Morphing Time

Anakin Dey

Last Edited on 7/8/23 at 14:40

Solution

We are presented with access to an oracle. Once we obtain some public parameters $g = 2, p$ a prime, and $A \equiv g^a \pmod{p}$ where a is a secret value as well as the two part ciphertext for the flag (c_1, c_2), we are given the opportunity to almost decrypt a ciphertext of our choice.

First, what is this encryption scheme? With the previously mentioned public parameters, we encrypt *flag* as follows. Let $k \xleftarrow{\$} \mathbb{Z}_p$ be a randomly sampled integer in $[2, p-1]$. All math is done in $\mathbb{Z}_p$, meaning that everything is done modulo p. Then we have that

$$\textsc{encrypt}(m) = \left(2^k, m \cdot A^k\right) = (c_1, c_2).$$

When we pass in our chosen ciphertext (c_1', c_2') we are given the decryption of $(c_1 \cdot c_1', c_2 \cdot c_2')$, the product of two ciphertexts. These ciphertexts would be generated separately, so we consider the natural question of what does the product of the encryption of two messages look like?

$$\begin{aligned}
\textsc{encrypt}(m_1) \cdot \textsc{encrypt}(m_2) &= \left(2^k, m_1 \cdot A^k\right) \cdot \left(2^{k'}, m_2 \cdot A^{k'}\right) \\
&= \left(2^{k+k'}, m_1 \cdot m_2 \cdot A^{k+k'}\right) \\
&= \textsc{encrypt}(m_1 \cdot m_2).
\end{aligned}$$

This means that $\textsc{encrypt}(m_1) \cdot \textsc{encrypt}(m_2) = \textsc{encrypt}(m_1 \cdot m_2)$ which makes encryption a homomorphism! In fact, it also turns out that decryption is also a homomorphism (*work it out yourself*).

Our solution is going to take advantage of this. We know that $\textsc{encrypt}(\mathit{flag}) = (c_1, c_2)$. Let $\textsc{encrypt}(m) = (c_1', c_2')$ for some m. Then by the homomorphism property we must have that $\textsc{decrypt}(c_1 \cdot c_1', c_2 \cdot c_2') = \mathit{flag} \cdot m$. Thus our strategy is going to be to choose some known plaintext m, encrypt it, and send that encryption as (c_1', c_2'). Now we get our decrypted text $\mathit{flag} \cdot m$ from the oracle. Then since we know m we can compute its modular inverse mod p, which is easily done in Python, and multiply it by $\mathit{flag} \cdot m$ and recover *flag*.

```python
from Crypto.Util.number import long_to_bytes
from random import randint
import pwnlib.tubes

r = pwnlib.tubes.remote.remote('127.0.0.1', 1337)

print("Getting public info...")
r.recvlineS()
r.recvlineS()
g = int(r.recvlineS().split("=")[1])
p = int(r.recvlineS().split("=")[1])
A = int(r.recvlineS().split("=")[1])
```

```python

print("Getting ciphertext...")
r.recvlineS()
c1_ = int(r.recvlineS().split("=")[1])
c2_ = int(r.recvlineS().split("=")[1])

print("Generating ciphertext...")
r.recvlineS()
known = int.from_bytes(b"knownplaintext", "big")
k = randint(2, p - 1)
c1_ = pow(g, k, p)
c2_ = pow(A, k, p)
c2_ = (known * c2_) % p
print("Sending...")
r.sendline(bytes(str(c1_), "utf-8"))
r.sendline(bytes(str(c2_), "utf-8"))

print("Decrypting...")
r.recvlineS()
m = int(r.recvlineS().split("=")[1])
r.close()
plain = pow(known, -1, p)
plain = (m * plain) % p
plain = long_to_bytes(plain)

if b'uiuctf{' not in plain:
    exit(1)
else:
    print(plain)
    # uiuctf{h0m0m0rpi5sms_ar3_v3ry_fun!!11!!11!!}
r.close()

exit(0)
```