



**UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA**

PRÁCTICA DE LABORATORIO N°2
Secret-Key Encryption Lab

Juan Manuel Porrero Almansa

Asignatura: Seguridad en Sistemas Informáticos

Titulación: Grado en Ingeniería Informática

Fecha: 30 de octubre de 2021

Introducción y Conceptos

El objetivo de esta práctica es familiarizarse con los conceptos de encriptación. De este laboratorio se espera que los alumnos cojan experiencia en temas de algoritmos de encriptación, modelos de encriptación, paddings y vectores de inicialización.

Muchos fallos cometidos por programadores a la hora de desarrollar software son errores de encriptación, que suelen desembocar en vulnerabilidades. Trataremos de mostrar algunas de esas debilidades.

Las herramientas que utilizaremos serán un sistema operativo de tipo Linux, en nuestro caso, una distribución Ubuntu y las herramientas de SSL. **SSL (Secure Socket Layer)** es un protocolo criptográfico que proporciona comunicaciones seguras por Internet y es de software libre.

A continuación, se adjuntan datos que se deberían tener en cuenta para la realización de las diferentes tareas:

Algoritmo	Tamaño de clave (bits)	Tamaño de bloque (bits)
DES	56	64
Triple DES	112 o 168	64
Triple DES	128, 192 o 256	128
IDEA	128	64
Blowfish	Variable hasta 448	64

Algoritmos de cifrado simétrico.

Modo de operación	Tipo de cifrador	Descripción	¿Vector de inicialización?
ECB (Electronic Codebook)	Bloques	Cada bloque de texto plano es cifrado de manera independiente utilizando la misma clave	NO
CBC (Cipher Block Chaining)	Bloques	La entrada al algoritmo de cifrado es el XOR del bloque de texto plano correspondiente y el bloque de texto cifrado anterior. En caso de que se produzca un error en un bloque cifrado, entonces pierde sentido el bloque descifrado a partir de ese error.	Si
CFB (Cipher Feedback)	Flujo	Un error se propaga en los siguientes sub-bloques de texto cifrado.	Si
OFB (Output Feedback)	Flujo	Una ventaja del modo OFB es que no transmiten los errores de transmisión en un bit.	Si

Modos de operación.

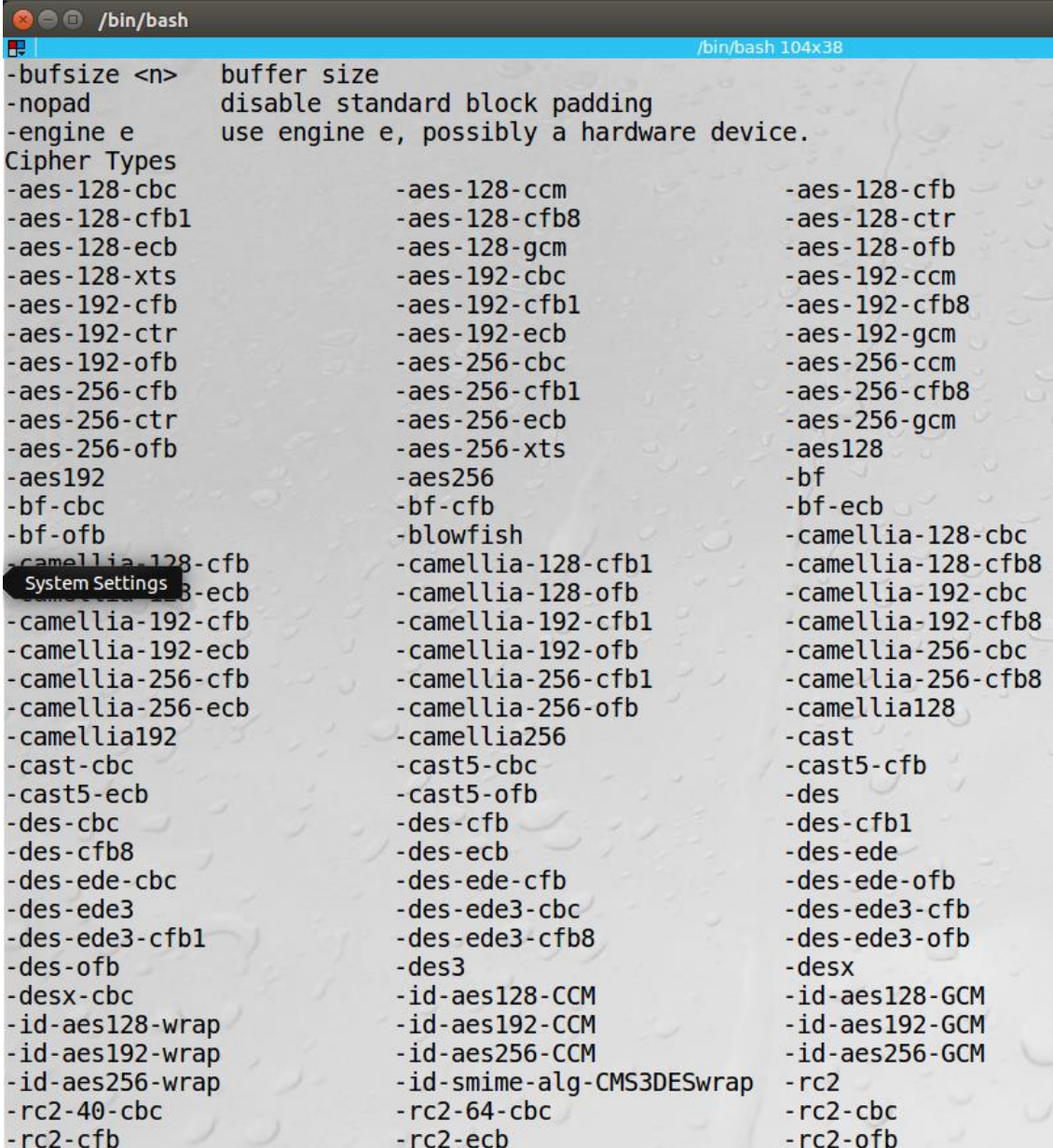
Tarea nº1: Encryption using different Ciphers and Modes

En la tarea 1 vamos a utilizar diferentes algoritmos de cifrado, se nos pide que mínimo sean tres. Se hará uso del comando **openssl enc** para cifrar o descifrar un archivo.

Para que nos sea más fácil recordarlo, haremos un esquema de cómo funciona el comando.

```
openssl enc -ciphername [-e|-d] [-in filename] [-out filename] [-K key] [-iv IV]
```

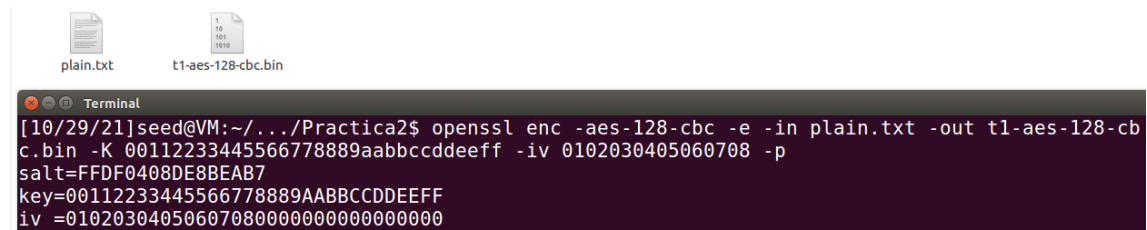
Utilizando el commando **openssl enc --help** podemos ver todos los cifrados:



```
/bin/bash
-bufsize <n>      buffer size
-nopad            disable standard block padding
-engine e         use engine e, possibly a hardware device.
Cipher Types
-aes-128-cbc      -aes-128-ccm      -aes-128-cfb
-aes-128-cfb1     -aes-128-cfb8      -aes-128-ctr
-aes-128-ecb      -aes-128-gcm       -aes-128-ofb
-aes-128-xts      -aes-192-cbc       -aes-192-ccm
-aes-192-cfb      -aes-192-cfb1      -aes-192-cfb8
-aes-192-ctr      -aes-192-ecb       -aes-192-gcm
-aes-192-ofb      -aes-256-cbc       -aes-256-ccm
-aes-256-cfb      -aes-256-cfb1      -aes-256-cfb8
-aes-256-ctr      -aes-256-ecb       -aes-256-gcm
-aes-256-ofb      -aes-256-xts       -aes128
-aes192           -aes256            -bf
-bf-cbc           -bf-cfb            -bf-ecb
-bf-ofb           -blowfish          -camellia-128-cbc
-camellia-128-cfb -camellia-128-cfb1 -camellia-128-cfb8
-camellia-128-ecb -camellia-128-ofb  -camellia-192-cbc
-camellia-192-cfb -camellia-192-cfb1 -camellia-192-cfb8
-camellia-192-ecb -camellia-192-ofb  -camellia-256-cbc
-camellia-256-cfb -camellia-256-cfb1 -camellia-256-cfb8
-camellia-256-ecb -camellia-256-ofb  -camellia128
-camellia192      -camellia256       -cast
-cast-cbc         -cast5-cbc         -cast5-cfb
-cast5-ecb        -cast5-ofb         -des
-des-cbc          -des-cfb           -des-cfb1
-des-cfb8         -des-ecb           -des-edede
-des-edede-cbc    -des-edede-cfb     -des-edede-ofb
-des-edede3       -des-edede3-cbc    -des-edede3-cfb
-des-edede3-cfb1  -des-edede3-cfb8   -des-edede3-ofb
-des-ofb          -des3              -desx
-desx-cbc         -id-aes128-CCM     -id-aes128-GCM
-id-aes128-wrap   -id-aes192-CCM     -id-aes192-GCM
-id-aes192-wrap   -id-aes256-CCM     -id-aes256-GCM
-id-aes256-wrap   -id-smime-alg-CMS3DESwrap -rc2
-rc2-40-cbc       -rc2-64-cbc        -rc2-cbc
-rc2-cfb          -rc2-ecb           -rc2-ofb
```

Ahora, vamos a proceder a utilizar 3 cifrados diferentes:

AES-128-CBC



```
plain.txt  t1-aes-128-cbc.bin

[10/29/21]seed@VM:~/.../Practica2$ openssl enc -aes-128-cbc -e -in plain.txt -out t1-aes-128-cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
```

Como podemos ver, al cifrarlo, se nos crea el archivo cifrado. Ahora procederemos a descifrarlo con el siguiente comando:

```
[10/29/21]seed@VM:~/.../Practica2$ openssl enc -aes-128-cbc -d -in t1-aes-128-cbc.bin -out plain_descifrado.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
```

Lo que hemos hecho ha sido cambiar la opción `-e` por `-d` (decode) tomando como entrada el archivo cifrado y por salida generando uno nuevo.

AES-256-CBC

```
[10/29/21]seed@VM:~/.../Practica2$ openssl enc -aes-256-cbc -e -in plain.txt -out t1-aes-256-cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF00000000000000000000000000000000
iv =01020304050607080000000000000000
```

AES-128-CFB

```
[10/29/21]seed@VM:~/.../Practica2$ openssl enc -aes-128-cfb -e -in plain.txt -out t1-aes-128-cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
```

BF-CBC

```
[10/29/21]seed@VM:~/.../Practica2$ openssl enc -bf-cbc -e -in plain.txt -out t1-bf-cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =0102030405060708
```

DES-CFB

```
[10/29/21]seed@VM:~/.../Practica2$ openssl enc -des-cfb -e -in plain.txt -out t1-des-cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
hex string is too long
invalid hex key value
```

Este es un caso curioso, ya que DES en la implementación de SSL utiliza una clave de 64 bits, por lo que el comando sería:

```
[10/29/21]seed@VM:~/.../Practica2$ openssl enc -des-cfb -in plain.txt -out t1-des-cfb.bin -K 00112233445566 -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=0011223344556600
iv =0102030405060708
```

Tarea nº2: Encryption Mode – ECB vs. CBC

El objetivo de esta tarea, en primer lugar, es cifrar una imagen con extensión *.bmp* para que las personas no autorizadas (aquellas que no dispongan de la clave) no puedan ver el contenido de esta imagen.

Se llevará a cabo un cifrado con los tipos ECB y CBC.

La imagen utilizada será la siguiente:

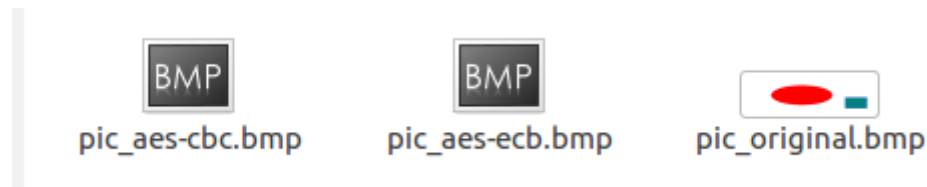


Los comandos para encriptar serán los siguientes:

- **Encriptado con aes-128-cbc**

```
openssl enc -aes-128-cbc -in pic_original.bmp -out pic_aes-cbc.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
```
- **Encriptado con aes-128-ecb**

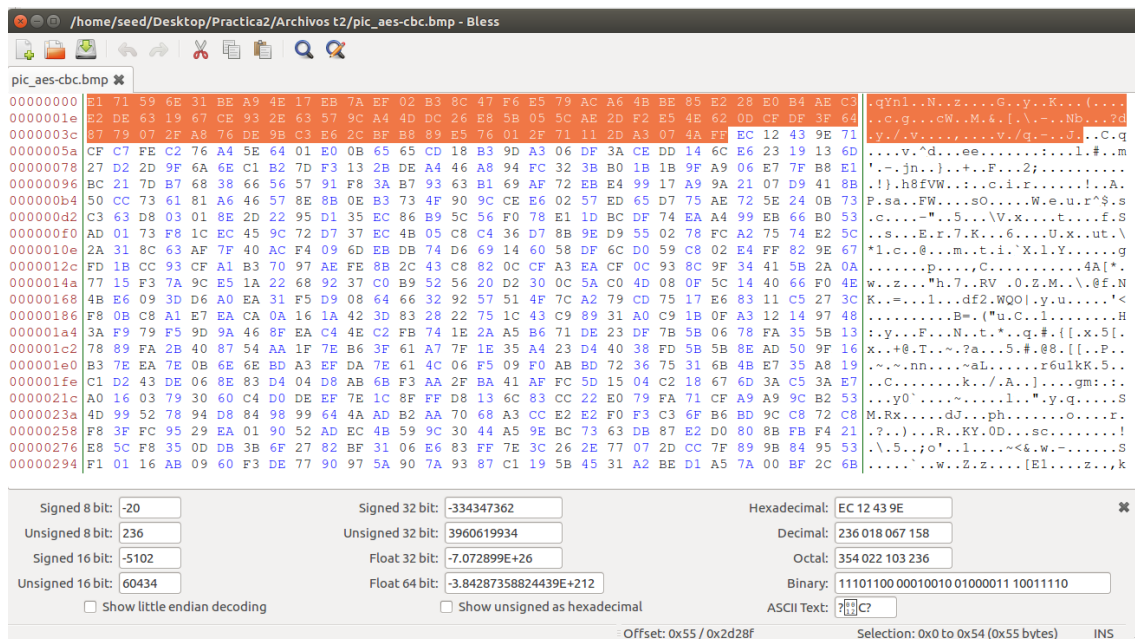
```
openssl enc -aes-128-ecb -in pic_original.bmp -out pic_aes-ecb.bmp -K 00112233445566778889aabbccddeeff -p
```



```
[10/29/21]seed@VM:~/.../Archivos t2$ openssl enc -aes-128-ecb -in pic_original.bmp -out pic_aes-ecb.bmp -K 00112233445566778889aabbccddeeff -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
[10/29/21]seed@VM:~/.../Archivos t2$ openssl enc -aes-128-cbc -in pic_original.bmp -out pic_aes-cbc.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
```

Como podemos observar, una vez cifrada la imagen de las dos formas, se nos crean los archivos correspondientes.

Se nos dice ahora, que, para los archivos *.bpm*, los 54 primeros bytes de la cabecera tienen información sobre la imagen. Tendremos que sustituir esos 54 de las imágenes cifradas por los de la imagen original.



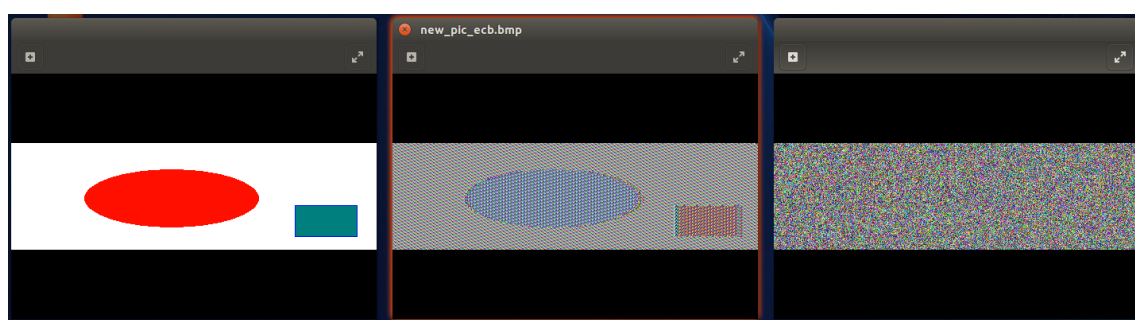
Estos son los 54 primeros bytes de una de las imágenes cifradas, que tendremos que sustituir. Hemos utilizado el editor hexadecimal Bless para abrirlos.

Podemos editar cada uno de los bytes desde ahí, aunque resulta más cómodo bajo mi gusto hacerlo con los comandos que se nos especifica en el enunciado de la práctica. Se trata de aislar los 54 bits de la original, para combinarlo con el *body* de la encriptada.

```

[10/29/21]seed@VM:~/.../Archivos t2$ head -c 54 pic_original.bmp > header
[10/29/21]seed@VM:~/.../Archivos t2$ tail -c +55 pic_aes-cbc.bmp > body_cbc
[10/29/21]seed@VM:~/.../Archivos t2$ cat header body_cbc > new_pic_cbc.bmp
[10/29/21]seed@VM:~/.../Archivos t2$
[10/29/21]seed@VM:~/.../Archivos t2$ tail -c +55 pic_aes-ecb.bmp > body_ecb
[10/29/21]seed@VM:~/.../Archivos t2$ cat header body_ecb > new_pic_ecb.bmp
[10/29/21]seed@VM:~/.../Archivos t2$

```



ORIGINAL

ECB

CBC

Podemos observar como en la codificada con ECB podemos ver y hacernos una idea de cómo era la imagen original. El **inconveniente de ECB** es que cuando se da el caso de que dos bloques de texto son iguales, estos se cifran con la misma clave y se da por salida dos bloques cifrados también idénticos.

En cuanto a **CBC**, es más seguro como podemos observar, ya que entre los bloques entrada y los bloques salida no encontramos semejanzas ni relación entre ellos. La razón de ello es que la entrada del algoritmo es el XOR entre el bloque de texto plano y el bloque de texto cifrado antes.

