Crypto Lab – One-Way Hash Function, MAC, and Digital Signature

IMPORTANT NOTES:
Study lecture materials at least 1 hour and prepare Lab Task 3.1 prior to the lab session. Prepared questions will be discussed in the lab session.

1 Overview

The learning objective of this lab is for students to get familiar with one-way hash functions, Message Authentication Codes (MACs) and Digital Signatures. After finishing the lab, in addition to gaining a deeper undertanding of the concepts, students should be able to use tools to generate one-way hash value, MACs and digital signatures for a given message.

2 Lab Environment

In this lab we will use gpg tool to generate the digital signatures for a file and use `sagemath` tool to generate message digest (hash value) and message authentication code values.

3 Lab Tasks

3.1 Generating Digital Signature

Please create a text file `plain.txt`. To sign the text file `plain.txt` using default user in gpg key ring:

```
$ gpg --sign plain.txt
```


will create a compressed file of the `plain.txt` together with the signature called `plain.txt.gpg`. To verify signature:

```
$ gpg --verify plain.txt.gpg
```

and

```
$ gpg --output plain.txt --decrypt plain.txt.gpg
```

to verify and retrieve the file.
To generate a detached signature use:

```
$ gpg --detach-sign --output sig.bin plain.txt
```

To verify a detached signature:

```
$ gpg --verify sig.bin plain.txt
```

3.2 Generating Message Digest and MAC

In this task, we will play with various one-way hash algorithms. You can use the `hashlib` module in the `sagemath` tool to generate the hash value for a message.

```
import hashlib
msg='hello world'
h=hashlib.sha256(msg.encode('utf-8'))
print(h.hexdigest())
```

You can replace the `msg` with your own message, and replace `sha256` with other specific one-way hash algorithms, such as `md5` or `sha1`, etc. In this task, you should try at least 3 different algorithms, and describe your observations. You can find the supported one-way hash algorithms by using `print(hashlib.algorithms_available)`.
Since the MD5 has 128 bits, SHA-1 has 160 bits, and SHA-256 has 256 bits hash value, the length of the hash value should be different:

```
sage: import hashlib
....: msg='hello world'
....: h_md5=hashlib.md5(msg.encode('utf-8'))
....: h_sha1=hashlib.sha1(msg.encode('utf-8'))
....: h_sha256=hashlib.sha256(msg.encode('utf-8'))
....: print(h_md5.hexdigest())
....: print(h_sha1.hexdigest())
....: print(h_sha256.hexdigest())
....:
5eb63bbbe01eeed093cb22bb8f5acdc3
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
```


3.3 Keyed Hash and HMAC

In this task, we would like to generate a keyed hash (i.e. MAC) for a message. We can use the `hmac` module in the `sagemath` tool. The following example generates a keyed hash for a message using the HMAC-SHA256 algorithm. The `key` argument in the `hmac.new` function is the key string, and the `digestmod` argument is the hash algorithm.

```
import hashlib, hmac
msg='hello world'
h=hmac.new(key=b'secret key', msg=msg.encode('utf-8'), digestmod=hashlib.sha256)
print(h.hexdigest())
```

Please generate a keyed hash using HMAC-MD5, HMAC-SHA256, and HMAC-SHA1 for any message that you choose. Please try several keys with different length. Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?
To generate the HMAC with different hash algorithms:

```
sage: import hashlib, hmac
....: msg='hello world'
....: hmac_md5=hmac.new(key=b'secret key', msg=msg.encode('utf-8'), digestmod=hashlib.md5)
....: hmac_sha1=hmac.new(key=b'secret key', msg=msg.encode('utf-8'), digestmod=hashlib.sha1)
....: hmac_sha256=hmac.new(key=b'secret key', msg=msg.encode('utf-8'), digestmod=hashlib.sha256)
```

```
....: print(hmac_md5.hexdigest())
....: print(hmac_sha1.hexdigest())
....: print(hmac_sha256.hexdigest())
....:
7ded3b97effe4c5d8549cd802d8fee9e
8ee55470e24ad2ee7610e8d5c39e16017fe6cc65
c61b5198df58639edb9892514756b89a36856d826e5d85023ab181b48ea5d018
```

Notice that the results are different with the previous question for the same message.
Since the HMAC will do the padding for the key string, the key can be **any** length:

```
sage: import hashlib, hmac
....: msg='hello world'
....: hmac_short=hmac.new(key=b'a', msg=msg.encode('utf-8'), digestmod=hashlib.sha256)
....: hmac_long=hmac.new(key=b'this is supposed to be a very very long key', msg=msg.encode('utf-8'), digestmod=hashlib.sha256)
....: print(hmac_short.hexdigest())
....: print(hmac_long.hexdigest())
....:
7b2e16b6e6ed86e0ba584bea4280b09e3cbc58c94787d0566d9c19c8dec3f49e
a3cbba7c73f8c5caffd314b268944ba8fb098d6f92e171c5aac66446f4681da2
```

Notice that the HMAC for the same message will be different if two keys are distinct.

4 Optional Further Explorations

4.1 Generating Message Digest and MAC Using the Openssl

We can also generate the hash value or the HMAC for a file by using the `openssl dgst` command. To see the manuals, you can type `man openssl` and `man dgst`.


```
% openssl dgst dgsttype filename
```

Please replace the `dgsttype` with a specific one-way hash algorithm, such as `-md5`, `-sha1`, `-sha256`, etc. You can find the supported one-way hash algorithms by typing "`man openssl`".

In addition, we can use the `-hmac` option (this option is currently undocumented, but it is supported by `openssl`) to generate the HMAC for a file. The following example generates a keyed hash for a file using the HMAC-MD5 algorithm. The string following the `-hmac` option is the key.

```
% openssl dgst -md5 -hmac "abcdefg" filename
```

```
echo "Say the year is the year of the phoenix" > plain.txt
openssl dgst -md5 plain.txt
MD5(plain.txt)= 1546dc201eca642182e9af233308d445
openssl dgst -sha1 plain.txt
SHA1(plain.txt)= 133e7e7c4bdd7a47dea1a828711b2d1e727afbc1
openssl dgst -sha256 plain.txt
SHA256(plain.txt)= 6f6078c5c7917e6089bf6386a3006856af7d04271ab5f3ab85c1f7152ef88f7b
```

You can also use the command `md5sum`, `sha1sum`, and `sha256sum` to produce the same output:

```
md5sum plain.txt
1546dc201eca642182e9af233308d445  plain.txt
sha1sum plain.txt
```

```
133e7e7c4bdd7a47dea1a828711b2d1e727afbc1  plain.txt
sha256sum plain.txt
6f6078c5c7917e6089bf6386a3006856af7d04271ab5f3ab85c1f7152ef88f7b  plain.txt
```

Note that the hash value for the file may be different from the hash value for the string in sagemath, because there are usually some additional control characters (e.g. the newline character) appended at the end when creating a text file. You can generate the same hash value in sagemath by appending the newline character:

```
sage: import hashlib
....: h=hashlib.sha256("Say the year is the year of the phoenix\n".encode('utf-8'))
....: print(h.hexdigest())
....:
6f6078c5c7917e6089bf6386a3006856af7d04271ab5f3ab85c1f7152ef88f7b
```

4.2 The Randomness of One-way Hash

To understand the properties of one-way hash functions, we would like to do the following exercise for MD5 and SHA256.:

1. Create a message of any length.
2. Generate the hash value H_1 for this message using a specific hash algorithm.
3. Flip one bit of the message.
4. Generate the hash value H_2 for the modified message.
5. Please observe whether H_1 and H_2 are similar or not. You can write a short program to count how many bits are the same between H_1 and H_2.

For example, we may change the first letter from "S" to "T" and generate the new hash value:

```
openssl dgst -sha256 plain.txt
SHA256(plain.txt)= c84c45a657ba085e3711fee4460e8e6f15800e88ab44fc7da88497d4c917d239
```

We can then compute the number of different bits in these two hash values in sagemath:

```
sage: h1=0x6f6078c5c7917e6089bf6386a3006856af7d04271ab5f3ab85c1f7152ef88f7b;
....: h2=0xc84c45a657ba085e3711fee4460e8e6f15800e88ab44fc7da88497d4c917d239;
....: count=0;
....: for i in [1..256]:
....:     x1=h1 & 0x1;
....:     x2=h2 & 0x1;
....:     if (x1!=x2):
....:         count=count+1;
....:     h1=h1>>1;
....:     h2=h2>>1;
....: print(count);
....:
139
```

There are 139-bit difference in this specific example. The number of flipped bits in the hash value should be about half of the size of the hash value on average due to the avalanche effect (e.g. 128 flipped bits for SHA-256).

4.3 Birthday Attack

Use the `sagemath` tool to find two 32-bit messages M_1 and M_2 (both interpreted as integers less than 2^{32} in `sagemath`) such that their SHA-256 hash values $SHA256(M_1)$ and $SHA256(M_2)$ have the same 32 leading (leftmost) bits (i.e. the first 8 hexadecimal digits of the hash values are equal), by exploiting the birthday paradox. Compare the number of hash trials you need in your experiments to the estimated expected number from tutorial question 11b.

Hint: To create a hash table in `sagemath`, you can define `H={}` and use `H[key]=m` to add a value `m` with key `key` into the hash table. To check whether a key `key1` is in the hash table, use `key1 in H`.

```
import hashlib
H={}
l=32
i=1;
while (True):
    m=ZZ.random_element(2^32)
    hm=hashlib.sha256(m.str().encode('utf-8')).hexdigest()[:(l/4)]
    if (hm in H and H[hm]!=m):
        print(m,H[hm],hm,i)
        break
    else:
        H[hm]=m;
    i=i+1
```


Sample output: (323364076, 2936718134, '33162def', 44080)

The collision found by `sagemath` and the number of trials may be different every time, since the integers are randomly generated. However, the expected number of trials should be about $\sqrt{2^{32}} = 2^{16} = 65536$.

To check the hash values for the collision:

```
sage: hashlib.sha256(323364076.str().encode('utf-8')).hexdigest()
'33162defb81dcc9b50de166ed34d222c6e48d48f4642dea6e1606593fff95cd3'
sage: hashlib.sha256(2936718134.str().encode('utf-8')).hexdigest()
'33162def6cb1aba0ad1d37d52020efe17142ea9f7e8e3c13eb0f0f4b7507f133'
```

We can see that the leading 8 hexadecimal digits are identical.