

# 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 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.

### Ten Core Security Principles

| **Principles** | Description |
| --- | --- |
| 1. ValidateInput Data | Ensuring the correctness and safety of input data is crucial. This involves verifying that data conforms to expected formats and ranges, thus protecting the system from potential threats like injection attacks and buffer overflows. |
| 1. Heed Compiler Warnings | Compiler warnings are not just suggestions; they often point to potential security vulnerabilities or logic errors. Addressing these warnings proactively can prevent security breaches and ensure code robustness. |
| 1. Architect and Design for Security Policies | Security should be integral to system architecture and design. This means implementing security policies at every stage, from conceptualization to deployment, to safeguard against vulnerabilities. |
| 1. Keep It Simple | Complexity often breeds security issues. A simpler design is easier to analyze for security flaws and reduces the likelihood of errors that could be exploited. |
| 1. Default Deny | In access control, the principle of 'default deny' means that access permissions are not granted unless explicitly defined. This reduces the attack surface by limiting access only to necessary entities. |
| 1. Adhere to the Principle of Least Privilege | Each component or user should have only the minimum privileges necessary to perform its functions. This limits the potential impact of a security breach. |
| 1. Sanitize Data Sent to Other Systems | Data sent to external systems must be sanitized to prevent the propagation of security issues. This includes escaping or removing potentially harmful data elements. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls. If one layer fails, others still provide protection, creating a more resilient defense against attacks. |
| 1. Use Effective Quality Assurance Techniques | Robust QA processes are essential to identify and rectify security vulnerabilities. This includes code reviews, testing, and employing static and dynamic analysis tools. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard, like the SEI CERT C++ guidelines, helps ensure consistent application of best practices in security across all development efforts. |

### 

### C/C++ Ten Coding Standards:

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Enforce Correct Integer Size Usage |

| **Rationalization** |
| --- |
| This standard ensures that integers are used with appropriate sizes, preventing issues such as integer overflows, which can lead to security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The function uses a signed integer int for a loop counter, which could lead to negative indexing or integer overflow issues. |
| void printNumbers(int count) {  for (int i = 0; i < count; ++i) {  std::cout << i << std::endl;  }  } |

| **Compliant Code** |
| --- |
| The function uses size\_t, an unsigned integer type, for the loop counter, ensuring non-negative values and appropriate range for counting or indexing. |
| #include <cstddef>  #include <iostream>  void printNumbers(size\_t count) {  for (size\_t i = 0; i < count; ++i) {  std::cout << i << std::endl;  }  } |

**Relevant Security Principle(s)**

| 6. Adhere to the Principle of Least Privilege |
| --- |
| **Rationalization:** Applying the principle of least privilege to integer size enforces using the smallest, adequate data type that fits the needed range of values, minimizing the risk of overflow and underflow errors. This principle ensures that data types are not granted more capacity than necessary for their intended use, reducing the potential for unexpected behaviors or vulnerabilities. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** |
| --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.MEM.BO |
| **Checker Description** | | |
| This checker identifies buffer overruns which can occur when incorrect integer sizes lead to memory allocation errors. It's chosen for its ability to detect situations where integer operations might exceed the allocated space, directly impacting the robustness of integer size usage. | | |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Robust String-to-Number Conversion |

| **Rationalization** |
| --- |
| This standard ensures the safe and accurate conversion of string data to numeric values, preventing errors such as out-of-range values, incorrect parsing, or misinterpretation of the string content, thereby enhancing the reliability and security of the software. |

| **Noncompliant Code** |
| --- |
| The atoi() function is used for converting a string to an integer. It lacks proper error handling for invalid or out-of-range inputs. |
| #include <cstdlib>  void convertString(const char \*buff) {  int number = atoi(buff); // Unsafe conversion  // Further processing...  } |

| **Compliant Code** |
| --- |
| The strtol() function is used for conversion, with checks for invalid input and number range. This method ensures that the conversion is within the bounds of an int and the string is a valid numeric representation. |
| #include <cstdlib>  #include <climits>  #include <iostream>  void convertString(const char \*buff) {  char \*end;  long number = strtol(buff, &end, 10);  if (\*end != '\0' || number > INT\_MAX || number < INT\_MIN) {  std::cerr << "Invalid or out-of-range number." << std::endl;  return;  }  int safeNumber = static\_cast<int>(number);  // Further processing...  } |

