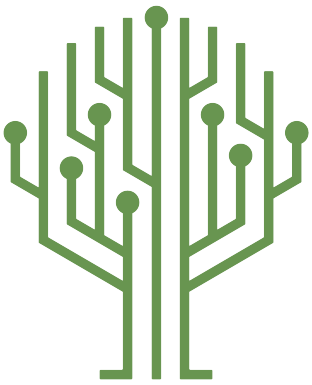


Green Pace

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	Ensures only valid data is entered into a system to prevent attackers from exploiting vulnerabilities and buffer overflow.
2. Heed Compiler Warnings	Compiler warnings report unusual conditions in code that may indicate a problem. Even though the compilation can continue, it is important to pay attention to each warning as going unattended could lead to as security risk.
3. Architect and Design for Security Policies	Looks into how information security controls are implemented in systems to protect data that is used, processed, and stored in those systems.
4. Keep It Simple	Avoiding complexity wherever possible to lessen the chance of errors and greater the chance of user acceptance.
5. Default Deny	If a request is not explicitly allowed, it is denied. This prevents unauthorized access that could have huge security risks.
6. Adhere to the Principle of Least Privilege	Principle of Least Privilege limits a user's access rights to only what is required for them to complete their job.
7. Sanitize Data Sent to Other Systems	Process of removing harmful elements to ensure a safe and uncompromised system. All unused sensitive data will be cleared as soon as a storage device is no longer in use.
8. Practice Defense in Depth	Strategy that uses multiple layers of protection to minimize the risk of a security breach.

Principles	Write a short paragraph explaining each of the 10 principles of security.
9. Use Effective Quality Assurance Techniques	Fuzz testing, penetration testing, and source code audits are all incorporated as part of an effective quality assurance program to effectively identify and eliminate vulnerabilities.
10. Adopt a Secure Coding Standard	Govern the coding practices, techniques, and decisions that developers make while building software, aiming to ensure that developers write code that minimizes security 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	Obey the One-Definition Rule
Data Type	[STD-001-CPP]	Nontrivial C++ programs are generally divided into multiple translation units that are later linked together to form an executable. To support such a model, C++ restricts named object definitions to ensure that linking will behave deterministically by requiring a single definition for an object across all translation units.

Noncompliant Code

Two different translation units define a class of the same name with differing definitions.

```
// a.cpp
struct S {
  int a;
};

// b.cpp
class S {
public:
  int a;
};
```

Compliant Code

Use of a header file to introduce the object into both translation units.

```
// S.h
struct S {
  int a;
};

// a.cpp
#include "S.h"

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

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

Principles(s):
3—Architect and Design for Security Policies
4—Keep It Simple
10—Adopt a Secure Coding Standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	High	P3	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	type-compatibility definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading type-file-spreading	Partially checked
LDRA tool suite	9.7.1	286 S, 287 S	Fully implemented
Parasoft C/C++test	2023.1	CERT_CPP-DCL60-a	A class, union or enum name (including qualification, if any) shall be a unique identifier
Axivion Bauhaus Suite	7.2.0	CertC++-DCL60	-

Coding Standard 2

Coding Standard	Label	Do Not Read Uninitialized Memory
Data Value	[STD-002-CPP]	Local, automatic variables assume unexpected values if they are read before they are initialized. As a result, objects of type T with automatic or dynamic storage duration must be explicitly initialized before having their value read as part of an expression unless T is a class type or an array thereof or is an unsigned narrow character type. If T is an unsigned narrow character type, it may be used to initialize an object of unsigned narrow character type, which results in both objects having an indeterminate value. This technique can be used to implement copy operations such as std::memcpy() without triggering undefined behavior.

Noncompliant Code

An uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.

