University of Technology

الجامعة التكنولوجية

Computer Science Department

قسم علوم الحاسوب

Software Security

امن البرمجيات

Assist Prof: Dr. Ayad Hazim

ا.م.د اياد حازم

cs.uotechnology.edu.iq

Software Security
Second level / second semester

Software and System Security Principles

1. Authentication
2. Access Rights
3. Confidentiality, Integrity, and Availability
4. Isolation
5. Least Privilege
6. Compartmentalization
7. Threat Model
8. Bug versus Vulnerability. .

Attack Vectors

1. Denial of Service (DoS)
2. Information Leakage
3. Confused Deputy
4. Privilege Escalation
5. Control-Flow Hijacking
6. Code Injection
7. Code Reuse

Defense Strategies

1. Software Verification
2. Language-based Security
3. Testing
   - Manual Testing
   - Sanitizers
   - Fuzzing
   - Symbolic Execution
4. Mitigations
   - Data Execution Prevention (DEP)/W^X 86
   - Address Space Layout Randomization (ASLR)
   - Stack integrity
   - Safe Exception Handling (SEH)
   - Fortify Source
   - Control-Flow Integrity
   - Code Pointer Integrity
   - Sandboxing and Software-based Fault Isolation

Software Security Principles, Policies, and Protection, Mathias Payer, July 2021, v0.37

Section One: Software and System Security Principles

Computer software, also called software, is a set of instructions and documentation that tells a computer what to do or how to perform a task. Software includes all different programs on a computer, such as applications and the operating system.

- Applications are programs that are designed to perform a specific operation, such as a game or a word processor.
- The operating system (e.g. Mac OS, Microsoft Windows, Android and various Linux distributions) is a type of software that is used as a platform for running the applications, and controls all user interface tools including display and the keyboard.

The word software was first used in the late 1960s to emphasize on its difference from computer hardware, which can be physically observed by the user. Software is a set of instructions that the computer follows. The word firmware usually refers to a piece of software that directly controls a piece of hardware. The firmware for a CD drive or a modem are examples of firmware implementation.

Software security refers to a set of practices that help protect software applications and digital solutions from attackers. Developers incorporate these techniques into the software development life cycle and testing processes. As a result, companies can ensure their digital solutions remain secure and are able to function in the event of a malicious attack.

Software Security Important: Secure software development is incredibly important because there are always people out there who seek to exploit business data. As businesses become more reliant on software, these programs must remain safe and secure. With strong software security protocols in place, you can prevent attackers from stealing potentially

sensitive information such as credit card numbers and trade secrets, and build trust among users. The theft of critical data can be catastrophic for customers and businesses alike. Malicious actors can abuse sensitive information and even steal users' identities. Additionally, companies can face legal penalties in the event of a data breach and suffer reputational harm.

Businesses can work to protect critical data by implementing software security techniques into their development life cycles. Applying security techniques enables organizations to proactively identify system vulnerabilities and better protect their software.

Authentication

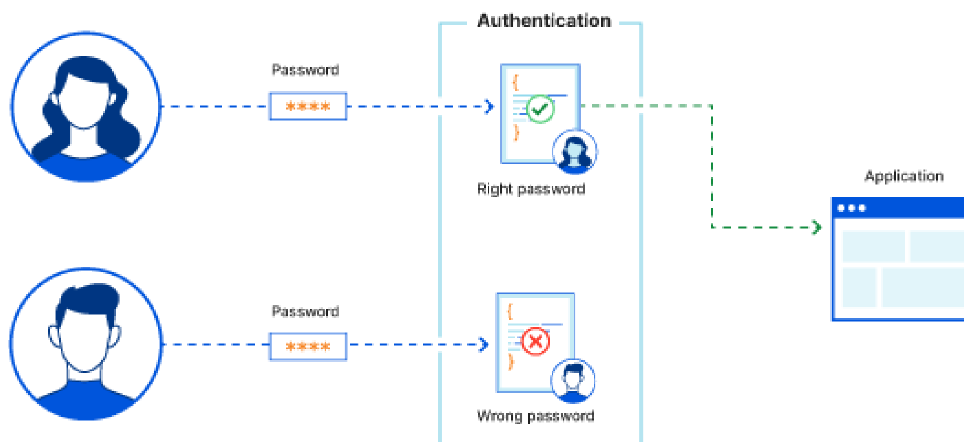
Authentication is a fundamental aspect of software security, ensuring that only authorized users or systems can access the software and its resources. It involves verifying the identity of a user, device, or other entity before granting access to the system. Here are the key components and methods of authentication in software security:

1. Password-Based Authentication

Strength and Complexity: Users are required to create strong passwords that combine uppercase and lowercase letters, numbers, and special characters to resist brute-force attacks.

2. Multi-Factor Authentication (MFA)

- **Something You Know:** This is usually a password or PIN.
- **Something You Have:** A physical token, smartphone, or a one-time password (OTP) generated by an authenticator app.
- **Something You Are:** Biometric authentication, such as fingerprint, facial recognition, or iris scanning.
- MFA adds an additional layer of security by requiring two or more forms of authentication.



Access Rights

Access control policies are a fundamental component of software development that governs the permissions and restrictions placed on users accessing a system or its resources. These policies define the rules and guidelines for granting or denying access to different functionalities, data, or areas within the software. There are several types of access control policies that can be implemented in software development to manage and enforce access to resources. These policies determine how permissions are granted or denied based on various factors, such as user roles, attributes, or predefined security levels.

1. Role-Based Access Control (RBAC). In RBAC, access rights are assigned to users based on their roles within the system. For example, an administrator may have full access to all functionalities, while a regular user may only have access to specific features.
2. Attribute-Based Access Control (ABAC) is another type of access control policy that considers additional attributes or characteristics of

users when granting or denying access. These attributes can include user location, time of access, device used, or any other relevant information.

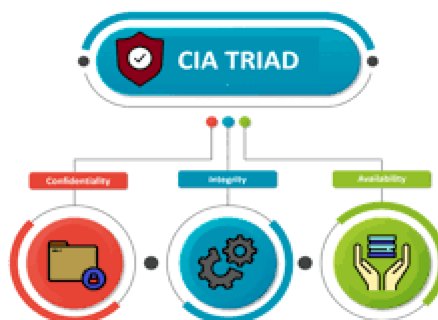
Security
Role

Default view * | Configure

+ Add Item | Edit List | Access Rights

Name	Permissions	Access Rights
Reader	(2 permissions)	Read / No Write
Contributor	(4 permissions)	Read / Write
Designer	(7 permissions)	Read / Write
Modeler	(10 permissions)	Read / Write
Admin	Admin	Read / Write

Confidentiality, Integrity, and Availability



Confidentiality, Integrity, and Availability (CIA) are the three core principles of information security, often referred to as the CIA triad. These principles form the foundation for designing and evaluating the security of systems, data, and processes. Here's a detailed overview of each component:

1. Confidentiality

- **Definition:** Confidentiality ensures that sensitive information is accessed only by authorized individuals or systems and is protected from unauthorized disclosure.
- **Purpose:** To protect information from being disclosed to unauthorized parties, thereby preventing breaches of privacy and security.
- **Key Concepts and Practices:**
  - **Encryption**
  - **Access Controls**
- **Examples of Confidentiality Breaches:**
  - **Data Leaks:** Sensitive information, like personal data or trade secrets, being exposed due to inadequate access controls.
  - **Unauthorized Access:** Hackers gaining access to confidential information through phishing, malware, or other attack vectors.

2. Integrity

- **Definition:** Integrity ensures that data is accurate, consistent, and has not been tampered with or altered by unauthorized parties.
- **Purpose:** To maintain the trustworthiness and accuracy of information, ensuring that it remains unchanged from its original state unless properly authorized.
- **Key Concepts and Practices:**
  - **Checksums and Hashing:** Using checksums or cryptographic hashing (e.g., SHA-256) to detect alterations in data. Any changes to the data will result in a different hash value.
  - **Digital Signatures:** Applying digital signatures to data to verify its origin and ensure it has not been modified during transmission.
- **Examples of Integrity Breaches:**

  - **Data Tampering:** Unauthorized modification of data, such as altering financial records, which can lead to fraud or misinformation.
  - **Man-in-the-Middle Attacks:** Attackers intercepting and altering data during transmission, potentially compromising the integrity of communication.

3. Availability

- **Definition:** Availability ensures that information and systems are accessible and usable when needed by authorized users.
- **Purpose:** To ensure that systems and data are available to users in a timely manner, supporting the continuity of business operations.
- **Key Concepts and Practices:**
  - **Redundancy:** Implementing redundant systems, such as backup servers, failover clusters, and data backups, to ensure continuous availability even if a component fails.
  - **Load Balancing:** Distributing workloads across multiple systems or servers to prevent overload and ensure that resources are available even under heavy usage.
- **Examples of Availability Breaches:**
  - **DDoS Attacks:** Overloading a system with traffic to make it unavailable to legitimate users.
  - **Hardware Failures:** System crashes or server outages leading to unavailability of critical services.

Isolation

Isolation in software security refers to the practice of separating different components, processes, or data within a system to enhance security. By isolating these elements, potential security risks are contained, preventing them from spreading to other parts of the system. Isolation is a key strategy in reducing the attack surface and minimizing the impact of security breaches. Here's a Types **of Isolation** in software security:

1. **Process Isolation:** Ensuring that each process runs in its own memory space, separate from other processes. This prevents one process from accessing or interfering with the memory of another.
2. **Virtualization:** Each **Virtual Machines** VM runs its own operating system instance, isolated from other VMs on the same physical host. Hypervisors manage the isolation between VMs.
3. **Network Isolation:** Dividing a network into smaller, isolated segments or zones to control traffic flow and limit the spread of potential attacks. For example, isolating the internal network from the public-facing network.
4. **Privilege Isolation: Principle of Least Privilege (PoLP),** users and processes are granted the minimum level of access rights necessary to perform their tasks, ensuring that higher-privileged functions are isolated from lower-privileged ones.
5. **Database Isolation:** Ensuring that data from different users or tenants is kept separate, especially in multi-tenant environments like cloud services.
6. **Code Isolation:**
   - **Module Isolation:** Structuring code into isolated modules or components with well-defined interfaces, limiting the impact of vulnerabilities within a single module.

- **Micro services:** A software architecture where applications are built as a collection of loosely coupled, independently deployable services, each running in its own isolated environment.
- **Benefits:** Enhances maintainability and security by containing vulnerabilities within specific components.

Least Privilege

The Principle of Least Privilege (PoLP) is a fundamental concept in software security that advocates for granting users, systems, and processes the minimum level of access or privileges necessary to perform their required tasks. By limiting access rights, the potential attack surface is reduced, minimizing the risk of accidental or intentional misuse of privileges.

