Software Security Overview

- The purpose of Software Security is to provide a structured overview of software implementation vulnerabilities and techniques to prevent or mitigate these vulnerabilities.
- This knowledge is crucial for both academic and industry professionals.
- Security in software involves maintaining confidentiality, integrity, and availability.

Categories of Vulnerabilities

- **Memory Management Vulnerabilities:** Include spatial (e.g., buffer overflow) and temporal (e.g., dangling pointers) vulnerabilities.
- **Structured Output Generation Vulnerabilities:** Includes SQL injection and script injection vulnerabilities.
- **Race Condition Vulnerabilities:** Occur when concurrent actors access shared resources, leading to non-deterministic behavior.
- **API Vulnerabilities:** Caused by incorrect usage of APIs, especially those handling security functions.
- **Side-channel Vulnerabilities:** Exploitation of physical aspects of software execution, such as timing or power consumption.

1. Memory Management Vulnerabilities

Memory Management in Programming Languages:

- Imperative programming languages allow for the allocation, access, and deallocation of memory cells.
- The correct use of these memory constructs is defined by the programming language, which acts as a contract for memory management.
- Some languages, like C and C++, do not enforce memory safety, leaving responsibility to the programmer, and leading to undefined behavior when memory is mismanaged.

Types of Memory Management Vulnerabilities:

- **Spatial Vulnerability**: Occurs when a program accesses memory outside the bounds of an allocated array, **such as in buffer overflow attacks.**
- **Temporal Vulnerability**: Arises when a program **accesses memory that has already been deallocated,** such as dereferencing a dangling pointer. If a program attempts to access or modify the memory through a dangling pointer, it can lead to unpredictable behavior, including crashes and data corruption.

Security Risks of Memory Management Vulnerabilities

- Memory management vulnerabilities are particularly dangerous because they can corrupt program code, control flow, and data.
- These vulnerabilities are prevalent in languages like C and C++ due to their lack of enforced memory safety.

Exploitation Techniques:

- **Code Corruption Attack**: The attacker modifies program code through invalid memory access.
- **Control-Flow Hijack Attack**: The attacker manipulates code pointers to execute malicious code or reuse existing code unexpectedly.
- **Data-Only Attack**: The attacker alters program data to gain unauthorized privileges.
- **Information Leak Attack**: The attacker reads memory to extract sensitive information like cryptographic keys or memory addresses, aiding further attacks.

2. Structured Output Generation Vulnerabilities

- **Definition and Importance**:
- Structured output generation involves creating dynamic output, such as SQL queries or HTML pages, which will be consumed by other programs.
- The structure of this output is intended to follow a specific contract, ensuring that input is correctly incorporated into the output.

Common Insecure Practice

- A frequent insecure practice is constructing structured output through string manipulation, which leaves the structure implicit and vulnerable to exploitation.
- This practice can lead to unintended output if maliciously crafted input is provided, leading to significant security risks.

Types of Structured Output Generation Vulnerabilities

- **SQL Injection:** Occurs when a SQL query is constructed with input that manipulates the query's structure, often bypassing security checks.
- **Command Injection**: Involves constructing shell commands that include user input, allowing attackers to execute unintended commands.
- **Script Injection (XSS):** This happens when user input is used to construct JavaScript code sent to a browser, enabling the execution of malicious scripts on the client side.

Challenges in Avoiding Vulnerabilities:

- **Complex Syntax**: Structured output languages, like HTML, may support sublanguages with different syntactic structures, complicating the prevention of vulnerabilities.
- **Multi-Phase Computation**: Structured output may be computed in phases, with output from one phase used as input in another, increasing the risk of vulnerabilities like stored XSS or higher-order SQL injection.

Exercise 1: Identify and Analyze Real-World SQL Injection Cases

- **Objective**: Research and identify at least three real-world examples of SQL injection attacks.
- **Tasks**:
  - Describe the context in which each attack occurred (e.g., the type of application or service that was targeted).
  - Explain how the SQL injection was executed by the attackers.
  - Discuss the impact of each attack on the targeted organization and its users.
  - Analyze what could have been done differently to prevent these vulnerabilities.

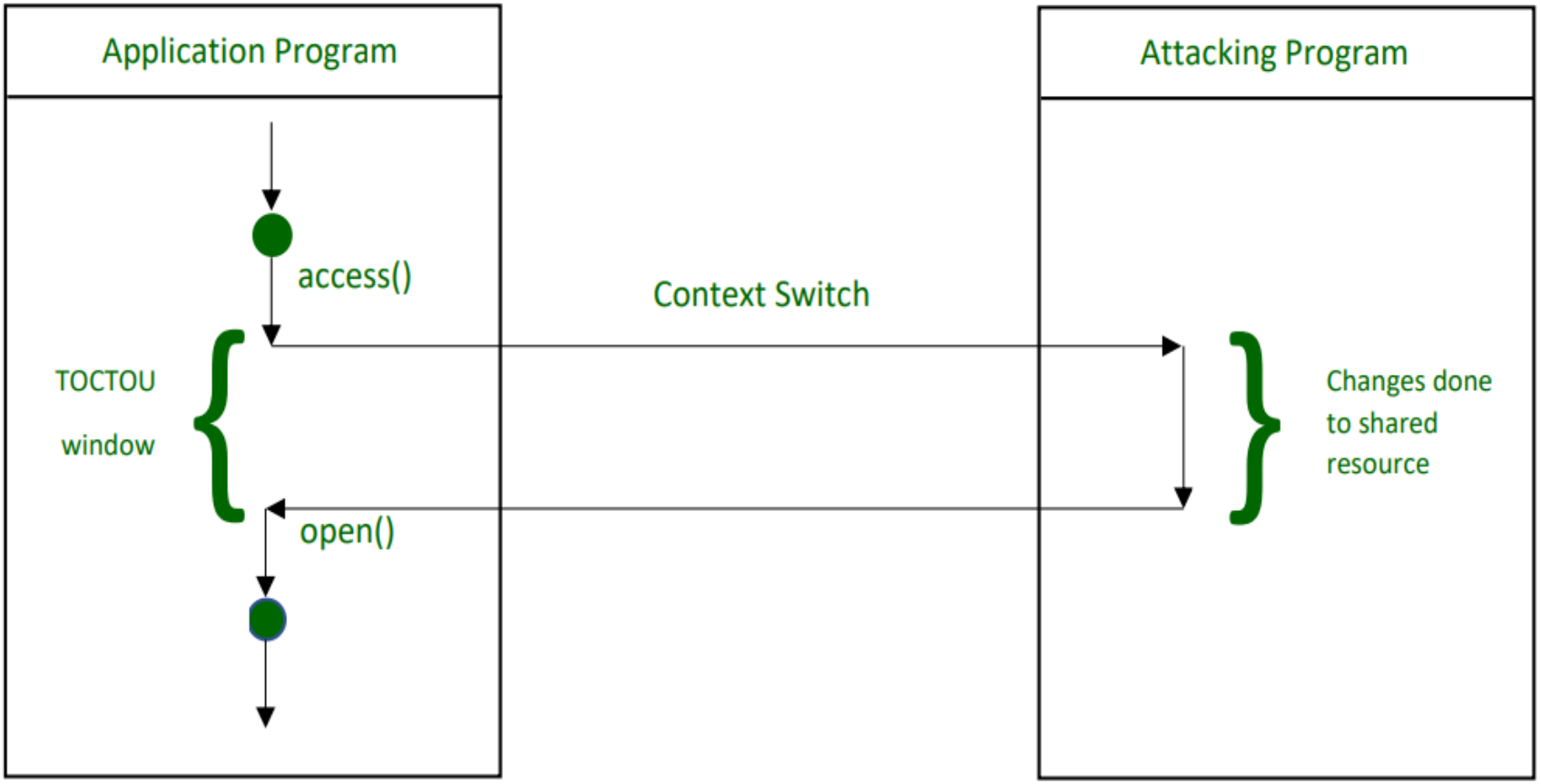
Exercise 2: Experiment with Input Sanitization Techniques

- **Objective**: Understand the role of input sanitization in preventing structured output generation vulnerabilities.
- **Tasks**:
  - Write a simple web application (e.g., using PHP or Python) that constructs SQL queries based on user input.
  - Introduce a SQL injection vulnerability by directly concatenating user input into the query.
  - Test the vulnerability by trying to inject malicious SQL commands.
  - Implement input sanitization (e.g., using prepared statements) and test how it prevents the injection.
  - Document your findings and explain the importance of proper input sanitization.

3. Race Condition Vulnerabilities

- **Race condition vulnerabilities** arise in software when multiple concurrent actors (e.g., threads or processes) access shared resources (like memory, files, or databases).
- The program often assumes that these actors will interact with shared resources in a specific way, but these assumptions might not always hold true, leading to unexpected behavior.
- This situation creates a "**race**" where the outcome depends on which actor accesses the resource first, hence the term "**race condition”**.

- **Explore: the example Time Of Check Time Of Use (TOCTOU) vulnerability.**


Application Program
Attacking Program
access()
Context Switch
TOCTOU
window
Changes done
to shared
resource
open()

Common Types of Race Condition Vulnerabilities:

- **File System Race Conditions:** Occur in privileged programs that need to check a condition on a file before acting on it for a less privileged user. If the check and action aren't done atomically (i.e., without interruption), an attacker can exploit the gap to change the file, leading to a security breach.

- **Session State Races in Web Applications:** Web servers often handle HTTP requests using multiple threads for better performance. If these threads access the same session state concurrently without proper management, it can lead to race conditions, potentially corrupting the session state.

4. API Vulnerabilities

- **API Vulnerabilities** refer to security issues that arise when a software component communicates with another component via an Application Programming Interface (API).
- APIs provide a set of rules and protocols that dictate how these components interact, including how services are requested and delivered.
- An API typically comes with a specification or contract that outlines how it should be used and what it offers.

Contract Violations and Security Risks:

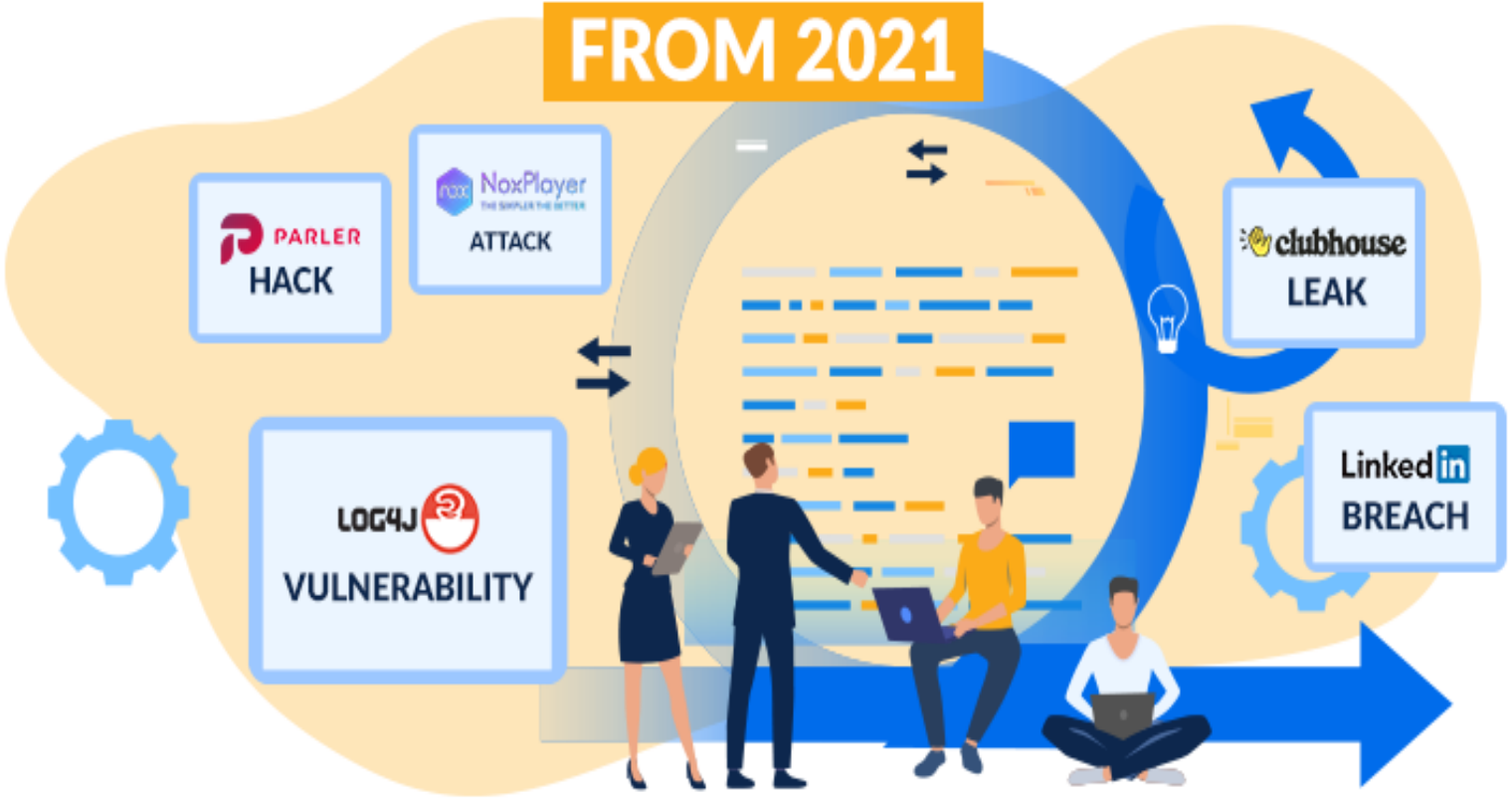
- Just like other contracts in software, if the client (the software component using the API) violates the API's contract, the system may enter an error state.
- This error state can have unpredictable consequences, potentially leading to security vulnerabilities.
- When an API's contract is violated, the resulting behavior of the software system often depends on the underlying implementation details of the API, which may be exploited by attackers.

Security-Sensitive APIs:

- Some APIs are inherently more sensitive than others, particularly those related to security functions such as cryptography and access control.
- For example, cryptographic libraries often provide flexible APIs, but using them correctly to achieve specific security objectives is notoriously difficult.
- Many developers unintentionally introduce vulnerabilities when using these APIs because of their complexity. Empirical studies have shown that mistakes in the use of cryptographic APIs are common, leading to significant security risks.
- **Explore: Heartbleed Bug in OpenSSL**

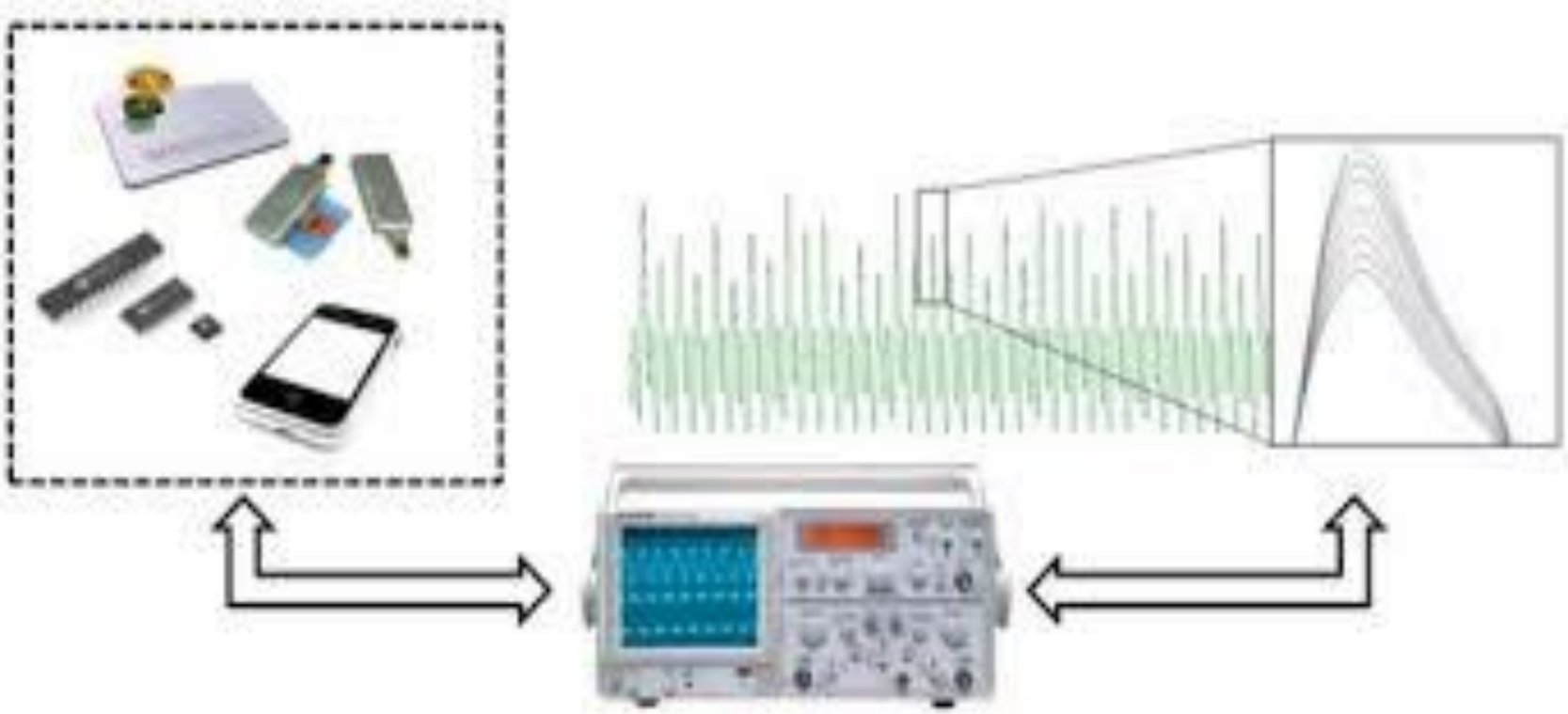
5 API SECURITY INCIDENTS

FROM 2021



5. Side-channel vulnerabilities

- **Side-channel vulnerabilities** exploit the physical and unintended byproducts of a program's execution, such as power consumption, electromagnetic radiation, or execution time, to gain unauthorized access to sensitive information.
- These vulnerabilities arise because the real-world execution of programs involves physical processes that aren't captured by abstract models of program execution.
- A **side channel** is an unintended information channel that leaks data based on these physical effects, which are abstracted away by the higher-level models.



Types of Side-channels:

- **Physical Side-channels:** Require physical access to the hardware, exploiting factors like power consumption or electromagnetic emissions.
- **Software-based Side-channels:** Can be exploited by software running on the same hardware as the target, such as by observing micro-architectural states like cache memory.

Significance

- A **covert channel** is a type of side-channel where the attacker deliberately uses the channel to exfiltrate information from a system. The attacker controls both the information sender and the receiver.
- Side-channel attacks are particularly significant in **cryptography**. The gap between the theoretical design of cryptographic algorithms and their physical implementation can be exploited.

Fault injection attacks

- While side-channels mostly pose a **confidentiality threat** (leaking information), they can also pose an **integrity threat** if an attacker can manipulate the system's state using side-channel techniques. These attacks are known as **fault injection attacks.**
- **Physical Fault Injection:** Techniques like voltage glitching, extreme temperatures, or electromagnetic radiation can induce faults in a system.
- **Software-based Fault Injection:** This involves driving hardware components outside their specification limits using software, which can induce faults.

PREVENTION OF VULNERABILITIES

- Language and API Design
  - Eradication by Design -incorporating features like garbage collection and static checks.
  - Trapped vs. Untrapped Errors- Designing languages and APIs to avoid untrapped errors is a powerful method to prevent vulnerabilities.
- Memory Management Vulnerabilities
  - Programming languages can prevent memory management vulnerabilities by being **memory-safe**, meaning they do not allow untrapped memory errors. Languages like Java and C# achieve this through a combination of static and dynamic checks, while languages like Rust use a type system to manage memory without garbage collection, ensuring memory safety.

- **Structured Output Generation Vulnerabilities**
  - These vulnerabilities often arise when the structure of output is handled implicitly through string manipulation. Languages can prevent these by offering features that allow the explicit definition of output structures, ensuring that no untrapped errors occur with respect to the intended structure. Examples include type systems for XML data or LINQ **in C#,** which explicitly defines the structure of queries.

- **Race Condition Vulnerabilities:**
  - **Ownership Types** are a static typing discipline used in programming languages to manage and enforce memory safety and resource access, especially in concurrent programming. The concept involves assigning an "**owner**" to a resource, which controls how other parts of the program can access and modify that resource.
  - **Coding Guidelines-** Coding guidelines are sets of best practices and recommendations that guide developers in writing secure and maintainable code. For preventing race conditions, specific guidelines focus on how to handle concurrency, shared resources, and synchronization.

- **Side Channel Vulnerabilities:**
  - **Randomization Techniques** -Randomization can be a powerful tool in preventing side channel attacks by introducing uncertainty and variability in the way operations are performed, making it harder for attackers to infer useful information.
  - **Secure Development Practices-** Incorporating secure development practices into the software development lifecycle helps prevent side-channel vulnerabilities by promoting awareness and adherence to security principles.
  - **Isolation and Compartmentalization-** Isolating sensitive operations and data from less secure parts of the system can prevent attackers from leveraging side channels.

Detection of Vulnerabilities and Mitigation

Detection of Vulnerabilities

- **Static Detection**: Involves analyzing code to identify potential vulnerabilities without executing it.
- **Dynamic Detection**: Involves monitoring code execution to identify vulnerabilities during runtime.
- **Heuristic and Sound Static Detection**: Heuristic focuses on compliance with best practices, while sound detection aims to be thorough and precise.

Memory management vulnerabilities

- **Detection of Vulnerabilities:**
  - **Static Analysis Tools**: These tools **analyze source code** without executing it, identifying potential vulnerabilities such as buffer overflows or use-after-free errors.
  - **Dynamic Analysis Tools**: These tools **monitor a program's behavior** during runtime to detect memory management errors. Example tools include Valgrind and AddressSanitizer.
  - **Fuzz Testing**: This involves providing a program with **random inputs** to uncover vulnerabilities.

- **Mitigating Exploitation of Memory management vulnerabilities:**

  - **Stack Canaries**: These are **special values placed on the stack** that help detect buffer overflows. If the **canary value changes, the program can take action before further damage occurs.**
  - **Address Space Layout Randomization (ASLR)**: **Randomizes the memory address space** of a program, making it more difficult for an attacker to predict the location of specific code and data segments.
  - **Control Flow Integrity (CFI)**: Enforces **strict controls over program flow**, preventing an attacker from diverting the program's execution path to malicious code.
  - **NX (No-eXecute):**NX, or **No-eXecute**, is a security feature that marks certain areas of memory as non-executable. This means that **even if an attacker manages to inject malicious code into these regions, the code cannot be executed.**
  - **Sandboxing**: By **isolating an application** from critical system files and user data, sandboxing ensures that any memory management vulnerability exploited within the application cannot be used to access or exfiltrate sensitive information.

Structured Output Generation Vulnerabilities

Detection Techniques

- **Static Analysis:**
  - **Code Scanners**: Use static analysis tools to **scan source code for patterns** that might lead to structured output generation vulnerabilities, such as improper handling of user input.
  - **Manual Code Review**: Perform **manual code reviews** focusing on sections of the code that generate structured output, ensuring that input validation and output encoding are properly implemented.
- **Dynamic Analysis:**
  - **Fuzz Testing**: Employ fuzz testing techniques to provide **random, unexpected, or malformed inputs** to the application and observe how it handles the output generation.
  - **Penetration Testing**: Conduct penetration tests to **simulate real-world attacks** that exploit structured output generation vulnerabilities, such as XSS or SQL injection.
- **Runtime Monitoring:**
  - **Intrusion Detection Systems (IDS):** Deploy IDS to monitor the application's runtime environment for signs of exploitation attempts, such as unusual SQL queries or suspicious HTML content being served to users.

Mitigation Techniques

- **Patching and Updates:**
  - **Regular Updates**: Ensure that the application and all its components (libraries, frameworks, etc.) are **regularly updated to the latest versions**, as these often include patches for known vulnerabilities.
  - **Security Patches:** Apply security **patches promptly to fix vulnerabilities** that could be exploited in structured output generation.
- **Runtime Protections:**
  - **Content Security Policy (CSP):** Implement **CSP headers** to **restrict the sources from which scripts can be loaded**, mitigating the impact of XSS vulnerabilities.
  - **Escaping and Canonicalization**: Ensure all output is **properly escaped and canonicalized before being rendered** to the user, especially when dealing with structured outputs like HTML, JSON, or XML.
- **Response Planning:**
  - **Incident Response:** Develop an incident response plan that **includes steps to take if a structured output generation vulnerability is exploited**. This should include containment, eradication, and recovery phases.
  - **Security Awareness Training**: Train developers and security teams on the importance of structured output generation vulnerabilities and the best practices for preventing, detecting, and mitigating them

Race Condition Vulnerabilities

Detection Techniques

- Effective detection typically involves a combination of static and dynamic analysis techniques.
- **Static Detection Techniques**
  - Static detection involves **analyzing an application's source code, bytecode, or binary code without executing it**. The goal is to identify potential race conditions by examining the code structure, data flow, and synchronization mechanisms.
- **Dynamic Detection Techniques**
  - Dynamic detection involves **analyzing the behavior of an application during runtime to identify race conditions**. This method monitors the actual execution of the code to detect race conditions as they occur.

Mitigating Exploitation of Race Condition Vulnerabilities

- **Sandboxing** is a security mechanism used to **run applications in a restricted environment**, limiting the potential damage caused by vulnerabilities, including race conditions. While sandboxing does not directly prevent race conditions, it plays a crucial role in mitigating their exploitation by containing the impact of any vulnerabilities that may arise.
- Sandboxing isolates different processes or threads from one another, limiting their access to shared resources. This isolation helps prevent a compromised process from exploiting race conditions in other parts of the system.
- Example: In **web browsers, each tab often runs in its own sandboxed process**, preventing a race condition in one tab from affecting others.

API Vulnerabilities

Detection Techniques for API Vulnerabilities

- **Runtime Checking of Pre- and Post-Conditions**:
  - Runtime Checking involves **validating certain conditions during the execution of an API**. Pre-conditions and post-conditions are specific assertions that must hold true before and after an API function or method is executed. This technique ensures that the API behaves as expected and helps detect anomalies that could indicate vulnerabilities.
- **Static Contract Verification:**
  - Static Contract Verification is a technique used to **analyze the code of an API without executing it**. This involves verifying that the code adheres to specified **contracts**, such as **API specifications, coding standards, and security policies**. This technique helps identify potential vulnerabilities at the code level before the API is deployed.

Mitigation Techniques for API Vulnerabilities

- Mitigation techniques aim to reduce the risk and impact of vulnerabilities in APIs by implementing protective measures.

- **Compartmentalization**
  - Compartmentalization involves **dividing an API's architecture into isolated components** or segments **to limit the spread of an attack** or the impact of a security breach. This approach is based on the principle of least privilege and defense-in-depth, ensuring that even if one part of the system is compromised, the damage is contained.

- **Principle of Least Privilege**
  - The **Principle of Least Privilege(PoLP)** is a security concept where users, systems, and processes are granted the minimum level of access necessary to perform their functions. This principle limits the potential damage if a vulnerability is exploited.

- **Defense-in-Depth:** Employ multiple layers of security controls to provide comprehensive protection.

Side channel vulnerabilities

- **Detection Techniques for Side Channel Vulnerabilities:**
- **Static Detection**
  - Detecting side channel vulnerabilities is a critical step in securing software systems, especially those that handle sensitive information.
  - Static detection techniques analyze the code without executing it, helping to identify potential sides channel vulnerabilities during the development phase

- **Mitigation Techniques for Side Channel Vulnerabilities:**

- **Isolation -** Isolation is a key mitigation technique that helps protect sensitive operations and data from being exposed to side channel attacks.

- Below are key isolation techniques used to mitigate side-channel vulnerabilities.

- **Hardware-Based Isolation**- Hardware-based isolation involves using specialized hardware features or components to **separate sensitive operations and data from less secure parts of the system.** This type of isolation is particularly effective against side-channel attacks because it can **prevent unauthorized access to critical resources like cryptographic keys or memory.**

- **Software-Based Isolation-** Software-based isolation involves **structuring the software** in a way that separates sensitive data and operations from less secure code. This method is crucial in environments where hardware-based isolation is not feasible or sufficient.

- **Cache Isolation Techniques -**Cache timing attacks are a common form of side channel attacks. **Cache isolation techniques aim to prevent sensitive operations from leaving exploitable traces in the CPU cache**, which could be used by attackers to infer information.

Cont…

- **Mitigation Techniques for Side Channel Vulnerabilities:**

- **Network and Communication Isolation-** Isolating sensitive data and operations in terms of network and communication channels is also essential to **prevent side channel vulnerabilities that might arise through network-based attacks.**

- **Compartmentalization and Least Privilege -**Compartmentalization involves dividing a system into distinct sections, each with its own level of security and access controls. This approach limits the spread of side channel attacks by ensuring that **each compartment has minimal access to sensitive data.**

Security in the Design of Operating Systems

Security in the Design of Operating Systems

- Design of OS
    1. Simplicity of Design
    2. Layered Design
    3. Kernelized Design
- Reference Monitor
- Correctness and Completeness
- Secure Design Principles
- Trusted Systems
- Trusted System Functions

1. Simplicity of Design

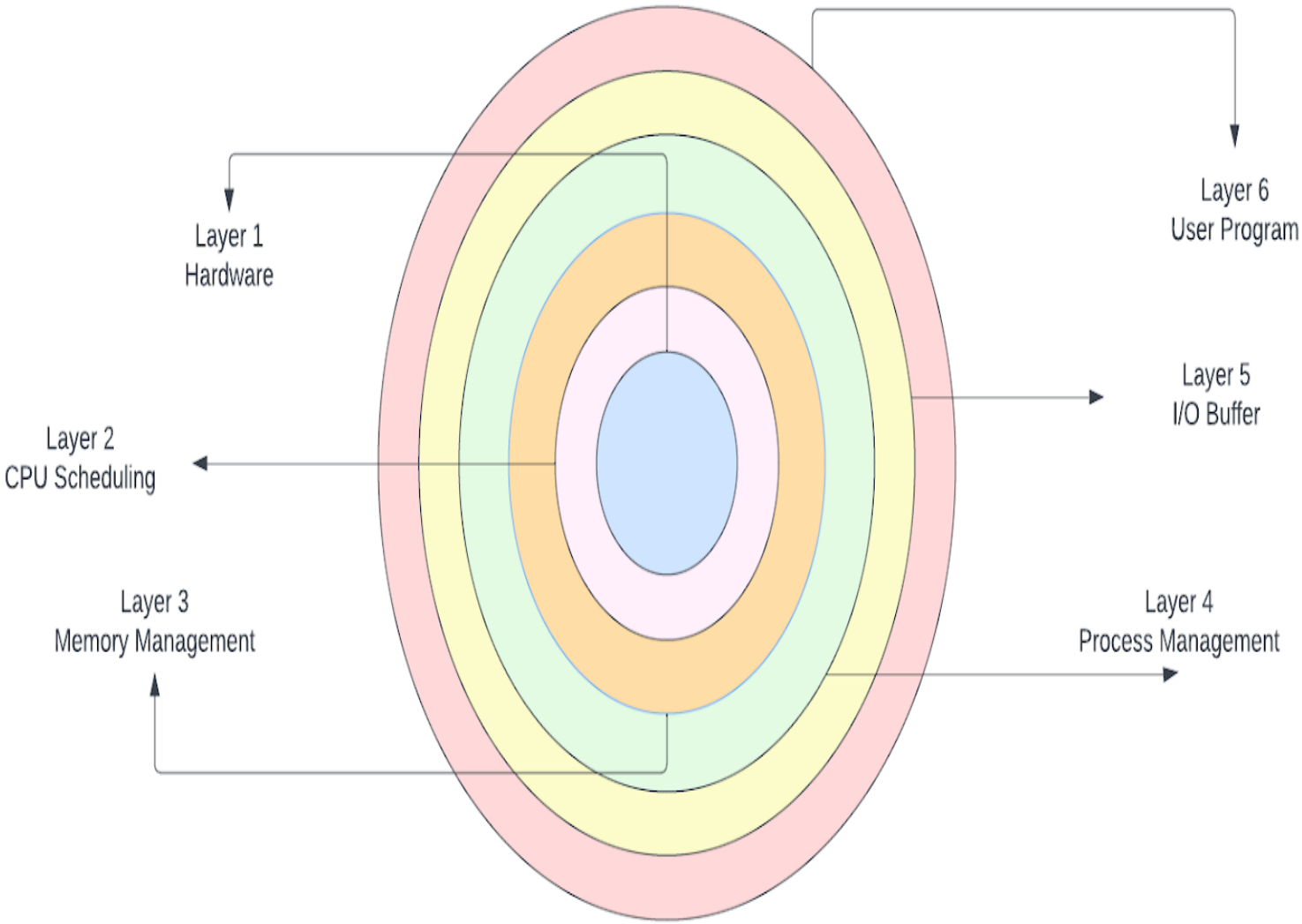
- Designing an operating system (OS) is inherently challenging due to its multiple responsibilities, including handling interruptions, context switches, and minimizing overhead to maintain user performance.
- The complexity of the design increases when security enforcement is added to the OS's responsibilities.
- Despite these challenges, integrating security from the beginning of the design process is crucial.
- Good software engineering practices emphasize the importance of designing security into the OS from the outset, rather than trying to add it later.

2. Layered Design

- Operating systems (OS) are composed of **multiple layers**, typically including **hardware, the kernel, the OS itself,** and the user layer, each potentially containing sublayers.

- For example, the kernel can have multiple distinct layers, and the **user level** may include **quasi-system programs like database managers or graphical user interfaces** that introduce additional layers of security.

- **Layered Trust-** The layered structure of **a secure OS can be visualized as concentric circles**, with the **most sensitive operations at the innermost layers.** Alternatively, it can be viewed as a building where the most critical tasks are on the lower floors. The trustworthiness and access rights of a process are determined by its proximity to the center or the bottom of this structure—**more trusted processes are closer to these central points.**


Layer 1
Hardware
Layer 2
CPU Scheduling
Layer 3
Memory Management
Layer 4
Process Management
Layer 5
I/O Buffer
Layer 6
User Program

- Layering inherently **involves separation, particularly logical (software-based) separation,** which is crucial in enforcing access controls.
- The inner or **lower layers of the OS must manage the accesses of outer** or higher layers to maintain this separation.
- This design strategy, known as **encapsulation, ensures that each layer depends on more central layers for services and provides functionality to outer layers.**
- This allows for "**peeling off**" layers while maintaining a functional system, though with reduced capabilities.

In a **conventional, nonhierarchically designed system** (shown in Table), **any problem—hardware failure, software flaw, or unexpected condition, even in a supposedly irrelevant nonsecurity portion**—can **cause disaster** because the effect of the problem is unbounded and because the system's design means that we cannot be confident that any given function has no (indirect) security effect.

Level	Functions	Risk
all	Noncritical functions	Disaster possible
all	Less critical functions	Disaster possible
all	More critical functions	Disaster possible

Hierarchical structuring has two benefits:

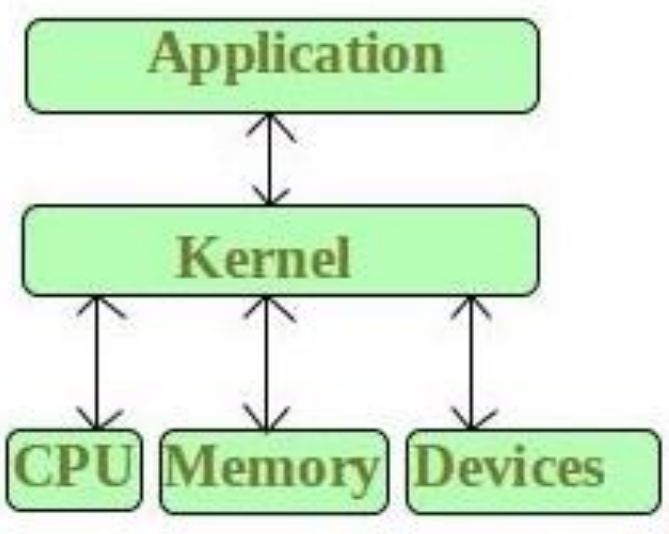
- Hierarchical structuring permits the **identification of the most critical parts**, which can then be analyzed intensely for correctness, so the number of problems should be smaller.

- **Isolation limits the effects of problems** to the hierarchical levels at and above the point of the problem, so the harmful effects of many problems should be confined.

Level	Functions	Risk
2	Noncritical functions	Few disasters likely from noncritical software
1	Less critical functions	Some failures possible from less critical functions, but because of separation, impact limited
0	More critical functions	Disasters possible, but unlikely if system simple enough for more critical functions to be analyzed extensively

3. Kernelized Design

- The **kernel is the core component of an operating system** responsible for performing the most **fundamental functions**, such as **synchronization, interprocess communication, message passing, and interrupt handling. It** is sometimes referred to as the nucleus or core of the OS.
- In a kernelized design, the **operating system is built around this central kernel,** a concept explored by Butler Lampson, Howard Sturgis, Gerald Popek, and Charles Kline in the 1970s.



Security kernel

- A **security kernel is a specialized component** within the operating system kernel that enforces the security mechanisms across the entire system.
- It **manages the security interfaces between the hardware, the OS, and other components of the computing system**. Typically, the **security kernel is integrated within the main OS kernel,** ensuring that security functions are tightly coupled with the OS's core operations.
- This **approach to design ensures that security is deeply embedded** in the system, providing a robust foundation for protecting the OS from potential threats.

There are several good design reasons why security functions may be isolated in a security kernel.

- **Coverage**. Every **access to a protected object must pass through the security kernel**. In a system designed in this way, the operating system can use the security kernel to ensure that every access is checked.
- **Separation**. **Isolating security mechanisms** both from the rest of the operating system and from the user space makes it **easier to protect** those mechanisms from penetration by the operating system or the users.
- **Unity**. **All security functions are performed by a single set of code**, so it is **easier to trace the cause of any problems** that arise with these functions.
- **Modifiability**. Changes to the security mechanisms **are easier to make and easier to test.** And because of unity, the effects of changes are localized so interfaces are easier to understand and control.
- **Compactness**. Because it performs **only security functions**, the security kernel is **likely to be relatively small**.
- **Verifiability**. Being **relatively** small, **the security kernel can be analyzed rigorously.** For example, formal methods can be used to ensure that all security situations (such as states and state changes) have been covered by the design.

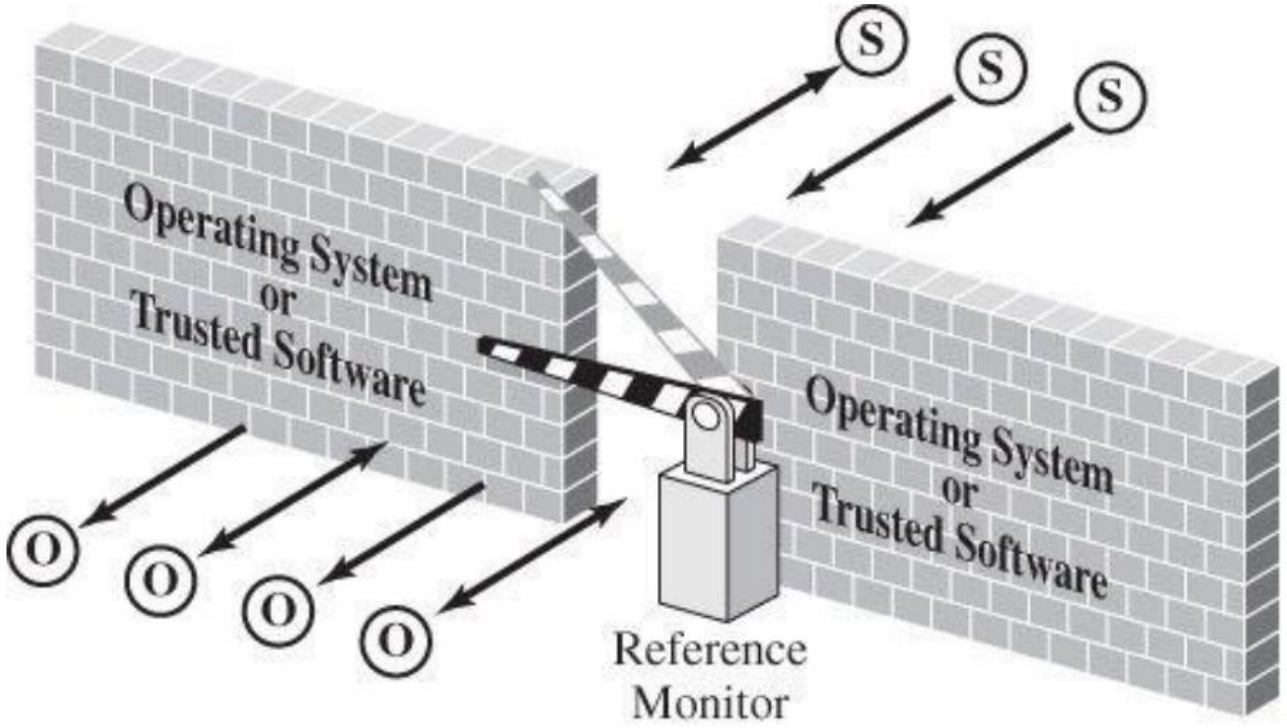
Disadvantage of implementing Security Kernel

- On the other hand, implementing a security kernel **may degrade system performance** because **the kernel adds yet another layer of interface between user programs and operating system resources**.
- Moreover, the **presence of a kernel does not guarantee that it contains all security functions** or that it has been implemented correctly.
- The **design and usefulness of a security kernel depend somewhat on the overall approach to the operating system's design**.
- There are many **design choices**, each of which falls into one of **two types**: Either **the security kernel is designed as an addition to the operating system** or **it is the basis of the entire operating system**.

Reference Monitor

- The most important **part of a security kernel is the reference monitor**, the **portion that controls access to objects**.
- The **reference monitor separates subjects and objects**, enforcing that a **subject can access only those objects expressly allowed by security policy**.
- A reference monitor is **not necessarily a single piece of code**; rather, it is the **collection of access controls for devices, files, memory, interprocess communication, and other kinds of objects.**
- As shown in Figure, a **reference monitor acts like a brick wall around the operating system or trusted software** to mediate accesses by subjects (S) to objects (O).

Reference Monitor



- A reference monitor must be
  - **tamperproof**, that is, impossible to weaken or disable.
  - **un-bypassable,** that is, always invoked when access to any object is required.
  - **analyzable,** that is, small enough to be subjected to analysis and testing, the completeness of which can be ensured.
- The reference monitor is **not the only security mechanism** of a trusted operating system.
- **Other parts of the security suite include auditing and identification and authentication processing, as well as setting enforcement parameters, such as who are allowable subjects and what objects they are allowed to access**.
- These **other security parts interact with the reference monitor, receiving data from the reference monitor or providing it with the data** it needs to operate.

Correctness and Completeness

- Security is a fundamental aspect that must permeate the design and structure of operating systems (OS). **Achieving security requires both correctness and completeness:**

- **Correctness**:
  - Given that an OS controls interactions between subjects (users, processes) and objects (files, resources), **security must be integrated into every design element**. This includes **clearly defining which objects are protected**, **how they are protected, and which subjects have access at various levels.**
  - A **clear mapping between security requirements and design is essential**, ensuring that developers understand how security is implemented. **After the design is completed, it must be checked to confirm it enforces the intended security measures**.
  - This can be done through **formal reviews, simulations, and testing**, with a mapping from requirements to design and tests to **verify that all security aspects work correctly.**

- **Completeness**:
  - Security **features must be implemented wherever necessary within the OS**. Although this might seem obvious, not all developers focus on security during design and coding.
  - **Retrofitting security into an OS that wasn't designed** with adequate security is extremely difficult and often **leads to compromises**.
  - **Last-minute security additions, made under time pressure, tend to be incomplete and less effective.**
- **Security enforcement must be correct and complete.**

Secure Design Principles

- Good design principles are always good for security. But several important design principles are particular to security and essential for building a solid, trusted operating system.
- These principles, are articulated well by Jerome Saltzer and Michael Schroeder.
  - least privilege
  - economy of mechanism
  - open design
  - complete mediation
  - permission based
  - separation of privilege
  - least common mechanism
  - ease of use

- Although design principles were suggested several decades ago, they are as accurate now as they were when originally written. The principles have been used repeatedly and successfully in the design and implementation of numerous trusted systems.

Trusted Systems

- Trusted systems are designed to counter the problem of malicious software by providing **a reliable foundation that users can trust** to perform specific activities securely and in line with their expectations.
- Unlike the common usage of the term **"trust," which may imply hope without assurance**, **trusted systems offer convincing evidence that they will protect against malicious code and operate as intended**.
- **This trust is not based on mere hope but on demonstrated and justified reliability.**
- **Trusted system: one with evidence to substantiate the claim it implements some function or policy.**

- **History of Trusted Systems**

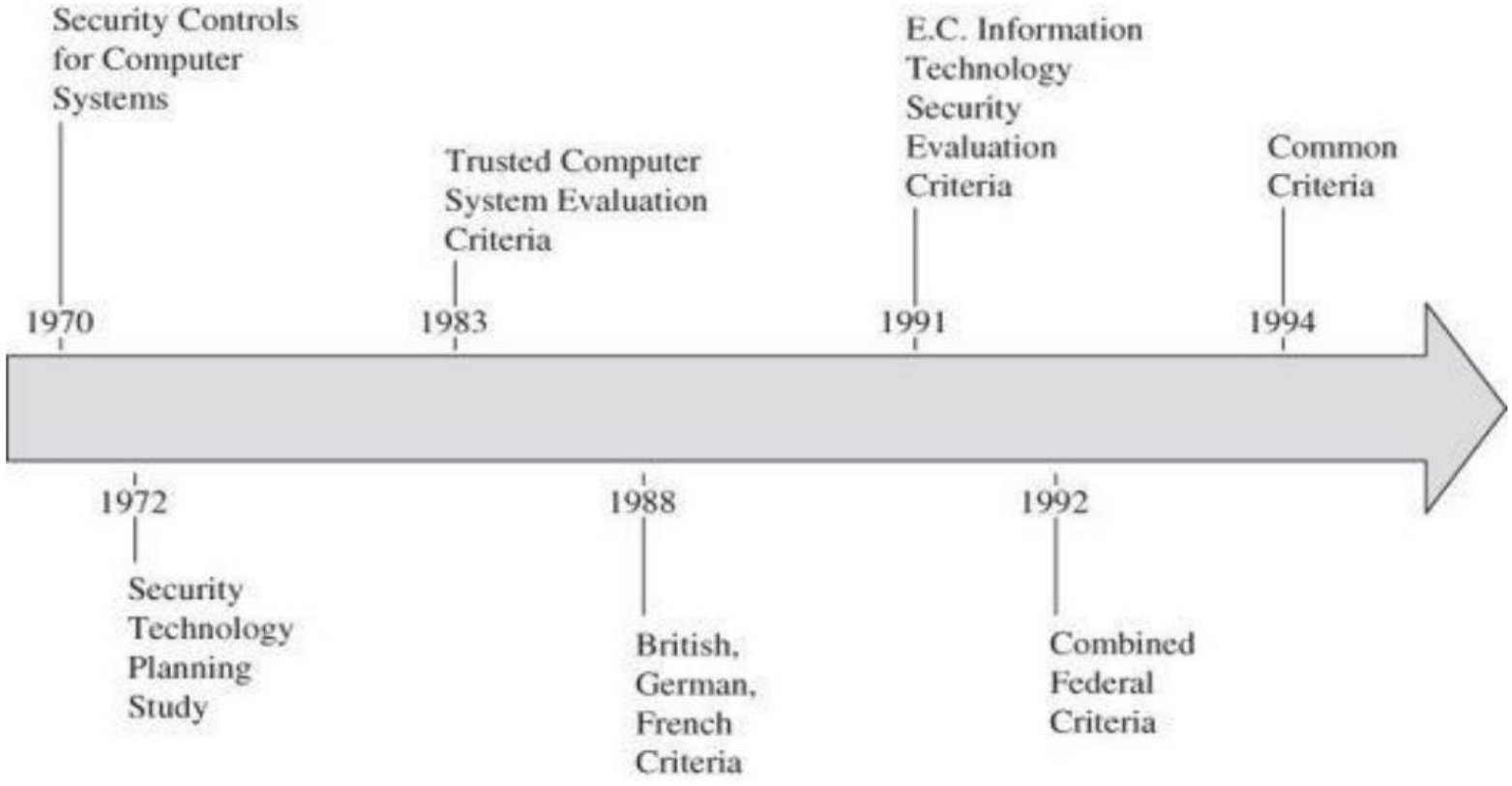
- The history of trusted systems in computer security dates back to the **1960s**, during the **early days of multiuser, shared computing**. The need for secure systems became evident, leading to the formation of **a committee chaired by Willis Ware**, which highlighted the necessity for stronger security enforcement.

- In the **1970s**, research and real-world systems demonstrated the feasibility and importance of trusted systems. This effort culminated in a report from **James Anderson's committee** in 1972, which recommended **the development of processes for creating more trustworthy systems.**

- **In the late 1970s**, the U.S. Department of Defense began drafting the **Trusted Computer System Evaluation Criteria (TCSEC), commonly known as the Orange Book** due to its cover color. This document **outlined the functionality, design principles, and evaluation methodologies for trusted computer systems**. Although it was later extended to network components and database management systems, the **TCSEC did not achieve widespread adoption as intended.**

- **Despite this, the TCSEC laid the foundation for further advancements** in the development and evaluation of trusted systems, influencing the evolution of computer security practices.

Trusted Systems



Trusted System Functions

Trusted systems contain certain **functions** to ensure security. **In this section we look over several aspects :**

1. **Trusted Computing Base**
2. **Key elements of the TCB**
3. **Four basic interactions**
4. **TCB Design**
5. **TCB Implementation**
6. **Secure Startup**
7. **Trusted Path**
8. **Object Reuse**
9. **Audit**

The Trusted Computing Base

- **The Trusted Computing Base, or TCB**, is the name we give to **everything** in the trusted operating system **necessary to enforce the security policy.**
- Alternatively, we say that the **TCB consists of the parts of the trusted operating system** on which we depend for the correct policy enforcement.
- The **TCB acts as a protective shell**, ensuring that even if malicious programmers control non-TCB parts of the system, **they cannot compromise security.** The security of the entire system hinges on the TCB, which must be both **correct and complete.**
- To design an effective TCB, it is **crucial to focus on the boundary between TCB and non-TCB elements** and ensure that the TCB operates flawlessly.

Key elements of the TCB include

- Hardware: Processors, memory, registers, clocks, and I/O devices.
- Processes: Mechanisms to separate and protect security-critical processes.
- Primitive Files: Security access control databases, identification, and authentication data.
- Protected Memory: Ensures that the reference monitor is shielded from tampering.
- Inter-process Communication: Securely allows different parts of the TCB to pass data and activate functions, such as the reference monitor invoking an audit routine.
- These components collectively ensure that the TCB can reliably enforce the system's security policy.

Four basic interactions

The Trusted Computing Base (TCB) **is responsible for maintaining the secrecy and integrity** of each domain within a system by monitoring four basic interactions:

1. **Process Activation:** In multiprogramming environments, processes frequently activate and deactivate. Switching from one process to another requires changing security-sensitive information like registers, relocation maps, file access lists, and process status data.

2. **Execution Domain Switching**: Processes in one domain often invoke processes in other domains to access more or less sensitive data or services, requiring careful monitoring by the TCB.

3. **Memory Protection:** Since each domain includes code and data stored in memory, the TCB must oversee memory references to ensure each domain's secrecy and integrity.

4. **I/O Operations**: I/O operations often involve software that connects a user program in an outer domain to an I/O device in the innermost hardware domain, potentially crossing all domains and requiring TCB oversight.

TCB Design

- The TCB's design divides the operating system into **TCB and non-TCB parts.**
- This separation ensures that **all security-relevant code is located in a protected**, logically distinct part, safeguarding it from interference or compromise by non-TCB code.
- This protected state of the TCB allows **developers to modify non-TCB components,** such as utilities or device drivers, **without affecting the system's security enforcement.**
- The conscious structuring of the TCB and non-TCB elements provides **self-protection and independence, simplifying the evaluation** of a trusted operating system's security.
- **Non-TCB code does not need to be considered during security evaluations**, focusing efforts solely on the TCB.

TCB Design

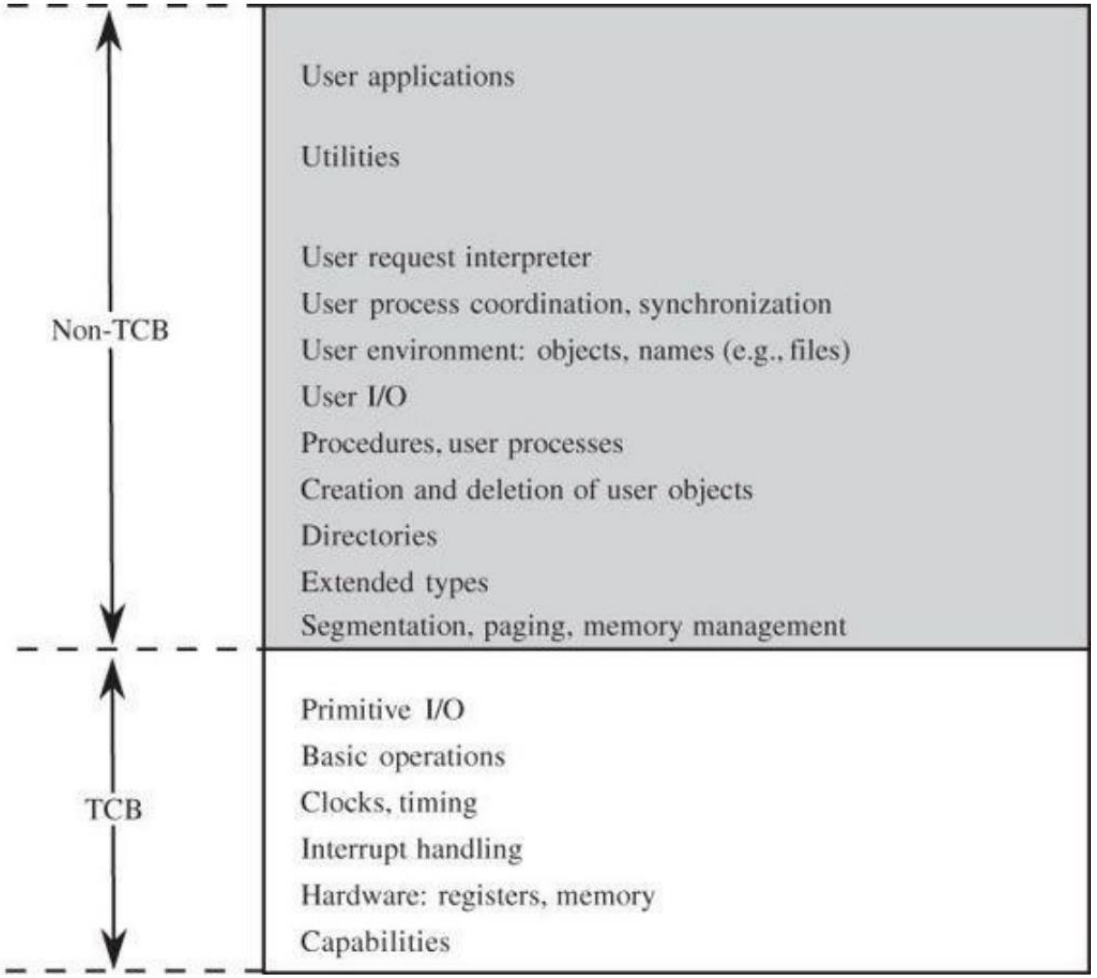


FIGURE 5-17 System Separated into TCB and Non-TCB Sections

TCB Implementation

- **The Trusted Computing Base (TCB)** involves security-related activities throughout various system functions, such as **memory access, I/O operations, file and program access, user management, thread creation, and interprocess communication.**

1. **In modular operating systems**, these activities are typically **handled** by **independent modules**, each performing both security-related and other functions.

2. **Consolidating all these security functions into a unified TCB could compromise the system's modularity and result in a TCB that is too large and complex to analyze effectively.**

3. However, d**esigners might choose to separate the security functions from the rest of the system to create a security kernel,** focusing on maintaining the integrity and manageability of the TCB while addressing security needs

a designer may decide to separate the security functions of an existing operating system, creating a security kernel.

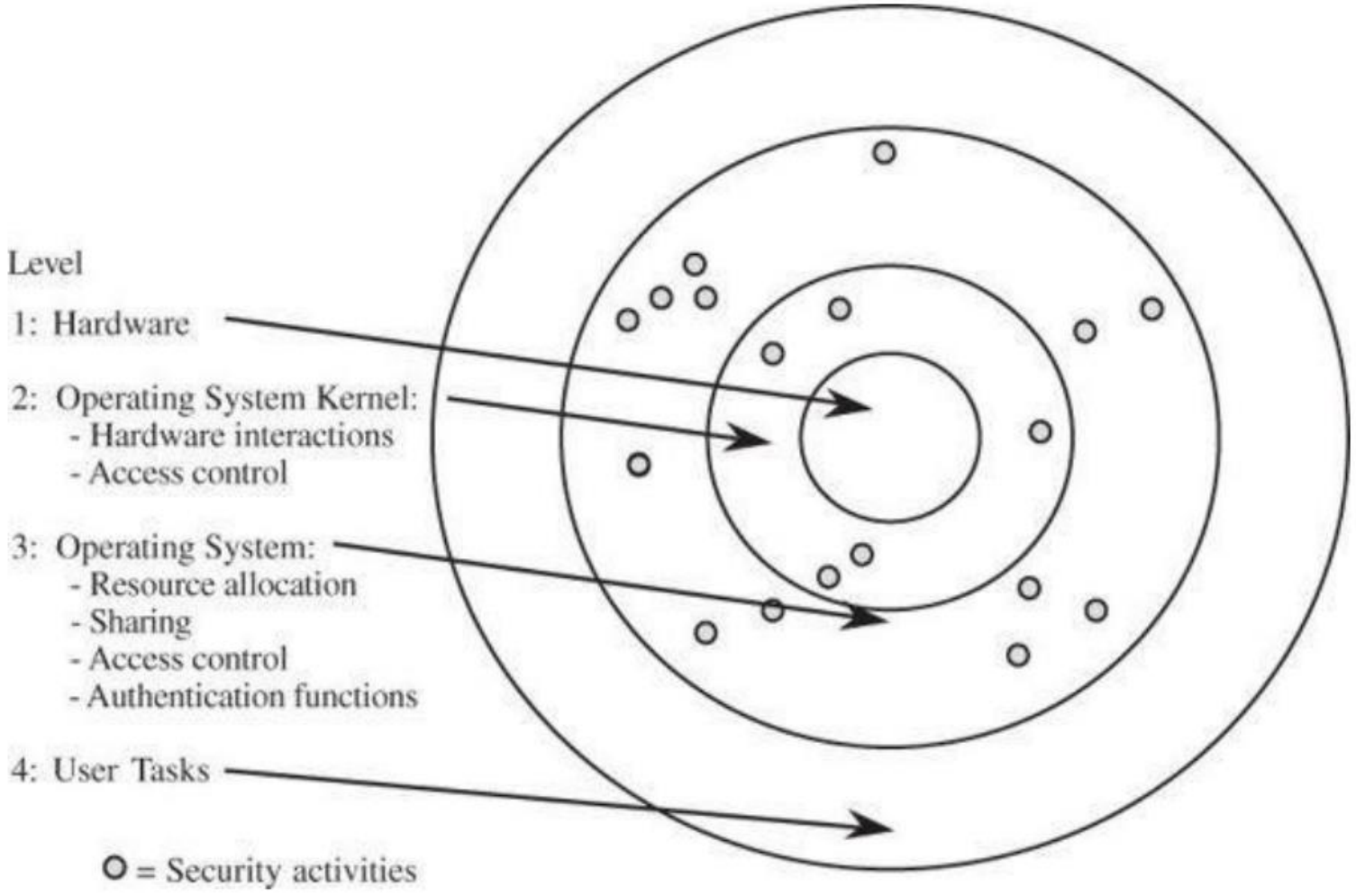


FIGURE 5-18 Security Kernel

- **A more sensible approach is to design the security kernel** first and then design the operating system around it.
- Initially, the security kernel **had around 20 modules** with **fewer than 1,000 lines of high-level code**. As development progressed, the **kernel's code expanded to about 10,000 lines**. In this design, the **security kernel acts as a thin, efficient layer directly above the hardware,** managing all security functions and monitoring hardware access.
- This **design allows the operating system to manage non-security functions,** leading to a partitioned system with at least **three execution domains**: the **security kernel, the operating system, and the user.**

Secure Startup

- Secure startup **addresses the vulnerabilities** present when a system is first powered on or restarted.
- During this phase, the **operating system's protection capabilities are limited**, and the system is particularly susceptible to threats since users are not yet active and network connections are not established.
- Trusted system designs **mandate that developers ensure all security functions are operational from** the start and that **no residual effects from previous sessions** impact the system.
- The goal is **to prevent malicious code from interfering** with or blocking security enforcement during the startup process.
- **Secure startup ensures no malicious code can block or interfere with security enforcement.**

Trusted Path

- The concept of a trusted path is essential for **securing the connection between a user and an operating system's internal identity system**.
- While **authentication methods verify user identity** to the operating system, **users also need assurance that they are interacting directly with the legitimate system**, not a fraudulent application.
- For example, in early Microsoft Windows, pressing **Control-Alt-Delete** would **trigger the login prompt**, **bypassing any fake login screens created by applications.**
- This **ensures that users are communicating directly with the operating system's authentication system**, providing a secure and verifiable method to enter credentials.
- **A trusted path precludes interference between a user and the security enforcement mechanisms of the operating system**.

Object Reuse

- Object reuse involves **reassigning freed resources**, such as disk space or memory, to new users or programs. However, **if not carefully managed, this can lead to security vulnerabilities.**
- When a file is created, the space for it may still contain data from a previous user. Although new data usually overwrites the old, **a malicious user could exploit this by accessing and reading the old data before writing their own**, **leading to unauthorized data disclosure**.
- To prevent such risks, known as object reuse, **operating systems must clear (overwrite) the space before it is reassigned.**
- This practice ensures that no sensitive data from previous users is accessible.
- Additionally, specific **threats like magnetic remanence**, where **remnants of old data may be recoverable**, further highlight the **need for thorough data sanitization by the operating system.**

Audit

- In trusted systems, **maintaining an audit log is crucial for tracking security-relevant changes** like new program installations or operating system modifications.
- This **log must be safeguarded against tampering**, modification, or deletion except by an authenticated security administrator.
- Additionally, the **audit log needs to be continuously active during system operation**. If the audit medium becomes full, such as when disk space is exhausted, the system should shut down to prevent any loss of audit data.