IDS04-J. Safely extract files from ZipInputStream

Java provides the `java.util.zip` package for zip-compatible data compression. It provides classes that enable you to read, create, and modify ZIP and GZIP file formats.

A number of security concerns must be considered when extracting file entries from a ZIP file using `java.util.zip.ZipInputStream`. File names may contain path traversal information that may cause them to be extracted outside of the intended directory, frequently with the purpose of overwriting existing system files. *Directory traversal* or *path equivalence* vulnerabilities can be eliminated by canonicalizing the path name, in accordance with FIO16-J. Canonicalize path names before validating them, and then validating the location before extraction.

A second issue is that the extraction process can cause excessive consumption of system resources, possibly resulting in a denial-of-service attack when resource usage is disproportionately large compared to the input data. The zip algorithm can produce very large compression ratios [Mahmoud 2002]. For example, a file consisting of alternating lines of *a* characters and *b* characters can achieve a compression ratio of more than 200 to 1. Even higher compression ratios can be obtained using input data that is targeted to the compression algorithm. This permits the existence of *zip bombs* in which a small ZIP or GZIP file consumes excessive resources when uncompressed. An example of a zip bomb is the file *42.zip*, which is a zip file consisting of 42 kilobytes of compressed data, containing five layers of nested zip files in sets of 16, each bottom layer archive containing a 4.3 gigabyte (4 294 967 295 bytes; ~ 3.99 GiB) file for a total of 4.5 petabytes (4 503 599 626 321 920 bytes; ~ 3.99 PiB) of uncompressed data. Zip bombs often rely on repetition of identical files to achieve their extreme compression ratios. Programs must either limit the traversal of such files or refuse to extract data beyond a certain limit. The actual limit depends on the capabilities of the platform and expected usage.

Noncompliant Code Example

This noncompliant code fails to validate the name of the file that is being unzipped. It passes the name directly to the constructor of `FileOutputStream`. It also fails to check the resource consumption of the file that is being unzipped. It permits the operation to run to completion or until local resources are exhausted.

```
static final int BUFFER = 512;
// ...

public final void unzip(String filename) throws java.io.IOException{
  FileInputStream fis = new FileInputStream(filename);
  ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
  ZipEntry entry;
  try {
    while ((entry = zis.getNextEntry()) != null) {
      System.out.println("Extracting: " + entry);
      int count;
      byte data[] = new byte[BUFFER];
      // Write the files to the disk
      FileOutputStream fos = new FileOutputStream(entry.getName());
      BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
      while ((count = zis.read(data, 0, BUFFER)) != -1) {
        dest.write(data, 0, count);
      }
      dest.flush();
      dest.close();
      zis.closeEntry();
    }
  } finally {
    zis.close();
  }
}
```

Noncompliant Code Example (`getSize()`)

This noncompliant code attempts to overcome the problem by calling the method `ZipEntry.getSize()` to check the uncompressed file size before uncompressing it. Unfortunately, a malicious attacker can forge the field in the ZIP file that purports to show the uncompressed size of the file, so the value returned by `getSize()` is unreliable, and local resources may still be exhausted.

```
static final int BUFFER = 512;
static final int TOOBIG = 0x6400000; // 100MB
// ...

public final void unzip(String filename) throws java.io.IOException{
  FileInputStream fis = new FileInputStream(filename);
  ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
  ZipEntry entry;
  try {
    while ((entry = zis.getNextEntry()) != null) {
      System.out.println("Extracting: " + entry);
      int count;
      byte data[] = new byte[BUFFER];
      // Write the files to the disk, but only if the file is not insanely big
      if (entry.getSize() > TOOBIG ) {
         throw new IllegalStateException("File to be unzipped is huge.");
      }
      if (entry.getSize() == -1) {
         throw new IllegalStateException("File to be unzipped might be huge.");
      }
      FileOutputStream fos = new FileOutputStream(entry.getName());
      BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
      while ((count = zis.read(data, 0, BUFFER)) != -1) {
        dest.write(data, 0, count);
      }
      dest.flush();
      dest.close();
      zis.closeEntry();
    }
  } finally {
    zis.close();
  }
}
```

Acknowledgement: The vulnerability in this code was pointed out by Giancarlo Pellegrino, researcher at the Technical University of Darmstadt in Germany, and Davide Balzarotti, faculty member of EURECOM in France.

Compliant Solution

In this compliant solution, the code validates the name of each entry before extracting the entry. If the name is invalid, the entire extraction is aborted. However, a compliant solution could also skip only that entry and continue the extraction process, or it could even extract the entry to some safe location.

Furthermore, the code inside the `while` loop tracks the uncompressed file size of each entry in a zip archive while extracting the entry. It throws an exception if the entry being extracted is too large—about 100MB in this case. We do not use the `ZipEntry.getSize()` method because the value it reports is not reliable. Finally, the code also counts the number of file entries in the archive and throws an exception if there are more than 1024 entries.

```
static final int BUFFER = 512;
static final long TOOBIG = 0x6400000; // Max size of unzipped data, 100MB
static final int TOOMANY = 1024;      // Max number of files
// ...

private String validateFilename(String filename, String intendedDir)
      throws java.io.IOException {
  File f = new File(filename);
  String canonicalPath = f.getCanonicalPath();

  File iD = new File(intendedDir);
  String canonicalID = iD.getCanonicalPath();

  if (canonicalPath.startsWith(canonicalID)) {
    return canonicalPath;
  } else {
    throw new IllegalStateException("File is outside extraction target directory.");
  }
}

public final void unzip(String filename) throws java.io.IOException {
  FileInputStream fis = new FileInputStream(filename);
  ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
  ZipEntry entry;
  int entries = 0;
  long total = 0;
  try {
    while ((entry = zis.getNextEntry()) != null) {
      System.out.println("Extracting: " + entry);
      int count;
      byte data[] = new byte[BUFFER];
      // Write the files to the disk, but ensure that the filename is valid,
      // and that the file is not insanely big
      String name = validateFilename(entry.getName(), ".");
      if (entry.isDirectory()) {
        System.out.println("Creating directory " + name);
        new File(name).mkdir();
        continue;
      }
      FileOutputStream fos = new FileOutputStream(name);
      BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
      while (total + BUFFER <= TOOBIG && (count = zis.read(data, 0, BUFFER)) != -1) {
        dest.write(data, 0, count);
        total += count;
      }
      dest.flush();
      dest.close();
      zis.closeEntry();
      entries++;
      if (entries > TOOMANY) {
        throw new IllegalStateException("Too many files to unzip.");
      }
      if (total + BUFFER > TOOBIG) {
        throw new IllegalStateException("File being unzipped is too big.");
      }
    }
  } finally {
    zis.close();
  }
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS04-J	Low	Probable	High	P2	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	**S5042**	Expanding archive files is security-sensitive

Related Guidelines

MITRE CWE	CWE-409, Improper Handling of Highly Compressed Data (Data Amplification)
Secure Coding Guidelines for Java SE, Version 5.0	Guideline 1-1 / DOS-1: Beware of activities that may use disproportionate resources

Related Vulnerabilities

Vulnerability	Description
Zip Slip	Zip Slip is a form of directory traversal that can be exploited by extracting files from an archive. It is caused by a failure to validate path names of the files within an archive which can lead to files being extracted outside of the intended directory and overwriting existing system files. An attacker can exploit this vulnerability to overwrite executable files to achieve remote command execution on a victim's machine. Snyk responsibly disclosed the vulnerability before public disclosure on June 5th 2018. Their blog post and technical paper detailing the vulnerability can be found at https://snyk.io/blog/zip-slip-vulnerability/.

Android Implementation Details

Although not directly a violation of this rule, the Android Master Key vulnerability (insecure use of `ZipEntry`) is related to this rule. Another attack vector, found by a researcher in China, is also related to this rule.

Bibliography

[Mahmoud 2002]	Compressing and Decompressing Data Using Java APIs
[Seacord 2015]	IDS04-J. Safely extract files from ZipInputStream LiveLesson