HackVent 2017

Write-Up

Stjubit

Day 01: 5th anniversary

time to have a look back

HV17 - 5YRS - 4evr - taken from 2014-12-01 - taken from 2015-12-01 - taken from 2016-12-01

The flag is constructed by parts of the flags from the last years.

- Flag of ch01 in 2014 → HV14-BAAJ-6ZtK-IJHy-bABB-YoMw
- Flag of ch01 in 2015 → HV15-Tz9K-4JIJ-EowK-oXP1-NUYL
- Flag of ch01 in 2016 → HV16-t8Kd-38aY-QxL5-bn4K-c6Lw

So this years flag is: **HV17-5YRS-4evr-IJHy-oXP1-c6Lw**

Day 02: Wishlist

The fifth power of two

Something happened to my wishlist, please help me.

Obviously, the wishlist is encoded with Base64, because the last 2 characters are **"==".** If we decode it, we get another Base64 encoded text. So we have to decode the file again and again until we get the flag. I used the following Python script to accomplish that:

```python
import base64

next_file = open("input.txt", "r")

for i in range(1, 100):
    content = next_file.read()
    next_file.close()

    content = content.replace("\n", "")
    content = content.replace(" ", "")

    decoded_content = base64.b64decode(content)

    write_file = open("decoded_" + str(i) + ".txt", "w")
    write_file.write(decoded_content)
    write_file.close()

    next_file = open("decoded_" + str(i) + ".txt", "r")
```

Decoding step #32 shows our flag: **HV17-Th3F-1fth-Pow3-r0f2-is32**

Day 03:

Strange Logcat Entry

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!

There are two lines in the logcat file, which are interesting:

```
1. 11-13 … I DEBUG: FAILED TO SEND RAW PDU MESSAGE
2. 11-13 … DEBUG: I
   07914400000000F001000B913173317331F300003AC7F79B0C52BEC52190F37
   D07D1C3EB32888E2E838CECF05907425A63B7161D1D9BB7D2F337BB459E8FD1
   2D188CDD6E85CFE931
```

So the application tried to send a PDU message, but it failed. PDU messages are used for sending and receiving SMS messages. The string in the DEBUG output has the format of a PDU message, so we can be sure that the application failed to send this SMS.

I used the following tool to decode the text:
https://www.diafaan.com/sms-tutorials/gsm-modem-tutorial/online-sms-pdu-decoder/

Message:
Good Job! Now take the Flag: HV17-th1s-isol-dsch-00lm-agic

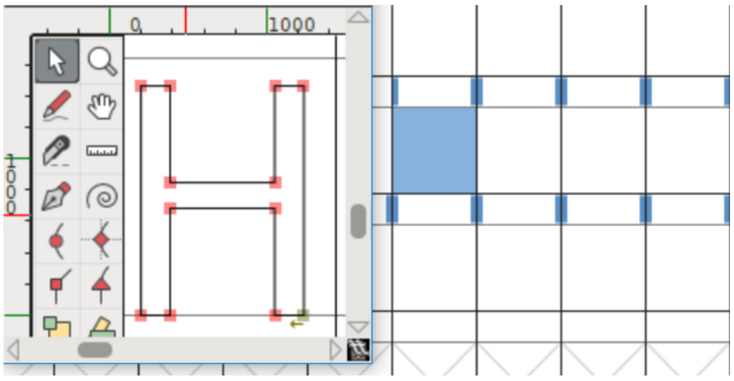
Day 04: HoHoHo

NOTE: New easyfied attachment available

Santa has hidden something for you

First, I used iText Rups to analyse the given PDF file, but without success. I extracted all the images, text and so on and tried several things, but that was not part of the challenge. After that, I searched for another tool to analyse the streams and finally found PDFStreamDumper. With it I was able to extract a **SFD SplineFontDB** file.

The next step I made was to view the font. **FontForge** is a free and open source font editor I used for this purpose. There are not many SFD Font Viewers, but this one worked perfectly. From Unicode 0080 to 009C are “invisible” font characters. Viewing them in the editing window revealed the flag character for character. Here is a screenshot of the first letter of the flag in FontForge.



The flag is: HV17-RP7W-DU6t-Z3qA-jwBz-jltj

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

The encoded flag consists of 6 lines and the flag format has 6 parts, too. Therefore, we can assume that every line represents a part of the flag. What we already know is that the first part decoded has to be 'HV17'. First, I thought that it had to be some kind of OTP. I tried to XOR the encrypted first line with the given plaintext (HV17) and got the OTP Key. The next thing I did was to try the same key for the other parts of the flag, but obviously, it was not OTP. So I searched for other encryption algorithms and hashing functions until I found out that it was CRC32 (just a lot of googling..).

CRC32('HV17') = 0x69355F71 = first line

The challenge was it to reverse the other CRC32 encrypted parts of the flag and it's possible:

"A CRC32 is only reversible if the original string is 4 bytes or less.".

I've used the following Python script to reverse the parts of the flag.
https://github.com/theonlypwner/crc32

Example:

```
INPUT:   python crc32.py reverse 0x69355f71
OUTPUT:  4 bytes: {0x48, 0x56, 0x31, 0x37}    (=HV17)
```

<u>The flag is:</u> HV17-7pKs-whyz-o6wF-h4rp-Qlt6

Day 06: Santa's journey

Make sure Santa visits every country

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

The challenge starts at a site where a QR-Code is displayed.

Every reload of the page displays a different QR-Code. That means that it’s not static. The decoded QR-Code always represents a country.

My thoughts on that challenge were that we just have to make requests to the webpage until we get the flag. Therefore, I wrote the following Python script:

```python
import qreader
import requests
from PIL import Image
from io import BytesIO

while True:
  r = requests.get("http://challenges.hackvent.hacking-lab.com:4200")

  img = Image.open(BytesIO(r.content))
  data = qreader.read(img)

  if "HV17" in data:
    print "Found egg: " + str(data)
    break
  else:
    print data
```

The flag is: HV17-eCFw-J4xX-buy3-8pzG-kd3M

Day 07: i know ...

... what you did last xmas

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

Please help us to recover it:

The file “SANTA.FILE” is in the attachments, which is a ZIP file. It contains the file “SANTA.IMA”. The command “file” produces the following output:

```
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
```

It looks like it’s a kind of image file. I used “**Autopsy**” to analyse the image. I was able to extract a file called “SANTA.PRIV” in the image. Again, here is the output of “**file**”:

```
SANTA.PRIV: MS Windows registry file, NT/2000 or above
```

Now we have a Windows Registry file we have to analyse. I simply tried to find the flag as text, because a reg file (almost) only contains plaintext.

```
# strings SANTA.PRIV | grep -i HV17
```

Well, here is **the flag**:
C:\Hackvent\HV17-UCyz-0yEU-d90O-vSqS-Sd64.exe

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...

Obfuscation, nice! The given file consists of two parts: “True” and “1337”. Let’s start with “True”:

True represents a Boolean and Boolean can only hold two values: 1 and 0. 1 stands for True and 0 for False. Therefore, the application just calculates 1+1+1+1… to represent the ASCII code for a character. We just have to print out the result (code beautified):

```
A=chr
__1337=exec
SANTA=input
FUN=print

def _1337(B):
  return A(B//1337)
```

Now we know what the 1337 part stands for. Again, we just have to run it and get the output:

```
C=SANTA("?")

if C=="1787569":
  FUN(''.join(chr(ord(a) ^ ord(b)) for a,b in zip("<non-printable-
      characters","31415926535897932384626433832")))
```

The flag is: HV17-th1s-ju5t-l1k3-j5sf-uck!

Day 09: JSONion

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

We get a JSON file as attachment. It contains 3 fields:

- op (=map)
- mapTo
- mapFrom

It was clear that we have to change the characters from **mapFrom** according to the key at **mapTo**. What we get is another JSON text with a different operation field. In the end, there were 6 operations necessary to get the flag. The other five are described below.

Operation	Description
gzip	• decode Base64 and unzip the gzip content
b64	• decode Base64
nul	• do nothing; just get the content and continue with the next step
xor	• Bitwise XOR
rev	• Reverse the text

Sometimes the decoded steps resulted in two other steps. If you only decoded the first step until the end, you didn't get the flag. It would have been better to implement it using recursive functions, but I had no time for it on that day. So, my code is one big mess!

```python
import json
import gzip
import base64
import binascii
import os
import fnmatch
from pprint import pprint


for i in range(0, 100):

  for file in os.listdir('.'):
    if fnmatch.fnmatch(file, "step_" + str(i) + "_*.json"):

      with open(file) as curr_file:
        json_content = curr_file.read()[1:-1]

      is_array = False

      try:
        curr_json = json.loads(json_content)
      except:
        curr_json = json.loads("[" + json_content + "]")
        is_array = True

      if is_array == True:
        iterator_length = len(curr_json)
      else:
        iterator_length = 1
      # iterate over every item in "JSONARRAY"
      for jp in range(0, iterator_length):

        if is_array == True:
          curr_json_obj = curr_json[jp]
        else:
          curr_json_obj = curr_json

        # BEGIN OPERATIONS
        operation = curr_json_obj["op"]
        result = ""

        if operation == "map":
          print "Step " + str(i) + " --> MAP"

          # PARAMS
          map_to = curr_json_obj["mapTo"]
          map_from = curr_json_obj["mapFrom"]
          content = curr_json_obj["content"]

          for ch in content:
            position = map_from.index(ch)
            result = result + str(map_to[position])

        elif operation == "gzip":
          print "Step " + str(i) + " --> GUNZIP"

          # PARAMS
          content = curr_json_obj["content"]
          gunzip_content = base64.b64decode(content)

          gz_file = open("step_" + str(i) + "_" + str(jp) + ".gz", "w")
          gz_file.write(gunzip_content)
          gz_file.close()

          with gzip.open("step_" + str(i) + "_" + str(jp) + ".gz", "rb")
as file_to_decompress:
            result = file_to_decompress.read()

        elif operation == "b64":
          print "Step " + str(i) + " --> Base64"

          # PARAMS
          content = curr_json_obj["content"]

          result = base64.b64decode(content)

        elif operation == "nul":
          print "Step " + str(i) + " --> NUL"

          # PARAMS
          content = curr_json_obj["content"]

          result = content

        elif operation == "xor":
          print "Step " + str(i) + " --> XOR"

          # PARAMS
          content = base64.b64decode(curr_json_obj["content"])
          mask = base64.b64decode(curr_json_obj["mask"])
          for ch in content:
            result = result + chr(ord(ch) ^ ord(mask))

        elif operation == "rev":
          print "Step " + str(i) + " --> REVERSE"

          # PARAMS
          content = curr_json_obj["content"]

          result = content[::-1]

        out_f = open("step_" + str(i + 1) + "_" + str(jp) + ".json", "w")
        out_f.write(result)
        out_f.close()
```

The flag is:
[{"op":"flag","content":"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!

The challenge is it to win 100 times against the AI in Tic Tac Toe without losing. The hard part of it was to find a good python script online ^^.

There is not really much more to say. Here is my final code:

```
from pwn import *


# The 'pairs' list is used to check and see if anyone has won the game.
# -------|           Verticals            |           Horizontals
|      Diagonals      |
pairs = ([0, 3, 6], [1, 4, 7], [2, 5, 8], [0, 1, 2], [3, 4, 5], [6, 7,
8], [0, 4, 8], [2, 4, 6])

board = [" ", " ", " ", " ", " ", " ", " ", " ", " "]

# 'Aiturn' lets the AI know how many turns it has taken.
aiturn = 1



def parse_rows(rows):
  parsed = []

  for i in range(0, len(rows)):
    rows[i] = rows[i].replace(" ", "")
    rows[i] = rows[i].replace("|", "")
    rows[i] = rows[i].replace("\n", "")
    rows[i] = rows[i].replace("*", "-")

    #print rows[i]

    for k in range(0, 3):
      parsed.append(rows[i][k])

  return parsed


def read_round():
  conn.readline()
  conn.readline() #  -------------
  first_row = conn.readline() # ROW 1
  conn.readline() #  -------------
  conn.readline() #  -------------
  second_row = conn.readline() # ROW 2
  conn.readline() #  -------------
  conn.readline() #  -------------
  third_row = conn.readline() # ROW 3
  conn.readline() #  -------------

  rows = [ first_row, second_row, third_row ]
  return parse_rows(rows)
```

```
def find_board_diff(old_board, new_board):
  diff = -1
  for i in range(0, 9):
    if old_board[i] != new_board[i]:
      diff = i
      break

  return diff


def check_ending():
  game_ending = conn.readline() # \n

  if "lost" in game_ending or "win" in game_ending in game_ending or
"Invalid" in game_ending:
    print game_ending
    return True
  else:
    return False


def checkforwin(board):
  for x in pairs:
    zero = board[x[0]]
    one = board[x[1]]
    two = board[x[2]]
    if zero == one and one == two:
      if zero == "X":
        return "AI_WIN"
      if zero == "O":
        return "HUMAN_WIN"
    else:
      filledspaces = 0
      for i in range(8):
        if board[i] != "-" and board[i] != "_":
          filledspaces += 1
        if filledspaces == 8:
          return "DRAW"
  return "INGAME"


def isWin(board):
    """
    GIven a board checks if it is in a winning state.
    Arguments:
          board: a list containing X,O or -.
    Return Value:
           True if board in winning state. Else False
    """
    ### check if any of the rows has winning combination
    for i in range(3):
        if len(set(board[i*3:i*3+3])) is  1 and board[i*3] is not '-':
return True
    ### check if any of the Columns has winning combination
    for i in range(3):
       if (board[i] is board[i+3]) and (board[i] is  board[i+6]) and
board[i] is not '-':
           return True
    ### 2,4,6 and 0,4,8 cases
    if board[0] is board[4] and board[4] is board[8] and board[4] is not
'-':
        return  True
    if board[2] is board[4] and board[4] is board[6] and board[4] is not
'-':
        return  True
    return False


def nextMove(board,player):
    """
    Computes the next move for a player given the current board state and
also
    computes if the player will win or not.
    Arguments:
        board: list containing X,- and O
        player: one character string 'X' or 'O'
    Return Value:
        willwin: 1 if 'X' is in winning state, 0 if the game is draw and
-1 if 'O' is
                    winning
        nextmove: position where the player can play the next move so
that the
                         player wins or draws or delays the loss
    """
    ### when board is '---------' evaluating next move takes some time
since
    ### the tree has 9! nodes. But it is clear in that state, the result
is a draw
    if len(set(board)) == 1: return 0,4

    nextplayer = 'X' if player=='O' else 'O'
    if isWin(board) :
        if player is 'X': return -1,-1
        else: return 1,-1
    res_list=[] # list for appending the result
    c= board.count('-')
    if  c is 0:
        return 0,-1
    _list=[] # list for storing the indexes where '-' appears
    for i in range(len(board)):
        if board[i] == '-':
            _list.append(i)
    #tempboardlist=list(board)
    for i in _list:
        board[i]=player
        ret,move=nextMove(board,nextplayer)
        res_list.append(ret)
        board[i]='-'
    if player is 'X':
        maxele=max(res_list)
        return maxele,_list[res_list.index(maxele)]
    else :
        minele=min(res_list)
        return minele,_list[res_list.index(minele)]


# ============= MAIN =================
conn = remote("challenges.hackvent.hacking-lab.com", 1037)

# Init game
conn.recvuntil("Press enter to start the game", drop=True)
conn.send("\n")



# === GAME LOGIC ===
round_counter = 1
win_counter = 0

while win_counter < 105:
  board = read_round()

  #print "PLAYER MOVED: " + str(board)

  move_index = nextMove(board, "X")
  aiturn = aiturn + 1
  board[move_index[1]] = "X"

  #print "AI MOVE:      " + str(board) + " --> " + str(move_index[1])

  try:
    conn.recvuntil("Field: ", drop=True)
    conn.send(str(move_index[1] + 1) + "\n")
  except:
    print "Round #" + str(round_counter) + ": " + conn.readline() # LOST
    break

  exit_win = checkforwin(board)
  if exit_win != "INGAME":
    board = read_round()
    board = read_round()
    #print board
    ending_string = conn.readline()

    if "Congratulations" in ending_string:
      win_counter = win_counter + 1

    if win_counter >= 100:
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")
      print conn.recvline().encode("hex")


    print "Round #" + str(round_counter) + ": " + ending_string
    round_counter = round_counter + 1

    #print conn.recvuntil("again", drop=True)

    conn.send("\n")
    conn.readline()
    #conn.readline()
    #break
  else:
    board = read_round()
```

The flag is: HV17-y0ue-kn0w-7h4t-g4me-sure

Day 11: Crypt-o-Math 2.0

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

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

find "a" to get your flag.

In this challenge we have to transform a formula so that we can calculate a. It was very hard, because we can not just get the remainder of a calculation (modulo). The remainder isn't only the result of one calculation of course. However, it is still possible to calculate a in our case → **Modular Multiplicative Inverse**. The following site was a big help to me: https://math.stackexchange.com/questions/684550/how-to-reverse-modulo-of-a-multiplication

Basically, we just have to make this calculation to get a:

```
    a = (modinv(b, p) * c) % p
```

My python script for solving the challenge:

```python
import binascii

c = 0x559C8077EE6C7990AF727955B744425D3CC2D4D7D0E46F015C8958B34783
p = 0x9451A6D9C114898235148F1BC7AA32901DCAE445BC3C08BA6325968F92DB
b = 0xCDB5E946CB9913616FA257418590EBCACB76FD4840FA90DE0FA78F095873

# Extended euclidian algorithm
def xgcd(a,b):
    prevx, x = 1, 0
    prevy, y = 0, 1
    while b:
        q = a/b
        x, prevx = prevx - q*x, x
        y, prevy = prevy - q*y, y
        a, b = b, a % b
    return a, prevx, prevy

# Modular Multiplicative Inverse
def modinv(a, m):
    g, x, y = xgcd(a, m)
    return x % m

u = modinv(b,p)
a = hex((u * c) % p)

print a
```

The flag is: HV17-zQBz-AwDg-1FEL-rUE9-GKgq

Day 12: giftlogistics

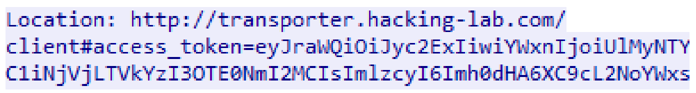
countercomplete inmeasure

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.

I started with analysing the traffic. I filtered for HTTP packets (port 80) and then followed the TCP streams. Stream #25 looks interesting. We can see 2 requests and 2 responses to:

- /**authorize** with the client_id and Session Cookie
- /**login** with a Session Cookie
- /**j_spring_security_check** with the username, password and a CSRF Token
- /**authorize** with the access token as response and a redirect to *transporter.hacking-lab.com:*



```
Location: http://transporter.hacking-lab.com/
client#access_token=eyJraWQiOiJyc2ExIiwiYWxnIjoiUlMyNTY
C1iNjVjLTVkYzI3OTE0NmI2MCIsImlzcyI6Imh0dHA6XC9cL2NoYWxs
```

There is also another very interesting packet with a request to **/giftlogistics/.well-known/openid-configuration**. We can make the same request to the challenge website to get the file. We now know that the Authentication was implemented with OAuth and JWT. The Authentication Endpoints are listed in the openid-configuration file. The next step I made was to call all the endpoints. I thought that I have to log in to santas account until I found the userinfo endpoint. This site was very useful:

https://identityserver.github.io/Documentation/docsv2/endpoints/userinfo.html

I just reconstructed the userinfo request with Santas token I got from the traffic and received the flag:

```
curl "http://challenges.hackvent.hacking-
lab.com:7240/giftlogistics/userinfo" -H "Authorization: Bearer
<Token>
```

The flag is:

```
{
   "sub":"HV17-eUOF-mPJY-ruga-fUFq-EhOx",
   "name":"Reginald Thumblewood",
   "preferred_username":"santa"
}
```

Day 13: muffin_asm

As M. said, kind of a different architecture!

ohai \o/

How about some custom asm to obsfucate the codez?

We get a Python script with a function **run()**. The function is passed a long Hex string, which are the Assembler Instructions, obviously. There is also a instruction array where we can see which word represents which instruction. So I just had to change the code a little bit to print out the function name and the arguments passed to the function to get the disassembly.

```
…
rchr 73
cmp 73 72
jne 19335 Jump = False
…
```

As we can see in the disassembly, our input (character 'I') is compared to the character 'H' (Ascii<->DEC). The jump is not taken, because the first character of the flag is wrong. It's clear what we have to do now: patch the cmp function.

```
def _cmp(r1, r2):
  r1 = r2
  print chr(r[r2])

  f[0] = (r[r1] == r[r2])
```

The flag is: HV17-mUff!n-4sm-!s-cr4zY

Day 14: Happy Cryptmas

Todays gift was encrypted with the attached program. Try to unbox your xmas present.

Flag:

```
7A9FDCA5BB061D0D638BE1442586F3488B536399BA05A14FCAE3F0A2E5F268F2F314
2D1956769497AE677A12E4D44EC727E255B391005B9ADCF53B4A74FFC34C
```

The given file is a Mach-O executable. Here is the output of **#file hackvent**

```
hackvent: Mach-O 64-bit x86_64 executable
```

I jumped directly to reversing instead of running it, because I hate Mach-O executables. I used IDA Pro to decompile the **_main** function. The executable has a few imports to functions of libgmp. Libgmp is mainly used for cryptography and working with big numbers/objects, like in the given application.

There are two calls to __gmpz_init_set_str:

- `__gmpz_init_set_str(&v11, "F66EB887F2B8A620FD03C7D…", 16LL);`
- `__gmpz_init_set_str(&v10, "65537", 10LL);`

It already looks like RSA and it was. After the initialization of these two constants, the following condition has to be true:

```
if ( __gmpz_cmp(&v9, &v11) > 0 )
```

This basically checks if v9 is smaller than v11. v9 is the argument passed to the executable (the string to encrypt). Therefore, the plaintext has to be smaller than the key.

After that, the following command is executed:

```
__gmpz_powm(&v8, &v9, &v10, &v11);
```

This is exactly the default RSA formula to encrypt a message:

$$c \equiv m^e \pmod{N}$$

Now we know what to do to solve the challenge. We must break RSA. v11 is our N in the formula. Luckily, n has already be factored and is in the factorization database (www.factordb.com). Consequently, we know **p** and **q** and can calculate the private key **d** with the following python script.

```
C = 6422364120026732410681865634854254183147125246054233723304764504977784
0830688444841588552434965602224626422836832323525577199101442006761403538
743492592460
E = 65537
n = 1290671746434809226595641021086028268426120023964931443682266661646O7
40520052403025774625130601134473716449192270880280937288228652858915015O4
4165744901457
p = 181329857570381356 91
q = 711781150511215724435363874088486910075853913118425049972912826148221
2974830650079671924316134224096940540647556585642437215555325358 27

phi = (p-1) * (q-1)
d = extgcd(e, phi)[1]
d = d % phi
```

Now we only have to decrypt our message with the calculated private key.

$$m \equiv c^d \pmod{N}$$

```
m = powermod4(c, d, n)
print str(hex(m))[2:-1].decode("hex")
```

The flag is: HV17-5BMu-mgD0-G7Su-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.

The link http://challenges.hackvent.hacking-lab.com:3958/gallery/bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw contains a Base64-looking string, which identifies the gallery of a user, obviously. To solve the challenge, we have to find out how this string is constructed. The b64 decoded identifier string has no printable characters, so I thought that it was a hash. I used the tool **hash-identifier** to see which hash algorithm it could be. At this point, I wrote a python script, which calculates the SHA1 hash of every possible combination of the user data in the leaked list of all users. However, I had no luck cracking it.

It was weird that there were no errors decoding the string with base64, so I concentrated on this one. First, I saved the result as hex with the command:

echo "bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw" | base64 -d | hexdump

The hex values looked like a hash, so I tried to hash parts of the user data with https://www.onlinehashcrack.com/hash-generator.php. Success! The SHA256 hash of the email of Danny is part of the base64-decoded hexdump, but the last 10 bytes are different.

SHA256("Danny.Dixon@sunflower.org"):

```
6E772A62E85D4157B2F6898A03AB40162E2B7A9D    7E143F91B43E07FFC7ED5A2C
```

Base64("bncqYuhdQVey9omKA6tAFi4rep1FDRtD4H8ftWiw")->HEX:

```
6E772A62E85D4157B2F6898A03AB40162E2B7A9D    450D1B43E07F1FB568B0
```

Now the guessing part of the challenge came in. I tried several things to find out where the last 10 bytes were coming from. In the end, the Base64 value of the SHA256-Hash contains the given identifier string:

`bncqYuhdQVey9omKA6tAFi4rep1+FD+RtD4H/8ftWiw=`

Now we are able to construct the URL to Thumper's gallery, where the flag is stored. The right Thumper is *Thumper.Lee@gmx.com*. The SHA256 hash of it is:

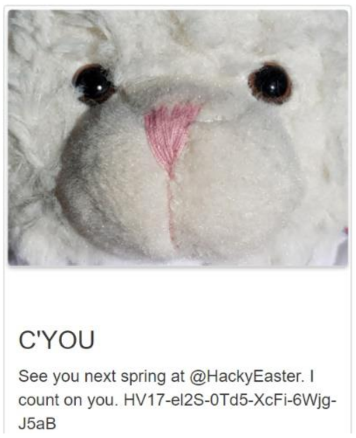
`DFBA8A61530036721D2765761036DEAC698CCFD2734B5A5F44B5566882AE043C`

…encode it with Base64:

`37qKYVMANnIdJ2V2EDberGmMz9JzS1pfRLVWaIKuBDw=`

Now we just have to delete all non string.ascii_letters and non string.digits characters from the base64 string like in Danny's identifier.

http://challenges.hackvent.hacking-lab.com:3958/gallery/37qKYVMANnIdJ2V2EDberGmMz9JzS1pfRLVWaIKuBDw



C'YOU

See you next spring at @HackyEaster. I count on you. HV17-el2S-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.

When we connect to the server, we get a first hint of what we have to do in order to get the flag:

```
The flag is stored super secure in the function SANTA!
```

So we need to somehow call the function SANTA to get the flag. Like in almost every other challenge, we have to play around a little bit and try things out to get an overview and idea of how we can exploit the application. I found out that only this characters are allowed:

```
123790 acdeilnoprstv._"'()[]+
```

So, santa is allowed, but the problem is that it's not possible to enter upper case letters, because we have to call SANTA, not santa. At least not that easy. I also found out the following functions are allowed to call:

- `print(a)`
- `a.__init__(), a.__class__, a.__doc__, a.__dir__()`
- `eval`

We can execute commands with eval(cmd) and we can also put together strings. My idea was to get the blacklisted letters from an "external" source like a help text and then execute the function upper() on santa, but that didn't work because the function upper() is explicitly blacklisted. Therefore, I had to find another way to make santa uppercase → **swapcase()**.

The 'w' is a blacklisted character, but I got it from the integer documentation text (__doc__). We are now able to call SANTA(), but there is no flag for us:

```
>>> a = "santa"
>>> a = eval(eval("a.s"+str(eval("denied"))[23]+"apcase()")+"()")
>>> a = print(a)
No flag for you!
```

When passing 1 as argument, we get the following error:

```
zip argument #2 must support iteration
```

Now we know that we get the flag if we pass the right argument to the zip() function. At this point I wrote a python script, which made it much easier to pass an argument to the function.

```python
from pwn import *

while True:
  argument = raw_input("Param: ")[:-1]

  conn = remote("challenges.hackvent.hacking-lab.com", 1034)
  conn.recvuntil(">>> a = ")
  conn.send('"santa"\n')
  conn.recvuntil(">>> a = ")

  cmd = 'eval(eval("a.s"+str(eval("denied"))[23]+"apcase()")+"(\'' +
argument + '\')")\n'
  conn.send(cmd)
  conn.recvuntil(">>> a = ")
  conn.send('print(a)\n')

  response = conn.recvuntil(">>> a = ").split("\n")[0]
  if response == "santa":
    print "DENIED!"
  else:
    print "Response: " + str(response)
```

After trying around a little bit I found the right 'key' to get the flag:

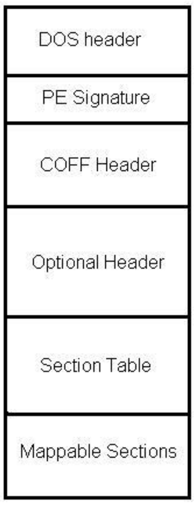
```
1337133713371337133713371337
```

The flag is: HV17-J41l-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?



There is a Portable Executable file given with corrupt header information. Our goal is it to change the header values in order to be able to execute it. The PE header consists of the following fields:

I used the tools **NikPEViewer** and **PEView** to analyse the given binary. This are the changes, which had to be done to make the application run:

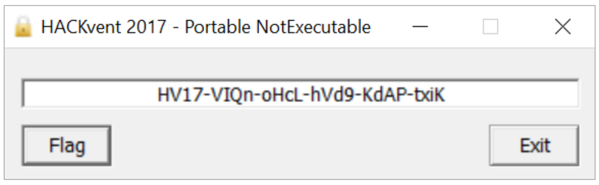
Step 1. MS = MZ
Step 2. Offset to exe header 20 = 40
Step 3. Delete N in PNE to PE
Step 4. Add 00 after PE because of deletion
Step 5. Change section from 06 to 04

Now we are able to run the application, but the flag is wrong. It is wrong because there is another field we have to change. Later, the following hint got released:

take the hint in the file serious, the black window should not appear

There is another field called subsystem we have to change. It identifies the Subsystem that will be invoked to run the executable. It was set to 3. That means that the executable runs as a command line tool and that's why the black window appeared.

Step 6. Change the subsystem from 03 to 02 (GUI, not CUI)


HACKvent 2017 - Portable NotExecutable

HV17-VIQn-oHcL-hVd9-KdAP-txiK

Hidden #1d0n7kn0w

The url to every challenge looks like this:

https://hackvent.hacking-lab.com/challenge.php?day=24

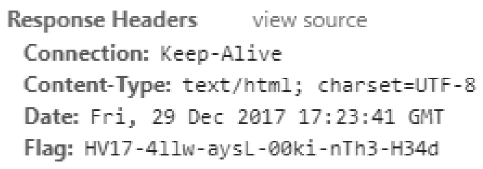
If we change it to 25, we get the following error:

```
The resource (#1959) you are trying to access, is not (yet) for your
eyes.
```

The resource number changes with a different day number. To get the flag, the resource number has to be 0 (or nothing), which happens at day 1984:

```
The resource you are trying to access, is hidden in the header.
```

As the websites told us, the flag is in the response header:



```
Response Headers      view source
  Connection: Keep-Alive
  Content-Type: text/html; charset=UTF-8
  Date: Fri, 29 Dec 2017 17:23:41 GMT
  Flag: HV17-4llw-aysL-00ki-nTh3-H34d
```

Hidden #1d0n7c4r3

It is always a good idea to look into the robots.txt file when searching for easter eggs. It contains:

```
We are people, not machines
```

Googling this sentence leads us to the following page:

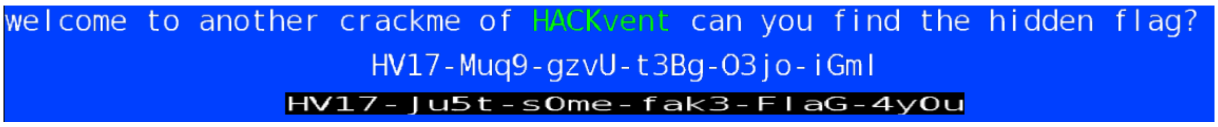
http://humanstxt.org/de

The flag is: HV17-bz7q-zrfD-XnGz-fQos-wr2A **on page:**

https://hackvent.hacking-lab.com/humans.txt

Hidden #h0w_4m_1_5upp053d_70_kn0w?

