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	Ensure that incoming data meets expected criteria to prevent potential security risks or system vulnerabilities.
2. Heed Compiler Warnings	Pay attention to warnings issued by the compiler during code development, as they often indicate areas where security issues may arise and need to be addressed.
3. Architect and Design for Security Policies	Plan and create software systems with security rules in mind from the start, so they're built securely from the ground up.
4. Keep It Simple	Make things as simple as possible, which reduces the chances of mistakes or vulnerabilities sneaking into the system.
5. Default Deny	Start with a mindset of declining any access requests unless they're explicitly allowed, which helps keep unauthorized users out.
6. Adhere to the Principle of Least Privilege	Only give users or programs the minimum amount of access they need to do their jobs, so if something goes wrong, the damage is limited.
7. Sanitize Data Sent to Other Systems	Clean and check data before sending it elsewhere, to prevent sending harmful or sensitive information accidentally.
8. Practice Defense in Depth	Use multiple layers of security measures to protect against different types of threats, making it harder for attackers to succeed.
9. Use Effective Quality Assurance Techniques	Test software thoroughly to find and fix security flaws before they can be exploited by attackers.

Principles	Write a short paragraph explaining each of the 10 principles of security.
10. Adopt a Secure Coding Standard	Follow established guidelines for writing code securely, to make sure that your software is as safe as possible from potential threats.

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]	Never qualify a reference type with const or volatile.

Noncompliant Code

This code attempts to const-qualify a reference type, resulting in undefined behavior as per the C++ standard.

```
char &const p;
```

Compliant Code

The const qualifier is correctly placed before the reference type to ensure the integrity of the code.

```
const char &p;
```

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

Principles(s): Correctness. By adhering to this standard, we ensure that the code behaves as expected and follows the rules defined by the C++ standard, preventing undefined behavior and maintaining correctness.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	P3	L3

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC++-DCL52	Detects and reports violations of DCL52-CPP - Never qualify a reference type with 'const' or 'volatile'.
Helix QAC	2023.3	C++0014	Ensures compliance with rule DCL52-CPP.
Klocwork	2023.3	CERT.DCL.REF_TYPE.CONST_OR_VOLATILE	Identifies const-qualified reference types and modification attempts.
Parasoft C/C++test	2023.1	CERT_CPP-DCL52-a	Analyzes code to identify instances of qualifying reference types with 'const' or 'volatile'.

Tool	Version	Checker	Description Tool
Polyspace Bug Finder	R2023b	-	Assesses code for const-qualified reference types and modifications, ensuring compliance with DCL52-CPP.
Clang	3.9	-	Produces error messages for violations of the rule without requiring special flags or options.
SonarQube C/C++ Plugin	4.10	S3708	Utilizes rule S3708 to detect and report violations related to const-qualifying reference types.

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	Do not read uninitialized memory.

Noncompliant Code

Reading uninitialized memory, which leads to undefined behavior.

```
#include <iostream>

void f() {
  int i;
  std::cout << i;
}
```

Compliant Code

Initializing variables before accessing their values.

```
#include <iostream>

void f() {
  int i = 0;
  std::cout << i;
}
```

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

Principles(s): This standard aligns with the principle of ensuring well-defined program behavior by initializing variables before use, thus preventing undefined behavior caused by reading uninitialized memory.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	P12	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	uninitialized-read	Partially checked

Tool	Version	Checker	Description Tool
Clang	3.9	Uninitialized	Flag for uninitialized variables (not exhaustive).
CodeSonar	8.0p0	LANG.STRUCT.RPL, LANG.MEM.UVAR	Checks for return pointer to local, uninitialized variable.
Helix QAC	2023.3	DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978	Rules targeting uninitialized variables.
Klocwork	2023.3	UNINIT.CTOR.MIGHT, UNINIT.CTOR.MUST, UNINIT.HEAP.MIGHT, UNINIT.HEAP.MUST, UNINIT.STACK.ARRAY.MIGHT, UNINIT.STACK.ARRAY.MUST, UNINIT.STACK.ARRAY.PARTIAL.MUST, UNINIT.STACK.MIGHT, UNINIT.STACK.MUST	Rules for detecting uninitialized variables.
LDRA tool suite	9.7.1	-	Partially implemented.
Parasoft C/C++test	2023.1	CERT_CPP-EXP53-a	Aims to avoid use before initialization.
Polyspace Bug Finder	R2023b	CERT C++: EXP53-CPP	Checks for non-initialized variable, non-initialized pointer (partially covered).
PVS-Studio	7.29	V546, V573, V614, V670, V679, V730, V788, V1007, V1050	Rules targeting uninitialized variables.
RuleChecker	22.10	uninitialized-read	Partially checked.
Astrée	22.10	uninitialized-read	Partially checked.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	Use correct integer precisions.

Noncompliant Code

Incorrect usage of integer precision calculation, leading to potential undefined behavior due to incorrect assumptions about the size of integer types.

```
#include <limits.h>

unsigned int pow2(unsigned int exp) {
  if (exp >= sizeof(unsigned int) * CHAR_BIT) {
    /* Handle error */
  }
  return 1 << exp;
}
```

Compliant Code

Correct determination of integer precision using a popcount() function to avoid undefined behavior.

```
#include <stddef.h>
#include <stdint.h>

/* Returns the number of set bits */
size_t popcount(uintmax_t num) {
  size_t precision = 0;
  while (num != 0) {
    if (num % 2 == 1) {
      precision++;
    }
    num >>= 1;
  }
  return precision;
}
#define PRECISION(umax_value) popcount(umax_value)

unsigned int pow2(unsigned int exp) {
  if (exp >= PRECISION(UINT_MAX)) {
    /* Handle error */
  }
  return 1 << exp;
```

Compliant Code

```
}
```

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

Principles(s): This standard ensures correct handling of integer precision to avoid undefined behavior and ensure the safety and correctness of integer manipulation operations. By accurately determining integer precision, the code becomes more reliable and resilient, preventing potential vulnerabilities and bugs.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	23.04	Supported	Reports overflows due to insufficient precision.
CodeSonar	8.0p0	LANG.ARITH.BIGSHIFT	Checks if shift amount exceeds the bit width.
Helix QAC	2023.4	C0582, C++3115	Checks for compliance with the standard regarding integer precisions.
Parasoft C/C++test	2023.1	CERT_C-INT35-a	Ensures the use of correct integer precisions when checking the right-hand operand of shifts.
Polyspace Bug Finder	R2023b	CERT C: Rule INT35-C	Checks for situations when integer precisions are exceeded.

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	Guarantee that library functions do not overflow.

Noncompliant Code

Copying data into a container without ensuring its sufficient capacity may result in a buffer overflow vulnerability.

```
#include <algorithm>
#include <vector>

void f(const std::vector<int> &src) {
  std::vector<int> dest;
  std::copy(src.begin(), src.end(), dest.begin());
  // ...
}
```

Compliant Code

Ensuring the destination container has sufficient capacity before copying data into it.

```
#include <algorithm>
#include <vector>
void f(const std::vector<int> &src) {
  // Initialize dest with src.size() default-inserted elements
  std::vector<int> dest(src.size());
  std::copy(src.begin(), src.end(), dest.begin());
  // ...
}
```

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

Principles(s): This standard aligns with the principle of ensuring memory safety by preventing buffer overflow vulnerabilities. It emphasizes the importance of validating container sizes before copying data into them, thus mitigating the risk of buffer overflow exploits.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	invalid_pointer_dereference	Reports instances of invalid pointer dereference.
CodeSonar	8.0p0	BADFUNC.BO.* LANG.MEM.BO LANG.MEM.TBA	A collection of warning classes that report uses of library functions prone to internal buffer overflows. Buffer Overrun. Tainted Buffer Access.
Helix QAC	2023.3	DF3526, DF3527, DF3528, DF3529, DF3530, DF3531, DF3532, DF3533, DF3534	Collection of warning classes that report various buffer overrun issues.
Parasoft C/C++test	2023.1	CERT_CPP-CTR52-a	Do not pass empty container iterators to std algorithms as destinations.
Polyspace Bug Finder	R2023b	CERT C++: CTR52-CPP	Checks for library functions overflowing sequence container (rule partially covered).

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	Guarantee that storage for strings has sufficient space for character data and the null terminator.

Noncompliant Code

Using a fixed-size character array buf to store input from std::cin. This approach can lead to buffer overflow if the input exceeds the size of the array.

```
#include <iostream>

void f() {
  char buf[12];
  std::cin >> buf;
}
```

Compliant Code

Using std::string which automatically manages the memory allocation for the input string, eliminating the risk of buffer overflow.

```
#include <iostream>
#include <string>

void f() {
  std::string input;
  std::string stringOne, stringTwo;
  std::cin >> stringOne >> stringTwo;
}
```

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

Principles(s): Ensure memory safety by using appropriate data structures that handle memory management automatically, such as std::string in C++, instead of manually managing memory with fixed-size arrays. This helps prevent buffer overflows and other memory-related vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	stream-input-char-array	Partially checked + soundly supported.
CodeSonar	8.0p0	MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO	No space for null terminator. Buffer overrun. Type overrun.
LDRA tool suite	9.7.1	489 S, 66 X, 70 X, 71 X	Partially implemented.
Parasoft C/C++test	2023.1	CERT_CPP-STR50-b CERT_CPP-STR50-c CERT_CPP-STR50-e CERT_CPP-STR50-f CERT_CPP-STR50-g	Avoid overflow due to reading a not zero terminated string. 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	R2023b	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.
RuleChecker	22.10	stream-input-char-array	Partially checked.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	Detect and handle memory allocation errors.

Noncompliant Code

Failing to check the return value of the memory allocation operation using ::operator new. If the allocation fails, it can throw a std::bad_alloc exception, which may result in abnormal termination of the program. The function is marked as noexcept, suggesting that it does not throw any exceptions, which can mislead the caller.

```
#include <cstring>

void f(const int *array, std::size_t size) noexcept {
  int *copy = new int[size];
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

Compliant Code

Using std::nothrow with the new operator, ensuring that it returns either a null pointer or a pointer to the allocated space. It checks the returned pointer to handle the error condition appropriately when the allocation fails and the returned pointer is nullptr. This approach prevents abnormal termination of the program and ensures proper error handling.

```
#include <cstring>
#include <new>

void f(const int *array, std::size_t size) noexcept {
  int *copy = new (std::nothrow) int[size];
  if (!copy) {
    // Handle error
    return;
  }
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

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

Principles(s): Emphasize proper error handling and robustness when dealing with memory allocation operations. By checking the return value of memory allocation functions, such as ::operator new, and handling error conditions appropriately, the code becomes more resilient and less prone to unexpected program termination. This approach ensures that potential memory allocation failures are gracefully handled, improving the reliability and stability of the software.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Coverity	7.5	CHECKED_RETURN	Finds inconsistencies in how function call return values are handled.
LDRA tool suite	9.7.1	45 D	Partially implemented.
Parasoft C/C++test	2023.1	CERT_CPP-MEM52-a CERT_CPP-MEM52-b	Check the return value of new. Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined.
Polyspace Bug Finder	R2023b	CERT C++: MEM52-CPP	Checks for unprotected dynamic memory allocation (rule partially covered).

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP	Close files when they are no longer needed.

Noncompliant Code

Failing to close the file before invoking std::terminate(), leading to improper management of file resources and potential resource leaks.

```
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
  // ...
  std::terminate();
}
```

Compliant Code

Ensuring that the file is properly closed before invoking std::terminate(), preventing resource leaks and ensuring proper resource management.

```
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
  // ...
  file.close();
  if (file.fail()) {
    // Handle error
  }
```

Compliant Code

```
  std::terminate();
}
```

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

Principles(s): This standard emphasizes the importance of properly managing file resources by ensuring that files are closed before program termination. By calling std::fstream::close() before invoking std::terminate(), the compliant code adheres to this principle, preventing potential resource leaks and ensuring proper cleanup of file resources.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.0p0	ALLOC.LEAK	Leak.
Parasoft C/C++test	2023.1	CERT_CPP-FIO51-a	Ensure resources are freed.
Polyspace Bug Finder	R2023b	CERT C++: FIO51-CPP	Checks for resource leak (rule partially covered).

Coding Standard 8

Coding Standard	Label	Name of Standard
[Student Choice]	[STD-008-CPP]	Detect errors when converting a string to a number.

Noncompliant Code

Failing to check for errors during string-to-number conversion, potentially resulting in unexpected values or uninitialized variables.

```
#include <iostream>

void f() {
  int i, j;
  std::cin >> i >> j;
  // ...
}
```

Compliant Code

Enabling exceptions and checks for conversion errors during string-to-number conversion, ensuring proper error handling and preventing unexpected behavior.

```
#include <iostream>

void f() {
  int i, j;

  std::cin.exceptions(std::istream::failbit | std::istream::badbit);
  try {
    std::cin >> i >> j;
    // ...
  } catch (std::istream::failure &E) {
    // Handle error
  }
}
```

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

Principles(s): This standard stresses the need to handle errors during string-to-number conversions. By checking for errors or handling exceptions, the code ensures reliable error handling and prevents unexpected issues or crashes, aligning with good programming practices for reliability and safety.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Clang	3.9	cert-err34-c	Checked by clang-tidy; only identifies use of unsafe C Standard Library functions corresponding to ERR34-C.
CodeSonar	8.0p0	BADFUNC.ATOF BADFUNC.ATOI BADFUNC.ATOL BADFUNC.ATOLL	Use of atof. Use of atoi. Use of atoll. Use of atoll.
Parasoft C/C++test	2023.1	CERT_CPP-ERR62-a	The library functions atof, atoi and atol from library stdlib.h shall not be used.
Polyspace Bug Finder	R2023b	CERT C++: ERR62-CPP	Checks for unvalidated string-to-number conversion (rule fully covered).

Coding Standard 9

Coding Standard	Label	Name of Standard
[Student Choice]	[STD-009-CPP]	Write constructor member initializers in the canonical order.

Noncompliant Code

The member initializer list in the constructor of class C attempts to initialize someVal after dependsOnSomeVal, leading to undefined behavior as dependsOnSomeVal relies on the value of someVal.

```
class C {
  int dependsOnSomeVal;
  int someVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

Compliant Code

The member variables are reordered so that the constructor's member initializer list correctly follows the canonical order, ensuring proper initialization and avoiding undefined behavior.

```
class C {
  int someVal;
  int dependsOnSomeVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

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

Principles(s): This standard ensures that member initializers are written in the canonical order to guarantee correct and predictable behavior during object initialization. By following the canonical order, the code avoids undefined behavior and ensures that member variables are properly initialized, contributing to the correctness and predictability of the program's behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Green Pace

Tool	Version	Checker	Description Tool
Astrée	22.10	initializer-list-order	Fully checked.
CodeSonar	8.0p0	LANG.STRUCT.INIT.OOMI	Out of Order Member Initializers.
LDRA tool suite	9.7.1	206 S	Fully implemented.
Parasoft C/C++test	2023.1	CERT_CPP-OOP53-a	List members in an initialization list in the order in which they are declared.
Polyspace Bug Finder	R2023B	CERT C++: OOP53-CPP	Checks for members not initialized in canonical order (rule fully covered).
RuleChecker	22.10	initializer-list-order	Fully checked.

Coding Standard 10

Coding Standard	Label	Name of Standard
[Student Choice]	[STD-010-CPP]	Preserve thread safety and liveness when using condition variables.

Noncompliant Code

Attempting to sequentially execute multiple threads based on their assigned step levels. However, it shares a single condition variable among all threads and may result in a violation of the liveness property due to potential deadlock scenarios.

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
    cond.wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing...
  std::cout << "Thread " << myStep << " is processing..." << std::endl;
  currentStep++;

  // Signal awaiting task.
  cond.notify_one();

  std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}

int main() {
  constexpr size_t numThreads = 5;
  std::thread threads[numThreads];
```

Noncompliant Code

```
  // Create threads.
  for (size_t i = 0; i < numThreads; ++i) {
    threads[i] = std::thread(run_step, i);
  }

  // Wait for all threads to complete.
  for (size_t i = numThreads; i != 0; --i) {
    threads[i - 1].join();
  }
}
```

Compliant Code

Modifying the code to use notify_all() instead of notify_one(), ensuring that all waiting threads are signaled appropriately. This adjustment enhances liveness by preventing potential deadlock situations and preserving the thread safety property.

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
    cond.wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing ...
  std::cout << "Thread " << myStep << " is processing..." << std::endl;
  currentStep++;

  // Signal ALL waiting tasks.
  cond.notify_all();

  std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}
```

Compliant Code

```
// ... main() unchanged ...
```

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

Principles(s): Ensure thread safety and liveness by appropriately using condition variables to avoid deadlocks and guarantee completion of operations in multithreaded environments.

Threat Level

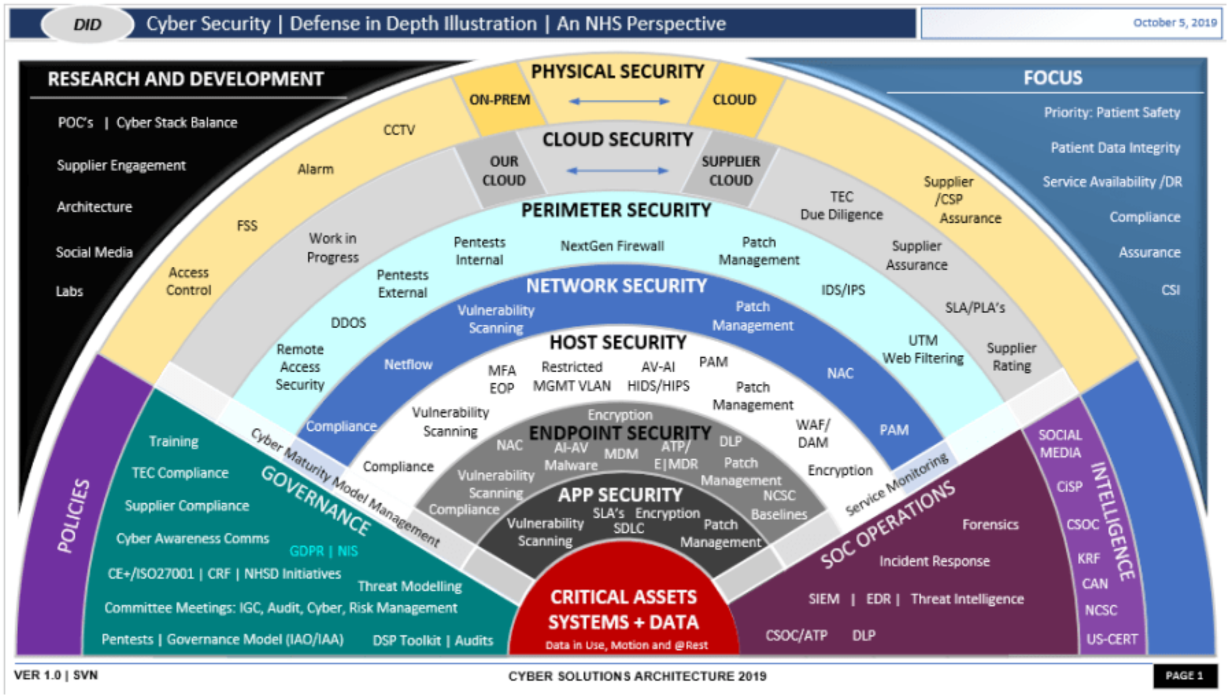
Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.0p0	CONCURRENCY.BADFUNC.CNDSIGNAL	Use of condition variable signal.
Parasoft C/C++test	2023.1	CERT_CPP-CON55-a	Do not use the 'notify_one()' function when multiple threads are waiting on the same condition variable.
Polyspace Bug Finder	R2023b	CERT C++: CON55-CPP	Checks for multiple threads waiting for same condition variable (rule fully covered).

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

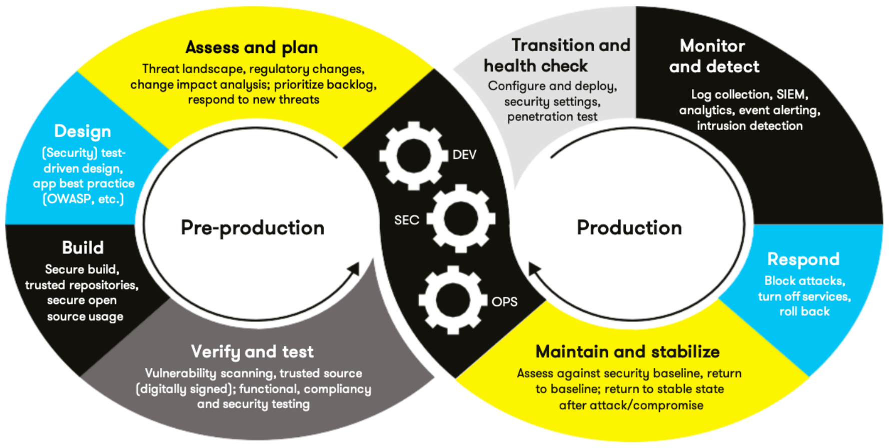
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

Integrating automation into the DevOps process ensures that security standards are consistently applied without the need for manual oversight. Automated tools can scan for vulnerabilities during the design phase, verify the security of components in the build stage, and continuously test for weaknesses before deployment. In the live environment, automated monitoring systems vigilantly track activities to detect security threats. Should an issue arise, automated mechanisms are in place to swiftly address and resolve the problem, maintaining system stability and upholding security protocols efficiently.

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	Low	Unlikely	Low	P3	L3
STD-002-CPP	High	Probable	Medium	P12	L1
STD-003-CPP	Low	Unlikely	Medium	P2	L3
STD-004-CPP	High	Likely	Medium	P18	L1
STD-005-CPP	High	Likely	Medium	P18	L1
STD-006-CPP	High	Likely	Medium	P18	L1
STD-007-CPP	Medium	Unlikely	Medium	P4	L3
STD-008-CPP	Medium	Unlikely	Medium	P4	L3
STD-009-CPP	Medium	Unlikely	Medium	P4	L3
STD-010-CPP	Low	Unlikely	Medium	P2	L3

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	This means keeping data safe when it's stored, like on a computer or a server. We do this by turning the data into a secret code that only authorized people can understand. We use this policy to make sure that if someone tries to steal our data, they can't read it because it's all scrambled up.
Encryption in flight	This is about protecting data while it's moving from one place to another, like when we send emails or browse the internet. We use special codes to keep the information safe while it's traveling through networks. This policy helps to keep our information private and secure, so nobody can spy on it while it's on its way.
Encryption in use	This is about protecting data while it's being used by programs or stored in computer memory. We use encryption to hide the information so that even if someone tries to peek at it, they can't understand what it means. This policy is important to keep sensitive information safe while it's being processed by computers or applications.

b. **Triple-A Framework***	**Explain what it is and how and why the policy applies.**
Authentication	Authentication verifies the identity of users before allowing them access to systems or data. This ensures that only legitimate users can log in, preventing unauthorized access to sensitive information. The policy applies by requiring secure login methods, such as passwords or biometrics, and monitoring user logins to detect any suspicious activities or unauthorized attempts to access the system.
Authorization	Authorization determines what actions authenticated users can perform within the system based on their roles or privileges. This includes controlling access to various resources, such as files, databases, or applications. The policy applies by assigning appropriate access levels to users, ensuring they only have access to the information and functionalities necessary for their roles. Additionally, it monitors user activities to detect any unauthorized attempts to access or modify data.
Accounting	Accounting involves tracking and recording user activities, including logins, changes to the database, addition of new users, user level of access, and files accessed by users. This helps maintain a comprehensive audit trail of system activities, facilitating accountability and compliance with regulatory requirements. The policy applies by implementing logging mechanisms to capture relevant user actions and regularly reviewing these logs to identify any anomalies or security breaches.

*****Use this checklist for the Triple A to be sure you include these elements in your policy:

- User logins

- Changes to the database
- Addition of new users
- User level of access
- Files accessed by users

Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it's time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

NOTE: Green Pace has already successfully implemented the following:

- Operating system logs
- Firewall logs
- Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

- Code compliance to standards
- Well-documented access-control strategies, with sampled evidence of compliance
- Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
- Historical evidence of sustained practice (emails, logs, audits, meeting notes)

Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

- Business or technical rationale
- Risk impact analysis
- Risk mitigation analysis
- Plan to come into compliance
- Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

Distribution
This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

Policy Change Control
This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
2.0	04/14/2024	In this task, the focus is on consolidating and refining coding standards based on the SEI CERT C++ Coding Standard. Risks associated with each standard will be assessed, automated detection methods determined, and guidance provided on integrating automation into the DevOps process. Additionally, policies for encryption and the Triple-A framework will be created, aligning them with previously established principles for a comprehensive security policy.	Hong Luu	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV

Reference

CON55-CPP. Preserve thread safety and liveness when using condition variables - SEI CERT C++ Coding Standard - Confluence. (n.d.). Wiki.sei.cmu.edu. Retrieved March 24, 2024, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON55-CPP.+Preserve+thread+safety+and+liveness+when+using+condition+variables