ENV01-J. Place all security-sensitive code in a single JAR and sign and seal it

In Java SE 6 and later, privileged code must either use the `AccessController` mechanism or be signed by an owner (or provider) whom the user trusts. Attackers could link privileged code with malicious code if the privileged code directly or indirectly invokes code from another package. Trusted JAR files often contain code that requires no elevated privileges itself but that depends on privileged code; such code is known as *security-sensitive code*. If an attacker can link security-sensitive code with malicious code, he or she can indirectly cause incorrect behavior. This exploit is called a *mix-and-match* attack.

Normally, execution of untrusted code causes loss of privileges; the Java security model rescinds privileges when a trusted method invokes an untrusted one. When trusted code calls untrusted code that attempts to perform some action requiring permissions withheld by the security policy, the Java security model disallows that action. However, privileged code may use a class that exists in an untrusted container and performs only unprivileged operations. If the attacker were to replace the class in the untrusted container with a malicious class, the trusted code might receive incorrect results and misbehave at the discretion of the malicious code.

According to the Java SE Documentation, "Extension Mechanism Architecture" [EMA 2014]:

> *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.*

Sealing a JAR file automatically enforces the requirement of keeping privileged code together. In addition, it is important to minimize the accessibility of classes and their members.

Noncompliant Code Example (Privileged Code)

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 {
      AccessController.doPrivileged(
        new PrivilegedExceptionAction<Void>() {
          public Void run() throws IOException, FileNotFoundException {
            final FileInputStream fis = new FileInputStream("file.txt");
            try {
              RetValue rt = new RetValue();

              if (rt.getValue() == 1) {
                // Do something with sensitive file
              }
            } finally {
              fis.close();
            }
            return null; // Nothing to return
          }
        }
      );
    } catch (PrivilegedActionException e) {
      // Forward to handler and log
    }
  }

  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;
  }
}
```

An attacker can provide an implementation of class `RetValue` so that the privileged code uses an incorrect return value. Even though class `MixMatch` consists only of trusted, signed code, an attacker can still cause this behavior by maliciously deploying a valid signed JAR file containing the untrusted `RetValue` class.

This example almost violates SEC01-J. Do not allow tainted variables in privileged blocks but does not do so. It instead allows potentially tainted code in its `doPrivileged()` block, which is a similar issue.

Noncompliant Code Example (Security-Sensitive Code)

This noncompliant code example improves on the previous example by moving the use of the `RetValue` class outside the `doPrivileged()` block:

```
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");
          }
        }
      );
      try {
        RetValue rt = new RetValue();

        if (rt.getValue() == 1) {
          // Do something with sensitive file
        }
      } finally {
        fis.close();
      }
    } catch (PrivilegedActionException e) {
      // Forward to handler and log
    }
  }

  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;
  }
}
```

Although the `RetValue` class is used only outside the `doPrivileged()` block, the behavior of `RetValue.getValue()` affects the behavior of security-sensitive code that operates on the file opened within the `doPrivileged()` block. Consequently, an attacker can still exploit the security-sensitive code with a malicious implementation of `RetValue`.

Compliant Solution

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
```

Exception

ENV01-J-EX0: Independent groups of privileged code and associated security-sensitive code (a "group" hereafter) may be placed in separate sealed packages and even in separate JAR files, subject to the following enabling conditions:

- The code in any one of these independent groups must lack any dynamic or static dependency on any of the code in any of the other groups. This means that code from one such group cannot invoke code from any of the others, whether directly or transitively.
- All code from any single group is contained within one or more sealed packages.
- All code from any single group is contained within a single signed JAR file.

Risk Assessment

Failure to place all privileged code together in one package and seal the package can lead to mix-and-match attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV01-J	High	Probable	Medium	**P12**	**L1**

Automated Detection

Detecting code that should be considered privileged or sensitive requires programmer assistance. Given identified privileged code as a starting point, automated tools could compute the closure of all code that can be invoked from that point. Such a tool could plausibly determine whether all code in that closure exists within a single package. A further check of whether the package is sealed is feasible.

Android Implementation Details

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

Related Guidelines

MITRE CWE	CWE-349, Acceptance of Extraneous Untrusted Data with Trusted Data

Bibliography

[EMA 2014]	Extension Mechanism Architecture, "Optional Package Sealing"
[McGraw 1999]	Rule 7, If you must sign your code, put it all in one archive file
[Ware 2008]	