Tema 1

Criptografia em Python

Objetivos:

- Módulos e bibliotecas Python para lidar com criptografia
- Cálculo de sínteses de ficheiros
- Cálculo de sínteses de senhas
- Cifra de ficheiros com criptografia simétrica e assimétrica

1.1 Introdução

A criptografia é um conceito antigo de transformação de conteúdos, que até há cerca de duas décadas era sobretudo usado em ambientes onde a segurança é um elemento fundamental (ambientes militares, serviços de informações, etc.). Hoje em dia, em virtude da massificação do uso da informática e da Internet para os mais variados fins, a segurança criptográfica faz parte do dia-a-dia do cidadão comum, mesmo que disso ele não se aperceba (por exemplo, quando usa comunicações seguras usando HTTPS).

O objetivo desta aula é o de mostrar como se conseguem realizar as transformações criptográficas mais comuns usando módulos Python.

Para uma referência mais profunda sobre a segurança e as suas aplicações recomenda-se a consulta da bibliografia seguinte [1][2].

1.2 Funções de síntese

As funções de síntese (*digest*) não são funções de cifra convencionais, como as que veremos nas demais secções, mas são funções que usam princípios relacionados com a criptografia no seu modelo de operação. Para além disso, são muitas vezes usadas em conjugação com funções de cifra para as completar de alguma forma (por exemplo, para calcular chaves de cifra de dimensão fixa a partir de senhas de dimensão arbitrária).

O objetivo de uma função de síntese é o de calcular um valor de dimensão fixa (em número de bits) a partir de conteúdos constituídos por conjunto arbitrários de bits. Normalmente diz-se que estas funções permitem criar *impressões digitais informáticas* (*digital fingerprints*) de conteúdos, porque é muito difícil (por requisito) encontrar dois conteúdos que tenham a mesma síntese, assim como é difícil encontrar dois humanos com as mesmas impressões digitais. Também é comum designar o valor calculado por estas funções como *soma de controlo* (*checksum*), muito embora existam inúmeras funções de cálculo de somas de controlo que não possuem as propriedades (criptográficas) das funções de síntese.[1]

As funções de síntese mais usadas são a MD5, a SHA-1, SHA-256 e SHA-512, mas há muitas mais. Estas funções usam-se de forma similar, mas produzem resultados de dimensão diferente (128, 160, 256 e 512 bits, respetivamente).

A forma usual de aplicar uma função de síntese num programa consiste sem seguir os 4 passos seguintes:

1. Iniciar o seu contexto interno;
2. Adicionar dados para serem processados pela função;
3. Repetir o passo anterior até que tenham acabado todos os dados do conteúdo a processar;
4. Calcular a síntese resultante de todos os dados fornecidos à função.

Em Python o módulo **`hashlib`**[2] possui as funções de síntese mais usuais. O exemplo seguinte mostra a utilização da função MD5 para calcular a síntese de uma frase longa dividida em duas partes.

```
$ python3
>>> import hashlib
>>>
>>> h = hashlib.md5()                                  # Iniciar contexto
>>> h.update("A long sentence ".encode("utf-8"))       # Adicionar dados
>>> h.update("broken in two halves".encode("utf-8"))   # Adicionar mais dados
>>> print(h.hexdigest())                               # Calcular síntese
f2b8308b3e84e032f5d7e6dee84e647a
```

O resultado apresentado por este programa é uma sequência de 32 algarismos hexadecimais. Cada um desses algarismos representa 4 bits ($0 \rightarrow 0000$, $1 \rightarrow 0001$, $2 \rightarrow 0010$, $\cdots$, $8 \rightarrow 1000$, $9 \rightarrow 1001$, a$\rightarrow 1010$, $\cdots$, d$\rightarrow 1101$, e$\rightarrow 1110$, f$\rightarrow 1111$). Logo, o resultado possui 32×4 bits, ou seja, 128 bits.

