**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 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_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 | A programmer should always ensure that they validate their input data before passing it along to other areas of the codebase. This includes ensuring that provided input will not lead to a buffer overflow by verifying that the input is within an acceptable range. It also includes making sure that provided input will not lead to a mathematical error such as attempting to divide by 0. |
| 1. Heed Compiler Warnings | Compiler warnings are warnings that are generated during compilation of code. While these warnings are not severe enough to prevent the code from compiling and being executed, they can indicate potential problems during runtime. For example, compiler warnings may generate if it detects loss of numerical precision, such as a floating-point value 3.1415 being assigned to an integer variable, which would result in said integer being assigned with the value of 3. Other compiler warnings may include using a deprecated function or using a non-portable function or macro. |
| 1. Architect and Design for Security Policies | Applications should be designed from the ground up with security in mind. This can include things such as identifying when and where encryption is necessary or building secure authentication systems to prevent users from accessing unauthorized data. This might also include data retention and backup policies. |
| 1. Keep It Simple | This principle related to how a programmer should prioritize simplicity in most cases over cleverness or “hacky” solutions. In some cases, this may result in more lines of code, such as in the case of a complicated mathematical operation being broken down into a series of smaller, easily explainable steps. This principle also relates to favoring standard, established design patterns in code that can be more easily recognized by other team members working on the project. |
| 1. Default Deny | Default Deny is the process of denying any process or person from performing any activity on the system by default, and instead requiring that each “user” specifically be allowed access on an individual basis. This might require that specific details above a device, such as the IP address or mac address, be specifically whitelisted in a firewall or other program before the device is allowed to connect or access anything. |
| 1. Adhere to the Principle of Least Privilege | This principle specifies that a user only be given as little permission or access to the system as necessary to complete their required tasks. Following this principle can prevent hackers from accessing the entirety of a system should they manage to get ahold of credentials of people that have access to the system. |
| 1. Sanitize Data Sent to Other Systems | Data sanitation is the act of stripping personal or sensitive data, particularly before it is stored or transferred. For example, a user may interact with a system that stores all of their information (including personally identifiable information), but that sensitive information may need to be removed before transferring it to a third-party processor. |
| 1. Practice Defense in Depth | Defense in Depth refers to determining the appropriate level of security required in a given application or system and ensuring that it meets those requirements. This will vary from codebase to codebase, but the general idea is that data that need be secured should be secured, such as personalized information or financial data. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance is the process of verifying that a code or project meets predetermined quality standards, generally by using a repeatable and measurable process. This can include things such as unit testing, where each unit of a code is evaluated against a number of test cases in order to verify that it functions as documented. This can also include the use of code analysis programs, such as a memory analyzer that ensures all dynamically declared memory is properly freed. |
| 1. Adopt a Secure Coding Standard | Standards are guidelines by which all code is typically expected to follow in a either an individual or team development environment. Adopting a standard can help ensure that all developers associated with a given project are aware of the security needs of the project so that the can code with them in mind. |

### 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-DCL51-CPP] | Ensure that the value for a variable is of the appropriate type. |

| **Noncompliant Code** |
| --- |
| In this example, the integer variable is assigned with a floating-point number. This will cause a compile-time warning, and the value of numGuests will actually be assigned with the truncated value of 3. If this assignment was made using user input, it would silently be truncated and could lead to an erroneous calculation. |
| int main()  {  int numGuests;  numGuests = 3.53;  std::cout << numGuests;  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version, the variable numGuests is declared as a double so that it can properly store the floating-point value that is assigned to it. |
| int main()  {  double numGuests;  numGuests = 3.53;  std::cout << numGuests;  return 0;  } |

