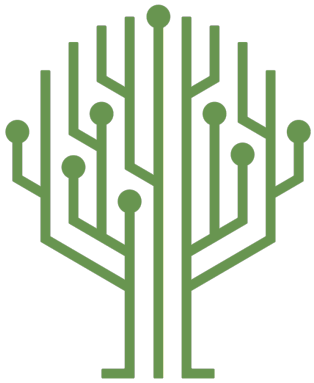


Green Pace Secure Development Policy

Michael Quilici

CS 405 - Module Six
Southern New Hampshire University
April 5, 2021

Contents

Overview 2
Purpose 2
Scope 2
Module Three Milestone 2
Ten Core Security Principles 2
C/C++ Ten Coding Standards 4
Coding Standard 1 **Error! Bookmark not defined.**
Coding Standard 2 6
Coding Standard 3 8
Coding Standard 4 10
Coding Standard 5 12
Coding Standard 6 14
Coding Standard 7 16
Coding Standard 8 19
Coding Standard 9 21
Coding Standard 10 23
Defense-in-Depth Illustration 25
Project One 25
1. Revise the C/C++ Standards 25
2. Risk Assessment 25
3. Automated Detection 25
4. Automation 25
5. Summary of Risk Assessments 27
6. Create Policies for Encryption and Triple A 28
7. Map the Principles 29
Audit Controls and Management 31
Enforcement 31
Exceptions Process 31
Distribution 32
Policy Change Control 32
Policy Version History 32
Appendix A Lookups 32
Approved C/C++ Language Acronyms 32

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 (SEI CERT, 2020)

Principles	Description
1. Validate Input Data	Validating input data from untrusted sources eliminates the majority of software security vulnerabilities. Do not trust input data from command line arguments, network interfaces, environmental variables, or user-controlled files as these can be manipulated. Failure to validate input can leave a program susceptible to buffer overflows and SQL injections.
2. Heed Compiler Warnings	Compilers will generate warnings when there are problems with code. Compile code with the highest warning level available to display additional warnings and then modify the code where necessary to eliminate the warnings. Use static and dynamic analysis tools to uncover additional security flaws.
3. Architect and Design for Security Policies	Implement a software architecture that enforces security policies. If code requires multiple privilege levels, divide the program into separate subsystems that each have their own privilege set.
4. Keep It Simple	Keep software design as simple and small as possible. Adding complexity to a design increases the likelihood that errors will be made in the implementation, configuration, and use. Complex designs also make security mechanisms more complex and increase the effort needed to achieve an appropriate level of assurance.
5. Default Deny	When implementing a protection scheme, base access decisions on permission rather than exclusion. Deny access by default and create a protection scheme that identifies the conditions under which access is permitted.
6. Adhere to the Principle of Least Privilege	The principle of least privilege states that processes should always be run with the lowest level of permission necessary to complete a job. When elevated privileges are required, ensure they are only used for the least amount of time needed for the privileged task.
7. Sanitize Data Sent to Other Systems	Data sent so complex subsystems like command shells and SQL databases must be properly formatted and cleaned of extraneous and potentially harmful commands. For

Principles	Description
	example, an unchecked database query can be manipulated by an attacker to perform an SQL injection.
8. Practice Defense in Depth	Defense in depth is a software security strategy that utilizes multiple overlapping layers of defense to protect sensitive data. When developing software, utilize multiple defensive strategies so that if one layer is compromised, another layer can prevent a vulnerability from being exploited.
9. Use Effective Quality Assurance Techniques	Use effective quality assurance techniques like fuzz testing, penetration testing, and source code audits to effectively identify and eliminate vulnerabilities. Have external reviewers perform independent security reviews to identify and correct invalid assumptions made during development.
10. Adopt a Secure Coding Standard	Coding standards consist of coding rules, guidelines, and best practices for a given programming language. Follow a coding standard for cleaner, more consistent code that is more secure and has an accelerated time to market.

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	Rationalization
Data Type	[STD-001-CPP]	Conversion from one data type of a given width to another data type of a smaller width can cause loss of high-order bits unless the magnitude of the value is small enough to be correctly represented. Similarly, converting from a signed data type to an unsigned data type can cause the resulting value to be misinterpreted. (SEI CERT, 2020)

Noncompliant Code

In this example, a signed integer with a value of -1 is converted to an unsigned integer, resulting in the unsigned integer taking on a large positive value. Because the new type is unsigned, the value is converted by repeatedly adding or subtracting 1 more than the maximum value that can be represented in the new type until the value is in the range of the new type.

```
#include <iostream>

int main()
{
    signed int si = -1;
    unsigned int ui;
    ui = (unsigned int)si;
}
```

Compliant Code

In this example, an unsigned data type is checked to ensure that it is within the range of the new unsigned data type before conversion. If the value cannot be represented in the unsigned data type, an error is printed.

```
#include <iostream>
#include <climits>

int main()
{
    signed int si = -1;
    unsigned int ui;
```

Compliant Code

```
    if (0 <= si && si <= UINT_MAX) {
        ui = (unsigned int)si;
    } else {
        std::cout << "Error: Number out of range.";
    }
}
```

Principles(s): (1) Validate Input: – One area where incorrect data types can cause a problem is input. For example, if a program prompts the user for an integer using std::cin and the user enters a character, the program may crash. To avoid this problem, input should be validated to ensure the entered value is of the correct type.

(2) Heed compiler warnings: Compilers and static analysis tools will often issue errors when there is a problem with data type conversion. For example, if a Boolean value is compared with an integer value like 2, Visual Studio will indicate that the comparison is always false because a Boolean can only be 0 or 1.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	2020.2	LANG.CAST.COERCE LANG.CAST.VALUE	Coercion alters value. Cast alters value.
Coverity*	2017.07	MISRA_CAST	Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted.
Polyspace Bug Finder	R2020a	CERT C: Rule INT31-C	Sign change integer conversion overflow.
Parasoft C/C++test	2020.2	CERT_C-INT31-c	The value of an expression shall not be assigned to an object with a narrower essential type. The value of an expression shall not be assigned to an object of a different essential type category.

Coding Standard 2

Coding Standard	Label	Rationalization
Data Value	[STD-002-CPP]	Data values should be checked to ensure that numerical overflow/underflow does not occur during operations. Operations like addition, subtraction, and multiplication can cause a value to exceed its numerical limits and wrap around. (SEI CERT, 2020)

Noncompliant Code

In this example, an integer is repeatedly added to itself in a loop. Prior to the final iteration, the integer exceeds the maximum value for a 32-bit signed integer and wraps around to a negative value.

```
#include <iostream>

int main()
{
    int num = 3;
    for (int i=0; i < 30; i++) {
        num = num + num;
    }
}
```

Compliant Code

In this example, a precondition check is used to determine if the addition operation results in an overflow. If an overflow does occur, an error is printed. (SEI CERT, 2020)

```
#include <iostream>

int main()
{
    int *result = new int[(sizeof(int))];
    int num = 3;
    for (int i=0; i < 30; i++) {
        if (__builtin_sadd_overflow(num, num, result)) {
            std::cout << "Error: Overflow occurred";
        } else {
            num = *result;
        }
    }
}
```

Principles(s): (1) Validate Input: One area where incorrect data values can cause a problem is input. For example, if a program prompts the user for an integer using std::cin and the user enters a value that is too large for an integer, an overflow will occur. To avoid this problem, input should be validated to ensure the entered value is not too large for the data type.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	integer-overflow	Fully checked
CodeSonar	6.0p0	ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD	Addition overflow of allocation size Integer overflow of allocation size Addition overflow of size Unreasonable size argument
Parasoft C/C++test	2020.2	CERT_C-INT32-b	Integer overflow or underflow in constant expression in '+', '-', '*' operator
Polyspace Bug Finder	R2020a	CERT C: Rule INT32-C	Checks for: • Integer overflow • Tainted division operand • Tainted modulo operand Rule partially covered.

Coding Standard 3

Coding Standard	Label	Rationalization
String Correctness	[STD-003-CPP]	String correctness can take many forms. One common error is to copy string data to a buffer that is not large enough to hold the data resulting in a buffer overflow. (SEI CERT, 2020)

Noncompliant Code

In this example, a C-string with five elements is used to store input from std::cin. If more than five characters are entered, a buffer overflow will occur.

```
#include <iostream>

int main()
{
  char buf[5];
  std::cin >> buf;
}
```

Compliant Code

In this example, a std::string is used instead of a bounded array to guard against buffer overflows and to ensure that data is not truncated.

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

int main() {
  std::string input;
  std::cin >> input;
}
```

Principles(s): (1) Validate Input: To ensure that string input is the correct format and length, input should be validated. For example, if a form asks for a phone number, the string entered by the user should contain only numbers and dashes and have a specific length.

(7) Sanitize data sent to other systems: String correctness is also important when passing data to other systems. For example, a search query passed to an SQL database should be checked for SQL injection before execution.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.0p0	MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO	No space for null terminator Buffer overrun Type overrun
LDRA tool suite	9.7.1	489 S, 66 X, 70 X, 71 X	Partially implemented
Parasoft C/C++test	2020.2	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	R2020a	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.

Coding Standard 4

Coding Standard	Label	Rationalization
SQL Injection	[STD-004-CPP]	An SQL injection modifies an SQL statement in a way that causes undesired operation or reveals data that the user should not be allowed to access. To guard against SQL injection, input should be verified for statements that are indicative of an injection before executing an SQL command. (SEI CERT, 2021)

Noncompliant Code

In this example, an SQL query string "sql" is passed to the run_query function is used as an argument for the sqlite3_exec method without first checking for injection.

```
bool run_query(sqlite3* db, const std::string& sql, std::vector<
user_record >& records)
{
  records.clear();
  char* error_message;
  if(sqlite3_exec(db, sql.c_str(), callback, &records, &error_message) !=
SQLITE_OK)
  {
    std::cout << "Data failed to be queried from USERS table. ERROR = "
<< error_message << std::endl;
    sqlite3_free(error_message);
    return false;
  }
  return true;
}
```

Compliant Code

In this example, a regular expression (regex) is used to scan the SQL query for an injection of the form "or value=value" and print an error message if an injection is found.

```
bool run_query(sqlite3* db, const std::string& sql, std::vector<
user_record >& records)
{
  std::regex reg(" or\\W+(\\w*)(?:\\W+\\1)",
std::regex_constants::icase);
  std::smatch mat;
  if (std::regex_search (sql, mat, reg)) {
    std::cout << "Suspected SQL Injection" << std::endl;
    return false;
  }
  records.clear();
```

Compliant Code

```
  char* error_message;
  if(sqlite3_exec(db, sql.c_str(), callback, &records, &error_message) !=
SQLITE_OK)
  {
    std::cout << "Data failed to be queried from USERS table. ERROR = "
<< error_message << std::endl;
    sqlite3_free(error_message);
    return false;
  }
  return true;
}
```

Principles(s): (7) Sanitize data sent to other systems: SQL is vulnerable to injection attacks using maliciously-crafted query strings. One way to guard against this is to sanitize query strings and prevent them from being executed. For example, a program can check user input for a query containing the string “or value=value” and issue an injection attack error.

(9) Use effective quality assurance techniques: Testing is also important for determining susceptibility to SQL injection attacks. Fuzz testing and penetration testing can be used to check a variety of input conditions to ensure that they are handled properly by the program.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	P12	L1

Automation

Tool	Version	Checker	Description Tool
The Checker Framework	2.1.3	Tainting Checker	Trust and security errors (see Chapter 8)
Coverity	7.5	SQLI FB.SQL_PREPARED_STATEMENT_GENERATED_ FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	Implemented
Findbugs	1.0	SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	Implemented
Fortify	1.0	HTTP_Response_Splitting SQL_Injection__Persistence SQL_Injection	Implemented

Coding Standard 5

Coding Standard	Label	Rationalization
Memory Protection	[STD-005-CPP]	It is necessary to properly pair memory management functions. Unlike the C programming language, C++ provides multiple ways to allocate and deallocate memory. For example, the allocator new() is paired with the deallocator delete() and the array allocator new[] is paired with the deallocator delete[]. One should never use the C memory deallocation function std::free() on resources allocated using C++ memory allocation functions. (SEI CERT, 2020)

Noncompliant Code

In this example, the C dellocator free() is used incorrectly to free memory allocated using the C++ new() allocator. Calling free() on an object allocated with new() is problematic because free() does not invoke the object's destructor.

```
#include <iostream>

int main() {
    int *ip = new int(5);
    …
    free(ip);
}
```

Compliant Code

In this variation, memory is allocated for an integer array using new int[10] and then deallocated using the appropriate delete[] operator.

```
#include <iostream>

int main() {
    int *ap = new int[10];
    …
    delete[] ap;
}
```

Principles(s): (10) Adopt a secure coding standard: Applying a secure coding standard, like CERT, to the target development language can help prevent vulnerabilities from incorrectly deallocating memory from occurring. Static analysis tools like Cppcheck can discover many of these vulnerabilities.

(2) Heed compiler warnings: Compilers may also issue a warning for many types of memory errors. For example, Visual Studio will warn if there is a dangling pointer. If developers see warnings like this, the errors should be fixed.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
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	6.0p0	ALLOC.FNH ALLOC.DF ALLOC.TM	Free non-heap variable Double free Type mismatch
Parasoft C/C++test	2020.2	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 assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources
Polyspace Bug Finder	R2020a	CERT C++: MEM51-CPP	Checks for: • Invalid deletion of pointer • Invalid free of pointer • Deallocation of previously deallocated pointer Rule partially covered.

Coding Standard 6

Coding Standard	Label	Rationalization
Assertions	[STD-006-CPP]	Assertions are used to test assumptions within a section of code. They take the form assert(expression) where the expression is true unless there is a bug in the program. If the expression evaluates as false, the program is terminated and an error message is displayed. Assertions should never be used to validate method arguments, rather, erroneous arguments should result in an appropriate run-time exception. An assertion failure will not throw an appropriate exception. (SEI CERT, 2020)

Noncompliant Code

In this example, an assertion is used to check for numerical overflow in a function that adds two numbers. While the assertion is reasonable, the validation code is not executed when assertions are disabled.

```
#include <iostream>
#include <climits>
#include <assert.h>

int addNums(int x, int y) {
    assert (x < INT_MAX - y);
    return x + y;
}

int main() {
    std::cout << addNums(INT_MAX,1);
}
```

Compliant Code

In this example, the assertion is replaced with a conditional statement that throws an exception when an overflow occurs. A try-catch block in the main function catches the exception and prints an appropriate error message.

```
#include <iostream>
#include <climits>
#include <assert.h>
#include <stdexcept>

int addNums(int x, int y) {
    if (x > INT_MAX - y) {
        throw std::overflow_error("overflow error");
    }
    return x + y;
}
```

Compliant Code

```
int main() {
    try {
        int result = addNums(INT_MAX,1);
    } catch(const std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

Principles(s): (10) Adopt a secure coding standard: Applying a secure coding standard, like CERT, to the target development language can help prevent side effects from occurring in assertion statements. Static analysis tools like Cppcheck can identify when assertions are used incorrectly.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

Automation

Tool	Version	Checker	Description Tool
Google Test	1.10.x	NA	NA
Microsoft Unit Testing Framework for C++	VS 2017 and later	NA	NA
Criterion	3.0.0	NA	NA
libcester	0.4	NA	NA

Coding Standard 7

Coding Standard	Label	Rationalization
Exceptions	[STD-007-CPP]	Exceptions are used to indicate when errors occur during the execution of a program. When an exception is thrown, control is transferred to the nearest handler with a type that matches the exception. If no matching handler is found, std::terminate() is called. Whether the stack is unwound or not before this call is implementation-defined. For this reason, it is important to handle all exceptions. (SEI CERT, 2020) Also, avoid error hiding, which is the practice of catching an exception or error and continuing without reporting that an error occurred. Error hiding is considered an anti-pattern and is bad practice.

Noncompliant Code

In this example, the addNums function throws an exception when the addition of two numbers results in an overflow. However, no handler exists to handle the exception so std::terminate() is called and the thread ends abruptly.

```
#include <iostream>
#include <climits>
#include <assert.h>
#include <stdexcept>

int addNums(int x, int y) {
    if (x > INT_MAX - y) {
        throw std::overflow_error("overflow error");
    }
    return x + y;
}

int main() {
    int result = addNums(INT_MAX,1);
}
```

Compliant Code

In this example, a try-catch block is used with the addNums function to handle an overflow error if it occurs.

```
#include <iostream>
#include <climits>
#include <assert.h>
#include <stdexcept>

int addNums(int x, int y) {
    if (x > INT_MAX - y) {
        throw std::overflow_error("overflow error");
```

Compliant Code

```
    }
    return x + y;
}

int main() {
    try {
        int result = addNums(INT_MAX,1);
    } catch(const std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

Noncompliant Code

In this example, a try-catch block is used, but no error is printed. This is known as error hiding.

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

int main() {
    std::string s("12345");
    try {
        std::cout << s.at(5);
    } catch (...) {
        // do nothing
    }
    // more code...
}
```

Compliant Code

In this example, an out-of-range error is printed instead of hiding the error.

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

int main() {
    std::string s("12345");
    try {
        std::cout << s.at(5);
    } catch (std::out_of_range &) {
        std::cerr << "Out of range error";
```

Compliant Code

```
    }
    // more code...
}
```

Principles(s): (9) Use effective quality assurance techniques: Compilers and static analysis tools do not always identify every error in a program. For example, Cppcheck does not indicate that the above example is missing a handler. To find errors like this, quality assurance techniques like source code audits can be used. Reviews and audits should be part of an effective quality assurance plan.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	main-function-catch-all early-catch-all	Partially checked
LDRA tool suite	9.7.1	527 S	Partially implemented
Parasoft C/C++test	2020.2	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	R2020a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)

Coding Standard 8

Coding Standard	Label	Rationalization
File Handling	[STD-008-CPP]	Files should be closed when they are no longer needed. A call to the std::basic_filebuf<T>::open() function should be matched with a call to std::basic_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first. (SEI CERT, 2020)

Noncompliant Code

In this example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic_filebuf<T>::open(), and the default std::terminate_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic_filebuf<T> object maintained by the object is not properly closed. (SEI CERT, 2020)

```
#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();
}
```

Compliant Code

In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. (SEI CERT, 2020)

```
#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();
}
```

Principles(s): (1) Validate input: File handling is an area where many security vulnerabilities can arise. For example, when saving or reading files, file names should be validated. Like with SQL injection, path names can be manipulated to gain access to, and modify sensitive files. Also, when reading files, a program should account for the possibility of formatting errors, corrupted information, or the file not existing.

(6) Adhere to the principle of least privilege: File handling should adhere to the principle of least privilege. For example, if a program needs elevated privileges to write files, then permission should only be provided for the least amount of time necessary to complete the task.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.0p0	ALLOC.LEAK	Leak
Parasoft C/C++test	2020.2	CERT_CPP-FIO51-a	Ensure resources are freed
Parasoft Insure++	NA	NA	Runtime detection
Polyspace Bug Finder	R2020a	CERT C++: FIO51-CPP	Checks for resource leak (rule partially covered)

Coding Standard 9

Coding Standard	Label	Rationalization
Containers	[STD-009-CPP]	Container indices and iterators should be within their valid range. The programmer should ensure that array references are within the bounds of the array. Similarly, integer indexes for standard template library vectors should be within the bounds of the vector. (SEI CERT, 2020)

Noncompliant Code

In this example, a function, insert_in_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. (SEI CERT, 2020)

```
#include <cstddef>

void insert_in_table(int *table, std::size_t tableSize, int pos, int
value) {
  if (pos >= tableSize) {
    // Handle error
    return;
  }
  table[pos] = value;
}
```

Compliant Code

In this compliant solution, the parameter pos is declared as size_t, which prevents the passing of negative arguments. (SEI CERT, 2020)

```
#include <cstddef>

void insert_in_table(int *table, std::size_t tableSize, std::size_t pos, int
value) {
  if (pos >= tableSize) {
    // Handle error
    return;
  }
  table[pos] = value;
}
```

Principles(s): (1) Validate input: To ensure containers like arrays are not accessed out of bounds, input should be validated. For example, if a program iterates over the characters of a user input string, then one must ensure that the loop index data type can accommodate the number of characters.

(9) Use effective quality assurance techniques: It is important to test boundary conditions to ensure that container indices are valid. One approach is to use fuzz testing, whereby invalid and random data is sent to a program. The program is then monitored for crashes, exceptions, and failed assertions.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.0p0	LANG.MEM.BO LANG.MEM.BU LANG.MEM.TO LANG.MEM.TU LANG.MEM.TBA LANG.STRUCT.PBB LANG.STRUCT.PPE	Buffer overrun Buffer underrun Type overrun Type underrun Tainted buffer access Pointer before beginning of object Pointer past end of object
LDRA tool suite	9.7.1	45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X	Partially implemented
Parasoft C/C++test	2020.2	CERT_CPP-CTR50-a	Guarantee that container indices are within the valid range
Polyspace Bug Finder	R2020a	CERT C++: CTR50-CPP	Checks for: • Array access out of bounds • Array access with tainted index • Pointer dereference with tainted offset Rule partially covered.

Coding Standard 10

Coding Standard	Label	Rationalization
Memory Management	[STD-010-CPP]	Dangling pointers are pointers to memory that has been deallocated. Attempting to access a pointer that has been deallocated is undefined behavior and can result in exploitable vulnerabilities. (SEI CERT, 2020)

Noncompliant Code

In this example, the pointer “s” is dereferenced after it has been deallocated. If this access results in a write-after-free, vulnerability that can be exploited to run arbitrary code with the permissions of the vulnerable process. (SEI CERT, 2020)

```
#include <new>

struct S {
  void f();
};

void g() noexcept(false) {
  S *s = new S;
  // ...
  delete s;
  // ...
  s->f();
}
```

Compliant Code

In this example, automatic storage duration is used instead of dynamic storage duration. Since s is not required to live beyond the scope of g(), this compliant solution uses automatic storage duration to limit the lifetime of s to the scope of g(). (SEI CERT, 2020)

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

void g() {
  S s;
  // ...
  s.f();
}
```

Principles(s): (9) Use effective quality assurance techniques: The use of static testing and source code audits can uncover errors like dangling pointers. For example, when attempting to use a dangling pointer, Cppcheck will issue an error.

(10) Adopt a secure coding standard: Applying a secure coding standard like CERT will help memory errors like dangling pointers from occurring.

Threat Level

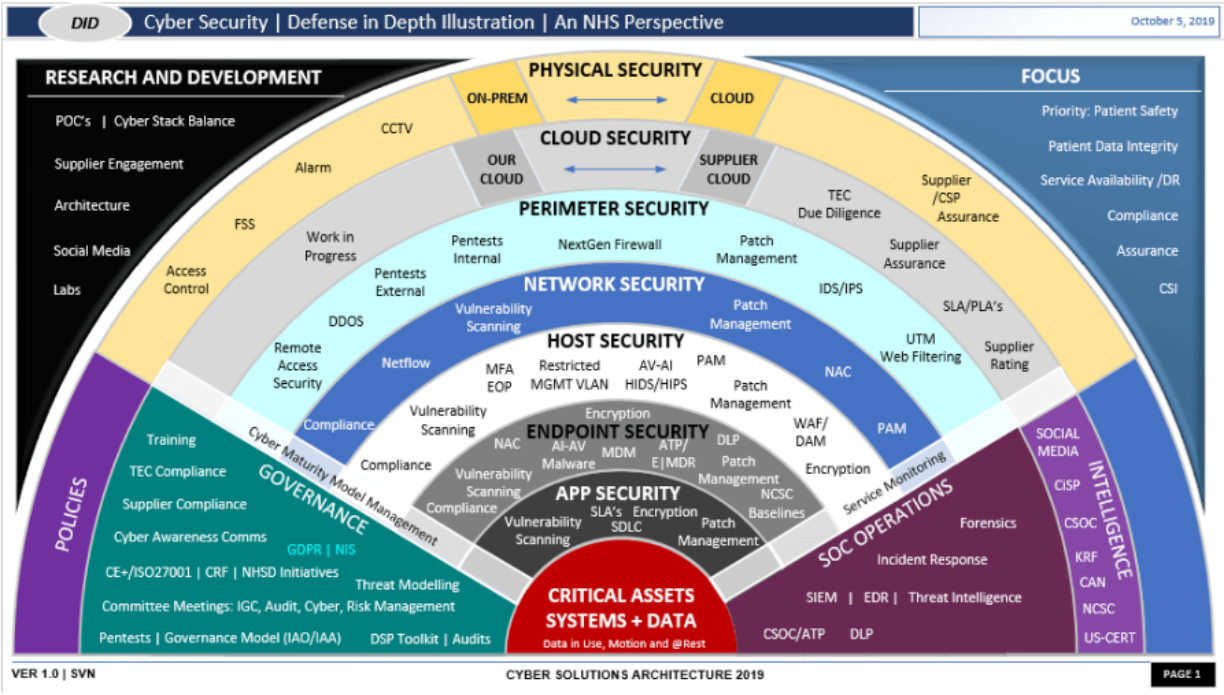
Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.0p0	ALLOC.UAF	Use after free
Coverity	v7.5.0	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
Parasoft C/C++test	2020.2	CERT_CPP-MEM50-a	Do not use resources that have been freed
Polyspace Bug Finder	R2020a	CERT C++: MEM50-CPP	Checks for: • Pointer access out of bounds • Deallocation of previously deallocated pointer • Use of previously freed pointer Rule partially covered.

Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



Project One

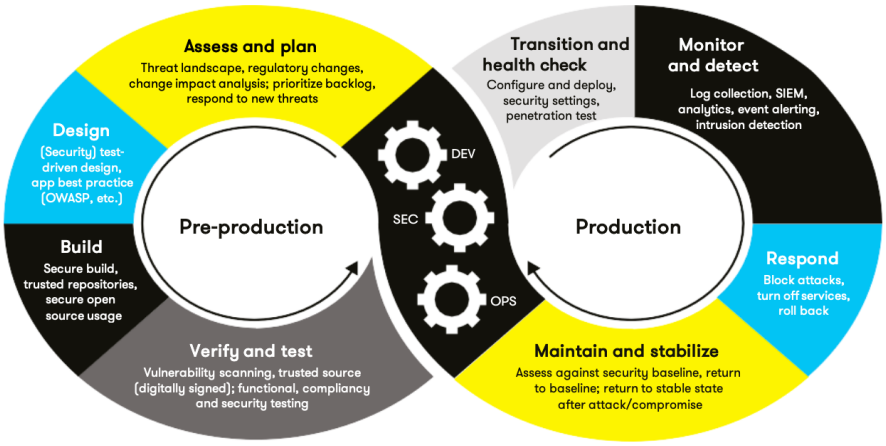
There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

1. **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.

2. **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.

3. **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.

4. **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.

- **Assess and Plan**: The assessment and planning phase is not easily automated as it involves assessing the threat landscape (treat modeling), regulatory changes, change impact analysis, prioritizing the backlog (Agile/Scrum), and determining how to respond to new threats. The security infrastructure should also be considered. For example, Active Directory (AD) or similar directory services system using Lightweight Directory Access Protocol (LDAP) can be used for authentication, group and user management, and policy administration.

- **Design**: The design phase includes test-driven design and following best practices. This stage can be automated using the IDE and plug-ins. For example, the Google Test framework can be used in test-driven design to create unit tests that enforce some of the standards in this policy. Also, incorporating assert statements in the code allows specific conditions to be enforced throughout the development process. Lastly, coding best practices can be enforced with plugins like StyleCop.

- **Build**: The build process can be automated using a build-automation server like Jenkins. Jenkins facilitates the continuous integration and delivery of software by automating parts of the build, test, and deployment process. Continuous integration is a process where development work is integrated as early as possible. This allows errors to be identified and fixed sooner. For example, Jenkins can execute can be set up to compile an application after a new commit to a Git repository or after a certain timeframe. Jenkins also includes plugins like Cppcheck, which scans report files in the build workspace and reports issues detected during code analysis.

- **Verify and Test**: The testing process includes static application security testing (SAST), dynamic application security testing (DAST), and interactive application security testing (IASP). SAST is used to find security vulnerabilities in the application source code earlier in the software development lifecycle. DAST does not use the source code but instead finds security vulnerabilities in a running application. DAST uses fault techniques to feed malicious data to the software to identify vulnerabilities like SQL injection. Because DAST requires a running application, it is used near the

end of the development cycle. IAST is a more modern security testing approach that places an agent within an application to perform analysis in real-time and collect data on security events.

- **Transition and Health Check**: Penetration testing, or pen testing, is used to gather information about a target, identify entry points, and attempt to break in. Pen testing can be used to identify weak points in security and measure compliance with security policies. One common framework used for pen testing is Metasploit. Metasploit contains a collection of tools that contains over a thousand exploits on multiple platforms including Android, Python, and Java. Metasploit.

- **Monitor and Detect**: Automated security information and event management software (SIEM) is used to analyze and report on log data in real-time. The software identifies security incidents like failed logins and malware activity and can send alerts based on defined rules.

- **Respond**: The response to an attack can involve blocking the source of the attack and turning off services. For example, if monitoring software discovers numerous failed attempts to gain access to a secured location, the account can be blocked to prevent the attacks from continuing. Tools that can be used in this situation include Norton Internet Security and Malwarebytes Anti-Exploit Premium.

- **Maintain and Stabilize**: After an attack or compromise, the goal is to return to a stable state of operation. This is not easily automated as what needs to be done depends on the nature of the attack. For example, if a network was compromised, the company should investigate what happened. This can involve taking parts of the network offline until a solution is found.

The components of this automation plan exemplify how defense-in-depth can be used to provide multiple layers of security at all stages of the preproduction and production pipeline. Layers of security range in scale from code-level static analysis tools to organization-wide directory access controls. This ensures that if one layer is compromised, there will be additional layers in place to prevent attackers from progressing

5. **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	Probable	High	P6	L2
STD-002-CPP	High	Likely	High	P9	L2
STD-003-CPP	High	Likely	Medium	P18	L1
STD-004-CPP	High	Probable	Medium	P12	L1
STD-005-CPP	High	Likely	Medium	P18	L1
STD-006-CPP	High	Likely	Medium	P18	L1
STD-007-CPP	Low	Probable	Medium	P4	L3
STD-008-CPP	Medium	Unlikely	Medium	P4	L3
STD-009-CPP	High	Likely	High	P9	L2
STD-010-CPP	High	Likely	Medium	P18	L1
STD-011-CPP					

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-012-CPP					
STD-013-CPP					
STD-014-CPP					

6. **Create Policies for Encryption and Triple A**
   Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided**.**
   a. Explain each type of encryption, how it is used, and why and when the policy applies.
   b. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

a. Encryption	**Explain what it is and how and why the policy applies.**
Encryption at rest	Encryption at rest refers to the encryption of stored data such as files on a hard drive or in the cloud. Use encryption at rest to securely store files containing sensitive information. Storage devices should utilize full-disk encryption to prevent attackers from retrieving information even if a device is stolen or compromised. The policy applies whenever a device is used to store sensitive information like emails, credit card numbers, Social Security numbers, etc. It also applies in circumstances where it is necessary to ensure compliance with government regulations such as HIPAA, PCI, and FedRAMP. (Microsoft, 2020) For example, HIPAA encryption requirements state that entities should “implement a mechanism to encrypt PHI (personal healthcare information) whenever deemed appropriate.” The recommendation is intentionally vague because it is recognized that encryption standards considered appropriate today may not be in the future as technology advances and criminals become more sophisticated. (HIPAA Journal, 2021)
Encryption in flight	Encryption in flight refers to the encryption of transmitted data such as web-based communications. One example of encryption in flight is HTTPS. HTTPS uses a secure transport layer security mechanism called SSL/TLS to encrypt HTTP traffic. Another example of encryption in flight is a virtual private network (VPN). VPNs encrypt network traffic and enable secure communications from outside an organization’s network. Encryption in flight should always be used when sensitive data is being transmitted. For example, if an organization allows employees to access its networks remotely, then encryption in flight should be used to ensure communications are secure. Without encryption in flight, communications could be subject to eavesdropping and man-in-the-middle attacks.
Encryption in use	Encryption in use refers to the encryption of “data in use” or memory. Memory can contain sensitive data like passwords, encryption keys, and personal information and is vulnerable to unauthorized access and theft. For example, if an attacker has physical access to a device, they can perform a memory dump and parse the data for encryption keys to gain access to encrypted data at rest and data in motion. Also, a technique known as memory scraping has been used in the past to read customer transaction information from point-of-sale terminals. To guard against this, encryption-in-use should be used to protect devices that hold sensitive information in memory that may be physically compromised.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Authentication is the first step in the Triple-A framework and refers to the process of verifying the identity of a user. Methods of authentication include passwords, biometrics, and multi-factor authentication. When using passwords for authentication, ensure that passwords have sufficient complexity, are not repeated, and are changed regularly. Also, limit the number of user login attempts to a few tries before locking the account to prevent brute-force attacks. This policy applies to all computers within an organization. Without strong authentication, attackers may be able to guess a user's password and gain access to sensitive information.
Authorization	Authorization refers to limiting the locations and resources to which a given user has access. For example, a systems administrator can add new users to a network, but a typical user should not be able to add new users. Similarly, a database administrator can make changes to a database, but a user might only have read access or no access at all. Authorization should be used on all systems where user access should be limited to prevent access to sensitive information or modification of the system. User-level access should be restricted to the fewest privileges necessary for users to do their work. This ensures that if an account is compromised the resources to which the attacker has access will be restricted.
Accounting	Accounting refers to recording usage information like who is accessing a system, what commands are issued, session duration, and data being sent and received. Network security tools with accounting can be used to detect and flag suspicious activity and send alerts to IT. These tools should be installed on networks that contain sensitive information such as customer records to thwart attacks and prevent data breaches. For example, files accessed by users can be monitored and an alert issued if unauthorized access is attempted.

*****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 ✓

1. **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	03/08/2021	First Draft	Michael Quilici	
1.2	04/05/2021	Final	Michael Quilici	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV

Works Cited

HIPAA Journal. (2021). *HIPAA Encryption Requirements*. Retrieved April 5, 2021, from HIPAA Journal: https://www.hipaajournal.com/hipaa-encryption-requirements/

Microsoft. (2020). *Azure Data Encryption-at-Rest*. Retrieved 3 5, 2021, from Microsoft Docs: https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-atrest#:~:text=Encryption%20at%20rest%20is%20designed,encryption%20to%20read%20the%20data.

SEI CERT. (2020). *CTR50-CPP*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range

SEI CERT. (2020). *ERR51-CPP*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions

SEI CERT. (2020). *FIO51-CPP*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed

SEI CERT. (2020). *INT31-C*. Retrieved March 15, 2021, from SEI CERT: https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data

SEI CERT. (2020). *INT32-C*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow

SEI CERT. (2020). *MEM50-CPP*. Retrieved March 15, 2020, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

SEI CERT. (2020). *MEM51-CPP*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources

SEI CERT. (2020). *MET01-J*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/java/MET01-J.+Never+use+assertions+to+validate+method+arguments

SEI CERT. (2020). *STR50-CPP*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

SEI CERT. (2020). *Top 10 Secure Coding Practices*. Retrieved March 15, 2021, from SEI Wiki Home: https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

SEI CERT. (2021). *IDS00-J*. Retrieved March 15, 2021, from SEI External Wiki Home: https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection