Lab 08: Secret-Key Encryption Lab

Due Wednesday April 19th @11:59PM

Secret-Key Encryption Lab

Adapted from SEED Labs: A Hands-on Lab for Security Education.

The learning objective of this lab is for students to get familiar with key concepts behind secret-key encryption and some common attacks on encryption. From this lab, students will gain first-hand experience in encryption algorithms, encryption modes, padding, and initialization vectors (IV). Moreover, students will learn how to use tools, and write programs, to encrypt and decrypt messages.

There are many common mistakes made by developers when using various encryption algorithms and modes. These mistakes weaken the strength of the encryption, which can make data vulnerable. This lab exposes students to some of these mistakes, and gives students the opportunity to work through the process of exploiting weak encryption.

This lab covers the following topics:

- Secret-key encryption
- Substitution cipher and frequency analysis
- Encryption modes, IV, and padding
- Common mistakes when using encryption algorithms

Resources

- Code related to this lab can be found in `08_ske/` of our class's GitHub repository.
- OpenSSL Cryptography and SSL/TLS Toolkit - Manuals
- Chapter 21 in the SEED Textbook.

Lab Tasks

This lab has been tested on the pre-built SEED VM (Ubuntu 20.04 VM).

Task 1: Breaking a Substitution cipher

Be sure to pull changes for our course repo (git pull). You will find files for this lab inside the `/08-ske` folder. In this task, you are given a cipher-text (ciphertext.txt) that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your task is to find out the original text using frequency analysis. You may assume that the original text is an article written in the English language.

You should use frequency analysis, just like we did in class. There is a `freq.py` in the course repo that will print out the most frequent 1-grams, 2-grams, and 3-grams. You should use the `tr` command to decode the cipher. For example, in the following, we replace letters a, e, and t in in.txt with letters X, G, E, respectively; the results are saved in out.txt.

```
tr 'aet' 'XGE' < ciphertext.txt > out.txt
```

Task 2: Encryption Ciphers and Modes

In this task, you will experiment with various encryption algorithms and modes.

You can use the following `openssl enc` command to encrypt/decrypt a file.

```
$ openssl enc -CIPHERTYPE -e -in plain.txt -out cipher.bin -K KEY -iv IV -p

# Summary of common `openssl enc` options:
# -in <file>     input file
# -out <file>    output file
# -e             encrypt
# -d             decrypt
# -K             the key (in hex format) must follow this option
# -iv            the IV (in hex format) must follow this option
# -[pP]          print the iv/key (then exit if -P)
```

To view the manual pages, type `man openssl` and `man enc`.

You need to replace `-CIPHERTYPE` with a specific cipher type, such as `-aes-128-cbc`, `-aes-128-cfb`, `-aes-128-ofb`, etc.

You also need to replace `KEY` and `IV` with the encryption/decryption key and initialization vector, respectively.

Your task is to use the above `openssl enc` command to encrypt data using at least 3 different ciphers.

Task 3: Comparing Encryption Modes

In the supplied files, `pic_original.bmp` is a simple picture in the BMP - or bitmap - file format.

We would like to encrypt this picture so that anyone without the encryption key is unable to know what the file contains.

Task 3.1

In this task, you must encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Since this encrypted picture is in fact a picture, please start by trying to view the encrypted picture as any other picture. Feel free to use your favorite picture viewing software. Note, however, that for a properly-formatted `.bmp` file, the first 54 bytes must contain the header information about the picture; because the image is in fact encrypted, we have to set the file header correctly so that the encrypted file will be recognized as a legitimate `.bmp` file. To achieve this, we must replace the header of the encrypted picture with that of the original picture.

   To make this change we can you could, for example, use the `bless` hex editor tool (already installed on our VM) to directly modify binary files. (In general you are free to use any hex editor you like.) Alternatively, we can also use the following commands to extract the **header** from `p1.bmp`, the **body** (data) from `p2.bmp` (starting from offset 55 to the end of the file), and then combine the header and body together into a new file (`new.bmp`).

   ```
   $ head -c 54 p1.bmp  > header
   $ tail -c +55 p2.bmp > body
   $ cat header body > new.bmp
   ```

2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called `eog` on the VM). Can you derive any useful information about the original picture from simply viewing the encrypted picture? Please explain your observations.

Task 3.2

Now, select a picture of your choice, repeat the experiment above, and report your process and observations.

Task 4: 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.

Task 4.1

Create three files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We can use the following command to create such files. The following example creates a file `f1.txt` with length 5 (note that, without the `-n` option, the length will be 6, because a newline character will be added by `echo`):

```
$ echo -n "12345" > f1.txt
```

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.

It is interested to examine what is added as padding during encryption. To achieve this goal, we will decrypt these files you created above using `openssl enc -aes-128-cbc -d`. Unfortunately, decryption will automatically remove any padding by default, making it impossible for us to examine the padding. However, the command does have an option called `-nopad`, which disables the step that attempts to remove padding. By looking at the resulting decrypted data, we can then see what data are used in the padding. Please use this technique to figure out what padding values are added to the three files.

It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. The following example shows how to display a file in the hex format:

```
$ hexdump -C p1.txt
00000000  31 32 33 34 35 36 37 38  39 49 4a 4b 4c 0a   |123456789IJKL.|
$ xxd p1.txt
00000000: 3132 3334 3536 3738 3949 4a4b 4c0a            123456789IJKL.
```

Task 4.2

Please repeat the previous task for each of the following modes of operation using the `aes` cipher with 128-bit keys: ECB, CFB OFB.

Please report which modes have padding and which ones do not.

Task 5: Error Propagation & Corrupted Ciphertext

To understand the error propagation property of various encryption modes, in this task you will create a ciphertext using a specific encryption mode, intentionally corrupt a bit in the ciphertext, decrypt the corrupted ciphertext, and then examine the result.

Task 5.1: Predictions

Before you actually conduct this task, 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?

Please note your answers (and provide any relevant justification for each) before proceeding with actually carrying out the steps below.

Task 5.2: ECB & Data Corruption

After you have answered the question in Task 5.1, please carry out the following steps:

1. Create a text file that is at least 1000 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Intentionally corrupt the file: change a single bit in some byte in the encrypted file. (You can achieve this corruption using any hex editor. `ghex ciphertext.txt` will open it up in a hex editor
4. Decrypt the corrupted ciphertext file using the correct key and IV.

How much information can you recover by decrypting the corrupted file if the mode of operation used in encryption is ECB? Please provide justification.

Task 5.3: CBC & Data Corruption

Please repeat Task 5.2, but this time use the CBC mode when encrypting/decrypting data.

Task 5.4: CFB & Data Corruption

Please repeat Task 5.2, but this time use the CFB mode when encrypting/decrypting data.

Task 6: Common Mistakes with IVs

Most of the encryption modes require an Initialization Vector (IV). Properties of an IV depend on the cryptographic scheme used. If we are not careful in selecting IVs, *the encrypted data may not be secure, even though we are using a secure encryption algorithm and mode!* The objective of this task is to help students understand some of the problems that arise if an IV is not chosen properly.

Task 6.1: Uniqueness of the IV

A basic requirement for the 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 observations and explain why the IV needs to be unique.

Task 6.2: Known Plaintext Attack

One may argue that if the *plaintext* does not repeat, using the same IV (and key) is safe. In this task we will investigate this matter.

> *The attack used in this experiment is known as the **known-plaintext attack**, which is an attack model for cryptanalysis where the attacker has access to both the plaintext and its encrypted version (ciphertext). If this situation can lead to the revealing of further secret information, the encryption scheme is not considered to be secure.*

In this task we will specifically look at the Output Feedback (OFB) mode.

Assume that the attacker gets hold of a plaintext ($P1$) and a ciphertext ($C1$).

Question: *Can they decrypt other (different) encrypted messages if the same IV is always used?*

To examine whether this is possible, assume 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
```

Hint: You may find it useful to have a tool that can help you XOR some values in this task. You are free to use whatever XOR tool you want. We provide a simple python script (see below). Others have used an online XOR calculator. Feel free to do whatever makes the most sense for you.

```
#!/usr/bin/python3

#
# XOR strings (ascii strings and hex strings).
#

MSG   = "A message"
HEX_1 = "aabbccddeeff1122334455"
HEX_2 = "1122334455778800aabbdd"

# XOR two bytearrays
def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))

D1 = bytes(MSG, 'utf-8')      # Convert ascii string to bytearray
D2 = bytearray.fromhex(HEX_1) # Convert hex string to bytearray
D3 = bytearray.fromhex(HEX_2) # Convert hex string to bytearray

r1 = xor(D1, D2)
r2 = xor(D2, D3)
r3 = xor(D2, D2)
print(r1.hex())
print(r2.hex())
print(r3.hex())
```

sample_xor.py hosted with ❤ by GitHub   view raw

This page was last updated: 05/12/2023