Secure Game

Security of Information and Organizations Project 2

Rafael Remígio 102435
Bruno Moura 97151
João Correia 104360

January 7, 2023

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro
Year 2022/2023

Contents

1 Introduction

The proposed assignment entails the designing and implementation of a robust protocol for handling a simplified game of bingo played on a distributed system.
Instead of trusting an authoritative party to authenticate messages and certify that the game is to be played fairly, it is assumed that every playing entity may cheat and as such trust is instead placed on the security mechanisms in place. To this end, we used *asymmetric and symmetric cryptography*, *digital signatures*, emphsmart cards and *certificates*.
This document aims to document the implementation and architecture of the distributed system and protocol, as well as the design decisions behind their creation.

2 Entities

The system comprises of three types of entities, the Playing Area, Caller and a Player.

Playing Area

The first is the playing area, a secure playing field. It does not interfere with the game directly, but instead handles authentication and authorization.
Players and callers all interact with the playing area instead of with each other directly. Its job is to verify messages and forward them around. It possesses a RSA 128 bit asymmetric key pair for signing messages.

Player

The second type of entity is the player. As the name implies, these are the entities that will actually play the game and are eligible for winning and losing. There must be at least two players per game.
During the game, the players will generate a card of numbers from the deck provided by the caller. The player whose card gets filled first wins, as per bingo rules.
It possesses a Portuguese citizen card capable of signing documents, a RSA 128 bit asymmetric key pair and a AES 128 bit symmetric key. Their use will be explained later own.

Caller

Lastly, the third and final type of entity is the caller, of which there is only one per game. The caller is responsible for generating the deck, an ordered list of numbers that will simulate the draw of balls of a traditional bingo game.
The caller possesses the same keys as the player. In fact, a caller is functionally identical to an user during the authentication and registration process. So whenever the term user is used, what is being said applies both to players and callers.

3 Walk-through

This section will outline the steps the system takes for running a game of bingo. Further details on the protocol messages will be covered in the next section.

3.1 Authentication

The game cannot start until there are enough users and a caller connected, authenticated and registered to the playing area. After connecting to the playing area via a web socket, the next step for a user is to authenticate themselves.
The authentication entails verifying that the user in question is a Portuguese citizen, as proved by their possession of a Portuguese citizen card (CC) and it's PIN.
The authentication process is done through a Challenge-Response approach, in which a four step handshake is done to successfully authenticate a user.
The handshake goes as follows:

1. The user asks the *Playing Area* to be authenticated by sending a authentication

message containing their RSA public Key (We were going to use the key present in the CC but it was no possible to complete this is time).

2. The *Playing Area* sends a nonce to the user as a challenge.

3. The *user* encrypts the challenge with their RSA private key and sends it back to the playing area.

4. The user verifies that the response it got matches the nonce and public key. If so, a message is sent to the *user* letting them know they are authenticated.

The nonce consists of a string of random characters of variable length (by default, 14). It is wise to not set this length too low otherwise repeated challenges might occur and adversaries may then employ replay attacks to bypass authentication.
Authenticated users are not ready to join the game as of yet. For that, they must be registered. At this step, the playing area merely recognizes them as a Portuguese system citizen. They can now, however, ask the playing area for message logs and a list of registered users.
Also, the playing area possesses a set of RSA public keys that are reserved for callers. As such, what determines if a user will be a player or a caller.

