**Green Pace Developer: Security Policy**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input ensures that all incoming data matches expected formats before it is processed. This approach helps prevent attacks such as SQL injection, cross-site scripting, and buffer overflow, by catching potentially harmful data at the point of entry. Ensuring input meets security standards reduces the risk of exploitation and strengthens application defenses. |
| 1. Heed Compiler Warnings | Compiler warnings can reveal hidden issues in code, such as unused variables or potentially risky operations. By addressing these warnings, developers can catch bugs early and prevent vulnerabilities from slipping into production. Attending to compiler alerts is an easy way to improve software reliability and security. |
| 1. Architect and Design for Security Policies | Designing software with security in mind means incorporating access controls, data encryption, and validation checks at the architectural level. By aligning system design with security policies from the start, developers build systems that are better equipped to prevent attacks and protect sensitive data, reducing the chance of significant security breaches. |
| 1. Keep It Simple | A simple design and implementation reduces the complexity of code, making it easier to maintain and secure. Systems with clear, straightforward code are less prone to errors and have fewer vulnerabilities, minimizing the risk of potential security flaws. |
| 1. Default Deny | Default-deny policies block access to all resources unless explicitly permitted. By denying access by default, developers reduce the chance of unauthorized users gaining entry. This restrictive approach provides a strong foundational security layer, only allowing access as needed. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege restricts users and systems to the minimum permissions needed to perform their tasks. Limiting permissions in this way reduces the risk of misuse or exploitation, as there are fewer opportunities for attackers to escalate privileges or gain unauthorized access. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before it is transmitted ensures that only safe, expected data is sent, preventing harmful code from entering other systems. This reduces the risk of vulnerabilities and keeps interactions between systems secure, maintaining the integrity of the connected environments. |
| 1. Practice Defense in Depth | Defense in depth involves using multiple layers of security, such as firewalls, monitoring, and encryption, to protect systems from attack. Even if one layer is breached, additional layers provide continued defense, ensuring comprehensive protection and reducing the likelihood of a single point of failure. |
| 1. Use Effective Quality Assurance Techniques | Employing quality assurance methods like code reviews, static analysis, and testing helps to identify and resolve security issues early. By catching vulnerabilities before software is deployed, quality assurance strengthens software security and ensures it is reliable and safe for users. |
| 1. Adopt a Secure Coding Standard | Secure coding standards provide guidelines for avoiding common security pitfalls and writing resilient code. By following these standards, developers reduce the risk of vulnerabilities, promoting a consistent, high level of security across the software and making it more resistant to attacks. |

### 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] | Obey the one-definition rule |

| **Noncompliant Code** |
| --- |
| Defining the same class differently across translation units leads to undefined behavior. |
| // a.cpp  struct S { int a; };  // b.cpp  class S { public: int a; }; |

| **Compliant Code** |
| --- |
| Use a header file to maintain consistency across translation units. |
| // S.h  struct S { int a; };  // a.cpp  #include "S.h"  // b.cpp  #include "S.h" |

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

| **Principles(s):** Single Source of Truth (SSOT)   * This principle ensures that a piece of information, such as the definition of a class, exists only in one place. Using a header file adheres to SSOT, reducing discrepancies and potential undefined behavior across translation units. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | classReassign | Identifies redefinitions of classes. |
| Clang-Tidy | 15.0 | misc-definition | Ensures consistent class definitions. |
| SonarQube | 9.9 | cpp:S1871 | Flags inconsistent class or struct usage. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not read uninitialized memory |

| **Noncompliant Code** |
| --- |
| Using an uninitialized variable causes undefined behavior. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| Initialize variables before usage. |
| #include <iostream>  void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** Fail-Safe Defaults   * This principle ensures that systems behave predictably and securely even in the face of errors. Initializing variables to a default value prevents reading from uninitialized memory, which could cause unpredictable behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | uninitvar | Detects uninitialized variables |
| Clang-Tidy | 2.11 | uninitvar | Detects uninitialized variables |
| SonarCube | 9.9 | cpp:S5534 | Identifies reads of uninitialized memory. |
| Coverity | 2024.3 | UNINIT | Highlights usage of unintialized variables |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| Constructing a std::string from a null pointer results in undefined behavior. |
| #include <cstdlib>  #include <string>  void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| Check for null before constructing a std::string. |
| #include <cstdlib>  #include <string>  void f() {  const char\* tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** Defensive Programming   * This principle involves anticipating and guarding against potential issues in the code. Checking for null pointers before constructing a std::string prevents undefined behavior and ensures robustness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | nullPointerUsage | Detects null pointer dereferences. |
| Clang-tidy | 15.0 | Modernize-avoid-nullptr | Flags potential null pointer issues. |
| SonarQube | 9.9 | cpp:S5736 | Identifies unsafe use of null pointers. |
| Coverity | 2024.3 | NULL\_RETURNS | Highlights unchecked null pointer returns. |

#### Coding Standard 4

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

| **Noncompliant Code** |
| --- |
| Without precautions, the untrusted data may maliciously alter the query. |
| uName = getRequestString("username"); uPass = getRequestString("userpassword"); sql = “SELECT \* FROM Users WHERE Name = " + uName + " AND Pass = " + uPass + ” |

| **Compliant Code** |
| --- |
| The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures. |
| PreparedStatement pStmt = PreparedStatement(); std::cin >> username; std::cin >> userpassword; sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username, userpassword}; |

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

| **Principles(s):** Validation and Sanitization   * This principle ensures that inputs are verified and cleaned to prevent injection attacks. By using parameterized queries, developers mitigate the risk of SQL injection and ensure safe database interactions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S3649 | Detects SQL injection vulnerabilities. |
| Cppcheck | 2.11 | sqlInjectionRisk | Identifies dynamic SQL queries vulnerable to injection |
| Fortify | 21.2 | SQL\_INJECTION | Flags untrusted data used in SQL queries. |
| OWASP ZAP | 2.13.0 | SQL\_InjectionScanner | Scans for SQL injection risks in code. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| Using a pointer after freeing memory. |
| #include <new>  struct S { void f(); };  void g() noexcept(false) {  S\* s = new S;  delete s;  s->f(); // Use-after-free  } |

| **Compliant Code** |
| --- |
| Deallocate memory only when no longer needed. |
| #include <new>  struct S { void f(); };  void g() noexcept(false) {  S\* s = new S;  s->f();  delete s;  } |

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

| **Principles(s):** Resource Management   * This principle ensures that resources are used and released responsibly. By avoiding access to freed memory, this standard prevents undefined behavior and maintains program stability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20.0 | memcheck | Detects use-after-free and invalid memory access. |
| Asan | 15.0 | addressSanitizer | Identifies memory access errors. |
| Cppcheck | 2.11 | useAfterFree | Flags use-after-free issues in the code. |
| Coverity | 2024.3 | USE\_AFTER\_FREE | Highlights invalid memory access patterns. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CLG] | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| Using assert() with constant expressions. |
| #include <assert.h>  struct timer { unsigned char MODE; unsigned int DATA; unsigned int COUNT; };  int func() {  assert(sizeof(timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| Use static\_assert for compile-time checks. |
| struct timer { unsigned char MODE; unsigned int DATA; unsigned int COUNT; };  static\_assert(sizeof(timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int), "Structure must not have padding"); |

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

| **Principles(s):** Compile-Time Verification   * This principle ensures that critical assumptions about the program are verified during compilation rather than at runtime, improving safety and efficiency. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 16.0 | Static-assert-check | Detects missing or incorrect static assertions. |
| Cppcheck | 2.11 | missingStaticAssert | Flags noncompliant usage of runtime assert. |
| Coverity | 2024.3 | STATIC\_ASSERT | Ensures proper use of static assertions. |
| GCC | 12.3.0 | Built-In Compiler Check | Highlights constant expressions for static\_assert. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| A call to f() registered as an exit handler may result in abrupt termination if throwing\_func() throws an exception. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() {  // Not invoked by the program except as an exit handler.  throwing\_func();  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| The f() function catches all exceptions and prevents abrupt termination. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() {  // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):** Graceful Error Handling   * This principle emphasizes robust and predictable program behavior even in the face of unexpected issues. By handling exceptions gracefully, programs ensure stability and user trust. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | static-assert-check | Detects missing or incorrect static assertions. |
| Cppcheck | 2.11 | missingStaticAssert | Flags noncompliant usage of runtime assert. |
| Coverity | 2024.3 | STATIC\_ASSERT | Ensures proper use of static assertions. |
| GCC | 12.3.0 | Built-in Compiler Warning | Highlights functions with uncaught exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | [STD-008-CPP] | Manage resources effectively and avoid leaks |

| **Noncompliant Code** |
| --- |
| Failure to deallocate resources leads to a memory leak. |
| #include <new>  struct S {  int a;  };  void f() {  S\* s = new S();  // Resource is not properly deallocated.  } |

| **Compliant Code** |
| --- |
| Resources are properly deallocated to avoid memory leaks. |
| #include <new>  struct S {  int a;  };  void f() {  S\* s = new S();  delete s;  } |

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

| **Principles(s):** Efficient Resource Management   * This principle ensures that resources such as memory, file handles, or sockets are properly acquired and released to avoid performance degradation, system crashes, or unpredictable behavior.   RAII (Resource Acquisition Is Initialization)   * Following RAII, resources should be tied to the lifecycle of objects to ensure automatic cleanup when objects go out of scope. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | memory-leak-check | Detects memory leaks in dynamic memory allocation. |
| Cppcheck | 2.11 | resourceLeak | Identifies resource leaks, including unfreed memory. |
| Valgrind | 3.21.0 | memcheck | Detects memory leaks during runtime. |
| Coverity | 2024.3 | RESOURCE\_LEAK | Identifies resource leaks in various types of resources. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | [STD-009-CPP] | Avoid data races and deadlocks |

| **Noncompliant Code** |
| --- |
| Accessing shared data without a lock may cause data races. |
| #include <thread>  int shared\_var = 0;  void increment() {  shared\_var++;  }  int main() {  std::thread t1(increment);  std::thread t2(increment);  t1.join();  t2.join();  } |

| **Compliant Code** |
| --- |
| Shared data is accessed under a lock to avoid data races. |
| #include <thread>  #include <mutex>  int shared\_var = 0;  std::mutex mtx;  void increment() {  std::lock\_guard<std::mutex> lock(mtx);  shared\_var++;  }  int main() {  std::thread t1(increment);  std::thread t2(increment);  t1.join();  t2.join();  } |

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

| **Principles(s):** Thread Safety and Synchronization   * Ensures that multiple threads can safely access and modify shared resources. Proper synchronization mechanisms, such as locks, eliminate data races, undefined behavior, and potential deadlocks.   Avoidance of Deadlocks   * Prevents situations where two or more threads are waiting indefinitely for resources held by each other. Using standardized locking practices, such as std::lock or std::lock\_guard, helps maintain thread safety and system stability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | Latest (Clang/GCC) | data-race detection | Detects data races and shared memory issues during runtime. |
| Cppcheck | 2.11 | threadSafety | Identifies thread-safety issues in code. |
| Clang-Tidy | 16.0 | concurrency checks | Analyzes and flags unsafe concurrent access patterns. |
| Helgrind | Valgrind Suite | thread-analysis | Detects data races and potential deadlocks. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Bounds Checking | [STD-010-CPP] | Perform bounds checking for arrays and pointers |

| **Noncompliant Code** |
| --- |
| Array access exceeds the valid bounds, leading to undefined behavior. |
| #include <iostream>  int main() {  int arr[3] = {1, 2, 3};  std::cout << arr[3]; // Out-of-bounds access  return 0;  } |

| **Compliant Code** |
| --- |
| Access is within bounds to ensure defined behavior. |
| #include <iostream>  int main() {  int arr[3] = {1, 2, 3};  for (int i = 0; i < 3; i++) {  std::cout << arr[i];  }  return 0;  } |

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

| **Principles(s):** Safety and Robustness   * Ensures program stability by preventing access beyond the defined bounds of an array or pointer, which could lead to undefined behavior, memory corruption, or security vulnerabilities.   Error Prevention   * By enforcing bounds checking, developers can prevent common programming errors like buffer overflows, which could otherwise lead to crashes or exploitation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | exit-handler-check | Flags out-of-bounds array accesses. |
| Cppcheck | 2.11 | exceptionCaught | Identifies unsafe array or pointer operations. |
| Coverity | 2024.3 | EXIT\_EXCEPTION | |  | | --- | |  |  |  | | --- | | Detects invalid memory reads or writes. | |
| GCC | 12.3.0 | Built-in Compiler Warning | Checks for potential buffer overruns. |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

By integrating security measures into every stage of the DevOps toolchain, the process evolves into DevSecOps. During the “Assess and Plan” phase, threat modeling is conducted to identify potential vulnerabilities. In the “Design” and “Build” phases, security within integrated development environments (IDEs) is addressed. The “Verify & Test” phase incorporates static application security testing (SAST), automated security scans, and traditional unit, integration, and other testing methodologies.

After deployment to production, automated testing continues with a focus on prevention through integrity checks and defense-in-depth strategies. Continuous threat detection is supported by techniques such as network monitoring, penetration testing, and analyzing performance logs. Security testing, like quality assurance testing, should be implemented early and performed frequently throughout the development lifecycle.

### 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 | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | High | Low | High | 2 |
| STD-003-CPP | High | Medium | Low | High | 1 |
| STD-004-CPP | Critical | High | Medium | High | 1 |
| STD-005-CPP | High | Medium | Medium | High | 1 |
| STD-006-CPP | Low | Medium | Low | Medium | 2 |
| STD-007-CPP | Low | Medium | Low | Medium | 2 |
| STD-008-CPP | High | Medium | Medium | High | 1 |
| STD-009-CPP | High | High | Medium | Critical | 1 |
| STD-010-CPP | High | High | Low | Critical | 1 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 protects data stored on a device or system, such as databases, file systems, or backups. Data is converted into an encrypted format using algorithms like AES-256, making it unreadable without the decryption key.  This policy ensures that all stored sensitive data, such as personally identifiable information (PII), financial records, and intellectual property, is encrypted to prevent unauthorized access. It applies whenever data is saved to persistent storage, including databases, cloud storage, and local drives. It helps mitigate risks from device theft or unauthorized access to storage systems. |
| Encryption in flight | Encryption in flight secures data as it is transmitted between systems or across networks, using protocols like TLS (Transport Layer Security) or HTTPS. This ensures data integrity and confidentiality during communication.  This policy mandates the use of secure transmission protocols for all sensitive data, such as login credentials, payment information, or confidential communications. It applies during API calls, email exchanges, or any network communication to prevent interception by attackers (e.g., man-in-the-middle attacks). |
| Encryption in use | |  | | --- | | Encryption in use protects data being actively processed or used in memory, often by leveraging secure enclaves or homomorphic encryption techniques. This ensures that data remains secure even when actively being computed. |  |  | | --- | | This policy is critical for environments like cloud computing or systems handling highly sensitive data. It applies when processing data such as cryptographic keys, financial transactions, or healthcare information, to prevent exposure to memory-based attacks like RAM scraping or unauthorized system access. | |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | |  | | --- | | Authentication verifies the identity of users or systems through mechanisms like passwords, biometrics, or multi-factor authentication (MFA). This step ensures that only legitimate entities can access the system. |  |  | | --- | | This policy enforces the use of strong and secure authentication methods for accessing systems, networks, and applications. It applies to all users and systems to prevent unauthorized access and breaches. Examples include requiring MFA for remote logins or integrating single sign-on (SSO) solutions. | |
| Authorization | |  | | --- | | Authorization determines what an authenticated user or system is permitted to do, based on roles, permissions, or policies. This ensures that access is granted strictly on a need-to-know basis. |  |  | | --- | | This policy applies role-based access control (RBAC) or attribute-based access control (ABAC) to define granular permissions for users and systems. It applies to critical resources, such as restricting access to sensitive files, APIs, or administrative tools, minimizing the impact of a compromised account. | |
| Accounting | Accounting (or auditing) tracks and records actions performed by users or systems, such as login attempts, data access, or system changes. This ensures accountability and aids in detecting suspicious activities.  This policy requires logging all relevant activities with secure log storage and retention. It applies to all critical systems and processes, ensuring a trail of evidence for compliance, forensic analysis, and troubleshooting. Periodic log reviews and automated anomaly detection are integral parts of the policy. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

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

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

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

|  |  |  |
| --- | --- | --- |
| **Standard** | **Principles** | **Justification** |
| * **STD-001-CPP** | * 1, 4, 7 | * **Principle 1 (Least Privilege):** Ensures array bounds checking to prevent unintentional access to sensitive memory. **Principle 4 (Secure Defaults):** Encourages secure coding practices by default. **Principle 7 (Fail Securely):** Protects against undefined behavior by ensuring secure code handling. |
| * **STD-002-CPP** | * 2, 5, 8 | * **Principle 2 (Defense in Depth):** Validates multiple layers of security checks in pointer operations. **Principle 5 (Separation of Duties):** Enforces strict handling of pointers to avoid overlap. **Principle 8 (Avoid Common Weaknesses):** Mitigates common vulnerabilities like dangling pointers. |
| * **STD-003-CPP** | * 3, 4, 6 | * **Principle 3 (Economy of Mechanism):** Simplifies input validation. **Principle 4 (Secure Defaults):** Ensures input is verified as secure by default. **Principle 6 (Open Design):** Enables review and validation of security mechanisms through transparent code practices. |
| * **STD-004-CPP** | * 1, 5, 8 | * **Principle 1 (Least Privilege):** Limits privileges needed to execute loops. **Principle 5 (Separation of Duties):** Segregates loop iteration logic and user-controlled inputs. **Principle 8 (Avoid Common Weaknesses):** Prevents common errors like infinite loops. |
| * **STD-005-CPP** | * 3, 7, 10 | * **Principle 3 (Economy of Mechanism):** Streamlines exception handling mechanisms. **Principle 7 (Fail Securely):** Ensures system stability during errors. **Principle 10 (Accountability):** Logs all exceptions for audit purposes. |
| * **STD-006-CPP** | * 2, 4, 6 | * **Principle 2 (Defense in Depth):** Adds extra checks for buffer security. **Principle 4 (Secure Defaults):** Validates buffer boundaries by default. **Principle 6 (Open Design):** Encourages collaborative buffer validation. |
| * **STD-007-CPP** | * 1, 3, 9 | * **Principle 1 (Least Privilege):** Restricts unnecessary access. **Principle 3 (Economy of Mechanism):** Makes resource management simpler. **Principle 9 (Keep It Simple):** Reduces complexity in resource allocation. |
| * **STD-008-CPP** | * 2, 8, 10 | * **Principle 2 (Defense in Depth):** Adds multiple layers of integrity checks. **Principle 8 (Avoid Common Weaknesses):** Protects against common data corruption vulnerabilities. **Principle 10 (Accountability):** Audits data integrity violations. |
| * **STD-009-CPP** | * 3, 4, 9 | * **Principle 3 (Economy of Mechanism):** Encourages minimalism in cryptographic routines. **Principle 4 (Secure Defaults):** Enforces strong encryption as default. **Principle 9 (Keep It Simple):** Simplifies encryption workflows for developers. |
| * **STD-010-CPP** | * 2, 7, 8 | * **Principle 2 (Defense in Depth):** Enhances security by validating boundary conditions. **Principle 7 (Fail Securely):** Ensures secure fallback in case of out-of-bounds access. **Principle 8 (Avoid Common Weaknesses):** Reduces risks of boundary-related vulnerabilities. |

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 |  |
| 1.1 | 11/16/2024 | Module 3 | Rogan Page | [Insert text.] |
| 1.2 | 11/22/2024 | Module 4 | Rogan Page | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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