# Cryptography Homework 3

## Required Reading

Cryptology 3 slides

## Optional Reading

Wikipedia has good articles on [stream](https://en.wikipedia.org/wiki/Stream_cipher) and [block](https://en.wikipedia.org/wiki/Block_cipher) ciphers, [block cipher modes of operation](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation), [DES](https://en.wikipedia.org/wiki/Data_Encryption_Standard) and [AES](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard).

## Friendly Advice

This is a complicated assignment. The assignment at the end of this document will be much easier if you duplicate the steps on your own computer as you read through the homework.

## Install PyCryptodome

Please use the document “PyCryptodome Installation.docx” to install PyCryptodome on your computer or VM if you have not already done so.

## OS Hints

There are several environments you can use to do this lab. In Windows:

* Command prompt (terminal) with Notepad++ [(download Notepad++)](https://notepad-plus-plus.org/downloads/)
* Command prompt plus Notepad
* IDLE (installed when you installed Python)
* Visual Studio Code ([download Code](https://code.visualstudio.com/Download))

For Linux:

* Terminal with text editor (gedit). Change preferences -> Editor to select “Insert spaces instead of tabs,” and Tab Width 4 spaces.
* Visual Studio Code ([download Code](https://code.visualstudio.com/Download))
* IDLE (sudo apt install idle3 for Ubuntu)

## Python and Strong Cryptography

There are several Python cryptography modules, but some of them are not suitable for use in production systems (see <https://theartofmachinery.com/2017/02/02/dont_use_pycrypto.html>.) A common module you will see in books and articles is PyCrypto. However, PyCrypto has exploitable bugs, and has not been supported since 2014. Do not use PyCrypto. A new project, PyCryptodome, is a fork of PyCrypto and is supported (<https://github.com/Legrandin/pycryptodome>.) However, even though PyCryptodome has all the encryption pieces we need, it still takes experience in cryptography to assemble those pieces in a secure manner. It is easy to create code based on standard functions like AES and accidentally implement flaws that severely weaken the cryptography.

Our goal in this lab is to look at the functions (primitives) that comprise current symmetric encryption, specifically AES. The Electronic Codebook mode of AES (AES-ECB) is not secure; we will use it to get started, but it should never be used in production.

# Symmetric Encryption with AES

## Overview

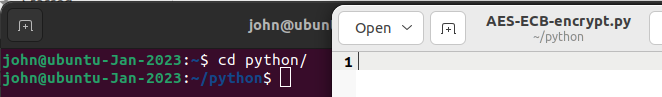
We will perform an AES encryption like that in slide 7 of the class notes. Instead of printing the results to the screen, we will save them to a file for practice.

We will create a key and plaintext, and then encrypt the plaintext with the simplest AES mode, Electronic Codebook (ECB). This is not secure, but it is an easy mode to use for our first attempt. The AES module will create ciphertext, which we will save to a file. Then we will open the file and decrypt it.

Note: This example will jump back and forth between Linux and Windows to demonstrate that it works in both places. We will use the terminal and a text editor, but you can use an IDE if you like.

## Encryption

Create a directory in a convenient location and use your text editor to create a file for our Python AES encryption. I am calling my file AES-ECB-encrypt.py. The terminal should be in the same directory as the file.  
Graphical user interface, application

Description automatically generated  
or  


Import the AES module from Crypto.Cipher. Note that Python is case sensitive, even on Windows. Run your script just to make sure Pycryptodome is installed. No errors (no output at all) is good.  
Graphical user interface, text, application, chat or text message

Description automatically generated

from Crypto.Cipher import AES

Create an AES object and give it the key you will use to encrypt. To start with, we will use ECB mode for simplicity and get more realistic later. Remember that the key must be exactly 128, 192, or 256 bits (16, 24, or 32 bytes) long. One byte is one ASCII character. If the length is not a multiple of 16 bytes, you will receive an error. Test your key to make sure it is 16 bytes (128 bits) long. This is an example of what happens when the key is too short.

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Note: In Python, the most relevant error message is usually the one at the bottom.

You can check the length of your string at the interactive Python prompt ( >>> ) using len().  
len(b'mykey123')  
Text

Description automatically generated

Text

Description automatically generated

With standard encoding, one character is one byte, so that should help you to find a key that is 16 bytes long. Once you have a key that is 16 bytes long, use it to create an AES object. Use your own key!

aes\_obj = AES.new(b'This is the key!', AES.MODE\_ECB)

**Graphical user interface, text, application, Word, email

Description automatically generated**

Ah, no errors. Good.

**Important.** Note that there is a “b” at the beginning of the key.  
b'This is the key!'  
The “b” tells Python3 that this is a byte literal (like a byte array) and not a UTF-8 string object. That is necessary because the AES module wants bytes as input instead of strings.

Now we will put our plaintext message into the variable plaintext. The variable ciphertext will hold the encrypted version of the message. AES will only accept plaintext blocks of 128 bits or 16 bytes. You will have to add your own padding to make the length of your message a multiple of 16 bytes. Make your own plaintext!

Text

Description automatically generated

Once your plaintext is a multiple of 16 bytes long add it to your text editor and run it to check for errors.  
Graphical user interface, text, website

Description automatically generated

No errors, good.

Encrypt the plaintext with this line and print the result  
ciphertext = aes\_obj.encrypt(plaintext)  
print(ciphertext)

Graphical user interface, text, application

Description automatically generated

When you print a variable, Python will automatically convert it to a viewable string if it can. Since plaintext started as a string, it is readable. The ciphertext is binary data, so Python shows ASCII where it can and renders the rest in hex notation (i.e. \xe6). Not pretty, but it works.

Now that we have encrypted the message, let’s save it to a file. There are different formats for reading and writing to files. I prefer the with-open syntax. It automatically closes the file when the operation is complete, so I cannot forget to close the file.

Once the file is written, I can attempt to read it with the cat command. Note that the file is binary data. The print statement in the Python code shows the data as raw bytes. The cat command shows what happens when the OS tries to render binary data as ASCII. The xxd command shows binary data in hex format.

Graphical user interface, text

Description automatically generated

Here is the entire encryption script.

from Crypto.Cipher import AES

aes\_obj = AES.new(b'This is the key!', AES.MODE\_ECB)

plaintext = b'Attack at dawn, regardless of the weather. No excuses! 12345678'

ciphertext = aes\_obj.encrypt(plaintext)

print(ciphertext)

with open('aes-ecb.bin', 'wb') as myfile:

myfile.write(ciphertext)

## Decryption

The process is just about the reverse of what we did for encryption. I will put the decryption script in a new file called AES-ECB-decrypt.py

from Crypto.Cipher import AES  
Graphical user interface, application

Description automatically generated

Next, we need to read the binary file that holds our ciphertext. I’ll print the result just to be sure I read the correct file.

Graphical user interface, text, application, chat or text message

Description automatically generated

from Crypto.Cipher import AES

with open('aes-ecb.bin', 'rb') as myfile:

ciphertext = myfile.read()

print(ciphertext)

Hopefully you encrypted your own message, so your ciphertext should be different from what is shown here.

Create an AES object using the same key that was used to encrypt the plaintext. Then call the decrypt method of the AES object and print the plaintext result.

Graphical user interface, text, chat or text message

Description automatically generated

Graphical user interface, text, application

Description automatically generated

from Crypto.Cipher import AES

with open('aes-ecb.bin', 'rb') as myfile:

ciphertext = myfile.read()

aes\_obj = AES.new(b'This is the key!', AES.MODE\_ECB)

plaintext = aes\_obj.decrypt(ciphertext)

print(plaintext)

Success!

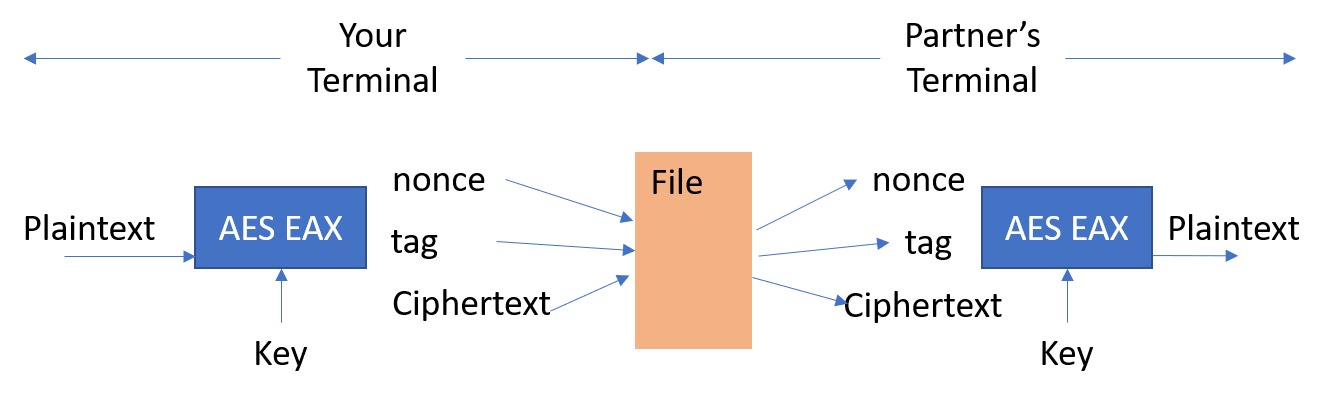
## More Advanced

A better block cipher mode can help us avoid two problems that are present in the simple ECB mode we have been using. First, the new mode should introduce a random element, a nonce or Initialization Vector (IV), so that each time we encrypt an identical block of plaintext, we get a different ciphertext. (Remember the Linux penguin picture from the class notes that was encrypted with ECB mode AES, where we could see the outline of the penguin in the ciphertext.) Secondly, it can provide an authentication code so that we are alerted when someone tries to tamper with our cipher text. This example from PyCryptdome uses EAX mode, which provides both protections. <https://www.pycryptodome.org/en/latest/src/examples.html#encrypt-data-with-aes>.

A mode with a nonce (same as IV) and an authentication code (Pycryptodome calls the code a tag) does make things more complicated, though. Now we have three values, cipher.nonce, tag, and ciphertext that need to be passed to the message recipient. For this example, we will just stuff them together (concatenate) into one file. We will trust that the recipient (or their script) knows that the first 16 bytes are the nonce, the next 16 bytes are the tag (message authentication code) and everything else is ciphertext.

The decryption side extracts the three pieces from the file and then decrypts the ciphertext. The nonce gives the encryption a random start and the tag allows us to detect corruption or malicious changes to the ciphertext.

A nice thing about Pycryptodome’s code for EAX mode is that it adds padding to the ciphertext so that the length is always a multiple of 16 bytes. The code for decryption recognizes the padding and removes it automatically.



## Encryption

We will use a modified version of this example from <https://www.pycryptodome.org/en/latest/src/examples.html#encrypt-data-with-aes>

Text

Description automatically generated

The AES object created in the example is cipher. Instead of using the cipher.encrypt() method we used before, the example uses cipher.encrypt\_and\_digest(). It creates the ciphertext as before, but it also creates an authentication code, which the example calls a tag. When the ciphertext is decrypted the ciphertext and the tag must match. The tag can only be computed from the ciphertext with knowledge of the key. An attacker cannot alter the ciphertext and create a correct tag because they do not know the key.

When the AES object (cipher) is created, a random number, or nonce is also created. The nonce gives the EAX mode a random starting point so that repeated encryption of the same plaintext always gives a different ciphertext. It is a property of cipher, so it is available as cipher.nonce. The result is that we have additional data that needs to travel with the ciphertext. The components are:

1. Ciphertext. This is the encrypted message created by the cipher.encrypt\_and\_digest() method.
2. Tag (Message authentication code). The tag is computed separately from the ciphertext during encryption. It will allow us to verify that the ciphertext has not been tampered with. The variable tag is the second output of the cipher.encrypt\_and\_digest() method.
3. Nonce. The nonce is the random number that gives our encryption a random starting place. It is created by the AES.new() method, and is available as cipher.nonce.

The nonce, tag, and ciphertext all travel together as a package. It does not matter if an attacker sees them as they transit the network, since they are of no value unless the attacker has key.

The key must be available to the receiver, and it must be sent to the receiver securely in a different channel than the message. If we made a mistake and included the key in our message, our encryption is worthless.

We will use a simpler method to write the nonce, tag, and ciphertext to a file.

## Decryption

To decrypt the message, we create an AES object using the key, EAX mode and the nonce. (The key was securely exchanged by some other method.) Then we call the cipher.decrypt\_and\_verify() method with the ciphertext and the tag. If the message is corrupted or tampered with, the method will throw an error. Otherwise, it will give us the plaintext of the message.  
Graphical user interface, text

Description automatically generated

The lines in the with open section may look strange, as they have read(16) and read(-1).

The code just reads the first 16 bytes of the file and puts them into the variable nonce. Then it reads the second 16 bytes and puts them into tag. Everything that is left goes into ciphertext.

Note: EAX uses the Counter Mode (CTR) to randomize the ciphertext, and to avoid the “penguin problem.” It creates the tag with the One-key Message Authentication Code (OMAC) method. The technical details of the EAX mode are here: <http://web.cs.ucdavis.edu/~rogaway/papers/eax.pdf>

# An exercise for you

Split up into pairs and use AES in EAX mode to securely transmit messages of your choosing to each other. That means two messages. Each student should encrypt and send one message and receive and decrypt one message.

Note: This lab assumes students are working in small groups. If you are working alone, just send the message to yourself.

## Key generation

Later, we will use a random number generator to create the key. Right now, make life easier for your partner and yourself by using a key that is easy to type. Make sure your key is 16 bytes long.

## Key exchange

Write your key on a scrap of paper and give it to your partner. If you can do that without the other pairs of students spying on you, we will call that “secure key exchange.”

## Create and encrypt a message

Create variables to contain your key and message. Note that in the example, the message plaintext is stored in the variable data. Also, MODE\_EAX in the Python module does another nice thing for us; it pads the plaintext so that it fits in 128 bit/16 byte blocks.  
key = b'whatever your key is'  
data = b'whatever you want the message to be, don’t worry about length. '

Use these lines from the example.  
cipher = AES.new(key, AES.MODE\_EAX)  
ciphertext, tag = cipher.encrypt\_and\_digest(data)

Write the three parts to a file.

with open("encrypted.bin", "wb") as file\_out:

for x in (cipher.nonce, tag, ciphertext):

file\_out.write(x)

This method will automatically close the file when it is finished. Note that the indentations are important; that is the way that Python identifies script blocks. Now your data, nonce, tag, and ciphertext, are stored in a binary file called encrypted.bin. The nonce and tag are 16 bytes each, and the rest is the ciphertext.

This is the script running on Windows.  
Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Here it is as a script in the Ubuntu text editor (gedit).  
Text

Description automatically generated

Text

Description automatically generated

## Transmit the message

## Transmit the message to your partner by emailing encrypted.bin as an attachment or using sneakernet (Copy encrypted.bin to a flash drive and hand it to your partner. In the old days we used floppy disks and called this sneakernet), or whatever method seems appropriate.

**Note**: If you are working in Linux but want to send the message by email in Windows, you will need to paste the encrypted.bin file to the Windows desktop first. Then attach the file to your email. If you try to paste the message directly into your email, it may not work as expected.

## Decrypt the message

Decrypt your partner’s message using their key and the code from the example below. Be sure to use the key that your partner uses to encrypt the message.  
key = b'This is my partner’s key'

Once you have your partner’s key saved in the variable key, you can read your partner’s file. This reads the first 16 bytes into nonce, the next 16 bytes into tag, and everything else into ciphertext.

with open('encrypted.bin', 'rb') as file\_in:

nonce = file\_in.read(16)

tag = file\_in.read(16)

ciphertext = file\_in.read(-1)

Next, create the AES object with the key and the nonce. Then decrypt the ciphertext. The method, cipher.decrypt\_and\_verify, will also use the tag to verify that the ciphertext has not been corrupted.

cipher = AES.new(key, AES.MODE\_EAX, nonce)

data = cipher.decrypt\_and\_verify(ciphertext, tag)

This is what the script looks like in Windows Notepad++.  
  
Graphical user interface, text, application

Description automatically generated

This is what the script looks like in Ubuntu.  
  
Text

Description automatically generated

# Hand in

1. Hand in screenshots of your Python terminal as you encrypted your message and decrypted your partner’s message.
2. I am trying to find a way to support students in this lab if they use either Windows/Notepad++ or Linux/gedit as their development environment. I have tried having separate documents for the two—this lab alternates between showing screenshots of Windows and Linux. Is it easy to understand this way, or should it be two separate documents? How did this method work for you?