Zombie Rolled

13 minutes to read

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

```
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, n):
        self.pub = pub
```

```
        self.f = self.magic(pub)
        self.n = n
        self.nb = (self.n.bit_length() + 7) // 8

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

    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.n) == r

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


class PrivateKey(PublicKey):

    def __init__(self, priv, pub, n):
        super().__init__(pub, n)
        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.n)

    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.n)
        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)
                n = prod(priv)
                return PrivateKey(priv, pub, n)
            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 = (-210460004064658545149323270027125585053769531535575407350548992758255183
mix = [275773011415907738914969843263592398183730172961383638482156597038541352 7
```

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`:

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

$$= \frac{(a+b)^2(c+a) + (b+c)^2(a+b) + (c+a)^2(b+c)}{(a+b)(b+c)(c+a)}$$

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

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:

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

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

Where $n = p \cdot q \cdot r$, the product of the private prime numbers.

In order to decrypt, the server calculates

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

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

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

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 $\text{magic}(m, h, c) = \frac{A}{B}$.

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

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

The signature is just a pair of values:

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

```
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.n)
    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 $R = s_1^N \mod n$ and

$c = s_2^N \mod n$, then it compares it with the expected value of R by using m, h and c:

```
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.n) == r
```

We are given some values that are related to the signature of the flag. Specifically, we have $(s_1 + s_2)^N \mod n$ and $(s_1 - s_2)^N \mod n$, 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 $n = p \cdot q \cdot r$ as the public modulus. Hence, in order to decrypt we need to find the private exponent, which needs to be $d = N^{-1} \mod \phi(n)$, where $\phi(n) = (p-1)(q-1)(r-1)$.

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)
                n = prod(priv)
                return PrivateKey(priv, pub, n)
            except ValueError:
                pass
```

The constructor does the following:

```
class PrivateKey(PublicKey):

    def __init__(self, priv, pub, n):
        super().__init__(pub, n)
        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:

```
class PublicKey:

    def __init__(self, pub, n):
        self.pub = pub
        self.f = self.magic(pub)
        self.n = n
        self.nb = (self.n.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:

```
        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

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

So, we know that

$$\frac{N}{D} = \frac{p+q}{q+r} + \frac{q+r}{r+p} + \frac{r+p}{q+r} = \frac{P+Q}{Q+R} + \frac{Q+R}{R+P} + \frac{R+P}{Q+R}$$

Rearranging the equations, we get that:

$$\begin{cases} D\big((p+q)^2(r+p) + (q+r)^2(p+q) + (r+p)^2(q+r)\big) = \\ \qquad = N(p+q)(q+r)(r+p) \\ \\ D\big((P+Q)^2(R+P) + (Q+R)^2(P+Q) + (R+P)^2(Q+R)\big) = \\ \qquad = N(P+Q)(Q+R)(R+P) \end{cases}$$

Cubic diophantine equation

As a result, we have that both (p, q, r) and (P, Q, R) are solutions to this equation:

$$D\big((x+y)^2(z+x) + (y+z)^2(x+y) + (z+x)^2(y+z)\big) = N(x+y)($$

The above is a diophantine equation, because we are working with integer coefficients and we are interested in integer solutions. Usually, diophantine equations have more than one solution, so we need to find a way to solve the above to find (p, q, r).

While reading about cubic diophantine equations, I found that if we interpret the above equation as a curve and its genus equals 1, then it can be expressed as an elliptic curve (more information at math.stackexchange.com). Notice that we already have a rational point (P, Q, R), which is a point in the cubic curve, so we can have a point in the elliptic curve using the conversion from the cubic equation to the elliptic curve.

In fact, rational points on curves with genus equal to 1 form an abelian group (see Wikipedia). As a result, if we have a point G on the elliptic curve, then any multiple of G is also a rational point. Therefore, we can use the inverse mapping to get another solution to the cubic diophantine equation.

In SageMath, we can test it:

```
sage: P, Q, R = map(ZZ, pub)
sage: f = (P + Q) / (Q + R) + (Q + R) / (R + P) + (R + P) / (P + Q)
sage: x, y, z = PolynomialRing(QQ, 'x, y, z').gens()
sage: eq = (x + y) / (y + z) + (y + z) / (z + x) + (z + x) / (x + y) - f
sage: eq.subs({x: P, y: Q, z: R})
0
sage: Curve(eq.numerator()).genus()
1
```

So, we can now find the transformation function from the cubic curve (the numerator of the fraction) to the elliptic curve and its inverse:

```
sage: F = EllipticCurve_from_cubic(eq.numerator(), [1, -1, -1])
sage: Fi = F.inverse()
sage: G = F((P, Q, R))
```

Notice that SageMath requires a projection point. We have used $(1, -1, -1)$ because it lies on the curve:

$$D((1-1)^2(-1+1)+(-1-1)^2(1-1)+(-1+1)^2(-1-1))=0$$

```
sage: eq.numerator().subs({x: 1, y: -1, z: -1})
0
```

Now, for any multiple of G on the elliptic curve, we can find a solution to the cubic equation:

```
sage: a, b, c = Fi(2 * G)
sage: eq.subs({x: a, y: b, z: c})
0
sage: a, b, c = Fi(3 * G)
sage: eq.subs({x: a, y: b, z: c})
0
```

However, these solutions are huge:

```
sage: int(a.numerator()).bit_length()
73531
sage: int(a.denominator()).bit_length()
73533
```

As a result, we can think that (p, q, r) must be mapped to some point X on the elliptic curve such that $X = k \cdot G$ for a small k. In SageMath, we can

use `division_points` to find such points if they exist:

```
sage: G.division_points(2)
[(-112168615286123579047093183237978251036116355542865779584753345503481511513151
sage: G.division_points(3)
[]
sage: G.division_points(4)
[]
sage: G.division_points(5)
[]
```

So, we must have $X = 2 \cdot G$:

```
sage: X = G.division_points(2)[0]
sage: a, b, c = Fi(X)
sage: eq.subs({x: a, y: b, z: c})
0
sage: type(a)
<class 'sage.rings.rational.Rational'>
```

Alright, but although we have another solution to the cubic equation, `a`, `b` and `c` are rational numbers. We must find integer solutions. For this, we can simply find a common denominator and multiply each rational number by that:

```
sage: cd = lcm(lcm(a.denominator(), b.denominator()), c.denominator())
sage: a, b, c = ZZ(a * cd), ZZ(b * cd), ZZ(c * cd)
sage: eq.subs({x: a, y: b, z: c})
0
```

The values are now integers, and they are solutions to the cubic equation. But the important thing is that we have found p, q and r:

```
sage: int(a).bit_length()
1024
sage: is_prime(a)
True
sage: int(b).bit_length()
1024
sage: is_prime(b)
True
sage: int(c).bit_length()
1024
sage: is_prime(c)
True
sage: p, q, r = a, b, c
```

RSA decryption

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

$$\begin{cases} s_p = s_1 + s_2 \mod n \\ s_m = s_1 - s_2 \mod n \end{cases} \iff \begin{cases} s_1 = 2^{-1} \cdot (s_p + s_m) \mod n \\ s_2 = 2^{-1} \cdot (s_p - s_m) \mod n \end{cases}$$

Remember the values of s_1 and s_2:

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

At this point, we can have the signatures R and c, and we can check that their bit lengths are correct:

```
sage: n = p * q * r
sage: N = f.numerator()
sage:
```

```
sage: nb = (int(n).bit_length() + 7) // 8
sage:
sage: d = pow(N, -1, (p - 1) * (q - 1) * (r - 1))
sage:
sage: sp = pow(mix[0], d, n)
sage: sm = pow(mix[1], d, n)
sage:
sage: s1 = (sp + sm) * pow(2, -1, n) % n
sage: s2 = (sp - sm) * pow(2, -1, n) % n
sage:
sage: R = pow(s1, N, n)
sage: c = pow(s2, N, n)
sage: assert c.bit_length() <= nb
```

Notice that $R = A \cdot B^{-1} \mod n$, where

$$\frac{A}{B} = \text{magic}(m, h, c) = \frac{m+h}{h+c} + \frac{h+c}{c+m} + \frac{c+m}{m+h}$$

LLL lattice reduction

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 n is around 3072 bits.

However, instead of finding m and h, let's find A and B. We know that both A and B are short compared to n, and it follows that

$$L = \begin{pmatrix} R & n \\ 1 & 0 \end{pmatrix}$$

So that we can use LLL to find a short vector (A, B), which belongs to the lattice because

$$\begin{pmatrix} R & n \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} B \\ -k \end{pmatrix} = \begin{pmatrix} A \\ B \end{pmatrix}$$

```
sage: A, B = Matrix(ZZ, [[R, n], [1, 0]]).transpose().LLL()[0]
sage: assert R == A * pow(B, -1, n) % n
sage: A.bit_length()
1148
sage: B.bit_length()
1022
```

System of multivariate polynomials

At this point, we know that:

$$\frac{A}{B} = \frac{m+h}{h+c} + \frac{h+c}{c+m} + \frac{c+m}{m+h}$$

Which is equivalent to:

$$\begin{cases} g \cdot A = (m+h)^2(c+m) + (h+c)^2(m+h) + (c+m)^2(h+c) \\ g \cdot B = (m+h)(c+m)(h+c) \end{cases}$$

For some positive integer g, probably small.

We might be tempted to try using multivariate Coppersmith method, but the above are 3-degree polynomials, so the variable values are not short enough to find small roots.

However, we can use Groebner basis to find a solution to the system of multivariate non-linear polynomials:

```
sage: m, h = PolynomialRing(QQ, 'm, h').gens()
sage:
sage: g = 1
sage:
sage: p1 = (m + h) ** 2 * (c + m) + (h + c) ** 2 * (m + h) + (c + m) ** 2 * (h +
sage: p2 = (m + h) * (c + m) * (h + c) - g * B
sage:
sage: Ideal([p1, p2]).variety()
[{h: 112505123084813325142525029007183899144558205055924378385155078283235959674
```

And so we find both m and h.

Flag

Here we have the flag:

```
sage: m = Ideal([p1, p2]).variety()[0].get(m)
sage: bytes.fromhex(hex(m)[2:])
b'HTB{3CC_1s_m4g1c___15nt_1t?!}'
```

The full script can be found in here: solve.py .

Contents