Lab 2: Symmetric Key

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.

🕮 **Web link (Weekly activities):** https://asecuritysite.com/esecurity/unit02

Demo: https://youtu.be/N3UADaXmOik

A OpenSSL

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	Use:
	
	`openssl list-cipher-commands`
	
	`openssl version`
		Outline five encryption methods that are supported:
		
		Outline the version of OpenSSL:

A.2	Using openssl and the command in the form: `openssl prime –hex 1111`	Check if the following are prime numbers: 42 [Yes][No] 1421 [Yes][No]
A.3	Now create a file named myfile.txt (either use Notepad or another editor). Next encrypt with aes-256-cbc `openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin` and enter your password.	Use the following command to view the output file: `cat encrypted.bin` Is it easy to write out or transmit the output: [Yes][No]
A.4	Now repeat the previous command and add the –base64 option. `openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin –base64`	Use following command to view the output file: `cat encrypted.bin` Is it easy to write out or transmit the output: [Yes][No]
A.5	Now Repeat the previous command and observe the encrypted output.	Has the output changed? [Yes][No]

	`openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin –base64`	Why has it changed?
A.6	Now let's decrypt the encrypted file with the correct format: `openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:`*`napier`*` -base64`	Has the output been decrypted correctly? What happens when you use the wrong password?
A.7	Now encrypt a file with Blowfish and see if you can decrypt it.	Did you manage to decrypt the file? [Yes][No]

B Padding (AES)

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:

🕮 **Web link (Padding):** http://asecuritysite.com/encryption/padding

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): Number of hex characters for block size:
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: Null: Space:
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": "foxtrot": "foxtrotanteater":

		“foxtrotanteatercastle”:
B.4	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. How many Base-64 characters would be used (remember 6 bits are used to represent a Base-64 character):	Hex characters: Base-64 characters:

C Padding (DES)

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):	Block size (bytes): Number of hex characters for block size:
C.2	Go to: 🕮 **Web link (DES Padding):** http://asecuritysite.com/encryption/padding_des Using 64-bit DES 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.	CMS: Null: Space:
C.3	For the following words, estimate how many hex characters will be used for the 256-bit DES encryption:	Number of hex characters: “fox”: “foxtrot”: “foxtrotanteater”: “foxtrotanteatercastle”:
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. How many Base-64 characters would be used (remember 6 bits are used to represent a Base-64 character):	Hex characters: Base-64 characters:

D Python Coding (Encrypting)

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:

📖 **Web link (Cipher code):** http://asecuritysite.com/cipher01.zip

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(plaintext))

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

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


plaintext = Padding.appendPadding(plaintext,blocksize=Padding.AES_blocksize,mode='CMS')
print "After padding (CMS): "+binascii.hexlify(bytearray(plaintext))

ciphertext = encrypt(plaintext,key,AES.MODE_ECB)
print "Cipher (ECB): "+binascii.hexlify(bytearray(ciphertext))

plaintext = decrypt(ciphertext,key,AES.MODE_ECB)
plaintext = Padding.removePadding(plaintext,mode='CMS')
print "  decrypt: "+plaintext


plaintext=val
```

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):

Message	Key	CMS Cipher
"hello"	"hello123"	0a7ec77951291795bac6690c9e7f4c0d
"inkwell"	"orange"	

“security”	“qwerty”	
“Africa”	“changeme”	

Now copy your code and modify it so that it implements **64-bit DES** and complete the table (Ref to: http://asecuritysite.com/encryption/padding_des):

Message	Key	CMS Cipher
“hello”	“hello123”	8f770898ddb9fb38
“inkwell”	“orange”	
“security”	“qwerty”	
“Africa”	“changeme”	

Now modify the code so that the user can enter the values from the keyboard, such as with:

cipher=raw_input('Enter cipher:')
password=raw_input('Enter password:')

E Python Coding (Decrypting)

Now modify your coding for 256-bit AES ECB encryption, so that you can enter the cipher text, and an encryption key, and the code will decrypt to provide the result. You should use CMS for padding. With this, determine the plaintext for the following (note, all the plain text values are countries around the World):

CMS Cipher (256-bit AES ECB)	Key	Plain text
b436bd84d16db330359edebf49725c62	“hello”	
4bb2eb68fccd6187ef8738c40de12a6b	“ankle”	
029c4dd71cdae632ec33e2be7674cc14	“changeme”	
d8f11e13d25771e83898efdbad0e522c	“123456”	

Now modify your coding for 64-bit DES ECB encryption, so that you can enter the cipher text, and an encryption key, and the code will decrypt to provide the result. You should use CMS for padding. With this, determine the plaintext for the following (note, all the plain text values are countries around the World):

CMS Cipher (64-bit DES ECB)	Key	Plain text
f37ee42f2267458d	“hello”	
67b7d1162394b868	“ankle”	
ac9feb702ba2ecc0	“changeme”	

de89513fbd17d0dc	**“123456”**	

Now update your program, so that it takes a cipher string in Base-64 and converts it to a hex string and then decrypts it. From this now decrypt the following Base-64 encoded cipher streams (which should give countries of the World):

CMS Cipher (256-bit AES ECB)	Key	Plain text
/vA6BD+ZXu8j6KrTHi1Y+w==	**“hello”**	
nitTRpxMhGlaRkuyXWYxtA==	**“ankle”**	
irwjGCAu+mmdNeu6Hq6ciw==	**“changeme”**	
5I71KpfT6RdM/xhUJ5IKCQ==	**“123456”**	

PS … remember to add “`import base64`”.

F Catching exceptions

If we try “1jDmCTD1IfbXbyyHgAyrdg==” with a passphrase of “hello”, we should get a country. What happens when we try the wrong passphrase?

Output when we use “hello”: Output when we use “hello1”:

Now catch the exception with an exception handler:

```
try:
      plaintext = Padding.removePadding(plaintext,mode='CMS')
      print "  decrypt: "+plaintext
except:
      print("Error!")
```

Now implement a Python program which will try various keys for a cipher text input, and show the decrypted text. The keys tried should be:

["hello","ankle","changeme","123456"]

Run the program and try to crack:

“`1jDmCTD1IfbXbyyHgAyrdg==`”

What is the password:

G Stream Ciphers

The Chacha20 cipher is a stream cipher which uses a 256-bit key and a 64-bit nonce. Currently AES has a virtual monopoly on secret key encryption. There would be major problems, though, if this was cracked. Along with this AES has been shown to be weak around cache-collision attacks. Google thus propose ChaCha20 as an alternative, and actively use it within TLS connections. Currently it is three times faster than software-enabled AES and is not sensitive to timing attacks. It operates by creating a key stream which is then X-ORed with the plaintext. It has been standardised with RFC 7539.

We can use node.js to implement ChaCha20:

```
var chacha20 = require("chacha20");

var key = new Buffer.alloc(32);
key.fill(0);
var nonce = new Buffer.alloc(8);
nonce.fill(0);

var plaintext = "testing";

var args = process.argv;
if (args.length>1) plaintext=args[2];


var ciphertext = chacha20.encrypt(key, nonce, new Buffer.from(plaintext));
console.log("Ciphertext:\t",ciphertext.toString("hex"));
console.log("Decipher\t",chacha20.decrypt(key,nonce,
     ciphertext).toString());
```

If we use a key of "qwerty", can you find the well-known fruits (in lower case) of the following ChaCha20 cipher streams:

```
e47a2bfe646a
ea783afc66
e96924f16d6e
```

What are the fruits?
What can you say about the length of the cipher stream as related to the plaintext?
How are we generating the key?
What is the salt used in this example?
How many bits will the salt use?

Download and run the JavaScript code within this page:

📖 **Web link (ChaCha20 code):** https://asecuritysite.com/encryption/chacha

and prove - for a key of "0…0" and a nonce of "0..0" - that you get a key stream of:

```
76b8e0ada0f13d90405d6ae55386bd28bdd219b8a08ded1aa836efcc8b770dc7da41597c51
57488d7724e03fb8d84a376a43b8f41518a11cc387b669
```

Search the Web for another test vector for ChaCha20 and show that you get the same results.

H Node.js for encryption

Node.js can be used as a back-end encryption method. In the following we use the crypto module (which can be installed with "**npm crypto**", if it has not been installed). The following defines a message, a passphrase and the encryption method.

```
var crypto = require("crypto");


function encryptText(algor, key, iv, text, encoding) {

        var cipher = crypto.createCipheriv(algor, key, iv);

        encoding = encoding || "binary";

        var result = cipher.update(text, "utf8", encoding);
        result += cipher.final(encoding);

        return result;
    }

function decryptText(algor, key, iv, text, encoding) {

        var decipher = crypto.createDecipheriv(algor, key, iv);

        encoding = encoding || "binary";

        var result = decipher.update(text, encoding);
        result += decipher.final();

        return result;
    }


var data = "This is a test";
var password = "hello";
var algorithm = "aes256"

\\const args = process.argv.slice(3);

\\data = args[0];
\\password = args[1];
\\algorithm = args[2];

console.log("\nText:\t\t" + data);
console.log("Password:\t" + password);
console.log("Type:\t\t" + algorithm);

var hash,key;
```

```
if (algorithm.includes("256"))
{
        hash = crypto.createHash('sha256');
         hash.update(password);


        key = new Buffer.alloc(32,hash.digest('hex'),'hex');
}
else if (algorithm.includes("192"))
{
        hash = crypto.createHash('sha192');
         hash.update(password);

        key = new Buffer.alloc(24,hash.digest('hex'),'hex');
}

else if (algorithm.includes("128"))
{
        hash = crypto.createHash('md5');
         hash.update(password);

        key = new Buffer.alloc(16,hash.digest('hex'),'hex');
}


const iv=new Buffer.alloc(16,crypto.pseudoRandomBytes(16));

console.log("Key:\t\t"+key.toString('base64'));
console.log("Salt:\t\t"+iv.toString('base64'));

var encText = encryptText(algorithm, key, iv, data, "base64");

console.log("\n===============");

console.log("\nEncrypted:\t" + encText);

var decText = decryptText(algorithm, key, iv, encText, "base64");

console.log("\nDecrypted:\t" + decText);
```

Save the file as "h_01.js" and run the program with:

```
node h_01.js
```

Now complete the following table:

Text	Pass phrase	Type	Ciphertext and salt (just define first four characters of each)
This is a test	hello	Aes128	
France	Qwerty123	Aes192	
Germany	Testing123	Aes256	

Now reset the IV (the salt value) to an empty string (""), and complete the table:

Text	Pass phrase	Type	Ciphertext
This is a test	hello	Aes128	

France	Qwerty123	`Aes192`	
Germany	Testing123	`Aes256`	

Does the ciphertext change when we have a fixed IV value?

Using an Internet search, list ten other encryption algorithms which can be used with `createCipheriv`:

I Reflective questions

1. If we have five 'a' values ("aaaaa"). What will be the padding value used for 256-bit AES with CMS:

2. If we have six 'a' values ("aaaaaa"). What will be the hex values used for the plain text:

3. The following cipher text is 256-bit AES ECB for a number of spaces (0x20):

```
c3f791fad9f9392116b2d12c8f6c4b3dc3f791fad9f9392116b2d12c8f6c4b
3dc3f791fad9f9392116b2d12c8f6c4b3dc3f791fad9f9392116b2d12c8f6c
4b3da3c788929dd8a9022bf04ebf1c98a4e4
```

What can you observe from the cipher text:

What is the range that is possible for the number of spaces which have been used:

How might you crack a byte stream sequence like this:

4. For ChaCha20, we only generate a key stream. How is the ciphertext then created:

J What I should have learnt from this lab?

The key things learnt:

- How to encrypt and decrypt with symmetric key encryption, and where we use a passphrase to generate the encryption key.
- How padding is used within the encryption and decryption processes.
- The core difference between a block cipher and a stream cipher.

Notes

The code can be downloaded from:

```
git clone https://github.com/billbuchanan/esecurity
```

If you need to update the code, go into the esecurity folder, and run:

```
git pull
```

To install a Python library use:

```
pip install libname
```

To install a Node.js package, use:

```
npm install libname
```

Possible solutions

Have a look at: `https://asecuritysite.com/esecurity/labcode`