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

| **Principles(s):**  Validate Input Data – If supplied data is expected to be of a certain type, it should be checked for validity before being passed to a system or operation that will already expect it to be correct.  Heed Compiler Warnings – Compilers may issue a warning if it detects truncation, or a cast from one data type to another. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | MEDIUM | LOW | LOW | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files. |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase. This can be used to ensure value assignment is working as expected. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-VAR01-CPP] | Make sure a variable is never assigned a value that is outside the allowed range. |

| **Noncompliant Code** |
| --- |
| According to the standard, this unsigned char is only guaranteed to be able to hold a maximum value of 255. As a result, this code is likely to result in overflow, thus printing out an unintended value. |
| int main()  {  unsigned char numGuests = 900;  std::cout << (int)numGuests;  } |

| **Compliant Code** |
| --- |
| It is important to always pick an appropriate data type that is capable of representing the maximum value that could be associated with a variable as per the logic of the program. In this case, numGuests is now declared as a short, which is more than capable of storing the value 900. |
| int main()  {  unsigned short numGuests = 900;  std::cout << numGuests;  } |

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

| **Principles(s):**  Validate Input Data – Input data should be checked to ensure it is in range before it is assigned to a variable type. Otherwise, overflow or underflow may occur.  Heed Compiler Warnings – A compiler may be able to warn against certain types of assignments that might exceed the range of a variable type. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | MEDIUM | LOW | MEDIUM | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files. |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase. This can be used to ensure value assignment is working as expected. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-STR50-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator. If attempting to write more characters to a string than can fit, a buffer overflow will occur. This can lead to hackers being able to run arbitrary code. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because the user can enter in a string of any length, and it will be read into the myString buffer. This will cause an overflow if the user provides more than nine characters. |
| #include <iostream>  #include <cstdio>  int main()  {  char myString[10];  gets\_s(myString);  std::cout << myString;  } |

| **Compliant Code** |
| --- |
| In this example, the fgets function will only store a total of 10 characters into myString. The first nine characters will come from the user, and anything in excess will not be stored. Afterwards, a null terminator will be inserted as the final character. This will prevent overflow. |
| #include <iostream>  #include <cstdio>  int main()  {  char myString[10];  fgets(myString, 10, stdin);  std::cout << myString;  } |

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

| **Principles(s):**  Validate Input Data – If a user is supplying the data for a string, ensure that it fits within the length that has been allocated for it before assignment.  Heed Compiler Warnings – Static analyzer tools may be able to detect a logic error that could lead to writing past the boundary of a string or character array.  Adopt a Secure Coding Standard – Buffer overflows can lead to users being able to run third-party code within a system. This is a major security risk. Careful attention should be taken to ensure that operations in the codebase will not attempt to write outside the bounds of a string or character array. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | HIGH | HIGH | HIGH | 8 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-SQL01-CPP] | Always sanitize inputs that will be used for SQL statements. |

| **Noncompliant Code** |
| --- |
| In this example, the user enters in a value for the name parameter that gets directly inserted into the sql statement. This is bad practice because the user could easily inject additional sql keywords, allowing them to get access to potentially unauthorized data. |
| #include <iostream>  #include <string>  int main()  {  std::string name;  getline(std::cin, name);  std::string sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='" + name + "'";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  } |

| **Compliant Code** |
| --- |
| In this example, the code specifically prevents the use of terms OR, AND, or \* in the query. If any of this terms are detected, the query will fail to execute. |
| #include <iostream>  #include <cstdio>  #include <string>  int main()  {  std::string name;  getline(std::cin, name);  const size\_t FORBIDDEN\_TERMS\_COUNT = 3;  std::string forbiddenTerms[FORBIDDEN\_TERMS\_COUNT] = { "OR", "AND", "\*" };  for (size\_t i = 0; i < FORBIDDEN\_TERMS\_COUNT; i++) {  if (name.find(forbiddenTerms[i]) != std::string::npos) {  std::cout << "Potential sql injection attempt";  return 1;  }  }  std::string sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='" + name + "'";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  } |

