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	Validating input data from untrusted sources such as users can help prevent vulnerabilities in the system such as network interfaces, command line arguments and environmental variables.
2. Heed Compiler Warnings	Make sure to use the highest compiler warning level available and make modifications to your code to get rid of the warnings. This will help to detect and prevent any additional security risks. Ignoring these warnings can lead to possible back doors to the system where a malicious threat could get in.
3. Architect and Design for Security Policies	When it comes to creating the architecture and design for your application make sure that you also lay out the proper security policies and make sure they are enforced to help ensure the security for your application.
4. Keep It Simple	Keep your code simple if you can, avoid making things overly complex when they don't need to be and make sure to give good comments detailing exactly what the functions in your code are responsible for doing. Failure to do so can leave the system more susceptible to errors.
5. Default Deny	The default of your system should be to deny access to everyone unless they have system permissions to access/change or create new data. This prevents you from accidentally granting access to anything that should not be available.
6. Adhere to the Principle of Least Privilege	Only information that is needed/relevant to the task should be given whether that be to users, staff, potentially moderators as well. The less privileges someone has the less damage could be done if they were to get hacked or are malicious in their own intent.
7. Sanitize Data Sent to Other Systems	Make sure you are removing any unused data in the system, only keep and use the data that is present/needed. This will prevent hackers from getting their hands on any stored information should they get in and will also help prevent injection attacks.

Principles	Write a short paragraph explaining each of the 10 principles of security.
8. Practice Defense in Depth	Defense in Depth (DiD) is the practice of using multiple layers of security that work together so that if one layer is compromised or fails another layer can prevent that flaw from being exploited.
9. Use Effective Quality Assurance Techniques	Have multiple testing phases, internal and external security reviews as well as penetration testing and do it often. Make sure the tests being ran on the system are effective which is why you want a wide variety of testing done by multiple groups. Doing it often will ensure that things are being done properly throughout the project and save time in the long run.
10. Adopt a Secure Coding Standard	Every programming language has secure coding standards, make sure that you are adopting those standards into your project. They will be different for different languages/platforms but they exist for a reason, ignoring them leads to troubles later down the road.

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 checks to see if value is within the range of enumeration values that are allowed, however this is taking place after casting to the enumeration which can lead to it not representing the given value.

```
enum EnumType {
 First,
 Second,
 Third
};
void f(int intVar) {
 EnumType enumVar = static_cast<EnumType>(intVar);

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

Compliant Code

This compliant code checks the value to make sure it can represented by the enumeration type before doing the conversion which ensures it does not result in an unspecified value.

```
enum EnumType {
 First,
 Second,
 Third
};
void f(int intVar) {
 if (intVar < First || intVar > Third) {
  // Handle error
 }
 EnumType enumVar = static_cast(intVar);
}
```

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

Principles(s): Casting out-of-range enumerations can result in a buffer overflow, resulting in arbitrary code execution, but the more likely outcome is data integrity violation.

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
Axivion Bauhaus Suite	7.2.0	**CertC++-INT50**	
CodeSonar	7.4p0	**LANG.CAST.COERCE** **LANG.CAST.VALUE**	Coercion Alters Value Cast Alters Value
Helix QAC	2023.3	**C++3013**	
Parasoft C/C++test	2023.1	**CERT_CPP-INT50-a**	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration
PVS-Studio	7.26	**V1016**	
RuleChecker	22.10	**cast-integer-to-enum**	Partially checked

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	STD-002-CCP	Use valid references, pointers, and iterators to reference elements of a container

Noncompliant Code

This noncompliant code has pos invalidated after the first call to insert() and the following loop iteration behavior is undefined.

```
#include <deque>

void f(const double *items, std::size_t count) {
 std::deque<double> d;
 auto pos = d.begin();
 for (std::size_t i = 0; i < count; ++i, ++pos) {
   d.insert(pos, items[i] + 41.0);
 }
}
```

Compliant Code

This compliant code has pos assigned to a valid iterator on each insertion which prevents the undefined behavior

```
#include <deque>

void f(const double *items, std::size_t count) {
 std::deque<double> d;
 auto pos = d.begin();
 for (std::size_t i = 0; i < count; ++i, ++pos) {
   pos = d.insert(pos, items[i] + 41.0);
 }
}
```

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

Principles(s): Using invalid references, pointers, or iterators to reference elements of a container results in undefined behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.4p0	**ALLOC.UAF**	Use After Free
Helix QAC	2023.3	**DF4746, DF4747, DF4748, DF4749**	
Parasoft C/C++test	2023.1	**CERT_CPP-STR52-a**	Use valid references, pointers, and iterators to reference elements of a basic_string
Polyspace Bug Finder	R2023a	CERT C++: STR52-CPP	Checks for use of invalid string iterator (rule partially covered).

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	Do not attempt to create a std::string from a null pointer

Noncompliant Code

This noncompliant code has a std::string object that is created by the results of std:getenv(). In this case because std:getenv() returns a null pointer on failure it can lead to undefined behavior when a variable does not exist.

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

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

Compliant Code

This compliant code has the results from std::getenv() 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): Dereferencing a null pointer results in undefined behavior, which typically causes the program to terminate. However, there are cases in which instead arbitrary code is executed which raises the severity heavily.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**assert_failure**	
CodeSonar	7.4p0	**LANG.MEM.NPD**	Null Pointer Dereference
Helix QAC	2023.3	**DF4770, DF4771, DF4772, DF4773, DF4774**	
Klocwork	2023.3	**NPD.CHECK.CALL.MIGHT** **NPD.CHECK.CALL.MUST** **NPD.CHECK.MIGHT** **NPD.CHECK.MUST** **NPD.CONST.CALL** **NPD.CONST.DEREF** **NPD.FUNC.CALL.MIGHT** **NPD.FUNC.CALL.MUST** **NPD.FUNC.MIGHT** **NPD.FUNC.MUST** **NPD.GEN.CALL.MIGHT** **NPD.GEN.CALL.MUST** **NPD.GEN.MIGHT** **NPD.GEN.MUST** **RNPD.CALL** **RNPD.DEREF**	
Parasoft C/C++test	2023.1	**CERT_CPP-STR51-a**	Avoid null pointer dereferencing
Polyspace Bug Finder	R2023a	CERT C++: STR51-CPP	Checks for string operations on null pointer (rule partially covered).

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP	Do not store already-owned pointer values in an unrelated smart pointer

Noncompliant Code

This noncompliant code has two unrelated smart pointers that are created from the same underlying pointer value, when the local automatic variable p2 is destroyed it also deletes the pointer value it manages. Then the same thing occurs when p1 is destroyed, resulting in a double-free vulnerability.

```
#include <memory>
void f() {
 int *i = new int;
 std::shared_ptr<int> p1(i);
 std::shared_ptr<int> p2(i);
}
```

Compliant Code

This compliant code the std::shared_ptr objects are related through copy construction so that when the local automatic variable p2 is destroyed the use count for the shared pointer is decremented instead of deleted. Now when p1 is also destroyed the shared pointer is decremented to zero and the manager pointer is destroyed.

```
#include <memory>
void f() {
 std::shared_ptr<int> p1 = std::make_shared<int>();
 std::shared_ptr<int> p2(p1);
}
```

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

Principles(s): Do not create an unrelated smart pointer object with a pointer value that is owned by another smart pointer object. This includes resetting a smart pointer's managed pointer to an already-owned pointer value, as this can lead to undefined behavior resulting in vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**dangling_pointer_use**	
Axivion Bauhaus Suite	7.2.0	**CertC++-MEM56**	
Helix QAC	2023.3	**DF4721, DF4722, DF4723**	
Parasoft C/C++test	2023.1	**CERT_CPP-MEM56-a**	Do not store an already-owned pointer value in an unrelated smart pointer
Polyspace Bug Finder	R2023a	CERT C++: MEM56-CPP	Checks for use of already-owned pointers (rule fully covered)
PVS-Studio	7.26	**V1006**	

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CPP	Properly deallocate dynamically allocated resources

Noncompliant Code

This noncompliant code has the local variable space passed as the expression to the placement new operator, the pointer is then passed to ::operator delete(), which causes undefined behavior because the delete operator is trying 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

This compliant code removes the call to ::operator delete() and instead explicitly calls s1's destructor (this case is one of the few times were 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() {
 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): Passing a pointer value to a deallocation function that was not previously obtained by the matching function results in undefined behavior that can lead to vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**invalid_dynamic_memory_allocation dangling_pointer_use**	
Axivion Bauhaus Suite	7.2.0	**CertC++-MEM51**	
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
CodeSonar	7.4p0	**ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK**	Free non-heap variable Double free Type mismatch Leak
Helix QAC	2023.3	**C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264**	
Klocwork	2023.3	**CL.FFM.ASSIGN CL.FFM.COPY CL.FMM CL.SHALLOW.ASSIGN CL.SHALLOW.COPY FMM.MIGHT FMM.MUST FNH.MIGHT FNH.MUST FUM.GEN.MIGHT FUM.GEN.MUST UNINIT.CTOR.MIGHT UNINIT.CTOR.MUST UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST**	
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
Parasoft C/C++test	2023.1	**CERT_CPP-MEM51-a CERT_CPP-MEM51-b CERT_CPP-MEM51-c CERT_CPP-MEM51-d**	Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy

Tool	Version	Checker	Description Tool
			assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2023a	CERT C++: MEM51-CPP	Checks for: • Invalid deletion of pointer • Invalid free of pointer • Deallocation of previously deallocated pointer Rule partially covered.
PVS-Studio	7.26	**V515**, **V554**, **V611**, **V701**, **V748**, **V773**, **V1066**	
SonarQube C/C++ Plugin	4.10	**S1232**	
Astrée	22.10	**invalid_dynamic_memory_allocation dangling_pointer_use**	

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	STD-006-CPP	Use a static assertion to test the value of a constant expression

Noncompliant Code

This noncompliant code uses an assert() class in order to assert a property concerning a memory-mapped structure that is essential for the code to behave properly.

```
#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

This compliant code since it regards assertions involving only constant expressions, a preprocessor conditional statement may be used like the following

```
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): Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The runtime 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.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	P3

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	7.4p0	**(customization)**	Users can implement a custom check that reports uses of the assert() macro
Compass/ROSE			Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation
ECLAIR	1.2	**CC2.DCL03**	Fully implemented
LDRA tool suite	9.7.1	**44 S**	Fully implemented

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Handle all exceptions thrown before main() begins executing

Noncompliant Code

This noncompliant code has a constructor for S that may throw an exception but it is not caught when globalS is constructed during start up.

```
struct S {
  S() noexcept(false);
};
  static S globalS;
```

Compliant Code

This compliant code turns the globalS into a local variable with static storage which allows any exception thrown while it is being constructed. This is because the constructor for S will be executed during the first call for globalS rather than during the start up.

```
struct S {
  S() noexcept(false);
};

S &globalS() {
 try {
   static S s;
   return s;
 }
 catch (...) {
  // Handle error, perhaps by logging it and gracefully terminating the
application.
 }
  // Unreachable.
}
```

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

Principles(s): All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Low	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**main-function-catch-all** **early-catch-all**	Partially checked
Axivion Bauhaus Suite	7.2.0	**CertC++-ERR51**	
CodeSonar	7.4p0	**LANG.STRUCT.UCTCH**	Unreachable Catch
Helix QAC	2023.3	**C++4035, C++4036, C++4037**	
Klocwork	2023.3	**MISRA.CATCH.ALL**	
LDRA tool suite	9.7.1	**527 S**	Partially implemented
Parasoft C/C++test	2023.1	**CERT_CPP-ERR51-a** **CERT_CPP-ERR51-b**	Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point
Polyspace Bug Finder	R2023a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)
RuleChecker	22.10	**main-function-catch-all** **early-catch-all**	Partially checked

Coding Standard 8

Coding Standard	Label	Name of Standard
Input Output	STD-008-CPP	Do not alternate input and output from a file stream without intervening positioning calls

Noncompliant Code

This noncompliant code appends data to the end of the file and then reads it. Since there is no intervening positioning call between the output and input calls the resulting behavior is undefined.

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

 file << "Output some data";
 std::string str;
 file >> str;
}
```

Compliant Code

[Compliant description]

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

 file << "Output some data";
 std::string str;
 file.seekg(0, std::ios::beg);
 file >> str;
}
```

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

Principles(s): Alternately input and output from a stream without an intervening flush or positioning call can cause undefined behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Medium	P6	L2

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	**CertC++-FIO50**	
CodeSonar	7.4p0	**IO.IOWOP** **IO.OIWOP**	Input After Output Without Positioning Output After Input Without Positioning
Helix QAC	2023.3	**DF4711, DF4712, DF4713**	
Parasoft C/C++test	2023.1	**CERT_CPP-FIO50-a**	Do not alternately input and output from a stream without an intervening flush or positioning call
Polyspace Bug Finder	R2023a	CERT C++: FIO50-CPP	Checks for alternating input and output from a stream without flush or positioning call (rule fully covered)

Coding Standard 9

Coding Standard	Label	Name of Standard
Object Oriented Programming	STD-009-CPP	Do not invoke virtual functions from constructors or destructors.

Noncompliant Code

This noncompliant code has a base class that attempts to seize and release an object's resources using virtual functions from the constructor/destructor. What happens is the B::B() constructor calls for B::seize() and release rather than D seize and release.

```
struct B {
 B() { seize(); }
 virtual ~B() { release(); }

protected:
 virtual void seize();
 virtual void release();
};
struct D : B {
 virtual ~D() = default;

protected:
 void seize() override {
 B::seize();
 // Get derived resources...
 }

 void release() override {
 // Release derived resources...
 B::release();
 }
};
```

Compliant Code

This compliant code uses a private member function instead of a virtual function for the constructor and destructor calls, this allows each class to seize and release its own resources.

```
Class B {
 void seize_mine();
 void release_mine();

public:
 B() { seize_mine(); }
 virtual ~B() { release_mine(); }
protected:
```

Compliant Code

```
 virtual void seize() { seize_mine(); }
 virtual void release() { release_mine(); }
};
class D : public B {
 void seize_mine();
 void release_mine();

public:
 D() { seize_mine(); }
 virtual ~D() { release_mine(); }
protected:
 void seize() override {
 B::seize();
 seize_mine();
 }

 void release() override {
 release_mine();
 B::release();
 }
};
```

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

Principles(s): Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**virtual-call-in-constructor invalid_function_pointer**	Fully checked
Axivion Bauhaus Suite	7.2.0	**CertC++-OOP50**	
Clang	3.9	clang-analyzer-alpha.cplusplus.VirtualCall	Checked by clang-tidy
CodeSonar	7.4p0	**LANG.STRUCT.VCALL_IN_CTOR LANG.STRUCT.VCALL_IN_DTOR**	Virtual Call in Constructor Virtual Call in Destructor
Helix QAC	2023.3	**C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282**	

Tool	Version	Checker	Description Tool
Klocwork	2023.3	**CERT.OOP.CTOR.VIRTUAL_FUNC**	
LDRA tool suite	9.7.1	**467 S, 92 D**	Fully implemented
Parasoft C/C++test	2023.1	**CERT_CPP-OOP50-a** **CERT_CPP-OOP50-b** **CERT_CPP-OOP50-c** **CERT_CPP-OOP50-d**	Avoid calling virtual functions from constructors Avoid calling virtual functions from destructors Do not use dynamic type of an object under construction Do not use dynamic type of an object under destruction
Polyspace Bug Finder	R2023a	CERT C++: OOP50-CPP	Checks for virtual function call from constructors and destructors (rule fully covered)
PVS-Studio	7.26	**V1053**	
RuleChecker	22.10	**virtual-call-in-constructor**	Fully checked
SonarQube C/C++ Plugin	4.10	**S1699**	

Coding Standard 10

Coding Standard	Label	Name of Standard
Miscellaneous	STD-010-CPP	Value returning functions must return a value from all exit paths

Noncompliant Code

This noncompliant code does not contain a return for the input value for positive input so not all paths return a value.

```
int absolute_value(int a) {
 if (a < 0) {
 return -a;
 }
}
```

Compliant Code

This compliant code contains the return for positive input so all paths now return a value.

```
int absolute_value(int a) {
 if (a < 0) {
 return -a;
 }
 return a;
}
```

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

Principles(s): A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, such as from a *function-try-block.*

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

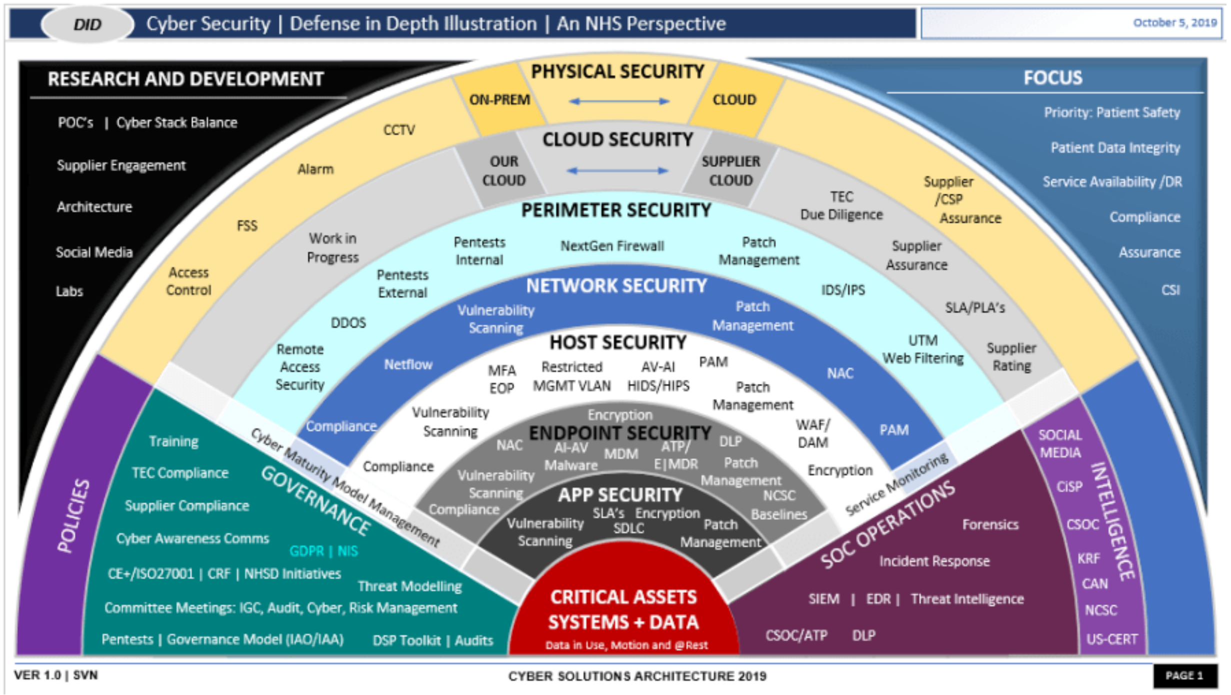
Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**return-implicit**	Fully checked
Axivion Bauhaus Suite	7.2.0	**CertC++-MSC52**	

Tool	Version	Checker	Description Tool
Clang	3.9	**-Wreturn-type**	Does not catch all instances of this rule, such as *function-try-blocks*
CodeSonar	7.4p0	**LANG.STRUCT.MRS**	Missing return statement
Helix QAC	2023.3	**DF2888**	
Klocwork	2023.3	**FUNCRET.GEN** **FUNCRET.IMPLICIT**	
LDRA tool suite	9.7.1	**2 D, 36 S**	Fully implemented
Parasoft C/C++test	2023.1	**CERT_CPP-MSC52-a**	All exit paths from a function, except main(), with non-void return type shall have an explicit return statement with an expression
Polyspace Bug Finder	R2023a	CERT C++: MSC52-CPP	Checks for missing return statements (rule partially covered)
SonarQube C/C++ Plugin	4.10	**S935**	
PVS-Studio	7.26	**V591**	
RuleChecker	22.10	**return-implicit**	Fully checked

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

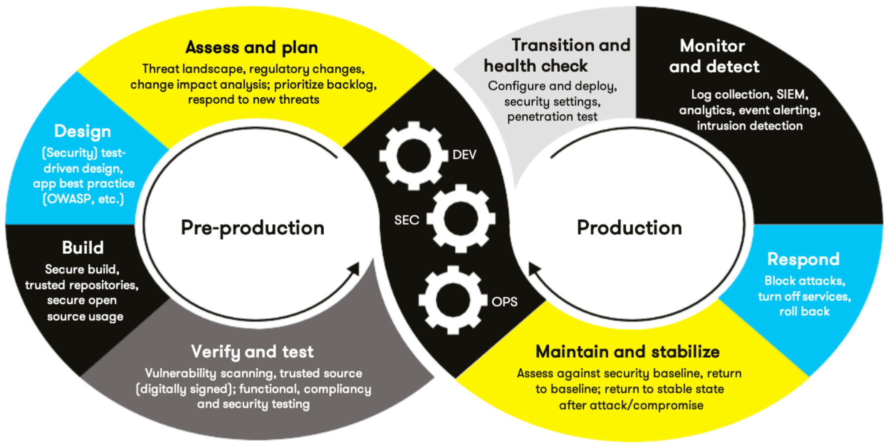
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

Implementing automation throughout the entire process is a good idea, it's important to note that ensuring testing is done from the beginning, thoroughly and often will help detect any vulnerabilities so they can be handled during the early stages of development as well.

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	Medium	P4	L3
STD-002-CPP	High	Probable	High	P6	L2
STD-003-CPP	High	Likely	Medium	P18	L1
STD-004-CPP	High	Likely	Medium	P18	L1
STD-005-CPP	High	Likely	Medium	P18	L1
STD-006-CPP	Low	Unlikely	High	P1	L3
STD-007-CPP	Low	Likely	Low	P9	L2
STD-008-CPP	Low	Likely	Medium	P6	L2
STD-009-CPP	Low	Unlikely	Medium	P2	L3
STD-010-CPP	Medium	Probable	Medium	P8	L2

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	Encryption in rest for data is a process in which data is securely encoded as it is written into storage and then decrypting that data when it is pulled from the storage. By utilizing a symmetric encryption key when writing the data to storage it protects the data from unauthorized access. This means even if someone manages to get into your system if they do not have the key, they cannot decrypt the information that is stored.
Encryption at flight	Encryption at flight for data is a process in which you securely encode data as it is being transmitted. The type of encryption used is going to depend on how you are transferring the data, as there are multiple ways in which you can transfer data securely. For instance, if you are using email, you should encrypt the data before you send it and use digital signatures.
Encryption in use	Encryption in use for data is a process in which you protect data as it is utilized in memory. Typically, the way this is done is by using password protected profiles since they protect the memory of each user and the data that they store in memory. An example of this is having multiple accounts on a computer, each account can save/store its own data and keep it separated from the other user accounts.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	This process is used to prove who the user is, typically by having the user provide details like a username and password. Though it can be taken to higher levels with thinks like secure tokens and hardware credentials as well.
Authorization	After a user has been authenticated, they are granted access to specific parts of the system, by limiting access using the principle of least privilege it helps to keep as much information secure as possible if a breach does occur. The system admin regulates which users or user roles and what information they are authorized to access such as folders, programs, and data.
Accounting	After the user has been authenticated and authorized it is still a good idea to monitor and record their activity, which we call accounting. Using accounting will let you know who is accessing or attempting to access your system as well as what they are doing to the data while they are authenticated and authorized.

*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	09/14/2023	Revision One	Austin Moore	
1.2	10/03/2023	Revision Two	Austin Moore	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV