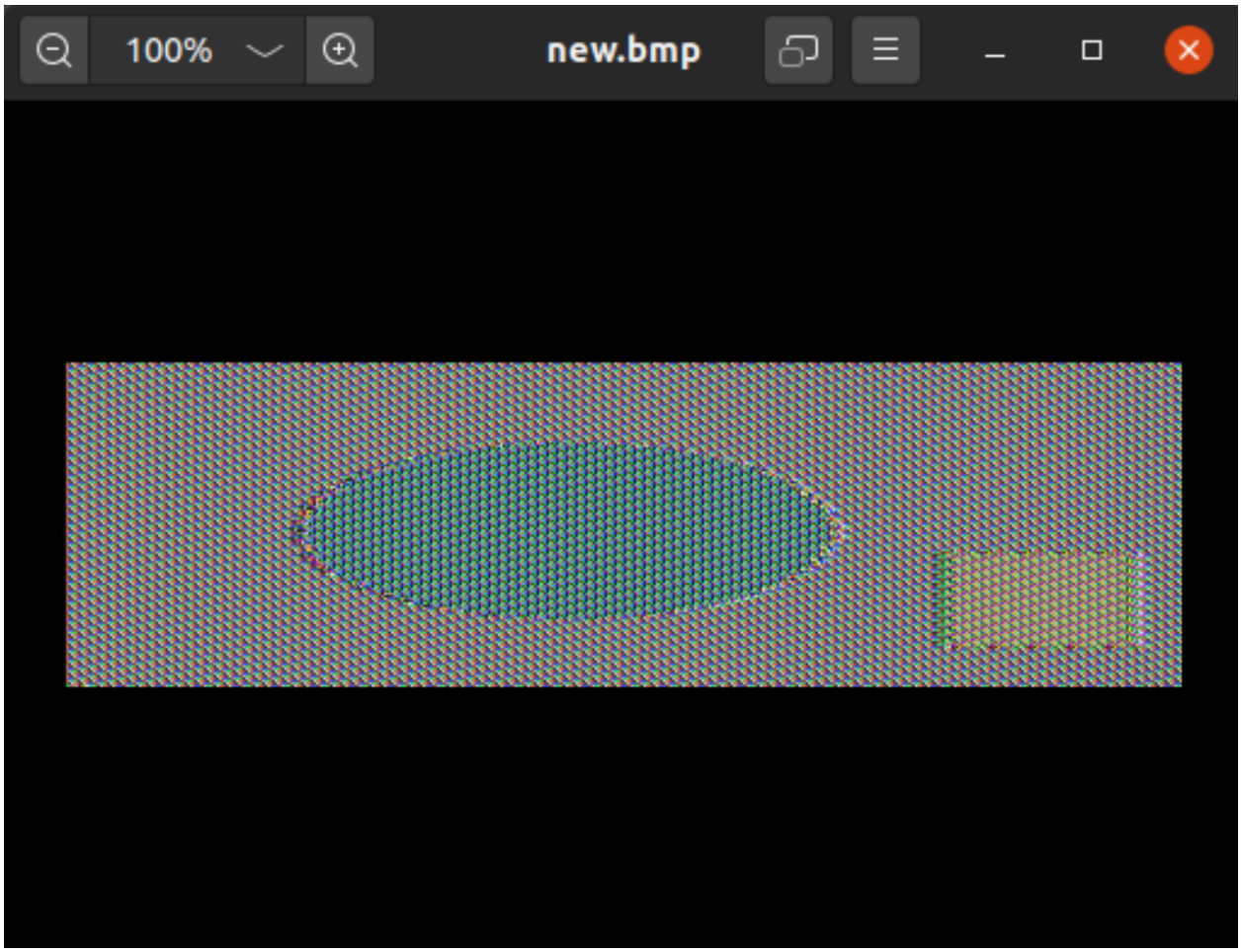
1. 42 mod 16 = 10 bytes in the last block. To make it a multiple of the block size, 6 bytes of padding are needed. The padding bytes will have the value of 6 or 06 in hex. So the padding will be 06 06 06 06 06 06. In the case of 64 bytes, since 64 is already a multiple of 16, the padding block will consist of 16 bytes, each having the value 16 or 10 in hex. So, the padding will be 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10.
2. In CFB he should be able to recover all but the 2nd bit in the 3rd block, nothing in the 4th block and everything after should be fine. In CTR, only the 2nd bit of the 3rd block will be affected, then everything after should be fine.
3. The main reasons that this is not a good hashing function is that it is fairly predictable and it does not have good enough collision resistance making it an unreliable hashing function.
4. It is better to keep the salt stored in plaintext because it will make it easier for the system to verify passwords. The reason it is okay to leave it in plaintext is because it will not give the attacker any meaningful information. The best way to improve security for the shadow file is to have a high quality hashing function instead.
5. I do not agree with this method and it would not be secure. If you send the hashed password to the server, then it may get intercepted and then the attacker can use it like a regular password to verify authenticity.
6. It is not a waste of time and is done for good reason. With this type of design it creates a resistance to some attacks like brute force attacks without really affecting the user experience.
7. If you know M, you can use SHA256(K || M) as a starting point. Then, you can create X by adding M and new content (T). This gives SHA256(K || M || T). X would be (M || T), so you can calculate the hash as SHA256(K || X) without needing to know K.
8. Yes, you do need to have an idea about the length of the key in order to correctly compute the padding.
9. For the padding, the length in bits is determined by 49 * 8 = 392 bits. The padding will start with a \x80 and be followed by some \x00s and will end with the 64 bit length of K:M in bits. Then to hash K:M:N, we need to append the padding P to K:M and then add the extra content of N.
10. First, Bob needs to send some message to Alice and it can be anything. Alice then needs to calculate the HMAC of the message Bob sent using the secret number K. After Bob receives the message, he can also calculate the HMAC of his message using K. If both of their HMAC values match, they can confirm that they are talking to their intended recipient.
11. This is a framework to hash functions, it breaks up messages into different blocks and feeds it into a compression function along with an IV. As it moves along, the output of one block will become the input for the next block. The final product will be the hash of the message.
12.

Section 2

1. a)

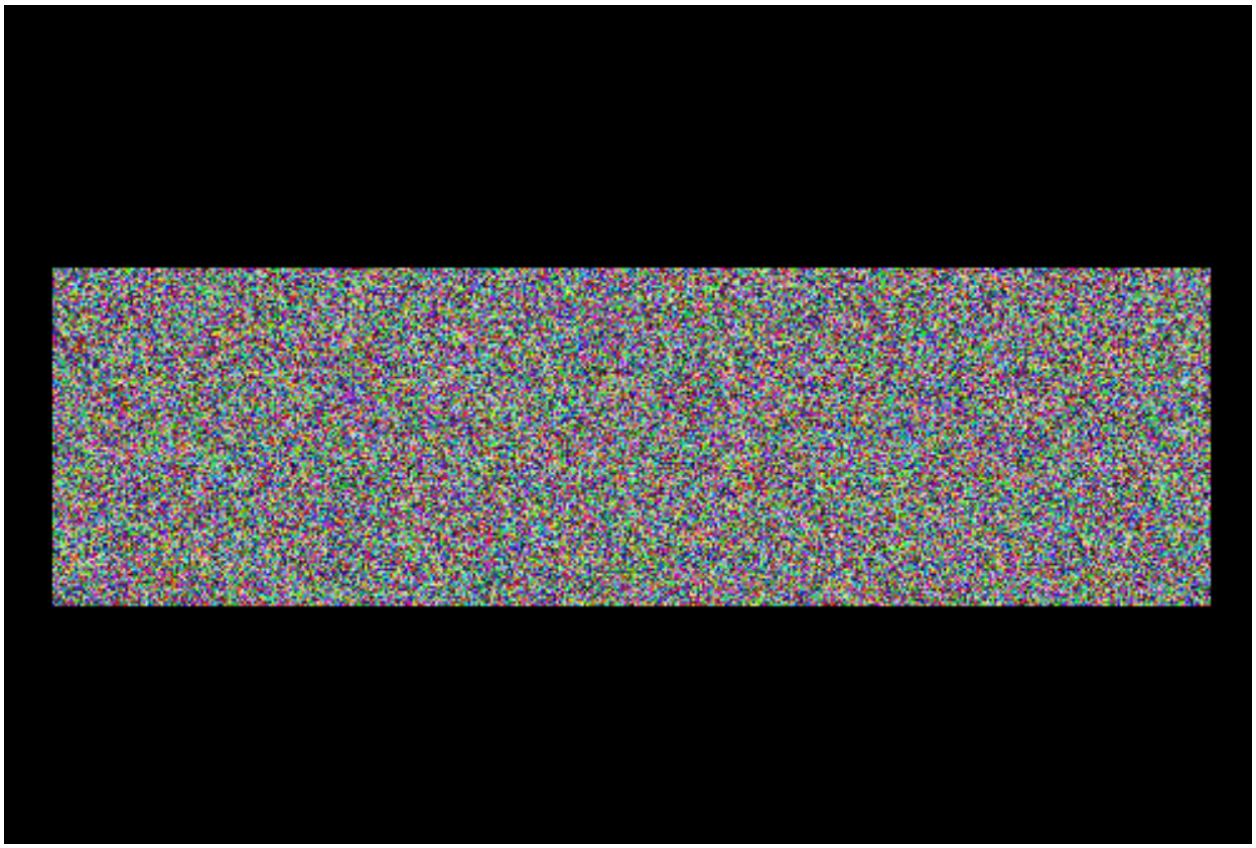
```
12/08/24]seed@VM:~/.../Assignment4_Files$ eog pic.bmp
12/08/24]seed@VM:~/.../Assignment4_Files$ head -c 54 pic_original.bmp > header
12/08/24]seed@VM:~/.../Assignment4_Files$ cat header
M66(X[12/08/24]seed@VM:~/.../Assignment4_Files$ tail -c +55 pic.bmp > body
12/08/24]seed@VM:~/.../Assignment4_Files$ cat header body > new.bmp
```

