Introduction to Cryptology
Generic attacks on SuffixMAC

2024-03

Grading

This TP is graded as the *contrôle continu* of this course. You must send a written report (in a portable format) **detailing** your answers to the questions, and the corresponding source code, *including all tests*, **with compilation and execution instructions** by Friday April 5, 18:00 (2024-04-05T18:00+0200) to:

`pierre.karpman@univ-grenoble-alpes.fr`.

Working in teams of two is allowed but not mandatory. In that case only a single report must be sent, with the two team members clearly identified.

❧

The use of dynamic-analysis *sanitizers* (through the options `-fsanitize=address` and `-fsanitize=undefined`) is strongly encouraged during development phase.

The use of compiler optimizations (through the option `-O3`) is strongly encouraged when running the more expensive attacks.

Using "artificial intelligence" software at any point during this work is strictly forbidden.

Apart from the standard C library, you are ***not*** allowed to rely on any external software or library functions.

1 The SuffixMAC `smht48`

The SuffixMAC construction is a generic transformation of a hash function into a MAC. Informally, given a hash function $\mathcal{H}$, the associated SuffixMAC $\mathcal{M}$ is defined as:

$$\mathcal{M}(k, m) = \mathcal{H}(m||k)$$

where k (resp. m) is the key (resp. message) of $\mathcal{M}$ and $\cdot||\cdot$ denotes string concatenation.

In this lab, we will use a toy SuffixMAC `smht48` based on a toy "narrow-pipe Merkle-Damgård" hash function `ht48`. The hash function is already implemented and available at `https://membres-ljk.imag.fr/Pierre.Karpman/ht48_2.tar.bz2`, but you need to implement `smht48` yourself.

Q.1: Implement the function `smht48` of following signature and specifications:

```
/*
 * Input k: a 48-bit key stored as an array of 6 bytes
 * Input blen: the byte length of m, stored on 64 bits
 * Input m: the message to be hashed, whose length is required to be an
↪  integral number of bytes
 * Input h: placeholder for the 48-bit resulting tag, to be stored as an
↪  array of 6 bytes. Must have been allocated by the caller.
 * Output: void, h is overwritten with the result ht48(m||k)
 * Warning: the key bytes in k must be appended _in order_ (k[0] first)
 */
void smht48(const uint8_t k[static 6], uint64_t blen, const uint8_t
↪  m[blen], uint8_t h[static 6]);
```

Q.2: Verify your implementation of `smht48` on the *test vectors* below. You may use the (already provided) `printhash` function to print the value of the tag on standard output.

1. Key value: `{0, 1, 2, 3, 4, 5}`
   Message value: `{9, 8, 7, 6, 5, 4}`
   Tag value: EE75794547B8
2. Key value: `{0xE4, 0x16, 0x9F, 0x12, 0xD3, 0xBA}`
   Message value: `{9, 8, 7, 6, 5, 4}`
   Tag value: 5F265B72B5EC

2 Exhaustive search for a low-weight key

We now wish to find a key `k` such that for the message value `{9, 8, 7, 6, 5, 4}`, one has a tag value 7D1DEFA0B8AD. By chance, we are aware of the useful fact that (one such possible) `k` only has a (bit) weight of 7 (that is, it has exactly 7 bits set to one).[*]

Q.3:

1. Implement a function `keyrec` to search for `k`.
2. What value(s) did you find for `k`?

Advice: A reasonably-well-implemented version of this attack takes about 100 seconds to run on an average laptop. You may first validate the correctness & efficiency of your implementation by searching for a key that you know, possibly of a smaller weight.

Q.4:

1. Explain how a *key-recovery* attack such as this one can be used as a preliminary step for a universal forgery attack.
2. Is the converse always possible? That is: does a universal forgery attack always lead to a key-recovery attack?

---

[*] This kind of information could possibly be learned from a *side-channel* physical attack, but assuming that keys are sampled uniformly, we would still be lucky to have one of weight only 7!

3 Existential forgeries from collisions

The design of SuffixMAC and the fact that `smht48` is based on the narrow-pipe Merkle-Damgård hash function `ht48` allows to use collisions for the latter to obtain existential forgeries for the former. In more details, let the compression function used in `ht48` be the function `tcz48_dm` of signature:

```
/*
 * Input m: a 128-bit message block stored as an array of 16 bytes
 * Input h: a 48-bit chaining value stored as an array of 6 bytes
 * Output: void, h is overwritten with the result
 */
void tcz48_dm(const uint8_t m[static 16], uint8_t h[static 6]);
```

and `IV` denote the initial value used in `ht48` (given in `ht48.h`). Then if the 16-byte messages `m1` and `m2` are such that the values computed by `tcz48_dm(m1, IV)` and `tcz48_dm(m2, IV)` are the same, one has that for all key `k`, the tags computed by `smht48(k, 16, m1, h)` and `smht48(k, 16, m2, h)` are the same.

Q.5:

1. Explain why the above is true.
2. How can this property be used in an existential forgery attack for `smht48`?

Q.6:

1. Implement a function `colsearch` that computes a collision of the above form for the (already implemented) `tcz48_dm` compression function.
2. Implement a function `smht48ef` that draws a 48-bit key `k` uniformly at random and that uses the collision in `tcz48_dm` to obtain a collision in `smht48` of the above form.

ADVICE: A reasonably-well-implemented version of the collision search takes about 4 seconds to run on an average laptop. A possible strategy is to use an efficient “search” data structure, that can for instance be implemented with an *ad hoc* hash-table of 2^{24} buckets.