INFORMATION SYSTEM SECURITY

PROJECT – REPORT

PEER-TO-PEER CHATTING – SSL

" By providing accessibility of the integrity of the confidentiality of information; Information owners are responsible. " (A hacker proverb)

UFUK GÜRBÜZ – 150113058

MUSTAFA YEMURAL – 150113053

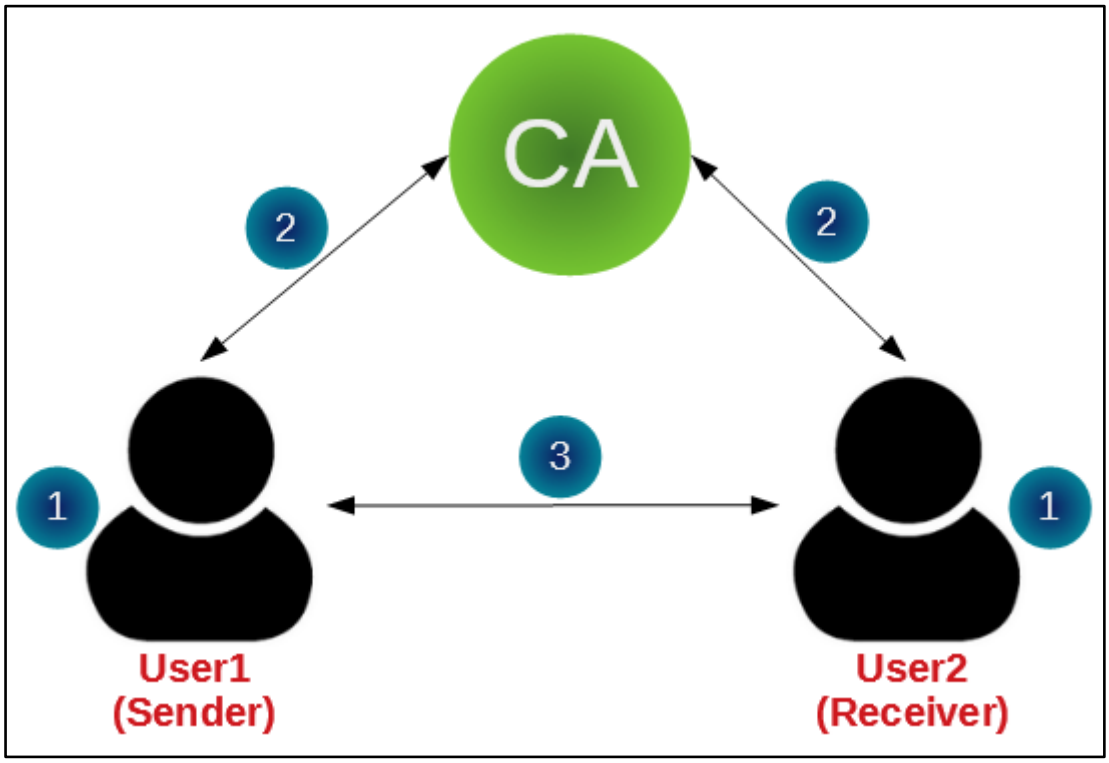
1. INTRODUCTION

In this project, in addition to the "P2P Chatting Application Protocol" project we developed in CSE4074 course, we developed a secure messaging system and integrated it into the other project. In CSE4074 project we also developed group chatting system. We have developed our security system without ever breaking this. The intent here is that our project guarantees the following security features:

- ✓ Message Confidentiality
- ✓ Peer Authentication
- ✓ Message Integrity

In addition, Public Key Authentication is provided and Replay Attacks are prevented. Public Key Authentication is provided by the Server that we use on the other project and that contains user information and Log information. We will name this Server Certificate Authority (CA) here. Instead of group chatting, the narration will be based on the conversation of 2 people. Information about group chatting will then be provided.

First of all, let's look at our basic components and our fluidics:



Here, there is a “sender”, a “receiver”, and a system or organization (“CA”) that provides a ‘Public Key Authentication’. In general terms, there are 3 transactions here:

1) Generating Public/Private Key Pair
2) Public Key Authentication
3) Handshaking and Messaging

We will now describe how we implement these three operations in the Project Phases section and how we provide security.

2. PROJECT PHASES

Our project has 6 stages. We have detailed information and design choices about what we do at each stage. Our project is written in Python 2.7 language, and for all processes, 2.1.4 version of "cryptography" module written for Python and containing most functions related to cryptography is used. You can find detailed information about this module **here**.

2.1) Public Key Certification

Firstly, we needed to set a standard for the certificate structure. We used X.509 format for this. A certificate created with this format contains the following information:

- **Version:** It is the X.509 version that the certificate accepts.
- **Serial Number:** Serial number given to distinguish the serise from others.
- **Algorithm Information:** Algorithm information used in signing the certificate.
- **Validity Period:** The validity period of the certificate.
- **Subject Information:** Information about the owner of the certificate.
- **Subject Public Key:** The certificate is the owner's Public Key.
- **Extensions (optional):** It contains additional information about the certificate.

The CA must obtain and sign a Certificate Signing Request (CSR) object to generate the certificate. The CSR object is the document that contains the user information and the user's Public Key and is also signed with the user's Private Key. This document is sent to CA and verifies this document. It then puts the user information and the user's Public Key into the certificate document. This document also sends its own private key to the signatures and the corresponding user. The user opens it with the CA's Public Key and obtains the verified Public Key. The standard file format ".pem" extension is used for all these operations.

We have created a module for all the cryptography processes we will use in our project, and we have collected all the functions on this module. We used the RSA algorithm as Assymetric Cryptography in our project. With this algorithm, each user created their own Public / Private Key pair and saved them with the ".pem" extension. Then, with the function below, users created their CSR files using their own information and Keys:

```
def generateCSRFile(fileName, privateKey, orgName, commonName, countryName=u"TR", localityName=u"Istanbul"):
    # Generate a CSR
    # All string parameters are unicode type
    csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
        # Provide various details about who we are.
        x509.NameAttribute(NameOID.COUNTRY_NAME, countryName),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"CA"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, localityName),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, orgName),
        x509.NameAttribute(NameOID.COMMON_NAME, commonName),
    ])).sign(privateKey, hashes.SHA256(), default_backend())

    # Write our CSR out to disk.
    with open(fileName, "wb") as f:
        f.write(csr.public_bytes(serialization.Encoding.PEM))

    return csr
```

Thus, a CSR file was created on the user's computer as well as the value that returned from this function and the object holding the CSR document. An example use of the function is as follows:

```
generateCSRFile("csr.pem", private_key, orgName=u"Ultimate",commonName=u"Mustafa")
```

Then we sent this file to CA. As soon as our Public Key was already in the file, we could easily confirm it. The CA collected the information and the Public Key in the certificate object and encrypted it with their Private Key. The function that does this on the server side and creates both the certificate ".pem" file and the certificate object is:

```
def generateCertificateFile(fileName, csr, subjectPublicKey, privateKeyCA, validationDay=10):

    subject = issuer = csr.subject

    cert = x509.CertificateBuilder().subject_name(
        subject
    ).issuer_name(
        issuer
    ).public_key(
        subjectPublicKey
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.datetime.utcnow()
    ).not_valid_after(
        # Our certificate will be valid for x days
        datetime.datetime.utcnow() + datetime.timedelta(days=validationDay)
    ).sign(privateKeyCA, hashes.SHA256(), default_backend())

    with open(fileName, "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

    return cert
```

An example use of this function is as follows:

```
cert = generateCertificateFile("certificate.pem", csr2, public_key, ca_private_key)
```

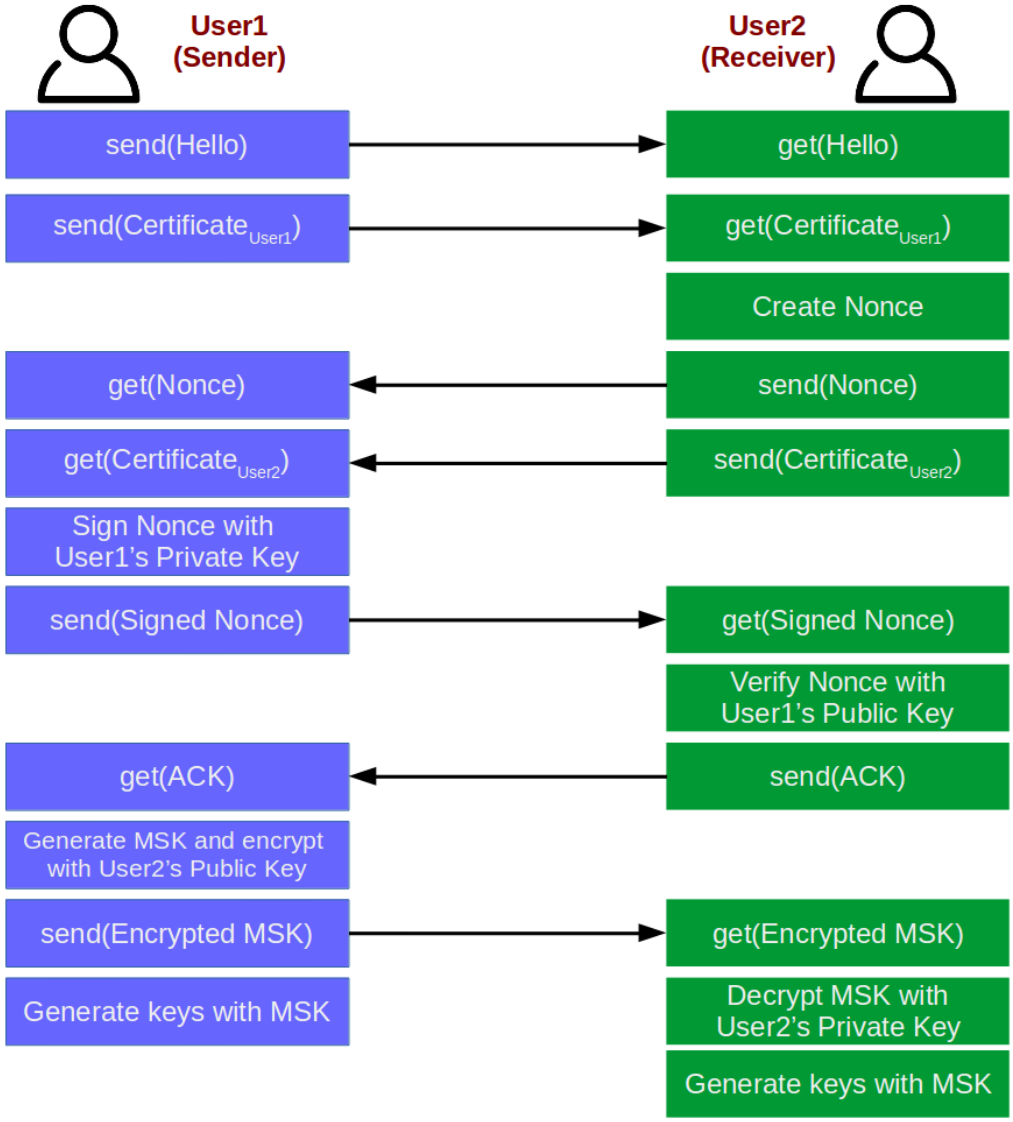
We send the certificate that we created with this function to the related user again and register it in the Server's Database. In this way, the database also includes certificates associated with users. After each user received his certificate file, he verified it and kept the verified certificate on his computer. Thus, Public Key Authentication was provided. Now we could send this generated certificate to the people we want to communicate with.

2.2) Handshaking

Our goal in this phase is to initiate communication between the two users and to realize the production of the Keys by bringing together the certified certificates of the users. Authentication of peers must be provided here. For this reason, the recipient must create a nonce and send it to the sender, the person who initiated the conversation. The sender must also sign it with their Private Key and send it back. At this stage, the recipient already has the Public Key of the sender, which confirms that this is a message coming from the sender and compares the nonce.

Nonces are involved in the creation of keys as well as providing authentication of peers. While generating Common Keys, a random number called Master Secret Key (MSK) is used. The MSK is initially created on the sender side and encrypted with the recipient's Public Key and sent to the recipient. The receiver also opens it so that there are the same CCSs and the same nonce on both sides. This way, both parties can start key generation.

All the events in this phase are shown in the picture below. The "send" statement in the picture indicates that the object in brackets is to be sent across, while the "get" statement specifies that the object used in parentheses is to be rested.



We did the same in our project. We have created the 'MSK' and 'nonce' in the following way:

```
def getRandomValue(length):
    return os.urandom(length)

nonce = getRandomValue(16)
MSK = getRandomValue(32)
```

We then used the Public Keys to send the 'nonce' and 'MSK' we created. In this way, both sides had the same 'nonce' and "MSK".

2.3) Key Generation

Since both sides have the same MSK and nonce, we need to insert them into a Key Derivation Function (KDF) to create the keys needed for MAC and encryption. In addition, when we use a symmetric key algorithm in CBC mode, we need to create the Initialization Vector and change it for each message. We created the first IV with MSK and the rest with messages. All the keys we create are as follows:

```
hmacKeySender = keyDerivation(MSK[0:8], nonce, size=hashes.SHA256().digest_size)
hmacKeyReceiver = keyDerivation(MSK[4:12], nonce, size=hashes.SHA256().digest_size)
aesKeySender = keyDerivation(MSK[8:16], nonce, size=32)
aesKeyReceiver = keyDerivation(MSK[12:20], nonce, size=32)
ivKeySender = keyDerivation(MSK[16:], nonce, size=16)
ivKeyReceiver = keyDerivation(MSK[20:], nonce, size=16)
```

The "hmacKey" key here is the key we will use in HMAC, the Hash-based Message Authentication Code. "aesKey" and "ivKey" are the keys we use in AES encryption. Names ending with "Sender" are the keys to the messages sent by the person who initiated the conversation, while names starting with "Receiver" are the keys of messages sent by the person who verified the conversation. Certain parts of the 'MSK' and nonce are used in the production of these keys. There are also length values of the keys generated in the last parameter. When we use SHA-2 (SHA256) in the HMAC algorithm, we say that we will generate as much key as the size of this algorithm. The function that generates these keys is shudder:

```
def keyDerivation(key, salt, size=16):
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=size,
        salt=salt,
        info=None,
        backend=default_backend()
    )
    resultKey = hkdf.derive(key)

    return resultKey
```

The "salt" parameter in this is a seed that specifies the mixing order of the random key to be generated. That is, the same key and the same "salt" value are formed. This ensures that the same keys are created on both sides.

We used AES256 as the Symmetric cryptography algorithm, so we created a 32 byte key. We also ran this algorithm in CBC mode and let it get the IV value. The IV value should be 16 bytes long due to the algorithm we use. This means that encryption will take place in blocks of 16 bytes. Also, the value of IV must change in each message. We will explain in detail the methods we have designed for this in the next phase.

2.4) Message Encryption

In our project, message encryption is provided by AES256. For this reason, 32 byte key and 16 byte IV are used as the key length. If the IV we are using is 16 bytes in size, it is mandatory that the message to be encrypted be 16 bytes and multiples. However, we have used the PKCS7 algorithm from the padding algorithms and fixed each message fragment to 16 bytes and multiples. This algorithm can rearrange the bytes into multiples of a certain number. For example, if the incoming message is 38 bytes, it can convert it to 48 bytes. This converted message is then encrypted.

After you open the encrypted message, we do an inverse conversion with PKCS7 and get our message. Thus, we can also encrypt messages that are not a multiple of 16 bytes. In the AES algorithm, the size of the resulting message is as much as the encrypted message. However, if we do a padding operation here, the cipher resulting in the encryption will be in a different size. Since the PKCS7 algorithm regulates the incoming message according to the next pass (eg if the size of the message is 12 bytes and the 16-byte padding is used, the message returns as 16 bytes), it will not be too difficult to guess the size of the resulting message. The function we use in AES encryption is as follows:

```
def encryptAES(plainText, key, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), default_backend())
    encryptor = cipher.encryptor()

    # Padding process:
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(plainText)
    padded_data += padder.finalize()

    # Encryption:
    cipherText = encryptor.update(padded_data) + encryptor.finalize()

    return cipherText
```

As you can see from the code, the incoming message is first padded and then encrypted. The function takes symmetric key and IV values as parameters as well as plain text. Now let's look at the decryption function:

```
def decryptAES(cipherText, key, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), default_backend())
    decryptor = cipher.decryptor()

    # Decryption:
    plainText = decryptor.update(cipherText) + decryptor.finalize()

    # Unpadding process:
    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(plainText)
    data += unpadder.finalize()

    return data
```

Here, on the other hand, decryption is followed by unpadding and plain text is obtained. We talked about how to get IV. But we need to do different for each message. This will be explained later.

2.5) Integrity Check

When the message arrives, it should be checked whether the content of the message has changed. At this point, we use the MAC and the non-reversible codes. When we receive the MAC of a message, there is no way to reverse it. We can only check the integrity of the messages by taking the MAC of the same message and making a comparison. We used Hash-based MACs in our project.

Even if a single letter or a letter in a message is replaced in the hash algorithms, the hash of the message varies considerably. For this reason, there is no way to get the message out of the Hash algorithm. Only the possible message can be subjected to comparison operations knowing.

The hash function SHA-2 that we use in our project is SHA256. With this algorithm, we can create MACs with a size of 32 bytes. When we create MACs, the text we give is the combination of the MAC values. For example, if we combine the length of the message with the message itself and put it in the MAC, and we send both the message and the length information together with this MAC, the user will be able to receive and verify the hash of the combined state of the message and length using the same HMAC function. The HMAC function we wrote is below:

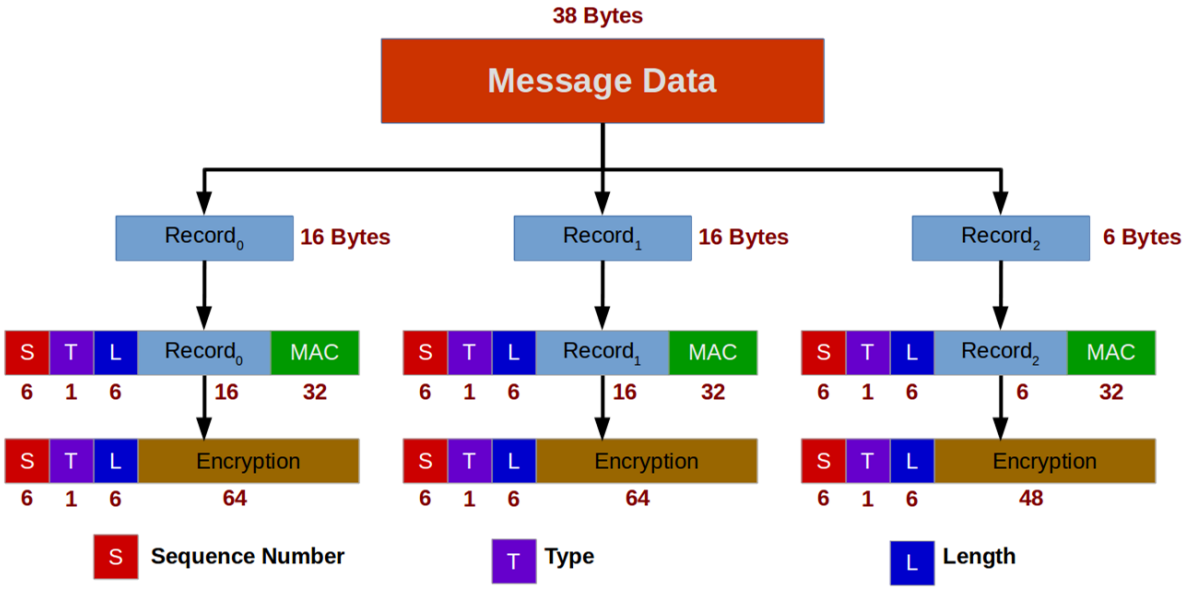
```
def getMAC(message, macKey):
    h = hmac.HMAC(macKey, hashes.SHA256(), default_backend())
    h.update(message)

    return h.finalize()
```

The value returned from this function will be the HMAC value of the incoming message. According to what we see in the information we use, now we can see how messages are divided and MAC codes are received and in which order they are packaged.

2.6) Resistance to Replay Attacks

Even if we take the MAC of the message, the messages we send out can be continuously sent and received by an attacker, or the sending order can be changed. This attack is called Replay Attack. To prevent this in our project, we fragmented the incoming message data. After dividing the message into 16 byte records, we took "length" information and added it to the side. We then put a "type" in the beginning of each record to see if the record was the last part of the message. Finally, to prevent Replay Attacks, we put a "sequence number" at the beginning of the record. Then we took all of this fragment's MAC and placed it at the end of the fragment. We combined Record and MAC to obtain an encrypted and encrypted fragment. The process until a message is encrypted and divided into encrypted fragments is summarized as follows:



The red numbers you see above indicate the byte size of each box. How to split a 38 byte message is shown. The "Sequence Number" value is incremented by 1 from the value 1. The "Type" value is 1 if the record is the last record of the record, and 0 otherwise. The "Length" value holds the size of each record. When 6 byte area is reserved for "Sequence Number" and "Length" values, 1 byte area is reserved for "Type" value.

When the first 13 bits are fixedly separated into these variables, the exact size of the other part to be encrypted is not obvious. The 32-byte MAC code is the result of combining the message, sequence number, type and length, and SHA-2 and hash. During encryption we said that pading was applied. Even though this process encounters a byte value that is a multiple of 16, such as 48 bytes, it adds 16 bytes of terminating sequences, so all data is 16 bytes longer. The last fragment has increased from 38 bytes to 48 bytes.

We designed our function to apply the necessary operations to the message and assign the fragments to a list as follows:

```
def encryptFragments(message, seqNumber, hmacKey, aesKey, ivKey, dataSize=16):
    messageLength = message.__len__()
    iteration = messageLength / dataSize

    dataArray = []

    for i in range(iteration+1):
        plainText = message[i*16:(i*16+dataSize)]
        sequenceNumberString = str(seqNumber).encode('ascii').zfill(6)
        if (iteration) == i:
            typeString = str(1).encode('ascii').zfill(1)
        else:
            typeString = str(0).encode('ascii').zfill(1)
        lengthString = str(plainText.__len__()).encode('ascii').zfill(6)

        macMessage = getMAC(plainText+sequenceNumberString+typeString+lengthString, hmacKey)

        cipherMessage = encryptAES(plainText+macMessage, aesKey, ivKey)

        dataArray.append(sequenceNumberString+typeString+lengthString+cipherMessage)

    return dataArray
```

With this function we take encrypted fragments and send them in sequence. Even if a Replay Attack occasionally occurs, the sequence number is safe with the MAC, causing problems in the validation phase and preventing the attack. Our function that opens, validates, and prints each incoming fragment is as follows:

```
def verifyFragmentGetMessage(fragment, hmacKey, aesKey, ivKey):
    fragmentSequence = fragment[0:6]
    fragmentType = fragment[6]
    fragmentLength = fragment[7:13]
    cipherText = fragment[13:]

    plainText = decryptAES(cipherText, aesKey, ivKey)
    data = plainText[0:int(fragmentLength)]
    mac = plainText[(int(fragmentLength)):]
    macNew = getMAC(data+fragmentSequence+fragmentType+fragmentLength, hmacKey)
    if mac == macNew:
        return data, int(fragmentSequence), int(fragmentType), int(fragmentLength)
    else:
        return False
```

The keys used in all these functions depend on who sent the message. The confirmation and collection of fragments and their passage into the message is as follows:

```
# Receiver-side decryption:
receivedMessage = b""
i = 0
while True:
    curMessage, curSeq, curType, curLength = verifyFragmentGetMessage(arrays[i], hmacKeySender, aesKeySender, ivKeySender)

    if curType != 1:
        receivedMessage += curMessage
    else:
        receivedMessage += curMessage
        break
    i += 1
```

Thus, our message "receivedMessage" is resolved and merged. The IV value of the next message will depend on the content of the current message.

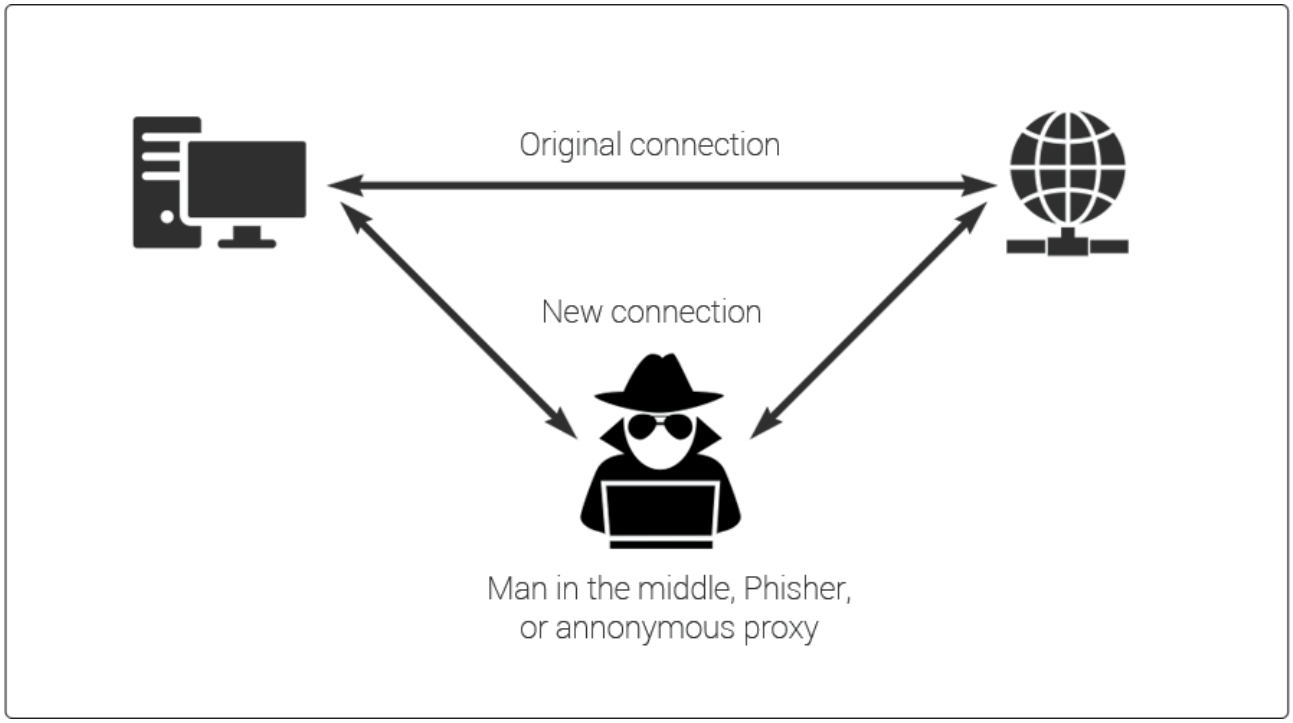
3. SECURITY HOLES AND COUNTERMEASURES

3.1) Replay Attacks

Replay attack is the attacker changing the order of outgoing packets. We have already told you how to avoid using "sequence number". Even if the sequence number of the outgoing fragments is changed by the sequence number, fragments will have a problem in the validation phase because they are contained in the sequence number MAC. This will prevent all replay attacks.

3.2) Man (or Woman) in the Middle Attack

In this type of attack, the attacker goes between two communicating parties and sends the sender's receipts to the buyer and the buyer's receipts to the sender itself. Since all of the packets go through it, they provide full control over the ciphers.



Since we provide certification using the Projemizdece, which provides public key authentication for each user, prevents someone from pretending to be someone else interfering with messaging. In such a case, the attacker's certificate or public key can not pass without verification and the conversation does not start. So with public key authentication, we have avoided a novu man-in-the-middle attack.

3.3) Truncation Attack (# BONUS #)

‘Truncation attack’ is that an attacker interferes with the passage of fragments and falsifies the receiver as if it were the last fragment of the message, and the receiver received the wrong message. Although the replay attack is recommended, the attacker can terminate the message.

In our project, we put a field named "type" in each fragment to prevent this. We also embed this field into the MAC. If this field is 1, it means that the message is the last message. But 0 means that there is no last message. Even if the Attacker changes this, MAC realizes that the receiver is a change and behaves accordingly.

4. GROUP CHATTING – SECURITY IMPLEMENTATION

Up to now, we have agreed that two people chat each other at each step above, and we have explained our security implementation in detail. However, we have made a project that also includes group chat in our CSE4074 course. So, we have provided all the security measures mentioned above for all participants in the group chat.

Let's talk briefly about how the group chat system works first. We used the "loby" logic to design such a chatting system. You absolutely have to play online games online (like Facebook 101 okey). In this game system, a room ("loby") is established and users communicate with each other by joining this room. Thus, you can systematically manage an unlimited number of users according to their own wishes. Our system works in a similar way as the general structure. A "peer" is selected as the "loby leader" before the beginning of the conversation, and the other "peers" connect to this "peer" and communicate with each other through it. In other words, the main person providing communication is 'lobby leader'. When you send a peer message, this message first goes to the 'lobby leader'. Lobby leader then sends this message to the other participant. So a 'peer' is unaware of the existence of other 'peers'. Moving from here, we can say that all communication is between the 'lobby leader' and the other 'peers'. At this point, we can use security here, which provides the conversation between the two people we talked about above. However, the 'lobby leader' should create different keys ('hMacKey', 'aesKey', 'ivKey') for each peer they connect to and keep them listed. When chatting with peers while chatting, you should use these keys which are specific to each peer. Thus, by using different keys for each peer-to-loop connection, the security of your peer-to-peer connection is even higher.