Coursework 1: Secret key Encryption

Task1: Encryption using different ciphers and modes.

Openssl provided various type of ciphers and modes. It have very simple command easy to learn, and simple to use. Any file type can be encrypted using scripts below.

#!/bin/bash

#!/Various encryption types

openssl enc -aes-128-cbc -e -in ./some.txt -out ./cipher\_aes\_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb1 -e -in ./some.txt -out ./cipher\_aes\_cfb1.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-ofb -e -in ./some.txt -out ./cipher\_aes\_ofb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -bf-cbc -e -in ./some.txt -out ./cipher\_bf\_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -bf-ofb -e -in ./some.txt -out ./cipher\_bf\_ofb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -bf-ecb -e -in ./some.txt -out ./cipher\_bf\_ecb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -des-cbc -e -in ./some.txt -out ./cipher\_des\_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -des-cfb1 -e -in ./some.txt -out ./cipher\_des\_cfb1.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -des-ecb -e -in ./some.txt -out ./cipher\_des\_ecb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Code1: Openssl command for encryption.

This scripts contain 3 different cipher type like AES (Advanced Encryption Standard), BF (Blowfish Encryption) and DES (Data Encryption Standard) with other 3 different encryption mode like ECB (Electronic codebook), CBC (Cipher-block chaining) and OFB (Output feedback).

Task2: Encryption mode ECB vs. CBC.

Encryption modes are operators that provide more securely transform of data. Most of block cipher come with a lot of mode. Block cipher itself only decrypt or encrypt data in fixed length of bit. If the target data is large, there are high possibility that the block cipher will contain some patterns and become vulnerable to attacker.

In this task we will compare 2 of encryptions mode ECB (Electronic codebook) and CBC (Cipher-block chaining) by encrypt a picture with both modes of the same cipher. Simple Openssl commands for doing this are below.

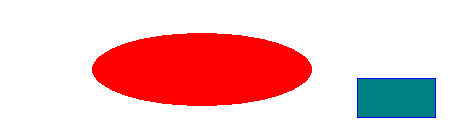
#!Encrypt picture

openssl enc -aes-128-cbc -e -in ./pic\_original.bmp -out ./pic\_cbc.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-ecb -e -in ./pic\_original.bmp -out ./pic\_ecb.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Code2: Openssl command using CBC and ECB encryption mode.

After edit encrypted files header with Hex editor, encrypted are now recognised as normal BMP file and can open with any picture viewer.



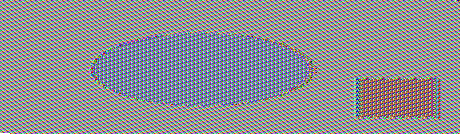


Figure1: Results of encryption (upper left: ECB, lower left: CBC and original on the right).

As expected, a result from ECB mode still keep identical layout as the original picture. ECB actually encrypt a block of plain data block by block; for a picture each byte contain different pixel colour, encrypted block of red colour are just going to change them to another pixel colour. On another hand, CBC used previous block of cipher data to encrypt next block of plain data result in randomly mixed in various pixel colour and leave no pattern behind.

Task3: Encryption mode – corrupted cipher text.

Unlike plain data file, a cipher text may not be able to decrypt back into a plain data even if just a tiny parts were damaged. This task compare recovery ability of 4 encryption modes ECB (Electronic codebook), CBC (Cipher-block chaining), CFB (Cipher feedback) and OFB (Output feedback).

In this task, the target file contain data at least 64bytes long and encrypted with Openssl command as below.

“CentOS is an Enterprise-class Linux Distribution derived from sources freely provided to the public by Red Hat, Inc. for Red Hat Enterprise Linux. CentOS conforms fully with the upstream vendors redistribution policy and aims to be functionally compatible.”

Figure2: Long plain text (plain.txt).

#!Encrypt long text files

openssl enc -aes-128-ecb -e -in ./plain.txt -out ./cipher\_long\_ecb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cbc -e -in ./plain.txt -out ./cipher\_long\_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb -e -in ./plain.txt -out ./cipher\_long\_cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-ofb -e -in ./plain.txt -out ./cipher\_long\_ofb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Code3: Openssl encryption command for 4 different mode.

By using Hex editor, all 4 encrypted files are corrupted at 30th digit. Now let’s look at the algorithm of encryption modes first. ECB is the likely to be easily recovered with less damage to the plain data because it is the simplest mode of all as we discuss earlier in Task2. The second must be OFB because OFB do not require a cipher text block to decrypt the next block. So damaging some cipher text block will not have much effect and it is will damage only on its own block. CBC and CFB decryption method are quite similar. It’s hard to tell which one will recover more data.

#!Decrypt corrupted files

openssl enc -aes-128-ecb -d -in ./cipher\_long\_ecb\_corrupted.bin -out ./recovered\_ecb.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cbc -d -in ./cipher\_long\_cbc\_corrupted.bin -out ./recovered\_cbc.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb -d -in ./cipher\_long\_cfb\_corrupted.bin -out ./recovered\_cfb.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-ofb -d -in ./cipher\_long\_ofb\_corrupted.bin -out ./recovered\_ofb.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Code4: Openssl for decrypt 4 corrupted files.

After encryptions all 4 corrupted files, the results are below in order.

1St OFB

“CentOS is an Enterprise-class Linux Distributi\_n derived from sources freely provided to the public by Red Hat, Inc. for Red Hat Enterprise Linux. CentOS conforms fully with the upstream vendors redistribution policy and aims to be functionally compatible.”

2nd ECB

“CentOS is an Enterprise-class L·iÅ·K˜\*¿èpŸš\_Ž n derived from sources freely provided to the public by Red Hat, Inc. for Red Hat Enterprise Linux. CentOS conforms fully with the upstream vendors redistribution policy and aims to be functionally compatible.”

3rd CFB

“CentOS is an Enterprise-class Linux DistributiÏg÷Ì%ŽÔ×m¥\_‚ources freely provided to the public by Red Hat, Inc. for Red Hat Enterprise Linux. CentOS conforms fully with the upstream vendors redistribution policy and aims to be functionally compatible.”

4th CBC

“CentOS is an Enterprise-class Lž6âê´Ô ¾”­Ž ?²n derived from Óources freely provided to the public by Red Hat, Inc. for Red Hat Enterprise Linux. CentOS conforms fully with the upstream vendors redistribution policy and aims to be functionally compatible.”

Task4: Reflections on ciphers and modes.

Result from the task3 show that an earlier discussion before the corrupted files were decrypted are almost correct. After return to lecture slide I found that it makes sense for OFB to be the 1st because OFB allow more error correction. The result showed that, it has only one digit missing from the plaintext. So OFB also be the best to use with a stream of data because this property.

CBC seems to be the one that hide the highest degree of information because of its chain encryption but surely it’s not the fastest one to encrypt a very large data.

CFB seems to be a very good choice for a very large data because it did not need to carry a block cipher decryption. Also it faster than OFB because CFB allow a parallel decryption but OFB is not allow.

ECB are only useful with a very small file. It’s easy to understand but should not use in a real life as we discuss before.

Task5: Checksums

[English word list](http://www.macs.hw.ac.uk/~hwloidl/Courses/F21CN/Labs/CryptoI/words.txt) SHA1 checksum: b3470280f84575a3db3ec3a6b9df2681ee0f5a18

[Plain text file](http://www.macs.hw.ac.uk/~hwloidl/Courses/F21CN/Labs/CryptoI/some.txt) SHA1 checksum: 0b6f3556e8773a3e7c0ed31c634b9fd2a108adcc

[Cipher text file](http://www.macs.hw.ac.uk/~hwloidl/Courses/F21CN/Labs/CryptoI/some.aes-128-cbc) SHA1 checksum: 92ce63d9f3495ca005237eb6cca47302b74c574f

3 files with checksum sequences are provided on this task by using shell script below, the result show that all 3 files are untouched. This ensure an accuracy and consistency of data over download process (Data integrity concept). But checksum sequence did not guarantee that these file is safe and come from the originator because checksum sequence is generated from the file. Someone may modify the file before upload the file to a fake website with obviously fake checksum.

#!/bin/bash

#!/Task 5

read -p "Please Enter a SHA checksum sequence > " checksum

read -p "Pleas Enter a file path into terminal > " filepath

echo "Your input SHA checksum is: $checksum"

if [ -f "$filepath" ]

then

I=0

for text in $(openssl sha1 "$filepath")

do

Sum[I]=$text

let I=I+1

done

echo "Your file SHA checksum is: ${Sum[1]}"

if [ "${Sum[1]}"=="$checksum" ]

then

echo "[RESULT]: File and Sequence ARE THE SAME."

else

echo "[RESULT]: File and Sequence ARE NOT THE SAME."

fi

else

echo "[ERROR]: Can not find $filepath."

fi

Code5: Shell script for checksum by input checksum sequence and a target file path.

Task6: Known-plaintext attack

Using files from task5 with some extra information such as cipher type and password that use as a key to encrypt the file (with no salt), we can easily find the right key and decrypt a file.

#!/bin/bash

#!/Task 5

read -p "Pleas Enter an encrypted file path into terminal > " filepath

if [ -f "$filepath" ]

then

while read LINE

do

length=${#LINE}

target=16

if [ $length -lt $target ]

then

echo "Length=$length:Password=$LINE"

openssl enc -aes-128-cbc -d -in ./some.aes-128-cbc -out ./dictionary\_atk.txt -nosalt -pass "pass:$LINE"

if [ "$?" -eq 0 ]

then

while read decrypttext

do

echo "Password:[$LINE] is a key. Result plain text is: $decrypttext" >> ./result.txt

done < ./dictionary\_atk.txt

fi

fi

done < $filepath

else

echo "[ERROR]: Can not find $filepath."

fi

Code6: Shell Script for cracking the key. This script will group the candidate keys and the right key in result.txt file.

