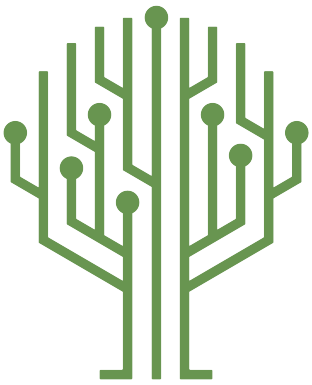


Green Pace

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	Involves checking and sanitizing input data from users or external systems to prevent malicious data from causing harm. Proper input validation can thwart many common attacks such as SQL injection, cross-site scripting (XSS), and buffer overflows by ensuring that only appropriately formatted data is processed and executed.
2. Heed Compiler Warnings	Compiler warnings are indicators of potential issues in the code that could lead to security vulnerabilities. Ignoring these warnings can result in overlooked flaws that attackers might exploit. By paying close attention to and resolving compiler warnings, developers can address weak spots in the code, enhancing the overall robustness and security of the application.
3. Architect and Design for Security Policies	Security should be integrated into the architecture and design phases of software development. This means creating a secure foundation by incorporating security policies, principles, and best practices from the beginning. Designing with security in mind helps ensure that security measures are not just add-ons but are intrinsic to the system, reducing the risk of vulnerabilities and simplifying future security updates.
4. Keep It Simple	Complex systems are more prone to errors and harder to secure because they have more potential points of failure and are more difficult to understand and manage. By keeping systems simple, developers can reduce the attack surface, make it easier to spot and fix security issues, and enhance maintainability.
5. Default Deny	The default deny principle asserts that access should be denied by default and only granted when explicitly allowed. This minimizes the risk of unauthorized access by ensuring that no one can access resources without explicit permission. Implementing a default deny policy requires careful configuration of access

Principles	Write a short paragraph explaining each of the 10 principles of security.
	controls and permissions, ensuring that only necessary and intended access is provided, which significantly enhances security.
6. Adhere to the Principle of Least Privilege	The principle of least privilege dictates that users, programs, and systems should only have the minimum level of access necessary to perform their functions. By restricting access rights, the potential damage from accidental or malicious actions is minimized. This approach reduces the risk of unauthorized access to sensitive information and critical systems, helping to contain security breaches and limit their impact.
7. Sanitize Data Sent to Other Systems	This principle dictates that developers should clean and validate data before it is transmitted to other systems to ensure it does not contain malicious content that could exploit vulnerabilities in the receiving system. Proper sanitization helps prevent attacks such as injection attacks and cross-site scripting by ensuring that only safe, expected data is shared between systems.
8. Practice Defense in Depth	Defense in depth is a layered security strategy that involves implementing multiple protective measures to guard against threats. This approach ensures that if one layer of defense is breached, additional layers are in place to continue protecting the system. By combining firewalls, intrusion detection systems, encryption, access controls, and other security measures, organizations can create a more resilient security posture that is harder for attackers to penetrate.
9. Use Effective Quality Assurance Techniques	Effective Quality assurance techniques such as code reviews, automated testing, and vulnerability scanning help ensure that software is free of critical bugs and security vulnerabilities before deployment. By incorporating these techniques throughout the development lifecycle, developers can catch and address issues early, reducing the likelihood of security breaches in the final product.
10. Adopt a Secure Coding Standard	Secure coding standards provide guidelines and best practices for writing code that is resistant to security vulnerabilities. Adopting these standards helps developers consistently implement security measures in their code, reducing the risk of common vulnerabilities like buffer overflows, injection flaws, and improper error handling. By adhering to a secure coding standard, organizations can improve the overall security of their software and make it easier to maintain and audit for security compliance.

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]	***Updated 08/07/2024 – Revision 2*** Use standard C++ primitive types such as int, float, double, char, and bool, instead of their C-style counterparts. This ensures clarity and compatibility with modern C++ features, avoiding confusion and potential bugs associated with C-style types. Additionally, avoiding C-style types helps prevent type mismatches and implicit conversions that can lead to security vulnerabilities, such as buffer overflows that can be exploited by attackers to corrupt memory, execute dangerous code, or crash the program.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code uses unsigned int and float without explicit “f” suffix, which might lead to implicit type conversion issues and less readable code. Implicit conversions can create unexpected results, especially in expressions involving mixed types, potentially leading to security vulnerabilities.

```
unsigned int age = 30;
float salary = 45000.50;
```

Compliant Code

Updated 08/07/2024 – Revision 2

This code uses standard C++ primitive types with explicit suffixes for better readability and avoids implicit type conversions. By using explicit types and suffixes, the code is less prone to errors related to type conversions, enhancing security by reducing the risk of undefined behavior and potential exploits.

```
int age = 30;
double salary = 45000.50;
float height = 5.9f;
```

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

Principles(s):

- Heed Compiler Warnings
- Adopt a Secure Coding Standard

Heeding compiler warnings principle is connected to this security standard because it can detect potential type-related issues, such as implicit conversions when using standard C++ primitive types.

