**jkd28\_1**

Attempts

My first attempt was made immediately after recitation on 2/20/17. I created a breakpoint at main and used the 'disas' keyword to examine the x86 for jkd28\_1. From this, the following instruction line appeared suspicious:

0x0804830c <+61>: repz cmpsb %es:(%edi),%ds:(%esi)

Upon further investigation, I learned that it was a comparison of strings. Using the debug and examine tools, I was able to determine that the register $esi contained the string to which input was compared. This yielded the correct result.

Solution

ycSzZqniOdtPSkhYkVbrAwosHgIfg

**jkd28\_2**

Attempts

I first set a breakpoint at main and disassembled the code. This allowed me to see a line that once again looked suspicious:

0x080485c0 <+121>: repnz scas %es:(%edi),%al

I set a breakpoint at this line and continued, entering “test” as my input. I noticed that this instruction repeated a number of times equal to the length of my string. This was interesting to me, but it didn’t really give me what I wanted. So I looked elsewhere. I noticed a strcmp:

0x080485db <+148>: call 0x80483d8 <strcmp@plt>

I set a breakpoint here and continued to reach it. I investigated %ebx and found that it contained “test\_2”. I then found that %esi contained “jkd28\_2) and determined that was the solution string. I concluded that the result would be my username, and the program would append “\_2” to the end. I had found the solution, but I wanted to find the point where the “\_2” was inserted onto my test string. I found the move word instruction that did this:

0x080485c8 <+129>: movw $0x325f,(%eax)

0x325f translates to “2\_” in ASCII, which would make sense to add onto our word, since x86 is Little-Endian. I had found the solution to the problem and how the code edited my input to check against the solution.

Solution

jkd28

**jkd28\_3**

Attempts

At first, I was unsure how to proceed since my usual “disas” command at main was not working. After some referencing and google searches, I found that I could see specific information about the file by entering “info file”. Using this method, I was able to identify the memory range of the .text segment (which I know to contain the assembly code I am looking for) and then enter the range with the “disas” command. This allowed me to view the assembly code for the file.

My first instinct was to search for function calls. I found a few calls to C Library functions like getchar, puts, printf, and tolower via my “mystrings.c” output when using “jkd28\_3” as an argument. This was useful, but not too much. I decided to look for other calls in the assembly. After a few dead-ends and useless breakpoints, I realized I should be looking at what happens immediately after getchar is called.

That was also a dead end. I then realized that the getchar call was actually inside a loop. Upon further investigation, this loop ran 9 times, each time taking a character and incrementing a counter stored in EBX. This meant the program was reading in 9 characters at a time. This was a good breakthrough.

Immediately after that loop terminated, another one started. By thoroughly analyzing each line and keeping track of the register values, I was able to determine that this next section was counting the number of vowels in my string. This was very difficult for me to figure out, and required a ton of testing with a large amount of breakpoints throughout the section of code.

After counting the number of vowels, it compares the count to a constant 3. If the count is equal to 3, the code moves on to print out a success string, else the code prints the incorrect statement.

Solution

An example of a successful string is “sasasarrr” or “uuummmttts” or “ThisShould”. Case does not matter as the code transforms it to lowercase anyway.