Capture the Flag Manual

Alessio Gjergji
Author 2
Author 3
Author 4
Author 5

Indice

Capitolo 1

Crittografia

1.1 Conversioni

1.1.1 Decodifica da numero in base 2 a big endian

```
result = '101010'
result = int(result, 2).to_bytes((result.bit_length() + 7) // 8, 'big').
    decode('utf-8')
```

1.1.2 Decodifica da numero in base 2 a little endian

```
result = '101010'
result = int(result, 2).to_bytes((result.bit_length() + 7) // 8, 'little')
    .decode('utf-8')
```

1.1.3 Decodifica da numero in base 8 a big endian

```
result = '1234'
result = int(result, 8).to_bytes((result.bit_length() + 7) // 8, 'big').
    decode('utf-8')
```

1.1.4 Decodifica da numero in base 8 a little endian

```
result = '1234'
result = int(result, 8).to_bytes((result.bit_length() + 7) // 8, 'little')
    .decode('utf-8')
```

1.1.5 Decodifica da numero in base 10 a big endian

```
result = 1234
result = result.to_bytes((result.bit_length() + 7) // 8, 'big').decode('
    utf-8')
```

1.1.6 Decodifica da numero in base 10 a little endian

```
result = 1234
result = result.to_bytes((result.bit_length() + 7) // 8, 'little').decode(
    'utf-8')
```

1.1.7 Decodifica da numero in base 16 a big endian

```
result = '1234'
result = bytes.fromhex(result).decode('utf-8')
```

1.1.8 Decodifica da numero in base 16 a little endian

```
result = '1234'
result = bytes.fromhex(result)[::-1].decode('utf-8')
```

1.1.9 Decodifica da numero in base 32 a big endian

```
import base64

result = '1234'
result = base64.b32decode(result).decode('utf-8')
```

1.1.10 Decodifica da numero in base 32 a little endian

```
import base64

result = '1234'
result = base64.b32decode(result)[::-1].decode('utf-8')
```

1.1.11 Decodifica da numero in base 58 a big endian

```
import base58

result = '1234'
result = base58.b58decode(result).decode('utf-8')
```

1.1.12 Decodifica da numero in base 58 a little endian

```
import base58

result = '1234'
result = base58.b58decode(result)[::-1].decode('utf-8')
```

1.1.13 Decodifica da numero in base 64 a big endian

```
import base64

result = '1234'
result = base64.b64decode(result).decode('utf-8')
```

1.1.14 Decodifica da numero in base 64 a little endian

```
import base64

result = '1234'
result = base64.b64decode(result)[::-1].decode('utf-8')
```

1.2 One-Time Pad

Un dettaglio fondamentale per ottenere sicurezza perfetta è che la chiave sia lunga tanto quanto il messaggio.

Nel caso in cui la chiave sia molto corta (*e ripetuta, per esempio*) potrebbe essere possibile un attacco a forza bruta: provare tutte le chiavi candidate e vedere per quale si ottiene un risultato sensato. La ripetizione di pattern all'interno del messaggio cifrato è un indizio che la chiave sia corta.

```
def xor(a, b):
    return bytes([x^y for x,y in zip(a,b)])

num = bytes.fromhex('104e137f42')

for i in range(256):
    key = i.to_bytes(1, 'big')*len(num)
    result = xor(num, key)
    if result.isascii():
        result = result.decode('ascii')
        print('key: ', i, 'result: ', result)
```

Nel caso descritto la chiave è di un solo byte, quindi si prova a cifrare il messaggio con tutte le chiavi possibili e si controlla se il risultato è un testo `ASCII`.

1.3 Crittoanalisi differenziale

Per utilizzare la crittoanalisi differenziale è necessario avere a disposizione un *oracolo* che cifri un messaggio arbitrario con una chiave segreta. Installando il tool `MTP` possiamo ottenere una parziale implementazione del messaggio.

```
$ pip3 install mtp
$ mtp <file>
```

1.4 Libreria `PyCryptodome`

1.4.1 `DES`

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes
from binascii import hexlify, unhexlify

```

```
    # Chiave in formato esadecimale
    key_hex = '1bb4545801ca1e93'
    key = unhexlify(key_hex)

    # Testo in chiaro
    plaintext = 'La lunghezza di questa frase non e divisibile per 8'

    # Applicare lo schema di riempimento x923
    plaintext = pad(plaintext.encode('utf-8'), DES.block_size, style='x923')

    # Generare un vettore di inizializzazione casuale
    iv = get_random_bytes(DES.block_size)

    # stampo Che IV hai utilizzato (in esadecimale)
    print('IV: ' + hexlify(iv).decode('utf-8') + '\n')

    # Creare un oggetto DES in modalita' CBC
    cipher = DES.new(key, DES.MODE_CBC, iv)

    # Cifrare il testo in chiaro
    ciphertext = cipher.encrypt(plaintext)

    # Stampare il testo cifrato in formato esadecimale
    print("Text: " + hexlify(ciphertext).decode('utf-8') + "\n")
```

1.4.2 AES

```
    from Crypto.Cipher import AES
    from Crypto.Util.Padding import pad
    from Crypto.Random import get_random_bytes
    from binascii import hexlify, unhexlify

    # Generare una chiave casuale AES256 in formato esadecimale
    key = get_random_bytes(32)  # 32 byte per AES256

    # Testo in chiaro
    plaintext = 'Mi chiedo cosa significhi il numero nel nome di questo
        algoritmo.'

    # Applicare lo schema di riempimento PKCS7
    plaintext = pad(plaintext.encode('utf-8'), AES.block_size)

    # Generare un vettore di inizializzazione casuale
    iv = get_random_bytes(AES.block_size)

    # Stampo che IV hai utilizzato (in esadecimale)
    print("IV:", hexlify(iv).decode('utf-8'))

    # Creare un oggetto AES in modalita' CFB con segment size di 24
    cipher = AES.new(key, AES.MODE_CFB, iv, segment_size=24)

    # Cifrare il testo in chiaro
    ciphertext = cipher.encrypt(plaintext)

    # Stampare la chiave in formato esadecimale
```

```
28      print("Chiave:", hexlify(key).decode('utf-8'))
29
30      # Stampare il testo cifrato in formato esadecimale
31      print("Testo cifrato:", hexlify(ciphertext).decode('utf-8'))
```

1.4.3 Stream cipher

```
1       from Crypto.Cipher import ChaCha20
2       from binascii import unhexlify
3
4       # Chiave in formato esadecimale
5       key_hex = '9d6a8f...'
6       key = unhexlify(key_hex)
7
8       # Testo cifrato in formato esadecimale
9       ciphertext_hex = 'b7e9ac6...'
10      ciphertext = unhexlify(ciphertext_hex)
11
12      # Nonce in formato esadecimale
13      nonce_hex = '76c24201d...'
14      nonce = unhexlify(nonce_hex)
15
16      # Creare un oggetto ChaCha20
17      cipher = ChaCha20.new(key=key, nonce=nonce)
18
19      # Decifrare il testo cifrato
20      plaintext = cipher.decrypt(ciphertext)
21
22      # Stampa il testo in chiaro
23      print(plaintext.decode('utf-8'))
```

1.4.4 Hash

```
1       from Crypto.Hash import SHA256
2       from binascii import hexlify
3
4       # Testo in chiaro
5       plaintext = 'La lunghezza di questa frase non e divisibile per 8'
6
7       # Creare un oggetto SHA256
8       h = SHA256.new()
9
10      # Aggiornare l'hash con il testo in chiaro
11      h.update(plaintext.encode('utf-8'))
12
13      # Calcolare l'hash
14      digest = h.digest()
15
16      # Stampa l'hash in formato esadecimale
17      print(hexlify(digest).decode('utf-8'))
```

1.5 Aritmetica Modulare

Diciamo che $a \equiv b \pmod{n}$ se n divide $a - b$.

- $a \equiv b \pmod{n} \iff a \bmod n = b \bmod n$
- $a \equiv b \pmod{n} \iff \exists k \in \mathbb{Z} \mid a = b + kn$
- $a \equiv b \pmod{n} \iff \exists k \in \mathbb{Z} \mid a \bmod n = b + kn$

1.5.1 Proprietà

- $a \equiv a \pmod{n}$
- $a \equiv b \pmod{n} \implies b \equiv a \pmod{n}$
- $a \equiv b \pmod{n} \land b \equiv c \pmod{n} \implies a \equiv c \pmod{n}$
- $a \equiv b \pmod{n} \land c \equiv d \pmod{n} \implies a + c \equiv b + d \pmod{n}$
- $a \equiv b \pmod{n} \land c \equiv d \pmod{n} \implies ac \equiv bd \pmod{n}$

1.5.2 Aritmetica Modulare

La *aritmetica modulare* è una branca dell'aritmetica che si occupa delle operazioni sui numeri interi modulari rispetto a un modulo dato. Un concetto fondamentale è il *resto* di una divisione, spesso indicato come $a \mod m$, che rappresenta il residuo della divisione di a per m.

Un'applicazione comune dell'aritmetica modulare è nel calcolo del *mcd* (massimo comune divisore) di due numeri. La funzione di Euclide è una tecnica classica che sfrutta l'aritmetica modulare per calcolare l'mcd. Ad esempio, se abbiamo due numeri a e b, possiamo utilizzare la funzione di Euclide estesa per trovare i coefficienti di Bezout x e y tali che $a \cdot x + b \cdot y = \texttt{mcd}(a, b)$.

$$\texttt{gcd}(a, b) = ax + by \tag{1.1}$$

In Python creiamo una funzione che calcola l'mcd di due numeri utilizzando la funzione di Euclide estesa.

```python
def gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, y, x = gcd(b % a, a)
        return g, x - (b // a) * y, y
```

Questo concetto è spesso applicato in crittografia, dove la sicurezza si basa su operazioni modulari e il teorema di Fermat.