Università di Pisa

Computer Engineering

Foundations of Cybersecurity

Secure Bank Application

Group Project Report

TEAM MEMBERS:
Federico Casu
Daniel Deiana

Academic Year: 2022/2023

Contents

Chapter 1

Application overview and requirements

A Secure Bank Application (SBA) is a client-server application that allows users to issue operations on their own bank accounts. Each user is modeled by `username` and a `password` whereas each bank account by an `accountID` and a `balance`. For simplicity we assume that a user owns a single account and that an account is owned by a single owner. An user may issue the following operations on its own bank account:

- `(accountId, balance)` ← `Balance()`, which returns the user's bank account's `accountId` and `balance`.
- `bool` ← `Transfer(UserName other, uint amount)`, which takes both the username of another user (`other`) and an `amount` of money as input parameters and transfers such an amount from the user's bank account to the other user's bank account. The operation returns `false` if the value of balance is smaller than the value of amount; it returns `true` otherwise.
- `ListofTranfers` ← `History()`, which returns the last `T` transfers performed by the user, where `T` is a system configuration parameter. Each transfer is a triple (user, amount, timestamp).

The SBA application is hosted by a server which maintains users and accounts. Users interact with the SBA server through a secure channel that must be established before issuing operations.

Requirements:

1. Each principal owns a pair of private and public keys.
2. The SBA server maintains the public key of every user.
3. Each user maintains the public key of the server.
4. Users are initialized at registration time which is off-line.
5. `OpenSSL` API `TLS` cannot be used.
6. The authentication protocol must fulfill **Perfect Forward Secrecy** (PFS).
7. Client-server communication must fulfill **confidentiality**, **integrity**, **no-replay** and **non-malleability**.
8. Password and history of transfers of any given user cannot be stored in the clear.

Chapter 2

Application design and implementation choices

2.1 Server

The server logic, encapsulated within the `Server` class, is implemented using a thread pool-based multithreaded server.

```cpp
class Server {
public:
    Server(int port, int n_workers, int backlog, volatile sig_atomic_t*
    g_signal_flag);
    ~Server();
    void accept_connections();

private:
    int port;                   // Port used to listen client's requests.
    int sock_fd;                // TCP socket used to listen
                                // the incoming requests.
    int backlog;                // Parameter used to decide how big
                                // the backlog will be.
    int n_workers;              // Number of workers (threads).
    sockaddr_in address;        // Server's address.
    jobs_t jobs;                // Shared structure used to share tasks
                                // among the workers.

    std::vector<std::thread> threads;
    std::vector<Worker*>     workers;

    volatile sig_atomic_t* g_signal_flag;

    int  create_socket();       // To create the listener socket.
    void bind_socket();         // To bind the listener socket.
    void listen_socket();       // To listen the incoming connection
                                // (using listener socket).
};
```

```cpp
struct jobs {
    std::vector<int> socket_queue;
    std::atomic_bool stop;
    std::mutex socket_mutex;
    std::condition_variable socket_cv;
};
typedef struct jobs jobs_t;
```

The server accepts incoming (TCP) connections and assigns each connection to a worker thread for processing. This task is handled by the `Server::accept_connections()` method.

More in details:

```cpp
void Server::accept_connections() {

    jobs.stop = false;

    threads.reserve(n_workers);
    workers.reserve(n_workers);

    for (int i = 0; i < n_workers; i++) {
        Worker* worker = new Worker(&jobs);
        workers.push_back(worker);
        threads.emplace_back([&worker]() { worker->Run(); });
    }

    struct sockaddr_in client_address;
    socklen_t addrlen = sizeof(struct sockaddr_in);
    int client_socket = -1;
    struct timeval timeout;

    while (!(*g_signal_flag)) {
        // Set up the file descriptor set for select()
        fd_set read_fds;
        FD_ZERO(&read_fds);
        FD_SET(sock_fd, &read_fds);

        // Set up the timeout value for select()
        timeout.tv_sec  = 1;
        timeout.tv_usec = 0;

        // Use select() to wait for incoming connections with a timeout
        int res = select(sock_fd+1, &read_fds, nullptr, nullptr, &timeout);

        if (res == -1) {
            if(errno != EINTR) {
                std::string error_message = strerror(errno);
                throw std::runtime_error("Failed to select(): "
                    + error_message);
            }
            continue;
        }

        if (res == 0) // Timeout occurred, no incoming connections
            continue;

        client_socket = accept(sock_fd,
                        reinterpret_cast<struct sockaddr*>(&client_address),
                        &addrlen);

        if (client_socket == -1) {
            std::cerr << "[RUNTIME EVENT] "
                      << "Failed to accept an incoming connection: "
                      << strerror(errno) << std::endl;;
            continue;
        }

        {
            // This lock ensures that 'socket_queue' is accessed safely
            // by preventing concurrent access from multiple threads.
            std::lock_guard<std::mutex> lock(jobs.socket_mutex);
            jobs.socket_queue.push_back(client_socket);
        }

        // This notifies one waiting worker thread that a
        // new connection is available in the queue.
        jobs.socket_cv.notify_one();
    }
}
```

1. The worker threads are created and associated with the shared `jobs` structure (Listing

2.1), which they will use to receive and process client connections. Each worker is then added to the `threads` vector. The latter will be used to join each running thread when the server is stopped.

