James Van Gilder 9081186117

```
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main ../unsafe_program
UNSAFE PROGRAM OUTPUT

Program Exit Code: 0
```

Example of input that allows attackers to traverse directories freely

Fixed Code:

```
import java.io.IOException;
import java.lang.ProcessBuilder.Redirect;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.File;

public class Main {

    /**
     * Execute the safe program named by the first argument to this program.
     *
     * @param args
     *          must be at least one non-empty string, namely the executable name
     *          for execSafeProgram
     */
    public static void main(String[] args) {
        if (args.length < 1) {
            // must have an argument
            System.err.println("Must have at least one argument.");
            System.err.println("Usage: java Main \"executable_name\"");
            System.exit(-1);
        }
        if (args[0].length() < 1) {
            // first argument must be non-empty executable name
            System.err.println("Executable name must not be empty.");
            System.err.println("Usage: java Main \"executable_name\"");
            System.exit(-1);
        }

        // execute the program and print the exit code
        int retVal = execSafeProgram(args[0]);
        System.out.println();
        System.out.println("Program Exit Code: " + retVal);
    }

    /**
```

```
     * Execute a program found within the directory "safe_programs" in the current
     * working directory. Use as helper function to execute any of a pre-specified
     * "safe" programs found in the safe directory.
     *
     * @param programName
     *          name of executable found in safe_programs
     * @return exit value of program or -1 if unable to start, wait for, and join
     *       the child process
     * @throws SecurityException
     *           if the program name tries to escape the safe directory
     */
    private static int execSafeProgram(String programName) {
        // find the program to execute
        Path safeDir = Paths.get("safe_programs");
        Path exePath = safeDir.resolve(programName);
        // create string representation of file path for secondary check
        File f = new File(exePath.normalize().toString());
        // check if the executable is even in the safe_programs directory
        // ensure that the normalized input path is the same as the path input to
        // prevent any sort of path shortcuts, users should not use this anyway
        // ensure that the parent path of the exe is the safe_programs directory
        // Check that the absolute path is the same when normalized (see above)
        // Make sure that the user input doesn't have any file separators
        if (!f.getAbsolutePath().contains("safe_programs")
          ||!exePath.normalize().equals(exePath)
          || !exePath.getParent().equals(safeDir)
          || !exePath.toAbsolutePath().equals(exePath.toAbsolutePath().normalize())){
          || programName.contains("\\") || programName.contains("/")) {
          System.out.println("Do not attempt to leave the expected directory!");
          return -1;
        }
        // configure program runtime to execute ./safe_programs/programName
executable
        ProcessBuilder procBuild = new ProcessBuilder(exePath.toString());

        // capture output and print to current shell
        procBuild.redirectErrorStream(true);
        procBuild.redirectOutput(Redirect.INHERIT);

        try {
            // execute the program
            Process p = procBuild.start();
            // wait for program to return and exit
            return p.waitFor();
        } catch (IOException ex) {
```

```
                // error starting process
                System.out.println("Error running program: " + ex.getMessage());
                // Stop printing the stack trace on errors that can be done by users
                return -1;
            } catch (InterruptedException ex) {
                // error waiting for process
                System.out.println("Error running program: " + ex.getMessage());
                // Stop printing the stack trace on errors that can be done by users
                return -1;
            }
        }
    }
```

```
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main pwd
/home/user/Desktop/EXERCISES/3.6.1_directory_traversal

Program Exit Code: 0
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main whoami
user

Program Exit Code: 0
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main ../unsafe_program
Do not attempt to leave the expected directory!

Program Exit Code: -1
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main ../safe_programs/../unsafe_program
Do not attempt to leave the expected directory!

Program Exit Code: -1
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main ./unsafe_program
Do not attempt to leave the expected directory!

Program Exit Code: -1
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main /safe_programs/../unsafe_program
Do not attempt to leave the expected directory!

Program Exit Code: -1
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main ~/Desktop/EXERCISES/3.6.1_directory_traversal/uns
afe_program
Do not attempt to leave the expected directory!

Program Exit Code: -1
```

```
user@software-security22:~/Desktop/EXERCISES/3.6.1_directory_traversal$ java Main cd
Error running program: Cannot run program "safe_programs/cd": error=2, No such file or directory
```

Examples of input that would have previously allowed attackers to traverse directories freely

Explanation:

My attack used the path link shortcut ".." to back out of the safe_programs directory that the program forced me into and then combined that with a call to the executable unsafe_program to attack the receiving program. Other attacks could have been included, such as

My mitigation includes redundancies, but the gist of it is that in either path or string form, the executable file that is being accessed by calling java Main is in the safe_programs directory. This is accomplished via the following four steps:

First, I ensure that the string contains no path shortcut commands because users have no need to be using links or shortcuts during legitimate use.

Second, I ensure that the canonical path matches the absolute path and that it passes through the directory safe_programs.

Third, I ensure that the executable has safe_programs as the parent directory, as that is where all legitimate executables should be stored.

Fourth, by preventing the user input from containing file separators, there is no way for the attacker to get into directories or make traversal calls that aren't approved of (there are no approved directory traversals).

These redundancies are why my mitigation is so robust and why I believe that directory traversal attacks are not possible in Main.java anymore.