# Cryptography Homework 3b

## 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 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.

A mode with a nonce (same as IV) and an authentication code (Pycryptodome calls the authentication 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 rectangular object with arrows pointing to the side

Description automatically generated

## 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.

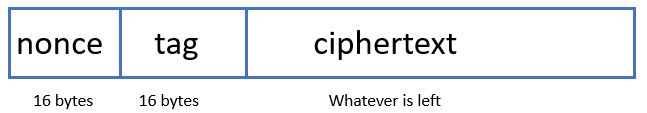
When the AES object (cipher) is created, a random number, or nonce is also created. The nonce gives the EAX mode a random starting point so that repeated encryption of the same plaintext always gives a different ciphertext. It is a property of cipher, so it is available as cipher.nonce. The result is 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.

Note that the last three lines just stuff the nonce, tag, and ciphertext together in the file without any markers to tell where one stops and another begins. This only works because we know that the nonce and tag are exactly 16 bytes long. When we decode the file, we know that the first 16 bytes are the nonce, the next 16 bytes are the tag, and everything else is the ciphertext.



## 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

Hand in screenshots of your Python terminal as you encrypted your message and decrypted your partner’s message.