Applied Cryptography Lab-04 Manual

29 September 2022 00:58

Prerequisites

- Visit the following website and download the lab manual andLabsetup.zip file https://seedsecuritylabs.org/Labs_Crypto/Crypto_Encryption/#:~:text=The%20leive%20of%20this,and%20initial%20vector%20(IV)

Task 1: Encryption using Different Ciphers and Modes

In this task, we will play with various encryption algorithms and modes.
You can use the following openssl enc command to encrypt/decrypt a file. To see the manuals, you can type man openssl and man enc.

```
$ openssl enc -ciphertype -e -in plain.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -bf-cbc, -aes-128-cfb, etc.
In this task, you should try at least 3 different ciphers. You can find the meaning of the command-line options and all the supported cipher types by typing "man enc".

Task 2: Encryption Mode – ECB vs. CBC

The file **pic_original.bmp** is included in the **Labsetup.zip** file, and it is a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes

```
$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out
```

```
$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out
pic_ecb.bmp -K 00112233445566778889aabbccddeeff -iv
0102030405060708
$ openssl enc -aes-128-cbc -e -in pic_original.bmp -out
pic_cbc.bmp -K 00112233445566778889aabbccddeeff -iv
0102030405060708
```

Now, let us view the encrypted files as images by attaching .bmp headers to them

```
$ head -c 54 pic_original.bmp > header
$ tail -c +55 pic_ecb.bmp > body_ecb
$ cat header body_ecb > new_ecb.bmp
$ tail -c +55 pic_cbc.bmp > body_cbc
$ cat header body_cbc > new_cbc.bmp
```

Display the encrypted picture using a picture viewing program and explain your observations.

Task 3: Padding

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. The PKCS#5 padding scheme is widely used by many block ciphers.
We will conduct the following experiments to understand how this type of padding works

Step 1
Create 3 files of length 5 bytes, 10 bytes and 16 bytes respectively.

```
$ echo -n "12345" > f1.txt
$ echo -n "1234567890" > f2.txt
$ echo -n "1234567890abcdef" > f3.txt
```

Step 2
We then use `"openssl enc -aes-128-cbc -e"` to encrypt these three files using 128-bit AES with CBC mode. Please describe the size of the encrypted files

```
$ openssl enc -aes-128-cbc -e -in f1.txt -o f1.bin -K
00112233445566778889aabbccddeeff  -iv  0102030405060708
$ openssl enc -aes-128-cbc -e -in f2.txt -o f2.bin -K
```

```
$ openssl enc -aes-128-cbc -e -in f2.txt -o f2.bin -K
00112233445566778889aabbccddeeff  -iv  0102030405060708
$ openssl enc -aes-128-cbc -e -in f3.txt -o f3.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Step 3

Decrypt the encrypted files with a "-nopad" flag to disable padding, and use hexdump (xxd) to view the contents of the decrypted files

```
$ openssl enc -aes-128-cbc -d -in f1.bin -o p1.txt -K
00112233445566778889aabbccddeeff -iv
0102030405060708 -nopad
$ openssl enc -aes-128-cbc -d -in f2.bin -o p2.txt -K
00112233445566778889aabbccddeeff -iv
0102030405060708 -nopad
$ openssl enc -aes-128-cbc -d -in f3.bin -o p3.txt -K
00112233445566778889aabbccddeeff -iv
0102030405060708 -nopad

$ xxd p1.txt
$ xxd p2.txt
$ xxd p3.txt
```

Task 4: Error Propagation – Corrupted Cipher Text

1. Create a text file with at least 1000 bytes 2. Encrypt the file using the AES-128 cipher `$ openssl enc -aes-128-cbc -e -in error.txt -o  enc.bin -K 00112233445566778889aabbccddeeff -iv  0102030405060708`
3. Unfortunately, a single bit of the 55th byte in the encrypted file got corrupted. You can achieve this corruption using the bless hex editor.
   ```
   $ bless enc.bin
   ```
4. Decrypt the corrupted ciphertext file using the correct key and IV.

```
$ openssl enc -aes-128-cbc -e -in enc.bin -o
err_dec.txt -K
00112233445566778889aabbccddeeff -iv
0102030405060708
```

Please answer the following question:

Please answer the following question:
How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

Task 5: Initial Vector (IV) and Common Mistakes

Task 5.1: IV Experiment

A basic requirement for IV is uniqueness, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Please describe your observation, based on which, explain why IV needs to be unique.

Task 5.2: Common Mistake: Use the Same IV

One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode.
Assume that the attacker gets hold of a plaintext (P1) and a ciphertext (C1), can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content of P2 based on C2, P1, and C1.

```
Plaintext (P1): This is a known message!
Ciphertext (C1):
a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159
Plaintext (P2): (unknown to you)
Ciphertext (C2):
bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

The file `sample_code.py` given inside `Labsetup.zip` will help you with it. Change the variables' values in the script to reflect the problem statement.

```
$ python3 sample_code.py
```

Task 7: Programming using the Crypto

Task 7: Programming using the Crypto Library

The C language provides APIs to commands used throughout this lab to encrypt/decrypt data using symmetric key block and stream ciphers.
In this task, you are given a plaintext and a ciphertext, and your job is to find the key that is used for the encryption, by writing a C code to automate your job.

Information available:

• The aes-128-cbc cipher is used for the encryption. • The key used to encrypt this plaintext is an English word shorter than 16 characters; the word can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), pound signs (#: hexadecimal value is 0×23) are appended to the end of the word to form a key of 128 bits.

Code:

```
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define DO_ENCRYPT 1
#define KEY_LEN 16
#define OUT_BUF_SIZE (1024 + EVP_MAX_BLOCK_LENGTH)
unsigned char ciphertext[] =
{0x8d,0x20,0xe5,0x05,0x6a,0x8d,0x24,0xd0,0x46,0x2c,0xe7,0x4e,0x49,0
x 04,0xc1,0xb5,
0x13,0xe1,0x0d,0x1d,0xf4,0xa2,0xef,0x2a,0xd4,0x54,0x0f,0xae,0x1c,0x
a 0,0xaa,0xf9};
unsigned char *plaintext = "This is a top secret.";
int pad_space(unsigned char* key, int key_len){
    if(key_len < KEY_LEN){
        for(int i=key_len; i< KEY_LEN; i++){
            key[i] = 0x20;
        }
    }
}
int encrypt(unsigned char* buf_in, int buf_in_len,
    unsigned char* buf_out, int* buf_out_len, unsigned char* key,
int key_len){
```

```
int key_len){
    int outlen;
    EVP_CIPHER_CTX *ctx;
    ctx = EVP_CIPHER_CTX_new();
    pad_space(key, key_len);
    unsigned char iv[] =
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    EVP_CIPHER_CTX_init(ctx);
    EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, NULL, NULL,
DO_ENCRYPT);
    OPENSSL_assert(EVP_CIPHER_CTX_key_length(ctx) == 16);
    OPENSSL_assert(EVP_CIPHER_CTX_iv_length(ctx) == 16);
    EVP_CipherInit_ex(ctx, NULL, NULL, key, iv, DO_ENCRYPT);
    EVP_CipherUpdate(ctx, buf_out, &outlen, buf_in, buf_in_len);
    *buf_out_len = outlen;
    EVP_CipherFinal_ex(ctx, buf_out + outlen, &outlen);
    *buf_out_len += outlen;
    EVP_CIPHER_CTX_cleanup(ctx);
    return 1;
}
/*
* return 0 as equal, others as different
*/
int compare(unsigned char* buf1, int buf1_len, unsigned char* buf2,
int buf2_len){
    if(buf1_len <=0 || buf2_len <=0 || buf1_len != buf2_len){
        return 1;
    }
    for(int i=0; i< buf1_len; i++){
        if(buf1[i] != buf2[i]){
            return 1;
        }
    }
    return 0;
}
int check_if_is_key(char* key){
    unsigned char outbuf[OUT_BUF_SIZE];
    int outbuf_len = 0;
    encrypt(plaintext, strlen(plaintext), outbuf, &outbuf_len, key,
strlen(key));
    return compare(ciphertext, sizeof(ciphertext), outbuf,
outbuf_len);
}
```

```
int main(){
    char line[256];
    FILE *fp = fopen("words.txt" , "r");;
    if(fp != NULL){
        while( fgets (line, 255, fp) != NULL ) {
            line[strcspn(line, "\r\n")] = 0;
            if ( 0 == check_if_is_key(line) ){
                printf("Got key: %s\n", line);
                break;
            }
        }
        fclose(fp);
```

```
        fclose(fp);
    }else{
        perror("Error open words file\n");
        return(-1);
    }
}
```

Compilation and execution:

```
$ gcc code.c -o findkey -lcrypto
$ ./findkey
```