**Task 2: Introduction to Web Application Security**

**1. Objective and Methodology**

The primary objective of this task was to identify and analyze common web application vulnerabilities in a controlled environment. The target application used was **WebGoat** and the primary testing tool was **OWASP ZAP (Zed Attack Proxy)**.

**1.1 Methodology**

1. **Environment Setup:** A testing environment was configured using a virtual machine (VM) hosting the vulnerable web application (WebGoat) and a separate machine for the testing tool (OWASP ZAP).
2. **Proxy Configuration:** OWASP ZAP was configured as a local proxy (127.0.0.1:8081) to intercept all traffic between the testing machine's browser and the WebGoat application [cite: Local Server Proxy.png]. This allowed for manual traffic inspection and manipulation. The session had persisted for detailed analysis [cite: Persist ZAP Session.png].
3. **Active Scanning:** The entire scope of the WebGoat application was subjected to an **Automated Scan** using ZAP [cite: OWASP ZAP about to scan WebGoat for discoverable pages.png]. This scan probed the application for common weaknesses, resulting in a detailed list of security alerts categorized by severity [cite: Automated Scan Complete.png].
4. **Vulnerability Triage and Manual Analysis:** The high-risk alerts identified by the scanner (specifically SQL Injection, Cross-Site Scripting, and Cross-Site Request Forgery) were manually analyzed and documented, using the ZAP interface for detailed inspection [cite: Alert Monitoring.png, CSRF vulnerability detected.png, SQLi Injection detected.png, XSS attack detected].

**2. Vulnerability Findings and Analysis**

Three critical vulnerabilities, aligned with the **OWASP Top 10** list, were successfully identified and analyzed.

**2.1 Finding 1: SQL Injection (SQLi)**

|  |  |  |  |
| --- | --- | --- | --- |
| **OWASP Top 10 Mapping** | **Vulnerability Description** | **Severity** | **Screenshot Evidence** |
| A01:2021 - Injection | This vulnerability occurs when user input is incorrectly handled in a database query, allowing an attacker to modify the query structure and execute malicious SQL commands. | High (Red Alert) | [cite: SQL.png, SQL Injection Analysis.png] |

**Discovery and Exploitation Process**

1. **Discovery:** The automated scan flagged a login page/data retrieval function that was directly concatenating user input into a SQL query (e.g., SELECT \* FROM users WHERE username = 'userInput').
2. **Exploitation:** By submitting a payload such as ' OR '1'='1 in the input field, the resulting query was structurally changed to: SELECT \* FROM users WHERE username = '' OR '1'='1'.
3. **Impact:** Since '1'='1' is always true, the database logic was bypassed, allowing the retrieval of all records or unauthorized login without requiring a valid password. This demonstrates a severe compromise of **Confidentiality** and **Integrity**.

**Mitigation Suggestions**

* **Primary Solution (Mandatory):** Use **Parameterized Queries (Prepared Statements)**. This ensures that the user input is always treated as data, never as executable code, preventing the query structure from being altered.
* **Secondary Solution:** Implement **Input Validation** to strictly check that user input conforms to the expected format (e.g., only alphanumeric characters for a username).

**2.2 Finding 2: Cross-Site Scripting (XSS)**

|  |  |  |  |
| --- | --- | --- | --- |
| **OWASP Top 10 Mapping** | **Vulnerability Description** | **Severity** | **Screenshot Evidence** |
| A03:2021 - Injection | XSS allows an attacker to inject client-side scripts into web pages viewed by other users. This typically occurs in fields where user input is rendered on the page without proper sanitation. | Medium (Yellow Alert) | [cite: XSS Analysis.png] |

**Discovery and Exploitation Process**

1. **Discovery:** ZAP identified fields (e.g., comment section, search bar) that reflected input back to the user's browser without adequate encoding.
2. **Exploitation:** A simple proof-of-concept payload, such as <script>alert('XSS by ZAP');</script>, was submitted. The browser executed the script, triggering a pop-up alert confirming the vulnerability.
3. **Impact:** Successful XSS exploitation can allow an attacker to steal session cookies (leading to **Session Hijacking**), capture keystrokes, or redirect the user to a malicious site. This severely compromises the **Confidentiality** of user data.

**Mitigation Suggestions**

* **Primary Solution (Mandatory):** Implement **Output Encoding**. Before rendering user-supplied data in the HTML, all special characters (e.g., <, >, ", ') must be converted to their corresponding HTML entities (e.g., < becomes &lt;).
* **Secondary Solution:** Use a strong, context-aware input validation and sanitization library on the server side to strip out known malicious tags and attributes.

**2.3 Finding 3: Cross-Site Request Forgery (CSRF)**

|  |  |  |  |
| --- | --- | --- | --- |
| **OWASP Top 10 Mapping** | **Vulnerability Description** | **Severity** | **Screenshot Evidence** |
| A07:2021 - Software and Data Integrity Failures (related to authentication) | CSRF tricks an authenticated user into submitting an involuntary request to a web application (e.g., changing their password, making a purchase) that the application trusts. | Low (Blue Alert) | [cite: CSRF Analysis.png] |

**Discovery and Exploitation Process**

1. **Discovery:** The ZAP scanner found an action endpoint (e.g., POST /update-profile) that did not include a unique, unpredictable token, meaning the request could be successfully forged from an external origin.
2. **Exploitation (Conceptual):** An attacker crafts an external webpage containing a malicious form or image tag that automatically submits a valid (but unintended) request to the target application's endpoint when the victim, who is logged in, visits the attacker's page.
3. **Impact:** An attacker could force an authenticated victim to perform any action they are authorized to do, such as changing their email or password, leading to account compromise (severe **Integrity** violation).

**Mitigation Suggestions**

* **Primary Solution (Mandatory):** Implement **Anti-CSRF Tokens**. A unique, randomly generated token must be included in every state-changing request (POST, PUT, DELETE). The server must verify this token matches the one generated for the user's session.
* **Secondary Solution:** Implement the **SameSite** cookie attribute (set to Strict or Lax) to prevent browsers from sending cookies with cross-site requests, which often defeats CSRF attacks.

**3. Reflection and Next Steps**

This exercise provided hands-on experience in the critical first phase of a Penetration Test (Vulnerability Assessment). The use of ZAP proved highly effective for quickly identifying high-risk areas in the application, which were then validated manually.

The next planned task, **Task 3: Enhancing Professional Presence (LinkedIn)**, will focus on presenting these technical findings and skills effectively to a professional audience.