Università di Pisa

A.A. 2018/2019

CyberSecurity

project

Davide Coccomini
Maurizio Pulizzi

INDEX

1 Introduction

The main scope of this project is to realize a secure client-server system that will be used to exchange files. In particular, the following operations have been implemented:

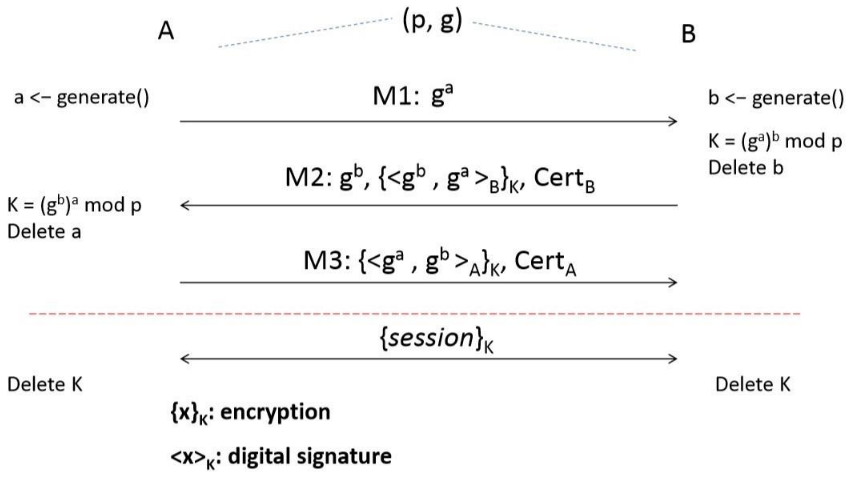
- The client can connect to the server and be authenticated;
- The client can ask for a list of the files on the server and receive a file containing this information;
- The client can download a file from the server to a local directory;
- The client can upload a file from a local directory to the server;
- The server can receive connections from verified clients;
- The server can read the list of the files in its directory and send this information to a client;
- The server can send one of its files to a client;
- The server can receive a new file, sent by the client, and store it.

Both sides are also able to manage error situations maintaining a substantial state.

2 Protocols

2.1 Initial handshake

When a client connects to the server, the two parties will have to authenticate each other and establish the two session keys. To grant mutual key and entity authentication, we have chosen to use the Station-to-Station protocol, a cryptographic key agreement scheme. The protocol is based on Diffie-Hellman but solve the problem of the Man-in-the-Middle attack. STS grant also the perfect forward secrecy and the direct authentication. To implement this protocol, it's needed that both parties have a certificate, which are used to sign messages. This protocol is summarized in the following scheme:



The messages grant the perfect forward secrecy thanks to the creation of a and b, used in Diffie Hellman that are deleted at the end of each session. In this way, an attacker will not be able to reconstruct the message even if he obtains Ya and Yb. Moreover, with this protocol both parties contribute to create the shared key, and this prevents replay attacks of a session. We refer to after a more detailed analysis of this protocol.
Once the shared key is created, the two session keys are derived from it using a hash function. More in detail a digest of the shared key, via sha512, is produced. The first 128 bits of the digest will constitute the session encryption key, the last 256 bits will constitute the session authentication key.

2.2 Communication

After the initial handshake to grant the security on the messages exchanged between client and server, every time one of them wants to send a message will do the following steps:

1. Calculate the cipher text as:
   $AES128(K_{sec}, plainText)$
2. Create the digest as:
   $HMAC(K_{aut}, counter||cipherText)$
3. Send the concatenated text:
   $digest||cipherText$

When the message is received, the following operations are made to extract the information:

1. The received text is splitted obtaining the digest and the cipher text;
2. The digest is checked by recalculating it starting from the received cipher text and comparing the result with the received one;
3. If the digest is correct, the cipher text is decrypted and the plain text is ready to be used;

Using this method to send messages, a received message is authenticated before being decrypted and this prevents Oracle Padding attacks.
Each message is encrypted and decrypted also through an initialization vector. The IV is created starting from a counter. Each time new IV is needed the counter is incremented and the result is given as input to a hash function, in this way the sequence of IVs will appear as a random sequence. Long messages are encrypted (for memory efficiency) in CBC mode.
Data patterns of the plaintext are therefore masked, and it is very difficult to carry out traffic analysis.
Each digest is produced (and checked) concatenating the ciphertext with a counter. Block reordering and block substitution results therefore infeasible; also replay attacks within a session are not possible.

