**Unreleased Resource: Streams Development Mitigation SOP**

When a program fails to release a system resource, it can potentially cause the entire system to crash. Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool. Two of the most common cause for Unreleased Resource Streams is improper handling of error conditions as well as other exceptional circumstances and confusion over which part of the part is responsible for releasing the resource.

**Classic Example**

private void processFile(String fName) throws FileNotFoundException, IOException {

FileInputStream fis = new FileInputStream(fName);

int sz;

byte[] byteArray = new byte[BLOCK\_SIZE];

while((sz = fis.read(byteArray)) != -1){

processBytes(byteArray, sz);

}

}

**Example**

InputStream inputStreamforFileType = null;

InputStream inputStreamForTransform = null;

try {

inputStreamForTransfrom = dataHandler.getInputStream();

LinearizeTranformation transformation = new LinearizeTransformation();

InputStream inputStream =

documentConverterService.transformDocument(transformation,

inputStreamForTransform);

InputStreamDataSource inputStreamDataSource = new

InputStreamDataSource(inputStream, “\*/\*”);

Document convertedDataHandler = new

DataHandler(inputStreamDataSource);

document.setContent(convertedDataHandler);

} catch(IOException e){

Throw new SystemFaultException(“Unable to linearize document.”, e);

} catch(UnknownFileTypeException e) {

Throw new SystemFaultException(“Unknown file type for conversion.”, e);

}

**Defense Against Unreleased Resource: Streams**

Never rely on any method that invokes the garage collector, such as finalize(), since the behavior of the garage collector is unpredictable. The garage collector isn’t explicitly called unless the JVM is low on memory and there is no guarantee that a method that will invoke the garage collector in a timely manner. There can be instances of the garage collector has a large number of resources to reclaim which can lead to “bursty” performances and reduce the effectiveness of the system throughput.

**Defense Example *(fixing Classic example above)***

private void processFile(String fName) throws FileNotFoundException, IOException {

**FileInputStream fis;**

**try {**

fis = new FileInputStream(fName);

int sz;

byte[] byteArray = new byte[BLOCK\_SIZE];

while((sz = fis.read(byteArray)) != -1){

processBytes(byteArray, sz);

}

**} finally {**

**if(fis != null){**

**safeClose(fis);**

**}**

**}**

}

**public static void safeClose(FileInputStream fis){**

**if(fis != null){**

**try {**

**fis.close();**

**} catch (IOException e){**

**log(e);**

**}**

**}**

**}**

**Specific Explanation**

The code snippet below demonstrates the recommended way to release a resource after usage. Regardless if an exception is thrown the resources are explicitly released. This solution checks to see if the object has been initialized to null to prevent further errors from happening.

Another solution to close this stream would be to use a try-with-resources statement. This forces a resource to be closed after the program is finished with it. A resource is given below that contains examples on the implementation of a try-with-resources statement.

**Recommendation**

} catch(UnknownFileTypeException e) {

Throw new SystemFaultException(“Unknown file type for conversion.”, e);

} **finally {**

**try {**

**if(inputStreamForFileType != null){**

**inputStreamForFileType.close();**

**}**

**if(inputStreamForTransform.close()){**

**inputStreamForTransform.close();**

**}**

**} catch(IOException e){**

**throw new SystemFaultException(“Unable to close input stream.”,**

**e);**

**}**

return document;

}

For all system resources initialized into the application should be explicitly released in all circumstances. Resources need to be released as soon as they are no longer needed. Also, check to see if the resource has been properly initialized, not set to null, before executing one of its methods.

**References**

1. [Try-with-resources tutorial](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html)