Here is the original Image.



Here is the image in ECB mode.

From what we can see, the general shape from the image is still clear, but things like color and “texture” are very different. And there is also a striping effect across the image. Also, the edges of the shape have a depth to them. Overall, you can still tell what the original image was supposed to be, but the finer details are encrypted. The reason why is ECB just reuses the same input into the encryption function and no extra input added, so it will look relatively similar to the original image.



Here is the image in CBC mode.

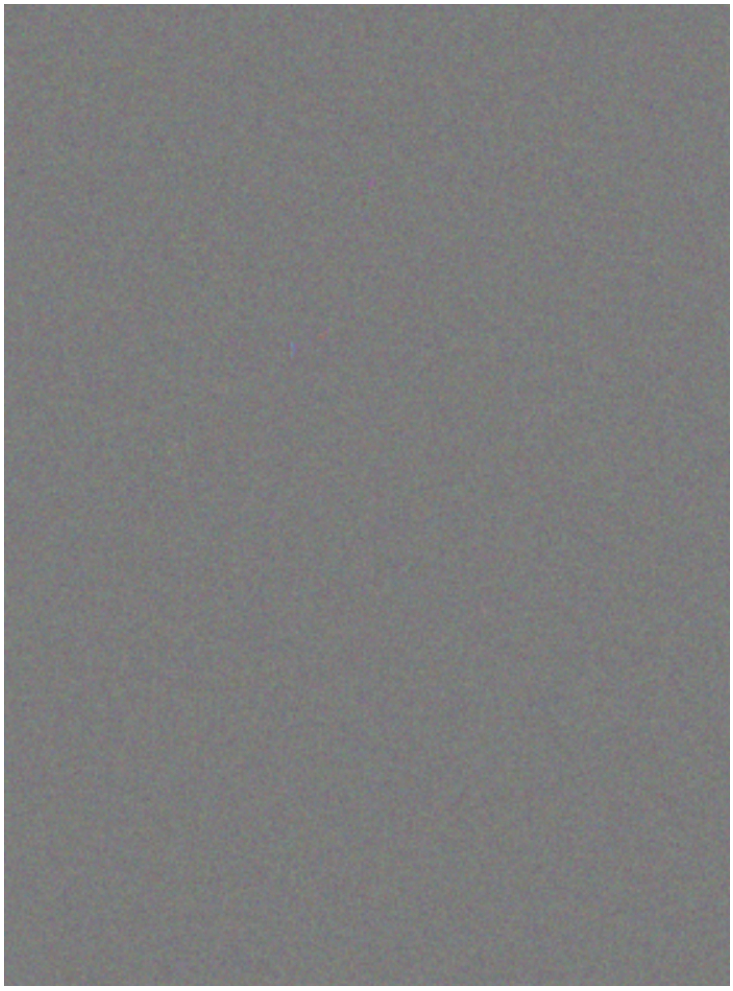
Here, we can see a much different image. It basically looks like garbled specs of color or TV static. We can get no idea of what this image was before encryption, no shape, color or anything. This seems to be a more effective encryption method if conveying no information whatsoever is necessary. This will feed the previous output of the first block into the next and so on, which will significantly randomize how the image is encrypted and produce an image like this.

2) a) Here is the original image

Frank.

```
[12/08/24]seed@VM:~/.../Assignment4_Files$ openssl enc -aes-128-ecb -e -in frank.
mp -out pic.enc
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[12/08/24]seed@VM:~/.../Assignment4_Files$ head -c 54 frank.bmp > header
[12/08/24]seed@VM:~/.../Assignment4_Files$ tail -c +55 pic.bmp > body
[12/08/24]seed@VM:~/.../Assignment4_Files$ cat header body > out.bmp
[12/08/24]seed@VM:~/.../Assignment4_Files$ eog out.bmp
[12/08/24]seed@VM:~/.../Assignment4_Files$ cp pic.bmp
cp: missing destination file operand after 'pic.bmp'
Try 'cp --help' for more information.
[12/08/24]seed@VM:~/.../Assignment4_Files$ cp pic.enc pic.bmp
[12/08/24]seed@VM:~/.../Assignment4_Files$ tail -c +55 pic.bmp > body
[12/08/24]seed@VM:~/.../Assignment4_Files$ cat header body > out.bmp
[12/08/24]seed@VM:~/.../Assignment4_Files$ eog out.bmp
```

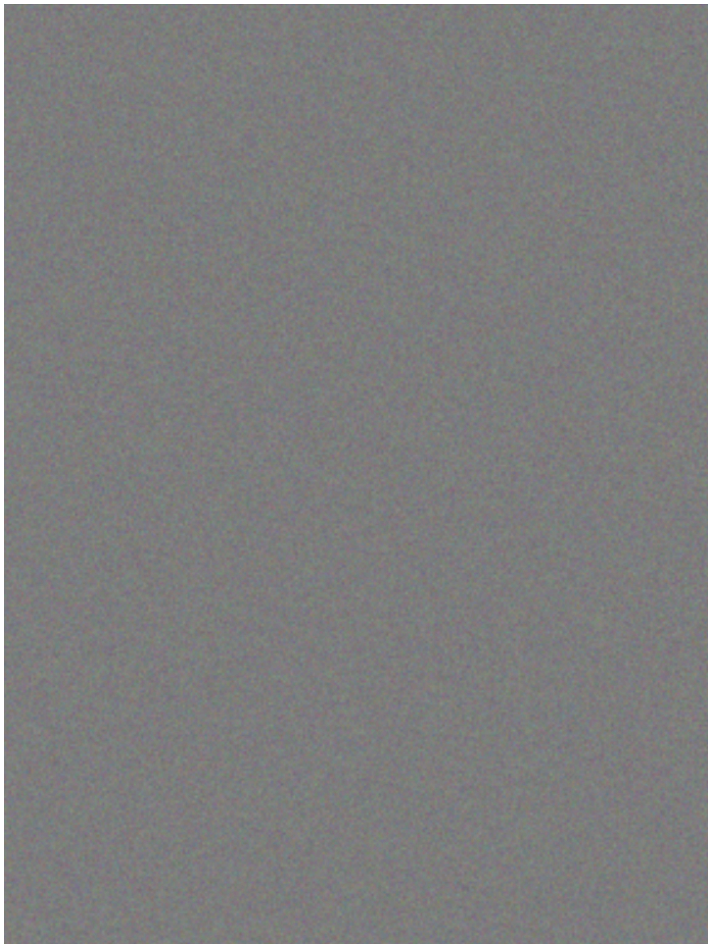
Here, we set up the encryption for ECB.



This was the resulting image. This was different from the first experiment and the encryption seems to be a lot more effective. This is likely because it includes a lot of different data and detail compared to pic_original. This allows it to have a more comprehensive encryption output.

```
12/08/24]seed@VM:~/.../Assignment4_Files$ openssl enc -aes-128-cbc -e -in frank.
p -out pic.enc
nter aes-128-cbc encryption password:
erifying - enter aes-128-cbc encryption password:
** WARNING : deprecated key derivation used.
sing -iter or -pbkdf2 would be better.
12/08/24]seed@VM:~/.../Assignment4_Files$ cp pic.enc pic.bmp
12/08/24]seed@VM:~/.../Assignment4_Files$ tail -c +55 pic.bmp > body
12/08/24]seed@VM:~/.../Assignment4_Files$ cat header body > out.bmp
12/08/24]seed@VM:~/.../Assignment4_Files$ eog out.bmp
```

Here is the setup for CBC.



Here is the output. It looks exactly the same as ECB, this was expected because of what we saw from the first experiment. Except now, ECB and CBC produced effectively the same encryption image. This shows they both have a good method to encrypt, but to be mindful of what kind of content in an image you want to encrypt to ensure the best result.