**Green Pace Developer: Security Policy Guide Template**



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 12](#_Toc52464067)

[Coding Standard 10 13](#_Toc52464068)

[Defense-in-Depth Illustration 14](#_Toc52464069)

[Project One 14](#_Toc52464070)

[1. Revise the C/C++ Standards 14](#_Toc52464071)

[2. Risk Assessment 14](#_Toc52464072)

[3. Automated Detection 14](#_Toc52464073)

[4. Automation 14](#_Toc52464074)

[5. Summary of Risk Assessments 15](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 15](#_Toc52464076)

[7. Map the Principles 16](#_Toc52464077)

[Audit Controls and Management 17](#_Toc52464078)

[Enforcement 17](#_Toc52464079)

[Exceptions Process 17](#_Toc52464080)

[Distribution 18](#_Toc52464081)

[Policy Change Control 18](#_Toc52464082)

[Policy Version History 18](#_Toc52464083)

[Appendix A Lookups 18](#_Toc52464084)

[Approved C/C++ Language Acronyms 18](#_Toc52464085)

# 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](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

# 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. ValidateInput Data | Validating input data refers to the proper testing of user or software input. By validating input, it would prevent potentially malicious or malformed data from entering a system or database. If data is not properly validated, it could lead to potential vulnerabilities such as overflows or a program malfunction. |
| 1. Heed Compiler Warnings | Heed complier warnings refers to compiling code with the highest warning level available and then modifying the code to resolve the warnings. This would ensure that potential security flaws and bugs are resolved to provide secure code. |
| 1. Architect and Design for Security Policies | This security standard revolves around creating a software architecture and design that would accommodate and implement the necessary security measures. If the software architecture and design does not take into account the security measures needed for the program/application, there would be open vulnerabilities that could be exploited and compromised. |
| 1. Keep It Simple | Keep the code as simple as possible. Too complex code would require unnecessary work and modifications that could create vulnerabilities and errors. At the same time, complex code would increase the difficulty of managing and maintaining the program. It would be harder to implement bug fixes and patches if the code is too complex. |
| 1. Default Deny | Bases access decisions on permissions instead of exclusions. Access would be denied if the inbound/outbound traffic is not specified on the list of permissions. |
| 1. Adhere to the Principle of Least Privilege | By adhering to the principle of least privilege, it only allows enough access or permissions to complete a job or task. For a specific job, permissions and/or access should not exceed the minimum required to complete the task. This would reduce the risks of exploiting low access accounts to compromise the system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data that is sent to other systems would aid in preventing injection attacks. In attacks such as SQL Injections, invalid or illegal code is inserted into data fields to be executed. Some SQL injection can be in the form of “ OR 1=1;”, or any other statement that returns true. By properly sanitizing data sent to other systems, it prevents illegal statements or code from compromising the system. |
| 1. Practice Defense in Depth | Defense in depth (DiD) refers to a layered security strategy to protect data and information. The objective of DiD is to provide layered redundant, security mechanisms that would replace each other if one failed. For instance, if the first layer of defense is compromised, another is there to take its place. This increases the overall security of the system. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques refer to techniques and concepts that revolve around maintaining a secure and functional program and/or application. Some of these techniques are the implementation of pen testing, source code audits, fuzz testing, etc. that would be effective in identifying and locating potential security flaws and eliminating them . |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves the development and/or application of secure coding standards for the target programming language and platform. These are rules and guidelines that would be used to help to prevent potential exploits and compromise of the application or program through coding vulnerabilities. |

