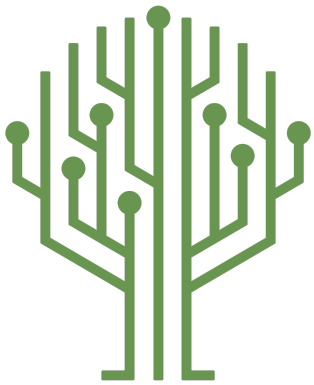


Green Pace

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	Validate Input Data prevents mistakes or malicious activity, ensuring that every input data is validated for accuracy, completeness, and correctness before it is utilized in the system.
2. Heed Compiler Warnings	Heed Compiler Warnings considers any warnings or problems generated by the compiler carefully, since they may represent vulnerabilities that attackers may exploit.
3. Architect and Design for Security Policies	To guarantee that security needs are satisfied, develop and build the system with security rules such as access control, authentication, and authorization in mind.
4. Keep It Simple	To avoid errors and vulnerabilities caused by complexity, use a basic and comprehensible design.
5. Default Deny	To guarantee that only authorized users have access to the system, the system should default to refusing access, and users should be specifically granted authorization.
6. Adhere to the Principle of Least Privilege	Users should be provided just the access necessary to complete their jobs, decreasing the danger of inadvertent or purposeful exploitation of system resources.
7. Sanitize Data Sent to Other Systems	Data transferred to other systems should be sanitized to prevent assaults like injection attacks, which involve injecting malicious code into a system.
8. Practice Defense in Depth	A layered approach to security, with various levels of security controls such as firewalls, intrusion detection, and access restrictions, should be employed.

Principles	Write a short paragraph explaining each of the 10 principles of security.
9. Use Effective Quality Assurance Techniques	To guarantee that the system is safe and free of vulnerabilities, quality assurance procedures such as code review and testing should be employed.
10. Adopt a Secure Coding Standard	To guarantee that every code created follows established security practices and standards, a secure coding standard should be developed.

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]	Do not cast to an out-of range enumeration value

Noncompliant Code

This noncompliant code sample attempts to determine whether a supplied number is inside the permitted enumeration value range.

```
enum EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  EnumType enumVar = static_cast<EnumType>(intVar);

  if (enumVar < First || enumVar > Third) {
    // Handle error
  }
}
```

Compliant Code

To ensure that the conversion does not result in an undefined value, this conforming solution validates that the value can be represented by the enumeration type before completing the conversion.

```
 enum EnumType {

   First,

   Second,

   Third

 };
```

Compliant Code

```
void f(int intVar) {

  if (intVar < First || intVar > Third) {

    // Handle error

  }

  EnumType enumVar = static_cast<EnumType>(intVar);

}
```

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

Principles(s): Enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, making buffer overflows more likely to result in data integrity violations than code execution.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**cast-integer-to-enum**	Partially checked
CodeSonar	7.3p0	**LANG.CAST.COERCE** **LANG.CAST.VALUE**	Coercion Alters Value Cast Alters Value
Parasoft C/C++test	2022.2	**CERT_CPP-INT50-a**	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration
RuleChecker	22.10	**cast-integer-to-enum**	Partially checked

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	Do not declare or define a reserved identifier

Noncompliant Code

A popular technique is to employ a macro in a preprocessor conditional to prevent duplicate header file additions. While this is a good idea, many applications employ reserved names as header guards. This name may conflict with reserved names specified by the C++ standard template library implementation in its headers, or with reserved names automatically predefined by the compiler even when no C++ standard library header is provided.

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_

// Contents of <my_header.h>

#endif // _MY_HEADER_H_
```

Compliant Code

This conforming method prevents the use of leading or trailing underscores in the header guard's name.

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Contents of <my_header.h>

#endif // MY_HEADER_H
```

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

Principles(s): Incorrect use of a variadic function can lead to unexpected program termination, accidental information exposure, or unauthorized code execution.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	P3	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**function-ellipsis**	Fully checked
Clang	3.9	`cert-dcl50-cpp`	Checked by `clang-tidy`.
CodeSonar	7.3p0	**LANG.STRUCT.ELLIPSIS**	Ellipsis
Parasoft C/C++test	2022.2	**CERT_CPP-DCL50-a**	Functions shall not be defined with a variable number of arguments

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	Never qualify a reference type with const or volatile

Noncompliant Code

Instead of a reference to a const-qualified char, a const-qualified reference to a char is produced in this noncompliant code sample. As a result, the behavior is undefined.

```
#include <iostream>

void f(char c) {
  char &const p = c;
  p = 'p';
  std::cout << c << std::endl;
}
```

Compliant Code

The const qualifier is removed in this compatible solution.

```
#include <iostream>

void f(char c) {
  char &p = c;
  p = 'p';
  std::cout << c << std::endl;
}
```

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

Principles(s): A const or volatile reference type may provide undefined behavior instead of a catastrophic diagnostic, resulting in the storage of unexpected values and potential data integrity violations.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	P3	L3

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++test	2022.2	**CERT_CPP-DCL52-a**	Never qualify a reference type with 'const' or 'volatile'
Polyspace Bug Finder	R2023a	CERT C++: DCL52-CPP	Checks for: • const-qualified reference types • Modification of const-qualified reference types Rule fully covered.
Clang	3.9		Clang checks for violations of this rule and produces an error without the need to specify any special flags or options.
SonarQube C/C++ Plugin	4.10	**S3708**	NA

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	Do not write syntactically ambiguous declaration

Noncompliant Code

By virtue of RAII, an anonymous local variable of type std::unique lock is intended to lock and unlock the mutex m in this noncompliant code sample. The declaration, however, is syntactically confusing since it might be viewed as defining an anonymous object and invoking its single-argument converting constructor or as declaring an object called m and default creating it. Because the syntax used in this example declares the latter rather than the former, the mutex object is never locked.

```
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex>(m);
  shared_resource += 42;
}
```

Compliant Code

The lock object is given an identifier (different than m) in this compatible solution, and the appropriate conversion constructor is called.

```
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex> lock(m);
  shared_resource += 42;
}
```

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

Principles(s): Declaratives that are syntactically ambiguous might result in unexpected program execution. However, simple testing is likely to reveal breaches of this criterion.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.3p0	**LANG.STRUCT.DECL.FNEST**	Nested Function Declaration
LDRA tool suite	9.7.1	**296 S**	Partially implemented
Parasoft C/C++test	2022.2	**CERT_CPP-DCL53-a** **CERT_CPP-DCL53-b** **CERT_CPP-DCL53-c**	Parameter names in function declarations should not be enclosed in parentheses Local variable names in variable declarations should not be enclosed in parentheses Avoid function declarations that are syntactically ambiguous
Polyspace Bug Finder	R2023a	CERT C++: DCL53-CPP	Checks for declarations that can be confused between: • Function and object declaration • Unnamed object or function parameter declaration Rule fully covered.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	Overload allocation and deallocation functions as a pair in the same scope

Noncompliant Code

An allocation function is overloaded at global scope in this noncompliant code sample. The necessary deallocation function, however, is not defined. If an object is allocated via the overloaded allocation function, any attempt to delete it will result in undefined behavior, which is a violation of MEM51-CPP. Deallocate dynamically allocated resources correctly.

```
#include <Windows.h>
#include <new>

void *operator new(std::size_t size) noexcept(false) {
  static HANDLE h = ::HeapCreate(0, 0, 0); // Private, expandable heap.
  if (h) {
    return ::HeapAlloc(h, 0, size);
  }
  throw std::bad_alloc();
}

// No corresponding global delete operator defined.
```

Compliant Code

The equivalent deallocation function is likewise defined at the global level in this compatible solution.

```
#include <Windows.h>
#include <new>

class HeapAllocator {
  static HANDLE h;
  static bool init;

public:
  static void *alloc(std::size_t size) noexcept(false) {
```

Compliant Code

```
    if (!init) {
      h = ::HeapCreate(0, 0, 0); // Private, expandable heap.
      init = true;
    }

    if (h) {
      return ::HeapAlloc(h, 0, size);
    }
    throw std::bad_alloc();
  }

  static void dealloc(void *ptr) noexcept {
    if (h) {
      (void)::HeapFree(h, 0, ptr);
    }
  }
};

HANDLE HeapAllocator::h = nullptr;
bool HeapAllocator::init = false;

void *operator new(std::size_t size) noexcept(false) {
  return HeapAllocator::alloc(size);
}

void operator delete(void *ptr) noexcept {
  return HeapAllocator::dealloc(ptr);
}
```

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

Principles(s): Incorrect use of new and delete might result in a denial-of-service attack.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Low	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**new-delete-pairwise**	Partially checked
Clang	3.9	`misc-new-delete-overloads`	Checked with `clang-tidy`.
Parasoft C/C++test	2022.2	**CERT_CPP-DCL54-a**	Always provide new and delete together
Polyspace Bug Finder	R2023a	CERT C++: DCL54-CPP	Checks for mismatch between overloaded operator new and operator delete (rule fully covered)

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	Avoid information leakage when applying a class object across a trust boundary

Noncompliant Code

This example of noncompliant code runs in kernel space and moves data from arg to user space. Padding bits, on the other hand, may be utilized within the object, for example, to maintain appropriate alignment of class data members. These padding bits might include sensitive information, which could be exposed when the data is transferred to user space, independent of how the data is duplicated.

```
#include <cstddef>

struct test {
  int a;
  char b;
  int c;
};

// Safely copy bytes to user space
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
  test arg{1, 2, 3};
  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Compliant Code

Before moving the structure data to an untrusted environment, this compatible approach serializes it.

```
#include <cstddef>
#include <cstring>

struct test {
  int a;
  char b;
```

Compliant Code

```
  int c;
};

// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
  test arg{1, 2, 3};
  // May be larger than strictly needed.
  unsigned char buf[sizeof(arg)];
  std::size_t offset = 0;

  std::memcpy(buf + offset, &arg.a, sizeof(arg.a));
  offset += sizeof(arg.a);
  std::memcpy(buf + offset, &arg.b, sizeof(arg.b));
  offset += sizeof(arg.b);
  std::memcpy(buf + offset, &arg.c, sizeof(arg.c));
  offset += sizeof(arg.c);

  copy_to_user(usr_buf, buf, offset /* size of info copied */);
}
```

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

Principles(s): Padding bits may include sensitive data, such as references to kernel data structures or passwords, accidentally. A pointer to such a structure might be provided to other methods, potentially leaking information.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.3p0	**MISC.PADDING.POTB**	Padding Passed Across a Trust Boundary
Helix QAC	2023.1	**DF4941, DF4942, DF4943**	

Tool	Version	Checker	Description Tool
Parasoft C/C++test	2022.2	**CERT_CPP-DCL55-a**	A pointer to a structure should not be passed to a function that can copy data to the user space
Polyspace Bug Finder	R2023a	CERT C++: DCL55-CPP	Checks for information leakage due to structure padding (rule partially covered)

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	Avoid cycles during initialization of static objects

Noncompliant Code

This noncompliant example uses caching to construct an efficient factorial function. Because the static local array cache initialization contains recursion, the function's behavior is unclear, even though the recursion is not endless.

```
#include <stdexcept>


int fact(int i) noexcept(false) {
  if (i < 0) {
    // Negative factorials are undefined.
    throw std::domain_error("i must be >= 0");
  }

  static const int cache[] = {
    fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),
    fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),
    fact(12), fact(13), fact(14), fact(15), fact(16)
  };

  if (i < (sizeof(cache) / sizeof(int))) {
    return cache[i];
```

Noncompliant Code

```
   }


   return i > 0 ? i * fact(i - 1) : 1;

 }
```

Compliant Code

Instead of initializing the static local array cache, this complying approach uses zero-initialization to check whether each member of the array has been assigned a value yet and, if not, recursively computes its value. It then either returns the cached value or computes the value as needed.

```
#include <stdexcept>

int fact(int i) noexcept(false) {
   if (i < 0) {
    // Negative factorials are undefined.
    throw std::domain_error("i must be >= 0");
  }

  // Use the lazy-initialized cache.
  static int cache[17];
  if (i < (sizeof(cache) / sizeof(int))) {
    if (0 == cache[i]) {
      cache[i] = i > 0 ? i * fact(i - 1) : 1;
    }
    return cache[i];
  }

  return i > 0 ? i * fact(i - 1) : 1;
}
```

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

Principles(s): An attacker may be able to create a crash or denial of service by reentering a function recursively during the instantiation of one of its static objects. Accessing an uninitialized object can result in unexpected behavior owing to indeterminately ordered dynamic initialization.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.3p0	**LANG.STRUCT.INIT.CYCLE** **LANG.STRUCT.INIT.UNORDERED**	Initialization Cycle Unordered Initialization
Helix QAC	2023.1	**C++1552, C++1554, C++1704**	
LDRA tool suite	9.7.1	**6 D**	Enhanced Enforcement
Parasoft C/C++test	2022.2	**CERT_CPP-DCL56-a**	Avoid initialization order problems across translation units by replacing non-local static objects with local static objects

Coding Standard 8

Coding Standard	Label	Name of Standard
[Student Choice]	[STD-008-CPP]	Do not let exceptions escape from destructors or deallocation functions

Noncompliant Code

The class destructor in this noncompliant code sample does not fulfill the implicit noexcept guarantee because it may produce an exception even if it was called as a result of an exception being raised. As a result, it is stated as noexcept(false), yet it can still cause undefined behavior.

```
#include <stdexcept>

class S {
  bool has_error() const;

public:
  ~S() noexcept(false) {
    // Normal processing
    if (has_error()) {
      throw std::logic_error("Something bad");
    }
  }
};
```

Compliant Code

A destructor should behave the same manner whether or not an exception is active. This usually implies that it should only call actions that do not throw exceptions, or that it should manage all exceptions and not rethrow them (even implicitly). The explicit return statement in the SomeClass destructor distinguishes this complying approach from the preceding noncompliant code sample. This statement stops control from reaching the exception handler's conclusion. As a result, when a bad member is deleted, this handler will catch the exception generated by Bad::Bad(). It will also catch any exceptions raised within the function-try-compound block's statement, but the SomeClass destructor will not end by throwing an exception.

```
class SomeClass {
  Bad bad_member;
public:
  ~SomeClass()
```

Compliant Code

```
  try {
    // ...
  } catch(...) {
    // Catch exceptions thrown from noncompliant destructors of
    // member objects or base class subobjects.

    // NOTE: Flowing off the end of a destructor function-try-block causes
    // the caught exception to be implicitly rethrown, but an explicit
    // return statement will prevent that from happening.
    return;
  }
};
```

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

Principles(s): Attempting to throw exceptions from destructors or deallocation routines might lead to unexpected behavior, resulting in resource leaks or denial-of-service attacks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Medium	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**destructor-without-noexcept** **delete-without-noexcept**	Fully checked
CodeSonar	7.3p0	**LANG.STRUCT.EXCP.CATCH** **LANG.STRUCT.EXCP.THROW**	Use of catch Use of throw
LDRA tool suite	9.7.1	**453 S**	Partially implemented
Parasoft C/C++test	2022.2	**CERT_CPP-DCL57-a** **CERT_CPP-DCL57-b**	Never allow an exception to be thrown from a destructor, deallocation, and swap Always catch exceptions

Coding Standard 9

Coding Standard	Label	Name of Standard
[Student Choice]	[STD-009-CPP]	Do not modify the standard namespaces

Noncompliant Code

The declaration of x is added to the namespace std in this noncompliant code sample, leading in undefined behavior.

```
namespace std {
int x;
}
```

Compliant Code

This conforming method assumes the author intended to put the declaration of x in a namespace to avoid clashes with other global identifiers. Instead of being placed in the namespace std, the declaration is placed in a namespace with no reserved name.

```
namespace nonstd {
int x;
}
```

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

Principles(s): Changing the standard namespace might result in unexpected behavior in the C++ standard library.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	Medium	P6	L2

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	**CertC++-DCL58**	NA
CodeSonar	7.3p0	**LANG.STRUCT.DECL.SNM**	Modification of Standard Namespaces

Tool	Version	Checker	Description Tool
Parasoft C/C++test	2022.2	**CERT_CPP-DCL58-a**	Do not modify the standard namespaces 'std' and 'posix'
Polyspace Bug Finder	R2023a	CERT C++: DCL58-CPP	Checks for modification of standard namespaces (rule fully covered)

Coding Standard 10

Coding Standard	Label	Name of Standard
[Student Choice]	[STD-010-CPP]	Do not define an unnamed namespace in a header file

Noncompliant Code

The variable v is defined in an unnamed namespace within a header file and referenced from two independent translation units in this noncompliant code sample. Each translation unit publishes the current value of v and then replaces it with a new value. Due to the fact that v is specified in an unnamed namespace, each translation unit acts on its own instance of v, resulting in unexpected results.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
int v;
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

void f() {
  std::cout << "f(): " << v << std::endl;
  v = 42;
  // ...
}

// b.cpp
#include "a.h"
#include <iostream>

void g() {
  std::cout << "g(): " << v << std::endl;
```

Noncompliant Code

```
  v = 100;
}

int main() {
  extern void f();
  f(); // Prints v, sets it to 42
  g(); // Prints v, sets it to 100
  f();
  g();
}
```

Compliant Code

v is specified in just one translation unit but is externally visible to all translation units in this conforming approach, resulting in the intended outcome.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

extern int v;

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

int v; // Definition of global variable v

void f() {
  std::cout << "f(): " << v << std::endl;
  v = 42;
  // ...
}

// b.cpp
#include "a.h"
```

Compliant Code

```
#include <iostream>

void g() {
  std::cout << "g(): " << v << std::endl;
  v = 100;
}

int main() {
  extern void f();
  f(); // Prints v, sets it to 42
  g(); // Prints v, sets it to 100
  f(); // Prints v, sets it back to 42
  g(); // Prints v, sets it back to 100
}
```

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

Principles(s): Defining an unnamed namespace within a header file might result in data integrity breaches and performance issues, although it is unlikely to go unnoticed with adequate testing. Violations of one-definition rules result in undefined behavior.

Threat Level

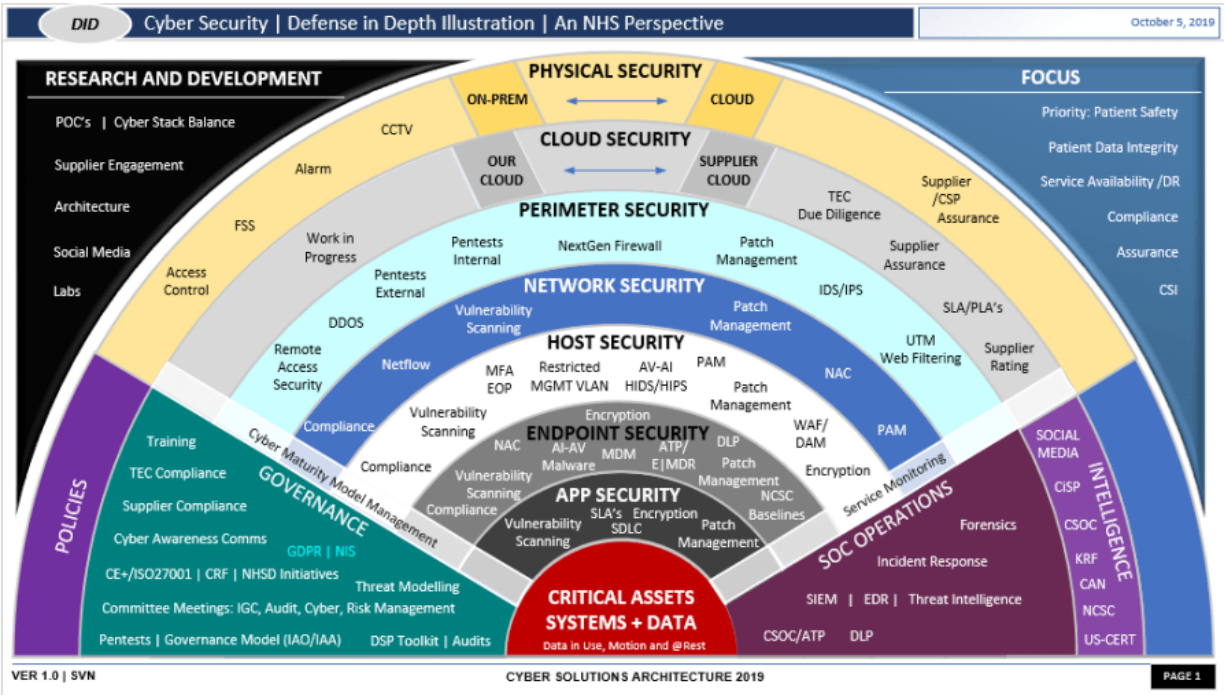
Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**unnamed-namespace-header**	Fully checked
Axivion Bauhaus Suite	7.2.0	**CertC++-DCL59**	NA
Clang	3.9	cert-dcl59-cpp	Checked by `clang-tidy`
CodeSonar	7.3p0	**LANG.STRUCT.DECL.ANH**	Anonymous Namespace in Header File

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

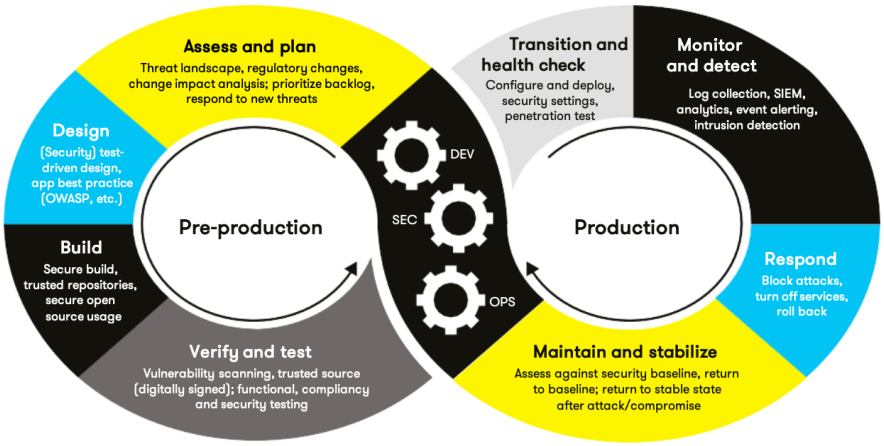
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

The choice to automate a process should be based on a comprehensive examination of the task's complexity, process flow, implementation costs, and possible advantages. Automation should ideally be deployed at the start of the process to have the most impact. However, it is critical to properly test the automation to verify that it functions as intended.

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	Unlikely	Medium	P4	L3
STD-002-CPP	Low	Unlikely	Low	P3	L3
STD-003-CPP	Low	Unlikely	Low	P3	L3
STD-004-CPP	Low	Unlikely	Medium	P2	L3
STD-005-CPP	Low	Probable	Low	P6	L2
STD-006-CPP	Low	Unlikely	High	P1	L3
STD-007-CPP	Low	Unlikely	Medium	P2	L3
STD-008-CPP	Low	Likely	Medium	P6	L2
STD-009-CPP	High	Unlikely	Medium	P6	L2
STD-010-CPP	Medium	Unlikely	Medium	P4	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 in rest	The encryption of data while it is at rest or held on a device or server is referred to as encryption in rest. All data saved on devices, servers, or other storage media should be protected with a recognized encryption technique. Encryption keys should be handled and safeguarded against unwanted access using a dedicated key management system. Encryption at rest is critical for preventing unwanted access to sensitive data if the data storage medium is stolen, lost, or compromised. Encryption guarantees that even if the data is obtained, it is incomprehensible without the encryption key, adding an extra degree of protection.
Encryption at flight	The encryption of data while it is in transit between two devices or servers is referred to as encryption at flight. All data sent across a network, internal or external, should be encrypted with an authorized encryption technique. Encryption keys should be handled and safeguarded against unwanted access using a dedicated key management system. Encryption in flight is critical for safeguarding sensitive data from interception and unwanted access while it is being sent over a network. Encryption guarantees that even if the data is intercepted, it is incomprehensible without the encryption key, adding an extra degree of protection.
Encryption in use	The encryption of data while it is being utilized or processed by a device or application is referred to as encryption in use. All data that a device or application uses or processes should be encrypted with an authorized encryption technique. Encryption keys should be handled and safeguarded against unwanted access using a dedicated key management system. Encryption in use is critical for preventing unwanted access to sensitive data if the device or application is hacked or compromised. Encryption guarantees that even if the data is obtained, it is incomprehensible without the encryption key, adding an extra degree of protection.

b. Triple-A Framework *	**Explain what it is and how and why the policy applies.**
Authentication	The process of authenticating the identity of a person or device seeking to access sensitive information is known as authentication. Verifying a login and password, biometric information, or a security token are all examples of this. Authentication is used to guarantee that sensitive information is only accessible to authorized users or devices. This is often accomplished through a login procedure in which users submit their credentials or use a security token. MFA (multi-factor authentication) can also be used to increase security. Any system or network that contains or processes sensitive information is subject to authentication procedures. It is vital to ensure that only authorized individuals or devices have access to sensitive information, since this can avoid data breaches and unauthorized access.
Authorization	The process of allowing or refusing access to specified resources or information depending on the user's identity, role, or other characteristics is known as authorization. Authorization ensures that users only have access to the resources or information they need to accomplish

b. Triple-A Framework *	Explain what it is and how and why the policy applies.
	their job tasks. Access controls, such as role-based access controls (RBAC) or attribute-based access controls, are commonly used to do this. (ABAC). Any system or network that contains or processes sensitive information is subject to authorization policies. It is vital to ensure that users have just the information they need to fulfill their job tasks, since this can avoid data breaches and unauthorized access to sensitive information.
Accounting	Accounting is the process of tracking and recording all sensitive information-related actions, such as who accessed the information, what they did with it, and when. Accounting is used to ensure that all sensitive information-related actions are tracked and recorded, which can be done through audit logs or other tracking mechanisms. Accounting policies apply to any system or network that stores or processes sensitive information.

*****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	
1.1	04/07/2023	Project One	Daniel Olson	Daniel Olson
[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