# Lab 5a: Mini-Project

**Objective:** In this lab we will build a basic infrastructure for integrating and testing cryptograph.

Open up your **Ubuntu instance** within vsoc.napier.ac.uk and conduct this lab.

Demo: <https://www.youtube.com/watch?v=1YC-nP3nCO4>

1. Open up the following page:

🕮 **Web link (Mini-project):** <https://asecuritysite.com/encryption/js10>

On this page, you will find RSA and ECC key pair generation. As this will run in the browser, we can assess how well a machine will cope with the key generation. On your VM (Ubuntu), on the computer desktop (such as Mac or Windows) and on your mobile phone, run the following tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **VM time** | **Desktop time** | **Mobile phone time** |
| RSA 1,024 |  |  |  |
| RSA 2,048 |  |  |  |
| ECC 128-bit |  |  |  |
| ECC 160-bit |  |  |  |
| ECC 256-bit |  |  |  |

What can you observe about the performance of the key pair generation?

Does the timing vary significantly for different browsers? Run the following browsers and note the time it takes to create the key pair:

IE:

Chrome:

Firefox:

Safari (if you have an Apple device):

If you are in a lab, share your results with others. What conclusions do you come to on the different devices and browsers for key pair generation?

2. Open up the following page:

🕮 **Web link (Mini-project):** <https://asecuritysite.com/encryption/js10>

We now want to build this page on your own virtual machine. The outline code is available here:

https://github.com/billbuchanan/esecurity/tree/master/z\_associated/projects/miniproject

The two files you are need are: crypto.html and cryptojs.js, along with the folder **scripts**.

Download these files from the following ZIP file and run the **crypto.html** file within your Web browser.

https://github.com/billbuchanan/esecurity/blob/master/z\_associated/projects/miniproject/cryptojs.zip

Does it run? Yes/No

3. Now you need to test the code. For the following test the hashing function of your code:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Word to hash** | **Result from your Web page (first two hex characters)** | **Test using Python [see code below](first two hex characters)** | **Prove with Openssl** |
| **MD5** | **“Hello”** |  |  |  |
| **SHA1** | **“Hello”** |  |  |  |
| **SHA256** | **“Hello”** |  |  |  |
| **SHA3** | **“Hello”** |  |  |  |
| **RIPEMD** | **“Hello”** |  |  |  |
| **PBKDF2 256-bit** | **“Hello”** |  |  |  |

If we test with Openssl:

echo -n Hello | openssl md5

echo -n Hello | openssl sha1

echo -n Hello | openssl sha256

echo -n Hello | openssl sha1 -ripemd160

The following is some sample code you can test your hashes against:

import hashlib;

string="password".encode()

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())

To test your PBKDF2 code, you will have to take the salt generated randomly from your Web page and copy it. For example:

Type: PBKDF2

Message: Hello

Salt: **0b72ad84e34c9fc218dc92bc13463fd3**

128-bit: 0e914d54afec72d31645c16be7da64f6

256-bit: 0e914d54afec72d31645c16be7da64f6d30d06271d0e76a2df77ae859ad2c562

512-bit: 0e914d54afec72d31645c16be7da64f6d30d06271d0e76a2df77ae859ad2c56246414ff7fa4a55382c5201bcd803c54bf340a5fd998f98a9580758f4a904dd48

The JavaScript integration has 1,000 iterations, so we can create a Python program which will convert this hex value for the salt into ASCII:

import hashlib;

import passlib.hash;

salt="**0b72ad84e34c9fc218dc92bc13463fd3**"

salt=salt.decode('hex')

print 'Salt is ',salt.encode('base64')

string="Hello"

print "PBKDF2 (SHA1):"+passlib.hash.pbkdf2\_sha1.encrypt(string, salt=salt,rounds=1000)

print "PBKDF2 (SHA256):"+passlib.hash.pbkdf2\_sha256.encrypt(string, salt=salt,rounds=1000)

When we run this example, we get:

PBKDF2 (SHA1):$pbkdf2$1000$C3KthONMn8IY3JK8E0Y/0w$sVnP8TwZ0pizjc0KrvmN/m31sTM

PBKDF2 (SHA256):$pbkdf2-sha256$1000$C3KthONMn8IY3JK8E0Y/0w$1c6YlCPSb4MdKTlqXGo/NrlpDQy0oivGTmtl2F3cyuk

We can see the salt value in Base64, and the hash value after it.

For RIPEMD160, can you implement your own checker? What is the code used:

By performing an on-line search, can you find an application where RIPEMD160 is used?

4. For the following test the MAC function of your code:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Word to hash** | **Password** | **Result from your Web page (first two hex characters)** | **Test using Python [see code below](first two hex characters)** |
| HMAC(MD5) | “Hello” | “qwerty” |  |  |
| HMAC(SHA1) | “Hello” | “qwerty” |  |  |
| HMAC(SHA256) | “Hello” | “qwerty” |  |  |

We can test with Openssl using:

echo -n Hello | openssl md5 -hmac qwerty

echo -n Hello | openssl sha1 -hmac qwerty

echo -n Hello | openssl sha256 -hmac qwerty

You can also use the format of:

echo -n "Hello" | openssl dgst -sha1 -hmac "qwerty"

Can you replicate this with Node?

A hint is given in the Appendix.

5. Now we will test for symmetric key encryption. For AES CBC a sample run is:

Type: AES (CBC)

Message: Hello

Password: qwerty

Salt: 241fa86763b85341

IV: 6be952ebc17eed10411eaa9892f19124

Key: 33a5820536f9eeb709d88af3b40fdbb100c04327c71b5accf48424c8eb40c3f9

Encrypted: U2FsdGVkX18kH6hnY7hTQZAGxV2faF01w6uhO+X6+9Q=

Decrypted: Hello

Now check with OpenSSL (remember to change to the value of the salt that you have generated):

echo -n Hello | openssl enc -aes-256-cbc -pass pass:"qwerty" -e -base64 -S **241fa86763b85341**

U2FsdGVkX18kH6hnY7hTQZAGxV2faF01w6uhO+X6+9Q=

What is “U2FsdGVkX1” (use a Base64 decoder to determine)?

The format of the encrypted value is: 'Salted\_\_' + salt + ciphertext

By using a Base64 to Hex convertor, convert the output to hex, can you pick-off the hex value of the salt (a convertor is a https://asecuritysite.com/coding/ascii)?

[Hint: 8 bytes of the signature, then 8 bytes of salt]

Now save the cipher to a file (enc.txt) and then decrypt with (remember to change to the value of the salt that you have generated):

openssl enc -aes-256-cbc -pass pass:"qwerty" -d -base64 -S **241fa86763b85341** -in enc.txt -out out.txt

What is the contents of the “out.txt” file?

The following Python program produces the same output as OpenSSL. By using the values you have for plaintext, key, and salt, prove that the output is the same as the ciphertext produced by your JavaScript program:

from Crypto.Cipher import AES

import hashlib

import sys

import binascii

import base64

import Padding

plaintext='Hello'

key='qwerty'

salt='241fa86763b85341'

def get\_key\_and\_iv(password, salt, klen=32, ilen=16, msgdgst='md5'):

mdf = getattr(\_\_import\_\_('hashlib', fromlist=[msgdgst]), msgdgst)

password = password.encode('ascii', 'ignore') # convert to ASCII

try:

maxlen = klen + ilen

keyiv = mdf(password + salt).digest()

tmp = [keyiv]

while len(tmp) < maxlen:

tmp.append( mdf(tmp[-1] + password + salt).digest() )

keyiv += tmp[-1] # append the last byte

key = keyiv[:klen]

iv = keyiv[klen:klen+ilen]

return key, iv

except UnicodeDecodeError:

return None, None

def encrypt(plaintext,key, mode,salt):

key,iv=get\_key\_and\_iv(key,salt.decode('hex'))

encobj = AES.new(key,mode,iv)

return(encobj.encrypt(plaintext))

def decrypt(ciphertext,key, mode,salt):

key,iv=get\_key\_and\_iv(key,salt.decode('hex'))

encobj = AES.new(key,mode,iv)

return(encobj.decrypt(ciphertext))

plaintext = Padding.appendPadding(plaintext,mode='CMS')

ciphertext = encrypt(plaintext,key,AES.MODE\_CBC,salt)

ctext = b'Salted\_\_' + salt.decode('hex') + ciphertext

print "Cipher (ECB): "+base64.b64encode(ctext)

plaintext = decrypt(ciphertext,key,AES.MODE\_CBC,salt)

plaintext = Padding.removePadding(plaintext,mode='CMS')

print " decrypt: "+plaintext

A sample run is:

$ python aes\_openssl.py

Cipher (ECB): U2FsdGVkX18kH6hnY7hTQZAGxV2faF01w6uhO+X6+9Q=

decrypt: Hello

Outline the method used to generate the iv and key values?

You can also check against this link:

🕮 **Web link (AES and Python):** https://asecuritysite.com/encryption/aes\_python

Now try DES, and check with:

echo -n Hello | openssl enc -des  -pass pass:"qwerty" -e -base64 -S b99d7b9a5fc533d2  
U2FsdGVkX1+5nXuaX8Uz0sy7jQgKtewQ

Is the cipher correctly generated?

6. The following page has ECC and RSA key generation. By right-clicking on the page, can you integrate the ECC and RSA code into your code?

🕮 **Web link (Mini-project):** <https://asecuritysite.com/encryption/js10>

7. With node.js we can do the same operations as the JavaScript implementations, but run it from a command prompt (Note: you may have to use **npm install crypto-js**):

// Node.js example Run with:

// node crypto.js message password

message ="Hello"

password="qwerty"

var SHA256 = require("crypto-js/sha256");

var MD5 = require("crypto-js/md5");

var SHA3 = require("crypto-js/sha3");

var SHA1 = require("crypto-js/sha1");

var SHA224 = require("crypto-js/sha224");

var SHA512 = require("crypto-js/sha512");

var SHA384 = require("crypto-js/sha384");

var RIP = require("crypto-js/ripemd160");

var AES = require("crypto-js/aes");

var CryptoJS = require("crypto-js");

var args = process.argv;

if (args.length>2) message=args[2];

if (args.length>3) password=args[3];

console.log("Message: ",message);

console.log("Password: ",password);

console.log("\n--- Hashes");

console.log("MD5: ",MD5(message).toString());

console.log("SHA-256: ",SHA256(message).toString());

console.log("SHA-1: ",SHA1(message).toString());

console.log("SHA-224: ",SHA224(message).toString());

console.log("SHA-512: ",SHA512(message).toString());

console.log("SHA-384: ",SHA384(message).toString());

console.log("ripemd160: ",RIP(message).toString());

console.log("\n--- AES");

var ciphertext = AES.encrypt(message, password);

var ciphertext = CryptoJS.AES.encrypt(message, password,mode=CryptoJS.mode.ECB);

var bytes = CryptoJS.AES.decrypt(ciphertext.toString(), password,mode=CryptoJS.mode.ECB);

var plaintext = bytes.toString(CryptoJS.enc.Utf8);

console.log("Cipher: ",ciphertext.toString());

console.log("Plaintext: ",plaintext);

console.log("\n--- HMAC-SHA1");

console.log("HMAC: ",CryptoJS.HmacSHA1(message, password).toString());

A sample run is:

$ node cryptojs.js Hello qwerty

Message: Hello

Password: qwerty

--- Hashes

MD5: 8b1a9953c4611296a827abf8c47804d7

SHA-256: 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969

SHA-1: f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0

SHA-224: 4149da18aa8bfc2b1e382c6c26556d01a92c261b6436dad5e3be3fcc

SHA-512: 3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf777d3d795a7a00a16bf7e7f3fb9561ee9baae480da9fe7a18769e71886b03f315

SHA-384: 3519fe5ad2c596efe3e276a6f351b8fc0b03db861782490d45f7598ebd0ab5fd5520ed102f38c4a5ec834e98668035fc

ripemd160: d44426aca8ae0a69cdbc4021c64fa5ad68ca32fe

--- AES

Hello qwerty

Cipher: U2FsdGVkX1+k/F8uNPiUeRzIeTajlxidwGfpRLPJyEA=

Salt: a4fc5f2e34f89479

IV: eb81d8b7e67223cf2a1a67aef93c1489

Plaintext: Hello

--- HMAC-SHA1

HMAC: 8c7cd4cb162bc91e4ee4573aba50ca00474e7c5d

7a. Now run the code and check the answers for the hashing methods from this page:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Word to hash** | **Result from your Web page (first two hex characters)** | **Test using node.js** |
| **MD5** | **“Hello”** |  |  |
| **SHA1** | **“Hello”** |  |  |
| **SHA256** | **“Hello”** |  |  |
| **SHA3** | **“Hello”** |  |  |
| **RIPEMD160** | **“Hello”** |  |  |

7b. The program implements AES, now implement two other modes: CBC and OFB, and make sure the program works.

7c. We can try some ciphertext by adding the Base64 cipher to the decrypt method:

var bytes = CryptoJS.AES.decrypt( "U2FsdGVkX1+k/F8uNPiUeRzIeTajlxidwGfpRLPJyEA=" , password,mode=CryptoJS.mode.ECB);

Using the technical (and with ECB), can you decrypt the following (and which use the passphrase of “qwerty”:

U2FsdGVkX187BmuVYneWcRn5sgDat6uHqmyKEa31Vys=

U2FsdGVkX19UMSQ9ZqKUfyc2ffU/fujbo9lrQLx54Eo=

U2FsdGVkX1+c0r64T4TsD9Bx1e0Okb3Q+Gflb6AknTA=

What are the words?

Why do we not have to provide the salt to the decryption method?

7d. The program implements AES, can you now implement RC4 and Rabbit, and prove that they can encrypt and decrypt.

7e. The program implements HMAC-SHA1. Now implement HMAC-SHA256, HMAC-SHA3 and HMAC-RIPEMD160, can verify the answers against the test Web page.

8. If you were developing a front-end application for a bank. How would you support the sending back encrypted data? Using the code that you have developed, could you generate an RSA key pair and use it to encrypt credit card details that the user enters?

## Reflective questions

Why didn’t we have to provide an additional salt value when we decrypted the ciphertext in Question 7b?

## Appendix

Some Hmac code:

var crypto = require('crypto');

var key = 'qwerty';

var message = 'Hello';

var hash = crypto.createHmac('md5', key).update(message);

console.log(hash.digest('hex'));

console.log(hash.digest('base64'));

A sample run:

$ node h.js

7f43007a026d9696566dc8c7bb2172e4