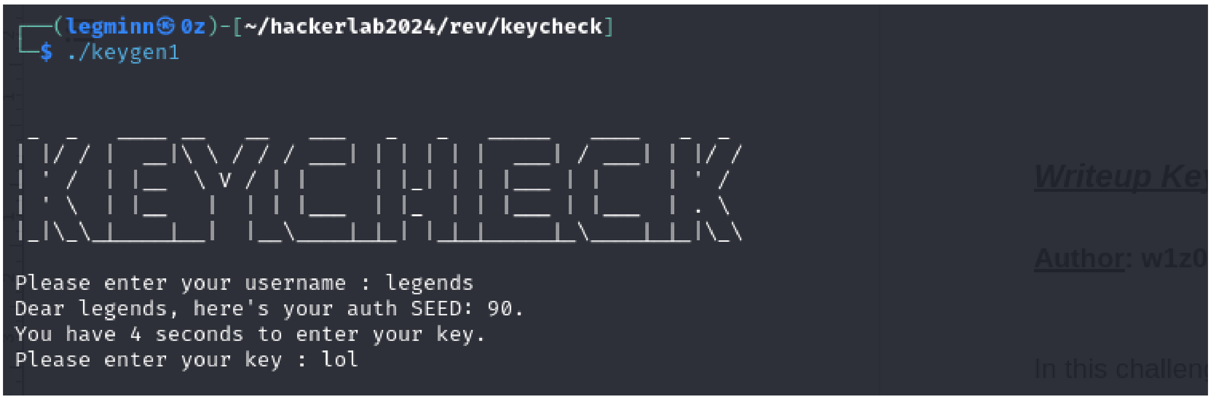
Writeup Keycheck

Author: w1z0z

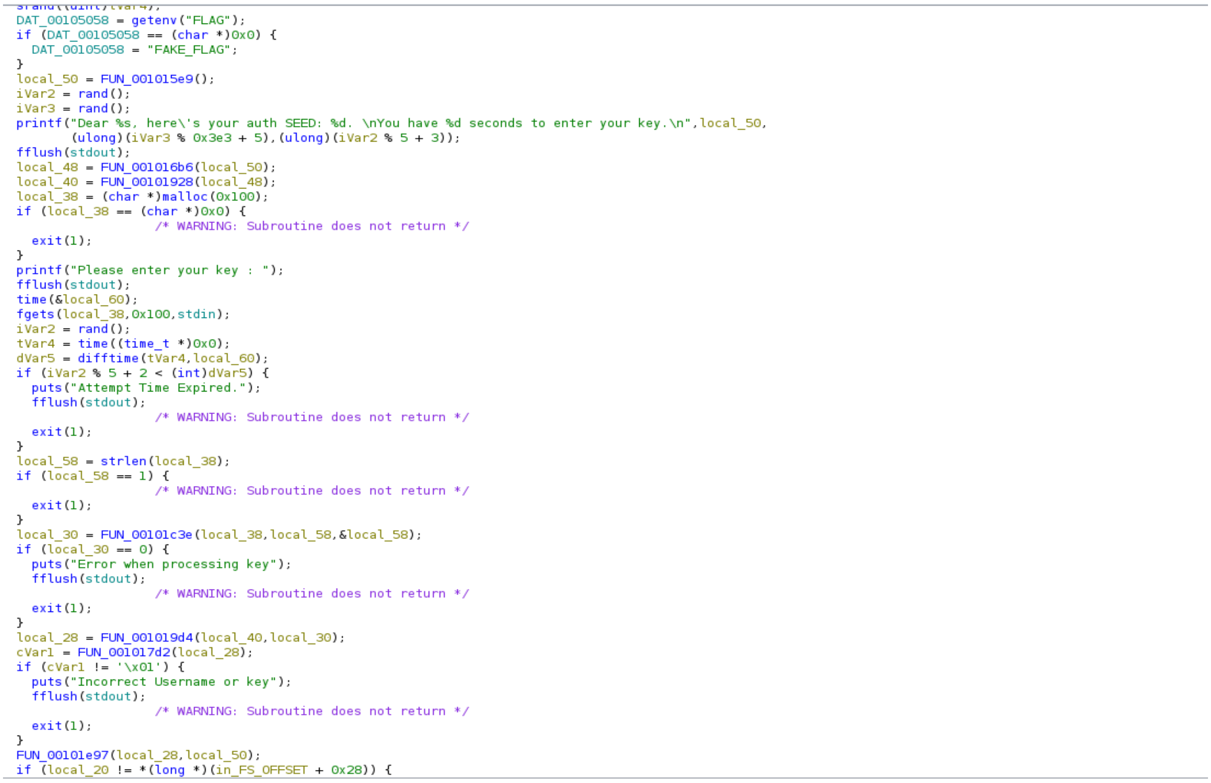
In this challenge we were provided a binary and a crypto library. We also have a server.

We add the current directory to the **LD_LIBRARY_PATH** and we are now able to launch the **keygen1** binary. Launching it we get a prompt asking us for a username and a key.



```
┌──(legminn㉿0z)-[~/hackerlab2024/rev/keycheck]
└─$ ./keygen1


 _  _________   __ ____ _   _ _____ ____ _  __
| |/ / ____\ \ / // ___| | | | ____/ ___| |/ /
| ' /|  _|  \ V /| |   | |_| |  _|| |   | ' /
| . \| |___  | | | |___|  _  | |__| |___| . \
|_|\_\_____| |_|  \____|_| |_|_____\____|_|\_\

Please enter your username : legends
Dear legends, here's your auth SEED: 90.
You have 4 seconds to enter your key.
Please enter your key : lol
```

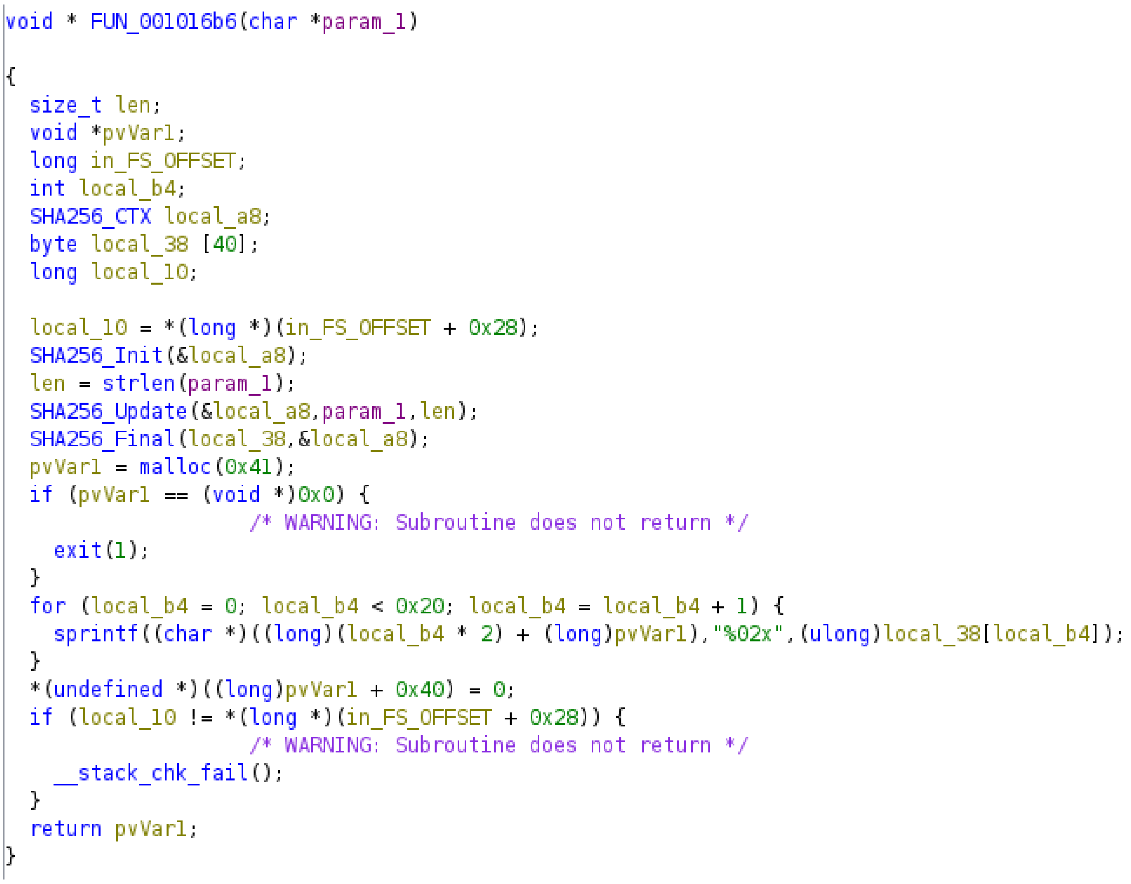
Hum. Nothing is coming back. Time to decompile it. First the main function.



```
srand((uint)tVar4);
DAT_00105058 = getenv("FLAG");
if (DAT_00105058 == (char *)0x0) {
  DAT_00105058 = "FAKE_FLAG";
}
local_50 = FUN_001015e9();
iVar2 = rand();
iVar3 = rand();
printf("Dear %s, here\'s your auth SEED: %d. \nYou have %d seconds to enter your key.\n",local_50,
       (ulong)(iVar3 % 0x3e3 + 5),(ulong)(iVar2 % 5 + 3));
fflush(stdout);
local_48 = FUN_001016b6(local_50);
local_40 = FUN_00101928(local_48);
local_38 = (char *)malloc(0x100);
if (local_38 == (char *)0x0) {
                  /* WARNING: Subroutine does not return */
  exit(1);
}
printf("Please enter your key : ");
fflush(stdout);
time(&local_60);
fgets(local_38,0x100,stdin);
iVar2 = rand();
tVar4 = time((time_t *)0x0);
dVar5 = difftime(tVar4,local_60);
if (iVar2 % 5 + 2 < (int)dVar5) {
  puts("Attempt Time Expired.");
  fflush(stdout);
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_58 = strlen(local_38);
if (local_58 == 1) {
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_30 = FUN_00101c3e(local_38,local_58,&local_58);
if (local_30 == 0) {
  puts("Error when processing key");
  fflush(stdout);
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_28 = FUN_001019d4(local_40,local_30);
cVar1 = FUN_001017d2(local_28);
if (cVar1 != '\x01') {
  puts("Incorrect Username or key");
  fflush(stdout);
                  /* WARNING: Subroutine does not return */
  exit(1);
}
FUN_00101e97(local_28,local_50);
if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
```

Ok first it is getting the flag from the environment and stocking it. After that it is making some random values for the seed and the time we’ve got to input the key after entering the username. After printing them and getting the username and the key we assist a successive call of diverse functions.
We will skip the **FUN001015e9** as it is only getting the username.

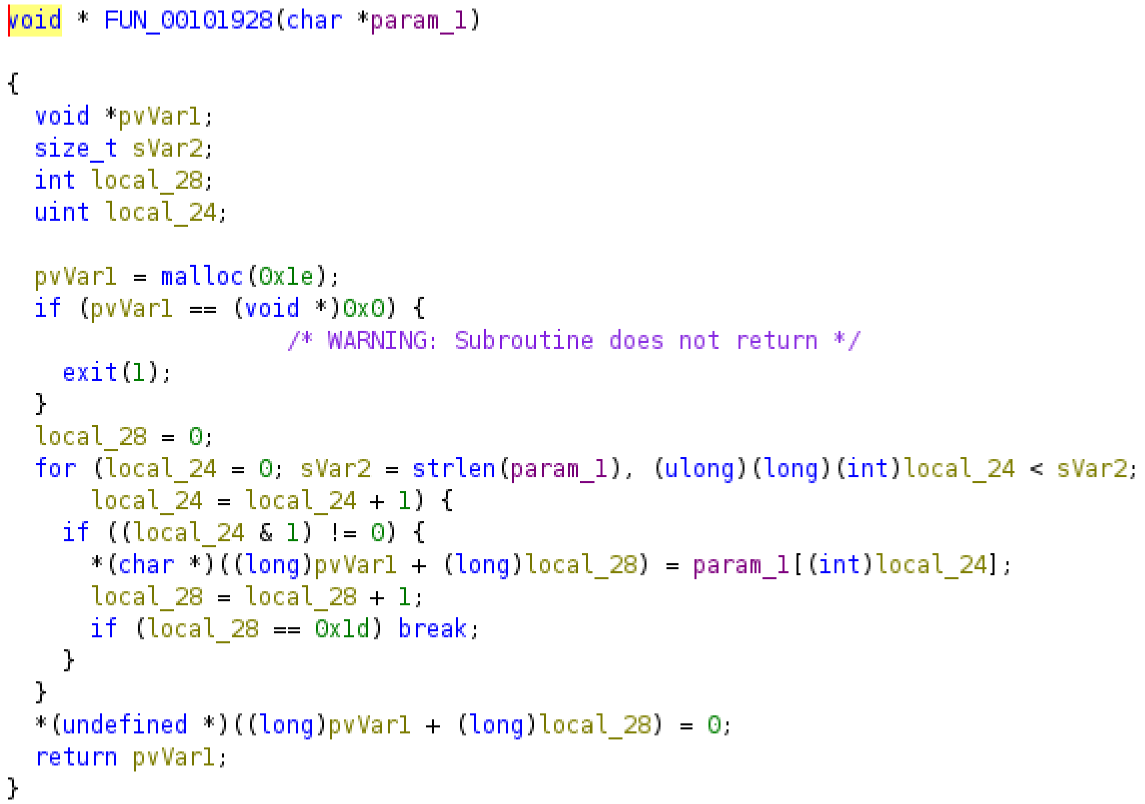
FUN_001016b6



```
void * FUN_001016b6(char *param_1)

{
  size_t len;
  void *pvVar1;
  long in_FS_OFFSET;
  int local_b4;
  SHA256_CTX local_a8;
  byte local_38 [40];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  SHA256_Init(&local_a8);
  len = strlen(param_1);
  SHA256_Update(&local_a8,param_1,len);
  SHA256_Final(local_38,&local_a8);
  pvVar1 = malloc(0x41);
  if (pvVar1 == (void *)0x0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  for (local_b4 = 0; local_b4 < 0x20; local_b4 = local_b4 + 1) {
    sprintf((char *)((long)(local_b4 * 2) + (long)pvVar1),"%02x",(ulong)local_38[local_b4]);
  }
  *(undefined *)((long)pvVar1 + 0x40) = 0;
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return pvVar1;
}
```

This function takes in parameters the username. It encodes it in SHA256 and returns the result as an hex encoded string.

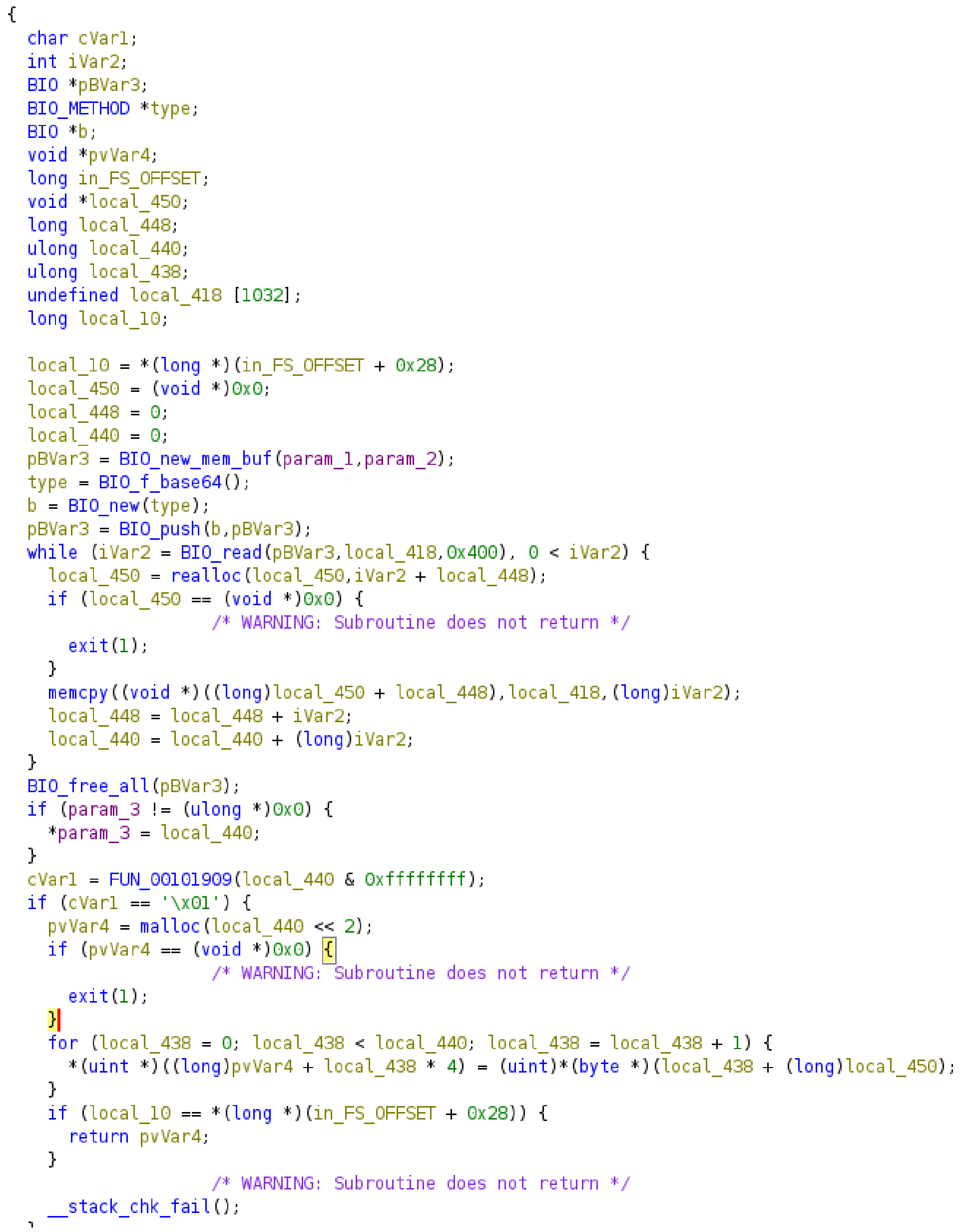
FUN_00101928



```
void * FUN_00101928(char *param_1)

{
  void *pvVar1;
  size_t sVar2;
  int local_28;
  uint local_24;

  pvVar1 = malloc(0x1e);
  if (pvVar1 == (void *)0x0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  local_28 = 0;
  for (local_24 = 0; sVar2 = strlen(param_1), (ulong)(long)(int)local_24 < sVar2;
      local_24 = local_24 + 1) {
    if ((local_24 & 1) != 0) {
      *(char *)((long)pvVar1 + (long)local_28) = param_1[(int)local_24];
      local_28 = local_28 + 1;
      if (local_28 == 0x1d) break;
    }
  }
  *(undefined *)((long)pvVar1 + (long)local_28) = 0;
  return pvVar1;
}
```

This function takes into parameter the result of the previous function. It goes through the string and keeps only values with even indexes to finally return a string of 30 chars length.

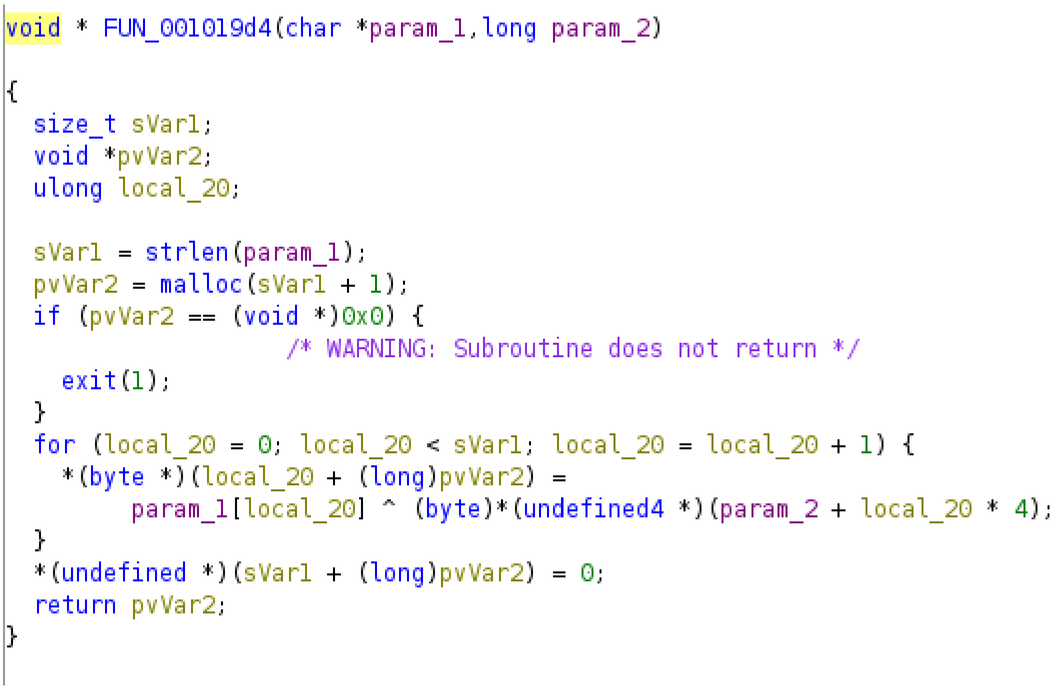
FUN_00101c3e



```
{
  char cVar1;
  int iVar2;
  BIO *pBVar3;
  BIO_METHOD *type;
  BIO *b;
  void *pvVar4;
  long in_FS_OFFSET;
  void *local_450;
  long local_448;
  ulong local_440;
  ulong local_438;
  undefined local_418 [1032];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_450 = (void *)0x0;
  local_448 = 0;
  local_440 = 0;
  pBVar3 = BIO_new_mem_buf(param_1,param_2);
  type = BIO_f_base64();
  b = BIO_new(type);
  pBVar3 = BIO_push(b,pBVar3);
  while (iVar2 = BIO_read(pBVar3,local_418,0x400), 0 < iVar2) {
    local_450 = realloc(local_450,iVar2 + local_448);
    if (local_450 == (void *)0x0) {
                    /* WARNING: Subroutine does not return */
      exit(1);
    }
    memcpy((void *)((long)local_450 + local_448),local_418,(long)iVar2);
    local_448 = local_448 + iVar2;
    local_440 = local_440 + (long)iVar2;
  }
  BIO_free_all(pBVar3);
  if (param_3 != (ulong *)0x0) {
    *param_3 = local_440;
  }
  cVar1 = FUN_00101909(local_440 & 0xffffffff);
  if (cVar1 == '\x01') {
    pvVar4 = malloc(local_440 << 2);
    if (pvVar4 == (void *)0x0) {
                    /* WARNING: Subroutine does not return */
      exit(1);
    }
    for (local_438 = 0; local_438 < local_440; local_438 = local_438 + 1) {
      *(uint *)((long)pvVar4 + local_438 * 4) = (uint)*(byte *)(local_438 + (long)local_450);
    }
    if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
      return pvVar4;
    }
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
```

This function takes into parameter the key, the length of the key and the address of the variable holding the length of the key. It decodes the key from base 64 and checks that the length of the decoded data is equal to 29. It proceeds to convert the decoded string into an integer array and return this array.

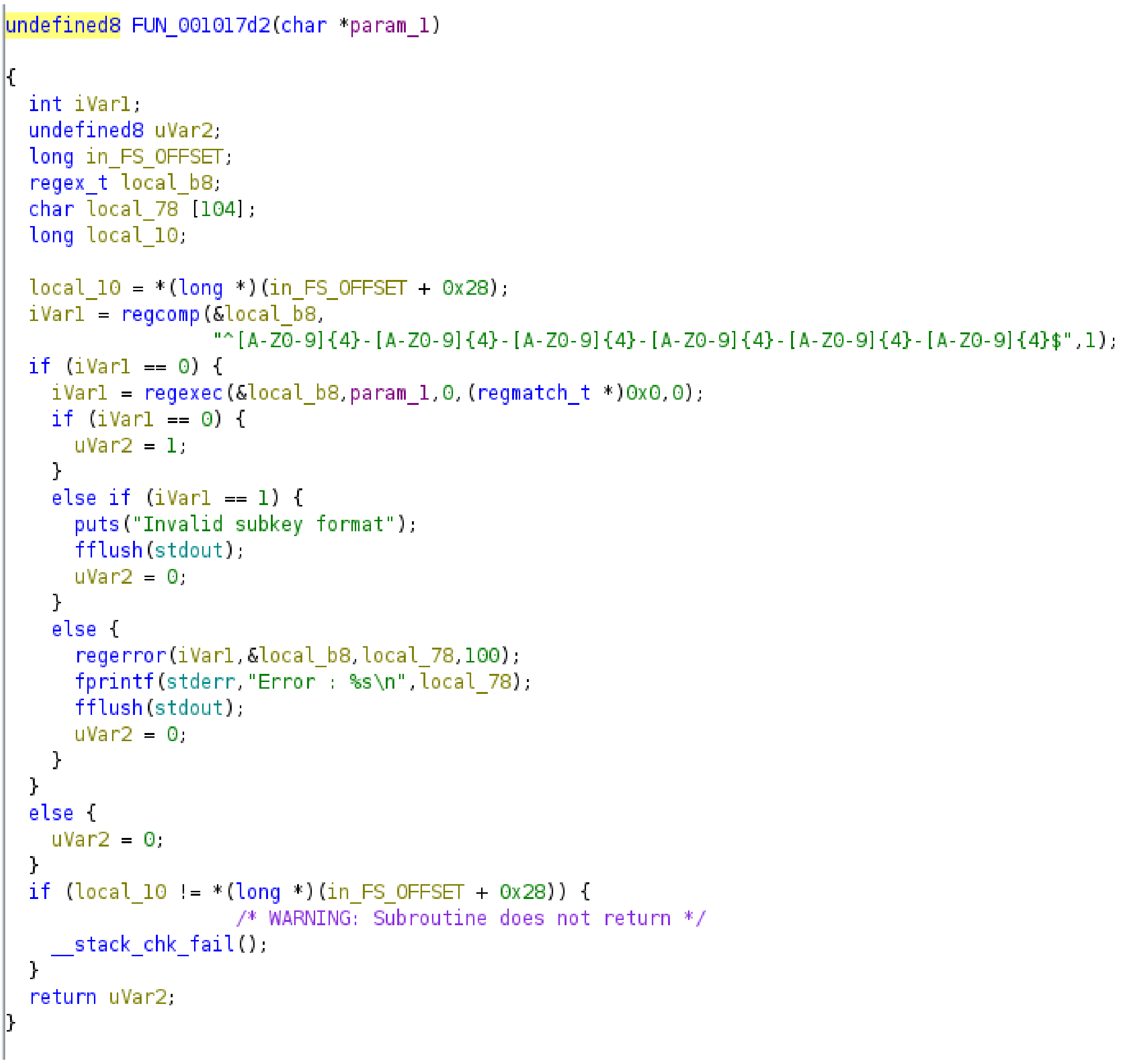
FUN_001019d4



```
void * FUN_001019d4(char *param_1,long param_2)

{
  size_t sVar1;
  void *pvVar2;
  ulong local_20;

  sVar1 = strlen(param_1);
  pvVar2 = malloc(sVar1 + 1);
  if (pvVar2 == (void *)0x0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  for (local_20 = 0; local_20 < sVar1; local_20 = local_20 + 1) {
    *(byte *)(local_20 + (long)pvVar2) =
         param_1[local_20] ^ (byte)*(undefined4 *)(param_2 + local_20 * 4);
  }
  *(undefined *)(sVar1 + (long)pvVar2) = 0;
  return pvVar2;
}
```

This function takes into parameter the remainder of the username (after sha256, hex, even indexed selection) and the int array from the previous function. It xor the two values and return the resulting string.

FUN_001017d2



```
undefined8 FUN_001017d2(char *param_1)

{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  regex_t local_b8;
  char local_78 [104];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = regcomp(&local_b8,
                  "^[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}$",1);
  if (iVar1 == 0) {
    iVar1 = regexec(&local_b8,param_1,0,(regmatch_t *)0x0,0);
    if (iVar1 == 0) {
      uVar2 = 1;
    }
    else if (iVar1 == 1) {
      puts("Invalid subkey format");
      fflush(stdout);
      uVar2 = 0;
    }
    else {
      regerror(iVar1,&local_b8,local_78,100);
      fprintf(stderr,"Error : %s\n",local_78);
      fflush(stdout);
      uVar2 = 0;
    }
  }
  else {
    uVar2 = 0;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar2;
}
```

This function takes the result of the previous xor and does a regex check on it. With that we know that at this step the value we inputted are supposed to give after all the encoding, decoding and the xor something with the format **^[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}$**

After this check the key is now valid and the program will go on to the last function to check the validity of that key. So far i made a script that based on a username and a key following the format previously shown give us the key to send as input to the program

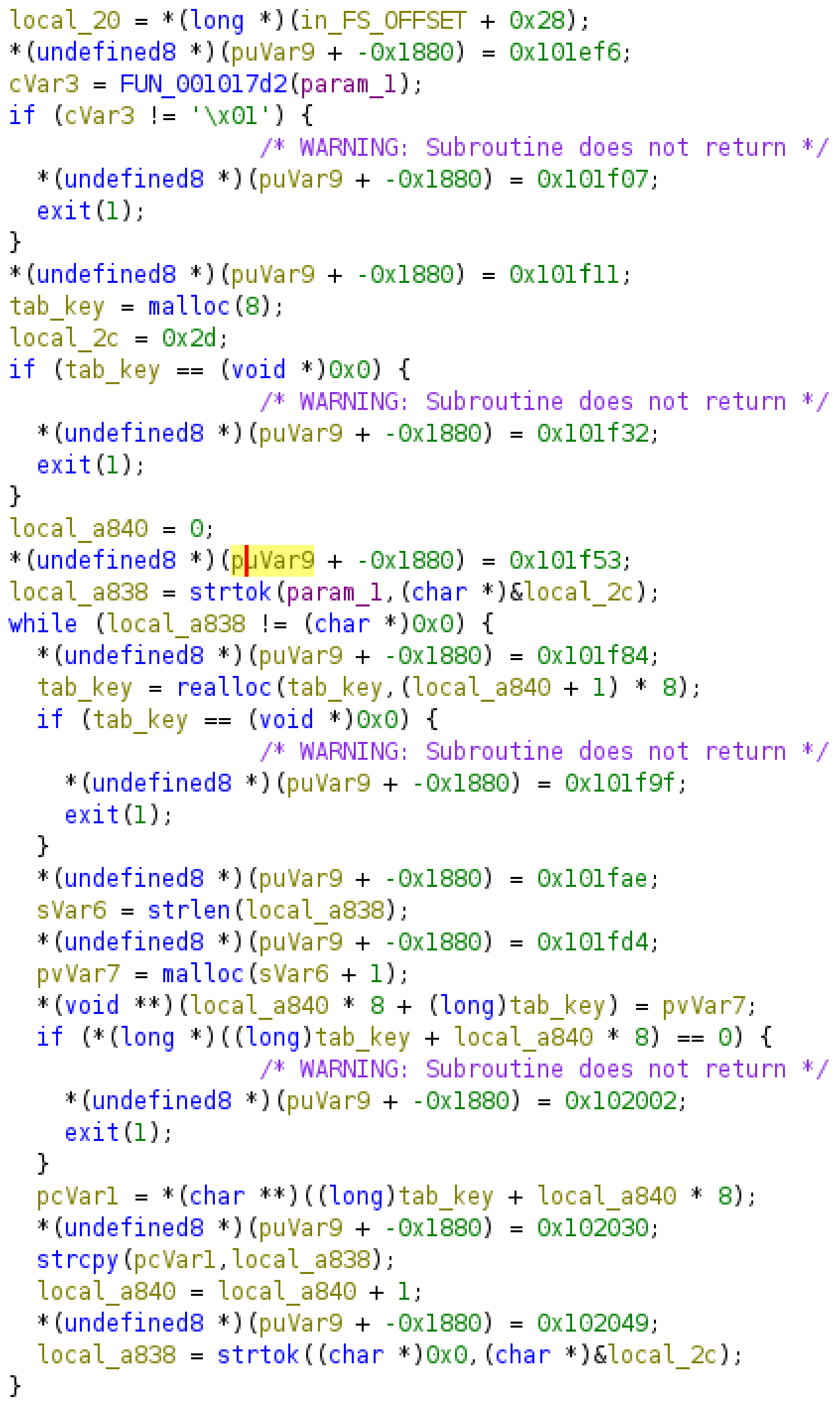
```
import hashlib
from pwn import xor
import base64 as b64

username = b"BJIZ-HACKERLAB"

def sha256hex(data):
     hasher = hashlib.sha256()
     hasher.update(data)
     return hasher.hexdigest()


step_0 = sha256hex(username)
step_2 = step_0[1::2][:29].encode('utf-8')

f_key = b"D18F-D18F-D18F-D18F-B6A5-D18F"

key = b64.b64encode(xor(step_2, f_key))

print(key)
```

Now we have a way to compute a key. Let’s move on to the validity check. The last function.

FUN_00101e97

The Function is splitted into 3 main parts

Part 1



```
local_20 = *(long *)(in_FS_OFFSET + 0x28);
*(undefined8 *)(puVar9 + -0x1880) = 0x101ef6;
cVar3 = FUN_001017d2(param_1);
if (cVar3 != '\x01') {
                  /* WARNING: Subroutine does not return */
  *(undefined8 *)(puVar9 + -0x1880) = 0x101f07;
  exit(1);
}
*(undefined8 *)(puVar9 + -0x1880) = 0x101f11;
tab_key = malloc(8);
local_2c = 0x2d;
if (tab_key == (void *)0x0) {
                  /* WARNING: Subroutine does not return */
  *(undefined8 *)(puVar9 + -0x1880) = 0x101f32;
  exit(1);
}
local_a840 = 0;
*(undefined8 *)(puVar9 + -0x1880) = 0x101f53;
local_a838 = strtok(param_1,(char *)&local_2c);
while (local_a838 != (char *)0x0) {
  *(undefined8 *)(puVar9 + -0x1880) = 0x101f84;
  tab_key = realloc(tab_key,(local_a840 + 1) * 8);
  if (tab_key == (void *)0x0) {
                  /* WARNING: Subroutine does not return */
    *(undefined8 *)(puVar9 + -0x1880) = 0x101f9f;
    exit(1);
  }
  *(undefined8 *)(puVar9 + -0x1880) = 0x101fae;
  sVar6 = strlen(local_a838);
  *(undefined8 *)(puVar9 + -0x1880) = 0x101fd4;
  pvVar7 = malloc(sVar6 + 1);
  *(void **)(local_a840 * 8 + (long)tab_key) = pvVar7;
  if (*(long *)((long)tab_key + local_a840 * 8) == 0) {
                  /* WARNING: Subroutine does not return */
    *(undefined8 *)(puVar9 + -0x1880) = 0x102002;
    exit(1);
  }
  pcVar1 = *(char **)((long)tab_key + local_a840 * 8);
  *(undefined8 *)(puVar9 + -0x1880) = 0x102030;
  strcpy(pcVar1,local_a838);
  local_a840 = local_a840 + 1;
  *(undefined8 *)(puVar9 + -0x1880) = 0x102049;
  local_a838 = strtok((char *)0x0,(char *)&local_2c);
}
```

In the first part it splits the key into an array of strings. So what this means is that

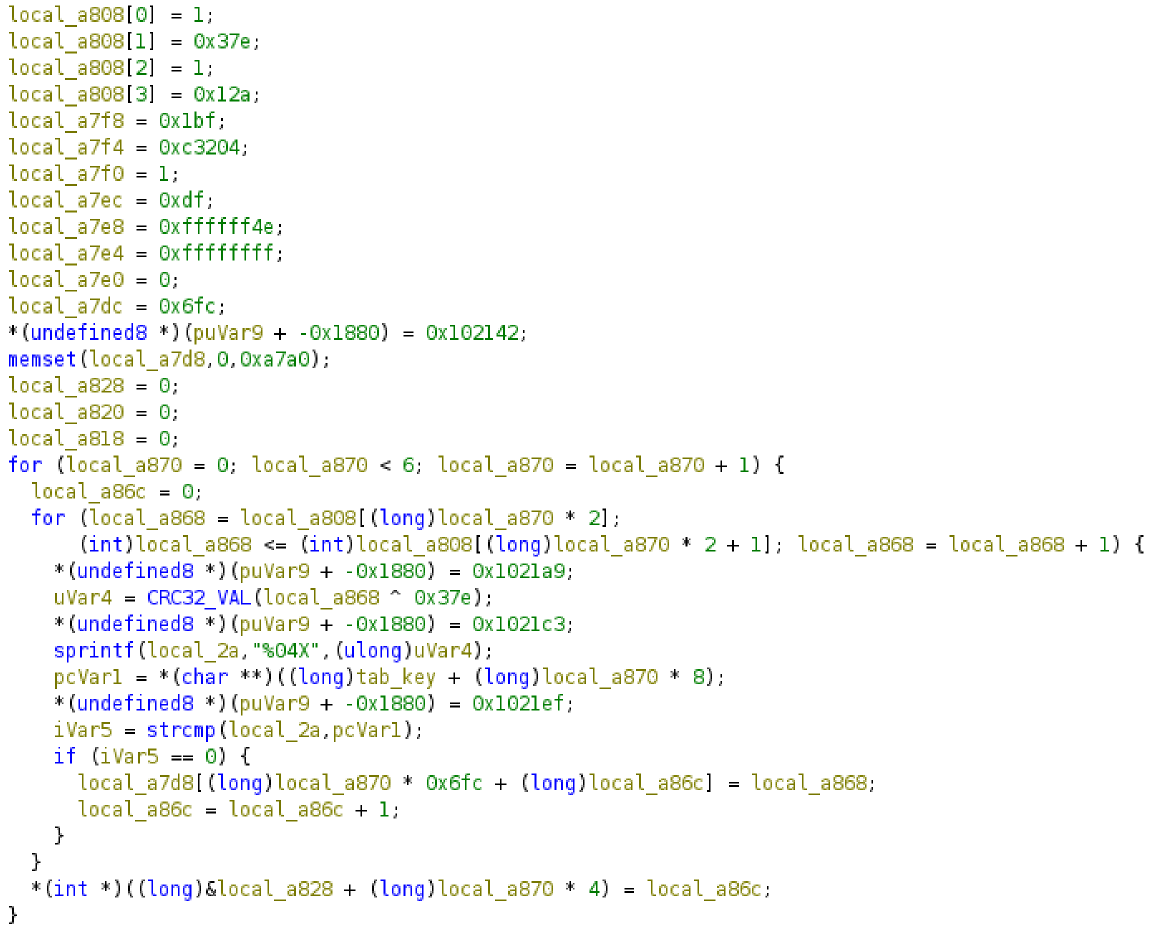
D18F-D18F-D18F-D18F-B6A5-D18F

will become

[0]D18F [1]D18F [2]D18F [3]D18F [4]B6A5 [5]D18F

It uses strtok for that.

Part 2



```
local_a808[0] = 1;
local_a808[1] = 0x37e;
local_a808[2] = 1;
local_a808[3] = 0x12a;
local_a7f8 = 0x1bf;
local_a7f4 = 0xc3204;
local_a7f0 = 1;
local_a7ec = 0xdf;
local_a7e8 = 0xffffff4e;
local_a7e4 = 0xffffffff;
local_a7e0 = 0;
local_a7dc = 0x6fc;
*(undefined8 *)(puVar9 + -0x1880) = 0x102142;
memset(local_a7d8,0,0xa7a0);
local_a828 = 0;
local_a820 = 0;
local_a818 = 0;
for (local_a870 = 0; local_a870 < 6; local_a870 = local_a870 + 1) {
  local_a86c = 0;
  for (local_a868 = local_a808[(long)local_a870 * 2];
      (int)local_a868 <= (int)local_a808[(long)local_a870 * 2 + 1]; local_a868 = local_a868 + 1) {
    *(undefined8 *)(puVar9 + -0x1880) = 0x1021a9;
    uVar4 = CRC32_VAL(local_a868 ^ 0x37e);
    *(undefined8 *)(puVar9 + -0x1880) = 0x1021c3;
    sprintf(local_2a,"%04X",(ulong)uVar4);
    pcVar1 = *(char **)((long)tab_key + (long)local_a870 * 8);
    *(undefined8 *)(puVar9 + -0x1880) = 0x1021ef;
    iVar5 = strcmp(local_2a,pcVar1);
    if (iVar5 == 0) {
      local_a7d8[(long)local_a870 * 0x6fc + (long)local_a86c] = local_a868;
      local_a86c = local_a86c + 1;
    }
  }
  *(int *)((long)&local_a828 + (long)local_a870 * 4) = local_a86c;
}
```

This is the binary main point.
It is quite complicated to understand at first but using many tools to decompile it we finally reach a quite solid level of comprehension of what is going on.
First **local_a808** is an array of 12 long and not 4 as shown in the picture (**ref hexrays**).

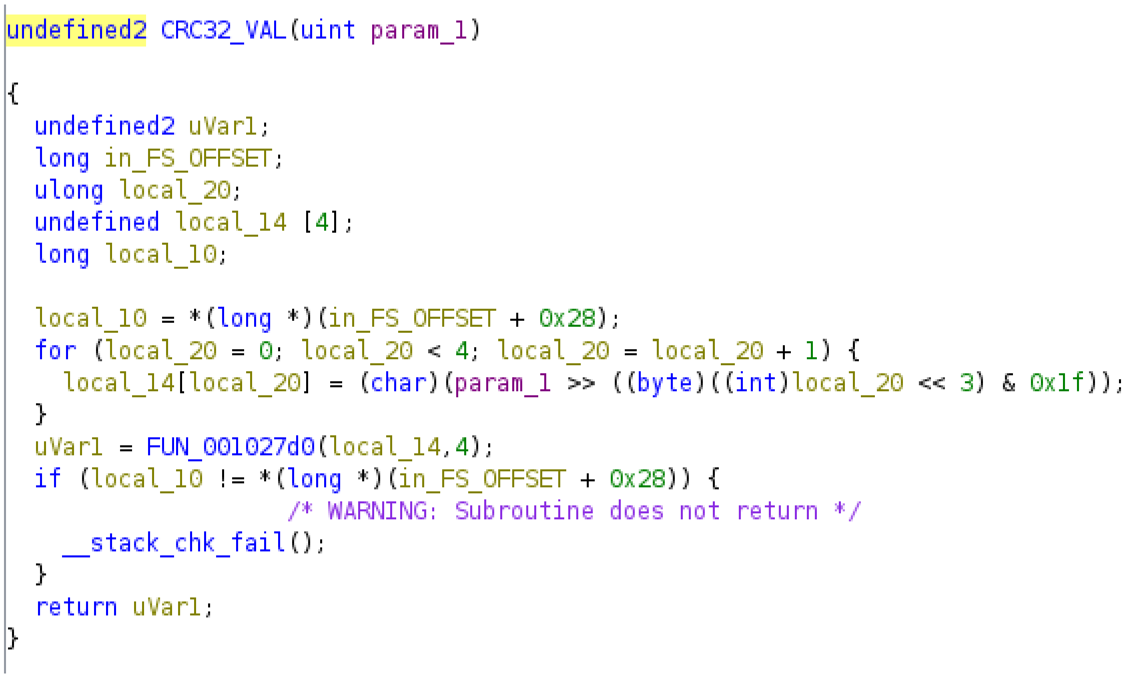
There are two loops. The first loop goes from **0 - 5.** The nested loop goes from the value located at the index **local_a870 * 2** to the value at **local_a870 * 2 + 1** in the **local_a808** tab. For readability purpose here is the **local_a808** table

[1,894,1,298,447,799236,1,223,-178,-1,0,178]

So the first iteration will go from 1 to 894, the second from 1 to 298….

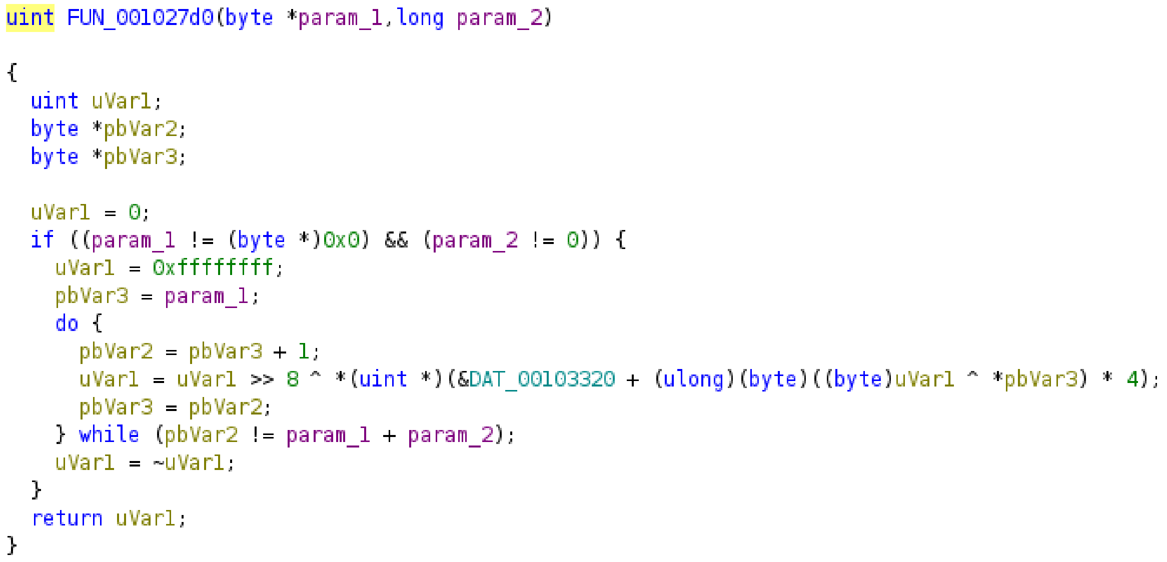
In the second loop at each iteration it takes the value and computes what I found out to the the **CRC32 checksum** of the value xor 0x37e.

CRC32_VAL



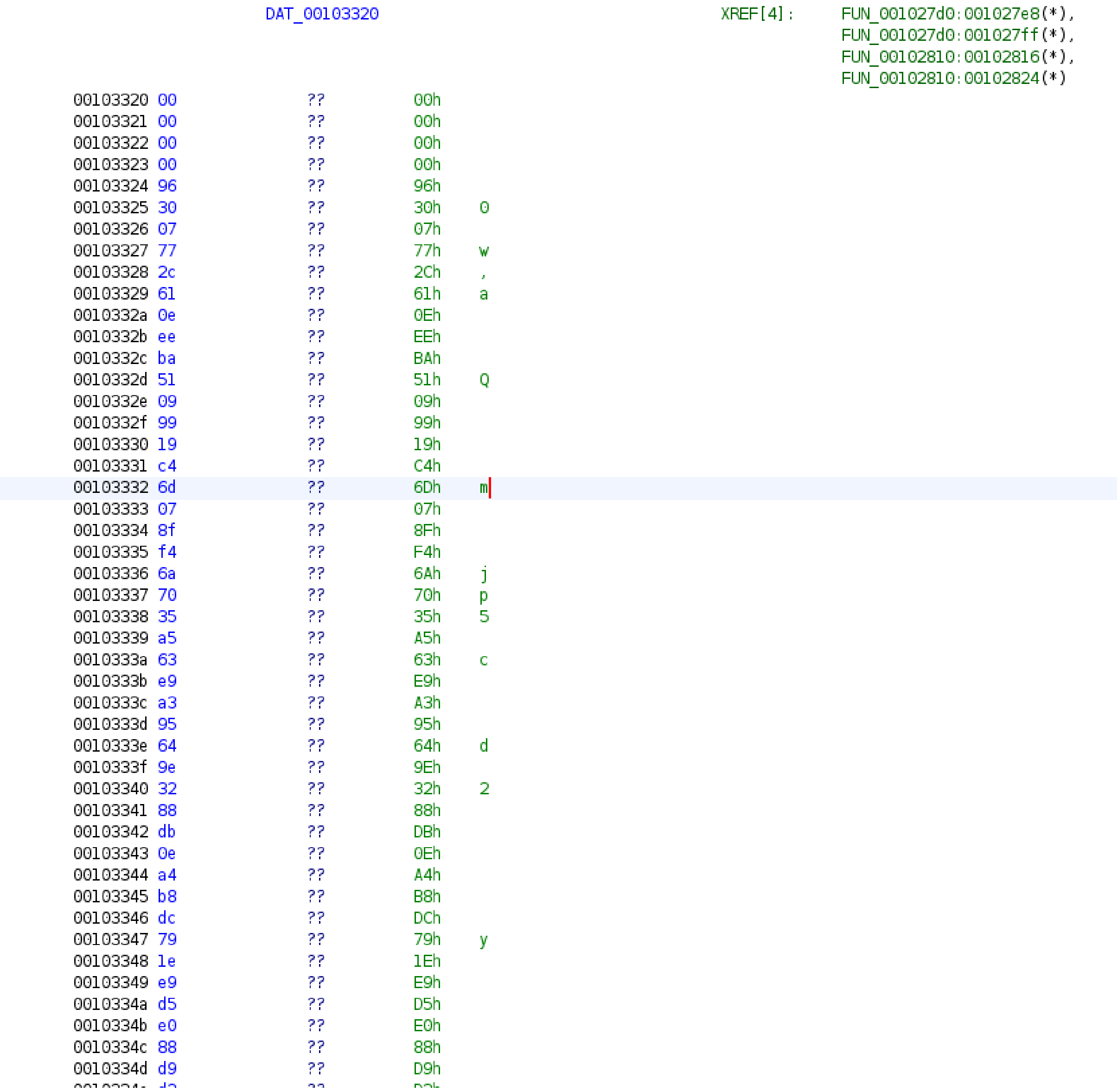
```
undefined2 CRC32_VAL(uint param_1)

{
  undefined2 uVar1;
  long in_FS_OFFSET;
  ulong local_20;
  undefined local_14 [4];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  for (local_20 = 0; local_20 < 4; local_20 = local_20 + 1) {
    local_14[local_20] = (char)(param_1 >> ((byte)((int)local_20 << 3) & 0x1f));
  }
  uVar1 = FUN_001027d0(local_14,4);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar1;
}
```

In this function it first stores each byte of the integer passed in parameter into an array of bytes of size 4 (integer size=4 bytes).
It then calls the function **FUN_001027d0.**



```
uint FUN_001027d0(byte *param_1,long param_2)

{
  uint uVar1;
  byte *pbVar2;
  byte *pbVar3;

  uVar1 = 0;
  if ((param_1 != (byte *)0x0) && (param_2 != 0)) {
    uVar1 = 0xffffffff;
    pbVar3 = param_1;
    do {
      pbVar2 = pbVar3 + 1;
      uVar1 = uVar1 >> 8 ^ *(uint *)(&DAT_00103320 + (ulong)(byte)((byte)uVar1 ^ *pbVar3) * 4);
      pbVar3 = pbVar2;
    } while (pbVar2 != param_1 + param_2);
    uVar1 = ~uVar1;
  }
  return uVar1;
}
```

This is where the real CRC32 computation takes place. It compute this using an CRC32 table which we can retrieve reading the memory



```
                    DAT_00103320                               XREF[4]:     FUN_001027d0:001027e8(*),
                                                                            FUN_001027d0:001027ff(*),
                                                                            FUN_00102810:00102816(*),
                                                                            FUN_00102810:00102824(*)
        00103320 00              ??         00h
        00103321 00              ??         00h
        00103322 00              ??         00h
        00103323 00              ??         00h
        00103324 96              ??         96h
        00103325 30              ??         30h    0
        00103326 07              ??         07h
        00103327 77              ??         77h    w
        00103328 2c              ??         2Ch    ,
        00103329 61              ??         61h    a
        0010332a 0e              ??         0Eh
        0010332b ee              ??         EEh
        0010332c ba              ??         BAh
        0010332d 51              ??         51h    Q
        0010332e 09              ??         09h
        0010332f 99              ??         99h
        00103330 19              ??         19h
        00103331 c4              ??         C4h
        00103332 6d              ??         6Dh    m
        00103333 07              ??         07h
        00103334 8f              ??         8Fh
        00103335 f4              ??         F4h
        00103336 6a              ??         6Ah    j
        00103337 70              ??         70h    p
        00103338 35              ??         35h    5
        00103339 a5              ??         A5h
        0010333a 63              ??         63h    c
        0010333b e9              ??         E9h
        0010333c a3              ??         A3h
        0010333d 95              ??         95h
        0010333e 64              ??         64h    d
        0010333f 9e              ??         9Eh
        00103340 32              ??         32h    2
        00103341 88              ??         88h
        00103342 db              ??         DBh
        00103343 0e              ??         0Eh
        00103344 a4              ??         A4h
        00103345 b8              ??         B8h
        00103346 dc              ??         DCh
        00103347 79              ??         79h    y
        00103348 1e              ??         1Eh
        00103349 e9              ??         E9h
        0010334a d5              ??         D5h
        0010334b e0              ??         E0h
        0010334c 88              ??         88h
        0010334d d9              ??         D9h
```

We’ll stock this for later.

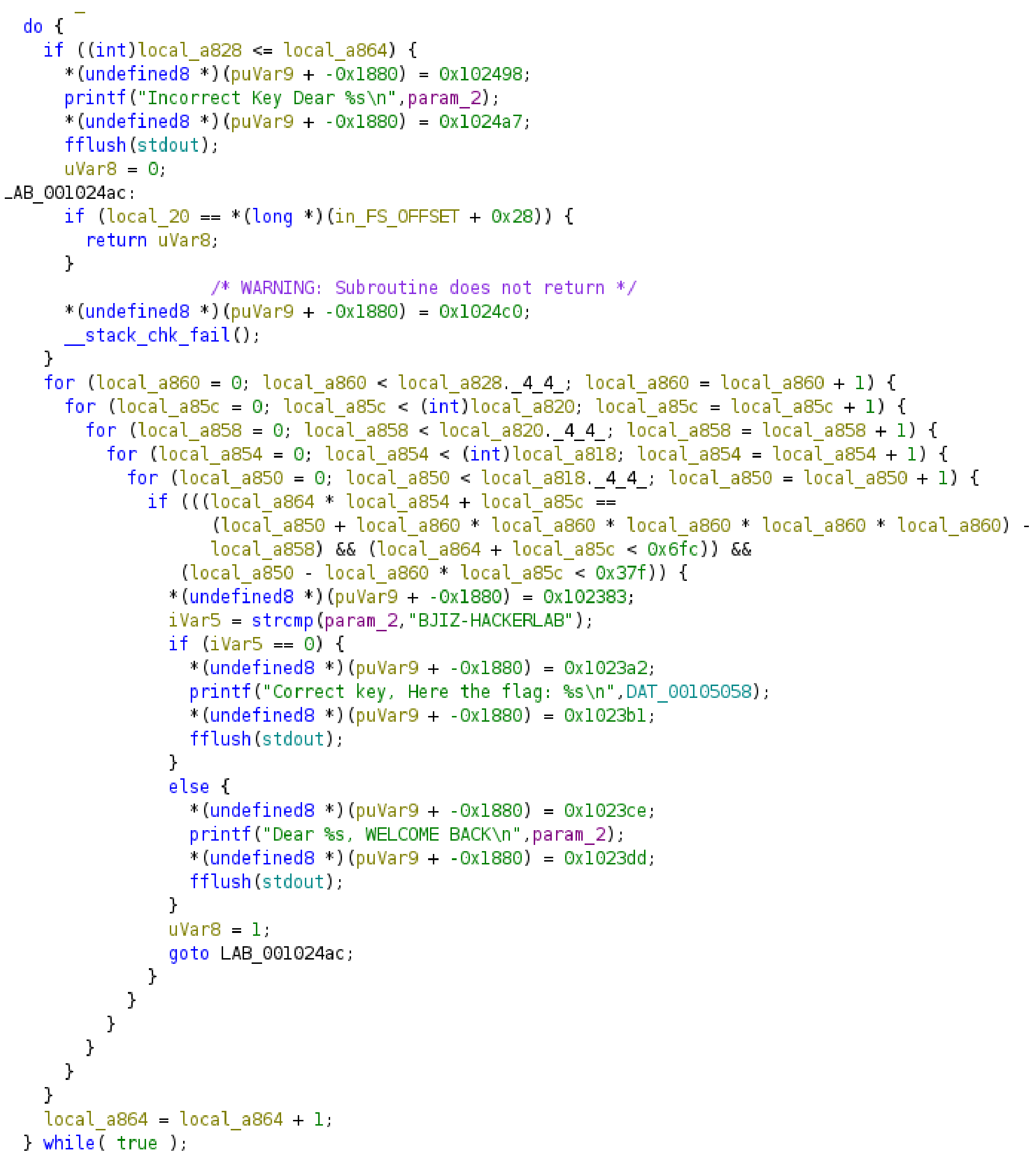
After computing the CRC32 of a value it converts it to its hexadecimal form and stores it into a string.

sprintf(local_2a,"%04X",(ulong)uVar4);

It then compares it with the actual value in our key tab (**tab_key[local_a870]**) and if it corresponds it increments a value which will be at the end of the second loop stored in an long array.

So in this second part the result will be an array of long containing values representing the number of correspondence of CRC32 value computed at each interval with each of the 6 parts of our key.

Part 3 (What was the point ???)



```
       _
  do {
    if ((int)local_a828 <= local_a864) {
      *(undefined8 *)(puVar9 + -0x1880) = 0x102498;
      printf("Incorrect Key Dear %s\n",param_2);
      *(undefined8 *)(puVar9 + -0x1880) = 0x1024a7;
      fflush(stdout);
      uVar8 = 0;
_AB_001024ac:
      if (local_20 == *(long *)(in_FS_OFFSET + 0x28)) {
        return uVar8;
      }
                    /* WARNING: Subroutine does not return */
      *(undefined8 *)(puVar9 + -0x1880) = 0x1024c0;
      __stack_chk_fail();
    }
    for (local_a860 = 0; local_a860 < local_a828._4_4_; local_a860 = local_a860 + 1) {
      for (local_a85c = 0; local_a85c < (int)local_a820; local_a85c = local_a85c + 1) {
        for (local_a858 = 0; local_a858 < local_a820._4_4_; local_a858 = local_a858 + 1) {
          for (local_a854 = 0; local_a854 < (int)local_a818; local_a854 = local_a854 + 1) {
            for (local_a850 = 0; local_a850 < local_a818._4_4_; local_a850 = local_a850 + 1) {
              if ((((local_a864 * local_a854 + local_a85c ==
                    (local_a850 + local_a860 * local_a860 * local_a860 * local_a860 * local_a860) -
                    local_a858) && (local_a864 + local_a85c < 0x6fc)) &&
                 (local_a850 - local_a860 * local_a85c < 0x37f)) {
                *(undefined8 *)(puVar9 + -0x1880) = 0x102383;
                iVar5 = strcmp(param_2,"BJIZ-HACKERLAB");
                if (iVar5 == 0) {
                  *(undefined8 *)(puVar9 + -0x1880) = 0x1023a2;
                  printf("Correct key, Here the flag: %s\n",DAT_00105058);
                  *(undefined8 *)(puVar9 + -0x1880) = 0x1023b1;
                  fflush(stdout);
                }
                else {
                  *(undefined8 *)(puVar9 + -0x1880) = 0x1023ce;
                  printf("Dear %s, WELCOME BACK\n",param_2);
                  *(undefined8 *)(puVar9 + -0x1880) = 0x1023dd;
                  fflush(stdout);
                }
                uVar8 = 1;
                goto LAB_001024ac;
              }
            }
          }
        }
      }
    }
    local_a864 = local_a864 + 1;
  } while( true );
```

So this is the most confusing part as I still don’t get what was the point. In this part it takes the long array resulting from the previous part and runs 6 nested loops looping from zero till the number contained in each of the indexes of the array. In the last loop it then checks a kind of mathematical expression and that the username entered is **BJIZ-HACKERLAB** and if those conditions are verified it sends us the flag.

Solve:

From all we've seen above the goal now is to get to this condition and see what we get.

In order to get to there the values in our correspondence long array should all be non zero because with a single value set to zero it won't be able to reach the condition which is contained inside of the last loop.

So my idea was to take the **CRC32_TABLE** compute all the results at every single index of the key (0-6) and take a value with a hex string of size 4 in order to at least have one match for each part of the key.

Doing so I didn't find much success. Couple more hours later I discovered this.

> *`%04X` format specifier truncates the hexadecimal string to the last 4 characters if it exceeds 4 characters in length.*

This was a revelation because what this meant was that any CRC32 hex value computed was valid just that we had to consider only the last 4 bytes for each of them as the **sprintf** function does when it gets a length format specifier.

With all that said I wrote a code to compute all the CRC32 values calculated inside of each interval of each part of the key.

```
import struct

CRC32_TABLE = b'REDACTED'

def bytes_to_int_array(byte_string):
      int_array = []
      chunk_size = 4
      num_chunks = len(byte_string) // chunk_size
      for i in range(num_chunks):
      chunk = byte_string[i * chunk_size : (i + 1) * chunk_size]
      int_value = struct.unpack('<I', chunk)[0]
      int_array.append(int_value)
      return int_array
```

```
CRC32_TABLE = bytes_to_int_array(CRC32_TABLE)

tab = [1,894,1,298,447,799236,1,223,-178,-1,0,178]

def crc32(data):
      crc = 0xFFFFFFFF
      for byte in data:
      crc = (crc >> 8) ^ CRC32_TABLE[(crc ^ byte) & 0xFF]
      return ~crc & 0xFFFFFFFF


def int_to_bytes(data):
      tab_d = []
      for i in range(0,4):
                  tab_d.append((data >> (8*i)) & 0xFF)
```

This code will print all the CRC32 values computed inside of each interval and print the last 4 bytes of their hex representation. We will just then have to take a single value into each interval and form a key that we'll test.

```
0 => D18F
0 => 7E61
0 => 1904

REDACTED...

0 => F960

1 => 8610
1 => 29FE
1 => 4E9B

REDACTED...

1 => B63A

2 => 62F3
2 => 0596
```

```
2 => AA78

REDACTED...

2 => 5F4F

3 => D18F
3 => 7E61
3 => 1904

REDACTED...

3 => 1F25

4 => BFF6
4 => D893
4 => 9F5A

REDACTED...

4 => 9609

5 => B6EA
5 => D18F
5 => 7E61
5 => 1904
5 => 21BD

REDACTED...

5 => 30FB
```

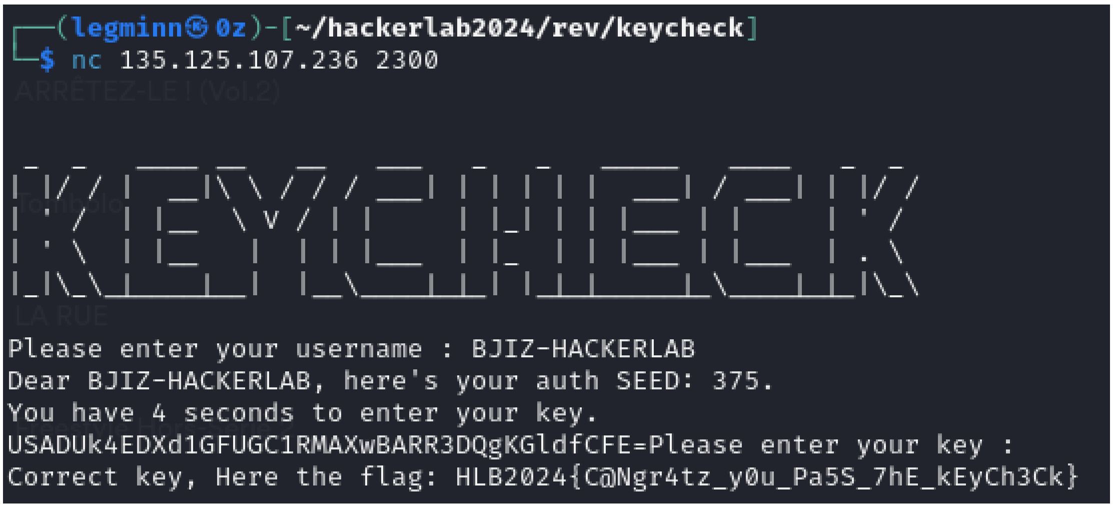
So know we form a key with a single value of each part like:

7E61-29FE-0596-7E61-D893-1904

We encode it using the script from the beginning

USADUk4EDXd1GFUGC1RMAXwBARR3DQgKGIdfCFE=

And we try to send it to the server.



```
┌──(legminn㉿0z)-[~/hackerlab2024/rev/keycheck]
└─$ nc 135.125.107.236 2300

 _  _________   _______ _   _ _____ ____ _  __
| |/ / ____\ \ / / ____| | | | ____/ ___| |/ /
| ' /|  _|  \ V / |   | |_| |  _|| |   | ' /
| . \| |___  | || |___|  _  | |__| |___| . \
|_|\_\_____| |_| \____|_| |_|_____\____|_|\_\

Please enter your username : BJIZ-HACKERLAB
Dear BJIZ-HACKERLAB, here's your auth SEED: 375.
You have 4 seconds to enter your key.
USADUk4EDXd1GFUGC1RMAXwBARR3DQgKGldfCFE=Please enter your key :
Correct key, Here the flag: HLB2024{C@Ngr4tz_y0u_Pa5S_7hE_kEyCh3Ck}
```

Directly the flag ???

That is why the last condition seemed to be a little bit weird for me as testing with any combination of valid keys at each part of the key will get us the flag. But still we have the flag. We won't ask for much. It has already been way too long lol.

flag: HLB2024{C@Ngr4tz_y0u_Pa5S_7hE_kEyCh3Ck}