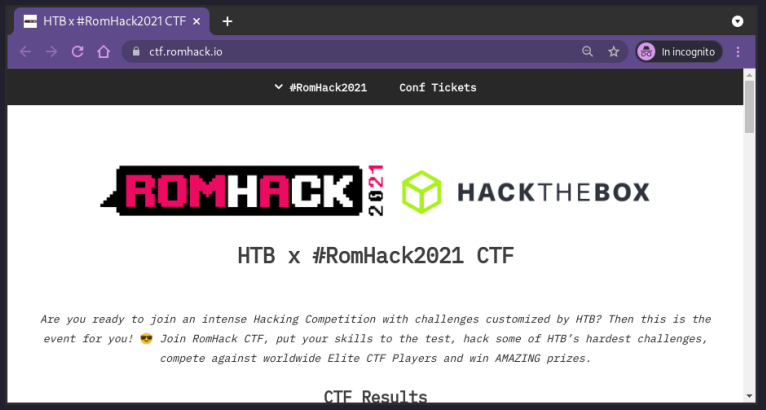
CTF HTB x ROMHACK 2021

Crypto 01 - Nonce-Sense

TABLE OF CONTENTS

INTRODUZIONE

Con la nuova edizione (2021) del **RomHack**, una conferenza organizzata dalla associazione non-profit **Cyber Saiyan**, è stata organizzata una **Capture The Flag** (CTF) assieme ad **Hack The Box**.



ctf.romhack.io

La CTF si è tenuta tra il 18 e il 19 settembre 2021.

In questo video voglio riportare la prima challenge della categoria **crypto**, chiamata

nonce-sense

CHALLENGE

La challenge consiste nel connettersi ad un server TCP scritto in python che permette di fare le seguenti cose:

```
Welcome to beta signing system of Best CA LTD.
[1] Sign a message.
[2] Verify a message.
[3] Get public key.
```

Opzione 1: firmare un messaggio

```
Insert message to sign:
HELLO
Message signed:
3e20222539
Signature:
43ea5657959d47e07fb731a32f28387c357cbaf16ca312db41086961,
55c73d31efc29f8e96ccde4f7ea05a6de6db65241ea326c0a9b36c86
```

Opzione 2: verificare la firma di un messaggio

```
Insert message to verify:
HELLO
Insert r:
0x43ea5657959d47e07fb731a32f28387c357cbaf16ca312db41086961
Insert s:
0x55c73d31efc29f8e96ccde4f7ea05a6de6db65241ea326c0a9b36c86
Valid signature.
```

Opzione 3: stampare i parametri pubblici del server

```
Public Key:
p = 0x924716506fc956bec56c22210097fa13817f1761584613ef
    ceaa4231db7f42fec45e2fb20d0bc4e21557c7c334f0b6f8fc
    ....

q = 0xc20d8f66f2fd7df6f6263c2bd8a019822e6f3b50d4706b0ecf68a747

g = 0x66cc1911444ca45148398c49085cdd9ef1f1f3e5f41f7de
    1e8c1c58e19185164db1552b3c5d54a708b63b129cb506e92
    ...

y = 0x12635c44eaefe103fd6ba70491ccc38d98d70b5e6ac14f4
    5b57fc3696256ccd4ac3078d082d42b7d5c49dbd475da453a
    86008234b26eafccd6fd68814eb4d9dcaa3e654e2df2c219b
    ...
```

Oltre a poter interagire col server abbiamo anche a disposizione il codice sorgente del server nel file `server.py`.

Dal codice vediamo che per risolvere la challenge ed ottenere la FLAG dobbiamo trovare un modo per ottenere la firma del messaggio "give me flag"

```
req.sendall(b'Insert message to verify:\n')
msg = req.recv(4096).strip()
req.sendall(b'Insert r:\n')
r = int(req.recv(4096).strip(), 16)
msg ==b'give me flag'
    req.sendall(b'Insert s:\n')
    s = int(req.recv(4096).decode().strip(), 16)
    if dsa.verify(msg,r,s):
            if msg == b'give me flag':
                req.sendall(FLAG + b'\n')
                exit(1)
            else:
                req.sendall(b'Valid signature.\n')
    else:
        req.sendall(b'Invalid signature.\n')
```

Il problema è che l'unico messaggio che non ci possiamo far firmare dal server è proprio la stringa "give me flag".

```
req.sendall(b'Insert message to sign:\n')
msg = req.recv(4096).strip()
if msg ==b'give me flag':
        req.sendall(b'Forbidden message!\n')
        continue
h = SHA.new(msg).digest()
msg,k = dsa.get_k(msg)
h = bytes_to_long(h)
r, s = dsa.sign(h,k)
req.sendall(b'Message signed:\n' +\
        msg.hex().encode() + b'\n' + \
        b'Signature:\n' + \
        hex(r)[2:].encode() + b',' + hex(s)[2:].encode() +b'\n')
```

Come possiamo procedere?

CRYPTOGRAPHY CONCEPTS

Prima di vedere la soluzione è importante ripassare alcuni concetti base ripresi dalla **crittografia**, e in particolare della **crittografia a chiave pubblica**.

SYMMETRIC CRYPTOGRAPHY

Per la maggior parte della storia, la tecnologia principale utilizzata per nascondere il contenuto di messaggi è stata la **crittografia a chiave simmetrica**.

Questo tipo di crittografia si basa sull'esistenza di una **chiave segreta** che viene utilizzata sia per cifrare il testo che per decifrarlo.

La chiave è detta **simmetrica** in quanto ha un ruolo simmetrico rispetto alla cifrazione/decifrazione.

Per far funzionare uno schema del genere, questa chiave segreta deve essere **condivisa** tra mittente e destinatario:

- Il mittente la utilizza per **cifrare** il messaggio.
- Il destinatario la utilizza per **decifrare** il messaggio.

ONE-TIME PAD (VERNAM CIPHER)

Il **cifrario di Vernam** è un primo esempio di crittosistema a chiave simmetrica.

L'idea dietro al **cifrario di Vernam** è quella di calcolare il messaggio cifrato facendo lo XOR tra i bytes del messaggio e i bytes di una **chiave random**.

$$\text{Plaintext} \oplus \text{Random key} \longrightarrow \text{Encrypted text}$$

Ricordiamo che l'operatore XOR lavora a livello dei bits ed è definito come segue

A	B	$A \oplus B$
0	0	0
1	0	1
0	1	1
1	1	0

Per fare lo XOR tra due bytes l'idea è quella di considerare i singoli bits e fare lo XOR bit a bit.

Esempio: XOR tra "hello" e "secret"

Consideriamo il messaggio "hello", e supponiamo che la chiave da utilizzare sia la chiave "secret".

Per capire i bytes di queste stringhe possiamo utilizzare il binario `hexdump`.

```
[leo@archlinux server]$ echo -n "hello" | hexdump -C
00000000  68 65 6c 6c 6f                                    |hello|

[leo@archlinux server]$ echo -n "secret" | hexdump -C
00000000  73 65 63 72 65 74                                 |secret|
```

Esempio: XOR tra "hello" e "secret"

In questo caso abbiamo

```
h   e  l  l  o
68 65 6c 6c 6f

s   e  c  r  e  t
73 65 63 72 65 74
```

Osservazione: la codifica utilizzata in questo contesto è quella **ASCII**. Esistono anche altre codifiche, come ad esempio **UTF-8**.

Esempio: XOR tra "hello" e "secret"

Lo XOR del primo byte delle due stringhe è quindi calcolato come segue

$$
\begin{aligned}
68 &\longrightarrow 01101000 \\
73 &\longrightarrow 01110011 \\
68 \oplus 73 &\longrightarrow 00011011
\end{aligned}
$$

una volta calcolato, possiamo rappresentare il byte risultante in formato esadecimale

$$00011011 \longrightarrow 0\text{x}1\text{b}$$

Iterando il passo precedente per ogni byte del messaggio originale – eventualmente ripetendo la chiave quante volte è necessario – otteniamo il seguente risultato

```
 h  e  l  l  o
 s  e  c  r  e  t
1b 00 0f 1e 0a
```

$$\text{"hello"} \oplus \text{"secret"} \longrightarrow 0\text{x}1\text{b}000\text{f}1\text{e}0\text{a}$$

In generale quindi troviamo il seguente schema

$$
\begin{aligned}
P &= p_1, p_2, \ldots, p_m \quad (\text{plaintext}) \\
K &= k_1, k_2, \ldots, k_l \quad (\text{key}) \\
C &= c_1, c_2, \ldots, c_m \quad (\text{ciphertext})
\end{aligned}
$$

con

$$
\begin{array}{ccc}
c_1 = p_1 \oplus k_1, & c_2 = p_2 \oplus k_2, & c_3 = p_3 \oplus k_3 \\
\ldots & c_i = p_i \oplus k_i & \ldots
\end{array}
$$

Questo tipo di cifratura necessita di alcune importanti ipotesi di utilizzo per poter essere considerata sicura:

- La chiave deve essere generata in modo random.
- La chiave deve essere lunga tanto quanto il messaggio.
- Per ogni nuovo messaggio da cifrare, si deve generare una nuova chiave.

Se una di queste ipotesi non è rispettata, allora il sistema è rotto in quanto diventa possibile **inferire informazioni sulla chiave**.

KNOWN PLAINTEXT ATTACK

Quando la chiave per cifrare non viene cambiata, il cifrario di Vernam è vulnerabile ad un attacco noto con il nome di **Known Plaintext Attack**.

In questo tipo di attacco si assume che l'attaccante ha a disposizione sia il testo cifrato che il testo originale.

Notiamo che questa conoscenza basta a rompere la cifratura con XOR.

Sia c_1 il risultato dello XOR tra il primo byte del messaggio p_1 e il primo byte della chiave k_1.

Come facciamo a calcolare il byte della chiave k_1 se conosciamo c_1 e p_1?

L'idea è quella di fare lo XOR tra c_1 e p_1.

Per come abbiamo calcolato c_1, segue che

$$\begin{aligned} c_1 \oplus p_1 &= (p_1 \oplus k_1) \oplus p_1 \\ &= p_1 \oplus k_1 \oplus p_1 \\ &= p_1 \oplus p_1 \oplus k_1 \\ &= \mathbf{0} \oplus k_1 \\ &= k_1 \end{aligned}$$

In generale quindi se la chiave non viene cambiata ci basta trovare una coppia (plaintext, ciphertext) in cui il plaintext è lungo tanto quanto la chiave, e così facendo siamo in grado di trovare i bytes della chiave.

ASYMMETRIC CRYPTOGRAPHY

Con l'avvento del web e la necessità di proteggere tante comunicazioni tra individui sconosciuti tra loro, la crittografia a chiave simmetrica è stata parzialmente sostituita con un nuovo schema crittografico:

la crittografia asimmetrica.

La crittografia asimmetrica si basa sulla generazione di **due chiavi** k_1 , k_2 legate tra loro dalle seguenti relazioni:

- Tutto ciò che è cifrato con una delle due chiavi può essere decifrato solamente dall'altra chiave.
- Da una delle due chiavi è possibile derivare l'altra in modo "efficiente", ma non vale il viceversa.

Chiamiamo **chiave privata** la chiave da cui è possibile derivare l'altra chiave in modo efficiente. La rimanente è detta **chiave pubblica**.

In altre parole, abbiamo che

Dalla chiave privata posso facilmente ottenere quella pubblica, ma da quella pubblica non posso ottenere quella privata in modo efficiente.

Per questa ragione la crittografia asimmetrica è anche molto spesso chiamata

crittografia a chiave pubblica/privata.

L'enorme potenziale della crittografia a chiave pubblica sta nel fatto che tutti possono utilizzare la mia chiave pubblica per cifrare i messaggi, ma solo io, il possessore della relativa chiave privata, sono in grado di leggere i messaggi cifrati con la mia chiave pubblica.

In altre parole, diventa possibile comunicare in modo sicuro anche senza lo scambio a priori di chiavi simmetriche.

MESSAGE INTEGRITY

Oltre a permette di **cifrare** i messaggi, la crittografia a chiave pubblica può anche essere utilizzata per **firmare** i messaggi, garantendone così l'**integrità**.

La **firma** di un messaggio non è altro che una sequenza di bytes che viene allegata al messaggio.

Se calcolata utilizzando uno schema crittografico, permette al destinatario di controllare:

- Che il messaggio è stato firmato con la chiave privata associata ad una determinata chiave pubblica.
- Che durante il trasporto il messaggio non è stato modificato.

Molti dei risultati utilizzati dalla crittografia sono stati ripresi dalla **teoria dei numeri**, e in particolare dall'**aritmetica modulare**.

MODULAR ARITHMETIC

L'**aritmetica modulare** è una tipologia di aritmetica che ha svariati utilizzi sia nell'informatica in generale, che nella crittografia nello specifico.

Questa tipologia di aritmetica infatti ci permette di lavorare con degli **insiemi di elementi finiti**.

Nell'aritmetica tradizionale l'insieme degli elementi preso in considerazione è l'insieme dei numeri naturali

$$\mathbb{N} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots\}$$

o eventualmente l'insieme degli interi

$$\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3, \ldots\}$$

Quando lavoriamo con un'aritmetica modulare invece abbiamo sempre un insieme finito di elementi

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$$

Il valore n è detto **modulo** e stabilisce la particolare aritmetica con cui vogliamo lavorare. Abbiamo una aritmetica diversa per ogni valore di $n \in \mathbb{N}^+$.

Dato che abbiamo un insieme di elementi finito, le operazioni di somma $(+)$ e prodotto $(\cdot)$ vengono modificate per far in modo che sommando due numeri da $\mathbb{Z}_n$ non usciamo mai da $\mathbb{Z}_n$.

Le operazioni tipiche in un contesto modulare $\mathbb{Z}_n$ vengono modificate come segue:

- Prima si esegue la tipica operazione della somma $(+)$ o del prodotto $(\cdot)$.
- Una volta ottenuto un valore, se quest'ultimo è più grande di n, si divide il risultato per n e si considera il resto nella divisione per n.

Esempio: Aritmetica modulare con $n = 5$

Siano $a = 3, b = 4 \in \mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$.

Troviamo,

$$\begin{aligned} a + b \mod 5 &= 3 + 4 \mod 5 \\ &= 7 \mod 5 \\ &= 2 \end{aligned}$$

Esempio: Aritmetica modulare con $n = 5$

Siano $a = 3, b = 4 \in \mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$.

Troviamo,

$$\begin{aligned} a \cdot b \mod 5 &= 3 \cdot 4 \mod 5 \\ &= 12 \mod 5 \\ &= 2 \end{aligned}$$

DIGITAL SIGNATURE ALGORITHM (DSA)

Il **Digital Signature Algorithm** è un particolare algoritmo di firma digitale basato sulla crittografia a chiave pubblica.

La sicurezza dell'algoritmo non è dimostrabile formalmente, ma è basata su determinate ipotesi relative alla difficoltà computazionale del problema del **logaritmo discreto**.

In particulare, sotto determinate ipotesi, si ha che

- Elevare un numero x ad un numero y modulo n può essere fatto in modo efficiente.

$$x, y, n \longrightarrow x^y \mod n \quad (\text{OK})$$

- L'operazione inversa invece non sembra essere calcolabile in modo efficiente.

$$x^y \mod n, x, n \longrightarrow y \quad (\text{NOT OK})$$

L'algoritmo DSA può essere descritto procedendo come segue:

- Generazione dei parametri.
- Firma dei messaggi.
- Verifica dei messaggi.
- Correttezza (non mostrata).

PARAMETERS GENERATION

La generazione dei parametri avviene in due steps:

Nel primo step una serie di **parametri pubblici** (p, q, g) vengono calcolati come segue:

- q è un numero `primo` con N bit, dove $N = 2048$.
- p è un numero primo con L bit, dove $L = 224$.
- h è scelto inn modo randomico tra $\{2, \ldots, p-2\}$.
- g è calcolato come $g = h^{(p-1)/q}$.

Osservazione: esistono altre scelte di lunghezze N e L per i primi generati.

Nel secondo step si calcola la coppia di chiavI (pubblica, privata) per l'utente che deve firmare i messaggi:

- La chiave privata x è calcolata in modo randomico tra $\{1, \ldots, q-1\}$.
- La chiave pubblica è calcolata come $g^x \mod p$.

Osservazione: risolvendo il logaritmo discreto siamo anche in grado di calcolare dalla chiave pubblica quella privata, rompendo l'intero sistema.

Nel server questa generazione viene fatta utilizzando la funzione `Crypto.PublicKey.DSA`

```
class DSA:
        def __init__(self):
                self.pKey = Crypto.PublicKey.DSA.generate(2048)
```

MESSAGE SIGNING

Una volta che i parametri del server sono stati generati è possibile firmare un messaggio m come segue:

- Si sceglie un intero k in modo `randomico` tra $\{1, \ldots, q-1\}$.
- Si calcola

$$r = (g^k \mod p) \mod q$$

- Si calcola

$$s = (k^{-1}(H(m) + x \cdot r)) \mod q$$

La firma è quindi data dalla coppia `(r, s)`.

MESSAGE VERIFICATION

Infine, per verificare una firma `(r, s)` per un messaggio `m` si procede come segue:

- Si verifica che $0 < r < q, \quad 0 < s < q.$
- Si calcola $w = s^{-1} \mod q$
- Si calcola $u_1 = H(m) \cdot w \mod q$
- Si calcola $u_2 = r \cdot w \mod q$
- Si calcola $v = (g^{u_1} y^{u_2} \mod p) \mod q$

La firma è valida se e solo se

$$v = r$$

Per maggiori informazioni rimando alla pagina su wikipedia.

DSA (wikipeda)

SOLUTION (THEORY)

Ricordiamo che in una implementazione corretta, per firmare un messaggio `m` l'algoritmo DSA genera un intero casuale `k` tra `1` e `q - 1`, dove `q` è un parametro reso pubblico.

Cosa fa però l'implementazione presente nel server di questa challenge?

prima di chiamare la funzione `dsa.sign(h, k)`, il server genera i due parametri `h` e `k`.

```
req.sendall(b'Insert message to sign:\n')
msg = req.recv(4096).strip()
if msg ==b'give me flag':
        req.sendall(b'Forbidden message!\n')
        continue
h = SHA.new(msg).digest()
msg,k = dsa.get_k(msg)
h = bytes_to_long(h)
r, s = dsa.sign(h,k)
```

`h` è ottenuto calcolando lo **SHA256** del messaggio.

`k` invece è calcolato dalla funzione `dsa.get_k(msg)`.

Andando a vedere l'implementazione della funzione, troviamo

```
def get_k(self, msg):
    kmax = self.pKey.q
    msg = [ a ^ b for (a,b) in zip(msg, cycle(KEY)) ]
    msg = bytes(msg)
    k = bytes_to_long(msg) % self.pKey.q
    return msg, k
```

Osserviamo il seguente importantissimo fatto:

k non è generato in modo casuale, ma è calcolato in funzione del messaggio e del valore di KEY

Questo significa che se scoprissimo il valore di KEY, saremmo in grado di generare il particolare k utilizzato per firmare un dato messaggio m.

Questa è la vulnerabilità che ci permetterà di ottenere la chiave privata del server x, e dunque di firmare qualsiasi messaggio vogliamo con la chiave del server.

L'attacco sarà strutturato come segue:

1. Come prima cosa dobbiamo trovare un modo per ottenere il valore di KEY.
2. Poi calcoliamo una firma valida `(r, s)` per un messaggio noto `m`.
3. Infine, otteniamo la chiave privata `x` del server utilizzando la seguente formula

$$x = (r^{-1} \mod q) \cdot (s \cdot k - h) \mod q$$

Una volta ottenuta la chiave x possiamo firmare qualsiasi messaggio vogliamo, e quindi anche il messaggio

"give me flag"

EXPLOITATION (PRACTICE)

Vediamo in pratica come effettuare i vari steps dell'attacco.

STEP 1: TROVARE IL VALORE KEY

Per estrarre il valore di KEY basta notare che quando firmiamo un messaggio il server ci ritorna il messaggio cifrato tramite la funzione `dsa.get_k`

```
h = SHA.new(msg).digest()
msg,k = dsa.get_k(msg)
h = bytes_to_long(h)
r, s = dsa.sign(h,k)
req.sendall(b'Message signed:\n' +\
        msg.hex().encode() + b'\n' + \
        b'Signature:\n' + \
        hex(r)[2:].encode() + b',' + hex(s)[2:].encode() +b'\n')
```

La funzione `dsa.get_k` non fa altro che calcolare lo XOR tra i bytes del messaggio e i bytes della chiave, andando a ripetere i bytes di quest'ultima quante volte è necessario per coprire tutti i bytes del messaggio.

```
def get_k(self, msg):
    kmax = self.pKey.q
    msg = [ a ^ b for (a,b) in zip(msg, cycle(KEY)) ]
    msg = bytes(msg)
    k = bytes_to_long(msg) % self.pKey.q
    return msg, k
```

Per trovare il valore della chiave dobbiamo:

1. Capire la lunghezza della chiave.
2. Capire i bytes che formano la chiave.

Al fine di estrarre la chiave dal server sfruttiamo due cose:

- La chiave è statica e non viene mai cambiata.
- Il server ci permette di firmare (e quindi anche di cifrare) qualsiasi messaggio di qualsiasi lunghezza utilizzando la stessa chiave.

In altre parole, abbiamo un **known-plaintext attack** con una chiave statica, e quindi siamo in grado di rompere lo XOR ed ottenere la chiave.

Facendoci firmare messaggi contenenti la stessa lettera ma di lunghezza sempre più grande, siamo in grado di stabilire l'esistenza di eventuali patterns che si ripetono.

```
(04) AAAA                -> 37242f28
(08) AAAAAAAA            -> 37242f2837282528
(12) AAAAAAAAAAAA        -> 37242f28372828528372822 8
(16) AAAAAAAAAAAAAAAA -> 37242f28372825283728222837242f28
```

Notiamo cosa succede nell'ultima riga: dopo i primi 12 bytes, i restanti 4 sono uguali ai primi 4.

Cosa succede quindi se proviamo a firmare una stringa contenente 24 A? Otteniamo il seguente valore

```
37242f2837282528372822837242f2837282528372822 8
```

è possibile vedere che tale stringa si ripete dopo 12 bytes

```
37 24 2f 28 37 28 25 28 37 28 22 28
37 24 2f 28 37 28 25 28 37 28 22 28
```

In altre parole, abbiamo stabilito che

la chiave è formata da 12 bytes

Per capire quali sono i particolari bytes della chiave adesso ci basta semplicemente firmare una qualsiasi stringa di 12 bytes ed effettuare un **known-plaintext attack**:

```
(plaintext)  41 41 41 41 41 41 41 41 41 41 41 41
(ciphertext) 37 24 2f 28 37 28 25 28 37 28 22 28
-----------------------------------------------
(key)         ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?
```

Il seguente codice python automatizza il calcolo della chiave a partire da una coppia (plaintext, ciphertext)

```
PLAINTEXT_BYTES = b"AAAAAAAAAAAA"
CIPHERTEXT = "37242f28372852837282228"
CIPHERTEXT_BYTES = binascii.unhexlify(CIPHERTEXT)

KEY_BYTES = [chr(a ^ b) for
              (a,b) in zip(PLAINTEXT_BYTES, CIPHERTEXT_BYTES) ]

KEY = "".join(KEY_BYTES)
print(KEY)
```

Andandolo ad eseguire troviamo la seguente situazione

```
              A  A  A  A  A  A  A  A  A  A  A  A
(plaintext)  41 41 41 41 41 41 41 41 41 41 41 41
(ciphertext) 37 24 2f 28 37 28 25 28 37 28 22 28
-----------------------------------------------
(key)        76 65 6e 69 76 69 64 69 76 69 63 69
              v  e  n  i  v  i  d  i  v  i  c  i
```

La chiave statica utilizzata dal server è quindi

venividivici

STEP 2: OTTENERE UNA FIRMA (R, S)

Ottenere la firma `(r, s)` di un messaggio è estremamente semplice, in quanto ci basta chiedere al server di firmare il messaggio "HELLO"

```
Welcome to beta signing system of Best CA LTD.
[1] Sign a message.
[2] Verify a message.
[3] Get public key.1
Insert message to sign:
HELLO
Message signed:
3e20222539
Signature:
b7695f1e06b189d5bfa49ed84603b12c501312c1361b6b632523fe2e,
8ef7781dca38dadd90310328ee65b1b60032ddf9bcc33c041df26590
```

Nel nostro caso abbiamo

```
r = 0xb7695f1e06b189d5bfa49ed84603b12c501312c1361b6b632523fe2e
s = 0x8ef7781dca38dadd90310328ee65b1b60032ddf9bcc33c041df26590
```

STEP 3: CALCOLARE IL VALORE DI X

Una volta che conosciamo la coppia `(r, s)` per il messaggio "HELLO" possiamo calcolare la chiave privata dal server `x` tramite la seguente formula

$$x = (r^{-1} \mod q) \cdot (s \cdot k - h) \mod q$$

Notiamo infatti che siamo in grado di calcolare tutte le variabili tranne x in quanto:

- q è un parametro pubblico del server.
- h e k sono calcolati in funzione del messaggio e della chiave KEY.
- r e s sono stati ottenuti firmando il messaggio "HELLO"

Il seguente codice automatizza il calcolo dell'estrazione della chiave privata x

```
p = int("0xa7cde5c...", 0)
q = int("0xde0b903...", 0)
g = int("0x46c68dd...", 0)
y = int("0x51c8804...", 0)

# -- message sign (r, s)
msg = b"HELLO"

r = int("0x16e1c420fb...", 0)
s = int("0x805ef39629...", 0)

# -- computing h and k
h = bytes_to_long(SHA.new(msg).digest())
k = bytes_to_long(bytes([ a ^ b for (a,b) in zip(msg, cycle(KEY))

# -- computing x
```

STEP 4: FIRMA DI MESSAGGIO PER FLAG

Una volta che abbiamo x possiamo firmare il messaggio "give me flag" come segue

```
msg = b"give me flag"
h = bytes_to_long(SHA.new(msg).digest())
k = bytes_to_long(bytes([ a ^ b for (a,b) in zip(msg, cycle(KEY))

r = pow(g, k, p) % q
s = (inverse(k, q) * (h + x * r)) % q

print(hex(r)[2:])
print(hex(s)[2:])
```

STEP 5: PROFIT :)

A questo punto possiamo submittare la firma del messaggio per ottenere la flag

```
Welcome to beta signing system of Best CA LTD.
[1] Sign a message.
[2] Verify a message.
[3] Get public key.2
Insert message to verify:
give me flag
Insert r:
a975b98d926d482285ef3e6d16a98b87876e2780d5705a10288e64a
Insert s:
b80bb666b23dc35f536d990ef9e76956799dba10ec6174e53125bc79
HTB{this_is_a_flag}
```