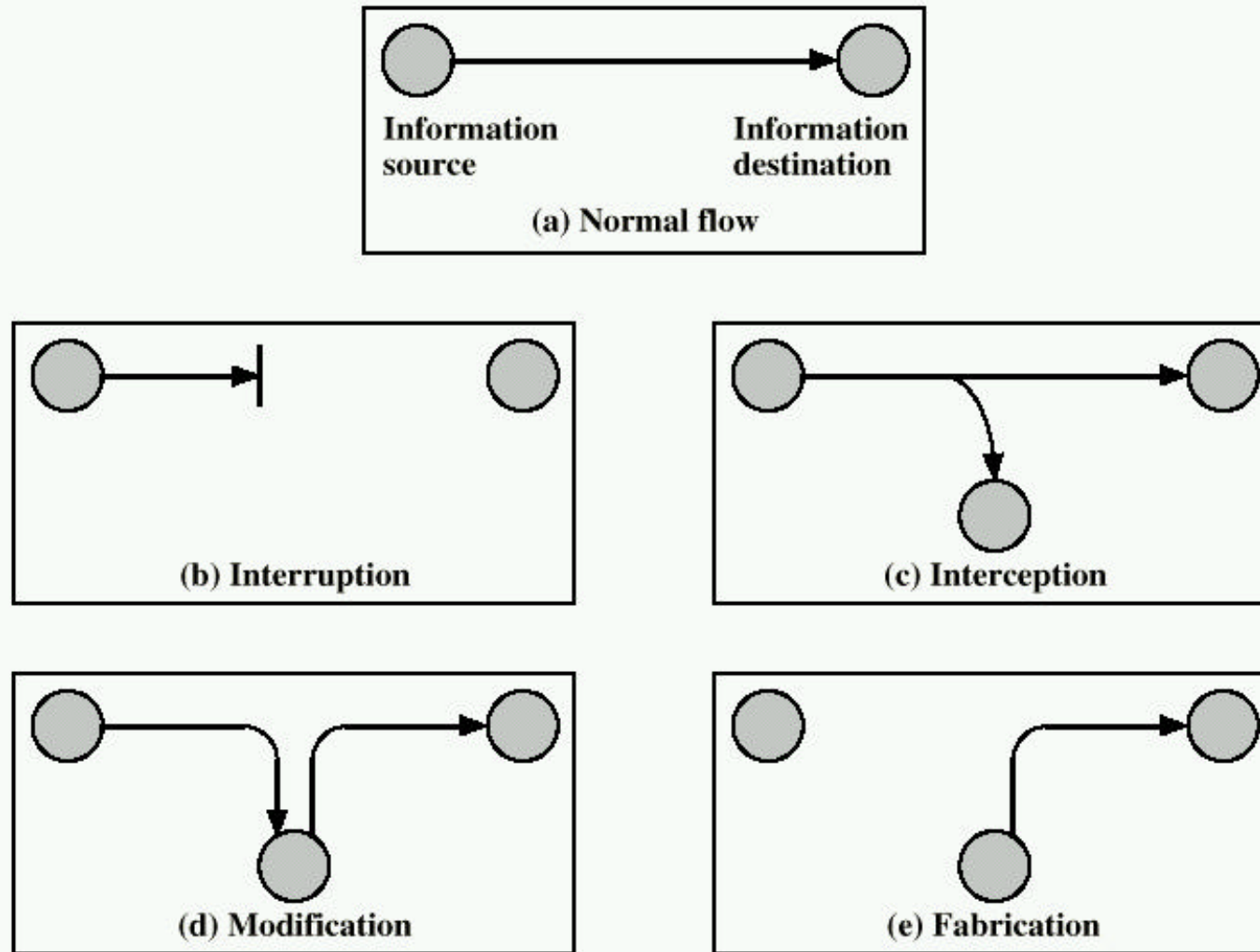
# Case Study: Windows NT

- Versatile and Extensible ACL protection scheme
  - Possible Drawbacks
    - complex maintenance and debugging of complex security setups
    - processing and storage overheads
- Access token
  - Each process object has an access token associated with it
    - Initial user process is assigned the access token that is associated with the username/password used to authenticate user
    - Subsequent processes that are spawned inherit the same access token
  - Fields:
    - *Security ID (SID)*: identifies the user to the system
    - *Group SIDs*: other SID groups this user belongs to
    - *Privileges*: system-sensitive services this user is allowed to call
    - *Default owner SID*: a process spawned by this process acquires this SID rather than the SID of this process
    - *Default ACL*: default access control list that applies to new objects this process creates

# Case Study: Windows NT

- Security Descriptor
  - Each resource (file) object has a security descriptor associated with it
  - Fields:
    - *Flags*: defines the type and contents of descriptor
    - *Owner SID*: a process with the same SID is able to modify the descriptor (e.g. change the access control lists)
    - *System ACL (SACL)*: lists the operations that will generate audit messages in the system log files
    - *Discretionary ACL (DACL)*: access control list for object
- Access Control List
  - List of (Access Mask, SID) access control entries (ACEs)
  - Access Mask (bit set = access granted ; bit not set = access denied)
    - *Generic Access Types*: default RWX access rights given to all users
    - *Standard Access Types*: type of access rights that apply to all objects
    - *Specific Access Types*: access rights specific to this type of object
- Access Control Mechanism
  - When process attempts to access object the NT executive retrieves the SID and Group SID's from the access token and scans the objects DACL for a match

