Security Development Cycle

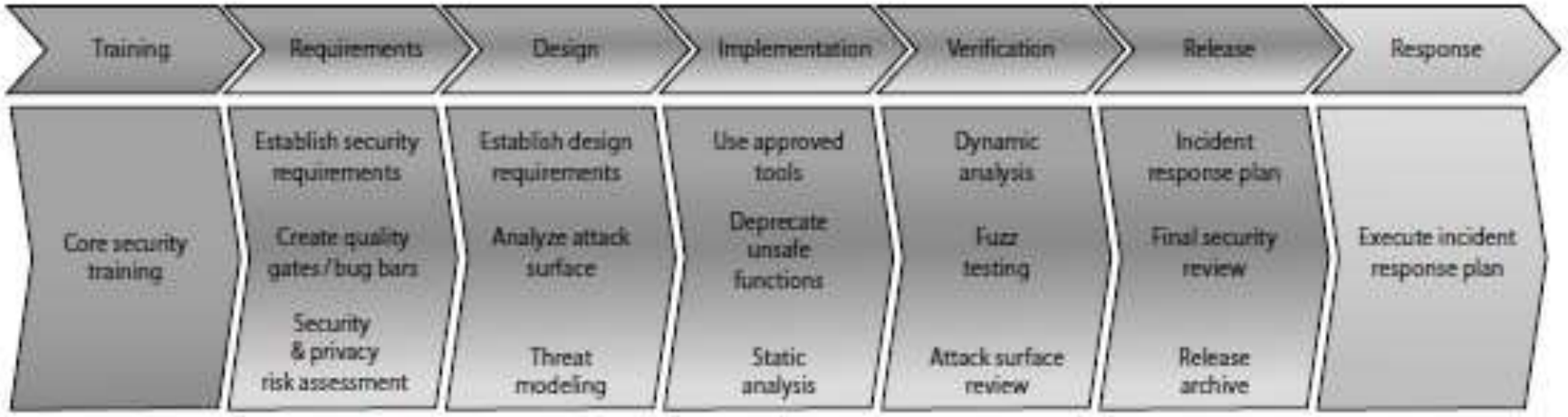

Figure 9.1 Security Development Lifecycle (© 2010 Microsoft Corporation. All rights reserved. Licensed under Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported.)

Security Development Cycle

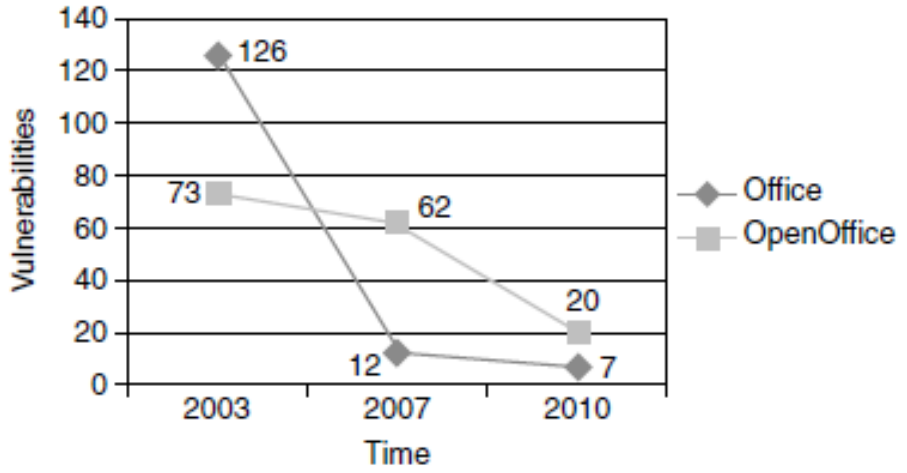

Figure 9.2 Vulnerabilities in Office versus OpenOffice (Source: [Kaminsky 2011])

Security Development Cycle

TSP Secure:-

- TSP-Secure is an extension of the Team Software Process (TSP) specifically designed to enhance the security of software applications by addressing common issues that lead to vulnerabilities.
- It tackles key challenges such as unclear security goals, undefined roles, insufficient planning, and overlooking security until late in the development cycle.
- By emphasizing proactive security planning, building self-directed teams, and managing quality throughout the development process, TSP-Secure ensures that security is integrated at every stage.
- Additionally, it promotes security awareness among developers through training, helping them recognize and address security risks early on. The approach builds on TSP's proven effectiveness in reducing defects, with a focus on producing secure, high-quality software.

Security Development Cycle

Planning and Tracking

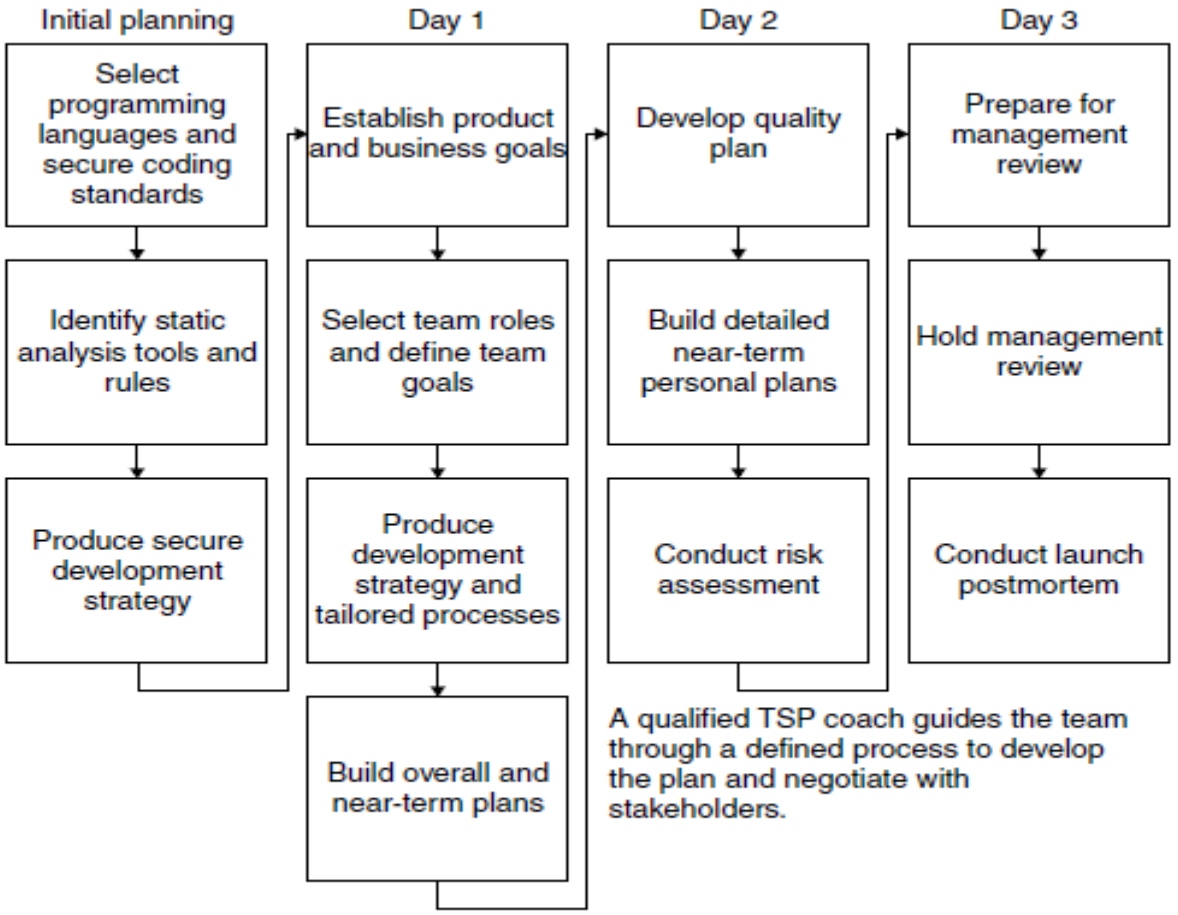

Figure 9.3 Secure launch

Security Development Cycle

Quality Management

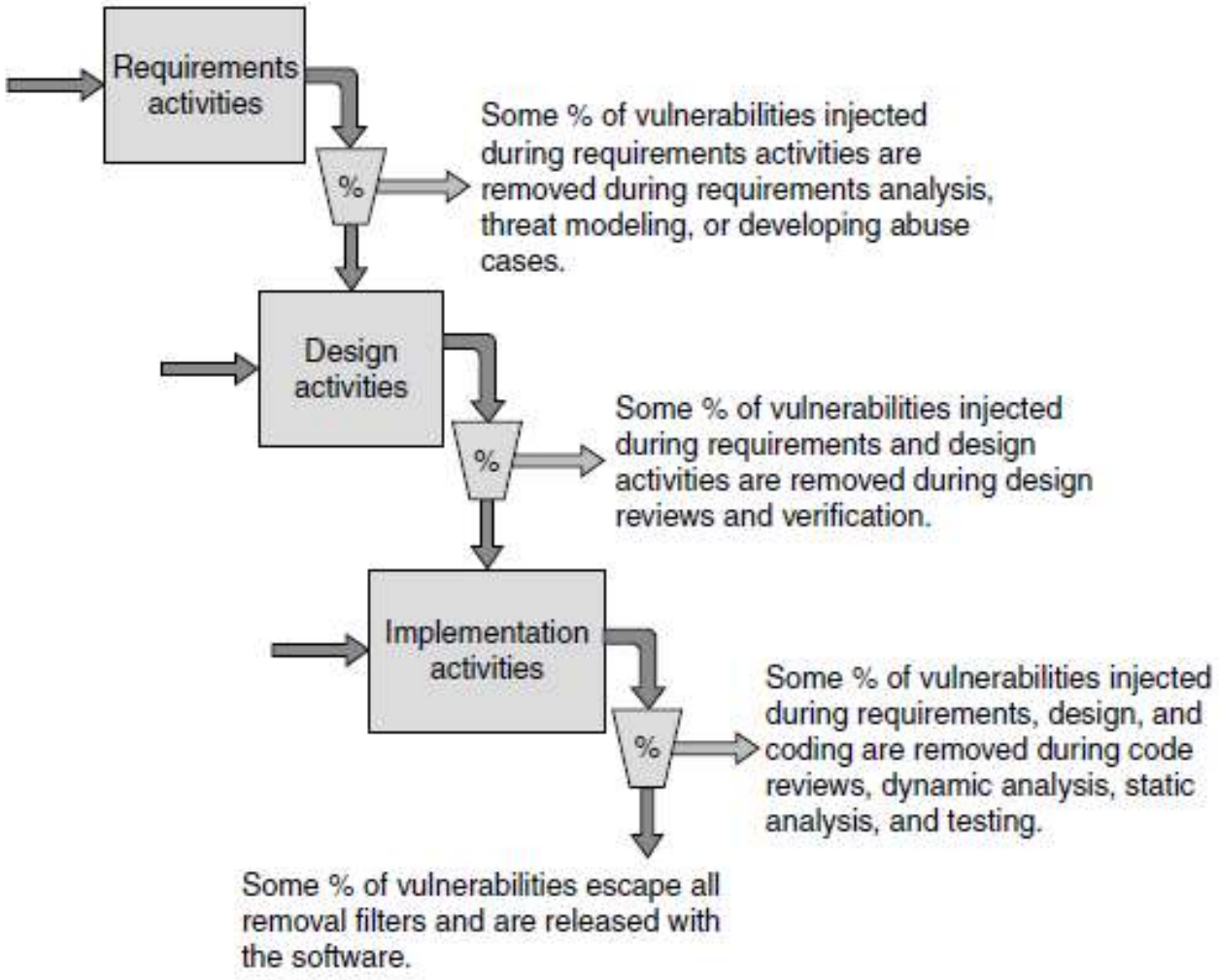

Figure 9.4 Filtering out vulnerabilities

Security Training

- Education plays a crucial role in addressing future cybersecurity challenges by integrating secure programming principles into curricula.
- In 2010, the National Science Foundation sponsored the Summit on Education in Secure Software (SESS) to guide this process, leading to recommendations like requiring computer security courses for all students and using innovative teaching methods.
- Universities like Carnegie Mellon offer specialized courses such as "Secure Programming" and "Secure Software Engineering," and there is a growing need for more secure coding education due to high demand for skilled developers.
- To meet this need, online courses and professional training programs, such as those by SEI and CERT, are being developed to scale and improve secure coding skills efficiently while ensuring high-quality learning outcomes.

Security Training

1. Require at least one computer security course for all college students:
   a. For CS students, focus on technical topics such as how to apply the principles of secure design to a variety of applications.
   b. For non-CS students, focus on raising awareness of basic ideas of computer security.
2. Use innovative teaching methods to strengthen the foundation of computer security knowledge across a variety of student constituencies.

Requirements

Security Coding Practices

- ISO/IEC 9899:1999, *Programming Languages—C, Second Edition* [ISO/IEC 1999]

- Technical Corrigenda TC1, TC2, and TC3
- ISO/IEC TR 24731-1, *Extensions to the C Library, Part I: Bounds-Checking Interfaces* [ISO/IEC 2007]
- ISO/IEC TR 24731-2, *Extensions to the C Library, Part II: Dynamic Allocation Functions* [ISO/IEC TR 24731-2:2010]

UNIT-5—Recommended Practices

Requirements

Security Quality Requirements Engineering

- **Agree on Definitions** : It is a prerequisite to requirement engineering. Engineering body of knowledge provide a wide range of definitions.
- **Identify Assets and Security Goals**: Assets to be protected and their associated security goals must be identified and prioritized for the organization and also for the information system to be developed.
- **Develop Artifacts:**
- Perform Risk Assessment
- Select Elicitation techniques
- Elicit Security Requirements
- Categorize requirements
- Prioritize Requirements
- Requirement Inspection

Requirements

Security Quality Requirements Engineering

Develop Artifacts: A lack of documentation including a concept of operations, succinctly stated project goals, documented normal usage and threat scenarios, misuse cases, and other documents needed to support requirements definition can lead to confusion and miscommunication.

- **Perform Risk Assessment:** There are a number of risk assessment methods to select from based on the needs of the organization. The artifacts from step 3 provide the input to the risk assessment process. Threat modeling can also provide significant support to risk assessment. The outcomes of the risk assessment can help identify high priority security exposures.

- **Select Elicitation techniques**: Selecting an elicitation technique is important when there are several classes of stakeholders. A more formal elicitation technique, such as structured interviews, can be effective when there are stakeholders with different cultural backgrounds. In other cases, elicitation may simply consist of sitting down with a primary stakeholder to try to understand his or her security requirements

UNIT-5—Recommended Practices

Requirements

Security Quality Requirements Engineering

- Elicit Security Requirements
- Categorize requirements
- Prioritize Requirements
- Requirement Inspection

Requirements

Use/Misuse Cases:

Table 9.2 Differences between Misuse Cases and Security Use Cases

	Misuse Cases	Security Use Cases
Usage	Analyze and specify security threats	Analyze and specify security requirements
Success criterion	Attacker succeeds	Application succeeds
Produced by	Security team	Security team
Used by	Security team	Requirements team
External actors	Attacker, user	User
Driven by	Asset vulnerability analysis Threat analysis	Misuse cases

Requirements

Use/Misuse Cases:

Table 9.3 Application-Specific Misuse Case

Misuse Case Name: Spoof Customer at ATM	
Summary:	
The misuser successfully makes the ATM believe he or she is a legitimate user. The misuser is consequently granted access to the ATM's customer services.	
Preconditions:	
1. The misuser has a legitimate user's valid means of identification and authentication, OR 2. The misuser has invalid means of identification and authentication but so similar to valid means that the ATM is unable to distinguish, OR 3. The ATM system is corrupted, accepting means of identification and authentication that would normally have been rejected.	
Misuser Interactions	**System Interactions**
Request access	
	Request identification and authentication
Misidentify and misauthenicate	
	Grant access
Postconditions:	
1. The misuser can use all the customer services available to the spoofed legitimate user, AND 2. In the system's log (if any), it will appear that the ATM was accessed by the legitimate user.	

UNIT-5—Recommended Practices

Design:

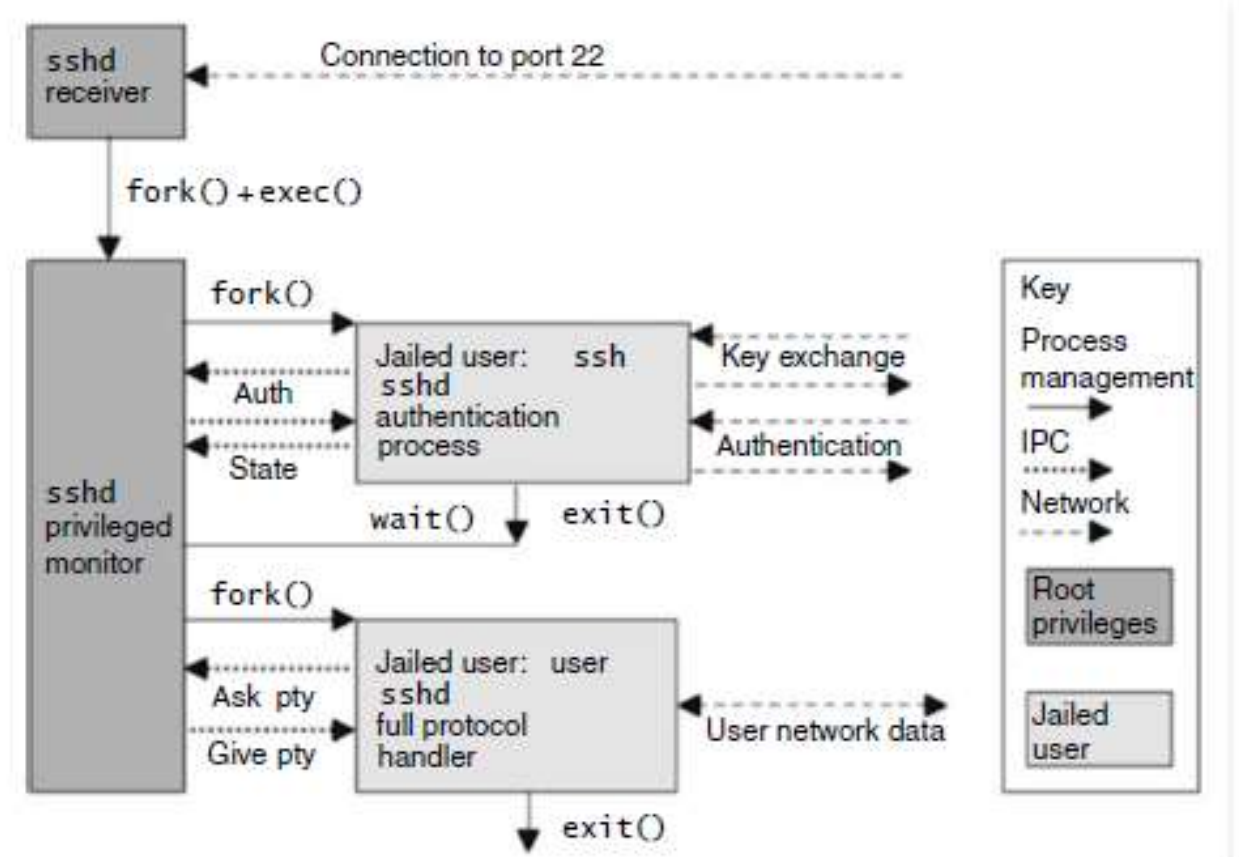

Figure 9.5 OpenSSH

UNIT-5—Recommended Practices

Design:

Secure Software Development Principles

1. Economy of Mechanism

Economy of mechanism emphasizes designing security systems to be simple and small, reducing the risk of errors in implementation and configuration. Simple designs are easier to verify, cost-effective, and more reliable, as complex systems increase the chances of mistakes and higher assurance efforts.

2. Fail Safe Defaults

Fail-safe defaults prioritize denying access by default and explicitly granting permission when conditions are met, making failures easier to detect and fix. If access is incorrectly allowed, it can go unnoticed, whereas incorrect denial is more easily identified and corrected. This approach aligns with the principle of ensuring security through default restrictions.

3. **Complete meditation**

Complete mediation ensures that every access request to a resource is thoroughly checked for proper authorization, with each request being verified for legitimacy. This prevents unauthorized access by consistently validating every action, rather than relying on previous checks.

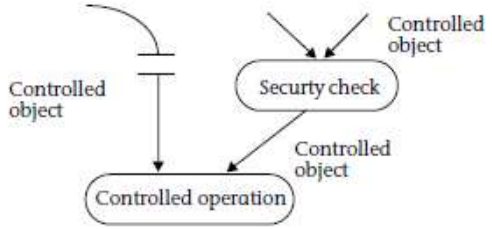

Figure 9.6 The complete mediation problem

UNIT-5—Recommended Practices

Design:

Secure Software Development Principles

Open design:
An open design in security ensures that systems remain secure even when their implementation is publicly reviewed. It avoids relying on obscurity, instead focusing on robust protection mechanisms that can be independently verified. Since attackers can inspect code or binaries, open designs allow for thorough testing and validation, ensuring the security of the system without depending on hidden details.

Separation of Privilege:

Separation of privilege enhances security by requiring multiple conditions to grant access, preventing a single point of failure. For example, two-factor authentication combines something you know (like a password) and something you have (like a security token), ensuring that both must be compromised for unauthorized access. This reduces risk, as losing or disclosing one factor alone doesn’t allow access. It's also related to the design of systems with subsystems having specific privileges, enabling a more detailed application of the principle of least privilege for better security.

UNIT-5—Recommended Practices

Design:

Secure Software Development Principles

Least Privileges:

The principle of least privilege emphasizes that programs and users should only be granted the minimum permissions necessary to perform their tasks. This reduces the risk of vulnerabilities being exploited, as any elevated privileges should be used for the shortest time possible. For example, a password-changing program should only allow a user to modify their own password and not have unrestricted access to the entire password database, which prevents potential abuse or unauthorized access.

Implementing least privilege involves granting only the essential permissions to system components, acquiring and discarding privileges as needed, and ensuring privileges are used early in the process and discarded after use. This approach minimizes the chance for attackers to gain elevated privileges, especially if the system allows privileges to be dynamically adjusted. However, the security model must also ensure that the user authorizes such privilege changes to prevent exploits from regaining control without detection.

UNIT-5—Recommended Practices

Design:

Secure Software Development Principles

Least Common Mechanism:

The principle of least common mechanism advises minimizing shared mechanisms across multiple users or processes, as these shared resources can become security risks. If one user manages to exploit a shared mechanism, they may gain access to or alter data from other users, potentially introducing malicious code. While distributed systems often aim to provide shared data repositories, this principle suggests that, in certain scenarios, it may be safer to design systems where each application or process has its own isolated data store, reducing the risk of a breach affecting other instances or clients.

UNIT-5—Recommended Practices

Design:

Secure Software Development Principles

Psychological acceptability:

Psychological acceptability, or usability, refers to the principle that systems should be designed to be user-friendly, as poor usability can lead to security vulnerabilities. When systems are difficult to configure, prone to misconfiguration, or provide misleading error messages, users may make mistakes that create security risks, such as insecure default configurations. Many vulnerabilities, including those cataloged in the US-CERT Vulnerability Database, stem from usability issues like hard-to-configure settings, installation problems, and flawed documentation. Therefore, prioritizing usability not only improves user experience but also enhances security by reducing the likelihood of errors that could compromise the system.

UNIT-5—Recommended Practices

Design:

Threat Modelling

1. **Identify assets.** Identify the assets that your systems must protect.
2. **Create an architecture overview.** Document the architecture of your application, including subsystems, trust boundaries, and data flow.
3. **Decompose the application.** Decompose the architecture of your application, including the underlying network and host infrastructure design, to create a security profile for the application. The aim of the security profile is to uncover vulnerabilities in the design, implementation, or deployment configuration of your application.
4. **Identify the threats.** Identify the threats that could affect the application, considering the goals of an attacker and the architecture and potential vulnerabilities of your application.
5. **Document the threats.** Document each threat using a template that defines a common set of attributes to capture for each threat.
6. **Rate the threats.** Prioritize the threats that present the biggest risk based on the probability of an attack and the resulting damage. Some threats may not warrant any action, based on comparing the risk posed by the threat with the resulting mitigation costs.

UNIT-5—Recommended Practices

Design:

Analyze Attack Surface

Threats and Attack Surface: A system's attack surface represents the possible ways an adversary can exploit vulnerabilities, focusing on resources that might be targeted for damage.

Reducing Attack Surface: Reducing a system’s attack surface lowers the likelihood of successful attacks by limiting exposure to potential threats. This can be done by controlling access, closing unused channels, and minimizing unnecessary resources.

Principle of Least Privilege: One effective method for reducing the attack surface is applying the least privilege principle, granting the minimum required access to resources for each user.

Analysis of Targets, Channels, and Access Rights: To reduce the attack surface, it's essential to analyze the system's targets (resources to control), channels (communication methods), and access rights (privileges assigned to resources).

Measuring Attack Surface: The attack surface can be compared across different system versions or configurations, with decisions to increase the surface (e.g., adding features) being made consciously to manage security risks.

UNIT-5—Recommended Practices

Design:

Vulnerabilities in existing code

Modern software often relies on off-the-shelf components and libraries, which can introduce security vulnerabilities even if the application's code is flawless. For example, C library functions like realpath() may contain flaws, such as buffer overflows, that can be exploited if the system uses an outdated or vulnerable library version. While newer versions of these libraries may fix known issues, dynamically linked libraries can still pose risks if the application is deployed with older versions.

One solution is to statically link the libraries to ensure consistency, but this can increase the size of the application and limit access to future updates. Another approach is to validate inputs to external functions to ensure they are within safe ranges, preventing vulnerabilities from being exploited regardless of the library version.

UNIT-5—Recommended Practices

Design:

Secure Wrappers

- **Wrappers protect systems by intercepting calls to vulnerable APIs and adding validation checks, but they only work for known vulnerabilities.**
- **Supervised environments, like Systrace, isolate untrusted programs and prevent harmful actions, such as unauthorized file access or network connections.**
- **Both approaches don’t require modifying the program’s source code and can work with existing software.**
- **The main challenge with supervised environments is creating accurate policies, which can be difficult for complex programs and may break functionality.**

UNIT-5—Recommended Practices

Design:

Input Validation

- To prevent vulnerabilities, all sources of input, like command-line arguments, network interfaces, and user-controlled files, must be identified.
- The system must specify and validate the data from these sources, ensuring it meets expected formats and prevents issues like overflows.
- Specifications should cover limits, valid content, lengths, encryption requirements, and other constraints to ensure input is safe.
- Input should be validated early, and incorrect input should be flagged immediately to prevent later errors in the system.
- A data dictionary or similar method can help define what valid input looks like, and it’s important to also validate data stored persistently to guard against tampering.

UNIT-5—Recommended Practices

Design:

Trust Boundaries

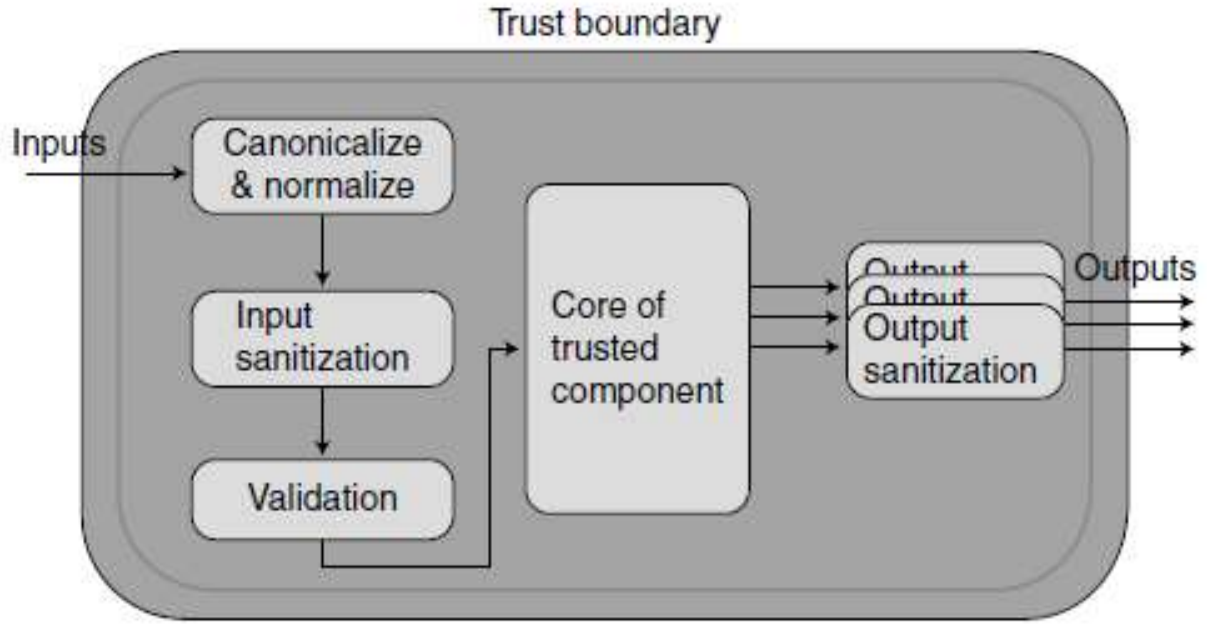

Figure 9.7 Trusted component

UNIT-5—Recommended Practices

Design:

Trust Boundaries

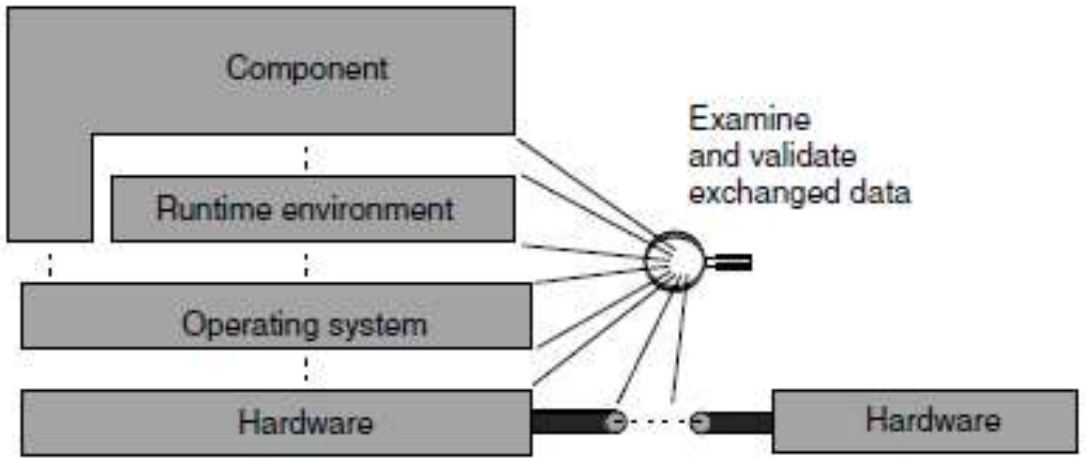

Figure 9.8 Exploitable interfaces (Source: [Wallnau 2002])

UNIT-5—Recommended Practices

Design:

Blacklisting:

One approach to data sanitization is to replace *dangerous* characters in input strings with underscores or other harmless characters

Example 9.1 Blacklisting Approach to Data Sanitization

```
01  int main(int argc, char *argv[]) {
02    static char bad_chars[] = "/ ;[]<>&\t";
03    char * user_data;
04    char * cp; /* cursor into example string */
```

```
06    user_data = getenv("QUERY_STRING");
07    for (cp = user_data; *(cp += strcspn(cp, bad_chars));)
08      *cp = '_';
09    exit(0);
10  }
```

UNIT-5—Recommended Practices

Design:

WhiteListing:

Example 9.2 tcp_wrappers Package Written by Wietse Venema

```
01  int main(void) {
02    static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz\
03    ABCDEFGHIJKLMNOPQRSTUVWXYZ\
04    1234567890_-.@";
05
06    char *user_data; /* ptr to the environment string */
07    char *cp; /* cursor into example string */
08
09    user_data = getenv("QUERY_STRING");
10    printf("%s\n", user_data);
11    for (cp = user_data; *(cp += strspn(cp, ok_chars)); )
12      *cp = '_';

13    printf("%s\n", user_data);
14    exit(0);
15  }
```

Implementation

Compiler Security Features

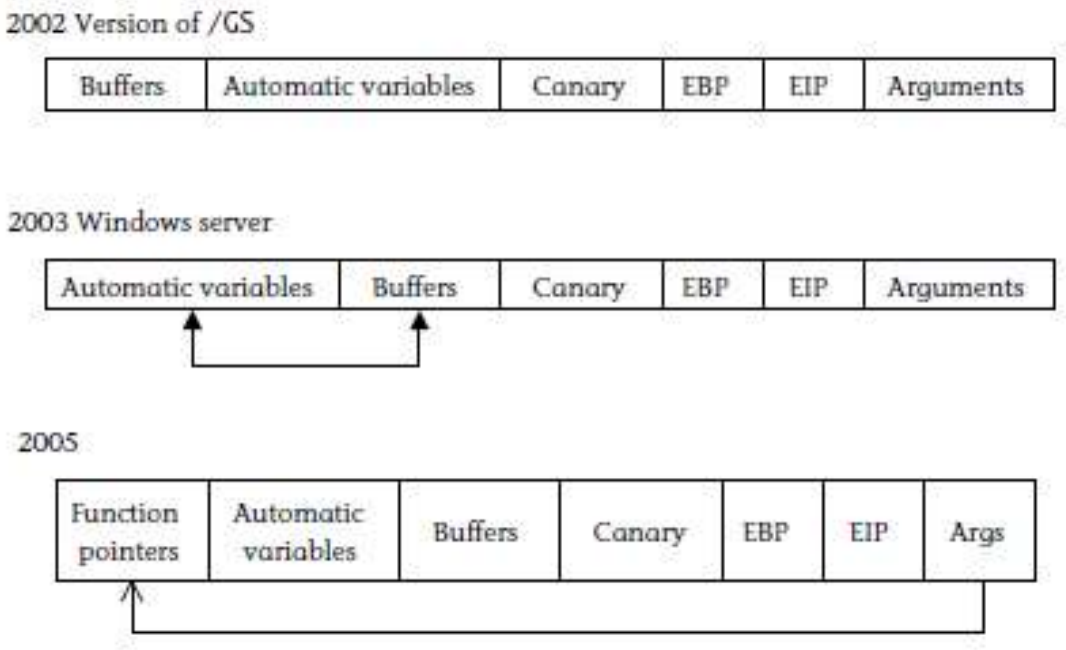

Figure 9.9 /GS flag for Visual C++

Implementation

Static Analysis methods

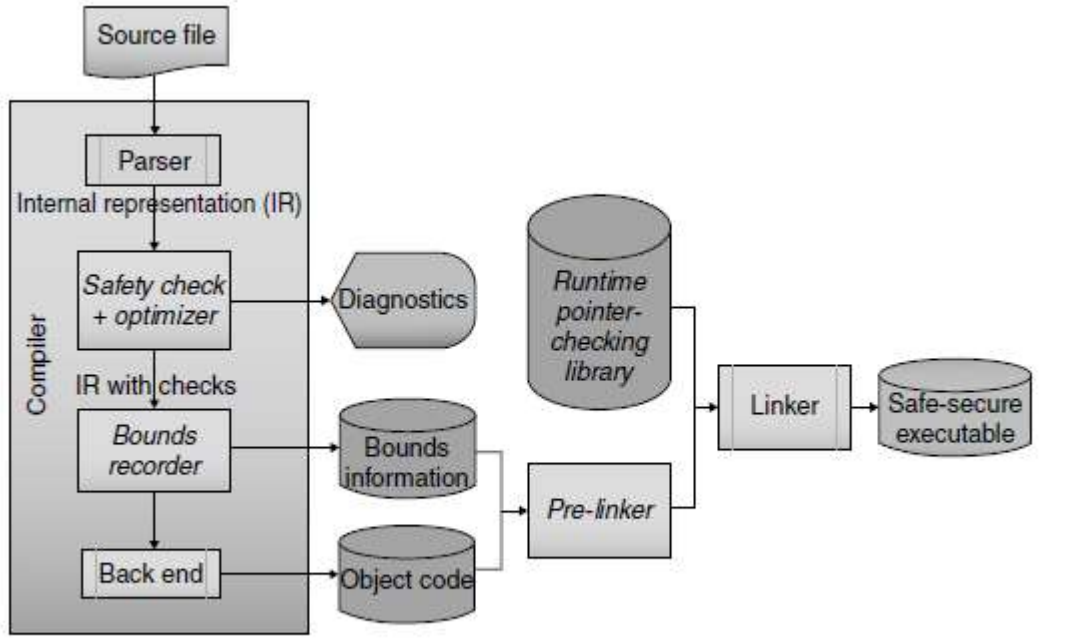

Figure 9.10 Safe-Secure C/C++ analysis methods

UNIT-5—Recommended Practices

Verification:

Static Analysis:

Finding the percentage of defects detected by static analysis tools is challenging, as demonstrated by Coverity's estimate that its tool might only identify 20% of actual defects while keeping false positives below 20%. Despite fixing defects identified by both static and dynamic analysis tools, their impact on customer-reported issues may be minimal, especially since vulnerabilities often arise from edge conditions that are not typically tested by regular users.

UNIT-5—Recommended Practices

Verification:

Penetration Testing:

Penetration testing involves testing an application, system, or network from an attacker’s perspective to find vulnerabilities, and it's most effective when informed by an architectural risk analysis. While penetration testing provides insights into real-world software security, basic black-box tests, which don’t consider the software’s architecture, often miss deeper security issues. Passing simple penetration tests doesn’t necessarily indicate strong security, but failing them shows significant risks. To thoroughly test security, it’s crucial to simulate serious attacks, scan for common vulnerabilities, and use various testing methods, such as white-box and black-box testing, to ensure both functional and nonfunctional security requirements are met.

Verification:

Fuzz Testing:

Fuzz testing is a software testing technique that involves providing random, unexpected, or invalid input to a program to identify vulnerabilities or bugs that might occur under unusual conditions. The goal is to uncover issues such as crashes, memory leaks, or security flaws by testing how the software handles inputs it wasn't specifically designed to handle. Fuzz testing can be automated, and it is particularly effective for finding vulnerabilities like buffer overflows or input validation errors, which can be exploited by attackers.

UNIT-5—Recommended Practices

Verification:

Code Audits:

Code audits in secure coding are systematic reviews of a software's source code to identify and fix potential security vulnerabilities before the software is deployed. The primary goal of a code audit is to ensure that the code follows best practices for security and does not contain weaknesses that could be exploited by attackers. During a code audit, security experts or developers manually inspect the code, looking for issues such as:

Buffer Overflows: These occur when data exceeds the boundaries of a buffer, potentially overwriting critical data or executing arbitrary code.

Injection Flaws: Such as SQL injection, where unsanitized user input is used in queries, potentially allowing attackers to manipulate the database.

Insecure Data Handling: Improper handling of sensitive data, such as hardcoded passwords, unencrypted sensitive information, or weak cryptographic algorithms.

Access Control Issues: These include flaws where users might gain unauthorized access to resources, functions, or data within the application.

Race Conditions: These occur when the outcome of a program depends on the timing or order of events, potentially allowing attackers to exploit concurrency issues.

UNIT-5—Recommended Practices

Verification:

Attack Surface reviews:

Identifying Entry Points: This includes all possible ways an attacker can interact with the system, such as network interfaces, APIs, user input fields, authentication mechanisms, and third-party integrations.

Evaluating Permissions and Access Control: Ensuring that only authorized users or systems can interact with sensitive parts of the application, and that the principle of least privilege is enforced.

Minimizing Exposed Interfaces: Reducing the number of open ports, services, or features that are unnecessary for the operation of the system, which limits the potential for attackers to exploit those areas.

Understanding Dependencies: Reviewing third-party libraries, services, and components that the system interacts with, as they can introduce additional attack vectors.

Code and Configuration Review: Examining the source code, server configurations, and security settings to detect weaknesses like insecure coding practices, misconfigurations, or over-permissive settings that could increase the attack surface.

The main purpose of attack surface reviews is to proactively reduce the risk of exploitation by shrinking the system's exposed attack points, making it more difficult for attackers to find and exploit vulnerabilities.