**Relevant Security Principle(s)**

| 1. **Validate Input Data** |
| --- |
| **Rationalization:** Robust string-to-number conversions are crucial for ensuring that input data is validated before use. This principle prevents errors and vulnerabilities related to incorrect, unexpected, or malicious input data by ensuring that string conversions to numbers are performed safely and accurately. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** |
| --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a |

| **Checker Description** |
| --- |
| This checker ensures that all container indices are validated, which is crucial for preventing out-of-bounds errors during string-to-number conversions. It's selected for its emphasis on boundary checking, which aligns with the need for robust data conversion routines. |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Prevent Buffer Overflow in String Operations |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | Ensures that string operations do not exceed the allocated memory, thereby preventing buffer overflows, a common source of security vulnerabilities. |   **Noncompliant Code** |
| The code attempts to copy a source string to a destination buffer without ensuring that the destination can accommodate the entire source string, including the null-terminator. |
| #include <cstring>  void unsafeCopy(const char \*src) {  char dest[10];  strcpy(dest, src); // Unsafe: may cause buffer overflow  } |

| **Compliant Code** |
| --- |
| The code uses strncpy() to copy the source string to the destination buffer, ensuring that no more than the size of the destination buffer is copied. It also manually null-terminates the destination string. |
| #include <cstring>  void safeCopy(const char \*src) {  char dest[10];  strncpy(dest, src, sizeof(dest) - 1); // Safe: limits the number of copied characters  dest[sizeof(dest) - 1] = '\0'; // Ensuring null-termination  } |

**Relevant Security Principle(s)**

| 8. Practice Defense in Depth |
| --- |
| **Rationalization:** Implementing checks to prevent buffer overflows in string operations aligns with the principle of layered security. By ensuring each string operation is safe, it adds an additional layer of protection against exploits. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** |
| --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.MEM.BU |

| **Checker Description** |
| --- |
| This checker focuses on detecting buffer underruns, which are closely related to buffer overflows, especially in string operations where off-by-one errors are common. It's chosen because it helps in identifying underflow scenarios that could lead to overflows in adjacent memory areas. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | Prevents SQL injection, a critical security vulnerability where an attacker can inject malicious SQL code through user input, potentially compromising data integrity and security. |   **Noncompliant Code** |
| This code constructs an SQL query by directly concatenating user input, which could allow an attacker to inject malicious SQL code. |
| #include <string>  #include <iostream>  void vulnerableQuery(const std::string& userInput) {  std::string query = "SELECT \* FROM data WHERE info = '" + userInput + "'";  std::cout << "Query: " << query << std::endl;  // Execute query  } |

| **Compliant Code** |
| --- |
| Uses parameterized queries to safely include user input in an SQL query. This method treats user input as data, preventing it from being interpreted as part of the SQL command. |
| #include <string>  #include <iostream>  // Assuming PreparedStatement is part of a database library  void safeQuery(const std::string& userInput) {  PreparedStatement stmt = PreparedStatement("SELECT \* FROM data WHERE info = ?");  stmt.setString(1, userInput);  std::cout << "Executing a safe query." << std::endl;  // Execute the prepared statement  } |

**Relevant Security Principle(s)**

| 1. Validate Input Data |
| --- |
| **Rationalization:** Validating all input data before constructing SQL queries prevents SQL injection, ensuring data integrity and system security. |

**Threat Level**

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

**Automation**

| Tool | Version | Checker |
| --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: STR53-CPP |

| Checker Description |
| --- |
| This checker identifies unsafe array accesses, including those with tainted indices, which can be analogous to the injection vectors in SQL injection attacks. It's selected for its ability to highlight risky data manipulations that could be exploited via SQL injection. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Ensure Correct Memory Allocation |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | Proper memory allocation is crucial for preventing buffer overflows and ensuring that sufficient space is allocated for data structures. This standard addresses the correct use of memory allocation functions to avoid errors such as using an incorrect size argument, which can lead to security vulnerabilities and unstable software behavior. |   **Noncompliant Code** |
| This noncompliant code example incorrectly uses the size of the pointer instead of the size of the pointed-to object for memory allocation, leading to insufficient memory allocation for a struct tm object. |
| #include <stdlib.h>  #include <time.h>  struct tm \*make\_tm(int year, int mon, int day, int hour, int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(tmb)); // Incorrect use of sizeof  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm){.tm\_sec = sec, .tm\_min = min, .tm\_hour = hour, .tm\_mday = day, .tm\_mon = mon, .tm\_year = year};  return tmb;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, sizeof(\*tmb) is correctly used to allocate memory for a struct tm object. This ensures sufficient memory allocation based on the size of the object pointed to by tmb. |
| #include <stdlib.h>  #include <time.h>  struct tm \*make\_tm(int year, int mon, int day, int hour, int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(\*tmb)); // Correct memory allocation  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm){.tm\_sec = sec, .tm\_min = min, .tm\_hour = hour, .tm\_mday = day, .tm\_mon = mon, .tm\_year = year};  return tmb;  } |

**Relevant Security Principle(s)**

| 4. Keep It Simple |
| --- |
| **Rationalization:** Simplifying memory allocation reduces the chance of errors, such as buffer overflows or memory leaks, enhancing system stability and security. |

**Threat Level**

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

**Automation**

| Tool | Version | Checker |
| --- | --- | --- |
| Helix QAC | 2023.3 | C++3162, C++3163, C++3164, C++3165 |

| Checker Description |
| --- |
| This set of checkers is dedicated to ensuring memory is allocated, used, and freed correctly, addressing common pitfalls in dynamic memory management. They're chosen for their comprehensive coverage of memory handling practices, essential for preventing leaks and corruption. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Correct Use of Assertions |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | This standard ensures that assertions are used correctly to validate assumptions in the code. Assertions should not perform assignments or other operations that could alter the program state or cause side effects. The intent is to use assertions solely as a debugging aid to check for conditions that should logically always be true in the executed code path. |   **Noncompliant Code** |
| This noncompliant code example mistakenly uses an assignment within an assertion. Such use can alter program state unexpectedly and lead to side effects, which is not the purpose of an assertion. |
| #include <cassert>  void updateValue(int& value, int newValue) {  // Incorrect: Assignment within an assertion  assert(value = newValue);  // Further code logic...  } |

| **Compliant Code** |
| --- |
| In this standard, assertions are strictly used for checking conditions without causing any side effects. The examples highlight the importance of using assertions correctly to ensure they serve their intended purpose of verifying assumptions in the code, without altering the program's state or behavior. |
| #include <cassert>  void updateValue(int& value, int newValue) {  value = newValue;  // Correct: Assertion checks the condition without side effects  assert(value == newValue);  // Further code logic...  } |

**Relevant Security Principle(s)**

| 10. Adopt a Secure Coding Standard |
| --- |
| **Rationalization:** Assertions are a development tool used to detect programming errors during the development phase. Aligning with secure coding standards ensures that assertions are used effectively to catch errors early without impacting the final product's security posture. |

**Threat Level**

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

**Automation**

| Tool | Version | Checker |
| --- | --- | --- |
| Astrée | 22.10 | assert\_failure |

| Checker Description |
| --- |
| This checker is focused on assertion failures, which are critical for debugging and ensuring code behaves as expected under various conditions. It's chosen for its direct relevance to monitoring and enforcing correct use of assertions within the codebase. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-STD | Proper Use of C++ Exceptions for Error Handling |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | This standard promotes the use of C++ exceptions to handle errors and unexpected conditions gracefully. Proper use of exceptions ensures that errors are caught and handled appropriately, maintaining program stability and preventing crashes or undefined behavior. |   **Noncompliant Code** |
| The noncompliant code example lacks proper exception handling. It uses functions that can throw exceptions (like std::stoi) without a try-catch block, risking program termination on encountering an error. |
| #include <string>  #include <iostream>  void parseAndProcess(const std::string& input) {  int number = std::stoi(input); // Risky: std::stoi can throw exceptions  // Processing with number  } |

| **Compliant Code** |
| --- |
| In the compliant solution, exceptions thrown by std::stoi are caught and handled, preventing the program from terminating unexpectedly. The code demonstrates how to use try-catch blocks effectively to manage errors. |
| #include <string>  #include <iostream>  #include <stdexcept>  void parseAndProcess(const std::string& input) {  try {  int number = std::stoi(input); // Safe: Exceptions are handled  // Processing with number  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid argument: " << e.what() << '\n';  } catch (const std::out\_of\_range& e) {  std::cerr << "Value out of range: " << e.what() << '\n';  } catch (const std::exception& e) {  std::cerr << "Unexpected error: " << e.what() << '\n';  }  } |

**Relevant Security Principle(s)**

| 9. Use Effective Quality Assurance Techniques |
| --- |
| **Rationalization:** Effective quality assurance techniques include robust error handling mechanisms. Proper use of C++ exceptions ensures that errors are managed in a controlled manner, enhancing the application's reliability and security by preventing unexpected crashes and vulnerabilities. |

**Threat Level**

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

**Automation**

| Tool | Version | Checker |
| --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.MEM.TBA |

| Checker Description |
| --- |
| This checker identifies tainted buffer accesses, indirectly related to error handling by preventing the use of potentially corrupted data. It's selected for its ability to enforce clean data handling, a prerequisite for robust exception-based error management. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation & Security | STD-008-CPP | Safe Handling of Format Strings in I/O Operations |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | This standard aims to prevent security vulnerabilities that can arise from improperly handling user input in formatted I/O functions. An attacker could exploit format string vulnerabilities to execute arbitrary code, view or modify memory content. Ensuring that format strings are used safely is crucial for the security and stability of the software. |   **Noncompliant Code** |
| The noncompliant code directly uses a user-supplied string (user) in a formatted output function ‘fprintf()’, potentially leading to a format string vulnerability. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void incorrect\_password(const char \*user) {  char msg[256];  sprintf(msg, "%s cannot be authenticated.\n", user); // Unsafe: user-controlled format string  fprintf(stderr, msg);  } |

| **Compliant Code** |
| --- |
| The compliant solution avoids using user-controlled strings as format specifiers. It safely incorporates user input into the output without interpreting it as a format string. |
| #include <stdio.h>  void incorrect\_password(const char \*user) {  static const char msg\_format[] = "%s cannot be authenticated.\n";  fprintf(stderr, msg\_format, user); // Safe: User input is not treated as a format string  } |

**Relevant Security Principle(s)**

| 1. Validate Input Data |
| --- |
| **Rationalization:** Input validation is crucial for security, especially in I/O operations where format string vulnerabilities can occur. Ensuring format strings are safely handled protects against injection attacks, aligning with the principle of validating all input data to maintain security integrity. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** |
| --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: STR53-CPP |

| **Checker Description** |
| --- |
| This checker ensures array and pointer operations are safe, which is vital for handling format strings securely to prevent vulnerabilities like format string attacks. It's chosen for its focus on validating data operations that are closely tied to secure I/O processing. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation & Security | STD-009-CPP | Robust External Input Validation |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | This standard emphasizes the importance of rigorously validating external inputs to ensure they meet the expected format, type, and range. Proper validation is critical in preventing various types of vulnerabilities, including injection attacks, data corruption, and unexpected behavior. |   **Noncompliant Code** |
| The noncompliant code fails to validate an external input before using it, which might lead to unexpected behavior or security vulnerabilities. |
| #include <string>  #include <iostream>  void processData(const std::string& inputData) {  // Process inputData directly without validation  std::cout << "Processing: " << inputData << std::endl;  } |

| **Compliant Code** |
| --- |
| The compliant solution includes checks to validate the input data for format, type, and range before processing it. |
| #include <string>  #include <iostream>  bool isValid(const std::string& data) {  // Implement validation logic (e.g., checking format, type, range)  return true; // Placeholder for actual validation logic  }  void processData(const std::string& inputData) {  if (isValid(inputData)) {  // Process inputData after validation  std::cout << "Processing: " << inputData << std::endl;  } else {  std::cerr << "Invalid input data." << std::endl;  // Handle invalid data  }  } |

**Relevant Security Principle(s)**

| 1. Validate Input Data |
| --- |
| **Rationalization:** External input validation is a cornerstone of secure coding, preventing a wide range of attacks by ensuring only expected and safe data is processed. This principle supports the creation of a secure code base by mitigating vulnerabilities associated with unvalidated inputs. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** |
| --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a |

| **Checker Description** |
| --- |
| This checker guarantees the safety of index operations, directly supporting the enforcement of stringent input validation routines. It's selected because validating indices and data ranges is foundational to securing external inputs against a variety of common attacks. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Managerment | STD-010-CPP | Proper Management and Cleanup of Dynamically Allocated Resources |

|  |  |  |
| --- | --- | --- |
| | **Rationalization** | | --- | | This standard addresses the efficient and safe management of dynamically allocated resources. Proper handling of such resources is essential to prevent memory leaks, avoid resource exhaustion, and ensure that the program operates within its resource constraints. |   **Noncompliant Code** |
| The noncompliant code demonstrates a scenario where dynamically allocated memory is not released, leading to a memory leak. |
| #include <cstdlib>  void createArray(int size) {  int\* arr = new int[size];  // Perform operations on arr  // Memory leak: arr is not deleted  } |

| **Compliant Code** |
| --- |
| The compliant solution ensures that dynamically allocated memory is properly released using delete[], preventing memory leaks. |
| #include <cstdlib>  void createArray(int size) {  int\* arr = new int[size];  // Perform operations on arr  delete[] arr; // Proper cleanup to prevent memory leaks  } |

**Relevant Security Principle(s)**

| 4. Keep It Simple |
| --- |
| **Rationalization:** Simplifying the management and cleanup of dynamically allocated resources reduces complexity, which in turn minimizes the risk of memory leaks and dangling pointers. This approach fosters more maintainable and error-free code, crucial for both security and application stability. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** |
| --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.MEM.TO |

| **Checker Description** |
| --- |
| This checker focuses on type overruns, which can lead to resource mismanagement issues. It's chosen for its ability to identify when dynamically allocated resources may exceed their intended use, ensuring proper management and cleanup to prevent leaks and potential exploits. |

### Defense-in-Depth Illustration

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



### Automation



Integrating automation into the DevSecOps cycle at Green Pace not only streamlines the development process but also embeds security at every stage, ensuring that coding standards are consistently met. From the initial planning and design phases, automated tools like SonarQube can be deployed to perform static code analysis, identifying potential security flaws before they progress further down the pipeline. This early detection is crucial for minimizing vulnerabilities and adhering to the security policy.

As the development moves into the build and deployment phases, automation plays a key role in enforcing compliance through automated testing frameworks that validate coding standards and security policies. Additionally, the integration of automated deployment tools ensures that applications are securely configured and deployed, reducing the risk of misconfigurations and vulnerabilities in production environments. In the maintenance and monitoring phases, automation tools continuously scan for and respond to emerging threats, keeping applications secure post-deployment. This comprehensive approach to automation across the DevSecOps cycle ensures that Green Pace’s applications are developed with security as a paramount concern, aligning with the company’s strategic objectives for secure, efficient, and reliable software delivery.

### Summary of Risk Assessments

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

| **Encryption Policies** | |
| --- | --- |
| Encryption in rest | This policy mandates encrypting data stored on any physical media. It applies to databases, file systems, and backups. The purpose is to protect data from unauthorized access if the storage medium is lost or stolen. Encryption algorithms like AES-256 should be used, with keys managed securely. |
| Encryption at flight | This policy requires data to be encrypted during transmission over networks. It applies to data exchanged between applications, services, and users over the internet or internal networks. SSL/TLS protocols should be used to ensure secure communication channels. |
| Encryption in use | This policy ensures data is encrypted even when being processed. It applies to data temporarily stored in memory or cache during computation. Techniques like homomorphic encryption are recommended, allowing operations on encrypted data without exposing its plaintext. |

| **Triple-A Framework Policies** | |
| --- | --- |
| Authentication | This policy mandates verifying the identity of users or systems before granting access to resources. It applies to user logins, system-to-system authentication, and device authentication. Methods include passwords, biometric verification, and multi-factor authentication. The policy ensures that only verified entities can access systems and data. |
| Authorization | After authentication, this policy controls the level of access granted to resources based on user roles or permissions. It applies to determining which files, databases, or services a user can access and modify. The policy uses mechanisms like Access Control Lists (ACLs) and role-based access control (RBAC) to enforce permissions. |
| Accounting | This policy involves tracking and recording user activities and system access to ensure accountability. It applies to logging user logins, database changes, file access, and system modifications. The policy requires maintaining secure, tamper-evident logs for auditing and forensic analysis. |

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 | 02/17/2024 | First Completed Version of Security Policy | Stanley Niles |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

**Matching Green Pace Coding Standards with SEI Coding Standards**

This appendix aims to align the Green Pace Secure Development Policy coding standards with the SEI (Software Engineering Institute) coding standards to demonstrate compliance and adherence to industry-recognized secure coding practices. Each Green Pace standard is mapped to the most relevant SEI standard(s), providing a reference for developers to understand how these guidelines interconnect and reinforce each other.

**Table of Correspondence between Green Pace and SEI Coding Standards**

|  |  |  |  |
| --- | --- | --- | --- |
| Green Pace Standard ID | Green Pace Standard Description | Corresponding SEI Standard(s) | SEI Standard Description |
| STD-001-CPP | Enforce Correct Integer Size Usage | INT30-C | Ensure that unsigned integer operations do not wrap |
| STD-002-CPP | Robust String-to-Number Conversion | EXP34-C, EXP36-C | Do not dereference null pointers; Do not cast pointers into more strictly aligned pointer types |
| STD-003-CPP | Prevent Buffer Overflow in String Operations | STR30-C | Do not attempt to modify string literals |
| STD-004-CPP | Prevent SQL Injection | N/A | General best practices for input validation |
| STD-005-CPP | Ensure Correct Memory Allocation | MEM30-C, MEM31-C | Do not access freed memory; Free dynamically allocated memory when no longer needed |
| STD-006-CPP | Correct Use of Assertions | N/A | General best practices for debugging and validation |
| STD-007-CPP | Proper Use of C++ Exceptions for Error Handling | ERR30-C, ERR34-C | Take care when reading errno; Detect errors when converting a string to a number |
| STD-008-CPP | Safe Handling of Format Strings in I/O Operations | FIO30-C | Exclude user input from format strings |
| STD-009-CPP | Robust External Input Validation | FIO30-C | Exclude user input from format strings |
| STD-010-CPP | Proper Management and Cleanup of Dynamically Allocated Resources | MEM30-C, MEM31-C | Do not access freed memory; Free dynamically allocated memory when no longer needed |

Complete list of SEI CERT C++ coding standards can be found at: <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>