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

| **Principles(s):**  Validate Input Data – If user input is fed directly into a SQL command, ensure that they are not passing any SQL keywords that would manipulate the intention of the statement, such as “AND”, “OR”, or “DROP TABLE”  Architect and Design for Security Policies – Designs for a system should take into account how to protect against SQL injection. This could include using an external SQL library that handles the security side of SQL statements, or manually coding functions that are capable of detecting SQL injection attempts.  Use Effective Quality Assurance Techniques – Use unit tests to verify what happens if a user attempts to use invalid keywords for input that leads to execution of a SQL statement. Input containing SQL keywords should be rejected.  Adopt a Secure Coding Standard – The entire team should be aware of the seriousness of SQL injection attempts and mitigation techniques. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | MEDIUM | HIGH | HIGH | 10 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files |
| Sqlmap | 1.8 | https://sqlmap.org/ | Tool capable of running SQL penetration attempts. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-MEM50-CPP] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| In this example, the variable s is accessed after the dynamically allocated memory is related. This is undefined behavior in C++ and should be avoided. |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| Here, the deletion of the pointer occurs after it is used. |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques – Unit tests can be used to ensure that unexpected operations are not occurring in cases where dynamically allocated memory is used. Manual code reviews can also verify that memory is not used again once it has been freed.  Adopt a Secure Coding Standard – The team should be aware of the risks of attempting to access memory once it has been freed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MEDIUM | MEDIUM | MEDIUM | 6 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Memcheck | Unknown | https://valgrind.org/docs/manual/mc-manual.html | Memcheck is a memory error detector that comes with Valgrind that can be used to determine memory leaks or other related issues |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase, specifically related to memory |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-MSC60-CPP] | Do not use assertions to verify the absence of runtime errors |

| **Noncompliant Code** |
| --- |
| The following code uses the assert function to verify that that the value assigned to input is 1. However, if the assert function does not yield true, then the program will terminate with an error message. This does not allow the code an opportunity to fix the error to avoid a crash. |
| #include <iostream>  #include <cassert>  int main()  {  unsigned int dividend = 12;  unsigned int divisor = 0;  double quotient;  // Check that we aren’t dividing by 0  assert(divisor != 0);  quotient = (double)dividend / (double)divisor;  std::cout << "Quotient is " << quotient;  return 0;  } |

| **Compliant Code** |
| --- |
| Instead, if statements and loops can be used to correct input before it leads to a runtime error. |
| #include <iostream>  #include <cassert>  int main()  {  unsigned int dividend = 12;  unsigned int divisor = 0;  double quotient;  while (divisor == 0) {  std::cout << "Cannot divide by 0. Try again: ";  std::cin >> divisor;  }  quotient = (double)dividend / (double)divisor;  std::cout << "Quotient is " << quotient;  return 0;  } |

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

| **Principles(s):**  Keep It Simple – If statements or try/catch blocks are more appropriate ways to check for certain conditions and perform specific actions for each case.  Use Effective Quality Assurance Techniques – A static code analyzer may be able to detect when assert statements are used in code intended for release builds. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | LOW | LOW | LOW | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files. This can specifically be used to identify any assert statements |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-ERR51-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| If an error is thrown by a program without being caught, it can result in a runtime exception. Depending on the severity, this may cause a program to crash with no warning. This can be especially jarring to a user. It is also possible that a user may intentionally manipulate input variables with the goal of crashing a program, which may lead to other security issues. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| The following code ensures that exceptions on the potentially throwing function will be caught and handled. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  try {  f();  }  catch (...) {  // Handle error  }  } |

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

| **Principles(s):**  Validate Input Data – Input collected from the user should be validated before it is used. In certain situations, this input cannot be properly checked before it is “used”, so a try/catch block should be used. If the operation results in an error, it can be caught and handled without crashing the system.  Use Effective Quality Assurance Techniques – Unit testing can demonstrate that erroneous input causes a unit of code to throw an error.  Adopt a Secure Coding Standard – The team should be informed that they are expected to use try/catch blocks around all code that has the potential to throw an error. User-defined functions that can throw an error should be labeled as such in the code, if applicable. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | HIGH | LOW | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files. |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase. This can be used to verify exceptions are being thrown and handled as appropriate |
| Vector | 2.1 | https://pclintplus.com/ | C++ code linter that could be capable of detecting unhandled exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-ERR50-CPP] | Use exceptions only for exceptional conditions |

