**Green Pace Developer: Security Policy**



# Green Pace Secure Development Policy

## Contents

Green Pace Secure Development Policy 0

Contents 1

Overview 2

Purpose 2

Scope 2

Ten Core Security Principles 2

C/C++ Ten Coding Standards 4

Coding Standard 1 5

Coding Standard 2 6

Coding Standard 3 8

Coding Standard 4 10

Coding Standard 5 12

Coding Standard 6 14

Coding Standard 7 16

Coding Standard 8 18

Coding Standard 9 20

Coding Standard 10 22

Defense-in-Depth Illustration 24

Automation 24

Summary of Risk Assessments 25

Policies for Encryption and Triple A 25

Audit Controls and Management 26

Enforcement 26

Exceptions Process 26

Distribution 27

Policy Change Control 27

Policy Version History 27

Appendix A Lookups 27

Approved C/C++ Language Acronyms 27

## 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 creates, deploys, or supports custom software at Green Pace.

### Ten Core Security Principles

| **Principles** | Explanation |
| --- | --- |
| 1. ValidateInput Data | Data input validation refers to verifying that any input received by a software program is appropriately handled. This means that input of the correct type and format should be accepted and processed, and any input of an incorrect type or format should be identified by the system and produce an error indication. This prevents attacks such as SQL injection and potentially harmful actions such as buffer overflows from occurring. |
| 1. Heed Compiler Warnings | The compiler warns the developer when code is implemented that may cause the program to crash or otherwise produce unexpected behavior. These warnings, and the consequences of what they warn of, should be carefully analyzed to determine whether the code should be altered to prevent unwanted program behavior. |
| 1. Architect and Design for Security Policies | Security policies dictate what should or should not be possible to achieve while using the system, whether or not the user intends to use the system as designed. Because these policies must be followed to prevent unwanted and potentially malicious use of the system, it is often most cost effective to design the system with these policies in mind from the very earliest stages of development. |
| 1. Keep It Simple | When developing programs that are secure from attack, it is often more effective to design simple solutions rather than consider every possible threat individually. Separating program logic into several distinct pieces often creates opportunities for attack. As such, programs should be designed to prevent misuse as simply as possible. |
| 1. Default Deny | Programs should deny access to privileged information by default. A user should only be able to access a secure system after being properly authorized, and this authorization should be revoked when the user leaves any part of the system. This helps to prevent malicious users from bypassing authorization processes to gain access to secure systems. |
| 1. Adhere to the Principle of Least Privilege | Users of the system should only have access to information or resources that are required for them to complete a given task, and the data that they are given access to should only be offered after the user has been authenticated and authorized to receive it. This prevents even authenticated users from acting maliciously with data that they are not authorized to use. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization refers to the clearing or removal of sensitive information associated with collected data. It is critical that data is sanitized before being sent to other systems in order to prevent the interception of this sensitive data, and provides a form of compartmentalization that helps to ensure customer data is protected against attack regardless of where the attack takes place. |
| 1. Practice Defense in Depth | Defense in depth is the practice of creating multiple overlapping layers of security to defend organizations from attack. These layers work together to fill gaps in security that may exist by using only one method of security, and they work to protect all aspects of the business in a way that produces a security mechanism greater than the sum of its parts. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques help to ensure that software is up to industry standard and provide methods of detecting flaws in the system. These include design reviews, code inspections, functional testing, static analysis, and other similar techniques. |
| 1. Adopt a Secure Coding Standard | It is important to create and adopt a coding standard to be applied to all aspects of software within the system, which ensures that the system is developed in a way that leaves no particular area less secure. Additionally, all developers working on the system will have an idea of how other areas have been implemented in code, which helps to efficiently detect and resolve security issues in the system when they do exist. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | All data types should be implemented in ways that do not conflict with the other data types that they interact with, and additional data types defined by the developer should use similar naming conventions to improve maintainability. |

| **Noncompliant Code** |
| --- |
| In this example, an unsigned integer is assigned the value of 300. When an unsigned char is assigned with the value held by the unsigned int, it is reduced and no longer accurately represents the original data. |
| unsigned int uint = 300;  unsigned char uchar = uint; /\* the value held by the new variable uchar is not the same value held by uint \*/ |

| **Compliant Code** |
| --- |
| To protect the integrity of the data, same data types should be used to represent the same data whenever possible. |
| unsigned int uint\_1 = 300;  unsigned int uint\_2 = uint\_1; /\* the value held by the new variable uint\_2 is the same value held by uint\_1 \*/ |

| **Principles:**  Heed Compiler Warnings:  Compilers often warn of mismatched data types before an error is formed during the program runtime. Because of this, compiler warnings can act as tools to detect when data types are improperly implemented during development, and not after the code is in use.  Keep It Simple:  Refraining from complex data type conversions can help to prevent improperly implemented data types. Keeping the code base simple is an effective strategy to limit the number of bugs caused by data types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.IF  LANG.STRUCT.DECL.IO | Inconsistent function declarations  Inconsistent object declarations |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Developers should be careful to properly use unsigned and signed integer values wherever necessary. Failing to account for changes in the data value associated with a variable can cause unexpected program behavior. |

| **Noncompliant Code** |
| --- |
| In this example, an unsigned integer is decremented past the value 0, which causes the value to wrap. |
| unsigned int uint = 1;  uint = uint - 2; /\* the value held by uint can no longer be safely used as it has wrapped \*/ |

| **Compliant Code** |
| --- |
| Using a signed integer when the value is expected to eventually become negative prevents wrapping and protects the integrity of the data. |
| signed int sint = 1;  sint = sint - 2; /\* the value held by sint is a negative number as expected \*/ |

| **Principle:**  Validate Input Data:  Ensuring that input data matches a particular data type’s range can prevent instances where the value of data is unexpectedly changed by integer wrapping. Compilers may not warn of this, so developers should be careful to properly examine the use cases of input data and use an appropriate data type to store it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | integer-overflow | Integer overflow check |
| CodeSonar | 8.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Integer overflow check |
| TrustInSoft Analyzer | 1.38 | unsigned overflow | Unsigned overflow check |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Verifying that strings have not been mutated is essential to correct program output and behavior in systems that rely on constant string values. |

| **Noncompliant Code** |
| --- |
| In this example, the correctPassword variable can be changed by the system because it is not declared as a constant value. If it is changed by the program, the user will not be able to gain access to the system. |
| unsigned int userAccess = 0;  std::string correctPassword = “password”;  std::string givenPassword = “”;  std::cin >> givenPassword;  correctPassword = “changed”; /\* accepted password is changed \*/  if (givenPassword == correctPassword) { /\* user will not be given access with their correct password \*/  userAccess = 1;  } |

| **Compliant Code** |
| --- |
| Declaring the variable as a constant ensures that the value will not be changed unintentionally. |
| unsigned int userAccess = 0;  const std::string correctPassword = “password”; /\* accepted password is constant \*/  std::string givenPassword = “”;  std::cin >> givenPassword;  correctPassword = “changed”; /\* error – accepted password cannot be changed \*/  if (givenPassword == correctPassword) { /\* user will be given access with their correct password \*/  userAccess = 1;  } |

| **Principle:**  Use Effective Quality Assurance Techniques:  When designing software that secures sensitive data, certain techniques should be followed from the earliest stages of development to prevent successful attacks. Declaring constant variables is one technique that can help to ensure string correctness and prevent the loss or alteration of sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | string-literal-modification  write-to-string-literal  assignment-to-non-modifiable-lvalue  write-to-constant-memory | String literal modification check  Write to string literal check  Assignment to non-modifiable value  Write to constant memory variable |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR30-a  CERT\_C-STR30-b | String literal shall not be modified  Do not modify string literals |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL injection attacks can occur when SQL commands are given to the program where user input is expected This input can be used to bypass validation checks to return data that the program should not allow to be returned. It is critical to prevent these attacks by validating user input and identifying when a user has entered potentially malicious input. |

| **Noncompliant Code** |
| --- |
| This example accepts user input and queries a database with the provided input with no verification that the input is acceptable. |
| std::string userinput = “”;  std::cin >> userInput;  if (sqlite3\_exec(database, userInput, callback, &records, &error) != SQLITE\_OK) {  sqlite\_free(error);  } |

| **Compliant Code** |
| --- |
| Using prepared statements prevents malicious users from bypassing verification checks by only accepting input that matches a particular pattern designed by the program developer. |
| std::string preparedStatement = “SELECT \* FROM USERS WHERE NAME=”;  std::string name = “”;  std::string userInput = “”;  std::cin >> userInput;  if (userInput != preparedStatement + name) {  if (sqlite3\_exec(database, userInput, callback, &records, &error) != SQLITE\_OK) {  sqlite\_free(error);  }  } |

| **Principles:**  Validate Input Data:  SQL injection attacks are commonly rooted in data input fields, where malicious users enter SQL commands that trick the program into returning results that it should not be able to return. Validating input is a crucial method of preventing successful SQL injection attacks.  Default Deny:  By default, programs should deny access to any sensitive information until a user is properly authenticated and authorized. This can help to prevent SQL injection attacks even when there is no input validation in place. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | IO.INJ.FMT  MISC.FMT  MISC.FMTTYPE | Format string injection  Format string  Format string type error |
| Coverity | 2017.07 | PW  TAINTED\_STRING | Report when number of arguments differs from required number defined by format string  Identify tainted string format |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Memory can be corrupted by attacks such as buffer overflows, where attackers input data that is too large for the data type being used to hold it and effectively rewrite the data in nearby memory. |

| **Noncompliant Code** |
| --- |
| This example receives user input immediately after a private variable is initialized in memory. If the user attempts a buffer overflow attack, they can succeed in overwriting the value held by the variable. |
| const std::string privateVariable = “12345”;  char userInput[6];  std::cin >> userInput; |

| **Compliant Code** |
| --- |
| To prevent this type of attack from successfully overwriting memory, the width field member of the std::cin function can be set to the expected input width. As a result, any additional data inputted at this point is discarded and the private variable is not overwritten. |
| const std::string privateVariable = “12345”;  char userInput[6];  std::cin.width(6);  std::cin >> userInput; |

| **Principle:**  Keep It Simple:  When developing secure programs, it is essential to consider potential unwanted actions being taken with the program and create mechanisms to prevent malicious users from carrying out these actions. To protect program memory, simple steps can be taken to ensure that user input does not allow memory to be overwritten. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU | Buffer overrun  Buffer underrun |
| PolySpace Bug Finder | R2023b | CERT C++: STR53-CPP | Array access out of bounds |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are used to verify that code is working as expected by the developer, and raise errors if not. Using them throughout the code base during development can help to pinpoint where the code has gone wrong in the event of particularly complex and unexpected program output. |

| **Noncompliant Code** |
| --- |
| In this example, a simple flaw is made when checking if the x and y variables are equal. Because of this mistake, the program will always output the statement within the if branch, then unnecessarily divide the x variable, which is unexpected program behavior. There is no assertion after the branch, and as a result, this bug will be slightly more difficult to catch. |
| int x = 50;  int y = 100;  if (x = y) {  std::cout << “it’s true!” << std::endl;  x = x / 2;  } |

| **Compliant Code** |
| --- |
| To verify that the x value should equal half of the y value as expected, an assertion should be placed in the if branch after the operation takes place. In this case, the assertion would fail, as the x value was not equal to the y value before being divided. |
| int x = 50;  int y = 100;  if (x = y) {  std::cout << “it’s true!” << std::endl;  x = x / 2;  assert(x = y / 2);  } |

| **Principles:**  Keep It Simple:  Assertions are used in unit testing to ensure that components in a program are working as expected at their smallest level. For this reason, assertions should be simple by design and only test one specific piece of functionality. As a result of their simplicity, the developer can be sure that the component functions as intended.  Use Effective Quality Assurance Techniques:  The use of assertions is considered an effective quality assurance technique and helps to prevent the high cost of fixing a program when an issue is discovered late in the development cycle. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Detect when assertion may have a side effect |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions are used when implementing program logic that may not always compile correctly. It is critical to use exceptions to handle unexpected behavior and prevent program crashes. |

| **Noncompliant Code** |
| --- |
| This example verifies that a user has inputted an age of at least 21. If the user has entered a number less than 18, there is no program output to handle this event and the user is granted access due to an additional flaw in the code. |
| int access = 0;  int age = 18;  if (age > 20) {  std::cout << “party on! << std::endl;  }  access = 1; |

| **Compliant Code** |
| --- |
| To properly handle the event that a user enters a number less than 21, the program tries to grant access, but an error is thrown after the program detects an input of less than 21. The catch block handles this event by appropriately outputting a message to the user. |
| int access = 0;  try {  int age = 18;  if (age > 20) {  std::cout << “party on! << std::endl;  } else {  throw (age);  }  access = 1;  } catch (int errorAge) {  std::cout << “you must wait “ << (21 - errorAge) << “ years to party.” << std::endl;  } |

| **Principles:**  Heed Compiler Warnings:  Compiler warnings often alert developers to potential issues that may arise during the runtime of an application, which may either result from the developer being unaware of potential conflicts or the developer misusing the code. To prevent program crashes from either of these scenarios, exceptions should be used when the compiler indicates a potential problem that may arise in any particular area of the code base to ensure that the program handles the problem appropriately.  Architect and Design for Security Policies:  Exceptions are considered an essential method of preventing security issues that arise when a program may behave unexpectedly. This also helps to ensure that malicious users cannot abuse the system to cause unexpected program output, which is a crucial aspect of program security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable catch |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Handle all explicitly thrown exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| Access Control | STD-008-CPP | Access to sensitive areas of a system should only be allowed after a user has been successfully authenticated and authorized, and the system should default deny access to any part of the system. |

| **Noncompliant Code** |
| --- |
| In this example, access to the secure part of the system is granted immediately after the user enters their password. If this password was somehow bypassed by a user, they would still have access to the secure part of the system. |
| std::string password = “password”;  std::string userInput = “”;  std::cin >> userInput;  if (userInput == password) {  // access secure part of system  } |

| **Compliant Code** |
| --- |
| To prevent users from gaining access after bypassing the password, an access control variable is changed to 1 only if the user has correctly entered a password. The program then checks whether the variable is changed to allow access. |
| int access = 0;  std::string password = “password”;  std::string userInput = “”;  std::cin >> userInput;  if (userInput == password) {  access = 1; /\* grant access \*/  }  if (access) { /\* check if access granted \*/  // access secure part of system  } |

| **Principles:**  Default Deny:  To help prevent users from gaining access to a system before being authenticated, the default behavior of the system should be to deny any user access. This is a common security practice that represents a core layer of defense.  Adhere to the Principle of Least Privilege:  If a user has been authenticated, they should only be given access to parts of the system that are directly related to the tasks that they must accomplish within the system, and nothing else. This acts as a form of damage control if any malicious user is authenticated by the system, and helps to prevent sensitive data from becoming lost. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | CONCURRENCY.DATARACE | Data race |
| Polyspace Bug Finder | R2023b | CERT C: Rule SIG31-C | Check for shared data access within signal handler |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| Input Validation | STD-009-CPP | Validating user input is critical to ensuring that the program behaves as expected and does not crash when bad input is received. |

| **Noncompliant Code** |
| --- |
| This example converts a string from user input into an integer variable without checking if the input is numerical. If the user enters input that is not numerical, the program crashes before it can operate on the variable. |
| std::string userInput = “”;  int userInputDigit = 0;  std::cin >> userInput;  userInputDigit = userInput; /\* if received input is not numerical, program will crash \*/  if (userInputDigit >= 0) {  // handle when digit is positive number  } else {  // handle when digit is not positive number  } |

| **Compliant Code** |
| --- |
| To validate that the input is numerical, the isdigit() function is used. If the program verifies that the input is numerical, it continues to operate on the variable, otherwise an error is sent to the user. |
| std::string userInput = “”;  int userInputDigit = 0;  std::cin >> userInput;  if (isdigit(userInput)) { /\* validate that input received is numerical \*/  userInputDigit = userInput;  if (userInputDigit >= 0) {  // handle when digit is positive number  } else {  // handle when digit is not positive number  }  } else {  std::cout << “error, input must be numerical” << std::endl;  } |

| **Principle:**  Validate Input Data:  Validating input data is a crucial step toward ensuring the security of a system and preventing any user from causing the program to produce unexpected output. This is relevant to attacks such as SQL injection and buffer overflows, which often depend on input validation that has been implemented poorly or not at all. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT05-a | Avoid unsafe string functions that do not check bounds |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Description of Standard** |
| --- | --- | --- |
| Data Range | STD-010-CPP | To avoid having to check every possible scenario given data that must be compared with other data, its range can be verified to ensure that it is within the correct range of values that must be compared. |

| **Noncompliant Code** |
| --- |
| This example checks which day of March is held by the dayOfMonth variable. It must check if it is greater than 15, less than 15, or equal to 15. It does not appropriately handle values that are less than 1 or greater than 31, which are not possible in the month of March. If the variable is set to an impossible value, the program behaves unexpectedly. |
| int month = 3;  int dayOfMonth = 1;  if (dayOfMonth > 15) {  std::cout << “Caesar is dead” << std::endl;  } else if (dayOfMonth < 15) {  std::cout << “not quite yet” << std::endl;  } else if (dayOfMonth == 15) {  std::cout << “beware this day” << std::endl;  } |

| **Compliant Code** |
| --- |
| To prevent this unwanted behavior, the range of the variable can be checked before the if branches. By verifying that the value is within the range 1-31, values less than 1 and greater than 31 are ignored. Additionally, the if branch can be simplified. If the value passes the range check, is not greater than 15, and is not less than 15, then it must be equal to 15. |
| int month = 3;  int dayOfMonth = 1;  if ((dayOfMonth >= 1) && (dayOfMonth <= 31)) {  if (dayOfMonth > 15) {  std::cout << “Caesar is dead” << std::endl;  } else if (dayOfMonth < 15) {  std::cout << “not quite yet” << std::endl;  } else {  std::cout << “beware this day” << std::endl;  }  } |

| **Principles:**  Validate Input Data:  One way that users can misuse the system is by inputting data that does not match the expected range of integers, such as a date. To prevent this, input validation should be used to ensure that input falls into the expected range.  Keep It Simple:  Rather than checking every possible input value, a potentially impossible task, a simple if-else statement can be implemented to catch any input that does not fall into the expected range. The statement should simply execute if the input is within the expected range, and otherwise display some form of error message to the user, and ideally allow the user to attempt a new input value. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: CTR51-CPP | Check for use of invalid iterator |

### Defense-in-Depth Illustration

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



### Automation



The existing DevOps process can be modified to automate enforcement of the standards in this policy by implementing the use of automated tools throughout the development cycle. Tools that verify the appropriate use of assertions and correctness of data types can be used during program development, while static analysis can be conducted after the code is considered complete to determine what vulnerabilities may still exist in terms of exception handling, memory protection, and other potential threats. After code has been deployed, tools should be used to further monitor activity throughout the system, ensuring that any issues discovered at this point are promptly addressed and cause minimal damage.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | Medium | 2 |
| STD-002-CPP | High | Unlikely | High | Medium | 3 |
| STD-003-CPP | Medium | Unlikely | Low | Low | 2 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | High | Likely | High | High | 5 |
| STD-006-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-007-CPP | High | Likely | Low | Medium | 3 |
| STD-008-CPP | High | Unlikely | High | High | 5 |
| STD-009-CPP | Medium | Likely | Medium | High | 4 |
| STD-010-CPP | Low | Unlikely | Low | Low | 1 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the encryption of data being held on physical storage devices, which are not being actively used or accessed. Creating a properly organized file structure to encrypt for this data is essential to protecting it against users of the database or server where it is located. |
| Encryption in flight | Encryption in flight refers to the encryption of data being accessed by various mechanisms within the system. Malicious users may attempt to exploit poorly written code to access data that they are not authorized to access, and therefore it is important to ensure that the data is properly encrypted by any mechanism of access. |
| Encryption in use | Encryption in use refers to the encryption of sensitive data that has been accessed by a user of the system. While a user may be authorized to receive the data, any data that will not be explicitly and necessarily used should remain encrypted to prevent its potential loss or misuse. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the system verifying which specific user is attempting to use it. It represents the first level of a secure system and helps to prevent sensitive data from being misused by users. Multi-factor authentication is a method of obtaining multiple forms of ‘proof’ that a user is actually the user that they claim to be, and this works to ensure that all users of the system are verified properly. |
| Authorization | Authorization refers to the system’s ability to allow access to particular aspects of the system to particular users that have been previously authenticated. This is the next step in securing sensitive data, as it ensures that not every user will have access to it. The principle of least privilege is a method of limiting the loss or misuse of sensitive data by authorizing only to those who must explicitly and necessarily use it. |
| Accounting | Accounting refers to the monitoring of the system after it has been deployed to ensure that any breach or misuse of the system is promptly addressed, further limiting the potential damage that can be caused to the system in the case that sensitive data is lost. |

## 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 | 03/24/2024 | Initial Version | Charlie Vinson |  |
| 2.0 | 04/14/2024 | Final Version | Charlie Vinson |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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