# CSE 127 Computer Security

Stefan Savage, Spring 2019, Lecture 15

## System Security I: Security Mechanisms and Principles

# Threat Model Review

- ***Threat Model*** defines ***Security Goals***: What are we trying to protect from whom?
  - ***Assets*** and ***Attackers***
- Let's return to definition of a secure system
  - "System that remains dependable in the face of malice" -Ross Anderson
- What does "remains dependable" mean?
  - ***Confidentiality***, ***Integrity***, and/or ***Availability (C.I.A.)*** of particular system, component, or data are preserved
  - These are the ***Assets*** we need to protect
    - "An attacker must not be able to compromise the [Confidentiality | Integrity | Availability] of…"
  - Assets have value: Must understand value in order to decide how much to spend on protection
- What does "malice" mean?
  - Malice from whom? Curious roommate? Criminal? Law enforcement? Foreign military?
  - These are potential ***Attackers***
    - Defined by ***Capability*** and ***Intent***

# System Security

- How do we protect confidentiality, integrity, and availability of system components from attackers?
- ***Security policy***: Set of allowed actions in a system
  - Who is allowed to do what to what
- ***Security mechanism***: Part of system responsible for implementing the security policy
  - How the security policy is enforced
  - Example: encryption
- ***Security model***: Abstraction used by security mechanism
  - Policy may be formulated using the model

# Security Model

- ***Subjects***: acting system principals (the who part)
  - Traditionally: Users and Processes
  - Also: Apps, Site Domains, Peripherals, HW Blocks, etc.
- ***Objects***: protected system resources (the what part)
  - Traditionally: Memory Pages and Files
  - Also: System Calls, APIs, Web DOM, Web cookies, DB Tables, etc.
- Subjects operate on objects
  - System mediates and facilitates subject-object interaction
- Subjects can be objects
  - Example: a user may have privileges to provision new users on the system

# Security Policy

- Policy: what action is subject allowed to do with object?
  - And who can introduce new subjects and objects into system?
- Nearly all security models are built on this idea

# Access Control Matrix

- Security Policy can be represented as an ***Access Control Matrix***
  - Specifies permitted actions by subjects on objects

| | Objects | | | |
|---|---|---|---|---|
| Subjects | | | | |
| | | { allowed actions } | | |
| | | | | |
| | | | | |
| | | | | |

# Access Control Matrix Example

| | file1 | file2 | file3 | file4 | file5 |
|---|---|---|---|---|---|
| Alice | RW | | R | | RW |
| Bob | R | | | | RW |
| Carl | R | | R | RW | RW |
| Dave | R | | R | | RW |
| Eve | | | R | | RW |

# Security Mechanisms

- Implementation of the security policy
- Two main types of access control mechanisms
  - ***Access Control Lists (ACLs)***
  - ***Capabilities***

# Access Control Lists (ACLs)

- An ***access control list (ACL)*** of an object identifies which subjects can access the object and what they are allowed to do
- ACLs are object-centric: access control is associated with objects in the system
- Each access to object is checked against object's ACL
- Example: guest list at a club

# Access Control List Example

- file1 ACL:
  - Alice: RW, Bob: R, Carl: R, Dave: R

| | file1 | file2 | file3 | file4 | file5 |
|---|---|---|---|---|---|
| Alice | RW | | R | | RW |
| Bob | R | | | | RW |
| Carl | R | | R | RW | RW |
| Dave | R | | R | | RW |
| Eve | | | R | | RW |

# Capabilities

- A ***capability*** grants a subject permission to perform a certain action
  - Unforgeable (or at least difficult to forge)
  - Usually transferrable
- Capabilities are subject-centric: access control is associated with subjects in the system
- Example: movie ticket

# Capability Example

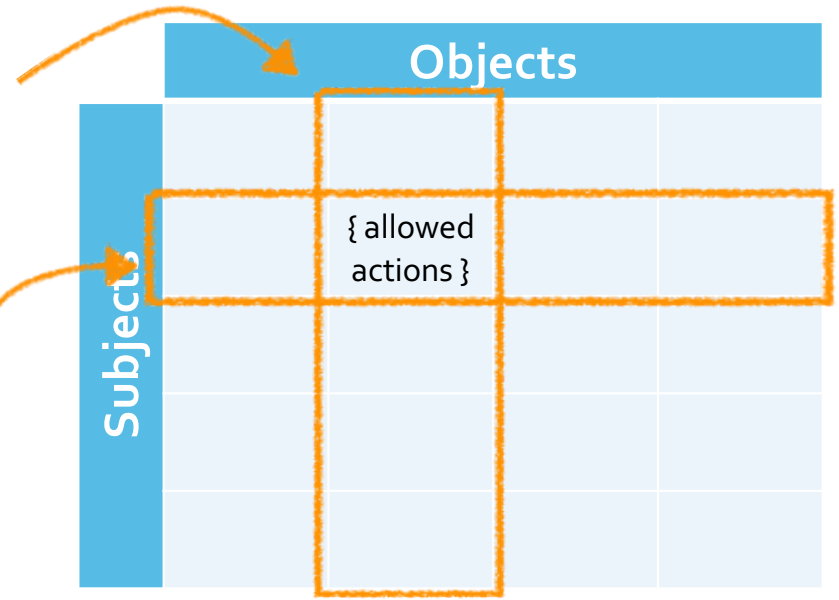
- Carl's Capability:
  - file1: R, file3: R, file4: RW, file5: RW

| | file1 | file2 | file3 | file4 | file5 |
|---|---|---|---|---|---|
| Alice | RW | | R | | RW |
| Bob | R | | | | RW |
| Carl | R | | R | RW | RW |
| Dave | R | | R | | RW |
| Eve | | | R | | RW |

# ACLs & Capabilities

- Columns of the Access Control Matrix define objects' ***ACLs***

- Rows of the Access Control Matrix define subjects' ***capabilities***

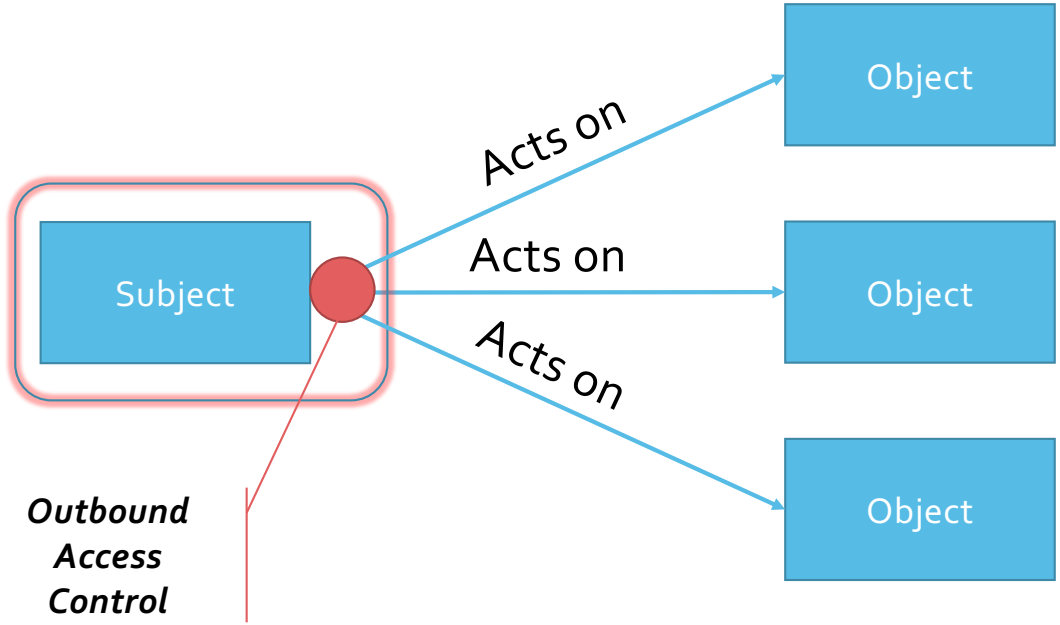


# Outbound vs Inbound Access Control

- An alternative formulation
- Subject(s) operate(s) on object(s)

