UNIVERSITÀ DI PISA

Foundations of Cybersecurity

Project Work: Cloud Storage

Francesco Bruno

Marco Imbelli Cai

Nicolò Picchi

Dipartimento di Ingegneria dell'Informazione

Università di Pisa

Master degree in Computer Engineering, course of Foundations Of Cybersecurity

Table of contents

Abstract

The produced software implements a Client-Server Cloud Storage, in which each user has a dedicated storage on the server, and in which each user can access only his/her files.

The operations that a user can perform are the operations of Upload, Download, Rename, Delete and LogOut.

Since the exchanged messages between clients and server needed to be confidential and authenticated, the communication is developed in two phases: the first one is the key establishment phase, in which the client that wants to use the application negotiates with the server a shared key (which will be used for further exchanges). The protocol used in this phase guarantees the requirement of Perfect Forward Secrecy.

The second phase is the actual use phase, in which the client and the server use the previously exchanged key to guarantee the requirements of the communication.

After a client inputs a LogOut command, the connection gets closed in a secure way.

Introduction

The produced software implements a Cloud Storage application based on the Client-Server architecture. Some assumptions/simplifications were made about the cryptographic material that each client and the server had before the start of the program. First of all, we assumed that each client already had the digital certificate of the Certification Authority, used to validate the certificate of the server. Moreover, a client has also available a pair of long-term RSA keys (a public one and a private one).

The server instead owns its own digital certificate, signed by the Certification Authority, already knows the username of each registered user (there was no need of implementing a register/login feature). Finally, it has the RSA public keys of the registered users already stored.

The architecture used is a Client-Server architecture: the multi-tasking server starts and waits for one/more clients to connect, and (through forking) serves the requests of each client.

When the client application starts, first of all the two parties, Server and Client, must authenticate to each other. During the authentication phase, a symmetric session key is negotiated in such a way that the protocol guarantees Perfect Forward Secrecy (if by any chance the long-term secret keying material gets compromised, this doesn't affect the secrecy of the exchanged messages from earlier runs).

Once the session key has been negotiated, the client can finally begin to use the application. The available operations are the following:

- Upload: the client specifies a file on the machine and sends it to the server, which will save it on the user's dedicated storage. The server is able to store files of a size up to 4GB.
- Download: the client specifies a file on the server machine, which sends it to him.
- Delete: the client specifies a file on the server machine. The server will delete the file from the client's storage.
- List: the server responds to the client with a list of the files stored in his personal storage.
- Rename: the client specifies a file on the server machine and the new file name that the server has to rename the specified file with.
- LogOut: the client closes the connection with the server.

Programming language, APIs and environment used

The software has been entirely developed using the C++ programming language.

The cryptographic operations have been possible thanks to the use of the OpenSSL APIs. The OpenSSL libraries provide several cryptographic APIs that can be used in C/C++ projects for cryptography purposes.

The certificate generation simulation and export has been performed with the SimpleAuthority applicative, on Windows.

Most of the coding phase has been done on the Visual Studio Code IDE and on Gedit, on the Ubuntu 18.04 OS.

Design choices

Authenticated encryption

The decision of which encryption and authentication algorithm was going to be used was based on the pros and the cons of two of the most widely used algorithms in these fields: 128-bit AES in CBC (Cipher Block Chaining) mode with HMAC and 128-bit AES in GCM (Galois Counter Mode) mode.

After carefully evaluating the different aspects of the two modes, the GCM mode was chosen for various reasons. Since the message exchanging between the client and the server had been structured in such a way that each message had always to be both confidential and intact (its integrity had to be guaranteed), using an algorithm that performs both those operations (by means of encryption and by using a TAG) looked like a good design choice.

The decision was also based on the easier writing and maintainability of the code, as well as the better performance of the GCM mode.

The choice of GCM over CBC with HMAC comes however also with its main disadvantage, i.e., the use of a unique key (used for both confidentiality and integrity): having two separate keys, typical aspect of the algorithms that guarantee the confidentiality and integrity mechanisms separately, is an advantage in cases of a single key compromission (scenario that leads only to the compromission of one of the two mechanisms).

Transport layer protocol

The choice for which transport layer protocol was going to be used ended up with the choice of the TCP protocol. Comparing the two possible protocol choices, TCP or UDP, the conclusion is that even if UDP is undoubtably faster than TCP, this last protocol guarantees the delivery of every packet that was sent by a peer (necessary condition in a file transferring application). Moreover, it guarantees that each packet will arrive in the same order that it was initially sent and, most importantly, intact.

This last property of the TCP protocol makes possible the assumption that if, for any reason, a packet arrives not intact to its destination, with high probability an attacker could be the cause of such integrity issue.

Canonicalization and Sanification

Under the assumption that each client could only have permission to access his personal folder, implementing a canonicalization mechanism was not necessary.

On the other hand, implementing a sanification mechanism was almost mandatory, as a malicious user could deliberately input dangerous filenames to move out of his personal folder and position himself on some other user's, possibly reading his data and/or damaging it. The server takes care of this task and reports an error message to the peer who specified an invalid filename.

Replay attacks and countermeasures

To avoid the possibility of replay attacks, a slightly different method than the one adopted in the key establishment phase was implemented.

In the above cited phase the countermeasure for this kind of attacks has been done by means of two random nonces, one for the client and one for the server. Since this phase is very critical and consists of a small number of messages, a less predictable methods which needs continuous synchronization was preferrable.

During the actual use phase, instead, quite a big number of messages can be exchanged by a user and the server before the session ends. The countermeasure for the possibility of replay attacks in this phase consists of keeping two counters, one for the client and one for the server, that are kept synchronized locally: the counters are incremented and checked by the two parties, that can in this way detect any possible replayed message.

Two options were available at this point: the first one was sending in the clear the counter of each message, and the second one was instead just adding it into the AAD (Additional Authenticated Data) of GCM to implement the TAG. The first solution saves bandwidth, because each party would have to send a parameter less in each exchanged message, but in case of a replay attack the peer would notice the attack at the end of the message decryption (resulting in a useless decryption overhead). The decision landed on the second option: thanks to a quick check of the counters before the decryption phase of the message, the receiving peer can understand right away if he is being the victim of a replay attack and discard the message if so, with no decryption overhead involved. This implementation comes of course with a cost on the bandwidth, as with every exchanged message each peer needs to also include his counter.

Length of the messages

The exchanged messages all have a fixed length of 2048 bytes (specified in the DIMMAX define), except from the messages involved in the upload and download operations. This approach on one hand clearly causes a waste of bandwidth, as a message with a content much smaller than 2048 bytes will result in being 2048 bytes long anyway, but on the other hand allows the peers to exchange fewer messages and with a fixed length and is so a safer method even if a little more restrictive.

The size of the messages was chosen by averaging the actual size of the messages exchanged between client and server. The messages involved in the operations of upload and download were excluded from the calculations, since they can operate with files up to 4GB big.

These two operations are done by splitting the target file into chunks of 1MB of dimension, and by sending each one singularly and sequentially.

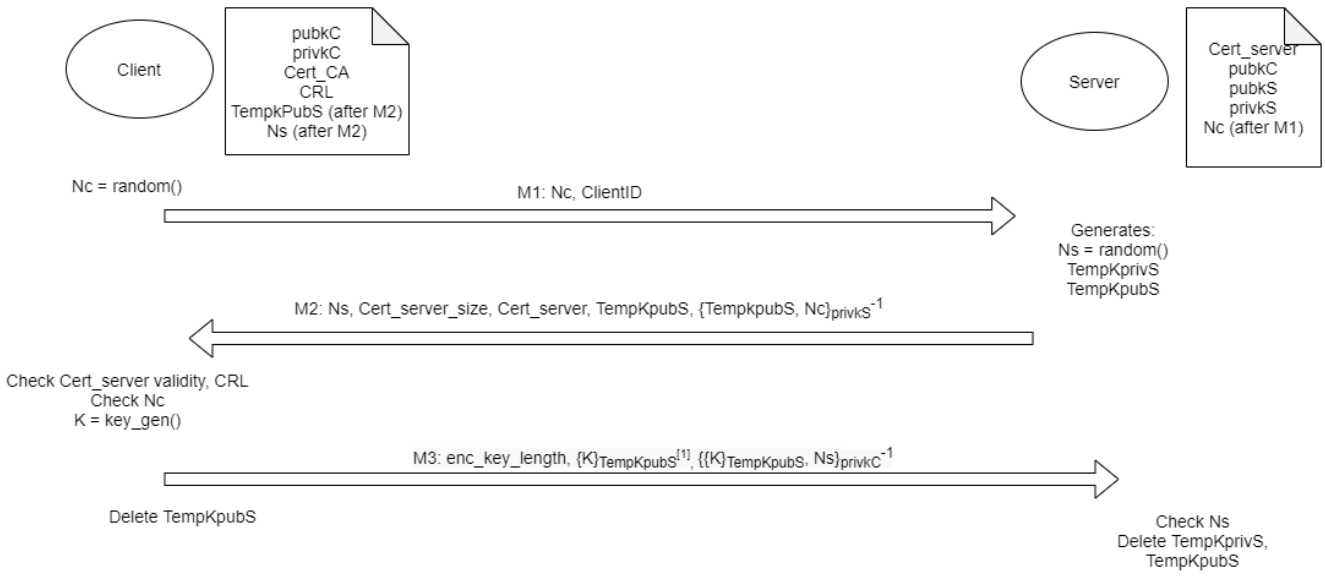
Protocols and message formats

The used protocols and the formats of the exchanged messages can be found in the following section.

The format of the messages has been written adopting the BAN logic syntax, where:

- Mi: refers to the i-th exchanged message, towards the sense of the arrow
- $\{x\}_k$: means that the quantity x has been encrypted by means of a symmetric key k
- $\{x\}_k^{-1}$: means that the quantity x has been digitally signed by means of k

Key establishment protocol and format of the messages

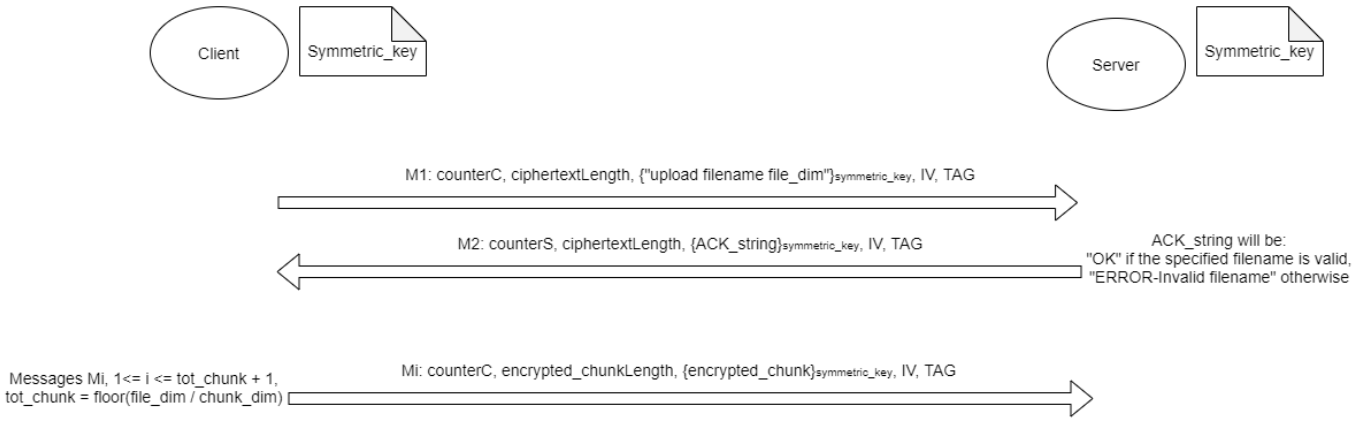


Notes:

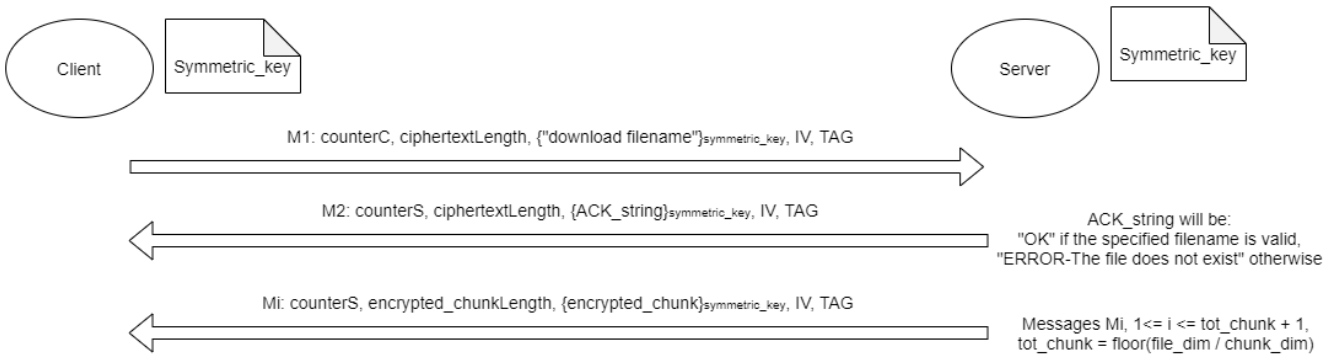
- The key establishment protocol has been developed by starting from the RSAE (Ephemeral RSA) protocol
- In M3, the session key K is transmitted using the digital envelope methodology. The used initialization vector, which has to be known by the server in order to obtain such key, is also transmitted but not shown in the diagram

User’s operations protocols and format of the messages

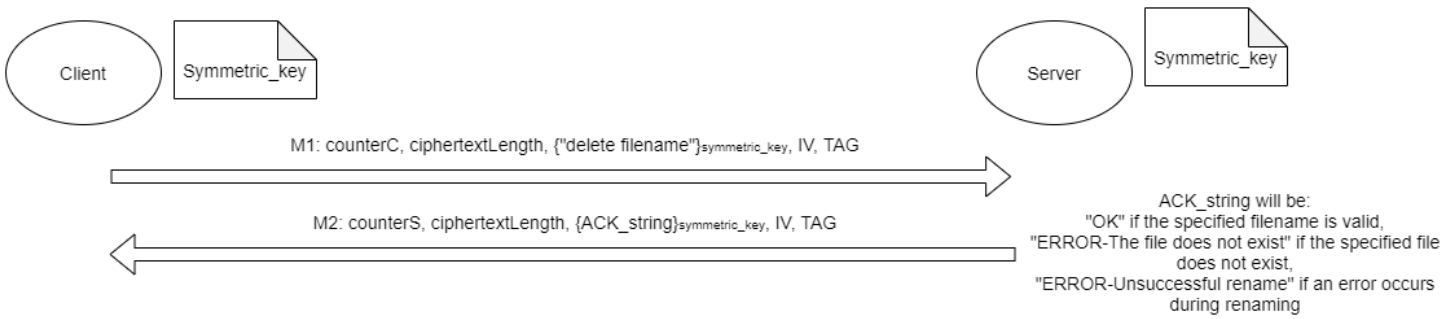
Upload



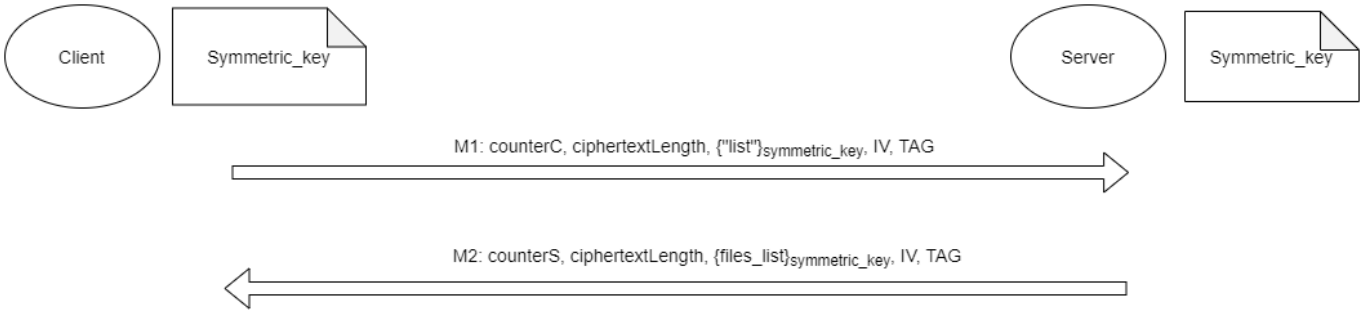
Download



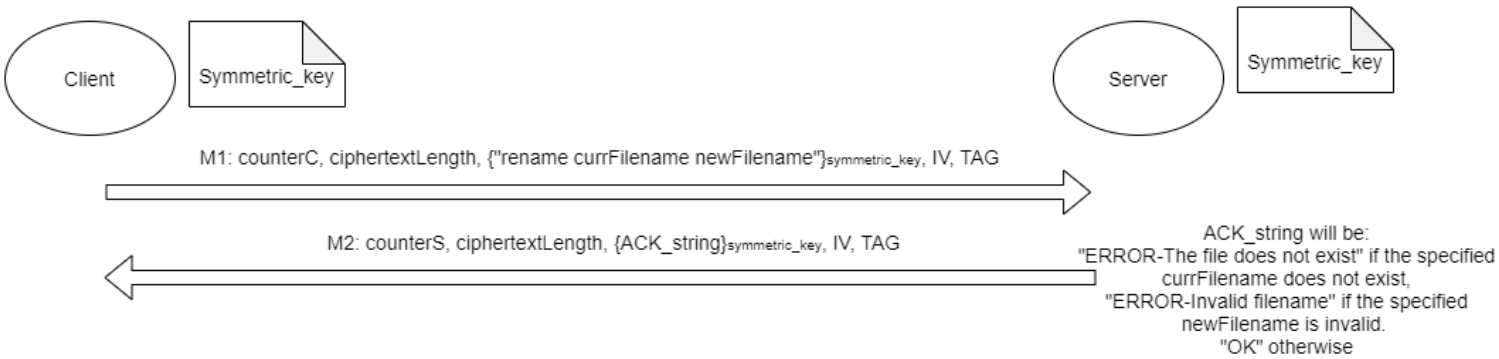
Delete



List



Rename



LogOut