IDS08-J. Sanitize untrusted data included in a regular expression

Regular expressions (regex) are widely used to match strings of text. For example, the POSIX `grep` utility supports regular expressions for finding patterns in the specified text. For introductory information on regular expressions, see the Java Tutorials [Java Tutorials]. The `java.util.regex` package provides the `Pattern` class that encapsulates a compiled representation of a regular expression and the `Matcher` class, which is an engine that uses a `Pattern` to perform matching operations on a `CharSequence`.

Java's powerful regex facilities must be protected from misuse. An attacker may supply a malicious input that modifies the original regular expression in such a way that the regex fails to comply with the program's specification. This attack vector, called a *regex injection*, might affect control flow, cause information leaks, or result in denial-of-service (DoS) vulnerabilities.

Certain constructs and properties of Java regular expressions are susceptible to exploitation:

- **Matching flags:** Untrusted inputs may override matching options that may or may not have been passed to the `Pattern.compile()` method.
- **Greediness:** An untrusted input may attempt to inject a regex that changes the original regex to match as much of the string as possible, exposing sensitive information.
- **Grouping:** The programmer can enclose parts of a regular expression in parentheses to perform some common action on the group. An attacker may be able to change the groupings by supplying untrusted input.

Untrusted input should be sanitized before use to prevent regex injection. When the user must specify a regex as input, care must be taken to ensure that the original regex cannot be modified without restriction. Whitelisting characters (such as letters and digits) before delivering the user-supplied string to the regex parser is a good input sanitization strategy. A programmer must provide only a very limited subset of regular expression functionality to the user to minimize any chance of misuse.

Regex Injection Example

Suppose a system log file contains messages output by various system processes. Some processes produce public messages, and some processes produce sensitive messages marked "private." Here is an example log file:

```
10:47:03 private[423] Successful logout  name: usr1 ssn: 111223333
10:47:04 public[48964] Failed to resolve network service
10:47:04 public[1] (public.message[49367]) Exited with exit code: 255
10:47:43 private[423] Successful login  name: usr2 ssn: 444556666
10:48:08 public[48964] Backup failed with error: 19
```

A user wishes to search the log file for interesting messages but must be prevented from seeing the private messages. A program might accomplish this by permitting the user to provide search text that becomes part of the following regex:

```
(.*? +public\[\d+\] +.*<SEARCHTEXT>.*)
```

However, if an attacker can substitute any string for `<SEARCHTEXT>`, he can perform a regex injection with the following text:

```
.*)|(.*
```

When injected into the regex, the regex becomes

```
(.*? +public\[\d+\] +.*.*)|(.*.*)
```

This regex will match any line in the log file, including the private ones.

Noncompliant Code Example

This noncompliant code example searches a log file using search terms from an untrusted user:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.CharBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LogSearch {
        public static void FindLogEntry(String search) {
                // Construct regex dynamically from user string
                String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";
                Pattern searchPattern = Pattern.compile(regex);
                try (FileInputStream fis = new FileInputStream("log.txt")) {
                        FileChannel channel = fis.getChannel();
                        // Get the file's size and map it into memory
                        long size = channel.size();
                        final MappedByteBuffer mappedBuffer = channel.map(
                                        FileChannel.MapMode.READ_ONLY, 0, size);
                        Charset charset = Charset.forName("ISO-8859-15");
                        final CharsetDecoder decoder = charset.newDecoder();
                        // Read file into char buffer
                        CharBuffer log = decoder.decode(mappedBuffer);
                        Matcher logMatcher = searchPattern.matcher(log);
                        while (logMatcher.find()) {
                                String match = logMatcher.group();
                                if (!match.isEmpty()) {
                                        System.out.println(match);
                                }
                        }
                } catch (IOException ex) {
                        System.err.println("thrown exception: " + ex.toString());
                        Throwable[] suppressed = ex.getSuppressed();
                        for (int i = 0; i < suppressed.length; i++) {
                                System.err.println("suppressed exception: "
                                                + suppressed[i].toString());
                        }
                }
                return;
        }
```

This code permits an attacker to perform a regex injection.

Compliant Solution (Whitelisting)

This compliant solution sanitizes the search terms at the beginning of the `FindLogEntry()`, filtering out nonalphanumeric characters (except space and single quote):

```
        public static void FindLogEntry(String search) {
                // Sanitize search string
                StringBuilder sb = new StringBuilder(search.length());
                for (int i = 0; i < search.length(); ++i) {
                        char ch = search.charAt(i);
                        if (Character.isLetterOrDigit(ch) || ch == ' ' || ch == '\'') {
                                sb.append(ch);
                        }
                }
                search = sb.toString();

                // Construct regex dynamically from user string
                String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";
        // ...
    }
```

This solution prevents regex injection but also restricts search terms. For example, a user may no longer search for "`name =`" because nonalphanumeric characters are removed from the search term.

Compliant Solution (`Pattern.quote()`)

This compliant solution sanitizes the search terms by using `Pattern.quote()` to escape any malicious characters in the search string. Unlike the previous compliant solution, a search string using punctuation characters, such as "`name =" is permitted.`

```
        public static void FindLogEntry(String search) {
                // Sanitize search string
        search = Pattern.quote(search);
                // Construct regex dynamically from user string
                String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";
        // ...
    }
```

The `Matcher.quoteReplacement()` method can be used to escape strings used when doing regex substitution.

Compliant Solution

Another method of mitigating this vulnerability is to filter out the sensitive information prior to matching. Such a solution would require the filtering to be done every time the log file is periodically refreshed, incurring extra complexity and a performance penalty. Sensitive information may still be exposed if the log format changes but the class is not also refactored to accommodate these changes.

Risk Assessment

Failing to sanitize untrusted data included as part of a regular expression can result in the disclosure of sensitive information.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS08-J	Medium	Unlikely	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
SonarQube	6.7	**S2631**	Regular expressions should not be vulnerable to Denial of Service attacks

Related Guidelines

MITRE CWE	CWE-625, Permissive Regular Expression

Bibliography

[CVE 05]	CVE-2005-1949
[Java Tutorials]	Regular Expressions
[Seacord 2015]	IDS08-J. Sanitize untrusted data passed to a regex LiveLesson