Homework 2

2. Answer the following questions:

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

Yes. SHA1PRNG can provide sufficiently random values for cryptographic operations.

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

- Always specify the exact PRNG and provider that you wish to use. If you just use the default PRNG, you may end up with different PRNGs on different installations of your application that may need to be called differently in order to work properly.
- 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, you 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?

SecureRandom.getInstanceStrong() method uses the securerandom.strongAlgorithms property in the java.security file to select a SecureRandom implementation. It returns a SecureRandom object that was selected by using the algorithms/providers specified in the securerandom.strong Algorithms Security property.

However, this method additionally throws a NoSuchAlgorithmException "if no algorithm is available", which should guarantee that you never get a weak implementation back. Because of this behavior, programmers should avoid using this method in any server-side code running on Solaris/Linux/MacOS where availability is important.

3. No throwing exceptions:

```
Generating key pair. Please wait....
Key generation complete.
Public Key:
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEKaWxg/8ija7Ir5JrIXzc8yNByRZWrjhq
0WjlNzLhWyON2tm5T/OpAS1dIU3iCKPVeAkYBEMpVzEeYjSV9bAWFw==
-----END PUBLIC KEY-----

secretKey=EC Private Key
            S: 5f6c600f4ecb4234aac1fcfde02ff448c7052641575872c9a41b4e9abffdc98b

secretKey.getAlgorithm()=ECDSA
recoveredKey=EC Private Key
            S: 5f6c600f4ecb4234aac1fcfde02ff448c7052641575872c9a41b4e9abffdc98b

recoveredKey.getAlgorithm()=ECDSA

Key recovery ok
Key algorithm ok
Signature: msg=Cryptocurrency is the future sig.len=71 sig=304502206C08505C0430A54E2069C66132BFBC45169543F35412FC29228B4380E21BECB5022100FBF7596C1C33B4DC703A026
Signature: msg=Decentralize money sig.len=72 sig=3046022100E343CD66AA0BA3E23C5C9C095166E7656F1D436997AA803B1D82C8A106510C1C02210091916DEA3C72DAA59971C2AD9C9B5A2
SUCCESS: signature verification succeeded.
SUCCESS: signature verification failed.
```

4. Running result: Scrooge's public key

```
Generating key pair. Please wait....
Key generation complete.
Public Key:
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEW2Y0QXTHdE1BpD6XbLvOSOcImvM8sWXf
LnL2lw3e5+6QiztvuwbDI4/rtbjiNzQFcYt7q/4XnnfDOhYccOUQ+A==
-----END PUBLIC KEY-----
```

5. Running result: the digital signature in hexadecimal

```
Signature
msg=Pay 3 bitcoins to Alice
sig.len=72
sig=3046022100A8435774DC39396EE21F452DD49E423CA253BD164F3D204E2235465F06149D9A022100B2E079004DE07E5B74D96382FF93893FB595270D06127D056E3B97C2E7165059
```