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)`.

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, 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?

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
```

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.

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 tries 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
```