[1] Por outras palavras, as funções de síntese podem ser consideradas como funções de cálculo de somas de controlo, enquanto o inverso não é verdade em geral.

[2] Disponível em `https://docs.python.org/3/library/hashlib.html`

O resultado apresentado seria o mesmo se a frase tivesse sido fornecida apenas de uma vez e não dividida em duas metades:

```
$ python3
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update("A long sentence broken in two halves".encode("utf-8"))
>>> print(h.hexdigest())
f2b8308b3e84e032f5d7e6dee84e647a
```

Porém, qualquer pequena alteração do texto processado pela função de síntese muda o resultado de forma radical:

```
$ python3
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update("A long sentence".encode("utf-8"))       # trailing space removed!
>>> h.update("broken in two halves".encode("utf-8"))
>>> print(h.hexdigest())
1d0b93b21eb945593abab4b1a04456d6
```

Exercício 1.1

Faça um programa que calcule e apresente a síntese de ficheiros usando a função de síntese SHA-1 (cujo nome, no módulo **hashlib**, é **sha1**).

O nome do ficheiro deverá ser passado como argumento ao programa na linha de comandos, acessível através da variável **sys.argv**. Neste tipo de comunicação de dados de entrada com o programa é fundamental validar a existência dos argumentos em número e do tipo pretendido. E terminar o programa em caso de dados insuficientes ou incorretos tal como se apresenta de seguida:

```
import os.path
if len (sys.argv) < 2 :
   print ("Usage: python3 %s filename" % (sys.argv[0]))
   sys.exit (1)

fname = sys.argv[1] # verify if it is a file
if not os.path.exists(fname) or os.path.isdir(fname) or not os.path.isfile(fname):
    print(fname + " is not a file", file=sys.stderr)
    sys.exit (2)
```

Sugestão: abra o ficheiro em modo textual e leia-o linha a linha.

Confirme os resultados apresentados pelo seu programa confrontando-os com os apresentados pelo comando **sha1sum** disponível na linha de comando UNIX.

Exercício 1.2

Reescreva o programa anterior para calcular a síntese de cada ficheiro usando blocos de 512 octetos de cada vez. Use, para esse fim, a função **`read`** para ler octetos dos ficheiros[a]:

```
f = open(name, "rb")
buffer = f.read(512)

# len(buffer) == 0 --> End-of-file reached
# len(buffer) > 0  --> buffer has len(buffer) bytes

while len(buffer) > 0:
    ...
    buffer = f.read(512)
```

Compare os resultados obtidos neste exercício com os do exercício anterior (não deverão mudar!).

[a] Torna-se útil abrir o ficheiro em modo de leitura binária usando "*rb*"

1.3 Biblioteca `pycryptodome`

A biblioteca (*toolkit*) **`pycryptodome`**[3] possui uma coleção muito interessante de funcionalidades relacionadas com criptografia, incluindo funções de síntese. Daqui em diante iremos usar esta biblioteca.

1.3.1 Instalação

O interpretador de Python tipicamente é acompanhado por duas ferramentas que auxiliam a instalação de módulos adicionais, necessários aos programas. Estes programas são respetivamente o **`easy_install`** e o **`pip`**. Para esta aula será necessário instalar a biblioteca **`pycryptodome`**, que fornece mecanismos para a utilização de métodos criptográficos.

Para instalar a biblioteca, no caso de se utilizar um sistema pessoal com Ubuntu ou Debian, será necessário executar:

```
sudo apt install python3-pycryptodome
```

ou em alternativa, usando um método mais demorado

[3] `https://www.dlitz.net/software/pycrypto/`

```
sudo apt-get install build-essential python3-dev python3-pip
```

seguido de:

```
pip3 install --user pycryptodome
```

Repare na utilização da opção `--user`, que indica que as bibliotecas adicionais deverão ser instaladas apenas para o utilizador atual (e não para todo o sistema). Isto é importante pois os utilizadores comuns não possuem permissões para instalar bibliotecas no sistema.