# Security Threats

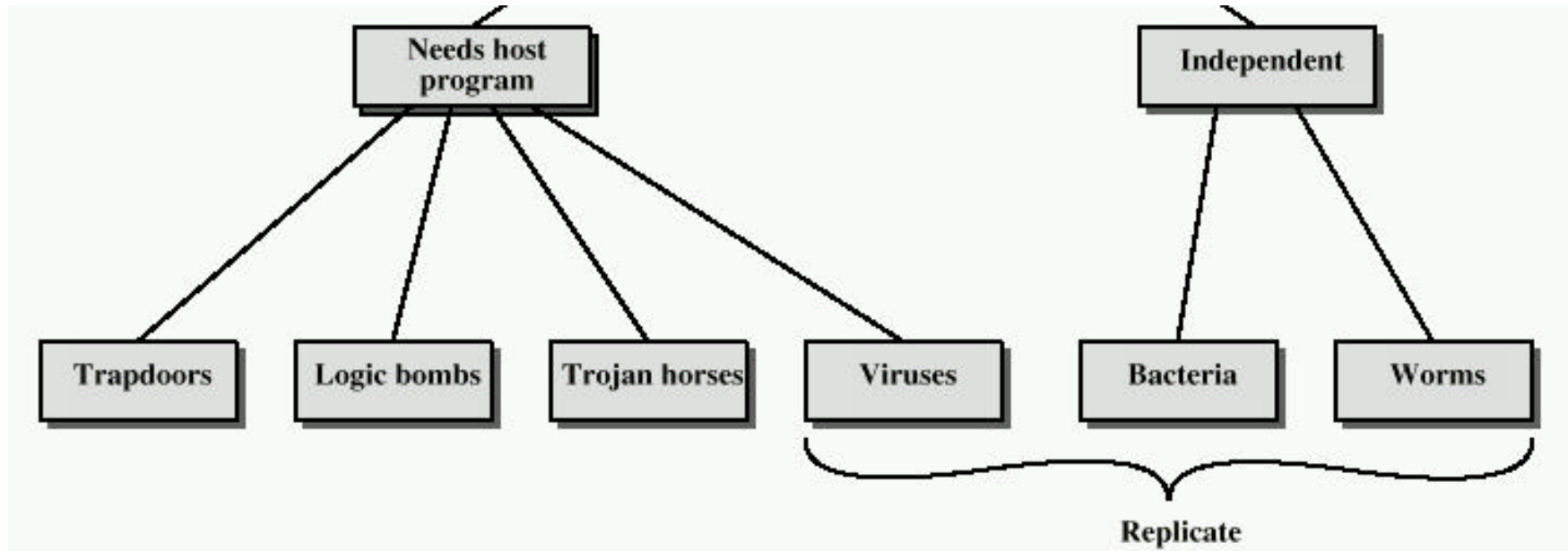


• Figure 15.2, OS3e: *Security Threats*

# Security Threats

- Passive Threats
  - Interception: also Eavesdropping, Monitoring
- Active Threats
  - Interruption: leads to *Denial of Service*
  - Modification and Fabrication: lead to Data Modification and Masquerade
    - *Data Modification*: delay, replay or re-order data to achieve unauthorised access
    - *Masquerade*: one entity (user or host) pretends to be another entity (e.g. by replaying an authentication sequence intercepted earlier)
- Design Principles
  - Least Privilege
    - Programs should execute with the least privilege allowable for the operations designed for
    - *UNIX Example*: Does `/bin/passwd` really require `{SETUID = root}` privileges?
  - Complete Mediation
    - Any and all accesses should be authenticated
  - Expect the Unexpected
    - All possible arguments and inputs should be expected and none should lead to unexpected program behaviour
    - *Classic Example*: Without array bounds checking, longer than expected inputs can overwrite the process stack → process code segment is modified!

# Security Threats: Malicious Programs



• Figure 15.8, OS3e: *Taxonomy of malicious programs*

- Needs Host Program
  - Code has to exist and be part of another program code or system operation
- Independent
  - Self-contained programs
- Replicate
  - Program replicates / clones itself to spread the infiltration to both widen the security breach coverage and increase likelihood of an important security breach

# Security Threats: Malicious Programs

- Trapdoors
  - Secret undocumented entry point into a program used to grant access without normal methods of access authentication
  - Fictional Example:
    - *War Games* the movie
- Logic Bomb
  - Logic embedded in a computer program that checks for a certain set of conditions to be present on the system. When these conditions are met, it executes some functioning that results in unauthorised access
  - Classic Example:
    - IT engineer plants a bomb that goes off if his/her payroll ID number is removed  
→ revenge exacted on employer if IT engineer is ever fired/sacked
- Trojan Horses
  - Secret undocumented routine embedded within a useful program. Execution of the program results in execution of the secret routine
  - Classic Example:
    - IT student compiles his/her own version of *'ls'* which in fact copies a *setuid* shell of the invoking user → relies on a PATH with a leading rather than trailing *'.'*

# Security Threats: Malicious Programs

- Viruses
  - Code embedded within a program that causes a copy of itself to be inserted in one or more other programs. In addition to propagation, the virus usually performs some unwanted function
  - Classic Example:
    - IBM PC Viruses
- Worms
  - Program that can replicate itself and send copies from computer to computer across network connections. Upon arrival, the worm may be activated to replicate and propagate again. In addition to propagation, the worm usually performs some unwanted function
  - Classic Example:
    - Nov. 2 1988 Internet worm attack by Cornell student, Robert Morris
- Bacteria
  - Program that consumes system resources by replicating itself, to the point where all system capacity (memory or disk space) is taken up
  - Variation on a theme Example:
    - Process which keeps spawning child processes indefinitely