3 Implementation

3.1 Classes

To implement this application two different classes have been developed, one for the client and another for the server.
These classes implement all the methods used from the user to interact with the server and from the server to react to the user requests. There are also the methods used for the initial handshake, implementing the station-to- station protocol, and the ones to certify the user identity.
The server initially loads the certificates, the authorized clients and the CRL, then create the store and start to wait for new connections using the function handleClient. To achieve these objectives the following methods have been implemented:

- loadClients(): load all the authorized clients written in the file authorizedClients.txt;
- socketActivation(): initialize the socket for the communication;
- keySharing(): loads the private key and the session key loading the values of p and g written in the file DH.h. Then starts the protocol station-to-station with the client;
- startToRun(): initializes the values of iv and counter, wait for the client's request and calls for the keySharing() function. After that, calls the function handleClient();
- verifyAndAcquireInput(): validate the received command and extracts the information from it;
- handleClient(): manages the communication with the client receiving the command and react accordingly with what the client is requesting.

From the other side, the client loads the shared keys and is ready to contact the server that is waiting for it. Every time a command is typed, it is validated using the function verifyAndAcquireInput and the information are obtained from it and sent to the server. To achieve these scopes the following methods have been implemented:

- keySharing(): loads the private key and the session key loading the values of p and g written in the file DH.h. After that answer to the message sent from the server and continue the station-to-station protocol;
- startToRun(): manage the commands written by the user and manage the errors;
- helpCommand(): prints the available commands for the client;
- verifyAndAcquireInput(): validates the typed command.

3.2 Utility functions

In the file utilityFunctions.cpp there are the functions useful both to the client and to the server. A subset of this functions is those function used for sending and receiving commands and files. These functions are:

- createDigest(): receives the cipher text and its size and generates a digest using HMAC;
- checkDigest(): receives a digest and a cipher text, generates a new digest starting from the received cipher text and checks if the received digest is equal with the generated one;
- sendSize(): sends to the other side the size of the message that is about to be sent using the explained communication protocol;
- receiveSize(): receives a concatenated text from the other side and extracts the size of the message that is about to be sent, as explained before;
- sendString(): calls to sendSize sending the plain text size and then sends a concatenated text using the explained communication protocol;
- receiveString(): calls to receiveSize receiving the plain text size and then receives the concatenated text sent by the other side. Then extract the plain text as explained before;
- sendFile(): calls to sendSize sending the file size and then sends the file content using a variable number of sends based on the file size;
- receiveFile(): calls to receiveSize receiving the file size and then receives the file blocks concatenating them to obtain the total file;

3.3 File transfer

Particular attention needs to be done about the files. In general, before starting a file transfer from the server to the client or vice versa, both sides calculate the number of chunks of fixed dimension that have to be sent (or received) and then the two sides start to make many sends and receives in each of which, only a little piece of the file is transferred. This specification is needed because of the memory efficiency problem. Indeed, in presence of very big files, it become very difficult (or impossible) to transfer the whole file in one sending. This because to transfer some piece of information, it have to be temporary stored in the RAM and this become very difficult for files of big sizes, due to the hardware limits. With this approach, instead of doing one big transfer, we make many transfers of fixed size (512 bytes), each of which doesn’t fill the available memory.

4 BAN Logic Analysis

4.1 Session Keys exchange protocol

Message 1	$A \rightarrow B$: Ya
Message 2	$B \rightarrow A$: Yb, { <Yb, Ya>$_{B}$ }$_{Kab}$ }, B's certificate
Message 3	$A \rightarrow B$: Ya, { <Ya, Yb>$_{A}$ }$_{Kab}$ }, A's certificate

4.2 Idealized protocol

Message 1	$A \rightarrow B$: Ya
Message 2	$B \rightarrow A$: $Yb, \{ \{Yb, Ya\}_{Kb}{}^{-1} \}_{Kab}, \ \{\vert\text{-}^{Kb}\text{->}B\}_{Kca}{}^{-1}$
Message 3	$A \rightarrow B$: $Ya, \{ \{Ya, Yb\}_{Ka}{}^{-1}\}_{Kab}, \ \{\vert\text{-}^{Ka}\text{ - }A\}_{Kca}{}^{-1}$

4.3 Assumptions

$A \mid\equiv \vert\text{-}^{Kca}\text{->}CA$,	$B \mid\equiv \vert\text{-}^{Kca}\text{->}CA$
$A \mid\equiv CA \Longrightarrow \vert\text{-}^{Kb}\text{->}B$,	$B \mid\equiv CA \Longrightarrow \vert\text{-}^{Ka}\text{->}A$
$A \mid\equiv \#(Ya)$,	$B \mid\equiv \#(Yb)$
$A \mid\equiv \#(a)$,	$B \mid\equiv \#(b)$

4.4 Objectives

- Key authentications: 1) $A \mid\equiv A \text{<-}^{Kab}\text{->} B$, 2) $B \mid\equiv A \text{<-}^{Kab}\text{->} B$
- Key confirmations: 3) $A \mid\equiv B \mid\equiv A \text{<-}^{Kab}\text{->} B$, 4) $B \mid\equiv A \mid\equiv A \text{<-}^{Kab}\text{->} B$
- Key freshness: 5) $A \mid\equiv \#(A \text{<-}^{Kab}\text{->} B)$, 6) $B \mid\equiv \#(A \text{<-}^{Kab}\text{->} B)$

4.5 Objectives verifications

Message 1:

- Assumptions: B ◁ Ya , B |≡ #(b) , B compute Kab = $(Ya)^{b}$
- Assertion: B |≡ #(Kab) *[Key freshness verified for B]*

Message 2:

- Assumptions: A ◁ { |-^{Kb} ->B }$_{Kca}{}^{-1}$, A |≡ |-^{Kca} ->CA
- Assertion: A |≡ CA |~ |-^{Kb} ->B

After decrypting the certificate:

- Assumptions: A |≡ CA |≡ |-^{Kb} ->B, A |≡ CA ==> |-^{Kb} ->B
- Assertion: for *Jurisdiction rule* A |≡ |-^{Kb} ->B

- Assumptions: A ◁ Yb , A |≡ #(a) , A compute Kab = $(Yb)^{a}$
- Assertion: A |≡ #(Kab) *[Key freshness verified for A]*

Decryption of { {Yb, Ya}$_{Kb}{}^{-1}$ }$_{Kab}$:

- Assumptions: A ◁ { {Yb, Ya}$_{Kb}{}^{-1}$ }$_{Kab}$, A |≡ A <-^{Kab}-> B (yet to be verified!)
- Assertion: A ◁ {Yb, Ya}$_{Kb}{}^{-1}$, for *message meaning rule* A |≡ B |~ {Yb, Ya}$_{Kb}{}^{-1}$

Signature verification:

- Assumptions: A ◁ {Yb, Ya}$_{Kb}{}^{-1}$, A |≡ |-^{Kb} ->B
- Assertions: A ◁ (Yb, Ya) , for *message meaning rule* A |≡ B |~ (Yb, Ya)

A |≡ #(Ya) implies that A |≡ #(Yb,Ya).
A |≡ #(Yb,Ya) and A |≡ B |~ (Yb, Ya) implies, for *nonce verification rule*, that
A |≡ B |≡ (Yb, Ya).
A |≡ B |≡ (Yb, Ya) implies that A |≡ A <-^{Kab}-> B and A |≡ B |≡ A <-^{Kab}-> B.
[Key authentication and key confirmation verified for B]

Message 3:

- Assumptions: B ◁ { |-^{Ka} ->A }$_{Kca}^{-1}$, B |≡ |-^{Kca} ->CA
- Assertion: B |≡ CA |~ |-^{Ka} ->A

After decrypting the certificate:

- Assumptions: B |≡ CA |≡ |-^{Ka} ->A, B |≡ CA ==> |-^{Ka} ->A
- Assertion: for *Jurisdiction rule* B |≡ |-^{Ka} ->A

Decryption of { {Yb, Ya}$_{Kb}^{-1}$ }$_{Kab}$:

- Assumptions: B ◁ { {Ya, Yb}$_{Ka}^{-1}$ }$_{Kab}$, B |≡ A <-^{Kab}-> B (yet to be verified!)
- Assertion: B ◁ {Ya, Yb}$_{Ka}^{-1}$, for *message meaning rule* B |≡ A |~ {Ya, Yb}$_{Ka}^{-1}$

Signature verification:

- Assumptions: B ◁ {Ya, Yb}$_{Ka}^{-1}$, B |≡ |-^{Ka} ->A
- Assertions: B ◁ (Ya, Yb) , for *message meaning rule* B |≡ A |~ (Ya, Yb)

B |≡ #(Yb) implies that B |≡ #(Ya, Yb).
B |≡ #(Ya, Yb) and B |≡ A |~(Ya, Yb) implies, for *nonce verification rule*, that
B |≡ A |≡ (Ya, Yb) .
B |≡ A |≡ (Ya, Yb) implies that B |≡ A <-^{Kab}-> B and B |≡ A |≡ A <-^{Kab}-> B.
[*Key* authentication *and key confirmation verified for A*]