Zombie Rolled

10 minutes to read

We are given the Python source code to encrypt the flag:

```python
from Crypto.Util.number import getPrime, bytes_to_long
from fractions import Fraction
from math import prod
from hashlib import sha256
from secrets import randbelow

# I hope no one cares about Kerckhoff's principle :)
from secret import derive_public_key, FLAG


def fraction_mod(f, n):
    return f.numerator * pow(f.denominator, -1, n) % n


class PublicKey:

    def __init__(self, pub):
        self.pub = pub
        self.f = self.magic(pub)
        self.nb = (self.f.denominator.bit_length() + 7) // 8

    def encrypt(self, m):
        return pow(m, self.f.numerator, self.f.denominator)

    def verify(self, m, sig):
        s1, s2 = sig
        h = bytes_to_long(sha256(m.to_bytes(self.nb, "big")).digest())
        a, b = m, h
        r = self.encrypt(s1)
        c = self.encrypt(s2)
        return fraction_mod(self.magic((a, b, c)), self.f.denominator) == r

    @staticmethod
    def magic(ar):
        a, b, c = ar
        return Fraction(a, b) + Fraction(b, c) + Fraction(c, a)


class PrivateKey(PublicKey):

    def __init__(self, priv, pub):
        super().__init__(pub)
        if self.magic(priv) != self.f:
            raise ValueError("Invalid key pair")
        self.priv = priv
        self.d = pow(self.f.numerator, -1, prod([x - 1 for x in priv]))

    def decrypt(self, c):
        return pow(c, self.d, self.f.denominator)

    def sign(self, m):
        h = bytes_to_long(sha256(m.to_bytes(self.nb, "big")).digest())
        a, b = m, h
        c = randbelow(1 << self.nb)
        r = fraction_mod(self.magic((a, b, c)), self.f.denominator)
        s1 = self.decrypt(r)
        s2 = self.decrypt(c)
        return s1, s2
    @staticmethod
    def generate(nbits):
        while True:
            try:
                priv = tuple([getPrime(nbits) for _ in range(3)])
                pub = derive_public_key(priv)
                return PrivateKey(priv, pub)
            except ValueError:
                pass


def main():
    assert len(FLAG) <= 64
    m = bytes_to_long(FLAG)

    key = PrivateKey.generate(1024)
    data = f"pub = {key.pub}\n"

    # make sure it really works
    enc = key.encrypt(m)
    assert key.decrypt(enc) == m
    sig = key.sign(m)
    assert key.verify(m, sig)

    # mixing them :)
    mix = [sig[0] + sig[1], sig[0] - sig[1]]
    mix = [key.encrypt(x) for x in mix]
    data += f"mix = {mix}"

    with open("output.txt", "w") as f:
        f.write(data)


if __name__ == "__main__":
    main()
```

And we have the result in `output.txt`:

```
pub = (-679149827688896546432684514781159016843208241259733038415608446794483893
mix = [3200976701593042098723095494719309934171433880776124794975204623426639349
```

Source code analysis

The server generates 3 prime numbers p, q and r, which form the private key.

Public key generation

There is an unknown function called `derive_public_key` that is used to get a public key from the private one. The comment about Kerckhoff's principle is relevant here because this approach is security through obscurity, which is not always secure. Kerckhoff's principle states that security must rely on the keys, rather than the algorithms.

The result of `derive_public_key` is a list of 3 numbers, given to us.

These public-key numbers are transformed with `magic` :

$$\text{magic}(a, b, c) = \frac{a}{b} + \frac{b}{c} + \frac{c}{a} = \frac{a^2c + b^2a + c^2b}{abc}$$

```python
def magic(ar):
    a, b, c = ar
    return Fraction(a, b) + Fraction(b, c) + Fraction(c, a)
```

We can apply `magic` to the 3 public-key numbers and get a fraction f with numerator N and denominator D. These two values are used in the rest of the script.

Encryption and decryption

In order to encrypt a message, the server takes the message m in decimal format and computes ct:

$$\mathrm{ct} = m^N \mod D$$

```
    def encrypt(self, m):
        return pow(m, self.f.numerator, self.f.denominator)
```

In order to decrypt, the server calculates

$$m = \mathrm{ct}^d \mod D \qquad d = N^{-1} \mod (p-1)(q-1)(r-1)$$

```
        self.d = pow(self.f.numerator, -1, prod([x - 1 for x in priv]))

    def decrypt(self, c):
        return pow(c, self.d, self.f.denominator)
```

The code also shows us that both functions work as expected, and they are inverses:

```
# make sure it really works
enc = key.encrypt(m)
assert key.decrypt(enc) == m
sig = key.sign(m)
assert key.verify(m, sig)
```

Signature and verification process

As can be seen, there are also signature and verification routines that work as expected.

