**Green Pace Developer: Security Policy Guide Template**



# 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 4](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 11](#_Toc52464065)

[Coding Standard 8 13](#_Toc52464066)

[Coding Standard 9 15](#_Toc52464067)

[Coding Standard 10 16](#_Toc52464068)

[Defense-in-Depth Illustration 17](#_Toc52464069)

[Project One 17](#_Toc52464070)

[1. Revise the C/C++ Standards 17](#_Toc52464071)

[2. Risk Assessment 17](#_Toc52464072)

[3. Automated Detection 17](#_Toc52464073)

[4. Automation 17](#_Toc52464074)

[5. Summary of Risk Assessments 18](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 18](#_Toc52464076)

[7. Map the Principles 19](#_Toc52464077)

[Audit Controls and Management 20](#_Toc52464078)

[Enforcement 20](#_Toc52464079)

[Exceptions Process 20](#_Toc52464080)

[Distribution 21](#_Toc52464081)

[Policy Change Control 21](#_Toc52464082)

[Policy Version History 21](#_Toc52464083)

[Appendix A Lookups 21](#_Toc52464084)

[Approved C/C++ Language Acronyms 21](#_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 | Validate input from all untrusted sources before processing. Proper input validation can eliminate the vast majority of software vulnerabilities—buffer overflows, injection attacks, and logic errors—ensuring that only well-formed and expected data is handled |
| 1. Heed Compiler Warnings | Compile your code with the highest warning level and address all warnings promptly. Doing so helps uncover potential vulnerabilities early and ensures more secure and stable software |
| 1. Architect and Design for Security Policies | Architect software to enforce security policies from the ground up, such as privilege separation. Designing systems around security requirements minimizes vulnerabilities and supports strong, enforceable protections |
| 1. Keep It Simple | Embrace simplicity in design and implementation. Complex systems are more prone to errors, hard to understand, and expensive to secure, increasing the likelihood of vulnerabilities |
| 1. Default Deny | Ensure access control decisions default to denying permission. Only explicitly permitted actions are allowed, reducing unauthorized access and shrinking the attack surface |
| 1. Adhere to the Principle of Least Privilege | Execute processes and grant permissions with the minimal privileges necessary—and only when needed. Limiting privileges in this way helps contain damage if a vulnerability is exploited |
| 1. Sanitize Data Sent to Other Systems | Cleanse and validate all data before sending it to external subsystems (e.g., databases, command shells). This prevents injection attacks and avoids misinterpreting data in contexts the sender doesn’t control |
| 1. Practice Defense in Depth | Use multiple, complementary layers of security. If one control fails, others continue to defend the system, reducing risk from successful exploits |
| 1. Use Effective Quality Assurance Techniques | Employ robust testing, fuzzing, penetration testing, code reviews, and independent audits. These techniques identify and mitigate vulnerabilities early in development |
| 1. Adopt a Secure Coding Standard | Follow established coding standards (such as SEI CERT C++) to ensure consistent, secure, and maintainable code practices across all development |

### 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] | **Data Type – Use Fixed‑Width Integer Types;** Using fixed-width types like int32\_t, uint64\_t ensures consistent integer sizes across platforms, avoiding overflow or truncation errors due to platform-dependent type sizes. |

| **Noncompliant Code** |
| --- |
| Uses default integer types that may vary in bit width across systems, leading to unpredictable behavior. |
| int count = 2147483647;  uint64\_t total = count; |

| **Compliant Code** |
| --- |
| Uses fixed-width integer types for clarity and consistent behavior across platforms. |
| int32\_t count = 2147483647;  uint64\_t total = static\_cast<uint64\_t>(count); |

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

| **Principles(s):** *Principle of Least Astonishment (POLA)* – This principle states that software should behave in a predictable and consistent manner. By using fixed-width integer types, developers avoid unexpected differences in integer size across platforms, which could cause logic errors, overflow, or data loss. This aligns with secure coding best practices by ensuring predictable behavior in numerical operations and reducing the risk of security vulnerabilities stemming from incorrect type assumptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 17.0+ | Modernize-use-using/cppcoreguidelines-avoid-magic-numbers | Detects inconsistent integer type usage and enforces coding standards |
| Cppcheck | 2.13+ | Portability | Identifies integer type mismatches that may cause portability issues |
| SonarQube | 10.5+ | C:S3981 | Flags the use of non-fixed-width integer types where fixed-width alternatices are recommended |
| Coverity scan | 2024.6 | MISRA C++:2008 Rule3-9-1 | Scans for violations of coding standards including enforcing fixed-width integer types for portability and safety |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | **Data Value – Avoid Magic Numbers;** Hard-coded “magic” constants are unclear and error-prone; named constants improve readability and simplify maintenance. |

| **Noncompliant Code** |
| --- |
| Uses a hard-coded value with no context, reducing clarity. |
| if (score > 65) {  // pass  } |

| **Compliant Code** |
| --- |
| Defines and uses a named constant for clarity and maintainability. |
| const int PASSING\_SCORE = 65;  if (score > PASSING\_SCORE) {  // pass  } |

