DPROT LAB WORK 2: Hash functions and MACs

Ismael Douha Prieto

MCYBERS - Q1 2020/21

1 Introduction

In this laboratory work I performed a MAC forgery attack against CBC-MAC, which is insecure when the length of the messages are different. A part from that, I also coded some scripts to create a Merkle hash tree, to add a new file to a given tree, to generate a proof of membership and to verify a given proof of membership.

2 Recreation of known MAC forgery attacks

2.1 CBC-MAC concatenation Attack

The first step to perform this attack is generating a 16-bytes key length and two messages with different length. In this case, I used the ones that the statement suggest. The key and both messages were stored in files, as the following commands show respectively.

```
$ openssl rand -hex 16 > key.txt
$ echo -n 'What about joining me tomorrow for dinner?' > ↩
    message1.txt
$ echo -n 'Oops, Sorry, I just remember that I have a meeting ↩
    very soon in the morning.' > message2.txt
```

To create the tag of each message, previously to the AES-128-CBC-MAC generation, is necessary to append at the beginning a header, in this case, formed by 16 null bytes.

```
$ echo -ne '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\↩
    x00\x00\x00\x00' > head.dat
$ cat head.dat message1.txt | openssl enc -aes-128-cbc -K `cat ↩
    key.txt` -iv 0 | tail -c 16 > tag1.dat
$ cat head.dat message2.txt | openssl enc -aes-128-cbc -K `cat ↩
    key.txt` -iv 0 | tail -c 16 > tag2.dat
```

With the first command, I created a file which contains 16 null bytes and with the next ones, I created the tag for each message.

In order to investigate how AES-128-CBC completes the last incomplete block of a message, I encrypted and decrypted each message with the following commands.

```
$ openssl enc -aes-128-cbc -K 4aff61636e89c9d4fb7bde4bc71c7d9b -iv 0 -in message1.txt -out cipher1.dat
$ openssl enc -aes-128-cbc -K 4aff61636e89c9d4fb7bde4bc71c7d9b -iv 0 -in message2.txt -out cipher2.dat
$ openssl enc -d -aes-128-cbc -K 4aff61636e89c9d4fb7bde4bc71c7d9b -iv 0 -nopad -in cipher1.dat -out padded1.dat
$ openssl enc -d -aes-128-cbc -K 4aff61636e89c9d4fb7bde4bc71c7d9b -iv 0 -nopad -in cipher2.dat -out padded2.dat
```

Now, *padded1.dat* and *padded2.dat* contains the message with the padding added. With *xxd* I inspected the files.

```
$ xxd padded1.dat
00000000: 5768 6174 2061 626f 7574 206a 6f69 6e69  What about joini
00000010: 6e67 206d 6520 746f 6d6f 7272 6f77 2066  ng me tomorrow f
00000020: 6f72 2064 696e 6e65 723f 0606 0606 0606  or dinner?......
```

```
$ xxd padded2.dat
00000000: 4f6f 7073 2c20 536f 7272 792c 2049 206a  Oops, Sorry, I j
00000010: 7573 7420 7265 6d65 6d62 6572 2074 6861  ust remember tha
00000020: 7420 4920 6861 7665 2061 206d 6565 7469  t I have a meeti
00000030: 6e67 2076 6572 7920 736f 6f6e 2069 6e20  ng very soon in
00000040: 7468 6520 6d6f 726e 696e 672e 0404 0404  the morning.....
```

At the end of each padded message, 6 bytes with 6 value and 4 bytes with 4 value are appended respectively. This kind of padding is defined in ***PKCS#7***, which consists in adding at the end of the message the number of bytes needed to complete the block. The number of bytes added also defines the number of each byte.

Once I knew how the padding behaves, it was the time to forge the tag. The first thing to do was creating a forgery message. This message was composed by the header (16 null bytes), the message 1, the padding that the encryption append (which was created previously and stored in *padding.txt*), the tag of the first message and the second message.

```
$ echo -ne '\x06\x06\x06\x06\x06\x06' > padding.txt
$ cat head.dat message1.txt padding.txt tag1.dat message2.txt > forgery.dat
```

With the forgery created, I created its tag and it was the same of *tag2.dat*. With *diff* command I verified and no differences were founded.

```
$ cat forgery.dat | openssl enc -aes-128-cbc -K `cat key.txt` -iv 0 | tail -c 16 > tag2b.dat
$ diff tag2.dat tag2b.dat
```

With these steps, I proved that it is possible to create the same tag for two different messages using CBC-MAC.

3 Building Merkle hash trees

For this part of the assignment I was requested to create four scripts related to Merkle hash trees. A Merkle hash tree generates an identifier for a file or group of files. In addition, when a Merkle tree hash is created, you can add files, verify if a file belongs to a specific Merkle hash tree and generate a proof of membership.

The first script that I developed (*build_tree.py*) creates a Merkle hash tree. When you execute it, it requests for the number of files that the tree will contain and the path of the files. With this information, it creates each node containing its respective hash, the header for a document and for a node and the hash tree. The hash tree is composed by a public info, which is placed in the first line, and contains the definition of the structure (MerkleTree), the hash algorithm used, the values of both headers, the number of nodes, the number of layers and the root hash. To be continue, each line contains the node identifier and the hash that contains. The next image is an example of hash tree file content.

```
$ cat hash_tree.txt
MerkleTree:sha1:353535353535:e8e8e8e8e8e8:3:3:c96afa6d50074a0992998e1803c8b99fff69337b
0:0:adbf5ff953e8afde86eb3d2da3618928f25c0263
0:1:d54b4273dac73bf4faec19abaff062ef2fa03b0b
0:2:66531d8b3b30ed7af1c388e2c8918cb7ab513ea4
1:0:a58d396ae7cbce80994299ce89b9f0fad17800b1
1:1:e491a62f7b76f0dfcd1e9bb93911b31f02ed7d4f
2:0:c96afa6d50074a0992998e1803c8b99fff69337b
```

The second one (*add_doc.py*) adds a node in the tree. To do it, it is necessary to have in the same directory that the script is stored, the hash tree file and all the nodes. When the script is executed, it requests for the path of the new node. Once the path is given, recomputes the tree, adding the node and modifying the necessary nodes and the hash tree.

```
$ cat hash_tree.txt
MerkleTree:sha1:353535353535:e8e8e8e8e8e8:4:3:cbf716884ed9bf1a0d9dfa5c21351d033e79ae5d
0:0:adbf5ff953e8afde86eb3d2da3618928f25c0263
0:1:d54b4273dac73bf4faec19abaff062ef2fa03b0b
0:2:66531d8b3b30ed7af1c388e2c8918cb7ab513ea4
0:3:dbc4a9798123215117fc3b48e9817b7184da4b3f
1:0:a58d396ae7cbce80994299ce89b9f0fad17800b1
```

```
1:1:15e4ef5905517ff79575d146a089975e156a335c
2:0:cbf716884ed9bf1a0d9dfa5c21351d033e79ae5d
```

The third script (*generate_proof.py*), prints the necessary info to verify if a file belongs to a hash tree. To check it, it is necessary to have the hash tree in the same directory that the script is stored and give the correct path and position of the file when the script requests it.

```
$ python3 generate_proof.py
Type the path of the file to generate the proof:add_doc.py
Type the position of the file to generate the proof:3
0:2:66531d8b3b30ed7af1c388e2c8918cb7ab513ea4
1:0:a58d396ae7cbce80994299ce89b9f0fad17800b1
2:0:cbf716884ed9bf1a0d9dfa5c21351d033e79ae5d
```

The last script I developed (*verify_proof.py*), it checks if a node belongs to a hash tree. To do it, it requests the public info of the hash tree, the path of the document to verify and the path of a file where the name of nodes needed to verify it are stored in a specific format (*node0.2:node1.0:node2.0*). With this info, it prints if the document belongs to the hash tree or not.

```
$ python3 verify_proof.py
Type the public info in the correct format: MerkleTree:sha1↩
   :353535353535:e8e8e8e8e8e8:4:3:↩
   cbf716884ed9bf1a0d9dfa5c21351d033e79ae5d
Type the path of the document to verify: add_doc.py
Type the path of the file with the proof nodes, separated by ":↩
   " and sorted: nodes.txt
***OK***
```