Secure Coding Standard for Java Development

While the Java security architecture can protect users and systems from hostile programs downloaded over a network, it cannot defend against implementation bugs that occur in trusted code. To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should consider following coding guidelines.

These twenty golden rules are to help building leak proof applications on java platform. These rules are written in no particular order, but to cover as much surface area as possible. Not all of the rules apply to all Java language programs; frequently, their applicability depends on how the software is deployed and your assumptions concerning trust.

Index of Rules:

Contents

Rule#1: Prevent SQL injection

SQL injection vulnerabilities helps malicious user alter's the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

Non-Compliant Solution:

```
String sqlString = "SELECT * FROM db_user WHERE username = '"
                         + username +
                         "' AND password = '" + pwd + "'";
      Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery(sqlString);
```

Compliant Solution:

This compliant solution uses a parametric query with a `?` character as a placeholder for the argument. This code also validates the length of the `username` argument, preventing an attacker from submitting an arbitrarily long user name.

```
      String sqlString =
      "select * from db_user where username=? and password=?";
    PreparedStatement stmt = connection.prepareStatement(sqlString);
    stmt.setString(1, username);
    stmt.setString(2, pwd);
    ResultSet rs = stmt.executeQuery();
```

Rule#2: Normalize strings before validating them

Normalization is the process of removing unnecessary "." and ".." segments/VALUES from the parameter.

Normalization is important because in Unicode, the same string can have many different representations

The `Normalizer.normalize()` method transforms Unicode text into the standard normalization forms.

This noncompliant code example attempts to validate the `String` before performing normalization

```
// \uFE64 is normalized to < and \uFE65 is normalized to > using the NFKC
normalization form
String s = "\uFE64" + "script" + "\uFE65";

// Validate
Pattern pattern = Pattern.compile("[<>]"); // Check for angle brackets
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
  // Found black listed tag
  throw new IllegalStateException();
} else {
  // ...
}

// Normalize
s = Normalizer.normalize(s, Form.NFKC);
```

This compliant solution normalizes the string before validating it. Consequently, input validation correctly detects the malicious input and throws an`IllegalStateException`.

```
String s = "\uFE64" + "script" + "\uFE65";

// Normalize
s = Normalizer.normalize(s, Form.NFKC);

// Validate
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
  // Found blacklisted tag
  throw new IllegalStateException();
} else {
  // ...
}
```

Rule#3: Do not log unsanitized user input

A log injection arises when a log entry contains unsanitized user input. A malicious user can insert fake log data and consequently deceive system administrators as to the system's behavior.

This noncompliant code example logs untrusted data from an unauthenticated user without data sanitization.

```
if (loginSuccessful) {
  logger.severe("User login succeeded for: " + username);
} else {
  logger.severe("User login failed for: " + username);
}
```

This compliant solution sanitizes the `username` before logging it, preventing injection attacks.

```
if (loginSuccessful) {
  logger.severe("User login succeeded for: " + sanitizeUser(username));
} else {
  logger.severe("User login failed for: " + sanitizeUser(username));
}
public String sanitizeUser(String username) {
  return Pattern.matches("[A-Za-z0-9_]+", username))
      ? username : "unauthorized user";
}
```

Rule#4: Sanitize untrusted data passed to the Runtime.exec() method

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `Runtime.getRuntime()` method. The semantics of `Runtime.exec()` are poorly defined, so it is best not to rely on its behavior any more than necessary, but typically it invokes the command directly without a shell.

This noncompliant code example provides a directory listing using the `dir` command. It is implemented using `Runtime.exec()` to invoke the Windows `dir` command.

```
class DirList {
  public static void main(String[] args) throws Exception {
    String dir = System.getProperty("dir");
    Runtime rt = Runtime.getRuntime();
    Process proc = rt.exec("cmd.exe /C dir " + dir);

}

}
```

When the task performed by executing a system command can be accomplished by some other means, it is almost always advisable to do so. This compliant solution uses the `File.list()` method to provide a directory listing, eliminating the possibility of command or argument injection attacks

```
import java.io.File;

class DirList {
  public static void main(String[] args) throws Exception {
    File dir = new File(System.getProperty("dir"));
    if (!dir.isDirectory()) {
      System.out.println("Not a directory");
    } else {
      for (String file : dir.list()) {
        System.out.println(file);
      }
    }
  }
}
```

Rule#5: Perform any string modifications before validation

It is important that a string not be modified after validation has occurred because doing so may allow an attacker to bypass validation. For example, a program may filter out the `<script>` tags from HTML input to avoid cross-site scripting (XSS) and other vulnerabilities. If exclamation marks (!) are deleted from the input following validation, an attacker may pass the string `"<scr!ipt>"` so that the validation check fails to detect the `<script>` tag, but the subsequent removal of the exclamation mark creates a `<script>` tag in the input.

Non Compliant Solution:

```
public class TagFilter {
  public static String filterString(String str) {
    String s = Normalizer.normalize(str, Form.NFKC);

    // Validate input
    Pattern pattern = Pattern.compile("<script>");
    Matcher matcher = pattern.matcher(s);
    if (matcher.find()) {
      throw new IllegalArgumentException("Invalid input");
    }

    // Deletes noncharacter code points
    s = s.replaceAll("[\\p{Cn}]", "");
    return s;
  }
```

Compliant Solution:

```
public class TagFilter {

  public static String filterString(String str) {
    String s = Normalizer.normalize(str, Form.NFKC);

    // Replaces all noncharacter code points with Unicode U+FFFD
    s = s.replaceAll("[\\p{Cn}]", "\uFFFD");

    // Validate input
    Pattern pattern = Pattern.compile("<script>");
    Matcher matcher = pattern.matcher(s);
    if (matcher.find()) {
      throw new IllegalArgumentException("Invalid input");
    }
    return s;
  }
```

Rule#6. Specify an appropriate locale when comparing locale-dependent data

Using locale-dependent methods on locale-dependent data can produce unexpected results when the locale is unspecified. Programming language identifiers, protocol keys, and HTML tags are often specified in a particular locale, usually `Locale.ENGLISH`. Running a program in a different locale may result in unexpected program behavior or even allow an attacker to bypass input filters. For these reasons, any program that inspects data generated by a locale-dependent function must specify the locale used to generate that data.

Non Compliant Code:

```
public static void processTag(String tag) {
  if (tag.toUpperCase().equals("SCRIPT")) {
    return;
  }
  // Process tag
}
```

Compliant Solution:

```
public static void processTag(String tag) {
  if (tag.toUpperCase(Locale.ENGLISH).equals("SCRIPT")) {
    return;
  }
  // Process tag
}
```

Rule#7: Limit accessibility of fields

Invariants cannot be enforced for public nonfinal fields or for final fields that reference a mutable object. A protected member of an exported class represents a public commitment to an implementation detail. Attackers can manipulate such fields to violate class invariants, or they may be corrupted by multiple threads accessing them concurrently. As a result, fields must be declared private or package-private.

```
public class Widget {
  public int total; // Number of elements

  void add() {
    if (total < Integer.MAX_VALUE) {
      total++;
      // ...
    } else {
      throw new ArithmeticException("Overflow");
    }
  }
}
```

Accessor methods provide controlled access to fields outside of the package in which their class is declared. This compliant solution declares `total` as private and provides a public accessor

```
public class Widget {
  private int total; // Declared private

  public int getTotal () {
    return total;
  }

  // Definitions for add() remain the same
}
```

Rule#8. Do not return references to private mutable class members

Returning references to internal mutable members of a class can compromise an application's security, both by breaking encapsulation and by providing the opportunity to corrupt the internal state of the class. As a result, programs must not return references to internal mutable classes.

This noncompliant code example shows a `getDate()` accessor method that returns the sole instance of the private `Date` object.

```
class MutableClass {
  private Date d;

  public MutableClass() {
    d = new Date();
  }

  public Date getDate() {
    return d;
  }
}
```

This compliant solution returns a clone of the `Date` object from the `getDate()` accessor method. Although `Date` can be extended by an attacker, this approach is safe because the `Date` object returned by `getDate()` is controlled by `MutableClass` and is known to be a nonmalicious subclass

```
public Date getDate() {
  return (Date)d.clone();
}
```

Rule#9: Be wary of letting constructors throw exceptions

An object is partially initialized if a constructor has begun building the object but has not finished. As long as the object is not fully initialized, it must be hidden from other classes.

This noncompliant code example defines the constructor of the `BankOperations` class so that it performs social security number (SSN) verification using the method `performSSNVerification()`. The implementation of the `performSSNVerification()` method assumes that an attacker does not know the correct SSN and trivially returns false.

```java
public class BankOperations {
  public BankOperations() {
    if (!performSSNVerification()) {
      throw new SecurityException("Access Denied!");
    }
  }

  private boolean performSSNVerification() {
    return false; // Returns true if data entered is valid, else false
                  // Assume that the attacker always enters an invalid SSN
  }

  public void greet() {
    System.out.println("Welcome user! You may now use all the features.");
  }
}
```

Rather than throwing an exception, this compliant solution uses an *initialized flag* to indicate whether an object was successfully constructed. The flag is initialized to false and set to true when the constructor finishes successfully.

```java
class BankOperations {
  private volatile boolean initialized = false;

  public BankOperations() {
    if (!performSSNVerification()) {
      return;                 // object construction failed
    }

    this.initialized = true; // Object construction successful
  }

  private boolean performSSNVerification() {
    return false;
  }

  public void greet() {
    if (!this.initialized) {
      throw new SecurityException("Invalid SSN!");
    }

    System.out.println(
        "Welcome user! You may now use all the features.");
  }}
```

Rule#10: Validate method arguments

Validate method arguments to ensure that they fall within the bounds of the method's intended design. This practice ensures that operations on the method's parameters yield valid results. Failure to validate method arguments can result in incorrect calculations, runtime exceptions, violation of class invariants, and inconsistent object state.

In this noncompliant code example, `setState()` fail to validate their arguments. A malicious caller could pass an invalid state to the library, consequently corrupting the library and exposing a vulnerability.

```
private Object myState = null;

// Sets some internal state in the library
void setState(Object state) {
  myState = state;
}
```

This compliant solution both validates the method arguments and verifies the internal state before use.

```
private Object myState = null;

// Sets some internal state in the library
void setState(Object state) {
  if (state == null) {
    // Handle null state
  }

  // Defensive copy here when state is mutable

  if (isInvalidState(state)) {
    // Handle invalid state
  }
  myState = state;
}
```

Rule#11: Do not catch NullPointerException or any of its ancestors

Programs must not catch `java.lang.NullPointerException`. A `NullPointerException` exception thrown at runtime indicates the existence of an underlying `null` pointer dereference that must be fixed in the application code.

This noncompliant code example defines an `isName()` method that takes a `String` argument and returns true if the given string is a valid name. A valid name is defined as two capitalized words separated by one or more spaces. Rather than checking to see whether the given string is null, the method catches `NullPointerException` and returns false.

```
boolean isName(String s) {
  try {
    String names[] = s.split(" ");

    if (names.length != 2) {
      return false;
    }
    return (isCapitalized(names[0]) && isCapitalized(names[1]));
  } catch (NullPointerException e) {
    return false;
  }

}
```

This compliant solution explicitly checks the `String` argument for `null` rather than catching.

```
NullPointerException:
boolean isName(String s) {
  if (s == null) {
    return false;
  }
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));

}
```

Rule#12: Do not allow serialization and deserialization to bypass the security manager

Serialization and deserialization features can be exploited to bypass security manager checks. A serializable class may contain security manager checks in its constructors for various reasons, including preventing untrusted code from modifying the internal state of the class. Such security manager checks must be replicated wherever a class instance can be constructed. For example, if a class enables a caller to retrieve sensitive internal state contingent upon security checks, those checks must be replicated during deserialization to ensure that an attacker cannot extract sensitive information by deserializing the object.

In this noncompliant code example, security manager checks are used within the constructor but are omitted from the `writeObject()` method that are used in the serialization-deserialization process. This omission allows untrusted code to maliciously create instances of the class.

```
public final class Hometown implements Serializable {
  // Private internal state
  private String town;
  private static final String UNKNOWN = "UNKNOWN";

  void performSecurityManagerCheck() throws AccessDeniedException {
    // ...
  }
  void validateInput(String newCC) throws InvalidInputException {
    // ...
  }
  public Hometown() {
    performSecurityManagerCheck();

    // Initialize town to default value
    town = UNKNOWN;
  }

  private void writeObject(ObjectOutputStream out) throws IOException {
    out.writeObject(town);
  }
   }
}
```

This compliant solution implements the required security manager checks in all constructors and methods that can either modify or retrieve internal state. Consequently, an attacker cannot create a modified instance of the object (using deserialization) or read the serialized byte stream to reveal serialized data.

```
public final class Hometown implements Serializable {
  // ... All methods the same except the following:

  // writeObject() correctly enforces checks during serialization
  private void writeObject(ObjectOutputStream out) throws IOException {
    performSecurityManagerCheck();
    out.writeObject(town);
  }
```

Rule#13: Generate strong random numbers

This noncompliant code example uses the insecure `java.util.Random` class. This class produces an identical sequence of numbers for each given seed value; consequently, the sequence of numbers is predictable. Consequently, the `java.util.Random` class must not be used either for security-critical applications or for protecting sensitive data.

```
import java.util.Random;
// ...

Random number = new Random(123L);
//...
for (int i = 0; i < 20; i++) {
  // Generate another random integer in the range [0, 20]
  int n = number.nextInt(21);
  System.out.println(n);
}
```

This compliant solution uses the `java.security.SecureRandom` class to produce high-quality random numbers

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
// ...

public static void main (String args[]) {
  SecureRandom number = new SecureRandom();
  // Generate 20 integers 0..20
  for (int i = 0; i < 20; i++) {
    System.out.println(number.nextInt(21));
  }
}
```

Rule#14: Never hard code sensitive information

Anyone who has access to the class files can decompile them and discover the sensitive information. Hard coding sensitive information also increases the need to manage and accommodate changes to the code.

This noncompliant code example includes a hard-coded server IP address in a constant `String`

```
class IPaddress {
  String ipAddress = new String("172.16.254.1");
  public static void main(String[] args) {
    //...
  }
}
```

A malicious user can use the `javap -c IPaddress` command to disassemble the class and discover the hard-coded server IP address.

This compliant solution retrieves the server IP address from an external file located in a secure directory

```
class IPaddress {
  public static void main(String[] args) throws IOException {
    char[] ipAddress = new char[100];
    int offset = 0;
    int charsRead = 0;
    BufferedReader br = null;
    try {
      br = new BufferedReader(new InputStreamReader(
              new FileInputStream("serveripaddress.txt")));
      while ((charsRead = br.read(ipAddress, offset, ipAddress.length -
offset))
          != -1) {
        offset += charsRead;
        if (offset >= ipAddress.length) {
          break;
        }
      }

      // ... Work with IP address

    } finally {
      Arrays.fill(ipAddress,  (byte) 0);
      br.close();
    }
  }
}
```

Rule#15: Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance (loadLibrary)

Because native code bypasses all of the security checks enforced by the Java Runtime Environment and other built-in protections provided by the Java virtual machine, only trusted code should be allowed to load native libraries.
In this noncompliant example, the Trusted class has permission to load libraries while the Untrusted class does not. However, the Trusted class provides a library loading service through a public method thus allowing the Untrusted class to load any libraries it desires.

```
// Trusted.java
public class Trusted {
    public static void loadLibrary(final String library){
       AccessController.doPrivileged(new PrivilegedAction<Void>() {
          public Void run() {
              System.loadLibrary(library);
              return null;
          }
       });}}
 // Untrusted.java

public class Untrusted {
    private native void nativeOperation();
    public static void main(String[] args) {
       String library = new String("NativeMethodLib");
       Trusted.loadLibrary(library);
       new Untrusted.nativeOperation();  // invoke the native method
   }}
```

In this compliant example, the Trusted class loads any necessary native libraries during initialization and then provides access through public native method wrappers. These wrappers perform the necessary security checks and data validation to ensure that untrusted code cannot exploit the native methods.

```
// Trusted.java
public class Trusted {
    // load native libraries
   static{
      System.loadLibrary("NativeMethodLib1");
   }
   // private native methods
   private native void nativeOperation1(byte[] data, int offset, int len);

   // wrapper methods perform SecurityManager and input validation checks
   public void doOperation1(byte[] data, int offset, int len) {
      // permission needed to invoke native method
      securityManagerCheck();

      if ((offset < 0) || (len < 0) || (offset > (data.length - len))) {
         throw new IllegalArgumentException();
      }
      nativeOperation1(data, offset, len);
   }  }
```

Rule#16: Do not allow tainted variables in privileged blocks

Do not operate on unvalidated or untrusted data (also known as tainted data) in a privileged block. An attacker can access a protected file by supplying its path name as an argument to this method that could result in privilege escalation attacks.

This noncompliant code example accepts a tainted path or file name as an argument

```
private void privilegedMethod(final String filename)
                                      throws FileNotFoundException {
  try {
    FileInputStream fis =
        (FileInputStream) AccessController.doPrivileged(
          new PrivilegedExceptionAction() {
        public FileInputStream run() throws FileNotFoundException {
          return new FileInputStream(filename);
        }
      }
    );
    // Do something with the file and then close it
  } catch (PrivilegedActionException e) {
    // Forward to handler
  }
}
```

This compliant solution invokes the `cleanAFilenameAndPath()` method to sanitize malicious inputs. Successful completion of the sanitization method indicates that the input is acceptable and the `doPrivileged()` block can be executed.

```
private void privilegedMethod(final String filename)
                                      throws FileNotFoundException {
  final String cleanFilename;
  try {
    cleanFilename = cleanAFilenameAndPath(filename);
  } catch (/* exception as per spec of cleanAFileNameAndPath */) {
    // Log or forward to handler as appropriate based on specification
    // of cleanAFilenameAndPath
  }
  try {
    FileInputStream fis =
        (FileInputStream) AccessController.doPrivileged(
          new PrivilegedExceptionAction() {
        public FileInputStream run() throws FileNotFoundException {
          return new FileInputStream(cleanFilename);
        }
      }
    );
    // Do something with the file and then close it
  } catch (PrivilegedActionException e) {
    // Forward to handler
  }
}
```

Rule#17. Protect sensitive operations with security manager checks

This noncompliant code example instantiates a `Hashtable` and defines a `removeEntry()` method to allow the removal of its entries. This method is considered sensitive, perhaps because the hash table contains sensitive information. However, the method is public and nonfinal, which leaves it exposed to malicious callers.

```
class SensitiveHash {
  private Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

  public void removeEntry(Object key) {
    ht.remove(key);
  }
}
```

This compliant solution installs a security check to protect entries from being maliciously removed from the `Hashtable` instance. A `SecurityException` is thrown if the caller lacks the permission.

```
class SensitiveHash {
  private Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

  public void removeEntry(Object key) {
    check("removeKeyPermission");
    ht.remove(key);
  }

  private void check(String directive) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      sm.checkSecurityAccess(directive);
    }
  }
}
```

Rule#18: Do not rely on the default automatic signature verification provided by URLClassLoader

Code should be signed only if it requires elevated privileges to perform one or more tasks.

The non-compliant solution uses URLClassLoader which performs automatic verification integrity check; it fails to authenticate the loaded class because the check uses the public key contained within the JAR without validating the public key. The legitimate JAR file may be replaced with a malicious JAR file containing a different public key along with appropriately modified digest values.

Compliant Solution (`jarsigner`)

Users can—but usually do not—explicitly check JAR file signatures at the command line. This solution may be adequate for programs that require manual installation of JAR files. Any malicious tampering results in a `SecurityException` when the `jarsigner` tool is invoked with the `-verify` option

```
jarsigner -verify signed-updates-jar-file.jar
```

Compliant Solution (Certificate Chain)

When the local system cannot reliably verify the signature, the invoking program must verify the signature programmatically by obtaining the chain of certificates from the `CodeSource` of the class being loaded and checking whether any of the certificates belong to a trusted signer whose certificate has been securely obtained beforehand and stored in a local keystore. This compliant solution demonstrates the necessary modifications to the `invokeClass()` method

```
public void invokeClass(String name, String[] args)
    throws ClassNotFoundException, NoSuchMethodException,
           InvocationTargetException, GeneralSecurityException,
           IOException {
  Class c = loadClass(name);
  Certificate[] certs =
      c.getProtectionDomain().getCodeSource().getCertificates();
  if (certs == null) {
    // Return, do not execute if unsigned
    System.out.println("No signature!");
    return;
  }

  KeyStore ks = KeyStore.getInstance("JKS");
  ks.load(new FileInputStream(System.getProperty(
      "user.home"+ File.separator + "keystore.jks")),
      "loadkeystorepassword".toCharArray());
  // User is the alias
  Certificate pubCert = ks.getCertificate("user");
  // Check with the trusted public key, else throws exception
  certs[0].verify(pubCert.getPublicKey());
}
```

Rule#19: Call the superclass's getPermissions() method when writing a custom class loader

When a custom class loader must override the `getPermissions()` method, the implementation must consult the default system policy by explicitly invoking the superclass's `getPermissions()` method before assigning arbitrary permissions to the code source. A custom class loader that ignores the superclass's `getPermissions()` could load untrusted classes with elevated privileges.

This noncompliant code example shows a fragment of a custom class loader that extends the class `URLClassLoader`. It overrides the `getPermissions()` method but does not call its superclass's more restrictive `getPermissions()` method. Consequently, a class defined using this custom class loader has permissions that are completely independent of those specified in the systemwide policy file. In effect, the class's permissions override them.

```
protected PermissionCollection getPermissions(CodeSource cs) {
  PermissionCollection pc = new Permissions();
  // Allow exit from the VM anytime
  pc.add(new RuntimePermission("exitVM"));
  return pc;
}
```

In this compliant solution, the `getPermissions()` method calls `super.getPermissions()`. As a result, the default systemwide security policy is applied in addition to the custom policy.

```
protected PermissionCollection getPermissions(CodeSource cs) {
  PermissionCollection pc = super.getPermissions(cs);
  // Allow exit from the VM anytime
  pc.add(new RuntimePermission("exitVM"));
  return pc;
}
```

Rule#20: Place all security-sensitive code in a single JAR and sign and seal it

Attackers could link privileged code with malicious code if the privileged code directly or indirectly invokes code from another package. A package sealed within a JAR specifies that all classes defined in that package must originate from the same JAR. Otherwise, a SecurityException is thrown.
This noncompliant code example includes a `doPrivileged()` block and calls a method defined in a class in a different, untrusted JAR file.

```
package trusted;
import untrusted.RetValue;

public class MixMatch {
  private void privilegedMethod() throws IOException {
    try {
      final FileInputStream fis = AccessController.doPrivileged(
        new PrivilegedExceptionAction<FileInputStream>() {
          public FileInputStream run() throws FileNotFoundException {
            return new FileInputStream("file.txt");
          }});

  public static void main(String[] args) throws IOException {
    MixMatch mm = new MixMatch();
    mm.privilegedMethod();
  }
}
// In another JAR file:
package untrusted;

class RetValue {
  public int getValue() {
    return 1;
  }
}
```

This compliant solution combines all security-sensitive code into the same package and the same JAR file. It also reduces the accessibility of the `getValue()` method to package-private. Sealing the package is necessary to prevent attackers from inserting any rogue classes.

```
package trusted;
public class MixMatch {
  // ...
}
// In the same signed & sealed JAR file:
package trusted;
 class RetValue {
  int getValue() {
    return 1;    }}
```

To seal a package, use the `sealed` attribute in the JAR file's manifest file header, as follows:

```
Name: trusted/ // Package name
Sealed: true  // Sealed attribute
```