Security Testing

Davide Donà

September 2025

Contents

1 Introduction

1.1 Definitions

We are now going to define some important terms that will be used throughout the document.

Software: software can be defined as a set of **programs** and **data**, that provides **functionality**. Software and functionality always come with certain risks.

Security: **Understanding** and **identifying** security risks, induced by a software, and how to **mitigate them**.

Secure software: a software is **considered secure** if it satisfies a series of security objectives, such as:

- **Confidentiality**: **unauthorized users** cannot **have access** or read information;
- **Integrity**: **unauthorized users** cannot **change** the information;
- **Availability**: **authorized users** can always have access to the information.

There are many other security objectives. Each software doesn't have to implement each of them, but it's useful to find a good **trade-off** between **security** and **functionality**.

Safety vs Security: there's a slight difference between the two definitions, that is important to denote:

- **Safety**: protecting the system from **accidental risks**;
- **Security**: **mitigating** the risk of danger caused by **intentional** or **malicious actions**.

Weakness: a condition in a software (bug) that, under certain circumstances, could contribute to the introduction of vulnerabilities.

Vulnerability: a chain of **weaknesses** linked by causality. If **exploited** it leads to a **security failure**.

Security risk: refers to the **probability** that a threat (any external actor) will **exploit a vulnerability**, combined with the **resulting impact** (damage).

CWE: **Common Weakness Enumeration**, provides a **public list** of the most common software and hardware **weaknesses** that can contribute to security vulnerabilities.

CVE: **Common Vulnerabilities and Exposure**, is a public list of **vulnerabilities** in widely-used software components.

1.2 Software Development Life Cycle

The **SDLC** is the series of **processes** and **procedures** that enables teams to **create software** and applications. The cycle is composed of various phases:

- **Planning**: the senior members of the team, with the inputs from the customers, **plans** the basic **project plan**;
- **Defining**: during this phase, the document for the **product requirements** should be defined;
- **Designing**: designing the architecture for the whole project;
- **Building**: generating the programming code. The output should be the working final system;
- **Testing**: during this phase, products defects are reported, tracked and fixed;
- **Deployment**: the final release. Feedback from the users is usually collected, to better the software.

1.2.1 From a security perspective

Now, we are going to analyze, from a **security perspective**, what should be done by a team during each of the just mentioned phases.

Planning: During this phase, the **security objectives** required by the software should be defined. The organization should also identify the regulations it needs to follow.

Defining: This phase is composed of three main activities:

- **Gap Analysis**: identify whether there is a need to provide training sessions (security and security awareness training) for the developers involved in the project;
- **Risk Assessment**: analyze potential threats and vulnerabilities, and evaluate their possible impact on the project;
- **Third-party software tracking**: a list containing both open-source and commercial **third-party software** must be defined. This allows the team to check the used software for updates and security patches.

Designing:

- **Least-privilege principle**: the program should always run on the account with the least privilege possible;

- **Deny by default**: each access should be denied by default, unless explicitly allowed.
- **Design Review**: the developer has to act as an attacker to discover vulnerabilities in the defined features.

Building:

Security guidelines: While the project is being built, is always important to follow the security guidelines, such as:

- Always validate and check the input data and memory;
- Check the command line arguments;
- Use external files to protect passwords;
- Deal with errors and exception;
- Avoid the creation of files when possible. Otherwise, be caution when handling them.

There are some frameworks, such as **OWASP** that can help in the secure coding rule knowledge.

Code review: conducted by a team of developers, that manually checks the code, focusing on implementation bugs. Usually, a checklist is used for checking secure coding.

Security Testing: mainly composed of two isolated phases:

- **Static Analysis**: tries to identify weaknesses, without executing the program;

Testing: Other testing should be done once the final product has been developed. Different types of tests are usually run, such as:

- **Dynamic Analysis**: identifies weaknesses by running software;
- **Fuzzing**: involves giving random data to a program. It helps in finding bugs that humans often miss;
- **Third-party penetration testing**: an attack simulation on the third-party software, with the intention to discover configuration flaws and vulnerabilities

Regardless, bugs should always be fixed ASAP, to reduce the cost of fixing them.

1.2.2 DevOps

The **DevOps** cycle is based on two main ideas: **rapid delivery** of new software and **quick feedback**. Developers continuously deliver new features in a short time span. Developers regularly merge their code changes in a central repository, where it gets automatically built, tested, and prepared for a release to production.

2 Code Weaknesses and Vulnerabilities

2.1 Most common Weaknesses

We now present a selection of the **most dangerous software weaknesses**, to illustrate common coding patterns that lead to security vulnerabilities and to guide analysis and testing efforts. Knowing this classification can help us with:

- **conducting an effective testing activity**: we know what to search while testing, so we can prioritize some issues, over others;
- **improving the development process**: know solutions (ex: patterns, guidelines) to help developers improve their software, avoiding the introduction of weaknesses;
- **driving bug-fixing**: prioritize the issues to solve, helping also to understand how to fix it.

2.1.1 Use-After-Free(CWE-416)

Description: The software **reuses** or **references memory**, after it has been **freed**. At some point afterwards, the memory may be allocated to another pointer. Any operation using the original pointer is no longer valid.

```
1  char* ptr = (char*)malloc (SIZE);
2  if(err){
3      abrt = 1;
4      free(ptr);
5  }
6  if(abrt){
7      //Assessing a pointer
8      logError("operation aborted before commit", ptr);
9  }
```

UAF can be found also when other resources, such as **file** or **network connections** are **mismanaged**.

Prevention: Once freed, all pointers should be set to **NULL**.

Detection: Using **static analysis**.

2.1.2 Heap-based Buffer Overflow(CWE-122)

Description: A particular case of the **buffer overflow**, where the buffer is allocated in the **heap** portion of the memory. This means that the buffer was allocated using **malloc()**.

```
1  #define BUFSIZE 4
2  int main(int argc, char **argv) {
3      char *buf;
4      buf = (char *)malloc(sizeof(char)*BUFSIZE);
5      strcpy(buf, argv[1]);
6  }
```

The problem here is caused by the function `strcpy()`, which has no limits on the length of the element that should be copied to the buffer.

Prevention: Other equivalents, but **safer** functions should be used. For example `strncpy()` takes as an argument the maximum number of characters that should be copied.
Also, automatic buffer overflow detection mechanisms can be used.

Detection: Using **static analysis**.

2.1.3 Out-of-bounds Write(CWE-787)

Description: The software writes data past the end, or before the beginning of the intended buffer.

```
1  int id_sequence[3];
2
3  id_sequence[0] = 123;
4  id_sequence[1] = 234;
5  id_sequence[2] = 345;
6  id_sequence[3] = 456;
```

Prevention: Always verify that the buffer is as large as specified and that its boundaries are respected.

Detection: Using **static analysis** and **dynamic analysis**

2.1.4 Improper Input Validation(CWE-20)

Description: The software receives input values, but it **does not validate** or **incorrectly validates** them. In case of no-validation, unintended input values can **alter the program control flow**.

```
public static final double price = 20.00;
int quantity = currentUser.getAttribute("quantity");
double total = price * quantity;
chargeUser(total);
```

In this example, if the user inputs a negative number for the quantity, it can lead to an account credit instead of a debit.

Prevention: Always assume that all inputs are malicious. An input validation framework can be used to check all the untrusted inputs.

