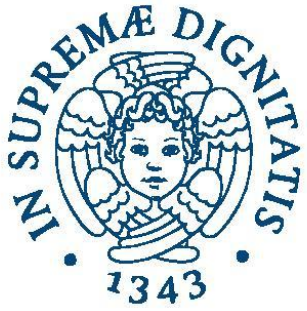


UNIVERSITÀ DI PISA

FOUNDATION OF CYBERSECURITY PROJECT

Encrypted Four in a Row

Gabriele Martino

a.a 2018/19

Chapter 1

Specification

Reported list of specifications of the project.

- Users are already registered on the server through public keyes. Users authenticate through said public key.
- After the log-in, a user can see other available users logged to the server.
- User can send a challenge to another user.
- The user who receives a challenge can either accept or refuse.
- If the challenge is accepted, the users proceed to play using a peer-to-peer communication.
- The game board must be printed at each move. The board is 6x7 (rows, columns).
- When the client application starts, Server and Client must authenticate.
  - Server must authenticate with a public key certified by a certification authority.
  - Client must authenticate with a public key (pre-installed on server). The corresponding private key is protected with a password on each client.
  - After authentication a symmetric session key must be negotiated.
  - The negotiation must provide Perfect Forward Secrecy.
  - All session messages must be encrypted with authenticated encryption mode (e.g., CCM, GCM)
  - Session with server is not interrupted by games.
- After a challenge is accepted, the server sends to both clients the ip address and public key of the adversary.
- Before starting the game a symmetric session key must be negotiated.
  - The negotiation must provide Perfect Forward Secrecy.
  - All session messages must be encrypted with authenticated encryption mode (e.g., CCM, GCM)

- When the game ends, clients disconnect from each other.
- When a client wants to stop playing, it shall log-off from the server
- Use C or C++ language, and OpenSSL library for crypto algorithms.
- Key establishment protocol must establish one (or more) symmetric session key(s) with public-key crypto
- Then, session protocol must use session key(s) to communicate.
- Communication must be confidential, authenticated, and protected against replay.
- No coding vulnerabilities (use secure coding principles)
- Menage malformed messages.
- Project report must contain:
  - Project specification and design choices
  - BAN-Logic proof of key exchange protocol
  - Format of all the exchanged messages

Chapter 2

Introduction

2.1 Specification Analysis

The specifications define how the application evolves and the properties that should be observed. For this reason drawing the different phases for which a client steps into, could help to understand the application. Here after the found phases:

- Client – Server authentication.
- Client's game request and wait. Client - Server Communication.
- Client - Client Authentication.
- Clients in game. Client - Client Communication.

Moreover, after the Client – Server authentication, the peer is able to require the list of logged users, necessary to the subsequent game request. After these considerations we are able to define an FSM (Finite State Machine).

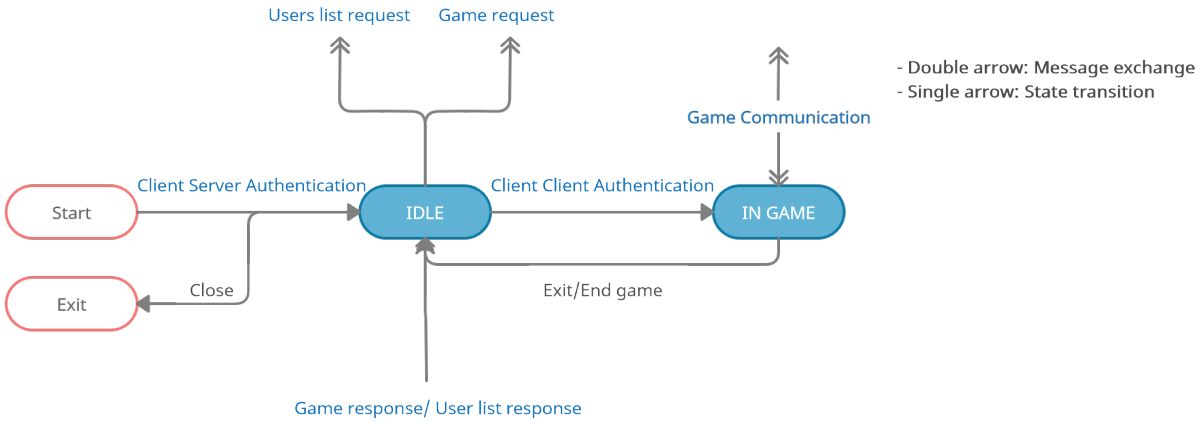


Figure 1: FSM diagram of the states of a Client.

The state transition, i.e. Authentications, and the communications should follow the specifications. The formers need to be double, that is both the peers should authenticate each other; these authentications bring to a key exchange with **Perfect Forward Secrecy (from now on PFS)**; the key confirmation is not specified. The communications should be encrypted with an authenticated mode: CCM or GCM. All these message exchange should be immune to replay attacks.

2.2 On Security Choices

Following the specifications, we will use certificates, public keys and ciphers. These need to be well and carefully dimensioned to cope with the state of the art of the cryptography attacks.
However, we remind that the aim of the project is to develop a game, so no critical real-world data are at the stake. So, we will try to not oversize the cryptography security just for fun.
Regarding the certificates we will use *SimpleAuthority[1]*, a simple application that create Certificates and CRL (Certificate Revocation List). This will be used for the Server Certificate. These certificates are **RSA2048**, nowadays the minimum required for secure signature as reported in [2][3], where has been shown a rapid decaying of 1024bit security. Just for conformance reason, **all the others RSA public keys of the clients, have been created at 2048bit**. Moreover, the respective **private keys have been encrypted with a password with AES 128bit-CBC**, more than enough.
The communications need to be authenticated as well as being encrypted, so **AES 128bit in GCM mode seems a good choice**. GCM allows an elegant way to create an encryption with authentication, it also uses AAD (Additional Authenticated Data), that allows us to protect the entire packet, with headers, identifiers and so on all together. This kind of encryption needs an IV (Initialization Vector). According to whom is the first sender, **the IV is chosen randomically** then the receiver saves it, and from that moment on it will be used incrementally (counting). This approach, as we will see later, avoids replay attacks for all the subsequent messages respect to the first. By the way the first message cannot be created from an attacker since he/she doesn't have the session key previously agreed between the part. After the first message it's possible to avoid sending the IV, but we decided to keep it in every message. This choice because the implementation is simpler, and then because if an attacker tries to forge the IV, we check it before checking the tag of the GCM; this at expense of the length of every message of course.
Given the requirement of PFS, with have to choice between Ephemeral RSA or Ephemeral DH, given the high computational cost of RSA we decided to use **Ephemeral DH with Elliptic Curve**, that allows us a lower number of bits.

Chapter 3

Client Server Authentication

3.1 The protocol

As specified, we have a predefined state of information in which each part starts with.

- The Client owns a Public Key or a Certificate of a Certification Authority.
- The Client owns his Private Key.
- The Server owns the Public Key of every Client already registered.
- The Server owns his Certificate signed by the Certification Authority.

Encryption Algorithms chosen:

- **RSA 2048bit**, key pairs for signatures.
- **ECDH**, for ephemeral key agreement for PFS.
- **SHA-256**, for entropy increasing after key agreement and hashing function.

Here we report the sequence diagram of the double authentication protocol developed.

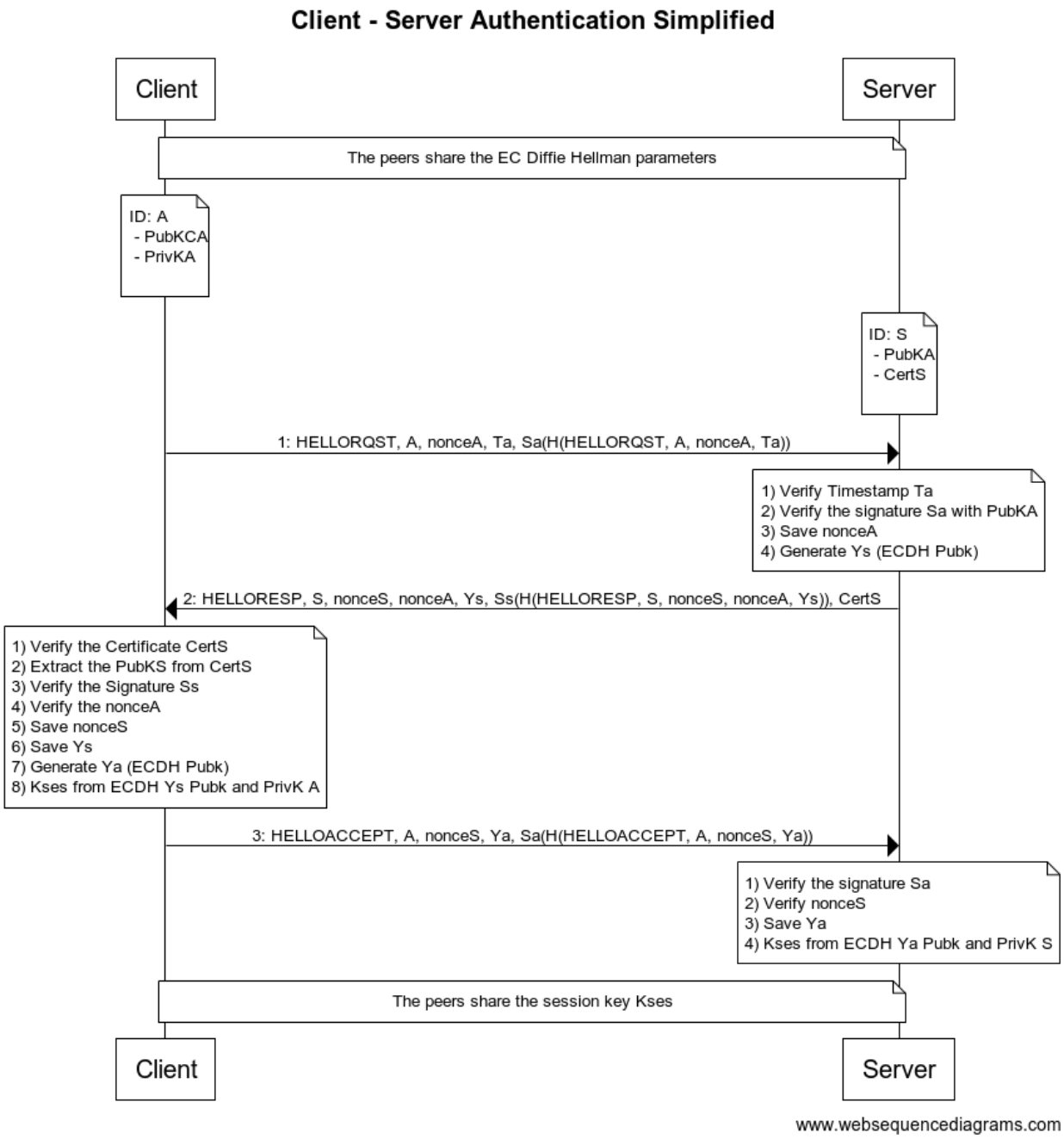


Figure 1: Sequence Diagram: Client-Server Authentication Simplified

We can see that the two peers share the Diffie-Hellman parameters. We used the standard **prime256v1**, that is fixed apriori in the server as in the client, so we don't have to exchange these parameters between the peers.
Every packet contains always the fields HEADER and ID. This two fields represent the name of the message, in order to distinguish it from others, and the name of the sender of the message.

3.1.1 HelloRequest Message

HEADER	ID	NONCE	TIMESTAMP	SIGNATURE
1 byte	30 byte	16 byte	8 byte	256 byte

Figure 2: HELLOREQST message format

The header of the message in this case is *HELLORQST* and the size is just one byte, so we can have at maximum 256 different headers. This convention of the header will be used for all the protocols used in this project. The ID is simply the username of the client who requests to log in over 30 bytes, considered as 30 characters in ASCII. The nonce is a set of random bytes used for just one instance of authentication and necessary to later authenticate the server.
Moreover, we also have a timestamp that indicates the actual time of generation of the packet, that is 8 bytes long. Then the client signs the digest of previous fields with his/her private key, since the pair of public/private key is 2048 bits long, we have 256bytes long signature. The total size of the frame is 311 bytes.

3.1.2 HelloResponse Message

HEADER	ID	NONCE-ID	NONCE-PEER	DH-PUBKEY	SIGNATURE	CERTIFICATE
1 byte	30 bytes	16 bytes	16 bytes	N bytes	256 bytes	P bytes

Figure 3: HELLORESP message format

This time the frame contains two nonce, the first generated from the sender (in this case the server), and the second one is the same received in the HelloRequest message before. Now we also have a DH-Public key generated from the sender. As before all of these fields will be hashed and signed. Now the tail of the frame is occupied by the Certificate of the server signed from the CA. Notice that the DH-Public key has no defined number of bytes. This because the size depends on the kind of encoding used, in our case we use PEM encoding. Same reasoning for the Certificate; in that case we used DER encoding (that brings to 955 bytes for example).

3.1.3 HelloAccept Message

HEADER	ID	NONCE-PEER	DH-PUBKEY	SIGNATURE
1 byte	30 bytes	16 bytes	N bytes	256 bytes

Figure 4: HELLOACCEPT message format

This is the last message of the authentication. Besides the header and the ID, already mentioned, we have the NONCE-PEER, the nonce received before from the server. The DH pubKey is generated with the same parameters shared with the server. Finally, as every message, everything is signed.

3.1.4 Protocol Analysis

Let's analyse the properties of this protocol. This process has the purpose to authenticate both the peers, so at the end both client and server are sure that they are communicating each other and not with others.
The first message is sent from the client and it contains, besides his ID, also a nonce (we'll let the timestamp for a later explanation), and everything is later signed. The signature is necessary for the receiver to ensure that this message is made from the client and from no one else, because nobody else has his private key (we assume), that is a long-term key. In this way the server can guarantee his identity. The nonce instead is inserted because in the following message, the *HELLORESP*, the client expects that it will be signed from the recipient and sent back, so he will be sure that the first message has been sent to him.
In the *HELLORESP* message is also sent another nonce and the DH public key. The nonce is sent for the same reason before. Since the *HELLORQST* message can be saved and later reused (replay attack), the server sends this kind of "challenge" in which it expect that the client will be able to sign it and resending back to him (we remember the double authentication), as we can already see from the *HELLOACCEPT* message. The DH public is already sent to the client in such way he already can generate his pair of DH key and create a session key. The DH public key is then sent in the *HELLOACCEPT*. We also see that in the *HELLORESP* message there's also a certificate. This is made from a CA and contains the public key of the server, so before the client can verify his signature, the public key must be extracted from the certificate.
Notice that **replay attack are not possible**. A replay on the first message is actually able to produce a right response from the server, but the attacker will no more able to create the *HELLOACCEPT* message to finish the authentication since he doesn't have the private key to sign the nonce sent from the server.
We still have to explain the reason of the Timestamp. As before mentioned, the protocol is already secure against replay attack but a replay of the *HELLORQST* message is still possible and cause a good response. So, we can imagine a scenario in which the same message is sent a multitude of time, overcharging the server, or worst repeating the same request until the same nonce of the server of a previous authentication is produced, so also an *HELLOACCEPT* message can be reused a considered valid. Even though this last scenario is theoretically possible, it could not be affordable since the nonce is on 128bits (16byte) and this can require at maximum $2^(128)$ messages (actually much less considering birthday attack). Moreover, the two DH public keys are still different so this kind of attack would just bring to an invalid authentication. Anyway, the server still response with a valid message and this could not be a good behaviour. Hence, to avoid apriori any kind of replay attack, we introduce the timestamp. The timestamp ensure that the message is actually created in that moment from the client, and it will be accepted if the Timestamp falls in the time window prepared from the server. This approach makes replay attack virtually impossible.

3.1.5 Complete Protocol

We also report the complete sequence diagram of the protocol, in which there's also the error message *HELLOREFUSE* that has different contents from the client to the server.

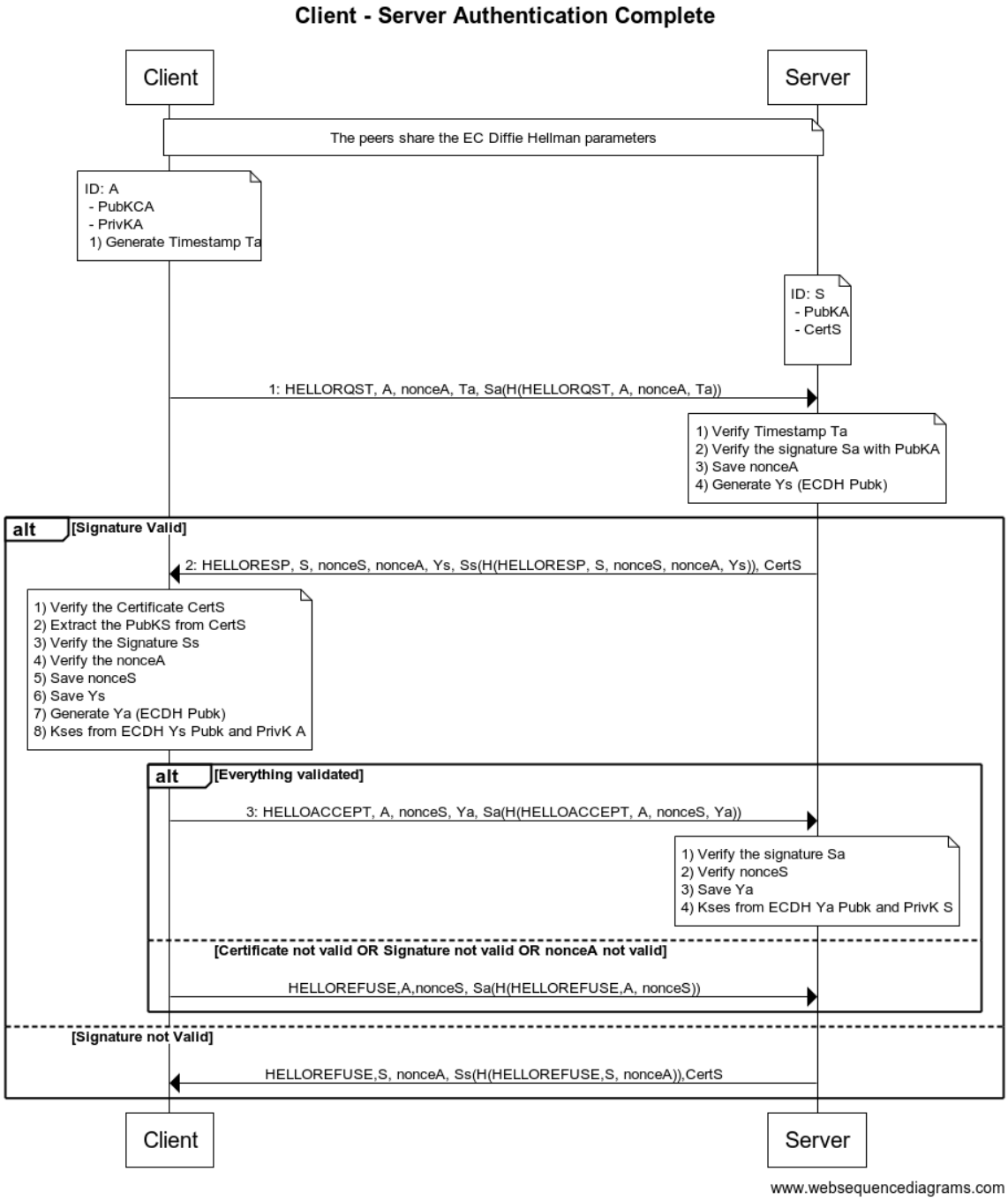
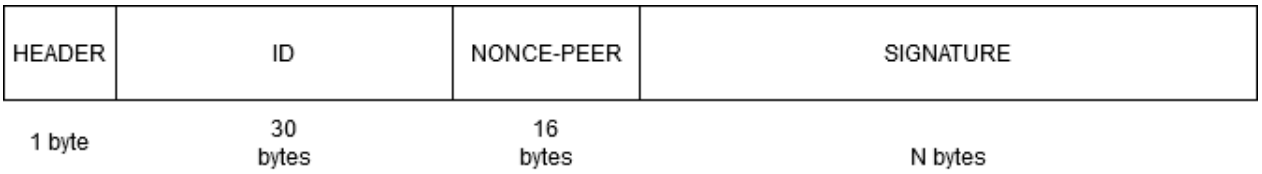


Figure 5: Sequence Diagram: Client - Server Authentication Complete

3.1.6 HelloRefuse Message

These messages still contain signatures. The reason is because an attack could make a MIM attack sending an unauthenticated refusing message blocking the authentication. In this way this is impossible. Moreover, the inserting of the nonce-peer (the nonce previously sent) ensure that is not a replay of another refusing message, but it a response to a request.

HelloRefuse response to HelloRequest



HelloRefuse response to HelloResponse

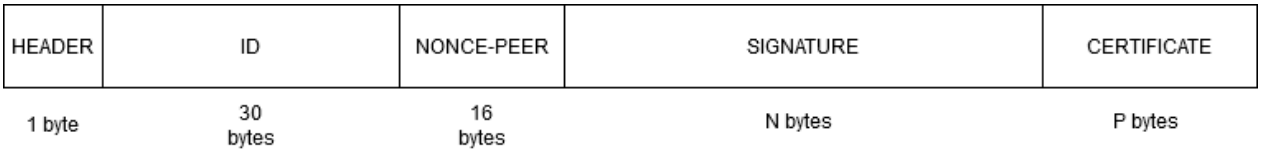


Figure 6: HELLOREFUSE message format 2

Headers and Message Codes

Headers	
Message Name	Header Code
HELLORQST	0x26
HELLORESP	0x27
HELLOACCEPT	0x28
HELLOREFUSE	0x2A

Table 1: Headers table

Chapter 4

Client Server Communication

4.1 The protocol

We assume that:

- Clients and server share session keys.
- The server has clients' public keys.

Once client and server are authenticated, they can now communicate each other in encrypted way. From the specification we need to implement also autenticity as well as the integrity of the messages. Encryption Algorithms chosen:

- **AES-128-GCM/GMAC**, for confidentiality, authenticity, and integrity.

We report the two kind of message used for all the communications. Both messages use of course the shared secret produced in the previous protocol.

Encrypted Message with Tag

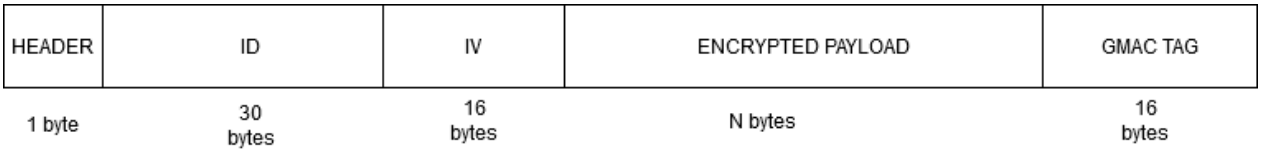


Figure 1: Encrypted Message with tag format

This message, as any other, contains Header and ID. The last three fields are used in the GCM mode. The IV sets the unicity of the encryption, the GMAC TAG ensure the integrity and the Encrypted payload represents the ciphertext produced. The cipher used is **AES-128bit** enough to ensure no possibility of bruteforce attacks nor known-plaintext-attacks. The TAG of the message is on 16byte and derive from the GCM encryption mode using as AAD (Additional Authenticated Data) the Header and the ID. It's not necessary to insert the IV in the AAD, since a forging of the IV, will affect the TAG anyway and so it would be detected.

Authenticated Message

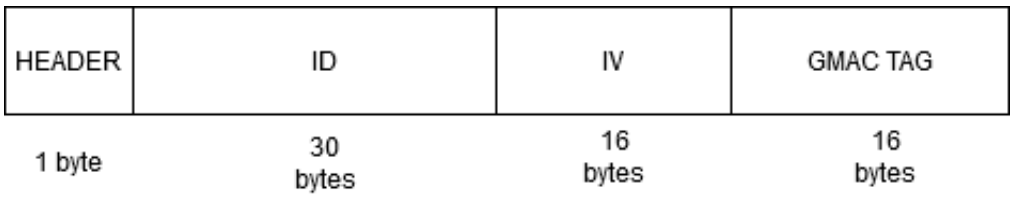


Figure 2: Authenticated message format

This message follows the same principle of the previous one. The difference is that there is no plaintext to cipher and so no ciphertext is produced.

As already mentioned we use the IV in couting mode, in this way we avoid replay attacks, but the first IV is chosen randomically from the client in this message. The server will then save this IV and it will be used incrementally for every following message as already said. This approach is used in both the messages.

Now let's see how this two types of messages are used in the exchange of information between Client and Server.

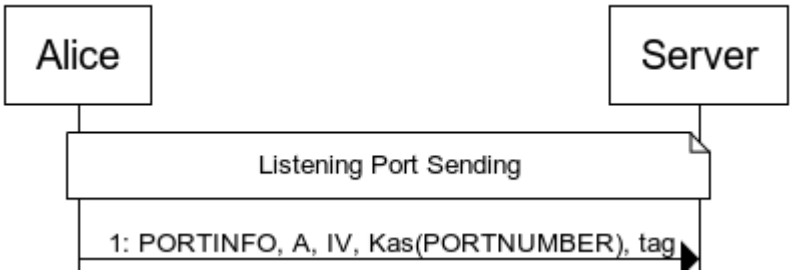


Figure 3: Sequence Diagram: PortInfo sending

In the little sequence diagram shown before, we can see just one message sent. This message, called *PORTINFO*, **is mandatory and is sent just after the Client – Server Authentication**. It contains the port number of a socket open for all the incoming game request, we'll see later how is used. The port number is encrypted.

Just for the sake of information, it's possible to notice that since the port number is number limited on 16bit so we have a maximum of 65536 ports (actually the first 1024 are reserved) and since the open ports on a machine are discoverable it would be theoritically possible to act a known-plaintext attack. Given the power of AES-128 this is not effectively possible.

4.1.1 Users list request from client

From the following sequence diagram, we can also see the request of the list of logged users made from the client. This message is optional since the client can decide if requiring the list of users. Actually, a first instance of this message is mandatory to initiate a game request, but if the client want to wait for a request from another client is not necessary.

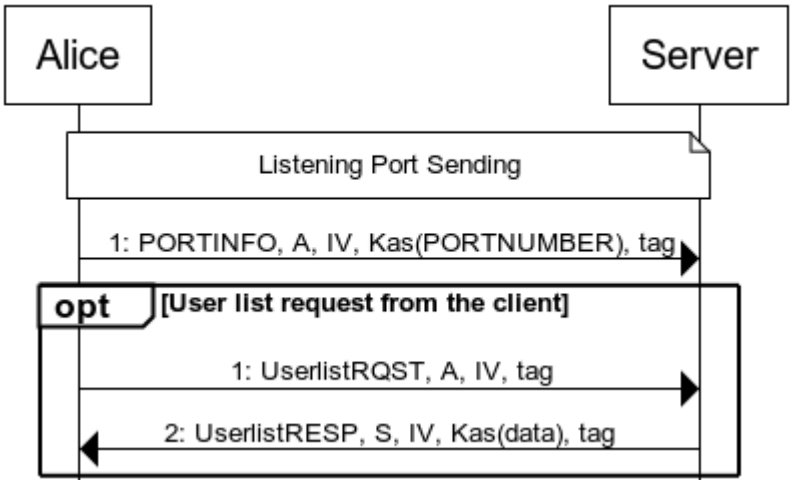
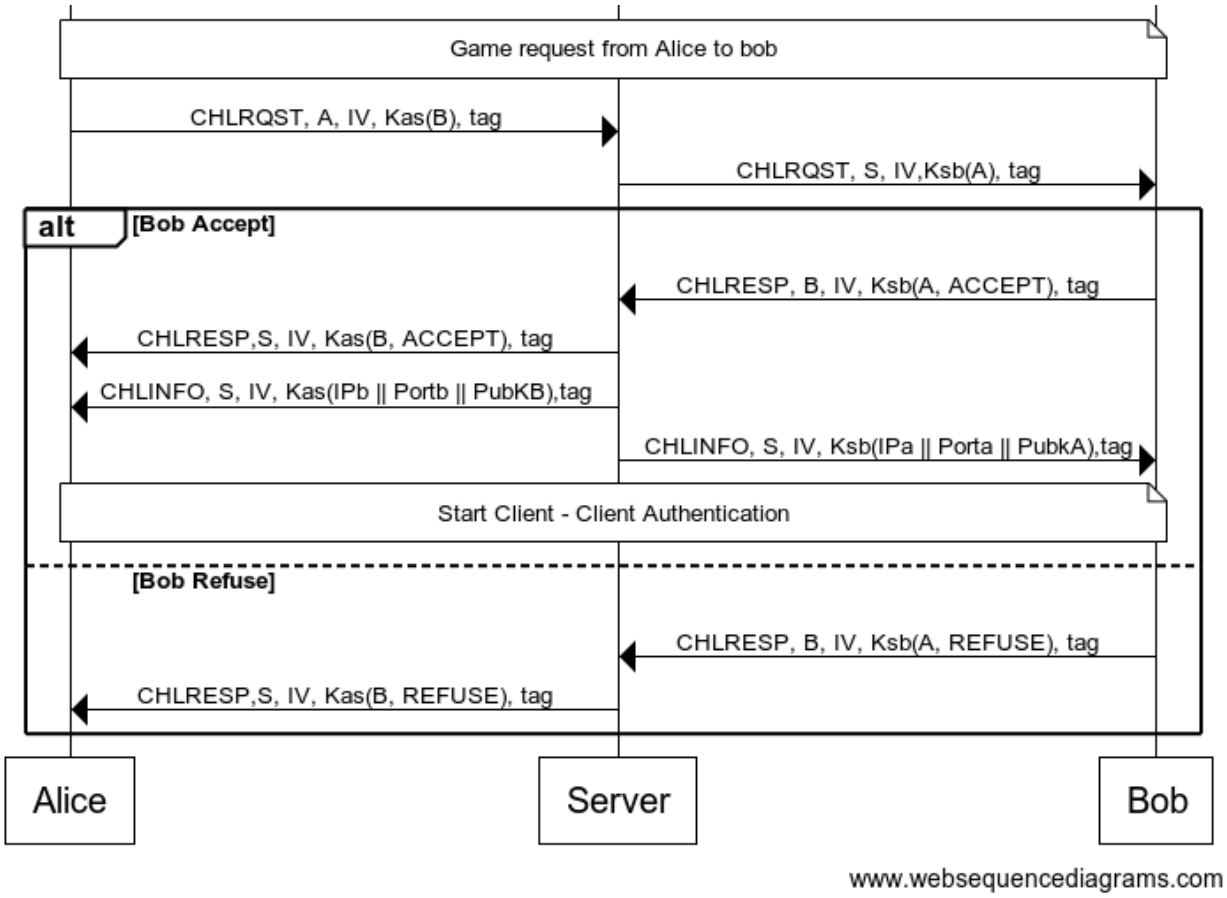


Figure 4: Sequence Diagram: Userlist request example

The message *USERLISTRQST* doesn't contain nothing to encrypt, so the *Authenticated Message* type has been used. The *USERLISTRESP* is an *Encrypted Message with tag* . The encrypted payload in this case will contain the list of the logged users, that will be decrypted and deserialized from the client.

4.1.2 Game request

In the following sequence diagram is reported an example of game request from Alice to Bob.



Everything starts with an *CHLREQST* message (challenge request) from Alice directed to the Server. It states that she wants to play with Bob. In fact, his ID will be the plaintext of the encrypted payload of the message. The server decrypts this message, reads the ID of the user, and forward the message to Bob (we now assume that Bob is logged). This time the payload contains the ID of the challenger.
Even though the message is the same with the same header, Bob recognizes that is a game request because it derives from the server. In fact, the only task of the server is being the "hub" for these messages. **Bob can either accept or refuse** the request. In either way the response message has the header *CHLRESP*, and the encrypted payload contains the ID of the challenger concatenated to the response *ACCEPT/REFUSE* encoded with predefined hex values. In both cases the response will be redirected to the challenger from the server at the same way of *CHLRQST*.
However, **in case of acceptance the two peers should start a Client – Client authentication**. To allow this subsequent phase, the server sends to both peers the respective information just after the previous message. This information contains the IP of the other player, the port number sent before in *PORTINFO* message, from which the users listen for new connections, and

the RSA Public key of the peer. As it's possible to see from the diagram all of these information are encrypted. Just after receiving the information, the peer who sent the challenge request start the Client – Client -Authentication.

4.1.3 Exceptional Cases

Request to a user not logged anymore

Since the choice of the user to play with another peer is made on the base of the list received from the server, it's possible that in the meantime some peers disconnected from the server. **The list not updated can cause a request to a user not logged anymore**. To handle this case the server sends back a message to the player of the unavailability of the peer and advices to update the list (sending another request). This is made with the use of the message *NOTLOGGEDUSER*, that is just an advise message of the type *Authenticated Message*.

Concurrent requests to the same peer

It's possible the two peers challenge the same user at the same time. This can provoke problems altering all the sequences of the messages exchanged. To avoid this problem, we set a busy state to both peers concurring to the challenge, just after the handling of the message *CHLRQST* from Alice. In this way any other request from a third peer will be refused by the server. This situation is state from a response message from the server with header: *BUSYPEER*. **Important to highlight that the state *busy* is set from the Server and is unset from him only when the recipient of the game request refuse to play. In other cases is task of the clients to free the state.** This is made by using the message *FREESTATE* that is an Authenticated Message. The use of this message will be shown later.

Headers and Message Codes

Headers	
Message Name	Header Code
PORTINFO	0x29
USERLISTRQST	0x2D
USERLISTRESP	0x2E
CHLRQST	0x38
CHLRESP	0x39
CHLINFO	0x64
BUSYPEER	0x3A
NOTLOGGEDPEER	0x3B
FREESTATE	0x96

Message Codes	
CHLRESP contents	Code
ACCEPT	0x62
REFUSE	0x63

Table 1: Headers and message codes

Chapter 5

Client Client Authentication

5.1 The protocol

We assume that the two peers received the mutual network information and their public key correctly from the server.

- The Client owns his Private Key.
- The Client owns the Public Key of the other peer.

Encryption Algorithms chosen:

- **ECDH**, to create an ephemeral shared secret.
- **RSA2048**, the algorithm of the long term key pair.

Here below is shown the sequence diagram of the Authentication. We assume that both have the respective public keys, that are trusted. The peer who request to play initiate the protocol sending the *KEYEXCHRQST* message. This message has exactly the same structure of the *HELLORQST* for the Client-Server Authentication. Hence, the second peer answers to the request exactly in the same way of the server in the previous authentication. The only difference is the absence of the certificate, since we already have his public key. The last message, *KEYEXCHCONFIRM* is instead exactly the same of the *HELLOACCEPT*.
It's important to state that in case something goes wrong in the authentication: signature not valid, nonce not valid and so on, both peers freeing their state to the server, in this way they are available to accept new game requests. This is made by the use of *FREESTATE* message. After the authentication a shared secret is used at the same way of client – server. We used the same Elliptic Curve DH parameters.

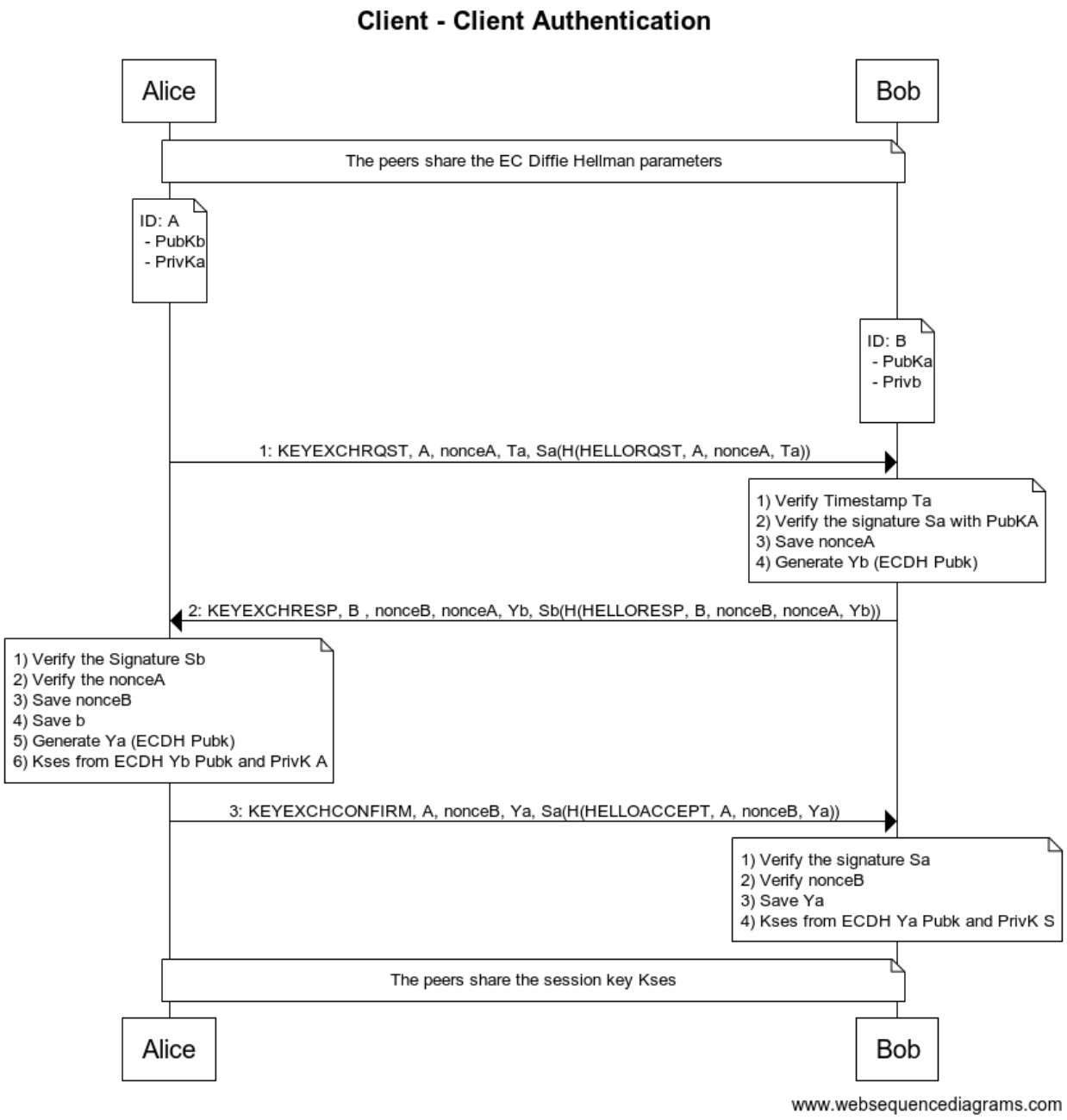


Figure 1: Sequence Diagram: Client - Client Authentication

Headers

Headers	
Message Name	Header Code
KEYEXCHRQST	0x86
KEYEXCHRESP	0x87
KEYEXCHCONFIRM	0x88

Table 1: Headers

Chapter 6

Client Client Communication

6.1 The protocol

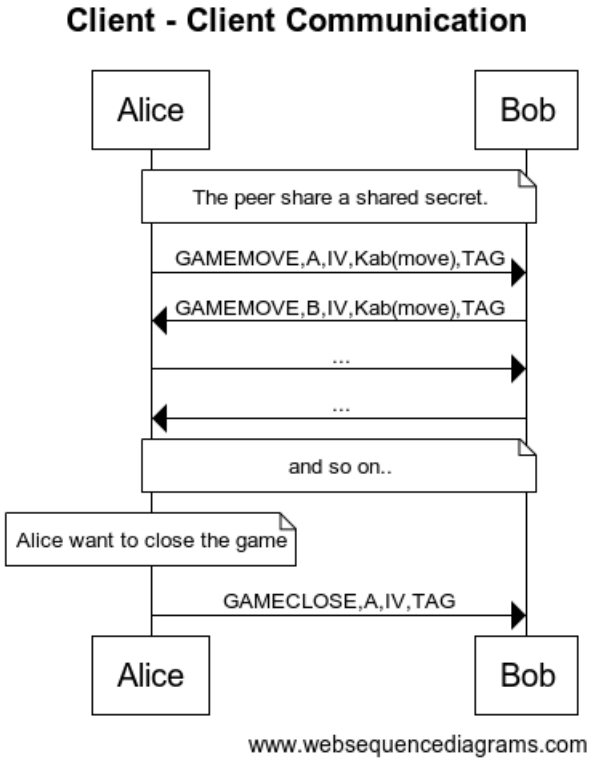
We assume that the two peers accomplished correctly their authentication.

- The Clients shared a secret.

Encryption Algorithms chosen:

- **AES-128-GCM mode**, used for the communication.

Here this record protocol is pretty simple since only two messages can be exchanged. The first one is a game move stated with the header *GAMEMOVE* that is an encrypted message with tag type. The content of this message can be a number between [1,7] as state from the game specification and denoted by the name "move" in the protocol, represent by a 16bit integer. The second with a closing statement where the player doesn't want to play anymore: *GAMECLOSE*, an Authenticated message.
We remember that after closing the game for any reason, we send *FREESTATE* message to the server.



Headers and Message Codes

Headers	
Message Name	Header Code
GAMEMOVE	0x91
GAMECLOSE	0x92

Table 1: Headers

Chapter 7

BAN Logic

Security protocols are three-line programs that people still manage to get wrong.

Roger M.Needham

7.1 Preliminaries

BAN logic is a kind of logic invented by **Burrows, Abadi, Needham** that uses a set of rules to analyse information exchange protocols. This logic starts with a set of basic rules and with the using of postulates and starting assumption test if the protocol reach some conditions at the end.

7.2 Notation

Figure 1 shows the main symbols used in BAN logic and their (informal) meaning.

$P \models X$	P believes X	$P \xleftrightarrow{K} Q$	K is a (good) symmetric key for P and Q
$P \mid\sim X$	P once said X	$\{X\}_K$	X is encrypted with key K
$P \lhd X$	P sees X	$\xmapsto{pk_p} P$	P has pk_p
pk_P	Public key of principal P	$P \Rightarrow X$	P has jurisdiction over X
pk_P^{-1}	Private key of principal P	$\sharp(X)$	X is fresh

Figure 1: Figure 1. Ban Logic Notation

Moreover, if *P says X that P believes X.*

7.3 Safe protocols

Typically one wants to prove that a certain protocol is *safe*. Which involves proving that a key is *good*. In BAN logic this means proving the following:

$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ $B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$	Key Authentication
$A \mid\equiv B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ $B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$	Key Confirmation
$A \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$ $B \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$	Key Freshness

Table 1: Typical Objectives

7.4 Postulates

These postulates derive from the base rules and from extended one especially for the public key ones. These are used to make conclusion.

Symmetric Key Rule

$$\frac{P \mid\equiv Q \overset{K}{\rightleftharpoons} P \ , \ P \triangleleft \{X\}_K}{P \mid\equiv Q \mid\sim X} \text{ (SK)}$$

Freshness Distribution Rule

$$\frac{P \mid\equiv \#(X)}{P \mid\equiv \#(X, Y)} \text{ (FD)}$$

Nonce Verification Rule

$$\frac{P \mid\equiv \#(X) \ , \ P \mid\equiv Q \mid\sim X}{P \mid\equiv Q \mid\equiv X} \text{ (NV)}$$

Weakening Rule

$$\frac{P \mid\equiv Q \mid\equiv (X, Y)}{P \mid\equiv Q \mid\equiv X} \text{ (W)}$$

Jurisdiction Rule

$$\frac{P \mid\equiv Q \Rightarrow X \ , \ P \mid\equiv Q \mid\equiv X}{P \mid\equiv X} \text{ (J)}$$

Public Key Rule

$$\frac{P \mid\equiv \overset{pk_p^{-1}}{\longmapsto} P \ , \ P \triangleleft \{X\}_{pk_p}}{P \triangleleft X} \text{ (PK)}$$

Sign Rule

$$\frac{P \mid\equiv \overset{pk_q}{\longmapsto} Q \ , \ P \triangleleft \{X\}_{pk_q^{-1}}}{P \mid\equiv Q \mid\sim X} \text{ (SPK)}$$

Extended Public Key Rule

$$\frac{P \mid\equiv \#(X) \ , \ P \mid\sim \{X, Y\}_{pk_q} \ , \ P \triangleleft \{X, Z\}_{pk_p}}{P \mid\equiv Q \triangleleft \{X, Y\}} \text{ (EPK)}$$

Sees Rule

$$\frac{Message\ n : P \rightarrow Q : X}{Q \triangleleft X} \text{ (S)}$$

7.5 Idealized Protocols

Since the notation in the literature for protocols is not suitable for formal analysis we need to idealize our protocol. A message in idealized form is a formula in BAN logic. This means that the information in the protocol must be coded in BAN formulas. The process of idealizing a protocol is not formally defined, but this step is crucial in an analysis. Wrong idealizations lead to incomplete (our worse: totally wrong) proofs.

7.6 BAN Logic Extensions

Ban Logic albeit a powerful tool, has several limitations. One of these limitations is that there are no postulates for Diffie-Hellman key agreement, even though is one of most popular approach to this problem nowadays. This problem has been solved adding several postulates for example in []. We report some of these postulates that we will use later.
We define as $PK_\delta^{-1}(A)$ the DH private key of A, and $PK_\delta(B)$ the DH public key of B.

Unqualified Key-agreement rule

$$\frac{A\ has\ PK_\delta^{-1}(A), A\ has\ PK_\delta(u)}{A\ has\ K}$$

where $K = f(PK_\delta^{-1}(A), PK_\delta(B))$.
This rule is unqualified because it doesn't exclude the possibility of anyone else can create a valid PK_δ.

Qualified Key-agreement rule

$$\frac{A \mid\equiv PK^{-1}(A),\ A \mid\equiv PK(B),\ A \mid\equiv PK_\delta^{-1}(B)}{A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B}(\text{QKA})$$

This rule assumes also the belief of the identity of the keys.
From some considerations we can derive that *if B reclaims in some way his identity on* $PK_\delta(u)$ *then we can assert that he owns* $PK_\delta^{-1}(B)$.

Key Identity confirmation

$$\frac{P \mid\equiv \overset{pk_q}{\longmapsto} Q\ ,\ P \triangleleft \{PK_\delta(Q)\}_{pk_q^{-1}}}{P \mid\equiv PK_\delta^{-1}(Q)}\ (\text{KIC})$$

7.7 Client-Server Authentication

We report a formal notation for the Client-Server Authentication protocol.

$$\begin{aligned}
&(1)\ A \to B\ : A, N_a, T_a, \{SHA\{A, N_a, T_a\}\}_{pk_a{}^{-1}}\\
&(2)\ B \to A\ : B, N_b, N_a, Y_b, \{SHA\{A, N_b, N_a, Y_b\}\}_{pk_b{}^{-1}}, Cert_b\\
&(3)\ A \to B\ : A, N_b, Y_a, \{SHA\{A, N_b, Y_a\}\}_{pk_a{}^{-1}}
\end{aligned}$$

7.7.1 Idealized Protocol

Now we will report the idealized protocol, the first step in the analysis.

$$\begin{aligned}
&(1)\ A \to B :\ \{A \mid\sim N_a, A \mid\sim T_a\}_{pk_a{}^{-1}}\\
&(2)\ B \to A :\ \{B \mid\sim N_b, N_a, B \mid\sim Y_b\}_{pk_b{}^{-1}}, Cert_b\\
&(3)\ A \to B :\ \{N_b, A \mid\sim Y_a\}_{pk_a{}^{-1}}\\
&\quad Idealized protocol, suitable for BAN logic
\end{aligned}$$

7.7.2 Assumptions

1. $A \mid\equiv \#(N_a)$
2. $A \mid\equiv \#(T_a)$
3. $B \mid\equiv \#(N_b)$
4. $A \mid\equiv \overset{pk_a{}^{-1}}{\longmapsto} A$
5. $A \mid\equiv \overset{pk_a}{\longmapsto} A$

6. $B \mid\equiv \xmapsto{pk_b{}^{-1}} B$

7. $B \mid\equiv \xmapsto{pk_b} B$

8. $B \mid\equiv \xmapsto{pk_a} A$

9. $A \mid\equiv \xmapsto{pk_{ca}} CA$

Assumption 1 and 2 assert that A believes that her nonce and timestamp are fresh. At the same way B believes that his nonce is fresh in the assumption 3. In the last assumptions we assert that both the peer own their key pair and A owns the public key of the CA, assumption 9.

7.7.3 Analysis

Now we report a step by step analysis of the protocol, using the postulate to proof the protocol safety.
Before sending message 1, we know:

0.1 $A \mid\sim \{N_a\}_{pk_a{}^{-1}}$

0.2 $A \mid\sim \{T_a\}_{pk_a{}^{-1}}$

Sending message 1 leads to:

$$(1)\ A \to B:\ \{A \mid\sim N_a, A \mid\sim T_a\}_{pk_a{}^{-1}}$$

1. $B \triangleleft \{A \mid\sim N_a, A \mid\sim T_a\}_{pk_b{}^{-1}}$ *M1,**S***
2. $B \mid\equiv A \mid\sim (N_a, T_a)$ *1,A8,**SPK***

From 2 is possible to see that B sees the timestamp sent by A. The freshness of the Timestamp is verifiable for antonomasia, so also:

3. $B \mid\equiv \#(T_a)$.

Before sending the message 2, B generate a random value b that will be private DH key ($PK_\delta^{-1}(B)$), then he creates the public key $PK_\delta(B)$. From now on we'll use b and Y_b to denote DH private and public key respectively.
Before sending the message we know:

0.3 $B \mid\equiv \#(b)$

0.4 $B \mid\equiv \#(Y_b)$

0.5 $B \mid\sim \{N_b\}_{pk_b{}^{-1}}$

0.6 $B \mid\sim \{Y_b\}_{pk_b{}^{-1}}$

Sending message 2 leads to:

$$(2)\ B \to A:\ \{B \mid\sim N_b, N_a, B \mid\sim Y_b\}_{pk_b{}^{-1}}, Cert_b$$

1. $A \triangleleft \{N_b, N_a, Y_b\}_{pk_b{}^{-1}}, Cert_b$ *M2, S*

Now, we know that the $Cert_b$ contains the Public key of B and that $CA \Rightarrow Cert_b$, so from the J rule we can say $A \mid\equiv Cert_b$. Moreover, using the assumption A9 and the rule SPK, we arrive to:

2. $A \mid\equiv \overset{pk_b}{\longmapsto} B$

Then:

3. $A \mid\equiv B \mid\sim (N_b, N_a, Y_b)$ *1,2,SPK*
4. $A \mid\equiv \#(N_b, N_a, Y_b)$ *3, A1, FD*
5. $A \mid\equiv B \mid\equiv (N_b, N_a, Y_b)$ *4, NV*
6. $A \mid\equiv Y_b$ *5, W, J, KIC*

Before the third message A generate his DH private key a and his public key Y_a, so $A \mid\equiv \#(Y_a)$ $A \mid\equiv \#(a)$.
Now, A is able to generate the shared secret. Furthermore, this secret is fresh because derives from fresh quantities.

7. $A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ *6, QKA*
8. $A \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$ *4*

Before sending the third message:

0.7 $A \mid\sim \{Y_a\}_{pk_a{}^{-1}}$

Sending message 3:

$$(3)\ A \to B :\ \{N_b, A \mid\sim Y_a\}_{pk_a{}^{-1}}$$

9. $B \triangleleft \{N_b, A \mid\sim Y_a\}_{pk_a{}^{-1}}$ *M3,S*
10. $B \mid\equiv A \mid\sim (N_b, Y_a)$ *9, SPK*
11. $B \mid\equiv \#(N_b, Y_a)$ *10, A3, FD*
12. $B \mid\equiv A \mid\equiv (N_b, Y_a)$ *10, 11, A3, NV*
13. $B \mid\equiv Y_a$ *12, W, J, KIC*

Now, B can create the share secret, and he knows it's fresh because derives from fresh quantities.

14. $B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$ *QKA, 13*
15. $B \mid\equiv \#(A \overset{K_{ab}}{\longleftrightarrow} B)$ *11*

Hence, we can state that we have proof of **Key Authentication and Key freshness** (7, 8, 14, 15). One thing that is missing is **Key Confirmation**.
If we add the first message of Client-Server Communication that actually as we already mentioned is mandatory in our application, we can proof also this property.

$$(4)\ A \rightarrow B:\ \{PN\}_{K_{ab}}$$

16. $B \triangleleft \{PN\}_{K_{ab}}$ *M4,S*
17. $B \mid\equiv A \mid\sim PN$ *16,SK*
18. $B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

Hence, finally we also have **Key Confirmation** but only for the B peer. The Key confirmation for the peer A will be accomplished only later in the communication.

7.8 Client-Client Authentication

This protocol respect to the previous one has no differences besides the absence of the Certificate. Hence, respect to the previous one we already start from an assumption in more: $A \mid\equiv \overset{pk_b}{\longmapsto} B$.

Chapter 8

Implementation

8.1 Analysis

This project even if is not big in the scope, can be complex since it can have overlapped and intertwined componenents. For this reason is a good practise using software engineering methodologies. The programming language used is C/C++.
The project has been developed over two executable:

- Client
- Server

These two executable work around four classes:

- Client class
- Server class
- FourInARowGame class
- SecureConnectionServlet class

The *Client class* and the *Server class* are the main classes that will later represent the main code of the two executable. *FourInARowGame* class is the class that develops the game. Finally the *SecureConnectionServlet* class will develop all the utility functions necessary for the creation of all the protocols. The former has been created to be totally decoupled from the "utilizers" (Client and Server classes). For this reason the actual *consecutio* of the protocol is implemented inside the two main classes.
This approach allows to develop protocols in such way Client and Server only know the name of the messages and what it's found inside but not how the message is encrypted/decrypted etc.