Overview

We are given some Sage code for deterministically signing messages, along with a public key and an example of a signature:

```
from hashlib import sha256

def keygen(password):
    while True:
        p = 2*random_prime(2^521) + 1
        if p.is_prime(proof=False):
            break
    base, h = 3, password
    for i in range(256):
        h = sha256(h).digest()
    x = int.from_bytes(h*2, "big")
    return base, p, pow(base, x, p), x


def sign(message, priv):
    h = int(sha256(message).hexdigest(), 16)
    k = next_prime(int.from_bytes(
        sha256(message + priv.to_bytes(128, "big")).digest() + \
        sha256(message).digest(),
        "big"
    ))
    r = int(pow(g,(k-1)/2,p))
    s = int(Zmod((p-1)/2)(-r*priv+h)/k)
    return r, s


g, p, pub, priv = keygen(b"[redacted]")
r, s = sign(b"blockchain-ready deterministic signatures", priv)


'''
----------------------------------------------------------------
sage: p
[...]
sage: pub
[...]
sage: r
[...]
sage: s
[...]
```

```
sage: "midnight{" + str(priv) + "}" == flag
True
'''
```

We are asked to recover the signer's private key.

We can summarize the signature scheme as follows. First, let's discuss key generation. A key consists of a generator g (fixed to be 3), a safe prime p (i.e., of the form $2q+1$, where q is also prime), a private key x, and a public key $g^x \bmod p$ (although one can think of the whole tuple $(g, p, g^x \bmod p)$ as being the public key). In this implementation, x is formed by taking the SHA256 hash of a password, concatenating this with itself, and converting to a 512-bit integer. Importantly, this means that x is divisible by $2^{256}+1$, and so we can write $x = (2^{256}+1)y$ for some 256-bit y. The prime p is chosen to be around 521 bits in size.

Signing a message works as follows. First we compute the SHA256 hash h of the message (interpreting it as a 256-bit integer). We then choose a prime k as follows. We first generate another 256-bit number h' by taking the hash of the message concatenated with our private key. We then let $k_0 = 2^{256}h' + h$, and let $k = \text{nextPrime}(k_0)$. Finally, we return the following two numbers as our signature:

1. r, which is equal to $g^{(k-1)/2} \bmod p$.
2. s, which is equal to $(h - rx)k^{-1} \bmod q$ (recall that $q = (p-1)/2$).

It is not too hard to see how to verify such a signature given the public key (from r we can compute $g^k \bmod p$, and then we can check if $(g^k)^s \equiv g^{(h-rx)} \bmod p$), but understanding this will not be necessary to solve the challenge. I suspect this signature scheme is based off of a real signature scheme, but I don't know its name.

Attack

Our attack will be based off the following three facts:

1. As already mentioned, the private key x is divisible by $(2^{256}+1)$, and thus can be written in the form $x = (2^{256}+1)y$ for a much smaller y.
2. The function $\text{nextPrime}(x)$ is very close to x; in particular, for 512-bit values of x, we can expect $\text{nextPrime}(x) - x$ to be at most 1000.
3. We know the bottom half of the bits of the value k_0 (they are just the hash of our message).

Let's therefore assume we know the value $\Delta = k - k_0 = \text{nextPrime}(k_0) - k_0$. By fact 2 above, we should expect $\Delta \leq 1000$, so we can try brute-forcing over all possible values of Δ until we hit the correct one.

We will now use the form of s to exploit our knowledge of facts 1 and 3. In particular, note that by the definition of s, this means that s satisfies the equation

$$ks + rx \equiv h \bmod q.$$

Now, since $k = (2^{256}h' + h + \Delta)$ and $x = (2^{256} + 1)y$, we can rewrite this as:

$$2^{256}sh' + (2^{256} + 1)ry \equiv h - (h + \Delta)s \bmod q.$$

We know all the variables in the above expression with the exception of h' and y. Furthermore, we know that h' and y are quite small compared to q; even though $q \approx 2^{521}$, both h' and y are at most 2^{256}. This suggests using a lattice reduction attack to recover h' and y. One way of doing this is to approximately solve CVP and find a vector in the lattice generated by the rows of

$$\begin{bmatrix} 2^{256}s \cdot B & 1 & 0 \\ (2^{256}+1)r \cdot B & 0 & 1 \\ q \cdot B & 0 & 0 \end{bmatrix}$$

that is close to the vector

$$((h - (h + \Delta)s)B, 0, 0).$$

Here B is some sufficiently large multiplier to encourage LLL to exactly match the first coordinate (I used $B = 2^{300}$). We can then read off the values of h' and y from the second and third coordinates respectively.

Implementation

Here is my Sage implementation of the above approach. The `Babai_CVP` code is taken from https://github.com/rkm0959.

```
from sage.modules.free_module_integer import IntegerLattice
from hashlib import sha256

p = 4035648853708381789256954324273674914702371551862442121539138986867637108964009710133438
g = 3
pub = 24641222545643177918082419938573295700344069666715233786452270366211300172713154182881
r = 1955692135575340621358830864429181364319679390886478096252939908744046303252388963634166
s = 1569096619843380076500264618255791799360035257909827076210713309748736154483054014253168
q = (p-1)//2

MESSAGE = b"blockchain-ready deterministic signatures"
h = int(sha256(MESSAGE).hexdigest(), 16)

BIG = 2^300
```

```python
def Babai_CVP(mat, target):
    M = IntegerLattice(mat, lll_reduce=True).reduced_basis
    G = M.gram_schmidt()[0]
    diff = target
    for i in reversed(range(G.nrows())):
        diff -=  M[i] * ((diff * G[i]) / (G[i] * G[i])).round()
    return target - diff

def get_key(diff):
    alpha = h + diff

    c_x = int(((2**256) * s) % q)
    c_key = int((r * (2^256 + 1)) % q)
    C = int((h - alpha * s) % q)

    mat = Matrix(ZZ, [[BIG*c_x,     1, 0],
                      [BIG*c_key,  0, 1],
                      [BIG*q,       0, 0]])

    real_target = vector(ZZ, [BIG*C, 0, 0])
    target = vector(ZZ, [BIG*C, 2^255, 2^255])

    ans = Babai_CVP(mat, target)
    print(ans)
    _, found_x, found_key = ans - real_target

    priv = int((2^256 + 1) * found_key)
    if pow(3, priv, p) == pub:
        print("Found actual priv:", priv)
        return True
    return False

def solve():
    diff = 0
    while True:
        print("----", diff)
        if get_key(diff):
            break
        diff += 1

solve()
```