[[1]](#footnote-1)

Secret-Key Encryption: Exploration with OpenSSL

David Warren, CS428, dlwarren2@crimson.ua.edu

*Abstract*—This lab is about the use of OpenSSL to encrypt/decrypt files and the implementation of the OpenSSL API “EVP” to figure out a private key, given a number of hints at the start. The purpose of this lab is the academic exploration of encryption algorithms and modes and how to practically implement them. There were two tasks accomplished: First, the exploration of OpenSSL to encrypt a file using three separate cipher types; Second, a program that given a known plaintext & ciphertext can find a private key that fits parameters given in the lab description. With the knowledge from the first task and from the sample EVP code, the program in the second task was able to successfully find the private key.

# INTRODUCTION and Lab Definition

T

HIS lab is an exploration of the concepts of secret-key encryption and the OpenSSL library. It allows students the chance to get a familiar grasp on encryption algorithms through first-hand experience. This is accomplished by using tools and writing programs to encrypt/decrypt messages. Beginning with Task 1, the first program used is the “openssl enc” command to encrypt/decrypt files [1]. A plaintext file was encrypted and then looked at using GHex to observe the changes in length and value. In Task 2, a program was constructed with the OpenSSL API “EVP.” “EVP” contains cipher routines which serve as a high-level interface for certain symmetric ciphers, in our case, aes-128-cbc [2]. This program reads in possible keys from an English word list, encrypts the given plaintext with aes-128-cbc, and compares the produced ciphertext with the known ciphertext until it finds a match and thus, our secret key. The result was then verified with “openssl enc” command.

# Lab Setup

## Creating the Lab Environment

The first step in this lab was to create a suitable lab environment to conduct our exploration. To do this, an Ubuntu VM was created using VirtualBox and an Ubuntu image from SEED security labs’ SEEDUbuntu12.04. A setup document was referenced to create the VM from the pre-built VM image [3]. A new VM was created and then the SEEDUbuntu12.04 was used as the image.

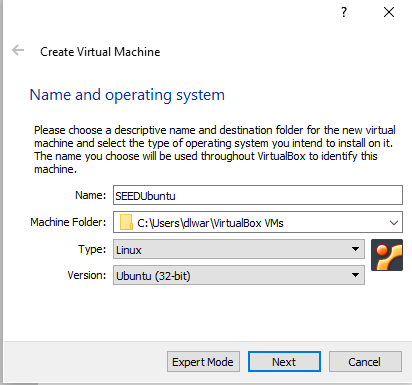


Fig. 1. A new Linux VM is created using the Ubuntu-32 bit OS.

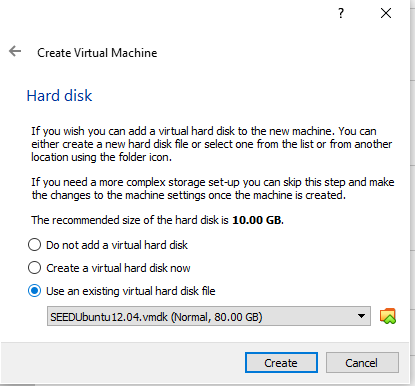


Fig. 2. The existing .vhdk was chosen downloaded from SEED

After this was completed, a suitable lab environment had been created. Note below the VM specifications.

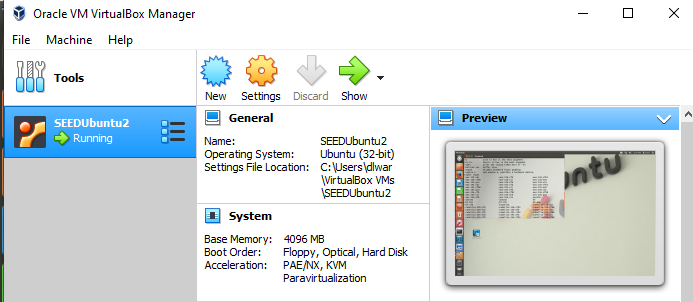


Fig. 3. The final VM specifications are checked for correctness before beginning the lab.

## Installing Necessary Components

Now that the VM has been setup and verified to be the correct specifications, the necessary setup OpenSSL must be conducted. The openssl binaries already exist on the pre-built image, so now all that needs to be done is to configure and install them. This was achieved by following the Lab Description [4]. First, openssl was configured, then the makefile was used to make, test, and install the final product following the commands given below.

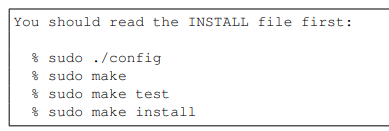


Fig. 4. Commands run to finalize setup and installation of openssl.

The option of downloading a separate hex editor is allowed, but for this lab only the standard GHex editor was used that is already installed on the pre-built image of the VM from SEED.

# Task 1

## Summary of task

This task is simply an exploratory one in which openssl is used to explore different behaviors of encryption algorithms and modes. The man page is used for reference in constructing the commands [1].

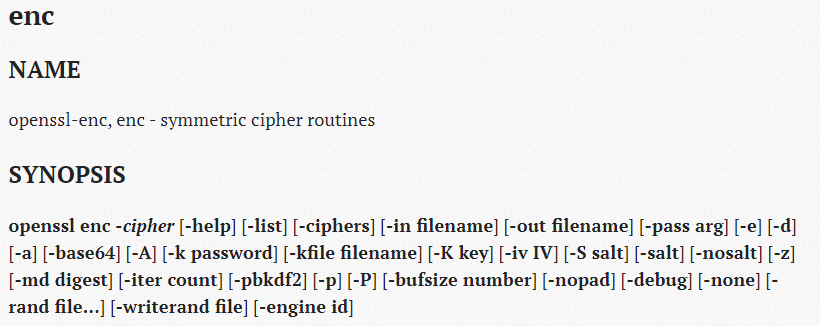


Fig. 5. Synopsis of the man page of the openssl enc command.

*­*The flag “*-cipher*” was replaced with various cipher types to observe the differences in output. The input file is the same as the input for Task 2, but in a text file. The plaintext can be seen below. Note that the text file is also 21 bytes. The input and output files are all viewed using the GHex viewer.

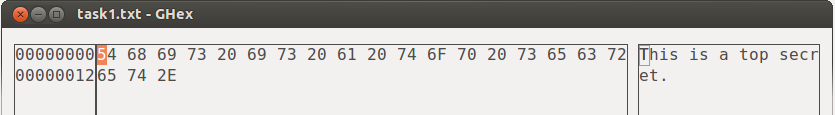


Fig. 6. Input file for openssl enc commands to be used.

## aes-128-cbc

The first cipher type that was used with the openssl command was -aes-128-cbc.

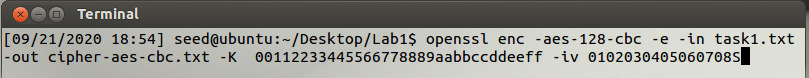


Fig. 7. openssl enc command for aes-128-cbc

Note that the key and initialization vector used are the same from the example in the lab description. Also note the encryption flag “-e” is specifically stated. Below you may see the output from the given command.

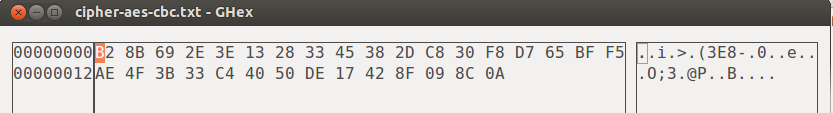


Fig. 8. openssl enc result for aes-128-cbc

It was observed that the resulting file is 32 bytes in length. This concurs with expectations of the cipher (16+16 bytes) for an input text that is 21 bytes in length of the aes-128-cbc cipher. There is also a null character that concludes the text file, while there is none in the input file.

## bf-cbc

The second cipher type that was used with the openssl command was -bf-cbc.

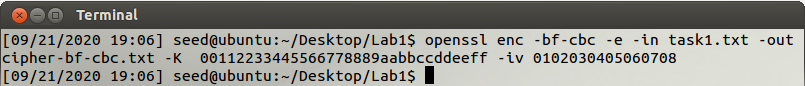


Fig. 9. openssl enc command for bf-cbc

Note that the key and initialization vector are the same for part A of this section. Just the same as above, the input file is the same as well as the “-e” flag is specifically stated. Below you may see the output from the command above.

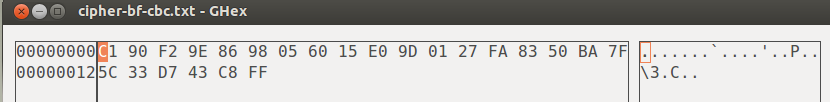


Fig. 10. openssl enc result for bf-cbc

It was observed that the resulting file is 24 bytes in length which is different from the previous block cipher. It was also observed that there was no null terminating character at the end of this output file. Perhaps the null character in the previous cipher is coincidental. The output length of the cipher also concurs with expectations of the cipher (8 + 8 + 8) of the blowfish cipher.

## des-cbc

The third and final cipher type that was used with the openssl command was -des-cbc.

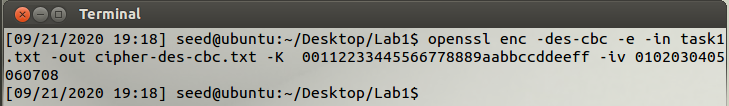


Fig. 11. openssl enc command for des-cbc

Again, note that the key, initialization vector, encryption flag, and input text are the same as the above examples. Below you may see the output from the command above.

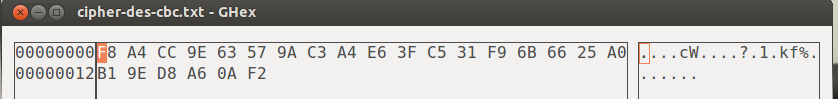


Fig. 12. openssl enc result for des-cbc

It was observed that this resulting file too had 24 bytes in length. As well as that there was no terminating null character. The output length matches the expectations of this cipher as well (8 + 8 + 8) as the DES cipher works in blocks of 64 bits.

# Task 2

## Summary of task

The goal of this task is to use openssl’s crypto library to encrypt/decrypt messages in a program. This will be accomplished via the EVP interface. After going over the sample code using the EVP API [2], we are ready to begin the task. In this task, a known plaintext and known ciphertext are provided along with various hints about the key and initialization vector. First, the initialization vector is known to be all zeros but not the ASCII zero. Next, we know that the key is in plain English so an English word list will be useful in finding the key and that the key is shorter than 16 characters. If a test word is used as a possible key that is shorter than 16 characters, it should be padded with spaces at the end to form a key of 128 bits [4]. To use the code written for Task 2, a makefile has been created (see below). Simply run “make” and then run the executable it generates. The program will use the specified plaintext and ciphertext internally initialized, and cross-reference encrypted plaintext English words from the word list to find a ciphertext match. This will then produce the correct key in plaintext and its relative ciphertext.

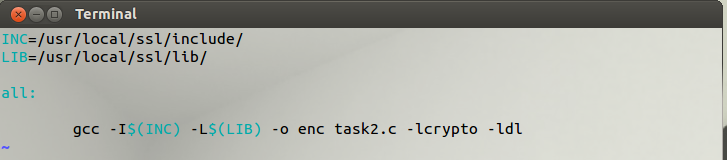


Fig. 13. Makefile for Task 2

## Process of Task 2

After the makefile is created for the task, we are ready to begin the programming for Task 2. The first goal is to setup File I/O for the English word list. A 2D array is used to store all words from the list. A dynamic array is not needed as we know the length of our specific list to be 25,143 words. This will be our first dimension. Next, we must assign a dimension for the length of the words. While we know the key cannot be more than 128 bits, there exist some words in the list that are longer. Therefore, we must make the buffer large in order to read in all words. An arbitrary value of 100 is chosen for the length. This is verified to be sufficient using regular expressions on the English word list. Through some quick use of regular expressions in notepad++, the longest word in the list is 22 characters long: “electroencephalography.” The regular expression used is “^.{22,}” to find any words of length greater or equal to 22. The same expression with a length of 23 finds no matches. Therefore, we have determined a buffer size of 100 to be safe to read in all words in the list.

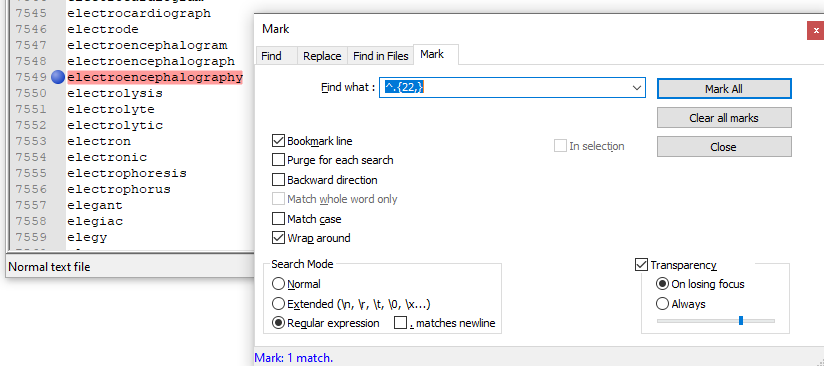


Fig. 13. Figuring out the buffer for our array of English words.

Now that we have safe dimensions for our 2D array, we can safely read in the entire list of words using File I/O which can be seen below.



Fig. 14. File I/O to safely read in entire English word list.

Note that the buffer and word count are defined. Now that we have our input, we may sanitize it by padding words that are shorter than 128 bits with spaces. Words longer than 128 bits have no need to be sanitized. The first 128 bits of the possible key will be attempted to match the encryption but the match will fail. This is one area of the code that can be improved for performance as all words greater than 128 bits can be ignored. For the purposes of this lab, however, the difference in performance is minimal for a list size of 25,143 words. If a larger English word list were to be used, this change would prove beneficial. See below the code to sanitize input.

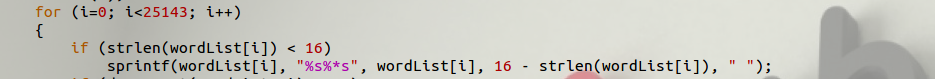


Fig. 15. Sanitize input by padding shorter words with spaces.

After sanitizing the input, we can now attempt an encryption on the English word. To do this, a helper function doEncrypt(char wordList[WORDS][BUF], int index) is used. This function returns -1 on error, 0 on no match found, and 1 on match found. The first parameter wordList[WORDS][BUF] is the 2D array that stores our English words and the second parameter is the index in that array we are currently on.

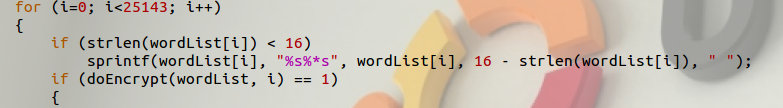


Fig. 16. Attempt encryption on each possible key.

Now the program enters the work loop to find the key. Using the sample code as reference [2] the EVP API is utilized to encrypt our words from the English word list. The variables and cipher context are initialized and we begin our encryption using EVP\_CipherUpdate and EVP\_CipherFinal\_ex. The first parameter is copied as an unsigned char array to be used as a key and the initialization vector is set to be 128 bits of 0, and not the ASCII ‘0.’ Finally, the cipher context is cleaned up.

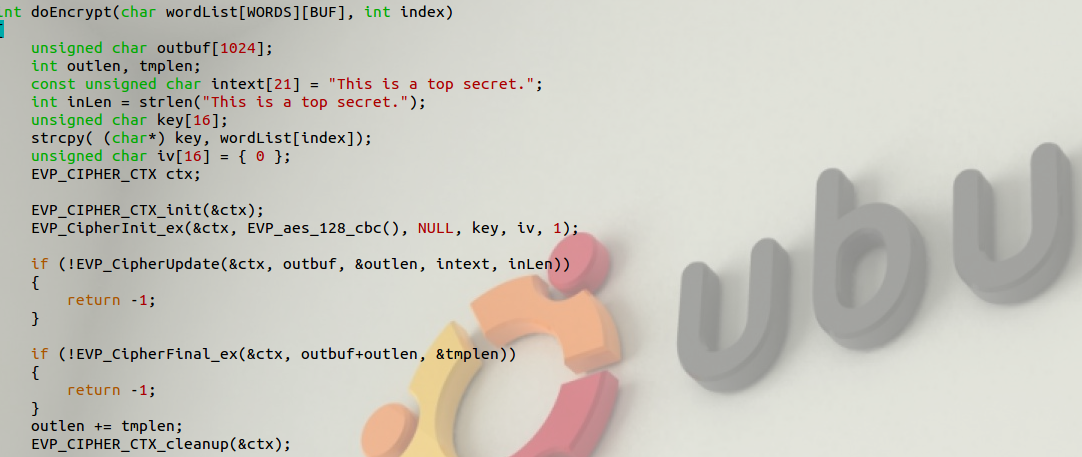


Fig. 17. Code for encryption of known plaintext using key from English word list

Now, we have our ciphertext from the encryption stored in outbuf[1024] and we are ready to compare to the known ciphertext. We only need to compare the first 32 bytes, as that is the length of the known ciphertext.

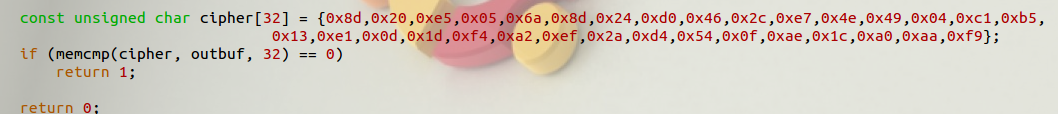


Fig. 18. Test for a match between generated ciphertext and known ciphertext.

If we have found a match, the condition in the for loop in Fig. 16 is satisfied and the for loop breaks on successful match. The plaintext key is printed along with the hex value of the key. If no match is found, the program states there is no match.

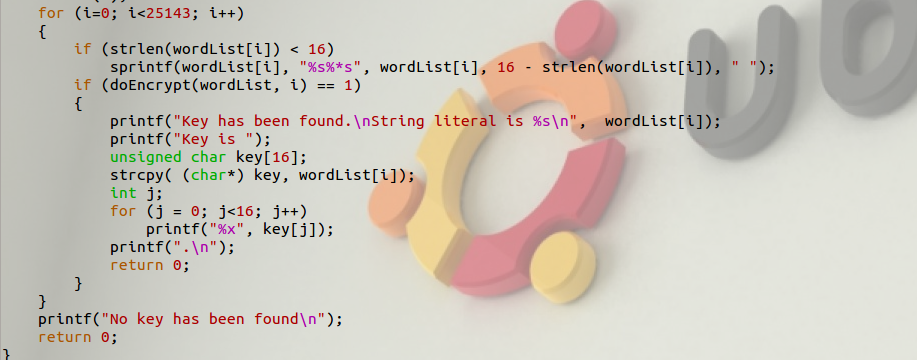


Fig. 19. Test for a match between generated ciphertext and known ciphertext.

After this, the program terminates.

# Results and Conclusion

We may now verify the results of our program. The results are displayed below of the program run on the provided English word list.

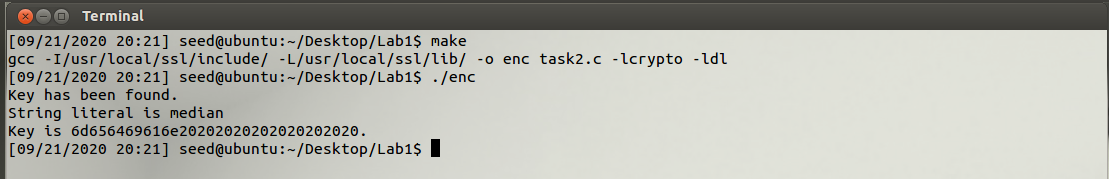


Fig. 20. A match has been found with the value of “median” padded with spaces to make a key of 128 bits.

We have run the make file and run the executable. After this, a match has been found. We now will verify using the openssl enc command with the “-d” decrypt flag using -aes-128-cbc, the cipher known to have been used to generate the known ciphertext. See the command used below.



Fig. 21. openssl decryption using given cipher as input, the found key’s hex value as the key, and the initialization vector of all zeros.

See below the file used for input to the decryption command.

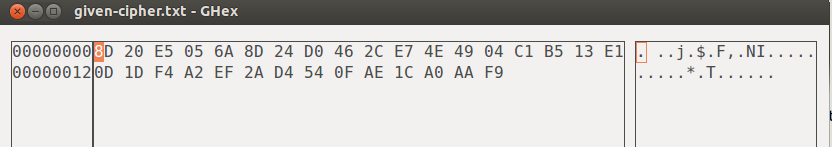


Fig. 22. Cipher generated from an openssl encryption command from task1.txt using aes-128-cbc that matches the given ciphertext.

This matches with the given ciphertext in the Lab Description. We will now look at the output of the decryption command.

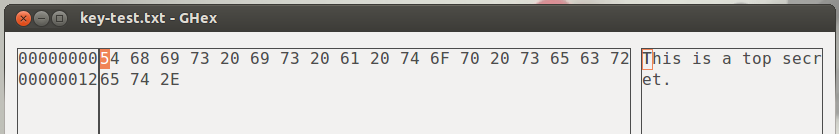


Fig. 23. openssl decryption output file. The produced plaintext from the command matches the known plaintext.

This produced plaintext matches the known plaintext from the Lab Description. Our program has succeeded in finding a key from the English word list as the key and initialization vector produce the correct known plaintext when decrypting the known ciphertext with aes-128-cbc. In this lab, I learned how to use the openssl enc commands to encrypt and decrypt files. Using this knowledge, I was able to verify the correct output of my program. I also learned to use the EVP API to write a program to encrypt plaintext. Using this knowledge, I was able to use the hints in the Lab Description, along with the known plaintext and ciphertext to find the secret key used in this encryption. In the future, there are many possible optimizations for my program. Further sanitization of the English world list would be necessary for larger files by optimizing the words for known length of the key. Next, multithreading could be used to speed up the encryptions of the words, terminating all child processes on finding matching ciphertext. This lab provided good context for the prospect of a brute-force attack and the importance of keeping information from attackers. This lab was relatively easy to understand. The provided documentation made Task 2 straightforward. I was stuck for a little while when my encryption from my program did not match the output of the openssl command when using the same ciphertext, key, and initialization vector, but I then realized that my plaintext had a terminating null character in the text file I was using to test. After this, my command output and code matched and I knew I was ready to look for the key. Task 1 really helped me to verify my code for Task 2. My knowledge of ciphers from CS 442 helped me to understand the output of the ciphers I used in Task 1. Overall, I feel that I have learned how to successfully implement encryption using EVP, however, I am not sure a brute-attack on a more complicated cipher like AES-256 or if this measure of attempt would be successful on asymmetric ciphers.

References

[1] OpenSSL Foundation, “OpenSSL,”

/docs/man1.1.1/man1/openssl-enc.html. Available: https://www.openssl.org/docs/man1.1.1/man1/openssl-enc.html

[2] OpenSSL Foundation, “OpenSSL,” /docs/man1.0.2/man3/EVP\_EncryptInit.html. Available: https://www.openssl.org/docs/man1.0.2/man3/EVP\_EncryptInit.html.

[3] “How to use VirtualBox to Run Our Pre-built VM Image?” Available: http://www.cis.syr.edu/~wedu/seed/Documentation/Ubuntu12\_04\_VM/UseVirtualBox.pdf.

[4] Crypto Lab - Secret-Key Encryption. Available: http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Crypto/Crypto\_Encryption/.

1. [↑](#footnote-ref-1)