**Green Pace Developer: Chad Salaets**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.h12pv73slrb)

[Purpose 2](#_heading=h.n1ca2vpd7xvx)

[Scope 2](#_heading=h.qrftxc5zh8xr)

[Module Three Milestone 2](#_heading=h.4y2yvv6js955)

[Ten Core Security Principles 2](#_heading=h.7ilkcizdqoi)

[C/C++ Ten Coding Standards 3](#_heading=h.ub3xoh738zqt)

[Coding Standard 1 4](#_heading=h.9p7e9i5ao88t)

[Coding Standard 2 5](#_heading=h.ana9tosqak1s)

[Coding Standard 3 6](#_heading=h.grbbe0aak3pq)

[Coding Standard 4 7](#_heading=h.h2089wb6xdfb)

[Coding Standard 5 8](#_heading=h.rvagwh1x1eu0)

[Coding Standard 6 9](#_heading=h.417j2y4d4831)

[Coding Standard 7 10](#_heading=h.lpkzc4hst6w0)

[Coding Standard 8 11](#_heading=h.43icj9tm0q70)

[Coding Standard 9 13](#_heading=h.od7iokv0g94z)

[Coding Standard 10 14](#_heading=h.wfmdvoncuz0r)

[Defense-in-Depth Illustration 15](#_heading=h.in5a1dfh2mdy)

[Project One 15](#_heading=h.j3mjqg1gmcjk)

[1.](#_heading=h.8hldwur8d9dj) Revise the C/C++ Standards 15

[2.](#_heading=h.2jbauiltzmu3) Risk Assessment 15

[3.](#_heading=h.emjzdjnwbor4) Automated Detection 15

[4.](#_heading=h.y1y763x4npmz) Automation 15

[5.](#_heading=h.655po6xu4nen) Summary of Risk Assessments 16

[6.](#_heading=h.vl02msj07xtw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.t7y2c6xlsoip) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.fqp9if5vk0ra)

[Enforcement 18](#_heading=h.cvdd6xh98iwn)

[Exceptions Process 18](#_heading=h.gj3kfj7zq6vq)

[Distribution 19](#_heading=h.2dck2ohbzfez)

[Policy Change Control 19](#_heading=h.644uks94pyuq)

[Policy Version History 19](#_heading=h.sencs08bwg4c)

[Appendix A Lookups 19](#_heading=h.bqp2urqqrvyi)

[Approved C/C++ Language Acronyms 19](#_heading=h.481euao22qww)

## 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 who create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input is the practice of checking all external inputs to ensure they conform to the expected formats, lengths, and data types to prevent attacks like buffer overflows, SQL injection, and command injection, which can lead to unintended program behavior. Failure to comply can lead to malicious input being passed to cause denial-of-service, data breaches, or other system compromises. |
| 1. Heed Compiler Warnings | Compiler warnings issued during build time will flag unsafe, deprecated functions, and type mismatches to identify portions of the code that are unsafe. Resolving these errors mitigates bugs that can be subtle, becoming fatal as the program is constructed. Failure to comply can lead to runtime errors, poor performance, and increased technical debt. |
| 1. Architect and Design for Security Policies | Robust and secure software design and architecture by enforcing secure best practices like least privilege, separation of concerns, and memory integrity. Security is most effective when it's integrated into the design. Failure to comply leads to snowballing technical debt, failed security requirements, and overall poor code – a disaster waiting to happen. |
| 1. Keep It Simple | An architectural style focused on simplicity and minimalism for straightforward design. Simple code is easy to understand, quick to audit, and easy to maintain. Simple code is less likely to face bugs and vulnerabilities caused by incredibly complex code. Failure to comply leads to buggy code riddled with vulnerabilities. |
| 1. Default Deny | The default deny is a policy that defaults to blocking all actions or access unless explicitly allowed to prevent unauthorized access. Denying extraneous access reduces the attack vectors and guarantees that features and users don’t inherit faulty permissions. Failure to uphold this leads to insecure codes that risk data breaches and privilege exploitation. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege ensures each process operates with the lowest privileges required to perform its function. Least privilege limits the damage area if a component is exploited, limiting the malicious actor’s abilities. Failure to adhere can lead to compromised functions, causing severe damage and leading to data breaches. |
| 1. Sanitize Data Sent to Other Systems | The practice of encoding sensitive data before transmission to outside systems ensures the data is not altered into malicious inputs. Sanitized data lessens the risk of injections, maintains data integrity, and protects the systems from malformed data. Failure to properly sanitize data leads to greater risks of a malicious actor compromising a system through injections or corruption. |
| 1. Practice Defense in Depth | Defense in depth(DiD) is a security strategy with multiple separate layers of security at different levels. DiD adds redundant security verifications, so attackers must employ laborious and complex attacks to succeed. A single defense of failure leads to a weak defense where one vulnerability can compromise the entire system. |
| 1. Use Effective Quality Assurance Techniques | Systematic testing, throughout the development process, like code reviews and static analysis, ensures the output is secure with the expected outputs. Techniques like these identify bugs and vulnerabilities early in the development cycle, meaning simple patches can be applied. Lacking QA techniques leads to flawed code that can be insecure, contain functional errors, or even data loss.. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard like the SEI CERT C++ Standard provides guidelines for consistent and safe code. The standards provide a baseline for quality and security to reduce the likelihood of bugs and exploitable design patterns. Neglect of a coding standard will cause inconsistent patterns with a high risk of security flaws or modules with varying security quality. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank, so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Safe Conversions of Type  Implicit type conversions in C++ can lead to undefined behavior. The SEI CERT C++ standard advises against such conversions unless correctly scoped, instead favoring explicit casts to signal developer intent, like with INT02-C from the C coding standard. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the two variables: integer intNum and double doubleNum are converted implicit conversion, which can lead to undefined behavior. |
| int intNum;  double doubleNum = 13.37;  a = doubleNum; // Data Type unsafely converted. |

| **Compliant Code** |
| --- |
| In this compliant code example, the solution is to use explicit conversions using the static\_cast<type> resource. |
| int intNum;  double doubleNum = 13.37;  Count = static\_cast<int>(std::round(doubleNum)); // Properly converts. |

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

| **Principles(s):** [INT02-C. Understand integer conversion rules](https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules) states that all data conversions must be done explicitly and safely instead of relying on implicit conversions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | Latest (2.18.0) | [INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data](https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data) | CppCheck will flag implicit type conversions that lead to undefined behavior |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Safe Constrained Data Values  Failure to check or constrain data values leads to severe vulnerabilities like overflows, underflows, or division by zero. Validating value bounds can prevent logic errors, memory corruption, and minimize security exploits. Standards like the INT30-C, INT32-C, and INT33-C uphold these safe patterns. |

| **Noncompliant Code** |
| --- |
| In this Noncompliant example, the array index is computed without validating it, making it open to over- or underflows. |
| int n = getUserInput();  int arr[10];  arr[n] = value; // no bounds check; out-of-range risk |

| **Compliant Code** |
| --- |
| In this compliant code, the inputs are verified before using them as an index of an array. |
| int n = getUserInput();  int arr[10];  if (n >= 0 && n < static\_cast<int>(std::size(arr))) {  arr[n] = value;  } else {  // Perform some command or flag an error; dev’s choice.  } |

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

| **Principles(s):** [CTR50-CPP. Guarantee that container indices and iterators are within the valid range](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range) states that all references within the bounds of any array must be checked to ensure they’re within range. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | Latest (2.18.0) | Bounds Checking: Array index out of bounds | Detects index out-of-range conditions and unsafe indexing behaviors |
| Clang-Tidy | Latest Stable (3.8) | Checks related to CERT Secure Coding Guidelines. | Performs deeper analysis to catch array indexing errors and conditional flows. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Safe String Handling  C+ string handling can trigger buffer overflows or underflows as well as null-pointer dereferencing. The space allocated to the strings must be within limits according to STR31-C and STR32-C, where strings need sufficient space for character data, including the null terminator. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the code copies data into a fixed-size buffer without regard for whether the length is within bounds. |
| char buf[32];  strcpy(buf, src.c\_str()); // unsafe |

| **Compliant Code** |
| --- |
| In this compliant example, the code uses safe string handling with std::string and bounds checking to prevent overallocating memory. |
| std::string src = getInput();  if (src.size() < 32) {  char buf[32];  std::strcpy(buf, src.c\_str()); // safe: size validated  } else {  // Code to truncate or flag error.  } |

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

| **Principles(s):** Per CERT rules like STR31‑C and STR32‑C (which emphasize safe storage and copying of strings) to ensure the memory is managed properly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| FlawFinder | Latest (2.0.19) | Checks for strcpy functions, not buffer overflows. | Scans for risky C/C++ functions like strcpy. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection via Parameterized Queries  A SQL injection attack is when untrusted input is inserted into SQL commands, letting attackers manipulate queries. OWASP and SEI Cert C++ recommend using parameterized queries and input validation to neutralize any attempts. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, a SQL query is assembled with unchecked user input. Any usage of an escape character could add an always-true condition, modifying the query. |
| std::string name = getUserInput();  std::string query = "SELECT \* FROM users WHERE name = '" + name + "';";  db.execute(query); |

| **Compliant Code** |
| --- |
| In this compliant code, parameterized queries are used to separate data from code, preventing injection. |
| std::string query = getUserInput();  auto stmt = db.prepare("SELECT \* FROM users WHERE name = ?;");  stmt.bind(1, name);  stmt.execute(); |

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

| **Principles(s):** [FIO30-C. Exclude user input from format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings) refers to excluding user input from format strings to avoid undefined behavior or malicious code execution, SQL injections are included. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | p18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Semgrep | Latest | Capable of checking for injections of many types. | Lightweight SAST tool that allows writing custom rules to detect SQL statements. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Prevent Use‑After‑Free and Buffer Overflows  Memory-related vulnerabilities are commonly exploited. It is important to uphold memory safety like Mem30-C, which states do not access freed memory, and STR31-CPP, where sufficient allocation is required for strings and buffers. Enforcing allocation safety will avert severe security flaws. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, a dynamically allocated buffer is deleted, but use is continued. |
| char\* buf = new char[50];  delete[] buf;  // Buffer used  std::strcpy(buf, "new data"); // Used after freed. |

| **Compliant Code** |
| --- |
| In this compliant code, the std::vector is used to manage memory, and should prevent manual deletion issues. |
| std::vector<char> buf(50);  // Buffer used  std::strcpy(buf.data(), "new data"); // safe, vector owns memory |

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

| **Principles(s):** [MEM35-C. Allocate sufficient memory for an object](https://wiki.sei.cmu.edu/confluence/display/c/MEM35-C.+Allocate+sufficient+memory+for+an+object) is one of the many principles for memory safety. This requires the memory to be kept safe and correctly allocated. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | Latest | [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) checks for BAD\_ALLOC\_STRLEN. | Coverity is an open-source static analyzer with memory management analysis. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions document and detect internal programming errors during development. Assertions are not to be used as error handling, but rather to verify the code’s logic is correct during development. SEI Cert C++ rule MSC11‑C recommends using assert to verify preconditions that are impossible in the current code. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, assert()validates user input, which may cause abrupt termination. |
| int index = getInput();  assert(index >= 0 && index < size); // Assertion kills the code.  array[index] = value; |

| **Compliant Code** |
| --- |
| In this compliant example, the assert is used to check for a impossible conditions. |
| int index = getInput();  if (index < 0 || index >= size) {  handleError("Index out of range");  } else {  array[index] = value;  }  assert(size > 0); // Must be true if the program ran properly. |

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

| **Principles(s):** [MSC11-C. Incorporate diagnostic tests using assertions](https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions) determines when and how to use assert() statements within debugging code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | Latest | Checks for ASSERT\_SIDE\_EFFECT which can detect MSC11-C violations. | Detects misuse of assert when accompanied by operations that have side effects |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exception-Safe and Defined Error Handling  Failure to implement proper exception handling in C++ can cause resource leaks, crashed programs, denial of service, or undefined behavior. SEI CERT C++ provides multiple ERR rules addressing exception safety like ERR56‑CPP (guarantee exception safety), ERR57‑CPP (avoid resource leaks), and ERR59‑CPP (avoid throwing across execution boundaries). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, a catch statement is neglected, so an application crash is inevitable. |
| int x = 10, y = 0;  int result = x / y; // This will cause the program to terminate with a division by zero error  std::cout << "Result: " << result << std::endl;  return 0; |

| **Compliant Code** |
| --- |
| In this compliant code, division by zero is tried, but the catch statement handles it, so the runtime continues. |
| try {  int x = 10, y = 0;  int result = x / y; // divide by zero can't work.  std::cout << "Result: " << result << std::endl;  }  catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** [ERR00-C. Adopting and implementing a consistent and comprehensive error-handling policy](https://wiki.sei.cmu.edu/confluence/display/c/ERR00-C.+Adopt+and+implement+a+consistent+and+comprehensive+error-handling+policy) is the general standard for handling errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Low | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| None | N/a | Ensure all risky statements use try-catch statements. | No Automation Tool Found; Training and Guidelines will have to be made to compensate. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Environmental Pointers | STD-008-CPP | Many functions return pointers to internal buffers, which can be overwritten by subsequent calls. According to ENV34‑C, storing and reusing these pointers leads to corrupted data. To avoid undefined behavior and ensure reentrancy, developers should immediately copy the returned data if future use is needed |

| **Noncompliant Code** |
| --- |
| In this non-compliant code, pointers from the two environment variables are compared directly. Issues can arise since the getenv reuses the buffer. |
| char \*tmp = getenv("TMP");  char \*temp = getenv("TEMP");  if (strcmp(tmp, temp) == 0) { // Faulty comparison  // Code  } |

| **Compliant Code** |
| --- |
| In this compliant example, strings are copied into their own variable after being returned by getenv(), avoiding reuse of the internal buffer. |
| const char \*tmp1 = getenv("TMP");  const char \*tmp2 = getenv("TEMP");  // Saved  if (tmp1 && tmp2) { // If valid, then save them to a string.  std::string s1(tmp1);  std::string s2(tmp2);  if (s1 == s2) {  // safe comparison using strings.  }  } |

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

| **Principles(s):** [ENV34-C: Do not store pointers returned by certain functions](https://wiki.sei.cmu.edu/confluence/display/c/ENV34-C.+Do+not+store+pointers+returned+by+certain+functions) from the C coding standard that the pointers should not be stored for use; instead, the pointers' output should be stored. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Low | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Checks for “cert-err34-c” | Can be configured to flag direct reuse of pointers from environment functions without copying. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Loop Counters | STD-009-CPP | Avoid Floating‑Point Loop Counters  Floating-point types struggle to represent many decimal values or maintain precision when handling very large or very small numbers. Floating point loop counters can have unpredictable iteration counts and incorrect logic flow, as outlined by SEI CERT C FLP30‑C. Instead, integer indices with calculable floating-point values should be used. |

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, a loop uses a floating point which could run more or less than the required values. |
| void func() {  for (float x = 0.1f; x <= 1.0f; x += 0.1f) { // Can run 10 times, 11, 9, 8, or 12.  // Some Other Code  }  } |

| **Compliant Code** |
| --- |
| In this compliant code, the integer counter calculates the floating-point values inside the loop a consistent number of times. |
| void func() {  for (std::size\_t count = 1; count <= 10; ++count) { // Uses a size\_t  float x = count / 10.0f;  // loop runs exactly 10 times  }  } |

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

| **Principles(s):** [FLP30-C. Do not use floating-point variables as loop counters](https://wiki.sei.cmu.edu/confluence/display/c/FLP30-C.+Do+not+use+floating-point+variables+as+loop+counters) as the SEI C coding standard guide requires developers to refrain from using the floating data type as a loop counter due to undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Very Low – Use int, not float. | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 3.9 | Flags “cert-flp30-c” when a loop and a float are used. | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Invalid Bitwise Shift | STD-010-CPP | Prevent Invalid Bitwise Shift Operations  Shifting a value by a negative count or by a value equal to or exceeding the bit-width of its type leads to undefined behavior. Within C++ standards, INT34-C recommends shifting left into an unsigned type. This can cause crashes due to logic errors. Validating shift operands prevents erroneous behavior and enhances code safety. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, a left shift is done without checking whether the count is within the valid range. |
| unsigned int val = 1u;  int count = getOffset();  unsigned int result = val << count; // Undefined if count < 0 or ≥ width |

| **Compliant Code** |
| --- |
| In this compliant code, the shift amount is validated to be non-negative and less than the number of bits within the operand. |
| unsigned int val = 1u;  int count = getOffset();  if (count < 0 || static\_cast<unsigned>(count) >= std::numeric\_limits<unsigned int>::digits) {  handleError("Invalid shift count");  } else {  unsigned int result = val << static\_cast<unsigned>(count);  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.3 | INT34-C violations will flag “shiftNegative: and “shiftTooManyBits” when encountered. | Flags shifts where the right operand is too large for the left operand’s width |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate the enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

The diagram provides a comprehensive roadmap for the deployment, enforcement, and maintenance of the policies; however, it could benefit from automation. The assessment and plan can be automated using tools like [IriusRisk](https://www.iriusrisk.com/threat-modeling-platform) that document the security design flaws early in the code’s development. The build phase is where code is committed to the repository, and some of the biggest threats can be through dependencies. [OWASP Dependency Check Tool](https://owasp.org/www-project-dependency-check/) can scan the list of vulnerabilities and detect CVEs (Common Vulnerabilities and Exposures) in the software’s dependencies, while [SonarQube](https://www.sonarsource.com/products/sonarqube/) can scan the code to identify issues with compliance. The testing stage needs vulnerability scanners like [OWASP's Zap](https://www.zaproxy.org/), which is a free alternative to the paid [IBM AppScan](https://www.ibm.com/docs/en/dsm?topic=guide-appscan-enterprise-scanner-overview). The transition and health check stage is where the project becomes a reality. Runtime Verification tools like [Osquery](https://osquery.io/) or [Tripwire](https://www.tripwire.com/) verify that the system runs as expected. The Monitor and detect stage (commonly called observe) is where the application is in the real production environment. The most vulnerable systems are often sensitive public endpoints like user account endpoints and database access. [Fidelis' Halo](https://fidelissecurity.com/fidelis-halo-cloud-native-application-protection-platform-cnapp/) and [Imperva](https://www.imperva.com/learn/application-security/runtime-security/) offer comprehensive protection, including support, verbose logs, flag anomalies, and perform application monitoring. The stages of response and maintenance, and stabilization can be automated by using Security orchestration automation and response (SOAR). Most popular SOAR solutions are

[ManageEngine](https://www.manageengine.com/log-management/sem/soar-security-orchestration-automation-and-response.html?utm_source=BraveSearch&utm_campaign=SOARLog360-US&utm_medium=cpc) and [Splunk SOAR.](https://www.splunk.com/en_us/blog/learn/soar-security-orchestration-automation-response.html)

### Summary of Risk Assessments

Consolidate all risk assessments into one table, including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Probable | Medium | P12 | L1 |
| STD-002-CPP | High | Likely | Low | P9 | L2 |
| STD-003-CPP | High | Likely | Medium | P9 | L2 |
| STD-004-CPP | High | Likely | Medium | P9 | L2 |
| STD-005-CPP | High | Probable | Medium | P6 | L2 |
| STD-006-CPP | Low | Unlikely | Low | P1 | L3 |
| STD-007-CPP | Medium | Probable | Low | P4 | L3 |
| STD-008-CPP | Low | Probable | Low | P4 | L3 |
| STD-009-CPP | Low | Probable | Very Low | P6 | L2 |
| STD-010-CPP | Low | Unlikely | Low | P2 | L3 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in-flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how, and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the conversion of disks, backups, databases, and cloud storage into a ciphertext, making it unreadable without the key. This encryption is common in secure databases, file servers, virtual disks, and network drives. The policy protects data in case of device theft, physical loss, or unauthorized access. Our force enforces encryption for all sensitive data at rest with symmetric algorithms like AES-256 and central key management (HSMs or cloud KMS). |
| Encryption in flight | Encryption in flight is data encryption during transmissions across networks and between clients. Often used autonomously in web APIs through HTTPS, Email services like SMTPS, and file transfers like SFTP. Policy is essential whenever information travels on untrusted paths like the public internet. The encryption prevents eavesdropping and tampering with the data. Our policy should require TLS for all external and internal data flows. |
| Encryption in use | Encryption in use protects the data that’s being processed in the device’s memory. The data must be decrypted to be used, or it could use [homomorphic encryption](https://www.ibm.com/think/topics/homomorphic-encryption). The process can use hardware-based solutions like Intel SGX or AMD SEV. Our policy should enforce encryption technologies where feasible, like when handling sensitive analytics or high-value IP in cloud infrastructure. |

| 1. **Triple-A Framework\*** | **Explain what it is and how, and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user or system through passwords, multi-factor authentication (MFA), biometrics, or digital certificates. Authentication requires credentials or verification for the user, ensuring the access is from the intended personnel, preventing impersonation or data breaches. Auuh |
| Authorization | Authorization determines what a user’s permissions are based on the role or policies. This policy would apply to User access to define and enforce the roles within the company, with each role having the least amount of user privilege to accomplish their job. This Policy prevents privilege abuse, forcing users to act within their scope of intended permissions. |
| Accounting | Accounting records logs from the user, which could include actions, resource usage, session duration, accessed files, and commands executed, leaving behind a record of what the user did. This policy can apply to files accessed, commands allowed to run, admin permissions, user logins, database changes, and much more. This policy is important for the accountability of each user’s actions. It leads to an easier forensic analysis, forces compliance with policies, and detects misuse of systems. |

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

| **Standard** | **Principle(s)** | **Justification** |
| --- | --- | --- |
| **STD-001-CPP** Safe Conversions of Type | 1 – Validate Input Data  3 – Architect and Design for Security Policies  10 – Adopt a Secure Coding Standard | Validating and casting input ensures type safety, preventing unintended behavior from unsafe conversions. Designing for security requires selecting explicit type handling as part of the architecture, while secure coding standards (SEI CERT INT02-C/INT31-C) mandate these safeguards. |
| **STD-002-CPP** Safe Constrained Data Values | 1 – Validate Input Data  5 – Default Deny  10 – Adopt a Secure Coding Standard | Bounds checking ensures invalid inputs are rejected by default, supporting a deny-by-default approach. Adhering to CERT INT30-C and INT33-C demands consistent validation patterns across all code. |
| **STD-003-CPP** Safe String Handling | 1 – Validate Input Data  4 – Keep It Simple  8 – Practice Defense in Depth  10 – Adopt a Secure Coding Standard | Validating string lengths prevents overflow. Using std::string for simple secure handling. Defense-in-depth is achieved by combining safe APIs with validation. CERT STR31-C/STR32-C upholds these rules. |
| **STD-004-CPP** Prevent SQL Injection | 1 – Validate Input Data  5 – Default Deny  8 – Practice Defense in Depth  10 – Adopt a Secure Coding Standard | Input validation and parameterized queries deny unsafe SQL execution by default. Multiple layers, Validation, query parameterization—support defense in depth. CERT and OWASP guidelines define these as required secure practices. |
| **STD-005-CPP** Memory Protection | 3 – Architect and Design for Security Policies  6 – Principle of Least Privilege  8 – Practice Defense in Depth  10 – Adopt a Secure Coding Standard | Secure architecture favors safe memory management constructs to reduce privilege misuse. Defense-in-depth combines coding discipline with runtime checks; CERT MEM30-C/MEM50-CPP enforce these principles. |
| **STD-006-CPP** Assertions | 3 – Architect and Design for Security Policies  4 – Keep It Simple  9 – Use Effective Quality Assurance Techniques  10 – Adopt a Secure Coding Standard | Proper assertion use is part of designing predictable, testable code. Keeping assertion logic simple avoids misuse. Assertions enhance QA when used only for invariant checking, as mandated per CERT MSC11-C. |
| **STD-007-CPP** Exception-Safe Handling | 3 – Architect and Design for Security Policies  5 – Default Deny  8 – Practice Defense in Depth  10 – Adopt a Secure Coding Standard | Designing for safe exception handling ensures systems fail securely. Default-deny behavior prevents continuing in unsafe states. Multiple safety nets (RAII, structured handling) provide defense in depth; CERT ERR56-CPP/ERR57-CPP enforce this. |
| **STD-008-CPP** Copy Pointers from Internal Buffers | 1 – Validate Input Data  8 – Practice Defense in Depth  10 – Adopt a Secure Coding Standard | Verifying environment variable data and copying it into safe containers prevents corruption. Defense in depth comes from both validation and isolating internal buffers. CERT ENV34-C formalizes this rule. |
| **STD-009-CPP** Avoid Floating-Point Loop Counters | 3 – Architect and Design for Security Policies  4 – Keep It Simple  10 – Adopt a Secure Coding Standard | Choosing integer counters in loop control is part of a secure, predictable architecture. Simpler numeric control flow reduces subtle bugs; CERT FLP30-C documents this best practice. |
| **STD-010-CPP** Prevent Invalid Bitwise Shift | 1 – Validate Input Data  3 – Architect and Design for Security Policies  4 – Keep It Simple  10 – Adopt a Secure Coding Standard | Validating shift values before use prevents UB and supports a secure architecture. Simple, clear checks make the code easier to verify; CERT INT34-C codifies the requirement. |

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 07/21/2025 | Milestone Completed. | Allen Salaets | Bill Gates |
| 3.0 | 08/09/2025 | Completed Document. | Allen Salaets | Bill Gates |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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