Path traversal mitigation

As we saw in the previous assignments, protecting a file upload can be daunting. The thing comes down to trust input without validating it. In the examples shown before, a solution might be not to trust user input and create a random file name on the server-side.

If you need to save it based on user input, the best way to keep you safe is to check the canonical path. For example, in Java:

```
var multiPartFile = ...
var targetFile = new File("/tmp", multiPartFile.getOriginalName());
var canonicalPath = targetFile.getCanonicalPath();

if (!canonicalPath.startWith("/tmp") {
  throw new IllegalArgumentException("Invalid filename");
}

IOUtils.copy(multiPartFile.getBytes(), targetFile);
```

The canonical path function will resolve to an absolute path, removing `.` and `..` etc. By checking whether the canonical the path is inside the expected directory.

For path traversals, while retrieving, one can apply the same technique described above, but as a defense in depth you can also implement mitigation by running the application under a specific not privileged user who is not allowed to read and write in any other directory.

Make sure that you build detection for catching these cases in any case, but be careful with returning explicit information to the user. Every tiny detail might give the attacker knowledge about your system.

Be aware…

As shown in the previous examples, be careful which method you use to retrieve parameters, especially query parameters. Spring Boot does a decent job denying invalid path variables. To recap:

```
@Getter("/f")
public void f(@RequestParam("name") String name) {
  //name is automatically decoded so %2E%2E%2F%2E%2E%2Ftest will become ../../test
}

@Getter("/g")
public void g(HttpServletRequest request) {
  var queryString = request.getQueryString(); // will return
}

@Getter("/h")
public void h(HttpServletRequest request) {
```

```
var name = request.getParam("name"); //will return ../../test
```

If you invoke `/f` with `/f?name=%2E%2E%2F%2E%2E%2Ftest` it will become `../../test`. If you invoke `g` with `/g?name=%2E%2E%2F%2E%2E%2Ftest` it will return `%2E%2E%2F%2E%2E%2Ftest` **NO** decoding will be applied. The behavior of `/h` with the same parameter will be the same as `/f`

As you can see, be careful and familiarize yourself with the correct methods to call. In every case, write a unit test in such cases, which covers encoded characters.

Spring Boot protection

By default, Spring Boot has protection for using, for example, `../` in a path. The projection resides in the `StrictHttpFirewall` class. This will protect endpoint where the user input is part of the `path` like `/test/1.jpg` if you replace `1.jpg` with `../../secret.txt`, it will block the request. With query parameters, that protection will not be there.

In the lesson about "File uploads" more examples of vulnerabilities are shown.