Jose Gomez
Professor Mill

CS 405

26 October 2025

Project One

Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines.

Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

Module Three Milestone

Ten Core Security Principles

Principles	Write a short paragraph explaining each of the 10 principles of security
1. Validate Input Data	Essentially makes sure that input from users or files is correct and safe before utilizing. Prevents Buffer Overflow as well as SQL Injection attacks by verifying said input.
2.Heed Compiler Warnings	Bring unreliable and sometimes dangerous coding practices to light with warnings. Signal vulnerabilities in the code such as unused variables which can then be addressed by the developer.
3.Architect and Design for Security Policies	Software follows a security protocol (particularly from its specific organization) to allow for safe communication as well as protection of data.
4. Keep it Simple	Simple code allows for better understanding and testing from not only the original developers, but also others looking to amend or modify said code (within the team). Additionally, complex coding practices increase the chance of vulnerabilities as well as identifying them.
5. Default Deny	Systems should deny open access by default, going out of its way to check for authorized individuals – Overall, this lessens the threat of information exposure to an outside source.
6. Adhere to the Principle of Least Privilege	Reduces the chance of security breaches by giving users the tools they need to complete only the tasks needed, and nothing else.

7. Sanitize data sent to other systems	Ensures that shared data with external components are validated and formatted to organizational default practice in the effort to prevent injection attacks.
8.Practice Defense in Depth	Increases safety by adding multiple layers of security such as encryption and authentication, so that there is less risk in case of one security plan failing.
9. Use Effective Quality Assurance Techniques	This includes peer review (which is also why “keeping it simple” is important) and testing, helping to catch vulnerabilities and ensure formatting is correct.
10. Adopt a secure coding standard	Utilizing a standard such as SEI CERT C++ allows for consistently secure code for all projects utilizing it.

C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

1) Data Type Coding Standard:

Label: STD-001-CPP

Name of Standard: Use Appropriate and Consistent Data Types

Rationale: Using consistent data types allows for safe operation, as opposed to using varying degrees of variables which can be exploited by any attackers.

Non-Compliant Code:

short total = 500;
total = total + 248;

Description: In this case, “short” variables can cause overflow when any value above its limit is added.

Compliant Code:

int integer = 500;
total = total + 248;

Description: “int” allows for more range and flexibility to prevent overflow.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Possible	Medium	High	2

2) Data Value Coding Standard

Label: STD-002-CPP

Name of Standard: Validate All Data Values Before Use

Rationale: Ensuring that all variables are verified allows for consistent logic within the code, preventing attacks as well as data corruption.

Non-Compliant Code:

```
int dividingValue = userInput();
int quotient = 5000 / dividingValue;
```

Description: Risk of failure as the user input can be zero, causing data corruption.

Compliant Code:

```
int dividingValue = userInput();
if (dividingValue !=0) {
    int quotient = 5000 / dividingValue;
}
```

Description: Ensures division is performed without dividing by zero.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Low	High	1

3) String Correctness Coding Standard:

Label: STD-003-CPP

Name of Standard: Use Safe String Handling Functions

Rationale: Utilizing unsafe strings may lead to memory corruption as well as the possibility of buffer overflow. Safe string functions may limit certain actions from the user.

Non-Compliant Code:

```
char value[6];
strcpy(value, “String is too long. Please try again.”);
```

Description: “strcpy” does not limit characters that can be copied, leading to possible overflow.

Compliant Code:

```
char value[6];
strncpy(value, “SafeStr”, sizeof(value) - 1);
value[sizeof(value) - 1] = ‘\0’;
```

Description: Instead uses “strncpy()” along with integrated size check for safer handling of values.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	1

4) SQL Injection Coding Standard

Label: STD-004-CPP

Name of Standard: Use Parameterized Queries to Prevent SQL Injection

Rationale: Utilizing user input within SQL queries leaves database vulnerable to breaches from attackers. Parameterized queries ensure user input is data, not code to be executed.

Non-Compliant Code:

```
string userInput = getInput();
string query = “SELECT * FROM products where id = “ + userInput;
executeQuery(query);
```

Description: User input is within SQL command, allowing breachers to inject statements with different values.

Compliant Code:

```
string userInput = getInput();
PreparedStatement stmt = conn.prepareStatement(“SELECT * FROM products WHERE id = ?”);
stmt.setString(1, userInput);
executeQuery(stmt);
```

Description: Uses parameterized query so that the user input is treated solely as data.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
Critical	Likely	Medium	Very High	1

5) Memory Protection Coding Standard

Label: STD-005-CPP

Name of Standard: Manage Memory Safely and Avoid Leaks

Rationale: Properly handles memory to avoid breaches of information from attackers.

Non-Compliant Code:

```
int* data = new int[3];
processData(data);
```

Description: Leaks can occur as a result of the memory not being released correctly.

Compliant Code:

```
#include <memory>
auto data = std::make_unique<int[]>(3);
processData(data.get());
```

Description: Utilizes smart pointers (std:: line) to manage the memory automatically.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Possible	Medium	High	2

6) Assertions Coding Standard

Label: STD-006-CPP

Name of Standard: Use Assertions to Enforce Critical Assumptions

Rationale: Assertions help identify errors in code logic earlier on, preventing crashes and vulnerabilities from attackers.

Non-Compliant Code:

```
int* pointer = getPointer();
int value = *pointer;
```

Description: This query assumes the pointer is valid, without checking first.

Compliant Code:

```
#include <cassert>
int* pointer = getPointer();
assert(pointer != nullptr);
int value = *pointer;
```

Description: Assertion verifies pointer validity to prevent logic errors

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Low	Medium	3

7) Exceptions Coding Standard

Label: STD-007-CPP

Name of Standard: Use Exceptions for Error Handling, Not Control Flow

Rationale: Exceptions would be not for the projects' logic, but for exceptional conditions. Overutilizing exceptions can lead to a more complex code and thus more openness to vulnerabilities within the program.

Non-Compliant Code:

```
try {
    for (int i = 0; i < 7; i++) {
        throw i;
    }
} catch (int e) {}
```

Description: Exceptions are misused within the loop, making code unclear and inefficient

Compliant Code:

```
for (int i = 0; i < 7; i++) {
    processItem(i);
}
```

Description: Utilizes processItem() to leave exceptions for use in actual errors.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Possible	Low	Medium	3

8) Secure File Handling Standard

Label: STD-008-CPP

Name of Standard: Validate File Access and use Secure File Operations

Rationale: Unsecure file handling may lead to breaches of information to unauthorized sources, leading to attacks. Permissions and file paths should be manually validated.

Non-Compliant Code:

```
ifstream file(“C:\\temp\\sensitive.txt”);
string data;
file >> data;
```

Description: Force opens a path without checking the file permissions

Compliant Code:

```
std::ifstream file;
file.open(safePath, ios::in);
if (!file.is_open()) {
    throw runtime_error(“File Access Denied.”);
}
```

Description: Validates path and checks permissions before reading file.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Possible	Medium	High	2

9) Secure Network Communication Standard

Label: STD-009-CPP

Name of Standard: Encrypt Network Traffic and Validate Certificates

Rationale: Encryption lessens the chance of data breaches or exposed data (DiD).

Non-Compliant Code:

connect(“http://virus.com”);
Description: Transmits data over an unsecure connection.

Compliant Code:

connect(“https://virus.com”);
verifyServerCertificate();

Description: Uses HTTPS instead and server certificate to verify connection.

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
Critical	Likely	High	Very High	1

10) Input Validation Reinforcement Standard

Label: STD-010-CPP

Name of Standard: Centralize Input Validation Logic

Rationale: All input should be validated by a consistent set of rules to prevent attackers from breaching more vulnerable parts in the system.

Non-Compliant Code:

int userBirthYear;
cin >> userBirthYear;

Description: Takes input without preparing bounds

Compliant Code:

int userBirthYear = getValidatedInput(1900, 2025);
Description: Uses validation function for safe input

Threat Level:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	1

Encryption:

Encryption at Rest	Data is protected as it is moved between systems
Encryption in Flight	Stored data is protected (usually with AES-256 bit encryption)
Encryption in Use	Sensitive data is protected in memory using secure enclaves to prevent breaching of information while computing values.

Triple-A Framework:

Authentication	Utilizes Multi-Factor authentication for increased layers of security.
Authorization	Utilizes least-privilege for the basic level of users with varying levels of permissions for different roles (admin, user, etc.)
Accounting	All steps in authentication and authorization are logged, encrypted, and reviewed in cases of suspicious behavior.

Plans for Automation

Automation within Green Pace's DevSecOps pipeline enforces compliance to these coding standards through continuous integration (CI), continuous delivery (CD), and continuous security validation. Automated static and dynamic analysis tools will be integrated at multiple stages in the software development lifecycle (SDLC) to detect, report, and block insecure code before deployment. During the code commit and build stages, static analysis tools such as SonarQube, Coverity, and clang-tidy will automatically scan code against SEI CERT C/C++ rules. Code that violates critical or high-severity rules (e.g., memory management or SQL injection vulnerabilities) will fail the build. Automation will also include continuous monitoring dashboards within SonarQube and Jenkins to report standard compliance percentages. Automated enforcement reduces manual review workload and ensures that Green Pace maintains compliance across all development teams.

Mapping of Standards to Principles

Standard	Principles
STD-001-CPP	1,4,9,10
STD-002-CPP	1,6,9
STD-003-CPP	1,8,9,10
STD-004-CPP	1,5,7,8
STD-005-CPP	3,4,8,9
STD-006-CPP	2,4,9
STD-007-CPP	4,9,10
STD-008-CPP	5,6,7,8
STD-009-CPP	3,5,8,10
STD-010-CPP	1,7,8,10

References

Admin. (n.d.). *Sei cert C++ coding standard*. SEI CERT C++ Coding Standard - SEI CERT C++ Coding Standard - Confluence. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682