

# A 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. Overall Python 2.7 has been used for the sample examples, but it should be easy to convert these to Python 3.x.

## 1 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. Let's start with this code:

```
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"	

## 2 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 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"	
niTTRpxMhG1aRkuyxWYxtA==	"ankle"	
i rwjGCAu+mmdNeu6Hq6ciw==	"changeme"	
5I71Kpft6RdM/xhUJ5IKCQ==	"123456"	

PS ... remember to add `"import base64"`.

## 3 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:

“1jDmCTD1IfbXbyyHgAyr dg==”

What is the password:

## 4 Stream Ciphers

The ChaCha20 cipher is a stream cipher which uses a 256-bit key and a 64-bit nonce (salt value). 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 crypto = require('crypto');

var keyname="test";
var plaintext = "testing";

var args = process.argv;
if (args.length>1) plaintext=args[2];
if (args.length>2) keyname=args[3];

var key = crypto.createHash('sha256').update(keyname).digest();
var nonce = new Buffer.alloc(8);
nonce.fill(0);
```

```
console.log( key);  
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 and what is the key length?

What is the first two bytes of the key if we use a pass-phrase of “qwerty”?

What is the salt used in the same code?

How would you change the program so that the cipher stream was shown in in Base64?

How many bits will the salt use? You may have to look at the node.js documentation on the method for this.

## 5 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:

<b>Text</b>	<b>Pass phrase</b>	<b>Type</b>	<b>Ciphertext and salt (just define first four characters of each)</b>
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:

<b>Text</b>	<b>Pass phrase</b>	<b>Type</b>	<b>Ciphertext</b>
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`:

# B Hashing

In this section we will look at some fundamental hashing methods.

## 1 LM Hash

The LM Hash is used in Microsoft Windows. For example, for LM Hash:

hashme gives: FA-91-C4-FD-28-A2-D2-57-AA-D3-B4-35-B5-14-04-EE  
network gives: D7-5A-34-5D-5D-20-7A-00-AA-D3-B4-35-B5-14-04-EE  
napier gives: 12-B9-C5-4F-6F-E0-EC-80-AA-D3-B4-35-B5-14-04-EE

Notice that the right-most element of the hash are always the same, if the password is less than eight characters. With more than eight characters we get:

networksims gives: D7-5A-34-5D-5D-20-7A-00-38-32-A0-DB-BA-51-68-07  
napier123 gives: 67-82-2A-34-ED-C7-48-92-B7-5E-0C-8D-76-95-4A-50

For “hello” we get:

LM: FD-A9-5F-BE-CA-28-8D-44-AA-D3-B4-35-B5-14-04-EE  
NTLM: 06-6D-DF-D4-EF-0E-9C-D7-C2-56-FE-77-19-1E-F4-3C

We can check these with a Python script:

```
import passlib.hash;
string="hello"
print "LM Hash:"+passlib.hash.lmhash.encrypt(string)
print "NT Hash:"+passlib.hash.nthash.encrypt(string)
```

which gives:

```
LM Hash:fda95fbeca288d44aad3b435b51404ee
NT Hash:066ddfd4ef0e9cd7c256fe77191ef43c
```

 **Web link (LM Hash):** <http://asecuritysite.com/encryption/lmhash>

No	Description	Result
B.1	Create a Python script to determine the LM hash and NTLM hash of the following words:	“Napier”  “Foxtrot”

## 2 APR1

The Apache-defined APR1 format addresses the problems of brute forcing an MD5 hash, and basically iterates over the hash value 1,000 times. This considerably slows an intruder as they try to crack the hashed value. The resulting hashed string contains “\$apr1\$” to identify it and

uses a 32-bit salt value. We can use both `htpasswd` and `Openssl` to compute the hashed string (where “bill” is the user and “hello” is the password):

```
# htpasswd -nbm bill hello
bill:$apr1$Pkwj6gM4$XGwpADBVPyypjL/cL0XMc1

# openssl passwd -apr1 -salt Pkwj6gM4 hello
$apr1$Pkwj6gM4$XGwpADBVPyypjL/cL0XMc1
```

We can also create a simple Python program with the `passlib` library, and add the same salt as the example above:

```
import passlib.hash;

salt="Pkwj6gM4"
string="hello"
print "APR1:"+passlib.hash.apr_md5_crypt.encrypt(string, salt=salt)
```

We can create a simple Python program with the `passlib` library, and add the same salt as the example above:

```
APR1:$apr1$Pkwj6gM4$XGwpADBVPyypjL/cL0XMc1
```

Refer to: <http://asecuritysite.com/encryption/apr1>

No	Description	Result
B.2	Create a Python script to create the APR1 hash for the following:  Prove them against on-line APR1 generator (or from the page given above).	“changeme”:  “123456”:  “password”

### 3 SHA

While APR1 has a salted value, the SHA-1 hash does not have a salted value. It produces a 160-bit signature, thus can contain a larger set of hashed value than MD5, but because there is no salt it can be cracked to rainbow tables, and also brute force. The format for the storage of the hashed password on Linux systems is:

```
# htpasswd -nbs bill hello
bill:{SHA}qvTGHdzF6KLavt4P00gs2a6pQ00=
```

We can also generate salted passwords with `crypt`, and can use the Python script of:

```
import passlib.hash;
salt="8sFt66rZ"
string="hello"
print "SHA1:"+passlib.hash.sha1_crypt.encrypt(string, salt=salt)
print "SHA256:"+passlib.hash.sha256_crypt.encrypt(string, salt=salt)
print "SHA512:"+passlib.hash.sha512_crypt.encrypt(string, salt=salt)
```

SHA-512 salts start with `$6$` and are up to 16 chars long.



SHA-256 salts start with \$5\$ and are up to 16 chars long

Which produces:

```
SHA1:$sha1$480000$8sFt66rZ$k1AZf7IPWRN1ACGNZIMxxuVaIKRj
SHA256:$5$rounds=535000$8sFt66rZ$.YYuHL27JtcOX8wpjwkf2VM876kLTGZSHwCBbq9x
TD
SHA512:$6$rounds=656000$8sFt66rZ$aMTKQH160VXFjiDAsyNFxn4gRezzOZarxHaK.TcpV
YLpMw6MnX01yPQU06SSvmSdmF/VNbvPkkmpOEONvSd5Q1
```

No	Description	Result
B.3	Create a Python script to create the SHA hash for the following:  Prove them against on-line SHA generator (or from the page given above).	“changeme”:  “123456”:  “password”

## 4 PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) is defined in RFC 2898 and generates a salted hash. Often this is used to create an encryption key from a defined password, and where it is not possible to reverse the password from the hashed value. It is used in TrueCrypt to generate the key required to read the header information of the encrypted drive, and which stores the encryption keys.

PBKDF2 is used in WPA-2 and TrueCrypt. Its main focus is to produce a hashed version of a password and includes a salt value to reduce the opportunity for a rainbow table attack. It generally uses over 1,000 iterations in order to slow down the creation of the hash, so that it can overcome brute force attacks. The generalised format for PBKDF2 is:

$$DK = \text{PBKDF2}(\text{Password}, \text{Salt}, \text{MIterations}, \text{dkLen})$$

where Password is the pass phrase, Salt is the salt, MIterations is the number of iterations, and dklen is the length of the derived hash.

In WPA-2, the IEEE 802.11i standard defines that the pre-shared key is defined by:

$$PSK = \text{PBKDF2}(\text{PassPhrase}, \text{ssid}, \text{ssidLength}, 4096, 256)$$

In TrueCrypt we use PBKDF2 to generate the key (with salt) and which will decrypt the header, and reveal the keys which have been used to encrypt the disk (using AES, 3DES or Twofish). We use:

```
byte[] result = passwordDerive.GenerateDerivedKey(16,
    ASCIIEncoding.UTF8.GetBytes(message), salt, 1000);
```

which has a key length of 16 bytes (128 bits - dklen), uses a salt byte array, and 1000 iterations of the hash (Minterations). The resulting hash value will have 32 hexadecimal characters (16 bytes).

 **Web link (PBKDF2):** <http://www.asecuritysite.com/encryption/PBKDF2>

