Lab1_Report

Name: Wenhui Zhang
Email: wuz49@ist.psu.edu

1. Using cryptographic notation, specify the cryptographic protocol used to generate the encryption and HMAC keys from the master password.

Solution:

enc_key = h(master_password ⊕ pseudo_random_number_1)
Where we are using SHA256 for hash function, ⊕ is XOR

hmac_key = h(master_password ⊕ pseudo_random_number_2)
Where we are using SHA256 for hash function, ⊕ is XOR

```
/*In this task , I init with srand for a fixed rand, and that is why it is named as
pseudo_random_number, the same thing is done in question 3 as well */
srand(1);
```

2. Using cryptographic notation, specify the cryptographic operations used to produce an encrypted password that prevents leakage of both the length and value of the password when written to disk.

Solution:

E(password + padding, enc_key) + HMAC(password + padding, hmac_key)
As length of password should be hidden, before doing encryption, I allocated a VALSIZE(128) memory for password_buffer, and this memory is init with all 0's. By putting password in start of this chunk of memory, we got concatenation of password and 0's, and its length is 128. In this way, we prevent ourselves from leakage of both the length and value of the password when written to disk.

3. Specify your password strengthening approach.

Solution:

For each strengthening round, XOR a pseudo_random_number_1 with passwords' first character and XOR a pseudo_random_number_2 with last character. This method works very well under the consideration that min length of input_password is 8 characters, and it achieves Inf after one round for all my tests.

```
/*In this task , I init with srand for a fixed rand, and that is why it is named as
pseudo_random_number, the same thing is done in question 1 as well */
srand(1);
```

4. Given the findings of attacks by Schmidt and Jaeger [3] against methods to strengthen passwords, suggest how you would improve your password strengthening approach to avoid such attacks?

Solution:

In Jaeger‘s paper, markov chains and PCFG-based cracking algorithms use password lists to find distribution of characters in passwords in order to guess the passwords. I added random number and XORed it with password, this XOR random number approach destroys the distribution. Thus is prone to these kind of attacks.
To be more efficient, we could use multiple threads for paralleling XORing random number with random character of password. This double randomness will add more diffusion and confusion for strengthening of passwords.

5. Specify how the tag is computed during the authenticated encryption using the OpenSSL API.
Solution:
We uses “clen = encrypt(pwdbuf, VALSIZE, (unsigned char *)NULL, 0, enc_key, iv, ciphertext, tag); “ to get tag, and tag is a cipher block chaining message authentication tag.

This function is calling OpenSSL API below:

```
/* Get the tag */
if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
handleErrors();
```
And by using EVP_rc2_40_cbc(), EVP_rc2_64_cbc(), we got tag calculated as cipher block chaining message authentication tag.

EVP_rc2_40_cbc(), EVP_rc2_64_cbc():

> RC2 algorithm in CBC mode with a default key length and effective key length of 40 and 64 bits. These are obsolete and new code should use EVP_rc2_cbc(), EVP_CIPHER_CTX_set_key_length() and EVP_CIPHER_CTX_ctrl() to set the key length and effective key length.

Ref:

1. https://wiki.openssl.org/index.php/Manual:EVP_EncryptInit(3)