Pode-se confirmar a existência da biblioteca executando os comandos que se apresentam no exemplo seguinte:

1.3.2 Utilização

O programa antes indicado que usava MD5 agora escrever-se-á assim para usar a função MD5 da biblioteca **`pycryptodome`**:

```
$ python3
>>> from Cryptodome.Hash import MD5
>>>
>>> h = MD5.new()
>>> h.update("A long sentence ".encode("utf-8"))
>>> h.update("broken in two halves".encode("utf-8"))
>>> print(h.hexdigest())
f2b8308b3e84e032f5d7e6dee84e647a
```

Exercício 1.3

Altere o programa que antes desenvolveu com a função SHA-1 do módulo **`hashlib`** para usar a função SHA-256 da biblioteca **`pycryptodome`**. Confirme os resultados confrontando-os com os produzidos pelo comando **`sha256sum`**.

1.4 Cifras simétricas

As cifras simétricas são funções de transformação (reversível) de conteúdos que usam duas chaves iguais na cifra e na decifra. Ou seja, se se cifrar um conteúdo original T com a função de cifra E e a chave K, produzindo o criptograma C, poder-se-á recuperar T a partir de C com a função de decifra D e a mesma chave K.

As cifras simétricas subdividem-se em duas grandes famílias: as contínuas (*stream*) e as por blocos.

1.4.1 Cifras contínuas (*stream cyphers*)

As cifras contínuas produzem um criptograma C por mistura de um conteúdo original T com uma chave contínua (*keystream*) KS. A decifra consiste em retirar do criptograma a componente KS que lhe foi misturada usando uma função inversa da de mistura. Por simplicidade, a função de mistura e a sua inversa são exatamente a mesma: a adição módulo 2 de bits, vulgarmente designada por `XOR` (de *eXclusive OR*, cujo símbolo matemático é $\oplus$).

Se A e B forem bits, que podem tomar os valores 0 e 1, a operação $\oplus$ é a seguinte:

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

É fácil de ver que sempre que um operando é 1, o resultado é o inverso do outro operando; pelo contrário, sempre que um operando é 0, o resultado é o valor exato do outro operando. Daqui resulta a seguinte propriedade, para qualquer valor de X:

$$\mathrm{A} \oplus \mathrm{X} \oplus \mathrm{X} = \mathrm{A}$$

Logo, é fácil de mostrar que a cifra e decifra se podem fazer usando a operação `XOR` e a mesma chave contínua KS:

$$\begin{aligned} \mathrm{C} &= \mathrm{T} \oplus \mathrm{KS} \\ \mathrm{T} &= \mathrm{C} \oplus \mathrm{KS} = \mathrm{T} \oplus \mathrm{KS} \oplus \mathrm{KS} = \mathrm{T} \end{aligned}$$

A diferença entre as várias cifras contínuas está na forma como é produzido o valor de KS. A grande maioria das cifras contínuas usa um gerador pseudoaleatório, controlado por uma chave de dimensão fixa, para gerar KS (tanto para a operação de cifra como para a de decifra).

As cifras contínuas são muito usadas em comunicações rádio envolvendo equipamentos móveis, porque existem geradores muito simples de realizar em hardware e de baixo consumo. É o caso dos geradores A5 (usado nas comunicações GSM) e RC4 (usado inicialmente nas comunicações WiFi).

Neste trabalho iremos usar esta última. O seu nome, porém, é diferente: ARC4 (de *Alleged ARC4*). A razão de ser desta diferença de nomes vem do facto do algoritmo RC4 ainda ser oficialmente secreto, sendo o ARC4 uma sua versão que alegadamente (e comprovadamente) é compatível com o RC4 (e que, por isso, é obviamente igual).

