Lab1A Writeup

By: Frank Cerny

7/2/18

First things first we head into the directory and run lab1A

```
lab1A@warzone:/levels/lab01$ ./lab1A
.---------------------------.
|---------  RPISEC  --------|
|+ SECURE LOGIN SYS v. 3.0 +|
|---------------------------|
|~- Enter your Username:  ~-|
'---------------------------'
helloWorl
.---------------------------.
| !! NEW ACCOUNT DETECTED !!|
|---------------------------|
|~- Input your serial:    ~-|
'---------------------------'
84848913841834081
lab1A@warzone:/levels/lab01$
```

We are prompted to enter a username and serial number, one after the other. I tested out some long unconventional strings just to make sure there was buffer overflow protection and possibly get an easy win.

```
lab1A@warzone:/levels/lab01$  ./lab1A
.---------------------------.
|---------  RPISEC  --------|
|+ SECURE LOGIN SYS v. 3.0 +|
|---------------------------|
|~- Enter your Username:  ~-|
'---------------------------'
478392049p3829r4rjfdfsafsfjklsjdafjklj;jrio2u9934085ujfkldjsfja93r90ufjsjfajj
39u309ajafjdsjafj
.---------------------------.
| !! NEW ACCOUNT DETECTED !!|
|---------------------------|
|~- Input your serial:    ~-|
'---------------------------'
```

For some reason when I enter an extremely long string, it does not allow me to enter the serial number. It is also interesting that there is no indication of a wrong combination, except that the elevated shell prompt we are searching for does not appear.

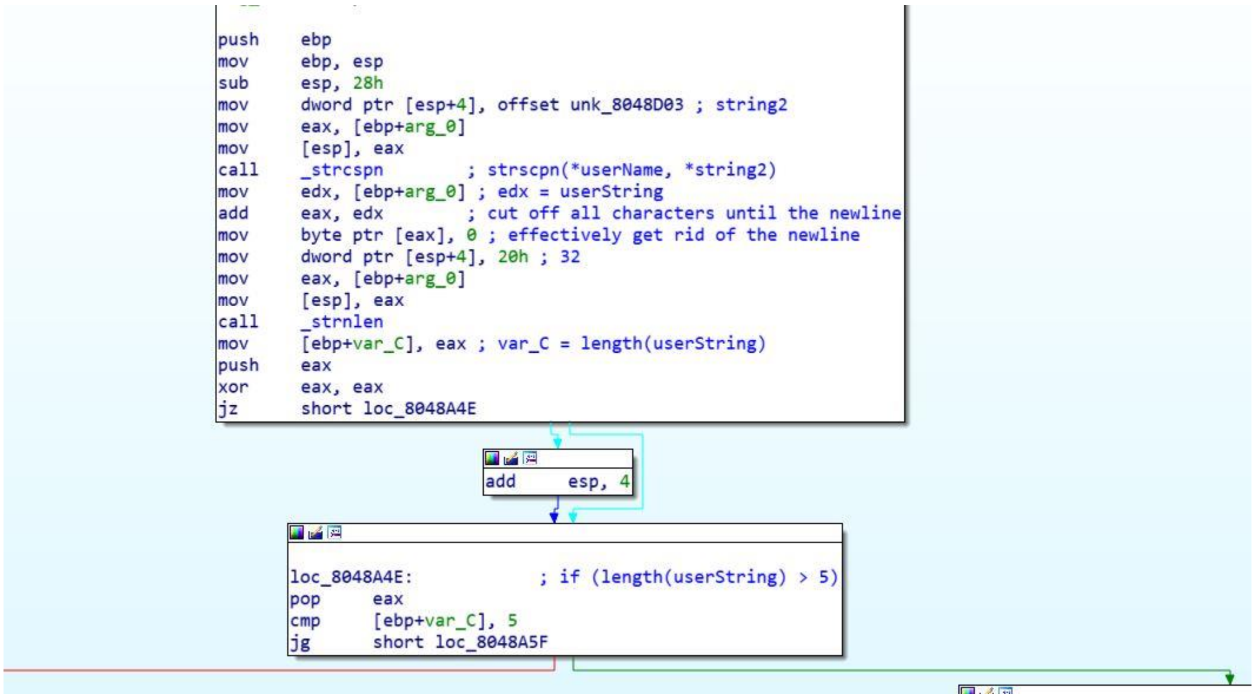
We head into IDA to get a better look of what is going on:

```
call    _fgets          ; fgets(*str, 32, fileStream)
mov     dword ptr [esp], offset asc_8048D73 ; ".--------------------------."
call    _puts
mov     dword ptr [esp], offset aNewAccountDete ; "| !! NEW ACCOUNT DETECTED !!|"
call    _puts
mov     dword ptr [esp], offset asc_8048DCD ; "|--------------------------|"
call    _puts
mov     dword ptr [esp], offset aInputYourSeria ; "|~- Input your serial:    ~-|"
call    _puts
mov     dword ptr [esp], offset asc_8048E09 ; "'--------------------------'"
call    _puts
lea     eax, [esp+18h]  ; &var
mov     [esp+4], eax
mov     dword ptr [esp], offset unk_8048D00 ; %u (unsigned int)
call    ___isoc99_scanf ; scanf("%u", &var)
mov     eax, [esp+18h]
mov     [esp+4], eax
lea     eax, [esp+1Ch]
mov     [esp], eax
call    auth            ; auth(*userName, *serialNumber)
test    eax, eax
jnz     short loc_8048C55 ; if eax != 0
```

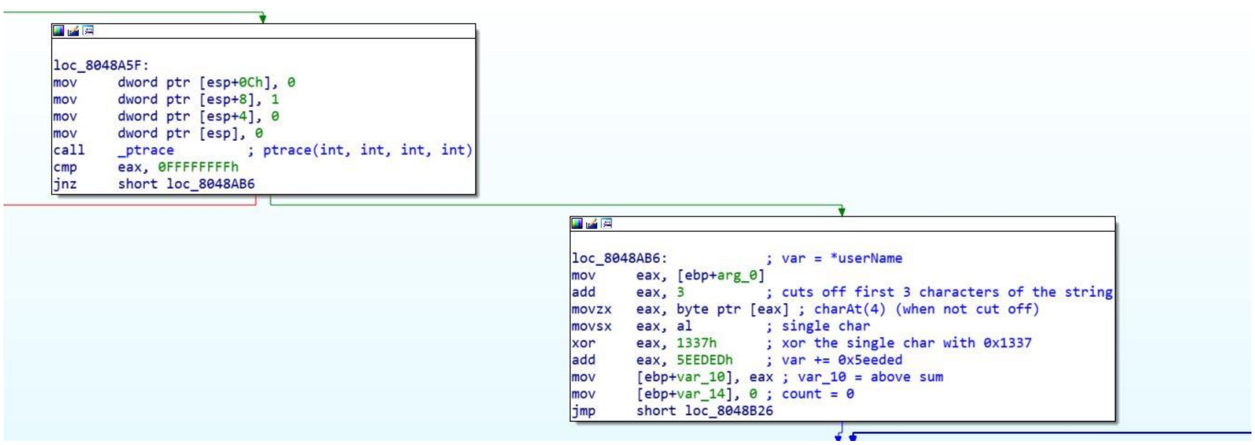
This is the main chunk of the main method. It looks like they use fgets() to grab the user name and then scanf() to get the serial number, which explains why buffer overflows do not work here. Both these values are passed to the method auth(). The jump at the end of the main method leads here:

Test eax, eax returns eax xored with itself, so if we want to get the elevated shell, we want auth() to return a value of 0 (Since 0 xored with 0, is 0).

So let's take a look at auth():



There are a few things going on here. In the beginning they call strscpn() on the passed in user name and "\n". Strscpn returns an integer value which corresponds to the length of the substring of the first parameter in which there are no characters from the second parameter. IE. we want to separate the new line from the user name string. Then they save the length of the user name string, and continue only if the string is longer than 5 characters.



If the string is longer than 5 characters, the program calls ptrace(), which as far as I can tell makes sure that the user is not running the program with a debugger. If they are, auth() returns a 1, and the program ends. (I only found this fact out by using gdb, more on this later). The right block chops the first 3 characters off the username, takes the new first character and runs it through a formula like this:

$$sum = (charAt(0)\ XOR\ 0x1337) + 0x5EEDED$$

With count initialized to 0, it seems we are headed into a while loop:

```
jmp     short loc_8048B26
```

```
loc_8048B26:
mov     eax, [ebp+var_14]
cmp     eax, [ebp+var_C]
jl      short loc_8048AD8 ; While (count < length(userString))
```

```
loc_8048AD8:            ; edx = count
mov     edx, [ebp+var_14]
mov     eax, [ebp+arg_0] ; eax = *userString
add     eax, edx        ; get the next char in the string
movzx   eax, byte ptr [eax] ; charAt(0)
cmp     al, 1Fh
jg      short loc_8048AEE ; if char value > 0x1F (31)
```

```
mov     eax, [ebp+arg_4]
cmp     eax, [ebp+var_10] ; all the summations of xored sums must equal the serial number
jz      short loc_8048B3D
```

```
mov     eax, 1
jmp     short locret_8048B42
```

```
loc_8048B3D:
```

```
loc_8048AEE:            ; edx = count
```

The program loops through the entire user name string, and the left block makes sure that every character has a ascii hex value > 31 (for what reason I do not know). Once the loop is over it seems that overall sum (which we will get to) must equal the serial number we entered, which makes a lot of sense.

Let's look at the body of the loop:

```
loc_8048AEE:            ; edx = count
mov     edx, [ebp+var_14]
mov     eax, [ebp+arg_0] ; eax = userString
add     eax, edx
movzx   eax, byte ptr [eax] ; charAt(0)
movsx   eax, al
xor     eax, [ebp+var_10] ; xor charAt(0) with weird sum
mov     ecx, eax        ; ecx = xored sum
mov     edx, 88233B2Bh  ; 2284010283
mov     eax, ecx        ; eax = xored sum
mul     edx             ; eax = eax * 2284010283
mov     eax, ecx        ; eax = above
sub     eax, edx        ; ecx * 2284010283 - 2284010283
shr     eax, 1          ; divide by 2
add     eax, edx        ; add 2284010283 back
shr     eax, 0Ah        ; divide by 2^10
imul    eax, 539h       ; multiply by 0x539 (1337)
sub     ecx, eax        ; ecx = ecx - multiplied sum
mov     eax, ecx        ; eax = multiplied sum subtracted from ecx
add     [ebp+var_10], eax ; add all the crap together
add     [ebp+var_14], 1 ; count++
```

At first glance I was unable to understand what was going on here, even with some of the math worked out. I will not go through the entire formula, because it is mostly self-explanatory, just a little hard to follow since the values are swapped a few times. The hardest part of this formula was determining what the opcode mul does, and what values are added and subtracted from eax in the below few lines:

```
mul     edx             ; eax = eax * 2284010283
mov     eax, ecx        ; eax = above
sub     eax, edx
shr     eax, 1          ; divide by 2
add     eax, edx        ; add some number back
```

The mul opcode in x86 does as follows:

mul returns a 32bit value, inside the pair eax:reg

Operand Size	Source 1	Source 2	Destination
Doubleword	EAX	EDX	EDX: EAX

Eax is implicit, in that it is always used. Mul takes one parameter, which is the one that is multiplied with eax. Thus, in a 32*32 multiplication, the high 32 bits are stored in edx, and the low 32 bits are stored in eax. This can be confirmed in gdb:

```
[----------------------------------registers-----------------------------------]
EAX: 0x5f015c
EBX: 0xb7fcd000 --> 0x1a9da8
ECX: 0x5f015c
EDX: 0x88233b2b
ESI: 0x0
EDI: 0x0
EBP: 0xbffff6a8 --> 0xbffff6f8 --> 0x0
ESP: 0xbffff680 --> 0x0
EIP: 0x8048b08 (<auth+249>:     mul    edx)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048aff <auth+240>:        mov    ecx,eax
   0x8048b01 <auth+242>:        mov    edx,0x88233b2b
   0x8048b06 <auth+247>:        mov    eax,ecx
=> 0x8048b08 <auth+249>:        mul    edx
   0x8048b0a <auth+251>:        mov    eax,ecx
   0x8048b0c <auth+253>:        sub    eax,edx
   0x8048b0e <auth+255>:        shr    eax,1
   0x8048b10 <auth+257>:        add    eax,edx
[------------------------------------stack-------------------------------------]
```

Here we have:

eax = 0x5f015c

edx = 0x88233b2b

Calculator Results of eax * edx = 3285cc04d96e74

If we split the 64-bit binary of that number into the high and low 32 bits we have:

High 32 Bits: 3285cc (edx)

Low 32 Bits: 4d96e74 (the second half of the calculator results) (eax)

So, let's check this in gdb:

```
Program received signal SIGALRM, Alarm clock.
[----------------------------------registers-----------------------------------]
EAX: 0x4d96e74
EBX: 0xb7fcd000 --> 0x1a9da8
ECX: 0x5f015c
EDX: 0x3285cc
ESI: 0x0
EDI: 0x0
EBP: 0xbffff6a8 --> 0xbffff6f8 --> 0x0
ESP: 0xbffff680 --> 0x0
EIP: 0x8048b0a (<auth+251>:     mov    eax,ecx)
EFLAGS: 0xa07 (CARRY PARITY adjust zero sign trap INTERRUPT direction OVERFLOW)
[-------------------------------------code-------------------------------------]
```

The math works out. Mul was confusing at first, but running through it in the debugger and doing it by hand helped me out. Now that we have the formula down, I created a python script that spits out a serial number given a user name and an original sum.

```
def getOriginalXor(userName):
    char = userName[3]
    xor = ord(char) ^ 0x1337
    xor = xor + 0x5EEDED
    return xor

def loop(userName, originalSum):
    orgSum = originalSum
    for i in range(len(userName)):
        # var1 = eax, var2 = edx, follows the psuedocode
        char = userName[i]
        var1 = ord(char) ^ orgSum
        var2 = var1
        temp1 = 0x88233b2b
        var1 = var2
        var1 *= temp1
        high32Bits = getHigh32BitsFrom64Bits(var1)
        var1 = var2
        var1 = var1 - high32Bits
        # use bitshifting to avoid float division, decimal places cause overestimation
        var1 = var1 >> 1
        var1 = var1 + high32Bits
        var1 = var1 >> 10
        var1 = var1 * 0x539
        var2 = var2 - var1
        var1 = var2
        orgSum = orgSum + int(var1)
    return orgSum
```

While the variable names are not great, it follows the formula in the IDA screenshot above perfectly. It is important to use integer division and not float division. Using float division can cause decimal places from the division, which can cause overestimation when multiplied by 0x539.

To get the high 32 bits from the mul opcode, we shifted down the 64-bit value to 32 bits (keeping the high 32 bits), then returned that value. Python code that does just that:

```
def getHigh32BitsFrom64Bits(number):
    number = number >> 32
    return number
```

Simple, yet effective.

If we run this script using a main method driver here:

```
def main():
    userName = "helper"
    orgSum = getOriginalXor(userName)
    serial = loop(userName, orgSum)
    print('Serial =  %d ' % (serial))
    return serial
```

Then we run this code in IDLE:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\Frank\Documents\MBE_release\levels\lab01\Work\RPILab1A.py
m
>>> a
>>> main()
Serial =  6230774
6230774
>>> |
```

Sweet, so we have the serial number we need to use, let's try this combo on the program:

```
lab1A@warzone:/levels/lab01$ ./lab1A
.---------------------------.
|---------  RPISEC  --------|
|+ SECURE LOGIN SYS v. 3.0 +|
|---------------------------|
|~- Enter your Username:  ~-|
'---------------------------'
helper
.---------------------------.
| !! NEW ACCOUNT DETECTED !!|
|---------------------------|
|~- Input your serial:    ~-|
'---------------------------'
6230774
Authenticated!
$ whoami
lab1end
$ cat /home/lab1end/.pass
1uCKy_Gue55
$
```

And there we have it, while a great solution, there exists a much quicker approach.

What if we just wanted to use gdb, and never touch IDA? Turns out this is the much easier way, and takes about $1/10^{th}$ the time. Run gdb, enter some input and step into the auth() function. All seems well until we step over ptrace() and it returns us from the program before ever hitting the loop:

```
EAX: 0xffffffff
EBX: 0xb7fcd000 --> 0x1a9da8
ECX: 0xb7e22940 (0xb7e22940)
EDX: 0xffffffbc
ESI: 0x0
EDI: 0x0
EBP: 0xbffff6a8 --> 0xbffff6f8 --> 0x0
ESP: 0xbffff680 --> 0x0
EIP: 0x8048a86 (<auth+119>:     jne    0x8048ab6 <auth+167>)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048a77 <auth+104>:        mov    DWORD PTR [esp],0x0
   0x8048a7e <auth+111>:        call   0x8048870 <ptrace@plt>
   0x8048a83 <auth+116>:        cmp    eax,0xffffffff
=> 0x8048a86 <auth+119>:        jne    0x8048ab6 <auth+167>
   0x8048a88 <auth+121>:        mov    DWORD PTR [esp],0x8048d08
   0x8048a8f <auth+128>:        call   0x8048810 <puts@plt>
   0x8048a94 <auth+133>:        mov    DWORD PTR [esp],0x8048d2c
   0x8048a9b <auth+140>:        call   0x8048810 <puts@plt>
                                                             JUMP is NOT taken
[------------------------------------stack-------------------------------------]
```

For some reason, EAX = 0xffffffff, and thus the jump is not taken and the program ends. I scratched my head at this until I realized it was because I was debugging the code that caused this error. A simple workaround it to set EAX to be anything but 0xffffffff. I set it to 0x5 for example:

```
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048a83 in auth ()
gdb-peda$ set $eax = 0x5
gdb-peda$ print $eax
$1 = 0x5
gdb-peda$
```

In this case then, the program will eventually get to the loop and do what we want. After going through the loop and verifying that our python script was correct, we reach the ending condition which checks our serial number against some encrypted sum:

```
EIP: 0x8048b36 (<auth+295>:     mov    eax,0x1)
EFLAGS: 0x293 (CARRY parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048b2e <auth+287>:        mov    eax,DWORD PTR [ebp+0xc]
   0x8048b31 <auth+290>:        cmp    eax,DWORD PTR [ebp-0x10]
   0x8048b34 <auth+293>:        je     0x8048b3d <auth+302>
=> 0x8048b36 <auth+295>:        mov    eax,0x1
   0x8048b3b <auth+300>:        jmp    0x8048b42 <auth+307>
   0x8048b3d <auth+302>:        mov    eax,0x0
   0x8048b42 <auth+307>:        leave
   0x8048b43 <auth+308>:        ret
[------------------------------------stack-------------------------------------]
0000| 0xbffff680 --> 0x0
0004| 0xbffff684 --> 0x0
```

Unfortunately, our serial number was not matching ☹. How can we ever figure it out without the python script? Simple. Line 0x8048b31 compares our serial number to some unknown total sum. What if we could just print this unknown value we are comparing to and save it for the next run of the program?

```
[------------------------------------code------------------------------------]
   0x8048b2e <auth+287>:        mov    eax,DWORD PTR [ebp+0xc]
   0x8048b31 <auth+290>:        cmp    eax,DWORD PTR [ebp-0x10]
   0x8048b34 <auth+293>:        je     0x8048b3d <auth+302>
=> 0x8048b36 <auth+295>:        mov    eax,0x1
   0x8048b3b <auth+300>:        jmp    0x8048b42 <auth+307>
   0x8048b3d <auth+302>:        mov    eax,0x0
   0x8048b42 <auth+307>:        leave
   0x8048b43 <auth+308>:        ret
[------------------------------------stack-----------------------------------]
0000| 0xbffff680 --> 0x0
0004| 0xbffff684 --> 0x0
0008| 0xbffff688 --> 0x1
0012| 0xbffff68c --> 0x0
0016| 0xbffff690 --> 0xbffff6f8 --> 0x0
0020| 0xbffff694 --> 0x6
0024| 0xbffff698 --> 0x5f12f6
0028| 0xbffff69c --> 0x6
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048b36 in auth ()
gdb-peda$ x ($ebp-0x10)
0xbffff698:     0x005f12f6
gdb-peda$
```

So, all we did was print the value in memory at [ebp-0x10]. This is the mystery sum that the cryptic loop has been building the entire time! 0x5f12f2 = 6230774 (look familiar 😉), let's give it a try:

```
lab1A@warzone:/levels/lab01$ ./lab1A
.---------------------------.
|---------  RPISEC  --------|
|+ SECURE LOGIN SYS v. 3.0 +|
|---------------------------|
|~- Enter your Username:  ~-|
'---------------------------'
helper
.---------------------------.
| !! NEW ACCOUNT DETECTED !!|
|---------------------------|
|~- Input your serial:    ~-|
'---------------------------'
6230774
Authenticated!
$ whoami
lab1end
$ cat /home/lab1end/.pass
1uCKy_Gue55
$
```

The password for lab1End is **luCKy_Gue55**

A very interesting problem that really should have only taken 30 mins had I known what I was doing (that's the fun part though). Of course, this is one of infinite combinations of user name and serial numbers, all leading to the same password. Still, it was good to write a python script and gain more experience using IDA and gdb. If I learned one thing it is to think about all the tools I have instead of hopping onto the one I use the most, especially if a simpler solution exists. This was my first write up ever, and I hope to produce one for each of the lab Part A's in the future. Onto the next one.