MSC62-J. Store passwords using a hash function

Programs that store passwords as cleartext (unencrypted text data) risk exposure of those passwords in a variety of ways. Although programs generally receive passwords from users as cleartext, they should ensure that the passwords are not stored as cleartext.

An acceptable technique for limiting the exposure of passwords is the use of hash functions, which allow programs to indirectly compare an input password to the original password string without storing a cleartext or decryptable version of the password. This approach minimizes the exposure of the password without presenting any practical disadvantages.

Cryptographic Hash Functions

The value produced by a hash function is the hash value or message digest. Hash functions are computationally feasible functions whose inverses are computationally infeasible. In practice, a password can be encoded to a hash value, but decoding remains infeasible. The equality of passwords can be tested through the equality of their hash values.

A good practice is to always append a salt to the password being hashed. A salt is a unique (often sequential) or randomly generated piece of data that is stored along with the hash value. The use of a salt helps prevent brute-force attacks against the hash value, provided that the salt is long enough to generate sufficient entropy (shorter salt values cannot significantly slow down a brute-force attack). Each password should have its own salt associated with it. If a single salt were used for more than one password, two users would be able to see whether their passwords are the same.

The choice of hash function and salt length presents a trade-off between security and performance. Increasing the effort required for effective brute-force attacks by choosing a stronger hash function can also increase the time required to validate a password. Increasing the length of the salt makes brute-force attacks more difficult but requires additional storage space.

Java’s MessageDigest class provides implementations of various cryptographic hash functions. Avoid defective functions such as the Message-Digest Algorithm (MD5). Hash functions such as Secure Hash Algorithm (SHA)-1 and SHA-2 are maintained by the National Security Agency and are currently considered safe. In practice, many applications use SHA-256 because this hash function has reasonable performance while still being considered secure.

Noncompliant Code Example

This noncompliant code example encrypts and decrypts the password stored in password.bin using a symmetric key algorithm:

```java
public final class Password {
    private void setPassword(byte[] pass) throws Exception {
        // Arbitrary encryption scheme
        byte[] encrypted = encrypt(pass);
        clearArray(pass);
        // Encrypted password to password.bin
        saveBytes(encrypted, "password.bin");
        clearArray(encrypted);
    }

    boolean checkPassword(byte[] pass) throws Exception {
        // Load the encrypted password
        byte[] encrypted = loadBytes("password.bin");
        byte[] decrypted = decrypt(encrypted);
        boolean arraysEqual = Arrays.equal(decrypted, pass);
        clearArray(decrypted);
        return arraysEqual;
    }

    private void clearArray(byte[] a) {
        for (int i = 0; i < a.length; i++) {
            a[i] = 0;
        }
    }
}
```

An attacker could potentially decrypt this file to discover the password, particularly when the attacker has knowledge of the key and encryption scheme used by the program. Passwords should be protected even from system administrators and privileged users. Consequently, using encryption is only partly effective in mitigating password disclosure threats.

Noncompliant Code Example

This noncompliant code example uses the SHA-256 hash function through the MessageDigest class to compare hash values instead of cleartext strings. It uses SecureRandom to generate a strong salt, as recommended by MSC02-J. Generate strong random numbers. However, it uses a String to store the password:
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public final class Password {
    private SecureRandom random = new SecureRandom();

    private void setPassword(String pass) throws Exception {
        byte[] salt = new byte[12];
        random.nextBytes(salt);
        MessageDigest msgDigest = MessageDigest.getInstance("SHA-256");
        // Encode the string and salt
        byte[] hashVal = msgDigest.digest((pass+salt).getBytes());
        saveBytes(salt, "salt.bin");
        // Save the hash value to password.bin
        saveBytes(hashVal,"password.bin");
    }

    boolean checkPassword(String pass) throws Exception {
        byte[] salt = loadBytes("salt.bin");
        MessageDigest msgDigest = MessageDigest.getInstance("SHA-256");
        // Encode the string and salt
        byte[] hashVal1 = msgDigest.digest((pass+salt).getBytes());
        // Load the hash value stored in password.bin
        byte[] hashVal2 = loadBytes("password.bin");
        return Arrays.equals(hashVal1, hashVal2);
    }
}

Even when an attacker knows that the program stores passwords using SHA-256 and a 12-byte salt, he or she will be unable to retrieve the actual password from password.bin and salt.bin.

Although this approach solves the decryption problem from the previous noncompliant code example, this program may inadvertently store the passwords as cleartext in memory. Java String objects are immutable and can be copied and internally stored by the Java Virtual Machine. Consequently, Java lacks a mechanism to securely erase a password once it has been stored in a String. See MSC59-J. Limit the lifetime of sensitive data for more information.

Compliant Solution

This compliant solution addresses the problems from the previous noncompliant code example by using a byte array to store the password:
import java.security.GeneralSecurityException;
import java.security.SecureRandom;
import java.security.spec.KeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

final class Password {
    private SecureRandom random = new SecureRandom();

    /* Set password to new value, zeroing out password */
    void setPassword(char[] pass) throws IOException, GeneralSecurityException {
        byte[] salt = new byte[12];
        random.nextBytes(salt);
        saveBytes(salt, "salt.bin");
        byte[] hashVal = hashPassword(pass, salt);
        saveBytes(hashVal, "password.bin");
        Arrays.fill(hashVal, (byte) 0);
    }

    /* Indicates if given password is correct */
    boolean checkPassword(char[] pass) throws IOException, GeneralSecurityException {
        byte[] salt = loadBytes("salt.bin");
        byte[] hashVal1 = hashPassword(pass, salt);
        // Load the hash value stored in password.bin
        byte[] hashVal2 = loadBytes("password.bin");
        boolean arraysEqual = timingEquals(hashVal1, hashVal2);
        Arrays.fill(hashVal1, (byte) 0);
        Arrays.fill(hashVal2, (byte) 0);
        return arraysEqual;
    }

    /* Encrypts password & salt and zeroes both */
    private byte[] hashPassword(char[] pass, byte[] salt) throws GeneralSecurityException {
        KeySpec spec = new PBEKeySpec(pass, salt, 65536, 128);
        Arrays.fill(pass, (char) 0);
        Arrays.fill(salt, (byte) 0);
        SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        return f.generateSecret(spec).getEncoded();
    }

    /* Indicates if both byte arrays are equal
    * but uses same amount of time if they are the same or different
    * to prevent timing attacks */
    public static boolean timingEquals(byte b1[], byte b2[]) {
        boolean result = true;
        int len = b1.length;
        if (len != b2.length) {
            result = false;
        }
        if (len > b2.length) {
            len = b2.length;
        }
        for (int i = 0; i < len; i++) {
            result &= (b1[i] == b2[i]);
        }
        return result;
    }

    private void saveBytes(byte[] bytes, String filename) throws IOException {
        // ... write bytes to the file
    }

    private byte[] loadBytes(String filename) throws IOException {
        // ... read bytes to the file
    }
}
In both the `setPassword()` and `checkPassword()` methods, the cleartext representation of the password is erased immediately after it is converted into a hash value. Consequently, attackers must work harder to retrieve the cleartext password after the erasure. Providing guaranteed erasure is extremely challenging, is likely to be platform specific, and may even be impossible because of copying garbage collectors, dynamic paging, and other platform features that operate below the level of the Java language.

Furthermore, we use a `timingEquals()` method to validate the password. While doing a simple byte comparison, it takes the same time for both successful matches and unsuccessful ones; consequently thwarting timing attacks.

Finally, it uses PBKDF2 which, unlike `MessageDigest`, is specifically designed for hashing passwords.

**Applicability**

Passwords stored without a secure hash are exposed to malicious users. Violations of this guideline generally have a clear exploit associated with them.

Applications such as password managers may need to retrieve the original password to enter it into a third-party application. This is permitted even though it violates this guideline. The password manager is accessed by a single user and always has the user’s permission to store his or her passwords and to display those passwords on command. Consequently, the limiting factor to safety and security is the user’s competence rather than the program’s operation.

**Bibliography**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[API 2013]</td>
<td>Class <code>MessageDigest</code>, Class <code>String</code></td>
</tr>
<tr>
<td>[Hirondelle 2013]</td>
<td><code>Passwords Never Clear in Text</code></td>
</tr>
</tbody>
</table>