**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. |
| --- | --- |
| * ValidateInput Data | The process of inspecting and purifying data from outside sources before it enters a system is known as validation of input data. By doing this, you can be confident the data is correct, well formatted, and devoid of any harmful content that might take advantage of holes in the system. Validation approaches that are effective in preventing unauthorized access, data corruption, and other security breaches include format checks, length limitations, and content verification. Organizations safeguard sensitive data and their applications from potential risks by validating input data, hence upholding system security and integrity.  Compiler warnings are generated throughout the code compilation process; it is important to pay close attention to and respond to these warnings. These alerts draw attention to possible problems, such as code that might not function as intended, possible security holes, or prospective trouble spots. Through heeding these cautions and fixing problems, programmers can guarantee that their code is more reliable, safe, and effective. Compiler warnings are important for maintaining high-quality code because ignoring them can lead to hidden mistakes that could cause software malfunctions or crashes. |
| * Heed Compiler Warnings | Compiler warnings are generated throughout the code compilation process; it is important to pay close attention to and respond to these warnings. These alerts draw attention to possible problems, such as code that might not function as intended, possible security holes, or prospective trouble spots. Through heeding these cautions and fixing problems, programmers can guarantee that their code is more reliable, safe, and effective. Compiler warnings are important for maintaining high-quality code because ignoring them can lead to hidden mistakes that could cause software malfunctions or crashes. |
| * Architect and Design for Security Policies | Integrating security measures into a system's basic architecture and design from the beginning is known as "architecting and designing for security policies." This methodology guarantees the incorporation of security considerations across the whole development process, from the preliminary stages to the final deployment. It entails laying down precise security requirements, putting safe coding techniques into reality, and utilizing threat-reduction design principles. Organizations can lower the risk of security breaches, safeguard their systems against vulnerabilities, and guarantee compliance with applicable rules by taking proactive security measures. This all-encompassing approach to security aids in building robust systems that are resistant to changing threats. |
| * Keep It Simple | "Keep it simple" is a fundamental design and problem-solving principle that highlights the significance of simplicity for efficiency and clarity. Systems and solutions become easier to use, understand, and manage when superfluous complexity is avoided. Error risk is decreased, user experience is improved, and future adjustments and debugging are made easier with simple designs. In order to produce more durable and dependable results, this idea promotes concentrating on the features and functionality that are truly necessary, optimizing workflows, and getting rid of unnecessary components. |
| * Default Deny | A security approach known as "default deny" limits system resource access by default, allowing only actions that are expressly authorized. By limiting access to vital data and functions to only authorized people and processes, this strategy reduces the attack surface. Any unknown or ambiguous actions are automatically prevented by the default of denying, which lowers the possibility of unwanted access and possible security lapses. By imposing stringent access controls and demanding specific authorization for all actions, a "default deny" policy can be implemented to assist create a secure environment and improve system security as a whole. |
| * Adhere to the Principle of Least Privilege | Respecting the least privilege concept entails giving users and systems the minimal amount of access required to carry out their responsibilities. This security approach limits access to sensitive information and vital functions, hence lowering the possibility of unintentional or deliberate resource misuse. Potential harm from security lapses or human error is reduced by making sure that people and processes only have the necessary authorizations to function. This idea improves overall system integrity and keeps an environment safe by limiting unneeded access. |
| * Sanitize Data Sent to Other Systems | Data must be cleaned and validated before being delivered to other systems to make sure it contains no malicious code or unexpected input. By doing this, security flaws like injection attacks are avoided, which can happen when untrusted data is handled by another system. Organizations can safeguard networked systems against potential exploitation and guarantee that only safe and intended information is delivered by fully cleaning data. This procedure is necessary to protect sensitive data, preserve data integrity, and guarantee the dependable operation of all related systems. |
| * Practice Defense in Depth | Defense in depth is a security approach that uses several defense layers to shield data and systems against intrusions. This method integrates different controls and safeguards at several levels, such as physical security, network security, and application security, because it acknowledges that no one security solution is infallible. It is far more difficult for attackers to get past many layers of security when there are multiple barriers in place. Defense in depth improves the security posture overall by offering redundancy and raising the likelihood of identifying and thwarting threats before they do serious damage. |
| * Use Effective Quality Assurance Techniques | Prior to deployment, software must satisfy the necessary requirements for quality, functionality, and dependability. This is ensured by using efficient quality assurance (QA) approaches. This entails a methodical approach to validation, code reviews, and testing in order to find and address errors early in the development cycle. High code quality is maintained and the likelihood of errors and vulnerabilities is decreased with the use of techniques like automated testing, continuous integration, and peer reviews. Organizations can improve user happiness, lower maintenance costs, and offer software that is more safe and stable by implementing strong quality assurance methods. Achieving consistent and dependable software performance requires effective QA. |
| * Adopt a Secure Coding Standard | Using a set of best practices and standards for writing code that reduces security risks is part of adopting a secure coding standard. These guidelines give developers detailed guidance on how to prevent typical coding mistakes including buffer overflows, injection issues, and inappropriate error handling that could result in security breaches. Organizations can make sure that their software is more robust overall and more resistant to threats by following a secure coding standard. This procedure improves applications' security posture while also encouraging uniformity and excellence across the development team, which eventually results in software that is safer and more dependable. |

### 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 Standard** | INT01-CPP | .Use rsize\_t or size\_t for all integer values representing the size of an object |

| **Noncompliant Code** |
| --- |
| INT01-CPP  A long number that is longer than an int's range in the non-compliant code sample is implicitly transformed to an int. Because the value is too big to be represented by an int, this conversion may cause data loss or misinterpretation. Such problems may give rise to unforeseen conduct and possible weaknesses. |
| //  #include <iostream>  void processValue(int value) {  // Process the integer value  std::cout << "Value: " << value << std::endl;  }  int main() {  long longValue = 2147483648; // Value exceeds the range of int  processValue(longValue); // Implicit conversion  return 0;  } |

| **Compliant Code** |
| --- |
| INT01-CPP  A helper's safe operationTo determine if a long number falls inside an int's acceptable range, use convert. An exception is raised if the value is outside of the range. By ensuring that only legitimate conversions take place, misinterpretation and data loss are avoided. By adhering to this guideline, the code handles integer conversions directly and properly, exhibiting increased reliability and safety. |
| #include <iostream>  #include <limits>  #include <stdexcept>  void processValue(int value) {  // Process the integer value  std::cout << "Value: " << value << std::endl;  }  int safeConvert(long value) {  if (value > std::numeric\_limits<int>::max() || value < std::numeric\_limits<int>::min()) {  throw std::out\_of\_range("Value out of range for int");  }  return static\_cast<int>(value);  }  int main() {  try {  long longValue = 2147483648; // Value exceeds the range of int  int safeValue = safeConvert(longValue);  processValue(safeValue);  } catch (const std::out\_of\_range& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

| **Principles(s):** The robustness principle highlights the requirement for software to be able to tolerate unforeseen events and inputs. The standard ensures resilient and dependable program behavior by checking the range of integer conversions, hence preventing mistakes caused by misunderstanding or data loss. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Type Conversion Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Integer Conversion | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Type Conversion Warnings Data Conversion Vulnerabilities | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| FlawFinder | 2.0 | Data Conversion Vulnerabilities | In order to reduce security risks and enhance code resilience, FlawFinder examines C/C++ code for possible security vulnerabilities, including problems resulting from incorrect data type conversions. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value Standard** | INT32-C | Ensure that operations on signed integers do not result in overflow |

| **Noncompliant Code** |
| --- |
| INT32-C  Without checking for overflow, the multiply function multiplies two int values. Incorrect results may arise from multiplication that overflows, provided that x and y have sufficiently large values. |
| #include <iostream>  int multiply(int a, int b) {  return a \* b; // Potential for overflow  }  int main() {  int x = 100000;  int y = 100000;  int result = multiply(x, y);  std::cout << "Result: " << result << std::endl; // Incorrect result if overflow occurs  return 0;  } |

| **Compliant Code** |
| --- |
| INT32-C  In order to make sure that multiplying the two integers won't cause an overflow, the multiply function has checks. When an overflow condition is found, these checks compare the operands to the int type's bounds and throw an overflow\_error. This rule guarantees that integer overflow is avoided in the code, making software safer and more dependable. |
| #include <iostream>  #include <limits>  #include <stdexcept>  int multiply(int a, int b) {  if (a > 0 && b > 0 && a > (std::numeric\_limits<int>::max() / b)) {  throw std::overflow\_error("Integer overflow on multiplication");  }  if (a < 0 && b < 0 && a < (std::numeric\_limits<int>::max() / b)) {  throw std::overflow\_error("Integer overflow on multiplication");  }  if (a > 0 && b < 0 && b < (std::numeric\_limits<int>::min() / a)) {  throw std::overflow\_error("Integer overflow on multiplication");  }  if (a < 0 && b > 0 && a < (std::numeric\_limits<int>::min() / b)) {  throw std::overflow\_error("Integer overflow on multiplication");  }  return a \* b;  }  int main() {  try {  int x = 100000;  int y = 100000;  int result = multiply(x, y);  std::cout << "Result: " << result << std::endl;  } catch (const std::overflow\_error& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** The idea of reliability highlights the necessity for software to generate accurate and consistent outcomes. The standard guards against overflow and truncation issues, which might result in inaccurate computations or system behavior, by guaranteeing that integer conversions are done correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Type Conversion Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Integer Conversion | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Type Conversion Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| FlawFinder | 2.0 | Data Conversion Vulnerabilities | In order to reduce security risks and enhance code resilience, FlawFinder examines C/C++ code for possible security vulnerabilities, including problems resulting from incorrect data type conversions. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **(Data) String Correctness Standard** | STR50-CPP | Ensure that there is enough room in string storage for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| STR50-CPP  Without determining if the buffer has the capacity to hold the source string and the null terminator, the unsafeCopy function copies a source string into a fixed-size buffer. If the source string is longer than the buffer, this could result in a buffer overflow. |
| #include <iostream>  #include <cstring>  void unsafeCopy(const char\* src) {  char buffer[10];  std::strcpy(buffer, src); // Potential overflow if src is larger than buffer  std::cout << "Copied string: " << buffer << std::endl;  }  int main() {  const char\* longString = "This is a very long string";  unsafeCopy(longString); // Unsafe, can cause buffer overflow  return 0;  } |

| **Compliant Code** |
| --- |
| STR50-CPP  The first thing the safeCopy function does is see if the source string's length is larger than or equal to the buffer size. An overflow\_error is raised if the source string is too long. It also explicitly sets the buffer's last character to the null terminator and copies the string using std::strncpy, which guarantees that the buffer does not overflow. This method prevents buffer overflow and ensures that the string is appropriately null-terminated, boosting the safety and accuracy of string operations. |
| #include <iostream>  #include <cstring>  #include <stdexcept>  void safeCopy(const char\* src) {  const size\_t bufferSize = 10;  char buffer[bufferSize];  if (std::strlen(src) >= bufferSize) {  throw std::overflow\_error("Source string is too large for the buffer");  }  std::strncpy(buffer, src, bufferSize - 1);  buffer[bufferSize - 1] = '\0'; // Ensuring null termination  std::cout << "Copied string: " << buffer << std::endl;  }  int main() {  try {  const char\* longString = "This is a very long string";  safeCopy(longString); // Safe, checks buffer size  } catch (const std::overflow\_error& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Security concept, the goal of this concept is to shield software against flaws that an attacker may exploit. The standard guards against buffer overflows and other similar vulnerabilities that might result in security breaches by guaranteeing that strings are handled correctly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | String Handling Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | String Handling | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | String Manipulation Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| FlawFinder | 2.0 | String Vulnerabilities | In order to reduce security risks and enhance code resilience, FlawFinder examines C/C++ code for possible security vulnerabilities, including problems resulting from incorrect data type conversions. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection Standard** | IDS53-CPP | "Prevent SQL injection," |

| **Noncompliant Code** |
| --- |
| IDS53-CPP  User input is immediately concatenated into a SQL query string using the unsafeQuery function. This method leaves this system open to SQL injection threats since malevolent users can run arbitrary SQL statements by manipulating the input. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  void unsafeQuery(MYSQL\* conn, const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  mysql\_query(conn, query.c\_str()); // Vulnerable to SQL injection  }  int main() {  MYSQL\* conn;  // Assume conn is initialized and connected to the database  std::string userInput = "admin' OR '1'='1";  unsafeQuery(conn, userInput); // Potential SQL injection  return 0;  } |

| **Compliant Code** |
| --- |
| IDS53-CPP  To stop SQL injection, the safeQuery method employs prepared statements in parameterized queries. Without directly concatenating the user input into the query string, the SQL query is securely connected to the input using the mysql\_stmt\_bind\_param and mysql\_stmt\_prepare functions. By ensuring that the input is handled as data rather than executable code, this method improves application security and guards against SQL injection attacks. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  void safeQuery(MYSQL\* conn, const std::string& userInput) {  MYSQL\_STMT\* stmt;  stmt = mysql\_stmt\_init(conn);  const char\* query = "SELECT \* FROM users WHERE username = ?";  mysql\_stmt\_prepare(stmt, query, strlen(query));  MYSQL\_BIND bind[1];  memset(bind, 0, sizeof(bind));  bind[0].buffer\_type = MYSQL\_TYPE\_STRING;  bind[0].buffer = const\_cast<char\*>(userInput.c\_str());  bind[0].buffer\_length = userInput.length();  mysql\_stmt\_bind\_param(stmt, bind);  mysql\_stmt\_execute(stmt);  // Handle result set (not shown)  mysql\_stmt\_close(stmt);  }  int main() {  MYSQL\* conn;  // Assume conn is initialized and connected to the database  std::string userInput = "admin' OR '1'='1";  safeQuery(conn, userInput); // Safe from SQL injection  return 0;  } |

| **Principles(s):** Security concept, the goal of this concept is to shield software against flaws that an attacker may exploit. The standard guards against buffer overflows and other similar vulnerabilities that might result in security breaches by guaranteeing that strings are handled correctly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Information Leak Detection | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Data Handling | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Sensitive Data Exposure Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| FlawFinder | 2.0 | Data Exposure Vulnerabilities | In order to reduce security risks and enhance code resilience, FlawFinder examines C/C++ code for possible security vulnerabilities, including problems resulting from incorrect data type conversions. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection Standard** | MEM50-CPP | "Do not access freed memory." |

| **Noncompliant Code** |
| --- |
| MEM50-CPP  Memory is allocated, released, and then attempted to be accessed via the useFreedMemory function. This leads to undefinable behavior, which could result in data corruption, software failures, or security flaws. |
| #include <iostream>  #include <cstdlib>  void useFreedMemory() {  int\* ptr = (int\*)std::malloc(sizeof(int) \* 10);  if (ptr == nullptr) {  std::cerr << "Memory allocation failed" << std::endl;  return;  }  std::free(ptr);  // Accessing freed memory  ptr[0] = 42;  std::cout << "Value: " << ptr[0] << std::endl;  }  int main() {  useFreedMemory();  return 0;  } |

| **Compliant Code** |
| --- |
| MEM50-CPP  The memory is allocated, used, and then released via the useMemorySafely function. In order to minimize the possibility of accessing faulty memory, the pointer is set to nullptr after the memory has been destroyed. In addition to preventing undefinable behavior and improving the application's dependability and security, this guarantees secure memory management techniques. |
| #include <iostream>  #include <cstdlib>  void useMemorySafely() {  int\* ptr = (int\*)std::malloc(sizeof(int) \* 10);  if (ptr == nullptr) {  std::cerr << "Memory allocation failed" << std::endl;  return;  }  // Use the allocated memory  ptr[0] = 42;  std::cout << "Value: " << ptr[0] << std::endl;  // Free the allocated memory  std::free(ptr);  // Set pointer to nullptr to avoid dangling pointer  ptr = nullptr;  }  int main() {  useMemorySafely();  return 0;  } |

| **Principles(s):** The idea of reliability highlights the necessity for software to generate accurate and consistent outcomes. The standard guards against overflow and truncation issues, which might result in inaccurate computations or system behavior, by guaranteeing that integer conversions are done correctly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Memory Management Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Memory Management | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Memory Leak Detection | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| Valgrind | 3.18 | Memcheck | By identifying memory-related issues including leaks, unauthorized memory access, and the use of uninitialized memory, Valgrind's Memcheck tool makes sure that C++ applications handle memory effectively and safely. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions Standard** | MSC50-CPP | Do not use std::assert() in production code |

| **Noncompliant Code** |
| --- |
| MSC50-CPP  The process function verifies that the value is non-negative by using std::assert(). But release builds usually disable assertions, so this check won't be carried out in production code. Should an invalid input be entered, this can result in unexpected behavior. |
| #include <cassert>  #include <iostream>  void process(int value) {  assert(value >= 0); // This assertion will be removed in release builds  std::cout << "Processing value: " << value << std::endl;  }  int main() {  process(-1); // This should trigger the assertion  return 0;  } |

| **Compliant Code** |
| --- |
| MSC50-CPP  The process function verifies that the value is not negative explicitly during runtime. The function throws a std::invalid\_argument exception if the value is invalid. This method permits appropriate error handling and guarantees that the check is always carried out, even in production builds. The code is made more robust and reliable by enforcing input validation uniformly when runtime checks are used in place of assertions. |
| #include <stdexcept>  #include <iostream>  void process(int value) {  if (value < 0) {  throw std::invalid\_argument("Value must be non-negative");  }  std::cout << "Processing value: " << value << std::endl;  }  int main() {  try {  process(-1); // This will throw an exception  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

| **Principles(s):** The notion of portability and maintainability highlights the need of creating code that can be effortlessly modified and updated to suit various platforms and settings. Through the standardization process, undefined or implementation-defined behavior is avoided, and code consistency is guaranteed across platforms and compilers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Portability Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Implementation-Defined Behavior | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Portability Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| GCC (GNU Compiler Collection) | 11.2 | Wpedantic | Code portability and adherence to the C++ standard are encouraged by GCC's ability to identify the use of implementation-defined behavior and non-standard extensions when it has the `-Wpedantic} option enabled. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions Standard** | ERR50-CPP | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| ERR50-CPP  The process function first determines whether the input value is negative. If it is, it runs std::abort() after writing an error message to the standard error output. This causes the program to terminate abruptly, which is usually not what you want in production code because it prevents cleanup procedures and error handling that is easy for users to understand. |
| #include <iostream>  #include <stdexcept>  void process(int value) {  if (value < 0) {  std::cerr << "Negative value error" << std::endl;  std::abort(); // Abruptly terminates the program  }  std::cout << "Processing value: " << value << std::endl;  }  int main() {  process(-1); // This will cause the program to terminate abruptly  return 0;  } |

| **Compliant Code** |
| --- |
| ERR50-CPP  If the input value is negative, the process function throws a std::invalid\_argument exception rather than terminating the program. This exception is caught by the main function, which gracefully handles it by showing an error message. This method preserves program stability and improves user experience by enabling a clean program termination and offering a controlled error management opportunity. |
| #include <iostream>  #include <stdexcept>  void process(int value) {  if (value < 0) {  throw std::invalid\_argument("Value must be non-negative");  }  std::cout << "Processing value: " << value << std::endl;  }  int main() {  try {  process(-1); // This will throw an exception  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl; // Handle the error gracefully  }  return 0;  } |

| **Principles(s):** The principles of robustness and reliability idea: According to this idea, software must consistently and elegantly handle failures. The standard contributes to the prevention of crashes, undefinable behavior, and other unexpected events by guaranteeing that error circumstances are appropriately identified and controlled, resulting in more robust and dependable software. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Error Handling Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Error Handling | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Handling Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| Coverity | 2024.3 | Error Handling Defects | Coverity is a static analysis tool that helps stop mistakes from leading to system failures by identifying flaws in error handling, such as neglecting to verify error codes, failing to handle exceptions, and handling unusual situations incorrectly. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Terminate Standard** | ERR51-CPP | Do not call std::terminate() directly |

| **Noncompliant Code** |
| --- |
| ERR51-CPP  Without doing the necessary resource cleanup, invoking std::terminate() immediately terminates the application, potentially resulting in resource leaks or undefined behavior. |
| #include <iostream>  #include <exception>  void myFunction() {  // Some code...  std::terminate(); // Noncompliant: Directly calling std::terminate()  }  int main() {  try {  myFunction();  } catch (...) {  std::cerr << "An exception was caught.\n";  }  return 0;  } |

| **Compliant Code** |
| --- |
| ERR51-CPP  By throwing an exception rather than using std::terminate(), the program may handle faults gracefully by properly unwinding the stack and cleaning away its resources. |
| #include <iostream>  #include <exception>  void myFunction() {  // Some code...  throw std::runtime\_error("An error occurred."); // Compliant: Use exceptions instead of std::terminate()  }  int main() {  try {  myFunction();  } catch (const std::exception &e) {  std::cerr << "Caught exception: " << e.what() << '\n';  }  return 0;  } |

| **Principles(s):**The concept of Reliability and Robustness: This concept highlights the significance of software error handling that is consistent and predictable. The standard helps avoid resource leaks, deadlocks, and other problems that might jeopardize the software's dependability and robustness by making sure that resources are released appropriately in the case of exceptions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Resource Management Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Resource Management | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Exception Safety Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| Coverity | 2024.3 | Resource Leak Detection | Coverity is a static analysis tool that helps stop mistakes from leading to system failures by identifying flaws in error handling, such as neglecting to verify error codes, failing to handle exceptions, and handling unusual situations incorrectly. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Honor Execptions Standard** | ERR55-CPP | Honor exception specifications |

| **Noncompliant Code** |
| --- |
| ERR55-CPP  The noexcept flag on the function mayThrow indicates that it shouldn't throw exceptions. It does, however, throw an exception, which results in ambiguous behavior. |
| #include <iostream>  void mayThrow() noexcept {  throw std::runtime\_error("This function should not throw");  }  int main() {  try {  mayThrow();  } catch (const std::exception& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| ERR55-CPP  Removing the noexcept statement appropriately reflects the possibility of an exception being thrown by the function. |
| #include <iostream>  void mayThrow() {  throw std::runtime\_error("This function may throw");  }  int main() {  try {  mayThrow();  } catch (const std::exception& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  }  return 0;  } |

| **Principles(s):** The Security and Reliability Principle focuses on how software must handle errors in a safe and consistent manner. This standard contributes to the maintenance of software security and dependability by guaranteeing that functions do not unintentionally reveal internal states or sensitive information when exceptions arise. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Exception Handling Issues | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Exception Safety | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Exception Handling Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| Coverity | 2024.3 | Exception Management Defects | Coverity is a static analysis tool that helps stop mistakes from leading to system failures by identifying flaws in error handling, such as neglecting to verify error codes, failing to handle exceptions, and handling unusual situations incorrectly. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Catch Exceptions Standard** | ERR60-CPP | Catch exceptions by reference |

| **Noncompliant Code** |
| --- |
| ERR60-CPP  Value-based exception handling may result in data loss and slicing of exception-specific data. |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  throw std::runtime\_error("An error occurred");  }  int main() {  try {  riskyOperation();  } catch (std::exception e) { // Catching by value  std::cerr << "Caught exception: " << e.what() << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| ERR60-CPP  The exception object and its polymorphic behavior are preserved in the complying code sample, which catches the exception via reference (const std::exception&). |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  throw std::runtime\_error("An error occurred");  }  int main() {  try {  riskyOperation();  } catch (const std::exception& e) { // Catching by reference  std::cerr << "Caught exception: " << e.what() << std::endl;  }  return 0;  } |

| **Principles(s):** The need of safe and dependable error handling procedures is emphasized by the Security and Reliability Principle. This standard contributes to preventing information leaks and maintaining system integrity by guaranteeing that functions do not return undefined values or sensitive information following an error. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.9 | Error Handling and Return Values | A static analysis tool for C/C++ code is called CppCheck. The particular checks for type conversion issues finds possible errors with implicit conversions that could cause data loss, like this one where an int is implicitly converted from a long value. |
| Clang Static Analyzer | 15.0 | Return Value Checks | By guaranteeing correct type handling, Clang Static Analyzer may detect unsafe implicit type conversions that can cause data loss or integer overflows, improving code safety. |
| PC-lint Plus | 1.4 | Error Return Value Warnings | A static analysis tool called PC-lint Plus looks for any problems with type conversions, assisting in the prevention of mistakes that could result from implicit type casting. |
| Coverity | 2024.3 | Incorrect Return Value Handling | Coverity is a static analysis tool that helps stop mistakes from leading to system failures by identifying flaws in error handling, such as neglecting to verify error codes, failing to handle exceptions, and handling unusual situations incorrectly. |

### 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.

[Insert your written explanations here.]

### 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 |
| --- | --- | --- | --- | --- | --- |
| INT01-CPP | High | Unlikely | Medium | High | 2 |
| INT32-C | High | Unlikely | Medium | High | 2 |
| STR50-CPP | High | Likely | Medium | High | 1 |
| IDS53-CPP | High | Likely | Medium | High | 1 |
| MEM50-CPP | High | Likely | Medium | High | 1 |
| MSC50-CPP | High | Possible | Low | Medium | 2 |
| ERR50-CPP | High | Likely | Medium | High | 1 |
| ERR51-CPP | High | Likely | Medium | High | 1 |
| ERR55-CPP | High | Possible | Medium | High | 1 |
| ERR60-CPP | High | Possible | Medium | High | 1 |

### 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 | Data that is kept on a disk, be it a local hard drive, cloud storage, or any other type of storage device, is protected when it is encrypted while at rest. With the help of encryption, data is kept safe and unreadable even in the event that the storage media is accessed without authority.  Files, databases, and backups containing sensitive information must all be secured with industry-accepted techniques like AES-256. Any data kept on local computers, servers, or cloud environments is covered by this policy. This guarantees that the data is safe and unreadable even in the case of physical theft, illegal access, or storage breach.  When physical storage devices are hacked, this strategy is essential for preventing unauthorized access to critical data. Regardless of how long-term or temporary the storage of data is, the policy has to be followed. It is especially important in situations when physical security isn't always assured, like with portable devices or cloud storage. |
| Encryption in flight | The term "encryption in flight" describes the safeguarding of data during network transmission. By using encryption, you can be sure that information sent between clients, servers, or other elements of the system is safe and won't be accessed or altered by unauthorized parties.  Every piece of data that is sent over a network, be it to or from external partners, cloud services, or internal systems, has to be secured using safe protocols like TLS (Transport Layer Security). Any data transmission over the internet, internal networks, or any other communication channel is covered by this policy.  This policy is necessary to stop data breaches that could be caused by man-in-the-middle attacks or interception of data while it is being sent. It must be used anytime data leaves a monitored, secure environment in order to guarantee that it is safe traveling across potentially unsafe networks. |
| Encryption in use | The term "encryption in use" describes the safeguarding of data in memory during processing or active usage. This guarantees that data is protected from unwanted access or disclosure even after it is loaded into memory for processing or modification.  Secure enclaves and hardware-based encryption are two encryption methods that must be used to safeguard sensitive data stored in memory. This policy is applicable to systems that manage extremely sensitive data, where there is a considerable risk associated with the data in use, such as financial transactions, personal identifying information (PII), or health records.  When processing extremely sensitive data in potentially unsafe settings, such shared cloud infrastructure or multi-tenant settings, the policy is essential. In order to guard against exposure via memory dumps, side-channel attacks, or unauthorized access to running processes, it should be used while processing sensitive data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying a user's or system's identity is called authentication. This guarantees that the organization's resources are only accessible to authorized users or systems.  Before being allowed access to any resources, all users and systems must undergo a rigorous authentication process employing multiple factor authentication. User logins, API access, and system interactions are all covered by this policy. It is recommended to use authentication methods like hardware tokens, OTP (One-Time Password), and passwords paired with biometrics.  In order to avoid unauthorized access and potential security breaches, this policy is crucial for making sure that only authorized entities may access the organization's resources. Every system and application where user access is necessary is covered by this policy, which makes sure that every access request is verified before any resources are made available.  This policy is crucial for preventing unauthorized access and potential security breaches by guaranteeing that only authorized entities may access the organization's resources. The policy makes sure that every access request is validated before any resources are made available, and it applies to all systems and applications where user access is necessary. |
| Authorization | Determining whether a user or system is authorized to utilize a given resource or carry out a specific operation is the process of authorization. By doing this, it is made sure that users only have access to the resources required for their job.  Role-based access control, or RBAC, is required to regulate resource access and guarantee that users have the minimal amount of rights required to carry out their responsibilities. File systems, network resources, database access, and application functionality are all covered by this policy. Access levels for users should be routinely evaluated and changed as needed.  This policy is essential for limiting illegal access to private data and system capabilities. It should be used to make sure that just the permissions required for their job are provided to any person or system that asks access to a resource. |
| Accounting | The logging and tracking of user activity within the system is referred to as accounting. This ensures accountability and provides the opportunity to audit user activity by keeping track of modifications made, resource access, and other noteworthy activities.  Every action taken by users or the system, such as logins, database modifications, user additions, and access to private files, needs to be recorded. This policy is applicable to all systems, and logs are kept safe and periodically examined for irregularities or indications of illegal access.  This policy is essential for guaranteeing accountability and helping to identify security incidents by allowing all activities inside the system to be tracked back to the individual or system that carried them out. It ought to be used consistently, with records kept and continually checked for any questionable activity. |

**\***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

1. Security Principle: guarantees that resources and data are shielded against unwanted access and modification.
2. Reliability Principle guarantees that systems generate accurate results on a regular basis.
3. Robustness Principle guarantees that software is capable of handling unforeseen situations and inputs.
4. Maintainability Principle makes ensuring that updating and maintaining systems is simple.
5. Portability Principle guarantees that systems are able to function in many situations.
6. Accountability Principle guarantees that activities may be linked to specific people or systems
7. Confidentiality Principle makes certain that private data is shielded from unwanted publication.
8. Integrity Principle guarantees the reliability and accuracy of the data.
9. Availability Principle makes certain that data and systems are available when needed.
10. makes ensuring that systemic acts can be monitored and reviewed.
11. INT01-CPP
    1. Reliability (2): Guarantees that integer values are represented correctly, avoiding overflow problems that can provide inaccurate outcomes.
    2. Robustness (3): Prevents crashes and undefinable behavior by employing proper data types to handle unexpected input sizes.
    3. Integrity (8): Prevents corruption by guaranteeing that integer sizes are handled correctly, maintaining the accuracy of the data.
12. INT32-C
    1. Reliability (2): Guarantees that operations yield accurate and consistent results by preventing overflow in signed integers.
    2. Robustness (3): Prevents system failures by providing protection against unexpected integer values that might lead to overflow.
    3. Integrity (8): Prevents overflows that might provide inaccurate data, safeguarding the calculations' correctness.
13. STR50-CPP
    1. Security(1): Prevents buffer overflows, which an attacker may use to run arbitrary code.
    2. Robustness (3): Prevents undefined behavior and crashes by making sure strings are ended appropriately.
    3. Integrity (8): Guarantees that strings are kept accurately and without truncation, protecting data correctness.
14. IDS53-CPP
    1. Security (1): Guards against data tampering and illegal access via SQL injection attacks.
    2. Confidentiality (7): Prevents unauthorized searches from obtaining information, hence guaranteeing the security of sensitive data.
    3. Integrity (8): Prevents unwanted changes via SQL injection, preserving the correctness and reliability of data.
15. MEM50-CPP
    1. Reliability (2): Avoids using memory that has already been released, which prevents crashes and undefinable behavior.
    2. Robustness (3): Ensures steady system functioning by carefully managing memory management to avoid access to invalid memory.
    3. Integrity (8): Prevents access to freed memory, which may contain inaccurate or outdated data, hence safeguarding data integrity.
16. MSC50-CPP
    1. Second reliability: makes sure that failed assertions don't cause production code to end abruptly.
    2. Maintainability (4): Reduces code maintenance effort by eliminating reliance on production-only debug-only features.
    3. Availability (9): By preventing sudden termination owing to assertions, this feature minimizes needless application downtime.
17. ERR50-CPP
    1. Reliability(2) makes sure that failures are handled by the program in a way that doesn't result in abrupt termination.
    2. Robustness (3): Prevents crashes by handling unforeseen circumstances, enhancing the application's overall stability.
    3. Accountability (6): Enables accurate error tracking and logging prior to program termination, guaranteeing auditability of problems.
18. ERR51-CPP
    1. Dependability (2): Guarantees that every possible deviation is managed, averting unanticipated software failures.
    2. Robustness (3): Enhances the program's capacity to handle unforeseen circumstances without malfunctioning.
    3. Accountability (6): Facilitates auditing and debugging by ensuring that exceptions are recorded and logged.
19. ERR55-CPP
    1. Reliability (2): Prevents unexpected behavior by making sure exceptions are handled according to specification.
    2. Maintainability (4): Makes maintenance easier by outlining the proper way to handle exceptions.
    3. Accountability (6): Makes sure that exceptions are handled in accordance with established guidelines, which creates a transparent audit trail.
20. ERR60-CPP
    1. Reliability (2): Guards against the possible hazards of copying exceptions by making sure they are detected in a safe and predictable way.
    2. Robustness (3): Increases program robustness by making sure exceptions are handled appropriately, averting crashes, and limiting undefined behavior.
    3. Maintainability (4): Applying recommended practices for managing exceptions makes the code easier to maintain.

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 |  |
| 2.0 | 07/11/24 | Final | Francis Cottrell-Eshaghi |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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