Unit 5
Systems Design

Topics

- Design principles
- Representing identity
- Control of access and information flow
- Confinement problem
- Assurance: Building systems with assurance, formal methods,

Evaluating systems.

Systems design

- **Systems design** is the process of defining the architecture, product design, modules, interfaces, and data for a system to satisfy specified requirements.

- Systems design could be seen as the application of systems theory to product development.

Secured Design properties

- Simplicity makes designs and mechanisms easy to understand.
- Simplicity reduces the potential for inconsistencies within a policy or set of policies.
- Minimizing the interaction of system components minimizes the number of sanity checks on data being transmitted among components.
- A sanity check is an essential procedure to check the presence of any errors in the initial process. A sanity check focuses on possible errors that may appear in the initial process of setting up.

The Importance of Good Design Principles

- Every design, whether it be for hardware or software, must begin with a design philosophy and standard guiding principles.

- These principles cover the design, are built in from the beginning, and are preserved (according to the design philosophy) as the design evolves.

Design principles

- There are eight principles for the design and implementation of security mechanisms.
  1. Principle of least privilege.
  2. Principle of Fail-Safe Defaults.
  3. Principle of Economy of Mechanism
  4. Principle of Complete Mediation
  5. Principle of Open Design
  6. Principle of Separation of privilege
  7. Principle of least common mechanism
  8. Principle of Psychological acceptability

Principle of least privilege

- Restricts how privileges are granted.
- Principle states that "a subject should be given only those privilege that it needs in order to complete its task."
- If a specific action requires that a subject's access rights be augmented, those extra rights be relinquished/claimed immediately on completion of the action.

Principle of Fail-safe Defaults

- Restricts how privileges are initialized when a subject or object is created.
- "The principle of Fail-Safe defaults states 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.

Principle of economy of mechanism

- Simplifies the design and implementation of security mechanisms.
- "The principle of economy of mechanism states that security mechanism should be as simple as possible"
- If a design and implementations are simple, fewer possibilities exist for errors.
- Simpler means less can go wrong And when errors occur, they are easier to understand and fix

Principle of complete mediation

- Complete mediation meaning every access to every object must be checked for authority.
- This principle 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 OS should mediate the action.
- First, it determines if the subject is allowed to read the object. If so, it provides the resources for the read to occur.
- If the subject tries to read it 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(future requests for that data can be served faster) the results of the first check and base the second access on the cached results.

Principle of open design

- The principle suggests 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".
- Designers of a program must not depend on secrecy of the details of their design and implementation to ensure security.
- If the strength of the program's security depends on the ignorance of the user, a knowledgeable user can defeat that security mechanism.
- Does not apply to information such as passwords or cryptographic keys

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 single condition."

- Company cheques for more than $75,000 must be signed by two officers of the company. If either does not sign, the cheque is not valid. The two conditions are the signatures of two officers.

- This provides a fine-grained control over the resource as well as additional assurance that the access is authorized.

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.
- Covert channels

Principle of psychological acceptability

- This principle recognizes the human element in 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."
- If security related software is too complicated to configure, system administrators may unintentionally set up software in a non secure manner.
- Security-related user programs must be easy to use and must output understandable messages.
- Captcha

Which principle(s) you would like to incorporate while designing Security systems for following:

- 1)Research Lab for developing COVID-19 vaccine
- 2)Library
- 3)Shopping mall

Representing Identity

- Identity is simply a computer's representation of an entity.
- **Definition**

A *principal* is a unique entity. An *identity* specifies a principal.

- Identities are used for several purposes.
- accountability
  - Accountability requires an identity that tracks principals across actions and changes of other identities, so that the principal taking any action can be unambiguously identified.
- access control
  - Access control requires an identity that the access control mechanisms can use to determine if a specific access (or type of access) should be allowed.

Files and objects

- The identity of a file or other entity (here called an “object”) depends on the system that contains the object.
- Local systems identify objects by assigning names.
- The name may be intended for human use (such as a file name), for process use (such as a file descriptor or handle), or for kernel use (such as a file allocation table entry).
- Each name may have different semantics(meaning, or truth.).

Users

- In general, a *user* is an identity tied to a single entity.
- Specific systems may add additional constraints.
- Systems represent user identity in a number of different ways.
- Indeed, the same system may use different representations of identity in different contexts.
- EXAMPLE:
- Versions of the UNIX operating system usually represent user identity as an integer between 0 and some large integer (usually 65,535). This integer is called the *user identification number*, or UID. Principals (called *users*) may also be assigned *login names*. Each login name corresponds to a single UID (although one UID may have many different login names)

Groups and Roles

- The "entity" may be a set of entities referred to by a single identifier.
- Principals often need to share access to files.
- Most systems allow principals to be grouped into sets called, logically enough, *groups*.
- Groups are essentially a shorthand tool for assigning rights to a set of principals simultaneously.
  - EXAMPLE: UNIX users are assigned membership to a group when they log in. Each process has two identities, a "user identification" and a "group identification."
- A *role* is a type of group that ties membership to function.
- When a principal assumes a role, the principal is given certain rights that belong to that role.
- When the principal leaves the role, those rights are removed.

