Lab Overview

Secret-key encryption, also known as symmetric encryption, is a cryptographic technique that involves the use of a shared secret key to both encrypt and decrypt data. In this form of encryption, the same key is used for both the encryption and decryption processes, hence the name "secret-key."

The key used in secret-key encryption is a long string of bits, typically generated randomly. This key is kept secret and known only to the sender and intended recipient of the encrypted data. The security of secret-key encryption relies on the secrecy of the key itself.

The encryption process in secret-key encryption involves applying a mathematical algorithm, known as a cipher, to the plaintext (original unencrypted data) along with the secret key. The output of this process is the ciphertext, which is the encrypted form of the data. To decrypt the ciphertext and retrieve the original plaintext, the recipient applies the same secret key and algorithm in the reverse order.

One of the primary advantages of secret-key encryption is its efficiency. The encryption and decryption processes are typically faster compared to other encryption methods, such as public-key encryption. Additionally, secret-key encryption is suitable for encrypting large amounts of data since the computational overhead is relatively low.

However, a major challenge with secret-key encryption is securely distributing the secret key between the sender and the recipient. If an attacker gains access to the secret key, they can decrypt the ciphertext and access the original data. Therefore, establishing a secure key exchange mechanism is crucial for maintaining the confidentiality of the encrypted information.

To address the key distribution problem, key management protocols and algorithms have been developed. These protocols often involve secure key exchange techniques, such as Diffie-Hellman key exchange or key distribution centers. They ensure that the secret key is securely shared between the communicating parties without being intercepted by unauthorized individuals.

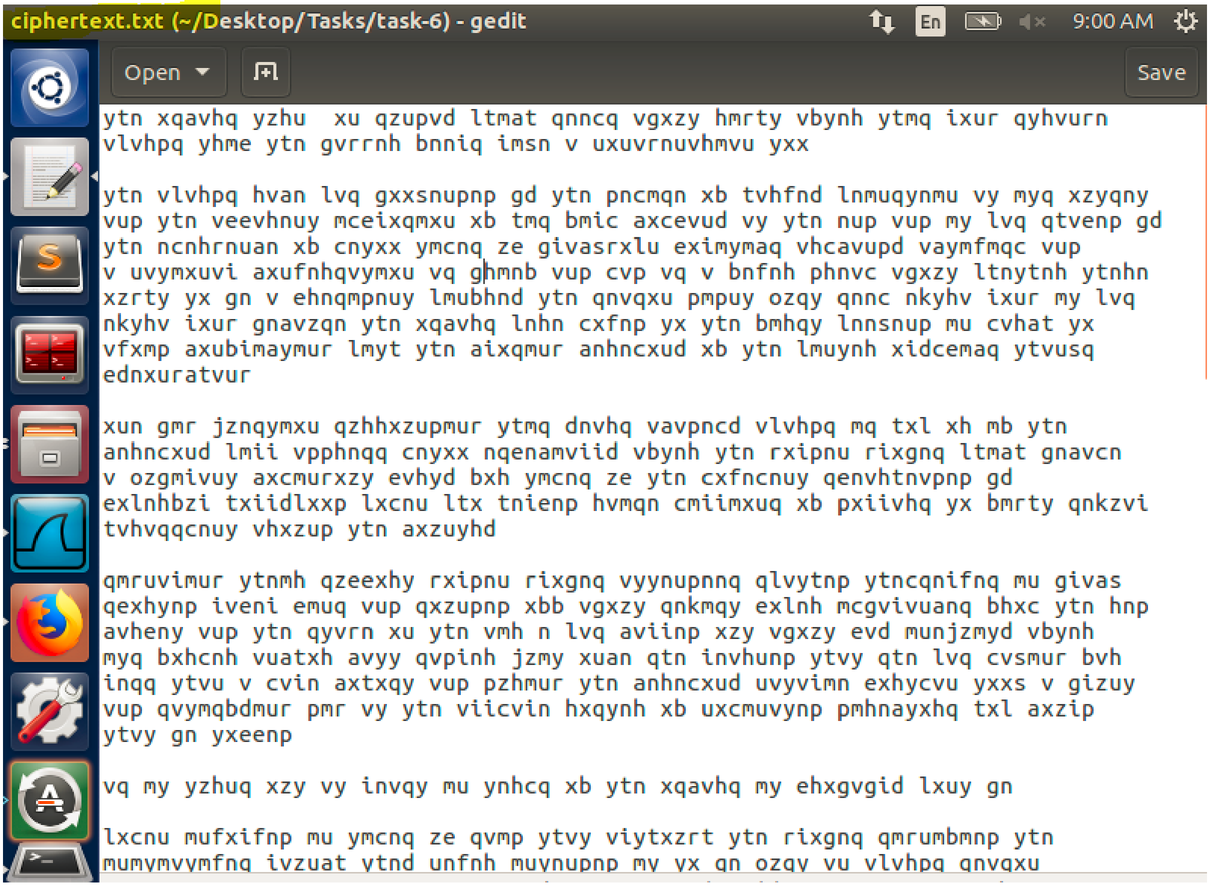
Some widely used secret-key encryption algorithms include the Data Encryption Standard (DES), Triple DES (3DES), and the Advanced Encryption Standard (AES). These algorithms provide varying levels of security and are implemented in various applications and systems where confidentiality is essential, such as secure communication channels, secure file storage, and virtual private networks (VPNs).

In summary, secret-key encryption is a cryptographic technique that uses a shared secret key to encrypt and decrypt data. It offers efficiency in processing large amounts of data, but key management and secure distribution are critical to maintaining the confidentiality of the encrypted information. Various algorithms and protocols have been developed to address these challenges and ensure secure communication.

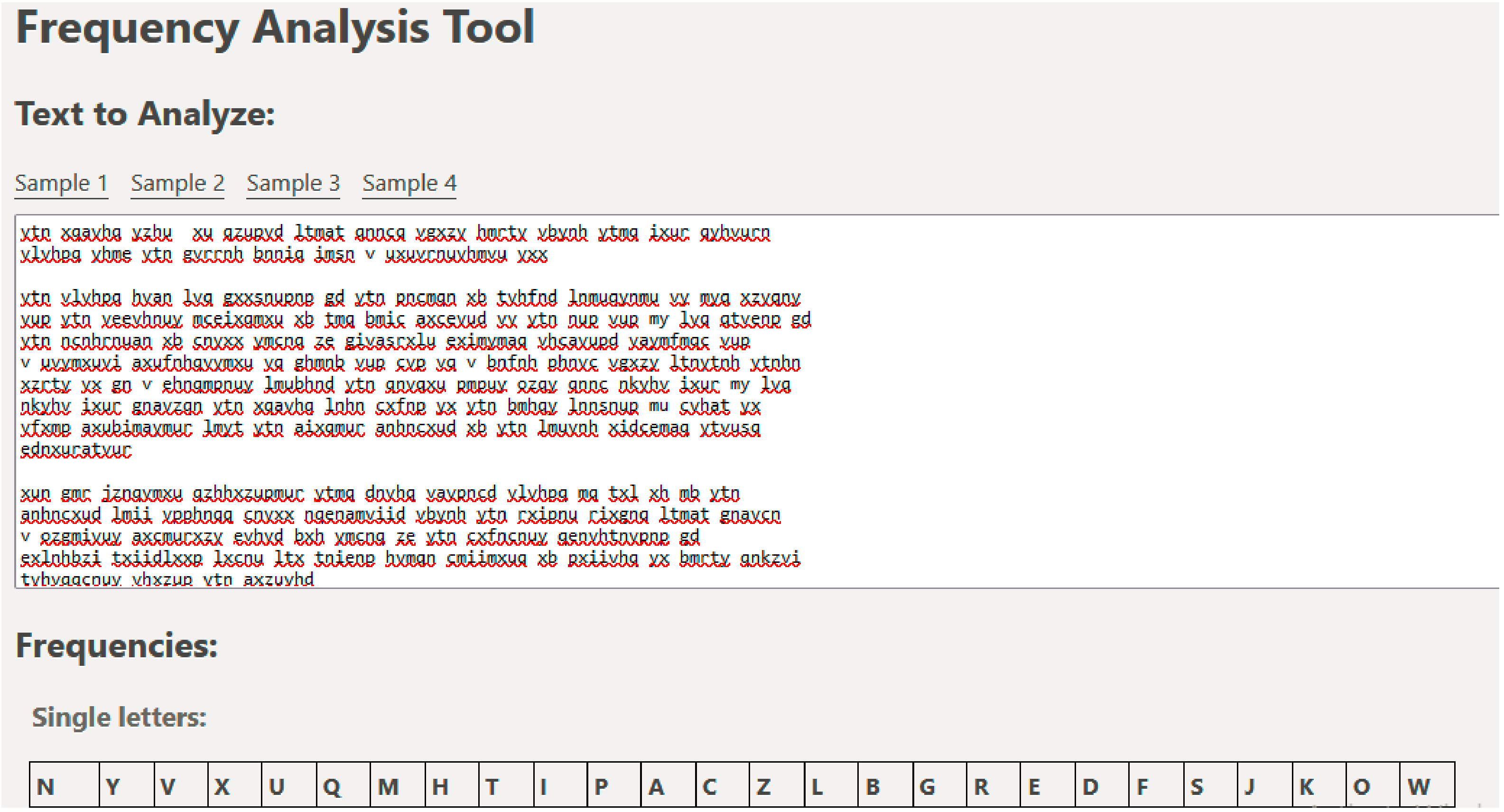
Task 1: Frequency Analysis

In this task we are given a cipher-text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Our job is to find out the original text using frequency analysis. It is known that the original text is an English article.

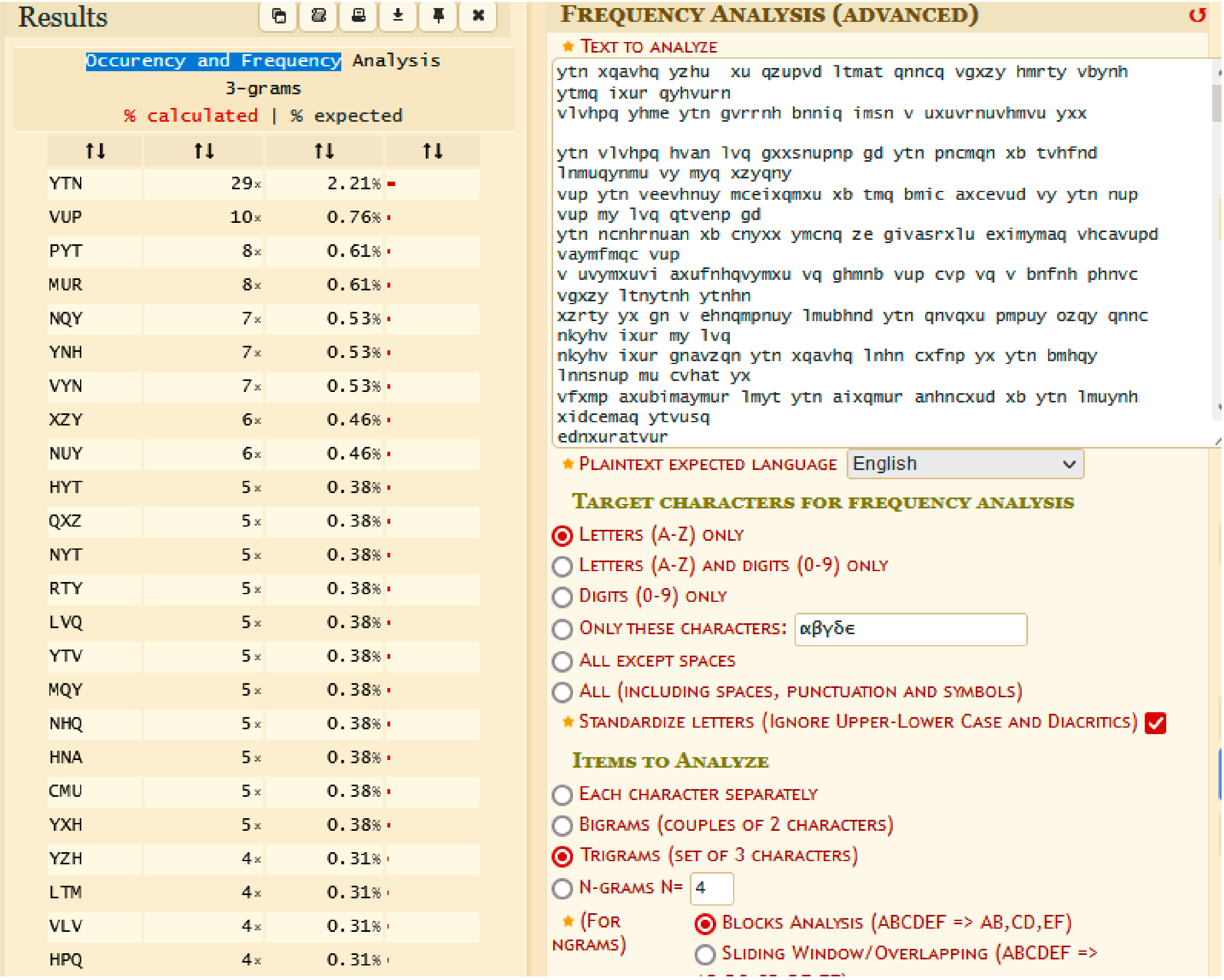
- First we download the file "ciphertext.txt" from "https://seedsecuritylabs.org/Labs_16.04/Crypto/Crypto_Encryption/" and save it on our machine.



- Using the website "https://wilsoa.github.io/gallery/frequency_analysis.html" we analyze the text that we have just downloaded. We are looking for the most common bigrams and trigrams.



- Using this website "https://www.dcode.fr/frequency-analysis" to calculate the occurrence and frequency of bigrams and trigrams letters.



- This website "https://www3.nd.edu/~busiforc/handouts/cryptography/Letter%20Frequencies.html" gives us all the common letters, bigrams, trigrams and quadrigrams.

Results from Project Gutenberg
Analysis of 9,481 English works (3.98 GiB) from Project Gutenberg (the extracted contents of the 2003 PG DVD, plain text files only, minus th
7-bit-clean encoding), after stripping off the common boilerplate text present in every file so as not to skew results, yielded the following freque
Letters
Of 3,104,375,038 letters scanned:
1. e (390395169, 12.575645%)
2. t (282039486, 9.085226%)
3. a (248362256, 8.000395%)
4. o (235661502, 7.591270%)
5. i (214822972, 6.920007%)
6. n (214319386, 6.903785%)
7. s (196844692, 6.340880%)
8. h (193607737, 6.236609%)
9. r (184990759, 5.959034%)
10. d (134044565, 4.317924%)
11. l (125951672, 4.057231%)
12. u (88219598, 2.841783%)
13. c (79962026, 2.575785%)
14. m (79502870, 2.560994%)
15. f (72967175, 2.350463%)
16. w (69069021, 2.224893%)
17. g (61549736, 1.982677%)
18. y (59010696, 1.900888%)
19. p (55746578, 1.795742%)
20. b (47673928, 1.535701%)
21. v (30476191, 0.981717%)
22. k (22969448, 0.739906%)
23. x (5574077, 0.179556%)
24. j (4507165, 0.145188%)
25. q (3649838, 0.117571%)
26. z (2456495, 0.079130%)

Bigrams
Of 2,383,373,483 bigrams scanned:
1. th (92535489, 3.882543%)
2. he (87741289, 3.681391%)

- This is the result

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
c	f	m	y	p	v	b	r	l	q	x	w	i	e	j	d	s	g	k	h	n	a	z	o	t	u

the oscars turn on sunday which seems about right after this long strange awards trip the bagger feels like a nonagenarian too the awards race was bookended by the demise of harvey weinstein at its outset and the apparent implosion of his film company at the end and it was shaped by the emergence of metoo times up blackgown politics armcandy activism and a national conversation as brief and mad as a fever dream about whether there ought to be a president winfrey the season didnt just seem extra long it was extra long because the oscars were moved to the first weekend in march to avoid conflicting with the closing ceremony of the winter olympics thanks pyeongchang one big question surrounding this years academy awards is how or if the ceremony will address metoo especially after the golden globes which became a jubilant comingout party for times up the movement spearheaded by powerful hollywood women who helped raise millions of dollars to fight sexual harassment around the country signaling their support golden globes attendees swathed themselves in black sported lapel pins and sounded off about sexist power imbalances from the red carpet and the stage on the air e was called out about pay inequity after its former anchor catt sadler quit once she learned that she was making far less than a male cohost and during the ceremony natalie portman took a blunt and satisfying dig at the allmale roster of nominated directors how could that be topped as it turns out at least in terms of the oscars it probably wont be women involved in times up said that although the globes signified the initiatives launch they never intended it to be just an awards season campaign or one that became associated only with redcarpet actions instead a spokeswoman said the group is working behind closed doors and has since amassed million for its legal defense fund which after the globes was flooded with thousands of donations of or less from people in some countries no call to wear black gowns went out in advance of the oscars though the movement will almost certainly be referenced before and during the ceremony especially since vocal metoo supporters like ashley judd laura dern and nicole kidman are scheduled presenters another feature of this season no one really knows who is going to win best picture arguably this happens a lot of the time inarguably the nailbiter narrative only serves the awards hype machine but often the people forecasting the race socalled oscarologists can make only educated guesses the way the academy tabulates the big winner doesnt help in every other category the nominee with the most votes wins but in

- Our analysis was conducted in the following sequential steps:
  - **Step 1:** Our initial focus was on identifying the frequency of single characters in the text. Given the common usage and frequency of letters in the English language, the character with the highest occurrence is likely to correspond to either 'e', 't', or 'a' which are the most commonly used letters in English. By mapping the frequency of letters in our cipher-text against their typical frequencies in English, we made educated assumptions about possible letter replacements.
  - **Step 2:** Subsequent to analyzing single character frequencies, we progressed to bigram analysis - pairs of letters in the text. Analyzing the frequency of these bigrams provided us with valuable information about possible word structures in the text. We noted the highest frequency bigrams in the cipher-text and compared them with common English language bigrams. This gave us further insights into the potential mapping of the cipher.

- **Step 3:** We then conducted a trigram analysis, identifying the three-letter sequences with the highest occurrence in the text. Trigrams provide a more specific glimpse into potential words or word fragments. Comparing these with common English language trigrams further narrowed down our list of possible monoalphabetic substitutions.

Throughout the process, we continually cross-verified our assumptions and refined our substitution mappings, improving the accuracy of our decryption as we included more complex letter groupings. The interplay between the single character, bigram, and trigram analyses gradually allowed us to piece together the underlying monoalphabetic cipher and ultimately decipher the original English text.

Task 2: Encryption using Different Ciphers and Modes

In this task, we will play with various encryption algorithms and modes. Running command “man enc” gives us a description about what encryption algorithms we can use.

```
       bf-cbc            Blowfish in CBC mode
       bf                Alias for bf-cbc
       bf-cfb            Blowfish in CFB mode
       bf-ecb            Blowfish in ECB mode
       bf-ofb            Blowfish in OFB mode

       cast-cbc          CAST in CBC mode
       cast              Alias for cast-cbc
       cast5-cbc         CAST5 in CBC mode
       cast5-cfb         CAST5 in CFB mode
       cast5-ecb         CAST5 in ECB mode
       cast5-ofb         CAST5 in OFB mode

       des-cbc           DES in CBC mode
       des               Alias for des-cbc
       des-cfb           DES in CBC mode
       des-ofb           DES in OFB mode
       des-ecb           DES in ECB mode

       des-ede-cbc       Two key triple DES EDE in CBC mode
       des-ede           Two key triple DES EDE in ECB mode
       des-ede-cfb       Two key triple DES EDE in CFB mode
       des-ede-ofb       Two key triple DES EDE in OFB mode

       des-ede3-cbc      Three key triple DES EDE in CBC mode
```

These are a few of many algorithms that we can use.
We choose: “aes-[128|192|256]-ofb”, “RC2-CBC” and “DES-CBC”.

aes-[128|192|256]-ofb:

AES-OFB (Advanced Encryption Standard - Output Feedback) is a mode of operation for the AES symmetric encryption algorithm. It is used to encrypt data in a secure and efficient manner. The mode operates by converting the block cipher (AES) into a stream cipher, where the encryption is applied on a stream of data rather than individual blocks.

The notation AES-[128|192|256]-OFB refers to different key sizes (128-bit, 192-bit, or 256-bit) for the AES cipher used in the OFB mode. These key sizes determine the level of security and the length of the secret key used in the encryption process.

The OFB mode works as follows:

1. Initialization: An initialization vector (IV) is required for the encryption process. The IV is a random value that is used to start the encryption. It needs to be unique for each encryption operation but does not need to be kept secret.

2. Key Expansion: The secret key used in AES-OFB mode is expanded into a set of round keys that are used in the encryption process.

3. Feedback: AES-OFB creates a keystream by encrypting the IV (or the previous ciphertext block) using the AES algorithm with the secret key. The output of this encryption becomes the keystream.

4. Encryption: The keystream is then XORed (bitwise exclusive OR) with the plaintext data. This XOR operation masks the plaintext, effectively encrypting it.

5. Ciphertext: The result of the XOR operation becomes the ciphertext. It is important to note that the encryption and decryption process in AES-OFB are the same.

One of the advantages of using AES-OFB is that it allows for random access encryption. This means that individual portions of a file or stream can be decrypted without having to decrypt the entire file. It also provides error propagation, where an error in one encrypted block does not affect the decryption of subsequent blocks.

AES-OFB is considered secure for most practical purposes, provided that strong keys and unique IVs are used. It is widely used in various applications, including secure communication protocols, disk encryption, and secure file storage.

- So first let's create the key. It will be a 128 bits (16 bytes) key.

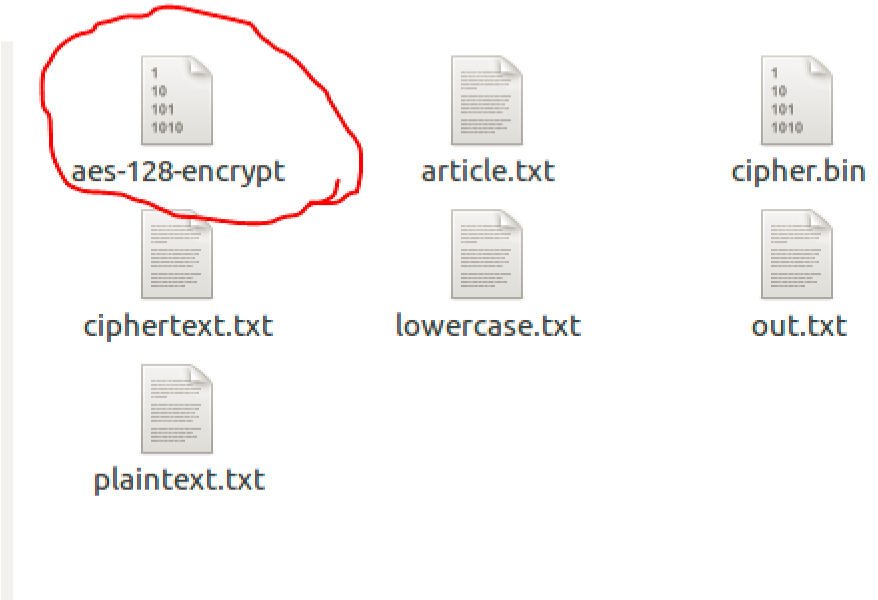
```
[05/25/2023 09:54] Attacker: openssl rand -hex 16
5a278e810765071f2e610f4f0217c120
```

- Now we initialize vector (IV). It will be a 128 bits vector.

```
[05/25/2023 11:01] Attacker: openssl rand -hex 16
7e454462bfc9dbb505567d194dc99684
```

- Now that we have the key and the initialize vector. We can finally encrypt.

```
[05/25/2023 11:34] Attacker: openssl enc -aes-128-ofb -
e -in out.txt -out aes-128-encrypt -K 5a278e810765071f2
e610f4f0217c120 -iv 7e454462bfc9dbb505567d194dc99684
[05/25/2023 11:34] Attacker: ls
aes-128-encrypt  ciphertext.txt  plaintext.txt
article.txt      lowercase.txt
cipher.bin       out.txt
[05/25/2023 11:34] Attacker: 
```



DES-CBC:

DES-CBC (Data Encryption Standard - Cipher Block Chaining) is a symmetric encryption algorithm that uses a 56-bit key and operates on fixed-size blocks of data. It was widely used in the past but is now considered relatively weak due to its short key length.

Here are some key features and characteristics of DES-CBC:

1. Block Cipher: DES-CBC operates on fixed-size blocks of data, typically 64 bits in length. It divides the input data into blocks and encrypts each block independently.

2. Key Size: DES-CBC uses a 56-bit key, which consists of 8 bytes. The key is used in the encryption and decryption process to transform the plaintext into ciphertext and vice versa.

3. Cipher Block Chaining (CBC): CBC is a mode of operation used with block ciphers like DES. In CBC mode, each plaintext block is XORed with the previous ciphertext block before encryption. This XOR operation introduces dependencies between blocks and provides better security than simple ECB (Electronic Codebook) mode.

4. Initialization Vector (IV): CBC mode requires an IV, which is a random value used to initialize the encryption process. The IV should be unique for each encryption operation but does not need to be kept secret. It is typically 64 bits (8 bytes) long, matching the block size of DES.

5. Security Strength: DES-CBC is considered relatively weak by modern cryptographic standards due to its short key length. The 56-bit key size

is susceptible to brute-force attacks, and advances in computing power have made it vulnerable to decryption attacks.

6. Legal Restrictions: DES was initially developed by IBM and later standardized by the National Institute of Standards and Technology (NIST). It was subject to export controls and patents, which led to restrictions on its usage and deployment.

7. Deprecation: Due to its vulnerabilities, DES has been deprecated as a secure encryption algorithm. It is no longer recommended for use in new systems or applications where stronger encryption algorithms like AES (Advanced Encryption Standard) are available.

While DES-CBC was once widely used, it is now considered insecure and outdated for most practical purposes. It is recommended to use stronger encryption algorithms with longer key lengths, such as AES, for better security.

- Let's create the key. It will be a 56 bits (7 bytes) key.

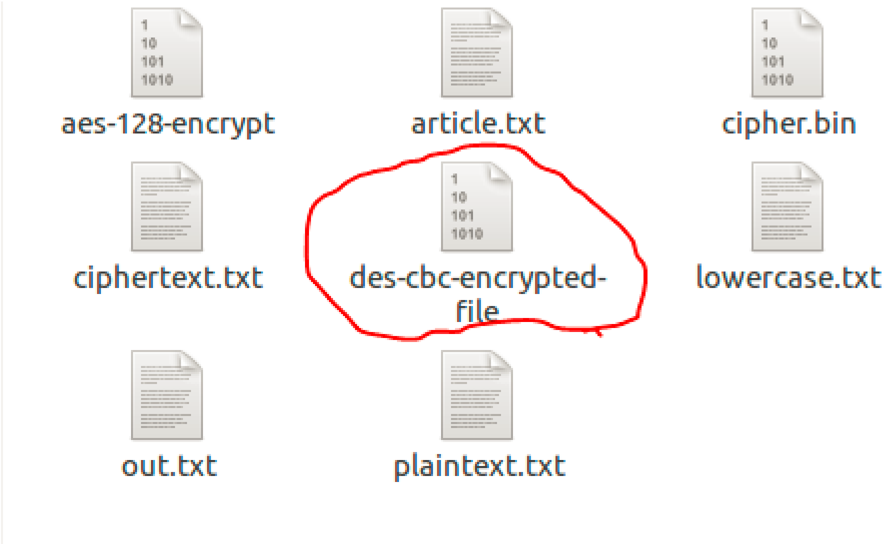
```
[05/25/2023 11:49] Attacker: openssl rand 8 -hex
87978f2105a81b59
```

- Initialize vector iv. It will be 64 bits.

```
[05/25/2023 11:50] Attacker: openssl rand 8 -hex
538e4203c70fce31
```

- Now let's encrypt the file with the key and the vector.

```
[05/25/2023 11:51] Attacker: openssl enc -des-cbc -e -in out.txt -out des-cbc-encrypted-fi
le -K 87978f2105a81b59 -iv 538e4203c70fce31
[05/25/2023 11:53] Attacker: ls
aes-128-encrypt  cipher.bin      des-cbc-encrypted-file  out.txt
article.txt      ciphertext.txt  lowercase.txt           plaintext.txt
[05/25/2023 11:53] Attacker:
```



RC2-CBC:

RC2-CBC (Rivest Cipher 2 - Cipher Block Chaining) is a variant of the RC2 encryption algorithm that operates in CBC mode. It combines the RC2 block cipher with the CBC mode of operation to provide confidentiality and data integrity.

Here are some key features and characteristics of RC2-CBC:

1. Block Cipher: RC2-CBC operates on fixed-size blocks of data, typically 64 bits in length. It encrypts and decrypts data in these fixed-size blocks using the RC2 algorithm.

2. Cipher Block Chaining (CBC): CBC is a mode of operation used with block ciphers. In RC2-CBC, each plaintext block is XORed with the previous ciphertext block before encryption. This XOR operation introduces dependencies between blocks and provides better security than simple ECB (Electronic Codebook) mode.

3. Initialization Vector (IV): CBC mode requires an IV, which is a random value used to initialize the encryption process. The IV should be unique for each encryption operation but does not need to be kept secret. It is typically 64 bits (8 bytes) long, matching the block size of RC2.

4. Variable Key Size: RC2-CBC supports variable key sizes, ranging from 8 to 1,024 bits. The key size can be any multiple of 8 bits. The effective security of RC2-CBC depends on the key size used.

5. Security Strength: RC2-CBC's security strength depends on the key size and the number of rounds applied during the RC2 encryption process. It is important to use an appropriate key size and a sufficient number of rounds to ensure a desired level of security.

6. Weaknesses: Like the original RC2 algorithm, RC2-CBC may have some vulnerabilities, especially with smaller key sizes. It is generally recommended to use larger key sizes and to choose stronger encryption algorithms, such as AES, for better security.

7. Deprecation: Due to its relatively weaker security and the availability of stronger encryption algorithms, RC2 is considered outdated. It is no longer recommended for new systems or applications where stronger encryption options are available.

It's worth noting that while RC2-CBC was once widely used, it has been largely replaced by stronger and more secure encryption algorithms, such as AES. It is recommended to use AES or other modern algorithms for better security in encryption scenarios.

- Creating a 128 bit key (16 bytes).

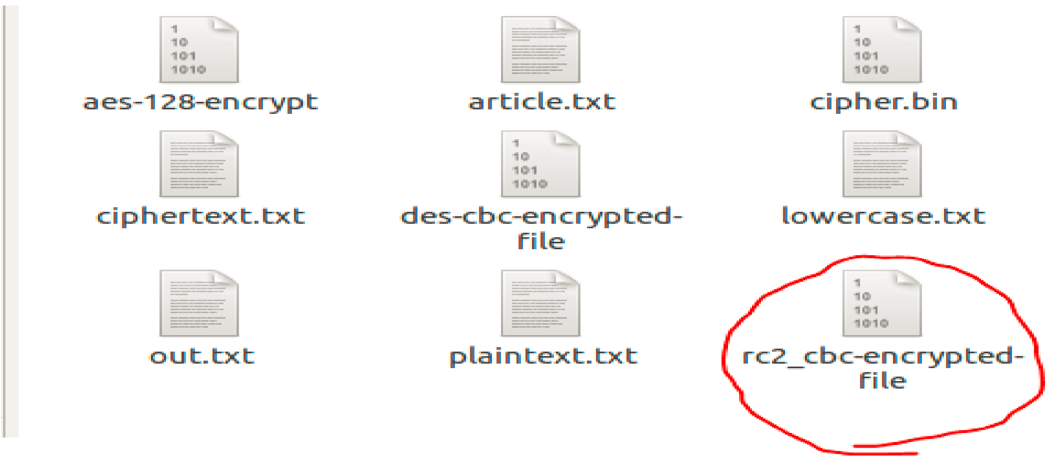
```
[05/25/2023 12:13] Attacker: openssl rand 16 -hex
4dded429f11a990f879af132a6f6f431
[05/25/2023 12:18] Attacker:
```

- Creating the initialize 64 bits (8 bytes) vector

```
[05/25/2023 12:26] Attacker: openssl rand 8 -hex
c798011472fe7f8c
```

- Now let's encrypt the file with the key and the vector.

```
[05/25/2023 12:35] Attacker: openssl enc -rc2-cbc -e -in out.txt
-out rc2_cbc-encrypted-file -K 4dded429f11a990f879af132a6f6f431 -
iv c798011472fe7f8c
[05/25/2023 12:35] Attacker: ls
aes-128-encrypt   des-cbc-encrypted-file   rc2_cbc-encrypted-file
article.txt       lowercase.txt            rc2-encrypted-file
cipher.bin        out.txt
ciphertext.txt    plaintext.txt
[05/25/2023 12:35] Attacker:
```



Task 3: Encryption Mode – ECB vs. CBC

On this task we're gonna encrypt picture using DES-ECB and DES-CBC.

- First, let's encrypt the picture using the DES-ECB algorithm. For the DES-ECB we need to create a key. The key should be a multiple of 56 bits. So we're creating a 56 bit key.

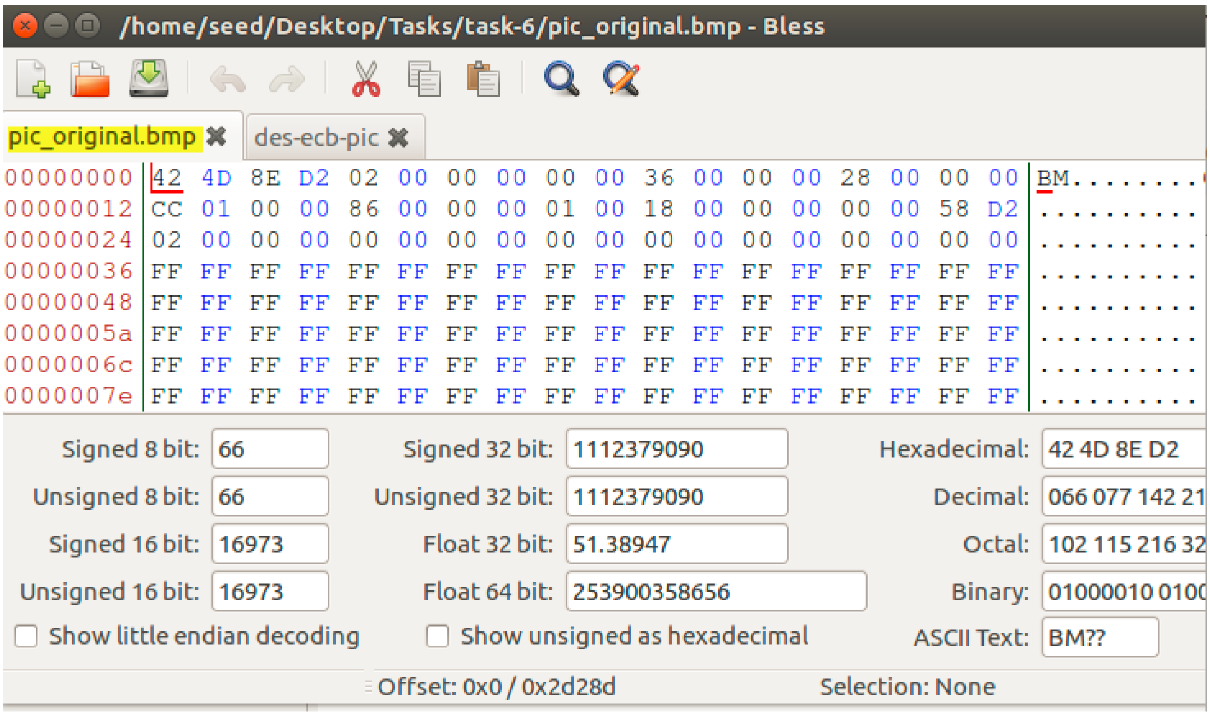
```
[05/26/2023 08:07] Attacker: openssl rand -hex 7
b2a5f92a2c5f28
```

- Now let's encrypt the picture using DES-ECB algo and with the key we have just created.

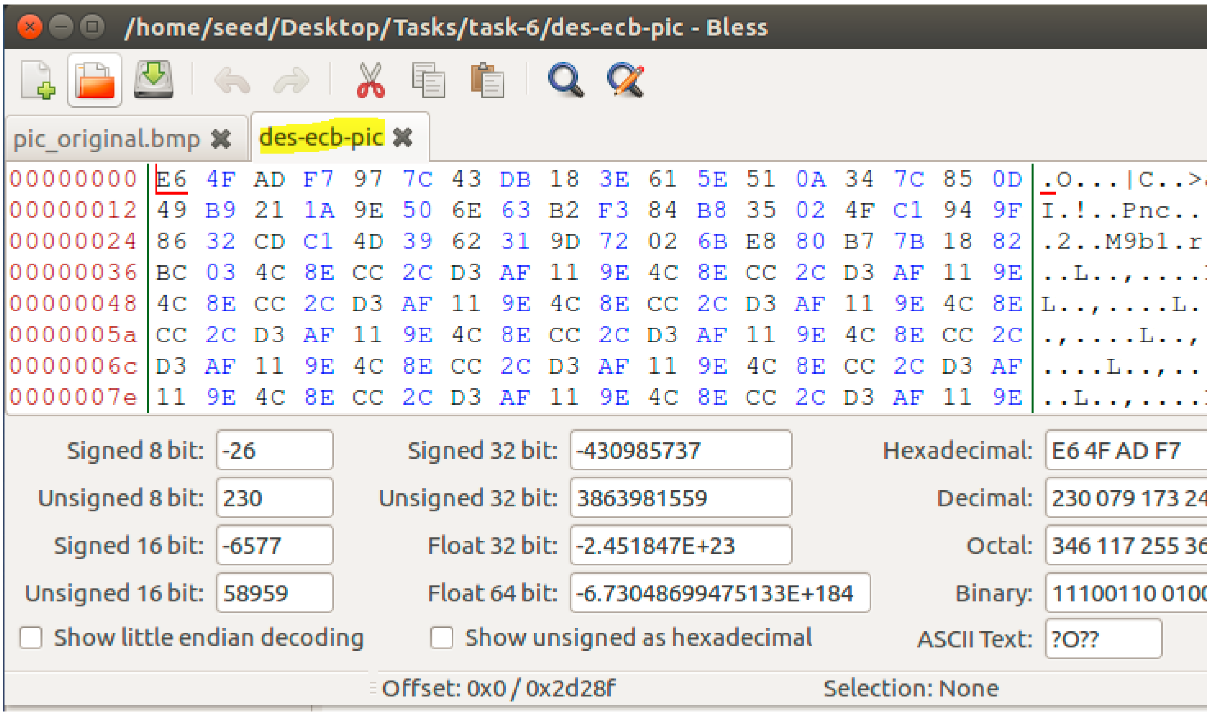
```
[05/26/2023 08:07] Attacker: openssl enc -des-ecb -e -in pic_orig
inal.bmp -out des-ecb-pic -K b2a5f92a2c5f28
[05/26/2023 08:07] Attacker: 
```

- Using "bless" we can see the binary of both files, "pic_original.bmp" and "des-ecb-pic".

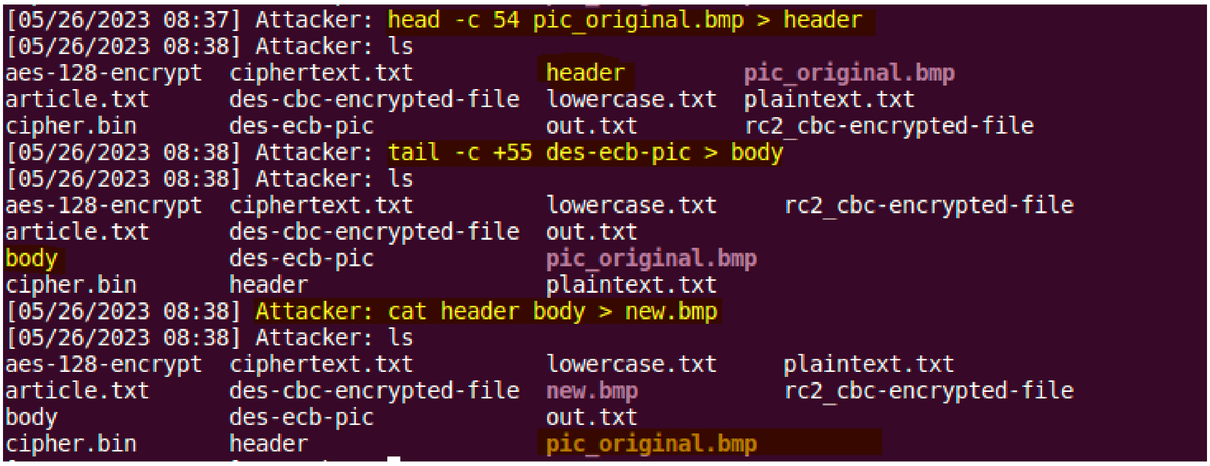
- "pic_original.bmp" information



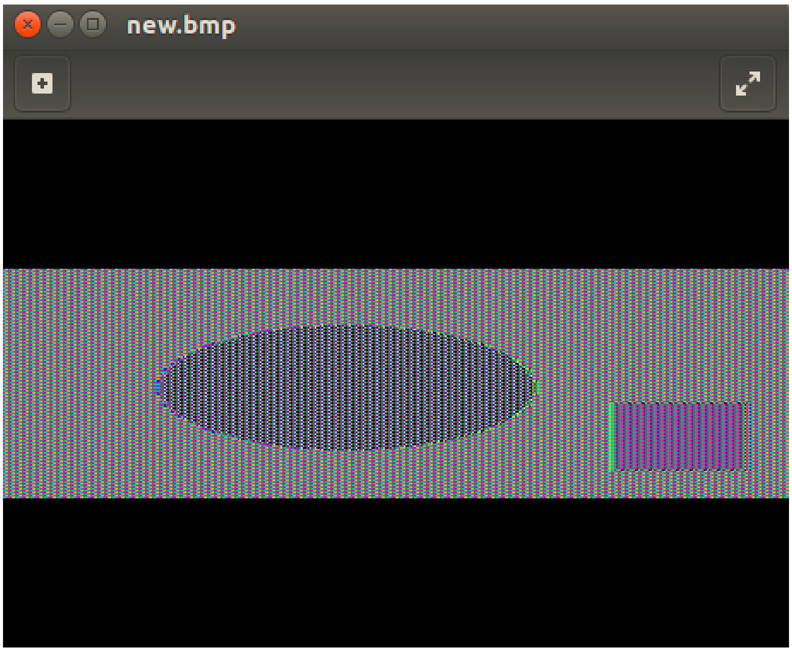
- "des-ecb-pic" information



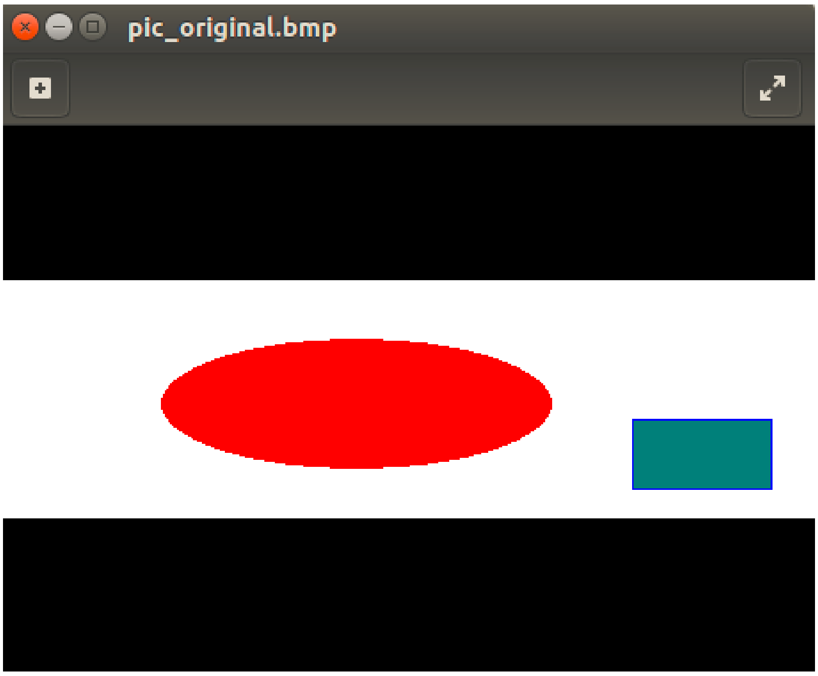
- To make the encrypted picture legitimate, we need to replace the header of the encrypted picture with that of the original picture. We create a new picture with the header of the original picture and the body of the encrypted picture.



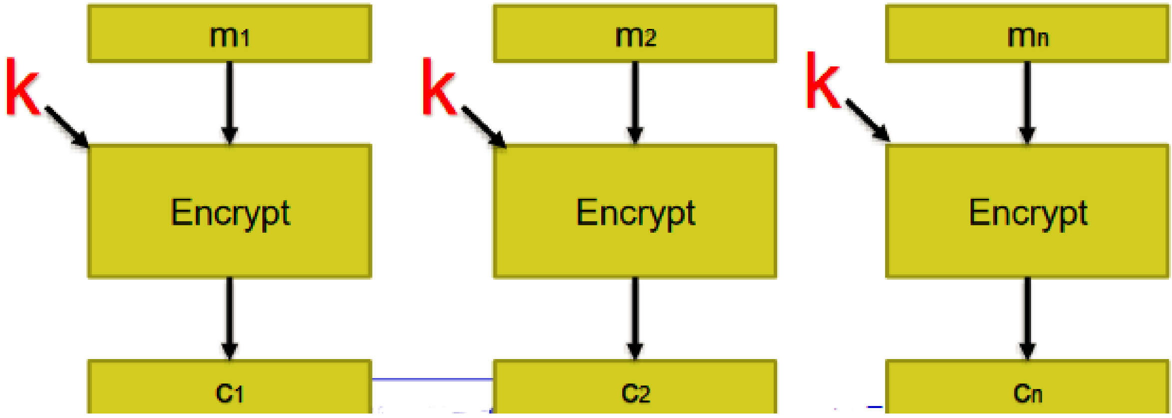
- The outcome is this - this is the encrypted picture.



- This is the original picture



In ECB mode, the plaintext is divided into fixed-size blocks (e.g., 128 bits) and each block is encrypted independently. This means that identical blocks of the image will produce identical ciphertext blocks. As a result, patterns or structures in the original image can be preserved in the encrypted image. For example, if there are repeated patterns or distinct areas in the image, they will be visible in the encrypted version.



- Now we are going to do the whole process but this time using CBC. For DES-CBC we need to create a 64 bit key and a 64 bit IV vector.

- Creating a 64 bit key

```
[05/26/2023 11:35] Attacker: openssl rand -hex 8
05f6a0db7f9bd5e9
```

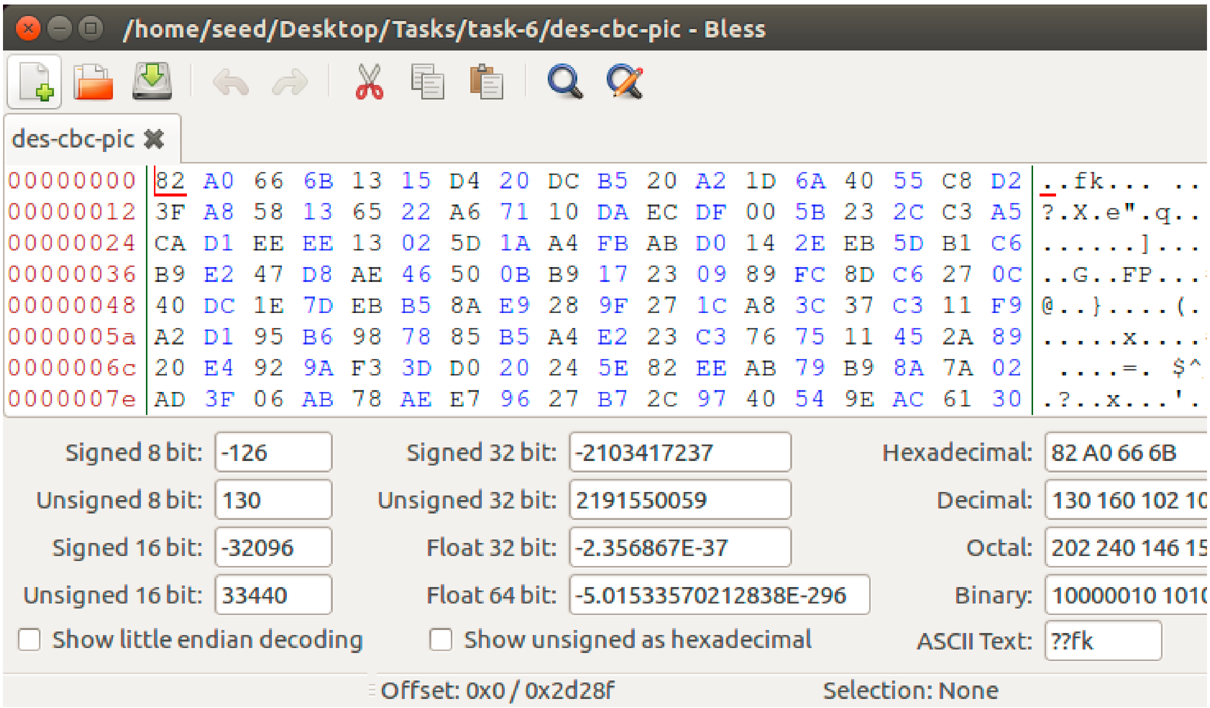
- Creating a 64 bit vector

```
[05/26/2023 11:38] Attacker: openssl rand -hex 8
020349e826e82e05
```

- Encrypting the picture using des-cbc

```
[05/26/2023 11:38] Attacker: openssl enc -des-cbc -e -in pic_orig
inal.bmp -out des-cbc-pic -K 05f6a0db7f9bd5e9 -iv 020349e826e82e0
5
[05/26/2023 11:40] Attacker: 
```

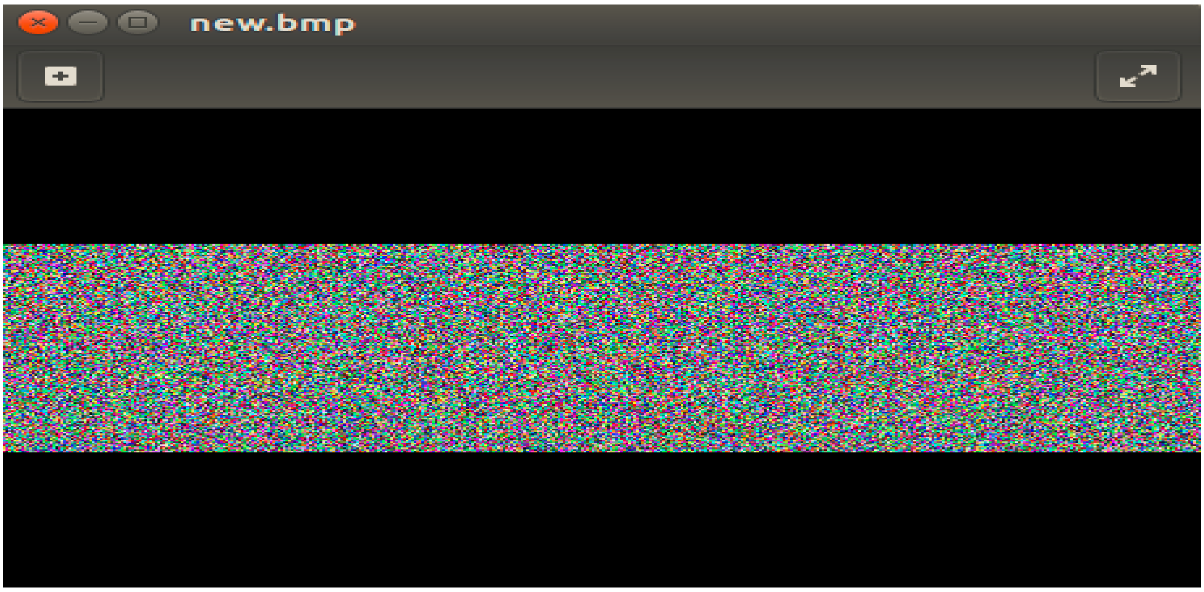
- Using “bless” to see the information of the encrypted picture

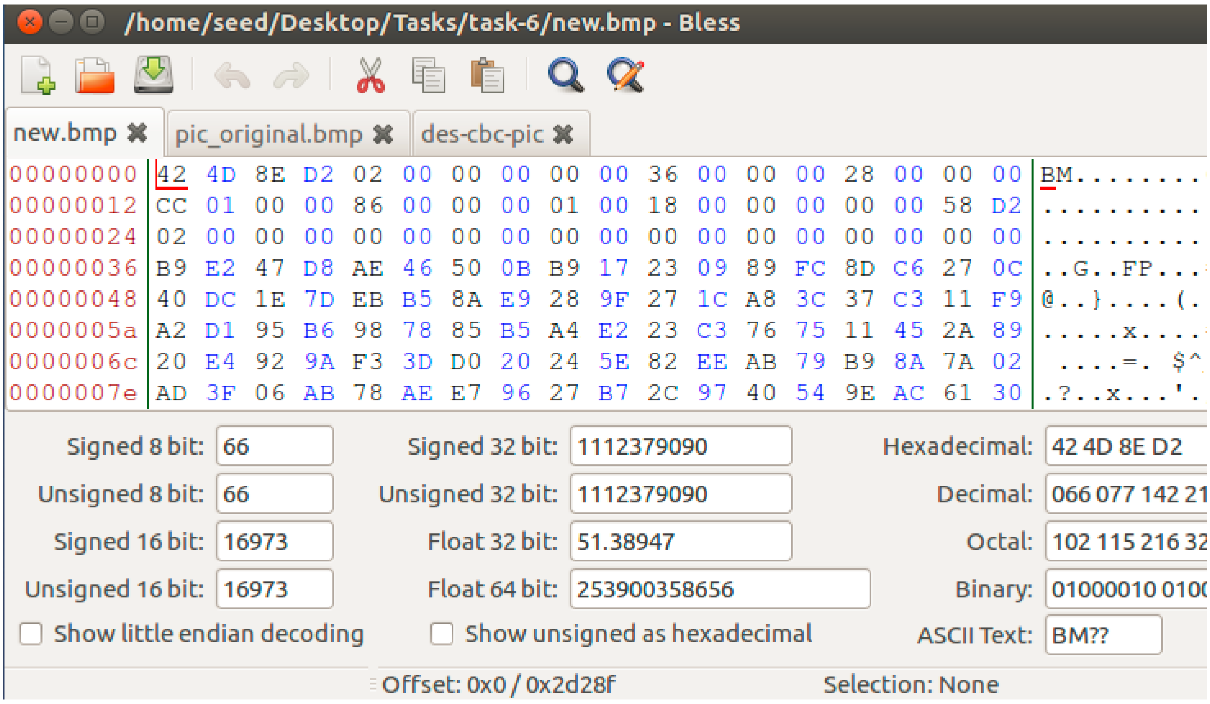


- Creating a new file with the header of the original picture and body of the encrypted picture.

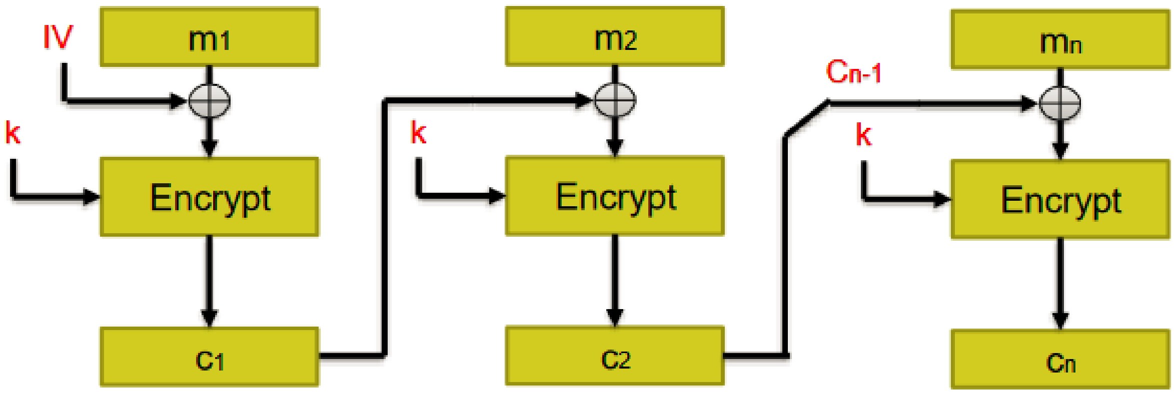
```
[05/26/2023 08:39] Attacker: head -c 54 pic_original.bmp > header
[05/26/2023 11:54] Attacker: tail -c +55 des-cbc-pic > body
[05/26/2023 11:54] Attacker: cat header body > new.bmp
[05/26/2023 11:54] Attacker:
```

- This is the outcome

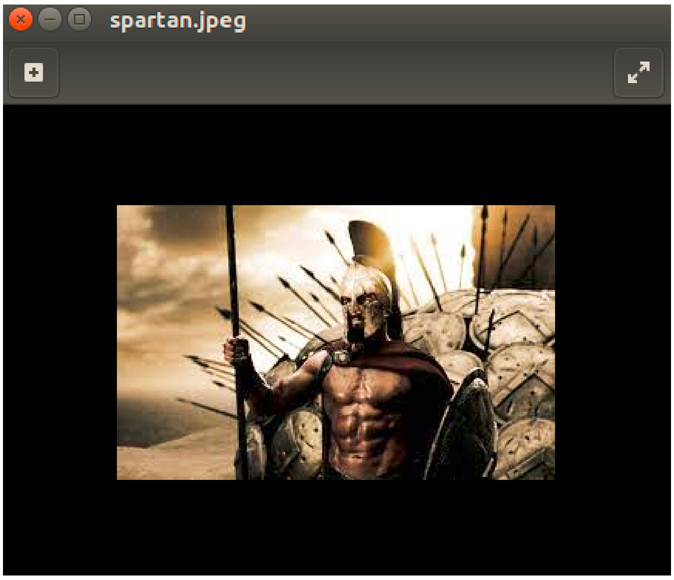




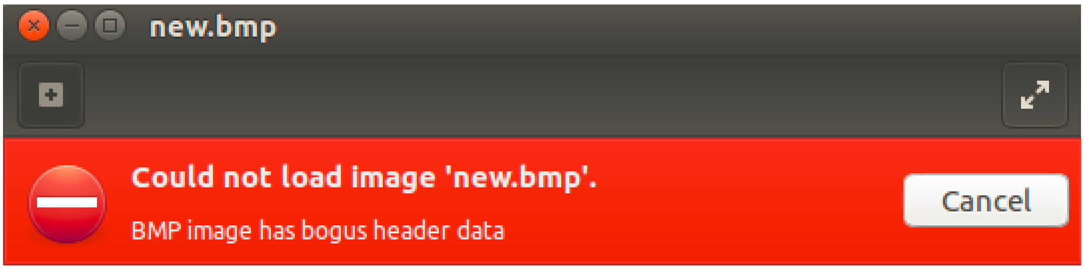
CBC mode provides diffusion, which means that changes in the plaintext will propagate throughout the encrypted image. Each plaintext block is XORed with the previous ciphertext block before encryption, introducing randomness and making it harder to detect patterns or structures in the original image. This ensures that even small changes in the input image will produce significant changes in the encrypted version.



● Now let’s do the same process but this time to picture that we choose. So we downloaded this picture “spartan.jpeg”.



Unfortunately with format jpeg it didn’t work, because it's only works with BMP formats.

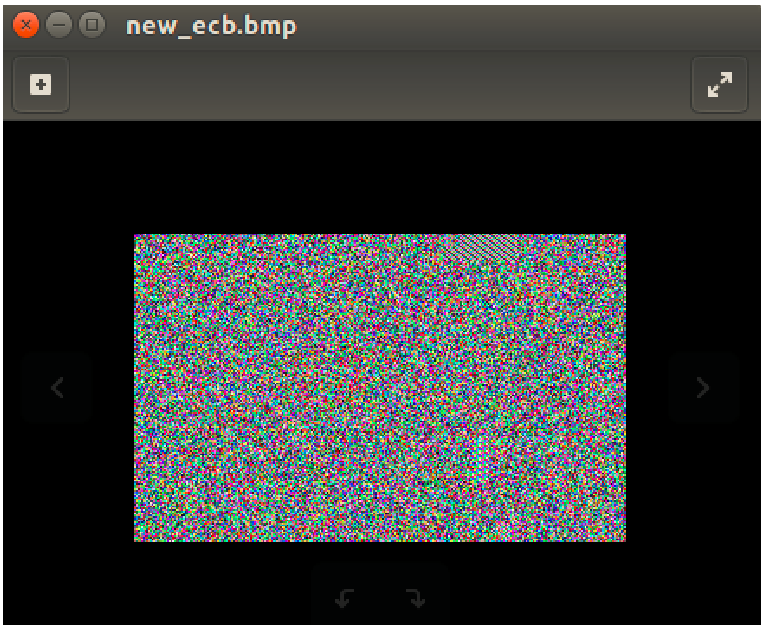


● Let’s convert the picture into BMP format and run the whole process again.

● Encrypting the picture with “aes-ecb-128” and creating a new legit picture with the header of the original picture and with the body of the encrypted picture.

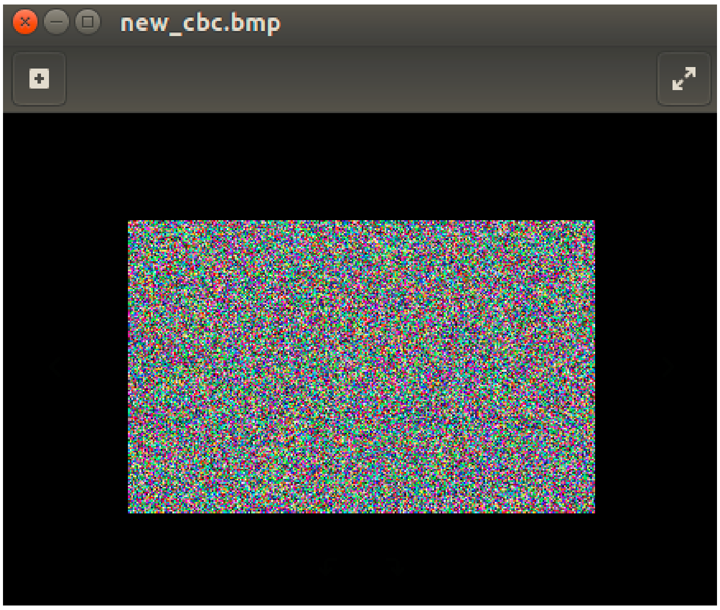
```
[05/31/2023 05:22] Attacker: openssl aes-128-ecb -e -in spartan.b
mp -out enc_ecb_spartan.bmp -K b4039c19ea0139f0b7feb41ba87791c2
[05/31/2023 05:30] Attacker: head -c 54 spartan.bmp  > header
[05/31/2023 05:30] Attacker: tail -c +55 enc_ecb_spartan.bmp > bo
dy
[05/31/2023 05:30] Attacker: cat header body > new_ecb.bmp
[05/31/2023 05:30] Attacker:
```

- This is the result



- Now the same thing only with “aes-128-cbc”

```
[05/31/2023 05:35] Attacker: openssl aes-128-cbc -e -in spartan.b
mp -out enc_cbc_spartan.bmp -K c190c1d19087d4243825cc006525e0ec -
iv 69fdbea83db627f0bec8726965ab650c
[05/31/2023 05:36] Attacker: head -c 54 spartan.bmp  > header_cbc
[05/31/2023 05:36] Attacker: tail -c +55 enc_cbc_spartan.bmp > bo
dy_cbc
[05/31/2023 05:37] Attacker: cat header_cbc body_cbc > new_cbc.bm
p
[05/31/2023 05:37] Attacker: 
```



We got the same results.

When we encrypt with AES-128-ECB, it takes the same size of blocks and encrypts them independently without considering the context or relationship with other blocks.

So probably the input of the picture has no repetitive patterns or blocks and that's why we get a perfect encrypted picture.

For conclusion: In this task, we encrypted BMP images using CBC and ECB encryption modes using the OpenSSL tool and modified the headers for the encrypted images to be able to display them using the BLESS software.

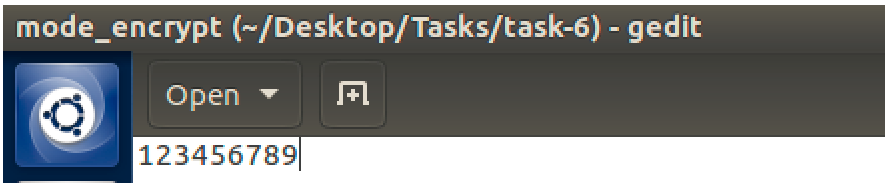
We observed that when using ECB mode, since it encrypts separate blocks independently, the encryption does not effectively conceal the image content, and it is still possible to identify the original image. In contrast, CBC mode encrypts each block in relation to the previous block and allows for more significant changes that prevent guessing the original appearance of the image.

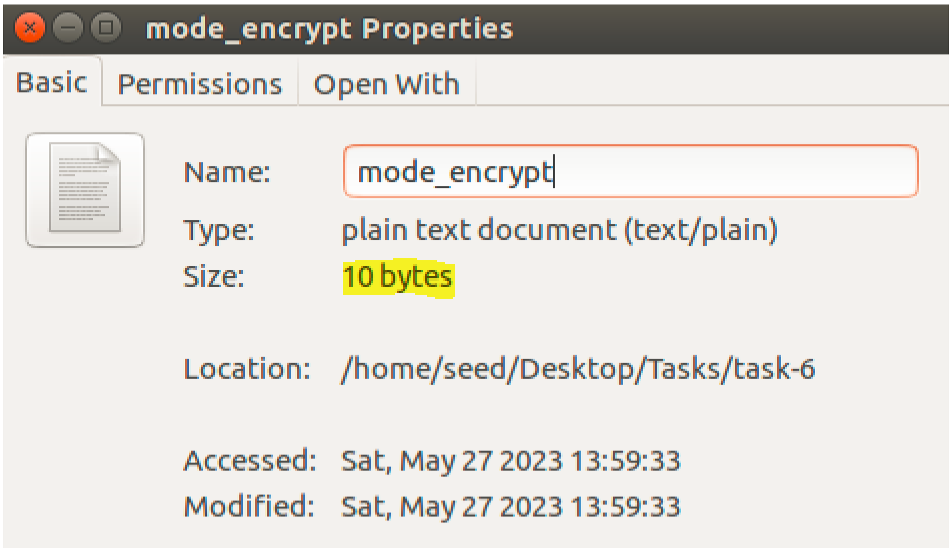
Task 4: Padding

The goal is to contact experiments to understand how this type of padding works.

1) Use ECB, CBC, CFB, and OFB modes to encrypt a file.

- First let's create one file that contains numbers 1-9. Its size 10 bytes.





- Encrypting the file using DES-ECB:

```
[05/27/2023 14:05] Attacker: openssl enc -des-ecb -e -in mode_encrypt -out des_ecb_mode -K f6bc780b05d12f
[05/27/2023 14:05] Attacker:
```

- Encrypting the file using DES-CBC:

```
[05/27/2023 14:40] Attacker: openssl enc -des-cbc -e -in mode_encrypt -out des_cbc_mode -K 4fbec8cd8052bfb2 -iv 52b5eb1283f7ce90
[05/27/2023 14:41] Attacker:
```

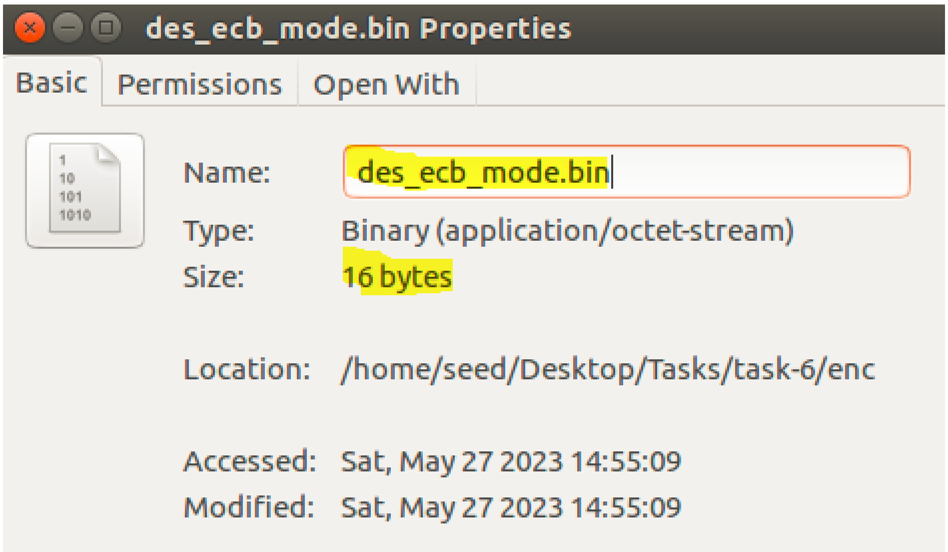
- Encrypting the file using DES-CFB:

```
[05/27/2023 14:48] Attacker: openssl enc -des-cfb -e -in mode_encrypt -out des_cfb_mode -K 6f48d3a93e50ac4b -iv e4717432014ebb67
[05/27/2023 14:48] Attacker:
```

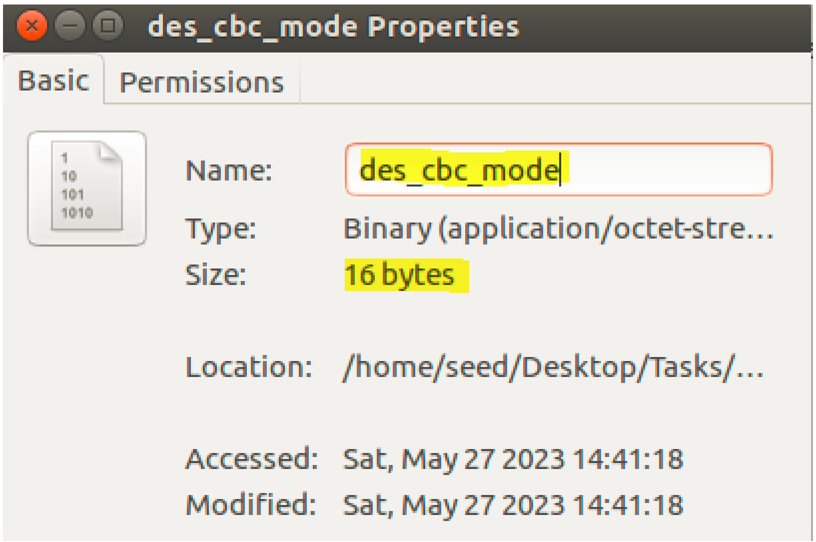
- Encrypting the file using DES-OFB:

```
[05/27/2023 14:50] Attacker: openssl enc -des-ofb -e -in mode_encrypt -out des_ofb__mode -K 6f48d3a93e50ac4b -iv e4717432014ebb67
[05/27/2023 14:50] Attacker:
```

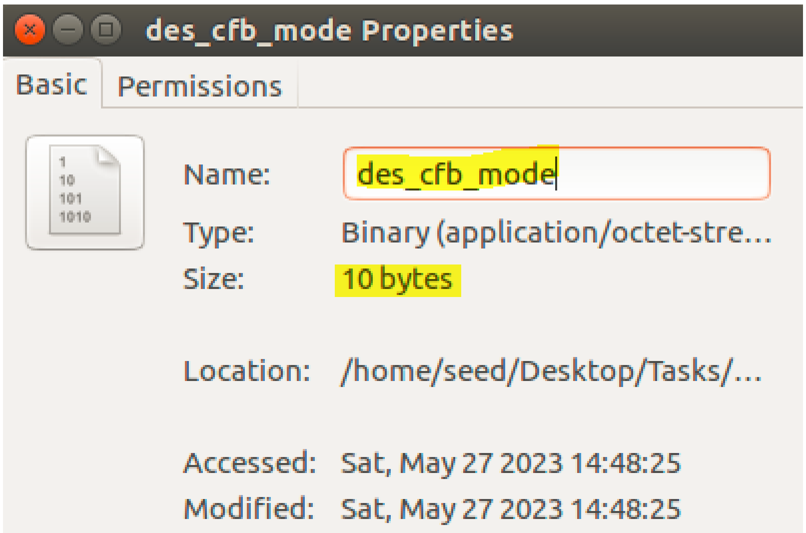
- Now let’s check the size of the encrypted files:
  - For ECB:



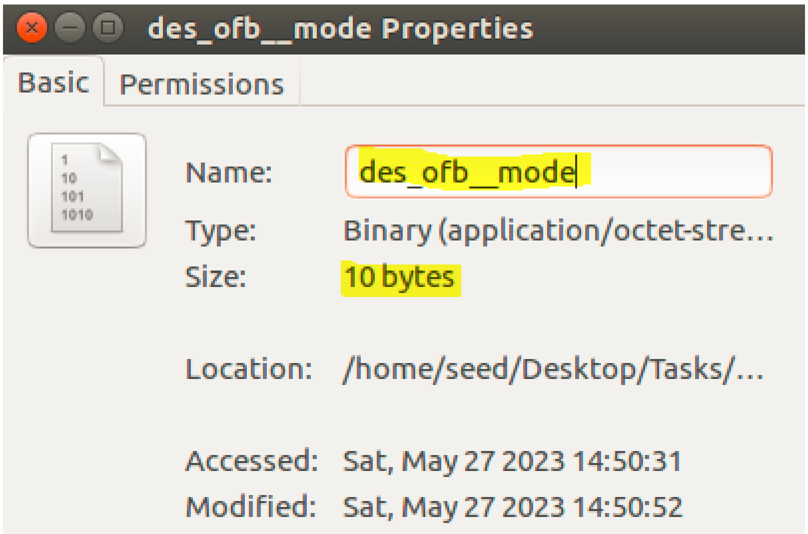
  - For CBC:



○ For CFB



○ For OFB



- Let’s break it down, when using different modes of operation (ECB, CBC, CFB, and OFB) to encrypt a file, the presence or absence of padding depends on the specific mode. Here's a breakdown of the modes and their padding requirements:

1. ECB (Electronic Codebook):
    - Padding: Yes
    - Explanation: ECB does require padding. In ECB mode, the plaintext is divided into fixed-size blocks, and each block is encrypted independently using the same key. If the plaintext is not a multiple of the block size, padding is necessary to ensure that each block is of the correct length. Padding ensures that the last block can be properly encrypted.
2. CBC (Cipher Block Chaining):
    - Padding: Yes
    - Explanation: CBC requires padding. In CBC mode, each plaintext block is XORed with the previous ciphertext block before encryption. Padding is necessary to ensure that the last block can be properly encrypted and that the decryption process can accurately remove the padding.
3. CFB (Cipher Feedback):
    - Padding: No
    - Explanation: CFB mode does not require padding. In CFB mode, the encryption process operates on smaller units called feedback blocks, which can be smaller than the block size. The feedback blocks are used to create a stream of pseudo-random bits that are XORed with the plaintext. Since the feedback blocks can have a smaller size, there is no need for padding to align with the block size.
4. OFB (Output Feedback):
    - Padding: No
    - Explanation: OFB mode does not require padding. OFB mode operates in a similar manner to CFB mode, generating a stream of pseudo-random bits that are XORed with the

plaintext. The feedback blocks used in OFB mode can have a smaller size, allowing for encryption of non-multiple-of-block-size plaintext without the need for padding.

In summary, ECB and CBC modes require padding to ensure that the plaintext is a multiple of the block size, while CFB and OFB modes do not require padding due to their feedback-based operation, which allows for encryption of non-multiple-of-block-size plaintext. However, it's important to note that ECB mode is not recommended for secure encryption due to its lack of diffusion, while CBC, CFB, and OFB modes provide better security features.

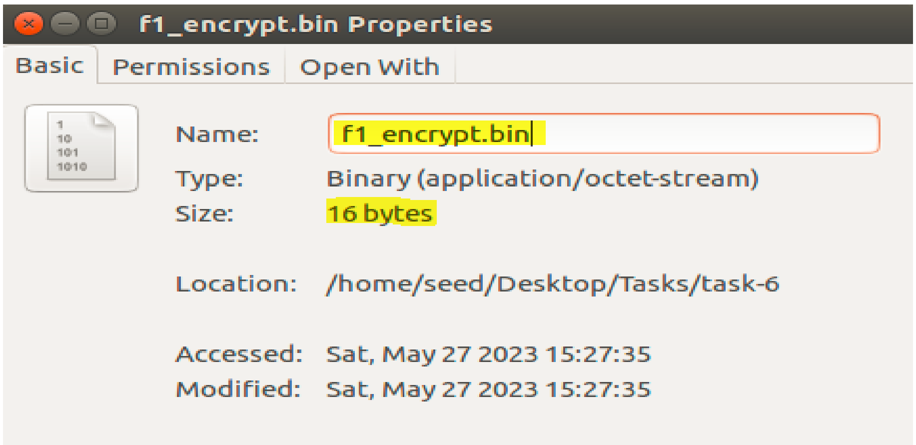
2) Creating 3 files with different sizes. Bytes, 10 bytes, and 16 bytes.

```
[05/27/2023 15:20] Attacker: echo -n "12345" > f1.txt
[05/27/2023 15:20] Attacker: echo -n "1234567890" > f2.
txt
[05/27/2023 15:20] Attacker: echo -n "1234567890ABCDEF"
 > f3.txt
[05/27/2023 15:20] Attacker: 
```

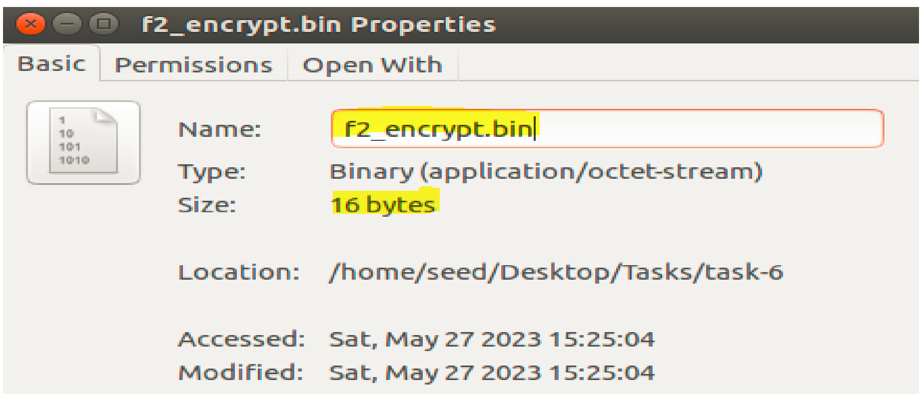
- Encrypting f1,f2 and f3 files with “AES-128-CBC”

```
[05/27/2023 15:23] Attacker: openssl enc -aes-128-cbc -
e -in f1.txt -out f1_encrypt.bin -K 9047a724710de500af9
a5ab0e9517407 -iv 1fb14ffef68938f30c889bb3887a2e19
[05/27/2023 15:24] Attacker: openssl enc -aes-128-cbc -
e -in f2.txt -out f2_encrypt.bin -K 9047a724710de500af9
a5ab0e9517407 -iv 1fb14ffef68938f30c889bb3887a2e19
[05/27/2023 15:25] Attacker: openssl enc -aes-128-cbc -
e -in f3.txt -out f3_encrypt.bin -K 9047a724710de500af9
a5ab0e9517407 -iv 1fb14ffef68938f30c889bb3887a2e19
[05/27/2023 15:25] Attacker:
```

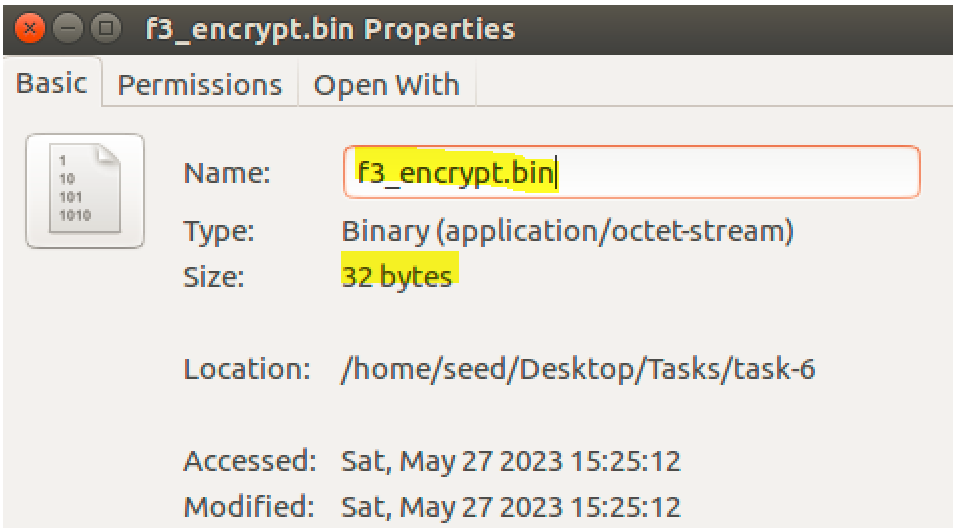
- Let’s see the files size:
  - f1_encryped.txt:



  - f2_encryped.txt:



- f3_encryped.txt:



- When encrypting a file using AES-128-CBC mode, the output encrypted file size may be different from the input file size due to the following reasons:
  1) Padding: AES operates on fixed-size blocks (128 bits or 16 bytes). If the input file size is not an exact multiple of the block size, padding is applied to ensure that the plaintext is properly aligned. Padding adds extra bytes to the plaintext to fill the remaining space in the last block. OpenSSL uses PKCS#5 padding (also known as PKCS#7 padding) by default. This padding ensures that during decryption, the original plaintext can be correctly retrieved.

  2) Initialization Vector (IV): The IV is an essential component of AES-CBC mode. It is used to randomize the encryption process and add uniqueness to each encryption operation, even if the same key is used. The IV is typically the same

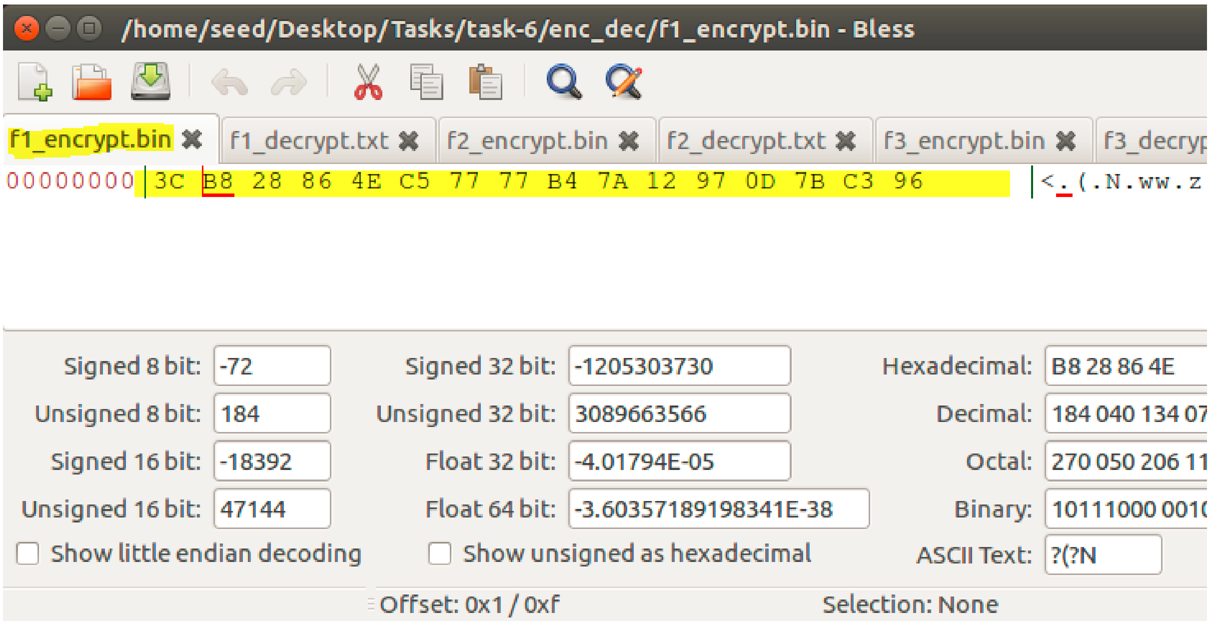
size as the block size (16 bytes for AES). When encrypting, the IV is usually prepended to the ciphertext to ensure it is available for decryption. Therefore, the output encrypted file size will be larger than the input file size by the length of the IV.

So in our case, where the input file sizes are 8 bytes, 10 bytes, and 16 bytes, the output encrypted file sizes being 16 bytes, 16 bytes, and 32 bytes respectively can be attributed to the addition of padding and the inclusion of the IV.

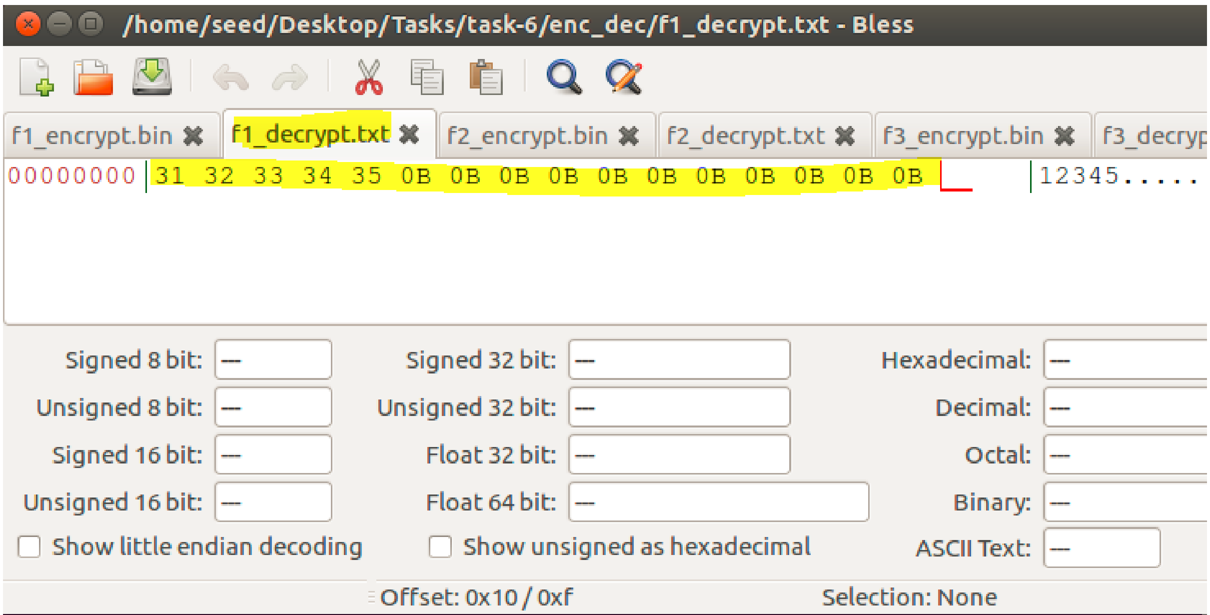
- We would like to see what is added to the padding during the encryption. So now we are going to decrypt the encrypted files. By using "bless" we will compare between the encrypted and decrypted files.

```
[05/27/2023 16:28] Attacker: openssl aes-128-cbc -d -in
 f1_encrypt.bin -out f1_decrypt.txt -K 9047a724710de500
af9a5ab0e9517407 -iv 1fb14ffef68938f30c889bb3887a2e19 -
nopad
[05/27/2023 16:28] Attacker: openssl aes-128-cbc -d -in
 f2_encrypt.bin -out f2_decrypt.txt -K 9047a724710de500
af9a5ab0e9517407 -iv 1fb14ffef68938f30c889bb3887a2e19 -
nopad
[05/27/2023 16:29] Attacker: openssl aes-128-cbc -d -in
 f3_encrypt.bin -out f3_decrypt.txt -K 9047a724710de500
af9a5ab0e9517407 -iv 1fb14ffef68938f30c889bb3887a2e19 -
nopad
[05/27/2023 16:29] Attacker:
```

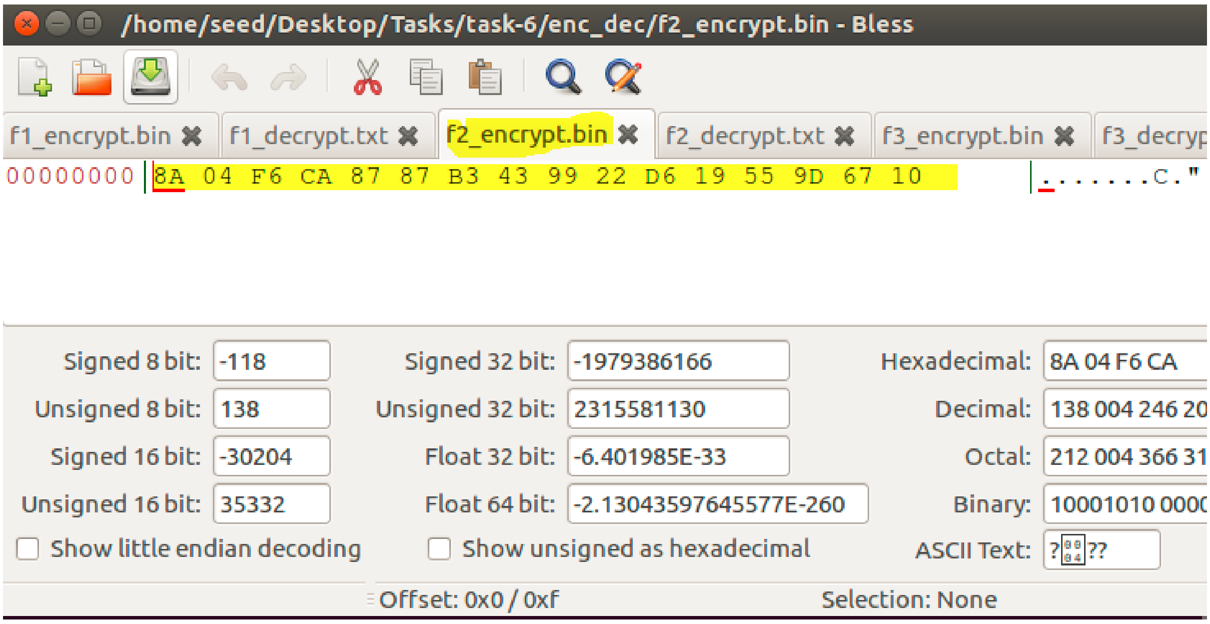
- Now let's see the difference between the encrypt file and the decrypt file.
    - **f1_encrypt.bin**



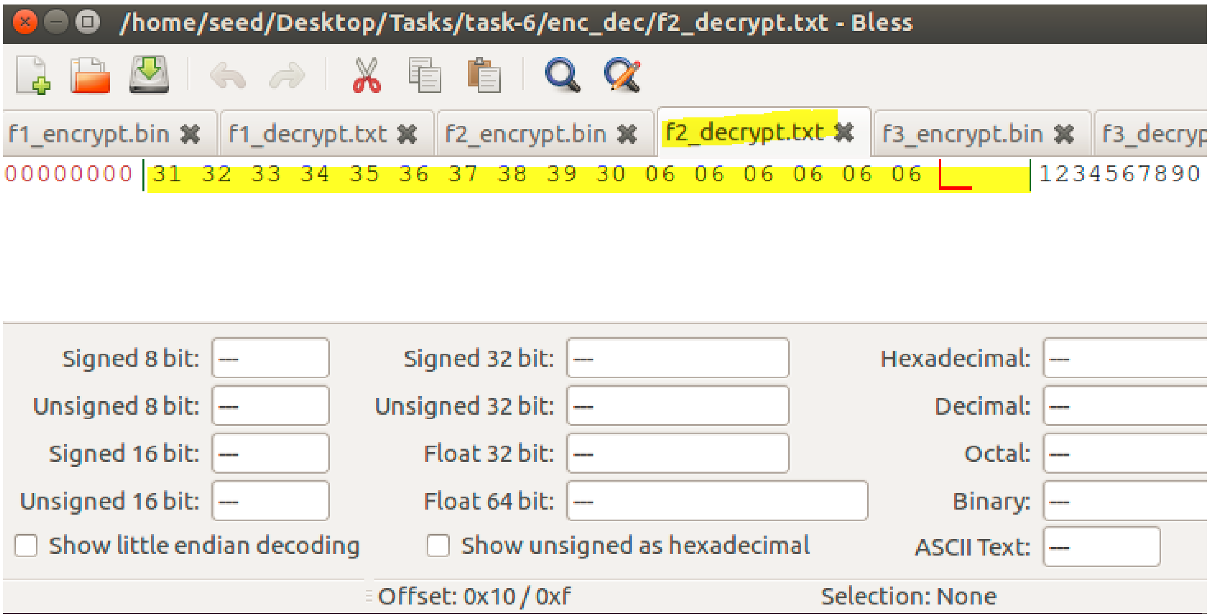
    - **f1_decrypt.txt**



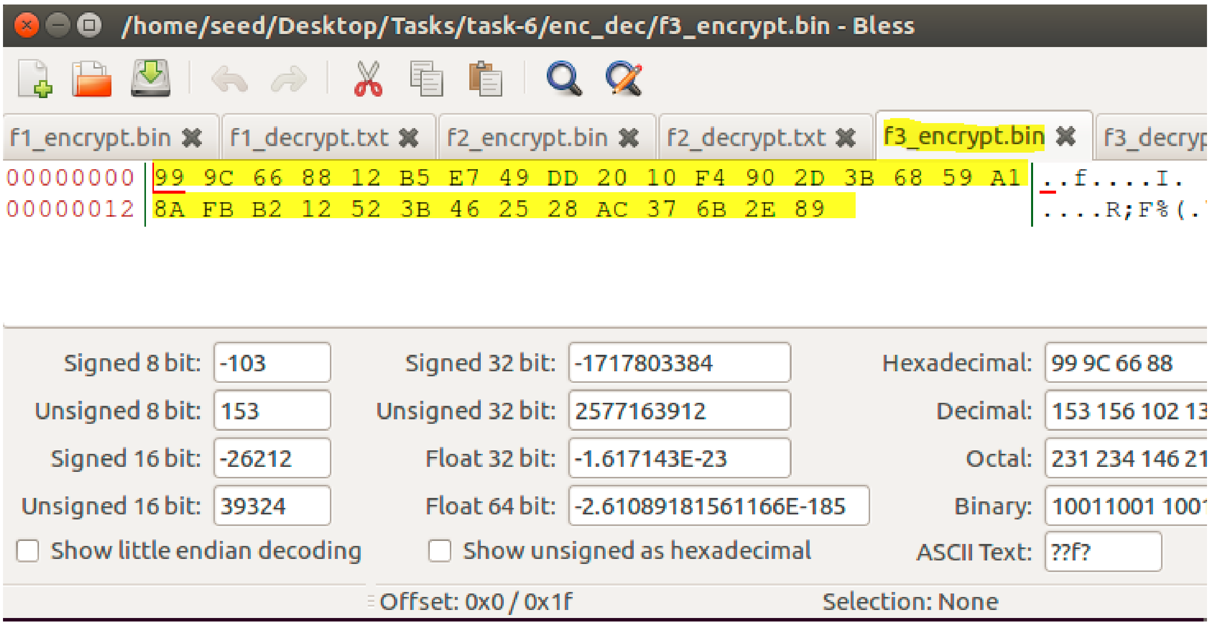
- f2_encrypt.bin



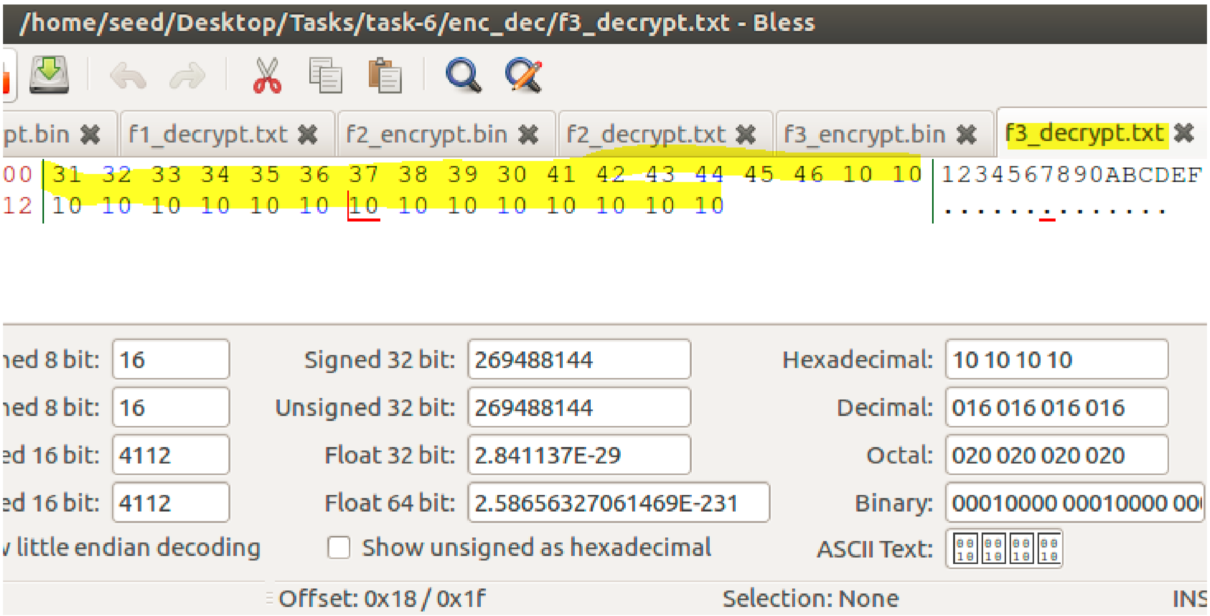
- f2_decrypt.txt



- **f3_encrypt.bin**



- **f3_decrypt.txt**



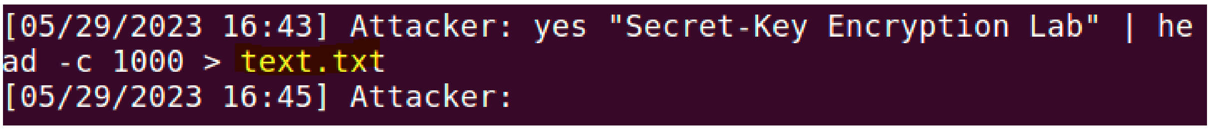
In conclusion, as we can see. To f1_decrypt.txt added 11 times “0b” which in decimal is 11. To f2_decrypt.txt added 6 times “06” which in decimal is 6. And the last file f3_decrypt.txt added 16 times “10” that in decimal is 16.

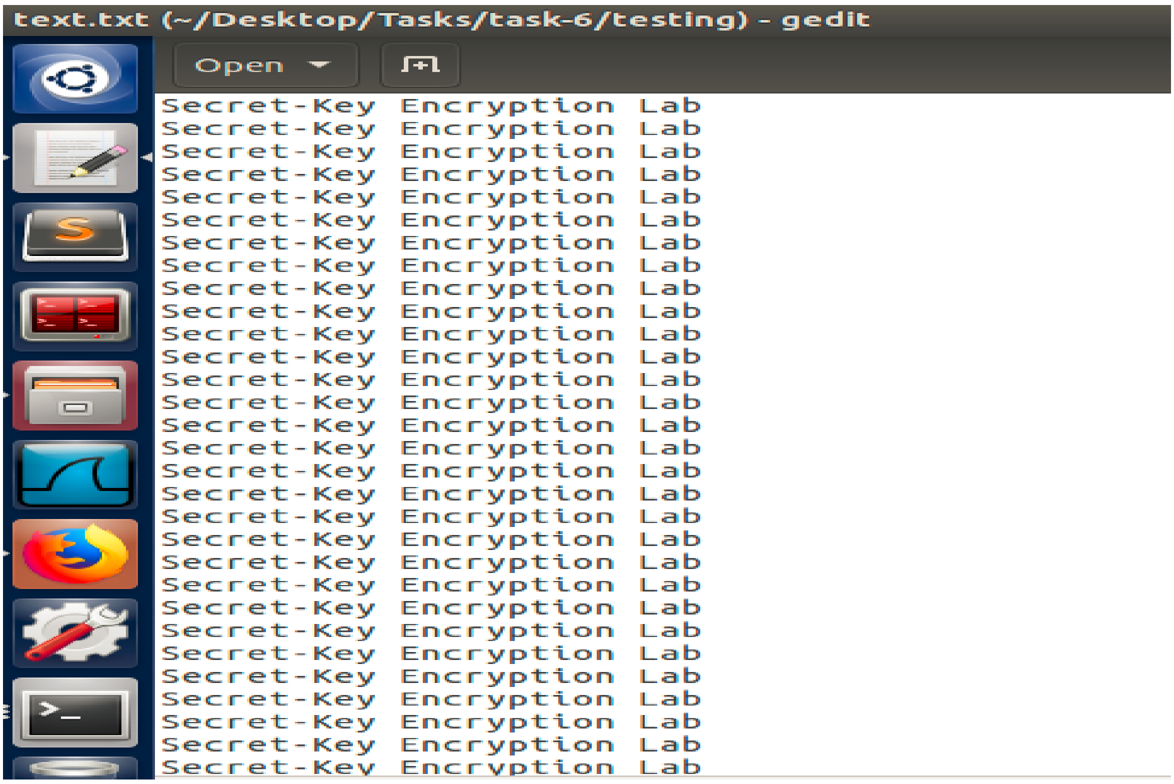
So we understand that the value of each padding byte is equal to the number of padding bytes added.

<u>Task 5: Error Propagation – Corrupted Cipher Text</u>

In this task we are going to create a text file that is at least 1000 bytes. This text file will be corrupted by us and we will see how it affects the decryption.

1) Create 1k text file

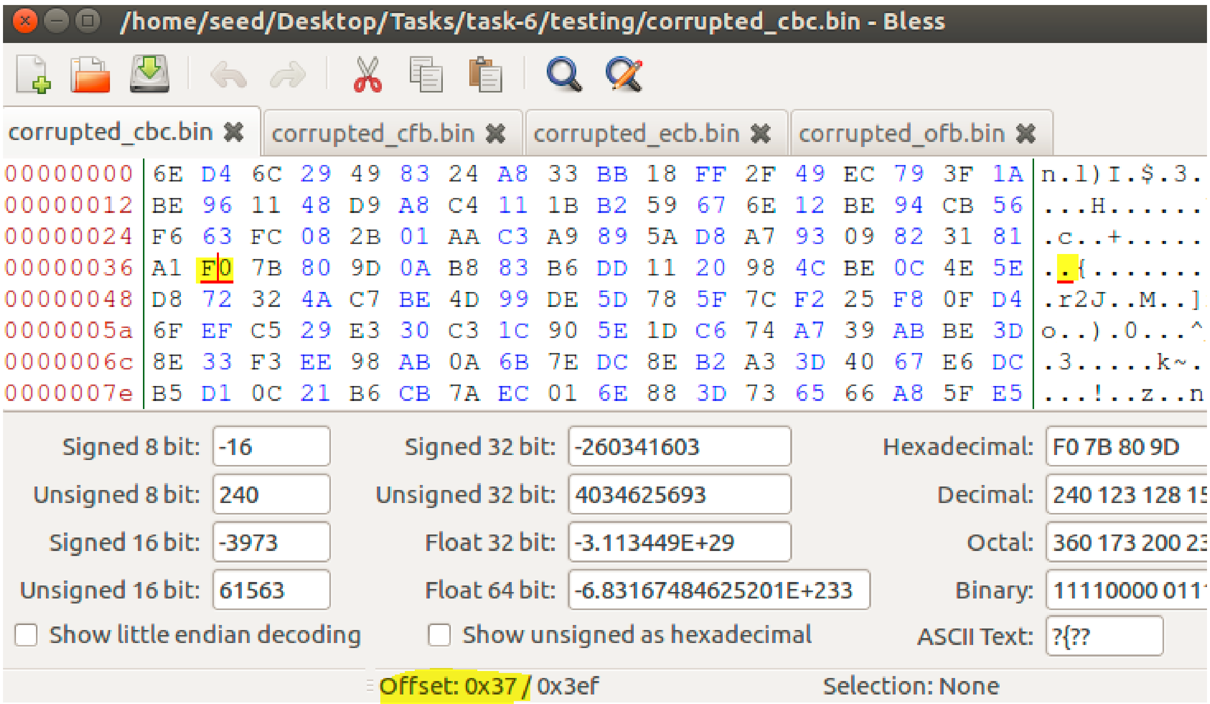




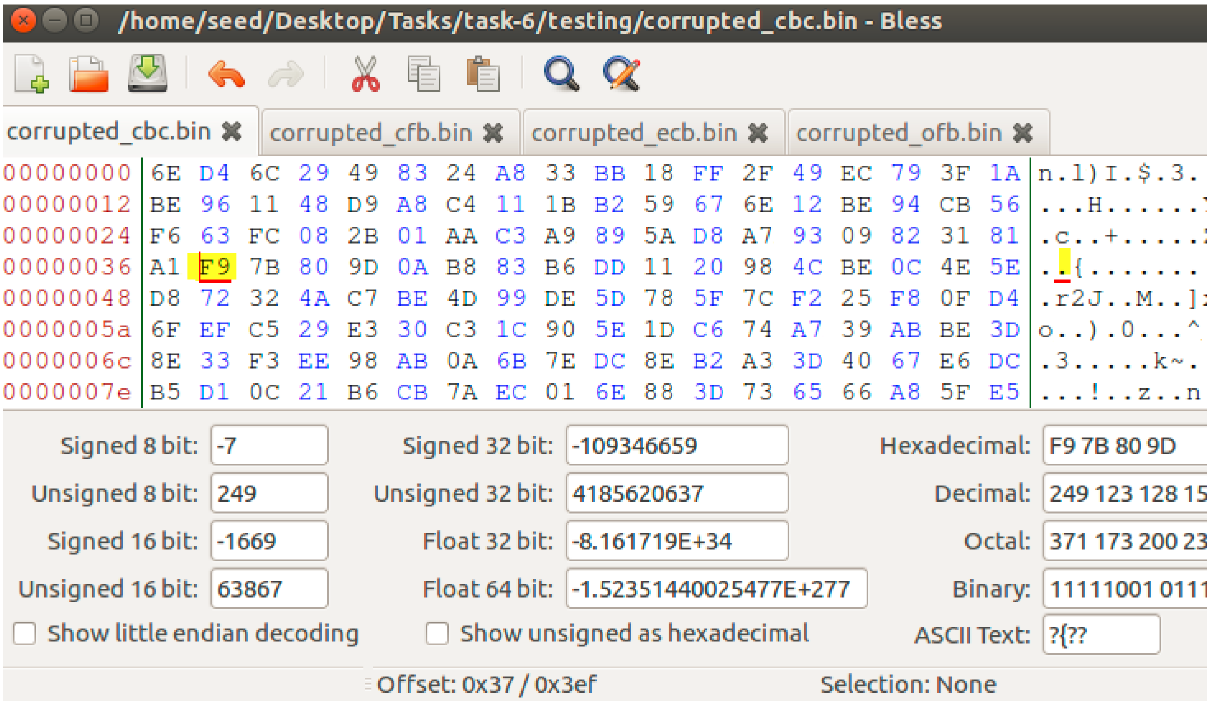
2) Now we will encrypt the file with ECB,CBC,CFB and OFB modes.

```
[05/29/2023 15:43] Attacker: openssl aes-128-ecb -in output.txt -out corrupted_ecb.bin -K 4fc74c475303d5d365cd82f995c99a90
[05/29/2023 15:43] Attacker: openssl aes-128-cbc -in output.txt -out corrupted_cbc.bin -K 4fc74c475303d5d365cd82f995c99a90 -iv 6a5d3af22d40920c592b73d0d2f9ede1
[05/29/2023 15:45] Attacker: openssl aes-128-cfb -in output.txt -out corrupted_cfb.bin -K 4fc74c475303d5d365cd82f995c99a90 -iv 6a5d3af22d40920c592b73d0d2f9ede1
[05/29/2023 15:46] Attacker: openssl aes-128-ofb -in output.txt -out corrupted_ofb.bin -K 4fc74c475303d5d365cd82f995c99a90 -iv 6a5d3af22d40920c592b73d0d2f9ede1
[05/29/2023 15:47] Attacker:
```

3) Changing a single bit of the 55th byte in the encrypted file.



Changing the 55th bit “F0” to “F9” (offset 0x37 is 55 in decimal).

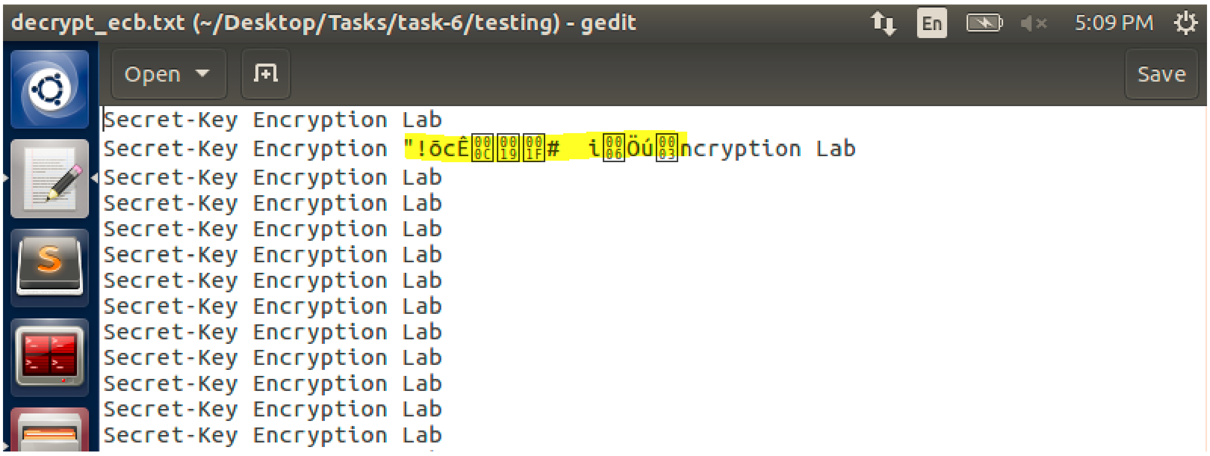


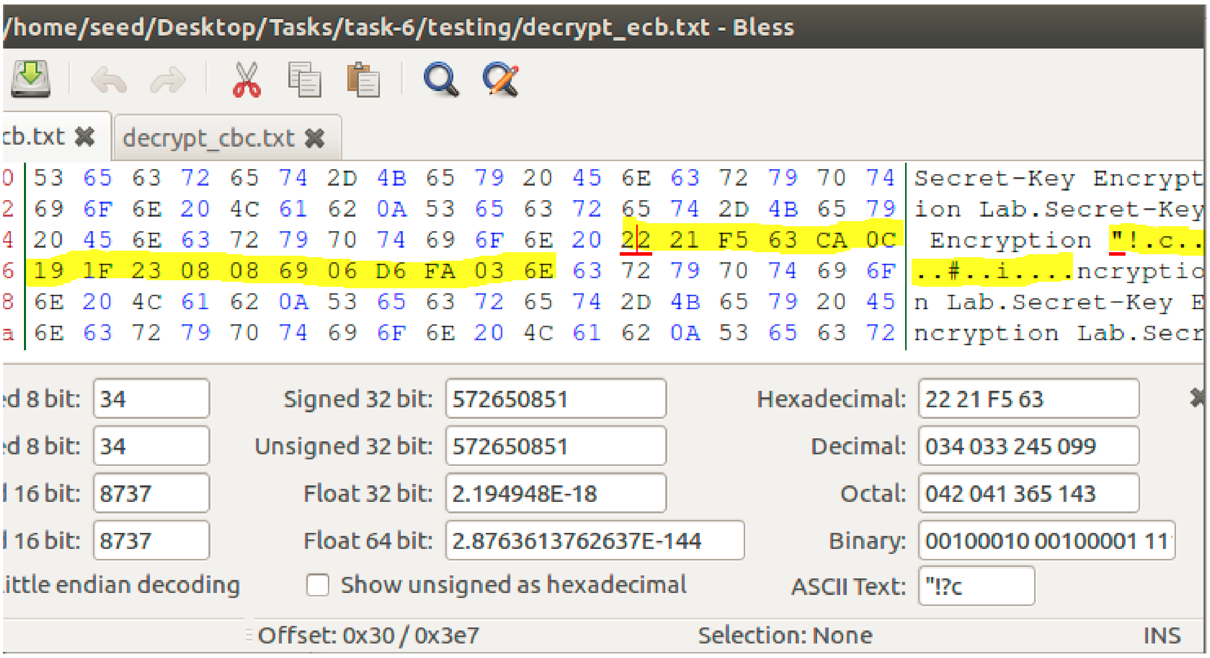
- We did it to the rest of the files, and changed the 55th bit (changed the right bit to “9”).

4) Now let’s decrypt all the encrypted files and let's see the results.

```
[05/29/2023 17:04] Attacker: openssl aes-128-ecb -d -in corrupted
_ecb.bin -out decrypt_ecb.txt -K 4fc74c475303d5d365cd82f995c99a90
[05/29/2023 17:04] Attacker: openssl aes-128-cbc -d -in corrupted
_cbc.bin -out decrypt_cbc.txt -K 4fc74c475303d5d365cd82f995c99a90
 -iv 6a5d3af22d40920c592b73d0d2f9ede1
[05/29/2023 17:05] Attacker: openssl aes-128-cfb -d -in corrupted
_cfb.bin -out decrypt_cfb.txt -K 4fc74c475303d5d365cd82f995c99a90
 -iv 6a5d3af22d40920c592b73d0d2f9ede1
[05/29/2023 17:05] Attacker: openssl aes-128-ofb -d -in corrupted
_ofb.bin -out decrypt_ofb.txt -K 4fc74c475303d5d365cd82f995c99a90
 -iv 6a5d3af22d40920c592b73d0d2f9ede1
[05/29/2023 17:06] Attacker:
```

- Let's see what changed following the change we made.
    - **ECB**

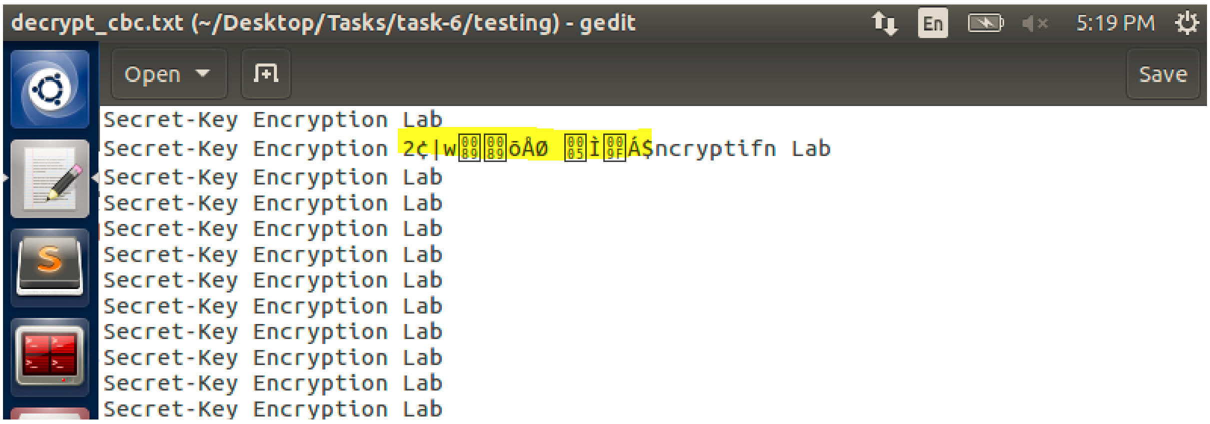


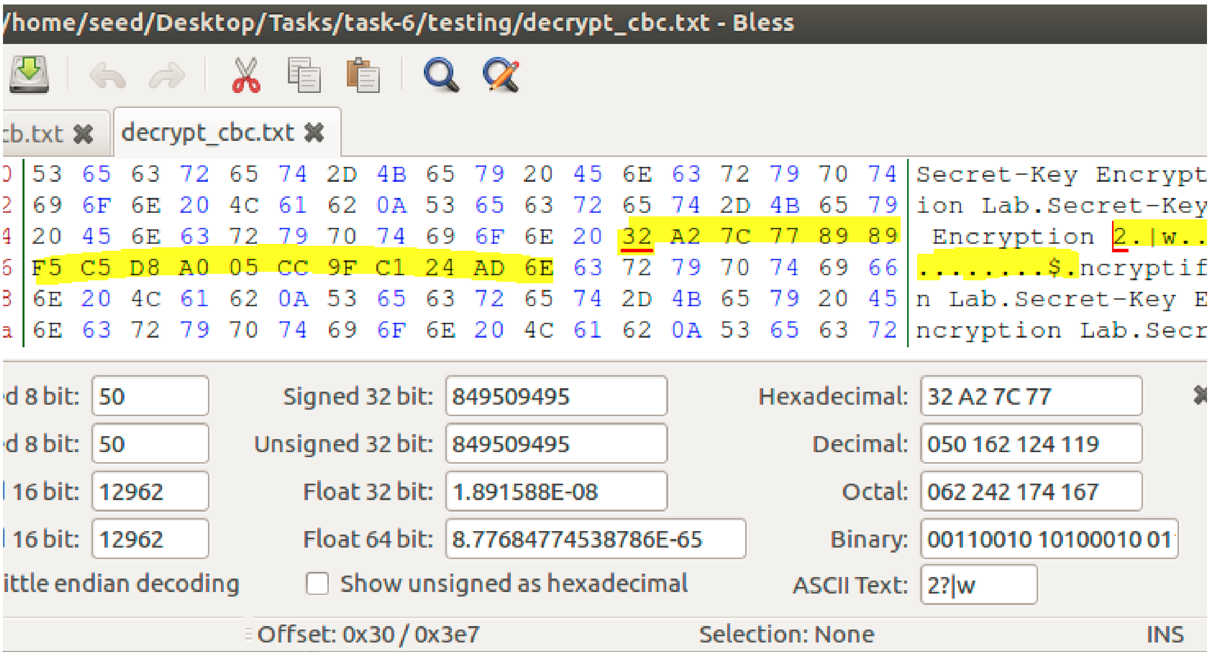


Since each block is encrypted independently using the same key, any modification to a single bit in the plaintext block will completely change the resulting ciphertext block. Moreover, the change will propagate throughout the entire block, affecting all bits.

Block size is 0x40 - 0x30 = 0x10 -> 16 in decimal. Means each block its size 0x10.

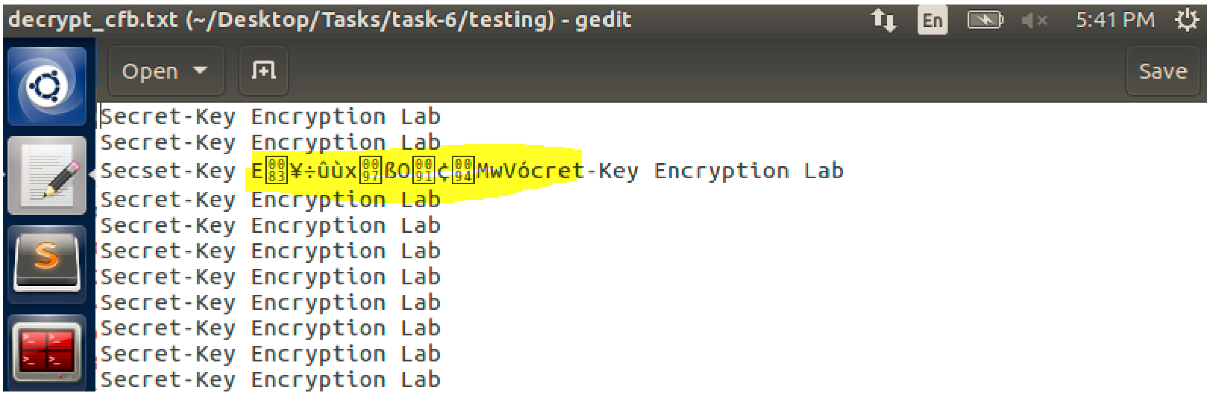
- **CBC**

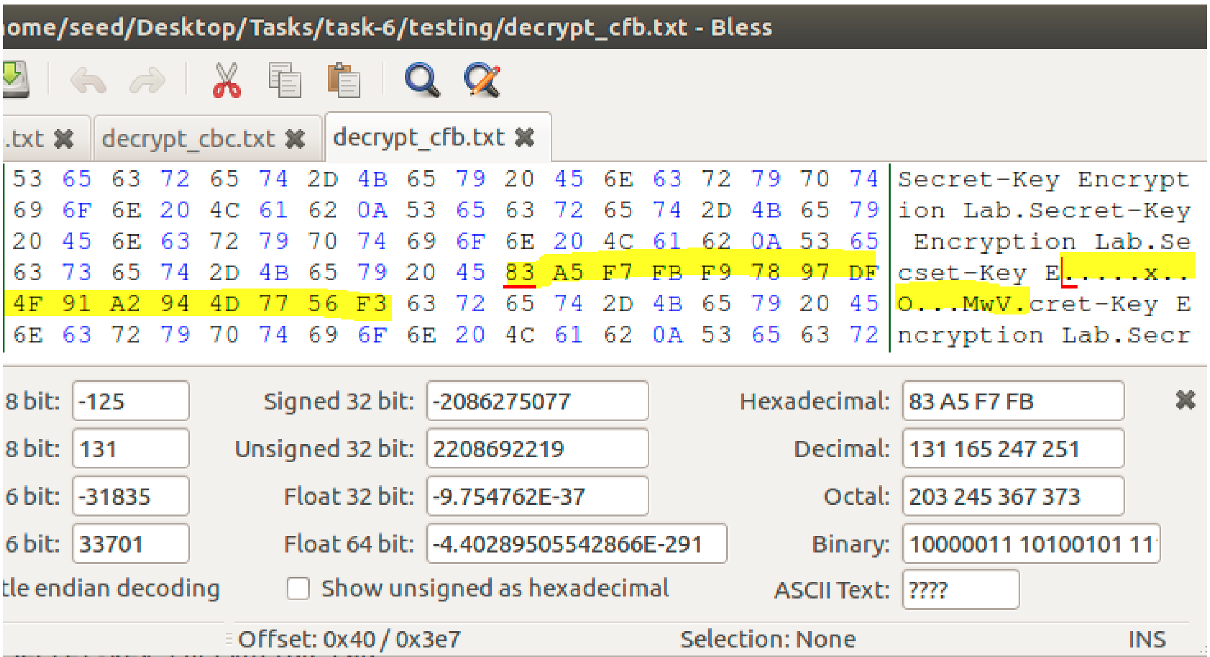




Here the corruption looks almost similar to ECB, but there is a small difference. The reason is if 1 bit is corrupted, then it affects its own block and the one after.

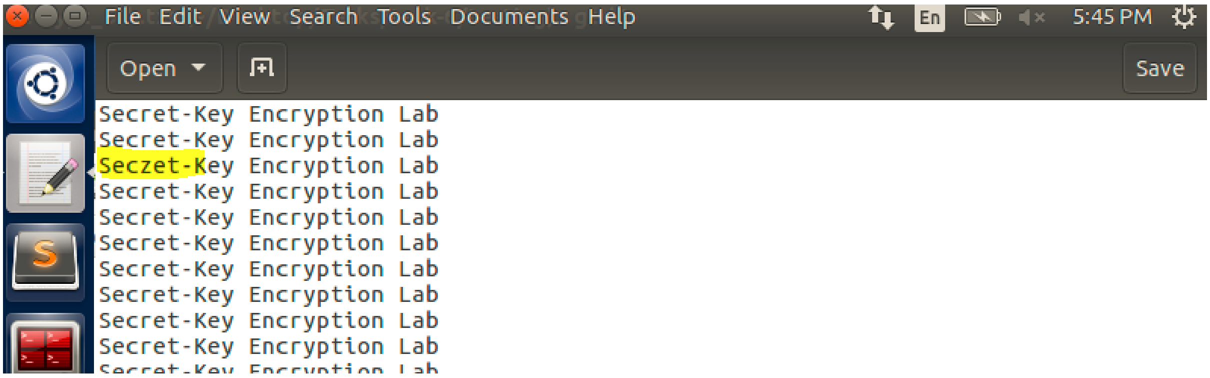
- **CFB**





The same effect as **CBC**.

- **OFB**



We can see that only one bit, the one we changed, has changed, this is because this mode encrypts each bit sequentially separately, and therefore does not affect the rest of the content.

For conclusion: In this task, we observed the impact of errors (faulty bits) in files after they were encrypted and how it affects their decryption. We learned that for encryption methods where bits or blocks are encrypted separately from the rest of the content, only those specific parts are affected. However, if encryption also incorporates information from previous blocks during the encryption process, then the impact spreads to those blocks as well, but not to the rest of the file.

Task 6: Initial Vector (IV) and Common Mistakes

The goal of the task is to help us understand the importance of selecting an appropriate Initial Vector (IV) in encryption. By exploring the potential issues that can arise if an IV is not chosen properly, we will gain insights into the impact of IV selection on the security of encrypted data. Through this task, we will develop an understanding of the significance of IVs and the considerations involved in their selection to ensure the overall security of encryption algorithms and modes.

Task 6.1

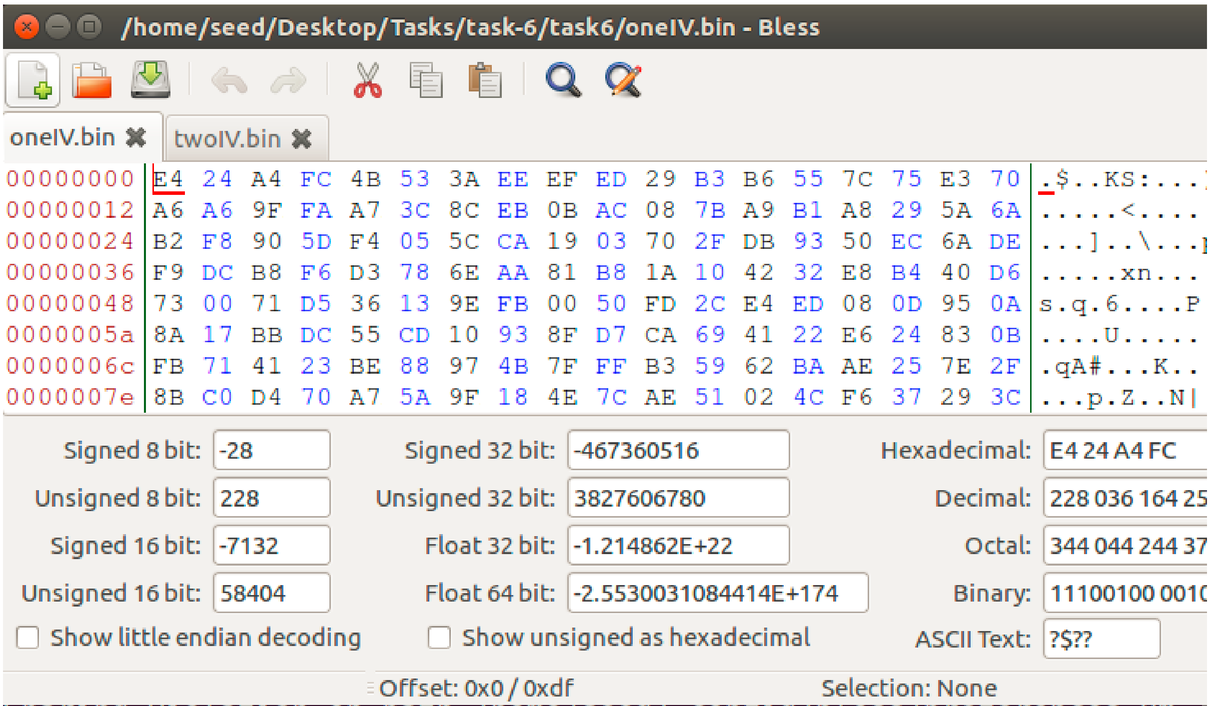
The goal of this task is to demonstrate the importance of uniqueness in IV selection by comparing the outcomes of encrypting the same plaintext with different IVs versus the same IV. The observation of different ciphertexts for different IVs reinforces the need for uniqueness to maintain the security of the encryption process.

- Let's create a text file and insert some text. Now encrypting the file using "aes-128-cbc" and "aes-128-cfb" with two different IVs.

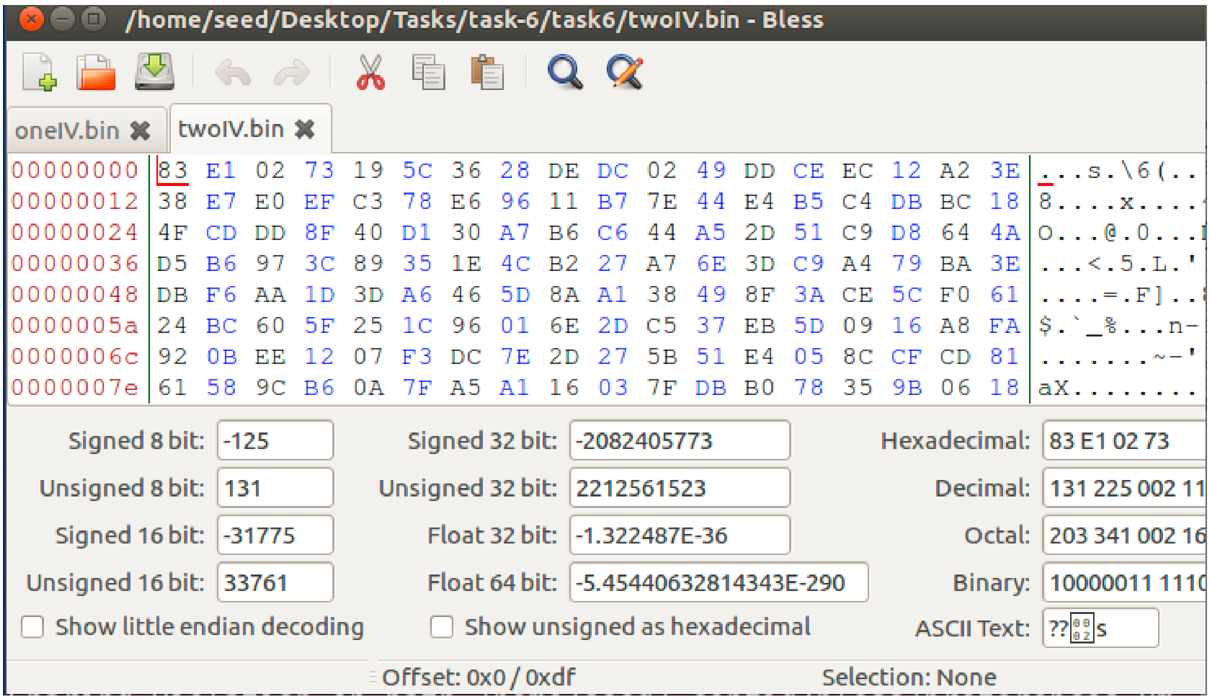
```
[05/30/2023 04:44] Attacker: openssl aes-128-cbc -e -in iv.txt -o
ut oneIV.bin -K da63eb8d56cc0a8fe364236c10e47c20 -iv 382f33a70626
20c496163754d5fc91ff
[05/30/2023 04:47] Attacker: openssl aes-128-cbc -e -in iv.txt -o
ut twoIV.bin -K da63eb8d56cc0a8fe364236c10e47c20 -iv 358fd9ece852
6dc4dbd15db6078c8b24
[05/30/2023 04:47] Attacker: 
```

We created two files, "oneIV.bin" with IV= "**382f33a7062620c496163754d5fc91ff**" and the second IV "**358fd9ece8526dc4dbd15db6078c8b24**"

- Using “bless” let’s see what's inside the files
    - File: “oneIV.bin”



- File: “twoIV.bin”



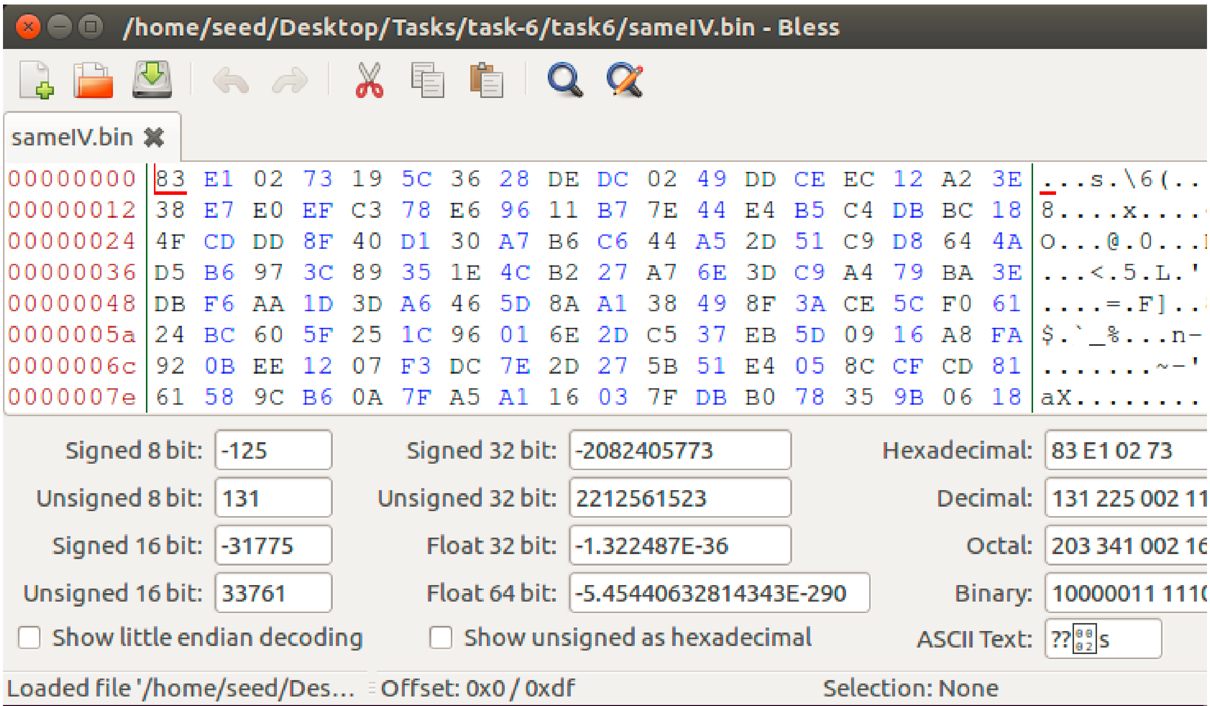
Two files were encrypted with the same key but different IV. So we can see the difference between two encryptions. The data is absolutely different.

- Now let's encrypt again the same file with one of the IV above.

```
[05/30/2023 05:04] Attacker: openssl aes-128-cbc -e -in
 iv.txt -out sameIV.bin -K da63eb8d56cc0a8fe364236c10e4
7c20 -iv 358fd9ece8526dc4dbd15db6078c8b24
[05/30/2023 05:04] Attacker: 
```

Same IV as file "twoIV.bin".

- Let's open text file "sameIV.bin" using "bless"



We can see that the encryption is exactly the same as the file "twoIV.bin".

In conclusion: When encrypting with the same key and IV, you will obtain the same ciphertext for the same plaintext. This is because the IV is used to initialize the encryption process and acts as the first block of input to the encryption algorithm.

In CBC (Cipher Block Chaining) mode, each plaintext block is XORed with the previous ciphertext block before encryption. The IV serves as the "previous ciphertext block" for the first plaintext block. By using the same IV, the XOR operation produces the same result, resulting in the same ciphertext.

However, when using a different IV for each encryption, the XOR operation produces different results, leading to different ciphertexts even for the same plaintext. The IV introduces randomness and ensures that even if the plaintext is the same, the encryption process is different due to the different IV values.

In summary, when using the same key and IV in AES-128-CBC encryption, you obtain the same encryption result. Different IVs produce different encryptions, ensuring the security and uniqueness of the ciphertext.

Task 6.2. Common Mistake: Use the Same IV

The goal of this experiment is to highlight the potential security vulnerability when using the same IV in certain encryption modes, specifically Output Feedback (OFB) mode. The experiment examines whether an attacker, having access to a known plaintext (P1) and its corresponding ciphertext (C1), can decrypt other encrypted messages when the IV remains the same.

- To do this kind of experiment, we had to create a python file and write some code.

```
common_mistake.py (~/Desktop/Tasks/task-6/task6-2) - gedit
import binascii

C1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
C2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"
P1 = "This is a known message!"

# Convert hex strings to byte arrays
C1_bytes = binascii.unhexlify(C1)
C2_bytes = binascii.unhexlify(C2)

# XOR C1 and C2 to obtain the keystream
keystream = ''.join(chr(ord(c1_byte) ^ ord(c2_byte)) for c1_byte, c2_byte in zip
(C1_bytes, C2_bytes))

# XOR the keystream with P1 to approximate P2
P2 = ''.join(chr(ord(p1_byte) ^ ord(ks_byte)) for p1_byte, ks_byte in zip(P1,
keystream))

print("Approximate P2: ", P2)
```

- Let's break it down:
  - First we convert the hex string into byte arrays.
  - Then we XOR C1 and C2 in order to obtain the keystream.
  - After that, we XOR the keystream with P1 to obtain P2.
  - Finally print P2

- Let's run the code

```
[05/30/2023 06:02] Attacker: python common_mistake.py
('Approximate P2: ', 'Order: Launch a missile!')
[05/30/2023 06:03] Attacker:
```

- Replace OFB in this experiment with CFB (Cipher Feedback), how much of `P2` can be revealed?

It can be seen that compared to OFB, CFB uses for each block, starting from the second block, the ciphertext of the previous block together with the key to perform XOR with the original content. In other words, each time there is a different value except for the first block, which can be found using the previous method since the IV and the key remain the same. In other words, if there is sensitive content in the first block, the attacker will be able to easily find its value.

In conclusion: When you XOR the keystream with P1, it essentially applies the same XOR operation to each corresponding byte in the keystream and P1. XORing a byte with itself cancels out the effect and results in zero. XORing a byte with any other value yields a different value.

In the given scenario, the keystream is generated by XORing the corresponding bytes of C1 and C2. Since C2 is the result of encrypting P2, XORing the keystream with P1 will "cancel out" the effect of the keystream on C1 and reveal the content of P2. This is based on the property of XOR operation, where `A ^ B ^ B = A`.

In other words, XORing the keystream with P1 essentially "undoes" the encryption applied to P1 using the keystream, resulting in the approximation of P2.

Task 6.3. Common Mistake: Use a Predictable IV

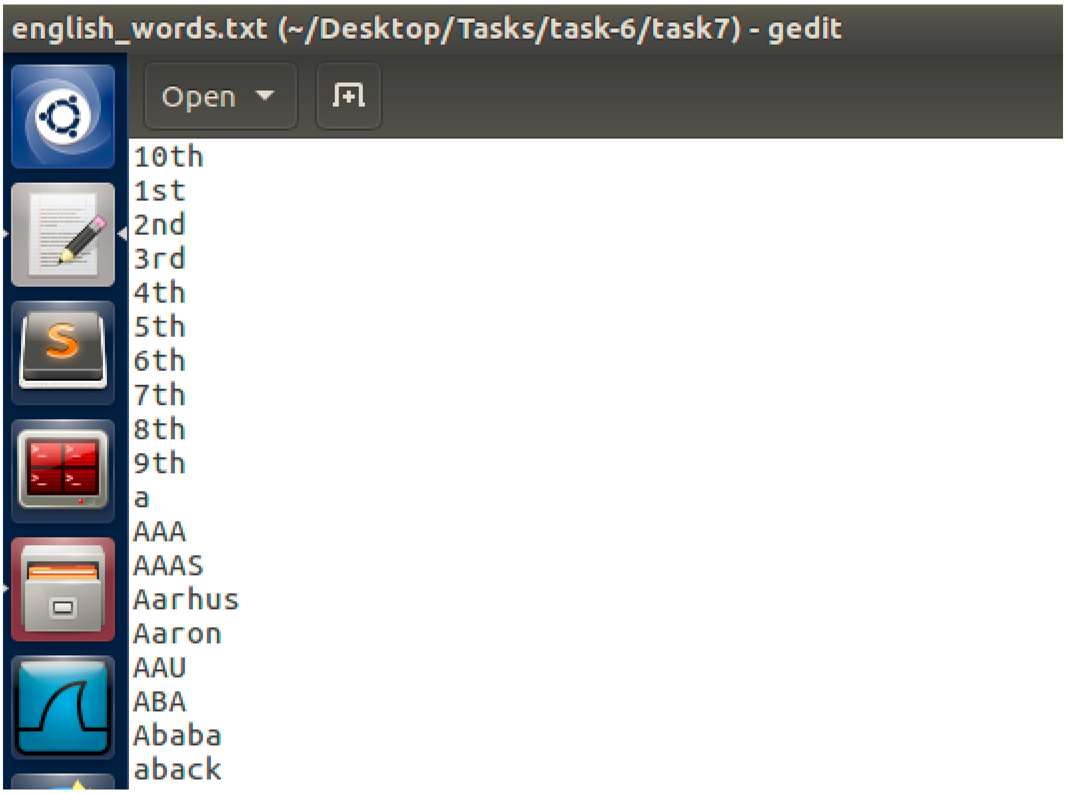
The goal of this task is to understand the importance of using unpredictable IVs (Initialization Vectors) in encryption schemes. It explores the scenario where an attacker, Eve, has access to the ciphertext and the IV used by Bob to encrypt a message. Although Eve cannot decipher the actual content of the message due to the strength of the AES encryption algorithm, she can predict the next IV that Bob will use because he follows a predictable pattern.

By examining this scenario, the task aims to demonstrate the potential security risks associated with using predictable IVs. It highlights the importance of generating IVs randomly and ensuring their unpredictability to enhance the security of encrypted messages.

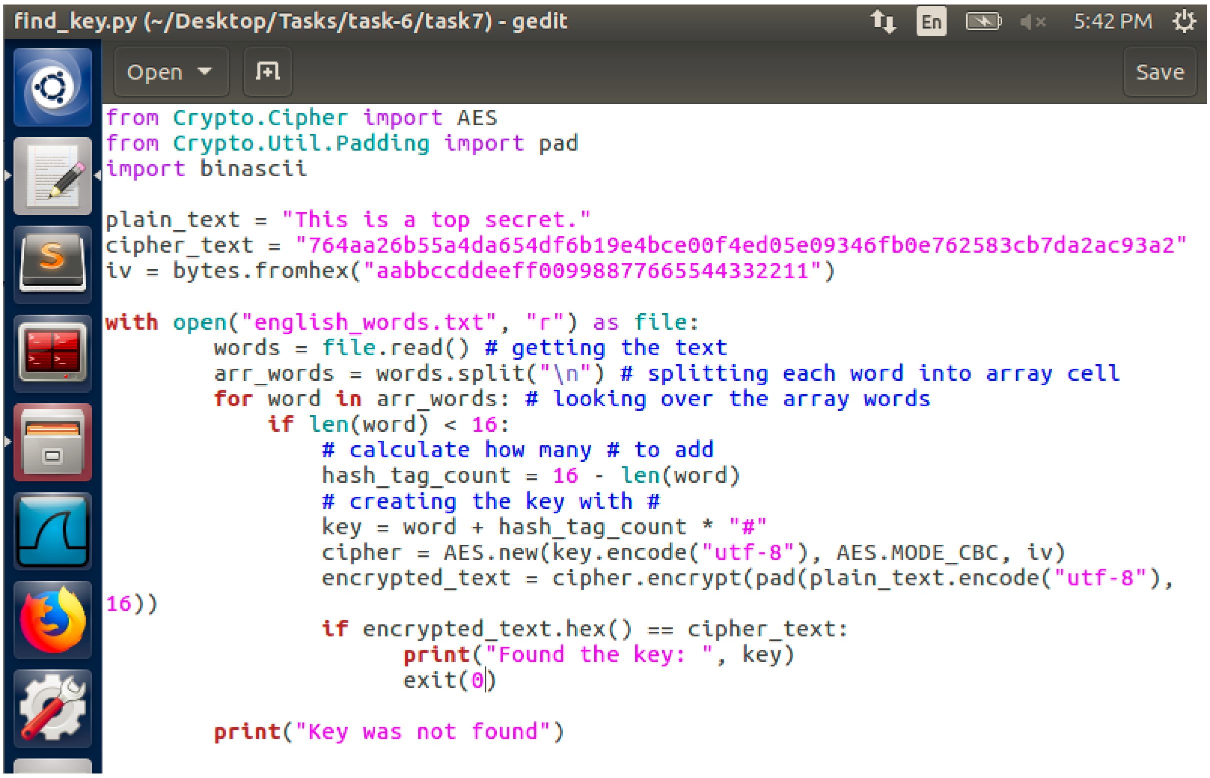
Task 7: Programming using the Crypto Library

The goal is to write a program to find out the encryption key using IV, plaintext and ciphertext that are given for the task.

- First let's download the file with english words



- Then let’s write a script that finds the key by the given information.



find_key.py (~/Desktop/Tasks/task-6/task7) - gedit

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import binascii

plain_text = "This is a top secret."
cipher_text = "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
iv = bytes.fromhex("aabbccddeeff00998877665544332211")

with open("english_words.txt", "r") as file:
        words = file.read() # getting the text
        arr_words = words.split("\n") # splitting each word into array cell
        for word in arr_words: # looking over the array words
            if len(word) < 16:
                # calculate how many # to add
                hash_tag_count = 16 - len(word)
                # creating the key with #
                key = word + hash_tag_count * "#"
                cipher = AES.new(key.encode("utf-8"), AES.MODE_CBC, iv)
                encrypted_text = cipher.encrypt(pad(plain_text.encode("utf-8"),
16))

                if encrypted_text.hex() == cipher_text:
                      print("Found the key: ", key)
                      exit(0)

        print("Key was not found")
```

- Let’s run the script

```
[05/30/2023 17:36] Attacker: python3 find_key.py
Found the key:  Syracuse########
[05/30/2023 17:45] Attacker:
```

Example of a possible attack - KPA

A known plaintext attack (KPA) is a type of cryptographic attack where the attacker possesses knowledge of both the plaintext (original, unencrypted message) and its corresponding ciphertext (encrypted message). The goal of a known plaintext attack is to uncover or deduce information about the secret key or the encryption algorithm used.

In a known plaintext attack, the attacker typically collects a significant number of plaintext-ciphertext pairs, which are obtained through various means. This could involve intercepting encrypted communications, accessing encrypted files, or even exploiting system vulnerabilities to gain access to both the plaintext and ciphertext.

Using the known plaintext-ciphertext pairs, the attacker analyzes the patterns, correlations, and relationships between the plaintext and ciphertext. They look for any weaknesses or vulnerabilities that can be exploited to deduce information about the secret key. The attacker may employ statistical analysis, mathematical techniques, or other methods to infer properties of the encryption algorithm or the key.

The success of a known plaintext attack depends on several factors, including the strength of the encryption algorithm, the amount and quality of known plaintext-ciphertext pairs, and the resources and computational power available to the attacker. A robust encryption algorithm should ideally resist known plaintext attacks by ensuring that the ciphertext does not reveal any meaningful information about the key or the plaintext.

One notable known plaintext attack (KPA) in 2015 was the "Bar Mitzvah" attack on the RC4 encryption algorithm.

RC4 is a widely used stream cipher that has been in use for over 20 years, particularly in wireless protocols like WEP and WPA. In 2015, researchers from the Weizmann Institute of Science and the University of California discovered vulnerabilities in RC4 that could be exploited through known plaintext attacks.

The Bar Mitzvah attack took advantage of weak key scheduling in RC4 when used in certain SSL/TLS implementations. By analyzing a large number of encrypted connections and collecting known plaintext-ciphertext pairs, the researchers were able to determine the encryption key.

This attack was particularly concerning because it could be used to recover sensitive information, such as usernames, passwords, and other data transmitted over encrypted connections. It highlighted the need for deprecating the use of RC4 in cryptographic protocols due to its vulnerabilities.

In response to the Bar Mitzvah attack and previous vulnerabilities found in RC4, organizations and standards bodies recommended discontinuing the use of RC4 in favor of stronger encryption algorithms, such as AES (Advanced Encryption Standard). Major browser vendors, including Mozilla, Google, and Microsoft, implemented changes to their software to mitigate the risks associated with RC4.

The Bar Mitzvah attack demonstrated the importance of continuous evaluation and scrutiny of encryption algorithms, as even widely used ciphers can have vulnerabilities exposed over time. It contributed to the ongoing effort to phase out weaker encryption algorithms and adopt more robust and secure alternatives.