Capture the Flag Manual

Alessio Gjergji
Nicolò Piccoli
Davide Rossignolo
Author 4
Author 5

Indice

1 Software Security OlyCyber **3**
- 1.1 Introduzione . . . 3
- 1.2 Assembly `x86_64` . . . 3
- 1.3 Buffer Overflow . . . 3
  - 1.3.1 Accessi out of bound . . . 4
- 1.4 Reverse Engineering . . . 4
  - 1.4.1 Binary . . . 4
  - 1.4.2 Memoria a basso livello . . . 4
  - 1.4.3 Spazio virtuale linux user space . . . 5
  - 1.4.4 metodologie di base di reverse . . . 5

2 Software Security **6**
- 2.1 Elf File . . . 6
- 2.2 Memoria . . . 6
  - 2.2.1 Storage Size . . . 6
  - 2.2.2 Hexadecimal . . . 6
  - 2.2.3 Registri `x86_32` . . . 7
  - 2.2.4 x86 Memory Management . . . 7
- 2.3 Intel x86 Instruction Set . . . 7
- 2.4 Debugging con GDB . . . 8
  - 2.4.1 Comandi di GDB . . . 8

3 Comandi terminale **9**
- 3.1 Introduzione . . . 9
- 3.2 Comandi di base . . . 9
- 3.3 Operazioni sulle directory . . . 9
  - 3.3.1 cd . . . 9
  - 3.3.2 ls . . . 10
- 3.4 Operazioni sui file . . . 10
  - 3.4.1 touch . . . 10
  - 3.4.2 cat . . . 10
  - 3.4.3 cp, mv, rm, file . . . 10
  - 3.4.4 head, tail . . . 11

3.4.5 strings . . . 11
3.5 Ricerca . . . 11
3.5.1 sort, unique . . . 11
3.5.2 grep . . . 11
3.5.3 find . . . 11
3.6 Connessioni ssh o tcp . . . 12
3.7 Gestione dei processi . . . 12

4 Crittografia **13**
4.1 Conversioni . . . 13
4.1.1 Decodifica da numero in base 2 a big endian . . . 13
4.1.2 Decodifica da numero in base 2 a little endian . . . 13
4.1.3 Decodifica da numero in base 8 a big endian . . . 13
4.1.4 Decodifica da numero in base 8 a little endian . . . 13
4.1.5 Decodifica da numero in base 10 a big endian . . . 13
4.1.6 Decodifica da numero in base 10 a little endian . . . 14
4.1.7 Decodifica da numero in base 16 a big endian . . . 14
4.1.8 Decodifica da numero in base 16 a little endian . . . 14
4.1.9 Decodifica da numero in base 32 a big endian . . . 14
4.1.10 Decodifica da numero in base 32 a little endian . . . 14
4.1.11 Decodifica da numero in base 58 a big endian . . . 14
4.1.12 Decodifica da numero in base 58 a little endian . . . 14
4.1.13 Decodifica da numero in base 64 a big endian . . . 14
4.1.14 Decodifica da numero in base 64 a little endian . . . 15
4.2 One-Time Pad . . . 15
4.3 Crittoanalisi differenziale . . . 15
4.4 Libreria `PyCryptodome` . . . 15
4.4.1 `DES` . . . 15
4.4.2 `AES` . . . 16
4.4.3 Stream cipher . . . 17
4.4.4 Hash . . . 17
4.4.5 RSA . . . 18
4.5 Aritmetica Modulare . . . 18
4.5.1 Proprietà . . . 18
4.5.2 Aritmetica Modulare . . . 19

Capitolo 1

Software Security OlyCyber

1.1 Introduzione

I numeri interi, in questo caso `int32`, possono essere rappresentati in due maniera, Big-Endian (*cifra significativa a sinistra*) e Little-Endian (*cifra significativa a destra*).

1.2 Assembly `x86_64`

Ogni istruzione assembly ha degli operandi (*registri*) e un'operazione.