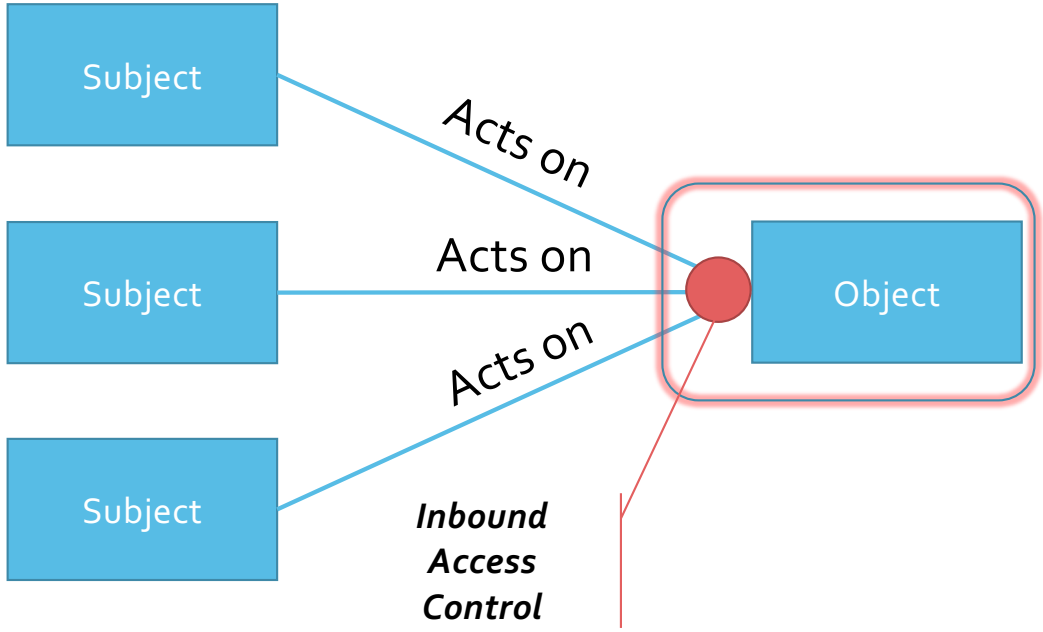
# Outbound Access Control

- We can enforce access control on action initiator



# Inbound Access Control

- We can enforce access control on action target



# Scaling Access Control

- Access control matrices can get very complex as number of subjects, objects, and possible operations grow
- "All problems in computer science can be solved by another level of indirection"
  - David Wheeler

| | file1 | file2 | file3 | file4 | file5 |
|---|---|---|---|---|---|
| Alice | RW | | R | | RW |
| Bob | R | | | | RW |
| Carl | R | | R | RW | RW |
| Dave | R | | R | | RW |
| Eve | | | R | | RW |

# Role-Based Access Control (RBAC)

- ***Role-Based Access Control***
  - Assign roles to subjects and control access to objects based on role
  - Examples: Administrator, HR, Student, etc.

# Attribute-Based Access Control (ABAC)

- RBAC is a special case of Attribute-Based Access Control
- ***Attribute-Based Access Control***
  - Assign Attributes to Subjects and/or Objects
    - Like tags
  - Access control matrix between subject attributes and object attributes
  - Access control matrix defines which attributes subjects need to have in order to access objects with given attributes
  - Example:
    - Subjects: students. Attribute: department
    - Objects: courses. Attributes: department, impacted
    - Only CSE students can take impacted CSE courses

# Attribute-Based Access Control (ABAC)

- Special cases of Attribute-based Access Control
  - Role-Based Access Control
    - Subjects have roles, object access depends on role
  - Hierarchical Access Control
    - Access policy depends on the structural relationship between objects (or subjects)
      - In a hierarchy of objects, children can inherit parent's access policy
      - Example: file access policy depends on parent directory access policy
  - Using context as an attribute
    - Time, location, movement, etc.

