Cryptography 2

Carmen Casulli & Fabio Giovanazzi

04/04/2024

Challenge 0: RSA intro

A tutorial for RSA and other utility functions

Challenge 0: RSA intro

Useful functions from **PyCryptodome**

- Use **bytes_to_long** and **long_to_bytes** from **Crypto.Util.Number** to convert bytes to numbers and viceversa
- Use Python **pow** function to compare powers modulo a number:

```
pow(a, b, n) = a^b   (mod n)
```

- You can use it to compute the inverse too:

```
pow(a, -1, n) = 1/a   (mod n)
```

UNIVERSITÀ DI TRENTO

Challenge 1: Random key

My previous XOR encryption algorithm was affected by the Many-Time Pad vulnerability. I fixed it by XORring each byte with a different pseudo-random number, so MTP no more!

```python
def generate_byte(self) -> int:
    prev_x = self.x
    self.x = (self.a * self.x + self.c) % MOD
    return prev_x
```

UNIVERSITÀ DI TRENTO

Challenge 1: Random key

My previous XOR encryption algorithm was affected by the Many-Time Pad vulnerability. I fixed it by XORring each byte with a different pseudo-random number, so MTP no more!

Hints:

- Can you obtain the first few numbers generated by the LCG?

Challenge 1: Random key

My previous XOR encryption algorithm was affected by the Many-Time Pad vulnerability. I fixed it by XORring each byte with a different pseudo-random number, so MTP no more!

Hints:

- Can you obtain the first few numbers generated by the LCG?
- Modular arithmetic FTW

Challenge 1 solution

The recurrence relation for an LCG is the following:

$$x_{n+1} \equiv a x_n + c \pmod{m}$$

=> there are only three parameters we need to figure out: `a`, `c` and x_0

Challenge 1 solution

The recurrence relation for an LCG is the following:

$$x_{n+1} \equiv a x_n + c \pmod{m}$$

=> there are only three parameters we need to figure out: `a`, `c` and x_0

Since we know the flag starts with `"Uni"`, we can XOR the first three bytes of the encrypted message to obtain x_0, x_1 and x_2.

UNIVERSITÀ DI TRENTO

Challenge 1 solution

$$x_1 \equiv ax_0 + c \pmod{m}$$

$$x_2 \equiv ax_1 + c \pmod{m}$$

$$\implies c \equiv x_1 - ax_0$$

$$\implies x_2 \equiv a^2 x_0 + ac + c \pmod{m}$$

$$\implies x_2 \equiv \cancel{a^2 x_0} + ax_1 - \cancel{a^2 x_0} + x_1 - ax_0 \pmod{m}$$

$$\implies x_2 - x_1 \equiv a(x_1 - x_0) \pmod{m}$$

$$\implies a \equiv (x_2 - x_1)(x_1 - x_0)^{-1} \pmod{m}$$

UNIVERSITÀ DI TRENTO

Challenge 1 solution

```
flag_first_piece = b"Uni"
x0 = enc_flag[0] ^ flag_first_piece[0]
x1 = enc_flag[1] ^ flag_first_piece[1]
x2 = enc_flag[2] ^ flag_first_piece[2]

a = (x2 - x1) * pow(x1 - x0, -1, MOD) % MOD
c = (x1 - a * x0) % MOD
lcg = LCG(a, c, x0)
```

Challenge 2: HLE Bank

The High Level Equity Bank presents myHLE, a new app to manage your accounts

```python
salt = os.urandom(32)
def generate_password(token: bytes) -> str:
    return hashlib.sha1(salt + token).hexdigest()
```

Challenge 2: HLE Bank

The High Level Equity Bank presents myHLE, a new app to manage your accounts

Hints:

- How could you login as `"richperson"`?

Challenge 2: HLE Bank

The High Level Equity Bank presents myHLE, a new app to manage your accounts

Hints:

- How could you login as `"richperson"`?
- Look for a library that does HLE for you

Challenge 2 solution

- The login procedure takes the login **token** and verifies that its hash corresponds with the **password**, which acts as a signature for the token

Challenge 2 solution

- The login procedure takes the login **token** and verifies that its hash corresponds with the **password**, which acts as a signature for the token
- We can try to forge a token ending in `"||||richperson"`, and login as `"richperson"`, who has enough money to buy the flag

Challenge 2 solution

- The login procedure takes the login **token** and verifies that its hash corresponds with the **password**, which acts as a signature for the token
- We can try to forge a token ending in `"||||richperson"`, and login as `"richperson"`, who has enough money to buy the flag
- But how can we generate a corresponding password?

Challenge 2 solution

- The login procedure takes the login **token** and verifies that its hash corresponds with the **password**, which acts as a signature for the token
- We can try to forge a token ending in `"||||richperson"`, and login as `"richperson"`, who has enough money to buy the flag
- But how can we generate a corresponding password?
- The SHA-1 hash is vulnerable to **Hash Length Extension**, so we can try to extend the hash of a token we already know with `"||||richperson"`

Challenge 2 solution

- The login procedure takes the login **token** and verifies that its hash corresponds with the **password**, which acts as a signature for the token
- We can try to forge a token ending in `"||||richperson"`, and login as `"richperson"`, who has enough money to buy the flag
- But how can we generate a corresponding password?
- The SHA-1 hash is vulnerable to **Hash Length Extension**, so we can try to extend the hash of a token we already know with `"||||richperson"`
- We can use a library to do the HLE for us, e.g. `hlextend.py`

UNIVERSITÀ DI TRENTO

Challenge 3: Diffie Hellman MITM

Perform a MITM attack against Alice and Bob in this simulator

Challenge 3 solution

The usual **Diffie Hellman** flow:

1. Alice and Bob agree on two numbers `p` and `g`, the modulus and the generator
2. A and B generate, respectively, `a` and `b`, and keep them for themselves
3. A and B send to each other, respectively, $g^a \bmod p$ and $g^b \bmod p$
4. A and B obtain the same shared secret, respectively, as $(g^b \bmod p)^a \bmod p$ and $(g^a \bmod p)^b \bmod p$
5. A and B construct a cipher using the shared key and can communicate freely

Challenge 3 solution

In case of a **Man In The Middle** attack:

2. - A and B generate a_{alice} and a_{bob}, and keep them for themselves
   - The attacker generates (or hardcodes), b_{alice} and b_{bob}
3. - A and B send to the attacker $g\hat{}a_{alice}$ mod p and $g\hat{}a_{bob}$ mod p
   - The attacker sends to A and B $g\hat{}b_{alice}$ mod p and $g\hat{}b_{bob}$ mod p
4. A and B obtain two different shared secrets, respectively, $g\hat{}a_{bob}\hat{}b_{bob}$ mod p and $g\hat{}a_{alice}\hat{}b_{alice}$ mod p, and the attacker knows them both

UNIVERSITÀ DI TRENTO

Challenge 4: Factorization

Screw symmetric algorithms and strange block modes, RSA is much more resilient!

```
p = getPrime(1024)
q = p + 1
while not isPrime(q):
    q += random.randint(1, 10000)
```

UNIVERSITÀ DI TRENTO

Challenge 4: Factorization

Screw symmetric algorithms and strange block modes, RSA is much more resilient!

Hints:

- `n = p*(p+x)`

Challenge 4 solution

Since `n = p*q` with `q=(p+x)` close to `p`, we can start from `p=q≈sqrt(n)`

Challenge 4 solution

Since `n = p*q` with `q=(p+x)` close to `p`, we can start from `p=q≈sqrt(n)`

We can gradually decrease `p` and increase `q`, while keeping `pq` as close to `n` as possible, until we find `pq=n`

Challenge 4 solution

Since `n = p*q` with `q=(p+x)` close to `p`, we can start from `p=q≈sqrt(n)`

We can gradually decrease `p` and increase `q`, while keeping `pq` as close to `n` as possible, until we find `pq=n`

```
p = q = math.isqrt(n)
while True:
    if p * q == n: break
    p -= 1
    while p*q < n: q += 1
```

Challenge 4 solution

Once we have p and q we can obtain $d \equiv e^{-1} \pmod{\varphi(n)}$

where $\varphi(n) = (p-1)(q-1)$ is Euler's totient function

Challenge 5: Fast RSA

Using a smaller public key should make things faster...

```
p, q = getStrongPrime(512), getStrongPrime(512)
n = p * q
e = 3
```

Challenge 5: Fast RSA

Using a smaller public key should make things faster...

Hints:

- What if the flag is also not too big?

Challenge 5: Fast RSA

Using a smaller public key should make things faster...

Hints:

- What if the flag is also not too big?
- You need an integer root algorithm

Challenge 5 solution

- A small `e` is being used, and let's suppose the message is also not too big

  $\approx> m^e < n$

  $=> c = (m^e \bmod n) = m^e$, i.e. the modulus has no effect

  $=> m = \sqrt[e]{c}$

Challenge 5 solution

- A small `e` is being used, and let's suppose the message is also not too big
  $\approx> m^e < n$
  $=> c = (m^e \bmod n) = m^e$, i.e. the modulus has no effect
  $=> m = \sqrt[e]{c}$
- In order to take the cubic root of `c` we need an integer root function in Python
- We can implement it ourselves with binary search…
  … or use something like `sagemath`

```
from sage.all import *
m = Integer(c).nth_root(3)
```

UNIVERSITÀ DI TRENTO