**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 | Ensure input data from user is properly formed and correct as early as possible. Any malicious or untrusted input must be scrutinized and deleted. |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available to the compiler and eliminate all warnings produced by the compiler. |
| 1. Architect and Design for Security Policies | Create and design software that implements and enforces security policies. If a system requires multiple permissions, create multiple subsystems and a communication system with appropriate privileges. |
| 1. Keep It Simple | Keep the design as simple and small as possible. Complex designs increase the chance that errors will be made. |
| 1. Default Deny | Base access decisions on permission over exclusion. This means that access will always denied unless explicitly granted by the user. |
| 1. Adhere to the Principle of Least Privilege | Every process should be done with the least amount of privileges possible. Any more permission needed should be temporary. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data sent to other systems such as command shells, databases, or other commercial software. |
| 1. Practice Defense in Depth | Mitigate potential risks ahead of time by enacting multiple layers of defense. In this case, if one layer ends up insufficient, the other layers have the ability to stop any potential danger. |
| 1. Use Effective Quality Assurance Techniques | Different types of testing and regular audits on source code should be included as part of a quality assurance program. |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for target development language and platform. |

### 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.

#### Data Type Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CLG] | Declare all variables and functions before using them. Do not allow implicit declarations. |

| **Noncompliant Code** |
| --- |
| Code does not declare variable type. |
| foo = 0; |

| **Compliant Code** |
| --- |
| Code specifies variable type. |
| int foo = 0; |

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

| **Principles(s):**  #1 Input must match defined data types.  #2 Correct all compiler warnings related to any data type declarations.  #4 Do not use implicit declaration – always explicitly declare variables.  #8 Variable, function, and return declarations add a layer of defense |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ECLAIR | 1.2 | CC2.DCL31 | Fully implemented |
| Astree | 22.04 | Type-specifier  Function-return-type  Implicit function declaration  Undeclared parameter | Fully implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-DCL31-a | All functions must be declared before use |
| Polyspace Bug finder | R2022b | CERT C: Rule DCL31-C | Checks for types not explicitly specified and implicit function declaration |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CLG] | Do not read uninitialized variables. |

| **Noncompliant Code** |
| --- |
| Uninitialized variable is included in a print statement which will result in undesired behavior. |
| #include <iostream>    void f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| Variable “i” is initialized to an integer value. When passed to print statement, behavior is as expected. |
| #include <iostream>    void f() {  **int** i = 25;    std::cout << i;  } |

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

| **Principles(s):**  #1 Input shall be within ranges defined by initialization  #2 Correct compiler warnings related to uninitialized variables  #8 Initializing all variables adds a layer of defense |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Uninitialized-local-read  Uninitialized-variable-use | Fully checked |
| CodeSonar | 7.1p0 | LANG.MEM.UVAR | Uninitialized variable |
| Cppcheck | 1.66 | Uninitvar  Uninitdata  Uninitstring  uninitMemberVar  uninitStructMember | Detects uninitialized variables, pointers, struct members, and array elements. |
| Parasoft C/C++Test | 2022.1 | CERT\_C-EXP33-a | Avoid use before initialization |

#### Coding Standard 3

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

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, it can lead to a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    **void** f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):**  #1 Storage for an input string must allow for a null terminator  #7 String lengths bounded by null terminator prevents SQL injections and buffer overruns  #8 Verifying string length adds another layer of defense |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoids accessing arrays out of bounds, overflow when writing to a buffer. Prevents buffer overflows from tainted data. Avoid buffer write overflow from tainted data. Avoid using unsafe string functions which may cause buffer overflows. |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CLG] | Always use prepared statements for querying. |

| **Noncompliant Code** |
| --- |
| Without the use of prepared statements, a user may alter the SQL statement directly. |
| uName = getRequestString("username");  uPass = getRequestString("password");  sql = “SELECT \* FROM Users WHERE Name = " + uName + " AND Pass = " +  uPass + ” |

| **Compliant Code** |
| --- |
| Using prepared statements, the user can not alter any SQL statements, limiting the potential for injection. |
| PreparedStatement pStatement = PreparedStatement();  std::cin >> username;  std::cin >> password;  sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username,  password}; |

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

| **Principles(s):**  #1 Validating input prevents SQL injection  #3 Use prepared statements to prevent SQL injections  #7 Using prepared statements allows data to be sanitized  #8 Use of prepared statements is another layer of defense against SQL injection |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format String |
| Parasoft C/C++ test | 2022.1 | CERT\_C-FIO30-a  CERT\_C-FIO30-a  CERT\_C-FIO30-a | Avoid calling functions printf/wprintf with only one argument other than string constant  Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable  Never use unfiltered data from an untrusted user as the format parameter |
| Polyspace Bug Finder | R2022b | CERT C: Rule FIO30-C | Checks for tainted string format |
| PC—lint Plus | 1.4 | 592 | Partially supported: reports non-literal format strings |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not allow access to free memory. |

| **Noncompliant Code** |
| --- |
| In this code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this code below, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):**  #2 Use static tools to detect freed memory issues  #9 Use standard and effective quality assurance to identify freed memory |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Dangling\_pointer\_use | Supported  Astree reports all accesses to freed allocated memory |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses after its deallocation and double memory deallocations |
| Coverity | 2017.07 | USE\_\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Polyspace Bug Finder | R2022 | CERT C: Rule MEM30-C | Checks for accessing previously freed pointer and freeing previously freed pointer. Rule is partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to test assumptions such as if a pointer is NULL or not. If the expression evaluates to false, the abort() function is called preventing unexpected behavior. |

| **Noncompliant Code** |
| --- |
| The pointer here may be NULL which can result in exploitation of the program. |
| #include <stdio.h> /\* printf \*/  #include <assert.h> /\* assert \*/  void print\_number(int\* myInt) {  printf ("%d\n",\*myInt);  }  int main ()  {  int a = 10;  int \* b = NULL;  int \* c = NULL;  b=&a;  print\_number (b);  print\_number (c);  return 0;  } |

| **Compliant Code** |
| --- |
| The assert() function is used added to prevent printing from dereferenced memory. |
| #include <stdio.h> /\* printf \*/  #include <assert.h> /\* assert \*/  void print\_number(int\* myInt) {  assert (myInt!=NULL);  printf ("%d\n",\*myInt); |

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

| **Principles(s):**  #8 By using assertions, problems can be solved early on rather when found as bugs after being deployed  #9 Use assertions to test code throughout development and incorporate them into every day writing of code |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasof C/C++ test | 2022.1 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions. Do not leave ‘catch’ blocks empty. |
| LDRA tool suite | 9.7.1 | 527 S, 56 D1 71 D | Partially implemented |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| In this code, the constructor for S may throw an exception that is not caught when globalS is constructed during program startup. |
| struct S {  S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| This code makes globalS into a local variable, allowing any exceptions thrown to be caught because the constructor for S will be executed the first time function globalS() is called. |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error.  }  // Unreachable.  } |

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

| **Principles(s):**  #3 Write code using try/catch/throw to prevent code crashing  #9 Always test code and throw exceptions to prevent unexpected behavior |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| CodeSoner | 7.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP\_ERR51-a  CERT\_CPP\_ERR51-b | Always catch exceptions. Each exception explicitly thrown I the code shall have a handler of a compatible type in all call paths that could lead to that pointer. |
| Polyspace Bug Finder | R2022b | CERT C++: ERR51-CPP | Checks for unhandled exceptions(rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-008-CPP] | Use valid iterator ranges. |

| **Noncompliant Code** |
| --- |
| The incorrect values are passed to the function. If the two iterators are not equal, c.end will continue to advance resulting in unwanted behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the iterator values passed to std::for\_each() are passed in the proper order. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):**  #3 Do not allow iterators to form overflow errors  #9 Use effective QA and unit testing to detect for any incorrect values, ranges, and outcomes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Overflow\_upon\_dereference |  |
| CodeSonar | 7.1p0 | LANG.MEM.BO | Buffer overrun |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn’t really a range. Do not compare iterators from different containers. |
| Polyspace Bug finder | R2022b | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | [STD-009-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| In this code, open() is called, but close() never is. The default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. This results in a file not properly being closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this code, close() is called before terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):**  #1 Validate the file before opening, to make sure what is in the file will not be malicious  #9 Always use unit testing to assure proper tasks are complete, like opening and closing files successfully. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2022b | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |
|  |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Arrays | [STD-010-CLG] | Ensure size arguments for variable length arrays are in a valid range |

