

LAB 1 - REPORT

Justin Kaipada - 100590167

January 27, 2020

Task #1: Encryption using different ciphers and modes

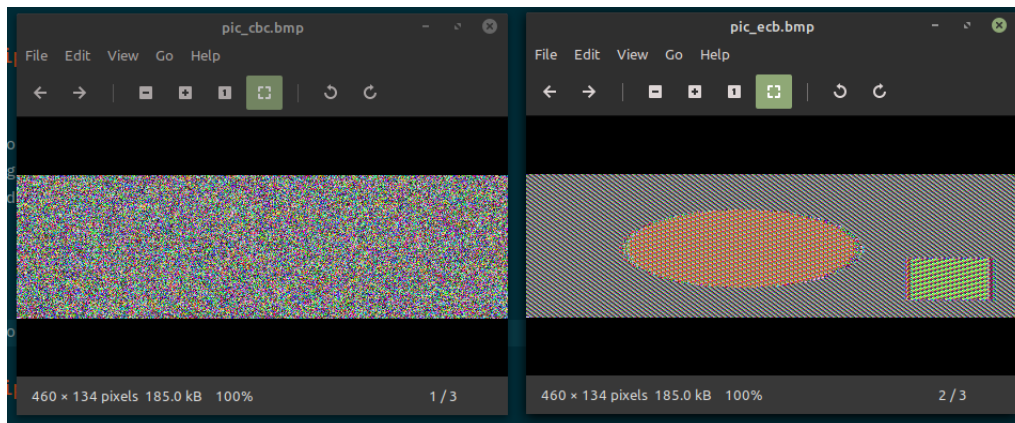
```
openssl enc -aes-128-cbc -e -in plain_text.txt -out cipher1.bin \  
-K 000102030405060708090a0b0c0d0e0f \  
-iv 000102030405060708090a0b0c0d0e0f  
  
openssl enc -aes-128-cfb -e -in plain_text.txt -out cipher2.bin \  
-K 000102030405060708090a0b0c0d0e0f \  
-iv 000102030405060708090a0b0c0d0e0f  
  
openssl enc -aes-192-gcm -e -in plain_text.txt -out cipher3.bin \  
-K 000102030405060708090a0b0c0d0e0f \  
-iv 000102030405060708090a0b0c0d0e0f  
  
openssl enc -des-ede3-cfb1 -e -in plain_text.txt -out cipher4.bin \  
-K 000102030405060708090a0b0c0d0e0f \  
-iv 000102030405060708090a0b0c0d0e0f
```

In the above 4 commands used, the 3rd one failed saying `enc: AEAD ciphers not supported`. This was a simple task that introduced us to the basics of using openssl to encrypt files.

Task #2: Encryption Mode - ECB vs CBC

After encrypting the file using an ecb encryption algorithm and correcting the header information, the image is somewhat representative of the original image, at least I can tell there is an outline of the original image preserved in the encrypted file. The encrypted file can be viewed in `pic_ecb.bmp`.

After modifying the file header information the file produced as a result of the ecb encryption, it didn't look anything like the original picture. The file `pic_cbc.bmp` shows the encrypted image with corrected header and as you can see the whole image is disfigured.



Task #3: Encryption Mode - Corrupted Cipher Text

Commands used for encryption

```
openssl enc -aes-128-ecb -e -in plain_text.txt -out task3ecb.bin \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f

openssl enc -aes-128-cbc -e -in plain_text.txt -out task3cbc.bin \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f

openssl enc -aes-128-cfb -e -in plain_text.txt -out task3cfb.bin \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f

openssl enc -aes-128-ofb -e -in plain_text.txt -out task3ofb.bin \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f
```

Before deciphering the files I say ecb will be most affected by corruption and ofb will be least affected, because ofb uses XOR operation on the bits so messing with one byte should only affect 1 byte when deciphering.

Commands used for decryption

```
openssl enc -aes-128-ecb -d -in task3ecb.c.bin -out plain_text_ecb.txt \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f
```

```

openssl enc -aes-128-cbc -d -in task3cbc_c.bin -out plain_text_cbc.txt \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f

openssl enc -aes-128-cfb -d -in task3cfb_c.bin -out plain_text_cfb.txt \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f

openssl enc -aes-128-ofb -d -in task3ofb_c.bin -out plain_text_ofb.txt \
-K 000102030405060708090a0b0c0d0e0f \
-iv 000102030405060708090a0b0c0d0e0f

```

After generating the encrypted file I corrupted them by replacing a random byte with a random character using the hex editor. It seems like after deciphering the corrupted files the damage done is more or less the same.

CBC This algorithm XOR's a raw text against an IV then encrypts it and propagates that encrypted cyphertext to be the next block's IV. So if a byte is corrupted the data following that will most likely get lost.

```

/home/justin/Downloads/lab 1/plain_text_cbc.txt [Text-mode]
Software & Compuj\254\220\224\247\211\v\337U5^GC\250JL3394.20170\221-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.

NORMAL plain_text_cbc.txt 1:0 All 3:45PM 2.72 ; - TAB 8 CRLFRAW-TEXT Text

```

ECB This algorithm encrypts each block of plaintext to a corresponding ciphertext block, so if byte is changed only that byte will get affected when deciphering back.

```

/home/justin/Downloads/lab 1/plain_text_ecb.txt [Text-mode]
Software & Compu"\212\367\213\313\355>\253^?T\355j\371^_JT3394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.

NORMAL plain_text_ecb.txt 3:57 All 3:45PM 2.72 ; - TAB 8 CRLFRAW-TEXT Text

```

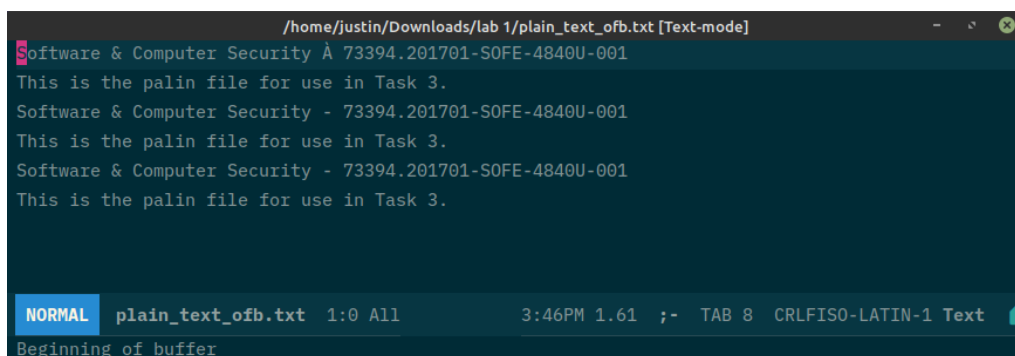
CFB This algorithm is like CBC but is also self-synchronizing. It is able to recover the data if some of the encrypted text is lost or messed up. There may be a lot of data after the corrupted byte affected, the algorithm will try to self correct.



```
/home/justin/Downloads/lab 1/plain_text_cfb.txt [Text-mode]
Software & Computer Security - 73394.201701-S0FE-4840U-0\200^
This\240'\233\274}\333\354\255\372\210^T\210\250\253@vle for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.

NORMAL plain_text_cfb.txt 1:0 All 3:46PM 1.61 ;~ TAB 8 CRLFRAW-TEXT Text
Quit
```

OFB This algorithm should be resistant to corruption of encrypted data. Only the bits that are modified in error will be lost. Neighbouring bits should be unaffected.



```
/home/justin/Downloads/lab 1/plain_text_ofb.txt [Text-mode]
Software & Computer Security Å 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.
Software & Computer Security - 73394.201701-S0FE-4840U-001
This is the palin file for use in Task 3.

NORMAL plain_text_ofb.txt 1:0 All 3:46PM 1.61 ;~ TAB 8 CRLFISO-LATIN-1 Text
Beginning of buffer
```

The implication is that different encryption methods must be chosen based on use cases and OFB seems to be an algorithm which is least affected by corrupted bytes, because the neighbouring bits of the corrupted bit is unaffected.

Task #4: Pseudo random number generation

In this task we learned how to generate good random numbers, strong enough for security standards.

After running the command to see kernels entropy

```
justin@justin-Precision-3520 10:40:06 ~/Downloads/lab 1
$ cat /proc/sys/kernel/random/entropy_avail
3876
```

After moving the cursor and typing some stuff the entropy number seems to be changing every time

```
justin@justin-Precision-3520 10:40:06 ~/Downloads/lab 1
$ cat /proc/sys/kernel/random/entropy_avail
3876

justin@justin-Precision-3520 11:44:24 ~/Downloads/lab 1
$ cat /proc/sys/kernel/random/entropy_avail
3893

justin@justin-Precision-3520 11:48:00 ~/Downloads/lab 1
$ cat /proc/sys/kernel/random/entropy_avail
3896

justin@justin-Precision-3520 11:48:07 ~/Downloads/lab 1
$ cat /proc/sys/kernel/random/entropy_avail
3901
```

For next task we explore `/dev/random`. Running `head -c 16 /dev/random | hexdump` several times we can see it generates 16 bytes of sudo random numbers very quickly.

```

justin@justin-Precision-3520 11:51:54 ~/Downloads/lab 1
$ head -c 16 /dev/random | hexdump
00000000 bcd0 ce00 0fec c867 2331 5ab4 5dd3 2a36
00000010

justin@justin-Precision-3520 11:51:55 ~/Downloads/lab 1
$ head -c 16 /dev/random | hexdump
00000000 c6bb ced2 b28b 7ac0 be72 8176 8c6e 8a20
00000010

justin@justin-Precision-3520 11:51:56 ~/Downloads/lab 1
$ head -c 16 /dev/random | hexdump
00000000 36ba 69aa 9bf1 5c20 f4a2 cb39 5fe7 774c
00000010

justin@justin-Precision-3520 11:51:56 ~/Downloads/lab 1
$ head -c 16 /dev/random | hexdump
00000000 75aa 1d79 aad3 0f9a c2ed 1db7 09e8 8719
00000010

justin@justin-Precision-3520 11:51:57 ~/Downloads/lab 1
$ head -c 16 /dev/random | hexdump
00000000 e36e bd78 8a9f 3cef 5d51 d232 4b60 e151
00000010

```

Once it gets blocked we can unblock it providing input to the system. Which is by moving the mouse and typing anything into the keyboard.

Using `/dev/urandom` to generate unblocking pseudo random numbers

```
head -c 1600 /dev/urandom | hexdump
```

Ran this command a lot of times and it never blocked.