UNIVERSITA' DEGLI STUDI DI CAGLIARI

FACOLTÀ DI SCIENZE

Corso di Laurea in Informatica

Titolo

Docente di riferimento

Prof. Massimo Bartoletti

Candidato

Michele Melis (matr.65798)

ANNO ACCADEMICO 2022-2023

Indice

Introduzione

1. Ambiente di sviluppo e strumenti

1.1. Python

Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti. Il suo design agevola la leggibilità del codice, permettendo di focalizzarsi maggiormente sull'idea da implementare rispetto che sull'implementazione. È uno dei linguaggi di programmazione più utilizzati nel mondo e la sua fama è ancora in crescita.

Codifica e decodifica di caratteri

La codifica è un'operazione che associa un carattere ad un valore (spesso numerico) comprensibile ad un elaboratore.

Nei capitoli a seguire verrà utilizzata maggiormente la codifica *UTF-8* in quanto ampiamente utilizzata e standard di Python.

UTF-8 utilizza un solo byte per tutti i caratteri ASCII, due o più per tutti gli altri caratteri.

In python possono essere chiamati i metodi *encode* e *decode* su una variabile di tipo stringa per effettuarne la codifica e/o la decodifica. I metodi prendono come parametro il nome della codifica. Per la codifica *UTF-8* può essere omesso.

```
message = 'Il gatto è sul tavolo.'
encoded_message = message.encode('UTF-8') # b'Il gatto \xc3\xa8 sul tavolo.'
decoded_message = encoded_message.decode() # 'Il gatto è sul tavolo.'
```

1.2. Librerie e moduli

PyCryptoDome

La libreria *PyCryptoDome* è una collezione di primitive crittografiche a basso livello. È una versione popolare e migliorata dell'ultima versione dell'ormai deprecata libreria *PyCrypto*.

La libreria introduce nuove funzionalità e algoritmi crittografici che verranno studiati ed utilizzati nei capitoli a seguire. Le funzionalità sono suddivise in pacchetti (package) contenenti i vari algoritmi crittografici. Tutti gli algoritmi sono stati implementati in puro Python, eccetto alcuni algoritmi critici che sono stati implementati come estensioni in linguaggio C per migliorarne le prestazioni.

```
pip install pycryptodome # Installazione libreria
```

os

Il modulo (built-in) *os* contiene una serie di funzionalità compatibili con il sistema operativo in uso.

La sua portabilità ed efficienza lo rendono essenziale per l'interazione con il file system e l'utilizzo di funzioni a basso livello messe a disposizione dal sistema operativo.

```
import os # Importazione modulo
```

hashlib

Il modulo (built-in) *hashlib* implementa un'interfaccia ai più comuni algoritmi di hashing.

Il modulo offre degli algoritmi implementati in linguaggio a basso livello (assembly) utilizzando istruzioni AVX (Advanced Vector Extension), il che lo rende più performante rispetto agli algoritmi di hashing implementati con *PyCryptoDome*.

```
import hashlib # Importazione modulo
```

hmac

Il modulo (built-in) *hmac* implementa l'algoritmo HMAC per la generazione di codici di autenticazione.

Il modulo utilizza delle librerie a basso livello tra cui gli algoritmi di hashing (implementati nel modulo *hashlib*) rendendolo un modulo efficiente e sicuro.

```
import hmac # Importazione modulo
```

secrets

Il modulo (built-in) *secrets* è utilizzato per la generazione di valori numerici casuali ritenuti fortemente sicuri per l'utilizzo in ambito crittografico.

```
import secrets # Importazione modulo
```

2. Hashing

Gli algoritmi di hashing permettono di garantire privacy, integrità ed autenticità di un'informazione. Le funzioni di hashing prendono in input delle stringhe binarie e producono un output casuale di lunghezza fissa chiamato *digestione* o *valore hash*. Data la natura degli algoritmi, è impossibile stabilire quale input è stato utilizzato per generare un certo output. Gli algoritmi forniscono anche una forte resistenza alle collisioni.

La libreria *PyCryptoDome* ci offre una serie di algoritmi, tra cui le famiglie degli algoritmi SHA-2, SHA-3 e BLAKE2. La libreria offre anche una serie di funzioni per la generazione di un valore hash di lunghezza variabile e degli algoritmi di autenticazione messaggi, tra cui: HMAC, CMAC e Poly1305.

Di seguito viene mostrato come effettuare una semplice digestione utilizzando l'algoritmo SHA-1:

```
from Crypto.Hash import SHA1 # Importazione classe
hashing = SHA1.new()         # Inizializzazione oggetto
hashing.update(b'Ciao, ')    # Aggiornamento hash
hashing.update(b'mondo!')    # Aggiornamento hash
hashing.hexdigest()          # Digestione
'f2a670cebb772a49e4cbe65d64c6688c8a02350f' # Valore restituito
```

Nel corso del capitolo impareremo a proteggere una password salvata in un contenitore non sicuro, ad effettuare il controllo d'integrità di un dato e a verificare l'autenticità di un messaggio.

2.1. Sicurezza di una password

Per garantire la sicurezza di una password in un contenitore non sicuro, viene utilizzato un algoritmo di hashing. Prendiamo ad esempio il database di un social network: se la password fosse salvata in chiaro nel database, quest'ultima potrebbe essere letta ed appresa da terzi. Utilizzando l'hashing, invece, solo l'utente conosce la propria password.

Vedremo come implementare un algoritmo di hashing per nascondere una password mantenendo possibile l'autenticazione da parte dell'utente. Nel nostro esempio andremo a salvare la password in un file, ma lo stesso procedimento può essere utilizzato anche per la memorizzazione in un altro supporto (ad esempio un database).

Per prima cosa viene importata la classe dell'algoritmo utilizzato (nel nostro caso SHA-512):

```
from Crypto.Hash import SHA512
```

Viene poi inizializzato l'oggetto che si occuperà di effettuare la digestione:

```
hashing = SHA512.new()
```

Viene chiesto all'utente di inserire una password:

```
password = input("Inserire una nuova password: ")
```

Viene infine eseguita la digestione e viene salvato il valore su file:

```
hashing.update(password.encode())
digest = hashing.hexdigest()
with open('digested_password.txt', 'wb') as file:
  file.write(digest.encode())
```

Il metodo *update()* permette di incorporare nuovi blocchi di informazione da cifrare. In questo caso viene passato come parametro la password.

Il metodo *hexdigest()* esegue la digestione della password e restituisce il risultato in formato esadecimale (128 bytes con SHA512, due byte per carattere).

Nell'esempio verrà creato un file chiamato *digested_password.txt* che conterrà il valore hash della password.

Aggiungiamo poi tutti i controlli e i comandi che permetteranno all'utente di reinserire la password per confrontarla con quella precedente.

Verrà quindi chiesto all'utente di reinserire la propria password e, a seconda che il valore hash della password inserita corrisponda o meno con il valore hash della precedente password, verrà stampato su terminale il risultato dell'operazione:

```
password = input("Renserire la password: ")
hashing.update(password.encode())
digested_password = hashing.hexdigest()
if digested_password == previous_digest:
  print("Password accettata.")
else:
  print("Password rifiutata.")
```

2.2. Controllo integrità dati

Uno dei principali utilizzi degli algoritmi di hashing è il controllo dell'integrità di un'informazione.

I due algoritmi comunemente più utilizzati per svolgere questo compito sono: MD5 e SHA256.

Nel nostro esempio li utilizzeremo entrambi per calcolare il valore checksum di un file e successivamente per assicurarci che il file sia integro e che non sia stato compromesso.

Il file (video) del quale calcoleremo il checksum è disponibile a questo link.

Come prima cosa vengono importate le classi degli algoritmi utilizzati:

```
from Crypto.Hash import MD5, SHA256
```

Successivamente viene impostata una dimensione (in bit) dei blocchi che verranno letti per volta dal file:

```
BLOCKSIZE = 65536 # 8192KB
```

Vengono dichiarati gli oggetti utilizzati per effettuare la digestione:

```
md5 = MD5.new()
sha256 = SHA256.new()
```

Vengono letti tutti i blocchi dal file e successivamente vengono incorporati ai rispettivi input dei due algoritmi:

```
with open(filename,"rb") as file:
  buffer = file.read(BLOCKSIZE)
  while len(buffer) > 0:
    md5.update(buffer)
    sha256.update(buffer)
    buffer = file.read(BLOCKSIZE)
```

Viene effettuata infine la digestione:

```
md5_digest = md5.hexdigest()
sha256_digest = sha256.hexdigest()
```

I valori di checksum prodotti per il nostro file sono rispettivamente:

- d9061d3da8601932e98f79ec8ba1c877 (MD5)
- 71944d7430c461f0cd6e7fd10cee7eb72786352a3678fc7bc0ae3d410f72aece (SHA256)

Da notare che per file molto grandi il tempo impiegato per l'esecuzione dello script è maggiore e questo potrebbe avere ripercussioni sull'efficienza dell'applicazione. Questo può essere migliorato andando a sostituire le funzioni della libreria *PyCryptoDome* con le funzioni della libreria *hashlib*.

Propaghiamo quindi le modifiche al nostro esempio. Sostituiamo l'importazione della libreria *PyCryptoDome* con il modulo *hashlib*:

```
import hashlib
```

Sostituiamo le dichiariazioni delle classi:

```
md5 = hashlib.md5()
sha256 = hashlib.sha256()
```

2.3. Autenticazione messaggio

Quando un messaggio viene trasmesso, questo potrebbe essere intercettato prima dell'arrivo al destinatario ed essere sostituito con un messaggio contraffatto. Il miglior modo per far si che il messaggio contraffatto non venga scambiato per quello originale è quello di utilizzare un sistema MAC (Message Authentication Code). A differenza degli algoritmi di hashing, HMAC prende in input tre parametri: il messaggio originale, un algoritmo di hashing e una chiave segreta.

Nell'esempio andremo a generare un codice di autenticazione utilizzando l'algoritmo HMAC con SHA256, simulando le operazioni effettuate dal mittente e dal destinatario di un messaggio.

Per prima cosa vengono importate le classi utilizzate:

```
from Crypto.Hash import HMAC, SHA256
```

Viene poi dichiarata la chiave utilizzata dall'algoritmo e il messaggio da trasmettere.

```
SECRET = 'chiavesegreta123'
message = 'Ciao, come va?'
```

Successivamente, viene inizializzato l'oggetto HMAC e viene eseguita la digestione:

```
hmac = HMAC.new(SECRET.encode(), message.encode(), SHA256)
mac = hmac.hexdigest()
```

A questo punto il mittente trasmette il messaggio.

Il messaggio (insieme al MAC) può essere trasmesso tramite una connessione non sicura (consapevoli del fatto che il messaggio potrebbe essere intercettato e letto da terzi), mentre la chiave deve essere trasmessa in modo sicuro per far si che non possa essere contraffatta. Nei prossimi capitoli vedremo come cifrare un messaggio prima di una trasmissione e come condividere una chiave privata in modo sicuro, per ora supponiamo che il mittente e il destinatario siano già in possesso della medesima chiave e che il messaggio da trasmettere non abbia bisogno di essere cifrato.

Il destinatario, che già conosce la password, riceve il messaggio insieme al MAC. Viene quindi creato ed inizializzato l'oggetto HMAC, andando a verificare l'autenticità del messaggio.

```
hmac = HMAC.new(SECRET.encode(), message.encode(), SHA256)
try:
  hmac.hexverify(mac)
  print('Il messaggio è autentico.')
except ValueError:
  print('Il messaggio NON è autentico.')
```

Da notare che il metodo *hexverify* lancia un'eccezione di tipo **ValueError** se il MAC ricevuto non corrisponde a quello atteso dal destinatario.

