Design Principals and Common Security Related Programming Problems

Unit 5 [4 Hrs]

Principles for the design and implementation of security mechanisms

- Saltzer and Schroeder describes eight principles for the design and implementation of security mechanisms.
- The principles draw on the ideas of simplicity and restriction.
- Simplicity makes designs and mechanisms easy to understand and causes less errors.
- Minimizing the interaction of system components minimizes the number of sanity checks on data being transmitted from one components to another.
- Simplicity also reduces the potential for inconsistencies within a policy or set of policies.
- Restriction minimizes the power of an entity. The entity can access only information it needs.
- Entities can communicate with other entities only when necessary, and in as few and narrow ways as possbile.

Design Principles

- Principle of Least Privilege
- Principle of Fail-Safe Defaults
- Principle of Economy of Mechanism
- Principle of Complete Mediation
- Principle of Open Design
- Principle of Separation of Privilege
- Principle of Least Common Mechanism
- Principle of Psychological Acceptability

Principle of Least Privilege

- The *principle of least privilege states* that a subject should be given only those privileges that it needs in order to complete its task.
- If a subject does not need an access right, the subject should not have that right.
- Furthermore, the *function* of the subject should control the assignment of rights.
- If a specific action requires that a subject's access rights be increased, those extra rights should be withdrawn immediately on completion of the action.

Principle of Least Privilege

- This is analogue of the “need to know” rule: if the subject does not need access to an object to perform its task, it should not have the right to access that object.
- More precisely, if a subject needs to append to an object, but not to alter the information already contained in the object, it should be given append rights and not write rights
- e.g. the UNIX operating system does not apply access controls to the user *root*. That user can terminate any process and read, write, or delete any file. Thus, users who create backups can also delete files. The *administrator* account on Windows has the same powers.

Principle of Fail-Safe Defaults

- This principle restrict how privileges are initialized when a subject or object is created.
- The *principle of fail-safe defaults* state that, unless a subject is given explicit access to an object, it should be denied access to that object.
- This principle requires that the default access to an object is none.
- Whenever access, privileges, or some security-related attributes is not explicitly granted, it should be denied
- Moreover, if the subject is unable to complete its action or task, it should undo those changes it made in the security state of the system before it terminated. This way, even if program fails, the system is still safe.
- E.g. If the mail server is unable to create a file in the spool directory (waiting queue for printing), it should close the network connection, issue an error message and stop. It should not try to store the message elsewhere.

Principle of Economy of Mechanism

- This principle simplifies the design and implementation of security mechanism.
- The *principle of economy of mechanism* states that security mechanisms should be as simple as possible.
- If design and implementation are simple, fewer possibilities exist for errors.
- The checking and testing process is less complex, because fewer components and cases need to be tested.
- Complex mechanisms often make assumptions about the system and environment in which they run.
- If these assumption are incorrect, security problems may result.

Principle of Complete Mediation

- This principles restricts *the caching of information, which often leads to simpler implementations of mechanisms*.
- The *principle of complete mediation* **requires that all accesses to objects be checked to ensure that they are allowed**.
- Whenever, a subject attempts to read an object, the operating system should mediate the action.
- First, it determines if the subject is allowed to read the object. If so, it provides the resource for the read to occur.
- If the subject tries to read the object again, the system should check that the subject is still allowed to read the object.
- Most systems would not make the second check. They would cache the results of the first check and base the second access on the cached results.

Principle of Complete Mediation

- Example:
- When a UNIX process tries to read a file, the OS determines if the process is allowed to read the file. If so, the process receives a file descriptor encoding the allowed access. Whenever, the process want to read the file, it presents the file descriptor to the kernel. The kernel then allows the access.
- If the owner of the file disallows the process permission to read the file descriptor is issued, the kernel still allows access. This scheme violates the principle of complete mediation, because the second access is not checked. The cached value is used, resulting in the denial of access being ineffective

Principle of Open Design

- This principle suggest that *complexity does not add security*.
- The *principle of open design states* **that the security of a mechanism should not depend on the secrecy of its design or implementation**.
- Designer and implementer of a program must not depend on secrecy of the details of their design and implementation to ensure security. Others can ferret out such details through technical means or non technical means.
- If the strength of the program's security depends on the ignorance of the user, a knowledgeable user can defeat that security mechanism.

