Documentation for the Project of Applied Cryptography

UNIVERSITÀ DI PISA

Students

Simone Bensi

Francesco Venturini

Professors

Prof. *Gianluca Dini*

Eng. *Mariano Basile*

Table of Contents

Introduction

This is the documentation for the project of the course in Applied Cryptography of the M.Sc. in Cybersecurity of the University of Pisa.

The implementation is available on GitHub at:

https://github.com/Portgas97/Secure-Cloud

https://github.com/xux22/Secure-Cloud

The purpose is to develop an application that resembles a Cloud storage service in which each user has its dedicated storage and cannot access the files of the other users.

The focus is on secure coding (C++) using standard best practices and on implementing a secure cryptographic protocol to establish a session key guaranteeing the Perfect Forward Secrecy property.

This document explains how all the communications implemented are formatted, presenting first the general packet exchange that happens in all the possible operations and then detailing the specific packet formatting of each message.

Handshake

Handshake Protocol

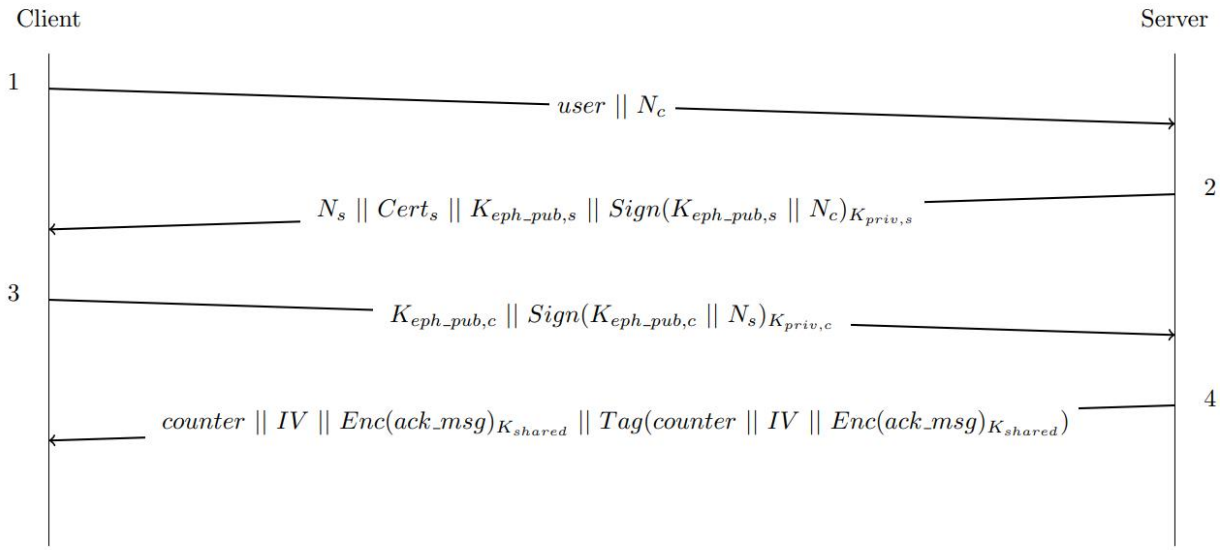

Figure 1: Perfect Forward Secrecy Handshake

Where:

- $user$: username
- N_*: peer's nonce
- $\mathrm{Cert_s}$: server X509 certificate
- $\mathrm{K_{\{eph\_pub,*\}}}$: ephemeral public key of the peer
- $\mathrm{Sign(\,)_{\{K_{\{priv,*\}}\}}}$: DSA executed with peer's private key
- $\mathrm{K_{\{priv,*\}}}$: peer's RSA private key
- counter: message counter
- IV: initialization vector
- $\mathrm{K_{\{shared\}}}$: derived shared key
- Enc( ): AES GCM 128 encryption routine
- ack_msg: ACK message string
- Tag( ): AES GCM 128 digest routine

Handshake Messages Format

Message 1

$username\_size$	$username$	$client\_nonce\_size$	$client\_nonce$

Where:

- $username\_size$: unsigned integer
- $username$: array of characters of size at most of $MAX\_USERNAME\_SIZE$
- $client\_nonce\_size$: unsigned integer
- $client\_nonce$: 16 random bytes

Message 2

$server\_nonce\_size$	$server\_nonce$	$server\_certificate\_size$	$server\_certificate$
$ephemeral\_public\_key\_size$	$ephemeral\_public\_key$	$signature\_size$	$signature$

Where:

- $server\_nonce\_size$: unsigned integer
- $server\_nonce$: 16 random bytes
- $server\_certificate\_size$: unsigned integer
- $server\_certificate$: X509 certificate
- $ephemeral\_public\_key\_size$: unsigned integer
- $ephemeral\_public\_key$: based on $NID\_X9\_62\_prime256v1$ elliptic curve
- $signature\_size$: unsigned integer
- $signature$: created with $sha256$ using the server RSA private key on the concatenation of the server ephemeral public key and the client nonce

When the client receives this message he is able to derive the shared (session) key that the client and the server will use during next communications.

Message 3

$ephemeral\_public\_key\_size$	$ephemeral\_public\_key$	$signature\_size$	$signature$

Where:

- $ephemeral\_public\_key\_size$: unsigned integer
- $ephemeral\_public\_key$: based on $NID\_X9\_62\_prime256v1$ elliptic curve
- $signature\_size$: unsigned integer
- $signature$: created with $sha256$ using the client RSA private key on the concatenation of the client ephemeral public key and the client nonce

Like in the previous case, here the server receives the message containing the ephemeral public key of the client, letting him derive the shared key that will be used during next communications.

Message 4

$message\_counter$	$initialization\_vector\_size$	$initialization\_vector$	$ciphertext\_size$
$ciphertext$		tag_size	tag

Where:

- $message\_counter$: works as a sequence number, is part of the AAD
- $initialization\_vector\_size$: unsigned integer
- $initialization\_vector$: 12 random bytes, is part of the AAD
- $ciphertext\_size$: unsigned integer
- $ciphertext$: result of the application of AES 128 GCM on the “ACK” string
- tag_size: unsigned integer
- tag: result of the application of AES 128 GCM on the “ACK” string and the AAD

Operations

The format is the same for all the messages of the communication, in particular each message exchanged between client and server is encrypted by means of Authenticated Encryption with Associated Data (AEAD), exploiting the shared key negotiated during the handshake phase. In addition to the ciphertext, each message contains also the Additional Authenticated Data (AAD) and the tag.

General Operation Message Format

$message\_counter$	$initialization\_vector\_size$	$initialization\_vector$	$ciphertext\_size$
$ciphertext$		tag_size	tag

Where:

- $message\_counter$: works as a sequence number, is part of the AAD. This value has to be the same both on client and on server side
- $initialization\_vector\_size$: unsigned integer
- $initialization\_vector$: 12 random bytes, is part of the AAD
- $ciphertext\_size$: unsigned integer
- $ciphertext$: result of the application of AES 128 GCM on the plaintext string, which varies for each operation
- tag_size: unsigned integer
- tag: result of the application of AES 128 GCM on the plaintext string and the AAD

UPLOAD Operation

Upload Protocol

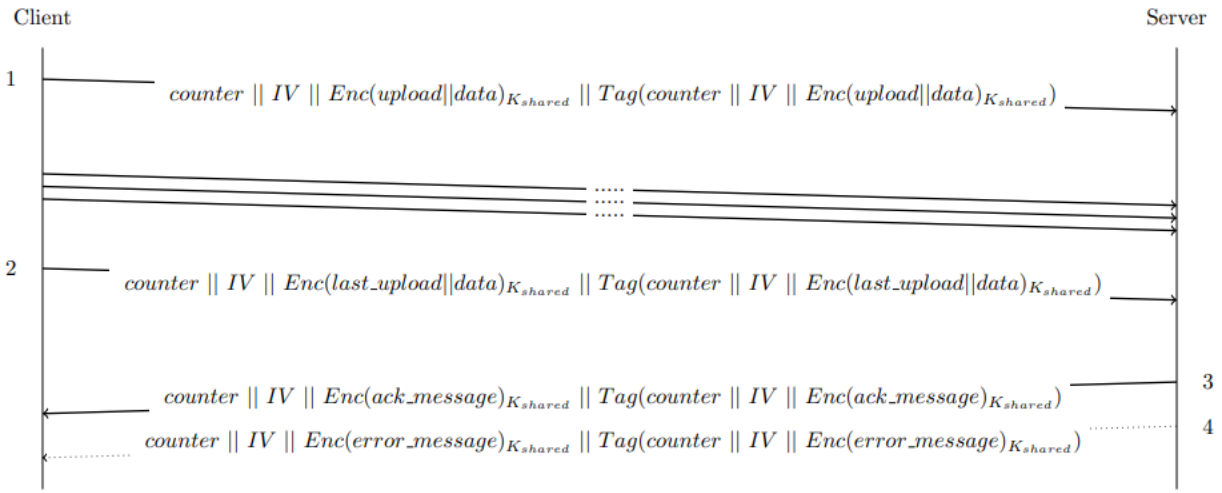

Figure 2: authenticated and encrypted upload operation

- Message 1: This is an optional message sent only when the file content has size larger then CHUNK_SIZE, so it has to be fragmented in multiple chunks having size at most of CHUNK_SIZE. The client asks for an upload operation passing as plaintext the concatenation of the following values: the UPLOAD_MESSAGE string, a space character, the string representing the filename, a space character and the actual content of the file. The filename is sent only in the first upload message sent

- Message 2: it allows to understand the last transmission performed by the client. The client asks for a last upload operation passing as plaintext the concatenation of the following values: the LAST_UPLOAD_MESSAGE string, a space character and the actual content of the file. The filename is sent only if the last upload message sent is also the first upload message sent

- Message 3: this is an optional message sent in case of success. The server sends back an ack passing as plaintext the ACK_MESSAGE string

- Message 4: this is an optional message sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string

DOWNLOAD Operation

Download Protocol

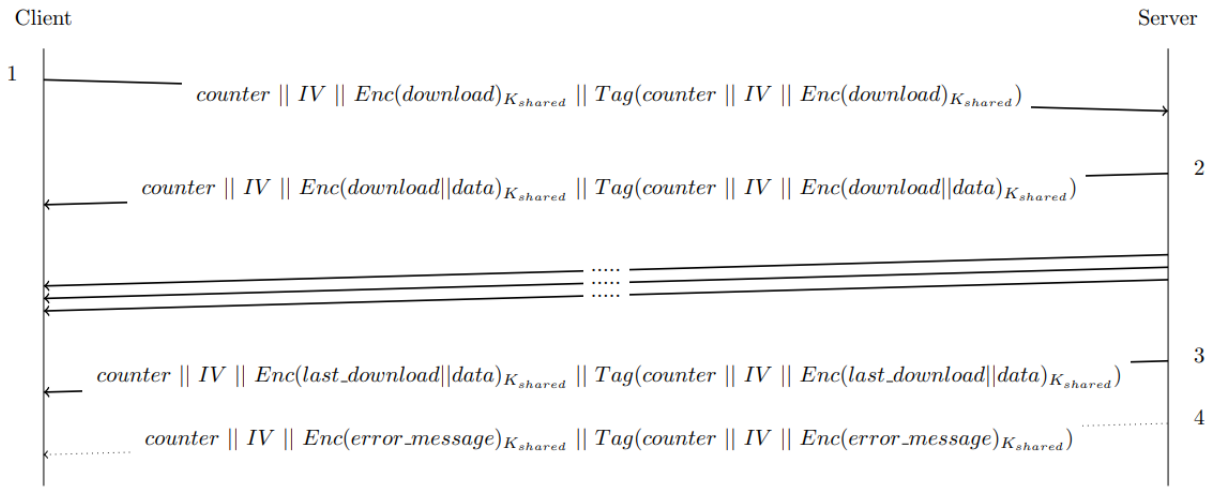

Figure 3: authenticated and encrypted download operation

- Message 1: the client asks for a download operation passing as plaintext the DOWNLOAD_MESSAGE string

- Message 2: this is an optional message sent only when the file content has size larger then CHUNK_SIZE, so it has to be fragmented in multiple chunks having size at most of CHUNK_SIZE. The server sends back the content of the selected file, the plaintext is the concatenation of the following values: the DOWNLOAD_MESSAGE string, a space character, the string representing the filename, a space character, and the actual content of the file. The filename is sent only in the first download message sent

- Message 3: it allows to understand the last transmission performed by the server. The server sends back the content of the selected file, the plaintext is the concatenation of the following values: the LAST_DOWNLOAD_MESSAGE string, a space character and the actual content of the file. The filename is sent only if the last download message sent is also the first download message sent

- Message 4: this is an optional message sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string

RENAME-LOGOUT Operation

Rename-Logout Protocol

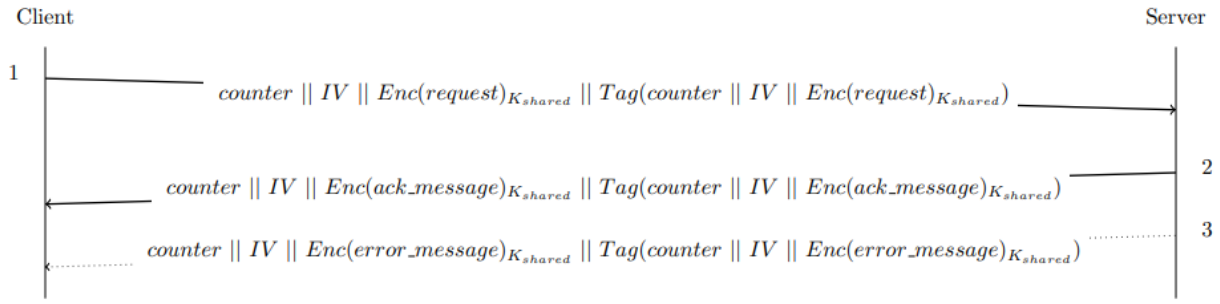

Figure 4: common authenticated and encrypted protocol for the rename, delete and logout operations

RENAME Operation

- Message 1: the client asks for a rename operation passing as plaintext is the concatenation of the following values: the RENAME_MESSAGE string, a space character, the string representing the original filename of the file to rename, a space character, and the string representing the new filename

- Message 2: this is an optional message sent in case of success. The server sends back an ack passing as plaintext the ACK_MESSAGE string

- Message 3: this is an optional message sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string

LOGOUT Operation

- Message 1: the client asks for a logout operation passing as plaintext the LOGOUT_MESSAGE string

- Message 2: this is an optional message sent in case of success. The server sends back an ack passing as plaintext the ACK_MESSAGE string

- Message 3: this is an optional message sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string.

LIST Operation

List Protocol

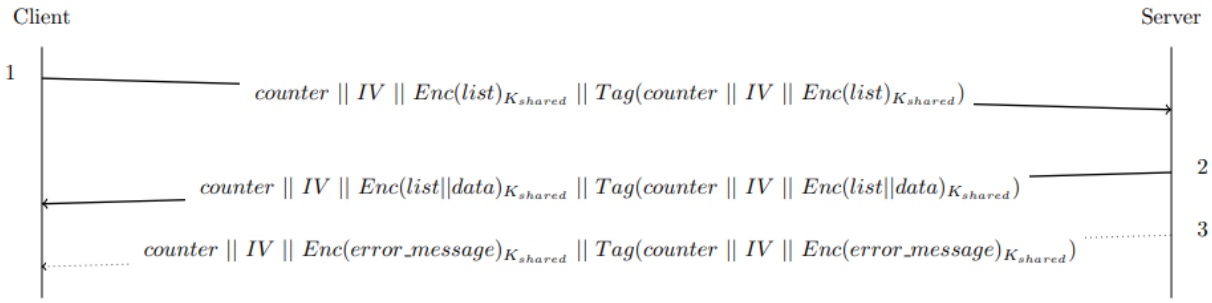

Figure 5: authenticated and encrypted list operation

- Message 1: the client asks for a list operation passing as plaintext the LIST_MESSAGE string

- Message 2: this is an optional message sent in case of success. The server sends back the filenames list passing as plaintext the concatenation of the following values: the LIST_MESSAGE string and the filenames list (data)

- Message 3: this is an optional message sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string

DELETE Operation

Delete Protocol

Figure 6: authenticated and encrypted delete operation

- Message 1: the client asks for a delete operation passing as plaintext the DELETE_MESSAGE string
- Message 2: (optional) message sent in case of success. The server sends back a confirmation message passing as plaintext the CONFIRM_MESSAGE string
- Message 3: (optional) sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string.
- Message 4: (optional) message sent in case the client confirms to delete the file. The server sends back an ack passing as plaintext the CONFIRM_MESSAGE string
- Message 5: (optional) message sent in case the client rejects to delete the file. The server sends back an error passing as plaintext the ERROR_MESSAGE string.
- Message 6: (optional) message sent in case of success. The server sends back an ack passing as plaintext the ACK_MESSAGE string
- Message 7: (optional) message sent in case of error. The server sends back an error passing as plaintext the ERROR_MESSAGE string.