Secure Programming Practices

ONAP has long agreed that its projects should follow the principles of Secure Design, which is one of the CII Badging requirements. The primary reference given by CII for Secure Design is Saltzer & Schroeder.

The following documents are recommended reading on the topic of Secure Programming Practices. (A web search on "secure programming practices" will also produce many articles on the topic.)

- Open Web Application Security Project (OWASP) Secure Coding Practices Quick Reference Guide
- Software Assurance Forum for Excellence in Code (SAFECode) Fundamental Practices for Secure Software Development
- Oracle Secure Coding Guidelines for Java
- CMU Software Engineering Institute (SEI) CERT Coding Standards (C, C++, Android, Java, Perl)
- Apple Shell Script Security

As you read these, you'll find a number of common themes.

The SAFECode document lays things out very nicely into separate practices for Design, Coding, and Testing & Validation. Under Design, they discuss: Secure Design Principles, Threat modeling, Encryption Strategy, Standardize Identity and Access, and Establish Log and Audit Practices. Under Coding, they discuss: Coding Standards, Using safe functions only, Using code analysis tools, Handling data safely, and Error Handling. And under Testing, they discuss both Automated and Manual Testing.

The other documents are oriented specifically to coding. For example, OWASP has some good checklists in these categories:

OWASP Coding Practice Checklists		
Input Validation	Error Handling and Logging	Cryptographic Practices
Output Encoding	Data Protection	Memory Management
Authentication and Password Management	Communication Security	File Management
Session Management	System Configuration	General Coding Practices
Access Control	Database Security	

Invoking External Processes

The SECCOM was recently asked to make specific recommendations about how to safely invoke external programs from a language such as Python. The following examples use shell and python for exposition, but the issue exists in all languages that provide mechanisms to execute sub-processes, including Java, C and C++.

- Use full paths for the programs you are executing. Note: While this can be done for shell scripts, this is most useful for **non**-shell scripts. For example,
  - shell (but see the note above):

    ```
    /bin/ls -l file
    ```

  - python:

    ```
    subprocess.Popen(["/bin/ls", "-l", "file"], ...)
    ```

- If your external programs can be found in a variety of locations depending on the system on which they are run, use exact control over the PATH used to find your programs. For example, you can add a line to the beginning of your program such as:
  - shell:

    ```
    export PATH=/bin:/usr/bin
    ```

    or

    ```
    export PATH=/sbin:/usr/sbin:/bin:/usr/bin
    ```

  - python:

    ```
    os.environ['PATH'] = '/bin:/usr/bin'
    ```

    or

    ```
    os.environ['PATH'] = '/sbin:/usr/sbin:/bin:/usr/bin'
    ```

- Make certain that the current directory ("`.`") and relative directories (those not starting with a "`/`") are not at the beginning or middle of the PATH.
  - Both an empty path element ("`::`") and "`:./:`" are equivalent to including "`.`". That is, "`::`", "`:.:`", and "`:./:`" are all equivalent.
  - A leading "`:`" is equivalent to a leading "`.:`".
- If you must depend on an externally-provided path, combine prepending known locations to the beginning of the PATH, with sanitizing the rest of it. For example, you can add a line to the beginning of your program such as:
  - shell:

```
export PATH="/bin:/usr/bin:$(sanitize "$PATH")"
```

- python:

```
os.environ['PATH'] = f"/bin:/usr/bin:{sanitize(os.environ['PATH'])}"
```

- You will have to write the sanitize() function. Some things to consider are:
  - The current directory is dangerous in the PATH, except *possibly* at the end. So one thing your sanitize function should do is to remove the current directory, or any aliases that map into the current directory (e.g. `::`, `:.:`, `:./:`, `:./.:`, etc.).
  - Relative directories, those not starting with "/" are similarly dangerous.
  - It's also dangerous for any of the directories found in the PATH to be world writable, where someone can create a program with the same name as a system tool invoked by your program. (Consider having a PATH with `/tmp` in it, where someone could have placed a script named "`ls`".)

- Use exact control over the command line arguments. For example, in Python ***avoid*** using "`shell=True`" with the subprocess module's methods, which can allow unexpected parsing or expansion of the values being passed. For shell, ALWAYS use quote (") marks around your variable expansions; otherwise whitespace within the values may wind up generating separate arguments.

- Use exact control over the current directory before invoking a program using a relative path. For example, invoking "../somewhere/something" will act differently depending on where it is invoked.
  - If your program does a cd/chdir, ALWAYS check that it succeeded before proceeding. For example,
    - shell:

```
cd somewhere || { echo "Cannot cd somewhere: $!" 1>&2; exit 99; }
```

    - python:

```
try:
    os.chdir(somewhere)
except Exception as e:
    sys.exit(f"Cannot chdir({somewhere}): {e}")
```

- Use a "lint" finding program. For example,
  - shell: use "shellcheck", which can be installed using "`apt install -y shellcheck`"
  - python: there are a number lint-finding programs, such as "pylint" and "flakes8".

Code quality evaluation

the use of code quality tool help the developer to fix vulnerabilities early.

Inside the community, sonarcloud is the reference.

- https://sonarcloud.io/features (Detect, understand, and fix issues in your code, at the very earliest in your workflow)
- https://sonarcloud.io/organizations/onap/projects?sort=name (ONAP project)

Checking results and fixing them regularly are one way to reduce risk

Software Composition Analysis

Like Code quality evaluation, software composition analysis helps the community reduce its risk.

https://snyk.io/blog/what-is-software-composition-analysis-sca-and-does-my-company-need-it/

What Is a software composition analysis (SCA)?

Software Composition Analysis (SCA) is an application security methodology for managing open source components. Using SCA, development teams can quickly track and analyze any open-source component brought into a project. SCA tools can discover all related components, their supporting libraries, and their direct and indirect dependencies. SCA tools can also detect software licenses, deprecated dependencies, as well as vulnerabilities and potential exploits. The scanning process generates a bill of materials (BOM), providing a complete inventory of a project's software assets.

The community uses whitesource and Nexus-IQ.

>>More information Code Scanning Tools and CI

Use Safe and Secure Docker Images

To build ONAP images, the community provides secure docker images, which are built from Alpine images.

- https://git.onap.org/integration/docker/onap-java11/about/
- https://git.onap.org/integration/docker/onap-python/about/

These images reduce the risk of threats; please use them.