Funciones de Hash

La librería PyCryptoDome tiene funciones de hash para varios algoritmos. Vamos a cargar algunas de ellas. La lista completa está en: https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html (https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html)

(Recuerda: MD5 está obsoleto y roto, no se tiene que utilizar en aplicaciones reales)

In [1]:

```
from Crypto.Hash import MD5, SHA256, SHA512, SHA3_256
```

Tradicionalmente, una función de hash se va alimentanto de datos (`update()`) hasta que necesitas el hash (también llamado "digest"). Este procedimiento de `update()` funciona, por ejemplo, si tienes que calcular el hash de un conjunto de mensajes o lees los bytes bloque a bloque de un archivo grande.

In [2]:

```
hasher = SHA256.new()
hasher.update(b'hola')
hasher.update(b'mundo')
hash1 = hasher.hexdigest()
print(hash1)
```

```
93fa3e4624676f2e9aa143911118b4547087e9b6e0b6076f2e1027d7a2da2b0a
```

Si ya conoces el mensaje puedes hacerlo todo en una sola línea como en el ejemplo siguiente. Observa que el hash coincide con el calculado antes.

In [3]:

```
hash2 = SHA256.new(data=b'holamundo').hexdigest()
print(hash2)
print(hash1 == hash2)
```

```
93fa3e4624676f2e9aa143911118b4547087e9b6e0b6076f2e1027d7a2da2b0a
True
```

¿Qué pasa si cambiamos ligeramente el mensaje? Por ejemplo, añadimos un espacio, o ponemos letras en mayúsculas, o signos de admiración...

In [4]:

```
hash2 = SHA256.new(data=b'hola mundo').hexdigest()
print(hash2)
print(hash1 == hash2)
```

```
0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f
False
```

Preguntas:

- Prueba varios cambios en el mensaje "hola mundo" y apunta los hashes, verás que cambian totalmente por muy pequeños que sean los cambios: algunas mayúsculas, números, signos de puntuación...
- ¿Cuántos mensajes existen que tengan el mismo hash que "hola mundo"?
- ¿Podrías encontrar alguno de estos mensajes que tengan el mismo hash que "hola mundo"?
- Calcula el valor de hash de un archivo de texto con el texto "hola mundo" en tu ordenador desde línea de comandos. ¿Coincide con el hash anterior?
  - Powershell en Windows: `Get-FileHash NOMBREDEARCHIVO`
  - Linux/OSX: `sha256sum NOMBREDEARCHIVO`
- Cambia el nombre del archivo y calcula su hash. ¿Ha cambiado el hash al cambiar el nombre del archivo?

Tamaño de un resumen hash

Fíjate: el hash SHA256 siempre tiene la misma longitud, sea como sea de largo el texto de la entrada.

- SHA256: longitud 256 bits
- SHA512: longitud 512 bits

In [5]:

```
print(SHA256.new(data=b'hola').hexdigest())
print(SHA256.new(data=b'hola mundo').hexdigest())
print(SHA256.new(data=b"""Cryptographic hash functions take arbitrary binary strings
and produce a random-like fixed-length output (called digest or hash value).

It is practically infeasible to derive the original input data from the digest. In c
words, the cryptographic hash function is one-way (pre-image resistance).

Given the digest of one message, it is also practically infeasible to find another
message (second pre-image) with the same digest (weak collision resistance).

Finally, it is infeasible to find two arbitrary messages with the same digest
(strong collision resistance).

Regardless of the hash algorithm, an n bits long digest is at most as secure as a
symmetric encryption algorithm keyed with n/2 bits (birthday attack).

Hash functions can be simply used as integrity checks. In combination with a
public-key algorithm, you can implement a digital signature.""").hexdigest())
```

```
b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f
000bdd11b8fe147e274127d1d18edcb9d4acff62c96f7e8543de3b7d90a185c5
```

En los ejemplos anteriores hemos utilizado la función `hexdigest`, que es como tradicionalmente se presentan los hashes para poder imprimirlos. Esa es la representación hexadecimal de un número. Por ejemplo, el número 14 (decimal) se representa como '0e' (hexadecimal) y el número 254 (decimal) como 'fe' (hexadecimal). Fíjate: 8 bits son un byte, es decir, un número entre 0 y 255 (en decimal), es decir, un número entre 00 y ff (en hexadecimal). **Un byte son dos caracteres hexadecimales**.

Podemos accede a la cadena binariade bytes, sin pasarla a hexadecimal, utilizando la función `digest()` en vez de `hexdigest()`. Pero no podríamos imprimirla.

Así que:

- El resumen SHA256 es de 256 bits, sea como sea el tamaño de la entrada
- 256 bits son **32 bytes**
- Que se representan como **64 caracteres hexadecimales**
- Pero ambas representaciones son equivalentes. Simplemente, una podemos imprimirla y la otra no. A veces querremos imprimir hashes y por eso es común que los veamos en hexadecimal

In [6]:

```
hasher = SHA256.new(data=b'hola')
hash_bytes = hasher.digest()
hash_hexa = hasher.hexdigest()

print(f'Valor de hash SHA256 en binario. Longitud={len(hash_bytes)} bytes Valor={has
print(f'Valor de hash SHA256 en hexadecimal. Longitud={len(hash_hexa)} caracteres Va
```

```
Valor de hash SHA256 en binario. Longitud=32 bytes Valor=b'\xb2!\xd9\x
db\xb0\x83\xa7\xf34(\xd7\xc2\xa3\xc3\x19\x8a\xe9%aMp!\x0e(ql\xca\xa7\x
cdM\xdby'
Valor de hash SHA256 en hexadecimal. Longitud=64 caracteres Valor=b221
d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
```

Por tradición, se ha preferido codificar los hashes en hexadecimal y no en base64, que hubiese sido otra opción perfectamente válida.

Velocidades de cálculo de hash

Vamos a calcular cuántos hashes podemos calcular por segundo.

OJO: este cálculo simplemente nos vale para comparar algoritmos entre sí. Jupyter no tiene acceso a la GPU de tu ordenador, ni Pycryptodome está pensada para gran velocidad. Si exprimes tu ordenador con programas externos seguro que obtendrás números mucho mayores.

La ejecución de estas líneas puede llevar desde varios segundos a un minuto. Fíjate en que el bloque empieza con `In[*]` mientras está haciendo cálculos.

In [7]:

```
import timeit

NUM=500000
print(f'Calculando {NUM} hashes en MD5, SHA256, SHA512, SHA3_256...')

time_md5 = timeit.timeit(lambda: MD5.new(data=b'hola').hexdigest(), number=NUM)
time_sha256 = timeit.timeit(lambda: SHA256.new(data=b'hola').hexdigest(), number=NUM
time_sha512 = timeit.timeit(lambda: SHA512.new(data=b'hola').hexdigest(), number=NUM
time_sha3 = timeit.timeit(lambda: SHA3_256.new(data=b'hola').hexdigest(), number=NUM

print(f'MD5: spent={time_md5} s speed={NUM / time_md5} H/s')
print(f'SHA256: spent={time_sha256} s speed={NUM / time_sha256} H/s')
print(f'SHA512: spent={time_sha512} s speed={NUM / time_sha512} H/s')
print(f'SHA3_256: spent={time_sha3} s speed={NUM / time_sha3} H/s')
```

```
Calculando 500000 hashes en MD5, SHA256, SHA512, SHA3_256...
MD5: spent=2.938345291999994 s speed=170163.7997962021 H/s
SHA256: spent=3.8322777079999923 s speed=130470.7117013559 H/s
SHA512: spent=5.016798667000003 s speed=99665.15166114792 H/s
SHA3_256: spent=3.3511497089999978 s speed=149202.52552644175 H/s
```

Preguntas:

- ¿Cuál de los algoritmos es más rápido? ¿Cómo afecta doblar el número de bits (es decir, pasar de 256 a 512 bits)?
- Calcula el hash SHA-256 y SHA-512 de un archivo de unos 500MB en tu ordenador (por ejemplo, una película) ¿Cuánto tiempo le lleva?

Firma Digital

Vamos a aprovechar lo que ya sabemos de cifrado asimétrico y hashes para ver cómo funciona una firma digital.

Alice firma un documento

In [8]:

```
document = b"""Cryptographic hash functions take arbitrary binary strings as input,
and produce a random-like fixed-length output (called digest or hash value).

It is practically infeasible to derive the original input data from the digest. In c
words, the cryptographic hash function is one-way (pre-image resistance).

Given the digest of one message, it is also practically infeasible to find another
message (second pre-image) with the same digest (weak collision resistance).

Finally, it is infeasible to find two arbitrary messages with the same digest
(strong collision resistance).

Regardless of the hash algorithm, an n bits long digest is at most as secure as a
symmetric encryption algorithm keyed with n/2 bits (birthday attack).

Hash functions can be simply used as integrity checks. In combination with a
public-key algorithm, you can implement a digital signature."""
```

Vamos a generar un par de claves RSA para Alice: una pública `alice_pk` y otra privada `alice_sk`. Recuerda: la clave pública la conoce todo el mundo, la clave privada solo la conoce Alice. Ya veremos cómo se distribuye esa clave pública.

NOTA: En un entorno real esto se hace mucho antes de firmar: ¡el par de claves debería estar preparado y la clave pública distribuida desde meses antes de la firma! Veremos esto en el tema de PKI

In [9]:

```
# Clave de 2048 bits de Alice, pública y secreta
from Crypto.PublicKey import RSA
alice_sk = RSA.generate(2048) # Clave secreta de Alice
alice_pk = alice_sk.publickey() # Clave pública de Alice
```

PyCryptodome ya incluye un módulo para firmar usando las recomendaciones PKCS1. Vamos a aprovechar el módulo, para aprender buenas costumbres y porque PyCryptodome no nos deja utilizar RSA de forma insegura.

Ese módulo de firma:

- Calcula el hash del documento utilizando el hasher que le pasemos (que será SHA256)
- Cifra el hash del documento utilizando la clave privada de Alice
- Todo lo hace siguiendo las recomendaciones PKCS1
- La firma la codifica en base64, para que podamos verla por pantalla (esto no es necesario en realidad)

Finalmente, Alice enviaría en un mensaje el documento y su firma. En realidad, lo más probable es que Alice además cifre el documento utilizando algún tipo de cifrado simétrico como AES para proteger su confidencialidad, pero vamos a obviar esa parte en este ejercicio.

NOTA: dado que se necesita la clave privada de Alice para firmar, **solo Alice puede generar esta firma de este documento ya que solo ella conoce su clave privada**.

In [10]:

```
from Crypto.Signature import pkcs1_15
from base64 import b64encode, b64decode

hasher = SHA256.new(data=document)
signature = pkcs1_15.new(alice_sk).sign(hasher)

msg = dict(document=document, signature=b64encode(signature))
print(msg)
```

```
{'document': b'Cryptographic hash functions take arbitrary binary stri
ngs as input,\nand produce a random-like fixed-length output (called d
igest or hash value).\n\nIt is practically infeasible to derive the or
iginal input data from the digest. In other\nwords, the cryptographic
hash function is one-way (pre-image resistance).\n\nGiven the digest o
f one message, it is also practically infeasible to find another\nmess
age (second pre-image) with the same digest (weak collision resistanc
e).\n\nFinally, it is infeasible to find two arbitrary messages with t
he same digest\n(strong collision resistance).\n\nRegardless of the ha
sh algorithm, an n bits long digest is at most as secure as a\nsymmetr
ic encryption algorithm keyed with n/2 bits (birthday attack).\n\nHash
functions can be simply used as integrity checks. In combination with
a\npublic-key algorithm, you can implement a digital signature.', 'sig
nature': b'jAdRhP3sMrBTuDDGrOcDbNzm/ytZb5Qlzsp5PoxjGYzUfuZDNLAXVZAkkZJ
hQig4T8sSb1GIL33OQm+dLsFjiNGbjI8bROTUPLlBwbtI44VfTXGQ+KEnr5JSUgQTDfPfM
+EkQOqEdjqSN1aykpNYu9fggkFXX81X1mQ04QoPpesd8Sn3tchsCFLt3dUvQaSlt91c4nR
Ti5jDLCwxw4RwRX7i0J7hfxxr4X7Zlg/XEhRbLnw5ecNHCKYDy3Luf/Yhoi6JiE0VWw8o
z/KxPiTyTKpSr5OgUsjG21eFIMnAsd1xHPpHogRhTG++Qgr8kXIm0ASt0pu3iV3QyDVMLw
A09A==''}
```

Pregunta:

- ¿Por qué crees que Alice cifra **solo** el hash del mensaje con RSA, en vez de cifrar directamente **todo** el mensaje con RSA?

Bob verifica la firma de Alice

Bob recibe el mensaje `msg`, que incluye el documento y la firma de Alice, y ya conoce la clave pública de alice `alice_pk` de alguna manera (ver tema PKI)

Así que Bob hace el proceso inverso:

- Calcula el hash SHA256 del documento recibido
- Decodifica el base64 y descifra la firma recibida utilizando la clave pública de Alice
- Todo lo hace siguiendo las recomendaciones PKCS1

Como curiosidad, la librería PyCryptodome lanza un error cuando la firma no es válida, y no hace nada si es correcta.

Si la verificación de la firma con la clave pública de Alice es correcta, entonces **Bob sabe que el documento lo ha enviado Alice, y no puede haberlo enviado nadie más**.

In [11]:

```
rcv_document = msg['document']
rcv_signature = b64decode(msg['signature'])

pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature)
print("La firma es válida")
```

```
La firma es válida
```

¿Qué pasa si un atacante intercepta el mensaje y cambia el documento? Aquí vemos un ejemplo: el atacante ha interceptado el documento y ha puesto información falsa. ¿Puedes identificar qué parte del documento ha cambiado el atacante?

Observa que ahora la firma de Alice no verifica y la función lanza un error.

In [12]:

```
rcv_document = b"""Cryptographic hash functions take arbitrary binary strings as inp
and produce a random-like fixed-length output (called digest or hash value).

It is practically infeasible to derive the original input data from the digest. In c
words, the cryptographic hash function is one-way (pre-image resistance).

Given the digest of one message, it is also practically infeasible to find another
message (second pre-image) with the same digest (weak collision resistance).

Finally, it is infeasible to find two arbitrary messages with the same digest
(strong collision resistance).

Regardless of the hash algorithm, an n bits long digest is at most as secure as a
symmetric encryption algorithm keyed with n/3 bits (birthday attack).

Hash functions can be simply used as integrity checks. In combination with a
public-key algorithm, you can implement a digital signature."""

pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature)
print("La firma es válida")
```

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
 last)
Input In [12], in <module>
      1 rcv_document = b"""Cryptographic hash functions take arbitrary
binary strings as input,
      2 and produce a random-like fixed-length output (called digest o
r hash value).
      3
   (...)
     16 Hash functions can be simply used as integrity checks. In comb
ination with a
     17 public-key algorithm, you can implement a digital signature.
"""
---> 19 pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), r
cv_signature)
     20 print("La firma es válida")

File /opt/homebrew/lib/python3.9/site-packages/Crypto/Signature/pkcs1_
15.py:137, in PKCS115_SigScheme.verify(self, msg_hash, signature)
    131 # Step 4
    132 # By comparing the full encodings (as opposed to checking each
    133 # of its components one at a time) we avoid attacks to the pad
ding
    134 # scheme like Bleichenbacher's (see http://www.mail-archive.co
m/cryptography@metzdowd.com/msg06537). (http://www.mail-archive.com/cr
yptography@metzdowd.com/msg06537).)
    135 #
    136 if em1 not in possible_em1:
--> 137     raise ValueError("Invalid signature")
    138 pass

ValueError: Invalid signature
```

Podemos gestionar los errores con un `try/except` e informar al usuario.

In [13]:

```
try:
    pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature)
    print("La firma es válida")
except ValueError:
    print("La firma NO es válida")
```

```
La firma NO es válida
```