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

Python ...

1.1. Fondamenti

Codifica e decodifica stringhe

Nel corso dei capitoli andremo a cifrare e a decifrare diversi tipi di dati, tra i quali le stringhe.

Per le stringhe verrà utilizzata la codifica UTF-8 in quanto standard di Python e compatibile con qualsiasi elaboratore moderno.

```
message = 'Ciao mondo!'
encoded_message = messagge.encode('UTF-8') # b'Ciao mondo!'
decoded_message = encoded_message.decode('UTF-8') # 'Ciao mondo!'
```

1.2. Librerie e moduli

PyCryptoDome

hashlib

salsa20

secrets

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

Il modulo permette l'accesso alla sorgente casuale ritenuta più sicura presente nel sistema operativo in uso.

os

2. Hashing

In questo capitolo andremo a vedere come utlizzare le funzioni di hashing per degli utilizzi comuni di tali algoritmi, come: nascondere e salvare una password in modo sicuro ed effettuare il controllo dell'integrita di un'informazione.

La libreria *PyCryptoDome* ci offre una serie di algoritmi, tra cui: MD5, BLAKE e la famiglia degli algoritmi SHA che possono essere utilizzati per l'adempimento di tali compiti.

2.1. Cifratura password

Sappiamo che l'hashing ci permette di ottenere un valore diverso per ogni input (diverso) che gli viene dato.

Questo torna molto utile quando vogliamo salvare delle password in un contenitore non sicuro e potenzialmente accessibile a terzi che potrebbero utilizzarla per accedere a dei dati sensibili.

Vedremo come implementare un algoritmo di hashing per nascondere una password mantenendo possibile l'autenticazione da parte dell'utente che la conosce.

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.

Per prima cosa viene importata la classe dell'algoritmo utilizzato:

```
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:

```
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 la password cifrata.

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 la password inserita corrisponda o meno con quella precedentemente salvata, verrà stampato su terminale il risultato dell'operazione:

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

2.2. Controllo integrità

Uno dei principali utilizzi degli algoritmi di hashing è quello di controllare che l'integrità di un'informazione non venga compromessa durante una trasmissione.

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

Nel nostro esempio utilizzeremo entrambi gli algoritmi sopra citati per calcolare il valore checksum di un file in formato mp4 e successivamente per assicurarci che il file sia integro e che non sia stato compromesso.

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

Come prima cosa vengono importate le classi:

```
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 che 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*.

Il miglioramente avviene in quanto la libreria *PyCryptoDome* si affida a delle funzioni scritte in linguaggio C, mentre la libreria *hashlib* utilizza delle funzioni più a basso livello di astrazione (assembly).

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()
```

3. Cifratura simmetrica

Dopo aver visto come effettuare la digestione di un dato tramite gli algoritmi di hashing, andremo a vedere come implementare degli algoritmi di cifratura simmetrica. Vedremo i più utilizzati passando dalla cifratura a flusso alla cifratura a blocco.

La libreria *PyCryptoDome* ci offre diversi algoritmi, quali: ChaCha20,XChaCha20, Salsa20 e AES. Per l'algoritmo AES ci offre una serie di modalità di cifratura a blocco che è possibile utilizzare, tra cui: ECB, CBC, CFB, OFB, CTR, OPENPGP, CCM, EAX, SIV, GCM, OCB.

Vedremo ora come implementare ed utilizzare alcuni di questi per delle semplici applicazioni.

3.1. One-Time Pad

Una funzione *One-Time Pad* consiste nel cifrare un messaggio, bit per bit, utilizzando una chiave generata casualmente che, successivamente all'utilizzo, verrà deprecata.

Per prima cosa andiamo ad importare i moduli che ci servono:

```
import binascii
import secrets
import string
```

Andiamo poi a definire le funzioni che andremo ad utilizzare.

La cifratura avverrà bit per bit tramite uno XOR quindi abbiamo bisogno di convertire il messaggio dichiarato come stringa in un tipo sul quale è permessa l'operazione logica. In Python è possibile effettuare lo XOR sia su valori binari che su valori interi.

```
# Conversione di un messaggio da stringa a intero
def toInt(message):
  msg = message.encode()      # Conversione in binario
  msg = binascii.hexlify(msg) # Conversione in esadecimale
  msg = int(msg, 16)          # Conversione in intero
  return msg

# Conversione di un messaggio da intero a stringa
def toStr(message):
  msg = format(message, 'x')          # Conversione in esadecimale
  msg = ('0' * (len(msg) % 2)) + msg # Padding del messaggio
  msg = binascii.unhexlify(msg)       # Conversione in binario
  msg = msg.decode()                  # Conversione in stringa
  return msg
```

Per l'ottenimento di una chiave, andiamo a definire una funzione che restituisce un token casuale ad ogni chiamata. La chiave sarà di tipo stringa e avrà come lunghezza il valore numerico passato come parametro (uguale alla lunghezza del messaggio da cifrare).

```
# Generazione chiave casuale
def genKey(length):
  key = ''
  for i in range(length):
    key += secrets.choice(string.ascii_letters)
  return key
```

A questo punto possiamo andare a cifrare e/o decifrare un dato messaggio nel seguente modo:

```
message = 'Ciao mondo!'
key = genKey(len(message))
encrypted_message = toInt(message) ^ toInt(key)
decrypted_message = toStr(encrypted_message ^ toInt(key))
```

Il carattere ^ permette di eseguire lo XOR logico tra i due valori dati.

3.2. Cifratura e decifratura messaggio testuale con cifrario 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 *XChaCha29*.

Per il nostro esempio andremo ad utilizzare l'algoritmo *Salsa20* in quanto algoritmo maggiormente conosciuto ed utilizzato.

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

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

Per prima cosa definiamo il messaggio da cifrare e generiamo una chiave casuale:

```
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:

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

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

A questo punto il messaggio può essere trasmesso al destinatario utilizzando un mezzo non sicuro, mentre la chiave tramite un mezzo sicuro. Vedremo nello specifico come fare in un capitolo successivo. Per ora facciamo finta di avere già un canale sicuro di trasmissione della password.

Quindi, una volta ricevuto sia il messaggio che la chiave da parte del mittente, andiamo a decifrare il messaggio.

Viene prelevato e rimosso il vettore iniziale dal messaggio:

```
nonce = message[:8]
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)
```

3.3. Cifratura e decifratura file con cifrario a blocco

Abbiamo visto nel paragrafo precedente come viene cifrato e decifrato un messaggio testuale. Andremo ora a vedere come cifrare e decifrare un file utilizzando l'algoritmo 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
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 presenti su 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. I 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)
```

Nella funzione di cifratura è stato eseguito il padding sui dati da cifrare per far combaciare la dimensione del blocco 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.

La funzione di decifratura eseguirà invece questi 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)
```

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