Lab 2: Block Cipher and Padding Oracle Attacks

1. Objectives (5% + 1% extra credit)

- Learn to use cryptographic APIs to perform block encryption.
- Practice padding oracle attacks.

2. (1%) Learn to use cryptographic APIs for block cipher encryption

1. On your Ubuntu VM, make sure Python 2 is installed.

```
$ python --version
```

You should see Python 2.6.x or Python 2.7.x. If not, install python.

2. Install pip

```
$ sudo apt-get install python-setuptools python-dev build-essential.
$ sudo easy_install pip
```

3. Install pycrypto

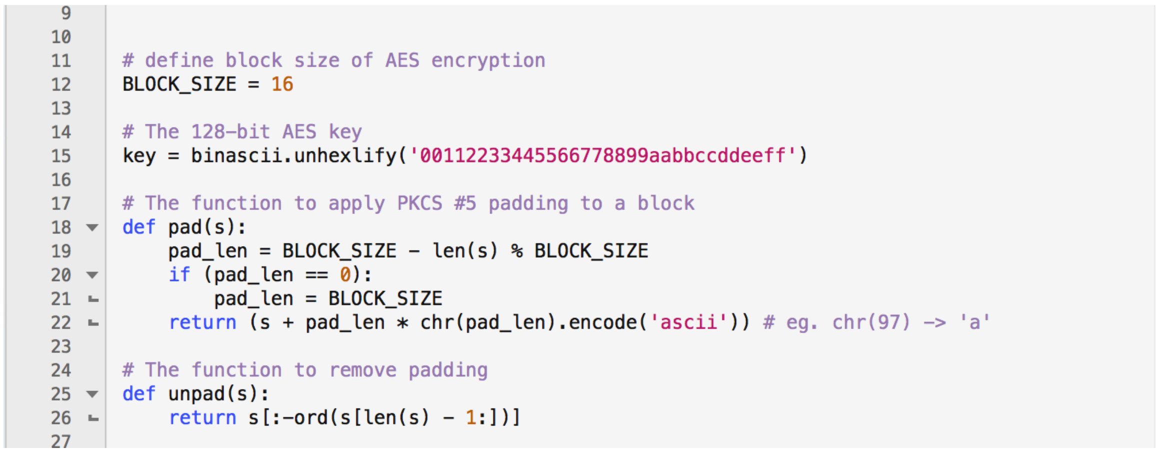
```
$ sudo pip install pycrypto
```

4. Create ecb.py

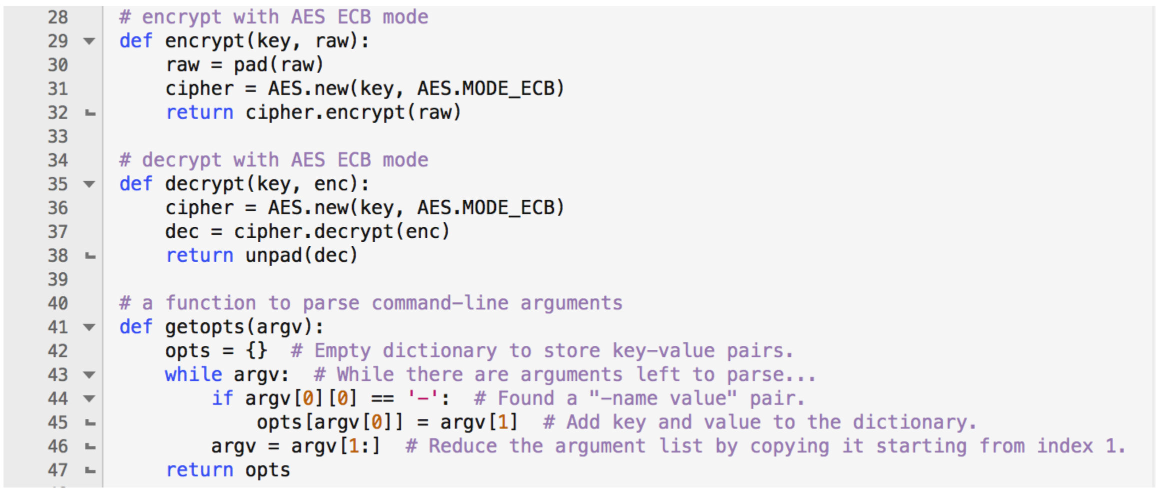
The following code snippet is a Python program to encrypt and decrypt messages using AES128 in the ECB mode. Follow the steps below: (tips: pay attention to indentation in python programming! Always using spaces to replace tabs is a good habit.)

```python
# Import AES symmetric encryption cipher
from Crypto.Cipher import AES

# Import class for hexadecimal string processing
import binascii

# support command-line arguments
import sys
```

