pjenik's Hidden Password

Saturday, February 19, 2022 5:35 PM

Examining the file in Ghidra, we find the good password string

00102004	Invalid password!	"Invalid password!"	ds
00102016	Good password!	"Good password!"	ds
00102025	Usage: %s <password>	"Usage: %s <password>\n"	ds

We see a key comparison before the program decides whether or not to jump to bad password

```
00101547 83 7d c8     CMP     dword ptr [RBP + local_40],0x0
         00
0010154b 74 0e        JZ      LAB_0010155b
0010154d 48 8d 3d     LEA     RDI,[s_Good_password!_00102016]        = "Good password!"
         c2 0a 00
         00
00101554 e8 d7 fa     CALL    <EXTERNAL>::puts                       int puts(char * __s)
         ff ff
00101559 eb 2c        JMP     LAB_00101587

                  LAB_0010155b                          XREF[1]:   0010154b(j)
0010155b 48 8d 3d     LEA     RDI,[s_Invalid_password!_00102004]     = "Invalid password!"
         a2 0a 00
```

We also see a long list of what are probably strings in the same function...

```
0010119e 53           PUSH    RBX
0010119f 48 81 ec     SUB     RSP,0x128
         28 01 00
         00
001011a6 89 bd dc     MOV     dword ptr [RBP + local_12c],EDI
         fe ff ff
001011ac 48 89 b5     MOV     qword ptr [RBP + local_138],RSI
         d0 fe ff
         ff
001011b3 48 b8 5b     MOV     RAX,0x28bf16683619a05b
         a0 19 36
         68 16 bf...
001011bd 48 ba 99     MOV     RDX,0x4dd3ce3a2552e799
         e7 52 25
         3a ce d3...
001011c7 48 89 85     MOV     qword ptr [RBP + local_108],RAX
         00 ff ff
         ff
001011ce 48 89 95     MOV     qword ptr [RBP + local_100],RDX
         08 ff ff
         ff
001011d5 48 b8 49     MOV     RAX,-0x5a12641e7dcfbbb7
         44 30 82
         e1 9b ed...
001011df 48 ba 37     MOV     RDX,0x6e27e1473b191037
         10 19 3b
         47 e1 27...
```

```
  uint local_1c;

  local_108 = 0x28bf16683619a05b;
  local_100 = 0x4dd3ce3a2552e799;
  local_f8 = 0xa5ed9be182304449;
  local_f0 = 0x6e27e1473b191037;
  local_e8 = 0x6da9ec4e7ac0daec;
  local_e0 = 0x8929723c31c59039;
  local_d8 = 0xea92ac15de3c3f69;
  local_d0 = 0x828dd2f713f6e8be;
  local_c8 = 0xbb4d607b1c553c6f;
  local_c0 = 0x7dc2d2f3ec43ef5b;
  local_b8 = 0x4daf64150084dc96;
  local_b0 = 0xe1f1361e21c67ab9;
  local_a8 = 0xa4b498c90be95f82;
  local_a0 = 0xb439b94451f266b5;
  local_98 = 0x2380c814a4f0145b;
  local_90 = 0x808581a5b7fb9d7e;
  local_88 = 0x589b2b23881c5633;
  local_80 = 0xbbaa188d8cde35d8;
  local_78 = 0xf6f8fd3aeb6dd0d2;
  local_70 = 0x2feaa6b6ae8530b8;
  local_68 = 0xb30edc56b009e85f;
  local_60 = 0xfbd9e747ffc36c8f;
  local_58 = 0x18194f7045e8f66;
  local_50 = 0xc27b4d434fe8beea;
  local_48 = 0xcb87ceb3;
```

Nothing immediately sticks out to we convert these to strings

```
(¿.h6. [MÓÎ:%Rç.¥í.á.0DIn'áG;..7m©ìNzÀÚì.)r<1Å.9ê.¬.Þ<?i..Ò÷.öè¾»M`{.U<o}ÂÒóìCï[M¯d...Ü.áñ6.!Æz¹¤´.É.é_.´9¹DQòfµ#.È.¤ð.[...
¥·û.~X.+#..V3»ª...Þ5Øöøý:ëmÐÒ/ê¦¶®.0¸³.ÜVº       è_ûÙçGÿÃl...OpEèöl'´Ô4þ.î¬¸|ë2.
```

It's also important to note that the program expects our password as an option instead of runtime input

```
printf("Usage: %s <password>\n",*param_2)
```

The decompiler did a pretty good job of giving us a good starting point for this program, so let's take a look at the code once we rename some of the variables for clarity

This first part of the code iterates through our password uses it as input to modify a constant. This is easy enough to replication in python

```
if (argc == 2) {
  big_number = 2147483647;
  for (j = 0; uVar1 = j, password_length = strlen(argv[1]), uVar1 < password_length; j = j + 1) {
    big_number = big_number + argv[1][j] * j;
  }
```

This next bit declares two arrays, its probably fair to guess that the program is expecting a 6 character password, but we'll see further on if that's true

```
array_1[0] = 123;
array_1[1] = 456;
array_1[2] = 789;
array_1[3] = 987;
array_1[4] = 654;
array_1[5] = 321;
array_2[0] = 92;
array_2[1] = 29;
array_2[2] = 380;
array_2[3] = 2;
array_2[4] = 497;
array_2[5] = 296;
```

Here we have our first significant comparison, it looks to see if the modulous of big number and array_1 is equal to array 2

```
for (i = 0; i < 6; i = i + 1) {
  if (big_number % array_1[i] != array_2[i]) {
    puts("Invalid password!");
    return 0;
  }
}
```

That's easy enough to brute force in python. This next bit is interesting...

So we have two for loops. The first generates a number...

```
uint generate_number(void)

{
  DAT_00104040 = DAT_00104040 * 1103515245 + 12345;
  return DAT_00104040 >> 16 & 32767;
}
```

```
UINT_00104040

    uint        1
```

Then iterates through a constant number in the code...

```
check_number_2 = 2936001189089117O7;
```

and stores it back at check_number_2

once that's done, it iterates through check_number_2 and multiplies it with an increasing count

and finally checks if that number is equal to 210479

the last check is a password length check, which is interesting

```
  int_to_long(big_number);
  for (k = 0; k < 197; k = k + 1) {
    check_number = generate_number();
    *(&check_number_2 + k) = *(&check_number_2 + k) ^ check_number;
  }
  local_2c = 0;
  for (l = 0; l < 197; l = l + 1) {
    local_2c = local_2c + *(&check_number_2 + l) * l;
  }
  local_3c = 2104079;
  if (local_2c != 2104079) {
    puts("Invalid password!");
    return 0;
  }
  password_length = strlen(argv[1]);
  local_40 = (*local_38)(argv[1],password_length);
  if (local_40 == 0) {
    puts("Invalid password!");
  }
```

Finally we get to the good password message

```
  else {
    puts("Good password!");
  }
```

We write a quick python script to brute force a solution to the first check

```
#!/usr/bin/env python3

__author__ = "triboulet"

array_1 = [123,456,789,987,654,321]
array_2 = [92,29,380,2,497,296]
count = 2147483647

while(True):
    if(count % array_1[0] == array_2[0]):
        ## print("worked on 0th value")
        if(count % array_1[1] == array_2[1]):
            print("worked on 1st value", count)
            if(count % array_1[2] == array_2[2]):
                print("worked on 2nd value", count)
                if(count % array_1[3] == array_2[3]):
                    print("worked on 3rd value", count)
                    if(count % array_1[4] == array_2[4]):
                        print("worked on 4th value", count)
                        if(count % array_1[5] == array_2[5]):
                            print("SOLVED:")
                            print(count)
                            quit()

    count += 1
```

```
└─$ python brute_check_1.py
worked on 1st value 2147491901
worked on 2nd value 2147491901
worked on 3rd value 2147491901
worked on 4th value 2147491901
SOLVED:
2147491901
```

We can use some deductive math to ball park some viable key space/length that we can maybe use to brute force a password

Final number we need:
2147491901

=>

2147491901 - 2147483647 (big_number) = 8254

What this means is that the number the loop needs to generate is going to be 8254

so our original equation looped like

$$\sum_{i=0}^{i=len(password)} y * i = 8254$$

Expanding this out a bit…

y*0 + y*1+y*2…+X*y = 8254

If we set y equal to the same value…and we know that typable characters are 33-126 we can further reduce the keyspace

X*33 = 8254 (largest possible factor in front of y)

=> X ≈ 251

solving for the coefficients in this case…
=> 1+2+3+4…+Z = 251
=> 1+2+3…+20+21 ≈ 251
=> max length of 22, but we removed the 0th element so the true count is 23

Doing the same for the maximum case

X*126 = 8254

=> X≈ 66
=> 1+2+3…+Z
=> 1+2+3…+10+11 = 66
=> min length of 11, but we removed the 0th element so the true count is 12

using the below script, we generate some viable answers

```
#!/usr/bin/env python3

__author__ = "triboulet"

def check_viable(pass_list, sum_of_pass): ## manipulates the ones digit of the equation in order to try to get a valid answer
    new_list = pass_list[:]
    out_sum = 0
    sub_num = sum_of_pass - 8254
    if(not(sub_num > 93 or sub_num < 0)):
        new_list[1] -= sub_num
```

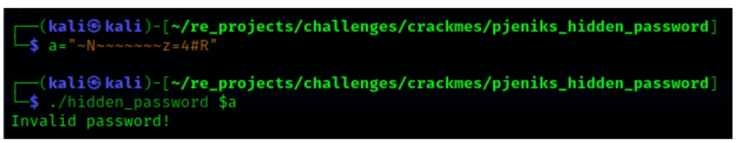
```
        for i in range(0, len(new_list)):
            out_sum += new_list[i]*i
        if(out_sum == 8254):
            key = ''
            print("Viable...")
            print(out_sum)
            for i in new_list:
                key += chr(i)
            print(new_list)
            print(key, end="\n\n")

def click_down_faster(pass_list): ## just trying to iterate through possibilities faster
    if 33 in pass_list:
        ind = pass_list.index(33)
        pass_list[pass_list.index(33)-1] -= 1
        pass_list[ind] = 126
    else:
        pass_list[-1] -= 93
    return pass_list

def click_down(pass_list):
    if 33 in pass_list:
        ind = pass_list.index(33)
        pass_list[pass_list.index(33)-1] -= 1
        pass_list[ind] = 126
    else:
        pass_list[-1] -= 1
    return pass_list

def reset(pass_list):
    iter_count = len(pass_list) + 1
    pass_list = []
    for i in range(0, iter_count):
        pass_list.append(126)
    # print(len(pass_list))
    return pass_list

sum_of_pass = 0
goal_number = 8254
key = ''
count = 0
pass_chars = [126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126]

## there's definitely a better way to do this, but this is my clunky solution

while(True):
    # print(sum_of_pass)
    # print(pass_chars)
    sum_of_pass = 0
    count += 1
    for i in range(0, len(pass_chars)):
        sum_of_pass += pass_chars[i]*i
    if (sum_of_pass > 9000):
        pass_chars = click_down_faster(pass_chars)
    elif(sum_of_pass > 8254 - (len(pass_chars) * 126)):
        # print("clicking down...")
        if(sum_of_pass < 8380):
            check_viable(pass_chars, sum_of_pass)
```

```
      pass_chars = click_down(pass_chars)
   elif(sum_of_pass < 8254 - (len(pass_chars) * 126)):
      print("Resetting...")
      pass_chars = reset(pass_chars)
      print(sum_of_pass)
      print(len(pass_chars))
      print(pass_chars)
      #quit()
   elif(sum_of_pass == 8254):
      key = ''
      print(pass_chars)
      print("\n\nSolved:")
      for i in pass_chars:
         key += chr(i)
      print(key, end="\n\n")
      quit()
```

We generate the following three test keys
~4~~~~~~~z=4#T

~A~~~~~~~z=4#S

~N~~~~~~~z=4#R



When loading the program in gdb, we have to bypass the ptrace protections and set a break point at
0x55555555519f

in gdb/pwndbg the following addresses are interesting

0x000055555555519f ## main
0x0000555555555349 ## first compare
0x00005555555553a9 ## second compare
0x00005555555553ae ## begin big comparison
0x000055555555544d ## big number comparison with arrays
0x0000555555555542 ## rdx function call
0x0000555555555544 ## rdx function call back in main

Stepping through the program we see that our test password (~A~~~~~~~z=4#S) does make it past the first two checks

```
*RAX  0x760e
 RBX  0xe
 RCX  0x141
 RDX  0x128
 RDI  0x8000203d
 RSI  0x7fffffffdf18 —▸ 0x7fffffffe272 ◂— '/home/kali/re_projects/challenges/crackmes/pjeniks_hidden_password/hidden_password'
 R8   0x0
 R9   0x7ffff7fdc1f0 (_dl_fini) ◂— push   rbp
 R10  0xfffffffffffffb8b
 R11  0x7ffff7f26d80 (__strlen_avx2) ◂— mov    eax, edi
 R12  0x555555555070 ◂— xor    ebp, ebp
 R13  0x0
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdce0 —▸ 0x7fffffffde20 ◂— 0x0
 RSP  0x7fffffffdce0 —▸ 0x7fffffffde20 ◂— 0x0
*RIP  0x555555555198 ◂— pop    rbp
──────────────────────────────────────[ DISASM ]──────────────────────────────────────
   0x55555555517b    add    rax, 0x3039
   0x555555555181    mov    qword ptr [rip + 0x2eb8], rax
   0x555555555188    mov    rax, qword ptr [rip + 0x2eb1]
   0x55555555518f    shr    rax, 0x10
   0x555555555193    and    eax, 0x7fff
 ► 0x555555555198    pop    rbp
   0x555555555199    ret
    ↓
   0x55555555548b    mov    byte ptr [rbp - 0x39], al
   0x55555555548e    mov    eax, dword ptr [rbp - 0x20]
   0x555555555491    cdqe
   0x555555555493    movzx  eax, byte ptr [rbp + rax - 0x100]
```

and the result of the generate_number() function is 0x760e (30222 in decimal)

while stepping through the program, I notice that the generate_number() function is called INSIDE of the loop. That's interesting, and it looks like every_loop is connected to the previous loop

```
*RAX  0x7e8
```

And that makes sense because we expect the following functionality

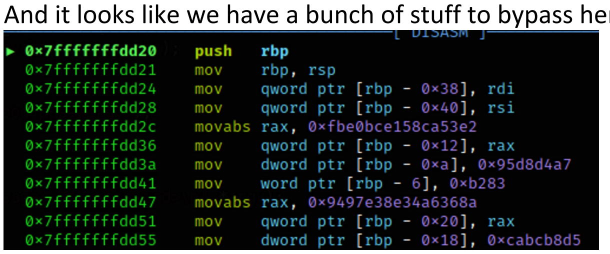
```
uint generate_number(void)

{
  DAT_00104040 = DAT_00104040 * 1103515245 + 12345;
  return DAT_00104040 >> 16 & 32767;
}
```

Looks like the next two loops are integrity checks and do nothing to our input…
so the next check to pass comes here

```
  length_of_pass = strlen(argv[1]);
  local_40 = (*local_38)(argv[1],length_of_pass);
  if (local_40 == 0) {
    puts("Invalid password!");
  }
  else {
    puts("Good password!");
  }
```

What we notice during this execution is that a function address gets passed and called from rdx, and into that function we pass our password and the length of the password (rdi, rsi)

And it looks like we have a bunch of stuff to bypass here…



Dumping the instructions at this address…

pwndbg> x/100i 0x7fffffffdd20

```
0x7fffffffdd20:   push   rbp
0x7fffffffdd21:   mov    rbp,rsp
0x7fffffffdd24:   mov    QWORD PTR [rbp-0x38],rdi load password into rbp-0x38
0x7fffffffdd28:   mov    QWORD PTR [rbp-0x40],rsi load password length into rbp-0x40
0x7fffffffdd2c:   movabs rax,0xfbe0bce158ca53e2
0x7fffffffdd36:   mov    QWORD PTR [rbp-0x12],rax
0x7fffffffdd3a:   mov    DWORD PTR [rbp-0xa],0x95d8d4a7
0x7fffffffdd41:   mov    WORD PTR [rbp-0x6],0xb283
0x7fffffffdd47:   movabs rax,0x9497e38e34a6368a ## move this value into rax
0x7fffffffdd51:   mov    QWORD PTR [rbp-0x20],rax ## put big number in rbp-0x20, this gets referenced later
0x7fffffffdd55:   mov    DWORD PTR [rbp-0x18],0xcabcb8d5
0x7fffffffdd5c:   mov    WORD PTR [rbp-0x14],0x80b7
0x7fffffffdd62:   mov    QWORD PTR [rbp-0x2e],0x0
0x7fffffffdd6a:   mov    DWORD PTR [rbp-0x26],0x0
0x7fffffffdd71:   mov    WORD PTR [rbp-0x22],0x0
0x7fffffffdd77:   cmp    QWORD PTR [rbp-0x40],0xe ## checks to see if length is 14, which it is by accident
0x7fffffffdd7c:   je     0x7fffffffdd85
0x7fffffffdd7e:   mov    eax,0x0
0x7fffffffdd83:   jmp    0x7fffffffdde3
0x7fffffffdd85:   mov    DWORD PTR [rbp-0x4],0x0 ## jump here if length is 14, initialize variable at 0
0x7fffffffdd8c:   jmp    0x7fffffffddd6
0x7fffffffdd8e:   mov    eax,DWORD PTR [rbp-0x4] ## LOOP ----------------------------------------------------------------------
0x7fffffffdd91:   movsxd rdx,eax
0x7fffffffdd94:   mov    rax,QWORD PTR [rbp-0x38] ## load password into rax
0x7fffffffdd98:   add    rax,rdx
0x7fffffffdd9b:   movzx  edx,BYTE PTR [rax] ## load character from password into edx
0x7fffffffdd9e:   mov    eax,DWORD PTR [rbp-0x4]
0x7fffffffdda1:   cdqe
0x7fffffffdda3:   movzx  eax,BYTE PTR [rbp+rax*1-0x12]
0x7fffffffdda8:   xor    edx,eax
0x7fffffffddaa:   mov    eax,DWORD PTR [rbp-0x4] ##LOOP COUNT into EAX
0x7fffffffddad:   cdqe
0x7fffffffddaf:   mov    BYTE PTR [rbp+rax*1-0x2e],dl ## store result of XOR on the stack
0x7fffffffddb3:   mov    eax,DWORD PTR [rbp-0x4]
0x7fffffffddb6:   cdqe   ## treat eax as QWORD
0x7fffffffddb8:   movzx  edx,BYTE PTR [rbp+rax*1-0x2e]
0x7fffffffddbd:   mov    eax,DWORD PTR [rbp-0x4]
0x7fffffffddc0:   cdqe
0x7fffffffddc2:   movzx  eax,BYTE PTR [rbp+rax*1-0x20] ## this is effectively [rbp * 1 - 0x20]
0x7fffffffddc7:   cmp    dl,al ## we want this to be true
0x7fffffffddc9:   je     0x7fffffffddd2 ## we want this jump
0x7fffffffddcb:   mov    eax,0x0
0x7fffffffddd0:   jmp    0x7fffffffdde3
0x7fffffffddd2:   add    DWORD PTR [rbp-0x4],0x1 ##rbp-0x4 is the loop counter
0x7fffffffddd6:   mov    eax,DWORD PTR [rbp-0x4] ## jumps here after loading 0 into eax
0x7fffffffddd9:   cmp    eax,0xd ## LOOP--------------------------------------------------------------------------------------
0x7fffffffdddc:   jbe    0x7fffffffdd8e
0x7fffffffddde:   mov    eax,0x1
0x7fffffffdde3:   pop    rbp
0x7fffffffdde4:   ret
```

So there's actually a few note worthy things happening in the loop

One character from our input is getting XOR'd with some data in memory
xor edx,eax

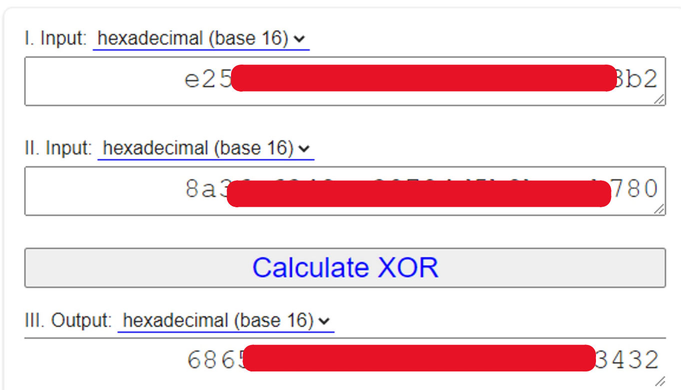
and then this data is getting compared to a different set of data on the stack
cmp dl,al ## we want this to be true

in short

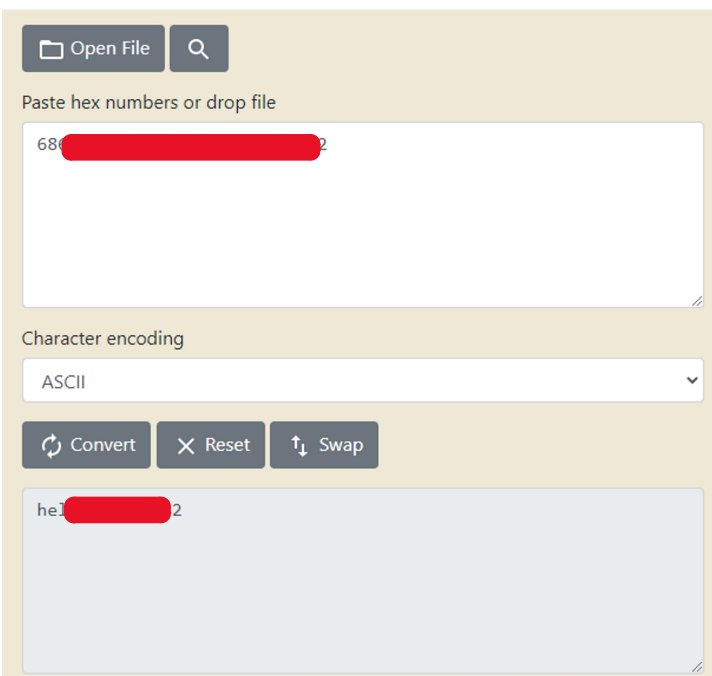
input ^ 0xe2 0x[redacted] 0x95 0x83 0xb2 =

0x8a 0x36 0[redacted]ca 0xb7 0x80 0xe2

We XOR the two hard coded strings together, and we get some interesting hex



Converting that string to ASCII we find the hidden key



We validate it in our program