Lab Exercise shell-injection

This is a lab exercise on developing secure software. For more information, see the introduction to the labs.

1. Task

Eliminate a shell injection vulnerability.

2. Background

In many cases, programmers run system commands from their applications. This happens for multiple reasons: running a complicated tool, avoiding to code a functionality that exists in another tool, or just because the application (often a web application) is an interface to a tool running on the server system. Doing this incorrectly, however, can lead to vulnerabilities.

Quite often the list of parameters of that external tool isn't static. One or multiple parameters are given by an untrusted end user. This may be the name they give to the project, for example. The application needs to avoid passing these parameters through the shell at all if the shell isn't needed.

In addition, the application need to carefully handle the given parameters to make sure they do not contain special characters like the end of the command (often ';'). Otherwise, an attacker might be able to craft input in such a way that they execute commands of their choice in addition to the one present in the application. We need to do that with an allowlist, that is, by expressly listing what is allowed (and forbidding everything else).

3. Task Information

In this task we are going to fix a shell command injection situation. You may want to test the result of different user input separately to see what happens.

Our scenario is a simple one. In our application in Python, we want to list files in a temporary directory used by our application for data processing. The original code was:

```
    subprocess.run("ls -l", shell=True)
```

Then, another developer added user input for the directory with the raw user data in the variable `dir_to_list` as follows:

```
  def list_directory(dir_to_list):
      subprocess.run(f"ls -l {dir_to_list}", shell=True)
```

This is a shell injection vulnerability. An attacker can put arbitrary text into the parameter, which will be executed by the shell. It would safer to ensure that only safe characters would be considered, *and* to *not* invoke the shell at all. By taking both steps, we make it much harder for an attacker to find a way to work around it.

We can offer a few hints. You'll need to modify a string to only contain alphanumerics. Python's `re.sub(PATTERN, REPLACETEXT, ORIGINAL_VALUE)` returns the result of substituting `PATTERN` with `REPLACETEXT` in `ORIGINAL_VALUE`, where `PATTERN` is a regular expression written as a string. By default `re.sub` substitutes all matches (that's what you want). In Python regular expressions are usually expressed using the "raw" string form, that is, using the `r'...'` syntax. You need to stop invoking the shell in Python's `subprocess.run`, including separating the arguments.

To be fair, a typical Python program wouldn't call "ls" directly. We'll do that to keep the example simple, since there are times when you need to invoke a shell.

Use the “hint” and “give up” buttons if necessary.

4. Interactive Lab (COMPLETE!)

Rewrite the following function to be safe:

```
def list_directory(dir_to_list):
  # Modify the directory so it only contains a-zA-Z0-9
  clean_dir = re.sub(r'[^a-zA-Z0-9]', '', dir_to_list)

  # Then use subprocess in a safer way
  subprocess.run(["ls", "-l",clean_dir])
```

Hint Reset Give up

This lab was developed by Marta Rybczynska.

Completed 2025-02-24T22:33:50.819Z 59dffa53-64c5-4ddb-a11b-9f542be99927 04fpc8c1d702ac