MSC61-J. Do not use insecure or weak cryptographic algorithms

Security-intensive applications *must* avoid use of insecure or weak cryptographic primitives to protect sensitive information. The computational capacity of modern computers permits circumvention of such cryptography via brute-force attacks. For example, the Data Encryption Standard (DES) encryption algorithm is considered highly insecure; messages encrypted using DES have been decrypted by brute force within a single day by machines such as the Electronic Frontier Foundation's (EFF) Deep Crack.

Noncompliant Code Example

This noncompliant code example encrypts a `String` input using a weak cryptographic algorithm (DES):

```java
SecretKey key = KeyGenerator.getInstance("DES").generateKey();
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, key);

// Encode bytes as UTF8; strToBeEncrypted contains
// the input string that is to be encrypted
byte[] encoded = strToBeEncrypted.getBytes("UTF8");

// Perform encryption
byte[] encrypted = cipher.doFinal(encoded);
```

Noncompliant Code Example

This noncompliant code example uses the Electronic Codebook (ECB) *mode of operation*, which is generally insecure.

```java
Cipher cipher = Cipher.getInstance("AES");  // defaults to ECB mode
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128); // 192 and 256 bits may be unavailable

SecretKey skey = kgen.generateKey();
byte[] raw = skey.getEncoded();

SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
cipher.init(Cipher.ENCRYPT_MODE, skeySpec);

// Encode bytes as UTF8; strToBeEncrypted contains
// the input string that is to be encrypted
byte[] encoded = strToBeEncrypted.getBytes("UTF8");

// Perform encryption
byte[] encrypted = cipher.doFinal(encoded);
```

Compliant Solution

This compliant solution uses the Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM) to perform the encryption. GCM has the benefit of providing authenticity (integrity) in addition to confidentiality. GCM is available by default in Java 8, but not Java 7. The same secret key can be used to encrypt multiple messages in GCM mode, but it is very important that a different initialization vector (IV) be used for each message. The below *encrypt_gcm* method produces a pair of (IV, ciphertext), which the *decrypt_gcm* method consumes. However, at the Java level, the *encrypt_gcm* method returns a single byte array that consists of the IV followed by the ciphertext, since in practice this is often easier to handle than a pair of byte arrays.
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;

class Msc61 {
    public static final int GCM_TAG_LENGTH = 16;
    public static final int GCM_IV_LENGTH = 12;

    public static SecretKey generateKey() {
        try {
            KeyGenerator kgen = KeyGenerator.getInstance("AES");
            kgen.init(128);
            return kgen.generateKey();
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException(e.toString());
        }
    }

    public static byte[] encrypt_gcm(SecretKey skey, String plaintext) {
        /* Precond: skey is valid and GCM mode is available in the JRE; */
        /* otherwise IllegalStateException will be thrown. */
        try {
            byte[] ciphertext = null;
            Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
            byte[] initVector = new byte[GCM_IV_LENGTH];
            (new SecureRandom()).nextBytes(initVector);
            GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * java.lang.Byte.SIZE, initVector);
            cipher.init(Cipher.ENCRYPT_MODE, skey, spec);
            byte[] encoded = plaintext.getBytes(java.nio.charset.StandardCharsets.UTF_8);
            ciphertext = new byte[initVector.length + cipher.getOutputSize(encoded.length)];
            for (int i=0; i < initVector.length; i++) {
                ciphertext[i] = initVector[i];
            }
            // Perform encryption
            cipher.doFinal(encoded, 0, encoded.length, ciphertext, initVector.length);
            return ciphertext;
        } catch (NoSuchPaddingException | InvalidAlgorithmParameterException | ShortBufferException |
            BadPaddingException | IllegalBlockSizeException | InvalidKeyException | NoSuchAlgorithmException e) {
            /* None of these exceptions should be possible if precond is met. */
            throw new IllegalStateException(e.toString());
        }
    }

    public static String decrypt_gcm(SecretKey skey, byte[] ciphertext) throws BadPaddingException, IllegalBlockSizeException /* these indicate corrupt or malicious ciphertext */ {
        /* Precond: skey is valid and GCM mode is available in the JRE; */
        /* otherwise IllegalStateException will be thrown. */
        try {
            Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
            byte[] initVector = Arrays.copyOfRange(ciphertext, 0, GCM_IV_LENGTH);
            GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * java.lang.Byte.SIZE, initVector);
            cipher.init(Cipher.DECRYPT_MODE, skey, spec);
            byte[] plaintext = cipher.doFinal(ciphertext, GCM_IV_LENGTH, ciphertext.length - GCM_IV_LENGTH);
            return new String(plaintext);
        } catch (NoSuchPaddingException | InvalidAlgorithmParameterException |
            InvalidKeyException | NoSuchAlgorithmException e) {
            /* None of these exceptions should be possible if precond is met. */
            throw new IllegalStateException(e.toString());
        }
    }
}
Compliant Solution

This compliant solution uses the Advanced Encryption Standard (AES) algorithm in Cipher Block Chaining (CBC) mode to perform the encryption. It uses the "AES/CBC/PKCS5Padding" transformation, which the Java documentation guarantees to be available on all conforming implementations of the Java platform. However, CBC mode does not incorporate any authentication checks. Therefore, a separate message authentication code (MAC) should be generated by the sender after encryption and verified by the receiver before decryption. (Note that verifying the MAC after decryption, rather than before decryption, can introduce a "padding oracle" vulnerability.)

```java
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;

class Msc61 {
    public static SecretKey generateKey() {
        try {
            KeyGenerator kgen = KeyGenerator.getInstance("AES");
            kgen.init(128);
            return kgen.generateKey();
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException(e.toString());
        }
    }

    public static byte[] encrypt_cbc(SecretKey skey, String plaintext) {
        /* Precond: skey is valid; otherwise IllegalStateException will be thrown. */
        try {
            byte[] ciphertext = null;
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            final int blockSize = cipher.getBlockSize();
            byte[] initVector = new byte[blockSize];
            (new SecureRandom()).nextBytes(initVector);
            IvParameterSpec ivSpec = new IvParameterSpec(initVector);
            cipher.init(Cipher.ENCRYPT_MODE, skey, ivSpec);
            byte[] encoded = plaintext.getBytes(java.nio.charset.StandardCharsets.UTF_8);
            ciphertext = new byte[initVector.length + cipher.getOutputSize(encoded.length)];
            for (int i=0; i < initVector.length; i++) {
                ciphertext[i] = initVector[i];
            }
            // Perform encryption
            cipher.doFinal(encoded, 0, encoded.length, ciphertext, initVector.length);
            return ciphertext;
        } catch (NoSuchPaddingException | InvalidAlgorithmParameterException |
            ShortBufferException | BadPaddingException | IllegalBlockSizeException |
            InvalidKeyException | NoSuchAlgorithmException e) {
            /* None of these exceptions should be possible if precond is met. */
            throw new IllegalStateException(e.toString());
        }
    }

    public static String decrypt_cbc(SecretKey skey, byte[] ciphertext) throws BadPaddingException, IllegalBlockSizeException /* these indicate corrupt or malicious ciphertext */ {
        try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            final int blockSize = cipher.getBlockSize();
            byte[] initVector = Arrays.copyOfRange(ciphertext, 0, blockSize);
            IvParameterSpec ivSpec = new IvParameterSpec(initVector);
            cipher.init(Cipher.DECRYPT_MODE, skey, ivSpec);
            byte[] plaintext = cipher.doFinal(ciphertext, blockSize, ciphertext.length - blockSize);
            return new String(plaintext);
        } catch (NoSuchPaddingException | InvalidAlgorithmParameterException |
            InvalidKeyException | NoSuchAlgorithmException e) {
            /* None of these exceptions should be possible if precond is met. */
            throw new IllegalStateException(e.toString());
        }
    }
}
```
Both of the above compliant solutions use 128-bit AES keys. Longer keys (192-bit and 256-bit) may be available if the "Unlimited Strength Jurisdiction Policy" files are installed and available to the Java runtime environment. A brute-force attack against 128-bit AES keys would take billions of years with current computational resources, so absent a cryptographic weakness in AES, 128-bit keys are likely suitable for secure encryption.

Applicability

Use of mathematically and computationally insecure cryptographic algorithms can result in the disclosure of sensitive information.

Weak cryptographic algorithms can be disabled in Java SE 7; see the Java PKI Programmer's Guide, Appendix D: Disabling Cryptographic Algorithms [Oracle 2011a].

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SonarQube</td>
<td>6.7</td>
<td>S2278</td>
<td></td>
</tr>
</tbody>
</table>

Related Guidelines

| SEI CERT C Coding Standard | MSC25-C. Do not use insecure or weak cryptographic algorithms |

Bibliography

<table>
<thead>
<tr>
<th>Oracle 2011a</th>
<th>Appendix D: Disabling Cryptographic Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle 2013b</td>
<td>Java Cryptography Architecture (JCA) Reference Guide</td>
</tr>
</tbody>
</table>