HW 3: Communication with Hash Function

1. Introduction

In the last two assignments, we are familiar with symmetric encryption to ensure the confidentiality of the messages. In this assignment, we will learn how to use the hash function to ensure the integrity of the messages.

Consider the following scenario, Alice wants to send a message to Bob containing a file. If an attacker manages to intercept the downloaded file, alter the file's contents, and then forward the altered file to the recipient, that malicious act won't go unnoticed. That's because, once the client runs the tampered file through the agreed hash algorithm, the resulting hash won't match the downloaded hash. However, while a hash function can establish data integrity, it can't establish authenticity. Therefore, HMAC comes out, which stands for Keyed-Hashing for Message Authentication [1]. It's a message authentication code obtained by running a cryptographic hash function (like MD5, SHA1, and SHA256) over the data (to be authenticated) and a shared secret key. Many popular transfer protocols such as HTTPS, SFTP, and FTPS use HMAC.

2. Tasks

Specifically, we'll implement the HMAC function, and we will use the same public API as in HW1/HW2, OpenSSL [2]. You can use your own code from HW1/HW2 or the answer codes to build the socket channel.
Note: We recommend this assignment be done in C/C++ language.

2.1 Implement HMAC Function

The general encryption/decryption function contains four steps:
1. Initialize the context variable (i.e., ctx).
2. Initialize the HMAC function by choosing the hash engines and key. There are many hash engines to choose from, such as EVP_md5(), EVP_sha1(), EVP_sha224(), EVP_sha512, etc. Generally, SHA-1 is cryptographically stronger than MD5 and SHA-2 (like SHA-224, SHA-256, and SHA-512) is likewise cryptographically stronger than SHA1. You may need to take that into consideration and specify in the report which one you choose and why. It is worth mentioning that for different hash engines, the output hash values have different lengths.
3. Use the initialized HMAC function to obtain the hashed output.
4. Clean up and free the context variable.

The code skeleton is as follows, please fill in the function parameters according to the documentation in Section 5 below.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/hmac.h>
#include <openssl/err.h>

void cal_hmac(unsigned char *mac,  char *message)
{
   /* The secret key for hashing */
   const char key[] = "";

   /* Change the length accordingly with your chosen hash engine.
   * Be careful of the length of string with the chosen hash engine. For
example, SHA1 needed 20 characters. */
   unsigned int len = 20;

   /* Create and initialize the context */
   HMAC_CTX *ctx;
   ctx = HMAC_CTX_new();

   /* Initialize the HMAC operation. */
   HMAC_Init_ex();

   /* Provide the message to HMAC, and start HMAC authentication. */
   HMAC_Update();

   /* HMAC_Final() writes the hashed values to md, which must have enough
space for the hash function output. */
   HMAC_Final();

   /* Releases any associated resources and finally frees context variable
*/
   HMAC_CTX_free(ctx);

   return;
}
```

2.2 Implement Main Function

In the main function of the client and server, first we will need to set up the communication channel as we did in HW1/HW2. Then in the client, we need to calculate the HMAC hashed values of the messages to be sent by using the HMAC function defined in Section 2.1, and then we send the messages and HMAC separately to the server. **Note: Be sure to include your name in the message sent from client to server.** In the server, we first receive the message and the hashed values. Then we re-calculate the HMAC hashed values of the received messages by using the same HMAC function defined in Section2.1. Lastly, we check if the calculated hashed value is equal to the received hashed value to authenticate the messages, if they are equal, then the received message is successfully authenticated.
Note: we could compare char one by one to check if two strings are equal.

Be sure that you send and receive the same messages and hashed values. Print out the sent messages and hashed values in the client. Also, print out received hashed values, calculated hashed values, and the result of the authentication in the server.

2.3 Compile and Troubleshooting

Note we use the OpenSSL header in our code, so we need to include the library during the compilation process.

For Ubuntu users, please first install OpenSSL through:
sudo apt-get install libssl-dev
Then Compile the program through:
gcc crypto.c -L/usr/lib -lssl -lcrypto -o server

For Mac users, the compilation process is relatively complex, there is a tutorial to follow: https://unix.stackexchange.com/questions/346864/how-to-link-openssl-library-in-macos-using-gcc.

3. Things to Turn-in

You are expected to turn in 1) all of your source code used to build the server and the client, and 2) your report that explains what you have done in this lab. Be sure to include the screenshot of the results on both the server and the client. To turn in the source code, just copy and paste the code and put it in the appendix of the report.

4. References

[1] Krawczyk, Hugo, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-hashing for message authentication*. No. rfc2104. 1997.

[2] OpenSSL. https://www.openssl.org/.

5. Useful Functions

Create and initialize the HMAC context, the function will return a pointer to a new HMAC_CTX on success or NULL if an error occurred.
*HMAC_CTX *HMAC_CTX_new(void);*

HMAC_Init_ex()* initializes or reuses an *HMAC_CTX* structure to use the hash function *evp_md* and *key*. If both are NULL, or if *key* is NULL and *evp_md* is the same as the previous call, then the existing key is reused. **md is the hash engine (e.g., EVP_sha1), you can set *impl to NULL.

```
int HMAC_Init_ex(HMAC_CTX *ctx, const void *key, int key_len,
                 const EVP_MD *md, ENGINE *impl);
```

HMAC_Update* calculates the HMAC hashed values of **data, len* represents the length of **data. HMAC_Update()* can be called repeatedly with chunks of the message to be authenticated (*len* bytes at data). For *HMAC_Final,* it writes the final hashed values to **md, len* is the length of **md.

```
int HMAC_Update(HMAC_CTX *ctx, const unsigned char *data, size_t len);
int HMAC_Final(HMAC_CTX *ctx, unsigned char *md, unsigned int *len);
```

Clean up the context variable.
*void HMAC_CTX_free(HMAC_CTX *ctx);*

For more details and usage, please refer to https://www.openssl.org/docs/man1.1.1/man3/HMAC_CTX_new.html.