PlaidCTF 2017 - FHE

The flag is encrypted with ElGamal and an additional layer of custom fully-homomorphic encryption. We can recover the FHE key under known-message attack by solving a linear system. The group used for ElGamal is weak (its order has small prime factors), so we can compute a discrete logarithm to recover the secret exponent and decrypt the flag.

Description

If you didn't get the memo, homomorphic encryption is the future. But we might have to work out a few bugs first.

Details

Points: 275

Category: Crypto

Validations: 14

A custom encryption scheme

We were given a Sage script describing the custom encryption scheme, along with a public key and ciphertext. Let's first understand the encryption scheme.

Custom FHE scheme

The custom FHE scheme works as follows. Given a prime p, the field $F = GF(p)$ and a dimension n, the secret key consists of a random vector $t \in F^n$. We denote by t' the first $n-1$ elements of t, i.e. $t' = (t_1, \dots, t_{n-1})$.

To encrypt a message $x \in F$, first generate a random matrix $M \in F^{(n-1) \times n}$ and compute the vector $v = t' \cdot M$. Then, the ciphertext matrix $C \in F^{n \times n}$ consists of the $n-1$ rows of M followed by the row $C_n = \frac{1}{t_n}(xt - v)$.

To decrypt it, first compute $w = t \cdot C$. By construction, this is equal to xt, so the plaintext x is equal to w_i / t_i for any index i.

ElGamal

The flag is encrypted as follows. We recognize ElGamal encryption in F, with all values encoded with the FHE scheme.

```
# Generate key
x = F.random_element()

g = F.multiplicative_generator()
G = fhe.encrypt(g)
H = G^x

z = F.random_element()
Z = fhe.encrypt(z)

# Public key
pubkey = (G, H, Z)

# Encrypt flag
m = int(FLAG.encode('hex'), 16)
M = fhe.encrypt(m)

y = F.random_element()
U = G^y
S = H^y

# Ciphertext
enc = (U, (M + Z) * S)
```

We are given two files containing the public key (G, H, Z) and the ciphertext (U, C).

Solution

We first break the FHE scheme and then ElGamal.

Known-message attack against custom FHE

We first note that knowing any non-zero multiple αt of the key vector is sufficient to decrypt messages (we compute αw instead of w and obtain $x = \alpha w_i / \alpha t_i$). In particular, we consider $\tau = t / t_n$ (such that $\tau_n = 1$) and aim at recovering τ.

We note that the last ciphertext row C_n is equal to $x\tau - \tau' \cdot M$. If we denote by C'_n and M' the first $n-1$ columns of respectively C_n and M, we obtain the following linear relation

$$C'_n = \tau' \cdot (xI_{n-1} - M')$$

If we know a plaintext-ciphertext pair such that $xI_{n-1} - M'$ is inversible (i.e. x is not an eigenvalue of M'), then we can solve this equation and recover the key τ.

However, we are given only 5 ciphertexts G, H, Z, U, C and do not know the corresponding plaintexts... But we know that the scheme is homomorphic, and that plaintexts are elements of the field F. In particular, for any $X = FHE(x)$, we know that $X^{p-1} = FHE(x^{p-1}) = FHE(1)$ because the multiplicative group of F has order $p-1$. This gives us the known plaintext-ciphertext pair that we need!

With the given values, it turns out that G^{p-1} does not work because the matrix is not inversible, but Z^{p-1} does the trick and we can recover the secret key τ. This Sage script recovers the key and decrypts the ciphertexts.

```
g = 19
h = 5277408455979627998693284545457492434625481971838358031947
z = 7480661922343631805748801975792334938833170910952610904826
u = 1612025406697104180305741723503884445079332594011737736370
c = 9958176780800074629262127222466629024851019704901560112644
```

Weak group for ElGamal

The challenge now consists of solving the ElGamal problem over the field $F = GF(p)$. This amounts to finding the secret exponent x such that $h = g^x$ (discrete logarithm problem). It turns out that the chosen prime p (of 337 bits) is weak because $p - 1$ has small factors:

```
sage: factor(P - 1)
2^8 * 3 * 5^2 * 7^3 * 13^3 * 17 * 23 * 41 * 191 * 727 * 2389 *
```

In particular, we can use the Pohlig-Hellman algorithm to compute the discrete log. Sage has a built-in `discrete_log` function but it used more than 4GB of RAM before we aborted. We wrote our own implementation of Pohlig-Hellman in the following script.

For the biggest prime factor $p_{max} = 695890117602047$, we split the look-up table of the baby-step giant-step algorithm into two passes, each fitting into 4GB of RAM. Indeed, this last prime requires a look-up table of $\sqrt{p_{max}} = 26.3M$ entries, each containing at least a group element of 337 bits (without taking into account the overhead of a hash table in Python).

Once x is recovered, we can decrypt the flag as $m = cu^{-x} - z$.

```
x = 1314262303709987066847071804553079487827695870600429138999
PCTF{eigen_see_a_valuable_flag_here}
```

Written on April 24, 2017