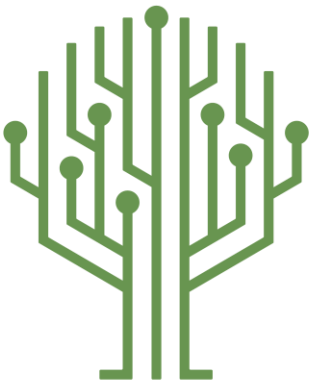


Green Pace Secure Development Policy

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	Validate input from all untrusted data sources. Input validation prevents injection attacks, buffer overflows, and other vulnerabilities by ensuring only expected, properly formatted data is processed.
2. Heed Compiler Warnings	Compile code with the highest warning level available for your compiler and treat warnings as errors where possible. Resolve all warnings promptly, as they often indicate potential security issues, undefined behavior, or subtle bugs that can be exploited.
3. Architect and Design for Security Policies	Incorporate security considerations into the architecture and design phases from the beginning. Threat modeling, secure design patterns and system structure are the baseline to support security requirements rather than a last resort or afterthought.
4. Keep It Simple	Minimize complex code, design, and security mechanisms. Simpler code is easier to understand, review, test, and maintain. This reduces the possibility of introducing or overlooking vulnerabilities.
5. Default Deny	Deny access or functionality by default unless explicitly allowed. This principle enforces least privilege and reduces vulnerability by requiring positive authorization.
6. Adhere to the Principle of Least Privilege	Grant only the minimum permissions, access rights, or capabilities necessary for a component, user, or process to perform its function. Effectively limiting damage if an element is compromised.
7. Sanitize Data Sent to Other Systems	Sanitize data by escape or neutralization before using it in contexts where data could be interpreted as code or commands. This complements input validation to prevent injection attacks.
8. Practice Defense in Depth	Implement multiple layers of security controls so that if one fails, others still provide protection (i.e. input validation + default deny + secure design patterns + data sanitization).

Principles	Write a short paragraph explaining each of the 10 principles of security.
9. Use Effective Quality Assurance Techniques	Static code analysis, dynamic testing, code reviews, and integration testing help identify and remediate vulnerabilities early ensuring code quality and security before deployment.
10. Adopt a Secure Coding Standard	The following secure coding standard consistently throughout all levels of development ensures uniform practices securing the product against common vulnerabilities and promotes secure coding habits. For use cases not covered in the following Ten Coding Standards adhering to SEI CERT C++ as additional guidance provides comprehensive coverage for consistent rules and guidelines.

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	Smallest Semantic Type
Data Type	[STD-001-CPP]	Select the data type that most clearly expresses the intended purpose, range, signedness, and encoding while using the smallest size that safely accommodates all possible values. This improves code readability, enhances portability, reduces memory footprint where appropriate, and prevents subtle vulnerabilities.

Noncompliant Code

Using int for a small non-negative counter, 0-200, wastes space and hides the limited range.

```
int counter = 0;
counter += parsed_value;
```

Compliant Code

Use std::uint8_t for small unsigned values, 0-255, clearly documenting the range

```
std::uint8_t counter = 0;
counter += static_cast<std::uint8_t>(parsed_value & 0xFF);
```

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

Principles(s): 3: Architect and Design for Security Policies, 4: Keep It Simple, 6: Adhere to the Principle of Least Privilege – Using the smallest appropriate type reduces attack surface, simplifies reasoning about ranges, and enforces minimal resource exposure in design.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Medium	Low	High	3

Automation

Tool	Version	Checker	Description Tool
Clang Static Analyzer	17+	Clang-tidy readability-simplify-boolean-expr + misc-misleading-*	Flags oversized types and suggests narrower alternatives
Cppchecktext	2.14+	unusedVariable + outOfBounds	Detects misuse of oversized types leading to waste or overflow
SonarQube	10+	Cpp:S101 + CERT rule mappings	Enforces minimal type usage per CERT guidelines

Coding Standard 2

Coding Standard	Label	Use Valid Values Within Expected Ranges
Data Value	[STD-002-CPP]	Ensure the actual value fits safely within the type's range and the program's expected limits. Invalid values cause overflows, crashes, logic errors, and security issues. Check and validate values early, especially from untrusted sources to keep behavior predictable and secure. This makes code readable and prevents vulnerabilities like buffer overflow.

