CSE 433S HW 3 - Comm with Hash Function

Zihan Chen

October 2024

Contents

1 Design

Hash function is used to achieve the integrity of the data.

There are several kinds of hash functions. MD5, SHA1, SHA256, etc.

MD5 is an old hash function and is recognized as an unsafe hash function in today's world.

SHA1 is also an old hash function and is found many vulnerabilities.

SHA256 is a recommend (recommended by NIST) hash function to use instead of MD5 and SHA1. We believe that SHA256 has a higher security traits than MD5 and SHA1.

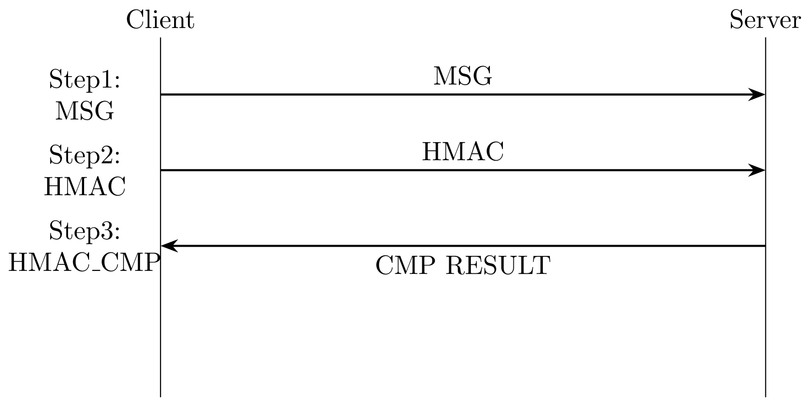
So in this homework, I use the SHA256 hash function in OPENSSL to calculate the hash value of the data.

The whole process is as follows:

1. Client send original MSG to server.

2. Client send Hash of MSG to server.

3. Server compare Hash from Client and Hash from calculating by itself and send the compare result back.



1.1 hmac_cmp

In my testing result, I find that if we use strcmp function, even if two hash result is the same, it still print out that they are not the same. So I should design a compare function by myself.

Since we know the MAC is 32bytes in SHA256, we only need to compare 32 bytes of the string. And use the following function we can compare 32 bytes one by one.

```
bool hmac_cmp(unsigned char *mac1, unsigned char *mac2) {
    for(int i = 0; i < 32; i++) {
        if(mac1[i] != mac2[i]) {
            return false;
        }
    }
    return true;
}
```

1.2 handle_errors

For future coding and debugging, a handle error function is important.

We use the following function to handle all the errors from OPENSSL library functions.

```
void handleErrors(void) {
    ERR_print_errors_fp(stderr);
    abort();
}
```

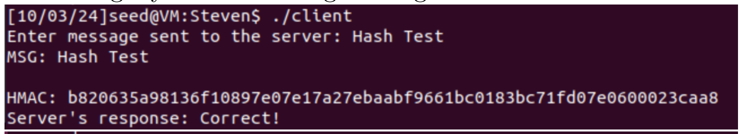
2 Result

The running result is as follows:

After TCP connection, the client will send a message and then send the HMAC of the message.

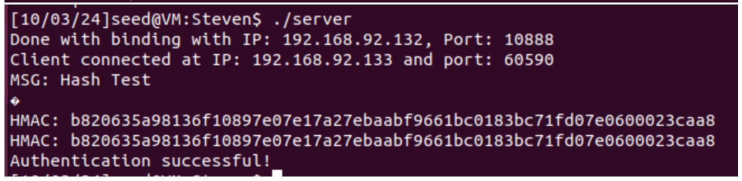
The server can receive the message and the HMAC and compare the HMAC from client with the one it calculate by itself.

We can find the two hash is the same, and after comparing we can make sure the integrity of the sending message.



```
[10/03/24]seed@VM:Steven$ ./client
Enter message sent to the server: Hash Test
MSG: Hash Test

HMAC: b820635a98136f10897e07e17a27ebaabf9661bc0183bc71fd07e0600023caa8
Server's response: Correct!
```



```
[10/03/24]seed@VM:Steven$ ./server
Done with binding with IP: 192.168.92.132, Port: 10888
Client connected at IP: 192.168.92.133 and port: 60590
MSG: Hash Test
�
HMAC: b820635a98136f10897e07e17a27ebaabf9661bc0183bc71fd07e0600023caa8
HMAC: b820635a98136f10897e07e17a27ebaabf9661bc0183bc71fd07e0600023caa8
Authentication successful!
```

3 Appendix

3.1 Server Code

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
```

```
#include <unistd.h>
#include <openssl/hmac.h>
#include <openssl/err.h>

void handleErrors(void) {
    ERR_print_errors_fp(stderr);
    abort();
}

void cal_hmac(unsigned char *mac, char *message, size_t
   msg_len) {
/* The secret key for hashing */
    const char key[] = "SECRET_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. */
    // sha256 needs 32 characters
    unsigned int len = 32;


    /* Create and initialize the context */
    HMAC_CTX *ctx;
    ctx = HMAC_CTX_new();
    if(ctx == NULL) {
        handleErrors();
    }


    /* Initialize the HMAC operation. */
    if(HMAC_Init_ex(ctx, key, strlen(key), EVP_sha256(),
       NULL) != 1) {
        handleErrors();
    }


    /* Provide the message to HMAC, and start HMAC
       authentication. */
    if(HMAC_Update(ctx, (unsigned char*)message, msg_len)
        != 1) {
        handleErrors();
    }
    /* HMAC_Final() writes the hashed values to md, which
         must have enough space for the hash function
        output. */
    if(HMAC_Final(ctx, mac, &len) != 1) {
        handleErrors();
    }

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

    return;
}

bool hmac_cmp(unsigned char *mac1, unsigned char *mac2) {
    for(int i = 0; i < 32; i++) {
        if(mac1[i] != mac2[i]) {
            return false;
        }
    }
    return true;
}

int main() {
    // Declare variables
    ssize_t varread;
    char server_message[1024];
    char client_message[1024];

    unsigned char mac[32];
    unsigned char server_mac[32];
    int msg_len;

    struct sockaddr_in server_addr;
    char server_ip[16]= "192.168.92.132";
    int server_port = 10888;

    struct sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);


    // Create socket
    int client_sock;
        int sever_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sever_sock < 0) {
        perror("Socket creation failed with error!");
        return 1;
    }
    if(setsockopt(sever_sock, SOL_SOCKET, SO_REUSEADDR,
        &(int){1}, sizeof(int)) < 0) {
        perror("Socket option failed with error!");
        return 1;
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
    server_addr.sin_addr.s_addr = inet_addr(server_ip);


    // Bind to the set port and IP
    if(bind(sever_sock, (struct sockaddr*)&server_addr,
        sizeof(server_addr)) < 0) {
        perror("Binding failed with error!");
        return 1;
    }
    printf("Done with binding with IP: %s, Port: %d\n",
        server_ip, server_port);

    // Listen for clients:
    if(listen(sever_sock, 3) < 0) {
        perror("Listening failed with error!");
        return 1;
    }

    // Accept an incoming connection
    if((client_sock = accept(sever_sock, (struct sockaddr
        *)&client_addr, &client_addr_len)) < 0) {
        perror("Accepting failed with error!");
        return 1;
    }
    char * client_ip = inet_ntoa(client_addr.sin_addr);
    int client_port = ntohs(client_addr.sin_port);
    printf("Client connected at IP: %s and port: %i\n",
        client_ip, client_port);

    // Receive client's message
    msg_len = recv(client_sock, client_message, 1024, 0);
    printf("MSG: %s\n", client_message);

    // Receive client's HMAC
    varread = recv(client_sock, mac, 32, 0);
    printf("HMAC: ");
    for(int i = 0; i < 32; i++) {
        printf("%02x", mac[i]);
    }
    printf("\n");

    // Calculate HMAC
    cal_hmac(server_mac, client_message, (size_t)msg_len)
        ;
    printf("HMAC: ");
    for(int i = 0; i < 32; i++) {
        printf("%02x", server_mac[i]);
    }
    printf("\n");


    if(hmac_cmp(mac, server_mac)) {
        printf("Authentication successful!\n");
        // Respond to client
        strcpy(server_message, "Correct!\n");
        send(client_sock, server_message, strlen(
            server_message), 0);
    } else {
        printf("Authentication failed!\n");
    }

    // Close the socket
    close(client_sock);
    close(sever_sock);


    return 0;
}
```

3.2 Client Code

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

void handleErrors(void) {
    ERR_print_errors_fp(stderr);
    abort();
}


void cal_hmac(unsigned char *mac, char *message, size_t
   msg_len) {
/* The secret key for hashing */
   const char key[] = "SECRET_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
       . */
   // sha256 needs 32 characters
   unsigned int len = 32;


   /* Create and initialize the context */
   HMAC_CTX *ctx;
   ctx = HMAC_CTX_new();
   if(ctx == NULL) {
        handleErrors();
   }


   /* Initialize the HMAC operation. */
   if(HMAC_Init_ex(ctx, key, strlen(key), EVP_sha256(),
      NULL) != 1) {
        handleErrors();
   }


   /* Provide the message to HMAC, and start HMAC
       authentication. */
   if(HMAC_Update(ctx, (unsigned char*)message, msg_len)
      != 1) {
        handleErrors();
   }

   /* HMAC_Final() writes the hashed values to md, which
       must have enough space for the hash function output
       . */
   if(HMAC_Final(ctx, mac, &len) != 1) {
        handleErrors();
    }

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

    return;
}

int main() {
    // Declare Variables
    struct sockaddr_in server_addr;

    char server_ip[16] = "192.168.92.132";
    int server_port = 10888;
    char server_message[1024];
    char client_message[1024];

    unsigned char mac[32];

    ssize_t varread;

    // Create socket:
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
    server_addr.sin_addr.s_addr = inet_addr(server_ip);

    // Send connection request to server, be sure to set
        port and IP the same as server-side
    if(connect(sock, (struct sockaddr*) &server_addr,
        sizeof(server_addr)) < 0) {
        perror("Connection failed with error!");
        return 1;
    }

    // Get input from the user:
    printf("Enter message sent to the server: ");
    fgets(client_message, sizeof(client_message), stdin);

    // print the message
    printf("MSG: %s\n", client_message);

    // Send the message to server:
    send(sock, client_message, strlen(client_message), 0)
```

```
        ;

    // Send the HMAC to the server:
    cal_hmac(mac, client_message, strlen(client_message))
        ;

    // print the HMAC
    printf("HMAC: ");
    for(int i = 0; i < 32; i++) {
        printf("%02x", mac[i]);
    }
    printf("\n");

    // Send the HMAC to the server:
    send(sock, mac, sizeof(mac), 0);

    // Receive the server's response:
    varread = read(sock, server_message, 1024);

    printf("Server's response: %s\n",server_message);

    // Close the socket:
    close(sock);

    return 0;

}
```