Cryptography

3 – Authentication and hash functions

G. Chênevert

September 30, 2019

ISEN ALL IS DIGITAL! LILLE

yncréa

Today

User authentication

Hash function design

Message authentication

User authentication

Applications often need to ask users (or devices...) to identify themselves in order to know how to behave.

id : `Alice`

id : `Bob`

id : `Alice`

gnark gnark

Authentication factors

Obviously such an input needs to be **authenticated** (confirmed).

Authentication methods usually rely on **factors** such as:

- something the user *knows*,
- something the user *has*,
- something the user *is* (or a way he *behaves*).

Password authentication

Upon registration, every user provides (or is assigned) a password.

id : `Alice`

pw : `Ii(H48s`

id : `Bob`

pw : `secret`

. . .

Naive implementation

All valid pairs $(\mathrm{id}, \mathrm{pw})$ are stored by the service provider Sammy.

When a pair $(\mathrm{id}, \mathrm{pw}')$ is received, Sammy checks whether

$$\mathrm{pw}' = \mathrm{pw}.$$

Problem

An attacker with read access recovers all the passwords.

(Equivalently: need absolute trust in Sammy!)

Storing encrypted versions $E(k, \mathrm{pw})$ seems better…

…is it ? (hint: not really)

NB: *sending* encyrpted passwords on the communication channel is certainly a good idea, though

Solution

Use *one-way (lossy) encryption*

i.e. a **hash function**

$$H : \{0,1\}^* \longrightarrow \{0,1\}^n.$$

Examples:

MD5 (deprecated), SHA-1 (deprecated),

SHA-2, SHA-3, BLAKE2, Whirlpool, ...

Usage

A hash function turns everything into a fixed-length hex word.

```
from Crypto.Hash import MD5, SHA, SHA256

message = b"Hello"

print("init:", message)
print()
print("MD5 :", MD5.new(message).hexdigest())
print("SHA1:", SHA.new(message).hexdigest())
print("SHA2:", SHA256.new(message).hexdigest())
```

```
init: b'Hello'

MD5 : 8b1a9953c4611296a827abf8c47804d7
SHA1: f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
SHA2: 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```

```
from Crypto.Hash import MD5, SHA, SHA256

message = b"hello"

print("init:", message)
print()
print("MD5 :", MD5.new(message).hexdigest())
print("SHA1:", SHA.new(message).hexdigest())
print("SHA2:", SHA256.new(message).hexdigest())
```

```
init: b'hello'

MD5 : 5d41402abc4b2a76b9719d911017c592
SHA1: aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
SHA2: 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

Better password management

Sammy stores, for every valid user, a hash of their password:

$$(\mathrm{id}, h) \quad \text{with} \quad h = H(\mathrm{pw}).$$

Authentication:

Upon reception of $(\mathrm{id}, \mathrm{pw}')$, Sammy checks if

$$H(\mathrm{pw}') = h.$$

Requirement

The hash function should be **preimage resistant**:

given h, it must be computationally hard to find m such that

$$H(m) = h.$$

Attacks:

- brute force
- dictionary (precomputed)
- rainbow tables (space-time tradeoff)

Improvements

- **Salting**: store $(\text{id}, s, H(s \,\|\, \text{pw}))$ where s is random *salt*
- **Key stretching**: more generally, use a *key derivation function* to generate

$$k = K(s, \text{pw}) \quad \text{and store} \quad (\text{id}, s, k)$$

  where K is made *deliberately slow*

Examples: PBKDF2, Bcrypt, scrypt

$\Longrightarrow$ this is what should **always** be used in practice

Today

User authentication

Hash function design

Message authentication

Cryptographic hash functions

Hash functions are useful for many things:

- id generation
- hash tables
- pattern detection
- serialization
- ...

but certain specific properties are required for their use in cryptography.

Requirements

- **determinism**: $m = m' \implies H(m) = H(m')$
- **uniformity**: every hash occurs with probability $1/2^n$
- **avalanche**: $m \approx m', m \neq m' \implies H(m) \not\approx H(m')$

  (exactly the inverse of **continuity**)

Things that should be hard

- given h, find m such that $H(m) = h$

  (**preimage resistance**)

- given m, find $m' \neq m$ such that $H(m') = H(m)$

  (**second preimage resistance**)

- find $m \neq m'$ such that $H(m) = H(m')$

  (**collision resistance**)

A textbook case: the story of SHA-1

- 1995: **Secure Hash Algorithm 1** standardized by NIST
- 2005: first "theoretical" collision attacks published
- 2010: collision complexity brought down to roughly 2^{60}

  Estimated cost of attack: 3 M$
- 2015: "the SHAppening" first practical attack demonstration

  Estimated cost of attack: 100 k$
- 2017: "SHAttered" first public collision
- 2019: Improved chosen prefix attack

The birthday problem

- generating $N > 2^n$ hashes $\Longrightarrow$ *certain* collision
- if N values are generated uniformly at random, the *probability* of a collision is

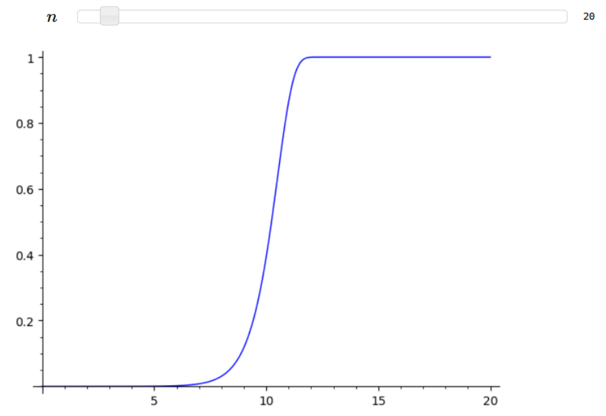
$$p = 1 - \prod_{k=0}^{N-1} \left(1 - \frac{k}{2^n}\right) \approx 1 - e^{-\frac{1}{2^n}\binom{N}{2}} \approx 1 - e^{-\frac{1}{2^{n+1}}N^2}$$

Example: The probability that 40 randomly chosen persons share a birthday is

$$\approx 1 - e^{-\frac{1}{365}\binom{40}{2}} \approx 88.2\%$$

NB: non-uniformity in the distibution of values only make collisions *more* probable

Collision probability as function of hash length



Birthday attack

One can show that the *average* number of values to be generated before a collision is found is approximately

$$\sqrt{\pi 2^{n-1}} \approx 1.25 \times 2^{\frac{n}{2}}.$$

Hence: a n-bit hash function provides $\leq \frac{n}{2}$ bits of security.

$\implies$ hashes need to be at least 256 bits long to provide 128 bits of security.

Pearson hash

An *insecure* construction

Divide the message m into k-bit blocks $(m_1, m_2, \dots)$

and choose a permutation σ of $\{0,1\}^k = [\![0, 2^k[\![$.

h = 0

for m_i in m:

$$h = \sigma(h \oplus m_i)$$

Nice, but specifying σ takes $k \cdot 2^k$ memory ...

Merkle-Damgård construction

Reuses the idea of Pearson hashing.

Pseudocode

$h = h_0$

for m_i in m:

$\quad h = F(h, m_i)$

where the *compression function* F is typically a simple operation iterated r times on the internal state (size s, divided into w-bit words)

Famous cryptographic hash functions

name	published	deprecated	n	k	s	w	r
MD5	1991	2000	128	512	512	32	64
SHA-1	1995	2005	160	512	160	32	64
SHA-2	2001	–	256 (224)	512	256	32	64
			512 (448)	1024	512	64	80
SHA-3	2012	–		...			

SHA-3 (Keccak)

Sponge construction

$(R, C) = (R_0, C_0)$

// absorption

for m_i in m:

$\quad (R, C) = F(R \oplus m_i, C)$

// then some more drying

eventually output R

Allows for certain freedom in choice of parameters

e.g. SHA3-224, SHA3-256, SHA3-384, SHA3-512, . . .

Today

User authentication

Hash function design

Message authentication

Hash as checksum

Hash functions can be used to verity message integrity.

Alice: appends to a message m its hash $h = H(m)$.

Bob: verifies upon reception of (m, h) that $h = H(m)$.

(If not: transmission problem detected)

Example

$m = \text{You owe me 10 \$}$

$h = \texttt{c7b12b33fdd17399}$

$m_{\text{received}} = \text{You owe me 10 \$}$

$h_{\text{received}} = \texttt{c7b12b33fdd17399}$

$h_{\text{computed}} = \texttt{c7b12b33fdd17399}$

Ok !

Example (cont'd)

$m =$ You owe me 10 \$

$h =$ `c7b12b33fdd17399`

$m_{\text{received}} =$ You owe me 100 \$

$h_{\text{received}} =$ `c7b12b33fdd17399`

$h_{\text{computed}} =$ `08821af9be531f29`

Error !

But also...

$m =$ You owe me 100 \$

$h =$ `08821af9be531f29`

$m_{\text{received}} =$ You owe me 100 \$

$h_{\text{received}} =$ `08821af9be531f29`

$h_{\text{computed}} =$ `08821af9be531f29`

Ok ! ...

Problem

Even if H cannot be manipulated ...

anybody can compute a valid hash!

Double-edged sword:

- falsification
- repudiation

$\implies$ no authentication at all

Idea: encrypt the hash

Alice: appends to m its encrypted hash $h = E(k, H(m))$

Bob: upon reception of (m, h), checks whether $H(m) = D(k, h)$

Problem: since $H(m)$ and h are public, the secret key k is exposed...

Message authentication codes

Definition

A **MAC** consists of a *tag* function $\mathcal{K} \times \mathcal{M} \to \mathcal{T}$ as well as a *verification algorithm* that decides whether a particular MAC is valid for a given message.

- **Correctness**: every generated MAC should be valid
- **Forgery resistance** no one should be able to create a valid pair (m, t) without knowing the key.

From a hash function

Standard construction:

$$\mathrm{HMAC}(k, m) := H(\,(k \oplus \mathrm{opad}) \,\|\, H(\,(k \oplus \mathrm{ipad}) \,\|\, m)\,)$$

Alice: appends to m its tag $t = \mathrm{HMAC}(k, m)$

Bob: verifies upon reception of (m, t) whether $t = \mathrm{HMAC}(k, m)$

From a block cipher

Idea: Encrypt $m = m_1 \| \cdots \| m_\ell$ in CBC-mode with $\mathrm{IV} = 0$.

$$\text{CBC-MAC}(k, m) := c_\ell$$

$+$ additional precautions to prevent *extension attack*

Never reuse the same key for different purposes!

Authenticated encryption

Given a secure cipher + a secure MAC:

- **encrypt then MAC**: always ok
- encrypt and MAC: weakens encryption
- MAC then encrypt: ok in some cases

End remarks

- AE provides confidentiality, authentication, integrity, non-repudiation
- modern approach is to provide AE as a single primitive
- examples: OCB, EAX, EtM, GCM, CCM modes
- AE does not prevent *replay attacks* by itself

$\Longrightarrow$ Authenticated Encryption with Associated Data (AEAD) as IV should be used.