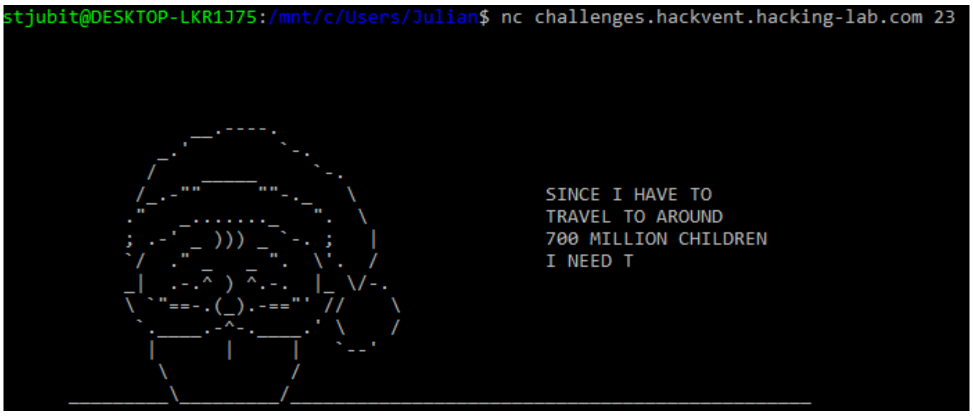
This one was hidden in challenge #18. The only thing we had to do was to run the given attachment. Reversing the EBOOT.BIN file with IDA revealed that it is a PS3 “game”. I used RPCS3 to run the game. The flag is displayed.



```
welcome to another crackme of HACKvent can you find the hidden flag?
                 HV17-Muq9-gzvU-t3Bg-O3jo-iGml
                 HV17-Ju5t-sOme-fak3-FlaG-4yOu
```

Hidden #1_c4n_n07_7h1nk_0f_4ny7h1n6_m0r3

Challenge #10 was the first on challenges.hackvent.hacking-lab.com. I simply ran a port scan on that Domain and found out that the port 23 is open. I connected to it with netcat:

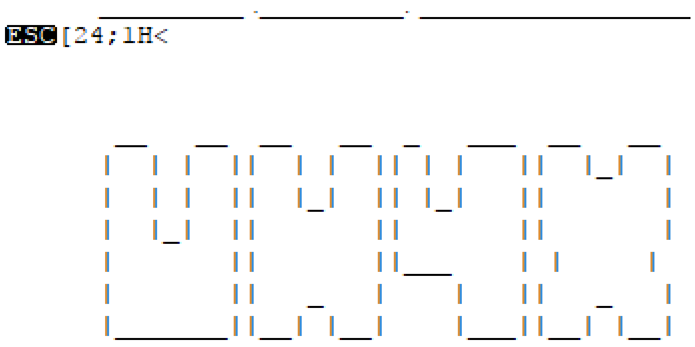


```
stjubit@DESKTOP-LKR1J75:/mnt/c/Users/Julian$ nc challenges.hackvent.hacking-lab.com 23

             __.----.
           .'        `-.
        _.'             `-.
       /     ______        \
      /_.-""      ""-._     \
      ."               ".    \
     .    _.......__    .     |
     ; .-'_ ))) _   `-. ;     |
     `/   ."        ".  \'.  /
     _|   .-.^ ) ^.-.  |_ \/-.
     \  `"==-.(_).-=="' //     \
      `.______.-^-.______.'\    /
        |         |       |  `--'
         \                /
__________\______________/____________________________________

                                  SINCE I HAVE TO
                                  TRAVEL TO AROUND
                                  700 MILLION CHILDREN
                                  I NEED T
```

There is a text showing very fast, so I saved the netcat output to a file.



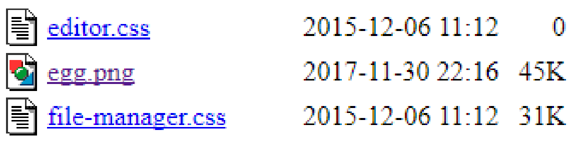
```
ESC[24;1H<

 __   __  __   __  _   ___  __   __
|  | |  ||  | |  || | |   ||  |_|  |
|  | |  ||  |_|  || |_|   ||        |
|  |_|  ||       ||       ||        |
|       ||       ||___    | |     |
|       ||   _   |    |   ||   _   |
|_______||__| |__|    |___||__| |__|
```

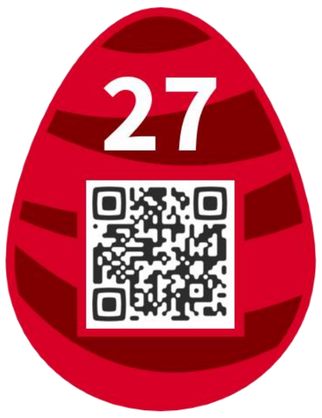
The flag is: HV17-UH4X-PPLE-ANND-IH4X-T1ME

Hidden #wh0_c4r35

I ran dirbuster on hackvent.hacking-lab.com, which revealed some interesting directories and files. In the end, there was a png image in the CSS folder.



editor.css	2015-12-06 11:12	0
egg.png	2017-11-30 22:16	45K
file-manager.css	2015-12-06 11:12	31K



https://hackvent.hacking-lab.com/css/egg.png