ELYSIUM

Arelithuk

Web Application Security Assessment

Pentester : Daniel Arconada

Index

Confidentiality Agreement

Elysium and Arelithuk agree to conduct a security assessment engagement on the web application WebGoat.

The purpose of this activity is to analyze the application's security posture, identify potential vulnerabilities, and determine the level of exposure of its servers, services, and web components.

The engagement will take place between June 29th, 2025, and July 3rd, 2025.

Both parties agree that all information obtained during the process — including technical data, findings, and results — will be handled under strict confidentiality. No element derived from the assessment shall be shared with any third party external to the entities mentioned without prior written consent.

The results of the analysis will be used exclusively to enhance Arelithuk's security posture and shall not be disclosed outside the framework of collaboration established with Elysium.

Contact Details

Name	Contact	Role
Daniel Arconada	d.arconada@elysium.com	Pentester
Pedro Lopez	p.lopez@arelithuk.com	CISO

Introduction

A security testing engagement was conducted on the WebGoat web application, provided by Arelithuk, with the objective of assessing its level of exposure to potential threats.

The primary goal of this activity was to strengthen Arelithuk's security posture and mitigate operational risks that could result from a security incident.

During the evaluation, several vulnerabilities and weaknesses were identified which, in a real-world scenario, could be exploited to compromise the confidentiality, integrity, or availability of the system, or to introduce malicious code into the environment.

This report compiles the detected findings and presents a set of recommendations and corrective actions aimed at improving the security of the WebGoat application and strengthening Arelithuk's defensive capabilities against external threats.

Scope

For this security assessment, the provided web application was analyzed with the purpose of evaluating its security level.

The objective was to identify potential vulnerabilities that could compromise the confidentiality, integrity, or availability of the system and the data it manages.

Web Application	IP Address
WebGoat	127.0.0.1

Out-of-Scope Scenarios

The following tests were explicitly excluded from the scope of this assessment: Denial-of-Service (DoS/DDoS) testing and any social engineering techniques (phishing, vishing, smishing, or impersonation).

These limitations ensure that the evaluation focuses exclusively on the web application under analysis, avoiding any potential impact on business operations, personnel, or third parties unrelated to the testing environment.

Risk Classification

The following table presents the risk rating criteria used to classify the identified vulnerabilities based on their severity and potential impact.

Risk	CVSS Range	Description
High	8.1-10.0	Indicates a critical risk that is readily exploitable. Immediate remediation is strongly recommended upon discovery.
Medium	6.1-8.0	Represents a significant vulnerability that may be exploited under certain conditions. These issues should be addressed promptly after high-risk findings have been resolved.
Low	4.1-6.0	Identifies a moderate weakness that may be difficult to exploit. Remediation efforts are recommended within 90 days of detection.

Executive Summary

This document provides a summary of the security assessment conducted on the WebGoat web application.

The following points were analyzed:

- A3 Injection – SQL Injection
- A3 Injection – Cross-Site Scripting
- A5 Security Misconfiguration
- A6 Vulnerable & Outdated Components
- A7 Identification & Authentication Failures

The engagement was carried out using a greybox approach, as internal information about the application was available. However, in other cases, traffic interception tools were required to analyze how the application processes requests.

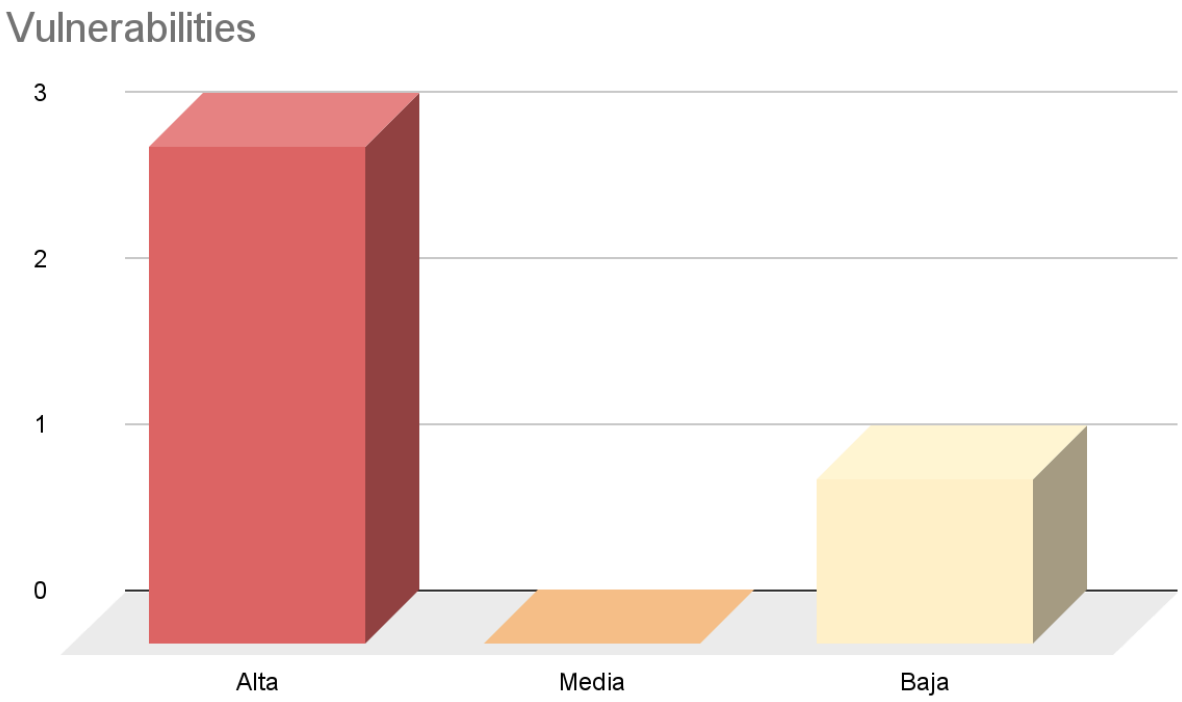
The most critical vulnerability identified was:

- SQL Injection

During testing, special emphasis was placed on vulnerabilities affecting the confidentiality, integrity, and availability of the application and its data.

Both manual techniques and tool-assisted attacks were used, leveraging tools such as *Burp Suite* and *Caido*.

Vulnerability Overview



Vulnerability Summary

ID Vulnerability Name	Recommendation	Impact	CVSS Rating
01 - SQL Injection	Implement strict input validation and use parameterized queries to prevent arbitrary execution of SQL commands.	Alta	10
02 - Reflected XSS	Sanitize user input and validate input fields on both client and server side.	Baja	4.3

03 - CSRF Forgery	Use anti-CSRF tokens, verify Origin/Referer headers, and configure secure cookies.	Alta	9.8
04 - XXE	Disable DTDs and external entities in XML parsers, update XML libraries, validate input using whitelists, and use JSON where possible.	Alta	9.2
05 - Insecure Passwords	Establish a secure password policy.	Informativa	1.3

Technical Analysis

The WebGoat web application, provided by Arelithuk, was analyzed within the defined scope.

During the evaluation, several high-severity vulnerabilities were identified that require immediate remediation, with SQL Injection being the most critical.

For the dynamic analysis, an HTTP/HTTPS interception proxy *Caido* was used to monitor and modify the requests sent to the application, allowing validation of attack vectors and confirmation of input validation flaws.

01 - SQL Injection

This vulnerability is located in A3 Injection - SQL Injection - Section 11.

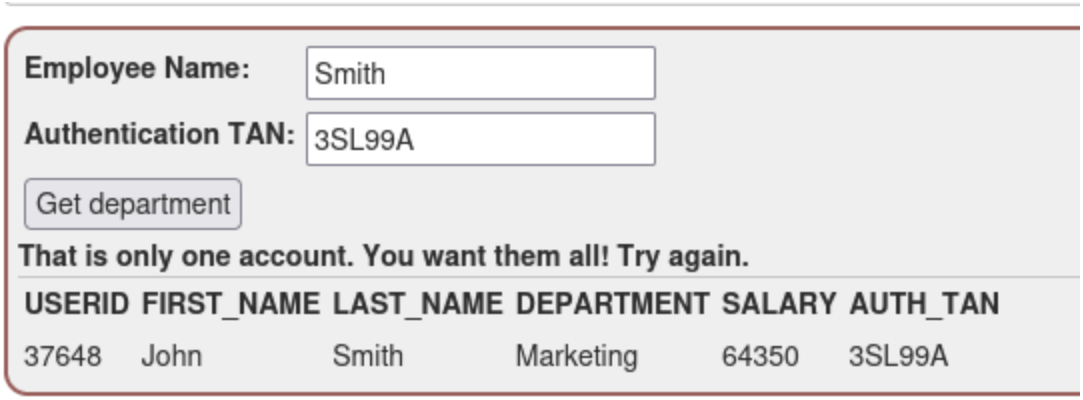
The WebGoat web application was assessed, and a vulnerability in database queries was identified.

Through this flaw, it is possible to expose the information stored in the database and even modify database objects such as tables, columns, data, and privileges.

During the assessment, it was possible to extract all company databases and alter tables related to access logs and query logging activities.

Information Provided:

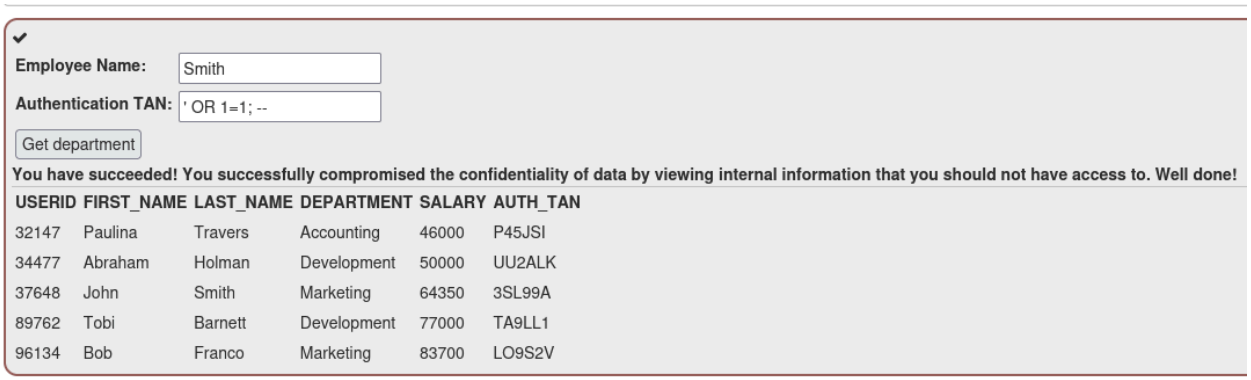
- Name: John Smith
- Authentication TAN: 3SL99A



Proof of Concept :

1. A reconnaissance of the web application was performed, and it was observed that certain input fields execute database queries without proper sanitization. For example, inserting a single quote (') generates a syntax error, indicating a lack of input validation.
2. In the TAN field, the following payload was injected:

' OR 1=1; --



This confirmed that the database was vulnerable, allowing potential compromise of its integrity and exposure of sensitive data.

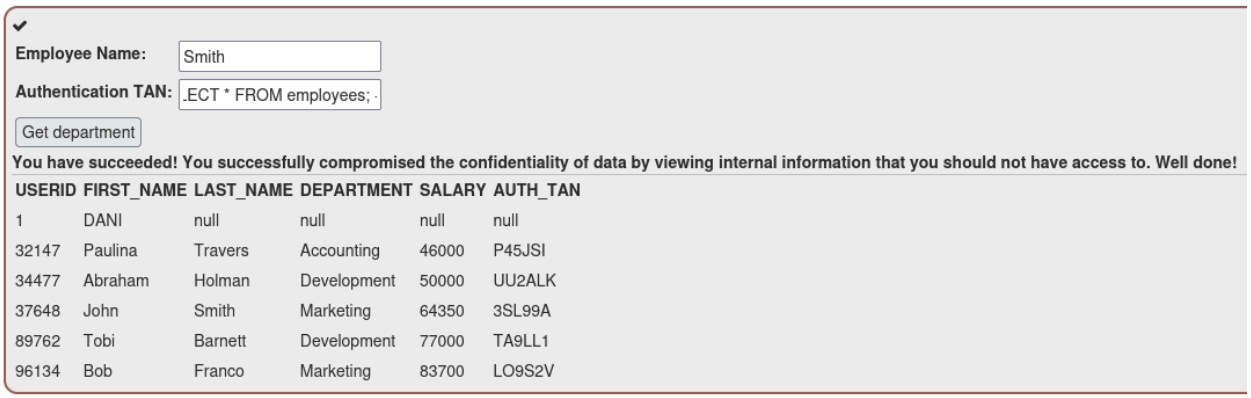
The following steps were carried out to demonstrate the potential for database manipulation. Once the injection was identified, additional queries were executed while progressively escalating privileges, yielding, among other things, the following results:

- Insert new entries into the employees table:

'; INSERT INTO employees (userid, first_name) VALUES (1,'DANI');--

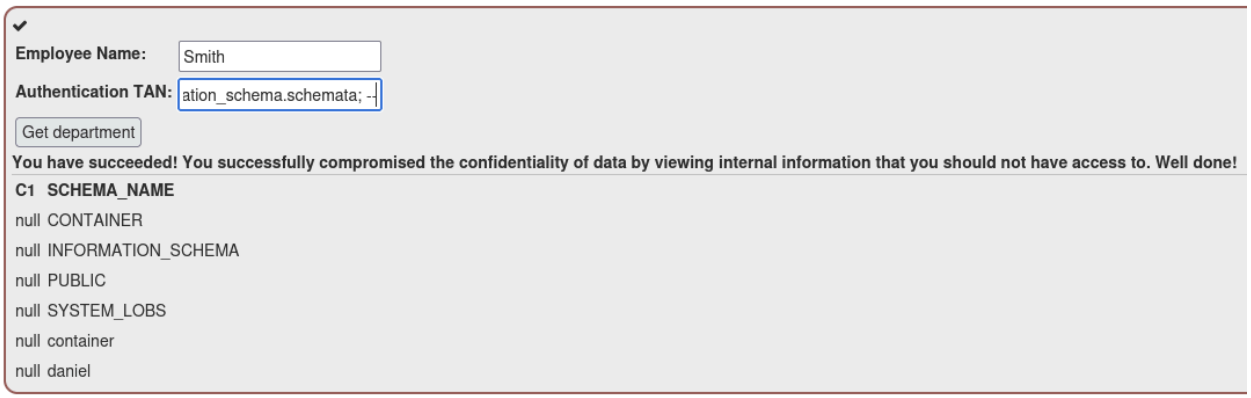
- Retrieve the full contents of the employees table:

*'; SELECT * FROM employees;-- -*



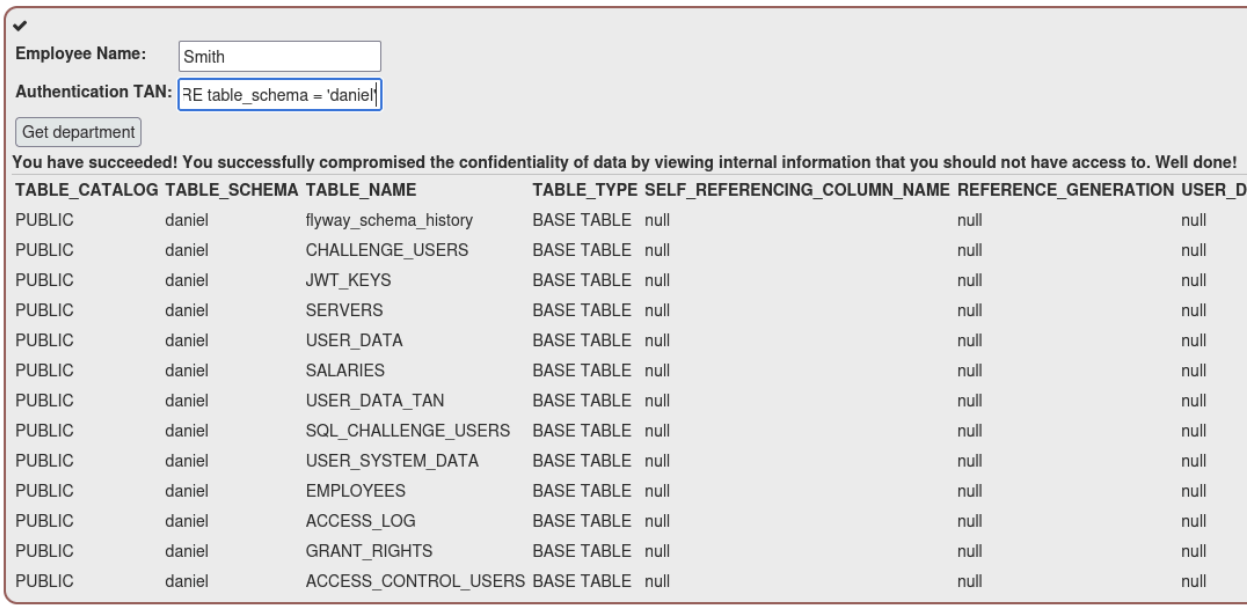
- Enumerate all databases:

' SELECT null, schema_name FROM information_schema.schemata;-- -



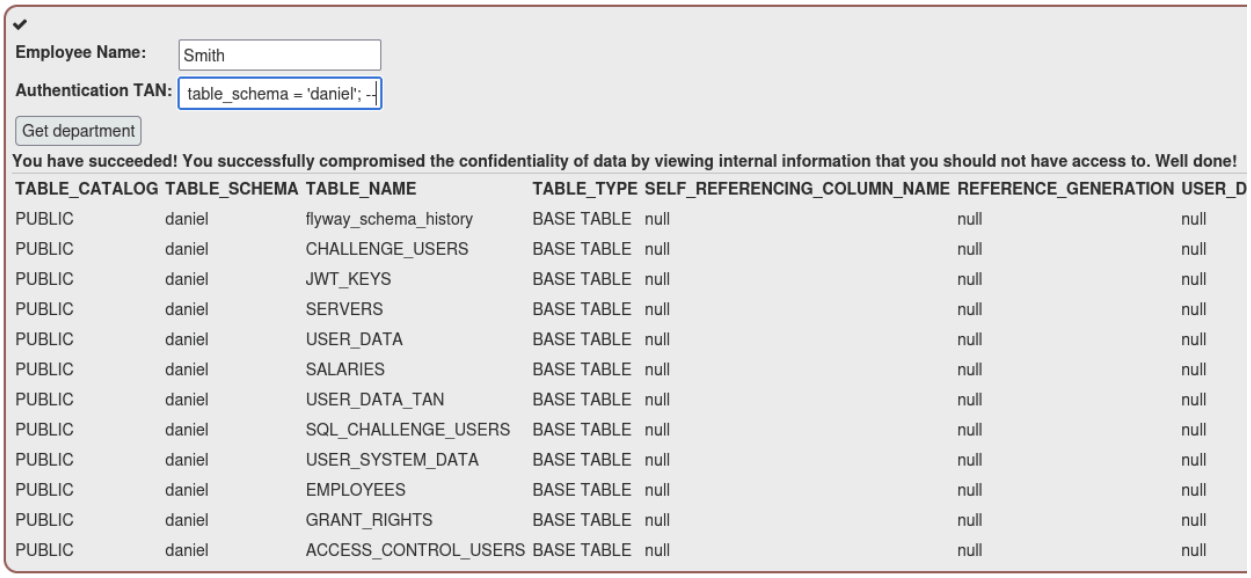
- List tables from a specific database:

*' SELECT * FROM information_schema.tables WHERE table_schema ='daniel';- -*



- Delete tables:

' DROP TABLE access_log;-- -



Employee Name: Smith

Authentication TAN: table_schema = 'daniel'; --

Get department

You have succeeded! You successfully compromised the confidentiality of data by viewing internal information that you should not have access to. Well done!

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE	SELF_REFERENCING_COLUMN_NAME	REFERENCE_GENERATION	USER_D
PUBLIC	daniel	flyway_schema_history	BASE TABLE	null	null	null
PUBLIC	daniel	CHALLENGE_USERS	BASE TABLE	null	null	null
PUBLIC	daniel	JWT_KEYS	BASE TABLE	null	null	null
PUBLIC	daniel	SERVERS	BASE TABLE	null	null	null
PUBLIC	daniel	USER_DATA	BASE TABLE	null	null	null
PUBLIC	daniel	SALARIES	BASE TABLE	null	null	null
PUBLIC	daniel	USER_DATA_TAN	BASE TABLE	null	null	null
PUBLIC	daniel	SQL_CHALLENGE_USERS	BASE TABLE	null	null	null
PUBLIC	daniel	USER_SYSTEM_DATA	BASE TABLE	null	null	null
PUBLIC	daniel	EMPLOYEES	BASE TABLE	null	null	null
PUBLIC	daniel	GRANT_RIGHTS	BASE TABLE	null	null	null
PUBLIC	daniel	ACCESS_CONTROL_USERS	BASE TABLE	null	null	null

After comparing with the database view (showing all tables under the schema daniel), it was confirmed that the access_logtable no longer exists.

Recommendation :

- Apply strict input sanitization and implement whitelist-based validation, allowing only expected values (input length, allowed characters, MIME types, and file extensions).
- Combine with contextual output encoding and proper error handling to prevent database errors or stack traces from being displayed.
- Operate the application under the principle of least privilege: the application's database account must only have the necessary permissions, without superuser or administrative rights. Use separate accounts for administration, application, and maintenance tasks.

References :

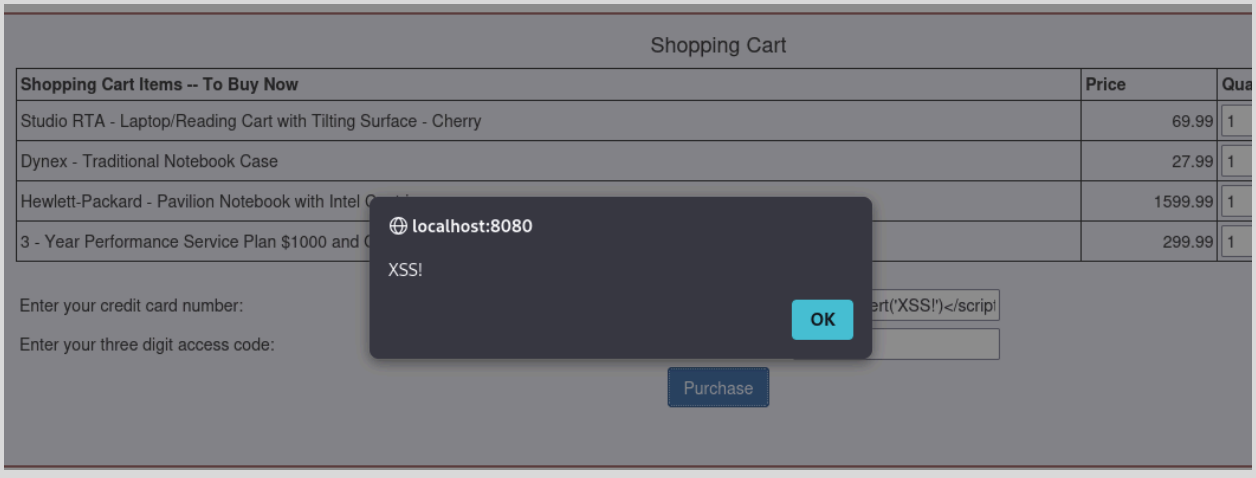
https://owasp.org/www-community/attacks/SQL_Injection

02 - Reflected XSS

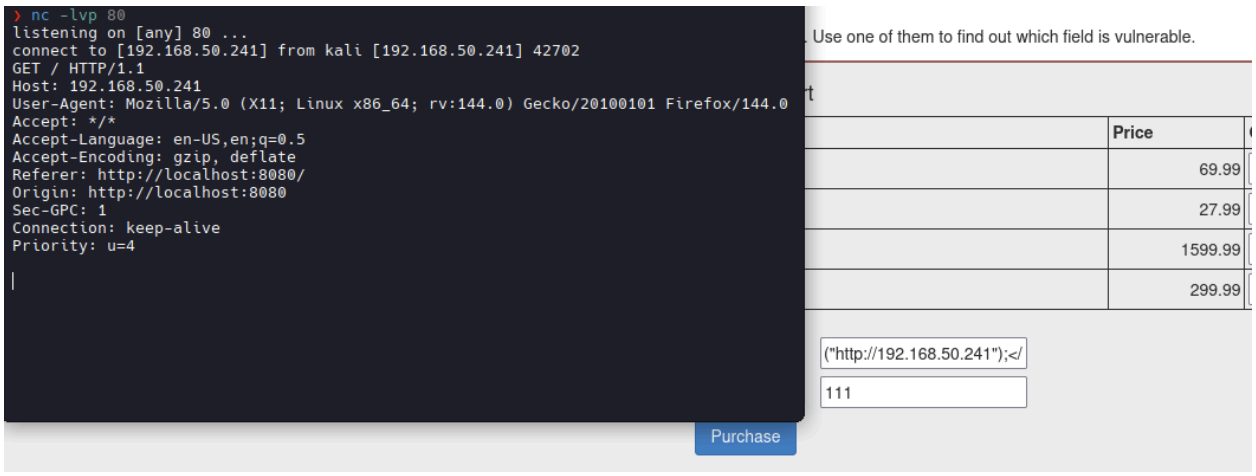
This vulnerability is located in A3 Injection - Cross-Site Scripting - Section 7.
The WebGoat web application was found to be vulnerable to Reflected Cross-Site Scripting (XSS).
This issue allows execution of malicious JavaScript code in the victim's browser.
The affected field corresponds to the credit card number input, which accepts HTML tags such as <script> and embedded JavaScript code.
As this is a Reflected XSS and no other related vulnerabilities were identified, its severity is considered low, since it does not allow server-side code execution.
Proof of Concept :

1. A reconnaissance phase was conducted to identify input fields controllable by the user.
2. A basic reflected XSS test was executed: while the access code field was not vulnerable, the credit card number field was. In this field, the HTML tag <script> could be injected, allowing JavaScript execution in the browser.
3. When a basic payload was tested, a positive response was observed, showing a popup message:

<script>alert('XSS!')</script>



A more advanced scenario was also tested — a redirection to a malicious page (e.g., a fake login page) designed to steal credentials.
Although this type of attack is possible, it executes only on the client side and does not imply any server-side execution.
To validate the redirection, a test request was sent to our IP, confirming the expected behavior.



Recommendation :

- Sanitize user input before processing or rendering it in the browser. Use frameworks such as DOMPurify to remove potentially malicious tags or content.
- Validate input both client-side and server-side, allowing only expected characters and formats for each field.
- Apply output encoding to all dynamic content inserted into the DOM or displayed in the browser, preventing interpretation as HTML or JavaScript.
- Limit the use of unsafe DOM functions such as innerHTML, document.write, or eval; replace them with secure alternatives.
- Implement security-related HTTP headers, such as Content-Security-Policy, to reduce the likelihood of script execution.

References :

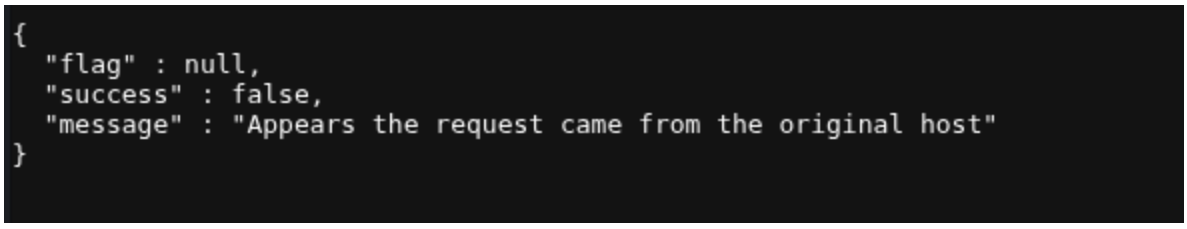
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Che

at_Sheet.html

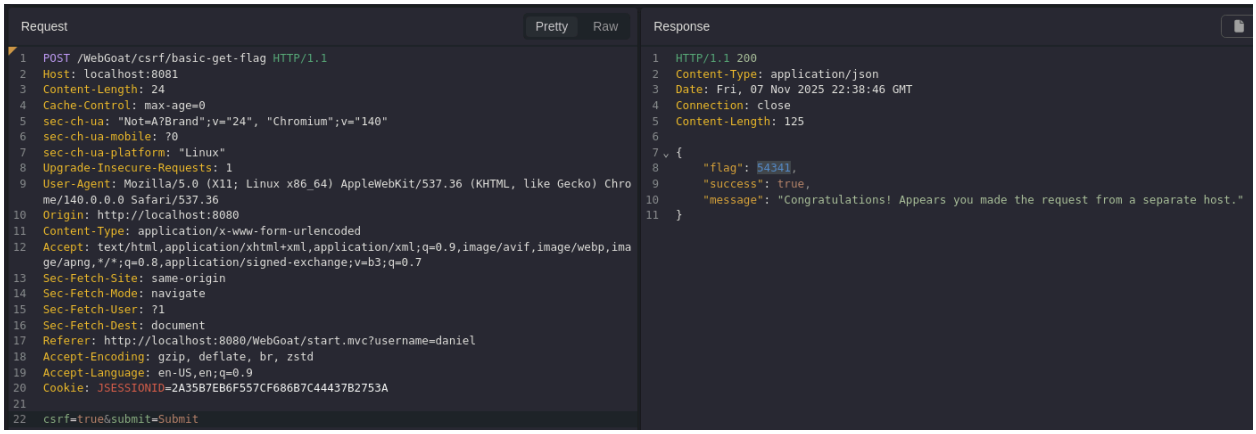
03 - CSRF Forgery

This vulnerability is located in A5 Security Misconfiguration — Cross-Site Request Forgery (CSRF), Sections 3 and 4 have been separated into parts A and B.
A. The link referenced in Section 3 was investigated to determine if the hidden flag could be extracted.
The response indicated that the request had to originate from a different host.
By modifying the request origin using *Caido*, the flag was successfully obtained.
B. The comment submission process was then analyzed.
When requests were sent from the same host, the action was denied. However, if the request appeared to come from an external host, the application allowed comment publication.
Caido was used as an interception proxy to manipulate HTTP headers and control request origins.
A.

