Description of the NSA manipulated SBT algorithm

stf

January 9, 2024

Contents

1 Description of the NSA manipulated SBT algorithm

1.1 Abstract

We reverse-engineered and analyzed the ROM of a cryptographic device manipulated by the NSA. The following paper summarizes the findings and gives a preliminary analysis. For more information about this device see the page in the cryptomuseum:

`https://www.cryptomuseum.com/crypto/philips/ua8295/sbt.htm`

This is a draft of a work in progress project.

1.2 Diffing the original and the manipulated ROM

1. The original ROM contained a DES implementation.
2. A diff with the manipulated ROM shows the region 0x1c5d-0x2181 to be changed.

3. The code between 0x1aed - 0x2181 has been entirely decompiled by hand.

   A lot of other code has also been hand decompiled, but is irrelevant for this topic and discussion of this has been omitted from this document.

4. One change is a modification of a constant, this seems to be a checksum function.

```
char maybe_checksum(char *param_1,char param_2) {
  char cVar1 = 0x12; // changed from 0xab
  if ((char)((ushort)param_1 >> 8) == 0x40) {
    return 0x40;
  }
  do {
    cVar1 += *param_1++;
  } while (param_2 != (char)((ushort)param_1 >> 8));
  return cVar1 + -1;
}
```

   This explains the claim by cryptomuseum that the checksum of the original ROM is equal to the checksum of the NSA manipulated ROM.

5. Another change is a one byte change in an unused data area.

   Speculation: this might be to force the colliding checksum with the original ROM.

6. The diff also shows a few small changes outside the main changed area, all these are call-sites into the main changed area.

   The targets of these call sites are:

   - 5 times: 1c6f (*sbt_init*())
   - 7 times: 1cf5 (*crypt_byte*())
   - once 1d13 (*sbt_short*())

1.3 Generic algorithm overview

1. The algorithm is a stream-cipher construction.

2. The Key is a 15 character string of values for characters A-Z, 0-9 and ',', '.', '-', space.

3. The key has a total entropy of ~80 bits.

   $\log_2(40^{15}) = 79.82892142331043$

4. The plaintext contains only characters A-Z,0-9 and ',', '.', '-', space and '_' used only for padding.

   It is possible that also newline is an allowed character. This must be verified, but doesn't influence anything significantly.

5. The algorithms I/O

   (a) The sbt() function takes 2 input parameters: 8B state, 7B message key.

   (b) The sbt() function outputs 2 results: an updated 8B state, and an 8B cipherblock.

   (c) The sbt() function updates the state by applying an LFSR for 64 steps: $(1 + x^{32} + x^{63})$.

6. Initialization

   (a) A 15 bit (with 12 bits of entropy) nonce is passed as a parameter. This means that a birthday collision of the key stream will have a 50% chance after just 64 messages with the same key.

   15 bits from 12 bits of entropy are the result of taking 3 4bit values and incremented by 1, which thus can be stored on 5 bits, but those 5 bits can only take 16 distinct values (1-16)

   (b) To initialize the cipher, the key is split into 2 parts.

   (c) The first 8 characters of the key are used for state.

   (d) The 3x5 bits of the nonce are xored with the low bits of the first 3 bytes of the state.

   (e) The last 7 characters of the key are used for the message key.

   (f) The core sbt() function is called with the state and the message key.

   (g) The first 7 bytes of the output of the sbt() function is used as the derived message key.

   (h) The state is set to a fixed value of { 0xf5, 0xc0, 0x7a, 0x10, 0x8a, 0xaf, 0x17, 0xcf }

   (i) sbt() is called a second time generating the first 8 bytes of the cipherstream.

(j) The state last 3 bytes are overwritten with 3x5 bits of the nonce with the 3 highest bits of the overwritten bytes set to 0.

7. Setting a key

   (a) The initialization is called with a nonce with all bits set to zero

   (b) The verification check group is calculated based on the value of the 8B cipherblock
   The code calculating and converting the output to ASCII looks as following:

```
static void key_id(const uint8_t *cipherblock) {
  printf("key_id: ");
  uint8_t *cb=cipherblock; // alias
  for(int i=7;i>0;i-=2) {
    putchar((((cb[i]>>4 ^ cb[i-1]) & 0xf) | 0x40) + 1);
  }
  putchar('\n');
}
```

   It is known, that a key consisting of 15 times the letter A - using the manipulated algorithm -, generates a verification code of JJDI.

8. A hard-coded default key exists, it is 2V58RGKLXN49TKD
   This key was also present in the original non-manipulated ROM.

9. Encrypting a string

   (a) Generate nonce by reading the 8bit timer TL0 5 times and xoring this with the previous encryption first five cipherstream bytes, keeping only the low nibbles.
   The following code shows the important details:

```
volatile uint8_t TL0; // 8b Timer 0 at SFR:0x89
// nonce is uninitialized assumed to be 0 after reset
// nonce is overwritten after init with cipherblock[:5]
static uint8_t nonce[5]={0};

for(int i = 0; i<5; i++) {
  uint8_t tmp = (nonce[i] ^ TL0) & 0xf;
  nonce[i] = tmp + 1;
}
```

Note, that this is simplified. In the ROM each generated "character" is also displayed on the device display and optionally also sent to the printer, which takes more cycles.

Hypothesis: according to nonces generated by a real device, it seems that if the printer is not enabled, and no interrupt is handled, then TL0 increases by 15 in each iteration.

Question, if the printer is enabled, how does TL0 increase in each iteration?

All in all it can be said that TL0 is a very bad random source.

(b) Initialize the derived message key using the "random" nonce generated in the previous step.

(c) Copy the first 5B of the initial cipherblock to the nonce.

This step can have two explanations:

- it is to improve the entropy of the next nonce
- it is to leak the first 5 lower nibbles of the initial cipherblock

The second explanation only makes sense if it is possible to reliably guess the delta of TL0 when generating successive values for the nonce.

(d) Iteratively xor 3 bytes of plaintext with the cipherblock.

(e) Expand the lower 6b of each ciphertext triple into 5 x 6b creating 5 letter groups.

This step discards the top 2b of each ciphertext byte, since the original plaintext only used the lower 6b anyway.

These 5 letter groups are also displayed on the device and optionally also sent to the printer.

(f) Whenever a cipherblock is consumed, genereate 8 more cipher stream bytes by calling sbt() with the state and the derived message key.

10. Block operation

```
for i in 0..4:
    nonce[i] = ((nonce[i] ^ TL0) & 0xf) + 1
state, cipherblock = sbt(key[0:8] ^ nonce[0:3],key[8:15])
message_key = cipherblock[0:7]
state0 = [0xf5, 0xc0, 0x7a, 0x10, 0x8a, 0xaf, 0x17, 0xcf]
state, cipherblock = sbt(state0, message_key)
# note state[0:5] is fully known, it is state0
# run through the LFSR 64 times
state = state[0:5] + nonce[0:3]
# since nonce is also fully known
# the full state from here on is known
nonce = cipherblock[0:5]
for each plaintextblock:
    # state is only modified using
    # 64 steps of the (1 + x^32 + x^63) LFSR
    state, cipherblock = sbt(state, message_key)
    ciphertextblock = cipherblock ^ plaintextblock
```

The following gives a quick overview of all the steps involved when running the sbt() function:

Input: state, message key

(a) state = lfsr(64), output state
(b) cipherblock = $fix\_bit\_permutation(state)$
(c) iterate 8 times
    i. rotate right both 28b halves of the message key by [5, 2, 2, 5, 5, 5, 2, 2] bits
    ii. $ctrl\_words[i : 0..3] = state[3-i] \oplus msgkey\_bits(bits...)$ (for the concrete bits, see details below)
    iii. rotate state right by one byte
    iv. for each nibble of the cipherblock, modify it based on *ctrl_words* & cipherblock
    v. permute cipherblock bytes, new order: 3 4 6 7 2 1 5
    vi. nibble swap cipherblock bytes if bits 26, 54, 1, 29, 4, 32, 7, 35 of message key are set
    vii. SBOX each cipherblock nibble

Output: cipherblock, state from step 1

For details about each of these steps, see the following section.

1.4 Core algorithm - $sbt(state, message\_key)$

The core algorithm is well described on the original cryptomuseum page at: `https://www.cryptomuseum.com/crypto/philips/ua8295/sbt.htm`

1. The state is updated by applying an LFSR for 64 steps: $(1+x^{32}+x^{63})$

   For the first cipherblock the initial state is 0xf5c07a108aaf17cf which is shifted to be 0xbfb7b6ef1a8c5090.

2. Cipherblock is initialized to the updated state

3. Cipherblock bits are permuted

   New order of bits:

   ```
   35, 10, 30, 57, 2, 55, 40, 20
   29, 0, 62, 15, 52, 34, 43, 23
   58, 28, 12, 46, 4, 19, 53, 38
   42, 31, 6, 36, 9, 48, 21, 60
   18, 51, 32, 1, 41, 56, 26, 13
   50, 37, 25, 45, 8, 17, 59, 5
   47, 61, 11, 27, 33, 54, 7, 22
   49, 3, 14, 63, 24, 16, 39, 44
   ```

   Bits are indexed from left to right, from 0 .. 63.

   For the first cipherblock, with the known fixed state of 0xf5c07a108aaf17cf, the result of this permutation is: 0xead04d72c73b23ed.

   For the basis of this analysis, see the file checkbitperm.py.

4. The following steps are iterated 8 times

   (a) Rotation of message key.

   The message key is split into two 28 bit chunks, and both of the chunks are rotated right by either 5 or 2 bits each round. The number of bits rotated for each of the eight rounds is: 5, 2, 2, 5, 5, 5, 2, 2.

   See the following illustration for the bits of the 7 bytes of the message key, each character is a bit, the number at the start of the line shows by how many bits the halves are rotated to the right, in each of the rounds. The first line is the original key that is rotated in the first round into the second line, thus the original order of the message key is only used in the last round.

```
  abcdefgh ijklmnop qrstuvwx yzAB | CDEF GHIJKLMN OPQRSTUV WXYZ0123
5 xyzABabc defghijk lmnopqrs tuvw | Z012 3CDEFGHI JKLMNOPQ RSTUVWXY
2 vwxyzABa bcdefghi jklmnopq rstu | XYZ0 123CDEFG HIJKLMNO PQRSTUVW
2 tuvwxyzA Babcdefg hijklmno pqrs | VWXY Z0123CDE FGHIJKLM NOPQRSTU
5 opqrstuv wxyzABab cdefghij klmn | QRST UVWXYZ01 23CDEFGH IJKLMNOP
5 jklmnopq rstuvwxy zABabcde fghi | LMNO PQRSTUVW XYZ0123C DEFGHIJK
5 efghijkl mnopqrst uvwxyzAB abcd | GHIJ KLMNOPQR STUVWXYZ 0123CDEF
2 cdefghij klmnopqr stuvwxyz ABab | EFGH IJKLMNOP QRSTUVWX YZ0123CD
2 abcdefgh ijklmnop qrstuvwx yzAB | CDEF GHIJKLMN OPQRSTUV WXYZ0123
```

After 8 rounds the message key is rotated into its original state.

For the basis of this analysis, see the file ctrlwords.py and updctrlvar.py.

(b) Create 4 box permutation control bytes by xoring the first 4 bytes of the state in reverse order with certain bits of the message key.

```
ctrl_words[0] = state[3] ^ msgkey_bits(10, 38, 13, 41, 16, 44, 19, 47)
ctrl_words[1] = state[2] ^ msgkey_bits(22, 50, 25, 53, 0, 28, 3, 31)
ctrl_words[2] = state[1] ^ msgkey_bits(6, 34, 9, 37, 12, 40, 15, 43)
ctrl_words[3] = state[0] ^ msgkey_bits(18, 46, 21, 49, 24, 52, 27, 55)
```

Bits are indexed from left to right, 0..55.

(c) Rotate the state by 1 byte right.

(d) Box permutation of cipherblock - defined by the 4 control bytes created previously in step (b).

This part is manipulating each nibble of the cipherblock in order, from left to right, it operates on nibbles (4 bit) and quads (2 bit) values.

Let's denote C the vector of 16 nibbles of the cipherblock. Then the current nibble is $n = C_i, i \in \{0..16\}$. Furthermore let's denote the high quad of a nibble by n_{hi} and the low quad n_{lo}.

Depending on

- the current control quad (generated in step (b) above), and
- the value (either 0 or 3) of one of the two quads of the current nibble,

the current nibble is either

- changed by a certain value (one of $\pm$ 1 or $\pm$ 4) or
- a second nibble is selected from C by xoring one bit in the index i: $n' = C_{i \oplus s}, s \in 8, 2, 4, 1$. The high and low quad of this second nibble are added to the quad of the current nibble

that is not used for deciding whether to do this. The quad used for deciding to do this is inverted.

cbtq	quad	2nd nibble	n_{hi}	n_{lo}	else
0	$n_{hi}==0$	$n'=C_{i \oplus 8}$	$=3$	$+n'_{lo}+n'_{hi}$	-4
1	$n_{lo}==0$	$n'=C_{i \oplus 2}$	$+n'_{lo}+n'_{hi}$	$=3$	-1
2	$n_{hi}==3$	$n'=C_{i \oplus 4}$	$=0$	$+n'_{lo}+n'_{hi}$	+4
3	$n_{lo}==3$	$n'=C_{i \oplus 1}$	$+n'_{lo}+n'_{hi}$	$=0$	+1

Let's see an example: the control byte top quad (cbtq) is 0, the index (i) of the current nibble is 5 and the value of the current nibble is 1 ($C_5 = 1$). Then cbqt selects the first row in the table above. Since the current nibble is 1, the high quad is all zeros, and thus we have to get the 2nd nibble, whose index is 13 ($5 \oplus 8$), let's say it's value is 9 ($C_{13} = 9$). This means the high quad of the output nibble will be 3, and the bottom quad will be $1(n_{lo}) + 2(n'_{hi}) + 1(n'_{lo}) = 4 mod 4 = 0$. Combining all this, the nibble C_5 will change from 1 to 12.

For the basis of this analysis, see the file boxperm-visu.py

(e) Cipherblock fix byte permutation, new order of bytes: 3, 4, 6, 7, 2, 1, 5, 0

(f) Cipherblock Nibble switch - controlled by derived message key
Depending on the bits 26, 54, 1, 29, 4, 32, 7, 35 message key, the according byte of the cipherblock has its nibbles swapped.

(g) Each nibble of the cipherblock is mapped with their own SBox
There is 16 4x4 SBoxes in total.

1.5 Test Vectors

The fine people of the cryptomuseum generously generated a few test vectors:

```
HELLO WORLD
(truncated) BAMLK GAFFJ EDBCP
(recovered) BAMLK GAFFJ EOBCP EEIMP CDAFP
NAVEL PLUIS
(truncated) HCAMM MBKJO DGCNC
(truncated) EGLLA IKMHG PHHJL
LEEIC BITKY DBCFE DPKLB ECILE
JDMNJ REJHP CPBLO JPJDD BEDNG
```

Unfortunately some of them have been truncated during transcription, but knowing the plaintext and the nonce, and having a working emulation of the original rom and our reverse-engineered re-implementation we were able to recover the missing 5 letter groups.

1.6 Analysis

1. Key-strength

   Although the "master" key is 15 characters from an alphabet of 40 signs and thus ~80 bits of entropy, after the initialization only 56 bits of entropy are used as a message key for the generation of the first cipherblock. Thus on the surface it looks like the strength is 80 bits, in reality it is reduced to 56 bits immediately.

   This means an attacker might want to recover the message key which will be useful to decrypt other messages using the same nonce and key.

2. Randomness

   The first message after rebooting uses only 12 bits of the the 8bit timer TL0 for generating a nonce, this is very predictable and very low quality. All subsequent messages have nonces that are using the 8bit timer xored with the low nibbles of the first 5 bytes of the previous key stream.

3. Nonce Reuse

   Since the nonce is effectively only 12b, collisions can happen very often due to the birthday paradox. With 64 messages the chances for a collision are 50%.

4. Information leakage

   After reboot, the very first message leaks the delta of TL0 between the 5 iterations generating the nonce. This delta is depending on whether the nonce is also sent to the printer, or only the display, another factor that might affect this is any interrupts occuring during the generation of the nonce.

   Starting with the second message the first 5 lower nibbles of the initial cipherblock of the previous message leaks - althought this is xored with the 8b timer TL0, which seems to have a very predictable delta, which can be learned by looking at the first message after a reboot.

5. Algebraic attack

   An algebraic attack without any extra information is not efficient, in our tests a bruteforce attack was significantly faster. It might be possible that an algebraic attack taking using the leakage of the initial cipherblock in the following messages nonce might improve, this needs to be tested.

6. Bruteforce attack

   A naive bruteforce attack against 4 out of 7 bytes of the derived message key completes in about 70 minutes and finds 16328869 candidate plaintexts for a the encrypted string "HELLO WORLD_". Interestingly some candidate plaintexts occur multiple times for example "WBMJN6W6V2QR" is generated using the derived message keys:

   - 51914e1b5951f5
   - 51914e1f5951f5
   - 11914e1f5951f5

   Hypothesis: This is due to the top 2 bits of each cipherblock byte being discarded in this character encoding scheme. And thus it seems plausible that each candidate plaintext can be generated using 4 different derived message keys.

7. Known and limited state

   After the generation of the derived message key the state is initialized to a known value of

   ```
   [0xf5, 0xc0, 0x7a, 0x10, 0x8a, 0xaf, 0x17, 0xcf]
   ```

   It is curious that this fixed state is being "LFSRed" and used for the generation of the first cipherblock, and only aftewards are the last 3 bytes of this "LFSRed" state overwritten by the first three bytes of the nonce. Why not use the nonce also for the first cipherblock? Could it be that there is some pre-computation look-up table that can somehow be used for this fixed state value?

   All later rounds modified the state only by running it 64 times per block through the LFSR. Since the 3 bytes of the nonce each have their top nibble set to all-zero, this means that for the second cipherblock there is only 4096 different starting points in the LFSR.

It is also notable, that since the inputs to the sbt() function are only the fixed message key and the trivially computable state, this function can be very efficiently parallelized, and any arbitrary cipherblock can be independently computed without computing all preceeding cipherblocks.

8. Message key analysis

   The message key is derived from the 15 character master key by running it through the sbt() algorithm once. It is 56 bits long. The message key:

   - is split in half and each half is rotated by 5 or 2 bits to the right.
   - then it is used in the during the 8 rounds of the sbt() algorithm to create the 32 bit control word for the box permutation.
   - and finally 8 bits of it are also used for the nibble switch operation.

   We run a symbolic simulation of these 8 rounds to see which bits of the message key are (un)used, and how often.

   The following shows the 56 bit message key, with each bit assigned its own unique symbol (a-zA-Z0-3):

   ```
   abcdefgh ijklmnop qrstuvwx yzABCDEF GHIJKLMN OPQRSTUV WXYZ0123
   ```

   In the following figure the first four lines show which bits of the message key were used for the four box perm control words in exactly this order, and the last line in this figure shows which message key bits were used for the nibble swap operation.

   The first column in each row shows the bits that are unused for the specific value. All bits that are used twice are highlighted with red.

   Box permutation control words (rounds in row, words in column):

   ```
   2A fHiKlNoQ dFgIjLmO bDeGhJkM y0B3cEfH tVwYz1aC oQrTuWxZ mOpRsUvX kMnPqStV
   Mk rTuWxZA2 pRsUvXy0 nPqStVwY iKlNoQrT dFgIjLmO A2bDeGhJ y0B3cEfH wYz1aCdF
   Yw bDeGhJkM B3cEfHiK z1aCdFgI uWxZA2bD pRsUvXy0 kMnPqStV iKlNoQrT gIjLmOpR
   Ig nPqStVwY lNoQrTuW jLmOpRsU eGhJkMnP B3cEfHiK wYz1aCdF uWxZA2bD sUvXy0B3
   ```

   Nibble swap control words for each round:

   ```
   Qo vXy0B3cE tVwYz1aC rTuWxZA2 mOpRsUvX hJkMnPqS cEfHiKlN aCdFgIjL A2bDeGhJ
   ```

   It is interesting to see, how the repeat bits occur always in the same position, and also how the same bits occur in exactly the same order in each line.

It is also notable, that the repeats happen always in pairs, or rather quads, which is very relevant for the box permutation. Although these control words are all xored with the state, this state is known.

The quads that are missing from one line, are double in the line above it.

For the basis of this analysis, see the file *msgkey_ana.py*.

9. Scrambling pads ciphertext to multiple of 3

   Due to the expansion from 3 chars to the 5 letter group, the very last two characters in a message might be the "_" character which is used for padding. Introducing a known plaintext crib.