



UNIVERSITA' DEGLI STUDI DI CAGLIARI

FACOLTÀ DI SCIENZE

Corso di Laurea in Informatica

PyCryptoDome

Implementazione e casi d'uso della libreria crittografica

Docente di riferimento

Prof. Massimo Bartoletti

Candidato

Michele Melis (matr.65798)

ANNO ACCADEMICO 2022-2023

Indice

1. Introduzione	1
1.1. Linguaggio Python	1
1.2. Librerie e moduli	2
1.3. Campioni e strumenti per i test	3
2. Algoritmi di hashing	4
2.1. Garantire la sicurezza di una password	4
2.2. Controllo integrità dati	5
Confronto con il modulo <i>hashlib</i>	6
2.3. Sistema di autenticazione di un messaggio	7
Confronto con il modulo <i>hmac</i>	8
3. Generazione valori casuali	9
4. Formati di importazione ed esportazione chiavi	10
5. Schemi di cifratura a chiave privata	12
5.1. Cifratura di un messaggio testuale tramite cifrario a flusso	12
Confronto con il modulo <i>salsa20</i>	13
5.2. Cifratura di un file tramite cifrario a blocco	14
Confronto con la libreria <i>Cryptography</i>	15
6. Schemi di cifratura a chiave pubblica	16
6.1. Condivisione chiave di sessione	16
Confronto con la libreria <i>Cryptography</i>	18
7. Meccanismi di firma digitale	19
7.1. Firma di un documento digitale	19
7.2. Sistema di autorizzazione tramite certificato ECDSA	20
8. Protocolli di condivisione e funzioni di derivazione chiavi	23
8.1. Garantire sicurezza a dati sensibili	23
8.2. Accesso privato ad un messaggio condiviso	25
8.3. Concordezza chiave di sessione tramite ECDH	26
9. Conclusioni	28

1. Introduzione

La crittografia è ormai un argomento presente in qualsiasi ambiente informatico. Abbiamo a che fare con sistemi crittografici quando mandiamo un messaggio, quando scarichiamo un file da internet, quando effettuiamo l'accesso ad un social network o quando effettuiamo degli acquisti tramite una carta di debito. Insomma, la crittografia è alla base di numerosi sistemi di uso quotidiano.

Al giorno d'oggi è possibile trovare centinaia di strumenti utili alla costruzione di tali sistemi. L'obiettivo dell'elaborato è quello di spiegare e illustrare come utilizzare uno di questi strumenti, ossia la libreria PyCryptoDome.

PyCryptoDome permette di creare delle applicazioni crittografiche in modo semplice, garantendone sicurezza ed efficacia. La libreria è stata sviluppata con il linguaggio di programmazione Python, il che la rende versatile e flessibile.

L'elaborato consiste nello sviluppo di sistemi informatici tramite l'utilizzo di PyCryptoDome, trattando un package diverso della libreria in ciascun capitolo. Ogni capitolo introduce quindi al package associato, descrivendo gli obiettivi delle primitive crittografiche implementate in esso e descrivendo nel dettaglio, in ciascuna sezione, l'utilizzo di tali primitive in relazione a dei comuni casi d'uso. Per alcune primitive, viene anche effettuato un test comparativo per valutare l'efficienza dell'implementazione della libreria rispetto ad altri moduli esistenti che ne implementano la medesima primitiva.

A fine lettura, si avrà una visione completa su tutte le funzionalità offerte dalla libreria e ci si potrà avvalere di script esemplificativi chiari, stilati seguendo le *good practices* suggerite dalla stessa libreria.

Ambiente di sviluppo e strumenti

Prima di cimentarci nell'implementazione della libreria, è necessario definire il proprio ambiente di sviluppo e gli strumenti che verranno utilizzati.

1.1. Linguaggio 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 all'implementazione. È uno dei linguaggi di programmazione più utilizzati nel mondo e la sua fama è ancora in crescita.

La versione a cui fanno riferimento gli script d'esempio è la 3.9.0.

Codifica e decodifica 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 la decodifica. I metodi prendono come parametro il nome della formato di codifica (per *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
```

```
import Crypto # Importazione 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).

```
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
```

salsa20

Il modulo *salsa20* implementa gli algoritmi di cifratura simmetrica *Salsa20* e *XSalsa20*.

```
pip install salsa20 # Installazione libreria
```

```
import salsa20 # Importazione modulo
```

Cryptography

La libreria *Cryptography* implementa delle interfacce, a basso livello e ad alto livello, degli algoritmi più comuni utilizzati per la crittografia, l'hashing e la derivazione di chiavi.

```
pip install cryptography # Installazione libreria
```

```
import cryptography # Importazione libreria
```

1.3. Campioni e strumenti per i test

I campioni utilizzati per comparare l'efficienza di un algoritmo tra due librerie (o moduli) sono stati ottenuti utilizzando un processore *AMD Ryzen 7 3800x*.

I test sono stati effettuati tramite la funzione di analisi dati di *Excel*.

2. Algoritmi di 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 molto difficile 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. Garantire la 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 per la generazione del valore hash. 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 al valore hash della precedente password, verrà stampato su terminale il risultato dell'operazione:

```
password = input("Reinserire 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.

Confronto con il modulo *hashlib*

Effettuiamo un confronto tra gli algoritmi MD5 e SHA256 dei due moduli *Crypto.Hash* e *hashlib*, prendendo in considerazione i tempi di esecuzione.

Analizziamo gli algoritmi utilizzando input di 100MB e valutando i risultati ottenuti tramite test t con significatività 1%.

MD5	PyCryptoDome	hashlib
Media	0,152104955	0,117522817
Varianza	4,88481E-06	8,03577E-06
Osservazioni	100	100
Varianza complessiva	6,46029E-06	
Differenza ipotizzata per le medie	0	
gdl	198	
Stat t	96,20792742	
P(T<=t) una coda	1,7404E-168	
t critico una coda	2,345328349	
P(T<=t) due code	3,4808E-168	
t critico due code	2,600887278	

SHA256	PyCryptoDome	hashlib
Media	0,325290117	0,047640123
Varianza	5,90451E-06	2,32721E-07
Osservazioni	100	100
Varianza complessiva	3,06862E-06	
Differenza ipotizzata per le medie	0	
gdl	198	
Stat t	1120,756572	
P(T<=t) una coda	0	
t critico una coda	2,345328349	
P(T<=t) due code	0	
t critico due code	2,600887278	

I risultati dei test suggeriscono una differenza significativa nei tempi di esecuzione e un tempo medio migliore per gli algoritmi implementati nel modulo *hashlib* (qualche millesimo di secondo per l'algoritmo MD5 e qualche decimo di secondo per l'algoritmo SHA256).

Possiamo quindi migliorare l'efficienza della nostra applicazione andando a sostituire le implementazioni del modulo *Crypto.Hash* con le implementazioni del modulo *hashlib*.

2.3. Sistema di autenticazione di un messaggio

Quando un messaggio viene trasmesso, questo potrebbe essere intercettato prima dell'arrivo al destinatario ed essere sostituito con un messaggio contraffatto. Un possibile metodo per far sì 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 sì 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 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.

Confronto con il modulo *hmac*

Effettuiamo ora un confronto tra gli algoritmi HMAC dei due moduli *Crypto.Hash* e *hmac* con input da 100MB.

Analizziamo i campioni utilizzando un test t con significatività 1%.

HMAC	PyCryptoDome	hashlib
Media	0,328573487	0,047934706
Varianza	4,8147E-05	3,96394E-06
Osservazioni	100	100
Varianza complessiva	2,60555E-05	
Differenza ipotizzata per le medie	0	
gdl	198	
Stat t	388,7613229	
$P(T \leq t)$ una coda	1,0201E-287	
t critico una coda	2,345328349	
$P(T \leq t)$ due code	2,0401E-287	
t critico due code	2,600887278	

Anche in questo caso abbiamo una differenza significativa tra le medie dei due moduli, suggerendo una miglior efficienza dell'algoritmo HMAC implementato dal modulo *hmac*.

3. Generazione valori casuali

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.

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 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](#).

Vediamo un esempio di utilizzo degli algoritmi per la generazione casuale di una stringa di byte:

```
from Crypto import Random
import secrets, os
LENGTH = 16
token = os.urandom(LENGTH)
print('os: ', token)
token = secrets.token_bytes(LENGTH)
print('secrets: ', token)
token = Random.get_random_bytes(LENGTH)
print('PyCryptoDome: ', token)
```

Risultati:

```
>> os:          b'sM\xca\x9dh{\xa6\x1f\xb1\xee\xfc\xd5\xed7&` '
>> secrets:      b'\x9e\x7f\xa9\xff0a\xee#\xad\xv\x3\xa1\x1b\xb0pq'
>> PyCryptoDome: b'\x91n\xbe1\xf9\xd7k\x8b\x13\x19\x8b\xa7~8!L'
```

4. Formati di importazione ed esportazione chiavi

Per la memorizzazione e la trasmissione di una chiave vengono utilizzati dei formati specifici. I formati più utilizzati sono: PEM e PKCS#8, entrambi implementati nel 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\x9300\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. Schemi di cifratura a chiave privata

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, AES e PKCS#1-OAEP.

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

5.1. Cifratura di un messaggio testuale tramite 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 *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:

```
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 trasmessi 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 supponiamo 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.

Confronto con il modulo *salsa20*

Effettuiamo un test t con significatività 1% sui tempi medi di cifratura di un dato di 100MB, confrontando l'algoritmo *Salsa20* dei moduli *Crypto.Cipher* e *salsa20*.

Salsa20	PyCryptoDome	salsa20
Media	0,429594457	0,242384276
Varianza	7,64953E-05	4,14814E-05
Osservazioni	100	100
Varianza complessiva	5,89883E-05	
Differenza ipotizzata per le medie	0	
gdl	198	
Stat t	172,3579766	
P(T<=t) una coda	5,3001E-218	
t critico una coda	2,345328349	
P(T<=t) due code	1,06E-217	
t critico due code	2,600887278	

Dai risultati del test possiamo osservare una differenza significativa tra le medie (circa 2 decimi di secondo).

Il modulo *salsa20* implementa un algoritmo più efficiente del modulo *Crypto.Cipher*.

5.2. Cifratura di un file tramite cifrario a blocco

Abbiamo visto nel paragrafo precedente come cifrare (e decifrare) un semplice messaggio testuale. Vediamo invece ora come cifrare un intero file utilizzando il cifrario a blocco AES in modalità CBC. Il procedimento è uguale a quello della sezione precedente, con la differenza che in questo caso viene letto il contenuto presente su un file e il dato viene cifrato un blocco alla volta.

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 può 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 cifrario
    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
    plaintext = pad(byteblock[RESERVED_BYTES:], AES.block_size)
    # Cifratura dati
    ciphertext = byteblock[:RESERVED_BYTES] + cipher.encrypt(plaintext)
    # 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 cifrario
    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: *encrypted_image.jpg* e *decrypted_image.jpg*.

Confronto con la libreria *Cryptography*

Effettuiamo ora un confronto tra i tempi di esecuzione dell'algoritmo AES in modalità CBC tra le implementazioni dei moduli offerti dalle librerie *PyCryptoDome* e *Cryptography*. Nello specifico effettuiamo un test t con significatività 1% e con input di 100MB.

AES (CBC)	PyCryptoDome	Cryptography
Media	0,163755975	0,138150549
Varianza	8,06165E-05	0,000139885
Osservazioni	100	100
Differenza ipotizzata per le medie	0	
gdl	185	
Stat t	17,24352319	
P(T<=t) una coda	1,18478E-40	
t critico una coda	2,34667322	
P(T<=t) due code	2,36955E-40	
t critico due code	2,602665303	

La differenza tra le medie dei tempi di esecuzione risulta essere significativa con una differenza di circa tre centesimi a favore del modulo di *Cryptography*.

6. Schemi di cifratura a chiave pubblica

I sistemi crittografici con chiave pubblica offrono confidenzialità nello scambio di informazioni. Il vantaggio di usare un sistema con chiave pubblica è quello di non dover trasmettere o rivelare la propria chiave privata, diminuendo le possibilità che un terzo possa avere accesso alla chiave durante una trasmissione. In genere i sistemi con chiave pubblica vengono considerati più sicuri di quelli con chiave privata. Questo vantaggio richiede però dei lunghi tempi di elaborazione, in quanto gli algoritmi di generazione di una chiave pubblica sono più lenti da eseguire rispetto a quelli con chiave privata. Per questo motivo, per lo scambio di messaggi, viene preferito un metodo misto che preveda lo scambio di una chiave privata (chiave di sessione) tramite un sistema a chiave pubblica.

La libreria *PyCryptoDome* ci offre quattro algoritmi per la generazione di una chiave pubblica, tra cui: RSA, DSA, ECC e ElGamal (quest'ultimo ritenuto obsoleto).

Nel corso del capitolo impareremo a trasmettere una chiave di sessione per lo scambio di messaggi, utilizzando la chiave pubblica messa a disposizione dal destinatario.

6.1. Condivisione chiave di sessione

Una chiave di sessione è una chiave privata utilizzata sia dal mittente che dal destinatario per cifrare/decifrare i messaggi scambiati durante una sessione. Le chiavi hanno una durata temporale limitata alla durata della sessione stessa e stabilita in base al caso d'uso; una volta terminata la sessione, la chiave viene invalidata e deprecata.

Vedremo in questa sezione come generare e condividere con il destinatario una chiave di sessione (e un messaggio). Verranno generate le chiavi (pubblica e privata) del destinatario utilizzando l'algoritmo RSA. Successivamente verrà utilizzato l'algoritmo AES per cifrare un messaggio testuale e l'algoritmo PKCS1-OAEP per cifrare la chiave di sessione.