Ahora probaremos el proceso con una imagen personalizada:



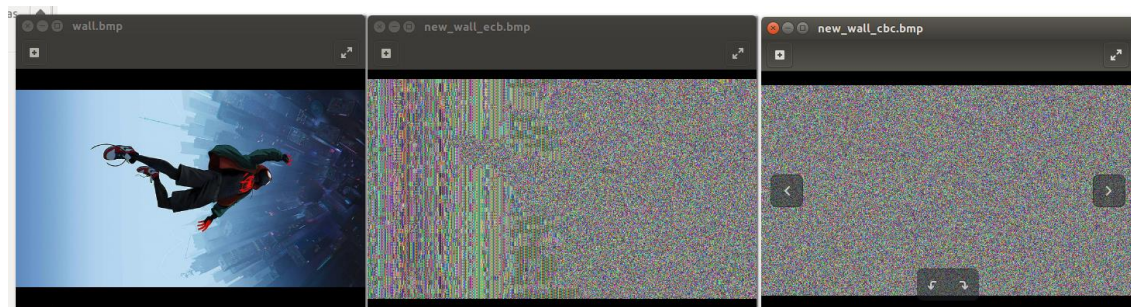
Ciframos como hemos hecho anteriormente con CBC y ECB

```
[10/29/21]seed@VM:~/.../Archivos t2$ openssl enc -aes-128-cbc -in wall.bmp -out wall_aes-cbc.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[10/29/21]seed@VM:~/.../Archivos t2$ openssl enc -aes-128-ecb -in wall.bmp -out wall_aes-ecb.bmp -K 00112233445566778889aabbccddeeff -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
```

Sustituimos los primero 54 bits:

```
[10/29/21]seed@VM:~/.../Archivos t2$ head -c 54 wall.bmp > wall_header
[10/29/21]seed@VM:~/.../Archivos t2$ tail -c +55 wall_aes-ecb.bmp > wall_body_ecb
[10/29/21]seed@VM:~/.../Archivos t2$ cat wall_header wall_body_ecb > new_wall_ecb.bmp
[10/29/21]seed@VM:~/.../Archivos t2$ tail -c +55 wall_aes-cbc.bmp > wall_body_cbc
[10/29/21]seed@VM:~/.../Archivos t2$ cat wall_header wall_body_cbc > new_wall_cbc.bmp
```

Los resultados obtenidos son:



Podemos observar distorsión en la parte de la izquierda con ECB y nada en la CBC.

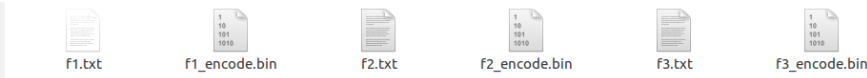
Tarea nº3: Padding

Cuando estamos usando un cifrador de bloque, tenemos que tener en cuenta la siguiente particularidad sobre ellos: cuando el tamaño del texto plano no es múltiplo del tamaño del bloque, utilizan un **esquema de relleno**. La finalidad de este esquema de relleno es que el texto plano sí que sea múltiplo del tamaño de bloque. Un ejemplo de relleno sería el de añadir ceros al final o al principio, repetir una letra, etc.

Según OpenSSL, el relleno más utilizado es el llamado **PKCS#5**. Para comprobar como funciona el relleno, en el guión de la práctica se nos pide que creamos tres archivos de 5 bytes, de 10 bytes y de 16 bytes, respectivamente. Utilizaremos el comando **echo -n** para ello (la opción -n es importante ya que si no tendríamos un byte más debido a que el carácter \n de nueva línea contaría como uno).

```
Terminal
[10/30/21]seed@VM:~/.../Archivos t3$ echo -n "12345" > f1.txt
[10/30/21]seed@VM:~/.../Archivos t3$ echo -n "1234567890" > f2.txt
[10/30/21]seed@VM:~/.../Archivos t3$ echo -n "1234567890123456" > f3.txt
```

Una vez creados, los cifraremos con **aes-128-cbc**:



```
Terminal
[10/30/21]seed@VM:~/.../Archivos t3$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_encode
.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[10/30/21]seed@VM:~/.../Archivos t3$ openssl enc -aes-128-cbc -e -in f2.txt -out f2_encode
.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[10/30/21]seed@VM:~/.../Archivos t3$ openssl enc -aes-128-cbc -e -in f3.txt -out f3_encode
.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
```

Ahora, lo que vamos a hacer, es ver si su tamaño ha cambiado una vez cifrados. Para ello haremos uso de un simple comando **ls -l**.

```
[10/30/21]seed@VM:~/.../Archivos t3$ ls -l
total 24
-rw-rw-r-- 1 seed seed 16 Oct 30 12:03 f1_encode.bin
-rw-rw-r-- 1 seed seed 5 Oct 30 11:44 f1.txt
-rw-rw-r-- 1 seed seed 16 Oct 30 12:04 f2_encode.bin
-rw-rw-r-- 1 seed seed 10 Oct 30 11:45 f2.txt
-rw-rw-r-- 1 seed seed 32 Oct 30 12:04 f3_encode.bin
-rw-rw-r-- 1 seed seed 16 Oct 30 11:45 f3.txt
```

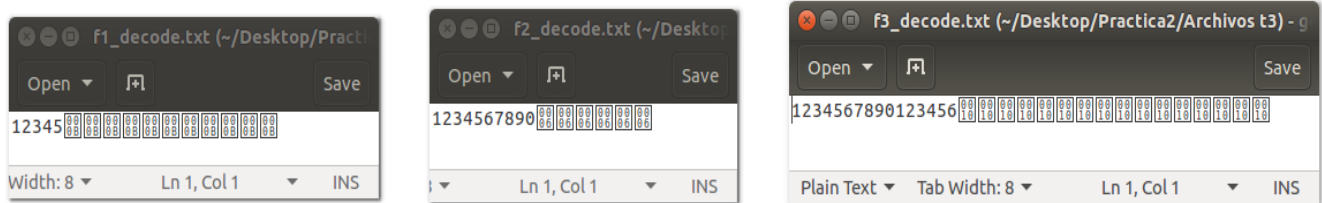
En esta captura vemos, como, efectivamente los archivos cifrados tienen más tamaño que los originales, por lo que se ha producido el **relleno o padding**.

Ahora, decodificaremos cada archivo para ver qué es lo que se ha añadido a cada uno. Cabe destacar que por defecto, cuando desciframos en OpenSSL, estos datos de relleno se eliminan, es por ello que usaremos la opción **-nopad** para que estos añadidos se conserven en el archivo decodificado.

```

[10/30/21]seed@VM:~/.../Archivos t3$ openssl enc -aes-128-cbc -d -in f1_encode.bin -out f1_decode.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p -nopad
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[10/30/21]seed@VM:~/.../Archivos t3$ openssl enc -aes-128-cbc -d -in f2_encode.bin -out f2_decode.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p -nopad
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[10/30/21]seed@VM:~/.../Archivos t3$ openssl enc -aes-128-cbc -d -in f3_encode.bin -out f3_decode.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p -nopad
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
  
```

Aquí podemos ver el contenido que se añadió a cada uno de los archivos:



Como no pueden ser imprimidos cuando los abrimos con un editor de texto como Gedit, utilizaremos los comandos **hexdump -C <archivo>** y **xxd <archivo>** para visualizarlo.

```

[10/30/21]seed@VM:~/.../Archivos t3$ hexdump -C f1_decode.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b  |12345.....|
00000010
[10/30/21]seed@VM:~/.../Archivos t3$ xxd f1_decode.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345.....

[10/30/21]seed@VM:~/.../Archivos t3$ hexdump -C f2_decode.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06  |1234567890.....|
00000010
[10/30/21]seed@VM:~/.../Archivos t3$ xxd f2_decode.txt
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890.....

[10/30/21]seed@VM:~/.../Archivos t3$ hexdump -C f3_decode.txt
00000000  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  |1234567890123456|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10  |.....|
00000020
[10/30/21]seed@VM:~/.../Archivos t3$ xxd f3_decode.txt
00000000: 3132 3334 3536 3738 3930 3132 3334 3536  1234567890123456
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  .....
  
```

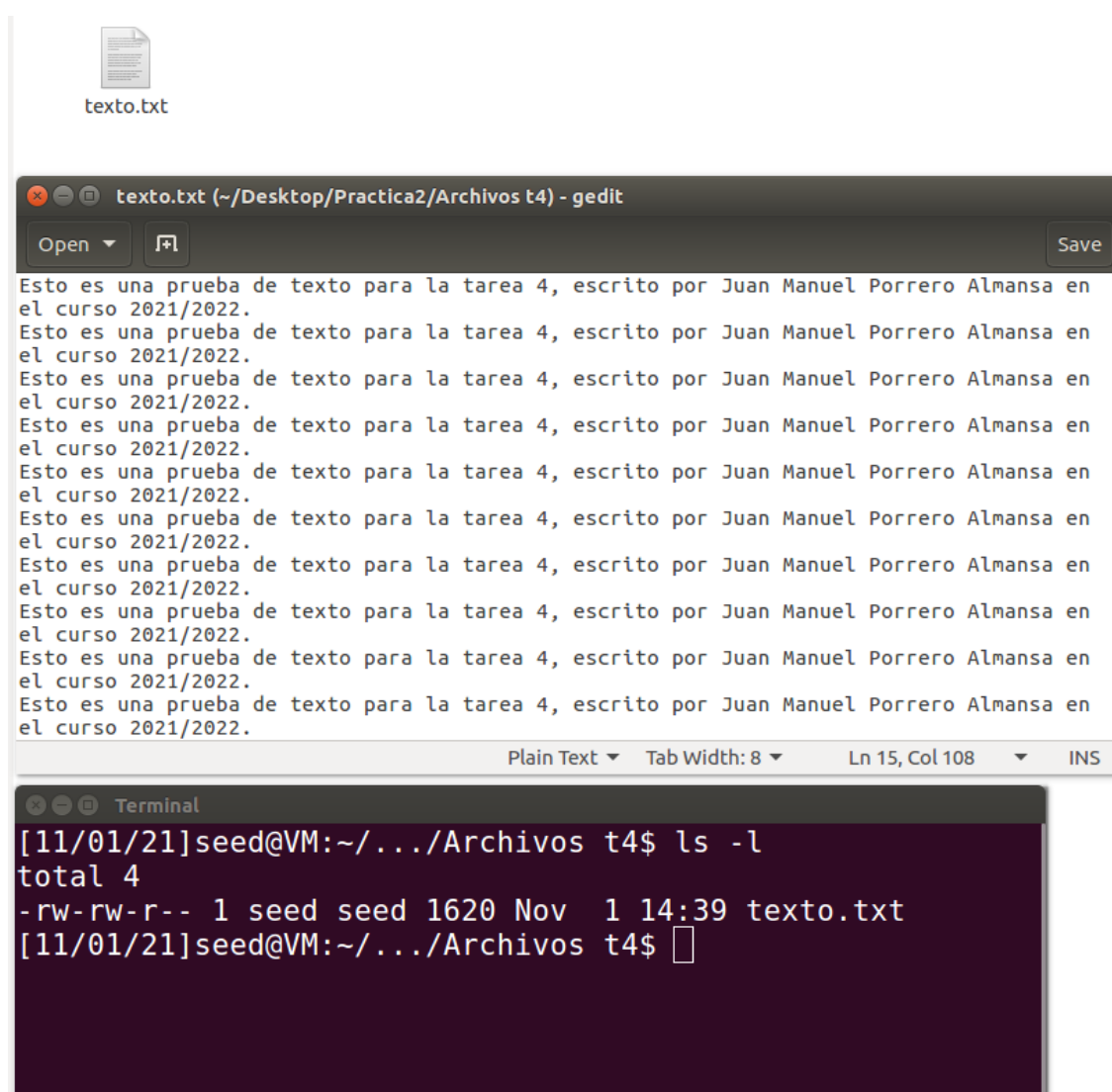
Como **conclusión**, podemos decir que el padding o esquema de relleno es utilizado y por lo tanto, necesario, en los tipos ECB y CBC debido que cifran por bloques. El tamaño de bloque depende del algoritmo, por ejemplo, AES utiliza uno de 16 bytes y 3DES y Blowfish uno de 8 bytes. CFB y OFB no necesitan relleno al ser cifradores de flujo.

Tarea nº4: Error Propagation – Corrupted Cipher Text

Esta tarea tiene como finalidad la propiedad de propagación de errores de los diferentes modos de encriptación. Buscando información de antemano, cabe anticipar que con los algoritmos CBC y ECB habrá más propagación de error ya que trabajan por bloques. El modo CFB, un bit erróneo provoca en el texto cifrado la aparición de $1+64/m$ bloques de texto plano incorrecto por lo que también habría propagación. En el modo OFB al intercambiar un bit en el texto cifrado, se genera texto cifrado con un bit erróneo, entonces solo notaríamos el error en ese bit.

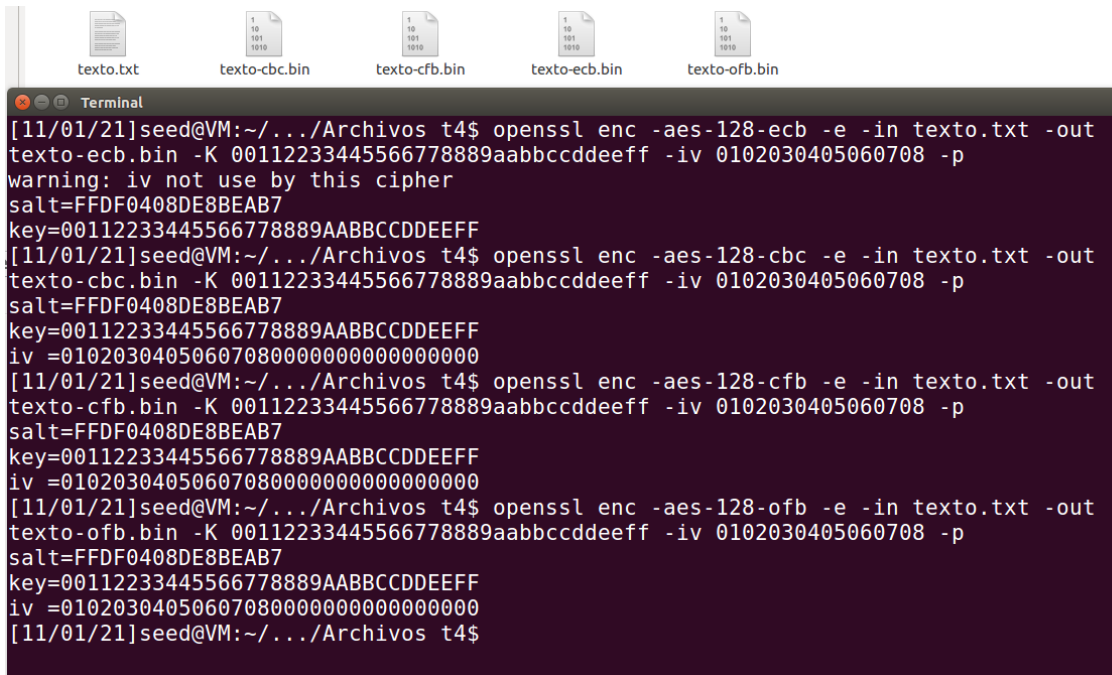
Se nos pide crear un archivo de texto plano con al menos 1000 bytes, encriptarlo usando ECB, CBC, CFB y OFB con AES-128 y cambiar el byte número 55 para después comprobar la propagación de errores.

Creamos el archivo:



Como podemos observar, el tamaño del archivo es de 1620 bytes, también se puede ver el contenido del mismo.

Lo encriptamos con los 4 diferentes modos:

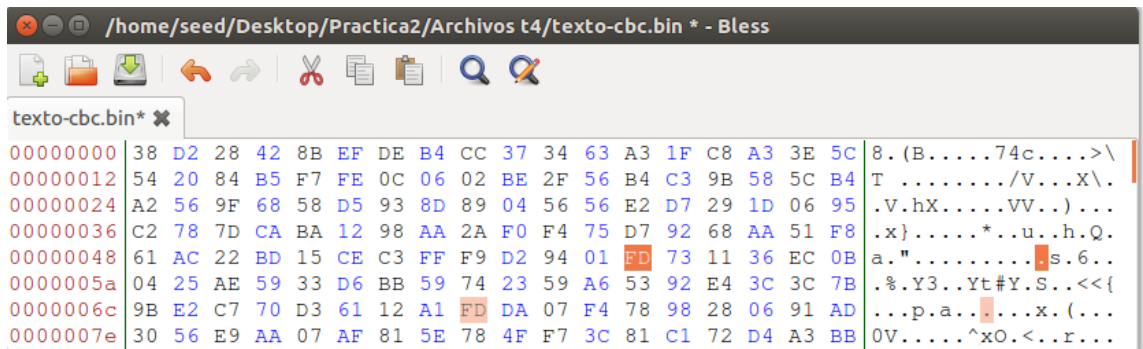


```
terminal
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-ecb -e -in texto.txt -out
texto-ecb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
warning: iv not use by this cipher
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-cbc -e -in texto.txt -out
texto-cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-cfb -e -in texto.txt -out
texto-cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-ofb -e -in texto.txt -out
texto-ofb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[11/01/21]seed@VM:~/.../Archivos t4$
```

NOTA: con ECB no hace falta utilizar vector de inicialización, como podemos observar en la tabla *modos de operación* al inicio de este documento. Es por ello que nos sale ese warning avisando de que no se utiliza, no obstante, el resultado es el mismo.

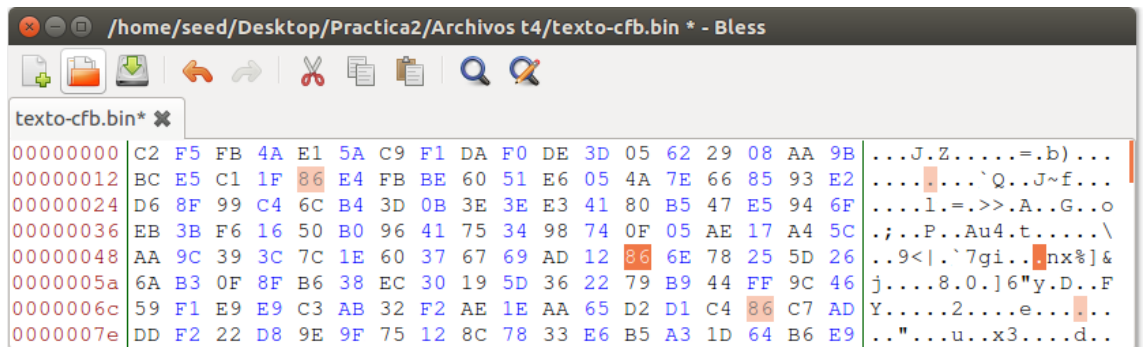
Ahora modificaremos el byte 55 de cada texto cifrado para ver qué ocurre:

- **CBC** FF -> FD



```
/home/seed/Desktop/Practica2/Archivos t4/texto-cbc.bin * - Bless
texto-cbc.bin* X
00000000 38 D2 28 42 8B EF DE B4 CC 37 34 63 A3 1F C8 A3 3E 5C 8. (B.....74c....>\
00000012 54 20 84 B5 F7 FE 0C 06 02 BE 2F 56 B4 C3 9B 58 5C B4 T ...../V...X\
00000024 A2 56 9F 68 58 D5 93 8D 89 04 56 56 E2 D7 29 1D 06 95 .V.hX.....VV..) ...
00000036 C2 78 7D CA BA 12 98 AA 2A F0 F4 75 D7 92 68 AA 51 F8 .x}.....*...u..h.Q.
00000048 61 AC 22 BD 15 CE C3 FF F9 D2 94 01 FD 73 11 36 EC 0B a.".....s.6..
0000005a 04 25 AE 59 33 D6 BB 59 74 23 59 A6 53 92 E4 3C 3C 7B .%.Y3..Yt#Y.S.<<{
0000006c 9B E2 C7 70 D3 61 12 A1 FD DA 07 F4 78 98 28 06 91 AD ...p.a...x.(...
0000007e 30 56 E9 AA 07 AF 81 5E 78 4F F7 3C 81 C1 72 D4 A3 BB 0V.....^xO.<..r...
```

- **CFB** 80 -> 86



```
/home/seed/Desktop/Practica2/Archivos t4/texto-cfb.bin * - Bless
texto-cfb.bin* X
00000000 C2 F5 FB 4A E1 5A C9 F1 DA F0 DE 3D 05 62 29 08 AA 9B ...J.Z.....=.b)...
00000012 BC E5 C1 1F 86 E4 FB BE 60 51 E6 05 4A 7E 66 85 93 E2 .....`Q...J~f...
00000024 D6 8F 99 C4 6C B4 3D 0B 3E E3 41 80 B5 47 E5 94 6F ....l.=.>>.A..G..o
00000036 EB 3B F6 16 50 B0 96 41 75 34 98 74 0F 05 AE 17 A4 5C ;;..P..Au4.t.....\
00000048 AA 9C 39 3C 7C 1E 60 37 67 69 AD 12 86 6E 78 25 5D 26 ..9<|. `7gi...nx%]&
0000005a 6A B3 0F 8F B6 38 EC 30 19 5D 36 22 79 B9 44 FF 9C 46 j....8.0.]6"y.D..F
0000006c 59 F1 E9 E9 C3 AB 32 F2 AE 1E AA 65 D2 D1 C4 86 C7 AD Y.....2.....e...
0000007e DD F2 22 D8 9E 9F 75 12 8C 78 33 E6 B5 A3 1D 64 B6 E9 .."....u..x3....d..
```

- ECB 26 -> 24

```

/home/seed/Desktop/Practica2/Archivos t4/texto-ecb.bin - Bless
texto-ecb.bin x
00000000 B2 1B F3 D9 58 2C BF DF C5 1F 3F 47 58 3F A3 08 EF EE ....X,...?GX?...
00000012 35 F9 33 F6 36 C5 8D 71 96 BB 05 FE 09 70 66 94 6C E1 5.3.6..q.....pf.l.
00000024 17 F9 BF 39 5A 9C A2 0A 94 44 19 3F 36 3E 5A 2B 61 FC ...9Z...D.?6>Z+a.
00000036 76 E8 BC F7 43 78 8E 8D D2 62 A8 EB 59 39 75 B9 BB FA v...Cx...b..Y9u...
00000048 CA 37 79 6B 14 44 73 5D 73 93 6F F5 24 99 05 9E D6 92 .7yk.Ds]s.o.$.....
0000005a 07 F3 0F B5 B7 C9 30 32 F0 C2 21 E0 F4 7D 41 CA 7E BD .....02...!..}A.~.
0000006c 7C 6C 41 CD C6 1D 17 11 7C 0D 56 4C 67 2A D3 54 49 8B |lA....|.VLg*.TI.
0000007e 3D 26 11 D4 7A 4A C1 DC D6 4A 88 DC 49 98 14 92 33 56 =&...zJ...J..I...3V

```

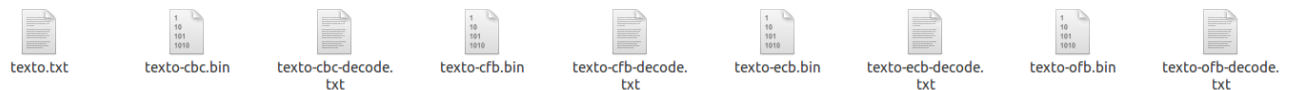
- OFB EA -> EE

```

/home/seed/Desktop/Practica2/Archivos t4/texto-ofb.bin - Bless
texto-ofb.bin x
00000000 C2 F5 FB 4A E1 5A C9 F1 DA F0 DE 3D 05 62 29 08 53 35 ...J.Z.....=.b).S5
00000012 B9 63 DE 7B 36 96 8F 2C 69 DC 23 08 3A FE F5 01 5D 41 .c.{6...i.#...}A
00000024 11 04 2D 30 74 86 5F A9 1D 58 DA CB 7C 4C C6 F5 45 E6 ..-0t._.X...|L..E.
00000036 8A 54 1B 3B 14 7B B1 5E 58 35 13 07 B8 BC F8 37 2A 8A .T;.{.^X5.....7*.
00000048 14 CB 35 8C 93 80 A7 26 7D 36 13 9E EE 09 1D 7E 4C 90 ..5....&}6...~L.
0000005a 97 05 1E 6E 46 D7 D5 6D F2 E3 D8 D3 51 2D 45 36 C5 A5 ...nF..m....Q-E6..
0000006c 96 E5 93 4F 39 EC 27 FA 27 DF 5E 9D 8E 3A 6E BF DB 5B ...O9.'.'.^...:n..[
0000007e 07 EE 63 11 D3 EB 97 AE 79 AC 71 E3 46 F7 7E ED 88 D0 .c.....y.q.F.~...

```

Ahora, los desciframos:



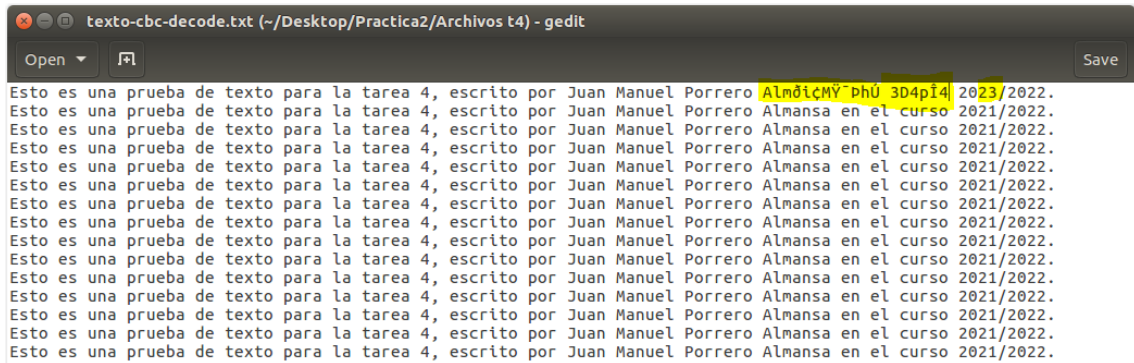
```

Terminal
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-cbc -d -in texto-cbc.bin -o
ut texto-cbc-decode.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-cfb -d -in texto-cfb.bin -o
ut texto-cfb-decode.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-ecb -d -in texto-ecb.bin -o
ut texto-ecb-decode.txt -K 00112233445566778889aabbccddeeff -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
[11/01/21]seed@VM:~/.../Archivos t4$ openssl enc -aes-128-ofb -d -in texto-ofb.bin -o
ut texto-ofb-decode.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=FFDF0408DE8BEAB7
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[11/01/21]seed@VM:~/.../Archivos t4$

```

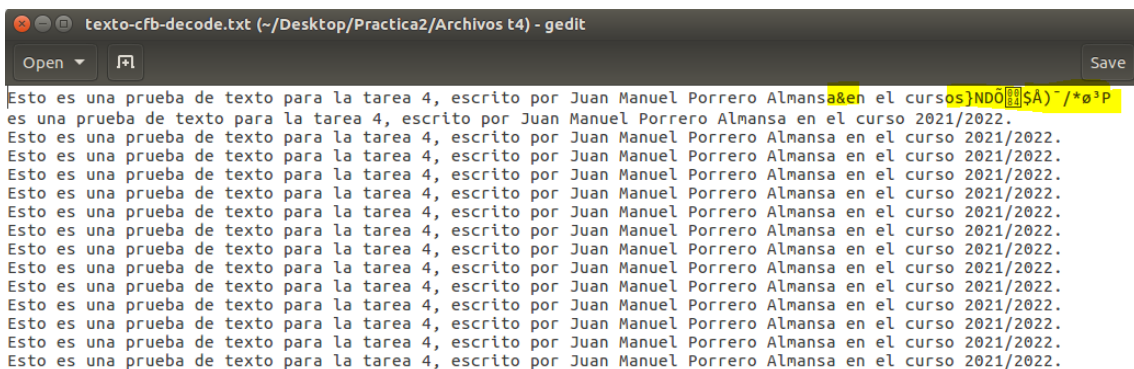

Ahora es turno de observar los ficheros modificados:

- CBC



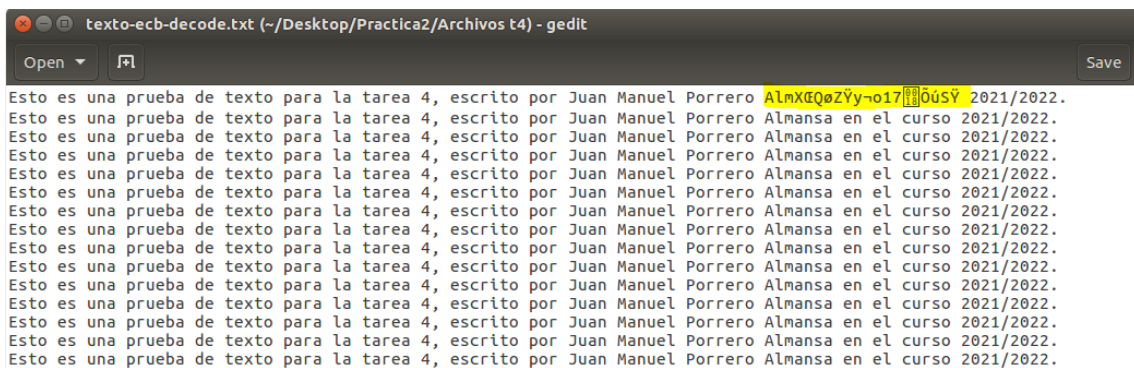
Como podemos comprobar en la parte resaltada, modificando un único byte se han visto afectados **varios bloques**, ya que, en este algoritmo, el texto cifrado depende mucho del bloque de texto plano de entrada. Observar cómo se escribe 2023 en vez de 2021.

- CFB



En el caso de CFB, la función de cifrado se usa para generar 1+64/m bloques de texto plano erróneos, entendiendo por m la longitud del flujo en la que se divide dicho bloque.

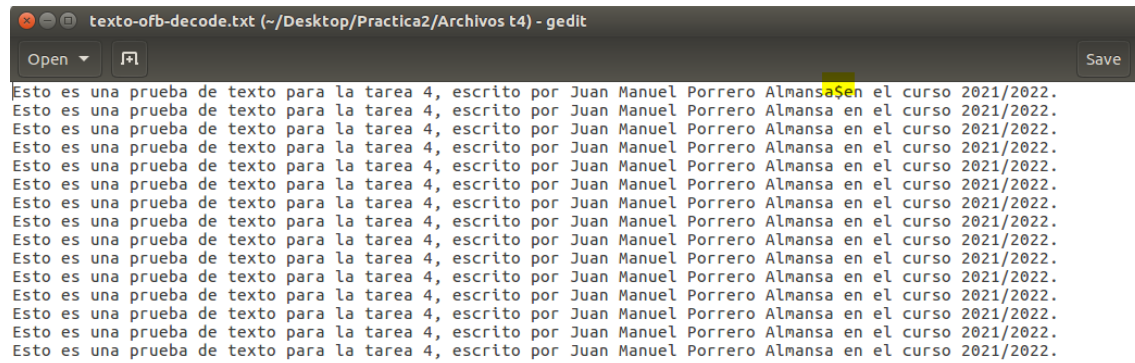
- ECB



Cuando usamos ECB, cada bloque se cifra de manera independiente, del mismo modo ocurre cuando se descifra, por lo que solo un bloque se ve afectado cuando hay errores.

Como hemos modificado un bit del byte 55 y el cifrado se hace de 8 en 8 bytes, ese error se extiende al resto del bloque.

- OFB



Como podemos observar, con OFB solo se ve afectado el bit que modificamos, como adelantábamos previamente.

Como **conclusión**, podemos decir que en lo referente a la propagación de errores, el modo OFB es el que mejor resultados nos ofrece.