ERR01-J. Do not allow exceptions to expose sensitive information

Failure to filter sensitive information when propagating exceptions often results in information leaks that can assist an attacker's efforts to develop further exploits. An attacker may craft input arguments to expose internal structures and mechanisms of the application. Both the exception message text and the type of an exception can leak information. For example, the `FileNotFoundException` message reveals information about the file system layout, and the exception type reveals the absence of the requested file.

This rule applies to server-side applications as well as to clients. Attackers can glean sensitive information not only from vulnerable web servers but also from victims who use vulnerable web browsers. In 2004, Schönefeld discovered an exploit for the Opera v7.54 web browser in which an attacker could use the `sun.security.krb5.Credentials` class in an applet as an oracle to "retrieve the name of the currently logged in user and parse his home directory from the information which is provided by the thrown `java.security.AccessControlException`" [Schönefeld 2004].

All exceptions reveal information that can assist an attacker's efforts to carry out a denial of service (DoS) against the system. Consequently, programs must filter both exception messages and exception types that can propagate across trust boundaries. The following table lists several problematic exceptions.

Exception Name	Description of Information Leak or Threat
`java.io.FileNotFoundException`	Underlying file system structure, user name enumeration
`java.sql.SQLException`	Database structure, user name enumeration
`java.net.BindException`	Enumeration of open ports when untrusted client can choose server port
`java.util.ConcurrentModificationException`	May provide information about thread-unsafe code
`javax.naming.InsufficientResourcesException`	Insufficient server resources (may aid DoS)
`java.util.MissingResourceException`	Resource enumeration
`java.util.jar.JarException`	Underlying file system structure
`java.security.acl.NotOwnerException`	Owner enumeration
`java.lang.OutOfMemoryError`	DoS
`java.lang.StackOverflowError`	DoS

Printing the stack trace can also result in unintentionally leaking information about the structure and state of the process to an attacker. When a Java program that is run within a console terminates because of an uncaught exception, the exception's message and stack trace are displayed on the console; the stack trace may itself contain sensitive information about the program's internal structure. Consequently, any program that may be run on a console accessible to an untrusted user must never abort due to an uncaught exception.

Noncompliant Code Example (Leaks from Exception Message and Type)

In this noncompliant code example, the program must read a file supplied by the user, but the contents and layout of the file system are sensitive. The program accepts a file name as an input argument but fails to prevent any resulting exceptions from being presented to the user.

```
class ExceptionExample {
  public static void main(String[] args) throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA") + args[0]);
  }
}
```

When a requested file is absent, the `FileInputStream` constructor throws a `FileNotFoundException`, allowing an attacker to reconstruct the underlying file system by repeatedly passing fictitious path names to the program.

Noncompliant Code Example (Wrapping and Rethrowing Sensitive Exception)

This noncompliant code example logs the exception and then wraps it in a more general exception before rethrowing it:

```
try {
  FileInputStream fis =
      new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {
  // Log the exception
  throw new IOException("Unable to retrieve file", e);
}
```

Even when the logged exception is not accessible to the user, the original exception is still informative and can be used by an attacker to discover sensitive information about the file system layout.
Note that this example also violates FIO04-J. Release resources when they are no longer needed, as it fails to close the input stream in a `finally` block. Subsequent code examples also omit this `finally` block for brevity.

Noncompliant Code Example (Sanitized Exception)

This noncompliant code example logs the exception and throws a custom exception that does not wrap the `FileNotFoundException`:

```
class SecurityIOException extends IOException {/* ... */};

try {
  FileInputStream fis =
      new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {
  // Log the exception
  throw new SecurityIOException();
}
```

Although this exception is less likely than the previous noncompliant code examples to leak useful information, it still reveals that the specified file cannot be read. More specifically, the program reacts differently to nonexistent file paths than it does to valid ones, and an attacker can still infer sensitive information about the file system from this program's behavior. Failure to restrict user input leaves the system vulnerable to a brute-force attack in which the attacker discovers valid file names by issuing queries that collectively cover the space of possible file names. File names that cause the program to return the sanitized exception indicate nonexistent files, whereas file names that do not return exceptions reveal existing files.

Compliant Solution (Security Policy)

This compliant solution implements the policy that only files that live in `c:\homepath` may be opened by the user and that the user is not allowed to discover anything about files outside this directory. The solution issues a terse error message when the file cannot be opened or the file does not live in the proper directory. Any information about files outside `c:\homepath` is concealed.

The compliant solution also uses the `File.getCanonicalFile()` method to canonicalize the file to simplify subsequent path name comparisons (see FIO16-J. Canonicalize path names before validating them for more information).

```
class ExceptionExample {
  public static void main(String[] args) {

    File file = null;
    try {
      file = new File(System.getenv("APPDATA") +
             args[0]).getCanonicalFile();
      if (!file.getPath().startsWith("c:\\homepath")) {
        System.out.println("Invalid file");
        return;
      }
    } catch (IOException x) {
      System.out.println("Invalid file");
      return;
    }

    try {
      FileInputStream fis = new FileInputStream(file);
    } catch (FileNotFoundException x) {
      System.out.println("Invalid file");
      return;
    }
  }
}
```

Compliant Solution (Restricted Input)

This compliant solution operates under the policy that only `c:\homepath\file1` and `c:\homepath\file2` are permitted to be opened by the user. It also catches `Throwable`, as permitted by exception ERR08-J-EX2 (see ERR08-J. Do not catch NullPointerException or any of its ancestors). It uses the `MyExceptionReporter` class described in ERR00-J. Do not suppress or ignore checked exceptions, which filters sensitive information from any resulting exceptions.

```
class ExceptionExample {
  public static void main(String[] args) {
    FileInputStream fis = null;
    try {
      switch(Integer.valueOf(args[0])) {
        case 1:
          fis = new FileInputStream("c:\\homepath\\file1");
          break;
        case 2:
          fis = new FileInputStream("c:\\homepath\\file2");
          break;
        //...
        default:
          System.out.println("Invalid option");
          break;
      }
    } catch (Throwable t) {
      MyExceptionReporter.report(t); // Sanitize
    }
  }
}
```

Compliant solutions must ensure that security exceptions such as `java.security.AccessControlException` and `java.lang.SecurityException` continue to be logged and sanitized appropriately (see ERR02-J. Prevent exceptions while logging data for additional information). The `MyExceptionReporter` class from ERR00-J. Do not suppress or ignore checked exceptions demonstrates an acceptable approach for this logging and sanitization.

For scalability, the `switch` statement should be replaced with some sort of mapping from integers to valid file names or at least an enum type representing valid files.

Risk Assessment

Exceptions may inadvertently reveal sensitive information unless care is taken to limit the information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR01-J	Medium	Probable	High	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**SECURITY.WSC.ACPST, SERVLET.CETS, SECURITY.ESD.ACW**	Implemented
SonarQube	6.7	**S1989**	Exceptions should not be thrown from servlet methods

Related Vulnerabilities

CVE-2009-2897 describes several cross-site scripting (XSS) vulnerabilities in several versions of SpringSource Hyperic HQ. These vulnerabilities allow remote attackers to inject arbitrary web script or HTML via invalid values for numerical parameters. They are demonstrated by an uncaught `java.lang.NumberFormatException` exception resulting from entering several invalid numeric parameters to the web interface.

CVE-2015-2080 describes a vulnerability in the Jetty web server, versions 9.2.3 to 9.2.8, where an illegal character passed in an HTML request causes the server to respond with an error message containing the text with the illegal character. But this error message can also contain sensitive information, such as cookies from previous web requests.

Related Guidelines

SEI CERT C++ Coding Standard	VOID ERR12-CPP. Do not allow exceptions to transmit sensitive information
MITRE CWE	CWE-209, Information Exposure through an Error Message CWE-497, Exposure of System Data to an Unauthorized Control Sphere CWE-600, Uncaught Exception in Servlet

Bibliography

[Gong 2003]	9.1, Security Exceptions
\[Gotham 2015\]	JetLeak Vulnerability: Remote Leakage Of Shared Buffers In Jetty Web Server
[Schönefeld 2004]	