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

| **Principles(s):** Principle of Clarity and Maintainability – This standard maps to the principle by requiring that code express intent clearly and be easy to modify. By replacing magic numbers with named constants, developers ensure that values have meaningful names, improving readability, reducing misunderstanding, and allowing easier updates in one location instead of searching for hard-coded values across the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | Cpp:S109 | Flags usage of magic numbers in comparisons or assignments |
| Cppcheck 2.14 | 2.14 | magicNumber | Detects hard-coded numeric values and suggests constant usage |
| PCS-Studio | 7.28 | V112 | warns when literals are usesd instead of symbolic constants |
| Clang-Tidy | 17.0 | Readability-magic-numbers | Identifies magic numbers and encourages names constants |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | **String Correctness – Null‑Terminate C‑Style Strings**; All C-style strings must be null-terminated to avoid buffer overruns and undefined behavior. |

| **Noncompliant Code** |
| --- |
| Omits null terminator, risking buffer overflow when reading. |
| char s[4] = {'t','e','s','t'}; // no '\0' |

| **Compliant Code** |
| --- |
| Ensures proper null-termination by using string literal. |
| char s[5] = "test"; // includes '\0' |

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

| **Principles(s):** **Fail-Safe Defaults (Default Deny)** – Ensuring all C-style strings are properly null-terminated prevents buffer overflows, a common security vulnerability that can lead to unpredictable behavior or exploits. This standard enforces safe defaults by explicitly terminating strings. **Defense in Depth** – Proper string termination adds a critical layer to prevent buffer overflows that could be exploited if unchecked. This complements other memory safety measures and reduces risk from input-related attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | Cpp:S5164 | Detects missing null terminators and unsafe string usage |
| Cppcheck | 2.14 | nullTerminator | Flags C-style strings missing proper null termination |
| Coverity Scan | 2024.6 | CWE-119 | Identifies buffer overflow risks due to missing nulls |
| Clang-Tidy | 17.0 | Bugprone-string-null-termination | Checks for string termination issues in C-style strings |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | **SQL Injection – Use Parameterized Queries**; Constructing SQL commands by concatenating strings with user input enables SQL injection attacks; parameterized queries avoid this risk. |

| **Noncompliant Code** |
| --- |
| Directly embeds user input into the SQL statement. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userName + "';";  db.execute(query); |

| **Compliant Code** |
| --- |
| Uses parameter binding to prevent injection. |
| auto stmt = db.prepare("SELECT \* FROM users WHERE name = ?");  stmt.bind(1, userName);  stmt.execute(); |

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

| **Principles(s):** **Validate Input Data** – Validating and sanitizing all user inputs ensures that SQL statements do not include malicious payloads. Using parameterized queries enforces strict separation of code and data, preventing injection attacks. **Default Deny** – By default, disallowing unsafe concatenation of input and only allowing parameterized queries ensures a secure baseline, reducing attack surface for unauthorized data access. **Defense in Depth** – Parameterized queries add a critical security layer preventing injection, which complements other security controls such as input validation and access control. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | Cpp:S3649 | Detects unsafe SQL string concatenations susceptible to injection |
| Fortify Static | 22.1 | SQL injection | Identifies direct concatenations of untrusted input in SQL queries |
| Cppcheck | 2.14 | Sql-injection | Flags concatenation of user input into SQL queries |
| Checkmarx | 9.2 | CWE-89 | Scans for injection vulnerabilities and improper query construction |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | **Memory Protection – Manage Memory with RAII;** Using RAII and smart pointers ensures automatic resource release and prevents memory leaks and dangling pointers. |

| **Noncompliant Code** |
| --- |
| Uses manual new and omits delete, leading to a memory leak. |
| void f() {  int\* p = new int[10];  } // memory never freed |

| **Compliant Code** |
| --- |
| Uses unique\_ptr to automatically manage memory lifecycle. |
| void f() {  std::unique\_ptr<int[]> p(new int[10]);  } // memory released at end of scope |

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

| **Principles(s):** **Validate Input Data** – Proper memory management prevents unintended behavior caused by invalid memory access or buffer overruns, helping maintain data integrity. **Keep It Simple** – Using RAII and smart pointers simplifies resource management by automating cleanup, reducing the risk of memory leaks from manual errors. **Defense in Depth** – Automated resource management layers additional safety against memory corruption vulnerabilities that could be exploited if unmanaged memory is accessed improperly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | Cpp:S3776 | Detects manual memory management issues and encourages RAII |
| Clang Static Analyzer | 15.0 | Memory leak checks | finds memory leaks and improper resource management |
| Coverity | 2024.1 | MEM\_001 | Identifies missing deletes and unsafe memory usage patterns |
| PVS\_studio | 7.20 | V501 | Detects memory leaks and unsafe manual memory management |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | **Assertions – Check Preconditions Using assert();** Assertions validate critical assumptions during development; they catch violation early without runtime cost in production builds. |

| **Noncompliant Code** |
| --- |
| Does not verify that input pointer is not null. |
| void process(int\* p) {  // assume p is valid  \*p = 5;  } |

| **Compliant Code** |
| --- |
| Asserts that the pointer must be valid before use. |
| void process(int\* p) {  assert(p != nullptr);  \*p = 5;  } |

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

| **Principles(s):** **Validate Input Data** – Ensuring input assumptions are met prevents processing invalid data that could lead to crashes or security vulnerabilities. **Use Effective Quality Assurance Techniques** – Using assertions during development acts as an early detection tool for bugs and incorrect assumptions. **Keep It Simple** – Assertions provide a clear, simple method to verify critical conditions without complex error handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang static analyzer | 15.0 | Assertion checks | Detects missing assertions and verifies assert usage |
| SonarQube | 10.5 | Cpp:S2583 | Detects tautological asserts and missing null checks |
| PVS-studio | 7.20 | V501 | Finds potential null pointer dereferences and missing asserts |
| Coverity | 2024.1 | ASSERT\_001 | Checks for appropriate usage of assertion in code |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | **Exceptions – Do Not Throw Exceptions from Destructors;** Throwing exceptions in destructors during stack unwinding leads to std::terminate(); destructors should be noexcept and handle errors internally. |

| **Noncompliant Code** |
| --- |
| Destructor throws a runtime error, risking program termination. |
| ~Obj() {  throw std::runtime\_error("error");  } |

| **Compliant Code** |
| --- |
| Destructor is marked noexcept and logs cleanup failure. |
| ~Obj() noexcept {  if (!cleanup()) {  std::cerr << "cleanup failed\n";  }  } |

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

| **Principles(s):** **Keep It Simple** — Handling errors within destructors avoids complex error propagation during stack unwinding, reducing risk of program termination.  **Practice Defense in Depth** — Avoiding exceptions from destructors adds a layer of protection preventing unexpected termination in error conditions.  **Use Effective Quality Assurance Techniques** — Proper error handling and noexcept destructors improve software stability and robustness. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | possible | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 15.0 | Modernize-noexcept | Warns about throwing exceptions from destructors |
| SonarQube | 10.5 | Cpp:S2486 | Detects throwing exceptions in destructors |
| PVS-Studio | 7.20 | V1001 | Identifies unsafe exception throwing in destructors |
| Coverity | 2024.1 | EXC\_002 | Checks for exceptions thrown during stack unwinding |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | **Input Validation – Reject Out‑of‑Range Values;** Even after type checks, numerical values must be validated to ensure they fall within an acceptable range to prevent logic errors or resource misuse. |

| **Noncompliant Code** |
| --- |
| Accepts negative array size without validation. |
| void allocate(int size) {  int\* arr = new int[size];  // ...  } |

| **Compliant Code** |
| --- |
| Validates parameter before allocation. |
| void allocate(int size) {  if (size <= 0 || size > MAX\_SIZE) return;  std::vector<int> arr(size);  // ...  } |

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

| **Principles(s):** **Validate Input Data** — Ensures all inputs are checked for correctness before use, preventing unexpected behavior or resource misuse.  **Default Deny** — Rejecting out-of-range inputs acts as an implicit denial of invalid or potentially harmful data.  **Keep It Simple** — Simple range checks help maintain clear and safe code behavior.  **Practice Defense in Depth** — Input validation adds a defensive layer protecting against invalid parameters that could cause runtime errors or security flaws. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.10 | Uninitvar | Detects uninitialized variables and missing input validation |
| SonarQube | 10.5 | Cpp:S1871 | Detects missing range checks on function parameters |
| PVS-Studio | 7.20 | V5015 | Flags unchecked input parameters and possible out-of-range usage |
| Clang static analyzer | 15.0 | Core.NullDereference | Finds potential null dereferences due to invalid outputs |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | **Resource Cleanup – Close All Opened Files;** Unclosed file handles may exhaust system resources; always close files using RAII wrappers or ensure proper cleanup. |

| **Noncompliant Code** |
| --- |
| Opens a file but doesn’t close it, leading to resource leaks. |
| FILE\* f = fopen("data.txt", "r");  if (!f) return;  // forgot to call fclose(f) |

| **Compliant Code** |
| --- |
| Uses RAII wrapper to ensure file is properly closed. |
| std::unique\_ptr<FILE, decltype(&fclose)> f(fopen("data.txt","r"), &fclose);  if (!f) return;  // automatically closed on scope exit |

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

| **Principles(s):** **Practice Defense in** Depth — Ensuring resource cleanup prevents cascading failures due to resource exhaustion.  **Adopt a Secure Coding Standard** — Using RAII to manage resources follows best coding practices, minimizing leaks.  **Keep It Simple** — Using RAII simplifies resource management and reduces human error.  **Use Effective Quality Assurance Techniques** — Automated tools and code reviews can detect resource leaks early. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.10 | resourceLeak | Detects files opened but not properly closed |
| SonarQube | 10.5 | Cpp:S2092 | Flags opened files handles that are not closed |
| PVS-Studio | 7.20 | V547 | Warns about missing fclose or RAII usage for file handles |
| Clang static analyzer | 15.0 | Core.resourceLeak | Detects resource leaks including unclosed files |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | **Logging – Avoid Logging Sensitive Data;** Writing sensitive data (passwords, PII) to logs can expose it to unauthorized individuals; sanitize or mask such data when logging. |

| **Noncompliant Code** |
| --- |
| Logs full password text, exposing it in logs. |
| std::string pwd = getPassword();  log("User password: " + pwd); |

| **Compliant Code** |
| --- |
| Logs only a placeholder to avoid exposing the actual password. |
| log("User password: [REDACTED]"); |

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

| **Principles(s):** **Default Deny** — Avoid exposing sensitive data by default; do not log unless explicitly safe.  **Practice Defense in Depth** — Masking sensitive data in logs adds an additional layer of protection against data leakage.  **Use Effective Quality Assurance Techniques** — Logging policies and automated scans help detect accidental sensitive data exposure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | Cpp:S2087 | Detects logging of sensitive data such as passwords |
| Veracode | 2025 | SSDL-LOG-001 | Identifies sensitive data exposure through logs |
| Checkmarx | 9.5 | LOGGING\_SENSITIVE\_DATA | Flags potential logging of confidential information |
| Fortify static code analyzer | 22.1 | Log sensitive data | Highlights logging statements that expose sensitive info |

### 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.

The DevSecOps diagram illustrates the continuous and integrated lifecycle of software development, emphasizing security throughout every phase. This continuous loop—depicted as an infinity symbol—ensures that security standards are enforced and compliance is automated as part of the normal development workflow at Green Pace.

**Assess and Plan:**  
In this initial phase, the security team analyzes the current threat landscape, regulatory changes, and prioritizes security backlogs based on risk impact. Automation can be applied here by integrating vulnerability management tools and regulatory compliance checkers into planning tools, ensuring that security requirements for each sprint are clearly defined and prioritized.

**Design:**  
Secure design practices are integrated early using automated static application security testing (SAST) and security-focused design validation tools such as OWASP guideline checkers. Automation in this phase enforces coding standards by scanning design documents and source code before implementation to identify and mitigate security weaknesses.

**Build:**  
During the build phase, automated security tools are integrated with trusted repositories and build pipelines. Tools such as dependency vulnerability scanners, code linters, and static analyzers run automatically on every build to ensure adherence to secure coding standards before code is merged or deployed.

**Verify and Test:**  
Automated testing is essential in this phase, including dynamic application security testing (DAST), penetration testing tools, and continuous vulnerability scanning. These tools validate that the code and configurations comply with the security standards and detect vulnerabilities early.

**Transition and Health Check:**  
Before deployment, automated security configuration and compliance checks (e.g., infrastructure as code scanning, penetration tests) ensure the environment is secure. This phase includes automated compliance audits and deployment validations to prevent misconfigurations or insecure deployments.

**Monitor and Detect:**  
Post-deployment, automation continues with log collection, SIEM (Security Information and Event Management), intrusion detection systems, and analytics to continuously monitor the environment. Automated alerts and incident detection enable rapid identification of security events.

**Respond:**  
Automated response mechanisms such as blocking attacks, shutting down compromised services, and triggering rollback workflows help to quickly contain and remediate threats, minimizing damage.

**Maintain and Stabilize:**  
Automated assessments compare current security baselines against post-attack conditions, and workflows ensure systems are restored to a secure state. Continuous feedback from monitoring informs planning and redesign, keeping the cycle ongoing.

**Integration with Green Pace DevOps Process**  
Green Pace’s existing DevOps infrastructure can be enhanced by embedding security automation tools at each stage of this DevSecOps lifecycle, ensuring the security standards are not an afterthought but a foundational part of the development process. For example:

* Integrate static and dynamic security scanning into CI/CD pipelines to automate compliance checks during build and test.
* Automate vulnerability scanning and patch management during pre-production phases.
* Use automated compliance and configuration validation during deployment.
* Employ continuous monitoring tools to enforce runtime security policies.
* Automate incident detection and response workflows to minimize human error and speed remediation.

By embedding these automated security controls throughout the DevSecOps cycle, Green Pace will ensure consistent enforcement and compliance with the policy standards, reduce security risks, and improve overall software quality.

### 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 | Possible | Low | High | 1 |
| STD-002-CPP | Medium | Possible | Low | Medium | 3 |
| STD-003-CPP | High | Possible | Low | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Possible | Low | High | 2 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | High | Possible | Low | High | 4 |
| STD-008-CPP | High | Likely | Low | High | 5 |
| STD-009-CPP | Medium | Possible | Low | Medium | 3 |
| STD-010-CPP | High | Likely | Mediu | High | 5 |
| [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 physical media such as hard drives, databases, and backups by encrypting it so unauthorized users cannot access the information without the proper key. This policy requires that all sensitive or regulated data stored on any device or storage system be encrypted using strong, industry-standard algorithms like AES-256. Encryption at rest is essential to safeguard data from breaches due to lost or stolen devices, unauthorized physical access, or insider threats. It must be applied whenever data is saved or archived, including on file servers, databases, backups, and cloud storage, ensuring confidentiality and compliance with data protection laws. |
| Encryption in flight | **Encryption in Flight** secures data as it travels across networks by encrypting communications, preventing interception or tampering. This policy mandates the use of secure communication protocols such as TLS 1.2 or higher and SSH to encrypt all data transmitted between clients, servers, and third parties. Encrypting data in flight protects against man-in-the-middle attacks, eavesdropping, and data manipulation during transmission. This encryption must be enforced on all internal and external network communications, including APIs, web applications, VPNs, and wireless connections, to maintain data privacy and integrity. |
| Encryption in use | **Encryption in Use** focuses on protecting data while it is actively being processed in memory or CPU, shielding it from unauthorized access during computation. This policy requires the implementation of technologies like Trusted Execution Environments (TEEs), homomorphic encryption, or memory encryption features to secure sensitive data in use. Encryption in use mitigates risks such as memory scraping, cold boot attacks, and malicious software exposure. It should be applied especially when processing highly sensitive information like cryptographic keys, passwords, or personal data, particularly in multi-tenant or cloud environments where runtime memory may be vulnerable. |

| 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 users, systems, or devices before allowing access. This policy requires strong authentication methods such as multi-factor authentication (MFA), robust password policies, or biometric verification to ensure that only authorized individuals gain access. Effective authentication prevents unauthorized logins and impersonation. It must be implemented for all user logins, system access points, and APIs to maintain secure access control. |
| Authorization | **Authorization** determines what authenticated users are permitted to do by restricting access based on their roles and privileges. This policy mandates the use of role-based access control (RBAC) or attribute-based access control (ABAC) to limit user access strictly according to their job functions and needs. Proper authorization prevents unauthorized changes, data exposure, and privilege escalation. It applies to all access involving sensitive files, databases, system configurations, user management, and any resources requiring controlled permissions. |
| Accounting | **Accounting** (or auditing) involves tracking and logging user activities, system changes, and resource access to create a detailed audit trail. This policy requires thorough logging of events such as user logins, file and database access, configuration changes, and user account modifications. Accounting supports forensic investigations, compliance audits, and helps detect suspicious or unauthorized behavior. It must be continuously enforced for all users and administrative operations, especially those involving database changes, user additions or removals, and access to sensitive information. |

**\***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

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 | 07/19/2025 | Module 3 edits | Lilly Berry | [Insert text.] |
| 3.0 | 08/08/2025 | Project one edits | Lilly berry | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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