**Green Pace Developer: Security Policy Guide**



# 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 in 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. Validate Input Data | This principle emphasizes the critical need to rigorously check and sanitize all data received from external sources before it is processed or used within an application. Failing to validate input can lead to various vulnerabilities, including buffer overflows, injection attacks, and unexpected program behavior, by allowing malicious data to corrupt program logic or data structures. |
| 1. Heed Compiler Warnings | This principle stresses the importance of treating compiler warnings as errors during the development process. Compiler warnings often highlight potential issues, subtle bugs, or insecure coding practices that, if ignored, could lead to vulnerabilities, undefined behavior, or runtime errors in the deployed software. Addressing them proactively enhances code quality and security. |
| 1. Architect and Design for Security Policies | Security must be an integral part of the software development lifecycle from the earliest stages of architecture and design. This principle means that security policies and considerations should guide the design decisions, ensuring that security is built-in rather than bolted on. This proactive approach helps to establish a robust security foundation and reduces the cost and complexity of addressing vulnerabilities later. |
| 1. Keep It Simple | This principal advocates for simplicity in design and implementation. Complex systems are harder to understand, test, and secure, often introducing more opportunities for vulnerabilities. By keeping code and design as simple as possible, the attack surface is reduced, and it becomes easier to identify, fix, and prevent security flaws. |
| 1. Default Deny | This principle, also known as "default to security" or "fail-safe defaults," dictates that access to resources or functionalities should be denied by default unless explicitly granted. This approach minimizes unauthorized access by ensuring that only approved operations are allowed, rather than attempting to enumerate and deny all possible unauthorized actions. |
| 1. Adhere to the Principle of Least Privilege | This principle asserts that every program, process, or user should be allocated only the essential permissions required to fulfill its intended function, and no more. By restricting privileges, the potential damage from a compromised component is minimized, which helps to contain the attack surface and mitigate the impact of a successful breach. |
| 1. Sanitize Data Sent to Other Systems | Data exchanged between different systems or components (e.g., databases, APIs, other services) must be thoroughly sanitized and encoded to prevent malicious content from being misinterpreted or executed by the receiving system. This principle is crucial for preventing cross-system vulnerabilities like injection attacks and ensuring data integrity across an entire ecosystem. |
| 1. Practice Defense in Depth | This principle involves implementing multiple layers of security controls throughout a system, rather than relying on a single point of defense. If one layer of defense is breached, another layer stands ready to protect the system. This layered approach provides redundancy and significantly increases the effort an attacker must expend to compromise the system |
| 1. Use Effective Quality Assurance Techniques | This principle emphasizes the importance of integrating comprehensive quality assurance (QA) and testing methodologies into the development process. This includes security testing (like penetration testing, vulnerability scanning, and fuzzing) to identify and mitigate security flaws early. Effective QA ensures that security standards are met and vulnerabilities are identified before deployment. |
| 1. Adopt a Secure Coding Standard | This principle highlights the necessity of establishing and consistently following a set of secure coding guidelines and best practices specific to the programming language(s) and environment in use. Adopting such a standard helps developers avoid common pitfalls that lead to vulnerabilities, promotes consistent secure coding practices across the team, and enhances the overall security posture of the software. |

### 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 non-compliant 001sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Integer Overflow Prevention |

| **Noncompliant Code** |
| --- |
| This code snippet performs an addition that can result in an integer overflow if a and b are large, potentially leading to a small positive or negative value being assigned to the result. |
| int a = 2000000000;  int b = 2000000000;  int result = a + b; |

| **Compliant Code** |
| --- |
| This compliant code checks for potential overflow before performing the addition. If an overflow is detected, it handles the error appropriately (e.g., by throwing an exception or returning an error code), preventing unintended behavior. |
| int a = 2000000000;  Int b = 2000000000;  long long result = (long long)a + b;  `if (result > std::numeric\_limits::max()  // Handle overflow error  throw std::overflow\_error("Integer overflow detected");  }  int final\_result = static\_cast<int>(result); |

| **Principles(s):** 1. Validate Input Data, 2. Heed Compiler Warnings |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | V8.9+ | Rule cpp:S3466 | Integer literals should not be implicitly converted to a narrower type |
| Cppcheck | v2.10+ | arrayIndexOutOfBounds check | Check for potential overflows affecting array indices |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Value | [STD-002-CPP] | Input Validation for Numeric Values |

| **Noncompliant Code** |
| --- |
| This function directly uses an integer value obtained from user input to determine an array size without any validation, making it vulnerable to denial-of-service or buffer overflow attacks if a negative or excessively large value is provided. |
| void process\_data(int size) {  int\* arr = new int[size]; // Vulnerable: size is not validated  // ... use arr ...  delete[] arr;  } |

| **Compliant Code** |
| --- |
| This compliant code validates the size parameter to ensure it is within a safe and reasonable range before allocating memory. This prevents negative sizes, excessively large allocations, and potential buffer overruns. |
| void process\_data(int size) {  ` if (size <= 0  throw std::invalid\_argument("Invalid size provided.");  }  int\* arr = new int[size];  // ... use arr ...  delete[] arr;  } |

| **Principles(s):** 1. Validate Input Data, 5. Default Deny |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v8.9+ | Rule cpp:S3626 | Literals should not be compared with other literals |
| Coverity | v2022.06+ | Out-of-bounds array access | Checks for out-of-bounds array access due to unvalidated input |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Secure String Handling (Null Termination) |

| **Noncompliant Code** |
| --- |
| This code uses strcpy, which does not check buffer boundaries, making it susceptible to buffer overflow if the source is longer than dest\_size - 1. |
| void copy\_string(char\* dest, const char\* source, size\_t dest\_size) {  strcpy(dest, source); // Vulnerable: No bounds checking  } |

| **Compliant Code** |
| --- |
| This compliant code uses strncpy, which takes the destination buffer size as an argument, preventing buffer overruns. It also explicitly null-terminates the destination string, unlike strncpy, which does not guarantee null termination if the source string fills the buffer. |
| void copy\_string(char\* dest, const char\* source, size\_t dest\_size) {  strncpy(dest, source, dest\_size - 1);  dest[dest\_size - 1] = '\0'; // Ensure null termination  } |

| **Principles(s):** 7. Sanitize Data Sent to Other Systems, 6. Adhere to the Principle of Least Privilege |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v8.9+ | Rule cpp:S1027 | Using strcpy, strcat, etc. is security-sensitive |
| Cppcheck | v2.10+ | strcpy-vuln check | Checks for strcpy and strcat vulnerabilities |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Parameterized Queries for Database Access |

| **Noncompliant Code** |
| --- |
| This function constructs an SQL query by concatenating user-provided input directly into the string, making it vulnerable to SQL injection. |
| void get\_user\_data(const std::string& username) {  std::string query = "SELECT \* FROM Users WHERE Username = '" + username + "';";  // Execute query...  } |

| **Compliant Code** |
| --- |
| This compliant code uses a parameterized query. The? Acts as a placeholder for the username, and the value is bound separately, ensuring that the database treats the input as data, not as executable SQL code, thus preventing injection attacks. |
| void get\_user\_data(const std::string& username) {  // Assuming a database API that supports prepared statements  std::string sql = "SELECT \* FROM Users WHERE Username = ?;";  PreparedStatement\* stmt = db\_connection->prepareStatement(sql);  stmt->setString(1, username);  ResultSet\* rs = stmt->executeQuery();  // Process results...  delete rs;  delete stmt;  } |

| **Principles(s):** 7. Sanitize Data Sent to Other Systems, 8. Practice Defense in Depth |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | v2.11+ | SQL Injection (DAST) | Can detect SQL injection vulnerabilities during dynamic analysis |
| SonarQube | v8.9+ | Rule csharp:S5147 | Detects SQL injection vulnerabilities in C# code |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Use Smart Pointers for Dynamic Memory |

| **Noncompliant Code** |
| --- |
| This code uses raw pointers for dynamic memory allocation, requiring manual deallocation. If an exception occurs before delete is called, a memory leak will occur. |
| void process\_resource() {  A\* obj = new A();  // Potentially complex operations that might throw  obj->do\_something();  delete obj; // If exception before this, memory leaks  } |

| **Compliant Code** |
| --- |
| This compliant code uses std::unique\_ptr to manage the dynamically allocated object. The memory is automatically deallocated when ptr goes out of scope, even if an exception occurs, preventing memory leaks and simplifying resource management. |
| void process\_resource() {  std::unique\_ptr<A> ptr = std::make\_unique<A>();  // Operations  ptr->do\_something();  // No explicit delete needed  } |

| **Principles(s):** 1. Validate Input Data, 2. Heed Compiler Warnings |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v8.9+ | Rule cpp:S3970 | Memory should be cleaned or cleared |
| Valgrind | v3.18+ | Memory leak checks | A dynamic analysis tool for memory-related issues |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Appropriate Use of Assertions |

| **Noncompliant Code** |
| --- |
| This code uses assert to validate user input. In a release build where assertions are compiled out, the validation check would be missing, potentially leading to a division by zero error at runtime. |
| double divide\_input(int numerator, int denominator) {  assert(denominator != 0); // Vulnerable: assert removed in release builds  return static\_cast<double>(numerator) / denominator;  } |

| **Compliant Code** |
| --- |
| This compliant code uses proper runtime error handling (e.g., throwing an exception) to validate input that originates from untrusted sources or represents a recoverable error condition. Assertions are reserved for internal program logic errors that should not occur in correct code. |
| double divide\_input(int numerator, int denominator) {  if (denominator == 0) {  throw std::invalid\_argument("Denominator cannot be zero.");  }  return static\_cast<double>(numerator) / denominator;  } |

| **Principles(s):** 3. Architect and Design for Security Policies, 4. Keep It Simple |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | v7.21+ | Checks for misuse of assertions | Detects potential issues with assertions, especially in release builds |
| SonarQube | v8.9+ | Rule cpp:S107 | Detects functions with too many parameters, which can indicate complexity |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Consistent Exception Handling |

| **Noncompliant Code** |
| --- |
| This function uses a broad catch(...) block without specific handling or resource cleanup, potentially masking critical errors and leading to resource leaks if do\_risky\_operation allocates resources before throwing. |
| void perform\_operation() {  Resource\* res = new Resource();  try {  do\_risky\_operation();  }  catch (...) {  // Catch all, no specific handling or cleanup  }  delete res; // If exception caught, this might not be reached  } |

| **Compliant Code** |
| --- |
| This compliant code uses RAII for resource management (e.g., smart pointers or custom resource wrappers) to ensure resources are automatically released when the scope is exited, regardless of whether an exception occurs. Specific exception types are caught for targeted recovery, avoiding broad catch-all blocks. |
| void perform\_operation() {  std::unique\_ptr<Resource> res = std::make\_unique<Resource>(); // RAII for resource  try {  do\_risky\_operation();  }  catch (const SpecificError& e) {  // Handle specific error  std::cerr << "Specific error: " << e.what() << std::endl;  }  catch (const std::exception& e) {  // Handle other standard exceptions  std::cerr << "Standard error: " << e.what() << std::endl;  }  // Resource 'res' is automatically cleaned up when ptr goes out of scope  } |

| **Principles(s):** 9. Use Effective Quality Assurance Techniques, 10. Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v8.9+ | Rule cpp:S112 | Exception handlers should be declared with the most specific exception type |
| C++ Core Guidelines Checker | v1.0+ | E.16. catch(T&) over catch(T) | Checks for proper exception handling practices |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Randomness | [STD-008-CPP] | Secure Use of Random Numbers |

| **Noncompliant Code** |
| --- |
| This code uses rand() and srand() seeded with time(NULL), which is a predictable and cryptographically weak source of randomness, unsuitable for security purposes. |
| void generate\_weak\_token() {  srand(time(NULL));  int token = rand(); // Cryptographically weak random number  // ... use token ...  } |

| **Compliant Code** |
| --- |
| This compliant code uses std::random\_device and std::mt19937 for generating random numbers. For cryptographic purposes, a cryptographically secure pseudo-random number generator (CSPRNG) should be used (e.g., from a dedicated crypto library) to ensure unpredictability and strong randomness. |
| void generate\_secure\_token() {  std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> distrib(1, 1000000);  int token = distrib(gen); // Stronger pseudo-random number  // For cryptographic use, prefer a CSPRNG from a security library.  } |

| **Principles(s):** 7. Sanitize Data Sent to Other Systems, 8. Practice Defense in Depth |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | v2.11+ | Predictable Session ID (DAST) | Can check for predictable session IDs which are often generated using weak random numbers |
| SonarQube | v8.9+ | Rule java:S2245 | Detects use of weak random number generators in Java code |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File I/O | [STD-009-CPP] | Secure Temporary File Creation |

| **Noncompliant Code** |
| --- |
| This code attempts to create a temporary file with a hardcoded or predictable filename, making it vulnerable to race conditions where an attacker could create the file or a symlink before the program does. |
| void create\_temp\_file\_insecure() {  std::string temp\_filename = "/tmp/myapp\_temp.txt";  std::ofstream file(temp\_filename); // Vulnerable to race condition  file << "Temporary data";  file.close();  } |

| **Compliant Code** |
| --- |
| This compliant code uses std::filesystem::temp\_directory\_path (C++17) to get a secure temporary directory and std::tmpfile or similar OS-specific secure functions to create unique, secure temporary files, which handle race conditions and permissions automatically. |
| void create\_temp\_file\_secure() {  // Using std::tmpfile for secure creation of binary temporary file  FILE\* tmpf = std::tmpfile();  if (tmpf) {  fputs("Temporary data", tmpf);  fclose(tmpf); // File is deleted when closed  }  // For text files or specific needs, use unique naming and proper permissions:  // #include <filesystem>  // std::filesystem::path temp\_dir = std::filesystem::temp\_directory\_path();  // std::string unique\_name = "myapp\_temp\_" + std::to\_string(std::chrono::high\_resolution\_clock::now().time\_since\_epoch ().count()) + ".txt";  // std::filesystem::path full\_path = temp\_dir / unique\_name;  ` // std::ofstream file(full\_path, std::ios::binary  // file << "Temporary data";  // file.close();  } |

| **Principles(s):** 5. Default Deny, 6. Adhere to the Principle of Least Privilege |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | v2022.06+ | INSECURE\_TEMPFILE | Checks for insecure use of mktemp() or tmpnam() |
| SonarQube | v8.9+ | Rule cpp:S3011 | Detects use of insecure temporary file functions |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Arguments | [STD-010-CPP] | Secure Handling of Command Line Arguments |

| **Noncompliant Code** |
| --- |
| This code directly uses a command-line argument in a printf format string, making it vulnerable to a format string vulnerability if a malicious string is provided as an argument. |
| int main(int argc, char\* argv[]) {  if (argc > 1) {  printf(argv[1]); // Vulnerable to format string attack  }  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code explicitly specifies the format string for printf, ensuring that argv[1] is treated as a string literal and not as a format specifier. This prevents format string vulnerabilities. Additionally, it checks for sufficient arguments. |
| int main(int argc, char\* argv[]) {  if (argc > 1) {  printf("%s\n", argv[1]); // Compliant: explicit format string  }  return 0;  } |

| **Principles(s):** 1. Validate Input Data, 7. Sanitize Data Sent to Other Systems |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | v2022.06+ | FORMAT\_STRING | Checks for format string vulnerabilities |
| SonarQube | v8.9+ | Rule cpp:S2636 | Detects unsafe use of I/O functions that can lead to format string vulnerabilities |

### 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 Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of complaint and noncompliant code. Coding standards begin with 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

.



Automation is a critical component of Green Pace's DevSecOps strategy, allowing for continuous enforcement of security policies and early detection of vulnerabilities. The goal is to "shift left" and integrate security tools directly into the development pipeline. The existing DevOps process will be modified to include security automation at key stages, as defined by our DevSecOps diagram:

* **Secure Code Development (Create):** During the **Create** phase, developers will use Integrated Development Environment (IDE) plugins and Git hooks to enforce secure coding practices in real-time. This includes static analysis tools that check for compliance with coding standards immediately upon saving or committing code.
* **Static Application Security Testing (Verify):** In the **Verify** phase, the Continuous Integration (CI) pipeline will automatically trigger Static Application Security Testing (SAST) tools, such as SonarQube. These tools will scan the entire codebase to identify any security vulnerabilities or policy violations before the code is merged. Any issues with a threat level of "High" or "Critical" will automatically fail the build.
* **Dynamic Application Security Testing (Predict):** In the **Predict** phase, the application will be deployed to a staging or pre-production environment. Dynamic Application Security Testing (DAST) tools, like OWASP ZAP, will then be used to scan the running application. DAST tools are crucial for identifying vulnerabilities that only appear at runtime, such as SQL injection, cross-site scripting (XSS), and misconfigurations.
* **Runtime Monitoring and Response (Detect/Respond):** The **Detect/Respond** phases focus on post-deployment security. In this stage, automated tools will monitor the live application for security incidents, threats, and anomalies. Security Information and Event Management (SIEM) systems will aggregate logs and alerts to detect potential attacks. In the event of a security incident, automated systems will trigger predefined incident response playbooks to mitigate the threat and log the incident for post-mortem analysis.

### 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 | Medium | Unlikely | Medium | Medium | 2 |
| STD-002-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-003-CPP | High | Likely | High | High | 4 |
| STD-004-CPP | Critical | Likely | High | Critical | 5 |
| STD-005-CPP | High | Likely | High | High | 4 |
| STD-006-CPP | Low | Unlikely | Low | Low | 1 |
| STD-007-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-008-CPP | Critical | Likely | High | High | 4 |
| STD-009-CPP | High | Likely | High | Critical | 5 |
| STD-0010-CPP | High | Unlikely | Medium | High | 3 |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 | This policy dictates that all sensitive data stored on any Green Pace system—including databases, file servers, and backups—must be encrypted. This protects data from unauthorized access if the physical storage media is compromised. It applies to all persistent data storage and is a core part of our defense-in-depth strategy. We will use industry-standard algorithms like AES-256 for encryption. |
| Encryption in flight | This policy requires that all data transmitted between applications, services, or users must be encrypted. This prevents eavesdropping and man-in-the-middle attacks. It applies to all network communication, whether internal or external, and is typically implemented using protocols like TLS/SSL. |
| Encryption in use | This policy addresses the security of data while it is being actively processed by a computer. The policy requires that data remains encrypted in memory, only being decrypted for the specific operation it is needed for. It applies to mission-critical applications that handle highly confidential information. |

| 1. Triple-A Framework\* | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | User Authentication Policy: This policy defines the rules for verifying the identity of a user or system. It requires the use of strong, multi-factor authentication (MFA) for all critical systems and applications. This policy applies to every user login and system-to-system access. It helps protect against unauthorized access from stolen credentials. |
| Authorization | Access Control and Authorization Policy: This policy specifies which authenticated users or systems can access specific resources or perform certain actions. It is based on the principle of least privilege, meaning users are granted only the minimum access necessary to perform their jobs. This policy applies to all resource access attempts after authentication and is enforced through role-based access control (RBAC). |
| Accounting | System Activity and Auditing Policy: This policy requires the logging and monitoring of all security-relevant events, such as user login, access attempts, and system modifications. This creates an audit trail that can be used to detect and investigate security incidents. The policy applies to all systems and services and is crucial for maintaining accountability and providing data for threat analysis. |

\*Use this checklist for 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

## Mapping Principles to Standards

This section maps the 10 principles to the coding standards to show how our security policy is based on widely accepted security principles.

* **Principle 1: Least Privilege** supports standards **1** (Data Type Correctness) and **5** (Memory Protection) by ensuring that code operates with the minimum required permissions, preventing unauthorized access and memory manipulation.
* **Principle 2: Secure Defaults** supports standards **1** (Data Type Correctness) and **5** (Memory Protection) by ensuring that the default state of all systems is secure, reducing the attack surface.
* **Principle 3: Fail-Safe Defaults** supports standards **2** (Data Value Correctness) and **6** (Assertions) by ensuring that if an action fails, the system defaults to a secure state, such as denying access.
* **Principle 4: Economy of Mechanism** supports standards **2** (Data Value Correctness) and **6** (Assertions) by favoring simple and elegant security mechanisms that are easier to analyze and trust.
* **Principle 5: Complete Mediation** supports standards **3** (String Correctness) and **8** (Input Validation) by requiring that every access attempt to an object is checked for authority.
* **Principle 6: Open Design** supports standards **3** (String Correctness) and **8** (Input Validation) by not depending on secrecy for security, allowing for community review and improvement.
* **Principle 7: Separation of Privilege** supports standards **4** (SQL Injection Policy) and **9** (Secure Communication) by requiring that a malicious actor must compromise multiple, independent components to breach the system.
* **Principle 8: Psychological Acceptability** supports standards **4** (SQL Injection Policy) and **9** (Secure Communication) by ensuring that security mechanisms are easy to use, so they don't hinder users and developers.
* **Principle 9: Defense in Depth** supports standards **7** (Exceptions Policy) and **10** (Secure Configuration) by layering multiple security controls to protect a system.
* **Principle 10: Timely Updates** supports standards **7** (Exceptions Policy) and **10** (Secure Configuration) by ensuring that security vulnerabilities are patched and systems are kept up to date.

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 with standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards define 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 of any exception must be granted by the chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates at the 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 | 08/10/2025 | Completed Policy | Terry Johnson Jr. |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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