Protecting sensitive information

Team members

Ramírez Fuentes Edgar Alejandro

Rodríguez Melgoza Ivette

Salmerón Contreras María José

Table of contents

Problem to solve

The CEO of a certain company is promoting less use of paper. He wishes that sensitive documents be digital. Thus, these documents usually are signed by the board of directors (a group of people that take decisions in the company). Also, only the board of directors can see the content of these sensitive documents. They do not want to share a unique key, i.e. every member of the board must have her/his own key or keys. Imagine that your team is hired to develop a solution for this company.

Cryptographic services

- ### Privacy

  This service will be useful to hide the information that each sensitive document contains. As mentioned in the problem to solve, only the board of directors will be able to see the content, and privacy is the service that will help us to hide the information from any other user that does not belong to the board of directors.

- ### Integrity

  As mentioned before the documents contain sensitive data that must not be altered in any way. That is why our system must provide integrity and keep the data without any alteration by a third party.

- ### Non-repudiation

  The problem to solve mentions that the documents are usually signed by the board of directors, which means that the system should implement a way to help the users to sign a certain document and prevent them from denying previous commitments or actions.

- ### Authentication

  To guarantee that each entity in a communication is who it claims to be, we propose a login with a secret password which is going to be hashed with SHA3, implementing this the communication is authenticated.

Cryptographic primitives

- ## RSA-PSS

RSA Probabilistic Signature Scheme (PSS) will be used as cryptographic primitive to the cryptographic service of Non-repudiation and Integrity, specifically in the part of the sign used for the documents. PSS was specifically developed to allow modern methods of security analysis to prove that its security directly relates to that of the RSA problem. [1]

The size of the key used is of 2048 bites.

- ## SHA-3

SHA-3 will be used in the login of our system to hash the password of our users, and in the process of signing a document. It will help us to provide privacy to our users.

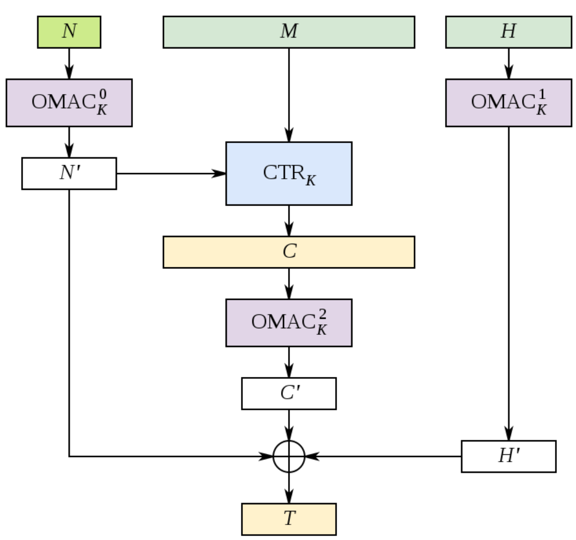
The size of the key used for the project is of 256 bits.

- ## AES128-EAX

Privacy is covered using AES, which is the cryptographic algorithm responsible for encrypting and decrypting the sensitive documents.

The mode of operation used is EAX with a size key of 16 bytes, the block size is of 128 bits and works with 10 rounds.

EAX diagram: [2]

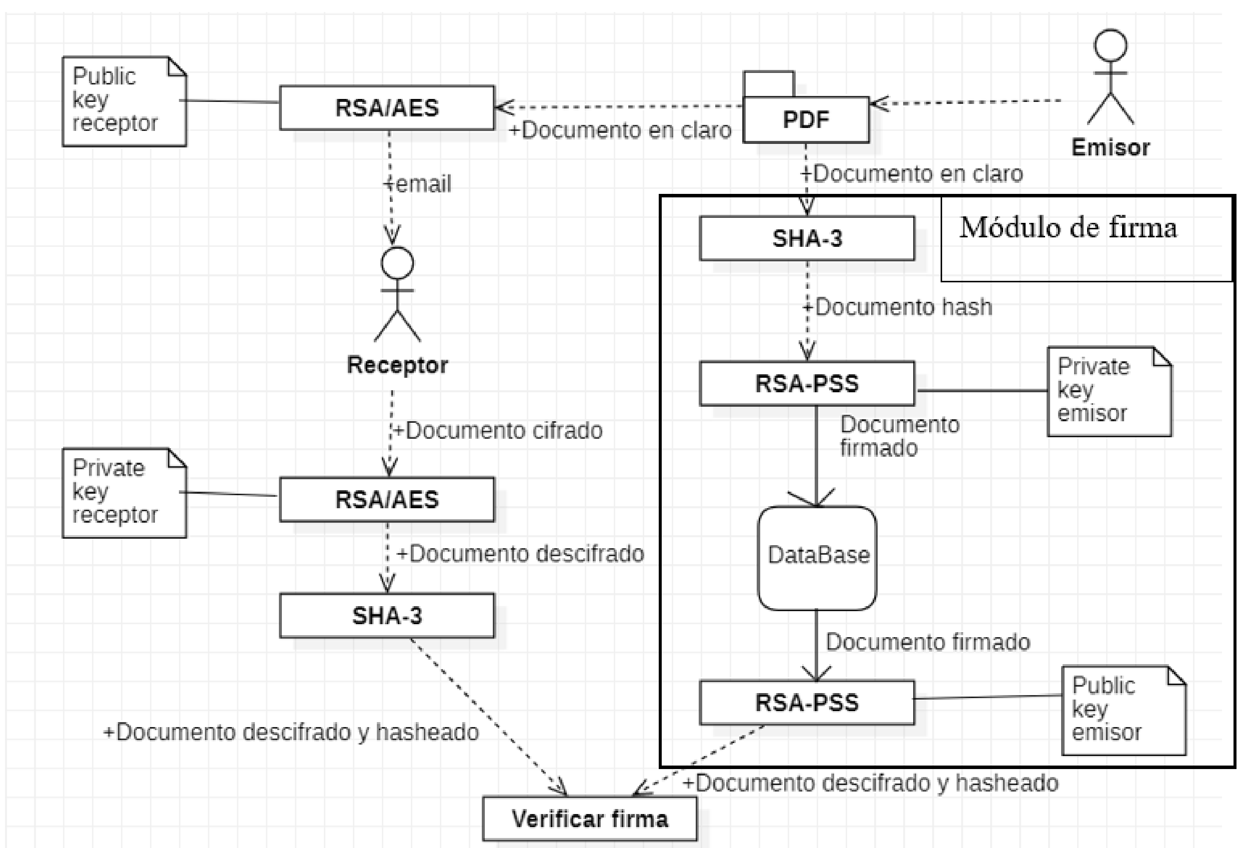


System architecture

The next figure shows the architecture of our system, it is easy to find the cryptographic primitives that we use and where in the program are used.

All starts with the sender and the pdf document, it is necessary to sign the document before is saved in the database, for that we use SHA-3 to hash the document and RSA-PSS (Probabilistic digital signature) to sign it, then the sign is saved in the database. In another thread, the pdf document is ciphered using a hybrid scheme (RSA/AES) and it is sent to the receiver.

The receiver deciphers the document using the same hybrid scheme and then using SHA-3 to compare this hashed message with the one saved in the database, if both are equal then the message is authentic if not, the message is not authentic.



Introduction to the system

To become true this system we decide to use Python as the main language to program the logic structure and the cipher and decipher modules, the cryptographic library that Python provides and the one that we use is pyCryptodome, this library contains AES, RSA and SHA functions which makes easier the programming part.

To save the sensitive information that it is necessary in the cipher and decipher process like the private and public keys, we use a database in mySQL.

And of course, to make a friendly environment for the user we made a html page and using Flask we connect the back and the frontpage.

To make this project we use three types of computers, the specification of each computer is listed in the follow table:

Computer model	Processor	Processor speed	Memory Storage
DELL Inspiron 15	Intel Core i7 7th Generation (x64)	2.90 GHz	16.0 GB
HP Pavillion 15	Intel Core i5 7th Generation (x64)	2.71 GHz	12.0GB
HP Pavillion Power 15	Intel Core i5 7th Generation (x64)	2.50 GHz	8.9 GB

Requirements

The system is a web app built using Flask (A Python framework), that is why you must have installed Python 3 v. 3.9 or newer.

The project depends on some Python packages that need to be installed using pip (The python packages administrator).

The packages that are essential are listed in a document called requirements.txt, and the name of the packages are:

- `pycryptodome`
- `Flask`
- `Flask-Session`
- `requests`
- `psycopg2-binary`
- `mysql-connector-python`

However, it is not necessary to install any of these requirements in your PC because all the requirements are installed in the hosting service.

Link: [Project Cryptography page]

Some code

In this section we are going to show some parts of the code where the magic happens.

Login module

This module is one of the most important in the system because it is useful to get the information about the person that will encrypt or decrypt a PDF document.

```python
@app.route("/login", methods=["GET","POST"])
@logout_required
def login():
    if request.method == "GET":
        return render_template("login.html")
    elif request.method == "POST":
        username = request.form.get('user')
        password = request.form.get("psw")

        if not username:
            flash("Username is required", "danger")
            return redirect(url_for('index'))

        if not password:
            flash('Password required', 'danger')
            return redirect(url_for('index'))

        # Hashing password
        password = bytes(password, 'utf-8')
        password = SHA3_256.new(password).hexdigest()

        connection = dataBaseConnection()
        if not connection:
            flash('There is a problem. Try later')
            return redirect(url_for('index'))


        userData = getCredentials(connection, username)

        if not userData:
            flash('The username is not registered', 'danger')
            connection.close()
            return redirect(url_for('index'))

        # Getting the user id and password
        idUser = userData[0]
        passwordUser = userData[1]
        usernameLogged = userData[2]

        if passwordUser == password:
            session["idUser"] = idUser
            session['username'] = usernameLogged
            flash(f'Welcome {username}!', 'info')
        else:
            flash('Wrong password', 'danger')

        connection.close()
        return redirect(url_for('index'))
```

Initially the user fills a form with their username and password and this module gets the values introduced. Once we have the username and password that were introduced, we create a connection to our database to validate that the credentials introduced by the user are correct. It is important to mention that to provide privacy to our users, their password is hashed using the cryptographic hash function SHA-3 of 256 bits. So, once we checked that the username exists in our database, it's time to check that the introduced password is like the one in the database, but as mentioned before the password in the database in hashed, thus we need to hash the received password using SHA-3 in order to compare whether the introduced password is correct or not. If the credentials are correct a new session (It contains the username and id) is created, otherwise a message of error is displayed.

The next piece of code is the responsible to get the credentials from the database to check if the data that the user submitted in the form is correct.

```python
def getCredentials(connection, username : str) -> tuple:
    '''
        Get the credentials of the username to login

        Parameters
        -----------
        connection
            It is the connection to the database

        username : str
            It is the username that will be looked for their credentials

        Return
        ---------
        credentials : tuple
            It is a tuple that contains the id and password of the username
    '''
    cursor = connection.cursor()
    cursor.execute("SELECT idUsuario, contrasena, nombreUsuario FROM usuario WHERE nombreUsuario = %s;", (username,))
    credentials = cursor.fetchone()
    return credentials
```

Encrypted module

```python
  ##############################
 #     Encrypt file route    #
##############################

@login_required
@app.route("/encrypt-file", methods=["POST",])
def encrypt_file():

    # Getting the id of the user that is logged in
    senderId = session.get('idUser')

    # Getting the receiver ID
    receiverId = request.form.get("receiverId")

    if not receiverId:
        flash(f'No receiver selected', 'danger')
        return redirect(url_for('index'))

    if int(receiverId) < 0:
        flash(f'Invalid receiver ID', 'danger')
        return redirect(url_for('index'))

    # Check if the post request has the file part
    if 'file' not in request.files:
        flash(f'No file part', 'danger')
        return redirect(url_for('index'))

    # Getting the file from the form
    file = request.files['file']

    # If user does not select file, browser also
    # submit an empty part without filename
    if file.filename == '':
        flash(f'No selected file', 'danger')
        return redirect(url_for('index'))

```

Initially in the development of this module it is required to capture the data provided by the frontend of the project when a user logs in and decides to encrypt a document.

The required information is:

- Sender private key
- Receiver public key
- Receiver ID
- Sender ID
- Receiver email
- PDF document

Each time any of the data is obtained, it is verified that its content is different from null, if so, continue with the program, otherwise a message is displayed on the page indicating the missing information.

Subsequently, it is verified that the file to be encrypted has been loaded correctly and that it is of type pdf, otherwise an alert message is displayed on the page.

If the previous filters are passed, the file is saved in a temporary folder, the path is stored for future use and the sender private key is obtained.

```
if file:
        try:
            filename = secure_filename(file.filename)
            # Check if the filename fulfills the standard
            # Filename standard: senderID_receiverID_encryptedFilename_encryptionID.bin
            filenameRegex = r"^[\w\-]+\.pdf$"

            if not re.match(filenameRegex, filename):
                flash('The file is not a PDF file or the filename is not valid', 'danger')
                return redirect(url_for('index'))

            filenameWithoutExtension = filename.split('.')[0]

            # It is the path where the uploaded file will be stored
            path = os.path.join(TMP_FOLDER, filename)

            # Store the file in the provided path
            file.save(path)

            # Getting the emisor private key
            with open(f"{PRIVATE_KEY_FOLDER}{senderId}.pem", "rb") as privateKeyFile:
                    emisorPrivateKey = privateKeyFile.read()
```

After the connection to the database is made (which is a process that will be explained later), it is verified that the connection is successful and the number of documents that the user has sent is obtained.

```
connection = dataBaseConnection()

            if not connection.is_connected():
                flash('There is a problem. Try later')
                return redirect(url_for('index'))

            # Getting the quantity of encrypted documents
            encryptedDocuments = getEncryptedDocumentsQuantity(connection, senderId)[0]
```

Within the web page the option is given to send the file only to one person or to all those who are registered in the database. The process is essentially the same for both, but the option "all" includes an iterative development.

In the code shown in the following image, the email of the person who is going to receive the message is obtained from the database, the number of documents sent by the sender is increased, the message is encrypted using the "encryptionProcess" function, send the encrypted file by mail and finally the files in the temporary folder are deleted.

```python
encryptedDocuments += 1
            receiverData = getReceiverData(connection, receiverId, senderId)

            if not receiverData:
                flash('Not valid receiver')
                connection.close()
                return redirect(url_for('index'))

            receiverEmail = receiverData[0]
            encryptedFilename = f"{senderId}_{receiverId}_{filenameWithoutExtension}_{encryptedDocuments}.bin"
            encryptionProcess(connection,receiverId,senderId,encryptedDocuments,path,emisorPrivateKey,receiverEmail,encryptedFilename)
            # Open a thread to delete the uploaded file
            uploadedThread = Thread(target=deleteFile, args=(f"{TMP_FOLDER}{filename}",))
            uploadedThread.daemon = True
            uploadedThread.start()
            return redirect(url_for("index"))
```

The "encryption Process" function receives as parameters the information of the receiver, the sender and the file to be encrypted. In this section the document is signed and encrypted, if everything comes out correctly the .bin file is created with the cipherText.

The "signDocument" function receives the RSA private key of the sender, the document and the path where the signature will be stored.

The signing process is carried out using RSA and the issuer's private key, the document is hashed using SHA-3 and finally the result is stored.

```python
def signDocument(document : bytes, senderPrivateKey : bytes, documentPath : str) -> bool:
    try:
        # Getting the sender private key
        key = RSA.import_key(senderPrivateKey)
        # Hashing the document
        h = SHA3_256.new(document)
        # Signing the hashed document
        signature = pss.new(key).sign(h)

        # Storing the signature
        with open(documentPath, "wb") as signFile:
            signFile.write(signature)
        return True
    except:
        return False

```

The "encryptDocument" function receives as inputs the document, the public key of the receiver and the name of the resulting file.

In this section a hybrid system of RSA and AES is used, initially a session key with random numbers is generated, said key is encrypted with the public key and RSA and it is the one that is used as the key for the encryption process with AES. At the end of the encryption process the encrypted file it is stored.

```python
def encryptDocument(document : bytes, receiverPublicKey : bytes, encryptedDocumentFilename : str) -> bool:
    try:
        # Getting the receiver public key
        key = RSA.importKey(receiverPublicKey)

        # Generating the AES session key
        sessionKey = get_random_bytes(16)

        # Encrypting the session key using the receiver public key
        cipherRSA = PKCS1_OAEP.new(key)
        encSessionKey = cipherRSA.encrypt(sessionKey)

        # Encrypting the document using the AES session key
        cipherAES = AES.new(sessionKey, AES.MODE_EAX)
        ciphertext, tag = cipherAES.encrypt_and_digest(document)

        # Storing the encrypted data
        with open(f"{TMP_FOLDER}{encryptedDocumentFilename}", "wb") as encrypted_file:
            [ encrypted_file.write(x) for x in (encSessionKey, cipherAES.nonce, tag, ciphertext) ]
        return True
    except:
        return False
```

Decryption module

This module is responsible to decrypt an encrypted file using the data submitted by the user in the form.

The required information to decrypt a message is:

- Receiver ID
- Sender ID
- Encrypted file
- Receiver private key
- Sender public key
- Signature

Each time the user submits an encrypted file, this method check that all the required information is available if not It displays an error message.

In the next image is shown the piece of code described above:

```python
@app.route("/decrypt-file", methods=["POST",])
@login_required
def decrypt_file():
    # Get the sender ID
    senderId = request.form.get('senderId')
    receiverId = session.get('idUser')
    # Check if the senderId was sent
    if not senderId:
        flash(f'No selected sender', 'danger')
        return redirect(url_for('index'))

    # Check if the post request has the file part
    if 'file' not in request.files:
        flash(f'No file part', 'danger')
        return redirect(url_for('index'))

    # Getting the file from the form
    file = request.files['file']

    # If user does not select file, browser also
    # submit an empty part without filename
    if file.filename == '':
        flash(f'No selected file', 'danger')
        return redirect(url_for('index'))

    if file:
        try:
            filename = secure_filename(file.filename)

            # Check if the filename fulfills the standard
            # Filename standard: senderID_receiverID_encryptedFilename_encryptionID.bin
            filenameRegex = r"^[0-9]+_[0-9]+_[\w\-]+_[0-9]\.bin$"

            if not re.match(filenameRegex, filename):
                flash('The filename was modified or it is not a bin file', 'danger')
                return redirect(url_for('index'))

            filenameWithoutExtension = filename.split('.')[0]

            documentPath = os.path.join(TMP_FOLDER, filename)

            # Store the file in the provided path
            file.save(documentPath)


            # Getting the sender public key
            with open(f"{PUBLIC_KEY_FOLDER}{senderId}.pem", "rb") as senderPublicKeyFile:
                senderPublicKey = senderPublicKeyFile.read()

            # Getting the user private key
            # Using a temporary  private key while the login module is finished
            with open(f"{PRIVATE_KEY_FOLDER}{receiverId}.pem", "rb") as receiverPrivateKeyFile:
                receiverPrivateKey = receiverPrivateKeyFile.read()

            # Getting the signature
            # Using a temporary signature while the database is connected
            with open(f"{SIGNATURES_FOLDER}{filename}", "rb") as signatureFile:
                signature = signatureFile.read()
```

Once the module validates that the required information to decrypt an encrypted file is fulfilled, it starts the decryption process, which is executed by the “decryptDocument” function, delete all the temporal files created to decrypt the message, and download the decrypted file to the user's PC. If the decryption process was not successfully executed, it displays an error message.

```python
    # Getting the reference to the file to decrypt
    encryptedDocument = open(documentPath, "rb")
    # Check if the file was decrypted successfully
    decrypted = decryptDocument(encryptedDocument, receiverPrivateKey, senderPublicKey, signature, filenameWithoutExtension)
    encryptedDocument.close()

    # Open a thread to delete the uploaded file
    uploadedFileThread = Thread(target=deleteFile, args=(f"{TMP_FOLDER}{filename}",))
    uploadedFileThread.daemon = True
    uploadedFileThread.start()

    if decrypted:
        # Open a thread to delete the decrypted file
        pdfThread = Thread(target=deleteFile, args=(f"{TMP_FOLDER}{filenameWithoutExtension}.pdf",))
        pdfThread.daemon = True
        pdfThread.start()

        return redirect(url_for('download_file', name=f"{filenameWithoutExtension}.pdf"))
    else:
        # Return an error message
        flash(f'The signature is not authentic. Try later', 'danger')
        return redirect(url_for('index'))
except:
    flash('Something went wrong', 'danger')
    return redirect(url_for('index'))
```

In the next image is shown the function responsible for the decryption process, which validate the signature of the encrypted file and decrypt the file. If the signature is correct, the document is decrypted, otherwise it displays an error message.

```python
def decryptDocument(encryptedDocument : bytes, receiverPrivateKey : bytes, senderPublicKey : bytes, originalSignature : bytes, filename : str) -> bool:
    '''
        Decrypt a document using a hybrid encryption scheme.
        It uses RSA with PKCS#1 OAEP for asymmetric encryption of an AES session key.

        Parameters
        -----------

        encryptedDocument : bytes
            It is a reference (pointer) to the document that contains the encrypted document

        receiverPrivateKey : bytes
            It is the receiver private key that will be used to decrypt the AES session key

        senderPublicKey : bytes
            It is the sender public key that will be used to verify the signature

        originalSignature : bytes
            It is the original signature of the file that will be decrypted

        filename : str
            It is the filename of the encrypted file

        Return
        -------
        True if the file was decrypted successfully. Otherwise, it returns False
    '''
    try:
        # Getting the receiver private key
        privateKey = RSA.import_key(receiverPrivateKey)

        # Getting the necessary information to decrypt the document
        encSessionKey, nonce, tag, ciphertext = \
        [ encryptedDocument.read(x) for x in (privateKey.size_in_bytes(), 16, 16, -1) ]

        # Decrypting the session key using the receiver private key
        cipherRSA = PKCS1_OAEP.new(privateKey)
        sessionKey = cipherRSA.decrypt(encSessionKey)

        # Decrypting the document using the AES session key
        cipherAES = AES.new(sessionKey, AES.MODE_EAX, nonce)
        decryptedDocument = cipherAES.decrypt_and_verify(ciphertext, tag)

        # Verifying the signature
        is_valid_signature = verifySignature(decryptedDocument, senderPublicKey, originalSignature)

        if is_valid_signature:
            # Writing the decrypted document in a new PDF file
            with open(f"{TMP_FOLDER}{filename}.pdf", "wb") as decryptedDocument_file:
                decryptedDocument_file.write(decryptedDocument)
            return True
        else:
            return False
    except:
        return False
```

The next function is the responsible to verify that the signature is correct in the decryption process:

```python
def verifySignature(decryptedDocument : bytes, senderPublicKey : bytes, signature : bytes) -> bool:
    '''
        Veryfy if the original signature is the same than the obtained signature from the decrypted document

        Parameters
        ----------
        decryptedDocument : bytes
            It is the decrypted document that will be used to generate a signature

        senderPublicKey : bytes
            It is the sender public key that will be used to verify if the obtained signature is the same than the original signature

        signature : bytes
            It is the original signature

        Return
        ---------
            True if the signatures match, otherwise False
    '''
    try:
        # Getting the sender public key
        key = RSA.import_key(senderPublicKey)

        # Hashing the decrypted document
        # h = SHA256.new(decryptedDocument)
        h = SHA3_256.new(decryptedDocument)

        verifier = pss.new(key)
        try:
            # Compare the hashed decrypted document signature with the original signature
            verifier.verify(h, signature)
            return True
        except (ValueError, TypeError):
            # If the signatures are not the same, it returns False
            return False
    except:
        # Something went wrong
        return False
```

Connection to the data base

In the "dataBaseConnection" function all the necessary information to connect to the database is entered and a connection is returned as a result.

```python
def dataBaseConnection():
    '''
        Get the connection to the database

        Return
        ---------
        connection
            It is the connection to the database
    '''
    connection = psycopg2.connect(
            user = os.environ.get('dbUser'),
            password = os.environ.get('dbPass'),
            host=os.environ.get('dbHost'),
            database=os.environ.get('dbName'),
            port=os.environ.get('dbPort')
            )
    return connection
```

To obtain and update the data, the following functions were used:

- getUserList()
- getCredentials()
- getReceiverData()
- getEncryptedDocumentsQuantity()
- updateEncryptedDocumentsQuantity()

Which receive the connection and the id of the user to be accessed in the database and return a tuple with the results.

References

[1] Wikipedia, "Probabilistic signature scheme," [Online]. Available: https://en.wikipedia.org/wiki/Probabilistic_signature_scheme.

[2] M. Bellare, P. Rogaway and D. Wagner, EAX: un modo de cifrado autenticado convencional, IACR, 2003.