Challenge 05

Flag: **Name: Fresh Candidate**

Email: fresh_candidate@recruitment.com

Access Key: 2521-2000-5370-7265-5787-1425

Running file command on linux allow us to see that it is an ELF executable. So I ran the file using test inputs:

```
(base) ntuintern@ntuintern-VirtualBox:~/Downloads$ file answer
answer: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
, interpreter /lib64/l, for GNU/Linux 2.6.32, BuildID[sha1]=bc88e8a6a08c19cde10
69b1c70954502d2487c00, not stripped
(base) ntuintern@ntuintern-VirtualBox:~/Downloads$ ./answer
Name: Fresh Candidate
Email: fresh_candidate@recruitment.com
Access  key: testing
Please Try Harder!!
(base) ntuintern@ntuintern-VirtualBox:~/Downloads$
```

I decided to use Ghidra to reverse the file. Checking the “Defined String”, I found some useful strings which brings me to the referencing function.

0040046d	__libc_start_main	"__libc_start_main"	ds
0040047f	__gmon_start__	"__gmon_start__"	ds
0040048e	GLIBC_2.4	"GLIBC_2.4"	ds
00400498	GLIBC_2.2.5	"GLIBC_2.2.5"	ds
004011e6	Name:	"Name: "	ds
004011ef	Email:	"Email: "	ds
004011f7	Access key:	"Access key: "	ds
00401205	Congratz you got ...	"Congratz you got...	ds
00401220	Please Try Harder!!	"Please Try Harde...	ds

Note: Variables and function signature has been modified for clearer understanding.

In the main function, the “Congratz you got the key!!” message is obtained when uVar3 is not “\0”. The value of uVar3 depends on the function verify(name,email,accesskey). Let’s check the function.

```
printf("Name: ");
fflush(stdout);
fgets(name,0x21,stdin);
sVar2 = strcspn(name,"\n");
name[sVar2] = '\0';
printf("Email: ");
fflush(stdout);
fgets(email,0x21,stdin);
sVar2 = strcspn(email,"\n");
email[sVar2] = '\0';
printf("Access  key: ");
fflush(stdout);
fgets(accesskey,0x21,stdin);
accesskeylength = strcspn(accesskey,"\n");
accesskey[accesskeylength] = '\0';
uVar3 = verify_key(name,email,accesskey);
if ((char)uVar3 == '\0') {
  puts("Please Try Harder!!");
  uVar4 = 0xffffffff;
}
else {
  puts("Congratz you got the key!!");
  uVar4 = 0;
}
```

From lines 35-44, the accesskey (token) is basically break into multiple parts with “-” as the delimiter. We’ll name it tokenPart.

```
35  i = 0;
36                      /* split accesskey into token with - */
37  token = strtok(accesskey,"-");
38  while (token != (char *)0x0) {
39                      /* str to int */
40    intToken = atoi(token);
41    tokenPart[(long)i] = intToken;
42    token = strtok((char *)0x0,"-");
43    i = i + 1;
44  }
```

At line 45, we know that there is 6 parts to the accesskey (tokenPart). Thus we know that the accesskey is in x-x-x-x-x-x format. From lines 50-64, email and name is being padded to ensure that they are of 32 characters. However, when I tried to key in 32 characters into the name and email field, it results in overflowing to the next input. This is because fgets takes in the new line character, “\n”, into the array. So technically we can only have 31 characters for our name and email, the maximum value for iVar1 in the if-statement at line 54 will be 31. This means that there will be one value padded no matter what. The padded value(pad[]) is stored in the memory.

```
45  if (i == 6) {
46    local_cc = 0;
47    local_c8 = 0;
48    local_88[0] = email;
49    local_88[1] = name;
50    while (local_c8 < 2) {
51      __s = local_88[(long)local_c8];
52      sVar3 = strlen(__s);
53      iVar1 = (int)sVar3;
54      if (iVar1 < 0x20) {
55        local_c4 = 0;
56        while (local_c4 < 0x20 - iVar1) {
57          __s[(long)(local_c4 + iVar1)] = pad[(long)(local_c4 + local_cc)];
58          local_c4 = local_c4 + 1;
59        }
60        local_cc = 0x20 - iVar1;
61        __s[0x20] = '\0';
62      }
63      local_c8 = local_c8 + 1;
64    }
```

```
                pad                                              X
006020a0 77 3c 1e        undefine...
         6b 39 13
         22 0f 24 ...
    006020a0 77               undefined177h              [0]
    006020a1 3c               undefined13Ch              [1]
    006020a2 1e               undefined11Eh              [2]
    006020a3 6b               undefined16Bh              [3]
    006020a4 39               undefined139h              [4]
    006020a5 13               undefined113h              [5]
    006020a6 22               undefined122h              [6]
    006020a7 0f               undefined10Fh              [7]
    006020a8 24               undefined124h              [8]
    006020a9 02               undefined102h              [9]
    006020aa 73               undefined173h              [10]
    006020ab 59               undefined159h              [11]
    006020ac 67               undefined167h              [12]
    006020ad 64               undefined164h              [13]
    006020ae 21               undefined121h              [14]
    006020af 73               undefined173h              [15]
    006020b0 17               undefined117h              [16]
    006020b1 1e               undefined11Eh              [17]
    006020b2 6d               undefined16Dh              [18]
    006020b3 5b               undefined15Bh              [19]
    006020b4 04               undefined104h              [20]
    006020b5 66               undefined166h              [21]
```

```
65    local_c0 = 0;
66    while (local_c0 < 0x20) {
67      email[(long)local_c0] = email[(long)local_c0] ^ 5;
68      name[(long)local_c0] = name[(long)local_c0] ^ 0xf;
69      local_c0 = local_c0 + 1;
70    }
```

Lines 65-70 modifies the email and name again.

Without spending more time to understand the code, I decided to check out the return values.

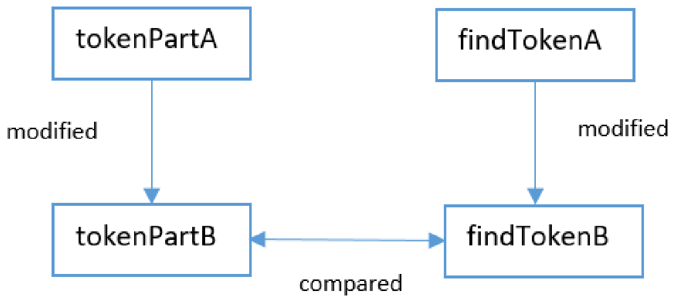
```
121     while (local_b0 < 6) {
122       local_38[(long)local_b0] = (local_58[(long)local_b0] * local_38[(long)local_b0]) % 10000;
123       local_b0 = local_b0 + 1;
124     }
125     local_d1 = 1;
126     local_ac = 0;
127     while (local_ac < 6) {
128       if (tokenPart[(long)local_ac] != local_38[(long)local_ac]) {
129         local_d1 = 0;
130       }
131       local_ac = local_ac + 1;
132     }
133     uVar2 = (ulong)local_d1;
134   }
135   else {
136     uVar2 = 0;
137   }
138   if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
139                     /* WARNING: Subroutine does not return */
140     __stack_chk_fail();
141   }
142   return uVar2;
143 }
```

uVar2 is returned. The else-statement in line 135 returns 0, that's when the if-statement at line 45 fails (tokenPart is not 6 parts). Within the if-statement, uVar2 = local_d1 in line 133. local_d1 is initialised at 1, which may then be modified to 0 in the while loop in line 127. That is when the tokenPart is compared with local_38. So from this, we understand that in order for uVar2 to return

1, the tokenPart must not only be of 6 parts but it must equate to local_38 as well. We can see that local_38 can be obtained from lines 102-124.

```
102   local_58[0] = 2;
103   local_58[1] = 4;
104   local_58[2] = 6;
105   local_58[3] = 8;
106   local_48 = 7;
107   local_44 = 5;
108   uVar2 = sumChars(email,0,10,1);
109   local_38[0] = (int)uVar2;
110   uVar2 = sumChars(email,10,0x19,1);
111   local_38[1] = (int)uVar2;
112   uVar2 = sumChars(email,0x19,0x20,1);
113   local_38[2] = (int)uVar2;
114   uVar2 = sumChars(name,0,0xd,1);
115   local_38[3] = (int)uVar2;
116   uVar2 = sumChars(name,0xd,0x14,1);
117   local_28 = (undefined4)uVar2;
118   uVar2 = sumChars(name,0x14,0x20,1);
119   local_24 = (undefined4)uVar2;
120   local_b0 = 0;
121   while (local_b0 < 6) {
122     local_38[(long)local_b0] = (local_58[(long)local_b0] * local_38[(long)local_b0]) % 10000;
123     local_b0 = local_b0 + 1;
124   }
```

We now look back to see if tokenPart is modified in any way before the comparison. Indeed, from lines 71-101, the tokenPart is modified. So now we understand how the algorithm works. When the user key in tokenPartA, tokenPartA is modified to tokenPartB, which is compared to findTokenB, which is obtained from findTokenA. Below is a diagram of the explanation.



We first obtain findTokenA(local_38), which is modified with local_58, email and name. We simply have to write a python script for lines 45-70 to obtain the modified name and email. In line 122, there are 6 iterations to modify each part of findToken. However local_58 and local_38 are initialised to hold 4 integer values, how can there be 6 iterations? In C, local variables are pushed into the stack when initialised. As we can see from the code, accessing them uses RBP (frame pointer) as a reference. This means that the next two indexes accessed for both local_38 and local_58 are the next two stack frames above them respectively. The below photo shows the variables accessed for their index[4] and index[5]. From this, we can write out a python code to obtain findTokenB.

```
                    ulong __stdcall verify_key(char * name, char * ema
     ulong              RAX:8           <RETURN>
     char *             RDI:8           name
     char *             RSI:8           email
     char *             RDX:8           accesskey
     int                EAX:4           intToken
     undefined8         Stack[-0x20]:8 local_20

     undefined4         Stack[-0x24]:4 local_24    local_38[5]
     undefined4         Stack[-0x28]:4 local_28    local_38[4]
     undefined4         Stack[-0x2c]:4 local_2c
     undefined4         Stack[-0x30]:4 local_30
     undefined4         Stack[-0x34]:4 local_34
     undefined4         Stack[-0x38]:4 local_38
     undefined4         Stack[-0x44]:4 local_44    local_58[5]
     undefined4         Stack[-0x48]:4 local_48    local_58[4]
     undefined4         Stack[-0x4c]:4 local_4c
     undefined4         Stack[-0x50]:4 local_50
     undefined4         Stack[-0x54]:4 local_54
     undefined4         Stack[-0x58]:4 local_58
     int[8]             Stack[-0x78]... tokenPart
```

Since tokenPartB must be equal to findTokenB, we can simply reverse the steps from lines 71-101 to obtain the accesskey(tokenPartA). As there are intermediary values during the modification of tokenPart from lines 71-101, we have to obtain these intermediary values in order to reverse the steps. Thus, we have to write the forward execution script before reversing. We have to rewrite the function for sumChars and swapArr as well. Running my script, we get our access key:

```
Using the access key below, tokenPartB:
[2074, 6208, 4134, 296, 3353, 3665]
findTokenB:
[2074, 6208, 4134, 296, 3353, 3665]
Access Key: 2521-2000-5370-7265-5787-1425
>>>
```

Inputting our results:

```
Name: Fresh Candidate
Email: fresh_candidate@recruitment.com
Access  key: 2521-2000-5370-7265-5787-1425
Congratz you got the key!!
(base) ntuintern@ntuintern-VirtualBox:~/Downloads$
```

Since the challenge is to find one key for a chosen pair, I did not fully reverse the executable in my script (Did not reverse the token delimiting part). Not too sure if it's because of that, I found out that there are some instances the access key will not work, when part of the access key is not 4 digits.

```
Name: a
Email: a
Access  key: 4618-5740-9589-602-1105-3663
Please Try Harder!!
```

```
Using the access key below, tokenPartB:
[1144, 4356, 2472, 6944, 2905, 3705]
findTokenB:
[1144, 4356, 2472, 6944, 2905, 3705]
Access Key: 4618-5740-9589-602-1105-3663
```

There are some instances which can crash the program.

```
Name: aa
Email: aa
Access  key: 1-1-1-1-1-1
Floating point exception (core dumped)
```

```
Traceback (most recent call last):
  File "D:\MHA Challenge\Challenge 5(Undone)\answerRE.py", line 102, in <module>
    tokenPart[local_b4] = uVar2 % uVar4 + tokenPart[local_b4]
ZeroDivisionError: integer division or modulo by zero
```

My name and personal email wouldn't work too ☹

```
Name: Chong Yu
Email: [redacted]@gmail.com
Access  key: -1579--561--694-6397-3625-7677
Please Try Harder!!
```

```
Using the access key below, tokenPartB:
[2098, 5040, 2382, 8392, 4074, 3240]
findTokenB:
[2098, 5040, 2382, 8392, 4074, 3240]
Access Key: -1579--561--694-6397-3625-7677
```

But my name and school email works though!

```
Name: lee chong yu
Email: [redacted]@e.ntu.edu.sg
Access  key: 1836-4010-8099-6822-7023-2902
Congratz you got the key!!
```

```
Using the access key below, tokenPartB:
[1514, 5228, 1926, 56, 3500, 4020]
findTokenB:
[1514, 5228, 1926, 56, 3500, 4020]
Access Key: 1836-4010-8099-6822-7023-2902
```