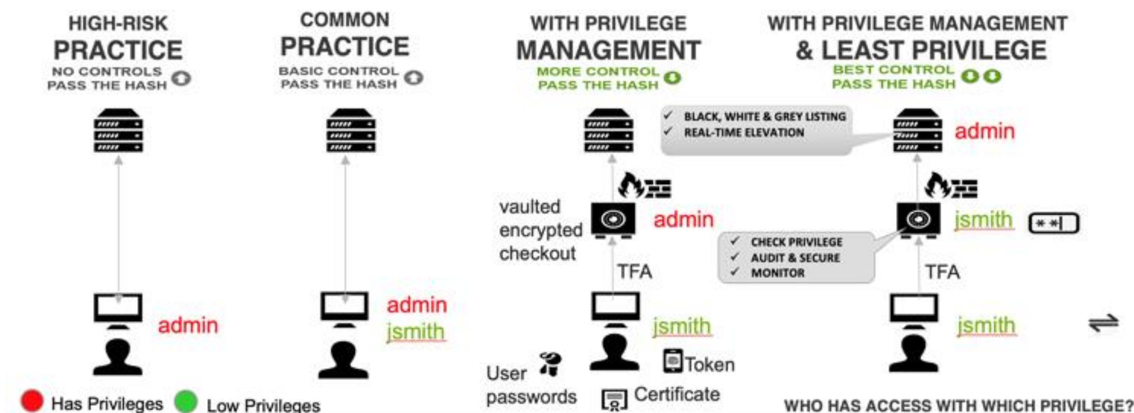
The goal is Least Privilege to reduce security risks by limiting the potential damage that could result from security breaches or errors. This minimizes the opportunities for malicious actors or malware to exploit elevated privileges.

Implementation Least Privilege in Different Contexts

- **User Accounts:**
  - Regular users should have access only to the files, applications, and systems needed for their job. Administrative privileges should be restricted to a few trusted individuals who need them to perform specific tasks.
- **System Processes:**
  - System services and applications should run under dedicated service accounts with minimal privileges, rather than under accounts with full administrative rights.
- **Applications:**
  - Applications should request only the permissions necessary to perform their functions. For example, a messaging app should

not require access to the device's camera unless it supports video calling.

- **Network Access:**
    - Network resources should be segmented, and access should be restricted based on the principle of least privilege, ensuring that users or systems can only access the parts of the network they need.



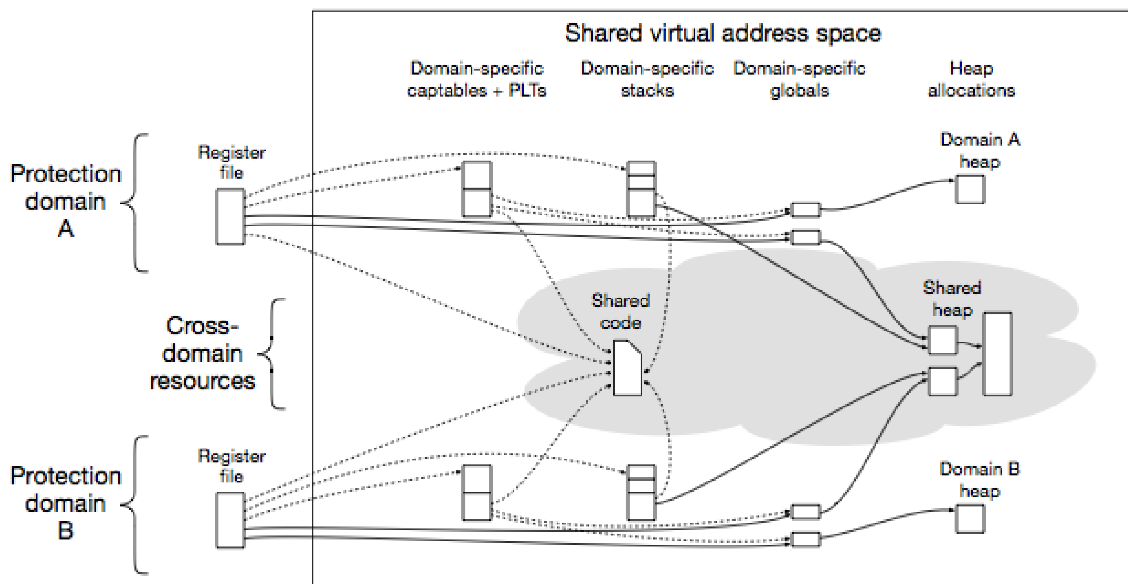
Compartmentalization

Compartmentalization in software security refers to the practice of dividing a system into distinct, isolated sections or compartments, each with specific functions, data, and access controls. The purpose is to limit the impact of a security breach by ensuring that if one compartment is compromised, the others remain unaffected. This approach enhances security by reducing the attack surface and containing potential threats. **Examples of Compartmentalization in Practice**

- **Military and Government Systems:**
    - Sensitive information in military and government systems is often compartmentalized based on classification levels (e.g.,

Confidential, Secret, Top Secret), with strict access controls and isolation between compartments.

- **Cloud Environments:**
  - In cloud computing, multitenant architectures use compartmentalization to ensure that each tenant's data and applications are isolated from others, protecting against cross-tenant data breaches.
- **Web Applications:**
  - Web applications often compartmentalize user sessions, isolating them from each other to prevent session hijacking and cross-site attacks.



Threat Model

A Threat Model in software security is a structured approach used to identify, evaluate, and address potential threats that could harm a software system. The purpose of threat modeling is to understand the security risks to a system, prioritize those risks, and develop strategies to mitigate them. This process is

crucial in building secure software, as it helps in anticipating and countering potential attacks before they occur.

Threat modeling is the process of systematically identifying security threats and vulnerabilities, assessing their potential impact, and planning mitigations to protect the system.

The primary goal is to improve the security posture of a system by proactively identifying and addressing potential threats, ensuring that security is integrated throughout the software development lifecycle. Key Components of Threat Modeling

- **Assets:** The valuable components of the system that need protection, such as data.
- **Threats:** Potential actions or events that could compromise the confidentiality, integrity, or availability of assets.
- **Vulnerabilities:** Weaknesses or flaws in the system that could be exploited by a threat to cause harm.
- **Attack Vectors:** The paths or methods that attackers use to exploit vulnerabilities and carry out threats.
- **Mitigations:** The security controls or countermeasures implemented to reduce or eliminate the risks posed by threats.

Bug versus Vulnerability

In software security, "bug" and "vulnerability" are terms that refer to different aspects of software flaws. While they are related, they have distinct meanings and implications.

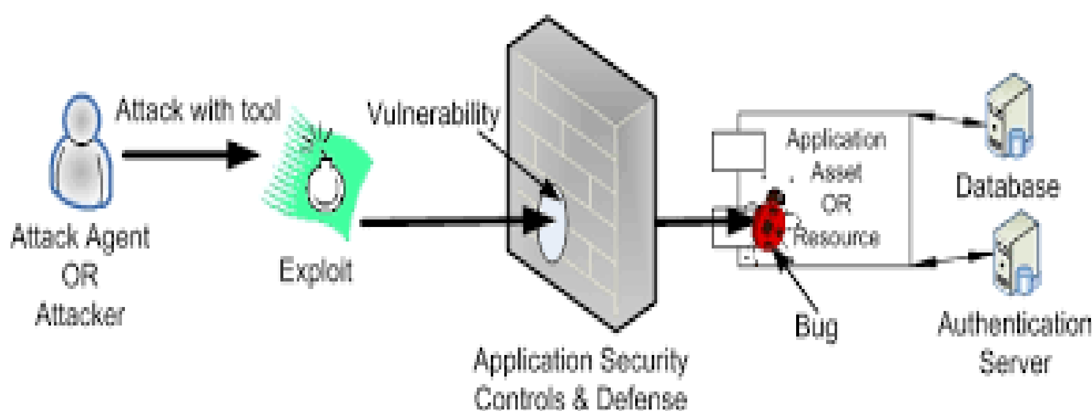
1. Bug

- **Definition:** A bug is a flaw, mistake, or unintended behavior in the software's code that causes the software to operate incorrectly or produce unexpected results. Bugs can arise from errors in logic, incorrect assumptions, or programming mistakes.
- **Scope:** Bugs can affect the functionality, performance, or usability of software. They are not necessarily security-related and may not have any impact on the security of the system.
- **Examples:**
  - A calculation error in a financial application that leads to incorrect results.
  - A typo in a user interface string that displays incorrect information to the user.
  - A crash in a software program due to improper memory management.

2. Vulnerability

- **Definition:** A vulnerability is a specific type of bug or flaw in software that can be exploited by an attacker to compromise the security of the system. Vulnerabilities create opportunities for unauthorized access, data breaches, or other malicious activities.
- **Scope:** Vulnerabilities directly impact the confidentiality, integrity, or availability of a system. They can be exploited to perform actions that should not be possible, such as gaining unauthorized access, executing arbitrary code, or causing a denial of service.

- **Examples:**
    - A buffer overflow vulnerability that allows an attacker to execute arbitrary code on a system.
    - An SQL injection vulnerability that enables an attacker to manipulate a database.
    - An authentication bypass vulnerability that lets an attacker access a system without proper credentials.



Section two: Attack Vectors

An attack vector, or threat vector, is a way for attackers to enter a network or system. Common attack vectors include social engineering attacks, credential theft, vulnerability exploits, and insufficient protection against insider threats.

Suppose a security firm is tasked with guarding a rare painting that hangs in a museum. There are several ways that a thief could enter and exit the museum — front doors, back doors, elevators, and windows. A thief could enter the museum in some other way too, perhaps by posing as a member of the museum's staff. All of these methods represent attack vectors, and the security firm may try to eliminate them by placing security guards at all doors, putting locks on windows, and regularly screening museum staff to confirm their identity.

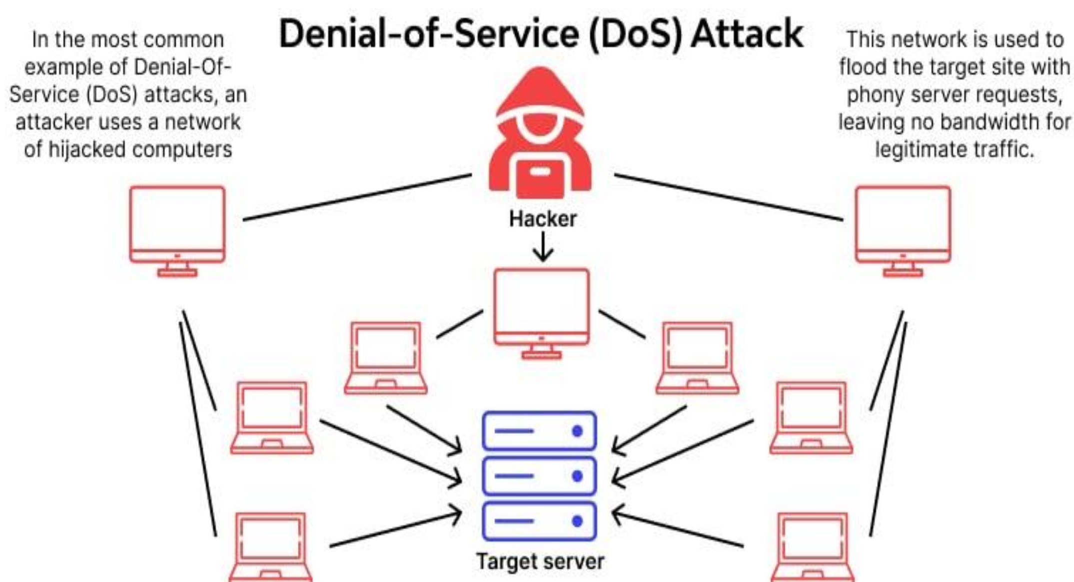
Similarly, digital systems all have areas attackers can use as entry points. Because modern computing systems and application environments are so complex, closing off all attack vectors is typically not possible. But strong security practices and safeguards can eliminate most attack vectors, making it far more difficult for attackers to find and use them.

Denial of Service (DoS)

A denial-of-service (DoS) attack occurs when legitimate users are unable to access information systems, devices, or other network resources due to the actions of a malicious cyber threat actor. Services affected may include email, websites, online accounts (e.g., banking), or other services that rely on the affected computer or network.

A denial-of-service condition is accomplished by flooding the targeted host or network with traffic until the target cannot respond or simply

crashes, preventing access for legitimate users. DoS attacks can cost an organization both time and money while their resources and services are inaccessible.



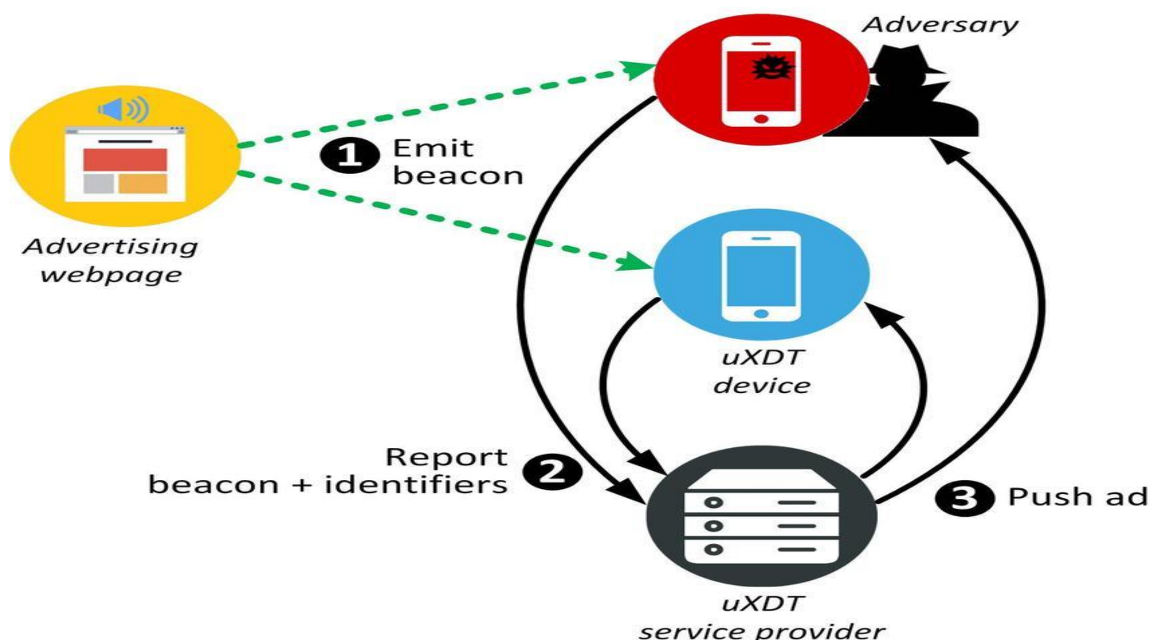
Information Leakage

Information leakage is the sharing of sensitive information with unauthorized parties. The leakage can be either;

1. Accidental, such as an employee sharing confidential information with an external party via email, or malicious, such as the exfiltration of data through phishing scams.
2. Regardless of the intent, however, the information shared is valuable to hackers and can be used to execute attacks on your organization's infrastructure, services or applications.

While information leaks originate from within an organization, data breaches are a result of actions that take place from unauthorized users from outside of the organization. Encryption, implementing security

controls and classifying sensitive data are all strategies organizations use to prevent data loss. In addition, many organizations have various data leak prevention strategies and technology in place to defend against data breaches.



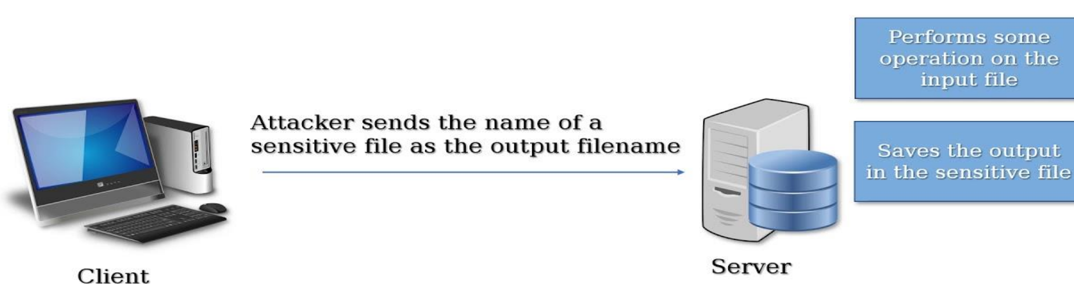
Confused Deputy

The "Confused Deputy" problem is a security issue that arises when a program (the "deputy") is tricked into performing actions on behalf of an attacker with more privileges than the attacker should have. This problem often occurs in systems where one program (the deputy) performs tasks on behalf of another program or user, and the deputy is tricked into executing actions it would not normally perform.

Here's a simplified example of how the Confused Deputy problem might manifest:

1. **Scenario**: Imagine a web application that allows users to upload files. The application processes these files on behalf of users, using a server-side script with permissions to read and write files in a specific directory.

2. **Exploitation**: An attacker might upload a file with a name that includes a path traversal attack (e.g., ../sensitive_file.txt). The application's script, operating with the server's higher privileges, might process this file and inadvertently read or write to sensitive files outside the intended directory.
3. **Result**: The attacker successfully accesses or modifies sensitive files by exploiting the higher privileges of the server-side script.



Privilege Escalation

Privilege escalation is a type of security vulnerability where an attacker gains elevated access to resources or permissions beyond what they are normally authorized to have. This can occur through various methods, leading to unauthorized access to sensitive data, system control, or administrative functions. Privilege escalation can be categorized into two main types:

Vertical privilege escalation occurs when a user with lower-level permissions gains access to higher-level privileges or administrative rights. For example:

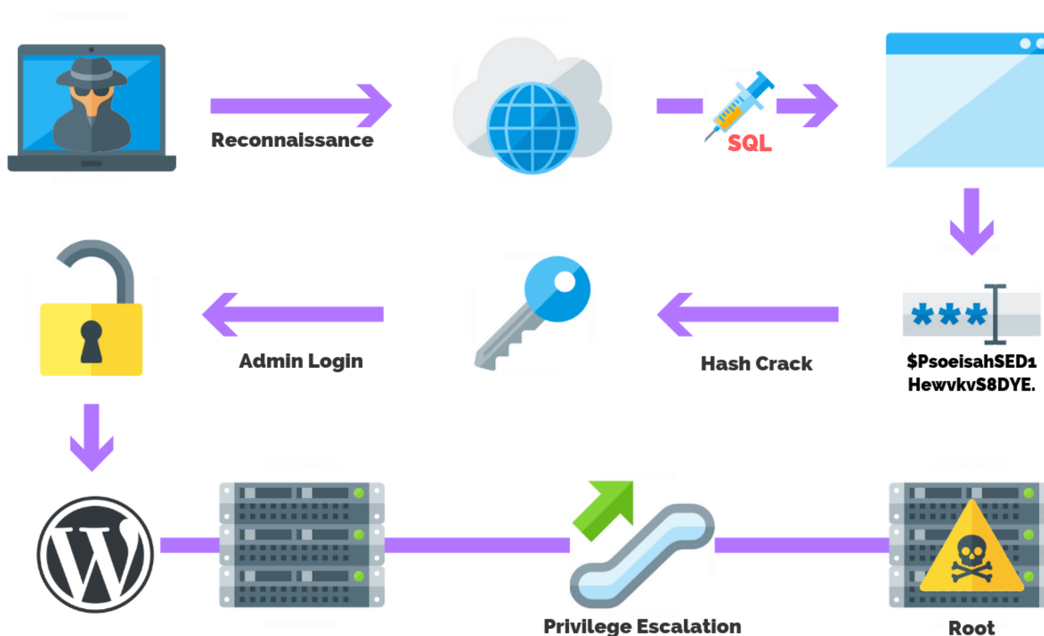
- **Exploiting Vulnerabilities**: Attackers might exploit software vulnerabilities to gain administrative access. For instance,

vulnerability in an application might allow a user to execute commands with root privileges.

- **Weak Authentication**: Weak or misconfigured authentication mechanisms can allow an attacker to impersonate an administrative user and gain elevated access.
- **Misconfigured Permissions**: Incorrectly configured file or directory permissions can enable a user to access or modify files they shouldn't.

Horizontal privilege escalation happens when a user gains access to resources or permissions of another user with the same level of privilege. For example:

- **Session Fixation**: An attacker might hijack a user's session to access resources that the user can access, even though the attacker shouldn't have access to those resources.
- **Insecure APIs**: APIs that do not enforce proper access controls may allow an attacker to perform actions intended for other users with similar permissions.

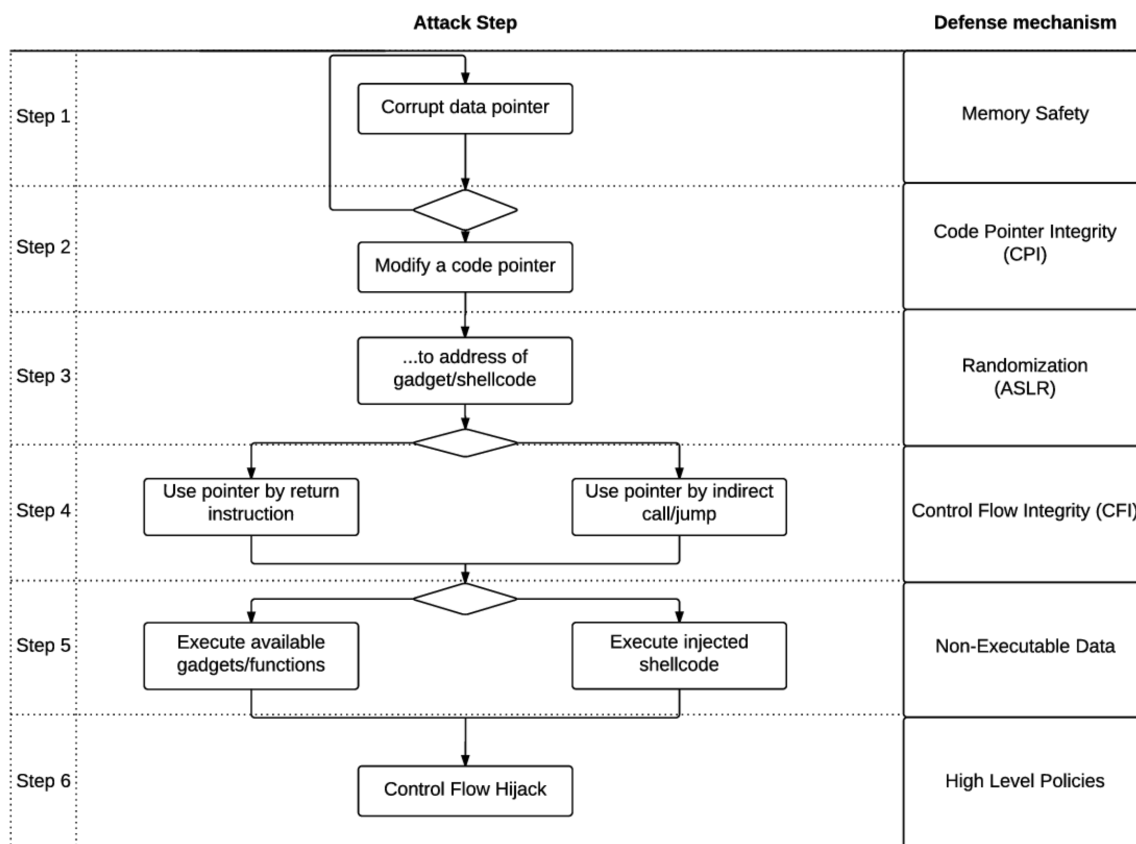


Control-Flow Hijacking

Control-flow hijacking is a type of security vulnerability that allows an attacker to alter the flow of execution in a program. This usually involves redirecting the program's execution to a location of the attacker's choosing. It's often used to exploit software vulnerabilities such as buffer overflows, where an attacker can overwrite parts of memory and redirect the execution flow to execute malicious code.

Here are some common types of control-flow hijacking techniques:

1. **Buffer Overflow Attacks**: By overwriting a buffer, an attacker can alter the return address on the stack, redirecting the program's execution to their own code.
2. **Return-Oriented Programming (ROP)**: An attacker uses existing code snippets (gadgets) that end in return instructions to execute malicious code without injecting new code into the process.
3. **Jump-Oriented Programming (JOP)**: Similar to ROP but uses jump instructions instead of return instructions to build a chain of gadgets for malicious purposes.
4. **Code Injection Attacks**: Injecting malicious code into the process's memory space and redirecting execution to it.
5. **Function Pointer Overwriting**: Manipulating function pointers to redirect the program's execution to malicious code.



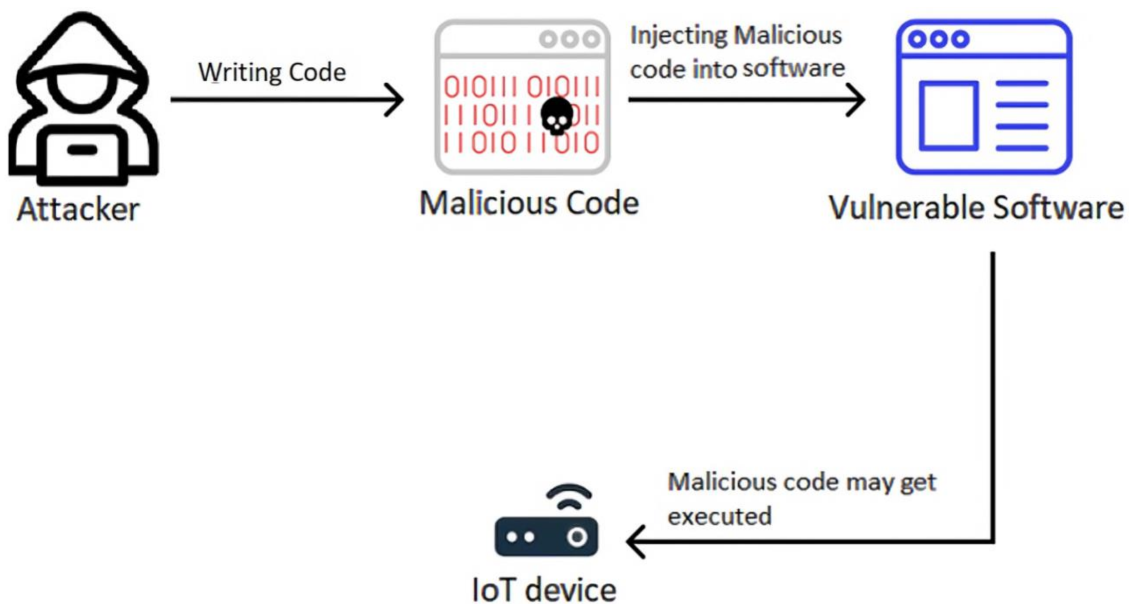
Code Injection

Code injection is a type of security vulnerability where an attacker is able to insert or "inject" malicious code into a program or system. This injected code is then executed by the system, leading to unauthorized actions or compromise. Code injection attacks can affect various types of applications and systems, including web applications, databases, and even local software.

Common Types of Code Injection

1. **SQL Injection**: An attacker inserts malicious SQL queries into a form input or URL parameter to manipulate or gain unauthorized access to a database.
2. **Cross-Site Scripting (XSS)**: Malicious scripts are injected into web pages viewed by other users, potentially leading to session hijacking, data theft, or other malicious actions.

3. **Command Injection**: Malicious commands are injected into an application's input fields, which are then executed by the server's command-line interface.
4. **Code Injection in Programming Languages**: Attacks that exploit weaknesses in how programming languages handle code execution, such as JavaScript eval(), PHP eval(), or Python exec().
5. **Script Injection**: Similar to XSS but more broadly involves injecting scripts into any environment where the script might be executed.



Code Reuse

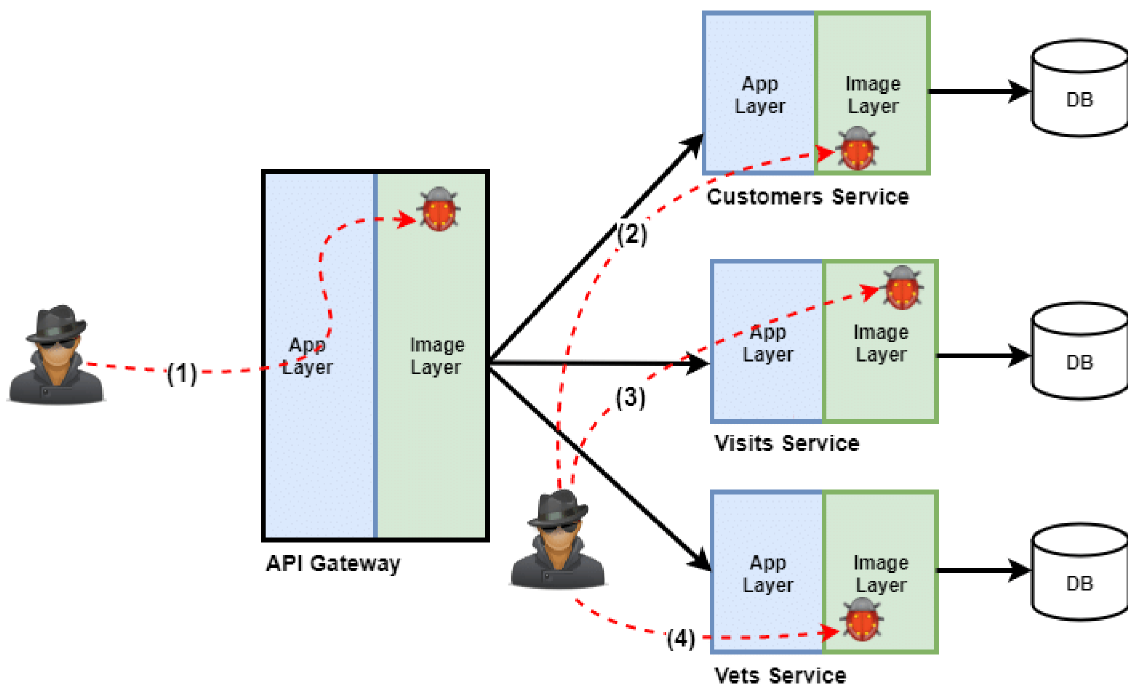
A code reuse attack is a type of security exploit where an attacker leverages existing code within a program or system to perform malicious actions. Instead of injecting new code, the attacker reuses legitimate code already present in the program, often to bypass security mechanisms. These attacks are typically sophisticated and require an understanding of the program's structure and available code snippets.

Common Types of Code Reuse Attacks

1. **Return-Oriented Programming (ROP)**: An attacker uses sequences of instructions (gadgets) ending in return instructions to build a chain of operations that perform malicious actions. This technique is used to exploit vulnerabilities like buffer overflows, where the attacker manipulates the program's control flow to execute their chosen sequence of gadgets.
2. **Jump-Oriented Programming (JOP)**: Similar to ROP, but instead of return instructions, JOP relies on jump instructions to control the execution flow. JOP can be used to bypass security defenses by exploiting existing code that uses jump instructions.
3. **Call-Oriented Programming (COP)**: This technique involves chaining together existing code that contains call instructions. By manipulating these calls, an attacker can execute a series of operations in a controlled manner.

Characteristics of Code Reuse Attacks

- **No Code Injection**: These attacks do not require injecting new code into the memory but instead exploit existing code, which can be harder to detect and mitigate.
- **Bypassing Security Mechanisms**: Code reuse attacks can bypass security measures like Data Execution Prevention (DEP) or Address Space Layout Randomization (ASLR) by avoiding the need to execute injected code.
- **Exploitation of Code Patterns**: Attackers exploit predictable patterns or behaviors in the existing code to achieve their goals.



Section three: Defense Strategies

A 'Defense Strategy' in the context of Computer Science refers to the systematic approach of layering defenses to enhance effectiveness and reduce costs in protecting against cyber-attacks. It involves implementing controls to mitigate potential impacts and improve decision-making for responding to attacks.

Defense strategies in software security aim to protect applications and systems from various types of attacks and vulnerabilities. A multi-layered approach is often used to ensure comprehensive protection. Here are some key defense strategies:

Software Verification

Software verification is a crucial process in ensuring that software systems meet their specified requirements and function correctly. It involves various techniques and methods to confirm that the software behaves as intended and is free of critical errors.

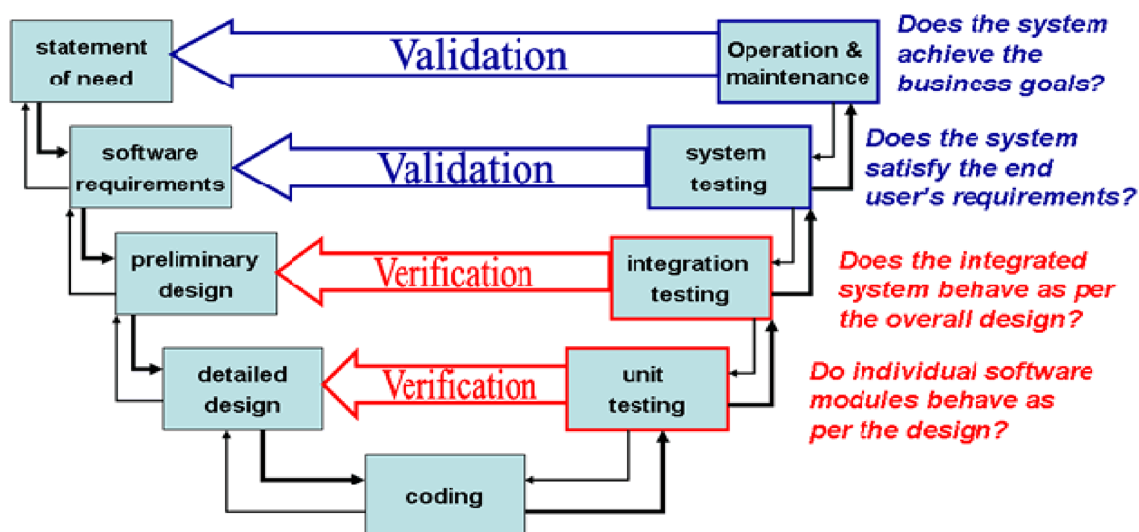
Software verification Ensure that software requirements are clear, complete, and feasible before development begins. Verify that each requirement is addressed by corresponding design elements and test cases.

Static Code Analysis: Use tools to analyze source code without executing it. This helps in identifying potential issues such as coding standards violations, security vulnerabilities, and other defects.

Dynamic Verification: Test individual components or units of code in isolation to ensure they work correctly. Test the interactions between integrated components or systems to identify issues related to their interaction. Verify the complete and integrated system against the requirements. This includes functional testing, performance testing, and security testing.

Verification Techniques

- **Model-Based Verification**: Create and analyze models of the software to verify that it meets its requirements and behaves correctly.
- **Formal Specification**: Use formal methods to specify and reason about software behavior mathematically.
- **Simulation and Emulation**: Test software in a simulated or emulated environment to verify its behavior under controlled conditions.

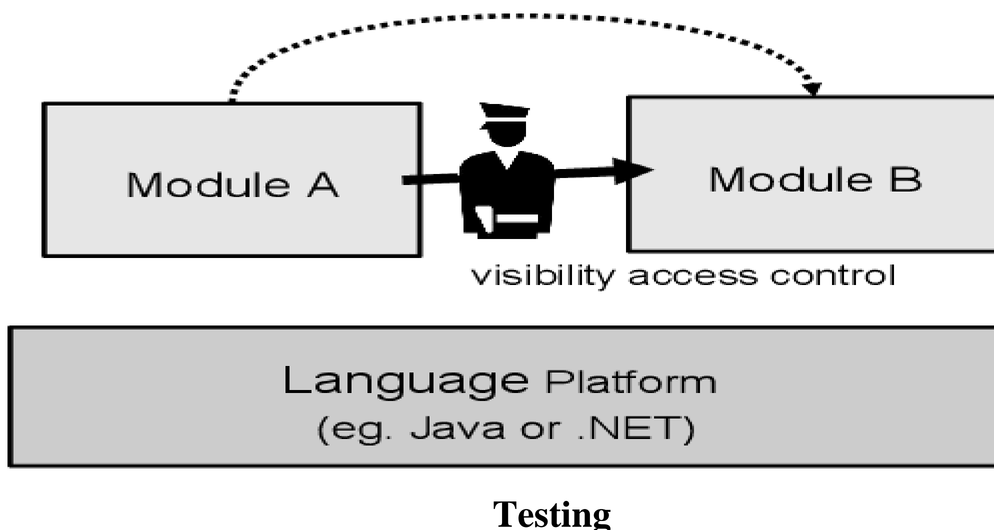


Language-based Security

Language-based security focuses on using programming languages and formal methods to enhance the security of software systems. This approach involves designing languages and language features that help prevent security vulnerabilities, such as buffer overflows or injection attacks. Here are some key concepts:

1. **Type Systems:** Strong type systems can help catch errors at compile time, preventing many common vulnerabilities. For example, languages like Rust use ownership and borrowing to manage memory safely.

2. **Formal Verification:** This involves mathematically proving that a program adheres to its specification and is free from certain types of errors. Languages designed with formal verification in mind, such as Ada or Coq, can ensure higher levels of reliability and security.
3. **Safe Programming Constructs:** Languages can provide constructs that limit the kinds of operations that can be performed, reducing the risk of errors. For instance, languages like Java and C# provide automatic garbage collection to avoid memory leaks and dangling pointers.
4. **Secure Language Design:** Some languages are designed with security as a primary concern. For example, languages like Haskell and Scala have features that support functional programming, which can make it easier to reason about and ensure the security of code.
5. **Security-focused Libraries and Frameworks:** Many modern languages come with libraries and frameworks designed to prevent common security issues, like SQL injection or cross-site scripting (XSS).



Testing

In the context of software testing, a defense strategy involves incorporating various practices and methodologies to safeguard the software from defects, vulnerabilities, and performance issues.

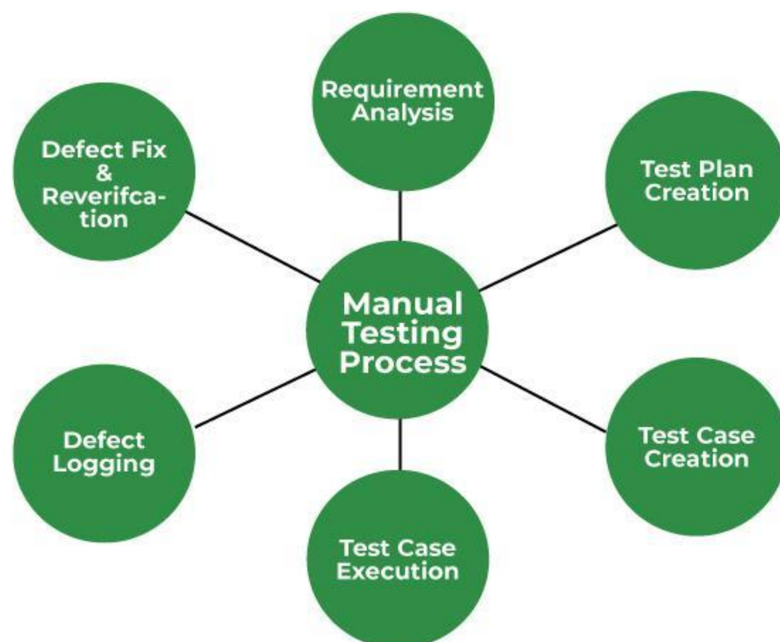
Manual Testing

Manual software testing involves a tester manually executing test cases without the use of automated tools. The goal is to identify bugs or issues in the software by simulating the end-user experience. Here's a basic outline of how it typically works:

1. **Test Planning**: Define the scope and objectives of testing. Create a test plan that outlines what to test, how to test, and the resources required.
2. **Test Case Design**: Develop detailed test cases based on the software requirements and specifications. Each test case should have clear steps, expected results, and conditions.
3. **Test Execution**: Manually execute the test cases on the software application. This involves navigating through the application, entering data, and verifying that the software behaves as expected.
4. **Defect Reporting**: When issues or bugs are found, document them in a defect tracking system. Include details such as steps to reproduce, expected vs. actual results, and severity.
5. **Test Closure**: Once testing is complete, review and analyze the results. Ensure all critical issues are resolved and finalize testing documentation.
6. **Regression Testing**: After fixes are applied, retest the affected areas to ensure that the changes didn't introduce new issues.

The below diagram lists the steps in the manual testing process:

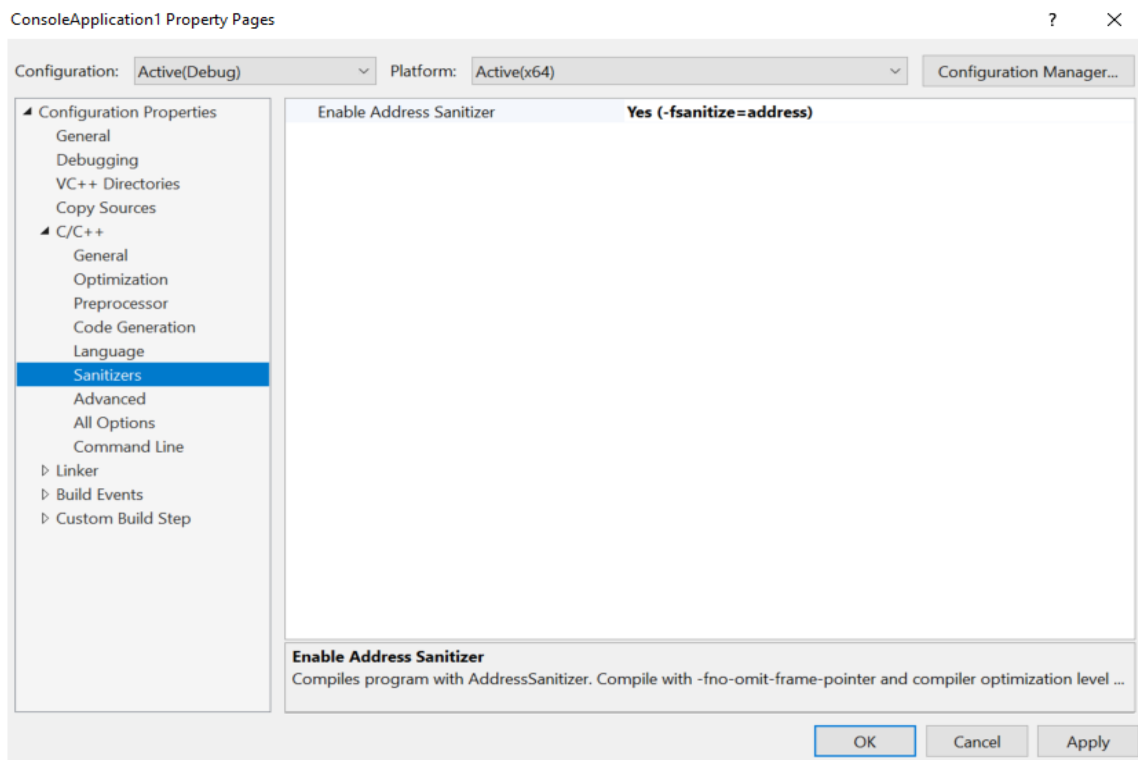
1. **Requirement Analysis:** Study the software project documentation, guides, and Application Under Test (AUT). Analyze the requirements from SRS.
2. **Test Plan Creation:** Create a test plan covering all the requirements.
3. **Test Case Creation:** Design the test cases that cover all the requirements described in the documentation.
4. **Test Case Execution:** Review and baseline the test cases with the team lead and client. Execute the test cases on the application under test.
5. **Defect Logging:** Detect the bugs, log and report them to the developers.
6. **Defect Fix and Re-verification:** When bugs are fixed, again execute the failing test cases to verify they pass.



Sanitizers

Sanitizers are tools designed to detect various types of bugs in programs, often related to memory safety and undefined behavior. They work by instrumenting the code to check for common issues during runtime. Some common types of sanitizers include:

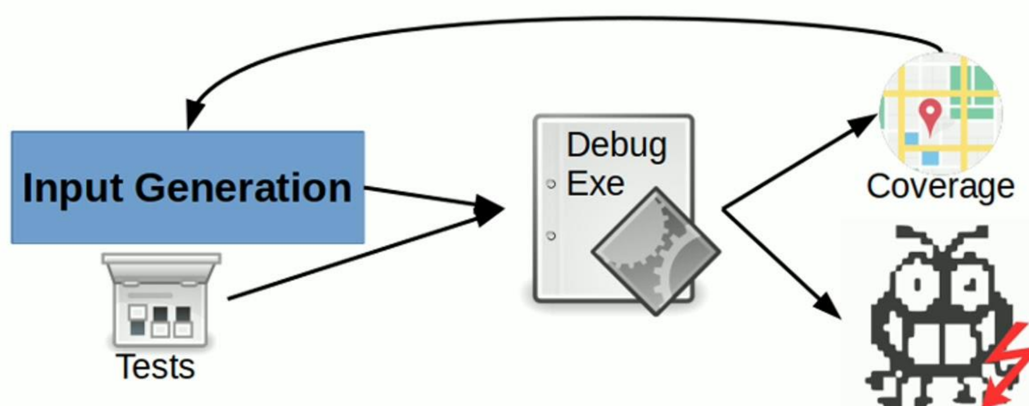
- **AddressSanitizer (ASan)**: Detects memory errors such as buffer overflows, use-after-free, and memory leaks.
- **MemorySanitizer (MSan)**: Identifies uninitialized memory reads.
- **ThreadSanitizer (TSan)**: Finds data races and threading issues.
- **UndefinedBehaviorSanitizer (UBSan)**: Catches undefined behaviors, such as integer overflows or invalid operations.



Fuzzing

Fuzzing is an automated testing technique that involves sending a large volume of random or semi-random inputs to a program to uncover vulnerabilities or crashes. It works by generating a wide range of inputs to test how the software handles unexpected or malformed data. Fuzzing can be categorized into different types:

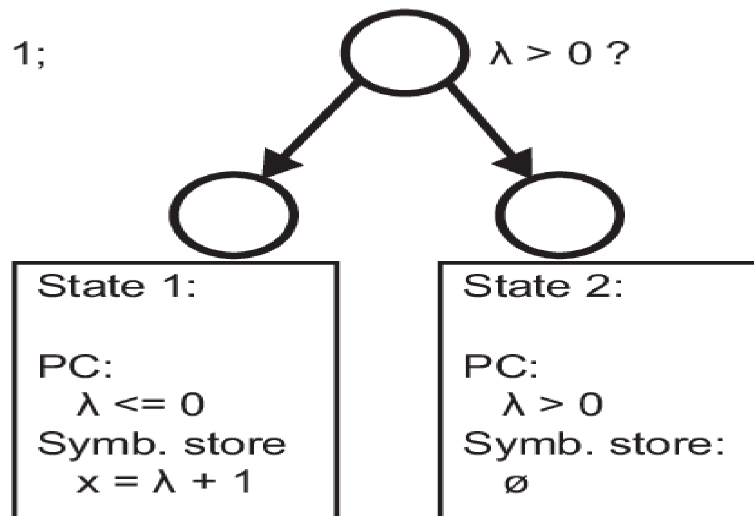
- **Mutation-Based Fuzzing**: Modifies existing inputs or seeds to generate new test cases.
- **Generation-Based Fuzzing**: Creates inputs from scratch based on input formats or protocols.
- **Coverage-Guided Fuzzing**: Uses feedback from the program's execution to guide the generation of more effective test inputs.



Symbolic Execution

Symbolic execution is a technique used to analyze a program by exploring different execution paths based on symbolic values rather than concrete inputs. It involves:

- **Path Exploration**: The symbolic execution engine explores possible execution paths of the program, which can lead to discovering hidden bugs and vulnerabilities.
- **Constraint Solving**: The system generates constraints based on the symbolic inputs and solves them to find possible values that would lead to different execution paths.
- **Error Detection**: Identifies potential issues by analyzing the constraints and the symbolic execution paths.



Section four: Mitigations

In software security, mitigations are techniques and practices used to reduce the risk of vulnerabilities and attacks. Here's a more comprehensive list of common mitigations:

1. Input Validation and Sanitization:
2. Output Encoding:
3. Authentication and Authorization:
4. Least Privilege Principle:
5. Secure Coding Practices:
6. Regular Updates and Patching:
7. Error Handling and Logging:
8. Encryption:
9. Security Testing and Code Reviews:
10. Threat Modeling:
11. Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR):
12. Sandboxing and Isolation:
13. Control-Flow Integrity (CFI):
14. Code Pointer Integrity (CPI):
15. Safe Exception Handling (SEH):
16. Fortify Source:
17. Secure Software Development Lifecycle (SDLC):

Data Execution Prevention (DEP)/W^X 86

Data Execution Prevention (DEP) and **W^X (Write XOR Execute)** are security features designed to prevent certain types of exploits by controlling the execution of code and manipulation of memory.

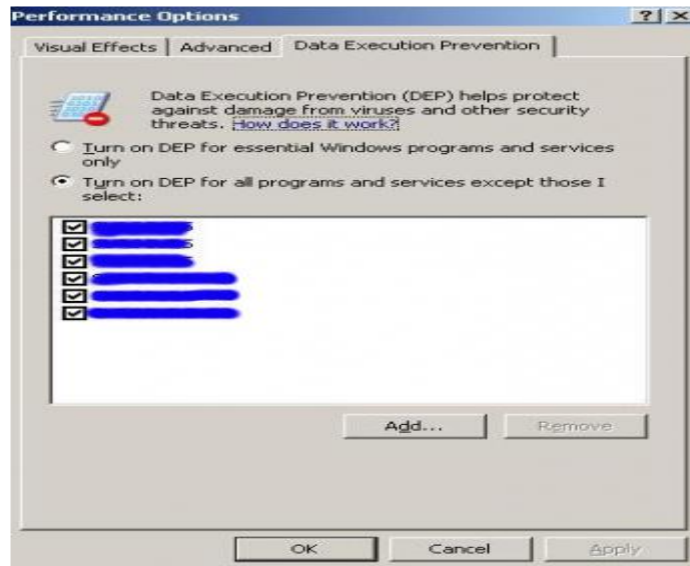
Data Execution Prevention (DEP)

- **Definition**: DEP is a security feature that prevents code from being executed in specific regions of memory that are intended only for data. This helps to protect against attacks where malicious code is injected into data regions and then executed.
- **Functionality**: It marks certain areas of memory as non-executable, which means that even if an attacker manages to insert code into these areas, it cannot be executed. Only designated executable memory regions, such as those containing the application code, are allowed to execute instructions.
- **Implementation**: DEP can be implemented at both the hardware and software levels. Hardware-based DEP relies on features provided by modern CPUs, while software-based DEP can be enforced by the operating system.

W^X (Write XOR Execute)

- **Definition**: W^X stands for "Write XOR Execute," which is a principle related to memory protection. It enforces that memory regions can either be writable or executable, but not both at the same time.
- **Functionality**: This principle ensures that if a memory region is writable (i.e., data can be written to it), it cannot be executed as code. Conversely, if a memory region is executable, it cannot be written to. This prevents attackers from injecting and then executing code in the same memory region.
- **Implementation**: W^X is implemented through memory protection mechanisms provided by the operating system and hardware. It is closely related to DEP and helps in mitigating certain types of attacks that exploit memory vulnerabilities.

Together, DEP and WˆX help to mitigate risks associated with buffer overflow attacks and other exploits that rely on executing arbitrary code from non-executable regions of memory.



Address Space Layout Randomization (ASLR)

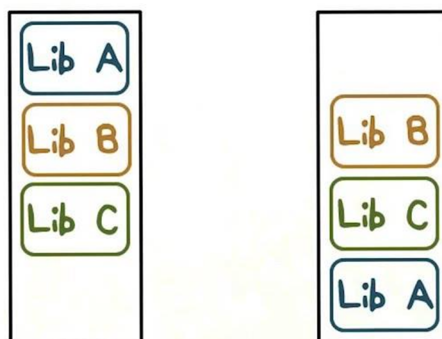
Address Space Layout Randomization (ASLR) is a security technique used to protect systems from certain types of attacks by randomizing the memory layout of a process. Here's a detailed overview:

Address Space Layout Randomization (ASLR)

- **Definition**: ASLR is a security feature that randomizes the memory addresses used by system and application processes, including the locations of executable code, libraries, heap, and stack.
- **Functionality**:
  - **Randomization**: By randomizing memory addresses, ASLR makes it more difficult for attackers to predict where specific code or data is located in memory. This is particularly useful in preventing attacks that rely on knowing the location of specific functions or buffers.

  - **Mitigation**: ASLR helps mitigate several types of exploits, including buffer overflows and return-oriented programming (ROP) attacks, which often require knowledge of specific memory addresses to be effective.
- **Implementation**:
  - **Kernel-Level**: The operating system kernel is responsible for implementing ASLR. It modifies the base addresses of various memory segments each time a process is started.
  - **User-Level**: ASLR can also be applied to user-space applications, randomizing the base addresses of executable and shared library code.
- **Configuration**:
  - **Granularity**: The degree of randomization can vary, with some systems offering fine-grained control over how different memory regions are randomized.
  - **Operating System Support**: Modern operating systems, including Windows, Linux, and macOS, support ASLR. Configuration options may vary, and administrators can typically enable or adjust ASLR settings through system configuration or security policies.
- **Effectiveness**:
  - **Enhanced Security**: ASLR enhances security by making it harder for attackers to exploit vulnerabilities that rely on predictable memory layouts.
  - **Combined with Other Mitigations**: ASLR is most effective when used in conjunction with other security measures, such as Data Execution Prevention (DEP) and Control-Flow Integrity (CFI).

Overall, ASLR adds a layer of defense by complicating the attacker's ability to guess memory addresses, thereby reducing the likelihood of successful exploits based on predictable memory layouts.



Stack integrity

Stack Integrity is a security mechanism aimed at protecting the stack memory from corruption, particularly from attacks like buffer overflows. Here's an overview of what it involves:

Stack Integrity

Definition: Stack integrity refers to techniques and mechanisms designed to ensure that the data on the stack remains uncorrupted and that the stack is protected against malicious attacks. The stack is used for managing function calls, local variables, and return addresses.

Key Techniques:

Stack Canaries:

A stack canary (or stack guard) is a known value placed on the stack between the buffer and the return address. When a function is called, a canary value is placed on the stack. If an attacker attempts to exploit a buffer overflow

and overwrite the return address, they would also need to overwrite the canary. Before returning from the function, the program checks if the canary value has been altered. If it has, the program detects the tampering and typically terminates, preventing further exploitation.

Stack Smashing Protection (SSP):

SSP is a broader term for techniques designed to prevent stack buffer overflows from compromising control flow. It includes methods like stack canaries and other mechanisms to detect and mitigate stack-based buffer overflow attacks.

Non-Executable Stack:

Marking the stack as non-executable prevents code from being run from the stack. This is part of Data Execution Prevention (DEP). Ensures that even if an attacker manages to inject code into the stack, it cannot be executed.

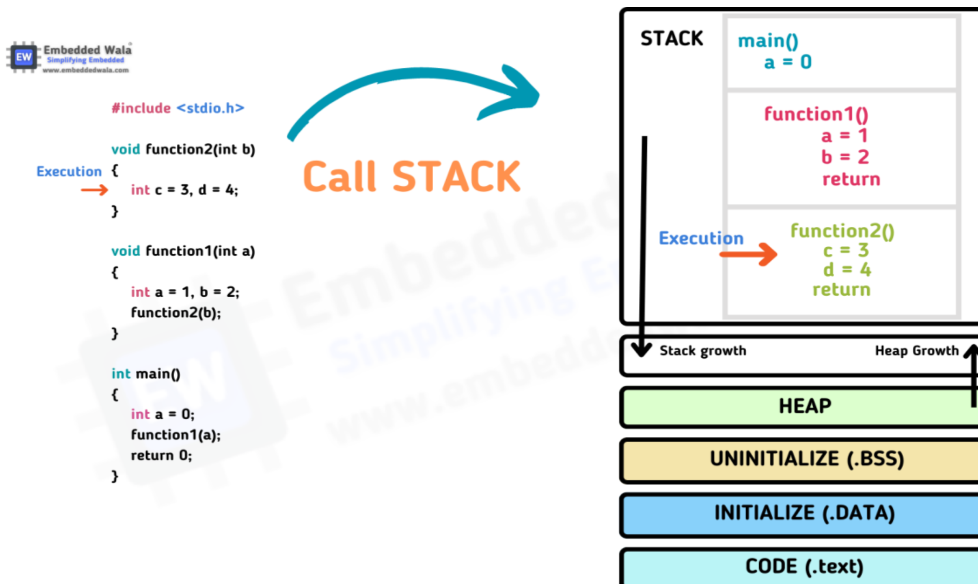
Control Flow Integrity (CFI):

While not specific to the stack, CFI helps ensure that the program's control flow adheres to legitimate paths. Protects against attacks that try to divert execution to malicious code, including those that could exploit stack vulnerabilities.

Effectiveness:

Mitigation of Buffer Overflows: Stack integrity mechanisms, particularly stack canaries, are effective at mitigating buffer overflow attacks by detecting and preventing attempts to overwrite critical stack data.

Combined Measures: For enhanced protection, stack integrity should be used in conjunction with other security features like ASLR, DEP, and secure coding practices.



Safe Exception Handling (SEH)

Safe Exception Handling (SEH) is a technique designed to enhance the security and reliability of software by managing exceptions in a way that mitigates potential vulnerabilities. Here's a detailed overview:

Safe Exception Handling (SEH)

Safe Exception Handling (SEH) involves implementing mechanisms to handle exceptions (unexpected errors or events during program execution) in a secure manner. The goal is to prevent attackers from exploiting vulnerabilities in the exception handling process to gain control over the application.

Key Concepts:

1. **Exception Handling**:

▪ Exception handling is a programming construct used to detect and manage runtime errors. It involves using try-catch blocks (or equivalent constructs) to handle exceptions gracefully. When an error occurs, the program transfers control to a predefined exception handler that can log the error, clean up resources, and prevent the application from crashing.

2. **SEH Mechanisms**:

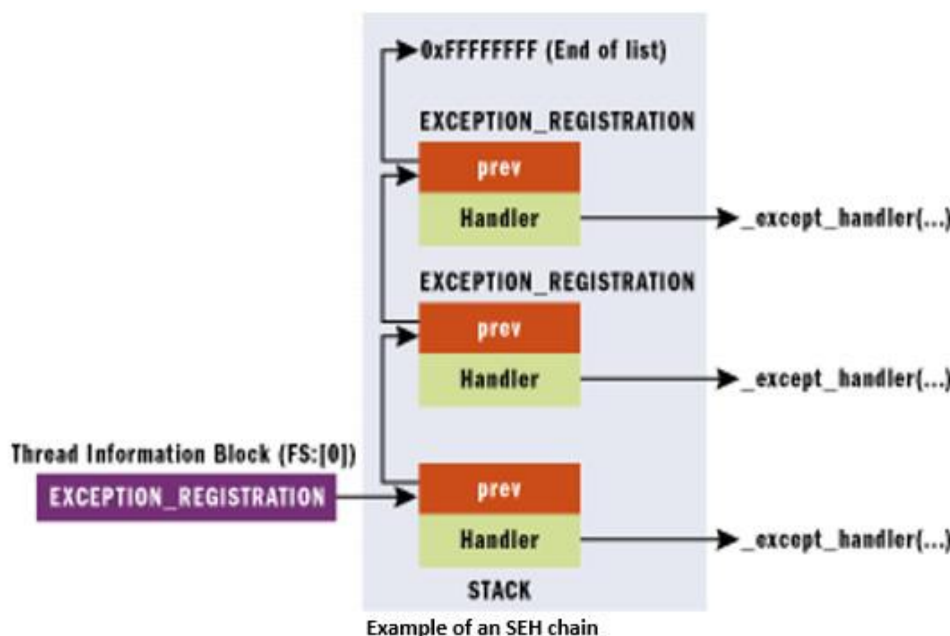
- **Exception Handlers**: Secure exception handlers are designed to handle exceptions in a way that minimizes the risk of exploitation. They should not expose sensitive information or allow attackers to control the flow of execution.
- **SEH Protection**: Modern operating systems and compilers implement protections to prevent exploitation of vulnerabilities in exception handling. For example, Microsoft Windows includes SEHOP (SEH Overwrite Protection), which helps protect against attacks that exploit the SEH mechanism.

3. **SEH Overwrite Protection (SEHOP)**:
   SEHOP is a security feature in Windows that protects against attacks that exploit the exception handling mechanism by overwriting exception handler pointers. SEHOP ensures that the exception handling pointers are not tampered with, making it more difficult for attackers to redirect execution flow through malicious exception handlers.
4. **Stack Frame Integrity**:

Stack frame integrity mechanisms ensure that the stack, where exception handling information is stored, is protected against corruption. Techniques like stack canaries and integrity checks can help prevent attacks that attempt to exploit stack-based vulnerabilities to bypass exception handling.

- **Effectiveness**:

1. **Mitigation of Exploits**: SEH and SEHOP are effective at mitigating attacks that exploit vulnerabilities in exception handling, such as those that involve manipulating exception handler pointers or corrupting stack frames.
2. **Combined Measures**: For enhanced security, SEH should be used in conjunction with other security measures like ASLR, DEP, and stack integrity techniques.



Example of an SEH chain

Fortify Source

Fortify Source refers to techniques and tools used to improve the security of software by enhancing the source code with additional checks and protections. This term is often associated with practices and tools that help identify and mitigate common vulnerabilities during the development process. Here's a detailed overview:

Fortify Source involves applying security enhancements and checks to the source code during the development phase to identify and address potential vulnerabilities before the code is compiled and deployed.

- **Key Concepts**:

1. **Static Code Analysis**:

Static code analysis involves examining the source code without executing it to detect potential security issues, such as buffer overflows, SQL injection, and other vulnerabilities. Tools like Fortify Static Code Analyzer, SonarQube, and Checkmarx can analyze source code for common security flaws and provide recommendations for remediation.

2. **Compiler-Based Protections**:

Many modern compilers offer options to fortify source code by adding runtime checks and protections that can help detect vulnerabilities.

Microsoft Visual Studio: Provides options like /GS to enable stack protection and /SAFESEH for safe exception handling.

3. **Secure Coding Standards**:

Adhering to secure coding standards helps prevent common programming errors that can lead to vulnerabilities. Standards like those from OWASP or CERT provide guidelines for writing secure code.

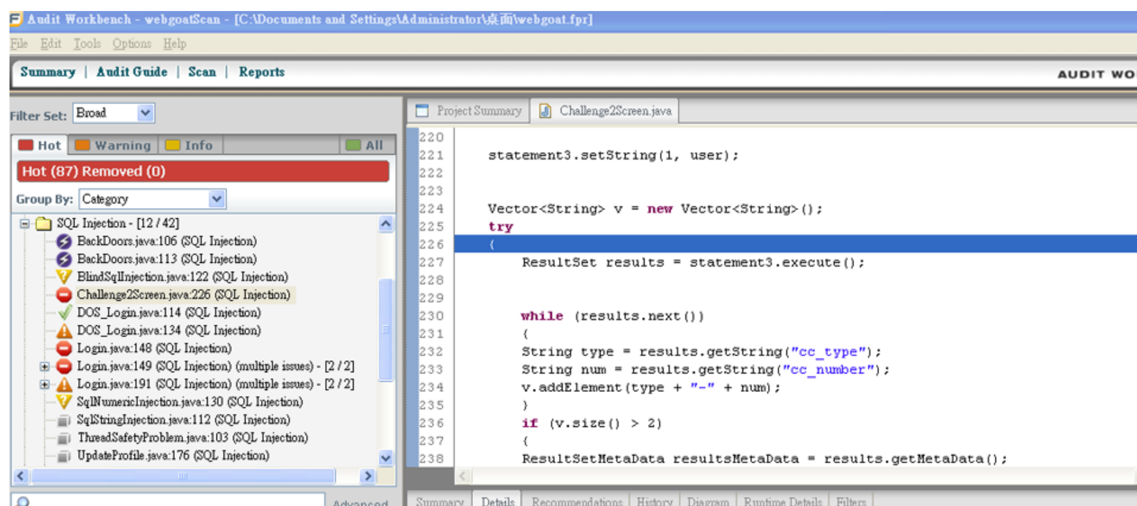
Examples include input validation, proper error handling, and avoiding unsafe functions.

4. **Code Reviews**:

Regular code reviews involve manual inspection of the source code by developers or security experts to identify and address security issues. Code reviews can be complemented by automated tools and static analysis to ensure comprehensive coverage of potential vulnerabilities.

- **Effectiveness**:

1. **Early Detection**: Fortify Source techniques help detect vulnerabilities early in the development process, reducing the risk of security issues reaching production.
2. **Enhanced Security Posture**: By incorporating security checks and following best practices, developers can build more secure applications and reduce the likelihood of security breaches.



Control-Flow Integrity

Control-Flow Integrity (CFI) is a security technique designed to protect the control flow of a program from being altered by attackers. By ensuring that the program executes only through legitimate paths, CFI helps to prevent exploits that rely on redirecting execution to malicious code. Here's an in-depth look at CFI:

Control-Flow Integrity (CFI)

Control-Flow Integrity is a security mechanism that enforces that the execution of a program adheres strictly to its intended control flow. This means that the program should only follow paths that are defined by its legitimate control flow graph, preventing unauthorized redirection of execution.

- **Key Concepts**:

1. **Control Flow Graph (CFG)**:

A control flow graph is a representation of all possible execution paths of a program. It shows how control moves between different instructions or functions. CFI uses the CFG to verify that the program's execution follows only the legitimate paths defined in this graph.

2. **Indirect Control Flow**:

Indirect control flow refers to scenarios where execution jumps to a target address determined at runtime, such as through function pointers, virtual table pointers, or return addresses. CFI focuses on protecting indirect control flow by ensuring that jumps or calls only target valid, intended locations.

3. **Instrumentation and Checks**:

CFI typically involves instrumenting the program to include additional checks that validate control flow at runtime. These checks ensure that function calls, returns, and jumps adhere to the legitimate control flow paths defined by the CFG.

4. **Types of CFI**:

- **Basic CFI**: Ensures that indirect control transfers (e.g., function calls through pointers) are made to legitimate addresses as defined in the CFG.
- **Advanced CFI**: Includes more granular checks, such as verifying the integrity of return addresses and preventing more sophisticated attacks like return-oriented programming (ROP).

- **Effectiveness**:

1. **Mitigation of Exploits**: CFI is effective at mitigating various types of exploits, including buffer overflows, format string vulnerabilities, and return-oriented programming attacks, by ensuring that execution cannot be diverted to malicious code.
2. **Complementary Measures**: For maximum security, CFI should be used in conjunction with other security features like Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), and stack integrity.

- **Challenges**:

1. **Performance Overhead**: Implementing CFI can introduce performance overhead due to the additional runtime checks and instrumentation.
2. **Complexity**: More advanced forms of CFI may require additional complexity in program analysis and implementation.

Code Pointer Integrity

Code Pointer Integrity (CPI) is a security technique designed to protect the integrity of pointers used to control the execution of code in a program. By ensuring that code pointers (such as function pointers) are not tampered with, CPI helps to prevent attacks that exploit vulnerabilities related to code execution.

Code Pointer Integrity (CPI)

Code Pointer Integrity is a security mechanism that ensures the pointers used to control the execution of code (e.g., function pointers, vtable pointers) point to legitimate and expected locations. This prevents attackers from redirecting execution to malicious code.

- **Key Concepts**:

Code Pointers:

Code pointers are pointers that refer to executable code or functions within a program. They are used for dynamic function calls, callbacks, and other

forms of indirect code execution. Function pointers, virtual table pointers (vtable), and jump tables.

Pointer Tampering:

Pointer tampering refers to attacks that modify code pointers to redirect execution to arbitrary or malicious code. This can be achieved through techniques like buffer overflows or format string vulnerabilities. CPI aims to detect and prevent such tampering by ensuring that code pointers are only set to valid and expected addresses.

Integrity Checks:

CPI involves implementing checks to validate the integrity of code pointers during runtime.

These checks can verify that pointers have not been altered from their legitimate values and can prevent unauthorized modifications.

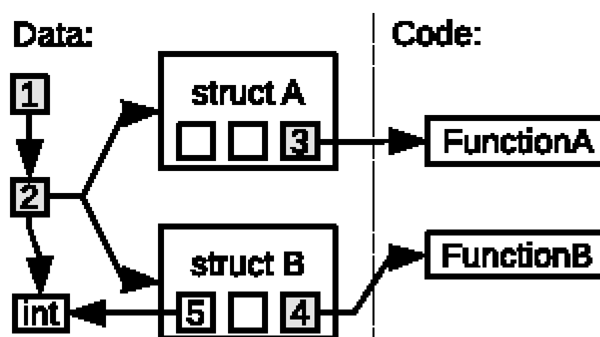
1. **Implementation Methods**:

- **Randomization**: Using techniques like Address Space Layout Randomization (ASLR) to randomize the locations of code pointers, making it harder for attackers to predict and exploit them.
- **Pointer Authentication**: Some modern CPUs support pointer authentication, which adds a cryptographic signature to pointers to verify their authenticity and integrity.
- **Compiler and Runtime Protections**: Compilers and runtime environments can include additional checks and protections for code pointers. For example, certain compilers provide options to enhance code pointer protection.

- **Effectiveness**:
  1. **Mitigation of Exploits**: CPI helps mitigate various types of attacks that rely on tampering with code pointers, such as function pointer attacks, vtable corruption, and return-oriented programming (ROP) attacks.

2. **Complementary Measures**: CPI should be used in conjunction with other security features like Control-Flow Integrity (CFI), Data Execution Prevention (DEP), and Address Space Layout Randomization (ASLR) for comprehensive protection.

- **Challenges**:
  1. **Performance Overhead**: Implementing CPI can introduce performance overhead due to the additional integrity checks and runtime protections.
  2. **Complexity**: Ensuring comprehensive protection of code pointers requires careful analysis and integration into the development and runtime environments.



Sandboxing and Software-based Fault Isolation

Sandboxing and **Software-Based Fault Isolation (SFI)** are security techniques designed to contain and isolate potentially untrusted or vulnerable code, preventing it from affecting other parts of the system. Here's a detailed overview of each:

Sandboxing is a security practice where applications or processes are executed in a restricted environment (sandbox) that limits their access to system resources and other applications. This containment prevents potentially malicious or faulty code from causing harm to the rest of the system.

1. **Isolation**:

- **Definition**: Sandboxes provide isolation between the application and the host system, restricting the application's ability to access files, network resources, and system functions.
- **Functionality**: By isolating an application within a sandbox, even if the application is compromised, the damage is contained within the sandbox environment.

2. **Controlled Access**:

- **Definition**: Sandboxes control and monitor the application's interactions with the operating system and other applications.
- **Functionality**: This control ensures that the application only has access to resources it is explicitly allowed to use, reducing the risk of exploitation.

3. **Types of Sandboxes**:

- **Process Sandboxing**: Runs applications in separate processes with restricted permissions. For example, web browsers often use process sandboxing to isolate tabs and plugins.
- **Containerization**: Uses containers (e.g., Docker) to encapsulate applications and their dependencies, providing an isolated environment for execution.
- **Virtual Machines**: Provides a more comprehensive form of isolation by running applications in a separate virtualized operating system instance.

- **Implementation**:

1. **Operating System Support**: Many modern operating systems support sandboxing natively. For example, Windows has built-in features like

Windows Defender Application Guard, and macOS uses app sandboxing for applications from the App Store.

2. **Security Tools**: Tools and frameworks like AppArmor, SELinux, and various container technologies provide sandboxing capabilities.

- **Effectiveness**:

1. **Containment**: Sandboxing effectively contains potential threats, preventing them from affecting the host system or accessing sensitive data.
2. **Mitigation of Zero-Day Exploits**: By isolating untrusted code, sandboxes can mitigate the impact of zero-day exploits and other vulnerabilities.

Software-Based Fault Isolation (SFI)

- **Definition**: Software-Based Fault Isolation is a technique that isolates the execution of potentially untrusted or faulty code within a process to prevent it from interfering with the rest of the system. Unlike hardware-based isolation, SFI relies on software mechanisms to enforce isolation.
- **Key Concepts**:

1. **Memory Protection**:
   - **Definition**: SFI enforces memory protection by ensuring that code can only access and modify specific regions of memory.
   - **Functionality**: This prevents untrusted code from accessing or modifying memory locations outside its allocated region.
2. **Runtime Checks**:
   - **Definition**: SFI often includes runtime checks to validate that memory accesses and control transfers are within permitted regions.
   - **Functionality**: These checks help ensure that even if code is compromised, it cannot perform unauthorized actions.
3. **Instruction-level Isolation**:
   - **Definition**: SFI can use instruction-level techniques to ensure that only safe operations are performed.

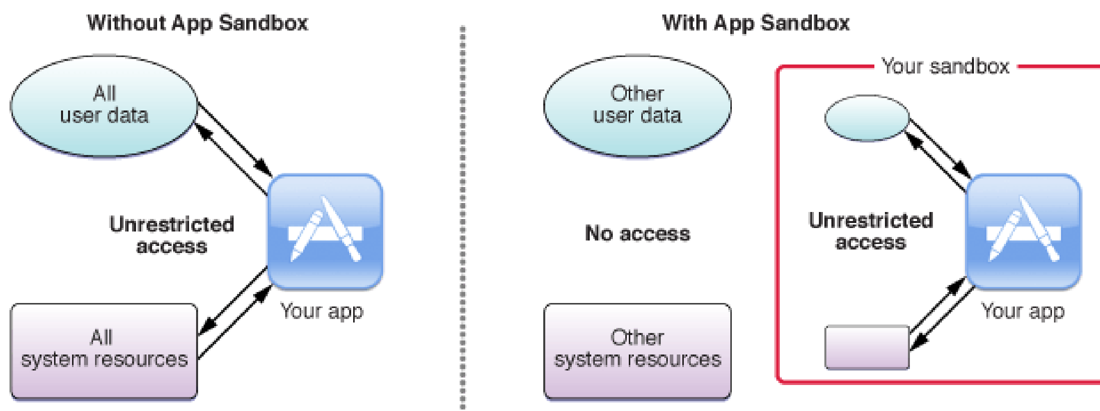
- **Functionality**: For example, instructions that could lead to unsafe memory accesses might be restricted or altered to enforce isolation.

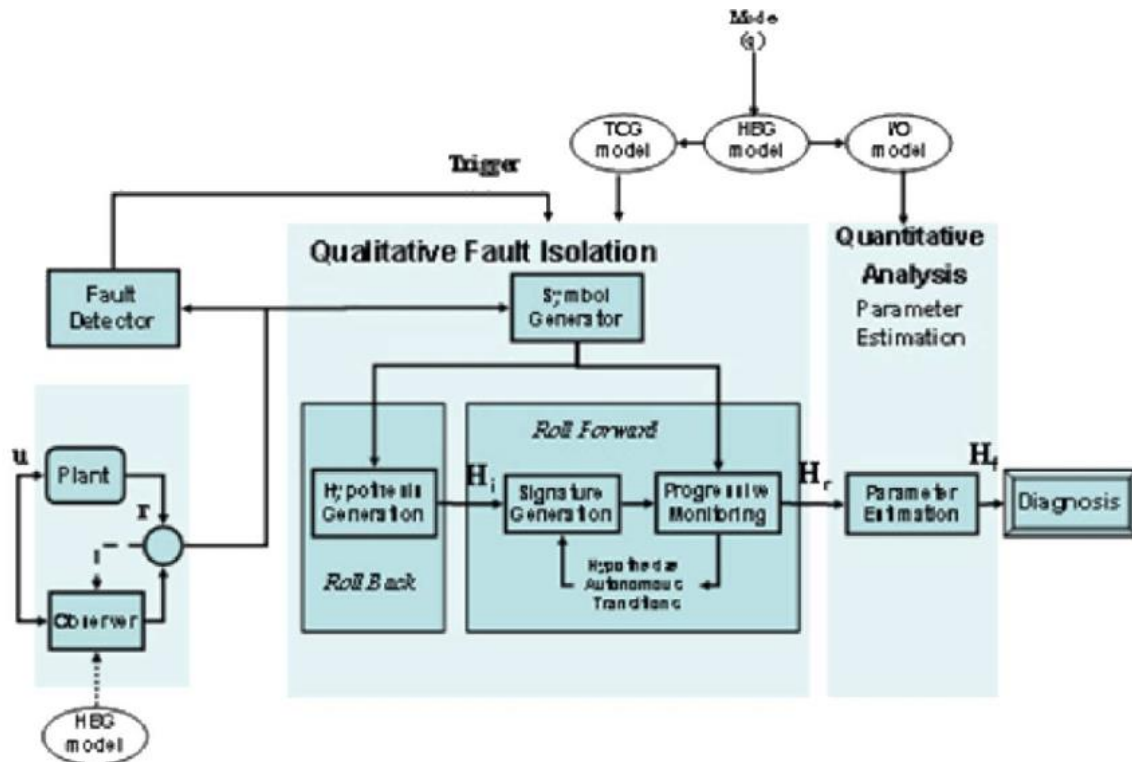
- **Implementation**:

1. **Compiler Support**: Some compilers and runtime environments include support for SFI. For example, Google's NaCl (Native Client) and the more recent WebAssembly use SFI principles to ensure safe execution of untrusted code.
2. **Security Models**: SFI is often used in conjunction with other security models, such as sandboxing, to provide layered protection.

- **Effectiveness**:

1. **Fault Containment**: SFI helps in containing faults and preventing them from propagating or causing widespread damage.
2. **Security Enhancement**: By ensuring that code operates within a controlled environment, SFI enhances the security of systems running potentially untrusted code.




TCG model
HBG model
I/O model
Trigger
Qualitative Fault Isolation
Quantitative Analysis
Parameter Estimation
Fault Detector
Symbol Generator
Roll Forward
u
Plant
r
Hypothesis Generation
Signature Generation
Progressive Monitoring
Parameter Estimation
Diagnosis
Roll Back
Observer
HBG model