Noncompliant Code

Using a fixed-size character array with cin allows buffer overflow if the user enters more characters than the array can hold, a classic security vulnerability.

```
char input[20];
Std::cin >> input;
```

Compliant Code

Use std::cin.getline() with a fixed-size character array to safely limit input to the buffer size preventing buffer overflow. This enforces an upper bound on input length makint it a secure alternative to unbounded cin. (NOTE: remember to clear any flags and remaining buffer for efficient input handling.)

```
char input[20];
std::cin.getline(input, sizeof(input));

// example for NOTE
if (std::cin.fail())
{
 // clear overflow and flags
 std::cin.clear();
 std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
```

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

Principles(s): 1: Validate Input Data, 2: Heed Compiler Warnings, 8: Sanitize Data Sent to Other Systems – Range validation directly enforces input checks and prevents overflow/underflow from propagating.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Low	High	4

Automation

Tool	Version	Checker	Description Tool
Clang	17+	-fsanitize=undefined + bounds	Catches range violations at runtime/compile
Cppcheck	2.14+	arrayIndexOutOfBounds	Detects values exceeding array/type
Coverity	2024	INTEGER_OVERFLOW	Identifies unchecked ranges leading to overflow

Coding Standard 3

Coding Standard	Label	Use Safe String Functions and Bounds Checking
String Correctness	[STD-003-CPP]	C-style string functions like strcpy and strcat do not check buffer sizes, leading to buffer overflow vulnerabilities. Prefer bounded alternatives or std::string.

Noncompliant Code

Strcpy can overflow destination buffer.

```
char dest[10];
strcpy(dest, user_input);
```

Compliant Code

Use strncpy with explicit size.

```
char dest[10];
strncpy(dest, user_input, sizeof(dest)-1);// exclude termination bit
dest[sizeof(dest)-1] = '\0'; // for terminal bit
```

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

Principles(s): 1: Validate Input Data, 8: Sanitize Data Sent to Other Systems, 9: Practice Defense in Depth – Bounds-checked string functions add a layer of protection beyond validation.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Low	High	4

Automation

Tool	Version	Checker	Description Tool
Cppcheck	2.14+	bufferAccessOutOfBounds	Flags unsafe string copies
Clang Static Analyzer	17+	Security.insecureAPI.DeprecatedOrUnsafeBufferHandling	Detects strcpy/strcat usage
SonarQube	10+	Cpp:S4721	Recommends bounded alternatives

Coding Standard 4

Coding Standard	Label	Use Parameterized Queries or Prepared Statements for Database Access
SQL Injection	[STD-004-CPP]	Concatenating user-supplied input directly into SQL query strings allows attackers to inject malicious SQL code. Parameterized queries separate the SQL structure from the data values, ensuring input is treated as literal data rather than code. This is the primary defense against SQL injection (per OWASP, SEI CERT guidelines, and CWE-89). Always use prepared statements/bind parameters when interfacing with databases from C++.

Noncompliant Code

Direct string concatenation of user input into SQL query allows classic SQL injection. An attacker entering ‘ Or 1=1 ‘— as username would make the query always true exposing a serious SQL injection vulnerability

```
std::string username = get_user_input();
std::string query = "SELECT * FROM users WHERE username = '" +
username + "'";
```

Compliant Code

Use prepared statements with parameter placeholders and sqlite3_bind_* functions to bind user input safely. This effectively escapes input and malicious input is treated as a literal string value, not part of the SQL logic, completely preventing injection.

```
std::string username = get_user_input();
Const char* sql_select = "SELECT * FROM users WHERE username = ?";

sqlite3_stmt* statement = nullptr;
int returnCode = sqlite3_prepare_v2(db, sql_select, -1, &statement,
nullptr);

if (returnCode != SQLITE_OK)
{
    std::cerr << "Prepare failed: " << sqlite3_errmsg(db) <<
std::endl;
    return;  // Or throw/handle as needed
}

sqlite3_bind_text(statement, 1, username.c_str(), -1,
SQLITE_TRANSIENT);

returnCode = sqlite3_step(statement);

if (returnCode == SQLITE_ROW)
{
```

Compliant Code

```
        // Process result safely
    }
    sqlite3_finalize(stmt); // clean up
```

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

Principles(s): 1: Validate Input Data, 8: Sanitize Data Sent to Other Systems, 9: Practice Defense in Depth – Parameterized queries are a core defense against injection.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Medium	High	5

Automation

Tool	Version	Checker	Description Tool
SonarQube	10+	Cpp:S2077	Detects dynamic SQL concatenation
Coverity	2024	SQL_INJECTION	Flags string-built queries
Cppcheck	2.14+	unusedFunction with custom rules	Can flag unsafe concatenation patterns

Coding Standard 5

Coding Standard	Label	Avoid Raw Pointers; Use Smart Pointers
Memory Protection	[STD-005-CPP]	Raw pointers often lead to leaks, double-frees, and use-after-free. Smart pointers (std::unique_ptr, std::shared_ptr) enforce Resource Acquisition Is Initialization and ownership.

Noncompliant Code

Manual (new and delete) are prone to leaks or mismatch.

```
Widget* w = new Widget();
delete w;
```

Compliant Code

Use std::unique_ptr for exclusive ownership

```
auto w = std::make_unique<Widget>(); // automatically deleted once
                                     // out of scope
```

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

Principles(s): 4: Keep It Simple, 5: Default Deny, 9: Practice Defense in Depth, 10: Use Effective Quality Assurance Techniques – Smart pointers simplify ownership and reduce memory errors.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Medium	High	4

Automation

Tool	Version	Checker	Description Tool
Clang Static Analyzer	17+	Core.uninitialized + cplusplus.NewDelete	Detects raw pointer leaks/double-free
Coverity	2024	RESOURCE_LEAK	Flags manual new/delete mismatches
Cppcheck	2.14+	Memleak	Identifies missing deletes

Coding Standard 6

Coding Standard	Label	Use Assertions for Testing and Quality Assurance, Not for Release Build Logic
Assertions	[STD-006-CPP]	Assertions are valuable for documenting developer assumptions, and catching logic errors during development and testing. They trigger a short diagnostic when violated. However, assertions are typically disabled in release builds so they cannot be relied upon for runtime security, error recovery, input validation, or any logic which must execute in production. This exposes logic errors to security vulnerabilities.

Noncompliant Code

Assertion used for runtime input validation which is disabled in the production release build allows invalid data to proceed through application silently.

```
void process_data(int value)
{
    Assert(value > 0 && value <= 100);
    int result = 100 / value;
}
```

Compliant Code

Use runtime exception for critical checks that must always run.

```
void process_data(int value)
{
    If (value <=0 || value > 100) // if check for production
    {
        throw std::invalid_argument("msg");
    }
    assert(value > 0 && value <= 100); // assert for dev validation
                                       // test
    int result = 100 / value;
}
```

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

Principles(s): 2: Heed Compiler Warnings, 9: Use Effective Quality Assurance Techniques, 10: Adopt a Secure Coding Standard – Assertions aid QA but must not replace runtime checks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Medium	Low	Medium	2

Automation

Tool	Version	Checker	Description Tool
Clang	17+	-Wassert-unused-result	Warns on unused assert results
SonarQube	10+	Cpp:S5956	Flags assertions used for runtime logic

Coding Standard 7

Coding Standard	Label	Make Every Throw Recoverable
Exceptions	[STD-007-CPP]	Every thrown exception must be designed for easy, local recovery with minimal code. Use specific exception types, Resource Acquisition Is Initialization for cleanup, and nearby try-catch blocks where practical. This ensures all exceptions can be handled gracefully.

Noncompliant Code

Throwing a very generic exception type with a vague message makes it hard for catchers to distinguish this error from others, leading to broad catch blocks or unhandled exceptions.

```
void validate_user_input(const std::string& input)
{
    if (input.empty())
    {
        throw std::runtime_error("Error");
    }
    // ...
}
```

Compliant Code

Throw a specific, narrow exception type with a clear, non-sensitive message making it easier to catch and recover gracefully

```
class EmptyInputError : public std::invalid_argument
{
public:
    EmptyInputError() : std::invalid_argument("Input cannot be
empty") {}
};

void validate_user_input(const std::string& input)
{
    if (input.empty())
    {
        throw EmptyInputError();
    }
    // ...
}
```

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

Principles(s): 4: Keep It Simple, 9: Practice Defense in Depth – Specific exceptions enable granular, recoverable handling.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Medium	Medium	Medium	3

Automation

Tool	Version	Checker	Description Tool
Clang Static Analyzer	17+	Security.throw-keyword-missing	Detects broad catch blocks
Cppcheck	2.14+	exceptThrowInNoexceptFunction	Flags poor exception design

Coding Standard 8

Coding Standard	Label	Prevent Numerical Overflow/Underflow in Arithmetic
Expressions	[STD-008-CPP]	Arithmetic operations on both integer and floating-point types can produce results outside the representable range of the type, leading to overflow, underflow , undefined behavior, wraparound, or special floating-point values. These issues can cause incorrect calculations, security vulnerabilities, denial-of-service, or silent data corruption. Always validate operands before operations, use wider types for intermediates, check results, or employ checked/safe arithmetic where possible.

