**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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 14](#_Toc52464063)

[Coding Standard 6 16](#_Toc52464064)

[Coding Standard 7 18](#_Toc52464065)

[Coding Standard 8 20](#_Toc52464066)

[Coding Standard 9 22](#_Toc52464067)

[Coding Standard 10 24](#_Toc52464068)

[Defense-in-Depth Illustration 26](#_Toc52464069)

[Project One 26](#_Toc52464070)

[1. Revise the C/C++ Standards 26](#_Toc52464071)

[2. Risk Assessment 26](#_Toc52464072)

[3. Automated Detection 26](#_Toc52464073)

[4. Automation 26](#_Toc52464074)

[5. Summary of Risk Assessments 27](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 28](#_Toc52464076)

[7. Map the Principles 29](#_Toc52464077)

[Audit Controls and Management 31](#_Toc52464078)

[Enforcement 31](#_Toc52464079)

[Exceptions Process 31](#_Toc52464080)

[Distribution 32](#_Toc52464081)

[Policy Change Control 32](#_Toc52464082)

[Policy Version History 32](#_Toc52464083)

[Appendix A Lookups 32](#_Toc52464084)

[Approved C/C++ Language Acronyms 32](#_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** | **Explanation** |
| --- | --- |
| 1. ValidateInput Data | Input validation is a common cause of vulnerabilities. Therefore, it is important to identify all input sources as well as specify and validate input from untrustworthy sources. Input validation can be implemented by checking that an input is of a specific data type and/or within a certain value range. |
| 1. Heed Compiler Warnings | Compilers communicate errors within code that can prevent an application from compiling or cause potential problems when executed. Paying attention to compiler warnings can inform developers of problems within existing code and allow them to be resolved before a problem becomes an even bigger problem. |
| 1. Architect and Design for Security Policies | Positioning security at the forefront of architecture and design mitigates the impact of security attacks and their consequences. By incorporating security policies into the architecture and design processes, developers can proactively protect their work from being compromised during later stages of development and, eventually, deployment. |
| 1. Keep It Simple | Developing code that is complex can be difficult to maintain and increases the likelihood that a vulnerability will arise. Keeping code simple allows developers to identify bugs more easily in their code. Additionally, it promotes teamwork since other developers can more easily infer the functionality behind code. |
| 1. Default Deny | Access that allows a user to interact with an application, its functionality, or the data it contains should be denied by default unless explicitly authorized. This form of access control enables security rules to govern who has access to what resources. It prevents unauthorized access and deters security risks. |
| 1. Adhere to the Principle of Least Privilege | Processes and users should have the bare minimum level of permission needed to perform the operations and access the data they are expected to. Establishing these limitations as a form of access control ensures that an unprivileged user or process cannot access critical and sensitive resources that can compromise the integrity and performance of an application. |
| 1. Sanitize Data Sent to Other Systems | It is necessary to ensure that data conforms to the requirements of the systems to which it is passed in order to prevent SQL, command, and other injection attacks. Data should be secured, cleaned, and filtered to ensure that it is in the expected format, of the expected type, and does not contain unsafe characters. |
| 1. Practice Defense in Depth | Multiple defense strategies should be layered in a redundant fashion such that if one layer of defense is inadequate then another layer can be implemented to prevent a security flaw from being exploited. Defensive strategies include security tools, mechanisms and policies that work in tandem to mitigate vulnerabilities. |
| 1. Use Effective Quality Assurance Techniques | Implementing quality assurance techniques such as fuzz testing, penetration testing, source code audits and independent security reviews can result in a more secure system by effectively identifying and eliminating vulnerabilities. Utilizing an external review process can correct false assumptions and catch bugs that would have otherwise went unnoticed by the original developer(s). |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard facilitates the writing of secure code and alignment throughout the design, development and deployment processes. Establishing the guidelines by which code is created serves to reduce security concerns and improve readability and maintainability since all developers will be held to the same expectations. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Declare identifiers before using them. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below omits the data type when declaring variable global\_x. It is difficult for the compiler to infer proper memory allocation when it does not know what its type-specifier is. |
| extern global\_x; |

| **Compliant Code** |
| --- |
| The compliant code example below explicitly declares the data type as an integer so that the compiler can allocate sufficient memory for its value. It is also important to select the appropriate data type. |
| extern int global\_x; |

