

# Machine Learning

Marchetto Alessandro

a.y. 2025/2026

Davide Donà

December 2025

Notes Repository

*These notes are for the course “Security Testing” held by professor Alessandro Marchetto.*

*They are not official material; rather, they have been created based on the lecture slides and additional reference resources, with the aim of supporting students during their study of the subject.*

*The goal of these notes is to provide a clear and structured overview of the topics covered in the course, while remaining as complete and accurate as possible.*

*We tried our best to avoid mistakes while compiling these notes. That being said, due to the breadth of the topics discussed, inaccuracies or imperfections may still be present. We do not take responsibility for any such errors; however, any feedback, corrections, or suggestions are highly appreciated.*

*In the GitHub repository linked on the title page, you can find the  $\text{\LaTeX}$  source code of this document. You are free to clone the project, modify it, or contribute by opening an issue or submitting a pull request.*

*We hope that this document will be helpful in the study of this interesting and important subject. Happy studying!*

*Davide Donà*

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Definitions . . . . .	1
1.2	Bug Exploitation . . . . .	2
1.3	Security Meta-Model . . . . .	2
1.4	Software Development Life Cycle . . . . .	3
1.4.1	From a security perspective . . . . .	3
1.4.2	DevOps . . . . .	5
<b>2</b>	<b>Code Weaknesses and Vulnerabilities</b>	<b>5</b>
2.1	Common Weakness Enumeration (CWE) . . . . .	5
2.2	Most common security risks . . . . .	10
<b>3</b>	<b>SQL Injection</b>	<b>13</b>
3.1	Underlying idea . . . . .	13
3.2	Example attack . . . . .	13
3.3	SQL Injection type . . . . .	13
3.3.1	In-Band SQL injection: . . . . .	14
3.3.2	Inferential SQL Injection: . . . . .	14
3.4	SQL Injection mitigation . . . . .	15
<b>4</b>	<b>XSS (Cross-Site Scripting)</b>	<b>15</b>
4.1	Underlying idea . . . . .	15
4.2	Example . . . . .	15
4.3	XSS Types . . . . .	16
4.3.1	short . . . . .	16
4.3.2	short . . . . .	17
4.3.3	short . . . . .	17
4.4	XSS mitigation . . . . .	18
<b>5</b>	<b>Request Forgery</b>	<b>18</b>
5.1	HTTP and Cookies introduction . . . . .	19
5.2	Session Hijacking . . . . .	19
5.3	Cross-Site Request Forgery . . . . .	21
5.4	Server-Side Request Forgery . . . . .	21
<b>6</b>	<b>Command Injection</b>	<b>23</b>
<b>7</b>	<b>File access</b>	<b>24</b>
7.1	File Inclusion . . . . .	24
7.2	File Access - Directory Traversal . . . . .	24
7.3	Filename-based Binding . . . . .	24
7.4	TOCTOU (Time Of Check, Time Of Use) . . . . .	25
7.5	Mitigation . . . . .	25

---

<b>8 Error Handling</b>	<b>25</b>
<b>9 Insecure Direct Object Reference (IDOR)</b>	<b>26</b>
<b>10 Client-side vulnerabilities</b>	<b>27</b>
10.1 Validation Bypass . . . . .	27
10.2 Data Filtering . . . . .	27
10.3 HTML tampering and Injection . . . . .	28
<b>11 Overflow</b>	<b>28</b>
11.1 Introduction . . . . .	28
11.2 Buffer Overflow . . . . .	29
11.3 String format vulnerabilities . . . . .	29
11.4 Integer overflow . . . . .	30
11.5 Heap overflow . . . . .	31
11.6 Final remarks and barriers . . . . .	31
<b>12 Supply Chain Attack</b>	<b>32</b>
12.1 Dependencies in Software Development . . . . .	32
12.2 What is a Supply Chain? . . . . .	32
12.3 Supply Chain Attacks . . . . .	32
<b>13 Security testing</b>	<b>34</b>
13.1 Verification and Validation . . . . .	34
13.2 Software Verification techniques . . . . .	34
13.2.1 Dynamic analysis (or Testing) . . . . .	34
13.2.2 Static analysis . . . . .	35
<b>14 Static Analysis</b>	<b>35</b>
14.1 Model Checking . . . . .	36
14.1.1 Linear Temporal Logic (LTL) . . . . .	36
14.1.2 Model checking process . . . . .	36
14.2 Pattern-based analysis . . . . .	37
14.3 Flow-based static analysis . . . . .	37
14.4 Model for Flow Analysis . . . . .	38
14.4.1 Control-Flow Graph . . . . .	38
14.4.2 Def-Use pairs . . . . .	39
14.4.3 Data Dependence Graph . . . . .	39
14.4.4 Control Dependence Graph . . . . .	40
14.5 Data-Flow analysis . . . . .	41
14.5.1 Reaching definition . . . . .	41
14.5.2 DF Algorithm: . . . . .	41
14.5.3 Meet over path . . . . .	42
14.5.4 Approximations in static analysis . . . . .	43
14.5.5 Pros and cons of static analysis . . . . .	43

---

<b>15 Taint Analysis</b>	<b>43</b>
15.1 Static taint analysis . . . . .	44
15.1.1 How to implement taint analysis . . . . .	44
15.1.2 short . . . . .	46
15.1.3 short . . . . .	46
15.2 Dynamic taint analysis . . . . .	47
<b>16 Dynamic Analysis</b>	<b>47</b>
16.1 Testing . . . . .	48
16.1.1 Test coverage criteria . . . . .	48
16.1.2 Mutation testing . . . . .	51
16.2 Fuzzing . . . . .	51
16.2.1 Types of Fuzzing . . . . .	52
16.2.2 Pros and cons of Fuzzing . . . . .	52
16.3 Black-box fuzzing . . . . .	53
16.3.1 Random testing . . . . .	53
16.3.2 Grammar based fuzzing . . . . .	53
16.3.3 Mutation based fuzzing . . . . .	54
16.3.4 Equivalence partitioning . . . . .	54
16.4 White-box fuzzing . . . . .	54
16.4.1 Symbolic execution . . . . .	55
16.4.2 Dynamic symbolic execution . . . . .	55
<b>17 AI for Vulnerability Detection</b>	<b>56</b>
17.1 Techniques . . . . .	56
17.1.1 Sequence-based models . . . . .	56
17.1.2 Graph-based models . . . . .	56
17.2 Explainable AI . . . . .	57
<b>18 Exercises</b>	<b>58</b>
18.1 Static Taint Analysis . . . . .	58
18.2 Test Coverage . . . . .	59
18.3 Mutation Testing . . . . .	64

## 1 Introduction

Before starting to analyze the different types of software vulnerabilities and the related testing techniques, it's important to define some basic concepts and terminology that will be used throughout the document.

### 1.1 Definitions

**Software** Software can be defined as a set of programs and data, that provides functionality. Software and functionality always come with certain security risks. Every new feature added to a software can introduce new vulnerabilities.

**Security** Managing the risks introduced by software. This includes understanding and identifying security risks, 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. If a software does not achieve the security requirements, it can lead to security failures.

**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 by an attacker, this can lead to a security failure.

**Security Risk** A security risk refers to the probability that a threat (any external actor) will exploit a vulnerability, combined with the resulting impact (damage). The risk can be calculated as:

$$\text{Risk} = \text{Likelihood} \times \text{Impact}$$

**Collections** We can find some important databases and collections related to software security:

- **OWASP:** Open Web Application Security Project, is a non-profit organization focused on improving the security of software. It provides a series of resources, such as the *OWASP Top 10*, a list of the most critical web application security risks.
- **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.
- **NVD:** National Vulnerability Database, public available database of CVEs. It provides additional information, such as severity scores and impact ratings.

## 1.2 Bug Exploitation

A bug is an error in the software code that can lead to unexpected behavior. An attacker will try to exploit a bug to make the software behave in an unintended way, to achieve its malicious goals. To achieve its goals, the attacker may need several exploits, chained together, composing an attack path.

## 1.3 Security Meta-Model

In the context of software security, a meta-model can be defined to represent the relationships between different security concepts, such as vulnerabilities, threats, and attacks. This meta-model can help security professionals understand the attack surface of a system and identify potential weaknesses.

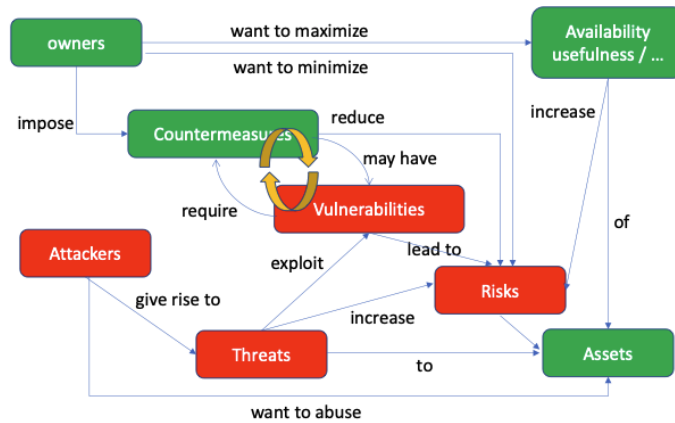


Figure 1: Security Meta-Model

## 1.4 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.

### 1.4.1 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;
- **Vulnerability Scanning:** injection of malicious inputs in the program, to identify vulnerabilities.

**Testing:** Testing is suggested to be done throughout the whole development process, to identify bugs as soon as possible. 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.

### 1.4.2 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, based on 8 phases. Developers regularly merge their code changes in a central repository (CI *Continuous Integration*), where it gets automatically built, tested, and prepared for a release to production (CD *Continuous Delivery*).

## 2 Code Weaknesses and Vulnerabilities

We now present a selection of the most dangerous and common 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:

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

### 2.1 Common Weakness Enumeration (CWE)

#### Use-After-Free (CWE-416)

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      //Accessing a pointer
8      logError("operation aborted before commit", ptr);
9  }
```

Use-after-free can be found also when other resources, such as files or network connections are mismanaged.

- **Cause:** access to a pointer after it has been freed;
- **Prevention:** once freed, all pointers should be set to NULL;
- **Detection:** using static analysis or fuzzing.

### Heap-based Buffer Overflow (CWE-122)

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.

- **Cause:** writing more data than the allocated buffer size;
- **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 or fuzzing.

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

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;
```

- **Cause:** writing data outside the boundaries of a buffer;
- **Prevention:** Always verify that the buffer is as large as specified and that its boundaries are respected.
- **Detection:** Using static analysis and dynamic analysis.

### Improper Input Validation (CWE-20)

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.

```
1 public static final double price = 20.00;
2 int quantity = currentUser.getAttribute("quantity");
3 double total = price * quantity;
4 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.

- **Cause:** missing or incorrect validation of input data;
- **Prevention:** Always validate input data, both on client-side and server-side;
- **Detection:** Using static analysis or dynamic analysis.

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

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

```
1 $userName = $_POST["user"];
2 $command = 'ls -l /home/' . $userName;
3 system($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.

- **Cause:** inclusion of untrusted input in an OS command, without neutralizing special elements;
- **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.

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

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

```
1  try {  
2      File file = new File("object.obj");  
3      ObjectInputStream in = new ObjectInputStream(new  
          FileInputStream(file));  
4      javax.swing.JButton button = (javax.swing.JButton)  
          in.readObject();  
5      in.close();  
6  }
```

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

- **Cause:** deserialization of untrusted data, without verifying its validity;
- **Prevention:** When deserializing data, a new object should be populated, rather than just deserializing it.
- **Detection:** Using static analysis.

### Server-Side Request Forgery (CWE-918)

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.

- **Cause:** Misconfigured server settings or lack of proper input validation;
- **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:** Using dynamic analysis or vulnerability scanning.

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

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.

- **Cause:** accessing a resource using a type incompatible with the original type;
- **Prevention:** Attention to implicit and explicit type conversion;
- **Detection:** Using static analysis.

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

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  }
```

- **Cause:** inclusion of untrusted input in a file or directory path, without neutralizing special elements;
- **Prevention:** Always assume inputs are malicious. Validate and sanitize user input, to avoid directory traversal attacks. When the set of acceptable objects, 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.

### Missing Authentication for Critical Function (CWE-306)

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

- **Cause:** missing authentication for a critical function;
- **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.

## 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. In the following, we present the OWASP Top-10 of 2021.

### Broken Access Control (A01-2021)

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

- **Example:** An application allows users to access other users' accounts by simply changing a parameter in the URL.

```
1 //Trying to access using this url
2 //https://example.com/app/accountInfo?acct=myacct
3 PreparedStatement pstmt = connect.prepareStatement(query
4 );
5 pstmt.setString(1, request.getParameter("acct"));
6 //myacct is a "reference" to an account
7 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:** Implement proper access controls standards, such as the least-privilege principle and deny by default.

### Cryptography Failures (A02-2021)

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:** Always use proper encryption mechanisms, with a correct setting.

### Injection (A03-2021)

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:** An application constructs SQL queries using user input directly.

```
1 String query = "SELECT * FROM accounts WHERE custID='" +  
2   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 parameterized queries (prepared statements) or stored procedures, to separate code from data. Always validate and sanitize user input.

### Insecure design (A04-2021)

Insecure design concerns weaknesses related to missing or ineffective control design. Note that this is different from implementation flaws, as it relates to design and architectural flaws.

- **Example:** An application implements a password recovery workflow that relies solely on easily guessable personal information to verify user identity.
- **Prevention:** Adopt secure design methodologies, such as threat modeling and secure design patterns.

### Security Misconfiguration (A05-2021)

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.

To prevent security misconfiguration, a secure installation process should be implemented. All the configurations should be defined, implemented and documented, to be maintained over time.

### **Vulnerable and Outdated Components (A06-2021)**

The components of a software can be vulnerable, unsupported, or out of date. To prevent this issue, the following practices should be adopted:

- **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**.

### **Identification and Authentication Failures (A07-2021)**

Authentication, confirmation of user identity and session management are critical to protect against authentication-related attacks. To prevent this issue, the following practices should be adopted:

- 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.

### **Software and Data Integrity Failures (A08-2021)**

Relates to code and infrastructure that does not protect against integrity violations. An example is the use of untrusted CI/CD pipelines, or the use of unverified software libraries. Attackers can exploit these weaknesses to inject malicious code into the application.

To prevent this issue, we should always ensure that all dependencies come from trusted sources, and verify their integrity.

### **Security Logging and Monitoring Failures (A09-2021)**

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. It's important to implement proper logging and monitoring mechanisms, to detect security incidents, without exposing sensitive information.

### Server-Side Request Forgery (A10-2021)

Already discussed in Section 2.1.

## 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.

### 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, without proper validation or sanitization, an attacker can craft input that changes the query semantics.

### 3.2 Example attack

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

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

the resulting query becomes:

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

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

### 3.3 SQL Injection type

There are two main types of SQL injection techniques:

### 3.3.1 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.

### 3.3.2 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. For example:

```
1  SELECT * FROM users WHERE userId = '1' AND '1'='2';  
   -- Expected to return no results as '1'='2' is  
   false  
2  SELECT * FROM users WHERE userId = '1' AND '1'='1';  
   -- Could return results as '1'='1' is true
```

If we see a different behavior between the two queries, we can infer that the userId '1' exists in the database.

- **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. For example, we could use the following payloads:

```
1  SELECT * FROM users WHERE userId = '1' AND sleep(10)  
   ; -- Introduces a delay of 10 seconds
```

If the application takes longer to respond, we can infer that the userId '1' exists in the database.

- **Out-of-Band:** this technique is used when the application show the same behavior regardless of the query executed. The attacker uses other channels to retrieve the output of the query, such as HTTP requests.

### 3.4 SQL Injection mitigation

To prevent SQL injection attacks, developers should implement the following best practices:

- 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. This allow to separate code from data, preventing the execution of malicious SQL code.

## 4 XSS (Cross-Site Scripting)

*Cross-Site Scripting* (XSS) attacks are a type of injection in which malicious scripts are injected into trusted websites. An attacker uses a web application to send malicious code, generally in the form of a browser-side script, to a different end user. 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 user input is directly included in a web page output without proper validation or sanitization. In such cases, an attacker can inject malicious code that is executed by the victim's browser. A typical reflected XSS attack follows this flow:

1. The attacker identifies a website that reflects user-controlled input in its responses;
2. The attacker crafts a malicious URL containing injected client-side code;
3. The attacker uses social engineering techniques (e.g., email or messaging) to induce a victim to visit the malicious URL;
4. The victim clicks the URL, and the injected code is executed in the context of the vulnerable website.

### 4.2 Example

Consider a web application that allows users to search for content and displays the searched term on the results page. Suppose the request is sent via the URL:

`http://search.com/?query=<UserInput>`

The server dynamically generates the following response:

```
1 <html>
2   <body>
3     Search results for: <UserInput>
4     <ol>
5       <li>http://example.com/result1</li>
6       <li>http://example.com/result2</li>
7     </ol>
8   </body>
9 </html>
```

If the attacker provides a malicious input such as:

```
<script>alert('XSS')</script>
```

the resulting URL becomes

```
http://search.com/?query=<script>alert('XSS')</script>.
```

When the victim visits this URL, the server reflects the input without sanitization, producing the following response:

```
1 <html>
2   <body>
3     Search results for: <script>alert('XSS')</script>
4     <ol>
5       <li>http://example.com/result1</li>
6       <li>http://example.com/result2</li>
7     </ol>
8   </body>
9 </html>
```

As a consequence, the browser interprets the injected code as legitimate JavaScript and executes it in the security context of `search.com`. This allows the attacker to run arbitrary client-side code, potentially enabling actions such as session hijacking, credential theft, or manipulation of page content.

### 4.3 XSS Types

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

#### 4.3.1 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. This is the most common type of XSS, and an example was already shown in the previous section.

#### 4.3.2 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.

```
1 <!-- User submits comment containing: -->
2 <div>Nice post!</div><script>alert('xss')</script>
3
4 <!-- Application stores the comment and renders it-->
5 <div class="comment">
6   <div>Nice post!</div><script>alert('xss')</script>
7 </div>
```

**Example:** 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.

#### 4.3.3 DOM-Based

A form of XSS in which the entire unsafe data flow occurs in the browser. For example, the malicious payload could be included in the URL of the page, and a sensitive JavaScript operation (e.g., `document.write`, `innerHTML`, ...) uses the data to modify the DOM in an unsafe way. In this case, the server does not reflect or store the malicious input; instead, the vulnerability arises from client-side JavaScript code that processes untrusted data without proper validation or sanitization.

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

### Vulnerable example (client-side)

## 4.4 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. After that, the three main types of request forgery attacks will be discussed:

- Session Hijacking;
- Cross-Site Request Forgery (CSRF);
- Server-Side Request Forgery (SSRF).

## 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 (e.g., item in the shopping cart).

Cookies are generated by the server when a new client connects to it. The server sends the cookie to the client browser, which stores it and sends it back to the server with every subsequent request. An attacker could exploit cookies in several ways:

- **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. Given this, is easy to understand that if an attacker steals a session ID, he could impersonate the authorized user. 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 using the same session ID;
- **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 session ID by trying multiple combinations until they find a valid one.

**Prevention:**

- To prevent MITM attacks, use encrypted communication channels;
- For session fixation, 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

*Server-Side Request Forgery* (SSRF) attack involves an attacker who can abuse server functionality to 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 two 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` in the requests.
- **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. It is a type of injection attack, in which no direct code injection occurs, but rather the injected data is used to compose commands that are executed by the system.

**Example:** An application takes a parameter as input from a user and uses it to construct a command:

```
<?php
2  //http://yourdomain.com?par=value
3  $command = "ls " . $_GET['par'];
4  $output = exec($command);
?>
```

An attacker could exploit this by providing a value such as `; rm -rf /`, resulting in the execution of both the `ls` command and the `rm -rf /` command, which would delete all files on the server.

**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

The vulnerabilities related to file access arise when an application improperly manages file operations, allowing unauthorized users to read, write, or execute files on the server. In the following sections, we are going to analyze the most common vulnerabilities related to 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). We can distinguish two main types of file inclusion vulnerabilities:

- **Local File Inclusion (LFI)**: 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)**: 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 File Access - 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 “../” ) to access files the application did not intend to expose.

For example, an attacker could try to access sensitive files by manipulating the URL as follows:

```
<?php
$template = 'blue.php';
if (isset($_COOKIE['TEMPLATE']))
4     $template = $_COOKIE['TEMPLATE'];
5
include("/home/users/phpguru/templates/" . $template);
?>
```

In this example, the application includes a template file based on the value of the `TEMPLATE` cookie. An attacker could set the cookie to a value like “../../../../etc/passwd” to attempt to access the system's password file.

### 7.3 Filename-based 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

To mitigate file access vulnerabilities, consider the following best practices:

- 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. Always validate and sanitize user inputs to prevent directory traversal attacks;
- Use file descriptors, rather than filenames.

# 8 Error Handling

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

Most of the attacks always begin with reconnaissance activities, where the attacker tries to gather as much technical information as possible about the target system.

**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:** unhandled exceptions or errors can cause the program to terminate unexpectedly, leading to a denial of service;
- **Information leakage:** detailed error messages such as stack traces may reveal sensitive information about the application's internal workings.

Improper error handling and unhandled exceptions can be exploited by attackers to collect useful information.

**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.

**Example:** Consider a web application that allows users to view their profile information by accessing a URL like:

```
https://example.com/user/profile?user_id=123
```

If authorization checks are not implemented, an attacker could change the `user_id` parameter to another user's ID (e.g., `user_id=124`) and gain access to that user's profile information.

**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

Client-side vulnerabilities are security weaknesses that exist within the client-side components of a web application. Since these components are often exposed to users, they can be manipulated or bypassed by attackers.

### 10.1 Validation Bypass

*Client-side validation bypass* refers to techniques used to get around 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 back-end. 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`). 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

To understand overflow vulnerabilities, it is important to have a basic understanding of how memory is organized in a running process.

### 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 with a fixed size;
- 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-validated string may be used as a *format string* in certain functions (e.g., `printf` in C/C++). If an attacker can insert format specifiers (like `%x`, `%s`, `%n`) into the string, they can read or write arbitrary memory locations. This is possible if the formatting option has an undeclared number of arguments, such as in the function:

```
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.

**Example:** Consider the following C code snippet:

```
#include <stdio.h>
int main() {
3   char user_input[100];
4   printf("Enter a string: ");
5   gets(user_input);
6   printf(user_input); // Vulnerable to format string
   attack
7   return 0;
}
```

If an attacker inputs the string `"%x %x %x %x"`, the program will print out four values from the stack, potentially revealing sensitive information.

**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

To understand integer overflow, it is important to know how integers are represented in computer memory.

**Integer Representation - 2's complement notation:** To allow positive and negative values, integers are usually represented using the *two's complement notation*. It is a binary representation where the *Most Significant Bit* (MSB) indicates the sign of the integer (0 for positive, 1 for negative). Given a sequence of  $n$  bits, the representable range of values is:

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

When working with signed integers using this representation, exceeding the upper or lower limit of this range causes the MSB value to change. This results in the value wrapping around to the opposite side of the range.

**Example:** For an 8-bit signed integer:

$$[-128, 127]$$

Adding 1 to 127 results in  $-128$ :

$$127 + 1 = 01111111 + 00000001 = 10000000 = -128$$

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 Supply Chain Attack

To understand supply chain attacks, it's essential to grasp the concept of dependencies in software development.

### 12.1 Dependencies in Software Development

In modern software development, applications often rely on external libraries, frameworks, and tools to enhance functionality and to avoid reinventing the wheel. These external components are known as dependencies, that are usually managed through package managers (like npm, Maven, pip, etc.).

Dependencies can be:

- **Direct dependencies:** libraries or packages that are explicitly included in a project's codebase. We are the owners of these dependencies;
- **Transitive dependencies:** libraries that are not directly included but are dependencies of the direct dependencies.

Dependencies can introduce vulnerabilities in our software, if they contain security flaws or if they are compromised.

### 12.2 What is a Supply Chain?

A software supply chain refers to the entire ecosystem of components, tools, and processes involved in the development, distribution, and maintenance of software applications. Usually, the inventory of a software supply chain is declared in a **Software Bill of Materials (SBOM)**.

### 12.3 Supply Chain Attacks

A supply chain attack targets third-party components or services that are part of the software supply chain, to compromise a target application. They are indirect, in the sense that attackers do not directly attack the target application, but rather exploit vulnerabilities in the components or services it relies on.

### Mechanism of Supply Chain Attacks

The typical steps of a supply chain attack are:

- The attacker identifies a **software vendor** to add malicious code to their product;
- The malicious code is then distributed to the vendor's customers through legitimate software updates;
- When the target application updates or installs the compromised component, the malicious code is executed within the target environment.

### Attack Vectors

Common vectors for supply chain attacks include:

- **Source code:** attackers compromise the proprietary code. Usually the credential of developers are needed as well as the access to the repository;
- **Developer accounts:** attackers gain access to developer accounts to inject malicious code into the software during development or distribution;
- **Dependencies:** attackers compromise third-party libraries or packages that are used as dependencies in the target application;
- **Build tooling:** attackers can compromise build tools or CI/CD pipelines to inject malicious code during the build process;
- **Updates:** attackers compromise the software update mechanism, distributing malicious updates to users.

### Mitigation Strategies

To mitigate supply chain attacks, organizations can implement several strategies:

- **Remove unnecessary dependencies:** regularly review and audit dependencies to remove any that are not essential;
- **Use trusted sources:** only use dependencies from reputable sources and verify their integrity;
- **Inventory management:** maintain an up-to-date SBOM to track all components and dependencies used in the software;
- **Monitor for vulnerabilities:** use automated tools to monitor dependencies for known vulnerabilities;
- **Monitor unmaintained dependencies:** be cautious when using dependencies that are no longer actively maintained. Consider finding alternatives or forking and maintaining them internally;

## 13 Security 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.

### 13.1 Verification and Validation

Software verification and validation are two procedures needed for checking that a software system meets specifications and fulfills its intended purpose.

- **Verification:** Are we building the product right?
- **Validation:** Are we building the right product?

More in detail, we can define:

- **Verification:** the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed by the requirements. During this phase, we usually check for bugs in the application;
- **Validation:** the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

### 13.2 Software Verification techniques

Software verification techniques can be classified into two main categories: *dynamic analysis* and *static analysis*.

#### 13.2.1 Dynamic analysis (or Testing)

Dynamic analysis involves 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. In fact, only a subset of possible executions can be tested.

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

- They are based on specifically defined input data. We will see later how to generate them to cover as many scenarios as possible;
- An oracle that describes what the system is supposed to do is needed. It will be necessary to check program results.

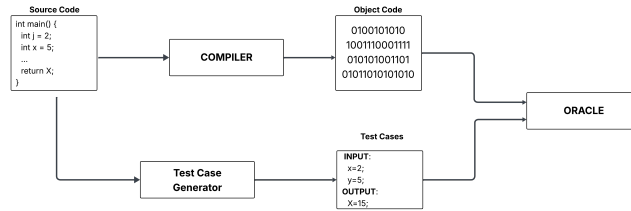


Figure 2: Dynamic analysis process

### 13.2.2 Static analysis

Static analysis examines the software's source code without executing the program. In this context, the oracle is represented by a set of rules or properties that the code must satisfy. It is general and fixed. Compared to dynamic analysis, static analysis has the following characteristics:

- It can demonstrate the absence of specific classes of errors;
- It can analyze all possible executions of the program;
- Doesn't require test cases or execution of the program;

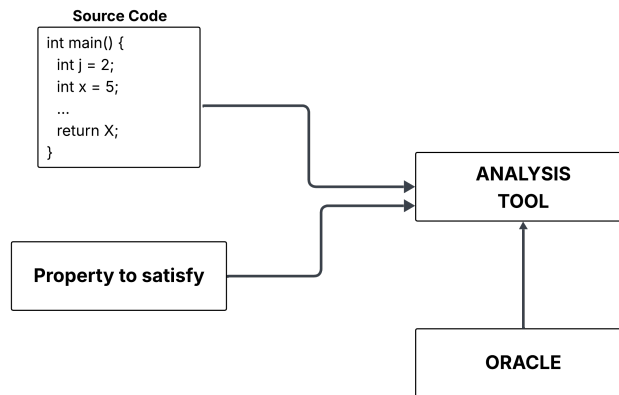


Figure 3: Static analysis process

## 14 Static Analysis

We will now focus on static analysis techniques. We can identify three main categories:

- **Model Checking:** it consists of an automatic technique for verifying finite-state concurrent systems;

- **Pattern-based analysis:** it consists of searching for known patterns in the code that may indicate potential vulnerabilities;
- **Flow-based analysis:** it focuses on the order in which individual statements, instructions, or function calls are executed or evaluated.

In the following sections, we will analyze these techniques in detail.

## 14.1 Model Checking

Model checking is an automated static analysis technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for the given model. It is based on Linear Temporal Logic (LTL).

### 14.1.1 Linear Temporal Logic (LTL)

LTL is used to determine patterns on infinite sequences of states. It is built on three elements:

- **Propositional variables:** atomic statements that can be true or false in a given state;
- **Logical operators:** standard logical connectives such as AND ( $\wedge$ ), OR ( $\vee$ ), NOT ( $\neg$ ), IMPLIES ( $\rightarrow$ );
- **Temporal operators:** operators that express temporal relationships between states, such as:
  - $p$ :  $p$  holds in the current state;
  - $Xp$ :  $p$  holds in the next state;
  - $Fp$ :  $p$  holds at some point in the future;
  - $Gp$ :  $p$  holds globally (always holds in the future);
  - $pUq$ :  $p$  holds until  $q$  holds.

### 14.1.2 Model checking process

The model checker explores all possible states of the system to verify if some property holds. For example:

- Let  $M$  be a model of a system (e.g., a finite state-transition graph);
- Let  $f$  be a property expressed in LTL;
- The model checker verifies whether  $M(f) = \text{True}$  (i.e., if the model  $M$  satisfies the property  $f$ ).

For example, consider a simple model  $G$  of a microwave oven, represented as a state-transition graph:

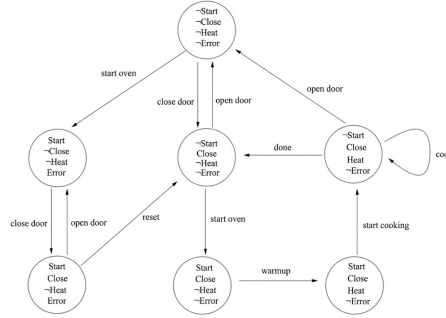


Figure 4: Microwave oven model

We can express properties in LTL, such as:

1.  $G(\text{Heat} \implies \text{Close})$ ;
2.  $G(\text{Start} \implies \neg \text{Heat})$ ;
3.  $G((\text{Start} \wedge \neg \text{Error}) \implies \neg \text{Heat})$ ;

Given the model  $G$ , an initial state and a property, the model will check if the property holds in all possible executions of the model. We have that a property is violated if:

- There exists a state in which the property does not hold;
- There exists a path in the model that leads to a state where the property does not hold;
- There exists a cycle in the model that leads to a state where the property does not hold.

## 14.2 Pattern-based analysis

Pattern-based analysis involves searching for known patterns in the code that may indicate potential vulnerabilities. These patterns can be derived from known security issues, coding standards, or best practices. Often, *static analysis tools* are used to automate this process, scanning the source code for these patterns and flagging potential issues for further review.

## 14.3 Flow-based static analysis

Flow-based static analysis reasons about the logical execution of a program in order to track:

- How data values propagate through the program (data flow analysis);
- How control structures affect the execution of the program (control flow analysis).

To do this, the program is modeled using graphs that represent the flow of data and control within the program.

## 14.4 Model for Flow Analysis

All the defined models are based on the **Control-Flow Graph** (CFG).

### 14.4.1 Control-Flow Graph

A *Control-Flow Graph* (CFG) is a special type of directed graph that represents all paths that might be executed by a program. It is formally defined as a 4-tuple:

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

where:

- $N$ : 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.

For example, consider the following simple program and its corresponding CFG:

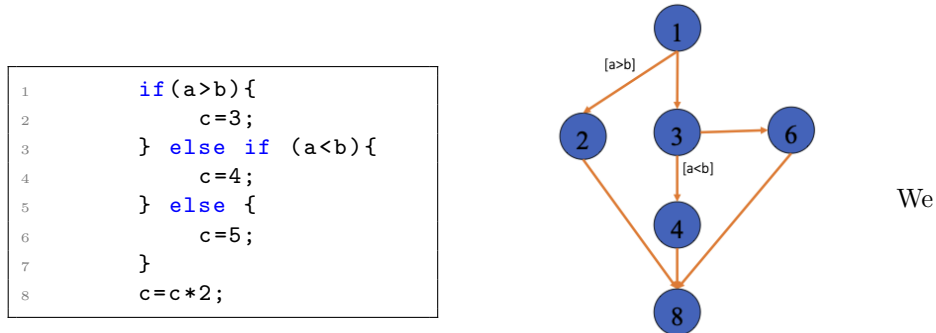


Figure 5: Control-Flow Graph

can then define some important concepts related to CFGs:

**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.

#### 14.4.2 Def-Use pairs

Another important concept in flow analysis is the *Def-Use pair*. It 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. It can be:
  - Declaration;
  - Initialization;
  - Assignment;
  - Values received by a parameter.
- **Use:** extraction of a value from a variable. It can be:
  - Expressions;
  - Conditional statements;
  - Parameter passing;
  - returns.

Given a CFG of the program, we can give the following definitions:

**Def-use pair:** a pair  $(d, u)$  where  $d$  is a node where a variable  $v$  is defined, and  $u$  is a node where the same variable  $v$  is used, and there exists at least one *definition-clear path* from  $d$  to  $u$  in the CFG.

**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.

#### 14.4.3 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.

The data dependence graph is derived from the CFG and makes explicit how values propagate between statements. It is useful for understanding *data dependence*: which statements depend on the values produced by other statements. For example, consider the following Java method that computes the greatest common divisor (GCD) of two integers using the Euclidean algorithm:

```

public int GCD(int x, int y) {
2   int tmp;                // A: Def of x, y, tmp
3   while (y != 0) {         // B: Use of y
4       tmp = x % y;         // C: Use of y, Def of tmp
5       x = y;               // D: Use of y, Def of x
6       y = tmp;             // E: Use of tmp, Def of y
7   }
8   return x;                // F: Use of x
}

```

We can build the following DDG:

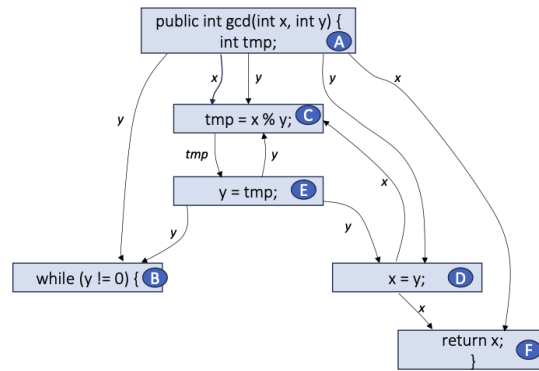


Figure 6: Data Dependence Graph for GCD method

#### 14.4.4 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. Given the code example of the GCD method 14.4.3, we can build the following CDG:

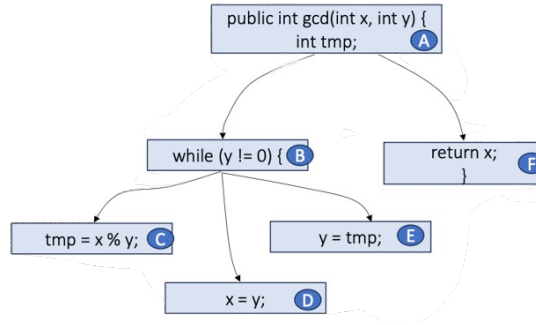


Figure 7: Control Dependence Graph for GCD method

## 14.5 Data-Flow analysis

### 14.5.1 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.

### 14.5.2 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 new 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 the 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:

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

Given these definitions, data flow analysis can be performed following these steps:

1. Build a CFG for the program;
2. Define the set of equations between  $In(n)$  and  $Out(n)$  for each node  $n$  in the CFG;
3. Solve the equations by finding a solution.

### 14.5.3 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.

#### 14.5.4 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.

#### 14.5.5 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);

## 15 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.
- **Untainted Variable:** a variable is considered untainted if it contains data that has been properly sanitized or validated;

It's important to ensure that tainted variables should never be used in security-critical statements (such as system calls, database queries, file operations). There are two main types of taint analysis:

- **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.

## 15.1 Static taint analysis

Before diving into the implementation of static taint analysis, we need to define some key concepts:

- **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 analyze all the possible data flow of the program. This is done following these steps:

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

Since we want to analyze all possible data flows, static taint analysis is usually implemented using data flow analysis techniques.

### 15.1.1 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:  $x \rightarrow \{T, F\}$ ;
- **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 permutation 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. Formally:

$$In(n) = \bigcup_{p \in pred(n)} Out(p)$$

- **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:

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

∪

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

This means that, at a statement  $n$ , a variable  $x$  is considered tainted if it is directly assigned a value from an untrusted source or if it is derived from another tainted variable  $y$ .

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

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

∪

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

This means that, at a statement  $n$ , a variable  $x$  is considered untainted if it is sanitized or if it is derived solely from untainted variables.

For example, consider the following code snippet:

```

def foo:
2   y = 5
3   x = input()
4   x = check(x)
5   z = x + y
6   return z

```

We can say that:

- at line 2, variable **x** is tainted because it receives data from an untrusted source (user input);
- at line 3, variable **x** is untainted because it is sanitized by the function `check()`;
- at line 4, variable **z** is also untainted because it is derived from two untainted variables (**x** and **y**).

### 15.1.2 Algorithm

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

---

#### 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:      $In[n] = \cup_{p \in pred(n)} Out(p)$ 
8:      $Out[n] = Gen[n] \cup (In[n] \setminus Kill[n])$ 
9:   end for
10: until no changes in any  $In(n)$  or  $Out(n)$ 

```

---

gorithm initializes the **Gen** and **Kill** sets for each node in the CFG. It then iteratively computes the **In** and **Out** sets for each node until no further changes occur. At the end of the analysis, we can check if any tainted variable reaches a sink without being sanitized, indicating a potential security vulnerability.

### 15.1.3 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.

## 15.2 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.

## 16 Dynamic Analysis

Dynamic analysis is a software verification technique that involves executing the program and observing its behavior. There are two main strategies to perform dynamic analysis:

- **Testing:** executing the program with specific inputs and checking the outputs against expected results;
- **Fuzzing:** generating random or semi-random inputs to the program and observing its behavior for crashes or unexpected outputs;

## 16.1 Testing

We can differentiate 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.

As we said before, testing involves executing the program with specific inputs and checking the outputs against expected results. A **test suite** is 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.

When designing a test suite, several key questions need to be addressed:

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

### 16.1.1 Test coverage criteria

Typically, for a tested software, there are an infinite number of possible inputs and execution paths. This makes it impossible to test all possible scenarios.

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 have 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.

We can now describe the different code coverage criteria.

### 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. We need to design test cases that exercise different portions of the code.

Considering the whole suite, we can define:

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

Given the previous example, we can see that test case 2 covers no new statements compared to test case 1, so it can be removed from the suite.

### 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:

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

For example, consider the previous code snippet. 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:

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

### 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:

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

### 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:

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

### 16.1.2 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. The code must still compile and run after the mutation;
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.

## 16.2 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.

We are not testing functionality, so we don't need to check for expected outputs. The goal is to simply observe the program's behavior when subjected to these inputs.

### 16.2.1 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.

### Application of fuzzing

Fuzzing can be applied to different types of software:

- **Desktop applications:** fuzzing can be used to test desktop applications, focusing on their UI, command line interfaces, and file handling capabilities;
- **Web applications:** fuzzing can be used to test web applications, focusing on URLs, forms and requests;
- **Protocols:** the fuzzer can send generated packets to the tested application or eventually, act as a proxy between two communicating entities;
- **Files format:** the fuzzer can generate malformed or unexpected files to test the application's file parsing capabilities.

### 16.2.2 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:**

- Tend to find simple bugs, while more complex vulnerabilities may require more sophisticated analysis;
- May generate a large number of false positives, that are hard to debug if we don't have access to the source code;
- 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.

### 16.3 Black-box fuzzing

Black-box fuzzing is a fuzzing technique that treats the target program as a "black box", focusing on its external behavior 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.

#### 16.3.1 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. Since the inputs are randomly generated, it may not be effective in finding specific types of bugs or vulnerabilities.

#### 16.3.2 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.

### 16.3.3 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;

### 16.3.4 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.

## 16.4 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.

### 16.4.1 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.

### 16.4.2 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.

## 17 AI for Vulnerability Detection

Vulnerability detection with AI tools involves the usage of *machine learning algorithms* to identify potential security flaws in software systems. These tools can analyze code, configurations and system behaviors to detect anomalies that may be exploited by attackers. Various task level are considered in vulnerability detection, including:

- **CWE classification:** AI models can be trained to classify vulnerabilities according to the Common Weakness Enumeration (CWE) taxonomy, helping security teams prioritize remediation efforts;
- **Source code classification:** binary classifier between vulnerable and non-vulnerable code snippets. These can be done at different granularity levels (statement-level, function-level, file-level).

Learning-based algorithms can generalize from known labeled vulnerabilities to identify new, previously unseen vulnerabilities, making them valuable tools in the ever-evolving landscape of software security.

In this section, we will explore the main techniques used in AI-based vulnerability detection, along with their limitations. We will take for granted the knowledge of basic machine learning concepts.

### 17.1 Techniques

AI-based vulnerability detection techniques can be broadly categorized based on the algorithms they use.

#### 17.1.1 Sequence-based models

Sequence-based models are machine learning models that process input data as sequences of elements (tokens, characters, etc.), to identify patterns indicative of vulnerabilities. Common sequence-based models include:

- **Recurrent Neural Networks (RNNs);**
- **Long Short-Term Memory (LSTM) networks;**
- **transformer-based models**, the most used.

The code is typically tokenized and represented as sequences before being fed into these models for training.

#### 17.1.2 Graph-based models

*Graph-based models* have shown capabilities in capturing the structural relationships within code, making them suitable for vulnerability detection tasks.

These models work via message passing between nodes in a graph representation of the code, allowing them to learn complex global features. Common graph-based models include:

- **Graph Neural Networks (GNNs);**
- **Graph Convolutional Networks (GCNs);**
- **Graph Attention Networks (GATs).**

Graph-based models often utilize Abstract Syntax Trees (ASTs) or Control Flow Graphs (CFGs) to represent the code structure.

**Training considerations** Before training graph-based models for vulnerability detection, several considerations must be addressed. First, it's important to decide on prediction granularity (file-level, function-level, line-level) before training. Usually an intermediate representation of the code is created to extract graph features.

Source code labelling is another important and challenging task, as it requires expert knowledge to accurately identify vulnerabilities. A common approach is:

- Code lines removed in vulnerability-fixing commits are labelled as vulnerable;
- All the other dependencies of the modified code are considered vulnerable.

## 17.2 Explainable AI

As always, AI-based approaches have some limitations regarding the explainability of the obtained results. In vulnerability detection, it's crucial to understand why a model flagged a particular piece of code as vulnerable.

To provide explanations for vulnerability predictions, several methods can be employed:

- **Model-agnostic methods:** these methods work independently of the underlying model architecture. They try to approximate the model's behavior using simpler, interpretable models.
- **Model-specific methods:** these methods are tailored to specific model architectures, and are based on the knowledge of the model's inner workings.

### SHAP

SHAP (SHapley Additive exPlanations) is a popular model-agnostic method for explaining the predictions of machine learning models. It shows the contribution of each feature to the final prediction. It is based on the breakdown of the prediction into to show the impact of each feature.

In the context of vulnerability detection, SHAP can assign to each token a score indicating its positive or negative contribution to the vulnerability prediction.

## 18 Exercises

### 18.1 Static Taint Analysis

Consider the following code snippet:

```
<?php
2  $userid = $_GET['userid'];
3  $passwd = $_GET['passwd'];
4  $passwd = addslashes($passwd);
5  echo $passwd;
6  $query = "SELECT * FROM users WHERE userid = '$userid'
          AND passwd = '$passwd'";
7  $result = mysql_query($query);
?>
```

1. First, we can build the CFG for the program. We create one node for each statement and connect them based on the control flow of the program.
2. Next, we compute the **Gen** and **Kill** sets for each node;
3. Then, we iteratively compute the **In** and **Out** sets for each node until no further changes occur;
4. Finally, we check if any tainted variable reaches a sink without being sanitized.

**Step 1: CFG** The CFG for the above code snippet can be represented as a line of nodes, where each node corresponds to a statement in the code. Since no control flow statements (like conditionals or loops) are present, the CFG is straightforward.

**Step 2: Gen and Kill sets** We can compute the **Gen** and **Kill** sets for each node remembering the definitions from the Taint Analysis section:

- $Gen(n)$ : the set of variables that are tainted at node  $n$ ;
- $Kill(n)$ : the set of variables that are untainted at node  $n$ ;

Node	Gen[n]	Kill[n]
1	{ $userid$ }	{}
2	{ $passwd$ }	{}
3	{}	{ $passwd$ }
4	{}	{}
5	{}	{}
6	{ $userid = T \vee passwd = T$ }\$query	{ $userid = T \wedge passwd = T$ }\$query}
7	{ $[query=T]$ \$result}	{ $[query=F]$ \$result}

Table 1: Gen and Kill sets for each node in the CFG

We can see that at node 6, the taint status of the variable `$query` is tainted if either `$userid` or `$passwd` is tainted. Since `$userid` is tainted and `$passwd` is untainted (due to sanitization at node 3), we could also conclude that the taint status of `$query` is tainted.

**Step 3: In and Out sets** We can now iteratively compute the **In** and **Out** sets for each node using the data flow equations:

- $In[n] = \cup_{p \in pred(n)} Out(p)$  Defined as the union of all outgoing taint sets from predecessor nodes;
- $Out[n] = Gen[n] \cup (In[n] \setminus Kill[n])$  Defined as the union of the generated taint set at node  $n$  and the incoming taint set minus the killed taint set (variables untainted) at node  $n$ ;

Node	In[n]	Out[n]
1	{}	{userid}
2	{userid}	{userid, passwd}
3	{userid, passwd}	{userid}
4	{userid}	{userid}
5	{userid}	{userid}
6	{query}	{query}
7	{query, userid}	{result, query, userid}

Table 2: In and Out sets for each node in the CFG

**Step 4: Check for vulnerabilities** Finally, we check if any tainted variable reaches a sink without being sanitized. In this case, we can see that at node 6, the variable `$query` is tainted because it depends on the tainted variable `$userid`. We also have that `$query` is used in a sink (the SQL query execution) without being sanitized. Therefore, we can conclude that there is a potential SQL injection vulnerability in the code snippet due to the tainted variable `$userid` reaching the sink without sanitization.

**Important** When considering the taint status of variables after being sanitized, we must ensure that the sanitization function effectively removes all the taint for the specific vulnerability. For example, in this case, the function `addslashes()` is used to sanitize `$passwd`. It is effective against *SQL Injection* attacks, but it does not affect the taint status in case of other vulnerabilities, such as *Cross-Site Scripting* (XSS).

## 18.2 Test Coverage

The exercise consists in calculating the code coverage for a set of test cases, given a code snippet. For example, consider the following code snippet:

```

1  if((a==0) || (b==0)){
2      *res = 0;
3      return 1;
4  } else if(a==1){
5      *res = b;
6      return 1;
7  } else if(b==1){
8      *res = a;
9      return 1;
10 }
11 *res = a*b;
12 return 1;

```

The test cases provided are:

1. **INPUT:** a=0, b=0; **EXPECTED OUTPUT:** 0
2. **INPUT:** a=0, b=1; **EXPECTED OUTPUT:** 0
3. **INPUT:** a=1, b=3; **EXPECTED OUTPUT:** 3
4. **INPUT:** a=2, b=1; **EXPECTED OUTPUT:** 2
5. **INPUT:** a=2, b=2; **EXPECTED OUTPUT:** 4

We can now compute all the coverage criteria described in the Dynamic Analysis chapter.

### Statement coverage

Statement coverage aims at ensuring that all possible statements in the code are executed at least once during testing. It is defined as:

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

In the given code snippet, there are a total of 11 statements. For each test case, we can track which statements are executed and compute the coverage.

Test Case	a	b	res	coverage%
1	0	0	0	27%
2	0	1	0	27%
3	1	3	3	36%
4	2	1	2	45%
5	2	2	4	45%

Table 3: Test cases for statement coverage

We can see that test case 2 covers no new statements compared to test case 1, so it can be removed from the suite. For example, test case 3 covers the statements

on line 1,4,5,6, resulting in 4 out of 11 statements being covered, which gives a coverage of 36%.

Given the full test suite, we can also compute:

- **Total suite coverage:** considering all test cases together, we can see which statements are covered. In our case, we cover all statements, resulting in a total coverage of 100%;
- **Additional suite coverage:** we can compute the additional coverage following this procedure:
  1. Rank the test cases based on their individual coverage, select the one with the highest coverage first;
  2. Compute the additional coverage provided by the selected test case;
  3. Select the next test case that provides the highest additional coverage, and repeat until all test cases are selected.

### Decision coverage

In decision coverage, we want to ensure that all possible outcomes of all boolean expressions are executed at least once during testing. It is defined as:

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

In the given code snippet, there are a three decision points, each with two possible outcomes (true and false), resulting in a total of 6 decision outcomes. For each test case, we can track which decision outcomes are executed and compute the coverage.

Test Case	a	b	res	coverage%
1	0	0	0	16.6%
2	0	1	3	33.3%
3	2	1	2	50%
4	2	2	4	50%

For example, test case 2 covers  $D_1 = F$  and  $D_2 = T$ , resulting in 2 out of 6 decision outcomes being covered, which gives a coverage of 33.3%.

To compute the full suite coverage metrics, we can create a table of all decision outcomes and mark which ones are covered by the test cases:

Decision Point	True	False
$D_1$	$T_1$	$T_2, T_3, T_4$
$D_2$	$T_2$	$T_3, T_4$
$D_3$	$T_3$	$T_4$

- **Total suite coverage:** from the table, we can see that all decision outcomes are covered by the test cases, resulting in a total coverage of 100%;
- **Additional suite coverage:** From the computed table, we can identify the additional coverage provided by each test case. It's important to note that each decision outcome has to be covered only once.

Given the computed table, we can see that:

- $T_4$  covers 3 additional decision outcomes:  $D_1 = F$ ,  $D_2 = F$ ,  $D_3 = F$ . Its additional coverage is:

$$\text{coverage} = \frac{3}{6} \cdot 100 = 50\%$$

- We still need to cover all the *True* outcomes. Since every test case covers only one *True* outcome, we can select one test case arbitrarily.
- For example, we can select  $T_3$  which covers  $D_3 = T$ . Its additional coverage is:

$$\text{coverage} = \left( \frac{3}{6} + \frac{1}{3} \right) \cdot 100 = 66.7\%$$

- We can repeat the same for  $T_2$  and  $T_1$ , resulting in a final additional coverage of 100%.

### Condition Coverage

Similar to decision coverage, condition coverage aims at ensuring that all possible outcomes of all individual boolean conditions within decision points are executed at least once during testing. It is defined as:

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

In the given code snippet, there are a total of 4 individual conditions. Thus, there are 8 possible condition outcomes (true and false for each condition). For each test case, we can track which condition outcomes are executed and compute the coverage.

Test Case	a	b	res	coverage%
1	0	0	0	12.25%
2	1	3	3	37.5%
3	2	1	2	50%
4	2	2	4	50%
5	1	0	0	25%

For example, test case 2 covers conditions  $(a == 0) = F$ ,  $(b == 0) = F$  and  $(a == 1) = T$ , resulting in 3 out of 8 condition outcomes being covered, which

gives a coverage of 37.5%.

To compute the full suite coverage metrics, similarly to decision coverage, we can create a table of all condition outcomes and mark which ones are covered by the test cases:

Condition	True	False
$(a == 0)$	$T_1$	$T_2, T_3, T_4, T_5$
$(b == 0)$	$T_5$	$T_2, T_3, T_4$
$(a == 1)$	$T_2$	$T_3, T_4$
$(b == 1)$	$T_3$	$T_4$

- **Total suite coverage:** from the table, we can see that all condition outcomes are covered by the test cases, resulting in a total coverage of 100%;
- **Additional suite coverage:** is computed similarly to decision coverage.

### Branch coverage

To compute branch coverage, we first need to identify all the edges in the control flow graph (CFG) of the code snippet. The CFG for the given code snippet can be represented as follows:

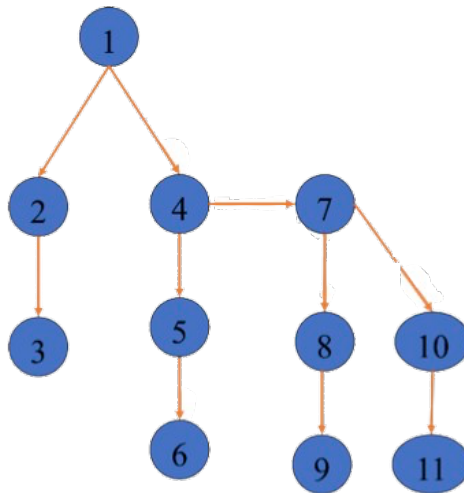


Figure 8: Control Flow Graph for branch coverage

In the CFG, we can identify a total of 10 edges. For each test case, we can track which edges are executed and compute the coverage.

Test Case	a	b	res	coverage%
1	0	0	0	20%
2	1	3	3	30%
3	2	1	2	40%
4	2	2	4	40%

For example, test case 2 covers nodes 1, 4, 5, 6 so it covers edges (1,4), (4,5), (5,6). We cover 3 out of 10 edges, resulting in a coverage of 30%.

To compute the full suite coverage metrics, we can create a table of all edges and mark which ones are covered by the test cases:

Edge	Covered by Test Cases
(1,2)	$T_1$
(2,3)	$T_1$
(1,4)	$T_2, T_3, T_4$
(4,5)	$T_2$
(5,6)	$T_2$
(4,7)	$T_3, T_4$
(7,8)	$T_3$
(8,9)	$T_3$
(7,10)	$T_4$
(10,11)	$T_4$

- **Total suite coverage:** from the table, we can see that all edges are covered by the test cases, resulting in a total suite coverage of 100%;
- **Additional suite coverage:** also in this case, it is computed similarly to decision coverage.

### 18.3 Mutation Testing

For this exercise, we are usually given a code snippet, a set of test case and a set of mutants. The goal is to determine which mutants are killed by the test cases. Consider the following code snippet:

```

1  public boolean isPositive(int number) {
2      boolean result = false;
3      if (number >= 0) {
4          result = true;
5      }
6      return result;
7  }
```

The test cases provided are:

1. **INPUT:** number=20; **EXPECTED OUTPUT:** true
2. **INPUT:** number=0; **EXPECTED OUTPUT:** true

The mutants provided are:

1. **Change line 3:** change `>=` with `>`;
2. **Change line 3:** change `>=` with `False`;
3. **Change line 6:** change `result` with `!result`;

We can now evaluate each mutant against the test cases to determine if they are killed.

Mutant	Test Case 1	Test Case 2	Killed
1	true	false	Yes
2	false	false	Yes
3	false	false	Yes

Table 4: Mutant evaluation against test cases

We can see that all three mutants are killed by the provided test cases. The mutation score can be calculated as:

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}} \cdot 100 = \frac{3}{3} \cdot 100 = 100\%$$