Noncompliant Code

No pre-checks are performed before addition. For signed types this causes undefined behavior on overflow/underflow. For unsigned types it silently wraps around. Both can lead to severe logic errors or security issues.

```
template <typename T>
T add_numbers(T const& start, T const& increment, unsigned long int
const& steps)
{
    T result = start;
    for (unsigned long int i = 0; i < steps; ++i)
    {
        result += increment;  // Blind addition → signed UB or
unsigned wraparound possible
    }
    return result;
}
```

Compliant Code

This templated function safely performs a single addition of an increment to a starting value by first checking for potential overflow or underflow using the type's numeric limits. If the operation would exceed the representable range, it throws a detailed std::overflow_error or std::underflow_error containing the current values, increment, type name, and boundary limits; otherwise, it executes the addition and returns the result.

```
template <typename T>
T add_numbers(T const& start, T const& increment)
{
    T result = start;

    // Check for underflow when adding a negative number
    if (increment < 0 && result < std::numeric_limits<T>::lowest() -
increment)
    {
```

Compliant Code

```
            // Create underflow error string
            std::ostringstream uError;
            uError << "\n\t***Adding " << +increment << " to " <<
    +result << " exceeds " << typeid(T).name()
                << " lowest value of: " <<
    +std::numeric_limits<T>::lowest() << "***";

            // Throw underflow exception
            throw std::underflow_error(uError.str());
        }
        // Check for overflow when adding a positive number
        else if (increment > 0 && result > std::numeric_limits<T>::max()
    - increment)
        {
            // Create overflow error string
            std::ostringstream oError;
            oError << "\n\t***Adding " << +increment << " to " <<
    +result << " exceeds " << typeid(T).name()
                << " max value of: " << +std::numeric_limits<T>::max()
    << "***";

            // Throw overflow exception
            throw std::overflow_error(oError.str());
        }

        // add increment to result
        result += increment;

        return result;

    }
```

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

Principles(s): 1: Validate Input Data, 3: Architect and Design for Security Policies, 9: Practice Defense in Depth – Checked arithmetic prevents overflow exploits.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Medium	High	4

Automation

Tool	Version	Checker	Description Tool
Clang	17+	-fsanitize=signed-integer-overflow	Runtime overflow detection
Covertiy	2024	INTEGER_OVERFLOW	Static overflow checks
Cppcheck	2.14+	integerOverflow	Flags unchecked arithmetic

Coding Standard 9

Coding Standard	Label	Always Pair Resource Acquisition with Release (RAII)
Memory: Resource Management	[STD-009-CPP]	Manual resource management (files, locks, sockets) leads to leaks on exceptions or early returns. RAII classes ensure automatic cleanup.

Noncompliant Code

File not closed on exception

```
FILE* f = fopen("file.txt", "r");
// … exception thrown here
fclose(f); // never reached
```

Compliant Code

Use RAII wrapper

```
std::ifstream f("file.txt");
// automatically closed at end of scope.
```

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

Principles(s): 4; Keep It Simple, 5: Default Deny, 10: Use Effective Quality Assurance Techniques – RAII enforces deterministic cleanup.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Low	High	4

Automation

Tool	Version	Checker	Description Tool
Clang Static Analyzer	17+	Core.StackAddressEscape + core.CallAndMessage	Detects resource leaks
Coverity	2024	RESOURCE_LEAK	Flags missing closes

Coding Standard 10

Coding Standard	Label	Handle All Exceptions
Exceptions	[STD-010-CPP]	Unhandled exceptions lead to abrupt program termination, resource leaks, undefined behavior, or insecure state exposure. Every thrown exception must be caught at an appropriate level. This ensures predictable failure modes, prevents denial-of-service or information disclosure from crashes.

