University of Miami

Crypto Lab 1

ECE 576

Nigel John
10-2-2019

Objectives:

1. To understand and use symmetric key cryptography
2. To securely exchange a file over an unsecure channel

Task 0: Install python (v3), python IDE, python cryptography

For this lab, you are going to need to have python installed on your laptop. If you already have this then you can skip this step.

1. Download the latest python (v3) from www.python.org for your operating system
    a. Your SEEDs systems have python 2 but not python 3
    b. Your Kali has both, but you may need to update
2. Follow the installation instructions for your system
    a. See: https://wiki.python.org/moin/BeginnersGuide
3. For programming information:
    a. https://wiki.python.org/moin/HowToEditPythonCode
4. Install the python cryptography module
    a. Open a Terminal window (or Command shell)
    b. Type: pip install cryptography
        i. check https://cryptography.io for installation issues
5. Download and install an IDE:
    a. IDLE: python ships with a standard IDE, IDLE, written in pure python
    b. PyCharm: www.jetbrains.com/pycharm
    c. Eclipse with PyDev: www.eclipse.org (my personal preference)
    d. Xcode on the Mac also supports python
6. Familiarize yourself with the platform
7. The python documentation is at:
    a. Main documents: https://docs.python.org/3/
    b. Tutorial: https://docs.python.org/3/tutorial/index.html

Task 1: Read/Write Files in Python

For this task you are going to get familiar with the File I/O in Java.

Task 1.1: Using buffered Byte Streams

The following code shows reading data from one file and writing data to another file. Familiarize yourself with the code (lookup the functions in the python documentation to understand what they do).

```
import os
import io

"""
A simple program to copy one file to another in fixed size blocks
variables:
fname: source file name
fname2: destination file name
blocksize: size of blocks in bytes
"""

# filenames
fname = 'infile.txt'
fname2 = 'outfile.txt'

# get the full path names
path = os.path.abspath(fname)
path2 = os.path.abspath(fname2)

# print message to user
print('copying ', path, 'to ', path2)
```

Setup the imports, filenames and print a message

```
# set the blocksize
blocksize = 16

# set the totalsize counter
totalsize = 0

# create a mutable array to hold the bytes
data = bytearray(blocksize)

# open the files, in buffered binary mode
file = open(fname, 'rb')
file2 = open(fname2, 'wb')
```

Set the blocksize for reading (16 bytes = 128bits) and open the files

Try experimenting with different block sizes.

```
# loop until done
while True:
    # read block from source file
    num = file.readinto(data)

    # adjust totalsize
    totalsize += num

    # print data, assuming text data
    print(num,data)
    # use following if raw binary data
    # print(num,data.hex())

    # check if full block read
    if num == blocksize:
        # write full block to destination
        file2.write(data)
    else:
        # extract subarray
        data2 = data[0:num]

        # write subarray to destination and break loop
        file2.write(data2)
        break
```

Keep reading blocks until none left

```
# close files (note will also flush destination file
file.close()
file2.close()

# print totalsize
print('read ',totalsize,' bytes')
```

Close the files and end

Now:

1. **Create a python file for this code in your IDE**
2. **Copy the code to your file**
3. **Create a file called infile.txt and enter some text into it.**
4. **Run the program and ensure that the data is copied to the file outfile.txt**

This program uses a buffered raw Stream in python. It is also possible to use a text based stream that reads into a python string.

Task 1.2: Using Character Streams

If the underlying data is character based, then you can connect the stream to a character based stream and read and write as strings instead.

Create a new version of the program that reads text data and copies it to another file.

Task 2: Create and use a symmetric key block cipher in Python using the cryptography package

Task 2.1: Use a KDF to generate the keys we need

In this task, you will create a random number generator and a seed values (password entered by the user) to create a pseudo-random sequence generator that will then be used to create a secret key for a cipher to be used for encryption and decryption.

For this we are using the cryptography module for python. This package provides both a high-level cryptographic interface and a low-level interface. We are going to use the low-level interface. This is within the hazmat module of cryptography as need to understand ciphers, keys, operational modes, etc. to use it properly.

We are going to be using AES as our cipher of choice (you can experiment with others if you like) in 128-bit block and 128-bit key mode. Since the key is 128-bit (16 bytes), we could simply create a readable key from 16 ASCII characters. This, though, may result in a limitation of the key space, so instead we use a Key Derivation Function (KDF) to generate the key from source material (a text password) provided by the user. PBKDF2 (Password Based Key Derivation Function 2) is typically used for deriving a cryptographic key from a password using an underlying hashing algorithm applied many times with a password as the seed value

The steps for this task are:

1. **Create a random number generator**
2. **Create a 'salt' value for this to provide to a key derivation function**
3. **Create a KDF for the password**
4. **Since we will use CBC mode for this cipher, we also need an Initialization Vector (IV)**
    a. **Use another KDF to generate this also**

```
import os
import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from encodings.base64_codec import base64_encode

backend = default_backend()
salt = os.urandom(16)

print(salt.hex())
```

Setup the imports needed, the backend to be used, create a random salt and print it

```
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=16,
    salt=salt,
    iterations=100000,
    backend=backend)

idf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=16,
    salt=salt,
    iterations=100000,
    backend=backend)

passwd = b'password'
ivval = b'hello'

key = kdf.derive(passwd)
iv = idf.derive(ivval)

print(key.hex())
print(iv.hex())
```

Create a KDF for the key and another for the IV, set the text seeds, run the KDFs to generate the values needed and print them

Try this program and report the results for the key and IV. Explain.

Task 2.2: The cipher

Now that we have some material for the key, we can create the cipher and use it to do some encryption

The steps here are:

1. **Create a chipher object and initialize it with**
    a. **The algorithm (AES) and the key we derived earlier**
    b. **The mode (CBC) with the IV we derived earlier**
    c. **The default backend**
2. **Encrypt some dummy data**
    a. **Get an encryptor**
    b. **Use update and finalize to carry out the encryption**
    c. **Print the results**
3. **Decrypt the data to see that we can retrieve the original result**
    a. **Get a decryptor**
    b. **Use update and finalize to carry out the decryption**
    c. **Print the results**

```
cipher = Cipher(
    algorithm=algorithms.AES(key),
    mode=modes.CBC(iv),
    backend=backend)

encryptor = cipher.encryptor()
```

Create the cipher, and get an encryptor

```
mydata = b'1234567812345678'

print(mydata)

ciphertext = encryptor.update(mydata) + encryptor.finalize()

print(ciphertext.hex())
```

Create some dummy data and encrypt it

Now for the decryption

```
decryptor = cipher.decryptor()

plaintext = decryptor.update(ciphertext) + decryptor.finalize()

print(plaintext.hex())
```

Get the decryptor, decrypt the ciphertext and print the result for comparison

Try this and make sure you understand the steps. Try with different data of different lengths and note the results.

You can also use the base64 encoder to output a more readable form of the data if you prefer.

```
print(base64_encode(key))
print(base64_encode(iv))
print(base64_encode(ciphertext))
```

Task 2.3: Effect of Padding

Now we can try to add some padding of the data so that we can properly handle data of any size. In this case the padding is handled separately from the cipher.

For this padding, we use PKCS7 padding which is a generalization of PKCS5 padding (also known as standard padding). PKCS7 padding works by appending N bytes with the value of chr(N), where N is the number of bytes required to make the final block of data the same size as the block size.

This is done by creating a padder and padding the data before applying it to the cipher.

```
padder = padding.PKCS7(128).padder()

mydata = b'1234567812345678'
mydata_pad = padder.update(mydata) + padder.finalize()

print(mydata_pad.hex())

ciphertext = encryptor.update(mydata_pad) + encryptor.finalize()
```

Create the padder, pad the data, print the padded data and encrypt it.

Try the padding with data that is not a multiple of the block size and data that is a multiple of the block size and comment.

Task 2.4: Using AES in other modes

So far, you have done the encryption/decryption in CBC mode with an IV. You may also run the cipher in ECB mode. **Try the cipher in ECB mode with the same block repeated at least twice and note the results. Compare this with CBC modes.**

The prior modes maintain the block nature of the cipher, the other modes, Counter (CTR), Cipher-Feedback (CFB), and Output-Feedback(OFB) allow you to turn the block cipher into a stream cipher.

Note that for modes, they require an IV or a Nonce, these do not necessarily have to remain secret and could be saved with the ciphertext as long as the same key and IV are not used for additional messages.

Task 3: Encrypting a File with AES

Now combine your file reading/writing code with your encryption code to allow the user to (there can be separate python programs):

1. **Either Encrypt or Decrypt a file**
2. **For Encrypt:**
    a. **Select a file to encrypt**
    b. **Provide a name for the output file**
    c. **Provide a password to be used as the seed in a KDF for generation of an encryption key**
    d. **Provide a password to be used to create the IV**
    e. **Read the data from the file**
    f. **Encrypt the data, remember to pad the data**
        i. **Note if the file is large, you may have to read and encrypt the data in chunks using multiple reads in a loop.**
        ii. **You would use the update method of cipher to continue the encryption**
        iii. **You must have a finalize() at the end to ensure that the cipher is completed properly.**
    g. **Write the encrypted data to a new file**
3. **For Decrypt:**
    a. **Select a file to decrypt**
    b. **Provide a name for the output file**
    c. **Choose an encryption algorithm**
    d. **Provide a password to be used as the seed in a KDF for generation of an encryption key**
    e. **Provide a password to be used to create the IV**
    f. **Read the data from the file**
    g. **Decrypt the data**
    h. **Write the decrypted data to a new file (remember to remove the padding)**
4. **Compare**

Note that one of the problems you have to overcome here is that the actual key (derived by the KDF using the password) must the exactly the same for both encryption and decryption.

The same applies for the IV or nonce, but those can be transmitted in the open as they do not absolutely have to remain secret.

Task 4: Transmit over insecure channel

Now to show the value of the encryption, you will capture the transfer of an unencrypted file over an insecure channel and an encrypted file over the same channel while capturing the network traffic.

1. **Start up two SEEDS VMs, A and B in the same NatNetwork**
2. **Startup a Kali Linux VM on the same NatNetwork**
3. **Copy a file that you have previously encrypted to machine A in both unencrypted and encrypted form.**
    a. **You may also transfer your code to the VM and run it there to create the encrypted data**
    b. **If you do this, make sure of the modules are available**
4. **Make sure that ftp is running on machine B**
    a. **You can try to ftp to machine B from machine A**
5. **Startup Wireshark on you Kali VM**
    a. **Start a capture on Wireshark**
6. **On machine A, ftp to machine B, 'ftp <machine_ip>'**
    a. **Transfer the unencrypted file to machine B**
    b. **For ftp, you use the put command**
        i. **'put <filename>'**
    c. **Exit from ftp**
7. **On your Kali**
    a. **Find the FTP packets**
    b. **Follow the TCP stream to see the data transmitted**
    c. **Restart the capture if you had stopped it**
8. **On machine A, ftp to machine B, 'ftp <machine_ip>'**
    a. **Transfer the encrypted file to machine B**
    b. **For ftp, you use the put command**
        i. **'put <filename>'**
    c. **Exit from ftp**
9. **On your Kali repeat step 7 and compare the data now.**

Report

1. Statement of what was done for each task
2. All source code for each task
3. Screen shots of all operations for each task

Reference:

Python reference: https://docs.python.org/3/
Cryptography module reference: https://cryptography.io/en/latest/