2. Inside the main loop, the server utilizes the `select` function to wait for incoming connections. This allows the server to monitor the listener socket for any activity. The purpose of using `select` with a timeout is to introduce a mechanism to catch terminating signals, such as when the user presses Ctrl + C. Since the `accept` function is blocking, the timeout ensures that the loop periodically checks for signals and prevents the server from being unresponsive. If the `select` function returns a value of -1, it checks if the error is due to a signal interruption (EINTR) and continues the loop in that case. If the `select` function returns 0, it means that the timeout occurred, indicating no incoming connections during that period.

3. The server accepts the incoming connection using the `accept` function, which returns a client socket for the new connection.

4. The client socket is added to the `socket_queue` in the `jobs` structure, ensuring thread-safe access using a `std::mutex`.

5. One waiting worker thread is notified that a new connection is available in the queue using the `std::condition_variable socket_cv`.

6. The loop continues until the `g_signal_flag` indicates that the server should stop.

The thread's logic is encapsulated within the `Worker` class.

```cpp
class Worker {
public:
    Worker(jobs_t* jobs);
    ~Worker();

    void Run();
private:
    std::vector<uint8_t> iv;
    std::vector<uint8_t> hmac_key;
    std::vector<uint8_t> session_key;
    std::string username;
    uint8_t counter;
    uint32_t max_list_transfers;
    const std::string server_private_key_path = "../res/private_keys/
    server_privkey.pem";

    int client_socket;
    jobs_t* jobs;

    // Communication methods
    ssize_t Receive(std::vector<uint8_t>& buffer, ssize_t buffer_size);
    ssize_t Send(const std::vector<uint8_t>& buffer);

    // Key exchange protocol
    void Handshake();
    bool ClientExists(uint8_t* username, ssize_t username_size);

    // Worker Logic
    ClientReq RequestHandler();
    void BalanceHandler();
    void TransferHandler(uint8_t* recipient, uint32_t msg_amount);
    void SendTransferResponse(bool outcome);
    void ListHandler();
};
```

Each thread executes `Worker::run()`:

```
void Worker::Run() {

    while (true) {
        {
            std::unique_lock<std::mutex> lock(jobs->socket_mutex);
            jobs->socket_cv.wait(lock, [&]() { return !jobs->socket_queue.empty()
     || jobs->stop; });

            if (jobs->stop) {
                #ifdef DEBUG
                std::cout << BLUE_BOLD << "[WORKER]" << RESET << " >> stop" <<
    std::endl;
                #endif
                return;
            }

            client_socket = jobs->socket_queue.front();
            jobs->socket_queue.erase(jobs->socket_queue.begin());
        }

        std::cout << BLUE_BOLD << "[WORKER]" << RESET
                  << " >> Client connected (socket: "
                  << client_socket << ")." << std::endl;

        try {
            Handshake();

            while(true) {
                ClientReq request = RequestHandler();

                std::cout << BLUE_BOLD << "[WORKER]" << RESET << " >> ";
                std::cout << "Client " << client_socket << " -> "
                        << request.request_code << ":"
                        << request.recipient << ":"
                        << request.amount << std::endl;

                CheckCounter(request.counter);

                switch(request.request_code) {
                    case CODE_BALANCE_REQUEST: {
                        BalanceHandler();
                        break;
                    }
                    case CODE_TRANSFER_REQUEST: {
                        TransferHandler();
                        break;
                    }
                    case CODE_LIST_REQUEST: {
                        ListHandler();
                        break;
                    }
                    default:
                        throw std::runtime_error("\033[1;31m[ERROR]\033[0m Bad
    format message (request_code not known).");
                }

                std::memset(reinterpret_cast<void*>(&request), 0, sizeof(
    ClientReq));
            }
        } catch(std::runtime_error& e) {
            std::cerr << BLUE_BOLD << "[WORKER]" << RESET << " >> "
                      << e.what() << std::endl;

            close(client_socket);

            // Something went wrong: we need to clear the session
            // (session key and HMAC key).
            std::memset(reinterpret_cast<void*>(hmac_key.data()),
                        0,
                        hmac_key.size());
            hmac_key.clear();

            std::memset(reinterpret_cast<void*>(session_key.data()),
                        0,
                        session_key.size());
            session_key.clear();

            continue;
        }
    }
}
```

1. The worker waits for a signal from the `jobs->socket_cv` condition variable. It waits until either the `socket_queue` is not empty or the stop flag is set. This synchronization is achieved using a unique lock on `jobs->socket_mutex`.
2. If the `stop` flag is set, the worker returns from the function, effectively terminating the thread.
3. If there is a job in the `socket_queue`, the worker takes the first job from the queue and removes it.
4. The worker then performs a handshake operation using the `Handshake()` method. During the handshake, server and client exchange the session key and the hmac key using a protocol based on the well known Station-to-Station protocol.
5. Based on the request code, the worker dispatches the request to the appropriate handler function (`BalanceHandler()`, `TransferHandler()`, or `ListHandler()`).
6. If an exception is caught during the processing of a request the client socket is closed and the session keys (`hmac_key` and `session_key`) are cleared.

The handshake phase is handled by `Worker::Handshake()` method. In particular, the session establishment protocol implemented by `Worker::Handshake()` method follows these steps:

1. A `Worker` waits until a new `M1` message arrives. The `M1` structure is presented in Figure 2.1. `M1` size is fixed: the first field of `M1` message (`ephemeral_key_size`) is used retrieve the meaningful bytes from the `ephemeral_key` field.
2. Upon receiving the `M1` message, the `Worker` thread can determine if the user who performed the request is actually registered with the Secure Bank service by examining the `username` field. If the user is registered, the `Worker` thread continues to execute the key establishment protocol. However, if the user is not registered, the `Worker` thread clears the newly generated session and notifies the client that he/she is not allowed to continue the key establishment protocol.
3. Shared secret:
   - Creates a `DiffieHellman` object and generates the ephemeral key `b`.
   - Retrieves the client's ephemeral key, i.e. g^a, from the `M1` message.
   - Generates a shared secret $(g^a)^b$ using the Diffie-Hellman algorithm.
   - Delete its ephemeral key, i.e. `b`.
4. The shared secret is used to generate session and HMAC keys using SHA-512.
5. The `Worker` thread signs the serialized ephemeral keys, i.e. `<`g^a`|`g^b`>` with the server's RSA private key.
6. The signature is encrypted using `AES` in `CBC` mode with the session key. The resulting ciphertext and `IV` (initialization vector) will be sent to the client.

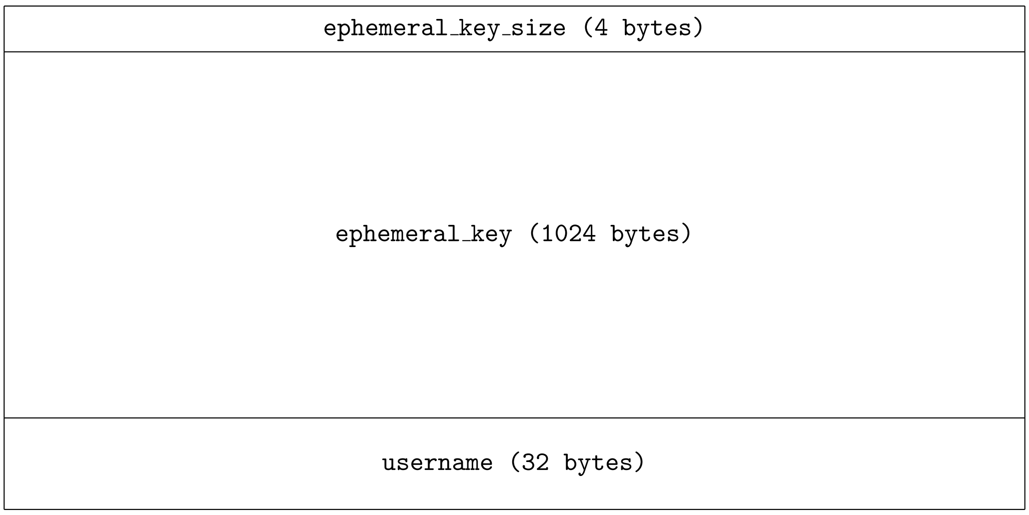


Figure 2.1: `M1` message structure.

7. The `Worker` thread sends a `M2` message (Figure 2.2) to the client. In this message, only the `encrypted_signature` field is encrypted.

8. The `Worker` thread waits until the client complete its key derivation phase. In particular, the client notifies the server that its key generation and server authentication have completed with success by sending a `M3` message (Figure 2.3). The encrypted client's signature in `M3` is decrypted using `AES-CBC` with the session key. The decrypted signature is verified using an RSA public key corresponding to the client. If the verification fails, the session will be cleared.

9. Moreover, to complete the login procedure, the `Worker` receives the encrypted client's password. The `Worker` proceds to check the password. If all checks pass, the session is considered established. Otherwise, the `Worker` will clear the session and notifies the client with a generic error.

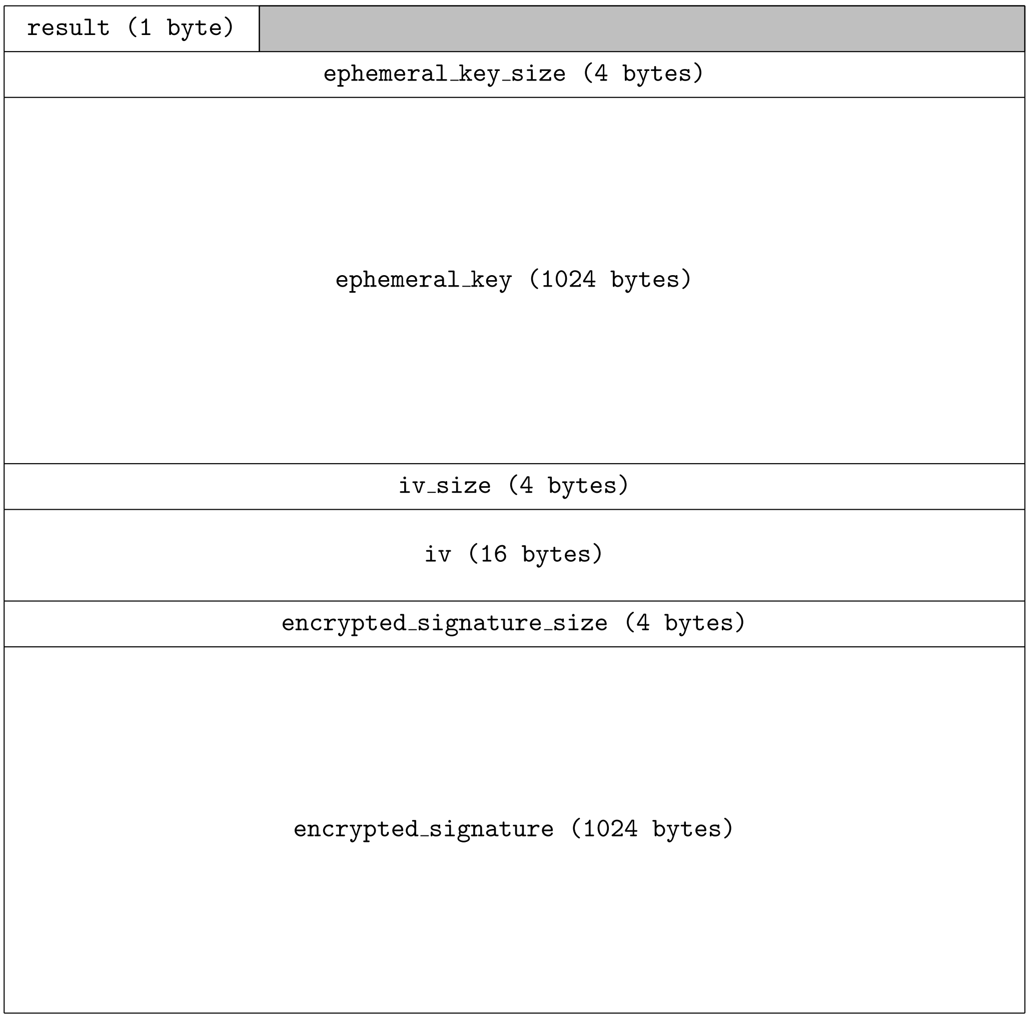

result (1 byte)
ephemeral_key_size (4 bytes)
ephemeral_key (1024 bytes)
iv_size (4 bytes)
iv (16 bytes)
encrypted_signature_size (4 bytes)
encrypted_signature (1024 bytes)

Figure 2.2: `M2` message structure.

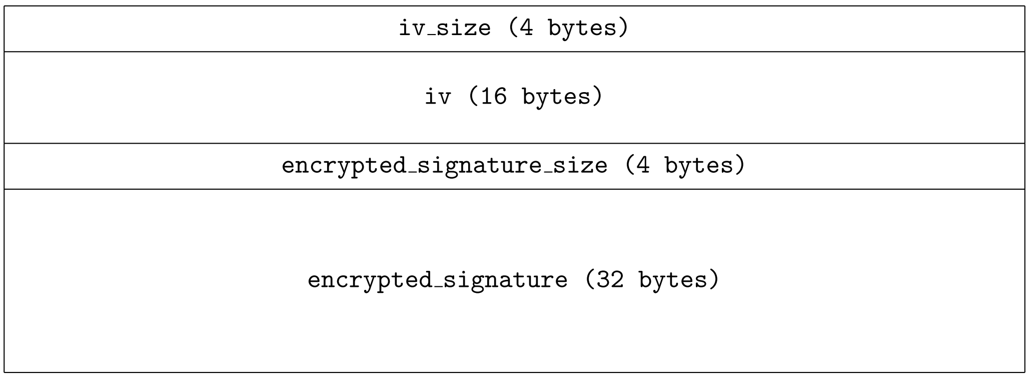

iv_size (4 bytes)
iv (16 bytes)
encrypted_signature_size (4 bytes)
encrypted_signature (32 bytes)

Figure 2.3: `M3` message structure.

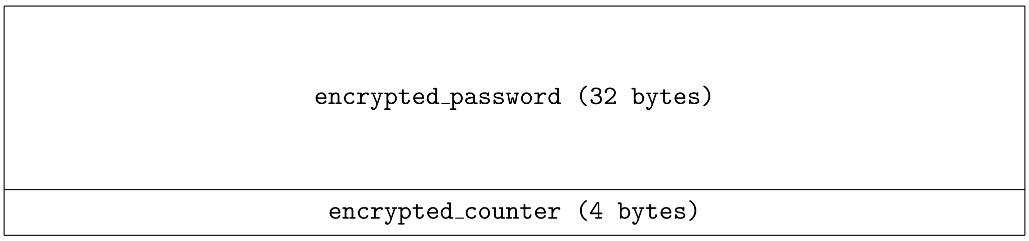


Figure 2.4: M4 message structure.

2.2 Client

The client's logic is encapsulated within the `Client` class. `Client` is defined with member functions for connecting to a server, sending and receiving data to/from the server, and performing operations such as balance inquiry, fund transfer, and listing transactions. More in details:

```
class Client {
public:
    Client(const std::string& server_ip, int server_port);
    ~Client();
    void connect_to_server();

    void handshake();

    void balance();
    void transfer();
    void list();

private:
    int sock_fd;
    struct sockaddr_in server_address;

    uint32_t counter;                   // Session counter
    std::vector<uint8_t> hmac_key;      // Session HMAC key
    std::vector<uint8_t> session_key;   // Session key used for AES CBC

    std::string username;   // Client's username

    void send_to_server(const std::vector<uint8_t>& buffer);
    void recv_from_server(std::vector<uint8_t>& buffer);

    void turnOffEcho();     // Turn off echo when user is typing the passwd
    void turnOnEcho();      // Turn on echo once the user typed the passwd
};
```

The client-side implementation of the handshake procedure follows these steps:

1. The handshake procedure starts by prompting the user to enter a username and password. The password input is masked with asterisks to provide privacy.
2. The `Client` generates an ephemeral key (using Diffie-Hellman algorithm), serializes it, and creates an instance of `M1`, which encapsulates the serialized ephemeral key and the username.
3. `M1` message is sent to the server.
4. The `Client` waits a `M2` message from the server, which contains the result of the existence of the user. If `M2.result` is 0, indicating that the user doesn't exist, the ephemeral key is deleted and a generic error is thrown.
5. If the user is registered to the Secure Bank service, the `Client` continues the key establishment protocol retrieving the server ephemeral key from `M2`, generates a shared secret using

its own ephemeral key and the server's ephemeral key, and derives session and HMAC keys from the shared secret using `SHA-512`.

6. Once the shared secret has been generated, the `Client` deletes its ephemeral key.
7. The `Client` prepares a buffer containing `<g^a|g^b>` and calculates the signature of the buffer using the user's private key. The signature is then encrypted with the session key.
8. The `Client` decrypts the encrypted signature received with `M2` using the session key.
9. The `Client` verifies the decrypted signature using the server's public key:
   - If the signature verification is successful, the `Client` creates a `M3`, which contains an initialization vector (IV) and the encrypted signed `<g^a|g^b>`, and sends it to the server.
   - If the signature verification fails, the `Client` clears the session (deleting the session and HMAC key) and throws a generic error.
10. Also, once the `M3` message has arrived to the `Server`, the `Client` encrypts its own password and sends it in order to complete the login procedure.
11. After successful completion of the handshake, the `Client` sets a counter and informs the user that the session has been established.

Chapter 3

Security Protocols

3.1 Session Establishment Protocol

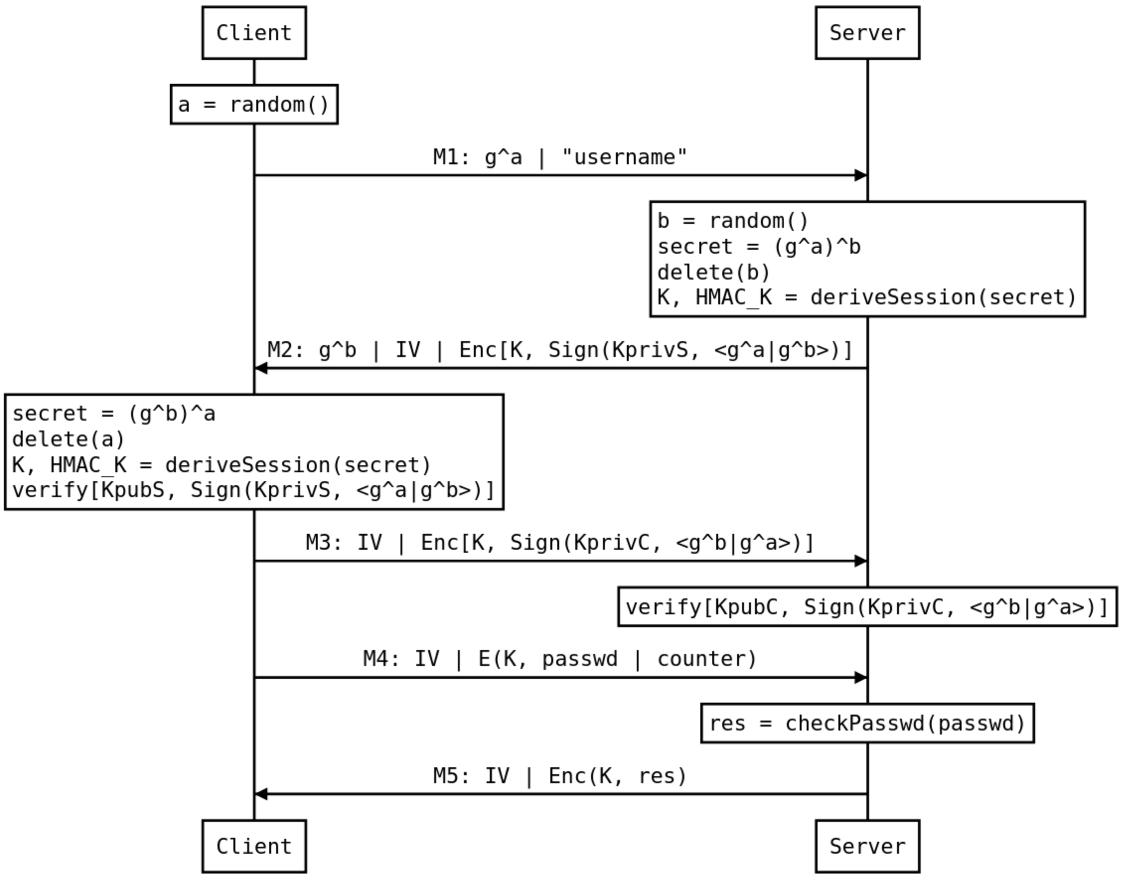
The protocol we implemented for the Secure Bank application is an authenticated key exchange protocol based on the well-known Station-To-Station (STS) protocol. The protocol consists of Diffie-Hellman key establishment, followed by an exchange of authentication signatures.

3.1.1 Assumptions

We assume that the parameters used for the key establishment (i.e., the specification of a particular cyclic group and the corresponding primitive element `g`) are fixed and known to both `Client` and `Server`. When a user opens a bank account, the Bank's manager installs the authentic Bank's public key on the user's application. Additionally, the user provides their own authentic public key to the Bank's manager. Therefore, we can assume that the `Client` knows the `Server`'s authentic public key, and vice versa.

3.1.2 Session establishment protocol: an overview

The protocol begins with the `Client` creating a random number `a` and sending the ***1)*** exponential `g^a` and ***2)*** its own username to the `Server` (see message `M1` in Figure 3.1). Upon the `Server` receiving the `M1` message, it verifies if the user is registered to the Secure Bank service. If the verification fails, the `Server` stops the session establishment protocol and notifies the `Client` with a generic error. Otherwise, the `Server` creates a random number `b` and uses `Client`'s exponential to compute the shared secret `(g^a)^b`. The `Server` responds with the exponential `g^b` and a token consisting of his signature on the exponentials, i.e. `RSA::Sign(KprivS, <g^a|g^b>)`. The token is encrypted with `K` using the `AES` algorithm in `CBC` mode. `K` is the session key derived from the shared secret `(g^a)^b`. In particular, we decided to hash the shared secret with `SHA-512`; the 256 least-significant bits of the hashed secret are assigned to the session key `K` and the 256 most-significant bits are assigned to the `HMAC_K` session key. Upon the `Client` receiving `g^b` and the signed encrypted token, computes `K` and `HMAC_K` key, decrypts the token using `K`, and verifies `Server`'s signature using `Server`'s public key. The `Client` sends to `Server` his/her corresponding token, i.e. `RSA::Sign(KprivS, <g^b|g^a>)`.. Once the `Server` receives `M3` message (see message `M3` in Figure 3.1), it verifies `Client`'s encrypted signature using `K` and `Client`'s public key. From now on, `Client` and `Server` share `K` and `HMAC_K`. The last step of the session establishment protocol is initiated by the `Client` sending its encrypted password (followed by the HMAC of `IV || Enc(K, password)`). Upon receiving the encrypted message, the `Server` decrypts it and performs an integrity check using the newly generated `HMAC_K`. Finally, the `Server` checks if the user's password is correct.


Client
Server
a = random()
M1: g^a | "username"
b = random()
secret = (g^a)^b
delete(b)
K, HMAC_K = deriveSession(secret)
M2: g^b | IV | Enc[K, Sign(KprivS, <g^a|g^b>)]
secret = (g^b)^a
delete(a)
K, HMAC_K = deriveSession(secret)
verify[KpubS, Sign(KprivS, <g^a|g^b>)]
M3: IV | Enc[K, Sign(KprivC, <g^b|g^a>)]
verify[KpubC, Sign(KprivC, <g^b|g^a>)]
M4: IV | E(K, passwd | counter)
res = checkPasswd(passwd)
M5: IV | Enc(K, res)
Client
Server
Figure 3.1: Session establishment protocol.

3.1.3 Security assurances

At this point we will consider what assurances the session establishment protocol provides to the participants. From `Server`'s point of view, as a result of the Diffie-Hellman key exchange, he shares a key known only to him and the other participant who may or may not be `Client`. Because the `Client` has signed the exponentials (`<g^b|g^a>`) associated with this run, one of which the `Server` himself has just created specifically for this run, his/her signature is tied to this run of the protocol. By encrypting his/her signature with `K`, the `Client` demonstrates to `Server` that he/she was the party who created `g^a`. This gives `Server` assurance that the party he/she carried the key exchange out with was, in fact, the `Client`. The `Client` gets a similar set of assurances from the `Server`: because the `Server` has signed the particular exponentials (`<g^a|g^b>`) associated with this run, one of which the `Client` himself/herself has just created specifically for this run, his signature is tied to this run of the protocol. By encrypting his signature with `K`, `Server` demonstrates to the `Client` that he was the party who created `g^b`.

Because the parties demonstrated knowledge of the exchanged key by encrypting their signatures, the authentication is **direct**. Moreover, the protocol also offers **perfect forward secrecy**. The only long-term secret keying material stored by users is their private keys for the signature scheme. If a secret key is compromised, the security of exchanged keys from earlier runs is not affected because Diffie-Hellman key exchange is used.

In terms of **protection against replay attacks**, we used a `counter`. Every time a message is sent across the channel to the other side, the counter is incremented. The receiver checks for replay attacks by comparing the received counter value with its own. If the check is passed then he/she updates its value of the counter to the one that he received. Given the fact that we used a counter there is the possibility of a **wrap around** of the value: the wrap around is checked every time we increment. In the case of a replay or wrap around we simply close the socket and delete session's resources so that a new run of the session establishment protocol must be completed. This following scheme don't perform an auto re-initialization because of the fact that a wrap around is a unlikely event to happen given the average number of interactions within a session between sever and client.

3.2 Session protocol

To encrypt/decrypt each messaged exchanged within a session, we decide to use `AES` in `CBC` mode. Using such a block cipher we are able to provide **confidentiality**. To ensure **integrity** and **non-malleability** we decide to adopt an `HMAC` based approach. The scheme used to generate the `HMAC` is described in Figure 3.2.

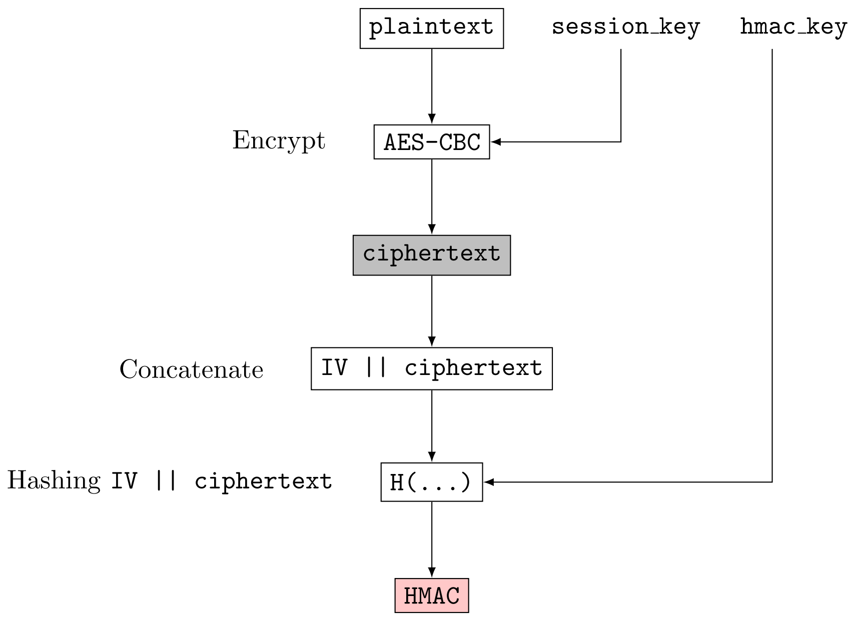


Figure 3.2: Procedure to generate HMAC.

Chapter 4

Application Level Protocols

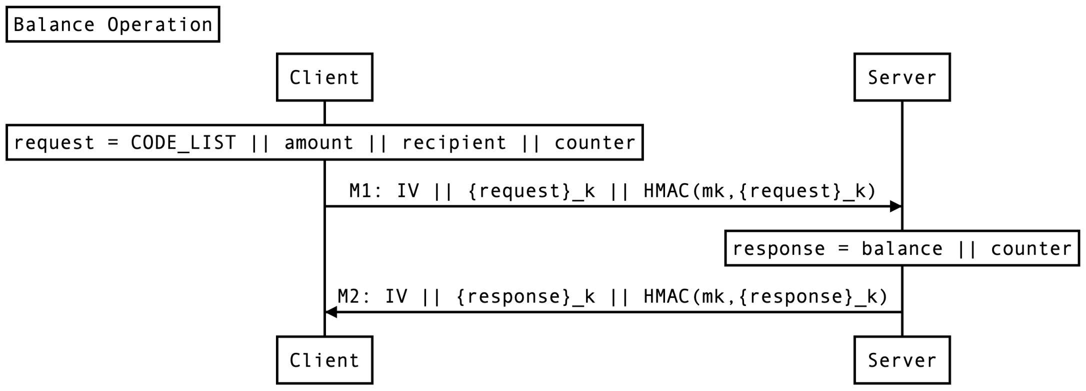

Balance Operation
Client
Server
request = CODE_LIST || amount || recipient || counter
M1: IV || {request}_k || HMAC(mk,{request}_k)
response = balance || counter
M2: IV || {response}_k || HMAC(mk,{response}_k)
Client
Server

Figure 4.1: Balance() operation diagram.

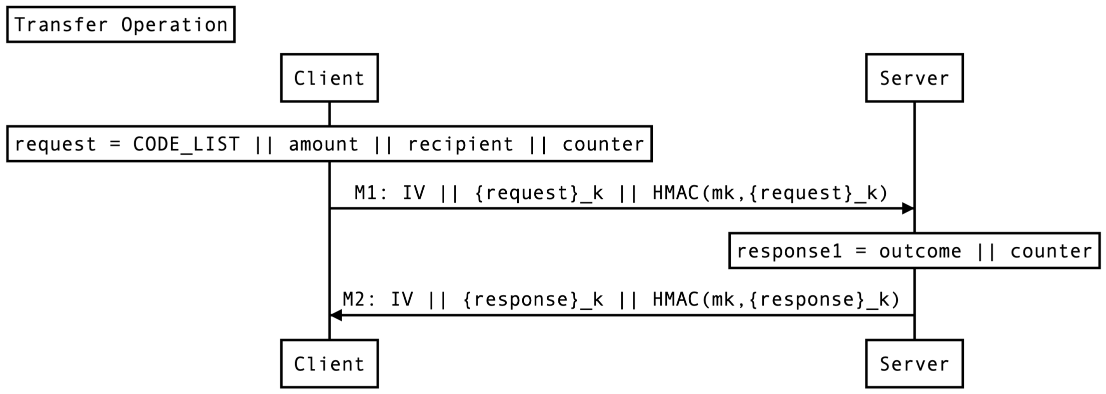

Transfer Operation
Client
Server
request = CODE_LIST || amount || recipient || counter
M1: IV || {request}_k || HMAC(mk,{request}_k)
response1 = outcome || counter
M2: IV || {response}_k || HMAC(mk,{response}_k)
Client
Server

Figure 4.2: Transfer(amount, recipient) operation diagram.

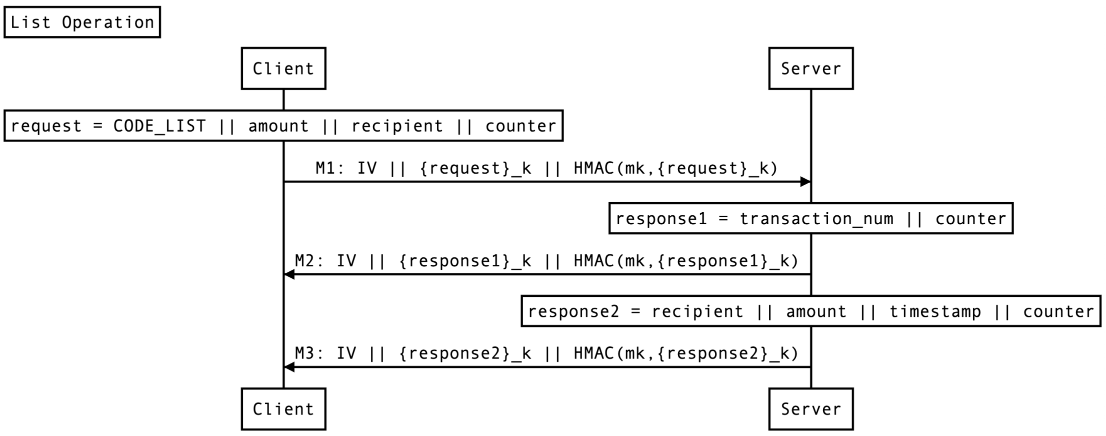

List Operation
Client
Server
request = CODE_LIST || amount || recipient || counter
M1: IV || {request}_k || HMAC(mk,{request}_k)
response1 = transaction_num || counter
M2: IV || {response1}_k || HMAC(mk,{response1}_k)
response2 = recipient || amount || timestamp || counter
M3: IV || {response2}_k || HMAC(mk,{response2}_k)
Client
Server

Figure 4.3: ListOfTransfers() operation diagram.

REQUEST_CODE (2 bytes)	
amount (4 bytes)	
recipient (32 bytes)	
counter (4 bytes)	

Figure 4.4: Client Request message structure.

balance (4 bytes)
counter (4 bytes)

Figure 4.5: Server balance response message structure.

outcome (1 byte)
counter (4 bytes)

Figure 4.6: Server transfer response message structure.

transaction_num (4 bytes)
counter (4 bytes)

Figure 4.7: Server List 1*st* response message structure.

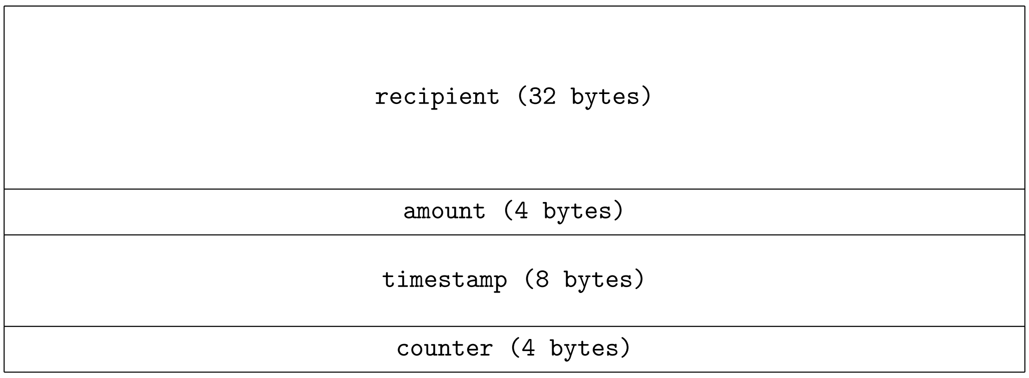


Figure 4.8: Server List $2nd$ response message structure.

4.1 File data structures

4.1.1 Account

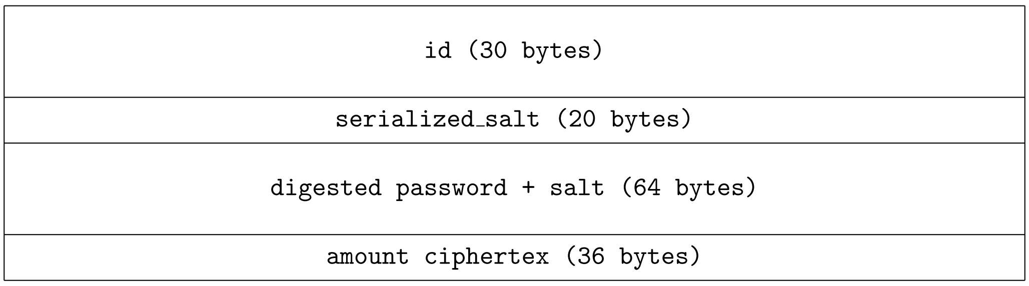

id (30 bytes)
serialized_salt (20 bytes)
digested password + salt (64 bytes)
amount ciphertex (36 bytes)

Figure 4.9: Account file data structure.

4.1.2 Transfers

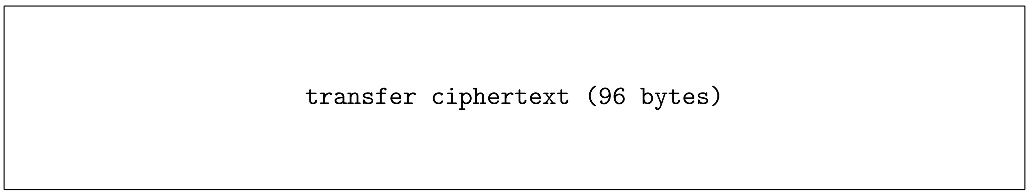


Figure 4.10: Transfers file data structure.