SEC04-J. Protect sensitive operations with security manager checks

Sensitive operations must be protected by security manager checks.

Noncompliant Code Example

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);
  }
}
```

Compliant Solution

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 `java.security.SecurityPermission` `removeKeyPermission`.

```
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);
    }
  }
}
```

The `SecurityManager.checkSecurityAccess()` method determines whether or not the action controlled by the particular permission is allowed.

Noncompliant Code Example (`check*()`)

This noncompliant code example uses the `SecurityManager.checkRead()` method to check whether the file `schema.dtd` can be read from the file system. The `check*()` methods lack support for fine-grained access control. For example, the `check*()` methods are inadequate to enforce a policy permitting read access to all files with the `dtd` extension and forbidding read access to all other files. Code that is not itself part of the JDK must not override the `check*()` methods because the default implementations of the Java libraries already use these methods to protect sensitive operations.

```
SecurityManager sm = System.getSecurityManager();

if (sm != null) {  // Check whether file may be read
  sm.checkRead("/local/schema.dtd");
}
```

Compliant Solution (`checkPermission()`)

Java SE 1.2 added two methods—`checkPermission(Permission perm)` and `checkPermission(Permission perm, Object context)`—to the `SecurityManager` class. The motivations for this change included

- Eliminating the need to hard code names of checks in method names.
- Encapsulating the complicated algorithms and code for examining the Java runtime in a single `checkPermission()` method.

- Supporting introduction of additional permissions by subclassing the `Permission` class.

The single-argument `checkPermission()` method uses the context of the currently executing thread environment to perform the checks. If the context has the permissions defined in the local policy file, the check succeeds; otherwise, a `SecurityException` is thrown.

This compliant solution shows the single-argument `checkPermission()` method that checks to see whether the files in the `local` directory that have the `dtd` extension can be read. `DTDPermission` is a custom permission that enforces this level of access. Even if the `java.io.FilePermission` is granted to the application with the action `read`, `DTD` files are subject to additional access control.

```
SecurityManager sm = System.getSecurityManager();

if (sm != null) {  // Check whether file may be read
  DTDPermission perm = new DTDPermission("/local/",  "readDTD");
  sm.checkPermission(perm);
}
```

Compliant Solution (Multiple Threads)

Occasionally, the security check code exists in one context (such as a worker thread), whereas the check must be conducted on a different context, such as on another thread. The two-argument `checkPermission()` method is used in this case. It accepts an `AccessControlContext` instance as the `context` argument. The effective permissions are those of the `context` argument only rather than the intersection of the permissions of the two contexts.

Both the single- and double-argument `checkPermission()` methods defer to the single-argument `java.security.AccessController.checkPermission(Permission perm)` method. When invoked directly, this method operates only on the current execution context and, as a result, does not supersede the security manager's two-argument version.

A cleaner approach to making a security check from a different context is to take a *snapshot* of the execution context in which the check must be performed, using the `java.security.AccessController.getContext()` method that returns an `AccessControlContext` object. The `AccessControlContext` class itself defines a `checkPermission()` method that encapsulates a context instead of accepting the current executing context as an argument. This approach allows the check to be performed at a later time, as shown in the following example.

```
// Take the snapshot of the required context, store in acc, and pass it to another context
AccessControlContext acc = AccessController.getContext();

// Accept acc in another context and invoke checkPermission() on it
acc.checkPermission(perm);
```

Risk Assessment

Failure to enforce security checks in code that performs sensitive operations can lead to malicious tampering of sensitive data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SEC04-J	High	Probable	Medium	P12	L1

Automated Detection

Identifying sensitive operations requires assistance from the programmer; fully automated identification of sensitive operations is beyond the current state of the art.

Given knowledge of which operations are sensitive, as well as which specific security checks must be enforced for each operation, an automated tool could reasonably enforce the invariant that the sensitive operations are invoked *only* from contexts where the required security checks have been performed.

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**SECURITY.WSC.SCF**	Implemented

Android Implementation Details

The `java.security` package exists on Android for compatibility purposes only, and it should not be used.

Bibliography

[API 2014]