Vengono per prima cosa importate le classi e le funzioni che verranno utilizzate:

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP
```

Vengono generate le chiavi del destinatario:

```
key = RSA.generate(2048) # Lunghezza chiave privata (256B)
private_key = key.export_key()
public_key = key.publickey().export_key()
```

Il metodo *export_key* prende in input: il tipo di formato per la codifica, una chiave di cifratura, il tipo di struttura, lo schema di protezione e una funzione di generazione di una chiave casuale. Di default, senza specificare parametri, viene codificata la chiave in formato PEM.

Le chiavi vengono memorizzate dal destinatario; la chiave pubblica viene trasmessa al mittente.

Ora che il mittente conosce la chiave pubblica del destinatario, può essere generata e trasmessa una chiave di sessione. Nel payload può essere aggiunto anche un messaggio testuale.

```
# Inizializzazione oggetto per la cifratura
rsa = PKCS1_OAEP.new(public_key)
# Generazione chiave di sessione
session_key = get_random_bytes(AES.block_size)
# Cifratura chiave di sessione tramite chiave pubblica
encrypted_session_key = rsa.encrypt(session_key)
# Cifratura messaggio tramite chiave di sessione
aes = AES.new(session_key, AES.MODE_EAX)
ciphertext, tag = aes.encrypt_and_digest(message)
# Scrittura informazioni su file
with open(ENCRYPTEDMESSAGE, "wb") as file_out:
    for x in (encrypted_session_key, aes.nonce, tag, ciphertext):
        file_out.write(x)
```

La chiave di sessione viene cifrata utilizzando la modalità EAX per garantire che il messaggio non venga compromesso. Nell'esempio vengono salvate tutte le informazioni (chiave di sessione cifrata, vettore iniziale, valore hash e messaggio cifrato) in un file che viene trasmesso al destinatario.

A questo punto il destinatario riceve il messaggio da decifrare:

```
# Lettura informazioni dal file
with open(ENCRYPTEDMESSAGE, "rb") as file_in:
    data = [ file_in.read(x) for x in (
        private_key.size_in_bytes(),
        AES.block_size, AES.block_size, -1
    )]
    encrypted_session_key, nonce, tag, ciphertext = data
# Inizializzazione oggetto per la decifratura
rsa = PKCS1_OAEP.new(private_key)
# Decifratura chiave di sessione con chiave privata
session_key = rsa.decrypt(encrypted_session_key)
# Decifratura messaggio testuale con chiave di sessione
aes = AES.new(session_key, AES.MODE_EAX, nonce)
message = aes.decrypt_and_verify(ciphertext, tag)
```

Sia il mittente che il destinatario conoscono ora la chiave di sessione: possono quindi continuare la conversazione utilizzando quest'ultima per cifrare i propri messaggi e deprecare la chiave pubblica del destinatario oramai non più utile.

Confronto con la libreria *Cryptography*

Effettuiamo un confronto sull'efficienza dell'algoritmo RSA, implementato dalle librerie *PyCryptoDome* e *Cryptography*, per la generazione di chiavi a 2048 bit.

Analizziamo i dati effettuando un test t con significatività 1%.

RSA	PyCryptoDome	Cryptography
Media	0,817756367	0,048573773
Varianza	0,318989591	0,000763744
Osservazioni	100	100
Varianza complessiva	0,159876667	
Differenza ipotizzata per le medie	0	
gdl	198	
Stat t	13,60259936	
P(T≤t) una coda	1,72766E-30	
t critico una coda	2,345328349	
P(T≤t) due code	3,45532E-30	
t critico due code	2,600887278	

I risultati ottenuti suggeriscono una differenza significativa tra le medie dei tempi di esecuzione dei due algoritmi per la generazione delle chiavi (circa 8 decimi di secondo di differenza), a favore dell'implementazione dell'algoritmo presente nella libreria *Cryptography*.

7. Meccanismi di firma digitale

Gli algoritmi di firma digitale garantiscono l'autenticità e l'integrità di un dato stabilendone la non ripudiabilità. Gli algoritmi si basano sulla crittografia asimmetrica, permettendo all'utente firmatario di firmare un dato con la propria chiave privata e successivamente validare tale dato tramite chiave pubblica.

La libreria *PyCryptoDome* ci offre una serie di meccanismi di firma, tra cui: PKCS#1 v1.5, PKCS#1 PSS, DSA, EdDSA ed ECDSA.

Nel corso del capitolo impareremo a firmare digitalmente un documento (file) e a realizzare un sistema di autorizzazione tramite certificato ECDSA.

7.1. Firma di un documento digitale

Con la crescente sostituzione dei documenti in forma cartacea con documenti in forma digitale, diviene essenziale l'utilizzo di uno strumento di firma digitale che possa garantire l'autenticità del soggetto firmatario.

In questa sezione vedremo come firmare un documento (file) tramite l'algoritmo di firma digitale PKCS#1 PSS andando poi a verificare la validità di tale firma.

Per prima cosa vengono importate le classi utilizzate:

```
from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
```

Vengono poi generate le chiavi (pubblica e privata) con l'algoritmo RSA. Abbiamo visto come generare la coppia di chiavi nel capitolo precedente.

A questo punto ci serve un documento da firmare. Nella cartella d'esempio è presente un file (pdf) che rappresenta un contratto di leasing che verrà utilizzato come esempio.

Vengono quindi svolte le seguenti operazioni sul file:

```
# Lettura documento da firmare
with open('document.pdf', 'rb') as file:
    document = file.read()
# Lettura chiave privata
with open('private.pem', 'rb') as file:
    key = RSA.import_key(file.read())
# Hashing del documento
h = SHA256.new(document)
# Firma dell'hash del documento
signature = pss.new(key).sign(h)
# Salvataggio firma su file
with open('signature.p7s', 'wb') as file:
    file.write(signature)
```

La firma digitale viene salvata su file e verrà successivamente utilizzata per verificare l'autenticità della firma e l'integrità del documento.

Ora che il documento è stato firmato e la firma è stata resa nota, è possibile procedere con la validazione di tale firma:

```
# Lettura chiave pubblica
with open(PUBLICKEY, 'rb') as file:
    key = RSA.import_key(file.read())
# Lettura documento da validare
with open(DOCUMENT, 'rb') as file:
    document = file.read()
# Hashing del documento
h = SHA256.new(document)
# Lettura firma dal file
with open(SIGNATURE, 'rb') as file:
    signature = file.read()
# Verifica firma
try:
    pss.new(key).verify(h, signature)
    print('La firma è autentica.')
except (ValueError, TypeError):
    print('La firma non è autentica.')
```

Il metodo utilizzato per la verifica (*verify*) lancia un'eccezione di tipo **ValueError** in caso la firma non sia valida per il dato documento, altrimenti procede con la normale esecuzione.

7.2. Sistema di autorizzazione tramite certificato ECDSA

Gli algoritmi di firma digitale possono essere utilizzati anche per l'implementazione di un sistema di autorizzazione. Per sistema di autorizzazione si intende un sistema che prevede la concessione di permessi speciali da parte di un utente ad alto livello ad un utente di basso livello, come ad esempio: autorizzazione modifiche di cartelle condivise, accesso protetto ad applicazioni e siti web (parental control) e permessi di esecuzione speciali.

Vedremo come implementare un sistema (generico) di autorizzazione tramite l'algoritmo di firma digitale ECDSA.

Per prima cosa vengono importate le classi e le funzioni utilizzate:

```
from Crypto.PublicKey import ECC
from Crypto.Signature import DSS
from Crypto.Hash import SHA256
from Crypto.Random import get_random_bytes
```

Il sistema prevede l'interazione tra due utenti: un utente amministratore ad alto livello (**SuperUser**) e un utente semplice a basso livello (**User**).

```
class SuperUser:
    def __init__(self, username, password):
        self.username = username
        key = ECC.generate(curve = 'P-256')
        self._private_key = key.export_key(
            format = 'PEM',
            use_pkcs8 = True,
            passphrase = password,
            protection = 'PBKDF2WithHMAC-SHA1AndAES128-CBC'
        )
        self.public_key = key.public_key()
```

```
class User:
    def __init__(self, username, superuser):
        self.username = username
        self.superuser = superuser
```

Il costruttore della classe **SuperUser** prende in ingresso due parametri: lo username dell'amministratore e una password. Viene poi generata una chiave privata, successivamente codificata in formato PEM, cifrata tramite la data password e salvata come attributo privato di classe; la chiave pubblica viene salvata in chiaro per poter essere reperita in seguito. In un sistema reale, si consiglia di memorizzare su supporto fisico/virtuale (hard disk, file, database ecc.) le chiavi in quanto gli oggetti potrebbero essere volatili e non più accessibili in un secondo momento.

Il costruttore della classe **User** prende invece in ingresso come parametri: uno username e un oggetto **SuperUser** a cui fare richiesta di concessione dei permessi speciali.

Viene definito poi un metodo che permetta ad un amministratore di concedere i permessi ad un utente semplice:

```
class SuperUser:
    def _grant_permissions_to_user(self, operation_token, password):
        try:
            key = ECC.import_key(self._private_key, passphrase = password)
            print('Password accettata.')
        except ValueError:
            print('Password rifiutata.')
            return ''
        # Firma del token di operazione
        hash = SHA256.new(operation_token)
        signer = DSS.new(key, 'fips-186-3')
        return signer.sign(hash)
```

Il metodo prende in ingresso un token che rappresenta l'operazione richiesta e la password dell'amministratore. Viene poi decifrata la chiave privata tramite password, calcolato il valore hash del token e creato un oggetto **DssSigScheme** passandogli la chiave privata decifrata e la modalità di firma digitale. Viene infine eseguita e restituita la firma dell'amministratore di tale token.

Nell'esempio viene anche definita una piccola interfaccia di autorizzazione:

```
class SuperUser:
    def permissions_request(self, user):
        print(f'Vuoi concedere i permessi a {user.username}?')
        print(f'Token: {user.operation_token}')
        if input().lower() == 'si':
            password = input('Inserire password amministratore: ')
            return self._grant_permissions_to_user(user.operation_token, password)
        else:
            return ''
```

A questo punto un utente semplice può fare richiesta di permessi speciali per svolgere una determinata operazione che li richiede:

```
class User:
    def do_something_with_permissions(self):
        # Generazione token operazione
        self.operation_token = get_random_bytes(16)
        # Richiesta permessi
        signature = self.superuser.permissions_request(self)
        # Inizializzazione oggetti per la verifica
        hash = SHA256.new(self.operation_token)
        verifier = DSS.new(self.superuser.public_key, 'fips-186-3')
        # Verifica permessi
        try:
            verifier.verify(hash, signature)
            print('Permessi concessi.')
        except ValueError:
            print('Permessi NON concessi.')
        # Ripristino token
        self.operation_token = None
```

In quest'ultimo metodo viene generato un token, chiesto all'amministratore di concedere i permessi all'utente richiedente ed eseguita la verifica della firma per validare il certificato. Se il certificato è valido, l'utente può godere di tali permessi.

8. Protocolli di condivisione e funzioni di derivazione chiavi

Il package *Crypto.Protocol* di *PyCryptoDome* presenta tre moduli distinti, ciascuno con un diverso utilizzo.

1. Key Derivation Functions

Le funzioni di derivazione chiavi vengono utilizzate per la derivazione di una o più chiavi segrete (secondarie) da una password o da una chiave segreta primaria (detta “master key”).

Queste funzioni permettono di isolare le chiavi secondarie tra loro, evitando di compromettere la sicurezza della master key in caso anche solo una delle chiavi secondarie venisse resa nota.

Tra le funzioni di derivazione chiavi offerte dal modulo *Crypto.Protocol.KDF* troviamo: PBKDF2, scrypt, bcrypt, HKDF, SP 800-180 (counter mode) e PBKDF1.

2. Secret Sharing Schemes

Il modulo *Crypto.Protocol.SecretSharing* di *PyCryptoDome* ci offre l’implementazione del protocollo “Shamir’s secret sharing”. Il protocollo genera N chiavi da una chiave segreta e permette la ricostruzione di quest’ultima se si è in possesso di almeno K chiavi ($K < N$).

L’implementazione del protocollo si basa sulle seguenti proprietà:

- il segreto deve essere una stringa di 16 byte
- ciascuna chiave generata ha una lunghezza di 16 byte
- gli identificativi delle chiavi iniziano dal numero 1

3. Diffie-Hellman Key Agreement

Il protocollo Diffie-Hellman è uno schema di concordanza tra due entità di una chiave privata tramite crittografia asimmetrica. Permette di elaborare una chiave segreta utilizzata dalle due parti senza necessità di trasmettere la chiave o parte di essa. Questa sua proprietà lo rende essenziale e sicuro per le comunicazioni effettuate tramite un canale non protetto.

8.1. Garantire sicurezza a dati sensibili

In un sistema che memorizza dati sensibili è opportuno garantire sicurezza e privacy a tali dati. Prendiamo ad esempio un sito web che memorizza le informazioni sui metodi di pagamento dei propri utenti e, al momento del pagamento di un bene o di un servizio, permette all’utente di non dover reinserire le informazioni necessarie all’acquisto recuperando i dati direttamente dal database.

Vedremo ora un esempio di come cifrare dei dati sensibili utilizzando una chiave secondaria ottenuta tramite funzione di derivazione. Nell’esempio viene cifrata una stringa contenente il codice di una carta di credito.

Verranno utilizzate: la funzione di derivazione chiavi *scrypt*, l’algoritmo di hashing *SHA256* e il cifrario a blocco *AES* in modalità *CBC*.

Per prima cosa vengono importate le classi e le funzioni utilizzate:

```
from Crypto.Protocol.KDF import scrypt
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Util.Padding import pad
```

Vengono poi definite delle macro:

```
TOKEN_LENGTH = 16
SCRYPT_PARAMS = (2**14, 8, 1)
```

- *TOKEN_LENGTH* rappresenta la lunghezza dei token generati dal sistema (sale per la funzione di derivazione della chiave e vettore iniziale per la cifratura dei dati);
- *SCRYPT_PARAMS* contiene i parametri di inizializzazione della funzione *scrypt*, ossia: il parametro di costo CPU/memoria, la dimensione del blocco e il parametro di parallelizzazione.

Vengono dichiarati la password e il dato da cifrare:

```
password = 'la mia password segreta'.encode()
plaintext = '3141-5926-5358-9793'.encode()
data = {}
```

Viene generato il valore hash della password per la memorizzazione e la validazione:

```
hashing = SHA256.new(password)
hash = hashing.hexdigest()
data["hash"] = hash
```

Viene generata la chiave secondaria utilizzata per la successiva cifratura dei dati:

```
salt = get_random_bytes(TOKEN_LENGTH)
secondary_key = scrypt(
    password,
    salt,
    TOKEN_LENGTH,
    N=SCRYPT_PARAMS[0],
    r=SCRYPT_PARAMS[1],
    p=SCRYPT_PARAMS[2]
)
```

Vengono infine cifrati i dati:

```
iv = get_random_bytes(TOKEN_LENGTH)
cipher = AES.new(secondary_key, AES.MODE_CBC, iv)
padded_plaintext = pad(plaintext, AES.block_size)
data["ciphertext"] = salt + iv + cipher.encrypt(padded_plaintext)
```

A questo punto la variabile *data* conterrà il valore hash della password e il testo cifrato con in testa il sale (utilizzato per la generazione della chiave derivata) e il vettore iniziale (utilizzato per la cifratura dei dati):

```
{
  'hash': 'c7baa6d7d6632770b49b287273e6300afbb15cd3d93ce10f40b6c86668d0762e',
  'ciphertext': b'\x1a&5<\xac?\xb2\xb6\x83\x912i\xc7\xf6\x8b\x9c\xfd:\n\xb2)
\n\x9bu\xbfv\xc9\xfa\xe8I\x01\x059~\xc7\xd2c\xb6*,3Z\x98\\\x19\xbf0\xe6\x940z#i]
\xb0\x94\xefN1\xb9\xf5='
}
```

8.2. Accesso privato ad un messaggio condiviso

Supponiamo di avere un messaggio cifrato che vogliamo possa essere decifrato solo se si entra in possesso di K chiavi ($1 \leq K$). Tramite il protocollo *Shamir's secret sharing* andremo a vedere come suddividere un segreto (chiave privata) in N parti e decifrare un messaggio tramite l'utilizzo di K chiavi ($K \leq N$). Nell'esempio andremo a studiare un sistema generico di cifratura.

Per prima cosa vengono importate le classi e le funzioni utilizzate:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Protocol.SecretSharing import Shamir
```

Successivamente vengono definite delle macro:

```
TOKEN_LENGTH = 16
NUMBER_OF_SHARES = 5
SHARES_FOR_RECONSTRUCTION = 2
```

- *TOKEN_LENGTH* rappresenta la dimensione delle chiavi generate;
- *NUMBER_OF_SHARES* rappresenta il numero di chiavi generate;
- *SHARES_FOR_RECONSTRUCTION* rappresenta il numero di chiavi sufficienti alla ricostruzione del segreto.

Vengono poi generate le chiavi:

```
key = get_random_bytes(TOKEN_LENGTH)
shares = Shamir.split(SHARES_FOR_RECONSTRUCTION, NUMBER_OF_SHARES, key)
```

Viene definito e cifrato un messaggio:

```
plaintext = "Ciao mondo!".encode()
cipher = AES.new(key, AES.MODE_EAX)
ct, tag = cipher.encrypt_and_digest(plaintext)
ciphertext = cipher.nonce + tag + ct
```

A questo punto abbiamo cifrato un messaggio tramite chiave segreta e abbiamo suddiviso quest'ultima in N chiavi distinte, ciascuna associata ad un identificativo.

Possiamo quindi procedere con la decifratura del messaggio.

Nell'esempio vengono scelte casualmente delle chiavi (tra quelle generate) in un numero (K) sufficiente alla ricostruzione della chiave privata:

```
reconstruction = []
for x in range(SHARES_FOR_RECONSTRUCTION):
    id = random.randint(0, len(shares)-1)
    reconstruction.append(shares[id])
    shares.pop(id)
key = Shamir.combine(reconstruction)
```

Infine viene decifrato il messaggio tramite chiave segreta, restituendo un messaggio di errore in caso di fallimento:

```
nonce = ciphertext[:16]
tag = ciphertext[16:32]
cipher = AES.new(key, AES.MODE_EAX, nonce)
try:
    result = cipher.decrypt(ciphertext[32:])
    cipher.verify(tag)
    print(f"Messaggio decifrato: {result.decode()}")
except ValueError:
    if len(reconstruction) < SHARES_FOR_RECONSTRUCTION:
        print("Il numero di chiavi inserite non è sufficiente.")
    else:
        print("Almeno una delle chiavi non è corretta.")
```

8.3. Concordanza chiave di sessione tramite ECDH

Nei capitoli precedenti abbiamo imparato a generare e condividere una chiave di sessione utilizzata per cifrare i messaggi di una conversazione tra due utenti. In questa sezione impareremo invece a concordare una chiave privata tra due entità tramite l'implementazione del protocollo ECDH (*Elliptic Curve Diffie-Hellman Key Exchange*), andando a costruire un sistema che segua le raccomandazioni presenti in *NIST SP 800-56A* secondo le quali bisogna legare alla chiave concordata delle informazioni relative al contesto in cui viene utilizzata (come ad esempio una descrizione del contesto, gli identificativi dei due utenti ecc.) e in modo tale da ottenere una *Perfect Forward Secrecy* tramite l'uso di chiavi effimere.

Nel nostro esempio vediamo come due utenti (Alice e Bob) concordano una chiave di sessione. L'esempio illustra i passi compiuti dall'elaboratore di Alice.

Per prima cosa vengono importate le classi e le funzioni utilizzate:

```
from Crypto.PublicKey import ECC
from Crypto.Hash import TupleHash128
from Crypto.Random import get_random_bytes
from Crypto.Protocol.DH import key_agreement
```

Vengono poi dichiarate le informazioni utilizzate per la generazione della chiave di sessione, ossia: il sale, gli identificativi delle due parti e le chiavi statiche ed effimere:

```
SESSION_SALT = get_random_bytes(16)
# Alice
alice_id = get_random_bytes(16)
alice_private_key = ECC.generate(curve='p256')
alice_ephemeral_private_key = ECC.generate(curve='p256')
# Bob
bob_id = get_random_bytes(16)
bob_public_key = ECC.generate(curve='p256').public_key()
bob_ephemeral_public_key = ECC.generate(curve='p256').public_key()
```


Viene poi definita una funzione per la derivazione della chiave condivisa, andando ad incapsulare tutte le informazioni sul contesto:

```
def kdf(x):  
    h = TupleHash128.new(digest_bytes=32)  
    h.update(  
        x,  
        SESSION_SALT,  
        alice_id,  
        bob_id,  
        b'Generic encryption',  
        b'TupleHash128',  
        b'AES256'  
    )  
    return h.digest()
```

La funzione restituisce una stringa di 32 byte (AES-256). Nota: la funzione di derivazione deve essere la stessa per entrambe le parti (viene quindi stabilita a priori).

Viene infine utilizzata la funzione per la generazione della chiave di sessione:

```
session_key = key_agreement(  
    static_priv=alice_private_key,  
    static_pub=bob_public_key,  
    eph_priv=alice_ephemeral_private_key,  
    eph_pub=bob_ephemeral_public_key,  
    kdf=kdf  
)
```

A questo punto sia Alice che Bob saranno in possesso della stessa chiave che potranno utilizzare per cifrare/decifrare i propri messaggi durante la conversazione.

9. Conclusioni

Abbiamo visto come implementare la libreria PyCryptoDome per la creazione di sistemi crittografici. Abbiamo anche visto quanto la libreria sia semplice ed efficace per lo svolgimento di una vasta gamma di operazioni.

Risultati dei test di comparazione

I test di comparazione effettuati non hanno avuto un esito positivo per la libreria, mostrando dei risultati a favore delle implementazioni dei moduli con il quale è avvenuta la comparazione. Questo è probabilmente da associare al fatto che la maggior parte delle primitive della libreria hanno delle implementazioni ad alto livello di astrazione, rendendo di fatto le operazioni più lunghe da eseguire, in quanto necessaria l'interpretazione.

Pregi e difetti

Tirando le somme, possiamo elencare i pregi e i difetti trovati.

Pregi:

- Primitive: la libreria implementa un gran numero di primitive, rendendola utile per lo svolgimento diversificato di un gran numero di operazioni crittografiche (e non);
- Sintassi: la libreria offre una sintassi semplice, favorendone la sua implementazione e la successiva manutenzione;
- QoL: la libreria viene mantenuta e aggiornata con regolarità, aggiungendo primitive e funzioni al passo con gli standard correnti;
- Retrocompatibilità: la libreria può essere utilizzata con standard deprecati (anche se non consigliato per motivi di sicurezza);

Difetti:

- Efficienza: la libreria ha dei tempi di esecuzione elevati per alcune primitive.

Sviluppi futuri della libreria

Come precedentemente accennato, la libreria viene aggiornata con regolarità, aggiungendo primitive e funzioni. Nella documentazione è possibile trovare una [pagina dedicata](#), dove vengono elencate le modifiche e le aggiunte pianificate per le future versioni della libreria.

Ringraziamenti

Si ringrazia Professor Massimo Bartoletti per il supporto e l'aiuto fornito durante la stesura dell'elaborato.