**Homework 2**

**2.**

**A. Can the Java SHA1PRNG be used securely for cryptographic operations such as generate private/public key pairs?**

Yes, the Java SHA1PRNG can be used securely for cryptographic operations as long as it is used appropriately.

**B. What pitfalls do programmers have be aware of when using pseudo-random number generators for cryptographic operations?**

The output of pseudorandom number generator might become predictable if seeded improperly. Instead of implementing a PRNG itself, the class java.security.SecureRandom uses PRNG implementations in other classes to generate random numbers.

**C. Why should a programmer be concerned about using SecureRandom.getInstanceStrong() in certain types of applications?**

Different default behaviors will appear on the usage of old mechanism and SecureRandom.getInstanceStrong().Due to the importance of availability, a programmer should avoid using it when executing on Solaris/Linux/MacOS server-side.

**4.**

**Fill in the GenerateScroogeKeyPair.java main method with code that does the following:**

**A. Generates a ECDSA key pair for Scrooge.**

**B. Stores the private key in an encrypted format on disk.**

**C. Store the public key in a separate, unencrypted file.**

**Run the class to generate the key pair for Scrooge. Name the key files as scrooge\_sk.pem and scrooge\_pk.pem so that it is clear who they belong to. You will use this key pair for the remaining parts of this lab.**

**In your submission, include your code and the contents of the file containing Scrooge's public key. Do not submit your secret key. Remember never to give out your secret key and to always encrypt the secret key file when storing it on disk.**

Code:

import org.bouncycastle.jce.provider.BouncyCastleProvider;

import org.bouncycastle.openssl.\*;

import org.bouncycastle.openssl.jcajce.\*;

import org.bouncycastle.operator.OperatorCreationException;

import org.bouncycastle.pkcs.PKCSException;

import javax.xml.bind.DatatypeConverter;

import java.io.\*;

import java.nio.charset.StandardCharsets;

import java.security.\*;

import java.security.spec.ECGenParameterSpec;

import java.security.spec.InvalidKeySpecException;

import static org.bouncycastle.cms.RecipientId.password;

public class GenerateScroogeKeyPair {

private static final String KEY\_ALGORITHM = "ECDSA";

private static final String PROVIDER = "BC";

private static final String CURVE\_NAME = "secp256k1";

private static ECGenParameterSpec ecGenSpec;

private static KeyPairGenerator keyGen\_;

private static SecureRandom random;

private static String password = "12345678";

public static void main(String[] args) throws Exception {

Security.addProvider(new BouncyCastleProvider());

random = SecureRandom.getInstanceStrong();

ecGenSpec = new ECGenParameterSpec(CURVE\_NAME);

keyGen\_ = KeyPairGenerator.getInstance(KEY\_ALGORITHM, PROVIDER);

// Generates a ECDSA key pair.

keyGen\_.initialize(ecGenSpec, random);

KeyPair kp = keyGen\_.generateKeyPair();

PublicKey publicKey = kp.getPublic(); //"pk" == "public key"

PrivateKey secretKey = kp.getPrivate(); //"sk" == "secret key" == "private key"

System.out.println(publicKey);

System.out.println(secretKey);

// Store the public key in a separate, unencrypted file as scrooge\_sk.pem

String pkFilename = "scrooge\_pk.pem";

StringWriter sw = new StringWriter();

JcaPEMWriter wr = new JcaPEMWriter(sw);

wr.writeObject(kp.getPublic());

wr.close();

Writer fw = new FileWriter(pkFilename);

fw.write(sw.toString());

fw.close();

System.out.println("Public Key:\n" + sw.toString());

// Stores the private key in an encrypted format on disk as scrooge\_pk.pem

String skFilename = "scrooge\_sk.pem";

JcaPEMWriter privWriter = new JcaPEMWriter(new FileWriter(skFilename));

PEMEncryptor penc = (new JcePEMEncryptorBuilder("AES-256-CFB"))

.build(password.toCharArray());

privWriter.writeObject(secretKey, penc);

privWriter.close();

}

}

The contents of the file containing Scrooge's public key:

-----BEGIN PUBLIC KEY-----

MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE3V8CR/X9r0Lbwxe1cSL1vJnASptgOXBA

TJTKttdmU2EPBUu67fJE7lwiN8AGW8nZkp4flZYG2/JEQha+v7OpOA==

-----END PUBLIC KEY-----

**5. Fill in the GenerateDigitalSignature main method with code that does the following:**

**A. Reads Scrooge's key pair from disk**

**B. Generate Scrooge's digital signature for the message "Pay 3 bitcoins to Alice". Do not include the quotations in the message. Capitalization matters.**

**In your submission, include your code and the digital signature in hexadecimal.**

Code:

import org.bouncycastle.jce.provider.BouncyCastleProvider;

import org.bouncycastle.openssl.\*;

import org.bouncycastle.openssl.jcajce.\*;

import org.bouncycastle.operator.OperatorCreationException;

import org.bouncycastle.pkcs.PKCSException;

import javax.xml.bind.DatatypeConverter;

import java.io.\*;

import java.nio.charset.StandardCharsets;

import java.security.\*;

import java.security.spec.ECGenParameterSpec;

import java.security.spec.InvalidKeySpecException;

public class GenerateDigitalSignature {

private static final String SIGNATURE\_ALGORITHM = "SHA256withECDSA";

private static SecureRandom random;

private static String password = "12345678";

public static void main(String[] args) throws Throwable {

random = SecureRandom.getInstanceStrong();

// Read Scrooge's key pair from disk

KeyPair kp = loadKeyFromEncrypted("scrooge\_sk.pem", password);

PrivateKey recoveredKey = kp.getPrivate();

PublicKey publicKey = kp.getPublic();

// Generate Scrooge's digital signature for the message "Pay 3 bitcoins to Alice"

Signature signature = Signature.getInstance(SIGNATURE\_ALGORITHM);

signature.initSign(recoveredKey, random);

String messageStr1 = "Pay 3 bitcoins to Alice";

byte[] message1 = messageStr1.getBytes(StandardCharsets.UTF\_8);

signature.update(message1);

byte[] sigBytes1 = signature.sign();

System.out.println("Signature: msg=" + messageStr1 + " sig.len=" + sigBytes1.length + " sig=" + DatatypeConverter.printHexBinary(sigBytes1));

}

public static KeyPair loadKeyFromEncrypted(String filename, String password) throws IOException, NoSuchAlgorithmException, InvalidKeySpecException, PKCSException, OperatorCreationException {

File secretKeyFile = new File(filename); // private key file in PEM format

PEMParser pemParser = new PEMParser(new FileReader(secretKeyFile));

Object object = pemParser.readObject();

PEMDecryptorProvider decProv = new JcePEMDecryptorProviderBuilder().build(password.toCharArray());

JcaPEMKeyConverter converter = new JcaPEMKeyConverter();

KeyPair kp = converter.getKeyPair(((PEMEncryptedKeyPair) object).decryptKeyPair(decProv));

return kp;

}

}

The digital signature in hexadecimal:

30450220714BFEF9B8E5B17F9660FE5BF60FC288A1BECB2B36E50DAC0872C30B13C76A9A0221009B9D3A340D5B7D3AC9365E744676B39E02AF008DC6929804F9BF1DC90425BCB2