universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Security

“Secure, multiplayer online domino game”

Final Report

Bruno Aguiar 80177, Fábio Nunes 80321,
José Moreira 79671, Paulo Sousa 80000

2020/2021

Index

Introduction

Serves this document as a report on the project about “Domino” within the scope of Security, a course unit part of Telematics and Computer Engineering master’s degree.

The objective of this project is to develop a system enabling users to create and participate in an online domino game (actually running in localhost).

The following quoted paragraph is an online (Wikipedia) description of Domino.

“Domino is a family of tile-based games played with gaming pieces, commonly known as dominoes. Each domino is a rectangular tile with a line dividing its face into two square ends. The traditional European domino set consists of 28 tiles, featuring all combinations of spot counts between zero and six.

One player begins by downing (playing the first tile) one of their tiles. This tile starts the line of play, in which values of adjacent pairs of tile ends must match. The players alternately extend the line of play with one tile at one of its two ends; if a player is unable to place a valid tile, they must continue drawing tiles from the stock until they are able to place a tile. The game ends when one player wins by playing their last tile, or when the game is blocked because neither player can play.“

The system is composed by a table manager and a bounded set of players, so it is based in a Server - Client architecture with the server representing the table manager dealing with three clients representing the players.

In this document we will exhibit how the game performs and which and how the security features are implemented.

Even with several modifications and due to the fact that it is not the primary objective of this project, it still needs to be mentioned that the bases for the non-secure game were not developed by us, they were developed by a course colleague, Guilherme Henriques (nmec: 80100). The game didn’t do much more than play pieces correctly if they were available, but saved some time to start implementing security as soon as possible, which we appreciated. We still built a lot of game logic on top of it.

In terms of project delivery, the commits done on our repository in *code.ua* do NOT represent who did the work, it was mostly merges of parallel work being done. The work was distributed equally by all the group members.

Game Logic

The game is intended to work as follows:

Initialization

1. When the server starts, it loads its private and public key from a PEM file.
2. It creates a socket, and waits for players to connect.
3. When the client (player) starts, it must choose if it wants to be authenticated with Citizen Card or not, before being assigned with a random name.
4. The client generates symmetric and RSA keys, and informs the server of both its name and its RSA public key before registration in the game, and of his symmetric key after his register is successful.

Deck Distribution Protocol

1. The table manager creates every domino tile by itself, which is immediately encrypted with the symmetric key it generated.
2. The table manager sends the deck to every player, one by one in arriving order.
3. The player receives the deck, encrypts with his symmetric key, signs it with his private key, shuffles it and returns it to the server. Repeat the process for every player until all have encrypted, shuffled and signed the tiles.
4. After all players did the #3, and the table manager received all the tiles back, the game is ready to start.

Game Start

1. Immediately before the game starts, the table manager gives all players the symmetric keys, respecting the encryption/decryption order and encrypted by RSA to make sure only the players will receive this information individually.
2. The tiles are distributed randomly one by one to each player.
3. Every time a player receives a tile, he decrypts the tile (reason why we needed the keys beforehand) and makes a commit on the tile value. We will talk about this commit more on topic VI.
4. The server finishes giving the 5 pieces to each player, and has received a commit for each individual piece.

Gameplay Loop

1. The server indicates to a player it's his turn to play.
2. The player has 3 possible actions:
    a. Play, if he finds a fit piece in his hand.
    b. Ask for a piece, if he doesn't find a fit piece in his hand, and chooses an index from the stock.
    c. Pass, if he doesn't have a fit piece and there are no tiles left on the stock to ask for.
3. The server responds accordingly:
    a. Accepts the piece to the table and the game moves on.
    b. Pops the indicated piece from the stock and gives it to the player.
    c. Moves on.
4. Sends a message to every player with the intent to:
    a. Give them a refresh on the table
    b. Let them claim a win, protest against cheating, or allow the game to proceed
5. If the player asked for a piece, he shall be given the next action again, so he doesn't skip a playing action.