Detection: Using **static analysis** and **dynamic analysis**

2.1.5 Improper Neutralization of Special Elements used in an OS Command(CWE-78)

Description: The software constructs an **OS command** using an **external input**, but it does not neutralize special elements that could modify the intended OS command.

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

This code takes the name of a user and lists the content of the user directory. There is no check on the variable `$userName`, that could contain an **arbitrary OS command** such as: `\;rm -rf /"`. Since the `;` works as a command separator in Linux, such input would delete the entire file system.

Prevention: Use library calls rather than external processes and try to block the user from using semi-colons or other special characters.

Detection: Using **dynamic analysis**

2.1.6 De-serialization of Untrusted Data(CWE-502)

Description: The software deserializes untrusted data without verifying that the resulting data will be valid.

```
try {
    File file = new File("object.obj");
    ObjectInputStream in = new ObjectInputStream(new
        FileInputStream(file));
    javax.swing.JButton button = (javax.swing.JButton)
        in.readObject();
    in.close();
}
```

The code deserializes an object from a file, received from an UI button and does not verify the source or contents of the received file.

Prevention: When deserializing data, a new Object should be populated, rather than just deserializing it.

Detection: Using **static analysis**

2.1.7 Server-Side Request Forgery(CWE-918)

Description: A security vulnerability in a **server A** allows an attacker to cause the **server-side application** to make requests to an unintended location (i.e., **another server B** or service).
The **attacker's request** is **sent from the back end of the server**, so the **target service believes it came from a legitimate source**. Attackers can **bypass access controls** (i.e., firewalls) that **prevent direct attacks on the target server** by exploiting trusted relationships between servers. The compromised server can also be used as a proxy to conduct port scanning of internal hosts or to access restricted URLs and documents.

Prevention:

- Limit outbound traffic from the application server.
- Build a whitelist of trusted domains and IP addresses.
- Sanitize user input to prevent potential risks.

Detection: Automated static analysis

2.1.8 Access of Resource Using Incompatible Type(CWE-843)

Description: The software initializes a resource such as a pointer, object or variable, using **one type**, but later accesses that resource using a type that is incompatible with the original type. Type confusion can lead to out-of-bounds memory access.

```
1  $value = $_GET['value'];
2  $sum = $value + 5;
3  echo "value parameter is '$value'<p>";
4  echo "SUM is $sum";
```

An attacker could supply an input string such as `value[] = 123`. From that point, value is treated as an array type, causing an error when the sum is calculated.

Prevention: Attention to implicit and explicit type conversion.

Detection: Using **static analysis**

2.1.9 Improper Limitation of a Pathname to a Restricted Directory(CWE-22)

Description: The product uses an **external input** to **construct a pathname** for a file or directory, located into a restricted parent directory. Without the neutralization of special elements, the path can relate to a different location, not supposed to be accessible.

```
1  String path = getInputPath();
2  if (path.startsWith("/safe_dir/")){
3      File f = new File(path);
4      f.delete()
5  }
```

Prevention:

- Always assume all input is malicious;
- When the set of acceptable object, such as filenames or URLs is limited and known, create a mapping from a set of **fixed input values** to the **actual filenames** or **URLs**, rejecting all the other inputs.

Detection: Using **static analysis** or **dynamic analysis**.

2.1.10 Missing Authentication for Critical Function

(CWE-306)

Description: The product **does not perform any authentication** for a functionality that requires a provable user identity.

Prevention: Where possible, avoid using custom authentication routines. Instead, consider using authentication capabilities as provided by libraries or frameworks.

Detection: Using **static analysis** or **dynamic analysis**, such as **vulnerability scanning**.

2.2 Most common security risks

OWASP is an open source project, providing guides, guidelines and suggestions to develop and test secure applications. Its Top-10 lists the most critical security risks for web applications.

2.2.1 Broken Access Control (A01-2021)

Description: The **access control** enforces policy such that **users cannot act outside their intended permissions**. Its failure leads to **unauthorized information** disclosure, modification, or destruction.

Example: The application uses unverified data in SQL to access information:

```
//Trying to access using this url
//https://example.com/app/accountInfo?acct=myacct
PreparedStatement pstmt = connect.prepareStatement(query);
pstmt.setString(1, request.getParameter("acct"));
//myacct is a "reference" to an account
ResultSet results = pstmt.executeQuery( );
```

If an attacker modifies the browser's **acct parameter** with whatever account number they want, the attacker can access any other user's account.

Prevention:

- Deny by default;
- Access control is effective only in truster server-side code;
- **Access control unit should be included in the tests**.

2.2.2 Cryptography Failures (A02-2021)

Description: Violation of the protection needed for exchanged data. Data can't be transmitted in clear text, but should always be encrypted, using properly set cryptographic algorithms.

Example: An application correctly encrypts credit card numbers in a database before storing them. Data is automatically decrypted once retrieved, allowing an SQL injection to retrieve all the information in clear text.

Prevention: Proper encryption, with a proper setting of the used cryptographic mechanism.

2.2.3 Injection (A03-2021)

Description: An application is vulnerable to this type of attack when user input data is not validated by the application before using it as command parameters.

Example:

```
String query = "SELECT \* FROM accounts WHERE custID='" +
    request.getParameter("id") + "";
```

If the attacker manages to set the id parameter for example to *' or '1'='1* the query will return all the records from the account table.

Prevention: Use server-side input validation, checking in particular for escape characters.

2.2.4 Insecure design (A04-2021)

Description: Insecure design concerns weaknesses related to missing or ineffective control design. An usual example could be the use of "questions and answers" for the credential recovery workflow. They in fact cannot be trusted as evidence of user identity.

Prevention: Always adopt a secure design methodology.

2.2.5 Security Misconfiguration (A05-2021)

Description: Refers to a variety of situations, such as:

- Missing appropriate security hardening or improperly configured permissions;
- Unnecessary features installed (ex: ports, accounts, privileges);
- Default account, with default passwords;
- Missing error handling, revealing stack traces or other important information;
- Out of date software.

Prevention: A secure installation process should be implemented. Always review and update configuration.

2.2.6 Vulnerable and Outdated Components (A06-2021)

Description: The component of a software can be vulnerable, unsupported, or out of date.

Prevention:

- Remove unused dependencies, unnecessary features, components and documentation;
- Continuously monitor the versions of both client and server components and their dependencies;
- Obtain components only from official sources;

2.2.7 Identification and Authentication Failures (A07-2021)

Description: **Authentication**, **confirmation** of user's **identity** and **session management** are critical to protect against authentication-related attacks.

Prevention:

- Implement proper identification and authentication processes (ex: MFA, to prevent brute force, stolen credential reuse attacks);
- Do not permit the usage of weak passwords;
- Use a **server-side**, secure, **session manager**.

2.2.8 Software and Data Integrity Failures (A08-2021)

Description:

Prevention:

2.2.9 Security Logging and Monitoring Failures (A09-2021)

Description: It relates to the logging and monitoring capability to detect security breaches. You will make yourself vulnerable to information leakage by making logging and alerting events visible to the users.

Prevention: Ensure proper logging and monitoring.

2.2.10 Server-Side Request Forgery (A10-2021)

Already talked about at 2.1.7.

3 SQL Injection

SQL injection is a vulnerability that allows an attacker to **inject unintended SQL code** into **queries** executed by an application. Any software that constructs SQL queries using external input may be vulnerable to this type of attack.

- **Severity**: very severe;
- **Priority**: often high;
- **Scope**: software that uses SQL;
- **Technical impact**: unauthorized access to database contents and metadata, data modification or deletion, and potential privilege escalation;
- **Worst-case scenario**: full system compromise (data exfiltration, administrative takeover, code execution via database features).

3.1 Underlying idea

The application executes an SQL query where some parameters are taken from user input. For example:

```
SELECT * FROM users WHERE userId = <user_id> AND password =
    <user_password>;
```

If the application naively substitutes user-supplied values into the query, an attacker can craft input that changes the query semantics.

3.2 Example attack

Suppose the application builds the query by concatenation. If an attacker supplies:

```
user_id:    ' OR '1'='1
user_password:    ' OR '1'='1
```

the resulting query becomes:

```
SELECT * FROM users
  WHERE userId = '' OR '1'='1'
  AND password = '' OR '1'='1';
```

Because the boolean expression `'1'='1'` is always true, the WHERE clause may be satisfied for many rows, allowing the attacker to bypass authentication or access data they shouldn't.

3.3 SQL Injection type

In-Band SQL injection In this technique, also called **CLASSICAL**, the attacker uses the same way to hack the database and get the data. The attacker has the possibility to **modify the original query** and receive the result from the database. This is composed of 3 variants:

- **Classic SQL Injection:** basic and traditional version, seen in the previous example;
- **Error-Based SQL Injection:** the attacker makes the database produce error messages that can help to gather information about the database structure.
- **Union-Based SQL Injection:** the attacker uses the `UNION` SQL operator to combine the results of the original query with the results of a malicious query.

Inferential SQL Injection This type of injection, also called **BLIND**, does not show any error message. It's more difficult to exploit, as it returns information only when the application is given SQL payloads that return true or false responses from the server. By observing the responses, an attacker can extract sensitive information.

- **Boolean Based:** the attacker observes the behavior of the database server and the application, after combining legitimate queries with malicious data, using boolean operators;
- **Time Based:** the attacker observes the behavior of the database server and the application, after combining legitimate queries with SQL commands that cause time delays.
- **Out-of-Band:** this technique is used when the attacker is not able to use the same channel to launch the attack and gather information.

3.4 SQL Injection mitigation

- **Escaping** all user supplied input. This means to treat all the data as plain text, rather than executable code or other harmful inputs;
- Define a list of allowed inputs, to ensure only them are executed;
- Use of prepared statements with parametrized queries. Programming languages allow to use predefined structures to compose SQL queries by accepting only values of specified types.

4 XSS (Cross-Site Scripting)

Cross-Site Scripting (XSS) attacks are a type of injection in which **malicious scripts** are injected into trusted websites. The end user's browser has no way to know that the script should not be trusted and will execute the script. It can access any cookies, session tokens or other sensitive information retained by the browser.

4.1 Underlying idea

The problem originates when the **user input** is directly **displayed in an output web page**, without any sanitization. The typical attack follows this flow:

1. The attacker identifies a web site with XSS vulnerabilities;
2. The attacker creates a malicious URL/ script sending malicious input to the attacker website;
3. The attacker tries to introduce the victim (user of the attacked website) to click on the URL;
4. The victim clicks the URL, executing the malicious code. The website response page includes malicious links or malicious JS code, executed on the victim's browser.

4.2 XSS Types

There are three main types of XSS attacks, based on how the malicious script is delivered and executed.

4.2.1 Reflected (non-persistent)

User-supplied input is immediately returned by the web application — for example, in search results, error messages, or redirects — without being properly encoded or escaped for safe rendering in the browser. The payload is not stored on the server; instead, an attacker crafts a URL (or form) containing the payload and convinces a victim (for example, via social engineering) to visit it. When the victim follows the link, the server responds with a page that includes the malicious code, which then executes in the victim's browser.

Example:

```
https://example.com/search?q=<script>script_here</script>
```

The server responds with a webpage that includes the parameter `q` directly; in this case, the script is executed in the victim's browser.

4.2.2 Stored (persistent)

User-supplied input is stored on the target server (for example, in a database, profile field, comment, or message) and later served to other users without proper encoding or sanitization. Because the malicious payload is persisted, any user who views the affected page may receive content that contains executable script, which will run in that user's browser.

Example:

```
<!-- User submits comment containing: -->
<div>Nice post!</div><script>alert('xss')</script>

<!-- Application stores the comment and renders it-->
<div class="comment">
  <div>Nice post!</div><script>alert('xss')</script>
</div>
```

When other users load the page that lists comments, the server includes the stored comment directly in the HTML, and the script executes in each viewer's browser.

4.2.3 DOM-Based

A form of XSS in which the entire unsafe data flow occurs in the browser: client-side JavaScript reads attacker-controlled data (for example from the URL fragment, query string, `window.name`, or `postMessage`) and then unsafely writes it into the DOM, causing execution — without the payload ever being reflected or stored by the web server.

Vulnerable example (client-side)

```
<!-- index.html (static) -->
<div id="msg"></div>
<script>
  // attacker-controlled: location.hash
  const frag = location.hash.substring(1);
  // UNSAFE: inserts raw HTML from the fragment
  document.getElementById('msg').innerHTML = frag;
</script>
```

4.3 XSS mitigation

No single technique can solve XSS, but rather a combination of strategies is required:

- **Treat all input as untrusted:** Any data coming from an external source — including users, third-party services, or even other internal systems — must be treated as untrusted.
- **Contextual escaping / encoding:** Always escape or encode user input according to the context in which it will appear (HTML body, HTML attribute, JavaScript, CSS, URL, etc.). Use the framework's built-in escaping functions whenever possible.
- **Sanitize HTML if necessary:** If user input needs to include HTML (i.e., formatted comments), use a whitelist-based sanitizer that removes unsafe tags and attributes.
- **Server-side validation:** Validate input for expected format, type, or length. Note that validation alone is insufficient; always combine it with encoding or escaping on output.

5 Request Forgery

Before discussing the Request Forgery vulnerability, it's important to understand how **HTTP** and **cookies** work, as they are the basis of this type of attack.

5.1 HTTP and Cookies introduction

HTTP is a protocol that only allows communication using **request** and **responses**. It's considered **stateless**, meaning that natively, it doesn't allow to preserve the state of a user.
Typically, the **state** is usually maintained via the use of **cookies**, sent by the server to the client's browser. They typically contain information to identify the client, but are also used for:

- **Identify and authenticate** the logged on users and the corresponding session;
- **Track** the user actions and navigation;
- **Maintain information** regarding user interaction (ex: item in the shopping cart).

Cookies are **stored in the client-side**. It could be possible to:

- **Predictable information**: attackers may guess session IDs or other values.
- **Compromise or leakage**: confidential data like passwords could be exposed.
- **Long expiration**: cookies may persist too long, increasing risk if stolen.

- **Client-side access**: JavaScript can read/modify cookies if `HttpOnly` is not set.
- **Interception**: cookies can be stolen through proxy or man-in-the-middle attacks if `Secure` is not set (i.e., not restricted to HTTPS).

Session: A session is a frame of **HTTP** communication in which the user **data are stored in the server**, instead of the client. The **session identifiers** are unique numbers, used to **identify every user**. They link the user with their information on the server. Cookies are often used to share the session identifiers.

5.2 Session Hijacking

Session Hijacking is an attack in which an **attacker** takes over a **valid user session**, gaining unauthorized access to the web application.

Scenario:

1. An authorized user **logs** into a website;
2. The web server generates a **session ID** and sends it to the client as a **cookie**;
3. While the user is connected, the attacker manages to **steal** the **session ID**;
4. The **attacker** uses the stolen session ID to **impersonate** the **authorized user**.

