Homework #2 – KeeLoq, MACs, and collisions

Deadline: 08.11.2022 23:59:59 Brno time
Points: 10 points
Responsible contact: Marek Sýs <syso@mail.muni.cz>
Submit: here

In this homework, you will be implementing and breaking MACs. You already know that the security of MACs is formalized via the *existential forgery* (EF) attack game. In an EF attack, the key must be kept secret: if we wanted to simulate the attack as a part of the HW, we would need to keep the key server-side, allowing you to query the MAC via suitable endpoints, as in HW01. However, in this HW, the number of queries could overwhelm our server. Hence, you will be implementing **collision** attacks against MACs, which can be done completely offline.

Important: This HW asks you to implement certain functions in Python. If this is for some grave reason impossible for you and you indeed must use a different language, contact the HW guarantor. However, since Python is taughted in entry-level programming courses at FI, granting exceptions is not guaranteed.

Apart from this assignment text, you are given the following files: `keeloq.py`, `utils.py`, `macs.py`, `collisions.py`. You are encouraged to read the accompanying documentation to further understand the code. The code provided in these files can be, of course, used in your solutions. In case you use any other code, cite it accordingly (or expect a points loss).

Hint: In this HW, you will frequently work with the KeeLoq block cipher, which is implemented in `keeloq.py`. While you are invited to learn more about the cipher, its knowledge is not required to solve the HW. In your attacks, treat KeeLoq as a black box and only focus on its interface: i.e. the block- and key-sizes.

a. (1.5 point) Implement a function `KeeLoq_CBC_enc` that is missing from the KeeLoq cipher implementation in the file **`keeloq.py`**. `KeeLoq_CBC_enc` must encrypt the message using the KeeLoq block cipher in the *Cipher Block Chaining (CBC)* mode. Your CBC mode should use PKCS_7 padding. You may use any function that is already implemented in **`keeloq.py`** or **`utils.py`**. There is a test setup in **`keeloq.py`** prepared for you to test your implementation, simply execute `python keeloq.py` to run the tests. You can create your own tests.

b. (0.5 point) Similarly, finish the implementation of `MAC_KeeLoq` function (inside the file **`macs.py`**) that uses KeeLoq in the **Vanilla CBC mode** (slide 8 from Lecture 6) to create a Message Authentication Code (MAC). For the initialization vector, use the vector comprising of zero bytes (make sure to use zero bytes `b'\x00'` and not the byte value for ASCII zero). For the purposes of this homework, we require that the tag size is variable determined by the `MAC_size` parameter of `MAC_KeeLoq`. A testing function using several test vectors is provided, execute `python macs.py` to run the tests.

Task 2 – Collision finding (6 points)

a. (2 points) Thanks to your work in Task 1b you have access to `MAC_KeeLoq` function. Your task is to implement `MAC_KeeLoq_collision` function (in the file `collisions.py`) that for a given key `K` and tag size `MAC_size` returns a collision on `MAC_KeeLoq`, i.e., pair of *distinct* messages `msg1, msg2` such that

```
MAC_KeeLoq(msg1, K, MAC_size) = MAC_KeeLoq(msg2, K, MAC_size)
```

You can assume that the `MAC_size` is always less than or equal to four bytes. Test that for `MAC_size = 3` your implementation can find a collision within a second!

b. (1 point) There is a known collision for SHA1 (https://shattered.io/). However, this is just a single collision, and for practical attacks we often need to generate multiple colliding pairs on demand. Your task is to implement a function **`SHA1_collision`** (in the file **`collisions.py`**) that will randomly generate SHA1 collision pairs. The requirements are:

   - for generation of tags, use the **`SHA1`** function from **`macs.py`**,
   - for randomization, use the **`token_bytes`** method from the imported **`secrets`** module,
   - the two messages that are returned in each call must be distinct, but produce the same SHA1 tag,
   - the length of each returned message must be less than four hundred bytes,
   - the probability of returning, within a sequence of **`N`** calls to **`SHA1_collision`**, the same pair of messages twice, is *small*. We do not give a precise definition of *small*; as a rule of thumb, we require that if we call **`SHA1_collision`** repeatedly, at least thousand different colliding pairs can be generated in reasonable time.

c. (3 points) Thanks to your work in Task 2a and 2b you now know enough to be able to find collisions against the **`MAC_combined`** function (from the file **`macs.py`**). **`MAC_combined`** uses both **`MAC_KeeLoq`** and SHA1 in order to create a presumably stronger MAC. For a given **`msg`** and a key **`K`** the **`MAC_combined`** function calculates the tag **`t`** as follows

```
t = SHA1(msg || MAC_KeeLoq(msg, K))
```

   Implement the function **`MAC_combined_collision`** inside the file **`collisions.py`** that returns a quadruple **`(msg1, key1, msg2, key2)`** such that **`msg1 != msg2`** and **`(msg1, key1)`** yields the same tag as **`(msg2, key2)`** under **`MAC_combined`**. Similarly to Task 2a, you can assume the **`Keeloq_MAC_size`** to be always less than or equal to four bytes.

Use less than 600 characters per item, i.e. 2400 in total:

a. (0.5 point) Describe a principle of your attack in Task 2a and its expected runtime.
b. (0.5 point) Describe how and why it is possible to generate new SHA1 collisions in Task 2b.
c. (0.5 point) Describe the principle of your attack in Task 2c.
d. (0.5 point) Explain why is the MAC scheme in Task 2c less secure than `mac' = SHA1(msg || K) || MAC_KeeLoq(msg, K)` – what is the key difference between `mac` and `mac'`? We expect a clear explanation of, e.g., the form "attack on `mac` needs XY expected number of steps ... while similar-style attack on `mac'` needs YZ... expected number of steps. Key difference is ..., which decreases the per-step probability of collision to ...

Overall requirements

You should submit a single `.zip` file containing:

- a directory called `code` containing the 4 files with your solutions of Tasks 1 and 2: `keeloq.py, utils.py, macs.py, collisions.py`; and
- a plain-text file `description.txt` containing your answers for Task 3.

Implement your solutions in precisely the file as we specified. Your zip file should not contain any file or folder not specified above. Use only standard Python and modules already imported in the files. Do not change the signature of the functions that you are implementing, that is the amount, order and names of the parameters or return values. Your solutions will be graded automagically and not conforming to those rules will lead to a point loss. As usual, a solution with no description is worth zero points.