Jordan Carr

Jcc117

CS0449

**Project 2 Summary**

Jcc117\_1

Passphrase: MYTkSNyjpeaYtIGwAuUU

The passphrase was hard coded into the program. The user would type in what they thought the password was, it would be compared to the string given above, and the answer to whether it was correct or not would be given. I determined this because the x86 code given was not complicated enough for the string to be based on anything else. It only uses the printf, puts, and chomp functions. I thought that the chomp function generated the passphrase at first, but after a little research I found out chomp functions just take off the newline character in a string. Knowing this, I saw that the two strings (input and the passphrase) were being compared and noticed that every time I ran the program the address of the passphrase wouldn’t change. From this I determined that it must have been hard coded into the program and nothing would alter it after every run. After running the mystrings program on the executable I found the string literals the program holds and found the passphrase next to all of the strings such as “Congratualtions” and “Sorry, not correct.” I knew it couldn’t be based on anything else since it uses no functions for time or anything else that would make it randomly generate. There were also no other strings literals within the code, so this is the only password that will work. After using gdb to look at the code I saw that the value at a set address is used to compare the inputted string to. I inspected the contents the address and it was the password.

Jcc117\_2

Passphrase: First letter of the current day in local time, followed by the first letter of the current month in local time (For example, on a Tuesday in February it would be “TF”)

I noticed that the executable uses functions related to the current time. My first guess was that either the passphrase is generated randomly through use of the clock or a number is randomly generated through the clock and an index is used to select which out of many passwords would be used for that run of the program. Running the mystrings code also cued me in to this since among the string literals is the phrase “Unlocked with passphrase %s”. However, this changed when I discovered some strings embedded within the code. After looking into code more I found the strings, “%A”, “%c” and “%B” were found within the code before the strftime function and sscanf function. This led me to believe that one of them, if not a combination of them, is used to find the passphrase. They correlated to the weekday, date and time, and month respectively when used with the strftime function. I tried using all of them in the password but to no avail. After looking into it further I found that the “%c” tag was actually used in the sscanf function and is meant to read in a single char from a string. The first sscanf does this from the current day of the week in local time, and the second one does the same thing to the current month. I looked at the addresses used to hold the two chars and found they were right next to each other (for me it was 0xffffd15e and 0xffffd15d). This led me to believe that these two chars made the password, and upon testing this theory it turned out to be correct.

Jcc117\_3

Passphrase: any combination of angle brackets <>, parentheses (), curly braces {}, or brackets [] as long as the number of them in the passphrase totals 8 and there are two extra ascii characters located anywhere in the passphrase. Note: you can type in as much as you like in the passphrase but the mentioned required items must be within the first 10 chars read in.

The program makes use of the getchar and tolower functions. From this, I guessed that the code takes in a single char at a time and then uses the tolower function to make the input lower case. I thought that perhaps it does this to make comparing the input to what the actual password is easier. After using the objdump on the file and looking through the text portion of the disassembled code I found that the code will take in 10 char values individually and store them. The code compares the values with the ascii values for <> [] {} and () (each part of the brackets, parentheses, etc. are listed as their own char). I used {[(<aa>)]} as the password and it worked (I used ‘a’ just so it could read in a 10th char). After messing around with it more my first guess is that it’s a good password if the brackets are nested properly. However, I found that any combination of the brackets will work. Upon inspecting the code I found that the code loops and inspects the 10 chars read in. There is a counter that keeps track of how many of those brackets are in the passphrase, and it will only work if there are exactly 8 in it. There can be any combination of the four possible as long as there are 8 of them. That being said, they must be within the first 10 chars read in since anything after the 10th char will be ignored. They can be in any order, and there must be two more ascii values located within the passphrase. Any will do as long as they are not <> {} [] or (). They can be located anywhere within the passphrase as long as the total number of brackets is exactly 8. For example, [][]a[][]b and {{}}()<cc> will work but <aaaa>[]() will not.