Lab #4: crack them again:

CYBR 570 : Reverse Engineering
Spring 2025

Submitted By: Lilly Carlascio
Submission Date: 4/30/25

crackme:

1. I first checked the file type by running "file crackme" to find out it was not stripped. Then I ran "strings crackme" which gave me some important strings information like "Incorrect password", "The password was correct", "Usage: ./crackme ¡key¿". This was important since it shows that you have to put the password with the execution of the binary, it won't prompt you for a password. It also returned some important function named like main, check_id_xor, check_id_sum, and error which hints at what might be going on in the program.

2. I then opened up the binary in ghidra to be able to analyze what was going on in each of the functions to try to crack the password. Opening up the main() function I could see that it expects one input which is the password typed at the command line when executing the binary. The main function verifies that the input is greater than 4 characters, less than 255 characters, and only consists of digits. If those checks pass, the input is copied to a global variable PASS. It then calls check_id_sum and check_id_xor.

   (a) Going into the check_id_xor function, the first and last digit of the PASS variable are taken as integers and the result of those two integers are XOR'd and then that result is checked if it is greater than or equal to both integers, it is valid and returned, if it is not valid, the result is returned as negative. Took me a while to figure out what the DAT_0010403F variable was pulling from since I thought it was a stored data table somewhere else in memory. Instead it was taking the first and last characters from PASS and storing them as integers and storing the XOR'd result in a local variable.

   (b) Next going into the check_id_sum function, it takes the returned result from the previous function. This function iterates over all the characters except the first and last character. It alternates between adding and subtracting digits based on their index: if the index is off, it adds, and if the index is even it subtracts. If the final result is equal to the passed in xor_value and the xor_value does not equal 1, the password passes the check.

3. I next wrote a cracking script to find all of the valid passwords for this binary (crackme_password.c) which brute forces digit-only strings of length 5-9 and checks both conditions. (see readme for execution instructions).

4. In conclusion, by analyzing main, check_id_xor and check_id_sum I was able to uncover the logic behind the password crackme binary.

sneaky:

1. First ran "file sneaky" to find out that the binary was not stripped. Then ran "strings sneaky" which gave us some important strings: "KurtAlan", "Enter the password:", "Access Granted!", "Access Denied!".

2. I then loaded up the file into ghidra for further analysis of the main and other functions. Opening main there was these functions within: generate_password(local_48, "KurtAlan"), encrypt_decrypt(local_48, 0xAA), the user input is read in with fgets, the encrypt function is ran again and the compared with strcmp(). This shows us that the binary is comparing an encrypted user input against a transformed password.

3. Going into the generate_password which takes a local variable for storing the new and the input string (KurtAlan) and iterates through each character in the input and adds 3 to that character and stores it in the output character for the output string. This can be classified as a Caesar's cipher since it shifts each letter by 3 characters. The KurtAlan string is obfuscated into NxuwDodq.

4. Next going into the encrypt_decrypt function, there looks to be a loop that XORs each byte of the user's input buffer with the key (0xAA), which is stored as local_88, to be compared against the generated password.

5. Back into the main function, the string comparison happens between the result of the generate_password function, which is the hardcoded cipher transformation from KurtAlan, and the result of the encrypt_decrypt function which is done twice on the user input, once to encrypt it and once to decrypt it, since XOR does the same operation. If these strings match, the password inputted by the user is correct.

6. The password can be cracked using a c script (sneaky_password.c) which just takes the hardcoded KurtAlan and performs the Caesar cipher on it, which gives the correct password of NxuwDodq (see readme for execution instructions).

cppfun:

1. I first ran "file cppfun" and saw that it wasn't stripped which is good. Then I ran "strings cppfun" which showed me a lot of the references to the Crypto++ library including "_ZN8CryptoPP8Rijndael3EncE" and "CBC_Encryption" which led me to believe that the binary possible encrypts/decrypts using ARS in CBC mode.

2. Opening the file in ghidra, I can enter the main program and find the first function mutexstr¡int¿(local_50, local_30) which seems to be the function that checks the password. The local_30 is the user input string, local_50 is a pointer to a Crypto++ AES object, and there is another variable declared, local_50, that is the current attempt number, since this program lets you try the password 3 times before it locks you out as an unsuccessful. The result of this function is a Boolean and if it is True that means the password is accepted but if all 3 attempts fail, it prints "Too many attempts. Exiting..."

3. Going into the mutexstr() function I was honestly really lost and couldn't get much out of all of the functions and lost my motivation to do so. I thought it was doing something similar to the sneaky binary but I couldn't find the right places to look to confirm this, and I wasn't able to find the password. I was able to find a strcmp() function comparing the user's input and the encrypted password, but I wasn't able to figure out what the decrypted password is. My best guess is Fudfnph but I wasn't able to test it fully since I am now travelling and only have my Mac to work on which won't let me execute the binary.