COMP2700 Lab 10 – Hash and MAC

Some of the exercises in this tutorial/lab are selected from Paar & Pelzl's "Understanding Cryptography" (Chapter 11 & 12).

For Exercise 3 and Exercise 5, you will need two python scripts, which you can download from the Wattle page for the course (lab10-code.zip), or download directly to your terminal using the following command:

```
$ wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab10-code.zip
```

Exercise 1.

One of the earlier applications of cryptographic hash functions was the storage of passwords for user authentication in computer systems. With this method, a password is hashed after its input and is compared to the stored (hashed) reference password. People realised early that it is sufficient to only store the hashed versions of the passwords.

Assume you are a hacker and you got access to the hashed password list. Of course, you would like to recover the passwords from the list to impersonate some of the users. Discuss which of the three attacks below allow this. Exactly describe the consequences of each of the attacks:

- Attack A: You can break the one-way property of h.
- Attack B: You can find second preimages for h.
- Attack C: You can find collisions for h.

Exercise 2.

We consider three different hash functions which produce outputs of lengths 64, 128 and 160 bit. After approximately how many random inputs do we have a probability of ε =0.5 for a collision?

Exercise 3.

Consider the following hash function constructed using AES in ECB mode: Given an input x of length n blocks, where each block is 128 bit (16 byte) long, the output of the hash function is obtained by first encrypting x and then computing the result of XOR of all the 128-bit blocks of the ciphertext. More precisely, if $x = x_1 \cdots x_n$, where each x_i is exactly one block (128-bit long), then the hash is

$$H(x) = e_k(x_1) \oplus e_k(x_2) \oplus \cdots \oplus e_k(x_n)$$

where $\oplus$ is the XOR function, and k is a fixed 128 bit key. Note that the encryption key here is not meant to protect secrecy of the input; it is part of the hash algorithm, and as such, its value needs to be publicly available.

The provided python script, ecbhash.py, gives a simple implementation of this hash function.

Use the echbhash.py script to compute the hash value of the following text:

```
0123456789ABCDEF0123456789abcdef
```

Then construct a second preimage of this text. Some useful python commands are provided below to help you solve this problem.

For this exercise, we will use python interactively to perform some (byte) strings manipulation.

To use the ecbhash.py script, load it in the python shell using the 'import' command. The hash function is called ecbhash; to call it, use 'ecbhash.ecbhash(input)', where 'input' is a byte string.

To construct a sample byte string, the easiest is to just use a prefix 'b' in front of a string – this causes python to interpret the string as a sequence of bytes. Here is an example python session, for calculating the hash value of an input (we assume this is run in a linux terminal):

```
alice@comp2700-lab:~/lab10$ python3
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ecbhash
>>>
>>> test=b'This is a test!!'
>>> ecbhash.ecbhash(test)
'9a23fcc53603ee56c2179f2156b2cc5e'
>>>
```

To access bytes in a byte string, you can use the range operator, e.g., x[i:j] denotes the sequence of bytes x[i], x[i+1], …, x[j-1]. The operator '+' is overloaded to allow concatenation of two byte strings. Here are some examples of these operators:

```
>>> x=b'01234567'
>>> x[0:4]
b'0123'
>>> x[4:8]
b'4567'
>>> x[4:8]+x[0:4]
b'45670123'
>>>
```

Exercise 4. Some cryptographic operations based on block ciphers (such as CBC encryption, hash or MAC computations) require that the input message size is a multiple of block size (e.g., for AES, this would be multiple of 16 bytes). If the input is not a multiple of block size, then some padding needs to be added to the input, before it is processed. The PKCS#7 padding[1] scheme is a widely used padding scheme to pad a message so that its length becomes a multiple of block size.

This padding scheme works as follows: Let's assume that the block size is b bytes. Given an input m, if the last block of m is of length $(b - r)$ bytes, then add the byte value of r to m until the last block

---

[1] The specification of PKCS#7 can be found here https://www.ietf.org/rfc/rfc2315.txt .

of m is b byte long. For example, assuming m is the following byte sequence (we use python notation here):

b'12345'

and the block size $b = 16$. So m is a 5-byte long byte array and it's 11 byte short of the block size (16 bytes). In this case, r=11 (or 0xb in HEX), so we pad the byte array m with 11 bytes, each byte containg the value r (0xb). The padded version m (in python byte notation) is therefore:

b'12345\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b'

The pycryptodome library (Crypto.Util.Padding) provides a function called **pad** that implements the PKCS#7 padding scheme. The pad function takes two arguments: the input bytes, and the block size (in bytes). There is also a corresponding **unpad** function, that reverses the padded message to its original. Here's an example of a python session for computing the padded version of m in the example above, with block size of 16 bytes:

```
>>> from Crypto.Util.Padding import *
>>> m=b'12345'
>>> s=pad(m,16)
>>> s

b'12345\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b'

>>> unpad(s,16)

b'12345'

>>>
```

In the official specification of PKCS#7 standard, if the length of the message m is already a multiple of block size, an additional block is added (each byte of that block contains the byte value of the block size). You can confirm this using the pycryptodome's pad function. Supposed we modify the PKCS#7 padding so that if the input is already a multiple of block size then no padding is added. What do you think would be an issue with this modified padding scheme?

Exercise 5. In SHA-1, the input message to the hash function is always padded, even when the length of the input is a multiple of the block size of SHA-1. Why is this the case? Justify your answer in terms of the attack(s) it counters.

Exercise 6. In this exercise, we will implement the forgery attack on CBC-MAC discussed in the lecture on MAC. We will use an implementation of CBC-MAC based on AES with 128-bit key (see the provided cbcmac.py script). For this exercise, we are going to use python in an interactive mode, as it is easier to manipulate byte strings and converting character encodings.

We first run python3 and imports the cbcmac.py script. We show here the output of an interactive python session, run in the lab VM:

```
alice@comp2700-lab:~/lab10$ python3
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cbcmac
```

We then define the input text that we will calculate the MAC for and the MAC key:

```
>>> input=b'TESTING CBC-MAC!'
>>> key=b'0123456789abcdef'
>>>
```

Note: the b prefix says that python should treat the string that follows as a sequence of bytes, rather than as a string. In python this is given the type 'bytes'.

For this exercise, we fix the IV to the following value: b'fedcba9876543210' (this is hard-coded in the cbcmac.py file).

(You don't need to use these exact values for the key and the IV – any values will do, as long as they are exactly 16 bytes each.)

Next, we compute the MAC of 'input' with key 'key', using the gen_mac function in cbcmac.py:

```
>>> mac=cbcmac.gen_mac(input,key)
>>> mac
'0a607906c6a1173a4655344f11a48b5b'
>>>
```

Note that the MAC output is given as a HEX string.

We can verify that 'mac' is the correct MAC for 'input', given the key 'key', using the verify_mac function:

```
>>> cbcmac.verify_mac(input, key, mac)
True
>>>
```

Your task is now to construct a different input, call it 'fake_input', such that the MAC of fake_input is exactly 'mac', without using the key given above.

Hint: Review the CBC-MAC lecture to see how the attack works. Here are a couple of python commands you may find useful:

- To concatenate two byte strings, you can use the operator '+'. For example:

```
>>> x=b'0123'
>>> y=b'4567'
>>> x+y
b'01234567'
```

- To convert a HEX string into bytes, use 'bytes.fromhex(<string>)'. For example:

```
>>> str='414243'
>>> bytes.fromhex(str)
b'ABC'
>>>
```

- To compute the xor of two byte strings, use strxor. For this you need to import Crypto.Util.strxor (see Lab 9).

Extension Exercise (Optional)

Exercise 7. For hash functions it is crucial to have a sufficiently large number of output bits, with, e.g., 160 bits, in order to thwart attacks based on the birthday paradox. Why are much shorter output lengths of, e.g., 80 bits, sufficient for MACs? For your answer, assume a message x that is sent in clear together with its MAC over the channel: $(x, MAC_k(x))$. Exactly clarify what an attacker has to do to attack this system.

Exercise 8. We study two methods for integrity protection with encryption.

a) Assume we apply a technique for combined encryption and integrity protection in which a ciphertext c is computed as

$$c = e\_k(x \parallel h(x))$$

where $h()$ is a hash function. This technique is not suited for encryption with stream ciphers if the attacker knows the whole plaintext x corresponding to the ciphertext c. Explain how an attacker can replace x with arbitrary x' and compute c' such that the receiver will verify the message correctly. Assume that x and x' are of equal length.

b) Is the attack still applicable if the checksum is computed using a MAC:

$$c = e_{k1}(x \,||MAC_{k2}(x))$$

Assume that $e()$ is a stream cipher as above.

Exercise 9.

In this exercise, we are going to test the collision attack on SHA-1 found by Google and CWI. To demonstrate the collision attack, we will create two PDF documents with different contents, but with the same SHA-1 hash. To compute the SHA-1 hash of a file, you can use the 'shasum' command in linux.

The collision attack works only on a very specific type of PDF documents, that contain embedded JPEG images. To create these PDFs, we will use a Word document, to first create two PDF documents that are almost the same size, but with slightly different contents. You can use the provided Word template (letter.docx) – vary the dollar value in that document to create two different documents, and export them as PDF. This attack works only on PDF document of size less than 64kb, so if the PDF file you produce is larger than 64kb you will need to reduce its size.

We will use the exploit script provided here https://github.com/nneonneo/sha1collider

Download the exploit script and run

./collide.py file1.pdf file2.pdf –progressive

where file1.pdf and file2.pdf are the PDF files you want to generate SHA-1 collision for. If successful, this should produce two files out-file1.pdf and out-file2.pdf. Confirm that they have the same SHA-1 hash using shasum.

Note that if the script complains about a missing 'cjpeg', you may need to install the libjpeg-progs package:

sudo apt install libjpeg-progs