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

| **Principles(s):**  *[2] Heed Compiler Warnings* – The compiler will automatically check for identifier declaration. When it identifies a missing data type declaration, the compiler should produce a warning that informs developers of an undeclared identifier error.  *[4] Keep It Simple* – Declaring identifiers maintains code simplicity such that it makes the design more transparent and is easier for developers to infer the purpose of a variable or function when it is accurately declared. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low (2) | Unlikely | Low | **P3** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule DCL31-C](https://www.mathworks.com/help/bugfinder/ref/certcruledcl31c.html) | This static code analysis tool checks for types not explicitly specified and implicit function declaration. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-DCL31-a | This integrated software testing solution checks that all functions are declared before they are used. |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | Type-specifier missing | This tool performs a comprehensive check to detect undefined behavior. |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | Type-specifier  Function-return-type  Implicit-function-declaration  Undeclared-parameter | This static code analyzer tool is a static code analyzer that detects runtime errors and invalid concurrent behavior. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensure that unsigned integer operations do not wrap. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below does not check if the arithmetic operation between two unsigned integers will cause a wraparound. Wraparound can occur by adding two unsigned values whose sum is larger than the maximum value that can be stored in the resulting type; this creates vulnerabilities. |
| unsigned int ui1, ui2, usum;  /\* Initialize ui1 and ui2 \*/  usum = ui1 + ui2; |

| **Compliant Code** |
| --- |
| The compliant code example below uses the limits.h header file to obtain the maximum unsigned integer value and a precondition test to check that the arithmetic operation will not cause a wraparound before it is performed. When the value exceeds the upper limit, the code within the if statement will execute to handle the error gracefully. |
| #include <limits.h>  unsigned int ui1, ui2, usum;  /\* Initialize ui1 and ui2 \*/  if (UINT\_MAX - ui1 < ui2) {  /\* handle error condition \*/  }  else {  usum = ui1 + ui2;  } |

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

| **Principles(s):**  *[1] Validate Input Data* – Proper input validation can eliminate the risk of wraparound and overflow/underflow and should be implemented whenever user input is used for an arithmetic operation.  *[2] Heed Compiler Warnings* – The compiler should generate warnings that detect potential wraparound and overflow/underflow risks which should then be mitigated by developers.  *[9] Use Effective Quality Assurance Techniques* – Good quality assurance techniques can be effective in identifying and eliminating wraparound vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High (4) | Likely | High | **P9** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) | unknown | Overflow detection | This tool can detect when unsigned integer operations wrap by ensuring that operations are checked for overflow before being performed |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | Unsigned overflow | This tool provides exhaustive test coverage that identifies all bugs and uses a series of formal methods to mathematically model code to prove that there are no bugs left once all issues detected are corrected. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule INT30-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint30c.html) | This static code analysis tool checks for unsigned integer overflow and unsigned integer constant overflow. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | This static code analysis tool checks for overflows that occur due to arithmetic operations where the resulting value cannot be represented by a given data type. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below does not check that the allocated space for the user\_input variable is sufficient for the value entered by a user before storing it. This creates a vulnerability that can lead to a buffer overflow when exploited. |
| #include <iomanip>  #include <iostream>  int main()  {  const std::string account\_number = "CharlieBrown42";  char user\_input[20];  std::cout << "Enter a value: ";  std::cin >> user\_input;  std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  } |

| **Compliant Code** |
| --- |
| The compliant code example below uses the getline function to collect the value of user\_input and set the maximum value that can be stored. It takes into consideration the null terminator and triggers an if statement when the value exceeds the maximum space allocated for it. Using a try-catch statement allows the try block to attempt to take user input and the catch block to handle errors that occur. This helps prevent buffer overflow. |
| #include <iomanip>  #include <iostream>  int main()  {  const std::string account\_number = "CharlieBrown42";  char user\_input[20];  int x = -1;  std::cout << "Enter a value: ";  try {  std::cin.getline(user\_input, sizeof(user\_input) + 1);  if (!std::cin) {  throw -1;  }  std::cout << "\nYou entered: " << user\_input << std::endl;  std::cout << std::endl << "Account Number = " << account\_number  << std::endl;  }  catch (int x) {  std::cout << "\nError: The value entered is too long."  << std::endl;  }  } |

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

| **Principles(s):**  *[1] Validate Input* – User input must be checked to ensure that whatever a user enters does not exceed the maximum space allocated for it, including null terminators.  *[2] Heed Compiler Warnings* – Compiler generated warnings like C4739 (reference to variable 'variable' exceeds its storage space) must always be addressed to ensure that a destination is of sufficient size to hold character the null-termination character.  *[7] Sanitize Data Sent to Other Systems* – A destination system must be checked to ensure that it has allocated a sufficient size of space for incoming data to prevent vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High (5) | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | Buffer overflow | This static code analyzer tool reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | CertC-STR31 | This tool detects calls to unsafe string functions that may cause buffer overflow and potential buffer overruns, including those caused by unsafe usage of fscanf(). |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | This static code analysis tool can detect buffer and type overruns and when there is no space for the null terminator. It also produces warnings when library functions prone to internal buffer overflows are used. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule STR31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr31c.html) | This static code analysis tool checks for the use of dangerous standard functions, missing null terminator in string arrays, buffer overflows from incorrect string format specifier, destination buffer overflows in string manipulation, and insufficient destination buffer sizes. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Sanitize data passed to complex subsystems. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below is susceptible to SQL injection because the run\_query function takes user input for the value of the username variable and uses it in an SQL data base query without first checking if it is tainted. |
| bool run\_query(sqlite3\* db, std::vector< user\_record >& records) {  std::string username;  std::string sql;  std::cin >> username;  sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='" + username  + "'";  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** |
| --- |
| The compliant code below uses regular expressions to ensure that tainted user input is not used to query a database. The if-statement will execute when the patterns defined by the regular expressions are identified within the value of the username variable and causes the function to return false; this prevents the code that accesses the database from being reached. |
| bool run\_query(sqlite3\* db, std::vector< user\_record >& records) {  std::string username;  std::string sql;  std::regex check\_str("or '[a-zA-Z]+'='[a-zA-Z]+'",  std::regex\_constants::ECMAScript | std::regex\_constants::icase);  std::regex check\_num("or [1-9]+=[1-9]+", std::regex\_constants::ECMAScript  | std::regex\_constants::icase);  std::cin >> username;  if (std::regex\_search(sql, check\_str) || std::regex\_search(sql,  check\_num)) {  /\* handle error \*/  return false;  }  sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='" + username  + "'";  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):**  *[7] Sanitize Data Sent to Other Systems* – Removing questionable characters and sequences of characters from data before the data is sent to another system will protect it from SQL, command, or other injection attacks.  *[8] Practice Defense in Depth* -Using multiple defensive layers is essential to protect complex subsystems from attacks such as SQL injection. An example is combining input validation and unit testing to reduce the likelihood of an injection attack.  *[9] Use Effective Quality Assurance Techniques* – Comprehensive testing can check for vulnerabilities created from data passed between complex subsystems. Penetration testing and source code audits are two ways to check that code can withstand injection attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High (5) | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | This static code analysis tool detect command injection, format string injection, SQL injection, LDAP injection, Library injection vulnerabilities. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | This integrated software testing solution protects against command injection, file name injection, and SQL injection. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rec. STR02-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str02c.html) | This static code analysis tool checks for execution of externally controlled command, commands executed from externally controlled paths, libraries loaded from externally controlled paths to identify potential vulnerabilities. |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | CERT C++  ISO/IEC TS 17961:2013 | This static code analyzer offers taint analysis to identify vulnerabilities in software by tracking the flow of data through the system and stubbing to test code in isolation. |

