Green Pace

Green Pace Secure Development Policy

Contents

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.

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	All external input should be treated as untrusted, including user input, file data, and network data. Input must be validated for type, length, format, and acceptable ranges as early as possible. Proper validation helps prevent common vulnerabilities such as buffer overflows, injection attacks, and data corruption before malicious input can reach application logic (Carnegie Mellon University [CMU], n.d.).
2. Heed Compiler Warnings	Compiler warnings often indicate unsafe operations, undefined behavior, or logic errors that can lead to security vulnerabilities. Developers should enable strict compiler warning levels and resolve all warnings instead of suppressing them. Treating warnings as errors ensures that potential defects are identified and corrected early in the development process (SEI, 2016).
3. Architect and Design for Security Policies	Security requirements must be addressed during system architecture and design rather than added later. Designing with security in mind ensures consistent enforcement of authentication, authorization, logging, and data protection across the application. A well-defined security architecture reduces the risk of gaps caused by inconsistent or ad hoc security decisions (SEI, 2016).
4. Keep It Simple	Complex designs increase the likelihood of implementation errors and hidden vulnerabilities. Simple, well-structured code is easier to understand, review, test, and secure. Reducing unnecessary complexity improves maintainability and lowers the risk of security flaws introduced by misunderstanding or misuse of code components (CMU, n.d.).
5. Default Deny	Systems should deny access by default and explicitly grant permissions only when required. This principle limits exposure when configurations are incorrect or incomplete. Default deny reduces the attack surface by ensuring that unintended access is not allowed unless explicitly permitted (SEI, 2016).
6. Adhere to the	Users, processes, and software components should operate with the minimum privileges

Principles	Write a short paragraph explaining each of the 10 principles of security.
Principle of Least Privilege	necessary to perform their tasks. Limiting privileges reduces the impact of a successful attack and prevents unauthorized access to sensitive resources. Least privilege is a foundational principle for reducing overall system risk (SEI, 2016).
7. Sanitize Data Sent to Other Systems	Data that is safe within one system may be dangerous when interpreted by another system, such as databases, command shells, or log files. Output should be sanitized or encoded according to its destination context to prevent injection attacks and unintended command execution (CMU, n.d.).
8. Practice Defense in Depth	Defense in depth involves implementing multiple layers of security controls so that the failure of one control does not compromise the entire system. Combining input validation, secure APIs, access control, monitoring, and secure configuration provides stronger protection than relying on a single safeguard (SEI, 2016).
9. Use Effective Quality Assurance Techniques	Security-focused quality assurance includes code reviews, static analysis, testing boundary conditions, and regression testing of known vulnerabilities. These practices help identify defects early and ensure that security fixes remain effective over time. QA is a critical component of maintaining secure software (CMU, n.d.).
10. Adopt a Secure Coding Standard	A secure coding standard provides consistent rules and guidance that reduce common classes of vulnerabilities. Following a recognized standard such as the SEI CERT C++ Coding Standard promotes uniform secure practices across development teams and improves code quality and security outcomes (SEI, 2016).

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	Incorrect or implicit data type conversions can introduce truncation, sign errors, and undefined behavior that undermine program correctness and security. Selecting appropriate data types and validating values before conversion helps prevent logic errors and memory vulnerabilities caused by unsafe type handling (Software Engineering Institute [SEI], 2016).

Noncompliant Code

Converts a negative value to an unsigned type, producing a large positive number that can be used as a size or loop bound.

```
#include <iostream>

int main() {
   int count = -1;
   unsigned int ucount = count; // Problem: -1 becomes a very large value
   std::cout << ucount << "\n";
}
```

Compliant Code

Validates signed input before conversion and uses explicit casting only after confirming the value is safe.

```
#include <iostream>
#include <stdexcept>

int main() {
   int count = -1;

   if (count < 0) {
      throw std::invalid_argument("count must be nonnegative");
   }
   unsigned int ucount = static_cast<unsigned int>(count);
   std::cout << ucount << "\n";
}
```

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

Principles(s): Validate Input Data; Use Effective Quality Assurance Techniques; Adopt a Secure Coding Standard

This standard enforces the validation of input values before performing type conversions, ensuring that data remains within safe and expected ranges. By validating signed values prior to conversion, the risk of unintended truncation, sign errors, and undefined behavior is reduced. Static analysis, compiler warnings, and code reviews support this standard by identifying unsafe or implicit conversions early in the development process. Adhering to the SEI CERT C++ Coding Standard ensures consistent handling of data types across the codebase and reduces the likelihood of exploitable logic or memory vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Low	High	4

Justification:
Improper data type conversions can result in logic errors, memory misuse, or unsafe loop bounds, potentially leading to denial-of-service or memory corruption vulnerabilities. These issues are relatively common in C and C++ codebases but are typically inexpensive to remediate once detected, making early detection and prioritization critical.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C4244, C4267	Detects potential data loss and unsafe implicit type conversions
Cppcheck	2.x	invalidConversion	Flags unsafe or suspicious type conversions
Clang-Tidy	15.x	cppcoreguidelines-narrowing-conversions	Identifies narrowing and lossy conversions
SonarQube	9.x	cpp:S3519	Detects unsafe type casts and conversion logic issues

Automation Rationale:
These tools are integrated into the development and verification stages of the DevSecOps pipeline to automatically identify unsafe data type conversions during compilation and static analysis. Automated detection ensures consistent enforcement of this standard and prevents risky conversions from reaching production environments.

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	Unchecked arithmetic operations can result in integer overflow or underflow, which may lead to incorrect memory allocation sizes, buffer overflows, or logic bypasses. Validating arithmetic bounds before performing calculations ensures data integrity and reduces the likelihood of exploitable conditions (SEI, 2016).

Noncompliant Code

Adds 1 to an unsigned max value and wraps to 0, breaking logic and potentially causing unsafe behavior.

```
#include <limits>
#include <iostream>

int main() {
    unsigned int x = std::numeric_limits<unsigned int>::max();
    x = x + 1; // Wraps to 0
    std::cout << x << "\n";
}
```

Compliant Code

Checks for overflow before performing the addition and fails safely if it would wrap.

```
#include <limits>
#include <iostream>
#include <stdexcept>

int main() {
    unsigned int x = std::numeric_limits<unsigned int>::max();

    if (x == std::numeric_limits<unsigned int>::max()) {
        throw std::overflow_error("addition would overflow");
    }
    x = x + 1;
    std::cout << x << "\n";
}
```

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

Principles(s): Validate Input Data; Heed Compiler Warnings; Use Effective Quality Assurance Techniques

This standard supports input validation by requiring that arithmetic operations be checked for overflow and underflow conditions before execution. By validating numeric boundaries, developers prevent unexpected wraparound behavior that could result in incorrect logic, unsafe memory allocations, or bypassed security checks. Compiler warnings, static analysis, and targeted testing help identify unsafe arithmetic operations early in the development lifecycle, reinforcing consistent and secure handling of numeric data values.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Low	High	4

Justification:
Unchecked integer overflow and underflow can lead to incorrect calculations, unsafe buffer sizes, and logic errors that attackers may exploit to cause denial-of-service conditions or memory corruption. These vulnerabilities are common in low-level languages such as C and C++ but are typically inexpensive to remediate once identified, making proactive detection and prioritization essential.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C26451, C26472	Detects arithmetic overflow and unsafe integer operations
Cppcheck	2.x	integerOverflow	Identifies potential integer overflow and wraparound conditions
Clang-Tidy	15.x	cppcoreguidelines-narrowing-conversions	Flags unsafe narrowing and arithmetic conversions
SonarQube	9.x	cpp:S3518	Detects integer overflow risks in arithmetic expressions

Automation Rationale:
These tools are executed during the development and verification stages of the DevSecOps pipeline to automatically detect unsafe arithmetic operations before code is merged or released. Automated analysis ensures consistent enforcement of numeric safety standards and prevents integer overflow vulnerabilities from propagating into production systems.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	Improper handling of strings, including missing null terminators and unchecked copy operations, can cause out-of-bounds reads and memory corruption. Enforcing explicit bounds checks and guaranteed string termination prevents common vulnerabilities associated with unsafe string manipulation (Carnegie Mellon University [CMU], n.d.).

Noncompliant Code

strncpy may leave the destination without a null terminator if the source is long, creating an out-of-bounds read risk.

```
#include <cstring>
#include <cstdio>

int main() {
   char dest[8];
   const char* src = "TOO-LONG-STRING";
   std::strncpy(dest, src, sizeof(dest)); // May not null-terminate
   std::printf("%s\n", dest);           // Undefined behavior risk
}
```

Compliant Code

Copies at most size-1 bytes and explicitly null-terminates the destination buffer.

```
#include <cstring>
#include <cstdio>

int main() {
   char dest[8];
   const char* src = "TOO-LONG-STRING";

   std::strncpy(dest, src, sizeof(dest) - 1);
   dest[sizeof(dest) - 1] = '\0';
   std::printf("%s\n", dest);
}
```

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

Principles(s): Validate Input Data; Keep It Simple; Use Effective Quality Assurance Techniques

This standard supports input validation by requiring strict bounds checking and guaranteed string termination when copying or formatting strings. Limiting copy operations to valid buffer sizes and explicitly adding a null terminator prevents out-of-bounds reads and memory corruption. Keeping string handling simple and consistent reduces implementation errors, while quality assurance techniques such as static analysis and targeted test cases help detect unsafe string operations early and prevent regressions over time.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Medium	High	4

Justification:
Improper string handling can cause out-of-bounds reads, memory corruption, and crashes, and in some cases may be leveraged for code execution depending on context. These issues are common in C and C++ when unsafe library functions or incorrect bounds are used. Fixes may require refactoring to safer APIs or redesigning string handling logic, making remediation cost slightly higher than simple arithmetic issues.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C6054, C6385	Detects potential lack of null termination and buffer overrun risks
Cppcheck	2.x	bufferAccessOutOfBounds	Flags potential out-of-bounds access and unsafe string usage
Clang-Tidy	15.x	cert-str34-c	Identifies risky string handling patterns and missing termination safeguards
SonarQube	9.x	cpp:S3519	Detects unsafe buffer and string manipulation patterns

Automation Rationale:
These tools run during the create and verify stages of the DevSecOps pipeline to detect unsafe string operations, missing null terminators, and out-of-bounds access risks before code is promoted. Automated detection strengthens enforcement of string correctness standards and reduces the likelihood that memory-corrupting defects reach production.

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP	Constructing SQL queries using string concatenation allows untrusted input to alter query logic, leading to unauthorized data access or modification. Using parameterized queries separates code from data, ensuring user input is treated strictly as data and preventing SQL injection attacks (SEI, 2016).

Noncompliant Code

Directly concatenates user input into a SQL statement, allowing injection.

```
#include <string>

// Example only: do not use this pattern
std::string buildQuery(const std::string& username) {
   return "SELECT * FROM Users WHERE username = '" + username + "'";
}
```

Compliant Code

Uses placeholders and binds values so the database treats input strictly as data.

```
#include <string>

// Pseudocode pattern: placeholders + bind parameters
struct PreparedStatement {
   void bind(int index, const std::string& value);
   void execute();
};

void safeQuery(PreparedStatement& stmt, const std::string& username) {
   // SQL: SELECT * FROM Users WHERE username = ?
   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; Sanitize Data Sent to Other Systems; Practice Defense in Depth

This standard supports secure handling of untrusted input by preventing user-provided data from being interpreted as executable SQL code. Parameterized queries separate code from data, ensuring that input values are treated strictly as data even if they contain SQL control characters. This aligns with sanitizing data sent to other systems because the database is a separate interpreter with its own execution context. Defense in depth is achieved by combining parameterized queries with additional controls such as least privilege database accounts, input validation, and logging to reduce impact if a single control fails.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	5

Justification:
SQL injection can lead to unauthorized data access, data modification, account takeover, and destruction of records, making impact severe for any system that processes sensitive information. The likelihood is high because injection is a common attack technique when applications build queries using string concatenation. Remediation often requires updating query patterns across the codebase and implementing consistent database access practices, which increases cost compared to isolated code fixes.

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.x	cpp:S3649	Detects SQL query construction patterns that risk injection
Semgrep	1.x	sql-injection-pattern	Flags unsafe string concatenation used to build SQL queries
CodeQL	Latest	SQL Injection Query	Performs dataflow analysis to detect user-controlled input reaching query execution
Visual Studio Static Code Analysis	2022	Security Analysis (taint patterns)	Helps identify risky data flow patterns and insecure input usage in code

Automation Rationale:
These tools are executed in the verify stage of the DevSecOps pipeline to detect injection risks through pattern matching and dataflow analysis before code is merged or released. Automated detection enforces consistent query safety practices and reduces the chance that unsafe SQL construction reaches production environments.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CPP	Manual memory management using raw pointers increases the risk of memory leaks, dangling pointers, and use-after-free vulnerabilities. Applying resource acquisition is initialization (RAII) principles and smart pointers ensures predictable object lifetimes and improves memory safety (SEI, 2016).

Noncompliant Code

Returns early without freeing allocated memory, causing a leak.

```
#include <stdexcept>

int doWork(bool fail) {
    int* p = new int(42);
    if (fail) {
        return -1; // Memory leak: p not deleted
    }
    delete p;
    return 0;
}
```

Compliant Code

Uses std::unique_ptr so memory is automatically released even on early returns or exceptions.

```
#include <memory>

int doWork(bool fail) {
    auto p = std::make_unique<int>(42);
    if (fail) {
        return -1; // No leak: unique_ptr cleans up
    }
    return 0;
}
```

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

Principles(s): Adhere to the Principle of Least Privilege; Practice Defense in Depth; Adopt a Secure Coding Standard

This standard strengthens memory safety by reducing reliance on manual memory management, which is a common source of memory leaks, dangling pointers, and use-after-free vulnerabilities. Using RAII and C++ smart pointers enforces predictable object lifetimes and ensures memory is released automatically, even when errors or early returns occur. Defense in depth is supported by combining safer language features (smart pointers, containers) with automated analysis and testing to detect memory misuse. Adopting a secure coding standard ensures consistent application of memory-safe patterns across the codebase.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	5

Justification:
Memory management defects such as leaks, use-after-free, and dangling pointers can lead to crashes, denial-of-service, and in some cases arbitrary code execution. These issues are common in C and C++ applications that use raw pointers and manual allocation. Remediation may require refactoring ownership and lifetimes across functions and modules, which increases cost, but it is critical due to the high security impact.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C6011, C6386	Detects potential null dereference and buffer misuse patterns
Cppcheck	2.x	memleak, nullPointer	Identifies possible memory leaks and null pointer dereferences
Clang-Tidy	15.x	modernize-use-nullptr, cppcoreguidelines-owning-memory	Flags unsafe ownership patterns and encourages safer constructs
SonarQube	9.x	cpp:S3584	Detects potential memory leaks and resource lifetime issues

Automation Rationale:
These tools are integrated into the create and verify stages of the DevSecOps pipeline to detect memory safety issues early, including ownership errors, leaks, and unsafe pointer usage. Automated detection ensures consistent enforcement of RAII-based practices and helps prevent memory-related vulnerabilities from reaching production systems.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	STD-006-CPP	Assertions may be disabled in production builds and should not be relied upon to enforce security-critical behavior. Placing side effects or required logic inside assertions can cause inconsistent behavior between debug and release builds, leading to reliability and security issues (CMU, n.d.).

Noncompliant Code

Has a side effect inside the assert; behavior changes if asserts are disabled.

```
#include <cassert>

bool checkAndAdvance(int& i) {
   return ++i > 0;
}

int main() {
   int i = 0;
   assert(checkAndAdvance(i)); // Side effect inside assert
}
```

Compliant Code

Performs side-effect work outside the assertion, and asserts only the condition.

```
#include <cassert>

bool checkAndAdvance(int& i) {
   return ++i > 0;
}

int main() {
   int i = 0;
   bool ok = checkAndAdvance(i);
   assert(ok); // No side effect inside assert
}
```

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

Principles(s): Keep It Simple; Use Effective Quality Assurance Techniques; Architect and Design for Security Policies

This standard supports secure development by ensuring assertions are used only for developer diagnostics and never for security-critical logic. Because assertions may be disabled in release builds, placing required checks or side effects inside assertions can cause production behavior to differ from debug behavior. Keeping logic outside of assertions simplifies the control flow and reduces the risk of hidden, build-dependent behavior. Quality assurance practices such as code reviews, static analysis, and test coverage help confirm that essential validation and security checks are implemented using enforceable runtime controls rather than optional debug-only assertions.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Medium	Low	Medium	3

Justification:
Misuse of assertions can lead to missing validation or inconsistent behavior between debug and release builds, which can cause reliability issues and weaken security controls if developers incorrectly rely on assertions for enforcement. While this is less likely to directly enable code execution compared to memory corruption defects, it can contribute to bypassed checks and unstable runtime behavior. Fixes are generally low cost because they typically involve moving logic out of assertions and replacing them with explicit runtime validation.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C6287, C6001	Helps detect suspicious conditional logic and uninitialized or inconsistent usage patterns
Cppcheck	2.x	assertWithSideEffect	Flags assertions that contain side effects or risky expressions
Clang-Tidy	15.x	cert-dcl03-c, bugprone-assert-side-effect	Detects side effects in assertions and build-dependent behavior
SonarQube	9.x	cpp:S2694	Flags assertions that should not be used for enforcing runtime behavior

Automation Rationale:
These tools are executed during the create and verify stages of the DevSecOps pipeline to identify assertions that contain side effects or are incorrectly used as enforcement mechanisms. Automated detection supports consistent enforcement by ensuring required validation occurs in runtime logic rather than in debug-only checks.

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Allowing exceptions to escape destructors or failing to design for exception safety can result in unexpected program termination or resource leaks. Ensuring destructors do not throw exceptions and using exception-safe patterns improves reliability and prevents denial-of-service conditions (SEI, 2016).

Noncompliant Code

Destructor throws an exception, which can terminate the program if another exception is active.

```
#include <stdexcept>

struct Bad {
   ~Bad() { throw std::runtime_error("fail"); } // Dangerous
};

int main() {
   Bad b;
}
```

Compliant Code

Destructor is non-throwing and handles errors internally instead of throwing.

```
#include <iostream>

struct Good {
   ~Good() noexcept {
      try {
         // cleanup that might fail
      } catch (...) {
         // log or handle, but do not throw
         std::cerr << "cleanup failed\n";
      }
   }
};

int main() {
   Good g;
}
```

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

Principles(s): Use Effective Quality Assurance Techniques; Keep It Simple; Practice Defense in Depth

This standard supports secure and reliable error handling by requiring exception-safe patterns and ensuring destructors do not throw exceptions. Throwing exceptions from destructors can cause unexpected program termination, especially if another exception is already active. Keeping exception handling simple and consistent reduces the likelihood of missed cleanup, resource leaks, or unstable control flow. Defense in depth is supported by combining exception safety with logging, monitoring, and controlled failure behavior so that errors are detected and handled without exposing sensitive information or causing unnecessary outages.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Medium	High	4

Justification:
Poor exception handling can trigger denial-of-service conditions through unexpected termination, resource leaks, or inconsistent state. A common high-risk scenario is allowing exceptions to escape destructors or relying on broad catch-all handlers that suppress failures without logging or corrective action. While not every exception issue becomes directly exploitable, failure behavior and error hiding can prevent detection of real security incidents and make recovery difficult, increasing operational and audit risk.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C26447, C26448	Detects exception-safety and noexcept related issues and risky throwing behavior
Cppcheck	2.x	exceptThrowInDestructor	Flags exceptions thrown in destructors and risky exception patterns
Clang-Tidy	15.x	cert-err58-cpp	Identifies improper exception handling patterns and unsafe exception flow
SonarQube	9.x	cpp:S3981	Flags catch-all handlers and patterns that hide or mishandle errors

Automation Rationale:
These tools run during the create and verify stages of the DevSecOps pipeline to detect exception handling problems such as throwing destructors, unsafe exception propagation, and catch-all handlers that may hide errors. Automated detection enforces consistent exception safety and encourages logging and controlled failure behavior so that exceptions do not silently bypass detection or lead to production outages.

Coding Standard 8

Coding Standard	Label	Name of Standard
Concurrency	STD-008-CPP	Concurrent access to shared resources without proper synchronization can cause data races and undefined behavior. Protecting shared state with mutexes or atomic operations ensures consistent behavior and prevents corruption of critical program data (CMU, n.d.).

Noncompliant Code

Two threads can read and write counter concurrently without synchronization, causing a data race.

```
#include <thread>

int counter = 0;

void inc() {
   for (int i = 0; i < 100000; i++) {
      counter++; // Data race
   }
}

int main() {
   std::thread t1(inc);
   std::thread t2(inc);
   t1.join();
   t2.join();
}
```

Compliant Code

Uses a mutex to protect updates so operations are serialized safely.

```
#include <thread>
#include <mutex>

int counter = 0;
std::mutex m;

void inc() {
   for (int i = 0; i < 100000; i++) {
      std::lock_guard<std::mutex> lock(m);
      counter++;
   }
```

Compliant Code

```
}

int main() {
   std::thread t1(inc);
   std::thread t2(inc);
   t1.join();
   t2.join();
}
```

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

Principles(s): Practice Defense in Depth; Use Effective Quality Assurance Techniques; Keep It Simple

This standard supports defense in depth by preventing data races and undefined behavior caused by unsynchronized concurrent access to shared resources. Protecting shared state with mutexes, locks, or atomic operations ensures consistent behavior and reduces the risk of corrupted data, crashes, or security-relevant logic failures. Keeping concurrency patterns simple and well-defined makes code easier to review and reduces the likelihood of subtle synchronization bugs. Quality assurance techniques such as static analysis, code reviews, stress testing, and concurrency-focused test cases help detect race conditions and deadlocks before deployment.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	High	High	5

Justification:
Concurrency defects can cause unpredictable behavior, crashes, data corruption, and intermittent failures that are difficult to reproduce and diagnose. Data races and deadlocks can create denial-of-service conditions and may undermine security controls if corrupted state affects authorization decisions or critical logic paths. Remediation cost is often high because fixes may require redesigning shared-state ownership, refactoring thread interactions, and adding synchronization while preserving performance and correctness.

Automation

Tool	Version	Checker	Description Tool
Clang-Tidy	15.x	concurrency-mt-unsafe	Flags thread-unsafe access patterns and unsafe concurrency usage
SonarQube	9.x	cpp:S2236	Detects synchronization issues and unsafe concurrent access patterns
Visual Studio Static Code Analysis	2022	Concurrency Checks	Identifies patterns associated with race conditions and thread-unsafe usage
Cppcheck	2.x	raceCondition	Flags suspicious shared-state access patterns that may lead to races

Automation Rationale:
These tools are applied during the verify stage of the DevSecOps pipeline to detect unsafe concurrent access and synchronization issues before code is promoted to pre-production. Automated detection improves consistency and

reduces the likelihood that race conditions or deadlocks reach production, supporting defense in depth through early identification and remediation.

Coding Standard 9

Coding Standard	Label	Name of Standard
Input/Output Safety	STD-009-CPP	Using untrusted input as a format string in output functions can expose memory contents or crash the program. Always using constant format strings and passing external data as arguments prevents format string vulnerabilities and improves output safety (SEI, 2016).

Noncompliant Code

Uses user input directly as the format string, enabling format string attacks.

```
#include <cstdio>
#include <string>

void logLine(const std::string& user) {
   std::printf(user.c_str()); // Dangerous: user controls format
}
```

Compliant Code

Uses a constant format string and prints user input as data.

```
#include <cstdio>
#include <string>

void logLine(const std::string& user) {
   std::printf("%s", user.c_str());
}
```

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

Principles(s): Validate Input Data; Sanitize Data Sent to Other Systems; Practice Defense in Depth

This standard enforces safe handling of input and output operations by ensuring that untrusted data is never interpreted as executable format instructions. Treating external input strictly as data and using constant format strings prevents attackers from injecting format specifiers that could expose memory contents or crash the application. Sanitizing data before output protects downstream systems and logging mechanisms that interpret formatted input. Defense in depth is achieved by combining safe API usage with input validation, logging controls, and automated analysis to reduce the impact of misuse at multiple layers.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Low	High	4

Justification:
Format string vulnerabilities can allow attackers to read arbitrary memory, corrupt program state, or cause application crashes, resulting in denial-of-service conditions. These vulnerabilities commonly occur when developers pass user-controlled input directly as a format string to output functions. Remediation is typically low cost because fixes involve using constant format strings and safe output patterns, making early detection and enforcement highly effective.

Automation

Tool	Version	Checker	Description Tool
Visual Studio Static Code Analysis	2022	C6262, C6287	Detects unsafe use of format strings and suspicious output patterns
Cppcheck	2.x	invalidPrintfArgType	Flags incorrect or unsafe printf-style usage
Clang-Tidy	15.x	cert-fio30-c	Identifies unsafe formatted I/O operations and format string misuse
SonarQube	9.x	cpp:S3519	Detects format string vulnerabilities and unsafe output handling

Automation Rationale:
These tools run during the create and verify stages of the DevSecOps pipeline to identify unsafe formatted I/O usage and prevent user-controlled input from being interpreted as executable format data. Automated detection ensures consistent enforcement of safe output practices and reduces the likelihood that format string vulnerabilities reach production environments.

Coding Standard 10

Coding Standard	Label	Name of Standard
Secure Randomness	STD-010-CPP	Standard pseudo-random number generators such as rand() are predictable and unsuitable for security-sensitive operations. Using stronger randomness mechanisms ensures that generated values cannot be easily guessed or reproduced by an attacker (CMU, n.d.).

Noncompliant Code

Uses rand() to generate a value that might be used as a token, which is predictable.

```
#include <cstdlib>

int insecureToken() {
   return std::rand(); // Not secure for tokens
}
```

Compliant Code

Uses C++ random facilities as a safer baseline for non-crypto needs; for true security tokens, use an approved CSPRNG per org policy.

```
#include <random>

int betterRandom() {
   std::random_device rd;
   std::mt19937 gen(rd());
   std::uniform_int_distribution<int> dist(0, 1000000);
   return dist(gen);
}
```

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

Principles(s): Adopt a Secure Coding Standard; Validate Input Data; Practice Defense in Depth

This standard ensures that random values used for security-sensitive purposes such as tokens, identifiers, or session values are not predictable or reproducible by an attacker. Standard pseudo-random generators such as rand() produce deterministic sequences that are unsuitable for security use. By enforcing stronger randomness mechanisms and approved cryptographic random number generators where required, this standard reduces the risk of guessable values. Defense in depth is supported by combining secure randomness with additional controls such as expiration, validation, and monitoring to limit the impact of compromised or reused values.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Medium	Low	Medium	3

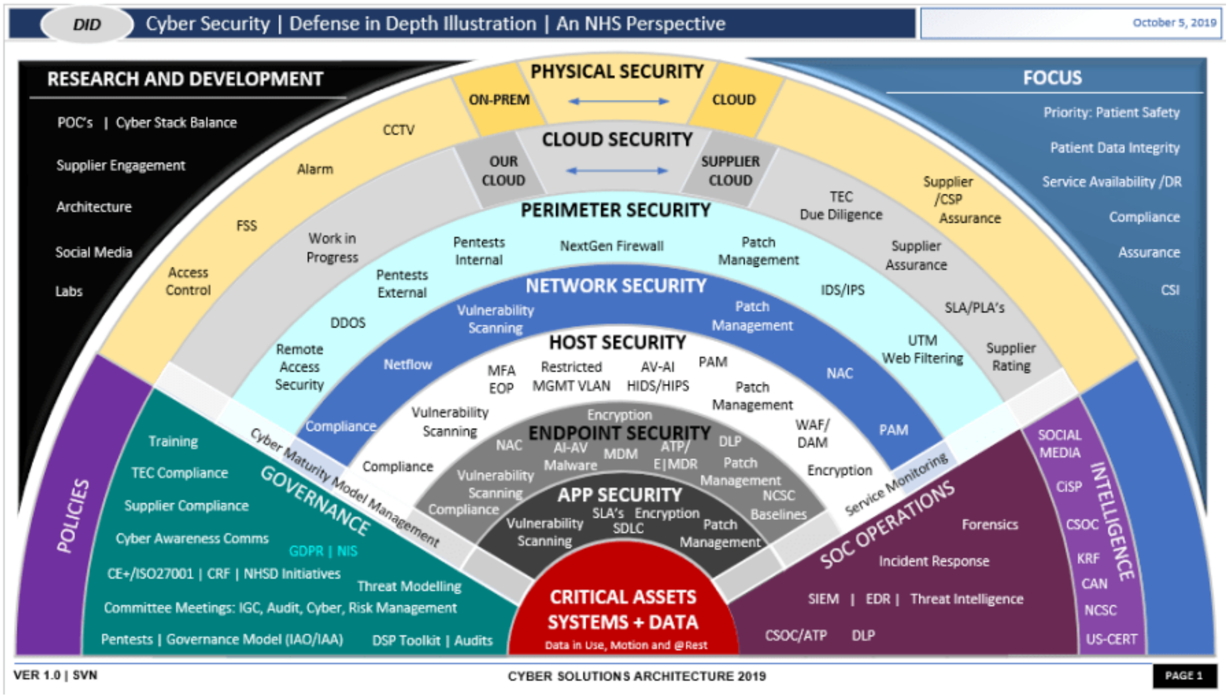
Justification:
Predictable random values can undermine authentication, authorization, and session management mechanisms by allowing attackers to guess tokens or identifiers. While misuse of weak randomness does not always result in immediate exploitation, it significantly weakens security controls and can contribute to broader compromise when combined with other vulnerabilities. Remediation is typically low cost because it involves replacing insecure random functions with approved alternatives.

Automation

Tool	Version	Checker	Description Tool
Clang-Tidy	15.x	cert-msc30-c	Flags use of weak or predictable random number generators
SonarQube	9.x	cpp:S2245	Detects use of weak cryptographic or pseudo-random functions
Cppcheck	2.x	insecureRandomness	Identifies use of non-cryptographic randomness in sensitive contexts
Semgrep	1.x	insecure-random	Flags insecure random number usage patterns

Automation Rationale:
These tools are applied during the create and verify stages of the DevSecOps pipeline to detect use of weak or predictable randomness in security-sensitive contexts. Automated enforcement ensures consistent adoption of approved randomness mechanisms and reduces the likelihood that insecure token generation reaches production systems.

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

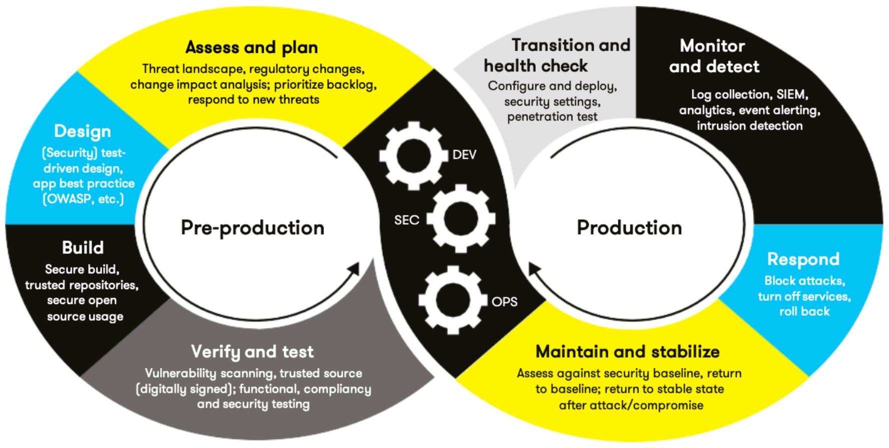
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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


Assess and plan
Threat landscape, regulatory changes, change impact analysis; prioritize backlog, respond to new threats
Design
(Security) test-driven design, app best practice (OWASP, etc.)
Build
Secure build, trusted repositories, secure open source usage
Verify and test
Vulnerability scanning, trusted source (digitally signed); functional, compliancy and security testing
Pre-production
DEV
SEC
OPS
Transition and health check
Configure and deploy, security settings, penetration test
Monitor and detect
Log collection, SIEM, analytics, event alerting, intrusion detection
Production
Respond
Block attacks, turn off services, roll back
Maintain and stabilize
Assess against security baseline, return to baseline; return to stable state after attack/compromise

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.

Automation (DevSecOps Pipeline)

- Green Pace will enforce this security policy through automated controls integrated into the existing DevOps pipeline to create a DevSecOps process that supports defense in depth. Automation begins in the Plan stage by addressing security technical debt, tracking security metrics, and performing threat modeling for new features and changes. During this stage, teams define acceptance criteria for secure coding standards, identify high-risk components such as database access and memory management, and ensure developers have training on approved secure libraries, secure coding patterns, and policy expectations.
- Automation continues in the Create stage by using IDE-integrated security tooling and compiler settings that promote secure coding by default. Developers will enable strict compiler warning levels and treat warnings as errors when feasible, ensuring issues such as narrowing conversions, suspicious casts, and unsafe constructs are detected early. Developers also run automated linting and static checks locally to prevent high-risk defects such as integer overflow, unsafe string handling, memory misuse, and exception safety problems from being introduced into the codebase.
- In the Verify stage, Green Pace will use automated analysis as a gate for pull requests and builds. Static application security testing and code quality scanning will run on every commit and pull request to detect standards violations and security vulnerabilities. Tools such as static analyzers and rule-based scanners will be configured to flag issues relevant to this policy, including unsafe type conversions, integer overflow risks, unsafe string operations, SQL query construction using concatenation, use-after-free patterns, and incorrect use of assertions and exceptions. When findings occur, the pipeline will require remediation or documented exception approval before changes can be merged.
- In Pre-Production, Green Pace will apply additional automated testing to validate security behavior in environments that closely mirror production. This includes integration testing and security-focused test scenarios such as input fuzzing for boundary cases and malformed data, along with targeted stress testing for concurrency issues. These activities help identify defects that are difficult

to detect through static analysis alone, such as race conditions, deadlocks, and edge-case failures in error handling.

- During Release, Green Pace will protect the integrity of the software supply chain through automated software signing and release verification controls. Release artifacts will be signed, and signature verification will be required as part of deployment to ensure only approved builds are promoted. In the Prevent stage, runtime integrity checks and configuration controls will reduce the likelihood of tampering and enforce baseline security expectations in production environments.
- Finally, Green Pace will use automation in the Detect and Respond stages to support rapid identification and containment of threats. Security monitoring, alerting, and logging will be used to detect abnormal activity and policy violations across systems. When an issue is detected, orchestration and response playbooks will guide containment actions, such as blocking malicious inputs, increasing monitoring, and applying compensating controls. The Adapt stage ensures continuous improvement by feeding incident lessons learned back into security technical debt tracking, updating tooling rules, refining secure coding standards, and improving incident response procedures. This continuous feedback loop ensures that security remains intrinsic to development and is continuously enforced rather than addressed only after vulnerabilities are discovered.

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	Medium	Low	High	4
STD-002-CPP	High	Medium	Low	High	4
STD-003-CPP	High	Medium	Medium	High	4
STD-004-CPP	High	High	Medium	High	5
STD-005-CPP	High	High	Medium	High	5
STD-006-CPP	Medium	Medium	Low	Medium	3
STD-007-CPP	High	Medium	Medium	High	4
STD-008-CPP	High	Medium	High	High	5
STD-009-CPP	High	Medium	Low	High	4
STD-010-CPP	Medium	Medium	Low	Medium	3

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***.***

a. Explain each type of encryption, how it is used, and why and when the policy applies.
b. 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.

a. Encryption	Explain what it is and how and why the policy applies.
Encryption at rest	Encryption at rest protects sensitive data stored on disk, including databases, configuration files,

a. Encryption	Explain what it is and how and why the policy applies.
	backups, and log files. Green Pace requires encryption at rest for all production systems that store sensitive or proprietary data to prevent unauthorized access if storage media is lost, stolen, or improperly accessed. Approved encryption mechanisms must use industry-standard algorithms and key management practices. This policy applies whenever data is persisted beyond runtime and supports defense in depth by ensuring data confidentiality even if system perimeter controls are bypassed.
Encryption in flight	Encryption in flight protects data transmitted between systems, services, and users over networks. Green Pace requires encryption in flight for all network communications that transmit sensitive data, including user authentication, database connections, API calls, and administrative access. Transport-level encryption prevents interception, eavesdropping, and man-in-the-middle attacks. This policy applies whenever data crosses a network boundary and ensures integrity and confidentiality during transmission.
Encryption in use	Encryption in use protects data while it is actively processed in memory or within execution environments. Green Pace applies encryption in use where supported by the platform, such as secure enclaves or protected memory regions, to reduce the risk of data exposure during runtime attacks. This policy is especially relevant for high-value data such as credentials, cryptographic keys, and sensitive computations. Encryption in use complements encryption at rest and in flight by extending protection throughout the full data lifecycle.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Authentication verifies the identity of users, services, and systems before access is granted. Green Pace requires strong authentication mechanisms for all user logins, service-to-service communications, and administrative access. Authentication controls ensure that only authorized identities can access systems and resources. This policy applies during initial access attempts and supports accountability by tying actions to verified identities.
Authorization	Authorization determines what authenticated users or systems are allowed to do once access is granted. Green Pace enforces role-based access control and least-privilege principles to restrict access to files, databases, system functions, and administrative actions. Authorization policies apply whenever users access protected resources or attempt to modify system state, ensuring that access levels align with business roles and security requirements.
Accounting	Accounting provides traceability and accountability by recording user and system activity. Green Pace requires logging of security-relevant events such as user logins, database changes, creation or removal of users, access level changes, and file access. Accounting supports threat detection, forensic analysis, and audit compliance by maintaining accurate records of system activity. Logs must be protected from unauthorized modification and retained according to organizational 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.

- Each coding standard defined in this policy is supported by one or more of the ten core security principles. For example, standards addressing data type correctness, integer overflow, string safety, and input/output safety directly support the principle of validating input data by preventing unsafe values from entering application logic. Memory protection, concurrency, and exception handling standards reinforce defense in depth by reducing the likelihood that a single failure results in system compromise.
- Standards related to SQL injection, secure randomness, and output sanitization align with the principle of sanitizing data sent to other systems by ensuring that untrusted input is treated strictly as data. The use of automated detection tools and continuous testing supports effective quality assurance techniques and reinforces adherence to a secure coding standard. Together, these mappings demonstrate that Green Pace's security policy is grounded in widely accepted security principles and implemented consistently across development and operations.

Map the Principles Table

Coding Standard	Principle Number(s)	Principle Name(s)	Justification (How the Principle Supports the Standard)
STD-001-CPP (Data Type)	1, 9, 10	Validate Input Data; Use Effective Quality Assurance Techniques; Adopt a Secure Coding Standard	Validating values before conversion prevents unsafe casts and unexpected sign or truncation behavior. QA techniques such as static analysis and code review detect risky conversions early. A secure coding standard ensures consistent safe type usage across the codebase.
STD-002-CPP (Data Value)	1, 2, 9	Validate Input Data; Heed Compiler Warnings; Use Effective Quality Assurance Techniques	Input validation and bounds checking prevent overflow and underflow from corrupting program logic or memory sizes. Compiler warnings help identify risky arithmetic and conversions. QA techniques catch edge cases through analysis and testing before release.
STD-003-CPP (String Correctness)	1, 4, 9	Validate Input Data; Keep It Simple; Use Effective Quality Assurance Techniques	Validating string length and enforcing explicit bounds prevents out-of-bounds reads and memory corruption. Simple, consistent string handling reduces implementation errors. QA techniques and static analysis detect unsafe string operations and prevent regressions.

Coding Standard	Principle Number(s)	Principle Name(s)	Justification (How the Principle Supports the Standard)
STD-004-CPP (SQL Injection)	1, 7, 8	Validate Input Data; Sanitize Data Sent to Other Systems; Practice Defense in Depth	Input is treated as untrusted and must not be interpreted as SQL logic. Parameterized queries ensure data is safely handled by the database system. Defense in depth is supported through layered controls such as least privilege database access, validation, and logging.
STD-005-CPP (Memory Protection)	6, 8, 10	Adhere to the Principle of Least Privilege; Practice Defense in Depth; Adopt a Secure Coding Standard	Least privilege limits the impact of memory-related failures by reducing what compromised code can access. Defense in depth is strengthened by using RAII and smart pointers in addition to analysis tools. A secure coding standard enforces consistent safe memory ownership patterns.
STD-006-CPP (Assertions)	3, 4, 9	Architect and Design for Security Policies; Keep It Simple; Use Effective Quality Assurance Techniques	Security enforcement must be designed into runtime logic rather than optional debug-only checks. Keeping assertion usage simple avoids hidden side effects. QA techniques ensure required validation remains present and testable in release builds.
STD-007-CPP (Exceptions)	4, 8, 9	Keep It Simple; Practice Defense in Depth; Use Effective Quality Assurance Techniques	Simple exception patterns reduce unstable control flow and missed cleanup. Defense in depth is supported by controlled failure behavior, logging, and preventing termination due to throwing destructors. QA techniques validate exception safety and prevent error hiding through catch-all misuse.
STD-008-CPP (Concurrency)	4, 8, 9	Keep It Simple; Practice Defense in Depth; Use Effective Quality Assurance Techniques	Simple concurrency patterns reduce synchronization mistakes that cause races or deadlocks. Defense in depth is supported by locking, atomic operations, and safe ownership to prevent corruption of shared state. QA techniques such as static analysis and stress testing help detect concurrency defects early.
STD-009-CPP (Input/Output Safety)	1, 7, 8	Validate Input Data; Sanitize Data Sent to Other Systems; Practice Defense in Depth	External input must not be interpreted as formatting instructions. Sanitizing output prevents downstream systems from executing or misinterpreting data. Defense in depth is achieved by combining safe output APIs, validation, and automated checks to reduce exploitation risk.
STD-010-CPP (Secure Randomness)	1, 8, 10	Validate Input Data; Practice Defense in Depth; Adopt a Secure Coding Standard	Security-sensitive values must not be guessable, so weak randomness must be avoided. Defense in depth is supported by combining secure randomness with additional controls such as expiration, validation, and monitoring. A secure coding standard ensures approved randomness mechanisms are used consistently.

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	
1.1	02/04/2026	Completed Project One security policy updates	Cortney Messinger	CISO

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV

Citations

- Carnegie Mellon University. (n.d.). *CERT C++ coding standard*. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/spaces/cplusplus/overview
- Carnegie Mellon University. (n.d.). *CERT C coding standard*. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/display/c
- Software Engineering Institute. (2016). *The CERT C++ coding standard* (2nd ed.). Carnegie Mellon University. https://wiki.sei.cmu.edu/confluence/spaces/cplusplus/overview