SY308: Security Fundamental Principles

Calendar	Policy

Lab: Implementing the CBC mode with AES

The goal of this lab is to implement AES encryption with CBC mode in Python. Fortunately, the Crypto library does a lot of the work for us. There are four main tasks you must do, which the library will not do for you:

1. Add/remove padding so that the message is a multiple of 16 bytes.
2. Create a random IV during encryption and store it along with the ciphertext so that it can be used again during decryption.
3. Put all this together and call the aes.encrypt and aes.decrypt functions correctly.
4. Reading and writing to files to store the keys, messages and ciphertexts.

Download the **starter code** here. In the file CBC.py I have broken out the tasks that you must do into separate functions. Each function has a description of what the inputs are, what the outputs should be, and an overview of the task that function must accomplish.

Padding

- You need to completely understand how PKCS#5 padding scheme works, and add the appropriate padding to the message.
- You can use the bytearray.append() function to prepare a padding. For example:

```
>>> b = 15
>>> p = bytearray()
>>> p.append(b)
>>> p.append(b)
>>> print(p)
bytearray(b'\x0f\x0f')
```

Creating and storing IVs

Each encrypted message must have a fresh random IV. Fortunately, we have already done this in last week's lab and it will be the same here. You can use urandom to generate the IV, and then when the ciphertext is created the iv should be added to the front as the first block. When decrypting, the first block of the ciphertext is peeled off to be used as the IV.

Specifying mode and IV

AES.new() function takes IV along with the key and the mode (i.e., AES.MODE_CBC in this case). The third argument, after the mode, should be the IV. For example:

```
aes = AES.new(key, AES.MODE_CBC, iv)
```

See the reference manual for additional details, but you should have everything you need to know from the class notes.

Testing

I have written a testing script for you, TestCBC.py. It will run each of your functions and check that they are producing the correct output. If all of your functions work successfully you should see the message "Good!" at the bottom, otherwise there will be an error pointing you to the function that isn't working correctly.

Putting it all together

When your code works successfully, your final task is to create three programs cbc_encrypt.py, cbc_decrypt.py and cbc_keygen.py. These will each have a main function that lets you run them from the command line to perform key generation, encryption or decryption on a file specified by the user. Essentially, you will be creating a command line utility that will let you use the CBC library you created to easily encrypt and decrypt files. This is a pretty neat tool to have!

1. cbc_keygen.py should take one argument, the name of the file to write the key to, and should write a 16-byte random key to that file.
2. cbc_encrypt.py should take three arguments in the following order: the name of the file containing the key, the name of the file containing the message to be encrypted and the name of the file where the ciphertext should be written.
3. cbc_decrypt.py should also take three arguments in the following order: the name of the file containing the key, the name of the file containing the ciphertext to be decrypted and the name of the file where the decrypted message should be stored.

You do not need to do any new cryptography in these programs, they are just wrappers for your existing CBC.py file. You should read/write the appropriate files from the command line arguments and just call functions from CBC.py. You should be able to run the following commands with any file "msg.txt":

```
python3 cbc_keygen.py key.bin
python3 cbc_encrypt.py key.bin msg.txt cipher.bin
python3 cbc_encrypt.py key.bin cipher.bin msg2.txt
diff msg.txt msg2.txt
```

In the end, msg2.txt should be identical to the original message, so the diff should return nothing.