Information Security and Cryptography
Assignment 2

教授：游家牧

黃川哲、黃晟慶、陳勇瑜

1 PyCrypto and Cryptography install steps

PyCrypto

1. 至老師提供的網址"https://pypi.python.org/pypi/pycrypto" 下載 package
2. 解壓縮後會得到：
3. 輸入下面指令以安裝
   python setup.py build
   python setup.py install
4. 安裝後就可以在程式中 import package 了

Cryptography

1. 開啟 terminal 輸入下面指令
   `$ sudo pip install cryptography`
   會出現如下圖的 log：

2. 輸入下面指令安裝 openssl
   ```
   $ brew install openssl
   $ env LDFLAGS="-L(brew --prefix openssl)/lib" CFLAGS="-I$(brew --prefix openssl)/
   include" pip install cryptography
   ```

2 Code Explanation

2.1 Symmetric encryption

2.1.1 Pycrypto SHA-512

- `from Crypto.Hash import SHA512`

Import SHA-512 from package Pycrypto

- `message = open("512M+7",'r+b')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- `finished = False`
- `block_size = 1024`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits).

- ```
  while not finished:
      chunk = message.read(1024*block_size)
      if len(chunk) == 0:
          finished = True
      hash = SHA512.new(
      hash.update(chunk)
      result.write(hash.digest()
      print len(hash.digest())
  ```

If the finish flag is not true, keep reading a chunk from input file.
And test if it is empty or not. If yes, that means there is no data hasn't been read, so we set the flag.
If no, just do nothing.
We start to create a object "hash", and use it hash the chunk.
Then write the output to output file.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.
  ```

2.1.2 Cryptography SHA-512

- `from cryptography.hazmat.primitives import hashes`
- `from cryptography.hazmat.backends import default_backend`

Import SHA-512 from package cryptography.

- `message = open("512M+7",'r+b')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- `finished = False`
- `block_size = 1024`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits).

- ```
  while not finished:
      chunk = message.read(block_size*1024)
      if len(chunk) == 0:
          finished = True
      digest = hashes.Hash(hashes.SHA512(), backend=default_backend())
      digest.update(chunk)
      cipher_text = digest.finalize()
      result.write(cipher_text)
      print len(cipher_text)
  ```

If the finish flag is not true, keep reading a chunk from input file.
And test if it is empty or not. If yes, that means there is no data hasn't been read, so we set the flag.
If no, just do nothing.
We start to create a object "digest". We set it's backend to default, and use it hash the chunk.
Then write the output to output file.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.1.3 Pycrypto AES-256-ECB

- `from Crypto.Cipher import AES`
- `import os`

Import AES from package Crypto.Cipher, and os.

- `key = os.urandom(32)`
- `iv = 'This is an IV'`

Randomly create a 32bits unsigned data as a key, and set a string as IV.
  ```

- `finished = False`
- `block_size = AES.block_size`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits), we use the default value of AES here.

- `aes_256_ecb = AES.new(key, AES.MODE_ECB, iv)`

Set a aes-256 object with parameters we create brfore and using ecb mode.

- `message = open("512M+7",'r+b')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- ```
  while not finished:
      chunk = message.read(block_size*1024)
      if len(chunk) == 0 or len(chunk) % block_size != 0:
          padding_length = block_size - (len(chunk) % block_size)
          chunk += padding_length * chr(padding_length)
          finished = True
      result.write(aes_256_ecb.encrypt(chunk))
  ```

If the finish flag is not true, keep reading a chunk from input file.
And test if it is empty or not "OR" if the chunk is the last chunk of input and it need padding.
If yes, we calculate how many bits we need to pad, and add it to the chunk, and set the flag.
If no, just do nothing.
We call a function to encrypt the chunk, and write it to the output.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.1.4 Cryptography AES-256-ECB

- `import os`
- `from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes`
- `from cryptography.hazmat.primitives.ciphers.modes import ECB`
- `from cryptography.hazmat.backends import default_backend`

Import some packages, and os.

- `backend = default_backend()`
- `key = os.urandom(32)`
- `iv = os.urandom(16)`

Randomly create a 32bits unsigned data as a key, and a 16bits unsigned data as IV.
And set backend as default.
  ```

- `cipher = Cipher(algorithms.AES(key), ECB(), backend=backend)`

Set a aes-256 object with parameters we create brfore.

- `finished = False`
- `block_size =16`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits).

