Data Integrity & Authentication - MACs

The goal is to ensure integrity of messages, even in presence of an attacker in the middle who sends own bad messages. So what we want is to be able to recognize bad messages and discard them once detected they are malicious. We have to remember that we are dealing now with integrity/authentication, that is ortogonal to secrecy/confidentiality, yet systems often require to provide both.

Data Integrity – Definition

We need an authentication algorithm *A*, a verification algorithm *V* and a shared key *k*. Now, when Alice has to send a message *m* to Bob, she computes $A_k(m)$, that is the authentication algorithm with *m* as input on the key *k*, getting as result a ***digest***, an ***authentication tag***. Finally Alice sends to Bob the pair $(m, A_k(m))$. On the receiver's side, Bob will run the verification algorithm *V*, that states: compute $A_k(m)$, if the outcome is equal to the authentication tag received by Alice, the message is okey so *accept* it, otherwise it is garbage so *reject* it.

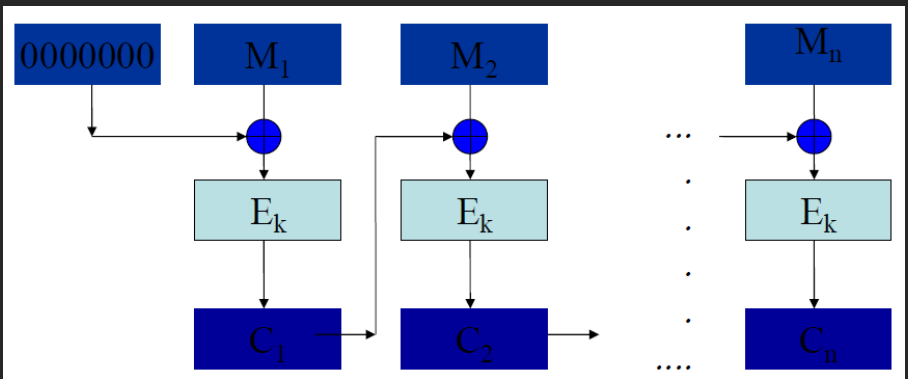
The Authentication Algorithm is called **MAC** (Message Authentication Code), so from now on it will be called MAC. The authentication tag will often be denoted by *t*. One important thing is that MAC functions are not 1-to-1 since the message space is potentially infinite while the digest space is smaller and of fixed size, and so MAC is not an injective function. From this follows that there can be two messages having the same authentication tag. From the adversay's point of view, it has to be hard to compute a legal pair $(m, MAC_k(m))$ without knowing the key, and this has to be hard even knowing n possible legal pairs. However, if the attacker succeeds in finding a valid pair, he has succeeded in performing a *forgery attack*. This was the concept of forgery. Actually there are different types of forgery, we will see it. So for now we know that if the attacker succedes in finding a legal pair, even if the message is meaningless, he has succesfully performed a forgery attack. Now, there are two patterns (maybe more, we study only 2) for creating a MAC function, and they are:

1. by means of a block-cipher encryptor (CBC-MAC);
2. by means of hash functions (HMAC).

Let's see them.

CBC-MAC

It is basically the CBC approach, we use the CBC mode of operation on the message m with the seed composed by all zeroes, and our authentication tag will be the last block-cipher.



From the receiver side, Bob will just compute the same encryption scheme that Alice did with the message m. Then if the C_n that Bob gets as the last block will be equal to the one sent by Alice, he will trust the message.

Note that the secret key is used in the encryptors, so this scheme provides both *data integrity* and *authentication*.

Security of CBC-MAC

Claim: if E_k is a pseudo random function, then the CBC-MAC used on fixed-length messages is resilient to forgery

CBC-MAC attack on variable length messages

For what concerns variable-length messages, an attack can be performed and it is the following: suppose the attacker knows two legal pairs (m,t) and (m',t'), then he can generate a new legal pair where the authentication tag will be t'. The attack can be performed by simply choosing the message:

$$m||(m_1' \oplus t)\ ||m_2'||\dots||m_x'$$

This because watching the scheme, at the end of the first message the tag is t, if we cuncatenate the message we have shown, what the CBC would do is to perform the $\oplus$ between t and $(m_1' \oplus t)$ getting as result again t, and this means that the final tag will be the same as t', whatever is m. Therefore the attacker is able to perform a forgery attack without knowing the key.

Combined Secrecy & CBC-MAC: we said that integrity/authentication and secrecy/condifentiality are two ortognal concepts. However one can decide to have both of them by simply sending the encrypted message togheter with the authentication tag. In order to achieve this goal, it is sufficient to take the message, performing $t=\text{CBC-MAC}_{k1}(m)$ on a key *k1* and choosing t as tag, then encrypting m with CBC mode on another key *k2* getting the ciphertext C, and then sending the pair (C,t). The receiver will first decrypt C in order to get m, then he will run the verification algorithm in order to decide wheter to accept or not the message.

MAC with Hash Functions

Recall that an hash function is a function which goal is to map a large domain D in a smaller one R. We now want to build MACs on has functions in order to get our authentication tags t. For the beginning we will not deal with *keyed-hash-functions*, so we will use hash functions without the needing of a secret key. Another thing that we have to keep in mind is that dealing with hash functions, since domains are of different size, collisions cannot be avoided, and in many cases the difference between the sizes of domain and codomain can have even 7-8 order of magnitude of difference, meaning that many messages will have the same authentication tag and this can be exploited by the attacker.

Collision resistance

We have two kinds of collision resistance: **strong** and **weak**. Here are the definitions:

- a hash function $h : D \rightarrow R$ is called *weakly collision resistant* for a message $x \in D$ if it is hard to find a message $x' \neq x$ s.t. $h(x) = h(x')$
- a hash function $h : D \rightarrow R$ is called *strongly collision resistant* if it is hard to find two messages x and x' s.t. $h(x) = h(x')$

The difference is that *weak resistance* assumes that a message x is given, and we have to find x', while with *strong resistance* we have to find both x and x' from scrath.

Claim. *strong collision resistance* $\Rightarrow$ *weak collision resistance*

Proof. we prove it by proving that $\neg$*weak* $\Rightarrow$ $\neg$*strong*. Assume our funcion h is not weak resistent. So we can imagine we have an algorithm A that takes as input a message x and returns a message x' s.t. x and x' collide. Then we can construct an algorithm B that takes nothing as input and performs the following operations:

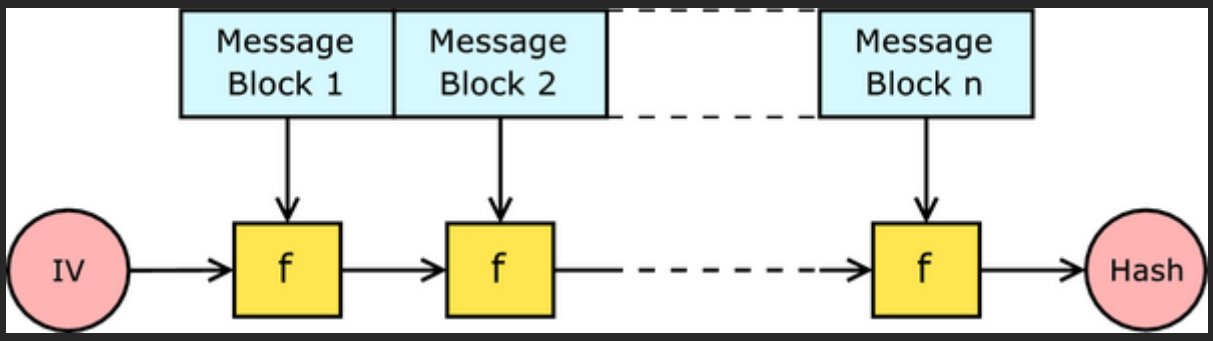
1. choose a random x ∈ D
2. outcome (x, A(x))

So at the end what happens is that we are easly able to find a pair of messages (x,x') which collide and this means to be ¬*strong*, and thus we have prooved that ¬*weak* ⇒ ¬*strong*, i.e. *strong* ⇒ *weak*.

Birthday paradox: given h : D → R, if we randomly choose $1.1774|R|^{1/2}$ elements of D then the probability that two of them collide is 0.5.

Please note that the number of choosen messages only depends on R, whatever is the size of D. Therefore given a tag, let's say it has n bits, we have 2^n possible tags, then we can choose $k \approx 2^{n/2}$ random messages in the domain and we get a probability of 0.5 to have found a collidant pair. This is known as birthday attack. WARNING: this does not mean that I can choose a fixed message and hope to find its collidant one with $2^{n/2}$ tries, this does not work. We should go bruteforce until we get the first collidant one.

Merkle–Damgård construction



This is a mode of operation in order to get a digest from a document. Imagine we have the document (cyan blocks in the picture) and an hashing function that uses a domain of n bits and a codomain of m bits. This means that every input for a block f has to take as input n bits and outcomes m bit. Of the n bits of the input block, m are coming from the previous block, and therefore n - m have to come from the next block. If we know n and m and we know the size of the message we can understand how many blocks we need. Every block of the document will have the size of n - m. Once we know the message size, we just divide the size by the size of the blocks (so size/(n-m)) and we get how many blocks we need. For the first block we will need some **seed**. The final result will be the hash of the message, so the digest, the authentication tag. Note that all this does not require any key (as long as the hash block doesn't).

Keyed Hashing Function

The concept is to combine messages, keys and hash functions in order to produce MAC. Why adding a key to an hash function since it works well without keys by its nature? The reason is easy understandable, and we are going to understand why. Another important thing that we may notice is that keyed hash functions provide the particular proprety to ensure both data integrity and authentication, the first comes from the usage of the hash function itself as we have seen so far, the second comes from the fact that since hash function now requires the key, and the key is shared only between Alice and Bob, it means that if Bob notices that the authentication is valid, it means that only Alice could produce it because only Alice had the key in order to do that.
Hash functions have been designed to take as input a message, more precisely a string, they do not offer the possibility to take 2 inputs (**Merkle–Damgård** seems to do so, but actually each block takes as input the cuncatenation between previous m bits of the digest and current n-m bits of the document), therefore if we

want the possibility to use keys with hashing functions we have to mix the message and the key, by cuncatenations. Now we are going to see some patterns that use key combined with message.

- $MAC_k(m) = h(k||m)$ unsecure, this pattern suffers the *length-extension attack*. The vulnerable hashing functions work by taking the input message, and using it to transform an internal state. After all of the input has been processed, the hash digest is generated by outputting the internal state of the function. It is possible to reconstruct the internal state from the hash digest, which can then be used to process the new data. In this way one may extend the message and compute the hash that is a valid signature for the new message. So the attacker just has to reconstruct the internal state and then he can add to the message whatever he wants. For further details see the following page: *https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks*
- $MAC_k(m) = h(m||k)$ this solves the previous problem, but it has another problem, it suffers from the problem that an attacker who can find collision in the (unkeyed) hash function has a collision in the MAC (as two messages m1 and m2 yielding the same hash will provide the same start condition to the hash function before the appended key is hashed, hence the final hash will be the same). In few words this method suffers from the birthday paradox, and pay attention that if the attacker finds a collision, this collision works for all the keys since the key is appendend at the end of the message
- $MAC_k(m) = h(k||m||k)$ okey, good! This parrern seems to be immune to previous issues

SHA-1

As any hash function, SHA-1 produces a *digest* of fixed length starting from a message of variable length. SHA-1 produces a digest of 160 bit of length, starting from a message which maximum length can be 2^{64}-1 bit (we are going to understand why). Here is the description:

Phase 1 (Padding) : to the original message they are added padding bits until the total length of the message in bits is a number congruent to 448 mod 512 (becaue simply the last 64 bits are reserved since they specify the message length).

Phase2 (Adding length) : to the final message (original message + padding) it is added an unsigned integer of 64 bits denothing the full length of the message. Now the full message is a multiple of 512 bit.

Phase 3 (MD buffer initialization) : a 160 bit buffer is subdivided in 5 registers, each one of 32 bits, they are:

- A = 67452301
- B = EFCDAB89
- C = 98BADCFE
- D = 10325476
- E = C3D2E1F0

Phase 4 (blocks elaboration) : now that we have the 5 registers, we can start computing hashing. The complete message (which length is a multiple of 512 bits) is subdivided into several blocks of 512 bits. The core of the SHA-1 algorithm is the *compression* function which is composed by 4 cycles of 20 steps each one. Starting from the first block of 512 bit, we give as input to the *compression function* the block, a costant K and the values of the five registers. At the end of the compression function a new state of the five registers is observed, and we go on with the next block. When the last block of the message is compressed, the final values of the five registers form the digest, so the outcome of SHA-1 will be the digest composed by A|B|C|D|E.

HMAC

Proposed in 1996 by Bellare-Canetti-Krawczyk and approved by Interneet Engineering Task Force RFC 2104, is a *keyed-hashing* MAC which uses a pattern similar to the last one we listed. Please note that it is not an

hash-algorithm (like MD5, SHA-1…) but it is a 'mode of operation', which can use inside an *hash-algorithm* like SHA-1.
The definition is the following:
HMAC receives as input a message *m*, a key *k* and an hash function *h*, then it computes:

$$HMAC_k(m,h) = h(k \oplus opad \,||\, h(k \oplus ipad \,||\, m))$$

where:
opad = 0x5c5c5c…5c5c
ipad = 0x363636…3636
both of length of one block.
One thing that can be noted is that if I find two messages that collide in the inner hash $h(k \oplus ipad \,||\, m)$, they will collide for the whole hashing scheme because the outer hash adds the same stuff to both the messages. Therefore HMAC still suffers from birthday-paradox, however the attacker cannot understand which are the colliding pairs: he is able to find two strings k1 and k2 which collide, i.e h(k1) = h(k2) but k1 and k2 will have the form $k \oplus ipad \,||\, m1$ and $k \oplus ipad \,||\, m2$ but since he does not know the key k, he cannot retrive m1 from from $k \oplus ipad \,||\, m1$ neither m2 from form $k \oplus ipad \,||\, m2$.