A forma usual de usar uma função de cifra/decifra num programa consiste sem seguir os três passos seguintes:

1. Iniciar o seu contexto interno, normalmente indicando uma chave;
2. Adicionar dados para serem processados pela função e recolher o resultado da sua operação;
3. Repetir o passo anterior até que tenham acabado todos os dados do conteúdo a processar.

Note-se que com uma cifra contínua o processamento dos dados (na cifra ou na decifra) é por omissão feito sequencialmente, não se podendo cifrar ou decifrar zonas de dados por uma ordem arbitrária. Há cifras contínuas que permitem essa liberdade, mas a sua utilização é diferente. Neste guião não vamos explorar essa faceta.

Em Python a escrita no monitor de conteúdo binário pode ser feita com a instrução **`write`** para enviar dados para o **`stdout`**, representado pelo seu descritor de ficheiro, tal como se apresenta no exemplo seguinte:

```
import os
from Cryptodome.Cipher import ARC4

cipher = ARC4.new("chave".encode("utf-8"))
cryptogram = cipher.encrypt("Text".encode("utf-8"))
os.write(1, cryptogram)     # o 1 representa o descritor do stdout
print()

decipher = ARC4.new("chave".encode("utf-8"))
decrypted = decipher.decrypt(cryptogram)
print(decrypted.decode("utf-8"))
```

Em Python os descritores dos ficheiros **`stdin`**, **`stdout`** e **`stderr`** são representados pelos números inteiros, respetivamente, 0, 1 e 2.

Exercício 1.4

Faça um programa (**`cifraComRC4.py`**) que use a cifra RC4 (ou ARC4, na biblioteca **`pycryptodome`**). O programa deverá receber como argumentos o nome de um ficheiro a cifrar e uma chave textual. **Sugestão: abra o ficheiro em modo binário e leia o seu conteúdo em blocos.**

O RC4 suporta chaves com uma dimensão entre 40 e 2048 bits (5 a 256 octetos), pelo que deverão ser tomados cuidados para adaptar a chave fornecida pelo utilizador a algo que seja aceitável.

Sugere-se a seguinte política: se a chave tiver menos do que 5 octetos (letras), deverá ser usada uma síntese da mesma (calculada, por exemplo, com SHA-256). Se tiver mais dos que 256 octetos, deverão ser usados apenas os primeiros 256. Caso contrário, deverão ser usados exatamente os octetos fornecidos.

O programa deverá escrever o criptograma para o **`stdout`** (por omissão, a consola), o qual poderá ser redirigido para um ficheiro (**`criptograma`**) usando os mecanismos do interpretador de comandos:

```
python3 cifraComRC4.py ficheiro chave > criptograma
```

Exercício 1.5

Cifre vários ficheiros executando várias vezes o seu programa e verifique se o comprimento dos ficheiros resultantes, contendo os criptogramas, têm a mesma dimensão dos originais (têm de ter!).

Exercício 1.6

Verifique que o programa está a funcionar corretamente decifrando o ficheiro (**`criptograma`**) usando novamente a mesma chave:

```
python3 cifraComRC4.py criptograma chave > textoDecifrado
```

Após executar este comando o ficheiro (**`textoDecifrado`**) deverá ter o conteúdo do ficheiro original (**`ficheiro`**).

Exercício 1.7

Execute o comando anterior usando uma chave diferente e veja o resultado. Explique o sucedido.

1.4.2 Cifras por blocos

As cifras por blocos consideram que os dados a transformar (e o resultado da sua transformação) são constituídos por blocos contíguos de dimensão constante; esta dimensão é imposta pela cifra. O modo mais simples de usar uma cifra (ou decifra) por blocos, denominado EBC (*Electronic Code Book*) consiste em realizar os seguintes passos:

1. Iniciar o seu contexto interno, normalmente indicando uma chave;
2. Selecionar o primeiro bloco dos dados a transformar para serem processados pela função e recolher o bloco resultante da sua operação;
3. Repetir o passo anterior para os blocos seguintes até que tenham acabado todos os dados do conteúdo a processar.

Este processo implica que a dimensão total dos dados a transformar seja múltipla da dimensão do bloco (diz-se que estão alinhados ao bloco). Porém, é natural que nem sempre assim aconteça, o que implica que se tenha de forçar esse alinhamento acrescentando dados extra, denominados excipiente (*padding*).

Estes dados extra são acrescentados na cifra e removidos na decifra. Há várias maneiras de lidar com os excipientes, mas independentemente do método usado, é preciso indicar ao decifrador a sua presença e dimensão.

Uma forma padrão de o fazer, denominada PKCS #7 [3], consiste em fazer o seguinte:

- Acrescentar sempre excipiente, mesmo quando à partida não é necessário (por os dados a cifrar já estarem alinhados);
- Cada octeto do alinhamento tem um valor igual ao comprimento desse alinhamento.

Neste exercício vamos usar a cifra por blocos que é atualmente o padrão, denominada AES (*Advanced Encryption Standard*), que foi a vencedora de um concurso de cifras que terminou pouco depois do início do atual milénio. Esta cifra processa blocos de 128 bits (16 octetos) usando para o efeito chaves de 128, 192 ou 256 bits (16, 24 ou 32 octetos). E vamos usá-la no modo ECB. Quando se usa a biblioteca **`pycryptodome`**, a dimensão do bloco pode ser obtida através da variável **`block_size`** de um objeto de cifra:

```
$ python3
>>> from Cryptodome.Cipher import AES
>>>
>>> key = "1234567890abcdef"   # Must provide a valid key (with 16, 24 or 32 bytes)
>>> cipher = AES.new(key.encode("utf-8"), AES.MODE_ECB)
>>> print(cipher.block_size)     # Prints the number of bytes in each block
16
>>> print(AES.MODE_ECB)           # Prints the cipher mode (1 for ECB)
1
```

Para se cifrar ou decifrar dados devem-se usar os métodos **encrypt** ou **decrypt**, respetivamente, do objeto de cifra:

```
$ python3
>>> from Cryptodome.Cipher import AES
>>>
>>> key = "1234567890abcdef" # key_size = 16 bytes
>>> cipher = AES.new(key.encode("utf-8"), AES.MODE_ECB)
>>> x = cipher.encrypt("texto para cifra".encode("utf-8"))
>>> print(x) # Prints the encrypted text
b'G\xee3\x93\x14\xc5_b__\xb9\xf9Y\xe8\xfd\xa6'

>>> print(cipher.decrypt(x)) # Prints the decrypted text
b'texto para cifra'
```

Exercício 1.8

Faça um programa (**cifraComAES.py**) que use a cifra AES. O programa deverá receber como argumentos o nome de um ficheiro a cifrar e uma chave textual. **Sugestão: abra o ficheiro em modo binário e leia o seu conteúdo em blocos.**

O AES suporta chaves com uma dimensão exata de 16, 24 ou 32 octetos, pelo que deverão ser tomados cuidados para adaptar a chave fornecida pelo utilizador a algo que seja aceitável. Sugere-se a seguinte política: se a chave tiver menos do que 16 octetos (letras), deverá ser usada uma síntese da mesma (calculada, por exemplo, com SHA-256), de cujo resultado serão usados apenas os 16 primeiros octetos. Caso contrário, deverão ser usados apenas os primeiros 16 octetos da senha fornecida.

O programa deverá escrever o criptograma para o **stdout** (por omissão, a consola), o qual poderá ser redirigido para um ficheiro usando os mecanismos do interpretador de comandos.

Exercício 1.9

Faça o programa correspondente de decifra (**decifraComAES.py**). Note que o programa será fundamentalmente igual ao de cifra, mas deverá ter os seguintes cuidados:

- Como os criptogramas estão necessariamente alinhados, não deverá aceitar fazer a decifra de ficheiros que não tenham um comprimento alinhado à dimensão do bloco de cifra. A dimensão de um ficheiro pode ser obtida com a função **os.path.getsize(nome_do_ficheiro)**.
- Não se esqueça de retirar (não escrever) o excipiente colocado durante a cifra. Não se esqueça de que, se usou o método de colocação de excipiente descrito, existe sempre excipiente no último bloco do ficheiro cifrado!

Os exemplos até agora apresentados mostram como encriptar/desencriptar texto. Mas as cifras simétricas por blocos também são aconselhadas para dados numéricos nomeadamente para números inteiros. Neste caso não deverá ser usado alinhamento (*padding*) porque o mesmo é inútil. Para esse efeito cada número inteiro deverá ser guardado num bloco completo a ser processado pela cifra por blocos.

O exemplo seguinte mostra como se pode cifrar e decifrar um número inteiro com esta função de cifra. Basicamente o número inteiro é convertido numa *string* binária com 128 *bits*, usando o formato **`%16d`** na função **`bytes`**, que depois é cifrada por uma chave de cifra binária aleatória de 16 *bytes* (**`key = os.urandom(16)`**). Uma vez que a cifra é aleatória o valor cifrado obtido vai ser diferente em cada simulação. Após a decifragem é preciso converter a *string* numérica para inteiro usando a função **`int`**.

```
$ python3
>>> import os
>>> import random
>>> from Cryptodome.Cipher import AES
>>>
>>> key = os.urandom(16) # random key with 16 bytes
>>> cipher = AES.new(key, AES.MODE_ECB)
>>>
>>> number = 23456
>>> data = cipher.encrypt(bytes("%16d" % (number), "utf8"))
>>> print(data) # Prints the encrypted number
b'\x974^\xc2m\x89\x8b\xd1\xe4\x8a\xec\xcbF~\xee\xfd'

>>> value = int(str(cipher.decrypt(data), "utf8"))
>>> print(value) # Prints the decrypted number
23456
```

Exercício 1.10

Faça um programa que leia números inteiros do teclado, por exemplo, até ser introduzido o número zero. Cada número deve ser cifrado e apresentado no monitor. Depois deve ser decifrado e apresentado no monitor. Faça um programa robusto que assegure que não são aceites valores que não sejam números inteiros.

Verifique o que acontece se introduzir números negativos.

1.5 Cifras Assimétricas

As cifras assimétricas são cifras que usam **duas chaves**, uma para cifrar e outra para decifrar (designadas por par de chaves). Uma destas chaves designa-se por privada e a outra por pública. A privada só é conhecida por uma entidade, que é dona do respetivo par de chaves; a pública pode ser universalmente conhecida. O conhecimento da chave pública não permite a dedução da correspondente chave privada. Estas cifras também são por vezes designadas por *cifras de chave pública*.

As cifras assimétricas, ou de chave pública, são historicamente muito recentes. Enquanto as cifras simétricas são tão antigas quanto a própria escrita, as assimétricas só existem desde meados de década de 1970.

Neste exercício iremos usar a primeira cifra assimétrica que foi publicada, denominada RSA, que é atualmente a mais usada.

1.5.1 RSA

Como se disse, as cifras assimétricas usam pares de chaves, uma privada e outra pública. O RSA permite cifrar com a pública e decifrar com a privada (para garantir confidencialidade) ou o inverso, cifrar com a privada e decifrar com a pública (para garantir autenticidade, o conceito que está na base das assinaturas digitais).

O RSA opera através da realização de operações matemáticas com números inteiros de grande dimensão (centenas ou milhares de bits). As operações são a exponenciação e o resto da divisão por um número inteiro. À combinação destas duas operações dá-se o nome de exponenciação modular.

Um par de chaves RSA possui 3 elementos:

- Um módulo, n, comum às componentes privada e pública;
- Um expoente, d, pertencente à componente privada;
- Um expoente, e, pertencente à componente pública.

Assim, a chave privada é formada pelo par de valores (d, n), enquanto a chave pública é formada pelo par de valores (e, n). As operações de transformação de dados usando estas chaves são as seguintes:

$C = T^e \mod n$	Cifra com a chave pública (confidencialidade)
$T = C^d \mod n$	Decifra com a chave privada
$C = T^d \mod n$	Cifra com a chave privada (autenticidade)
$T = C^e \mod n$	Decifra com a chave pública

onde a expressão $x \mod n$ representa o resto da divisão inteira de x por n (em Python seria calculado com a expressão `x % n`).

Ao contrário das cifras anteriores, as cifras assimétricas não usam chaves indicadas por uma pessoa, nem uma pessoa é capaz de memorizar um par de chaves. Os pares de chaves são gerados por programas, usando para o efeito geradores aleatórios de bits, e as chaves geradas por esses programas têm de ser guardadas algures (por exemplo, em ficheiros) para poderem ser usadas mais tarde.

```
$ python3
>>> from Cryptodome.PublicKey import RSA
>>>
>>> keypair = RSA.generate(1024)
>>> fout = open("keypair.pem", "wb")
>>> kp = keypair.exportKey("PEM", "senha")
>>> fout.write(kp)
>>> fout.close()
...
>>> fin = open("keypair.pem", "rb")
>>> keypair = RSA.importKey(fin.read(), "senha")
>>> fin.close()
>>> print(keypair)
```

O exemplo acima mostra como se gera um par de chaves RSA com um módulo de 1024 bits e se guarda o mesmo num ficheiro (**keypair.pem**) codificando o seu conteúdo em PEM (um formato textual) e protegendo a chave privada da observação de terceiros através da cifra com a senha **senha**. A parte final do exemplo mostra como se recupera a senha do ficheiro.

O conteúdo do ficheiro terá alguma semelhança com o seguinte:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,598E724355A23433

OzlRd4XsS9bTFSR1Y8p/2F29mVd9kth1toy1iBZiBQh6YozcdoHO4Jls0XtMl9iJ
4/H5LWzZRWnKMN7Spmk7ybhi5arLLT9jEmMzkKComgnnnd0UHYgtZLYHQA0rCWCX
3UxXFVl5ngVKN+Poll3FGSfwtGwfWCCmtiBD8XAtGX8WMCoSkh7Tc0YNP2axFcm3
LV59tQ/5hWkXunI1eZxLR39/5Mv97OE4kMyYm57QPAZsOOV0RsTTrbRG2wbRP3Z8
wefoYJtBzEr8TF2aLuS58gBKdplU2KR4Z4suX67TQcs85ahGnG7SvIFwIabxACxN
1je58FF8S9NrD4FI/vLHUad5G0a88XTrqXLkrCCQk/m/A2IrlIA2SkIWxmHHIEFe
XXgUqFxcOWlam11tplRlQW0AlnthmzXe5fH6Y5cz7U1bTqdVyRlxlC7VBpja5hKs
s04ChBZL9x3L/2BoHRVqqjYIiIQv8IKCEeS+FlQKzQs8q9TmzdEq5KBxFrgDN4B1
I+c4ghRqe5MdH+QBOREmM1BWlsfap7utCyu3+etcUKoHrwJRck5nIVkZBOxfPGo6
b1RY+i4ftO9lAIDilfBzZ9r2x8kdT2U+Ztqrz45/yC8S64DrCyOwox6JKd1vmb12
PdjOAg3sbwuuquXYH5+pXwDqLw8NdJYNC87LVwXed5/yjzh2Y3KJ2g3ySZ+YR55V
A0ipRew7KoO9N6CCUHSrEoetf/IjZplrFJWxEf8SuAovvFGArILpRrIbsJl6k/dJ
BkaEeCaOM/QaYo2qqQRG4xXucLbcuQ0nfVXiSUld4Zs=
-----END RSA PRIVATE KEY-----
```

Exercício 1.11

Faça um programa que gere um par de chaves e que o guarde num ficheiro. O programa deverá ter como parâmetros o nome do ficheiro para o par de chaves, a chave de cifra e o número de bits da chave.

A cifra RSA não é usada diretamente tal como indicado anteriormente nas expressões matemáticas. Com efeito, muito embora se usem as operações de exponenciação modular referidas, os valores que as mesmas processam na operações de cifra são pré-processados para obter algumas funcionalidades adicionais (controlo de erros e randomização). Há duas formas fundamentais de fazer esse pré-processamento, designadas por PKCS #1 v1.5 e PKCS #1 OAEP (*Optimal Asymmetric Encryption Padding*).

Neste guião vamos considerar apenas esta última pois é a mais correta de ser utilizada em novas implementações.

```
$ python3
>>> from Cryptodome.PublicKey import RSA
>>> from Cryptodome.Cipher import PKCS1_OAEP
>>>
>>> f = open("keypair.pem", "r")
>>> keypair = RSA.importKey(f.read(), "senha")
>>> cipher = PKCS1_OAEP.new(keypair)
>>> # Encryption with public key
>>> x = cipher.encrypt("The quick brown fox jumps over the lazy dog".encode("utf-8"))
>>> # Decryption with private key
>>> print(cipher.decrypt(x).decode("utf-8"))
The quick brown fox jumps over the lazy dog
```

Exercício 1.12

Altere os programas que usam AES para usarem também o RSA. O objetivo genérico é fazer a cifra do ficheiro com uma chave pública, recorrendo à privada para fazer a sua decifra. Na prática, vai-se recorrer à chamada cifra mista que tem um resultado semelhante mas um custo muito menor em termos de desempenho. O processo é o seguinte:

- Gera-se uma chave simétrica aleatória para cifrar os dados do ficheiro;
- Cifra-se a chave simétrica com a chave pública do destinatário e acrescenta-se o resultado ao ficheiro cifrado.

Para gerar a chave simétrica aleatória use a função **`os.urandom()`** com um parâmetro que indique o número de octetos aleatórios desejados. Use a senha fornecida pelo utilizador para decifrar a chave privada do par de chaves RSA.

1.6 Para Explorar

Exercício 1.13

Avalie a performance das diferentes cifras e sínteses que foram exploradas. Para isto, use o módulo `time` para medir quantas operações são efetuadas em 10 segundos.

Exercício 1.14

Sabe que pode enviar mensagens cifradas mesmo sobre canais não seguros, como um chat? Experimente cifrar mensagens de texto e, antes da impressão, converter o criptograma para Base64 usando o módulo `base64`. O resultado final será um texto com caracteres compatíveis com aplicações de comunicação por mensagens. Antes de as decifrar terá de reverter a codificação de Base64.

Glossário

AES	Advanced Encryption Standard (cifra simétrica por blocos)
ECB	Electronic Code Book (modo de cifra por blocos elementar, sem realimentação)
MD5	Message Digest 5
OAEP	Optimal Asymmetric Encryption Padding
RSA	Cifra assimétrica, acrónimo dos nomes dos criadores (Rivest, Shamir, Adleman)
SHA-1	Secure Hashing Algorithm (versão 1)
SHA-256	Secure Hashing Algorithm (versão 2 com resultado de 256 bits)
SHA-512	Secure Hashing Algorithm (versão 2 com resultado de 512 bits)

Referências

[1] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*, 2nd. New York: John Wiley e Sons Inc., 1995, ISBN: 0–471–12845–7.

[2] A. Zúquete, *Segurança em Redes Informáticas: 6ª Edição Atualizada e Aumentada*. FCA - Editora de Informática, Lda., 2021, ISBN: 978-972-722-923-9.

[3] R. Housley, *Cryptographic Message Syntax (CMS)*, RFC 5652 (Standard), Internet Engineering Task Force, set. de 2009.