**Saishnu Ramesh Kumar (300758706)**

**CSCI 191T (Assignment 4) – XSS, Secret Key Encryption, and One-Way Hash Functions**

**Section 1:**

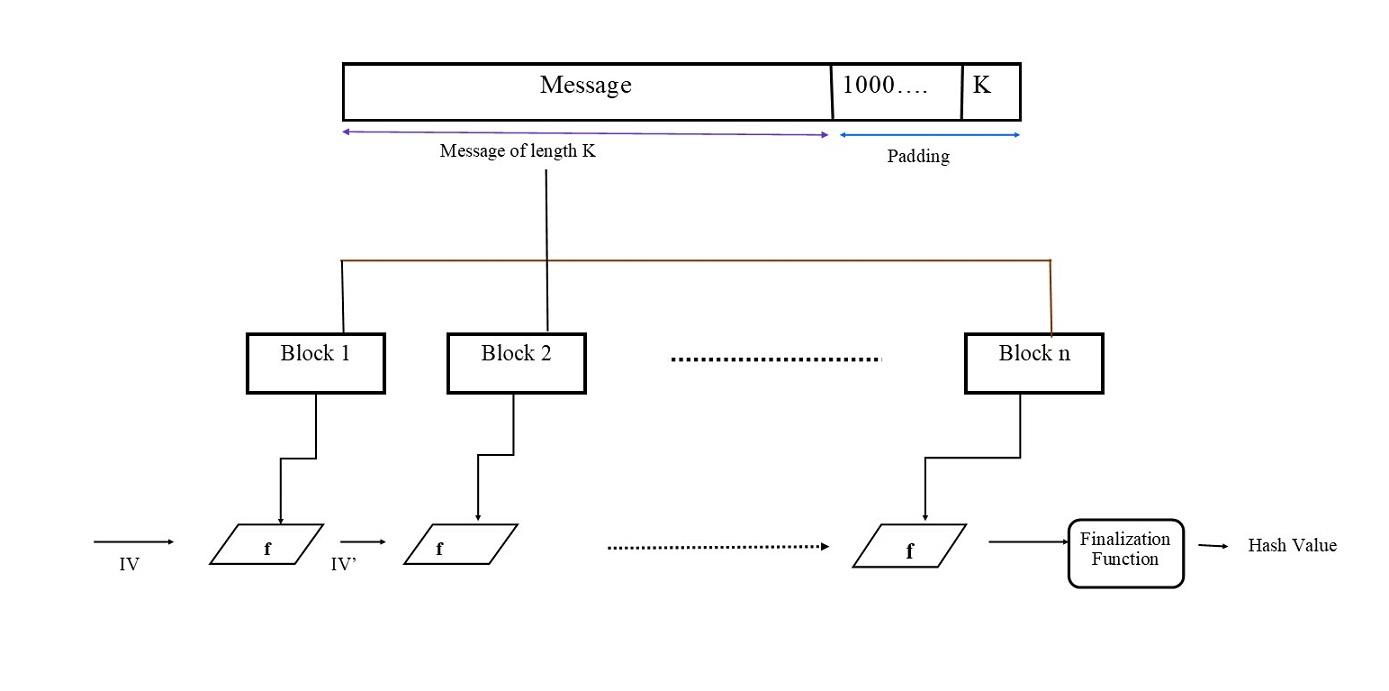
1. Potentially considering if the logic is perfect, yes. The main cause of XSS attacks is when the attacker embeds malicious code, mainly JavaScript, into the victim’s browser, therefore developing a filtering system can help prevent any JavaScript code from being both uploaded and downloaded.
2. The CSP is effective in defeating Cross-Site Scripting attacks because it prevents malicious code from being injected into a site. It can prevent these forms of threats because it only limits the sources of content that is allowed to be displayed. The downside of this approach is that it can block actual safe scripts from running which are not owned by the CSP and may cause the site to not work properly.
3. The padding data will be 77777. If the message has 64 bytes, it will be easier to run the program because the 64 bytes padding memory will be increased and it will allow the message to run faster and it is the more standard for the system to compute.
4. Bob will still be able to receive the entire plaintext because the plaintext in both the OFB and CTR models do not interfere with the encryption mode and in both instances, they are just added to the XOR statement to then obtain the ciphertext. If it was on the decryption mode, Bob will be able to recover all the bits except for the 2nd bit and this applies to both models as well.
5. For a one-way hash function, this function is not a good method because it will be difficult to find x if the database becomes big, in this case out of 10000 and it may also cause possible collision among data, hence having a collision resistant to reduce/prevent that.
6. Yes, it is better to save salt in the plaintext because the salt is already provided with security as it must be added with plaintext. They also are easier to save, and it will be a more difficult time for the attack to breach its security. Moreover, when using salt, the same input results in different hashes.
7. Yes, I do agree that it can improve the security because when the hash is sent, it is more encrypted and protected from an attacker potential stealing that data. For password verification, the user would need to tell a secret password which cannot be stored using plaintext. If the user were to login to their profile with their password, it would need to verify itself with the password that is stored on the server to see if they match. Nonetheless, the more secure way is by using salt as it ensures that the hashes are always distinct from one another.

1. In Linux, the Operating System stores passwords in the /etc/shadow file. Within the shadow file, the passwords stored have 3 parts: the algorithm used, salt, and password hash. The salt and password hash are encoded as printable characters whereas the algorithm is not. The reasoning as to why Linus applies a password hash for many rounds, like 5000, is because it can slow down any potential brute force attacks and keeps the passwords secured from any attacker.

1. We can treat M = message, K = secret. When we apply SHA256 to K || M, it will hash M’ = M || Padding. This is because SHA256 goes through blocks iteratively. Here, the padding bits will be filled into the last block by the hash function. So, with no padding involved, SHA256(K || M) will hash M’ = M. Now, M’ = M || X. So, it generates a valid signature without knowing a secret key. So, if we know SHA256(secret || constant), we can then compute SHA256 as SHA256(secret || constant || newData).

1. Yes, we would need to know the length of the key to execute an attack, the information/content of the message is not needed. The attacker would only require two attributes to initiate the attack, the hash message, and the length of the message.

1. The padding used will be when it multiplies the hash with the exact length to form the encoded message. The hash value for N with the given string “extra content” is "54d7d4c9f8c16de2dd189eadeeb239359f253fc59221665b7e22b6e6e23a3ce3". Therefore, the hash(K:M:N) is: abcd9313x:1234567890123456789012345678901234567890:54d7d4c9f8c16de2dd1 9eadeeb239359f253fc59221665b7e22b6e6e23a3ce3.
2. Alice would be able to compute HMAC (K, “This is Alice”) and send the result to Bob. Bob will then be able to verify the result if it is correct by computing the same, HMAC (K, “This is Alice”) by himself. Alice could also send the secret number K to Bob, and he can decrypt it to verify it is Alice, and if she knows the secret number. Alice could also request Bob to compute HMAC (K, “Alice’s message”) himself and she could send her results to Bob to verify each other. Alternatively, Alice could send Bob a challenge whereby Bob would have to compute a random string challenge through HMAC (K, challenge) and he would then send the results back to Alice for verification.
3. The Merkle-Damgard construction is a method to create a cryptographic hash function that uses a one-way compression function. The construction is grounded on the notion that if the compression function is collision-resistant, so will the hash function too. The construction has been used to design various hash functions like MD5, SHA-1, and SHA-2. The process of the construction begins with the lengthening of the input message to a multiple of some fixed number of bits. Since the compression function only works on fixed-length inputs. From the diagram below, we get two blocks in one of the Merkle-Damgard system. The first is a padding block with 1 and 0, the second is the message size block. The input message is split into blocks of a set length. After that, the blocks are processed with a one-way compression function (f) and an initial vector (IV). The initial vector is a predefined value that changes with every iteration, and finally produces the hash. This will then be processed further with the finalization function, which ensures a fixed-length output.



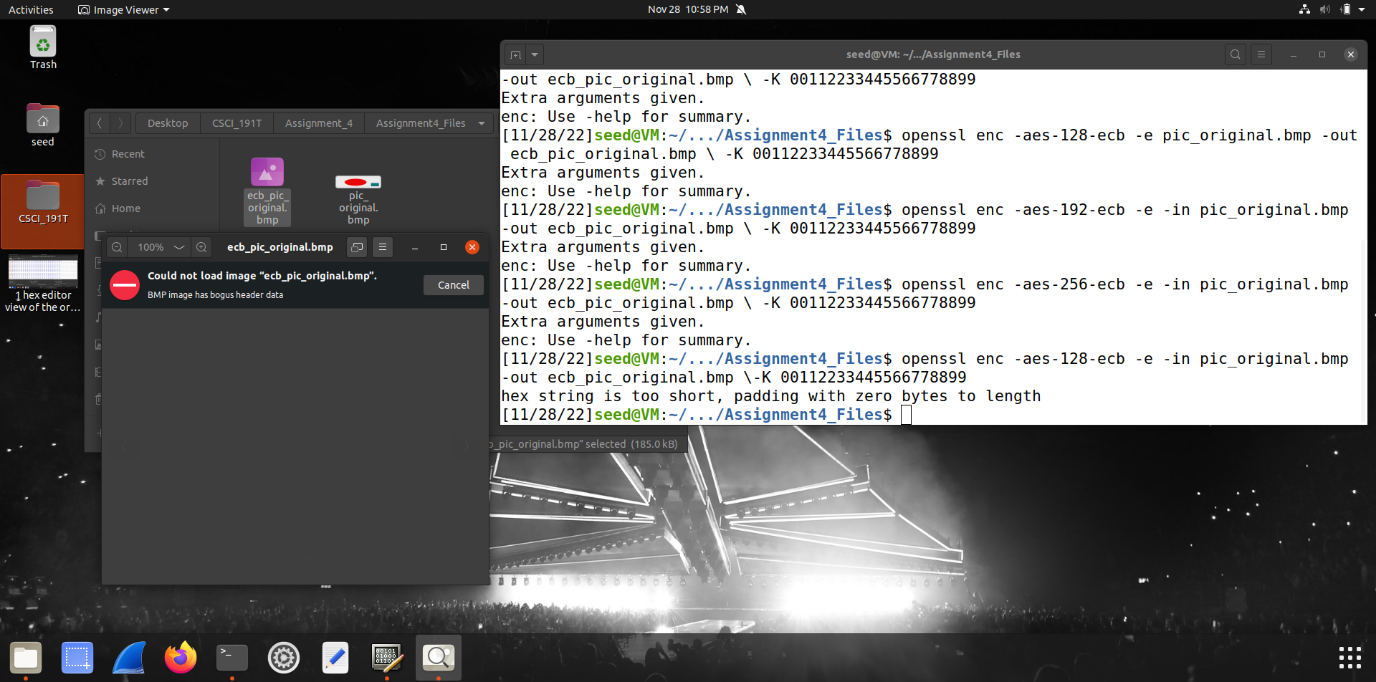
1. If Bob found two different messages M1 and M2 where they both have the same SHA256 hash, it is extremely rare to do so. This is because SHA256 is made to create unique strings of fixed length 256 bit whenever an input of any size is passed into it. Hashing algorithms are considered secure unless it intentionally generates collisions. In the case for the pair M3 and M4, it is not possible to find another pair where SHA256 hashes resulted in being the same.

**Section 2:**

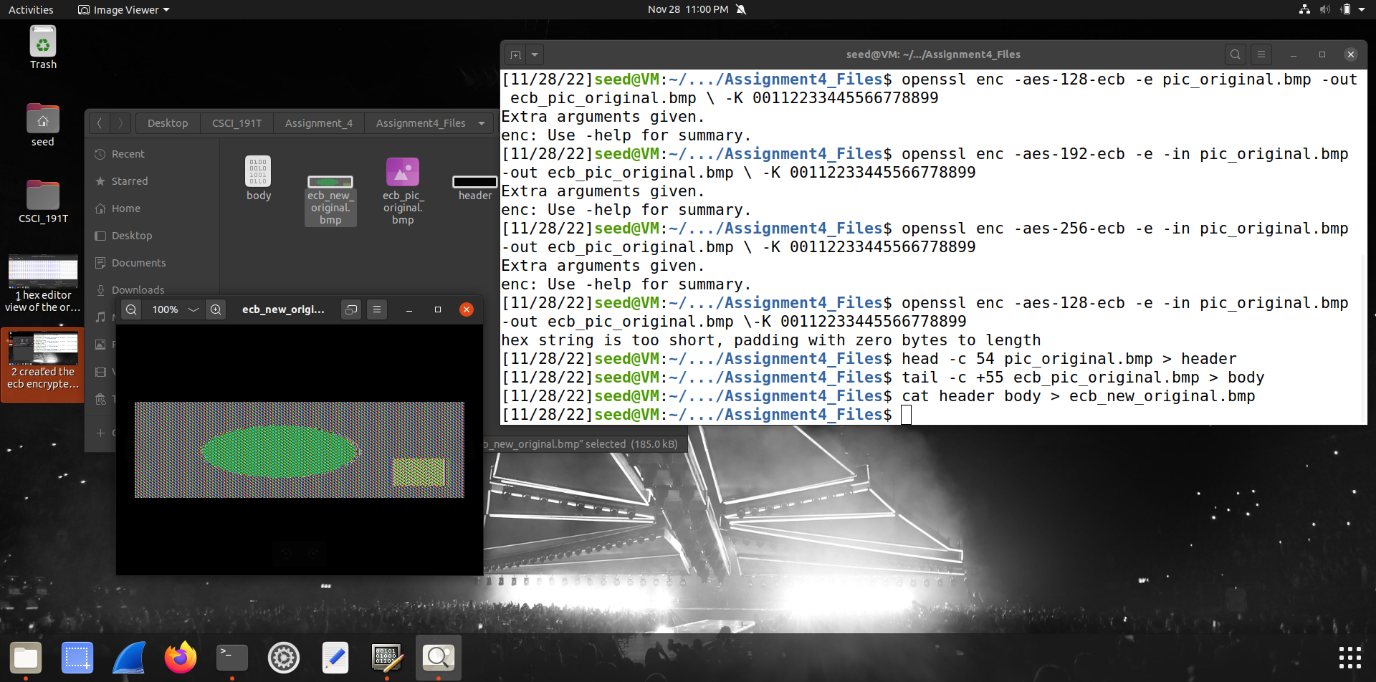
**Part 1 (Using pic\_original.bmp):**



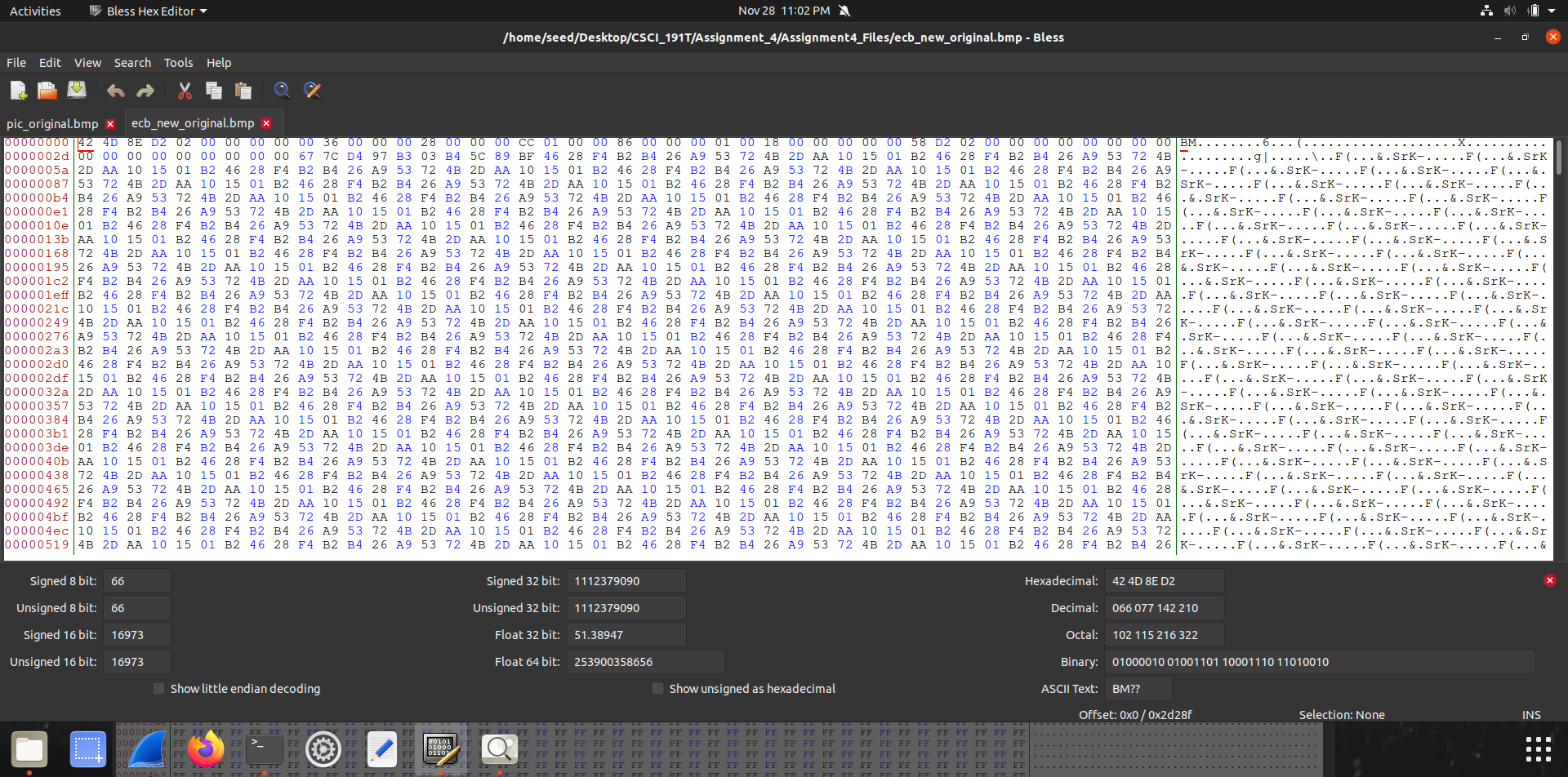
**Step 1:** Observing the hex editor of the original image before any encryption occurs.



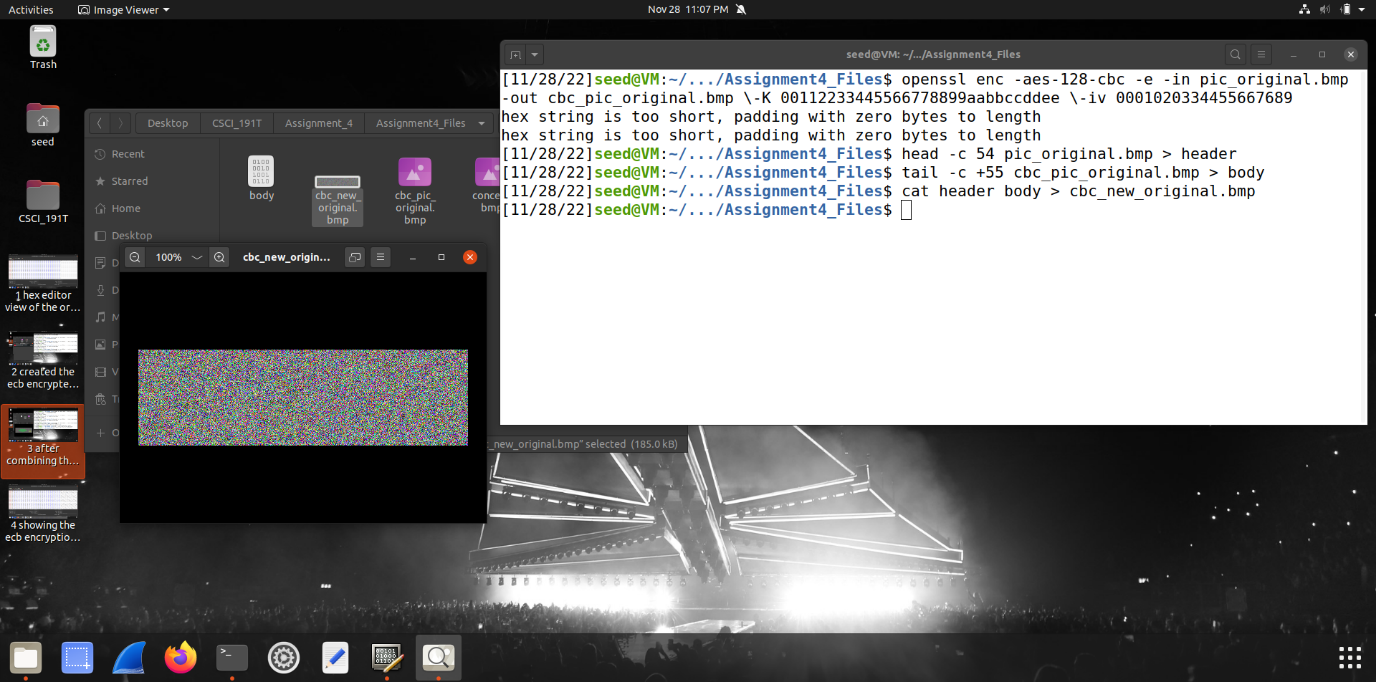
**Step 2:** After creating the ecb encryption of the original image, the image cannot be opened on the image viewer.



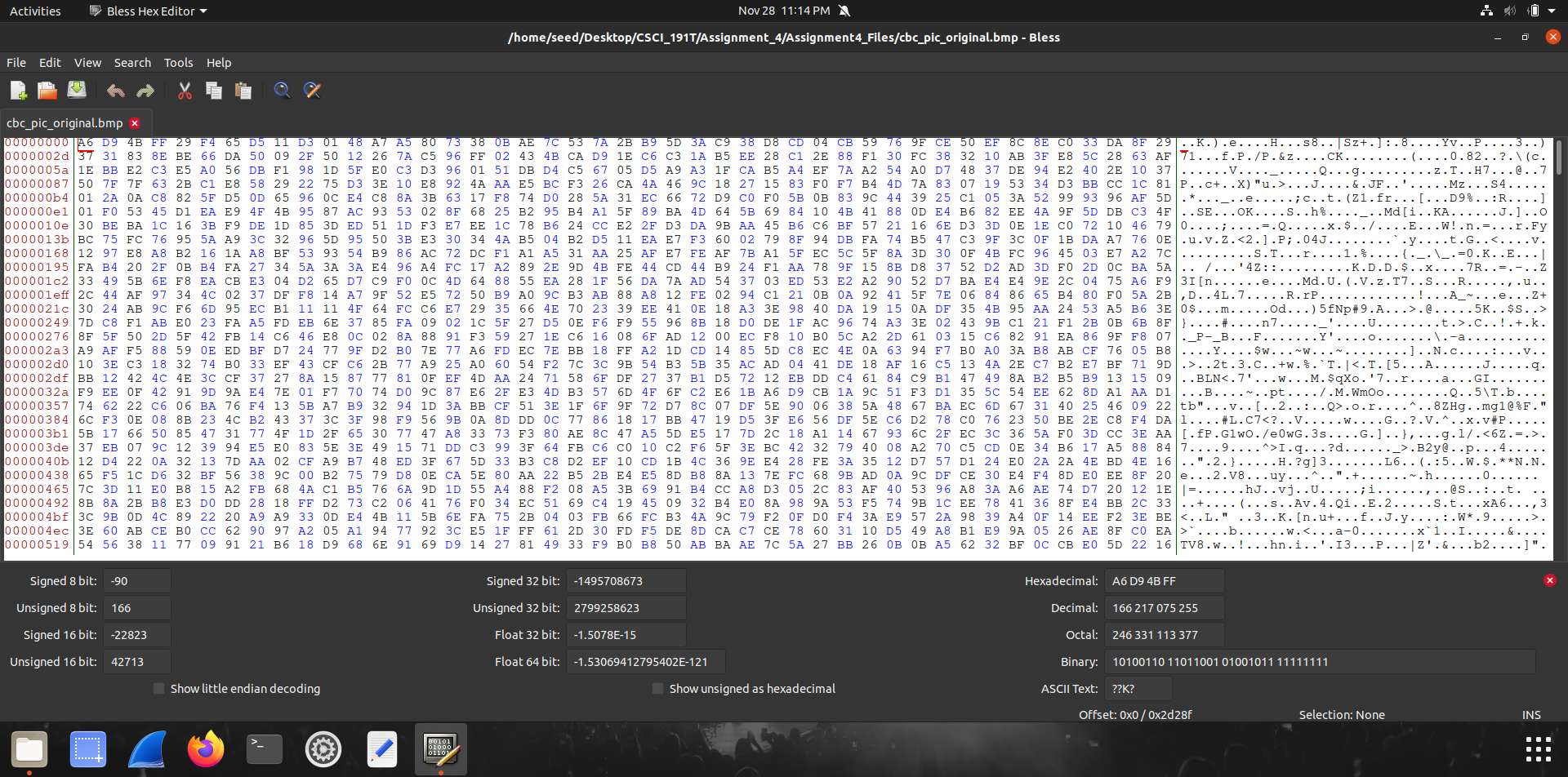
**Step 3:** After combining the header of the original image and the body of the encrypted image, we can get an encrypted version of the image to show in the image viewer that is slightly encrypted and can still be observed as the shapes are still visible but are not in the original color as before.



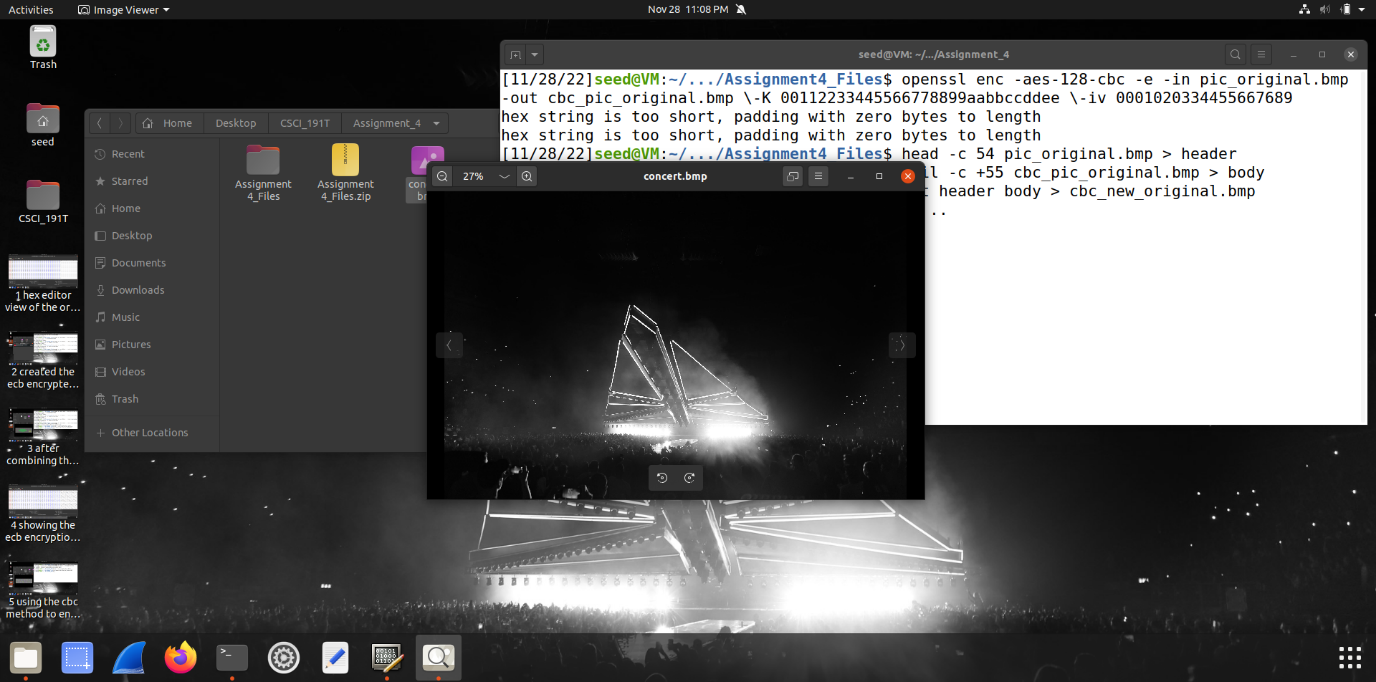
**Step 4:** This was the editor showing the ECB encryption that has taken place for the specific file.

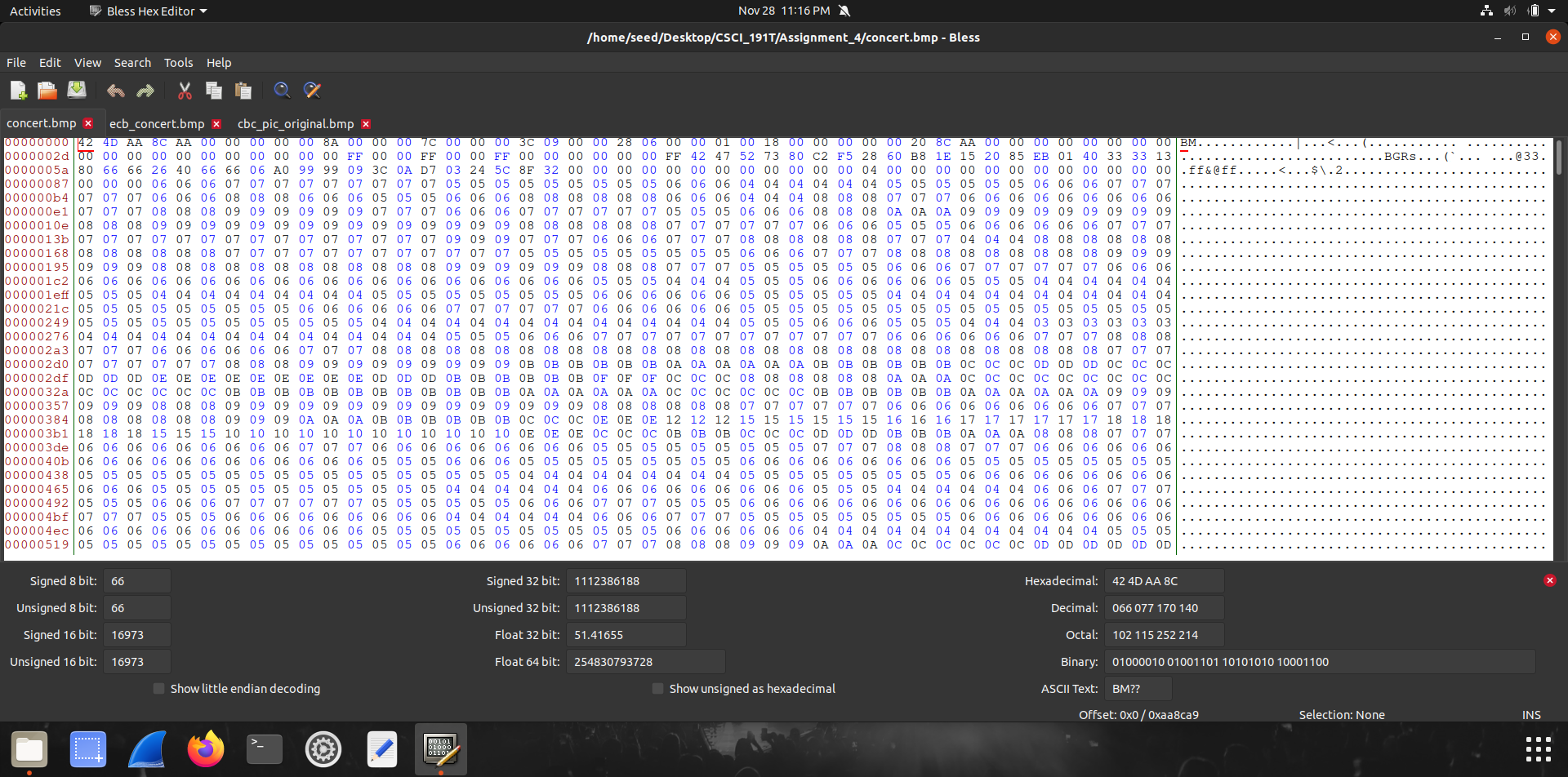


**Step 5:** When doing the CBC encryption, we can see that the image in the image viewer is now completely hidden, we are unable to see the original image and it is all blurred out.

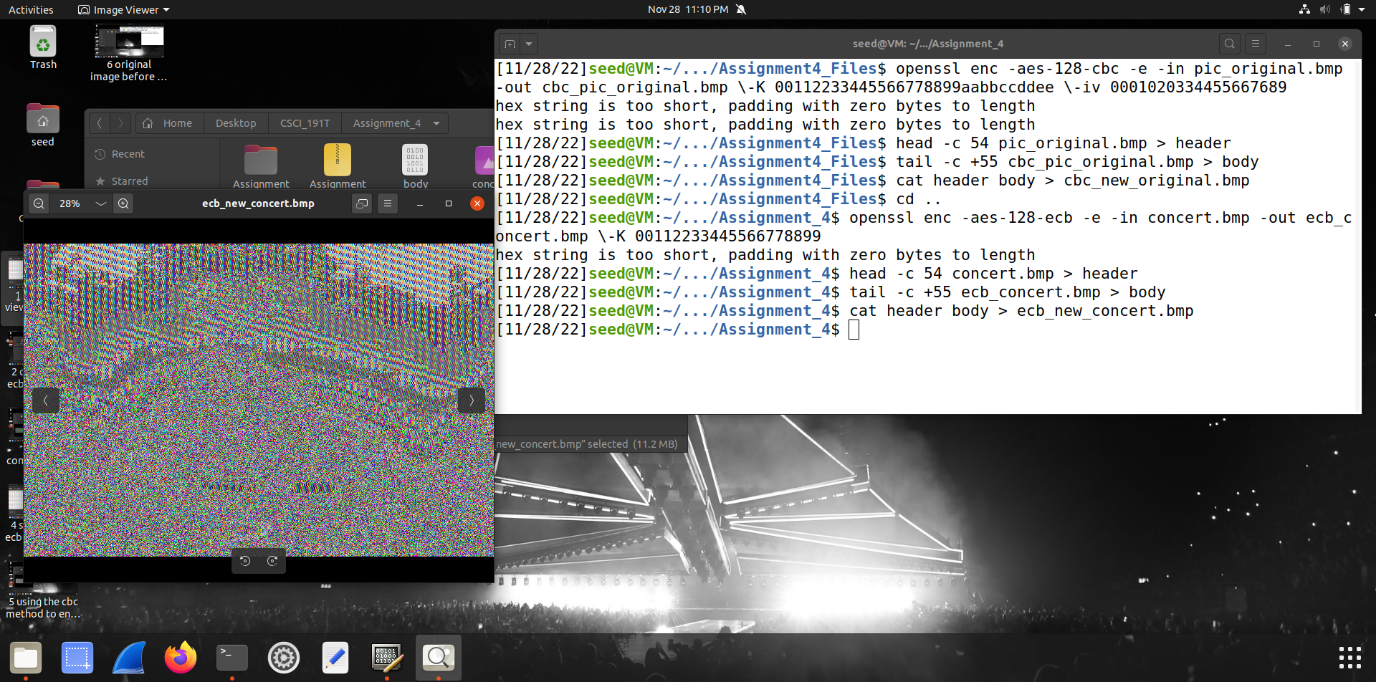


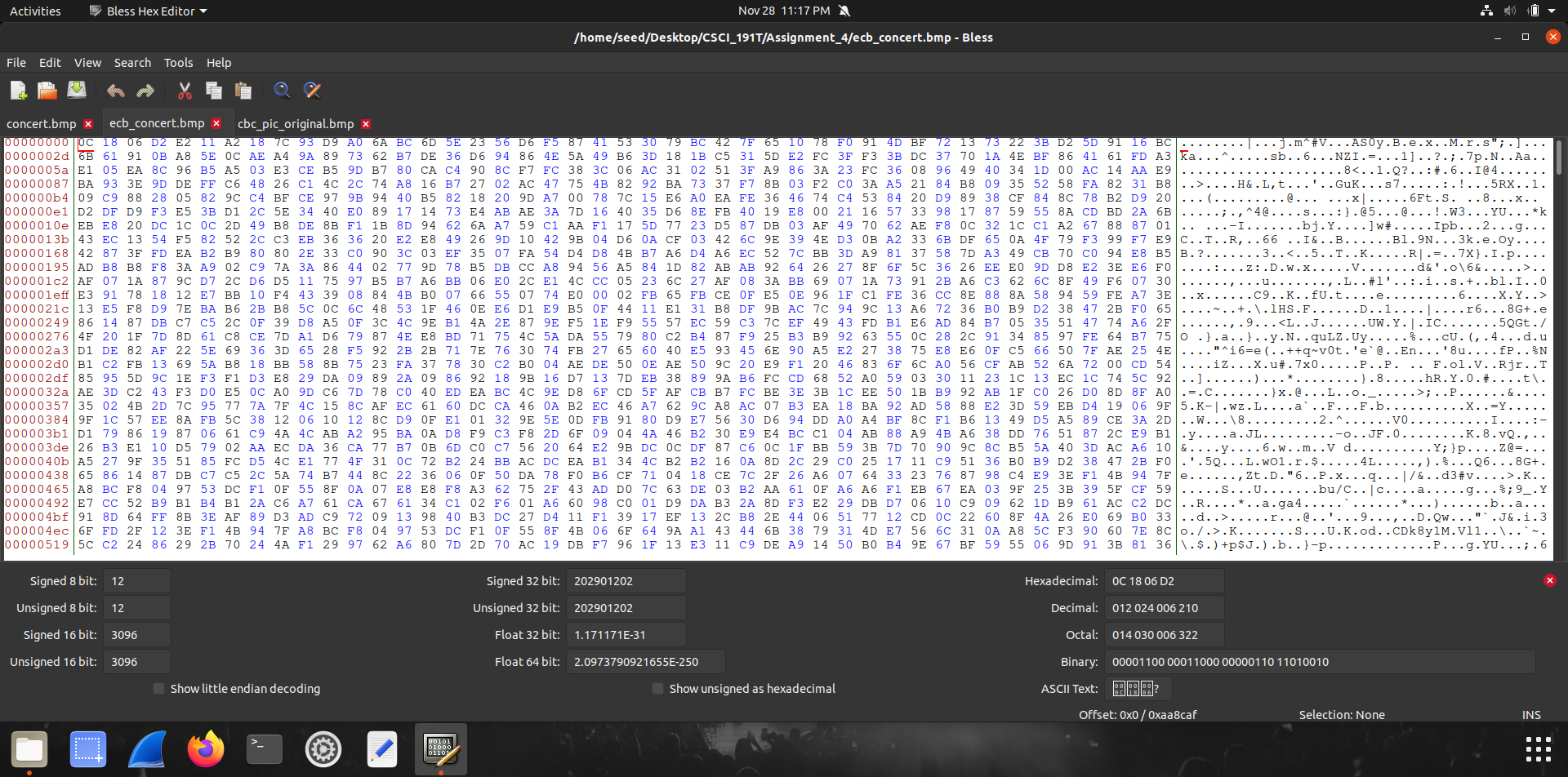
**Step 6:** In the CBC mode, once again the image’s hex editor displays different variables.

**Part 2 (Using own image):**

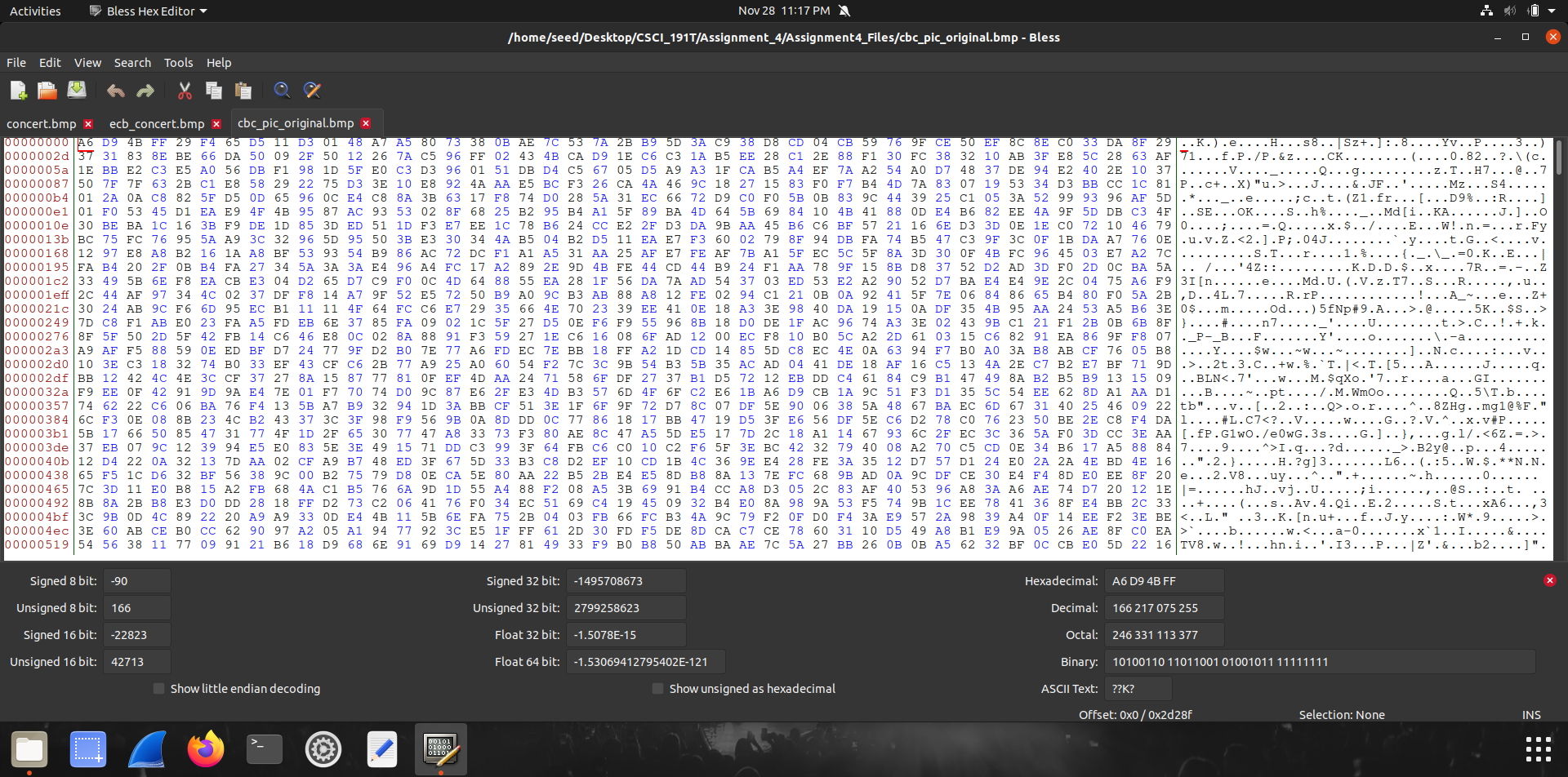
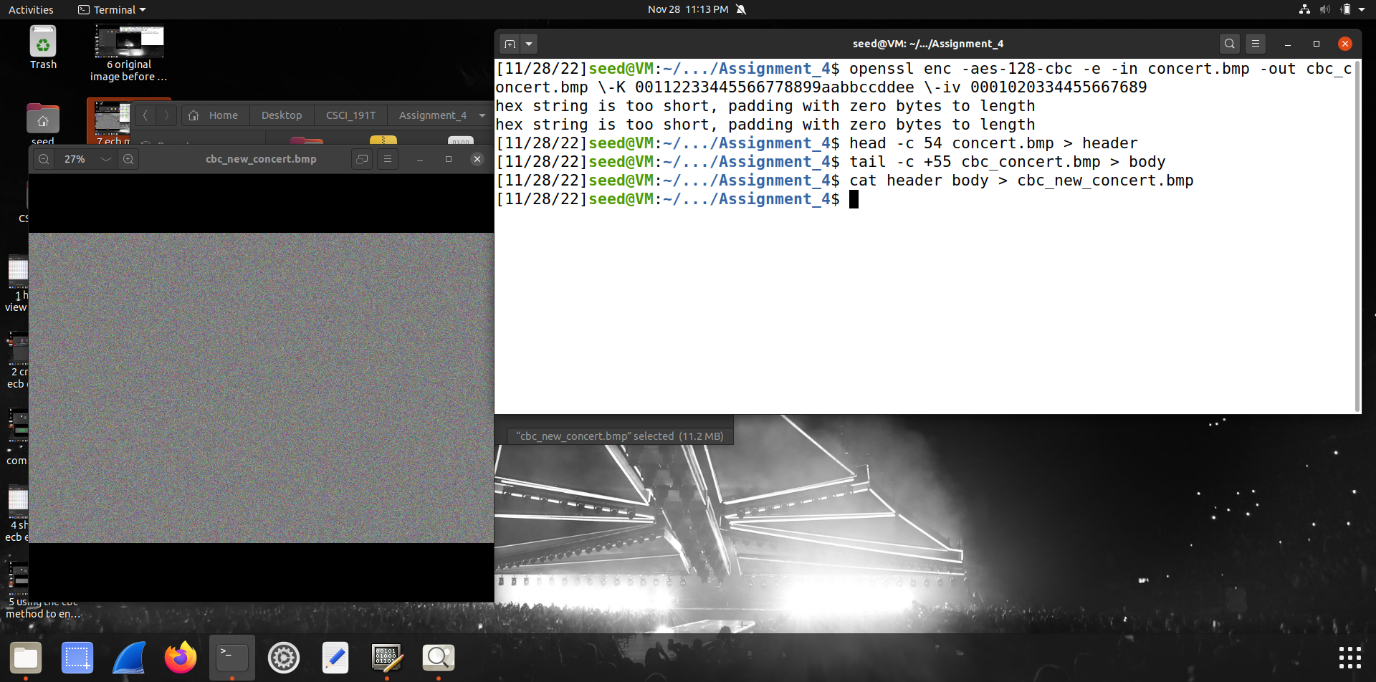


**Step 1:** This is the original image that will be encrypted in both the ECB and CBC method. The second image is the hex editor for the original image.





**Step 2:** Using the ECB method, the image in this case is very hard to see but there are some outlines that can be observed on the image. The hex editor also changes once again in the ECB encryption mode for this image.



**Step 3:** Like the previous CBC encryption mode, this mode blocks out the image entirely, therefore not making it viewable even through the image viewer. The hex editor has also changed with the mode of encryption being used.

To conclude, from both images used in Part 1 and Part 2, it can be observed that the ECB mode somewhat partially encrypts the files whereas the CBC mode completely prevents anyone from viewing the image. Despite the EBC mode blurring the image, the image does not show its original colors and makes the image viewer to display it a little differently.