**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 6](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 14](#_Toc52464063)

[Coding Standard 6 16](#_Toc52464064)

[Coding Standard 7 18](#_Toc52464065)

[Coding Standard 8 20](#_Toc52464066)

[Coding Standard 9 23](#_Toc52464067)

[Coding Standard 10 25](#_Toc52464068)

[Defense-in-Depth Illustration 27](#_Toc52464069)

[1. Automation 27](#_Toc52464074)

[2. Summary of Risk Assessments 28](#_Toc52464075)

[3. Policies for Encryption and Triple A 28](#_Toc52464076)

[Audit Controls and Management 30](#_Toc52464078)

[Enforcement 30](#_Toc52464079)

[Exceptions Process 30](#_Toc52464080)

[Distribution 32](#_Toc52464081)

[Policy Change Control 32](#_Toc52464082)

[Policy Version History 32](#_Toc52464083)

[Appendix A Lookups 32](#_Toc52464084)

[Approved C/C++ Language Acronyms 32](#_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** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input means leveraging control over untrusted data coming from external sources. This looks like encasing input-fed values in try/catch blocks to catch exceptions from faulty data, using regular expressions to detect injection-based attacks, and otherwise imposing strict limitations on what input values are passable to the rest of the system. Controlling acceptable input means protecting against attacks *and* human error, such as by enforcing a format the user must follow when entering a date, for example. |
| 1. Heed Compiler Warnings | Compiler warnings exist to inform the developer of specific issues with their code. Warnings may not prevent compilation but ignoring them can result in vulnerabilities by means of using a deprecated or a poorly organized structure. Heeding compiler warnings means editing code until no further warnings are thrown. |
| 1. Architect and Design for Security Policies | Designing software architecture with security policies in mind means including security in the design phase of the software development lifecycle (SDLC) rather than at the end. This kind of planning makes it easier for developers to implement security best practices as they code, resulting in more secure programs. |
| 1. Keep It Simple | The simpler the code, the easier it is to manage. For large, real-world software composed of thousands or millions of lines, overly complex code makes it that much harder to develop a patch, and it also introduces more entryways for a would-be attacker. Simple code is accessible for any team member to jump in and fix, and it limits the number of vulnerabilities a program might have. |
| 1. Default Deny | When providing permissions to users of a system, the default scheme should be to deny access at all levels. Then, access permission is granted for each level that’s appropriate to the user. Maintaining a default of denial is a safer choice than manually blacklisting what a user *cannot* access, as the latter makes it easier for a malicious user to find a workaround. |
| 1. Adhere to the Principle of Least Privilege | When a task is executed in the system, it should use the least privileges necessary to successfully execute. This is like writing simple code in that unnecessary complexity provides more entries for an attack. If a task does not require admin-level permission, it should *not* be executed with it. Needless high-level privilege opens possibilities for attacks that seek greater access. The more tasks there are with high-level permissions, the more targets for attack a program has. |
| 1. Sanitize Data Sent to Other Systems | Similar to validating input, some data must be sanitized before it’s passed to subsystems like a database. Data sanitization is the process of removing unneeded or dangerous data from storage, and in this context, it means carefully constricting the format of data sent to subsystems by means of white- and blacklisting. The entirety of the data may not be rejected, but some of it may be removed if it does not comply. For example, a string with blacklisted characters may be altered to pass *without* those characters. This practice keeps sensitive subsystems safe from attacks and human error. |
| 1. Practice Defense in Depth | Defense in depth means, when one security measure is breached, others exist to stop the breach from going any further. To illustrate with a metaphor, this is akin to an intruder breaking through one door but finding five locked doors ahead of them, each with different kinds of locks. This principle works to prevent as much damage as possible in an attack and, ultimately, enclose the most important parts of a system (e.g., sensitive information, systemwide access) in an impenetrable shell. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques are useful for catching vulnerabilities missed during the SDLC. Various forms of static and dynamic testing probe completed code for areas that need patching before a system is launched for use. Having a non-development team rigorously test code allows it to be viewed without bias, possibly resulting in greater bug detection. |
| 1. Adopt a Secure Coding Standard | A secure coding standard is a best practice for developers that emphasizes developing with security throughout, rather than at the end of (if ever), the SDLC. This means developers write code with security in mind, such as by composing unit and integrated tests as new branches are added, or by implementing an input validation check when data is sought from the user or another external source. By emphasizing security throughout the SDLC, stronger code with less vulnerabilities is produced, and resources are saved as compared to the post-development scramble of patching exposed vulnerabilities. |