La notazione per architetture intel è del tipo ⟨`op`⟩⟨`destinazione`⟩⟨`sorgente`⟩, i registri hanno la seguente struttura: `AH`, `AL` → 8 bit, `AX` → 16 bit, `EAX` → 32 bit, `RAX` → 64 bit.

Tra le operazioni di base che troviamo ci sono:

- `MOV`⟨`dst`⟩⟨`src`⟩
- `PUSH`⟨`src`⟩ oppure `POP`⟨`src`⟩
- `ADD` oppure `SUB`⟨`dst`⟩⟨`src`⟩
- `CALL`⟨`pc`⟩ oppure `RET`

Tra i salti condizionali invece abbiamo:

- `CMP`⟨$\texttt{opn1}_1$⟩⟨$\texttt{opn}_2$⟩: confronta due valori e imposta delle flag
- `J`⟨`condizione`⟩⟨`pc`⟩: salta a PC se le flag soddisfano la condizione

1.3 Buffer Overflow

Consideriamo di aver dichiarato in `C` un'istruzione del tipo "`char name[100];`", cosa succede se l'utente ha un nome che supera i 100 caratteri?. Può succedere che con `scanf()` andiamo a scrivere caratteri oltre la fine di name, andando a sovrascrivere la memoria che lo segue,

generando quindi un *buffer overflow* che va a corrompere la memoria, ossia scriviamo dati in posizioni che il programmatore non aveva previsto fossero modificate.

Se corrompiamo abbastanza bene la memoria possiamo addirittura prendere il controllo del processo (*arbitrary code execution*).

1.3.1 Accessi out of bound

Se abbiamo una `struct` che dichiara un `int a[2]` e un `int b[3]`, nel momento in cui scrivo in `a[2]` o in `b[0]` non cambia niente, sono equivalenti.

1.4 Reverse Engineering

1.4.1 Binary

Gli eseguibili nativi sono file che contengono codice macchina eseguibile dal processore, contengono anche informazioni usate dal sistema operativo per caricarlo in memoria.

Il formato `.elf` è flessibile e serve a rappresentare i file binari, in linux è usato per rappresentare eseguibili e librerie condivise, ad alto livello invece risulta come un insieme di strutture che descrivono come caricare in memoria i dati salvati nello stesso file.

Per analizzare file `ELF` abbiamo alcuni strumenti tra cui:

- `readelf`: stampa le informazioni contenute nei file `.elf`
- `nm`: stampa tutti i simboli contenuti nel file `.elf`
- `objdump`: stampa le informazioni contenute nel file oggetto, è più specifico rispetto a readelf
- `lld`: stampa gli oggetti condivisi necessari all'esecuzione del programma
- `lief`: libreria python per analizzare e modificare file `.elf`.

1.4.2 Memoria a basso livello

Se vogliamo considerare un'astrazione della memoria troviamo vari livelli:

- **dati tipati**: byte interpretati
- linguaggi di programmazione
- textbfmemoria virtuale: sequenza di byte indirizzabili, spazio indipendente per processo, solo alcune aree mappate(con la fisica)
- sistema operativo
- textbfmemoria fisica: sequenza di byte indirizzabili

1.4.3 Spazio virtuale linux user space

- `Text` → codice eseguibile
- `Data` → dati globali inizializzati
- `BSS` → dati globali azzerati
- `Heap` → allocazioni dinamiche
- `Librerie` → binari librerie dinamiche
- `Stack` → var locali, record di attivazione

1.4.4 metodologie di base di reverse

I più famosi tool per analisi statica sono *Ghidra, IDA, Binary Ninja, Radare2* e servono per analizzare il codice binario.

- `JADX`: reverse bytecode `java`
- `dnSpy`, `ILSpy`: reverse bytecode `.NET`
- `uncompyle6/unpyc`: reverse bytecode `python`
- `Iuadec`: reverse bytecode `LUA`.

Tool per analisi dinamica:

- `gdb`: da usare con `GEF` o `PWNDBG`
- `radare2` integra features comode per il reversing
- rr timeless debugging
- `frida`: inietta codice js in qualsiasi punto del programma
- `ida`: debugger gui disponibile free.

Capitolo 2

Software Security

2.1 Elf File

È il file standard per gli eseguibili `UNIX`, `Elf` sta per Executable and Linkable Format, è essenzialmente un file binario che contiene varie informazioni tra cui:

- **header**: descrive il contenuto del file per l'esecuzione
- **Pht (program header table)**: da informazioni su come si crea l'immagine del processo
- **Sequenza di sezioni**: contengono ciò che serve per il linking
- **Section header table**: descrizione delle sezioni precedenti

Per Windows l'equivalente è PE, per Mac Mach-O.

2.2 Memoria

2.2.1 Storage Size

- `WORD`: 2 bytes;
- `DWORD`: 4 bytes;
- `QWORD`: 8 bytes;

La cifra più significativa è a sinistra, quella meno è a destra.

2.2.2 Hexadecimal

Si usa come forma compatta dei numeri binari, si va da 0 a 9 e da A a F. Ogni digit in hex rappresenta 4 bits, quindi 2 digit un bytes, in `C` sono scritti come `0XFA1D`...

2.2.3 Registri `x86_32`

Ha registri general purpose a 32 bit. Sono strutturati nella seguente maniera: AH,AL 8 bit ciascuno, AX 16 bit, `EAX` 32 bit. EAX è storicamente usato come accumulatore, `ECX` come counter, inoltre ci sono `ESP` (*stack pointer*) e EBP (*base pointer*). Se passiamo a `x86_64` si aggiungono i registri a 64 bit che si nominano del tipo: `RAX`.

2.2.4 x86 Memory Management

La memoria è semplicemente una sequenza di bytes, ognuno con un indirizzo unico. I compilatori potrebbero introdurre padding per cambiare l'ordine dei dati per ottimizzare, per questo motivo ci torna comodo vederla come una matrice di tot righe con n bytes (n=processor word), se sono a 64 bit n=8. Un indirizzo è una locazione in memoria, un **puntatore** è un oggetto che mantiene un indirizzo. I bytes possono essere ordinati in memoria in due maniere:

- **Big-Endian**: il byte meno significativo ha indirizzo più alto
- **Little-Endian**: il byte meno significativo ha indirizzo più basso

2.3 Intel x86 Instruction Set

La notazione per architetture Intel è del tipo $\langle op \rangle \langle destinazione \rangle \langle sorgente \rangle$. Vediamo ora alcune istruzioni:

- `mov`: muove dati da un src a un registro dst
- `push`: mette l'operando sullo stack
- `pop`: rimuove l'operando dallo stack
- `lea`: carica ciò che c'è all'indirizzo [] in dst
- `add`: fa la somma e salva in op1
- `sub`: fa la differenza e salva in op1
- `inc`: incrementa di 1
- `and/or/xor`: fa l'operazione e salva in op1
- `jump`: salto non condizionale a label
- `cmp`: compara contenuto di op1 con op2
- `je/jne/jz/jg/jge/jl/jle`: in base a cmp fa salto condizionale
- `call`: usato per chiamare funzioni
- `ret`: implementa il ritorno da funzione

Per quanto riguarda lo stack, questo è formato da degli stack frames, uno per ogni funzione chiamata. Il **stack pointer** punta all'ultimo elemento nello stack (il primo inserito). Per

quanto riguarda ogni singolo frame, abbiamo che il **base pointer (frame pointer)** punta all'indirizzo dal quale partono le variabili locali.

