UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

ICT risk assessment

Experimental evaluation of Caido security tool

Final Report

Elisa De Filomeno

Academic Year: 2023/2024

Indice

1 Introduction **4**
- 1.1 Caido Overview . . . 4
  - 1.1.1 Why Use Caido? . . . 4
  - 1.1.2 Who Should Use Caido? . . . 5
  - 1.1.3 Caido's strengths . . . 5
- 1.2 Installation . . . 6

2 Caido main features **7**
- 2.1 Workspace . . . 7
- 2.2 Sitemap . . . 8
- 2.3 Scope . . . 9
- 2.4 Filters . . . 10
- 2.5 Intercept . . . 11
- 2.6 HTTP History . . . 12
- 2.7 WS History . . . 13
- 2.8 Match & Replace . . . 13
- 2.9 Replay . . . 14
- 2.10 Automate . . . 14
- 2.11 Workflows . . . 15

3 Experimental tests **17**
- 3.1 Testing for XSS (Cross-site scripting) . . . 17
  - 3.1.1 Reflected XSS test using Caido . . . 18
  - 3.1.2 Possible Solutions to Prevent Reflected XSS Attacks . . . 20
- 3.2 OWASP Juice Shop . . . 22
- 3.3 Testing for SQL Injection . . . 23
  - 3.3.1 Explanation of Attack Mechanism . . . 24
  - 3.3.2 Possible Solutions to Prevent SQL Injection Attacks . . . 25
- 3.4 Brute Forcing Login Pages . . . 26
  - 3.4.1 Possible Solutions to Prevent Brute Force Login Attacks . . . 27
- 3.5 Testing for CSRF (Cross-Site Request Forgery) . . . 29
  - 3.5.1 CSRF Attack Test Execution . . . 29
  - 3.5.2 Generating the CSRF PoC using Caido . . . 30
  - 3.5.3 Explanation of Attack Mechanism . . . 33

3.5.4 Possible Solutions to Prevent CSRF Attacks 33

Capitolo 1

Introduction

1.1 Caido Overview

Caido is a powerful and versatile **web security tool** designed for testing web applications and identifying vulnerabilities.

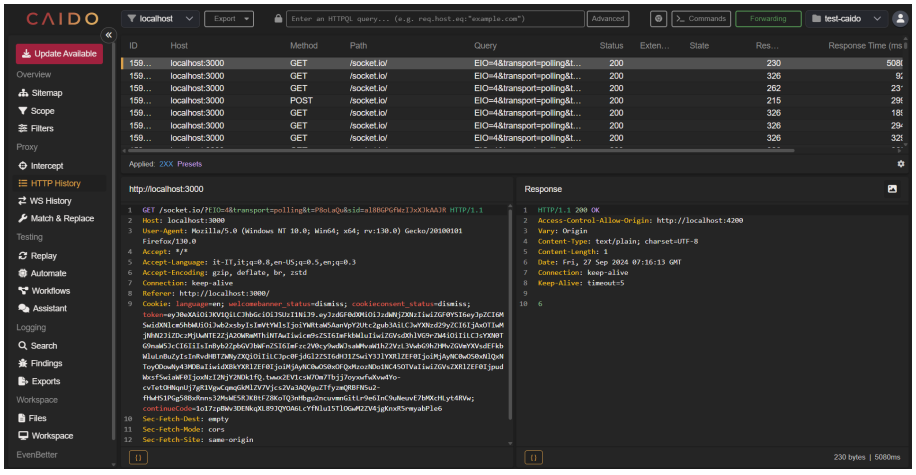

Figura 1.1: Overview of Caido's Interface and Features

1.1.1 Why Use Caido?

Caido is an essential tool for anyone involved in **web security** and **penetration testing**[1]. Its primary function is to **intercept** and analyze HTTP(S) traffic between a **client** (typically a user's device, such as a web browser) and a **server** that hosts the web application.

The following figure illustrates how Caido functions as a **proxy**[2]:

[1]Penetration testing: also known as ethical hacking, is the practice of simulating cyberattacks on a system to identify vulnerabilities that could be exploited by malicious actors.

[2]A proxy is an intermediary that forwards requests and responses between a client and a server.

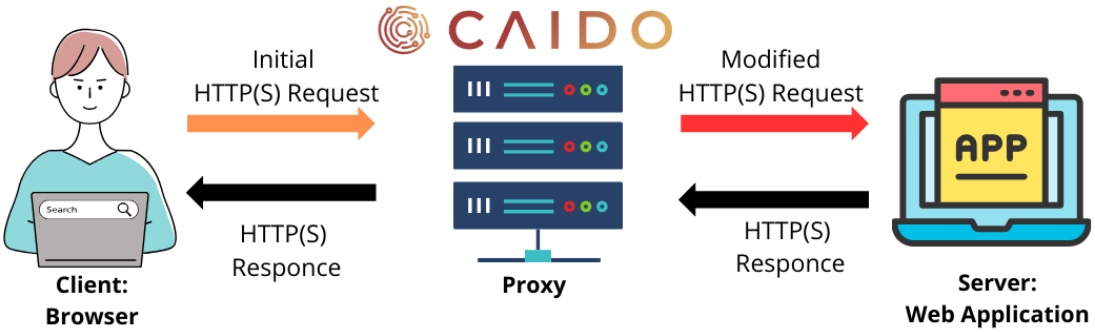

Figura 1.2: Caido as a proxy

In its role as a proxy, Caido captures all HTTP(S) requests from the client before they reach the server. This functionality allows security professionals to examine and modify these requests to test for vulnerabilities effectively. After the server processes the request, Caido analyzes the response before it is returned to the client, facilitating the identification of misconfigurations or security flaws within the application.

1.1.2 Who Should Use Caido?

Caido is highly beneficial for a wide range of professionals involved in web security, including:

- **Penetration testers and ethical hackers**: Utilize Caido to simulate attacks, identify vulnerabilities in web applications, and ensure systems are secure.
- **Web developers**: Employ Caido to ensure their applications comply with industry best practices and are resilient against potential attacks.
- **Security analysts**: Monitor web traffic, assess security configurations, and detect vulnerabilities in real time for continuous security assessment.

While Caido is designed for ethical purposes, its traffic interception capabilities could potentially be misused by malicious actors. This dual-use nature underscores the importance of responsible usage and adherence to legal guidelines.

1.1.3 Caido's strengths

Caido is optimized for **high performance**, providing a more **resource-efficient** alternative to tools like *Burp Suite*[3]. Built with **Rust**[4], Caido utilizes a **client/-server architecture**. In this system, the *client* is the web interface used by the

[3]Burp Suite is a popular web vulnerability scanner used to identify security flaws in web applications.

[4]Rust is a systems programming language focused on safety, performance, and concurrency.

user, while the *server* processes the requests. This architecture enables the client to interact with the server remotely from any device with a web browser, making Caido more flexible and efficient. This design is especially advantageous when the local machine lacks sufficient resources to run Caido without compromising speed or performance.

1.2 Installation

To install and configure Caido, I followed the official Caido documentation. Below are the steps I took to install the software:

1. **Download the Installation Package**: I downloaded the application from https://caido.io/, selecting the appropriate version for my operating system (Windows).[2]

2. **Run the Package and Follow the Installation Wizard**: After the download, I extracted the contents of the installation package and ran the installer, following the prompts in the installation wizard to complete the setup process.

3. **Import Caido's CA Certificate to Capture Web Traffic**: I imported and trusted Caido's *CA certificate*[5] in my browser (Firefox). This step is necessary for Caido to intercept HTTP(S) traffic. By default, Caido listens on IP address *127.0.0.1* and port *8080*, restricting access to the local machine. After completing this step, the browser was configured to route its traffic through Caido.

[5]CA certificate: a digital certificate that enables secure communication over the internet by establishing a secure connection between a client and server.

Capitolo 2

Caido main features

Let's start with an overview of Caido main features. In the following picture we can see the starting layout of Caido and its main features.

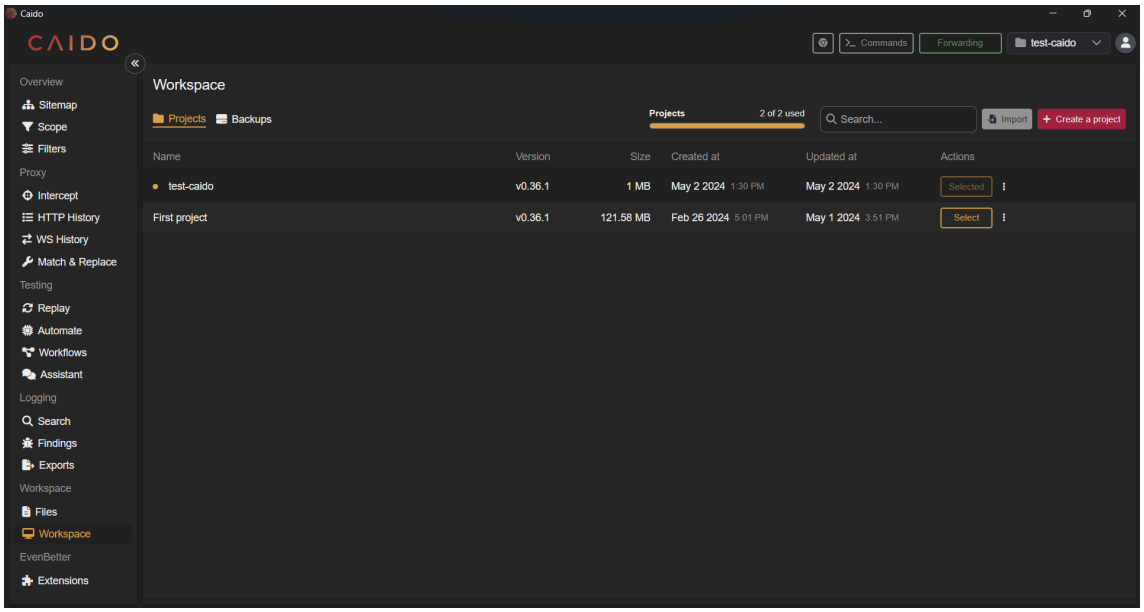

Figura 2.1: First look at Caido

2.1 Workspace

The workspace allows users to create and manage projects in order to organize tasks under specific projects.In fact one of Caido's standout features is its ability to manage multiple projects simultaneously. Unlike Burp Suite's free version, which restricts users to a single project, Caido allows for seamless switching between projects. This is particularly beneficial for testers who juggle various web applications. For example in figure 2.12, we can see that I created two projects under the names of *First project* and *test-caido*, selecting the second one as my current project.

2.2 Sitemap

The *Sitemap* interface allows the user to visualize and navigate the file structure of any website that is proxied through Caido. The Sitemap page offers a clear and organized view of the website's structure, allowing users to easily identify and navigate different sections.

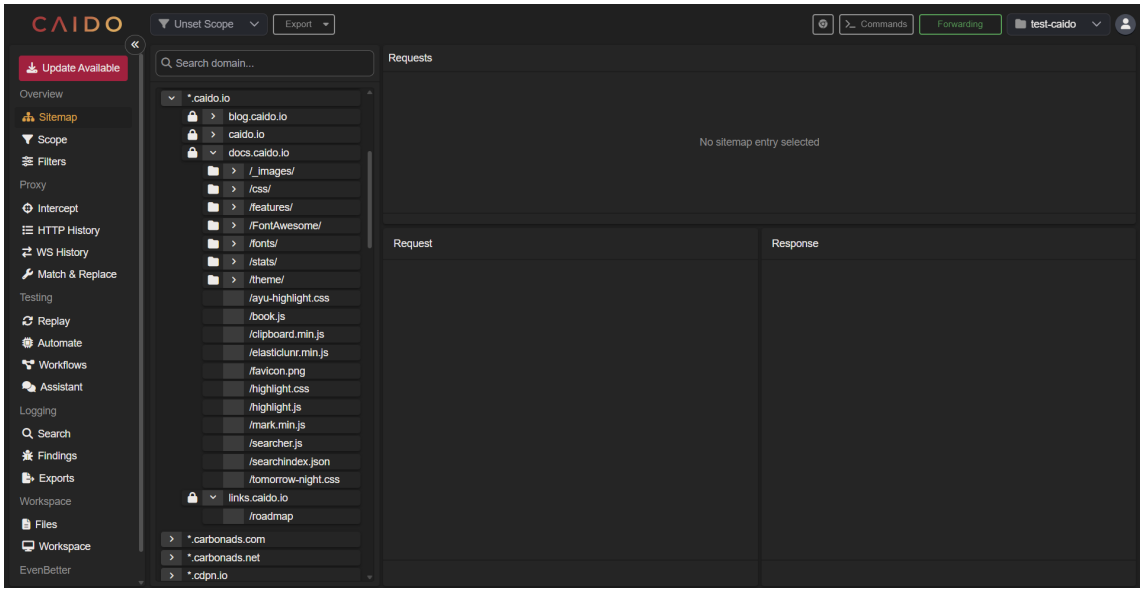

Figura 2.2: Sitemap view

It features a tree-like structure, with the root node representing the main domain of the site. Each branch indicates a subdomain or subfolder, while the leaves correspond to individual requests. For example the following image illustrates the sitemap of the *OWASP Juice Shop* website:

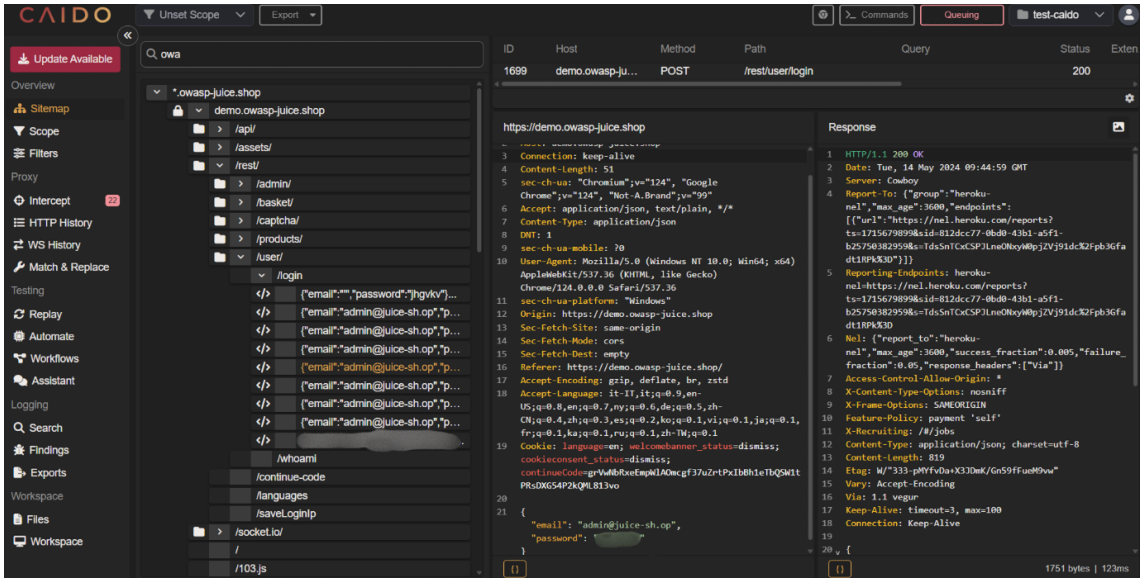

Figura 2.3: Sitemap view of the *OWASP Juice Shop*, showing the hierarchical structure of domains, subdomains, and requests.

2.3 Scope

The *Scope* interface enables users to create *Scope Presets*, which match requests across the application using a specified list of *Hosts*[1] designated as either *In Scope* or *Out of Scope*. In the following image, I demonstrate how I created a preset called *Juice Shop Herokuapp* to capture all requests from the host "juice-shop.herokuapp.com".

[1]Hosts refer to specific domain names or IP addresses from which requests are made.

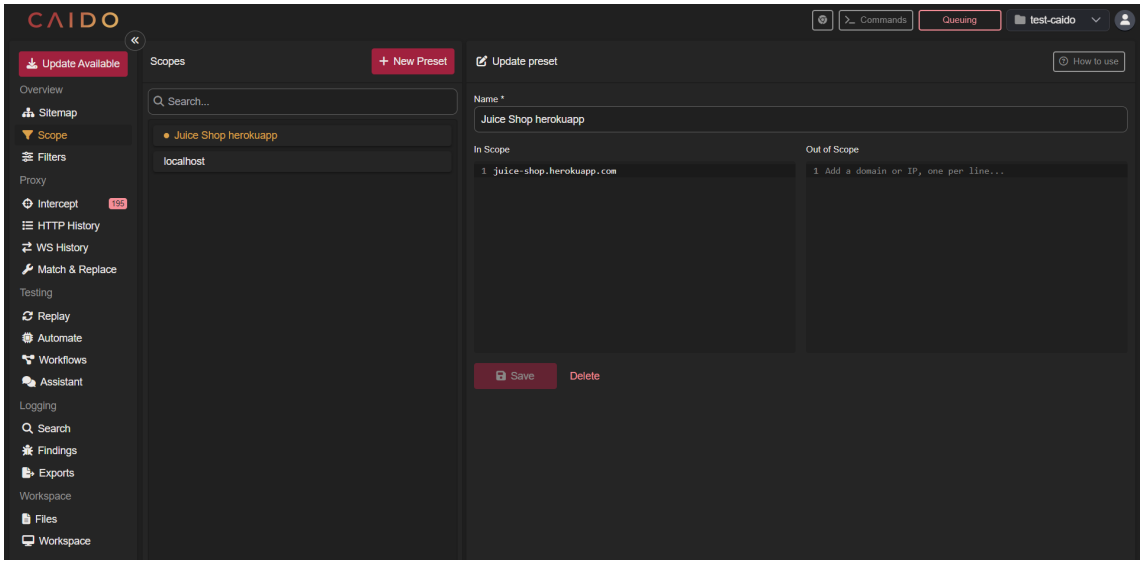

Figura 2.4: Caido *Scope* page

2.4 Filters

This tool allows users to filter HTTP requests and responses that are proxied through Caido. The following image shows the *Filters* feature:

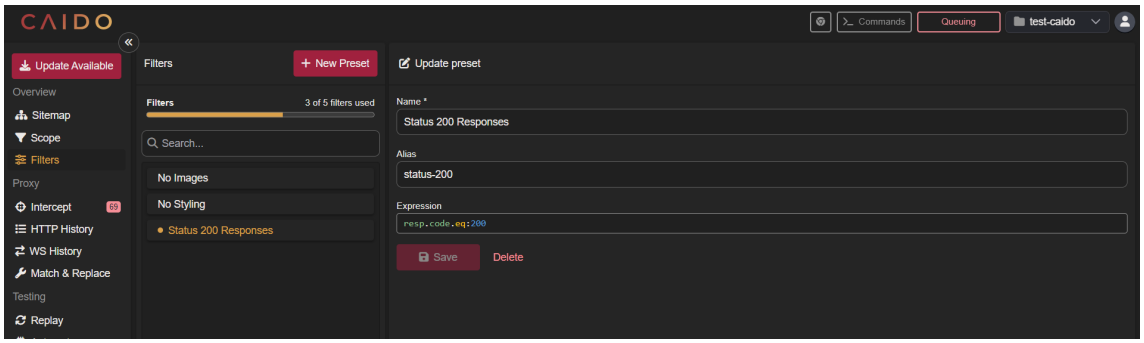

Figura 2.5: Caido *Filters* page

The query language used is *HTTPQL*, recently developed by Caido to enable users to create precise search conditions [7]. In the image, I created three *Presets*, which are HTTPQL queries. The last preset is called "Status 200 Responses" and is used to filter HTTP responses where the status code is exactly "200", meaning it retrieves only successful responses from the server. The structure of the HTTPQL query is shown in the following image:

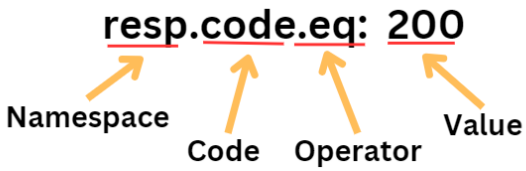

Figura 2.6: Constructing primitives of a HTTPQL Filter Clause

2.5 Intercept

The Intercept page allows the user to control the flow of requests through the proxy by pausing and resuming *forwarding*. When *forwarding* is paused, requests and responses that go through the proxy are temporarily stored in the *Intercept Table*, where the user can review, edit, or drop them before forwarding them.

The ***Forwarding*** button in the top right corner of the page must be toggled to prevent requests from being forwarded through the proxy, queuing them in the *Intercept table* instead.

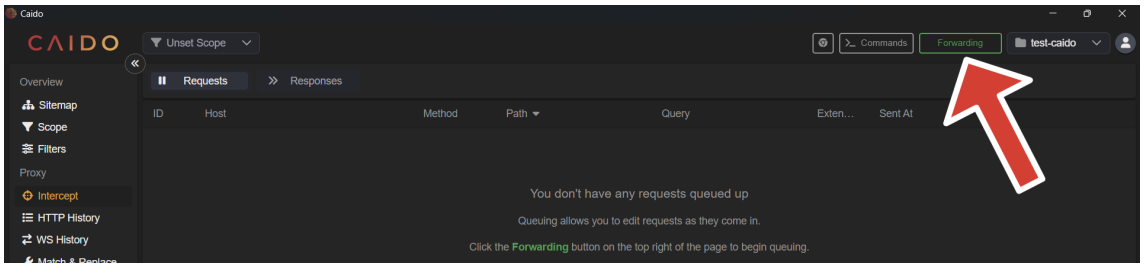

Figura 2.7: Caido Intercept feature

In the following picture we can see all requests intercepted from the browser after clicking on the *Forwaring* button.

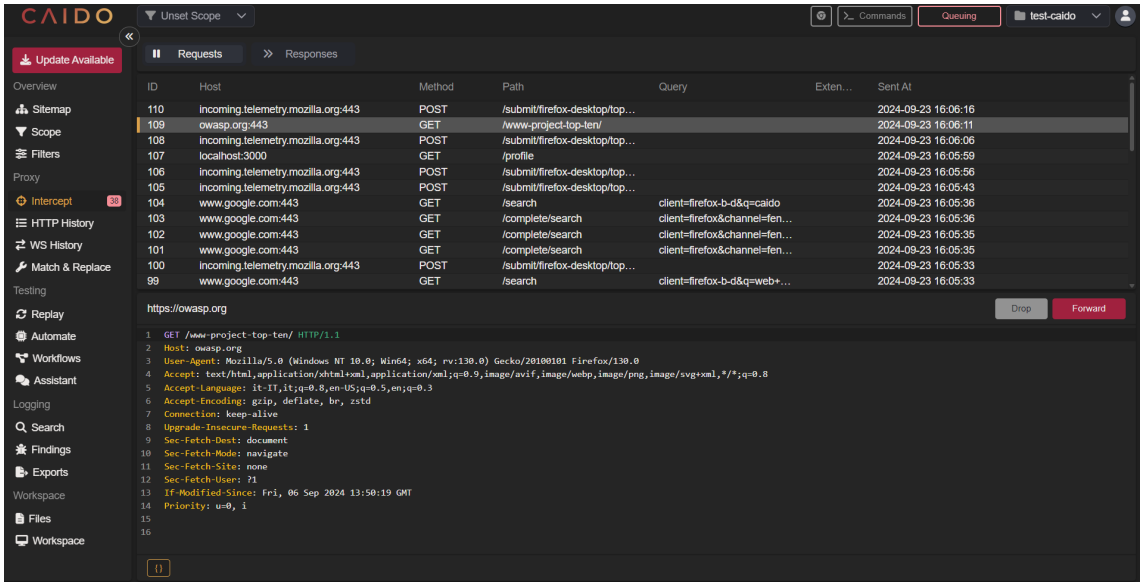

Figura 2.8: The Caido *Intercept table*, displaying queued requests and responses

Each request in the *Intercept table* can be chosen and **edited** its contents, forward or drop it. The ***Queuing*** button in the top right corner of the page must be toggled to resume forwarding. This will forward all the queued up requests in the *Intercept table*.

2.6 HTTP History

The *HTTP History* feature provides a detailed record of all HTTP requests and responses as they pass through the Caido proxy.

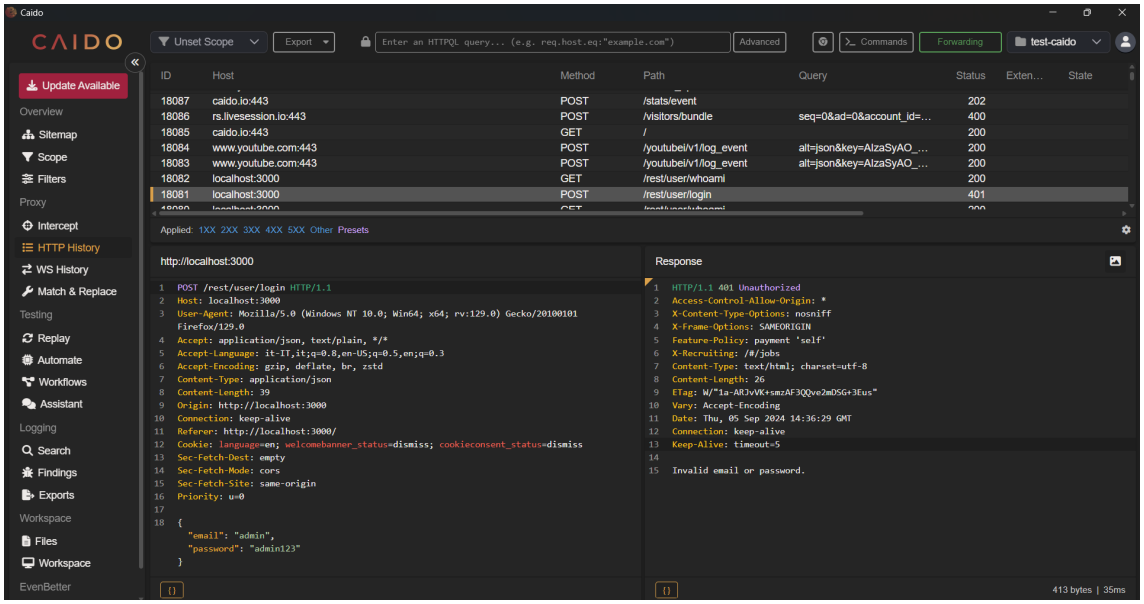

Figura 2.9: HTTP History that shows how the client sent a *login request* to the server with email and password credentials. The server responded with *401 Unauthorized*, returning the message "Invalid email or password."

In this page it's possible to:

- **Export** HTTP History data as *JSON* or *CSV*
- In the **Enter an HTTPQL query...** input bar it's possible to filter the history feed using the *presets* created with the *Filters* feature.
- By using the **Advanced** option, it is possible to filter the history feed for HTTP responses with status codes in the 1xx, 2xx, 3xx, 4xx, or 5xx ranges. For example, codes in the 2xx range indicate successful responses (such as 200).

2.7 WS History

The *WS History* page enables users to view the data exchanged between the client and the server through the proxy, and some additional information about the messages.

2.8 Match & Replace

The *Match & Replace* page enables users to match and replace rules using HTTPQL and regex syntax. For example, it can be used to search for a specific term in the body of HTTP responses and replace it with another term, possibly containing a different value.

2.9 Replay

The *Replay* page enables users to **edit** individual requests. Once edited, the modified request can be *forwarded* to view the corresponding response. I will utilize this feature in the experiments conducted and reported in the following chapter.

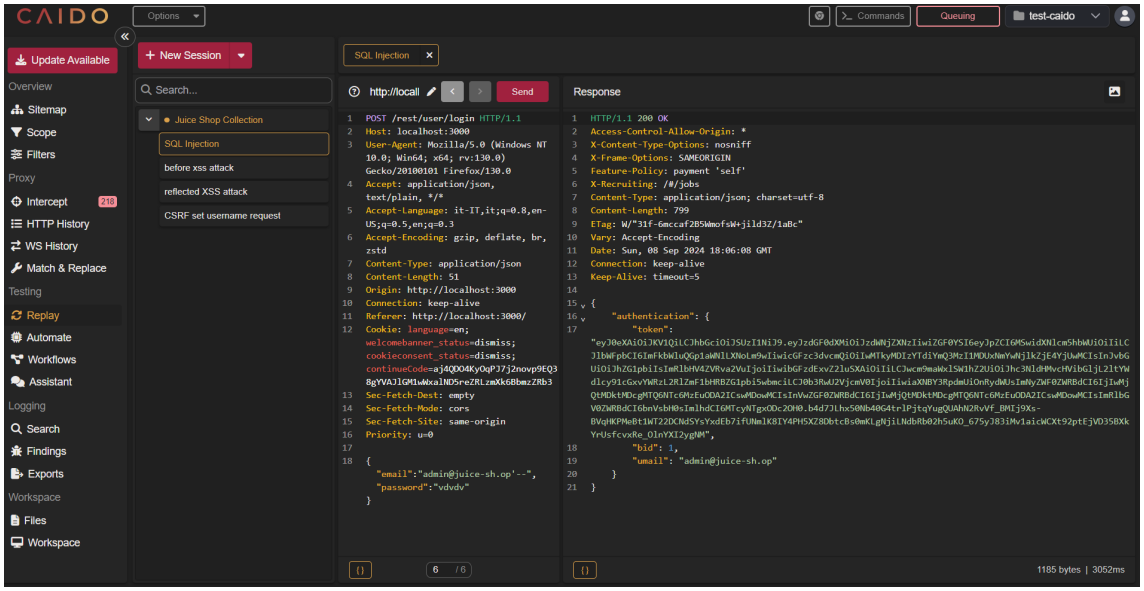

Figura 2.10: Replace view

2.10 Automate

The *Automate* page enables users to send requests in bulk. This feature is useful for *brute forcing*[2] and *fuzzing*[3] as it allows rapid modification of certain request parameters using *wordlists*[4]. This feature will be utilized in the following experiments to *brute-force* the password of a login page in an application, testing the application for this type of vulnerability.

After selecting one or more **placeholder terms** to be replaced, the user can choose the ***Attack Strategy*** from the following options:

- **Sequential**: This option replaces markers one by one. If you have multiple markers, only one will be changed in each request.
- **All**: This option replaces all the markers with the same value.

[2]Brute forcing is a method of trial and error used to decode encrypted data or gain unauthorized access by systematically attempting all possible combinations.

[3]Fuzzing is a testing technique used to identify vulnerabilities in software by inputting random, unexpected, or invalid data.

[4]Wordlists are predefined lists of words or values used in testing, typically for password cracking or input validation.

- **Parallel**: This option replaces all the markers with different values from different payloads. Each payload must have the same number of elements.
- **Matrix**: This option replaces all the markers with every possible combination of payloads. The payloads can have different numbers of elements, but be careful—this can create a lot of requests.

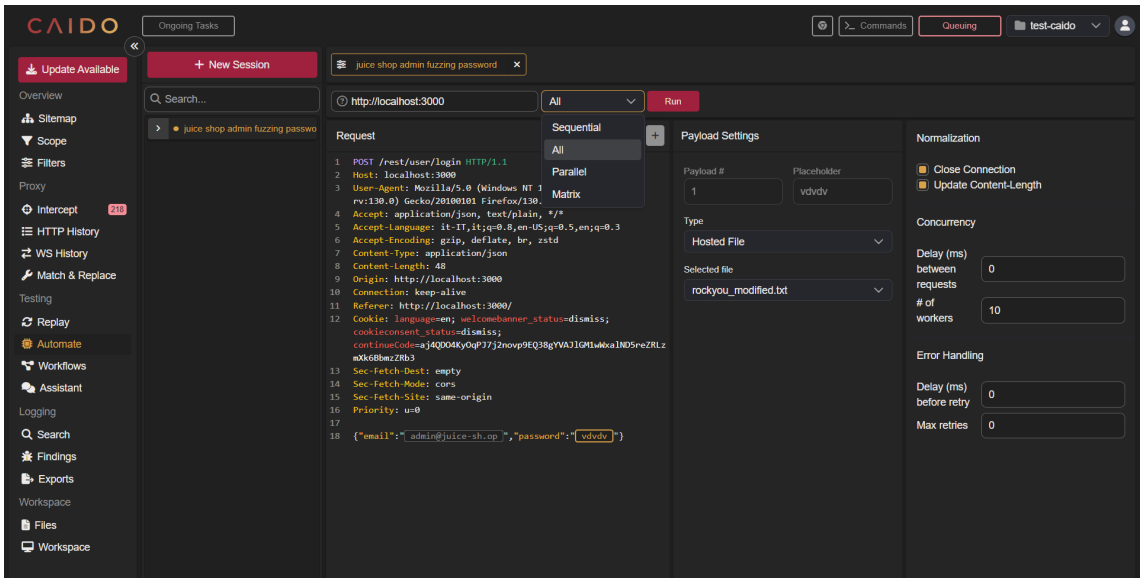

Figura 2.11: Caido *Automate* feature

2.11 Workflows

This feature allows users to automate repetitive tasks and create custom workflows that streamline the testing process. By using Flows, you can define a sequence of actions to be executed automatically, such as sending specific requests, modifying headers, or handling responses in a predefined manner.

The Workflows are executed server-side - thereby offloading processing power, providing enhanced performance and allowing seemless usage across multiple devices. Workflows created by others can also be downloaded and imported into your Caido instance.

In order to create a Workflow, Caido implements **Nodes**.

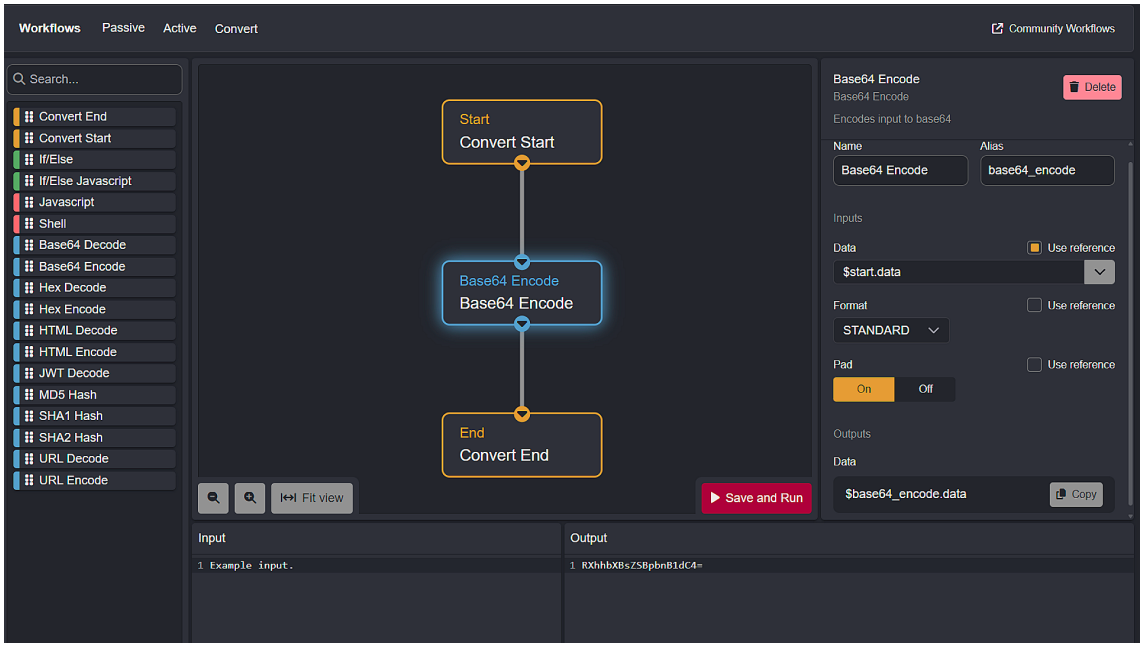

Figura 2.12: a Workflow that will take user-provided input, base64 encode it and then output the results.

There are three main types of Workflows, each of which are applied differently:

- **Passive Workflows**: will automatically trigger based on the specifications set when creating the Workflow.
- **Active Workflows**: must be manually triggered.
- **Convert Workflows**: will perform actions against supplied input.

Capitolo 3

Experimental tests

In this chapter, I will present several tests I conducted to demonstrate how Caido can be used to assess whether a web application is vulnerable to common web attacks.

The tool was used to test for the following vulnerabilities:

- Reflected Cross-Site Scripting (XSS) (see Section 3.1)
- SQL injection (see Section 3.3)
- Brute Forcing Login Pages (see Section 3.4)
- Cross-Site Request Forgery (CSRF) (see Section 3.5)

3.1 Testing for XSS (Cross-site scripting)

Cross-Site Scripting (XSS) is a web vulnerability that allows an attacker to inject malicious scripts into web pages viewed by users. Web browsers can execute commands embedded in HTML pages, supporting languages like JavaScript, VBScript, and ActiveX.

The term "cross-site" refers to malicious scripts being sent from the web server[1] to the client, where the client's browser executes the code with the trust of the server. The browser assumes that all scripts received from the server are safe, which leads to vulnerabilities when untrusted scripts are executed. This attack can lead to *theft of access credentials*[2], *denial-of-service*[3], *modification of web pages*, such as their appearance or functionality, and *execution of arbitrary commands on the client machine*.

Types of XSS attacks:

[1] A web server is a system that delivers web content, such as HTML pages, to clients (browsers) over the internet.

[2] Attackers can steal session tokens, cookies, or other authentication data.

[3] XSS can be used to overload or crash the web application.

- **Non-persistent (Reflected) XSS**: The most common form of XSS, where malicious code is immediately executed and not stored on the server.
- **Stored/Permanent XSS**: The malicious data is stored by the server and displayed on regular pages, affecting all users who view those pages.
- **DOM-based XSS**: This type of XSS occurs entirely on the client side, without server involvement. It is triggered by client-side JavaScript manipulating the DOM (Document Object Model), typically through URL parameters or fragments. The malicious script is executed in the victim's browser by interacting with these DOM elements.

3.1.1 Reflected XSS test using Caido

The test was conducted on a mock website named *Four or Four*, launched by Google to train for *Reflected XSS* vulnerability. [6] The site looks like the following image:

Figura 3.1: FourOrFour search bar, where users can enter queries.

After the user enters a query into the search bar, the input is passed as a parameter in the URL. In figure 3.1 the parameter passed is *?query=a*, which will search for the keyword 'a' on the website.

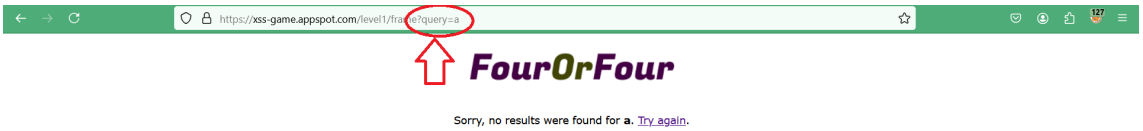

Figura 3.2: Four or Four simple query search

In figure 3.2 we can see how the search query for the string "a" was reflected in the URL as *?query=a* .

Using *Caido*, I intercepted this search query and then used Caido's *Replay* feature to observe the request-response cycle. The request was processed successfully, as shown in the Caido screenshot below:

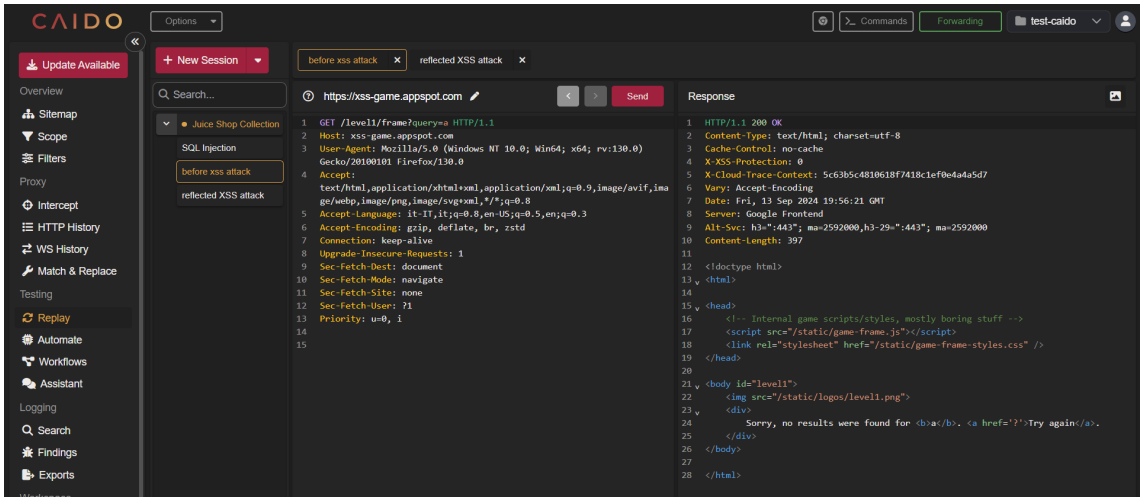

Figura 3.3: Caido screenshot showing the request for "a" and its corresponding response.

Next I tried the Reflected XSS attack by modifying the search "query" parameter to include *JavaScript*[4] code:

```
?query=<script>alert('XSS')</script>
```

This modification was replayed through Caido, and upon executing the request, the site displayed a reflected XSS alert popup, demonstrating the vulnerability. The screenshot below shows the successful XSS execution:

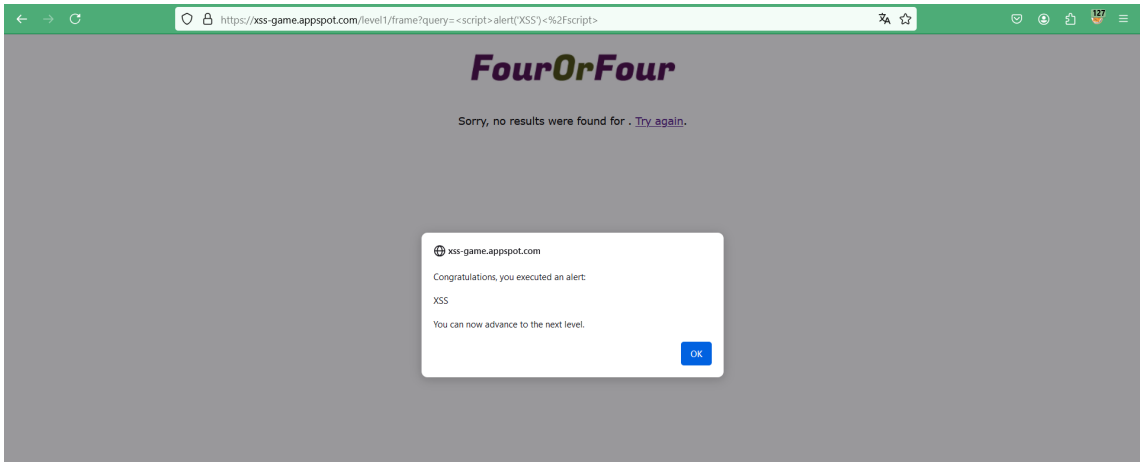

Figura 3.4: Reflected XSS alert popup demonstrating a successful attack.

The following screenshot highlights the request-response process during the attack using Caido:

[4]A programming language commonly used to create interactive effects within web browsers.

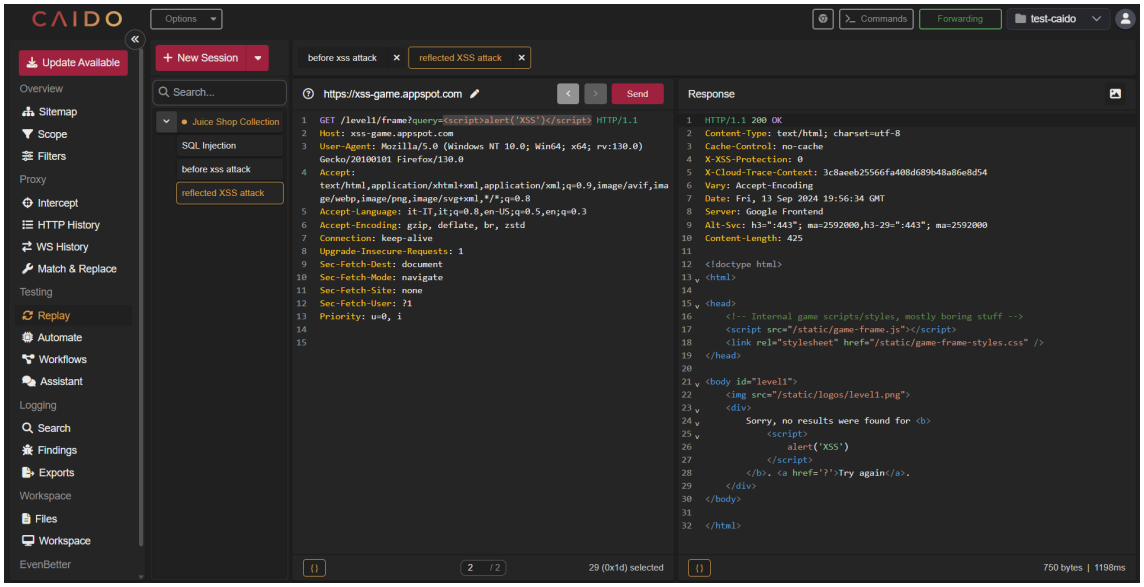

Figura 3.5: Caido tool showing a successful XSS attack with a modified query parameter.

3.1.2 Possible Solutions to Prevent Reflected XSS Attacks

To mitigate the risk of reflected XSS attacks, developers should implement several security measures:

- **Input Validation**: check if input is what expeted trhough *white listing*.
- **HTML Encoding**: encode output data when rendering user inputs in the browser, to prevent script execution. This ensures that any HTML tags or scripts are treated as text rather than executable code.
- **Web Application Firewall (WAF)**: A WAF monitors and filters incoming traffic to detect and block malicious input values and modifications of *read-only* parameters. It can be particularly useful for protecting legacy applications[5] where source code, expertise, or time for proper security implementations are lacking. However, it is not a one-size-fits-all solution, and its effectiveness depends on the specific application being protected; not all applications can be adequately secured with a WAF.
- ***HttpOnly* Cookies and *Secure* Flags**: Cookies should be tagged with the HttpOnly flag to restrict access solely to the server, preventing client-side scripts[6] from accessing them and reducing the risk of disclosure via DOM. Caution

[5]Software that continues to be used despite being outdated or no longer supported, often built on obsolete technologies.

[6]Scripts that run in the user's browser and can manipulate web content.

is advised due to potential compatibility issues, especially with older browsers like Internet Explorer. It is important to note that cookies are sent with every HTTP request, making them vulnerable to attacks like the Trace Method. While "Secure" Cookies[7] are transmitted only over SSL[8], they can still be compromised through XSS if the application is not adequately secured.

[7]Cookies that are transmitted only over secure HTTPS connections.

[8]Secure Sockets Layer, a protocol for establishing secure communication over the internet.

3.2 OWASP Juice Shop

For the following experimental tests I used the OWASP Juice Shop web application.[4] This application is entirely written in JavaScript and is listed in the OWASP VWA Directory[5], a registry of known vulnerable web and mobile applications currently available. It's commonly used for security training since it contains many vulnerabilities.

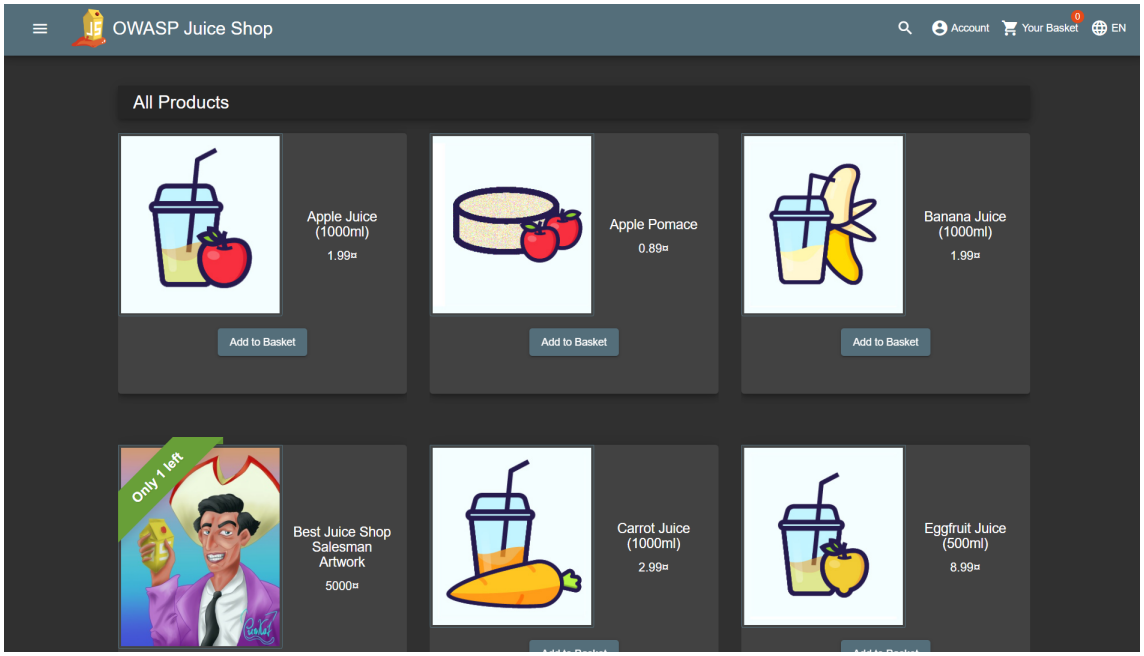

Figura 3.6: OWASP Juice Shop page.

3.3 Testing for SQL Injection

SQL Injection (SQLi) is a technique where malicious SQL commands are inserted into input fields to manipulate the database or gain unauthorized access. In the context of a login form, this often involves crafting input that tricks the system into bypassing authentication checks.

I conducted a simple SQL injection test on the *OWASP Juice Shop* login page, using the *Replay* feature of Caido.

In the figure below, you can see a typical *login* request using the admin email with a random password. The credentials used are:

- **email:** admin@juice-sh.op
- **password:** vdvdv

As expected, this resulted in an *HTTP 401 Unauthorized*[9] response due to invalid credentials.