# Unix File System Security Model

# Unix File System Security Model

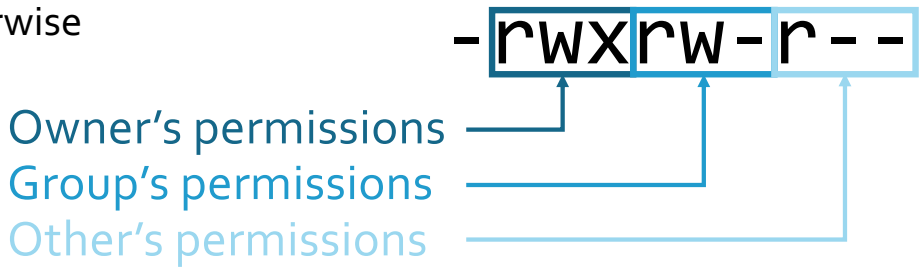
- *Subjects:* Users
- *Objects:* Files and Directories
- *Actions:* Read, Write, Execute
  - Execute a file means can call exec() on file
  - Directory "execute" means user can *traverse* it

# Unix Groups

- A user may belong to several ***groups***
  - Used for role-based access control
- Each user is associated with
  - one ***primary group***
  - optionally multiple ***supplementary groups***

# Unix File System Security Model

- Each file has an owner and a group
- File permissions specify what owner, group, and other (neither owner nor group) is allowed (read, write, exec)
- User's allowed actions on file are:
  - Owner's permissions if the user is the owner,
  - Group's permissions if the user is in the group,
  - Other's permissions otherwise



# Unix Superuser

- ***Superuser*** allowed to do anything that is possible
- Called ***`root`*** and mapped to user id **`0`**
- A superuser is a ***role*** rather than a particular user
- System administrators assume the superuser role to perform privileged actions
  - Good practice to assume superuser role only when necessary

# Permissions

- Only owner and superuser can change permissions
  - `chmod`
- Only superuser can change owner
  - `chown`
- Only owner and superuser can change group
  - `chgrp`
  - Owner can only change to group she belongs to

# Permissions

- Users interact with system via processes acting on their behalf
  - When you interact with system via terminal, the command shell acts on your behalf
- Each process is associated with a user
  - Every process is executing with an ***effective user id***, which determines allowed permissions

# Login

- When user connects to the system via a physical terminal, system runs the `login` process as root to start session
  - Authenticates user using username and password
  - Changes its user id and group id to that of user
    - This is possible because superuser allowed to do anything
  - Executes user's shell
- `sshd` performs similar actions
- *Critical*: **dropping privileges** from root to regular user

# Changing Privilege

- Superuser can drop privileges to become a regular user
- Regular users occasionally need elevated privileges
  - Example: change password
- How?

# Elevating Privilege

- Executable files have a `setuid` bit
  - Can only be set by the owner (or superuser)
- Normally, files are executed with the id (and associated privileges) of the user which executed them
- But, if `setuid` is set, the file is executed with the effective id (and associated privileges) of the file <u>owner</u>
  - ***Real user id*** (`ruid`) is that of executing user, but effective user if (`euid`) is that of owner
- The `passwd` command is owned by `root` and has `setuid` bit set.

# Elevating Privilege

- If a `setuid` binary has vulnerabilities that can be exploited by a user, it can lead to unauthorized ***elevation of privilege***
  - May allow arbitrary code execution with privileges of file owner

# Everything Is A File

- Unix has (mostly) a single abstraction for handling a wide variety of I/O operations
  - Files and directories …
  - Sockets and pipes …
  - Devices …
  - Kernel objects …
  - Process information…
  - Etc.
- Single abstraction for access control to above resources