Python ci offre un modulo HMAC built-in più performante di quello fornito dalla libreria *PyCryptoDome*. Il modulo in questione è chiamato *hmac* e può essere utilizzato per migliorare i tempi di esecuzione del programma. È possibile visualizzare l'implementazione del modulo nello script d'esempio.

3. Random

La casualità nel mondo della crittografia gioca un ruolo fondamentale. Un numero casuale viene generato tramite RNG (Random Number Generator) per essere utilizzato successivamente come chiave o come vettore iniziale. In informatica non esiste un algoritmo che ci restituisca un valore realmente casuale, questo avviene in quanto un elaboratore è deterministico per sua natura.

La libreria *PyCryptoDome* ci propone il package *Crypto.Random* per la generazione di valori (pseudo)casuali. Il package fornisce il metodo *get_random_bytes(N)* per la generazione di una stringa di byte di lunghezza N.

Il modulo *random* del package ci offre invece le seguenti operazioni per la generazione o scelta di un valore (pseudo)casuale:

- *random.getrandbits(N)*: genera un intero di lunghezza N bit
- *random.randrange([start,]stop[, step])*: genera un intero compreso nel range definito sull'insieme dei valori ottenibili partendo dal valore start e arrivando al valore stop con passo step
- *random.randint(a, b)*: genera un intero nel range a-b (a incluso, b escluso)
- *random.choice(seq)*: sceglie un elemento casuale presente nella data sequenza seq
- *random.shuffle(seq)*: mischia e restituisce la sequenza seq passata come parametro
- *random.sample(population, k)*: sceglie e restituisce casualmente k elementi presenti nella lista population

Come si può leggere dalla documentazione del package, *Crypto.Random* genera dei valori casuali utilizzando (tramite chiamate di sistema) funzioni e metodi implementati dal sistema operativo sul quale viene eseguito il programma (*Random numbers get sourced directly from the OS*). In parole povere il package di *PyCryptoDome* non è nient'altro che un incarto (wrapper) del metodo *os.urandom*, offrendoci una serie di funzioni già implementate e pronte all'uso.

Per svolgere operazioni di questo tipo viene comunemente consigliato l'utilizzo del modulo built-in *secrets* di Python, che svolge le medesime funzioni del package *Crypto.Random* seppur con un numero di metodi pre-implementati inferiore. Il modulo in questione è a sua volta un wrapper del metodo *os.urandom*.

Su Github possiamo trovare le implementazioni dei due moduli: *Crypto.Random.random* e *secrets*.

Effettuiamo un confronto sui tempi di esecuzione dei metodi principali di ciascun modulo:

```
from Crypto import Random
import secrets, os, time

LENGTH = 100000000 # 100'000'000 ns

start = time.time()
token = os.urandom(LENGTH)
end = time.time() - start
print("os:      ", end)
start = time.time()
token = secrets.token_bytes(LENGTH)
end = time.time() - start
print("secrets: ", end)
start = time.time()
token = Random.get_random_bytes(LENGTH)
end = time.time() - start
print("Random:  ", end)
```

Risultati:

```
>> os:       0.03300023078918457
>> secrets:  0.04400229454040527
>> Random:   0.046004295349121094
```

In conclusione: sia i metodi implementati nel package *Crypto.Random* che i metodi implemenentati nel modulo *secrets* sono ritenuti sicuri ed affidabili per la generazione di valori pseudo(casuali) in un'applicazione crittografica in quanto basati sull'entropia dell'algoritmo più efficiente implementato dal sistema operativo in uso, mentre il tempo di esecuzione aumenta all'aumentare del livello di astrazione dell'implementazione di tale funzione.

4. IO

Per la memorizzazione e la trasmissione di una chiave vengono utilizzati dei formati specifici per tale compito. I formati più utilizzati sono: PEM e PKCS#8, entrambi implementati dal package *Crypto.IO* di *PyCryptoDome*.

Il formato PEM appartiene ad un vecchio standard, ancora ampiamente utilizzato nei certificati di sicurezza utili a stabilire un canale di comunicazione sicuro tra client e server.

Il modulo ci offre le seguenti funzioni:

- *PEM.encode(data, marker, passphrase=None, randfunc=None): stringa codificata*
  - *data*: stringa binaria da codificare
  - *marker*: tipo di chiave
  - *passphrase*: password dalla quale derivare una chiave di cifratura per il blocco PEM
  - *randfunc*: funzione per la generazione di un numero casuale. La funzione deve prendere in ingresso un valore N e deve restituire una stringa di lunghezza N byte.
- *PEM.decode(pem_data, passphrase=None): (data, marker, encrypted)*
  - *pem_data*: stringa di dati in formato PEM
  - *passphrase*: password utilizzata per la cifratura dei dati durante la codifica

```
# Esempio
from Crypto.IO import PEM
# Codifica
pem = PEM.encode(
  data = b'dati da salvare',
  marker = 'KEY TYPE',
  passphrase = b'password segreta opzionale'
)
print('Codifica:')
print(pem)
# Decodifica
dec = PEM.decode(
  pem_data = pem,
  passphrase = b'password segreta opzionale'
)[0]
print('Decodifica:')
print(dec)
# Risultato
```

```
Codifica:
-----BEGIN KEY TYPE-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,84528DF2841A636F

oyHrGKKIm3Q9EgJXybMeBA==
-----END KEY TYPE-----
Decodifica:
b'dati da salvare'
```

Il formato PKCS#8 è una sintassi standard per la memorizzazione delle informazioni di una chiave privata. La chiave può essere cifrata utilizzando un algoritmo di derivazione oppure tenuta in chiaro.

Il modulo ci offre le seguenti funzioni:

- *PKCS8.wrap(private_key, key_oid, passphrase=None, protection=None, prot_params=None, key_params=<Crypto.Util.asn1.DerNull object>, randfunc=None): stringa codificata*
  - *private_key*: chiave privata codificata in byte
  - *key_oid*: l'identificatore dell'oggetto
  - *passphrase*: password dalla quale derivare la chiave per la cifratura
  - *protection*: l'identificatore dell'algoritmo da utilizzare per cifrare la chiave
  - *prot_params*: dizionario con i parametri da passare all'algoritmo di wrapping
  - *key_params*: i parametri da usare nella sequenza dell'algoritmo di identificazione
  - *randfunc*: funzione per la generazione di un numero casuale. La funzione deve prendere in ingresso un valore N e deve restituire una stringa di lunghezza N byte.
- *PKCS8.unwrap(p8_private_key, passphrase=None): (key_oid, private_key, key_params)*
  - *p8_private_key*: chiave privata codificata in PKCS#8
  - *passphrase*: password utilizzata per la cifratura dei dati durante la codifica

```
# Esempio
from Crypto.IO import PKCS8
# Wrapping
pkcs8 = PKCS8.wrap(
  private_key = b'chiave privata da registrare',
  key_oid = '1.2.840.113549.1.1.1',
  passphrase = b'password segreta opzionale',
  protection = 'scryptAndAES256-CBC'
)
print('Wrap:')
print(pkcs8)
# Unwrapping
key = PKCS8.unwrap(
  p8_private_key = pkcs8,
  passphrase = b'password segreta opzionale'
)[1]
print('Unwrap:')
print(key)
# Risultato
```

```
Wrap:
b'0\x81\x930O\x06\t*\x86H\x86\xf7\r\x01\x05\r0B0!\x06\t
+\x06\x01\x04\x01\xdaG\x04\x0b0\x14\x04\x08\xac}
\xc5\xa9L0\x85\xe2\x02\x02@\x00\x02\x01\x08\x02\x01\x010\x1d\x06\t
`\x86H\x01e\x03\x04\x01*\x04\x10Q\xe6b\x8f"(\x96\xf3\x94;\x81\xa5\x1fd(\xfc\x04@\xea\xb4p,?
\xee9U\xd6\x9c"qj7\x82\x07\x88\x0cE\xaa\xffs\x16d\xb9\xcb^\x17a\n\xc7P\xbd
+\xda\xa6\x08\x0cQ\x0c\x96\xd9\xc9\x10\x9e\xd8\xeb(\x9d\x99c\x9a\x15w\xc5\x92U>%/
\x8f\x94\x83\xe8'
Unwrap:
b'chiave privata da registrare'
```

5. Cifratura

Gli algoritmi di cifratura ci permettono di garantire la privacy di comunicazioni e informazioni. Esistono tre tipi di cifratura: cifratura simmetrica, cifratura asimmetrica e una combinazione delle due. Gli algoritmi di cifratura simmetrica sono più veloci e possono processare un numero di dati superiore rispetto agli algoritmi di cifratura asimmetrica.

La libreria *PyCryptoDome* ci offre una serie di algoritmi utilizzabili, tra cui: Salsa20, ChaCha20, XChaCha20 e AES.

Nel corso del capitolo impareremo a cifrare un messaggio tramite cifratura a flusso e un file tramite cifratura e blocco.

5.1. Cifratura messaggio a flusso

Abbiamo un messaggio che vogliamo mandare a qualcuno senza che terzi possano avere accesso al suo contenuto. Per farlo abbiamo bisogno di un metodo sicuro ed efficiente da utilizzare. La libreria *PyCryptoDome* ci offre tre algoritmi di cifratura a flusso: *Salsa20*, *ChaCha20* e *XChaCha20*.

Per il nostro esempio andremo ad utilizzare l'algoritmo *Salsa20* in quanto uno dei più efficienti ed efficaci.

```
from Crypto.Cipher import Salsa20
```

Andiamo a costruire un sistema di cifratura e decifratura di un messaggio tramite chiave privata.

Per prima cosa viene definito il messaggio da cifrare e viene generata una chiave privata:

```
plaintext = input("Inserire un messaggio da cifrare: ").encode()
key = secrets.token_bytes(32)
```

Inizializziamo l'oggetto utilizzato per la cifratura e con esso cifriamo il messaggio, andando a porre in testa il vettore iniziale generato dallo stesso oggetto:

Viene inizializzato l'oggetto utilizzato per la cifratura e con esso viene cifrato il messaggio. L'oggetto genera in automatico un vettore iniziale.

```
cipher = Salsa20.new(key)
message = cipher.nonce + cipher.encrypt(plaintext)
```

Il destinatario deve essere messo al corrente del metodo utilizzato per la trasmissione del vettore iniziale. Nel nostro caso verrà trasmesso in testa al messaggio cifrato.

Nello script d'esempio vengono salvati sia il messaggio cifrato che la chiave nei rispettivi file: *message.txt* e *key.pem*.

A questo punto vengono trasmissi sia il messaggio che la chiave. Il messaggio cifrato può essere trasmesso tramite un canale non sicuro, mentre per la chiave abbiamo necessariamente bisogno di un mezzo sicuro. Vedremo come creare un canale protetto (tramite crittografia) nei capitoli a seguire, per ora immaginiamo che la chiave sia stata trasmessa in totale sicurezza.

Il destinatario riceve sia il messaggio che la chiave.

Viene separato il vettore iniziale dal messaggio cifrato:

```
nonce = message[:8] # Primi 8B
ciphertext = message[8:]
```

Viene inizializzato l'oggetto passando al costruttore la chiave ed il vettore iniziale come parametri:

```
cipher = Salsa20.new(key, nonce)
```

Viene infine decifrato il messaggio:

```
plaintext = cipher.decrypt(ciphertext)
```

È importante ricordarsi che il vettore iniziale deve essere diverso per ogni trasmissione per poter garantire la sicurezza dei dati.

5.2. Cifratura file a blocco

Abbiamo visto nel paragrafo precedente come cifrare (e decifrare) un semplice messaggio testuale. Vediamo invece ora come cifrare un file utilizzando il cifrario a blocco AES in modalità CBC.

Vengono per prima cosa importati i moduli, le classi e le funzioni che verranno utilizzate:

```
import os
import secrets
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
```

Vengono dichiarate le dimensioni dei blocchi e la lunghezza della chiave privata.

```
IV_LENGTH = AES.block_size # 16B
KEY_LENGTH = 16
RESERVED_BYTES = 64
```

- IV_LENGTH rappresenta la lunghezza del vettore iniziale che in AES è lungo esattamente quanto un blocco;
- KEY_LENGTH rappresenta la lunghezza della chiave privata utilizzata. Nell'esempio viene usato lo standard AES-128 che utilizza chiavi di 16B (128b);
- RESERVED_BYTES rappresenta la lunghezza dei metadati del file. In genere i file system mantengono le informazioni sui file (dimensione, tipo di compressione, data e ora, diritti d'autore ecc.) nei primi byte del file stesso. Il numero di byte utilizzati possono cambiare da un tipo di file ad un altro. Per il nostro esempio 64B sono più che sufficienti.

Vengono generati il vettore iniziale e la chiave.

```
iv = secrets.token_bytes(IV_LENGTH)
key = secrets.token_bytes(KEY_LENGTH)
```

A questo punto è possibile scindere il funzionamento del sistema in due parti: una parte per la cifratura del file e una parte per la decifratura del file.

La funzione di cifratura eseguirà i seguenti compiti:

```
def encrypt(file_in, file_out):
  # Inizializzazione classe crittografica
  cipher = AES.new(key, AES.MODE_CBC, iv)
  # Lettura file da cifrare
  with open(file_in, "rb") as file:
    byteblock = file.read()
  # Padding dei dati da cifrare
  ciphertext = pad(byteblock[RESERVED_BYTES:], AES.block_size)
  # Cifratura dati
  ciphertext = byteblock[:RESERVED_BYTES] + cipher.encrypt(ciphertext)
  # Scrittura file con dati cifrati
  with open(file_out, "wb") as file:
    file.write(ciphertext)
```

La funzione di decifratura eseguirà invece i seguenti compiti:

```
def decrypt(file_in, file_out):
  # Inizializzazione classe crittografica
  cipher = AES.new(key, AES.MODE_CBC)
  # Lettura file da decifrare
  with open(file_in, "rb") as file:
    byteblock = file.read()
  # Decifratura dati
  plaintext = cipher.decrypt(byteblock[RESERVED_BYTES:])
  # Unpadding dei dati da decifrare
  plaintext = unpad(plaintext, AES.block_size)
  # Concatenazione byte riservati con dati decifrati
  plaintext = byteblock[:RESERVED_BYTES] + plaintext
  # Scrittura dati decifrati
  with open(file_out, "wb") as file:
    byteblock = file.write(plaintext)
```

Entrambe le funzioni prendono come parametro il percorso del file da elaborare e il percorso dove verrà salvato il file ottenuto dopo il processo di cifratura/decifratura.

Nella funzione di cifratura è stato eseguito il padding sui dati da cifrare per far combaciare la dimensione del blocco letto con la dimensione del blocco richiesta dalla modalità CBC, ossia un multiplo di 16B. I metadati vengono posti in chiaro in testa ai dati cifrati.

Nell'esempio viene utilizzata un'immagine, già presente nella cartella d'esempio, della quale vengono generati due file: *encripted_image.jpg* e *decripted_image.jpg*.

6. Chiave pubblica

7. Firma

8. Protocolli

9. Conclusioni