How? There are many ways in which an attacker could steal session IDs:

- **Traffic sniffing**: attackers can **intercept** HTTP communications between the web server and a client. This is typically defined as a **Man In the Middle** (MITM) attack and it's only feasible if cookies are sent over using plain HTTP (No HTTPS);
- **Session fixation**: attackers can forge a valid, but unused session ID, and then convince an authenticated user into using that ID. The attacker can then access to the user's session;
- **Malware / Trojan injection**: attacker can use a malware to monitor the user activity and steal data, such as the session IDs;
- **XSS**: attacker can exploit the XSS vulnerability to leak the cookie information;
- **Brute force**: attackers can predict the value of the cookie.

Prevention

- Use **encrypted** communication channels, to prevent MITM attacks;
- Don't pass the session IDs as a URL parameter;
- When defining a cookie, this guidelines should be followed:
  - Use an **adequate algorithm** to generate session IDs. They should have an adequate length and randomness;
  - Set a short **expiration time**;
  - Set HttpOnly parameter to true (no access in the cookie in the client);
  - Set Secure to true (cookie is only transmitted via secure channels).

5.3 Cross-Site Request Forgery

Cross-Site Request Forgery(CSRF) is an attack in which an attacker tricks a victim's browser into **sending unauthorized requests** to a **trusted web application** in which the victim is already authenticated.

The user is lured into visiting a malicious website through social engineering. That malicious website causes the victim's browser to send crafted requests to the target application. These requests inherit the victim's identity and permissions, allowing the attacker to perform actions on behalf of the victim (i.e., changing account settings, performing transactions).

In this attack, the victim is effectively used as a proxy between the attacker and the web server. It's typically used to perform legitimate, but unauthorized actions on the legitimate user account.

Fixing CSRF: Anti-CSRF token is a secure, random value generated by the server and inserted into forms or requests to prevent unauthorized actions. The basic idea is that, for every sensitive action, the server must verify that the request was intentionally generated by the authenticated user and not by an attacker.
Also, we could possibly protect this type of actions by prompting another login form. This way, only the authorized user can access those pages.

5.4 Server-Side Request Forgery

SSRF attack involves an attacker who can **abuse server functionality** access or modify resources.

- An attacker can send a request from the backend of the software, to another server;
- The server that receives the request believes that the request came from the application, and it's legitimate;

It's based on the existence of a trust relation between the servers and of a vulnerability, that allows attackers to abuse such trust relation. The attacker uses the server as a proxy to access data stored in the same server, but not accessible by common users.

Example An application queries a server-internal service to obtain current weather forecasts. The application passes a URL containing the API request from the browser to the server.

```
POST /forecasts HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 250

weatherApi=http://weatherapp.com/forecasts/
check%3FcurrentDateTime%3D6%26cityId%3D1
```

This causes the server to make a request to the specified external URL, retrieve the weather forecast, and return the information to the user.

An attacker can modify the request to specify a URL that points to a local resource on the server itself.

```
POST /forecasts HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 1100

weatherApi=http://localhost/admin
```

The server, executing the request internally, accesses the URL `http://localhost/admin`. This may cause the server to expose sensitive information (i.e., configuration files or admin panels) that are normally inaccessible from outside. Because the request is executed from within the trusted, it bypasses traditional access controls and security boundaries.

Identification To identify this type of issues, it's important to understand how data flows. Determine how user-supplied input (i.e., URL parameters) is propagated to server-side components that perform outbound connections.
To discover this type of issue this methods can be applied:

- **Spider the web:** crawl the application to find forms, endpoints and parameters that may accept URLs.
- **Search requests for URL-like parameters:** look for parameter names such as `url` or `=http`.
- **Inject SSRF payloads:** submit controlled URLs (i.e., pointing to an attacker-controlled domain) and inspect the server's behavior and responses.

Mitigation

- **Validate** request and response also from **internal services**;
- **Add authentication** also for internal and third-party services that don't have it by default;
- Have a **whitelist** of IP addresses allowed to do internal requests;

6 Command Injection

Command Injection is an attack in which an attacker's goal is the **arbitrary execution of commands** on the **host operating system**.

Problem: Untrusted external data is directly passed to a command interpreter. If the data is crafted to include commands, those commands may be executed and the interpreter may be forced to perform actions beyond its intended function.

Underlying conditions: For command injection to succeed, the application typically meets three main conditions:

- the application has the **privileges/permissions** to execute system commands;
- the application uses **user-provided data** as part of a system command;
- the user-provided data is not properly **escaped** or **sanitized** before use.

Mitigation:

- Avoid invoking shell/OS execution functions when possible;
- Treat all external input as untrusted: validate, normalize, and apply **whitelisting** where feasible;
- Use **parameterized APIs** or pass user input as separate arguments rather than concatenating it into command strings.
- Run the application with the **least privileges** necessary to reduce the impact of a successful attack.

7 File access

7.1 File Inclusion

File inclusion vulnerabilities can affect applications that rely on **run-time tasks** that work with **files** (i.e. when an application allows users to access files, submit input into files, upload files to the server).

Local File Inclusion (LFI): LFI attack occurs when an application uses the path to a local file as input. It aims at forcing the application to expose or run files on the server.

Remote File Inclusion (RFI): RFI attack occurs when an application dynamically references external resources, such as files or scripts. It aims at **exploiting the referencing function** to **upload a malware** from a remote URL.

7.2 Directory Traversal:

A path traversal (directory traversal) attack aims to access files and directories that are stored outside the web server's document root. If the web server is misconfigured, an attacker may be able to modify the path in the URL (for example, using sequences like "`../`" or encoded variants) to access files the application did not intend to expose.

7.3 File Binding

Often, file names are used to bind a file to a program object. However, every time a file name is referenced in an operation, this binding is reasserted — meaning that the program may end up operating on a different file if the file name has been changed or replaced in the meantime. Therefore, checks are required to ensure that the intended file is actually being accessed.

It is recommended to access files using **file descriptors** rather than just file names, as they provide a higher degree of certainty about which file object is actually being acted upon.

7.4 TOCTOU (Time Of Check, Time Of Use)

Time Of Check, Time Of Use (TOCTOU) is a race condition that occurs when an application performs:

- a **check operation**, to verify the existence or state of a file at time t;
- a **use operation**, to read or write data at a later time $t + \Delta t$.

If another process modifies or deletes the file between the check and the use, the application may operate on an unintended or invalid file, leading to race conditions or security vulnerabilities.

A more effective solution involves using **atomic operations** and accessing files via **file descriptors**, which ensures that the file being operated on is the same one that was originally opened.

7.5 Mitigation

- Avoid the use of user inputs within the **file system calls**;
- If forced to use user input for file operations, resolve and validate the paths;
- Use file descriptors, rather than filenames.

8 Error Handling

Error handling is a mechanism used to detect and manage errors that occur during the execution of a program. It allows the application to recover gracefully from unexpected conditions and avoid crashes. Proper error handling is an essential part of an application's overall security.

Problem: Improper error handling occurs when software fails to correctly handle certain error conditions, leaving the program in an insecure or unstable state. This can lead to program crashes (potentially causing a **Denial of Service**) or unintentional disclosure of sensitive information such as stack traces, file paths, or internal logic — all of which may help an attacker gather useful insights about the system.