#### 

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below dereferences p after having first freed the memory referenced by p. While reading from freed memory may succeed, it can cause undefined behavior and there is no guarantee that the contents of the memory has not been altered. If the memory has been reallocated, it may overwrite the memory referenced by p too. |
| for (p = head; p != NULL; p = p->next) {   free(p);  } |

| **Compliant Code** |
| --- |
| The compliant code example below uses the variable q to store a reference to p->next before freeing p. The ensures that freed memory is not accessed during the iterations of the for-loop and maintains the integrity of the code. It prevents a Use-After-Free vulnerability from being exploited which can cause arbitrary code to be executed. |
| for (p = head; p != NULL; p = q) {  q = p->next;  free(p);  } |

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

| **Principles(s):**  *[2] Heed Compiler Warnings* – Modify code that generate warnings when a Use-After-Free vulnerability is detected to prevent exploits during deployment.  *[9] Use Effective Quality Assurance Techniques* – Implement protocols to detect when access to freed memory is attempted and output an error message to inform of the mistake. Also, use error handling to ensure that code is able to gracefully deal with improper memory management. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High (5) | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | v7.5.0 | USE\_AFTER\_FREE | This proprietary static code analysis tool can detect specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule MEM30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemem30c.html) | This static code analysis tool checks for out of bounds pointer access, deallocation of previously deallocated pointer, and use of a previously freed pointer. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-MEM50-a | This integrated software testing solution prevents the use of resources that have been freed. |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | This front-end compiler tool uses clang-tidy to check for access violations, but it does not catch all of them. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use static assertions to test the values of constant expressions. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below has an assertion placed within a function. The diagnostic occurs only at runtime and only if the function containing the assertion is called. |
| #include <assert.h>    struct counter {  unsigned char STATE;  unsigned int COUNT;  unsigned int TOTAL;  };    int check\_counter(void) {  assert(sizeof(struct counter) == sizeof(unsigned char) +  sizeof(unsigned int) + sizeof(unsigned int), “ERROR!”);  } |

| **Compliant Code** |
| --- |
| The compliant code example below places a static assertion below the definition of the actual structure to which it refers. This allows the static assertion to be performed during compile time rather than runtime and facilitates the diagnoses of software defects that may result in vulnerabilities. |
| #include <assert.h>    struct counter {  unsigned char STATE;  unsigned int COUNT;  unsigned int TOTAL;  };    static\_assert(sizeof(struct counter) == sizeof(unsigned char) +  sizeof(unsigned int) + sizeof(unsigned int), “ERROR!”); |

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

| **Principles(s):**  *[4] Keep It Simple* – Utilizing static assertions facilitates development by catching errors sooner at compile time instead of runtime.  *[8] Practice Defense In Depth* – Static assertions can be used as a layer to test logic in code at compilation time. It should be part of a layered approach where another layer tests logic in code during runtime.  *[10] Adopt A Secure Coding Standard* – Applying a coding standard for all applicable projects will help to ensure compliance by the development team and will encourage proper implementation and adoption of static assertions in way that their use in future projects may become second nature. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low (1) | Unlikely | High | **P1** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | (customization) | This static code analysis tool allows users to implement a custom check that reports uses of the assert() macro. |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) | Unknown | static-assert | This tool can be used to recognize macro invocation and to look for calls to assert(). It can inform when code should use static-assert. |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.DCL03 | This static code analysis tool can be used to automatically check for static assertions in C++ programs to validate that coding standards are adhered to. |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 7.2.0 | [misc-static-assert](https://releases.llvm.org/16.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/misc/static-assert.html#:~:text=misc-static-assert%C2%B6%20cert-dcl03-credirects%20here%20as%20an%20alias%20for%20this,compile%20time%20which%20is%20safer%20and%20more%20efficient.) | This front-end compiler tool can replace assert() with static\_assert() when the condition is evaluable at compile time. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example below, the function function\_that\_throws() uses noexcept(false) to specify that the function can propagate an exception. However, there is no matching handler within other\_function() or in main to catch an exception when one is thrown which can cause the process to terminate abnormally. |
| void function\_that\_throws() noexcept(false);    void other\_function() {  function\_that\_throws();  }    int main() {  other\_function();  } |