Noncompliant Code

An exception propagates unhandled to the top of the program, causing abrupt termination without cleanup or logging.

```
void risky_operation()
{
    throw std::runtime_error("Critical failure");
}

int main()
{
    risky_operation();   // Unhandled → program aborts
    std::cout << "This line is never reached" << std::endl;
    return 0;
}
```

Compliant Code

All exceptions are caught at the top level (main()) with specific handlers first, then a catch-all, ensuring graceful failure, secure logging, and proper exit code.

```
void risky_operation()
{
    throw std::runtime_error("Critical failure");
}

int main()
{
    try
    {
        risky_operation();
        std::cout << "Success" << std::endl;
    }
    catch (const std::runtime_error& e)
    {
        std::cerr << "Runtime error: " << e.what() << std::endl;
```

Compliant Code

```
            return 1;
        }
        catch (const std::exception& e)
        {
            std::cerr << "Standard exception: " << e.what() <<
    std::endl;
            return 1;
        }
        catch (...)
        {
            std::cerr << "Unknown exception caught – shutting down
    safely" << std::endl;
            return 1;
        }
    }
```

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

Principles(s): 9: Practice Defense in Depth, 10: Use Effective Quality Assurance Techniques – Handling all exceptions prevents crashes and leaks.

Threat Level

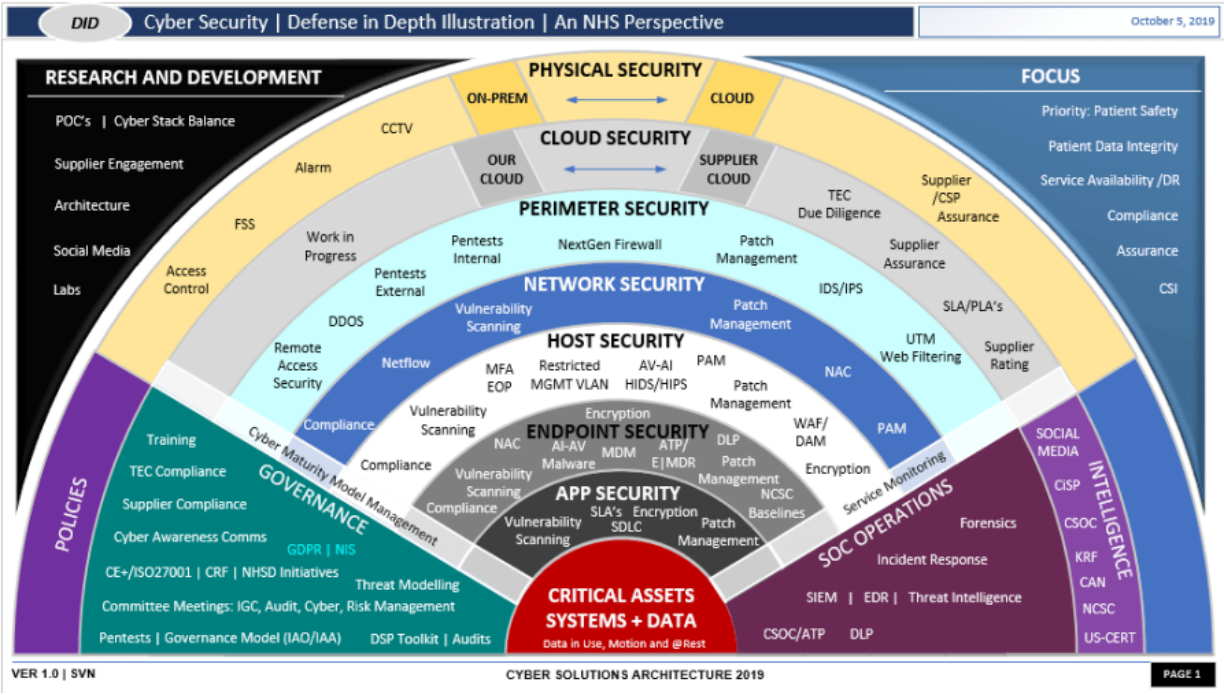
Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Medium	High	4

Automation

Tool	Version	Checker	Description Tool
Clang	17+	-Wunreachable-code + analyzer	Flags unhandled paths
SonarQube	10+	cpp:S00112	Recommends catching std::exception

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

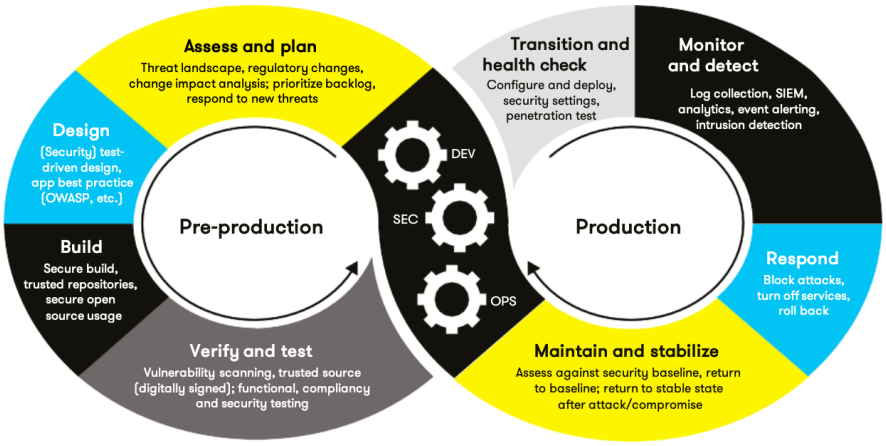
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

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. To automate enforcement, integrate static analysis tools (Clang Static Analyzer, Cppcheck, SonarQube with CERT rules, Coverity) into the Build and Verify phases of the DevSecOps toolchain. Run automated scans during code commit and in the CI pipeline to catch violations early. Dynamic checks, like -fsanitize, and fuzzing can occur in Pre-production and Production phases. Use the Monitoring and Analytics gear to feed results into dashboards for adaptive feedback loops. This shifts security left while supporting defense-in-depth across the toolchain.

Summary of Risk Assessments

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	Medium	Medium	Low	High	3
STD-002-CPP	High	High	Low	High	4
STD-003-CPP	High	High	Low	High	4
STD-004-CPP	High	High	Medium	High	5
STD-005-CPP	High	High	Medium	High	4
STD-006-CPP	Medium	Medium	Low	Medium	2
STD-007-CPP	Medium	Medium	Medium	Medium	3
STD-008-CPP	High	High	Medium	High	4
STD-009-CPP	High	High	Low	High	4
STD-010-CPP	High	Medium	Medium	High	4

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	Protects inactive stored data (databases, files, backups) using AES-256 or better. Applies to all persistent storage to prevent unauthorized access if physical/media is compromised. Mandatory for sensitive data.
Encryption in flight	Secures data during transmission using TLS 1.3 with strong ciphers. Applies to all network communication (API calls, web, internal) to prevent interception, MITM, and tampering. Mandatory for all external and sensitive internal traffic.
Encryption in use	Protects data during active processing (memory, cache) using techniques like homomorphic encryption or confidential computing. Applies when sensitive data must remain encrypted during computation to guard against memory scraping or privileged access. Required for high-sensitivity workloads.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Verifies user identity (multi-factor, strong passwords, SSO). Applies to all logins, API access, and sessions to ensure only legitimate users gain entry. Mandatory for all access points.
Authorization	Enforces what authenticated users can do (role-based access control, least privilege, attribute-based access control). Applies to database changes, file access, and user management to limit damage from compromised accounts. Mandatory across all systems.
Accounting	Logs auditable events (logins, DB changes, file access, privilege escalations). Applies to provide forensic evidence, detect anomalies, and support compliance. Mandatory with retention per policy and integration with existing OS/firewall/anti-malware logs.

*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.

- STD-001-CPP: Principles 3, 4, 6 – Narrow types support secure-by-design, simplicity, and least privilege.
- STD-002-CPP: Principles 1, 2, 8 – Range validation directly implements input validation and sanitization.
- STD-003-CPP: Principles 1, 8, 9 – Bounds checking adds defense layers against string errors.
- STD-004-CPP: Principles 1, 8, 9 – Parameterized queries are a primary injection defense.
- STD-005-CPP: Principles 4, 5, 9, 10 – Smart pointers simplify memory handling and reduce defects.
- STD-006-CPP: Principles 2, 9, 10 – Assertions support QA but require runtime alternatives.
- STD-007-CPP: Principles 4, 9 – Specific exceptions enable clean, layered recovery.
- STD-008-CPP: Principles 1, 3, 9 – Checked arithmetic prevents overflow in design.
- STD-009-CPP: Principles 4, 5, 10 – RAII enforces reliable resource management.
- STD-010-CPP: Principles 9, 10 – Full exception handling prevents insecure crashes.

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	01/24/2026	Core Security Principles and Coding standards revision	Joshua Shoemaker	[Insert text.]
1.2	02/12/2026	Completed Document	Joshua Shoemaker	[Insert text.]

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV