Matthew Kachar

COSC4931

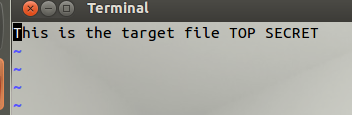
Project 8

Project 8 Crypto Lab – Secret-Key Encryption

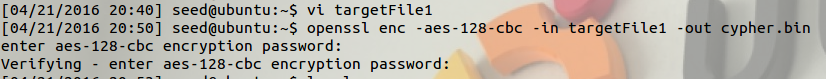
Objective: Become familiar with in secret-key encryption as well as building a foundation for understanding encryption algorithms, modes, paddings, and initial vector, as well as writing programs to encrypt and decrypt.

For this lab it is necessary to install the OpenSSL software, a commercial application for transmitting sensitive data along a network. SSL stands for secure socket layer, and usually contains a library of encryption functions

For the first task we encrypt a simple text file and test several encryption functions. Here is the original plain text file being encrypted

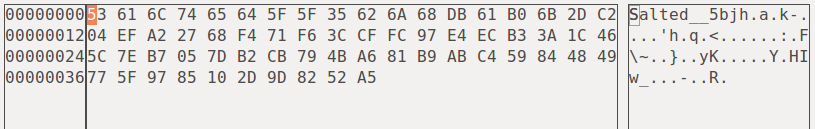


The *openssl* command to encrypt the file using one of the AES functions.

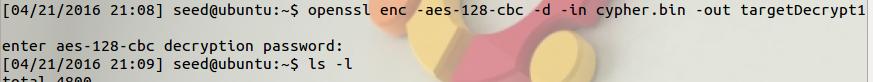


.

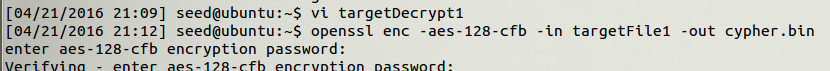
Looking at cypher.bin inside the GHex application after using the aes-128-cbc function.

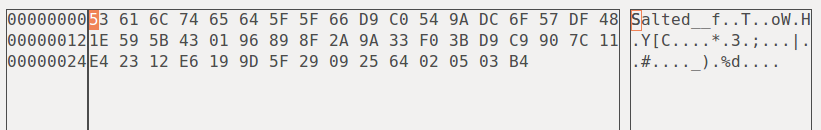


To read this encrypted file we can reverse the openssl command *enc* adding the –d (decrypt) operator.

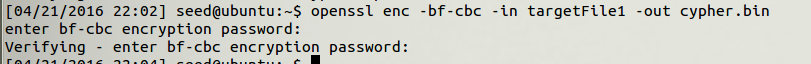


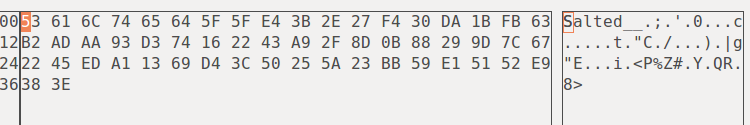
We can try the same process with a couple other encryption functions; first aes-128-cfb.





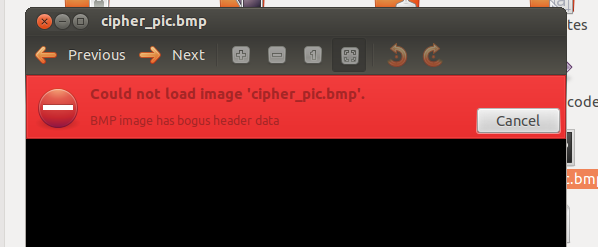
Second the –bf-cbc cipher.



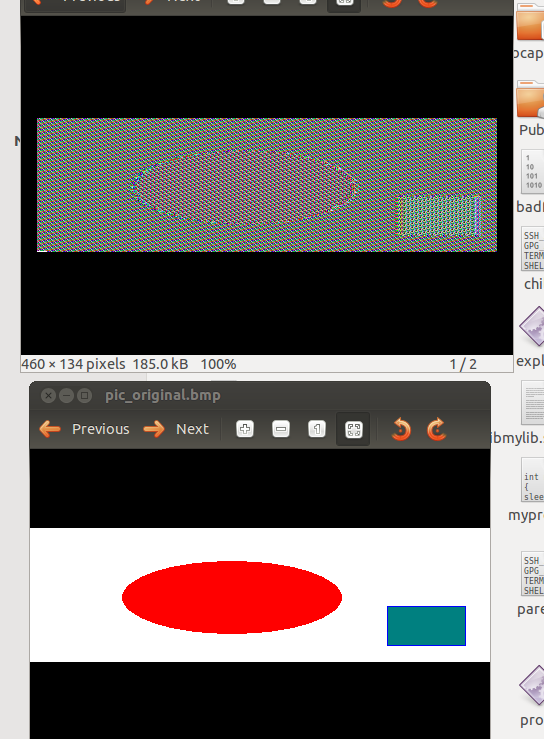


We can observe how the hex representation of the encrypted file changes with each function, however the cipher seems to be somewhat similar between the three since the binary files are very close. The first five to six characters are the same for each.

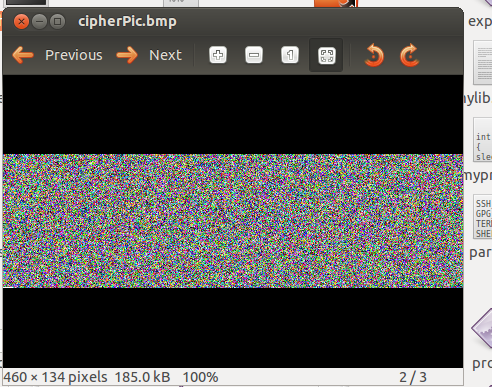
In the second task we encrypt the file pic\_original.bmp. Trying to open the encrypted file gives us the following prompt since the header of the picture is change during encryption so we must correct the header file such that the picture viewing application recognizes the file as a picture. This involves opening the binary file in GHex and changing the first 56 bytes to match the original.



The top image is the encrypted file and the lower is the original file after using the ECB cipher.



The following image is the encrypted image from the CBC cipher method



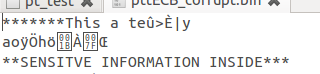
Form the ECB cipher the shapes are still discernable from the encrypted image and the colors are still distinguishable under the sort of fuzzy layer. The CBC method proves to be much better for the .bmp file since the encrypted image is just white noise. This make sense because the CBC method uses an XOR operation in addition to the simple 64 bit cipher. The result is a much stronger encrypted file.

In the next task we show how a corrupted encrypted file can create problems when trying to decrypt. The following screen shots are respectively the original text file and decrypted text files after corrupting the 30th byte of each method— ECB, CBC, CFB and OFB cipher methods.

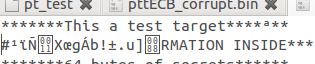
Original



CBC ECB

CFB OFB

The CBC suffered changes to 32 bytes of the original text while the ECB only had 16 bytes changed but the result seems to produce undefined characters. The CFB method produced 32 bytes of encrypted characters but seemed to have a mismatch in the bit length from the original. The OFB only had one byte changed.

To see the padding maintained during encryption we can look at the size of the plaintext file and that of the encrypted file and compare their sizes. The encrypted file should pad in 64 bit blocks.

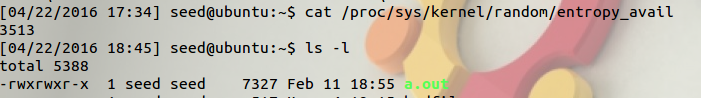
Both ECB and CBC methods pad the 23 byte text file rounding up to the next block.



The CFB and OFB methods do not pad the text file



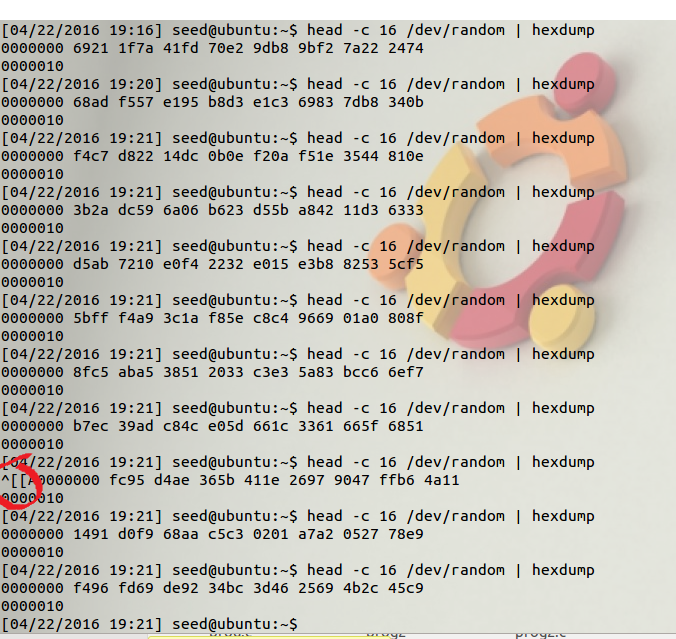
This tasks observes how random numbers are generated for the purpose of security. One of these methods uses kernel entropy as a basis for random number generation. Linux uses mouse movement, keyboard input, and interrupts to calculate random numbers of bits. Using the *cat ../entropy\_avail* command we can see this calculated value change while performing other task in the terminal.



The first time the command is made the number that comes up is 3513 then after calling the list command and performing some directory changes the value changes.



For the next part we can *hexdump* the random data pool which is where the measured entropy data is stored. The following screen shot is of the data and where the block had started before other movement (up arrow) was logged and the block lifted.



The hexdump operator in the *urandom* directory does not allow the block with either 16 or 1600 bytes even after running the command dozens of times.