| **Noncompliant Code** |
| --- |
| This code relies on an error being thrown after every element in the array is processed. This would occur because the code attempts to index the array out of bounds. This is predictable and expected behavior, and therefore it should not be caught with an exception. The code should use traditional logic to determine when the entire array has been processed. |
| #include <string>  public String processSingleString(String string) {  // ...  return string;  }  public String processStrings(String[] strings) {  String result = "";  int i = 0;  try {  while (true) {  result = result.concat(processSingleString(strings[i]));  i++;  }  }  catch (ArrayIndexOutOfBoundsException e) {  // Ignore, we're done  }  return result;  } |

| **Compliant Code** |
| --- |
| In this code, the array is properly indexed based on its length. The strings are concatenated, and the result is returned. The loop is properly bound, so it will not go out of bounds. |
| #include <string>  public String processStrings(String[] strings) {  String result = "";  for (int i = 0; i < strings.length; i++) {  result = result.concat(processSingleString(strings[i]));  }  return result;  } |

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

| **Principles(s):**  Keep It Simple – Do not use exceptions in situations where standard logical checks will work. For example, if input is expected to be between 1 and 5, but the user enters 7, an exception should not be manually thrown and caught. Instead, use a simple if statement check and loop to try to collect the input again. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | LOW | LOW | LOW | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Vector | 2.1 | https://pclintplus.com/ | C++ code liner that could be capable of detecting forced exceptions |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase. This can be used to verify that exceptions are not being thrown |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-NUM50-CPP] | Convert integers to floating point for floating-point operations |

| **Noncompliant Code** |
| --- |
| In this code, an integer is being divided by another integer. The true value of this operation would be 2.4. However, since both operands are integers, integer division will be performed. As a result, the decimal portion of the value will be discarded, yielding a quotient of 2. |
| #include <iostream>  int main()  {  unsigned int dividend = 12;  unsigned int divisor = 5;  double quotient = dividend / divisor;  // Real value should be 2.4  // Truncated value will be 2  std::cout << "Quotient is " << quotient;  return 0;  } |

| **Compliant Code** |
| --- |
| Here, the operands are properly cast to doubles before division occurs. This allows floating-point division to occur, which will preserve the decimal portion of the quotient, yielding 2.4 as expected. |
| #include <iostream>  int main()  {  unsigned int dividend = 12;  unsigned int divisor = 5;  double quotient = (double)dividend / (double)divisor;  std::cout << "Quotient is " << quotient;  return 0;  } |

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

| **Principles(s):**  Heed Compiler Warnings – A compiler may be able to detect when an operation involving operands of different types may result in the loss of precision. This may be an indicator that an integer should be explicitly promoted to a floating point before performing a mathematical operation.  Use Effective Quality Assurance Techniques – Unit tests can verify that inputs of different types still lead to the expected result. A failing test may indicate that an explicit cast or promotion is required if not already present. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | LOW | LOW | LOW | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files. This can detect type conversion errors |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-EXP50-CPP] | Do not confuse abstract object equality with reference equality |

| **Noncompliant Code** |
| --- |
| Objects in code cannot be compared using the == operator, as it will actually make a comparison on the memory addresses of the objects. This code actually checks if str1 and str2 are the same object, which will evaluate to false. |
| public class StringComparison {  public static void main(String[] args) {  String str1 = new String("one");  String str2 = new String("one");  System.out.println(str1 == str2); // Prints "false"  }  } |

| **Compliant Code** |
| --- |
| In this code, the strings are correctly compared using the equals function member of the string class. This returns true if the contents of both strings are the same, which is the desired behavior. |
| public class StringComparison {  public static void main(String[] args) {  String str1 = new String("one");  String str2 = new String("one");  System.out.println(str1.equals(str2)); // Prints "true"  }  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques – Unit checks can verify whether object equality works as expected.  Adopt a Secure Coding Standard – The team should be aware that the result of the == operator can depend on the context of the operands being compared. In some cases, a value check is initiated. In other cases, a reference check is performed instead. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDUM | MEDIUM | LOW | HIGH | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | v1.14.0 | <https://github.com/google/googletest> | Automated test suite for C++ that allows automatically running tests on the codebase. This can be used to check equalities |
| Cppcheck | V2.14 | <https://cppcheck.sourceforge.io/> | Static analysis tool for detecting various potential issues in C++ code files. This could point out potential oversights in development |

### 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.

There are currently multiple phases in the DeOps pipeline for Green Pace. In order to accommodate the new automated tools, I propose the following changes:

* + - 1. Introduce Cppcheck and Vector into the “build “phase. Once source has been built, it can be run through these static analyzers and linters to detect issues such as loss of precision, incorrect data typing, incorrect exception handling, and more.
      2. Once the program has been built, Memcheck and Valgrind can be used to check for memory leaks.
      3. Introduce Google Test into the “Verify and test” phase. This will allow for automated unit testing, which will verify that modules are working as defined by their specification.
      4. Introduce Sqlmap into the “Verify and test” phase. This can be used check for any SQL injection issues.

### 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-DCL51-CPP]** | LOW | MEDIUM | LOW | LOW | 2 |
| **[STD-VAR01-CPP]** | LOW | MEDIUM | LOW | MEDIUM | 3 |
| **[STD-STR50-CPP]** | HIGH | HIGH | HIGH | HIGH | 8 |
| **[STD-SQL01-CPP]** | HIGH | MEDIUM | HIGH | HIGH | 10 |
| **[STD-MEM50-CPP]** | MEDIUM | MEDIUM | MEDIUM | MEDIUM | 6 |
| **[STD-MSC60-CPP]** | LOW | LOW | LOW | LOW | 1 |
| **[STD-ERR51-CPP]** | MEDIUM | HIGH | LOW | HIGH | 5 |
| **[STD-ERR50-CPP]** | LOW | LOW | LOW | LOW | 2 |
| **[STD-NUM50-CPP]** | LOW | LOW | LOW | LOW | 2 |
| **[STD-EXP50-CPP]** | MEDUM | MEDIUM | LOW | HIGH | 4 |

### 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 at rest | Encryption at rest refers to data not currently moving or in use, such as data stored in a database. This may include private user information. This data should be encrypted, meaning that it is “transformed” from their original values into new values that cannot be easily decoded. The benefit of this is that, should a security breach occur, hackers will not easily be able to determine what the original values were. In this way, customer information can remain secure. |
| Encryption in flight | Encryption in flight refers to data actively being transmitted from the client to the server or from server to client. An example of where this applies is the case where a user is submitting a form to a website, or when a server responds with information to populate the fields on a web page while a user is modifying their personal details. This is important because without encrypting this data, a man-in-the-middle attack can occur, and personal user information can be exposed. |
| Encryption in use | Encryption in use refers to data actively loaded in a program’s main memory. The purpose of it is to maintain protection associated with data, even while in use. An example of this might include user passwords. In a best-case scenario, user passwords are hashed or encrypted before being stored in a database. While authenticating a user, the password needs to be loaded into main memory. The password would be loaded in while still encrypted, and then the user’s input would also be hashed and then compared against the encrypted value to determine if they match. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is the process of ensuring user credentials are properly validated before they can access any part of a system. There are usually specific user id or password requirements to increase security. For example, user ids may not be allowed to be composed of the user’s email address, and the password may need to be of a minimum length and contain special characters. This can include addition of new users. |
| Authorization | This is the process of operating under the principle of least privilege, which states that users of a system should only have as much access as necessary to perform their duties. Users can be grouped into roles which dictate which system resources they can access. This is helpful, because it means that even if users fall victim to a phishing or social engineering attempt, the bad actor still may not have access to the entirety of a system. This can include a user’s level of access. |
| Accounting | This is the process of keeping proper records for system access or activities completed within the system. These records may include details such as date, time, user, action taken, previous values, and new values. Attempts made by users to access resources they are not allowed to may also be kept. The overall purpose is to keep visibility on the system and be able to identify where changes occurred if need be. This can also include user logins, changes to any databases, or files accessed by 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 | 05/21/2024 | 10 core security principles were explained, and 10 coding standards were defined and partially explained |  |  |
| 3.0 | 06/09/2024 | Risk assessments were performed for coding standards |  |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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