

Lab Work 2: Hash functions and MACs

Aleix Galan Figueras - 26062500G

Data Protection

UNIVERSITAT POLITÈCNICA DE CATALUNYA

20 d'octubre de 2020

4.1 CBC-MAC concatenation attack

The CBC-MAC schema requires of a fixed message length to be secure. If not, an attacker that have access to two tags and two clear messages could perform a concatenation attack and forge the second tag.

Let (m, t) and (m', t') be two valid message/tag pairs. In this attack we cant to create a m'' that produces $t'' = t'$. To do so we need to have in mind the structure of the CBC mode of operation. In CBC-MAC, the message data in each block is xored with the result of the previous block (xored with $IV = 0$ in the first block), then applied the cryptographic function $E_k()$ and passed to the next block.

If we want to forge t' we need the first m'' block entry to $E_k()$ identical as if we were computing just the m' tag. What we know is that the CBC mode performs a xor before entering the $E_k()$ function and that the xor function applied 2 times is transparent.

The tag t is the result of the last block from the MAC computation of m . Thus, if we merely concatenate $m'' = m|m'$, the message entering the $E_k()$ function for the first block of m' part will not be the desired m'_1 but $m'_1 \oplus t$. But we can leverage on this xor function because we have the tag t previously intercepted, so instead of concatenate m and m' we can xor t with the first block of m' . Then, in the first block of m' we have $m'_1 \oplus t$ and is then xored with the result of the previous block (which is the tag for m) having $(m'_1 \oplus t) \oplus t = m'_1$ entering the $E_k()$ function, exactly what we need to forge t' .

In this lab, all messages have a header with the same length of a block (so, the same length of a tag) will all 0. So instead of xoring the previous tag with the first block (all 0) of the message we just concatenate it.

The messages and the new crafted message are showed in the Figure 1. We can also appreciate that the message1 has been padded manually to produce the tag1 in the CBC chain before

the message2 part. If we hadn't padded the message, the block sizes would not align correctly to produce the desired result.

```
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DPROT/practiques
00000000: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000010: 5768 6174 2061 626f 7574 206a 6f69 6e69 What about joini header +
00000020: 6e67 206d 6520 746f 6d6f 7272 6f77 2066 ng me tomorrow f message1
00000030: 6f72 2064 696e 6e65 723f or dinner?
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DPROT/practiques
00000000: 5dcd 77ae 2595 f23c a47d 0a9d fd62 4c36 ].w.%..<}.bL6 tag1
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DPROT/practiques
00000000: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000010: 4f6f 7073 2c20 536f 7272 792c 2049 206a Oops, Sorry, I j header +
00000020: 7573 7420 7265 6d65 6d62 6572 2074 6861 ust remember tha message2
00000030: 7420 4920 6861 7665 2061 206d 6565 7469 t I have a meeti
00000040: 6e67 2076 6572 7920 736f 6f6e 2069 6e20 ng very soon in
00000050: 7468 6520 6d6f 726e 696e 672e the morning.
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DPROT/practiques
00000000: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000010: 5768 6174 2061 626f 7574 206a 6f69 6e69 What about joini
00000020: 6e67 206d 6520 746f 6d6f 7272 6f77 2066 ng me tomorrow f h + m1 |
00000030: 6f72 2064 696e 6e65 723f 0606 0606 0606 or dinner?..... tag1 |
00000040: 5dcd 77ae 2595 f23c a47d 0a9d fd62 4c36 ].w.%..<}.bL6 m2
00000050: 4f6f 7073 2c20 536f 7272 792c 2049 206a Oops, Sorry, I j
00000060: 7573 7420 7265 6d65 6d62 6572 2074 6861 ust remember tha
00000070: 7420 4920 6861 7665 2061 206d 6565 7469 t I have a meeti
00000080: 6e67 2076 6572 7920 736f 6f6e 2069 6e20 ng very soon in
00000090: 7468 6520 6d6f 726e 696e 672e the morning.
```

Figure 1: Message files for the CBC-MAC concatenation attack.

Once executed the script `cbc_mac_attack.sh` the resultant tags are compared, as shown in the Figure 2

```
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DPROT/practiques/practica2/exercice_1$ bash cbc_mac_attack.sh
hex string is too short, padding with zero bytes to length
hex string is too short, padding with zero bytes to length
hex string is too short, padding with zero bytes to length
Original tag2.dat: 98cf14f608e8b0d574c6920935869a96
Forged tag2_forget.dat: 98cf14f608e8b0d574c6920935869a96
```

Figure 2: Execution of the script `cbc_mac_attack.sh`.

The corresponding code for this exercise can be found at: github.com/Algafix/problemes-dprot/tree/master/practica2/exercice_1

5 Building Merkle hash trees

The notation of this exercise will be the same as described in the laboratory guide. The documents will be named as `doc k .dat` with k from $[0, n-1]$, being n the number of documents. The documents are stored in the `docs/` folder.

The nodes are named `node i . j` . The leaves are at the $i = 0$ level. The nodes are stored in the `nodes/` folder.

The folder with the code and the example documents is stored at github.com/Algafix/problemes-dprot/blob/master/practica2/exercice_2.

Merkle tree generator

This section describes the script `create_merkle.sh` that can be found at github.com/Algafix/problemes-dprot/blob/master/practica2/exercice_2/create_merkle.sh.

In order to generate the tree we give as parameter to the script the number of documents or leafs (n). The script looks into the `docs/` folder for the documents $[0, n-1]$ and calculate the hash with the document prepend (in our case `\x00`). The variable `max_j` is the number of files in the current layer, in this case is equivalent to n . This block of code also save the node position and hash in the `merkle_tree.txt` file that will define the Merkle tree.

```
1 # Generate leaf nodes from the documents
2 for j in $(seq 0 $((max_j-1)));
3 do
4     cat docs/doc.pre docs/doc$j.dat | openssl dgst -sha1 -
        binary > nodes/node$i.$j;
5     echo $i:$j:$(cat nodes/node$i.$j | xxd -p) >> merkle_tree.
        txt
6 done;
```

Then, for each layer the script compute the hash of the nodes as the concatenation of the node prepend (in our case `\x01`), the hash of left node and the hash of right node. The number of nodes in each layer (`max_j`) is half the nodes of the previous one, ceiled. As bash only works with integers truncating the operations, we compute the nodes of the next layer as: `max_j=$((max_j/2 + max_j%2))`.

We also need to know if the number of nodes is odd because then the hash for the last node of the layer will have no right partner (but minding in the if condition that the root node is only one).

At the end, we just append the Merkle tree definition at the beginning of the `merkle_tree.txt` file.

```
1 # Iterate over the tree levels
2 # Each level has half (ceiled) the nodes of the previous one
3 while [ $((($max_j/2)) -gt 0 ]
4 do
5 i=$((($i+1))
6 odd=$((($max_j%2))
7 max_j=$((($max_j/2 + $odd))
8
9 for j in $(seq 0 $((($max_j-1)));
10 do
11     if [ $j -eq $((($max_j-1)) ] && [ $odd -eq 1 ] # If there is
12         no right node
13     then
14         cat nodes/node.pre nodes/node$((($i-1)).$((2*$j)) |
15             openssl dgst -sha1 -binary > nodes/node$i.$j;
16     else
17         cat nodes/node.pre nodes/node$((($i-1)).$((2*$j)) nodes/
18             node$((($i-1)).$((2*$j+1)) | openssl dgst -sha1 -
19             binary > nodes/node$i.$j;
20     fi
21     echo $i:$j:$(cat nodes/node$i.$j | xxd -p) >> merkle_tree.
22         txt
23 done;
24 done;
25
26 echo -e "MerkleTree:sha1:${doc_prepend:2}:${node_prepend:2}:$n:
27     $((($i+1))):$(cat nodes/node$i.$j | xxd -p)\n$(cat merkle_tree
28     .txt)" > merkle_tree.txt
```

After executing the script with 5 files, the file `merkle_tree.txt` lies as follows:

```
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DP
ROT/practiques/practica2/exercice_2$ cat merkle_tree.txt
MerkleTree:sha1:00:01:5:4:4da8bbff8136e98ebb780817d01dcb14d684f00e
0:0:a8a30180f28963bf2b8b41f7c2873434c00f987d
0:1:9004f47f8f2f7e5811947ac3969d2c78bdaed441
0:2:42bale06f20e86ac9df56048f62a869c39045b43
0:3:d6ef868f4bedd1df15ed209dedd1f7cbc90c79e9
0:4:616b8b02c6e89c6547b1be6ce5394bf88305fd4d
1:0:d4e69b444bf0fd23496772e2e9eacecc11928df4
1:1:8a0422b3b2656a38c0b9522d07f6e9b6fcbb037b
1:2:168c876753804ee878c3530f6ee0625c539b9462
2:0:48c6a1092af77c5a48e87c53fd21f606e86c4d95
2:1:e8649e8407076b48a419cb48cf984334779c6d8e
3:0:4da8bbff8136e98ebb780817d01dcb14d684f00e
```

Figura 3: Merkle tree definition file with 5 files.

Merkle tree insertion

This sections describes the script `add_node.sh` that can be found at github.com/Algafix/problems-dprot/blob/master/practica2/exercice_2/add_node.sh.

This script suppose that the new node is correctly named and placed in the `docs/` directory. For example if we continue with the previous section tree, the new node name is `doc5.dat`.

First of all the script reads the tree definition from the first line of `merkle_tree.txt` file and creates the leaf for the new node. Then inserts the node information at the correct position in the definition file.

```
1 IFS=":";
2 read -r name alg doc_pre node_pre n max_i root < merkle_tree.
  txt
3
4 i=0
5 cat docs/doc.pre docs/doc$n.dat | openssl dgst -sha1 -binary >
  nodes/node$i.$n
6 sed -i "/^$i:$((n-1)):/a $i:$n:$(cat nodes/node$i.$n | xxd -p)
  " merkle_tree.txt
```

Then we compute in a similar way that in the previous section witch nodes we have to update. The updated node is always the last node of the level, so updating the j variable in the same way as we updated max_j we can select which nodes need to be recomputed. We

have to mind that there may be new nodes without right node and we have to treat that case.

Is also necessary to modify the tree definition file with the new hash of the nodes. We use the `sed` command for that.

```
1 # Updating the tree
2 j=$n
3 while [ $j -gt 0 ]
4 do
5 i=$((i+1))
6 odd=$((j%2))
7 j=$((j/2))
8 if [ $odd -eq 1 ] # If it has no right node
9 then
10     cat nodes/node.pre nodes/node$((i-1)).$((2*j)) nodes/node
        $((i-1)).$((2*j+1)) | openssl dgst -sha1 -binary >
        nodes/node$i.$j;
11 else
12     cat nodes/node.pre nodes/node$((i-1)).$((2*j)) | openssl
        dgst -sha1 -binary > nodes/node$i.$j;
13 fi;
14 sed -i "s/^$i:$j:.*$/i:$j:$(cat nodes/node$i.$j | xxd -p)/g"
        merkle_tree.txt
15 done;
16
17 sed -i "s/^$name.*/$name:$alg:$doc_pre:$node_pre:$((n+1)):$((n
        i+1)):$$(cat nodes/node$i.$j | xxd -p)/g" merkle_tree.txt
```

After executing the command, the `doc5.dat` file is inserted in the tree and the `merkle_tree.txt` file is updated as in Figure 4. If we want to add another file, we have to store the file in the `docs/` folder with the proper name and execute the command `bash add_node.sh` again.

```
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DP
ROT/practiques/practica2/exercice_2$ cat merkle_tree.txt
MerkleTree:sha1:00:01:6:4:772091d12e956cc60c666307fb7c7ef7c302f3eb
0:0:a8a30180f28963bf2b8b41f7c2873434c00f987d
0:1:9004f47f8f2f7e5811947ac3969d2c78bdaed441
0:2:42ba1e06f20e86ac9df56048f62a869c39045b43
0:3:d6ef868f4bedd1df15ed209dedd1f7cbc90c79e9
0:4:616b8b02c6e89c6547b1be6ce5394bf88305fd4d
0:5:1910c50d15e4a83d6ef93adfacc2999103c7ee5a0
1:0:d4e69b444bf0fd23496772e2e9eacecc11928df4
1:1:8a0422b3b2656a38c0b9522d07f6e9b6fcbb037b
1:2:aad1ae08487b2a37e3d5267ab409b57a65a34d68
2:0:48c6a1092af77c5a48e87c53fd21f606e86c4d95
2:1:399fbb1c2c17c320ac4f14bd8f8d0a42c3dec1ee
3:0:772091d12e956cc60c666307fb7c7ef7c302f3eb
```

Figura 4: Merkle tree definition file with 6 files (added from 5 files).

Merkle tree proof generator

This section describes the script `proof_generator.sh` that can be found at github.com/Algafix/problemes-dprot/blob/master/practica2/exercice_2/proof_generator.sh.

First, we read the public definition of the tree and write it in the proof verification file. The script receives a parameter with the number of the position of the file from which we want the proof.

```
1 IFS=":";
2 read -r name alg doc_pre node_pre n max_i root < merkle_tree.
  txt;
3 doc_k=$1
4 j=$1
5 i=0
6 max_j=$n
7
8 echo "$name:$alg:$doc_pre:$node_pre:$n:$max_i:$root" > proof_
  doc$doc_k.txt
```

Then, we search which nodes are necessary for the verification in the same way as we did to update the nodes in the previous section. But instead of computing the hash we store the hash value in the proof file. If there is no right node, an empty hash is written.


```

1 while [ $i -lt $((($max_i-1)) )
2 do
3     if [ $((j%2)) -eq 0 ]
4     then
5         if [ $((($j+1)) -lt $max_j ]
6         then
7             echo "$i:$((($j+1))):$(cat nodes/node$i.$((($j+1)) |
8                 xxd -p)" >> proof_doc$doc_k.txt
9         else
10            echo "$i:$((($j+1)):" >> proof_doc$doc_k.txt
11        fi
12    else
13        echo "$i:$((($j-1))):$(cat nodes/node$i.$((($j-1)) | xxd -
14            p)" >> proof_doc$doc_k.txt
15    fi
16    j=$((j/2))
17    i=$((i+1))
18    max_j=$((($max_j/2+$max_j%2))
19 done;

```

For example, in our current example and after adding the 5th node. The proof file for the files 0 and 5 are as follow:

```

(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DP
ROT/practiques/practica2/exercice_2$ cat proof_doc0.txt
MerkleTree:sha1:00:01:6:4:772091d12e956cc60c666307fb7c7ef7c302f3eb
0:1:9004f47f8f2f7e5811947ac3969d2c78bdaed441
1:1:8a0422b3b2656a38c0b9522d07f6e9b6fcbb037b
2:1:399fbb1c2c17c320ac4f14bd8f8d0a42c3declee
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DP
ROT/practiques/practica2/exercice_2$ cat proof_doc5.txt
MerkleTree:sha1:00:01:6:4:772091d12e956cc60c666307fb7c7ef7c302f3eb
0:4:616b8b02c6e89c6547b1be6ce5394bf88305fd4d
1:3:
2:0:48c6a1092af77c5a48e87c53fd21f606e86c4d95

```

Figura 5: Proof file for document 0 and 5.

Merkle tree proof verifier

This section describes the script `proof_verifier.sh` that can be found at github.com/Algafix/problemes-dprot/blob/master/practica2/exercice_2/proof_verifier.sh.

This script accepts 2 parameters, the document position and the verification file. First, it reads the public Merkle tree information from the first line of the verification file and computes the hash of the document to be verified (it assumes that the document is in the `docs/` directory with the appropriate naming schema).

```
1 doc_k=$1
2 IFS=":";
3 read -r name alg doc_pre node_pre n max_i root < $2;
4
5 read -r doc_content < docs/doc${doc_k}.dat
6 hash=$(echo -n -e "\x${doc_pre}${doc_content}" | openssl dgst -
    sha1 -binary | xxd -p)
```

Then, for every line of the verification document and knowing the position of the document to be verified in the tree, it computes every hash until the root. It uses the same idea of knowing if the new j variable is odd or even to determine the position of the nodes in the hash function. The first read discards the first line because it doesn't contain node info.

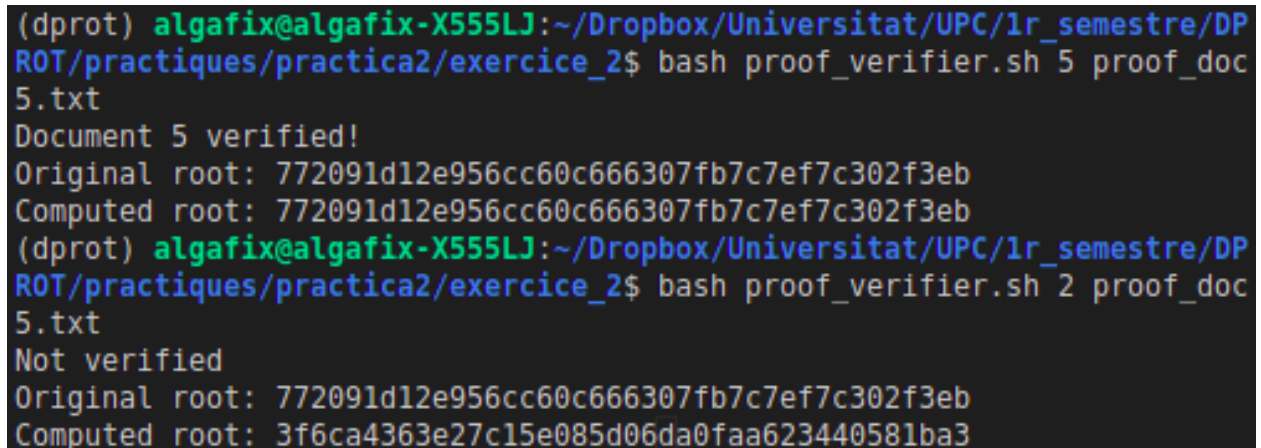
```
1 {
2 read
3 while IFS=":"; read v_i v_j node_hash;
4 do
5     if [ $((j%2)) -eq 0 ]
6     then
7         hash=$(writehex ${node_pre}${hash}${node_hash} |
8             openssl dgst -sha1 -binary | xxd -p)
9     else
10        hash=$(writehex ${node_pre}${node_hash}${hash} |
11            openssl dgst -sha1 -binary | xxd -p)
12    fi
13    j=$((j/2))
14    i=$((i+1))
15 done;
16 } < $2
```

The function `writehex` allows us to convert the ASCII text hash read from the file to a byte

version to pipe to the openssl module.

```
1 writehex  ()
2 {
3     local i
4     while [ "$1" ]; do
5         for ((i=0; i<${#1}; i+=2))
6         do
7             printf "\x${1:i:2}";
8             done;
9             shift;
10        done
11 }
```

The result of a successful verification and a failed one as well with the commands is shown in Figure 6.



The image shows a terminal window with two verification attempts. The first attempt uses document 5 and succeeds, showing matching original and computed roots. The second attempt uses document 2 and fails, showing a mismatch between the original and computed roots.

```
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DP
ROT/practiques/practica2/exercice_2$ bash proof_verifier.sh 5 proof_doc
5.txt
Document 5 verified!
Original root: 772091d12e956cc60c666307fb7c7ef7c302f3eb
Computed root: 772091d12e956cc60c666307fb7c7ef7c302f3eb
(dprot) algafix@algafix-X555LJ:~/Dropbox/Universitat/UPC/1r_semestre/DP
ROT/practiques/practica2/exercice_2$ bash proof_verifier.sh 2 proof_doc
5.txt
Not verified
Original root: 772091d12e956cc60c666307fb7c7ef7c302f3eb
Computed root: 3f6ca4363e27c15e085d06da0faa623440581ba3
```

Figura 6: Verification of different documents.