# 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.

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 you VM, on the computer desktop 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 |  |  |  |
| ECC 512-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;

import passlib.hash;

string="password"

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

Python … TBC

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:

echo -n Hello | openssl enc -aes-256-cbc -pass pass:"qwerty" -e -base64 -S 241fa86763b85341 -iv 6be952ebc17eed10411eaa9892f19124

U2FsdGVkX18kH6hnY7hTQZAGxV2faF01w6uhO+X6+9Q=

What is “U2FsdGVkX1”?

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>