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

Librerie

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!'
```

Valori pseudocasuali

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.

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 ai nostri account.

Vedremo come implementare un algoritmo di hashing per nascondere una password mantenendo possibile l'autenticazione 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 importiamo la classe dell'algoritmo che vogliamo utilizzare:

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

Andiamo poi ad inizializzare l'oggetto che si occuperà di effettuare la digestione:

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

Chiediamo poi all'utente di inserire una password che verrà salvata su file:

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

Eseguiamo infine la digestione e salviamo il risultato di quest'ultima in un 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 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())
password = hashing.hexdigest()
if 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 importiamo le classi:

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

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

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

Dichiariamo gli oggetti che utilizzeremo per effettuare l'hashing:

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

Leggiamo tutti i blocchi dal file e li andiamo ad incorporare ai rispettivi input dei due algoritmi:

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

Effettuiamo 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)

Una cosa che notiamo quando eseguiamo lo script per file con una grande dimensione è il tempo impiegato per l'esecuzione.

Questo può essere migliorato e reso inferiore andando a sostituire le funzioni della libreria *PyCryptoDome* con le funzioni della libreria *hashlib*.

Questo 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 la libreria:

```
import hashlib
```

Sostituiamo le dichiariazioni delle classi:

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