| **Compliant Code** |
| --- |
| The compliant code example below uses a try-catch statement within other\_function() to handle all exceptions thrown by function\_that\_throws(). Aside from catching exceptions, the matching catch-all handler allows the process to terminate gracefully when an exception is caught. |
| void function\_that\_throws() noexcept(false);    void other\_function() {  try {  function\_that\_throws();  } catch(…) {  /\* handle error condition \*/  }  }    int main() {  other\_function();  } |

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

| **Principles(s):**  *[1] Validate Input* – Exception handling can be used to validate input to ensure it conforms to standards and does not cause erroneous behavior.  *[3] Architect and Design For Security Policies* – Developing code that handles all exceptions is a way to implement and enforce security policies as well as maintain the code’s integrity.  *[8] Practice Defense In Depth* – Handling all exceptions can be a layer in a defense in depth approach that ensures code runs the way it is supposed to and that it can deal with problems in an efficient way. Assertions can be combined to check that all exceptions are being handled.  **NOTE**: It is important to note that error hiding by using a catch-all should be avoided whenever possible. Distinctly identifying exceptions thrown allows for proper handling. However, it is acceptable for certain situations such as prototyping. Contact the chief information security officer if you have any questions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low (2) | Probable | Medium | **P4** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | This integrated software testing solution will check that a program will always catch exceptions and that each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | This static code analysis tool checks for unhandled exceptions. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.STRUCT.UCTCH | This static code analysis tool checks for unreachable catches. |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork?_gl=1*1oeg9jj*_ga*MTQ4NTAwNjEyMy4xNjg2MjMyNDcw*_ga_87WECW6HCS*MTY4NjI1NjMwOS4zLjEuMTY4NjI1NjMxMy4wLjAuMA..) | 2023.1 | [MISRA.CATCH.ALL](https://help.klocwork.com/current/en-us/concepts/misraccheckerreferencenolinks.htm) | This static code analysis tool checks if there is no ellipsis exception handler in a try-catch block. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exceptions | STD-008-CPP | Detect errors when converting a string to a number. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below does not check whether the value of age is an integer or within range after the cin function accepts the input from a user. This can result in an unexpected value for age or age may be uninitialized. Additionally, other inputs may fail that follow. |
| #include <iostream>    void getAge() {  int age;  std::cin >> age;  } |

| **Compliant Code** |
| --- |
| The compliant code example below enables exceptions to be used so that a conversion failure for the value of age variable results in an exception being thrown which can be caught and handled gracefully. It sets the badbit and failbit flags to treat conversion errors and loss of integrity as exceptions. |
| #include <iostream>    void getAge() {  int age;  std::cin.exceptions(std::istream::failbit | std::istream::badbit);  try{  std::cin >> age;  } catch (std::istream::failure &E) {  /\* handle error \*/  }  } |

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

| **Principles(s):**  *[1] Validate Input* – Input should be checked to ensure that it can be converted into a numeric value before it is used. Any conversion failure should result in an exception being thrown.  *[8] Practice Defense in Depth* – Exception handling coupled with unit testing can provide a defense in depth approach to ensuring that errors are detected when converting a string to a number. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium (2) | Unlikely | Medium | **P4** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | BADFUNC.ATOF  BADFUNC.ATOI  BADFUNC.ATOL  BADFUNC.ATOLL | This static code analysis tool checks for the use of atof, atoi, atol, atoll because they obsolete functions and are the incorrect way to convert. |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | [cert-err34-c](https://clang.llvm.org/extra/clang-tidy/checks/cert/err34-c.html) | This front-end compiler tool flags calls to string-to-number conversion functions that do not verify the validity of the conversion, such as atoi() or scanf(). |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-ERR62-a | This integrated software testing solution ensures that the library functions atof, atoi and atol from library stdlib.h will not be used. |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | [CERT.ERR.CONV.STR\_TO\_NUM](https://help.klocwork.com/2023/en-us/reference/cert.err.conv.str_to_num.htm) | This static code analysis tool detects errors when converting a string to a number. When calling a formatted input stream function like istream::operator>>(), information about conversion errors is queried through the basic\_ios::good(), basic\_ios::bad(), and basic\_ios::fail() inherited member functions or through exception handling if it is enabled on the stream object. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | STD-009-CPP | Do not depend on the order of evaluation for side effects. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below attempts to read the value of i, modify it, and then read the value again without an intervening sequence point. This can result in undefined behavior as the values are passed as the arguments for the parameters of the someFucntion function. |
| extern void someFunction(int i, int j);    void c(int i) {  someFunction(i++, i);  } |

| **Compliant Code** |
| --- |
| The compliant code example below demonstrates two ways to introduce an intervening sequence point so that the values of the arguments passed to some function are equivalent or one value is 1 greater than the other. This maintains the integrity of the code so that no unspecified behavior occurs. |
| extern void someFunction(int i, int j);    void c(int i) {  i++;  someFunction(i, i);  }  /\* or \*/  void c(int i) {  int j = i++;  func(j, i);  } |

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

| **Principles(s):**  *[2] Heed Compiler Warnings* – Some compilers generate warnings such as “This expression has side effects and will not be evaluated” which should be addressed to prevent undefined behavior.  *[4] Keep It Simple* - Unsequenced ordering creates confusion because it is not immediately clear which order expression should be evaluated first. They may be performed in any order and may overlap. Proper sequencing expressions with distinct sequence points simplifies code and improves readability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium (3) | Probable | Medium | **P8** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | [-Wunsequenced](https://clang.llvm.org/docs/DiagnosticsReference.html#wunsequenced) | This front-end compiler tool can detect simple violations where path-sensitive analysis is not required and is enabled by default. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.STRUCT.SE.DEC  LANG.STRUCT.SE.INC | This static code analysis tool checks for side effects in expressions with decrement and side effects in expressions with increment. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-EXP50-a  CERT\_CPP-EXP50-b  CERT\_CPP-EXP50-c  CERT\_CPP-EXP50-d  CERT\_CPP-EXP50-e  CERT\_CPP-EXP50-f | This integrated software testing solution checks that the value of an expression is the same under any order of evaluation that the standard permits and that, between sequence points, an object’s stored value is modified at most once by the evaluation of an expression. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: EXP50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp50cpp.html) | This static code analysis tool checks for situations where expression value depends on order of evaluation. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | STD-010-CPP | Ensure that operations on signed integers do not result in overflow |

| **Noncompliant Code** |
| --- |
| The noncompliant code example below can result in a signed integer overflow during the addition of the signed operands si\_a and si\_b when the resulting value cannot be represented in the number of bits allocated to the integer’s representation. Buffer overflows occur when the value of sum is written outside of the boundaries of the memory allocated to a signed integer. |
| void addNumbers(signed int si\_a, signed int si\_b) {  signed int sum = si\_a + si\_b;  std::cout << "The sum of " << si\_a << " and " << si\_b << " is "  << sum << std::endl;  } |

| **Compliant Code** |
| --- |
| The compliant code example below ensures that the result of addition operation cannot overflow by using a precondition test that checks if the maximum and minimum signed integer limits will be passed before the sum is calculated. If the result exceeds the limits, the code within the if-statement will execute and can handle the error gracefully. |
| void addNumbers(signed int si\_a, signed int si\_b) {  signed int sum;  if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||  ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {  / \* handle error \*/  } else {  sum = si\_a + si\_b;  std::cout << "The sum of " << si\_a << " and " << si\_b << "  is " << sum << std::endl;  }  } |

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

| **Principles(s):**  *[1] Validate Input* – User input that is used for operations on signed integers should always be checked to ensure that the result will no cause an overflow.  *[7] Sanitize Data Sent to Other Systems* – Values sent to other systems should be sanitized if they will be used for signed integer operations so they do not introduce an overflow vulnerability.  *[8] Practice Defense in Depth* – Multiple layers from a defense in depth approach should be used to detect and respond to overflows caused by signed integer operations. Conditional if-else statements that throw an exception if the result of an operation will exceed the value range and assertions can be combined to prevent overflows and prove that overflows cannot occur, making the code especially secure from the vulnerability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High (4) | Likely | High | **P9** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule INT32-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint32c.html) | This static code analysis tool checks for integer overflow, tainted division operand, and tainted modulo operand. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | [ALLOC.SIZE.ADDOFLOW](http://164.70.20.97:7340/install/codesonar/doc/html/WarningClasses/ALLOC/ALLOC.SIZE.ADDOFLOW.html)  ALLOC.SIZE.IOFLOW  [ALLOC.SIZE.MULOFLOW](http://164.70.20.97:7340/install/codesonar/doc/html/WarningClasses/MISC/MISC.MEM.SIZE.MULOFLOW.html)  ALLOC.SIZE.SUBUFLOW  [MISC.MEM.SIZE.ADDOFLOW](http://164.70.20.97:7340/install/codesonar/doc/html/WarningClasses/MISC/MISC.MEM.SIZE.ADDOFLOW.html)  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | This static code analysis tool checks for addition overflow of allocation size, integer overflow of allocation size, multiplication overflow of allocation size, subtraction underflow of allocation size, addition overflow of size, unreasonable size arguments, multiplication overflow of size, and subtraction underflow of size. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.25 | [V1026](https://pvs-studio.com/en/docs/warnings/v1026/), [V1070](https://pvs-studio.com/en/docs/warnings/v1070/), [V1081](https://pvs-studio.com/en/docs/warnings/v1081/), [V1083](https://pvs-studio.com/en/docs/warnings/v1083/), [V1085](https://pvs-studio.com/en/docs/warnings/v1085/), [V5010](https://pvs-studio.com/en/docs/warnings/v5010/) | This proprietary static code analyzer emits alarms for signed arithmetic computations where the possibility of an overflow exists. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | This integrated software testing solution helps avoid integer overflows by checking for integer overflow or underflow in constant expression that use '+', '-', '\*', and '<<' operators. |

### 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



The shift from the well-established DevOps process and infrastructure to DevSecOps will integrate security earlier in the software development life cycle (SDLC) and utilize automation to facilitate enforcement of and compliance to the standards defined in this policy. It also helps quality assurance and streamline product development because vulnerabilities can be identified and mitigated earlier in development. The existing DevOps process can be modified beginning in the early, pre-production stages of development by assessing the threat landscape applicable to a project and identifying initial strategies to mitigate potential vulnerabilities. The next step is to design test cases using best practices and the security policy as a guide from which code can then be designed around. The build phase can be modified to incorporate automated security analyses such as unit tests and static analysis tools that check that code additions and updates comply with standards established by the security policy. IDEs should be modified to use appropriate security plug-ins that check for security flaws in code. As code is being developed and changes are made, security tests can be automatically triggered to scan for and identify vulnerabilities. This can be performed through a combination of Static Application Security Testing, Dynamic Application Security Testing, and Interactive Application Security Testing tools. This also serves to verify security policy compliance. Before code can be sent to production it must be thoroughly tested for proper functionality and vulnerabilities. Penetration and integration testing can be automated to evaluate a system's security architecture against the security policy and check that combined parts of an application are able to work together without error or introducing vulnerabilities. During the production stages of development, the existing DevOps process can be modified by reviewing differences in configurations between the production environment and development environment before deployment. Once code is deployed, automated event alerts should be used to inform developers of errors and questionable interactions. Automated logging should be performed to maintain a history of interactions so that mitigation can be expedited when issues arise. During the maintenance/observe phase, continuous assessment of security should be executed using automated security checks and security monitoring loops to detect attacks and leaks. As the process transitions into the planning phase of the next iteration, responses to new threats should be prioritized to address live application flaws and vulnerabilities.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low (2) | Unlikely | Low | **P3** | **L3** |
| STD-002-CPP | High (4) | Likely | High | **P9** | **L2** |
| STD-003-CPP | High (5) | Likely | Medium | **P18** | **L1** |
| STD-004-CPP | High (5) | Likely | Medium | **P18** | **L1** |
| STD-005-CPP | High (5) | Likely | Medium | **P18** | **L1** |
| STD-006-CPP | Low (1) | Unlikely | High | **P1** | **L3** |
| STD-007-CPP | Low (2) | Probable | Medium | **P4** | **L3** |
| STD-008-CPP | Medium (2) | Unlikely | Medium | **P4** | **L3** |
| STD-009-CPP | Medium (3) | Probable | Medium | **P8** | **L2** |
| STD-010-CPP | High (4) | Likely | High | **P9** | **L2** |

### 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 is a security practice that provides protection for stored data by transforming it from plaintext to an encoded, unreadable form. Data that is stored in a database must be encrypted at all times to protect it from potential threats such as hackers. Even if a hacker obtains the data, it will be unusable because of its encrypted state. Sensitive data is especially vulnerable and must always be stored in an encrypted state. Encryption of all data at rest should be performed by default using the Advanced Encryption Standard (AES) algorithm, AES-256 and before it is written to a database storage system or disk. |
| Encryption in flight | Encryption in flight is a security practice that provides protection for data that is moving between devices or networks by encrypting it before it is sent and decrypting it after it has reached its destination. It protects data in case it is intercepted by an unauthorized entity. Authentication and authorization must be performed to ensure that a user is who they say they are and that they have permission to access and view the data they request. After authentication and authorization have been performed, Transport Layer Security (TLS) should be used to encrypt data before it is sent over the internet and to verify that data has not been tampered with upon its arrival. Data can then be decrypted client-side using a shared key. |
| Encryption in use | Encryption in use is a security policy that protects data while it is accessed or consumed to run analytics or computations. Data in use is the most vulnerable to threats because it is traditionally in a plaintext state. It needs to be protected from unauthorized entities that use tactics like phishing attacks to obtain authorized credentials. Homomorphic encryption allows basic operations to be performed on encrypted data without first having to decrypt it. Homomorphic encryption should be used whenever mathematical computations need to be performed on data and decrypted access is not necessary to complete a task. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a form of access control that verifies that supplied user credentials match stored user credentials to prove that a user is who they claim they are and prevent unwanted access to a system. Users gain access to a system only after they have been authenticated. Authentication must be implemented in order to access any system. Additionally, multifactor authentication should be used to gain access to sensitive systems or data. User login credentials must be established for applicable users, stored in an encrypted state, conform to a secure standard\*, and be updated every quarter. User logins will be implemented to authenticate a user before they gain access to any system or resource.  Secure standard: 8 or more characters that include an upper case letter, lower case letter, number, and special symbol. |
| Authorization | Authorization is another form of access control that determines what resources an authenticated user has access to. It helps to ensure that users have permission to access resources and can be combined with the principle of least privilege to provide an extra layer of security. Levels of access and permissions are established upon user account creation and only allow the necessary actions required to perform what a user needs to perform. A users level of access dictates what they can do after they are authenticated. Actions such as making changes to the database and adding new users must have high-level permissions before they can be performed. |
| Accounting | Accounting is the process of monitoring usage information between a user and a system. It involves measuring resources consumed, logging session statistics and user information, and tracking data sent and received. Changes to the database and files accessed by users must be logged for authorization control, resource utilization evaluations, and audits. Accounting allows administrators to observe any actions performed, who performed them, and when. It facilitates mitigation when a problem occurs or something has been compromised by enabling tracing which speeds up the recovery time. Accounting must be implemented at all times and the data collected from it must only be accessible to those with authorization. |

**\***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 | P.K. |
| 2.0 | 05/21/2023 | 10 security principles  10 coding standards tables | Paul Kenaga | O.M. |
| 3.0 | 06/08/2023 | Risk Assessment  Automated Detection  Automation  Summary of Risk Assessments  Encryption Policies  Map Principles | Paul Kenaga |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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