SSLP
SSLP Secure to Lucifer Protocol

MSc in Computer Engineering
Cybersecurity

Luigi Leonardi, Matteo Suffredini, Filippo Storniolo

Academic Year 2018/2019

Contents

Abstract

The main goal of SSLP is to get a secure way to exchange files, between Clients and a Server. Security may be a generic word, so in detail this protocol can withstand:

- Replay Attacks
- Man in The Middle Attacks
- Any form of Tampering

Moreover Confidentiality, Authentication of both parties, and non-reputability for any exchanged file are also guaranteed. The way it is achieved is by exploiting Asymmetric, Symmetric Cryptography and HMAC.

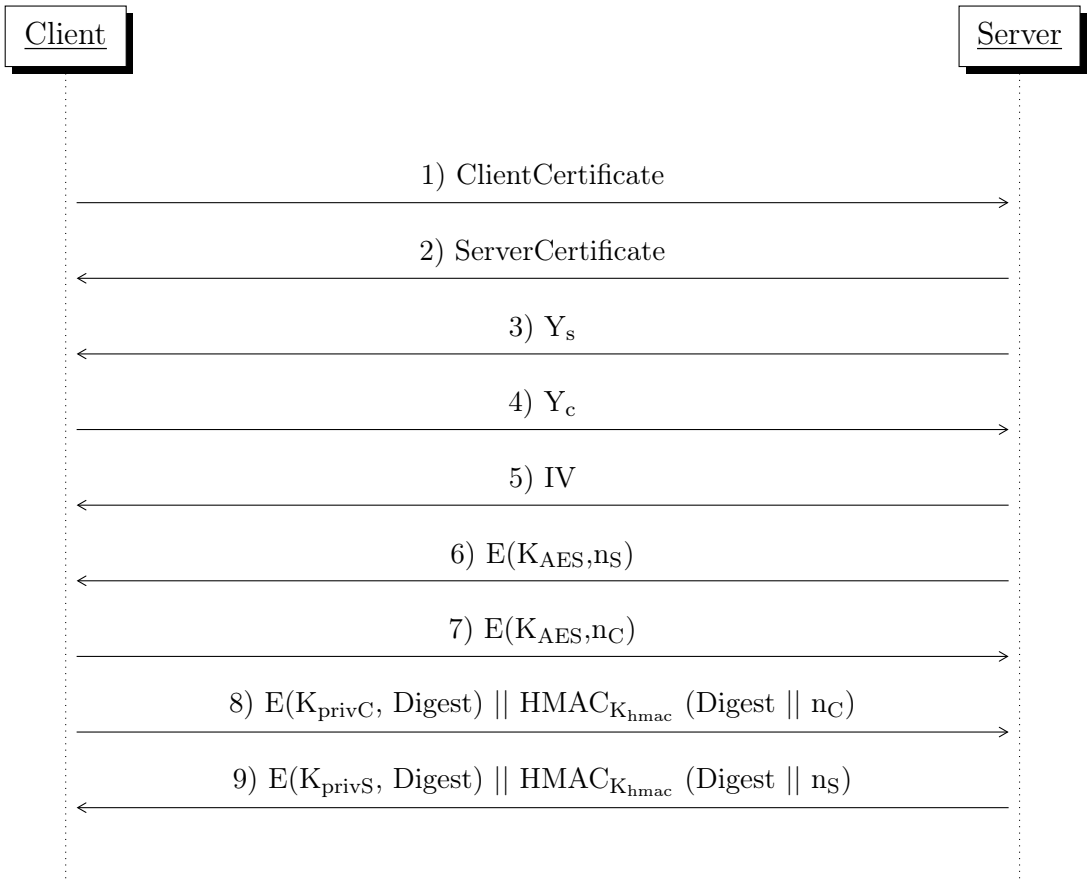
At the beginning of the initial handshake we suppose that the Server already knows the list of the all possible Clients, and both Client and Server have the Certificate and CRL[1] of the Certification Authority (CA).

The chosen schema is an hybrid one, this is due to the the high computational demands of encrypting or decrypting just by means of public and private keys, so it just used for authentication in digital signature.

