May
01

Google CTF – Jekyll (Crypto)

- Writeups

by hellman

> Can you access the admin page? You can look at the crypto here.
>
> source.py

Summary: finding a preimage for a simple 64-bit ARX-based hash.

Here’s the code of the web server:

```
def jekyll32(data, seed):
  def mix(a, b, c):
    a &= 0xFFFFFFFF; b &= 0xFFFFFFFF; c &= 0xFFFFFFFF;

    a -= b+c; a &= 0xFFFFFFFF; a ^= c >> 13
    b -= c+a; b &= 0xFFFFFFFF; b ^=(a <<  8)&0xFFFFFFFF

    c -= a+b; c &= 0xFFFFFFFF; c ^= b >> 13
    a -= b+c; a &= 0xFFFFFFFF; a ^= c >> 12
    b -= c+a; b &= 0xFFFFFFFF; b ^=(a << 16)&0xFFFFFFFF
    c -= a+b; c &= 0xFFFFFFFF; c ^= b >>  5
    a -= b+c; a &= 0xFFFFFFFF; a ^= c >>  3
    b -= c+a; b &= 0xFFFFFFFF; b ^=(a << 10)&0xFFFFFFFF
    c -= a+b; c &= 0xFFFFFFFF; c ^= b >> 15

    return a, b, c
```

```
    a = 0x9e3779b9
    b = a
    c = seed
    length = len(data)

    keylen = length
    while keylen >= 12:
        values = struct.unpack('<3I', data[:12])
        a += values[0]
        b += values[1]
        c += values[2]

        a, b, c = mix(a, b, c)
        keylen -= 12
        data = data[12:]

    c += length

    data += '\x00' * (12-len(data))
    values = struct.unpack('<3I', data)

    a += values[0]
    b += values[1]
    c += values[2]

    a, b, c = mix(a, b, c)

    return c

def jekyll(data):
    return jekyll32(data, 0x60061e) | (jekyll32(data, 0x900913) << 32)

...
cookie = self.request.cookies.get('admin')
if cookie is not None and jekyll(base64.b64decode(cookie)) == 0x203b1b70cb122e29:
    self.response.write('Hello admin!\n'+FLAG)
else:
    self.response.write('Who are you?')
...
```

So we need to find preimage of 203b1b70cb122e29 with hash described by the jekyll function, which simply concatenates two calls to jekyll32 with different seeds.

The core of jekyll32 is the mix function. It takes three 32-bit workds and transforms them using ARX operations. Note that mix is easily invertible if we have all three values. However the jekyll32 function returns only the third value.

The message is processed in blocks of 12 bytes and is padded with at least one zero. Let' s see what we can do with one block. The hash then works like this:

$$jekyll32(m_1||m_2||m_3, seed) = \\ mix((\text{9e3779b9}, \text{9e3779b9}, seed + length) + (m_1, m_2, m_3)).$$

We can set some random values to the outputs a, b, and invert the mix function. Then, we subtract the initial constants and deduce a message which results in the given triple a, b, c, where c is equal to the 32-bit half of the hash. Now we can change the seed and compute the

hash and check if it matches the other half. That is, we need 2^{33} evaluations of the mix function.

However, there is a problem: at least one zero byte is added, so with one block we can control only 11 bytes. That is, when we invert the mix function, we don't control the least significant byte of the third word, which need to be equal to $seed + length$. Thus, we have to try 2^8 times more. It is still doable, but takes quite a lot of time.

Let's instead consider messages with two blocks. We won't care about the second block, we will use only the fact that the first block is fully controlled by us. So we can actually let the second block be the zero pad. And the general scheme stays the same.

To sum up the attack:

- let h_1, h_2 be 32-bit halves of the target hash;
- choose random a, b;
- compute $t = mix^{-1}(a, b, h_1)$;
- subtract $length = 12$;
- compute $s = mix^{-1}(t - 12)$;
- deduce $m = s - (\mathrm{9e3779b9}, \mathrm{9e3779b9}, seed1)$;
- check if $jekyll32(m, seed_2) == h_2$.

We will have to repeat this around 2^{32} times, each time we do 4 evaluations of mix or mix^{-1}.

Here's C++ code:

```
#include <bits/stdc++.h>
// g++ brute.cpp -O3 -std=c++11 -o brute && time ./brute

struct State {
  uint32_t a, b, c;
  void mix() {
    a -= b+c;
    a ^= c >> 13;
    b -= c+a;
    b ^= a << 8;
    c -= a+b;
    c ^= b >> 13;
    a -= b+c;
    a ^= c >> 12;
    b -= c+a;
    b ^= a << 16;
    c -= a+b;
    c ^= b >> 5;
    a -= b+c;
    a ^= c >> 3;
    b -= c+a;
    b ^= a << 10;
    c -= a+b;
    c ^= b >> 15;
  }
  void unmix() {
    c ^= b >> 15;
    c += a+b;
    b ^= a << 10;
```

```
    b += c+a;
    a ^= c >> 3;
    a += b+c;
    c ^= b >> 5;
    c += a+b;
    b ^= a << 16;
    b += c+a;
    a ^= c >> 12;
    a += b+c;
    c ^= b >> 13;
    c += a+b;
    b ^= a << 8;
    b += c+a;
    a ^= c >> 13;
    a += b+c;
  }
};

uint32_t STARTCONST = 0x9e3779b9;
uint32_t LENGTH = 12;
uint32_t SEED1 = 0x60061e;
uint32_t SEED2 = 0x900913;
uint32_t HASH1 = 0xcb122e29;
uint32_t HASH2 = 0x203b1b70;

int main() {
  for(uint64_t a = 0; a < 1ll << 32; a++) {
    if ((a & 0xffffff) == 0) {
      printf("%08x\n", a);
    }
    State s = {a, 0x31337, HASH1};
    s.unmix();
    s.c -= LENGTH;
    // subtract message, but we set it to zeroes
    // so do nothing
    s.unmix();

    uint32_t p[3];
    p[0] = s.a - STARTCONST;
    p[1] = s.b - STARTCONST;
    p[2] = s.c - SEED1;
    s.a = p[0] + STARTCONST;
    s.b = p[1] + STARTCONST;
    s.c = p[2] + SEED2;
    s.mix();
    s.c += LENGTH;
    s.mix();

    if (s.c == HASH2) {
      printf("GOOD: %08x %08x %08x\n", p[0], p[1], p[2]);
      printf("PLAIN: ");
      for(int i = 0; i < 8; i++)
        printf("%02x", (char*)p + i);
      printf("\n");
    }
  }
  return 0;
}
```

```
GOOD: 5cc80e2e e7fee109 d6d486f1
PLAIN: 2e0ec85c09e1fee7f186d4d6
2m2.185s
```

The flag: CTF{diD_y0u_ruN_iT_0N_Y0uR_l4PtoP?}

Tags: 2016, arx, c++, crypto, ctf, google, hash, python, writeup

Leave a Reply

Your email address will not be published.

Message:

You may use these HTML tags and attributes: <a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <i> <q cite=""> <s> <strike>

Name:

Email:

Website:

Submit Comment

Submit Comment

Archives

- March 2017 (3)
- October 2016 (6)
- September 2016 (3)
- May 2016 (5)
- April 2016 (2)
- March 2016 (5)
- September 2015 (2)
- May 2015 (4)
- April 2014 (4)
- March 2014 (1)
- February 2014 (4)
- January 2014 (2)
- December 2013 (1)
- October 2013 (1)