```
import hashlib;
import passlib.hash;
import sys;

salt="ZDzPE45C"
string="password"

if (len(sys.argv)>1):
    string=sys.argv[1]

if (len(sys.argv)>2):
    salt=sys.argv[2]

print "PBKDF2 (SHA1):"+passlib.hash.pbkdf2_sha1.encrypt(string, salt=salt)
print "PBKDF2 (SHA256):"+passlib.hash.pbkdf2_sha256.encrypt(string, salt=salt)
```

No	Description	Result
<b>B.4</b>	Create a Python script to create the PBKDF2 hash for the following (uses a salt value of "ZDzPE45C"). You just need to list the first six hex characters of the hashed value.	"changeme":  "123456":  "password"

## 5 Bcrypt

MD5 and SHA-1 produce a hash signature, but this can be attacked by rainbow tables. Bcrypt (Blowfish Crypt) is a more powerful hash generator for passwords and uses salt to create a non-recurrent hash. It was designed by Niels Provos and David Mazieres, and is based on the Blowfish cipher. It is used as the default password hashing method for BSD and other systems.

Overall it uses a 128-bit salt value, which requires 22 Base-64 characters. It can use a number of iterations, which will slow down any brute-force cracking of the hashed value. For example, "Hello" with a salt value of "\$2a\$06\$NkYh0RCM8pNWPAYvRLgN9." gives:

**\$2a\$06\$NkYh0RCM8pNWPAYvRLgN9.LbJw4gcnWCOQYIom0P08UEZRQQjbfpy**

As illustrated in Figure 1, the first part is "\$2a\$" (or "\$2b\$"), and then followed by the number of rounds used. In this case is it **6 rounds** which is  $2^6$  iterations (where each additional round doubles the hash time). The 128-bit (22 character) salt values comes after this, and then finally there is a 184-bit hash code (which is 31 characters).

The slowness of bcrypt is highlighted with an AWS EC2 server benchmark using hashcat:

- Hash type: MD5 Speed/sec: 380.02M words
- Hash type: SHA1 Speed/sec: 218.86M words

- Hash type: SHA256 Speed/sec: 110.37M words
- Hash type: bcrypt, Blowfish(OpenBSD) Speed/sec: 25.86k words
- Hash type: NTLM. Speed/sec: 370.22M words

You can see that Bcrypt is almost 15,000 times slower than MD5 (380,000,000 words/sec down to only 25,860 words/sec). With John The Ripper:

- md5crypt [MD5 32/64 X2] 318237 c/s real, 8881 c/s virtual
- bcrypt ("2a\$05", 32 iterations) 25488 c/s real, 708 c/s virtual
- LM [DES 128/128 SSE2-16] 88090K c/s real, 2462K c/s virtual

where you can see that BCrypt over 3,000 times slower than LM hashes. So, although the main hashing methods are fast and efficient, this speed has a down side, in that they can be cracked easier. With Bcrypt the speed of cracking is considerably slowed down, with each iteration doubling the amount of time it takes to crack the hash with brute force. If we add one onto the number of rounds, we double the time taken for the hashing process. So, to go from 6 to 16 increase by over 1,000 ( $2^{10}$ ) and from 6 to 26 increases by over 1 million ( $2^{20}$ ).

The following defines a Python script which calculates a whole range of hashes:

```
import hashlib;
import passlib.hash;

salt="ZDzPE45C"
string="password"
salt2="11111111111111111111"

print "General Hashes"
print "MD5:"+hashlib.md5(string).hexdigest()
print "SHA1:"+hashlib.sha1(string).hexdigest()
print "SHA256:"+hashlib.sha256(string).hexdigest()
print "SHA512:"+hashlib.sha512(string).hexdigest()

print "UNIX hashes (with salt)"
print "DES:"+passlib.hash.des_crypt.encrypt(string, salt=salt[:2])
print "MD5:"+passlib.hash.md5_crypt.encrypt(string, salt=salt)
print "Sun MD5:"+passlib.hash.sun_md5_crypt.encrypt(string, salt=salt)
print "SHA1:"+passlib.hash.sha1_crypt.encrypt(string, salt=salt)
print "SHA256:"+passlib.hash.sha256_crypt.encrypt(string, salt=salt)
print "SHA512:"+passlib.hash.sha512_crypt.encrypt(string, salt=salt)
print "Bcrypt:"+passlib.hash.bcrypt.encrypt(string, salt=salt2[:22])
```

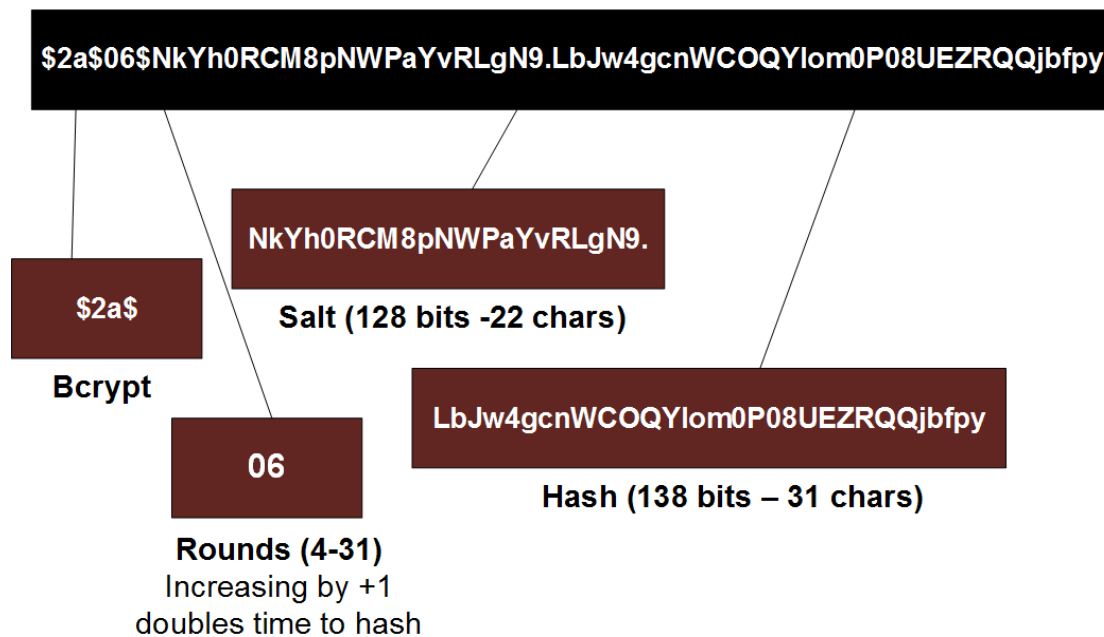


Figure 1 Bcrypt

No	Description	Result
B.5	<p>Create the hash for the word “hello” for the different methods (you only have to give the first six hex characters for the hash):</p> <p>Also note the number hex characters that the hashed value uses:</p>	<p>MD5:</p> <p>SHA1:</p> <p>SHA256:</p> <p>SHA512:</p> <p>DES:</p> <p>MD5:</p> <p>Sun MD5:</p> <p>SHA-1:</p> <p>SHA-256:</p> <p>SHA-512:</p>

## 6 HMAC

Write a Python or Node.js program which will prove the following:

Type:	HMAC-MD5
Message:	Hello
Password:	qwerty123
Hex:	c3a2fa8f20dee654a32c30e666cec48e
Base64:	7376b67daf1fdb475e7bae786b7d9cdf47baeba71e738f1e

If you get this to work, can you expand to include other MAC methods. You can test against this page:

<https://asecuritysite.com/encryption/js10>

## 7 Additional

The following provides a hash most of the widely used hashing method. For this enter the code of:

```

import hashlib;
import passlib.hash;
import sys;

salt="ZDzPE45C"
string="password"
salt2="11111111111111111111111111111111"

if (len(sys.argv)>1):
    string=sys.argv[1]

if (len(sys.argv)>2):
    salt=sys.argv[2]

print "General Hashes"
print "MD5:"+hashlib.md5(string).hexdigest()
print "SHA1:"+hashlib.sha1(string).hexdigest()
print "SHA256:"+hashlib.sha256(string).hexdigest()
print "SHA512:"+hashlib.sha512(string).hexdigest()

print "UNIX hashes (with salt)"
print "DES:"+passlib.hash.des_crypt.encrypt(string, salt=salt[:2])
print "MD5:"+passlib.hash.md5_crypt.encrypt(string, salt=salt)
print "Sun MD5:"+passlib.hash.sun_md5_crypt.encrypt(string, salt=salt)
print "SHA1:"+passlib.hash.sha1_crypt.encrypt(string, salt=salt)
print "SHA256:"+passlib.hash.sha256_crypt.encrypt(string, salt=salt)
print "SHA512:"+passlib.hash.sha512_crypt.encrypt(string, salt=salt)

print "APR1:"+passlib.hash.apr_md5_crypt.encrypt(string, salt=salt)
print "PHPASS:"+passlib.hash.phpass.encrypt(string, salt=salt)
print "PBKDF2 (SHA1):"+passlib.hash.pbkdf2_sha1.encrypt(string, salt=salt)
print "PBKDF2 (SHA256):"+passlib.hash.pbkdf2_sha256.encrypt(string, salt=salt)
#print "PBKDF2 (SHA512):"+passlib.hash.pbkdf2_sha512.encrypt(string, salt=salt)
#print "CTA PBKDF2:"+passlib.hash.cta_pbkdf2_sha1.encrypt(string, salt=salt)
#print "DLITZ PBKDF2:"+passlib.hash.dlitz_pbkdf2_sha1.encrypt(string, salt=salt)

print "MS windows Hashes"
print "LM Hash:"+passlib.hash.lmhash.encrypt(string)
print "NT Hash:"+passlib.hash.nthash.encrypt(string)
print "MS DCC:"+passlib.hash.msdcc.encrypt(string, salt)
print "MS DCC2:"+passlib.hash.msdcc2.encrypt(string, salt)

#print "LDAP Hashes"
#print "LDAP (MD5):"+passlib.hash.ldap_md5.encrypt(string)
#print "LDAP (MD5 Salted):"+passlib.hash.ldap_md5_salt.encrypt(string, salt=salt)
#print "LDAP (SHA):"+passlib.hash.ldap_sha1.encrypt(string)
#print "LDAP (SHA1 Salted):"+passlib.hash.ldap_salt_encrypt(string, salt=salt)
#print "LDAP (DES Crypt):"+passlib.hash.ldap_des_crypt.encrypt(string)
#print "LDAP (BSDI Crypt):"+passlib.hash.ldap_bsd_crypt.encrypt(string)
#print "LDAP (MD5 Crypt):"+passlib.hash.ldap_md5_crypt.encrypt(string)
#print "LDAP (Bcrypt):"+passlib.hash.ldap_bcrypt.encrypt(string)
#print "LDAP (SHA1):"+passlib.hash.ldap_sha1_crypt.encrypt(string)
#print "LDAP (SHA256):"+passlib.hash.ldap_sha256_crypt.encrypt(string)
#print "LDAP (SHA512):"+passlib.hash.ldap_sha512_crypt.encrypt(string)

print "LDAP (Hex MD5):"+passlib.hash.ldap_hex_md5.encrypt(string)
print "LDAP (Hex SHA1):"+passlib.hash.ldap_hex_sha1.encrypt(string)
print "LDAP (At Lass):"+passlib.hash.atlassian_pbkdf2_sha1.encrypt(string)
print "LDAP (FSHP):"+passlib.hash.fshp.encrypt(string)

print "Database Hashes"
print "MS SQL 2000:"+passlib.hash.mssql2000.encrypt(string)
print "MS SQL 2000:"+passlib.hash.mssql2005.encrypt(string)
print "MS SQL 2000:"+passlib.hash.mysql323.encrypt(string)
print "MySQL:"+passlib.hash.mysql41.encrypt(string)
print "Postgres (MD5):"+passlib.hash.postgres_md5.encrypt(string, user=salt)
print "Oracle 10:"+passlib.hash.oracle10.encrypt(string, user=salt)
print "Oracle 11:"+passlib.hash.oracle11.encrypt(string)

print "Other Known Hashes"
print "Cisco PIX:"+passlib.hash.cisco_pix.encrypt(string, user=salt)
print "Cisco Type 7:"+passlib.hash.cisco_type7.encrypt(string)
print "Django DES:"+passlib.hash.django_des_crypt.encrypt(string, salt=salt)

```

```
print "Dyango MD5:" +passlib.hash.django_salt_md5.encrypt(string, salt=salt[:2])
print "Dyango SHA1:" +passlib.hash.django_salt_sha1.encrypt(string, salt=salt)
print "Dyango Bcrypt:" +passlib.hash.django_bcrypt.encrypt(string, salt=salt2[:22])
print "Dyango PBKDF2 SHA1:" +passlib.hash.django_pbkdf2_sha1.encrypt(string,
salt=salt)
print "Dyango PBKDF2 SHA1:" +passlib.hash.django_pbkdf2_sha256.encrypt(string,
salt=salt)
print "Bcrypt:" +passlib.hash.bcrypt.encrypt(string, salt=salt2[:22])
```

No	Description	Result
B.6	<p>In the code, what does the modifier of “[:22]” do?</p> <p>In running the methods, which of them take the longest time to compute?</p> <p>Of the methods used, outline how you would identify some of the methods. For APR1 has an identifier of \$apr1\$.</p>	

For the following identify the hash methods used:

- [illegible]

# C Key Exchange

## 1 Discrete Logarithms

---

**C.1** ElGamal and Diffie Hellman use discrete logarithms. This involves a generator value ( $g$ ) and a prime number. A basic operation is  $g_x \pmod{p}$ . If  $p=11$ , and  $g=2$ , determine the results (the first two have already been completed):

x	$g=2, p=11$ $g_x \pmod{p}$
1	2
2	4
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Note: In Python you can implement this as:

```
g=2
p=11
x=3
print g**x % p
```

What happens to the values once we go past 10?

What happens to this sequence if we use  $g=3$ ?

**C.2** We can determine the values of  $g$  which will work for a given prime number with the following:

```
import sys
import random

p=11
```

```
def getG(p):
    for x in range (1,p):
        rand = x
        exp=1
        next = rand % p

        while (next <> 1 ):
            next = (next*rand) % p
            exp = exp+1

        if (exp==p-1):
            print rand

print getG(p)
```

Run the program and determine the possible g values for these prime numbers:

p=11:

p=41:

On the Internet, find a large prime number, and determine the values of g that are possible:

**C.3** We can write a Python program to implement this key exchange. Enter and run the following program:

```
import random
import base64
import hashlib
import sys

g=9
p=997

a=random.randint(5, 10)
b=random.randint(10,20)

A = (g**a) % p
B = (g**b) % p

print 'g: ',g,' (a shared value), n: ',p, ' (a prime number)'

print '\nAlice calculates:'
print 'a (Alice random): ',a
print 'Alice value (A): ',A,' (g^a) mod p'

print '\nBob calculates:'
print 'b (Bob random): ',b
print 'Bob value (B): ',B,' (g^b) mod p'

print '\nAlice calculates:'
keyA=(B**a) % p
print 'Key: ',keyA,' (B^a) mod p'
```



```

print 'Key: ',hashlib.sha256(str(keyA)).hexdigest()

print '\nBob calculates:'
keyB=(A**b) % p
print 'Key: ',keyB,' (A^b) mod p'
print 'Key: ',hashlib.sha256(str(keyB)).hexdigest()

```

Pick three different values for  $g$  and  $p$ , and make sure that the Diffie Hellman key exchange works.

$g =$        $p =$

$g =$        $p =$

$g =$        $p =$

Can you pick a value of  $g$  and  $p$  which will not work?

The following program sets up a man-in-the-middle attack for Eve:

```

import random
import base64
import hashlib
import sys

g=15
p=1011

a= 5
b = 9
eve = 7

message=21

A=(g**a) % p
B=(g**b) % p

Eve1 = (A**eve) % p
Eve2 = (B**eve) % p

Key1= (Eve1**a) % p
Key2= (Eve2**b) % p

print 'g: ',g,' (a shared value), n: ',p,' (a prime number)'

print '\n== Random value generation =='

print '\nAlice calculates:'
print 'a (Alice random): ',a
print 'Alice value (A): ',A,' (g^a) mod p'

print '\nBob calculates:'
print 'b (Bob random): ',b

```

```


print 'Bob value (B): ',B,' (g^b) mod p'
print '\n==Alice sends value to Eve ==='
print 'Eve takes Alice\'s value and calculates: ',Eve1
print 'Alice gets Eve\'s value and calculates key of: ',key1
print '\n==Bob sends value to Eve ==='
print 'Eve takes Bob\'s value and calculates: ',Eve2
print 'Bob gets Eve\'s value and calculates key of: ',key2

```

## 2 Elliptic Curve Diffie-Hellman (ECDH)

ECDH is now one of the most used key exchange methods, and uses the Diffie Hellman method, but adds in elliptic curve methods. With this Alice generates (a) and Bob generates (b). We select a point on a curve (G), and Alice generates aG, and Bob generates bG. They pass the values to each other, and then Alice received bG, and Bob receives aG. Alice multiplies by a, to get abG, and Bob will multiply by b, and also get abG. This will be their shared key.

**C.4** Copy and paste the code from (you may have to run “pip install eccsnacks”):

 **Web link (ECDH):** <https://asecuritysite.com/encryption/curve>

and confirm that Bob and Alice will always get the same shared key.

```

from os import urandom
from eccsnacks.curve25519 import scalarmult, scalarmult_base
import binascii

a = urandom(32)
a_pub = scalarmult_base(a)

b = urandom(32)
b_pub = scalarmult_base(b)

k_ab = scalarmult(a, b_pub)
k_ba = scalarmult(b, a_pub)

print "Bob public: ",binascii.hexlify(b_pub)
print "Alice public: ",binascii.hexlify(a_pub)
print "Bob shared: ",binascii.hexlify(k_ba)
print "Alice shared: ",binascii.hexlify(k_ab)

```

Do Bob and Alice end up with the same key?

How large are the random numbers that Bob and Alice generate?

Do you think that this program will be secure? How might Eve discover the shared secret?

Estimate the time it would take her to discover the key if she can try one billion keys per second:

How would you modify that program so that it was more secure?

## E Simple Key Distribution Centre (KDC)

Rather than using key exchange, we can setup a KDC, and where Bob and Alice can have long-term keys. These can be used to generate a session key for them to use. Enter the following Python program, and prove its operation:

```
import hashlib
import sys
import binascii
import Padding
import random

from Crypto.Cipher import AES
from Crypto import Random

msg="test"

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

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

def pad(s):
    extra = len(s) % 16
    if extra > 0:
        s = s + (' ' * (16 - extra))
    return s

rnd = random.randint(1,2**128)
keyA= hashlib.md5(str(rnd)).digest()
rnd = random.randint(1,2**128)
keyB= hashlib.md5(str(rnd)).digest()

print 'Long-term Key Alice=',binascii.hexlify(keyA)
print 'Long-term Key Bob=',binascii.hexlify(keyB)


rnd = random.randint(1,2**128)
keySession= hashlib.md5(str(rnd)).hexdigest()

ya = encrypt(keySession,keyA,AES.MODE_ECB)
yb = encrypt(keySession,keyB,AES.MODE_ECB)

print "Encrypted key sent to Alice:",binascii.hexlify(ya)
print "Encrypted key sent to Bob:",binascii.hexlify(yb)

decipherA = decrypt(ya,keyA,AES.MODE_ECB)
```

```
decipherB = decrypt(yb, keyB, AES.MODE_ECB)
print "Session key:", decipherA
print "Session key:", decipherB
```

 **Web link (Simple KDC):** <https://asecuritysite.com/encryption/kdc01>

The program above uses a shared 128-bit session key (generated by MD5). Now change the program so that you generate a 256-bit session key. What are the changes made:

## Notes

---

The code can be downloaded from:

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

If you need to update the code, go into the cyber\_data folder, and run:

```
git pull
```

To install a Python library use:

```
pip install libname
```

To install a Node.js package, use:

```
npm install libname
```