**Green Pace Developer: Security Policy**



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc184575787)

[Contents 1](#_Toc184575788)

[Overview 2](#_Toc184575789)

[Purpose 2](#_Toc184575790)

[Scope 2](#_Toc184575791)

[Ten Core Security Principles 2](#_Toc184575792)

[C/C++ Ten Coding Standards 3](#_Toc184575793)

[Coding Standard 1 4](#_Toc184575794)

[Coding Standard 2 6](#_Toc184575795)

[Coding Standard 3 8](#_Toc184575796)

[Coding Standard 4 10](#_Toc184575797)

[Coding Standard 5 12](#_Toc184575798)

[Coding Standard 6 14](#_Toc184575799)

[Coding Standard 7 16](#_Toc184575800)

[Coding Standard 8 18](#_Toc184575801)

[Coding Standard 9 20](#_Toc184575802)

[Coding Standard 10 22](#_Toc184575803)

[Defense-in-Depth Illustration 24](#_Toc184575804)

[Automation 24](#_Toc184575805)

[Summary of Risk Assessments 25](#_Toc184575806)

[Policies for Encryption and Triple A 25](#_Toc184575807)

[Map the Principles 26](#_Toc184575808)

[Audit Controls and Management 28](#_Toc184575809)

[Enforcement 28](#_Toc184575810)

[Exceptions Process 28](#_Toc184575811)

[Distribution 29](#_Toc184575812)

[Policy Change Control 29](#_Toc184575813)

[Policy Version History 29](#_Toc184575814)

[Appendix A Lookups 29](#_Toc184575815)

[Approved C/C++ Language Acronyms 29](#_Toc184575816)

[Appendix B Related SEI Code Standards 29](#_Toc184575817)

## 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** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input data is one of the main ways to compromise a computer system. Without checking input data before use this can cause crashes, buffer overflows, SQL injections (if the input is used directly in the query), data overflows, etc. All input data needs to be checked before use to make sure it will not compromise the system in any way. |
| 1. Heed Compiler Warnings | Compiler warnings are there for a reason, not just an annoyance. They are an indication that something may be wrong with the code. It is important to not take the stance that you know better than the compiler. Yes, the compiler may flag something that it does not fully understand, but usually these warnings can be corrected to make for better more secure code. Any compiler warning should be looked at with the same scrutiny as a compiler error in order to make the software safe and secure. |
| 1. Architect and Design for Security Policies | Security should not just be thought about while writing code. To properly implement a robustly secure system, security needs to be thought about at each level of the SDLC. This includes the planning, design, and test phases. If security is incorporated into each of these phases there will less likely be a security issue when it comes to the final product. If the implementation phase of the SDLC is the only phase where security is considered, then the system will be much more vulnerable to security issues. |
| 1. Keep It Simple | Elegant and compact code isn’t always the best. Sometimes this causes code to be overly complicated and hard to follow. Keeping code simple, easy to read, and full of comments will make it much easier for code to be maintained in the future. |
| 1. Default Deny | A computer system should be created for a specific purpose. There should be no action or access that is given to a system if it does not help accomplish that purpose. That is the rationale behind this principle. Any action or access should be denied by default, and only if there is a specific rule to allow that action or access should it be allowed. This principle goes hand in hand with the below principle of least privilege. |
| 1. Adhere to the Principle of Least Privilege | No user should have full access to the system to perform their tasks. Users should only be given the necessary privileges to accomplish what they need to do and that’s it. If a user is given the privilege to execute a function, or access a database, or write to a file that they don’t have permission to then this creates a major security risk. By restricting unnecessary access, the risk of a security breach is greatly reduced. |
| 1. Sanitize Data Sent to Other Systems | Much like we should not trust input provided to us, other systems should not trust input that we provide. However, we cannot assume that other systems have taken the necessary precautions to validate and sanitize our inputs. So we need to make sure that any output that we provide to other systems have been checked to ensure they will not cause any intended harm. |
| 1. Practice Defense in Depth | Security attacks can come from many different sources. Security vulnerabilities can occur in many different ways. There is not a single tool or programming language that is free of these things. Defense in depth is a procedure in which security vulnerabilities can be caught by different layers. Each layer has specific things that it can prevent and certain things it cannot. If all layers are combined, any holes within one layer should be caught by another. These layers need to be designed and implemented to create a much more secure system. |
| 1. Use Effective Quality Assurance Techniques | No company wants to release a program full of bugs and crashes. Quality Assurance and testing is the last step before a program makes it out the door. Quality tests need to be created and ran in order to catch any issues with the program. Tests related to common security vulnerabilities (data input, buffer overflow, SQL injection, etc.) are essential. If principle 3 is adhered to, then designing tests with security in mind will become a lot easier. |
| 1. Adopt a Secure Coding Standard | Creating computers systems is difficult. When there are many employees all contributing to the same project it becomes even more difficult. Having a specific coding standard allows all employees to be on the same page and create consistent code. The coding standard should give guidance and examples of what to do and what not to do when it comes to the code. This allows the project to be developed more smoothly because anyone can pitch in to help pick up the slack, or new employees can be added to the project and there will be less confusion or down time to get up and running and contributing. This also helps maintain the code in the future. Less time will be needed to understand what the code is doing and how it works if the coding standard is known and can be referenced. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Use Minimum Data Types** |
| --- | --- | --- |
| **Data Type** | [STD-DDT-001] | Data types should be defined as the minimum storage size to hold the maximum value that is intended by the variable. |