### C/C++ Ten Coding Standards (see following page for start)

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | “INT00-C. Understand the data model used by your implementation(s).” (SEI CERT Coding Standard)  Ensure that correct data types are chosen based on the type and amount of input they’re expected to hold.  Choosing correct data types goes along with preventing under- and overflow issues. It also follows best practices for simple, human readable code by following expected logic. |

| **Noncompliant Code** |
| --- |
| In the code example below, there’s a lot going wrong with data types and logic. The float variable is assigned a string, and even then, the number within it (“80”) isn’t a floating-point number. Further, while “isHot” is technically allowed, it’s bad practice to store what should be a Boolean true/false as a string. |
| #include <iostream>  #include <string>  using namespace std;  int main() {  float temperature = "80F";  string isHot = "false";  if (temperature >= 80) {  isHot = "true";  }  else {  isHot = "false";  }  cout << isHot << endl;  } |

| **Compliant Code** |
| --- |
| The compliant code below works to correct the above mistakes by assigning appropriate values and logic when dealing with float and bool data types. |
| #include <iostream>  #include <string>  using namespace std;  int main() {  float temperature = 80.0;  bool isHot = false;  if (temperature >= 80.0) {  isHot = true;  }  else {  isHot = false;  }  cout << isHot << endl;  } |

| **Principles(s):**  4 – Keep it Simple. Sticking to logical data types (like int for a whole number value that *never* becomes a fraction) makes the code human readable and prevents unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PC-lint Plus | 1.4 | 559, 705, 706, 2403 | Assistance provided: Reports data type inconsistencies in format strings |
| Polyspace Bug Finder | R2024b | CERT C: Rec. INT00-C | 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. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | “INT32-C. Ensure that operations on signed integers do not result in overflow.” (SEI CERT Coding Standard)  This standard is valuable for protecting data integrity when it comes to values different data types can hold. Signed integers are one example of a data type that must be validated when accepting input to ensure they are not overflowing, thus resulting in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The code below results in overflow when various data types are put through addition. Because no check is in place to detect whether a data type can continue to hold an increasing amount, overflow goes unchecked and results in faulty values. NOTE: This is from the Module 1 assignment. |
| template <typename T>  T add\_numbers(T const& start, T const& increment, unsigned long int const& steps)  {  T result = start;  for (unsigned long int i = 0; i < steps; ++i)  {  result += increment;  }  return result;  } |

| **Compliant Code** |
| --- |
| The code below is the fix developed to prevent overflow for various data types undergoing addition. Using the limits library, overflow is detected before it occurs and damages data integrity. NOTE: This is from the Module 1 assignment. |
| #include <limits.h>  template <typename T>  T add\_numbers(T const& start, T const& increment, unsigned long int const& steps)  {  T result = start;  for (unsigned long int i = 0; i < steps; ++i)  {  // Check for overflow before adding next increment  // Logic is: if the result is greater than the max value minus the increment, then one more increment will overflow  if (result > std::numeric\_limits<T>::max() - increment) {  std::cout << "Overflow detected! Cannot add " << +increment << " to " << +result << ", please consider using a different data type." << std::endl;  return std::numeric\_limits<T>::max(); // Break the loop  }  // If no overflow is detected, addition can continue safely  else {  result += increment;  }  }  // Return the result after incrementing is complete, i.e. no overflow occurred  return result;  } |

| **Principles(s):**  2 – Heed compiler warnings. A compiler or external checker tool may alert the developer to an unsuited data type for a value, such as if a value will under- or overflow the type. Listening to this warning and changing the type is key for avoiding unwanted behaviors or a program vulnerability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Astrée | 24.04 | Integer-overflow | Fully checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | “STR02-C. Sanitize data passed to complex subsystems.” (SEI CERT Coding Standard)  Validate and sanitize strings before passing them to subsystems like a database or command shell.  Sanitizing strings means reducing them to only their acceptable content so that they don’t harm the subsystems they’re being passed to. For example, a string with an injection command would be caught and flagged before passing to a database. In case of human error, a string might be corrected for proper syntax (such as by switching \ to /) before being run in a command shell. |

| **Noncompliant Code** |
| --- |
| The example below shows a string being read from input without any validation, passing the input variable straight to the system. |
| #include <iostream>  #include <string>  using namespace std;  int main() {  string input;  cout << "Enter name: ";  getline(cin, input);  cout << "Hello, " << input << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The code below adds a simple check for ‘<’ and ‘>’ in the string, which could indicate nefarious behavior. Blacklisting like this isn’t the best or only step to take when ensuring string correctness, but given the unlikelihood of a user having such characters in their name, in this example, it’s a safe step toward defense in depth. |
| #include <iostream>  #include <string>  using namespace std;  int main() {  string input;  cout << "Enter name: ";  getline(cin, input);  if (input.find('<') != std::string::npos || input.find('>') != std::string::npos) {  std::cout << "Invalid input.\n";  return 1;  }  cout << "Hello, " << input << endl;  return 0;  } |

| **Principles(s):**  1 – Validate input.  7 – Sanitize data sent to other systems.  Properly validating input can prevent damage due to malicious intent or just human error. Validating and sanitizing strings that communicate with a database or command shell is so vital here to prevent unintended damage to integral systems. While validation looks for correct input, sanitization may truncate or replace parts of input to make them suitable for subsystem use. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | “IDS00-J. Prevent SQL injection.” (SEI CERT Coding Standard)  Preventing SQL injection is done via careful input validation and parameterizing queries to ensure a user gets no more access to a database than necessary. |

| **Noncompliant Code** |
| --- |
| The code below accepts queries without question and only halts when an error is detected, resulting in injection. NOTE: This code is from the Module 2 assignment. |
| #include "sqlite3.h"  bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  // clear any prior results  records.clear();  char\* error\_message;  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "Data failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  return true;  } |

| **Compliant Code** |
| --- |
| The code below uses the <regex> library to detect tautology patterns in a user’s query like ‘1=1’ that are commonly used to perpetrate injection. NOTE: This code is from the Module 2 assignment. |
| #include "sqlite3.h"  bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  // clear any prior results  records.clear();  // convert the query (stored in localCopy variable) to lowercase for consistent checking  // localCopy logic borrowed from run\_query\_injection()  std::string localCopy(sql);  std::transform(localCopy.begin(), localCopy.end(), localCopy.begin(), ::tolower);  // using Regex library, establish the tautology pattern to check for when testing the query  // this pattern checks for any 1+ characters, numbers, or apostrophes being set as equal to themselves  std::regex pattern(R"(\b([a-z0-9']+)\s\*=\s\*\1\b)");  // injection suspected - conditional check for tautology pattern  if (std::regex\_search(localCopy, pattern))  {  // Return suspicious query as-is (not localCopy) to avoid divulging the use of ::tolower  std::cout << "\nSQL injection suspected! Blocked query: " << sql << std::endl;  return false;  }  char\* error\_message;  // data retrieval failure - generic error message  if (sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "Data failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  return true;  } |

| **Principles(s):**  1 – Validate input.  6 – Adhere to the principle of least privilege.  7 – Sanitize data sent to other systems.  SQL injection can be prevented with similar techniques as Coding Standard 3. Injection attacks are born of accepting queries as concatenated strings and passing them directly as a query, resulting in malicious tautologies like ‘or 1=1’ that return all results in a database. Queries should be parameterized rather than accepted as string literals. Sanitization may be necessary to remove malicious additions like a tautology.  As a bonus, adhering to the principle of least privilege can help mitigate the damage of an SQL injection attack. If a user has as little access as possible to a database, then the returned results will be fewer than if they had total access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors (see Chapter 8) |
| CodeSonar | 9.0p0 | JAVA.IO.INJ.SQL | SQL injection |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | “MEM50-CPP. Do not access freed memory.” (SEI CERT Coding Standard)  Accessing freed memory leads to undefined behavior, which can irreparably damage a system in hard-to-track ways, such as by changing values of variables. |

| **Noncompliant Code** |
| --- |
| In the code below, a pointer is created, deleted, then (attempted to be) accessed in output. When agePtr is deleted, it becomes a dangling pointer, and attempting to access its freed memory could make the system crash. |
| #include <iostream>  using namespace std;  int main() {  int\* agePtr = new int(30);  delete agePtr;  cout << \*agePtr;  return 0;  } |

| **Compliant Code** |
| --- |
| The code below safely addresses the dangling pointer by setting it to NULL after deletion. This prevents any accidental (or malicious) access to freed memory. |
| #include <iostream>  using namespace std;  int main() {  int\* agePtr = new int(30);  delete agePtr;  agePtr = nullptr;  return 0;  } |

| **Principles(s):**  3 – Architect and design for security policies.  10 – Adopt a secure coding standard.  Avoiding security issues when using pointers comes from having foundational knowledge of how pointers work and their memory leak risks. Knowledge of dangling pointers, as shown in the example here, is the key to writing code that prevents them. Although this might seem too general, this sort of issue can appear as a hard-to-spot bug if the foundational knowledge is lacking. Designing and coding with security in mind makes sure pointers are treated with caution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| CodeSonar | 9.1p0 | ALLOC.UAF | Use after free |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | “DCL03-C. Use a static assertion to test the value of a constant expression.” (SEI CERT Coding Standard)  When testing code, use assertions to confirm that blocks and branches do what they’re supposed to do.  As a program is expanded to house more blocks and branches, unit tests with assertions are crucial for making sure everything functions as intended after changes are made. In other words, assertions assist with catching mistakes at both the unit and integrated testing levels. |

| **Noncompliant Code** |
| --- |
| The example below is relatively inoffensive, but it wasn’t created with test-driven development in mind. No unit test exists to assert the validity of the arithmetic. In the event of unchecked under- or overflow, such an assertion could sound the alarm. |
| #include <iostream>  using namespace std;  int main() {  int age;  int birthYear = 1990;  age = 2025 - birthYear;  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant code uses two assertions with fixed int values to test that 1) the code fails in that age does not equal 30, and 2) the code succeeds in that age is a positive value. These examples show how assertions are used to check the branching possibilities a code block may result in. |
| #include <cassert>  #include <iostream>  using namespace std;  #define ASSERT(condition, message) \  if (!(condition)) { \  cerr << "Assertion failed: " << message << endl; \  exit(EXIT\_FAILURE); \  }  int main() {  int age;  int birthYear = 1990;  age = 2025 - birthYear;  ASSERT(age == 30, "Age should be 30");  ASSERT(age > 0, "Age must be positive");  return 0;  } |

| **Principles(s):**  3 – Architect and design for security policies.  9 – Use effective quality assurance techniques.  Using assertions is crucial during testing to ensure code does what it’s supposed to do. Designing for security policies ensures time and effort is allotted for this kind of practice. Asserting is also part of quality assurance in that main and edge cases can be tested and observed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | “ERR51-CPP. Handle all exceptions.” (SEI CERT Coding Standard)  Exceptions must be caught and handled using try/catch blocks to avoid system crashes or revealing default error messages. |

| **Noncompliant Code** |
| --- |
| The code below asks for and accepts user input without question. It is rigidly expecting an integer value and has no failsafe in place to react gracefully if the user enters a different data type (e.g., “ten”). |
| #include <iostream>  using namespace std;  int main() {  int number;  cout << "Please enter a number: " << endl;  cin >> number;  return 0;  } |

| **Compliant Code** |
| --- |
| This version of the code implements a try/catch block to, at the very least, detect if the user entered anything other than an integer and close the program with a custom error message. |
| #include <iostream>  using namespace std;  int main() {  int number;  try {  cout << "Please enter a number: " << endl;  cin >> number;  // if number is not int  if (number != static\_cast<int>(number)) {  throw invalid\_argument("Input is not a valid integer.");  }  }  catch (const exception& e) {  cerr << "An error occurred: " << e.what() << endl;  return 1;  }  return 0;  } } |

| **Principles(s):**  3 – Architect and design for security policies.  8 – Practice defense in depth.  10 – Adopt a secure coding standard.  Using try/catch blocks to handle exceptions is one way to keep the system in control of itself, thereby adding to defense in depth and secure architecture. Unhandled exceptions can lead to system failure, unexpected behavior, or error messages that reveal sensitive details (such as source code).  For developers debugging, writing descriptive exception handling into the code also helps to avoid error-hiding or error-swallowing. It’s best to anticipate specific exceptions rather than a “catch all” to provide as much detailed information to the system as possible. This is part of maintaining a secure coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | 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 | R2024b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer Overflow | STD-008-CPP | “STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.” (SEI CERT Coding Standard)  Prevent buffer overflow by writing code that catches and corrects it before it’s passed to the system or subsystems.  Buffer overflow can result in unexpected behavior in a system, like data being overwritten or sensitive information being exposed. Because it’s unclear exactly what will happen when buffer overflow occurs, it’s important to prevent it to protect system integrity. |

| **Noncompliant Code** |
| --- |
| The code below does not protect against buffer overflow, meaning the array can be overloaded and rewritten to contain any value. Considering this variable is an account name, rewriting data here is especially dangerous given that it could deny a typical user access or grant a malicious user access. NOTE: This code is from a Module 2 assignment. |
| #include <iomanip>  #include <iostream>  int main()  {  std::cout << "Buffer Overflow Example" << std::endl;  const std::string account\_number = "CharlieBrown42";  char user\_input[20];  std::cout << "Enter a value: ";  std::cin >> user\_input;  std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  } |

| **Compliant Code** |
| --- |
| The code below works to detect buffer overflow by encasing the input attempt in a try/catch block, comparing the input against the maximum array size, and looping until valid input is given. NOTE: This code is from a Module 2 assignment. |
| #include <iomanip>  #include <iostream>  int main()  {  std::cout << "Buffer Overflow Example" << std::endl;  const std::string account\_number = "CharlieBrown42";  char user\_input[20];  bool valid\_input = false;  while (!valid\_input) {  try {  std::cout << "Enter a value: ";  std::cin.getline(user\_input, sizeof(user\_input));  // cin fails if the input received exceeds the limit declared using sizeof()  if (std::cin.fail()) {  std::cin.clear(); // clear the fail state  std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // discard the input buffer  std::cout << "\nYou entered: " << user\_input << std::endl;  throw std::runtime\_error("The entered value is too long. Please input a value with less than 20 characters.\n");  }  else {  valid\_input = true; // input is valid, exit the loop  }  }  catch (const std::exception& e) {  std::cerr << "ERROR: " << e.what() << std::endl;  }  }  std::cout << "\nYou entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  } |

| **Principles(s):**  2 – Heed compiler warnings.  9 – Use effective quality assurance techniques.  Overflow issues can often be avoided by heeding compiler or tool warnings, especially when using arrays in CPP and attempting to access out of bounds.  Quality assurance should be thorough enough to catch this kind of logical error if all else fails. Dynamic tests that attempt arithmetic or index access with increasing values could reveal overflow if the data returns an incorrect value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow  Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| CodeSonar | 9.1p0 | 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 |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Encapsulation | STD-009-CPP | “OBJ01-J. Limit accessibility of fields.” (SEI CERT Coding Standard)  Per object-oriented programming (OOP) standards, encapsulate class variables by making them “private” to avoid their accidental overwriting or misuse. |

| **Noncompliant Code** |
| --- |
| In this non-example, the variables of DemoCode are made publicly accessible to the rest of the system that has access to the class. Data integrity can be compromised because the variables themselves are being manipulated, possibly negatively affecting the innerworkings of the class. |
| class DemoCode {  public:  int x;  int y;  }; |

| **Compliant Code** |
| --- |
| The variable within DemoCode is made private and is only accessible through its public getX() function, as is standard for proper encapsulation. |
| class DemoCode {  private:  int x;  public:  int getX() const {  return x;  }  }; |

| **Principles(s):**  5 – Default deny.  6 – Adhere to the principle of least privilege.  Encapsulating variables is all about keeping them safe from accidental overwrite or unintended use elsewhere in a program. This restricts access privilege considerably, which is why get() functions are used to return a copy rather than the variable itself. Encapsulation can be thought of as a form of defaulting to denial to protect variables. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klockwork | 2025.2 | SV.EXPOSE.MUTABLEFIELD  SV.EXPOSE.FIELD  SV.EXPOSE.IFIELD  SV.STRUTS.PRIVATE  SV.STRUTS.STATIC |  |
| SonarQube | 9.9 | S2386 | Mutable fields should not be "public static"  Implemented for public static array, Collection, Date, and awt.Point members. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointers | STD-010-CPP | “EXP53-CPP. Do not read uninitialized memory.” (SEI CERT Coding Standard)  Eliminate wild pointers in code by initializing pointers to NULL and/or immediately pointing them to a valid variable. |

| **Noncompliant Code** |
| --- |
| The pointer in this code is initialized without a value. Outputting its value results in undefined behavior. |
| #include <iostream>  using namespace std;  int main() {  int\* wildPtr;  cout << "This here's a wild pointer! " << \*wildPtr << endl;  } |

| **Compliant Code** |
| --- |
| This pointer has been “tamed” in comparison because it is initialized to NULL then immediately set to point to a variable. When outputting its value, it returns 30, as expected. |
| #include <iostream>  using namespace std;  int main() {  int\* tamePtr = nullptr;  int value = 30;  tamePtr = &value;  cout << "This pointer is valid: " << \*tamePtr << endl;  } |

| **Principles(s):**  4 – Keep it simple.  Initializing pointers to NULL makes sense. At the inception of a pointer, it should point to nothing, *not* something unknown. This is similar to initializing an integer variable to 0, if beginning at 0 makes sense in context. This practice keeps things simple and follows a logical flow, meaning it makes for more human readable code, as well. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Uninitialized-read | Partially checked |
| CodeSonar | 9.1p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |

### Defense-in-Depth Illustration

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



### Automation



*Automation will be used for enforcement of and compliance with the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure and will modify these elements to automate enforcement of the standards in this policy. The transition from DevOps to DevSecOps is described below in reference to the above diagram.*

The transition from DevOps to DevSecOps does not sacrifice the continuous integration/continuous delivery (CI/CD) ideology but rather incorporates security as an integral part of every step. This addition creates a culture of secure coding and test-driven development that holds standards in mind and anticipates and responds to vulnerabilities consciously. Green Pace will provide training and education for its developers that covers common security threats, the coding principles and standards found in this document, and test-driven development.

Automation tools like the ones listed for every coding standard will be used as part of static code analysis. Static analysis plays a crucial role in catching errors or bugs early and often in the SDLC. The sooner they’re caught, the less problematic these issues are. For example, scrambling to find and resolve a bug after a system has launched requires more time, effort, and testing to resolve given how integrated the bug has become into the completed software. Catching a hazard (like a dangling pointer) with automated tools helps development continue onward, fixing issues along the way, thus integrating testing into the SDLC rather than at the end. In all, these tools are the attractive alternative to damage control in that they offer preventative care.

These tools will be used by developers during the Build, Test, Deploy, Transition, Respond, and Maintain stages. The specific tools for each standard will be planned and set up for use in the Plan and Design stages. Other tools, like LDAP in conjunction with Microsoft’s Active Directory (AD), are useful for automating authentication and accounting processes. LDAP can quickly authenticate a user amongst a large AD, and built-in AD tools can help track suspicious activity.

### Summary of Risk Assessments

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

### 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 the protection of stored data that is not in transit. Stored records on a USB are data at rest.  **Policy:**  The encryption algorithm AES-256 will be used to encrypt data at rest, such as database records not in use. This means that, even if the physical storage for the data is stolen or otherwise compromised, the data itself would be obscured.  AES-256 is ideal for protecting large amounts of data and it’s currently infeasible to crack, making it the perfect choice. |
| Encryption in flight | Encryption in flight is the protection of data in transit on a network. Accessing data through the Internet is one way to encounter data in flight.  **Policy:**  Web and email communications will be protected by HTTPS, which effectively combines the utility of HTTP and the security of TLS to authorize users via certificates. This also works to encrypt sensitive data (such as a credit card number) that’s sent from a user (client) to a storefront (server). |
| Encryption in use | Encryption in use is the protection of data actively being processed. This is data actively used by short-term memory (RAM) to achieve an outcome for the user. Encryption in use is newer than the former two types as it’s historically harder to achieve.  **Policy:**  Homomorphic encryption will be used to obscure plain text as it’s in use by a user and the software. This works by performing operations with the data before it’s decrypted, and it produces accurate results without risky exposure, especially for cloud-serviced software. |

| 1. **Triple-A Framework** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process that verifies a user is who they say they are, so to speak. It often involves multiple steps to protect a user’s account and other sensitive information from intruders.  **Policy:**  All users must keep track of their login information, which consists of a username (likely email address) and passwords. Passwords must be at least 16 characters and contain at least one number and one special character. Users will be reminded *not* to include easily identifiable details in their passwords (such as a pet’s name). Additionally, multi-factor authentication (MFA) will be used so that users must confirm via phone, email, or app that their login attempt is genuine. |
| Authorization | Authorization is the level of access a user has for completing tasks or accessing components of software. For example, users of a program that allows access to multiple databases would have varying levels of access depending on their allocations at account creation. An admin-level user might have access to all databases while a database-specific user would only have access to said database.  **Policy:**  All users of a system will be restricted using role-based access control that upholds the principle of least privilege. When new users are added, their roles will be designated at creation. A developer will have access to source code, a payroll staff member will have access to work logs, and a customer will have access to the storefront, for example. |
| Accounting | Accounting, in cybersecurity, refers to the audit of a system to scrutinize its resource use and identify suspicious behavior before it goes further undetected. Anything unaccounted for (such as abnormal CPU use or strange files) is useful for investigation efforts, as it can be compared to normal, day-to-day system usage as a baseline.  **Policy:**  Queries and changes made to a database will have identifying information linking them to specific users. The same is true for file access. This creates a traffic log in the system and makes it easier to pinpoint where a breach might be coming from, such as a compromised user’s account. |

## 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 with 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 always comply with this policy.

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 | SNHU |
| 1.1 | 07/18/2025 | Ten Core Security Principles, Coding Standards (partial) | Noelle Bishop | Alan Spencer |
| 1.2 | 08/05/2025 – 8/06/2025 | Coding Standards (full), Risk Assessment, Automation, Encryption Types, Triple A | Noelle Bishop | *Pending* |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |
| Hypertext Transfer Protocol Secure | HTTPS |
| Hypertext Transfer Protocol | HTTP |
| Transport Layer Security | TLS |
| Random Access Memory | RAM |
| Lightweight Directory Access Protocol | LDAP |