# Android App Security Model

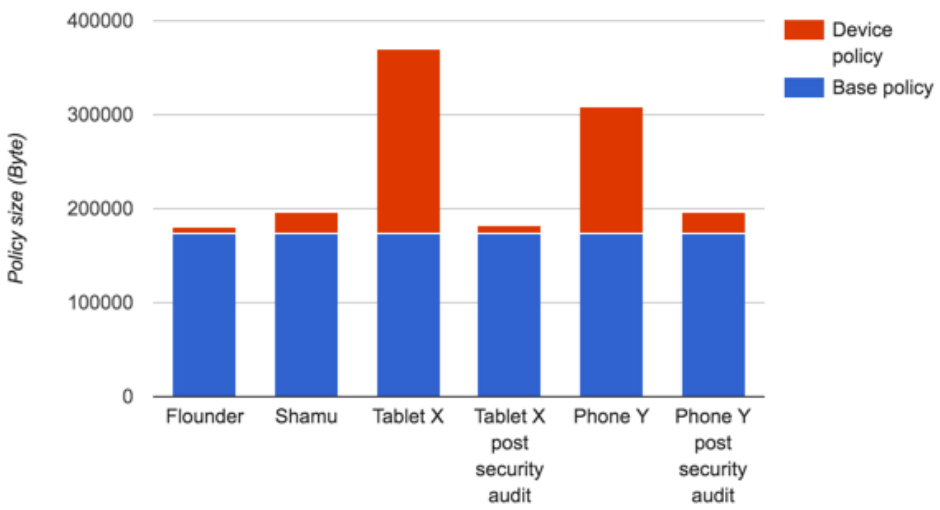
- Underlying OS for Android is Linux (which exports the Unix API)
- If system resources (peripheral devices) are abstracted as file descriptors, and we want to restrict which applications have access to which devices…
- Each app is a different user
  - Unique userid per app
  - File system security model and mechanism can be used to set and enforce app permissions
- Initially designed when mobile devices were single-user
  - Workaround to support multiple users: `u1_a23`

# DAC vs MAC

- ***Discretionary Access Control (DAC)***
  - Resource owners set security policy on their resources
  - Example: Unix file system security model
- ***Mandatory Access Control (MAC)***
  - Different MAC than in cryptography. Acronym collision!
  - Centralized system authority sets security policy on all system resources
  - Example: SELinux
    - A fine-grained MAC system for Linux (and derivatives)
    - Allows for further restriction of process privileges beyond those enforced by the standard file system security model

# SELinux

- A Security Context is associated with every subject and object
  - `user:role:type`
- A Security Policy describes the rules using the SELinux policy language.
- A Security Server that makes decisions regarding the subjects rights to perform the requested action on the object, based on the security policy rules.




https://source.android.com/security/selinux/device-policy

# Additional Resources

- SELinux
  - https://selinuxproject.org/page/NewUsers
- Android Security
  - https://source.android.com/security/
- iOS Security
  - https://www.apple.com/business/docs/iOS_Security_Guide.pdf

# Security Principles

# Security Principles

- 1974 classic *The Protection of Information in Computer Systems* by Jerome Saltzer and Michael Schroeder

- *"experience has provided some useful principles that can guide the design and contribute to an implementation without security flaws. Here are eight examples of design principles that apply particularly to protection mechanisms…"*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Economy of Mechanism

- ***Keep the design as simple and small as possible.***
  - *This well-known principle applies to any aspect of a system, but it deserves emphasis for protection mechanisms for this reason: design and implementation errors that result in unwanted access paths will not be noticed during normal use (since normal use usually does not include attempts to exercise improper access paths). As a result, techniques such as line-by-line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

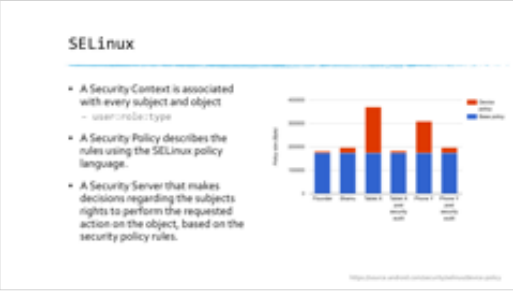
# Fail-safe Defaults

- ***Base access decisions on permission rather than exclusion.***
  - *This principle, suggested by E. Glaser in 1965, means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.*
- i.e. Whitelists are better than blacklists.

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Complete Mediation

- ***Every access to every object must be checked for authority.***
  - *This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.*




https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Open Design

- ***The design should not be secret.***
  - *The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user may be allowed to convince himself that the system he is about to use is adequate for his purpose. Finally, it is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.*
- i.e. Do not rely on security through obscurity.

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Separation Of Privilege

- ***Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.***
  - *The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information. This principle is often used in bank safe-deposit boxes. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command. In a computer system, separated keys apply to any situation in which two or more conditions must be met before access should be permitted. For example, systems providing user-extendible protected data types usually depend on separation of privilege for their implementation.*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Separation of privilege: AND-Redundancy vs OR-Redundancy

- We can categorize redundancy into two types:
  - When a system provides several alternative mechanisms, <u>any</u> of which can be used to accomplish a task, I call it ***OR-redundancy***
  - When a system provides several alternative mechanisms, <u>all</u> of which need to be used together to accomplish a task, I call it ***AND-redundancy***

http://dilbert.com/strip/1997-01-30

# Separation of privilege: OR-Redundancy

- When a system provides several alternative mechanisms, <u>any</u> of which can be used to accomplish a task, lets call it OR-redundancy
  - I can accomplish my task by satisfying mechanism X <u>or</u> mechanism Y.
  - Examples?
    - Multiple doors into a building – any one can serve as an entryway.
  - Generally bad for security.
    - Violates "economy of mechanism", makes it hard to enforce "complete mediation"

# Separation of privilege: AND-Redundancy

- When a system provides several alternative mechanisms, <u>all</u> of which need to be used together to accomplish a task, lets call it AND-redundancy
  - I can accomplish my task by satisfying mechanism X <u>and</u> mechanism Y
  - Examples?
    - Multiple locks on a door – all of them need to be unlocked to open the door.
  - Generally good for security.
    - Supports "separation of privilege" and "defense in depth"

# Least Privilege

- ***Every program and every user of the system should operate using the least set of privileges necessary to complete the job.***
  - *Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.*
- How much cash do you carry?

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Least privilege in practice: Isolation and Compartmentalization

- In order to exercise least privilege, it must be possible to isolate actions and responsibilities.
- Achieved via compartmentalization and isolation of individual components.
  - If privilege X is needed for accomplishing task A and privilege Y for task B, then we can minimize the privilege any one component needs by having two separate components working on A and B.
- Examples:
  - Process separation
  - Application sandboxing

# Least Common Mechanism

- ***Minimize the amount of mechanism common to more than one user and depended on by all users.***
  - *Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. For example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it.*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Psychological Acceptability

- ***It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.***
  - *Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

- *"Analysts of traditional physical security systems have suggested two further design principles which, unfortunately, apply only imperfectly to computer systems."*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Work Factor

- ***Compare the cost of circumventing the mechanism with the resources of a potential attacker.***
  - *The cost of circumventing, commonly known as the "work factor," in some cases can be easily calculated. For example, the number of experiments needed to try all possible four letter alphabetic passwords is $26^4$ = 456 976. If the potential attacker must enter each experimental password at a terminal, one might consider a four-letter password to be adequate. On the other hand, if the attacker could use a large computer capable of trying a million passwords per second, as might be the case where industrial espionage or military security is being considered, a four-letter password would be a minor barrier for a potential intruder. The trouble with the work factor principle is that many computer protection mechanisms are not susceptible to direct work factor calculation, since defeating them by systematic attack may be logically impossible. Defeat can be accomplished only by indirect strategies, such as waiting for an accidental hardware failure or searching for an error in implementation. Reliable estimates of the length of such a wait or search are very difficult to make.*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Compromise Recording

- ***It is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.***
  - *For example, if a tactical plan is known to have been compromised, it may be possible to construct a different one, rendering the compromised version worthless. An unbreakable padlock on a flimsy file cabinet is an example of such a mechanism. Although the information stored inside may be easy to obtain, the cabinet will inevitably be damaged in the process and the next legitimate user will detect the loss. For another example, many computer systems record the date and time of the most recent use of each file. If this record is tamperproof and reported to the owner, it may help discover unauthorized use. In computer systems, this approach is used rarely, since it is difficult to guarantee discovery once security is broken. Physical damage usually is not involved, and logical damage (and internally stored records of tampering) can be undone by a clever attacker.*

https://www.cs.virginia.edu/~evans/cs551/saltzer/

# Next Lecture…

System Security II