Adopt a secure coding standard principle maps this standard because it ensures that developers consistently use modern C++ features, reducing the risk of errors and improving code clarity and security.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Low	Medium	3

Automation

Tool	Version	Checker	Description Tool
Clang-Tidy	15.0	MODERNIZE-USE-DEFAULT-MEMBER-INIT	This checker ensures the use of modern C++ constructs, including default initialization for class members.
MISRA C++ Compliance Checker	Various	MISRA C++:2008 RULE 3-9-2	Ensures that appropriate types are used according to MISRA C++ guidelines.
Cppcheck	2.11	TYPE	Detects potential issues related to type usage and mismatches in C++.
Klocwork	2023.1	UNSIGNED_TYPES	Analyzes and ensures the correct use of signed and unsigned types in C++ code.

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	***Updated 08/07/2024 – Revision 2*** Use default values when declaring variables: this ensures that variables are initialized properly, preventing undefined behavior and improving code safety. Uninitialized variables can be a source of serious security vulnerabilities. Attackers can exploit these variables to read residual data from memory, leading to information leakage, or to inject malicious data that could alter program execution. For example, an uninitialized pointer could point to an arbitrary memory location, which an attacker could use to execute arbitrary code or cause a denial-of-service attack. By initializing variables with default values, developers ensure that variables always have predictable, safe starting values, reducing the attack surface.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code declares variables without initializing them. If these variables are used before being assigned values, they will contain arbitrary, potentially dangerous data, leading to undefined behavior or security vulnerabilities such as memory leaks or data corruption.

```
int count;
double price;
```

Compliant Code

Updated 08/07/2024 – Revision 2

This code initializes variables with default values, ensuring they have well-defined values when first used. This reduces the risk of security vulnerabilities related to the use of uninitialized memory, improving the program stability and security.

```
int count = 0;
double price = 0.0;
```

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

Principles(s):

- Validate Input Data
- Adopt a Secure Coding Standard

Validate input data principle maps this standard because validating input data and initializing variables ensures that the data being handled is within expected ranges, preventing undefined behavior.

Adopt a secure coding standard because it reinforces this practice, ensuring consistency and reducing the likelihood of uninitialized variables leading to security vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Low	Medium	4

Automation

Tool	Version	Checker	Description Tool
PVS-Studio	7.19	V730	Detects uninitialized variables and ensures that variables have default values.
SonarQube	9.9 LTS	CPP:S1003	Checks for uninitialized variables and ensures proper default initialization.
Klocwork	2023.1	UNINIT.CTR	Detects cases where variables are used without being initialized.
Parasoft C/C++test	2023.1	CERT DCL00-C	Ensures that all variables are initialized before use, aligning with CERT C guidelines.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	***Updated 08/07/2024 – Revision 3*** Guaranteeing sufficient storage for strings is critical for preventing buffer overflows, a common and severe security vulnerability. Buffer overflows occur when data is copied to a buffer that isn't large enough to hold it, leading to potential overwriting of adjacent memory. This can result in program crashes, data corruption, or even arbitrary code execution if exploited by attackers. While C-style strings require careful handling to ensure sufficient space for both data and the null terminator, using std::string from the C++ standard library eliminates many of these risks by automatically managing memory and ensuring that string operations are safe and bounded.

Noncompliant Code

Updated 08/07/2024 – Revision 2

In functionOne(), inputting more than 11 characters can cause a buffer overflow, as the buffer does not have enough space to accommodate the extra characters and the null terminator.

In functionTwo(), using “strcat” on “str1” and “str2” without ensuring sufficient buffer size can result in a buffer overflow, potentially leading to memory corruption.

```
void functionOne() {
  char buffer[12];
  std::cin >> buffer;
}
```

```
void functionTwo() {
  char str1[50] = "Hello";
  char str2[50] = "World";
  strcat(str1, str2);
}
```

Compliant Code

Updated 08/07/2024 – Revision 2

In functionOne(), using std::string instead of a fixed-size char array eliminates the risk of buffer overflow by dynamically adjusting the size of the string to fit the input.

Compliant Code

In functionTwo(), the std::string class is used for concatenation, ensuring safe and efficient string handling without the risk of exceeding buffer limits, thus preventing potential security vulnerabilities associated with buffer overflows.

```
std::string stringOne, stringTwo;
std::cin >> stringOne >> stringTwo;
```

```
std::string str1 = "Hello";
std::string str2 = "World";
std::string result = str1 + str2;
```

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
- Adopt a Secure Coding Standard

Validate input data principle maps this standard because it helps to avoid buffer overflows when working with strings.

Sanitizing data sent to other systems because it ensures that strings passed between systems do not introduce vulnerabilities.

Adopt a secure coding standard because it guides the use of safer string handling techniques, such as using std::string instead of C-style strings.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	5

Automation

Tool	Version	Checker	Description Tool
Coverity	2023.06	BUFFER_OVERFLOW	Detects potential buffer overflows and unsafe string operations.

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO	Detects when there is not space for null terminator. Detects Buffer overrun. Detects Type overrun.
Parasoft C/C++test	2023.1	CERT_CPP-STR50-b	Avoid overflow due to reading a not zero terminated string.
		CERT_CPP-STR50-c CERT_CPP-STR50-e CERT_CPP-STR50-f CERT_CPP-STR50-g	Avoid overflow when writing to a buffer. Prevent buffer overflows from tainted data. Avoid buffer write overflow from tainted data. Do not use the 'char' buffer to store input from 'std::cin'
Polyspace Bug Finder	R2024a	CERT C++: STR50-CPP	Checks for: • Use of dangerous standard function • Missing null in string array • Buffer overflow from incorrect string format specifier • Destination buffer overflow in string manipulation • Insufficient destination buffer size Rule partially covered.

