**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 | Validate any input that comes from an outside source. Sources can be automated or manual user input, CLI commands, database queries, and file access. External users cannot be trusted until their input is verified, whether the user has intentional inputs or accidental. |
| 1. Heed Compiler Warnings | Compiler warnings tell the developer the risks of a certain implementation. Some implementations of code can have a vulnerability that would not even compile, but some compilers only provide a warning. Adjust code with the help of tools and IDE’s to remove all warnings properly. |
| 1. Architect and Design for Security Policies | The best security outside of software is the actual hardware and architecture supporting that software. If there are different authorizations for actions, consider decoupling the higher and lower privileges into separate applications and networks. |
| 1. Keep It Simple | As complexity increases, so does the risk of introducing errors that cause vulnerabilities. A system that is kept simple and does each action separately in code and architecture is both easier to maintain and pen test for any risks. More complex systems take more resources to test and verify its integrity. A simple system does what it is designed to do, and nothing more. |
| 1. Default Deny | Always deny a request for permissions or resources regardless of origin if it does not meet the security standards. Whitelist security policies only let in what is allowed and deny everything else. For example, there is no limit to how many wrong passwords a user can enter, so we should only check for the one we expect and reject anything else. This reduces complexity, maintenance, and effort. |
| 1. Adhere to the Principle of Least Privilege | The least number of permissions should be granted to complete a process, and only for the duration of that process. This reduces the amount of time and opportunity for an attack targeting the escalation of privileges and causing damage. Even when an attack breaks through, the damage is secluded to the privileges of that exploited system or user. |
| 1. Sanitize Data Sent to Other Systems | Complex or decoupled systems do not necessarily know where or how the request is coming through, so the caller of this system must sanitize the data before sending. Systems such as databases can be exposed to this attack, commonly in the form of an SQL injection, where the database is simply doing what it is told. The caller of the database has to sanitize this request to make sure the database does only what it is intended to do. |
| 1. Practice Defense in Depth | Secure systems and reduce uncertainty by layering multiple security practices onto one another for processes. This reduces the likelihood of a vulnerability becoming an exploit by enabling systems to withstand some layers failing yet others succeeding. |
| 1. Use Effective Quality Assurance Techniques | Use proven security techniques to ensure that a system is thoroughly tested and ready for any appearance of attacks, exploits, and/or vulnerabilities. Code reviews and pen tests allow for more robust code. Third-party companies and testers can also greatly help in finding vulnerabilities by offering different approaches and views on the same code that the company already reviewed itself. |
| 1. Adopt a Secure Coding Standard | Some languages and platforms are vastly different from one another. Adopt a standard that addresses these issues. For example, C++ can have issues with buffer overflows and Python can have issues with abstractly written code. |

### 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 | Making sure the data types of inputs and allocated memory match or is handled improves the reliability of the system. For example, it prevents infinite loops on input requests such as the “cin” input stream. |

| **Noncompliant Code** |
| --- |
| The “num” variable requires an integer, but if the user enters a character, the next call to “cin” breaks and the loop runs forever. |
| using namespace std;  int main() {  int num = 0;  char loopAgain = ‘y’;  while (loopAgain == ‘y’ || loopAgain == ‘Y’) {  cout << “Enter a number: “ << endl;  cin >> num;  cout << “You entered: “ << num << endl;  cout << “Want to enter another number?” << endl;  cin >> loopAgain;  }  cout << “done.” << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This code places the “cin” input stream in a while loop to verify that the input is not breaking the input stream and keeps requesting another number for the “num” variable. The “cin” input stream is reset if it is broken by the wrong input by clearing the error state and ignoring that input. |
| using namespace std;  int main() {  int num = 0;  char loopAgain = ‘y’;  while (loopAgain == ‘y’ || loopAgain == ‘Y’) {  cout << “Enter a number: “ << endl;  while (!(cin >> num)) {  cout << “Only enter a valid number: “ << endl;  cin.clear();  cin.ignore(200, ‘\n’);  }  cout << “You entered: “ << num << endl;  cout << “Want to enter another number?” << endl;  cin >> loopAgain;  }  cout << “done.” << endl;  return 0;  } |

