SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

Reflection enables a Java program to analyze and modify itself. In particular, a program can discover the values of field variables and change them [Forman 2005], [Sun 2002]. The Java reflection API includes a method that enables fields that are normally inaccessible to be accessed under reflection. The following code prints out the names and values of all fields of an object `someObject` of class `SomeClass`:

```
Field fields[] = SomeClass.class.getDeclaredFields();
for (Field field : fields) {
  if ( !Modifier.isPublic(field.getModifiers())) {
    field.setAccessible(true);
  }
  System.out.print("Field: " + field.getName());
  System.out.println(", value: " + field.get(someObject));
}
```

A field could be set to a new value as follows:

```
String newValue = reader.readLine();
field.set(someObject, returnValue(newValue, field.getType()));
```

When the default security manager is used, it prevents fields that are normally inaccessible from being accessed under reflection. The default security manager throws a `java.security.AccessControlException` in these circumstances. However, `java.lang.reflect.ReflectPermission` can be granted with action `suppressAccessChecks` to override this default behavior.

For example, the Java Virtual Machine (JVM) normally protects private members of a class from being accessed by an object of a different class. When a method uses reflection to access class members (that is, uses the APIs belonging to the `java.lang.reflect` package), the reflection uses the same restrictions. That is, a foreign object that cannot access private members of a class normally also cannot use reflection to access those members. However, a class with private members but also with a public method that uses reflection to indirectly access those members can inadvertently enable a foreign object to access those private members using the public method, bypassing the intended accessibility restrictions. Consequently, unwary programmers can create an opportunity for a privilege escalation attack by untrusted callers.

The following table lists the APIs that should be used with care [SCG 2009].

APIs That Mirror Language Checks
`java.lang.Class.newInstance()`
`java.lang.reflect.Constructor.newInstance()`
`java.lang.reflect.Field.get*()`
`java.lang.reflect.Field.set*()`
`java.lang.reflect.Method.invoke()`
`java.util.concurrent.atomic.AtomicIntegerFieldUpdater.newUpdater()`
`java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater()`
`java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater()`

Because the `setAccessible()` and `getAccessible()` methods of class `java.lang.reflect.Field` are used to instruct the JVM to override the language access checks, they perform standard (and more restrictive) security manager checks and consequently lack the vulnerability discussed in this rule. Nevertheless, these methods should be used only with extreme caution. The remaining `set*()` and `get*()` field reflection methods perform only the language access checks and are vulnerable.

Use of reflection complicates security analysis and can easily introduce security vulnerabilities. Consequently, programmers should avoid using the reflection APIs when it is feasible to do so. Exercise extreme caution when the use of reflection is necessary.

In particular, reflection must not be used to provide access to classes, methods, and fields unless those items are already accessible without the use of reflection. For example, the use of reflection to access or modify fields is not allowed unless those fields are already accessible and modifiable by other means, such as through getter and setter methods.

This rule is similar to MET04-J. Do not increase the accessibility of overridden or hidden methods, but it warns against using reflection, rather than inheritance, to subvert accessibility.

Noncompliant Code Example

In this noncompliant code example, the private fields `i` and `j` can be modified using reflection via a `Field` object. Furthermore, any class can modify these fields using reflection via the `zeroField()` method. However, only class `FieldExample` can modify these fields without the use of reflection.

Allowing hostile code to pass arbitrary field names to the `zeroField()` method can

- Leak information about field names by throwing an exception for invalid or inaccessible field names (see ERR01-J. Do not allow exceptions to expose sensitive information for additional information). This example complies with ERR01-J by catching the relevant exceptions at the end of the method.
- Access potentially sensitive data that is visible to `zeroField()` but is hidden from the attacking method. This privilege escalation attack can be difficult to find during code review because the specific field or fields being accessed are controlled by strings in the attacker's code rather than by locally visible source code.

```
class FieldExample {
  private int i = 3;
  private int j = 4;

  public String toString() {
    return "FieldExample: i=" + i + ", j=" + j;
  }

  public void zeroI() {
    this.i = 0;
  }

  public void zeroField(String fieldName) {
    try {
      Field f = this.getClass().getDeclaredField(fieldName);
      // Subsequent access to field f passes language access checks
      // because zeroField() could have accessed the field via
      // ordinary field references
      f.setInt(this, 0);
      // Log appropriately or throw sanitized exception; see EXC06-J
    } catch (NoSuchFieldException ex) {
      // Report to handler
    } catch (IllegalAccessException ex) {
      // Report to handler
    }
  }

  public static void main(String[] args) {
    FieldExample fe = new FieldExample();
    System.out.println(fe.toString());
    for (String arg : args) {
      fe.zeroField(arg);
      System.out.println(fe.toString());
    }
  }
}
```

Compliant Solution (Private)

When you must use reflection, make sure that the immediate caller (method) is isolated from hostile code by declaring it private or final, as in this compliant solution:

```
class FieldExample {
  // ...

  private void zeroField(String fieldName) {
    // ...
  }
}
```

Note that when language access checks are overridden using `java.lang.reflect.Field.setAccessible`, the immediate caller gains access even to the private fields of other classes. To ensure that the security manager will block attempts to access private fields of other classes, never grant the permission `ReflectPermission` with action `suppressAccessChecks`.

Compliant Solution (Nonreflection)

When a class must use reflection to provide access to fields, it must also provide the same access using a nonreflection interface. This compliant solution provides limited setter methods that grant every caller the ability to zero out its fields without using reflection. If these setter methods comply with all other rules or security policies, the use of reflection also complies with this rule.

```
class FieldExample {
  // ...

  public void zeroField(String fieldName) {
    // ...
  }

  public void zeroI() {
    this.i = 0;
  }
  public void zeroJ() {
    this.j = 0;
  }
}
```

Noncompliant Code Example

In this noncompliant code example, the programmer intends that code outside the `Safe` package should be prevented from creating a new instance of an arbitrary class. Consequently, the `Trusted` class uses a package-private constructor. However, because the API is public, an attacker can pass `Trusted.class` itself as an argument to the `create()` method and bypass the language access checks that prevent code outside the package from invoking the package-private constructor. The `create()` method returns an unauthorized instance of the `Trusted` class.

```
package Safe;
public class Trusted {
  Trusted() { } // Package-private constructor
  public static <T> T create(Class<T> c)
      throws InstantiationException, IllegalAccessException {
    return c.newInstance();
  }
}

package Attacker;
import Safe.Trusted;

public class Attack {
  public static void main(String[] args)
      throws InstantiationException, IllegalAccessException {
    System.out.println(Trusted.create(Trusted.class)); // Succeeds
  }
}
```

In the presence of a security manager `s`, the `Class.newInstance()` method throws a security exception when (a) `s.checkMemberAccess(this, Member.PUBLIC)` denies creation of new instances of this class or (b) the caller's class loader is not the same class loader or an ancestor of the class loader for the current class, and invocation of `s.checkPackageAccess()` denies access to the package of this class.

The `checkMemberAccess` method allows access to public members and classes that have the same class loader as the caller. However, the class loader comparison is often insufficient; for example, all applets share the same class loader by convention, consequently allowing a malicious applet to pass the security check in this case.

Compliant Solution (Access Reduction)

This compliant solution reduces the access of the `create()` method to package-private, preventing a caller from outside the package from using that method to bypass the language access checks to create an instance of the `Trusted` class. Any caller that can create a `Trusted` class instance using reflection can simply call the `Trusted()` constructor instead.

```
package Safe;
public class Trusted {
  Trusted() { } // Package-private constructor
  static <T> T create(Class<T> c)
      throws InstantiationException, IllegalAccessException {
    return c.newInstance();
  }
}
```

Compliant Solution (Security Manager Check)

This compliant solution uses the `getConstructors()` method to check whether the class provided as an argument has public constructors. The security issue is irrelevant when public constructors are present because such constructors are already accessible even to malicious code. When public constructors are absent, the `create()` method uses the security manager's `checkPackageAccess()` method to ensure that all callers in the execution chain have sufficient permissions to access classes and their respective members defined in package `Safe`.

```
import java.beans.Beans;
import java.io.IOException;
package Safe;

public class Trusted  {
  Trusted() { }

  public static <T> T create(Class<T> c)
      throws InstantiationException, IllegalAccessException {

    if (c.getConstructors().length == 0) {  // No public constructors
      SecurityManager sm = System.getSecurityManager();
      if (sm != null) {
        // Throws an exception when access is not allowed
        sm.checkPackageAccess("Safe");
      }
    }
    return c.newInstance(); // Safe to return
  }
}
```

The disadvantage of this compliant solution is that the class must be granted reflection permissions to permit the call to `getConstructors()`.

Unknown macro: 'mc'

Compliant Solution (`java.beans` Package)

This compliant solution uses the `java.beans.Beans` API to check whether the `Class` object being received has any public constructors:

```
public class Trusted {
  Trusted() { }

  public static <T> T create(Class<T> c)
      throws IOException, ClassNotFoundException {

    // Executes without exception only if there are public constructors
    ClassLoader cl = new SafeClassLoader();
    Object b = Beans.instantiate(cl, c.getName());
    return c.cast(b);
  }
}
```

The `Beans.instantiate()` method succeeds only when the class being instantiated has a public constructor; otherwise, it throws an `IllegalAccessException`. The method uses a class loader argument along with the name of the class to instantiate. Unlike the previous compliant solution, this approach avoids the need for any reflection permissions.

Related Vulnerabilities

CERT Vulnerability #636312 describes an exploit in Java that allows malicious code to disable any security manager currently in effect. Among other vulnerabilities, the attack code exploited the following method defined in `sun.awt.SunToolkit`, for Java 7:

```
public static Field getField(final Class klass, final String fieldName) {
  return AccessController.doPrivileged(new PrivilegedAction<Field>() {
       public Field run() {
           try {
               Field field = klass.getDeclaredField(fieldName);
               assert (field != null);
               field.setAccessible(true);
               return field;
           } catch (SecurityException e) {
               assert false;
           } catch (NoSuchFieldException e) {
               assert false;
           }
           return null;
       }//run
  });
}
```

This code operates inside a `doPrivileged()` block. It then uses the reflection method `Class.getDeclaredField()` to obtain a field given the field's class and name. This method would normally be blocked by a security manager. It then uses the reflection method `Field.setAccessible()` to make the field accessible, even if it were protected or private. But this method is public, so anyone can call it.

Risk Assessment

Misuse of APIs that perform language access checks only against the immediate caller can break data encapsulation, leak sensitive information, or permit privilege escalation attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SEC05-J	High	Probable	Medium	P12	L1

Automated Detection

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**CODSTA.BP.ARM**	Implemented
SonarQube	6.7	**S3011**	Changing or bypassing accessibility is security-sensitive

Related Guidelines

Secure Coding Guidelines for Java SE, Version 5.0	Guideline 9-10 / ACCESS-10: Be aware of standard APIs that perform Java language access checks against the immediate caller

Android Implementation Details

Reflection can be used on Android, so this rule is applicable. Also, the use of reflection may allow a developer to access private Android APIs and so requires caution.

Bibliography

[Chan 1999]	`java.lang.reflect AccessibleObject`