Lab Work 2

Data Protection
Master in Cyber Security (UPC) 2025
Jorge L. Villar

Last updated: Sep 1 13:22:11 2025

Contents

1 Using hash functions in practice

The most commonly used hash functions are probably implemented in any personal computer. In a typical Linux installation, the following command line tools compute the hash digest of a file:

```
$ md5sum foo.dat
```

It uses the hash function MD5 (nowadays considered insecure) to compute the digest of the file foo.dat and output the result in a printable (text) format. Similar commands exist for the other common hash functions, like `shasum` or `sha1sum`, `sha256sum` and `sha3sum`. Probably the last one needs the installation of extra packages.

Another example of use of the previous command is

```
$ echo -n | md5sum
```

that produces the output

```
d41d8cd98f00b204e9800998ecf8427e  -
```

corresponding to the MD5 value of the empty file.

You can also compute the different hash values for a collection of files. For instance,

```
$ md5sum *
```

will produce a list of all the files in the working directory along with with their corresponding hashes. For more details, refer to the `man` pages of the commands.

You can also use the command line tool `openssl` to compute digests of files. For instance,

```
$ echo -n 'Hello world!'  | openssl dgst -md5 -binary | xxd
```

computes the MD5 value corresponding to the ASCII string "Hello world!". The result produced by the call to `openssl` is in binary format, and in the example it is processed by `xxd` to produce the following printable output:

```
00000000:  86fb 269d 190d 2c85 f6e0 468c eca4 2a20  ..&...,...F...*
```

There are other options of `openssl dgst` (used instead of `-binary`) that produces different output formats (e.g., `-hex` produces a printable hexadecimal output).

You can also select other (more secure) hash functions by replacing `-md5` with `-sha1`, `-sha256`, etc.

Another way to use hash functions with OpenSSL is directly accessing their implementation in the OpenSSL libraries from your C or C++ programs. In that case, you will need the corresponding C header files, that can be installed in your computer from the OpenSSL development package, with

```
$ sudo apt-get install libssl-dev
```

For instance, the header file for the MD5 implementation can be accessed in your C program with

```
#include <openssl/md5.h>
```

2 Using Message Authentication Codes in OpenSSL

Some well-known message authentication codes, like CMAC or HMAC, are implemented in OpenSSL. For instance, you can compute the CMAC of a message with the following command:

```
$ mykey=00112233445566778899aabbccddeeff

$ echo -n 'Hello world!'  | openssl dgst -mac cmac -macopt cipher:aes-128-cbc -macopt hexkey:$mykey
```

which produces the output

```
(stdin)= fcae3b5af732a72611ce06b9f680db72
```

The option `-macopt` is used to specify the parameters of CMAC, like the block cipher and the key, with the syntax `-macopt name:value`. The length of the key must match the one required by the specified block cipher.

If you want to compute the CMAC of a file (not the standard input), you only need to provide the filename:

```
$ openssl dgst -mac cmac -macopt cipher:aes-128-cbc -macopt hexkey:$mykey -binary foo.dat |
xxd
```

In this example, the call to `openssl` produces binary output, and it is converted to a printable representation with `xxd`.

The use of HMAC is simpler, because it is the default choice for a MAC algorithm in OpenSSL.

```
$ echo -n 'Hello world!'  | openssl dgst -hmac "My secret key"
```

Here, the key is specified, as an ASCII string, in the `-hmac` option, and there is no restriction on its size (because of the flexible specification of the algorithm HMAC).

The key can be also specified in hexadecimal representation with a syntax similar to the CMAC example:

```
$ echo -n 'Hello world!'  | openssl dgst -mac hmac -md5 -macopt hexkey:$mykey
```

3 Implementing some insecure MACs

Insecure MAC designs like CBC-MAC for arbitrarily long messages or the naïve "one-pass" HMAC (a simplification of the actual HMAC) are not directly implemented in OpenSSL, but they can be built from the implemented hashes and ciphers. The simplified version of HMAC, that computes the tag of a message m as only the hash of the concatenation of the key and the message itself, can be implemented with a single call to the command line tool:

```
$ cat key.dat message.dat | openssl -dgst -md5 -binary > tag.dat
```

Here, from the binary files containing the key and the message, the computed tag is saved in the file `tag.dat`.

Similarly, you can just use encryption in CBC mode to obtain a CBC-MAC implementation, by just selecting the last ciphertext block as the tag. For instance, the AES-128-CBC-MAC tag of the file `message.dat` with the AES-128 key contained in `key.dat` can be computed with

```
$ openssl enc -aes-128-cbc -K ‘cat hexkey.dat‘ -iv 0 -in message.dat | tail -c 16 > tag.dat
```

where a zero initializacion value is used, as specified in the definition of CBC-MAC, and the file `hexkey.dat` contains the hexadecimal representation of the binary file `key.dat`. You can build this file with

```
$ xxd -p key.dat hexkey.dat
```

The last part of the command line, `tail -c 16`, takes only the last 16 bytes of the resulting CBC encryption, that corresponds to the last 128-bit block of it.

With these two custom implementation of the one-pass HMAC and CBC-MAC, you can recreate the two well-known length extension forgery attacks.

4 Practical work: recreation of known MAC forgery attacks

4.1 CBC-MAC concatenation attack

CBC-MAC is insecure if used with variable length messages.

Recreate the attack, with the following steps:

- Create a random AES-128 key (which is just a random string of 16 bytes), and choose two arbitrary messages. For instance, message1 is:

  "What about joining me tomorrow for dinner?",

  and message2 is:

  "Oops, Sorry, I just remember that I have a meeting very soon in the morning.".

  Probably, the two messages came from completely different contexts (perhaps with different recipients).

- For simplicity, assume that the system adds a header to the messages consisting of 16 zero bytes. Then, create two files mess1.dat and mess2.dat with the previous text messages, and a file head.dat with 16 zero bytes.

- Generate the corresponding AES-128-CBC-MACs for the two messages with headers and store them in the files tag1.dat and tag2.dat.

- Investigate the padding that AES-128-CBC introduces in the last incomplete block. To do that, you can encrypt a message with AES-128-CBC, and then decrypt the result with the option `-nopad`. You will recover the padded version of the message (do it with the first message).

  The encryption/decryption lines to investigate the padding would be something like

  ```
  $ openssl enc -aes-128-cbc -K $key -iv 0 -in message.dat -out cipher.dat

  $ openssl enc -d -aes-128-cbc -K $key -iv 0 -nopad -in cipher.dat -out padded.dat

  $ xxd padded.dat
  ```

  In the second line, the `-d` switch means decryption, and `-nopad` assumes no padding was used and does not discard any byte of the resulting plaintext.

  The padding scheme used here is not the one I've explained in class. Namely, if there are n missing bytes in the block, then the padding is n bytes all with the value n. If the last block is complete, then add an entire block with all its bytes equal to the block length in bytes (16 or 0x10 in the case of AES-128).

- Create the forgery by appending the files head.dat, mess1.dat, the necessary padding for the first message, the tag tag1.dat and the second message mess2.dat. Store it into forgery.dat. Compute the CBC-MAC of the resulting file and check whether it is exactly the same as tag2.dat.

- You have got a forged message/tag pair.

  The file forgery.dat contains some non-printable characters, but if you print it, it probably gives the impression that the forged message is the concatenation of the two original messages, which substantially changes the content of the first one.

In a sort of practical attack you, as the adversary, will be given the plaintexts (including the header) and the two tags, but not the key. Then, the padding for the first message can be easily inferred from the length of the missing part in the last incomplete message block, and the first 16 bytes of the second message must be XORed with tag1. To avoid computing this XOR operation, in this recreation the all-zero header is prepended to the messages (and therefore, XORing is just replacing the header by the tag itself).

4.2 One-pass HMAC length extension attack

In a quite similar way, the one-pass HMAC can be easily forged witout knowing the key. But the forgery is even more general than in CBC-MAC. Indeed, the attacker knows only a message mess1.dat (no header is necessary here) and the corresponding tag tag1.dat. Any string can be appended to mess1.dat in a way that the adversary can compute the new valid tag for the concatenation of messages. The only limitation is that the concatenated string must start with the padding that the hash function adds to mess1.dat.

The main problem here is the difficulty to continue the Merkle-Damgård chain from scratch. There are two ways to do that:

1. Implement the compression function and the Merkle-Damgård chain (that means implementing the whole hash function).
2. Tweak the code in the OpenSSL library (or use low-level access to it)

The second approach is simpler if you have the development version of the libraries installed (to have access to the C header files), which would require the installation of the package `libssl-dev`.

The computation of a hash function in OpenSSL consists of three steps: initialize, update, finalize. All the internal state of the computation is placed into a context object, that you should not access directly (but you can!). The initialize step sets up the initial values of the internal state of the hash function (like the iv specified in the Merkle-Damgård construction). The update step is called every time new sections of data are added to the hash computation. This updates the internal state. The finalize step is called after the last message block is processed by the update step. Then, the necessary padding is added and the hash value is extracted from the internal state.

For the MD5 hash function, you can access and manually modify the internal state of the hash function. In particular, you can set this internal state as if the MD5 function were in the intermediate state corresponding to any desired point in the computation of the hash of a long message. Moreover, given the MD5 hash value of mess1.dat and the total length of the key (the HMAC key) and the message, it is straightforward to reconstruct the last internal state of the computation.

From this point, you can continue performing the update step as many times as you want with the new appended data, and then the finalize step will give you the final hash value. This value is the one-pass HMAC of the concatenation of messages, computed without knowing the key!

In MD5, the padding string takes the form `0x80 0x00 ... 0x00 n0 n1 ... n7`, where the last 8 bytes encode the length in bits of the unpadded message.

The forged message is the concatenation of mess1.dat, the padding string and the new appended data. Again, the padding string could add strange nonprintable characters that, in practice, make the forgery look suspicious.

Implementing this attack is challenging because you have to deal with the low-level implementation of the hash function, and also there could be some issues about the byte-order of the integer representation (for the length encoding), which depends on the particular architecture of your computer (integers can be represented in either big-endian or little-endian byte order!).

As an example of how the MD5 initialization tweaking code looks like, the following function is intended to be called just after `MD5_Init(MD5_CTX *)`:

```
void set_ctx(MD5_CTX *pctx, const char *digest, unsigned long nblocks) {
        pctx->A = gethexword32(digest);
        pctx->B = gethexword32(digest+8);
        pctx->C = gethexword32(digest+16);
        pctx->D = gethexword32(digest+24);
        nblocks <<= 9; // converting into bits
        pctx->Nh = nblocks>>32;
        pctx->Nl = nblocks&0xFFFFFFFFul;
```

}

where `digest` is a string containing the hexadecimal representation of the MD5 hash value of mess1.dat, and `nblocks` is the total number of data blocks (512 bits each) of the concatenation of the key and mess1.dat and the corresponding padding (that in some cases can add an extra block).

The function `MD5_LONG gethexword32(const char *)` (that also needs to be implemented) reads a 32-bit integer value from its hexadecimal representation. It could be replaced by `strtod()` but the difference is that `gethexword32()` reads exactly 8 hexadecimal digits and stops, and the bytes must be interpreted in little-endian order (not in network order: the first byte is the least significant one). For instance, the string `"12efc5ca"` is interpreted as the number `0xcac5ef12`.

5 Practical work: Building Merkle hash trees

It is quite easy to implement the Merkle hash tree functionality from the previously described tools. All you need is to organize the different calls to the corresponding hash function. Indeed, you only need two types of hash computations:

- Compute the hash of a document doc3.dat with something like

  ```
  $ openssl dgst -sha1 -in doc3.dat -binary > node0.3
  ```

- Aggregate two existing hashes node2.4, node2.5 into one single value node3.2 with

  ```
  $ cat node2.4 node2.5 | openssl dgst -sha1 -binary > node3.2
  ```

To prevent some possible attacks that mixes hashes of nodes and documents, you chan choose two different prefixes (e.g., 0x35 for documents and 0xE8 for nodes) and prepend the corresponding one to the data to be hashed.

```
$ cat doc.pre doc3.dat | openssl dgst -sha1 -binary > node0.3
```

```
$ cat node.pre node2.4 node2.5 | openssl dgst -sha1 -binary > node3.2
```

where doc.pre and node.pre contain the selected prefixes.

In the previous examples the following labelling convention is taken (parenthesized expressions like $(i-1)$ are replaced by their corresponding values):

- Documents to be hashed are named doc0.dat, doc1.dat, ..., doc$(n-1)$.dat, assuming there are n documents in total.
- Tree nodes at level i (level 0 corresponds to the leaves) are named node$i.j$, where the hash contained in node$i.j$ is computed from the nodes node$(i-1).(2j)$ and node$(i-1).(2j+1)$.
- Node node$i.j$ is computed only whenever node$(i-1).(2j)$ exists.
- If node$(i-1).(2j+1)$ does not exist, then it is replaced by the empty file in the computation of node$i.j$. Or simply, the missing file is removed from the hash computation, as for example,

```
$ cat node.pre node2.4 | openssl dgst -sha1 -binary > node3.2
```

Each tree layer has about half of the nodes than the previous layer. The root node is obtained when a layer has only one node, and this is the hash value of the collection of documents.

You can add a new node to the tree by just naming it as docn.dat and updating the corresponding nodes (including the root). When n is an exact power of 2, then you will need to add a new layer into the tree with a new root, along with some new nodes from it to the new leave. For instance, if a fifth document doc4.dat is added to the Merkle tree of four documents, then the old root is node2.0 and the root of the new tree is node3.0, and the new nodes node2.1, node1.2 and node0.4 are added to the tree.

On the contrary, removing a document implies recomputing the whole tree from scratch (we will not consider document removal in this implementation).

One interesting feature of Merkle hash trees is that the tree creator stores the whole tree, but she only needs to send the root node to other parties. Then at some point she can convince anyone that a certain document belongs to the collection and it is at a specific position by sending a limited number of node values (the necessary nodes to allow the computation of all the intermediate hashes in the path from the document itself to the root). For instance, in a collection of 4 documents she wants to show that a document test.dat belongs to that collection and it is at the fourth position. The nodes that she must open are node0.2 and node1.0, because then the verifier can compute the values corresponding to node0.3 (the hash of the document, properly prefixed), node1.1 (from node0.2 and node0.3) and node2.0 (from node1.0 and node 1.1), and compare the last with the hash of the collection.

As a practical work, you can try to implement the tree building, the tree updating (only adding a new document) and the generation of the proof of membership and position.

- Start from a list of n arbitrary nodes, and name them doc0.dat, doc1.dat, ...
- Choose two prefixes, one for the computation of the hashes of the documents (leaves of the Merkle tree), and the other for the other tree nodes. You can store the documents and the node files into separate folders (e.g., docs and nodes).
- Write a script or program that on input n it computes all hashes and stores them into the files named node$i.j$ for all suitable values of i and j. Remember that you would need to deal with non-existent nodes (e.g., node$(i-1).(2j)$ exists but node$(i-1).(2j+1)$ does not).
- Create a text file with one line per node, containing in each line the values of i, j and the node hash (i.e., the contents of node$i.j$ in hexadecimal). The file can have a header (e.g., the first line) containing the hash algorithm used (e.g. sha1), the two selected prefixes (in hexadecimal), the number n of documents (decimal number), the depth of the tree (the number of layers including the leaves and the root, as a decimal number) and the root hash (the contents of the tree root in hexadecimal):

```
MerkleTree:sha1:3C3C3C3C:F5F5F5F5:12:5:3f786850e387550fdab836ed7e6dc881de23001b
0:0:89e6c98d92887913cadf06b2adb97f26cde4849b
0:1:2b66fd261ee5c6cfc8de7fa466bab600bcfe4f69
...
4:0:3f786850e387550fdab836ed7e6dc881de23001b
```

- The first line is the public information of the Merkle tree, and the remaining lines give the private information that allows updating the tree and producing proofs of document membership.
- Write a script or program that inserts a new document in the tree, adding and modifying the necessary nodes and updating the text file describing the tree.

- Write a script or program that produces a proof of membership for a given document doc.dat at position k. The proof consists of a list of nodes that allows the verifier to recompute the hashes in the path from the k-th leave to the root and check that the root value is the same as the given one (the verifier was given in advance the public information of the tree). The proof can be just the corresponding lines of the Merkle tree text file description.
- Write a script or program that verifies the proof given the public information of the Merkle tree (first line of the text file), the document and the proof (text file with the necessary nodes).