- `message = open("512M+7",'rb')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- ```
  while not finished:
      encryptor = cipher.encryptor()
      chunk = message.read(1024*block_size)
      if len(chunk) == 0 or len(chunk) % block_size != 0:
          padding_length = block_size - (len(chunk) % block_size)
          chunk += padding_length * chr(padding_length)
          finished = True
      result.write(encryptor.update(chunk) + encryptor.finalize())
  ```

If the finish flag is not true.
We set the value of ”encryptor” to ”cipher.encryptor()”.
And read a chunk from input file.
Test if it is empty or not ”OR” if the chunk is the last chunk of input and it need padding.
If yes, we calculate how many bits we need to pad, and add it to the chunk, and set the flag.
If no, just do nothing.
We call a function to encrypt the chunk, concatenate it with encryptor.finalize(), and write it to the output.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.1.5 Pycrypto AES-256-CBC

- `from Crypto.Cipher import AES`
- `import base64`

Import some packages, and ”base64” package.

- `key = 'This is a KEY123This is a key123'`
- `iv = 'This is an IV123'`

Randomly create a 32bits data as a key, and a 16bits data as IV.

- `finished = False`
  ```

- `block_size = AES.block_size`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits), we use the default value of AES here.

- `aes_256_cbc = AES.new(key, AES.MODE_CBC, iv)`

Using the data before to create a AES object.

- `message = open("512M+7",'r+b')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- ```
  while not finished:
      chunk = message.read(1024*block_size)
      if len(chunk) == 0 or len(chunk) % block_size != 0:
          padding_length = block_size - (len(chunk) % block_size)
          chunk += padding_length * chr(padding_length)
          finished = True
      result.write(encryptor.update(chunk))
  ```

If the finish flag is not true. We read a chunk from input file.
Test if it is empty or not "OR" if the chunk is the last chunk of input and it need padding.
If yes, we calculate how many bits we need to pad, and add it to the chunk, and set the flag.
If no, just do nothing.
We call a function to encrypt the chunk, and write it to the output.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.1.6 Cryptography AES-256-CBC

- `import os`
- `from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes`
- `from cryptography.hazmat.primitives.ciphers.modes import CBC`
- `from cryptography.hazmat.backends import default_backend`

Import some packages, and os.

- `backend = default_backend()`
- `key = os.urandom(32)`
- `iv = os.urandom(16)`

Randomly create a 32bits unsigned data as a key, and a unsigned 16bits data as IV.
And set backend as default.

- `cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)`
  ```

Set a aes-256 object with parameters we create brfore.

- `message = open("512M+7",'rb')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- `finished = False`
- `block_size =16`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits).

- ```
  while not finished:
      encryptor = cipher.encryptor()
      chunk = message.read(1024*block_size)
      if len(chunk) == 0 or len(chunk) % block_size != 0:
          padding_length = block_size - (len(chunk) % block_size)
          chunk += padding_length * chr(padding_length)
          finished = True
      result.write(encryptor.update(chunk) + encryptor.finalize())
  ```

If the finish flag is not true.
We set the value of ”encryptor” to ”cipher.encryptor()”.
And read a chunk from input file.
Test if it is empty or not ”OR” if the chunk is the last chunk of input and it need padding.
If yes, we calculate how many bits we need to pad, and add it to the chunk, and set the flag.
If no, just do nothing.
We call a function to encrypt the chunk, concatenate it with encryptor.finalize(), and write it to the output.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.1.7 Pycrypto AES-256-CTR

- `import os`
- `from Crypto.Cipher import AES`
- `from Crypto import Random`
- `from Crypto.Util import Counter`

Import some packages, and os.

- `key = os.urandom(32)`

Randomly create a 32bits unsigned data as a key.

- `rand_gen = Random.new()`
  ```

- `iv = rand_gen.read(8)`
- `ctr_e = Counter.new(64, prefix=iv)`

Create a random object, and use it to set initial value of iv.
Create a counter object by iv.

- `aes_256_ctr = AES.new(key, AES.MODE_CTR, counter=ctr_e)`

Set a aes-256 object with parameters we create brfore.

- `message = open("512M+7",'rb')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- `finished = False`
- `block_size = AES.block_size`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits), we use the default value of AES here.

- ```
  while not finished:
      chunk = message.read(1024*block_size)
      if len(chunk) == 0 or len(chunk) % block_size != 0:
          padding_length = block_size - (len(chunk) % block_size)
          chunk += padding_length * chr(padding_length)
          finished = True
      result.write(aes_256_ctr.encrypt(chunk))
  ```

If the finish flag is not true, we read a chunk from input file.
Test if it is empty or not "OR" if the chunk is the last chunk of input and it need padding.
If yes, we calculate how many bits we need to pad, and add it to the chunk, and set the flag.
If no, just do nothing.
We call a function to encrypt the chunk, and write it to the output.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.1.8 Cryptography AES-256-CTR

- `import os`
- `from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes`
- `from cryptography.hazmat.primitives.ciphers.modes import CBC`
- `from cryptography.hazmat.backends import default_backend`

Import some packages, and os.

- `backend = default_backend()`
  ```

- `key = os.urandom(32)`
- `nonce = os.urandom(16)`

Randomly create a 32bits unsigned data as a key, and a 16bits unsigned data as nonce.
And set backend as default.

- `cipher = Cipher(algorithms.AES(key), modes.CTR(nonce), backend=backend)`

Set a aes-256 object with parameters we create brfore.

- `message = open("512M+7",'rb')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- `finished = False`
- `block_size =16`

Set two variables, the former is a flag to show if the input has finished.
The latter is a constant which is the size of a block(bits).

- ```
  while not finished:
      encryptor = cipher.encryptor()
      chunk = message.read(1024*block_size)
      if len(chunk) == 0 or len(chunk) % block_size != 0:
          padding_length = block_size - (len(chunk) % block_size)
          chunk += padding_length * chr(padding_length)
          finished = True
      result.write(encryptor.update(chunk) + encryptor.finalize())
  ```

If the finish flag is not true.
We set the value of ”encryptor” to ”cipher.encryptor()”.
And read a chunk from input file.
Test if it is empty or not ”OR” if the chunk is the last chunk of input and it need padding.
If yes, we calculate how many bits we need to pad, and add it to the chunk, and set the flag.
If no, just do nothing.
We call a function to encrypt the chunk, concatenate it with encryptor.finalize(), and write it to the output.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.2 Asymmetric encryption

2.2.1 RSA key create

Since RSA is a Asymmetric encryption, we will need two keys, public key and private key.
We use openssl to create them. And use two statement below to translate them to other codings.

- `openssl genrsa -out privkey.pem 2048`
- `openssl pkcs8 -topk8 -in rsa_private_key_2048.pem -out pkcs8_rsa_private_key_2048.pem -nocrypt`
  ```

2.2.2 Pycrypto RSA-2048

- `import base64`
- `from Crypto.PublicKey import RSA`
- `from Crypto.Cipher import PKCS1_v1_5`
- `from Crypto.Hash import SHA`
- `from Crypto import Random`

Import some packages, and os.

- ```
  with open('rsakey/rsa_public_key_2048.pem', 'r') as f:
      pub = f.read()
  ```
- ```
  with open('rsakey/pkcs8_rsa_private_key_2048.pem', 'r') as f:
      pri = f.read()
  ```

Open two files, one is public key and another is private key. And read them to two variables.

- `finished = False`
- `block_size = 245`
- `dsize = SHA.digest_size`

Set three variables, the first one is a flag to show if the input has finished.
The second one is a constant which is the size of a block(bits), we use the maximum value of RSA can accept here.
The third one is a digest size, we use SHA default digest size here.

- `message = open("512M+7",'r+b')`
- `result = open("512M+7_result", 'wb')`

Open the flow of two file. To read the former and write the latter.
512M+7 is the file we randomly create.
And 512M+7 result is a empty file to store our hashed output.

- ```
  while not finished:
      chunk = message.read(block_size)
      if len(chunk) == 0:
          finished = True
      key = RSA.importKey(pub)
      sentinel = Random.new().read(15 + dsize)
      cipher = PKCS1_v1_5.new(key)
      cipher_text = base64.b64encode(cipher.encrypt(chunk))
      privkey = RSA.importKey(pri)
      cipher2 = PKCS1_v1_5.new(privkey)
      print cipher2.decrypt(cipher.encrypt(chunk), sentinel)==chunk
  ```

If the finish flag is not true, we read a chunk from input file.
Test if it is empty or not.
If yes, we set the flag. If no, just do nothing.
Use "RSA.importKey(pub)" to import public key.
Create a random object and take a str whose length is dsize+15 from random object.
Use the public key to create a "`PKS_v1.5`" object, and call it cipher.
Then use it encrypt our chunk and encode to b64.
  ```

Use "RSA.importKey(pri)" to import private key.
Use the private key to create a "`PKS_v1.5`" object, and call it cipher2.
And use it to decrypt the encrypt chunk with sentinel, if they are the same, we'll know since it'll be print out.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

2.2.3 Cryptography RSA-2048

- `import base64`
- `from cryptography.hazmat.primitives import serialization`
- `from cryptography.hazmat.backends import default_backend`
- `from cryptography.hazmat.primitives.asymmetric import rsa`
- `from cryptography.hazmat.primitives.asymmetric import padding`
- `from cryptography.hazmat.primitives import hashes`

Import some packages, and os.

- ```
  def load_public_key():
      with open("./rsakey/public.pem", "rb") as key_file:
          b64data = '\n'.join(
                    key_file.read().decode('utf-8').splitlines()[1:-1])
          derdata = base64.b64decode(b64data)
          return serialization.load_der_public_key(
              derdata, backend=default_backend())
  ```

A function about loading publick key.
Read public key in the file first, decode it ,and use "`serialization.load_der_public_key()`".
Use deocded public key and default backend to set its parameter.

- ```
  def load_private_key():
      with open("./rsakey/private.pem", "rb") as key_file:
          b64data = '\n'.join(
                    key_file.read().decode('utf-8').splitlines()[1:-1])
          derdata = base64.b64decode(b64data)
          return serialization.load_der_private_key(
              derdata, password=None, backend=default_backend())
  ```

A function about loading publick key.
Almost the same as load public key.

- ```
  def get_ciphertext(plaintext):
      public_key = load_public_key()
      return public_key.encrypt(
          plaintext.encode('utf-8'),
          padding.OAEP(
              mgf=padding.MGF1(algorithm=hashes.SHA1()),
              algorithm=hashes.SHA1(),
              label=None
          )
      )
  ```
  ```

A function about using publick key to encrypt our plaintext.
We call "`load_public_key()`" to load public key.
And return ciphertext by using "`public_key.encrypt()`" with some appropriate parameter.

- ```
  def get_plaintext(ciphertext):
      private_key = load_private_key()
      return private_key.decrypt(
          ciphertext,
          padding.OAEP(
              mgf=padding.MGF1(algorithm=hashes.SHA1()),
              algorithm=hashes.SHA1(),
              label=None
          )
      )
  ```

A function about using private key to dencrypt our ciphertext.
We call "`load_private_key()`" to load private key.
And return plaintext by using "`private_key.encrypt()`" with some appropriate parameter.

- ```
  def encrypt(message):
      BLOCK_SIZE = 214

      TOTAL_BLOCKS = int(len(message) / BLOCK_SIZE)

      blocks = []
      block_content = ''
      for word in message:
          block_content += word
          if len(block_content) == BLOCK_SIZE:
              ciphertext = get_ciphertext(block_content)
              plaintext = get_plaintext(ciphertext)

              if plaintext.decode('utf-8') == block_content:
                  block_content = ''
                  blocks.append(ciphertext)
                  print('Finished ' + str(len(blocks)) + ' blocks')
                  print(str(TOTAL_BLOCKS - len(blocks)) + ' to go...')
              else:
                  print('ERROR: Plaintext does not match decrypted data')
                  return None

      return blocks
  ```

A function about how to implement encrypt.
We set block size as 214byte, because (2048/8) -42 = 256 -42 = 214
Calcuate the block numbers, create an empty block array, and an empty string to store contents of block.
And for each character in message, we add it to block string until string is full.
Then we call functions to encrypt it, and then also decrypt it so we can check if it's right.
We decode out plaintext, if it's equal to content of block string, we clear string, and copy ciphertext to block array.
And print some detail message, but if the decoded plaintext is not equal to block string, print out error and return.

- ```
  if __name__ == '__main__':
  ```
  ```

```
with open("512M+7", 'r+b') as f_message, open("512M+7_result", 'wb') as f_result:
     f_result.write(encrypt(f_message.read().decode('utf-8')))
     f_message.close()
     f_result.close()
```

So in the main function, we only need to open two file as input and output.
And use "encrypt()" to encrypt decoded input then write it to output file.
And close the two files at the end.

- `message.close()`
- `result.close()`

At the end of the program, exit the flow of two files.

3 Time comparison of implement

Pycrypto	SHA-512	AES-ECB	AES-CBC	AES-CTR	RSA-2048
Time	2.36s	4.58s	4.10s	5.53s	
Cryptography	SHA-512	AES-ECB	AES-CBC	AES-CTR	RSA-2048
Time	1.53s	1.81s	2.81s	1.83s	

Pycrypto RSA-2048, Cryptography RSA-2048

And there is a problem is, when we implement both package of RSA-2048, we found because of the block size limit. The loop times will become too large to execute. So the program will process forever.
We are sure it's not infinite loop or whatever else. And we can't find a better way to solve this problem, so we don't have execute time images of RSA-2048.