**Green Pace Developer: Security Policy Guide**



# 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 | Ensure that corruptible data has been validated to be what is expected. Examples of corruptible data include but are not limited to databases, user input, and sensory input. As a general rule of thumb, if the data is coming from outside of the program then validate it before using it. |
| 1. Heed Compiler Warnings | Compiler warnings tend to be showing up for a reason. Especially with implicit conversions of datatypes, ensure that all compiler warnings are heeded with the utmost scrutiny. |
| 1. Architect and Design for Security Policies | Ensure that when designing and creating software for Green Pace you are adhering to all of the security policies laid out in this document. Furthermore, if you see examples of problems within the current codebase then please reach out to senior staff to determine the fix for these problems. |
| 1. Keep It Simple | Ensure that functions and classes are as single purposed as possible. Having very large functions and classes that have too much responsibility creates a more difficult environment to troubleshoot and debug. Simplicity is key, always make sure there isn’t already a solution to a problem built into C/C++ or the libraries we use before you recreate a solution. |
| 1. Default Deny | For authentication purposes, default to a denial of entry. This means that if a user cannot remember their password then they need to contact technical support to regain access to their account. There should be no easy way to bypass the password requirement for gaining access to any resources built into our applications. |
| 1. Adhere to the Principle of Least Privilege | Users should only ever have the amount of privilege that they truly need. This means that a user using our application to purchase things should never have access to changing other user account info. Furthermore, within our company itself, an employee who is a software engineer should only have access to the tools to perform that job. |
| 1. Sanitize Data Sent to Other Systems | Data sent from our system to other systems should be sanitized of any unneeded artifacts. This may mean compressing the data to be as small as possible as well as sending only what is needed for the other system. If a system requires user payment info, that system should not be seeing the users username, password, purchase history, etc. |
| 1. Practice Defense in Depth | Defense in depth is a strategy of having multiple layers of defense in regard to cyber security. This document acts as one layer of security where we aim to dictate employee behavior to align with security principles. Other layers would include an intranet where all of the computers in the company are on a private network that is protected by a firewall. Another layer may be for the company to always use a VPN to access the internet and to limit which websites employees are allowed to visit. |
| 1. Use Effective Quality Assurance Techniques | Always ensure the quality of code created. This means creating unit tests to test the individual chunks of code as they are created but furthermore this also means utilizing integration testing as the project nears completion. Additionally, continued support of any launched applications to quickly and effectively resolve any bugs or security violations is paramount to maintaining a high level of quality. |
| 1. Adopt a Secure Coding Standard | Having a consistent guideline for how to create code is incredibly important in ensuring that the code is easily readable by anyone within the company but also that the code is following secure standards. This means following the standards provided in the next section when creating new code for our products. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Data types should always be used to hold the highest value expected for a given variable. As an example, a char may be suitable to hold the current month of the year however this would not be suitable to hold the value of current users on a website. Overflow and underflow attacks can be easily performed if the data types are incorrectly used, causing an increase in vulnerabilities. Furthermore, ensure that data type conversion occurs from a smaller data type to a large data type. The other way around could lead to truncation of data leading to unexpected outcomes of overflow / underflow vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The code below shows the two issues described in the rationale above. The current\_users variable is being assigned a value larger than can be held by a char data type. This will lead to and overflow vulnerability, causing unexpected behavior from the program. Next, the more\_users variable is being assigned a large value to a data type of long long which can store this value, however the value is then being implicitly converted into a char on the next line of code. This again will cause truncation of data or undefined behavior in some implementations of the compiler. |
| char current\_users = 100000;  long long more\_users = 1000000;  char more\_users\_char = more\_users; |

| **Compliant Code** |
| --- |
| The code below is compliant as the char is only used to store the number of the month which will only ever be expected to be 1 – 12. This is a small enough number to ensure that the char will never overflow / underflow and allow this data type to be used safely. The int some\_users is initialized with a suitable data value and is then statically cast into a data type that is larger than the int, ensuring that no truncation occurs and the data remains safe. |
| char months = 12;  int some\_users = 1000;  long long more\_users = static\_cast<long long>(some\_users); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques is the principle that closely matches this standard. In this case – with ensuring high quality code that aligns with needed expectations of the data types, this problem will no longer be an issue. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Somewhat Likely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |
| CPPcheck | 2.14 | https://cppcheck.sourceforge.io |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensure that all data values for variables are initialized to a suitable value. As a safety precaution variable declaration should also include the initialized value, even if that value will be switched to something more default. This avoids reading memory that has not been initialized which has undefined behavior. |

