ProjectSA – eCommerce Communication Security - Part1 BuildIt

Santiago Benites fc54392
Inês Morais fc54462
Miguel Carvalho fc54399

April 2023

1 Introduction

In this report we will give a brief overview of the general protocol used by our program to communicate between the MBec (client), Store and Bank (server). Moreover we will talk about 5 of the vulnerabilities we took into account when designing the system and how we found ways to solve them.

2 Communication Protocol

In this section we will give an overview of the general considerations all connections in our protocol use, these being

1. Hashes
2. Encryption
3. Signatures
4. Diffie Hellman
5. RollBack
6. Timeouts
7. ACK Message
8. Permanent Client

With this in mind we will do a deep dive into how we implemented each of these properties, and how they work in a general sense. In lieu of this, we will then analyse the messages sent in each features protocol and the specific messages sent down to the fields used and specific requirements

2.1 General Considerations

2.1.1 Hashes

To verify if messages transmitted were changed we used SHA-256, due to the fact that is one of the strongest hash functions available, and has not yet been compromised in any way.

2.1.2 Encryption

The process of encryption is made by resorting to AES in CFB mode, moreover we make usage of IVs when making usage of this algorithm, this adds an extra level of security. We decided upon AES considering it was on the the algorithms though in class, and we chose CFB considering the lack of padding made it easier to not bloat our messages more than they needed.

2.1.3 Signatures

To guarantee the authenticity of messages we used a pair of RSA keys. The private Key, combined with SHA-256, and PSS(Probabilistic Signature Scheme) are used to make the signature. PSS padding is the recommended padding algorithm for RSA signatures [2].

2.1.4 Diffie Hellman

When implementing our variation of Diffie Hellman we reanalysed the class slides regarding this topic and we decided upon researching alternative Diffie Hellman importations to solve the main constraints of the base version, specifically the vulnerability to Man-In-The-Middle attacks.
Taking this into account we researched multiple sources and decided upon an alternative implementation of Diffie Hellman that can deal with this type of attack named Ephemeral DH (to know more about this algorithm [1]).
We have to Versions of Diffie Hellman in our project the first is ECDHE-RSA(Elliptic Curve Diffie-Hellman Ephemeral with RSA digital signatures) that will be used between the client and Bank, and between the Store and the Bank, and the second version is ECDHE (Elliptic Curve Diffie-Hellman Ephemeral).
Elliptic Curve Cryptography is a public-key cryptography approach based on the mathematical properties of elliptic curves. It offers the same security level as traditional methods like RSA, but with smaller key sizes. This results in faster computations and reduced storage and bandwidth requirements.

Diffie-Hellman: DH is a key exchange protocol allowing two parties to establish a shared secret over an insecure channel. This shared secret can then be used to derive encryption keys for secure communication.
In ECDHE, a unique key pair is generated for each communication session. This ensures forward secrecy, which means that if a single session's key is compromised, it won't affect the security of past or future sessions.
RSA is a widely-used public-key cryptosystem that provides encryption and digital signatures. In ECDHE-RSA, RSA is used to authenticate the parties involved in the key exchange.
The ECDHE-RSA key exchange process can be summarized in the following steps:

1. Both parties agree on an elliptic curve and a base point on that curve.
2. Each party generates an ephemeral private key (a random scalar) and uses it to derive a corresponding public key by multiplying it with the base point on the elliptic curve.
3. The public keys are then exchanged.
4. Both parties use their private key and the received public key to compute a shared secret (the x-coordinate of the resulting point on the curve).
5. The shared secret is used to derive symmetric encryption keys for secure communication.
6. RSA digital signatures are used to authenticate the parties and ensure the integrity of the key exchange process.

