INFO 7500 Cryptocurrency/Smart Contract — Homework2

Mibin Zhu / 001424937

Question2

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

Answer:

Yes, Java SHA1PRNG can be used securely for cryptographic operations when we use it appropriately. Take the hint in the given link, if we call the setSeed() function before getting output results in predictable outputs on Windows, we can securely generate private/public key pairs.

If we also look into the definition of the SHA1PRNG, we can find that this algorithm uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used. SHA1PRNG uses hash functions, counters, and seeds. The algorithm is relatively simple and is generally considered safe.

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

Answer:

Note that according to Sun's documentation, the returned SecureRandom instance is not seeded by any of these calls. If after one of these calls, nextBytes(byte[]) is called, then the PRNG is seeded using a secure mechanism provided by the underlying operating system. If setSeed(long) or setSeed(byte[]) is called before a call to nextBytes(byte[]), then the internal seeding mechanism is bypassed, and only the provided seed is used to generate random numbers.

By bypassing the internal secure seeding mechanism of the SHA1PRNG, programmer may compromise the security of your PRNG output. If programmer seed it with anything that an attacker can potentially predict, then using SecureRandom may not provide the level of security that you need.

Always specify the exact PRNG and provider that programmer wish to use. If programmer just use the default PRNG, programmer may end up with different PRNGs on different installations of their application that may need to be called differently in order to work properly. Using the following code to get a PRNG instance is appropriate : SecureRandom sr = SecureRandom.getInstance("SHA1PRNG", "SUN");

When using the SHA1PRNG, always call java.security.SecureRandom.nextBytes(byte[]) immediately after creating a new instance of the PRNG. This will force the PRNG to seed itself securely. If for testing purposes,

programmer need predictable output, ignoring this rule and seeding the PRNG with hard-coded/predictable values may be appropriate.

Use at least JRE 1.4.1 on Windows and at least JRE 1.4.2 on Solaris and Linux. Earlier versions do not seed the SHA1PRNG securely.

Periodically reseed your PRNG as observing a large amount of PRNG output generated using one seed may allow the attacker to determine the seed and thus predict all future outputs.

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

Answer:

Because the getInstanceStrong() method uses the strongAlgorithms property in the java.security file to select a SecureRandom implementation. This method returns an instance of the strongest SecureRandom implementation available on each platform. And for different platform, the implementation of the getInstanceStrong() method is different.

For using SecureRandom.getInstanceStrong() function, the following defaults are used instead. When the operation system is Windows, the default SecureRandom implementation is Windows-PRNG which simply outputs bytes from the Windows CryptGenRandom() API.

When the operation system is Solaris/Linux/macOS, the default SecureRandom implementation is NativePRNGBlocking which may bring the blocking problems.

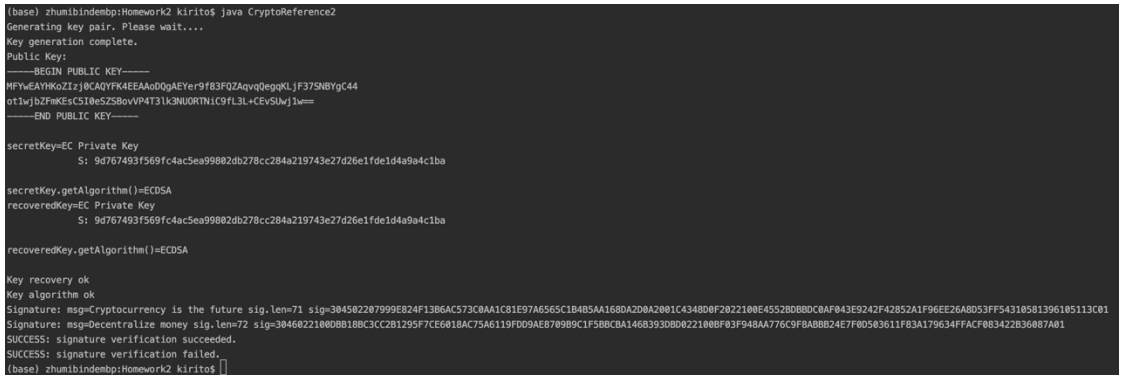
Because of this default behavior, programmer should avoid using SecureRandom.getInstanceStrong() in any server-side code running on Solaris/Linux/macOS where availability is important.

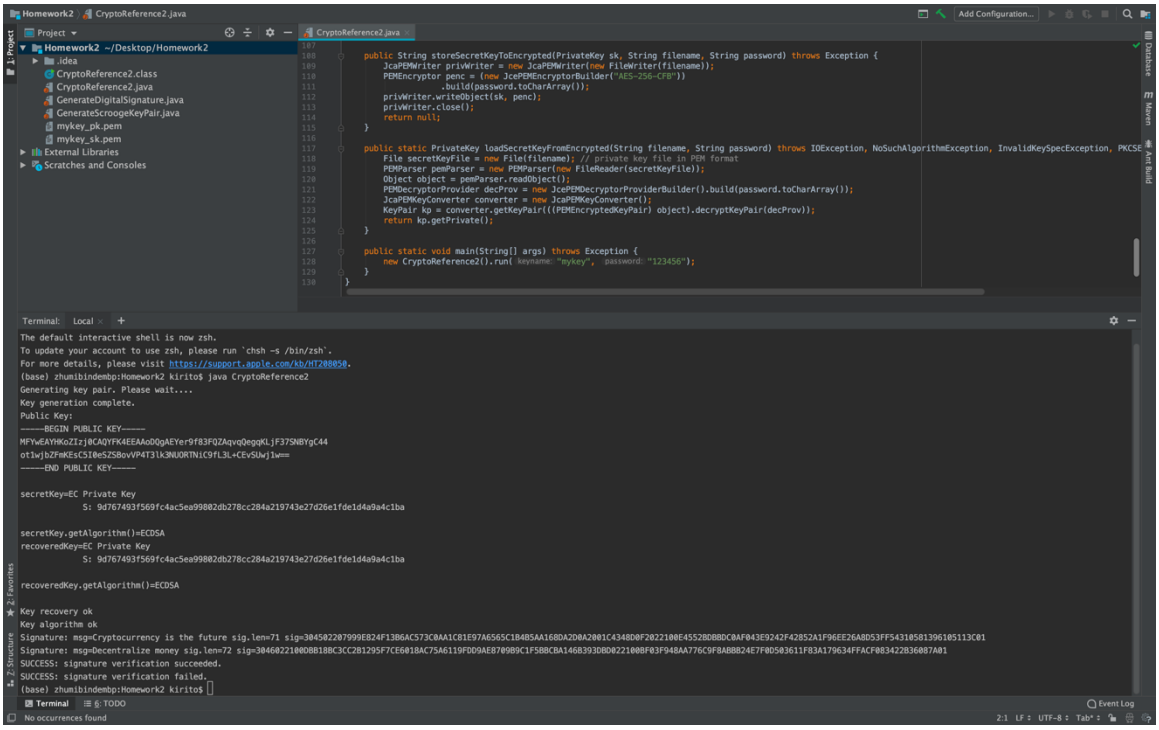
Question3

Try running the CryptoReference2 program on your computer and confirm that it completes successful without throwing exceptions. This program generates ECDSA keys.

Answer:

The following picture is how I run the code after putting the jar file into the right location.





Question4

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

A. Generates an ECDSA key pair for Scrooge.

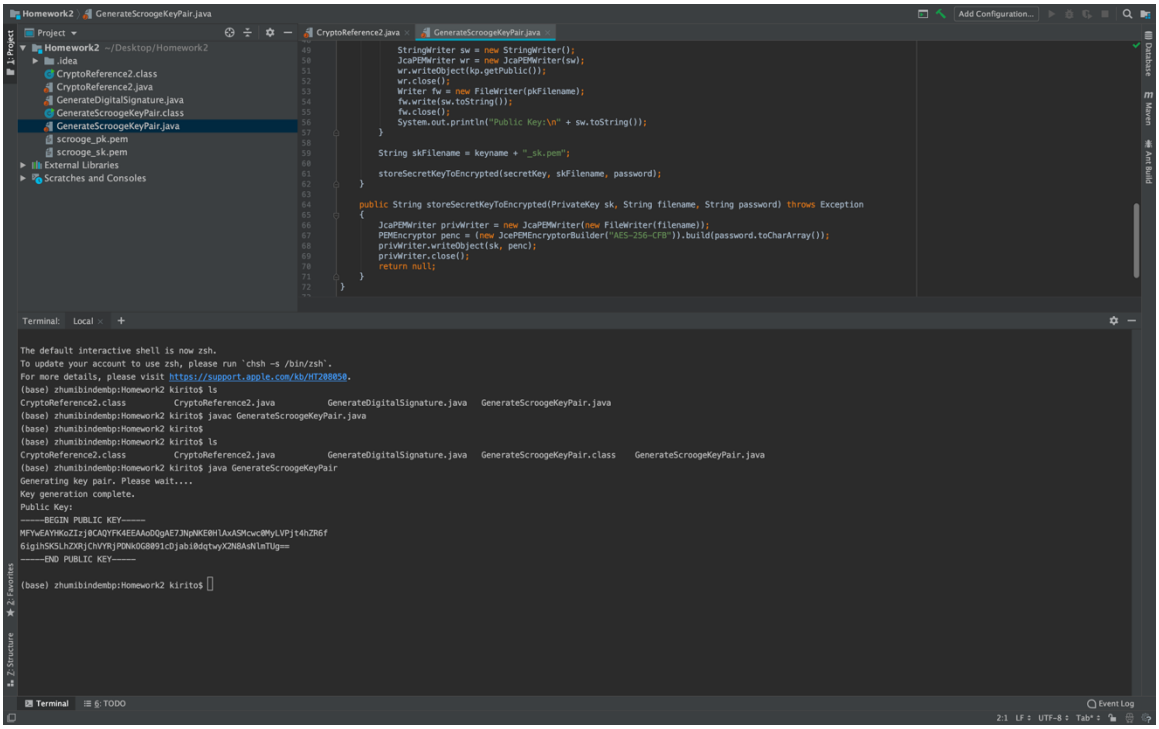
B. Stores the private key in an encrypted format on disk.

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

Answer:

The run function can be separated to two parts, the first part is how to generate the key pair, and the other part is for making the signature. The public key can be shown to anyone so it does not need to be encrypted, but the secrete key should be private, and I use the method storeSecretKeyToEncrypted to realize this function. Then the private key is in an encrypted format on disk, the code is upload separately to the BB. The public key is MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEN9z9JExJ6AZnFcfNCMD7+mTJV1reJeFZZ3SP/NZuC6Tn2G3VPsJSQSQilV9sR2gAuEcb93uaKrbMCxYVwZmbYg==.

```
(base) zhumibindembp:Homework2 kirito$ ls
CryptoReference2.class          CryptoReference2.java           GenerateDigitalSignature.java   GenerateScroogeKeyPair.java
(base) zhumibindembp:Homework2 kirito$ javac GenerateScroogeKeyPair.java
(base) zhumibindembp:Homework2 kirito$
(base) zhumibindembp:Homework2 kirito$ ls
CryptoReference2.class          CryptoReference2.java           GenerateDigitalSignature.java   GenerateScroogeKeyPair.class    GenerateScroogeKeyPair.java
(base) zhumibindembp:Homework2 kirito$ java GenerateScroogeKeyPair
Generating key pair. Please wait....
Key generation complete.
Public Key:
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE7JNpNKE0HlAxASMcwc0MyLVPjt4hZR6f
6igihSK5LhZXRjChVYRjPDNkOG8091cDjabi0dqtwyX2N8AsNlmTUg==
-----END PUBLIC KEY-----
```



Question5

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

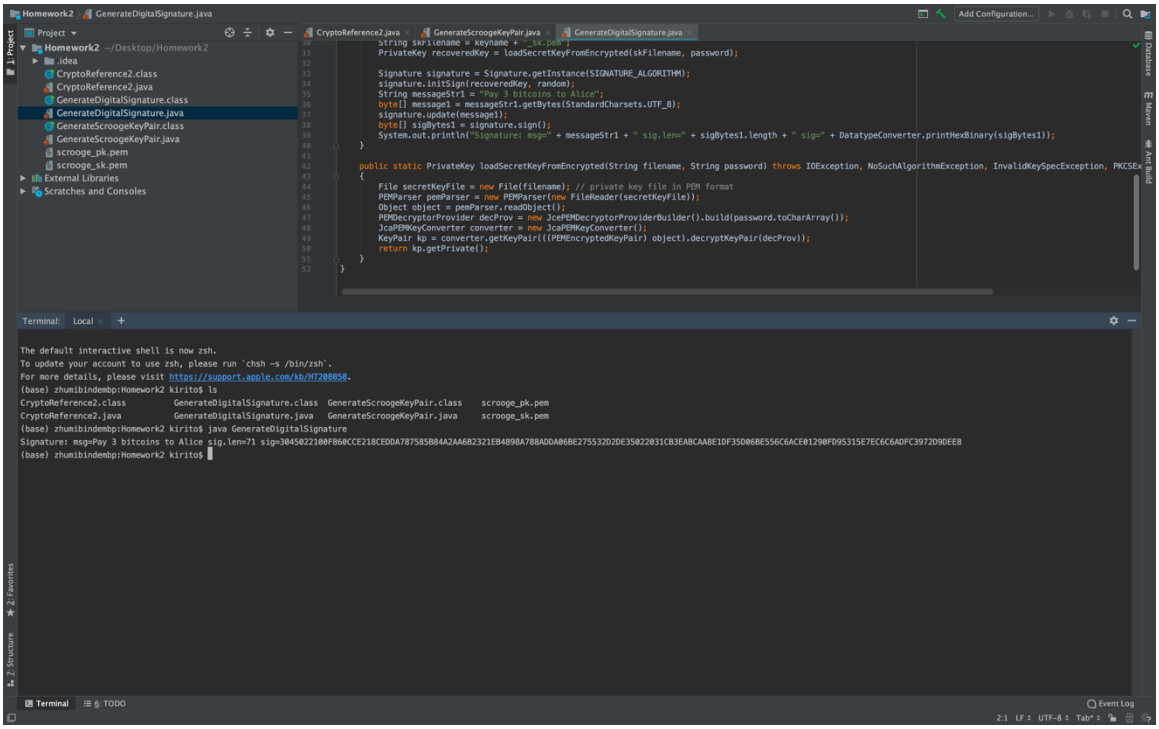
A. Reads Scrooge's key pair from disk

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

Answer:

304402202CD0710FBF3E97FF463FB8D3C2A9B6D1812B23D9A362B5DB29CD64FD918ECDC402204
35F23458C2FFF85FB263F877BB97062352AA8D7825C7D3E65B9257C804BC983 is the hex formation of the digital signature, the code is upload separately to the BB.

```
(base) zhumibindembp:Homework2 kirito$ ls
CryptoReference2.class          GenerateDigitalSignature.class  GenerateScroogeKeyPair.class    scrooge_pk.pem
CryptoReference2.java           GenerateDigitalSignature.java   GenerateScroogeKeyPair.java     scrooge_sk.pem
(base) zhumibindembp:Homework2 kirito$ java GenerateDigitalSignature
Signature: msg=Pay 3 bitcoins to Alice sig.len=71 sig=3045022100F860CCE218CEDDA787585B84A2AA6B2321EB4898A788ADDA06BE275532D2DE35022031CB3EABCAA8E1DF35D06BE556C6ACE01290FD95315E7EC6C6ADFC3972D9DEEB
(base) zhumibindembp:Homework2 kirito$
```


Homework2 / GenerateDigitalSignature.java
Add Configuration...
Project
Homework2 ~/Desktop/Homework2
.idea
CryptoReference2.class
CryptoReference2.java
GenerateDigitalSignature.class
GenerateDigitalSignature.java
GenerateScroogeKeyPair.class
GenerateScroogeKeyPair.java
scrooge_pk.pem
scrooge_sk.pem
External Libraries
Scratches and Consoles
CryptoReference2.java
GenerateScroogeKeyPair.java
GenerateDigitalSignature.java
String skFilename = keyname + "_sk.pem";
PrivateKey recoveredKey = loadSecretKeyFromEncrypted(skFilename, password);
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 PrivateKey loadSecretKeyFromEncrypted(String filename, String password) throws IOException, NoSuchAlgorithmException, InvalidKeySpecException, PKCSEx
{
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.getPrivate();
}
}
Terminal: Local
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) zhumibindembp:Homework2 kirito$ ls
CryptoReference2.class GenerateDigitalSignature.class GenerateScroogeKeyPair.class scrooge_pk.pem
CryptoReference2.java GenerateDigitalSignature.java GenerateScroogeKeyPair.java scrooge_sk.pem
(base) zhumibindembp:Homework2 kirito$ java GenerateDigitalSignature
Signature: msg=Pay 3 bitcoins to Alice sig.len=71 sig=3045022100F860CCE218CEDDA78758SB84A2AA6B2321EB4898A788ADDA06BE275532D2DE35022031CB3EABCAA8E1DF35D06BE556C6ACE01290FD95315E7EC6C6ADFC3972D9DEE8
(base) zhumibindembp:Homework2 kirito$
Terminal
6: TODO
Event Log
2:1 LF UTF-8 Tab*