```
#include <iostream>

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

Compliant Code

The object is initialized prior to printing its value.

```
#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):
1—Validate Input Data
4—Keep It Simple
10—Adopt a Secure Coding Standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	P12	L1

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Uninitialized-read	Partially checked
Helix QAC	2024.4	DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978	-
LDRA tool suite	9.7.1	53 D, 69 D, 631 S, 652 S	Partially implemented
RuleChecker	22.10	Uninitialized-read	Partially checked

Coding Standard 3

Coding Standard	Label	Do Not Attempt to Create a std::string From a Null Pointer
String Correctness	[STD-003-CPP]	The std::basic_string type uses the traits design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Specifically, the std::basic_string class is paired with std::char_traits to create the std::string, std::wstring, std::u16string, and std::u32string classes. The std::char_traits class is explicitly specialized to provide policy-based implementation details to the std::basic_string type. One such implementation detail is the std::char_traits::length() function, which is frequently used to determine the number of characters in a null-terminated string. According to the C++ Standard, passing a null pointer to this function is undefined behavior because it would result in dereferencing a null pointer.

Noncompliant Code

A std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist.

```
#include <cstdlib>
#include <string>

void f() {
  std::string tmp(std::getenv("TMP"));
  if (!tmp.empty()) {
    // ...
  }
}
```

Compliant Code

The results from the call to std::getenv() are checked for null before the std::string object is constructed.

```
#include <cstdlib>
#include <string>

void f() {
  const char *tmpPtrVal = std::getenv("TMP");
  std::string tmp(tmpPtrVal ? tmpPtrVal : "");
  if (!tmp.empty()) {
    // ...
  }
}
```

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

Principles(s): 2—Heed Compiler Warnings

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Assert_failure	-
CodeSonar	8.1p0	LAND.MEM.NPD	Null Pointer Dereference
Parasoft C/C++ test	2023.1	CERT_CPP-STR51-a	Avoid null pointer dereferencing
Polyspace Bug Finder	R2024a	CERT C++: STR51-CPP	Checks for string operations on null pointer

Coding Standard 4

Coding Standard	Label	Prevent SQL Injection
SQL Injection	[STD-004-CPP]	SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

Noncompliant Code

The prepared statement permits a SQL injection attack by incorporating the unsanitized input argument username into the prepared statement.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class Login {
  public Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new
            com.microsoft.sqlserver.jdbc.SQLServerDriver());
    String dbConnection =
      PropertyManager.getProperty("db.connection");
    // Can hold some value like
    // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"
    return DriverManager.getConnection(dbConnection);
  }

  String hashPassword(char[] password) {
    // Create hash of password
  }

  public void doPrivilegedAction(
    String username, char[] password
  ) throws SQLException {
    Connection connection = getConnection();
    if (connection == null) {
      // Handle error
    }
    try {
      String pwd = hashPassword(password);
      String sqlString = "select * from db_user where username=" +
        username + " and password =" + pwd;
```

Noncompliant Code

```
      PreparedStatement stmt = connection.prepareStatement(sqlString);

      ResultSet rs = stmt.executeQuery();
      if (!rs.next()) {
        throw new SecurityException("User name or password incorrect");
      }

      // Authenticated; proceed
    } finally {
      try {
        connection.close();
      } catch (SQLException x) {
        // Forward to handler
      }
    }
  }
}
```

Compliant Code

Use of a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long username.

```
public void doPrivilegedAction(
  String username, char[] password
) throws SQLException {
  Connection connection = getConnection();
  if (connection == null) {
    // Handle error
  }
  try {
    String pwd = hashPassword(password);

    // Validate username length
    if (username.length() > 8) {
      // Handle error
    }

    String sqlString =
      "select * from db_user where username=? and password=?";
    PreparedStatement stmt = connection.prepareStatement(sqlString);
    stmt.setString(1, username);
    stmt.setString(2, pwd);
    ResultSet rs = stmt.executeQuery();
```

Compliant Code

```
    if (!rs.next()) {
      throw new SecurityException("User name or password incorrect");
    }

    // Authenticated; proceed
  } finally {
    try {
      connection.close();
    } catch (SQLException x) {
      // Forward to handler
    }
  }
}
```

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

Principles(s):
1—Validate Input Data
7—Sanitize Data Sent to Other Systems
10—Adopt a Secure Coding Standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
The Checker Framework	2.1.3	Tainting Checker	Trust and security errors
Coverity	7.5	SQLI FB.SQL_PREPARED_STATEMENT_GENERATED_ FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	Implemented
Findbugs	1.0	SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	Implemented
Klockwork	2024.4	SV.DATA.DB SV.SQL SV.SQL.DBSOURCE	Implemented

Coding Standard 5

Coding Standard	Label	Properly Deallocate Dynamically Allocated Resources
Memory Protection	[STD-005-CPP]	The C programming language provides several ways to allocate memory, such as std::malloc(), std::calloc(), and std::realloc(), which can be used by a C++ program. However, the C programming language defines only a single way to free the allocated memory: std::free().

Noncompliant Code

The local variable space is passed as the expression to the placement new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined behavior due to ::operator delete() attempting to free memory that was not returned by ::operator new().

```
#include <iostream>

struct S {
  S() { std::cout << "S::S()" << std::endl; }
  ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
  alignas(struct S) char space[sizeof(struct S)];
  S *s1 = new (&space) S;

  // ...

  delete s1;
}
```

Compliant Code

Removal of the call to ::operator delete(), instead explicitly calling s1's destructor. This is one of the few times when explicitly invoking a destructor is warranted.

```
#include <iostream>

struct S {
  S() { std::cout << "S::S()" << std::endl; }
  ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
```

Compliant Code

```
  alignas(struct S) char space[sizeof(struct S)];
  S *s1 = new (&space) S;

  // ...

  s1->~S();
}
```

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

Principles(s):
2—Heed Compiler Warnings
5—Default Deny
6—Adhere to the Principle of Least Privilege
9—Use Effective Quality Assurance Techniques

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
LDRA tool suite	9.7.1	232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D	Partially implemented
Polyspace Bug Finder	R2024a	CERT C++: MEM51-CPP	Checks for invalid deletion of pointer, invalid free of pointer, deallocation of previously deallocated pointer Rule partially covered.
Parasoft Insure++	-	-	Runtime detection
Clang	3.9	clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator	Checked by clang-tidy, but does not catch all violations of this rule

Coding Standard 6

Coding Standard	Label	Use a Static Assertion to Test the Value of a Constant Expression
Assertions	[STD-006-CPP]	Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems.

Noncompliant Code

Uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. The runtime assertion needs to be placed in a function and executed.

```
#include <assert.h>

struct timer {
  unsigned char MODE;
  unsigned int DATA;
  unsigned int COUNT;
};

int func(void) {
  assert(sizeof(struct timer) == sizeof(unsigned char)
+ sizeof(unsigned int) + sizeof(unsigned int));
}
```

Compliant Code

A preprocessor conditional statement may be used, as in this compliant solution. Using #error directives allows for clear diagnostic messages.

```
struct timer {
  unsigned char MODE;
  unsigned int DATA;
  unsigned int COUNT;
};

#if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned
int) + sizeof(unsigned int)))
  #error "Structure must not have any padding"
#endif
```

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

Principles(s):
2—Heed Compiler Warnings
10—Adopt a Secure Coding Standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	L3

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC-DCL03	-
Clang	3.9	Misc-static-assert	Checked by clang-tidy
CodeSonar	8.1p0	(customization)	Users can implement a custom check that reposts users of the assert() macro
ECLAIR	1.2	CC2.DCL03	Fully implemented

Coding Standard 7

Coding Standard	Label	Do Not Leak Resources When Handling Exceptions
Exceptions	[STD-007-CPP]	Reclaiming resources when exceptions are thrown is important. An exception being thrown may result in cleanup code being bypassed or an object being left in a partially initialized state. Such a partially initialized object would violate basic exception safety. It is preferable that resources be reclaimed automatically, using the RAII design pattern [Stroustrup 2001], when objects go out of scope. This technique avoids the need to write complex cleanup code when allocating resources.

Noncompliant Code

pst is not properly released when process_item throws an exception, causing a resource leak.

```
#include <new>

struct SomeType {
  SomeType() noexcept; // Performs nontrivial initialization.
  ~SomeType(); // Performs nontrivial finalization.
  void process_item() noexcept(false);
};

void f() {
  SomeType *pst = new (std::nothrow) SomeType();
  if (!pst) {
    // Handle error
    return;
  }

  try {
    pst->process_item();
  } catch (...) {
    // Process error, but do not recover from it; rethrow.
    throw;
  }
  delete pst;
}
```

Compliant Code

The exception handler frees pst by calling delete.

Compliant Code

```
#include <new>

struct SomeType {
  SomeType() noexcept; // Performs nontrivial initialization.
  ~SomeType(); // Performs nontrivial finalization.

  void process_item() noexcept(false);
};

void f() {
  SomeType *pst = new (std::nothrow) SomeType();
  if (!pst) {
    // Handle error
    return;
  }
  try {
    pst->process_item();
  } catch (...) {
    // Process error, but do not recover from it; rethrow.
    delete pst;
    throw;
  }
  delete pst;
}
```

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

Principles(s):
9—Use Effective Quality Assurance Techniques
10—Adopt a Secure Coding Standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	High	P2	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	ALLOC.LEAK	Leak
Helix QAC	2024.4	DF4756, DF4757, DF4758	-

Tool	Version	Checker	Description Tool
LDRA tool suite	9.7.1	50 D	Partially implemented
Klockwork	2024.4	CL.MLK MLK.MIGHT MLK.MUST MLK.RET.MIGHT MLK.RET.MUST RH.LEAK	-

Coding Standard 8

Coding Standard	Label	Write Constructor Member Initializers in the Canonical Order
[Student Choice]	[STD -008-CPP]	The member initializer list for a class constructor allows members to be initialized to specified values and for base class constructors to be called with specific arguments. However, the order in which initialization occurs is fixed and does not depend on the order written in the member initializer list. Consequently, the order in which member initializers appear in the member initializer list is irrelevant. The order in which members are initialized, including base class initialization, is determined by the declaration order of the class member variables or the base class specifier list. Writing member initializers other than in canonical order can result in undefined behavior, such as reading uninitialized memory.

Noncompliant Code

the member initializer list for C::C() attempts to initialize someVal first and then to initialize dependsOnSomeVal to a value dependent on someVal. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of someVal results in an unspecified value being stored into dependsOnSomeVal.

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

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

Compliant Code

Changing the declaration order of the class member variables so that the dependency can be ordered properly in the constructor's member initializer list.

```
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): 4—Keep it simple

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Initializer-list-order	Fuly checked
CodeSonar	8.1p0	LANG.STRUCT.INIT.OOMI	Out of Order Member Initializers
Clang	3.9	-Wreorder	-
LDRA tool suite	9.7.1	206 S	Fully implemented

Coding Standard 9

Coding Standard	Label	Use Valid Iterator Ranges
[Student Choice]	[STD-009-CPP]	When iterating over elements of a container, the iterators used must iterate over a valid range. An iterator range is a pair of iterators that refer to the first and past-the-end elements of the range respectively. An empty iterator range (where the two iterators are valid and equivalent) is considered to be valid. Using a range of two iterators that are invalidated or do not refer into the same container results in undefined behavior.

Noncompliant Code

The two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, std::for_each() compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results in undefined behavior.

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

void f(const std::vector<int> &c) {
  std::for_each(c.end(), c.begin(), [](int i) { std::cout << i; });
}
```

Compliant Code

The iterator values passed to std::for_each() are passed in the proper order.

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

void f(const std::vector<int> &c) {
  std::for_each(c.begin(), c.end(), [](int i) { std::cout << i; });
}
```

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

Principles(s):
3—Architect and Design for Security Policies
4—Keep It Simple
10—Adopt a Secure Coding Standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Overflow_upon_dereference	-
CodeSonar	8.1p0	LANG.MEM.BO	Buffer Overrun
Helix QAC	2024.4	C++3802	-
Polyspace Bug Finder	R2024a	CERT C++: CTR53-CPP	Checks for invalid iterator range Partially covered

Coding Standard 10

Coding Standard	Label	Do Not Access an Object Outside of its Lifetime
[Student Choice]	[STD-010-CPP]	Every object has a lifetime in which it can be used in a well-defined manner. The lifetime of an object begins when sufficient, properly aligned storage has been obtained for it and its initialization is complete. The lifetime of an object ends when a nontrivial destructor, if any, is called for the object and the storage for the object has been reused or released. Use of an object, or a pointer to an object, outside of its lifetime frequently results in undefined behavior.

Noncompliant Code

A pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in undefined behavior.

```
struct S {
  void mem_fn();
};

void f() {
  S *s;
  s->mem_fn();
}
```

Compliant Code

Storage is obtained for the pointer prior to calling S::mem_fn().

```
struct S {
  void mem_fn();
};