## C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-clg] | It is vital for secure coding to select the most appropriate data types. For instance, an unsigned integer should only be assigned to represent integer values that would never be negative, while signed integers should be used to represent negative integer values. At the same time, it is important to use the smallest signed or unsigned type that can represent a given range of values for a given variable. By doing so, it would help to save memory storage and prevent wraparounds and overflows. |

| **Noncompliant Code** |
| --- |
| The code block example depicts an operation between the unsigned integers A and B, which is then stored into the unsigned integer, sum. There is no check to determine if the memory storage in sum is sufficient to store the operation between A and B. If the memory storage is insufficient, it could result in a wraparound, which then can open a vulnerability. |
| //addition example  void example(unsigned int A, unsigned int B) {  //there is no function to determine if the data storage in sum is sufficient for the operation: A + B.  //If the storage in insufficient, it could result in a wraparound.  unsigned int sum = A + B;  /\* …\*/  } |

| **Compliant Code** |
| --- |
| Unlike the previous code block, this code block uses an if-else statement to determine if the memory in the variable sum is sufficient enough to hold the operation between A and B. If the variable B has a greater value than the operation between the max data type and A, it would result in an error. However, if B is less than the operation between A and the max data type, it would assign the operation between A and B to the variable, sum. |
| //addition example  void example(unsigned int A, unsigned int B){  unsigned int sum;  //the if statement is used to determine if the value of  //one of the operators( A or B) is greater than an operation between the max //data type and the other operator.  if(UINT\_MAX – A < B){  //handle errors  //if B is less than UNIT\_MAX – A, the program will assign the operation (A + B)  // to the variable, sum.  }else{  sum = A + B;  }  //continue program  } |

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

|  |
| --- |
| **Principles(s):** Validate Input Data  The principle, Validate Input Data, maps to this coding standard as it describes the necessity of using proper input. If input data is not properly validated, it leads to overflow and other types of vulnerabilities. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-int30 | Implemented |
| Coverity | 2017.07 | Integer-overflow | Implemented |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-cpp  ] | Data validation is an important step in secure coding. This is to ensure that application and/or user input is valid. If invalid data is submitted, it could open vulnerabilities used to view the memory contents, write arbitrary memory locations, and even crash vulnerable processes. |

| **Noncompliant Code** |
| --- |
| This noncompliant asks the user their age and returns a number of statements depending on the value submitted. Although it uses a few value checks to prevent invalid integers such as negative integers from being submitting, it does not ensure that the user input is an integer. By sending a character or string input, it causes the program to continuously run. |
| int main()  {  signed int age;  //there is no input validation that would test for invalid input data  std::cout << "enter user age :";  std::cin >> age;  if (age < 15)  std::cout << "Invalid Age";  else if (age > 16)  std::cout << "Valid age for driving";  else if (age > 100)  std::cout << "maybe don't drive..";    } |

| **Compliant Code** |
| --- |
| Unlike the noncompliant code, the compliant code utilizes both the valid integer checks and the valid input checks. The portion of the code, while(!(std::cin >> age)), instructs the code to only accept integer values. If a character or a string were submitted, it would instruct the user to enter a number value. This prevents the program from accepting invalid data input. |
| int main()  {  signed int age;    std::cout << "enter user age :";  std::cin >> age;  //while statement sets a condition that must be met to execute the rest of program  //without the while statement, any user input that’s not an int would cause the program to run haywire  while (!(std::cin >> age)) {  std::cout << "Try a number ";  std::cin.clear();  std::cin.ignore(20, '\n');  }  if (age < 15)  std::cout << "Invalid Age";  else if (age > 16)  std::cout << "Valid age for driving";  else if (age > 100)  std::cout << "maybe don't drive..";    } |

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

|  |
| --- |
| **Principles(s):** Validate Input Data  The principle, validate input data, correlates to this coding standard. Validating input data refers to the proper testing and verification that user or program input adheres to the data input requirements. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Integer overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-int30 | Implemented |
| Coverity | 2017.07 | Integer-overflow | Implemented |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-cpp] | String correctness is essential to utilize to prevent potential vulnerabilities such as buffer overflow. The incorrect usage of string arguments could result in insufficient memory storage and eventually buffer overflows. If a string does not have sufficient storage it could result in a buffer overflow, which could then be used to overwrite the memory. If incorrect string arguments are used then it could lead to unexpected behavior. |

| **Noncompliant Code** |
| --- |
| This example portrays the usage of the wcsncpy() function to copy wide characters from nar\_stringA to nar\_stringB. The usage of the wcsncpy() function to copy wide characters to a narrow string would result into a buffer overflow, as the narrow strings would not have enough data memory to store the copied characters. |
| //narrow strings combined with wide string functions: wcsncpy()  void example(void) {  char nar\_stringA[] = “01234567890123456789”;  char nar\_stringB[] = “0000000000”;  wcsncpy(nar\_stringB, nar\_stringA, 10);  } |

| **Compliant Code** |
| --- |
| The compliant code blocks assign the correct string functions to the associated characters. For instance, the first example shows the usage of narrow character strings and a narrow string function (strncpy()). This establishes enough memory storage for the copied string which ensures that the buffer does not overflow. Similarly, the usage of a wide string function for a wide character string ensures that the data is not truncated. |
| void example(void) {  char nar\_stringA[] = “01234567890123456789”;  char nar\_stringB[] = “0000000000”;  strncpy(nar\_stringB, nar\_stringA, 10);  }  //if wide strings were used the compliant code would be:  void example(void) {  char wide\_stringA[] = L“01234567890123456789”;  char wide\_stringB[] = L“0000000000”;  wcsncpy(wide\_stringB, wide\_stringA, 10);  } |

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

|  |
| --- |
| **Principles(s):** validate input data, adopt a secure coding standard  The principle, validate input data, refers to the proper testing of user or software input. This validation and verification process would prevent the insert of invalid data that could result in overflows and other vulnerabilities. In relation to this standard, validating input data would help ensure that the allocated memory is not exceeded. Limits can be put into place that verifies that the buffer overflow cannot occur.  The adopt a secure coding standard involves the development and/or application of secure coding standards for the target programming language and platform. In terms of this coding standard, practicing this principle would ensure that the proper rules and guidelines are applied to the code. This would prevent potential exploits and compromise of the application through coding vulnerabilities such as that related to this coding standard. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Wide-narrow-string-string  Wide-narrow-string-cast-implicit | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR38 | Fully implemented |
| Clang | 3.9 | -Wincompatible-pointer-types | [Insert text.] |
| Coverity | 2017.07 | PW | Implemented |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-cpp] | SQL Injections is a security vulnerability that uses SQL queries to grant access to a database, therefore it is essential to prevent SQL injections. SQL injection attacks follow a format that always informs the program that the input is true. For instance, “ OR 1=1;”, specifies a blank space OR the statement 1 = 1, which is always true. This grants unauthorized access to any program or application that fails to protect against SQL injections. The best way to prevent SQL injections would be to utilize parameterized queries. |

| **Noncompliant Code** |
| --- |
| This sample code is used to get a StudentID from StudentRecords. The code uses an SQL query to select the student record of which matches the specified student ID. The statement, “SELECT \* FROM students WHERE StudentID = “ + studentID;, which does not prevent users from submitting invalid data. For example, the data input, “ OR 1=1;”, returns always true and thus provides unauthorized access to the system. |
| studentID = getString(“studentID”);  studentRecord = “SELECT \* FROM students WHERE studentID = “ + studentID;  //SQL Injection attack:  SELECT \* FROM students WHERE StudentID = 2 OR 1=1;  //this code example does not prevent an SQL Injection |

| **Compliant Code** |
| --- |
| To resolve or prevent SQL injections, parameterized queries are utilized. This eliminates the acceptance of invalid input, such as “ OR 1=1;”, as it enables the database or program to identify what should be accepted and what is not. It would only allow specific parameters to be accepted, making input like, “ OR 1=1;”, invalid. |
| StudentID = getString(“StudentID);  studentRecord = “SELECT \* FROM students WHERE studentID=@0”;  //use parameterized queries to prevent SQL injections  parameters.add(“@0”, studentID); |

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

|  |
| --- |
| **Principles(s):** Sanitize Data Sent to Other Systems, Validate Input Data  The principle, Sanitize Data Sent to Other Systems, applies to this coding standard as it would aid in preventing injection attacks such as SQL Injections. By properly sanitizing data sent to other systems, it prevents illegal statements or code from compromising the system.  By validating input data it would ensure that the user and/or program input adheres to the program’s specifications. This would prevent the occurrence of overflows as well as set conditions that are to be met. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| FindBugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Parasoft Jtest | 2021.1 | CERT.IDS00.TDSQL | Protect against SQL injection |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and Security errors |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-cpp] | Allocating enough memory for an object is essential in preventing buffer overflows and possible hacker manipulation. Incorrect or insufficient memory allocation can result in buffer overflows, truncation, and inadequate range checking. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code block, sizeof(int) is used to determine the memory allocation size. Because the variable does not have sufficient memory allocation, it can result in a buffer overflow as sizeof(long) is greater than sizeof(int). |
| void example(size\_t len){  long \*num;  if(len == 0 || len >SIZE\_MAX / sizeof(long)){  //checks for unsigned int overflow  // handle unsigned int overflow error  }  num = (long \*)malloc( len \* sizeof(int));  if (num == NULL){  /\* ….\*/  }  free(num);  } |

| **Compliant Code** |
| --- |
| Instead of using sizeof(int) to compute the memory size, the compliant code block utilizes the num variable. Sizeof(\*num) contains a larger memory allocation in comparison to sizeof(int). |
| void example(size\_t len){  long \*num;  //uses sizeof(\*num) to compute memory  if(len == 0 || len >SIZE\_MAX / sizeof(\*num)){  //checks for unsigned int overflow  // handle unsigned int overflow error  }  num = (long \*)malloc( len \* sizeof(\*num));  if (num == NULL){  /\* ….\*/  }  free(num);  } |

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

|  |
| --- |
| **Principles(s):** validate input data  The principle, validate input data, refers to the proper testing of user or software input. This validation and verification process would prevent the insert of invalid data that could result in overflows and other vulnerabilities. In relation to this standard, validating input data would help ensure that the allocated memory is not exceeded. Limits can be put into place that verifies that the buffer overflow cannot occur. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Malloc-size-insufficient | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM35 | [Insert text.] |
| Klocwork | 2021.1 | INCORRECT.ALLOC\_SIZE | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-JAV] | Assertions ensure that expected conditions in a program are met. If the assertion fails, it interrupts or prevents the program from running. For this reasoning, it is essential to use assertions correctly. If an assertion is used to validate method arguments, it could not only interrupt the program when it is not met, but it could also result in integer overflow and other errors. |

| **Noncompliant Code** |
| --- |
| This code block uses assertions to validate arguments of a public method. The assertions are used to validate that the variables x and y are not equal to the Integer.MIN\_VALUE. An assertion is also used to validate that absX is less than or equal to the Integer.MAX\_VALUE – absY. This is an illustration of how assertions should not be used. |
| Public static int getAbsAdd(int x, int y){  //assertions should not be used to validate method arguments.  assert x != Integer.MIN\_VALUE;  assert y != Integer.MIN\_VALUE;  int absX = Math.abs(x);  int absY = Math.abs(y);  assert (absX <= Integer.MAX\_VALUE – absY);  return absX + absY;  } |

| **Compliant Code** |
| --- |
| This code block demonstrates the correct usage of assertions. Instead of using an assertion to validate integer variables, if-else statements are used to set a condition that must be met to obtain a specific result. An assertion is used to assert that the id variable must not be equal to null. If the id is equal to null, the assertion would interrupt the program. |
| //correct usage of assertions  public class example {  public static void main(String[] args){  int id = 15;  if (id > 10){  System.out.print(“Valid ID”);  }else{  System.out.print(“Submit valid ID”);  }  assert id != NULL;  return id;  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques, Architect and design for security policies  Effective quality assurance techniques refer to techniques and concepts that revolve around maintaining a secure and functional program and/or application. Proper usage and implementation of assert statements would adhere to this coding principle.  The architect and design for security policies principle revolves around creating a software architecture and design that would accommodate and implement the necessary security measures. Incorporating proper practice of assert statements would allow the program to verify that functions and objects meet requirements and expected values. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully Implemented |
| CodeSonar | 6.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-jav] | An exception is a problem that occurs during program execution. It disrupts the program to have it abnormally terminate. The cause of an exception can revolve around various reasons including invalid data entry, a file that cannot be found, etc. It is important to address exceptions. There are instances where checked exceptions are suppressed or ignored. This is done through the usage of empty try-catch blocks. |

| **Noncompliant Code** |
| --- |
| This code block only prints out the exception’s stack trace and does not handle the exception. The exception’s stack trace would help to debug the program but at the same time exposes a vulnerability that can be exploited by attackers. |
| try {  /\* …\*/  }catch (IOException ioe){  Ioe.printStackTrace();  } |

| **Compliant Code** |
| --- |
| Unlike the noncompliant code block, this code block rethrows the exception(s). The try block generally contain the statement(s) that may cause the exception. The catch block handles the exception and then throws it. |
| try {  // statements that could cause the exception  }catch (Exception e){  //how to handle if exception thrown  System.out.println(“Error”);  //rethrows the exception  throw e;  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques, Architect and Design for Security Policies  The use effective quality assurance techniques refer to the techniques and concepts utilized in maintaining a secure and functional application. This principle applies to the coding standard as it describes the proper usage and effectiveness revolved around using exceptions. Proper exceptions usage would try a code portion that would throw an exception. The catch block would then catch the exception and handle the error as specified. This is essential in ensuring that the program is instructed on how to handle errors and exceptions.  Architect and Design for Security Policies describes the software architecture and design that would accommodate and implement the necessary security measures. This principle correlates to the coding standard as it describes how proper exception usage are beneficial in catching and handling errors that would result in vulnerabilities. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | JAVA.STRUCT.EXCP.EEH | Empty Exception Handler(Java) |
| Coverity | 7.5 | MISSING\_THROW | Implemented |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | [STD-008-jav] | Input validation is essential to ensure that the proper user/data input is being submitted into the program and/or application. By sanitizing data, it would ensure that only valid input is accepted, and it would prevent information leaks, denial of service attacks, and other potential issues. |

| **Noncompliant Code** |
| --- |
| The noncompliant code illustrates the absence of proper input validation and the use of untrusted data in format strings. As a result, the code block reveals information regarding a user’s credit card. The parameter, arg[0], is not validated and can reveal the date verified by the input. |
| //credit card example  class Example {  static Calendar c = new GregorianCalendar(1995, GregorianCalendar.MAY, 23);  public static void main(String[] args){    System.out.format(arg[0] + “didn’t match!  HINT: It was issued %1$terd of some month”, c);  }  } |

| **Compliant Code** |
| --- |
| The compliant code block demonstrates a more secure method of using untrusted user input. This method involves excluding the untrusted source from the format string. For instance, the above code block incorporated arg[0] within the string format:  (arg[0] + “didn’t match! HINT: it was issued %1$terd of some month”, c)  This can reveal the user’s credit card information to potential attackers. The compliant code block eliminates the usage of untrusted input in the format string. Rather, it incorporates the arg[0] parameter outside the format string.  (“didn’t match! HINT: it was issued %1$terd of some month”, arg[0], c); |
| //credit card example  class Example{  static Calendar c = new GregorianCalendar(1995, GregorianCalendar.May, 23);  public static void main(String[] args){    //unlike the noncompliant example, this example excludes the untrusted data from  //the format string.  System.out.format(“%s didn’t match! HINT: issued %1$terd of some month”,  arg[0], c);  }  } |

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

|  |
| --- |
| **Principles(s):** Validate Input Data, Sanitize data sent to other systems  Validate input data is a principle that refers to the proper testing of user or software input. It would prevent potentially malicious or malformed data from entering a system or database.  The principle, Sanitize data sent to other systems, describes the prevention of inserting invalid data into the program or application which could result in injection attacks, information leaks, and other vulnerabilities. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| Parasoft Jtest | 2021.1 | CERT.IDS06.VAFS | Ensure the correct number of arguments for varags methods with format strings |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Smart Pointer | [STD-009-cpp] | Smart pointers are a class type that overloaded the -> and \* operators to act as pointers. Compared to raw pointers, smart pointers are a more secure option as they provide augmented behavior such as garbage collection and null checks. It is not optimal to store an already owned pointer value in an unrelated smart pointer. In doing so could result in the destruction of a pointer value. The deletion or destruction of the pointer value could result in a vulnerability. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code block, two unrelated smart pointers are constructed with the same pointer value. By doing so, it destroyed both the p1 and p2 variables. This leads to a double-free vulnerability. |
| void Example(){  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| This compliant code block is the solution to the previous code block. In this code portion, the pointer objects are related to each other through a copy construction. This prevents the occurrence of a double-free vulnerability. If the variable, p2, is destroyed, the shared pointer value use count is decremented, however, it maintains a nonzero value. In similar circumstance, the destruction of the p1 variable results in the use count’s decrement to zero and the destruction of the managed pointer. |
| void Example(){  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

|  |
| --- |
| **Principles(s):** Architect and design for security policies  The architect and design for security policies describes the creation of software architecture and design that would accommodate and implement security measures. By using smart pointers, it creates more security in the program. It would check for nulls and act as garbage collectors. The destruction of pointer objects such as p1 and/or p2 would result in the destruction of the managed pointer. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Dangling\_pointer\_use | [Insert text.] |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM56 | [Insert text.] |
| Polyspace Bug Finder | R2021a | CERTC++:MEM56-cpp | Checks for use of already-owned pointers (rule fully covered) |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Hard Coding | [STD-010-jav] | Hard coding is the practice of coding data directly into a program’s source code. By hard coding sensitive data into a program’s source code, it can expose that information to hackers and other unauthorized individuals. This could lead to remote exploitation and system compromise. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates the usage of hard coding user login information (username and password). By hard coding these variables, it allows hackers and other unauthorized individuals to view user login information. If an attacker were to view the user login information, they could access the system via remote exploitation. |
| //example of user login info hard coded  public final connect getConnection() throws SQLexception{  return DriverManager.getConnection(  “mysql://localhostname/databaseName”, “username”, “password”);  } |

| **Compliant Code** |
| --- |
| The compliant code example depicts the proper method of coding user login information. Instead of hard coding the sensitive information directly into the program’s source code, string variables are used to hold the information. This would prevent user credentials from being exposed to unauthorized individuals. |
| //example of proper user login  public final connect getConnection() throws SQLexception{  //stores user credentials in string arrays by doing so it prevents attackers //from viewing and exploiting a user’s login information  String user;  String pass;  return DriverManager.getConnection(“mysql://localhostname/databaseName”,  user, pass);  } |

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

|  |
| --- |
| **Principles(s):** Adopt a secure coding standard  The adopt a secure coding standard principle involves the development and application of secure coding standard for the target programming language and platform that would prevent potential exploits and compromise of the system through coding vulnerabilities. By hard coding user credentials and other information, it allows hackers and other unauthorized individuals to view this information. By using string variables to hold sensitive information it would prevent unauthorized individuals from exploiting that information. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | JAVA.HARDCODED.PASSWD  JAVA.MISC.SD.EXT | Hardcoded Password(Java)  Sensitive Data Written to External Storage (Java) |
| Fortify | 1.0 | Password\_management  Password\_manadament\_hardcoded\_password | Partially Implemented |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Defense-in-Depth Illustration

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



# Project One

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

## Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

## Risk Assessment

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

## Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

## Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation in terms of cyber security is the execution of security actions to detect, investigate and remediate cyberthreats and vulnerabilities with and/or without any interference. This would prevent bugs, undefined behavior, avoid security breaches and attacks, and enhance maintainability. In regard to the DevSecOps diagram, potential threats, vulnerabilities, and assets are identified. This would indicate what is at most risk during a security or data breach. Once these threats are properly identified, IDE security plug-ins are installed. This would scan the program for threats, errors, bugs, and vulnerabilities that have been overlooked. The principles and coding standards associated with the design section of the DevSecOps diagram would provide coding specifications and guidelines to ensure securely structured code. Within the Build portion of the diagram, coding standards such as validate data input would be placed here. This area defines a secure program build. For instance, data validation would promote the secure build of a program and application. In the Verify and Test portion of the diagram, specifications regarding assertions and exceptions can be placed here. This would verify that the program reacts to issues as expected and/or generates expected outputs.

## 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-001-CPP | High | Likely | High | P9 | 2 |
| STD-002-CPP | High | Likely | High | P9 | 2 |
| STD-003-CPP | High | Likely | Low | P27 | 1 |
| STD-004-CPP | High | Probable | Medium | P12 | 1 |
| STD-005-CPP | High | Probable | High | P6 | 2 |
| STD-006-JAV | Medium | Probable | Medium | P8 | 2 |
| STD-007-JAV | Low | Probable | Medium | P4 | 3 |
| STD-008-JAV | Medium | Unlikely | Medium | P4 | 3 |
| STD-009-CPP | High | Likely | Medium | P18 | 1 |
| STD-010-JAV | High | Probable | Medium | P12 | 1 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## 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***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. 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.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest refers to the prevention of unauthorized access of unencrypted data by ensuring the data is encrypted when on the disk. It provides protection for data that is stored. A key would be required to encrypt and decrypt the data. This is useful because if an attacker were to acquire the hard drive with encrypted data but did not acquire the encryption keys, it would take more time and effort to break through the encryption to read the data. |
| Encryption at flight | Encryption at flight describes the encryption of data while being transmitted. This provides protection for the data as it is transmitted to its destination. Similar to encryption in rest, encryption at flight uses an encryption key to encrypt and decrypt the information. This would ensure that if the data is hijacked or copied during transmission, the encryption would pose a challenge to the unauthorized parties. Without the encryption key, the encrypted data would require large amounts of time, effort, and resources to break through. This would help deter any malicious party from compromising and exploiting the data. |
| Encryption in use | Encryption in use refers to the application of encryption regardless of the data’s state (i.e., at rest, in flight). This ensures that data is never left unsecured. However, since, the files are always encrypted despite their state, it is always protected from unauthorized individuals. Like the previous encryption types, encryption keys are utilized to encrypt and decrypt the data. If a system breach were to occur, the data would be secured as it would be in an encrypted state. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user to be legitimate. This can be applied to user logins to ensure the identity of the individual. The authentication also ties in with the authorization portion of the triple A framework. By verifying a user, it also helps determine the level of access granted. |
| Authorization | Authorization determines the user level of access based on their authentication. This determines what a user can do, and access based on their level. Some users may be permitted to make changes to a database if their authorization permits functions. Other users may have read-only access to a database or other files. This all depends on the level of authorization each user has been given. |
| Accounting | Accounting documents the activities that have taken place within a system or database. For instance, if changes are made to a database, such as a new entry is added, accounting would record this change. It would note which user made the change, how long did it take the user to complete the task, did the user go somewhere else other than that database, and so on. If a user were to delete important files, the accounting framework would provide information on who deleted the files and the time the files were deleted. It would also note the files accessed by the users. |

**\***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 | 08/07/2021 | Edited Template | Danielle Parham | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |