**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc163748114)

[Contents 1](#_Toc163748115)

[Overview 3](#_Toc163748116)

[Purpose 3](#_Toc163748117)

[Scope 3](#_Toc163748118)

[Module Three Milestone 3](#_Toc163748119)

[Ten Core Security Principles 3](#_Toc163748120)

[C/C++ Ten Coding Standards 4](#_Toc163748121)

[Coding Standard 1 5](#_Toc163748122)

[Coding Standard 2 7](#_Toc163748123)

[Coding Standard 3 9](#_Toc163748124)

[Coding Standard 4 11](#_Toc163748125)

[Coding Standard 5 13](#_Toc163748126)

[Coding Standard 6 15](#_Toc163748127)

[Coding Standard 7 17](#_Toc163748128)

[Coding Standard 8 19](#_Toc163748129)

[Coding Standard 9 21](#_Toc163748130)

[Coding Standard 10 23](#_Toc163748131)

[Defense-in-Depth Illustration 25](#_Toc163748132)

[Project One 25](#_Toc163748133)

[Revise the C/C++ Standards 25](#_Toc163748134)

[Risk Assessment 25](#_Toc163748135)

[Automated Detection 25](#_Toc163748136)

[Automation 25](#_Toc163748137)

[Summary of Risk Assessments 26](#_Toc163748138)

[Create Policies for Encryption and Triple A 27](#_Toc163748139)

[Map the Principles 28](#_Toc163748140)

[Audit Controls and Management 29](#_Toc163748141)

[Enforcement 29](#_Toc163748142)

[Exceptions Process 29](#_Toc163748143)

[Distribution 30](#_Toc163748144)

[Policy Change Control 30](#_Toc163748145)

[Policy Version History 30](#_Toc163748146)

[Appendix A Lookups 30](#_Toc163748147)

[Approved C/C++ Language Acronyms 30](#_Toc163748148)

## 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](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## 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. ValidateInput Data | Before processing any data received from users, external services, or file inputs, it’s essential to validate this data rigorously. Validation includes checking for correctness, type, length, format, and range. Proper input validation can prevent various forms of attacks, such as SQL injection, cross-site scripting (XSS), and buffer overflows, which protects the application from malicious inputs that could compromise security or functionality. |
| 1. Heed Compiler Warnings | Compiler warnings are not just suggestions; they often point to potential flaws in code that could lead to security vulnerabilities, unexpected behavior, or crashes. Treating warnings as errors and resolving them before proceeding ensures a higher code quality and reduces the risk of security vulnerabilities being exploited in production environments. |
| 1. Architect and Design for Security Policies | Security should be an integral part of the software design and architecture process, not an afterthought. By considering security policies and requirements from the beginning, systems can be designed to inherently resist attacks, protect data privacy, and ensure data integrity and availability. |
| 1. Keep It Simple | Complexity often leads to security vulnerabilities because the more complex a system is, the harder it is to understand, maintain, and secure. A simple design reduces the risk of errors, makes security analysis easier, and helps ensure that security features are implemented correctly. |
| 1. Default Deny | In the context of access control, the principle of default deny means that system access should be denied by default, and permissions should only be granted explicitly to actions, users, or systems that require them. This minimizes potential attack vectors and reduces the risk of unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Each part of a system should operate with the least amount of privilege necessary to complete its tasks. This principle limits the damage that can be done in the event of a security breach, as attackers or compromised components have minimal access to the broader system. |
| 1. Sanitize Data Sent to Other Systems | When data is transferred to external systems or services, it should be sanitized to remove or encode potentially malicious content. This prevents data from being used in a manner that could lead to security breaches, such as injection attacks or data leakage. |
| 1. Practice Defense in Depth | Security should not rely on a single measure but rather multiple layers of defense that include physical, technical, and administrative controls. If one layer fails, others stand to prevent a breach, making a system more resilient to attacks. |
| 1. Use Effective Quality Assurance Techniques | Employ comprehensive testing strategies, including static and dynamic analysis, code reviews, and penetration testing, to identify and mitigate vulnerabilities. Quality assurance is crucial for early detection of security issues and ensuring the reliability and security of software products. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard, such as the SEI CERT C++ Coding Standard, helps developers avoid common pitfalls and vulnerabilities associated with insecure coding practices. Adherence to a coding standard improves code quality and security posture by providing guidelines and best practices for secure coding. |

### 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.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Strong Type Checking** |
| --- | --- | --- |
| **Data Type** | [STD-001-DTP] | Enforcing strong type checking helps prevent type-related errors that could lead to security vulnerabilities, such as buffer overflows and arbitrary code execution. Using explicit data types ensures that operations on variables are intended and consistent, reducing the risk of data corruption or unexpected behavior. |

| **Noncompliant Code** |
| --- |
| Implicit type conversion can lead to loss of precision or unintended behavior. |
| int sum(int x, double y) {  return x + y; // Implicit conversion from double to int  } |

| **Compliant Code** |
| --- |
| Use explicit casting or correct data types to ensure values are handled appropriately. |
| int sum(int x, int y) {  return x + y; // Both operands are of type int  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  1: Validate Input Data  4: Keep it Simple  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | c:S1248 | SonarQube's static analysis can identify places where implicit type conversions occur. |
| Clang Static Analyzer | 12.0.1 | ImplicitConversionChecker | Clang Static Analyzer can automatically detect instances where implicit type conversions might result in data loss or unexpected behavior. |
| Coverity | 2021.03 | MISRA C:2004, 10.1 | Coverity identifies implicit conversions that violate safe coding practices. |
| CodeQL | Latest | TypeCheckingQueries | CodeQL is a semantic code analysis engine used for automating security checks. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate Data Values** |
| --- | --- | --- |
| **Data Value** | [STD-002-DVL] | Validating data values beyond just their types ensures that the data fits within expected boundaries or constraints. This principle is crucial for avoiding vulnerabilities like buffer overflows, underflows, or logic errors that could be exploited by attackers to compromise system integrity or availability. |

| **Noncompliant Code** |
| --- |
| Failing to validate that a number falls within a required range before using it. |
| void setAge(int age) {  // No validation, assuming age is always valid  this->age = age;  } |

| **Compliant Code** |
| --- |
| Ensuring the age falls within a reasonable and expected range before setting it. |
| void setAge(int age) {  if (age > 0 && age < 130) {  this->age = age;  } else {  throw std::invalid\_argument("Age is out of range");  }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  1: Validate Input Data  4: Keep It Simple  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.9.0 | Active Scan Rules | OWASP ZAP can simulate attacks on web applications to identify inputs that are not adequately validated. |
| Fortify | 20.1.0 | Data Validation | Fortify's static code analysis identifies missing or inadequate validation of data values. |
| SonarQube | 8.9 LTS | DataValidation | SonarQube can be configured with custom rules to specifically target the validation of data values across a codebase. |
| ReSharper | 2021.2 | ValueAnalysis | ValueAnalysis feature can automatically detect potential issues in the handling of data values. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Safe String Handling** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | Strings are a common source of vulnerabilities in C/C++ programs, particularly due to buffer overflow, underflow, or improper null termination. Ensuring safe handling by using bounded string operations prevents these common attack vectors, thereby protecting the application from potential exploits. |

| **Noncompliant Code** |
| --- |
| Using functions that do not perform bounds checking can lead to buffer overflows. |
| char buffer[10];  strcpy(buffer, userInput); // Dangerous if userInput is longer than 9 characters (+ null terminator) |

| **Compliant Code** |
| --- |
| Using bounded string functions to prevent buffer overflows. |
| char buffer[10];  strncpy(buffer, userInput, sizeof(buffer) - 1); // Ensures null termination and bounds checking  buffer[sizeof(buffer) - 1] = '\0'; // Explicitly null-terminate |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  1: Validate Input Data  5: Default Deny  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.03 | TAINTED\_STRING | Detects unsafe string operations that could lead to buffer overflow vulnerabilities. |
| Clang Static Analyzer | 12.0.1 | alpha.security.taint.TaintedBuffer | Scans C/C++ code for patterns that could result in tainted buffer operations. |
| Fortify | 20.1.0 | Buffer Overflow: Unbounded String Operations | Focuses on identifying usage of functions known to cause buffer overflows. |
| Klocwork | 2021.2 | NNTS.MIGHT | Analyzes the code to find non-null terminated strings (NNTS) that might result in buffer overflows. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevention of SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | SQL Injection attacks allow attackers to execute malicious SQL commands through user input fields, leading to unauthorized data access or manipulation. Utilizing parameterized queries and prepared statements ensures that user input is treated as data, not as part of the SQL command, thereby mitigating the risk of SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| Directly including user input in SQL queries without validation or sanitation, making the application vulnerable to SQL injection attacks. This can allow attackers to manipulate queries to access or modify unauthorized data. |
| // Assume user\_input is a variable holding input from the user  std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "';";  executeQuery(query); // Hypothetical function to execute SQL query |

| **Compliant Code** |
| --- |
| Using parameterized queries to prevent SQL injection. User input is treated as a parameter rather than being directly included in the SQL string, effectively neutralizing potentially malicious content. |
| // Prepared statement with parameterized query to prevent SQL injection  std::string query = "SELECT \* FROM users WHERE username = ?";  preparedStatement(query, user\_input); // Hypothetical function, binds user\_input as parameter |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  1: Validate Input Data  8: Practice Defense in Depth  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.10.0 | Active Scan Rules for SQL Injection | Automatically tests web applications for SQL injection vulnerabilities. |
| Fortify | 20.2.0 | SQL Injection | Scans code to identify patterns that could lead to SQL injection. |
| SonarQube | 8.9 LTS | SQL Injection Vulnerability | Detects unsafe SQL query constructions in code that might be vulnerable to SQL injection. |
| IBM Security AppScan | 10.0.2 | SQL Injection | Performs dynamic and static analysis to uncover SQL injection vulnerabilities in web applications. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Secure Memory Management** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MPR] | Incorrect memory management can lead to vulnerabilities such as buffer overflows, use-after-free errors, and memory leaks, which can compromise application security. Employing secure memory management practices, like using smart pointers in C++ and avoiding manual memory management, helps prevent these issues. |

| **Noncompliant Code** |
| --- |
| Manual memory management without proper checks can lead to vulnerabilities. |
| char\* buffer = new char[10];  // Unsafe operations on buffer  delete[] buffer; |

| **Compliant Code** |
| --- |
| Using smart pointers for automatic memory management. |
| std::unique\_ptr<char[]> buffer(new char[10]);  // Safe operations on buffer |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  6: Adhere to the Principle of Least Privilege  8: Practice Defense in Depth  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.16.1 | Memcheck | Valgrind's Memcheck tool detects memory management errors such as buffer overflows, leaks, and improper allocation/deallocation. |
| ASan | Integrated with GCC and Clang | Runtime memory error detector | ASan is a fast memory error detector that can identify buffer overflows, use-after-free errors, and other memory-related issues during runtime. |
| Visual Studio Code Analysis | Latest | C/C++ Code Analysis | Includes checks for memory allocation errors, uninitialized memory, and pointer errors to ensure safe memory operations in C++ applications. |
| Coverity | 2021.03 | USE\_AFTER\_FREE", "OVERRUN | Coverity analyzes source code to identify dangerous memory management patterns that could lead to vulnerabilities. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Proper Use of Assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-AST] | Assertions are used to detect critical errors in code by checking conditions that must always be true. They should not be used for error handling in production code but can be a valuable tool for catching programming errors during development. Misuse of assertions can lead to exposing potential vulnerabilities or causing unexpected behavior. |

| **Noncompliant Code** |
| --- |
| Using assertions for error handling in production code. |
| assert(user != nullptr); // Incorrect use of assert for error handling |

| **Compliant Code** |
| --- |
| Using error handling instead of assertions for production code. |
| if (user == nullptr) {  // Handle error appropriately  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  6: Adhere to the Principle of Least Privilege  9: Use Effective Quality Assurance Techniques  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | alpha.core.AssertSideEffect | This check warns about assertions that produce side effects, which should not occur in production code. |
| Coverity | 2021.03 | ASSERT\_SIDE\_EFFECT | Identifies assertions in the code that could affect the program’s state and cause unpredictable behavior. |
| SonarQube | 8.9 LTS | S2583 | This rule warns developers when an assertion is used in a way that could be replaced by more robust error handling mechanisms. |
| ReSharper | 2021.2 | InappropriateAssertionUse | Analyzes C# and C++ code to find assertions that are used for purposes other than debugging, such as controlling program flow or error handling. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Safe Exception Handling** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | Proper exception handling ensures that an application can respond to errors or unexpected conditions without crashing or exposing vulnerabilities. It's important to catch exceptions gracefully and manage resources correctly to prevent leaks and ensure the application remains stable and secure. |

| **Noncompliant Code** |
| --- |
| Not catching exceptions, leading to potential crashes. |
| int divide(int a, int b) {  return a / b; // If b is 0, an exception is thrown and not caught  } |

| **Compliant Code** |
| --- |
| Catching and handling exceptions to prevent crashes. |
| int divide(int a, int b) {  if (b == 0) {  // Handle error appropriately  return 0; // Example handling  }  return a / b;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  5: Default Deny  6: Adhere to the Principle of Least Privilege  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | java:S2221 | Ensures that exceptions are only caught with clear intent and purpose. |
| ReSharper | 2021.2 | CSharp.Errors.CaughtExceptions | Analyzes C# code to find all exceptions caught and provides insights into whether they are handled properly or simply logged and rethrown. |
| Coverity | 2021.03 | UNCAUGHT\_EXC | Identifies exceptions that are thrown but not caught within the same function. |
| Visual Studio | 2019 | CA1031 | Warns developers against catching overly broad exception types, encouraging more precise handling that can address specific error conditions effectively. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Ensuring Strong Cryptographic Practices** |
| --- | --- | --- |
| Secure Cryptography Practices | [STD-008-SCP] | Ensuring the use of strong, up-to-date cryptographic practices is critical for protecting sensitive information. This includes choosing secure algorithms, securely managing keys, and following best practices for encryption, decryption, and hashing to prevent unauthorized access or disclosure of sensitive data. |

| **Noncompliant Code** |
| --- |
| Utilizing a deprecated cryptographic hash function for sensitive data operations. |
| std::string hashPassword(const std::string& password) {  // Using MD5, which is considered insecure  return MD5(password); // Hypothetical MD5 function  } |

| **Compliant Code** |
| --- |
| Employing a strong, recommended cryptographic hash function with salt. |
| std::string hashPassword(const std::string& password, const std::string& salt) {  // Using SHA-256, which is considered secure  return SHA256(password + salt); // Hypothetical SHA256 function  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  3: Architect and Design for Security Policies  5: Default Deny  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cryptosense Analyzer | Latest | Cryptographic Security Analysis | Provides analysis of cryptography usage in applications, detecting weak cryptographic standards and suggesting upgrades to more secure alternatives. |
| Checkmarx | 2021.04 | Weak Cryptography | Scans source code for the use of deprecated cryptographic functions like MD5 and recommends more secure functions such as SHA-256. |
| SonarQube | 8.9 LTS | Cryptographic Weakness | Detects unsafe use of cryptographic APIs and outdated algorithms, ensuring compliance with current best practices. |
| Fortify | 20.2.0 | Insecure Cryptography | Identifies improper use of cryptography, such as using weak algorithms and keys, and provides guidance on secure encryption practices. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Safe File Access and Manipulation** |
| --- | --- | --- |
| Secure File I/O Operations | [STD-009-SFO] | Properly handling file input/output operations is vital to prevent vulnerabilities such as path traversal. By validating and sanitizing file paths, using secure APIs, and implementing proper error handling, the application can mitigate risks associated with insecure file operations. |

| **Noncompliant Code** |
| --- |
| Directly using user input to construct file paths, leading to potential directory traversal vulnerabilities. |
| void saveUserProfileImage(const std::string& userName, const std::string& imageData) {  std::string filePath = "/user/profile/images/" + userName + ".png";  std::ofstream file(filePath);  file << imageData;  file.close();  } |

| **Compliant Code** |
| --- |
| Validating and sanitizing user input before using it in file paths. |
| void saveUserProfileImage(const std::string& userName, const std::string& imageData) {  // Sanitize userName to remove any path traversal characters  std::string safeUserName = sanitizeUserName(userName);  std::string filePath = "/user/profile/images/" + safeUserName + ".png";  std::ofstream file(filePath);  file << imageData;  file.close();  }  std::string sanitizeUserName(const std::string& userName) {  // Remove dangerous characters from userName  return userName; // Placeholder for actual implementation  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  1: Validate Input Data  5: Default Deny  8: Practice Defense in Depth |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | File Manipulation Vulnerability | Detects potential vulnerabilities in file handling operations, such as using unvalidated user input in file paths. |
| Fortify | 20.2.0 | Path Manipulation | Identifies security risks involved with file path manipulations, suggesting best practices to validate and sanitize inputs. |
| Checkmarx | 2021.04 | Improper File Path Handling | Scans code to find improper handling of file paths that could result in path traversal or other file-related security issues. |
| OWASP Dependency-Check | Latest | Dependency Vulnerability Scanner | While primarily used for checking dependencies, it can be configured to alert developers to insecure APIs or libraries that might lead to unsafe file operations. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Utilization of Secure Communication Channels** |
| --- | --- | --- |
| Secure Communication Protocols | [STD-010-SCP] | Using secure communication protocols such as TLS/SSL for transmitting data ensures the confidentiality and integrity of the data in transit. This is essential for protecting sensitive information from interception, eavesdropping, or tampering by malicious actors. |

| **Noncompliant Code** |
| --- |
| Transmitting sensitive data over an insecure communication channel. |
| void sendCreditCardInformation(const std::string& creditCardInfo) {  // Insecurely sending data over HTTP  httpRequest("http://example.com/payment", creditCardInfo);  } |

| **Compliant Code** |
| --- |
| Ensuring all sensitive data is transmitted over secure channels. |
| void sendCreditCardInformation(const std::string& creditCardInfo) {  // Securely sending data over HTTPS  httpsRequest("https://example.com/payment", creditCardInfo);  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  5: Default Deny  7: Sanitize Data Sent to Other Systems  10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Qualys SSL Labs | Latest | SSL Test | Provides a deep analysis of the configuration of any SSL web server on the public Internet. |
| TestSSL.sh | Latest | Command-line tool for testing SSL/TLS | A script that checks a server's service on any port for the support of TLS/SSL ciphers, protocols as well as some cryptographic flaws. |
| OWASP ZAP | 2.10.0 | SSL/TLS Scanner | Scans for SSL/TLS vulnerabilities in applications to ensure secure transmission channels are utilized. |
| Burp Suite | 2021.8 | SSL/TLS Auditor | Analyzes the SSL/TLS configuration of web applications, verifying that only strong protocols and ciphers are in use. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

During the build and test phases, continuous integration tools such as Jenkins are integrated with static application security testing (SAST) tools like Fortify or Checkmarx. This setup ensures that every build is scanned for security vulnerabilities, maintaining compliance with established coding standards. Dynamic application security testing (DAST) is also automated alongside regular functional testing during the test phase to catch runtime vulnerabilities.

Post-deployment, the use of configuration management tools ensures that all deployments adhere to secure configurations, while automated vulnerability scanners continuously monitor the production environment for new risks. Security information and event management (SIEM) systems then provide ongoing monitoring and real-time alerting, facilitating rapid response to detected threats. This comprehensive approach not only secures the application across its lifecycle but also embeds a culture of security within the team, ensuring continuous improvement in security practices and compliance with security policies.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-DTP | High | Medium | Low | High | 3 |
| STD-002-DVL | High | Medium | Medium | High | 4 |
| STD-003-STR | High | High | Medium | High | 4 |
| STD-004-SQL | High | High | Medium | High | 4 |
| STD-005-MPR | High | High | Medium | High | 4 |
| STD-006-AST | Medium | Medium | Low | Medium | 3 |
| STD-007-EXC | High | Medium | Medium | High | 4 |
| STD-008-SCP | High | Medium | High | High | 4 |
| STD-009-SFO | High | High | Medium | High | 4 |
| STD-010-SCP | High | High | Medium | High | 4 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is essential for protecting stored data on physical and digital storage media. This policy mandates the use of full disk encryption technologies and encryption methods for databases and other storage solutions to secure data against unauthorized access. |
| Encryption in flight | Encryption in flight safeguards data being transferred over networks. This policy requires the use of secure communication protocols such as TLS 1.2 or higher for all data exchanges across internal and external networks. |
| Encryption in use | Encryption in use protects data actively being processed by ensuring that operations on data are performed securely. This includes scenarios where data is temporarily decrypted for processing and should be protected against exposure. |

Source: <https://www.mimecast.com/blog/data-in-transit-vs-motion-vs-rest/>

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users, services, or devices that interact with Green Pace’s systems. This policy outlines the use of strong authentication mechanisms to ensure only authorized entities gain access. |
| Authorization | Authorization determines the resources a user, service, or device is permitted to access. This policy ensures that permissions are granted according to the principle of least privilege, limiting access rights to the minimum necessary to perform legitimate functions. |
| Accounting | Accounting involves tracking and recording all user activities on systems, particularly those that access sensitive information. This policy aids in maintaining comprehensive logs to support effective monitoring and forensic analysis. |

Source: <https://www.ccsinet.com/blog/aaa-identity-management/>

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 03/21/2024 | Module 3 Milestone | Vincent Valente |  |
| 1.2 | 04/11/2024 | Module 6 Milestone | Vincent Valente |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |