Solving andrewl.us SDDecoder Junior

Numernia
crackmes.de

Nov 24, 2009

1 Brief overview

This crackme is all about the bits, as you probably seen, this crackme is not about reading or analyzing Assembler code, the crackme is given as a .cpp file, the whole scheme is reviewed. The main thing in this crackme is the maths.

2 Analysis

If you check the main.cpp you will directly see alot of constants, we will get to those later on. Also you will see the core function f_evaluate, transform, verify_parity and main, off course. This text will briefly described all of them.

2.1 Main

Main function, first part of it is just parsing the string to a 64bit integer. Then it calls the important code(indeed andrewl also wrote this in the source :)):

```
UINT64 y;

y = transform(x,L1);
y = f_evaluate(y);
y = transform(y,L2);

if(verify_parity(y))
{
    if((y & 0xFFFFF) > 30) // low 20 bits are the id of the key
        printf("accepted!");
    else
        printf("blacklisted!");
}
```

As you see the scheme is very clear and those three core functions called one after each other.

2.2 Transform

```
UINT64 transform(UINT64 x, UINT64 v[64])
{
    UINT64 result=0;

    for(INT i=0; i<64; ++i)
    {
        UINT64 a = x & v[i];
```

```
        UINT64 b = 0;
        while(a)
        {
            b ^= (a & 1);
            a /= 2;
        }
        result |= (b << (63-i));
    }

    return result;
}
```

This code looks quite wierd, but its actually quite simply when you see it, somethings makes it "harder to understand", for example instead of >> 1, /2 is used, but they are doing the same thing. So if you change `a /= 2;` to `a >>= 1;`. Maybe a bit clearer, it performs first a AND operation with the input and current table value. After this it "counts" the bits to b, that is XOR's all bits in a. Then b(which is either 0 or 1) gets OR'd to the result(return) number.

If you look closer, the bit counting is actually a addition over GF(2), addition is XOR since if you add the coffiencents and mod 2, this will be the same. So for example 1+1+0+1 = 1. So to reverse this, you need to put up a equation system. Due to the AND, the table values can be seen as position determiners(actually those values whome decids how the equation look like), since the AND will only take those values from X. So to reverse this you need to convert result(Y) value to binary and put up equation like this

$$\begin{pmatrix} x_0v_0 + x_1v_0 & \cdots & x_{63}v_0 \\ x_0v_1 + x_1v_1 & \cdots & x_{63}v_1 \\ \vdots & \ddots & \vdots \\ x_0v_{63} + x_1v_{63} & \cdots & x_{63}v_{63} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{63} \end{pmatrix}$$

Each variable is a bit from each number, also remember everything is calculated over GF(2). So to solve this system we have to apply Gaussian Elimination over GF(2), check the keygenerator source code, its simple.

2.3 f_evaluate

This procedure is the most complex(I think atleast), and I couldn't figure out what it was doing, so I thought of bruteforcing it, since if you watch it:

```
UINT64 f_evaluate(UINT64 x)
{
    UINT64 y = 0;
    INT cursor = 0;
    for(INT i=1; i<=64; ++i)
        for(INT j=i; j<=64; ++j)
        {
            if((x & ((UINT64)1<<(64-i))) && (x & ((UINT64)1<<(64-j))))
                y ^= lsvec[cursor];

            cursor++;
        }
    return y;
}
```

You can check and test that each 16bit gives specific 16bit as result. For example

```
f_evaluate(0x12345678ABCD1122) = 0x0463ACFBFE9C1D5A
```

```
f_evaluate(0x12345678ABCD0000) = 0x0463ACFBFE9C0F4B
f_evaluate(0x1234567800000000) = 0x0463ACFBF1455E53
f_evaluate(0x1234000000000000) = 0x0463A1C146C41EC9
```

So we can bruteforce this function, however it takes some seconds to find a value(and sometimes it dont exist). This bruteforce is taking the main computation time.

2.4 verify_parity

```
INT verify_parity(UINT64 x)
{
    UINT64 sig = x & 0xFFFFF;
    UINT64 parity = (x & 0xFFFFFFFFFFF00000) >> 20;
    UINT64 result = 0;

    for(int i=0; i<20; ++i)
        if(sig & ((UINT64)1 << i))
            result ^= (parity << i);

    for(int i=0; i<24; i++)
        if(result & ((UINT64)1 << (63-i)))
            result ^= (0xBD36E46C30800000 >> i);

    return result == 1;
}
```

So first it extracts the first 20 bits from X, load to sig, then the rest is extracted. The loop under is just repeated addition over GF(2), so multiplication, it compute `result = parity * sig`

The second loops is quite tricky, but after a bit thinking and blackbox analysis trial and error I figured it was a modulo in GF(2) by 0x17A6DC8D861. Then it checks if this value is == 1. So check is:

$$if((parity * sig)(mod\ 0x17A6DC8D861) == 1) \tag{1}$$

However, since its all in GF(2) makes it a bit trickier. To reverse this you need to first find a value so:

$$0x17A6DC8D861 * R + 1(mod\ sign) == 0 \tag{2}$$

3 Writing the keygenerator

- 1. Take in some sign variable, S.
- 2. Find a value K_0 = (0x17A6DC8D861 * R + 1) that sign divides equal.
- 3. Integrate S and K_0 0xAAAAAAAAAAABBBBB, A is K_0 and S is B. = K_0
- 4. Solve the L2 equation system for value $\mathrm{K}_0, transform^{-1}(\mathrm{K}_0) = K_1$
- 5. Bruteforce the f_evaluate value: f_evaluate_brute(K_1) = K_2, make sure there is a proper value, if not found, no serial for that signature exist.
- 6. Solve the L1 equation system for value K_2, transform^{-1}(K_2) = K_3
- 7. K_3 is the serial number.

4 End

4.1 Example serials

```
0x0BABE 5A8BBB95DCBF4F13
0xC0DEE E41DD4309293ED57
0xABCD0 8CD3EC780F1994AD
0x04E55 585BD974D0787B59
```

4.2 Thx

Thanks alot to andrewl.us for this crackme, and to all my other friends.

4.3 Information

A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2) A. Bogdanov, M.C. Mertens, C. Paar, J. Pelzl, A. Rupp