

Encryption

CPE 549

KYLE RAY

September 30, 2019

Contents

Task 2 (Encryption Mode ECB vs. CBC)	2
Procedure	2
Conclusions	3
Task 3 (Encryption Mode – Corrupted Cipher Text)	3
Procedure	3
Conclusions	6
ECB.....	6
CBC	6
CFB.....	7
OFB	7
Task 4 (Padding).....	7
Procedure	7
Conclusions	11

Task 2 (Encryption Mode ECB vs. CBC)

PROCEDURE

In order to visualize the effects of the two encryption modes Electronic Code Book (ECB) and Cipher Block Chaining (CBC), this task requires an original image, refer to **Figure 1** below, be encrypted with both modes and then viewed to determine how well the data is hidden with each encryption mode.

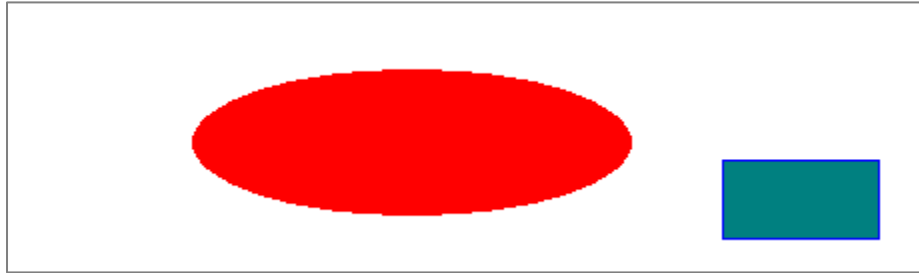


Figure 1: Original Image

Openssl is the tool used for encryption and decryption throughout this lab. The following openssl command was used to encrypt the original image with 128-bit AES encryption using the ECB mode.

```
openssl enc -aes-128-ecb -e -in pic_original.bmp -out encrypted_image_ecb.bin -K 00112233445566778899aabbccddeeff
```

In order to view the encrypted binary file as a bmp image, the 54-byte bmp header was copied from the original image and placed in the first 54-bytes of the encrypted file, producing the image in **Figure 2**.

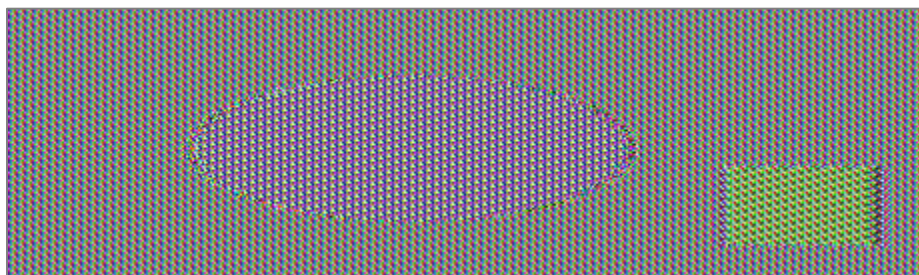


Figure 2: Original Image with 128-bit AES ECB Encryption

While the encrypted image does differ from the original, the data is still visible (the shapes). Electronic Code Book encryption works on independent blocks and therefore if the variance of the data being encrypted is not high, then in the case of encrypting images, the important bits of data may still be visible after encryption.

The following openssl command was used to generate the 128-bit AES CBC encrypted version of the original image. The only differences from the ECB example above are the change in encryption mode as well as supplying the required initialization vector (-iv).

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out encrypted_image_cbc.bin -K  
00112233445566778899aabbccddeeff -iv 000102030405060708090a0b0c0d0e0f
```

The same method of replacing the 54-byte bmp header was used to produce the encrypted image in **Figure 3**.



Figure 3: Original Image with 128-bit AES CBC Encryption

The result of the CBC encryption mode is more random, and the original image is not visible in the encrypted image.

CONCLUSIONS

The use of ECB for encrypting images does not seem secure as the encrypted image still contained the important information from the original, the image of a circle and a square. ECB encrypts each block independently and the same plain text produces the same cipher text; therefore, if the variance in the data is low then a similar image will be produced from the encryption. However, the encrypted image resulting from the CBC encryption mode is more random and the information from the original image is not visible.

Task 3 (Encryption Mode – Corrupted Cipher Text)

PROCEDURE

This task attempts to help demonstrates the properties of various encryption modes. The following are the steps taken to illustrate these properties:

- 1.) Create a text file that is at least 64 bytes long.
- 2.) Encrypt the file using the following AES-128 cipher.
 - a. Electronic Code Book (ECB)
 - b. Cipher Block Chaining (CBC)
 - c. Cipher Feedback (CFB)
 - d. Output Feedback (OFB)
- 3.) Change exactly 1 bit in the encrypted output using a hex editor

4.) Decrypt the corrupted file using the correct key and initialization vector.

First, a text file called *Message.txt* was created with the following 69 bytes of content:

Encryption and Decryption are very useful tools for protecting data. This message was then encrypted with each of the encryption modes given in step 2. Again, the tool openssl was used to perform the encryption on the files using the following commands, the same key and initialization vector are used for each mode when necessary:

ECB:

```
openssl enc -aes-128-ecb -e -in Message.txt -out Message_encrypt_ecb.bin -K  
00112233445566778899aabbccddeeff
```

CBC:

```
openssl enc -aes-128-cbc -e -in Message.txt -out Message_encrypt_cbc.bin -K  
00112233445566778899aabbccddeeff -iv 000102030405060708090a0b0c0d0e0f
```

CFB:

```
openssl enc -aes-128-cfb -e -in Message.txt -out Message_encrypt_cfb.bin -K  
00112233445566778899aabbccddeeff -iv 000102030405060708090a0b0c0d0e0f
```

OFB:

```
openssl enc -aes-128-ofb -e -in Message.txt -out Message_encrypt_ofb.bin -K  
00112233445566778899aabbccddeeff -iv 000102030405060708090a0b0c0d0e0f
```

In order to see the effects of corruption for step 3 a single bit was changed in each of the encrypted files generated from the commands above. The first byte of file was changed by a single bit. For the first file a screen shot of *wxHexEditor* is shown in **Figure 4** to illustrate the changing of the field. For the rest of the files the description of which bit was changed is all that is given.

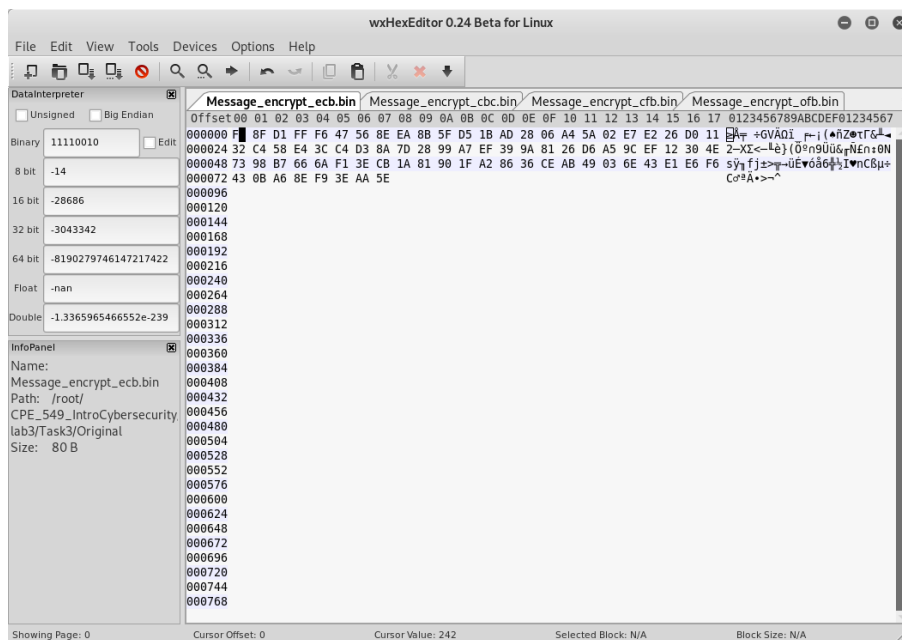


Figure 4: wxHexEditor Editing Single Bit for ECB Encrypted Message

The following changes were made to each encrypted file to simulate a small amount of corruption. Byte positioning is presented as zero-based; therefore, byte position 0 is the first byte in the file:

ECB: Byte position 0 changed from 0xE2 to 0xF2

CBC: Byte position 0 changed from 0x0F to 0x1F

CFB: Byte position 0 changed from 0x62 to 0x72

OFB: Byte position 0 changed from 0x62 to 0x72

These changes were saved into a separate file following the naming scheme “Message_encrypt_**mode**_corrupted.bin”. After the corruption was applied to the files they were decrypted using the following openssl commands:

ECB:

```
openssl enc -aes-128-ecb -d -in Message_encrypt_ecb_corrupted.bin -out
Message_decrypt_ecb_corrupted.txt -K 00112233445566778899aabbccddeeff
```

CBC:

```
openssl enc -aes-128-cbc -d -in Message_encrypt_cbc_corrupted.bin -out
Message_decrypt_cbc_corrupted.txt -K 00112233445566778899aabbccddeeff -iv
000102030405060708090a0b0c0d0e0f
```

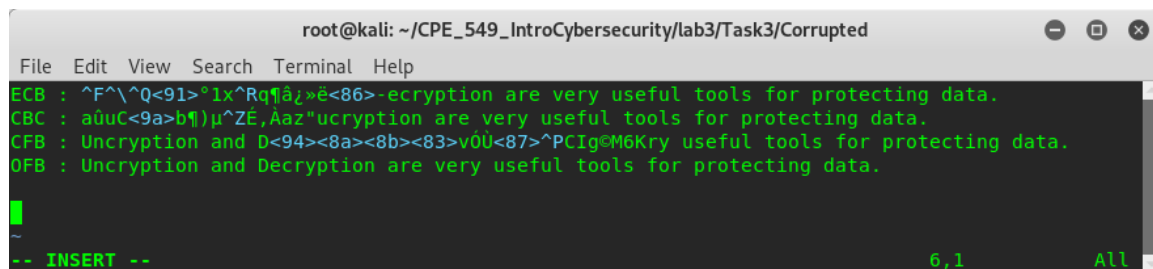
CFB:

```
openssl enc -aes-128-cfb -d -in Message_encrypt_cfb_corrupted.bin -out  
Message_decrypt_cfb_corrupted.txt -K 00112233445566778899aabbccddeeff -iv  
000102030405060708090a0b0c0d0e0f
```

OFB:

```
openssl enc -aes-128-ofb -d -in Message_encrypt_ofb_corrupted.bin -out  
Message_decrypt_ofb_corrupted.txt -K 00112233445566778899aabbccddeeff -iv  
000102030405060708090a0b0c0d0e0f
```

The results of decrypting each file were very interesting and there are noticeable differences between each mode. Recall the original message: **Encryption and Decryption are very useful tools for protecting data.** Figure 5 below contains a screenshot of the decryption results for each encryption mode.



```
root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task3/Corrupted  
File Edit View Search Terminal Help  
ECB : ^F^\^Q<91>*1x^Rq!â¿»è<86>-ecryption are very useful tools for protecting data.  
CBC : aûuC<9a>b¶)µ^ZÊ,Äaz"ucryption are very useful tools for protecting data.  
CFB : Uncryption and D<94><8a><8b><83>v0Ü<87>^PCIgM6Kry useful tools for protecting data.  
OFB : Uncryption and Decryption are very useful tools for protecting data.  
-- INSERT -- 6,1 All
```

Figure 5: Decryption Results of Corrupted Files For Each Encryption Mode

CONCLUSIONS

As expected, the decryption of the corrupted cipher text varies based on the encryption mode. Below are the conclusions derived for each encryption mode.

ECB

Due to the fact that ECB independently decrypts blocks of data and the cipher text was corrupted via a single bit in the first block, the expected result was that the first of the message be corrupted. Referring to **Figure 5**, this is the effect observed as the first of the message is unreadable, but the rest of the message is still intact. The message is quite damaged and thus the implication is that the message needs to be resent to the target.

CBC

Decryption using the CBC mode, the cipher text blocks are used not only to generate the current block of plain text but the next block of plain text as well. Therefore, changing a bit in the first of the message should have impacted at least two blocks of plain text. Referring to the figure above this does seem to be the case as approximately 17 bytes of the message has been corrupted, yet the rest of the message is still intact. The message is quite damaged and thus the implication is that the message needs to be resent to the target.

CFB

Since CFB is similar to CBC, changing a bit in the first block of cipher text should have propagated through to the next blocks. Because the previous block of cipher text is used for decryption in the next block and then xor'd with the next block of cipher text, the propagation of corruption should be more severe. Referring to the figure above, the resulting decrypted message is very damaged, with over half of the message being unreadable. The message is heavily damaged and the implication is that it must be resent to the target.

OFB

OFB is considered good for network communication; therefore, changing a single bit in the cipher text should have little effect to uphold this description. Flipping a single bit in cipher text flips the same bit in plain text and that is the exact behavior seen in the figure above. The first byte was changed from 0x62 to 0x72 thus changing the first character only and leaving the rest of the message intact. The message is still overall intact and thus the damaged portion can probably be corrected via some correction algorithm and the message doesn't have to be resent to the target.

Task 4 (Padding)

PROCEDURE

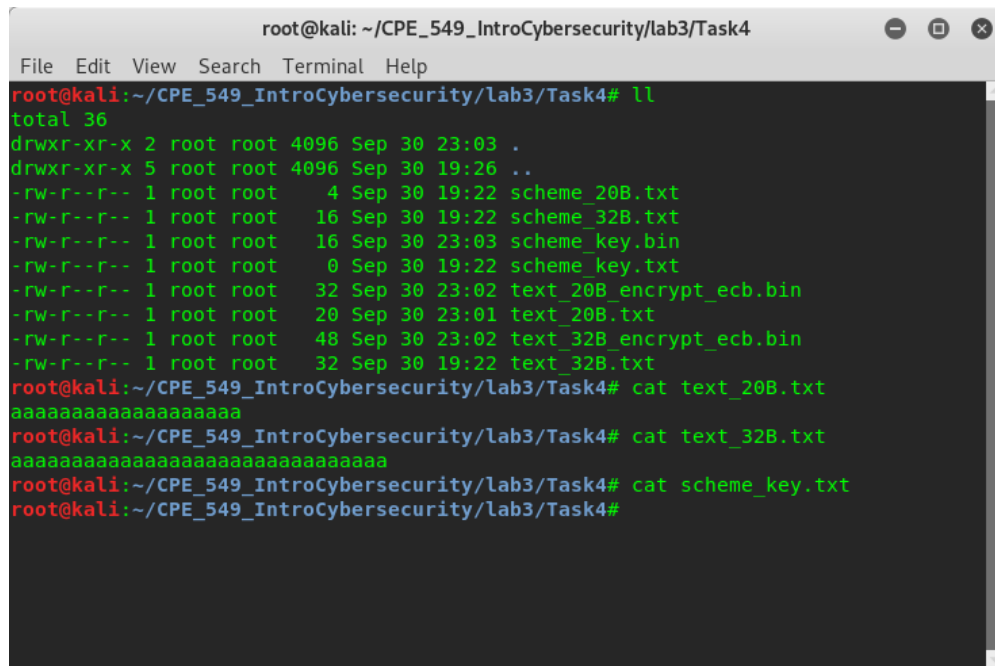
The AES encryption algorithms for openssl utilize the PKCS#7 padding algorithm to maintain the correct block size for encryption. The block size for AES is 16 bytes. When padding a block PKCS#7 adds filler bytes, always adding at least one byte, to the plain text in order to make the block size a multiple of 16. The filler bytes contain the number of bytes added, if 12 bytes added then each filler byte is 0x0C, to achieve this multiple and then the file is encrypted. When decrypting, the algorithm reads the last byte and knows how much padding to remove in order to regain the original plain text.

In order to view the padding used in the plain text by the PKCS#7 algorithm the following scheme was derived:

- 1.) Encrypt plain text with 128-bit AES ECB encryption
 - a. A 20-byte plain text file.
 - b. A 32-byte plain text file.
- 2.) Encrypt a plain text key, a 0-byte file in this case, with 128-bit AES ECB encryption.
- 3.) Concatenate each original cipher text with the cipher text of the key from step 2.
- 4.) Decrypt the new cipher text.

The 20-byte and 32-byte files used in this task consisted simply of the letter 'a' and the null terminating character to achieve the length requirement. In order to view the padding used during encryption the independent encrypt and decrypt of the ECB algorithm is utilized. To achieve this, a 0-byte file, "scheme_key.txt", is encrypted using the same method as for the 20 and 32-byte files. PKCS#7 adds 16 bytes of padding to the 0-byte file and thus is its own block; therefore,

during decryption the algorithm will remove this as padding thus leaving the original padding from the 20 and 32-byte files intact. **Figure 6** below shows the file contents and their sizes.



```

root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task4
File Edit View Search Terminal Help
root@kali:~/CPE_549_IntroCybersecurity/lab3/Task4# ll
total 36
drwxr-xr-x 2 root root 4096 Sep 30 23:03 .
drwxr-xr-x 5 root root 4096 Sep 30 19:26 ..
-rw-r--r-- 1 root root  4 Sep 30 19:22 scheme_20B.txt
-rw-r--r-- 1 root root 16 Sep 30 19:22 scheme_32B.txt
-rw-r--r-- 1 root root 16 Sep 30 23:03 scheme_key.bin
-rw-r--r-- 1 root root  0 Sep 30 19:22 scheme_key.txt
-rw-r--r-- 1 root root 32 Sep 30 23:02 text_20B_encrypt_ecb.bin
-rw-r--r-- 1 root root 20 Sep 30 23:01 text_20B.txt
-rw-r--r-- 1 root root 48 Sep 30 23:02 text_32B_encrypt_ecb.bin
-rw-r--r-- 1 root root 32 Sep 30 19:22 text_32B.txt
root@kali:~/CPE_549_IntroCybersecurity/lab3/Task4# cat text_20B.txt
aaaaaaaaaaaaaaaaaaaa
root@kali:~/CPE_549_IntroCybersecurity/lab3/Task4# cat text_32B.txt
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
root@kali:~/CPE_549_IntroCybersecurity/lab3/Task4# cat scheme_key.txt
root@kali:~/CPE_549_IntroCybersecurity/lab3/Task4#

```

Figure 6: Task4 File Contents and Sizes

Referring to the figure above the PKCS#7 has applied 12 bytes of padding to the 20-byte text file thus giving a 32-byte binary file and applied 16 bytes of padding to the 32-byte file thus giving a 48-byte binary file. Recall that PKCS#7 adds at least one byte of padding; therefore, in the case of the 32-byte file it needs to add 16 bytes to keep the multiple of 16 rule. This is the same behavior for the 0-byte scheme file.

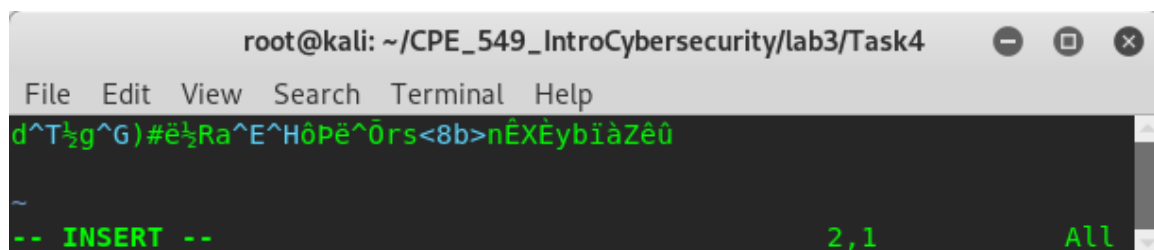
The cipher text is acquired for both the 20 and 32-byte files using the following openssl command, changing the file name as necessary:

```

openssl enc -e -aes-128-ecb -in text_20B.txt -out text_20B_encrypt_ecb.bin -K
00112233445566778899aabbccddeeff

```

The following figures contain the results of the encryption of each file.



```

root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task4
File Edit View Search Terminal Help
d^T½g^G)#ë½Ra^E^HôPë^Ôrs<8b>nÊXËybîàZÊû
~
-- INSERT --                                2,1                                All

```

Figure 7: 20-byte File Encrypted

```

root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task4
File Edit View Search Terminal Help
d^T½g^G)#ë½Ra^E^HôPë²÷¶ÿ°EØP6ëÄ'≡!ÖÑ^@e~¡@eZDx'Gp]B/
~
~
-- INSERT --                                2,1          All

```

Figure 8: 32-byte File Encrypted

```

root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task4
File Edit View Search Terminal Help
^@e~¡@eZDx'Gp]B/
~
~
-- INSERT --                                2,1          All

```

Figure 9: Scheme Key Encrypted

Following the scheme, the cipher text from the scheme key is concatenated with the cipher text of the other files, resulting in the cipher texts below.

```

root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task4
File Edit View Search Terminal Help
d^T½g^G)#ë½Ra^E^HôPë^Ôrs<8b>nÊXËybïàZêû^@e~¡@eZDx'Gp]B/
~
~
-- VISUAL --                                15-17          1,48-42        All

```

Figure 10: 20-byte Encrypted File with Scheme Key Concatenated

```

root@kali: ~/CPE_549_IntroCybersecurity/lab3/Task4
File Edit View Search Terminal Help
d^T½g^G)#ë½Ra^E^HôPë²÷¶ÿ°EØP6ëÄ'≡!ÖÑ^@e~¡@eZDx'Gp]B/ ^@e~¡@eZDx'Gp]B/
~
~
-- VISUAL --                                15-17          1,70-56        All

```

Figure 11: 32-byte File with Scheme Key Concatenated

Referring to **Figure 10** and **Figure 11**, the highlighted portion depicts the cipher text from the 16-byte scheme key. When passing these cipher texts through the decryption algorithm, the algorithm pulls off the padding applied by the scheme key and preserves the original padding. The openssl decryption command is as follows, changing the file name as necessary:

```
openssl enc -d -aes-128-ecb -in text_20B_encrypt_ecb.bin -out text_20B_decrypt_ecb.txt -K
00112233445566778899aabbccddeeff
```

The following figures show the padding values in *wxHexEditor*. For the 20-byte file the expected padding is 12 bytes, or 0x0C, and for the 32-byte file the expected padding is 16 bytes, or 0x10, which are highlighted in the figures below.

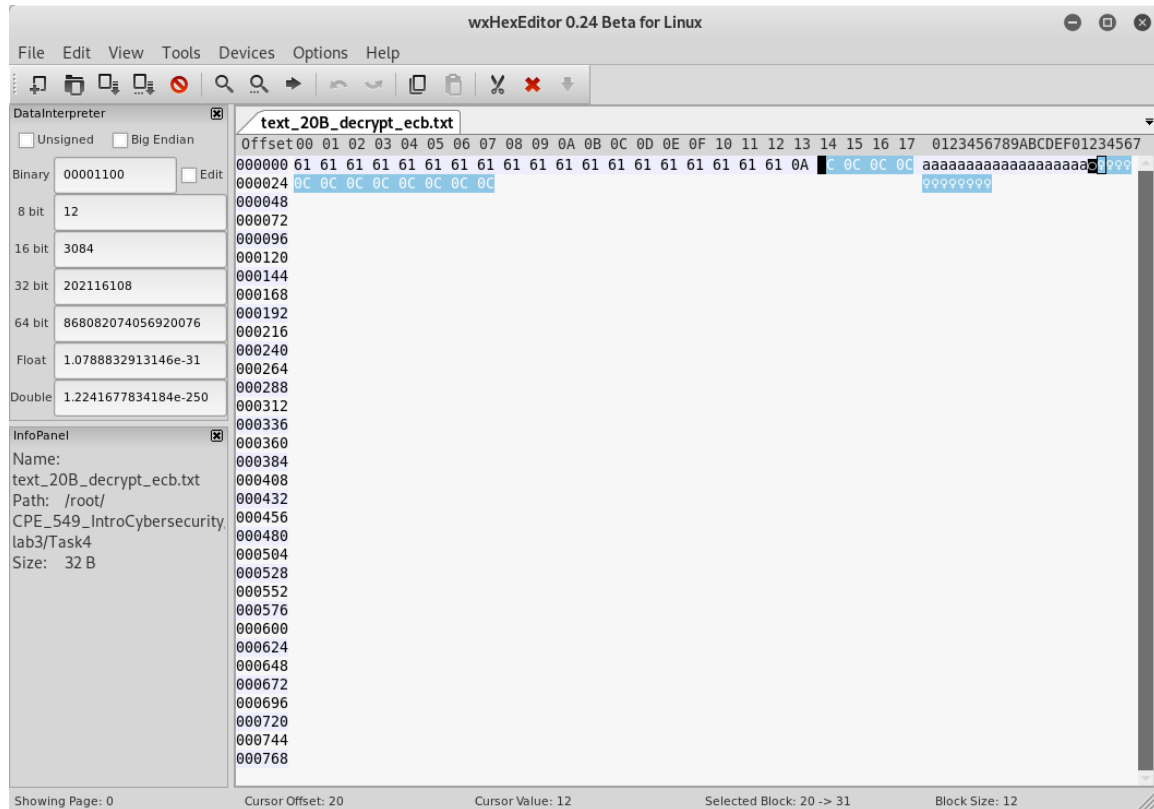


Figure 12: Hex View of 20-byte Decrypted File with Padding

