

Unit 3- Application Security within Cyber Security

Introduction:

- landscape of web security is constantly evolving.

- **Malicious Websites:** Research has indicated that around 1 in 10 websites may be infected with some form of malware or malicious code.
- **Phishing Sites:** Approximately 30,000 to 50,000 phishing sites are created daily, according to various security reports.
- Estimates suggest that over 70% of infected websites are used to distribute malware.
- notable increase in the number of attacks targeting web applications.

- Common Threats

- Security Principles:

- **Least Privilege:** Users and systems should have the minimum level of access necessary to perform their functions.
- **Defense in Depth:** Use multiple layers of security controls to protect data and assets.
- **Input Validation:** Always validate and sanitize user inputs to prevent injection attacks.

- Best Practices:

- **Use HTTPS:** Encrypt data in transit to protect against eavesdropping.
- **Regular Updates:** Keep software, frameworks, and libraries up to date to mitigate vulnerabilities.
- **Security Testing:** Conduct regular security assessments, including penetration testing and vulnerability scanning.

Defense in Depth.

- **Network Security**

- **Firewalls:** Use web application firewalls (WAFs) to filter and monitor HTTP traffic, blocking malicious requests before they reach the application.
- **Intrusion Detection/Prevention Systems (IDS/IPS):** Implement systems to detect and respond to suspicious activities in real time.

- **Secure Application Development**

- **Secure Coding Practices:** Train developers in secure coding techniques to minimize vulnerabilities such as SQL injection and XSS.
- **Code Reviews and Static Analysis:** Conduct regular code reviews and use static analysis tools to identify and fix security issues during development.

- **Input Validation and Output Encoding**

- **Input Validation:** Sanitize and validate user inputs to prevent injection attacks.
- **Output Encoding:** Encode outputs to prevent XSS by ensuring that user inputs are safely rendered in the browser.

- **Authentication and Access Control**

- **Strong Authentication:** Implement multi-factor authentication (MFA) to enhance user verification.
- **Role-Based Access Control (RBAC):** Enforce least privilege principles by restricting access to sensitive data and functionality based on user roles.

- **Regular Security Assessments**

- **Penetration Testing:** Conduct periodic penetration tests to identify vulnerabilities and weaknesses.
- **Vulnerability Scanning:** Use automated tools to scan applications for known vulnerabilities regularly.

Types of Web Application Security Testing

- **Black Box Testing:** The tester has no prior knowledge of the system architecture or code. This simulates an external attacker's perspective, focusing on finding vulnerabilities through external interfaces.
- **White Box Testing:** The tester has full knowledge of the system, including source code, architecture, and configurations. This allows for thorough testing, examining internal logic, and identifying security flaws.
- **Gray Box Testing:** A hybrid approach where the tester has partial knowledge of the system. This simulates an insider threat or an attacker with limited information, allowing for a more targeted assessment.
- **Network Penetration Testing:** Focuses on identifying vulnerabilities in network infrastructure, such as firewalls, routers, and switches. It assesses how well the network can withstand attacks.
- **Web Application Penetration Testing:** Targets web applications to identify vulnerabilities like SQL injection, cross-site scripting (XSS), and other common web-based attacks.
- **Mobile Application Penetration Testing:** Evaluates mobile apps for security vulnerabilities, including insecure data storage, poor authentication, and flaws in mobile API integration.

Vulnerability Assessment and Penetration Testing (VAPT) involves a series of structured steps to identify and exploit vulnerabilities in systems. Here's a typical process:

1. Planning and Scope Definition

1. Define the objectives of the VAPT.
2. Identify the scope (assets, systems, applications).
3. Obtain necessary permissions and legal clearances.

2. Information Gathering

1. Passive Reconnaissance: Collect information without interacting directly with the target (WHOIS, DNS records, etc.).
2. Active Reconnaissance: Use tools to scan and identify live hosts, open ports, and services.

3. Vulnerability Assessment

Use automated tools (e.g., Nessus, OpenVAS) to scan for known vulnerabilities.

Manually verify findings and identify additional vulnerabilities that tools may miss.

4. Exploitation

Attempt to exploit identified vulnerabilities to gain unauthorized access or control.

Use tools like Metasploit or custom scripts.

Ensure this step is conducted in a controlled manner to avoid damage.

5. Post-Exploitation

Assess the extent of access gained.

Establish persistence (if needed, for testing purposes).

Gather sensitive information to evaluate risk.

6. Reporting

Document all findings, including vulnerabilities, exploits, and risk assessments.

Provide clear recommendations for remediation.

Ensure the report is tailored to the audience (technical vs. non-technical).

7. Remediation and Re-testing

Work with the client to address identified vulnerabilities.

Conduct a follow-up assessment to verify that remediation efforts were effective.

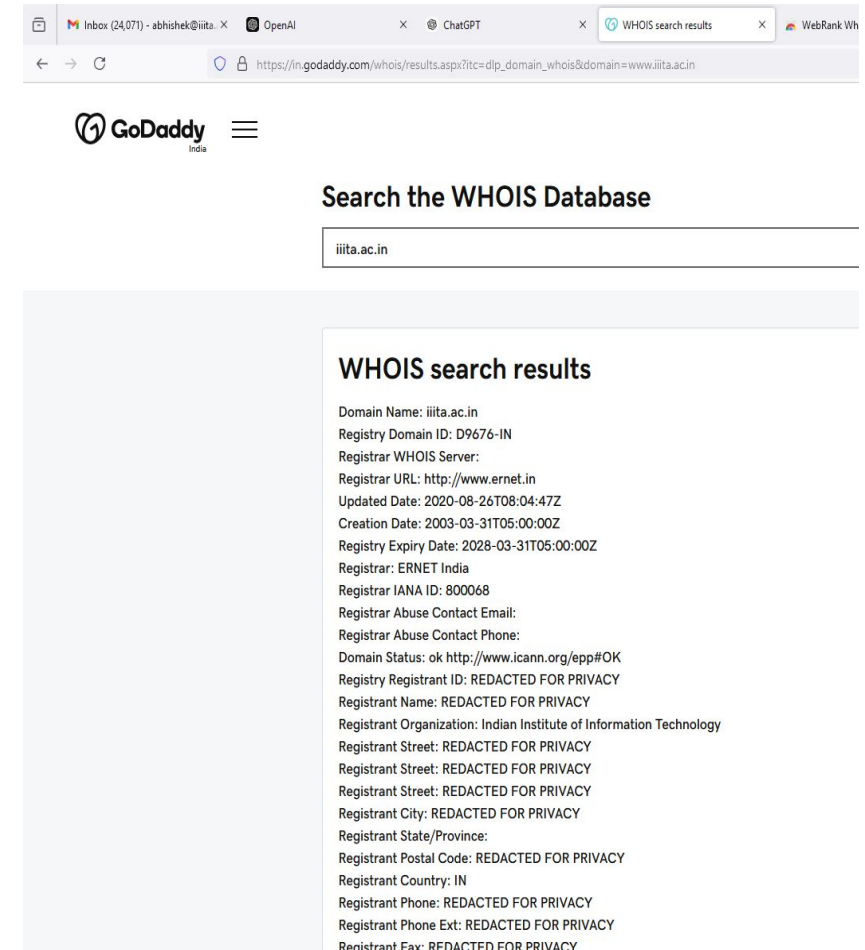
8. Continuous Monitoring

Encourage ongoing security practices and regular assessments to adapt to new threats.

Each step may vary based on the specific requirements and context of the organization being tested.

Passive Reconnaissance

- **WHOIS Lookup:** Retrieve domain registration information to find details about the owner and contact.
- **DNS Enumeration:** Identify DNS records (A, MX, TXT) to gather information about the domain and its services. Tools: dig, nslookup
- **Social Media Profiling:** Collect data from social media to find employee names, roles, and potential vulnerabilities related to human behavior.
- **Publicly Available Information:** Search for documents, reports, or any data available on public repositories, websites, or forums.
- **Google Dorking:** Use advanced search operators in Google to find specific information about the target, such as sensitive files or exposed servers.



The screenshot shows a web browser window with multiple tabs. The active tab is titled "WHOIS search results" and shows the URL https://in.godaddy.com/whois/results.aspx?itc=dlp_domain_whois&domain=www.iiita.ac.in. The GoDaddy logo is visible in the top left. The main heading is "Search the WHOIS Database". Below it, a search bar contains the text "iiita.ac.in". The search results are displayed in a light blue box with the heading "WHOIS search results". The results list the following information:

- Domain Name: iiita.ac.in
- Registry Domain ID: D9676-IN
- Registrar WHOIS Server:
- Registrar URL: <http://www.ernet.in>
- Updated Date: 2020-08-26T08:04:47Z
- Creation Date: 2003-03-31T05:00:00Z
- Registry Expiry Date: 2028-03-31T05:00:00Z
- Registrar: ERNET India
- Registrar IANA ID: 800068
- Registrar Abuse Contact Email:
- Registrar Abuse Contact Phone:
- Domain Status: ok <http://www.icann.org/epp#OK>
- Registry Registrant ID: REDACTED FOR PRIVACY
- Registrant Name: REDACTED FOR PRIVACY
- Registrant Organization: Indian Institute of Information Technology
- Registrant Street: REDACTED FOR PRIVACY
- Registrant Street: REDACTED FOR PRIVACY
- Registrant Street: REDACTED FOR PRIVACY
- Registrant City: REDACTED FOR PRIVACY
- Registrant State/Province:
- Registrant Postal Code: REDACTED FOR PRIVACY
- Registrant Country: IN
- Registrant Phone: REDACTED FOR PRIVACY
- Registrant Phone Ext: REDACTED FOR PRIVACY
- Registrant Fax: REDACTED FOR PRIVACY

Active Reconnaissance

- **Port Scanning:** Use tools like Nmap to identify open ports and services running on the target system.
- **Service Fingerprinting:** Determine the versions of services running on identified ports to assess potential vulnerabilities.
- **Network Mapping:** Create a visual map of the network to understand the structure and identify critical systems.
- **Banner Grabbing:** Retrieve service banners to gather more information about software versions and configurations.
- **Ping Sweeping:** Check the availability of hosts within a network by sending ICMP echo requests.

Starting Nmap (<http://nmap.org>) at 2024-10-22 14:30 UTC

Nmap scan report for example.com (192.168.1.1)

Host is up (0.0020s latency).

Not shown: 999 closed ports

PORT	STATE	SERVICE	VERSION
------	-------	---------	---------

22/tcp	open	ssh	OpenSSH 7.6 (protocol 2.0)
--------	------	-----	----------------------------

80/tcp	open	http	Apache httpd 2.4.41
--------	------	------	---------------------

443/tcp	open	ssl/http	nginx 1.18.0
---------	------	----------	--------------

Other Techniques

- **Web Application Scanning:** Use tools to analyze web applications for vulnerabilities like SQL injection, XSS, etc.
- **Physical Reconnaissance:** If applicable, gather information through physical observation or social engineering (e.g., impersonating employees).
- **API Enumeration:** Identify and analyze any exposed APIs to find vulnerabilities related to data exposure or improper authentication.

Scan Report for <http://example.com>

1. ****Critical Vulnerabilities**:**

- SQL Injection on /login.php
- Description: User input is not properly sanitized.
- Recommended Fix: Use prepared statements to handle SQL queries.

2. ****Medium Vulnerabilities**:**

- Cross-Site Scripting (XSS) on /search.php
- Description: Unsanitized input allows script execution.
- Recommended Fix: Implement input validation and output encoding.

3. ****SSL/TLS Issues**:**

- SSL Certificate Expiration
- Status: Expires in 30 days.
- Recommended Action: Renew SSL certificate.

4. ****Open Ports**:**

- Port 80 (HTTP) - Open
- Port 443 (HTTPS) - Open

5. ****Technologies Detected**:**

- Framework: Django 3.1.1
- Database: PostgreSQL 12.2

Penetration Testing

- **Black Box Testing**

Description: The tester has no prior knowledge of the system architecture or code.

Focus: Simulates an external attacker's perspective to find vulnerabilities without any internal insight.

- **White Box Testing**

Description: The tester has full knowledge of the system, including source code, architecture, and configurations.

Focus: Allows for thorough testing of internal vulnerabilities, often covering more ground than black box testing.

- **Gray Box Testing**

Description: The tester has partial knowledge of the system, often limited to certain areas.

Focus: Combines elements of both black and white box testing, simulating an insider threat or a partially informed attacker.

- **Web Application Penetration Testing**

Description: Focuses specifically on web applications to identify vulnerabilities such as SQL injection, cross-site scripting (XSS), and insecure configurations.

Focus: Assesses the security of web-based interfaces.

Exploit Demonstration - ****Payload Used****: `admin' OR '1'='1` - ****Response****: The application returned a successful login for the admin account. - ****Evidence****: Response contains user information indicating successful SQL injection.

OWASP

These are the most recognized and widely used OWASP resources.
Some flagship projects include:

- **OWASP Top Ten** (e.g., the most critical security risks for web applications)
- **OWASP ZAP** (Zed Attack Proxy)
- **OWASP Dependency-Check**
- **OWASP SAMM** (Software Assurance Maturity Model)

Key Features of OWASP ZAP

1. Automated Scanning

1. ZAP provides an **automated scanner** that can crawl web applications and automatically detect many common security vulnerabilities, such as injection flaws, cross-site scripting (XSS), and insecure HTTP headers.
2. The automated scanner can be configured to scan specific areas of a website, or you can allow it to scan the entire site.

2. Manual Testing Tools

1. ZAP provides a rich set of tools for manual security testing. These tools include the ability to intercept and modify HTTP/HTTPS traffic, manually test requests and responses, and modify them in real-time.
2. The **Request Editor** and **Response Editor** allow for in-depth inspection and manipulation of HTTP requests and responses.

3. Passive Scanning

1. ZAP includes a **passive scanner** that monitors traffic between the browser and the web application without altering it. It looks for security flaws by observing normal application behavior and scanning for known vulnerabilities.
2. The passive scanner does not modify any requests or responses; it only observes, making it safe for use in production environments.

4. Active Scanning

1. ZAP also offers an **active scanner**, which actively sends different payloads (e.g., SQL injection strings, XSS vectors) to test for vulnerabilities.
2. This scanner can be more intrusive than the passive scanner and is typically used in staging or testing environments, as it can cause errors if misused.

Spidering/Crawling

- The **Spider** tool in ZAP automatically crawls a web application to discover its pages and links. This is useful for discovering hidden URLs, parameters, and other resources that might not be immediately obvious.
- Spidering is essential for creating a map of the application and identifying areas that should be scanned.

Session Management and Authentication

- ZAP supports advanced session management and authentication capabilities, allowing you to log in and interact with web applications that require user credentials.
- It also has features for handling different types of authentication, including **form-based authentication**, **HTTP basic authentication**, **cookie-based sessions**, and **OAuth**.

ZAP HUD (Heads-Up Display)

- The **HUD** is a lightweight overlay designed to provide real-time security testing feedback in the browser itself. It is particularly useful for developers who want to get immediate insights into potential security risks while interacting with their web applications.
- It can be integrated into your browser to give feedback as you test, showing possible security issues in real-time.

API Integration

- ZAP provides a robust **REST API** that allows you to automate security testing tasks, integrate ZAP into continuous integration (CI) pipelines, and retrieve scan results programmatically.
- The API can be used for various tasks, such as initiating scans, retrieving reports, and integrating with other testing tools.

Reporting and Results

- ZAP generates comprehensive **reports** detailing vulnerabilities discovered during a scan, including descriptions, severity levels, and suggested mitigation steps.
- Reports can be generated in multiple formats, including HTML, XML, and JSON, making it easy to share results with stakeholders or integrate them into other tools.

OWASP Dependency Check

- **Vulnerability Identification:**
 - Dependency-Check performs deep analysis of a project's dependencies (e.g., libraries, frameworks, modules) to detect known vulnerabilities in them.
 - It checks the versions of the libraries or components in use and compares them with publicly available databases (e.g., NVD, GitHub Advisory Database, and others) to see if any of the dependencies are vulnerable to known issues.
- **Support for Multiple Programming Languages:**
- **Dependency-Check can scan a wide variety of project formats, making it language-agnostic:**
 - **Java:** Works with Maven, Gradle, and other Java-based build systems.
 - **JavaScript/Node.js:** Supports npm and yarn-based projects.
 - **.NET:** Integrates with MSBuild or NuGet to analyze .NET projects.
 - **Python:** Works with Python's requirements.txt and pip-based environments.
 - **Ruby:** Compatible with RubyGems.
 - **Other formats:** It can also support other common dependency management systems via plugins or custom configurations.
- **Automated Scanning:** It automates the process of identifying and alerting on vulnerabilities. Developers can set up Dependency-Check as part of their continuous integration (CI) or continuous delivery (CD) pipelines, ensuring that vulnerabilities are detected as part of the build process.
- **Database Integration:**
 - OWASP Dependency-Check integrates with several well-known vulnerability databases, most notably:
 - **National Vulnerability Database (NVD)** : The NVD contains a vast repository of vulnerabilities and is the primary source of vulnerability data for Dependency-Check.
 - **GitHub Advisory Database** : Dependency-Check can also query GitHub's vulnerability advisory database for known issues in open-source libraries.
- **Plugin Support:**
 - Dependency-Check offers integration plugins for a variety of build tools and environments:
 - **Maven Plugin** : For Maven-based Java projects.
 - **Gradle Plugin** : For Gradle-based Java projects.
 - **Jenkins Plugin** : For Jenkins CI/CD pipeline integration.
 - **Command Line Interface (CLI)** : For running Dependency-Check from the command line.

OWASP

- While not a formal part of the VAPT model, the OWASP Top Ten provides a critical framework for identifying common vulnerabilities in web applications, such as:
 - Injection flaws (SQL, NoSQL, Command Injection)
 - Broken Authentication
 - Sensitive Data Exposure
 - XML External Entities (XXE)
 - Broken Access Control
 - Security Misconfiguration
 - **Cross-Site Scripting (XSS)**
 - Insecure Deserialization
 - Using Components with Known Vulnerabilities
 - Insufficient Logging & Monitoring
- Incorporating the OWASP model into VAPT processes helps ensure a comprehensive, methodical approach to identifying and mitigating security risks in web applications and systems.

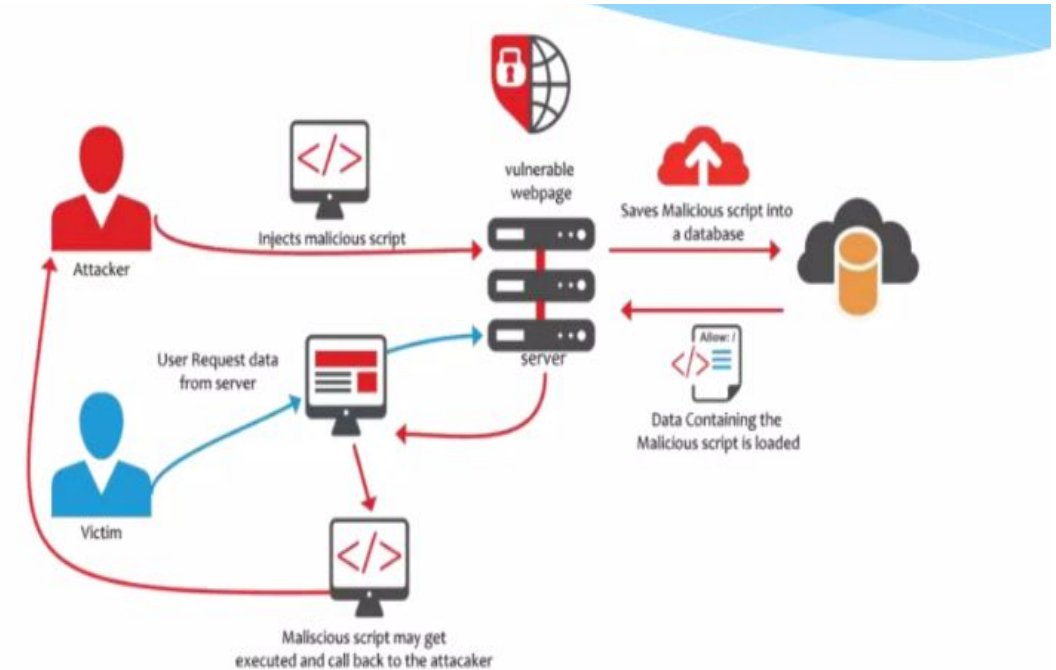
Cross-Site Scripting (XSS)

