***Secure Programming*** *Course Proposal: Sample Exam Question*

1. (a) CWE/SANS declared the top 5 most dangerous software errors in 2011 as:

   1. CWE-89 SQL Injection
   2. CWE-78 OS Command Injection
   3. CWE-120 Classic Buffer Overflow
   4. CWE-79 Cross-site Scripting
   5. CWE-306 Missing Authentication for Critical Function

   For each error type, give a brief explanation of how you would *program defensively* to avoid it. [*5 marks* ]

   (b) Imagine that you are working on a Java based web application together with a colleague. You are writing the front end code and your colleague is providing the back end for managing the database which stores the application data and the user registration information. It was decided to handle the user information directly in Java with a persistence layer for storing serialised objects to disk.

   Your colleague has proposed a class `UserInfo` for managing the user information and logins. The public methods form the API for using the class and the interface for your code. A first draft of the source code is shown on the next page.

   Conduct a code review of the `UserInfo` class, making ten distinct security-relevant observations. You may include the following kinds of observation:

   - security assumptions made by the code and how far these are justified;
   - security provisions the code successfully provides and how;
   - programming or design flaws that have security implications and how they might be repaired;
   - missing or superfluous functionality; additional recommendations.

   Please use line numbers in your answers. Each observation that is relevant to security and explained with specific reference to the code and the given scenario will attract 2 marks. If you give more than ten observations, only the first ten will be considered. [*20 marks* ]

*QUESTION CONTINUES ON NEXT PAGE*

```
1  import java.security.MessageDigest;
2  import java.util.Random;
3  import java.util.Arrays;
4  import java.io.Serializable;
5
6  public final class UserInfo implements Serializable {
7      /* per-user info */
8      private String userName;
9      private String[] userAddress;
10     private boolean authenticated;
11     private final byte[] passwordHash = new byte[16];
12
13     /* shared utilities */
14     private static MessageDigest sha1 = MessageDigest.getInstance("SHA1");
15     private static Random rand = new Random(System.currentTimeMillis());
16
17     /* constructor, assumes user name is well-formed and new */
18     public UserInfo(String name, String[] address, String password) {
19         assert password != null && password.length() >= 8;
20         userName = name;  userAddress = address;
21         System.arraycopy(getHash(password),0,passwordHash,
22                          0,passwordHash.length);
23     }
24     /* accessors */
25     public String   getUserName() { return userName; }
26     public String[] getUserAddress() { return userAddress; }
27     public boolean  isAuthenticated() { return authenticated; }
28
29     /* login */
30     public void tryLogin(String inputPassword) {
31         if (Arrays.equals(getHash(inputPassword),passwordHash)) {
32             authenticated = true;
33         }
34     }
35     /* support lost password action */
36     public String resetPassword() {
37         String newPassword = null;
38         if (!authenticated) {
39             newPassword = Integer.toString(rand.nextInt(Integer.MAX_VALUE));
40             byte[] hash = getHash(newPassword);
41             System.arraycopy(hash,0,passwordHash,0,hash.length);
42         }
43         return newPassword;
44     }
45     /* utility */
46     private byte[] getHash(String msg) {
47         sha1.reset();
48         sha1.update(msg.getBytes());
49         return sha1.digest();
50     }
51 }
```

1. (a) Briefly:

   i. SQL injection, OS Command Injection and Cross-site Scripting are all avoided by sanitizing inputs, e.g. by quoting and escaping special characters; this should be done anywhere that a prepared query is used, a command for the Operating System is issued, or some HTML output is produced that is partially based on user/external content. [3 marks, ideally some specifics in each case would be given]

   ii. Classic Buffer Overflow is avoided simply by checking that whenever input is copied between one location and another, the destination location is big enough to contain the source data being copied. [ 1 mark ]

   iii. Missing Authentication: the programmer should be aware of operations which are assumed to be allowed only for authorized users, and this should be enforced in the code path by making sure that authentication checks are made, or have been made *recently*. [ 1 mark ]

   (b) The given code clearly shows some awareness of security and an attempt to provide some guarantees. A good answer to this question should pick up points such as those following.

   - **Assumptions**:
     - the code uses language features of Java and assumes these provide intended encapsulation, i.e., `private` and `final` in lines 6, 8-11, 14-15. This assumption is probably justified although with the JVM's dynamic loader it is possible to load additional classes at runtime and some of the Java-level encapsulation mechanisms are not enforced at the bytecode level.
     - the code uses a SHA1 message digest which is assumed to properly implement SHA1, lines 41, 45-47. The precise implementation of SHA1 that is chosen can be platform dependent, and should be checked to be a standard trusted version in the deployment environment.
     - the code uses random numbers from `java.util.Random`, which is used to generate random numbers when performing password resets. If an attacker discovers a password, this should give no help in finding out the next password if a user resets their password, so this ought to be a cryptographically secure random number generator. Unfortunately, that guarantee isn't provided for `java.util.Random`.
   - **Provisions**:
     - There is an attempt to insist on non-empty passwords at least 8 characters long, although this is flawed: the check on line 19 is inside an `assert` statement, but these are only executed when assertion checking is turned on the Java runtime. This check (or an

improved version to eliminate easily guessed passwords) should not be optional.

  - The design does not store passwords in plaintext, which is a good defence against mass password theft. Using hashes would not be necessary if we only relied on the abstraction barriers of the language (and assume attacks on memory are impossible during execution), but it is useful because objects are persisted on disk, where they may be vulnerable to inspection and tampering, depending on the serialisation mechanism.

- **Flaws**:
  - Although password hashes are stored there is no salting, so an attacker who accesses the persisted object store can perform efficient offline dictionary attacks.
  - Although passwords are obscured on disk, it is likely that user names and user addresses are not encrypted when in storage, so they may be read or corrupted.
  - The use of reference variables without copying violates the abstraction boundary. This occurs in two places: on line 11, the user address reference is copied from the input, and on line 17 the user address reference is returned to client code. The callee could therefore subsequently modify the internal state of the `UserInfo`.
  - The use of `Random` is flawed by using a seed based on the current time which is an anti-recommendation for a source of randomness, because it may be influenced or predicted by an attacker. In this case it might happen that the attacker could cause the server to crash and restart, resetting the random seed to some value within a predictable window.
  - The `resetPassword` method only chooses passwords in a small space; the reset password on line 39 is simply a number in the range 0 to 32767. Perhaps this is a deliberate choice if it is part of a policy to make password resets easy (by emailing the user a short-lived token) but the temporary password ought really to have as much entropy as an ordinary password.
  - The use of a shared static instance for `Random` is suspect because a password reset for one user might be used to help predict the value of a password reset for the next user, especially since the reset password value is just an integer in the range 0 to 32767.
  - The size of the hash value is misunderstood, leading to a buffer overflow on line 33. The declared hash is 16 bytes long, corresponding to a 128-bit hash, whereas SHA-1 is 160-bits (20 bytes) long. This is simply an outright bug in the code that should be caught by testing,

because it will lead to an overflow as soon as `resetPassword()` is called. But if a bug like this in the password recovery mechanism escapes early detection it could lead to easy DoS attacks on end users.

  - The class is not thread-safe: if two threads are accessing the same `UserInfo` object there can be race conditions, for example, allowing an old password to be using in `tryLogin` while another thread has invoked `resetPassword` meanwhile. Similarly, if two different `UserInfo` objects at the same time, there will be race conditions on the static fields `sha1` and `rand`.

- **Additional functionality, recommendations**:
  - An obvious gap is that once a user is authenticated, there is no way to log out to reset the `authenticated` flag.
  - Another obvious gap is that the API offers a way for an initial user-generated password to be given, but there is no way to reset the password with a user-generated password.
  - A general concern on the strategy here is that the authentication check is separated from its use (TOCTOU), by storing the authentication flag and leaving any time-out policy to the client code of this module. A much better approach is to use the scoped **permissions** mechanism provided by Java.

[ 20 marks, following the marking advice given in the question ]