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	[Insert text.] All input data must be treated as untrusted and validated for type, length, format, and range before processing. This prevents common vulnerabilities like buffer overflows, SQL injection, and cross-site scripting (XSS) by ensuring the application only accepts data that conforms to strict expectations.
2. Heed Compiler Warnings	[Insert text.] Compiler warnings should be addressed and resolved, not ignored. Compilers are sophisticated tools that can detect code patterns leading to undefined behavior, potential bugs, and security flaws. Maintaining a zero-warning policy ensures code quality and reduces the attack surface from easily preventable errors.
3. Architect and Design for Security Policies	[Insert text.] Security must not be an afterthought, but rather a fundamental component of the original architecture and design process. Data flow, trust boundaries, and security controls should all be considered while designing systems to make sure that the application's core architecture upholds the company's security guidelines from the very beginning.
4. Keep It Simple	[Insert text.] Security is hampered by complexity. Design and code should be as straightforward as feasible. Simple designs are less likely to introduce errors and are more difficult for attackers to discover and take advantage of hidden vulnerabilities since they are simpler to examine, test, and maintain.
5. Default Deny	[Insert text.] Unless specifically allowed, the default security posture should be to prohibit access. This idea reduces the attack surface by making sure that only resources that are required are accessible and that everything else is prohibited by default, whether it is applied to file system access, network traffic, or user rights.
6. Adhere to the Principle of Least Privilege	[Insert text.] Each user and process should only have the bare minimum of privileges required to carry out its function. This reduces the potential harm that an accident, mistake, or attack could do. A user account used to see data, for instance, shouldn't be able to remove it.

Principles	Write a short paragraph explaining each of the 10 principles of security.
7. Sanitize Data Sent to Other Systems	[Insert text.] Databases, operating system instructions, and other programs are examples of external systems that require data to be sanitized or appropriately escaped. By doing this, injection attacks are avoided, in which a hacker manipulates data to fool a downstream system into carrying out destructive orders.
8. Practice Defense in Depth	[Insert text.] There should be several overlapping layers of security implementation. Another line of defense should stop a breach if the first one fails. This strategy increases the system's resistance to attacks by preventing any one point of failure from compromising the entire system.
9. Use Effective Quality Assurance Techniques	[Insert text.] Strong quality assurance procedures, including as penetration testing, fuzz testing, and static and dynamic code analysis, are crucial for finding vulnerabilities that developers might have overlooked. These methods aid in making sure that security is checked at every stage of the development process.
10. Adopt a Secure Coding Standard	[Insert text.] Developers are given a consistent set of guidelines to follow when a secure coding standard is established and upheld. This establishes a baseline for code reviews and automated analysis tools and lowers frequent programming errors that result in vulnerabilities.

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-nnn-LLL]	[Rationalize the standard.]

Noncompliant Code

[Noncompliant description] Using a legacy C-style enum which pollutes the global namespace and allows implicit conversion to integral types, leading to potential logical errors.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
enum Color { Red, Green, Blue };
enum TrafficLight { Red, Yellow, Green }; // Error: Red and Green are redefined

void setSignal(Color c);
setSignal(1); // Implicit conversion from int, logic error
```

Compliant Code

[Compliant description] Using enum class which is scoped and does not allow implicit conversions, ensuring type safety.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
enum class Color { Red, Green, Blue };
enum class TrafficLight { Red, Yellow, Green }; // No conflict

void setSignal(Color c);
setSignal(Color::Green); // Correct, explicit scope
// setSignal(1);       // Compile-time error
```

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

Principles(s): [Name the principle and explain how it maps to this standard.] Principles(s): #4 Keep It Simple & #9 Effective QA. Strong typing simplifies code reasoning and prevents a class of errors that are difficult to catch during testing, making QA more effective.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] low	[Insert text.] Probable	[Insert text.] Low	[Insert text.] Medium	[Insert text.] 2

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] modernize-use-enum-decl	[Insert text.] Suggests replacing C-style enum with enum class.
[Insert text.] SonarQube (C++ Plugin)	[Insert text.] 8.x+	[Insert text.] Rule S2810	[Insert text.] Flags C-style enums that cause namespace pollution.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP	[Rationalize the standard.] Ensure integer operations do not wrap unexpectedly

Noncompliant Code

[Noncompliant description] Performing arithmetic operations without checking for overflow/underflow can lead to undefined behavior or logic errors.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
#include <cstdint>
void allocate_bytes(std::size_t len) {
   // Potential overflow: if len is large, len * 8 can wrap around.
   int64_t total_bits = len * 8;
   char* buffer = new char[total_bits]; // Allocation size may be incorrect
   // ...
}
```

Compliant Code

[Compliant description] Using checked arithmetic or testing operands to prevent wrapping.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <cstdint>
#include <limits>
void allocate_bytes(std::size_t len) {
   if (len > (std::numeric_limits<std::size_t>::max() / 8)) {
      // Handle error: overflow would occur
      return;
   }
   std::size_t total_bits = len * 8;
   char* buffer = new char[total_bits];
   // ...
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.] Principles(s): #1 Validate Input Data. This standard is a direct application of input validation, ensuring that the results of arithmetic operations are within expected and safe bounds.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] High	[Insert text.] Probable	[Insert text.] Low	[Insert text.] High	[Insert text.] 4

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] clang-analyzer-security.insecureAPI	[Insert text.] Warns on use of strcpy, gets, etc.
[Insert text.] GCC	[Insert text.] 8+	[Insert text.] -Wformat-overflow, -Wstringop-overflow ________	[Insert text.] Warns about buffer overflows in string functions.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	[Rationalize the standard.] Use bounds-checked functions for string manipulation

Noncompliant Code

[Noncompliant description] Using unsafe C-style string functions like strcpy and strcat can lead to buffer overflows.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
void copy_username(const char* input) {
    char username[16];
    strcpy(username, input); // Potential buffer overflow if input > 15 chars
}
```

Compliant Code

[Compliant description] Using the C++ std::string class or bounds-checked functions like strncpy_s.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <string>
void copy_username(const char* input) {
    std::string username = input; // Safe, manages its own memory
}

// Alternatively, using C bounds-checked functions (Microsoft)
void copy_username_s(const char* input, size_t input_size) {
    char username[16];
    strncpy_s(username, sizeof(username), input, _TRUNCATE);
}
```

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

Principles(s): #1 Validate Input Data & #8 Defense in Depth. Bounds checking is primary input validation. Using a safe string class provides a layer of defense against memory corruption.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] High	[Insert text.] Probable	[Insert text.] Low	[Insert text.] High	[Insert text.] 4

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] clang-analyzer-security.insecureAPI	[Insert text.] Warns on use of strcpy, gets, etc.
[Insert text.] GCC	[Insert text.] 8+	[Insert text.] -Wformat-overflow, -Wstringop-overflow _______	[Insert text.] Warns about buffer overflows in string functions.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP]	[Rationalize the standard.] Use parameterized queries (prepared statements

Noncompliant Code

[Noncompliant description] Concatenating user input directly into a SQL query string creates a SQL injection vulnerability.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
// DANGEROUS: User input concatenated directly into query
void search_user(sql::Connection* conn, const std::string& user_input) {
   std::string query = "SELECT * FROM users WHERE name = '" + user_input + "';";
   sql::Statement* stmt = conn->createStatement();
   stmt->executeQuery(query);
   // An input of " ' OR '1'='1 " would bypass authentication.
}
```

Compliant Code

[Compliant description] Using parameterized queries separates SQL code from data, preventing injection.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
// SAFE: Using a prepared statement
void search_user(sql::Connection* conn, const std::string& user_input) {
   sql::PreparedStatement* pstmt = conn->prepareStatement("SELECT * FROM users WHERE name = ?");
   pstmt->setString(1, user_input); // Data is safely bound to the parameter
   pstmt->executeQuery();
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]
Principles(s): #1 Validate Input Data & #7 Sanitize Data Sent to Other Systems. Parameterized queries are the most robust form of sanitizing data sent to a database system.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Low	High	5

Automation

Tool	Version	Checker	Description Tool
[Insert text.] SonarQube (C++ Plugin)	[Insert text.] 8.x+	[Insert text.] Rule RSPEC-3649	[Insert text.] Detects SQL injection vulnerabilities from string concatenation.
[Insert text.] Checkmarx	[Insert text.] N/A	[Insert text.] CxQL Rules	[Insert text.] Scans for tainted data flowing into SQL execution functions.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	[Rationalize the standard.] Use smart pointers for automatic resource management (RAII)

Noncompliant Code

[Noncompliant description] Using raw pointers and manual new/delete can lead to memory leaks, double-frees, and dangling pointers.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
void process_data() {
  MyClass* obj = new MyClass();
  if (some_condition()) {
    delete obj; // Early return, must delete here
    return;
  }
  // ... more code ...
  delete obj; // And also delete here
  // Easy to forget a delete path, leading to a leak.
}
```

Compliant Code

[Compliant description] Using std::unique_ptr or std::shared_ptr ensures memory is automatically freed when the pointer goes out of scope.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <memory>
void process_data() {
  auto obj = std::make_unique<MyClass>();
  if (some_condition()) {
    return; // obj is automatically deleted here
  }
  // ... more code ...
  // obj is automatically deleted when the function ends
}
```

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

Principles(s): #4 Keep It Simple & #9 Effective QA. RAII and smart pointers simplify memory management, eliminating entire categories of memory-related bugs and making code easier to reason about and test.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] High	[Insert text.] Probable	[Insert text.] Low	[Insert text.] High	[Insert text.] 4

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] modernize-use-unique_ptr, cppcoreguidelines-owning-memory	[Insert text.] Suggests replacing new/delete with smart pointers.
[Insert text.] PVS-Studio	[Insert text.] 7.x+	[Insert text.] V774	[Insert text.] Detects dangling pointers and memory leaks.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	[Rationalize the standard.] Use assertions for internal programming errors only, not for runtime input validation

Noncompliant Code

[Noncompliant description]
Using assert() to validate user or external input. Assertions are typically disabled in release builds, removing the check and creating a vulnerability.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
#include <cassert>
void set_index(int index) {
   // BAD: Assert is removed in NDEBUG builds
   assert(index >= 0 && index < MAX_SIZE);
   array[index] = 0;
}
```

Compliant Code

[Compliant description]
Use assertions for internal invariants that should never be false. Use proper error handling for runtime inputs.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <stdexcept>
void set_index(int index) {
   // GOOD: Runtime check is always active
   if (index < 0 || index >= MAX_SIZE) {
      throw std::out_of_range("Index out of range");
   }
   array[index] = 0;
}

// Use assert for an internal logic check
void internal_function(MyClass* ptr) {
   assert(ptr != nullptr && "ptr must not be null at this point"); // Internal error if false
   ptr->doSomething();
}
```

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

Principles(s): #1 Validate Input Data & #8 Defense in Depth. This standard ensures that input validation is always active, providing a consistent defensive layer, while assertions act as a development-time depth check for internal logic.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] Medium	[Insert text.] Probable	[Insert text.] low	[Insert text.] Medium	[Insert text.] 3

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] cppcoreguidelines-pro-bounds-array-to-pointer-decay	[Insert text.] Can be configured to warn on certain assert uses.
[Insert text.] CodeSonar	[Insert text.] N/A	[Insert text.] Custom Query	[Insert text.] Can find instances where assert is used with tainted (user) data.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	[Rationalize the standard.] Throw by value, catch by const reference

Noncompliant Code

[Noncompliant description]
Throwing a pointer or catching by value can lead to memory leaks or unnecessary object slicing/copying.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
// Noncompliant: Throwing a pointer
try {
  if (error) {
    throw new MyException("error"); // Who deletes this?
  }
} catch (MyException* e) { // Catching by pointer
  // ... handle ...
  delete e; // Must remember to delete
}

// Noncompliant: Catching by value (object slicing)
try {
  throw DerivedException();
} catch (BaseException e) { // Slicing: DerivedException info is lost
  // ...
}
```

Compliant Code

[Compliant description]

Throwing an object directly and catching by const reference is safe and efficient.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
// Compliant: Throw by value, catch by const reference
try {
  if (error) {
    throw MyException("error");
  }
} catch (const MyException& e) { // No copy on throw, no slicing, automatic cleanup
```

Compliant Code

```
    // ... handle ...
}
```

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

Principles(s): #4 Keep It Simple & #9 Effective QA. This standard simplifies exception handling, preventing subtle bugs like memory leaks and object slicing, which are difficult to identify during testing.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] Low	[Insert text.] Unlikely	[Insert text.] low	[Insert text.] low	[Insert text.] 1

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] cert-err09-cpp, cppcoreguidelines-exceptions	[Insert text.] Flags throwing of non-exception types and other bad practices.
[Insert text.] PVS-Studio	[Insert text.] 7.x+	[Insert text.] V1034	[Insert text.] Detects potential object slicing in catch blocks.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 8

Coding Standard	Label	Name of Standard
[Student Choice] Input Validation	[STD-008-CPP]	[Rationalize the standard.] Canonicalize and validate all file paths

Noncompliant Code

[Noncompliant description]

Using user-supplied paths directly can lead to path traversal attacks (e.g., ../../../etc/passwd).

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
#include <fstream>
void open_user_file(const std::string& username) {
  // Vulnerable to path traversal
  std::string path = "/home/userfiles/" + username;
  std::ifstream file(path);
  // A username of "../../../etc/passwd" would break out of the intended directory.
}
```

Compliant Code

[Compliant description]

Canonicalize the path and then validate it against a whitelist of allowed directories.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <filesystem>
#include <fstream>
void open_user_file(const std::string& username) {
  namespace fs = std::filesystem;
  fs::path base_dir = "/home/userfiles";
  fs::path user_path = base_dir / username; // Combines paths safely

  // Canonicalize and check if the result is within the base directory
  fs::path canonical_path = fs::canonical(user_path);
  if (canonical_path.string().find(base_dir.string()) != 0) {
    // Handle error: Path traversal attempted
    return;
  }
  std::ifstream file(canonical_path);
```

Compliant Code

```
}
```

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

Principles(s): #1 Validate Input Data & #5 Default Deny & #6 Least Privilege. This standard validates and restricts file system access to a specific, allowed directory, enforcing least privilege on file operations.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] High	[Insert text.] Probable	[Insert text.] Medium	[Insert text.] High	[Insert text.] 4

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang Static Analyzer	[Insert text.] 12+	[Insert text.] alpha.security.PathTraversal	[Insert text.] Can detect simple path traversal vulnerabilities.
[Insert text.] SonarQube (C++ Plugin)	[Insert text.] 8.x+	[Insert text.] Rule S2083	[Insert text.] Detects path manipulations that could lead to path traversal.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 9

Coding Standard	Label	Name of Standard
[Student Choice] Concurrency	[STD-009-CPP]	[Rationalize the standard.] Protect all shared data with locks (mutexes)

Noncompliant Code

[Noncompliant description]
Accessing shared data from multiple threads without synchronization causes data races and undefined behavior.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
#include <thread>
int global_counter = 0;

void increment_counter() {
    for (int i = 0; i < 1000000; ++i) {
        ++global_counter; // Data race!
    }
}

int main() {
    std::thread t1(increment_counter);
    std::thread t2(increment_counter);
    t1.join(); t2.join();
    // global_counter will likely not be 2000000.
}
```

Compliant Code

[Compliant description]

Using a std::mutex to synchronize access to the shared data.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <thread>
#include <mutex>
int global_counter = 0;
std::mutex counter_mutex;

void increment_counter() {
    for (int i = 0; i < 1000000; ++i) {
        std::lock_guard<std::mutex> lock(counter_mutex);
```

Compliant Code

```
        ++global_counter; // Safe, mutually exclusive access
    }
}
```

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

Principles(s): #8 Defense in Depth & #9 Effective QA. Proper locking is a fundamental layer of defense in concurrent systems. Data races are notoriously difficult to detect with QA, so preventing them by design is critical.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] High	[Insert text.] Probable	[Insert text.] High	[Insert text.] High	[Insert text.] 4

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang ThreadSanitizer (TSan)	[Insert text.] N/A	[Insert text.] Dynamic Analysis	[Insert text.] Detects data races at runtime.
[Insert text.] Helgrind (Valgrind)	[Insert text.] N/A	[Insert text.] Dynamic Analysis	[Insert text.] Detects synchronization errors in multithreaded programs.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 10

Coding Standard	Label	Name of Standard
[Student Choice] Error Handling	[STD-010-CPP]	[Rationalize the standard.] Do not ignore return values from functions

Noncompliant Code

[Noncompliant description] Ignoring the return value of a function, especially one that indicates an error state, can leave the program in an inconsistent state.

[Noncompliant code block; code should be indented using 12-point Courier New font.]

```
#include <cstdio>
void read_file() {
   FILE* f = fopen("data.txt", "r");
   // DANGEROUS: What if the file didn't open?
   char buffer[100];
   fgets(buffer, sizeof(buffer), f); // Potential null pointer dereference
   // ...
}
```

Compliant Code

[Compliant description] Always check the return value of functions that can fail and handle errors appropriately.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <cstdio>
#include <cstdlib>
void read_file() {
   FILE* f = fopen("data.txt", "r");
   if (f == nullptr) {
      // Handle error: log message, throw exception, etc.
      perror("Failed to open file");
      return;
   }
   char buffer[100];
   if (fgets(buffer, sizeof(buffer), f) != nullptr) {
      // ... process buffer ...
   }
```

Compliant Code

```
    fclose(f);
}
```

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

Principles(s): #9 Use Effective Quality Assurance Techniques. Handling errors explicitly makes the code's behavior predictable and testable, which is a cornerstone of effective QA. It prevents crashes and undefined behavior.

Threat Level

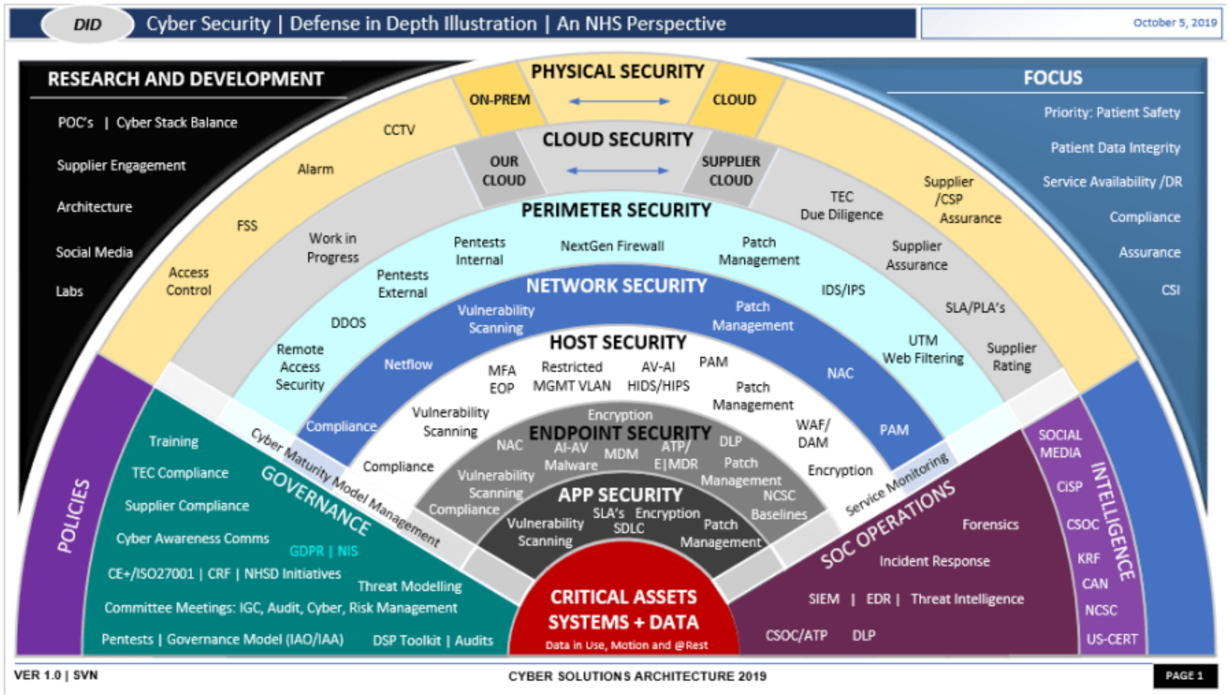
Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.] Medium	[Insert text.] Probable	[Insert text.] Low	[Insert text.] Medium	[Insert text.] 3

Automation

Tool	Version	Checker	Description Tool
[Insert text.] Clang-Tidy	[Insert text.] 12+	[Insert text.] clang-analyzer-valist.Uninitialized / cppcoreguidelines-pro-type-vararg	[Insert text.] Warns on ignoring specific critical return values like printf.
[Insert text.] PVS-Studio	[Insert text.] 7.x+	[Insert text.] V1001	[Insert text.] Detects ignored function return values that are critical.
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

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

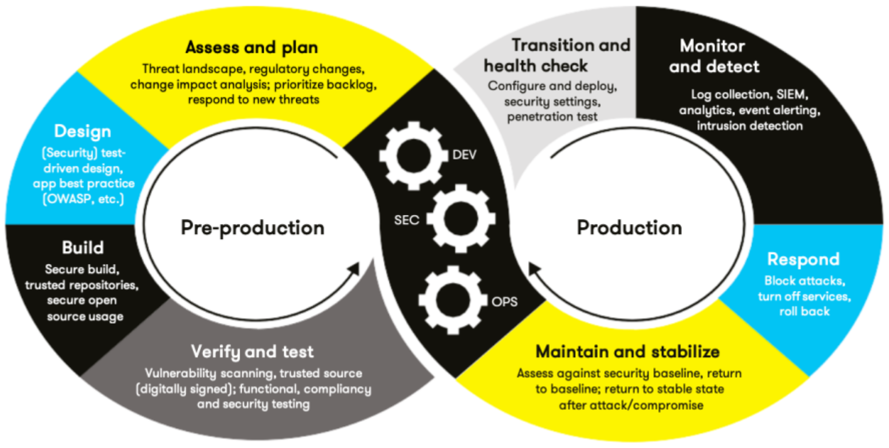
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

[Insert your written explanations here.]

Green Pace will convert its current DevOps pipeline into a DevSecOps pipeline by integrating security controls directly into it in order to automate the enforcement of the security requirements outlined in this policy. Security checks will be incorporated at the following crucial points, using the above diagram as a guide:

Plan/Develop: Integrated development environments (IDEs) with plugins for static analysis tools (such as Clang-Tidy and the Clang Static Analyzer) will be used by developers throughout this first phase. This keeps problems from being submitted to the code repository by giving instant, real-time feedback on the coding standards (such as STD-001-CPP and STD-005-CPP) as the code is being created.

Commit: Automated gates will be activated when a developer commits code to the source code repository (such as Git). The entire codebase will be subjected to a comprehensive suite of static application security testing (SAST) tools (like SonarQube or PVS-Studio) by a Continuous Integration (CI) server, like Jenkins or GitLab CI. If any coding standards violations with a high threat level (such as Level 4 or 5) are found, this build will fail, stopping susceptible code from moving forward.

Test/Staging: The application will be moved to a staging environment following a successful build. Here, the running program and its dependencies will be scanned for vulnerabilities using software composition analysis (SCA) and dynamic application security testing (DAST) technologies.

Release/Deploy: The application can be made available after passing all automated security checks. As required by the Audit Controls and Management portion of this policy, the pipeline can be set up to automatically produce proof of compliance (such as SAST/DAST reports and code coverage metrics) for audit purposes.

Monitor/Operate: Security monitoring is ongoing in the production environment. Intrusion Detection

Systems (IDS) and runtime application self-protection (RASP) technologies are able to identify and notify users of attempted exploits, offering the last line of automated defense and supplying data to the "Plan" stage for bettering future code.

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	PriorityLevel	
STD-001-CPP	Low	Probable	Low	Medium	2
STD-002-CPP	High	Probable	Medium	High	4
STD-003-CPP	High	Probable	Low	High	4
STD-004-CPP	High	Probable	Low	High	5
STD-005-CPP	High	Probable	Low	High	4
STD-006-CPP	Medium	Probable	Low	Medium	3
STD-007-CPP	Low	Unlikely	Low	Low	1
STD-008-CPP	High	Probable	Medium	High	4
STD-009-CPP	High	Probable	High	High	4
STD-010-CPP	Medium	Probable	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	[Insert text.] When data is kept on persistent media (hard drives, file servers, databases, backups), it is encrypted. How/Why: Industry-standard encryption techniques, such as AES-256, must be used for all configuration files, application secrets, and sensitive client data. In the event of physical theft, unauthorized media access, or backup tape loss, this policy is applicable to avoid data breaches. It is necessary to utilize either filesystem-level encryption (such as BitLocker or LUKS) or database-level encryption. In-flight encryption
Encryption in flight	[Insert text.] When data travels across a network, it is encrypted. How/Why: Robust cryptographic methods (such as TLS 1.2 or higher) must be used for all client-server and internal microservices communication. The purpose of this policy is to guard against data manipulation during transmission, man-in-the-middle attacks, and eavesdropping. HTTPS

a. **Encryption**	**Explain what it is and how and why the policy applies.**
	must be used for internal HTTP traffic as well.
Encryption in use	[Insert text.] While data is being processed in memory, it is encrypted. How/Why: This sophisticated control guards against cold-boot and memory-scraping attacks. As required by certain data classification regulations, it must be utilized for processing extremely sensitive data (such as cryptographic keys and top-secret data) using technologies like Intel SGX or other confidential computing platforms, even though its use is not universal due to performance overhead.

b. **Triple-A Framework***	**Explain what it is and how and why the policy applies.**
Authentication	[Insert text.] The procedure for confirming a user, system, or other entity's identification. How/Why: Green Pace applications require authentication for all user and system access. All administrative access and applications handling sensitive data must use multi-factor authentication (MFA). This policy serves as the cornerstone of access control, guaranteeing that only authentic, validated identities have access to our systems. This includes adding new users and logging in as a user.
Authorization	[Insert text.] the procedure for figuring out what rights an authenticated identity possesses. How/Why: The least privilege principle must be followed while granting access. Effective permission management will be achieved by the implementation of Role-Based Access Control (RBAC). Every request to access a resource or carry out an operation needs to pass authorization checks. By specifying the user degree of access and the files [are] viewed by users, this policy makes sure users can only access the information and features that are absolutely required for their role.
Accounting	[Insert text.] the procedure for recording and keeping track of system and user activity. How/Why: Every occurrence that is important to security needs to be recorded. Logs must be kept for at least a year in a central location in a safe, unchangeable format. Successful and unsuccessful login attempts, database modifications (particularly those involving schema and permission changes), privilege escalations, and access to sensitive data records are all examples of this. This policy is applicable to provide a trail for the detection of harmful conduct, compliance evidence, and forensic analysis.

***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

Standard	Principle(s)	Justification for Connection
STD-001-CPP	4, 9	Strong typing simplifies code (4) and prevents type-related errors, making QA more effective (9).
STD-002-CPP	1	This is a direct application of validating input data, ensuring arithmetic results are within safe bounds (1).
STD-003-CPP	1, 8	Bounds checking validates input (1), and using safe string classes provides a layer of defense against memory corruption (8).
STD-004-CPP	1, 7	Parameterized queries are the most robust form of sanitizing (7) data sent to another system (the database), which is a form of input validation (1).
STD-005-CPP	4, 9	Smart pointers simplify memory management (4), eliminating entire categories of bugs that are hard to find via QA (9).
STD-006-CPP	1, 8	Ensures runtime input validation is always active (1), while assertions act as a development-time depth check for internal logic (8).
STD-007-CPP	4, 9	Simplifies exception handling (4) and prevents subtle bugs like memory leaks, which are difficult to identify during testing (9).
STD-008-CPP	1, 5, 6	Validates and restricts file paths (1) to a specific, allowed directory, enforcing default deny (5) and least privilege (6) on file ops.
STD-009-CPP	8, 9	Proper locking is a fundamental layer of defense in concurrent systems (8). Data races are hard to detect with QA, so prevention is key (9).
STD-010-CPP	9	Handling errors explicitly makes the code's behavior predictable and testable, which is a cornerstone of effective QA (9).

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	
[Insert text.] 2.0	[Insert text.] 10/27/2023	[Insert text.] Full Policy Implementation: Added 10 Principles, 10 C++ Standards, Risk Assessments, Automation, Encryption & AAA Policies.	[Insert text.] Steven Foltz	[Insert text.] N/A
[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