STR04-J. Use compatible character encodings when communicating string data between JVMs

A *character encoding* or *charset* specifies the binary representation of the coded character set. Every instance of the Java Virtual Machine (JVM) has a default charset, which may or may not be one of the standard charsets. The default charset is determined during virtual-machine startup and typically depends on the locale and charset being used by the underlying operating system [API 2014]. The default character encoding can be set at startup, for example:

```
java -Dfile.encoding=UTF-8 ... com.x.Main
```

The available encodings are listed in the *Supported Encodings* document [Encodings 2014]. In the absence of an explicitly specified encoding, conversions use the system default encoding. Compatible encodings must be used when characters are output as an array of bytes then input by another JVM and subsequently converted back to characters.

According to the Java API [API 2014] for the `String` class:

> *The behavior of this constructor when the given bytes are not valid in the given charset is unspecified.*

Disagreement over character encodings can result in data corruption.

Noncompliant Code Example

This noncompliant code example reads a byte array and converts it into a `String` using the platform's default character encoding. If the byte array does not represent a string, or if it represents a string that was encoded using other than the default encoding, the resulting `String` is likely to be incorrect. The behavior resulting from malformed-input and unmappable-character errors is unspecified.

```
FileInputStream fis = null;
try {
  fis = new FileInputStream("SomeFile");
  DataInputStream dis = new DataInputStream(fis);
  byte[] data = new byte[1024];
  dis.readFully(data);
  String result = new String(data);
} catch (IOException x) {
  // Handle error
} finally {
  if (fis != null) {
    try {
      fis.close();
    } catch (IOException x) {
      // Forward to handler
    }
  }
}
```

Compliant Solution

This compliant solution explicitly specifies the character encoding used to create the string (in this example, `UTF-16LE`) as the second argument to the `String` constructor. The LE form of UTF-16 uses little-endian byte serialization (least significant byte first). Provided that the character data was encoded in `UTF-16LE`, it will decode correctly.

```
FileInputStream fis = null;
try {
  fis = new FileInputStream("SomeFile");
  DataInputStream dis = new DataInputStream(fis);
  byte[] data = new byte[1024];
  dis.readFully(data);
  String result = new String(data, "UTF-16LE");
} catch (IOException x) {
  // Handle error
} finally {
  if (fis != null) {
    try {
      fis.close();
    } catch (IOException x) {
      // Forward to handler
    }
  }
}
```

An explicit character encoding may be omitted on the receiving side when the data is produced by a Java application that uses the same platform and default character encoding and is communicated over a secure communication channel (see MSC00-J. Use SSLSocket rather than Socket for secure data exchange for more information).

Risk Assessment

Using incompatible encodings when communicating string data between JVMs can result in corrupted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR04-J	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Sound automated detection of this vulnerability is not feasible.

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
SonarQube	6.7	**S1943**	Classes and methods that rely on the default system encoding should not be used

Bibliography

[API 2014]	
[Encodings 2014]	
[Seacord 2015]	STR04-J. Use compatible character encodings when communicating string data between JVMs LiveLesson