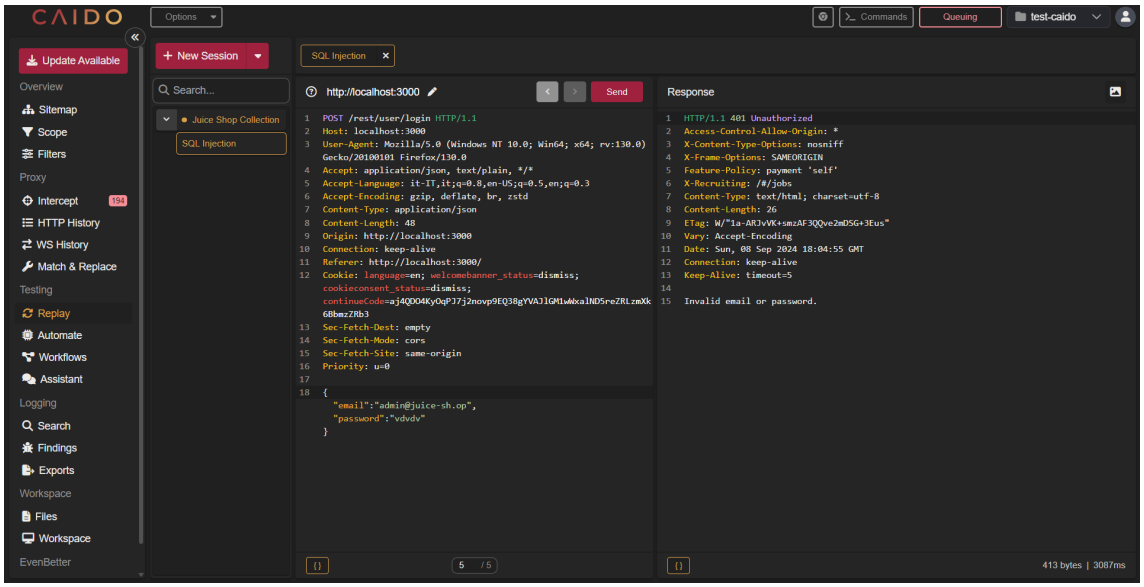

Figura 3.7: Caido tool showing a *login* request of the *admin@juice-sh.op* account, with a random password, and how it resulted in a *HTTP 401 Unauthorized* responce.

The SQL query[10] executed by the login form in the backend[11] likely looks something like this:

[9]HTTP 401 Unauthorized: This status code indicates that the request lacks valid authentication credentials. The server requires authentication to grant access to the requested resource.

[10]SQL Query: A command used to interact with a database, allowing data retrieval, insertion, updating, and deletion.

[11]Backend: The server-side part of a web application responsible for processing requests, managing data, and handling business logic.

```
SELECT * FROM users WHERE username = 'admin@juice-sh.op'
AND password = 'vdvdv';
```

Instead of entering *admin@juice-sh.op* as the email, I input *admin@juice-sh.op'–* in the email field. The result is shown below:

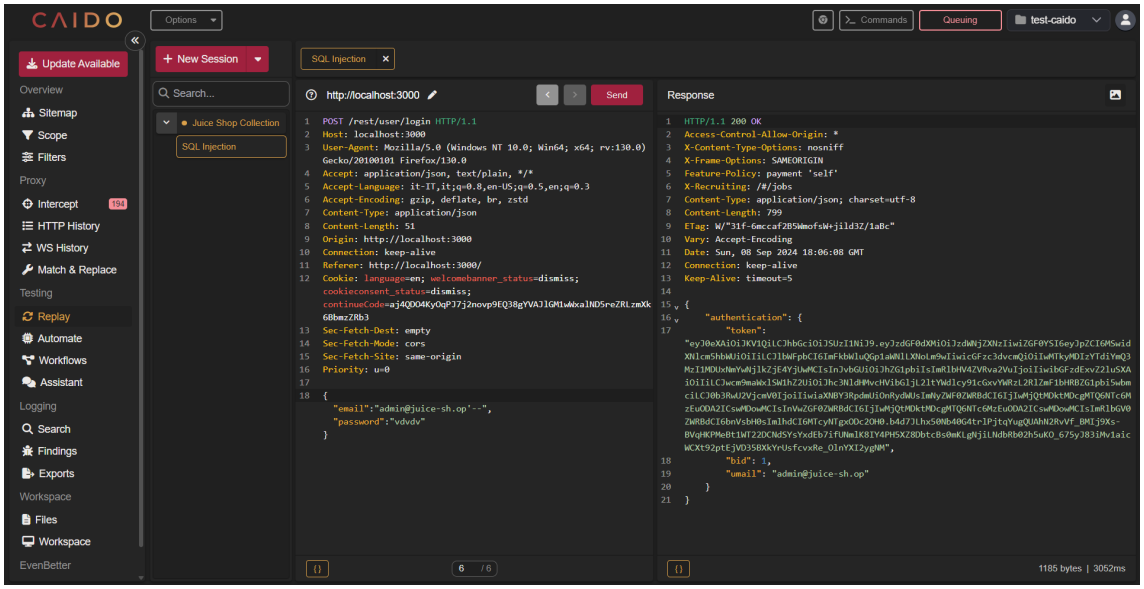

Figura 3.8: Caido tool showing a successful *SQL Injection* attack, resulting in an *HTTP 200 OK* response, indicating valid credentials.

3.3.1 Explanation of Attack Mechanism

This attack succeeded because the injected SQL code ('–) altered the query, bypassing the password check. Specifically, the – comment syntax caused the query to ignore everything after the username check, transforming it into:

```
SELECT * FROM users WHERE username = 'admin@juice-sh.op';
```

In SQL, the – sequence denotes the beginning of a comment, effectively terminating the current SQL statement at that point. When the database executes this modified query, it no longer evaluates the password condition, allowing access without verifying if the password provided is correct.

[11]HTTP 200 OK: This status code indicates that the request was successful and that the server has returned the requested resource. It is the standard response for successful HTTP requests.

3.3.2 Possible Solutions to Prevent SQL Injection Attacks

This vulnerability arises when user inputs are directly concatenated into SQL statements without proper sanitization or validation. To mitigate the risk of SQL Injection attacks, the following solutions can be implemented:

- **Input Validation through Whitelisting**: This method involves allowing only well-defined sets of safe values, typically implemented using regular expressions (for example `[A-Za-z0-9]*` for usernames or `[0-1][0-9]` for numeric inputs). Whitelisting is preferred over blacklisting because it avoids the risks of missing harmful characters or blocking valid inputs, like the apostrophe in O'Brien. This approach can also be implemented effectively in a *web application firewall*[12], providing an additional layer of protection against SQL Injection and other vulnerabilities.

- **Use Prepared Statements and Bind Variables**: Utilize precompiled SQL statements that can be executed multiple times with different parameters, ensuring that user input is treated as data, not executable code. By using placeholders in SQL statements, known as *bind variables*, actual values are safely inserted at runtime, further protecting against SQL Injection attacks.

- **Least Privilege Principle**[13]: Limit database user permissions to only what is necessary for application functionality. [3]

[12]Web Application Firewall (WAF): A security system designed to monitor and filter HTTP traffic to and from a web application, protecting it from attacks such as SQL Injection, Cross-Site Scripting (XSS), and other vulnerabilities.

[13]Least Privilege Principle: a security principle stating that a user or system should only have the minimum access rights necessary to perform its functions.

3.4 Brute Forcing Login Pages

A **brute-force** attack is a method used to gain unauthorized access by systematically trying all possible combinations of usernames and passwords until the correct one is found.

To demonstrate how Caido can be used for this type of attack, I performed a brute-force attack on the login page of *OWASP Juice Shop*. Specifically, I attempted to discover the password for the account *admin@juice-sh.op*.

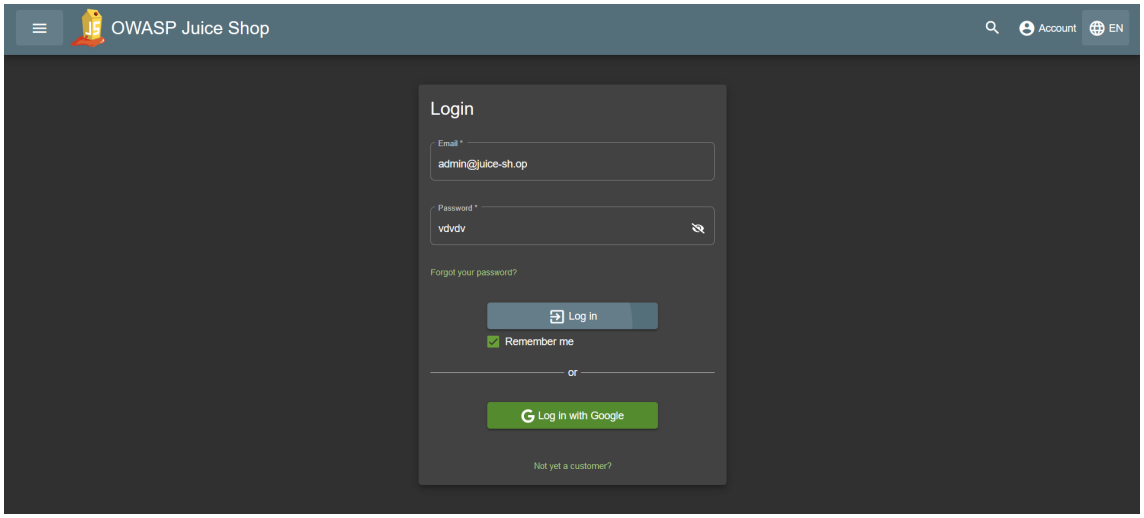

Figura 3.9: OWASP Juice Shope login page with a random password

The strategy was to queue the login request using Caido and employ the *Automate* feature to send multiple login attempts with different passwords. Lists of commonly used passwords, known as *wordlists*[14], such as the popular *rockyou.txt*, are easily accessible online. If the password is weak, it can be identified through this method.

[14] A wordlist is a list of words or phrases used to test authentication credentials or discover vulnerabilities through techniques like brute-force attacks or fuzzing.

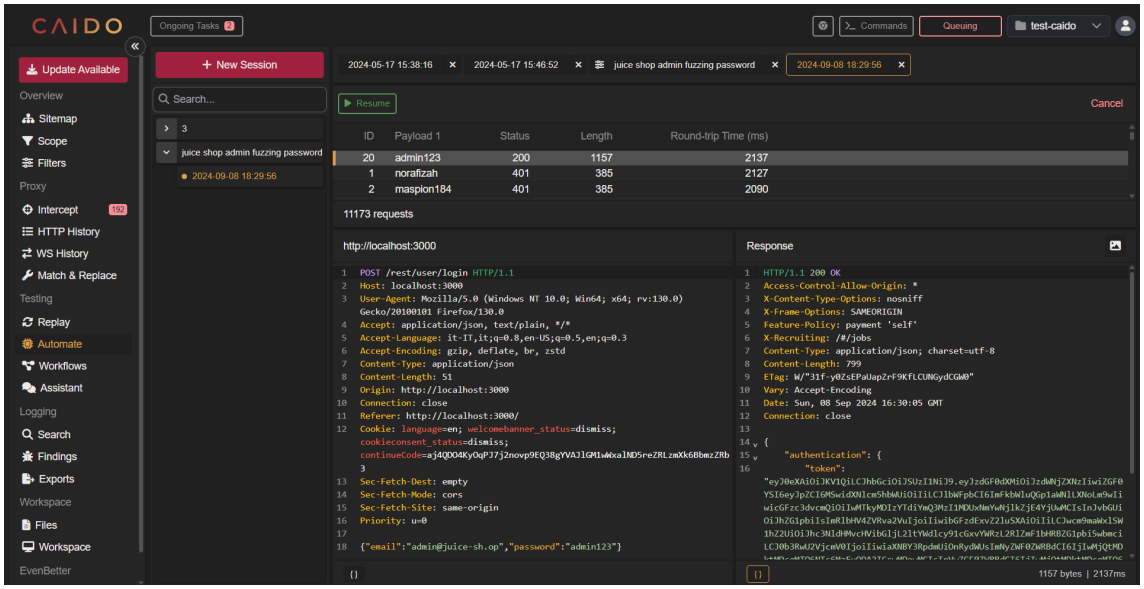

Figura 3.10: Caido tool showing a successful *brute-force* attack with the word *admin123*, resulting in an HTTP status 200 response, indicating valid credentials.

In figure 3.10, we can observe the outcome of the attack. The Caido tool successfully identified the correct password, *admin123*, as shown by the HTTP status 200 response, which confirms valid login credentials.

3.4.1 Possible Solutions to Prevent Brute Force Login Attacks

Brute force attacks attempt to gain unauthorized access by systematically trying various combinations of usernames and passwords. To mitigate the risk of brute force login attacks, the following solutions can be implemented:

- **Use Strong Passwords**: A strong password should be unique, long, and hard to guess. Passwords should avoid common patterns and personal information to resist cracking attempts. Additionally, strong recovery questions should complement strong passwords to enhance security.[8]

- **Restrict Access to Authentication URLs**: Changing the login page URL to a less predictable one can help reduce the risk of brute force attacks. For instance, moving from `/wp-login.php` to `/mysite-login` can deter automated tools, although advanced attacks may still target guessable links.

- **Limit Login Attempts**: By restricting the number of login attempts per user, the effectiveness of brute force attacks can be significantly reduced. Implementing temporary bans on IP addresses after a set number of failed attempts (e.g., five) can further protect against unauthorized access.

- **Use CAPTCHAs**: CAPTCHAs present challenges that are easy for humans to solve but difficult for bots, effectively blocking their attempts to gain access. For example, users may be required to check a box labeled "I'm not a robot," which analyzes their browsing behavior to ensure they are genuine users.
- **Use Two-Factor Authentication (2FA)**: Users must provide a second form of verification, such as a code sent to their email or a unique code generated by an authentication app, making unauthorized access more challenging.
- **Set Up IP Access Restrictions**: Restricting access to login and admin pages based on approved IP addresses can provide robust protection against brute force attacks. Requests from unapproved IPs can be denied with a HTTP response or blocked by a **firewall**.

3.5 Testing for CSRF (Cross-Site Request Forgery)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing.

3.5.1 CSRF Attack Test Execution

For this test, I used the OWASP Juice Shop web application. The figure below shows the User Profile page, specifically the admin account. The CSRF attack I conducted was designed to trick the user into executing, withouth its knowledge, the *Set Username* action with a modified username of my choosing.

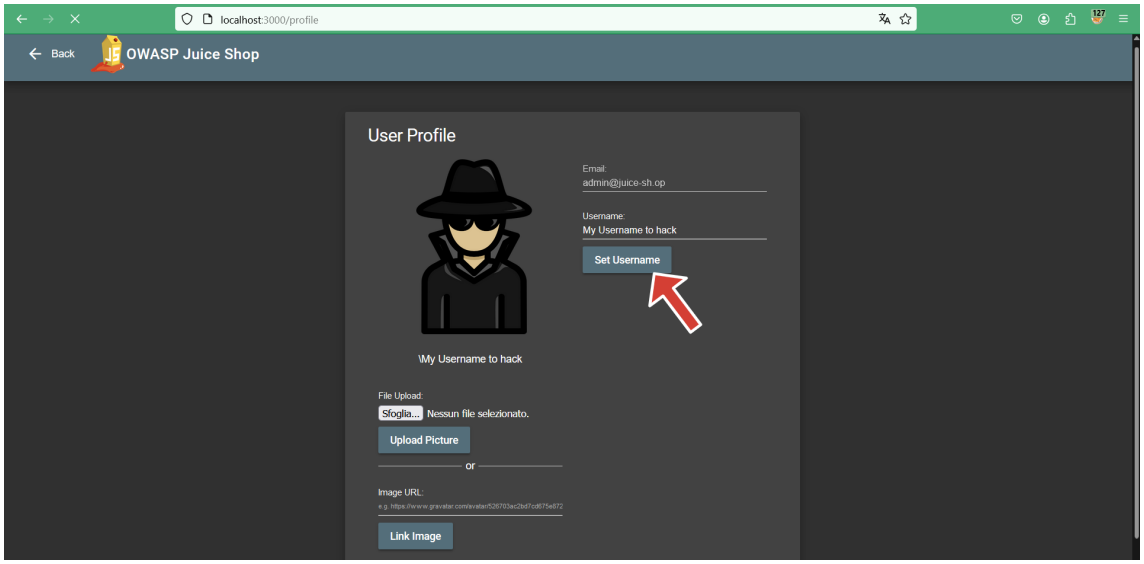

Figura 3.11: OWASP Juice Shope profile and *Set Username* action.

The key concept behind the CSRF attack I performed involved intercepting the form submission used to update the username. Using Caido, I captured the HTTP request and modified the username value in the intercepted request. Next, I generated a CSRF PoC by creating a malicious HTML form, which contained the forged *HTTP POST* request with the altered username.

Once the malicious HTML form was created, the attack could be executed by tricking the user into opening it—possibly by sending a link that redirects them to the malicious HTML file. If the user unsuspectingly clicks the link and opens it in the **same browser** where they are logged into Juice Shop, the forged *POST* request would be sent automatically without their knowledge or consent.

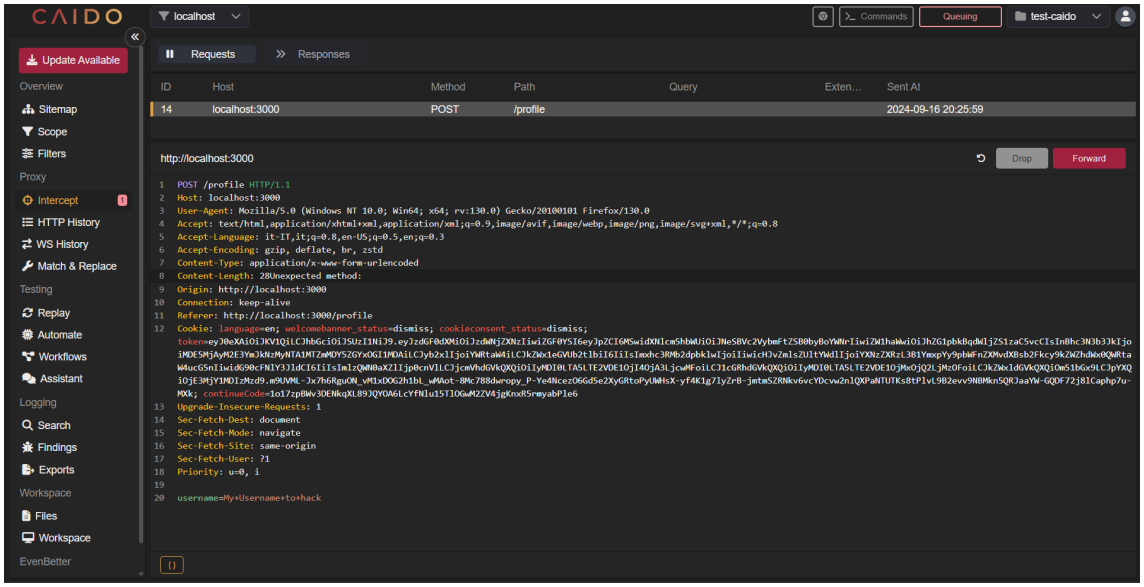

Figura 3.12: Caido tool showing the intercepted post request.

3.5.2 Generating the CSRF PoC using Caido

In Caido it's possible to generate a CSRF PoC (Proof of Concept)[15] using its *Assistant* feature. Unfortunately, this feature is only available for Caido Pro users. However, I'll demonstrate how to achieve the same result using the *Workflow* feature available in the free version of Caido.

[15] A CSRF PoC is a demonstration of how a CSRF attack can be executed. It typically involves crafting a malicious HTML form or URL to exploit a vulnerability, allowing unauthorized actions to be carried out on behalf of an authenticated user.

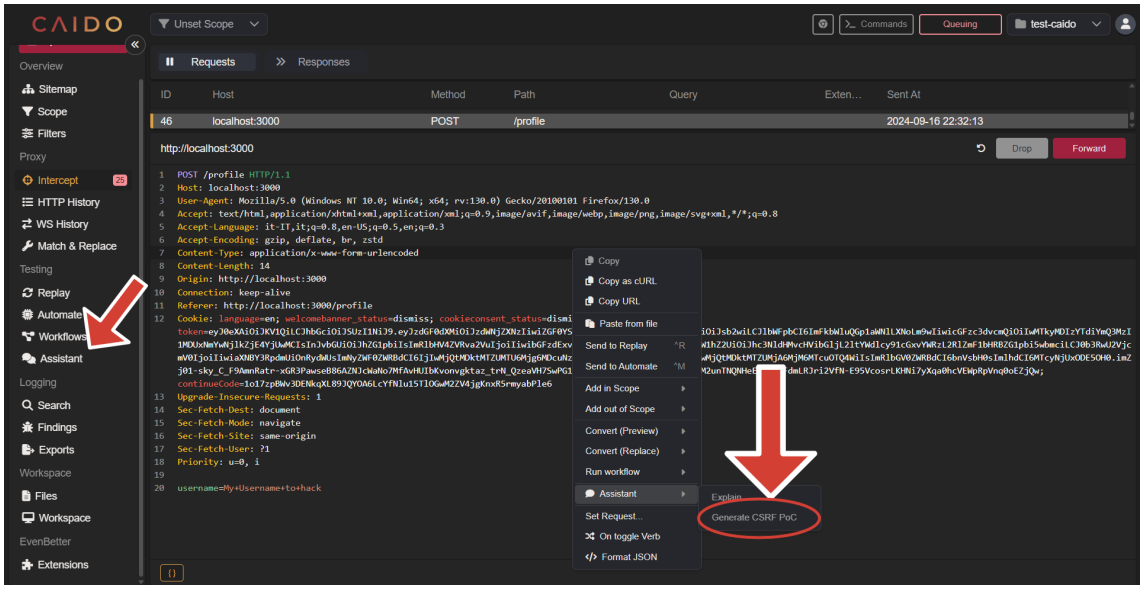

Figura 3.13: Assistant feature to Generate CSRF PoC, only for Caido Pro' users.

To bypass this limitation, a custom workflow can be used. I downloaded the JSON file from this link [1].

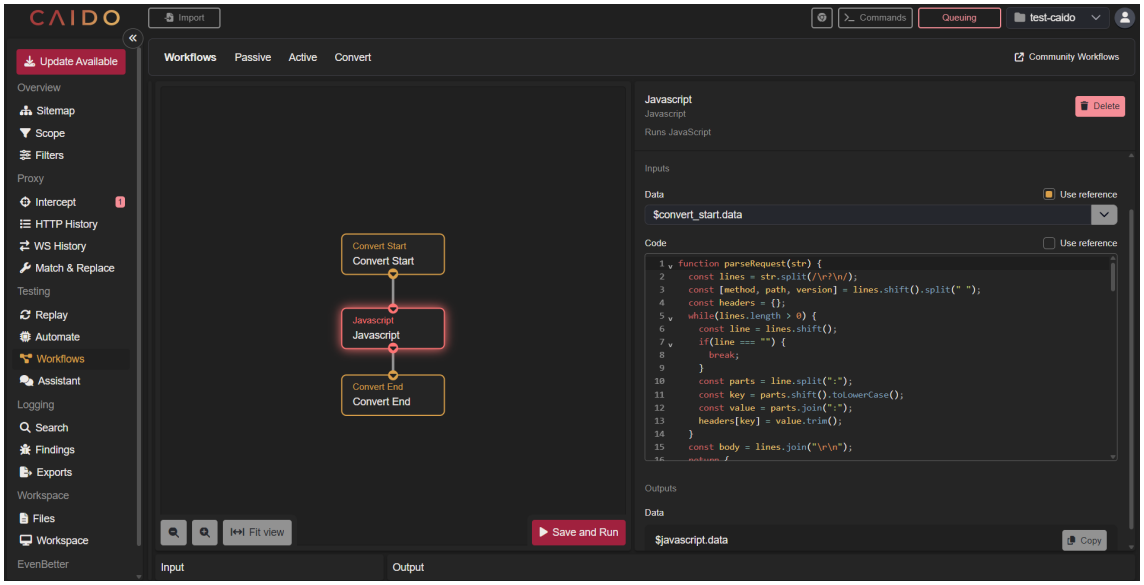

Figura 3.14: Caido tool showing the workflow to generate CSRF PoCs.

In particular, the *Javascript* node of the workflow contains the script that processes the data (the intercepted HTTP POST request) captured in the *Convert Start* node. The input data is contained in the variable *$convert_start.data*, and this node parses and structures the data, generating the CSRF PoC. The PoC consists of an

HTML form that mimics the original POST request, with the request body encoded as URL-encoded[16] data.

The figure below shows the HTTP POST request provided as input to the workflow, alongside the generated output. In this case, I modified the username value.

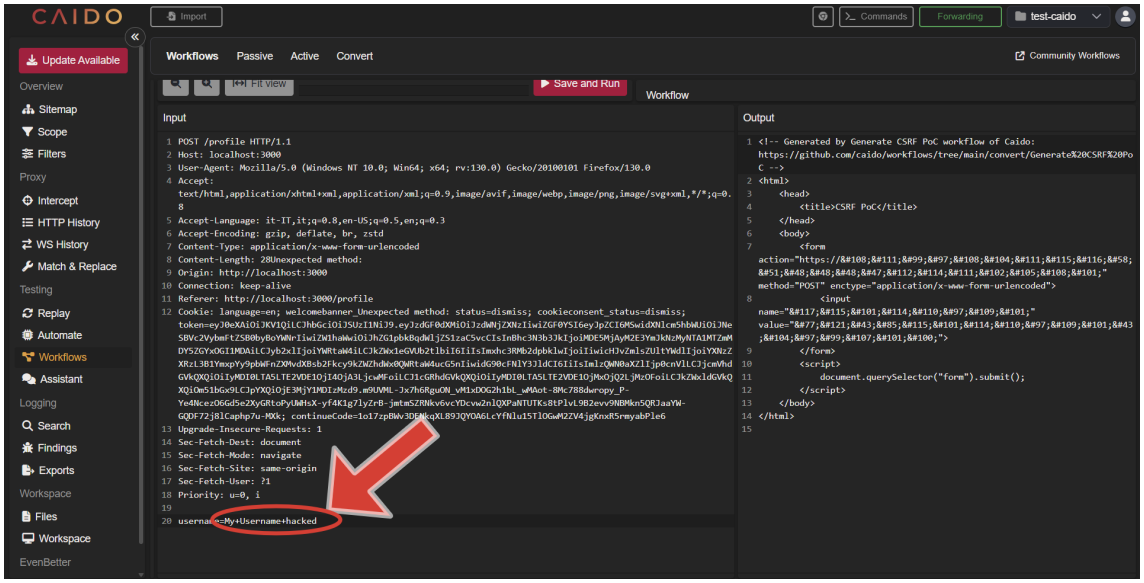

Figura 3.15: Caido tool showing the input HTTP POST request, with the modified username, and the generated CSRF PoC using the workflow.

At this point, the CSRF attack could be executed. If the targeted user is tricked into opening the malicious HTML file in the **same browser** where they are logged into the Juice Shop app, the username would be changed automatically, without the user's knowledge.

The result is shown below:

[16]URL-encoded data refers to a way of formatting data in a URL by encoding special characters (such as spaces or symbols) as a series of characters starting with % followed by two hexadecimal digits. For example, a space is encoded as %20. This encoding is often used in query strings or the body of HTTP POST requests to safely transmit data over the web.

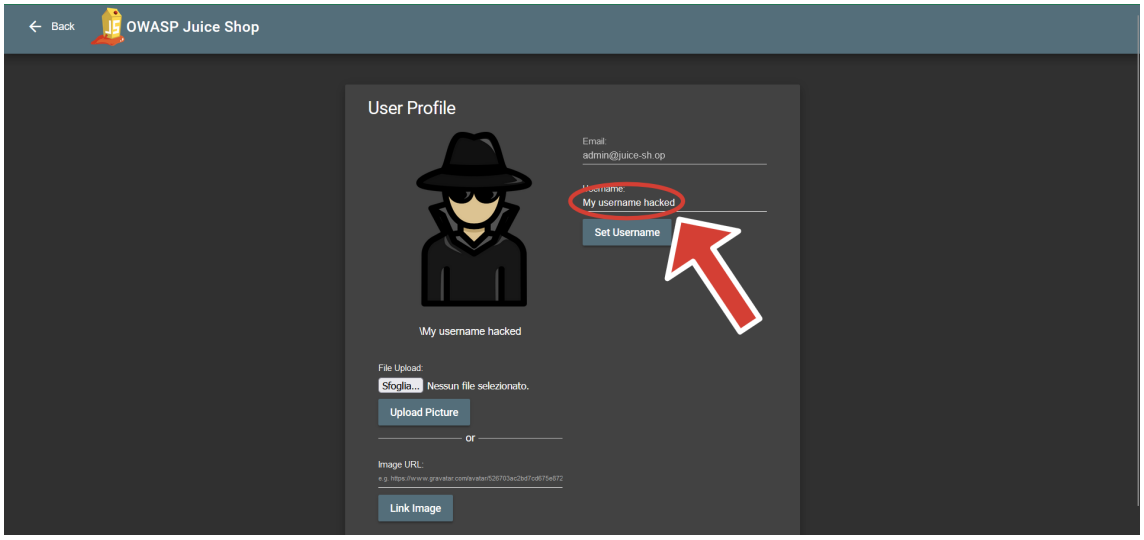

Figura 3.16: Juice Shop profile showing the hacked username, meaning the CSRF was succesful

3.5.3 Explanation of Attack Mechanism

This CSRF attack was possible because, when a user is authenticated, the browser automatically sends the session cookie with every request to the website. This cookie allows the server to recognize the user and authorize actions, but it is sent regardless of whether the request is legitimate or malicious.

3.5.4 Possible Solutions to Prevent CSRF Attacks

To prevent such attack, it is not enough to rely solely on the session cookie for user authentication. Several mitigation techniques can be employed:

- **Anti-CSRF Tokens:** A common mitigation technique is to include an *Anti-CSRF* token in every state-changing request (such as form submissions). This token is generated by the server and validated on each request, ensuring that the request comes from the legitimate user interface and not from an external or malicious source.
- **SameSite Cookie Flag:** Set the *SameSite* attribute on cookies to limit when cookies are sent. This makes it more difficult for an external site to send a request with the authenticated user's cookies. The *SameSite* attribute can be set to "Strict" or "Lax" depending on the level of protection required.
- **Check Referer Header:** Validate the Referer header in requests to ensure that they originate from your own site. While this method is not foolproof, it can help in identifying and blocking unauthorized requests.

- **Custom Headers:** Require custom headers in state-changing requests. Since custom headers cannot be set by third-party sites, this approach helps ensure that requests are coming from legitimate sources.
- **User Interaction:** Implement mechanisms that require user interaction, such as re-authentication or confirmation steps, for sensitive actions. This can help mitigate the risk of unauthorized actions being performed without the user's knowledge.

Bibliografia

[1] Caido. Generate csrf poc - json file. https://github.com/caido/workflows/tree/main/convert/Generate%20CSRF%20PoC, 2024.

[2] Caido Security. Caido - security tool for web application testing. https://caido.io/.

[3] EnterpriseDB. Protecting against sql injection.

[4] OWASP Foundation. Juice shop - an intentionally insecure web application for security training. https://juice-shop.herokuapp.com/#/, 2017.

[5] OWASP Foundation. Owasp vulnerable web applications directory. https://owasp.org/www-project-vulnerable-web-applications-directory/, 2024.

[6] Google. Xss game level 1. xss-game.appspot.com, 2015.

[7] Caido Security. Introducing httpql. https://blog.caido.io/introducing-httpql, 2024.

[8] Sucuri. What is a brute force attack? https://sucuri.net/guides/what-is-brute-force-attack/, 2019.