| **Noncompliant Code** |
| --- |
| The code below shows that the integer my\_num is being declared without being initialized. This means that the value at my\_num is unknown and likely garbage data, leading to undefined behavior when reading this data for the console out statement. |
| int my\_num;  std::cout << my\_num << std::endl; |

| **Compliant Code** |
| --- |
| The code has now been updated to show the proper way of initializing all variables with a default value. This then ensures that the behavior is well defined when printing to the console on the next line. |
| int my\_num = 0;  std::cout << my\_num << std::endl; |

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

| **Principles(s):** Adopting a secure coding standard will help to eliminate problems such as this one. This problem may not be as extreme as others, however using a secure coding standard in this case will ensure that this is never going to be an issue. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | Medium | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |
| CPPcheck | 2.14 | https://cppcheck.sourceforge.io |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Never use C-Style strings and if necessary to use them then ensure they are properly buffered with space for a null character. C-Style strings are a dangerous data structure because the buffer for a string can be easily overlooked, especially with the null character. Reading beyond the bounds of a string leads to undefined behavior or these bounds could be used to perform buffer overflow attacks, causing malicious code to be executed. If possible, always use a secure string library such as the one built into C++ std::string. |

| **Noncompliant Code** |
| --- |
| This codes presents the use of a C-Style string, creating a buffer sized at 255. The vulnerability with this code lies in the fact that the buffer has a finite size and the input has direct access to this buffer. If the user types anything greater in length than 254 characters (remember the null character at the end) then this string will overflow, leaving this program open to a buffer overflow attack. |
| char input\_string[255];  std::cout << “Enter your input: “;  std::cin >> input\_string; |

| **Compliant Code** |
| --- |
| The updates code now uses the built in std::string class from C++ which ensures that the input is safe from an overflow. This is not only simpler than managing C-Style strings but also has tested and verifiable safety when being used in place of C-Style strings. |
| std::string my\_string;  std::cout << “Enter your input: “;  std::cin >> my\_string; |

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

| **Principles(s):** This is an example of several Principles such as Validate Input Data, Practice Defense in Depth and Adopt a Secure Coding Standard. In this case, C-Style strings can create a lot of issues and require validating input thoroughly every time they are used. Simply using std::string will eliminate the need to check for buffer overflows and this ensures an additional layer of defense. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |
| CPPcheck | 2.14 | https://cppcheck.sourceforge.io |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Ensure that all interfaces with an SQL database have the input thoroughly sanitized before using the SQL queries. SQL queries can easily be injected with malicious inputs if there is a lack of attention to how the input is being used in the query. This can lead to the attacker accessing more data than they should have access to, creating a lack of user trust and increased vulnerabilities. |

| **Noncompliant Code** |
| --- |
| As shown in the code below, the SQL query is being concatenated directly with user input. This allows the user to not only type their name, but they could also type a command such as “FAKENAME OR 1=1” which would bypass the query and return everything in the table. |
| std::string sql = “SELECT \* FROM USERS WHERE NAME=”;  std::string username;  std::cout << “Enter your username: “;  std::cin >> username;  sql += username;  sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message); |

| **Compliant Code** |
| --- |
| The updated code would use a built-in function to sanitize the input from the user before adding it to the query. This could be left to interpretation but some examples of how this could achieve this could be to tokenize each part of what the user has input to see if it matches expected values. This function could also be used to check for SQL injection patterns and reject the input if the pattern exists. This ensures that there is a checking layer in place to make sure that the SQL being provided is what will always be expected. |
| std::string sql = “SELECT \* FROM USERS WHERE NAME=”;  std::string username;  std::cout << “Enter your username: “;  std::cin >> username;  sql += sanitize\_sql\_input(username);  sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message); |

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

| **Principles(s):** This aligns very well with the principle of Validating Input Data. Anything coming from an outside source should be treated as untrustworthy data as this could lead to injection attacks or overflow attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cycode |  | <https://cycode.com> |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Avoid using normal pointers as much as possible and instead utilize smart pointers. Utilizing pointers and using the new and delete operators are prone to having memory leaks and vulnerabilities. If using pointers, always initialize the pointer to a value such as nullptr and when using the new operator ensure there is an associated delete operator. |

| **Noncompliant Code** |
| --- |
| In the code below we have a pointer that isn’t initialized to any value, this causes undefined behavior when reading this value for the standard output. Furthermore, we also have a pointer pointing to a new chunk of memory with the new operator but no deletion of this memory. This leads to a memory leak which could be the source of a denial of service attack. |
| int \* my\_ptr;  std::cout << \*my\_ptr << std::endl;  int \* my\_other\_ptr = new int; |

| **Compliant Code** |
| --- |
| The code has been updated, showing my\_ptr has been initialized before printing to the console and then properly being deleted using the delete operator. |
| int \* my\_ptr = new int(3);  std::cout << \*my\_ptr << std::endl;  delete my\_ptr; |

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

| **Principles(s):** This is an example of Keeping it Simple and Adopting Secure Coding Standards. Pointers can be useful tools; however they are also highly prone to issues. Smart pointers allow the use-case of pointers without as many potential issues as what the pointers deal with. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Somewhat Likely | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assert expected values where possible to reduce the likelihood of input vulnerabilities. This is a layer of defense that prevents vulnerabilities from being found by ensuring that data coming in is as expected before it is used. |

| **Noncompliant Code** |
| --- |
| This code is expecting that the value of num is going to be 0 – 11 for the 12 months of the year, however the data type provided is a char – meaning that the value could be 0 – 256. This would cause the program to read into the array at an index unintended for the design of the program. |
| std::string convert\_char\_to\_month(char num){  std::string months[] = {“January”, “February”, “March”, “April”…}  return months[num];  } |

| **Compliant Code** |
| --- |
| With the updated code we are now using the assert function to determine if the value provided is within the expected range for the months. This will now ensure that the program will not have undefined behavior but instead halt if the value does not meet the expected criteria. |
| std::string convert\_char\_to\_month(char num){  std::string months[] = {“January”, “February”, “March”, “April”…}  assert(num < 12);  return months[num];  } |

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

| **Principles(s):** This is an example of Using Effective Quality Assurance Techniques and Sanitizing Data from an outside source. Having a function individually check that the data provided is correct helps to ensure that security is happening even at the lowest layer of the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Manual Inspection | N/A | Static Code Analysis | This may require a person to observe the code since this may be a use-case basis. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Use exceptions and catch for exceptions when there may be chunks of code that could fail at execution. As an example if you expect the user may try to divide by zero or if the user attempts to open a file that does not exists, ensure there are proper try / catch blocks in place to prevent the program from unexpectedly halting. |

| **Noncompliant Code** |
| --- |
| This code is vulnerable to undefined behavior and an unexpected halt as there is no check in-place to ensure that the user hasn’t entered an invalid divisor. In some cases it may be easier to simply ensure the user cannot enter an incorrect value but in this case the user should be able to type 0 into the input as they may wish to divide by a number that contains zero such as 10. |
| int numerator = 5;  int divisor = 0;  std::cout << “Please enter the divisor: “;  std::cin >> divisor;  std::cout << numerator / divisor << std::endl; |

| **Compliant Code** |
| --- |
| The updated code will now use a try block to catch any exceptions such as a division by zero and then communicate to the user that such an issue has occurred. This allows the program to still operate as expected while ensuring that the user can’t break the program with invalid input. |
| int numerator = 5;  int divisor = 0;  std::cout << “Please enter the divisor: “;  std::cin >> divisor;  try {  std::cout << numerator / divisor << std::endl;  } catch (std::exception e) {  Std::cout << “You cannot divide by zero” << std::endl;  } |

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

| **Principles(s):** This is an example of Quality Assurance and Adopting a secure coding standard. In this case you could get away without using the try-catch block, however this could at some point lead to the application crashing. This may leave room for a Denial-of-Service attack which could have higher costs than the system simply crashing. Adhering to quality coding standards will avoid issues like this. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Somewhat Likely | Medium / High | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Do Not Depend on Order of Operations | STD-008-CPP | Do not assume that the order of operations will work as expected and rely on this. Firstly, having code without distinct operational ordering (using parenthesis) makes the code more difficult to read and understanding and can cause unexpected behavior. |

| **Noncompliant Code** |
| --- |
| This code is an example of relying on order of operations to conquer the issue. You may have confused the order and be expecting the code to mean that var1 or var2 need to be true AND var3 or var2 need to be true in order for this to pass this if statement. However, the code will actually require var1 to be true, OR var2 and var3 to be true, OR just var 2 to be true. This code is ambiguous and can lead to unexpected results and bugs. |
| if (var1 || var2 && var3 || var2) |

| **Compliant Code** |
| --- |
| The corrected code simply uses parenthesis to indicate the expected order of operations. This ensures that the order that is expected always executes, even in the cases where the order of operations would ensure that. |
| if ((var1 || var2) && (var3 || var2)) |

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

| **Principles(s):** This is an example of heeding compiler warnings as some compilers will see this issue and output warnings regarding this. This may also be another example of quality assurance as this issue could easily be identified with a basic review of the code and by following the coding standards can be avoided. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |
| CPPcheck | 2.14 | https://cppcheck.sourceforge.io |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Single Purpose Functions | STD-009-CPP | Ensure that all functions are small and single purposed as best as possible. Having overly complex functions with a high amount of responsibility leads to a lot of problems that may be difficult to troubleshoot. Having functions that are simple and single purposed means that they will be easier to test the functionality and ensure each piece of the function is working as intended. |

| **Noncompliant Code** |
| --- |
| This is a highly simplified version of the noncompliant code, however as shown in the code this would be an overly complicated function. It has the file path wrote into the function meaning that the function itself must be changed to alter the image loaded. Furthermore, this function is handling reading the file from memory and then also displaying the image to the screen. This may require using a graphics library as well as a file input-output library to achieve its goal, all of which could fail at any point. This makes troubleshooting bugs with this function very difficult. |
| void read\_and\_display\_image\_from\_file() {  std::string image\_path = “images/example.jpeg”  // Code to read image from file  // Code to display image to window  } |

| **Compliant Code** |
| --- |
| This code has now separated the roles of each function to be single purposed, allowing the functions to pass data as needed to each other while also being able to be independent. This means that these functions could be compiled and changing their behavior simply means giving them a different argument. |
| std::string image\_path = “images/example.jpeg”  Image get\_image\_from\_file(std::string file\_path) {  // Code to read the image from file  return image;  }  void display\_image\_to\_screen(Image image) {  // Code to paint the image to the screen.  } |

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

| **Principles(s):** This is an example of keeping it simple. With smaller functions that are more single purposes it becomes much easier to identify where an issue may arise within the function and it becomes much easier to thoroughly test that function with a high coverage. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Manual Inspection | N/A | Static Code Analysis | This may require a person to observe the code since this may be a use-case basis. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Array Indexing | STD-010-CPP | Ensure that arrays are properly bounded when they are needed as this may lead to undefined behavior. Arrays should never be accessed beyond their defined size and furthermore should be avoided in place of secure dynamic arrays such as the std::vector that comes with C++. |

| **Noncompliant Code** |
| --- |
| This code is a very common mistake for beginners as the array was created with a size of 10 and you are indexing up to i = 10. However, this is typically overlooked because the size vs the indexing which always starts at 0. This code will index 1 beyond the bounds of the array, causing undefined behavior. |
| int my\_array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  for (int i = 1; i <= 10; ++i) {  std::cout << my\_array[i] << std::endl;  } |

| **Compliant Code** |
| --- |
| The corrected code will now only increment up to one less than the size and uses a variable to define the size. This ensures that the size can now be easily changed and for loops that use this array will remain unaffected by the change in the size. |
| int size = 10;  int my\_array[size] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  for (int i = 0; i < size; ++i) {  std::cout << my\_array[i] << std::endl;  } |

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

| **Principles(s):** This is an excellent example of utilizing Quality Assurance techniques. In this case, a simple code review would reveal that there is an issue with this code, but this can also be overlooked. This is a common issue that requires a high level of attention to detail with security in mind. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Somewhat Likely | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 10.6 | <https://www.sonarsource.com> |  |
| CPPcheck | 2.14 | https://cppcheck.sourceforge.io |  |

### 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.

When adding security operations to the DevSecOps workflow, it is important that security is considered at each step. The steps laid out in the pre-production phase are where a lot of the security will be implemented and where security must start. With Assessing and creating a plan, it is very important to consider the threat-landscape, the attack vectors that could be used against the system, and which type of actors would be interested in this system. With more simple systems such as a website to display a portfolio, it is unlikely that any attacker will target that system, whereas with something like a hospital, it is very likely an attacker will try to attack that system.

Designing the system with security in mind is what comes next and that is where adhering to the standards created in this document come into the DevSecOps workflow. By following the coding guidelines provided we can ensure that our software is as secure as possible before it goes into production. This will involve unit testing and utilizing our tools to our advantage to make sure we are covering all of the gaps that may exist.

