**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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 12](#_Toc52464063)

[Coding Standard 6 14](#_Toc52464064)

[Coding Standard 7 16](#_Toc52464065)

[Coding Standard 8 18](#_Toc52464066)

[Coding Standard 9 20](#_Toc52464067)

[Coding Standard 10 22](#_Toc52464068)

[Defense-in-Depth Illustration 24](#_Toc52464069)

[Project One 24](#_Toc52464070)

[1. Revise the C/C++ Standards 24](#_Toc52464071)

[2. Risk Assessment 24](#_Toc52464072)

[3. Automated Detection 24](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 26](#_Toc52464076)

[7. Map the Principles 28](#_Toc52464077)

[Audit Controls and Management 29](#_Toc52464078)

[Enforcement 29](#_Toc52464079)

[Exceptions Process 29](#_Toc52464080)

[Distribution 30](#_Toc52464081)

[Policy Change Control 30](#_Toc52464082)

[Policy Version History 30](#_Toc52464083)

[Appendix A Lookups 30](#_Toc52464084)

[Approved C/C++ Language Acronyms 30](#_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 | All input that comes from a user should be validated to prevent program crashing bugs or exploits from occurring. Anywhere a user will be inputting data, there should be checks that verify the input and handle unexpected inputs. |
| 1. Heed Compiler Warnings | Do not assume that just because your code compiles that it is good code. Compiler warnings are there to help improve code and should be looked into. Addressing compiler warnings will only benefit your code. |
| 1. Architect and Design for Security Policies | Systems should be designed around security so that security is the goal and not an afterthought. This means security policies should be built into the system as it is created, not added at the end as a top-level layer. |
| 1. Keep It Simple | Code should not be unnecessarily complex. Adding unnecessary complexity allows for the introduction of bugs and even vulnerabilities that can be avoided if code is kept simple. |
| 1. Default Deny | When building systems that will contain sensitive information, access to them should be denied by default. Users that require access to these systems should be given this access explicitly, all other attempts to access them should be rejected. |
| 1. Adhere to the Principle of Least Privilege | When determining what data users should have access to, always give them the least access necessary to complete their work. It is easier to give someone additional access than it is to take access away from them. |
| 1. Sanitize Data Sent to Other Systems | Any data that is to be sent to other systems, for example SQL queries to a database, needs to be sanitized to prevent the chance of returning unauthorized data to users. Data should also be sanitized so that users can’t send arbitrary code to be run on the system the data is sent to. |
| 1. Practice Defense in Depth | Defense in depth is a strategy where there are multiple security methods used to protect a company’s data or assets. Practicing defense in depth provides safeguards where if one method gets compromised, the data will still be secure because there are other methods protecting it that are still secure. This reduces the risk of an attacker gaining access to all of the data if they were to compromise a single method. |
| 1. Use Effective Quality Assurance Techniques | Each new process and system created should go through a rigorous quality assurance phase that includes manual code review, unit testing, automated testing, and more. This will ensure that each new process or system created is checked for functionality and potential bugs. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard from the start of development will help prevent the introduction of vulnerabilities throughout the entirety of the software development lifecycle. A secure coding standard should be chosen and adhered to for the best results. |

### 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-001-CPP | Data types should be chosen based on the type of information they will be storing to best fit their needs. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the variables x and y are declared as integers, which is acceptable, and then used for division. The resulting value, z, is also declared as an int, which is not acceptable as the precision of the division is then lost. This should be avoided as it can result in values that are increasingly wrong the more times they are used in calculations. |
| int x;  int y;  int z;  z = x / y; |

| **Compliant Code** |
| --- |
| This compliant code declares the variables x and y as integers but variable z as a float so that the precision isn’t lost during the calculation. |
| int x;  int y;  float z;  z = x / y; |

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

| **Principles(s):** 3, 4. Designing the system for security includes choosing variable types that best fit what they will be used for. Keeping it simple also means not picking variables that are much larger than needed, for example a double instead of a float for a value that will never be greater than 100. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-810: Appropriate char types should be used for character and integer values, <https://rules.sonarsource.com/cpp/RSPEC-810> | This checker will flag noncompliant code that uses incorrect char data types. |
| Sonar Rules | 7.8 | RSPEC-5276: Implicit casts should not lower precision, <https://rules.sonarsource.com/cpp/RSPEC-5276> | This checker will flag code that uses implicit conversion to reduce data precision, resulting in potential loss of data. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | The values of all data inputs from the user should be verified prior to using it in any functions or returning any results. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the user inputs a value that is supposed to be positive, which is then directly used in another function. This allows the user to input whatever data they want without validating it is within the bounds of its use case. |
| String positiveNumber;  cout << “Enter a positive number: “ << endl;  cin >> positiveNumber >> endl;  doSomething (positiveNumber); |

| **Compliant Code** |
| --- |
| In this compliant code, the input from the user is checked to make sure it is positive before allowing the value to be used in a function. If the user doesn’t enter a positive value, then the program prompts the user for another value. |
| String positiveNumber;  do {  cout << “Enter a positive number: “ << endl;  cin >> positiveNumber >> endl;  } while (positiveNumber < 0);  doSomething (positiveNumber); |

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

| **Principles(s):** 1. Validating data that is being entered into the system is essential to ensuring that all functions will perform their tasks as they were designed and without unintentional side effects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-1081: Insecure functions should not be used, <https://rules.sonarsource.com/cpp/RSPEC-1081> | This checker is used to ensure that variables that could be manipulated by a user are properly checked to handle unexpected user inputs. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Strings should always be given sufficient memory for both the string data itself and the null terminator. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the user\_password variable is declared as an array of characters with a length of 10. This is unacceptable as users can enter more than 10 characters which could result in a buffer overflow of other data. |
| char user\_password[10];  cout << “Enter your password: “;  cin >> user\_password >> endl; |

| **Compliant Code** |
| --- |
| This compliant code updates user\_password to be a string instead of an array of characters. This prevents buffer overflows without limiting the length of the input from the user. Limitations on the length of user input will need to be enforced using other methods. |
| string user\_password;  cout << “Enter your password: “;  cin >> user\_password >> endl; |

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

| **Principles(s):** 3. Designing around security means ensuring that strings used in the program will not be altered due to a lack of memory allocation which could result in buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-1081: Insecure functions should not be used, <https://rules.sonarsource.com/cpp/RSPEC-1081> | This checker will flag lines of code where noncompliant functions that don’t check for bounds are used to get user input such as gets() or strcpy(). |
| Sonar Rules | 7.8 | RSPEC-3519: Memory access should be explicitly bounded to prevent buffer overflows, <https://rules.sonarsource.com/cpp/RSPEC-3519> | This checker will raise a flag if arrays or buffers have a chance to go beyond the boundary of what was allocated. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Failure to prevent SQL Injection attacks allows malicious actors to access information not meant for them and puts customer data at risk. All SQL queries should check for injection prior to running the query. |

| **Noncompliant Code** |
| --- |
| This noncompliant code does not check for injection of unauthorized parameters and queries what it is passed. |
| String username;  String sql\_query;  cin >> username >> endl;  sql\_query = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='" + username + “’”;  if (!run\_query(db, sql\_query, records)) return; |

| **Compliant Code** |
| --- |
| This compliant code checks the SQL query variable for any injection attempts before running the query. If a SQL injection is detected, the query should not be ran. |
| String username;  String sql\_query;  cin >> username >> endl;  sql\_query = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='" + username + “’”;  if (sql\_query.contains(“\’ OR “) || sql\_query.contains(“\’ or “)) {  return false;  }  else {  if (!run\_query(db, sql\_query, records)) return;  } |

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

| **Principles(s):** 6, 7. Adhering to the principle of least privilege ensures that even if a SQL injection is attempted, it will fail due to the username entered not having complete access to the database. Sanitizing data sent to other systems means ensuring that what the user entered doesn’t include arbitrary code that could be executed to return more data than what the user should have access to. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sqlmap | 1.8 | SQL Injection: <https://sqlmap.org/> | This tool will test the database server for vulnerabilities that allow a variety of SQL injection flaws to take place. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Memory should be protected in a way that ensures the program isn’t using more memory than necessary and is releasing memory that is no longer needed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code a new structure is declared using points and is never deleted after being used, therefore continuing to hold any memory it was using. |
| Void exampleFunc() {  Example \*example = new Example();  // use struct for some purpose  } |

| **Compliant Code** |
| --- |
| In this compliant code, the new example structure is used for its purpose and is then deleted once it is no longer necessary, freeing up any memory that was held by it. |
| Void exampleFunc() {  Example \*example = new Example();  // use struct for some purpose  delete example;  } |

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

| **Principles(s):** 2. Compilers will often provide a warning if the memory of an object is going to be used in an unusual way. These warnings are generated for a reason and should be investigated to ensure that memory will not be accessed when it is not supposed to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-1232: Appropriate memory de-allocation should be used, <https://rules.sonarsource.com/cpp/RSPEC-1232> | This checker will raise a flag if allocated memory is not being de-allocated properly. |
| Sonar Rules | 7.8 | RSPEC-3519: Memory access should be explicitly bounded to prevent buffer overflows, <https://rules.sonarsource.com/cpp/RSPEC-3519> | This checker will raise a flag if arrays or buffers have a chance to go beyond the boundary of what was allocated. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions should be used to verify that code we expect to be true, does in fact return true while the program is running. Assertions should be included wherever it is not obvious that a block of code will return the same result every time. |

| **Noncompliant Code** |
| --- |
| Noncompliant assertions should not have side effects. This noncompliant assertion updates a variable in its check, which doesn’t test the actual value. This second noncompliant assertion includes a function call in the statement, which can cause side effects if not careful. Both examples should be avoided. |
| ASSERT(variable++ > 0); |
| ASSERT(testFunc(0) == true); |

| **Compliant Code** |
| --- |
| In this compliant code, the program performs a calculation that will always result in returning a positive value. Using the assertion on this value verifies that the value is positive and will let us know if this function ever results in an unexpected value that is negative. |
| int testFunc(int num) {  return num \* num;  }  int value = testFunc(0);  ASSERT(value >= 0); |

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

| **Principles(s):** 9, 10. Assertions are an excellent way to use effective QA techniques as they test the code in places where we know a variable should always return a specific value. Assertions are also essential in a secure coding standard due to the information they can provide to us. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Low | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.14 | ASSERT\_EQ, ASSERT\_TRUE, <https://google.github.io/googletest/reference/assertions.html> | This checker will take the place of in code assertions so that there aren’t attempts at modifying variables in the code that wouldn’t exist in production. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions should be used to properly handle errors so that the program doesn’t crash when the code produces an unexpected output. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, an exception is not used, and the function produces a result that is outside the bounds of what is possible. This results in the program abruptly stopping when an exception could have been used to prevent this. |
| void function() {  function\_that\_fails();  } |

| **Compliant Code** |
| --- |
| This compliant code uses an exception to not only catch an error, but also to handle it in such a way that the program isn’t abruptly stopped and can continue to run. |
| void function() {  try {  function\_that\_fails();  } catch(…) {  // Handles the error  }  } |

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

| **Principles(s):** 8. Exceptions are a great way to add layers to an application in the event that unexpected data is entered by a user. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-2486 - Exceptions should not be ignored, <https://rules.sonarsource.com/cpp/RSPEC-2486> | This checker will raise a flag if there is an exception that is thrown and caught but not handled, or at the very least logged. |
| Sonar Rules | 7.8 | RSPEC-2698: Non-exception types should not be caught, <https://rules.sonarsource.com/cpp/RSPEC-3698> | This will check the code to see if any exceptions that are not derived from std::exception are being thrown as this is a bad practice. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Variable Comparison | STD-008-CPP | Variable types, such as float and double, that can be impacted by rounding should not be used directly for comparisons. Rounding can result in comparisons that should be true resulting in false. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, two floats are compared to each other in an if statement. Due to rounding, the if state results in false even though both variables are declared as 0.3. |
| float x = 0.3;  float y = 0.3;  if (x == y) {  do\_something();  } |

| **Compliant Code** |
| --- |
| In this compliant code, the absolute value of the difference is taken and compared to a very small number, in this case 1e-9, to show that the values are similar enough to be considered equal. Doing this protects against comparisons returning false for values that are assumed to be equal. |
| float x = 0.3;  float y = 0.3;  if (abs(x – y) < 1e-9) {  do\_something();  } |

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

| **Principles(s):** 4. Keeping code simple and not comparing variables that could possibly be stored differently in memory is a great way to prevent bugs from occurring. If the comparison is necessary, the comparison should be carefully crafted in a way that allows the comparison without directly comparing the two values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Solar Rules | 7.8 | RSPEC-1244: Floating point numbers should not be tested for equality, <https://rules.sonarsource.com/cpp/RSPEC-1244> | This checker will raise a flag if there is any comparison between floating point numbers. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integer Overflow | STD-009-CPP | Integers, and other numeric data types, have limits that can be reached if not handled carefully. Reaching these limits can result in the value returning incorrectly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, addition is performed using variables x and y without checking to see if the resulting value would be outside the bounds of what an integer is capable of being. This results in an incorrect value being returned. |
| int x;  int y;  int result;  result = x + y; |

| **Compliant Code** |
| --- |
| This compliant code checks to see if the resulting value would be greater than the bounds of an integer and if not, performs the calculation. If it were greater than the bounds of an integer, then the program would let the user know and would not perform the calculation. |
| int x;  int y;  int result;  if (INT\_MAX – x < b) {  cout << “Addition results in overflow.” << endl;  } else {  result = x + y;  } |

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

| **Principles(s):** 1. Validating that data entered by a user will fit inside the bounds of the variable type we have chosen will prevent any potential overflows from happening. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-3949: Integral operations should not overflow, <https://rules.sonarsource.com/cpp/RSPEC-3949/> | This checker will raise a flag if there is code where a value is outside of the bounds of the respective data type. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Accessing Freed Memory | STD-010-CPP | Memory that has been freed should not be overwritten or read from as it may still contain the previous contents. This can be exploited if not handled correctly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the structure is dereferenced after it has been deallocated. This can open up a vulnerability that allows code to be ran with the permissions of this process. |
| Struct Struct {  Void function();  }  void example() {  Struct \*struct = new Struct;  // Use Struct  Delete struct;  Struct->function();  } |

| **Compliant Code** |
| --- |
| This compliant code deallocates the structure after it has been dereferenced. Not deallocating the structure until after it has completed its use protects against the aforementioned vulnerability. |
| Struct Struct {  Void function();  }  void example() {  Struct \*struct = new Struct;  // Use Struct    Struct->function();  Delete struct;  } |

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

| **Principles(s):** 3. Designing for security in this situation means designing the program in a way that prevents the access of freed memory by other variables. This will protect any potential data that may still reside at that memory location. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar Rules | 7.8 | RSPEC-3529: Freed memory should not be used, <https://rules.sonarsource.com/cpp/RSPEC-3529/> | This rule flags the access of freed memory when the code is analyzed. |
| Sonar Rules | 7.8 | RSPEC-3520: Memory locations should not be released more than once, <https://rules.sonarsource.com/cpp/RSPEC-3520> | This checker will flag code that attempts to release or free memory from a location twice. Releasing the same memory twice results in undefined behavior. |

### 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.

There are multiple places where automation could be added to the DevOps process to detect and enforce the standards we have created. On the pre-production side, we can add automation both to the design and verification and testing steps. In the design step, we can utilize IDE extensions that will analyze our code for potential bugs and vulnerabilities. One example of an IDE extension that analyzes code is Sonarlint. The verification and testing step would benefit from automation as it will make it easier for developers to design and build new features without having to run unit tests on their code changes manually. By implementing a product such as SonarCloud or Selenium, we will be able to create and automate tests that would otherwise be repetitive, labor intensive, or time consuming to run manually.

On the production side of the DevOps process, we can add automation to the monitoring and detection and response steps. Monitoring and detection can be automated by sending logs from network devices, such as a firewall, to an analyzing tool that can efficiently sift through them to find dangerous and unwanted connections. This automated process can send an alert to the proper team that will allow them to investigate the findings further. The response step can be automated by having a tool that can reject unwanted or malicious connections or even quarantine devices that get infected before further infection can occur. An automated response here will give teams more time to investigate the logs to determine how the connection was made so that it can be corrected by the development team.

There are other steps within the DevOps process that automation could benefit, but starting with the ones listed will ensure that we have a strong base in the security of our applications.

### 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 | High | Probable | Medium | Medium | 2 |
| STD-003-CPP | High | Very Likely | Medium | High | 4 |
| STD-004-CPP | Extreme | Very Likely | High | Very High | 5 |
| STD-005-CPP | High | Likely | Low | Medium | 3 |
| STD-006-CPP | Medium | Probable | Low | Low | 2 |
| STD-007-CPP | Low | Probable | Medium | Low | 2 |
| STD-008-CPP | Medium | Unlikely | Low | Low | 2 |
| STD-009-CPP | Medium | Likely | Medium | Medium | 3 |
| 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.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is data that is currently being stored in one place and not being used. Is it data that has been altered in some way from its original form to make it unreadable to another person without the decryption key. This policy applies to all data that is stored in a location where it is not being used and should be used to protect sensitive information provided by customers such as passwords, banking information, Social Security Number, or anything else that could be used against the customer or the company. |
| Encryption in flight | Encryption in flight is the process of sending data in an encrypted format so that if it is intercepted by a third party, they are unable to use the data. Encryption in flight is done by ensuring both the sender and the receiver have a secure connection and that the data is encrypted using a well-known standard, such as TLS or SFTP. This makes it so only the sender and a receiver with a decryption key can see the data. This policy applies to any data sent from the database to clients. |
| Encryption in use | Encryption in use is the protection of data while it is being accessed by a user. Encryption in use is applied by securing the memory that is used to hold the data that is being accessed or even encrypting the data that is currently being used, such as hashing a password as soon as it is entered by the user. This policy needs to be applied because data in use is the most vulnerable data. It is often one of the few times when the data is in its plaintext form and therefore is a highly sought after target. Encrypting data in use will protect this data from attackers. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a user providing information about who they are to login to a system. Using authentication ensures that only specific people are able to access a system, such as user logins on a company’s network. Authentication is used to confirm that a user is who they say they are. This keeps information private so that it is only seen by those that need to be able to see it. |
| Authorization | Authorization is what allows a user to be able to see data or make changes to the data. This is applied by limiting who can make certain changes to a database and who has access to see different levels of information. Authorization can also be utilized to determine who has the ability to add new users to a system. Authorization is necessary because it limits what users have access to and also what they can do with the information they have access to. |
| Accounting | Accounting is keeping track of what is being accessed and who is accessing it. This policy is applied by keeping logs of what accounts are interacting with what documents or objects. It is necessary so that we know what files are accessed by users and so that if an unauthorized change is made to the database, we are able to go back to determine who made the change. |

**\***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 | 03/28/2024 | Added coding standards and examples of compliant and non-compliant code | Colton Berger | [Insert text.] |
| 1.2 | 04/13/2024 | Added Risk Assessment, Automation recommendations, Encryption standards and Triple A, and mapped principles to coding standards | Colton Berger | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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