Carlos Martín Sanz
Grado en Ingeniería Informática – Mención de Computación

Práctica 4 – Uso de Openssl para cifrado de Mensajes y Ficheros

Parte 1

1.Usa la página de manual de openssl para obtener información sobre la forma de usar esta herramienta para cifrar y descifrar ficheros.

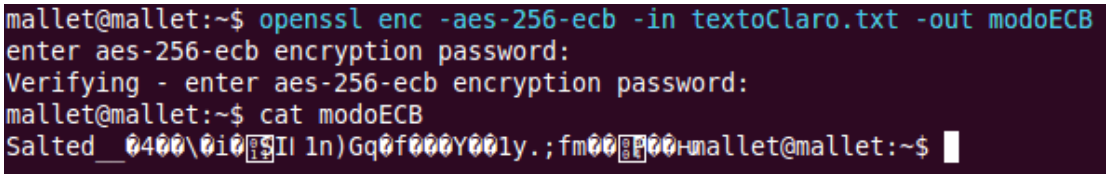
Antes de comenzar con el manual voy a comentar que es openssl. Es una herramienta que contiene un conjunto de funciones que son de gran utilidad para la criptografía aplicada, además OpenSSL implementa los protocolos más conocidos y utilizados en el mundo de la computación, como SSL y TLS.

Para usarlo voy a utilizar una de máquinas virtuales proporcionadas como por ejemplo Mallet, esta es una máquina Linux, y por tanto dispone de la herramienta openssl. Esto lo sabemos ya que ponemos *"openssl"* en la línea de comandos y se modifica el promt a OpenSSL > . Ahora ejecutamos enc -help, y nos va a devolver como se ve en la captura de la parte inferior, una serie de opciones, así como tipos de cifrado. Viendo las opciones, así como esta página web que muestra ejemplos de cifrado (https://www.openssl.org/docs/man1.1.1/man1/enc.html) los comandos para cifrar con openssl serían: "openssl *algoritmoCifrado -in ficheroACifrar -out ficheroCifrado"* para codificar y *"openssl algoritmoCifrado -d -in ficheroCifrado -out ficheroACifrar".*

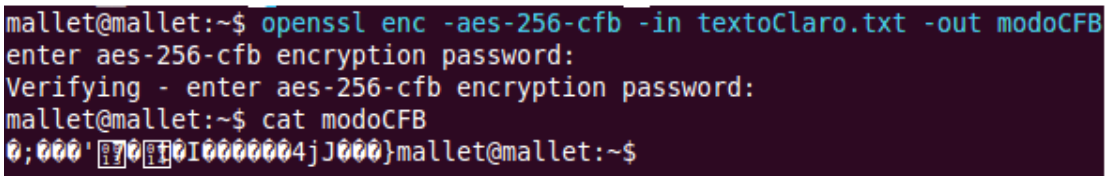
```
mallet@mallet:~$ openssl
OpenSSL> enc -help
unknown option '-help'
options are
-in <file>     input file
-out <file>    output file
-pass <arg>    pass phrase source
-e             encrypt
-d             decrypt
-a/-base64     base64 encode/decode, depending on encryption flag
-k             passphrase is the next argument
-kfile         passphrase is the first line of the file argument
-md            the next argument is the md to use to create a key
                 from a passphrase.  One of md2, md5, sha or sha1
-K/-iv         key/iv in hex is the next argument
-[pP]          print the iv/key (then exit if -P)
-bufsize <n>   buffer size
-engine e      use engine e, possibly a hardware device.
Cipher Types
-aes-128-cbc               -aes-128-cfb               -aes-128-cfb1
-aes-128-cfb8              -aes-128-ecb               -aes-128-ofb
-aes-192-cbc               -aes-192-cfb               -aes-192-cfb1
-aes-192-cfb8              -aes-192-ecb               -aes-192-ofb
-aes-256-cbc               -aes-256-cfb               -aes-256-cfb1
-aes-256-cfb8              -aes-256-ecb               -aes-256-ofb
-aes128                    -aes192                    -aes256
-bf                        -bf-cbc                    -bf-cfb
-bf-ecb                    -bf-ofb                    -blowfish
-cast                      -cast-cbc                  -cast5-cbc
-cast5-cfb                 -cast5-ecb                 -cast5-ofb
-des                       -des-cbc                   -des-cfb
-des-cfb1                  -des-cfb8                  -des-ecb
-des-ede                   -des-ede-cbc               -des-ede-cfb
-des-ede-ofb               -des-ede3                  -des-ede3-cbc
-des-ede3-cfb              -des-ede3-ofb              -des-ofb
-des3                      -desx                      -desx-cbc
-rc2                       -rc2-40-cbc                -rc2-64-cbc
-rc2-cbc                   -rc2-cfb                   -rc2-ecb
-rc2-ofb                   -rc4                       -rc4-40
```

2. Experimenta con diversos algoritmos (AES, DES, CAMELLIA,) y modos de cifrado (ECB. CFB. CBC). Utiliza diferentes ficheros de entrada (texto y binarios) y con diferentes claves y vectores de inicialización.

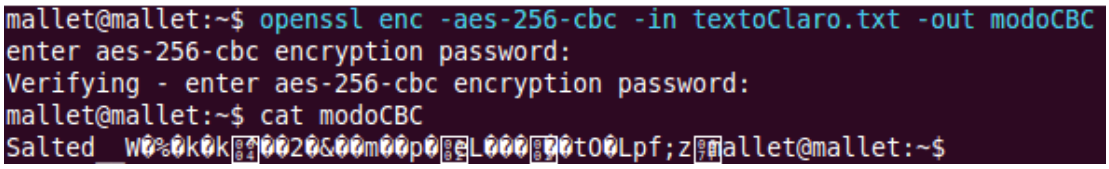
Para ello creo un archivo llamado textoClaro.txt con algo escrito, y lo cifro primero con el algoritmo AES 256, y este algoritmo tiene 3 modos diferentes de cifrado por tanto primero utilizando ECB, como clave de encriptación utilizo “123456”:



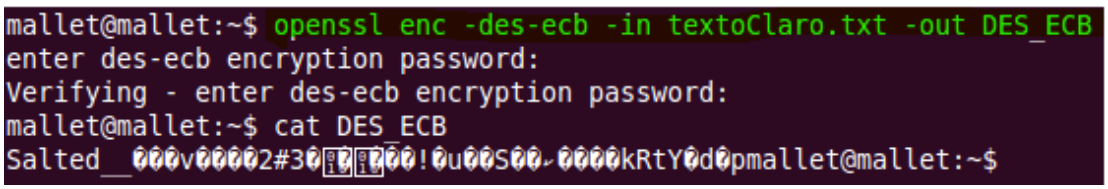
Ahora utilizando CFB:



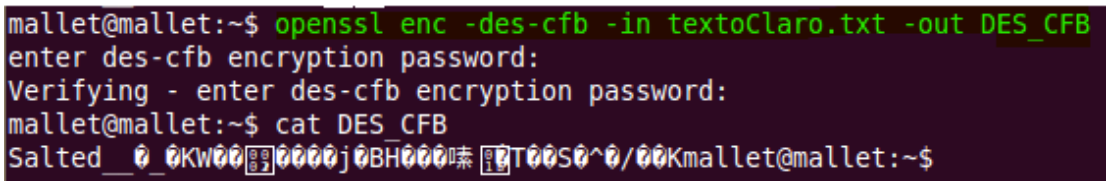
Y por último utilizo CBC:



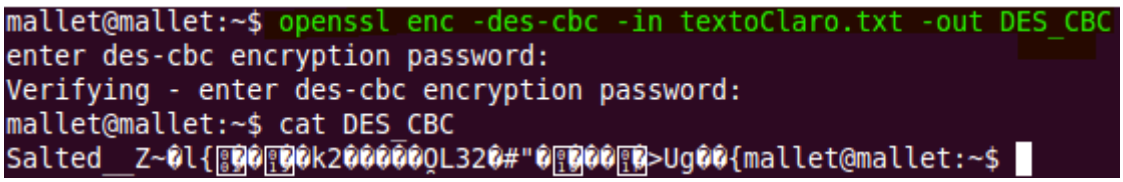
Ahora en vez de utilizar AES 256 cambio el algoritmo de cifrado y utilizo DES con sus 3 modos diferentes de cifrado, y cambio la clave de encriptación a “abcd”, primero empiezo utilizando ECB:



Ahora utilizando CFB:



Y por último utilizo CBC:



Pero antes un pequeño inciso, como se puede ver en la captura del apartado 1 de la práctica en Cipher Types, mallet no tiene CAMELLIA, por tanto, a partir de ahora cambio la máquina en la que voy a continuar con la realización de la práctica, voy a pasar a utilizar el Ubuntu 20.0.4 LTS de la práctica anterior. Como podemos ver ahora, esta nueva máquina sí que dispone de CAMELLIA.

```
Cipher commands (see the `enc' command for more details)
aes-128-cbc        aes-128-ecb        aes-192-cbc        aes-192-ecb
aes-256-cbc        aes-256-ecb        aria-128-cbc       aria-128-cfb
aria-128-cfb1      aria-128-cfb8      aria-128-ctr       aria-128-ecb
aria-128-ofb       aria-192-cbc       aria-192-cfb       aria-192-cfb1
aria-192-cfb8      aria-192-ctr       aria-192-ecb       aria-192-ofb
aria-256-cbc       aria-256-cfb       aria-256-cfb1      aria-256-cfb8
aria-256-ctr       aria-256-ecb       aria-256-ofb       base64
bf                 bf-cbc             bf-cfb             bf-ecb
bf-ofb             camellia-128-cbc   camellia-128-ecb   camellia-192-cbc
camellia-192-ecb   camellia-256-cbc   camellia-256-ecb   cast
cast-cbc           cast5-cbc          cast5-cfb          cast5-ecb
cast5-ofb          des                des-cbc            des-cfb
des-ecb            des-ede            des-ede-cbc        des-ede-cfb
des-ede-ofb        des-ede3           des-ede3-cbc       des-ede3-cfb
des-ede3-ofb       des-ofb            des3               desx
rc2                rc2-40-cbc         rc2-64-cbc         rc2-cbc
rc2-cfb            rc2-ecb            rc2-ofb            rc4
rc4-40             seed               seed-cbc           seed-cfb
seed-ecb           seed-ofb           sm4-cbc            sm4-cfb
sm4-ctr            sm4-ecb            sm4-ofb
```

Y para acabar con el cifrado de texto claro, el último algoritmo de cifrado que se pide, voy a utilizar CAMELLIA-256 con sus 3 modos diferentes de cifrado, y modifico la clave de cifrado a “a1b2c3”, primero empiezo utilizando ECB:

```
carlos@Ubuntu20:~$ openssl enc -camellia-256-ecb -in textoClaro.txt -out C-ECB
enter camellia-256-ecb encryption password:
Verifying - enter camellia-256-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ cat C-ECB
◆carlos@Ubuntu20:~$ ◆$◆◆Y◆◆◆v}=2◆SP2◆◆◆Vr◆
```

Ahora utilizando CFB:

```
carlos@Ubuntu20:~$ openssl enc -camellia-256-cfb -in textoClaro.txt -out C-CFB
enter camellia-256-cfb encryption password:
Verifying - enter camellia-256-cfb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ cat C-CFB
Salted__ilFZ◆Z◆(◆◆6◆j◆◆FT~;◆◆◆U$ê"carlos@Ubuntu20:~$
```

Y por último utilizo CBC:

```
carlos@Ubuntu20:~$ openssl enc -camellia-256-cbc -in textoClaro.txt -out C-CBC
enter camellia-256-cbc encryption password:
Verifying - enter camellia-256-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ cat C-CBC
 jG◆◆◆/◆◆◆◆◆0H◆◆R◆vz◆◆恋◆◆      W◆carlos@Ubuntu20:~$
```

Pero de momento solo hemos cifrado textos en claro, como pide el enunciado también necesitamos archivos binarios. Para ello he buscado en internet y la ejecución de este comando: "xxd -b textoClaro.txt textoBin.bin" te convierte el contenido de textoClaro.txt a binario y lo guarda en textoBin.bin como se puede ver en la siguiente captura.

```
carlos@Ubuntu20:~$ xxd -b textoClaro.txt  textoBin.bin
carlos@Ubuntu20:~$ cat textoBin.bin
00000000: 01100101 01110011 01110100 01101111 00100000 01100101  esto e
00000006: 01110011 00100000 01110101 01101110 01100001 00100000  s una
0000000c: 01110000 01110010 01110101 01100101 01100010 01100001  prueba
00000012: 00100000 01100100 01100101 00100000 01100011 01101001   de ci
00000018: 01100110 01110010 01100001 01100100 01101111 00001010  frado.
```

Como ahora este nuevo fichero es binario, voy a realizar el cifrado, usando este fichero como fichero de entrada. Al igual que antes primero voy a utilizar el algoritmo AES-256 con sus 3 modos de cifrado ECB, CFB, CBC en este orden:

```
carlos@Ubuntu20:~$ openssl enc -aes-256-ecb -in textoBin.bin -out AES_ECB
enter aes-256-ecb encryption password:
Verifying - enter aes-256-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ cat AES_ECB
Salted__◆◆ip◆!
             ◆µ◆◆}M◆h7H/<w2@◆S◆◆(◆◆sd$◆◆
◆◆◆a◆◆)p◆◆◆%◆◆L◆$◆+◆◆◆/◆◆◆`◆◆◆zp◆Y?G◆+◆◆◆◆◆◆ ]◆W◆◆,>◆◆n◆t◆GyK-◆◆◆n[◆B/◆
                                                                ◆◆◆"W◆◆
V`◆/}
     EiO+◆◆◆
m◆3S◆H◆
◆◆H◆◆◆H◆f◆bo?◆[illegible]H[illegible]◆◆I◆)◆◆◆[illegible]◆H◆v◆.◆8◆◆&?]◆◆◆◆◆.◆d}H◆◆5_y◆f6◆◆◆8
                                                          ◆◆Tjl◆m◆[illegible]◆      ◆
1dFd◆◆^◆X+◆,1%[illegible]◆◆◆◆(◆`qB4/◆}f◆◆◆&ń0◆◆y"◆[illegible]◆n◆◆[illegible]'◆c◆◆◆L◆◆a◆9◆z5;M◆carlos@Ubuntu
20:~$
```

Como podemos observar el archivo que contiene la información cifrada es bastante extenso, y para evitar alagar la práctica, ya no voy a mostrar el cifrado únicamente el comando de cifrado que utilizo.

```
carlos@Ubuntu20:~$ openssl enc -aes-256-cfb -in textoBin.bin -out AES_CFB
enter aes-256-cfb encryption password:
Verifying - enter aes-256-cfb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ openssl enc -aes-256-cbc -in textoBin.bin -out AES_CBC
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

Ahora que ya he cifrado con AES al igual que antes voy a cambiar el algoritmo de cifrado a DES, y siguiendo el mismo orden con los distintos 3 modos de cifrado primero ECB, CFB y CBC:

```
carlos@Ubuntu20:~$ openssl enc -des-ecb -in textoBin.bin -out DES_ECB
enter des-ecb encryption password:
Verifying - enter des-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ openssl enc -des-cfb -in textoBin.bin -out DES_CFB
enter des-cfb encryption password:
Verifying - enter des-cfb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ openssl enc -des-cbc -in textoBin.bin -out DES_CBC
enter des-cbc encryption password:
Verifying - enter des-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

Y por último y de la misma manera que en los anteriores, pero modificando el algoritmo de cifrado a CAMELLIA.

```
carlos@Ubuntu20:~$ openssl enc -camellia-256-ecb -in textoBin.bin -out CAM_ECB
enter camellia-256-ecb encryption password:
Verifying - enter camellia-256-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ openssl enc -camellia-256-cfb -in textoBin.bin -out CAM_CFB
enter camellia-256-cfb encryption password:
Verifying - enter camellia-256-cfb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ openssl enc -camellia-256-cbc -in textoBin.bin -out CAM_CBC
enter camellia-256-cbc encryption password:
Verifying - enter camellia-256-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

3. Obtenga información sobre el concepto de entropía de Shannon y elabore un breve informe y discusión sobre el tema. Usando el programa Python (enlaces interesantes). obtenga la entropía de los ficheros usados en el apartado segundo (tanto cifrados como descifrados), así como la del fichero obtenido en el apartado cuarto y la de un fichero que contenga un único byte repetido un número de veces arbitrario.

Primero voy a comentar que es y en qué consiste la entropía de Shannon. El concepto de entropía o cantidad de información sirve para medir la incertidumbre de una fuente de información.

La entropía de Shannon se podría calcular de esta forma, dada una variable aleatoria X:

$$H(X) = -\sum_{\forall x \in X} p(x) \log_2 p(x)$$

Como se puede observar en la fórmula la entalpía depende muy directamente de la distribución de probabilidad.

En esta página web (https://code.activestate.com/recipes/577476-shannon-entropy-calculation/) he conseguido encontrar un código que calcula la entropía de Shannon de un fichero que recibe como parámetro. Por tanto, creo un archivo Python llamado entropy_shannon.py donde voy a copiar dicho código de internet, realizando unas pequeñas modificaciones ya que realizando únicamente copia y pega no compila.

Este es el código que lo realiza:

```
# Shannon Entropy of a file
# FB - 201012153
import sys
import math
if len(sys.argv) != 2:
    print 'Usage: file_entropy.py [path]filename'
    sys.exit()
f = open(sys.argv[1], 'rb')
byteArr = bytearray(f.read())
f.close()
fileSize = len(byteArr)
print 'File size in bytes:', fileSize
print

freqList = [0] * 256
for b in byteArr:
    freqList[b] += 1

ent = 0.0
for f in freqList:
    if f > 0:
        freq = float(f) / fileSize
        ent = ent + freq * math.log(freq, 2)
ent = -ent
print 'Shannon entropy (min bits per byte-character):', ent
print
print 'Min possible file size assuming max theoretical compression efficiency:'
print (ent * fileSize) / 8, 'bytes'
```

Y ahora voy a ejecutar este código para todos los archivos que pide el enunciado.

Primero voy a empezar por los ficheros descifrados, es decir, el fichero de texto claro (textoClaro.txt) y también el fichero binario (textoBin.bin).

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py textoClaro.txt
File size in bytes,  30

Shannon entropy (min bits per byte-character)
 3.7614063297218423

Min possible file size assuming max theoretical compression efficiency:
  14.105273736456908  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py textoBin.bin
File size in bytes,  360

Shannon entropy (min bits per byte-character)
 2.194094346156106

Min possible file size assuming max theoretical compression efficiency:
  98.73424557702478  bytes.
```

Ahora que ya he hallado las entropías de los textos sin descifrar, voy a hallar las entropías de los ficheros cifrados generados a partir del fichero textoClaro.txt que como indica su nombre contiene texto claro:

CIFRADOS CON AES 256

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py AES_ECB_C
File size in bytes,  48

Shannon entropy (min bits per byte-character)
 5.376629167387826

Min possible file size assuming max theoretical compression efficiency:
  32.259775004326954  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py AES_CFB_C
File size in bytes,  46

Shannon entropy (min bits per byte-character)
 5.289760053836066

Min possible file size assuming max theoretical compression efficiency:
  30.416120309557378  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py AES_CBC_C
File size in bytes,  48

Shannon entropy (min bits per byte-character)
 5.31923567775942

Min possible file size assuming max theoretical compression efficiency:
  31.915414066556522  bytes.
```

CIFRADOS CON DES

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py DES_ECB_C
File size in bytes,  48

Shannon entropy (min bits per byte-character)
 5.334962500721159

Min possible file size assuming max theoretical compression efficiency:
  32.009775004326954  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py DES_CFB_C
File size in bytes,  46

Shannon entropy (min bits per byte-character)
 5.393127173448314

Min possible file size assuming max theoretical compression efficiency:
  31.010481247327807  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py DES_CBC_C
File size in bytes,  48

Shannon entropy (min bits per byte-character)
 5.376629167387826

Min possible file size assuming max theoretical compression efficiency:
  32.259775004326954  bytes.
```

CIFRADOS CON CAMELLIA 256

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py C-ECB
File size in bytes,  48

Shannon entropy (min bits per byte-character)
 5.33496250072116

Min possible file size assuming max theoretical compression efficiency:
  32.009775004326954  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py C-CFB
File size in bytes,  46

Shannon entropy (min bits per byte-character)
 5.26269239083962

Min possible file size assuming max theoretical compression efficiency:
  30.260481247327814  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py C-CBC
File size in bytes,  48

Shannon entropy (min bits per byte-character)
 5.501629167387827

Min possible file size assuming max theoretical compression efficiency:
  33.00977500432696  bytes.
```

Hasta ahora he hallado las entropías de los ficheros de texto claro cifrados, pero faltan para los ficheros binarios cifrados.

CIFRADOS CON AES 256

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py AES_ECB
File size in bytes,  384

Shannon entropy (min bits per byte-character)
 7.398104486494482

Min possible file size assuming max theoretical compression efficiency:
  355.10901535173514  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py AES_CFB
File size in bytes,  376

Shannon entropy (min bits per byte-character)
 7.322325745304481

Min possible file size assuming max theoretical compression efficiency:
  344.14931002931064  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py AES_CBC
File size in bytes,  384

Shannon entropy (min bits per byte-character)
 7.401019630970431

Min possible file size assuming max theoretical compression efficiency:
  355.2489422865807  bytes.
```

CIFRADOS CON DES

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py DES_ECB
File size in bytes,  384

Shannon entropy (min bits per byte-character)
 7.346510436032723

Min possible file size assuming max theoretical compression efficiency:
  352.6325009295707  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py DES_CFB
File size in bytes,  376

Shannon entropy (min bits per byte-character)
 7.366836293254817

Min possible file size assuming max theoretical compression efficiency:
  346.2413057829764  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py DES_CBC
File size in bytes,  384

Shannon entropy (min bits per byte-character)
 7.454481414858408

Min possible file size assuming max theoretical compression efficiency:
  357.81510791320363  bytes.
```

CIFRADOS CON CAMELLIA 256

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py CAM_ECB
File size in bytes,  384

Shannon entropy (min bits per byte-character)
 7.436461403039277

Min possible file size assuming max theoretical compression efficiency:
  356.9501473458853  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py CAM_CFB
File size in bytes,  376

Shannon entropy (min bits per byte-character)
 7.416605026026616

Min possible file size assuming max theoretical compression efficiency:
  348.58043622325096  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py CAM_CBC
File size in bytes,  384

Shannon entropy (min bits per byte-character)
 7.537227345876161

Min possible file size assuming max theoretical compression efficiency:
  361.78691260205574  bytes.
```

Puesto que todavía no he comenzado a realizar el apartado 4 primero voy a calcular la entropía de un fichero que contenga un único byte repetido un número de veces arbitrario. El fichero se va a llamar byteRep.txt, como se ve en esta captura.

```
carlos@Ubuntu20:~$ vi byteRep.txt
carlos@Ubuntu20:~$ cat byteRep.txt
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
```

Ahora como el fichero contiene un único byte que esta repetido muchas veces, el valor de la entropía de Shannon debería de ser muy bajo, ya que puede tomar valores ente 0 y 8. Siendo 0 el mínimo valor posible que puede tomar (cuando el fichero está vacío) y 8 el valor máximo. De nuevo voy a ejecutar el programa para ver qué valor toma la entropía:

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py byteRep.txt
File size in bytes,  63

Shannon entropy (min bits per byte-character)
 0.11759466565886476

Min possible file size assuming max theoretical compression efficiency:
  0.92605799206356  bytes.
```

Como hemos supuesto en el párrafo anterior el valor de la entropía es bajo 0.1176.

Ahora voy a calcular la entropía de la imagen que he seleccionado para el apartado 4, así como de ambas imágenes cifradas “copia_imagen_ecb.bmp” y “copia_imagen_cbc.bmp”. Primero voy a hallar la entropía de la imagen original.

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py index.bmp
File size in bytes,  202638

Shannon entropy (min bits per byte-character)
 5.324012871977199

Min possible file size assuming max theoretical compression efficiency:
  134855.91504396446  bytes.
```

Y ahora de ambas copias cifradas, primero la hallada con el modo ECB y después con el modo CBC:

```
carlos@Ubuntu20:~$ python3 entropy_shannon.py copia_imagen_ecb.bmp
File size in bytes,  202656

Shannon entropy (min bits per byte-character)
 7.922329081354241

Min possible file size assuming max theoretical compression efficiency:
  200688.44028886565  bytes.
carlos@Ubuntu20:~$ python3 entropy_shannon.py copia_imagen_cbc.bmp
File size in bytes,  202656

Shannon entropy (min bits per byte-character)
 7.999147259536068

Min possible file size assuming max theoretical compression efficiency:
  202634.39837856768  bytes.
```

Como se puede ver, tanto en los textos como en las imágenes cifradas, pero sobretodo en estas últimas, el valor de la entropía es mucho mayor y cercano a 8. Sin embargo, en los fichero o imágenes sin cifrar la entropía toma valores menores, entorno a 5. Esto es debido a que al cifrarlo altera los bytes para que no puedan ser interpretados, y como hemos visto la repetición de los bytes está muy relacionado con el valor que toma la entropía de Shannon.

4. Obtenga de la red un fichero que contenga una imagen en formato BMP (libre de derechos, a poder ser) y cífrelo usando AES con modos ECB y CBC. Salve copias de la imagen cifrada, sustituya por la cabecera (54 bytes) del fichero original las cabeceras de los ficheros obtenidos y cárguelos en un programa de visualización de imágenes. Comente el resultado. Nota: Para trasladar la cabecera de un fichero in.bmp a otro out.bmp dejando el resto inalterado, se puede usar: 'dd if=in.bmp of=out.bmp bs=54 count=1 conv=notrunc'

Para ello he buscado en internet imágenes con formato bmp, (https://people.math.sc.edu/Burkardt/data/bmp/snail.bmp) y en esta página he encontrado una serie de imágenes que te permitían descargar formato bmp, gratis. La imagen elegida es esta:

Ahora voy cifrar dicha imagen con algunos de los algoritmos de cifrado que hemos utilizado antes, nos pide con AES con los modos ECB y CBC:

```
carlos@Ubuntu20:~$ openssl enc -aes-256-ecb -in snail.bmp -out imagencifrada_ae
s-ecb.bmp
enter aes-256-ecb encryption password:
Verifying - enter aes-256-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
carlos@Ubuntu20:~$ openssl enc -aes-256-cbc -in snail.bmp -out imagencifrada_ae
s-cbc.bmp
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

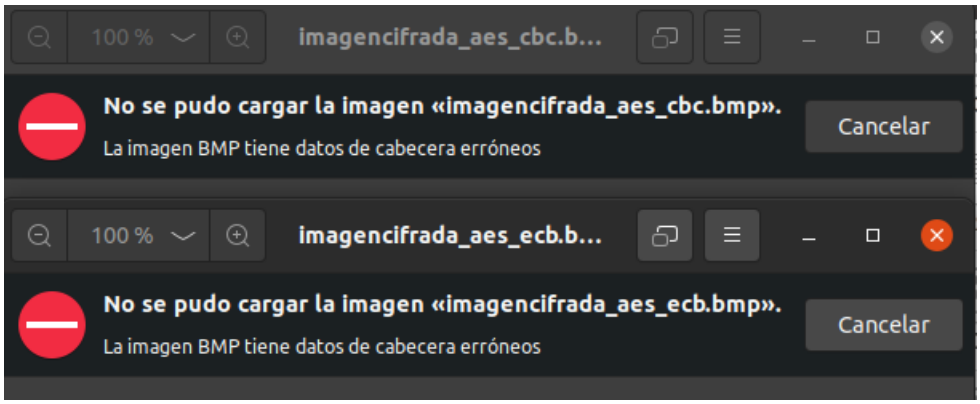
Realizo las dos copias de las imágenes cifradas con el comando *"cp ficheroACopiar nombreCopia".* Y después ejecuto el comando *"ls"* para comprobar que se han creado dichas copias.

```
carlos@Ubuntu20:~$ cp imagencifrada_aes_ecb_.bmp copia_imagen_ecb.bmp
carlos@Ubuntu20:~$ cp imagencifrada_aes_cbc_.bmp copia_imagen_cbc.bmp
carlos@Ubuntu20:~$ ls
AES_CBC       C-CBC                 DES_CFB_C                 Plantillas
AES_CBC_C     C-CFB                 DES_ECB                   Público
AES_CFB       C-ECB                 DES_ECB_C                 sinRep.txt
AES_CFB_C     contrasenas.txt       Documentos                snail.bmp
AES_ECB       copia_imagen_cbc.bmp  entropy_shannon.py        textoBin.bin
AES_ECB_C     copia_imagen_ecb.bmp  Escritorio                textoClaro.txt
byteRep.txt   Descargas             imagencifrada_aes_cbc_.bmp Videos
CAM_CBC       DES_CBC               imagencifrada_aes_ecb_.bmp
CAM_CFB       DES_CBC_C             Imágenes
CAM_ECB       DES_CFB               Música
```

Ahora falta sustituir la cabecera de las copias por la cabecera de la imagen original (la descargada). Utilizando el comando que dice el enunciado del apartado.

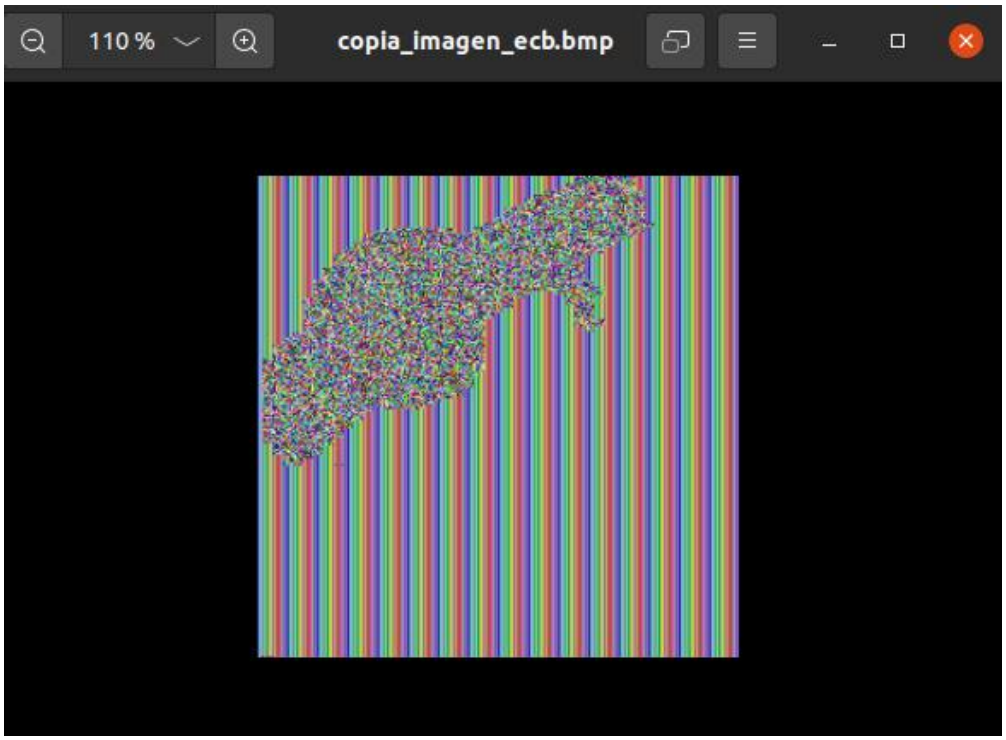
```
carlos@Ubuntu20:~$ dd if=snail.bmp of=copia_imagen_ecb.bmp bs=54 count=1 conv=n
otrunc
1+0 registros leídos
1+0 registros escritos
54 bytes copied, 0,000142677 s, 378 kB/s
carlos@Ubuntu20:~$ dd if=snail.bmp of=copia_imagen_cbc.bmp bs=54 count=1 conv=n
otrunc
1+0 registros leídos
1+0 registros escritos
54 bytes copied, 0,00013923 s, 388 kB/s
```

A la hora de intentar visualizar las imágenes, las imágenes que han sido cifradas tanto con ECB como con CBC, no deja visualizarlas.

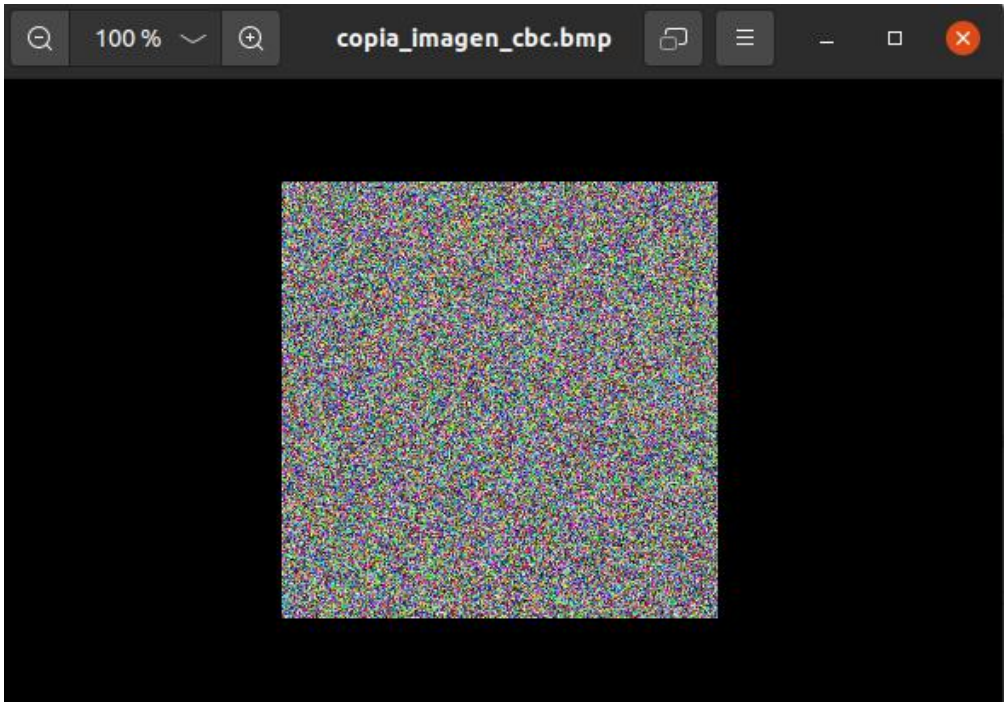


Pero las imágenes que son copias de las cifradas cuya cabecera ha sido modificada, sustituyendo la cabecera de la imagen cifrada, por la de la imagen original nos permite visualizarla. Esto es el resultado que obtenemos:

CIFRADA CON AES, MODO ECB



CIFRADA CON AES, MODO CBC



Como las imágenes han sido cifradas, para ninguno de los dos métodos se permite distinguir la imagen original, únicamente los píxeles alterados de forma que sea imposible. Aunque en el modo de cifrado ECB, se permite distinguir la forma de la figura del caracol, pero no se llega a distinguir nada, es decir, se sabe que hay algo por la forma pero no el qué.

Parte 2

1. Selecciona un servidor web público (puedes utilizar el de la UVa) e investiga cuáles son los puertos a nivel de transporte que utiliza para brindar el servicio web. ¿Qué sentido tiene?

Para ello selecciono el servidor web de la UVa (www.uva.es). Los puertos a nivel de transporte que utiliza para brindar el servicio web y por tanto están abiertos, son el puerto 80 y 443 TCP.

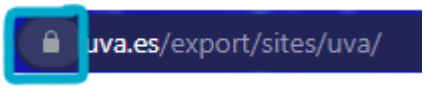
El puerto 80 es el que se utiliza para la navegación web usando el protocolo HTTP (HyperText Transfer Protocol) este protocolo no es seguro ya que la conexión no está protegida de posibles ataques y escuchas.

El puerto 443 es el que se utiliza para la navegación web utilizando el protocolo HTTPS (HyperText Transfer Protocol Secure), este protoco sí que es seguro ya que implementa el protocolo de seguridad TLS (Transport Layer Security). Además aporta al cliente un certificado digital que asegura que el sitio web en concreto posee confidencialidad, integridad, autenticidad, disponibilidad y no repudio.

Por tanto, con la herramienta nmap podemos ver si estos dos puertos de nivel de tansporte se encuentran abiertos. Como podemos ver en la siguiente captura ejecutando el comando *"nmap www.uva.es -p80"* y *"nmap www.uva.es -p443"*, ambos puertos para dicho servidor web, están abiertos.

```
carlos@Ubuntu20:~$ nmap www.uva.es -p80
Starting Nmap 7.80 ( https://nmap.org ) at 2022-11-10 12:19 CET
Nmap scan report for www.uva.es (157.88.25.8)
Host is up (0.061s latency).
rDNS record for 157.88.25.8: sostenibilidad.uva.es

PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.46 seconds
carlos@Ubuntu20:~$ nmap www.uva.es -p443
Starting Nmap 7.80 ( https://nmap.org ) at 2022-11-10 12:19 CET
Nmap scan report for www.uva.es (157.88.25.8)
Host is up (0.018s latency).

PORT    STATE SERVICE
443/tcp open  https

Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

Cuando nosotros realizamos una búsqueda a una página web que utiliza el puerto 443, es decir conexión segura, podemos ver que nos aparece un candado en la parte superior izquierda de la barra de búsqueda.



Ahora suponga que en vez de conectarnos a dicho servicio de manera segura (https://www.uva.es). Nos conectamos poniendo (http://www.uva.es).

De forma automática el servidor web nos redirige del puerto 80 TCP al 443 TCP de esta forma pasamos de usar HTTP a usar HTTPS, es decir pasamos a un modo seguro. Esta redirección se puede observar en el apartado Network del menú de desarrollador.

www.uva.es	307	document / Redirect	Other
www.uva.es	302	document / Redirect	www.uva.es/
uva/	302	document / Redirect	www.uva.es/
uva/	302	document	/export/sites/uva/

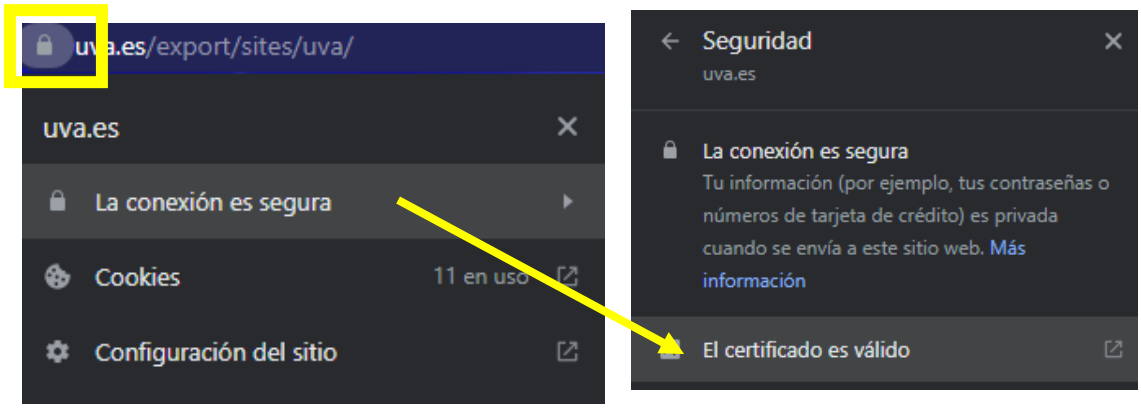
El código de estado 302 que se ve en la captura, es el código de respuesta que http cuando se produce una redirección. En este caso del puerto 80 al 443, o lo que es lo mismo de utilizar el protocolo HTTP a HTTPS.

Respecto a la pregunta ¿Qué sentido tiene?, nos permite que la comunicación entre el cliente y el servidor web vaya cifrada. Ya que si únicamente tuviese abierto el puerto 80 TCP las conexiones se realizarían sin seguridad y serian mucho más vulnerables a posibles ataques. Esto es más difícil que ocurra si la conexión se efectúa a través del puerto 443, ya que las conexiones se realizan de manera segura ya que están cifradas.

2. Analiza el certificado digital presenta en el servidor web. Puedes hacer uso de las herramientas sslscan y ssltest del apartado anterior e indica:

Carlos Martín Sanz
Grado en Ingeniería Informática – Mención de Computación

Antes de comenzar a responder las preguntas concretas sobre el certificado digital que presenta el servidor web, en este caso www.uva.es voy a comentar brevemente como se accede a este, siguiendo los pasos de las siguientes capturas:



Clicando en "El certificado es válido" ya estaríamos accediendo al certificado digital que presenta en este caso el servidor web de la UVa.

a. Protocolo/s criptográfico y versión utilizado a nivel de transporte.

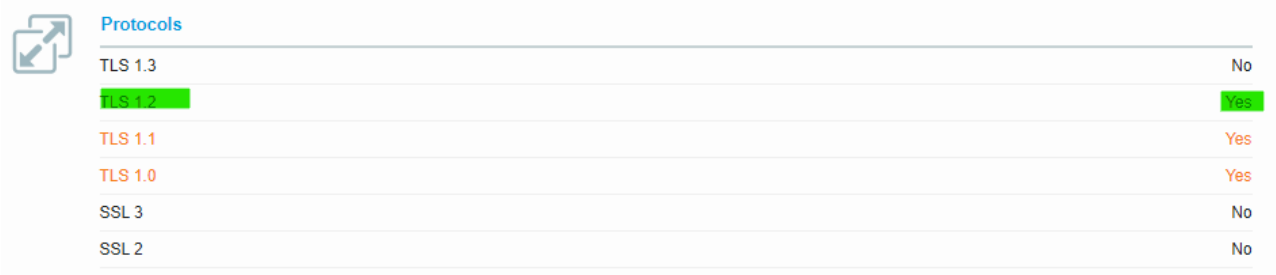
Como no dispongo de una máquina Ubuntu que me permita instalar sslscan, ya que a la hora de intentarlo instalar todo el rato obtengo el mismo error use el comando que use, como se puede ver en esta captura:

```
carlos@Ubuntu20:~$ sudo apt-get install sslscan
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
E: No se ha podido localizar el paquete sslscan
```

Voy a utilizar la herramienta online proporcionada por el profesor, (https://www.ssllabs.com/ssltest/) donde en la barra de búsqueda nos pide introducir el servidor web que queremos que escaneé, donde introducimos www.uva.es.

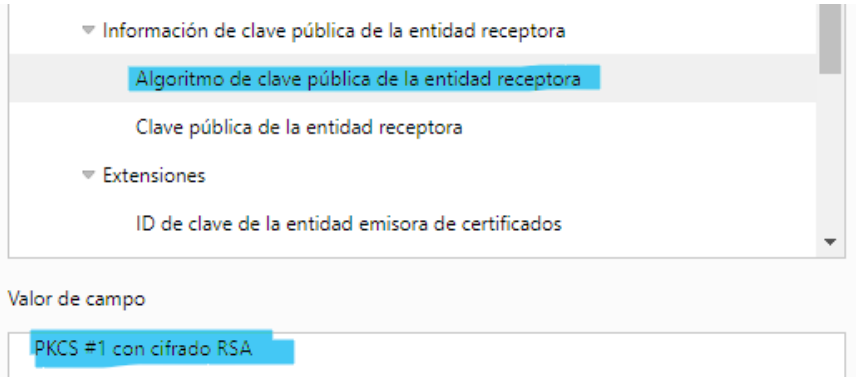
Aquí vemos que tras unos segundos nos ha generado un informe con toda la información.

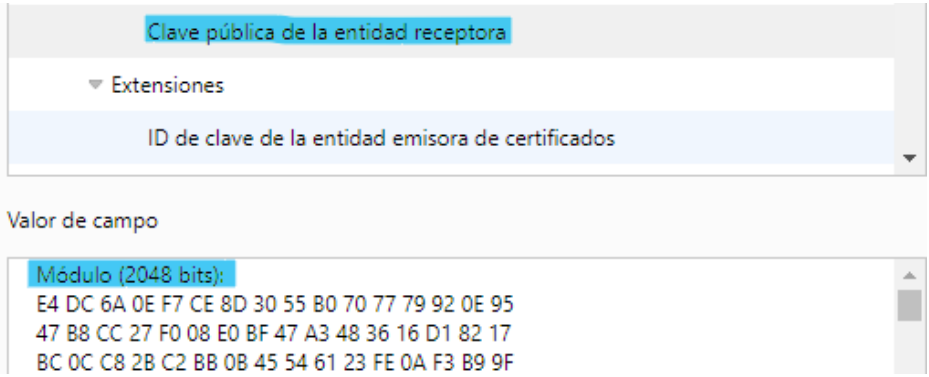
En el apartado de protocolos podemos ver que se esta usando el protocolo TLS (Transport Layer Security) en su versión 1.2.



b. Algoritmo de criptografía asimétrica (clave pública) utilizado y longitud de la clave pública.

Esta información podemos conocerla a través del apartado de detalles, del certificado digital que utiliza dicho servidor web. Como algoritmo de criptografía asimétrica utiliza el RSA y la longitud de la clave pública es de 2048 bits**.**





c. Indica la clave pública presente en el certificado digital. ¿Por qué se utiliza esa y no otra?

Para obtener la clave pública accedemos a la herramienta sslscan online y vemos:

Como podemos ver la clave pública es 65537, número primo (4º de Fermat, suponiendo que no tenemos en cuenta el 0), la representación de este número en binario es, 10000000000000001. Es un primo bastante grande por lo que se garantiza la seguridad, además su representación y los cálculos son bastante fáciles, puesto que este compuesto por 15 ceros y 2 unos.

d. Algoritmo de criptografía simétrica (clave privada) utilizado y longitud de la clave privada.

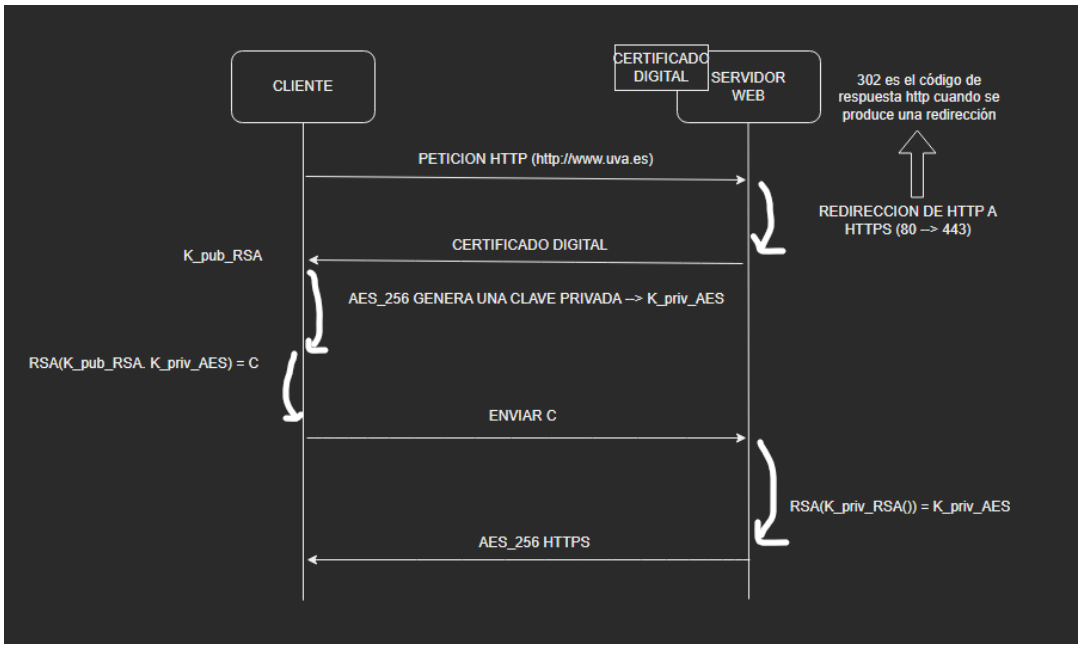
En el apartado Cipher Suites, sslscan muestra los algoritmos de criptografía de clave privada en orden de preferencia. Como se puede ver en la captura aparecen 4 algoritmos de los cuales los dos últimos se consideran débiles, por tanto, elegimos de los dos primeros el mejor. TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384.

Este algoritmo es el ECDH (Elliptic-Curve Diffie-Hellman), y en este caso de 256, es decir, la clave privada es de 256. Dicha clave privada ha de tenerla el servidor, ya que como alguien descubra o consiga esa clave va a poder conocer todas las comunicaciones.

e. Algoritmo de firma digital utilizado en el certificado digital.

Observando el certificado digital en el apartado “Algoritmo de firma de certificado” podemos ver que el que se utiliza en el certificado digital es SHA-384 con RSA.

3. Explica con un diagrama de secuencia por qué en un certificado digital se usa criptografía de clave pública y privada.



En el diagrama anterior podemos ver los pasos o fases que tienen lugar cuando intentamos acceder a un servidor web utilizando el puerto 80 TCP es decir, una conexión HTTP en vez de una conexión HTTPS con el puerto 443.

Como podemos ver primero realiza la petición HTTP, y este automáticamente realiza un redireccionamiento de puertos, como comento en el diagrama. Después el servidor devuelve la solicitud del certificado digital, dentro de este se encuentra la clave pública, K_pub_RSA. Ahora el cliente crea su clave privada con el algoritmo AES obteniendo K_priv_AES. Ahora RSA con esa clave privada, la pública cifra el mensaje a enviar M, obteniendo C (criptograma) y este criptograma se envía al servidor. Cuando el mensaje cifrado (criptograma) llega al servidor este puede conocer la clave privada del cliente y acceder al contenido del mensaje. Con esto habría finalizado la conexión que el cliente ha iniciado como HTTP con el servidor.

El certificado digital se utiliza en criptografía de clave pública y privada para evitar que otros "usuarios" no deseados interfieran en la comunicación cliente servidor.

4. Utiliza la herramienta OpenSSL para la generación de un certificado digital que tenga una longitud de clave de 1024 bits y analízalo con sslscan y ssltest.

Primero voy a crear un directorio donde voy a crear tanto el certificado como las claves. Una vez creado el directorio generamos la clave privada de RSA. La longitud de dicha clave es 1024 bits, ya que lo indica el enunciado, pero en la realidad se utilizan de 2048 o 4096.

Como estamos generando una clave la extensión de esta ha de ser .key

```
carlos@Ubuntu20:~$ sudo mkdir /opt/rsa
carlos@Ubuntu20:~$ cd /opt/rsa/
carlos@Ubuntu20:/opt/rsa$ sudo openssl genrsa -out alice.key 1024
Generating RSA private key, 1024 bit long modulus (2 primes)
...+++++
.......+++++
e is 65537 (0x010001)
```

Utilizando OpenSSL podemos ver información sobre la clave "alice.key", entre toda la información se puede ver por ejemplo el valor de n:

```
carlos@Ubuntu20:/opt/rsa$ sudo openssl rsa -text -in alice.key
RSA Private-Key: (1024 bit, 2 primes)
modulus:
    00:bd:d6:59:99:eb:1c:52:58:9b:f0:46:83:83:57:
    a0:60:1a:25:f8:17:0e:89:6a:14:8b:69:e4:ae:9c:
    5f:92:b9:8f:b2:26:69:e9:d2:f6:d3:73:61:84:65:
    62:d5:10:a3:7f:71:18:ce:ac:a7:b6:e9:dd:9f:d5:
    4b:69:0f:d6:80:24:56:5d:9e:31:24:5b:90:d2:9b:
    de:c5:2c:dc:fa:6f:c8:66:c8:ff:7e:c5:b6:a1:b3:
    7c:1a:ab:91:e6:44:5f:c3:8f:06:4e:a0:9f:13:ca:
    56:cd:4b:43:e2:6a:bf:14:7e:bc:29:34:ea:55:b4:
    d0:14:01:60:7b:e3:49:6f:b5
publicExponent: 65537 (0x10001)
privateExponent:
```

Y podemos ver la clave privada, mediante el comando "*sudo cat alice.key",* ya que sin permisos de super usuario nos deniega el permiso:

```
carlos@Ubuntu20:/opt/rsa$ cat alice.key
cat: alice.key: Permiso denegado
carlos@Ubuntu20:/opt/rsa$ sudo cat alice.key
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQC91lmZ6xxSWJvwRoODV6BgGiX4Fw6JahSLaeSunF+SuY+yJmnp
0vbTc2GEZWLVEKN/cRjOrKe26d2f1UtpD9aAJFZdnjEkW5DSm97FLNz6b8hmyP9+
xbahs3waq5HmRF/DjwZOoJ8TylbNS0Piar8UfrwpNOpVtNAUAWB740lvtQIDAQAB
AoGACUxYcyuK5ZWlfa1Wf7ZpCov9VBt9AMoLW5DCPLM4Ey23lhg/k85bfJvoKN32
a003VOSrPGREircFvfBqkyjNbr+gm1WF0OMTLMl41MRURt5hyzgRsimfly/Fk58n
iijy2wzAt4lFROFcyWCMFMoqD2FmuLUtz2M/GB/VCFvBXnkCQQD1ZbIKfEso66pZ
Edp3GWFeEqKZyk/0zb0T3QkwlEMz5sGIYXQKyyeh2r31o7vc1PBh4k5HU6Jx2CXf
yoF4npW3AkEAxgodIbsJS9dpSTYCYhocmvzvlGsTb6IrVWRril6rKPR2VXTSIfcp
CZFfxsFhOzmb8PrdAnEsFFsH5YVf9NFF8wJAbpdRIHaZbKygTZnFKc1vWvTtAH/z
frDFmFER1inLC3XgHDVHFaIujzvERhJvFHuGhaZ2fqCLYeD2WJ+rr49QiwJBAKeq
FmHhVQyFsljORhRCYSeCr19rjAf+N0vTegyq+6Pv3GObRPCGrZKxJnJCdIxMyM6r
xQN1R0lwjG3wjn3Z06sCQDyD0mneLzz/6xwMegt/yCuLMRTBwx820moX2aZsrC1I
P2VKn1zSt34hgNTtiqu6XrUbS7Y2lhOAxAHGNFpmbVQ=
-----END RSA PRIVATE KEY-----
```

Y ahora procedemos a crear la solicitud de creación de certificado digital, que tiene como clave la privada que acabamos de crear para ello:

```
carlos@Ubuntu20:/opt/rsa$ sudo openssl req -new -key alice.key -out alice.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Valladolid
Locality Name (eg, city) []:Valladolid
Organization Name (eg, company) [Internet Widgits Pty Ltd]:4ck.com
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.4ack.com
Email Address []:admin@4ck.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Debemos de ir rellenando los campos que nos van solicitando, aunque tenemos la posibilidad de dejar alguno de ellos en blanco, no tenemos por qué completar todo. Como podemos ver ya tenemos tanto la clave como la solicitud de certificado

```
carlos@Ubuntu20:/opt/rsa$ ls
alice.csr  alice.key
```

Antes de continuar, dos cosas importantes sobre la solicitud que acabamos de crear. La primera es que nunca debemos de poner el correo en texto claro cuando nos lo solicita, en este caso lo hemos hecho porque es un ejemplo académico pero en la realidad nunca deberíamos de hacerlo.

Y la segunda es la extensión de la solicitud ¿Por qué se pone csr?, CSR contiene un bloque encriptado de texto que identifica el solicitante del certificado e incluye datos

encriptados para cada uno de los datos que nos solicita. Además, este CSR está utilizado por una Potestad de Certificado para establecer prueba de identidad para sitios Web.

Ahora podemos ver el contenido de dicha solicitud con OpenSSL:

```
carlos@Ubuntu20:/opt/rsa$ sudo openssl req -noout -text -in alice.csr
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www.4ack.com, emailAddress
 = admin@4ck.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (1024 bit)
                Modulus:
                    00:bd:d6:59:99:eb:1c:52:58:9b:f0:46:83:83:57:
                    a0:60:1a:25:f8:17:0e:89:6a:14:8b:69:e4:ae:9c:
                    5f:92:b9:8f:b2:26:69:e9:d2:f6:d3:73:61:84:65:
                    62:d5:10:a3:7f:71:18:ce:ac:a7:b6:e9:dd:9f:d5:
                    4b:69:0f:d6:80:24:56:5d:9e:31:24:5b:90:d2:9b:
                    de:c5:2c:dc:fa:6f:c8:66:c8:ff:7e:c5:b6:a1:b3:
                    7c:1a:ab:91:e6:44:5f:c3:8f:06:4e:a0:9f:13:ca:
                    56:cd:4b:43:e2:6a:bf:14:7e:bc:29:34:ea:55:b4:
                    d0:14:01:60:7b:e3:49:6f:b5
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha256WithRSAEncryption
         9d:43:c7:ef:a0:c8:37:f7:a8:98:6c:1f:da:d8:08:af:0a:67:
         95:ee:60:01:f1:00:f6:d9:d9:b9:bd:30:95:c2:00:9b:8a:94:
         ff:30:88:d0:2d:9d:6a:07:d8:0c:6d:46:53:c6:59:66:cd:da:
         29:2f:c5:fc:5e:59:a0:97:1e:db:55:8b:5e:9b:9e:63:26:6d:
         2c:34:5b:18:8c:f1:47:ef:f8:87:44:7c:21:43:ff:a3:f5:84:
         db:41:6d:47:b5:15:e9:b6:af:9c:e0:e1:64:6e:c5:ce:d6:c1:
         37:a4:2d:e1:fd:2d:c0:87:37:1b:49:1f:dc:40:da:c2:ac:be:
         5d:60
```

Ya tenemos tanto la clave “alice.key” como la solicitud de certificado “alice.csr”, con ello podemos crear un certificado digital autofirmado compatible con el estándar X.509. Para ello ejecutamos el siguiente comando:

```
carlos@Ubuntu20:/opt/rsa$ sudo openssl x509 -req -days 365 -in alice.csr -signk
ey alice.key -out alice.crt
Signature ok
subject=C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www.4ack.com
, emailAddress = admin@4ck.com
Getting Private key
```

Ahora el certificado utiliza la extensión .crt, y lo hemos creado con el nombre “alice.crt”.

Para que no todo el mundo pueda acceder a la clave y certificado, vamos a mover (en nuestro caso copiar) los archivos creados a sus correspondientes directorios, ya que estos como luego veremos tienen los permisos adecuados.

```
carlos@Ubuntu20:/opt/rsa$ sudo cp alice.crt /etc/ssl/certs
carlos@Ubuntu20:/opt/rsa$ sudo cp alice.key /etc/ssl/private
```

Con el comando *“sudo ls –l /etc/ssl/”* podemos ver los permisos de los directorios que hay dentro del directorio /etc/ssl, y como vemos, /private únicamente tiene permisos de lectura el root. De esta forma, no puede acceder cualquiera a la clave privada, solo el dueño de la máquina o quién pueda acceder a los permisos de super usuario de dicha máquina.

```
carlos@Ubuntu20:/opt/rsa$ sudo ls -l /etc/ssl/
total 32
drwxr-xr-x 2 root root     16384 nov 10 13:23 certs
-rw-r--r-- 1 root root     10909 abr 20  2020 openssl.cnf
drwx--x--- 2 root ssl-cert  4096 nov 10 13:23 private
```

Ahora ya estaría creado, pero para poderlo analizar con sslscan o con ssltest he tenido problemas puesto que sería necesario crear un servidor web. Por tanto, la forma de poder analizarlo con estas dos herramientas es creando un servidor web e introduciendo el certificado ahí. Esto es debido a que sslscan y ssltest solo permite analizar servidores web, y con ello certificados presentes dentro de ellos, no únicamente certificados aislados.

Aunque con las herramientas que nos pide el enunciado no se puede analizar el certificado OpenSSL si que lo permite, por tanto con el comando "openssl x509 -in alice.crt -text -noout" que he obtenido de la página web (https://www.ibm.com/support/pages/openssl-commands-check-and-verify-your-ssl-certificate-key-and-csr) podemos analizar el certificado y la salida que obtenemos es la que se ve en la siguiente captura.

```
carlos@Ubuntu20:/opt/rsa$ openssl x509 -in alice.crt -text -noout
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            30:31:ec:e3:17:94:4c:cd:85:83:81:04:3e:3b:af:7d:fe:66:ff:c3
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www.
4ack.com, emailAddress = admin@4ck.com
        Validity
            Not Before: Nov 10 12:22:02 2022 GMT
            Not After : Nov 10 12:22:02 2023 GMT
        Subject: C = ES, ST = Valladolid, L = Valladolid, O = 4ck.com, CN = www
.4ack.com, emailAddress = admin@4ck.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (1024 bit)
                Modulus:
                    00:bd:d6:59:99:eb:1c:52:58:9b:f0:46:83:83:57:
                    a0:60:1a:25:f8:17:0e:89:6a:14:8b:69:e4:ae:9c:
                    5f:92:b9:8f:b2:26:69:e9:d2:f6:d3:73:61:84:65:
                    62:d5:10:a3:7f:71:18:ce:ac:a7:b6:e9:dd:9f:d5:
                    4b:69:0f:d6:80:24:56:5d:9e:31:24:5b:90:d2:9b:
                    de:c5:2c:dc:fa:6f:c8:66:c8:ff:7e:c5:b6:a1:b3:
                    7c:1a:ab:91:e6:44:5f:c3:8f:06:4e:a0:9f:13:ca:
                    56:cd:4b:43:e2:6a:bf:14:7e:bc:29:34:ea:55:b4:
                    d0:14:01:60:7b:e3:49:6f:b5
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha256WithRSAEncryption
```