Generic attacks on SuffixMAC

Andrei Ilin MOSIG M1

April 2024

Directory overview and execution

Build is done via CMake. To build all targets, from root project directory write in terminal:

```
mkdir build
cd build
cmake ..
cmake --build .
```

This will create 3 executable files:

- mac
- keyrec
- colsearch

Each executable is a demonstration of a particular feature implemented during the lab. To each executable there exists a corresponding .c file named main_*feature name*.c.

As the project is relatively small, I decided to write both declarations and definitions of my functions in *.h*-files.

Code in the report is adapted from the source code of the lab, debug printing and collection of metadata are omitted.

1 The SuffixMAC smht48

SuffixMAC function is declared and defined in *smht*48.*h*.

It takes key k, message m and writes their concatenation in buffer *conc* from which then computes the tag and writes it to h.

```
void smht48(const uint8_t k[static 6], uint64_t blen,
        const uint8_t m[blen], uint8_t h[static 6])
{
        const uint8_t conc[6 + blen];
        memcpy((void*)conc, (void*)m, blen);
        memcpy((void*)conc + blen, (void*)k, 6);
        ht48(6 + blen, conc, h);
}
```

Function *smht*48.*h* is tested on proposed examples in *main_mac.c*.

2 Exhaustive search for a low-weight key

2.1 Combinations

The goal is to efficiently enumerate over all bit vectors of size N that have exactly K bits set to 1.

We will call a *combination* object an integer vector of size K where each number in vector represents the position of a bit which is set to 1. For example, for $N = 5, K = 3$ combination object $[0, 2, 3]$ corresponds to the bit vector 10110.

To make a bijectional relation between bit vectors and combination objects we also have to restrict the numbers in combination object vector to be in strictly increasing order.

To iterate through all combinations we define functions *init* and *next*.

```
void combination_init(uint64_t n, uint64_t k, uint64_t comb[k])
{
        for (uint64_t i = 0; i < k; i++) {
                comb[i] = i;
        }
}
```

Initial combination is corresponding to all 1-bits being in the beginning of the bit vector.

The *combination_next* function generates the next combination given the current combination.

```
int combination_next(uint64_t n, uint64_t k, uint64_t comb[k])
{
        comb[k - 1]++;
        int i = 0;
        if (comb[k - 1] >= n) {
                while (i < k && comb[k - 1 - i] >= n - i - 1) i++;
                if (i == k) return 0;
                int reset_val = comb[k - 1 - i];
                for (int j = 0; j <= i; j++) {
                        comb[k - 1 - i + j] = reset_val + 1 + j;
                }
        }
        return 1;
}
```

Here, we are always increasing the last element in combination vector which corresponds to moving the rightmost "1" bit vector one step right. Once we have reached the border, we increase the rightest possible elements, and repeat the movement from new position.

Example: this is the enumeration of combinations $\binom{5}{3}$ with corresponfing bit vectors.

```
0 1 2 - 11100
0 1 3 - 11010
0 1 4 - 11001
0 2 3 - 10110
0 2 4 - 10101
0 3 4 - 10011
1 2 3 - 01110
1 2 4 - 01101
1 3 4 - 01011
2 3 4 - 00111
```

To convert combination into the bit vector *combination_bitstring* is used (defined in *combination.h*).

2.2 Key recovery

Using the notion of combinations it becomes easy to implement the key recovery function. In our case $N = 48$ and $K = 7$. We iterate through all combinations converting each to bit vector which is to be passed into *smht*48.

```
void keyrec(uint64_t blen, const uint8_t m[blen],
        uint8_t h[static 6], uint8_t key[static 6])
{
        uint64_t n = 48;
        uint64_t k = 7;
        uint64_t comb[k];
        uint8_t result[6];

        combination_init(n, k, comb);
        while (combination_next(n, k, comb)) {
                combination_bitstring(k, comb, 6, key);
                smht48(key, 6, m, result);
                if (memcmp(result, h, 6) == 0) return;
        }
}
```

2.3 Key recovery result

For message $\{9, 8, 7, 6, 5, 4\}$ and tag $7D1DEFA0B8AD$ following key was found:

```
001618009000
or in binary format (zeros represented by underscores):
-------- ---1-11- ---11--- -------- 1--1---- --------
```

2.4 Universal forgery attack

Once the key is recovered, we can sign any messages with it using $smht48$ which implementation is known which is the universal forgery.

2.5 Does a universal forgery attack always lead to a key-recovery attack?

If the adversary has infinite resources, the answer is yes, because knowing the algorithms to sign a message, it is possible to verify if a key is valid or not by encrypting all possible messages (of fixed size) and comparing the result of encryption with the result of forgery algorithm that it has.

Thus, doing so for every possible key can reveal the valid key.

However it is computationally expensive, so not all adversaries can are capable of performing this operation. So, in general case the answer is no.

3 Existential forgeries from collisions

3.1 tcz48_dm property

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.

This is true because $ht48$ which is used in $smht48$ is splitting the message into 16-bit chunks and after applies $tcz48\_dm$ to each of them. The important detail here is that in $smht48$ the message comes before the key in concatenated string which is passed to $ht48$. This is why in case of 16-bit message, first the whole message is passed through $tcz48\_dm$ and then the key. So for $m1$ and $m2$ the first result of chaining is the same and so is the second as they are concatenated with the same key.

3.2 Existential forgery attack for smht48

If adversary knows that message $m1$ of length 16 is signed by some tag h, it can pick some $m2$ s.t. $tcz48\_dm(m1, IV) = tcz48\_dm(m2, IV)$ and because of the property given above $m2$ is also signed by h. So the adversary is able to produce a valid pair of message and tag which is an existential forgery.

3.3 Collisions in tcz48_dm

3.3.1 Hash table

Hash table is implemented in **hashset.h**.

To provide an efficient data structure for searching hash table is implemented. The goal is to map $uint8\_t[6]$ to $uint8\_t[16]$.

The table is the array of size 2^{24} each position of which is a linked list. Each node of the list consists of half-key $uint8\_t[3]$ and value $uint8\_t[16]$.

```
struct ListNode {
        uint8_t key[3];
        uint8_t value[16];
        struct ListNode *next;
};
```

Each key is divided into two 3-byte parts. The first corresponds to the position in the array and the second is used to search into the linked list.

For example, this is how value is found in the table by key:

```
#define HS_SIZE (1<<24)
typedef struct ListNode *HashSet[HS_SIZE];

bool hs_find(HashSet hs, const uint8_t h[static 6], uint8_t value[static 16])
{
        uint8_t part_r[3];
        memcpy(part_r, h + 3, 3);

        uint64_t pos = h[0] + h[1] * (1<<8) + h[2] * (1<<16);
        return lst_find(hs[pos], part_r, value);
}
```

3.3.2 Search for collisions

Given the data structure, we just generate different messages saving and comparing their hash to the table.

At first, std random was used to generate messages. However, it appeared to be not the most effective way of generation

```
void num_to_message(uint64_t num, uint8_t m[static 16])
{
        for (int i = 0; i < 8; i++) {
                ((uint16_t*)m)[i] = rand() % (1<<16);
        }
}
```

The *colsearch* function takes the number of iterations for which it will be trying to find the collision and placeholders for messages $m1$ and $m2$. On each iteration it generates a new message, calculates its hash and tries to find the same hash in the table. If the collision is not found, the pair hash-message is added to the table and search continues.

```
bool colsearch(uint64_t iters, uint8_t m1[static 16], uint8_t m2[static 16]) {
        struct ListNode** hs = malloc(HS_SIZE * sizeof(struct ListNode*));
        hs_init(hs);
        srand(time(NULL));
        uint8_t h[6];
```

```
        for (uint64_t i = 0; i < iters; i++) {
                num_to_message(i, m1);
                h[0] = IVB0; h[1] = IVB1; h[2] = IVB2;
                h[3] = IVB3; h[4] = IVB4; h[5] = IVB5;
                tcz48_dm(m1, h);
                if (hs_find(hs, h, m2) && memcmp(m1, m2, 16) != 0) {
                        hs_cleanup(hs); free(hs);
                        return true;
                }
                hs_add(hs, h, m1);
        }
        printf("No collision found, try again\n");
        hs_cleanup(hs); free(hs);
        return false;
}
```

Trying an alternative way of generating messages

Also I tried to iterate over consecutive messages instead of drawing each one randomly:

```
void num_to_message(uint64_t num, uint8_t m[static 16]) {
        memset(m, 0, 16);
        *(uint64_t*)m = num;
}
```

This way the collision was found much faster than using the random approach: for the random search 20-30 million iterations were required to find the collision while the straightforward enumeration of messages gives the result in approximately 4 million iterations (and it took about 4 seconds as was proposed).

The result of straightforward enumeration:

```
m1: 218F3E000000000000000000000000000
m2: D14B0A000000000000000000000000000
```

4 Collisions in smht48

Using the property of *smht*48 and *tcz*48*_dm* to find collision in *smht*48 we simply need to find *tcz*48*_dm*-collision for two 16-bit messages and then calculate their tags using the same key.

```
bool smht48ef(uint64_t iters, uint8_t key[static 6],
    uint8_t m1[static 16], uint8_t m2[static 16])
{
        for (int i = 0; i < 3; i++) {
                ((uint16_t*)key)[i] = rand() % (1<<16);
        }
        colsearch(iters, m1, m2);
        uint8_t tag1[6];
        uint8_t tag2[6];
        smht48(key, 16, m1, tag1);
        smht48(key, 16, m2, tag2);
}
```

After searching for collision in *smht*48, we can ensure that tags produced by different messages are the same.

The result of *smht*48*ef* function: two messages and their *smht*48-tags using random message generation approach

```
m1: 656E24667AAD54812955A777D942BBDD
m2: 71DC6B973853F163CFB0F859C750FC88
smht48 tag of m1 and m2: 42443F38F759
```