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	***Updated 08/07/2024 – Revision 2*** Use parameterized queries and prepared statements to ensure that user input is treated as data, not executable code, by safely escaping potentially dangerous characters and preventing malicious input from altering the structure of the SQL query. If user input is directly concatenated into SQL queries, attackers can inject SQL code that alters the behavior of the query, leading to unauthorized access, data breaches, and data corruption. This can compromise the confidentiality, integrity, and availability of the database and its data. By using parameterized queries and prepared statements, developers can prevent SQL injection attacks by separating user input from the SQL command, ensuring that the input is treated strictly as a value and not as a part of the SQL code.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code constructs an SQL query by concatenating user input directly into the query string, making it vulnerable to SQL injection attacks. If an attacker inputs something like "'; DROP TABLE users; --", it could alter the query and execute unintended SQL commands, leading to a potential data breach or loss.

```
#include <iostream>
#include <string>
#include <sqlite3.h>

void executeQuery(sqlite3* db, const std::string& userInput) {
    std::string query = "SELECT * FROM users WHERE username = '" +
userInput + "'";
    sqlite3_exec(db, query.c_str(), nullptr, nullptr, nullptr);
}
```

Compliant Code

Updated 08/07/2024 – Revision 2

This code uses a prepared statement with parameterized queries in SQLite, ensuring that user input is safely handled and not directly concatenated into the SQL query. This method ensures that the input is treated as

Compliant Code

data and cannot alter the structure of the query, effectively preventing SQL injection attacks and enhancing the security of the database operations.

```
#include <iostream>
#include <string>
#include <sqlite3.h>

void executeQuery(sqlite3* db, const std::string& userInput) {
    sqlite3_stmt* stmt;
    const char* query = "SELECT * FROM users WHERE username = ?";
    sqlite3_prepare_v2(db, query, -1, &stmt, nullptr);
    sqlite3_bind_text(stmt, 1, userInput.c_str(), -1, SQLITE_STATIC);
    sqlite3_step(stmt);
    sqlite3_finalize(stmt);
}
```

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
- Adopt a Secure Coding Standard

Validate input data and Sanitize data sent to other system principles map this security standard because they help preventing SQL injection attacks by ensuring that user input cannot be misinterpreted as SQL commands.

Adopt a secure coding standard principle maps this standard because it promotes the use of parameterized queries and prepared statements, which are key defenses against SQL injection.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	5

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9 LTS	CPP:S2077	Detects potential SQL injection vulnerabilities by analyzing SQL queries that are built using untrusted input.

Tool	Version	Checker	Description Tool
Fortify Static Code Analyzer	23.1	SQL INJECTION	Analyzes SQL queries for injection vulnerabilities, particularly those involving user input.
Checkmarx	9.5	SQL INJECTION	Scans code for potential SQL injection risks, particularly in concatenated SQL queries.
Veracode	2023.2	CWE-89	Identifies SQL injection vulnerabilities by checking user input directly embedded in SQL queries.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	***Updated 08/07/2024 – Revision 2*** Use modern C++ techniques and tools like smart pointers, RAII (Resource Acquisition Is Initialization), and bounds-checking functions to ensure that memory is managed safely and efficiently. This avoids common issues such as memory leaks, dangling pointers, and buffer overflows that can be exploited by attackers to execute arbitrary code, gain unauthorized access, or crash the program, leading to denial of service. Off-by-one errors in loops, improper use of raw pointers, and failure to check array bounds can result in writing or reading beyond allocated memory, causing memory corruption.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code contains an off-by-one error in the loop that writes beyond the bounds of the array, leading to a buffer overflow. This can cause memory corruption, crashes, and potential security vulnerabilities as the program may overwrite adjacent memory locations with arbitrary data.

```
void useArray(int* arr, size_t size) {
    for (size_t i = 0; i <= size; ++i) { // Off-by-one error
        arr[i] = i;
    }
}

int main() {
    int arr[10];
    useArray(arr, 10);
    return 0;
}
```

Compliant Code

This code uses std::array and its size() method to ensure that the loop does not exceed the array bounds, preventing buffer overflows.

```
void useArray(std::array<int, 10>& arr) {
    for (size_t i = 0; i < arr.size(); ++i) {
        arr[i] = i;
    }
```

Compliant Code

```
}

int main() {
    std::array<int, 10> arr;
    useArray(arr);
    return 0;

}
```

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

Principles(s):

- Heed Compiler Warnings
- Keep It Simple
- Practice Defense in Depth
- Adopt a Secure Coding Standard

These principles connect to this security standard because:

Compiler warnings can help detect memory-related issues, such as buffer overflows or unsafe pointer usage.

Keeping the code simple, such as using std::array instead of raw pointers, reduces complexity and potential security flaws.

Defense in depth ensures that multiple layers of checks and balances are in place to protect memory.

A secure coding standard provides guidelines for safe memory management practices.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	5

Automation

Tool	Version	Checker	Description Tool
Fortify Static Code Analyzer	23.1	BUFFER OVERFLOW	Detects vulnerabilities related to buffer overflows, including off-by-one errors.
GCC AddressSanitizer	GCC 12.2	-FSANITIZE=ADDRESS	Detects buffer overflows and other memory issues at runtime.

Tool	Version	Checker	Description Tool
Clang Static Analyzer	15.0	ALPHA.SECURITY.ARRAYBOUND	Analyzes code statically to detect buffer overflows and array bounds issues.
Polyspace Bug Finder	2023a	BUFFER OVERFLOW	Checks for buffer overflows and other memory-related issues in embedded systems.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	***Updated 08/07/2024 – Revision 2*** User assertion to validate assumptions. This ensures that the code operates under expected conditions by verifying assumptions and invariants. Do not use assertions for runtime error handling, as they are typically disabled in production builds. When assertions are used appropriately, developers can catch logical errors early in the development process, leading to more robust and maintainable code. Failing to validate assumptions in the code can lead to undefined behavior, which attackers may exploit to cause unexpected program behavior, crashes, or even execute arbitrary code. For example, dividing by zero without checking can crash the program or cause unpredictable results. While assertions help catch such errors early during development, relying on them in production without proper error handling can result in security risks if the assertions are disabled.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code assumes that the divisor “b” is not zero without verifying the assumption, leading to undefined behavior when attempting to divide by zero. This oversight can result in program crashes or unpredictable behavior, potentially introducing security vulnerabilities into the application.

```
int divide(int a, int b) {
    // Assumes b is not zero
    return a / b;
}

int main() {
    int result = divide(10, 0); // Undefined behavior
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

Compliant Code

This code uses an assertion to verify that the divisor “b” is not zero, catching the logical error early during development and preventing undefined behavior.

Compliant Code

```
int divide(int a, int b) {
    assert(b != 0 && "Divisor must not be zero");
    return a / b;
}

int main() {
    int result = divide(10, 2); // Valid division
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

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

These principles connect to this security standard because:

Assertions are a quality assurance technique that can help catch logical errors during development, ensuring that the code operates under expected conditions.

Keeping the code simple and focused on verifying assumptions helps prevent complex issues that might arise from unchecked assumptions.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Low	Medium	2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.4	ASSERT_USAGE	Checks for the proper use of assertions in the code and flags improper usage.
GCC Static Analyzer	GCC 12.2	ASSERT	Ensures proper usage of assertions and flags misuse.
Parasoft Insure++	2023.1	ASSERTIONS	Detects assertion failures and logical errors related to assumption checks.
CodeSonar	7.4	ASSERT_USAGE	Analyzes the correctness of assertions in C++ code.

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	***Updated 08/07/2024 – Revision 2*** Use exceptions for error handling. Exceptions provide a robust mechanism for handling errors in C++ by separating error-handling code from regular code. This separation enhances code readability and maintainability. Exceptions allow propagating error information up the call stack, enabling centralized error handling and recovery. Inconsistent or improper error handling might lead to vulnerabilities where attackers could exploit unhandled exceptions to crash the application or gain unauthorized access.

Noncompliant Code

This code handles errors by printing error messages and returning from the function, which mixes error handling with regular code and does not propagate the error.

```
void divide(int a, int b) {
    if (b == 0) {
        std::cerr << "Error: Division by zero" << std::endl;
        return;
    }
    std::cout << "Result: " << a / b << std::endl;
}

int main() {
    divide(10, 0); // Error handling via error messages
    return 0;
}
```

Compliant Code

Updated 08/07/2024 – Revision 2

This code uses exceptions to handle errors, separating error-handling code from regular code and propagating the error for centralized handling. When a division by zero is detected, it throws an std::invalid_argument exception, which is then caught and handled in the main function.

```
int divide(int a, int b) {
    if (b == 0) {
```

Compliant Code

```
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
        std::cout << "Result: " << result << std::endl;
    } catch (const std::invalid_argument& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

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

Principles(s):

- Architect and Design for Security Policies
- Use Effective Quality Assurance Techniques

These principles connect to this security standard because:

Exceptions must be architected and designed into the codebase to handle errors securely, ensuring that errors are not only caught but also managed in a way that doesn't leave the system in an inconsistent or insecure state.

Effective quality assurance techniques include testing exception handling paths to ensure they behave as expected.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Medium	Medium	4

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	EXCEPTION-USAGE	Analyzes proper use of exceptions in C++ and ensures that error handling is done correctly.
Helix QAC	2023.1	CERT_CPP-ERR56-A	Always catch exceptions.
		CERT_CPP-ERR56-B	Do not leave 'catch' blocks empty.

Tool	Version	Checker	Description Tool
Polyspace Bug Finder	R2024a	CERT C++: ERR58-CPP	Checks for exceptions raised during program startup (rule fully covered)
Clang Static Analyzer	15.0	CPLUSPLUS.EXCEPTIONSAFETY	Analyzes the safety of exception handling and ensures proper cleanup in case of exceptions.

Coding Standard 8

Coding Standard	Label	Name of Standard
Data Type	[STD-008-CPP]	***Updated 08/07/2024 – Revision 2*** Use “enum class” to ensure type-safe and scoped enumerations. This prevents name conflicts and unintended implicit conversions. Unscoped enumerations can introduce security vulnerabilities by allowing implicit conversions, which may lead to errors if not carefully managed. They can also lead to name conflicts, where enumeration values may unintentionally overlap with other identifiers, causing bugs or unpredictable behavior.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code uses unscoped enumerations, which can lead to name conflicts and implicit conversions. This can result in unintended behavior or type mismatches if not properly managed.

```
enum Color { Red, Green, Blue };

void printColor(Color color) {
    if (color == Red) {
        std::cout << "Red" << std::endl;
    } else if (color == Green) {
        std::cout << "Green" << std::endl;
    } else if (color == Blue) {
        std::cout << "Blue" << std::endl;
    }
}

int main() {
    Color color = Red; // Implicit conversion allowed
    printColor(color);
    return 0;
}
```

Compliant Code

This code uses scoped enumerations (enum class), ensuring type safety and preventing name conflicts.

Compliant Code

```
enum class Color { Red, Green, Blue };

void printColor(Color color) {
    switch (color) {
        case Color::Red:
            std::cout << "Red" << std::endl;
            break;
        case Color::Green:
            std::cout << "Green" << std::endl;
            break;
        case Color::Blue:
            std::cout << "Blue" << std::endl;
            break;
    }
}

int main() {
    Color color = Color::Red; // No implicit conversion
    printColor(color);
    return 0;
}
```

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

Principles(s):

- Heed Compiler Warnings
- Adopt a Secure Coding Standard

These principles connect to this security standard because:

Using enum class leverages the compiler's ability to enforce type safety, reducing the risk of errors.

A secure coding standard ensures that developers consistently use safer, scoped enumerations, preventing issues like name conflicts and unintended implicit conversions.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Low	Medium	3

Automation

Tool	Version	Checker	Description Tool
Clang-Tidy	15.0	MODERNIZE-USE-ENUM-CLASS	Encourages the use of enum class instead of traditional enumerations.
Klocwork	2023.1	MODERN_CPP	Encourages the use of modern C++ constructs like enum class for type safety.
PVS-Studio	7.19	V1037	Flags unscoped enumerations and suggests using enum class for better type safety.
MISRA C++ Compliance Checker	Various	RULE 7-3-1	Ensures that enumerations are used safely and appropriately according to MISRA C++ guidelines.

Coding Standard 9

Coding Standard	Label	Name of Standard
Data Type	[STD-009-CPP]	***Updated 08/07/2024 – Revision 2*** Use “struct” and “class” appropriately for user-defined types, this ensures clear and organized code, with “class” for complex behavior and “struct” for simple data structures. Inappropriate use of “struct” and “class” can lead to design issues and potential security vulnerabilities. Using “class” for simple data structures might inadvertently introduce unnecessary complexity and access control, while using “struct” for complex types can expose internal data and logic, leading to potential misuse or security risks. Properly distinguishing between struct and class helps ensure that data is encapsulated and protected correctly, reducing the risk of unintended data exposure or manipulation.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code uses “class” for a simple data structure, which is less appropriate for such use cases compared to “struct”.

```
class Point {
public:
    int x;
    int y;
};

int main() {
    Point p;
    p.x = 5;
    p.y = 10;
    return 0;
}
```

Compliant Code

Updated 08/07/2024 – Revision 2

This code uses “struct” for a simple data structure (Point) and “class” for a type with more complex behavior (Circle), following best practices.

Compliant Code

```
struct Point {
    int x;
    int y;
};

class Circle {
private:
    Point center;
    double radius;
public:
    Circle(Point c, double r) : center(c), radius(r) {}

    // Methods to interact with Circle
    double getArea() const {
        return 3.14159 * radius * radius;
    }
};

int main() {
    Point p = {5, 10};
    Circle c(p, 5.0);
    return 0;
}
```

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

Principles(s):

- Keep It Simple
- Architect and Design for Security Policies

These Principles connect to this security standard because:

Using struct for simple data structures and class for more complex types keeps the code simple and intuitive, aligning with the principle of simplicity.

Proper architecture and design ensure that the distinction between struct and class is clear, enhancing maintainability and security by using the right tool for the job.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	Low	2

Automation

Tool	Version	Checker	Description Tool
Cppcheck	2.11	STYLE	Cppcheck can provide warnings if a class is used in a context where a struct might be more appropriate.
SonarQube	9.9 LTS	CPP:S1003	Analyzes the use of struct versus class and ensures proper usage.
PVS-Studio	7.19	V730	Encourages the correct use of struct for POD (Plain Old Data) types and class for more complex structures.
Cppcheck	2.11	STYLE	Flags incorrect usage of struct and class based on context.

Coding Standard 10

Coding Standard	Label	Name of Standard
Memory Protection	[STD-010-CPP]	***Updated 08/07/2024 – Revision 2*** Use “static” for class-level constants to ensure that these constants are shared among all instances of the class. This approach prevents redundant copies of the constant in each object instance saving memory and reducing the risk of inconsistencies and makes the code easier to maintain. Without “static”, each instance of the class will have its own copy of the constant, leading to increased memory usage and potential inconsistencies if the constant's value is changed. This can be particularly problematic in applications with many instances of the class, potentially leading to higher memory consumption and making the code harder to maintain. Using static ensures that the constant is consistently managed, reducing the risk of memory-related issues and ensuring that the constant's value remains consistent throughout the application.

Noncompliant Code

Updated 08/07/2024 – Revision 2

This code declares a constant “maxValue” as a non-static member of “MyClass”. As a result, each instance of “MyClass” has its own copy of “maxValue”, leading to redundant memory usage and potential inconsistencies if the value were to be modified. This is less efficient and harder to maintain.

```
class MyClass {
public:
    const int maxValue = 10; //Non-static constant
};

int main() {
    MyClass e1;
    MyClass e2;
    std::cout << "Max Value: " << e1.maxValue << std::endl;
    std::cout << "Max Value: " << e2.maxValue << std::endl;
    return 0;
}
```

Compliant Code

Updated 08/07/2024 – Revision 2

Compliant Code

This code uses “static” to declare “MAX_VALUE” as a class-level constant, This ensures that MAX_VALUE is shared among all instances of “MyClass”, saving memory and ensuring consistency.

```
class MyClass {
public:
    static const int MAX_VALUE = 10; //static constant
};

int main() {
    Std::cout << “Max value: “ << MyClass:: MAX_VALUE << std::endl;
    Return 0;
}
```

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

Principles(s):

- Practice Defense in Depth
- Adopt a Secure Coding Standard

These principles connect to this security standard because:

Using static constants reduces memory usage and prevents redundancy, which aligns with defense in depth by minimizing potential attack vectors related to memory management.

A secure coding standard enforces the consistent use of static constants, ensuring that memory protection is maintained across the codebase.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	Low	2

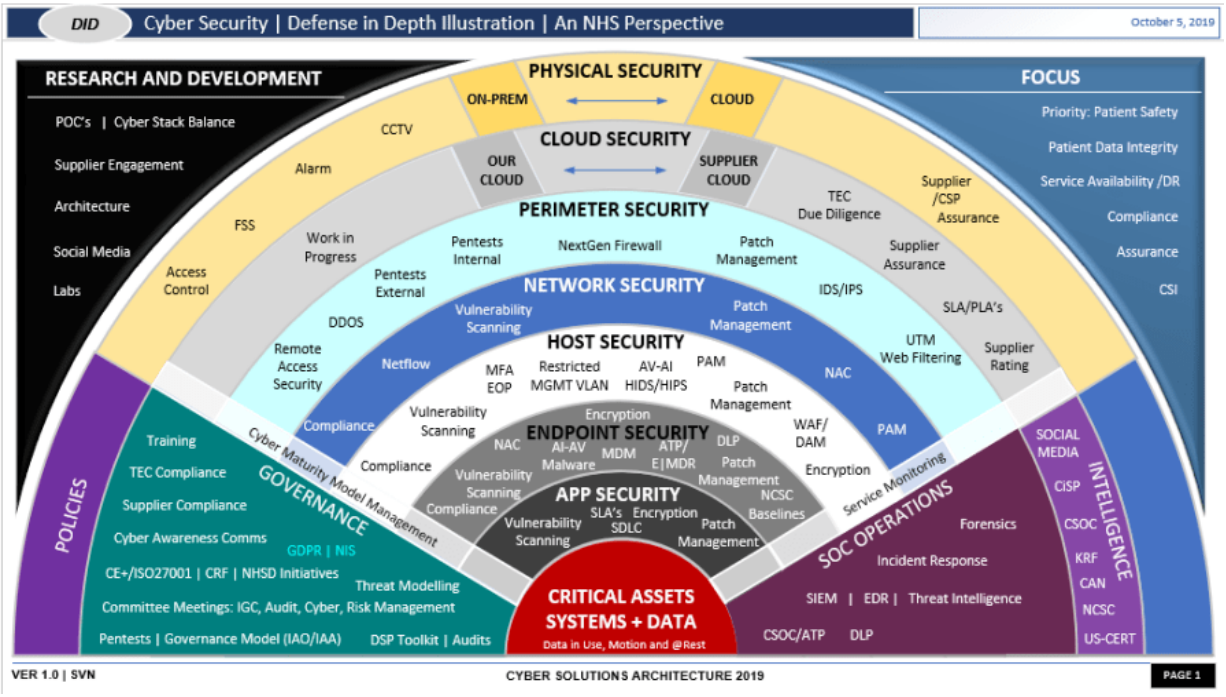
Automation

Tool	Version	Checker	Description Tool
PVS-Studio	7.19	V730	Checks for the correct use of static keyword and ensures that static members are used efficiently.
Coverity	2023.0623 q	STATIC_CONST	Ensures that constants are declared static to avoid unnecessary memory usage.

Tool	Version	Checker	Description Tool
Clang-Tidy	15.0	READABILITY-AVOID-CONST-PARAMS-IN-DECLS	Encourages proper use of const and static keywords for memory efficiency.
Klocwork	2023.1	STATIC_MEM_USAGE	Analyzes memory usage of static members and flags inefficient usage patterns

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

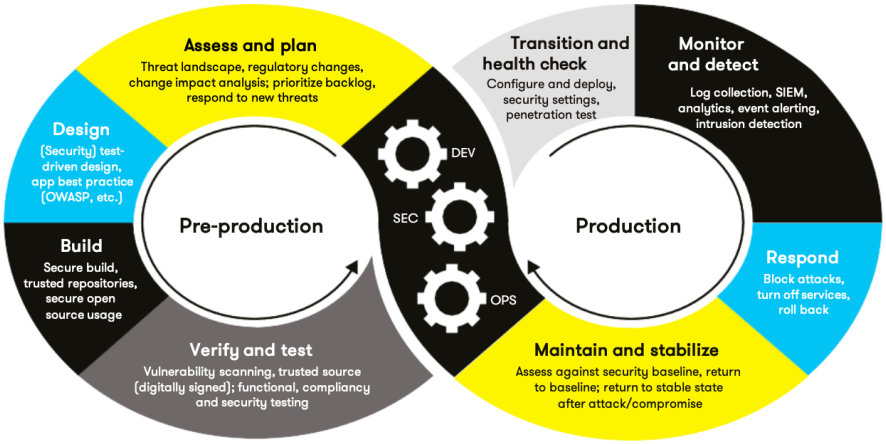
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

To enforce compliance with the security standards in this policy efficiently, I recommend Green Pace to integrate automation tools throughout DevSecOps pipeline following this approach:

Pre-Production:

Assess and Plan: In this initial stage, automation tools can be used to assess the codebase for potential vulnerabilities and ensure that security policies are integrated into the planning phase. Tools like SonarQube can scan the existing codebase for non-compliance with the coding standards in this policy. And can also help prioritize security-related tasks in the backlog, ensuring that potential vulnerabilities are addressed early.

Automation Focus: Automated code review tools, threat modeling tools, and backlog prioritization based on security risk.

Design: During the design phase, automation tools can enforce secure coding practices by integrating with design tools and IDEs to suggest security best practices (e.g., OWASP guidelines) and ensure compliance with standards such as proper use of data types, memory protection, and exception handling.

Automation Focus: Security test-driven design integration, automated code linters, and static analysis for design patterns.

Build: In the build stage, automation tools can enforce secure coding standards by failing builds that do not meet the defined security criteria. Tools like Jenkins with security plugins can automate this process. Dependency management tools can also scan for vulnerabilities in third-party libraries, ensuring that only secure and trusted components are used.

Automation Focus: Continuous Integration (CI) tools with security checks, automated dependency scanning, and build failure triggers for non-compliance.

Verify and Test: Automation tools for vulnerability scanning, penetration testing, and static and dynamic application security testing (SAST and DAST) should be integrated into the pipeline.

These tools can automatically test for issues like SQL injection vulnerabilities, memory protection flaws, and incorrect use of assertions or exceptions.

Automation Focus: Automated SAST/DAST tools, vulnerability scanners, and compliance testing tools.

Production:

Transition and Health Check: Before transitioning to production, automated tools should perform final security checks, including configuration checks and penetration testing. Tools like Chef Inspec can verify that security settings are correctly applied, and any last-minute vulnerabilities are addressed.

Automation Focus: Automated configuration and deployment checks, final security testing, and compliance verification.

Monitor and Detect: Once in production, automated tools for log collection, Security Information and Event Management (SIEM), and Intrusion Detection Systems (IDS) can be used to monitor for compliance with security standards in real time and alert when anomalies are detected.

Automation Focus: Continuous monitoring tools, SIEM integration, and automated event alerting.

Respond: In case of a detected threat or vulnerability, automated tools should be capable of taking immediate action, such as blocking attacks, rolling back deployments, or disabling services to contain the issue. Automation ensures that response times are minimized and that security policies are enforced consistently.

Automation Focus: Automated incident response tools, rollback mechanisms, and service disabling.

Maintain and Stabilize: Post-incident, automation tools should help assess the system against the security baseline and ensure that it returns to a stable state. Regular automated security assessments ensure ongoing compliance with the established standards and policies.

Automation Focus: Automated security assessments, baseline checks, and recovery tools.

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	Likely	Low	Medium	3
STD-002-CPP	Medium	Likely	Low	Medium	4
STD-003-CPP	High	Likely	Medium	High	5
STD-004-CPP	High	Likely	Medium	High	5

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-005-CPP	High	Likely	Medium	High	5
STD-006-CPP	Medium	Unlikely	Low	Medium	2
STD-007-CPP	Medium	Likely	Medium	Medium	4
STD-008-CPP	Medium	Unlikely	Low	Medium	3
STD-009-CPP	Low	Unlikely	Low	Low	2
STD-0010-CPP	Low	Unlikely	Low	Low	2

Create Policies for Encryption and Triple A

Include all three types of encryptions (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided**.**

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* **refers to** the process of encrypting data stored on physical media, such as hard drives, SSDs, or backup tapes. This ensures that data remains secure even when it is not actively being used or transmitted. Encryption at rest **is applied** to all forms of data storage, including databases, file systems, and backup storage. This encompasses data stored on both company-owned hardware and in cloud environments. Encryption at rest **should be used because** it protects data from unauthorized access in the event of physical theft or unauthorized access to storage systems. **Policy Application:** ♦ All sensitive data stored on physical or virtual media must be encrypted using strong encryption algorithms (e.g., AES-256). ♦ Sensitive files must be stored on encrypted file systems. Encryption tools such as BitLocker, LUKS, or cloud provider-native encryption should be used. ♦ All backups containing sensitive or critical information must be encrypted before storage.

a. Encryption	Explain what it is and how and why the policy applies.
	♦ Data stored in cloud environments must be encrypted using the cloud provider's encryption services or via client-side encryption before uploading. ♦ Access to encrypted data at rest must be controlled through strong authentication mechanisms. ♦ Regular audits and assessments must be conducted to ensure compliance with encryption at rest policies.
Encryption in flight	Encryption *in flight* **refers to** the process of encrypting data as it is transmitted over networks. This ensures that data is protected from interception and tampering during transfer between systems, whether internally or externally. Encryption in flight **is applied** whenever data is transmitted across any network. This includes data sent between client devices and servers, between servers within the data center, and between our systems and third-party services. Encrypting data in flight **should be used because** it prevents unauthorized users from intercepting or altering the data while it's being transmitted. This is critical in preventing man-in-the-middle attacks, eavesdropping, and data leaks during transmission. **Policy Application:** ♦ All sensitive data transmitted over the internet or internal networks must be encrypted using secure protocols such as TLS (Transport Layer Security) or SSL (Secure Sockets Layer). ♦ Virtual Private Networks (VPNs) must be used to secure internal communications when crossing public or untrusted networks. ♦ Emails containing sensitive information must be encrypted using S/MIME or PGP. ♦ All API interactions should be secured with HTTPS, and tokens or credentials transmitted over the network must be encrypted. ♦ Any application that transmits personal, financial, or proprietary information must implement encryption in flight. ♦ Testing and verification of encryption in flight must be included in the development and deployment processes**.**

a. Encryption	Explain what it is and how and why the policy applies.
Encryption in use	Encryption *in use* **refers to** the process of protecting data while it is being processed, such as when it is loaded into memory or being manipulated by an application. This is typically achieved through techniques like secure enclave technologies and homomorphic encryption. Encryption in use **is applied** during the processing of sensitive data, particularly in environments where data confidentiality is paramount even during computation, such as in financial transactions, medical data processing, and secure multiparty computations. Encryption in use **should be used because** it protects data during the most vulnerable stages of its lifecycle, where it is typically decrypted for processing. This ensures that even if the system is compromised, the data remains secure. **Policy Application:** ♦ All sensitive data processing must utilize secure enclave technologies (e.g., Intel SGX or AWS Nitro Enclaves) or equivalent measures to protect data in use. ♦ In scenarios where data must be processed without decryption, implement homomorphic encryption to enable computation on encrypted data without exposing it. ♦ Ensure that sensitive data loaded into memory is encrypted and that memory encryption features are enabled on servers where applicable.

b. Triple-A Framework *	Explain what it is and how and why the policy applies.
Authentication	Authentication **is the process of** verifying the identity of a user or system. It ensures that only legitimate users can access Green Pace's resources. Authentication **must be implemented** for all user logins, including web applications, internal tools, and APIs. Multi-factor authentication (MFA) should be enabled wherever possible to add an extra layer of security. Authentication policies **should be used because** they are critical in preventing unauthorized access to sensitive data and systems. They apply at all times and are especially important when accessing sensitive applications or systems, such as databases and cloud environments.

b. Triple-A Framework *	Explain what it is and how and why the policy applies.
	Policy Application: ♦ All user logins must be authenticated using strong passwords and MFA. Passwords should meet complexity requirements and be stored using secure hashing algorithms. ♦ The creation of new user accounts must follow a standardized onboarding process, which includes identity verification and approval from a supervisor.
Authorization	Authorization **is the process of** determining what actions a user is allowed to perform after they have been authenticated. Authorization must be implemented to control access to various levels of resources based on the user's role within Green Pace. Role-Based Access Control (RBAC) should be employed to enforce least privilege, ensuring users only have access to the resources necessary for their role. Authorization policies **should be used because** they protect critical resources by ensuring that users can only perform actions for which they have explicit permission. These policies must be enforced continuously and reviewed regularly to adjust for changes in user roles and responsibilities. **Policy Application:** ♦ User roles must be clearly defined, and access levels must be granted accordingly. Changes to user roles or permissions must be logged and reviewed periodically. ♦ Access to files and directories must be restricted based on the user's role. Sensitive files must be encrypted, and access should be monitored.
Accounting	Accounting **involves tracking** user activities and system events to provide an audit trail for security and compliance purposes. Accounting **must be implemented** across all systems to log user actions, changes to configurations, and access to sensitive resources. Logs should be stored securely and reviewed regularly. Accounting policies **should be used because** they are essential for detecting and responding to security incidents and for compliance with regulatory requirements.

b. Triple-A Framework *	Explain what it is and how and why the policy applies.
	Logs must be maintained continuously and stored for an appropriate duration based on Green Pace's data retention policy. **Policy Application:** • All changes to databases, including schema modifications and data updates, must be logged with details of the user performing the action and the changes made. • Access to critical files and directories must be logged, including the time of access and the actions performed.

***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	07/18/2024	Initial Template	Carmen Mosquera	
2.0	08/07/2024	Final Template	Carmen Mosquera	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV