Zachary Whitney

Pitt ID: 3320178 zdw9

Project #2: What’s the Password?

Solution 1, filename zdw9\_1

The password is: **ngcbHOljuuruZEQLOwPTSdjXLhiCA**

This executable was by far the simplest of the three to crack. Before doing anything else, I ran the program to see what kinds of output it would give. I just put in random data when prompted and received the message “Sorry! Not correct!” No surprise there.

At this point I ran the strings bash utility on the file and it dumped an awful lot of information out to the console. To help parse it, I piped the output of strings to a text file and used the editor search (CTRL + F) to find the message I’d gotten, figuring it’d lead me to the String table of the file. Lo and behold, nearby were two other helpful strings. First was the success String literal “Congratulations\nUnlocked with passphrase %s\n.” The other was the passphrase for the file stored in plaintext. It looked something a keygen would dispense, so I tried ran the program and tried it. Bingo!

Solution 2, filename zdw9\_2

The passwords are: **any palindrome (same backwards as forwards) 7 characters or longer, ignoring**

**the terminating newline when pressing ENTER**

I think I had the most trouble with this file of the three. When I ran the file, I saw that it operated in much the same way as the first. It took a passphrase all in one go as before, then told me that I had failed when I entered something with the same failure message. I figured I might as well dump out the strings in the file first to see if I would learn anything by doing so. I saw the success and failure messages as before, but none of the other strings seemed promising so I figured I’d need another approach to crack this file.

After some trial and error with a few utilities, I ran objdump with the -S flag on the executable and got a comprehensive dump of the x86 assembly for the program. Our MIPS programming helped with understanding the instructions, but the registers were all different to what I was used to. It was around this point that I decided it would be a good idea to consult some reference sources to figure out what I was dealing with. I read the textbook appendix on using GDB. I read the objdump man page. I also did a little research on what the x86 registers, calling conventions, and instructions all meant so that I could parse what was happening a little more clearly.

When I printed the x86 code for this file, I noticed there were three other methods besides main that were unhelpfully called c(), s(), and p(). I set up breakpoints at the beginning and ends of these functions to poke around the registers and see what I could learn. What I did NOT see was any hint of comparing the user input to a secondary passphrase, though I could see that the functions had a reference to the string I entered. Still, I was able to deduce from doing this that the c() function’s purpose was to strip off the newline character in the input, partly because I saw the transformation and partly because I saw a comparison instruction with the hex literal 0x0a, which is ASCII ‘\n’.

The s() function was called many times and seemed to be performing something like strlen(), since it counted the input length in $eax. That meant the work I was really interested in was probably in the p() method, which I saw ran only one time. I was able to parse out a simple if/else blocking the success and failure messages in main. There was a simple hex literal as the condition which tested what I knew to be the input’s length against the value 6. 6 or less led to failure and 7 or more led to success. I tried running the program again with a codeword longer than 6 characters, but I still got the failure message. I figured this meant that p() was checking some property of the input in addition to this minimum length check. It also seemed like more than one valid codeword might exist because it didn’t check for one specific input length, only a minimum length.

I ran the p() function through gdb with more scrutiny and noticed that it was saving the last character of the string at one point during its execution. I also saw that it was performing some shifting. Since it wanted to know JUST the last character of the string and it seemed to be testing a property of the input rather than testing the input against another source, I guessed at this point that it was probably looking for palindromes. I did a little trial and error to verify my prior guess about the minimum length and newline removal before I was certain I’d worked out all the kinks. There was a lot of wasted time working out assembly lines and pointlessly printing register and pointer values in gdb, but it was helpful practice for solving the third file since some similar reasoning.

Solution 3, filename zdw9\_3

The passwords are: **any 16-character passphrase containing exactly 9 of the following ASCII characters: {0, 4, 9, c, s} The characters can be in any order or combination but exactly 9 must be from this set. Since the program uses getchar(), they can be entered in separate parts line by line, but the whitespace characters count toward the 16 characters assessed.**

**Examples: 000000000~~~~~~ works, but 0000000000~~~~~ would be rejected**

I ran the file first and noticed that this file was taking in characters one at a time rather than as one unit. A little trial and error here showed me that the file was expecting a passphrase of length 16. Dumping out the file’s strings didn’t help, though I didn’t really expect it to. I decided to print a copy of the file’s output to objdump -S to help guide my use of gdb through the file, since I had found this strategy very helpful on the last file. When I did this, however, I noticed some differences from the second executable. There was no clear main method, and the other function names were lost also (except the library calls to puts(), getchar(), and printf(). Hmm. Where to begin…

I went off on a brief tangent here where I tried to decompile the file to C++ using a free utility called Snowman. I was kind of curious what decompiled C++ source would look like, and it was even worse than I’d heard. There were long lists of functions which all had gotos to the same place, lots of places with code wrapped in if(0) blocks for no clear reason, functions which did nothing but call other functions (all of which were just labelled by number in no order whatsoever). It was so asinine that I figured the x86 was better, even though I don’t really understand x86. Still, I was able to see that the success and failure paths were being guarded by a simple if/else with another magic number here, this time 9. If a certain variable was 9, the path went to success, otherwise we went to failure. I could also see that there were conditions which incremented that variable elsewhere, so I figured it represented the count of something or other, but the decompiled C++ was WAY too cryptic to decipher what was being counted.

At this point I tried some codewords to see what happened. I tried, for example, 9 ‘X’s and the rest ‘Y’s. No good. It wasn’t just counting that there were 9 of the same thing then; there was some other condition I was missing. Since I had at least worked out where the main work seemed to be happening in the decompiled source, I looked in the same general vicinity in the x86 code. I started by looking for constant values and values which were stored in non-stack related registers. I quickly found and confirmed the literal 0x9 I’d seen guarding the success path, as well as the 0x10 (decimal 16) which was determining how many characters to read in.

In addition to these literals, I saw a sequence of other promising hex literals. There were a bunch of cmp instructions against the values 0x39 (this one occurs twice), 0x30, 0x63, 0x73, and 0x34. I pulled up an online ASCII table to see what these values would be in decimal and ASCII. In ASCII, they’re 0, 4, 9, c, and s. That was enough information to work out the rules of the passphrase by just trying combinations of these in different orders and mixtures. For instance, uppercase C and S don’t work.

All told, I learned a fair bit about gdbing my way through a maze, the differences between MIPS and x86, what an executable file contains, and why decompiled code is Satan. Not too shabby!