Building the product with security in mind means that we use tools such as OWASP to help research the libraries that we may be using and research the Open-Source Code for any vulnerabilities. We cannot control all of the factors of our code, but we can research to find any known vulnerabilities and patch work arounds to ensure that this side of the system will not be corrupted by a bad version of a library.

Finally, in the testing phase we can utilize static analyzers as different testing techniques to ensure that the code will sustain the potential attacks that may be used against it. Using something such as CPPcheck is a great example of a tool to assist in ensuring that the code meets the expected standards and performing penetration testing will help to ensure the code doesn’t have any obvious weaknesses.

Security does not stop when the code is finally out of pre-production. Responding to security events, creating patches for newfound security holes and continuing other forms of testing is the way to ensure the system remains stable and secure for the users. As shown in the diagram, this is not a linear path to the end of the DevSecOps workflow, but a loop of iterative patches that may be needed as more information becomes available on what the current issues with the codebase are.

### 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 | Medium | Somewhat Likely | Medium | High | 2 |
| STD-002-CCP | Low | Likely | Medium | Medium | 1 |
| STD-003-CCP | High | Likely | High | High | 5 |
| STD-004-CCP | High | Likely | High | High | 5 |
| STD-005-CCP | Medium | Somewhat Likely | Medium | Medium | 2 |
| STD-006-CCP | Low | Likely | Low | Low | 1 |
| STD-007-CCP | Medium | Somewhat Likely | Medium / High | Medium | 3 |
| STD-008-CCP | Low | Likely | Low | Medium | 2 |
| STD-009-CCP | Low | Unlikely | Low | Low | 1 |
| STD-010-CCP | Medium | Somewhat Likely | Medium | High | 3 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 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 | Data at rest should always remain in an encrypted state utilizing modern cryptographic libraries. We don’t need to reinvent encryption, especially when there are trusted libraries that already exist. Furthermore, any data at rest should always be encrypted. Saving plain-text data will always be a point of concern as any breaches, inside the corporation or outside the corporation, will create an ease of access for private information. |
| Encryption in flight | Data in flight should also remain encrypted in a way that is reversable via cryptographic hashes and keys that are not easily able to be replicated such as large prime numbers. We want the data to be able to be decrypted when it needs to be used, however data in flight is still a point of concern, especially for man-in-the-middle attacks. Notably, in flight means any form of transit, including within the company network as attackers may come from within the company as well. |
| Encryption in use | Data in use should also remain encrypted for as long as possible for appropriate use cases such as the handling of very sensitive data. This means that the data should be passed to processing units in the encrypted format and only be decrypted in the processing state and immediately encrypted again when the processing completes. This ensures that sensitive data remains encrypted for as long as possible, further reducing the likelihood of a breach gaining access to this data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of gaining access to a system, this may involve a username and password along with a multi-factor authenticator to gain full access to any system. We should fully adopt this policy including a multi-factor authenticator as this will ensure that people entering the system are meant to be there. Especially in the case where the system has access to sensitive information, it is vital that there is an authentication wall that prevents people from simply entering the system. |
| Authorization | Authorization is how much privilege a user has when they have entered the system. This determines if a user can access only a slice of sensitive data, if the user can modify sensitive data or change other users on the system. The best policy to adopt is the one of least privilege. Users should only have as much privilege as they need to accomplish their job. This means that a software engineer should not have access to change other users accounts while admins should not have access to change the source code of our systems. Malicious attackers could come from within the corporation or outside the corporation, this policy ensures that any attacker that gains access to our system will only be able to access a small slice of the overall system. |
| Accounting | Accounting is the process of tracking user traffic via something like logs. This will create text logs describing what user went to which part of our system and how they decided to interact with that part of the system. This should include changes to the database, login attempts and any modifications that are attempted on the system as a whole. This acts as a trail to be followed when there are security breaches. This is incredibly important on covering the cases where a successful security breach makes it through our defense in depth strategy. Using the logs we will be able to trace back what the malicious user did within our system to gain an understanding of how they bypassed our security measures. This ultimately allows us to then fix these holes in our security and prevent similar issues from happening again. |

**\***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 |  |
| 1.1 | 07/21/2024 | Coding Standards Added | Aaron McDonald |  |
| 1.2 | 08/11/2024 | Defense-in-Depth Strategy Added | Aaron McDonald |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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