Description: XSS occurs when an attacker injects malicious scripts into web pages viewed by other users. This can happen through input fields that don't validate or sanitize user input.

- **Types:**

- **Stored XSS:** Malicious code is stored on the server (e.g., in a database) and executed when users access the affected page.
- **Reflected XSS:** Malicious code is reflected off a web server, often via a URL, and executed immediately without being stored.
- **DOM-based XSS:** The vulnerability occurs in the client-side JavaScript rather than the server.

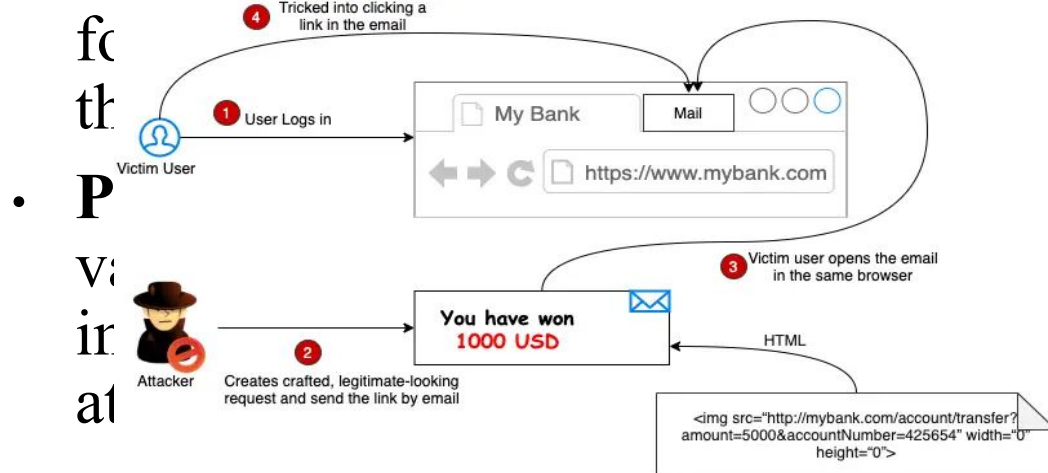
- Eg: `<script>alert('XSS')</script>`
- ``



Cross-Site Request Forgery (CSRF)

- **Description:** CSRF tricks the victim's browser into sending requests to a different site where the user is authenticated, potentially altering their data without consent.

- **Example:** An attacker could create a



```
•<html>
•<head>
•    <title>CSRF Token Demo</title>
•</head>
•<body>
•    <form action="process" method="POST">
•        <input type="hidden" name="_csrf" value="<%= csrfToken %>">
•        <div>
•            <label>Amount:</label><input type="text" name="amount">
•        </div>
•        <br/>
•        <div>
•            <label>Transfer To:</label><input type="text" name="account">
•        </div>
•        <br/>
•        <div>
•            <input type="submit" value="Transfer">
•        </div>
•    </form>
•</body>
•</html>
```

Injection Flaws

An **injection flaw** in a web application occurs when untrusted data is improperly handled and then executed by the system, often leading to unintended actions or security vulnerabilities.

SQL Injection (SQLi)

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

' OR '1'='1

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

Command Injection:

Command injection occurs when a user can execute arbitrary operating system commands on the server through the web application.

Example: If an application uses user input to construct a shell command like:

```
system("ping " + user_input);
```

An attacker could input:

```
; rm -rf /important-data
```


OWASP TESTING CATEGORIES

Information Gathering

- Reconnaissance phase: identifying the structure, functionality, and architecture of the application.
- Tools and techniques include WHOIS lookups, DNS queries, HTTP headers, and so on.

Authentication Testing

- Testing for weak or broken authentication mechanisms.
- Includes testing for weak password policies, login brute-force protections, and account enumeration vulnerabilities.

Session Management Testing

- Verifying that sessions are securely managed, including testing for issues like session fixation, session hijacking, and cookie security.
- Testing for proper session expiration, invalidation, and cookie attributes like HttpOnly, Secure, etc.
 - E.g.: Secure flag on the cookies.
 - SameSite = LAX attribute is set to Strict or Lax on session cookies to prevent cross-site request forgery (CSRF) attacks.
 - HttpOnly flag is set on session cookies, which prevents client-side JavaScript from accessing the cookie, mitigating XSS attacks.
 - predict session IDs using tools like Burp Suite

```
setcookie('sessionid', 'abc123', [  
    'expires' => time() + 3600, // 1 hour from now  
    'path' => '/',  
    'domain' => 'example.com',  
    'secure' => true,    // Ensures the cookie is only sent over HTTPS  
    'httponly' => true, // Makes the cookie inaccessible to JavaScript  
    'samesite' => 'Lax', // Mitigates CSRF  
]);
```

- **Access Control Testing**

Ensuring that proper authorization mechanisms are in place to restrict access to resources based on the user's role or permissions; testing for privilege escalation and broken access control issues.

e.g: admin/dashboard → Try changing to /admin/settings or /admin/user-management

/profile?id=123 → Change id=123 to id=1 (which could be the admin user)

/document/view?docId=456 → Try changing docId=456 to docId=457 and check if you can access another user's document.

/uploads/user123/file.jpg → change to /uploads/user124/file.jpg

Input Validation Testing

- Verifying that the application properly handles user input to prevent injection attacks, cross-site scripting (XSS), and other input-related vulnerabilities. Techniques include testing for SQL injection, command injection, file upload issues, etc.
 - **Form fields** (e.g., login forms, registration forms, search boxes)
 - **Query parameters** in the URL (e.g., /search?query=abc)
 - **HTTP headers** (e.g., User-Agent, X-Forwarded-For, Cookie)
 - **File uploads** (e.g., images, documents)
 - **API endpoints** (e.g., RESTful APIs)
- **Error Handling Testing**
 - Testing how the application handles errors and exceptions, ensuring that sensitive information is not exposed in error messages.

Error message: "Fatal error: Uncaught Exception: SQL syntax error at line 48 in file /var/www/html/index.php"

Expected behavior: The system should return a generic error message, like:

"An unexpected error occurred. Please try again later."

http://example.com/admin/delete-user?id=-1 → Should return 403 or 500 with a generic error message, not a detailed database error.

OWASP TESTING CATEGORIES

- **Cryptography Testing**
 - Assessing the strength and implementation of cryptographic mechanisms, such as SSL/TLS, password hashing, and data encryption.
- **Business Logic Testing**
 - Testing the application to identify vulnerabilities in business logic that could be exploited, even if all technical controls are in place.
 - Example: Bypassing payment systems or exploiting discount codes.
- **Denial of Service (DoS) Testing**
 - Assessing the application's ability to handle high traffic or load without crashing or being degraded in functionality.
- **Client-side Testing**
 - Verifying security in front-end code (JavaScript, HTML) to prevent issues like Cross-Site Scripting (XSS), insecure direct object references (IDOR), and CSRF vulnerabilities.

Buffer overflow

buffer overflow occurs when a program writes more data to a buffer (a temporary storage area in memory) than it can hold. This overflow can overwrite adjacent memory locations, leading to unpredictable behavior, crashes, or security vulnerabilities.

1. Memory Layout:

- In a typical program, memory is organized into different segments: the stack, heap, and data segment.
- Buffers are usually allocated in the stack for local variables.

2. Data Input:

- When a program reads input (e.g., from user input or a file), it typically places this data into a buffer.
- If the input exceeds the allocated size of the buffer, it starts to overwrite adjacent memory.

• 1. Vulnerable Code:

```
void vulnerable_function() {  
    char buffer[10];  
    gets(buffer); // Unsafe function that doesn't check input length  
}
```

1. Overwriting Memory:

- As data exceeds the buffer's capacity, it can overwrite other important memory areas, such as:
 - Return addresses (pointers to where the program should continue execution after a function call)
 - Other variables or control data

2. Exploitation:

- Attackers can craft input that not only overflows the buffer but also manipulates the return address.
- By overwriting the return address with a pointer to malicious code (often referred to as "shellcode"), the attacker can redirect the program's execution flow.

3. Execution of Malicious Code:

- When the function returns, instead of going back to the legitimate location, the program jumps to the attacker's code, allowing them to execute arbitrary commands or gain unauthorized access.

```

#include <stdio.h>

#include <string.h>

void vulnerable_function(char *input) {
    char buffer[64]; // A buffer that can hold 64 characters
    strcpy(buffer, input); // Unsafe: no bounds checking
}

int main() {
    char *data = "This is a long string that will overflow the
buffer!";

    vulnerable_function(data); // Passes data with more
than 64 characters

    return 0;
}

```

Exploiting Buffer Overflow (Attacker's Perspective)

Step 1: Inject Malicious Code (Shellcode)

Step 2: Overwrite the Return Address

Step 3: Execute Malicious Code

```

#include <stdio.h>
#include <string.h>

void vulnerable_function(char *input) {
    char buffer[64]; // A buffer that can hold 64 characters
    strcpy(buffer, input); // Unsafe: no bounds checking
}

int main() {
    char *data =
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA";

    vulnerable_function(data); // Passes a string that overflows the buffer
    return 0;
}

```

AA
AA
AAAAA" (64 A's) + [Shellcode] + [New Return Address]

Mitigation

- **Bounds Checking:** Always validate input lengths to ensure they fit within allocated buffers.
- **Safe Functions:** `gets()`, `strcpy()`, and `sprintf()`. Instead, use safer alternatives like `fgets()`, `snprintf()`, and `strncpy()`. Avoid using unsafe functions like `gets()`, `strcpy()`, and `sprintf()`. Instead, use safer alternatives like `fgets()`, `snprintf()`, and `strncpy()`.
- **Compiler Security Features:** Enable stack protection mechanisms (like stack canaries) that help detect and prevent buffer overflows.
- **Address Space Layout Randomization (ASLR):** Randomizes memory addresses to make it harder for attackers to predict where their malicious code will reside. `0x7fffffff040`

Buffer overflows are a classic example of a security vulnerability that can lead to significant exploitation if not adequately managed. Understanding and mitigating these risks is crucial in software development and security practices.

Integer Overflow

Integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside the range that can be represented with a given number of bits. When this happens, the value "wraps around" to the opposite end of the range, leading to unexpected results.

1. Data Types and Ranges:

- Different data types in programming languages have specific ranges. For example, a signed 32-bit integer can typically hold values from -2,147,483,648 to 2,147,483,647.
- If an operation exceeds these limits, overflow occurs.

2. Overflow Behavior:

- In most programming languages, exceeding the maximum value of an integer type will result in wrapping around to the minimum value.
- For example, adding 1 to 2,147,483,647 would result in -2,147,483,648 (the minimum for a signed 32-bit integer).

```
int main() {  
    int a = INT_MAX; // Maximum value for a signed integer  
    printf("a: %d\n", a);  
  
    // Performing an operation that causes overflow  
    int b = a + 1; // This will overflow  
    printf("b: %d\n", b); // Output will be unexpected  
  
    return 0;  
}
```