

Secure Communications

Week 5

Symmetric Key (Part 1)

Sections

A. OpenSSL

A.1 `systeminfo >> VolatileDataFile.txt openssl version`

```
[~(b08167321# kali)-[~]
[~$ openssl list -cipher-commands
aes-128-cbc      aes-128-ecb      aes-192-cbc      aes-192-ecb
aes-256-cbc      aes-256-ecb      aria-128-cbc      aria-128-cfb
aria-128-cfb1     aria-128-cfb8     aria-128-ctr      aria-128-ecb
aria-128-efb      aria-192-cbc      aria-192-cfb      aria-192-cfb1
aria-192-cfb8     aria-192-ctr      aria-192-ecb      aria-192-efb
aria-256-cbc      aria-256-cfb      aria-256-cfb1     aria-256-cfb8
aria-256-ctr      aria-256-ecb      aria-256-efb      bf
bf-cbc           bf-cfb           bf-ecb           bf-efb
camellia-128-cbc camellia-128-ecb camellia-192-cbc camellia-192-ecb
camellia-256-cbc camellia-256-ecb cast               cast-cbc
cast5-cbc        cast5-cfb        cast5-ecb        cast5-efb
des              des-cbc          des-cfb          des-ecb
des-cde          des-cde-cbc      des-cde-cfb      des-cde-efb
des-cde3         des-cde3-cbc     des-cde3-cfb     des-cde3-efb
des-efb          des3             desx             rc2
rc2-40-cbc       rc2-64-cbc       rc2-cbc          rc2-cfb
rc2-efb          rc2-efb          rc4              rc4-40
seed             seed-cbc         seed-cfb         seed-ecb
seed-efb         sm4-cbc          sm4-cfb          sm4-ctr
sm4-ecb          sm4-efb
```

```
[~(b08167321# kali)-[~]
[~$ openssl version
OpenSSL 3.5.0 8 Apr 2025 (Library: OpenSSL 3.5.0 8 Apr 2025)
```

— Outline five encryption methods that are supported:

→ **aes-128-cbc aria-128-cbc bf-cbc camellia-128-cbc des-cbc**

— Outline the version of OpenSSL:

→ **OpenSSL 3.5.0**

A.2 `openssl prime -hex 1111` `openssl prime 42` `openssl prime 1421`

```
(b08167321# kali)-[~]  
[~]$ openssl prime -hex 1111  
1111 (1111) is not prime  
  
(b08167321# kali)-[~]  
[~]$ openssl prime 42  
2A (42) is not prime  
  
(b08167321# kali)-[~]  
[~]$ openssl prime 1421  
58D (1421) is not prime  
  
(b08167321# kali)-[~]  
[~]$
```

— Check if the following are prime numbers:

- **1111 (1111) is not prime**
- **2A (42) is not prime**
- **58D (1421) is not prime**

A.3 `openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin`

```
(b08167321# kali)-[~]  
[~]$ echo "secret" > myfile.txt  
  
(b08167321# kali)-[~]  
[~]$ openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin  
Enter AES-256-CBC encryption password:  
Verifying - enter AES-256-CBC encryption password:  
*** WARNING : deprecated key derivation used.  
Using -iter or -pbkdf2 would be better.  
  
(b08167321# kali)-[~]  
[~]$ cat encrypted.bin  
Salted __X__20_____.  
  
(b08167321# kali)-[~]  
[~]$
```

— Is it easy to write out or transmit the output:

- **No**

A.4 `openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64`

```
(b0816732i# kali)-[~]
[~]$ echo "secret" > myfile.txt

(b0816732i# kali)-[~]
[~]$ openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

(b0816732i# kali)-[~]
[~]$ cat encrypted.bin
U2FsdGVkX189IY123uM1amkSgia/KCsy9IM6onNB6FI=

(b0816732i# kali)-[~]
[~]$
```

— Is it easy to write out or transmit the output:

→ **Yes**

A.5 `openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64`

```
(b0816732i# kali)-[~]
[~]$ echo "secret" > myfile.txt

(b0816732i# kali)-[~]
[~]$ openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

(b0816732i# kali)-[~]
[~]$ cat encrypted.bin
U2FsdGVkX189IY123uM1amkSgia/KCsy9IM6onNB6FI=

(b0816732i# kali)-[~]
[~]$ openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

(b0816732i# kali)-[~]
[~]$ cat encrypted.bin
U2FsdGVkX189IY123uM1amkSgia/KCsy9IM6onNB6FI=

(b0816732i# kali)-[~]
[~]$
```

— Has the output changed?

→ **Yes**

— Why has it changed?

→ **Because OpenSSL uses a random initialization vector (IV) for each encryption, ensuring different ciphertexts even for identical inputs.**

A.6 `openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:napier - base64`

```
(b0016732i# kali)-[~]
[~]$ openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:napier -base64
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
secret
[~]$
```

— Has the output been decrypted correctly?

→ **Yes**

— What happens when you use the wrong password?

→ **Decryption fails**

```
(b0016732i# kali)-[~]
[~]$ openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:napier -base64
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
bad decrypt
12042132770000:error:12000064:Provider routines:ossl_cipher_unpadblock:bad decrypt:~/providers/implementations/ciphers/ciphercommon_block.c:107:
```

A.7 `openssl enc -bf-cbc -in myfile.txt -out encrypted.bin`

`openssl enc -d -bf-cbc -in encrypted.bin -out decrypted.txt`

```
(b0016732i# kali)-[~]
[~]$ echo "secret" > myfile.txt
[~]$ cat myfile.txt
secret
[~]$

(b0016732i# kali)-[~]
[~]$ openssl enc -bf-cbc -in myfile.txt -out encrypted.bin
enter BF-CBC encryption password:
Verifying - enter BF-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[~]$ cat encrypted.bin
Salted _____
[~]$


(b0016732i# kali)-[~]
[~]$ openssl enc -d -bf-cbc -in encrypted.bin -out decrypted.txt
enter BF-CBC decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[~]$ cat decrypted.txt
secret
[~]$
```

— Did you manage to decrypt the you can decrypt it. file?

→ **Yes**

B. Padding (AES)


B.1 With AES which uses a 256-bit key, what is the normal...



bill's
A security site.com
 + profsims.com - Networksims


Welcome to Asecuritysitem.com

[HOME](#)
[INDEX](#)
[CIPHER](#)
[BLOGS](#)
[IP](#)
[IDS](#)
[MAGIC](#)
[NET](#)
[CISCO](#)
[CYBER](#)
[TEST](#)
[FUN](#)
[SUBJ](#)
[ABOUT](#)



Padding

[Encryption Home][Home]



Padding is used in a block cipher where we fill up the blocks with padding bytes. AES uses 128-bits (16 bytes), and DES uses 64-bit blocks (8 bytes). The main padding methods are:

- CMS (Cryptographic Message Syntax). This pads with the same value as the number of padding bytes. Defined in RFC 5652, PKCS#5, PKCS#7 (X.509 certificate) and RFC 1423 PEM.
- Bit. This pads with 0x80 (10000000) followed by zero (null) bytes. Defined in ANSI X.923 and ISO/IEC 9797-1.
- ZeroLength. This pads with zeros except for the last byte which is equal to the number (length) of padding bytes.
- Null. This pads with NULL bytes. This is only used with ASCII text.
- Space. This pads with spaces. This is only used with ASCII text.
- Random. This pads with random bytes with the last byte defined by the number of padding bytes.

— Block size (bytes):

→ **16 bytes (128-bits)**

— Number of hex characters for block size:

→ **32** (1 byte = 2 hex chars → $16 \times 2 = 32$)

B.2 message - "kettle" password - "oxtail"

```
In this implementation, only CMS has been implemented
After padding (CMS): 6b6574746c650a0a0a0a0a0a0a0a0a0a
Cipher (ECB): 61e14593c46cf5747d2c0e7fe78a3ec8
[YaFFK8R89XR91Ax/54c+yA==]
decrypt: kettle

After padding (Bit): 6b6574746c658000000000000000000000
Cipher (ECB): cc6203662893bd75c008a31d0953860
[zGIDZiITvbdcAtOx0JU4YA==]
decrypt: kettle

After padding (ZeroLen): 6b6574746c650000000000000000000000
Cipher (ECB): b6e70dd15ba0e98e5e745bd9b3c02827
[tucN0Vug6Y5edFvZs8AoJw==]
decrypt: kettle

After padding (Null): 6b6574746c650000000000000000000000
Cipher (ECB): 18eff3cb08f13139c517a25992fc246b
[GO/zywgfExnFF6JZkvwkaw==]
decrypt: kettle

After padding (Space): 6b6574746c6552020202020202020202020
Cipher (ECB): 8e460b692e3158d04e85cef7db6aeb2e
[jkYLaS4xWNBOhc732r2rLg==]
decrypt: kettle

After padding (Random): 6b6574746c6553678797a80249605309
Cipher (ECB): 96d70454d894498bffb921a7ee9c21fa
[ltcEVNIUSYv/uSgn7pwh+g==]
decrypt: kettle
```

— CMS:

→ **61e145** (61e14593c46cf5747d2c0c7fe78a3ec8)

— Null:

→ **18eff3** (18eff3cb081f1319c517a25992fc246b)

— Space:

→ **8e460b** (8e460b692e3158d04e85cef7db6aeb2e)

B.3 How many hex characters will be used for the 256-bit AES encryption

```
In this implementation, only CMS has been implemented
After padding (CMS): 666f780d0d0d0d0d0d0d0d0d0d0d0d0d
Cipher (ECB): 961068ddb012e940efe96e033f517cfd
[1hBo3bAS6UDv6W4DP1F8/Q==]
decrypt: fox
```

```
In this implementation, only CMS has been implemented
After padding (CMS): 666f7874726f74090909090909090909
Cipher (ECB): ea8e0d5147c3eda4d3cec4f06ef60652
[6o4NUUfD7aTTzsTwbvYGUG==]
decrypt: foxtrot
```

```
In this implementation, only CMS has been implemented
After padding (CMS): 666f7874726f74616e74656174657201
Cipher (ECB): 7947a0505f4abeeedf6d295aed3c10fc
[eUegUF9Kvu7fbS1a7TwQ/A==]
decrypt: foxtrotanteater
```

```
In this implementation, only CMS has been implemented
After padding (CMS): 666f7874726f74616e746561746572636173746c650b0b0b0b0b0b0b0b0b0b
Cipher (ECB): 69590213ee77a52938ab6370f1f02a65a8344a83393552683cccd0b6b049ea5
[aVkcE+53pSk4q2Nw8fAq2ag0SoM5NVJoPMzEC2sEngU=]
decrypt: foxtrotanteatercastle
```

— Number of hex characters:

- **“fox”**: 32 hex chars (3 bytes) → 1 block
- **“foxtrot”**: 32 hex chars (7 bytes) → 1 block
- **“foxtrotanteater”**: 32 hex chars (15 bytes) → 1 block
- **“foxtrotanteatercastle”**: 64 hex chars (21 bytes) → $\text{ceil}(21/16)=2$ blocks

B.4 With 256-bit AES, for n characters in a string, generalise the calculation of the number of...

— Hex characters:

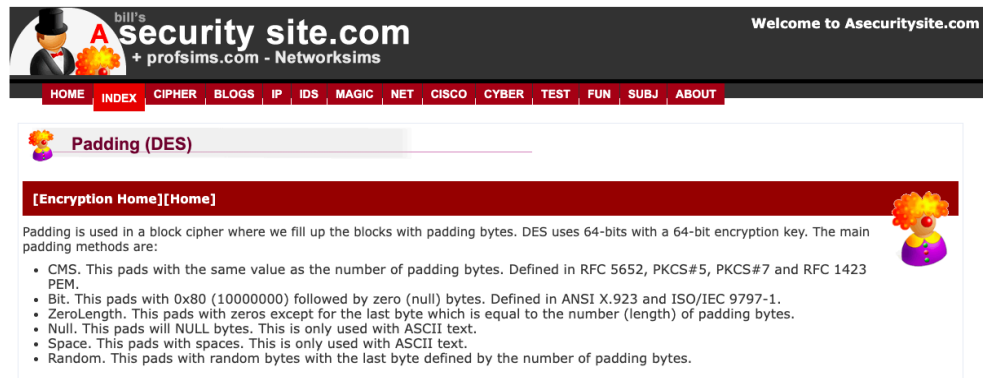
- $2 \times (16 \times \text{ceil}(n / 16)) = 32 \times \text{ceil}(n / 16)$

— Base-64 characters:

- $4 \times \text{ceil}(b / 3) = 4 \times \text{ceil}((16 \times \text{ceil}(n / 16)) / 3)$
- $b = 16 \times \text{ceil}(n / 16)$

C. Padding (AES)

C.1 With DES which uses a 64-bit key, what is the normal...



bill's **Asecurity site.com**
+ profsims.com - Networksims

Welcome to Asecuritysite.com

HOME INDEX CIPHER BLOGS IP IDS MAGIC NET CISCO CYBER TEST FUN SUBJ ABOUT

Padding (DES)

[Encryption Home][Home]

Padding is used in a block cipher where we fill up the blocks with padding bytes. DES uses 64-bits with a 64-bit encryption key. The main padding methods are:

- CMS. This pads with the same value as the number of padding bytes. Defined in RFC 5652, PKCS#5, PKCS#7 and RFC 1423 PEM.
- Bit. This pads with 0x80 (10000000) followed by zero (null) bytes. Defined in ANSI X.923 and ISO/IEC 9797-1.
- ZeroLength. This pads with zeros except for the last byte which is equal to the number (length) of padding bytes.
- Null. This pads with NULL bytes. This is only used with ASCII text.
- Space. This pads with spaces. This is only used with ASCII text.
- Random. This pads with random bytes with the last byte defined by the number of padding bytes.

— Block size (bytes):

→ **8 bytes (64-bits)**

— Number of hex characters for block size:

→ **16** (1 byte = 2 hex chars → $8 \times 2 = 16$)

C.2 message - "kettle" password - "oxtail"

```
In this implementation, only CMS has been implemented
DES
After padding (CMS): 6b6574746c650202
Cipher (ECB): 0d74d0510d32caaa
[DXTQUQ0yyqo=]
decrypt: kettle

After padding (Bit): 6b6574746c650202
Cipher (ECB): 0d74d0510d32caaa
[DXTQUQ0yyqo=]
decrypt: kettle

After padding (ZeroLen): 6b6574746c650202
Cipher (ECB): 0d74d0510d32caaa
[DXTQUQ0yyqo=]
decrypt: kettle

After padding (Null): 6b6574746c650202
Cipher (ECB): 0d74d0510d32caaa
[DXTQUQ0yyqo=]
decrypt: kettle

After padding (Space): 6b6574746c650000
Cipher (ECB): 8400ede37908c60c
[hADt43kIgw=]
decrypt: kettle0000000000

After padding (Random): 6b6574746c650202
Cipher (ECB): 0d74d0510d32caaa
[DXTQUQ0yyqo=]
decrypt: kettle
```

— CMS:

→ **0d74d0** (0d74d0510d32caaa)

— Null:

→ **0d74d0** (0d74d0510d32caaa)

— Space:

→ **8400ed** (8400ede37908c60c)

C.3 How many hex characters will be used for the 256-bit AES encryption

```
In this implementation, only CMS has been implemented
DES
After padding (CMS): 666f780505050505
Cipher (ECB): 9e9a68b4cecef3fd
[!ppotM708/0=]
decrypt: fox
```

```
In this implementation, only CMS has been implemented
DES
After padding (CMS): 666f7874726f7401
Cipher (ECB): 85af0285d8bcc6a1
[!ha8Chdi8xqE=]
decrypt: foxtrot
```

```
In this implementation, only CMS has been implemented
DES
After padding (CMS): 666f7874726f74616e74656174657201
Cipher (ECB): 0d2780ea10b35444e39ac485977c13b8
[!DSeA6hCzVETjmsSF13wTuA==]
decrypt: foxtrotanteater
```

```
In this implementation, only CMS has been implemented
DES
After padding (CMS): 666f7874726f74616e746561746572636173746c65030303
Cipher (ECB): 0d2780ea10b35444e39ac485977c13b8
[!DSeA6hCzVETjmsSF13wTuA==]
decrypt: foxtrotanteatercastle
```

— Number of hex characters:

- **“fox”**: 16 hex chars (3 bytes) → 1 block
- **“foxtrot”**: 16 hex chars (7 bytes) → 1 block
- **“foxtrotanteater”**: 32 hex chars (15 bytes) → 2 block
- **“foxtrotanteatercastle”**: 48 hex chars (21 bytes) → 3 blocks

C.4 With 64-bit DES, for n characters in a string, generalise the calculation of the number of...

— Hex characters:

- $2 \times (8 \times \text{ceil}(n / 8)) = 16 \times \text{ceil}(n / 8)$

— Base-64 characters:

- $4 \times \text{ceil}(b / 3) = 4 \times \text{ceil}((8 \times \text{ceil}(n / 8)) / 3)$
- $b = 8 \times \text{ceil}(n / 8)$

Objective: The key objective of this lab is to understand the range of symmetric key methods used within symmetric key encryption. We will introduce block ciphers, stream ciphers and padding. The key tools used include OpenSSL, Python and JavaScript. Overall Python 2.7 has been used for the sample examples, but it should be easy to convert these to Python 3.x.

Demo: <https://youtu.be/N3UADaXmOik>


OpenSSL is a standard tool that we used in encryption. It supports many of the standard symmetric key methods, including AES, 3DES and ChaCha20.

No	Description	Result
A.1	<p>Use:</p> <pre>openssl list-cipher-commands</pre> <p>openssl version</p>	<p>Outline five encryption methods that are supported:</p> <pre>aes-128-cbc aes-128-ecb aes-192-cbc aes-256-cbc aes-256-ecb</pre> <p>Outline the version of OpenSSL:</p> <pre>openssl 3.3.4</pre>
A.2	<p>Using openssl and the command in the form:</p> <pre>openssl prime -hex 1111</pre>	<p>Check if the following are prime numbers:</p> <pre>42 [Yes][No] 1421 [Yes][No]</pre>
A.3	<p>Now create a file named myfile.txt (either use Notepad or another editor).</p> <pre>Next encrypt with aes-256-cbc</pre> <pre>openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin</pre> <p>and enter your password.</p>	<p>Use the following command to view the output file:</p> <pre>cat encrypted.bin</pre> <p>Is it easy to write out or transmit the output: [Yes][No]</p>
A.4	<p>Now repeat the previous command and add the -base64 option.</p> <pre>openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64</pre>	<p>Use following command to view the output file:</p> <pre>cat encrypted.bin</pre> <p>Is it easy to write out or transmit the output: [Yes][No]</p>
A.5	<p>Now Repeat the previous command and observe the encrypted output.</p>	<p>Has the output changed? [Yes][No]</p>

	<pre>openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64</pre>	<p>Why has it changed?</p> <p>Blowfish is a stream cipher that encrypts your data byte by byte. It uses a key to generate a stream of random numbers, which are then combined with the data to encrypt it.</p>
A.6	<p>Now let's decrypt the encrypted file with the correct format:</p> <pre>openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:napter - base64</pre>	<p>Has the output been decrypted correctly? Yes</p> <p>What happens when you use the wrong password? Decrypting fails</p>
A.7	<p>Now encrypt a file with Blowfish and see if you can decrypt it.</p>	<p>Did you manage to decrypt the file? [Yes]/[No]</p>

With encryption, we normally use a block cipher, and where we must pad the end blocks to make sure that the data fits into a whole number of block. Some background material is here:

In the first part of this tutorial we will investigate padding blocks:

No	Description	Result
B.1	With AES which uses a 256-bit key, what is the normal block size (in bytes).	Block size (bytes): 32 bytes (256 bits) Number of hex characters for block size: 64 <small>1 byte = 2 hex characters</small>
B.2	Go to:  Web link (AES Padding): http://asecuritysite.com/encryption/padding Using 256-bit AES encryption, and a message of "kettle" and a password of "oxtail", determine the cipher using the differing padding methods (you only need to show the first six hex characters). If you like, copy and paste the Python code from the page, and run it on your Kali instance.	CMS: 0x0000 Null: 0x0000 Space: 0x0000
B.3	For the following words, estimate how many hex characters will be used for the 256-bit AES encryption:	Number of hex characters: "fox": 12 hex chars <small>6 bytes = 12 hex chars</small> "foxtrot": 12 hex chars <small>6 bytes = 12 hex chars</small> "foxtrotatleast": 12 hex chars <small>6 bytes = 12 hex chars</small>

		<div> <div>64 hex char</div> <div>128 hex = concatenated</div> <div>“foxtrotantecastle”;</div> </div>
B.4	<p>With 256-bit AES, for n characters in a string, how would you generalise the calculation of the number of hex characters in the cipher text.</p> <p>How many Base-64 characters would be used (remember 6 bits are used to represent a Base-64 character)?</p>	<div> <div>Hex characters:</div> <div> $2 \times (256 \times \text{ceil}(n / 32))$ $32 \times \text{ceil}(n / 16)$ </div> <div>Base-64 characters:</div> <div> $\frac{4 \times \text{ceil}(n / 3)}{3}$ $4 \times \text{ceil}(16 \times \text{ceil}(n / 256) / 3)$ $n = 16 \times \text{ceil}(n / 16)$ </div> </div>

In the first part of this lab we will investigate padding blocks:

No	Description	Result
C.1	With DES which uses a 64-bit key, what is the normal block size (in bytes):	<p>Block size (bytes): Number of hex characters for block size: 10 <code>16 bytes = 128 bits = 2⁷ bytes</code></p>
C.2	Go to: Web link (DES Padding): http://asecuritysite.com/encryption/padding_des Using 64-bit DES key encryption, and a message of "kettle" and a password of "oxtail", determine the cipher using the differing padding methods. If you like, copy and paste the Python code from the page, and run it on your Kali instance.	<p>CMS: 04f00910632e0000</p> <p>Null: 04f00910632e0000</p> <p>Space: 0400da3728000000</p>
C.3	For the following words, estimate how many hex characters will be used for the 64-bit key DES encryption:	<p>Number of hex characters:</p> <p>"fox": 16 hex chars <small>16 bytes = 128 bits</small></p> <p>"foxtrot": 16 hex chars <small>16 bytes = 128 bits</small></p> <p>"foxtrotantester": 32 hex chars <small>32 bytes = 256 bits</small></p> <p>"foxtrotantestercafe": 32 hex chars <small>32 bytes = 256 bits</small></p>
C.4	With 64-bit DES, for n characters in a string, how would you generalise the calculation of the number of hex characters in the cipher text.	<p>Hex characters: $16 \times n = \text{chars} / 8$ $16 \times n \text{ chars}$</p> <p>Base-64 characters: $\lceil \frac{16 \times n + 3}{4} \rceil$ $\lceil \frac{16 \times n + 3}{4} \rceil \times 4$</p>

	How many Base-64 characters would be used (remember 6 bits are used to represent a Base-64 character)?	
--	--	--

In this part of the lab, we will investigate the usage of Python code to perform different padding methods and using AES. First download the code from:

The code should be:

```
from Crypto.Cipher import AES
import hashlib
import sys
import binascii
import padding

val='hello'
password='hello'
plaintext=val

def encrypt(plaintext,key,mode):
    encobj = AES.new(key,mode)
    return(encobj.encrypt(padding(plaintext)))

def decrypt(ciphertext,key,mode):
    decobj = AES.new(key,mode)
    return(decobj.decrypt(ciphertext))

key = hashlib.sha256(password).digest()

plaintext = padding.appendpad(plaintext,AES.blocksize-AES.blocksize+padding.AES.MODE_CBC)
print "After padding (0x)": binascii.hexlify(bytesarray(plaintext))

ciphertext = encrypt(plaintext,key,AES.MODE_CFB)
print "Cipher (0x)": binascii.hexlify(bytesarray(ciphertext))

plaintext = decrypt(ciphertext,key,AES.MODE_CFB)
plaintext = padding.removepad(plaintext,mode='cm')
print "decrypt": plaintext

plaintext=wal
```

Now update the code so that you can enter a string and the program will show the cipher text. The format will be something like:

```
python cipher01.py hello mykey
```

where “hello” is the plain text, and “mykey” is the key. A possible integration is:

```
import sys
if (len(sys.argv)>1):
    val=sys.argv[1]
if (len(sys.argv)>2):
    password=sys.argv[2]
```

Now determine the cipher text for the following (the first example has already been completed):

4