Francisco Melendez Laureano 801-19-3512 francisco.melendez4@upr.edu

Prof. Rafael A. Arce

CCOM4702 **Lab 03 - Common Structures and Constant Arithmetic**

1. Multiply this

In this exercise we can utilize *objdump -d <program> -M intel* , and see the executable's instructions. Note this one was compiled with **-O0**

We spot a **<pig>** function, it takes one argument, *$rdi* and returns value as well, but first let's see what we read in **<main>**:

```
    11c7:   89 7d fc                mov     DWORD PTR [rbp-0x4],edi
    11ca:   48 89 75 f0             mov     QWORD PTR [rbp-0x10],rsi
    ....
   11de:   48 8b 45 f0              mov     rax,QWORD PTR [rbp-0x10]
    ....
   11e9:   48 89 c7                 mov     rdi,rax         // <- Notice rdi
```

This indicates that we're reading a command line argument (*CLA*), we investigate further to understand that **C** reads them as a sequence of characters.

Back in **<pig>** we notice:

```
    119e:   01 d0                   add     eax,edx
    11a0:   83 e8 30                sub     eax,0x30
```

To put it shortly, the function subtracts 48 from the character which turns it into it's Integer value. This is an 'ATOI' function (atoi, atol, atoll - convert a string to an integer -*Linux Man Page*)

Therefore <*pig*> 's argument is a char sequence. After the call we observe the following:

```
11f1:   89 c2              mov     edx,eax
11f3:   89 d0              mov     eax,edx
11f5:   c1 e0 04           shl     eax,0x4
11f8:   01 d0              add     eax,edx
11fa:   3d 15 0b 02 00     cmp     eax,0x20b15
```

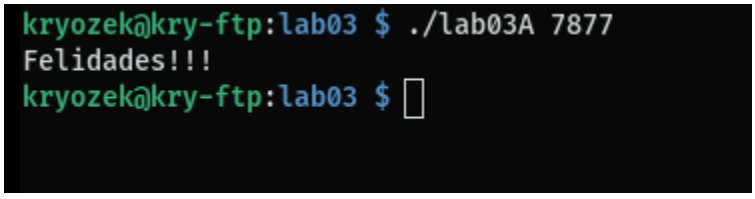
In layman's term, we're multiplying *eax* by *16* (*shl eax,0x4*) , which shifts *eax* to the left 4 times.

Then adding it's value again one time, therefore the final operation is *eax * 17*.

Afterwards we compare it with 133909 *(0x20b15)* .

If we trace the operations inversely we trace back the required number.

$input \frac{133909}{17} = 7877$.



That concludes the first exercise.

2. Multiply that

This exercise was compiled with **-O1** , we see the changes notable in <*pig*> , which changes a few of the ways it turns a character sequence (*string*) to **int**.

Let's skip some of the ASM code in main as it is very similar to the previous and jump to the **calls** and **cmp**'s towards the end.

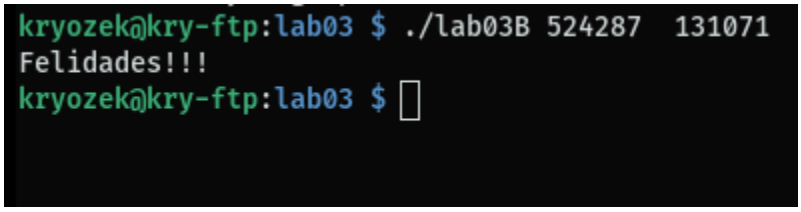
We'll notice that there's *two* <*pig*> calls, this indicates we're inputting two *CLA*'s.

After we compare that neither value are 1 then the following happens:

```
11cc:   48 0f af d8                 imul   rbx,rax
11d0:   48 b8 01 00 f6 ff 0f    movabs rax,0xffff60001
11d7:   00 00 00
11da:   48 39 c3                    cmp      rbx,rax
```

We multiply *rbx * rax* , then compare their product to 68718821377 (*0xffff60001*).

With the help of a divisors calculator, we find: 524287 131071 , these two values' product corresponds to the required number.



3. Mongo cypher

In this exercise we're to input a *CLA* that'll compare to '**ThePassword**' after performing a series of operations on it. Let's skip over to the main logic…

The program loops over each character of our input, the following seems to be the cypher's logic.

```
10bb:   8d 04 d5 00 00 00 00    lea  eax,[rdx*8+0x0]
10c2:   29 d0                   sub      eax,edx
10c4:   48 63 d0                movsxd rdx,eax
10c7:   89 c7                   mov      edi,eax
10c9:   48 69 d2 1f 85 eb 51    imul   rdx,rdx,0x51eb851f
10d0:   c1 ff 1f                sar      edi,0x1f
10d3:   48 c1 fa 23             sar      rdx,0x23
10d7:   29 fa                   sub      edx,edi
10d9:   8d 14 92                lea      edx,[rdx+rdx*4]
10dc:   8d 14 92                lea      edx,[rdx+rdx*4]
10df:   29 d0                   sub      eax,edx
10e1:   83 c0 61                add      eax,0x61
10e4:   38 01                   cmp      BYTE PTR [rcx],al
```

Over at : lea eax,[rdx*8+0x0]

sub eax,edx

We multiply by 8 then subtract 1,

*eax = edx * 7*

Then some divisions and multiplications:

```
10d0:   c1 ff 1f                sar     edi,0x1f  // 2^31
10d3:   48 c1 fa 23             sar rdx,0x23 // 2^35
10d7:   29 fa                   sub     edx,edi  // edx - edi
10d9:   8d 14 92                lea     edx,[rdx+rdx*4]
10dc:   8d 14 92                lea     edx,[rdx+rdx*4] // Done two times... (10)
10df:   29 d0                   sub     eax,edx
10e1:   83 c0 61                add     eax,0x61      // 'a' -> 97 decimal
```

We'll simplify some operations $2^{35} / 2^{31} => 2^4 = 16$ [0-15] For the shifting actions, then in the multiplication (*lea*) we add:

5 + 5 = 10 . We may be able to abbreviate division by 25 (15 from the shifts and +10 from the add)..

Then +97 in add *<eax,0x61>*

At first glance I thought that the Cypher is: C = (((chr / 7) * 25) / 25) - 97.

After some revelations by a higher being, if we turn this into a modulus operation as a compiler would synthetize it, we've the following:

$$C = A - B\,(A/B) \;\; -> \;\; C = A \,\%\, B$$

Therefore;

$$A = c * 7,\; B = 25 \;\; -> \;\; (c * 7) \;\%\; 25$$

Using an aid program to automate the transformation process:

```
password = "ThePassword"

output = ""

for c in password:

        result = ((ord(c) * 7 ) % 25 + 97)

        output += chr(int(result))

print(output)
```

Will net us with this great sight: