

Instituto Politécnico Nacional
Escuela Superior de Cómputo



Cryptography

Project:

“Protecting sensitive information”

Team: Golden Chicken

Students:

Hernández Rodríguez Armando Giovanni

Morales Hernández Carlos Jesús

Ramírez Hidalgo Marco Antonio

Teacher:

Díaz Santiago Sandra

Group: 3CM15

Deadline: December 16, 2021

Protecting sensitive information

Problem Statement

The CEO of certain company is promoting less use of paper. He wishes that sensitive documents be digital. Thus, this documents usually are signed by the board of directors (group of persons 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 primitives and services

Cryptographic services	Cryptographic primitives	Description
Data confidentiality	AES 256 bits mode CBC	<p><i>Confidentiality</i> is the property whereby sensitive information is not disclosed to unauthorized entities.</p> <p>Encryption is used to provide confidentiality for data. The unprotected form of the data is called plaintext. Encryption transforms the plaintext data into ciphertext, and ciphertext can be transformed back into plaintext using decryption. Data encryption and decryption are generally provided using symmetric-key block cipher algorithms. AES is approved for data encryption using all three key sizes.</p>
Data Integrity	<p>Digital signature: RSA 2048 bits</p> <p>Hash function: SHA- 256</p>	<p><i>Data integrity</i> (often referred to as simply integrity) is concerned with whether data has changed between two specified times.</p> <p>Digital signatures are used to provide data integrity.</p> <p>RSA is approved for computing digital signatures as long as the key size is bigger or equal than 2048 bits.</p>

		<p>A cryptographic hash function is a function that maps a bit string of arbitrary length to a fixed length bit string.</p> <p>With these services we guarantee the identity authentication of the information because SHA-256 generates a hash value that can provide some assurance of the integrity of the data over which a hash value is generated, but it is used along with RSA to assure data integrity.</p> <p>The use of SHA-256 is acceptable for all hash function applications.</p>
Identity Authentication	Digital signature: RSA 2048 bits	<p><i>Identity Authentication</i> (often referred to as simply authentication) is used to provide assurance of the identity of an entity interacting with a system. The authentication process usually requires that the entity produce some proof of its identity (e.g., using a token, fingerprint, PIN, or some combination thereof) before access to some data or resource can be granted.</p> <p>Digital signatures are used to provide identity authentication.</p>
Source Authentication	Digital signature: RSA 2048 bits	<p><i>Source authentication</i> is used to provide assurance of the source of information to a receiving entity. A special case of source authentication is called non-repudiation.</p> <p>Providing nonrepudiation and message integrity can be realized with digital signature algorithms.</p>

Architecture

To design the architecture of our project, a block diagram was elaborated. This block diagram describes each of the components and their interaction with the cryptographic primitives used.

The block diagram begins in the triangle figure, the first component is the main menu where the user will be shown (as the case may be) the option to log in or create a directive. In the case of the CEO, his pair of keys (private and public) and password must have been created for communication between him and the directives, therefore, a user of the CEO type doesn't need to be created.

On the other hand, a user must be created for the directive, to make the respective login. It's important to mention that when a directive is created, the keys are generated and exchanged with the CEO, that is, the RSA key pair, and the AES key are generated for the directive. For the exchange, the CEO's public key is shared with the directive, then the directive encrypts his AES key with the CEO's public RSA key and is sent along with the directive's public RSA key. When the creation of the user is finished, the login is returned for the directive.

For the login of a CEO or directive, a username, password, and a one-time password provided by the Google Authenticator application will be entered, performing a two-factor verification, providing security and protection when logging in. It's necessary that the CEO and the directive have this application installed.

When the CEO logs in, a menu is displayed showing three options: choose a document, delete a document, and verify signatures.

If the CEO decides to choose a document, one file must be selected by the CEO to be encrypted as many times as there are directors in the board of directors using each director's AES key. These AES keys are decrypted employing the CEO's RSA private key. Finally, the encrypted file and the IV vector used in the encryption are sent to every director to be signed by them.

In case the CEO chooses to delete a document, that document and its respective related files will be eliminated from the CEO's and each director's storage.

When the third option in the CEO's menu is chosen, another menu is displayed to the CEO asking if the verification of the signatures is to be done for all documents uploaded by the CEO or just a specific document.

The process of verifying the signatures of a single document consists of the validation of the signature checking if it matches the encrypted document hash.

Then, a report is shown to the CEO about which signatures have been done, which are missing, and which signatures do not match the hash.

Finally for the login of the directors, it includes 3 services that are “Sign a document”, “Documents to sign” and “Signed documents”. If one of the directives selected “Sign a document”, the directive decrypts the document to be signed with their AES key and the corresponding IV, then the directive can see the document to be signed.

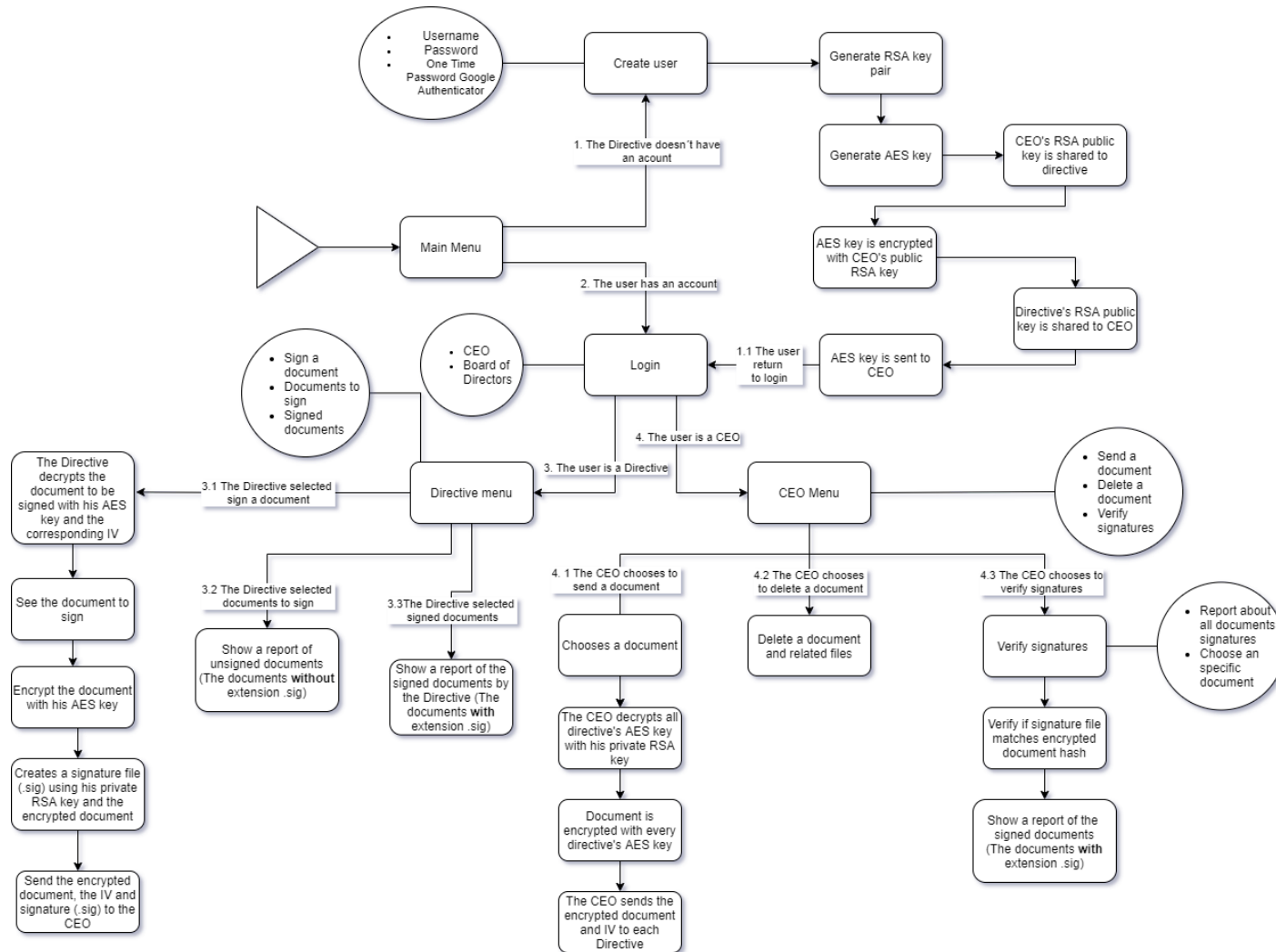
The document is encrypted with their AES key, then the system creates a signature file with extension “.sig” using their private RSA key and the encrypted document. In the end the encrypted document, the IV and signature (.sig) is sent to the CEO.

If the directive selects the option “Document to sing”, the system will show the directive a report of unsigned documents, these documents do not have the extension “.sig”, this way the directive can identify whether the documents are already signed or not.

If the directive selects the option “Signed documents”, the system will show a screen to the directive about all the documents that have the extension “.sig”, with this extension the directive can identify the documents that are already signed.

The block diagram of the architecture can be seen in the following image.

Block diagram



Characteristics of our computer

Hernández Rodríguez Armando Giovanni

- OS: Windows 10 Pro-64 bits
- System model: Aspire A515-51
- Processor: Intel® Core (TM) i5-8250U CPU @ 1.6GHz (8 CPUs) ~ 1.8GHz
- Computer memory: 20480 MB RAM

Morales Hernández Carlos Jesús

- OS: Windows 10 Home-64 bits
- System model: Asus X555QA
- Processor: ADM A12-9700P RADEN R7, 10 COMPUTE CORES 4C+6G 2.50 Ghz
- Computer memory: 16 GB RAM

Ramirez Hidalgo Marco Antonio

- OS: Windows 10 Pro-64 bits
- System model: ASUS Prime B250M-A
- Processor: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz 3.00 GHz
- Computer memory: 16.0 GB RAM

List of technologies we use

Programming language:

- Python

Crypto library name:

- **Pycryptodome:** is a self-contained Python package of low-level cryptographic primitives. It supports Python 2.7, Python 3.5 and newer, and PyPy. You install it with:

```
pip install pycryptodome
```

- **Python-RSA:** is a pure-Python RSA implementation. It supports encryption and decryption, signing and verifying signatures, and key generation according to PKCS#1 version 1.5. Installation can be done in various ways. The simplest form uses pip:

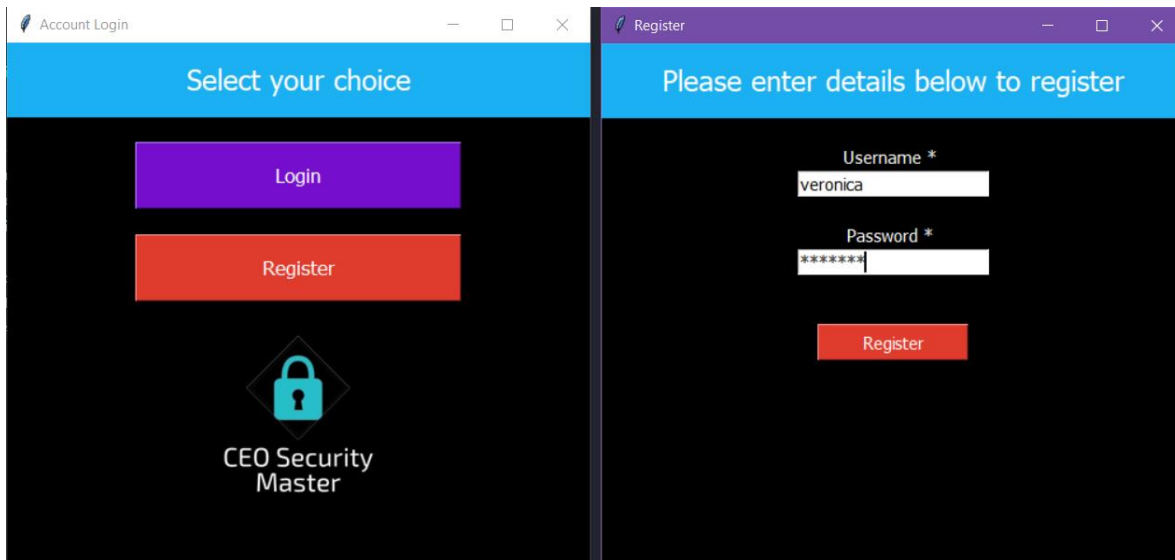
```
pip install rsa
```

- **Database Manager:** A database manager wasn't used, however, for the user registry files were used, where the username, password and OTP are stored. Thus, the login that the CEO or directive performs can be validated.

Operation of cryptographic components

Key generation

It's important to mention that for the generation of symmetric and asymmetric keys, the respective directive must be registered. In the registration, the directive must enter the username and password.



After registration, the generation of keys begins.

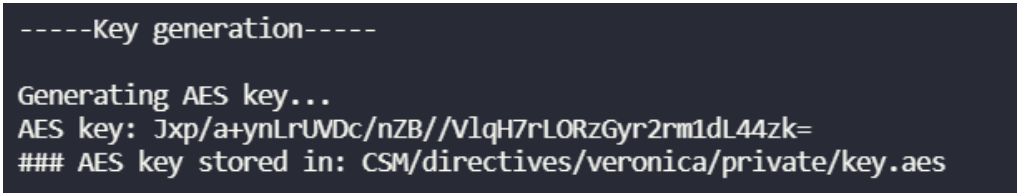
- **Symmetric keys**

To generate the AES keys of the different directives, the following function was used, which returns 32 bytes (256 bits) randomly.



```
#Generates random 256-bit AES key
def generate256Key():
    return get_random_bytes(32)
```

The following is a screenshot of the AES key generation for a directive named Veronica. The directive is shown the AES key in base 64 and the path where it was stored.

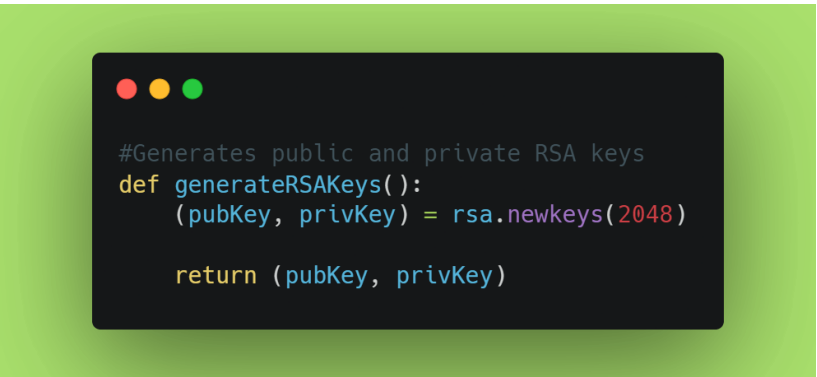


```
-----Key generation-----
Generating AES key...
AES key: Jxp/a+ynLrUVDc/nZB//VlqH7rLORzGyr2rm1dL44zk=
### AES key stored in: CSM/directives/veronica/private/key.aes
```

- **Asymmetric keys**

To generate the private and public RSA keys, the function *generateRSAKeys()* is used, which returns the 2048-bit public and private RSA keys.

Generating a key pair may take a long time, depending on the number of bits required. The number of bits determines the cryptographic strength of the key, as well as the size of the message you can encrypt.



```
#Generates public and private RSA keys
def generateRSAKeys():
    (pubKey, privKey) = rsa.newkeys(2048)

    return (pubKey, privKey)
```

The process of generating *veronica's* RSA public key is shown below. The public key is stored in the directive's folder (*veronica*) with the filename "*pub.pem*". In addition, the corresponding key is shown to the user

```
Generating RSA key pair...
Saving rsa key pair...
### Public key stored in: CSM/directives/veronica/pub.pem
Opening...Please CLOSE the file and press enter to continue...

-----BEGIN RSA PUBLIC KEY-----
MIIBBgKCAQEAA5B0s7PH1y+r1YTKuVpt05z/AP5zG0HHyVzLz3PrJmb3THrhQtQAE
hujh239F5o71GxTAF7FfbPcBBcNG+rUk+L23UU02xhF4bx0aX7uU1zq1rAOKIUbk
g8xgSwM/h9P6iT2q7BZwuKq3M1rpcEwzfcX09/en5DwevEvGQW3SMsKmpS2EN+x9
0061t9jk/+OKGIuClgHgV3XdM5TSWQQI8X0E8VcDjT9fIEDYrLoRUsLTQKOBRI4c
baBG7Sa4KeBRZpAhGwt82DXa2k0haAOQ6LUJa+9yW3+1+iomgRdCCZnOhZUWjzk
huwv9+63CrmQ5hkQ07dnrBJWC/EYBcDyvQIDAQAB
-----END RSA PUBLIC KEY-----
```

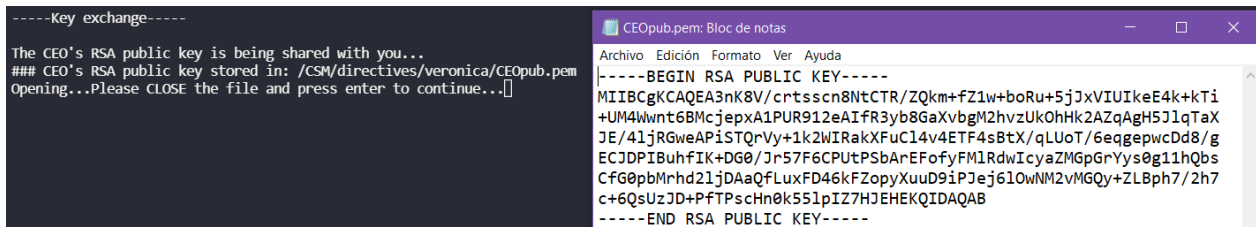
The same happens with the generation of the RSA private key, it's stored in the corresponding directory of *veronica* in the "private" folder with the filename "*priv.pem*". After viewing the RSA key file, the user is prompted to close the files.

```
### Private key stored in: CSM/directives/veronica/private/priv.pem
Opening...Please CLOSE the file and press enter to continue...

-----BEGIN RSA PRIVATE KEY-----
MIIEgQIBAAKCAQEAA5B0s7PH1y+r1YTKuVpt05z/AP5zG0HHyVzLz3PrJmb3THrhQ
tQAehuJh239F5o71GxTAF7FfbPcBBcNG+rUk+L23UU02xhF4bx0aX7uU1zq1rAOK
IUbkG8xgSwM/h9P6iT2q7BZwuKq3M1rpcEwzfcX09/en5DwevEvGQW3SMsKmpS2E
N+x90061t9jk/+OKGIuClgHgV3XdM5TSWQQI8X0E8VcDjT9fIEDYrLoRUsLTQKOB
RI4cbaBG7Sa4KeBRZpAhGwt82DXa2k0haAOQ6LUJa+9yW3+1+iomgRdCCZnOhZU
Wjzkhuwv9+63CrmQ5hkQ07dnrBJWC/EYBcDyvQIDAQABaoIBAQCyo4nUowuVcDy
E50Tm1c8gJwdrchgeEtwIvkVotXEv+ogCOWSsQy1CmiGCFAG9g4aqseT83zaEo
amJfapyznaIlvKNPpkUKA3Evbw/l0XkSiZf0curC2/uUjfkA8Rb7iXpv772nL7nc
pLgv/777KVJcM4dZD35U1A0I82H1NXp0BS0912X+Sx4FKuoThLRIf3BTvGd4N06E
cMQGtgqvM95mR6VHXD13i15Qs6zhUEdH26QHs8cc7WNNFXPYoMITEHzh18b5wydy
SkrJGG3DhDr/SO69wwQL9aVdNaUGDvXtq/CNXyroonC4rjuSzMZFSWwu5YCT8iMi
CCXxKYudAoGJAPc+zSv7k1pw7+8+Sp+faRtFDV/vFo3tS95miZp38eUy71ZC5/1U
zeeuXjN3RaXZrdFbpI0vNIqNe1JcZ8zGKTh28bzBnB/LS4nhjreDdnihKeBCKv97
uGPiYMHnWQ11wfAwmv3nz9QOT1/9g3NsH1NpUNzd1tKXTbmV3uN/FjrVpNm20TaB
fksCeQDsMPRdQrRN2vjJmbztAemF9IWFmPqGGtDnsJh1LiP0MBFxoadaHHLoEDv
KFzcWqSwS/Zy78LQNNjac0ALIw5L20bbvywVZlgSB+bbFHYxfJhtjMQ985B4aBAD
k0jZB0fL4ZrrD8yCc9saRPnT1isYk7hHHfWTjhCgYhURrji7B7/JTE16KLBJDJM
1E4JzFJw0FkTFvqCh1EG4kTIR4eqR3TkKwB7JIW+EaP4//boFRYgwDbYyhyrpuQ5
tfmScKBKCh7WgeG0K3aUn0Wrx+/KoOgUtFJRmqx9/ym7ia1jHhHub6asb9df7
s6NX1cSqrD69isNt4kzugu+FkA8ScDNfAnhLfAXM/HGFZFMycXwVskd350Aqnh+
fX14E1VRGGAHRwN8PcXmz7t2mXvHMz4iSnHRYiU0bH31b0NGhYjVR1neUa70Uhwz
rSa+dHoKUEzVWUOZ12AscHgekxtzGRIyEkqW1T5em4j/tCk1BxBU7GCP1T5TdD0
9fccYghspTk0wzJ15pG8/igNk7yAjUpC1D8f/11k7dT87HBKxwGpCt1Q8Dm3Jc2x
TkFrrei9yDxJ/V7KrRwKhJ2/M31xj1q2GamYC6UrVr8CXShN+DSZg8NgJj/syH1W
Z/jQXNHGB+1rDGF+bIB0ZIogBRk8WgfmFpFp2R1s8zg+nqe1chaDzQgPvZWZ
-----END RSA PRIVATE KEY-----
```

Key exchange

To establish communication between the CEO and the directive, it's necessary to exchange keys. For this, the CEO's RSA public key is shared with the directive (*Veronica*), a file called *CEOpub.pem* is displayed, this file is stored in her directive folder.



The screenshot shows two windows. The left window is a terminal with the following text:

```
-----Key exchange-----
The CEO's RSA public key is being shared with you...
### CEO's RSA public key stored in: /CSM/directives/veronica/CEOpub.pem
Opening...Please CLOSE the file and press enter to continue...[]
```

The right window is a Notepad application titled "CEOpub.pem: Bloc de notas". It contains the following RSA public key:

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEA3nK8V/crtsscn8NtCTR/ZQkm+fZ1w+boRu+5jJxVIUIkeE4k+kTi
+UM4Wwnt6BMcjepxA1PUR912eAIFR3yb8GaXvbgM2hVzUkOhHk2AZqAgH5J1qTaX
JE/41jRGweAPiSTQrVy+1k2WIRakXFuCl4v4ETF4sBtX/qLUoT/6eqgepwcDd8/g
ECJDPiBuhfIK+DG0/Jr57F6CPUtPSbArEFofyFM1RdwIcyazMGpGrYys0g11hQbs
CfG0pbMrhd21jDAaQfLuxFD46kFZopyXuuD9iPJej61OwNM2vMGQy+ZLBph7/2h7
c+6QsUzJD+PFTPscHn0k551pIZ7HJEHEKQIDAQAB
-----END RSA PUBLIC KEY-----
```

The CEO's public key (*CEOpub.pem*) allows the directive's AES key to be encrypted. After encryption is done, the encrypted AES key is shared with the CEO and stored in his private folder.

The directive's public key is also shared in the CEO's directives folder. The directive's public key will allow the CEO to verify the directive's signature.



The screenshot shows a terminal window with the following text:

```
The AES key is being encrypted with the CEO's public RSA key...
Reading rsa public key of the ceo...
Reading your AES key...
Encrypting...

AES key encrypted:
2QBsuc5/IwAch1KrJWeBC0YtZurnin+Bu2Hgl/hw/+YxPpmFM7iJnABLRdymp88
HJtt+KZ/dR9pcX1ufcX5FWzmgrYNta46lUcvGwXPNz2UunIyyP6IIE9NC9l3KH5
hJTADTdXm33UEBSNAnfgVat+Uhhx1ZRnr/B/Ztg6VtGib2A114xFvU5M1216c71E
1k1DkbpzWtpoKrlBLvO8RzLwCieNiwijsy030hBv+OdSq17VEcTp8juIU8nQNAoa
3zOX2Dw1csuFGFF0w1b2o4lfx9fjm3dltahLYk0iTcWlXKHknXV+zjw/XMZYox1D
A9t7hIMIcVVI8gDZ1wtuyw==

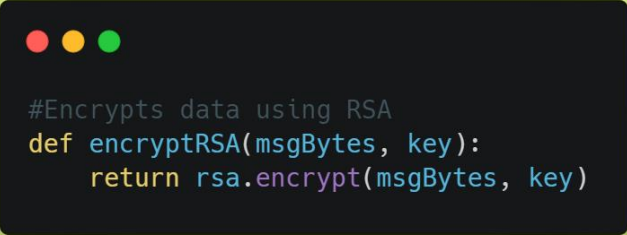
Your RSA public key and encrypted AES key are being shared with
the CEO..
### Encrypted AES key(directive) stored in: CSM/ceo/directives/v
eronica/private/encryptedkey.aes
### Public key(directive) stored in: CSM/ceo/directives/veronica
/pub.pem

-----Congratulations sharing and successful key generation!-----
```

Encryption

- **RSA encryption**


To share the directive's AES key with the CEO, it was encrypted with the CEO's public key. For this, the *encryptRSA()* function that receives a message in bytes and the asymmetric key was used. This function returns the bytes of the message (AES key) encrypted with RSA, the encrypted AES key is stored in the CEO's private folder in base 64.



```
#Encrypts data using RSA
def encryptRSA(msgBytes, key):
    return rsa.encrypt(msgBytes, key)
```

- **AES encryption**

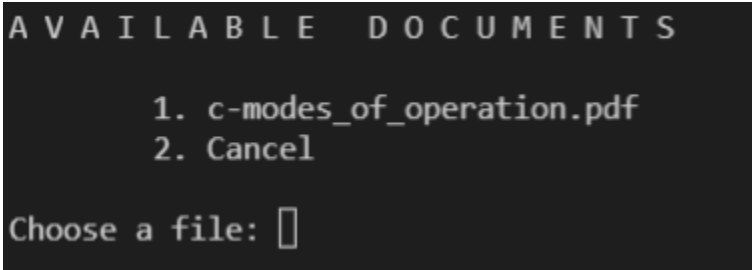
The process of AES encryption is done via the following function:



```
def encryptAES(key, msgBytes):
    cipher = AES.new(key, AES.MODE_CBC)
    return cipher.encrypt(pad(msgBytes, 16)), cipher.iv
```

As noted in the code, CBC mode is employed which also needs a padding of 16 bytes.

When the CEO wants to share a document, it is selected using this menu:



```
AVAILABLE DOCUMENTS

    1. c-modes_of_operation.pdf
    2. Cancel

Choose a file: █
```

Once the desired file is selected, the process of encryption starts. First, the CEO's private RSA key is retrieved to decrypt every directive's AES key which is encrypted with the CEO's public RSA key. Then, each directive's decrypted AES key is used to

encrypt the document, the encryption is done as many times as there are directives. Then, the encrypted file and IV vector used for encryption is sent to every directive at their respective directories.

This process can be seen in the image below.

```

----- Encryption -----
Getting CEO's RSA private key...
----- carlos -----

Decrypting AES key with CEO's private RSA key...
Encrypting file with AES key...
Encryption stored in: directives/carlos/documents/c-modes_of_operation.pdf/encFile.enc
IV stored in: directives/carlos/documents/c-modes_of_operation.pdf/iv.data

----- gabriela -----

Decrypting AES key with CEO's private RSA key...
Encrypting file with AES key...
Encryption stored in: directives/gabriela/documents/c-modes_of_operation.pdf/encFile.enc
IV stored in: directives/gabriela/documents/c-modes_of_operation.pdf/iv.data

----- giovanni -----

Decrypting AES key with CEO's private RSA key...
Encrypting file with AES key...
Encryption stored in: directives/giovanni/documents/c-modes_of_operation.pdf/encFile.enc
IV stored in: directives/giovanni/documents/c-modes_of_operation.pdf/iv.data

```

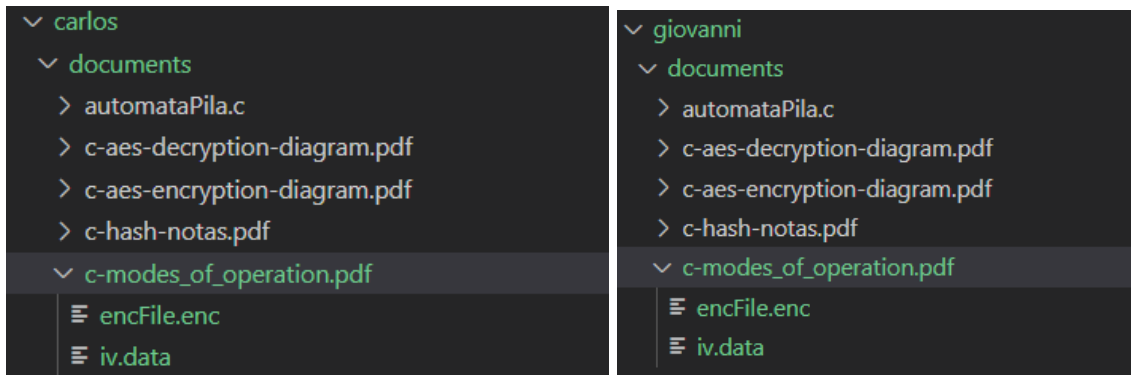
This code performs the AES encryption.

```

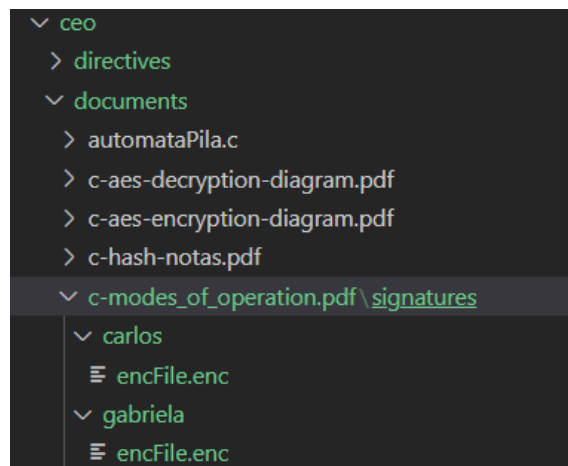
encContent, iv = aes.encryptAES(decAESKey, contentFile)
print("Encrypting file with AES key...")

```

For demonstration's sake, the files generated in the directories of the directives *carlos* and *giovanni* are exhibited below.



It is worth mentioning that the encrypted file is also saved for the CEO without the IV vector to use it later when the process of signature verification takes place. This is also done for every directive. Some of these files are shown below.



Decryption

- **RSA decryption**

For the RSA decryption, we use the next function

```
def decryptRSA(cipherBytes, key):
    try:
        return rsa.decrypt(cipherBytes, key)
    except:
        return False
```

This function is used when the CEO needs to send the encrypted files to all the directives, because the directive's AES key is encrypted with the CEO's public key, and the CEO needs to send the file encrypted with the AES key of the directive. Since the AES key is encrypted, it's decrypted with the CEO's private RSA key. The implementation of this function is showing bellow:

```
decAESKey = base64.b64decode(rsa.decryptRSA(dirEncAESKey, CEOPrivKey))
print("Decrypting AES key with CEO's private RSA key...")
```

- **AES decryption**

In this project, we use the decryption for the files that receive the directive from CEO, for the decryption we use AES and the next function

```
def decryptAES(key, cipherBytes, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(cipherBytes), 16)
```

When a directive wants to see a document, first, this document is cipher in AES mode, using the operation mode CBC, all the directives have the documents in their carpets, and then he only wants to select the document that he or she wants to sign. Like this

```

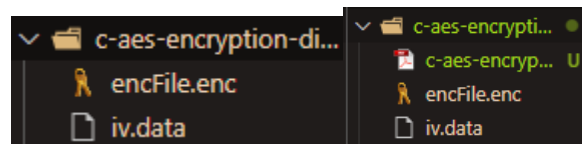
SIGN A DOCUMENT

Select a document to sign
  1. c-aes-encryption-diagram.pdf
  2. c-hash-notas.pdf
  3. Divide y vencerás.pdf
  4. substitution-box.png

Your option: 

```

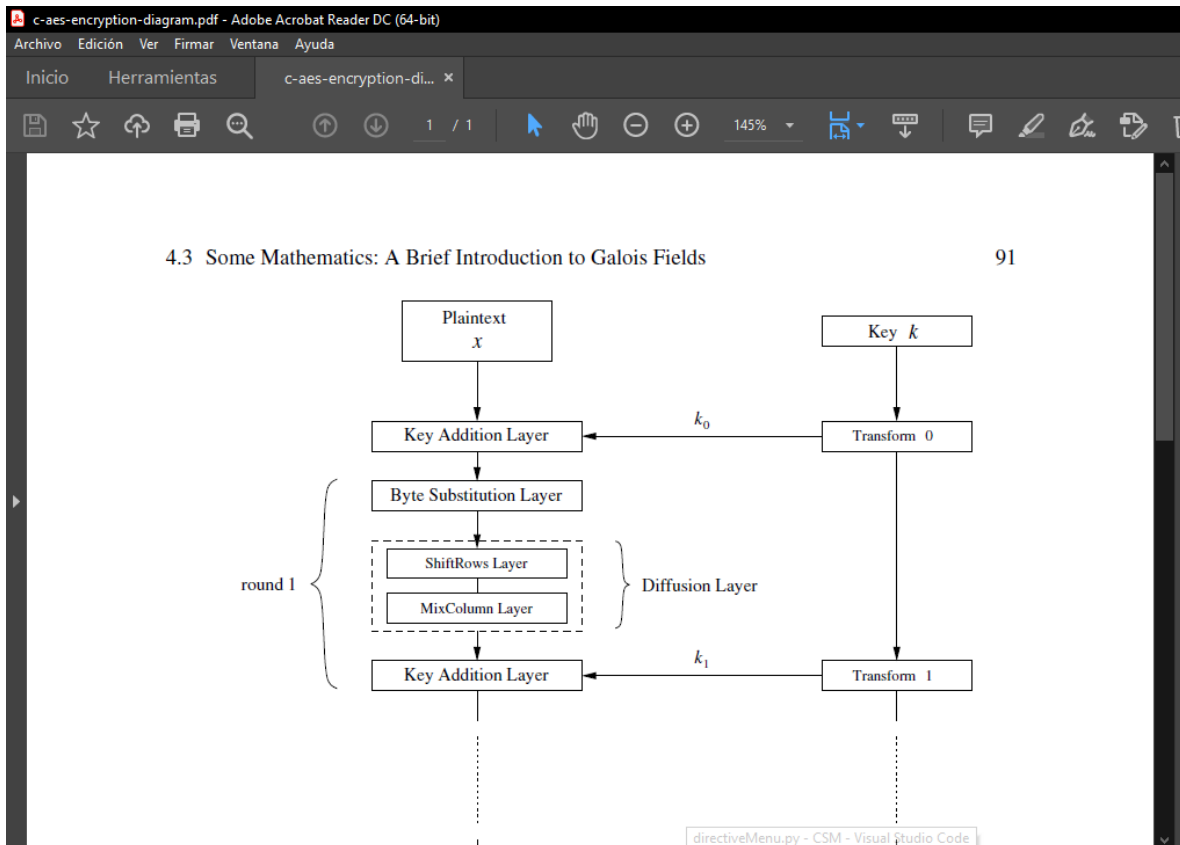
And the file that the directive selects, first, as we mention, is encrypted, we decrypted the document with the iv of the files that is saved in the directory that the directive selects, and the AES key of the directive. When the file is decrypted, this file saves in the directory of the filename, and this document is shown to the directive.



```

print("\nDecrypting " + file_sel_name + "...")
document_desc = Aes.decryptAES(key, document_sel, document_sel_iv)
fiMan.saveFile(routeDocSel, file_sel_name, '', document_desc)
fiMan.openFile2(routeDocSel, file_sel_name, "")|

```

Signature

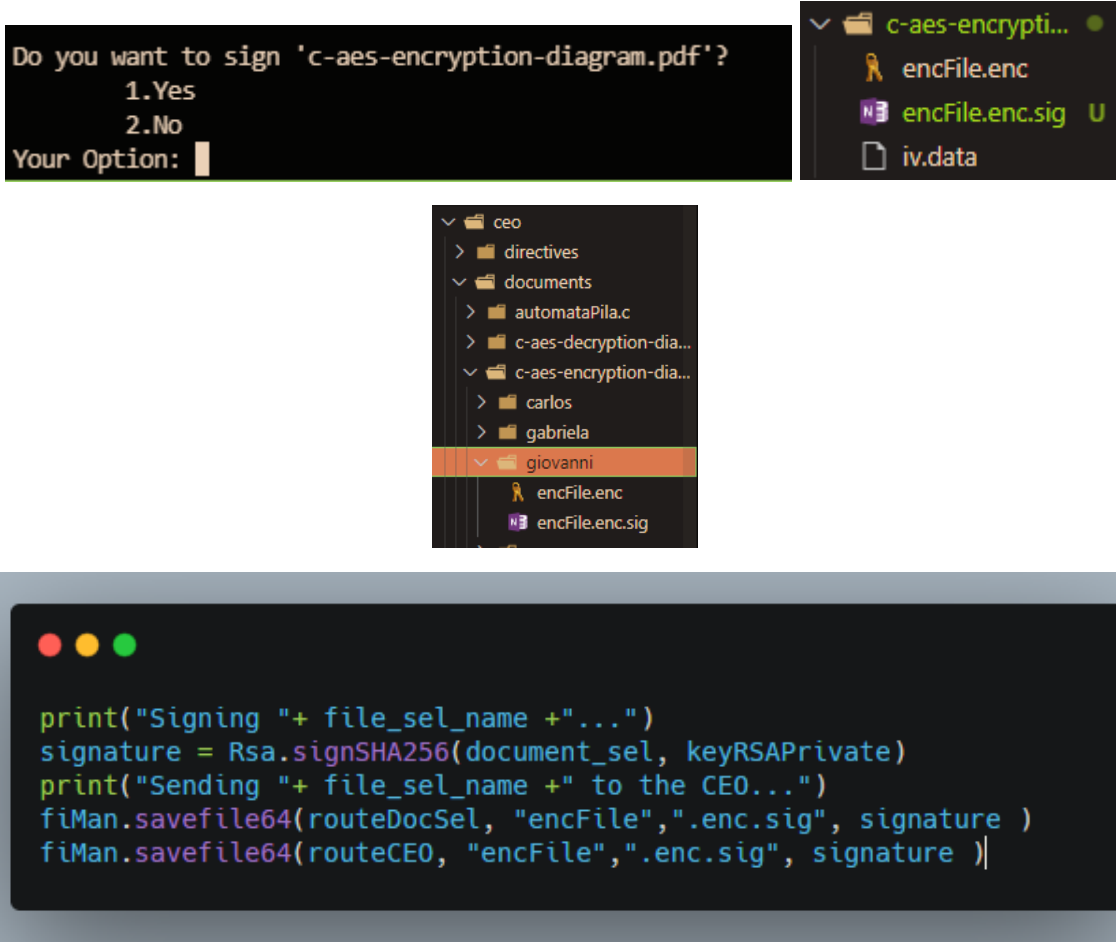
For the signature, we use the next function