2.1.5 RollBack

As indicated from the project description, if any message were not to be successfully completed, the server should revert the effects of that message.
With this in mind, we implemented rollback for each message, but this came at a cost vulnerability wise. An attacker can send a rollback of another message with the intent of undermining the original message.
To prevent against this, instead of making rollbacks a consequence of another message, or storing messages in the server, we made a new message type specified in rollback. That made it so that in order for message rollback to occur, the client that sent the rollback message would have to authenticate it self again with the server, and send to the server the original message it wants to rollback, this allowed only clients which had permissions to execute the original message to be able to execute the rollback message.
Moreover, when executing a rollback in order to keep protect from blocking attacks, the rollback message is sent but the party that has sent the rollback message doesn't wait for confirmation the rollback has been successfully completed.
This happens because this could lead to infinite rollback cycles where we could have rollback of rollback, and so forth.

2.1.6 Timeouts

As indicated in the project description, if any member of the system takes more than 10 seconds to send a response message, the member waiting for the response should timeout, and the other side should know the other member timed out. To implement this all our sockets have a default timeout of 10 seconds, and if they are waiting for a message for over 10 seconds they timeout, in normal execution, the other member or even the two other members of the network will know the first has timed out, and will continue execution accordingly.
Even if the client times out after a message has been sent, but before it as been completed. Both the store and the Bank know that they should now rollback the changes the message has done, this in order to keep the bank and the client synced.
Take into account this all can only be done in authenticated clients, and that this rollback message is still scrutinized like any other rollback message having the pass all verification's a normal rollback as to pass through.

2.1.7 ACK Message

In order to implement the majority of send methods we found ourselves with a problem. Given we are using TCP sockets, there was the possibility that the host part of the connection could be sending the message to a connection that had been closed by the client side (host is not always the Bank and the Client not always the Store or the MBec).
In order to solve such a problem, whenever a "host" sends a message to a connection that could have been closed, it will wait for an acknowledge message, and if said message isn't sent back it will revert the changes made with a rollback.
In order to know if the message wasn't received, the connection in which it waits for the ACK message is the same as the one it send the original message through and as such will have the same timeout, therefore a malicious attacker cant just deny the ACK message to try to block the server.

2.2 Permanent Client

During the implementation of the project, and after the explanations given in the professor's office hours, we decided to use a permanent client which instead of taking the input from the command line takes input from the stdin. This makes no difference from a practical standpoint considering the client doesn't keep state during the execution except for a variable called lastUsedAccount which is used in order to detect the last credit card used (a consideration made by the professor), and event this variable is optional considering the client can work without using it (it will just obtain the card with the highest index).

2.3 Protocols

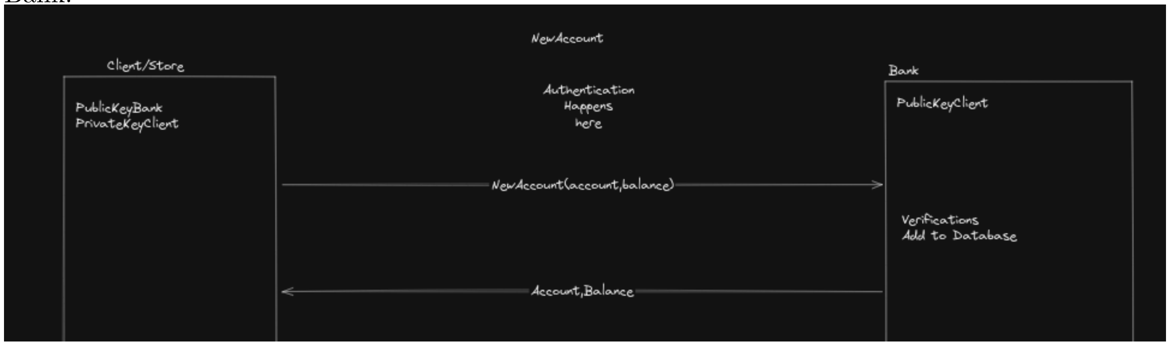
In this section we will consider that the auth file is a self signed certificate RSA public key of the bank, and the user file is a private of the user. Before the store or MBec send a message to the bank, it will occur mutual authentication, first from the MBec or store and then from the bank. The authentication occurs a follows:

1. Client get public key of Bank using the auth file.
2. Client sends message with the account number encrypted with the public key of the Server, and the public key in clear, the message will be hashed.
3. Server receives message and verifies the hash, and using his private key decrypts the account number, and verifies if already has a public Key associated with the account number, if it does will use this key in the following verification's. And sends to the client a nonce with a timestamp.
4. Client receives nonce and verifies the timestamp validity, if is valid sends nonce signed with his private key.
5. Server verifies signature using the public key of the client. If verifies sends OK to client.
6. When receiving Ok, client will send a nonce with a timestamp to Server.
7. Server verifies timestamp, and signs nonce with his private e sends to client
8. Client verifies signature and sends OK.

The messages exchange after authentication are signed with a RSA private key and have hashes using SHA-256.
The messages exchange between the store or MBec and the bank, will have the following structure {HashedMessage: {message: signature: } hash: } and contain timestamps.
The receiver will verify the hash, signature, timestamp validity of the message received, and other necessary verifications.

2.3.1 NewAccount

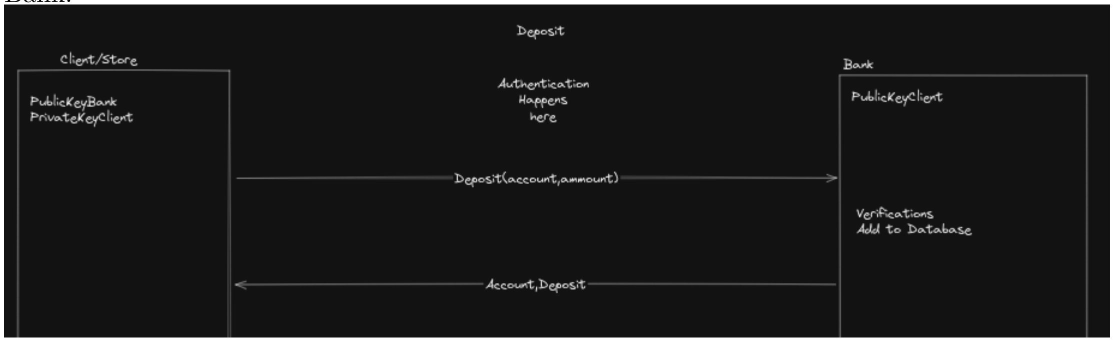
In this mode is created the RSA pair of keys used by the MBec, if register in server succeed the MBec will save the private key in the user file. And the bank will store the public key of the client associated with the account number and the initial balance of the account. This the public Key is the one that the Bank will use to compare the signature of the message will be send after.
Message send to server has the following structure: {"MessageType": "NewAccount", "account": account, "balance": balance, "timeStamp": } .
The message received from the server will have the following structure { "account":accountName, "initial_balance":balance, "timeStamp":)} or { "Error": , "timeStamp":}
If hash of the message received by the client does not match message, the client will send a rollback message to the Bank.



2.3.2 DepositMode

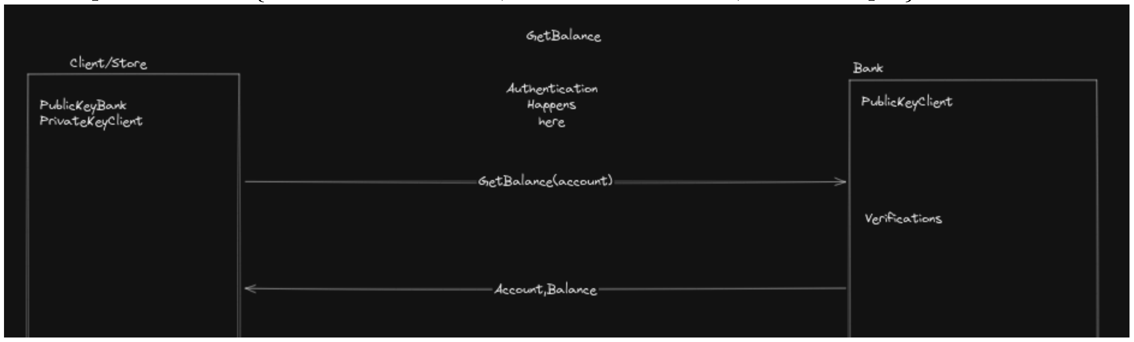
In this mode the client will read from the user file his private key, and use it to sign the message that will be sent to server. The message send from MBec will have the following structure: {"MessageType": "Deposit", "Amount":amount, "account":account, "timeStamp": }

The Server will obtain the public key associates with the account and use it to verify the signatures of the message send. And update the balance of the account.
The response send from the server will be: {"account":account, "deposit":deposit, "timeStamp":} or { "Error": , "timeStamp":}
If hash of the message received by the client does not match message, the client will send a rollback message to the Bank.



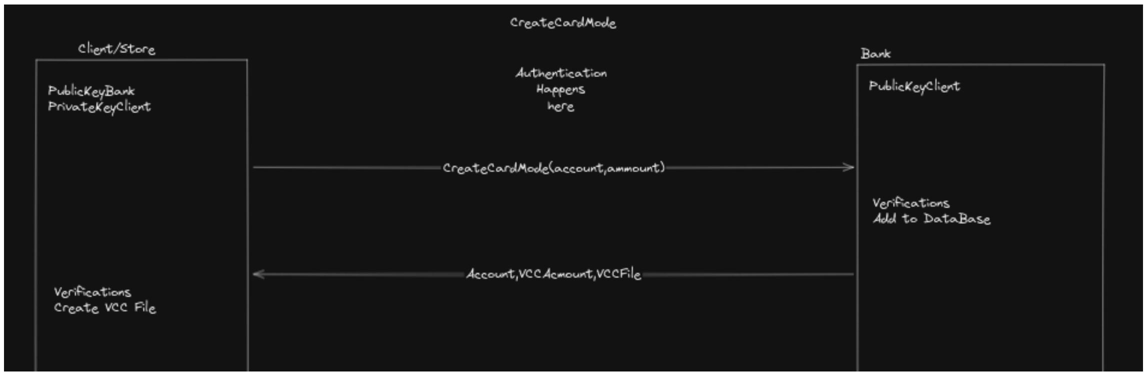
2.3.3 GetBalance

In this mode the client will read from the user file his private key, and use it to sign the message that will be sent to server. The message send from MBec will have the following structure: {"MessageType": "Balance", "account":account, "timeStamp": }
The Server will obtain the public key associates with the account and use it to verify the signatures of the message send. And obtain the balance of the account.
The response will be { "account": account, "balance": balance,"timeStamp":}



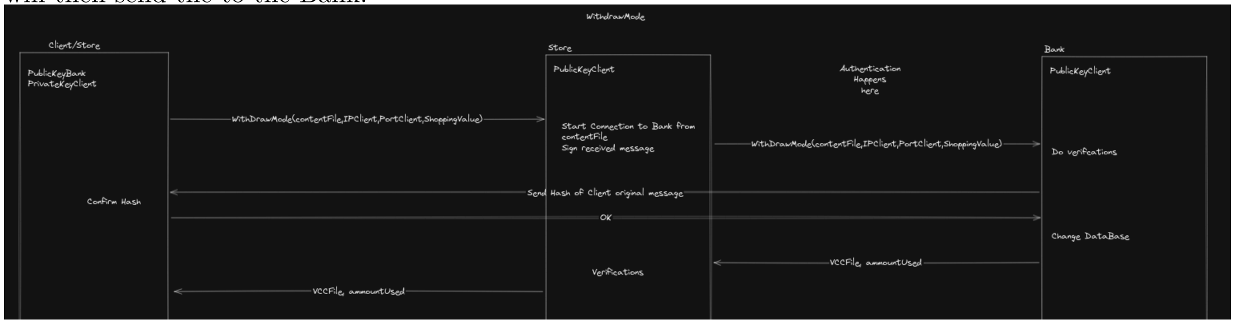
2.3.4 CreateCardMode

In this mode the client will read from the user file his private key, and use it to sign the message that will be sent to server.
The message send to server will be as follows: { "MessageType": "CreateCard", "account": account, "amount": amount, "timeStamp": } .
The Server will obtain the public key associates with the account and use it to verify the signatures of the message send. Then it will verify the validity of the amount, and creates a active card associated with the account with the given amount.And send a message with the following information. {"account":accountName, "vcc_amount":amount, "vcc_file": cardName.card, "timeStamp": }
When receiving message client will perform the necessary verification's. After this client will create a vcc_file with the name cardName.card, the file will contain the following {"ip": ipBankAddress, "port": bkPort, "message":messageEncripedPublicKeyBank, "signature": signature } where ipBankAddress and bkPort are the ip and the port of the Bank that store should contact. messageEncripedPublicKeyBank is the message that the client received from the bank encrypted with the public key of the bank, signature is a signature of the messageEncripedPublicKeyBank using a private key to client.
Encrypts with the public key of the bank will make sure that the only server capable of reading the file will be the Bank. Signing the message allows to verify the identity of the client that created the card.



2.3.5 WithdrawMode

In this operation we do not have access to the auth file or the user file. The Message send to the store will only have hashes, and will not be signed. The message as the following structure: { "MessageType": "WithdrawCard", "contentFile": p, "ShoppingValue": shoppingValue, "IPClient": ip , "portClient":port, "timeStamp": } where p in the content of the vcc_file, ip and port are the ip and the port where the client is listen, The ip and port of the client can be changed, to a malicious client.
The Store will verify the timestamp, and will access the contentFile and read the ip and port of the bank that is supposed to connect. The Store and the bank will mutual authenticate, and then the Store will send the following message.
{"MessageType": "WithdrawCard", "messageClient": message, "timeStamp": } where the message is the message sent originally by the client. The stored will sign the message with his private key and add a hash.
When receiving the message the Bank will verify the signature and the hash of the message, and the timestamp of the message send by the client and the store. The Bank will decrypt the contentFile present in the message send by the client to the store, this away the bank will be able to access the account number, the vcc_file and the vcc_amount, using the account number will access the Public Key associated with account and verify the signature present in the contentFile. The Bank will also verify if the shopping value is valid and if the card is active.
The Bank will calculate the hash of the message sent by the client, after this will access the ip:port where the client is accepting connection, and send it to the client. The client will compare the hash send by the bank with the hash that this as calculated before sending the message. If hashes match client will send "ok" to the bank and the bank will authorize the transaction, and update the data of the account and the card.
The bank will send the following message to the store {"vcc_file": vcc_file , "vcc_amount_used": shoppingValue,"timeStamp": }, signed with the private key of the bank, and with hashes.
The Store will verify the signature, hash and timestamp of the message, if hash does not match will send a RollBack Message to the Bank and a Error message to the client.
The Store will had a timestamp to the message send by the bank and send it to the Client. Client will verify the hash and the timeStamps of the message. If hash does not match will send a RollBack message to the store that will then send the to the Bank.



3 Attacks

3.1 Replay Attacks

Replay attacks are a type of network security attack in which an attacker intercepts and replays valid data transmissions between two parties to gain unauthorized access to a network or system.
In a typical replay attack scenario, an attacker intercepts valid data packets transmitted between two parties, and then replays those packets at a later time. This can allow the attacker to bypass security measures that rely on the assumption that a given transmission is valid only if it is unique and not previously transmitted.
In order to defend against this type of attacks we implemented timestamps, a common defense against replay attacks.
For this purpose we used the following functions (present in Cripto.py):

```
def getTimeStamp():
    dt = datetime.datetime.now()
    return datetime.datetime.timestamp(dt)

def verifyTimeStampValidity(ClientTimeStamp):
    dt = getTimeStamp()
    #10 min
    return (dt-ClientTimeStamp) < 600
```

Whenever a communication made between 2 members of the network, the sender adds to the message an attribute where the timestamp is store and when the receiver receives it it checks the timestamp and if it has expired it will decline the message.
Take into account we hard coded the expiration to 10 minutes, a value we considered reasonable.

3.2 Personification Attacks

Personification attacks are a type of cyber attack where an attacker impersonates a legitimate user or system to gain unauthorized access to sensitive data or resources.
In a typical personification attack scenario, the attacker will use various techniques to obtain the login credentials of a legitimate user or administrator. Once the attacker has obtained these credentials, they can use them to impersonate the legitimate user or system and gain access to sensitive data or resources.
To defend against this type of attack we implemented 2 main defenses

1. Signatures
2. Authentication

To prevent personification, all relevant messages were signed with the private key of the sender we were expected to be communicating with.
Therefore in order to check that a certain message came from a verified source we only had to have his public key to verify the signature, this could easily be done for the bank considering we already had his public key in the bank authentication file, but for the client the process was different.
The code responsible for signing and verifying signatures (present in Cripto.py):

```
def verifySignature(publicKey, signature, message):
    try:
        publicKey.verify(
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
            )
    except cryptography.exceptions.InvalidSignature:
        return False
    return True
```

```
def signwithPrivateKey(privateKey, message):
    return privateKey.sign(
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
```

To apply the same principle to the client, we made sure that whenever we were creating a new account with the bank, we passed along with the new account a user file in which we stored the clients public key.
This could after be used to authenticate the messages the client sent, and verify the messages signature.
Obviously we couldn't send the clients public key in clear, so whenever the client wanted to register and account it would encrypt the message with the banks public key, so that only him could see it.
After this we did the "normal" routine of authentication of 2 parties, when one sends to the other a nonce and waits for the other party to sign it with his private key and the same the other way around. Only after all of this could we be 100% sure we were talking to the other party.
The code responsible for this in the MBec (present in MBecCommunication.py) is the following (we didn't present its counter part because it very similar):

```
    #Authentication of MBEC

        nonceReceived = s.recv(1024)
        nonceMsg = pickle.loads(nonceReceived)

        if "nonce" not in nonceMsg or "timeStamp" not in nonceMsg:
            s.close()
            return None

        if not verifyTimeStampValidity(nonceMsg["timeStamp"]):
            s.close()
            return None

        s.sendall(signwithPrivateKey(privateKey,nonceReceived))
        if(s.recv(1024).decode()!="OK"):
            s.close()
            return None


    #Authenticates Bank
        nonce = secrets.token_bytes(100)
        nonceMSg = pickle.dumps({"nonce": nonce, "timeStamp": getTimeStamp()})

        s.sendall(nonceMSg)
        nounceSigned = s.recv(1024)
        #verify signature from server
        if not verifySignature(publicKeyBank,nounceSigned,nonceMSg):
            #s.sendall("NOK".encode())
            s.close()
            return None
        s.sendall("OK".encode())
```

3.3 MITM Attacks

A Man-in-the-Middle (MITM) attack is a type of cyber attack where an attacker intercepts communication between two parties in order to eavesdrop, steal information, or manipulate the communication.

In a typical MITM attack scenario, the attacker positions themselves between the two parties, intercepting and potentially modifying the communication as it passes through.
To defend against this type of attack we implemented

1. Authentication
2. Encryption
3. Ephemeral Diffie Hellman

As the first 2 were already explained above we will now go into dept regarding ephemeral Diffie Hellman (EDH).
This algorithm as the name implies is a variation of Diffie Hellman, it main purpose is to solve the DH biggest vulnerability the possibility of MITM attacks.
EDH solves this vulnerability by generating a new set of public and private keys for each session. This means that even if an attacker intercepts the key exchange, they cannot use the keys to decrypt any previous or future communications.
Moreover our implementation of EDH also makes sure to sign message, a recommendation we found in some of the forums in which we researched, this alteration is meant to added an extra layer of security against MITM attacks considering with this we can make sure we are communicating with the intended party.
The following is the Implementation of the EDH in the Bank without verification's, this because the entire function is too big for a report (it can be found in BankConnection.py):

```
def ephemeralEllipticCurveDiffieHellmanReceiving(connection,PublicKeyClient,privateKey):

    # Creating Elliptic Curve Public Key
    private_key = ec.generate_private_key(ec.SECP384R1())
    public_key = private_key.public_key().public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    signedPublicKey = signwithPrivateKey(privateKey,public_key)

    hasedMessage = pickle.loads(connection.recv(5000))

    signMessage = pickle.loads(hasedMessage["messageHashed"])

    signed_client_public_key_bytes,client_public_key_bytes =signMessage["signedPublicKey"], s
    client_public_key = serialization.load_pem_public_key(client_public_key_bytes)

    # Generate a shared secret
    shared_secret = private_key.exchange(ec.ECDH(), client_public_key)

    # Derive a key from the shared secret
    derived_key = ConcatKDFHash(
        algorithm=hashes.SHA256(),
        length=32,
        #salt=None,
        otherinfo=None
    ).derive(shared_secret)

    signMsg = pickle.dumps({"signedPublicKey":signedPublicKey,"public_key":public_key})
    return derived_key
```

3.4 Path Transversal Vulnerability

Path Traversal Vulnerability is a type of attack that exploits a vulnerability in applications that accept user input and use it to access files or directories on the server. In a Path Traversal attack, an attacker uses a crafted input to navigate outside of the intended directory structure and access files or directories that are not intended to be accessible.
In order to defend this type of vulnerability as recommended in the project description we made sure to sanitize the input the server could get from messages.

Specifically we made sure that the name of files we are trying to access is only compromised of underscores, hyphens, dots, digits, and lowercase alphabetical characters.
This allowed us to remove problematic characters like .(dots)
(Backwards slashes) amongst others.
The code responsible to defend against this type of vulnerability is (found in utils.py):

```
def argsAreValidFileNames(str:str):
    validSize = 1 < len(str) < 127
    validChars = True if re.search("[\_\-\.0-9a-z]+",str) else False
    notDots = str != "." and str != ".."
    singleMatch = len(re.findall("[\_\-\.0-9a-z]+",str)) == 1
    return validSize and validChars and notDots and singleMatch
```

3.5 Eavesdropping

Eavesdropping attack is a type of cyber attack in which an attacker intercepts and monitors network traffic or communication between two parties without their knowledge or consent. The goal of an eavesdropping attack is to obtain sensitive information such as login credentials, credit card numbers, or other confidential data.
To prevent this type of attack we implemented encryption into our all our communications. As mentioned above encryption was done using the AES algorithm in CFB mode and using IV's.
We made usage of the Cripto python package to implement AES.
The specific template we used as base of all our calls to AES was an example from the Cripto website with some small alterations made by us, the example code was:

```
# Receive the encrypted data (including the IV) from the server
ciphertext = s.recv(1024)

# Extract the IV and ciphertext from the received data
iv = ciphertext[:AES.block_size]
ciphertext = ciphertext[AES.block_size:]

# Decrypt the ciphertext using AES-CFB with PKCS7 padding
cipher = AES.new(key=derived_key, mode=AES.MODE_CFB, iv=iv)
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)

# Print the decrypted message
print(plaintext.decode('utf-8'))
```

4 Conclusion

In sum, during the course of this project we implemented 5 different operations, those being:

1. NewAccount
2. DepositMode
3. GetBalance
4. CreateCardMode
5. WithdrawMode

Each of these had specific caveats, either regarding security constraints, efficiency constrains or problems regarding implementation.
In a nutshell, even though we took some creative freedoms regarding our interpretation of the project we believe it to be secure.
Lastly, we believe we defended against more attacks but the ones referenced above are the most significant ones, and the ones we believe give a best overview of the entire code base.

References

[1] Ephemeral Diffie Hellman Stack Exchange Explanation
[2] RFC3447