| **Noncompliant Code** |
| --- |
| The below code block defines a variable to hold a year value, but is only defined to be an unsigned char. The data type should be able to hold a year larger than 255. |
| unsigned char year;  year = 2024; // Incorrect, unsigned char cannot hold this value |

| **Compliant Code** |
| --- |
| The below code block defines a variable to hold a year value as an unsigned short. The data type should be able to hold the largest year because the maximum value is 65535. |
| unsigned short year;  year = 2024; // Correct, unsigned short can hold this value |

| **Principles(s):**  2: Some compilers can be set to check strict things like incorrect types and issue warnings if detected, so the principle of addressing compiler options relates to this standard.  4: Using correct types is a way to keep code as simple as possible. Using incorrect types for usage complicates code unnecessarily. Keeping code simple relates the principle to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-INT00** |  |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **559, 705, 706, 2403** | Assistance provided: Reports data type inconsistencies in format strings |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rec. INT00-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.int00c.html) | Checks for:   * Use of basic numerical types instead of typedef-s * Integer overflow or integer constant overflow * Format string specifiers and arguments mismatch   Rec. partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.33 | [**V629**](https://pvs-studio.com/en/docs/warnings/v629/)**,** [**V5004**](https://pvs-studio.com/en/docs/warnings/v5004/) |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Check for Wrap Around** |
| --- | --- | --- |
| **Data Value** | [STD-DTV-001] | When performing integer operations, wrap around should be checked and processed accordingly. |

| **Noncompliant Code** |
| --- |
| The below code performs an addition on two unsigned shorts but does not test whether this will cause wrap around. |
| unsigned short num1\_p\_num2;  unsigned short num1 = 60000;  unsigned short num2 = 30000;  num1\_p\_num2 = num1 + num2; // Incorrect, the addition will wrap around |

| **Compliant Code** |
| --- |
| The below code properly checks for the unsigned short wrap around and informs the user. |
| unsigned short num1\_p\_num2;  unsigned short num1 = 60000;  unsigned short num2 = 30000;  if (num1 > USHRT\_MAX – num2) // correctly check for wraparound  {  /\* inform user of wrap around error \*/  }  else  {  num1\_p\_num2 = num1 + num2; // has no chance of wraparound  } |