Finishing State

1. There are 3 possible finishing states:
    a. A player claimed victory
    b. A player claimed he detected cheating
    c. The game finished in a draw due to no possible play and no pieces left on the stock.
2. The answer from the table manager will vary to each finishing state:
    a. To ensure a player has claimed victory correctly, the table manager verifies if his hand is empty, and checks that the tiles he played are correct according to the commitments the player made and sent to the server. He then declares everyone who was the winner, and awards the winner 2 points.
    b. After someone protested against cheating, they will show proof by showing which tile was a duplicate. Either two from the table, or one from the table and one from his hand. The game finishes without awarding anyone points.
    c. The table manager awards everyone 1 point and finishes the game.

3. Game accounting is assured through a file, where the points from each game played are stored.

4. The server does a general configuration reset, and is now ready to accept another 3 clients, to repeat the game cycle.

Messaging Methods and Protection

Regarding communication, every message is exchanged through sockets. Every time the server needs to communicate with the client, or vice-versa, a dictionary is created, with a key and the message to be sent. For example in the function `givePiece(self, Player)`, when the server wants to send a piece to the client, the following dictionary is created: `msg = { 'piece': piece}`.

In total, all the entities have an *AES* key each, and a private and public key each, generated through *RSA*. In a simple way to put it: all the tiles are encrypted/decrypted with the *AES* keys, and the signatures are dependent on the private/public keys.

To ensure safety and that the message transmitted actually comes from the server, and since the content to be sent is already encrypted, the message is then signed by the server. The signed message is yet serialized using the *dumps* function of the *Pickle* module, and sent to the client. Had the sent data not been encrypted before the signing moment, an encryption would be made before the signing.

The client side must be ready to receive this information. To continue the piece requesting/receiving example, after the request was made, the client waits for the data, and is expecting the message (`msg = {'piece': piece}` *) to* contain the key *'piece'.* For this to happen, first the data must be deserialized, using the *loads* function of the *Pickle* module. After both these things happen, the client verifies the signature, to confirm if the message was in fact sent from the server, and if this succeeds, the piece is decrypted and sent to the player's hand. The player now sends a message to the server, with the same steps referred above, confirming the piece was received.

The details of the encryption and decryption itself will be talked on the Fernet topic.

It is also important to say we did not implement sessions between the players themselves but we thought of doing that by having Diffie–Hellman key exchanges for encryption and RSA keys for signing and verification.

Deck Distribution Protocol

The tile deck is created by the table manager (meaning, the server). Due to security reasons, the deck must be hidden.

In order to do that, the server encrypts the deck using **Fernet**.

Fernet AES 128

After some research and debate about the best method to use for encryption, we chose Fernet.

This is a system for symmetric encryption and decryption. It is included in the *cryptography* library. It uses 128-bit *AES*, in *CBC* mode, *PKCS7* padding, and HMAC authentication which is made using *SHA256*.

Fernet takes a message, a key and produces a *token*, which contains the encrypted message. It guarantees that a message encrypted using it cannot be manipulated or read without the key. It also authenticates the message, which means that the recipient can tell if the message has been altered in any way from what was originally sent.

```
f = Fernet(self.aes_key)
encrypted = base64.b64encode(msg)
encrypted = f.encrypt(encrypted)
```

At this point, the deck is symmetrically encrypted by the server, and it’s delivered to the first player. Now, with the objective of maximizing security, the deck is also encrypted and shuffled by each one of the players. Here, it is also used Fernet for the encryption.

By the end of this process the deck will be encrypted by the server, encrypted and shuffled by all the players and only then is ready to be distributed, as shown in the next scheme.

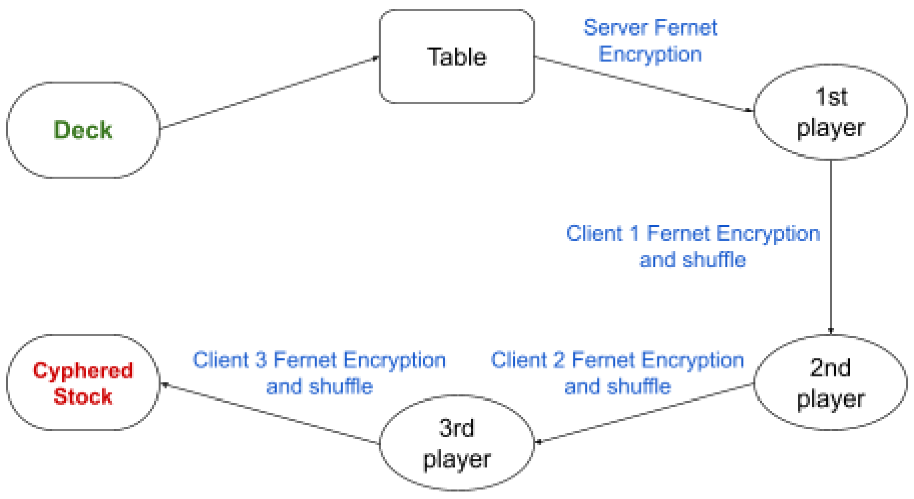
Table
Server Fernet Encryption
Deck
1st player
Client 1 Fernet Encryption and shuffle
Cyphered Stock
Client 3 Fernet Encryption and shuffle
Client 2 Fernet Encryption and shuffle
2nd player
3rd player

Fig. 1 Deck Encryption and shuffle

In the beginning of the game, all three players receive five domino pieces. Server delivers the five pieces to each player and now each one of them must decrypt their hand using their public keys and the ones of the other players. After this process is completed, all three players must have in their hand a fully decrypted set of domino pieces.

The following block of code is the idea of the decryption in reverse order taking into account all keys and the way the deck was ciphered.

```
for keys in reversed(self.allkeys[1:]):
      f = Fernet(keys)
      cleartext = f.decrypt(cipheredtext)
      cipheredtext = cleartext
```

For visualization purposes, each piece is represented by a pair of numbers between square brackets and separated by comma. *E.g* a tile that has one dot in one end and four dots in the other end will be represented by “[1, 4]”.

Picking of Stock Tiles

"In a draw game, players are additionally allowed to pick as many tiles as desired from the stock before playing a tile, and they are not allowed to pass before the stock is empty."

Our approach to this game *nuance* was to allow the players to ask the table for tiles every time they do not have a suitable tile given the current state of the table.

The players ask for a piece and the table answers with the available pieces in the deck.

To prevent colluding, the server is not supposed to select a tile for the player, the cyphered deck is made available for the player to choose which tile it will pick. However, even the player having the "free will" to choose the tile, it must be a blind pick. So, and to simulate graphically a tile turned upside down, the palette displayed to the player is composed with sets of empty pairs of square brackets.

The player chooses an arbitrary tile from the available deck until it gets a suitable one to play. In the process, the server notices the player when it delivers the tile and the client notices the server when the tile is received, if received properly.

Validation of the Tiles

To assure honesty, every player must commit to their hand, and this is achieved by executing an *SHA256* hash function on every decrypted tile the players receive. It is done according to the shown code lines:

```
hash_object = SHA256.new(bytes(finalPiece))
bitCommit = hash_object.hexdigest()
```

The results are then sent to the server that keeps them associated with the right player, which all together end up being a type of game log of commits.

At the end of the game, to prove the legitimacy of the pieces they played, the table manager applies the same hash to the tiles the player played, and if they match the ones he received and saved every time a player received a piece, we can assure the player did play the tiles he committed to.

When a player accuses someone of cheating, a similar procedure could be executed, where the cheating player played tiles would be hashed and compared to the commits he gave after receiving every piece, and it would prove a disparity. We ended up not doing the comparison in this case, due to the fact that it would be redundant. Whenever a cheating player is accused of cheating, there's already enough proof on the board. It would only be useful if someone was wrongly accused of cheating, which is not implemented in the game at the moment.

It was opted to proceed with this strategy due to the fact that if the player only committed the initial hand, and was accused of cheating, there was a possibility that the tile he was accused of playing illegally was actually played legally, but was only drew mid-game, from the stock, looking for a piece to play, therefore being both obtained and played legally, but with no way to assure it. So, due to the fact that players can draw tiles mid-game, we had to adapt the commit execution.

The overall procedure leads to a weakness discussed in the "*Known Issues and Weaknesses*" section.

Cheating and Protesting Cheating

Both cheating and protesting against cheating are implemented in this game version. When a player does not possess a valid tile to play, there is a chance they will cheat. This chance is predefined in the client configuration, and can go from 0, where the player never cheats, to 100, where the player cheats every time it does not have a tile to play. The cheating method is very straightforward. The client drops a piece from its hand, because if it is about to cheat, it cannot keep the same number of tiles before and after cheating. Then, a new tile is “created”, by joining the right side of the last tile in the table, with the left side of the first tile in the table, originating something like the following scheme.

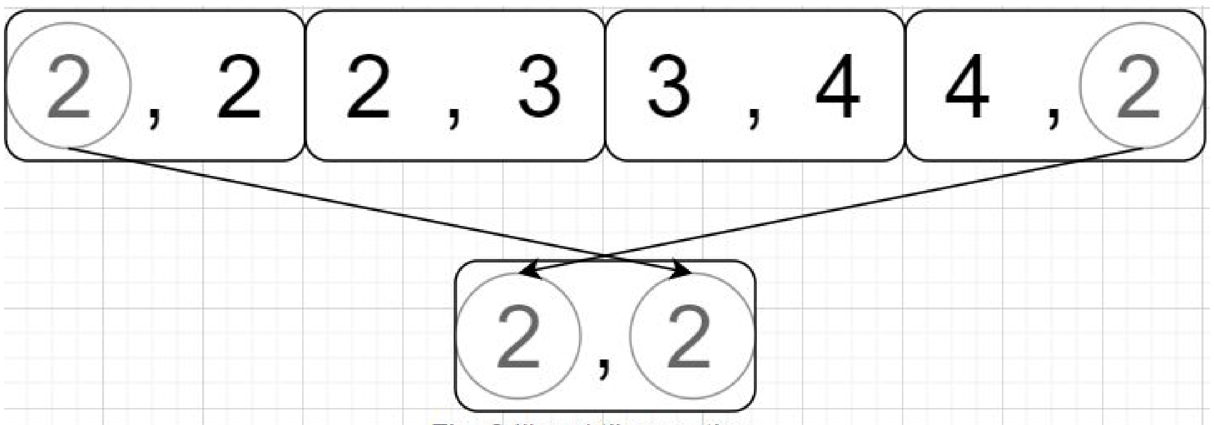

Fig. 2 Illegal tile creation

The new tile, originated illegally, will be immediately played to the end of the table, originating something like:

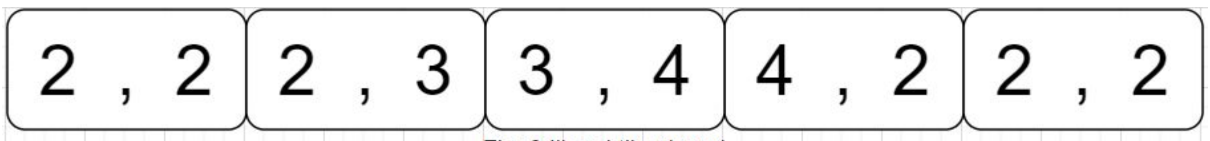

Fig. 3 Illegal tile played

This method is not the smartest possible in terms of avoiding detection, but will always allow the created piece to be played.

After the tile is played, the server will ask everyone if everything is ok, or if any suspicious activity is detected. Here, the other players will perform a quick checkup on the tiles on the table. Checking for cheating consists in 2 mechanisms: observing the table and finding 2 duplicate pieces, or observing the table and finding a piece that is also on the player’s hand. If there are no tiles repeated, the game proceeds normally. If there are tiles repeated, a message is sent to the server that confirms the cheating, and aborts the game. This can be confirmed manually by either looking at the table to find the 2 repeated pieces, or by looking at the player’s hand and finding a piece that was played by someone else and is illegally on the table.

Game Accounting

The game accounting is done through a file. Every time a game ends, if there is a winner, it will be awarded 3 points. In the case of a draw, every player will be awarded 1 point.

Citizen Card

When the client program initiates, it is asked if the user wants to authenticate using the Portuguese Citizen Card. If the user wants to authenticate itself with it, a set of tasks and verifications are made to prove that that Citizen Card is valid to use.

First, there is a use of the *PKCS#11* interface to use the cryptographic token and retrieve its useful certificates. With them, it is possible to find its private key to sign the message to send the user's *RSA* Public Key, generated when the game was initiated, and also the Citizen Card certificates. Whenever a message is signed with the Citizen Card's private key, a PIN code is asked. *RSA* keys are generated and sent along with Citizen Card's certificates to, after the user passed in its authentication test, replace its Citizen Card's private key to *RSA* private key to sign the rest of the messages exchanged during the game, because it would be annoying to enter the PIN code every time a message is exchanged.

The following code represents the retrieval of Citizen Card's private key, after getting its certificates:

```
privKey = session.findObjects([( CKA_CLASS, CKO_PRIVATE_KEY ),
( CKA_LABEL , 'CITIZEN AUTHENTICATION KEY' )])[0]
```

Below there is an example of the initial message being sent signed with the Citizen Card's private key with Citizen Card's list of certificates and user's generated RSA public key.

```
msg = {
    "name": self.name,
    "cc_auth" : True,
    "certs" : certlist,
    "rsa_public" : self.rsa_public.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
}
msg.update({"sign" : bytes(session.sign(privKey,
pickle.dumps(msg),           Mechanism(CKM_SHA1_RSA_PKCS)))})
self.s.sendall(pickle.dumps(msg))
```

On the server side, it checks the authentication method. If it receives a message saying that the client wants to authenticate with the Citizen Card, then the server proceeds to load its public key by retrieving from the Citizen Authentication Certificate.

```
pubKey = x509.load_pem_x509_certificate(data["certs"][0],
default_backend()).public_key()
```

With that public key, the server is able to check if the message was really sent by the user that wants to really join the game, by verifying the digital signature of the message.

Before that, the server checks if there is already a user with the same Citizen Card's public key. If there is, the server messages the client to shutdown and try again (if they want), signed with its private key loaded from the PEM file.

The last verification of the Citizen Card authentication method is the Chain of Trust of it. It requires validation from the end entity all the way up to the root certificate. That is accomplished by saving the Subject names of the list of certificates that the Citizen Card have, and comparing them to the issuer of those certificates. The last verification serves to prove if the last certificate is equal to the last certificate subject.

If some verification fails, the Citizen Card is not trusted and the user exits the game.

If all verifications are successful, the user is able to join the game. The server saves its details (the name, internet address, Citizen Card's public key and its RSA public key). Also, the server sends its Public Key for future digital signature verifyings on the client side and waits for the user to send its AES key, for future encryptions.

Sessions between Players and Table Manager

Secure sessions were established between players and the table manager (the server) by implementing an authentic channel between them. We do not ensure its confidentiality (see topic *Sessions between Players*, in *Known Issues and Weaknesses* for that) but its authenticity is guaranteed.

In order to implement that channel, the sender must encrypt the content of the message and sign it with its RSA private key. The receiver verifies the origin of the message with the sender's Public Key and decrypts the content.

The encryption and decryption is made by using the AES cipher. Both server and clients use the client's AES key to do that.

An example of a digital signature in the server is exemplified below.

```
 msg={
     'play':Fernet(self.players[player]['aes_key'])
            .encrypt(pickle.dumps(self.table))
}


msg.update({ "sign":self.privkey.sign(pickle.dumps(msg),
     padding.PSS( mgf=padding.MGF1(hashes.SHA256()),
     salt_length = padding.PSS.MAX_LENGTH),
     hashes.SHA256())
})

self.players[player]['conn'].sendall(pickle.dumps(msg))
```

Here is the verifying in the client of the message sent.

```
if 'play' in data:
    try:
        self.table =
pickle.loads(Fernet(self.aes_key).decrypt(data['play']))

        self.server_pubkey.verify(
           data['sign'], pickle.dumps({d : data[d] for d in
list(data)[:-1]}),
           padding.PSS(
                   mgf=padding.MGF1(hashes.SHA256()),
                   salt_length=padding.PSS.MAX_LENGTH
               ),
           hashes.SHA256()
       )
except:
    print("*** verification of play failed ****")
```

Known Issues and Weaknesses

Picking of stock tiles

Even though it looks like the player is choosing his own piece, in reality, when the table manager picks the “chosen” tile, he could swap the chosen by any piece he wants. It sounds like the server could manipulate the piece the player is receiving, but in the end, the server actually has no way of knowing what the pieces are without the player finding out. This means that replacing a chosen piece by a random piece is completely redundant - the player is still getting a piece that would be as random as the one he chose. It’s not perfect, but it doesn’t interfere with the game at all.

Validation of Tiles

Currently, the commit is only done after the player decrypts the tile. The core idea would be only correct if it was done while the tile was still encrypted. Right now the method used is leaving a small window for the player to decide he received another piece, and commit to that fake one. We had the intentions to implement the right way, but taken into account the fact that this functionality was one of the last ones, and there was a need for game logic change to implement it correctly, we decided to keep this version implemented anyway, providing some sense of security and closing down to a minimal window of cheating.

Game accounting and Claiming points

The first goal to this functionality was saving the points in the server, allowing to use the Citizen Card authentication to claim the points from a certain username to a certain Citizen Card.

To do that, we save the game’s history after the game finishes, which includes its players and its details, the winner of the game, the table and all *AES* user keys.

Our perspective is that the server never stops working, initializing other games, with the same users or not. To do that, a new function was created, `def reset_configs(self)` to reset some server attributes that need to be restored.

After that, there was an attempt to implement the rest of the steps needed to enable the claiming points feature, but it wasn't achieved. The next step was to ask the users if they wanted to exit or stay in the game. If they responded that they wanted to stay in the game, all of their data would be preserved in the server. There was a try to implement that, by sending a message to the server saying that they want to stay in the game, and to not reset the configs. When a player claims the points, they must prove their authenticity. They could do this in two different ways.

If they are already authenticated with the Citizen Card, all of the certificates are already verified and therefore, there's no need to authenticate the user again, it is only needed to ensure that the digital signature of the message sent claiming the points was verified correctly.

If they are not authenticated with the Citizen Card, then it is required to ask them to send the certifications to derive their Citizen Card's Public Key. That verification is very similar to the one made in the beginning of the game. The player must also sign their message with their Citizen Card's private key and there is a need to verify the authenticity of that message. After that, it is possible to look for players with the same Citizen Card's Public Keys in `self.games_history` and sum the points.

How to test

To test the project, there is a series of modules that need to be installed. They are, *Numpy*, *Cryptography*, *PyCryptodome*, *PyPKCS11*, *PyOpenSSL* and *PyBase64*. After all the modules are installed, all it takes is, inside the project directory, to run the server “python3 server.py” and three clients “python3 client.py”. Note that after one game ends, it is possible to connect another three clients and start a different game without rebooting the server. This is allowed to be done multiple times.

Conclusion

Except for the factors mentioned in the previous point, the game works correctly.

Regarding the relevant points to the security factors, we consider that we have addressed them all with some degree of complexity. The encryption and authentication of the messages exchanged, the creation of secure sessions between clients and the server, the guarantee of security and validation of the deck and the possibility and detection of cheating.

Remains the will of, perhaps, implement something more visually appealing and interactive, but taking into account the scope of the course, we decided not to waste too much time on these factors.

Thus, this project served to deepen and condense the techniques, methods and systems learned during the semester in the subject of Security.