Perfect Secrecy

This is a write-up for the “Perfect Secrecy” challenge in Google CTF 2018 Quals. It was solved by RonXD from The Maccabees team.

The challenge

In this challenge we have a server holding a private RSA key, and clients may communicate with it using a special protocol allowing some information to be leaked about ciphertexts.
The server code is as follows:

```
m0 = reader.read(1)
m1 = reader.read(1)
ciphertext = reader.read(private_key.public_key().key_size // 8)
dice = RsaDecrypt(private_key, ciphertext)
for rounds in range(100):
        p = [m0, m1][dice & 1]
        k = random.randint(0, 2)
        c = (ord(p) + k) % 2
        writer.write(bytes((c,)))
writer.flush()
```

In this piece of code, `reader` is the input stream from the client and `writer` is the output stream to the client.
The server holds a private key used in the protocol, the we get the public counterpart.

The protocol

The protocol is a simple one round protocol:

- The client sends two bytes to the server, m_0 and m_1.
- The client sends an RSA ciphertext.

- The server decrypts the ciphertext using its private key, takes the least significant bit of the plaintext, b, and gets m_b where m_b is one of $\{m_0, m_1\}$ according to b. Then, for 100 iterations, it picks a random number $k \in \{0, 1, 2\}$ and sends the least significant bit of $m_b + k$ to the client. Note that k may be different for each iteration.

Protocol Analysis

It is pretty obvious we have to employ a CCA on the server to extract data about the plaintext. We would, theoretically, want to extract the LSB of a chosen plaintext each round, since this is all the input for the server's response (assuming we don't use timimg analysis which is pretty impractical), but we have a small problem - we don't get the bit directly, but we get 100 bits based on the wanted bit. However, since the number k is uniformly-distributed, we get $P(m_b + k = m_b (\mod 2)) = \frac{2}{3}$, since only $k = 1$ will change the result's parity. So in order to get b we send $m_0 = 0, m_1 = 1$ and extract b according to a majority vote of the returned bits - we have a very small chance to be wrong.

The CCA

From now on, we will assume the correctness of the bits extraction method. If you aren't familiar with RSA I recommand going to its Wikipedia page.
We denote by n the modulus, d the private exponent and e the public exponent. The encryption function is simply $E(x) = x^e$ and the decryption function is $D(x) = x^d$.
Now let's observe the simple fact the for any $c_1, c_2 \in \mathbb{Z}/n\mathbb{Z}$ we have:

$$D(x) \cdot D(y) = x^d \cdot y^d = (x \cdot y)^d = D(x \cdot y)$$

Assume c is a ciphertext (any ciphertext, not just ours) we wish to decrypt. We can use the server to get the parity of $D(c)$. But how do we go on from here?
Let's first assume $D(c)$ is even (the parity bit is 0). Since we have the public key we can get $E(2^{-1})$ (where $2^{-1} \cdot 2 = 1 (\mod n)$). If we look at the number $c' = c \cdot E(2^{-1})$ we get $D(c') = \frac{D(c')}{2}$ since the ring elements should behave the same as the lifted numbers (the numbers in $\mathbb{Z}$ corresponding the elements in $\mathbb{Z}/n\mathbb{Z}$). This is great - we could just use the same technique on c' and recursively get all the bits! But what if $D(c)$ is odd?
This is where some tricks get in. We first observe that n is always odd. So, if $D(c)$ is odd, then

$-D(c)$ must be even (unless $c = 0$ which is very unlikely). But how do we get $-D(c)$? Since the private exponent d is also odd (we don't know d but we know it's parity since it's odd for every RSA key) we know:

$$-D(c) = -x^d = (-x)^d = D(-c)$$

So, now we can go on with $-c$, and just invert our result!

The algorithm

Let's write a function CCA in pesudocode. The function CCA receives two arguments, c and bits, where c is the ciphertext and bits is the amount of bits we want to extract. It uses an oracle $D(x)$ by using the protocol, and a number $I = E(2^{-1})$

```
CCA(c, bits):
        if bits = 0:
                return 0
        b = D(c)
        if b = 0:
                return CCA(c*I, bits-1)*2
        else:
                return n - CCA((n-c)*I, bits-1)*2
```

So this algorithm is a simple recursive algorithm that extracts one bit each iteration. If we wish to decrypt a 1024-bit ciphertext c we just have to call $CCA(c, 1024)$.

And we made it!

After removing garbage around the resulting plaintext we get the flag:
`CTF{h3ll0__17_5_m3_1_w45_w0nd3r1n6_1f_4f73r_4ll_7h353_y34r5_y0u_d_l1k3_70_m337}`
See you around next time!

> Written with [StackEdit](https://stackedit.io/).