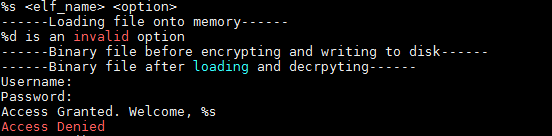
# Analysis from running program

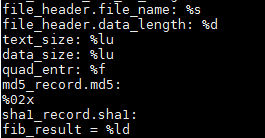
# 

The program does not seem to need an external password file meaning the values for authentication are embedded in the program.

# Analysis from strings -a proj1:

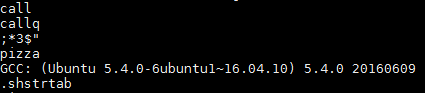


These strings show that the program does have an encryption scheme to protect a binary file and write it to disk, as well as the ability to pull up a previously encrypted file and decrypt it.



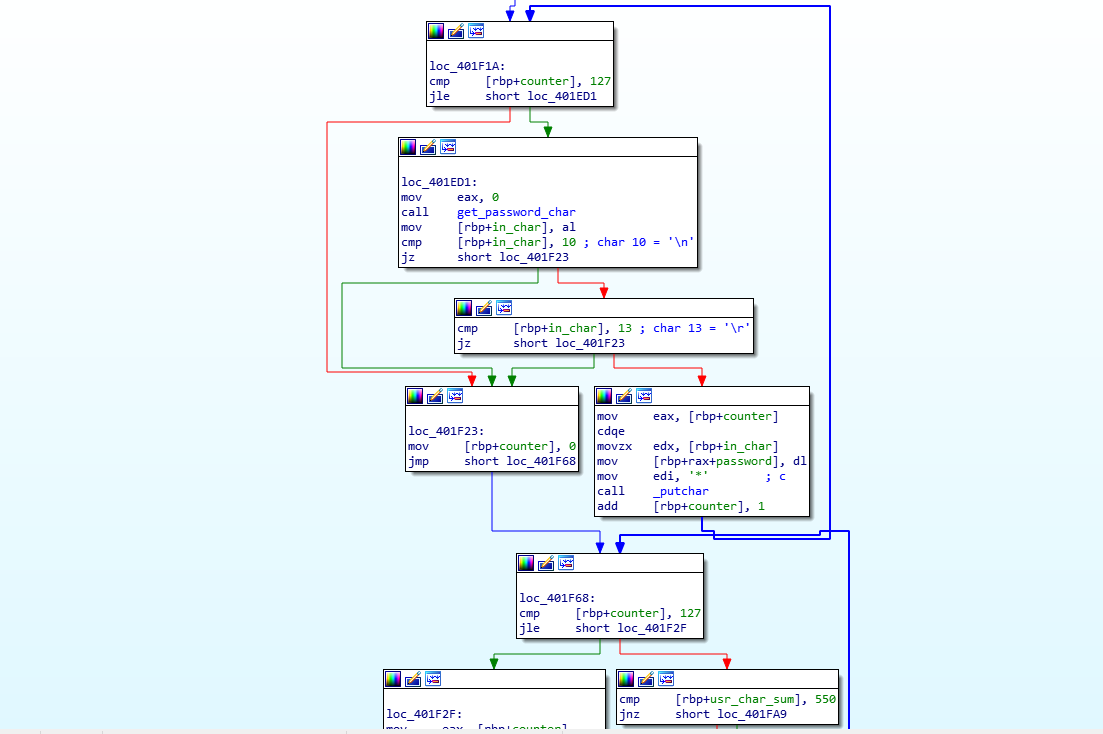
These strings show that the program will analyze:

* file\_name
* data\_length
  + Not sure what this field means right now, how is it different from data\_size
* text\_size
* data\_size
* quad\_entr
  + Probably Renyi quadratic entropy
* md5 hash
* sha1 hash
* fib\_result
  + Not sure what this one is but is of type long signed

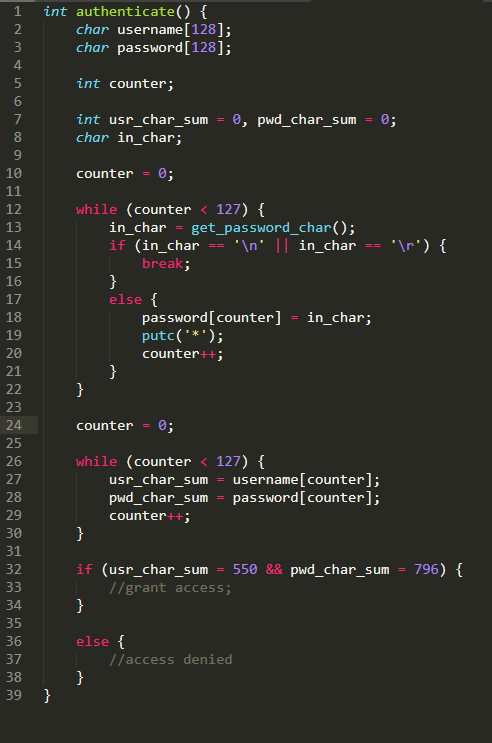


Here’s an interesting snippet, inside this program is the string “pizza” somewhere. Perhaps this is a password of some sort?

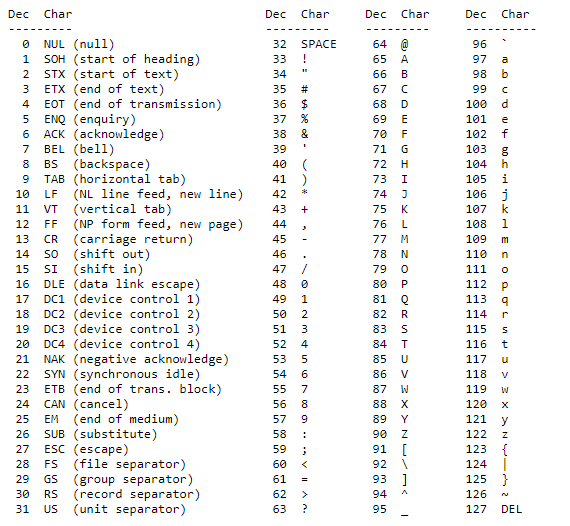
# Disassembly of the authentication function



The authentication method does not do any string comparisons, and after much thought and reversing, the C code looks something like this:

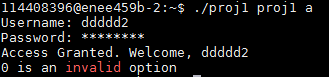


We can see that on line 32, we compare the sum of the char values in the inputted username and password to see if they add to 550 and 796 respectively. Using an ascii table:



We can pick a username of **dddd2** (100+100+100+100+100+50)

and we pick a password of **ddddddd`** (100+100+100+100+100+100+100+96)

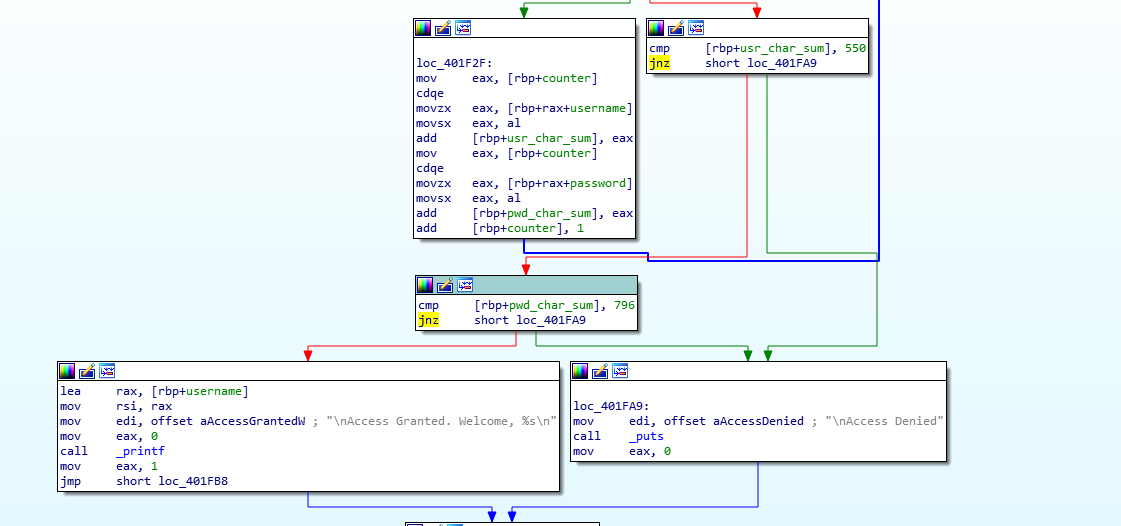


And we are in!

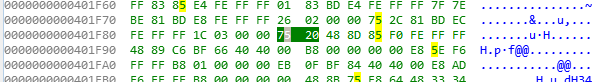
Clearly this is not a secure system, there is a huge number of possible usernames and passwords, and passwords are not unique to any particular username.

# Bypassing the password

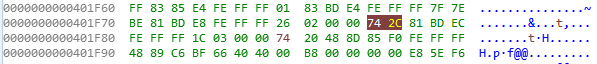
I got tired of entering the password



In hex, this jump instruction (and another) determines whether authentication passes or fails



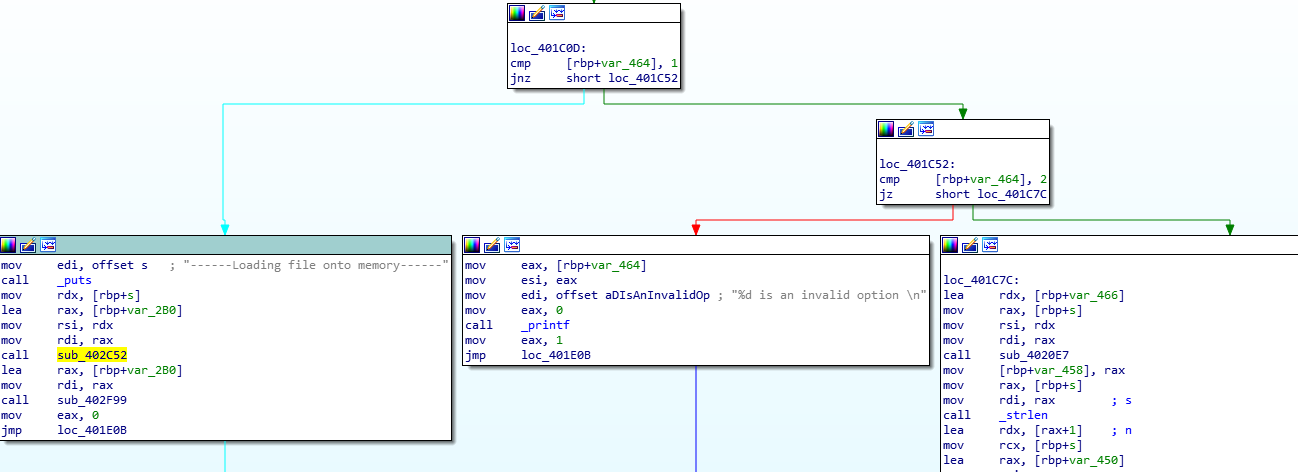
Changing this 75 to a 74 should change both jnz to a jz and bypass authentication





Any password/username will now work.

# Running the program

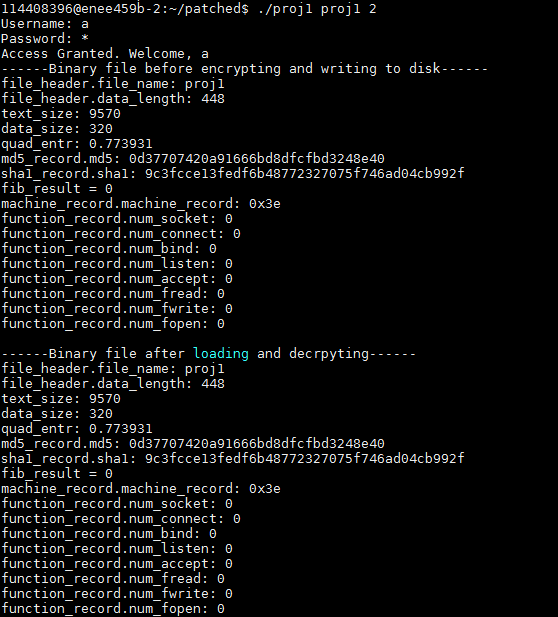


Looking at what options are available after authenticating, it seems like there are two options for <option>:

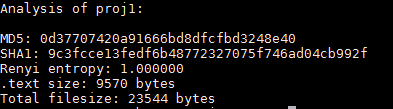
1: decrypt an output log

2: analyze an ELF binary and save an encrypted output log

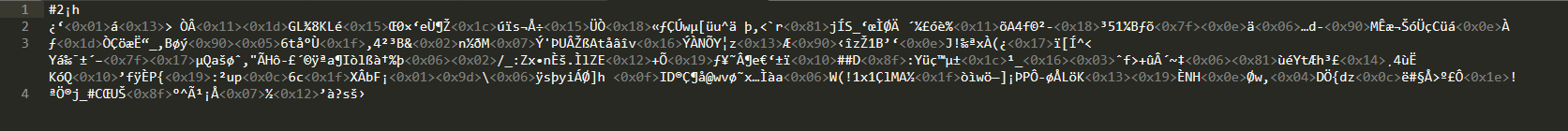
Trying out 2 first, the program outputs



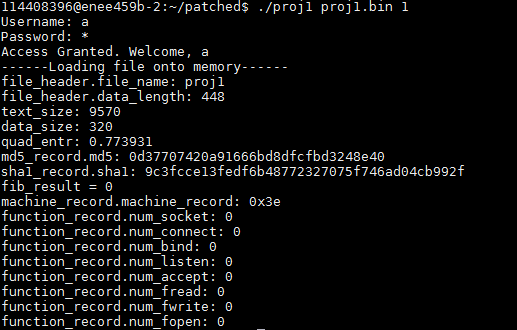
Comparing this to the output of my program shows that the MD5, SHA1, and text size are all correct.



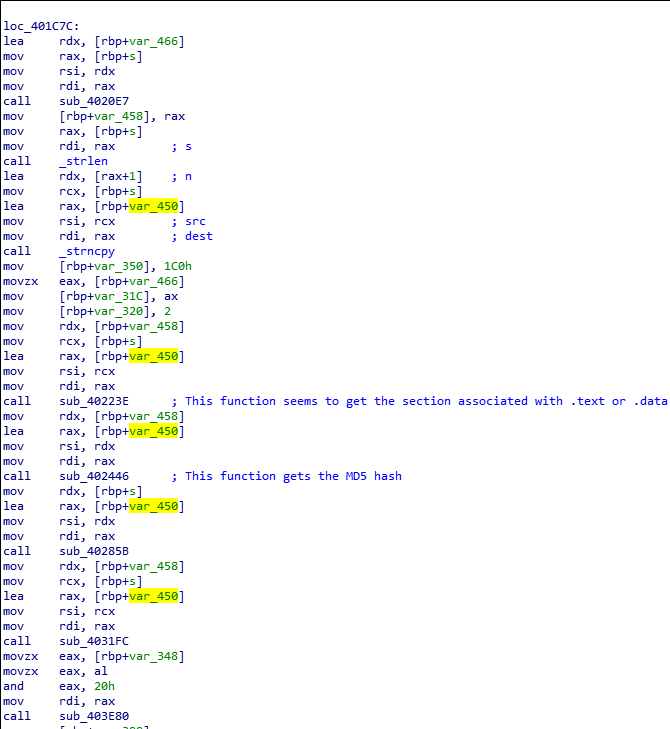
After running with option 2, we get an output file called proj1.bin with the contents



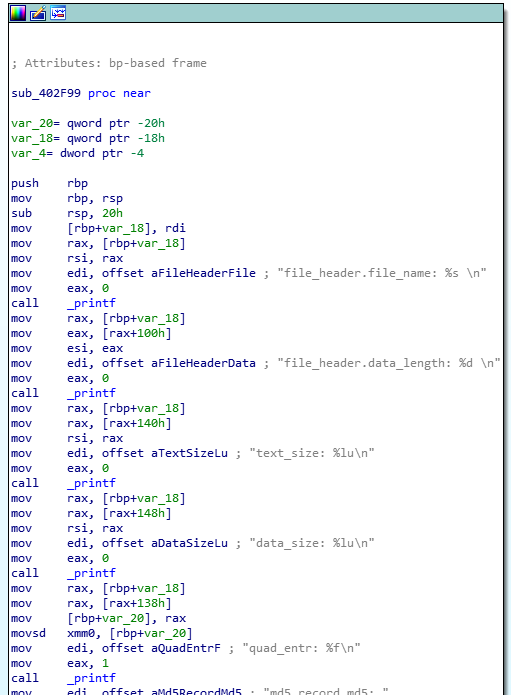
Which is very unreadable. Using option 1 in the command ‘./proj1 proj1.bin 1’ reverses this encryption and prints out the stored analysis:



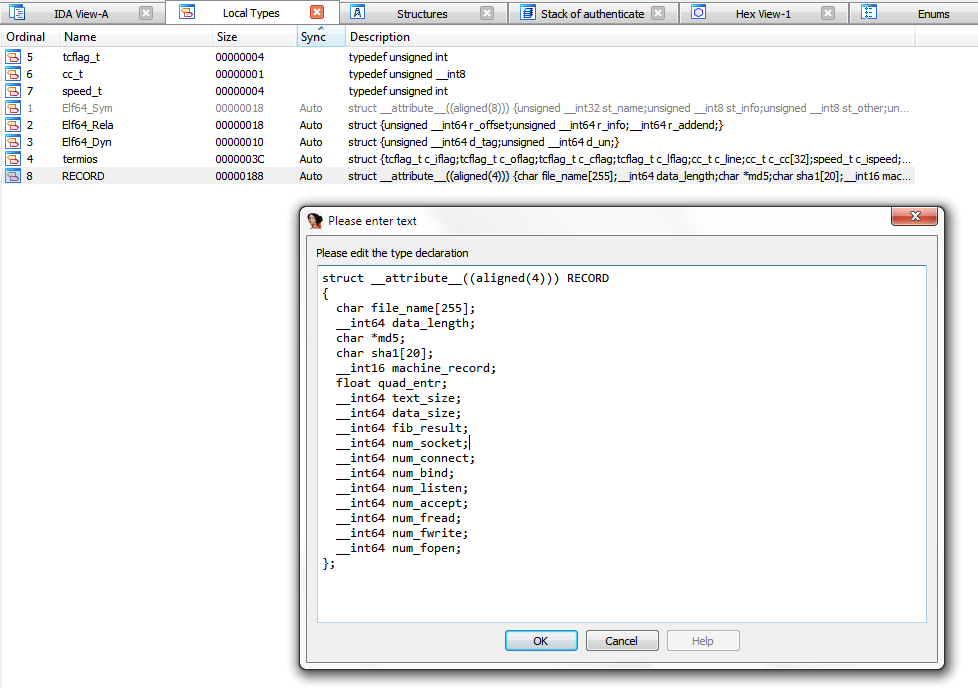
# How the program stores information



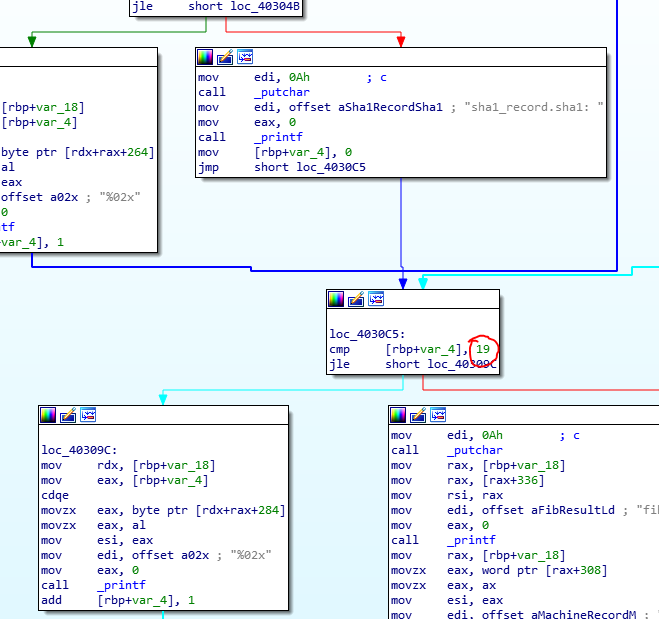
This block in the program repeatedly calls sub functions while passing in [rbp+var\_450]. Examining sub\_402F99 later on in the code shows that [rbp+var\_450] is passed into this function and fields are extracted from it:



Here, we see that the passed in variable is incremented before being printed. This tells us that the passed in variable is most likely a struct and tells us about the fields of the struct and their size. Using IDA pro’s features we can rename it to reflect this discovery.



Though these are only a few of the fields in the struct as getting the alignment of them proved to be difficult since accesses to the struct were not sequential (ie first listed type accessed first) so the offsets jumped around. However we were able to deduce that the file\_name char array is of length 255 and the sha1 is a 20 byte array:



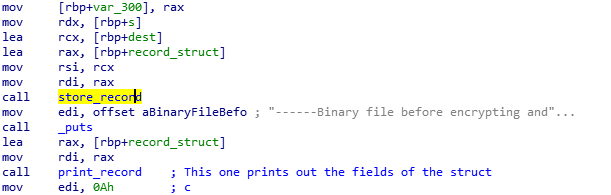
Knowing that [rbp+450h] is a struct, we can tell that the program keeps its information in a struct, and then passes that struct pointer into functions which get information about the binary file and writes it to the struct as seen here:



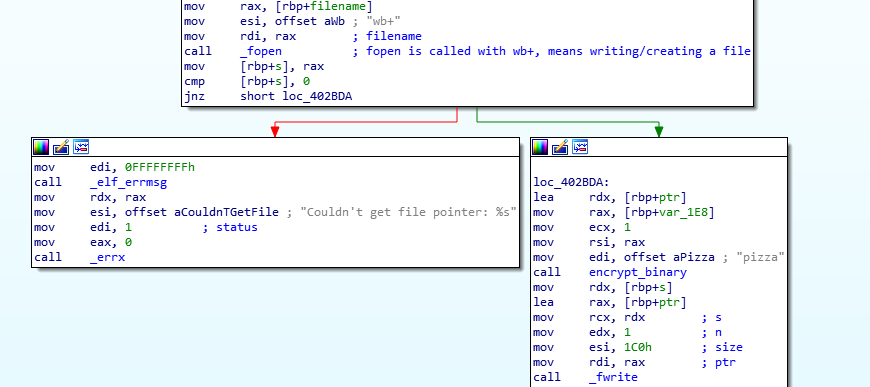
This struct is then later written as a binary to a file.

# Encrypting and decrypting the record

After the program gathers all necessary information and puts it into the record, it writes it to a file but not before encrypting it:



And in store\_record



We see that it opens a file but before writing at loc\_402BDA, it passes in ‘pizza’ to a function I’ve named “encrypt\_binary”. There, from the strings, it is obvious what encryption scheme was used:



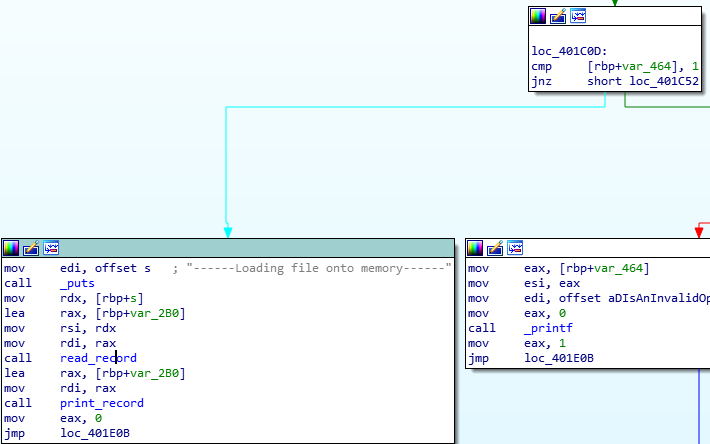
A quick Google search of the strings passed into the OpenSSL functions show

<https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption>

which describes the encryption scheme used.

One big potential vulnerability is that the key used to encrypt the plaintext is hardcoded into the program. The Wiki specifically suggests against this!

On the opposite end, when the program is called with option 1, it runs a function I’ve named “read\_record”



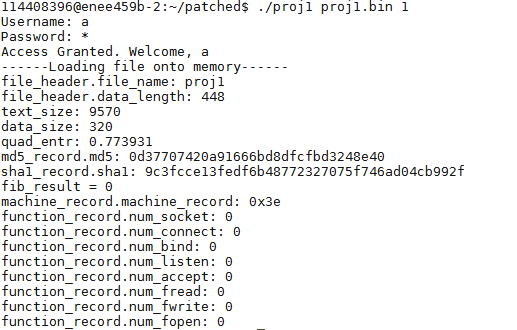
It has a similar structure to store\_record but instead of opening a file with options ‘wb+’, it uses ‘rb+’ which indicates this function reads files.

Since the encryption scheme is symmetric, encrypting the stored record again with the key “pizza” will yield the plaintext from before.

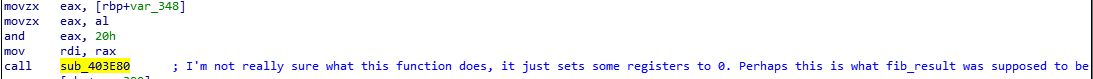
# Main takeaways:

Going down the list of goals, we have discovered that:

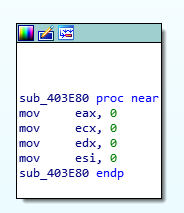
1. This software runs off one command passing in a valid elf file and an option either 1 or 2. It does not have any secret/administrator modes and runs standalone since the user authentication method does not do any comparisons to a password database.
2. The program authenticates by checking that the sum of the char values of the username is equal to 550, and the sum of the char values of the password is 796. It can be defeated in the sense that it is very insecure. Also I patched it out if that counts.
3. The information gathered from the elf is evident from normal operation



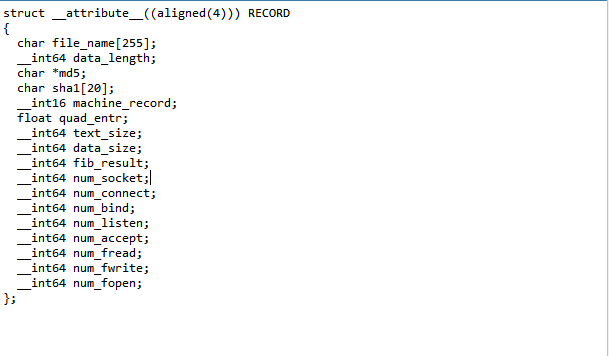
1. Compared to my program, the hash values and the text sizes are correct. One value does not seem to work and that is fib\_result. My guess is that this segment handles fib\_result



But looking into the actual code for sub\_403E80 shows a skeleton function that just returns 0:



1. After analyzing an executable, the program stores the results in proj1.bin, a binary file. The struct used to record this data looks something like this:



But some of the ordering may be incorrect.

1. The stored proj1.bin binary file is obfuscated and encrypted through AES using the key “pizza”. Since it is symmetric, the key “pizza” is used to decrypt the file as well. The username and password are kind of obfuscated in the sense that it’s not really a proper username/password system.
2. Two obvious vulnerabilities are the user authentication process which doesn’t actually store any usernames or passwords, and the encryption of the binary file which uses a password stored in plaintext hardcoded into the program.