[1] Certification Revocation List

1 SSLP

1.1 Sequence Diagram



In this figure is shown the initial handshake of the protocol.

1.2 Initial Handshake Description

The initiator is always the client and the first operation is the mutual exchange of Certificates, signed by the CA, followed by the verification that the Client's Common Name, inside the certificate, is in the Server's client list.

After that the Diffie-Hellman key exchange is performed[2], in order to get a symmetric key. The Initialization Vector is sent by the Server.[3]
From now on the communication is encrypted using AES-128.

[2]Using DH two keys are computed, one for HMAC and one for AES
[3]It is supposed to be fresh; CBC is used.

The first two encrypted messages exchanged, are two nonces, that will be used later as an anti-replay measure.

The last two messages are the most important ones:
They contain a Digitally Signed Digest of the entire handshake phase, followed by an HMAC in which nonces are present.

1.2.1 Security

Despite Diffie-Hellman is a good protocol for key exchange, since it provides key freshness and can be used over an insecure channel, but it is vulnerable to the Man In The Middle Attack and it does not provide any kind of authentication. Moreover after receiving a certificates, no-one guarantees that its actually yours.

These two problems are solved in SSLP in the last messages, because the received digest, that is calculated upon all received and sent data, is digitally signed by each party, and no-one should be able to create a DS, apart who owns the key. This way also ensures that the exchanged certificate is actually ours, because it is possible to verify the signature using the key inside the certificate itself.
In addition to that, the freshness of the handshake is ensured by this digest because of the freshness of Y_s and Y_c that are inside the signed digest, so cannot be a replay of an old execution of the protocol.
For the tampering problem an HMAC is attached to the message.

1.3 Communication

After the end of the initial part of the protocol, Server and Client share keys (for encryption and HMAC) and nonces, that were exchanged in a secure way. Before talking about how a single file is sent, first lets talk about how messages are built and handled.
Each message is built in the same way: the content of any message is encrypted using AES-128-cbc and then, on this, an HMAC function is computed also considering the current nonce value. Sending this two values to the counterpart allows the protocol to exchange messages in a secure way: HMAC guarantees integrity, nonce freshness and AES-CBC confidentiality.
The messages can be either a command, sent from the client to the server, or a part of a file, that will be called chunks, from now on.

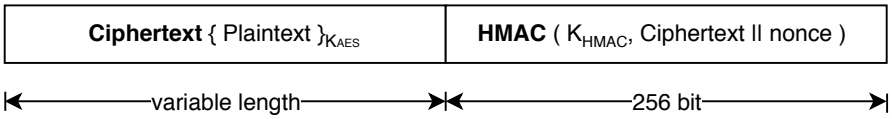


Figure 1: Message Format

1.3.1 File Transfer Description

A single file is seen by the protocol as a set of chunks, each one with size (at most) of 256KiB. The reason of this is due to the fact that the size of a file, can be up to 4GiB, and cannot be (probably) fully loaded in memory, so it must be handled and used in a proper way.

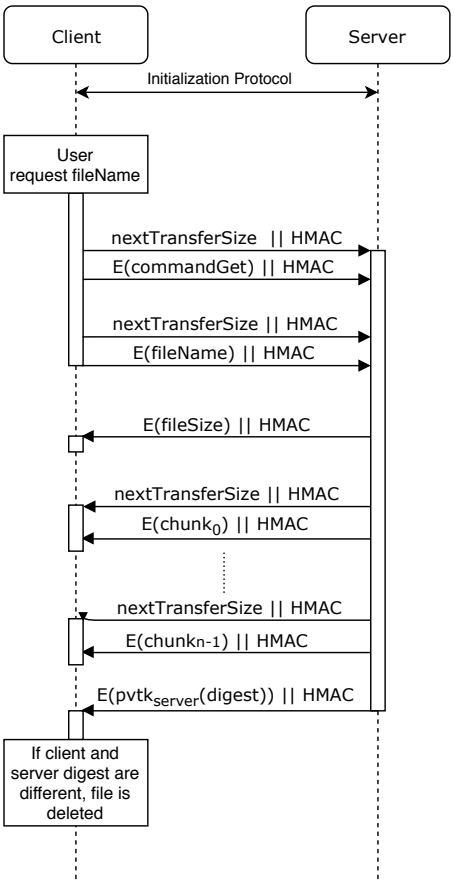


Figure 2: Example of a File Exchange

When a client asks for a file, the server must check the filename format, to avoid any kind of problem, such as the reverse-path attack. After this check, server has