```
def signSHA256(msgBytes, key):
    return rsa.sign(msgBytes, key, 'SHA-256')
```

This function we implemented with the library “rsa” of python and returns the sign. To complete the sign, we need the directive’s private key, and the directive’s RSA private key.

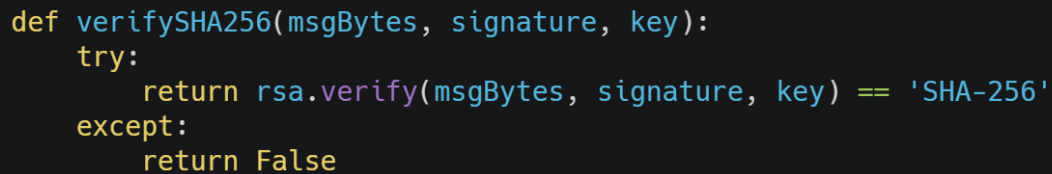
The program shows the directive if he wants to sign the document, if he selects yes, the file will sign with the private key of the directive and the encrypted document that he selects, this sing will send to the CEO, and it will store in the directory of the

document that he or she selects with the name “*encFile.enc.sig*” and in his own directories. If he selects no, the directive is redirect to the principal directives menu. And finally, when the directive is redirect to the menu, the document that showed to the directive is delete. The next pictures show the previous process.



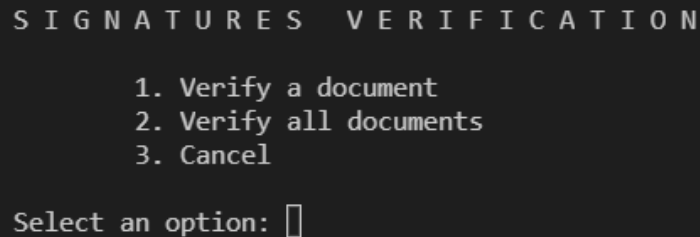
Signature verification

The next function is used to verify a signature using SHA-256.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains a Python function definition for verifying SHA256 signatures.

```
def verifySHA256(msgBytes, signature, key):  
    try:  
        return rsa.verify(msgBytes, signature, key) == 'SHA-256'  
    except:  
        return False
```

The CEO via the CEO's principal menu can verify all the directives' signatures of a document or all documents at once. The second option will be selected because it includes the process of verifying a single document.

A terminal-style window titled "SIGNATURES VERIFICATION" in all caps. It lists three options: "1. Verify a document", "2. Verify all documents", and "3. Cancel". At the bottom, it says "Select an option:" followed by a small square cursor.

```
SIGNATURES VERIFICATION  
  
1. Verify a document  
2. Verify all documents  
3. Cancel  
  
Select an option: □
```

The process of verifying the signatures of a single document consists of getting all the directives that must sign that document. For each directive, the respective RSA public key, the encrypted document generated when encrypting with AES and the signature are retrieved to verify if the signatures is valid. The following image illustrates part of this process.

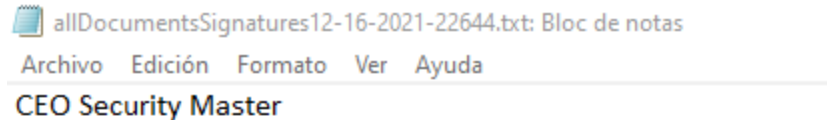
```
----- c-aes-decryption-diagram.pdf -----  
  
----- carlos -----  
  
carlos's RSA public key retrieved  
Retrieving encrypted file and signature...  
Verifying signature with RSA public key...  
carlos's signature is valid  
  
----- gabriela -----  
  
gabriela's RSA public key retrieved  
Retrieving encrypted file and signature...  
Verifying signature with RSA public key...  
gabriela's signature is valid  
  
----- giovanni -----  
  
giovanni's RSA public key retrieved  
Retrieving encrypted file and signature...  
Verifying signature with RSA public key...  
giovanni's signature is valid
```

This code performs the signature verification.



```
rsa.verifySHA256(document, signature, keyRSADirPub)
```

The report for an enhanced visualization of all signatures is also generated.



Directives' signatures:

- * carlos's signature is valid
- * gabriela has not signed this document
- * giovanni's signature is valid
- * marco has not signed this document
- * sandy has not signed this document

File: c-aes-decryption-diagram.pdf

Directives' signatures:

- * carlos's signature is valid
- * gabriela's signature is valid
- * giovanni's signature is valid
- * marco's signature is valid
- * sandy's signature is valid

Link to github repository

<https://github.com/Giovanni-Hernandez/CSM>

Link of the .zip folder in Drive

<https://drive.google.com/drive/folders/1zsmXK3geIplSSFshzqJ31FSoB2P704Fq?usp=sharing>

Bibliography

- Barker, E. (2021). NIST Special Publication 800-175B. Retrieved 1 December 2021, from <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175Br1.pdf>
- Barker, E. (2021). Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Length. Retrieved 1 December 2021, from <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf>
- Paar, C., & Pelzl, J. (2011). Understanding cryptography. Berlin: Springer.
- Knudsen, L., & Robshaw, M. (2011). The block cipher companion. Heidelberg: Springer-Verlag Berlin Heidelberg.