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

| **Principles(s):** Don’t Trust the User. A user should not be trusted or relied on to always enter expected data types or amounts of data. Sometimes the user will enter wrong data accidentally or to intentionally break the application in order to expose a vulnerability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | If a data’s value exceeds the memory that it is allocated to, then it will overflow or underflow, leading to abnormally large or small values. Also known as a numeric overflow. This can cause system failures and even exploits as systems use the value for other operations. |

| **Noncompliant Code** |
| --- |
| The “largeNumber” variable is the maximum value that an “int’ type can have. If more is added to it, it will wrap around to the lower end of the range of numbers such as a negative number, depending on the implementation of the system. The “newNumber” variable will be incorrect, but the program may continue to run. |
| int largeNumber = std::numeric\_limits<int>::max();  int newNumber = 100 + largeNumber; |

| **Compliant Code** |
| --- |
| Here, the operation is compared before it is completed to track whether a buffer overflow has occurred. It is done by subtracting the number from the max value possible (avoiding another overflow if we added it), then comparing that to the number before the operation. |
| int largeNumber = std::numeric\_limits<int>::max();  if (largeNumber < std::numeric\_limits<int>::max() - 100) {  cout << “No Overflow” << endl;  int newNumber = 100 + largeNumber;  }  else {  cout << “Overflow Occurred! Exiting...” << endl;  return –1;  }  return 0; |

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

| **Principles(s):** Don’t trust the user (or arbitrary data inputs). When working with numeric values, a variable can only be represented by a limited number of bytes, yet a user can enter more than expected. If the buffer overflows or underflows, then an application can have unpredictable behavior, and worse if tied to financial systems such as banks and loan services. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Strings should be defined as a “string” type from the string library to prevent errors related to buffer overflow. |

| **Noncompliant Code** |
| --- |
| This uses a char array, and so it has to be allocated carefully. |
| char[12] someString = “hello world!”; |

| **Compliant Code** |
| --- |
| This uses a string type to dynamically allocate memory by using the “string” keyword from the string library. |
| #include <string>  string someString = “hello world!”; |

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

| **Principles(s):** Don’t trust the user. This string data can be used by an attacker to load arbitrary code into the target processing server or system.  Sanitize data sent to other systems. Sometimes this error can even occur internally, where incompatible data is sent in large or small amounts to another system. The system must be able to expect the right type and amount of data before it sets it on variables and other code within the system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Very low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Sanitizing input is a must for any external source of information. Strings and requests can contain an SQL injection if they are executed directly by a database management system. This can cause a loss or theft of data. |

| **Noncompliant Code** |
| --- |
| Here the SQL query is run without any sanitation. The query can contain a “or true” statement at the end of its string, and cause all data to be lost or stolen. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records) {   records.clear();   char\* error\_message;  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "Data failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }   return true; } |

| **Compliant Code** |
| --- |
| This code runs a function that will throw an exception if an sql injection is detected, and will not run the query by returning. SQL injections have a consistent pattern of “or true” at the end of the string. This can be detected by reading whether the word “true” appears at the end, or reading whether two instances appear twice before and after an equal sign, such “or 1 == 1”. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records) {  try {  checkForInjection(sql)  }  catch(sql\_injection\_found& error) {  cerr << error.what() << endl;  return false;  }  records.clear();   char\* error\_message;  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "Data failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }   return true; } |

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

| **Principles(s):** Validate input. Code must validate data from all untrusted sources. This avoids execution of arbitrary code, or in this case, illegal access to SQL database commands and data.  Sanitize data sent to other systems. Commonly this data is sent from one system to another. Usually, a database has a server layer to avoid exposing the database directly to the public. However, SQL injection can still occur depending on how the data is handled and sent from one layer or more to the database layer. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | An attacker can overwrite memory with their own data. This data can only be the size of 4 bytes, enough to overwrite the return pointer on the stack with an address to arbitrary malicious code. |

| **Noncompliant Code** |
| --- |
| A technique called “unlink” is used to exploit the buffer overflow vulnerability to trick the “unlink()” macro into writing 4 bytes of data to any location of choice. |
| #include <stdlib.h>  #include <string.h>  int main(int argc, char \*argv[]) {  char \*first, \*second, \*third;  first = malloc(666);  second = malloc(12);  third = malloc(12);  strcpy(first, argv[1]);  free(first);  free(second);  free(third);  return(0);  } |

| **Compliant Code** |
| --- |
| This code uses a runtime tool called Valgrind that detects a buffer overflow if a longer string is given for input on a memory allocation. (Seacord 2013) |
| /\* caesar.c \*/  #define LINELENGTH 80  // other code...  if (!(inbuf = malloc(LINELENGTH)))  errx(1, "Couldn't allocate memory.");  while (fgets(inbuf, 100, infile) |

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

| **Principles(s):** Automation is your friend. It can be difficult to track what kind of memory allocations will cause vulnerabilities and bugs. Automated tools can help speed up and assist the process of discovering and mitigating errors in memory allocation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20.0 | Dynamic Analysis tool for C and C++ | Valgrind is an automation dynamic analysis tool for C that checks programs for detecting memory leaks and profiles. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use assertions for when code should never break. Using assertions helps find errors and exact reasons for these errors by showing the comparison that failed given certain values and data types. |

| **Noncompliant Code** |
| --- |
| This code gets a number from an arbitrary source, determines if it is even or odd, and executes code accordingly. The problem is that the comparison assumes the number is always positive, thus causing an error that is not obvious at first glance. |
| int i = getSomeNumber();    if( i % 2 == 1 ) {  handleOdd(i);  }  else {  handleEven(i);  } |

| **Compliant Code** |
| --- |
| This code uses an assertion that shows the number is in fact even, or else the program should crash. Since there is a new discovery of the possibility that the comparison can resolve to “-1”, other code should handle this possibility through exceptions, or simply allowing the program to halt. |
| int i = getSomeNumber();    if( i % 2 == 1 ) {  handleOdd(i);  }  else {  ASSERT(i % 2 == 0);  handleEven(i);  } |

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

| **Principles(s):** Use effective quality assurance techniques. Code should be thoroughly tested to ensure that logical errors never occur, or should never occur in practice. Assertions help with ensuring that certain operations should produce predictable results.  Heed compiler warnings. Commonly, compiler warnings will display information regarding logical or security issues. These warnings are meant to assist with developing a secure and robust application code. Depending on the compiler and IDE version, the errors should be listed whenever they occur in code, and provide a means to remediate the issue. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |
| Visual Studio | 17.6.0 | Integrated Development Environment | Visual Studio is an IDE that analyzes and assists with the development of code. It will report compiler warnings and general coding best practices, as well as some logical errors. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | For a language such as C++, a function must return a type. Sometimes a function has to return information about a failure within itself, without returning a value. An exception must be put within functions with user input as arguments in order to prevent vulnerabilities or edge cases from causing issues in the program. Thus, exceptions must be handled properly when called, otherwise the program execution will halt. |

| **Noncompliant Code** |
| --- |
| This code defines and calls a function with a throw statement. The main function calls this function erroneously as there is a possibility of an exception being thrown. The compiler will most likely fail to compile this code. |
| int someFunction(int param1) {  if (param1 == 1) {  throw invalid\_argument(“Cannot handle ‘1’ as argument etc...”);  }  }  int main() {  someFunction(2);  } |

| **Compliant Code** |
| --- |
| This code handles the exception and allows the program to continue. A “try and catch” block is used around the called function to catch a specifically thrown exception and display the error on the console. This allows the exception to do what it is supposed to do while not interfering with the flow of the program if it needs to keep running. |
| int someFunction(int param1) {  if (param1 == 1) {  throw invalid\_argument(“Cannot handle ‘1’ as argument etc...”);  }  }  int main() {  try {  someFunction(2);  }  catch (invalid\_argument& error) {  cerr << error.what() << endl;  return –1;  }  return 0;  } |

