IDS52-J. Prevent code injection

Code injection can occur when untrusted input is injected into dynamically constructed code. One obvious source of potential vulnerabilities is the use of JavaScript from Java code. The `javax.script` package consists of interfaces and classes that define Java scripting engines and a framework for the use of those interfaces and classes in Java code. Misuse of the `javax.script` API permits an attacker to execute arbitrary code on the target system.

This guideline is a specific instance of IDS00-J. Prevent SQL injection.

Noncompliant Code Example

This noncompliant code example incorporates untrusted user input into a JavaScript statement that is responsible for printing the input:

```
private static void evalScript(String firstName) throws ScriptException {
  ScriptEngineManager manager = new ScriptEngineManager();
  ScriptEngine engine = manager.getEngineByName("javascript");
  engine.eval("print('"+ firstName + "')");
}
```

An attacker can enter a specially crafted argument in an attempt to inject malicious JavaScript. This example shows a malicious string that contains JavaScript code that can create a file or overwrite an existing file on a vulnerable system.

```
dummy\');
var bw = new JavaImporter(java.io.BufferedWriter);
var fw = new JavaImporter(java.io.FileWriter);
with(fw) with(bw) {
   bwr = new BufferedWriter(new FileWriter(\"config.cfg\"));
   bwr.write(\"some text\"); bwr.close();
}
// ;
```

The script in this example prints `"dummy"` and then writes `"some text"` to a configuration file called `config.cfg`. An actual exploit can execute arbitrary code.

Compliant Solution (Whitelisting)

The best defense against code injection vulnerabilities is to prevent the inclusion of executable user input in code. User input used in dynamic code must be sanitized, for example, to ensure that it contains only valid, whitelisted characters. Sanitization is best performed immediately after the data has been input, using methods from the data abstraction used to store and process the data. Refer to IDS00-J. Sanitize untrusted data passed across a trust boundary for more details. If special characters must be permitted in the name, they must be normalized before comparison with their equivalent forms for the purpose of input validation. This compliant solution uses whitelisting to prevent unsanitized input from being interpreted by the scripting engine.

```
private static void evalScript(String firstName) throws ScriptException {
  // Allow only alphanumeric and underscore chars in firstName
  // (modify if firstName may also include special characters)
  if (!firstName.matches("[\\w]*")) {
    // String does not match whitelisted characters
    throw new IllegalArgumentException();
  }

  ScriptEngineManager manager = new ScriptEngineManager();
  ScriptEngine engine = manager.getEngineByName("javascript");
  engine.eval("print('"+ firstName + "')");
}
```

Compliant Solution (Secure Sandbox)

An alternative approach is to create a secure sandbox using a security manager (see SEC54-J. Create a secure sandbox using a security manager.) The application should prevent the script from executing arbitrary commands, such as querying the local file system. The two-argument form of `doPrivileged()` can be used to lower privileges when the application must operate with higher privileges, but the scripting engine must not. The `RestrictedAccessControlContext` reduces the permissions granted in the default policy file to those of the newly created protection domain. The effective permissions are the intersection of the permissions of the newly created protection domain and the systemwide security policy. Refer to SEC50-J. Avoid granting excess privileges for more details on the two-argument form of `doPrivileged()`.

This compliant solution illustrates the use of an `AccessControlContext` in the two-argument form of `doPrivileged()`.

```
class ACC {
  private static class RestrictedAccessControlContext {
    private static final AccessControlContext INSTANCE;

    static {
      INSTANCE = new AccessControlContext(
                  new ProtectionDomain[] {
                    new ProtectionDomain(null, null) // No permissions
                  });
    }
  }

  private static void evalScript(final String firstName)
      throws ScriptException {
    ScriptEngineManager manager = new ScriptEngineManager();
    final ScriptEngine engine = manager.getEngineByName("javascript");
    // Restrict permission using the two-argument form of doPrivileged()
    try {
      AccessController.doPrivileged(
        new PrivilegedExceptionAction<Object>() {

          public Object run() throws ScriptException {
            engine.eval("print('" + firstName + "')");
            return null;
          }
        },
        // From nested class
        RestrictedAccessControlContext.INSTANCE);

    } catch (PrivilegedActionException pae) {
      // Handle error
    }
  }
}
```

This approach can be combined with whitelisting for additional security.

Applicability

Failure to prevent code injection can result in the execution of arbitrary code.

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)

Bibliography

[API 2013]	Package `javax.script`
[OWASP 2013]	Code Injection in Java