IDS16-J. Prevent XML Injection

The extensible markup language (XML) is designed to help store, structure, and transfer data. Because of its platform independence, flexibility, and relative simplicity, XML has found use in a wide range of applications. However, because of its versatility, XML is vulnerable to a wide spectrum of attacks, including *XML injection*.

A user who has the ability to provide input string data that is incorporated into an XML document can inject XML tags. These tags are interpreted by the XML parser and may cause data to be overridden.

An online store application that allows the user to specify the quantity of an item available for purchase might generate the following XML document:

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity>
</item>
```

An attacker might input the following string instead of a count for the quantity:

```
1</quantity><price>1.0</price><quantity>1
```

In this case, the XML resolves to the following:

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity><price>1.0</price><quantity>1</quantity>
</item>
```

An XML parser may interpret the XML in this example such that the second price field overrides the first, changing the price of the item to $1. Alternatively, the attacker may be able to inject special characters, such as comment blocks and `CDATA` delimiters, which corrupt the meaning of the XML.

Noncompliant Code Example

In this noncompliant code example, a client method uses simple string concatenation to build an XML query to send to a server. XML injection is possible because the method performs no input validation.

```
import java.io.BufferedOutputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class OnlineStore {
  private static void createXMLStreamBad(final BufferedOutputStream outStream,
      final String quantity) throws IOException {
    String xmlString = "<item>\n<description>Widget</description>\n"
        + "<price>500</price>\n" + "<quantity>" + quantity
        + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
}
```

Compliant Solution (Input Validation)

Depending on the specific data and command interpreter or parser to which data is being sent, appropriate methods must be used to sanitize untrusted user input. This compliant solution validates that `quantity` is an unsigned integer:

```
import java.io.BufferedOutputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class OnlineStore {
  private static void createXMLStream(final BufferedOutputStream outStream,
      final String quantity) throws IOException, NumberFormatException {
    // Write XML string only if quantity is an unsigned integer (count).
    int count = Integer.parseUnsignedInt(quantity);
    String xmlString = "<item>\n<description>Widget</description>\n"
        + "<price>500</price>\n" + "<quantity>" + count + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
}
```

Compliant Solution (XML Schema)

A more general mechanism for checking XML for attempted injection is to validate it using a Document Type Definition (DTD) or schema. The schema must be rigidly defined to prevent injections from being mistaken for valid XML. Here is a suitable schema for validating our XML snippet:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="item">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
      <xs:element name="quantity" type="xs:nonNegativeInteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The schema is available as the file `schema.xsd`. This compliant solution employs this schema to prevent XML injection from succeeding. It also relies on the `CustomResolver` class defined in IDS17-J. Prevent XML External Entity Attacks to prevent XML external entity (XXE) attacks.

```
import java.io.BufferedOutputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.StringReader;

import javax.xml.XMLConstants;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;

import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;

public class OnlineStore {

  private static void createXMLStream(final BufferedOutputStream outStream,
      final String quantity) throws IOException {
    String xmlString;
    xmlString = "<item>\n<description>Widget</description>\n"
        + "<price>500.0</price>\n" + "<quantity>" + quantity
        + "</quantity></item>";
    InputSource xmlStream = new InputSource(new StringReader(xmlString));
    // Build a validating SAX parser using our schema
    SchemaFactory sf = SchemaFactory
        .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    DefaultHandler defHandler = new DefaultHandler() {
      public void warning(SAXParseException s) throws SAXParseException {
        throw s;
      }
      public void error(SAXParseException s) throws SAXParseException {
        throw s;
      }
      public void fatalError(SAXParseException s) throws SAXParseException {
        throw s;
      }
    };
    StreamSource ss = new StreamSource(new File("schema.xsd"));
    try {
      Schema schema = sf.newSchema(ss);
      SAXParserFactory spf = SAXParserFactory.newInstance();
      spf.setSchema(schema);
      SAXParser saxParser = spf.newSAXParser();
      // To set the custom entity resolver,
      // an XML reader needs to be created
      XMLReader reader = saxParser.getXMLReader();
      reader.setEntityResolver(new CustomResolver());
      saxParser.parse(xmlStream, defHandler);
    } catch (ParserConfigurationException x) {
      throw new IOException("Unable to validate XML", x);
    } catch (SAXException x) {
      throw new IOException("Invalid quantity", x);
    }
    // Our XML is valid, proceed
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
}
```

Using a schema or DTD to validate XML is convenient when receiving XML that may have been loaded with unsanitized input. If such an XML string has not yet been built, sanitizing input before constructing XML yields better performance.

Risk Assessment

Failure to sanitize user input before processing or storing it can result in injection attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS16-J	High	Probable	Medium	**P12**	**L1**

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
Fortify	1.0	**Missing_XML_Validation**	Implemented
Parasoft Jtest	2020.2	**BD.SECURITY.TDXML**	Protect against XML data injection

Related Vulnerabilities

CVE-2008-2370 describes a vulnerability in Apache Tomcat 4.1.0 through 4.1.37, 5.5.0 through 5.5.26, and 6.0.0 through 6.0.16. When a `RequestDispatcher` is used, Tomcat performs path normalization before removing the query string from the URI, which allows remote attackers to conduct directory traversal attacks and read arbitrary files via a `..` (dot dot) in a request parameter.

Related Guidelines

SEI CERT C Coding Standard	STR02-C. Sanitize data passed to complex subsystems
SEI CERT C++ Coding Standard	VOID STR02-CPP. Sanitize data passed to complex subsystems
SEI CERT Perl Coding Standard	IDS33-PL. Sanitize untrusted data passed across a trust boundary
ISO/IEC TR 24772:2013	Injection [RST]
MITRE CWE	CWE-116, Improper Encoding or Escaping of Output

Bibliography

[OWASP 2005]	A Guide to Building Secure Web Applications and Web Services
[OWASP 2007]	OWASP Top 10 for Java EE
[OWASP 2008]	Testing for XML Injection (OWASP-DV-008)
[Seacord 2015]	IDS00-J. Prevent SQL Injection LiveLesson
[W3C 2008]	Section 4.4.3, "Included If Validating"