Principle of Separation of Privilege

- This principle is *restrictive because it limits access to system entities*.
- The principle of separation of privilege **states that a system should not grant permission based on a single condition**.
- The systems and programs granting access to resources should allowed to met more than one condition. This provides a fine-grained control over the resource as well as additional assurance that the access is authorized.

Principle of Separation of Privilege

- Example:
- On Berkeley-based versions of the UNIX operating system, users are not allowed to change from their accounts to the root account unless two conditions are met.
  - The first condition is that the user knows the root password.
  - The second condition is that the user is in the wheel group (the group with GID 0).
- Meeting either condition is not sufficient to acquire root access; meeting both conditions is required.

Principle of Least Common Mechanism

- This principle is restrictive because it limits sharing.
- The *principle of least common mechanism* **states that mechanisms used to access resources should not be shared**.
- Sharing resources provides a channel along which information can be transmitted, and so such sharing should be minimized.

Principle of Psychological Acceptability

- This principle recognizes the human element is computer security.
- The *principle of psychological acceptability* **states that security mechanism should not make the resource more difficult to access than if the security mechanisms were not present**.
- Configuring and executing a program should be easy, and any output should be clear, direct, and useful.
- If security-related software is too complicated to configure, system administrators may unintentionally set up the software in a non-secure manner.
- Similarly, security-related user programs must be easy to use and must output understandable messages.
- Example: Giving clear message when login fails due to wrong message. Giving correct parameter information when error occurs during configuration of program.

Common Security Related Programming Problems

- Programmers make mistakes sometimes and those errors can have disastrous consequences in protection domains.
- Attackers who exploit these errors may acquire extra privileges (such as access to a system account such as root or Administrator).
- They may disrupt the normal functioning of the system by deleting or altering services over which they should have no control.
- They may simply be able to read files to which they should have no access.
- So the problem of avoiding these errors, or security holes, is a necessary issue to ensure that the programs and system function as required.

Common Security Related Programming Problems

- Improper Choice of Initial Protection Domain
- Improper isolation of implementation detail
- Improper Change
- Improper Naming
- Improper Validation
- Improper Indivisibility
- Improper Sequencing
- Improper Choice of Operand or Operation

Improper Choice of Initial Protection Domain

- Flaws involving improper choice of initial protection domain arise from incorrect setting of permissions or privileges.
- There are **three objects** for which permission to be set properly: *the file containing the program*, *the access control file*, and *the memory space of the process*.

Improper Choice of Initial Protection Domain

Process privileges

- The principle of least privilege dictates that no process have more privileges than it needs to complete its task, but the process must have enough privileges to complete its task successfully.
- Ideally, set of privileges should meet both criteria.
- In practice, different portions of the process will need different set of privileges.

Improper Choice of Initial Protection Domain

Process privileges

- *Implementation Rule*: Structure the process so that all sections requiring extra privileges are modules. The modules should be as small as possible and should perform only those tasks that require those privileges.
- *Management Rule*: check that the process privileges are set properly.

Improper Choice of Initial Protection Domain

Access Control File Permissions

- Biba's models emphasize that the integrity of the process relies on both the integrity of the program and the integrity of the access file.
- The former requires managers requires that the program be properly protected so that authorized personnel can alter it.
- The system managers must determine who the "authorized personnel" are.
- Verifying the integrity of the access control file is critical, because that file controls the access to role accounts.

Improper Choice of Initial Protection Domain

Memory Protection

- Consider sharing memory: If two subjects can alter the contents of memory, then one could change data on which the second relies.
- Unless such sharing is required (for example, by concurrent processes), it poses a security problem because the modifying process can alter variable that control the action of the other process.
- Thus, each process should have a protected unshared memory space.
- If the memory is represented by an object that processes can alter, it should be protected so that only trusted processes can access it.

Improper Choice of Initial Protection Domain

Trust in the system

- This analysis overlooks several system components.
- For example, the program relies on the system authentication mechanisms to authenticate the user, and on the user information database to map users and roles into their corresponding UIDs.
- It also relies on the inability of ordinary users to alter the system clock.
- If any of this supporting infrastructure can be compromised, the program will not work correctly.

Improper Isolation of Implementation Detail

- The problem of improper isolation implementation detail arises when an abstractions is improperly mapped into an implementation detail.
- e.g. database query producing error or fail in some other way.
- The first rule is to catch errors and failures of the mappings.
- This requires an analysis of the functions and a knowledge of their implementation.
- The action to take on failure also requires thought.
- Implementation rule: The error status of every function must be checked. Do not try to recover unless the cause of the error, and its effects, do not affect any security considerations. The program should restore the state of the system to the state before the process began, and then terminate.
- The abstractions in this program are the notion of a user and a role, the access control information, the creation of a process with the rights of the role.

Improper Isolation of Implementation Detail

Resource Exhaustion and User Identifiers

- The notion of a user and a role is an abstraction because the program can work with role names and the operating system user integers (UIDs).
- The question is how those user and role names are mapped to UIDs.
- Typically, this is done with a user information database that contains the requisite mapping, but the program must detect any failures of the query and respond appropriately.

Improper Isolation of Implementation Detail

Validating the Access Control Entries

- The access control information implements the access control policy (an abstraction).
- The expression of the access control information is therefore the result of mapping an abstraction to an implementation.
- The question is whether or not the given access control information correctly implements the policy.
- Answering this question requires someone to examine the implementation expression of the policy.

Improper Isolation of Implementation Detail

Restricting the Protection Domain of the Role Process

- Creating a role process is the third abstraction. There are two approaches.
- Under UNIX-like systems, the program can spawn a second, *child*, process. It can also simply start up a second program in such a way that the parent process is replaced by the new process.
- This technique, called ***overlaying***, is intrinsically simpler than creating a child process and exiting.
- It allows the process to replace its own protection domain with the (possibly) more limited one corresponding to the role. The programmers elected to use this method. The new process inherits the protection domain of the original one. Before the overlaying, the original process must reset its protection domain to that of the role. The programmers do so by closing all files that the original process opened, and changing its privileges to those of the role.

Improper Change

- This category describes data and instructions that change over time. The danger is that the changed values may be inconsistent with the previous values. The previous values dictate the flow of control of the process. The changed values cause the program to take incorrect or non-secure actions on that path of control.
- The data and instructions can reside in shared memory, in non-shared memory, or on disk. The last includes file attribute information such as ownership and access control list.

Improper Change

Memory

- First comes the data in shared memory. Any process that can access shared memory can manipulate data in that memory.
- Unless all processes that can access the shared memory implement a concurrent protocol for managing changes, one process can change data on which a second process relies; this could cause the second process to violate the security policy.
- A second approach applies whether the memory is shared or not. A user feeds bogus information to the program, and the program accepts it. The bogus data overflows its buffer, changing other data, or inserting instructions that can be executed later.

Improper Change

Changes in File Contents

- File contents may change improperly. In most cases, this means that the file permissions are set incorrectly or that multiple processes are accessing the file, which is similar to the problem of concurrent processes accessing shared memory.
- Components that may change between the time the program is created and the time it is run.

Improper Change

Race Conditions in File Access

- A race condition in this context is *the time-of-check-to-time-of-use* problem.
- As with memory accesses, the file being used is changed after validation but before access.
- The file must be protected so that no untrusted user can alter it, or the process must validate and use it correctly.

Improper Naming

- Improper naming *refers to an ambiguity in identifying an object*.
- Most commonly, two different objects have the same name. The programmer intends the name to refer to one of the objects, but an attacker manipulates the environment and the process so that the name refers to a different object.
- Avoiding this flaw requires that every object be unambiguously identified. This is both a management concern and an implementation concern.
- Objects must be uniquely identifiable or completely interchangeable.

Improper Deallocation and Deletion

- Failing to delete sensitive information raises the possibility of another process seeing that data at a later time. In particular, cryptographic keywords, passwords, and other authentication information should be discarded once they have been used. Similarly, once a process has finished with a resource, that resource should be deallocated. This allows other processes to use that resource, inhibiting denial of service attacks.
- A consequence of not deleting sensitive information is that dumps of memory, which may occur if the program receives an exception or crashes for some other reason, contain the sensitive data. If the process fails to release sensitive resources before spawning unprivileged subprocesses, those unprivileged subprocesses may have access to the resource.

Improper Validation

- The problem of improper validation arises when data is not checked for consistency and correctness.
- Ideally, a process would validate the data against the more abstract policies to ensure correctness. In practice, the process can check correctness only by looking for error codes (indicating failure of functions and procedures) or by looking for patently incorrect values (such as negative numbers when positive ones are required).
- As the program is designed, the developers should determine what conditions must hold at each interface and each block of code. They should then validate that these conditions hold.

Improper Validation

Bounds Checking

- Errors of validation often occur when data is supposed to lie within bounds.
- For example, a buffer may contain entries numbered from 0 to 99.
- If the index used to access the buffer elements takes on a value less than 0 or greater than 99, it is an invalid operand because it accesses a nonexistent entry.

Improper Validation

Type Checking

- Failure to check types is another common validation problem. If a function parameter is an integer, but the actual argument passed is a floating point number, the function will interpret the bit pattern of the floating point number as an integer and will produce an incorrect result.
- So, the types of functions and its parameter must be checked.
- Also, while compiling the program, the compiler flags must report the inconsistencies in types.

Improper Validation

Error Checking

- A third common problem involving improper validation is failure to check return values of functions.
- For example, suppose a program needs to determine ownership of a file. It calls a system function that returns a record containing information from the file attribute table. The program obtains the owner of the file from the appropriate field of the record. If the function fails, the information in the record is meaningless.
- So, if the function's return status is not checked, the program may act erroneously.
- Implementation Rule: Check all function and procedure executions for errors.

Improper Validation

Checking for Valid, Invalid Data

- Validation should apply the principle of *fail-safe defaults*.
- This principle requires that valid values be known, and that all other values be rejected.

Improper Validation

Checking Input

- All data from untrusted sources must be checked. Users are untrusted sources. The checking done depends on the way the data is received: into an input buffer (bounds checking) or read in as an integer (checking the magnitude and sign of the input).
- I*mplementation Rule*: Check all user input for both form and content. In particular, check integers for values that are too big or too small, and check character data for length and valid characters.

Improper Validation

Designing for Validation

- Sometimes data cannot be validated completely.
- For example, in the C programming language, a programmer can test for a NULL pointer (meaning that the pointer does not hold the address of any object), but if the pointer is not NULL, checking the validity of the pointer may be very difficult (or impossible).
- Using a language with strong type checking is another example.
- *Implementation Rule*: Create data structures and functions in such a way that they can be validated.

Improper Indivisibility

- Improper indivisibility arises when an operation is considered as one unit (indivisible) in the abstract but is implemented as two units (divisible).
- The checking of the access control file attributes and the opening of that file are to be executed as one operation.
- Unfortunately, they may be implemented as two separate operations, and an attacker who can alter the file after the first but before the second operation can obtain access illegally.

Improper Indivisibility

- Another example arises in exception handling.
- Often, program statements and system calls are considered as single units or operations when the implementation uses many operations.
- An exception divides those operations into two sets: the set before the exception, and the set after the exception.
- If the system calls or statements rely on data not changing during their execution, exception handlers must not alter the data.
- *Implementation Rule*: If two operations must be performed sequentially without an intervening operation, use a mechanism to ensure that the two cannot be divided.

Improper Sequencing

- Improper sequencing means that operations are performed in an incorrect order.
- For example, a process may create a lock file and then write to a log file.
- A second process may also write to the log file, and then check to see if the lock file exists.
- The first program uses the correct sequence of calls; the second does not (because that sequence allows multiple writers to access the log file simultaneously).
- *Implementation Rule*: Describe the legal sequences of operations on a resource or object. Check that all possible sequences of the program(s) involved match one (or more) legal sequences.

Improper Choice of Operand and Operation

- Preventing errors of choosing the wrong operand or operation requires that the algorithms be thought through carefully (to ensure that they are appropriate).
- At the implementation level, this requires that operands be of an appropriate type and value, and that operations be selected to perform the desired functions.
- The difference between this type of error and improper validation lies in the program.
- Improper implementation refers to a validation failure.
- The operands may be appropriate, but no checking is done.
- In this category, even though the operands may have been checked, they may still be inappropriate.
- Assurance techniques help detect these problems.
- As a management rule, use software engineering and assurance techniques (such as documentation, design reviews, and code reviews) to ensure the operation and operands are appropriate.