IDS56-J. Prevent arbitrary file upload

Java applications, including web applications, that accept file uploads must ensure that an attacker cannot upload or transfer malicious files. If a restricted file containing code is executed by the target system, it can compromise application-layer defenses. For example, an application that permits HTML files to be uploaded could allow malicious code to be executed—an attacker can submit a valid HTML file with a cross-site scripting (XSS) payload that will execute in the absence of an output-escaping routine. For this reason, many applications restrict the type of files that can be uploaded.

It may also be possible to upload files with dangerous extensions such as .exe and .sh that could cause arbitrary code execution on server-side applications. An application that restricts only the Content-Type field in the HTTP header could be vulnerable to such an attack.

To support file upload, a typical Java Server Pages (JSP) page consists of code such as the following:

```
<s:form action="doUpload" method="POST" enctype="multipart/form-data">
  <s:file name="uploadFile" label="Choose File" size="40" />
  <s:submit value="Upload" name="submit" />
</s:form>
```

Many Java enterprise frameworks provide configuration settings intended to be used as a defense against arbitrary file upload. Unfortunately, most of them fail to provide adequate protection. Mitigation of this vulnerability involves checking file size, content type, and file contents, among other metadata attributes.

Noncompliant Code Example

This noncompliant code example shows XML code from the upload action of a Struts 2 application. The interceptor code is responsible for allowing file uploads.

```
<action name="doUpload" class="com.example.UploadAction">
  <interceptor-ref name="upload">
    <param name="maximumSize"> 10240 </param>
    <param name="allowedTypes"> text/plain,image/JPEG,text/html </param>
  </interceptor-ref>
</action>
```

The code for file upload appears in the `UploadAction` class:

```
public class UploadAction extends ActionSupport {
  private File uploadedFile;
  // setter and getter for uploadedFile

  public String execute() {
    try {
      // File path and file name are hardcoded for illustration
      File fileToCreate = new File("filepath", "filename");
      // Copy temporary file content to this file
      FileUtils.copyFile(uploadedFile, fileToCreate);
      return "SUCCESS";
    } catch (Throwable e) {
      addActionError(e.getMessage());
      return "ERROR";
    }
  }
}
```

The value of the parameter type `maximumSize` ensures that a particular `Action` cannot receive a very large file. The `allowedTypes` parameter defines the type of files that are accepted. However, this approach fails to ensure that the uploaded file conforms to the security requirements because interceptor checks can be trivially bypassed. If an attacker were to use a proxy tool to change the content type in the raw HTTP request in transit, the framework would fail to prevent the file's upload. Consequently, an attacker could upload a malicious file that has a .exe extension, for example.

Compliant Solution

The file upload must succeed only when the content type matches the actual content of the file. For example, a file with an image header must contain only an image and must not contain executable code. This compliant solution uses the Apache Tika library [Apache 2013] to detect and extract metadata and structured text content from documents using existing parser libraries. The `checkMetaData()` method must be called before invoking code in `execute()` that is responsible for uploading the file.

```
public class UploadAction extends ActionSupport {
  private File uploadedFile;
  // setter and getter for uploadedFile

  public String execute() {
    try {
      // File path and file name are hardcoded for illustration
      File fileToCreate = new File("filepath", "filename");

      boolean textPlain = checkMetaData(uploadedFile, "text/plain");
      boolean img = checkMetaData(uploadedFile, "image/JPEG");
      boolean textHtml = checkMetaData(uploadedFile, "text/html");

          if (!textPlain && !img && !textHtml) {
            return "ERROR";
      }

          // Copy temporary file content to this file
      FileUtils.copyFile(uploadedFile, fileToCreate);
      return "SUCCESS";
    } catch (Throwable e) {
      addActionError(e.getMessage());
      return "ERROR";
    }
  }

  public static boolean checkMetaData(
    File f, String getContentType) {
    try (InputStream is = new FileInputStream(f)) {
      ContentHandler contenthandler = new BodyContentHandler();
      Metadata metadata = new Metadata();
      metadata.set(Metadata.RESOURCE_NAME_KEY, f.getName());
      Parser parser = new AutoDetectParser();
      try {
        parser.parse(is, contenthandler, metadata, new ParseContext());
      } catch (SAXException | TikaException e) {
        // Handle error
        return false;
      }

      if (metadata.get(Metadata.CONTENT_TYPE).equalsIgnoreCase(getContentType)) {
        return true;
      } else {
        return false;
      }
    } catch (IOException e) {
      // Handle error
      return false;
    }
  }
}
```

The `AutoDetectParser` selects the best available parser on the basis of the content type of the file to be parsed.

Applicability

An arbitrary file upload vulnerability could result in privilege escalation and the execution of arbitrary code.

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)

Bibliography