2.4 Debugging con GDB

Posso usare gdb in varie maniere:

- gdb >program <
- gdb >program <>pid <
- gdb -p >pid <

2.4.1 Comandi di GDB

- run: fa partire il programma in gdb
- set args: imposta gli argomenti del programma
- show args: mostra gli argomenti del programma
- help: comandi disponibili
- break: mette breakpoint alla prossima istruzione
- break location: mette un breakpoint alla locazione location(preceduta da *)
- break [location] if >condition <: mette un breakpoint data una condizione
- continue(c): va al prossimo breakpoint(se esiste)
- nexti(ni): esegue solo un'istruzione
- frame [>selection <]: stampa una descrizione dello stack frame selezionato
- info frame [>selection <]: stampa una descrizione più informativa rispetto a frame
- disas: disassembla una funzione
- print(p): stampa il contenuto di un indirizzo o registro(*0x092a3e or $eax)
- x: non ho ben capito
- call: chiama una funzione
- set: modifica il valore di una locazione di memoria o di un registro

Capitolo 3

Comandi terminale

3.1 Introduzione

Durante una ctf potremmo trovarci di fronte ad alcune challenge in cui è necessario l'utilizzo di vari comandi della shell per recuperare la flag richiesta e quindi passare alla challenge successiva. Di seguito vedremo alcuni comandi della shell per sistemi UNIX o macOS che possono tornare utili.

3.2 Comandi di base

Se si conosce un comando ma non si sa come utilizzarlo è bene consultare il manuale scrivendo sul terminale **man** ⟨*command*⟩. Se non è presente la pagina del manuale provare a specificare il flag –**help**.

3.3 Operazioni sulle directory

3.3.1 cd

Per navigare attraverso il filesystem utilizziamo il comando **cd dir**.

- **cd esempio** (ci spostiamo nella cartella *esempio*)
- **cd Dekstop/esempio** (ci spostiamo nella cartella *esempio* identificata dal suo path)
- **cd ..** (ci permette di spostarci nella cartella superiore)
- **cd** ∼ (ci spostiamo nella home directory)
- **cd /** (ci spostiamo nella root directory)

Per creare una cartella utilizziamo il comando **mk dir**, se invece vogliamo vedere la cartella corrente utlizziamo il comando **pwd**.

3.3.2 ls

Se vogliamo vedere i file all'interno di una cartella utilizziamo il comando **ls**.

- **ls** (mostra i file nella cartella corrente)
- **ls Desktop/esempio** (mostra i file all'interno della cartella *esempio* identificata dal path)

Tra le opzioni del comando **ls** possiamo trovare:

- **-a** (mostra tutti i file, inclusi quelli nascosti)
- **-r** (inverte l'ordine della lista)
- **-t** (ordina in base all'ultimo modificato)
- **-S** (ordina per dimensione del file)

3.4 Operazioni sui file

3.4.1 touch

Per creare un file utilizzare il comando **touch file**.

3.4.2 cat

Il comando **cat** permette di concatenare file e stampare il loro contenuto sullo standard output.

- **cat file** (stampa il contenuto del file, se vengono specificati più file li concatena e stampa il contenuto, e.g. cat file file2)
- **cat < -file** (permette di stampare il contenuto di un file con il nome che inizia con un dash)
- **cat** "$nomefileconspazi$" (permette di stampare il contenuto di un file che contiene spazi nel nome)
- **cat .file** (stampa il contenuto del file nascosto)

3.4.3 cp, mv, rm, file

- **cp file file2** (copia file in file2)
- **mv file file2** (rinomino file in file2)
- **rm file** (elimino file)
  - **rm -r** (rimuovo le directory e i loro contenuti)
  - **rm -d** (rimuovo direcotry vuote)
- **file file1** (ritorna il tipo di file1)

3.4.4 head, tail

- **head file1** (ritorna le prime 10 linee di file1)
- **tail file1** (ritorna le ultime 10 linee di file1)

3.4.5 strings

Il comando **strings** stampa una sequenza di stringhe leggibili all'interno di un file.

- **strings file**
  - con il flag **-n number-of-lines** (specifichiamo la lunghezza minima delle stringhe)
  - con il flag **-e encoding** (specifichiamo la codifica)
  - con il flag **-w** (includiamo gli spazi bianchi)
  - con il flag **-s** (il separatore per l'output)

3.5 Ricerca

3.5.1 sort, unique

Il comando **sort file** permette di ordinare le linee all'interno di un file, il comando **unique -u** permette di mostrare le linee uniche non duplicate, questi due comandi possono essere comodi da usare in combinazione attraverso l'utilizzo di una pipe: **sort nomefile — unique -u**.

3.5.2 grep

Il comando **grep pattern files** permette di cercare un determinato pattern in ogni file, i pattern andrebbero specificati sempre compresi tra doppi apici.

- **grep -i** (ricerca case-insensitive)
- **grep -r** (ricerca ricorsiva)
- **grep -v** (ricerca invertita)
- **grep -o** (mostra solo la parte di file che ha matchato il pattern)

3.5.3 find

Il comando **find** permette di cercare dei file all'interno del filesystem.

- **find /percorso -name** "$filename$" (ricerca per nome)
- **find /percorso -name** "$*.txt$" (ricerca per estensione)
- **find /percorso -type f -size +1M** (ricerca per dimensione)
- **find /percorso -user utente -group gruppo** (ricerca per proprietario e gruppo)
- **find /percorso -mtime -7** (ricerca per data di modifica)

3.6 Connessioni ssh o tcp

Connessione ad una risorsa in ssh:

- **ssh -p numero-porta utente@indirizzo-del-server**

Connessione tramite tcp:

- **nc host port**

3.7 Gestione dei processi

- **ps** (mostra uno snapshot dei processi)
- **top** (mostra i processi real-time)
-
- **kill pid** (termina un processo con il pid=pid)
- **pkill name** (termina un processo col nome=name)
- **killall name** (termina tutti i processi con il nome che inizia per name)

Capitolo 4

Crittografia

4.1 Conversioni

4.1.1 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')
```

4.1.2 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')
```

4.1.3 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')
```

4.1.4 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')
```

4.1.5 Decodifica da numero in base 10 a big endian

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

4.1.6 Decodifica da numero in base 10 a little endian

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

4.1.7 Decodifica da numero in base 16 a big endian

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

4.1.8 Decodifica da numero in base 16 a little endian

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

4.1.9 Decodifica da numero in base 32 a big endian

```
    import base64

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

4.1.10 Decodifica da numero in base 32 a little endian

```
    import base64

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

4.1.11 Decodifica da numero in base 58 a big endian

```
    import base58

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

4.1.12 Decodifica da numero in base 58 a little endian

```
    import base58

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

4.1.13 Decodifica da numero in base 64 a big endian

```
    import base64

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

4.1.14 Decodifica da numero in base 64 a little endian

```
import base64

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

4.2 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`.

4.3 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>
```

4.4 Libreria `PyCryptodome`

4.4.1 `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")
```

4.4.2 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'))
```

4.4.3 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'))
```

4.4.4 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'))
```

4.4.5 RSA

L'algoritmo funziona con due primi p e q e un esponente pubblico e. L'algoritmo è il seguente:

1. Calcolare $n = p \cdot q$
2. Calcolare $\phi(n) = (p-1)(q-1)$
3. Calcolare $d = e^{-1} \pmod{\phi(n)}$
4. La chiave pubblica è (n, e), la chiave privata è (n, d)

Per cifrare un messaggio m si calcola $c = m^e \pmod{n}$, per decifrarlo si calcola $m = c^d \pmod{n}$.

```
p = 36652254321470221037
q = 31629674241453983353
e = 65537
n = p * q
phi = (p - 1) * (q - 1)
d = pow (e, -1, phi)
print("n = ", n)

print("phi(n) = ", phi)

print("d = ", d)

stringa = "Stringa di prova"

numero = int(''.join(format(ord(char), '08b') for char in stringa), 2)

cifrato = pow(numero, d, n)

print("La stringa cifrata: ", cifrato)
```

4.5 Aritmetica Modulare

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

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

4.5.1 Proprietà

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

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

4.5.2 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 $a \cdot x + b \cdot y = \texttt{mcd}(a, b)$.

$$\texttt{gcd}(a, b) = ax + by \tag{4.1}$$

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

```
1 def gcd(a, b):
2     if a == 0:
3         return b, 0, 1
4     else:
5         g, y, x = gcd(b % a, a)
6         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.