JAVA SE

Quiz yourself: Securing untrusted code's access to file systems

Do you know how to protect your system's /etc/passwd file?

by Simon Roberts and Mikalai Zaikin

January 5, 2021

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

The objective of this Java SE 11 quiz is to see if you know how to secure applications.

In this scenario, imagine that you are building an application that gathers and provides statistical information about employees, departments, and contracts. The application allows third-party plugins that can create reports by specific criteria—and the third-party plugins are not trusted code.

Your own code provides a utility method that reads files on behalf of the untrusted code. In the utility, the action of reading from the file system is wrapped in a `doPrivileged()` method so that it executes with the permissions of your trusted code rather than being rejected.

Which code fragment will provide the best security for your application? Choose one.

A.

```
public static Stream<String> loadData(String
  try {
    return (Stream<String>)AccessController.d
      public Stream<String> run() throws IOEx
        return Files.lines(Path.of(absoluteFi
      }
    });
  } catch (PrivilegedActionException e) {
    throw new RuntimeException(e);
  }
}
```

The answer is A.

B.

```
public static Stream<String> loadData(String
  String dir = "/opt/application/";
  try {
    return (Stream<String>)AccessController.d
      public Stream<String> run() throws IOEx
        return Files.lines(Path.of(dir, relat
      }
    });
  } catch (PrivilegedActionException e) {
    throw new RuntimeException(e);
  }
}
```

The answer is B.

C.

```
private static enum Data {EMPLOYEES, DEPARTME
private static Map<Data, Path> pathMap = Map.
  Data.EMPLOYEES, Path.of("/opt/application/e
  Data.DEPARTMENTS, Path.of("/opt/application
  Data.CONTRACTS, Path.of("/opt/application/c
);
public static Stream<String> loadData(Data da
  Path path = pathMap.get(data);
  try {
    return (Stream<String>)AccessController.d
      public Stream<String> run() throws IOEx
        return Files.lines(path);
      }
    });
  } catch (PrivilegedActionException e) {
    throw new RuntimeException(e);
  }
}
```

The answer is C.

D.

```
public static Stream<String> loadData(String
  String dir = "/opt/application/";
  try {
    return (Stream<String>)AccessController.d
      public Stream<String> run() throws IOEx
        return Files.lines(Path.of(dir,  rela
      }
    });
  } catch (PrivilegedActionException e) {
    throw new RuntimeException(e);
  }
}
```

The answer is D.

Answer. Before diving into this question, we should mention that an alternative approach to the problem would be to grant specific permissions to the untrusted plugin code. However, that idea won't be investigated in this discussion.

The general goal of a `doPrivileged` invocation is to lend privilege from the code that's invoking `doPrivileged` to the thread stack that originated the call. In this way, trusted local code can perform actions on behalf of untrusted calling code. Perhaps surprisingly, however, is the fact that the `doPrivileged` mechanism and its details are not the focus of this question. Instead, this question explores how the writing of that trusted code to ensure that the untrusted code can't trick the system into doing anything unintended.

As usual with security, sound guidance is "that which is not expressly permitted is denied." So, whatever it is that you want the calling code to be able to do must be the only thing that can be done. In the question, there's no explicit statement of exactly what files the caller might need, but it should be clear that granting arbitrary file system access would be a very bad mistake. In fact, in line with the sound guidance, you can deduce that whichever of the options allows the untrusted code the least access to the file system is the right answer.

Let's examine what's allowed by each of the options.

Option A allows the untrusted code to pass in any absolute filename, and the application's privileged code will read and return the content of the file to that untrusted caller. The only requirement is that the user ID running the program must have access to the file. It should be immediately obvious that this is a very bad idea. It would, for example, grant access to the `/etc/passwd` file that UNIX systems use to store user IDs. From this, you can be fairly confident that option A is incorrect.

Option B is superficially better. The caller provides a relative filename, and the code prepends the application's base directory to that and then reads the file. This is clearly intended to restrict the accessible files to those in a particular subdirectory intended

for use by this program. However, the provided relative path can contain double dots that navigate up to a directory in Windows or UNIX. This would allow the caller to navigate to any arbitrary file on the file system. Once again, the caller would probably be able to access something such as `../../etc/passwd` with the incumbent risks. Clearly option B is also incorrect.

Option D is again superficially better. It calls `Path.normalize()` to clean up the provided path. However, that call does not prevent the use of double dot path elements; rather, it simply removes redundant elements of the path. For example, `/opt/application/../../etc/passwd` will become simply `/etc/passwd`. Consequently, option D must also be incorrect.

Option C takes a different approach entirely. By defining three `enum` values and accepting only an argument of that `enum` type, the called code can be asked to open a file based on a description, rather than by a path. The description is translated into an actual path by looking it up in a map. Because the map is in the control of the trusted code, and it cannot be altered by the untrusted code, this option gives access to three specific files—and only those three files. Based on this, you can see that option C provides the greatest control and must be the correct answer.

Conclusion: The correct answer is option C.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Share this Page