Naming

- The identifier corresponds to a principal.
- The identifier must uniquely identify the principal to avoid confusion.
- Suppose the principals are people. The identifiers cannot be names, because many different people may have the same name.
- The identifiers must include supportive information to distinguish the "Matt Bishop" who teaches at UC Davis from the "Matt Bishop" who works at Microsoft Corporation.

Certificates

- Certification authorities (CAs) vouch, at some level, for the identity of the principal to which the certificate is issued. Every CA has two policies controlling how it issues certificates.
  1. A *CA authentication policy* describes the level of authentication required to identify the principal to whom the certificate is to be issued.
  2. A *CA issuance policy* describes the principals to whom the CA will issue certificates

The Meaning of the Identity

- The authentication policy defines the way in which principals prove their identities.
- Each CA has its own requirements.
- All rely on non electronic proofs of identity, such as biometrics (fingerprints), documents (driver's license, passports), or personal knowledge.
- If any of these means can be compromised, the CA may issue the certificate in good faith to the wrong person

Trust

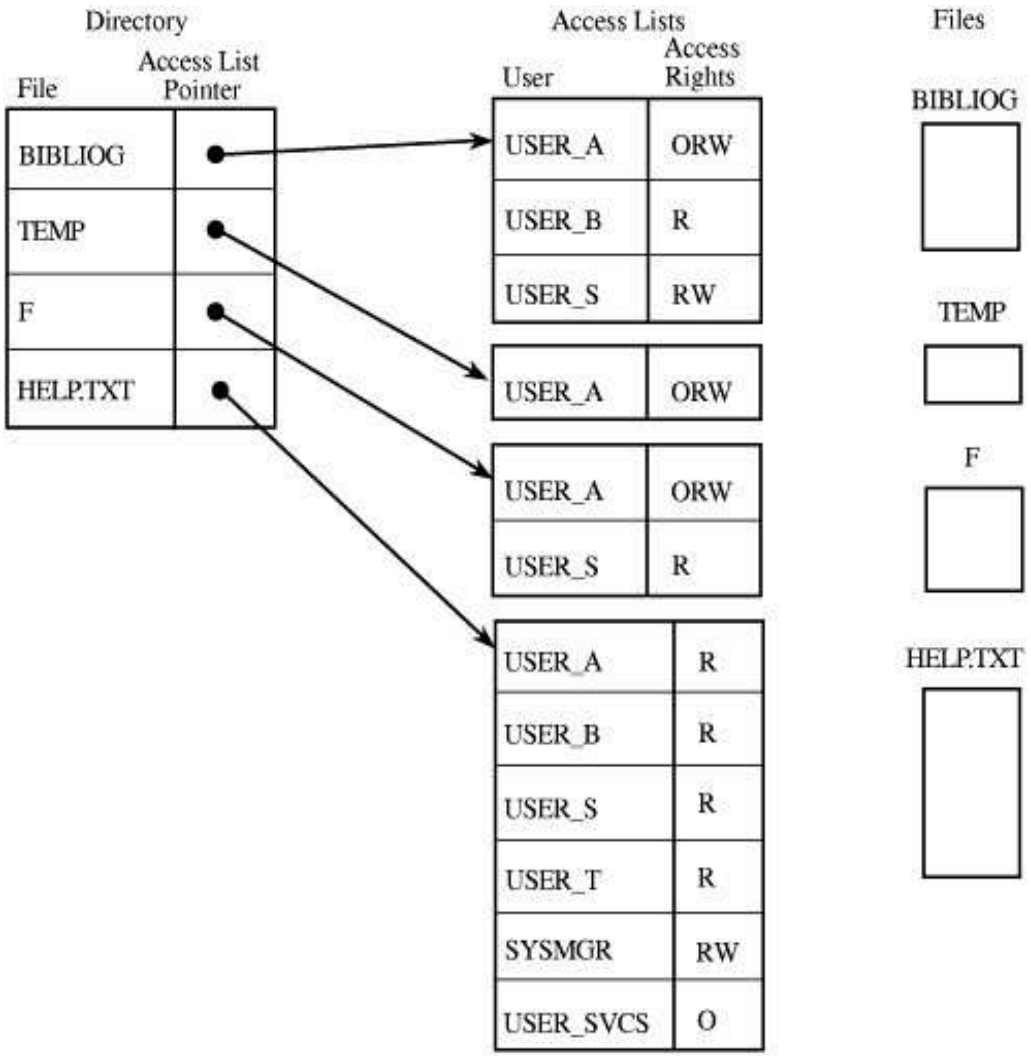
- The goal of certificates is to bind the correct identity to the public key.
- When a user obtains a certificate, the issuer of that certificate is vouching, to some degree of certainty, that the identity corresponds to the principal owning the public key.
- EXAMPLE: Consider the CA that requires a passport to issue a certificate. The certificate will have the name in the passport, the name of the country issuing the passport, and the passport number.

Identity on the Web

- Certificates are not common on the Internet.
- Several other means attach identity to information, even though the binding may be very transient.
- The Internet requires every host to have an address. The address may be fixed or may change, and without cryptography the binding is weak.

Control of Access

Access Control List



- There is one such list for each object, and the list shows all subjects who should have access to the object and what their access is.

Creation and Maintenance of ACL

Specific implementations of ACLs differ in details. Some of the issues are as follows.

- 1. Which subjects can modify an object's ACL?
- 2. If there is a privileged user (such as *root* in the UNIX system or *administrator* in Windows NT), do the ACLs apply to that user?
- 3. Does the ACL support groups or wildcards?
- 4. How are contradictory access control permissions handled? If one entry grants read privileges only and another grants write privileges only, which right does the subject have over the object?
- 5. If a default setting is allowed, do the ACL permissions modify it, or is the default used only when the subject is not explicitly mentioned in the ACL?

Which Subjects Can Modify an Object's ACL?

- When an ACL is created, rights are instantiated. Chief among these rights is the one we will call *own*.
- Possessors of the *own* right can modify the ACL.
- Creating an object also creates its ACL, with some initial value.
- By convention, the subject with *own* rights is allowed to modify the ACL.
- However, some systems allow anyone with access to manipulate the rights.

Does the ACL Support Groups and Wildcards?

- In its classic form, ACLs do not support groups or wildcards.
- In practice, systems support one or the other (or both) to limit the size of the ACL and to make manipulation of the lists easier.
- A group can either refine the characteristics of the processes to be allowed access or be a synonym for a set of users (the members of the group).

ACLs and Default Permissions

- When ACLs and abbreviations of access control lists or default access rights coexist (as on many UNIX systems), there are two ways to determine access rights.

1. The first is to apply the appropriate ACL entry, if one exists, and to apply the default permissions or abbreviations of access control lists otherwise.
2. The second way is to augment the default permissions or abbreviations of access control lists with those in the appropriate ACL entry.

Revocation of Rights

- *Revocation*, or the prevention of a subject's accessing an object, requires that the subject's rights be deleted from the object's ACL.
- Preventing a subject from accessing an object is simple.
- The entry for the subject is deleted from the object's ACL.
- If only specific rights are to be deleted, they are removed from the relevant subject's entry in the ACL.

Capabilities

- Conceptually, a capability is like the row of an access control matrix.
- Each subject has associated with it a set of pairs, with each pair containing an object and a set of rights.
- The subject associated with this list can access the named object in any of the ways indicated by the named rights.

Locks and Keys

- The locks and keys technique combines features of access control lists and capabilities.
- A piece of information (the lock) is associated with the object and a second piece of information (the key) is associated with those subjects authorized to access the object and the manner in which they are allowed to access the object.
- When a subject tries to access an object, the subject's set of keys is checked. If the subject has a key corresponding to any of the object's locks, access of the appropriate type is granted.
- locks and keys are dynamic in nature. An access control list is static in the sense that all changes to it are manual; a user or process must interact with the list to make the change.
- Locks and keys, on the other hand, may change in response to system constraints, general instructions about how entries are to be added, and any factors other than a manual change.

Ring-Based Access Control

- To understand its simplicity and elegance, one must realize that files and memory are treated the same from the protection point of view.
- For example, a procedure may occupy a segment of the disk. When invoked, the segment is mapped into memory and executed.
- Data occupies other segments on disk, and when accessed, they are mapped into memory and accessed.
- In other words, there is no conceptual difference between a segment of memory and a segment on a disk.

Propagated Access Control Lists(PACL)

- It provides the creator of an object with control over who can access the object.
- The creator (originator) is kept with the PACL, and only the creator can change the PACL.
- When a subject reads an object, the PACL of the object is associated with the subject.
- When a subject creates an object, the PACL of the subject is associated with the object.

Information flow

- Information flow policies define the way information moves throughout a system.
- Typically, these policies are designed to preserve confidentiality of data or integrity of data.
- In the former, the policy's goal is to prevent information from flowing to a user not authorized to receive it.
- In the latter, information may flow only to processes that are no more trustworthy than the data.

Compiler-Based Mechanisms

- Compiler-based mechanisms check that information flows throughout a program are authorized.
- The mechanisms determine if the information flows in a program *could* violate a given information flow policy.

- **Definition :** A set of statements is *certified* with respect to an information flow policy if the information flow within that set of statements does not violate the policy.

Declarations

- For our discussion, we assume that the allowed flows are supplied to the checking mechanisms through some external means, such as from a file.
- The specifications of allowed flows involve security classes of language constructs.
- The program involves variables, so some language construct must relate variables to security classes.
- One way is to assign each variable to exactly one security class.

Program Statements

- A program consists of several types of statements. Typically, they are
- 1. Assignment statements
- 2. Compound statements
- 3. Conditional statements
- 4. Iterative statements
- 5. Goto statements
- 6. Procedure calls
- 7. Function calls
- 8. Input/output statements.
- We consider each of these types of statements separately, with two exceptions.

1. Function calls can be modeled as procedure calls by treating the return value of the function as an output parameter of the procedure.
2. Input/output statements can be modeled as assignment statements in which the value is assigned to (or assigned from) a file.

- Hence, we do not consider function calls and input/output statements separately.

Execution-Based Mechanisms

- The goal of an execution-based mechanism is to prevent an information flow that violates policy.
- Checking the flow requirements of explicit flows achieves this result for statements involving explicit flows.
- EXAMPLE: Let x and y be variables. The requirement for certification for a particular statement y *op* x is that $x \leq y$.
- The conditional statement

  if x = 1 then y = a;

  causes a flow from x to y.

  Now, suppose that when $x \neq 1$, x = *High* and y = *Low*.

If flows were verified only when explicit, and $x \neq 1$, the implicit flow would not be checked.

Example Information Flow Controls

- Like the program-based information flow mechanisms discussed in last slide, both special purpose and general-purpose computer systems have information flow controls at the system level.
- File access controls, integrity controls, and other types of access controls are mechanisms that attempt to inhibit the flow of information within a system, or between systems.
  - The first example is a special-purpose computer that checks I/O operations between a host and a secondary storage unit. It can be easily adapted to other purposes.
  - A mail guard for electronic mail moving between a classified network and an unclassified one follows.
- The goal of both mechanisms is to prevent the illicit flow of information from one system unit to another.

The Confinement Problem

The confinement problem deals with prevention of processes from taking disallowed actions. OR

According to Lampson:

The confinement problem is the problem of preventing a server from leaking information that the user of the service considers confidential.

- Consider a client and a server. When the client issues a request to the server, the client sends the server some data.
- The server then uses the data to perform some function and returns a result (or no result) to the client.
- Access control affects the function of the server in two ways.
  - 1. The server must ensure that the resources it accesses on behalf of the client include only those resources that the client is authorized to access.
  - 2. The server must ensure that it does not reveal the client's data to any other entity not authorized to see the client's data.

- According to Lampson:
- **Definition :**The *confinement problem* is the problem of preventing a server from leaking information that the user of the service considers confidential.

- **Definition:** A *covert channel* is a path of communication that was not designed to be used for communication.

- **Definition:** The *rule of transitive confinement* states that if a confined process invokes a second process, the second process must be as confined as the caller.

Isolation

- Systems isolate processes in two ways.

1. In the first, the process is presented with an environment that appears to be a computer running only that process or those processes to be isolated.
2. In the second, an environment is provided in which process actions are analyzed to determine if they leak information.

- The first type of environment prevents the process from accessing the underlying computer system and any processes or resources that are not part of that environment.
- The second type of environment does not emulate a computer. It merely alters the interface between the existing computer and the process(es).

Virtual Machines

- The first type of environment is called a *virtual machine*.
- **Definition :** A *virtual machine* is a program that simulates the hardware of a (possibly abstract) computer system.
- A virtual machine uses a special operating system called a *virtual machine monitor* to provide a virtual machine on which conventional operating systems can run.

Sandboxes

- The computer sandbox provides a safe environment for programs to execute in.
- If the programs "leave" the sandbox, they may do things that they are not supposed to do.
- Both types of sandboxes restrict the actions of their occupants.

Definition : A *sandbox* is an environment in which the actions of a process are restricted according to a security policy.

- Systems may enforce restrictions in two ways.
- First, the sandbox can limit the execution environment as needed.
- The second enforcement method is to modify the program (or process) to be executed.

Covert Channels

- Covert channels use shared resources as paths of communication.
- This requires sharing of space or sharing of time.
- **Definition :** A *covert storage channel* uses an attribute of the shared resource. A *covert timing channel* uses a temporal or ordering relationship among accesses to a shared resource.

- **Definition :** A *noiseless covert channel* is a covert channel that uses a resource available to the sender and receiver only. A *noisy covert channel* is a covert channel that uses a resource available to subjects other than the sender and receiver, as well as to the sender and receiver.

Mitigation of Covert Channels

- Covert channels convey information by varying the use of shared resources.
- An obvious way to eliminate all covert channels is to require processes to state what resources they need before execution and provide these resources in such a manner that only the process can access them.
- This includes runtime, and when the stated runtime is reached, the process is terminated and the resources are released.
- The resources remain allocated for the full runtime even if the process terminates earlier.

Mitigation .....

- An alternative approach is to obscure the amount of resources that a process uses.
- This can be done in two ways.
  - First, the resources devoted to each process can be made uniform. In essence, the system eliminates meaningful irregularities in resource allocation and use.
  - Second, a system can inject randomness into the allocation and use of resources. The goal is to make the covert channel a noisy one and to have the noise dominate the channel.

Introduction to Assurance

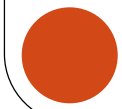
Assurance for secure and trusted systems must be an integral part of the development process. Confidence gained as result of evidence.

Trust

- *Trustworthy* entity has sufficient credible evidence leading one to believe that the system will meet a set of requirements.

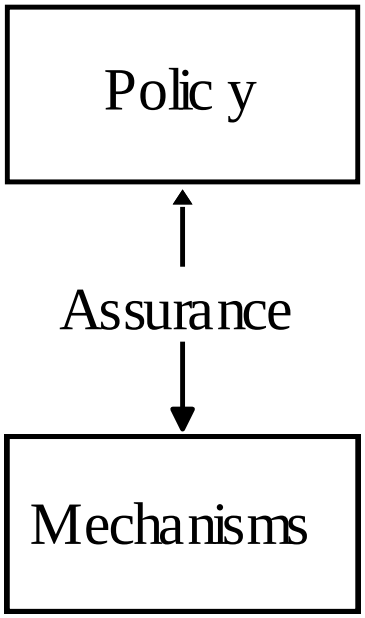
- *Trust* is a measure of trustworthiness relying on the evidence.
  - To trust makes one vulnerable to violations trust.

- *Assurance* is confidence that an entity meets its security requirements based on evidence provided by applying assurance techniques.
  - Meets security requirements" == Enforces policy



Trusted System

- A *trusted system* is a system that has been shown to meet well-defined requirements under an evaluation by a credible body of experts who are certified to assign trust ratings to evaluated products and systems.



Statement of requirements that explicitly defines the security expectations of the mechanism(s)

Provides justification that the mechanism meets policy through assurance evidence and approvals based on evidence

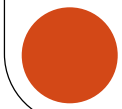
Executable entities that are designed and implemented to meet the requirements of the policy

The need of Assurance

- Applying assurance techniques is time-consuming and expensive.
- Accidental or unintentional failures of computer systems, as well as intentional compromises of security mechanisms, can lead to security failures.

Problem Sources -- Neumann's list

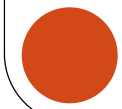
1. Policy Flaws
2. Requirements definitions, omissions, and mistakes
3. System design flaws
4. Hardware implementation flaws, such as wiring and chip flaws
5. Software implementation errors, program bugs, and compiler bugs
6. System use and operation errors and inadvertent mistakes
7. Willful system misuse (A serious or high degree of negligence and unmistakable abuse of duty of legal right towards others)
8. Hardware, communication, or other equipment malfunction
9. Environmental problems, natural causes, and acts of God
10. Evolution, maintenance, faulty upgrades, and decommissions



- Implementation assurance deals with hardware and software implementation errors (items 3, 4, and 7),
- errors in maintenance and upgrades (item 9),
- willful misuse (item 6), and environmentally induced problems (item 8).
- Thorough security testing as well as detailed and significant vulnerabilities assessment find flaws that can be corrected prior to deployment of the system.
- Operational assurance can address system use and operational errors (item 5)
- as well as some willful misuse issues (item 6).
- Neumann's list is not exclusive to security

Examples

- Space Shuttle Challenger explosion
  - Sensors removed from booster rockets to meet accelerated launch schedule
- Deaths from faulty radiation therapy system
  - Hardware safety interlock removed
  - Flaws in software design
- Intel 486 chip
  - Bug in trigonometric functions
  - Intel's public reputation was damaged, and replacing the chips cost Intel time and money.

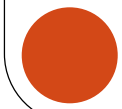


Role of Requirements

- *Requirements* are statements of goals that must be met.
  - Vary from high-level, generic issues to low-level, concrete issues.

- *Security objectives* are high-level security issues.

- *Security requirements* are specific, concrete issues.

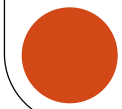


Types of Assurance

1. *Policy assurance* **is evidence** establishing security requirements in policy is complete, consistent, technically sound.

2. *Design assurance* **is evidence** establishing design sufficient to meet requirements of security policy.

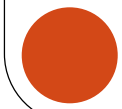
3. *Implementation assurance* **is evidence** establishing implementation consistent with security requirements of security policy



Types of Assurance

4. Operational assurance is evidence establishing system sustains the security policy requirements during installation, configuration, and day-to-day operation

- Also called *administrative assurance*
- One fundamental operational assurance technique is a thorough review of product or system documentation and procedures, to ensure that the system cannot accidentally be placed into a non-secure state.
- This emphasizes the importance of proper and *complete* documentation for computer applications, systems, and other entities.



Assurance Throughout the Life Cycle

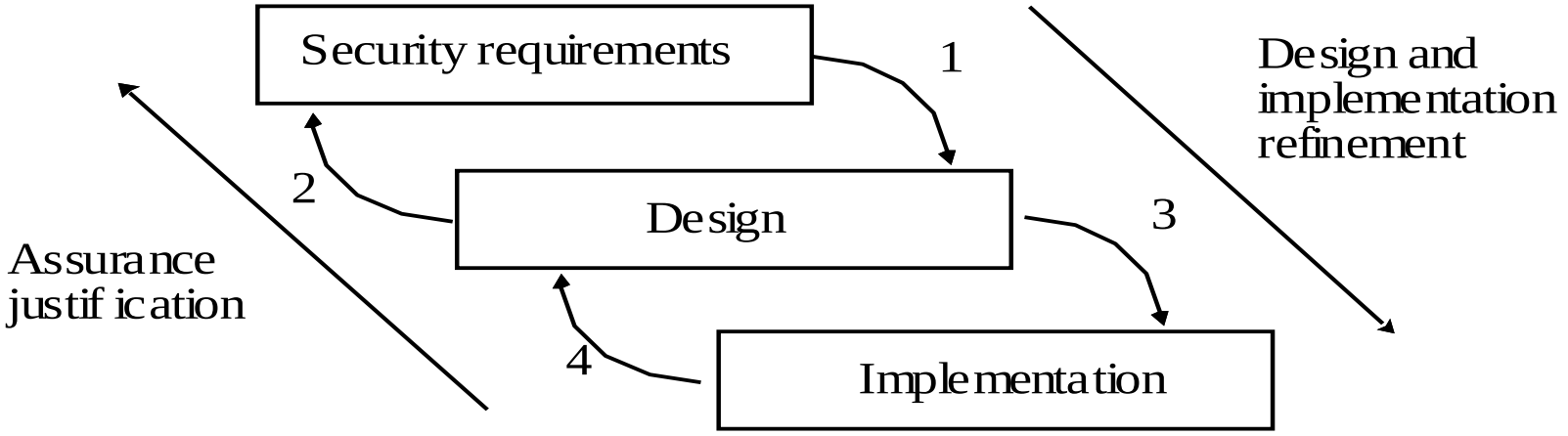
- The goal of assurance is to show that an implemented and operational system meets its security requirements throughout its life cycle.
- Because of the difference in the levels of abstraction between high-level security requirements and low-level implementation details, the demonstration is usually done in stages.

Building Secure and Trusted Systems

- Building secure and trusted systems depends on standard software engineering techniques augmented with specific technologies and methodologies.
- Hence, a review of the life cycles of systems will clarify much of what follows.
  - Life cycle

Life Cycle

- This process is usually iterative, because assurance steps identify flaws that must be corrected. When this happens, the affected steps must be rechecked.



- Assurance must continue throughout the life of the system. Because maintenance and patching usually affect the system design and implementation.
- **Note that the refinement steps alternate with the assurance steps.**

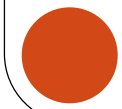
Life Cycle Assurance

1. Conception
   - Initial focus is on policy and requirements
2. Manufacture
   - Select mechanisms to enforce policy
   - Give evidence that mechanisms are appropriate
3. Deployment
   - Prepare operational plans that realize policy goals
   - Provide mechanism for distribution and delivery that assures product integrity
   - Support appropriate configuration
4. Fielded Product Life
   - Update and patch mechanism
   - Customer support
   - Product decommissioning and end of life

Waterfall Life Cycle Model

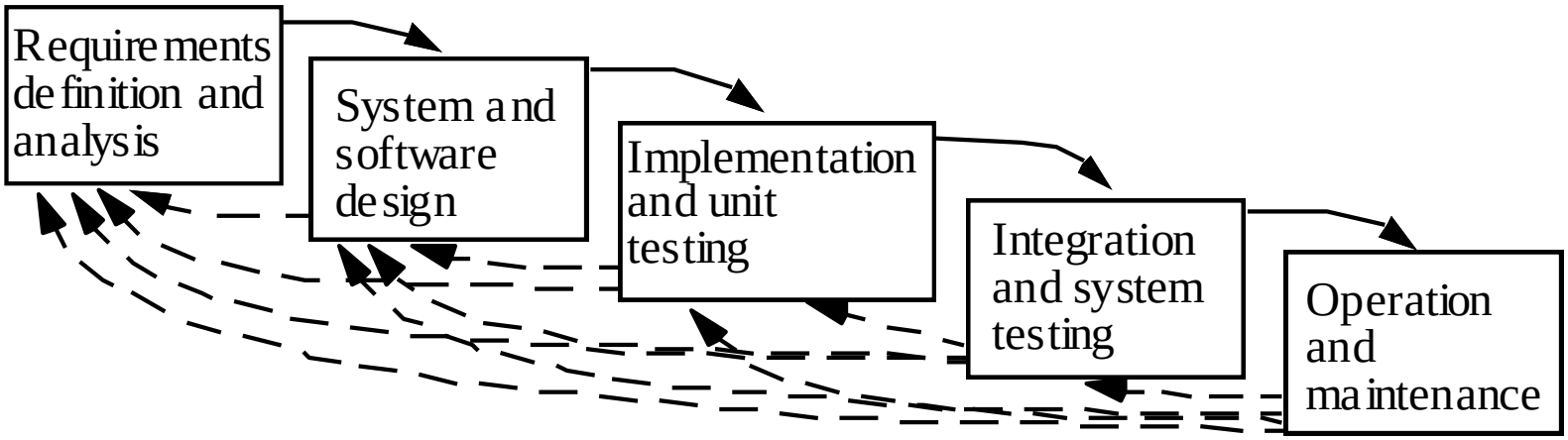
- The *waterfall life cycle model* is the model of building in stages, whereby one stage is completed before the next stage begins.
- Five Steps are :

1. Requirements definition and analysis
   - Functional and non-functional
   - General (for customer), specifications
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance



Relationship of Stages

- **The waterfall life cycle model. The solid arrows represent the flow of development in the model. The dashed arrows represent the paths along which information about errors may be sent.**



1. Requirements Definition and Analysis

- In this phase, a feasibility study and may examine whether or not the requirements are correct, consistent, complete, realistic, verifiable, and traceable.
- It is likely that there will be some iteration between the requirements definition step and the architecture step before either can be completed.
  - Functional requirements describe interactions between the system and its environment.
  - Nonfunctional requirements are constraints or restrictions on the system that limit design or implementation choices.

System and Software Design

- This stage is sometimes broken into the two phases *system design*, in which the system as a whole is designed, and *program design*, in which the programs of the system are individually designed.
- Software design further partitions the requirements into specific executable programs.
  - The external functional specifications describe the inputs, outputs, and constraints on functions that are external to the entity being specified,
  - whereas the internal design specifications describe algorithms to be used, data structures, and required internal routines.

Implementation and Unit Testing

- *Implementation* is the development of software programs based on the software design from the previous step. Typically, the work is divided into a set of programs or program units.

- *Unit testing* is the process of establishing that the unit as implemented meets its specifications. It is in this phase that many of the supporting processes described earlier come into play.

Integration and System Testing

- *Integration* is the process of combining all the unit-tested program units into a complete system.

- *System testing* is the process of ensuring that the system as a whole meets the requirements. System testing is an iterative step because invariably bugs and errors are found that have to be corrected.

Operation and Maintenance

- Once the system is finished, it is moved into production. This is called ***fielding the system***.
- Maintenance involves correction of errors that have been reported from the field and that have not been corrected at earlier stages.
- This stage also involves routine maintenance and the release of new versions of the system.
- Finally, retirement/deployment of the system also falls under this phase.

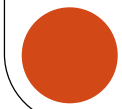
Other Models

1. Exploratory programming

- Develop working system quickly and then modified until it performs adequately.
- No requirements or design specification, so difficulty in assurance.
- Therefore, this model is not particularly useful for building secure and trusted systems because such systems need precise requirements and detailed verification that they meet those requirements as implemented.

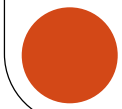
2. Prototyping

- Objective is rapid development to establish system requirements



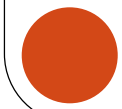
Other Models

3. Formal transformation
   - Create formal specification
   - Translate it into program using correctness-preserving transformations
   - Very conducive to assurance methods
4. System assembly from reusable components
   - Depends on whether components are trusted
   - Must assure connections, composition as well
   - Very complex, difficult to assure



Other Models

5. Extreme programming
   - Rapid prototyping and "best practices"
   - Project driven by business decisions
   - Requirements open until project complete
   - Programmers work in teams
   - Components tested, integrated several times a day
   - Objective is to get system into production as quickly as possible, then enhance it
   - Evidence adduced *after* development needed for assurance



Key Points

- Assurance critical for determining trustworthiness of systems
- Different levels of assurance, from informal evidence to rigorous mathematical evidence
- Assurance needed at all stages of system life cycle
- Building security in is more effective than adding it later

Evaluation

- Evaluation is a process in which the evidence for assurance is gathered and analyzed against criteria for functionality and assurance. Perfect security is an ultimate, but unachievable, goal for computer systems
- A *formal evaluation methodology* is a technique used to provide measurements of trust based on specific security requirements and evidence of assurance.

An evaluation methodology provides the following features.

- A set of requirements defining the security functionality for the system or product.
- A set of assurance requirements that delineate the steps for establishing that the system or product meets its functional requirements. The requirements usually specify required evidence of assurance.
- A methodology for determining that the product or system meets the functional requirements based on analysis of the assurance evidence.
- A measure of the evaluation result (called a *level of trust*) that indicates how trustworthy the product or system is with respect to the security functional requirements defined for it.

Evaluation system are :

1. TCSEC(Trusted Computer System Evaluation Criteria, also known as the Orange Book): 1983–1999
2. FIPS(Federal Information Processing Standard) 140: 1994–Present
3. The Common Criteria(CC): 1998–Present
4. SSE-CMM: 1997–Present

SSE-CMM : 1997- Present

- System Security Engineering Capability Maturity Model (SSE-CMM) is a process-oriented methodology for developing secure systems.
- The SSE-CMM became ISO Standard 21827 in 2002.
- The SSE-CMM is organized into processes and maturity levels.
- Generally speaking, the processes define what needs to be accomplished by the security engineering process and the maturity levels categorize how well the process accomplishes its goals.

- A *process capability* is the range of expected results that can be achieved by SSE CMM process. It is a predictor of future project outcomes.

- *Process performance* is a measure of the actual results achieved.

- *Process maturity* is the extent to which a process is explicitly defined, managed, measured, controlled, and effective.

SSE-CMM contains 11 process areas.

1. Administer Security Controls
2. Assess Impact
3. Assess Security Risk
4. Assess Threat
5. Assess Vulnerability
6. Build Assurance Argument
7. Coordinate Security
8. Monitor System Security Posture
9. Provide Security Input
10. Specify Security Needs
11. Verify and Validate Security

- The definition of the Assess Threat process area contains the goal that threats to the security of the system be identified and characterized. The base processes are :
  - Identify Natural Threats
  - Identify Human-Made Threats
  - Identify Threat Units of Measure
  - Assess Threat Agent Capability
  - Assess Threat Likelihood
  - Monitor Threats and Their Characteristics

The five Capability Maturity Levels that represent increasing process maturity are as follows.

1. *Performed Informally*. Base processes are performed.
2. *Planned and Tracked*. Project-level definition, planning, and performance verification issues are addressed.
3. *Well-Defined*. The focus is on defining and refining a standard practice and coordinating it across the organization.
4. *Quantitatively Controlled*. This level focuses on establishing measurable quality goals and objectively managing their performance.
5. *Continuously Improving*. At this level, organizational capability and process effectiveness are improved.

James Anderson's "Computer Security Planning Study" provides a blueprint

- Needs analysis:
  - Multi-level operation
  - Systems connected to the world
  - On-line operation
  - Networks
- Vision
  - Security engineering
  - Secure components (hardware & software)
  - Handbook of Computer Security Techniques

Evaluation Assurance Level

- EAL 1: functionally tested
- EAL 2: structurally tested
- EAL 3: methodically tested and checked
- EAL 4: methodically designed, tested and reviewed
- EAL 5: semiformally designed and tested
- EAL 6: semiformally verified design and tested
- EAL 7: formally verified design and tested

Common Criteria

- International standard
- EAL 1 -- 5 transferred across borders
- EAL 6 and 7 are not