Message Authentication Code

Gianluca Dini
Dept. of Ingegneria dell'Informazione
University of Pisa
Emai: gianluca.dini@unipi.it
Version: 2022-04-02

1

Message Authentication Codes (MACs)

PRINCIPLES OF MACS

apr. '23 Hash functions 2

2

Message Authentication Code

- Synonims
  - Cryptographic checksum
  - Keyed hash function
- Similarly to digital signatures, MACs provide message authentication and integrity
- Unlike digital signatures, MACs are symmetric schemes and do not provide nonrepudiation
- MACs are much faster than digital signatures

Communication model

k k

Alice and Bob share a secret key k

MAC Generation S: $\{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^m$ ➔ $t \leftarrow S(k, x)$

MAC Verification V: $\{0,1\}^* \times \{0,1\}^k \times \{0,1\}^m \rightarrow$ Boolean ➔ $\{true, false\} \leftarrow V(x, k, t)$

MAC generation
$t = S(k, x)$

MAC verification
$t' = MAC(k, x)$
return $(t == t')$

Alice and Bob want to be assured that any manipulations of a message m in transit are detected

Message Authentication Code (MAC)

- A MAC is defined by (Gen, Mac, Vrfy)
  - Gen takes as input 1^n and outputs a key k
  - Mac takes an input a key k and a message $x \in \{0, 1\}^*$ and outputs a tag t, s.t. $t = Mac_k(x)$
  - Vrfy takes as input a key k, a message x and a tag t and returns true or false
- Consistency property
  - For all key k and message x, $Vrfy_k(x, Mac_k(x))$ = true

apr. '23 Hash functions 5

5

Properties of MACs (→)

- Cryptographic checksum
  - A MAC generates a cryptographically secure authentication tag for a given message.
- Symmetric
  - MACs are based on secret symmetric keys. The signing and verifying parties must share a secret key.
- Arbitrary message size
  - MACs accept messages of arbitrary length.
- Fixed output length
  - MACs generate fixed-size authentication tags.

apr. '23 Hash functions 6

6

Properties of MACs

- Message integrity
  – MACs provide message integrity: Any manipulations of a message during transit will be detected by the receiver.
- Message authentication
  – The receiving party is assured of the origin of the message.
- No nonrepudiation
  – Since MACs are based on symmetric principles, they do not provide nonrepudiation

Security

- Threat model
  – Adaptive chosen-message attack
  – Assume the attacker can induce the sender to authenticate messages of the attacker’s choice
- Security goal
  – Existential unforgeability
  – Attacker should be unable to forge a valid tag on any message not authenticated by the sender

Security

- Computation-resistance (chosen message attack)
  - For each key k, given zero o more (x_i, t_i) pairs, where t_i = S(k, x_i), it is computationally infeasible to compute (x, t), s.t. t = S(k, x), for any new input x ≠ x_i (including possible t = t_i for some i)
    - Adaptive chosen-message attack
    - Existential forgery

apr. '23 Hash functions 9

9

Replay

- Mac does not prevent replay
  - No stateless mechanism can
- Replay attack can be a significant real-world concern
- Need to protect against replay at a higher layer

apr. '23 Hash functions 10

10

Types of forgery

- Selective forgery
  - Attacks whereby an adversary is able to produce a new text-MAC pair for a text of his choice (or perhaps partially under his control)
    - Note that here the selected value is the text for which a MAC is forged, whereas in a chosen-text attack the chosen value is the text of a text-MAC pair used for analytical purposes (e.g., to forge a MAC on a distinct text).
- Existential forgery
  - Attacks whereby an adversary is able to produce a new text-MAC pair, but with no control over the value of that text.

Implications of a secure MAC

- FACT 1 - Computation resistance ➔ key non-recovery (but not vice versa)
  - It must be computationally infeasible to compute k from (x_i, t_i)s
  - However, it may be possible to forge a tag without knowing the key

Implications of a secure MAC

- FACT 2 **-** Attacker cannot produce a valid tag for any new message
  - Given (x, t), attacker cannot even produce (x, t’) –a collision– for t’ ≠ t

Implications of a secure MAC

- FACT 3 - For an adversary not knowing k
  - S must be 2nd-preimage and collision resistant;
  - S must be preimage resistant w.r.t. a chosen-text attack;
- FACT 4 - Secure MAC definition says nothing about preimage and 2nd-preimage for parties knowing k
  - Mutual trust model

How to use MACs in practice

- In combination with encryption
  - x: plaintext; t: tag; y: ciphertext; z: transmitted message; ek: encryption key; ak: MAC key (authentication key)
  - Option 1 (SSL)
    - $t = S_{ak}(x)$; $y = E_{ek}(x \mid\mid t)$, $z = c$
  - Option 2 (IpSec)
    - $y = E_{ek}(x)$; $t = S_{ak}(c)$; $z = y \mid\mid t$
  - Option 3 (SSH)
    - $y = E_{ek}(x)$; $t = S_{ak}(x)$; $z = y \mid\mid t$

apr. '23 Hash functions 15

15

Other uses

- One-time password
  - Based on time-syncronization
  - Based on challenge-response

apr. '23 Hash functions 16

16

Message Authentication Codes (MACs)

HOW TO BUILD A MAC

How to build a MAC

- From Block Ciphers (more in general from PRF)
  - CBC-MAC
  - NMAC
  - PMAC
- From a hash functions
  - HMAC

HMAC

How to build a MAC from ah hash function

- Insecure constructions
  - Secret prefix scheme
    - S(k, x) = H(k||x), H hash function
  - Secret suffix scheme
    - S(k, x) = H(x || k), H hash function
  - Forgery is possible in both cases
  - HMAC construction is necessary

apr. '23 Hash functions 19

19

Insecurity of prefix scheme

- Let x = (x_1, x_2, x_3, …, x_n)
- Let t = S(k, x) = H(k || x_1,x_2, …, x_n)
- Existential forgery attack
  - *Objective*: construct t' of x' = x_1, x_2,… x_n, x_{n+1} without knowing k (x_{n+1}: additional block)
  - *Assumption*: H follows the Merkle-Damgard scheme
  - *The attack:* t' = h(x_{n+1}, t) with h compression function
  - Forging (x', t') only needs the previous hash output t (but not k)

apr. '23 Hash functions 20

20

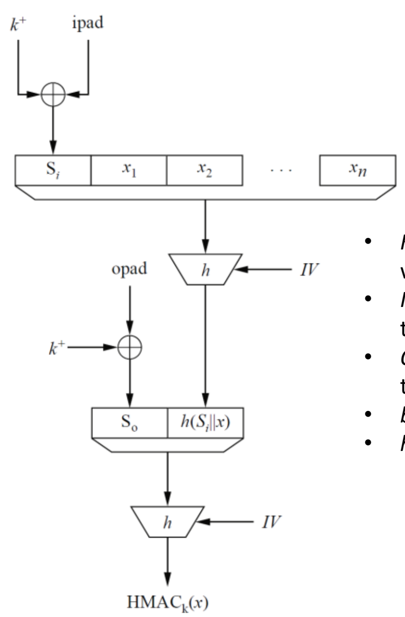
Insecurity of the suffix scheme

- Let t = S(k, x) = H(x || k)
- Existential forgery attack
  - *Objective*: Construct t' of a x' without knowing the key k
  - *Assumption*: H follows the Merkle-Damgard scheme
  - *The attack*
    - Assume the adversary is able to find a collision H(x) = H(x')
    - Then, t = h(H(x), k) = h(H(x'), k), thus t' = t, where *h* compression function

apr. '23 Hash functions 21

21

HMAC



- *K+*: padded key (key K padded with 0 up to b bit)
- *Ipad* = 0x36 repeated b/8 times
- *Opad* = ox5C repeated b/8 times
- *b* block size
- *h* incorporated hash function

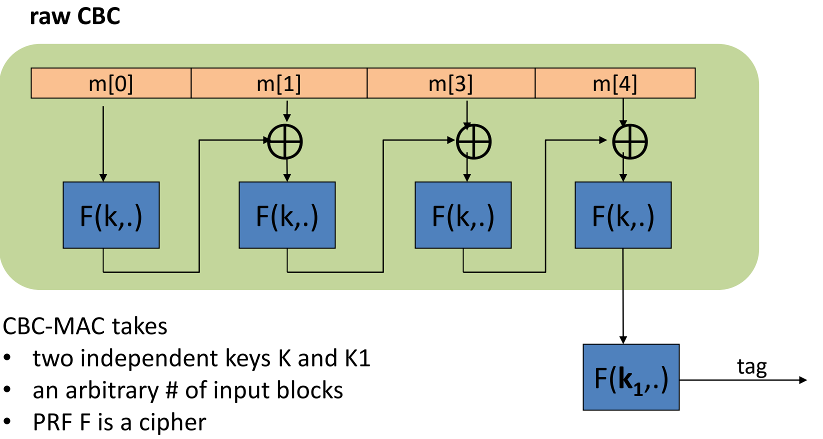
apr. '23 Hash functions 22

22

HMAC

- Computational efficiency
  - The message is hashed in the inner hash
  - The outer hash only hashes two blocks
- Security
  - There exists a proof of security in HMAC
  - THM - If an attacker can break HMAC then (s)he can break H

CBC-MAC: construction



CBC-MAC takes
- two independent keys K and K1
- an arbitrary # of input blocks
- PRF F is a cipher
- F = DES ➔ Data Auth. Data (DAA, FIPS PUB 113, ANSI X9.17)

Without the last encryption, rawCBC would be insecure

CBC-MAC: security

- Normally CBC-MAC does not use the last encryption ➔ rawCBC
- rawCBC is insecure
  - Proof (An existential forgery attack)
    1. The adversary chooses a one-block message x
    2. The adversary requests t = rawCBC(k, x) where t = E(k, x)
    3. The adversary outputs t' = t as MAC forgery of the two-block message x' = x, (t ⊕ x)

apr. '23 Hash functions 25

25

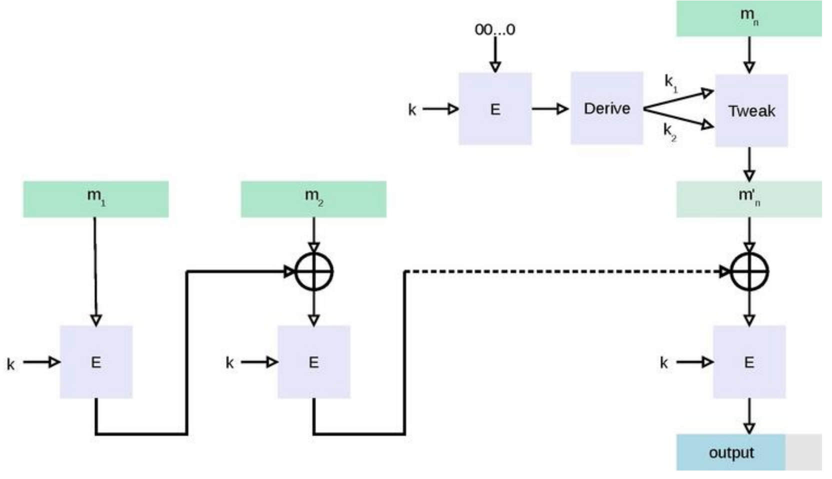
CBC-MAC: security

- Proof (for brevity rawCBC = H)
  - Let t' = H(k, x') = H(k, (x, (t ⊕ x)) =
    E(k, (E(k, x) ⊕ (t ⊕ x))) =
    E(k, t ⊕ (t ⊕ x)) =
    E(k, x) = t, where E is the cipher Q.E.D

apr. '23 Hash functions 26

26

Cipher-Based Message Authentication Code (CMAC)



apr. '23 Hash functions 27

27

CMAC

- CMAC overcomes the problems of CBC-MAC
- CMAC uses three keys K, K_1, K_2
  - K is k bit, K_1 and K_2 are n bit
  - K_1 and K_2 can be derived from K (NIST 800-38B)
    - L = $E_K(0^n)$
    - K1 = L ▪ x (if len(x) is an integer multiple of n)
    - K2 = L ▪ x^2 (if len(x) is not an integer multiple of n)
    - Polynomials $x, x^2 \in \mathrm{GF}(2^n)$, multiplication $\cdot$ in $\mathrm{GF}(2^n)$
- E = AES, 3DES

apr. '23 Hash functions 28

28

CBC-MAC & CMAC drawbacks

- CBC-MAC and CMAC are not suitable for high-speed implementations because they are neither pipelineable nor parallelizable

apr. '23 Hash functions 29

29

Message Authentication Code (MAC)

PADDING

30

MAC Padding

- Pad by zeroes ⇒ insecure
  - pad(m) and pad(m||0) have the same MAC
- Padding must be an invertible function
  - m0 ≠ m1 ⇒ pad(m0) ≠ pad(m1)
- Standard padding (ISO)
  - Append "100…00" as needed
    - Scan right to left
    - "1" determines the beginning of the pad
  - Add a dummy block if necessary
    - When the message is a multiple of the block
    - The dummy block is necessary or existential forgery arises

Padding by 0es is a bad idea

- Proof
  - Let x = x_1, x_2, x_3 where x_3 is shorter than a block
  - Let's pad x_3 as follows $x_3^* \leftarrow x_3||000$ (for example)
  - Let t be the tag outputted.
  - Consider know a message x' = x||0.
    - x' would be composed of three blocks $x'_1 = x_1$, $x'_2 = x_2$, and $x'_3 = x_3||0$.
    - x'_3 needs padding and becomes $x_3'^* = x_3||0||00 = x_3||000$.
    - So, x and x' after padding are equal and thus have the same tag.

QED

On dummy block

- Without dummy block, existential forgery arises
- Proof
  - Let x = x1, x2 which needs padding
  - Build x* = x1, x2||100, where x* is the padded message
  - Consider now x' = x1, x2||100
    - Since x' is a multiple of the block we don't pad it
  - It follows that x' = x* and thus x ad x' have the same tag

QED

apr. '23 Hash functions 33

33

TIMING ATTACK

apr. '23 Hash functions 34

34

Timing Attack

- Example: Keyczar crypto library (Python) [simplified]

```
def Verify(key, msg, tag):
        return HMAC(key, msg) == tag
```

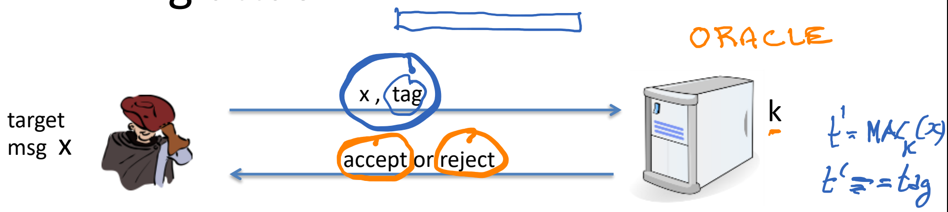
- The problem: operator '==' is implemented as a byte-by-byte comparison
  - It returns false when first inequality found

apr. '23 Hash functions 35

35

Timing attack

128



Timing attack: to compute tag for target message do:

Step 1: Query server with random tag

Step 2: Loop over all possible first bytes of tag and query server. Stop when verification takes a little longer than in step 1

Step 3: Repeat for all tag bytes until valid tag found

3	47	*	*	*	*

apr. '23 Hash functions 36

36

Defense – solution #1 🤔

- Make string comparator always take same time
- Solution 1:

```
return false if  tag  has wrong length
result = 0
for x, y in zip( HMAC(key,msg) , tag):
    result  |= ord(x) ^ ord(y)
return result == 0
```

- Can be difficult to ensure due to optimizing compiler

apr. '23 Hash functions 37

37

Defense – solution #2 😊

- Make string comparator always take same time
- Solution 2

```
def Verify(key, msg, tag):
    mac = HMAC(key, msg)
    return HMAC(key, mac) == HMAC(key, tag)
```

- Attacker doesn't know values being compared

apr. '23 Hash functions 38

38