| **Noncompliant Code** |
| --- |
| In this code, a variable length array of size “size” is declared. |
| #include <stddef.h>    extern void do\_work(int \*array, size\_t size);    void func(size\_t size) {  int vla[size];  do\_work(vla, size);  } |

| **Compliant Code** |
| --- |
| This code ensures the size argument used to allocate the variable length array is in a valid range (between 1 and a programmer defined max). Otherwise, an algorithm is used that depends on dynamic memory allocation. |
| #include <stdint.h>  #include <stdlib.h>    enum { MAX\_ARRAY = 1024 };  extern void do\_work(int \*array, size\_t size);    void func(size\_t size) {  if (0 == size || SIZE\_MAX / sizeof(int) < size) {  /\* Handle error \*/  return;  }  if (size < MAX\_ARRAY) {  int vla[size];  do\_work(vla, size);  } else {  int \*array = (int \*)malloc(size \* sizeof(int));  if (array == NULL) {  /\* Handle error \*/  }  do\_work(array, size);  free(array);  }  } |

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

| **Principles(s):**  #3 Write code with correct argument sizes when dealing with arrays and functions  #9 Use QA and unit testing to make sure output is expected and correct |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  MISC.MEM.SIZE.BAD | Integer overflow of allocation size  Multiplication overflow of allocation size  Unreasonable size argument |
| Coverity | 2017.07 | REVERSE\_NEGATIVE | Fully implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-ARR32-a | Ensure the size of the variable length array is in valid range |
| Polyspace Bug Finder | R2022b | CERT C: Rule ARR32-C | Checks for memory allocation with tainted size and tainted size of variable length array. Rule fully covered. |

### 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.

DevOps becomes DevSecOps when security is included within each step of the software development life cycle. During the pre-production phases, things like threat modeling and security training on tools available to developers should be included under Asses and Plan. Under Design and Build phases, the security of the specific IDE can be dealt with. Next, in Verify and Test, it would include items like static code analysis and unit tests. Once the software is in production, automated testing should continue with defense-in-depth in mind. Items like network monitoring and performance logs can detect any threats to the software.

### 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-CLG | Low | Unlikely | Low | 3 | 3 |
| STD-002-CPP | High | Probable | Medium | 12 | 1 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CLG | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | High | Likely | High | 9 | 2 |
| STD-007-CPP | Low | Probable | Medium | 4 | 3 |
| STD-008-CPP | High | Probable | High | 6 | 2 |
| STD-009-CPP | Medium | Unlikely | Medium | 4 | 3 |
| STD-010-CLG | High | Probable | High | 9 | 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 | Data will be protected when it is stored, whether it be on a computer, phone, cloud, or database. There are many different tools available to help keep data encrypted such as AxCrypt to name one. This is to protect data from either being physically or virtually stolen. |
| Encryption at flight | Protect data while it is moving from one place to another, like when sending an email or web browsing. There are different tools available for email encryption, like S/MIME for example. Web browsing traffic should only be sent over secure sockets layer and by obtaining a SSL/TSL HTTPS certificate from an authorized issuer. |
| Encryption in use | Data shall be protected while it is being created, accessed, edited, or viewed. This is between both at-flight and in-rest. This is generally when accessing a website on a server or if the CPU is busy processing applications. Encryption in use is important because even memory can be insecure and hacked. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication answers the question “Who are you?”. Authentication will verify a user and their respective credentials and can be done in multiple ways. Username/password verification, multi-factor authentication steps or even biometrics. All users shall be verified. If a user is not authenticated, the potential for identity theft and/or unauthorized access is increased. |
| Authorization | Authorization answers the question “What can I do?”. Authorization is set for every user and defines their level of access to different files, directories, or even applications. Each user shall have the ability to read, write, and or execute on data depending on the user’s role. For example, an administrator may have the ability to revoke some privileges or block certain apps for other workers. |
| Accounting | Accounting answers the questions “What exactly happened and when did it happen?”. Accounting means record keeping, like log files, which can include the details of items like logins, new users, file access events, or database events. All actions should be timestamped so there is a record of when specific events happened. Accounting is done all the time to know who is doing what. |

**\***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 | 11/03/2022 | Updates Coding Standards | Matthew Trembley |  |
| 3.0 | 12/04/2022 | Updated automation and risk assessment | Matthew Trembley | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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