TASK 2- Web Application Security

REPORT BY: UJJWAL AHUJA

INTRODUCTION

In today's digital world, web applications are a primary target for cyberattacks. This task was designed to build foundational knowledge in web application security by exploring how attackers exploit common vulnerabilities and how these issues can be prevented. WebGoat, a deliberately insecure application developed by OWASP, was used to simulate real-world attack scenarios. OWASP ZAP, a powerful security testing tool, was used to analyse WebGoat and discover security flaws such as SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). This hands-on task allowed for both automated vulnerability scanning and manual testing, providing a well-rounded introduction to web application penetration testing and defence strategies.

TASK OBJECTIVE

The objective of this task was to explore and understand common web application vulnerabilities such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). By analyzing a vulnerable application (WebGoat) and using a security testing tool (OWASP ZAP), the goal was to identify, exploit, and understand these vulnerabilities from both an attacker's and a defender's perspective.

TOOLS AND METHODS USED

- **WebGoat**: A deliberately insecure web application provided by OWASP, used to simulate common security flaws.
- **OWASP ZAP (Zed Attack Proxy)**: A free, open-source security scanner used to detect vulnerabilities automatically and analyze traffic between client and server.
- **Mozilla Firefox**: Configured to route its traffic through ZAP for inspection.
- **Manual testing techniques** for input manipulation and payload insertion.

Steps for Installation

1. **Installing Webgoat**

- Install Java JDK from Oracle (required for WebGoat)

https://www.oracle.com/java/technologies/javase-downloads.html

- After installation, verify it:

  java -version

- Download WebGoat ***.jar*** from github:

  https://github.com/WebGoat/WebGoat/releases

- Open Command Prompt in the download directory:

  java -jar webgoat-2025.3.jar

- Then launch WebGoat at:

  http://localhost:8080/WebGoat

- Installation of OWASP ZAP (for vulnerability scanning)
- Downloading ZAP with default settings

  https://www.zaproxy.org/download/

- Configure ZAP and change the configuration of port to 8888

  As both works default on 8080 but using it already for WebGoat

2. Working

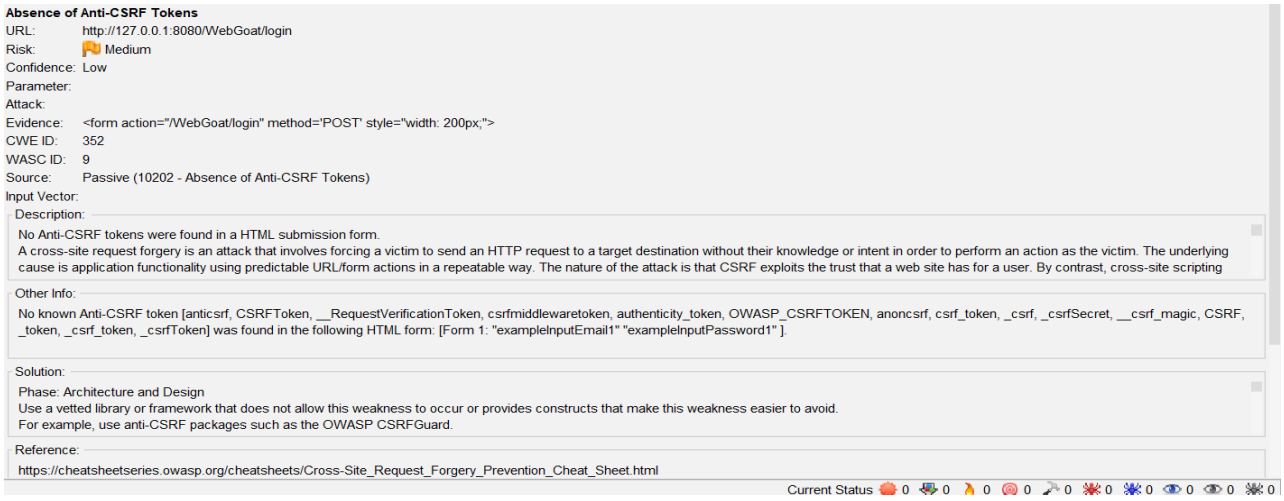
- In ZAP's Sites pane add address of WebGoat

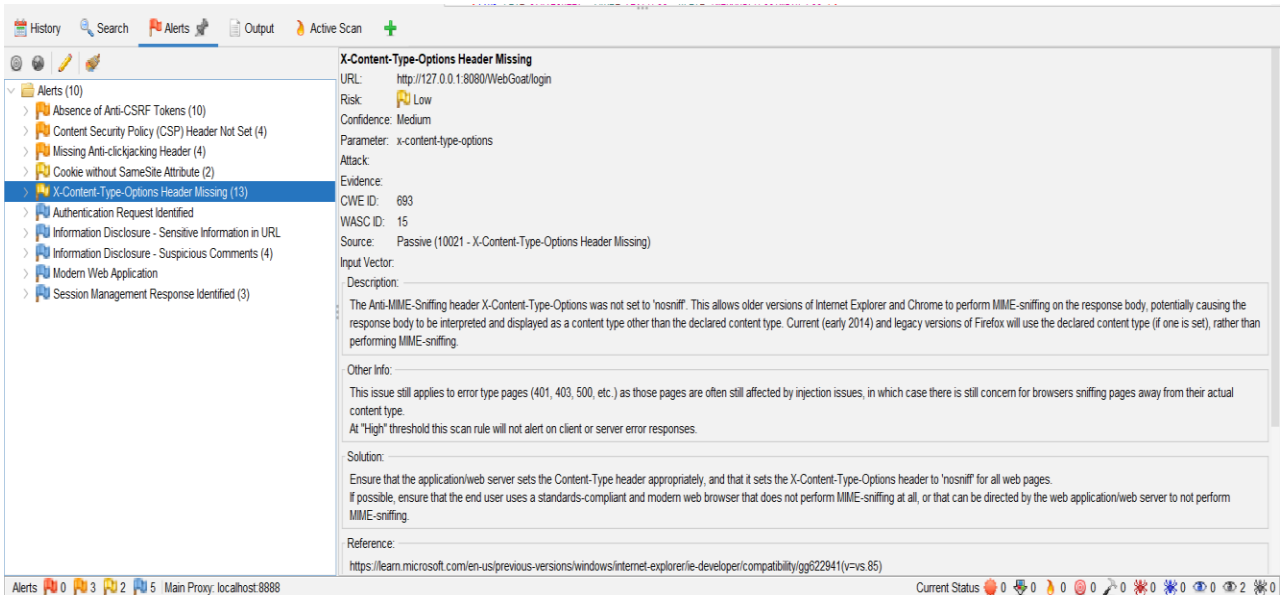
  http://localhost:8080

- Select **Active Scan** to detect vulnerabilities
- ZAP will report detected issues like:
  - SQL Injection
  - XSS
  - CSRF, etc.

3. Proof:


History
Search
Alerts
Output
Active Scan
Alerts (10)
Absence of Anti-CSRF Tokens (10)
Content Security Policy (CSP) Header Not Set (4)
Missing Anti-clickjacking Header (4)
Cookie without SameSite Attribute (2)
X-Content-Type-Options Header Missing (13)
Authentication Request Identified
Information Disclosure - Sensitive Information in URL
Information Disclosure - Suspicious Comments (4)
Modern Web Application
Session Management Response Identified (3)
Attack:
Evidence: <form action="/WebGoat/login" method="POST" style="width: 200px;">
CWE ID: 352
WASC ID: 9
Source: Passive (10202 - Absence of Anti-CSRF Tokens)
Input Vector:
Description:
No Anti-CSRF tokens were found in a HTML submission form.
A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting
Other Info:
No known Anti-CSRF token [anticsrf, CSRFToken, __RequestVerificationToken, csrfmiddlewaretoken, authenticity_token, OWASP_CSRFTOKEN, anoncsrf, csrf_token, _csrf, _csrfSecret, __csrf_magic, CSRF, _token, _csrf_token, _csrfToken] was found in the following HTML form: [Form 1: "exampleInputEmail1" "exampleInputPassword1"].
Solution:
Phase: Architecture and Design
Alerts 0 3 2 5 Main Proxy: localhost:8888
Current Status 0 0 0 0 0 0 0 0 0 0 0


Absence of Anti-CSRF Tokens
URL: http://127.0.0.1:8080/WebGoat/login
Risk: Medium
Confidence: Low
Parameter:
Attack:
Evidence: <form action="/WebGoat/login" method='POST' style="width: 200px;">
CWE ID: 352
WASC ID: 9
Source: Passive (10202 - Absence of Anti-CSRF Tokens)
Input Vector:
Description:
No Anti-CSRF tokens were found in a HTML submission form.
A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting
Other Info:
No known Anti-CSRF token [anticsrf, CSRFToken, __RequestVerificationToken, csrfmiddlewaretoken, authenticity_token, OWASP_CSRFTOKEN, anoncsrf, csrf_token, _csrf, _csrfSecret, __csrf_magic, CSRF, _token, _csrf_token, _csrfToken] was found in the following HTML form: [Form 1: "exampleInputEmail1" "exampleInputPassword1"].
Solution:
Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, use anti-CSRF packages such as the OWASP CSRFGuard.
Reference:
https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
Current Status 0 0 0 0 0 0 0 0 0 0 0


History
Search
Alerts
Output
Active Scan
Alerts (10)
Absence of Anti-CSRF Tokens (10)
Content Security Policy (CSP) Header Not Set (4)
Missing Anti-clickjacking Header (4)
Cookie without SameSite Attribute (2)
X-Content-Type-Options Header Missing (13)
Authentication Request Identified
Information Disclosure - Sensitive Information in URL
Information Disclosure - Suspicious Comments (4)
Modern Web Application
Session Management Response Identified (3)
X-Content-Type-Options Header Missing
URL: http://127.0.0.1:8080/WebGoat/login
Risk: Low
Confidence: Medium
Parameter: x-content-type-options
Attack:
Evidence:
CWE ID: 693
WASC ID: 15
Source: Passive (10021 - X-Content-Type-Options Header Missing)
Input Vector:
Description:
The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy versions of Firefox will use the declared content type (if one is set), rather than performing MIME-sniffing.
Other Info:
This issue still applies to error type pages (401, 403, 500, etc.) as those pages are often still affected by injection issues, in which case there is still concern for browsers sniffing pages away from their actual content type.
At "High" threshold this scan rule will not alert on client or server error responses.
Solution:
Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages.
If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform MIME-sniffing at all, or that can be directed by the web application/web server to not perform MIME-sniffing.
Reference:
https://learn.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/compatibility/gg622941(v=vs.85)
Alerts 0 3 2 5 Main Proxy: localhost:8888
Current Status 0 0 0 0 0 0 0 0 0 0 2 0

FOCUS AREAS FOR VULNERABILITY IDENTIFICATION

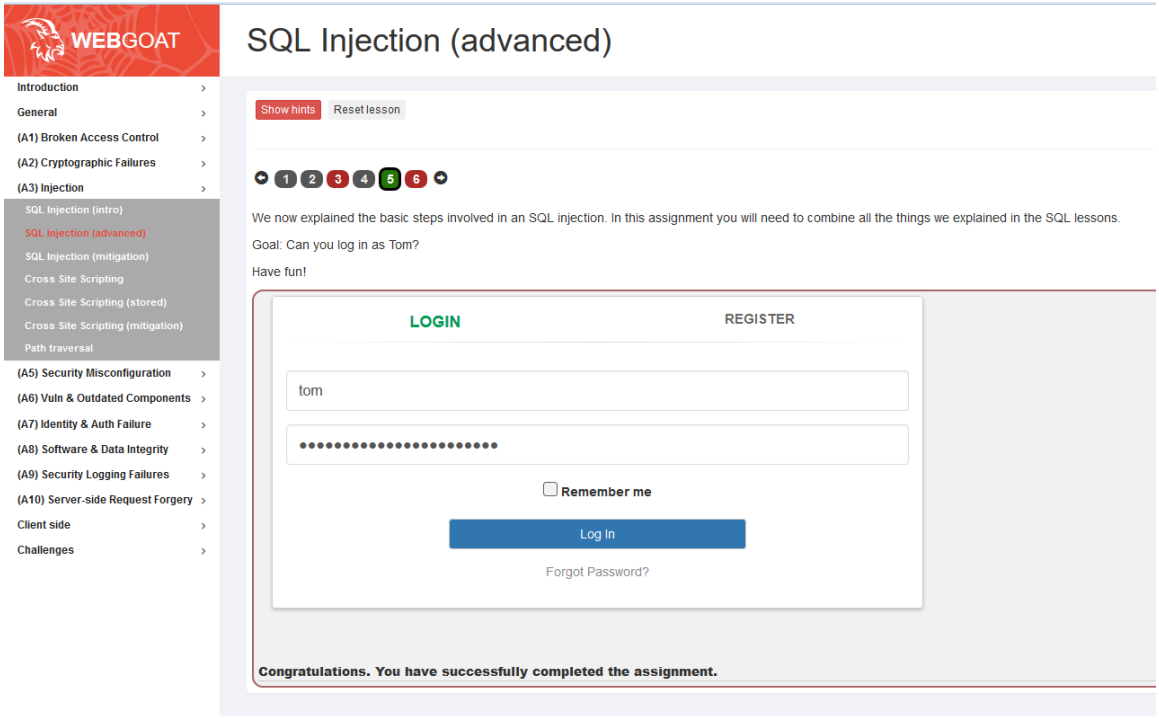
- **SQL Injection (SQLi):**
  Deliberately attempt to inject malicious SQL statements — for example, by submitting inputs like ' OR '1'='1 in login forms or search fields. This helps determine if the application improperly handles user-supplied data in database queries, potentially allowing unauthorized access or data leakage.

- **Cross-Site Scripting (XSS):**
  Assess input fields for inadequate output encoding by submitting harmless JavaScript payloads, such as <script>alert(1)</script>. Observe whether these scripts execute in the browser, indicating that the application fails to neutralize untrusted input.

- **Cross-Site Request Forgery (CSRF):**
  Examine forms and state-changing requests to identify the absence of protective anti-CSRF mechanisms (such as unique tokens). This vulnerability could allow malicious sites to perform unauthorized actions on behalf of an authenticated user.

MANNUAL EXPLOITATION

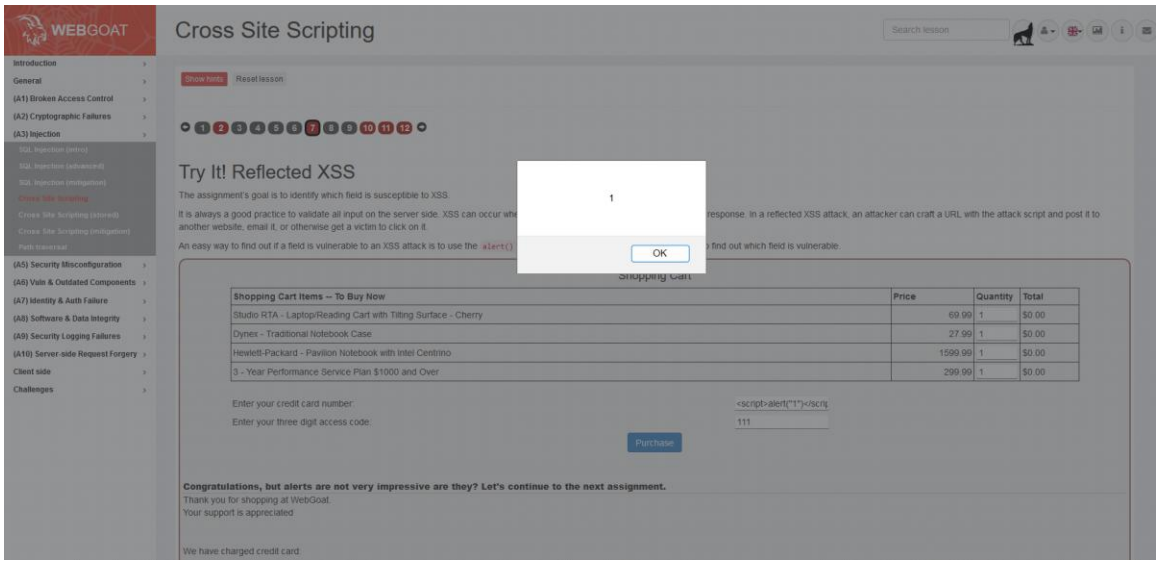
1. SQL Injection (SQLi)
    - How it's done:
        - Test input fields (login forms, search bars) with special characters like ' OR '1'='1.
        - Use simple payloads to bypass login or retrieve extra data.
        - Experiment with UNION SELECT to extract database tables and columns.
    - Goal:
        - Access, modify, or leak database information without permission.

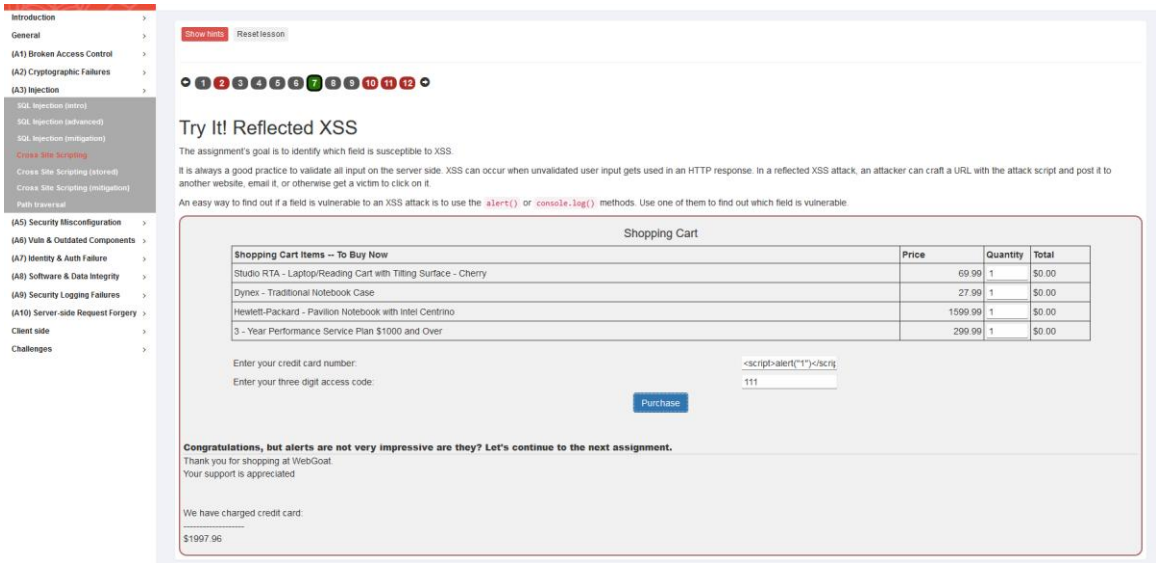


USERNAME: tom

PASSWORD: thisisasecretfortomonly

2. Cross-Site Scripting (XSS)
    - How it's done:
        - Enter harmless HTML or JavaScript in input fields (e.g., <script>alert(1)</script>).
        - See if the script runs in the browser.
        - Try different variations to bypass filters.
    - Goal:
        - Run malicious scripts in other users' browsers to steal cookies or hijack sessions.





Credit card number: <script>alert("1")</script>

3. Cross-Site Request Forgery (CSRF)
    - How it's done:
        - Identify forms or actions that change data (e.g., change password) without CSRF tokens.
        - Create a fake HTML form that auto-submits using the victim's session.
        - Send the malicious link to the victim.
    - Goal:

- Trick users into performing unwanted actions without their knowledge.

VULNERABILITY EXPLORATION AND VERIFICATION

To gain a strong understanding of web application security, I thoroughly examined each vulnerability detected during the assessment using both **OWASP ZAP** and **manual testing** methods. I reviewed the technical details and severity information provided by ZAP, and supported this with external research to better understand how these vulnerabilities are exploited in real-world attacks and the impact they can have.

Comprehensive Understanding of Vulnerabilities

Each vulnerability flagged by ZAP included useful details such as the affected request, description, risk level, and suggestions for mitigation. I used these insights to better understand the root cause of the issue. Additionally, I studied real-life case studies where similar vulnerabilities were exploited, which helped me connect theory with practical implications in cybersecurity.

Manual Testing and Exploitation

In addition to automated scanning, I performed **controlled manual testing** to validate the vulnerabilities and simulate how attackers might exploit them:

SQL Injection (SQLi)

I tested input fields such as login forms by inserting specially crafted SQL commands. These inputs were designed to bypass login checks, extract hidden data, or interfere with database queries. For example, I used inputs like ' OR '1'='1 or tom'-- to test if the application executed raw SQL without proper validation or parameterization.

Cross-Site Scripting (XSS)

I tested various input fields by entering harmless script tags like <script>alert('XSS')</script>. If the browser executed the script, it confirmed the presence of an XSS vulnerability. This showed that the application failed to sanitize user input, which in a real scenario, could allow attackers to steal session cookies, inject malicious code, or redirect users to harmful websites.

Cross-Site Request Forgery (CSRF)

I created a simple HTML form that automatically sent a POST request to a WebGoat endpoint. When the user was already logged in, the form was able to perform an action without their knowledge or consent, demonstrating a CSRF vulnerability. This test confirmed that the server was not validating the origin of requests or enforcing CSRF protection mechanisms like anti-CSRF tokens.

MITIGATION MEASURES FOR COMMON WEB VULNERABILITIES

1. Preventing SQL injection
    - Use prepared statements or parameterized queries instead of dynamic SQL.
    - Validate and sanitize all user inputs.
    - Use the principle of least privilege for database accounts.
2. Preventing Cross-Site Scripting (XSS)
    - Escape or encode user-generated content before displaying it.
    - Implement input validation and sanitization.
    - Use Content Security Policy (CSP) to limit script execution.
3. Preventing Cross-Site Request Forgery (CSRF)
    - Implement anti-CSRF tokens for forms and state-changing requests.
    - Use SameSite cookie attributes.
    - Require re-authentication for sensitive actions.

CONCLUSION

Task 2 was a hands-on and eye-opening experience that helped me understand how real-world web vulnerabilities work. Using tools like WebGoat and OWASP ZAP, I didn't just read about common attacks like SQL Injection, XSS, and CSRF I actually got to simulate and exploit them in a safe environment.

Going through these lessons showed me how small coding mistakes can lead to serious security risks, and how attackers can take advantage of them to gain access or control. At the same time, I also learned how simple security practices like validating inputs, escaping outputs, and using CSRF tokens can go a long way in protecting applications.

This task really helped me think from both a developer's and an attacker's point of view. It strengthened my understanding of ethical hacking and made me more aware of the importance of secure development in every stage of a project. Overall, it was a valuable learning experience that gave me both confidence and curiosity to keep exploring cybersecurity further.

Report Submitted To: REDYNOX BY THE RED USER