The key cracking script using shell script and Openssl library is fairly easy to implement. But it really hard to control outputs such as echo messages for fail attempted to decrypt and in reduced a computing performance of the code. So I went to try cracking by C code. At first it seem simple but I stuck at a strange problem when I tried to call “EVP\_DecryptUpdate ()” function. The function return noting and just stop without any error message. Openssl are quite hard to debug due to the very slim detail document and example. If I could finish the C coding, I quite sure that the performance will be greater and more customable and control than shell script version.

#include <errno.h>

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

// include the EVP headers

#include <openssl/evp.h>

/\*BytesToKey recommened.

You know that

aes-128-cbc

was used to generate the ciphertext from the plaintext. No salting was used during encryption. You also

know that the numbers in the initialisation vector (IV) are all zeros (not the ASCII character â€˜0â€™). Another

clue is that the key, used to encrypt this plaintext, is an English word shorter than 16 characters; the word

that can be found from a typical English dictionary.\*/

int do\_crack(char \*targetFile, char \*dictionary){

unsigned char \*ciphertext = NULL;

unsigned char \*plaintext;

unsigned char key[EVP\_MAX\_KEY\_LENGTH];

unsigned char iv[EVP\_MAX\_IV\_LENGTH];

//unsigned char zeroiv[]="00000000";

int i;

int len = 0;

int plaintext\_len;

int ciphertext\_len;

printf("Cracking aes 128 file: %s\n", targetFile);

printf("Using dictionary file: %s\n", dictionary);

//buffer for password loaded from dictionary line by line.

char password[128];

FILE \*encrypted = fopen(targetFile, "r");

FILE \*file = fopen(dictionary, "r");

while(!feof(file)){

fgets(password, 128, file);

//remove newline command.

if (password[strlen(password)-1] == '\n')

password[strlen(password)-1] = '\0';

//Process only passwords that have length less than 16 digits.

if(strlen(password) < 16){

printf("Plain text password: %s\n", password);

//Transform password into a binary key using openssl.

EVP\_BytesToKey(EVP\_aes\_128\_cbc(),NULL,NULL,password,strlen(password),1,key,iv);

printf("Key: "); for(i=0; i<EVP\_aes\_128\_cbc()->key\_len; ++i) { printf("%02x", key[i]); } printf("\n");

printf("IV: "); for(i=0; i<EVP\_aes\_128\_cbc()->iv\_len; ++i) { printf("%02x", iv[i]); } printf("\n");

//Load cipher text

fseek(encrypted, 0, SEEK\_END);

long encrypted\_len = ftell(encrypted);

ciphertext = malloc(encrypted\_len);

fseek(encrypted, 0, SEEK\_SET);

fread(ciphertext , 1, encrypted\_len, encrypted);

ciphertext\_len = encrypted\_len;

// Create and initialise the context

EVP\_CIPHER\_CTX \*ctx;

ctx = EVP\_CIPHER\_CTX\_new();

EVP\_CIPHER\_CTX\_init(ctx);

// Initialise the decryption.

int decryptResult = EVP\_DecryptInit\_ex(ctx, EVP\_aes\_128\_cbc(), NULL, key, iv);

if(decryptResult == 1){

//Provide cipher text

decryptResult = EVP\_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext\_len);

if(decryptResult == 1){

plaintext\_len = len;

//Finalize

EVP\_DecryptFinal\_ex(ctx, plaintext + len, &len);

plaintext\_len += len;

//print plaintext.

printf("Decrypted text is: %s\n",plaintext);

}else{

printf ("EVP\_DeccryptUpdate failure.\n");

}

}else{

printf ("EVP\_DecryptInit\_ex failure.\n");

}

// Clean up

EVP\_CIPHER\_CTX\_free(ctx);

}

}

fclose(file);

fclose(encrypted);

return 0;

}

// fairly minimal main function, driving the encryption

int main(void){

// output file

static char dictionary[] = "/home/Paul/workspace/filecracker/src/words.txt";

static char filename[] = "/home/Paul/workspace/filecracker/src/some.aes-128-cbc";

// call the main cracking function

do\_crack(filename,dictionary);

printf("CRACKING COMPLETE");

}

Code7: C code for cracking a key (incomplete).

Conclusion

The Coursework help me understand a lot of overall concept of cryptography. It showed me how easily it is for an unaware employee’s account to be compromise. Also it is clearly show the important of the concept in real-life.

I know that the course did not focus on coding ability but it should be better to provided more step by step exercise for from programing in the future course. Especially on Java and C coding with Opensssl library. Shell script is good but I think everyone can learn it without problem. Also C or java programming its will displayed more ability to customise and might lead to an idea for dissertation.