Project 1 Code Vulnerabilities

Carnegie Mellon University's Software Engineering Institute developed a set of coding standards called the SEI CERT Oracle Coding Standard for Java. These rules and recommendations were designed to eliminate common vulnerabilities found inside Java code. For this project, we used this set of standards to identify and fix vulnerabilities that we discovered in this sample code, which takes an input file from a runtime configuration and prints out a list of emails contained inside the file.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CSC245_Project2 {

   public static void main(String[] args) {
       // Read the filename from the command line argument
       // FIO16-J Pathname not Canonicalized
       // IDS01-J Filename is not normalized.
       // IDS50-J Filename is not validated for malicious characters.
       String filename = args[0];
       BufferedReader inputStream = null;

       String fileLine;
       try {
           inputStream = new BufferedReader(new FileReader(filename));

           System.out.println("Email Addresses:");
           // Read one Line using BufferedReader
           while ((fileLine = inputStream.readLine()) != null) {
               // IDS01-J String not canonicalized before validation.
               // IDS51-J Strings are not sanitized or validated.
               System.out.println(fileLine);
           }
       } catch (IOException io) {
           // ERROR01-J, the io.getMessage() displays too much information
           System.out.println("File IO exception" + io.getMessage());
       } finally {
           // Need another catch for closing
           // the streams
           try {
               if (inputStream != null) {
                   inputStream.close();
               }
           } catch (IOException io) {
               // ERROR01-J, the io.getMessage() displays too much information
```

```
                System.out.println("Issue closing the Files" + io.getMessage());
            }
        }
    }
    // IDS08-J was not found to be a threat, the emails do not need to be
sanitized for creation of a regex.
}
```

The vulnerabilities discovered in this code are as follows:

- ERROR01-J
- FIO16-J
- IDS01-J
- IDS50-J
- IDS51-J

When we run this sample code, the output is this. (02-17-2026 at 3:20 PM)

```
Email Addresses:
doug@arapahoe.edu
nina@arapahoe.edu
tracy@arapahoe.edu
TESTING!@#$%@arapahoe.edu
jodi@arapahoe.edu
kristi@arapahoe.edu
TestText@#$..edu
TestingAgain@@..comdf
jessie@arapahoe.edu
laura%@arapahoe.edu
tina@arapahoe.edu
todd@arapahoe.edu
douglas.lundin@student.arapahoe.edu
<script>test</script>
ed@arapahoe.edu

Process finished with exit code 0
```

ERRO1-J & FIO16-J: Matt

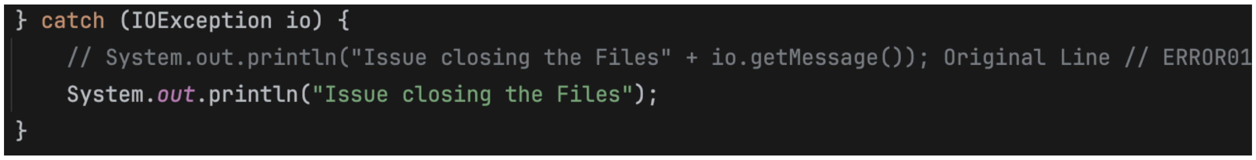
Core Idea: NEVER let raw exceptions, especially those containing system details, reach UNTRUSTED users!!

Requires developers to **sanitize, filter, and control** all **exception** information that could reach untrusted users, ensuring that neither messages, types, nor stack traces reveal internal details that attackers can exploit.

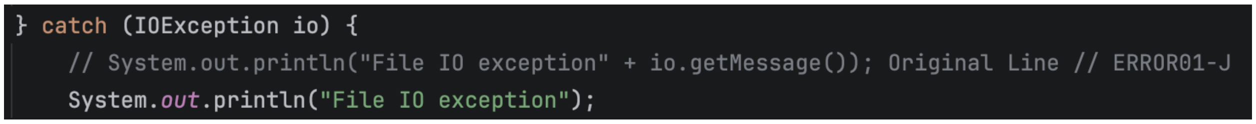
Exception Examples:

- `FileNotFoundException` can reveal file paths, directory layouts, or usernames.
- `SQLException` can reveal the database structure.
- `BindException` can reveal which ports are open.

Code Errors:



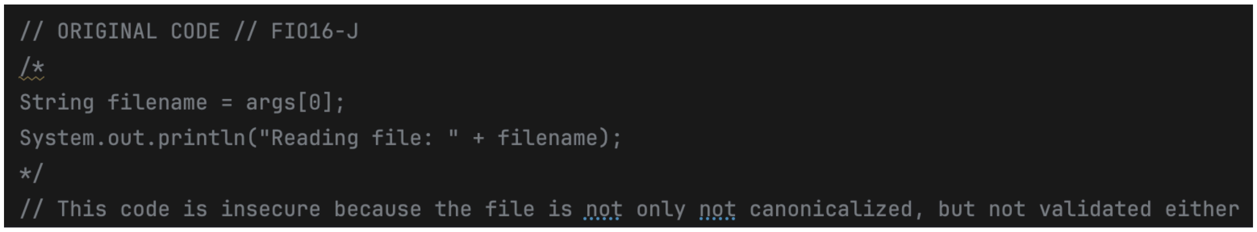
```
} catch (IOException io) {
    // System.out.println("Issue closing the Files" + io.getMessage()); Original Line // ERROR01
    System.out.println("Issue closing the Files");
}
```



```
} catch (IOException io) {
    // System.out.println("File IO exception" + io.getMessage()); Original Line // ERROR01-J
    System.out.println("File IO exception");
```

The inclusion of io.getMessage() is noncompliant and reveals the file name in the output

Code Solution (ERR01-J AND FIO16-J Compliant):



```
// ORIGINAL CODE // FIO16-J
/*
String filename = args[0];
System.out.println("Reading file: " + filename);
*/
// This code is insecure because the file is not only not canonicalized, but not validated either
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class CSC245_Project2 {

   public static void main(String[] args) {

       // Check argument
       if (args.length == 0) {
           System.out.println("Invalid file.");
           return;
```

```
        }

        try {
            // Define base directory (allowed directory)
            File baseDir = new File("C:\\homepath").getCanonicalFile();

            // Build requested file path
            File requestedFile = new File(baseDir, args[0]).getCanonicalFile();

            // Ensure file stays inside allowed directory
            if (!requestedFile.getPath().startsWith(baseDir.getPath())) {
                System.out.println("Invalid file.");
                return;
            }

            // Open and read file safely
            try (BufferedReader reader =
                         new BufferedReader(new FileReader(requestedFile))) {

                System.out.println("Email Addresses:");

                String line;
                while ((line = reader.readLine()) != null) {
                    System.out.println(line);
                }
            }

        } catch (IOException e) {
            // Generic error message only (ERR01-J compliant)
            System.out.println("Invalid file.");
        }
    }
}
```

This code:

- Does NOT print `e.getMessage()`
- Does NOT print stack traces
- Does NOT reveal file paths
- Returns the same message for all failures
- Validates the file path and name

Even if:

- File not found

- Permission denied
- Path traversal attempt
- Canonicalization fails

User only sees:

```
Invalid file.
```

`get.CanonicalFile()` prevents attackers from inputting something like:
`..\..\Windows\System32\drivers\etc\hosts`
> Resolves (..)
> Resolves Symbolic links
> Produces the true absolute path

IDS01-J and IDS50-J: Claire Kayton

IDS01-J is the rule that dictates that strings need to be normalized before validation. Normalizing a string involves breaking the string down into its truest form. Many characters in a string can look the same to us, but they can have different binary representations. This is important because when we create our regular expressions to blacklist certain characters (or some other validation process), they may not catch malicious characters because we are checking for a different "version" of the same character.

IDS50-J is the recommendation that says that filenames should use conservative naming conventions. This includes making sure the file names don't contain potentially harmful characters that could interact with the shell on the computer, start scripting attacks, or mess with the program in any way.

Original Code Vulnerabilities:

```
public class CSC245_Project2 {

   public static void main(String[] args) {
       // Read the filename from the command line argument
       // FIO16-J Pathname not Canonicalized
       // IDS01-J Filename is not normalized.
       // IDS50-J Filename is not validated for malicious characters.
       String filename = args[0];
       BufferedReader inputStream = null;

       String fileLine;
```

```
        try {
            inputStream = new BufferedReader(new FileReader(filename));
```

IDS01-J:
The filename for the file being opened is not being normalized, which allows unrecognized characters to potentially slip through a validation process that will be implemented in the secured version of this program.

IDS50-J:
The filename is not being validated in any way to make sure the filename input in args[0] does not contain these malicious characters.

- “*”
- “?”
- “|”
- “<”
- “>”
- “/”
- “:”
- “\”

```
try {
   inputStream = new BufferedReader(new FileReader(filename));

   System.out.println("Email Addresses:");
   // Read one Line using BufferedReader
   while ((fileLine = inputStream.readLine()) != null) {
       // IDS01-J String not canonicalized before validation.
       // IDS51-J Strings are not sanitized or validated.
       System.out.println(fileLine);
   }
```

IDS01-J:
In the original code, none of the lines of the file being read were being normalized before a potential validation process and then printing. Leading to the same issues as listed previously.

Secured Version:

```
// IDS50-J the filename that we are reading needs to be validated with this
regular expression
Pattern filePattern = Pattern.compile("[*?|<>/:\\\\]");

// IDS01-J The Filename needs to be normalized before being validated with this
regular expression
```

```
String filename = Normalizer.normalize(args[0], Normalizer.Form.NFKC);
Matcher fileBlacklist = filePattern.matcher(args[0]);

if (fileBlacklist.find()) {
   throw new IllegalArgumentException("Bad Filename");
}

//FIO16-J
try {

   // Define base directory (allowed directory)
   File baseDir = new File("/").getCanonicalFile();

   // Build requested file path
   File requestedFile = new File(baseDir, filename).getCanonicalFile();

   // Ensure file stays inside allowed directory
   if (!requestedFile.getPath().startsWith(baseDir.getPath())) {
       System.out.println("Invalid file.");
       return;
   }

   // Open and read file safely
   try (BufferedReader reader =
                new BufferedReader(new FileReader(requestedFile))) {
```

IDS01-J:
In the secure version of this code, the filename (args[0]) is being normalized by the line:

```
String filename = Normalizer.normalize(args[0], Normalizer.Form.NFKC);
```

This line converts the filename string into the NFKC format, which normalizes the string, allowing it to safely go through the steps of validation. Another version of this line is used later in the program when each line of text is being read from the file.

```
while ((line = reader.readLine()) != null) {
   // IDS01-J
   // There was originally no normalization of input
   // we need to normalize the string "fileLine" that is being read
   String fileLine = Normalizer.normalize(line, Normalizer.Form.NFKC);
   // NFKC Format will convert the string into its canonicalized form,
   // so it will not be in an ambiguous form
```

IDS50-J:

In this secure code, the filename is being validated via pattern matching with a regular expression designed to blacklist the malicious characters.

```
Pattern filePattern = Pattern.compile("[*?|<>/:\\\\]");
Matcher fileBlacklist = filePattern.matcher(args[0]);
if (fileBlacklist.find()) {
   throw new IllegalArgumentException("Bad Filename");
}
```

Examples:
IDS50-J:
If we change the filename in args[0] to Email_Addresses_20210205<>.txt instead of Email_Addresses_20210205.txt, we can see how this validation catches the malicious files in both the insecure and secure code.

Insecure Code:

```
Email Addresses:
doug@arapahoe.edu
nina@arapahoe.edu
tracy@arapahoe.edu
TESTING!@#$%@arapahoe.edu
jodi@arapahoe.edu
kristi@arapahoe.edu
TestText@#$..edu
TestingAgain@@..comdf
jessie@arapahoe.edu
laura%@arapahoe.edu
tina@arapahoe.edu
todd@arapahoe.edu
douglas.lundin@student.arapahoe.edu
<script>test</script>
ed@arapahoe.edu

Process finished with exit code 0
```

In the insecure code, the program runs as normal, without any exceptions occurring.

Secure Code:

```
Exception in thread "main" java.lang.IllegalArgumentException: Bad Filename
	at CSC245_Project2.main(CSC245_Project2.java:28)
```

Process finished with exit code 1

In the secure code, the program spits out an exception telling the user “Bad Filename”.

IDS01-J:
NOTE: This is not code used in the secure or insecure version of the code; it is simply an example of how normalization prevents characters like this from slipping through the cracks.

Insecure:

```
Pattern filePattern = Pattern.compile("[*?|<>/:\\\\]");

// IDS01-J The Filename needs to be normalized before being validated with this
regular expression
// String filename = Normalizer.normalize(filename, Normalizer.Form.NFKC);

String filename = args[0];

filename = "\uFE64" + filename;

Matcher fileBlacklist = filePattern.matcher(filename);

if (fileBlacklist.find()) {
   throw new IllegalArgumentException("Bad Filename");
}
```

The code above is an example of validation without normalization. The input string from args[0] is concatenated with “\uFE64”, which is a smaller version of the “<” symbol. When this code is run without normalization, the regular expression fails to detect the malicious character, resulting in no exception.

Result:
Process finished with exit code 0

Secure:

```
Pattern filePattern = Pattern.compile("[*?|<>/:\\\\]");

// IDS01-J The Filename needs to be normalized before being validated with this
regular expression


String filename = args[0];

```

```
filename = "\uFE64" + filename;

filename = Normalizer.normalize(filename, Normalizer.Form.NFKC);

Matcher fileBlacklist = filePattern.matcher(filename);

if (fileBlacklist.find()) {
   throw new IllegalArgumentException("Bad Filename");
}
```

In this secure example, the string is being normalized after the malicious character is concatenated to it. When this is run, the regular expression will catch the character, and the program will correctly result in an exception.

Result:
```
Exception in thread "main" java.lang.IllegalArgumentException: Bad Filename
	at CSC245_Project2.main(CSC245_Project2.java:35)

Process finished with exit code 1
```

IDS51-J : Roger

Properly encode or escape output:

The original code did not provide normalization of the email string before being passed, the while loop was left empty.

```
public class CSC245_Project2 {
   // Read the filename from the command line argument
   String filename = args[0];
   BufferedReader inputStream = null;
//Validates line in file is in format of email
private static boolean validEmail(String email) {
   if (email == null)
       return false;
```

An expression pattern and validEmail() method were added to provide validation.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class CSC245_Project2 {

   private static final Pattern EMAIL_PATTERN =
           Pattern.compile("[a-z0-9._%+-]+"
                + "@[a-z0-9.-]+\\.[a-z]{2,3}", Pattern.CASE_INSENSITIVE);
   private static Matcher matcher;


//Validates line in file is in format of email
public static boolean validEmail(String email) {
   if (email == null)
       return false;
   matcher = EMAIL_PATTERN.matcher(email);
   return matcher.matches();
}
```

Input entering the system must be sanitized, and the data leaving the system must be validated before it is displayed. When the input validation is implemented, the string will be normalized before being evaluated. Each line read from the file is normalized to its canonical form before being passed to validEmail(). This prevents characters from bypassing the regex check.

The Pattern uses regex to define what a valid email looks like. [a-z0-9._%+-]+ matches all the characters, numbers, and special characters (periods, underscore, percent, add, subtract) up to the @ symbol. @ [a-z0-9.-]+ compiles and matches the characters after the @symbol or the domain name. \.[a-z]{2,3} this will define the letters and whether it is 2-3 letters.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class CSC245_Project2 {

   private static final Pattern EMAIL_PATTERN =
            Pattern.compile("[a-z0-9._%+-]+"
                + "@[a-z0-9.-]+\\.[a-z]{2,3}", Pattern.CASE_INSENSITIVE);
   private static Matcher matcher;
//Validates line in file is in format of email
public static boolean validEmail(String email) {
   if (email == null)
       return false;
   matcher = EMAIL_PATTERN.matcher(email);
   return matcher.matches();
}
```

This is the output when run:

```
Email Addresses:
doug@arapahoe.edu
nina@arapahoe.edu
tracy@arapahoe.edu
```

```
jodi@arapahoe.edu
kristi@arapahoe.edu
jessie@arapahoe.edu
laura%@arapahoe.edu
tina@arapahoe.edu
todd@arapahoe.edu
douglas.lundin@student.arapahoe.edu
ed@arapahoe.edu

Process finished with exit code 0
```

```java
public static boolean validEmail(String email) {
//creates the filter to check for allowed characters
//first part applies to the username
//second half applies to the domain
//third applies to final part of the domain (.edu, .com, etc)
   Pattern EMAIL_PATTERN = Pattern.compile("[a-z0-9._%+-]+" +
"@[a-z0-9.-]+\\.[a-z]{2,3}", Pattern.CASE_INSENSITIVE);
   Matcher matcher;
// checks for some cases not contained in the regex, then returns false if
the email fails them
   if (email == null || email.contains("..") || email.startsWith(".")){
       return false;
   }
// Checks the string against the matcher and returns true or false.
   matcher = EMAIL_PATTERN.matcher(email);
   return matcher.matches();
}
```

```java
// passes the fileLine into a function to check if the email is real
Boolean email = validEmail(fileLine);
// if the email passes the tests in validEmail, print the out the email.
if (email == true) {
    System.out.println(fileLine);
}
```

IDS08-J : Liam

Any string that is to be used in a regular expression(regex) should be sanitized before use. This is to prevent unwanted effects of the regex when it is used, and can vary depending on how it is used. Any inputs should not be able to edit the original function of the regex.

“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.”(Carnegie Mellon)

The original code that was assigned to be worked on and the solutions that were implemented did not have this error.

Sources:

https://wiki.sei.cmu.edu/confluence/spaces/java/pages/88487679/ERR01-J.+Do+not+allow+exceptions+to+expose+sensitive+information

https://wiki.sei.cmu.edu/confluence/display/java/FIO16-J.+Canonicalize+path+names+before+validating+them

https://wiki.sei.cmu.edu/confluence/display/java/IDS01-J.+Normalize+strings+before+validating+them

https://wiki.sei.cmu.edu/confluence/display/java/IDS08-J.+Sanitize+untrusted+data+included+in+a+regular+expression

https://wiki.sei.cmu.edu/confluence/display/java/IDS50-J.+Use+conservative+file+naming+conventions

https://wiki.sei.cmu.edu/confluence/display/java/IDS51-J.+Properly+encode+or+escape+output

https://docs.oracle.com/javase/tutorial/essential/regex/index.html

https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

Github:
Original Code:
https://github.com/douglundin/CSC2045_Project1_Insecure_Spring2026.git

Secured Code:

https://github.com/Thortone/CSC245_Project1_Secure_Spring2026.git