1. Accessed the link to observe the response and analyze its behavior.

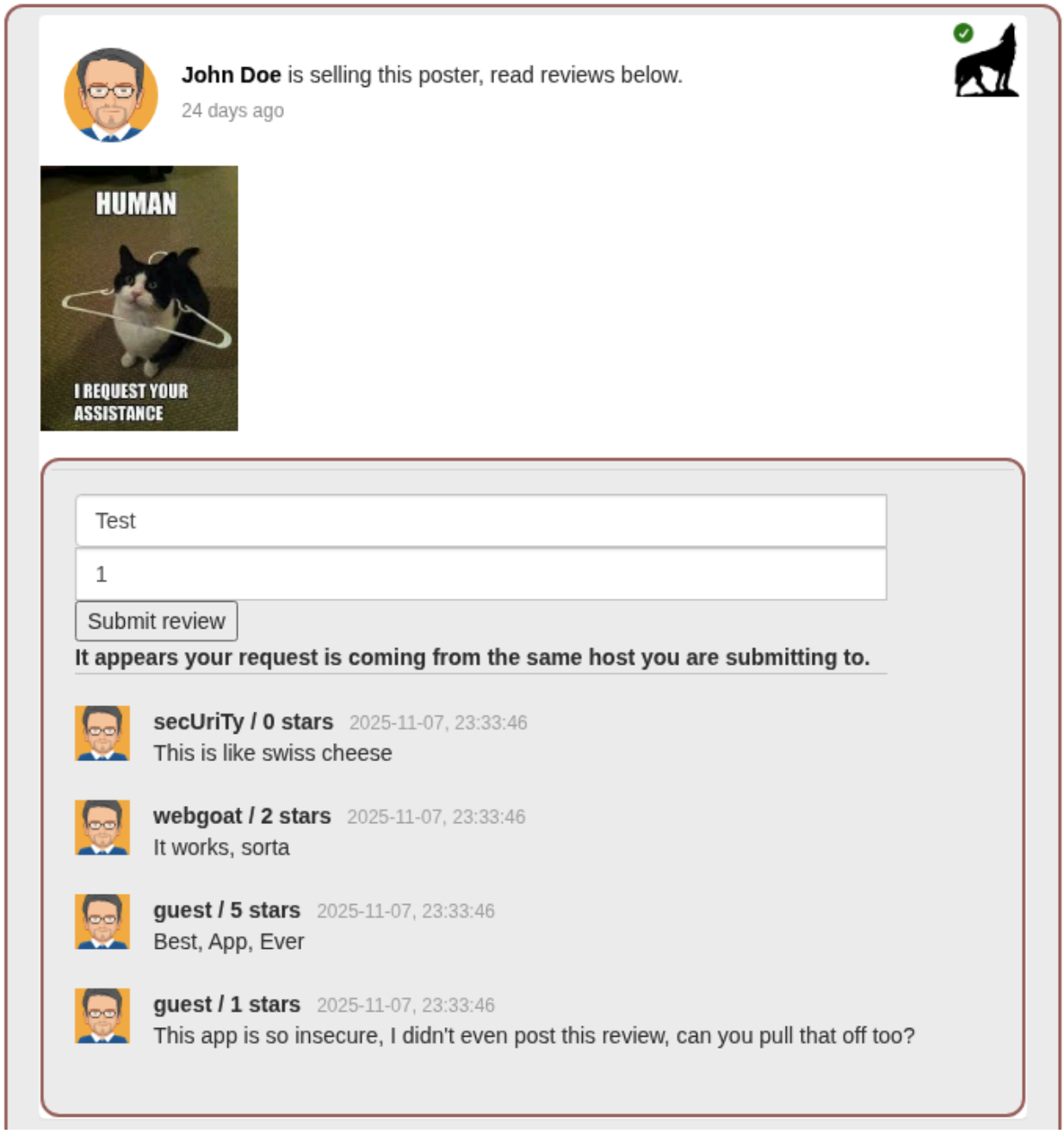


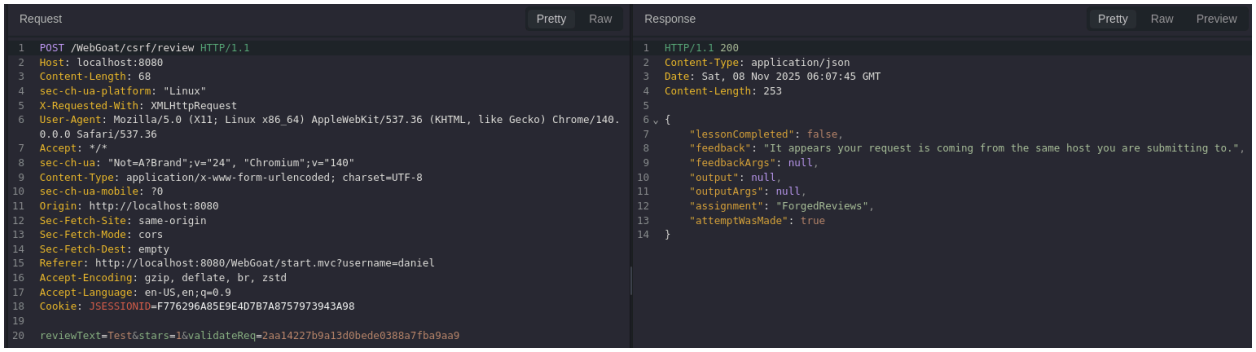
2. The response message indicated the request originated from the same host, so it was intercepted with Caido.
3. The host header was modified from http://127.0.0.1:8080 to http://127.0.0.1:8081. After this change, the request succeeded, as it appeared to come from a different host.

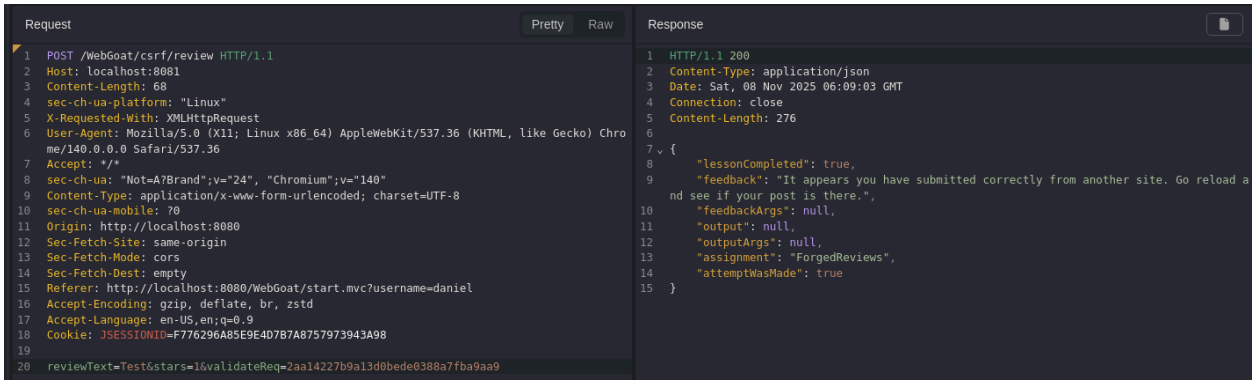


B.

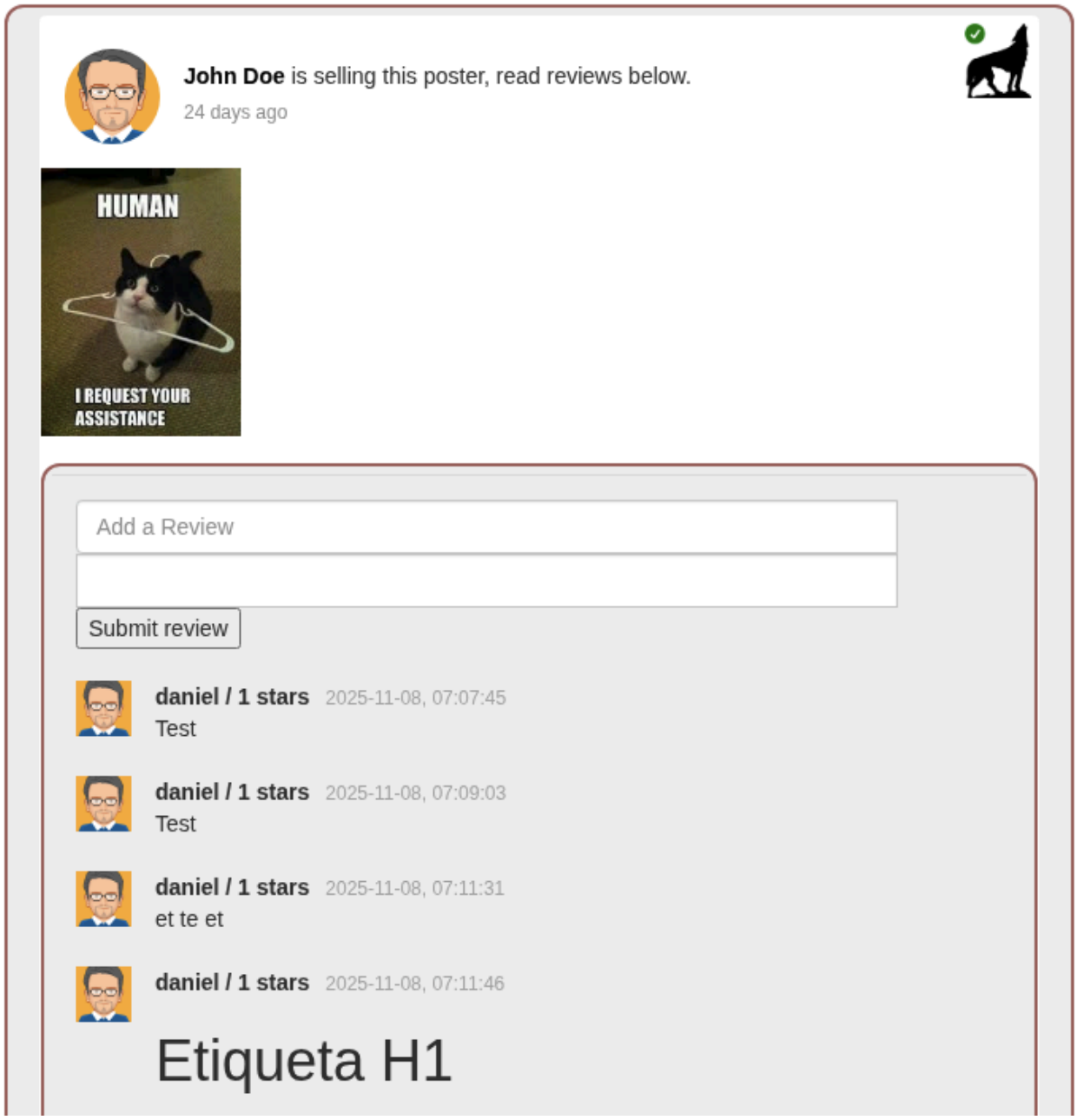
1. In Section 4, a comment form was identified.
2. Sending the request through *Caido* and modifying its origin allowed comments to be posted.


John Doe is selling this poster, read reviews below.
24 days ago
HUMAN
I REQUEST YOUR ASSISTANCE
Test
1
Submit review
It appears your request is coming from the same host you are submitting to.
secUriTy / 0 stars 2025-11-07, 23:33:46
This is like swiss cheese
webgoat / 2 stars 2025-11-07, 23:33:46
It works, sorta
guest / 5 stars 2025-11-07, 23:33:46
Best, App, Ever
guest / 1 stars 2025-11-07, 23:33:46
This app is so insecure, I didn't even post this review, can you pull that off too?

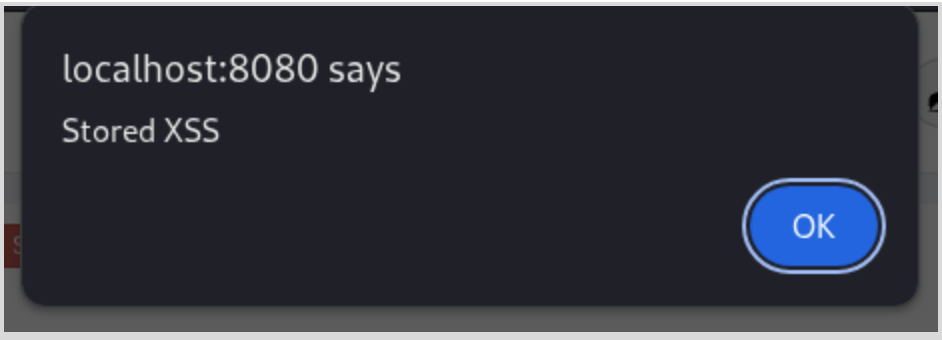




3. To test for XSS, a simple payload was injected: `<h1>Test</h1>` it rendered successfully.



4. A subsequent payload was tested to confirm script execution:
   <script>alert(stored XSS)</script>



The script was persistently stored, and the alert executed each time the page was reloaded, confirming a stored XSScondition.

In a real-world attack, a malicious link could be sent to a victim, redirecting to a fake login page to harvest credentials.

Recommendation :

- Implement anti-CSRF tokens. The server should generate two random tokens — one sent as a cookie and another as a hidden field in the form. Both must match when the request is received; otherwise, it should be rejected.
- Verify Origin and Referer headers for sensitive requests. Reject any requests coming from unauthorized domains.
- Configure secure cookies. Mark cookies as HttpOnly and Secure, and use the SameSite attribute to prevent cross-site transmission.
- Require reauthentication for critical actions and enforce least-privilege access controls.
- Mitigate XSS risks since CSRF and XSS combined can amplify impact. Apply

input validation and output encoding to prevent injection of malicious content.

References:

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

04 - XXE

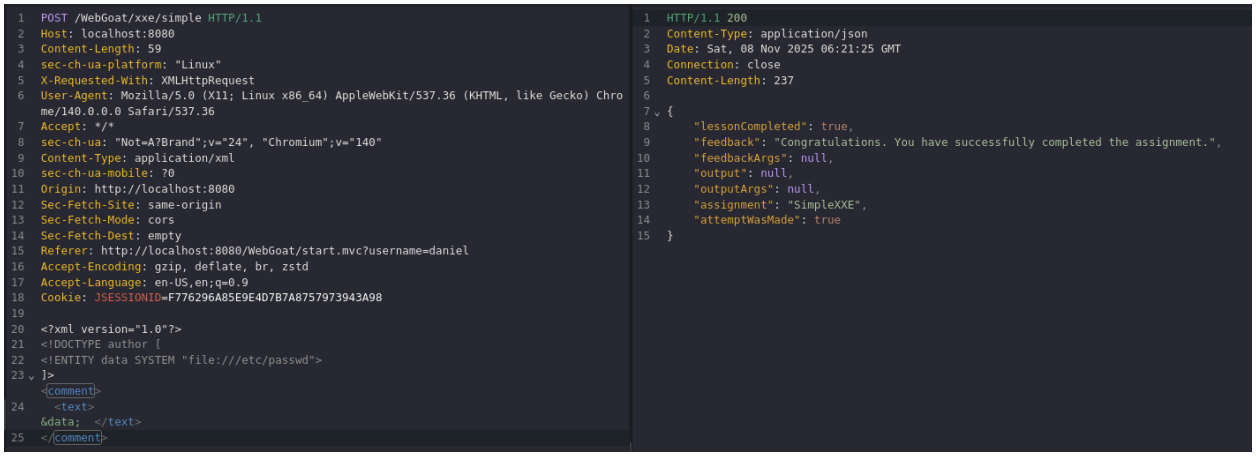
This vulnerability is located in A5 Security Misconfiguration - XXE - Section 4.

The WebGoat application's XML External Entity (XXE) module was tested, and a vulnerability was identified that allows injection of malicious external entities, enabling potential disclosure of sensitive system information due to insecure XML input processing.

To facilitate manipulation of requests and modification of HTTP headers during testing, *Caido* was used as an interception proxy.

Proof of Concept :

1. Requests were intercepted using *Caido*, and a basic comment submission was analyzed to observe how the XML data was processed.
2. The XML structure was modified to test for XXE vulnerability by injecting a basic malicious entity, which the application accepted.
3. An additional test was performed using an entity referencing a local file, confirming sensitive data exposure.

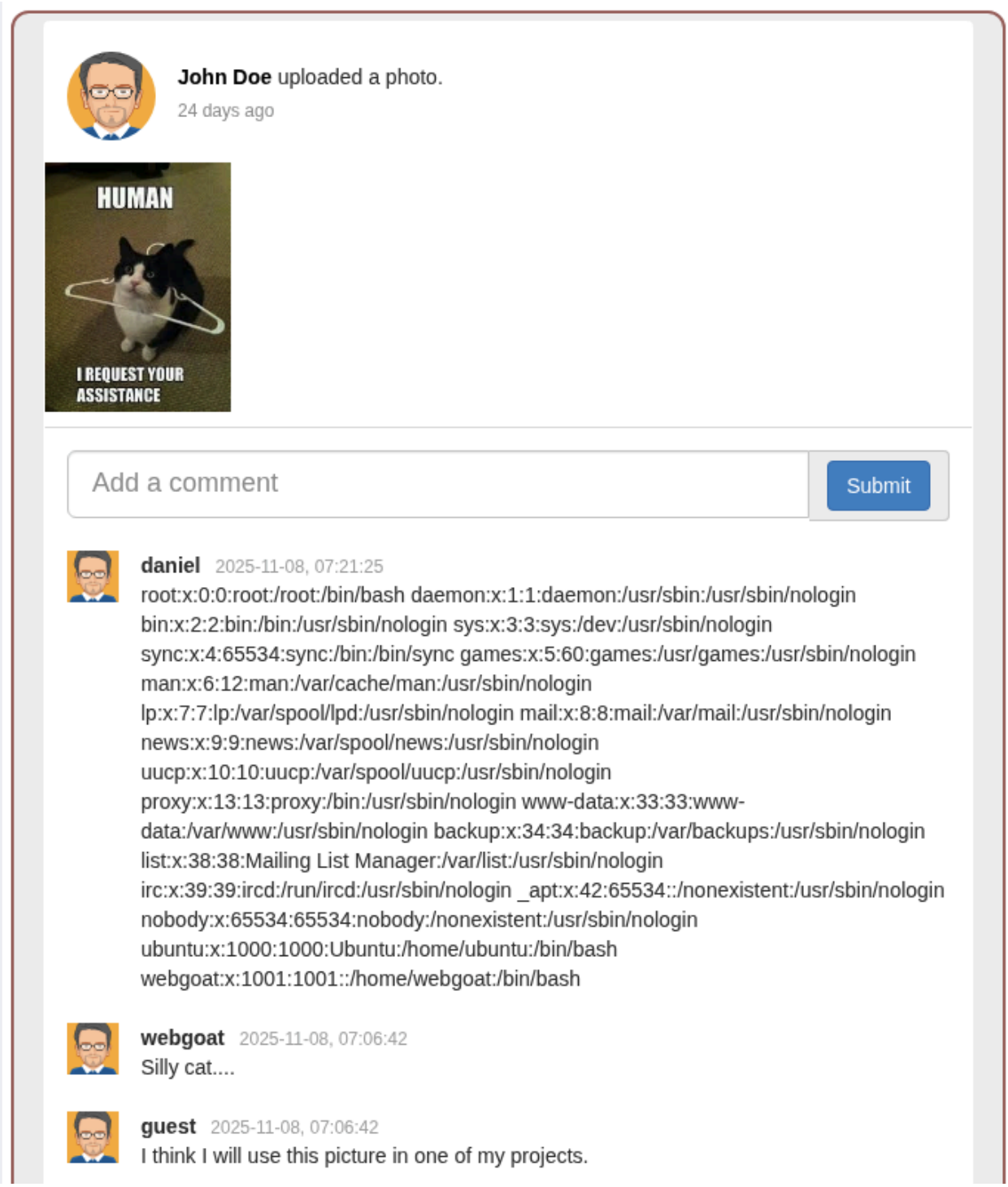


```
POST /WebGoat/xxe/simple HTTP/1.1
Host: localhost:8080
Content-Length: 59
sec-ch-ua-platform: "Linux"
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36
Accept: */*
sec-ch-ua: "Not=A?Brand";v="24", "Chromium";v="140"
Content-Type: application/xml
sec-ch-ua-mobile: ?0
Origin: http://localhost:8080
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8080/WebGoat/start.mvc?username=daniel
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9
Cookie: JSESSIONID=F776296A85E9E4D7B7A875973943A98

<?xml version="1.0"?>
<!DOCTYPE author [
<!ENTITY data SYSTEM "file:///etc/passwd">
]>
<comment>
  <text>
&data;  </text>
</comment>
```

```
HTTP/1.1 200
Content-Type: application/json
Date: Sat, 08 Nov 2025 06:21:25 GMT
Connection: close
Content-Length: 237

{
    "lessonCompleted": true,
    "feedback": "Congratulations. You have successfully completed the assignment.",
    "feedbackArgs": null,
    "output": null,
    "outputArgs": null,
    "assignment": "SimpleXXE",
    "attemptWasMade": true
}
```



Recommendation :

- Disable external entities and DTDs in XML parsers when not required. Explicitly

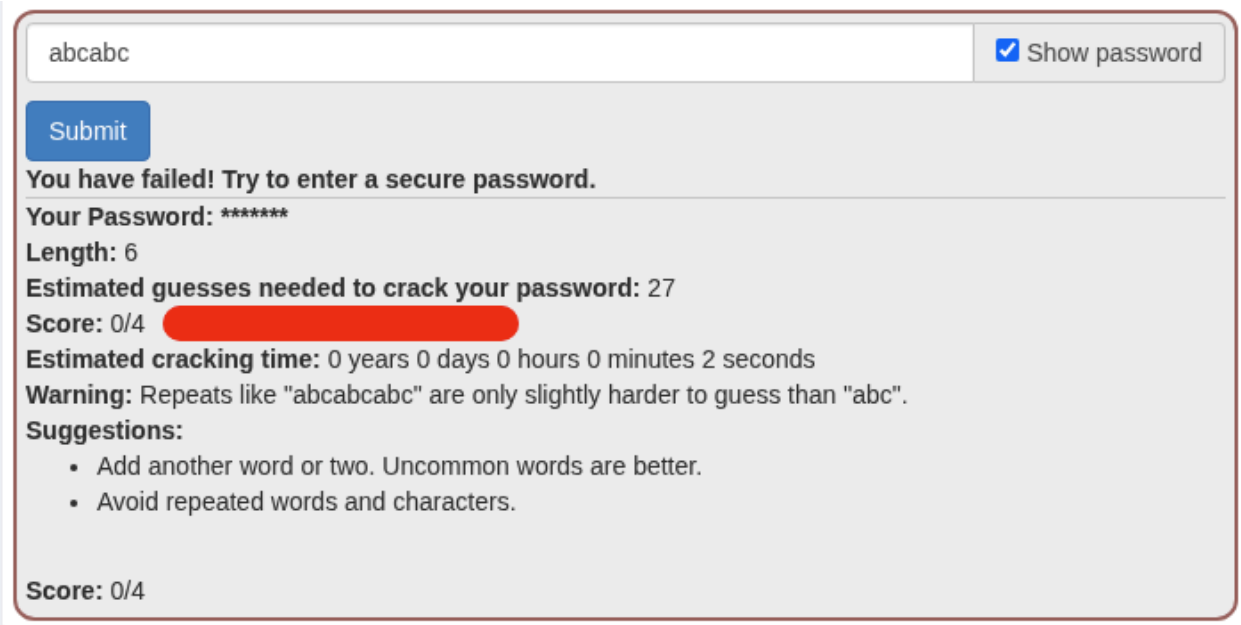
configure XML processors to reject DTDs.

- Update and harden XML libraries, ensuring they are current and configured securely per vendor guidelines.
- Avoid XML when possible: use safer formats such as JSON to exchange data, avoiding unnecessary XML deserialization.
- Validate and sanitize user input through whitelists before processing XML. Log and monitor anomalous XML requests for early detection.

References:
https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

05 - Insecure Passwords

This vulnerability is located in A7 Identity & Authentication Failure - Secure Password - Section 4.
Password management must follow strong security practices. It is necessary to define minimum complexity requirements to prevent users from choosing weak or easily guessable passwords.
For example, the password abcabc could be cracked in approximately 27 iterations, taking about 2 seconds on a modern computer.
In contrast, a password combining letters and numbers, such as P3P3T3V1G1l4, would require exponentially more iterations to brute-force, with an estimated cracking time of approximately 128 years, making brute-force attacks impractical.



Recommendation :

- Enforce a secure password policy that requires users to comply with minimum complexity parameters defined by the application.
- Require a minimum of 8 characters for all passwords.
- Include uppercase and lowercase letters, avoiding simple or repetitive patterns.
- Require at least one numeric digit in the password.
- Enforce the inclusion of a special character (e.g., @, #, $, %) to increase entropy and reduce susceptibility to brute-force or dictionary attacks.

References:

https://medium.com/@mrummanhasan/organization-password-policy-a-practical-hardening-guide-9b9079e6532b