**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 | Any data coming into a system, whether from users, files, or the network needs to be checked before it's used. If we don’t validate input properly, it can lead to security issues like buffer overflows or injection attacks. It’s important to make sure input data is the right type, length, and format so the program doesn’t do anything unexpected or unsafe. |
| 1. Heed Compiler Warnings | When the compiler gives warnings, it’s usually trying to tell you something’s not quite right. It might not stop the code from running, but it could still lead to bugs or security issues later. It’s better to fix these early before they turn into bigger problems. |
| 1. Architect and Design for Security Policies | Security should be part of the plan from the beginning. If you design your system with security in mind, like who gets access to what and how data is protected, it makes everything safer and more solid overall. |
| 1. Keep It Simple | The simpler your code and design, the less likely you are to run into issues. Complex setups are harder to manage and can hide bugs or holes in security. Keeping things straightforward makes it easier to spot problems and fix them. |
| 1. Default Deny | Don’t just allow access to everything by default. It’s safer to block everything at first and only allow what’s actually needed. That way, you’re not accidentally giving away access you shouldn’t. |
| 1. Adhere to the Principle of Least Privilege | Only give users or programs the access they really need. The less they can do, the less damage they can cause if something goes wrong. It’s a good way to limit the risk. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to another system, like a database or website, you should clean it up. That keeps bad data or code from getting through and causing issues on the other end. |
| 1. Practice Defense in Depth | It’s not enough to rely on just one security measure. Using several layers of protection means if one thing fails, there’s something else in place to catch it. It’s all about having backup security. |
| 1. Use Effective Quality Assurance Techniques | Testing and checking your code helps catch bugs and security problems early. Using tools and doing code reviews can help you catch things before they become real issues. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard helps you write safer code. It’s basically a list of good practices that helps avoid common mistakes and keeps things consistent across your project. |

### 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 Type** | [STD-nnn-LLL] | Use appropriate types for the data being represented. Choosing the correct data type helps prevent unexpected behavior or vulnerabilities like overflows or type confusion. Using a char to hold integers or mixing signed/unsigned types can cause logical or security issues. |

| **Noncompliant Code** |
| --- |
| This code uses the wrong data type and can cause overflow problems. |
| char count = 300; // char doesn’t support values this high |

| **Compliant Code** |
| --- |
| This code uses a proper type that can hold the value safely. |
| int count = 300; |

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

| **Principles(s):** **Validate Input Data** – using the right data type makes sure the input doesn’t break the program **Keep It Simple** – keeps bugs from happening because of bad type choices |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cppcoreguidelines-pro-type-member-init | |  | | --- | | Helps find type-related problems |  |  | | --- | |  | |
| SonarQube | 9.9 | |  | | --- | | c:S3511 |  |  | | --- | |  | | |  | | --- | | Flags risky type usage |  |  | | --- | |  | |
| CodeQL | Latest | |  | | --- | |  |  |  | | --- | | cpp/type-confusion | | Looks for unsafe type conversions |
| Cppcheck | 2.13 | -- inconclusive | Detects unsafe type assignments and casts |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Don’t use magic numbers, name your constants** |
| --- | --- | --- |
| **Data Value** | [STD-nnn-LLL] | Don’t use magic numbers and name your constants. Hardcoded numbers make code confusing and harder to change later. Naming them as constants makes the code clearer and easier to maintain. |

| **Noncompliant Code** |
| --- |
| This code uses a number without saying what it means. |
| if (score > 75) {  std::cout << "Pass\n";  } |

| **Compliant Code** |
| --- |
| This code uses a constant with a clear name. |
| const int PASS\_THRESHOLD = 75;  if (score > PASS\_THRESHOLD) {  std::cout << "Pass\n";  } |

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

| **Principles(s):** **Keep It Simple** – names make the code easier to understand **Use Effective Quality Assurance Techniques** – constants make testing and updates easier |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | |  | | --- | |  |  |  | | --- | | readability-magic-numbers | | Finds magic numbers in code |
| |  | | --- | | SonarQube |  |  | | --- | |  | | 9.9 | |  | | --- | | cpp:S109 |  |  | | --- | |  | | Warns when constants aren't named |
| |  | | --- | | PVS-Studio |  |  | | --- | |  | | 7.25 | V567 | Detects hardcoded numeric literals |
| Cppcheck | 2.13 | style | Identifies readability issues |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-nnn-LLL] | Don’t use unsafe string functions like strcpy. Using unsafe string functions can lead to buffer overflows and memory errors. Safer functions help protect the system and prevent data corruption. |

| **Noncompliant Code** |
| --- |
| This code uses strcpy and can overflow the buffer. |
| char dest[10];  strcpy(dest, "This is a long string"); // Overflows! |

| **Compliant Code** |
| --- |
| This version limits how much gets copied and adds a null terminator. |
| char dest[10];  strncpy(dest, "This is a long string", sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems** – keeps memory safe from bad input **Practice Defense in Depth** – adds another layer of protection |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-err34-c | Flags risky string operations |
| Fortify | 22.1 | STR01-C | Looks for buffer overflows from strings |
| PVS-Studio | 7.25 | V575 | Detects unsafe string usage |
| Cppcheck | 2.13 | --enable=all | |  | | --- | |  |  |  | | --- | | Flags insecure string handling | |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-nnn-LLL] | Protect your code from SQL injection. SQL injection happens when user input gets added to a database query without being cleaned up or handled properly. Hackers can mess with the input to run their own SQL commands, which could let them steal or change data. The safest way to prevent this is to use parameterized queries and validate input before sending it to the database. |

| **Noncompliant Code** |
| --- |
| This code lets user input go straight into the SQL query. It’s wide open to attacks. |
| std::string query = "SELECT \* FROM users WHERE username = '" + input + "'";  db.execute(query); |

| **Compliant Code** |
| --- |
| This version uses a parameterized query, which treats the input as data, not code. |
| sqlite3\_stmt\* stmt;  std::string query = "SELECT \* FROM users WHERE username = ?";  sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, input.c\_str(), -1, SQLITE\_TRANSIENT);  sqlite3\_step(stmt);  sqlite3\_finalize(stmt); |

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

| **Principles(s):** **Validate Input Data** – user input should never be trusted blindly **Sanitize Data Sent to Other Systems** – always clean and isolate user input |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Very Likely | Medium | Very High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | |  | | --- | |  |   cert-env33-c | Promotes secure use of external data |
| SonarQube | 9.9 | cpp:S3649 | |  | | --- | |  |  |  | | --- | | Flags unsafe string concatenation inSQL | |
| PVS-Studio | 7.25 | |  | | --- | |  |  |  | | --- | | V1001 | |  | | |  | | --- | |  |   Looks for suspicious SQL string patterns |
| Fortify | 22.1 | SQLi | Detects unvalidated or unparameterized queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-nnn-LLL] | Don’t use null or dangling pointers. Using a pointer that’s null or pointing to something that’s been deleted can crash your program or cause security bugs. Always check your pointers and don’t access memory that’s no longer valid. |

| **Noncompliant Code** |
| --- |
| This pointer is deleted and then used again. |
| int\* num = new int(5);  delete num;  std::cout << \*num; // undefined behavior |

| **Compliant Code** |
| --- |
| This version avoids using a pointer after deletion. |
| int\* num = new int(5);  delete num;  num = nullptr; |

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

| **Principles(s):** **Keep It Simple** – safe pointer handling avoids crashes **Practice Defense in Depth** – reduces memory vulnerabilities |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | misc-dangling-handle | Detects use-after-delete |
| SonarQube | 9.9 | Cpp:S2259 | Finds null dereferences |
| PVS-Studio | 7.25 | V769 | |  | | --- | |  |  |  | | --- | | Catches use of freed memory | |
| Cppcheck | 2.13 | nullPointer | |  | | --- | |  |  |  | | --- | | Detects null pointer dereference | |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-nnn-LLL] | Use assertions to catch problems early. Assertions are like checkpoints. They help you catch mistakes while you’re developing so the program doesn’t quietly keep going when something’s wrong. If something that *should* always be true ends up false, an assertion stops the program so you can fix it right away. |

| **Noncompliant Code** |
| --- |
| This code assumes size is never zero—but doesn’t check. |
| int average = total / size; // if size is 0, crash! |

| **Compliant Code** |
| --- |
| This version uses an assertion to make sure size is valid. |
| assert(size != 0);  int average = total / size; |

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

| **Principles(s):** **Use Effective Quality Assurance Techniques** – helps find problems while testing **Keep It Simple** – simple checks lead to fewer surprises |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | misc-static-assert | Flags risky unchecked assumptions |
| SonarQube | 9.9 | cpp:S5703 | |  | | --- | |  |  |  | | --- | | Suggests adding assertions where needed | |
| Cppcheck | 2.13 | uninitvar + checkOther | |  | | --- | |  |  |  | | --- | | Detects unsafe use without validation | |
| PVS-Studio | 7.25 | V713 | |  | | --- | |  |  |  | | --- | | Identifies places missing safety checks | |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-nnn-LLL] | Always code for what you expect to happen, and what you don’t. It’s not enough to hope your code works, you’ve got to plan for when it doesn’t. Expect things to go wrong: files might not open, inputs could be wrong, or connections might fail. If your code handles failure well, it’s going to be more reliable and safer. |

| **Noncompliant Code** |
| --- |
| This code assumes the file will open fine, with no checks. |
| std::ifstream file("data.txt");  file >> data; |

| **Compliant Code** |
| --- |
| This version checks that the file is ready before using it. |
| std::ifstream file("data.txt");  if (file.is\_open()) {  file >> data;  } else {  std::cerr << "Couldn’t open the file." << std::endl;  } |

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

| **Principles(s):** **Architect and Design for Security Policies** – handles errors safely **Validate Input Data** – don’t assume things will always work perfectly |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | bugprone-throw | Warns when code doesn’t properly handle errors |
| SonarQube | 9.9 | cpp:S3626 | |  | | --- | |  |  |  | | --- | | Flags code that assumes success blindly | |
| PVS-Studio | 7.25 | V595 | Finds unhandled return values |
| Cppcheck | 2.13 | checkReturn | Highlights missing error handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Pointer Safety** | [STD-nnn-LLL] | Don’t use null or dangling pointers. Using a pointer that’s null or pointing to something that’s been deleted can crash your program or cause security bugs. Always check your pointers and don’t access memory that’s no longer valid. |

| **Noncompliant Code** |
| --- |
| This pointer is deleted and then used again. |
| int\* num = new int(5);  delete num;  std::cout << \*num; // undefined behavior |

| **Compliant Code** |
| --- |
| This version avoids using a pointer after deletion. |
| int\* num = new int(5);  delete num;  num = nullptr; |

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

| **Principles(s):** **Keep It Simple** – safe pointer handling avoids crashes **Practice Defense in Depth** – reduces memory vulnerabilities |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | misc-dangling-handle | Detects use-after-delete |
| SonarQube | 9.9 | Cpp:S2259 | Finds null dereferences |
| PVS-Studio | 7.25 | V769 | |  | | --- | |  |  |  | | --- | | Catches use of freed memory | |
| Cppcheck | 2.13 | nullPointer | |  | | --- | |  |  |  | | --- | | Detects null pointer dereference | |

**Coding Standard 9**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Encryption | [STD-nnn-LLL] | Encrypt sensitive data before storing or sending it. If you’re storing or sending personal or private data, it needs to be protected. Encryption turns that data into something unreadable unless you have the key. Without it, attackers could steal passwords, credit card numbers, or personal info. |

| **Noncompliant Code** |
| --- |
| This stores a password as plain text. That’s bad. |
| std::ofstream file("passwords.txt");  file << password; |

| **Compliant Code** |
| --- |
| This version encrypts the password before saving it. |
| std::string encrypted = encryptAES(password, key);  std::ofstream file("passwords.txt");  file << encrypted; |

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

| **Principles(s):** **Default Deny** – don’t assume data is safe unless it’s protected **Adopt a Secure Coding Standard** – follow best practices for data security |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | Very High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.1 | Crypto01 | Flags weak or missing encryption |
| SonarQube | 9.9 | cpp:S5542 | |  | | --- | |  |  |  | | --- | | Looks for unencrypted sensitive data | |
| Clang-Tidy | 15.0 | cert-msc51-cpp | |  | | --- | |  |  |  | | --- | | Recommends strong cryptographic libraries | |
| PVS-Studio | 7.25 | |  | | --- | |  |  |  | | --- | | V1064 | | |  | | --- | |  |  |  | | --- | | Detects unsafe handling of credentials | |

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Practice Defense in Depth | [STD-nnn-LLL] | Don’t rely on just one layer of protection. You should never count on just one security control to stop every threat. Think of it like layers, if one fails, the others are still there to catch problems. Add checks at different places: input validation, memory safety, access control, logging, etc. Together, they make the system much harder to break. |

| **Noncompliant Code** |
| --- |
| This only checks for a valid password once. If that fails, there's no backup. |
| if (password == "admin123") {  accessGranted();  } |

| **Compliant Code** |
| --- |
| This adds multi-layered checks: password, user role, and access level. |
| if (checkPassword(user, inputPassword) &&  userHasRole(user, "admin") &&  hasPermission(user, "access\_dashboard")) {  accessGranted();  } |

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

| **Principles(s):** **Practice Defense in Depth** – stack up protections so one mistake doesn’t break everything **Architect and Design for Security Policies** – make security part of the whole system |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S4529 | |  | | --- | |  |  |  | | --- | | Flags missing redundancy in access control | |
| Fortify | 22.1 | Auth01 + InputVal01 | Detects lack of layered protections |
| Clang-Tidy | 15.0 | |  | | --- | |  |  |  | | --- | | cert-dcl50-cpp | | |  | | --- | |  |  |  | | --- | | Warns when critical checks are missing | |
| PVS-Studio | 7.25 | |  | | --- | |  |  |  | | --- | | V1042 | | |  | | --- | |  |  |  | | --- | | Flags weak design that lacks fallback checks | |

### 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.

Automation will be integrated into the existing Green Pace DevOps process to ensure continuous enforcement of security standards. By embedding checks into each stage of the DevSecOps pipeline, security will not be treated as a final step but instead as an ongoing part of development. For example, static analysis tools like Clang-Tidy, SonarQube, PVS-Studio, and Cppcheck will run automatically during the build stage, flagging violations against our coding standards. Automated unit tests will validate that code revisions meet security requirements, while continuous integration pipelines will block builds that do not comply. In deployment, container and environment scanning tools will verify that configurations meet security policies. This layered automation ensures every commit, build, and deployment is reviewed for compliance, reducing human error and improving consistency across the development lifecycle.

### 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 | High | 2 |
| STD-002-CPP | Medium | Likely | Low | Medium | 3 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | Critical | Very Likely | Medium | Very High | 5 |
| STD-005-CPP | Critical | Likely | Medium | High | 5 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | High | Likely | Low | High | 4 |
| STD-008-CPP | Critical | Likely | Medium | High | 5 |
| STD-009-CPP | Critical | Likely | Medium | Very High | 5 |
| STD-010-CPP | High | Likely | 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.

|  |  |
| --- | --- |
| **a. Encryption** | **Explain what it is and how and why the policy applies.** |
| Encryption at Rest | Encryption at rest ensures that data stored on any Green Pace system—databases, backups, logs, or file systems, is protected from unauthorized access if storage media is stolen, lost, or compromised. AES-256 or stronger encryption must be used for structured data, and full-disk encryption must be applied to servers, laptops, and portable devices. This policy applies at all times to safeguard sensitive and personally identifiable information. |
| Encryption in Flight | Encryption in flight protects data while it is being transmitted between systems, applications, or users. All traffic, whether internal or external, must use secure communication protocols such as TLS 1.2 or higher, HTTPS, and secure VPNs. This prevents interception, tampering, and man-in-the-middle attacks. The policy applies whenever data is transmitted over networks, including local networks, cloud environments, and the internet. |
| Encryption in Use | Encryption in use ensures sensitive data remains protected while actively processed in memory. Data such as authentication tokens, encryption keys, and passwords must be stored only in secure memory locations. Techniques like memory isolation, trusted execution environments, and secure enclaves should be applied where available. This policy applies when applications are running and handling sensitive data to prevent unauthorized access through debugging, memory dumps, or insider threats. |

|  |  |
| --- | --- |
| **b. Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| Authentication | Authentication verifies that all users and systems accessing Green Pace resources are who they claim to be. Multi-factor authentication (MFA) must be enforced for user logins, and strong password policies must be followed. System-to-system communication must use certificates or secure tokens. This policy ensures that only authorized individuals and services can initiate access. |
| Authorization | Authorization ensures users and systems only have the exact access needed to perform their tasks, following the principle of least privilege. Role-based access control (RBAC) must be applied for managing access rights. All privileged actions, such as database changes or administrative tasks, must be limited to authorized personnel only. This policy prevents over-permissioning and reduces risks from compromised accounts. |
| Accounting | Accounting requires all activities to be logged and monitored. Logs must record login attempts, file access, changes to user accounts, database modifications, and administrative actions. These logs must be securely stored, protected from tampering, and retained for compliance and auditing. This policy ensures accountability and provides a clear trail for investigations in the event of a security incident. |

**\***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.

| Coding Standard | Principles Supported | Justification |
| --- | --- | --- |
| STD-001-CPP (Data Types) | 1, 4 | Validating input with correct data types prevents unexpected behavior; keeping data types simple reduces errors. |
| STD-002-CPP (Magic Numbers) | 4, 9 | Simplicity makes code easier to read; naming constants supports testing and quality assurance. |
| STD-003-CPP (String Correctness) | 7, 8 | Sanitizing data avoids unsafe memory handling; layered defenses protect against overflow. |
| STD-004-CPP (SQL Injection) | 1, 7 | Input validation and sanitization prevent attackers from injecting malicious SQL commands. |
| STD-005-CPP (Memory Protection) | 4, 8 | Safe memory handling avoids crashes; layered defenses reduce exploitation opportunities. |
| STD-006-CPP (Assertions) | 4, 9 | Assertions keep the code simple and catch errors early, supporting quality assurance. |
| STD-007-CPP (Exceptions) | 3, 1 | Designing for failure and validating input ensures systems remain safe when errors occur. |
| STD-008-CPP (Pointer Safety) | 4, 8 | Simplified safe pointer handling prevents crashes; multiple layers protect against memory misuse. |
| STD-009-CPP (Encryption) | 5, 10 | Default deny applies by assuming data is unsafe until encrypted; secure coding standards enforce cryptography. |
| STD-010-CPP (Defense in Depth) | 8, 3 | Layering protections ensures resilience; secure design integrates security throughout the system. |

**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 |  |
| 1.2 | 8/10/2025 | |  | | --- | |  |  |  | | --- | | Completed security principles,  coding standards, automation,  encryption, Triple-A, and mappings | | Ashley Kimrey | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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