Breaking a legacy NSA backdoor

stf

<2022-07-02 Sat>

The device

Previously on camp++

- `https://pretalx.hsbp.org/camppp7e5/talk/HGP9HS`
- `https://www.youtube.com/watch?v=8VTmfiifkRU`
- `https://hsbp.s3.eu-central-1.amazonaws.com/camppp7e5/backdoor.mp4`

Elsewhere

- `https://www.alchemistowl.org/pocorgtfo/pocorgtfo21.pdf` 21:12
- `https://github.com/stef/px1000cr`
- `https://www.ctrlc.hu/~stef/blog/posts/pocorgtfo_21_12_apocrypha.html`

The attack

We're building a full key recovery attack based on only the ciphertext.

- Algebra! We are building a whole lot of equations and then
- feed the equations to Z3 to solve them.
- Super simple, if you know how. ;)

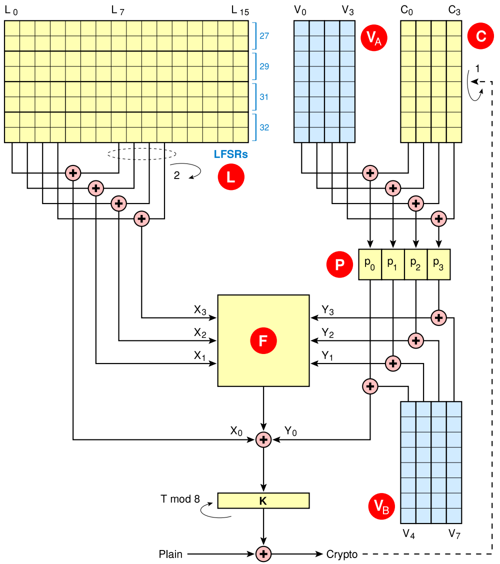
The Tool

We use claripy from the angr project which makes it easier to work with bits and Z3.

The Key

- The key is entered as 16 ASCII characters
- only the lower 4 bits of each character are use.
- 64-bit key, stronger than DES!

The Schema



(c) cryptomuseum

"Mirror Low Nibble"

Before we get into the components, there's an important function used during initialization of most blocks:

```
unsigned char invertLoNibble2Hi(unsigned char x) {
  return ((~x) << 4) | x;
}
```

V and C initialization

```
for(i=0;i<4;i++) {
  V[i]   = invertLoNibble2Hi(key[i]   ^ key[i+4]);
  V[i+4] = invertLoNibble2Hi(key[i+8] ^ key[i+12]);
  C[i] = V[i] ^ V[i+4] ^ 0xf0;
}
```

C[i] can only be one of these 16 values after initialization:

```
{0x0f, 0x1e, 0x2d, 0x3c, 0x4b, 0x5a, 0x69, 0x78,
 0x87, 0x96, 0xa5, 0xb4, 0xc3, 0xd2, 0xe1, 0xf0}
```

Strange!

V_a and C combination through P S-box

```
for(i=0;i<4;i++) {
  tmp = V[i] ^ CiphertextFifo[i];
  acc = map4to4bit[i][tmp >> 4] << 4;
  acc |= map4to4bit[i][tmp & 0xf];
  pbuf[i] = acc ^ V[i+4];
}
```

Another anomaly

For the first plaintext byte

`tmp = V[0] ^ CiphertextFifo[0]`

where

`CiphertextFifo[0] = V[0] ^ V[4] ^ 0xf0`

which drops out V[0] and thus:

`tmp == V[4] ^ 0xf0`

and we know that all values of V are values where the high nibble is just the inversion of the low nibble, and if we xor that with 0xf0, we get that tmp can only be one of these 16 values:

```
{0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff}
```

WTF?

Anomaly cont'd

recap:

```
for(i=0;i<4;i++) {
  tmp = V[i] ^ CiphertextFifo[i];
  acc = map4to4bit[i][tmp >> 4] << 4;
  acc |= map4to4bit[i][tmp & 0xf];
  pbuf[i] = acc ^ V[i+4];
}
```

resulting also acc being of type 0xYY
Later the ciphertext fifo is filled with ciphertext and loses this strange structure.

LFSRs initialization

```
for(i=0;i<15;i++) {
  lfsr[i] = invertLoNibble2Hi(key[i]);
}
lfsr[15]=0xff;
```

Note: if you know the internal state of the LFSRs at any point in time, you can rewind the state until you get a state were all bytes have mirrored nibbles and the last byte if 0xff. This is a very cheap operation! This recovers 60 out of the 64 key bits.

LFSRs update

```
for(round_counter = 31;round_counter>=0;round_counter--) {
  acc = 0;
  for(i=0;i<16;i++) { // in the FW this loop is unrolled
    acc ^= lfsr[i] & lookupTable[(round_counter+i) & 0xf];
  }

  acc = ((acc >> 1) ^ acc) & 0x55;

  // tmp is twice the sequence 15..0
  tmp=(round_counter ^ 0xff) & 0xf;
  lfsr[tmp] = ((lfsr[tmp] << 1) & 0xAA) | acc;
}
```

LFSRs extract

```
for(i=0;i<4;i++) {
  tmp = lfsr[i+7];
  // rotate tmp left by 2 bits
  tmp = (tmp << 2) | (tmp >> 6);
  lfsr_out[i] = tmp ^ lfsr[i];
}
```

The non-linear mapping F

maps six bytes to one byte.

```
for(i=8, acc=0;i>0;i--) {
  for(j=1,tmp=0;j<4;j++) {
    tmp = (tmp << 1) | (lfsr_out[j] >> 7);
    lfsr_out[j]<<=1;
    tmp = (tmp << 1) | (pbuf[j] >> 7);
    pbuf[j]<<=1;
  }
  tmp=lookupTable6To1bit[tmp];

  acc=(acc<<1) + ((tmp>>(i-1)) & 1);
}
```

the inner loop interleaves $lfsr_{out}[1]$, pbuf[1], ... $lfsr_{out}[3]$, pbuf[3]
The lookuptable is 64 bytes, indexed by this 6 bit interleaved value, and taken the i^{th} bit as a result, very compact and efficient.

The keystream output

acc is the output from the non-linear mapping F:

```
acc ^= pbuf[0] ^ lfsr_out[0];

tmp = (curChar + 1) & 7;
// rotate acc left by tmp
acc = (acc << tmp) | (acc >> (8-tmp));

ciphertext[curChar] = plaintext[curChar] ^ acc
```

Updating the ciphertext FIFO

```
CiphertextFifo[4] = ciphertext[curChar];
for(i=0;i<4;i++) {
  // CiphertextFifo[i] = rot_left(CiphertextFifo[i+1]
  CiphertextFifo[i] = (CiphertextFifo[i+1] << 1) |
                      (CiphertextFifo[i+1] >> 7);
}
```

So far so good

- Most of the simple things are already algebra.
- Except for the LFSRs, non-linear mapping F and the P S-box.
- Let's start with the mappings

```python
def moebius(f,n):
    blocksize=1
    for step in range(1,n+1):
        source=0
        while(source < (1<<n)):
            target = source + blocksize
            for i in range(blocksize):
                f[target+i]^=f[source+i]
            source+=2*blocksize
        blocksize*=2
```

- a.k.a ANF transform, Zhegalkin transform, Positive Polarity Reed-Muller transform.
- converts a lookup table into another lookup table

The F lookup table

```
static unsigned char lookupTable6To1bit[64]={
    0x96, 0x4b, 0x65, 0x3a, 0xac, 0x6c, 0x53, 0x74,
    0x78, 0xa5, 0x47, 0xb2, 0x4d, 0xa6, 0x59, 0x5a,
    0x8d, 0x56, 0x2b, 0xc3, 0x71, 0xd2, 0x66, 0x3c,
    0x1d, 0xc9, 0x93, 0x2e, 0xa9, 0x72, 0x17, 0xb1,
    0xb4, 0xe4, 0xa3, 0x4e, 0x27, 0x5c, 0x8b, 0xc5,
    0xe8, 0x95, 0xe1, 0xd1, 0x87, 0xb8, 0x1e, 0xca,
    0x1b, 0x63, 0xd8, 0x2d, 0xd4, 0x9a, 0x99, 0x36,
    0x8e, 0xc6, 0x69, 0xe2, 0x39, 0x35, 0x6a, 0x9c
};
```

Applying the ANF transform for bits 0..7

f_0= 011000100110101010111000111010..
g_0= 011001010000111110110111010010..
f_1= 110100100011010101110110001101..
g_1= 101110001001111011001001110010..
. . .
f_7= 100010000101010010010100011010..
g_7= 111100001011000100011011001100..

The algebraic normal form of F

this new lookup table g can be converted into multivariate polynomial over F_2

$$f(x_0,\ldots,x_{n-1}) = \bigoplus_{(a_0,\ldots,a_{n-1})\in\mathbb{F}_2^n} g(a_0,\ldots,a_{n-1}) \prod_i x_i^{a_i}$$

In python:

```
' ^ '.join(f'{c}')'
  for c in ['&'.join(
                f"x{i}"
                for i, x in enumerate(reversed(f'{a:06b}'))
                if x == "1")
              for a in range(64)
              if moebius[a]=='1']
  if c)
```

The ANF of f_4 - TADA! Algebra!

1 ^ (x0) ^ (x1) ^ (x2) ^ (x0&x2) ^ (x4) ^ (x1&x4) ^ (x0&x1&x4) ^
(x1&x2&x4) ^ (x3&x4) ^ (x0&x1&x3&x4) ^ (x0&x2&x3&x4) ^
(x0&x1&x2&x3&x4) ^ (x0&x1&x5) ^ (x0&x2&x5) ^ (x1&x2&x5)
^ (x3&x5) ^ (x1&x3&x5) ^ (x0&x1&x3&x5) ^ (x2&x3&x5) ^
(x4&x5) ^ (x2&x4&x5) ^ (x0&x2&x4&x5) ^ (x3&x4&x5) ^
(x0&x3&x4&x5) ^ (x1&x3&x4&x5) ^ (x1&x2&x3&x4&x5)

The P transform

- The P transform is a 4 bit to 4 bit S-box
- We can reduce this problem to the F solution
- Simply decompose the 4-to-4 mapping into 4 times 4-to-1 bit mappings
- TADA! Algebra!

The LFSRs I

the update of the LFSRs happens like this:

```
acc ^= lfsr[i] & lookupTable[(round_counter+i) & 0xf];
```

The LFSRs II

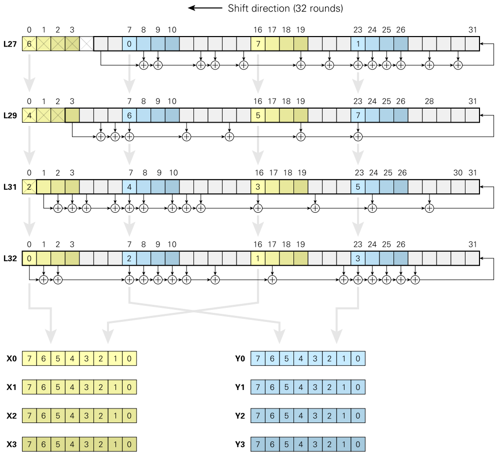
It turns out the lookup table contains the taps in a bit-sliced representation. They can be recovered with this python snippet:

```python
taps = (
   0x06, 0x0B, 0x0A, 0x78, 0x0C, 0xE0, 0x29, 0x7B,
   0xCF, 0xC3, 0x4B, 0x2B, 0xCC, 0x82, 0x60, 0x80)

def extract(taps, i):
    # left to right
    tr = [''.join(str((taps[j] >> b) & 1)
            for j in range(16))
            for b in range(8)]
    # horizontal bottoms-up lines appended
    return (tr[(i*2)+1]+tr[(i*2)])

for i in range(4):
  print(extract(taps, i))
```

The LFSRs III



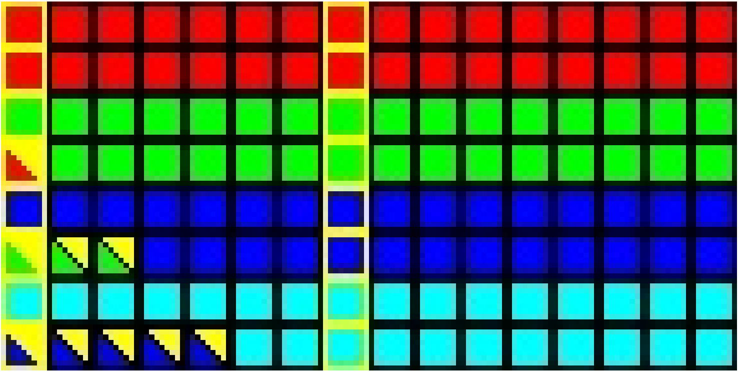
(c) cryptomuseum

LFSR VI

In another representation the so-called taps are as following:

```
32 bit: 11100001111101000100001111110000  e1f443f0
31 bit: 01111011101110001000100010001000  7bb88888
29 bit: 00010111000100100001000100000000  17121100
27 bit: 00000100110011010001010111101010  04cd15ea
```

The LFSRs VII



The algebraization of the LFSRs

- totally ignore that these are LFSRs
- angr & symbolic execution to the rescue
- consider the value of the 16 bytes of the LFSRs as symbolic
- run the update loop in symbolic execution
- dump the symbolic value of the output state of the LFSRs block

Claripy constraints for the 127th bit after the LFSR update loop

```
<BV128 lfsr_state[87:87] ^ lfsr_state[63:63] ^
       lfsr_state[55:55] ^ lfsr_state[31:31] ^
       lfsr_state[23:23] ^ lfsr_state[7:7] ^
       lfsr_state[102:102] ^ lfsr_ state[86:86] ^
       lfsr_state[70:70] ^ lfsr_state[62:62] ^
       lfsr_state[54:54] ^ lfsr_state[46:46] ^
       lfsr_state[30:30] ^ lfsr_state[14:14] ..
```

cleaned up:

$ls_{(127,i+1)} = ls_{7,i} \text{ ^ } ls_{14,i} \text{ ^ } ls_{23,i} \text{ ^ } ls_{30,i} \text{ ^ } ls_{31,i} \text{ ^ } ls_{46,i} \text{ ^ } ls_{54,i} \text{ ^ } ls_{55,i}$
$\text{ ^ } ls_{62,i} \text{ ^ } ls_{63,i} \text{ ^ } ls_{70,i} \text{ ^ } ls_{86,i} \text{ ^ } ls_{87,i} \text{ ^ } ls_{102,i}$

TADA! Algebra!

Handing it over to Z3

- creating the equations takes "significant" time, about 40-something seconds! But these can be reused!
- dumped out as ASCII they take about 22 MB
- passing it 17 bytes of ciphertext and solving it for the 64 bits of the key takes less than 4 seconds on this laptop.
- if n ciphertext bytes less than 17 are provided then $2^{(17\text{-}n)}$ key candidates are the result, each key candidate can be tested if decryption results in meaningful results.

Other attacks

- Unsure if the NSA had a SMT solver like z3 back in the 80ies.
- What they certainly had were correlation attacks.
- It seems reasonable that most of Armknechts[1] work was known by the NSA back then.
- Detecting key-reuse is trivial due to 16 + (message length - 1) keystream bits leaking in every message.
- consulted Willi Meier, besides Armknecht **the** other LFSR expert, he said this:

  The boolean functions look involved. Quite striking that you found an attack. For the moment, I don't see the trapdoor[sic] behind.

[1] http://madoc.bib.uni-mannheim.de/1352/1/Armknecht.pdf

Conclusion

- full key-recovery in less than 4 seconds with 17 bytes of ciphertext
- px1000Cr is catastrophically weaker than DES
- multiple backdoors working together

Thanks

Thanks to ben, phr3ak, the Crypto Museum people, jonathan, antoine, Valentin, the angr devs, asciimoo and dnet for their support!

Questions

. . . or silence and staring? :)