0CTF 2019

- zer0mi
  - zer0mi write-up

zer0mi

zer0mi write-up

zer0mi

DISCLAIMER: Linear algebra heavy write-up

Intro

As in task description I searched for Matsumoto-Imai, the first thing in google is this paper: link In there we find explanations of how does the system work and how to break it.

Lets call G = GF(256), that means Galois field with 256 elements. So in output file there is array of length n = 63 containing elements from G[x_1, ..., x_n], let us call elements (f_1, ..., f_n) and that's the public key. To encode a value one takes blocks of n bytes, maps bytes to G and count values for each f_i. We also have 63 bytes in hex, that's ciphertext.

Brief explanation

We have find vector space of all such A (matrix n x n with values from G) that foreach X : ($X * A * F^T = 0$) for X = (x1, ..., xn), F = (f1, ..., fn). (($X * A * F^T$) is matrix 1 x 1 with its value being one polynomial of degree 3 over field G and variables (x_1, ..., x_n)). In other words ($X * A * F^T$) is a zero polynomial.

Then we swap F with our cipher text (converted to vector of bytes) and get some new equations for X

(it may not be not clear now, but it will make sense).

A bit longer explanation

Finding vector space.
Lets say A consists of elements $a_{i,j}$ (from G of course). Then we count ($X * A * F^T$). (If you read the paper and wonder why we don't need extra y and x variables equal to one, it's because in encrypt.py l_1 and l_2 are linear and not only affine.) The result will be combination of terms of following type: ($x_i * x_j * x_k * a_{l,m}$).

It must be zero polynomial so coefficients for each ($x_i * x_j * x_k$) must be zero. But each coefficient is combination of $a_{l,m}$ so we get equation for $a_{l,m}$ for each term ($x_i * x_j * x_k$). Lets say ($n * n * n$) equations, just some of them will be ($0 = 0$).

So we solve it and we should get exactly n free variables (cause math, won't explain and after some thought I'm not entirely sure why). But this gives us equations that's describes vector space of possible A.
Lets call free $a_{l.m}$ b_k. So we convert all $a_{l,m}$ to combination of the free ones and put it back to ($X\ A\ F^T$).

We also change F from public key to ciphertext Y. That's because we know ($y_i = f_i(flag_1, ..., flag_n)$).
The product must be zero (because we defined that it's equal to zero for general form of Y, so it'll be zero for specific).
After multiplying it all we get combination of terms of following type ($x_i * b_j$).

Lets look at it as polynomial from $G[b_1, ..., b_k]$. It must be a zero polynomial so all coefficients must be zero.

So we write equation for each coefficient and get system of equation for ($flag_1, ..., flag_n$).

We solve it and as far as I tested there's one free value in this system. We count result for each possible value and we get flag as one of 256 possibilities. (could also assume that $flag_1$ = 'f' or sth like that).

Solution

For starters I wrote python script to change public key format. (file extract.py. it's an abomination but it works, so no need to upgrade). Output is in format: n matrixes n x n with values divided by spaces, and then n bytes of ciphertext also in numbers. By numbers I mean integers, cause there is easy bijection from G to (0, 1, ..., 255).

Then it's time to code the solver (in c++ cause speed is needed). The file is solve.cpp, also don't forget -O3 if you wanna compile.

First 40 lines are includes and implementation on G (from now called GF). Addition in GF is just a simple xor on number representation. For multiplication I counted all products in python and then copied it to const array cause it's easier and faster :P

Then lest go to main. First three loops are self-explanatory (in third there are two fake flags for n = {7, 25} used for sanity check). We count inverses array, read matrixes and read Y.

Then there is:

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int a = 0; a < N; a++) {
            for (int b = 0; b < N; b++) {
                int it[3];
                it[0] = i;
                it[1] = a;
                it[2] = b;
                sort(it, it+3);                int equationNum = it[0] + N * (it[1] + N *
(it[2]));
```

```
                int nrA = i + N * j;
                GF var = matrixes[j][a][b];                equations[equationNum][nrA] +=
var;
            }
        }
    }
}
```

Lets explain. Let A, B, C be matrixes of size (1 x n), (n x n) and (n x 1). Product (A * B * C) is:

```
                  [b_1,1, ..., b_1,n]   [c_1](a_1, ..., a_n) * [....., ..., .....] * [...] = Sum
(a_i * b_i,j * c_j) for i, j in 1, ..., n
                  [b_n,1, ..., b_n,n]   [c_n]
```

Simple.
So $(X * A * F^T)$ is (Sum $x_i * a_{i,j} * f_j$).

Also (f_j is Sum x_a * matrix[j][a][b] * x_b). I count nrA of $a_{i.j}$ as (i + n * j) so it's from (0, 1, ..., n * n - 1).

I also give each term ($x_i * x_a * x_b$) unique equationNum from (0, 1, ..., n * n * n - 1). (see that numbers for $x_i * x_a * x_b$ and $x_a * x_b * x_i$ etc. are the same).
And then I add value of matrix[j][a][b] to equation with number equationNr on nrA-th position, cause variable is a a_{nrA} and the equation is derived from term equationNr.

So now I have equations describing out vector space to find.

Then there is:

```
line:153 some code removed
for (int i = 0; i < eq_count; i++) {
    auto &row = equations[i];
    int pos = 0;
    updateFirst(row, pos);
```

```
    while(pos < stairCount && stairs[pos].second) {        addVect(row,
mulVect(stairs[pos].first, row[pos]));
        updateFirst(row, pos);
    }

    if (pos < stairCount ) {
        stairs[pos].first= mulVect(row, inverse[row[pos].v]);        stairs[pos].second =
true;
        full++;
    }
    else {
        empty++;
    }
}
```

I'm counting echelon form for this system of equations. Basicly I have array stairs denoting valid stairs. Then I take equations one by one and:

1. if there isn't a stair with this number of leading spaces I add it to stair array
2. if it's zero then disregard
3. otherwise increase number of leading zeros by subtracting appropriate stair and go back to 1.

Next important part is in inside if (line 218) and corresponding else from line 229:

```
# FROM IF
stairs[i].first[i] = GF(0);
# FROM ELSE
stairs[i].second = emptyStairs - 1000;
emptyStairs++;
stairs[i].first[i]  = GF(1);
```

From equation for bound variables I delete part corresponding for the variable itself, so only the combination of free variables is left. So now all equations are just description of variable in free variables.

For free variables I enumerate them (such variables that their stair is equal to zero). (- 1000) is to keep number lower than zero.

Last nontrivial part of code is:

```
for (int i = 0; i < stairCount; i++) {
    int x = i % N;
    int y = i / N;
    assert(x + N * y == i && x < N && y < N && x >= 0 && y >= 0);

    auto tmp = mulVect(stairs[i].first, Y[y]);
    // assert(tmp.size() == stairCount);

    for (int j = 0; j < stairCount; j++) {        assert(tmp[j] == GF(0) || stairs[j].second <
0);
        if (tmp[j] != GF(0)) {            assert(stairs[j].second + 1000 >= 0 &&
stairs[j].second + 1000 < emptyStairs);            result[stairs[j].second + 1000][x] +=
tmp[j];
        }
    }
}
```

Here I construct equations for $flag_i$. So I count ($X * A * Y^T = Sum\ x_i * a_{i,j} * y_i$).

```
    int x = i % N;
    int y = i / N;
```

is counting $a_{i,j}$ indices (from now x,y is i,j).

$a_{x,y}$ = stairs[i].first so I multiply it by Y[y]. (that stair is combination of free variables equal to $a_{x,y}$)
Then for each term in equation:

1. if it's zero do nothing,
2. if it's nonzero it means that's the free value in combination. Also that means one of result terms is $x_x * b_{stairs[j].second+1000} * y_i$. So I it add to result equation with number

stairs[j].second+1000 (cause that's the number of free variable and each result equation is derived from single free variable) and on x-th position (x as index, and because the result variable is x_x).

Finally we have system of equations, solve the system of equation, count all solutions (here 256), and somewhere there will be the flag.

Curiosities

1. Encrypting script works more than 15min, my decryptor less than 2min on my machine.
2. Due to it being late night (or more like early morning) I spent couple hours working on wrong code because there was

```
typedef uint8_t byte;
```

and then cin read bytes as characters and not numbers.