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

| **Principles(s):** Keep it secure and simple. Sometimes it is simpler to throw an exception than try to work around a function’s logic. If a return value is needed in a function that can fail, then it is improper to return an arbitrary value to then have to be checked outside the function call explicitly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Visual Studio | 17.6.0 | Integrated Development Environment | Visual Studio is an IDE that analyzes and assists with the development of code. It will report compiler warnings and general coding best practices, as well as some logical errors. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Specific Exceptions | STD-008-CPP | Just like coding standard STD-007-CPP “Exceptions”, exceptions can be used to halt the execution of code and/or change the execution of code if faulty input is used on function arguments or dynamic code in general. But specific exceptions must be thrown in order to track bugs and code execution. If exceptions are caught in general, then there is no way of knowing what error occurred, and whether the handling code is suited for that error. |

| **Noncompliant Code** |
| --- |
| The catch block uses “...” as a way to catch all exceptions. This may handle all exceptions, but there is no way to document or audit what happened at runtime. |
| int someFunction(int param1) {  if (param1 == 1) {  throw invalid\_argument(“Cannot handle ‘1’ as argument etc...”);  }  }  int main() {  try {  someFunction(2);  }  catch (...) {  cerr << error.what() << endl;  return –1;  }  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant code below catches the exact exception (invalid\_argument) that can be thrown by the function “someFunction()” and does not catch any other exception to prevent abstract handling and unpredictable code. |
| int someFunction(int param1) {  if (param1 == 1) {  throw invalid\_argument(“Cannot handle ‘1’ as argument etc...”);  }  }  int main() {  try {  someFunction(2);  }  catch (invalid\_argument& error) {  cerr << error.what() << endl;  return –1;  }  return 0;  } |

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

| **Principles(s):** Adopt a secure coding standard. A secure standard should be implemented for exceptions where specific exceptions are thrown and/or created as necessary. Catching all exceptions does not allow for debugging and audit capabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer Overflow | STD-009-CPP | Preventing a buffer overflow stymies many exploits that can occur such as overwriting the control address to malicious code. Buffer overflows also crash programs which can lead to other exploits. Even if the crash is handled, malicious code can still be run by using the overflowing memory access caused by the buffer overflow. |

| **Noncompliant Code** |
| --- |
| The variable “someCharacters” has only 10 bytes of memory allocated, yet the user can simply enter more than 10 characters to cause a buffer overflow. The buffer overflow can be used to read, delete, or update the “secretPassword” variable arbitrarily. Also, subsequent calls to the “cin” input stream will also be filled with the remaining characters that are truncated. |
| using namespace std;  int main() {  char[10] someCharacters;  char[12] secretPassord = “hello world!”;  cout << “Enter a word: “ << endl;  cin >> someCharacters;  return 0;  } |

| **Compliant Code** |
| --- |
| This code uses the “string” type in the standard library to prevent any buffer overflows by dynamically allocating memory. Truncated strings are also mitigated by allowing the “cin” input stream to discard the extra characters safely. |
| #include <string>;  using namespace std;  int main() {  string someCharacters;  char[12] secretPassord = “hello world!”;  cout << “Enter a word: “ << endl;  cin >> someCharacters;  return 0;  } |

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

| **Principles(s):** Don’t Trust the User. A malicious user can send more data than expected, and send this data with arbitrary code, or even get access to otherwise secure data.  Sanitize data sent to other systems. A buffer overflow can still happen even if it is passed to another system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Static analysis tool for C++ Code | Cppcheck is a static analysis tool that analyzes written code and reports any vulnerabilities or logic errors. It provides a list of errors and a suggestion for solving the issue. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Global Variables | STD-010-CPP | The use of global variables can cause many issues related to maintainability and vulnerability. Global variables are not scoped out of memory and can be overwritten by using other vulnerabilities within the code. This can cause many issues for the program, especially if much of the code is dependent on it. |

| **Noncompliant Code** |
| --- |
| An arbitrary function uses a global variable as an argument, which can cause issues if the variable is changed within the function by mistake, or modified through another vulnerability. |
| using namespace std;  int importantVariable = 24;  int main() {  useVariableInArgument(importantVariable);  } |

| **Compliant Code** |
| --- |
| Here, the needed variable is placed in a class as a static member to allow the function to use the variable safely, also without risk of modification by using the “const” keyword. |
| using namespace std;  class someClass {  static const int importantVariable = 24;  }  int main() {  someClass anObject;  useVariableInArgument(anObject.importantVariable);  } |

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

| **Principles(s):** Architect and design for security policies. The software should be implemented in a way that adheres to best practices and secure systems. Global variables can lead to issues in application behavior and performance. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Visual Studio | 17.6.0 | Integrated Development Environment | Visual Studio is an IDE that analyzes and assists with the development of code. It will report compiler warnings and general coding best practices, as well as some logical errors. |

### 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.

Verifying and testing for vulnerabilities can be automated in the DevSecOps diagram. This will increase the speed of planning and pre-production. Much of what occurs at this stage can be safely automated using reputable tools to ensure that the code builds and repositories used are secured as required. Next, monitoring and detecting can be automated as well. This includes logging and alerts in a system that tracks access and manipulation of data and systems. Automation can greatly increase the quality and quantity of the logs and allow for better auditing in the future. Response to attacks can be automated in some ways such as turning off services and rolling back. When a system is exposed, shutting down the system should be graceful and automatically handled by code and architecture to ensure that damage is limited and secluded. Lastly, maintenance and stabilization after an attack can also be automated to ensure that services are back online as soon as possible for users and developers. If a system needs complex configurations and setup, then automation can assist with the task of returning a system back to its original state and function after an attack.

### 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 | Likely | Low | High | 4 |
| STD-002-CPP | High | Unlikely | Medium | High | 5 |
| STD-003-CPP | High | Unlikely | Very low | High | 4 |
| STD-004-CPP | High | Likely | [Medium | High | 5 |
| STD-005-CPP | Medium | Unlikely | Medium | Medium | 3 |
| STD-006-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-007-CPP | Low | Unlikely | Low | Medium | 3 |
| STD-008-CPP | Low | Very Unlikely | Low | Low | 1 |
| STD-009-CPP | Medium | Likely | Low | Medium | 3 |
| STD-010-CPP | Low | Unlikely | [Medium | Medium | 2 |

### 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 | All data should be encrypted when it is stored short and long term to avoid exposure to exploits. This data is usually in a database, where it should be encrypted using encryption keys and algorithms. This applies in many situations where sensitive data such as credit card numbers must be stored for convenience or other application requirements. When a database is exposed, the data should be secured through encryption, denying attackers the ability to read the data. |
| Encryption at flight | All data must be encrypted while in transit to and from users by networks to any server. Sometimes attackers can use website and application traffic to track and eavesdrop on communication to steal credit card information or other personal details, most commonly on platforms such as e-commerce. |
| Encryption in use | All data should be encrypted while stored in Random Access Memory or other means while being used for business logic. Some systems connected to the world wide web can be exposed to viruses or coding bugs that can accidentally reveal data as it is being processed, such as a social security number while processing tax documents online. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Users must be identified when interacting with any sensitive information. Users will submit credentials, usually an email and password, to identify themselves as who they are and what role they might have. This policy applies when data needs to be accessed by any user that may or may not be allowed to access the data. Usually, the user would need to be authenticated when they request private or restricted data. |
| Authorization | Users must have a limited amount of access to data. Users should only access the data they need at the time they need it, and nothing more. This applies best in situations where users are divided into teams or groups based on their profession, where they need access to separate data. By restricting data access by role, any exposure to an attacker on a user’s account would be secluded to that data they have access to, therefore limiting the impact. |
| Accounting | Actions by users and administrators must be tracked and recorded for both security and development reasons. Users may try to access certain data that they don’t have permissions to, and so a system must record that to track how sensitive data is being requested. This allows attacks to be exposed earlier and with greater detail. This also applies to developers who need to access logs or audit the system to find out how security has been violated during or after development or testing. |

**\***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 | 05/21/2023 | 10 Coding Standards: Module 3 | Ethan Klukkert |  |
| 2.1 | 06/11/2023 | Finish Policy standards, automation, summary, risk assessment. | Ethan Klukkert |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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