**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## 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 | We validate input data to ensure it matches the expected formats and values for the specific system. If input isn’t properly managed and validated, it can expose a system to vulnerabilities such as injection attacks or buffer overflows. |
| 1. Heed Compiler Warnings | Compiler warnings are able to highlight potential issues within code at an early stage and prevent those issues from becoming much worse if left unchecked. We can use the compiler to mitigate the risk of memory leaks and buffer overflow attacks. This is simply a matter of acknowledging a tool that is at the disposal of the developer. |
| 1. Architect and Design for Security Policies | Involving security into the very architecture of a system means to include security within the design of the system. This means to be as forward thinking as possible and not reactive. We should not wait for vulnerabilities to expose themselves and then deal with them as they appear. Secure design mitigates risk before the system is even exposed to those risks. |
| 1. Keep It Simple | Keeping it simple means avoiding creating complex or even overly cryptic code, especially when it is entirely avoidable. The simplicity of the code is not just beneficial to its creator, but others as they may need to interact with it as well. If there is a pressing security matter, it is all the better if the relevant code can be easily scanned and adjusted. |
| 1. Default Deny | To default to deny means to only allow access when express permission is accepted. The idea behind this methodology is to limit exposure to the absolute bare minimum. If our system was a building, default deny would look like a shut door that would not allow anyone in who did not ask and verify with the security guard. If access permission is not requested, entry is not even considered. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege is similar to our last definition of default deny. Here all users are given access to absolutely nothing more than what is necessary for their specific role. So, even if the user of the system is an employee, they are only given access to what they need, nothing more. This limits exposure to the system. Whether that employee makes a mistake, is tricked via a phishing email, or becomes disgruntled and has mal intent towards the system, employing this principle helps combat those risks. |
| 1. Sanitize Data Sent to Other Systems | Data shared among multiple systems exposes the data and systems to greater risk with each additional system included. Different systems often have different security protocols and varying quality in the same regard. Sanitizing the data can aid in maintaining valid and safe data is transferred. |
| 1. Practice Defense in Depth | Defense in depth aims to employ a multi layered security approach. The idea is to have multiple forms of security so that you’re not putting all of your eggs in one basket. If one layer is breached, another is there to hopefully catch the issue. Multiple barriers make it more difficult for attackers to breach the defense. |
| 1. Use Effective Quality Assurance Techniques | This involves techniques such as automated testing, code reviews, and static analysis to mitigate risk and catch issues early on in development. Quality assurance can be carried out by dedicated testers and developers alike. |
| 1. Adopt a Secure Coding Standard | Using a secure coding standard like SEI CERT C++ promotes consistency during development and reduces the chances for vulnerabilities. These standards offer guidelines that address common errors made by developers. |

### 

### 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** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-001-CPP] | Using improper data types introduces the risks of memory corruption and buffer overflow. Using correct data types for variables and functions helps mitigate that risk. |

| **Noncompliant Code** |
| --- |
| A buffer without a check on the size risks buffer overflow if the input is greater than that buffer. |
| char buffer[10];  strcpy(buffer, input); // Input could be larger than buffer size (Danger) |

| **Compliant Code** |
| --- |
| We can use strncpy() to prevent writing beyond that buffer size. |
| char buffer[10];  strncpy(buffer, input, sizeof(buffer) - 1); // Input is limited to buffer size  buffer[9] = '\0'; // Ensures null termination |

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

| **Principles(s):** P1 Validate Input Data, directly applicable given it ensures data values meet expected formats, preventing buffer overflow and memory corruption.  P10 Adopt a secure coding standard will promote the consistent use of data types and reduce common errors and vulnerabilities involving data management. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P4 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S5863 | Detects risky strcpy along with similar functions at risk of buffer overflow |
| Cppcheck | 2.7 | Buffer overflow | Identifies buffer overflow vulnerabilities with detailed reporting |
| Clang-Tidy | 12.0 | cppcoreguidelines-pro-bounds-array-to-printer-decay | Aids in finding improper array to pointer conversions |
| CodeQL | Latest | cpp/buffer-overflow | Finds and flags buffer overflows and unsafe memory operations in C++ |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-002-CPP] | Another that helps prevent buffer overflows along with out-of-bound errors and more. If data values are unchecked this can lead to programs accessing data they otherwise would not, which can result in crashes or exploits. We should make sure data values are within expected ranges. |

| **Noncompliant Code** |
| --- |
| There are no bounds checking on input, which can lead to out-of-bounds memory access. |
| int index = user\_input;  array[index] = 100; // Does not check the boundary on the index (Danger) |

| **Compliant Code** |
| --- |
| Checking if the input is within the array bounds before the operation is carried out. |
| int index = user\_input;  if (index >= 0 && index < array\_size) {  array[index] = 100; // Validated index  } |

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

| **Principles(s):** P1 Validate Input Data to ensure it is verified to match expected data types and ranges. This will help reduce the risk of out-of-bounds errors and unauthorized access to memory.  P6 Adhere to the principle of least privilege which will control data and enforce strict data types. This principle limits the scope of what can be processed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P3 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S5448 | Detects out-of-bounds access |
| Cppcheck | 2.7 | bounds-check | Highlights code where potential out-of-bounds array access exists |
| Clang-Tidy | 12.0 | cppcoreguidelines-bounds-array-to-pointer-decay | Ensures array bounds are maintained/flags risky array pointer casts |
| Coverity Scan | Latest | out-of-bounds | Analysis of indexing operations to prevent out-of-bounds issues |

**Coding Standard 3**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Handling strings properly will prevent issues and vulnerabilities like string truncation, improper null-termination, and again, buff overflows. String manipulation can also lead to unauthorized access, which is a major security concern. |

| **Noncompliant Code** |
| --- |
| String is left unchecked |
| char buffer[10];  strcpy(buffer, input); // Input may be too large for buffer (Danger) |

| **Compliant Code** |
| --- |
| We can use strncpy() again to limit the string to the buffer size. |
| char buffer[10];  strncpy(buffer, input, sizeof(buffer) - 1); // Limits the copy to the buffer size  buffer[9] = '\0'; // Ensure null termination |

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

| **Principles(s):** P1 Validate input data will allow for proper string validation which ensures inputs meet required formats. This reduces risks such as buffer overflow.  P10 Adopt a secure coding standard to consistently handle strings. This will aid in preventing common vulnerabilities within the application. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | P2 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S5863 | Detect unsafe string functions leading to buffer overflow |
| Cppcheck | 2.7 | strcpy warnings | Finds unsafe string manipulation functions without bounds checking |
| Clang-Tidy | 12.0 | cppcoreguidelines-avoid-c-arrays | Flags risky C-style string manipulation |
| CodeQL | Latest | Cpp/unsafe-string-copy | Flags unchecked string operations risking overflow/truncation |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL Injection can occur if user input is directly embedded into SQL queries without the necessary sanitization or escaping, which allows the attacker to potentially access unauthorized data. This could involve deleting records and taking control of a database, a major security concern. |

| **Noncompliant Code** |
| --- |
| Here, the user input is inserted into the SQL query, leaving it vulnerable to modification. |
| std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| We can parameterize user input securely into SQL queries. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  stmt->prepare(query);  stmt->bind(1, user\_input); |

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

| **Principles(s):** P1 Validate input data to ensure that only expected values are processed in SQL queries, which will block injection vectors.  P7 Sanitize data sent to other systems before sending it to SQL queries prevents injection attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | P1 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Sql-injection | Finds patterns indicating sql-injection vulnerabilities |
| cppcheck | 2.7 | Sql injection | Flags potential sql injection issues where user input is concatenated into queries |
| Clang Static Analyzer | 12.0 | security.SQLInjection | Analyzes code for injection risk with a focus on dynamic sql statements |
| Fortify Static Code Analyzer | Latest | SQL Injection | Analyzes and detects for sql injection vulnerabilities in C++ |

#### 

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Handling memory management correctly is critical so a system can avoid data leaks, buffer overflows, and other serious vulnerabilities. We can prevent crashes and attackers alike with proper memory management. |

| **Noncompliant Code** |
| --- |
| Memory must be properly released, or we risk memory leaks. |
| char\* buffer = new char[1024];  // Forgetting to delete buffer (Danger) |

| **Compliant Code** |
| --- |
| Safely managing memory via deallocation after use |
| char\* buffer = new char[1024];  delete[] buffer; // Cleanup of allocated memory |

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

| **Principles(s):** P8 Practice defense in depth to reduce the risk of data leaks and more appropriately control memory access.  P10 Adopt a secure coding standard for consistent memory management practices which will comply with guidelines and minimize vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P3 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S1871 | Detects memory leaks via unfreed memory allocations |
| Cppcheck | 2.7 | memory leaks | Highlights memory allocations not released |
| Valgrind | Latest | memcheck | Detects memory leak during runtime along with dynamic memory errors |
| Clang Static Analyzer | 12.0 | unix.Malloc | Flags issues with dynamic memory allocation |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions can aid in handling unexpected conditions by verifying conditions that should always be true, but misuse can lead to crashing and insufficient error handling. |

| **Noncompliant Code** |
| --- |
| Assertion is checking a condition that may fail |
| assert(ptr != nullptr); // Can cause crash in release builds (Danger) |

| **Compliant Code** |
| --- |
| Validate assumptions without overreliance on assertions. |
| if (ptr == nullptr) {  // Handle the error appropriately  } |

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

| **Principles(s):** P4 Keep it simple in that proper use of assertions keeps code straightforward. Complex error handling could introduce vulnerabilities, where simple code with assertions can aid in early error detection.  P9 Use effective quality assurance techniques, where assertions allow developers to find issues early by verifying conditions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | P4 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S1764 | Flags misuse of assertions |
| Cppcheck | 2.7 | Assertions | Flags misuse of assertions that could lead to crashes |
| Clang-Tidy | 12.0 | Cppcoreguidelines-assert | Highlights areas where assertions are wrongly used in release builds |
| PVS-Studio | Latest | V3001 | Finds assertions that could be replaced with better error handling |

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP | Exception handling helps ensure that runtime errors are caught and managed properly. Unhandled exceptions within a system can expose sensitive info and leave the system vulnerable. |

| **Noncompliant Code** |
| --- |
| Thrown exceptions are not being properly handled. |
| throw std::runtime\_error("Error occurred"); // No exception handling (Danger) |

| **Compliant Code** |
| --- |
| Exceptions are caught and handled to prevent a crash. |
| try {  // Code that may throw an exception  throw std::runtime\_error("Error occurred");  } catch (const std::exception& e) {  // Handle the exception  std::cerr << "Exception caught: " << e.what() << std::endl;  } |

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

| **Principles(s):** P9 Use effective quality assurance techniques as proper handling of exceptions aids in managing unexpected conditions. This could include crashes and protect sensitive information with errors being handled in a controlled manner.  P10 Adopt a secure coding standard promotes following standardized exception handling, which leads to more predictable behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P3 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp:S2488 | Flags unhandled exceptions/helps manage runtime errors |
| Cppcheck | 2.7 | Exceptions | Finds unhandled exceptions |
| Clang-Tidy | 12.0 | Cppcoreguidelines-avoid-throwing | Highlights code that throws exceptions without a catch mechanism |
| PVS-Studio | Latest | V601 | Checks for exception best practices, flags risky patterns |

#### 

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Validation** | [STD-008-CPP] | Validating input ensures that the data given by users or other sources is verified/checked before it is processed. If we do not properly validate input our system will be exposed to injection attacks and buffer overflow. This could corrupt data or compromise the system as a whole. |

| **Noncompliant Code** |
| --- |
| This code accepts user input without validating it. |
| std::cin >> input; // Unchecked input (Danger) |

| **Compliant Code** |
| --- |
| This code validates the user input, ensuring it meets the necessary criteria. |
| std::cin >> input;  if (isValid(input)) {  // Process input safely  } |

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

| **Principles(s):** P1 Validate input data ensures input is validated before processing. This will protect the system from injection attacks and buffer overflows.  P5 Default deny to ensure only validated data is processed, blocking untrusted inputs by default. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | P1 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp:S5883 | Detects insufficient input validation |
| Cppcheck | 2.7 | Input validation | Flags input handling issues where validation is missing/poor |
| Clang Static Analyzer | 12.0 | Security.UncheckedReturnValue | Warns of unchecked user input potentially leading to injections/buffer overflow |
| Fortify Static Code Analyzer | Latest | Input Validation | Offers analysis for validation gaps with user input |

**Coding Standard 9**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Deallocation** | [STD-009-CPP] | Deallocating memory ensures that dynamically allocated memory is released after use. We employ this technique to avoid memory leaks which result in performance loss and crashes. |

| **Noncompliant Code** |
| --- |
| Memory is allocated but not released |
| int\* data = new int[100];  // No delete statement (Danger) |

| **Compliant Code** |
| --- |
| Using delete[] we release the data after use. |
| int\* data = new int[100];  delete[] data; // Deallocate memory |

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

| **Principles(s):** P8 Practice defense in depth, where adding in proper memory deallocation adds a protective layer against memory leaks.  P10 Adopt a secure coding standard for consistent memory management that aligns with best practices. This will help minimize security and performance related issues. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P2 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S1871 | Detects unreleased memory allocations |
| Cppcheck | 2.7 | Memory leaks | Flags code of allocated memory not being deallocated |
| Valgrind | Latest | Memcheck | Runtime analysis for dynamic memory errors/memory leak |
| Clang Static Analyzer | 12.0 | Unix.Malloc | Finds memory allocation without deallocation/flags potential leaks |

**Coding Standard 10**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Secure File Handling** | [STD-010-CPP] | Through this standard we can ensure files are handled in a safe way. If files are managed improperly, they can expose private information or allow users unauthorized access to them. File permissions must be established. |

| **Noncompliant Code** |
| --- |
| This code does not check for errors prior to opening the file, or at all for that matter. |
| std::ifstream file("data.txt"); (Danger) |

| **Compliant Code** |
| --- |
| Here we check for errors when opening the file and establish a way for the program to handle an error. |
| std::ifstream file("data.txt");  if (!file) {  // Handle the error  std::cerr << "Error opening file" << std::endl;  } |

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

| **Principles(s):** P6 Adhere to the principle of least privilege will control file permissions and restrict access to only those who absolutely need the access.  P7 Sanitize data sent to other systems will ensure that data handled by the file system is secure and validated, preventing unauthorized access. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P3 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S2091 | Finds unsafe file operations |
| Cppcheck | 2.7 | File handling | Finds improper file handling practices such as missing error checks |
| Clang-Tidy | 12.0 | security.FileAccess | Warns about insecure file access/poor permission handling |
| Fortify Static Code Analyzer | Latest | File Handling | Analysis for secure file handling in C++ |

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.

Green Pace’s DevSecOps pipeline consists of standards for pre-production and production that will ensure security is integrated through development and deployment. This focus will reduce the potential for vulnerabilities and leave the system more robust.

**Asses and plan**

Let’s begin with discussing pre-production’s Assess and plan portion. Automation will allow for continuous monitoring of threats, but it will also update risk models and adjust to regulation automatically. In the process, we’ll be able to identify new risks early on and plan mitigation strategies.

**Design**

In the design phase automated tools can analyze the structure and logic of our software during creation. Tools include Cppcheck and SonarQube, which can scan the code and detect vulnerabilities like SQL injection risks. If these issues are found early, we can significantly combat serious flaws down the road.

**Build**

For our build phase, we can use a tool such as Clang-Tidy to automatically review our code when it is updated. Each time the software is compiled, Clang-Tidy will offer immediate feedback on potential problems such as unsafe code patterns. The idea is to address vulnerabilities before testing and production.

**Verify and test**

The final phase of pre-production is verification and testing. This includes automated vulnerability scanning and testing solutions like CodeQL, which are integrated to continuously run security tests. CodeQL or Fortify can help identify vulnerabilities and run compliance checks. Given the use of these tools we will meet standards before deployment.

**Transition and health check**

Now we move on to the right side of the diagram into the transition and health check phase. During this we move into production and automated tools ensure security baselines are met. They will deploy monitoring tools, verify security checks, and schedule penetration tests to get an understanding of the security measures.

**Monitor and detect**

Following this begins the monitor and detect phase where tools track the environment for potential threats. SIEM systems such as ELK Stack allow for real-time alerting, anomaly detection, and logging. Here we can find signs of unauthorized access and have the ability to quickly respond.

**Respond**

That takes us to the next phase, being respond. Given a security issue has occurred, automation like

Failover protocol and rollback systems help us return to a secure state. The human security team is alerted in tandem allowing for a joint effort of automation and manual solutions; ultimately leading to restoring functionality quickly and efficiently.

**Maintain and stabilize**

Once response has occurred the next objective and final phase of the ongoing cycle is to maintain and stabilize. Automation will support maintenance of the system security and enforce baselines along with vulnerability testing on the regular with scheduling. Automation of updating improves systems robustness and adaptability to newly presented threats and vulnerabilities. This stage consists of issue prevention and mitigation strategy, where issues are to be found before they can escalate.

### 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-CPP | High | Unlikely | Medium | P4 | 2 |
| STD-002-CPP | High | Medium | Low | P3 | 3 |
| STD-003-CPP | High | High | Low | P2 | 3 |
| STD-004-CPP | Critical | High | Low | P1 | 3 |
| STD-005-CPP | High | Medium | Medium | P3 | 3 |
| STD-006-CPP | Medium | Medium | Low | P4 | 3 |
| STD-007-CPP | High | Medium | Medium | P3 | 3 |
| STD-008-CPP | Critical | High | Low | P1 | 3 |
| STD-009-CPP | High | Medium | Low | P2 | 3 |
| STD-010-CPP | High | Medium | Medium | P3 | 3 |

### 

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 | At rest means to protect the data that is stored on physical media like a hard drive or solid-state drive. This protection involves preventing unauthorized access. AES-256 encryption is used for sensitive data like user credentials. The use of this is crucial in maintaining data protection and confidentiality. |
| Encryption in flight | While at rest involves data that is static, in flight involves data that is moving across network and must be protected from interception by unauthorized users. Green Pace uses TLS protocols to encrypt data moving between clients and servers to safeguard info such as sensitive financial data and passwords. This policy is applied to all data communications. |
| Encryption in use | In use defends the data while it is processing by encrypting it while it is being used. Green Pace uses secure computation protocols along with hardware-based encryption. This ensures data is protected during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of a user accessing Green Pace systems. Multi-factor authentication (MFA) is required for all logins. Implementation of a strong password policy, along with biometric checks and one-time passwords help ensure only the right users gain access to the system. |
| Authorization | Authorization restricts access based on the user’s role, known as role-based access control (RBAC). This is paired with the principle of least privilege which aims to grant only the absolutely necessary access to all users based on their specific needs. This will minimize the potential for intentional and unintentional risks to the system by limiting overall access. |
| Accounting | Accounting tracks and records user activity. This activity includes logging in, accessing data, and making changes. The aim is to comprehensively log for security and compliance purposes. Green Pace is able to maintain a record of access and actions across the system. |

**\***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 | 09/19/2024 | Module 3 | Gavin Bliss |  |
| 1.2 | 10/10/2024 | Project One | Gavin Bliss |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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