void f() {
  S *s = new S;
  s->mem_fn();
  delete s;
}
```

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

Principles(s):
2—Heed Compiler Warnings
10—Adopt a Secure Coding Standard

Threat Level

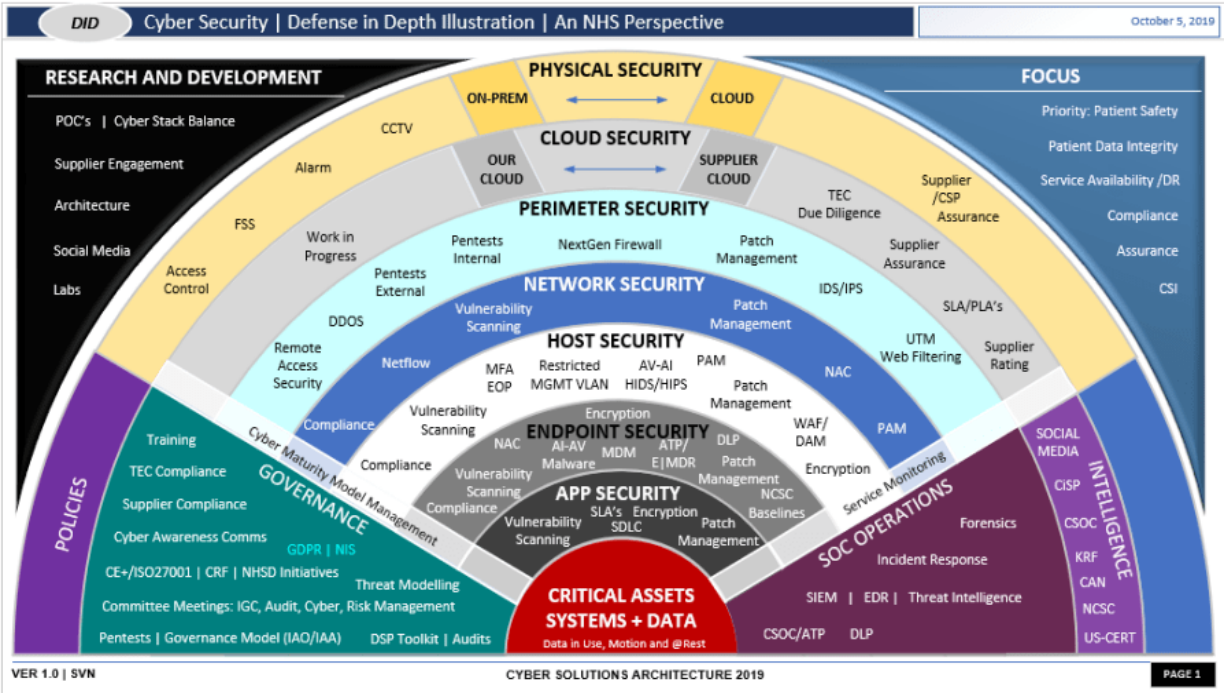
Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Return-reference0local Dangling_pointer_use	Partially checked
Clang	3.9	-Wdangling-initializer-list	Catches some lifetime issues related to incorrect use of std::initializer_list<>
CodeSonar	8.1p0	IO.UAC ALLOC.UAF	Use after close Use after free
LDRA tool suite	9.7.1	42 D, 53 D, 77 D, 1 J, 71 S, 565 S	Partially implemented

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

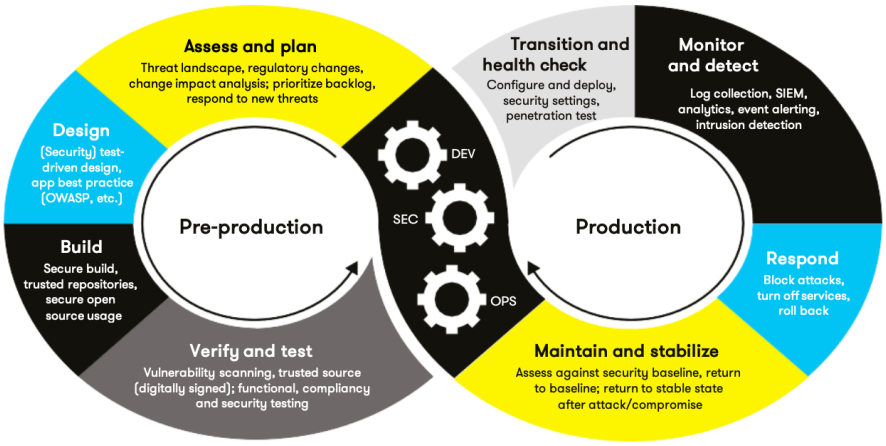
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

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	High	Unlikely	High	3	3
STD-002-CPP	High	Probable	Medium	12	1
STD-003-CPP	High	Likely	Medium	18	1
STD-004-CPP	High	Likely	Medium	18	1
STD-005-CPP	High	Likely	Medium	18	1
STD-006-CPP	Low	Unlikely	Medium	1	3
STD-007-CPP	Low	Probable	High	2	3
STD-008-CPP	Medium	Unlikely	Medium	4	3
STD-009-CPP	High	Probable	High	6	2
STD-010-CPP	High	Probable	High	6	2

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	Encryption at rest is referring to data being in an encrypted state while it is in storage. The data will be protected regardless of if a user is granted access to the data because they don't have the key to decrypt it. This is essential because this keeps the data safe even if there was a breach of security. If unauthorized access was granted the sensitive information would remain encrypted.
Encryption in flight	This policy is about keeping data encrypted during transit. While the data is through the network, for example, an email, it will stay encrypted. This is vital now that we are using the cloud more than ever.
Encryption in use	This applies to encrypting data while it is being processed by a system. Even as the data is created, updated, or read it is staying protected. Encrypting in-use data addresses this vulnerability by allowing computations to run directly on encrypted files without the need for decryption.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Authentication confirmed a user's identity, making sure they are who they're claiming to be. This includes users creating accounts, logging in, and changing their passwords. Authentication is usually done by sending out one-time passwords, verifying a username and password combination, or answering security questions that were established at the account creation time.
Authorization	As authentication will verify a user's identity, authorization will confirm the access level of a user. The level of access the user will have is dependent on their authorization given to them by the system. This will limit vulnerabilities and interactions users can have with sensitive data by keeping them locked out of certain access points.
Accounting	Accounting refers to the process of keeping track of all activity made by a user. Interactions include logging in, making changes to the database, and accessing files. This policy if vital for keeping a trail of what is always happening in the system. If something ever goes wrong, accounting allows for a backlog of all activity to take the guesswork out of what was happening in the system. Tracking down the issue is made much easier.

*****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		Completed Ten Core Security Principles and Defense in Depth	Iesha Sahin	Iesha Sahin
1.2	12/15/2024	Completed Coding Standards, Automation Detection, Encryption, and Triple A Policies	Iesha Sahin	Iesha Sahin

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV