1 Fundamental Network Services and Ports

In the study of cybersecurity, understanding the foundational communication protocols and their associated port numbers is crucial, as these ports represent the initial attack surface for many exploits. Network protocols define the rules for data exchange, while ports are logical endpoints used to differentiate between simultaneous services running on a single host. Monitoring, securing, and hardening these default ports are essential practices for network defense.

Common Network Protocols and Default Ports

The following table summarizes key protocols frequently encountered in network environments:

Protocol	Port(s)	Function/Relevance
FTP	20 (Data), 21 (Control)	**File Transfer Protocol**. Used for transferring files between a client and server. Due to its use of cleartext for credentials, it is often replaced by SFTP or FTPS.
SSH	22	**Secure Shell**. Provides secure, encrypted access for remote command execution and file transfer. A critical administrative tool.
Telnet	23	Used for interactive, text-oriented communication. Highly insecure as it transmits data in cleartext, making it obsolete for modern network management.
SMTP	25	**Simple Mail Transfer Protocol**. Used for sending email messages between servers.
DNS	53	**Domain Name System**. Translates domain names into IP addresses, a critical infrastructure service.
HTTP	80	**Hypertext Transfer Protocol**. The foundation of data communication for the World Wide Web, typically unsecured.
POP3	110	**Post Office Protocol version 3**. Used by local email clients to retrieve emails from a mail server.
LDAP	389	**Lightweight Directory Access Protocol**. Used to access and manage directory services information, such as user accounts and centralized policies.
HTTPS	443	**HTTP Secure**. Encrypted version of HTTP, utilizing TLS/SSL for secure communication over the web.
RDP	3389	**Remote Desktop Protocol**. Proprietary protocol developed by Microsoft, allowing users to remotely connect to another computer. High-value target for attackers seeking lateral movement.

2 The Evolving Classification of Security Professionals

The term 'hacker' carries various connotations depending on the intent behind the access or manipulation of computer systems. In the field of cybersecurity, practitioners are generally categorized by the nature of their activities, often described using the metaphor of hat colors, distinguishing between ethical, malicious, and ambiguous behavior.

2.1 White Hat Hackers

A **white hat hacker** operates legally and ethically, serving as an asset to organizations by proactively identifying and remediating vulnerabilities. They are frequently referred to today as a **penetration tester** or Ethical Hacker. Their primary function is to simulate real-world attacks against a target system, but only with the explicit, written permission and authorization of the system owners. This sanctioned activity ensures that security flaws are found and fixed before malicious actors can exploit them.

2.2 Black Hat Hackers (Crackers)

In direct contrast, a **black hat hacker** gains unauthorized access to systems with the intention of causing harm, stealing data, disrupting services, or achieving financial gain through illegal means. Their actions are malicious and violate ethical and legal boundaries. Black hat hackers are sometimes referred to as **crackers**, a term that historically emphasized the breaking of security measures, particularly those related to intellectual property.

2.3 Gray Hat Hackers

A **gray hat hacker** occupies the middle ground. While they may generally be law-abiding citizens, they sometimes venture into activities that technically violate laws or ethical guidelines, often without inherently malicious intent. For example, a gray hat might discover a vulnerability in a company's system without permission, but then notify the company of the flaw. Although their ultimate goal may be public disclosure or security improvement, their initial unauthorized access classifies them distinctly from white hat professionals.

3 Penetration Testing Methodology

3.1 Defining Penetration Testing

Penetration testing is a controlled, authorized simulation of a cyber attack against a computer system, network, or application. Professionals who conduct these operations are known as **Ethical Hackers** or **Pen Testers**. The goal is to obtain authorization to simulate a full cyber attack against a target to identify security weaknesses. Pen Testers are typically employed by specialized security firms and provide a comprehensive report detailing any vulnerabilities found and concrete steps on how to fix the identified issues.

3.2 Five High-Level Penetration Testing Stages

The penetration test follows a structured methodology to ensure comprehensive coverage. One common framework divides the process into five distinct stages:

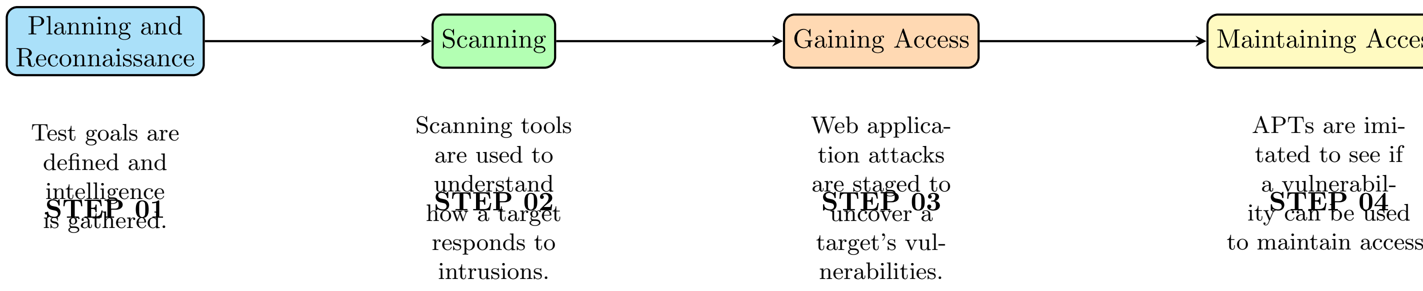
Planning and Reconnaissance
Scanning
Gaining Access
Maintaining Acces
Test goals are defined and intelligence is gathered.
STEP 01
Scanning tools are used to understand how a target responds to intrusions.
STEP 02
Web application attacks are staged to uncover a target's vulnerabilities.
STEP 03
APTs are imitated to see if a vulnerability can be used to maintain access
STEP 04

Figure 1: Five High-Level Stages of Penetration Testing

3.3 Detailed Seven-Step Penetration Testing Flow

A more granular representation of the penetration testing process highlights the iterative cycle of discovery, exploitation, and post-exploitation actions, particularly focusing on expanding scope and moving laterally within the target network.

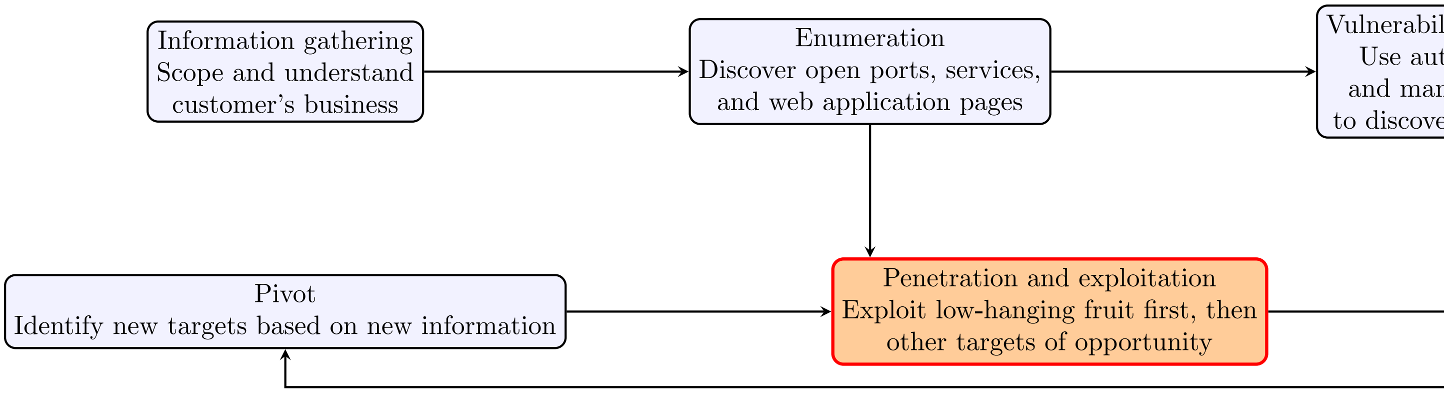
Information gathering
Scope and understand
customer's business
Enumeration
Discover open ports, services,
and web application pages
Vulnerabil
Use aut
and man
to discove
Pivot
Identify new targets based on new information
Penetration and exploitation
Exploit low-hanging fruit first, then
other targets of opportunity

Figure 2: Detailed Penetration Testing Flow Highlighting Iterative Steps

The steps in this detailed process cycle through intelligence gathering, discovery, and successful compromise:

1. **Information Gathering**: Establishing the scope and thoroughly understanding the customer's business and network topography.
2. **Enumeration**: Discovering exposed network services, open ports, and accessible web application pages.
3. **Vulnerability Identification**: Employing both automated vulnerability scans and manual techniques to find exploitable weaknesses within the identified services.
4. **Attack Surface Analysis**: Utilizing the collected vulnerability data to determine feasible attack vectors and developing a concrete plan of action for exploitation.
5. **Penetration and Exploitation**: The critical phase where the tester actively exploits vulnerabilities, prioritizing easy targets (*low-hanging fruit*) before attempting more complex exploits.
6. **Privilege Escalation**: Post-exploitation actions focused on elevating permissions (e.g., from a standard user account to a root or system administrator account) to gain greater control, often followed by further enumeration.
7. **Pivot**: Utilizing the compromised system as a launching point to identify and attack new targets laterally on the network that were previously shielded. This new information feeds back into the exploitation phase, restarting the discovery cycle on a new host.

4 The Open Web Application Security Project (OWASP)

The **Open Web Application Security Project (OWASP)** is a globally recognized, non-profit foundation dedicated to improving the security of software. OWASP achieves its mission by focusing on practical security development and providing resources for developers and security professionals.

OWASP primarily provides three key resources:

- **Tools**: Offering open-source projects and utilities designed to test and assess security.
- **Community and Networking**: Facilitating collaboration among security experts worldwide.
- **Education and Training**: Developing materials to increase security awareness and competency.

4.1 The OWASP Top Ten List

OWASP is renowned for publishing the **OWASP Top Ten**, a periodically updated report detailing the ten most critical security flaws found in software applications. This list serves as a vital benchmark for developers and organizations worldwide to prioritize security risks. The list is released approximately every three to four years, with the current iteration dating from 2021.

Vulnerabilities are assessed and ranked based on four main criteria:

- **Exploitability**: How simple or complex the attack technique is.
- **Prevalence**: How frequently the vulnerability is discovered in web applications.
- **Detectability**: How easy it is for penetration testers or automated tools to find the flaw.
- **Technical Impact**: The potential severity of the harm caused if the vulnerability is successfully exploited.

4.2 Evolution of the OWASP Top Ten (2017 to 2021)

The transition from the 2017 list to the 2021 list involved significant reorganization, reflecting modern application architectures, the rise of API security concerns, and supply chain threats.

2017 Rank	**2017 Flaw**	$\longrightarrow$	**2021 Flaw (2021 Rank)**
A01	Injection	$\longrightarrow$	Injection (A03)
A02	Broken Authentication	$\longrightarrow$	Identification and Authentication Failures (A07)
A03	Sensitive Data Exposure	$\longrightarrow$	Cryptographic Failures (A02)
A04	XML External Entities (XXE)	$\longrightarrow$	Insecure Design (**A04 - New**)
A05	Broken Access Control	$\longrightarrow$	**Broken Access Control (A01)**
A06	Security Misconfiguration	$\longrightarrow$	Security Misconfiguration (A05)
A07	Cross-Site Scripting (XSS)	$\longrightarrow$	(Merged into A03 Injection)
A08	Insecure Deserialization	$\longrightarrow$	Software and Data Integrity Failures (**A08 - New**)
A09	Using Components with Known Vulnerabilities	$\longrightarrow$	Vulnerable and Outdated Components (A06)
A10	Insufficient Logging & Monitoring	$\longrightarrow$	Security Logging and Monitoring Failures (**A09 - N**
A10 (2021): Server-Side Request Forgery (SSRF)			

The most notable change was the promotion of **Broken Access Control** to the number one spot, confirming that unauthorized access remains the most prevalent and impactful application flaw.

4.3 A1: Broken Access Control

Access control enforces security policy such that users are restricted to actions and resources authorized by their permissions. A **Broken Access Control** vulnerability arises when an application fails to properly enforce these restrictions, allowing users to manipulate parameters or circumvent checks to act outside of their intended boundaries.

Failures in access control can lead to devastating security consequences, as they allow an attacker to behave as a more privileged user. These failures typically result in:

- Unauthorized disclosure of sensitive information.
- Modification or destruction of data, often system-wide.
- The ability to perform high-level business functions or administrative tasks that should be restricted.

Robust access control mandates validating every user request server-side, ensuring strict adherence to the principle of least privilege.

5 Access Control Vulnerabilities

Access control, the process by which a system determines whether a user is authorized to perform a specific action, is a critical component of application security. Failures in this area often lead to serious security breaches, allowing unauthorized viewing, modification, or destruction of data.

The foundation of secure access control rests on the **principle of least privilege**, which dictates that a user or system process should only have the minimum permissions necessary to perform its function. Relatedly, the system should operate under a **deny by default** mechanism, where access is granted only when explicitly permitted. Violations of this principle constitute a major security flaw.

Common access control weaknesses include:

- **Insecure Direct Object Reference (IDOR) and Parameter Tampering**: This occurs when an attacker modifies parameters within the URL (or other data fields) to bypass access controls. For example, changing a user ID from `user=123` to `user=456` to access someone else's account, potentially viewing or editing their data.
- **Vertical and Horizontal Privilege Escalation**: This involves an authorized user gaining access to resources they are not permitted to see. *Horizontal escalation* involves accessing the data of a peer user (e.g., viewing another customer's account). *Vertical escalation* involves a standard user gaining administrative rights or accessing pages meant only for privileged users (often referred to as **force browsing**).
- **Unauthenticated Access and Role Confusion**: This involves elevating privilege by acting as a user without being properly logged in, or accessing system functionality via APIs where authentication or authorization checks are completely missing, particularly for data modification methods like **POST**, **PUT**, and **DELETE**.
- **Metadata Manipulation**: This often involves tampering with or replaying session tokens or claims, such as a **JSON Web Token (JWT)**, to impersonate another user or modify the claims granting access rights.

5.1 Web HTTP Methods and Status Codes

To understand these vulnerabilities, it is necessary to grasp how web communications are handled using the underlying Hypertext Transfer Protocol (HTTP). The web sends data back and forth from the server to the client using specific HTTP Methods:

- **GET**: Used to retrieve data or a resource from the server.
- **PUT**: Used to send updated data to the server, typically for replacing an existing resource.
- **POST**: Used to send new or created data to the server, often resulting in a new resource.
- **DELETE**: Used to remove data or a specific resource from the server.

Security implementations must ensure that modifying methods (**POST**, **PUT**, **DELETE**) are rigorously checked for authorization, as these methods can permanently alter application state.

When the server responds to a client request, it returns an **HTTP Status Code** to indicate the result of the request. These codes are grouped into five classes:

HTTP Status Code Classes

20X Success Codes

- 200 - OK: The request succeeded.
- 201 - Created: A new resource was created (common response to a POST request).
- 204 - No Content: The server successfully processed the request but is not returning any content (common response to a DELETE request).

30X Redirection Codes

- 301 - Moved Permanently
- 304 - Not Modified

40X Client Error Codes (Security Relevant)

- 400 - Bad Request: The server cannot process the request due to a client error.
- **401 - Unauthorized**: Authentication failed. The request requires user authentication credentials.
- **403 - Forbidden**: Authorization failed. The server understands the request but refuses to authorize access to the resource (e.g., a user is logged in, but lacks the necessary permissions).
- 404 - Not Found

50X Server Error Codes

- 500 - Internal Server Error
- 503 - Service Unavailable

5.2 Mitigating Access Control Failures

Preventing access control vulnerabilities requires strict adherence to security best practices throughout the development lifecycle:

1. **Implement Deny by Default**: This is the foundational countermeasure. All access must be denied unless explicitly permitted by defined authorization rules.
2. **Validate Authorization Context**: Before executing any action, especially those involving data modification (POST, PUT, DELETE), the application must verify two key facts: that the user is authenticated, and that the authenticated user is authorized to perform that specific action on that specific resource (ensuring the correct user is making the change and preventing IDOR).
3. **Use Strong Identifiers**: To prevent simple parameter tampering attacks, identifiers (IDs) should be difficult to guess and non-sequential. Using universally unique identifiers (**GUIDs** or **UUIDs**) instead of simple numeric IDs makes enumeration attempts significantly harder.
4. **Secure Authentication and Session Management**: Use tokens, such as **JWTs** (`https://jwt.io`), for robust authentication. These tokens must be properly signed and validated on the server side to prevent unauthorized tampering or replay attacks.
5. **Enforce Secure Configuration**: Disable web server directory listing to prevent information leakage. Utilize **Cross-Origin Resource Sharing (CORS)** policies strictly to control which external domains can interact with the application's resources.
6. **Monitor and Audit**: Log all access control failures (401 and 403 responses) and review these logs periodically. Frequent failures may indicate active attack attempts or widespread security flaws requiring immediate remediation.

6 A2: Cryptographic Failures

The current OWASP Top Ten list identifies **A2: Cryptographic Failures** (previously referred to as Sensitive Data Exposure) as a critical risk. This category addresses fundamental weaknesses in protecting sensitive data both when it is stored (**at rest**) and when it is transmitted (**in transit**).

Cryptographic failures can lead to the exposure of highly sensitive information, including:

- Passwords and authentication credentials.
- Credit Card numbers and financial data.
- Health Records (subject to regulations like **HIPAA**).
- Personally Identifiable Information (**PII**), such as names, addresses, and social security numbers.

6.1 Assessing Cryptographic Weakness

A comprehensive security review must systematically evaluate the cryptographic hygiene of an application. Key questions to determine vulnerability include:

- **Data Transmission Integrity**: Is any sensitive data transmitted across the network in **clear text**? All traffic, especially authentication and transactional data, must use secure protocols like HTTPS.
- **Algorithm Strength**: What version of cryptography is being used? Older, known-weak hashing algorithms, such as **SHA1** or **MD5**, should be banned and replaced with modern, salted, adaptive hashing methods (e.g., Argon2, bcrypt).
- **Key Management**: Are the default cryptographic keys used in the system weak, or are they being used across multiple servers? How were these keys generated, and are they protected according to industry best practices? Key leakage or weak key generation exponentially increases the risk of successful decryption by attackers.
- **Certificate Validation**: Is the certificate chain obtained from the Certificate Authority (**CA**) properly validated? Have any necessary security certificates expired, potentially forcing users to ignore warnings and establish insecure connections?

6.2 Data Classification and Handling

Effective cryptographic protection begins with data classification. It is essential to determine the sensitivity of all data elements being transmitted and stored to select the appropriate handling methods.

6.2.1 Data Classification

Data must be classified based on its inherent sensitivity and any applicable legal requirements:

- **Confidentiality**: Is the data a company secret, financial record, or regulated PII?
- **Legal Requirements**: Does the data fall under regulatory frameworks such as HIPAA (Health Data) or GDPR/CCPA (PII)?
- **Authentication Credentials**: Is the data a password, PIN code, or security question answer, requiring specific, irreversible protection?

6.2.2 Data Handling Based on Classification

Once classified, the mechanism for storage and transmission must be chosen carefully:

- **Passwords and PIN codes**: These must be protected using one-way transformation mechanisms. They should be **Hashed** using strong, salted algorithms, never encrypted, as they should never need to be converted back into clear text.
- **User Personal Information, Health Information, and Company Private Info**: Data that needs to be reversible (i.e., must be readable by the application later) must be strongly **Encrypted** both at rest and in transit.
- **Public Information (Company and User)**: Information intended to be publicly viewable can be stored and transmitted as **Plain Text**.

6.2.3 Securing Data in Transit

Using inherently secure protocols is vital for protecting data in transit:

- **Web Traffic**: HTTP (Hypertext Transfer Protocol) is not secure. All websites that handle sensitive data must use **HTTPS** (HTTP Secure), which utilizes Transport Layer Security (TLS) encryption.
- **File Transfer**: FTP (File Transfer Protocol) is insecure. Servers must utilize **FTPS** (Secure File Transfer Protocol) or SFTP (SSH File Transfer Protocol) for secure file exchange.
- **Email (SMTP)**: The standard Simple Mail Transfer Protocol (SMTP) is not inherently secure. While there is no universally adopted, mandatory Secure Mail protocol, modern mail services (like Gmail) offer transport layer encryption or allow for the use of add-ons that facilitate end-to-end encryption.

6.2.4 Secure Email Implementation

Achieving true end-to-end secure email typically relies on **Public Key Cryptography**. Add-ons for email clients enable this functionality: the sender uses the recipient's publicly available key to encrypt the message. Only the recipient possesses the corresponding private key necessary to decrypt and read the message. This method ensures confidentiality but requires coordination and key exchange between both the sender and receiver.

7 Secure Handling and Storage of Sensitive Data

The storage and handling of consumer financial information are tightly regulated by the **Payment Card Industry Data Security Standard (PCI DSS) set by the Payment Card Industry (PCI). These standards dictate the required security controls that organizations must implement to protect cardholder data.

The overarching principle of the PCI DSS is **data minimization**. Organizations are strongly discouraged from storing actual credit card numbers. If a web application requires payment information for recurring billing, the recommendation is to employ a strategy known as **tokenization**.

Tokenization involves replacing the sensitive Primary Account Number (PAN) with a non-sensitive, encrypted value (the token) generated by a credit card clearing company, such as First Data. This encrypted token can be stored locally, but since it is worthless outside the specific processing environment, it drastically reduces the organization's compliance scope and liability in the event of a breach.

7.1 Principles of Data Minimization and Protection

Beyond financial data, general principles apply to all sensitive personal information. The fundamental security recommendation is simple: **Do not store sensitive data if it is not absolutely necessary**.

If data storage is unavoidable, several techniques must be employed to mask or protect it:

- **Truncation:** This involves retaining only a portion of the sensitive data, such as keeping only the last four digits of a Social Security Number (SSN) or credit card number, rendering the original data unusable by an attacker.
- **Encryption at Rest:** If the full data must be stored, it must be encrypted. Encryption transforms the data into an unreadable format, requiring a secure key for decryption.
- **Hashing for Comparison:** Data that only needs to be verified or compared (such as passwords) should be run through a one-way hashing algorithm. A hash is impossible to reverse-engineer to retrieve the original input, making it highly secure for comparison and matching purposes.
- **Data Isolation:** Where possible, the most sensitive data should be moved off the public-facing application server and isolated into a hardened, internal system.

7.2 Cryptography and Regulatory Compliance

Preventing data breaches requires proactive measures rooted in policy and technical implementation:

First, all data, whether processed, stored, or transmitted, must be subject to **data classification**. This involves categorizing the information into sensitivity levels (e.g., public, confidential, secret). This classification process must take into consideration various legal and regulatory requirements (e.g., HIPAA, GDPR) that dictate protection mandates.

Technically, systems must enforce robust cryptographic standards:

1. All sensitive data must be **encrypted at rest** (when stored) using up-to-date and standard cryptography algorithms.
2. All data transmission must utilize modern transport layer security, specifically using **TLS 1.2 or above**. Older protocols like SSL and earlier versions of TLS are considered deprecated and highly vulnerable.
3. To prevent leakage of credentials or personal information, applications must be configured to **disable caching of sensitive data** on client-side browsers or intermediary systems.

8 A03: Understanding and Preventing Injection Attacks

Injection Attacks are consistently listed among the most critical application security risks (often designated as A03 by the OWASP Top 10 list). These attacks occur when an application processes **untrusted data** without sufficient validation, allowing an attacker to inject hostile data that is then interpreted by the system as executable commands.

While the most famous and commonly exploited form is the **SQL Injection (SQLi)** attack, injection is a broad category encompassing several types of data execution vulnerabilities, including:

- **NoSQL Injection** (targeting non-relational databases).
- **Command Injection** (allowing attackers to run operating system commands).
- **LDAP Injection** (targeting directory service queries).

8.1 Indicators of Vulnerability

An application is vulnerable to injection attacks if it exhibits any of the following traits:

- **Lack of Validation:** User-supplied data is accepted and processed without stringent input validation, sanitization, or escaping.
- **Unparameterized Queries:** Dynamic database queries are built using simple string concatenation. The safest defense against SQLi is the use of **prepared statements** or parameterized queries, which ensure the database driver distinguishes between executable code (the query structure) and user input (the data).
- **Direct Usage of Hostile Data:** Hostile data is directly used or concatenated into the input stream, allowing the user's input to change the fundamental logic of the underlying command being executed.

9 Relational Database Fundamentals

One of the most efficient and structured ways to store and manage data is through a **relational database**. Relational databases organize data to reveal relationships between different data points, ensuring integrity and consistency.

Relational data is structured into named entities called **Tables**. Tables are composed of **Columns** (which define the data attributes or fields, such as "Name" or "Email") and **Rows** (which represent individual records or instances of data). For example, in a "Users" table, the columns would define attributes like ID, Name, Username, and Password, while each row would contain the specific data for a single user.

These structured tables can be queried, manipulated, and managed using a highly standardized and robust language known as **Structured Query Language (SQL)**.

9.1 Common Database Management Systems (DBMS)

Relational databases are managed by Database Management Systems (DBMS), which are broadly categorized by licensing and use case:

- **Free/Open Source Options:** Popular open-source systems include **MySQL** and **PostgreSQL**. MySQL, which is owned by Oracle, remains one of the most widely deployed free relational databases due to its performance, reliability, and support for multiple operating systems, including Windows, MacOS, and Unix-based platforms.
- **Enterprise Options:** These commercial systems are often deployed in large-scale corporate environments and include Microsoft SQL Server, Oracle, and IBM DB2.

9.2 Data Manipulation: CRUD Operations

All relational database systems must provide a means to manage persistent data through four essential functions, collectively known as **CRUD**: Create, Read, Update, and Delete. These abstract operations map directly to specific keywords in SQL used to interact with the stored data.

SQL CRUD Operations

- **Create:** Used to add new rows of data to a table, corresponding to the SQL command `INSERT INTO`.
- **Read:** Used to retrieve existing data from the database, corresponding to the SQL command `SELECT`.
- **Update:** Used to modify existing data in one or more rows, corresponding to the SQL command `UPDATE`.
- **Delete:** Used to remove one or more rows of data from a table, corresponding to the SQL command `DELETE`.

10 The Fundamentals of SQL Data Manipulation Language (DML)

Structured Query Language (**SQL**) is the standard language for managing and querying data stored in relational database management systems (RDBMS). Data Manipulation Language (DML) encompasses the core operations required to handle data records within a table: Read, Create, Update, and Delete (CRUD). Understanding the precise syntax and structure of these operations is foundational to both database management and security analysis.

10.1 Reading Data: The `SELECT` Statement

The most frequently used SQL operation is the `SELECT` statement, used for retrieving data records from one or more tables. The basic structure requires specifying which columns are desired and from which table the data should be drawn:

```
SELECT [columns] FROM [table]
```

The `[columns]` argument specifies the attributes to be returned. This can be a single column, or multiple column names provided in a comma-delimited list. Alternatively, the asterisk symbol (`*`) is used as a wildcard to signify that all columns from the specified table should be returned.

Examples of SELECT Queries

1. Retrieving all data from the `Users` table:
   `SELECT * FROM Users`
2. Retrieving only the names and emails:
   `SELECT Name, Email FROM Users`

10.2 Filtering Data: The `WHERE` Clause

While the basic `SELECT` retrieves all rows, filtering records based on specific criteria is essential. This is achieved using the `WHERE` clause. The `WHERE` clause restricts the rows returned by applying conditional logic.

The general structure of a filtered query is:

```
SELECT [columns] FROM [table] WHERE [column] = [value]
```

The condition specified after `WHERE` dictates which rows are included in the final result set.

Condition Filtering Examples

1. Retrieving the record where the Name is exactly 'Jeff':
   `SELECT * FROM Users WHERE Name='Jeff'`
2. Retrieving the record where the numeric identifier (`Id`) is 2:
   `SELECT * FROM Users WHERE Id=2`

10.2.1 Advanced Filtering with `LIKE`

For pattern matching rather than exact equality, the `LIKE` operator is employed, often in conjunction with the percent symbol (`%`) wildcard. The wildcard represents zero or more characters.

For instance, to retrieve all users whose names begin with the letter 'J':

```
SELECT * FROM Users WHERE Name LIKE 'J%'
```

10.3 Creating Data: The `INSERT INTO` Statement

The `INSERT INTO` statement is used to add new records (rows) into an existing table. When creating data, it is crucial that the number of columns specified matches the number of values provided, and that the data types align.

The required syntax structure is:

```
INSERT INTO [table] ([columns]) VALUES ([values])
```

Example of INSERT INTO

To add a new user named Hank with Id 4:

```
INSERT INTO Users (Id, Name, Email,...)  VALUES (4, 'Hank', 'hank@okcu.edu',...)
```

10.4 Modifying Data: The `UPDATE` Statement

The `UPDATE` statement is used to modify existing records in a table. It requires specifying the table, using the `SET` keyword to define the new values for specific columns, and, critically, using the `WHERE` clause to specify which rows should be affected.

```
UPDATE [table] SET [column]=[value] WHERE [column]=[value]
```

Caution: It is imperative to include a `WHERE` clause when using `UPDATE`. If the `WHERE` clause is omitted, the specified changes will be applied to *every single record* in the table, potentially leading to widespread data corruption.

Example of UPDATE

To change the name of the user with Id 1 to 'Jeff Maxwell':

```
UPDATE Users SET Name='Jeff Maxwell' WHERE Id=1
```

10.5 Removing Data: The `DELETE` Statement

The `DELETE FROM` statement permanently removes rows from a table. Like `UPDATE`, it relies entirely on the `WHERE` clause to target specific records for deletion.

```
DELETE FROM [table] WHERE [column]=[value]
```

Extreme Caution: Omitting the `WHERE` clause in a `DELETE FROM` statement will result in the deletion of *all records* in the table. This operation is often irreversible without recent backups.

Example of DELETE

To delete the user record with Id 1:

```
DELETE FROM Users WHERE Id=1
```

11 Introduction to SQL Injection

SQL Injection is one of the most common and dangerous web application vulnerabilities. It occurs when an attacker can interfere with the queries that an application makes to its database, allowing them to view, modify, or delete data they are not authorized to access.

11.1 The Context: A Sample User Table

To illustrate the vulnerability, we use a hypothetical `Users` table, representative of an authentication database schema:

Figure 3: Hypothetical `Users` Table Data

Id	**Name**	**Username**	**Email**	**Password**
1	Jeff	Jmaxwell	jmaxwell@okcu.edu	Password#123
2	John	Jsmith	jsmith@okcu.edu	Password#123
3	Jane	jsmith2	jsmith2@okcu.edu	Password#123

A standard login screen checks the provided username and password against a query to this database. If a matching row is found, the user is authenticated.

11.2 How SQL Injection Works

The vulnerability arises primarily from the practice of constructing SQL queries using simple string concatenation based on user input, rather than using parameterized queries.

11.2.1 The Expected Query Construction

If a user attempts to log in with username `jmaxwell` and password `Password#123`, the application code (often written in a language like Java or PHP) dynamically builds the following query string:

```
SELECT * FROM Users WHERE username='jmaxwell' AND password='Password#123';
```

The application code often handles this concatenation similarly to:

```
String sql = "SELECT * FROM Users WHERE username='" + username
           + "' AND password='" + password + "'";
```

11.2.2 Exploitation via Logical Bypass

An attacker exploits this vulnerability by entering malicious characters into the input fields, specifically characters that change the logical structure of the resulting SQL query.

Consider an attacker who enters the username `jmaxwell` and uses a malicious input for the password field: `jmaxwell' or '1'='1' --`.

When the application concatenates this input, the resulting SQL query becomes drastically altered:

```
SELECT * FROM Users WHERE username='jmaxwell'
    OR '1'='1' -- AND password='xxx'
```

(Note: We use `xxx` here to represent whatever else the application expected to wrap the password input.)

Analysis of the Malicious Query:

1. The attacker's first single quote (`'`) closes the quote initiated by the application for the username parameter.
2. The `OR '1'='1'` segment is injected. Since the condition $1 = 1$ is always true, the entire `WHERE` clause now evaluates to true for the first record encountered.
3. The double hyphen (`--`) is a standard SQL comment operator. This operator effectively causes the database engine to ignore the rest of the original query, including the subsequent `AND password='xxx'` section.

Because the `WHERE` clause evaluates to true for the first user record (Id 1, Jeff), the database returns that row. If the application logic simply checks if *any* rows were returned, the attacker is successfully authenticated as Jeff, regardless of the correct password. This ability to inject logical conditions (`OR 1=1`) and terminate the remainder of the query (`--`) demonstrates the core mechanism of a classic SQL injection attack.

12 A04: Insecure Design and Secure Design Principles

12.1 The OWASP Top 10: Insecure Design (A04)

Insecure Design (A04) was introduced to the OWASP Top 10 in 2021, marking a shift toward recognizing architectural and design deficiencies as a major source of application risk. This category is exceptionally broad, encompassing issues that arise not from faulty coding practices, but from fundamental security flaws embedded in the system's architecture.

The most critical symptom of Insecure Design is the **Missing or Ineffective Control Design**. This means that security controls were either entirely overlooked during the planning phase or were designed inadequately to meet modern threat vectors. When evaluating a system architecture, academic focus must therefore shift to critical questions regarding risk:

- What specific **risks** are introduced by this design? This includes risks related to trust boundaries, unauthorized data access, or the inability to enforce separation of duties.

- How are these identified risks currently being **managed** and mitigated? A robust design must clearly articulate the control mechanisms used to reduce the probability or impact of identified threats.

12.2 Defining Secure Design

Secure Design is far more than a technical blueprint or a simple diagram; it is a foundational change in organizational philosophy. It mandates a transformation in thinking, culture, and approach concerning the development of both software and hardware systems.

Definition

Secure Design requires integrating security considerations, defined methodologies, and architectural patterns from the very initial phase of system conceptualization. Its goal is to ensure that security is a non-negotiable, inherent quality of the system, not an add-on or a post-development fix.

Organizations must adopt structured methodologies (such as security requirements engineering and dedicated architecture review) and maintain a constant, corporate-wide drive to prioritize security improvements, starting precisely with the system design.

13 Risk Management through Threat Modeling

13.1 Introduction to Threat Modeling

Threat Modeling is the proactive, structured practice of reviewing current application diagrams, data flow maps, and architectural specifications to identify potential security issues inherent in the design itself. By examining the system from an attacker's perspective, architects can anticipate how assets might be compromised.

Effective threat modeling focuses on critical interrogation points within the system design to ensure adequate controls are in place:

- **Data Flow Analysis:** Defining how information moves between trust boundaries and components.
- **Encryption Verification:** Checking if appropriate cryptographic measures are being used for data in transit and at rest.
- **Access Control Mapping:** Clearly establishing **who** (users, roles, services) has access to **what** specific data or functionality.

13.2 Example: Application Architecture Diagram

The following diagram illustrates a multi-tiered application architecture used for threat analysis. Key elements requiring security scrutiny include the demarcation of trust zones (DMZ, LAN, Restricted Network) and the placement of authentication and encryption controls.

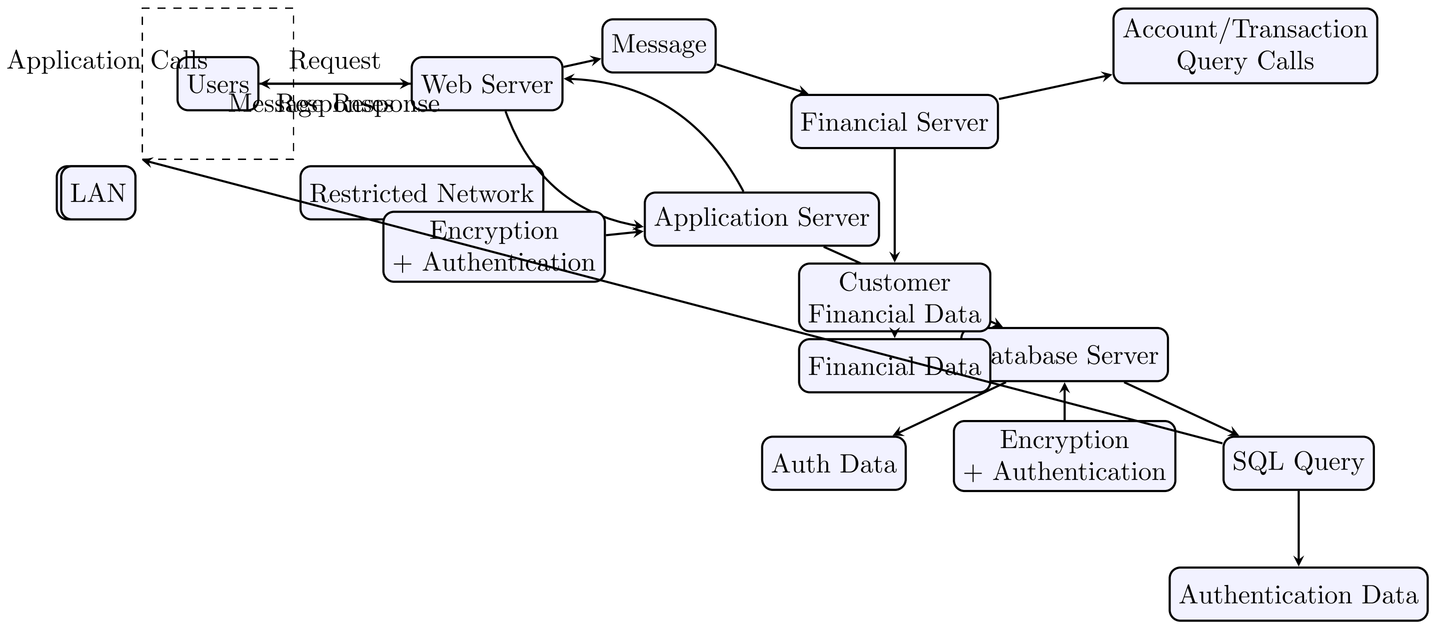

Figure 4: Threat Modeling Data Flow Diagram Example

13.3 Example: Banking Transaction Data Flow

Analyzing business process flows, such as a banking withdrawal, allows for the identification of potential gaps in authorization enforcement. The following diagram focuses on the decision points for authentication (AuthN) and authorization (AuthZ) and highlights how sensitive information (like bad account lists or policy checks) interacts with the core transaction process.

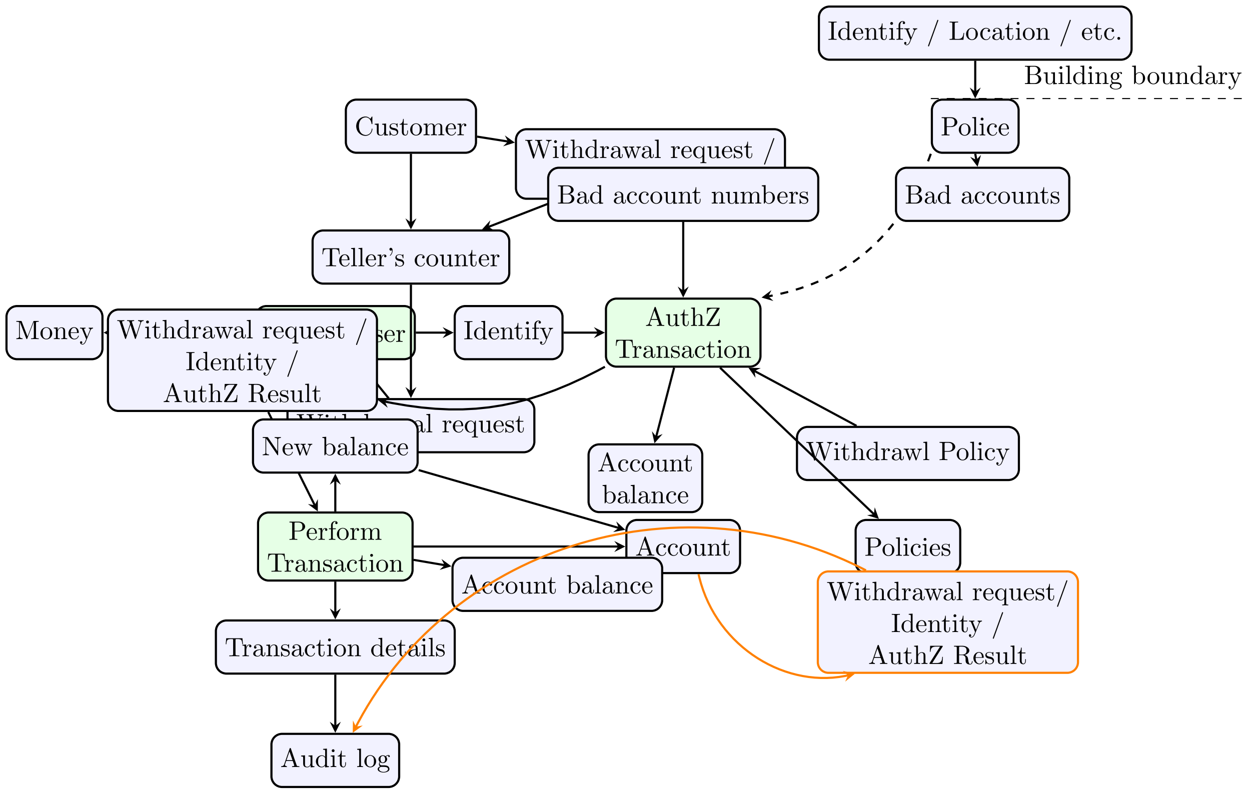

Figure 5: Threat Modeling Example: Banking Withdrawal Data Flow Diagram

14 Data Governance and Flow Analysis

Robust Secure Design necessitates detailed governance over the data itself and strict control over its movement throughout the system architecture.

14.1 Data Management Concerns

Before deployment, system designers must finalize data classification, storage parameters, and access policies:

- **Data Inventory:** What categories of data are being stored? This inventory must guide the subsequent application of security controls.
- **Classification Standards:** How is the data classified (e.g., Public, Confidential, Highly Sensitive)? Classification dictates the mandatory security measures (e.g., access logging, encryption standards).
- **Cryptographic Justification:** What data is being **Hashed/Encrypted**? A clear operational reason must be established for cryptographic protection, and, equally important, a justification (**Why Not?**) must exist for any sensitive data transmitted or stored in the clear.
- **Access Matrix:** Defining an explicit matrix showing **who** (which user role, application service, or external system) has verified, permissible access to the data.
- **External Sharing:** What data is being sent to other sources, such as external companies, APIs, or third-party processors? This data flow introduces external trust boundaries that require contractual and technical security assurances.

14.2 Analyzing Data Flow

Data flow analysis focuses on the movement and transformation of information between components, ensuring that security boundaries are respected during transfer. Key considerations include:

- **Inter-System Flow:** How does the data flow between interconnected systems and services? Every transfer point is a potential exposure risk.
- **Inputs/Outputs Integrity:** What data enters and exits the system, and are the inputs and outputs validated and filtered against expected formats to prevent injection or corruption?
- **Transport Security:** Is the data transferred in **plain text** or **cipher text**? All data crossing trust boundaries, especially network boundaries, should use strong transport security.
- **Encryption Methods:** If cipher text is utilized, what specific encryption methods are being used (e.g., TLS version, cipher suites, key management practices), and are they compliant with current best practices?

15 Security Assessment and Prevention

15.1 Performing Security Assessments

Secure Design must be validated through formal assessments that check the theoretical design against the implemented reality. Continuous assessment is vital for maintaining security posture:

- **Assessment Status:** Has a security assessment, such as a penetration test or architectural review, been formally performed on the application or system?
- **Assessment Recency:** When was the last assessment conducted? Designs evolve, and regular re-assessment is crucial.
- **Findings and Remediation:** What were the key findings of the assessment? Tracking remediation of design flaws found during these tests is paramount.
- **Methodology:** How was the assessment performed? Did it utilize standardized, reproducible methodologies?

Organizations can measure and improve their overall capacity for secure software practices using the **OWASP Software Assurance Maturity Model (SAMM)**. SAMM provides a flexible framework to help organizations formulate and implement a strategy tailored to their specific risks and resources. Details can be found at https://owaspsamm.org.

15.2 Preventing Insecure Design

Preventing insecure design requires institutionalizing security within the development lifecycle through specific roles, tooling, and documentation requirements:

1. **Early AppSec Integration:** Application Security (AppSec) professionals must be mandatory participants in the design phase. They should initiate discussions about security requirements and threat landscapes early in the design progress.
2. **Secure Design Library:** Organizations should build and curate a library of established **Secure Design patterns** and reference examples of secure code. This minimizes the risk of individual teams inventing insecure solutions for common problems.
3. **Systemic Threat Modeling:** Use Threat Modeling not only for general awareness but specifically for critical systems. This includes diagramming and enforcing all aspects of the authentication, access control, and authorization patterns utilized.

4. **Data Flow Documentation:** Thoroughly diagramming and documenting the entire data flow between all systems is essential. This documentation serves as the foundational artifact for all subsequent risk analysis, security audits, and compliance checks.

16 Application Security Validation and Defense

To ensure application resilience against common threats—such as **SQL Injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)**—development teams must formalize security testing early in the development lifecycle. This involves creating detailed user stories and functional requirements specifically designed to test and validate that the system cannot be exploited by known attack vectors. Furthermore, thorough testing of the system design itself is crucial to uncover architectural flaws before deployment. Finally, external validation through a professional **Penetration Tester** (PenTester) is essential, allowing an independent security expert to actively attempt to compromise the application and servers within the system to discover real-world weaknesses.

17 A05: Security Misconfiguration

The fifth item on the OWASP Top 10 list addresses **Security Misconfiguration (A05)**. This category encompasses flaws arising when systems, applications, or software components are not set up or maintained correctly according to secure operational standards. Misconfiguration is one of the most common vulnerabilities because it is easy to overlook simple settings that can have profound security implications. Attackers frequently seek out unpatched flaws—vulnerabilities that have already been publicly disclosed but remain unfixed. They also target organizational shortcomings, such as unprotected, accessible files and folders, and frequently exploit default configurations, which often leave servers unnecessarily exposed and vulnerable to exploitation.

17.1 Manifestations of Security Misconfiguration

Security misconfigurations manifest in numerous ways across the system architecture:

- **Lack of OS/System Hardening**: Failure to apply necessary configuration controls to reduce the attack surface of the operating system or server software.
- **Missing Patches**: Operating systems and applications are missing the latest security updates or patches, often leaving holes open that are simple to exploit.
- **Excessive Services/Features**: Services or features are running that are unnecessary for the application's function and should have been removed or disabled.
- **Open Ports or Protocols**: Network ports are left open or non-essential network protocols are enabled, creating unnecessary exposure points.
- **Insecure Error Handling**: Error messages expose sensitive data, such as showing a full stack trace, which provides attackers with valuable internal security information about the system structure and technology versions.
- **Excessive Default Access**: Relying on default settings which frequently grant too much access or privilege to users or services, violating the principle of least privilege.

17.2 Root Causes of Misconfiguration

This pervasive vulnerability arises due to a combination of human error and process failure. Frequently, system administrators deploy and configure systems without full knowledge of the secure default values or the optimal secure state of the software. Without rigorous training or standardized deployment scripts, secure settings are often skipped. Furthermore, organizations often fail to implement robust, automated tools for patch management, leading to critical systems being overlooked or missing required security updates. A common and dangerous pitfall is the reliance on the insecure default configurations provided by vendors, rather than implementing a customized, hardened baseline before deployment.

17.3 Preventing Security Misconfiguration

Preventing security misconfiguration requires a rigorous and proactive approach to system administration and deployment:

1. **System Hardening**: Collaborate closely with network and system administrators to "harden" the server, a process often described as "locking it down." This process must include removing any default accounts (or immediately changing their default credentials), disabling all unused services, and turning off any unused network ports.
2. **Least Privilege**: Implement the principle of **least privilege** across the entire system. This critical security policy ensures that every user, process, and application component has only the minimum permissions necessary to perform its required task.
3. **Standardized Secure Builds**: Create a secure, hardened baseline version of each operating system image or server template that can be used when building new systems. This prevents new systems from inheriting insecure default configurations.
4. **Secure Error Handling**: Implement default error handling routines that prevent sensitive information leakage. These routines should provide minimal, generic information to the end user (e.g., a generic 404 page) while ensuring detailed error information, including stack traces, is securely logged on the server.
5. **Monitoring and Review**: Systematically log all server errors and application errors. Critically, establish a dedicated procedure to review these logs regularly, as they often reveal attempted attacks, configuration issues, or signs of compromise.
6. **Security Testing**: Utilize both manual security testing and automated configuration auditing tools to verify adherence to the security baseline prior to system launch.

18 A06: Vulnerable and Outdated Components

Modern software development relies heavily on the integration of existing code libraries, modules, and frameworks, known as **third-party components**, to rapidly provide features and functionality. The sixth major threat identified by OWASP is **Vulnerable and Outdated Components (A06)**. A single vulnerability discovered within one of these external components can potentially compromise the security of the entire application that uses it.

18.1 Assessing Application Vulnerability

An application is highly susceptible to this risk if it meets any of the following criteria:

- It incorporates any third-party components, frameworks, or libraries, especially older or unsupported versions.
- The development team does not perform regular, automated scanning of dependencies for known security vulnerabilities.
- The application relies on frameworks or tools that are multiple versions behind the current stable release, thereby missing crucial security patches. If an application is several versions behind, the probability of it being vulnerable is exceptionally high.

18.2 Causes of Outdated Components

The prevalence of A06 issues stems primarily from the decentralized nature of software maintenance. A significant portion of third-party software is **Open Source**, often hosted on platforms like GitHub or GitLab. The responsibility for updating the software, scanning for vulnerabilities, fixing flaws, and deploying new versions rests entirely with the original author or volunteer community. If an Open Source project does not generate revenue or is maintained by a small group, addressing security vulnerabilities might not be a high-priority task, leading to long delays in patches. Conversely, organizations must hold commercial (paid for) software providers accountable for timely patching and security releases; failure to provide regular updates should prompt the organization to seek new software solutions.

18.3 Tools for Component Vulnerability Management

Mitigating the risk associated with vulnerable components requires constant vigilance using specialized tools designed to scan dependencies against known flaw databases. Key resources and tools include:

- **National Vulnerability Database (NVD)**: A U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP). (https://nvd.nist.gov)
- **Code Repository Scanners**: Modern code management platforms like GitHub and GitLab often incorporate built-in security scanners capable of alerting developers when known Common Vulnerabilities and Exposures (CVEs) exist in the project's dependency list.
- **OWASP Dependency Checker**: An open-source utility that scans project dependencies and checks whether they contain any known, publicly disclosed vulnerabilities. (https://owasp.org/www-project-dependency-check/)
- **Snyk.io**: A commercial and community-driven tool specializing in finding and fixing vulnerabilities in dependencies and container images across the development lifecycle. (https://snyk.io)

19 Managing Software Dependencies and Supply Chain Security

The security of modern applications is heavily dependent on the integrity of the external libraries and packages they utilize. The **Node Package Manager (NPM)** is the default package manager for the JavaScript runtime environment Node.js, and it provides essential tools for managing the hundreds, sometimes thousands, of dependencies an application might rely on.

Effective dependency management requires regular use of specific commands to maintain a secure and stable codebase:

- `npm ls [package name]`: This command allows developers to list and inspect the installed dependency tree, aiding in the discovery of redundant or unauthorized packages.
- `npm audit [fix]`: This is a critical command used to scan the project dependencies for known vulnerabilities. When run with the `[fix]` argument, NPM attempts to automatically update or patch vulnerable packages to a secure version.
- `npm update`: Ensures that packages are brought up to their latest compatible versions, reducing the window of opportunity for attackers relying on known, older exploits.

19.1 Proactive Strategies for Preventing Dependency Attacks

A vulnerability in a single external component can compromise an entire application. To mitigate this pervasive risk, organizations must integrate rigorous security practices directly into their development and deployment workflows.

1. **Continuous Patch Management:** Security patching must be viewed as a mandatory, normal process within the software development lifecycle, rather than an occasional, reactive task.

2. **Dependency Hygiene:** Regularly review and **remove unused dependencies**. While challenging in large, complex projects, minimizing the dependency footprint drastically reduces the attack surface.
3. **Automated Scanning:** Integrate vulnerability scanners directly into the **Continuous Integration/Continuous Deployment (CI/CD)** pipelines. This ensures that new vulnerabilities are detected and flagged before code enters production.
4. **Source Verification:** Only procure and use software components and libraries from **official sites or trusted sources**. Using components from unverified repositories introduces the risk of cryptomining malware or backdoors being included in the code supply chain.
5. **Local Repository Management:** Development teams should utilize a centralized, local or private repository (such as Nexus or Artifactory) to pull updates. This repository must be updated and curated regularly, providing a trusted source that insulates development from potential transient risks in public package registries.
6. **Scheduled Updates and Version Control:** Implement scheduled releases for updates (e.g., quarterly, monthly, or weekly). Critically, frameworks used within the application should never be allowed to fall behind by more than **one or two major versions**, as legacy versions often contain unpatched, publicly known flaws.

20 A07: Identification and Authentication Failures

20.1 Understanding Authentication Failures

The OWASP Top 10 list designated this category as **A07: Identification and Authentication Failures**. Formerly known as "Broken Authentication" (A02 in 2017), the vulnerability has dropped in rank due to industry improvements in secure login implementation and better session management practices. However, this failure remains a critical risk.

Definition

Identification and Authentication Failures relate to defects in how an application verifies a user's identity, manages sessions, and handles credential storage. When implemented incorrectly, these flaws allow attackers to exploit the system, assume the identity of other users, or gain unauthorized access to protected pages or systems.

Applications must rigorously enforce security standards across all steps of the authentication process. Failure points often stem from lax policies regarding automated attacks, weak credential standards, or flawed recovery mechanisms.

20.2 Common Vulnerabilities in Application Authentication

An application is immediately vulnerable if it:

- **Permits Automated Attacks:** Systems that lack lockout mechanisms, CAPTCHAs, or rate limiting can easily be exploited by automated attacks such as **credential stuffing** (where stolen username/password pairs are rapidly tested) or high-volume **brute force attacks**.
- **Allows Weak or Default Passwords:** Accepting credentials such as "Password1" or relying on default pairings like "admin/admin" makes unauthorized access trivial.
- **Uses Weak Recovery Processes:** Relying on simplistic knowledge-based answers (e.g., "What was your mother's maiden name?") for password recovery is highly insecure, as this information is often easily obtained through social engineering or public records.

20.3 Insecure Login Management and Validation

The methods used to validate and manage a user's session are paramount to security:

- **Unsalted Passwords:** As highlighted in A02: 2021 Cryptographic Failures, storing passwords using plain text, simple encryption, or hashing without proper salting is a severe security flaw.
- **Lack of Multi-factor Authentication (MFA):** Failing to implement MFA significantly increases the risk of successful account takeover, especially in environments where phishing and credential stuffing are common.
- **Invalid Session Management:** Pages that fail to validate the user session or do not check if the authenticated user has appropriate permissions to access a specific resource are susceptible to authorization bypass attacks.

20.4 Specific Attack Vectors

20.4.1 Credential Stuffing

Credential stuffing occurs when an attacker uses large lists of account credentials stolen from external breaches (often aggregated by services like "Have I Been Pwned," which tracks billions of compromised accounts from entities such as Adobe, Equifax, Yahoo, and Marriott) and systematically attempts to use them to log into a targeted application. This attack relies on the widespread practice of users reusing passwords across multiple sites.

20.4.2 Brute Force Attacks and Weak Passwords

A brute force attack involves systematically trying common or dictionary passwords against an account. Resources such as public SecLists (which compile the top 10,000 common passwords) provide attackers with highly effective tools. Systems that permit simple passwords are easy targets. Tools like **HASHCAT** are frequently used by attackers to crack large volumes of hashed passwords efficiently.

20.4.3 Weak Forgot-Password Processes

A password recovery mechanism is critically flawed if it exposes the underlying security posture or allows for user enumeration:

- **Returning Plain Text Passwords:** If a system emails the user their password, it means the application is storing the password in a readable format (plain text or reversible encryption), meaning it is **NOT HASHED**. This is a critical security failure.
- **User Enumeration:** When a password reset attempt provides a message stating, "That email/username does not exist," it confirms which accounts are valid versus which are not. This is bad because it allows an attacker to easily compile a list of valid user accounts for targeted attacks, rather than having to guess both the username and password simultaneously.

20.5 Principles of Secure Password Storage

The storage of credentials must adhere to modern cryptographic standards:

- **Plain text passwords ARE BAD.** This is a fundamental concept of security hygiene.
- **Encrypted passwords are ALSO BAD.** While better than plain text, any encrypted value can eventually be decrypted if the key or algorithm is compromised, providing the attacker with the original credential.
- **Mandatory Hashing and Salting:** Applications must **ALWAYS Hash** passwords using a robust, slow hashing algorithm (like bcrypt or Argon2). Crucially, this hash must be combined with a **RANDOM Salt** value that is unique for each user. The salt ensures that identical passwords result in different hashes, defending against pre-computed rainbow tables and general dictionary attacks.

21 Remediating Broken Authentication Vulnerabilities

Broken authentication, often ranking highly on the OWASP Top Ten list, encompasses a wide array of implementation flaws that allow attackers to compromise credentials, sessions, or even impersonate other users. Fixing these vulnerabilities requires a layered approach focusing on password policy, storage mechanisms, recovery procedures, and post-login session validation.

21.1 Strategies for Strong Credential Handling

The first line of defense involves managing credentials securely and preventing automated attacks like brute force and credential stuffing.

Definition

Credential Stuffing is an automated attack where large lists of leaked username/password pairs (obtained from third-party breaches) are systematically injected into an application's login page.

To counter these attacks, applications must maintain strict monitoring over failed login attempts. Systems should track these attempts and implement automatic lockouts for specific accounts or IP addresses exceeding a set threshold. Crucially, alerts should be generated to notify system administrators or the legitimate user of suspicious activity. Services such as https://haveibeenpwned.com/ provide a valuable resource for identifying passwords that have already been compromised in public breaches, allowing developers to preemptively block users from setting weak or known-bad passwords.

21.1.1 Implementing Robust Password Requirements

Weak passwords remain a primary entry vector for attackers utilizing brute force methods. Effective password policy should be enforced not only during account creation but also when users attempt to change their credentials.

- **Complex Rules:** Passwords must adhere to minimum length requirements, typically eight characters or more, though security experts recommend significantly longer passphrases (e.g., 15+ characters) for robust protection. Furthermore, passwords should contain a mix of uppercase and lowercase letters, numbers, and special characters (!@#$%ˆ&).
- **Passphrase Prioritization:** Developers should not impose strict, small limits on password length. Long passphrases, such as mnemonic sentences (e.g., ”Hello my name is Jeff Maxwell and I like to Teach #1 3 5”), are significantly harder to crack than highly complex but short patterns (e.g., P@$$w0rd123). Length contributes exponentially more to entropy than character complexity alone.
- **Prohibiting Bad Passwords:** The application should check any newly set or updated password against a continuously updated list of known compromised or commonly used ”bad” passwords, preventing users from selecting credentials easily found in dictionaries or leaked databases.

Beyond complexity rules, defenses against automated attacks often rely on technologies like **CAPTCHA** (Completely Automated Public Turing test to tell Computers and Humans Apart), which require the user to perform tasks that are easy for humans but difficult for bots, such as transcribing distorted text or identifying images.

21.2 Secure Password Storage

A foundational principle of authentication is that the application must never store passwords in plain text or in a reversibly encrypted format.

Data Integrity

You should **ALWAYS** Hash passwords with a **RANDOM** Salt value for each user.

When a user provides a password, the system must use a strong, one-way cryptographic hashing function (like Argon2 or bcrypt) to derive a non-reversible string of characters (the hash). To prevent attackers from using precomputed lookup tables (rainbow tables) to match hashes back to passwords, a unique, random string of characters, known as a **SALT**, must be concatenated with the password before hashing. Because the salt is unique for every user, even if two users choose the exact same password, they will generate two completely different hashes, securing the system against large-scale dictionary attacks.

21.3 Hardening the Forgot-Password Process

Password recovery is a frequent target for attackers attempting to exploit logical flaws or gain information about valid user accounts. A weak recovery process can undermine an otherwise strong authentication system.

21.3.1 Preventing User Enumeration

When a user attempts to recover an account, the application must avoid providing feedback that indicates whether a given username or email address exists in the system. If the application confirms that an account exists (e.g., "Reset instructions sent to your email") or denies existence (e.g., "User not found"), an attacker can quickly enumerate all valid users in the system before launching a targeted attack.

To prevent this **user enumeration** vulnerability, the system must always respond with a neutral message, such as: “If an email/username exists, password reset instructions will be emailed.”

21.3.2 Implementing Secure Reset Mechanisms

A secure password recovery process relies on a verification mechanism sent out-of-band (e.g., via email or text message) that proves the user controls the account’s registered contact information.

- **Token-Based Reset:** The application should send an email containing a link (URL) with a time-limited, random, and cryptographically secure reset token unique to that recovery request and associated user. When the user clicks the link, the token is verified, and only then is the user permitted to set a new password.
- **MFA Code Requirement:** For heightened security, the system can require the user to type in a one-time code sent via a secondary channel, such as a text message (SMS), before allowing the password reset to proceed.

21.4 Implementing Multi-Factor Authentication (MFA)

Perhaps the most significant defense against credential compromise is the implementation of Multi-Factor Authentication (MFA). MFA requires users to provide two or more verification factors to gain access, making unauthorized access exceedingly difficult even if the password (the first factor) is stolen.

MFA Factors

MFA requires verification across multiple categories:

- **Knowledge Factor:** Something the user knows (e.g., password).
- **Possession Factor:** Something the user has (e.g., physical token, RSA SecurID, mobile device receiving a text).
- **Inherence Factor:** Something the user is (e.g., fingerprint, facial scan).

Common MFA implementations include codes sent via email or text, QR code scans via authenticator applications, or hardware tokens like the RSA SecurID device, which generates a continually changing one-time password (OTP).

22 Session Management

Once a user has successfully authenticated, the security onus shifts to **Session Management**. The system must ensure that the user remains authenticated securely and, critically, that for every subsequent page or resource access, the user is authorized to view that specific resource.

Inadequate session management often leads to authorization flaws, particularly **Insecure Direct Object Reference (IDOR)**, where an attacker can modify a parameter (like a user ID in a URL) to access data belonging to another user.

22.0.1 The Panera Bread Case Study

A notable real-world example of poor session and authorization management involved Panera Bread. In 2018, the company was notified that their online ordering system was not properly validating users when they navigated between pages. The vulnerability allowed unauthorized access to millions of customer records, including names, email addresses, physical addresses, and partial credit card numbers.

The flaw manifested as an IDOR vulnerability: a resource URL might look like `https://panerabread.com/users/1234`. The system failed to check if the session associated with the currently logged-in user was actually authorized to access the data tied to user ID 1234. An attacker could simply increment the user ID parameter in the URL (`/users/1235`, `/users/1236`, etc.) and gain access to arbitrary accounts, as the web application retrieved the data based purely on the URL parameter without validating the user's permissions against the requested resource.

Despite the initial responsible disclosure to Panera Bread, the issue remained unpatched for months. This delay eventually led security researcher Brian Krebs to publish a blog post detailing the severe data leak, which garnered significant media attention and forced the company to take immediate corrective action. This case highlights the necessity of implementing strict, server-side access control checks on every request involving sensitive data.

23 Addressing Integrity Failures and Monitoring Deficiencies

The assessment of application security necessitates a continuous review of weaknesses, which are frequently updated and refined based on emerging threats. The most recent revisions emphasize the often-overlooked necessity of verifying the legitimacy of code and data, coupled with rigorous monitoring and logging practices.

23.1 A08: Software and Data Integrity Failures

The category of **Software and Data Integrity Failures** (A08) was introduced in 2021 to specifically address risks arising from an application making dangerous assumptions about the state or origin of its components without sufficient verification. This type of failure occurs when a system relies on external sources or automated processes that could be compromised, leading to unauthorized manipulation of application data or logic.

The Panera Bread security incident, highlighted in reports by KrebsOnSecurity, serves as a high-profile example where a lack of system integrity allowed extensive personal customer data to remain publicly exposed through an API for eight months, underscoring the severe consequences of failure in this domain.

23.2 Mechanisms for Integrity Violation

Integrity violations stem from both improper code implementation and weak infrastructure controls. Key areas of vulnerability include:

1. **Infrastructure Weakness:** Utilizing cloud services, Content Delivery Networks (CDNs), or open-source repositories without validating the security of the content delivered.

2. **Deployment Pipelines:** Insecure **CI/CD pipelines** (Continuous Integration/Continuous Deployment) provide a significant risk vector. If the pipeline itself is compromised, an attacker can inject malicious code directly into the production environment without triggering traditional security alerts.
3. **Auto-Update Functionality:** Many modern applications feature auto-update mechanisms. If these updates are downloaded and installed without sufficient cryptographic **integrity verification**, an attacker can distribute trojanized versions of the software under the guise of an official patch.

23.3 Mitigating Integrity Failures

Preventing integrity failures requires a layered approach focusing on source validation, dependency management, and continuous verification of deployed assets.

23.3.1 Source Validation and Dependency Control

Updates and dependencies must be rigorously authenticated. The implementation of **Digital Signatures** is fundamental, providing cryptographic proof to validate both the creator and the integrity of the update package. Furthermore, organizations should exercise strict control over third-party components by managing internal mirrored libraries (such as internal **NPM** or **NuGet** repositories). This practice ensures that all components, even open-source dependencies, are vetted for known vulnerabilities before being made available to development teams.

23.3.2 Continuous Integrity Checking

Automated checks must be integrated into the operational environment to verify the deployed code base. Specifically, after each deployment, a cryptographic **HASH** should be calculated on all critical files and folders. This hash must then be compared against a trusted, expected hash value stored securely. This verification process should be run routinely, ideally at least once per day, to rapidly detect unauthorized changes.

23.3.3 Secure Development Practices

Preventative measures must also be baked into the development lifecycle:

- Routine use of automated **Security and Code Scans** (SAST/DAST) helps identify code-level vulnerabilities before deployment.
- Implement mandatory **Manual Code Reviews** of all changes, particularly those touching sensitive logic or external integrations.
- Secure the CI/CD environment by ensuring that sensitive credentials (passwords, tokens, API keys) are never stored in plain text or within repository configurations, and that access to the pipeline execution environment is strictly limited via robust access controls.

23.4 A09: Security Monitoring and Logging Failures

The category of **Security Monitoring and Logging Failures** (A09) addresses the failure of an application to adequately track operations, allowing breaches to go undetected or making effective forensic analysis impossible. Previously ranked as "Insufficient Logging and Monitoring," its elevation in prominence to position #9 reflects the necessity of robust observability for timely detection and incident response. This failure category mainly deals with inadequate tracking of application health, traffic flow, and security-relevant events.

23.5 Logging Best Practices

A secure logging strategy must prioritize comprehensiveness, isolation, and security.

23.5.1 Data Capture and Storage

The system must be configured to capture **all traffic and activity** directed at the application or service. Given the affordability of storage, applications should log as much relevant data as possible. However, this data must be secured through isolation: logs must be immediately offloaded to a separate, centralized logging server or dedicated Security Information and Event Management (SIEM) platform. A fundamental security principle is to **never leave forensic logs on the same server** hosting the application. If the server is compromised, the attacker will almost certainly delete or alter the local logs to conceal their actions.

23.5.2 Operational Logging Strategy

Effective logging requires careful planning and definition of scope:

- What events is the application logging by default (often insufficient)?
- What specific security and transaction events should be logged?
- What highly sensitive data should explicitly be excluded from all logs?
- Are logs stored remotely and securely, or are they accessible locally?
- Is the system logging errors, exceptions, and warnings consistently?

23.6 Sensitive Data Exclusion

A critical aspect of secure logging is data minimization. Logging sensitive information creates unnecessary exposure and introduces regulatory risk. Under no circumstances should the following items be recorded in application logs:

- Any **Personally Identifiable Information (PII)**.
- Financial details, including **Credit Cards** or **Bank Numbers**.
- Authentication artifacts such as **Passwords**, security keys, or temporary access **Tokens**.

Definition

Logging sensitive secrets (passwords, tokens) and PII constitutes a severe security flaw. If an attacker gains access to logs containing this information, a local system intrusion instantly escalates into a catastrophic data breach, incurring high penalties and violating privacy regulations.

23.7 Categorization of Log Data

Application logging can be broadly categorized based on the purpose of the data being recorded:

- **Exception Logs:** Generated when the application encounters a critical runtime issue, such as code crashes or unexpected failures. These are essential for debugging and maintaining application stability.
- **Audit Logs:** Detailed records tracking every significant business event, user action, page view, or transaction. These logs provide accountability and a complete historical trace of system workflows.
- **Usage Logs:** Metrics focused on application adoption and performance, recording frequency of feature use or page views. These are typically used for capacity planning and business intelligence rather than security.
- **Security Logs:** The cornerstone of incident response, tracking events specifically related to the security posture, such as login success/failure attempts, authorization denials, and access control violations.

24 Effective Logging and Monitoring Strategy

Effective application security relies fundamentally on a robust system of logging and monitoring. If security-critical events are not recorded, detecting and investigating breaches becomes nearly impossible.

24.1 Fundamental Logging Rules

A comprehensive logging strategy must capture both normal operational flow and exceptional events. The core tenets of secure logging include:

1. **Logging Normal Traffic:** It is crucial to log all successful operations and pathways within the application. This establishes a reliable baseline of expected behavior. Without knowing what 'normal' looks like, detecting anomalies or malicious activity is extremely difficult.
2. **Logging Errors and Warnings:** All system errors, application warnings, and exceptions must be logged immediately. These entries often highlight instability, unauthorized attempts, or conditions that could be exploited by attackers.
3. **Specific Security Logging:** Security-sensitive operations require detailed logging. Specifically, this includes logging all login attempts, distinguishing between **Successful Logins** and **Failed Logins**. Tracking failed attempts is critical for identifying potential brute-force or credential stuffing attacks. Furthermore, any instance of **Abnormal Traffic** that falls outside established operational parameters must be flagged and logged.

To effectively handle the sheer volume of data generated by these rules, organizations must utilize dedicated log management solutions. These tools ingest, index, and analyze data from various sources. A standard industry tool used for this purpose is **Splunk**, which provides powerful search, analysis, and visualization capabilities necessary for security operations.

24.2 Monitoring for Security and System Health

Simply logging events is insufficient; logging is **WORTHLESS** unless the logs are systematically reviewed and acted upon. Monitoring transforms raw log data into actionable intelligence.

The Criticality of Review

Security teams must establish daily routines to review system errors and critical logs. Any identified errors or unexpected events should be reported immediately to development teams to ensure vulnerabilities or bugs are patched before they can be exploited.

Effective monitoring requires the establishment of an automated system designed to alert personnel when certain conditions are met. These conditions typically involve defining **Thresholds** based on observed traffic patterns. For instance, an alert might trigger if there are twenty failed login attempts from a single IP address in one minute, or if unusually high volumes of database read requests occur outside of business hours. This alerting capability is essential for minimizing the window of opportunity for an attacker.

24.3 Prevention and Integrity Controls

Preventative measures ensure that logging itself contributes directly to security and that log data cannot be compromised.

1. **Logging Failure Attempts:** All failures related to login, access control checks, and server-side input validation must be logged. Critically, these failures should be flagged immediately as potential hack attempts, providing early warning of active reconnaissance or attack.
2. **Log Standardization and Encoding:** Logs must be generated in a consistent format that is easily consumed, parsed, and indexed by logging tools. Furthermore, logs must be correctly **Encoded** (e.g., using proper sanitization or encoding on user-supplied data that ends up in the log entry) to ensure

that log viewing systems are not vulnerable to attacks like cross-site scripting (XSS) or SQL Injection (if the logs are stored in a database).

3. **Integrity Controls for High-Value Transactions:** For high-value or sensitive transactions, a detailed audit trail is required. This trail must possess **Integrity Controls** to prevent tampering or undetected modification. Techniques such as using **Append-Only Database Tables** or utilizing cryptographic controls like **Blockchain** technology ensure that records, once written, cannot be altered, preserving the forensic value of the audit log.
4. **Visibility and Response Planning:** Alerting for suspicious activity should be robustly established, and key performance indicators (KPIs) and operational dashboards must be visible to the security and operations teams. Finally, the organization must maintain a comprehensive **Incident Response and Recovery Plan**, along with dedicated teams trained to respond and react swiftly if a security breach occurs.

25 Vulnerability Deep Dive: Server-Side Request Forgery (A10)

Server-Side Request Forgery (SSRF) is a significant web application vulnerability that has demonstrated persistent relevance in modern application architectures. It was previously featured on the OWASP Top 10 list, dropped off in 2017, but was reinstated in the 2021 list, reflecting its increased severity due to complex cloud environments and API interactions.

25.1 Understanding SSRF

Definition: Server-Side Request Forgery (SSRF)

SSRF occurs when a web application attempts to fetch a remote resource based on a URL provided or modified by a user, but fails to properly validate the user-supplied URL prior to processing the request.

The danger of SSRF lies in the fact that the application is tricked into making a request on behalf of the attacker. This request originates from the trusted server environment, allowing attackers to send specially crafted requests to achieve two main outcomes:

1. They can force the server to return unexpected results by fetching resources the application should not access.
2. More critically, they can compel the server to interact with internal, non-public resources, such as accessing internal APIs, manipulating the server's filesystem using file/dict schemes, or interacting with cloud service metadata endpoints (e.g., AWS EC2 metadata endpoints) to steal credentials.

25.2 Mitigating SSRF

The fundamental principle for preventing SSRF, as with many other injection attacks, is rigorous input validation.

Consider an example of user-supplied input used by the application:

https://site.com/users/1234

If an application fetches internal data based on the numerical ID provided in the URL without validating or authorizing the user's rights to access that specific ID, the attacker could easily change the number (e.g., to 5678) and access different user accounts. This type of failure to validate an identifier against the authorized user (Insecure Direct Object Reference or IDOR) is closely related to the underlying issues that facilitate SSRF.

To prevent SSRF:

1. **Need to Validate Input:** This is the single most critical defense. The application must enforce strict validation rules on any user-supplied URL.
2. **Whitelisting Approach:** Preferably, use a **Whitelisting** approach for allowed domain names or IP addresses that the application is permitted to connect to.
3. **Blocklisting Private Networks:** If whitelisting is not feasible, strong **Blocklisting** must be implemented to prevent connections to private IP ranges (e.g., 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, and localhost 127.0.0.1/8) and internal network schemes (like file:///, dict://, gopher://).
4. **URL Canonicalization:** Ensure that input validation is performed after the provided URL has been completely canonicalized (resolved) to prevent attackers from using redirects or non-standard notation to bypass filters.