Report S3/L4:

Esercizio Crittografia+ Esercizio Bonus

Data la traccia:

“Esercizio di oggi: Crittografia.

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

Messaggio cifrato: " HSNFRGH"

Secondo esercizio (esercizio bonus)

Messaggio

cifrato: “QWJhIHZ6b2VidHl2bmdyIHB1ciB6ciBhciBucHBiZXRi”

Buon divertimento”

Sviluppo esercizio Crittografia

Essendo che si tratta di un messaggio cifrato di cui non abbiamo nessuna chiave o metodo di cifratura, ho provato a decifrarlo usando il cifrario di cesare.

Dopo aver effettuato diversi tentativi, ho concluso che decifrandolo in chiave 3 (quindi sostituendo le lettere date con quella che viene prima di 3 posizioni in ordine alfabetico) possiamo ottenere la seguente parola decifrata: “EPICODE”.

Sviluppo esercizio bonus

Per l’esercizio bonus ho adottato la stessa metodologia, ovvero, essendo una stringa priva di una chiave per decifrarla o qualsiasi altro metodo, ho provato nuovamente con il “Cifrario di Cesare”.

Ho fatto diversi tentativi come per l’esercizio precedente, questa volta la chiave di cifratura è la 13 (anche conosciuta come Rot13).

Ma prima di arrivare alla parte del Cifrario, ho constatato che si tratta di una stringa codificata a base 64.

Ho quindi cercato metodi per decodificare questa stringa, e ho trovato quello che ritengo il metodo più esplicativo:

La codifica Base64 utilizza una tecnica di conversione dei dati basata su 64 simboli del formato ASCII.

Questa codifica è utilizzata principalmente nella posta elettronica al fine di convertire i dati binari nello standard ASCII, ma è molto utilizzata anche sul web ad esempio per il passaggio di dati attraverso il metodo GET.

I dati criptati con l'algoritmo Base64 non possono e non devono essere considerati sicuri in quanto un dato criptato con questo algoritmo può essere facilmente riconvertito (come dimostra il tool presente in questa pagina). Come funziona l'algoritmo Base 64 L'algoritmo di codifica Base64 ha un funzionamento piuttosto semplice. Vediamo di seguito il funzionamento del meccanismo di codifica: La sequenza di bit originaria viene divisa in segmenti

di 6 bit ciascuno; ogni segmento di 6 bit viene trasformato in un carattere secondo le seguenti regole: ai valori da 0 a 25 sono fatti corrispondere i caratteri alfabetici maiuscoli (da 'A' a 'Z'); ai valori da 26 a 51 sono fatti corrispondere i caratteri alfabetici minuscoli (da 'a' a 'z'); i valori da 52 a 61 sono fatti corrispondere i caratteri numerici da 0 a 9; il valore 62 corrisponde al carattere '+'; il valore 63 corrisponde al carattere '/'; se la sequenza di bit originale non è multipla di 6, vengono aggiunti in fondo alla sequenza tanti zeri quanti è necessario per raggiungere il multiplo di sei più prossimo; la stringa risultante in Base64 deve essere multipla di quattro caratteri (ogni tre byte di input danno origine a quattro caratteri Base64), se necessario viene aggiunto un certo numero di occorrenze del carattere '='; le stringhe Base64 sono infine organizzate in linee della lunghezza massima di 76 caratteri, seguiti da CR e LF.

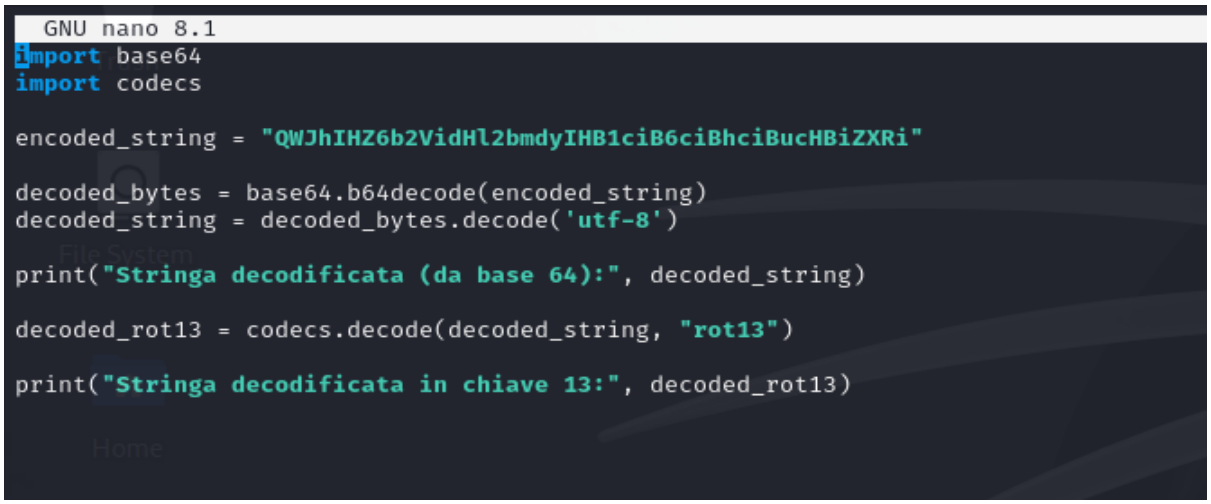
Come si può notare in questa spiegazione trovata su Internet (Base64 Encoder / Decoder - Converti una stringa in base64 e viceversa) nomina un convertitore presente sulla pagina stessa, linkata precedentemente tra le parentesi tonde.

Utilizzando il tool sopracitato ho concluso che la stringa da decodificare il “Cifrario di Cesare” è:

“Aba vzoebtyvngr pur zr ar nppbetb”

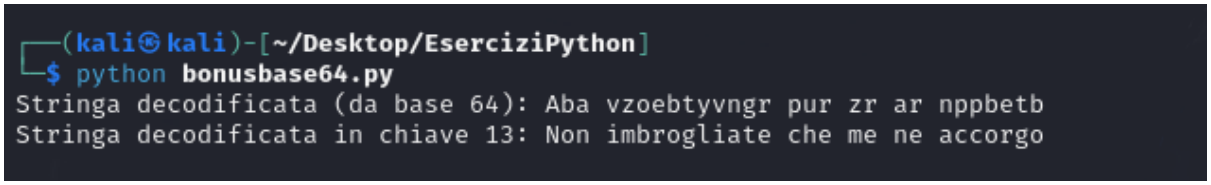
Come accennato prima, ho utilizzato il metodo “ROT13” e ho scritto un codice per poterne dare dimostrazione.

Riporto immagini di seguito:



```
  GNU nano 8.1
import base64
import codecs

encoded_string = "QWJhIHZ6b2VidHl2bmdyIHB1ciB6ciBhciBucHBiZXRi"

decoded_bytes = base64.b64decode(encoded_string)
decoded_string = decoded_bytes.decode('utf-8')

print("Stringa decodificata (da base 64):", decoded_string)

decoded_rot13 = codecs.decode(decoded_string, "rot13")

print("Stringa decodificata in chiave 13:", decoded_rot13)
```

Che ci darà come risultato quando andremo ad avviarlo:



```
┌──(kali㉿kali)-[~/Desktop/EserciziPython]
└─$ python bonusbase64.py
Stringa decodificata (da base 64): Aba vzoebtyvngr pur zr ar nppbetb
Stringa decodificata in chiave 13: Non imbrogliate che me ne accorgo
```

Esercizio su Kali opzionale

Data la traccia:

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.

Ho seguito i passaggi che riporto di seguito in ordine:

Installazione di OpenSSL:

```
sudo apt update
sudo apt install openssl
```

Installazione della libreria per Python:

```
sudo apt install python3-pip
pip3 install cryptography # risolvete eventuali errori
```

Comando per generare la chiave privata RSA:

```
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048
```

Comando per estrarre la chiave pubblica:

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Creiamo il file: **encdec.py** e al suo interno

Importiamo

Padding che 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

che useremo più avanti.

```python
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())
```

```python
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'))
```

Qui il **message** può essere preso da un input.

Non è possibile visualizzare il binario nel terminale e quindi si è scelto di convertirlo in base64 per renderlo leggibile.

Quando lanciate lo script assicuratevi di essere nella medesima cartella in cui avete generato i file pem.

Lanciando **encdec.py** dovreste ottenere:

Messaggio originale: Ciao, Epicode spacca!
Messaggio criptato: rBpx8RU9bCuaRZNIUH ...
Messaggio decriptato: Ciao, Epicode spacca!

Creiamo il file: **firma.py** e al suo interno

Identico al file precedente ma con

hashes che serve a creare l'hash del messaggio.

con **open** leggiamo le chiavi e le trasferiamo nelle variabili:

private_key
public_key

che useremo più avanti.

```python
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())
```

```python
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))
```

Qui il **message** può ess preso da un input.

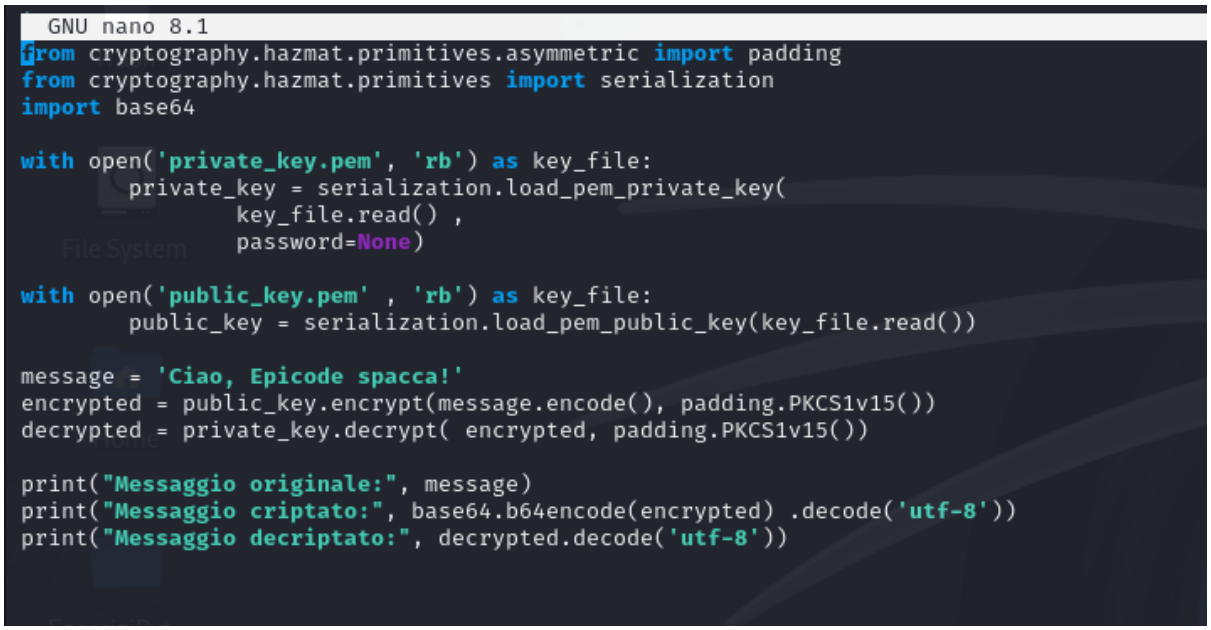
Notate che abbiamo us le stesse chiavi sia per firma che per la criptazione.

Lanciando **firma.py** dovreste ottenere:

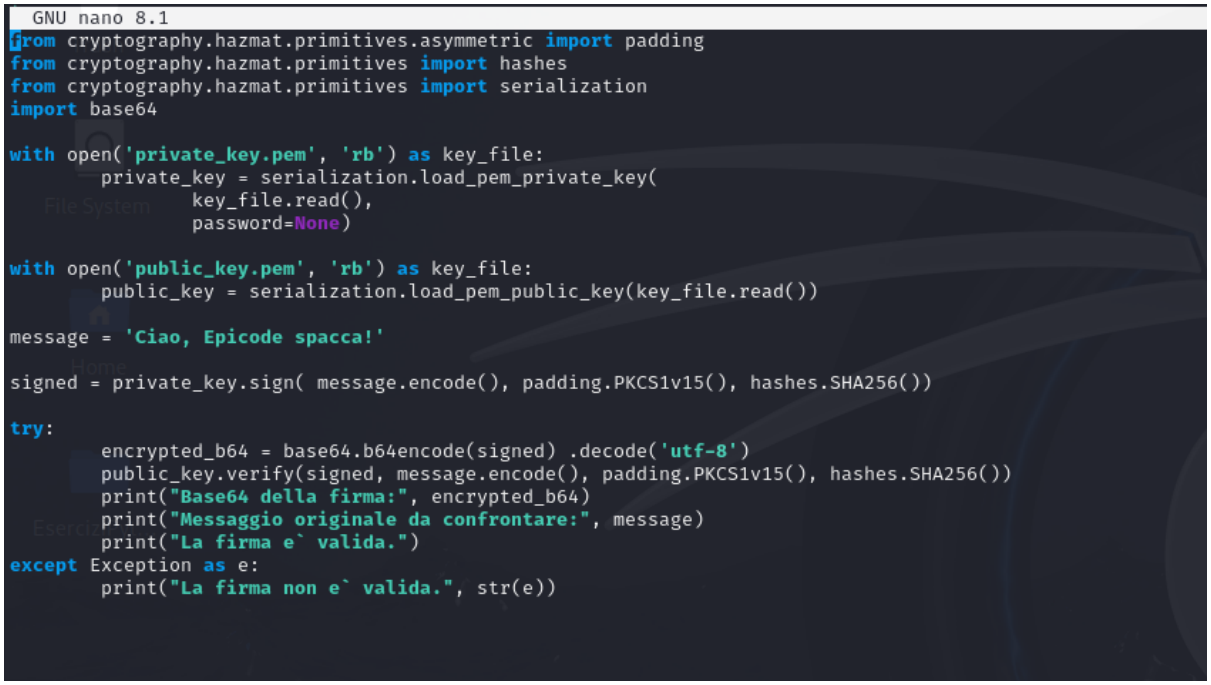
Base64 della firma: JSFBGOt0Fc...
Messaggio originale da confrontare: Ciao, Epicode spacca!
La firma è valida.

Dimostrazione dei passaggi eseguiti e risultato

Infine, a verifica dei passaggi appena enunciati, riporto gli screen per dimostrare che tutto funzioni come ci aspettavamo:



```
  GNU nano 8.1
from cryptography.hazmat.primitives.asymmetric import padding
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!'
encrypted = public_key.encrypt(message.encode(), padding.PKCS1v15())
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'))
```



```
  GNU nano 8.1
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
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())

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 e` valida.")
except Exception as e:
        print("La firma non e` valida.", str(e))
```


┌──(kali㉿kali)-[~/Desktop/EserciziPython]
└─$ python encdec.py
Messaggio originale: Ciao, Epicode spacca!
Messaggio criptato: mSvks6CDvVOX/+K4H3tiJvFAT6iDrQrxyPLiX9zBU0dpMknzkr/sSe/IImM097Z2H
E3RedJhWI56rwWsYBWgn3lkb+hMaif//WIg6aO9gLr+WdU9suZzld1O8ovnVi2VxjJD+5FXit5KVGASMZZRp7
xwto0huZ0uzeoF6utvELYuXTK2J0hOrOKTXeqdVUQZJrKWVDnLeLqaeT21aJk+HA6UQrC+P2VWa5fyeYO5Y4h
AIlPo88wLcD81UJJ8buvhxeUygN1LB/VsfSX8ZplS+twAbIJWdKzMHbZiiS5um44J8vfTHK1lg3Y8SluzBBRH
Y6g3ShZz5MVKZPQzkNAh0Q==
Messaggio decriptato: Ciao, Epicode spacca!
┌──(kali㉿kali)-[~/Desktop/EserciziPython]
└─$ nano firma.py
┌──(kali㉿kali)-[~/Desktop/EserciziPython]
└─$ python firma.py
Base64 della firma: ekI4e+nGotyulEfpnIHjmQ8WVnYZ1umFr4qS0KRSfBqCZMldUVtVqzttr9zV9+M6A4E6ONynQocmu1Ew9X4HU
sSHpsy3BpJ94owAHCVQGLyHPL1PWjchvEmiW/l6zM5/MiCgpaw4r6aUQQG5JoT80aaqgjA5mwYWMhV+qAo5hEGLfr5+1LtRTevMne8WYQ
GUmRW/g30NVRH0Z9LcZjCFJvLxoFinGl7MncXv2nVz9NG3PeAme1QlUGER5htYgZpjOpfBKsxIvSEkQ7tBfFZ2LInHETX9YHpUBzZE9oX
WzZFJOM/NxE4U++ffVezj7Cj4AHMRG6C8y5J+JBkRqqJdTA==
Messaggio originale da confrontare: Ciao, Epicode spacca!
La firma e` valida.
┌──(kali㉿kali)-[~/Desktop/EserciziPython]
└─$