

Report S3/L4

Matteo Congiu

Esercizio di oggi: Crittografia.

Dato un messaggio cifrato cercare di trovare il testo in chiaro:

1) Messaggio cifrato: "HSNFRGH"

Svolgimento

Utilizzando il cifrario di Cesare, ogni lettera viene sostituita con la lettera di 3 posti dopo.

H=E;

S=P;

N=I;

F=C;

R=O;

G=D;

H=E.

HSNFRGH=EPICODE

2) Esercizio di oggi Criptazione e Firmatura con OpenSSL e Python:

Obiettivi dell'esercizio:

- Generare chiavi RSA.
- Estrarre la chiave pubblica da chiave privata.
- Criptare e decriptare messaggi.
- Firmare e verificare messaggi.

Strumenti utilizzati:

- OpenSSL per la generazione delle chiavi.
- Libreria cryptography in Python.

Svolgimento:

Per iniziare sono entrato nella modalità root e poi ho installato OpenSSL con i comandi:

-sudo apt update

-sudo apt install openssl

```
zsh: corrupt history file /home/kali/.zsh_history
(kali@kali)~$ sudo su
[sudo] password for kali:
(root@kali)~$ sudo apt update
Get:1 http://kali.download/kali kali-rolling InRelease [41.5 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 Packages [20.2 MB]
Get:3 http://kali.download/kali kali-rolling/main amd64 Contents (deb) [48.8 MB]
Get:4 http://kali.download/kali kali-rolling/contrib amd64 Packages [109 kB]
Get:5 http://kali.download/kali kali-rolling/non-free amd64 Packages [196 kB]
Fetched 69.4 MB in 21s (3,256 kB/s)
2030 packages can be upgraded. Run 'apt list --upgradable' to see them.
(root@kali)~$ sudo apt install openssl
Upgrading:
libssl3t64 openssl
Installing dependencies:
openssl-provider-legacy
Summary:
  Upgrading: 2, Installing: 1, Removing: 0, Not Upgrading: 2028
  Download size: 3,951 kB
  Space needed: 431 kB / 63.4 GB available
Continue? [Y/n] Y
Get:1 http://kali.download/kali kali-rolling/main amd64 openssl-provider-legacy amd64 3.3.2-2 [298 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 libssl3t64 amd64 3.3.2-2 [2,271 kB]
Get:3 http://kali.download/kali kali-rolling/main amd64 openssl amd64 3.3.2-2 [1,382 kB]
Fetched 3,951 kB in 1s (2,825 kB/s)
(Reading database ... 391276 files and directories currently installed.)
Preparing to unpack .../libssl3t64_3.3.2-2_amd64.deb ...
Unpacking libssl3t64:amd64 (3.3.2-2) over (3.2.1-3) ...
Selecting previously unselected package openssl-provider-legacy.
Preparing to unpack .../openssl-provider-legacy_3.3.2-2_amd64.deb ...
Unpacking openssl-provider-legacy (3.3.2-2) ...
Setting up libssl3t64:amd64 (3.3.2-2) ...
Setting up openssl-provider-legacy (3.3.2-2) ...
(Reading database ... 391280 files and directories currently installed.)
Preparing to unpack .../openssl_3.3.2-2_amd64.deb ...
Unpacking openssl (3.3.2-2) over (3.2.1-3) ...
Setting up openssl (3.3.2-2) ...
Installing new version of config file /etc/ssl/openssl.cnf.original ...
Processing triggers for libc-bin (2.38-10) ...
Processing triggers for man-db (2.13.0-1) ...
Processing triggers for kali-menu (2023.4.7) ...
```

```
-sudo apt installpython3-pip
```

-pip3 install cryptography → Con questo comando la libreria mi risultava già presente

Ho digitato per primo il comando per generare la chiave privata RSA e successivamente il comando per estrarre la chiave pubblica:

Con il comando `-touch` ho creato il file Python con nome `encdec.py` e ho iniziato a scriverlo:

Padding è un processo che aggiunge dati extra a un messaggio per far sì che abbia una lunghezza adeguata per essere crittografato.

Serialization per leggere le chiavi private e pubbliche

base64 per visualizzare il criptato in base64

Con `open` leggiamo le chiavi e le trasferiamo nelle variabili: `private_key`; `public_key`.

-with open('private_key.pem', 'rb') as key_file: **Apri il file private_key.pem in modalità lettura binaria 'rb'.**

-private_key= serialization.load_pem_private_key(**Carica la chiave privata dal file pem.**

-key_file.read(),

-password=None)

-With open('public_key.pem', 'rb') as key_file: **Apri il file public_key.pem in modalità lettura binaria (rb).**

-public_key= serialization.load_pem_public_key(key_file.read()) **Carica la chiave pubblica dal file pem.**

-message ='Ciao, Epicode spacca!' **Messaggio che si vuole criptare.**

-encrypted= public_key.encrypt(message.encode(), padding.PKCS1v15())

 **Utilizza la chiave pubblica per criptare il messaggio.**

-decrypted= private_key.decrypt(encrypted, padding.PKCS1v15())

 **Utilizza la chiave privata per decriptare il messaggio.**

-print("Messaggio originale:", message) **Stampa il messaggio originale**

-print("Messaggio criptato:", base64.b64encode(encrypted).decode('utf-8')) **Stampa il messaggio criptato in base64**

-print("Messaggio decriptato:", decrypted.decode('utf-8')) **Stampa il messaggio decriptato**

```
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization
import base64

# Carica la chiave privata
with open('private_key.pem', 'rb') as key_file:
    private_key = serialization.load_pem_private_key(
        key_file.read(),
        password=None)

# Carica la chiave pubblica
with open('public_key.pem', 'rb') as key_file:
    public_key = serialization.load_pem_public_key(key_file.read())

message = 'Ciao, Epicode spacca!'

# Criptazione con la chiave pubblica
encrypted = public_key.encrypt(message.encode(), padding.PKCS1v15())

# Decriptazione con la chiave privata
decrypted = private_key.decrypt( encrypted , padding.PKCS1v15())

print ("Messaggio originale:", message)
print ("Messaggio criptato:", base64.b64encode(encrypted).decode('utf-8'))
print ("Messaggio decriptato:", decrypted.decode('utf-8'))
```

Ho avviato il programma con il comando: **-python endec.py**

```
(kali@kali)~$ python endec.py
Messaggio originale: Ciao, Epicode spacca!
Messaggio criptato: CVB50VVe690TFpJ1DCBx4gaS1aIQc0r/RL0qv2/AwnStxp1s+ZLSsB2NyATD6Vf79GvMtaLVHu+GGuJ13Wcu+x4hy8hW5cJo4SK0lgCJXwLRVgVtgnV4bYYvze8NnqtB40gKGA5tQtNV5MgmHZBkKKBmtrrhUU7jL4jshKkxqQT6TJ41XD6Gd1dt2NdJP2tL6S7UX0/J/tp6C2mDvqZ
Eryj3BSvCNK7HAzcl40Be8UeZFLhyZe84j/7EYQqZ8niXh6ESGKi3Er52LFm8/ngIqBx68ZQ6hFjcjCgwzuVvhn9m+top+p77fRr6go3ZTtxtuvN43ZzHt3p+Z+g=
Messaggio decrittato: Ciao, Epicode spacca!
```

Successivamente ho creato il file **firma.py** attraverso il comando touch e l'ho aperto con nano, ho scritto il programma come quello precedente apportando alcune modifiche che evidenzierò:

-from cryptography.hazmat.primitives.asymmetric import padding

-from cryptography.hazmat.primitives import hashes

Serve a creare l'hash del messaggio

-from cryptography.hazmat.primitives import serialization

-import base64

-with open('private_key.pem', 'rb') as key_file:

-private_key= serialization.load_pem_private_key(

-key_file.read(),

-password=None)

-With open('public_key.pem', 'rb') as key_file:

-public_key= serialization.load_pem_public_key(key_file.read())

-message ='Ciao, Epicode spacca!'

-signed = private_key.sign(message.encode(), padding.PKCS1v15(), hashes.SHA256())

 **Viene utilizzata la chiave privata per firmare il messaggio e viene usato l'hash SHA256**

-try:

-encrypted_b64 = base64.b64encode(signed).decode('utf-8')

-public_key.verify(signed, message.encode(), padding.PKCS1v15(), hashes.SHA256())

 **Utilizza la chiave pubblica per verificare la firma digitale**

- print("Base64 della firma:", encrypted_b64)

Stampa la firma in formato base64

-print("Messaggio originale da confrontare:", message)

Stampa il messaggio originale

-print("La firma è valida.")

Stampa la convalida della firma

-except Exception as e:

-print("La firma non è valida.", str(e))

Se ci dovessero essere difformità viene stampata la scritta: non è valida

```

from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
import base64
# Carica la chiave privata

with open('private_key.pem', 'rb') as key_file:
    private_key = serialization.load_pem_private_key(
        key_file.read(),
        password=None)

# Carica la chiave pubblica

with open('public_key.pem', 'rb') as key_file:
    public_key = serialization.load_pem_public_key(key_file.read())

message = 'Ciao, Epicode spacca!'

# Firma con la chiave privata
signed = private_key.sign( message.encode(), padding.PKCS1v15(), hashes.SHA256())

# Verifica della firma con la chiave pubblica

try:
    encrypted_b64 = base64.b64encode(signed).decode('utf-8')
    public_key.verify(signed, message.encode(), padding.PKCS1v15(), hashes.SHA256())

    print ("Base64 della firma:", encrypted_b64)
    print ("Messaggio originale da confrontare:", message)
    print ("La firma è valida.")
except Exception as e:
    print ("La firma non è valida.", str(e))

```

Ho avviato il programma con `python firma.py`

```

(kali@kali)~$ python firma.py
Base64 della firma: ferc2QnI2vYA3IDPulV1Ar6REbBwRuRWR1a0YsMbuFBh+KXpmKXmVGVDEJlpZhuXG0WIIPCP0vLAsC4kwM9J3/yB1mgKRmPLGnCd/XmS3RmkBC7vPf+dvR9ZVsmL4F+ftzd622YUDUy/9UzRBNlrL93SGI2UxTYAqB1w2QVEZsgJ4ZJGvZPlpe4KDzV4lyFhH9QZ1LRVYL828e
Messaggio originale da confrontare: Ciao, Epicode spacca!
La firma è valida.
(kali@kali)~$

```

