

Solved:

Challenge 1	2
Challenge 2	2
Challenge 3	2
Challenge 5	3
Challenge 7	3
Challenge 9	5
Challenge 11	5
Challenge 12	6
Challenge 13	7
Challenge 15	8
Challenge 18	8

Challenge 1

The given image needs to be untwisted before it will be scannable. Using GIMP and swirl counter-clockwise warp transform with size equal the size of the image and hardness set to 40, the following image with scannable QR code was obtained:



Challenge 2

GIF frames show the password using the sign language. First, extract frames from the GIF using GIMP. Then with help of the table from https://en.wikipedia.org/wiki/American_Sign_Language decode manually each sign.

Password: **givemeasign**

Challenge 3

The provided ruby script takes user input, converts it to hex ASCII, interprets the hex string as a hexadecimal number, converts the number to decimal and multiplies it by 5. Then the number is multiplied by 1111...111, that is the number composed of 101 digits "1". The result is then converted to hexadecimal and interpreted as hex ASCII and base64 encoded.

To decrypt the ciphertext, all operations need to be reverted. The following python script will perform the decryption:

[illegible]Plaintext: **n00b_style_crypto**

Challenge 5

First, I noticed that the creator of the .docx file is "SCIpher". Googling it led to an online encoder: SCIpher - A Scholarly Message Encoder (<https://pdos.csail.mit.edu/archive/scigen/scipher.html>). Then, I extracted the text from the document using an online converter (<https://www.onlineconverter.com/docx-to-txt>). The obtained text, however, could not be properly decoded. After playing for a while with the decoder, I observed that whitespace characters also matter in this encoding scheme. After removing all tabs and adjusting newlines I was able to decode the IAPLI document:

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

Challenge 7

First, to get the real source code of the script we have to replace "\$Ax2\$хTT" at the beginning of the third line which will be evaluated to "eval" with "echo". After executing the modified script, we will get the source code:

```
1. z=""
2. ";Cz='s: ';qz='.p';fz='8a';az='e9';Oz='co';Xz='a6';hz='7e';Rz='im';Bz='tp';lz='62';K
   z='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.';n
   z='c7';Qz='r/';ez='d8';cz='ac';gz='12';bz='75';oz='4a';mz='42';jz='6e';dz='b7';
3. if [ $# -lt 1 ]; then
4. echo "Give me some arguments to discuss with you"
5. exit -1
6. fi
7. if [ $# -ne 10 ]; then
8. echo "I only discuss with you when you give the correct number of arguments. Btw: o
   nly arguments in the form /[a-zA-Z] .../ are accepted"
9. exit -1
10. fi
11. if [ "$1" != "-R" ]; then
12. echo "Sorry, but I don't understand your argument. $1 is rather an esoteric stateme
   nt, isn't it?"
13. exit -1
14. fi
15. if [ "$3" != "-a" ]; then
16. echo "Oh no, not that again. $3 really a very boring type of argument"
17. exit -1
18. fi
19. if [ "$5" != "-b" ]; then
20. echo "I'm clueless why you bring such a strange argument as $5?. I know you can do
   better"
21. exit -1
22. fi
23. if [ "$7" != "-I" ]; then
24. echo "$7 always makes me mad. If you wanna discuss with be, then you should bring t
   he right type of arguments, really!"
25. exit -1
26. fi
27. if [ "$9" != "-t" ]; then
28. 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"
29. exit -1
30. fi
31. echo "Ahhhh, finally! Let's discuss your arguments"
32. function isNr() {
33. [[ ${1} =~ ^[0-9]{1,3}$ ]]
34. }
35. if isNr $2 && isNr $4 && isNr $6 && isNr $8 && isNr ${10} ; then
36. echo "..."
37. else
38. echo "Nice arguments, but could you formulate them as numbers between 0 and 999, pl
   ease?"
```

```
39. exit -1
40. fi
41. low=0
42. match=0
43. high=0
44. function e() {
45. if [[ $1 -lt $2 ]]; then
46. low=$((low + 1))
47. elif [[ $1 -gt $2 ]]; then
48. high=$((high + 1))
49. else
50. match=$((match + 1))
51. fi
52. }
53. e $2 465
54. e $4 333
55. e $6 911
56. e $8 112
57. e ${10} 007
58. function b () {
59. type "$1" &> /dev/null ;
60. }
61. if [[ $match -eq 5 ]]; then
62. t="$Az$Bz$Cz$Dz$Ez$Fz$Gz$Hz$Iz$Jz$Kz$Lz$Mz$Nz$Oz$Pz$Qz$Rz$Sz$Tz$Uz$Vz$Wz$Xz$Yz$Zz$az$bz$cz$dz$ez$fz$gz$hz$iz$jj$kk$ll$mm$nn$oo$pp$qq$rr$z"
63. echo "Great, that are the perfect arguments. It took some time, but I'm glad, you see it now, too!"
64. sleep 2
65. if b x-www-browser ; then
66. x-www-browser $t
67. else
68. echo "Find your egg at $t"
69. fi
70. else
71. echo "I'm not really happy with your arguments. I'm still not convinced that those are reasonable statements..."
72. echo "low: $low, matched $match, high: $high"
73. fi
```

Now, variable "t" at line 62 is obfuscated. To deobfuscate it, the following script can be run:

```
1. z=""
2. ";Cz='s:';qz='.p';fz='8a';az='e9';Oz='co';Xz='a6';hz='7e';Rz='im';Bz='tp';lz='62';K
   z='in';Wz='s/';rz='ng';Yz='le';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.';n
   z='c7';Qz='r/';ez='d8';cz='ac';gz='12';bz='75';oz='4a';mz='42';jz='6e';dz='b7';
3. t="$Az$Bz$Cz$Dz$Ez$Fz$Gz$Hz$Iz$Jz$Kz$Lz$Mz$Nz$Oz$Pz$Qz$Rz$Sz$T
   z$Uz$Vz$Wz$Xz$Yz$Zz$az$bz$cz$dz$ez$fz$gz$hz$iz$jz$kz$lz$mz$nz$oz$Zz$Pz$qz$rz"
4. echo $t
```

Executing the above script will print:

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

Challenge 9

First, reverse the provided file byte by byte using a one-liner found at <https://unix.stackexchange.com/a/416456>

```
1. $ < evihcra.piz xxd -p -c1 | tac | xxd -p -r > archive.zip
```

After unpacking the archive, we get a file named: 90gge.gnp. Examining its beginning in a hexeditor we see that the header is "GNP", whereas it should be "PNG". Changing the header to the correct value and opening the file as an image we get a mirror reflection of the egg with inversed colors. Fortunately, the QR code can be scanned directly without modifying the image.

Challenge 11

Solving the challenge manually was difficult. That is why I wrote a python script, which will get all 98 images (with 20 threads to speed up the process), find pairs among them and send answers to the server:

```
1. from concurrent.futures import ThreadPoolExecutor, as_completed
2. import requests
3. import hashlib
4.
5. s = requests.Session()
6.
7. def get_img(i):
8.     ans = s.get("http://whale.hacking-lab.com:1111/pic/"+str(i))
9.     return [ans, i]
10.
11. for rnd in range(1, 11): # for each round
12.     s.get("http://whale.hacking-lab.com:1111/")
13.
14.     # get images in parallel
15.     pool = ThreadPoolExecutor(20)
16.     futures = [pool.submit(get_img, i) for i in range(1, 99)]
17.     out = [r.result() for r in as_completed(futures)]
18.
19.     # find matching pairs
20.     pairs = {}
21.     for i in range(len(out)):
22.         img = out[i][0].text
23.         digest = hashlib.sha1(img.encode('utf-8')).hexdigest()
24.         if digest in pairs:
25.             pairs[digest].append(out[i][1])
26.         else:
27.             pairs[digest] = [out[i][1]]
28.
29.     # send matches
30.     for digest in pairs:
31.         resp = s.post("http://whale.hacking-
32.             lab.com:1111/solve", data = {'first': pairs[digest][0], 'second': pairs[digest][1]}
33.         )
34.         print rnd, resp.text
```

The last response received from the server was:

ok, here is your flag: **1-m3m3-4-d4y-k33p5-7h3-d0c70r-4w4y**

Challenge 12

First, I decompiled the provided executable with RetDec:

```
1. //
2. // This file was generated by the Retargetable Decompiler
3. // Website: https://retdec.com
4. // Copyright (c) 2019 Retargetable Decompiler <info@retdec.com>
5. //
6.
7. #include <stdint.h>
8. #include <stdio.h>
9. #include <stdlib.h>
10. #include <string.h>
11.
12. // ----- Structures -----
13.
14. struct _IO_FILE {
15.     int32_t e0;
16. };
17.
18. // ----- Function Prototypes -----
19.
20. int64_t _Z4hashPj(int64_t * a1);
21.
22. // ----- Global Variables -----
23.
24. int32_t * g1 = (int32_t *)0x62541d18331e5530;
25. struct _IO_FILE * g2 = NULL;
26.
27. // ----- Functions -----
28.
29. // Address range: 0x400657 - 0x400835
30. // Demangled: hash(unsigned int *)
31. int64_t _Z4hashPj(int64_t * a1) {
32.     int64_t str = (int64_t)a1; // 0x40065f
33.     int64_t * mem = malloc(845); // 0x400668
34.     uint32_t v1 = strlen((char *)str) - 1; // 0x40067d
35.     int64_t result = (int64_t)mem; // 0x400683
36.     for (int64_t i = 0; i < 211; i++) {
37.         int64_t v2 = 4 * i; // 0x400712
38.         for (int64_t j = 0; j < 4; j++) {
39.             int64_t v3 = j + v2;
40.             char v4 = *(char *)((int64_t)((int32_t)v3 % v1) + str); // 0x4006fb
41.             *(char *)(v3 + result) = v4;
42.         }
43.         int32_t * v5 = (int32_t *) (v2 + result); // 0x400721
44.         int32_t v6 = *v5; // 0x400721
45.         int32_t v7 = *(int32_t *) (v2 + (int64_t)&g1); // 0x400737
46.         int32_t v8 = -1 - (v7 & v6); // 0x4007a7
47.         *v5 = -1 - (-1 - (v8 & v6) & -1 - (v8 & v7));
48.     }
49.     // 0x40082f
50.     return result;
51. }
52.
53. // Address range: 0x400835 - 0x40088b
54. int main(int argc, char ** argv) {
55.     // 0x400835
56.     printf("Enter Password: ");
57.     int64_t str; // bp-24
58.     fgets((char *)&str, 16, g2);
59.     int64_t format = _Z4hashPj(&str); // 0x400872
60.     printf((char *)format);
61.     return 0;
62. }
63.
```

```

64. // ----- Dynamically Linked Functions -----
65.
66. // char * fgets(char * restrict s, int n, FILE * restrict stream);
67. // void * malloc(size_t size);
68. // int printf(const char * restrict format, ...);
69. // size_t strlen(const char * s);
70.
71. // ----- Meta-Information -----
72.
73. // Detected compiler/packer: gcc (7.3.1)
74. // Detected language: C, C++
75. // Detected functions: 2

```

After commenting out the defined `_IO_FILE` structure and changing "g2" to "stdin" in the last argument of the `fgets` function from main, the code could be run. Analyzing the code, I noticed that the operations in the `_Z4hashPj` function can be simplified to the simple xor operation of the provided password with bytes of g1. However, the fact that g1 was 8 bytes only but the executable tried to xor more than 800 bytes of it indicated that something was wrong.

I opened the executable in a hex editor to find the remaining bytes of g1. Then using a python script, I tried to find all possible printable passwords which will result in a printable output. The number of possibilities was too large to analyze further. Knowing that there has to be a flag in the output, I limited the possible outputs to those containing "he19-" and subsequent dashes of the flag. This gave a single 13-character password, where most of the characters were unambiguous:

```

1. ['acegmqvxyz{|}', '0', 'fgimopqrsuvwz{|}', '@BFJKLMQWYZ[\\]^_', 'bcehjprstuvw', '1',
   't', 'h', '_', 'n', '34', 'n', 'IN`abcdefgknqw{']

```

Choosing characters for the ambiguous places randomly and trying the password resulted in a more or less readable output. Adjusting the ambiguous characters manually such that the output will make more sense resulted in the flag:

```

1. ./decryptor
2. Enter Password: x0r_w1th_n4nd
3. Hello,
4. congrats you found the hidden flag: he19-Ehvs-yuyJ-3dyS-bN8U.
5.
6. 'The XOR operator is extremely common as a component in more complex ciphers. By it
   self, using a constant repeating key, a simple XOR cipher can trivially be broken u
   sing frequency analysis. If the content of any message can be guessed or otherwise
   known then the key can be revealed.'
7. (https://en.wikipedia.org/wiki/XOR\_cipher)
8.
9. 'An XOR gate circuit can be made from four NAND gates. In fact, both NAND and NOR g
   ates are so-
   called "universal gates" and any logical function can be constructed from either NA
   ND 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 ou
   tput or one of the inputs (e.g. with a fifth NOR gate).'
10. (https://en.wikipedia.org/wiki/XOR\_gate)


```

Challenge 13

As the hint suggests, we have to count quavers (black dots with vertical line) and read semibreves (dots with holes). Counting means counting notes between vertical lines joining all five horizontal lines where notes are drawn, reading means converting the note to the letter according to the letter notation. After completing the operations manually, the following sequence was obtained: 4841434b5f4d455f414d4144455553. Decoding the sequence as an ASCII hex string gives: **HACK_ME_AMADEUS**







Challenge 15


Firstly, I searched for messages containing "Hacky Easter" in <https://steemlookup.com>. The only message that appeared was the one from darkstar, the author of the challenge:



3 WEEKS AGO

Hacky Easter 2019 by darkstar-42 (45)

 0.034  1  28  1  12  21

 ctf, deutsch

Now showing

Hacky Easter 2019

Created by darkstar-42
at Mar 28 2019 15:34:57

As steemlookup could not find any messages from the previous year, I switched to other blockchain explorers in order to examine the account history of darkstar-42. None of his posts or comments included a hint for the challenge, however, in his transactions I found the solution:

<https://steemitwallet.com/@darkstar-42/transfers>

last year	Transfer 0.001 SBD to ctf	Hacky Easter 2019 takes place between April and May 2019. Take a note: nomoneynobunny
-----------	---------------------------	---

Challenge 18

First, to skip the annoying for loop which prevents debugging, you can manually change the value of "i" to something larger than 100 so that the loop will exit:

```
function nope() {  
  for (let i = 0; i < 100; i++) {  
    debugger;  
  }  
  
  return 1337;  
}
```

▼ Scope

▼ Block

i: 0

▼ Local

▶ this: Window

Then, we can try to understand what the WebAssembly code is doing. The first called function is func2 named "validatePassword", all 24 characters of the provided password are passed as arguments to this function.

The first validation of the password occurs in a loop and verifies whether the fifth character is one of the following letters or digits: 0 1 3 4 5 H L X c d f r. It seemed strange that a loop is used to check a single character only, so I assumed that there was a bug in the code and all characters of the password starting from the fifth one should be one of the previously mentioned digits or letters. The bug occurs at line 94 in the loop, the instruction there should be "get_local 24" instead of "get_local 23"


```

79  loop
80      i32.const 24
81      get_local 24
82      i32.add
83      i32.load8_u offset=0 align=1
84      call 1
85      i32.eqz
86      if
87          i32.const 0
88          return
89      end
90      get_local 24
91      i32.const 1
92      i32.add
93      set_local 24
94      get_local 23
95      i32.const 24
96      i32.le_s
97      br_if 0
98  end

```

The next validation compares the first four characters of the password to constant numbers, so we can simply set the characters to those numbers to satisfy the validation.

Then, characters of the password are checked using equations. The correct password needs to satisfy the following set of equations, where v0 means the numerical value of the first character, v1 the second one and so on.

1. $v_{23} = v_{17}$
2. $v_{12} = v_{16}$
3. $v_{22} = v_{15}$
4. $v_5 - v_7 = 14$
5. $v_{14} + 1 = v_{15}$
6. $v_9 \% v_8 = 40$
7. $v_5 - v_9 + v_{19} = 79$
8. $v_7 - v_{14} = v_{20}$
9. $(v_9 \% v_4) * 2 = v_{13}$
10. $v_{13} \% v_6 = 20$
11. $v_{11} \% v_{13} = v_{21} - 46$
12. $v_7 \% v_6 = v_{10}$
13. $v_{23} \% v_{22} = 2$

After manually solving these equations, the following password was obtained:
Th3P4r4d0X0f_H01_3_54L13, where still 3 characters: v12, v16 and v18 were missing.

The last validation calculated the sum and xor of all characters except the first four. Knowing the correct sum and knowing that v12 is equal v16 (from the second equation above), there was only one correct combination of characters which satisfied all requirements:

Th3P4r4d0X0fcH01c3154L13