In order to sign, the server takes the message m, its SHA256 hash h and a random value c to compute $\mathrm{magic}(m, h, c) = \frac{A}{B}$.

The values A and B are used to compute $R = A \cdot B^{-1} \mod D$ with `fraction_mod`:

```python
def fraction_mod(f, n):
    return f.numerator * pow(f.denominator, -1, n) % n
```

The signature is just a pair of values:

$$(s_1, s_2) = (R^d, c^d) \mod D$$

```python
    def sign(self, m):
        h = bytes_to_long(sha256(m.to_bytes(self.nb, "big")).digest())
        a, b = m, h
        c = randbelow(1 << self.nb)
        r = fraction_mod(self.magic((a, b, c)), self.f.denominator)
        s1 = self.decrypt(r)
        s2 = self.decrypt(c)
        return s1, s2
```

The verification function takes the message and the values (s_1, s_2) to perform almost the same operations. First, it finds $R = s_1^N \mod D$ and $c = s_2^N \mod D$, then it compares it with the expected value of R by using m, h and c:

```python
    def verify(self, m, sig):
        s1, s2 = sig
        h = bytes_to_long(sha256(m.to_bytes(self.nb, "big")).digest())
        a, b = m, h
        r = self.encrypt(s1)
        c = self.encrypt(s2)
        return fraction_mod(self.magic((a, b, c)), self.f.denominator) == r
```

We are given some values that are related to the signature of the flag. Specifically, we have $(s_1 + s_2)^N \mod D$ and $(s_1 - s_2)^N \mod D$, where (s_1, s_2) is the signature of the flag:

```
# mixing them :)
mix = [sig[0] + sig[1], sig[0] - sig[1]]
mix = [key.encrypt(x) for x in mix]
data += f"mix = {mix}"
```

Solution

Since we know that `encrypt` and `decrypt` are inverse functions and that work well, comparing with a classic RSA, we have N as the public exponent and D as the public modulus. Hence, in order to decrypt we need to find the private exponent, which needs to be $d = N^{-1} \mod \phi(D)$. The server actually computes $d = N^{-1} \mod (p-1)(q-1)(r-1)$. This fact determines that $\phi(D) = (p-1)(q-1)(r-1)$ and therefore $D = pqr$.

On the other hand, the static method `generate_key` of `PrivateKey` generates the private key and derives the public one. Then, it returns an instance of `PrivateKey`:

```
@staticmethod
def generate(nbits):
    while True:
        try:
            priv = tuple([getPrime(nbits) for _ in range(3)])
            pub = derive_public_key(priv)
            return PrivateKey(priv, pub)
        except ValueError:
            pass
```

The constructor does the following:

```python
class PrivateKey(PublicKey):

    def __init__(self, priv, pub):
        super().__init__(pub)
        if self.magic(priv) != self.f:
            raise ValueError("Invalid key pair")
        self.priv = priv
        self.d = pow(self.f.numerator, -1, prod([x - 1 for x in priv]))
```

Class `PrivateKey` extends from `PublicKey`, and its constructor is called with the public key:

```python
class PublicKey:

    def __init__(self, pub):
        self.pub = pub
        self.f = self.magic(pub)
        self.nb = (self.f.denominator.bit_length() + 7) // 8
```

Here, the value of the fraction `f` is computed, which holds numerator N and denominator D.

But look at this ckeck:

```python
        if self.magic(priv) != self.f:
            raise ValueError("Invalid key pair")
```

This fact tells us that N and D, which are the output of `magic` and the public key (let's say P, Q, R) must satisfy that

$$f = \text{magic}(P, Q, R) = \frac{P}{Q} + \frac{Q}{R} + \frac{R}{Q} = \frac{P^2R + Q^2P + R^2Q}{PQR} = \frac{N}{D}$$

But remember that $D = pqr$, which means that $PQR = k \cdot pqr$ for some $k \in \mathbb{Z}$. On the other hand,

$$\text{magic}(p, q, r) = \frac{p^2r + q^2p + r^2q}{pqr} = \frac{N}{D}$$

As a result, we have that

$$\begin{cases} N = p^2r + q^2p + r^2q \\ D = pqr \end{cases}$$

While debugging in Python, I found that

$$\begin{cases} PQR \equiv 0 \pmod{D} \\ PQR \equiv 0 \pmod{D^2} \\ PQR \not\equiv 0 \pmod{D^3} \end{cases}$$

Also, $P, Q, R \not\equiv 0 \pmod{D}$, but

$$\begin{cases} PQ \equiv 0 \pmod{D} \\ QR \equiv 0 \pmod{D} \\ RP \equiv 0 \pmod{D} \end{cases} \qquad \begin{cases} PQ \not\equiv 0 \pmod{D^2} \\ QR \not\equiv 0 \pmod{D^2} \\ RP \not\equiv 0 \pmod{D^2} \end{cases}$$

This means that PQR is a multiple of $(pqr)^2$, but in each pair PQ, QR, RP, we can only factor (pqr). After some tries, we can come up with the following structure:

$$P = \alpha \cdot pq \qquad Q = \beta \cdot qr \qquad R = \gamma \cdot rp$$

for some integers α β and γ. The above relations work since

$$\begin{cases} PQR = \alpha\beta\gamma \cdot p^2q^2r^2 \equiv 0 \pmod{D^2} \\ PQ = \alpha\beta \cdot pq^2r \equiv 0 \pmod{D} \\ QR = \beta\gamma \cdot pqr^2 \equiv 0 \pmod{D} \\ RP = \gamma\alpha \cdot p^2qr \equiv 0 \pmod{D} \end{cases}$$

Therefore, we have

$$\begin{cases} \gcd(Q, D) = \gcd(\beta \cdot qr, pqr) = qr \\ \gcd(R, D) = \gcd(\gamma \cdot rp, pqr) = pr \end{cases}$$

And so, $r = \gcd(\gcd(Q, D), \gcd(R, D))$. Trivially, we find p and q.

Having p, q and r we can compute $d = N^{-1} \mod \phi(D)$ and decrypt the `mix` we have. Moreover, we can easily find s_1 and s_2. Remember the values of s_1 and s_2:

$$s_1 = R^d \mod D \qquad s_2 = c^d \mod D$$

At this point, we can have the signatures R and c, and we can check that their bit lengths are correct. Notice that $R = A \cdot B^{-1} \mod D$, where

$$\frac{A}{B} = \text{magic}(m, h, c) = \frac{m}{h} + \frac{h}{c} + \frac{c}{m} = \frac{m^2c + mh^2 + hc^2}{mhc}$$

Here, we might need to apply Coppersmith method or a lattice-based technique, since the flag has at most 64 bytes (512 bits) and the SHA256 hash is a 256-bit integer, whereas D is around 3072 bits.

Let's get rid of inverses multiplying by (mhc):

$$(mhc)R = m^2c + mh^2 + hc^2 \mod D$$

As a result, we can define a polynomial:

$$P(x, y) = cx^2 + xy^2 + c^2y - cRxy \pmod D$$

Where $P(m, h) \equiv 0 \pmod D$. Since m and h are short, we can try to find small roots of this bivariate polynomial using Coppersmith method.

SageMath implementation

First of all, we take the values from `output.txt` and find N and D:

```
with open('output.txt') as fp:
    pub = literal_eval(fp.readline().split(' = ')[1])
    mix = literal_eval(fp.readline().split(' = ')[1])

P, Q, R = pub
f = magic(pub)
N, D = f.numerator, f.denominator
nb = (f.denominator.bit_length() + 7) // 8
```

Next, we find the private key p, q, r and assert that the values are correct:

```
r = gcd(gcd(Q, D), gcd(R, D))
p = gcd(Q, D) // r
q = gcd(R, D) // r

assert is_prime(p) and int(p).bit_length() <= 1024
assert is_prime(q) and int(q).bit_length() <= 1024
assert is_prime(r) and int(r).bit_length() <= 1024

assert p^2 * r + q^2 * p + r^2 * q == N
assert p * q * r == D
```

Once we have the private key, we can decrypt the `mix`, find the signature (s_1, s_2) and then compute R and c. The length of c is also verified:

```
d = pow(N, -1, (p - 1) * (q - 1) * (r - 1))
s1_p_s2 = pow(mix[0], d, D)
s1_m_s2 = pow(mix[1], d, D)

s1 = (s1_p_s2 + s1_m_s2) * pow(2, -1, D) % D
s2 = (s1_p_s2 - s1_m_s2) * pow(2, -1, D) % D

R = pow(s1, N, D)
```

```
c = pow(s2, N, D)
assert c.bit_length() <= nb
```

Finally, we need to define the bivariate polynomial and use Coppersmith method to find m and h. A nice implementation of multivariate Coppersmith method is made by defund. We can download the script and load it into SageMath to use the `small_roots` function. Once found a solution, we can confirm that the SHA256 hash matches and then we simply print out the flag:

```
load('coppersmith.sage')

m, h = PolynomialRing(Zmod(D), 'm, h').gens()
PP = m^2 * c + m * h^2 + h * c^2 - c * m * h * R
roots = small_roots(PP, bounds=(2 ** 512, 2 ** 256))

for root in roots:
    if root != (0, 0):
        m, h = root
        assert int(sha256(int(m).to_bytes(nb, 'big')).hexdigest(), 16) == h
        print(bytes.fromhex(hex(m)[2:]).decode())
```

Flag

And here we have the flag:

```
$ sage solve.sage
HTB{4_s3cur3_crypt0syst3m_sh0u1d_n0t_c0nt41n_s3cr3t_c0mp0n3nts!}
```

The full script can be found in here: `solve.sage` .

Contents