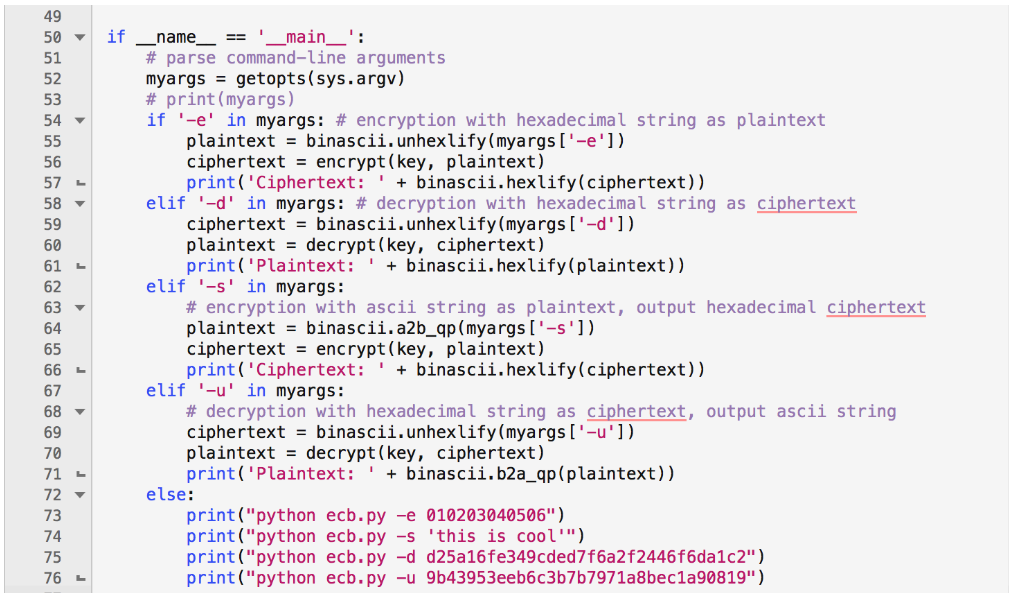
The program first imports AES functions from Crypto.Cipher.


```python


# define block size of AES encryption
BLOCK_SIZE = 16

# The 128-bit AES key
key = binascii.unhexlify('00112233445566778899aabbccddeeff')

# The function to apply PKCS #5 padding to a block
def pad(s):
    pad_len = BLOCK_SIZE - len(s) % BLOCK_SIZE
    if (pad_len == 0):
        pad_len = BLOCK_SIZE
    return (s + pad_len * chr(pad_len).encode('ascii')) # eg. chr(97) -> 'a'

# The function to remove padding
def unpad(s):
    return s[:-ord(s[len(s) - 1:])]

```

Functions to add PKCS#5 paddings and remove paddings are defined. They AES key is a hexadecimal string 0x00112233445566778899aabbccddeeff.


```python
# encrypt with AES ECB mode
def encrypt(key, raw):
    raw = pad(raw)
    cipher = AES.new(key, AES.MODE_ECB)
    return cipher.encrypt(raw)

# decrypt with AES ECB mode
def decrypt(key, enc):
    cipher = AES.new(key, AES.MODE_ECB)
    dec = cipher.decrypt(enc)
    return unpad(dec)

# a function to parse command-line arguments
def getopts(argv):
    opts = {}  # Empty dictionary to store key-value pairs.
    while argv:  # While there are arguments left to parse...
        if argv[0][0] == '-':  # Found a "-name value" pair.
            opts[argv[0]] = argv[1]  # Add key and value to the dictionary.
        argv = argv[1:]  # Reduce the argument list by copying it starting from index 1.
    return opts
```

Define the functions to perform encryption and decryption. Getopts() helps you parse the command-line arguments into a dictionary. Note this code does not check the correctness of padding during decryption.



```python

if __name__ == '__main__':
    # parse command-line arguments
    myargs = getopts(sys.argv)
    # print(myargs)
    if '-e' in myargs: # encryption with hexadecimal string as plaintext
        plaintext = binascii.unhexlify(myargs['-e'])
        ciphertext = encrypt(key, plaintext)
        print('Ciphertext: ' + binascii.hexlify(ciphertext))
    elif '-d' in myargs: # decryption with hexadecimal string as ciphertext
        ciphertext = binascii.unhexlify(myargs['-d'])
        plaintext = decrypt(key, ciphertext)
        print('Plaintext: ' + binascii.hexlify(plaintext))
    elif '-s' in myargs:
        # encryption with ascii string as plaintext, output hexadecimal ciphertext
        plaintext = binascii.a2b_qp(myargs['-s'])
        ciphertext = encrypt(key, plaintext)
        print('Ciphertext: ' + binascii.hexlify(ciphertext))
    elif '-u' in myargs:
        # decryption with hexadecimal string as ciphertext, output ascii string
        ciphertext = binascii.unhexlify(myargs['-u'])
        plaintext = decrypt(key, ciphertext)
        print('Plaintext: ' + binascii.b2a_qp(plaintext))
    else:
        print("python ecb.py -e 010203040506")
        print("python ecb.py -s 'this is cool'")
        print("python ecb.py -d d25a16fe349cded7f6a2f2446f6da1c2")
        print("python ecb.py -u 9b43953eeb6c3b7b7971a8bec1a90819")
```

Finally, define main(). The program supports 4 modes: ‘-e’ encrypts a hexadecimal string into a hexadecimal string; ‘-d’ decrypts a hexadecimal string into a hexadecimal string; ‘-s’ encrypts a printable string into a hexadecimal string; ‘-u’ decrypts a hexadecimal string into a printable string.

The examples of the 4 use cases are given in the code.

Question: Type the code into ecb.py. Pay special attention to indentation!!

Test the program as follows: select 4 hexadecimal strings so that their last block contains padding of 01, 0202, 030303, 04040404, respectively. (1) Encrypt these strings: print the plaintexts after padding, also print the ciphertexts after encryption. (2) Decrypt the results of the first step: Do you get the original plaintexts? (3) Create a string whose length is exactly one block, repeat the string 3 times to construct a new string. Encrypt the new string: What does the padding block look like? Will you see repeated ciphertext blocks after encryption?

3. (1.5%) Implement a CBC encryption/decryption program using ECB mode encryption.

Question: Implement cbc.py to support CBC mode encryption and decryption. It is required that the CBC encryption/decryption is implemented by applying the ECB encryption/decryption on each individual block. The new program should also support 4 modes (-e, -d, -s, -u) as ecb.py. The AES key remains the same as ecb.py.

Test the program with the following input:

Congratulations! You have earned the extra credit!

What is the ciphertext?

4. (1%) Implement a CBC padding oracle.

Question: Modify the CBC decryption program to implement a CBC padding oracle: It accepts hexadecimal string as ciphertext, decrypt it, returns the "yes" or "no" to indicate the correctness of padding (with respect to PKCS#5 padding standard).

5. (1.5%) Conduct padding oracle attacks.

*Question: Use the padding oracle you implemented in question 4 to decrypt the **first block** of the ciphertext in question 3. Note, you cannot use your CBC decryption program developed in question 3, which gives you the decryption result directly!*

Tips: You only need to decrypt one block to get the credits of this question.

6. (1%) Extra Credit

Question: Decryption the entire cipher text in question 3. Discuss the difficulty you have encountered there.

7. Submission Instructions

Report your answers in one pdf file. You don't need to explain your code in the file, but show the test results and your thoughts (if any).

Besides the pdf file, submit 3 Python programs: cbc.py (for question 3), oracle.py (for question 4), attack.py (for question 5 and 6).

8. Code of Conduct

These labs are intended for educational purposes only, to provide a safe and legal means to gain an understanding of security by understanding threats and vulnerabilities. They are not intended for (and are not to be used for) any purposes other than for education.

Some of these labs are based on existing exploits, and students are to exploit their own virtual machines ONLY. Do not try them outside your personal devices. Use of anything learned in, during, or resulting from this class that is in any manner illegal, unauthorized, or unethical is forbidden. There are serious consequences for illegal computer hacking. Any student who violates the rules is subject to legal action, will take sole responsibility of his/her actions, and cannot hold any claim on the responsibility of the faculty, staff, or the university. Students who violate these conditions of the labs will get a failing grade in the class and may be subject to legal action.

Do not incorporate or implement viruses, worms, spyware and/or Trojan horses in ANY of these labs. Only the tools and resources specified in the given lab may be used. Any student who exploits fellow student's accounts or gains the solutions to the labs by means other than specified is engaging in academic misconduct. Academic misconduct will be treated seriously.