

Morinsky Cryptographic Tools

Rafael González

Empezaré a morir el día que ya no quiera aprender nada nuevo

rgonzalezmoro@gmail.com

Tabla de contenido

1. Introducción	3
2. Librerías Python utilizadas	3
3. Algunas consideraciones.....	3
4. Módulos de MCT.....	4
5. Modulo funcRSA.py – Operaciones con claves criptográficas	4
6. Serializar objetos.....	9
6.1. <i>Cómo guardar una clave RSA en una base de datos</i>	<i>9</i>
6.1.1. PRIMER INTENTO: Guardar una clave RSA directamente	10
6.1.2. SEGUNDO INTENTO Serializar la clave RSA.....	11
6.2. <i>Guardar un HASH en una base de datos.....</i>	<i>12</i>
6.2.1. PRIMER INTENTO Guardar un hash directamente	12
6.2.2. SEGUNDO INTENTO Serializar un json.dumps.....	13
6.2.3. TERCER INTENTO Hay que bajar de nivel.....	13
7. Firmar un objeto y verificar firma.....	16
7.1. <i>Firma de un hash.....</i>	<i>17</i>
7.2. <i>Verificación de firma.....</i>	<i>18</i>
8. Nuestro main() al completo	19
9. ANEXO A - Variables de sistema entorno en Jupyter	22

1. Introducción

Las Morinsky Crypto Tools (MCT) nacen como consecuencia de agilizar y clarificar el uso de funciones criptográficas en un proyecto de blockchain

Puedes usarlas libremente. No me gustaría que cobraras por ello y lo más que pido es que cites al autor de las mismas, o sea yo

Si las mejoras te rogaría me lo dijeras para poder enriquecerme yo con tu conocimiento

2. Librerías Python utilizadas

Este desarrollo se ha hecho utilizando *PyCryptodome*. Puedes encontrar toda la documentación sobre esta librería en <https://pycryptodome.readthedocs.io/en/latest/index.html>

El motivo es que en esta librería se encuentran soluciones para cifrado, criptografía de clave pública, hash etc

3. Algunas consideraciones

- MCT son clases y funciones para uso interno de tus propios desarrollos. Hay quien tiene la costumbre de partir de la base que los parámetros que van a recibir las funciones son correctos. Yo no tengo esa costumbre. Cada función tiene su correspondiente validación, y en caso de no superarla, hace un `sys.exit()`. Prefiero que un error de esa naturaleza no pase desapercibido en fase de pruebas
- Como ya he dicho, yo utilizo *PyCryptodome*. Por lo tanto si quieres usar *haslib*, *PyCrypto* o *cryptographic* deberás adaptar las llamadas a las funciones propias de cada módulo
- Las claves para cifrados y otras variables críticas deben ir definidas como variables del sistema de tal forma que, en producción, puedan ser accedidas sin que puedan ser comprometidas. Para evitar tener que andar modificando el `.bash_profile` (o el Shell que utilices) las he definido en un fichero [environ.py](#) que deberás adaptar a tu entorno
- Las MCT son, como ya se ha dicho, tres ficheros extensión `.py` pero las pruebas para documentar este manual están hechas en Jupyter Lab. Al final de este documento hay un ANEXO – A Variables entorno en Jupyter donde se explica como se pueden manejar variables de este tipo dentro de Anaconda Jupyter Lab

En Jupyter no aplica el mapeo de variables que se utiliza en Python dado que Kupyter no es un entorno interactivo (y esto es así aunque sea confuso)

4. Módulos de MCT

Las MCT están compuestas de tres módulos '.py'

- [funcRSA.py](#): es un módulo con funciones propias de la generación y manejo de claves criptográficas (objetos RSA)
- [funcHash.py](#): es un módulo que contiene funciones para operaciones con hash
- [classHass.py](#): es una clase cuyos atributos son los propios de las operaciones con un hash

Mas adelante veremos por qué hay un módulo con funciones de *hashing* y además hay una clase que hace lo mismo

Comencemos con las funciones relacionadas con la gestión de claves criptográficas

5. Modulo funcRSA.py – Operaciones con claves criptográficas

[setRSAkey\(*params\)](#)

Devuelve una clave criptográfica (privada) y opcionalmente permite guardarla en un fichero cifrada o sin cifrar

Recibe	<p>Uno o tres parámetros (no vale 2 ó 4)</p> <ul style="list-style-type: none">• Longitud <i>params[0]</i>: Tipo int. Longitud de la clave RSA• Nombre <i>params[1]</i>: Tipo str. En caso de querer guardar la clave en un fichero (extensión de la variable de entorno DEFAULT_EXTENSION)• Cifrado <i>params[2]</i>: Boolean. Si True cifra la clave con un AES256
Valida	<ul style="list-style-type: none">• Que recibe uno o tres parámetros• Que los tipos de los parámetros recibidos son los adecuados
Acciones y valor devuelto	<ul style="list-style-type: none">• Si solo recibe un parámetro devuelve una objeto clave RSA• Si recibe 3 parámetros:<ul style="list-style-type: none">◦ Además de devolver la clave RSA crea un fichero con el nombre indicado◦ Si el tercer parámetro es True cifra la clave y la guarda en el fichero (formato del fichero PEM pero el contenido es un JSON). LA clave de cifrado sale de la variable de entorno CIPHER_KEY◦ Si tercer parámetro es False guarda en formato PEM la clave serializada (SIN CIFRAR)• Valida el tipo de cada parámetro recibido, es decir, 'int', 'str', 'boolean'
Llama a	<code>cipherCTR(key)</code> para cifrar la clave RSA y guardarla en un fichero PEM, formato JSON
Posibles errores	<ul style="list-style-type: none">• Por validación errónea del tipo de argumento recibido• Por error en operación al escribir en fichero

getRSAkey(*params)

Lee una clave RSA de un fichero

Recibe	Un nombre de fichero sin extensión. La extensión sale de la variable del entorno DEFAULT_EXTENSION
Valida	<ul style="list-style-type: none">• Que el recibe un string• Que existe la ruta – fichero indicado. La ruta sale de la variable de entorno FILES_PATH
Acciones y valor devuelto	<p>Devuelve un objeto RSA o provoca un sys.exit() si se produce algún error insalvable</p> <p>Si el fichero recibido existe se lee su contenido que siempre será un string. La función detecta si el fichero indicado tiene la clave cifrada o la tiene serializada pero sin cifrar (solo para pruebas)</p> <p>Ese string tiene un diccionario dentro siempre que en su momento se guardara la clave cifrada (tercer parámetro de setRSAkey a True)</p> <p>Si la clave se guardó sin cifrar (serializada) simplemente se obtiene una cadena de caracteres que es la clave con el formato BEGIN END tradicional de una clave serializada</p>
Llama a	decipherCTR()
Posibles errores	<ul style="list-style-type: none">• No se recibe una cadena de caracteres• No existe la ruta-fichero indicado• Error al abrir el fichero

cipherCTR(inRSA)

Devuelve un JSON con un AES256 CTR de la clave recibida

Recibe	Una clave RSA
Valida	<ul style="list-style-type: none">• Que el recibe una clave RSA
Acciones y valor devuelto	Si el objeto recibido es una clave RSA devuelve un JSON que contiene un diccionario. La clave RSA recibida es cifrada con un AES256 CTR en el campo 'cipherprivate'
Llama a	isRSA() para comprobar si se ha recibido un objeto RSA

Posibles errores	<ul style="list-style-type: none"> • No se recibe un objeto RSA • Error interno en el cifrado AES (error PyCryptodome no controlable)
------------------	---

decipherCTR(_json_read)

Devuelve descifrada una clave RSA que estaba cifrada en el JSON recibido

Recibe	Una fichero JSON que contiene un diccionario
Valida	No hace validaciones. Vienen hechas de getRSAkey()
Acciones y valor devuelto	Descifra la clave y la devuelve serializada
Llama a	No llama a ninguna función interna
Posibles errores	<ul style="list-style-type: none"> • Error interno al descifrar AES (error PyCryptodome no controlable)

isRSA(_object)

Comprueba si el objeto recibido es de la clase RSA

Recibe	Un objeto
Valida	No hace validaciones
Acciones y valor devuelto	<ul style="list-style-type: none"> • True si el objeto recibido pertenece a la clase RSA.RsaKey • False si no pertenece a esa clase
Llama a	No llama a ninguna función interna
Posibles errores	No hay control de errores

isPrivateKey(_key)

Comprueba si el objeto recibido es de la clave privada RSA

Recibe	Una clave RSA
Valida	<ul style="list-style-type: none"> • Que es una clave RSA • Que tiene el atributo has_private(), es decir, que es una clave privada

Acciones y valor devuelto	<ul style="list-style-type: none"> • True si el objeto es una clave privada RSA • False si no es una clave privada
Llama a	isRSA()
Posibles errores	Que el objeto recibido es una clave RSA

isPublicKey(_key)

Comprueba si el objeto recibido es de la clave publica RSA. (Reamente el False de isPrivateKey() es el True de esta función)

Recibe	Una clave RSA
Valida	<ul style="list-style-type: none"> • Que es una clave RSA • Que no tiene el atributo has_private(), es decir, que se trata de una clave pública
Acciones y valor devuelto	<ul style="list-style-type: none"> • True si el objeto es una clave pública RSA • False si no es una clave pública
Llama a	isRSA()
Posibles errores	Que el objeto recibido es una clave RSA

serializeKey(_key)

Convierte a binario (bytes) una clave RSA

Recibe	Una clave RSA
Valida	<ul style="list-style-type: none"> • Que es una clave RSA
Acciones y valor devuelto	Si se trata de una clave RSA devuelve el exportKey() de la misma, ya sea pública o privada
Llama a	isRSA() en un if de una línea
Posibles errores	Que el objeto recibido es una clave RSA

deSerializeKey(_key)

Convierte a objeto RSA una clave serializada (binario / bytes)

Recibe	Un bytes string
Valida	Que se ha recibido una cadena de bytes
Acciones y valor devuelto	Devuelve el importKey() del objeto recibido
Llama a	No llama a ninguna función interna
Posibles errores	Que el objeto recibido no sea una cadena de bytes

6. Serializar objetos

Serializar un objeto es convertirlo a binario (o bytes como también se dice en Python). Esto que parece trivial en determinado tipo de datos puede ser más complejo en objetos de clases 'especiales'

El siguiente código muestra como se serializa una cadena de caracteres. Todo aquello que se puede convertir a cadena de caracteres será fácilmente serializable

```
cadena = 'Esto es una cadena de caracteres'
cadena_serial = cadena.encode('utf-8')
print(type(cadena_serial))
print(cadena_serial.__class__)
print(cadena_serial)

<class 'bytes'>
<class 'bytes'>
b'Esto es una cadena de caracteres'
```

Hay otros objetos que tienen sus propios métodos de serialización como por ejemplo las claves RSA

```
key = setRSAkey(2048)
key.exportKey()

b'-----BEGIN RSA PRIVATE KEY-----
QndK\nbxbrn+v/wVdW5Q7dzvZrl0Soczv
paUyGHcNP6YbzDC27nmRGgLwZ07ZxjhdA
p3ugAoqDIue1\nG/+CJnnvurRbSA/Vji5
SUzGITa9IqEFjWZlrjMVA+QIDAQABaoIB
T+vx/0MNS5Ft4PmR/Y/s\nnt2mr170ftk
```

En el código anterior se utiliza la función `exportKey()` del objeto RSA para convertir al bytes la clave RSA generada. Sabemos que está en bytes porque el objeto devuelto empieza por `b'`

6.1. Cómo guardar una clave RSA en una base de datos

Partimos de:

- A. Una base de datos SQLITE3 llamada `bdusers` que tendrá una tabla que llamaremos `tableusers` para poder hacer 'juegos' guardando claves. De momento vamos a poner dos campos
 - **nombre:** que será un tipo TEXT
 - **clave_rsa:** que va a contener una clave RSA que ya sabemos que no es ni de tipo INT, REAL, NUMERIC etc ni de tipo TEXT ni BOOLEAN. Será por tanto BINARIO (BLOB)

- **timestamp**: que será la fecha hora en la que el usuario fue añadido a la base de datos. Será tipo texto

Nuestra tabla quedaría

▼ tableusers	
nombre	TEXT
clave_rsa	BLOB
timestamp	TEXT

- B. Un modulo Jupyter [main.ipynb](#) para lanzar las pruebas
- C. Un módulo en Jupyter denominado [bdoper.ipynb](#) para hacer un INSERT, SELECT y DELETE con lo cual nuestras primeras líneas de [main.ipynb](#) serán

```
# Import librerías python
import sys
import os

# Import de modulos ipynb
import import_ipynb
import environ
import bdoper

sys.path.append(os.getenv('MCT_PATH'))

from funcRSA import setRSAkey
```

(Recuerda que en el ANEXO del final se explica como manejar variables de entorno en Jupyter)

6.1.1. PRIMER INTENTO: Guardar una clave RSA directamente

Generamos nuestra clave RSA y comprobamos que su tipo

```
key = setRSAkey(2048)
type(key)

Crypto.PublicKey.RSA.RsaKey
```

Llamamos a la función que inserta un registro en la tabla de usuarios y observa el error que aparece

```

nombre = "Alberto Perez"
bdoper.insert_user(nombre, key)

```

ERROR:root:Internal Python error in the inspect module.
Below is the traceback from this internal error.

Traceback (most recent call last):
File "<string>", line 26, in insert_user
sqlite3.InterfaceError: Error binding parameter 1 - **probably unsupported type.**

Esto ocurre porque nuestro campo clave_rsa es BLOB y estamos intentando insertar algo que no es binario, es un objeto RSA

6.1.2. SEGUNDO INTENTO Serializar la clave RSA

Vamos a serializar esa clave RSA y a intentar guardarla a ver que pasa

```

nombre = "Alberto Perez"
bdoper.insert_user(nombre, key.exportKey())

```

Usuario añadido a la tabla

Vemos que la cosa ha cambiado y si consultamos la base de datos veremos que la tabla contiene

	nombre	clave_rsa	timestamp
	Filtro	Filtro	Filtro
1	Alberto Perez	——BEGIN RSA PRIVATE KEY——MIIEogIBA...	23/05/2020, 15:43:33

ATENCIÓN: Que ahí hemos guardado en base de datos una CLAVE PRIVADA. SOLO ES UN EJEMPLO, esto no debe hacerse. En el ejemplo final guardaremos la clave pública para comprobar que un hash ha sido firmado con su correspondiente clave privada

Te recuerdo que las MCT tienes una función que es serializaKey() que te permite hacer un código más limpio. (No olvides importar serializeKey desde funcRSA)

```

nombre = "Alberto Perez"
bdoper.insert_user(nombre, serializeKey(key))

```

Usuario añadido a la tabla

Conclusión:

Solo podemos guardar claves RSA serializadas en un campo binario de una base de datos

6.2. Guardar un HASH en una base de datos

Antes de nada te recomiendo leas estos dos artículos que hablan de cuando un objeto es *hasehable* o no

- <https://www.edureka.co/blog/hash-in-python/>
- <https://medium.com/better-programming/3-essential-questions-about-hashable-in-python-33e981042bcb>

Vamos a añadir a nuestra tabla un campo hash de tipo BLOB (*realmente no hay posibilidad a pensar que pueda ser de otro tipo*). No olvides modificar el INSERT de [bdoper.ipynb](#)

 tableusers	
 nombre	TEXT
 clave_rsa	BLOB
 hash	BLOB
 timestamp	TEXT

6.2.1. PRIMER INTENTO Guardar un hash directamente

Vamos a usar la función `itemHash` del modulo `funcHash` que lógicamente tendrás que importar

```
from funcRSA import setRSAkey, serializeKey
from funcHash import itemHash
```

Esta función recibe un objeto string y una parámetro que es la longitud de la clave del hash

```
nombre = "Alberto Perez"
hash_nombre = itemHash(nombre, 256)
print(type(hash_nombre))
bdoper.insert_user(nombre, serializeKey(key), hash_nombre)
```

ERROR:root:Internal Python error in the inspect module.
Below is the traceback from this internal error.

```
<class 'Crypto.Hash.SHA3_256.SHA3_256_Hash'>
```

Traceback (most recent call last):

File "<string>", line 26, in insert_user

sqlite3.InterfaceError: Error binding parameter 2 - **probably unsupported type.**

Intento fallido. El campo BLOB de la base de datos no entiende lo que es un tipo hash (*recuadro azul*)

6.2.2. SEGUNDO INTENTO Serializar un json.dumps

La forma habitual de serializar objetos JSON es mediante el método dumps. Veamos el siguiente código

```
1 import import_ipynb
2 import environ
3 import sys
4 import os
5 import json
6
7 sys.path.append(os.getenv('MCT_PATH'))
8 from funcHash import itemHash
9
10 hash_original = itemHash('Alberto Pradena', 256)
11 print(hash_original.__class__)
12 dict = {'hash': hash_original}
13
14 hash_json = json.dumps(dict)|
```

Para probar el json.dumps metemos el hash en un diccionario

Si ejecutamos vemos que la línea 13 nos dice que tenemos un objeto hash y luego sale un recuadro de error que recorto porque es muy grande.

```
<class 'Crypto.Hash.SHA3_256.SHA3_256_Hash'>
-----
TypeError                                     Trac
```

Al final de este recuadro podemos leer

```
TypeError: Object of type SHA3_256_Hash is not JSON serializable
```

Lo que significa que json.dumps no es capaz de serializar un diccionario que contiene un hash

6.2.3. TERCER INTENTO Hay que bajar de nivel

Serializar un hash por los métodos convencionales no es fácil. Lo peor además es que dependiendo de la librería que se use puede parecer que se ha conseguido pero uno se da cuenta que al hacer la vuelta atrás (de serializar) no se obtiene realmente un objeto hash aunque el `__class__` diga que si lo es (intenta hacer un *hexdigest* de ese supuesto objeto hash y verás que da error)

Una **primera** solución es intentarlo con el “JSON Encoder and Decoder” (<https://docs.python.org/3.7/library/json.html>). Básicamente consiste en customizar la clase `json.JSONEncoder` con una función que encapsule el hash. Esto me pareció farragoso y sentía que me movía en un terreno en el que no me sentía cómodo

Lo que si fue útil es la idea de encapsular el hash en una clase propia y dotarle de los métodos y propiedades adecuados para luego intentar serializarlo / des serializarlo usando `jsonpickle.encode` y `jsonpickle.decode`

Vamos a probar esta **segunda** solución. Se trata que al instanciar una clase `classHash` se estableciera una propiedad que llamara a la función `hashItem()` y que tratara el valor del hash como propiedad de la clase. Posteriormente habría que hacer pruebas para ver si somos capaces de guardar esa clase serializada en SQLITE y recuperar el valor del hash tras un SELECT

```
import import_ipynb
import environ
import bdoper

import sys
import os
import jsonpickle
import sys

sys.path.append(os.getenv('MCT_PATH'))

from classHash import hashClass
from funcRSA import setRSAkey, serializeKey
```

Si vemos la clase `hashClass` veremos que tiene una propiedad (`@property`) que es obtiene el hash de la cadena pasada al instanciar

```
class hashClass(object):
    def __init__(self, _string, _long):
        self.string = _string
        self.long = _long

    @property
    def hash(self):
        return itemHash(self.string, self.long)
```

Esto permitiría actualizar el hash de forma dinámica si se cambia el valor de `_string` o de `_long` en la llamada (y esto NO ES algo que yo diga que hay que hacer así. Desde el punto de vista de la seguridad habría que ver si esto es bueno tratar el hash como una propiedad dinámica)

Si instanciamos un objeto `hashClass` podemos comprobar su valor y asegurarnos que realmente es un hash SHA3_256 tal y como hemos pasado en la llamada

```
c_hash = hashClass('Arturo Gonzalez',256)
print('Tipo c_hash: ', type(c_hash))
print('Parametros al instanciar {} {}: '.format(c_hash.string, c_hash.long))
print('Propiedad hash: ',c_hash.hash)
print('Hexdigest() del hash: ',c_hash.hash.hexdigest())
```

```
Tipo c_hash: <class 'classHash.hashClass'>
Parametros al instanciar Arturo Gonzalez 256:
Propiedad hash: <Crypto.Hash.SHA3_256.SHA3_256_Hash object at 0x113301690>
Hexdigest() del hash: 135f0b192d72c3ffeeba13eb8ffcc2a1d5e46284852f4387768fc575437bc576
```

Serializamos la clase y vemos que obtenemos algo absolutamente simple

```

class_serializada = jsonpickle.encode(c_hash, keys=True)
print('Tipo class_serializada: ', class_serializada.__class__)
class_serializada

```

Tipo class_serializada: <class 'str'>

```
'{"py/object": "classHash.hashClass", "string": "Arturo Gonzalez", "long": 256}'
```

Vamos a ver si funciona el INSERT en la BBDD

```

key = setRSAkey(2048)
bdoper.deleteTable('tableusers')
bdoper.insert_user(c_hash.string, serializeKey(key), class_serializada)

```

Usuario añadido a la tabla

Tiene buen aspecto. Vamos a comprobar qué es lo que se ha añadido a la tabla. Vemos que el campo hash contiene lo mismo que nos ha dicho el `class_serializada.__class__`

hash
Filtro
{'py/object': 'classHash.hashClass', 'string': 'Arturo Gonzalez', 'long': 256}

Ahora llega la hora de la verdad. Vamos a ver si recuperamos el registro con todos los datos incluido la clase `classHash` y el valor de la propiedad `hash`

```

registro = bdoper.select_user()
registro

```

```

('Arturo Gonzalez',
 b'-----BEGIN RSA PRIVATE KEY-----\nMIIEpQIBAAKCAQEA3jRBQ/qQ/Ahae62QR1CeF3CV5Yxd1tr-
3pl0zo8uQb1ZhnUJ+HchqbgURVa3yi3n0Vm2jaK5diphKqhGLXUAievIg3G\ntGdl8B9L8iDdgr/wmg6fvl
8yB0FVV/ecHHI2lmIUT/\nodMxGNwFtRCu5Kik8pqG0mQfnfS9pJqNlu/71+yqgr24R0PcV8Q79sWmQhTzD
44103/nZfY4Cey1JBr+DPFap3JKYF+9SeoKfYYRpf20UAW\nhKn1LZt6JSAP4QHgS1myaPR4ka8it/A+532l
6m70rI\na0voVWr/IDrM/s1/BDi0UBQBquTtLPW80/43rxEB1qf1Tjk8usJki7WAc0+/UFyo\ndicv7PoQ9l
/IUE2FjebN9D9PdWpSv02b0FuteBTzJB\ny2E29ovuc01CNwlmicFJX0nkUJ7tYMQowRWrum+hY9i1bHe2Yi
gbYsfwL/QUN+xQjpB11s3kNhlerJCeXnvUCu7bLl/F+gqhg26ZV09EpGqN\n2npSKJ3EjN2tUYbvQHD8Bynl
XZrNAWSyv10Lx5UIyD\nMHJ/akECgYEA5XEgUkiQ+YS0fejeWvE1iGbYX52AtLQ7vIF1Mea38Dl+jSW3fw+
qEH004Zsz0SG68jn2cJ1KJeIxx5x0UxLkQbFpLKsJ0U\nCwPAfpjtxl2bqiVBjzq6V2H2ShKXyHNCxxzc8
9+yo\nPxLES2GoZ9k+PmTU/travdl4ZbIu6yHp3Tl0ZcP890HVIXMK09RGMJE3IsWT0f9\nnNIAQWTCik2Ji
VG3ZwWjgPXC+tWr0FkonyX04Woy6ee\n91xSKGNZwvo05INpit9chGgrkfrBZoZyyZBlj/0CgYEA4XC402FI
YJC/KUdiabey//zeFb37gIM4fM4ptKb7GRvRDy5Dvme4m/jCC0LX4LKi\nnx0ZOWS6yg0HxiCaf1xEDs9wDal
n+YqdfWxU7LVJIXs\nnFUyM48jcRjpXR0DoJMuKAeECgYEAgrsQm+V80taTWKD11oK00yHHHYqcn9i6Nhf\i
7xjSeub14frUdDPC7neni3tAhVqtXBjSKfJiDgxhdV\nnT06IKL8vv6SgR8S1z23uSe834R6pV+RiE84vuni
ae\nN/jJ6sECgYEAqXKJKHnub7bYM+6+LWmQJ/JhynwZu/cNY00AEetCA/aHm9DJ7h/D\nnnwNB0g4RGg46VI
3End3pZhr3qyWqR1uF0WpLkx0PG/\nJRIfL9w6UhmJ+55Mh0AUvAQJVIm00DcuVfGCj5bLJpJ3eEMrprJBI
TE KEY-----',
 '{"py/object": "classHash.hashClass", "string": "Arturo Gonzalez", "long": 256}',
 '24/05/2020, 13:31:00')
```


Vemos que devuelve una tupla en cuya posición 2 (empezando por 0) está nuestra claseHash serializada

```
reg_hash = jsonpickle.decode(registro[2])
reg_hash

<classHash.hashClass at 0x113341b10>
```

Según podemos ver tiene buen aspecto. Vamos a comprobar que nuestra clase recuperada tiene la propiedad `hash`

```
reg_hash.hash

<Crypto.Hash.SHA3_256.SHA3_256_Hash at 0x113341f50>
```

Sigue teniendo buen aspecto pero nos queda la **prueba de fuego**

Si recuerdas, cuando instanciamos la clase `c_hash` pusimos unas trazas (*print*) para que, entre otras cosas, se imprimiera el `hexdigest()`. Solamente estaremos seguros que nuestro hash recuperado de la base de datos es el mismo que el original si sus `hexdigest()` son idénticos

Nuestra clase `c_hash` original era

```
Tipo c_hash: <class 'classHash.hashClass'>
Parametros al instanciar Arturo Gonzalez 256:
Propiedad hash: <Crypto.Hash.SHA3_256.SHA3_256_Hash object at 0x11331f590>
Hexdigest() del hash: 135f0b192d72c3ffeeba13eb8ffcc2a1d5e46284852f4387768fc575437bc576
```

Y el `hexdigest()` actual es

```
print(reg_hash.hash.hexdigest())

135f0b192d72c3ffeeba13eb8ffcc2a1d5e46284852f4387768fc575437bc576
```

Lo que significa que hemos sido capaces de *hashear* un string, serializarlo, guardarlo en base de datos, recuperar el objeto serializado, deserializarlo y que coincida

7. Firmar un objeto y verificar firma







Resumiendo

- Ya podemos guardar claves RSA en nuestra base de datos y recuperarlas
- También podemos guardar hashes y recuperarlos manteniendo todas las propiedades de la clase en la que los hemos encapsulado (como los alimentos congelados)

Siguientes pasos

- Vamos a añadir a la base de datos un **campo firma** cuyo contenido será la firma del hash del objeto instanciado. Firmaremos con la clave privada del usuario
- Vamos a incluir un proceso que compruebe que la firma se hizo con la clave privada del usuario y no con otra

Modificamos la base de datos. Incluimos un identificador numérico auto incrementable

▼ tableusers		
	identificador	INTEGER
	nombre	TEXT
	clave_rsa	BLOB
	hash	BLOB
	signed_hash	BLOB
	timestamp	TEXT

7.1. Firma de un hash

Si vemos la clase hashClass veremos que hay un método que es [signHash](#)

```
def signHash(self, _priv_key):
    try:
        # isPrivateKey valida que se trata de un objeto RSA
        assert isPrivateKey(_priv_key)
    except AssertionError:
        sys.exit('Se esperaba recibir una clave publica RSA')

    try:
        signedObject = pss.new(_priv_key).sign(self.hash)
        return signedObject
    except:
        sys.exit('Error al firmar el objeto')
```

Por lo tanto, para firmar un hash solo hay que llamar a ese método pasando la clave privada del firmante. La función signHash devuelve un binario y por tanto no hay que serializar nada

```

key = setRSAkey(2048)
nombre = "Alberto Perez"
c_hash = hashClass(nombre, 256)
signed_hash = c_hash.signHash(key)
signed_hash|

b'|^\x0bj\x1b\xcf\x08\x02k\xab\xd7e;\^ \xb0\\\xc1
d\xa3\x96k\xb1\x9a\xfe\xa4\xcc\x85\xca\x83\x95\
xf0_\xbab\xda\xa5\xbe\x15!\xce\xd7w\xa6ec\x12\x
a5!%\x00\xa8\x9fu\x90\xbf\x007\xc0y9\x9f\x9bVo
b0>\x80\x9e$`.\xc3"\xea\xdc\x8f\x19\xdc\xfd\x1f
cd5%?}=lw\x06y\xf7f\xf3\xb0\x93)2\x90\x14\x06\x
T\xd8N\x0f\xd9\x9e$\xb9\xe4\x91WR\x16\xac\x05\x

```

Vemos que el `signed_hash` ya es binario y no hay que serializarlo

Da la sensación que lo que estoy firmando es toda la clase y no el hash únicamente. Esto no es cierto como puede verse en la línea 32 de `classHash.py`

```

signedObject = pss.new(_priv_key).sign(self.hash)
return signedObject

```

Como se ve, se está firmando la propiedad `hash` de la clase `hashClass`

7.2. Verificación de firma

La clase `hashClass` tiene un método `verifiSign()` que recibe una clave pública y la firma del hash y comprueba que esa firma se hizo con la clave privada correspondiente a la clave pública que ha recibido. Devuelve `True` o `False` en función de si existe correspondencia de claves o no

No tiene que recibir el hash porque ya es una propiedad de la clase que estamos recuperando de la base de datos

8. Nuestro main() al completo

En este punto nuestro programa quedaría como sigue. Primero la parte de importación

```
# Import librerías python
import sys
import os
import jsonpickle

# Import de módulos ipynb
import import_ipynb
import environ
import bdoper

sys.path.append(os.getenv('MCT_PATH'))

from funcRSA import setRSAkey, serializeKey, deSerializeKey
from classHash import hashClass
```

Nos hacemos una lista de usuarios y les vamos generando claves, hashes y firmas para añadirlos a la base de datos

```
# Generar par de claves
key = setRSAkey(2048)
pubkey = key.publickey()

lista_nombres = ["Alberto Perez",
                 "Juan Garcia",
                 "Ana Junquera",
                 "Pedro Garcia",
                 "Emilia Gutierrez",
                 "Francisco Herrador"
                ]

bdoper.deleteTable('tableusers')
for nombre in lista_nombres:
    # Instanciamos la clase hash
    c_hash = hashClass(nombre, 256)

    # Firmamos el hash con la clave privada obtenida
    signed_hash = c_hash.signHash(key)

    # Serializamos la instancia para poder guardarla en BBDD
    clase_serializada = jsonpickle.encode(c_hash, keys=True)

    # INSERT en bbdd
    bdoper.insert_user(nombre, serializeKey(pubkey), clase_serializada, signed_hash)
```

Nos vamos a definir estas dos funciones

```

# Comprobación de firma
# Tenemos en base de datos la clave publica y la firma
# Recuperamos la hashClass (columna 3) del usuario y lo des-serializamos
# para poder obtener el self.hash que fue firmado

def checkSignedHash():
    for record in bdoper.select_all_user():
        user_class = jsonpickle.decode(record[3])
        rsa_key = deSerializeKey(record[2])
        signed_hash = record[4]
        print(f'id: {record[0]} - {record[1]} - \
              Verificación firma: {user_class.verifySign(rsa_key, signed_hash)}')

# Vamos a engañar al sistema para poner una firma falsa
# Generamos un nueva clave y hacemos un UPDATE del signed_hash de un usuario
def updateSignedHash(id_user):
    fake_key = setRSAkey(2048)
    fake_hash = hashClass('Anonymous', 256)
    fake_signed = fake_hash.signHash(fake_key)
    bdoper.update_sign(id_user, fake_signed)

```

`checkSignedHash()` nos saca un listado de todos los registros de la base de datos y comprueba si la firma es correcta

Con `updateSignedHash(id_user)` elegimos un usuario cuya firma vamos a falsear y hacemos el UPDATE con esa firma falsa

Si volvemos a ejecutar `checkSignedHash()` veremos como nos detecta que la firma ha sido falsificada

La siguiente captura muestra el estado de la base de datos y la comprobación de firmas

```

checkSignedHash()
id: 106 - Alberto Perez - Verificación firma: True
id: 107 - Juan Garcia - Verificación firma: True
id: 108 - Ana Junquera - Verificación firma: True
id: 109 - Pedro Garcia - Verificación firma: True
id: 110 - Emilia Gutierrez - Verificación firma: True
id: 111 - Francisco Herrador - Verificación firma: True

```

Vamos a falsear al firma del usuario 110

```

updateSignedHash(110)
Falseada la firma del hash de 110

```

Y volvemos a verificar con `checkSignedHash()`

```
checkSignedHash()
```

```
id: 106 - Alberto Perez - Verificación firma: True  
id: 107 - Juan Garcia - Verificación firma: True  
id: 108 - Ana Junquera - Verificación firma: True  
id: 109 - Pedro Garcia - Verificación firma: True  
id: 110 - Emilia Gutierrez - Verificación firma: False  
id: 111 - Francisco Herrador - Verificación firma: True
```

9. ANEXO A - Variables de sistema entorno en Jupyter

Jupyter no lee variables de entorno definidas en el `.bash_profile` o en el `shell` que uses. Esto es así porque, en contra de lo que pueda parecer, no es un entorno interactivo. Esto significa, dicho de una forma muy básica, que Jupyter no es capaz de actualizar valores una vez arrancado

Para poder manejar variables de entorno en Jupyter tienes que

- instalar la librería `dotenv`
- créate un fichero `.ipynb` que contenga el código (`environ.ipynb` en nuestro caso)

```
%load_ext dotenv
%dotenv manual/config.env
```

Donde “config” es la carpeta donde tu vas a instalar el fichero de configuración con las variables de entorno. Dentro habrá un fichero con extensión “.env” (obligatorio)

- Pon tus variables de entorno en el fichero que has creado

```
# Ruta utilidades criptograficas
MCT_PATH=/Users/...../crypto/utlsMCT

# Ruta base de datos
BBDD_PATH=/Users/...../crypto/utlsMCT/bbdd/bdusers.db

# Ruta ficheros de nombres y claves
FILES_PATH=/Users/...../crypto_mct/files/

# Extension por defecto para ficheros de claves
DEFAULT_EXTENSION=.pem
```

- Obtén el valor de la variable mediante (previamente “import os”)

```
bbdd_path = os.getenv('BBDD_PATH')
```

OJO cuando hagas cambios en el `config\config.env` porque o bien reinicias el kernel o bien usas comandos “reload” (busca “comandos mágicos” en Google y descubrirás un mundo sobre Jupyter)