to send to the client the size of the entire file and then it starts to send chunks to client, using the already mentioned format.
When the last chunk is sent, server sends to the client the hash, of the entire encrypted file (ciphertext), signed using its private key, in order to guarantee non repudiation. This message can't be a replay of an old communication, because a nonce is also present in the digest.

1.3.2 File Transfer Security

The reason why the hash function is computed on the ciphertext and not on the plaintext, is due to the fact that an adversary, even though is not able to understand the previous messages, can always read the last one, because is not encrypted but just signed, so it is possible for him to use server's public key and get the digest of the whole file. Even if it doesn't seem to be a problem, an adversary can perform an "oracle attack": he can be sure that a certain file is sent if he knows what file could have been sent, so this way he has some information about what files has been exchanged during the execution.
The tampering and eavesdropping problems are all solved by the use of the already mentioned message format.

2 BAN Logic

2.1 Hypothesis

- $\mathrm{C} \mid\equiv \#(n_c)$
- $\mathrm{S} \mid\equiv \#(n_s)$
- $\mathrm{C} \mid\equiv \overset{K_{\mathrm{publCA}}}{\longmapsto} \mathrm{CA}$
- $\mathrm{S} \mid\equiv \overset{K_{\mathrm{publCA}}}{\longmapsto} \mathrm{CA}$
- $\mathrm{CA} \mid\equiv \overset{K_{\mathrm{pubC}}}{\longmapsto} \mathrm{C}$
- $\mathrm{CA} \mid\equiv \overset{K_{\mathrm{pubS}}}{\longmapsto} \mathrm{S}$
- $\mathrm{S} \mid\equiv \mathrm{CA} \Rightarrow \overset{K_{\mathrm{pubC}}}{\longmapsto} \mathrm{C}$
- $\mathrm{C} \mid\equiv \mathrm{CA} \Rightarrow \overset{K_{\mathrm{pubS}}}{\longmapsto} \mathrm{S}$
- $\mathrm{C} \mid\equiv \#(Y_c)$
- $\mathrm{S} \mid\equiv \#(Y_s)$

2.2 Objectives

- $\mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{C} \mid\equiv \mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{C} \mid\equiv \mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{S} \mid\equiv \mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}$
- $\mathrm{S} \mid\equiv \mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S}$

2.3 Idealized Protocol

- M1: $\mathrm{C} \rightarrow \mathrm{S}\ \{K_{\mathrm{pubC}}\}_{\mathrm{K_{privCA}}}$
- M2: $\mathrm{S} \rightarrow \mathrm{C}\ \{K_{\mathrm{pubS}}\}_{\mathrm{K_{privCA}}}$
- M3: $\mathrm{S} \rightarrow \mathrm{C}\ (Y_{\mathrm{s}})$
- M4: $\mathrm{C} \rightarrow \mathrm{S}\ (Y_{\mathrm{c}})$
- M5: $\mathrm{S} \rightarrow \mathrm{C}\ (IV)$
- M6: $\mathrm{S} \rightarrow \mathrm{C}\ \{n_s, \mathrm{S} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{C}\}_{K_{\mathrm{AES}}}$
- M7: $\mathrm{C} \rightarrow \mathrm{S}\ \{n_c, \mathrm{S} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{C}\}_{K_{\mathrm{AES}}}$
- M8: $\mathrm{C} \rightarrow \mathrm{S}\ (\{Digest\}_{K_{\mathrm{privC}}}, \mathrm{HMAC(Digest}\ ||\ n_c), \mathrm{S} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{C})$
- M9: $\mathrm{S} \rightarrow \mathrm{C}\ (\{Digest\}_{K_{\mathrm{privS}}}, \mathrm{HMAC(Digest}\ ||\ n_s), \mathrm{S} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{C})$

2.4 Proof

In message 1 the server receives, from the client, its certificate signed by the CA. Using the *message meaning rule*:

$$\frac{\mathrm{S} \mid\equiv \overset{K_{\mathrm{publCA}}}{\longmapsto} \mathrm{CA}, \mathrm{S} \triangleleft \{K_{\mathrm{pubC}}\}_{K_{\mathrm{privCA}}}}{\mathrm{S} \mid\equiv \mathrm{CA} \mid\sim (K_{\mathrm{pubC}})}$$

This gives us the proof that this certificate was actually signed by a trusted CA. Since a check on the CRL and expiry date of the certificate is performed, we can actually believe that it is fresh.

$$\mathrm{S} \mid\equiv \#(\overset{K_{\mathrm{publC}}}{\longmapsto} \mathrm{C})$$

Using the *nonce verification rule*:

$$\frac{\mathrm{S} \mid\equiv \#(\overset{K_{\mathrm{publC}}}{\longmapsto} \mathrm{C}), \mathrm{CA} \mid\sim (\overset{K_{\mathrm{publC}}}{\longmapsto} \mathrm{C})}{\mathrm{S} \mid\equiv \mathrm{CA} \mid\equiv \overset{K_{\mathrm{publC}}}{\longmapsto} \mathrm{C}}$$

So using the the *jurisdiction rule* the Server is able to believe that this is actually the PubKey of the Client:

$$\frac{\mathrm{S} \mid\equiv \mathrm{CA} \Rightarrow \overset{K_{\mathrm{pubC}}}{\longmapsto} \mathrm{C}, \mathrm{S} \mid\equiv \mathrm{CA} \mid\equiv \overset{K_{\mathrm{pubC}}}{\longmapsto} \mathrm{C}}{\mathrm{S} \mid\equiv \overset{K_{\mathrm{pubC}}}{\longmapsto} \mathrm{C}}$$

Message 2, just like in the previous message, the Client receives the certificate signed by the CA, and extracts the Server's PubKey, due to the check on the expiry and CRL, we can believe in its freshness. Using the *message meaning rule*:

$$\frac{\mathrm{C} \mid\equiv \overset{K_{\mathrm{publCA}}}{\longmapsto} \mathrm{CA}, \mathrm{C} \triangleleft \{K_{\mathrm{pubS}}\}_{K_{\mathrm{privCA}}}}{\mathrm{C} \mid\equiv \mathrm{CA} \mid\sim (K_{\mathrm{pubS}})}$$

$$\mathrm{C} \mid\equiv \#(\overset{K_{\mathrm{publS}}}{\longmapsto} \mathrm{S})$$

Using the *nonce verification rule*:

$$\frac{\mathrm{C} \mid\equiv \#(\overset{K_{\mathrm{publS}}}{\longmapsto} \mathrm{S}), \mathrm{CA} \mid\sim (\overset{K_{\mathrm{publS}}}{\longmapsto} \mathrm{S})}{\mathrm{C} \mid\equiv \mathrm{CA} \mid\equiv \overset{K_{\mathrm{publS}}}{\longmapsto} \mathrm{S}}$$

Using the *jurisdiction rule*:

$$\frac{\mathrm{C} \mid\equiv \mathrm{CA} \Rightarrow \overset{K_{\mathrm{pubS}}}{\longmapsto} \mathrm{S}, \mathrm{C} \mid\equiv \mathrm{CA} \mid\equiv \overset{K_{\mathrm{pubS}}}{\longmapsto} \mathrm{S}}{\mathrm{C} \mid\equiv \overset{K_{\mathrm{pubS}}}{\longmapsto} \mathrm{S}}$$

So far both Client and Server know each other's public key, but they are not (yet) sure that they are truly talking to whom they claim to be.

After message 4, both Client and Server can compute the two session keys independently, $\mathrm{K_{AES}}$ and $\mathrm{K_{MAC}}$, from Yc and Ys.

$$\mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S},\ \mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}$$
$$\mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S},\ \mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S}$$

Yc and Ys are generated by Client and Server respectively, and are fresh by hypothesis. Since the two keys are obtained from these two values, by applying the *freshness postulate*, we have the proof that the session keys are fresh too:

$$\frac{\mathrm{C} \mid\equiv \#(Yc)}{\mathrm{C} \mid\equiv \#(Yc, Ys)}$$

$$\frac{\mathrm{S} \mid\equiv \#(Yc)}{\mathrm{S} \mid\equiv \#(Yc, Ys)}$$

$$\frac{\mathrm{C} \mid\equiv \#(Yc, Ys)}{\mathrm{C} \mid\equiv \#(\mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S})}$$

$$\frac{\mathrm{S} \mid\equiv \#(Yc, Ys)}{\mathrm{S} \mid\equiv \#(\mathrm{C} \overset{K_{\mathrm{MAC}}}{\longleftrightarrow} \mathrm{S})}$$

$$\frac{\mathrm{C} \mid\equiv \#(Yc, Ys)}{\mathrm{C} \mid\equiv \#(\mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S})}$$

$$\frac{\mathrm{S} \mid\equiv \#(Yc, Ys)}{\mathrm{S} \mid\equiv \#(\mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S})}$$

After message 6 client receives a message encrypted by means of $\mathrm{K_{AES}}$ containing a nonce n_s. Using the *message meaning rule*:

$$\frac{\mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}, \mathrm{C} \triangleleft (K_{\mathrm{AES}}, n_S)}{\mathrm{C} \mid\equiv \mathrm{S} \mid\sim (K_{\mathrm{AES}}, n_S)}$$

Using the *nonce verification rule*:

$$\frac{\mathrm{C} \mid\equiv \#(\mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}), \mathrm{C} \mid\equiv \mathrm{S} \mid\sim (K_{\mathrm{AES}})}{\mathrm{C} \mid\equiv \mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}}$$

This way the Client has the confirmation about the AES session key.

After message 7 server receives a message encrypted by means of $\mathrm{K}_{\mathrm{AES}}$ containing a nonce n_c. Using the *message meaning rule*:

$$\frac{\mathrm{S} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}, \mathrm{S} \triangleleft (K_{\mathrm{AES}}, n_C)}{\mathrm{S} \mid\equiv \mathrm{C} \mid\sim (K_{\mathrm{AES}}, n_C)}$$

Using the *nonce verification rule*:

$$\frac{\mathrm{S} \mid\equiv \#(\mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}), \mathrm{S} \mid\equiv \mathrm{C} \mid\sim (K_{\mathrm{AES}})}{\mathrm{S} \mid\equiv \mathrm{C} \mid\equiv \mathrm{C} \overset{K_{\mathrm{AES}}}{\longleftrightarrow} \mathrm{S}}$$

So far has been proven that:

- Keys are fresh
- Certificates are signed by a CA
- Client and Server have the confirmation of the key

It hasn't been proven yet that Client and Server are effectively who they claim to be.

In message 8 the server receives an encrypted digest, of the whole communication, digitally signed by the client. The integrity and freshness, of this message, is guaranteed by the use of HMAC and nonce. Using the *message meaning rule*:

$$\frac{\mathrm{S} \mid\equiv \overset{K_{\mathrm{pubC}}}{\longmapsto} \mathrm{C}, \mathrm{S} \triangleleft \{Digest\}_{K_{\mathrm{privC}}}}{\mathrm{S} \mid\equiv \mathrm{C} \mid\sim (Digest)}$$

This proves three things:

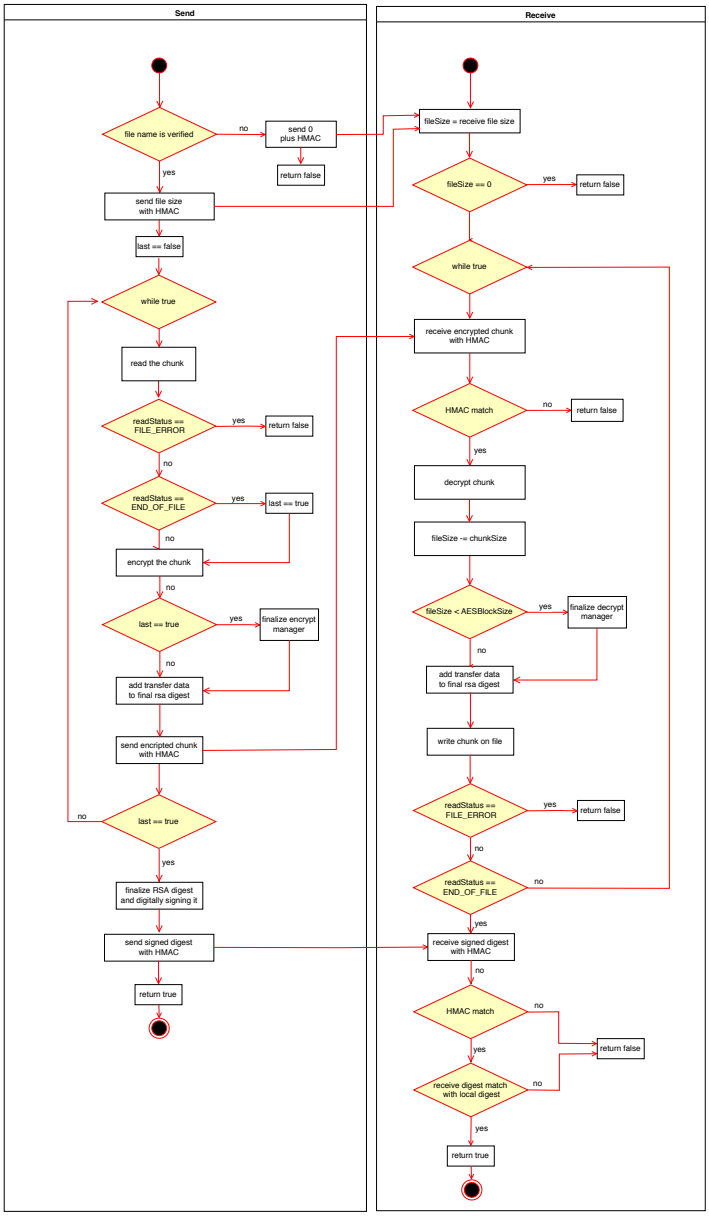
- Client is authenticated: the digest is digitally signed, and this is possible only to whom owns the client's private key and nobody, but the Client itself, should.

- Integrity: The digest is calculated upon all the messages, sent and received during this part of the protocol, so if for any reason a message has been tampered, Client's digest and Server's digest would not match.
- Freshness: Inside the digest there are Yc and Ys that are considered to be fresh, so this can't be a replay of an old execution. Moreover Inside the HMAC there's the nonce that also guarantees the freshness of this message.

In the last message, just like in the previous one, same considerations but from the Client's side: it receives an encrypted digest, of the whole communication, digitally signed by the Server. Using the *message meaning rule*:

$$\frac{\mathrm{C} \mid\equiv \overset{K_{\mathrm{pubS}}}{\longmapsto} \mathrm{S}, \mathrm{C} \triangleleft \{Digest\}_{K_{\mathrm{privS}}}}{\mathrm{C} \mid\equiv \mathrm{S} \mid\sim (Digest)}$$

Appendices


Send
file name is verified
no
send 0 plus HMAC
return false
yes
send file size with HMAC
last == false
while true
read the chunk
readStatus == FILE_ERROR
yes
return false
no
readStatus == END_OF_FILE
yes
last == true
no
encrypt the chunk
no
last == true
yes
finalize encrypt manager
no
add transfer data to final rsa digest
send encripted chunk with HMAC
no
last == true
yes
finalize RSA digest and digitally signing it
send signed digest with HMAC
return true
Receive
fileSize = receive file size
fileSize == 0
yes
return false
while true
receive encrypted chunk with HMAC
HMAC match
no
return false
yes
decrypt chunk
fileSize -= chunkSize
fileSize < AESBlockSize
yes
finalize decrypt manager
no
add transfer data to final rsa digest
write chunk on file
readStatus == FILE_ERROR
yes
return false
no
readStatus == END_OF_FILE
no
yes
receive signed digest with HMAC
no
HMAC match
no
return false
yes
receive digest match with local digest
no
yes
return true