**Name:Kaushik Kotian Roll No.:29 Div:D20B Batch:B**

# EXPERIMENT No:- 7

**AIM:- Experiment to Validation data and Authentication.**

**THEORY:-**

**1. SQL Injection (SQLi) Vulnerability**

Definition:
SQL Injection occurs when an attacker can manipulate an application's database queries by injecting malicious SQL code into input fields, resulting in unauthorized data access or modification.

Key Points:

Injection Point: SQLi typically occurs when user input is directly incorporated into SQL queries without proper sanitization.
Types of SQLi: There are different types of SQLi, including Classic SQLi, Blind SQLi, and Union-based SQLi.
Impact: SQLi can lead to unauthorized access to sensitive data (e.g., usernames, passwords), deletion of data, or even total system compromise.
Prevention: Use parameterized queries (prepared statements), ORM frameworks, and input validation to mitigate the risk.

**2. LDAP Injection Vulnerability**

Definition:
LDAP Injection is an attack that targets applications that construct LDAP queries based on user input. Attackers can manipulate queries to bypass authentication or extract sensitive information from an LDAP directory.

Key Points:

Injection Point: Occurs when user input is not properly sanitized before being inserted into an LDAP query.
Impact: An attacker can bypass authentication, retrieve unauthorized data from the directory, or modify directory content.
Exploitation: Attackers typically inject special characters (like * or |) into input fields to manipulate LDAP queries.
Prevention: Use parameterized LDAP queries, properly escape user input, and validate all incoming data.

**3. XPath Injection Vulnerability**

Definition:

XPath Injection allows attackers to inject malicious XPath expressions into queries that are used to retrieve data from XML documents. This can lead to unauthorized data access or manipulation.

Key Points:

Injection Point: Occurs when user input is directly incorporated into an XPath query without validation or sanitization.
Impact: Can lead to data leakage, unauthorized access to sensitive XML-stored data, or bypassing authentication.
Exploitation: By injecting crafted XPath expressions, attackers can modify the logic of queries.
Prevention: Use parameterized XPath queries and properly validate/escape all user input.

**4. Cross-Site Scripting (XSS) Vulnerability**
Definition:
XSS allows attackers to inject malicious scripts into web pages viewed by other users. These scripts execute in the victim's browser, often leading to data theft or session hijacking.

Key Points:

Types: XSS is divided into Stored, Reflected, and DOM-based XSS.
Impact: Can result in session hijacking, redirection to malicious websites, and stealing of sensitive data.
Exploitation: Attackers inject malicious code into input fields, URLs, or even cookies, which is then executed in another user's browser.
Prevention: Sanitize inputs, encode outputs, use Content Security Policy (CSP), and implement secure cookie attributes (HttpOnly, Secure).

**5. OS Command Injection Vulnerability**
Definition:
OS Command Injection occurs when an attacker can execute arbitrary system-level commands on the host operating system via a vulnerable application.

Key Points:

Injection Point: Occurs when user input is used to construct system commands without proper sanitization.
Impact: Allows attackers to execute commands with the same privileges as the application, potentially leading to total system compromise.
Exploitation: Attackers typically inject special characters like ; or && to chain their own commands with the original command.
Prevention: Validate and sanitize inputs, avoid direct system command execution, and use secure API functions instead.

**6. Local File Inclusion (LFI) / Remote File Inclusion (RFI)**
Definition:
LFI and RFI are vulnerabilities that allow attackers to include unauthorized files in the execution context of a web application, either from the local system (LFI) or remotely (RFI).

Key Points:

LFI (Local File Inclusion): Attackers can exploit LFI to read sensitive files on the server (e.g., /etc/passwd).
RFI (Remote File Inclusion): RFI allows attackers to include external files, often resulting in the execution of remote code.
Impact: Can lead to data leakage, unauthorized file access, and remote code execution.
Prevention: Use whitelisting to limit file inclusions, sanitize user inputs, and disable dangerous PHP functions (e.g., allow_url_include).

**7. Unvalidated File Upload Vulnerability**
Definition:
Unvalidated file upload vulnerabilities occur when an application allows users to upload files without properly checking their contents, file types, or security.

Key Points:

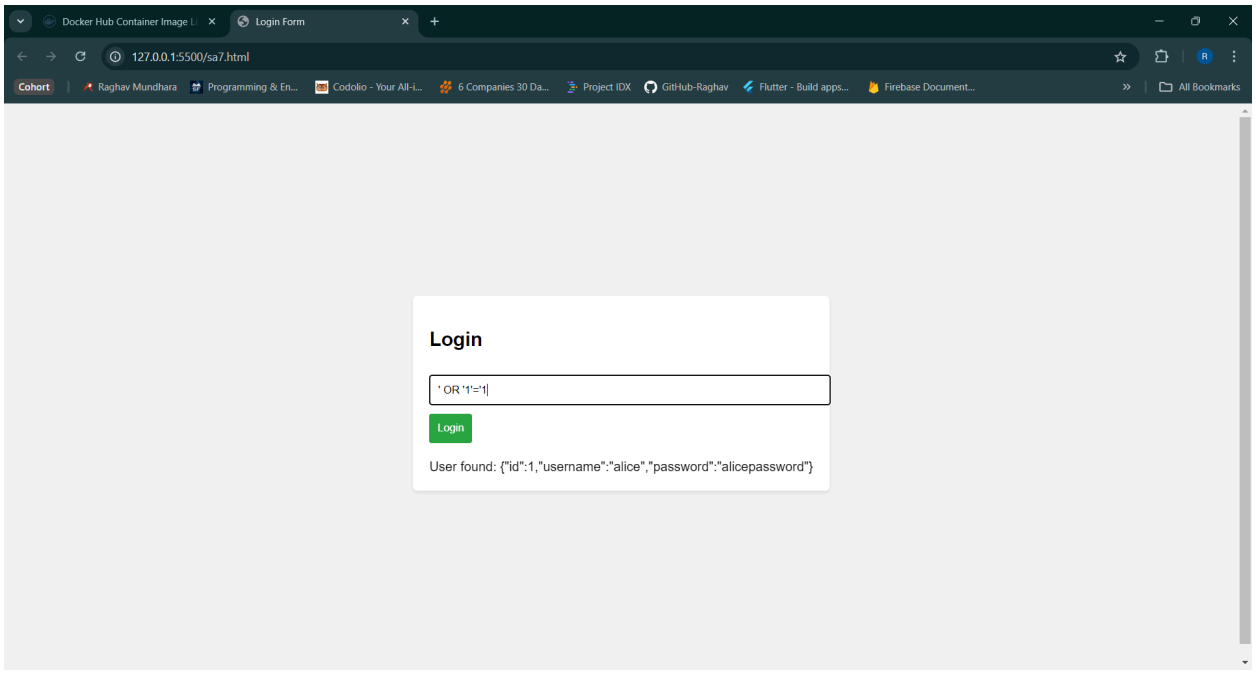
Exploitation: Attackers can upload malicious files (e.g., PHP shells, malware) that the server processes or executes.
Impact: Can lead to remote code execution, malware injection, or server compromise.
File Type Validation: Always check the file types and limit uploads to trusted formats (e.g., images, PDFs).
Prevention: Use proper file validation, set file size limits, restrict file permissions on the server, and store uploaded files outside the web root.

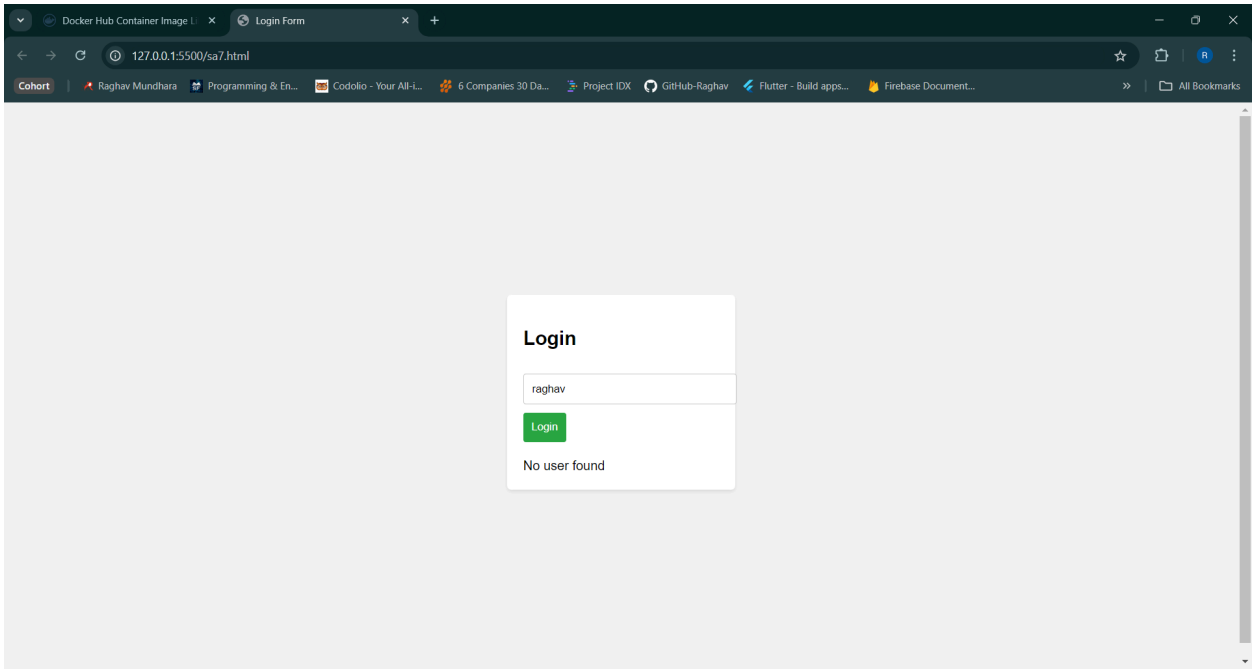
**8. Buffer Overflow Vulnerability**
Definition:
Buffer overflow occurs when an application writes more data to a buffer than it can hold, leading to data corruption, crashes, or arbitrary code execution.

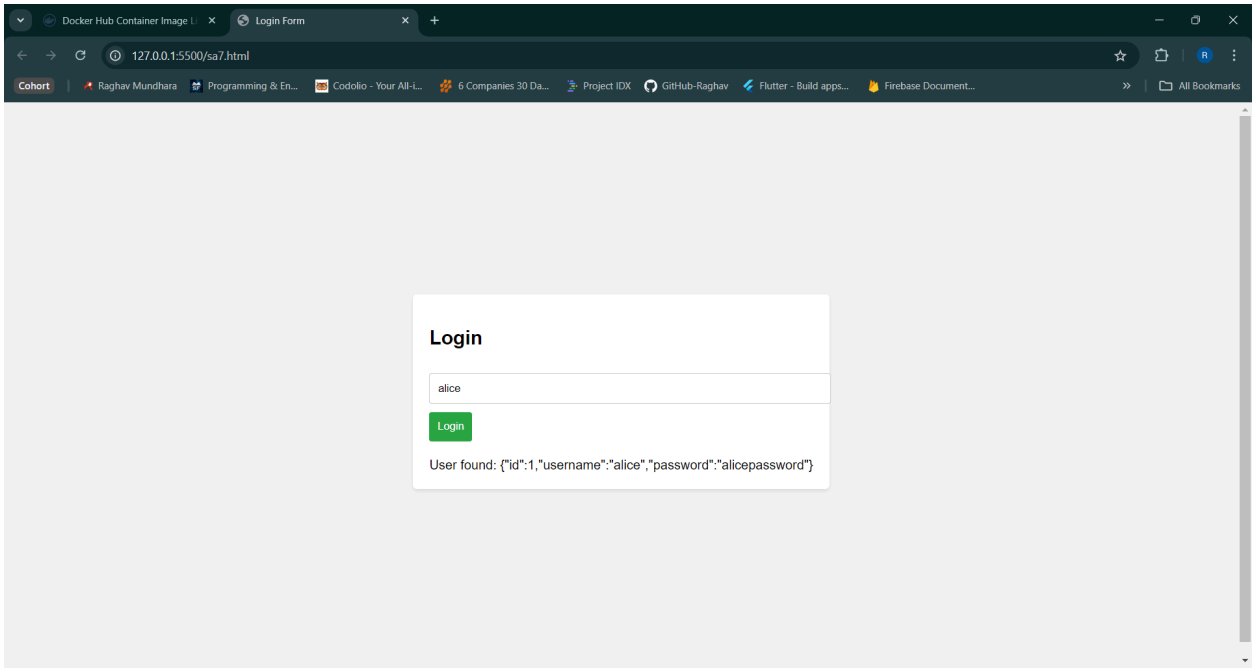
Key Points:

Exploitation: Attackers exploit buffer overflow by sending more data than expected, allowing them to overwrite adjacent memory and execute arbitrary code.
Impact: Can lead to system crashes, denial of service, or full remote code execution.
Stack vs. Heap Overflows: Buffer overflows can occur on the stack (stack-based overflow) or heap (heap-based overflow), depending on where the vulnerable buffer is located.

Prevention: Use modern programming languages that manage memory safely (e.g., Java, Python), implement buffer size checks, and enable security mechanisms like Address Space Layout Randomization (ASLR) and stack canaries.






SQL Commands:-

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(50) NOT NULL
);

INSERT INTO users (username, password) VALUES
('alice', 'alicepassword'),
('bob', 'bobpassword'),
('charlie', 'charliepassword'),
('dave', 'davepassword'),
('eve', 'evepassword'),
('frank', 'frankpassword'),
('grace', 'gracepassword'),
('heidi', 'heidipassword'),
('ivan', 'ivanpassword'),
('judy', 'judypassword');
```

Server.js

```
const express = require('express');
const cors = require('cors'); // Add this line
const { Client } = require('pg');

const app = express();
app.use(cors()); // Enable CORS
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

const client = new Client({
    user: 'postgres',
    host: 'localhost',
    database: '',
    password: 'password',
    port: 5432,
});

client.connect(err => {
    if (err) {
        console.error('Database connection failed:', err.stack);
    } else {
        console.log('Connected to PostgreSQL database.');
    }
});

app.post('/login', (req, res) => {
    const username = req.body.username;

    const sqlQuery = `SELECT * FROM users WHERE username = '${username}'`;

    client.query(sqlQuery, (error, results) => {
        if (error) {
            console.error(error);
            res.status(500).send('Internal server error');
        } else {
            if (results.rows.length > 0) {
                res.send(`User found:
${JSON.stringify(results.rows[0])}`);
            } else {
```

```
                res.send('No user found');
            }
        }
    });
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Form</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            background-color: #f4f4f4;
        }
        .login-container {
            background: white;
            padding: 20px;
            border-radius: 5px;
            box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
        }
        h2 {
            margin-bottom: 20px;
        }
        input {
```

```
            width: 100%;
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ccc;
            border-radius: 3px;
        }
        button {
            padding: 10px;
            background-color: #28a745;
            color: white;
            border: none;
            border-radius: 3px;
            cursor: pointer;
        }
        button:hover {
            background-color: #218838;
        }
        .response {
            margin-top: 20px;
        }
    </style>
</head>
<body>

<div class="login-container">
    <h2>Login</h2>
    <input type="text" id="username" placeholder="Enter your username"
required>
    <button id="loginBtn">Login</button>
    <div class="response" id="response"></div>
</div>

<script>
    document.getElementById('loginBtn').addEventListener('click',
function() {
        const username = document.getElementById('username').value;

        // Send POST request to the backend
        fetch('http://localhost:3000/login', {
            method: 'POST',
```

```
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ username }),
        })
        .then(response => response.text())
        .then(data => {
            document.getElementById('response').innerText = data;
        })
        .catch(error => {
            console.error('Error:', error);
            document.getElementById('response').innerText = 'An error
occurred. Please try again.';
        });
    });
</script>

</body>
</html>
```

**Conclusion :-** Hence, we implemented SQL injection and studied various vulnerabilities.