3.2 Registration

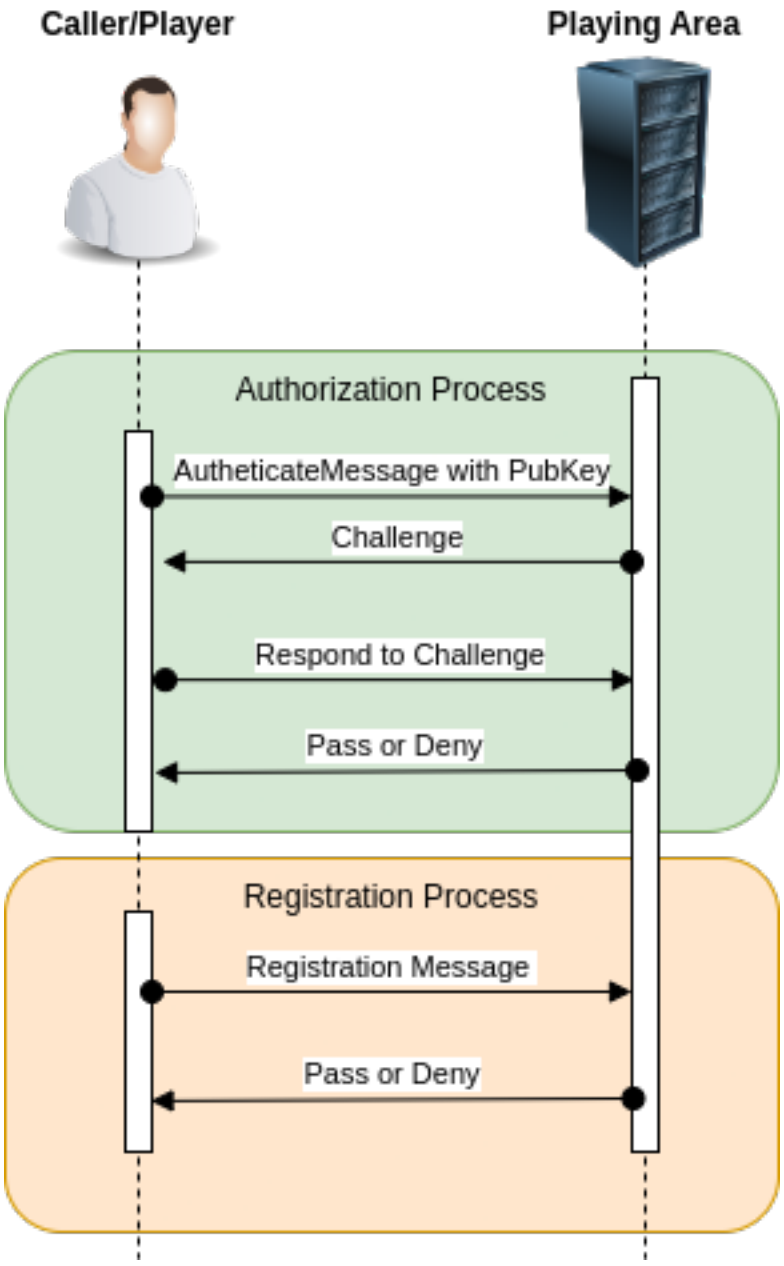
The next and final step for a user to be fully ready to participate in the game is the registration process. It's purpose is to establish what is the public key that will be used during the game, as the CC's key pair is only used for authentication purposes. The user must also register himself a nickname to aid identification by humans.
The registration also is done through a handshake.

1. The user asks the playing area to be registered by sending a registration message. The message contains the public key from the key pair just now generated (the playing key), their nickname, the CC public key and a signature.

2. The playing area receives the message and validates the signature. If it's valid, the user is now registered. The playing area sends the message back confirming the registration.

After the registration, the playing area follows up with a message informing about some data about the game – namely the user's sequence (sequential ID), the size of the cards and the size of the deck.
At this point, the playing area will automatically start the game once a caller and a variable amount of players (default 2) have registered themselves. Once a user is registered, every other registered user is notified by the playing area (the same also happens if a registered user disconnects) During earlier stages of development, we considered merging the authentication and registration; The user would provide their nickname, the public key that will be used for the game, their CC's public key and a CC digital signature.
We ended up opting for a Challenge-Response authentication approach to prevent a particular kind of attack, in which an adversary somehow gets a hold of the private key of a

key pair used by a victim in a game they were in. They can then reuse the authentication message used in the original game to play as the victim as long as they use the compromised key pair as playing key



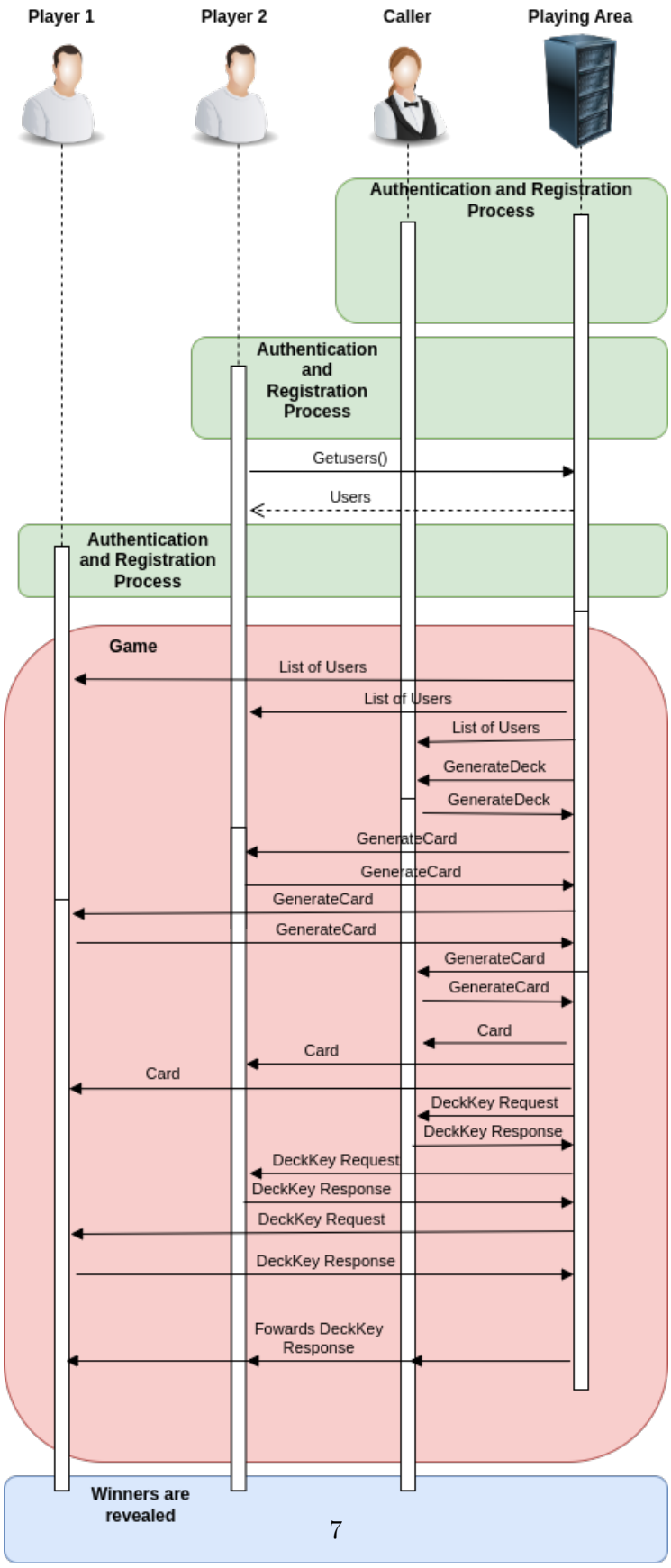
3.3 Deck and Card Generation

Once the game automatically starts, the first step is to generate the deck using the procedure described in the project guidelines.
The *playing area* notifies the *caller* that the game has started and that they should generate

the deck. The caller, that by now already knows how long the deck should be, generates a shuffled list of **N** numbers from 0 to **N-1** (where N is the deck size). Each element from this list is encrypted using their *AES symmetric key* (known as deck key) and signed with their private playing key.
From here on out, this deck gets passed around to all players in sequential order. Each player encrypts the deck with their own deck key, shuffles it around and appends their signature to it. Once every user has done it, the deck goes back to the caller who signs it once more.
Now that the final version of the deck has been created, it is committed to the playing area for all to see. The players and caller must now make their deck key known so that they can reverse the process done to arrive at the original deck and every player's card.
So now every user is decrypting and unshuffling the card in reverse order, verifying the signatures along the way. At the very end, they all will arrive at the unencrypted, original deck issued by the caller. If something goes wrong along the way, the game is aborted as someone is trying to cheat.
It's important to note that every shuffling described in this section was done deterministically, using the deck key of the shuffler as seed, which was at that point a secret. Once they are known by all, the shuffling can be undone.
With the unencrypted deck known by all, each user can redo the shuffling in order to arrive at the unencrypted deck as shuffled by each player. The first **M** numbers are taken to be that player's card, where **M** is the size of the card as determined by the playing area.

3.4 Winner Detection

At this point, every player and caller knows the original deck and each others cards in a way and knows that nobody could have cheated.
All that is left to do is to go through each card and see which one gets filled the earliest. There is always at least one winner.
The users don't need to compare their findings as the process itself demands that no foul play must have taken place. Now that everyone knows who the winners are, they all cleanly disconnects from the playing area.


Player 1
Player 2
Caller
Playing Area
Authentication and Registration Process
Authentication and Registration Process
Getusers()
Users
Authentication and Registration Process
Game
List of Users
List of Users
List of Users
GenerateDeck
GenerateDeck
GenerateCard
GenerateCard
GenerateCard
GenerateCard
GenerateCard
GenerateCard
Card
Card
Card
DeckKey Request
DeckKey Response
DeckKey Request
DeckKey Response
DeckKey Request
DeckKey Response
Fowards DeckKey Response
Winners are revealed

4 Protocol

Communication Protocol and Cryptography Protocol documentation.

4.1 Communication Protocol

4.1.1 Messages

- **Message**

Description	Generic message
Methods	– to_json() – should_log()

- **Authorization**

Description	Message for Users to authenticate themselves to the playing area. Uses challenge-response authenticatio
Extends	Message
Methods	– parse()

- **Register**

Description	Message for players to register themselves to the playing area
Extends	Message
Methods	parse()
Parameters	– nickname – playing_key – auth_key – signature – success – sequence

- **GetUsers**

Description	Message for getting a list of registered users
Extends	Message
Methods	parse(), should_log() **returns false**
Parameters	– public_key – signature – response

- **CardSize**

Description	Simple message for letting users know the card size
Extends	Message
Methods	parse(), should_log() **returns false**
Parameters	– card_size

- **GetLog**

Description	Message for getting a list of logged messages
Extends	Message
Methods	parse(), should_log() **returns false**
Parameters	– public_key – signature – response

- **PartyUpdate**

Description	Message for updating registered users on how big the party is
Extends	Message
Methods	parse(), should_log() **returns false**

Parameters	– signature – public_key – response

- **GenerateDeck**

Description	Message telling the caller to generate the deck and initiate the card generation proccess
Extends	Message
Methods	parse()
Parameters	– deck_size

- **GenerateCard**

Description	Message Players will pass around until everyone has commited their card
Methods	parse(), sign(), verify
Parameters	– sequence (int) – deck (list) – signatures (list) – done (bool)

Methods

- sign(private_key : str)
  Description: Signs the deck with private_Key and append the public key to the signatures array Call our Cryptography function **sign**
- verify(publicKey : str, signature)
  Description: Verifies the signature of the deck with the public key Call our Cryptography function **verify**

- **DeckKeyRequest**

Description	Message requesting that players and caller reveal their symmetric key after the deck is commited
Extends	Message
Methods	parse()
Parameters	sequence

- **DeckKeyResponse**

Description	Response to the deck key request
Extends	Message
Methods	parse(), sign()
Parameters	– sequence – response – signature

- **GameOver**

Description	Message for when the game is over / aborted
Extends	Message
Methods	parse() should_log() **returns false** __ str __()
Parameters	– status

4.2 Cryptography Protocol, Crypto (Class)

Cryptographic utilities

Uses **cryptography.hazmat.primitives**

4.2.1 Methods

- **sym_gen**

Description	Generates a new AESGCM (key, nonce) tuple
Parameters	
Return	tuple

- **sym_encrypt**

Description	Encrypts data given with given AESGCM key

Parameters	– key: bytes – data – nonce: bytes
Return	bytes

- **sym_decrypt**

Description	Decrypts encrypted data given with given AESGCM key
Parameters	– key: bytes – crypted_data – nonce: bytes
Return	bytes

- **do_hash**

Description	Returns an hash of a given data; Uses SHA256
Parameters	data: bytes
Return	bytes

- **asym_gen**

Description	Encrypts data using given public key
Parameters	
Return	bytes

- **sign**

Description	Returns Signature of given data signed with given private key; Uses SHA256
Parameters	– private_key – data
Return	bytes

- **verify**

Description	Verifies if given message matches with given signature; Uses SHA256
Parameters	– public_key – signature: bytes – message
Return	bool