

## **Unit -IV Attack Landscape - Web Application Security**

OWASP 10 Ten – Injection, **Broken Authentication**, Sensitive Data Exposure, XML External Entities, **Broken Access Control**, **Security Misconfiguration**, **Cross Site Scripting**, Insecure Deserialization, Using Components with Known Vulnerabilities, Insufficient Logging & Monitoring, **Cross Site Request Forgery**, File Inclusion, Click Jacking, File Inclusion, File Upload, Insecure Captcha, SSRF/XSPA.

---

### **1. Injection**

Injection refers to a security attack wherein malicious code is inserted into a computer program or system to alter its behavior. This attack exploits vulnerabilities in an application's input processing mechanisms, allowing an attacker to execute unauthorized commands, manipulate data, or gain access to sensitive information. Here are some key points about injections:

#### **Types of Injections:**

**SQL Injection (SQLi):** Attackers insert malicious SQL queries into input fields of a web application to manipulate the database or gain unauthorized access to sensitive data.

**Cross-Site Scripting (XSS):** Malicious scripts are injected into web pages viewed by other users, enabling attackers to steal session cookies, redirect users, or perform other malicious actions.

**Code Injection:** This involves injecting malicious code into an application to compromise its integrity or gain control over the system.

**Command Injection:** Attackers inject commands into applications that execute system-level commands, enabling unauthorized access or control of the underlying operating system.

#### **Impact of Injections:**

**Data Leakage:** Attackers can access, modify, or delete sensitive information stored in databases.

**System Compromise:** Injection attacks can lead to complete system compromise,

enabling attackers to control systems or gain unauthorized access to sensitive areas.

**Financial Loss:** Businesses can suffer financial losses due to theft of sensitive data, reputational damage, or system downtime caused by these attacks.

**Regulatory and Legal Consequences:** Companies may face legal repercussions for failing to protect user data, especially under data protection laws.

### **Prevention and Mitigation:**

**Input Validation:** Implement strict input validation techniques to ensure that user-supplied data meets expected criteria.

**Parameterized Queries:** Use parameterized queries or prepared statements in databases to prevent SQL injection.

**Escaping Input:** Employ techniques to escape user input data to prevent execution of malicious code.

**Web Application Firewalls (WAFs):** Use WAFs to filter and block malicious traffic and attempts at injection attacks.

**Regular Security Audits:** Conduct regular security audits and penetration testing to identify and fix vulnerabilities.

### **Best Practices:**

**Educate Developers:** Train developers in secure coding practices and raise awareness about the risks associated with injections.

**Update and Patch Systems:** Keep systems, frameworks, and software up-to-date with the latest security patches to prevent known vulnerabilities.

By understanding the various types, impacts, and prevention measures against injection attacks, organizations can significantly reduce the risk of falling victim to these malicious exploits.

### **1 (a) SQL Injection (SQLi):**

#### **SQL Injection:**

SQL Injection is a type of cyber attack where malicious SQL statements are inserted into an entry field for execution. These attacks exploit vulnerabilities in an application's software, allowing attackers to interfere with the application's SQL query, thereby manipulating the database.

### **How SQL Injection Works:**

**Identifying Vulnerable Input Fields:** Attackers look for input fields (like login forms, search bars, or comment sections) that accept user input and interact with a database.

**Injecting Malicious SQL Code:** By inputting specially crafted SQL commands, attackers manipulate the structure of the original SQL query. For instance, they might append additional SQL code to an existing query.

**Executing Malicious SQL Commands:** Once the injected SQL code is processed by the database, it performs unintended operations such as retrieving, modifying, or deleting data.

### **Example of SQL Injection:**

Let's consider a simple login form with two input fields: username and password. The backend SQL query might look like:

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

An attacker can manipulate this query by entering a malicious input like:

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

So, if an attacker enters this string in the password field, the resulting SQL query will look like:

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

In this case, the condition '1'='1' is always true, so the query would return a result, effectively bypassing the password check and granting unauthorized access.

### **Impact of SQL Injection:**

**Data Leakage:** Attackers can access, modify, or delete sensitive information from the database.

**Account Takeover:** Gain unauthorized access to user accounts or administrative privileges.

**Database Manipulation:** Alter or delete entire databases, leading to severe data loss.

**System Compromise:** Gain control over the entire system hosting the database.

### **Prevention of SQL Injection:**

**Input Sanitization:** Validate and sanitize user inputs to prevent unauthorized characters or SQL commands from being executed.

**Parameterized Queries/Prepared Statements:** Use prepared statements or parameterized queries to separate SQL logic from user input.

**Least Privilege Principle:** Restrict database user privileges to limit the damage caused by successful attacks.

**Regular Security Audits:** Perform regular security checks and penetration testing to identify and patch vulnerabilities.

### **Best Practices:**

**Educate Developers:** Train developers on secure coding practices to prevent SQL Injection vulnerabilities.

**Security Measures:** Implement firewalls, intrusion detection systems, and monitoring tools to detect and prevent SQL Injection attacks.

**Keep Software Updated:** Ensure that databases and frameworks are updated with the latest security patches to address known vulnerabilities.

Understanding the mechanics of SQL Injection and implementing preventive measures is crucial to protect databases and sensitive information from such attacks.

### **Example of SQL Injection Vulnerability in Python:**

Let's simulate a simple login functionality in Python using SQLite as the database. We'll show how a vulnerable query can be manipulated using SQL injection.

National Forensic Sciences University  
School of Cyber Security and Digital Forensics  
M Sc Digital Forensics and Information Security, Sem - II  
Web Application Security  
Dr. Digvijaysinh Rathod

```
import sqlite3

# Connect to an SQLite database (for demonstration purposes)
conn = sqlite3.connect(':memory:') # Creating an in-memory database
cursor = conn.cursor()

# Create a users table
cursor.execute("""
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        username TEXT,
        password TEXT
    )
""")

# Insert dummy user data
cursor.execute("INSERT INTO users (username, password) VALUES ('alice', 'password123')")
cursor.execute("INSERT INTO users (username, password) VALUES ('bob', 'securepass456')")
conn.commit()

# Simulated login function vulnerable to SQL injection
def login(username, password):
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
```

```
cursor.execute(query)

user = cursor.fetchone()

if user:

    print("Login successful!")

else:

    print("Login failed. Invalid username or password.")

# Attempt to login with SQL injection vulnerability

login('bob', "' OR '1'='1' --")

conn.close()
```

In this example, the login function constructs a SQL query using string concatenation to check the provided username and password against the database. The injected value ' OR '1'='1' -- manipulates the query to always evaluate to true ('1'='1' is always true in SQL) and comments out the rest of the original query, allowing unauthorized access.

### **Preventing SQL Injection in Python:**

To prevent SQL injection, you should use parameterized queries or prepared statements. Here's an improved version of the login function that utilizes

#### **parameterized queries:**

# Improved login function with parameterized queries to prevent SQL injection

```
def login_secure(username, password):

    query = "SELECT * FROM users WHERE username = ? AND password = ?"

    cursor.execute(query, (username, password))

    user = cursor.fetchone()

    if user:

        print("Login successful!")
```

else:

```
print("Login failed. Invalid username or password.")
```

# Attempt to login using the secure function (no SQL injection)

```
login_secure('bob', "" OR '1'='1' --")
```

By using parameterized queries (? placeholders) and passing the input separately, the database driver handles proper escaping and prevents the input from altering the query structure, effectively mitigating the SQL injection vulnerability.

## 2. Broken Authentication:

Broken authentication refers to security vulnerabilities that arise from improper implementation or management of authentication and session management mechanisms in web applications. When authentication mechanisms are poorly configured or flawed, attackers can exploit these weaknesses to gain unauthorized access to user accounts, compromising sensitive data.

### Common Causes of Broken Authentication:

**Weak Passwords:** Users employing easily guessable passwords or reusing passwords across multiple accounts.

**Insecure Storage of Credentials:** Storing passwords or sensitive information in plaintext or weakly hashed formats.

**Session Management Issues:** Improper handling of session IDs, such as exposing them in URLs, leading to session hijacking.

**Brute Force Attacks:** Repeated attempts to guess passwords or session IDs through automated scripts.

### Example of Broken Authentication:

Consider a scenario where a web application stores user passwords in plaintext in its database. If an attacker gains access to this database (due to a security breach or vulnerability), they can view all user passwords directly without any encryption.

With these plaintext passwords, the attacker can easily access user accounts, posing as legitimate users.

Another example involves poorly managed session IDs. If the application exposes session IDs in URLs or does not rotate them properly, attackers can intercept these IDs and use them to hijack authenticated sessions. For instance, an attacker might capture a session ID through packet sniffing or cross-site scripting (XSS) attacks and impersonate a logged-in user without needing their credentials.

### **Impact of Broken Authentication:**

**Account Takeover:** Attackers gain unauthorized access to user accounts, potentially compromising sensitive information.

**Data Breach:** Access to user data, financial details, or personal information.

**Financial Loss:** Theft of funds or unauthorized transactions from compromised accounts.

**Reputational Damage:** Loss of trust among users due to security breaches.

### **Prevention of Broken Authentication:**

**Secure Password Policies:** Enforce strong password requirements and encourage users to create unique, complex passwords.

**Hashing and Salting:** Store passwords securely using strong hashing algorithms with salt.

**Secure Session Management:** Implement proper session handling techniques, including session timeout, secure cookies, and rotating session IDs.

**Multi-Factor Authentication (MFA):** Use additional layers of authentication like MFA to enhance security.

**Regular Audits and Monitoring:** Conduct security audits and monitor authentication logs for suspicious activities.

### **Best Practices:**

**Regular Updates and Patches:** Keep software, frameworks, and libraries updated to address known vulnerabilities.

**Security Training:** Educate users and developers about secure authentication practices to mitigate potential risks.

**Security Headers:** Implement security headers like HTTP Strict Transport Security (HSTS) and Content Security Policy (CSP) to enhance web security.

Addressing broken authentication vulnerabilities requires a multi-layered approach that includes robust authentication mechanisms, secure storage of credentials, and continuous monitoring to detect and prevent unauthorized access.

### **Example:**

#### **Example of Sensitive Data Exposure in Python:**

Let's consider a scenario where sensitive information (like passwords) is stored in plaintext in a Python application.

# Example of storing passwords in plaintext (for demonstration purposes only)

```
users = {  
    'user1': 'password1',  
    'user2': 'password2',  
    # ... more users  
}  
  
def login(username, password):  
    if username in users and users[username] == password:  
        print("Login successful!")  
    else:  
        print("Login failed. Invalid username or password.")  
# Attempt to login with sensitive data exposure vulnerability  
login('user1', 'password1')
```

In this example, the application stores usernames and passwords in a dictionary without any encryption or hashing. If an attacker gains access to this Python script or the data it accesses, they can easily retrieve all the passwords because they are stored in plaintext.

### **Prevention of Sensitive Data Exposure in Python:**

To prevent sensitive data exposure, especially in handling passwords or other sensitive information, you should use proper encryption and secure coding practices:

```
import hashlib
```

```
# Example with hashed passwords
```

```
users = {  
    'user1': hashlib.sha256('password1'.encode()).hexdigest(),  
    'user2': hashlib.sha256('password2'.encode()).hexdigest(),  
    # ... more users  
}
```

```
def login(username, password):
```

```
    if username in users and users[username] ==  
        hashlib.sha256(password.encode()).hexdigest():
```

```
        print("Login successful!")
```

```
    else:
```

```
        print("Login failed. Invalid username or password.")
```

```
# Attempt to login with hashed passwords (properly secured)
```

```
login('user1', 'password1')
```

In this improved example, passwords are hashed using a secure hashing algorithm (SHA-256 in this case) before being stored in the dictionary. During the login process, the entered password is also hashed and compared with the stored hashed password.

### **Key Points to Note:**

Storing passwords as plaintext is a vulnerability. Always hash passwords using strong cryptographic hashing algorithms like SHA-256 or bcrypt before storing them.

When comparing passwords during authentication, hash the entered password and compare it with the stored hashed password instead of directly comparing plaintext passwords.

By employing secure practices like hashing passwords and encrypting sensitive data, Python applications can significantly reduce the risk of sensitive data exposure.

### **3. Broken Access Control**

Broken Access Control refers to a security vulnerability where users gain unauthorized access to resources or functionalities they shouldn't have access to within a web application. It occurs when access controls (such as authentication, authorization, and session management) are improperly implemented or enforced.

#### **Key Points about Broken Access Control:**

**Authentication vs. Authorization:** Authentication confirms a user's identity, while authorization determines what an authenticated user can access or do within an application.

#### **Common Vulnerabilities:**

**Insecure Direct Object References (IDOR):** When an attacker can directly access resources by manipulating references (e.g., changing a URL to access another user's profile).

**Privilege Escalation:** Unauthorized elevation of privileges, allowing users to access administrative functions or sensitive data.

**Missing Function-Level Access Control:** Allowing users to access functionalities or APIs they shouldn't have access to.

**Impact of Broken Access Control:**

**Data Exposure:** Unauthorized users accessing sensitive information.

**Data Modification:** Unauthorized users altering or deleting critical data.

**System Compromise:** Granting excessive privileges can lead to system takeover or manipulation.

**Example of Broken Access Control:**

Consider a scenario where a web application allows users to access sensitive information, such as user profiles. However, the application fails to properly authenticate and authorize users, leading to an Insecure Direct Object Reference (IDOR) vulnerability:

**Example of Broken Access Control in Python:**

Let's simulate a basic web application scenario where users have different roles (admin and regular user), but the access control checks are insufficiently enforced.

```
class User:
```

```
    def __init__(self, username, role):
```

```
        self.username = username
```

```
        self.role = role
```

```
# Simulated database of users and their roles
```

```
users_db = {
```

```
    'user1': User('user1', 'regular'),
```

```
    'user2': User('user2', 'admin'),
```

```
    # ... more users
```

```
}
```

# Function to view user profile (with broken access control)

```
def view_profile(current_user, profile_user):
```

```
    if current_user.role == 'admin' or current_user.username ==  
profile_user.username:
```

```
        print(f"Viewing profile of {profile_user.username}")
```

```
        # Here would be code to retrieve and display the profile
```

```
    else:
```

```
        print("Access Denied. You don't have permission to view this profile.")
```

# Simulate a scenario where a regular user tries to view an admin profile (broken access control)

```
current_user = users_db['user1'] # Regular user
```

```
profile_user = users_db['user2'] # Admin user
```

# Attempt to view the admin profile (broken access control allows unauthorized access)

```
view_profile(current_user, profile_user)
```

In this example:

User class simulates user details with usernames and roles (admin or regular user).

The view\_profile function is supposed to restrict access based on the user's role. Admins can view any profile, while regular users can only view their own profiles.

However, the access control logic is insufficiently implemented, allowing a regular user (user1) to view an admin's (user2) profile due to a flawed conditional check.

### **Prevention of Broken Access Control in Python:**

**To mitigate Broken Access Control vulnerabilities in Python, you should:**

- Implement robust access control mechanisms based on roles and permissions.
- Regularly review and test access controls to ensure they're correctly enforcing restrictions.
- Enforce proper authentication and authorization checks to prevent unauthorized access to sensitive functionalities or data.
- Use frameworks or libraries that offer built-in security features for authentication and authorization.

By improving access control logic and thoroughly testing scenarios where users try to access resources they shouldn't, you can prevent Broken Access Control vulnerabilities in Python applications.

#### 4. Security Misconfiguration

Security misconfiguration refers to the improper implementation or setup of security controls within a system, application, or network. It occurs when default configurations, unnecessary features, or weak security settings are left unchanged or improperly configured, leading to potential vulnerabilities.

##### Key Points about Security Misconfiguration:

###### Common Causes:

**Default Configurations:** Leaving default settings unchanged, which might include default passwords, ports, or settings that are easily exploitable.

**Unused Features and Services:** Failure to disable or remove unused features or services, which can be potential entry points for attackers.

**Improper Error Handling:** Revealing too much information in error messages, providing attackers with insights into system architecture or vulnerabilities.

**Outdated Software:** Using outdated software or libraries that contain known security vulnerabilities.

###### Examples of Security Misconfiguration:

**Default Credentials:** A web application's administrative interface uses default credentials that were not changed, allowing unauthorized access.

**Exposed Configuration Files:** Configuration files (like .env or properties files) containing sensitive information, such as API keys or database credentials, are exposed publicly.

**Unnecessary Open Ports:** Servers or services running on ports that are not required for the application's functionality, providing unnecessary attack surfaces.

**Improper Error Handling:** Detailed error messages displayed to users or in logs, exposing internal system information to potential attackers.

### **Impact of Security Misconfiguration:**

**Data Breaches:** Unauthorized access to sensitive data due to improperly configured access controls.

**System Compromise:** Attackers gaining control over systems, leading to manipulation, data loss, or disruption of services.

**Reputational Damage:** Loss of trust among users or clients due to security incidents or breaches.

### **Prevention of Security Misconfiguration:**

**Secure Configuration Standards:** Follow industry best practices and secure configuration guidelines for servers, frameworks, and applications.

**Regular Patching and Updates:** Keep systems and software up-to-date with the latest security patches and updates.

**Least Privilege Principle:** Grant minimal necessary access and disable or remove unused features or services.

**Secure Error Handling:** Implement generic error messages to users and detailed logs for developers/administrators without revealing sensitive information.

### **Best Practices:**

**Automated Tools:** Use automated security scanners or tools to identify misconfigurations and vulnerabilities.

**Regular Audits and Reviews:** Conduct periodic security audits and reviews to identify and rectify misconfigurations.

**Documentation and Training:** Educate developers, administrators, and users about secure configuration practices.

**Example of Security Misconfiguration:**

Let's consider a scenario where an application's debug mode is left enabled in a production environment:

```
# Flask web application with debug mode enabled (insecure configuration)

from flask import Flask

app = Flask(__name__)

app.config['DEBUG'] = True # Debug mode enabled

@app.route('/')

def home():

    return 'Welcome to the home page!'

if __name__ == '__main__':

    app.run()
```

Enabling debug mode in a production environment can expose sensitive information, stack traces, or even allow remote code execution, presenting a significant security risk.

By understanding security misconfiguration risks and actively implementing secure configuration practices, organizations can significantly reduce the chances of falling victim to such vulnerabilities.

**Example of Security Misconfiguration in Python:**

```
# Simulated application with a security misconfiguration in error handling

def get_user_profile(user_id):

    # Assume this function fetches user profile from a database (for demonstration purposes)
```

# In this example, the function is simplified and includes a potential security misconfiguration

# Retrieving user profile from the database (simulated)

# For demonstration, assume an error might occur due to an invalid user ID

try:

    # Code to fetch user profile (simulated)

    # ... (database query)

    # Simulating an error due to invalid user ID

    if user\_id == 999:

        raise ValueError("Invalid user ID")

    # Return user profile (simulated)

    return f"User profile for user ID {user\_id}: Name, Email, Address, etc."

except Exception as e:

    # Insecure error handling exposing sensitive information

    # For demonstration, this code reveals detailed error messages to the end user

    return f"Error: {str(e)}"

# Simulate a scenario where an attacker tries to access user profiles with an invalid user ID

invalid\_user\_id = 999

user\_profile = get\_user\_profile(invalid\_user\_id)

print(user\_profile) # Displaying the fetched user profile (simulated)

### **Explanation:**

#### **In this Python example:**

The `get_user_profile` function attempts to fetch a user's profile from a database based on the provided user ID.

For demonstration purposes, an error is intentionally raised when the user ID is set to 999, simulating an invalid user ID.

However, the error handling mechanism in this example is insecure. It reveals detailed error messages directly to the end user, potentially exposing sensitive information about the system's internal workings, such as database errors or invalid user IDs.

### **Prevention:**

#### **To prevent such security misconfigurations related to error handling:**

Implement secure error handling that provides generic error messages to end users, avoiding the exposure of sensitive information.

Ensure that error messages are logged or reported to administrators/developers for debugging purposes without revealing internal details to users.

By implementing secure error handling practices, organizations can mitigate the risk of exposing sensitive information in error messages, thus reducing the impact of security misconfigurations.

### **5. Cross-Site Scripting (XSS):**

Cross-Site Scripting is a type of security vulnerability commonly found in web applications. It occurs when attackers inject malicious scripts (usually JavaScript) into web pages viewed by other users. This vulnerability arises due to inadequate input validation and improper encoding or escaping of user-generated content.

#### **Key Points about Cross-Site Scripting (XSS):**

##### **Types of XSS:**

**Reflected XSS:** Malicious scripts are injected into a website and executed when users visit a specially crafted URL.

**Stored (Persistent) XSS:** Malicious scripts are stored on the server and executed whenever a user accesses a compromised page or resource.

**DOM-based XSS:** Occurs when client-side scripts manipulate the Document Object Model (DOM) in an unsafe way.

### **Common Causes:**

**Improper Input Sanitization:** Failing to validate or properly encode/escape user inputs before displaying them on web pages.

**Dynamic Content Injection:** Inserting unfiltered user-supplied data into HTML content without proper validation.

### **Impact of XSS:**

**Session Hijacking:** Attackers can steal session cookies, enabling unauthorized access to user accounts.

**Data Theft:** Access or modification of sensitive data stored in cookies or the DOM.

**Malicious Actions:** Redirection to phishing sites, defacement of web pages, or performing actions on behalf of users.

### **Example of Reflected XSS:**

Consider a search functionality vulnerable to XSS:

```
<!-- Simulated search page vulnerable to Reflected XSS -->
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <title>Search Page</title>
```

```
</head>
```

```
<body>
```

```
<h1>Search Page</h1>
<form action="/search" method="GET">
  <input type="text" name="query" placeholder="Search query">
  <input type="submit" value="Search">
</form>
<div>
  Results for: <!-- Outputting user-supplied query without proper sanitization -->
  <span id="searchResult"></span>
</div>
<script>
  // Simulating JavaScript rendering search results
  const urlParams = new URLSearchParams(window.location.search);
  const query = urlParams.get('query');
  document.getElementById('searchResult').innerText = query; // Vulnerable to
XSS
</script>
</body>
</html>
```

An attacker can craft a malicious URL with a script as the query parameter:

[http://vulnerable-website.com/search?query=<script>alert\('XSS'\)</script>](http://vulnerable-website.com/search?query=<script>alert('XSS')</script>)

When a victim visits this URL, the script gets executed, showing an alert pop-up in the victim's browser.

### **Prevention of XSS:**

**Input Sanitization:** Validate and sanitize user inputs to prevent the execution of injected scripts.

**Encoding/Output Escaping:** Encode user-generated content before rendering it in HTML to neutralize potential malicious code.

**Content Security Policy (CSP):** Implement a strong CSP to control which sources scripts can be loaded from.

**XSS Auditors and Security Headers:** Use XSS filters and security headers (like X-XSS-Protection) to mitigate XSS attacks.

### **Best Practices:**

**Security Awareness Training:** Educate developers about secure coding practices to prevent XSS vulnerabilities.

**Regular Security Testing:** Perform security audits, code reviews, and penetration testing to identify and fix vulnerabilities.

**Updates and Patches:** Keep frameworks, libraries, and systems updated to address known security issues.

By applying proper input validation, output encoding, and security measures, web applications can mitigate the risks associated with Cross-Site Scripting vulnerabilities.

### **Example:**

example in Python illustrating a simple web page vulnerable to Reflected Cross-Site Scripting (XSS):

### **Example of Reflected XSS Vulnerability in Python:**

Let's create a simple web application using Flask framework that includes a search functionality vulnerable to Reflected XSS:

```
from flask import Flask, request, render_template_string

app = Flask(__name__)

# Simulated search page vulnerable to Reflected XSS

@app.route('/search', methods=['GET'])

def search_page():
```

```
query = request.args.get('query', "")
# Simulating rendering search results in HTML (vulnerable to XSS)
search_result = f"Results for: {query}"
return render_template_string(
    """
    <!DOCTYPE html>

    <html>

    <head>

        <title>Search Page</title>

    </head>

    <body>

        <h1>Search Page</h1>

        <div>

            {{ search_result }}

        </div>

    </body>

    </html>

    """, search_result=search_result
)

if __name__ == '__main__':
    app.run(debug=True)
```

This code creates a simple web application using Flask. The /search route takes a query parameter from the URL and displays it on the page. However, it does not properly sanitize or escape the query parameter, making it vulnerable to XSS attacks.

### **Exploiting the XSS Vulnerability:**

An attacker can craft a malicious URL containing a script as the query parameter:

[http://localhost:5000/search?query=<script>alert\('XSS'\)</script>](http://localhost:5000/search?query=<script>alert('XSS')</script>)

When a victim visits this URL, the JavaScript code `<script>alert('XSS')</script>` gets executed in the victim's browser, displaying an alert with the text 'XSS'.

### **Prevention of XSS in Python:**

To prevent XSS vulnerabilities in Python-based web applications:

Sanitize and validate user inputs by properly escaping or encoding them before displaying in HTML.

Use templating systems or libraries that automatically escape user inputs to prevent script execution.

Implement security headers like Content Security Policy (CSP) to restrict the sources from which scripts can be loaded.

By applying proper input validation and output encoding techniques, developers can prevent XSS vulnerabilities and protect their web applications from such attacks.

## **6. Insecure deserialization**

Insecure deserialization is a vulnerability that occurs when a web application or system deserializes untrusted data without proper validation or integrity checks. This vulnerability can be exploited by attackers to execute arbitrary code, manipulate data, or perform various types of attacks. Here are some important points about insecure deserialization along with an example:

### **Deserialization Process:**

Deserialization is the process of converting serialized data (often in the form of JSON, XML, or binary data) into an object that can be used by the application.

### **Vulnerabilities Associated:**

Insecure deserialization vulnerabilities arise when the application deserializes data from untrusted or manipulated sources without verifying the integrity of the serialized data or the objects being created.

### **Common Attack Scenarios:**

**Remote Code Execution (RCE):** Attackers can craft malicious serialized objects that, when deserialized by the application, execute arbitrary code.

**Data Tampering:** Manipulating serialized objects to modify their content, leading to unexpected behavior or unauthorized access.

**Denial of Service (DoS):** Crafting malicious serialized data that causes resource exhaustion or crashes the application upon deserialization.

### **Example of Insecure Deserialization:**

Consider a hypothetical scenario where a web application uses serialized objects to manage user sessions:

```
# Serialized session data
```

```
serialized_data = '...malicious_data_here...'
```

```
# Deserialization process
```

```
session = deserialize(serialized_data)
```

An attacker tampers with the serialized session data to inject malicious code:

```
{  
  "class": "com.example.UserProfile",  
  "data": {  
    "username": "attacker",  
    "isAdmin": true,  
    "maliciousCode": "System.exec('rm -rf /')"  
  }  
}
```

When the application deserializes this manipulated data, it creates an instance of the `UserProfile` class with an `isAdmin` attribute set to `true` and executes the malicious code `System.exec('rm -rf /')`, which attempts to delete files from the system.

## **Prevention and Mitigation:**

**Validate Serialized Data:** Implement integrity checks and validation mechanisms to ensure the integrity and authenticity of serialized objects.

**Use Safe Deserialization Libraries:** Employ secure deserialization libraries or frameworks that offer protection against insecure deserialization vulnerabilities.

**Least Privilege Principle:** Restrict the capabilities of deserialized objects and avoid executing code from user-controlled or untrusted sources.

**Input Sanitization:** Sanitize input data to prevent malicious content from being deserialized.

## **example demonstrating insecure deserialization vulnerability in Python:**

Let's say we have a simple Python application that deserializes user input using the pickle module, which can be vulnerable to insecure deserialization if used without proper precautions.

### **Python Insecure Deserialization Example:**

```
import pickle
```

```
# Function to deserialize user input
```

```
def deserialize_user_input(user_input):
```

```
    return pickle.loads(user_input)
```

```
# Example usage
```

```
if __name__ == "__main__":
```

```
    # Simulating user input (could come from an attacker)
```

```
    user_input
```

```
=
```

```
b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00\x8c\x08builtins\x94\x8c\x07globals\x94\x8c\x02x1\x94\x93\x94.'
```

try:

```
# Deserialization of user input
```

```
deserialized_data = deserialize_user_input(user_input)
```

```
print("Deserialized data:", deserialized_data)
```

except Exception as e:

```
print("Deserialization error:", e)
```

In this example:

- The pickle module in Python is used for serialization and deserialization.
- The `deserialize_user_input` function attempts to deserialize user input using `pickle.loads`.
- The `user_input` variable contains a serialized object (can be manipulated by an attacker).

Attack Scenario:

An attacker might craft a malicious serialized object and send it as user input:

```
# Serialized malicious payload (attacker's input)
```

```
malicious_input =  
b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00\x00\x8c\x08builtins\x94\x8c\x07globals\x94\x8c\x02x1\x94\x93\x94.'
```

### **Exploitation:**

When the application attempts to deserialize this input:

The `pickle.loads` method processes the malicious serialized data.

If the deserialization occurs without proper validation or sanitization, it might execute arbitrary code embedded in the serialized object.

### **Prevention:**

To prevent insecure deserialization in Python:

- Avoid using serialization libraries (like pickle) with untrusted or unsanitized user input.
- Implement strict input validation and use safer serialization formats like JSON or use libraries that provide secure deserialization mechanisms.
- Always exercise caution while deserializing data, especially when it originates from untrusted sources, to prevent potential security vulnerabilities.

## 7. Using components with known vulnerabilities

Using components with known vulnerabilities refers to the practice of incorporating third-party libraries, frameworks, or software modules within an application that contain publicly documented security flaws or weaknesses. Here are some notes on this topic along with an example:

### Understanding the Issue:

### Dependency Management:

- Modern software development often relies on various third-party components to speed up development and add functionalities.
- However, these dependencies can introduce vulnerabilities into an application if not managed carefully.

### Known Vulnerabilities:

Developers might unintentionally integrate components with documented security vulnerabilities into their projects.

These vulnerabilities can range from weaknesses in encryption algorithms to critical remote code execution flaws.

### Example Scenario:

Consider a web application built using multiple libraries and frameworks, one of which has a known vulnerability:

```
# Python Flask application with a vulnerable dependency
```

```
from flask import Flask
```

```
from vulnerable_library import vulnerable_function
```

```
app = Flask(__name__)  
  
@app.route('/')  
def index():  
    # Using a function from a vulnerable library  
    vulnerable_function()  
    return 'Hello, world!'
```

### **In this scenario:**

- The `vulnerable_library` being imported contains a function `vulnerable_function` with a documented security flaw.
- Even if the rest of the application is secure, the presence of this vulnerable dependency poses a risk.

### **Risks and Impact:**

#### **Exploitation Potential:**

Attackers can exploit known vulnerabilities to gain unauthorized access, execute arbitrary code, or compromise sensitive data.

#### **Impact on Reputation and Compliance:**

Using components with known vulnerabilities can damage an organization's reputation and might lead to non-compliance with industry standards or regulations.

#### **Best Practices for Mitigation:**

##### **Regular Updates and Patching:**

Stay informed about security advisories and update components to patched versions that address known vulnerabilities.

##### **Vulnerability Monitoring:**

Employ tools or services that track and notify about vulnerabilities in dependencies.

##### **Dependency Scanning and Auditing:**

Conduct regular security audits to identify and replace vulnerable components.

### **Defense-in-Depth Approach:**

Implement multiple layers of security measures beyond just patching, such as firewalls, input validation, and secure coding practices.

### **Risk Assessment and Remediation:**

Prioritize vulnerabilities based on severity and potential impact, focusing on critical issues first.

### **Conclusion:**

Utilizing components with known vulnerabilities can significantly compromise the security posture of an application. Proactive measures, such as continuous monitoring, timely updates, and thorough risk assessment, are crucial to mitigate these risks and ensure the overall security of software systems.

## **8. Insufficient logging and monitoring**

Insufficient logging and monitoring is a significant security weakness that occurs when an application or system lacks proper mechanisms to detect, log, and respond to security incidents or suspicious activities in a timely manner. Here are notes along with an example to illustrate this issue:

### **Understanding the Problem:**

#### **Logging and Monitoring:**

Logging involves recording events, errors, and activities within an application or system.

Monitoring involves actively observing these logs for suspicious or potentially malicious activities.

#### **Insufficient Logging:**

When an application does not log sufficient information or logs are not adequately reviewed, security incidents can go undetected.

#### **Example Scenario:**

Consider a web application that lacks proper logging and monitoring:

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/login', methods=['POST'])
def login():

    username = request.form.get('username')
    password = request.form.get('password')

    # Inadequate logging
    # Logins are not being adequately logged
    # No monitoring mechanism to detect multiple failed login attempts

    if username == 'admin' and password == 'admin123':
        return 'Login successful'
    else:
        return 'Login failed'
```

**In this scenario:**

- The application lacks detailed logging of login attempts, especially failed login attempts.
- There's no monitoring mechanism to detect multiple failed login attempts or potential brute-force attacks.
- Without comprehensive logs or monitoring, it's difficult to identify or respond to suspicious activities or potential security threats.

**Risks and Impact:**

**Delayed Incident Response:**

Insufficient logging delays incident detection and response, allowing attackers more time to exploit vulnerabilities or compromise systems.

**Difficulty in Forensics:**

Inadequate logs make it challenging to conduct post-incident forensics or investigations, hindering the understanding of the attack vector or impact.

### **Best Practices for Mitigation:**

#### **Comprehensive Logging:**

Log relevant events, errors, user activities, and security-related actions in detail, including both successful and failed attempts.

#### **Real-time Monitoring:**

Implement monitoring systems that actively analyze logs for anomalies or suspicious patterns, triggering alerts for further investigation.

#### **Response Mechanisms:**

Establish incident response procedures to take immediate action upon detection of suspicious activities or security breaches.

#### **Log Retention and Encryption:**

Retain logs securely for an appropriate duration and consider encryption to protect sensitive information within logs.

#### **Regular Auditing and Review:**

Conduct regular reviews and audits of logs to ensure they contain sufficient information and are actively monitored.

#### **Conclusion:**

Insufficient logging and monitoring significantly weaken an application's security posture by hindering the timely detection and response to security incidents. Robust logging practices, coupled with proactive monitoring and response mechanisms, are essential to identify and mitigate security threats effectively. Incorporating these practices is crucial for maintaining a resilient security stance in applications and systems.

## 9. Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a type of security vulnerability in web applications where an attacker tricks a user into unknowingly performing actions on a website that they are authenticated to, without the user's consent or awareness. Here are notes on CSRF along with an example to illustrate the concept:

Understanding CSRF:

### Attack Mechanism:

- CSRF occurs when an attacker exploits the trust that a website has in a user's browser.
- The attacker forces the user's browser to execute unwanted actions on a targeted website, utilizing the user's active session or authentication.

### Attack Vectors:

- Maliciously crafted links in emails, forums, or social media.
- Hidden form submissions, image tags, or JavaScript included on compromised websites.

### Example Scenario:

Consider a simple banking application vulnerable to CSRF:

```
<!DOCTYPE html>
<html>
<head>
  <title>Transfer Money</title>
</head>
<body>
  <h1>Transfer Money</h1>
  <form action="https://bank.example.com/transfer" method="post">
    <input type="hidden" name="amount" value="1000">
```

```
<input type="hidden" name="to_account" value="attacker_account">
<input type="submit" value="Click to Claim $1000 Reward">
</form>
</body>
</html>
```

### **In this scenario:**

- The attacker crafts a malicious HTML page with a hidden form that submits a request to transfer money from the victim's account to the attacker's account.
- The victim, who is logged into their bank account in the same browser, visits the attacker's page and unknowingly triggers the transfer by clicking a seemingly harmless button.

### **Risks and Impact:**

#### **Unauthorized Actions:**

CSRF attacks enable attackers to perform unauthorized actions on behalf of authenticated users.

#### **Data or Financial Loss:**

Attackers can manipulate transactions, change account details, or perform unwanted operations leading to financial or data loss.

### **Best Practices for Mitigation:**

#### **CSRF Tokens:**

Implement CSRF tokens unique to each user session. Validate these tokens on each request to ensure the request originated from the legitimate user.

#### **SameSite Cookies:**

Utilize the SameSite attribute for cookies to restrict cross-origin requests and prevent CSRF attacks.

#### **Referrer Policy:**

Set strict Referrer Policy headers to control information sent to other websites, mitigating the risk of CSRF attacks.

### **Security Headers:**

Implement Content Security Policy (CSP) and X-Frame-Options headers to mitigate CSRF risks and frame-based attacks.

### **Educate Users:**

Educate users to log out of sensitive applications after use and be cautious of clicking suspicious links or visiting untrusted websites.

### **Conclusion:**

CSRF attacks exploit the trust between a user's browser and a website. Mitigating CSRF vulnerabilities requires implementing robust security measures within web applications to prevent unauthorized actions and protect user data and assets from exploitation. Implementing best practices and maintaining awareness about CSRF risks are crucial to defend against such attacks.

### **Python-based example demonstrating a simple CSRF attack scenario:**

Suppose there is a vulnerable web application that allows users to change their email address without proper CSRF protection:

#### **Vulnerable Python Flask Application:**

```
from flask import Flask, request, render_template_string
```

```
app = Flask(__name__)
```

```
# Simulated user data (for demonstration purposes)
```

```
user_data = {  
    'username': 'user123',
```

National Forensic Sciences University  
School of Cyber Security and Digital Forensics  
M Sc Digital Forensics and Information Security, Sem - II  
Web Application Security  
Dr. Digvijaysinh Rathod

```
'email': 'user@example.com'
}

# Endpoint to change email address (vulnerable to CSRF)
@app.route('/change_email', methods=['POST'])
def change_email():
    new_email = request.form.get('new_email')
    user_data['email'] = new_email    # Simulated change of email (no CSRF
    protection)
    return 'Email changed successfully'

# Profile page displaying user details
@app.route('/profile')
def profile():
    return render_template_string(
        '''
        <h1>Welcome, {{ user_data['username'] }}!</h1>
        <p>Email: {{ user_data['email'] }}</p>
        <form action="/change_email" method="post">
            <input type="text" name="new_email" placeholder="Enter new email">
            <input type="submit" value="Change Email">
        </form>
        '''
    )
```

```
if __name__ == '__main__':
```

```
    app.run(debug=True)
```

### **CSRF Attack Page (Malicious HTML):**

Now, let's create a malicious HTML page that triggers a CSRF attack against the above Flask application:

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <title>Malicious Page</title>
```

```
</head>
```

```
<body>
```

```
    <h1>Malicious CSRF Page</h1>
```

```
    <p>This page will perform a CSRF attack on the vulnerable web application.</p>
```

```
    <!-- Malicious form triggering the CSRF attack -->
```

```
    <form          id="csrf-form"          action="http://localhost:5000/change_email"
method="post" style="display: none;">
```

```
        <input type="hidden" name="new_email" value="attacker@example.com">
```

```
    </form>
```

```
    <!-- JavaScript to automatically submit the form -->
```

```
    <script>
```

```
        document.addEventListener('DOMContentLoaded', function() {
```

```
            var form = document.getElementById('csrf-form');
```

```
form.submit();  
  
});  
  
</script>  
  
</body>  
  
</html>
```

### **CSRF Attack Scenario:**

- The vulnerable Flask application allows changing the user's email without proper CSRF protection.
- The attacker tricks a logged-in user into visiting the malicious HTML page.
- The hidden form within the malicious page automatically submits a request to change the user's email to attacker@example.com without the user's knowledge or consent when the page loads.

This demonstrates how a CSRF attack can exploit the trust a website has in a user's browser to perform unauthorized actions on behalf of the authenticated user. Implementing CSRF protection, such as unique tokens associated with each user session, is crucial to prevent such attacks in real-world applications.

## **10.File Inclusion**

File Inclusion vulnerabilities occur when an application allows an attacker to include a file, usually a script, that can be executed within the web server's environment. This vulnerability is often found in web applications that dynamically include files or scripts based on user input or unsanitized input from other sources. There are two main types: Local File Inclusion (LFI) and Remote File Inclusion (RFI).

### **Understanding File Inclusion:**

- **Local File Inclusion (LFI):**

Involves including local files already present on the server. Attackers exploit this by manipulating file paths to execute arbitrary code.

- **Remote File Inclusion (RFI):**

Involves including files from remote locations (external URLs). Attackers insert malicious URLs to execute code hosted on external servers.

### **Example Scenario (Local File Inclusion - LFI):**

Suppose a web application dynamically includes files based on a parameter passed in the URL:

```
<?php
```

```
// Example vulnerable PHP code susceptible to LFI
```

```
$file = $_GET['file'];
```

```
include('/path/to/files/' . $file . '.php');
```

```
?>
```

### **Attack by Manipulating URL:**

An attacker can manipulate the URL parameter to include sensitive system files:

<https://vulnerable-website.com/?file=../../../../etc/passwd>

If the application is vulnerable to LFI, the attacker can traverse directories (../../../../) to access sensitive system files (/etc/passwd) and potentially retrieve sensitive information.

### **Risks and Impact:**

- **Information Disclosure:**

Attackers can access and retrieve sensitive data, such as configuration files, user credentials, or system files.

- **Arbitrary Code Execution:**

Inclusion of malicious code allows attackers to execute arbitrary commands within the server's context.

### **Best Practices for Mitigation:**

- **Input Validation and Sanitization:**

Validate and sanitize user inputs to prevent directory traversal and restrict file inclusion to allowed directories.

- **Whitelisting:**

Maintain a whitelist of allowed files or directories that can be included by the application.

- **Use Absolute Paths:**

Use absolute paths instead of relative paths to include files, ensuring that files are sourced from specific directories.

- **File Permissions:**

Restrict access permissions on sensitive files and directories to prevent unauthorized access.

- **Disable Remote Includes:**

Avoid including files from remote locations unless absolutely necessary and done securely.

**Conclusion:**

File Inclusion vulnerabilities are critical security weaknesses that allow attackers to access sensitive information or execute malicious code. Proper input validation, strict file inclusion policies, and secure coding practices are essential to prevent such vulnerabilities and enhance the overall security of web applications.

## **11. File Upload vulnerability**

File Upload vulnerability refers to security weaknesses in web applications that allow attackers to upload malicious files to the server. Such vulnerabilities can be exploited to compromise the server's integrity, execute arbitrary code, or perform other malicious activities. Here are detailed notes on File Upload vulnerabilities:

### **Understanding File Upload Vulnerabilities:**

#### **File Upload Functionality:**

- Enables users to submit files to the server via web forms or APIs.
- Insufficient validation or security measures can lead to potential exploits.

### **Common Risks:**

- Malicious file uploads (e.g., scripts or malware) can compromise the server's security.
- File upload vulnerabilities can enable attackers to execute arbitrary code, perform denial-of-service attacks, or bypass security measures.

### **Example Scenario:**

Suppose there's a web application that allows file uploads without proper validation:

```
<?php
```

```
// Example PHP code for handling file uploads (vulnerable)
```

```
$target_dir = "uploads/";
```

```
$target_file = $target_dir . basename($_FILES["fileToUpload"]["name"]);
```

```
$uploadOk = 1;
```

```
$imageFileType = strtolower(pathinfo($target_file, PATHINFO_EXTENSION));
```

```
// Check if the file is an image
```

```
if(isset($_POST["submit"])) {
```

```
    $check = getimagesize($_FILES["fileToUpload"]["tmp_name"]);
```

```
    if($check !== false) {
```

```
        echo "File is an image - " . $check["mime"] . ".";
```

```
        $uploadOk = 1;
```

```
    } else {
```

```
        echo "File is not an image.";
```

```
        $uploadOk = 0;
```

```
    }
```

National Forensic Sciences University  
School of Cyber Security and Digital Forensics  
M Sc Digital Forensics and Information Security, Sem - II  
Web Application Security  
Dr. Digvijaysinh Rathod

```
}

// Check file size
if ($_FILES["fileToUpload"]["size"] > 500000) {
    echo "Sorry, your file is too large.";
    $uploadOk = 0;
}

// Allow only certain file formats
if($imageFileType != "jpg" && $imageFileType != "png" && $imageFileType !=
"jpeg"
&& $imageFileType != "gif" ) {
    echo "Sorry, only JPG, JPEG, PNG & GIF files are allowed.";
    $uploadOk = 0;
}

// Check if $uploadOk is set to 0 by an error
if ($uploadOk == 0) {
    echo "Sorry, your file was not uploaded.";
// If everything is fine, try to upload the file
} else {
    if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {
        echo "The file ". htmlspecialchars( basename(
$_FILES["fileToUpload"]["name"])) . " has been uploaded.";
    } else {
        echo "Sorry, there was an error uploading your file.";
    }
}
```

}

?>

### **Risks and Impact:**

#### **Execution of Malicious Code:**

- Uploading scripts or malware could lead to arbitrary code execution on the server.
- Server Compromise:
- Attackers may exploit vulnerabilities to compromise server integrity or perform Denial-of-Service (DoS) attacks.

#### **Mitigation Measures:**

##### **Strict File Type Validation:**

Allow only specific, safe file types for upload.

##### **File Size Limitations:**

**Enforce limitations on file sizes to prevent large or excessive uploads.**

##### **Secure Storage and Permissions:**

Store uploaded files in secure, non-web-accessible directories with appropriate permissions.

##### **Scanning and Sanitization:**

Implement antivirus or malware scanning to detect and remove malicious files.

##### **Rename Files:**

Rename uploaded files to prevent the execution of scripts by manipulating file names.

#### **Conclusion:**

File Upload vulnerabilities pose significant risks to web application security. By implementing stringent file type validation, secure storage practices, and continuous monitoring, developers can effectively mitigate these vulnerabilities and enhance the overall security posture of their applications. Regular security audits and staying

updated on best practices are crucial for addressing File Upload vulnerabilities effectively.

## 12. SSRF (Server-Side Request Forgery) / XSPA (Cross-Site Port Attacks) along with some comprehensive notes on each:

### **SSRF (Server-Side Request Forgery):**

#### **What is SSRF:**

SSRF is a web security vulnerability that allows an attacker to induce the server to make requests on behalf of the attacker.

#### **Attack Vector:**

Exploits the ability to make requests from the vulnerable server to internal or external resources accessible to the server.

#### **Example Scenario:**

An attacker could trick the server into making requests to internal services or private networks, extract sensitive information, or perform actions that were not intended.

#### **Mitigation Measures:**

- Input validation and whitelisting of allowed resources.
- Implement firewalls, network segmentation, and strict access controls to limit server access.

### **XSPA (Cross-Site Port Attacks):**

#### **What is XSPA:**

XSPA is a variant of SSRF that focuses on port scanning to determine if certain ports are open or closed on the internal network.

#### **Attack Vector:**

Exploits the server's ability to connect to specific ports on arbitrary IP addresses or domains, checking for open ports.

#### **Example Scenario:**

An attacker could leverage XSPA to determine the accessibility of sensitive services like databases or internal servers.

### **Mitigation Measures:**

- Restrict outgoing traffic to prevent servers from accessing internal IPs or ports not necessary for operation.
- Employ web application firewalls to detect and block suspicious outgoing connections.

### **Common Mitigation Strategies for Both SSRF and XSPA:**

#### **Input Validation:**

Sanitize and validate user inputs, especially URLs or parameters used in making external requests.

#### **Whitelisting:**

Define and enforce strict whitelists for permitted resources or URLs that the server can access.

#### **URL Parsing and Filtering:**

Use libraries or functions that parse and validate URLs, ensuring they conform to expected formats and protocols.

#### **Restricting Access:**

Employ network-level controls, firewalls, and access restrictions to limit server-side access to internal or sensitive resources.

#### **Logging and Monitoring:**

Implement comprehensive logging and monitoring mechanisms to detect and respond to suspicious activities or unexpected outbound traffic.

#### **Conclusion:**

SSRF and XSPA are critical vulnerabilities that attackers exploit to interact with internal resources or networks. To mitigate these risks, robust input validation, strict

National Forensic Sciences University  
School of Cyber Security and Digital Forensics  
M Sc Digital Forensics and Information Security, Sem - II  
Web Application Security  
Dr. Digvijaysinh Rathod

access controls, and continuous monitoring are essential. By implementing these best practices and staying vigilant about emerging threats, organizations can significantly reduce the likelihood of SSRF and XSPA vulnerabilities in their systems.