Joshua Spisak

CS 449

Professor Khattab

March 18, 2019

**Project 2 Report**

**1. Program 1**

**1.1 Procedure**

The first thing I attempted was printing the program with strings. Since the executable was compiled with debug this produced a lot of noise that I could not interpret to figure out the password. So I went to GDB and put a breakpoint at main.

I immediately notices a cmpsb command so I put a break point there. I printed the registers that were being compared and noticed the first was the input I put in, the second was the string ‘CpBgjMgfAWVVIdOKuRlZ’. I disabled my breakpoints and ran again entering this string and found it was the password.

1.2 Password

The password I found was simply ‘CpBgjMgfAWVVIdOKuRlZ’

1.3 Post-Mortem

This was much simpler to debug since there were no obvious loops are transforms to the input string.

If I ran a grep command on the output of strings I easily found the password string which was obviously plain-coded in the file.

**2. Program 2**

**2.1 Procedure**

I started this time in gdb, when I disassembled the main function I realized that it was calling a function called ‘d’, so I set a breakpoint at that function and continued. This led to c which appeared to be a function that removed the \n at the end of the string and replaced it with a 0. In this function is s which seems to return the string length in eax.

Reading further it looks like there are two tests at the end that must be passed for success. The latter is a call to s comparing to 0x6, so it looks like the string must be of length greater than 6. Then there is a call to r which returns something to eax… that must be non-zero to pass at a glance.

Setting a breakpoint in r and calling there. Looks like it does some weird register stuff but still ends up with a pointer to the entered string in ebx (was similar in d). 0x080484b1 seems to load the first character into dl… if that is zero then it will jump to the end and set eax to 1.. this will pass the first test but fail the second because the first character is 0 which will terminate the string. If it’s non-zero It compares it to the end character. If they are equal then they do weird testing stuff. If they are not equal it also seems to jump down to the end and set eax to 0 then exit. So let’s see what that weird testing is.

Weird testing is incrementing the char pointer to the next char in the string then loading that into dl and comparing it to 0. If it is 0 then it sets eax to 1. If is not 0 then it loops again and decrements eax (originally last character of the string). If that character does not match the first character of the shortened string then it will set eax to 0 and exit.

Once I started tinkering with it I realized that eax was still only a pointer to the string. So there was a last char pointer and a first char pointer and it continued until it hits a 0 char. So I made a string that had the nth char == (k – n)th char and was at least 6 chars long and got this: “gfeddefg” which was a successful password.

**2.2 Password**

“gfeddefg” or “kjihhijk” pretty much any ascending-descending ascii code palindrome with length > 6 chars

**2.3 Post-Mortem**

There were a lot more complex function calls here and non-helpful names. Writing out the program in pseudo-code helps. The loops were somewhat confusing but were also fun to follow through.

**3 Program 3**

**3.1 Procedure**

I originally struggled with getting an entry point in the code. But after referencing the object dump I eventually found the entry point at 0x080484ee (originally referenced on line 0x8048387). This called 0x80484da which calls 0x80484cd which calls 0x8048424. This function seems to load a string of length 15 into memory char by char.

After loading the string to -0x21($ebp) in a loop it then basically does logic on the values. I outlined it in a table below as I read through:

For each value in char array {

Current\_value -> eax

If eax==0x39 (9) ->

addl $0x1,-0x10(%ebp) // give points it contains those values

else if eax >0x39 (9) ->

if eax == 0x63 (c) || eax == 0x73 (s) ->

addl $0x1,-0x10(%ebp) // give points

else if eax == 0x30 (0) || eax == 0x34 (4)->

addl $0x1,-0x10(%ebp) // give points

addl $0x1,-0xc(%ebp) (go to next value)

}

It essentially tabulates a score for the string based on the characters. For the specified characters the score stored in -0x10(%ebp) is incremented. Shortly thereafter at line 0x804849f it compares that score to 1 and if it equals one then it seems to present a success message (loaded from 0x80485c4 on line 0x80484a5).

After this I took a stab and entered 9hhhhhhhhhhhhhh and voila! A success message.

**3.2 Password**

Any message with one (and only one) occurrence of the following characters should successfully pass:

9, s, 4, 0, c

Tested Passwords:

“9hhhhhhhhhhhhhh”

“hhhhhhhhhhhhhhs”

“4dkjdjdlkjfaklj”

“dljdjlckki87837”

“*087583728231561”*

Failed passwords to verify:

“00ksjdjakskdjfj”

“hhhhhhhhhhhhhhh”

**3.3 Post-Mortem**

I tried solving this one simply by reading the assembly with minimal testing of the code itself in gdb. This worked for the most part and was also convenient considering how annoying it was to perform disas in gdb without frames. Reading through the code and setting intelligent breakpoints to check flow and register/memory state worked very well.

This was also interesting because the first break I did was in the loading code. This was obviously architecture differently from the static linked files we used before. There were also landmarks of PIC code within the linking code with calls to update reference tables. I walked through the code with gdb, it was very expansive.