Project 2: What's the password?

Christian Boni (cjb90@pitt.edu)
8/19/2014

1. Part 1

1.1 Procedure

First I started off by using the *mystrings* program created in the first section of the lab with the 1st executable given. I found that this file contained a large amount of text so it I thought it might be reasonable that the password might be stored as some value in the data segment or string table of the address space. Furthermore, to find where that section is I searched the output for the string "Sorry! Not correct!". Strangely enough in the output above that string was another string with a plethora of letters "DAFSrAYMIHUhuXjYiBsnuoWuhlzrTo". Finally I re-ran the program trying this string as the input and found that it was in fact the correct password.

1.2 Solution

The password to part 1 is "DAFSrAYMIHUhuXjYiBsnuoWuhlzrTo".

2. Part 2

2.1 Procedure

First I started off by using the *mystrings* program created in the first section of the lab with the 2nd executable given. On the contrary, I found that this file contained a small amount of text and after quickly scanning it I found that the password was probably not stored in the address space. I then attempted to debug the program by using *gdb*. I started by placing a breakpoint in the function main and running the program and disassembling it when it reached the breakpoint. Through the assembly code I found that main was now calling a function named "d" so I also placed a breakpoint at "d" and continued the program. By disassembling again at the breakpoint I found that "d" was calling a few library functions: fgets, printf, and puts. It was also calling a few other non-library functions: "c", "r", and "s". I placed a breakpoint at each of these non-library functions being called and continued the program. The program paused because it required input so I input the string "hello". The program reached the breakpoint at "c" and after disassembling it; I found that it was calling the function "s". I then placed a breakpoint at "s", continued, and disassembled again. After reading over and tracing the assembly code in "s" I found that it is similar to the strlen function and it is returning the length of the word in the %eax register. Now that I understood what the "s" function did, I looked back at the assembly code in the "c" function. I found that the instruction:

cmpb $0xa,-0x1(%ebx,%eax,1)

was a check to see if the last character in the input string is equal to ASCII code 0xA which is the new line ('\n') symbol. Tracing the code some more shows that if this comparison is true, the new line character is replaced with a null terminator (NUL).

Now it was clear that the “c” function removes the new line character in the input string. Heading back to the function “d” showed the next non-library function being called was “r”. Continuing the program lead to the breakpoint at function “r”. I found a few instructions of importance in “r”:

1. movzbl (%ebx),%edx
2. lea -0x1(%esi,%eax,1),%eax
3. cmp (%eax),%dl

I can see that the first instruction is loading what is in the first byte of the input string into register %edx. Now the second instruction is loading what is in the last byte of the input string into %eax and the final instruction is comparing the two. In my example I found that %eax contained the letter “o” and %dl contained the number 0x68, which is ASCII code for letter “h”. While further looking at the assembly code I found that these comparisons were placed in a loop and realized that this function “r” checks to see if the input string is symmetric. Now knowing what the function “r” performs I continued the program and found that the input string “hello” was incorrect. I then re-ran the program using a new input string “racecar” since this word was symmetric and also a palindrome I figured that this was the correct password. However, I found that this string was still not correct. Looking at the assembly code in “d” after the function “r” was called I found another instruction of importance:

cmp $0xd,%eax

The register %eax contains the length of the input string since this instruction was placed after the “s” function was called. This is where I found that the input string’s length must be greater than 13 characters long. Finally I re-ran the program trying the string “aaaaaaaaaaaaa” as the input and found that it was in fact the correct password. Furthermore, I also tried the string “aaaaaaabaaaaaaa” and found that this password was also correct

2.2 Solution

The password to part 2 is any symmetric string that is greater than 13 characters and less than 100 characters long. For clarity here is my definition of a symmetric string. Assume the input string is a char array A. The character (any ASCII printable character is valid) at A[i] must equal the character at A[(length-1) - i].

3. Part 3

3.1 Procedure

First I started off by using the *mystrings* program again, with the 3rd executable given. Similar to Part 2 I found that this file contained a small amount of text and after quickly scanning it I found that the password was probably not stored in the address space. I then attempted to debug the program by using *gdb*. I placed a breakpoint in main, ran the program, and tried to disassemble it whenever it reached the breakpoint. However, I found that the function main was not defined.

Next I chose to use another useful tool given to us on the project website called *objdump*. I used *objdump* with the switch "S" (to display the source code with disassembly) and the name of the 3rd executable as the parameters. Next I took all the disassembled code that was output and copied into a text editor program to further analyze it. After further analysis of the assembly code I came to the conclusion that this executable was dynamically loaded since the function main was listed as a library function <__libc_start_main@plt>. Now not knowing where to place the breakpoint in the program, I used the trick described in the book and ran the program and pressed "Control + C" and entered "backtrace". I then chose the first address in the backtrace that seemed like a reasonable place to insert a breakpoint and found this address (0x08048473) in the assembly code I had in the text editor. Now I finally had a place to start tracing and analyzing the code. Next, I quickly found an instruction of importance:

cmpl $0x9,-0xc(%ebp)

Here is where I discovered that the input string's length must be at least 9 characters. Using my knowledge that to unlock the program the printf function must be called; I found the address where it was called (0x80484ea) and backtracked from that address while tracing and analyzing the code. As a result I found a few more instructions of importance:

sub $0x30,%eax
cmp $0x8,%eax
ja 80484cb <tolower@plt+0x143>

After some more tracing of the the code I found that this was encapsulated inside a loop which checked each character in the input string and each pass through this loop stored the current character in the register %eax. I realized that for this jump to be skipped the current character must be between hexadecimal numbers 0x30 and 0x38, which in ASCII is the decimal numbers 0 through 8. The next block of assembly code also gave me a little bit more information about the password:

mov $0x1,%edx
mov %edx,%ebx
mov %eax,%ecx
shl %cl,%ebx
mov %ebx,%eax
and $0x155,%eax
test %eax,%eax
je 80484cb <tolower@plt+0x143>
addl $0x1,-0x10(%ebp)

In short, this block of assembly code takes the number 0x1 and performs a left shift on it by the decimal number of the current character (Which can only be numbers 0-8). Lastly it performs an "and" with the left shifted number and the hexadecimal number 0x155. I found that 0x155 in binary is "000101010101" and 0x1 in binary is "000000000001". Now stopping for a second and looking further down the assembly code lies the instructions:

cmpl $0x2,-0x10(%ebp)
jne 80484f1 <tolower@plt+0x169>

This compare instruction must evaluate to true for the jump to be skipped and the printf function to be called. This means that -0x10(%ebp) must be equal to 2. Now looking back at the previous block of code shows that each time the "and" of 0x155 and the left shifted number is not equal to 0 then -0x10(%ebp) is incremented by 1. I found that 0x155 in binary is "000101010101" and 0x1 in binary is "000000000001". By looking at the binary code we find that 0x155 "anded" with 0x1 left shifted by an even number from 0 to 8 will cause this increment to occur. Therefore, as a result I ran the program, and entered the string "02abcdefg" and found that this in fact was a correct password.

3.2 Solution

The password to part 3 is any string that contains exactly two even numbers from 0 to 8 and is exactly 9 characters long. Additionally the string can also be longer than 9 characters but the 2 even numbers must be within the first 9 characters. This is because the program stops checking the string after the 9th character.