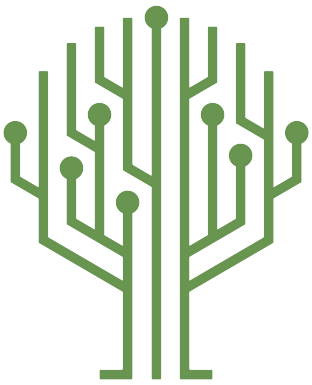


Green Pace

Green Pace Secure Development Policy

Contents


Green Pace

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	Accepting user input blindly opens the door exploitation. Input data should be subjected to rigorous testing and sanitization before being allow to activate any further code or be passed on to other system components. SQL injection and buffer overflow are two examples we've looked at in this course of ways that user input can cause problems. Testing needs to be in place to make sure that user data fits our validation schema – that is, that it is entered in the form we're looking for and is error handled in some manner if not. This may include actual characters we may allow or disallow, or whole phrases in the case of SQL injection, or the size of user inputs. All these factors need to be mitigated to ensure security.
2. Heed Compiler Warnings	Compiler warnings are generated during compilation, and point out errors that may occur due to some manner in which code has been written. For example, I recently had a compiler warning that stated that an unnecessary copy of a local variable may have been generated by the way I had written my code. While these warnings don't end compilation, the final compiled product may not work how we intended at run-time. That unnecessary variable copy may have compromised the stack frame, or caused a function to behave erroneously because the wrong copy of the variable was been used. Compiler warnings are there to point out possible issues that can bear bitter fruit at run-time, and it is a general rule to mitigate the errors or possible errors they point out.
3. Architect and Design for Security Policies	Designing a secure system goes beyond making sure that code is error-free. How do we design a system that lets those who are authentic in, lets them do only what they are authorized to do, and keeps out malicious actors? Visualizing security flow, we need to implement strategic points where these checks can be made. For example, in an ABAC system, once a user has been authenticated, every action that they attempt to do will be checked to see if they are authorized by a PDP (or policy decision point). Depending on what one's role and rights within any given system decides what one can and can't do. This allow granular

Principles	Write a short paragraph explaining each of the 10 principles of security.
	control over what users do and access. ABAC is only one access-control paradigm for authorization control. Whatever schema is used, user authentication must be verified and checked as often as necessary, and user authorization should be checked each time an action is provided. Having strategic policy points where these decision are made are vital when designing how a system needs to handle behavior.
4. Keep It Simple	Complexity, when mismanaged, invites chaos. There is nothing with writing simple code as long as it meets the functionality, resource-usage, and security needs of its requirements. Complex code with too many components can actually cause more problems that less “advanced” code. We now have a system that is complicated, with a lot going on that can inadvertently cause errors or compromise some layer of our security. No one needs to build a digraph from scratch to store user information when a simple linked list will do the same and behave exactly how it is supposed to in the larger systemview.
5. Default Deny	The default on any security system should be to deny access first unless present on a predefined whitelist. Until authentication and authorization can be confirmed, any attempt to access a system or resource should be denied. This allows us to strictly control the system and maintain a select access protocol.
6. Adhere to the Principle of Least Privilege	Allow users access only to what they need and nothing more. Access needs to be limited only to what is necessary to complete a given task. For example, a user needs to send an email using his or her company's in-house email service. During that session, they should only be authorized to interface with resources related to the email system. Giving user wider or blanket access to systems is how vulnerabilities occur via account breaches that give hackers access to that same wide array of system resources.
7. Sanitize Data Sent to Other Systems	We cannot trust data from any source system-to-system, even data that one system generates automatically without human input. The best practice is always to ensure that every packet of data sent is sanitized to ensure that it will not cause any errors. In terms of human input, we're looking at SQL injection or invalid input that can cause system errors. In terms of system-to-system generated data, formatting issues can cause errors. These need to be mitigated through sanitization.
8. Practice Defense in Depth	In security, each layer of protection we add is suited for a certain role. Where it covers somethings, it may be insufficient with others. Layering defensive measures that each serve a certain purpose helps us construct a much stronger barrier against intrusion.
9. Use Effective Quality Assurance Techniques	The old adage is that writing code is only a small part of designing software: the vast majority of time is spent testing and fixing all the breaking points you find. Testing needs to be complete in order to ensure that the system will operate as it should at scale. This involves user testing and review, deep analysis of the overall architecture of the code, integration testing with other systems and subsystems, et cetera. There are infinite ways code can break or fail to serve users' purposes, and QA should strive to minimize them.
10. Adopt a Secure	Coding securely is not something that just one member of a team is responsible

Principles	Write a short paragraph explaining each of the 10 principles of security.
Coding Standard	for. Establishing a central coding standard allows all programmers to adhere and understand exactly what is expected in terms of security when designing and implementing their code. With everyone on the same page, the risk of security flaws diminishes (or at least, the variety of errors will be reduced).

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]	**Guarantee that storage for strings has sufficient space for character data and the null terminator.** Attempting to copy data to a buffer that is not large enough to hold the data will result in a buffer overflow. C-style strings use bounded arrays with a null character signifying termination. Using this older method may result in overflow or truncated strings. In order to mitigate this, it is best to use std::string, as the memory can be allocated dynamically.

Noncompliant Code

In this example, we create a bounded char array to store input. However, our input via std::cin is unbounded, and can lead to buffer overflow. It is also possible that input could match the size of the char array; in this case, truncation is possible as the null character may replace the final character in the array to signify the end.

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

Compliant Code

This code uses std::string, which is not truncated and does not use a bounded array for input. The string class uses dynamic memory which uses an internally allocated buffer. If input data exceeds the current buffer, a new buffer is allocated with increased size and the data is copied into it. Therefore, strings have virtually unlimited size.

```
void f() {

      std::string input;

      std::string stringOne, stringTwo;

      std::cin >> stringOne >> stringTwo;

}
```

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

Principles(s):
2. Heed Compiler Warnings – Many IDEs will warn you when buffer overrun is a possibility. Paying attention to this and other analysis tools help us mitigate code errors before deployment
4. Keep it Simple – While using C-strings are still something we can do in C++ using char arrays, we have the lovely String data type to work with, which allots dynamic memory and mitigates buffer overflow. Use the simple solution that the language itself gives us!
1. Validate User Input – If for some reason we need to use the old-school C-strings method, validate user input to ensure it is the proper length before writing it the char arrays!

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.0p0	**MISC.MEM.NTERM** **LANG.MEM.BO** **LANG.MEM.TO**	No space for null terminator Buffer overrun Type overrun
Parasoft C/C++ test	2022.1	**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	R2022a	**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 Rule partially covered.
LDRA tool suite	09/07/01	**489 S, 66 X, 70 X, 71 X**	Partially implemented

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	**Ensure that unsigned integer operations do not wrap.** Computations involving unsigned operands may result in wrapping, wherein the value simply wraps back around to either the low or high side of the range following computation.

Noncompliant Code

In this example, ui_a has a value of UINT_MAX, which is the maximum value that can be stored in a unsigned integer. Adding ui_b, with its value of 1, wraps back around UINT_MIN. In theory, we can never have overflow. However, this can cause unforeseen errors in subsequent code execution and vulnerabilities that can be exploited.

```
void func(unsigned int ui_a, unsigned int ui_b) {

      // ui_a = UINT_MAX, ui_b = 1

     unsigned int usum = ui_a + ui_b;  //usum equals UINT_MIN

      /* … */
}
```

Compliant Code

This code checks to see if there will be unsigned wrap by checking if the result of subtracting ui_a from UINT_MAX is less than ui_b. If not, the computation may be made; otherwise, the system throws an error.

```
void func(unsigned int ui_a, unsigned int ui_b) {

     unsigned int usum;
     if (UINT_MAX - ui_a < ui_b) {
         // Handle Error
     } else {
         usum = ui_a + ui_b;
     }
     /* … */
}
```

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

Principles(s):

9. Use Effective Quality Assurance Techniques – We can use assertions to test our computations at various stages to ensure that these types of errors do not occur.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Coverity	2017.07	**INTEGER_OVERFLOW**	Implemented
Astree	22.04	**Integer-overflow**	Fully checked
Parasoft C/C++ Test	2022.1	**CERT_C-INT30-a** **CERT_C-INT30-b** **CERT_C-INT30-c**	Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '*' operator Integer overflow or underflow in constant expression in '<<' operator
Polyspace Bug Finder	R2022a	CERT C: Rule INT30-C	Checks for: •Unsigned integer overflow •Unsigned integer constant overflow Rule partially covered.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	**Use pointers to const when referring to string literals.** String literals are considered constant. Omitting const may lead to errors if modification is attempted.

Noncompliant Code

In this example, the string literal has been declared without the const keyword. Any attempt following this declaration to mutate the string (i.e. a[4] = '1') will result in a undefined assignment.

```
char  *a = "CS 405";
```

Compliant Code

With the const keyword added, any attempt to modify the string will throw an error. If the goal is to create a mutable string, we can simply use char a[] = "CS 405", and we can modify it without a problem.

```
const char  *a = "CS 405";
```

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

Principles(s):

2. Heed Compiler Warnings – Many IDEs will catch errors for attempts to modify string literals
9. Use Effective Quality Assurance Techniques – We can use assertions and other testing methods to ensure that string literals are not modified

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	**P3**	**L3**

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++ test	2022.1	**CERT_C-STR05-a**	A string literal shall not be modified
Astree	22.04	**Literal-assignment**	Fully checked
CodeSonar	7.0p0	**LANG.TYPE.NCS**	Non-const string literal
PC-lint Plus	1.4	**1776**	Fully supported

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	**Always use parameterized statements where available.** This is the best line of defense against SQL injection because it allows the parameterized string and the user-input parameters to be passed to the database separately.

Noncompliant Code

In this example, the string is fully constructed and then passed to the executeQuery function. The system cannot tell the difference between what is code and what is data. Therefore, user input could be maliciously crafted to extend that SQL query's functionality beyond what the developer intended.

```
void  func() {

        String email;
        cin >> email;  // email = “me@me.com OR 1=1”

        Connection conn = DriverManager.getConnection(URL, USER, PASS);
        Statement stmt = conn.createStatement();

        String sql = "SELECT * FROM users WHERE email = '" + email + "'";

        ResultSet results = stmt.executeQuery(sql);

        /* … */

    }
```

Compliant Code

In this example, the query is constructed using parameterized statements. The body of the query and the user input are kept separated. Even though the user has input text designed to circumvent the security of the query, it will be treated as data, not code. Therefore, the query will not execute maliciously.

```
    Void func() {

        String email;
        cin >> email;  // email = “me@me.com OR 1=1”

        Connection conn = DriverManager.getConnection(URL, USER, PASS);
        String sql = "SELECT * FROM users WHERE email = ?”;
        PreparedStatement stmt = conn.prepareStatement(sql);

        stmt.setString(1, email);

        ResultSet results = stmt.executeQuery(sql);
```

Compliant Code

```
        /* ... */

    }
```

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.]
3. Architect and Design for Security Policies – By securing input, we can allow users access only to what the function needs at that time
7. Sanitize Data Sent to Other Systems – Protect our databases and other systems by securing what users are accessing
10. Adopt a Secure Coding Standard – Prepared Statements are the best line of defense against SQL injection and should be in every developer's arsenal from the onset

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.0p0	**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
Parasoft C/C++ test	2022.1	**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	R2022a	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.
Astree	22.04		Supported by stubbing/taint

Tool	Version	Checker	Description Tool
			analysis

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	**Free dynamically allocated memory when no longer needed.** When working with pointers, we should free the memory allocated at the end of its lifetime to ensure memory leak and/or exhaustion is not a problem.

Noncompliant Code

In this example, we have dynamically allocated memory for the char pointer. However, as it is not used outside the scope of this function, we should free it before moving on.

```
    enum { BUFFER_SIZE = 32 };

    int f(void) {

        char *txt_buffer = (char *)malloc(BUFFER_SIZE);  //generates a
char pointer with size of 32

        if (txt_buffer == NULL) {
            return -1;
        }
        return 0;
    }
```

Compliant Code

In this example, before we exit the function, we deallocate the memory that the char pointer was taking up by using the free() function call.

```
    enum { BUFFER_SIZE = 32 };

    int f(void) {

        //generates a char pointer with size of 32
        char *txt_buffer = (char *)malloc(BUFFER_SIZE);

        if (txt_buffer == NULL) {
            return -1;
        }

        free(txt_buffer);
        return 0;
    }
```

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

Principles(s):
2. Heed Compiler Warnings – Most IDEs can warn you about non-deallocated memory spaces in these scenarios
4. Keep it Simple – Just like your mom always said: if you're not using it, put it back. Not using that memory anymore? Free it up so the system can use it.
10. Adopt a Secure Coding Standard – One developer using more memory than necessary can be dealt with; an entire team of developers not paying attention to memory management can cripple a system with too much overhead usage. Get the team on board with secure memory allocation and deallocation.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	**P8**	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.0p0	**ALLOC.LEAK**	Can detect dynamically allocated resources that are not freed
Coverity	2017.07	**RESOURCE-LEAK** **ALLOC_FREE_MISMATCH**	Finds resource leaks from variables that go out of scope while owning a resource
CppCheck	1.66	**LeakReturnValNotUsed**	Doesn't use return value of memory allocation function
Parasoft C/C++ Test	2022.1	**CERT_C-MEM31-a**	Ensure resources are freed

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	**Assertions should be used as both preconditional, intrafunction, and postconditional checks as often as possible.**

Noncompliant Code

In this example, there are things that we can use assertions for to make sure that the code will execute how we need. For example, perhaps we wanted to have a and b be certain values before continuing, or we wanted to validate the input.

```
    void func(int a, int b) {

        int userInput;
        int sum;

        cout << "Input a number between 1 and 9";
        cin  >> userInput;

        sum = userInput + a + b;
    }
```

Compliant Code

Using the Google Assertions framework, we can put preconditions and check for validity along the way.

```
void func(int a, int b) {
        //a and b should be greater than zero
        EXPECT_TRUE(a > 0 && b > 0);

        int userInput;
        int sum;

        cout << "Input a number between 1 and 9";
        cin  >> userInput;
        ASSERT_TRUE(userInput.isDigit && (userInput > 0 && userInput <
10));

        sum = userInput + a + b;
        ASSERT_TRUE(sum > 0);
}
```

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

Principles(s):
4. Keep it Simple – If it can break the code, it should be tested!
9. Use Effective Quality Assurance Techniques – Assertions are a simple yet effective way to monitor code behavior and ensure that everything is running smoothly. They take very little time but have a big impact.
10. Adopt a Secure Coding Standard – Every member of the team should be using assertions to make in-code accuracy checks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	**P1**	**L3**

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.0p0	**(customization)**	Users can implement a custom check that reports uses of the assert() macro
Clang	3.9	Misc-static-assert	Checked by clang-tidy
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]	**Utilize try-catch blocks to handle all exceptions that may occur during code execution.**

Noncompliant Code

With no error handling in place, what happens if a is zero?

```
    void func() {

        int a;
        int userInput;
        int result;

        cin >> userInput >> a;

        result = userInput / a;

        cout >> result;

     }
```

Compliant Code

With this try-catch block, we can throw an exception if the user enters 0 for the second operand, therefore removing the possibility of a divide-by-zero error.

```
void func() {

        int a;
        int userInput;
        int result;

        try {
            cin >> userInput >> a;
            if (a == 0) {
                throw e;
            }
            else {
                result = userInput / a;
                cout >> result;
            }
        }
        catch (...) {
            cout << "You have entered an input of zero for the second
operand!\n";
        }
```

Compliant Code

```
}
```

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

Principles(s):

9. Use Effective Quality Assurance Techniques – Every error that can be thrown should be caught, either by a specific exception or a generic handler.
10. Adopt a Secure Coding Standard – Again, good exceptions usage only works if everyone in a team is on board. The standard needs to be universal coverage of exceptions for all use-cases.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	**P4**	**L3**

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++ test	2022.1	**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	R2022a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)
CodeSonar	7.0p0	**LANG.STRUCT.UCTCH**	Unreachable Catch
Astree	20.10	**main-function-catch-all** **early-catch-all**	Partially checked

Coding Standard 8

Coding Standard	Label	Name of Standard
Logical Completeness	[STD-008-CPP]	**Strive for logical completeness.** When writing code, we should consider all possible data states. These include, but are not limited to, constructing complete logic for if-then and switch statements.

Noncompliant Code

In this example, we have not defined a default case. While it may seem unnecessary at times, the default case should always be included to complete the logic of the statement.

```
void func() {

    int userInput;

    while (userInput < 1 || userInput > 10) {
        cin >> userInput;
    }

    switch (userInput) {
        case 1: return 1;
        case 2: return 2;
        …
        case 10: return 10;
    }

    /* . . . */

}
```

Compliant Code

[Compliant description]

```
void func() {

    int userInput;

    while (userInput < 1 || userInput > 10) {
        cin >> userInput;
    }

    switch (userInput) {
        case 1: return 1;
        case 2: return 2;
        …
        case 10: return 10;
        default: return 0;  //now we are logically complete
    }
```

Compliant Code

```
    /* . . . */

}
```

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

Principles(s):

4. Keep it Simple – You wouldn't start with the beginning, go to the middle, then skip the end. Make sure that your logic has complete coverage.

10. Adopt a Secure Coding Standard – Make sure that logical completeness is the expectation, not the exception!

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	**P8**	**L2**

Automation

Tool	Version	Checker	Description Tool
Astree	22.04	**Missing-else** **switch-default**	Partially checked
LDRA tool suite	9.7.1	**48 S, 59 S**	Fully implemented
Polyspace Bug Finder	R2022a	CERT C: Rec. MSC01-C	Checks for missing case for switch condition (rule partially covered)
Parasoft C/C++ test	2022.1	**CERT_C-MSC01-a** **CERT_C-MSC01-b**	All 'if…else-if' constructs shall be terminated with an 'else' clause The final clause of a switch statement shall be the default clause

Coding Standard 9

Coding Standard	Label	Name of Standard
Variable Declaration	[STD-009-CPP]	**Minimize the scope of variables and functions.**

Noncompliant Code

In this example, the count variable is intialized globally and then used in func(). For our purposes, this is the only time that count is used. Therefore, we could actually declare this within func() and keep its scope minimal.

```
int count = 0;

void func(int userCount) {

      if (count++ > userCount) {
          /* . . . */
      }
}
```

Compliant Code

By using static to declare the variable within the function, the local variable will persist for as long as the program continues to run. We have removed it from a global to a local scope, which is its minimum possible scope.

```
int count = 0;

void func(int userCount) {
      static int count = 0;

      if (count++ > userCount) {
          /* . . . */
      }
}
```

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.]
2. Heed Compiler Warnings – Many IDEs will warn you about variables that are either unused, used out-of-scope, or cases where scope can be reduced.
8. Practice Defense-in-Depth – Creating tigher runtime environments limits exposure more than declaring variables as global or not securing them using mutators and accessors.
10. Adopt a Secure Coding Standard – Limiting variables to only their most-narrow scope ensure that they are only being modified or used specifically when we need them.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	**P2**	**L3**

Automation

Tool	Version	Checker	Description Tool
Polyspace Bug Finder	R2022a	CERT C: Rec. DCL19-C	Checks for: •Function or object declared without static specifier and referenced in only one file •Object defined beyond necessary scope Rec. partially covered.
CodeSonar	7.0p0	**LANG.STRUCT.SCOPE.FILE** **LANG.STRUCT.SCOPE.LOCAL**	Scope could be file static Scope could be local static
Parasoft C/C++ test	2022.1	**CERT_C-DCL19-a**	Declare variables as locally as possible
RuleChecker	22.04	**Local-object-scope** **global-object-scope**	Partially checked

Coding Standard 10

Coding Standard	Label	Name of Standard
Proper Syntax Usage	[STD-010-CPP]	**Treat relational and equality operators as if they non-associative.** These operators are interpreted starting with the leftmost pair. Trying to use them in multi-operand statements can get confusing and lead to unintended errors in code and possible vulnerabilities.

Noncompliant Code

In this example, if we're intending to see if these integers are associative (that is, if a is less than b and less than c, or a is equal to both b and c), the system will not provide what we're looking for. Because of the left-associative property of the operand, the code will be interpreted as (a < b) < c and (a == b) == c. These may have different outcomes than what we need.

```
Int x;
int y;
int z;

if (x < y < z) {
    /* . . . */
}

if (x == y == z) {
    /* . . . */
}
```

Compliant Code

This structure allows us to properly check association between a and c without causing any unintended errors in how we wanted interpretation to occur.

```
int x;
int y;
int z;

if ((x < y) && (x< z)) {
    /* . . . */
}

if ((x == y) && (x == z)) {
    /* . . . */
}
```

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

Principles(s):
2. Heed compiler warnings – Many IDEs can spot and warn you about potential mishaps with relational and equality checks
4. Keep it Simple – Always clarify what you are trying to accomplish in relational and equality checks using parentheses to organize

Threat Level

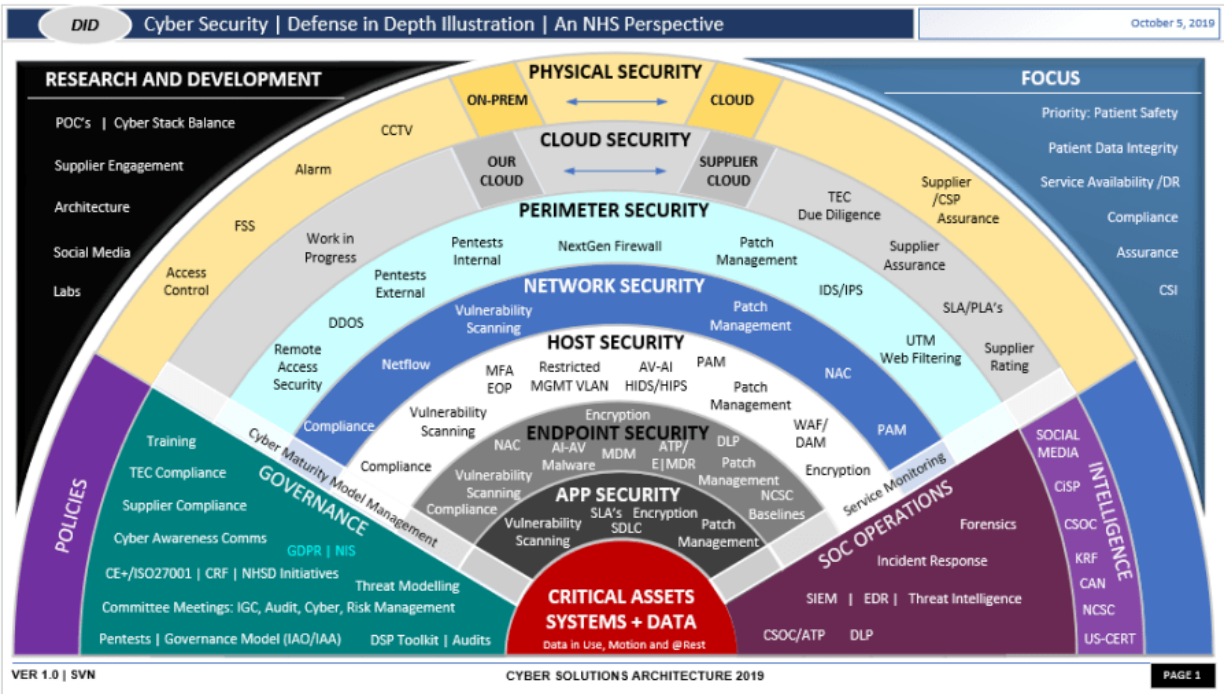
Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	**P2**	**L3**

Automation

Tool	Version	Checker	Description Tool
Polyspace Bug Finder	R2022a	CERT C: Rec. EXP13-C	Checks for possibly unintended evaluation of expression because of operator precedence rules (rec. fully covered)
ECLAIR	1.2	**CC2.EXP13**	Fully implemented
RuleChecker	22.04	**Chained-comparison**	Fully checked
GCC	4.3.5		Option `-Wparentheses` warns if a comparison like `x<=y<=z` appears; this warning is also enabled by `-Wall`

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 non-compliant code. Coding standards begin on the security policy.

Risk Assessment

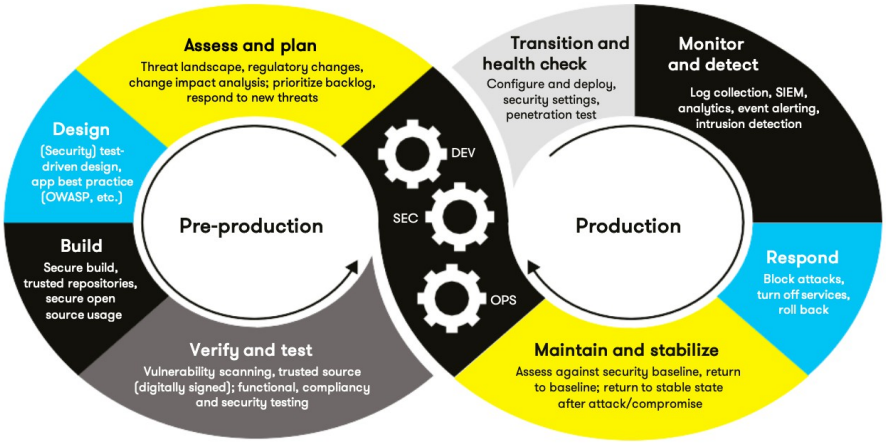
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

During pre-production, we need to focus on what our projects open us up to in terms of security. Different languages, dependencies, and deployments have trade-offs in terms of features and vulnerabilities. Finding a good combination of technologies that serve our purpose while aiding minimization of risk is the goal. Threat modeling can help us understand current intrusion methods and how they can affect the product we're building.

Secure coding principles should be a mainstay of developer education at the company. We need to strive to write strong code from the get-go. Of course, when we begin to design code, testing should be hand-in-hand. The CI/CD pipeline should push code into repos or containers which can be checked thoroughly. Using automated tools to do unit tests and smaller-scale integration testing as we build features should occur daily. Secure coding principles should be a mainstay of developer education at the company. We need to strive to write strong code from the get-go as this is our first line of defense. Tools like static analysis and code reviews will help on a developer level to strengthen code.

After a given product is deployed, we should continue to monitor for emerging vulnerabilities in both our code and the security landscaper as a whole. Just because our system is secure when it launches doesn't ensure that in perpetuity. Malicious actors are always innovating. Pentesting and practicing security scenarios are key to evolving our system to be durable. There needs to be hard-and-fast policy in place to automatically handle security attacks: stopping services, implementation of emergency security measures like complete levels of access suspension, et cetera. Once an attack is dealt with, the security team can stabilize the system and then dig into what happened and what is at fault. New findings can act as the catalyst for improved measures. There is not such thing as a perfect system. We have to a defense-in-depth approach to layer our protections and continuously monitor and grow them.

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

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 at rest describes how data is encrypted as it is written into storage and decrypted as it is read out it. It is a means to protect stored data on disk. When Green Pace stores any sensitive data on its servers, we should be encrypting it. If any malicious actors are able to access the data while it is stored, they cannot read it without the key. Azure uses a symmetric key to both encrypt and decrypt its data. We can adopt a similar model, and we can even partition data and use different encryption keys for each partition. The encryption keys themselves may be encrypted usage a separate key; all keys should be kept in a secure location which only select individuals can access (IBAC).
Encryption at	Encryption at flight refers to the encryption of data that is being transmitted across

a. Encryption	Explain what it is and how and why the policy applies.
flight	networks. This is vital because this is when our data is the most vulnerable. The ways that malicious actors can access data as it moves are myriad – you can even hack the optical fiber itself which carries the information. If they find a way through to our data, we want it to be gibberish without the right keys. Always-on encryption should be our standard when moving data.
Encryption in use	Encryption in use allows us to access and perform tasks with encrypted data. This ensures that data is encrypted at every possible stage. While previous ways to do this involved trusted hardware execution environments, there are new types of software solutions that provide access environments. There are also newer types of encryption like homomorphic encryption which allow computation and data sharing on ciphertext data as though it were plaintext.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Authentication is primarily concerned with confirming the identity of a user or system accessing our networks. Login breaches occur all the time, and in order to mitigate this, we need a robust authentication schema. Certificates, strong username and password requirements, and IP and MAC address verification are among the many tools we can use. Default Deny is our first line of defense! Until authentication can be iron-clad, no user should be allowed access.
Authorization	Authorization refers to whether a user has a given privilege to access or modify parts of a system. We should subscribe to a policy of least-privilege: you only get access to what you need to get whatever task you're working on at the time done. This is a modified RBAC policy, where you only get access to things that fit within your role, and then the system determines what you need based on what you're doing. If you work in accounting, you don't need access to the source code of the company's flagship Android app. Additions of new users or modifying databases are both tasks that only certain users should be able to perform. The access policy needs to be firm and granular yet scalable. Deciding roles and then funneling access through activities will yield better results than a generic RBAC or ABAC policy.
Accounting	Accounting refers to the logging of system activity. The system can log what files are being accessed, what the user role and level of access is of the individual doing so, their location, et cetera. We want granular-level data. This helps administrators have a complete picture of who is accessing what over a given time period, and is invaluable at finding security flaws or as a post-mortem in the case of a breach.

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

User logins
Changes to the database
Addition of new users
User level of access

Files accessed by users

Map the Principles

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

NOTE: Green Pace has already successfully implemented the following:

Operating system logs
Firewall logs
Anti-malware logs

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

Audit Controls and Management

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

Evidence will include the following:

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

Enforcement

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

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

Exceptions Process

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

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

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

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

Distribution

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

Policy Change Control

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

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
2.0	08/07/2022	Edited and expanded principles and coding standards	Mark Webster	Management
3.0	08/12/2022	Filled in automation, Triple A, and other empty sections	Mark Webster	Management

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV