HW 4: Communication with Authenticated Encryption

1. Introduction

In the last two assignments, we've learned how to use symmetric encryption to ensure the confidentiality of the messages and how to use the hash function to ensure the integrity of the messages. In this assignment, we try to combine these two and learn to use authenticated encryption to ensure both confidentiality and integrity.

Galois/Counter Mode (GCM) is a mode of operation for symmetric-key cryptographic block ciphers which is widely adopted for its performance [1]. The operation is an authenticated encryption algorithm designed to provide both data authenticity (integrity) and confidentiality. GCM is defined for block ciphers with a block size of 128 bits. GCM is adopted in many popular communication protocols, such as IEEE 802.1AE (MACsec) Ethernet security, IEEE 802.11ad (also dubbed WiGig), ANSI (INCITS) Fibre Channel Security Protocols (FC-SP), etc.

2. Tasks

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

2.1 Implement Encryption Function

There are four inputs for authenticated encryption: the secret key, initialization vector (IV), the plaintext itself, and optional additional authentication data (AAD). There are two outputs: the ciphertext, which is exactly the same length as the plaintext, and an authentication tag. The tag is sometimes called the message authentication code (MAC).

The general encryption function contains four steps:

1. Initialize the context variable (i.e., ctx).
2. Initialize the encryption operation by setting the mode, key, and iv. There may be separate steps to complete the initialization. First, it will set the encryption mode, then it will set the length of iv, and finally, it will set the key and iv. There are many models to choose from, such as EVP_aes_256_gcm() and EVP_aes_256_ccm().

3. Encrypt the messages by GCM to obtain the output ciphertext and authentication tag. First, AAD data is provided to the encryption, the AAD data is not encrypted, and is typically passed to the recipient in plaintext along with the ciphertext. Then the encryption is conducted. In the following stage, the encryption is like the normal block cipher, but there is one additional stage, where the authentication tag is generated.
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.

```
int gcm_encrypt(unsigned char *plaintext, int plaintext_len,
                unsigned char *aad, int aad_len,
                unsigned char *key,
                unsigned char *iv, int iv_len,
                unsigned char *ciphertext,
                unsigned char *tag)
{
   EVP_CIPHER_CTX *ctx;
   int len,ciphertext_len;

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

   /* Initialize the encryption operation. */
   EVP_EncryptInit_ex();

   /*
    * Set IV length if default 12 bytes (96 bits) is not appropriate
    */
   EVP_CIPHER_CTX_ctrl();

   /* Initialize key and IV */
   EVP_EncryptInit_ex();

    /*
    * Provide any AAD data. This can be called zero or more times as
    * required
    */
   EVP_EncryptUpdate();

   /*
```

```
    * Provide the message to be encrypted, and obtain the encrypted
output.
    * EVP_EncryptUpdate can be called multiple times if necessary
    */
   EVP_EncryptUpdate();

   ciphertext_len = len;

   /*
    * Finalize the encryption. Normally ciphertext bytes may be written at
    * this stage, but this does not occur in GCM mode
    */
   EVP_EncryptFinal_ex();

   ciphertext_len += len;

   /* Get the tag */
   if(EVP_CIPHER_CTX_ctrl();

   /* Clean up */
   EVP_CIPHER_CTX_free(ctx);

   return ciphertext_len;
}
```

2.2 Implement Decryption Function

In the decryption function, it will first authenticate the messages, if the authentication is successful, then the messages will be decrypted.

The general encryption function contains four steps:

1. Initialize the context variable (i.e., ctx).
2. Initialize the decryption operation by setting the mode, key, and iv. There may be separate steps to complete the initialization. First, it will set the decryption mode, then it will set the length of iv, and finally, it will set the key and iv.
3. Authenticate and then decrypt the messages by GCM to obtain the plaintext. First, AAD data is provided for the decryption, then the decryption is conducted with the given ciphertext. In the next stage, the expected tag value is provided, and the decryption module is finalized with authenticated values and possible plaintext if the authentication is successful.

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.

```
int gcm_decrypt(unsigned char *ciphertext, int ciphertext_len,
                unsigned char *aad, int aad_len,
                unsigned char *tag,
                unsigned char *key,
                unsigned char *iv, int iv_len,
                unsigned char *plaintext)
{
    EVP_CIPHER_CTX *ctx;
    int len, plaintext_len, ret;

    /* Create and initialize the context */
    EVP_CIPHER_CTX_new();

    /* Initialize the decryption operation. */
    EVP_DecryptInit_ex();

    /* Set IV length. Not necessary if this is 12 bytes (96 bits) */
    EVP_CIPHER_CTX_ctrl();

    /* Initialize key and IV */
    EVP_DecryptInit_ex();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    EVP_DecryptUpdate();

    /*
     * Provide the message to be decrypted, and obtain the plaintext
output.
     * EVP_DecryptUpdate can be called multiple times if necessary
     */
    EVP_DecryptUpdate();
    plaintext_len = len;
```

```
    /* Set expected tag value. Works in OpenSSL 1.0.1d and later */
    EVP_CIPHER_CTX_ctrl();

    /*
     * Finalize the decryption. A positive return value indicates success,
     * and anything else is a failure - the plaintext is not trustworthy.
     */
    ret = EVP_DecryptFinal_ex();

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    if(ret > 0) {
        /* Success */
        plaintext_len += len;
        return plaintext_len;
    } else {
        /* Verify failed */
        return -1;
    }
}
```

2.3 Implement Main Function

In the main function of the client and server, firstly we will need to set up the communication channel as we did in previous assignments. Then in the client, we need to calculate the ciphertext and authenticated tag of the messages to be sent, and then we send the ciphertext and tag separately to the server. In the server, we first receive the ciphertext and the tag. Then we authenticate and decrypt the received ciphertext by using the received tag. Lastly, we check if the authentication is successful, if so, decrypt the messages.

For AAD data, you can just hard-code it, as it is sent as plaintext. Be sure that you send and receive the same ciphertext and tag values. Print out the ciphertext and tag values in the client. Also, print out authentication results and decrypted messages on 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] Lemsitzer, Stefan, et al. "Multi-gigabit GCM-AES architecture optimized for FPGAs." International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, 2007.
[2] OpenSSL. https://www.openssl.org/.

5. Useful Functions

Create and initialize the cipher context, the context will hold state data and intermediate calculations.
*EVP_CIPHER_CTX *EVP_CIPHER_CTX_new(void);*

Initialize the encryption cipher, *EVP_EncryptInit_ex()* sets up cipher context *ctx* for encryption with cipher type from ENGINE *impl.* If impl is NULL then the default implementation is used. *ctx* must be created before calling this function. *type* is normally supplied by a function such as *EVP_aes_256_gcm().* You will need to set the mode and key/iv separately in the code.
*int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,*
*ENGINE *impl, const unsigned char *key, const unsigned char *iv);*

Set and get parameters that are used by ciphers, *EVP_CIPHER_CTX_ctrl()* set the parameters of *cmd* to *p1.* In the encryption function, you will need to use this function to

set the length of iv. In the decryption function, you will need to use it to set the expected tag values.
*int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int cmd, int p1, void *p2);*

Encrypt the plaintext, *EVP_EncryptUpdate* encrypts *inl* bytes from the buffer *in* and writes the encrypted version to *out*. For stream ciphers, the amount of data written can be anything from zero bytes to *inl* bytes. Thus, *out* should contain sufficient room for the operation being performed. If padding is enabled (the default) then *EVP_EncryptFinal_ex()* encrypts the "final" data. There is one additional stage that takes AAD data as input.
*int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,*
*int *outl, const unsigned char *in, int inl);*
*int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);*

Initialize the decryption cipher, the usage is the same with initializing the encryption cipher.
*int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,*
*ENGINE *impl, const unsigned char *key, const unsigned char *iv);*

Decrypt the ciphertext, *EVP_DecryptUpdate* decrypts *inl* bytes from the buffer *in* and writes the encrypted version to *out*. *EVP_DecryptFinal_ex* is used to decrypt the final data when the padding is enabled (the default). There is one additional stage that takes AAD data as input.
*int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,*
*int *outl, const unsigned char *in, int inl);*
*int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);*

Clean up the cipher.
*void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *ctx);*

For more details and usage, please refer to https://www.openssl.org/docs/.