Mitigation:

- Always handle errors and exceptions appropriately, considering the specific context of each case;
- Avoid displaying detailed error messages to end users — log them securely instead;
- Do not silently ignore or mask exceptions, as this can hide critical failures and lead to unpredictable behavior;
- Always check the return value or status code of functions and system calls for possible errors;

9 Insecure Direct Object Reference (IDOR)

Insecure Direct Object Reference (IDOR) is a type of **access control vulnerability** that occurs when an application uses user-supplied input to access internal objects directly, without proper authorization checks. This happens when a developer exposes a reference to internal implementation objects — such as files, database records, or keys — without adequate validation, allowing attackers to manipulate those references and gain unauthorized access to data or functionality.

How:

- **URL tampering:** modifying the value of a parameter directly in the browser's address bar (e.g., changing `user_id=123` to `user_id=124`);
- **Body manipulation:** similar to URL tampering, but the attacker modifies parameters in the request body (e.g., in POST or PUT requests);
- **Path traversal:** already discussed in Section 7.2;
- **Cookie or JSON ID manipulation:** if the application is vulnerable to IDOR, an attacker can alter identifiers stored in cookies or JSON payloads to access other users' data.

10 Client-side vulnerabilities

10.1 Validation

Client-side validation bypass refers to techniques used to circumvent validation checks performed on the **client side**. When the server relies solely on protection mechanisms implemented in the client, an attacker can modify the client's behavior to bypass these mechanisms, leading to unexpected or unauthorized interactions between the client and the server.

How:

- Modifying the HTML code directly (e.g., removing input restrictions or changing field attributes);
- Using JavaScript to alter form data before submission;
- Using proxies or intercepting tools to capture and modify HTTP requests.

Mitigation: All validation performed on the front-end must also be enforced on the backend. Client-side checks are still useful to improve user experience and reduce server load, but they should never be considered a security mechanism.

10.2 Data Filtering

The problem arises when an application sends all the data to the client and performs filtering only on the client side. This approach allows users to access information that should remain hidden, creating a potential access control vulnerability.

It is a good practice to send the client only the data they are authorized to access. Exposing excessive or unnecessary information can lead to serious security risks and data leakage.

10.3 HTML tampering and Injection

HTML injection is a type of injection vulnerability that occurs when an attacker can control an input point and inject arbitrary HTML (or script) into a page. The targeted browser cannot reliably distinguish between legitimate and malicious parts of the page, which can lead to content manipulation, UI spoofing, or cross-site scripting (XSS).

Problem: Untrusted input is directly embedded in HTML output (for example via `document.write` or unsanitized template insertion). If the input contains markup or script, the browser will render and execute it in the context of the vulnerable page.

Unsafe example:

```
var userposition = location.href.indexOf("user=");
var user = location.href.substring(userposition + 5);
document.write("<h1>Hello, " + user + "</h1>");
```

To exploit the vulnerabilities, specific url must be built, using for example:

```
http://vulnerable.site/page.html?user<img%20src='aaa'%20onerror=alert(1)>
```

Use:

- **Defacing** — modify visible page content to mislead or damage reputation.
- **Exfiltrating anti-CSRF tokens** — force the browser to render attacker-supplied markup that exposes hidden tokens.
- **Exfiltrating stored credentials** — inject forms that may be auto-filled by password managers and thus stolen.

11 Overflow

11.1 Introduction

The memory of a process is divided into several distinct areas, each serving a specific purpose for data storage and program execution. It can be generally divided as follows:

- **Reserved memory area**: contains the machine instructions and data reserved for the operating system;
- **Text area**: stores the assembly instructions of the program currently being executed;
- **Static data area**: contains global and static variables declared in the program;

- **Heap**: a dynamic memory area, managed by the developer. It contains dynamically allocated data and variables created during runtime;
- **Stack**: keeps track of active functions, their parameters, return addresses, and local variables.

11.2 Buffer Overflow

A **buffer overflow** occurs when a program attempts to write more data into a fixed-length memory buffer than it can hold.
Since user data and control flow information are mixed together in the memory, the user data exceeding a buffer may corrupt other data and control flow information.

Spotting buffer overflow: The following conditions frequently lead to buffer overflow vulnerabilities:

- input originates from untrusted sources (network, files, or command line);
- the input is copied or transferred into internal structures such as fixed-size buffers;
- data or strings are manipulated using **unsafe functions** that do not perform bounds checking (for example: `strcpy`, `strcat`, `gets`).

Vulnerabilities of this kind can be detected using dynamic testing techniques such as fuzzing.

Fixing buffer overflow: When working with strings, always use safe versions of the needed functions, that take into account the count of characters to work on. Always check loop termination and array boundaries.

11.3 String format vulnerabilities

A **non-verified string** used as a format string may allow an attacker to inject format specifiers (for example `%s`, `%x`) that read values from the stack. This becomes possible when a formatting function accepts a variable number of arguments (ellipsis), e.g.

```
int printf(const char *format, ...);
```

Passing untrusted input directly as the `format` parameter is unsafe because the attacker can control how the function interprets stack contents.

Fixing string formats: always sanitize user input before using it as a format string. Use constant strings whenever possible and avoid the use of unsafe functions, such as `printf();`

11.4 Integer overflow

2's complement notation: Integers are usually represented using the **two's complement** notation. Given a sequence of n bits, the representable range of values is:

$$[-2^{n-1}, 2^{n-1} - 1]$$

This means that, when working with signed integers, exceeding the upper or lower limit of this range causes the value to **wrap around** to the opposite side of the range.

Example: For an 8-bit signed integer:

$$[-128, 127]$$

Adding 1 to 127 results in -128:

$$127 + 1 = -128$$

Similarly, subtracting 1 from -128 results in 127.

Causes and risks: Integer overflow occurs when an arithmetic operation produces a value outside the representable range of the integer type. In C and C-like languages, this can happen due to:

- arithmetic operations (`+`, `-`, `*`);
- insufficient range checking on user-controlled input (e.g., array indices, buffer sizes, loop counters).

Fix integer overflow:

- Use sufficiently large integer types (**short** = 16 bits, **int** = 32 bits, **long** = 64 bits);
- Explicitly check arithmetic operations that may exceed type limits, by comparing with data type limit;

11.5 Heap overflow

As described in Chapter 11.1, the heap is a dynamic memory area managed at runtime. Incorrect use can cause **heap overflows**, which may crash the application or corrupt heap metadata. This can happen when:

- allocating excessively large buffers without validating sizes;
- repeatedly allocating memory and failing to free it (memory leaks) combined with unchecked writes;
- writing beyond the bounds of a dynamically allocated buffer (off-by-one, missing length checks).

11.6 Final remarks and barriers

Base attack pattern: usually **overflow-based** attacks follow these steps:

1. **Inject attack code** into the buffer. Code that is already in the program can be used or otherwise, code can be injected;
2. **Redirect control flow** to the injected code. This way the execution jumps to the malicious code;
3. **Execute** the attack code.

Barriers to exploitation: there are 4 main approaches to barriers the exploitation of this vulnerability:

- **Address space** layout randomization: use a form of randomization to store data in the memory. This way, we cannot know the location of certain functions;
- **Canaries**: fixed-byte known values, placed between a buffer and control data on the stack. This way, when the buffer overflows, the first data to be corrupted will be this fixed data, alerting of the overflow;
- **Executable space protection (ESP)**: marks regions as non-executable. Any attempt to execute machine code in these regions will cause an exception;
- **testing**: fuzzing and static analysis can be used to detect this type of vulnerability.

12 Static Analysis and Testing

Software analysis and testing are applied to detect security issues. These are usually driven by the programmer's experience to recognize patterns and situations in the code. Its automation is a key aspect if we want to build **secure software projects**.

Verification: check the consistency of the implementation with a specification. This phase usually checks for the presence of bugs in the application. The target of the verification can be different (unit testing, set of units, system testing) and for each target, different types of analysis must be done;

Validation: check the degree at which a software system fulfills the customer requirements.

12.1 Dynamic analysis (or Testing)

Software testing concerns **exercising the software** and observing the behavior and the produced output.

- Test data are needed to execute the software;
- Can reveal the presence of errors, not their absence

To conduct Dynamic analysis, firstly test cases must be identified to test the software under stress.

- They are based on specifically defined input data;
- An **oracle** that describes what the system is supposed to do is needed. It will be used to check program results.

12.2 Static analysis

Static analysis examines the software's source code or documentation **without executing the program**.

- Both code and design documents can be analyzed;
- It can demonstrate the absence of specific classes of errors.

In this context, the **test oracle** is the tool's input — a set of desired properties or correctness rules that the program must satisfy.

12.2.1 Approaches to static analysis

Several **approaches to static analysis** exist, each with different goals and levels of automation:

- **Model checking**: a formal verification technique that systematically explores the state space of a program to verify if it satisfies certain properties;
- **Pattern-based analysis**: enforces coding practices or policies by matching code against known patterns, templates, or weaknesses. It is easy to apply but often limited in precision;
- **Flow-based static analysis**: simulates the logical execution of the program to track data propagation and its effect on control flow. It does not depend on user input and can identify data-dependent bugs.

12.2.2 Approximations in static analysis

Static analysis often relies on **approximations** of the program's reachable states and behaviors to make the analysis tractable.

- **Sound approximation**: every property that holds in the approximation also holds in the actual program. It may produce false positives but guarantees no false negatives;
- **Complete approximation**: every property that holds in the approximation also holds in the actual program. Also referred as under-approximation, it may produce false negatives but guarantees no false positives;
- **Conservative approximation**: the approximation is sound and complete.

12.2.3 Pros and cons of static analysis

- **Pros**:
  - Scales really good for large codebases;
  - Can establish the absence of certain types of vulnerabilities;
  - Can produce an explanation of why properties hold or not.
- **Cons**:
  - It's not always easy to scale the problem of approximating program behaviors;
  - Non trivial semantic properties are undecidable (Rice's theorem);

13 Flow analysis

Several models can be used to describe an application and represent its code. Different models may emphasize different aspects of program behavior — such as **control flow** or **data dependence**.
Flow analysis focuses on understanding how information moves through the program. By analyzing both control and data dependencies, it helps identify relationships among variables, track how data values are produced and consumed, and evaluate the effects of data changes across the codebase.

Program analysis and test design techniques often leverage data flow information to enhance their effectiveness.

13.1 Control-flow graph

A **Control-Flow Graph (CFG)** is defined as follows:

$$CFG = (N, E, n_e, n_x)$$

- N: **set of nodes**, one for each **statement** (instruction) of the program;
- $E \subseteq N \times N$: **set of edges**, where $(n, m) \in E$ if statement m can be executed immediately after n;

- n_e: **entry node** of the program;
- n_x: **exit node** of the program.

Path: represents a possible program execution. It is composed of a sequence of nodes and edges starting from the entry node and ending at a terminal node.

Linearly independent path: a path is **linearly independent** if it introduces at least one new edge that has not been traversed by any previously defined path. These paths are used to identify distinct behaviors implemented in a program. Typically, a test case should be defined to verify each independent path.

13.1.1 Def-Use pairs

A **def-use pair** associates a point in a program code where a value is defined with a point where it's used.

Definition: where a variable gets a value:

- Declaration;
- Initialization;
- Assignment;
- Values received by a parameter.

Use: extraction of a value from a variable:

- Expressions;
- Conditional statements;
- Parameter passing;
- returns.

Def-clear path: a path along the CFG from a definition to a use of the same variable, without another definition of the variable in between.

13.1.2 Data Dependence Graph

A direct data dependence graph is defined using the two definitions we just gave:

- Its nodes are the same as in the CFG;
- Its edges are the def-use pairs.

This graph can be used to identify **data dependence**, where values are from.

13.1.3 Control Dependence Graph

Also a **control dependence graph** can be defined as follows:

- Its nodes are the same as the same as in the CFG;
- Its edges are unlabeled, direct control dependencies.

This type of graph shows **control dependence**: which **statement controls** whether a statement is executed or not.

13.2 Data-Flow analysis

13.2.1 Reaching definition

There is an association (d, u) between a **definition of a variable** `v` at code statement d and a **use of variable** `v` at code statement u iff:

- there is at least one control flow path from d to u;
- there is no intervening definition of `v`.

In this case, we say that v_d **reaches** v_u.

Normal graph exploration Even if we consider a loop-free path, the number of paths can be exponentially larger than the number of nodes and edges. Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

DF Algorithm: an efficient algorithm for **computing reaching definitions**, and other properties, is based on the way **reaching definitions** at one node are related to the one of its adjacent nodes.

Suppose we are calculating the **reaching definitions** of node n, and there is an edge (p, n) from an immediate predecessor, node p:

- if the predecessor node p, assigns a value to a variable `v`, then the definition v_p **reaches** n;
- If a definition v_p of a variable `v` reaches a predecessor node p, then the definition is propagated on from p to n.

Formal definition: we can formally define the idea expressed by the algorithm we just cited. At each node n, we have:

- **Reaching definitions** ($In(n)$): definitions flowing out of th predecessor nodes m. Can be defined as

$$\bigcup_{m \in pred(n)} Out(m)$$

- **Out flow** $Out(n)$: the output of a node n can formally be defined as:

$$Out(n) = In(n) \setminus Kill(n) \cup Gen(n)$$

  Where:

  - $Gen(n) = \{v_n | v \text{ is defined or modified at } n\}$
  - $Kill(n) = \{v_x | v \text{ is defined or modified at x AND } v \text{ is modified at} n\}$

13.2.2 Meet over path

Another solution for data-flow analysis is the **Meet Over Path** (MOP).
At a node n, the MOP is defined as:

$$MOP[n] = \bigwedge_{p \in P_n} f_p(T)$$

where:

- P_n the set of all path from the entry node to node n;
- f_p is the composition of transfer function along path p;
- T is the initial information
- $\wedge$ represents the meet operator.

Exact solution: the exact solution requires, instead of considering all path from the entry node, to the node n, to just consider the **feasible paths**.

$$EX[n] = \bigwedge_{p \in feas(P_n)} f_p(T)$$

This is an **undecidable** problem.

14 Taint Analysis

Taint analysis is one possible solution of the data-flow analysis problem. It aims to track the flow of sensitive information through a program and identify potential security vulnerabilities.
To understand how taint analysis works, we need to define some concepts:

Tainted Variable: a variable is considered **tainted** if it contains data from an untrusted source (for example, user input), and has not yet been sanitized or validated. When it gets checked, it becomes untainted;

Untainted Variable: a variable is considered **untainted** if it contains data that has been properly sanitized or validated;

Variables containing unsanitized user input (tainted variables) should never be used in security-critical statements (such as system calls, database queries, file operations).
Taint analysis aims at keeping track of tainted variables, along the execution path of of the program.

- **Static taint analysis:** analyzes the program's source code without executing it, to identify potential taint flows and vulnerabilities. It's conducted on the Control Flow Graph (CFG) of the program;
- **Dynamic taint analysis:** tracks the flow of tainted variables during program execution, allowing for more precise detection of vulnerabilities in real-time.

14.1 Static taint analysis

Source: a source is the point of the program where a tainted variable is defined. This could be user input, data read from a file, or any other untrusted data.

Sink: a sink is the point of the program where a tainted variable is used in a potentially dangerous way, such as being passed to a system call or used in a database query. The possible sinks can differ based on the application context and on the vulnerability we want to detect;

Taint propagation rules: specify the taint status for data derived from tainted/untainted operand;

Sanitization: the operation conducted on a tainted variable to check / sanitize it.

The goal of static taint analysis is, for all possible inputs, to prove that the tainted data will never be used where untainted data is expected. The solution requires to analize all the possible data flow of the program. This is done following these steps:

1. Identify possible **sources** in the program;
2. Identify possible **sinks** in the program;
3. Identify and analyze all possible data flows in which sources can reach sinks;
4. Identify wheter a tainted sources flows into a security-critical sink.

14.1.1 How to implement taint analysis

Static Taint Analysis can be implemented by customizing the data flow analysis framework.

Taint status: the taint status of a variable can be either **true** (tainted) or **false** (untainted). It is expressed as: $x \to \{T, F\}$;

Flow information: the flow information propagated for taint analysis consists of taint sets. Formally: $V = \mathcal{P}(X)$; where X is the set of all variables in the program, and $\mathcal{P}(X)$ is the power set of X;

Meet operator: at a join point, the taint status of a variable is tainted if it is tainted along at least one incoming path. The meet operator is therefore the union of the taint sets of its predecessors.

Transfer function: the transfer function for taint analysis has the form:

$$f_n(x) = Gen[n] \cup (x \setminus Kill[n])$$

where:

- x: a var $\in In(n)$;
- $Gen[n]$: the set of variables that are tainted at node n. It's formally defined as:

$$Gen[n] = \{x \in X | x \text{ is assigned an input value at statement } n\}$$

$$\cup$$

$$\{x \in X | \exists y \in X : x \text{ is assigned a value obtained from } y \wedge y \to T\}$$

- $Kill[n]$: the set of variables that are untainted at node n. It's formally defined as:

$$Kill[n] = \{x \in X | x \text{ is sanitized at statement } n\}$$

$$\cup$$

$$\{x \in X | \forall y \in X : x \text{ is assigned a value obtained from } y \wedge y \to F\}$$

14.1.2 Algorithm

The algorithm for static taint analysis can be summarized as follows:

14.1.3 Limitations

Static taint analysis has some limitations:

- **False positives:** the analysis may report vulnerabilities that do not actually exist in the program, due to overtainting;
- **Hard implementation:** implementing a precise and efficient static taint analysis can be complex, especially for large codebases with intricate control flows;
- we do not know what actual value might cause the vulnerability to be exploited.

Algorithm 1 Static Taint Analysis Algorithm

```
1: for all node n do
2:     init Gen[n]
3:     init Kill[n]
4: end for
5: repeat
6:     for all node n do
7:         In[n] = ∪_{p∈pred(n)} Out(p)
8:         Out[n] = Gen[n] ∪ (In[n] \ Kill[n])
9:     end for
10: until no changes in any In(n) or Out(n)
```

14.2 Dynamic taint analysis

Dynamic taint analysis tries to overcome some of the limitations of static taint analysis by tracking tainted variables during program execution.

Traces: flow of data through the program;

Taint sources: points where tainted data is introduced into the program;

Taint propagation: how taint status is transferred between variables during execution;

Sinks: points where tainted data is used in a potentially dangerous way.

Code instrumentation is usually used to implement dynamic taint analysis. It consists in modifying the program code to insert additional instructions that track the taint status of variables during execution. This type of instrumentation can be done at different levels:

- **Source code level:** modify the source code of the application;
- **Binary level:** modify the compiled binary code of the application;
- **Runtime level:** use a runtime environment that supports taint tracking.

The higher the level, the more information is available for taint tracking.

15 Testing

Dynamic analysis, also referred to as **testing**, involves examining a program's behavior during its execution. We can differientiate between two main types of testing:

- **Testing**: verifies the functionality of the software against sensible inputs and border cases. It checks that functionalities work as expected;
- **Security testing**: looks for wrong, unwanted and strange behaviors of the software that could lead to vulnerabilities and security issues.

Test suite: a collection of test cases that are intended to be executed together to validate the behavior of a software application. Each test case is composed of a **test input** and an **expected output**, which is used to verify the correctness of the software's behavior.

Problem of testing:

- How to **identify** the **test cases**?
- How to identify the **test oracle**?
- When to **stop testing**?

15.1 Test coverage criteria

A **test coverage criterion** defines a set of requirements that a test suite must satisfy to be considered adequate. It aims at insuring that a test suite is comprehensive enough and that all relevant and critical application aspects and functionality are covered.
Test coverage criteria can:

- **Increase test effectiveness**: helps insuring that the test suite covers a wide range of scenarios and edge cases;
- **Remove redundant tests**: can identify and eliminate redundant tests that do not contribute to the overall coverage;
- **Discover uncovered areas**: can highlight areas of the code that are not covered by existing test suites;
- **Improve regression testing**: can help ensure that changes to the code do not introduce new bugs or vulnerabilities.

Several test coverage criteria exist, each focusing on different aspects of the software. We will mainly focus on **code coverage criteria**. It measures how much of the application source code is executed during testing. Several code criteria exist:

- **Statement coverage**: ensures that each statement in the code is executed
- **Branch coverage**: ensures that each branch of control structures (like if-else statements) is executed
- **Decision coverage**: ensures that each decision point (like conditions in if statements) is evaluated to both true and false
- **Condition coverage**: ensures that each individual condition in a decision point is evaluated to both true and false

We first havw to introduce a few definitions:

- **Coverage**: the percentage of code elements (statements, branches, etc.) that have been executed during a single test case;
- **Test suite coverage**: measures how effectively a set of test cases covers the codebase. It can be computed in different ways:
  - **Total suite coverage**(TSC): obtained by aggregating the coverage achieved by all test cases in the suite;
  - **Additional suite coverage**(ASC): consider the incremental coverage provided by each test case. Starting from the case with covers the largest number of new elements, add subsets based on the number of new elements they cover.

15.1.1 Statement coverage criteria

Statement coverage aims at involving the execution of **all possible statements** in the code at least once during testing. It can be defined as:

$$\text{coverage} = \frac{\text{executed statements}}{\text{total statements}} \cdot 100$$

The goal is to identify test cases that covers different portion of the code.

Considering the whole suite, we can define:

$$\text{TSC} = \frac{\text{covered statement}}{\text{total statements}} \cdot 100$$

15.1.2 Decision coverage criteria

Decision coverage aims at ensuring that **the possible outcomes of all boolean expressions** are executed at least once during testing. It can be defined as:

$$\text{coverage} = \frac{\text{executed decision outcomes}}{\text{total decision outcomes}} \cdot 100$$

To fully cover all decision outcomes, test cases must be designed to evaluate each boolean expression to both true and false.

Considering the whole suite, we can define:

$$\text{TSC} = \frac{\text{covered decision}}{\text{total decision}} \cdot 100$$

15.1.3 Condition coverage criteria

Similar to decision coverage, condition coverage focuses on ensuring that **each individual condition** within a decision point is evaluated to both true and false at least once during testing. It can be defined as:

$$\text{coverage} = \frac{\text{executed condition outcomes}}{\text{total condition outcomes}} \cdot 100$$

Considering the whole suite, we can define:

$$\text{TSC} = \frac{\text{covered condition}}{\text{total condition}} \cdot 100$$

15.1.4 Branch coverage criteria

Branch coverage aims at ensuring that **all possible branches** of control structures are executed. To ensure this, it checks that every possible edge in the control flow graph is traversed during testing. It can be defined as:

$$\text{coverage} = \frac{\text{executed branches}}{\text{total branches}} \cdot 100$$

Considering the whole suite, we can define:

$$\text{TSC} = \frac{\text{covered branches}}{\text{total branches}} \cdot 100$$

15.2 Mutation testing

The goal of **mutation testing** is to evaluate the effectiveness of a test suite in discovering real faults in the software. It involves creating **mutants** of the original program by introducing small changes (mutations) to its code.

The process of mutation testing can be summarized in the following steps:

1. **Mutant generation:** create a set of mutants by applying mutation operators to the original program. Each mutant should differ from the original program by a small single change;
2. **Test execution:** run the existing test suite against each mutant. Record whether the test suite detects the mutation;
3. **Mutant analysis:** analyze the results of the test executions to determine which mutants were "killed" (detected by the test suite) and which survived (not detected);

The effectiveness of the test suite can be measured using the **mutation score**:

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}} \cdot 100$$

We say that a test suite is **mutation adequate** if it achieves 100% mutation score.

15.3 Fuzzing

Fuzzing is a **highly effective** and **automated** security testing technique. It involves generating **random or semi-random inputs** to a program and check for crashes, memory leaks, or unexpected behavior. Some examples of test inputs are:

- Very **long** or **empty** strings;
- **min / max** values for numeric inputs, zero or negative numbers;
- **Malformed** data structures or files;
- **Unexpected** sequences of inputs or commands, usually specific to the application domain.

15.3.1 Types of Fuzzing

- **Dump fuzzing**: fuzzing that does not consider the context and state of the software. It focuses on generating random inputs without any knowledge, fed directly to the program and observing its behavior.
- **Smart fuzzing**: generates inputs based on a predefined model, using a degree of understanding of the target application. This approach aims to create **valid** inputs that conform to the expected structure of the application, taking into account also the code coverage. Examples of algorithm used for generating inputs are:
  - **Random**;
  - **Template / Grammar**: follows specific templates or grammars to generate inputs that conform to expected formats;
  - **Guided**: the input generation is guided by feedback from previous test executions, such as code coverage information or execution paths taken by the program;
  - **Mutation based**: starts with a set of valid inputs and applies mutations to create new test cases;
  - **Generation based**: creates inputs from scratch based on a model of the input space.

15.3.2 Pros and cons of Fuzzing

- **Pros**:
  - Can discover unknown vulnerabilities and give insights into the robustness of the software;
  - Automated and scalable. With minimal human effort it can generate a large number of test cases;

- **Cons**:
  - Fuzzers don't guarantee full coverage of the codebase;
  - Crashes may be hard to reproduce and debug;
  - For programs with complex input formats, generating valid inputs and achieving good coverage can be challenging.

15.4 Black-box fuzzing

Black-box fuzzing is a software testing technique, based on the concept of **fuzzing**, that focuses on testing the external behavior of a program without any knowledge of its internal structure or implementation. To derive test cases, black-box fuzzing relies only on the program's input and output specifications. There are several types of black-box fuzzing techniques, including:

- **Random testing**: input values are randomly sampled from the appropriate input space;
- **Grammar based fuzzing**: input values are generated based on a predefined grammar that describes the valid input format;
- **Mutation based fuzzing**: existing valid inputs are modified or mutated to create new test cases;
- **Equivalence partitioning**: inputs are divided into equivalence classes, and test cases are generated from each class to ensure coverage of all possible scenarios.

15.4.1 Random testing

Random testing is the simplest form of black-box fuzzing, where input values are randomly generated from the input space of the program. This technique is easy to implement and can quickly generate a large number of test cases, but it may not be effective in finding specific types of bugs or vulnerabilities.

15.4.2 Grammar based fuzzing

In grammar based fuzzing, test cases are generated from a predefined grammar that describes the valid input format for the program. This technique is particularly useful for testing programs that accept structured inputs, only creating valid inputs.
The steps involved in grammar based fuzzing usually include:

1. **Definition of the grammar**: a formal grammar is defined that describes the valid input format for the program;
2. **Generation of test cases**: such grammar is provided to the fuzzer, which generates valid input values;

3. **Execution of test cases**: the generated test cases are executed on the program, and the output is monitored for any unexpected behavior or crashes.

15.4.3 Mutation based fuzzing

In mutation based fuzzing little or no knowledge of the input format is required. Instead, existing valid inputs are modified or mutated to create new test cases. An example of mutation-based fuzzing, in the context of a PDF viewer application, could involve:

1. **Collection of seed inputs**: a set of valid PDF files is collected to serve as seed inputs for the fuzzer;
2. **Test of collected inputs**: the seed inputs are executed on the PDF viewer application to ensure they are valid and do not cause any crashes or unexpected behavior;
3. **Mutation of inputs**: using a mutation algorithm, grab the seed inputs and apply various mutations to them. Feed the mutated inputs back into the application;

15.4.4 Equivalence partitioning

In equivalence partitioning, the input space of the program is divided into equivalence classes, called **partitions**. We can assume that:

- all inputs within a partition will be treated similarly by the program;
- testing one input from each partition is sufficient to cover all possible scenarios.

At least **valid** and **invalid** partitions should be created for each input parameter.

15.5 White-box fuzzing

White-box fuzzing is a software testing technique that is based on the fact that the tester has **full knowledge** of the **internal structure** and **implementation** of the program being tested.
This technique combines traditional fuzzing methods with program analysis techniques, making it possible to generate significant test cases.

15.5.1 Symbolic execution

Symbolic execution is a **static code analysis** technique used in white-box fuzzing to **explore all possible execution paths** of a program. Instead of using concrete input values, symbolic execution uses **symbolic values** to represent inputs, allowing it to reason about the program's behavior more generally. It is composed of the following steps:

1. **Symbolic representation**: assign symbolic values to the program's input variables;
2. **Path exploration**: execute the program symbolically, exploring all possible execution paths;
3. **Path constraints**: collect path constraints for each execution path, representing the conditions that must be satisfied for that path to be taken;
4. **Checking satisfiability**: use a constraint solver to determine if the path constraints are satisfiable.

The output of symbolic execution is a **set of path constraints** that can be used to generate test cases that cover all possible execution paths of the program.

15.5.2 Dynamic symbolic execution

Dynamic symbolic execution (DSE), is a hybrid technique that combines symbolic execution with concrete execution. It executes the program with concrete inputs while simultaneously tracking symbolic constraints along the execution path. Solving these constraints generates new inputs that explore different paths in subsequent executions.
The steps involved in dynamic symbolic execution are:

1. **Concrete execution**: execute the program with a set of well-defined initial inputs;
2. **Symbolic tracking**: collect constraints on symbolic inputs during execution;
3. **Solve constraints**: use a constraint solver to generate new inputs that explore different execution paths;
4. **Fuzzing**: generate new test cases by negating constraints one by one;
5. **Repeat**: repeat the process with the newly generated inputs to explore more paths.