| **Principles(s):**  2: Some compilers can be set to check strict things like integer wrap around and issue warnings if detected, so the principle of addressing compiler options relates to this standard.  8: Integer wrap around could lead to code execution by an attacker so preventing this in the code is one layer in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **integer-overflow** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=125337650) | 7.2.0 | **CertC-INT30** | Implemented |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **INTEGER\_OVERFLOW** | Implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rule INT30-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint30c.html) | Checks for:   * Unsigned integer overflow * Unsigned integer constant overflow   Rule partially covered. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do Not Use Unbounded String Operations** |
| --- | --- | --- |
| **String Correctness** | [STD-STC-001] | String manipulation functions that do not check if they copy beyond bounds (strcpy(), sprintf(), strcat(), etc.) should not be used. Instead safer functions like strncpy should be used to guarantee nothing is written beyond the bounds of the output string. |

| **Noncompliant Code** |
| --- |
| The below function uses strcpy(), but does not check that the length of the input string is less than or equal to the length of the output string. |
| char dest\_str[10];  char src\_str[] = “0123456789”;  strcpy(dest\_str, src\_str); // Incorrect, this will cause an extra  // byte of 0 to be written past the end  // of dest\_str |

| **Compliant Code** |
| --- |
| The below function uses strncpy() to make sure nothing gets copied beyond the bounds of the output string.  NOTE: strncpy() does not guarantee that the string is null-terminated (See STD-STC-002). |
| char dest\_str[10];  char src\_str[] = “0123456789”;  strcnpy(dest\_str, src\_str, sizeof(dest\_str)); // Better, this will  // not cause a write beyond the end of dest\_str, but a null  // terminator should be checked for (see STD-STC-002) |

| **Principles(s):**  2: Compilers should issue warnings if any deprecated or unsafe string function exists within the code, so the principle of addressing compiler options relates to this standard.  8: String manipulation issues can lead to serious vulnerabilities. These issues must be recognized and fixed in the code in order to adhere in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 |  | Supported |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR07** |  |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rec. STR07-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str07c.html) | Checks for:   * Use of dangerous standard function * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rec. partially covered. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151949) | 3.11 | [**S1081**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-1081) |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use Prepared Statements** |
| --- | --- | --- |
| **SQL Injection** | [STD-SQI-001] | SQL injection is a serious threat. Prepared statements are queries where user input is not added directly into the query. Instead a statement is prepared, and then the user inputs are added, this eliminates the possibility of a user changing the actual structure of a query. |

| **Noncompliant Code** |
| --- |
| The below code uses user input directly to create an SQL query. This is vulnerable to SQL injection. |
| void runQuery(std::string user\_input)  {  sql::Connection\* connection;  sql::Statement\* r\_statement;  sql::ResultSet\* result  /\* processing \*/  // Below query uses user input directly which is vulnerable to  // SQL injection  std::string query = “SELECT \* FROM USERS WHERE NAME = “;  query.append(user\_input);  statement = con->createStatement();  result = stmt→executeQuery(query);  } |

| **Compliant Code** |
| --- |
| The below code uses a prepared statement so that the user input cannot modify the intent of the SQL query. |
| void runQuery(std::string user\_input)  {  sql::Connection\* connection;  sql::PreparedStatement\* p\_statement;  sql::ResultSet\* result  /\* processing \*/  // Below query uses a prepared statement to remove  // SQL injection potential  p\_statement = con->prepareStatement(  "SELECT \* FROM USERS WHERE NAME = ?");  p\_statement->setString(1, user\_input);  result = p\_statement->executeQuery(); // not vulnerable  // to SQL Injection  } |

| **Principles(s):**  1. SQL injections can occur based on the input of the user. This input needs to be validated and sanitized before being used in an SQL query, so the principle directly relates to this standard.  5: SQL injections can cause actions or access to something that is supposed to be restricted. By protecting against SQL injection, the principle of default deny is adhered to by this standard.  8: SQL injection is one of the most serious threats to a system. Data can be stolen, corrupted, or deleted very easily by this vulnerability. This issue must be recognized and fixed in the code in order to adhere in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **108 D, 109 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rec. STR02-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str02c.html) | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Don’t Mix and Match Memory Management Functions** |
| --- | --- | --- |
| **Memory Protection** | [STD-MEP-001] | Memory allocation/deallocation functions for C++ are new/delete. However, C memory management functions may be used such as malloc/free. These should not be mixed and matched. If new is used, delete should be used. If malloc is used, free should be used. |

| **Noncompliant Code** |
| --- |
| The below code block allocates a memory block using new (C++), but then attempts to deallocate it using free (C). |
| Book\* bp = new Book();  free (bp); // Incorrect, delete should be used |

| **Compliant Code** |
| --- |
| The below code block properly allocates a memory block using new (C++), and then deallocates it using delete (C++) |
| Book\* bp = new Book();  delete bp; // Correct, no mismatch of memory functions |

| **Principles(s):**  2. Compilers should issue warnings if any memory that was allocated does not have a corresponding deallocation within the same scope. Although this may not catch all issues related to this standard it should be able to catch some, so the principle of addressing compiler options relates to this standard.  4. Using consistent allocation and deallocation functions throughout the entire code base keeps things simple. Trying to use different allocation and deallocation methods in various places will cause the code to be complex. Keeping the code simple relates the principle to this standard.  8: Incorrectly deallocating memory can cause undefined behavior that can lead to serious vulnerabilities. These issues must be recognized and fixed in the code in order to adhere in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **invalid\_dynamic\_memory\_allocation dangling\_pointer\_use** |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MEM51** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MEM51-a** **CERT\_CPP-MEM51-b** **CERT\_CPP-MEM51-c** **CERT\_CPP-MEM51-d** | Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: MEM51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem51cpp.html) | Checks for:   * Invalid deletion of pointer * Invalid free of pointer * Deallocation of previously deallocated pointer   Rule partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Do Not Modify Values in Assertions** |
| --- | --- | --- |
| **Assertions** | [STD-ASS-001] | Assertions are not evaluated when a program is compiled in release mode. This means that all assertions should simply be a comparing expression, nothing more. |

| **Noncompliant Code** |
| --- |
| The below code block performs an increment inside an ASSERT statement. This will not be performed in release mode, so it should not be done. |
| ASSERT(numEggs++ < 10); // Incorrect, the increment  // will not happen in release mode |

| **Compliant Code** |
| --- |
| The below code correctly increments the variable before the ASSERT statement so that it is always evaluated. |
| numEggs++; // Correct, the increment will always happen  ASSER(numEggs < 10); |

| **Principles(s):**  2. Compilers should issue warnings if any data is manipulated within an assert statement, so the principle of addressing compiler options relates to this standard.  4. Using asserts is a good way to perform validation testing so something bad does not occur later in the code execution, but the assert statements should be very simple like comparing a single variable to a value. They should not be complex at all. Keeping the code simple relates the principle to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **ASSERT\_SIDE\_EFFECTS** | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-PRE31-b** **CERT\_C-PRE31-c** **CERT\_C-PRE31-d** | Assertions should not contain assignments, increment, or decrement operators Assertions should not contain function calls nor function-like macro calls Avoid side effects in arguments to unsafe macros |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rule PRE31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulepre31c.html) | Checks for side effect in arguments to unsafe macro (rule partially covered) |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exceptions Should be Handled** |
| --- | --- | --- |
| **Exceptions** | [STD-EXC-001] | Handlers should be added for any exception. Abnormal program termination is a way for certain security attacks to occur. Even if the program cannot recover from the exception, it should still be handled. |

| **Noncompliant Code** |
| --- |
| The below code calls a function that could potentially throw an exception, but it is not handled at all, which will result in abnormal program termination. |
| funtionThatMayExcept(); // Incorrect, the exception is not handled |

| **Compliant Code** |
| --- |
| The below code surrounds the potential exception throwing function with a try...catch block to catch and process the exception. |
| try  {  funtionThatMayExcept();  }  catch  {  /\* handle exception \*/  } |

| **Principles(s):**  8: An exception means something bad has happened and can indicate a potential attack. This is because during normal code execution, an exception is usually not expected. Making sure exceptions are handled provides a layer of protection for the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.3 | **MISRA.CATCH.ALL** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Avoid Integer Truncation** |
| --- | --- | --- |
| **Data Value** | [STD-DTV-002] | Care must be taken so that inadvertent integer truncation does not occur. This could cause calculations to be incorrect possibly causing errors in the system. |

| **Noncompliant Code** |
| --- |
| The below code incorrectly sets an unsigned char value to a variable of type unsigned int. This is a potential truncation issue that may cause a later error when the truncated value is used. |
| unsigned int full\_value = 10000;  unsigned char used\_value = full\_value; // Incorrect, used\_value  // will not contain 10000  /\* perform processing with used\_value \*/ |

| **Compliant Code** |
| --- |
| The below code correctly avoids truncation by defining the second variable as the same type as the first. |
| unsigned int full\_value = 10000;  unsigned int used\_value = full\_value; // Correct, no truncation  // is possible  /\* perform processing with used\_value \*/ |

| **Principles(s):**  2: Some compilers can be set to check strict things like possible integer truncation and issue warnings if detected, so the principle of addressing compiler options relates to this standard.  8: Similar to integer wrap around, truncation of integers could also lead to code execution by an attacker so preventing this in the code is one layer in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.CAST.PC.AV LANG.CAST.PC.CONST2PTR LANG.CAST.PC.INT**  **LANG.CAST.COERCE LANG.CAST.VALUE**  **ALLOC.SIZE.TRUNC MISC.MEM.SIZE.TRUNC**  **LANG.MEM.TBA** | Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer  Coercion alters value Cast alters value  Truncation of allocation size Truncation of size  Tainted buffer access |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Can detect violations of this rule. However, false warnings may be raised if limits.h is included |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rule INT31-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint31c.html) | Checks for:   * Integer conversion overflow * Call to memset with unintended value * Sign change integer conversion overflow * Tainted sign change conversion * Unsigned integer conversion overflow   Rule partially covered. |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **signed\_downcast** | Exhaustively verified. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Null Terminate All Strings** |
| --- | --- | --- |
| **String Correctness** | [STD-STC-002] | Strings should be checked to make sure they are null terminated before being used. A non-null terminated string can cause things like endless loops or memory being overwritten if not recognized. |

| **Noncompliant Code** |
| --- |
| The below code does not explicitly null terminate a string and then uses that string in a loop, potentially overwriting memory. |
| char dest\_str[10];  char src\_str[] = “0123456789”;  strcnpy(dest\_str, src\_str, sizeof(dest\_str));  int i = 0;  while (dest\_str[i] != ‘\0’)  {  // ERROR, string was not null terminated so  // this may become an endless loop or overwrite memory  /\* perform processing \*/  i++;  } |

| **Compliant Code** |
| --- |
| The below code correctly null terminates the string before using it in the loop. This guarantees that memory will not be overwritten. |
| char dest\_str[10];  char src\_str[] = “0123456789”;  strcnpy(dest\_str, src\_str, sizeof(dest\_str));  dest\_str[sizeof(dest\_str)-1] = ‘\0’; // Correct, string is now  // properly null terminated  int i = 0;  while (dest\_str[i] != ‘\0’)  {  /\* perform processing \*/  i++;  } |

| **Principles(s):**  8: String manipulation issues can lead to serious vulnerabilities. These issues must be recognized and fixed in the code in order to adhere in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.MEM.BO LANG.MEM.TO MISC.MEM.NTERM BADFUNC.BO.\*** | Buffer overrun Type overrun **No space for null terminator** A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| [Coverity](https://www.securecoding.cert.org/confluence/display/seccode/Coverity) | 2017.07 | **STRING\_OVERFLOW BUFFER\_SIZE OVERRUN STRING\_SIZE** | Fully implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rule STR31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr31c.html) | Checks for:   * Use of dangerous standard function * **Missing null in string array** * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Avoid Freeing Memory Multiple Times** |
| --- | --- | --- |
| **Memory Protection** | [STD-MEP-002] | All dynamic memory allocated should be deallocated properly per STD-MEP-001. However, care must be taken that memory is not deallocated more than once. |

| **Noncompliant Code** |
| --- |
| The below code block incorrectly frees the allocated memory more than once. |
| Book\* bp = new Book();  /\* perform processing on bp \*/  delete bp;  /\* perform other processing \*/  delete bp; // Incorrect, bp was already deleted |

| **Compliant Code** |
| --- |
| The below code correctly only frees the allocated memory once. |
| Book\* bp = new Book();  /\* perform processing on bp \*/  /\* perform other processing \*/  delete bp; // Correct, bp is only deleted once |

| **Principles(s):**  2. Compilers should issue warnings if any memory that was deallocated is attempted to be deallocated again, so the principle of addressing compiler options relates to this standard.  4. Making sure deallocation only happens once is a form of keeping the code simple, so the principle relates to this standard.  8: Deallocating memory more than once can lead to serious vulnerabilities. These issues must be recognized and fixed in the code in order to adhere in the defense in depth strategy, so the principle relates to this standard.  9: The static analysis tools listed below can be helpful in finding and correcting any violations of this coding standard. The static analysis tools are a form of quality assurance, so the principle relates to this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **ALLOC.DF ALLOC.LEAK** | **Double free** Leak |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **RESOURCE\_LEAK** | Fully implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rec. MEM00-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.mem00c.html) | Checks for:   * Invalid free of pointer * **Deallocation of previously deallocated pointer** * Use of previously freed pointer   Rec. partially covered. |

### Defense-in-Depth Illustration

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



### Automation



In order to provide the best automation procedure for complying with the security coding standards in this policy document, the DevSecOps diagram should be modified to insert a step between the Build and Verify and test stages. After the Build stage of development, the code has been written. All of the tools for automatic detection of coding standard violations are static analysis tools. These tools can only be run on code that has been written. It makes sense to run these tools at the end of the Build step, but before the formal verification tests are created. This ensures that time is not wasted during the Verify and test step and that all coding standard violations have already been addressed before reaching this step. So, in reality the automation is at the end of the build step, but it makes more sense to include a separate step to show the importance that automation has on following the coding standards.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-DDT-001 | High | Unlikely | High | Medium | 2 |
| STD-DTV-001 | High | Likely | High | Medium | 2 |
| STD-DTV-002 | High | Probable | High | Medium | 2 |
| STD-STC-001 | High | Probable | Medium | High | 1 |
| STD-STC-002 | High | Likely | Medium | High | 1 |
| STD-SQI-001 | High | Likely | Medium | High | 1 |
| STD-MEP-001 | High | Likely | Medium | High | 1 |
| STD-MEP-002 | High | Probable | Medium | High | 1 |
| STD-ASS-001 | Low | Unlikely | Low | Low | 3 |
| STD-EXC-001 | Low | Probable | Medium | Low | 3 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is used for data that is not actively being used or transmitted within a network. This would include things like data stored in databases or just data in general stored on a server somewhere. It is important to encrypt this data because it usually includes things like usernames, passwords, or other personal information for users of the system. Attackers can attempt to access these servers or databases and steal this data for personal or financial gain or to gain further access to a system if they are able to steal login information. In order to prevent the misuse of this data even if it is stolen, it is important that all important data stored be encrypted. The encryption algorithm used for this data can be more complex and take more time because this is data that is not used as often. The encryption will be done before the data is stored into the database or on the server. |
| Encryption in flight | Encryption in flight is used for data that is being transmitted across a network. Attackers can listen to internet traffic and decode packets in order to extract information from them. Any sort of sensitive information should be encrypted prior to being sent across a network. Common things sent across the network are login information, personal information, and account information for users of the system. A good rule of thumb is that any data that was encrypted at rest should be encrypted in flight as well. Transport Layer Security or TLS is a common standard that is used encrypt data in flight and is recommended. The encryption will be done prior to sending the data across the network, and will be decrypted once it is received. |
| Encryption in use | Encryption in use refers to data that has been loaded into memory while a program is running. This data is normally not able to be seen or utilized by anything other than the program using it. However, because of other security vulnerabilities like buffer overflows and memory management errors, things like remote code execution or jumping to functions that should not be able to be accessed directly can happen. Because of these other security vulnerabilities, sensitive data loaded into memory for use should be encrypted as well. Depending on how much sensitive data is loaded into the system and how often it is used, the encryption algorithm used may need to be an algorithm that takes less time to encrypt/decrypt because before using any of the encrypted data it must be decrypted. This may take time so an analysis must be done to determine the best algorithm to use. The encryption will be done prior to saving the data off in memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a security process that determines if a user is allowed to use the system. This is typically done in the form of usernames and passwords. When a user attempts to login their credentials are sent to the system to verify that a valid user with that name and password matches one within the system. If there is a match, then the user is allowed in. To protect the sensitivity of the passwords these should not be stored in plain text, instead they should be encrypted using the encryption at rest policy described above. Account information can sometimes be compromised so a good policy would be to incorporate two-factor authentication as an extra layer of security to make sure someone isn’t using someone else’s credentials. Two-factor authentication means that in addition to the password a secure code will be sent to a device or e-mail registered to the user for them to also send to the system to make sure there is a match. |
| Authorization | Authorization is a security process that makes sure users are only able to perform actions that they have been restricted to perform. The principle of least privilege states that a user should only be allowed to perform the operations necessary to complete their task and nothing more, and this principle should be followed. It doesn’t make sense to allow a random user to go in and modify tables to a database. Someone with administrator privileges should be allowed to do those type of things, but they should be restricted from normal users. An administrator should also be the only one allowed to add and remove users as well as set the privileges of other users. |
| Accounting | Accounting is a security process that refers to logging information about the use of the system. When users log in, where they logged in from, how long they are logged in for, what actions did they perform, and what files did they modify are some of the things that need to be logged so that they can be reviewed if an attack were to occur. Other things that should be logged are modifications to databases and account information. If an attack does occur the logs can be helpful to narrow down who performed the attack, how they performed the attack, and even give insight on how to fix the vulnerability so an attack does not happen in the future. These accounting logs need to be kept in a secure separate location from the sensitive databases and other information for the system. This way an attacker who has infiltrated the system cannot simply find and delete the logs. |

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 11/17/2024 | Initial Template | Andrew Laipple |  |
| 2.0 | 12/08/2024 | Filled out threat level, automation and principles sections for each coding standard. Added an automation description for the devsecops process. Filled out policies for encryption and triple A defense. Added appendix B. | Andrew Laipple |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

## Appendix B Related SEI Code Standards

| Standard (with link) | Description |
| --- | --- |
| [INT00-C](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152417) | Understand the data model used by your implementation(s) |
| [INT30-C](https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap) | Ensure that unsigned integer operations do not wrap |
| [STR07-C](https://wiki.sei.cmu.edu/confluence/display/c/STR07-C.+Use+the+bounds-checking+interfaces+for+string+manipulation) | Use the bounds-checking interfaces for string manipulation |
| [STR02-C](https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems) | Sanitize data passed to complex subsystems |
| [MEM51-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources) | Properly deallocate dynamically allocated resources |
| [PRE31-C](https://wiki.sei.cmu.edu/confluence/display/c/PRE31-C.+Avoid+side+effects+in+arguments+to+unsafe+macros) | Avoid side effects in arguments to unsafe macros |
| [ERR51-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) | Handle all exceptions |
| [INT31-C](https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data) | Ensure that integer conversions do not result in lost or misinterpreted data |
| [STR31-C](https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) | Guarantee that storage for strings has sufficient space for character data and the null terminator |
| [MEM00-C](https://wiki.sei.cmu.edu/confluence/display/c/MEM00-C.+Allocate+and+free+memory+in+the+same+module%2C+at+the+same+level+of+abstraction) | Allocate and free memory in the same module, at the same level of abstraction |