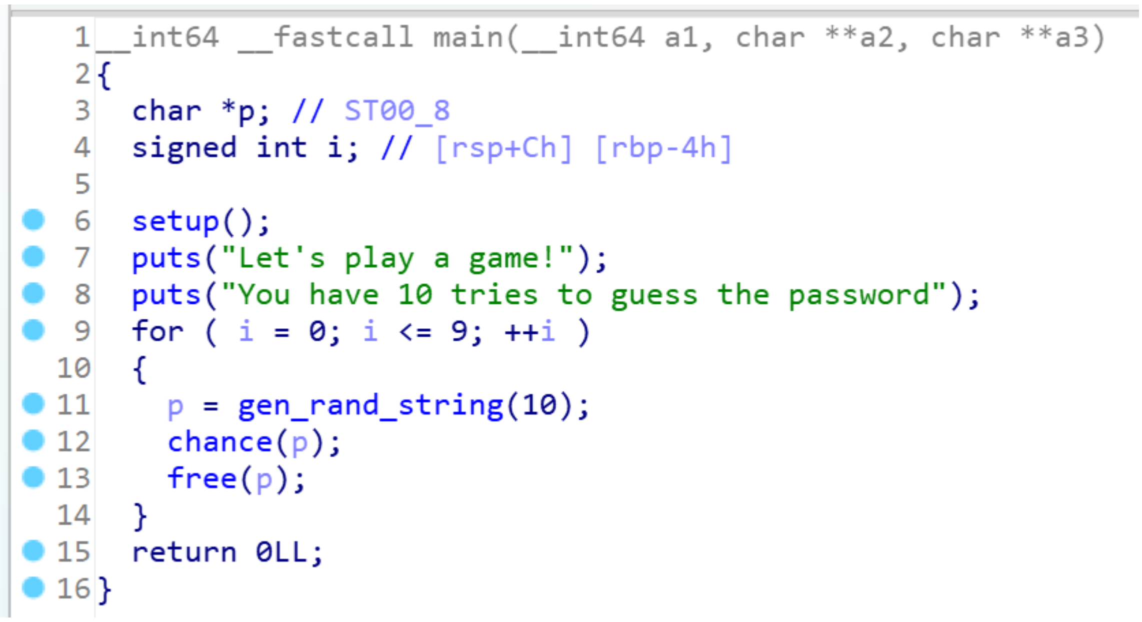
# Reverse Engineering Lab 3

Dilirici Radu, 510

## 4.1 Reverse engineering with spoilers

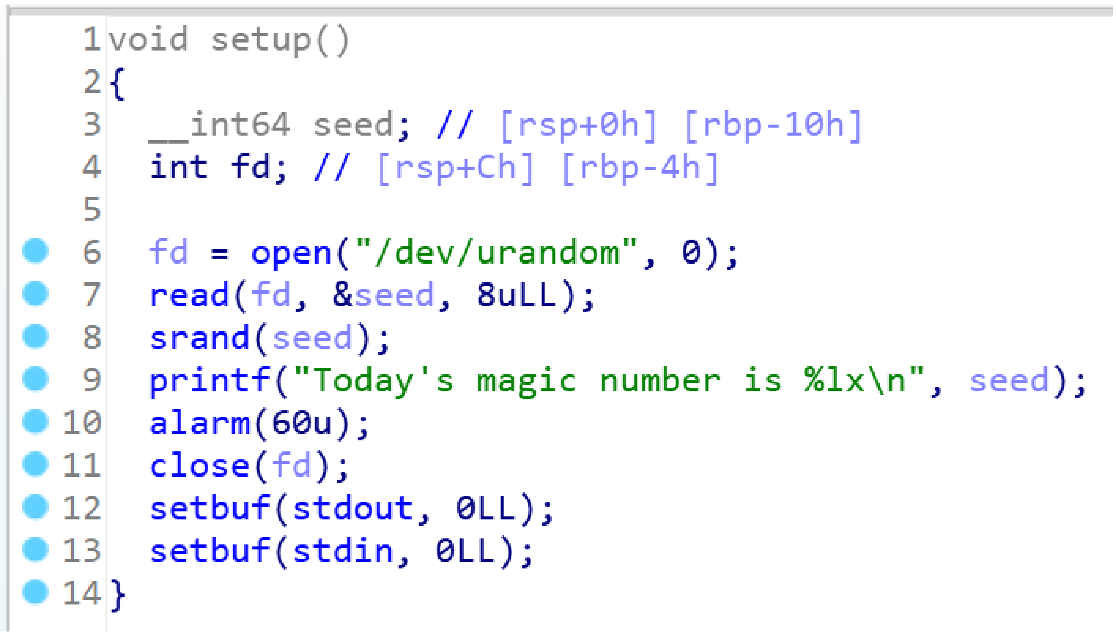
Dupa urmarea pasilor din laborator am ajuns la urmatoarele configuratii (care se pot observa si in salvarea IDA):

### main



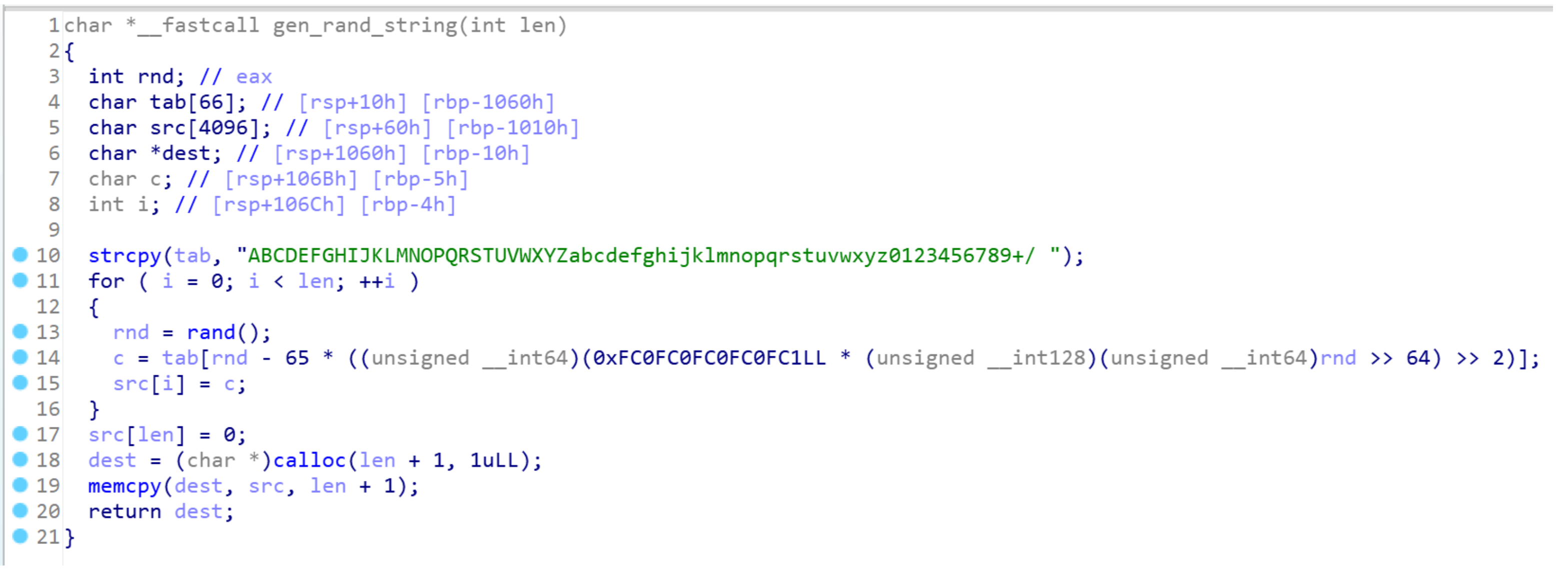
```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  char *p; // ST00_8
  signed int i; // [rsp+Ch] [rbp-4h]

  setup();
  puts("Let's play a game!");
  puts("You have 10 tries to guess the password");
  for ( i = 0; i <= 9; ++i )
  {
    p = gen_rand_string(10);
    chance(p);
    free(p);
  }
  return 0LL;
}
```

### setup



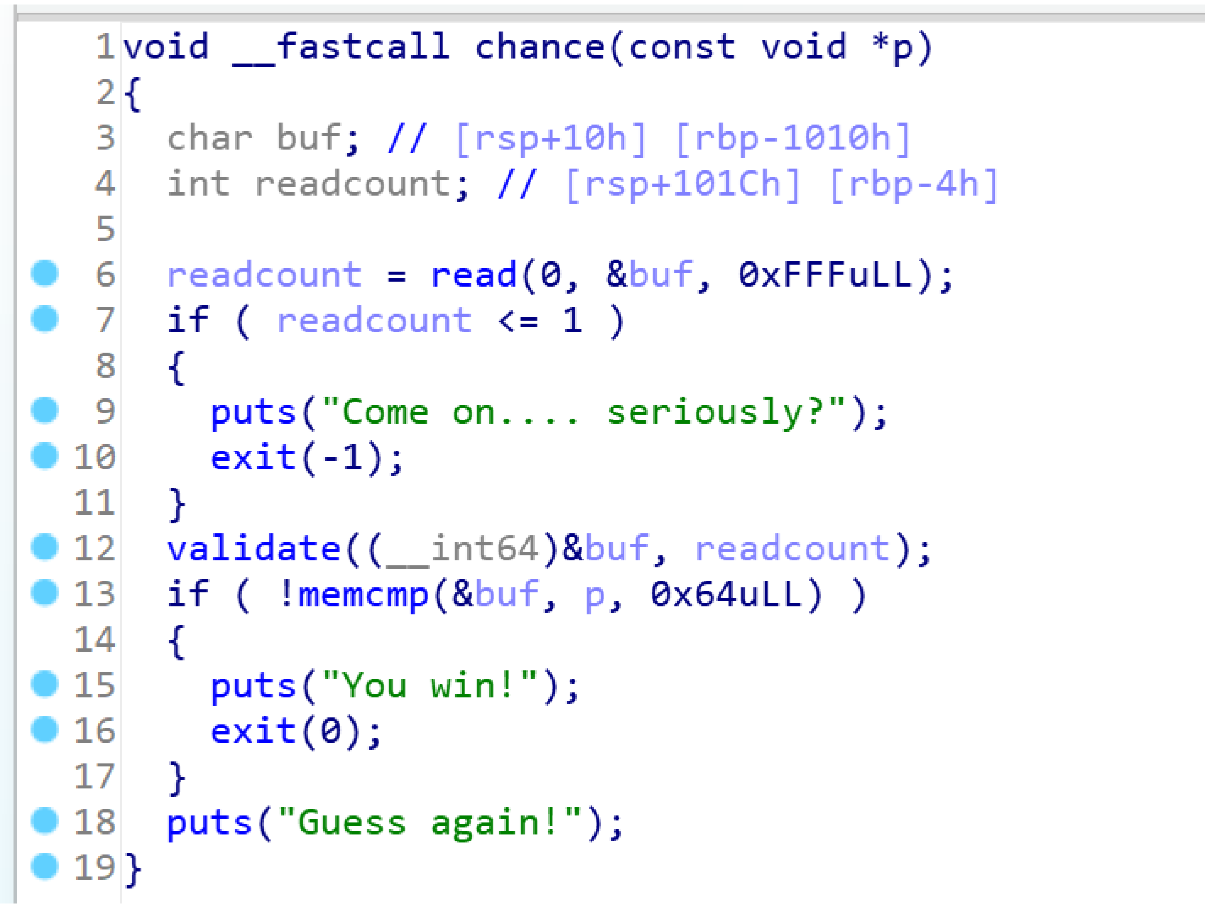
```
void setup()
{
  __int64 seed; // [rsp+0h] [rbp-10h]
  int fd; // [rsp+Ch] [rbp-4h]

  fd = open("/dev/urandom", 0);
  read(fd, &seed, 8uLL);
  srand(seed);
  printf("Today's magic number is %lx\n", seed);
  alarm(60u);
  close(fd);
  setbuf(stdout, 0LL);
  setbuf(stdin, 0LL);
}
```

## gen_rand_string



```c
char *__fastcall gen_rand_string(int len)
{
  int rnd; // eax
  char tab[66]; // [rsp+10h] [rbp-1060h]
  char src[4096]; // [rsp+60h] [rbp-1010h]
  char *dest; // [rsp+1060h] [rbp-10h]
  char c; // [rsp+106Bh] [rbp-5h]
  int i; // [rsp+106Ch] [rbp-4h]

  strcpy(tab, "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ ");
  for ( i = 0; i < len; ++i )
  {
    rnd = rand();
    c = tab[rnd - 65 * ((unsigned __int64)(0xFC0FC0FC0FC0FC1LL * (unsigned __int128)(unsigned __int64)rnd >> 64) >> 2)];
    src[i] = c;
  }
  src[len] = 0;
  dest = (char *)calloc(len + 1, 1uLL);
  memcpy(dest, src, len + 1);
  return dest;
}
```

## chance

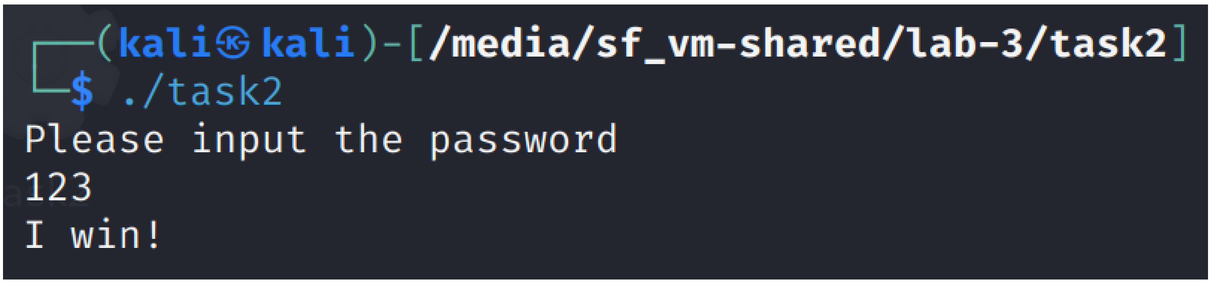


```c
void __fastcall chance(const void *p)
{
  char buf; // [rsp+10h] [rbp-1010h]
  int readcount; // [rsp+101Ch] [rbp-4h]

  readcount = read(0, &buf, 0xFFFuLL);
  if ( readcount <= 1 )
  {
    puts("Come on.... seriously?");
    exit(-1);
  }
  validate((__int64)&buf, readcount);
  if ( !memcmp(&buf, p, 0x64uLL) )
  {
    puts("You win!");
    exit(0);
  }
  puts("Guess again!");
}
```

Dupa modificari, am exportat noul cod intr-un fisier binar si l-am rulat, pentru a ma asigura ca am mentinut functionalitatea.

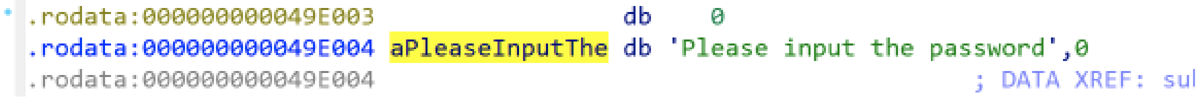
## 4.2 Statically linked crackme

La rularea programului observam ca este ceruta o parola.



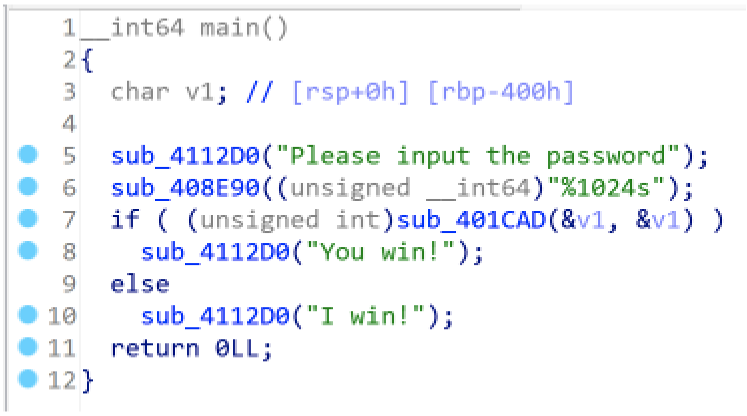
```
┌──(kali㉿kali)-[/media/sf_vm-shared/lab-3/task2]
└─$ ./task2
Please input the password
123
I win!
```

Am intrat in sectiunea **.rodata** si am cautat stringul *“Please input the password”*, presupunand ca acesta este afisat la inceputul functiei **main**.



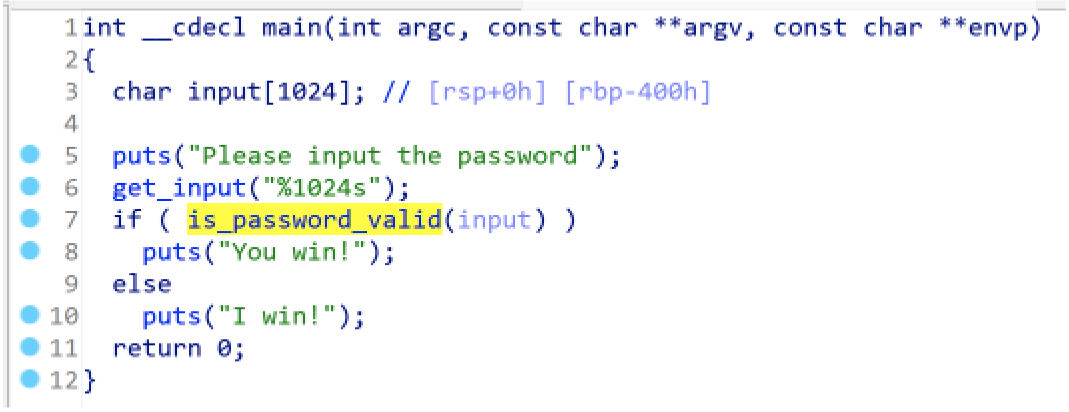
```
.rodata:000000000049E003                  db    0
.rodata:000000000049E004 aPleaseInputThe db 'Please input the password',0
.rodata:000000000049E004                                         ; DATA XREF: sul
```

Pare ca aceasta functie detine toata logica, asa ca am presupus ca este intr-adevar **main**.



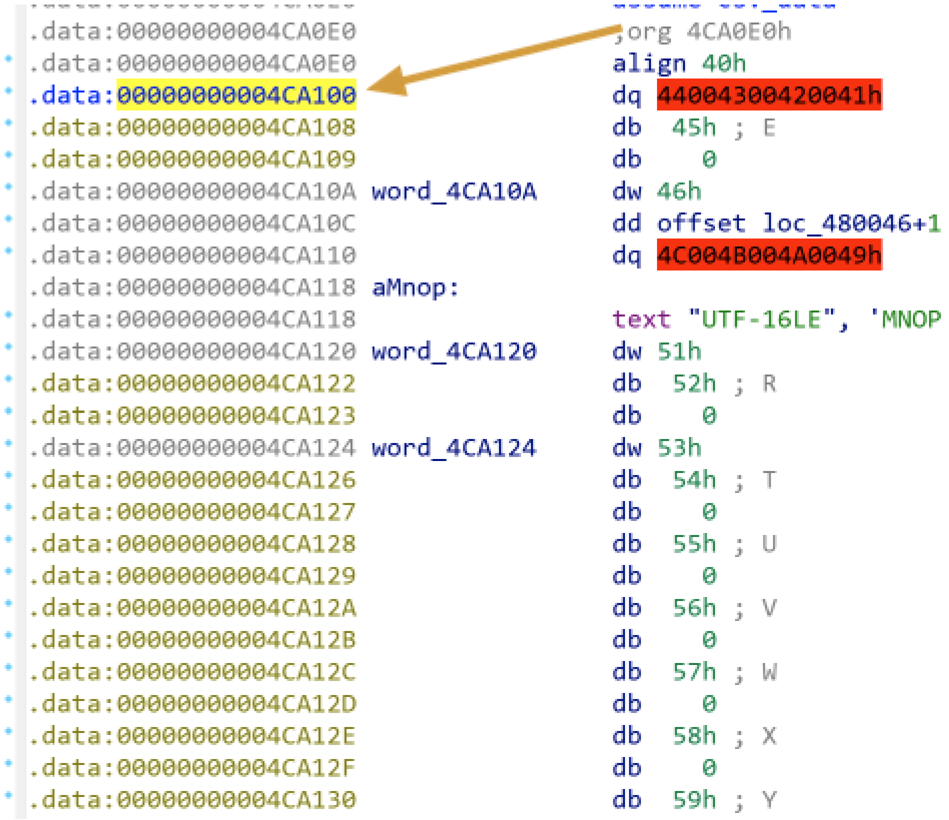
```
__int64 main()
{
  char v1; // [rsp+0h] [rbp-400h]

  sub_4112D0("Please input the password");
  sub_408E90((unsigned __int64)"%1024s");
  if ( (unsigned int)sub_401CAD(&v1, &v1) )
    sub_4112D0("You win!");
  else
    sub_4112D0("I win!");
  return 0LL;
}
```

Dupa o analiza, am ajuns la concluzia ca functia din interiorul **if**-ului este cea care verifica daca parola este cea corecta.



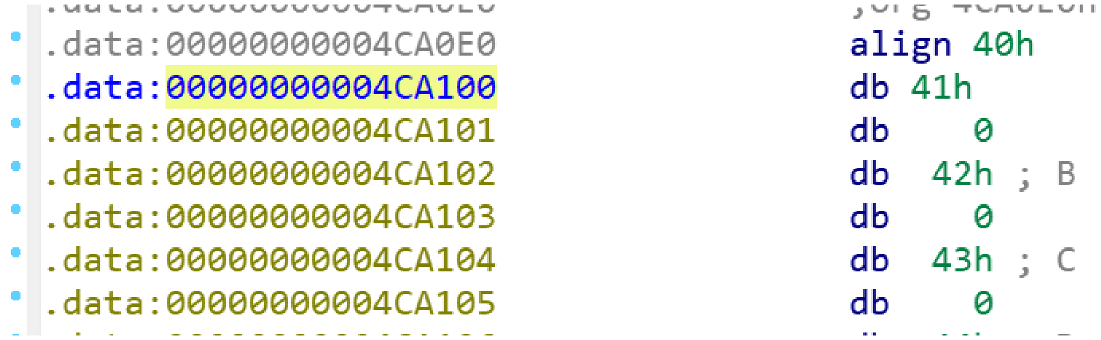
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char input[1024]; // [rsp+0h] [rbp-400h]

  puts("Please input the password");
  get_input("%1024s");
  if ( is_password_valid(input) )
    puts("You win!");
  else
    puts("I win!");
  return 0;
}
```

Am urmat instructiunile si am mers la locatia unei variabile **word_…** si am gasit inceputul alfabetului. Initial am cautat litera **A** mare (41 in hexadecimal), pentru ca este prima ca ordine in tabelul ASCII, insa nu am gasit-o. Am prespus ca este la locatia din urmatoarea poza, pentru ca era singura optiune inainte de **E**.



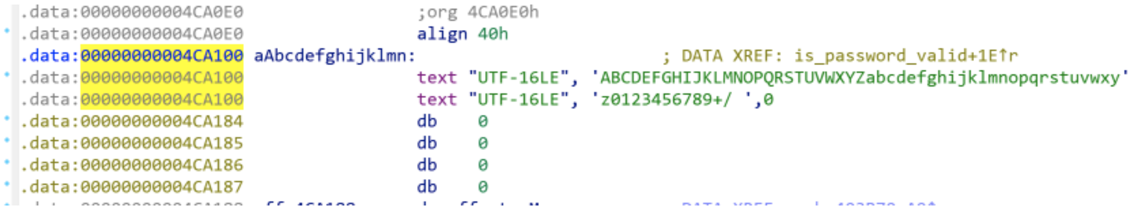
```
.data:00000000004CA0E0                          ;org 4CA0E0h
.data:00000000004CA0E0                          align 40h
.data:00000000004CA100                          dq 44004300420041h
.data:00000000004CA108                          db  45h ; E
.data:00000000004CA109                          db    0
.data:00000000004CA10A word_4CA10A              dw 46h
.data:00000000004CA10C                          dd offset loc_480046+1
.data:00000000004CA110                          dq 4C004B004A0049h
.data:00000000004CA118 aMnop:
.data:00000000004CA118                          text "UTF-16LE", 'MNOP
.data:00000000004CA120 word_4CA120              dw 51h
.data:00000000004CA122                          db  52h ; R
.data:00000000004CA123                          db    0
.data:00000000004CA124 word_4CA124              dw 53h
.data:00000000004CA126                          db  54h ; T
.data:00000000004CA127                          db    0
.data:00000000004CA128                          db  55h ; U
.data:00000000004CA129                          db    0
.data:00000000004CA12A                          db  56h ; V
.data:00000000004CA12B                          db    0
.data:00000000004CA12C                          db  57h ; W
.data:00000000004CA12D                          db    0
.data:00000000004CA12E                          db  58h ; X
.data:00000000004CA12F                          db    0
.data:00000000004CA130                          db  59h ; Y
```

Ulterior, am aflat ca pot schimba interpretarea tipului de date (in cazul asta de la **dq** la **4 db**). Prin aceasta metoda se poate vedea mult mai usor locatia lui **A**.



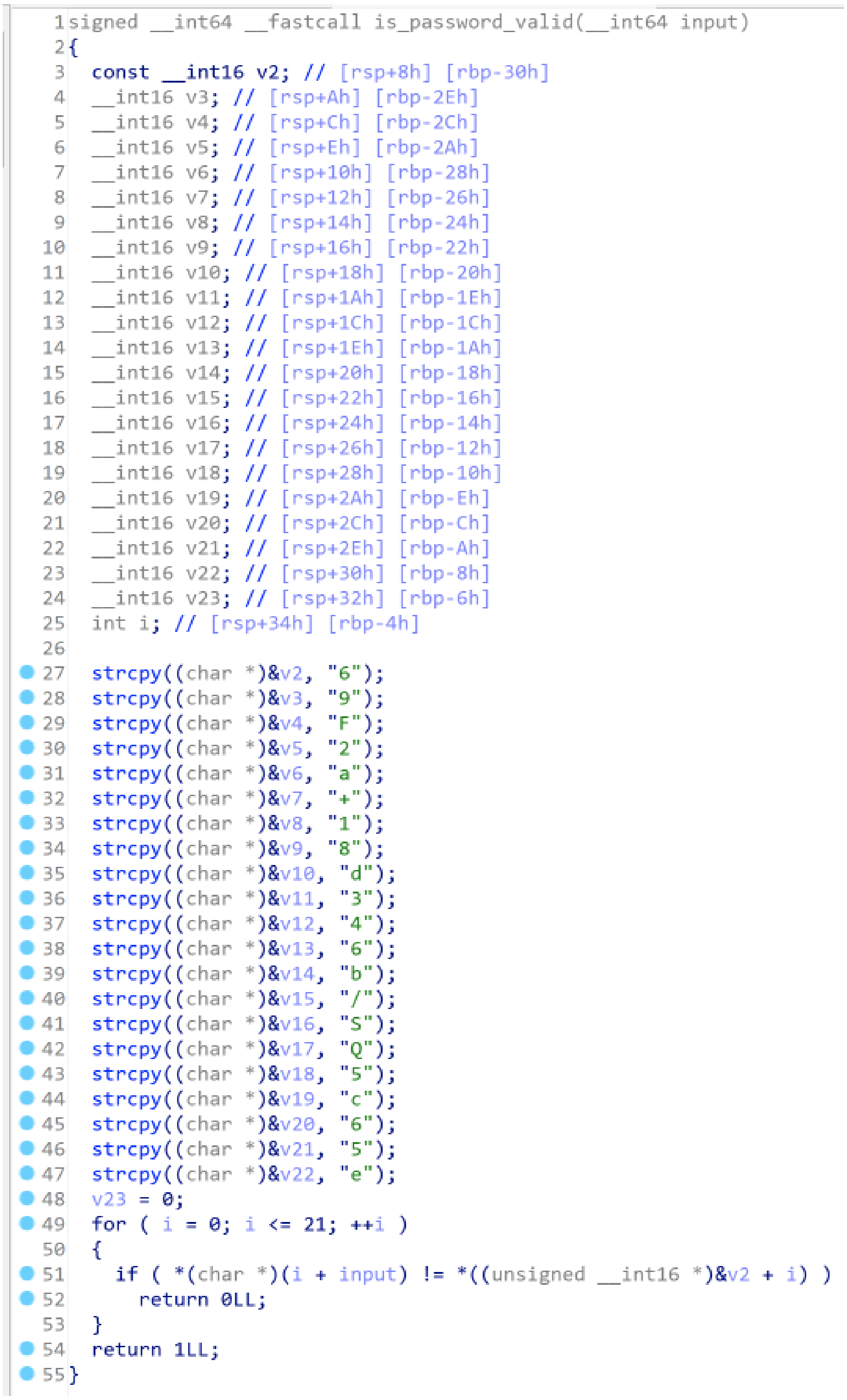
```
.data:00000000004CA0E0          align 40h
.data:00000000004CA100          db 41h
.data:00000000004CA101          db    0
.data:00000000004CA102          db  42h ; B
.data:00000000004CA103          db    0
.data:00000000004CA104          db  43h ; C
.data:00000000004CA105          db    0
```

Dupa declararea sectiunii ca string, acesta arata asa:



```
.data:00000000004CA0E0                       ;org 4CA0E0h
.data:00000000004CA0E0                       align 40h
.data:00000000004CA100 aAbcdefghijklmn:                          ; DATA XREF: is_password_valid+1E↑r
.data:00000000004CA100                       text "UTF-16LE", 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy'
.data:00000000004CA100                       text "UTF-16LE", 'z0123456789+/ ',0
.data:00000000004CA184                       db    0
.data:00000000004CA185                       db    0
.data:00000000004CA186                       db    0
.data:00000000004CA187                       db    0
```

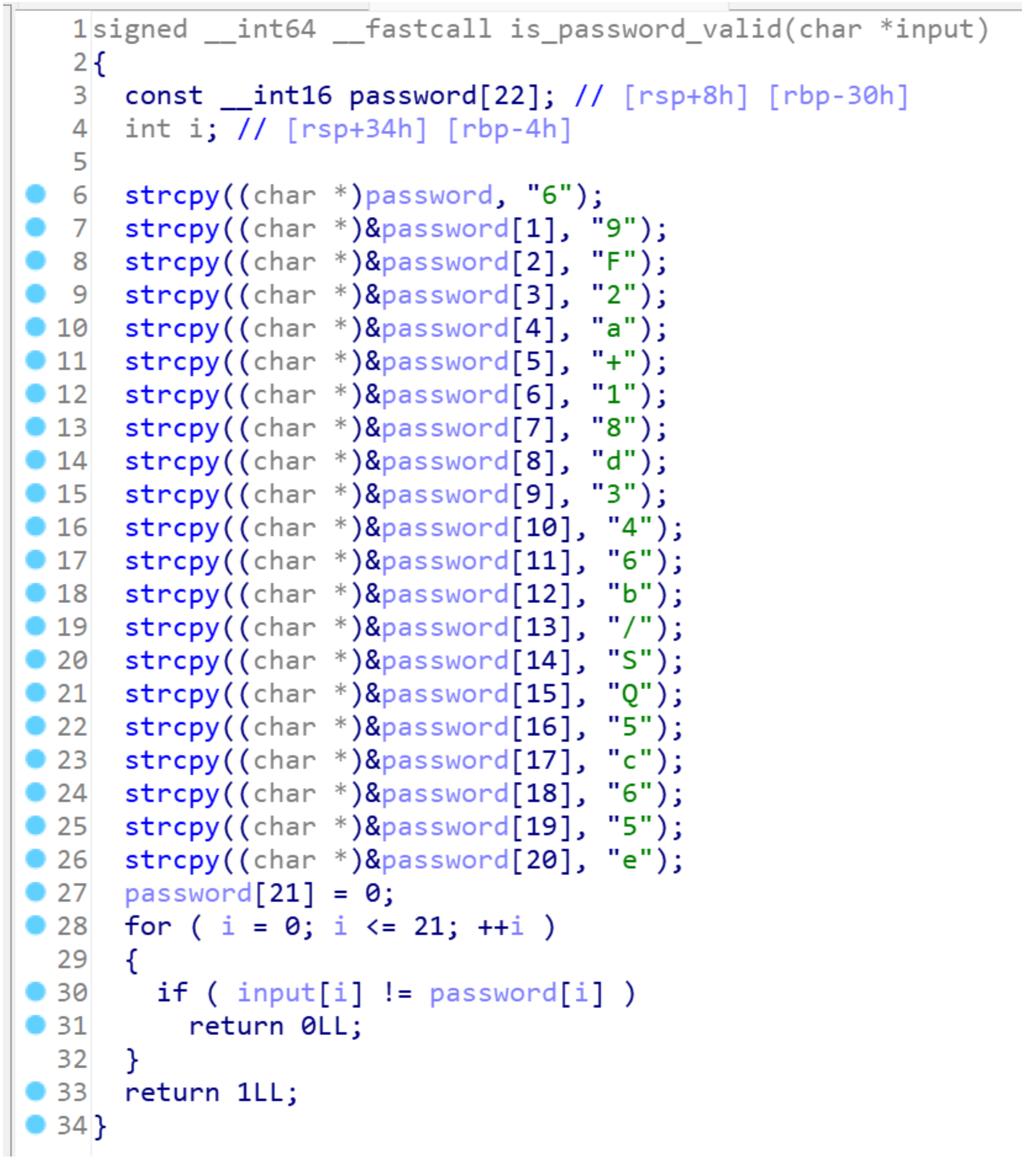
Iar dupa acest pas se poate observa mult mai bine ce se intampla in functia de verificare a parolei:



```
signed __int64 __fastcall is_password_valid(__int64 input)
{
  const __int16 v2; // [rsp+8h] [rbp-30h]
  __int16 v3; // [rsp+Ah] [rbp-2Eh]
  __int16 v4; // [rsp+Ch] [rbp-2Ch]
  __int16 v5; // [rsp+Eh] [rbp-2Ah]
  __int16 v6; // [rsp+10h] [rbp-28h]
  __int16 v7; // [rsp+12h] [rbp-26h]
  __int16 v8; // [rsp+14h] [rbp-24h]
  __int16 v9; // [rsp+16h] [rbp-22h]
  __int16 v10; // [rsp+18h] [rbp-20h]
  __int16 v11; // [rsp+1Ah] [rbp-1Eh]
  __int16 v12; // [rsp+1Ch] [rbp-1Ch]
  __int16 v13; // [rsp+1Eh] [rbp-1Ah]
  __int16 v14; // [rsp+20h] [rbp-18h]
  __int16 v15; // [rsp+22h] [rbp-16h]
  __int16 v16; // [rsp+24h] [rbp-14h]
  __int16 v17; // [rsp+26h] [rbp-12h]
  __int16 v18; // [rsp+28h] [rbp-10h]
  __int16 v19; // [rsp+2Ah] [rbp-Eh]
  __int16 v20; // [rsp+2Ch] [rbp-Ch]
  __int16 v21; // [rsp+2Eh] [rbp-Ah]
  __int16 v22; // [rsp+30h] [rbp-8h]
  __int16 v23; // [rsp+32h] [rbp-6h]
  int i; // [rsp+34h] [rbp-4h]

  strcpy((char *)&v2, "6");
  strcpy((char *)&v3, "9");
  strcpy((char *)&v4, "F");
  strcpy((char *)&v5, "2");
  strcpy((char *)&v6, "a");
  strcpy((char *)&v7, "+");
  strcpy((char *)&v8, "1");
  strcpy((char *)&v9, "8");
  strcpy((char *)&v10, "d");
  strcpy((char *)&v11, "3");
  strcpy((char *)&v12, "4");
  strcpy((char *)&v13, "6");
  strcpy((char *)&v14, "b");
  strcpy((char *)&v15, "/");
  strcpy((char *)&v16, "S");
  strcpy((char *)&v17, "Q");
  strcpy((char *)&v18, "5");
  strcpy((char *)&v19, "c");
  strcpy((char *)&v20, "6");
  strcpy((char *)&v21, "5");
  strcpy((char *)&v22, "e");
  v23 = 0;
  for ( i = 0; i <= 21; ++i )
  {
    if ( *(char *)(i + input) != *((unsigned __int16 *)&v2 + i) )
      return 0LL;
  }
  return 1LL;
}
```

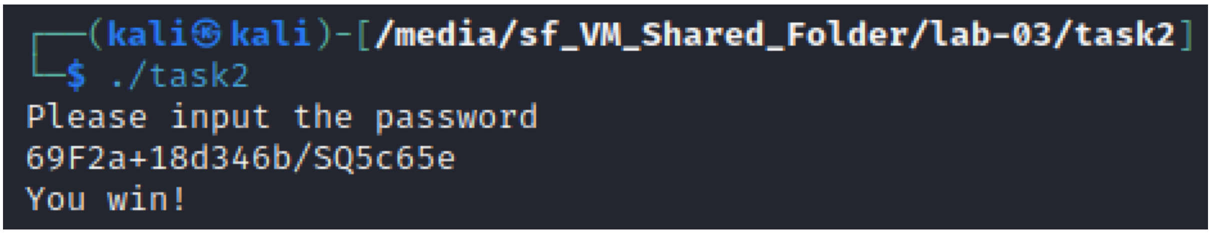
Este verificat, caracter cu caracter, daca inputul este egal cu stringul **69F2a+18d346b/SQ5c65e**. In interpretarea initiala erau folsite diferite offset-uri pentru a extrage anumite caractere din alfabet. Acum IDA a completat singur caracterele folosite.

Am restrans codul functiei modificand tipul de date pentru v2 intr-o lista de 22 de elemente (cate variabile sunt). Am incercat si modificarea la o lista de caractere, insa nu mi s-a parut ca a ajutat. Codul era similar, dar parcurgerea parolei corecte se facea din doua in doua pozitii.



```
signed __int64 __fastcall is_password_valid(char *input)
{
  const __int16 password[22]; // [rsp+8h] [rbp-30h]
  int i; // [rsp+34h] [rbp-4h]

  strcpy((char *)password, "6");
  strcpy((char *)&password[1], "9");
  strcpy((char *)&password[2], "F");
  strcpy((char *)&password[3], "2");
  strcpy((char *)&password[4], "a");
  strcpy((char *)&password[5], "+");
  strcpy((char *)&password[6], "1");
  strcpy((char *)&password[7], "8");
  strcpy((char *)&password[8], "d");
  strcpy((char *)&password[9], "3");
  strcpy((char *)&password[10], "4");
  strcpy((char *)&password[11], "6");
  strcpy((char *)&password[12], "b");
  strcpy((char *)&password[13], "/");
  strcpy((char *)&password[14], "S");
  strcpy((char *)&password[15], "Q");
  strcpy((char *)&password[16], "5");
  strcpy((char *)&password[17], "c");
  strcpy((char *)&password[18], "6");
  strcpy((char *)&password[19], "5");
  strcpy((char *)&password[20], "e");
  password[21] = 0;
  for ( i = 0; i <= 21; ++i )
  {
    if ( input[i] != password[i] )
      return 0LL;
  }
  return 1LL;
}
```

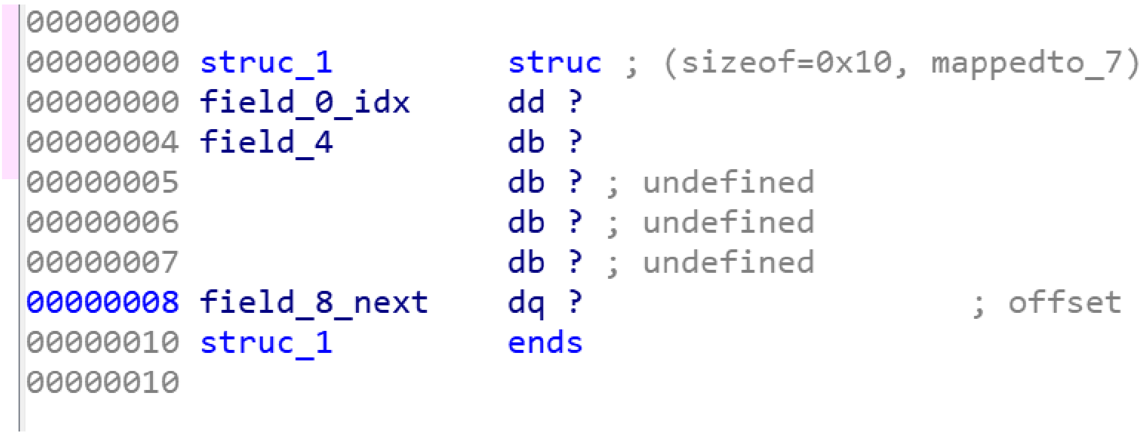
Am rulat din nou programul utilizand aceasta parola, iar rezultatul a fost un succes.



```
┌──(kali㉿kali)-[/media/sf_VM_Shared_Folder/lab-03/task2]
└─$ ./task2
Please input the password
69F2a+18d346b/SQ5c65e
You win!
```

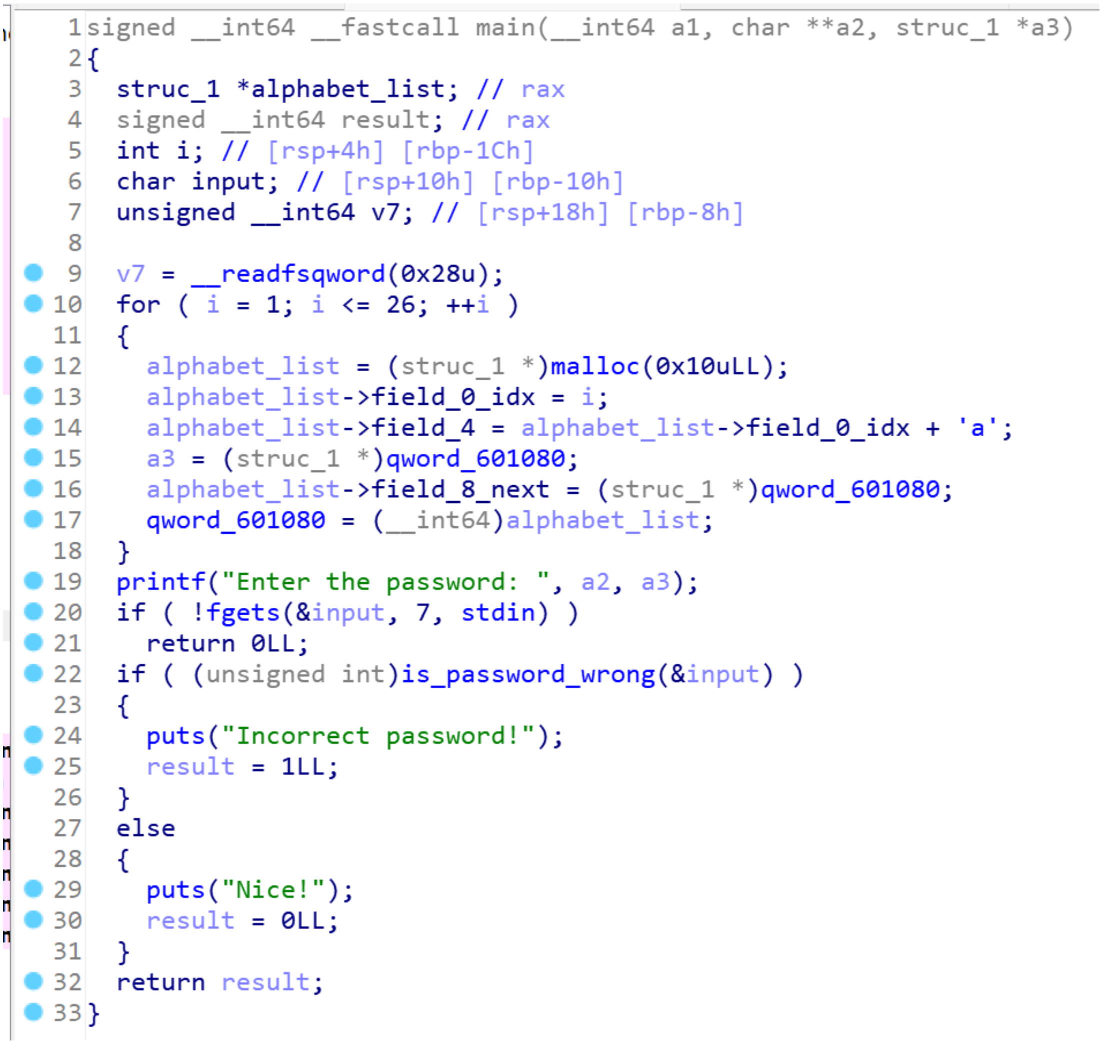
## 4.3 Data Structures

Am declarat structura si am folosit-o in **main** si in functia de verificare:

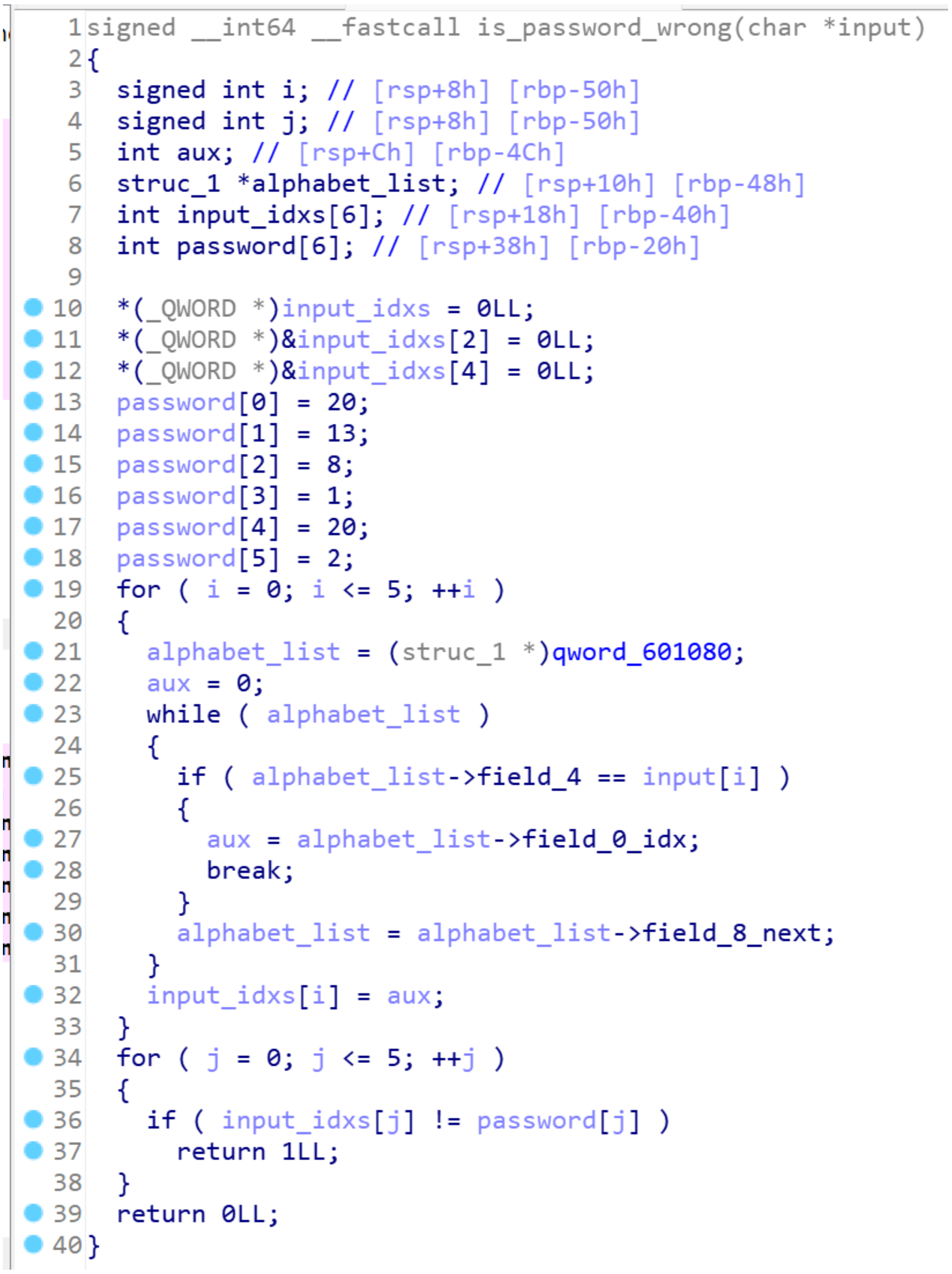


```
00000000
00000000 struc_1         struc ; (sizeof=0x10, mappedto_7)
00000000 field_0_idx     dd ?
00000004 field_4         db ?
00000005                 db ? ; undefined
00000006                 db ? ; undefined
00000007                 db ? ; undefined
00000008 field_8_next    dq ?                    ; offset
00000010 struc_1         ends
00000010
```

In **main** pare ca se retin toate literele mici ale alfabetului in lista inlantuita, impreuna cu indecsii la care se gasesc caracterele in alfabet. Apoi sunt citite primele 6 caractere de la tastatura si sunt trimise mai departe catre verificare.



```
signed __int64 __fastcall main(__int64 a1, char **a2, struc_1 *a3)
{
  struc_1 *alphabet_list; // rax
  signed __int64 result; // rax
  int i; // [rsp+4h] [rbp-1Ch]
  char input; // [rsp+10h] [rbp-10h]
  unsigned __int64 v7; // [rsp+18h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  for ( i = 1; i <= 26; ++i )
  {
    alphabet_list = (struc_1 *)malloc(0x10uLL);
    alphabet_list->field_0_idx = i;
    alphabet_list->field_4 = alphabet_list->field_0_idx + 'a';
    a3 = (struc_1 *)qword_601080;
    alphabet_list->field_8_next = (struc_1 *)qword_601080;
    qword_601080 = (__int64)alphabet_list;
  }
  printf("Enter the password: ", a2, a3);
  if ( !fgets(&input, 7, stdin) )
    return 0LL;
  if ( (unsigned int)is_password_wrong(&input) )
  {
    puts("Incorrect password!");
    result = 1LL;
  }
  else
  {
    puts("Nice!");
    result = 0LL;
  }
  return result;
}
```



```
signed __int64 __fastcall is_password_wrong(char *input)
{
  signed int i; // [rsp+8h] [rbp-50h]
  signed int j; // [rsp+8h] [rbp-50h]
  int aux; // [rsp+Ch] [rbp-4Ch]
  struc_1 *alphabet_list; // [rsp+10h] [rbp-48h]
  int input_idxs[6]; // [rsp+18h] [rbp-40h]
  int password[6]; // [rsp+38h] [rbp-20h]

  *(_QWORD *)input_idxs = 0LL;
  *(_QWORD *)&input_idxs[2] = 0LL;
  *(_QWORD *)&input_idxs[4] = 0LL;
  password[0] = 20;
  password[1] = 13;
  password[2] = 8;
  password[3] = 1;
  password[4] = 20;
  password[5] = 2;
  for ( i = 0; i <= 5; ++i )
  {
    alphabet_list = (struc_1 *)qword_601080;
    aux = 0;
    while ( alphabet_list )
    {
      if ( alphabet_list->field_4 == input[i] )
      {
        aux = alphabet_list->field_0_idx;
        break;
      }
      alphabet_list = alphabet_list->field_8_next;
    }
    input_idxs[i] = aux;
  }
  for ( j = 0; j <= 5; ++j )
  {
    if ( input_idxs[j] != password[j] )
      return 1LL;
  }
  return 0LL;
}
```

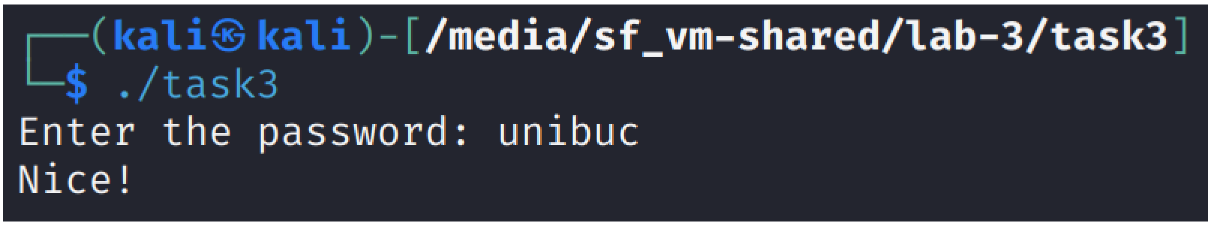
In functia de verificare am facut diferite modificari de tipuri. Dupa mai multe incercari am ajuns la varianta din poza, care parea sa fie cea mai intuitiva. In functie se intampla urmatoarele:

1. Este contruita lista indecsilor caracterelor din parola corecta (aici **password**)
2. Inputul este parcurs, iar pentru fiecare element:
   a. Se cauta cu ce element este egal, din lista inlantuita construita in main. Pentru ca stim ca in ea sunt literele mici ale alfabetului, asta inseamna ca se extrage indexul caracterului din input (0 pentru a, 1 pentru b, etc.)

    b. Se adauga acest index intr-o lista statica (aici input_idxs)
3. Lista de indecsi rezultata este comparata cu cea retinuta in **password**. Inputul este valid daca contin aceleasi elemente.

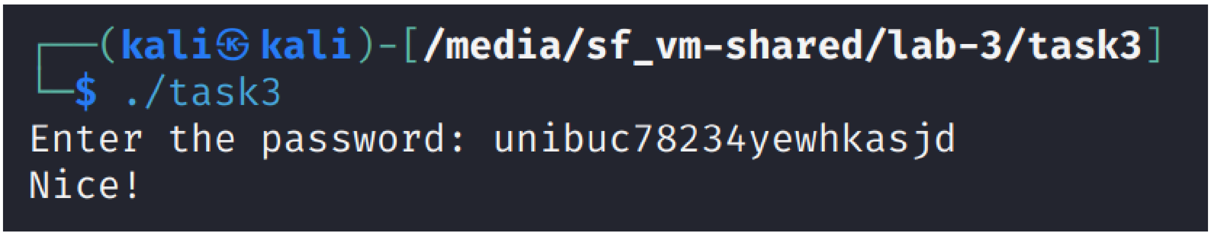
Practic, se verifica daca indecsii caracterelor inputului sunt 20, 13, 8, 1, 20, 2. Tradus, asta inseamna u, n, i, b, u, c -> **unibuc**.

Am rulat programul si am introdus parola aceasta, care s-a dovedit a fi cea corecta.



```
┌──(kali㉿kali)-[/media/sf_vm-shared/lab-3/task3]
└─$ ./task3
Enter the password: unibuc
Nice!
```

Pentru ca sunt citite doar primele 6 caractere, orice string care incepe cu **unibuc** este o parola valida.



```
┌──(kali㉿kali)-[/media/sf_vm-shared/lab-3/task3]
└─$ ./task3
Enter the password: unibuc78234yewhkasjd
Nice!
```