Assignment #4

CPEN 442

October 30th
Ming Hin Matthew Lam 33056145
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

I. Problem #1

1. Steps to finding out the password:
   - First copied my corresponding password to the python bash to compute length of password which was 42
   - Noticed that there were two characters that were standing out, which were the lower cased letters of "th"
   - Noticed that other passwords also followed a similar pattern of hex characters, and two lower cased letters that weren't necessarily hex characters
   - After I subtracted the two characters of salt from my password, I knew there were 40 characters left. Since hex numbers represented 4 bits, this would be a total of 160 bits, and could correspond to a SHA1 encryption.
   - Created a python script that utilizes the hash lib module's SHA1 implementation to brute force the password.
   - There are 4 digits in the password, so each digit has 10 options from 0-9, and we can simply combine 4 for loops to make different permutations of the password.
   - Appended the salt to the end of the password permutation at first, but after computing the hex digest of the SHA1 encryption, none of the password permutations were able to match my given password from the file.
   - Tried instead by adding the salt at the start of the password permutations, and was able to get a match to print out.

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q1/question1.py

2.
   - Password found: 7731, time taken was constant (pretty instantaneous).
   - Entropy b value is 10, l is 4, which equates to approximately 13 bits of entropy.

II. Problem #2

1. Cracking the password:
   - Noticed that the given hashed value of the password is the same as the format from the previous password
   - Instead of only digits, the password has a different character set and is made up of six characters.
   - Tried running code similar to the previous question, except there were six for loops and we iterate through all the possible characters. However, this took too long, and didn't work properly, but source code of failed attempt linked here:

     https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q2/question2.py.

   - Next attempt was through a password cracker hashcat.
   - The hashed value we are cracking is put into a text file.

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q2/hashq2.txt

   - In another file we have the character set provided for each character, and then the mask of the password "us?1?1?1?1?1?1", representing the salt of "us" first followed by six random characters from our first character set. Link of text file here:

     https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q2/charset_q2.txt

   - Ran with command "hashcat -m 100 -a 3 hashq2.txt charset_q2.txt", where m means the mode and 100 means SHA1, a is attack and 3 means brute force.

2.
   - Password Found: usVXpVI3, cpu time taken was approximately 18 minutes.

- Entropy b value is 76, l is 6, which equates to approximately 37 bits of entropy.

III. Problem #3

A)

- Utilized IDA Pro Freeware version to open up and analyze my corresponding exe file.
- Ran my exe file in the command prompt to see how the executable would run, and what type of inputs it was requiring from the user.
- Noticed that the executable had strings such as “Please enter password” and “Access Denied”, so went looking for these strings.
- I entered “Hex View-1” in IDA, and knew that the memory space would hold the strings at a certain address, and kept scrolling down until I found the strings I was looking for.
- Then after returning back to “IDA View-A”, my address was at the strings of “Please enter password” already. I followed the DATA_XREF: sub block by clicking on it which led me to where the string was getting called.
- Noticed that after there were multiple calls to “sub_401520”, and they were all after strings that were printed out, so I knew this call was a print function.
- Stepped through the call of “sub_40C3A0” and found the function was determining the length of the input, and since [ebp+var18] was moved into eco to push, I knew [ebp+var18] held the user inputted by the user.
- The line “cmp eax, 0Bh” also told me it was comparing the length of the input password with value of 11, so the password had to be a length of 11.
- Kept following down the graph view of different blocks that represent the path of the correct password, until I hit “movsx edu, byte ptr [ecx + 1Eh]” from the offset of “off_42E000”.
- Noticed that there was a loop, and figured out that this represented comparing each character of the input password with each character of the correct password, which was offset by an amount of 1Eh from 42E000.
- Double click of the “off_42E000” led me to led me to “.data:0042E000”, and I noticed a string “hnMMBdP…” at this address, and recalled that I previous saw this string while looking for “Please enter password” in hex view. This string was used to compare each character with the input password, so I knew this was the correct password.
- Copied the whole string into the python bash into a variable password. Then I applied the offset and found the following 11 characters by outputting password[30:41], since 0x1E represents 16 + 14 = 30.
- Output was “fz(6ZOSZF(I”, and tested this string with the executable, and confirmed the password by successful login.

B)

- For the patch I noticed that [ebp + var_D] was initially set to 1, and would be set to 0 if the length of the input password was not the same, or if one of the characters from the input didn’t equal that of the correct password. So it acted in the form of a flag.
- If this flag was still set to 1, then at “loc_405318” the “test edx, edx” it would be ANDing 1 with 1, therefore not jumping to “short loc_40535E”, which is the access denied section.
- I knew I had to make sure the instructions would always flow into the correct password section, and this would happen as long as the jump didn’t happen, because the correct section followed right after “loc_405318”.
- I decided to replace the “jz short loc_40535E” instruction with a “nop” instruction, so it’d always go into the correct password section.
- Did so by clicking on the instruction, and going into “Edit -> Patch Program -> Change Word”, and changing the value to “0x9090”. I knew the 90 value corresponded to “nop” by searching up x86 instruction sets online.
- Then I applied the patch by “Edit -> Patch Program -> Apply patches to input file…”, and tested the new executable with a variety of different passwords including my previous determined correct password.
- All passwords allowed for successful login, so I knew the patch worked properly. Then created the DIF file to show the patch applied, linked here:

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q3/33056145.program1.dif

Password: fz(6ZOSZF(I, CPU time taken was constant. Entropy assuming same character set, b value is 76, l is 11, which equates to approximately 68 bits of entropy.

IV. Problem #4

A)

- Took similar steps as previous question to find the strings I was looking for from the executable such as “Please enter password” again.
- The difference this time is that there is a call to SHA1.
- There is still a loop that iterates through each character of the input and each character of the correct password, and this loop runs through 20 iterations as determined by “cmp [ebp+var_14], 14h”.
- I noticed that it was comparing value from “byte_41F1E6” would which represent the SHA1 hashed value of the correct password, but the values at this address had strange symbols.
- Entered Hex View, and I knew that each hex value was 4 bits, so 2 hex values represented a byte. Starting from address 0x0041F1E6, I collected the next 20 bytes or 40 hex values that represented the

SHA1 value of the correct password "4271C42ACC0243410C6F33475C6C69CAE7A4CDF0".
- I had the SHA1 hashed value of the correct password now, so utilized hash cat to help crack for the password.
- Entered the hash value of the password into a text file linked here:

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q4/hashq4.txt

- Next, had the same character set as the previous question 2, except removed the salt of "us", and made a new text file linked here:

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q4/charset_q4.txt

- Performed password cracking through command similar to previous question "hashcat -m 100 -a 3 hashq4.txt charset_q4.txt", which helped me decrypt the password of "&uC=PM".
- Confirmed this password by successfully logging into the executable.

B)
- Performed the same steps as question 3B), by replacing the "jz short loc_405369" instruction after "test ecx, ecx" with a "nop" instruction.
- Created new Dif file for the patch linked here:

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q4/33056145.program2.dif

C)
- Python script asks user for an input, which represents the new password that will be saved in the executable.
- Open the file as a binary file, and allow the script to read or write with the mode of "rb+"
- Required to open as binary file as executable programs are saved in binary format, which allows computers to interpret them.
- Opened up my executable in my text editor, and noticed that each line was composed of 16 pairs of hex values, where each pair presented 1 byte of data.
- Used the find feature in my text editor to find the hex values representing the SHA1 hashed value of the password stored, "4271C42ACC0243410C6F33475C6C69CAE7A4CDF0".
- The values were found on line 7743 of the file, after the 6th pair of hex values. I knew I needed to write to this address of the file, and needed to offset my file pointer to write to this address.
- Utilized python's seek function to offset my file pointer, and because the password was saved on line 7743 after the 6th hex pair, this meant I needed to offset with seek 16 * 7742 + 6 bytes to the file pointer.
- After offsetting the file pointer, I needed to compute the SHA1 value of the user's input.
- The SHA1 value was represented by a hex value string, but because the executable is opened as a binary file, we needed to convert this string back into binary data.
- Incorporated binascii module's unhexlify function to convert the hex string back into binary data.
- After we converted into binary data, we could simply write to the file, which was previously offset from the seek function, and this would replace the address of the executable with the new inputted password.
- Close the file
- Test python script by entering new passwords, and running the executable after, and only the new password allows for successful login.
- Source code of script linked here:

https://github.com/minghinmatthewlam/CPEN442_ass4/blob/master/q4/replacePassword.py

Password: &uC=PM, cpu time taken was approximately 7 minutes.

Entropy b value is 76, l value is 6, which equates to approximately 37 bits of entropy.